第 3 章

CHAPTER 3

夯实 FFmpeg 基础：重要数据结构及 API

本章主要介绍 FFmpeg 编程中用到的基础知识，包含常见音视频概念、常用 API 函数、常用结构体、解封装流程、解复用器流程及注册等。FFmpeg 是编解码的利器，功能强大，使用起来比较方便，但 FFmpeg 更新非常快，有些 API 在新版本中可能被替换掉了，本书以新版本 5.0 为主要依据进行讲解，同时尽量兼顾旧版本。

7min

3.1 FFmpeg 的读者入门案例

网络上关于 FFmpeg 的帖子非常多，但不太适合读者入门，例如直接给出一个解码案例，涉及十几条数据结构和几十个 API。如果读者不了解音视频的概念，往往会有一种雾里看花的感受，虽然也能编译并运行成功，但不理解这些函数和数据结构的应用原理。

3.1.1 初识 FFmpeg 的 API

使用 FFmpeg 可以输出日志，也可以操作目录等，下面通过一个简单的案例来快速了解 FFmpeg 的 API 函数应用。

1. 使用 FFmpeg 输出日志的头文件和 API

使用 FFmpeg 输出日志，涉及的头文件包括< libavutil/log.h >，具体的 API 函数包括 av_log_set_level、av_log 等，伪代码如下：

```
//chapter3/3.1.help.txt
//日志操作的头文件
#include <libavutil/log.h>
//设置 log 打印级别
av_log_set_level(AV_LOG_Debug)
//打印输出 log 日志
av_log(NULL,AV_LOG_INFO,"... % s\n",op)
//常用 log 日志级别如下
AV_LOG_ERROR、AV_LOG_WARNING、AV_LOG_INFO、AV_LOG_Debug
```

2. 使用 FFmpeg 操作目录的头文件和 API

使用 FFmpeg 操作目录，涉及的头文件包括< libavformat/avformat.h >，具体的 API 函数包括 avio_open_dir、avio_read_dir 和 avio_close_dir 等，伪代码如下：

```
//chapter3/3.1.help.txt
//目录操作的头文件
#include <libavformat/avformat.h>
//打开目录
avio_open_dir()
//读取目录中文件的每项信息
avio_read_dir()
//关闭目录
avio_close_dir()

//操作目录的上下文结构
AVIODirContext
//目录项:用于存放文件名、文件大小等信息
AVIODirEntry
```

3. 创建 Qt 工程并使用 FFmpeg 操作目录

1）创建 Qt 工程

打开 Qt Creator，创建一个 Qt Console 工程，具体操作步骤可以参考“1.4 搭建 FFmpeg 的 Qt 开发环境”，工程名称输入 QtFFmpeg5_Chapter3_001，如图 3-1 所示。由于使用的是 FFmpeg 5.0.1 的 64 位开发包，所以编译套件应选择 64 位的 MSVC 或 MinGW。

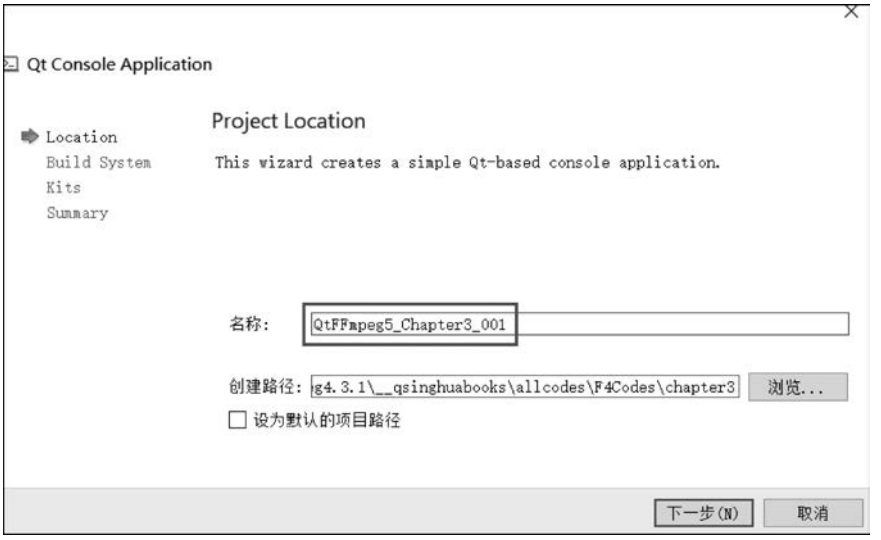

图 3-1　Qt 工程之 FFmpeg 操作目录

2）引用 **FFmpeg** 的头文件和库文件

打开配置文件 QtFFmpeg5_Chapter3_001.pro，添加引用头文件及库文件的代码，如图 3-2 所示。由于笔者的工程目录 QtFFmpeg5_Chapter3_001 在 chapter3 目录下，而 chapter3 目录与 ffmpeg-n5.0-latest-win64-gpl-shared-5.0 目录是平级关系，所以项目配置文件里引用 FFmpeg 开发包目录的代码是 $$PWD/../../ffmpeg-n5.0-latest-win64-gpl-shared-5.0/，$$PWD 代表当前配置文件（QtFFmpeg5_Chapter3_001.pro）所在的目录，../../代表父目录的父目录。

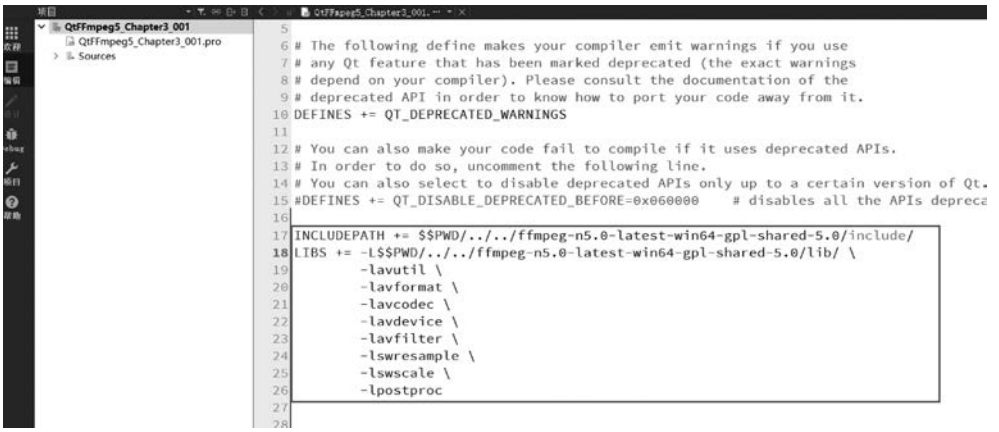

图 3-2　Qt 工程引用 FFmpeg 的头文件和库文件

在.pro 项目配置中，添加头文件和库文件的引用，代码如下：

```
//chapter3/3.1.help.txt
INCLUDEPATH += $$PWD/../../ffmpeg-n5.0-latest-win64-gpl-shared-5.0/include/
LIBS += -L$$PWD/../../ffmpeg-n5.0-latest-win64-gpl-shared-5.0/lib/ \
        -lavutil \
        -lavformat \
        -lavcodec \
        -lavdevice \
        -lavfilter \
        -lswresample \
        -lswscale \
        -lpostproc
```

3）使用 **FFmpeg** 输出不同级别的日志信息

打开 main.cpp 文件，添加日志操作的头文件#include <libavutil/log.h>，由于是 C++ 代码调用 FFmpeg 的纯 C 函数，所以需要使用 extern "C"将头文件括起来。调用 av_log_set_level()函数可以设置 FFmpeg 的日志级别，调用 av_log()函数可以输出日志。av_log()函数在 libavutil/log.h 文件中，定义如下：

```
//chapter3/3.1.help.txt
/**
 * Send the specified message to the log if the level is less than or equal
 * to the current av_log_level. By default, all logging messages are sent to
 * stderr. This behavior can be altered by setting a different logging callback
 * function.
 * 如果日志级别小于或等于当前设置的日志级别,则将具体的日志消息发送到日志输出系统
 * 注意:默认情况下,所有的日志都输出到 stderr,而不是 stdtou
 * @see av_log_set_callback
 *
 * @param avcl A pointer to an arbitrary struct of which the first field is a
 *        pointer to an AVClass struct or NULL if general log.
 * @param level The importance level of the message expressed using a @ref
 *        lavu_log_constants "Logging Constant".
 * @param fmt The format string (printf-compatible) that specifies how
 *        subsequent arguments are converted to output.
 */
void av_log(void *avcl, int level, const char *fmt, ...) av_printf_format(3, 4);
```

注意：FFmpeg的日志输出系统,默认情况下会将所有的信息都输出到stderr,而不是stdout。这一点读者一定要注意,如果在管道流中只关注stdout,则可能看不到任何输出消息。

调用avlog()函数,可以输出日志信息,代码如下：

```
//chapter3/QtFFmpeg5_Chapter3_001/main.cpp
extern "C"{
    //日志操作的头文件
    #include <libavutil/log.h>
}

int main(int argc, char *argv[])
{
    //QCoreApplication a(argc, argv);
    //return a.exec();

    av_log_set_level(AV_LOG_DEBUG);   //设置日志级别
    av_log(NULL, AV_LOG_DEBUG, "Hello FFmpeg!=>%s\n", "debug 信息");
    av_log(NULL, AV_LOG_INFO, "Hello FFmpeg!=>%s\n", "info 信息");
    av_log(NULL, AV_LOG_WARNING, "Hello FFmpeg!=>%s\n", "warning 信息");
    av_log(NULL, AV_LOG_ERROR, "Hello FFmpeg!=>%s\n", "error 信息");

    return 0;
```

在该案例中，先调用 av_log_set_level()函数将当前日志级别设置为 AV_LOG_DEBUG，然后调用 av_log()函数来输出不同级别的信息，分别是 AV_LOG_DEBUG、AV_LOG_INFO、AV_LOG_WARNING 和 AV_LOG_ERROR，控制台上会显示不同的颜色，如图 3-3 所示。

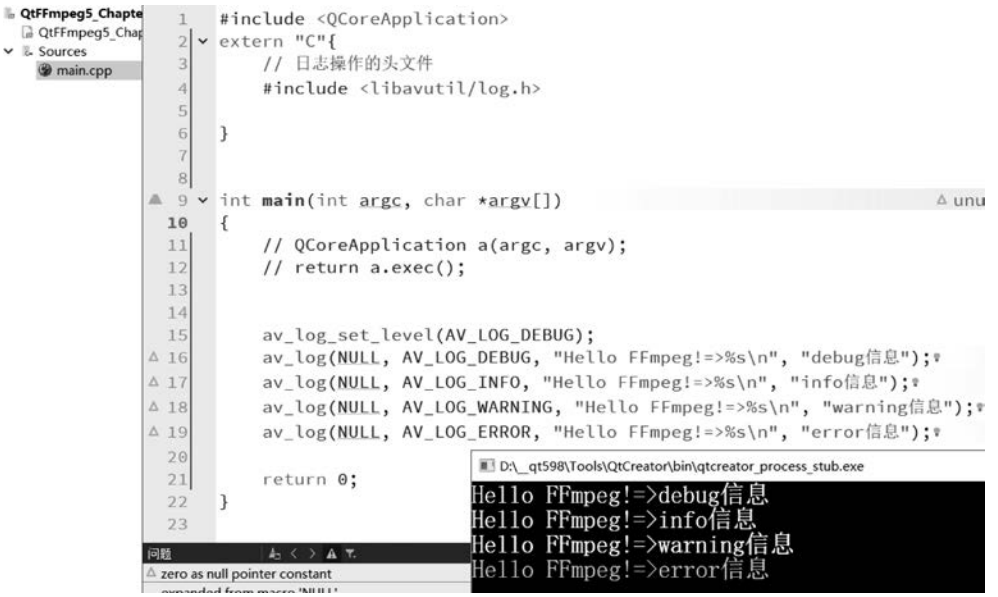

图 3-3　Qt 工程显示 FFmpeg 的不同颜色的日志级别

4）使用 C 语言操作目录

使用 C 语言可以遍历文件夹下的所有文件，dirent.h 文件是用于目录操作的头文件，Linux 系统中默认在/usr/include 目录下（会自动包含其他文件），常见的函数如下：

```
//chapter3/3.1.help.txt
//1. opendir() :打开目录,并返回句柄
//2. readdir() :读取句柄,返回 dirent 结构体
//3. telldir() :返回当前指针的位置,表示第几个元素
//4. close() :关闭句柄

#include <dirent.h>
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);

struct __dirstream
{
    void *__fd;
    char *__data;
```

```
    int __entry_data;
    char * __ptr;
    int __entry_ptr;
    size_t __allocation;
    size_t __size;
    __libc_lock_define (, __lock)
};
typedef struct __dirstream DIR;


struct dirent
{
long d_ino;
off_t d_off;
unsigned short d_reclen;
char d_name [NAME_MAX + 1];
}
```

Linux 系统下遍历目录的方法一般为打开目录→读取→关闭目录，对应的相关函数为 opendir()→readdir()→closedir()。opendir()函数用于打开目录，类似于流的方式，返回一个指向 DIR 结构体的指针，参数 * dirname 是一个字符数组或者字符串常量；readdir()函数用于读取目录，只有一个参数，即 opendir 返回的结构体指针，或者叫作句柄更容易理解。这个函数返回一个结构体指针 dirent * 。

头文件 dirent.h 是 Linux 系统中的一个应用程序接口，主要用于文件系统的目录读取操作，提供了几个目录数据读取函数，但 Windows 平台的 MSVC 编译器并没有提供这个接口，对于跨平台的项目开发来讲就会带来一些麻烦，当在 MSVC 下编译时可能因为 Windows 平台缺少这个接口而导致编译失败，所以需要为 Windows 平台写一些代码实现 dirent.h 文件的功能，开源链接网址为 https://github.com/tronkko/dirent，使用方法有以下两种：

(1) 将解压后的 include/dirent.h 文件复制到 VS 的 include 目录下，如 C:\Program Files (x86)\MicrosoftVisualStudio12.0\VC\include。

(2) 直接将解压后的 include/dirent.h 复制到工程目录下即可。

所以，使用 C 语言可以实现跨平台编译文件夹的功能，代码如下：

```
//chapter3/QtFFmpeg5_Chapter3_001/cdodir.cpp
#include <stdio.h>

//#define __STDC_CONSTANT_MACROS

#if _WIN32
#include <Windows.h>
```

```
#include <io.h>
#include "dirent.h"
#else
#include <unistd.h>
#include <dirent.h>
#endif

//Windows 下载网址为 https://github.com/tronkko/dirent
int main (int argc, char *argv[])
{
    DIR *dir;
    struct dirent* mydirent;

    if((dir=opendir("./"))!=NULL)
    {
        while((mydirent=readdir(dir))!=NULL)
        {
            printf("FileName:%s\n",mydirent->d_name);
        }
    }
    else{
        printf("cannot open %s",argv[1]);
        return -1;
    }
    closedir(dir);

    return 0;
}
```

在 Qt 工程中添加一个文件 cdodir.cpp，添加上述代码，然后运行程序。由于该程序的生成路径为 build-QtFFmpeg5_Chapter3_001-Desktop_Qt_5_9_8_MSVC2015_64bit-Debug，所以遍历当前目录./后显示的是该文件夹下的所有文件，如图 3-4 所示。

图 3-4　Qt 工程之列举当前目录下的文件

注意：读者需要将 main.cpp 文件中的 main()函数修改为 main123 或其他，否则 cdodir.cpp 文件中的 main()函数会提示重定义。

5）使用 **FFmpeg** 操作目录

使用 FFmpeg 也可以操作目录，需要在 Linux 平台下使用这些函数功能（注意目前 Windows 平台下的 FFmpeg 并没有实现此功能）。头文件是 libavformat/avformat.h，相关的函数和结构体的伪代码如下：

```
//chapter3/3.1.help.txt
//打开目录
avio_open_dir()
//读取目录中文件的每项信息
avio_read_dir()
//关闭目录
avio_close_dir()

//操作目录的上下文
AVIODirContext
//目录项:用于存放文件名、文件大小等信息
AVIODirEntry
```

使用 FFmpeg 遍历当前目录的相关代码如下（详见注释信息）：

```
//chapter3/QtFFmpeg5_Chapter3_001/ffmpegdodir.cpp
#include <stdio.h>
#define __STDC_CONSTANT_MACROS

extern "C"{ //C++调用 C 函数
    #include <libavutil/avutil.h>
    #include <libavformat/avformat.h>
}

//注意:该宏的定义要放到 libavformat/avformat.h 文件的下边
char av_error[10240] = { 0 };
#define av_err2str(errnum) av_make_error_string(av_error, AV_ERROR_MAX_STRING_SIZE,
errnum)

int main  (int argc, char *argv[]) {
    AVIODirContext *ctx = NULL; //AVIO:目录操作上下文
    AVIODirEntry *entry = NULL; //AVIO:目录项
    av_log_set_level(AV_LOG_Debug); //设置日志级别

    int ret = avio_open_dir(&ctx, "./", NULL); //打开文件夹
    if (ret < 0) {
```

```
        av_log(NULL, AV_LOG_ERROR, "Cant open dir: %s\n", av_err2str(ret) );
        goto __fail;
    }

    while (1) { //遍历文件夹
        ret = avio_read_dir(ctx, &entry);  //读取文件项:AVIODirEntry
        if (ret < 0) {
            av_log(NULL, AV_LOG_ERROR, "Cant red dir:111\n");
            //跳转到__fail
            goto __fail;
        }
        //如果 entry 是 NULL,则代表是目录最末尾,退出
        if (!entry) break;

        //打印输出文件信息
        av_log(NULL, AV_LOG_INFO,
               "文件名:%s ,文件大小:%" PRId64"\n", //注意:PRId64 这个宏
               entry->name,
               entry->size);

        //释放 entry
        avio_free_directory_entry(&entry);
    }

    __fail:
    avio_close_dir(&ctx);

    return 0;
}
```

在 Qt 的项目中添加一个文件 ffmpegdodir.cpp,添加上述代码,运行程序,如图 3-5 所示。

在 Windows 平台下,运行该程序,会输出红色的错误信息：Can't open dir:Function not implemented。说明 Windows 平台下没有实现该函数,不支持该功能。

打开 Linux 系统,将 ffmpegdodir.cpp 文件从 Windows 系统中复制到 Linux 系统中,编译命令如下：

```
//chapter3/3.1.help.txt
gcc -o ffmpegdodir ffmpegdodir.cpp -I /root/ffmpeg-5.0.1/install5/include/ -L /root/
ffmpeg-5.0.1/install5/lib/  -lavcodec -lavformat -lavutil

./ffmpegdodir
```

生成可执行文件 ffmpegdodir,然后运行该程序,如图 3-6 所示,可以看出 Linux 系统中成功地输出了当前文件夹下的所有文件信息。

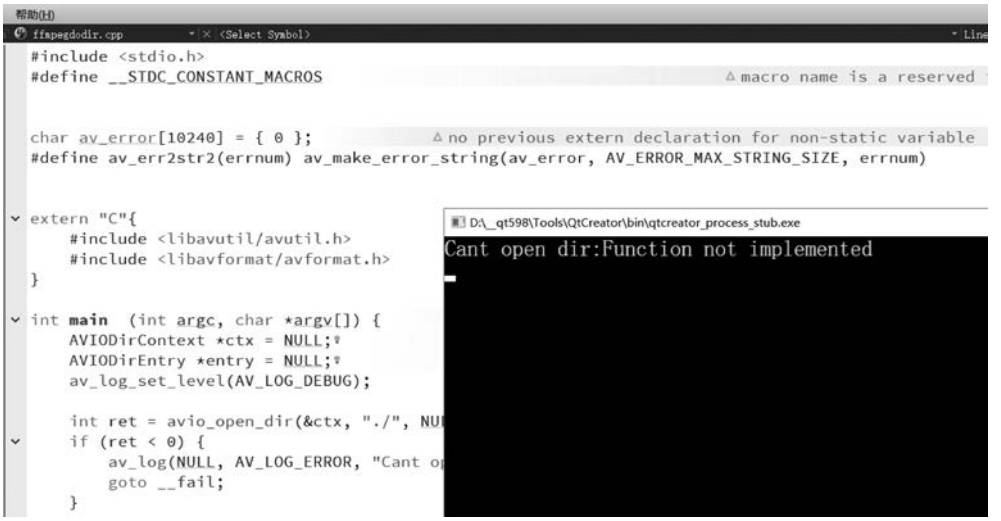

图 3-5 FFmpeg(Windows 平台下)列举当前目录下的文件

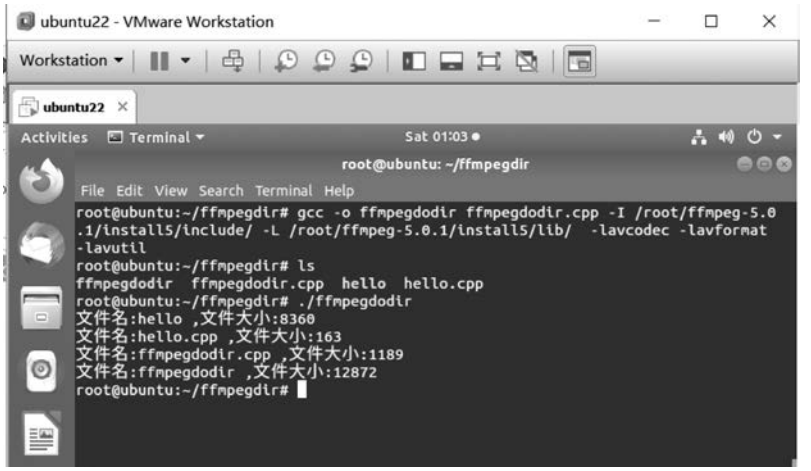

图 3-6 FFmpeg(Linux 平台下)列举当前目录下的文件

在该案例中,代码虽然不多,但知识点很多,如下所示。

(1) Windows 平台下目前没有实现 avio_open_dir、avio_read_dir 等目录操作函数。

(2) av_err2str 编译时出错。在 C++项目中使用 FFmpeg 中的 av_err2str 函数时会报错,这跟 C++与 C 语言的编译方式有关系,av_err2str 被定义为一个 C 语言级别的静态内联函数,有数组空间的定义和开销。C++编译时编译器存在内存方面开销和引用的冲突问题,所以编译不能通过。可以在调用该函数的文件开始加上以下代码,需要添加到 #include < libavformat/avformat.h >之后,相当于重新定义了该宏,如图 3-7 所示,代码如下:

```
//chapter3/3.1.help.txt
char av_error[AV_ERROR_MAX_STRING_SIZE] = { 0 };
#define av_err2str(errnum) av_make_error_string(av_error, AV_ERROR_MAX_STRING_SIZE,
errnum)
```

图 3-7　C++处理 av_err2str 宏

(3) PRID64 编译时出错。这是因为这个宏是定义给 C 语言用的，如果 C++要用它，就要定义一个__STDC_FORMAT_MACROS 宏打开它。C++使用 PRID64，需要分为两步，第 1 步是包含头文件< inttypes.h >，第 2 步是定义宏__STDC_FORMAT_MACROS，可以在编译时加-D__STDC_FORMAT_MACROS 参数，或者在包含文件之前定义这个宏。int64_t 用来表示 64 位整数，在 32 位系统中是 long long int，在 64 位系统中是 long int，所以打印 int64_t 的格式化方法，代码如下：

```
printf("%ld", value); //64 位 OS
printf("%lld", value); //32 位 OS
```

int64_t 也有跨平台的格式化方法，代码如下：

```
//chapter3/3.1.help.txt
#include < inttypes.h >
printf("%" PRId64 "\n", value);
//相当于 64 位的
printf("%" "ld" "\n", value);
//或 32 位的
printf("%" "lld" "\n", value);
```

3.1.2　FFmpeg 的解码及播放流程

使用 FFmpeg 对音视频文件进行解码并播放是非常方便的，有优秀的架构和通俗易懂的 API，并遵循一定的流程。

1. 使用 FFmpeg 进行解码的流程简介

视频文件有许多格式，例如 avi、mkv、rmvb、mov 和 mp4 等，这些被称为容器(Container)，不同的容器格式规定了其中音视频数据(也包括其他数据，例如字幕等)的组织方式。容器中一般会封装视频和音频轨，也称为视频流(stream)和音频流，播放视频文件的第 1 步是根据视频文件的格式，解析(demux)出其中封装的视频流、音频流及字幕流，解析的数据被读到包 (packet)中，每个包里保存的是视频帧(frame)或音频帧，然后分别对视频帧和音频帧调用相应的解码器(decoder)进行解码，例如使用 H.264 编码的视频和 MP3 编码的音频，会相应地调用 H.264 解码器和 MP3 解码器，解码之后得到的就是原始的图像(YUV 或 RGB)和声音(PCM)数据。至此，完成了解码流程，如图 3-8 所示。

注意：图 3-8 显示的是 FFmpeg(2.0)老版本的解码流程相关的 API，新版本与此略有区别，但整体流程和解码框架是一致的。

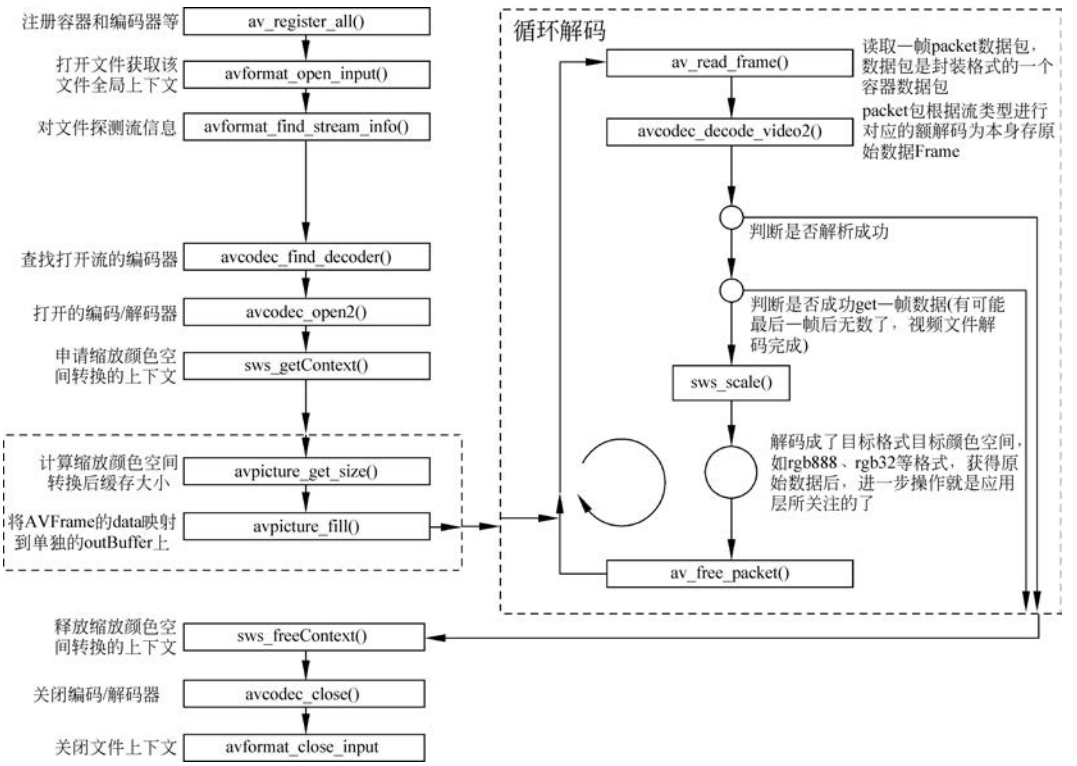

图 3-8　FFmpeg 的解码流程

2. 使用 FFmpeg 进行播放的流程简介

解码完成后，可以根据同步好的时间将图像显示到屏幕上，将声音输出到声卡，这个属

于音视频播放流程中的渲染工作，如图 3-9 所示。FFmpeg 的 API 大体上就是根据这个过程（解协议、解封装、解码、播放）进行设计的，因此使用 FFmpeg 来处理视频文件的方法非常直观简单。

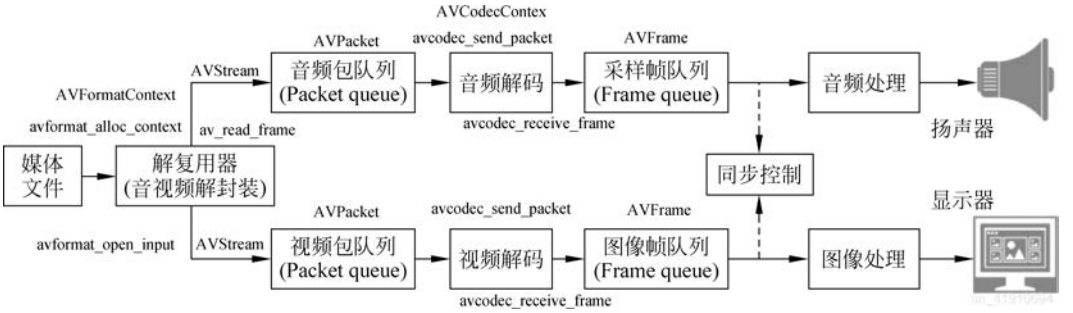

图 3-9　FFmpeg 的播放流程

3. 使用 FFmpeg 的解码流程与步骤分析

对于一位没有音视频基础的初学者来讲，解码音视频文件需要掌握大约十几个非常重要的函数及相关的数据结构，具体步骤如下。

（1）注册：使用 FFmpeg 对应的库，这些库都需要进行注册，可以注册子项也可以注册全部。

（2）打开文件：打开文件，根据文件名信息获取对应的 FFmpeg 全局上下文。

（3）探测流信息：需要先探测流信息，获得流编码的编码格式，如果不探测流信息，则其流编码器获得的编码类型可能为空，后续进行数据转换时就无法知道原始格式，从而导致错误。

（4）查找对应的解码器：依据流的格式查找解码器，软解码还是硬解码是在此处决定的，但是应特别注意是否支持硬件，需要自己查找本地的硬件解码器对应的标识，并查询其是否支持。普遍操作是，枚举支持文件后缀解码的所有解码器进行查找，查找到了就可以硬解码。注意解码时需要查找解码器，而编码时需要查找编码器，两者的函数不同。

（5）打开解码器：打开获取的解码器。

（6）申请缩放数据格式转换结构体：一般情况下解码的数据都是 YUV 格式，但是显示的数据是 RGB 等相关颜色空间的数据，所以此处转换结构体就是进行转换前到转换后的描述，给后续转换函数提供转码依据，是很关键并且常用的结构体。

（7）申请缓存区：申请一个缓存区（outBuffer），填充到目标帧数据的 data 上，例如 RGB 数据，QAVFrame 的 data 上存储的是有指定格式的数据，并且存储有规则，而填充到 outBuffer（自己申请的目标格式一帧缓存区）则是需要的数据格式存储顺序。

（8）进入循环解码：获取一帧（AVPacket），判断数据包的类型进行解码，从而获得存储的编码数据（YUV 或 PCM）。

（9）数据转换：使用转换函数结合转换结构体对编码的数据进行转换，获得需要的目标宽度、高度和指定存储格式的原始数据。

(10) 自行处理：获得的原始数据可以自行处理，例如添加水印、磨皮美颜等，然后不断循环，直到 AVPacket()函数虽然可以成功返回，但是无法得到一帧真实的数据，代表文件解码已经完成。

(11) 释放 QAVPacket：查看源代码，会发现使用 av_read_frame()函数读取数据包时，自动使用 av_new_packet()函数进行了内存分配，所以对于 packet，只需调用一次 av_packet_alloc()函数，解码完后调用 av_free_packet()函数便可释放内存。执行完后，返回执行“(8)进入循环解码：获取一帧”，至此一次循环结束。以此循环，直至退出。

(12) 释放转换结构体：全部解码完成后，安装申请顺序，进行对应资源的释放。

(13) 关闭解码/编码器：关闭之前打开的解码/编码器。

(14) 关闭上下文：关闭文件上下文后，要对之前申请的变量按照申请的顺序依次释放。

注意：FFmpeg 的解码与播放流程包括很多数据结构及 API，涉及很多音视频和流媒体相关的概念。如果读者没有基础，则会比较懵，建议先熟悉整体流程及进行宏观把握。

3.1.3 使用 FFmpeg 解封装并读取流信息的案例

使用 FFmpeg 可以读取音视频文件并解析出流信息，其间会用到几个经典的结构体及 API，主要包括 AVFormatContext、AVStream、AVCodecParameters，以及 avformat_open_input 和 avformat_find_stream_info 等。下面打开一个音视频文件 hello4.mp4，并读取其中的音视频流信息。首先打开 Qt 的项目 QtFFmpeg5_Chapter3_001，添加一个 C++文件 ffmpeganalysestreams.cpp，如图 3-10 所示。

图 3-10 使用 FFmpeg 解析音视频的流信息

ffmpeganalysestreams.cpp 文件的代码如下(详见注释信息)：

```
//chapter3/QtFFmpeg5_Chapter3_001/ffmpeganalysestreams.cpp
#include <stdio.h>
#define __STDC_CONSTANT_MACROS

extern "C"{   //C++调用C函数
    #include <libavutil/avutil.h>
    #include <libavformat/avformat.h>
    #include <libavcodec/avcodec.h>
}

//注意:该宏的定义要放到libavformat/avformat.h文件的下边
static char av_error[10240] = { 0 };
#define av_err2str(errnum) av_make_error_string(av_error, AV_ERROR_MAX_STRING_SIZE,
errnum)

//确保time_base的分母不为0
static double _r2d(AVRational r)
{
    return r.den == 0 ? 0 : (double)r.num / (double)r.den;
}

int main  (int argc, char *argv[]) {

    AVFormatContext *ic = NULL;
    char path[] = "./hello4.mp4";
    //char path[] = "/sdcard/test4.flv";

    //1.打开媒体文件
    int ret = avformat_open_input(&ic, path, 0, 0);
    if (ret != 0) {
        printf("avformat_open_input() called failed: %s\n", av_err2str(ret));
        return -1;
    }
    printf("avformat_open_input() called success.\n");
     //获取媒体总时长(单位为毫秒)及流的数量
    printf("duration is: %lld, nb_stream is: %d\n", ic->duration, ic->nb_streams);

    //2.探测获取流信息
    if (avformat_find_stream_info(ic, 0) >= 0 ) {
        printf("duration is: %lld, nb_stream is: %d\n", ic->duration, ic->nb_streams);
    }

    /**帧率*/
    int fps = 0;
    int videoStream = 0;
    int audioStream = 1;

    for (int i = 0; i < ic->nb_streams; i++) {
        AVStream *as = ic->streams[i];
```

```
        //3.1  查找视频流
        //新版的 API 可以使用:av_find_best_stream(ic, AVMEDIA_TYPE_VIDEO, -1, -1, NULL, 0);
        if (as->codecpar->codec_type == AVMEDIA_TYPE_VIDEO) {
            //videoStream = av_find_best_stream(ic, AVMEDIA_TYPE_VIDEO, -1, -1, NULL, 0);
            printf("video stream.....\n");
            videoStream = i;
            fps = (int)_r2d(as->avg_frame_rate);
            printf("fps = %d, width = %d, height = %d, codecid = %d, format = %d\n",
                 fps,
                 as->codecpar->width,          //视频宽度
                 as->codecpar->height,         //视频高度
                 as->codecpar->codec_id,       //视频的编解码 ID
                 as->codecpar->format);        //视频像素格式:AVPixelFormat
        }
        //3.2 查找音频流
        //audioStream = av_find_best_stream(ic, AVMEDIA_TYPE_AUDIO, -1, -1, NULL, 0);
        else if (as->codecpar->codec_type == AVMEDIA_TYPE_AUDIO) {
            printf("audio stream.....\n");
            audioStream = i;
            printf("sample_rate = %d, channels = %d, sample_format = %d\n",
                 as->codecpar->sample_rate,    //音频采样率
                 as->codecpar->channels,       //音频声道数
                 as->codecpar->format          //音频采样格式:AVSampleFormat
            );
        }
    }

    //4.关闭并释放资源
    avformat_close_input(&ic);


    return 0;
}
```

由于使用的是 Qt Creator,所以代码中的 char path[]=". /hello4. mp4"表示的是当前目录下的 hello4. mp4 文件,即项目的编译输出路径,如图 3-11 所示。

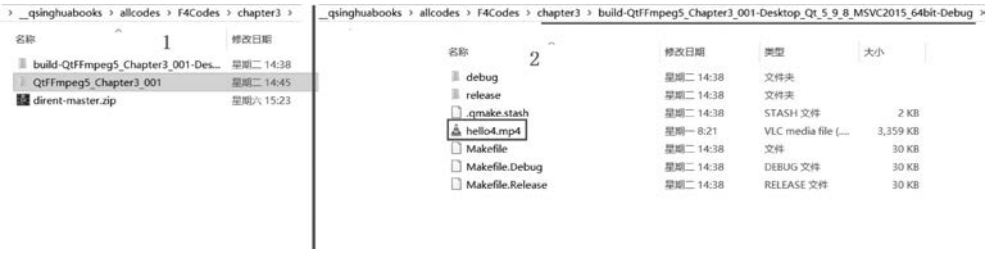

图 3-11　Qt Creator 项目中的./当前路径

可以看到，QtFFmpeg5_Chapter3_001 和 build-QtFFmpeg5_Chapter3_001-Desktop_Qt_5_9_8_MSVC2015_64bit-Debug(项目编译输出路径，在该目录中有 hello4.mp4 文件)是平级关系。

该案例的主要步骤如下：

(1) 打开媒体文件，使用 avformat_open_input()函数，并存储到 AVFormatContext 结构体变量 ic 中。

(2) 探测，以便获取流信息，使用 avformat_find_stream_info()函数，可以获取更多的音视频流信息，同样存储到 AVFormatContext 结构体变量 ic 中。

(3) 查找视频流/音频流，遍历 ic 变量中的流(AVStream)，通过判断 as->codecpar->codec_type 类型来判断是音频(AVMEDIA_TYPE_AUDIO)还是视频(AVMEDIA_TYPE_VIDEO)，然后可以读取详细的参数，包括视频的宽、高、帧率、像素格式，以及音频的采样率、采样格式和声道数等。

(4) 关闭并释放资源，调用 avformat_close_input()函数关闭流并释放资源。

3.2　FFmpeg 的经典数据结构

使用 FFmpeg 对音视频文件(或网络流)进行解码、播放，需要遵循一定的步骤，其间会用到很多数据结构和 API 函数。通常来讲，会按照协议层、封装/解封装层、编解码层这 3 个层次逐步展开。大概需要至少 4 条线程，包括读取源文件的线程(read_thread)、视频解码的线程(video_dec_thread)、音频解码的线程(audio_dec_thread)和渲染线程(render_thread)。FFmpeg 解码及播放的整体框架及流程如图 3-12 所示。

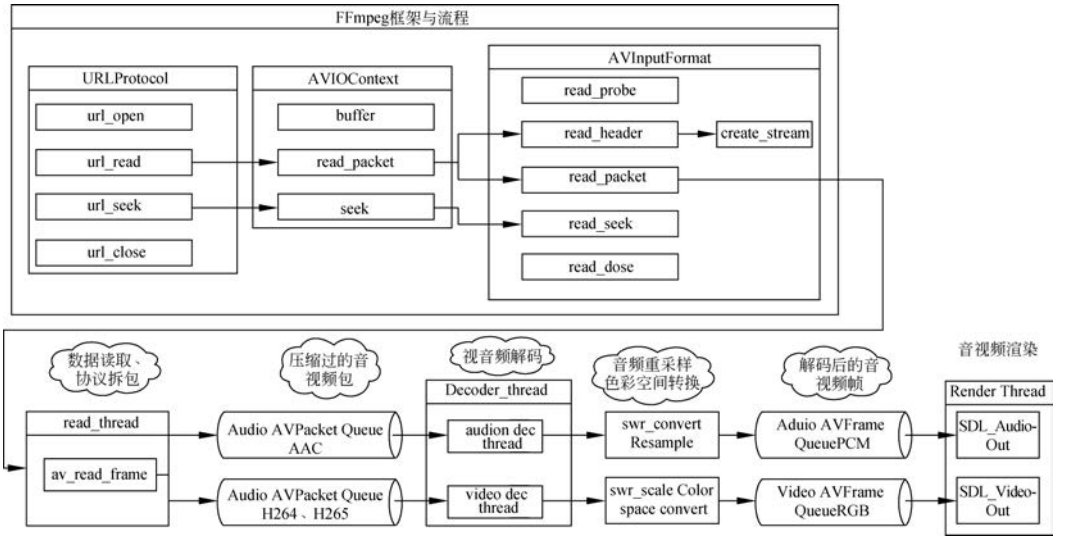

图 3-12　FFmpeg 的整体框架及流程

3.2.1 使用 FFmpeg 进行解码的 10 个经典结构体

使用 FFmpeg 进行解码几乎是必不可少的操作，这里列举了 10 个最基础的结构体（其实远远不止这 10 个），如图 3-13 所示。

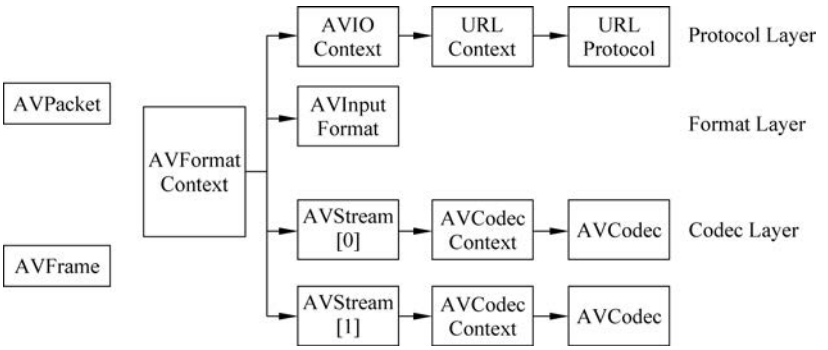

图 3-13 FFmpeg 解码音视频所涉及的 10 大经典数据结构

这几个结构体的解释如下：

（1）AVFormatContext 是贯穿全局的数据结构。

（2）协议层包括 AVIOContext、URLContext 和 URLProtocol 共 3 种结构。

（3）每个 AVStream 存储一个视频/音频流的相关数据。

（4）每个 AVStream 对应一个 AVCodecContext，存储该视频/音频流所使用的解码方式的相关数据。

（5）每个 AVCodecContext 中对应一个 AVCodec，包含该视频/音频对应的解码器。

（6）每种解码器都对应一个 AVCodec 结构。

（7）数据存储包括 AVPacket 和 AVFrame 两种结构。

这 10 个最基础的结构体可以分成以下几类。

1. 解协议

在 FFmpeg 中常见的协议包括 FILE、HTTP、RTSP、RTMP、MMS、HLS、TCP 和 UDP 等。AVIOContext、URLContext 和 URLProtocol 这 3 个结构体主要用于存储音视频使用的协议类型及状态。URLProtocol 用于存储音视频使用的封装格式。每种协议都对应一个 URLProtocol 结构。

注意：FFmpeg 中文件也被当作一种协议：FILE 协议。

---

2. 解封装

在 FFmpeg 中常见的封装格式包括 FLV、AVI、RMVB、MP4、MKV、TS 和 MOV 等。AVFormatContext 结构体主要用于存储音视频封装格式中包含的信息，是统领全局的最基

本的结构体；AVInputFormat 用于存储输入的音视频使用的封装格式(输出格式对应的结构体是 AVOutputFormat)。每种音视频封装格式都对应一个 AVInputFormat 结构。

AVFormatContext 结构体按名字来讲应该将其归为封装层，但是，从整体的架构上来讲，它是 FFmpeg 中提纲挈领的最外层结构体，在音视频处理过程中，该结构体保存着所有信息。这些信息一部分由 AVFormatContext 的直接成员持有，另一部分由其他数据结构所持有，而这些结构体都是 AVFormatContext 的直接成员或者间接成员。总体来讲，AVFormatContext 结构体的作用有点类似于“管家婆”的角色。FFMPEG 是用 C 语言实现的，AVFormatContext 持有数据，方法与其是分开的。

3. 解码

在 FFmpeg 中常见的编解码格式包括 H.264、H.265、MPEG2、MP3 和 AAC 等。AVStream 结构体用于存储一个视频/音频流的相关数据；每个 AVStream 对应一个 AVCodecContext，用于存储该视频/音频流使用解码方式的相关数据；每个 AVCodecContext 中对应一个 AVCodec，包含该视频/音频对应的解码器。每种解码器都对应一个 AVCodec 结构。

4. 存数据

在 FFmpeg 中常见的存储数据的结构体包括 AVPacket 和 AVFrame，其中 AVPacket 是解封装后保存的压缩数据包，AVFrame 是解码后保存的原始音视频帧(PCM 或 YUV)。每个结构体存储的视频一般是一帧，而音频有可能是几帧。

3.2.2 AVPacket 与 AVFrame

1. AVPacket 结构体

AVPacket 结构体在旧版本中放在 avcodec.h 头文件中，在 FFmpeg 4.4 以后放在单独的 packet.h 头文件中。官方对 AVPacket 的说明如下：

```
//chapter3/3.2.help.txt
/**
 * This structure stores compressed data. It is typically exported by demuxers
 * and then passed as input to decoders, or received as output from encoders and
 * then passed to muxers.
 *
 * For video, it should typically contain one compressed frame. For audio it may
 * contain several compressed frames. Encoders are allowed to output empty
 * packets, with no compressed data, containing only side data
 * (e.g. to update some stream parameters at the end of encoding).
 *
 */
```

AVPacket 结构体定义，代码如下：

```
//chapter3/3.2.help.txt
typedef struct AVPacket {
    AVBufferRef * buf;
    //显示时间戳,单位为 AVStream->time_base
    int64_t pts;
    //解码时间戳,单位为 AVStream->time_base
    int64_t dts;
    //音视频数据
    uint8_t * data;
    //数据包大小
    int   size;
    //码流索引下标
    int stream_index;
    //帧类型
    int flags;
    //额外数据
    AVPacketSideData * side_data;
    int side_data_elems;
    //帧显示时长,单位为 AVStream->time_base
    int64_t duration;
    //数据包所在码流的 position
    int64_t pos;

} AVPacket;
```

AVPacket 的分配与释放有对应的 API,需要注意的是释放所传的参数为 AVPacket 指针的地址。这些 API 与示例代码如下:

```
//chapter3/3.2.help.txt
//API:
AVPacket * av_packet_alloc(void);              //分配包空间
void av_packet_unref(AVPacket * pkt);          //解引用包
void av_packet_free(AVPacket ** pkt);          //释放包空间

//参考示例代码
AVPacket * pkt = av_packet_alloc();
av_packet_unref(pkt);
av_packet_free(&pkt);
```

2. AVFrame 结构体

AVFrame 结构体位于 frame.h 头文件,用于存储解码后的音视频帧数据,使用 av_frame_alloc 进行分配,使用 av_frame_free 进行释放。AVFrame 分配一次,可多次复用,使用 av_frame_unref 可以解引用。官方关于 AVFrame 的描述如下:

```
//chapter3/3.2.help.txt
/**
 * This structure describes decoded (raw) audio or video data.
 *
 * AVFrame must be allocated using av_frame_alloc(). Note that this only
 * allocates the AVFrame itself, the buffers for the data must be managed
 * through other means (see below).
 * AVFrame must be freed with av_frame_free().
 * 此结构描述解码(原始)音频或视频数据.必须使用 av_frame_alloc()分配 AVFrame.
 * 需要注意,仅分配 AVFrame 本身,必须通过其他方式管理数据的缓冲区.
 * AVFrame 必须使用 av_frame_free()释放.
 *
 * AVFrame 通常分配一次,然后可以重复使用多次以保存不同的数据
 *AVFrame is typically allocated once and then reused multiple times to hold
 * different data (e.g. a single AVFrame to hold frames received from a
 * decoder).
 * In such a case, av_frame_unref() will free any references held by the frame and reset it to
its original clean state before it
 * is reused again.
 * 在这种情况下,av_frame_unref()将释放框架所持有的所有引用,
 * 并将其重置为原始的干净状态,然后重新使用
 */
```

AVFrame 在解码后用于存储音视频帧,包括 data 数组、width、height、pts、pkt_dts、pkt_size、pkt_duration、pkt_pos 等信息,在判断是否为关键帧时,可以使用 key_frame 参数。该结构体的定义,代码如下:

```
//chapter3/3.2.help.txt
typedef struct AVFrame {
#define AV_NUM_DATA_POINTERS 8

    //pointer to the picture/channel planes.//指向图片和通道平面的指针
    uint8_t *data[AV_NUM_DATA_POINTERS];

    /**
      * For video, size in Bytes of each picture line.
      * For audio, size in Bytes of each plane.
      * 对于视频,每张图片行的大小(字节);
      * 对于音频,每个平面的大小,以字节为单位

      * For audio, only linesize[0] may be set. For planar audio, each channel
      * plane must be the same size.
      * 对于音频,只能设置 linesize[0].对于平面音频,每个通道平面的大小必须相同
      * For video the linesizes should be multiples of the CPUs alignment
      * preference, this is 16 or 32 for modern desktop CPUs.
```

```
     * 对于视频,线条大小应为 CPU 对齐的倍数(对于 16 或 32 位的现代桌面 CPU)
     * Some code requires such alignment other code can be slower without
     * correct alignment, for yet other it makes no difference.
     */
    int linesize[AV_NUM_DATA_POINTERS];

    /**
     * pointers to the data planes/channels.
     *
     * For video, this should simply point to data[].对于视频,只需指向 data[]
     *
     * For planar audio, each channel has a separate data pointer, and
     * linesize[0] contains the size of each channel buffer.
     * 对于平面音频,每个通道都有一个单独的数据指针,
     * 并且 linesize[0]包含每个通道缓冲区的大小
     * For packed audio, there is just one data pointer, and linesize[0]
     * contains the total size of the buffer for all channels.
     * 对于压缩音频,只有一个数据指针和 linesize[0],包含所有通道缓冲区的总大小.
     */
    uint8_t **extended_data;

    /**
     * @name Video dimensions:视频的宽度和高度
     */
    int width, height;

    /**
     * number of audio samples (per channel) described by this frame
     * 此帧描述的音频采样数(每个通道)
     */
    int nb_samples;

    /**
     * format of the frame, -1 if unknown or unset
     * Values correspond to enum AVPixelFormat for video frames,
     * enum AVSampleFormat for audio):
     * 像素格式(视频:AVPixelFormat,音频:AVSampleFormat)
     */
    int format;

    /**
     * 1 -> keyframe, 0 -> not          //是否为关键帧
     */
    int key_frame;
```

```
/**
 * Picture type of the frame.          //帧的图像类型
 */
enum AVPictureType pict_type;

/**
 * Sample aspect ratio for the video frame, 0/1 if unknown/unspecified.
 */
AVRational sample_aspect_ratio;      //宽高比

/**
 * Presentation timestamp in time_base units.
 */
int64_t pts; //pts:显示时间戳

/**
 * DTS copied from the AVPacket that triggered returning this frame.
 * This is also the Presentation time of this AVFrame calculated from
 * only AVPacket.dts values without pts values.
 */
int64_t pkt_dts;                     //dts:解码时间戳

/**
 * picture number in bitstream order :编码的图像序号
 */
int coded_picture_number;
/**
 * picture number in display order :显示的图像序号
 */
int display_picture_number;

/**
 * quality (between 1 (good) and FF_LAMBDA_MAX (bad))
 */
int quality;

/**
 * for some private data of the user :私有数据
 */
void *opaque;

/** 解码时,这会指示图片必须延迟多长时间
 * When decoding, this signals how much the picture must be delayed.
 * extra_delay = repeat_pict / (2*fps)
 */
int repeat_pict;
```

```
/** 图片的内容是隔行扫描的
 * The content of the picture is interlaced.
 */
int interlaced_frame;

/** 如果内容隔行扫描,则需确定是否"顶场优先"
 * If the content is interlaced, is top field displayed first.
 */
int top_field_first;

/** 告诉用户应用程序调色板已从上一帧更改
 * Tell user application that palette has changed from previous frame.
 */
int palette_has_changed;

/**
 * reordered opaque 64 bits.
 */
int64_t reordered_opaque;

/**
 * Sample rate of the audio data. 音频数据的采样率
 */
int sample_rate;

/** 音频数据的声道布局
 * Channel layout of the audio data.
 */
uint64_t channel_layout;

/** AVBuffer 引用备份此帧的数据.
 * 如果此数组的所有元素都为 NULL,则该帧不进行参考计数
 * AVBuffer references backing the data for this frame.
 * If all elements of this array are NULL,
 * then this frame is not reference counted.
 */
AVBufferRef * buf[AV_NUM_DATA_POINTERS];

/** 帧标志
 * Frame flags, a combination of @ref lavu_frame_flags
 */
int flags;


/**
```

```
     * YUV colorspace type. YUV 颜色空间类型
     * - encoding: Set by user
     * - decoding: Set by libavcodec
     */
    enum AVColorSpace colorspace;



    //number of audio channels, only used for audio.
    int channels; //音频通道数,仅用于音频

    //size of the corresponding packet containing the compressed frame.
    int pkt_size; //包含压缩帧的相应数据包的大小

   ......//此处省略了一些代码
} AVFrame;
```

其中,帧类型使用 AVPictureType(位于 libavutil/avutil.h 文件中)表示,枚举定义,代码如下:

```
//chapter3/3.2.help.txt
enum AVPictureType {
    AV_PICTURE_TYPE_NONE = 0,      //< Undefined 未知
    AV_PICTURE_TYPE_I,             //< Intra  关键帧,I 帧
    AV_PICTURE_TYPE_P,             //< Predicted P 帧
    AV_PICTURE_TYPE_B,             //< Bi-dir predicted, B 帧
    AV_PICTURE_TYPE_S,             //< S(GMC)-VOP MPEG-4, S 帧
    AV_PICTURE_TYPE_SI,            //< Switching Intra, SI 帧,用于切换
    AV_PICTURE_TYPE_SP,            //< Switching Predicted, SP 帧
    AV_PICTURE_TYPE_BI,            //< BI type, BI 帧
};
```

AVFrame 的分配与释放也有相应的 API,具体的 API 及使用实例,代码如下:

```
//chapter3/3.2.help.txt
//API:
AVFrame *av_frame_alloc(void);
void av_frame_unref(AVFrame *frame);
void av_frame_free(AVFrame **frame);

//demo:
AVFrame *frame = av_frame_alloc();
av_frame_unref(frame);
av_frame_free(&frame);
```

3.3 协议层的三大重要数据结构

协议层，用于处理各种协议，也可以理解为 FFmpeg 的 I/O 处理层，提供了资源的按字节读写能力。这一层的作用：一方面根据音视频资源的 URL 来识别该以什么协议访问该资源（包括本地文件和网络流）；另一方面识别协议后，就可以使用协议相关的方法，例如使用 open()打开资源、使用 read()读取资源的原始比特流、使用 write()向资源中写入原始比特流、使用 seek()在资源中随机检索、使用 close()关闭资源，并提供缓冲区 buffer，所有的操作就像访问一个文件一样。

FFmpeg 的协议层提供了这样一个抽象，像访问文件一样去访问资源，这个概念在 Linux 系统中普遍存在，一切皆是文件。这一层的主要结构体有 3 个，分别为 URLProtocol、URLContext 和 AVIOContext，可以认为这 3 个结构体在协议层也有上下级关系，如图 3-14 所示。

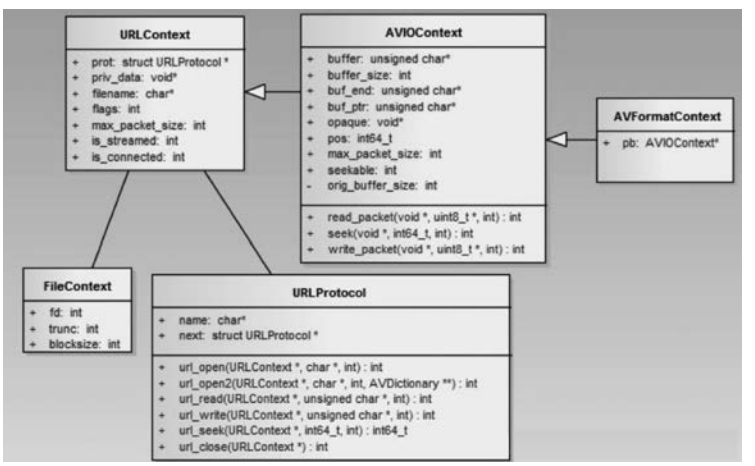

图 3-14　FFmpeg 协议层所涉及的 3 个数据结构

协议操作的顶层结构是 AVIOContext，这个对象实现了带缓冲的读写操作；FFmpeg 的输入对象 AVFormat 的 pb 字段指向一个 AVIOContext。AVIOContext 的 opaque 实际指向一个 URLContext 对象，这个对象封装了协议对象及协议操作对象，其中 prot 指向具体的协议操作对象，priv_data 指向具体的协议对象。URLProtocol 为协议操作对象，针对每种协议，会有一个这样的对象，每个协议操作对象和一个协议对象关联，例如，文件操作对象为 ff_file_protocol，它关联的结构体是 FileContext。

1. URLProtocol 结构体

URLProtocol 是这层中最底层的结构体，持有协议访问方法：每个协议都有其专属的 URLProtocol 结构体，在 FFmpeg 中以常量的形式存在，命名方式是 ff_xxx_protocol，其中

xxx是协议名。URLProtocol的成员函数指针族提供了上述类文件操作的所有方法，如果是网络协议，则网络访问的一切也被封装在这些方法中，可以认为URLProtocol提供了协议的访问方法。该结构体的声明，代码如下：

```
//chapter3/3.3.help.txt
typedef struct URLProtocol {
    const char * name; //协议名称
    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);
    int ( * url_get_file_handle)(URLContext * h);
    struct URLProtocol * next; //指向下一个URLProtocol对象(所有URLProtocol以链表链接在一起)
    int priv_data_size;        //和该URLProtocol对象关联的对象的大小
    const AVClass * priv_data_class;
} URLProtocol;
```

URLProtocol是FFmpeg中操作文件的结构(如文件、网络数据流等)，包括open、close、read、write、seek等回调函数实现具体的操作。在av_register_all()函数中，通过调用REGISTER_PROTOCOL()宏，所有的URLProtocol都保存在以first_protocol为链表头的链表中。

以文件协议为例，ff_file_protocol变量定义的代码如下：

```
//chapter3/3.3.help.txt
URLProtocol ff_file_protocol = {
    .name                = "file",
    .url_open            = file_open,
    .url_read            = file_read,
    .url_write           = file_write,
    .url_seek            = file_seek,
    .url_close           = file_close,
    .url_get_file_handle = file_get_handle,
    .url_check           = file_check,
    .priv_data_size      = sizeof(FileContext),
    .priv_data_class     = &file_class,
};
```

可以看出，.priv_data_size的值为sizeof(FileContext)，即ff_file_protocol和结构体FileContext相关联。

FileContext对象的定义代码如下：

```
//chapter3/3.3.help.txt
typedef struct FileContext {
    const AVClass *class;
    int fd;         //文件描述符
    int trunc;      //截断属性
    int blocksize;  //块大小,每次读写文件的最大字节数
} FileContext;
```

返回去看 ff_file_protocol 里的函数指针,以 url_read 成员为例,它指向 file_read()函数,该函数的定义如下:

```
//chapter3/3.3.help.txt
static int file_read(URLContext *h, unsigned char *buf, int size)
{
    FileContext *c = h->priv_data;
    int r;
    size = FFMIN(size, c->blocksize);
    r = read(c->fd, buf, size);
    return (-1 == r)?AVERROR(errno):r;
}
```

从代码可以看出:

(1) 调用此函数时,URLContext 的 priv_data 指向一个 FileContext 对象。

(2) 该函数每次最大只读取 FileContext.blocksize 大小的数据。

另外,还可发现一个重要的对象:URLContext,这个对象的 priv_data 指向一个 FileContext。

可以看一下 FILE 协议的几个函数(其实就是读、写文件等操作),定义在 libavformat/file.c 文件中,代码如下:

```
//chapter3/3.3.help.txt
/* standard file protocol:标准文件协议 */
//读文件,最终调用 read()
static int file_read(URLContext *h, unsigned char *buf, int size)
{
    int fd = (intptr_t) h->priv_data;
    int r = read(fd, buf, size);
    return (-1 == r)?AVERROR(errno):r;
}
//写文件,最终调用 write()
static int file_write(URLContext *h, const unsigned char *buf, int size)
{
    int fd = (intptr_t) h->priv_data;
    int r = write(fd, buf, size);
```

```
    return ( - 1 == r)?AVERROR(errno):r;
}
//获取文件句柄
static int file_get_handle(URLContext * h)
{
    return (intptr_t) h - > priv_data;
}


# if CONFIG_FILE_PROTOCOL
  //打开文件:最终调用 open()
static int file_open(URLContext * h, const char * filename, int flags)
{
    int access;
    int fd;

    av_strstart(filename, "file:", &filename);

    if (flags & AVIO_FLAG_WRITE && flags & AVIO_FLAG_READ) {
        access = O_CREAT | O_TRUNC | O_RDWR;
    } else if (flags & AVIO_FLAG_WRITE) {
        access = O_CREAT | O_TRUNC | O_WRONLY;
    } else {
        access = O_RDONLY;
    }
# ifdef O_BINARY
    access | = O_BINARY;
# endif
    fd = open(filename, access, 0666);
    if (fd == - 1)
        return AVERROR(errno);
    h - > priv_data = (void * ) (intptr_t) fd;
    return 0;
}

/ * XXX: use llseek * /
//定位文件:最终调用 lseek()
static int64_t file_seek(URLContext * h, int64_t pos, int whence)
{
    int fd = (intptr_t) h - > priv_data;
    if (whence == AVSEEK_SIZE) {
        struct stat st;
        int ret = fstat(fd, &st);
        return ret < 0 ? AVERROR(errno) : st.st_size;
    }
    return lseek(fd, pos, whence);
```

```
}
//关闭文件:最终调用 close()
static int file_close(URLContext * h)
{
    int fd = (intptr_t) h->priv_data;
    return close(fd);
}
```

2. URLContext 结构体

URLContext 是协议上下文对象,是 URLProtocol 上一层的结构体,持有协议访问方法及当前访问状态信息:通过持有 URLProtocol 对象而持有协议访问方法,并且通过持有另外一个协议相关的状态信息结构体来持有当前协议访问的状态信息。持有状态信息的这个结构体名称与协议名相关,以 HTTP 协议为例,相应结构体名称为 HTTPContext。注意一点:有些相关的协议会映射到同一种状态信息的结构体上,例如 http、https、httpproxy 对应的 URLProtocol 结构体为 ff_http_protocol、ff_https_protocol 和 ff_httpproxy_protocol,但是这 3 个协议对应同一种状态信息上下文结构体 HttpContext。再例如 FILE、PIPE 协议对应的 URLProtocol 结构体为 ff_file_protocol 和 ff_pipe_protocol,二者对应同一种状态信息上下文结构体 FileContext。该结构体的声明,代码如下:

```
//chapter3/3.3.help.txt
typedef struct URLContext {
    const AVClass * av_class;     /**< information for av_log(). Set by url_open(). */
    const struct URLProtocol * prot; //指向某种 URLProtocol
    void * priv_data;          //一般用来指向某种具体协议的上下文,如 FileContext
    char * filename;           /**< specified URL */
    int flags;
    int max_packet_size;       /**< if non zero, the stream is packetized with this max packet
size */
    int is_streamed;           /**< true if streamed (no seek possible), default = false */
    int is_connected;
    AVIOInterruptCB interrupt_callback;
    int64_t rw_timeout;        /**< maximum time to wait for (network) read/write operation
completion, in mcs */
    const char * protocol_whitelist;
    const char * protocol_blacklist;
    int min_packet_size;       /**< if non zero, the stream is packetized with this min packet
size */
} URLContext;
```

可以看出,URLContext 的 filename 保存了输入文件名,prot 字段保存了查找到的协议操作对象指针,flags 由入参指定,is_streamed 及 max_packet_size 的默认值为 0,priv_data 指向了一个由协议操作对象的 priv_data_size 指定大小的空间,并且该空间被初始化为 0。URLContext 和 URLProtocol 的关系如图 3-15 所示。

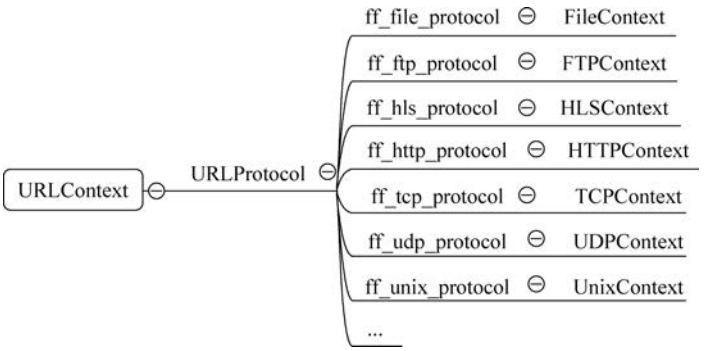

图 3-15　URLContext 和 URLProtocol 的关系

3. AVIOContext 结构体

AVIOContext 是协议层最上一层的结构体，可以认为是协议层的公共 API，提纲挈领的 AVFormatContext 通过持有 AVIOContext 而具备 IO 访问能力。AVIOContext 通过持有 URLContext 而持有协议访问方法及访问状态，同时内部再提供一个读写缓冲区。注意是读写缓冲区，既可以作为读缓冲区，也可以作为写缓冲区，当然同一时刻只支持读或者写，代码如下：

```
//chapter3/3.3.help.txt
typedef struct AVIOContext {
    const AVClass *av_class;
    unsigned char *buffer;  /**< 读写缓冲 buffer 起始地址 */
    int buffer_size;        /**< buffer 大小 */
    unsigned char *buf_ptr; /**< Current position in the buffer */
    unsigned char *buf_end; /**< End of the data */
    void *opaque;           /**< 指向 URLContext */
    int (*read_packet)(void *opaque, uint8_t *buf, int buf_size); /* 指向 ffurl_read
() */
    int (*write_packet)(void *opaque, uint8_t *buf, int buf_size); /* 指向 ffurl_write
() */
    int64_t (*seek)(void *opaque, int64_t offset, int whence); /* 指向 ffurl_seek() */
    int64_t pos;            /**< 当前 buffer 对应的文件内容中的位置 */
    int must_flush;         /**< true if the next seek should flush */
    int eof_reached;        /**< true if eof reached */
    int write_flag;         /**< true if open for writing */
    int max_packet_size;
    unsigned long checksum;
    unsigned char *checksum_ptr;
    unsigned long (*update_checksum)(unsigned long checksum, const uint8_t *buf, unsigned
int size);
```

```
    int error;                /** < contains the error code or 0 if no error happened */
    int (* read_pause)(void * opaque, int pause);/
    int64_t (* read_seek)(void * opaque, int stream_index,
                          int64_t timestamp, int flags);

    int seekable;             //是否可搜索,0 表示不可搜索
    int64_t maxsize;
    int direct;
    int64_t Bytes_read;
    int seek_count;
    int writeout_count;
    int orig_buffer_size;
}AVIOContext;
```

URLContext 再往上一层是 AVIOContext，AVIOContext. buffer 指向申请到的 buffer，AVIOContext. orig_buffer_size 和 AVIOContext. buffer_size 值为 buffer_size，buf_ptr 字段初始化为 buffer 的开始地址，opaque 指向 URLContext 对象。

AVIOContext 中有以下几个变量比较重要。

(1) unsigned char * buffer：缓存开始位置。

(2) int buffer_size：缓存大小(默认为 32 768)。

(3) unsigned char * buf_ptr：当前指针读取的位置。

(4) unsigned char * buf_end：缓存结束的位置。

(5) void * opaque：URLContext 结构体。

在解码的情况下，buffer 用于存储 FFmpeg 读入的数据。例如当打开一个视频文件时，先把数据从硬盘读入 buffer，然后送给解码器用于解码；其中 opaque 指向了 URLContext，这样就可以和 URLContext 关联上。

3.4 封装层的四大重要数据结构

1. AVFormatContext 结构体

AVFormatContext 可以说是贯穿全局的数据结构，很多函数要用它作为参数。此结构包含了一个视频流的格式内容，其中 AVInputFormat 或者 AVOutputFormat(同一时间 AVFormatContext 内只能存在其中一个)、AVStream 和 AVPacket 等几个重要的结构及一些其他信息，例如 title、author、copyright 等，还有一些可能在编解码中会用到的信息，例如 duration、file_size、bit_rate 等。在 FFmpeg 中，绝大多数结构体的使用是通过指针来使用的，而分配和释放内存的地方都由特定的函数调用来决定，不需要开发者手动操作，例如为 AVFormatContext 分配内存的代码如下：

```
AVFormatContext * formatCtx = avformat_alloc_context();
```

AVFormatContext结构体非常重要，函数声明在libavformat/avformat.h文件中，代码如下（各个字段的含义详见注释信息）：

```
//chapter3/3.4.help.txt
typedef struct AVFormatContext {
const AVClass * av_class; //与 logging 及 avoptions 相关的 class,由 avformat_alloc_context()
//设置
    struct AVInputFormat * iformat;  //输入容器格式,只用在解复用(demuxing only),
//由 avformat_open_input()设置
    struct AVOutputFormat * oformat;  //输出容器格式,只用在复用(muxing only),必须在调用
//avformat_write_header()之前设置
    void * priv_data;   //格式私有数据,在复用时由 avformat_write_header()设置 demuxing:
//由 avformat_open_input() 设置
     AVIOContext * pb;  //I/O context. 在解复用时,在 avformat_open_input()调用之前设置,
//或者由 avformat_open_input()设置;在复用时,在 avformat_write_header()之前由使用者设置
    /* 流信息 */
    int ctx_flags;
    unsigned int nb_streams;  //流的数目
    AVStream ** streams;  //文件所有流的列表,如果要创建新的流,则可以通过 avformat_new_
//stream()创建
    char filename[1024];  //输入或者输出的文件名,解复用时由 avformat_open_input()设置,
//复用时可以在 avformat_write_header()之前设置
    int64_t start_time;  //开始帧的位置,不要直接设置.只在解复用时用到(Demuxing only)
    int64_t duration;   //流的时长
    int64_t bit_rate;   //整个流的比特率
    unsigned int packeet_size;
    int max_delay;
    int flags;   //Demuxer/Muxer 的状态
    //此处省略宏定义
    int64_t probesize;   //通过 AVInputFormat 从输入流中读到数据的最大大小,在 avformat_
//open_input()之前设置,只在解复用使用(Demuxing only)
    int64_t max_analyze_duration;  //从来自 avformat_find_stream_info()输入流中读到数据的
//最大大小,在 avformat_find_stream_info()前设置
    const uint8_t * key;
    int keylen;
    unsigned int nb_programs;
    AVProgram ** programs;
    enum AVCodecID video_codec_id;   //视频编解码器 id,在解码时由 user 设置
    enum AVCodecID audio_codec_id;  //音频编解码器 id,在解码时由 user 设置
    enum AVCodecID subtitle_codec_id;  //字母编解码器 id,在解码时由 user 设置
    unsigned int max_index_size;   //每条流的最大内存字节数
    unsigned int max_picture_buffer;  //Buffering frames 的最大内存字节数
    unsigned int nb_chapters;  //AVChapters array 的 chapters 的数量
```

```
    AVChapter **chapters;
    AVDictionary *metadata;   //元数据
    int64_t start_time_realtime;  //起始时间,从 PTS = 0 开始
    int fps_probe_size;  //用在 avformat_find_stream_info()中,用于确定帧率,其值为帧数.
//只在解复用中
    int error_recognition;  //错误检测
    AVIOInterruptCB interrupt_callback;  //自定义
    int Debug;  //flags to enable Debugging
#define FF_FDebug_TS         0x0001
    int64_t max_interleave_delta; //最大交叉 Buffering(缓冲数据)时长,在 muxing(复用)时使用
    int strict_std_compliance;   //允许非标准拓展
    int event_flags;  //用户检测文件发生事件的标识
    int max_ts_probe;   //解码第 1 帧(第 1 个时间戳)时读取的最大 Packet 数目
    int avoid_negative_ts;  //在复用(muxing)过程中避免无效的时间戳(timestamps)
#define AVFMT_AVOID_NEG_TS_AUTO              -1 //< Enabled when required by target format
#define AVFMT_AVOID_NEG_TS_MAKE_NON_NEGATIVE 1 //< Shift timestamps so they are non negative
#define AVFMT_AVOID_NEG_TS_MAKE_ZERO         2 //< Shift timestamps so that they start at 0
    int ts_id;   //ts(Transport)流的 id
    int audio_preload;  //音频提前加载,不是所有格式都支持
    int max_chunk_duration;  //最大 chunk 时长,不是所有格式都支持
    int max_chunk_size;   //最大 chunk 以字节为单位,不是所有格式都支持
    int use_wallclock_as_timestamps;  //强制使用 wallclock 时间戳作为数据包的 pts/dts,如果
//有 b 帧存在,则会有未定义的结果出现
    int avio_flags;
    enum AVDurationEstimationMethod duration_estimation_method;   //可以通过不同的方式估计
//持续时间字段
    int64_t skip_initial_Bytes;  //当打开流时,跳过初始字节
    unsigned int correct_ts_overflow;  //纠正单个时间戳溢出
    int seek2any;   //强制搜索到任一帧
    int flush_packets;   //在每个 packet 之后,刷新 I/O Context
    int probe_score;   //格式探测评分,最大评分是 AVPROBE_SCORE_MAX
    int format_probesize;  //读取最大的字节数来确定格式
    char *codec_whitelist;  //由','分隔的所有可用的 decoder(解码器)
    char *format_whitelist;   //由','分隔的所有可用的 demuxers(解复用器)
    AVFormatInternal *internal;   //libavformat 内部私有成员
    int io_repositioned;   //I/O 更改的标志
    AVCodec *video_codec;  //Forced video codec,特殊解码器或者相同 codec_id 的视频 Codec
    AVCodec *audio_codec;   //Forced audio codec, 特殊解码器或者相同 codec_id 的音频 Codec
    AVCodec *subtitle_codec;//Forced subtitle codec, 特殊解码器或者相同 codec_id 的字幕 Codec
    AVCodec *data_codec;  //Forced data codce,特殊解码器或者相同 codec_id 的数据 Codec
    int metadata_header_padding;  //在 metadata(元数据)头设置 padding 值
    void *opaque;   //用户私有数据
    av_format_control_message control_message_cb;  //设备和应用通信用的 Callback
    int64_t output_ts_offset;   //输出时间戳偏移量
    uint8_t *dump_separator;  //转储分隔格式, 可以是","或者"\n"或者其他
    enum AVCodecId data_codec_id;  //Forced Data codec_id
```

```
    int ( * open_cb)(struct AVFormatContext * s, AVIOContext ** p, const char * url, int
flags, const AVIOInterruptCB * int_cb, AVDictionary ** options);   //过时函数,用 io_open_
//and_io_close 代替
    char * protocol_whitelist;                                      //协议白名单,用','分隔
    int ( * io_open)(struct AVFormatContext * s, AVIOContext ** pb, const char * url, int
flags, AVDictionary ** options);   //当 I/O 流打开时,解复用操作的回调函数
    void ( * io_close)(struct AVFormatContext * s, AVIOContext * pb);   //AVFormatContext 打
//开时的回调函数
    char * protocol_blacklist;   //协议黑名单
    int max_streams;   //流的最大数量,在 decodeing 时设置
}AVFormatContext;
```

2. AVInputFormat 结构体

AVInputFormat 是类似 COM 接口的数据结构，表示输入文件容器格式，着重于功能函数，一种文件容器格式对应一个 AVInputFormat 结构，在程序运行时有多个实例，位于 libavformat/avoformat.h 文件中。

AVInputFormat 在解复用器(解封装)时读取媒体文件并将其拆分为数据块(数据包)。每个数据包包含一个或者多个编码帧。比较重要的字段如下。

(1) long_name：格式的长名称(相对于短名称而言，更易于阅读)。

(2) mime_type：mime 类型，它用于在探测时检查匹配的 mime 类型。

(3) next：用于链接下一个 AVInputFormat。

(4) (* read_probe)：判断给定文件是否有可能被解析为此格式。提供的缓冲区保证为 AVPROBE_PADDING_SIZE 字节大小，因此除非需要更多，否则无须检查。

(5) (* read_header)：读取格式头，并初始化 AVFormatContext 结构体。

(6) (* read_packet)：读取一个 packet 并存入 pkt 指针中。

该结构体的声明代码如下(各个字段的含义详见注释信息)：

```
//chapter3/3.4.help.txt
typedef struct AVInputFormat {
    const char * name; //输入格式的短名称
    const char * long_name; //格式的长名称(相对于短名称而言,更易于阅读)
    /**
     * Can use flags: AVFMT_NOFILE, AVFMT_NEEDNUMBER, AVFMT_SHOW_IDS,
     * AVFMT_GENERIC_INDEX, AVFMT_TS_DISCONT, AVFMT_NOBINSEARCH,
     * AVFMT_NOGENSEARCH, AVFMT_NO_BYTE_SEEK, AVFMT_SEEK_TO_PTS.
     */
    int flags;
    const char * extensions; //如果定义了扩展,就不会进行格式探测,但因为该功能目前支持不
//够,不推荐使用
    const struct AVCodecTag * const * codec_tag; //编解码标签,4 字节码
    const AVClass * priv_class; //< AVClass for the private context
```

```
    const char * mime_type; //mime 类型,它用于在探测时检查匹配的 mime 类型
        /* 此行下方的任何字段都不是公共 API 的一部分. 它们不能在 libavformat 之外使用,可
以随意更改和删除.
         * 应在上方添加新的公共字段. */
    struct AVInputFormat * next; //用于链接下一个 AVInputFormat
    int raw_codec_id; //原始 demuxers 将它们的解码器 id 保存在这里
    int priv_data_size; //私有数据大小,可以用于确定需要分配多大的内存来容纳下这些数据
        /**
         * 判断给定文件是否有可能被解析为此格式. 提供的缓冲区保证为 AVPROBE_PADDING_
SIZE 字节大小,因此除非需要更多,否则无须检查.
         */
    int ( * read_probe)(AVProbeData * );
    /**
     * 读取格式头,并初始化 AVFormatContext 结构体
     * @return 0 表示操作成功
     */
    int ( * read_header)(struct AVFormatContext * );
    /**
     * 读取一个 packet 并存入 pkt 指针中. pts 和 flags 会被同时设置.
     * @return 0 表示操作成功, < 0 表示发生异常
     *          当返回异常时,pkt 可定没有 allocated 或者在函数返回之前被释放了.
     */
    int ( * read_packet)(struct AVFormatContext * , AVPacket * pkt);
     //关闭流,AVFormatContext 和 AVStreams 并不会被这个函数释放
    int ( * read_close)(struct AVFormatContext * );
    /**
     * 在 stream_index 的流中,使用一个给定的 timestamp,搜索到附近帧.
     * @param stream_index 不能为 - 1
     * @param flags 如果没有完全匹配,则决定向前还是向后匹配.
     * @return >= 0 成功
     */
    int ( * read_seek)(struct AVFormatContext * ,
                       int stream_index, int64_t timestamp, int flags);
    //获取 stream[stream_index]的下一个时间戳,如果发生异常,则返回 AV_NOPTS_VALUE
    int64_t ( * read_timestamp)(struct AVFormatContext * s, int stream_index,
                                int64_t * pos, int64_t pos_limit);
    //开始或者恢复播放,只有在播放 RTSP 格式的网络格式才有意义
    int ( * read_play)(struct AVFormatContext * );
    int ( * read_pause)(struct AVFormatContext * );  //暂停播放,只有在播放 RTSP 格式的网络
//格式才有意义
    /**
     * 快进到指定的时间戳
     * @param stream_index 需要快进操作的流
     * @param ts 需要快进到的地方
     * @param min_ts max_ts seek 的区间,ts 需要在这个范围中.
     */
```

```
    int ( * read_seek2)(struct AVFormatContext * s, int stream_index, int64_t min_ts, int64_t
ts, int64_t max_ts, int flags);
    //返回设备列表和其属性
    int ( * get_device_list)(struct AVFormatContext * s, struct AVDeviceInfoList * device_
list);
    //初始化设备能力子模块
      int ( * create _ device _ capabilities ) ( struct AVFormatContext * s, struct
AVDeviceCapabilitiesQuery * caps);
    //释放设备能力子模块
    int ( * free_device_capabilities)(struct AVFormatContext * s, struct AVDeviceCapabilitiesQuery
* caps);
} AVInputFormat;
```

3. AVOutputFormat 结构体

AVOutpufFormat 与 AVInputFormat 类似，是类似于 COM 接口的数据结构，表示输出文件容器格式，着重于功能函数，位于 libavformat/avoformat.h 文件中。FFmpeg 支持各种各样的输出文件格式，包括 MP4、FLV、3GP 等，而 AVOutputFormat 结构体则保存了这些格式的信息和一些常规设置。每种封装对应一个 AVOutputFormat 结构，FFmpeg 将 AVOutputFormat 按照链表存储，如图 3-16 所示。

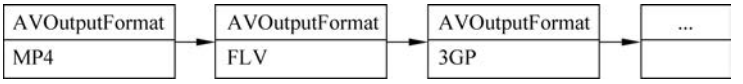

图 3-16 AVOutputFormat 按照链表结构存储

```
//chapter3/3.4.help.txt
typedef struct AVOutputFormat {
    const char * name;
    /**
     * Descriptive name for the format, meant to be more human-readable
     * than name. You should use the NULL_IF_CONFIG_SMALL() macro
     * to define it.
     */
    const char * long_name;
    const char * mime_type;
    const char * extensions; /**< comma-separated filename extensions */
    /* output support */
    enum AVCodecID audio_codec;     /**< default audio codec */
    enum AVCodecID video_codec;     /**< default video codec */
    enum AVCodecID subtitle_codec; /**< default subtitle codec */
    /**
     * can use flags: AVFMT_NOFILE, AVFMT_NEEDNUMBER,
     * AVFMT_GLOBALHEADER, AVFMT_NOTIMESTAMPS, AVFMT_VARIABLE_FPS,
```

```
     * AVFMT_NODIMENSIONS, AVFMT_NOSTREAMS, AVFMT_ALLOW_FLUSH,
     * AVFMT_TS_NONSTRICT, AVFMT_TS_NEGATIVE
     */
    int flags;

    /**
     * List of supported codec_id - codec_tag pairs, ordered by "better
     * choice first". The arrays are all terminated by AV_CODEC_ID_NONE.
     */
    const struct AVCodecTag * const * codec_tag;

    int (* write_header)(struct AVFormatContext *);
    /**
     * Write a packet. If AVFMT_ALLOW_FLUSH is set in flags,
     * pkt can be NULL in order to flush data buffered in the muxer.
     * When flushing, return 0 if there still is more data to flush,
     * or 1 if everything was flushed and there is no more buffered
     * data.
     */
    int (* write_packet)(struct AVFormatContext *, AVPacket *pkt);
    int (* write_trailer)(struct AVFormatContext *);
   ...
} AVOutputFormat;
```

该结构体中的常见字段及其作用,伪代码如下:

```
//chapter3/3.4.help.txt
const char * name;                                  //名称
const char * long_name;                             //格式的描述性名称,易于阅读
enum AVCodecID audio_codec;                         //默认的音频编解码器
enum AVCodecID video_codec;                         //默认的视频编解码器
enum AVCodecID subtitle_codec;                      //默认的字幕编解码器
struct AVOutputFormat * next;
int (* write_header)(struct AVFormatContext *);
int (* write_packet)(struct AVFormatContext *, AVPacket *pkt);   //写一个数据包. 如果在标
//志中设置 AVFMT_ALLOW_FLUSH,则 pkt 可以为 NULL
int (* write_trailer)(struct AVFormatContext *); //写文件尾
int (* interleave_packet)(struct AVFormatContext *, AVPacket * out, AVPacket * in, int
flush);
int (* control_message)(struct AVFormatContext * s, int type, void * data, size_t data_size);
                                                    //允许从应用程序向设备发送消息
int (* write_uncoded_frame)(struct AVFormatContext *, int stream_index,   AVFrame ** frame,
unsigned flags);                                    //写一个未编码的 AVFrame
int (* init)(struct AVFormatContext *);   //初始化格式.可以在此处分配数据,并设置在发送数
//据包之前需要设置的任何 AVFormatContext 或 AVStream 参数
```

```
void ( * deinit)(struct AVFormatContext * );          //取消初始化格式
int ( * check_bitstream)(struct AVFormatContext * , const AVPacket * pkt);  //设置任何必要的
//比特流过滤,并提取全局头部所需的任何额外数据
```

4. AVStream 结构体

AVStream 是存储每个视频/音频流信息的结构体，在 AVFormatContext 结构体中有两个参数，代码如下：

```
unsigned int nb_streams;
AVStream ** streams;
```

nb_streams 是当前轨道数，也就是流数量，streams 是轨道的指针数组。一般而言一个视频文件会有一个视频流和一个音频流，也就是两个 AVStream 结构。

AVStream 结构体是在 libavformat/avformat.h 文件中声明的，由于代码较长，这里只显示部分基础字段，代码如下(详细含义可以参考注释信息)：

```
//chapter3/3.4.help.txt
/**
 * Stream structure. :流结构体
 * New fields can be added to the end with minor version bumps.
 * Removal, reordering and changes to existing fields require a major
 * version bump.
 * 可以在较小的版本变更中在末尾添加新字段.
 * 如果删除、重新排序和更改现有字段,则需要进行主要版本升级
 * sizeof(AVStream) must not be used outside libav *
 * sizeof(AVStream)不能在 libav 之外使用
 */
typedef struct AVStream {
    //AVFormatContext 中的流索引
    int index;    /** < stream index in AVFormatContext */
    /**
     * Format-specific stream ID. :特定格式的流 ID
     * decoding: set by libavformat,被 libavformat 解码
     * encoding: set by the user, replaced by libavformat if left unset
     * 编码:优先被用户设置,否则被 libavformat 设置
     */
    int id;
#if FF_API_LAVF_AVCTX
    /**
     * @deprecated use the codecpar struct instead,
     * 该字段已过时,推荐使用 codecpar 字段
     */
    attribute_deprecated
```

```
    AVCodecContext *codec;
#endif
    void *priv_data;

    /**
     * This is the fundamental unit of time (in seconds) in terms
     * of which frame timestamps are represented.
     *这是时间的基本单位(秒),表示帧时间戳.
     * decoding: set by libavformat
     * encoding: May be set by the caller before avformat_write_header() to
     *           provide a hint to the muxer about the desired timebase. In
     *           avformat_write_header(), the muxer will overwrite this field
     *           with the timebase that will actually be used for the timestamps
     *           written into the file (which may or may not be related to the
     *           user-provided one, depending on the format).
     * 可以由调用方在 avformat_write_header()函数之前设置为向 muxer 提供有关所需时基的提
示信息.在 avformat_write_header()函数的内部,muxer 将覆盖此字段,实际用于时间戳的时基写入
文件(可能由用户根据格式提供)
     */
    AVRational time_base;

    /**
     * Decoding: pts of the first frame of the stream in presentation order, in stream time
base.流的第 1 帧在流时基中按表示顺序的 pts
     * Only set this if you are absolutely 100 % sure that the value you set
     * it to really is the pts of the first frame.
     *如果百分之百确定所设置的值是第 1 帧的 pts,则可以设置该字段的值
     * This may be undefined (AV_NOPTS_VALUE),也可能未定义(AV_NOPTS_VALUE).
     * @note The ASF header does NOT contain a correct start_time the ASF
     * demuxer must NOT set this.
     * ASF 格式头不包含正确的 start_time,所以一定不可以设置该值
     */
    int64_t start_time;

    /**
     * Decoding: duration of the stream, in stream time base.流时间基
     * If a source file does not specify a duration, but does specify
     * a bitrate, this value will be estimated from bitrate and file size.
     * 如果源文件未指定 duration,但指定了比特率,则该值将根据比特率和文件大小进行估计
     * Encoding: May be set by the caller before avformat_write_header() to
     * provide a hint to the muxer about the estimated duration.
     * 编码:可以由调用方在 avformat_write_header()函数之前
     *设置为向 muxer 提供有关估计持续时间的提示
     */
    int64_t duration;
```

```
 //帧数,未知,设为 0
int64_t nb_frames;  //< number of frames in this stream if known or 0

AVDictionary * metadata; //元数据

/**
  * Average framerate :平均帧率
  *
  *  - demuxing: May be set by libavformat when creating the stream or in
  *              avformat_find_stream_info().
  *  - muxing: May be set by the caller before avformat_write_header().
  */
AVRational avg_frame_rate;


/**
  * Real base framerate of the stream.流的实际基本帧率
  * This is the lowest framerate with which all timestamps can be
  * represented accurately (it is the least common multiple of all
  * framerates in the stream). Note, this value is just a guess!
  * For example, if the time base is 1/90000 and all frames have either
  * approximately 3600 or 1800 timer ticks, then r_frame_rate will be 50/1.
  * 这是可以准确表示所有时间戳的最低帧速率(它是流中所有帧速率中最不常见的倍数)
  * 注意,这个值只是一个猜测值
  * 例如,如果时基为 1/90 000,并且所有帧都有大约 3600 或 1800 个计时器刻度,
  * 则 r_frame_rate 将为 50/1
  */
AVRational r_frame_rate;


/**
  * For streams with AV_DISPOSITION_ATTACHED_PIC disposition, this packet
  * will contain the attached picture.
  * 对于具有 AV_DISPOSITION_ATTACHED_PIC DISPOSITION 的流,
  * 此数据包将包含所附图片
  * decoding: set by libavformat, must not be modified by the caller.
  * encoding: unused
  */
AVPacket attached_pic;


/**
  * Codec parameters associated with this stream. Allocated and freed by
  * libavformat in avformat_new_stream() and avformat_free_context()
  * respectively.
  * 与此流关联的编解码器参数.由 avformat_new_stream()进行分配,
```

```
         * 由 avformat_free_context()进行释放.
         *  - demuxing: filled by libavformat on stream creation or in
         *              avformat_find_stream_info()
         *  - muxing: filled by the caller before avformat_write_header()
         */
        AVCodecParameters *codecpar;


    } AVStream;
```

各个字段信息的说明如下：

(1) int index：标识该视频/音频流。

(2) AVCodecContext *codec：指向该视频/音频流的 AVCodecContext(它们是一一对应的关系)。

(3) AVRational time_base：时间基。通过该值可以把 PTS、DTS 转换为真正的时间。

(4) int64_t duration：该视频/音频流的时长。

(5) AVDictionary *metadata：元数据信息。

(6) AVRational avg_frame_rate：帧率。

(7) AVPacket attached_pic：附带的图片，例如一些 MP3、AAC 音频文件附带的专辑封面。

(8) nb_frames：帧个数。

(9) AVCodecParameters *codecpar：包含音视频参数的结构体，该字段非常重要，可以用于获取音视频参数中的宽度、高度、采样率和编码格式等信息。

3.5 编解码层的三大重要数据结构

1. AVCodecParameters 结构体

AVCodecParameters 结构体是将 AVCodecContext 中编解码器参数抽取出而形成的新的结构体，在新版本的 FFmpeg 中，有些结构体中的 AVCodecContext 已经被弃用，取而代之的是 AVCodecParameters，先介绍几个比较重要的字段，如图 3-17 所示。

(1) enum AVMediaType codec_type：编码类型，说明这段流数据是音频还是视频。

(2) enum AVCodecID codec_id：编码格式，说明这段流的编码格式，例如 H.264、MPEG4、MJPEG 等。

(3) uint32_t codecTag：编解码标志值，一般是 4 字节的英文字符串(FourCC 格式)。

(4) int format：格式。对于视频来讲指的是像素格式；对于音频来讲，指的是音频的采样格式。

(5) int width，int height：视频的宽和高。

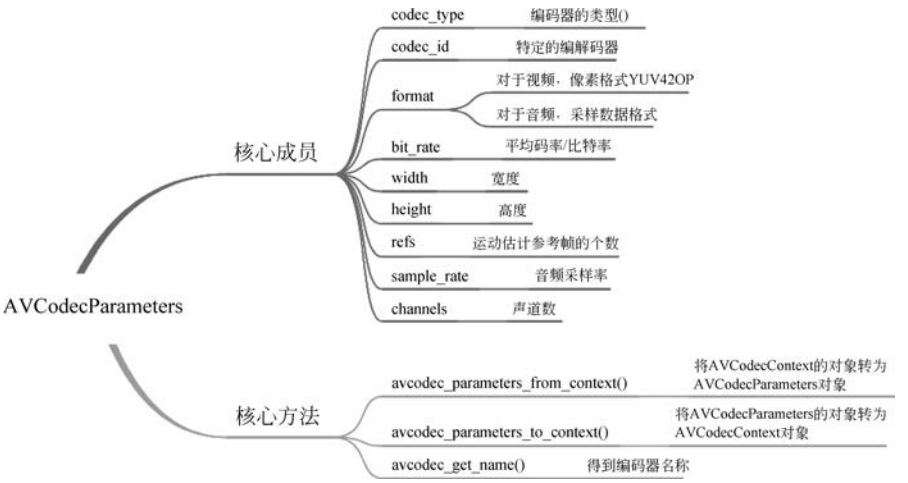

图 3-17　AVCodecParameters 的成员及相关 API

(6) uint64_t channel_layout：声道模式，例如单声道、立体声等。

(7) int channels：声道数。

(8) int sample_rate：采样率。

(9) int frame_size：帧大小，只针对音频，指一帧音频的大小。

AVCodecParameters 结构体定义在 libavcodec/codec_par.h 文件中，代码如下(详见注释信息)：

```
//chapter3/3.5.help.txt
/**
 * This struct describes the properties of an encoded stream.
 * 此结构用于描述编码流的属性.sizeof(AVCodecParameters)不是公共 ABI 的一部分
 * sizeof(AVCodecParameters) is not a part of the public ABI, this struct must
 * be allocated with avcodec_parameters_alloc() and freed with
 * avcodec_parameters_free().
 * 必须被 avcodec_parameters_alloc()分配，被 avcodec_parameters_free()释放
 */
typedef struct AVCodecParameters {
    /**
      * General type of the encoded data. 编码数据的类型
      */
    enum AVMediaType codec_type;
    /**
      * Specific type of the encoded data (the codec used).
      */
    enum AVCodecID   codec_id;
    /** 编码器的标志信息，FourCC 格式
      * Additional information about the codec (corresponds to the AVI FOURCC).
```

```
     * /
    uint32_t          codec_tag;

    /**
     * Extra binary data needed for initializing the decoder, codec-dependent.
     * 初始化解码器所需的额外二进制数据,取决于编解码器
     * Must be allocated with av_malloc() and will be freed by
     * avcodec_parameters_free(). The allocated size of extradata must be at
     * least extradata_size + AV_INPUT_BUFFER_PADDING_SIZE, with the padding
     * Bytes zeroed. 分配的字节数至少是 sizeof(extradata) +填充字节数
     * /
    uint8_t * extradata;
    /**
     * Size of the extradata content in Bytes. 额外数据的字节数
     * /
    int       extradata_size;

    /**
     * - video: the pixel format, the value corresponds to enum AVPixelFormat.
     * - audio: the sample format, the value corresponds to enum AVSampleFormat.
     * 音频或视频格式,分别对应结构体:AVSampleFormat、AVPixelFormat
     * /
    int format;

    /** 编码数据的平均比特率,单位为 b/s
     * The average bitrate of the encoded data (in bits per second).
     * /
    int64_t bit_rate;

    /**
     * The number of bits per sample in the codedwords.
     * 码字中每个样本的位数
     * This is basically the bitrate per sample. It is mandatory for a bunch of  formats to
actually decode them. It's the number of bits for one sample in the actual coded bitstream.
     * 这基本上是每个样本的比特率.对于一组格式来讲,必须对其进行实际解码.它是实际编码
比特流中一个样本的比特数
     * This could be for example 4 for ADPCM
     * For PCM formats this matches bits_per_raw_sample
     * Can be 0
     * /
    int bits_per_coded_sample;

    /** 这是每个输出样本中的有效位数
     * This is the number of valid bits in each output sample. If the
     * sample format has more bits, the least significant bits are additional
     * padding bits, which are always 0. Use right shifts to reduce the sample
```

```
     * to its actual size. For example, audio formats with 24 bit samples will
     * have bits_per_raw_sample set to 24, and format set to AV_SAMPLE_FMT_S32.
     * To get the original sample use "(int32_t)sample >> 8"."
     * 如果样本格式有更多的位,则最低有效位是额外的填充位,通常为0.使用右移将样本缩小到实际大小.例如,具有24位采样的音频格式将bits_per_raw_sample设置为24,将format设置为AV_sample_FMT_S32
     * For ADPCM this might be 12 or 16 or similar
     * Can be 0
     */
    int bits_per_raw_sample;

    /** 流符合的特定于编解码器的位流限制,例如libx264的profile和level参数
     * Codec-specific bitstream restrictions that the stream conforms to.
     */
    int profile;
    int level;

    /** 视频帧的宽和高,单位为像素
     * Video only. The dimensions of the video frame in pixels.
     */
    int width;
    int height;

    /**
     * Video only. The aspect ratio (width / height) which a single pixel
     * should have when displayed. 单像素的纵横比(宽/高)
     * 当纵横比未知/未定义时,分子应设置为0(分母可以为任何值)
     * When the aspect ratio is unknown / undefined, the numerator should be
     * set to 0 (the denominator may have any value).
     */
    AVRational sample_aspect_ratio;

    /** 隔行扫描视频中的场顺序,例如顶场优先、底场优先
     * Video only. The order of the fields in interlaced video.
     */
    enum AVFieldOrder                  field_order;

    /** 额外的颜色空间特性
     * Video only. Additional colorspace characteristics.
     */
    enum AVColorRange                  color_range;
    enum AVColorPrimaries              color_primaries;
    enum AVColorTransferCharacteristic color_trc;
    enum AVColorSpace                  color_space;
    enum AVChromaLocation              chroma_location;
```

```
    /**
     * Video only. Number of delayed frames. 延迟帧数
     */
    int video_delay;

    /**
     * Audio only. The channel layout bitmask. May be 0 if the channel layout is  unknown or
unspecified, otherwise the number of bits set must be equal to the channels field. 声道布局位掩
码.如果声道布局未知或未指定,则可能为0,否则设置的位数必须等于声道字段
     */
    uint64_t channel_layout;
    /**
     * Audio only. The number of audio channels.音频的声道数
     */
    int      channels;
    /**
     * Audio only. The number of audio samples per second. 采样率
     */
    int      sample_rate;
    /**
     * Audio only. The number of Bytes per coded audio frame, required by some
     * formats.
     * 某些格式所需的每个编码音频帧的字节数
     * Corresponds to nBlockAlign in WAVEFORMATEX.
     */
    int      block_align;
    /**
     * Audio only. Audio frame size, if known. Required by some formats to be static. 音频帧大
小(如果已知).某些格式要求为静态
     */
    int      frame_size;

    /**
     * Audio only. The amount of padding (in samples) inserted by the encoder at the beginning
of the audio. I.e. this number of leading decoded samples must be discarded by the caller to get
the original audio without leading padding.编码器在音频开头插入的填充量(以样本为单位).例如
调用者必须丢弃这一数量的前导解码样本,以便在没有前导填充的情况下获得原始音频
     */
    int initial_padding;

     /**
     * Audio only. The amount of padding (in samples) appended by the encoder to the end of the
audio. I.e. this number of decoded samples must be discarded by the caller from the end of the
stream to get the original audio without any trailing padding.
编码器附加到音频末尾的填充量(以样本为单位).例如调用者必须从流的末尾丢弃此数量的解码样
本,以获得原始音频,而不需要任何尾随填充
```

```
         * /
        int trailing_padding;

         /** 不连续后要跳过的样本数
         * Audio only. Number of samples to skip after a discontinuity.
         * /
        int seek_preroll;
    } AVCodecParameters;
```

2. AVCodecContext 结构体

AVCodecContext 是 FFmpeg 使用过程中比较重要的结构体，该结构体位于 libavcodec/avcodec.h 文件中，保存了编码器上下文的相关信息。不管是编码，还是解码都会用到，但在两种不同的应用场景中，结构体中部分字段的作用和说明并不一致，在使用时要特别注意。该结构体中的定义很多，下面介绍一些比较重要的字段。

(1) enum AVMediaType codec_type：编解码器的类型。

(2) const struct AVCodec * codec：编解码器，初始化后不可更改。

(3) enum AVCodecID codec_id：编解码器的 id。

(4) int64_t bit_rate：平均比特率。

(5) uint8_t * extradata；int extradata_size：针对特定编码器包含的附加信息。

(6) AVRational time_base：根据该参数可以将 pts 转换为时间。

(7) int width，height：视频帧的宽和高。

(8) int gop_size：一组图片的数量，编码时由用户设置。

(9) enum AVPixelFormat pix_fmt：像素格式，编码时由用户设置，解码时可由用户指定，但是在分析数据时会覆盖用户的设置。

(10) int refs：参考帧的数量。

(11) enum AVColorSpace colorspace：YUV 色彩空间类型。

(12) enum AVColorRange color_range：MPEG、JPEG、YUV 范围。

(13) int sample_rate：采样率，仅音频有效。

(14) int channels：声道数(音频)。

(15) enum AVSampleFormat sample_fmt：采样格式。

(16) int frame_size：每个音频帧中每个声道的采样数量。

(17) int profile：配置类型。

(18) int level：级别。

3. AVCodec 结构体

AVCodec 是存储编解码器信息的结构体，每种视频(音频)编解码器对应一个该结构体，下面先介绍几个比较重要的字段。

(1) const char * name：编解码器的名字，比较短。

(2) const char * long_name：编解码器的名字，全称，比较长。

(3) enum AVMediaType type：媒体类型，视频、音频或字幕。

(4) enum AVCodecID id：ID，不重复。

(5) const AVRational * supported_framerates：支持的帧率(仅视频)。

(6) const enum AVPixelFormat * pix_fmts：支持的像素格式(仅视频)。

(7) const int * supported_samplerates：支持的采样率(仅音频)。

(8) const enum AVSampleFormat * sample_fmts：支持的采样格式(仅音频)。

(9) const uint64_t * channel_layouts：支持的声道数(仅音频)。

(10) int priv_data_size：私有数据的大小。

AVCodec 结构体的定义位于 libavcodec/codec.h 文件中，代码如下：

```
//chapter3/3.5.help.txt
/**
 * AVCodec. 编解码器
 */
typedef struct AVCodec {
    /**
     * Name of the codec implementation.
     * The name is globally unique among encoders and among decoders (but an
     * encoder and a decoder can share the same name).
     * This is the primary way to find a codec from the user perspective.
     */
    const char *name;
    /**
     * Descriptive name for the codec, meant to be more human readable than name.
     * You should use the NULL_IF_CONFIG_SMALL() macro to define it.
     */
    const char *long_name;
    enum AVMediaType type;
    enum CodecID id;
    /**
     * Codec capabilities.
     * see CODEC_CAP_*
     */
    int capabilities;
    const AVRational *supported_framerates; //< array of supported framerates, or NULL if
any, array is terminated by {0,0}
    const enum PixelFormat *pix_fmts;  //< array of supported pixel formats, or NULL if
unknown, array is terminated by -1
    const int *supported_samplerates;  //< array of supported audio samplerates, or NULL if
unknown, array is terminated by 0
    const enum AVSampleFormat *sample_fmts; //< array of supported sample formats, or NULL if
unknown, array is terminated by -1
```

```
    const uint64_t *channel_layouts;   //< array of support channel layouts, or NULL if
unknown. array is terminated by 0
    uint8_t max_lowres;  //< maximum value for lowres supported by the decoder
    const AVClass *priv_class;  //< AVClass for the private context
    const AVProfile *profiles;  //< array of recognized profiles, or NULL if unknown, array
is terminated by {FF_PROFILE_UNKNOWN}

    /**********************************************************************
     * No fields below this line are part of the public API. They
     * may not be used outside of libavcodec and can be changed and
     * removed at will.
     * New public fields should be added right above.
     **********************************************************************
     */
    int priv_data_size;
    struct AVCodec *next;
    /**
     * @name Frame-level threading support functions
     * @{
     */
    /**
     * If defined, called on thread contexts when they are created.
     * If the codec allocates writable tables in init(), re-allocate them here.
     * priv_data will be set to a copy of the original.
     */
    int (*init_thread_copy)(AVCodecContext *);
    /**
     * Copy necessary context variables from a previous thread context to the current one.
     * If not defined, the next thread will start automatically; otherwise, the codec
     * must call ff_thread_finish_setup().
     *
     * dst and src will (rarely) point to the same context, in which case memcpy should be
skipped.
     */
    int (*update_thread_context)(AVCodecContext *dst, const AVCodecContext *src);
    /** @} */

    /**
     * Private codec-specific defaults.
     */
    const AVCodecDefault *defaults;

    /**
     * Initialize codec static data, called from avcodec_register().
     */
    void (*init_static_data)(struct AVCodec *codec);
```

```
    int ( * init)(AVCodecContext * );
    int ( * encode)(AVCodecContext * , uint8_t * buf, int buf_size, void * data);
    / * *
      * Encode data to an AVPacket.
      *
      * @param       avctx          codec context
      * @param       avpkt          output AVPacket (may contain a user - provided buffer)
      * @param[in]   frame          AVFrame containing the raw data to be encoded
      * @param[out] got_packet_ptr encoder sets to 0 or 1 to indicate that a
      *                             non - empty packet was returned in avpkt.
      * @return 0 on success, negative error code on failure
      * /
    int ( * encode2)(AVCodecContext * avctx, AVPacket * avpkt, const AVFrame * frame,
                     int * got_packet_ptr);
    int ( * decode)(AVCodecContext * , void * outdata, int * outdata_size, AVPacket * avpkt);
    int ( * close)(AVCodecContext * );
    / * *
      * Flush buffers.
      * Will be called when seeking
      * /
    void ( * flush)(AVCodecContext * );
} AVCodec;
```

4. 其他相关的枚举

FFmpeg 的编解码器类型 AVMediaType 是个枚举类型，代码如下：

```
//chapter3/3.5.help.txt
enum AVMediaType {
    AVMEDIA_TYPE_UNKNOWN =  - 1,   //< Usually treated as AVMEDIA_TYPE_DATA
    AVMEDIA_TYPE_VIDEO,            //视频
    AVMEDIA_TYPE_AUDIO,            //音频
    AVMEDIA_TYPE_DATA,             //< Opaque data information usually continuous
    AVMEDIA_TYPE_SUBTITLE,         //字幕
    AVMEDIA_TYPE_ATTACHMENT,       //< Opaque data information usually sparse
    AVMEDIA_TYPE_NB
};
```

FFmpeg 的采样格式 AVSampleFormat 是个枚举类型，代码如下：

```
//chapter3/3.5.help.txt
enum AVSampleFormat {
    AV_SAMPLE_FMT_NONE =  - 1,
    AV_SAMPLE_FMT_U8,          //< unsigned 8 bits
```

```
    AV_SAMPLE_FMT_S16,          //< signed 16 bits
    AV_SAMPLE_FMT_S32,          //< signed 32 bits
    AV_SAMPLE_FMT_FLT,          //< float
    AV_SAMPLE_FMT_DBL,          //< double

    AV_SAMPLE_FMT_U8P,          //< unsigned 8 bits, planar
    AV_SAMPLE_FMT_S16P,         //< signed 16 bits, planar
    AV_SAMPLE_FMT_S32P,         //< signed 32 bits, planar
    AV_SAMPLE_FMT_FLTP,         //< float, planar
    AV_SAMPLE_FMT_DBLP,         //< double, planar
    AV_SAMPLE_FMT_S64,          //< signed 64 bits
    AV_SAMPLE_FMT_S64P,         //< signed 64 bits, planar

    //< Number of sample formats. DO NOT USE if linking dynamically
    AV_SAMPLE_FMT_NB
};
```

FFmpeg 的编解码器 ID(AVCodecID)是个枚举类型,代码如下:

```
//chapter3/3.5.help.txt
enum AVCodecID {
    AV_CODEC_ID_NONE,

    /* video codecs */
    AV_CODEC_ID_MPEG1VIDEO,
    AV_CODEC_ID_MPEG2VIDEO, //< preferred ID for MPEG-1/2 video decoding
    AV_CODEC_ID_MPEG2VIDEO_XVMC,
    AV_CODEC_ID_H261,
    AV_CODEC_ID_H263,
    AV_CODEC_ID_RV10,
    AV_CODEC_ID_RV20,
    AV_CODEC_ID_MJPEG,
    AV_CODEC_ID_MJPEGB,
    AV_CODEC_ID_LJPEG,
    AV_CODEC_ID_SP5X,
    AV_CODEC_ID_JPEGLS,
    AV_CODEC_ID_MPEG4,
    AV_CODEC_ID_RAWVIDEO,
    AV_CODEC_ID_MSMPEG4V1,
    AV_CODEC_ID_MSMPEG4V2,
    AV_CODEC_ID_MSMPEG4V3,
    AV_CODEC_ID_WMV1,
    AV_CODEC_ID_WMV2,
    AV_CODEC_ID_H263P,
    AV_CODEC_ID_H263I,
```

```
    AV_CODEC_ID_FLV1,
    AV_CODEC_ID_SVQ1,
    AV_CODEC_ID_SVQ3,
    AV_CODEC_ID_DVVIDEO,
    AV_CODEC_ID_HUFFYUV,
    AV_CODEC_ID_CYUV,
    AV_CODEC_ID_H264,
    ...
}
```

FFmpeg 的像素格式(AVPixelFormat)是个枚举类型,代码如下:

```
//chapter3/3.5.help.txt
enum AVPixelFormat { //P 结尾代表平面模式(planar),否则是打包模式(packed)
    AV_PIX_FMT_NONE = -1,
    AV_PIX_FMT_YUV420P,   //< planar YUV 4:2:0, 12bpp, (1 Cr & Cb sample per 2x2 Y samples)
    AV_PIX_FMT_YUYV422,   //< packed YUV 4:2:2, 16bpp, Y0 Cb Y1 Cr
    AV_PIX_FMT_RGB24,     //< packed RGB 8:8:8, 24bpp, RGBRGB...
    AV_PIX_FMT_BGR24,     //< packed RGB 8:8:8, 24bpp, BGRBGR...
    AV_PIX_FMT_YUV422P,   //< planar YUV 4:2:2, 16bpp, (1 Cr & Cb sample per 2x1 Y samples)
    AV_PIX_FMT_YUV444P,   //< planar YUV 4:4:4, 24bpp, (1 Cr & Cb sample per 1x1 Y samples)
    AV_PIX_FMT_YUV410P,   //< planar YUV 4:1:0,  9bpp, (1 Cr & Cb sample per 4x4 Y samples)
    AV_PIX_FMT_YUV411P,   //< planar YUV 4:1:1, 12bpp, (1 Cr & Cb sample per 4x1 Y samples)
    AV_PIX_FMT_GRAY8,     //<       Y       ,  8bpp
    AV_PIX_FMT_MONOWHITE, //<       Y       ,  1bpp, 0 is white, 1 is black, in each Byte
pixels are ordered from the msb to the lsb
    AV_PIX_FMT_MONOBLACK, //<       Y       ,  1bpp, 0 is black, 1 is white, in each Byte
pixels are ordered from the msb to the lsb
    AV_PIX_FMT_PAL8,      //< 8 bit with PIX_FMT_RGB32 palette
    AV_PIX_FMT_YUVJ420P,  //< planar YUV 4:2:0, 12bpp, full scale (JPEG), deprecated in favor
of PIX_FMT_YUV420P and setting color_range
    AV_PIX_FMT_YUVJ422P,  //< planar YUV 4:2:2, 16bpp, full scale (JPEG), deprecated in favor
of PIX_FMT_YUV422P and setting color_range
    AV_PIX_FMT_YUVJ444P,  //< planar YUV 4:4:4, 24bpp, full scale (JPEG), deprecated in favor
of PIX_FMT_YUV444P and setting color_range
    AV_PIX_FMT_XVMC_MPEG2_MC,//< XVideo Motion Acceleration via common packet passing
    AV_PIX_FMT_XVMC_MPEG2_IDCT,
    ...
}
```

3.6 FFmpeg 的重要 API 函数

FFmpeg 提供了很多 API,使用起来非常方便,读者需要掌握八大核心模块中的常用函

数。每个模块少则十几个，多则几十个，但很难一次性掌握全部。

使用FFmpeg进行音视频解码涉及的API非常多，相对比较枯燥，各个函数的参数更加复杂。这里笔者列举出一些比较重要的API函数，如图3-18所示，后续章节会分门别类地按照模块进行详细分析与案例剖析。

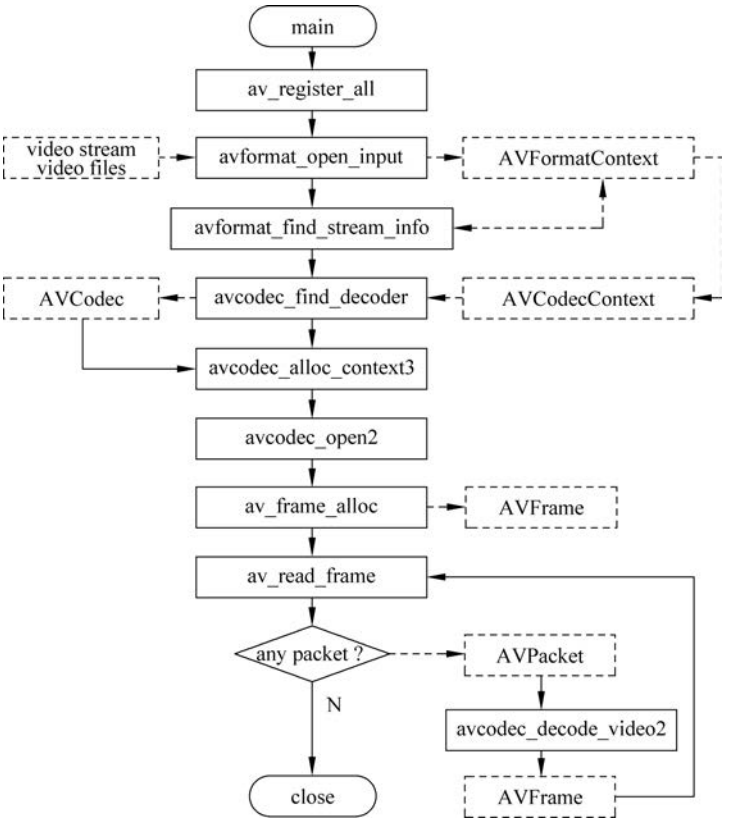

图 3-18 FFmpeg 解码相关的重要 API

图3-18中列出的这些结构体和API是针对FFmpeg 2.0版本的，虽然个别函数已经被标记为“过时的”(deprecated)，但整体流程却一直没有变化。例如FFmpeg 5.0版本中，av_register()函数已经不用再手工调用了。之前的视频解码函数avcodec_decode_video2()和avcodec_decode_audio4()音频解码函数被设置为deprecated，对这两个接口进行了合并，使用统一的接口，并且将音视频解码步骤分为两步，第1步调用avcodec_send_packet()函数发送编码数据包，第2步调用avcodec_receive_frame()函数接收解码后的数据。

1. 初始化函数

FFmpeg与初始化操作相关的主要函数如下。

(1) av_register_all()：注册所有组件，4.0版本后已经弃用。

(2) avdevice_register_all()：对设备进行注册，例如V4L2等。

(3) avformat_network_init()：初始化与网络卡及网络加密协议相关的库，例如openssl 等。

1) av_register_all()函数

使用 FFmpeg，首先要执行 av_register_all()函数(新版本中已经不需要手工调用该函数了)，把全局的解码器、编码器等结构体注册到各自全局的对象链表里，以便后面查找调用。该函数的声明如下：

```
//chapter3/3.6.help.txt
/**
 * Initialize libavformat and register all the muxers, demuxers and
 * protocols. If you do not call this function, then you can select
 * exactly which formats you want to support.
 *
 * @see av_register_input_format()
 * @see av_register_output_format()
 * attribute_deprecated:表示该函数已过时
 */
attribute_deprecated
void av_register_all(void);

attribute_deprecated
void av_register_input_format(AVInputFormat *format);
attribute_deprecated
void av_register_output_format(AVOutputFormat *format);
#endif
```

这些工作由 FFmpeg 内部去做，不需要用户调用 API 去注册。以 codec 编解码器为例，在 configure 时生成要注册的组件，会生成一个 codec_list.c 文件，里面有 static const AVCodec * const codec_list[]数组。在 libavcodec/allcodecs.c 文件中会将 static const AVCodec * const codec_list[]的编解码器用链表的方式组织起来。对于 demuxer/muxer(解复用器，也称作容器)则对应 libavformat/muxer_list.c 和 libavformat/demuxer_list.c 这两个文件，也是在 configure 时生成；在 libavformat/allformats.c 文件中将 demuxer_list[]数组和 muexr_list[]数组以链表的方式组织。其他组件也是类似的方式。

FFmpeg 中 av_register_all()函数用于注册所有 muxers、demuxers 与 protocols。FFmpeg 4.0 以前是用链表存储 muxer/demuxer，FFmpeg 4.0 以后改为数组存储，并且 av_register_all()方法已被标记为过时，av_register_input_format 和 av_register_output_format 也被标记为过时。av_register_all()的声明位于 libavformat/avformat.h 头文件中，在版本 4.0 以后，不需要调用该方法，可以直接使用所有模块。如果不调用此函数，则可以选择想要支持的那种格式，通过 av_register_input_format()和 av_register_output_format()函数实现。

2）avformat_network_init()函数

avformat_network_init()函数用于初始化网络，默认状态下 FFmpeg 是不允许联网的，必须调用该函数初始化网络后 FFmpeg 才能进行联网。如果是非 Windows 平台，则实际上什么事情都没有做，如果是 Windows 平台，则需要特别地调用 WSAStartup() 函数去初始化 Winsock 服务，这个是 Windows 平台特有的行为。函数声明如下：

```
//chapter3/3.6.help.txt
/**
 * Do global initialization of network libraries. This is optional,
 * and not recommended anymore.
 *
 * This functions only exists to work around thread-safety issues
 * with older GnuTLS or OpenSSL libraries. If libavformat is linked
 * to newer versions of those libraries, or if you do not use them,
 * calling this function is unnecessary. Otherwise, you need to call
 * this function before any other threads using them are started.
 *
 * This function will be deprecated once support for older GnuTLS and
 * OpenSSL libraries is removed, and this function has no purpose
 * anymore.
 */
int avformat_network_init(void);
```

2. 封装相关函数

FFmpeg 的封装步骤及调用的相关 API：首先使用 avformat_alloc_context()函数分配解复用器上下文，接着使用 avformat_open_input()函数根据 url 打开本地文件或网络流，然后使用 avformat_find_stream_info()函数读取媒体的部分数据包以获取码流信息，再从文件中读取数据包，主要使用 av_read_frame()函数，或定位文件：avformat_seek_file()、av_seek_frame()函数，最后使用 avformat_close_format()函数关闭解复用器。FFmpeg 与封装操作相关的主要函数如下。

（1）avformat_alloc_context()：负责申请一个 AVFormatContext 结构的内存，并进行简单初始化。

（2）avformat_free_context()：释放 AVFormatContext 结构里的所有相关内存及该结构本身。

（3）avformat_close_input()：关闭解复用器，关闭后就不再需要使用 avformat_free_context()函数进行释放。

（4）avformat_open_input()：打开输入视频文件或网络音视频流。

（5）avformat_find_stream_info()：获取音视频文件的流信息。

（6）av_read_frame()：读取音视频包。

（7）avformat_seek_file()：定位文件。

(8) av_seek_frame()：定位帧。

这里介绍几个比较重要的函数，一定要注意源码中的英文注释信息，对于理解函数功能非常有帮助。

1) avformat_alloc_context()函数

avformat_alloc_context()函数用来申请 AVFormatContext 类型变量并初始化默认参数。该函数用于分配空间并创建一个 AVFormatContext 对象，强调使用 avformat_free_context()函数来清理并释放该对象的空间。注意内存需要分配在堆上；给 AVFormatContext 的成员赋默认值；完成 AVFormatContext 内部使用对象 AVFormatInternal 结构体的空间分配及其部分成员字段的赋值。函数声明的代码如下：

```
//chapter3/3.6.help.txt
/**
 * Allocate an AVFormatContext.//:分配 AVFormatContext 空间
 * avformat_free_context() can be used to free the context and everything
 * allocated by the framework within it.
 * 需要使用 avformat_free_context()释放
 */
AVFormatContext *avformat_alloc_context(void);
```

在该函数内部，通过 av_malloc 为 AVFormatContext 分配内存空间，并且给 internal 字段分配内存，供 FFmpeg 内部使用，并设置相关字段的初始值。具体的实现代码如下：

```
//chapter3/3.6.help.txt
AVFormatContext *avformat_alloc_context(void)
{
    AVFormatContext *ic;          //创建一个 AVFormatContext 对象 ic
    //使用 av_malloc 分配空间，分配空间的作用是存储数据
    ic = av_malloc(sizeof(AVFormatContext));
    if (!ic) return ic;           //判断 ic 是否为 NULL，如果为空，则返回 ic
     //用于设置 AVFormatContext 的字段的默认值
    avformat_get_context_defaults(ic);
    //给 internal 字段分配内存，供 FFmpeg 内部使用
    ic->internal = av_mallocz(sizeof(*ic->internal));

     //判断 ic->internal 是否为 NULL，如果为 NULL，则释放上下文和内容，返回 NULL
    if (!ic->internal) {
        avformat_free_context(ic);
        return NULL;
    }
    ic->internal->offset = AV_NOPTS_VALUE;
    ic->internal->raw_packet_buffer_remaining_size = RAW_PACKET_BUFFER_SIZE;
    ic->internal->shortest_end = AV_NOPTS_VALUE;

    return ic;
}
```

可以看到，avformat_alloc_context()函数调用 av_malloc()函数为 AVFormatContext 结构体从堆上分配了内存，而且同时也给 AVFormatContext 中的 internal 字段分配了内存，供 FFmpeg 内部使用。此外还调用了一个 avformat_get_context_defaults()函数，用于设置 AVFormatContext 的字段的默认值。

2）avformat_free_context()函数

AVFormatContext 结构体的初始化函数是 avformat_alloc_context()，而销毁函数是 avformat_free_context()。avformat_free_context()函数声明在 libavformat\avformat.h 文件中，代码如下：

```
//chapter3/3.6.help.txt
/**
  * 释放 AVFormatContext 及其所有流.
  * @param s 要释放的上下文
  */
/**
 * Free an AVFormatContext and all its streams.
 * @param s context to free
 */
void avformat_free_context(AVFormatContext *s);
```

avformat_free_context()函数定义在 libavformat\utils.c 文件中，代码如下：

```
//chapter3/3.6.help.txt
void avformat_free_context(AVFormatContext *s)
{
    int i;

    if (!s)
        return;

    av_opt_free(s);
    if (s->iformat && s->iformat->priv_class && s->priv_data)
        av_opt_free(s->priv_data);
    if (s->oformat && s->oformat->priv_class && s->priv_data)
        av_opt_free(s->priv_data);

     //针对流进行释放，同时把 number 置零
    for (i = s->nb_streams - 1; i >= 0; i--)
        ff_free_stream(s, s->streams[i]);

     //针对 programs 进行释放，同时把 number 置零
    for (i = s->nb_programs - 1; i >= 0; i--) {
        av_dict_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--) {
            av_dict_free(&s->chapters[s->nb_chapters]->metadata);
            av_freep(&s->chapters[s->nb_chapters]);
        }
        av_freep(&s->chapters);
        av_dict_free(&s->metadata);
        av_dict_free(&s->internal->id3v2_meta);
        av_freep(&s->streams);
        flush_packet_queue(s);
        av_freep(&s->internal);
        av_freep(&s->url);
        av_free(s);
    }
```

可以看到，avformat_free_context()函数调用了很多销毁函数，包括 av_opt_free()、av_freep()和 av_dict_free()等，这些函数分别用于释放不同种类的变量。

3）avformat_find_streaminfo()函数

avformat_find_stream_info()函数主要用于获取媒体信息，函数声明的代码如下：

```
//chapter3/3.6.help.txt
/**
 * Read packets of a media file to get stream information. This
 * is useful for file formats with no headers such as MPEG. This
 * function also computes the real framerate in case of MPEG-2 repeat
 * frame mode.
 * 读取媒体文件的数据包以获取流信息.这对于没有标头的文件格式(如 MPEG)很有用.
 * 此函数还用于计算 MPEG-2 重复帧模式下的实际帧速率
 * The logical file position is not changed by this function;
 * examined packets may be buffered for later processing.
 *
 * @param ic media file handle
 * @param options  If non-NULL, an ic.nb_streams long array of pointers to
 *                 dictionaries, where i-th member contains options for
 *                 codec corresponding to i-th stream.
 *                 On return each dictionary will be filled with options that were not found.
 * @return >= 0 if OK, AVERROR_xxx on error
 *
 * @note this function isn't guaranteed to open all the codecs, so
 *       options being non-empty at return is a perfectly normal behavior.
 *
```

```
 * @todo Let the user decide somehow what information is needed so that
 *       we do not waste time getting stuff the user does not need.
 */
int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options);
```

该函数会从媒体文件中读取音视频包(packet,即未解码之前的音视频数据),以便获取音视频流的信息,对于没有文件头的视频格式非常好用,例如 MPEG 文件(因为视频文件经常使用文件头 header 来标注这个视频文件的各项信息,例如 muxing 格式等)。还可以用于计算帧率,例如 MPEG-2 重复帧模式下。它不会更改逻辑文件的位置,被检查的数据包可以被缓冲以供以后处理。既然这个函数的功能是更新流的信息,那么可以得知它的作用就是更新 AVStream 这个结构体中的字段。

该函数有两个参数,一个是 AVFormatContext 的上下文句柄,即这个函数实际的操作对象;还有一个是 AVDictionary 数组,如果第 2 个参数传入时不为空,则它应该是一个长度等于 ic 中包含 nb_stream 的 AVDictionary 数组的长度,第 i 个数组对应视频文件中第 i 个 stream,函数返回时,每个 dictionary 将填充没有找到的选项。

该函数的返回值为大于或等于 0 的数字,返回大于或等于 0 代表没有其他情况,其他数字代表不同的错误。该函数中不能保证 stream 中的编解码器会被打开,所以在返回之后如果 options 不为空也是正常的。

4) avformat_open_input()函数

播放一个音视频多媒体文件之前,首先要打开该文件,文件的地址类型可以是文件路径地址,也可以是网络地址。avformat_open_input()函数用于打开输入流并读取头,但并不负责打开编解码器,必须使用 avformat_close_input()关闭,返回 0 表示成功,负数为失败的错误码,函数声明如下:

```
//chapter3/3.6.help.txt
/**
 * Open an input stream and read the header. The codecs are not opened.
 * The stream must be closed with avformat_close_input().
 *
 * @param ps Pointer to user-supplied AVFormatContext (allocated by avformat_alloc_
context).
 *           May be a pointer to NULL, in which case an AVFormatContext is allocated by this
 *           function and written into ps.
 *           Note that a user-supplied AVFormatContext will be freed on failure.
 * @param url URL of the stream to open.
 * @param fmt If non-NULL, this parameter forces a specific input format.
 *            Otherwise the format is autodetected.
 * @param options  A dictionary filled with AVFormatContext and demuxer-private options.
 *                 On return this parameter will be destroyed and replaced with a dict containing
```

```
 *                  options that were not found. May be NULL.
 *
 * @return 0 on success, a negative AVERROR on failure.
 *
 * @note If you want to use custom IO, preallocate the format context and set its pb field.
 */
int avformat_open_input(AVFormatContext **ps, const char *url,
                        const AVInputFormat *fmt, AVDictionary **options);
```

该函数的参数含义如下。

(1) 第 1 个参数指向用户提供的 AVFormatContext(由 avformat_alloc_context 分配)的指针。

(2) 第 2 个参数表示要打开的流的 url。

(3) 第 3 个参数 fmt 如果非空,则此参数强制使用特定的输入格式;否则将自动检测格式。

(4) 第 4 个参数为包含 AVFormatContext 和 demuxer 私有选项的字典;返回时此参数将被销毁并替换为包含找不到的选项,如果所有项都有效,则返回空(NULL)。

5) av_read_frame()函数

对于音视频频的编解码来讲,要对数据进行解码,首先要获取视频帧的压缩数据,av_read_frame()的作用就是获取视频的数据。av_read_frame()用于获取视频的一帧,不存在半帧的说法,但可以获取音频的若干帧。av_read_frame()函数是 FFmpeg 新型的用法,旧用法之所以被抛弃,是因为以前获取的数据可能不是完整的,而新版的 av_read_frame()保证了一帧视频数据的完整性。av_read_frame()函数的作用是读取一帧视频数据或者读取多帧音频数据,读取的数据都是待解码的数据,该函数的声明如下:

```
//chapter3/3.6.help.txt
/**
 * Return the next frame of a stream.
 * This function returns what is stored in the file, and does not validate
 * that what is there are valid frames for the decoder. It will split what is
 * stored in the file into frames and return one for each call. It will not
 * omit invalid data between valid frames so as to give the decoder the maximum
 * information possible for decoding.
 *
 * On success, the returned packet is reference-counted (pkt->buf is set) and
 * valid indefinitely. The packet must be freed with av_packet_unref() when
 * it is no longer needed. For video, the packet contains exactly one frame.
 * For audio, it contains an integer number of frames if each frame has
 * a known fixed size (e.g. PCM or ADPCM data). If the audio frames have
 * a variable size (e.g. MPEG audio), then it contains one frame.
 *
```

```
 * pkt->pts, pkt->dts and pkt->duration are always set to correct * values in AVStream.
time_base units (and guessed if the format cannot
 * provide them). pkt->pts can be AV_NOPTS_VALUE if the video format
 * has B-frames, so it is better to rely on pkt->dts if you do not
 * decompress the payload.
 *
 * @return 0 if OK, < 0 on error or end of file. On error, pkt will be blank
 *         (as if it came from av_packet_alloc()).
 *
 * @note pkt will be initialized, so it may be uninitialized, but it must not
 *       contain data that needs to be freed.
 *  返回流的下一帧.
 *  此函数用于返回存储在文件中的内容,但不验证解码器是否有有效帧.
 *  它将把文件中存储的内容拆分为帧,并为每个调用返回一个帧.
 *  它不会省略有效帧之间的无效数据,以便给解码器最大可能的解码信息.
 *  如果 pkt->buf 为 NULL,则直到下一个 av_read_frame()或直到 avformat_close_input()包都是
有效的.否则数据包将无限期有效.在这两种情况下,当不再需要包时,必须使用 av_free_packet 释
放包.
 *  对于视频,数据包只包含一帧.
 *  对于音频,如果每个帧具有已知的固定大小(例如 PCM 或 ADPCM 数据),则它包含整数帧数.
 *  如果音频帧有一个可变的大小(例如 MPEG 音频),则它包含一帧.
 *  在 AVStream 中,pkt->pts、pkt->dts 和 pkt->持续时间总是被设置为恰当的值.
 *  time_base 单元(猜测格式是否不能提供它们).
 *  如果视频格式为 B-frames,pkt->pts 可以是 AV_NOPTS_VALUE,所以如果不解压缩有效负载,则
最好依赖 pkt->dts.
 */
int av_read_frame(AVFormatContext *s, AVPacket *pkt);
```

参数及返回值的说明如下。

(1) *AVFormatContext s：文件格式上下文,输入的 AVFormatContext。

(2) *AVPacket pkt：这个值不能传 NULL,必须是一个内存空间,用于存储输出的 AVPacket。

(3) 返回值：返回 0 表示成功,负数表示失败或到达文件尾(return 0 is OK,< 0 on error or end of file)。

av_read_frame()函数非常重要,其内部的操作流程如图 3-19 所示。

读出当前流数据,并存储于 AVPacket 包中；如果当前流为视频帧,则只读出一帧；如果当前流为音频帧,则根据音频格式读取固定的数据。读取成功后,流数据会自动指向下一帧。如果 AVPacket 包内存为空,表示读取失败,则读取无效,需要等待 av_read_frame()或者 avformat_close_input()函数来处理。该函数中主要调用了两个函数,如果 paketList 中有数据,则调用 ff_packet_list_get 直接从 list 中读取一帧数据,如果没有数据,则调用 read_

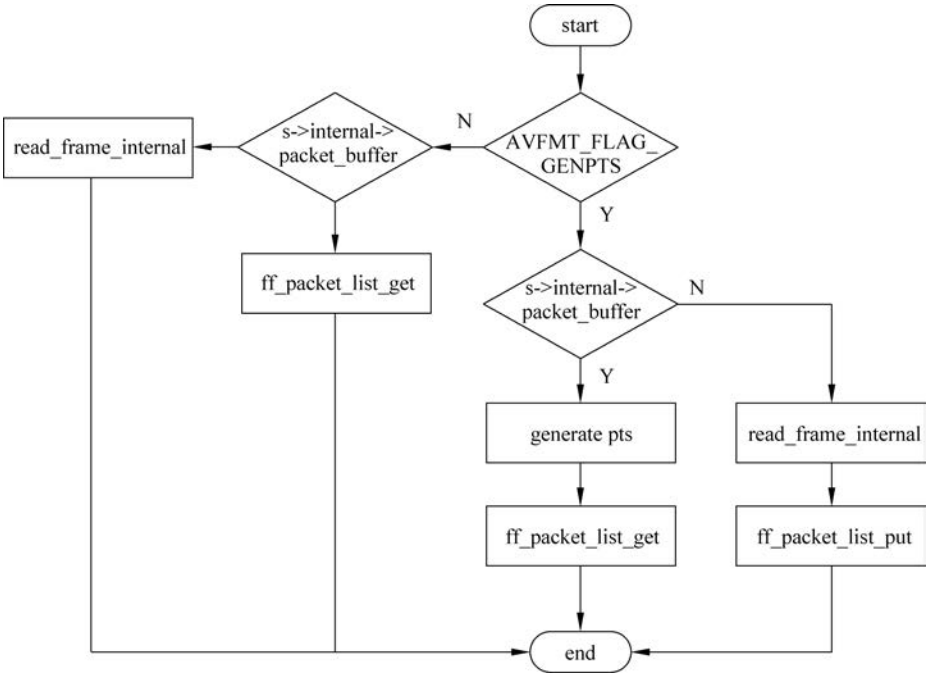

图 3-19　av_read_frame()函数的内部流程

frame_internal 重新读取一帧数据并放到 list 中。

6）avformat_close_input()函数

avformat_open_input()函数用于打开输入媒体流并读取头部信息，包括本地文件、网络流、自定义缓冲区等，而对应的释放函数为 avformat_close_input()，该函数位于 libavformat/avformat.h 文件中，函数原型如下：

```
//chapter3/3.6.help.txt
/**
 * Close an opened input AVFormatContext. Free it and all its contents
 * and set *s to NULL. 释放资源,并释放与之关联的内容,然后将*s设置为NULL
 */
void avformat_close_input(AVFormatContext **s);
```

下面查看 avformat_close_input()的源代码，位于 libavformat\utils.c 文件中，代码如下：

```
//chapter3/3.6.help.txt
void avformat_close_input(AVFormatContext **ps)
{
    AVFormatContext *s;
    AVIOContext *pb;
```

```
    if (!ps || !*ps)
        return;

    s  = *ps;
    pb = s->pb;

    if ((s->iformat && strcmp(s->iformat->name, "image2") && s->iformat->flags &
AVFMT_NOFILE) ||
        (s->flags & AVFMT_FLAG_CUSTOM_IO))
        pb = NULL;

    flush_packet_queue(s);

    if (s->iformat)
        if (s->iformat->read_close)
            s->iformat->read_close(s);

    avformat_free_context(s);

    *ps = NULL;

    avio_close(pb);
}
```

从源代码中可以看出，avformat_close_input()主要做了以下几步工作：

(1) 调用 AVInputFormat 的 read_close()函数关闭输入流。

(2) 调用 avformat_free_context()释放 AVFormatContext。

(3) 调用 avio_close()关闭并且释放 AVIOContext。

AVInputFormat 的 read_close()是一个函数指针，指向关闭输入流的函数。不同的 AVInputFormat 包含不同的 read_close()方法。例如，FLV 格式对应的 AVInputFormat 的定义如下：

```
//chapter3/3.6.help.txt
AVInputFormat ff_flv_demuxer = {
    .name           = "flv",
    .long_name      = NULL_IF_CONFIG_SMALL("FLV (Flash Video)"),
    .priv_data_size = sizeof(FLVContext),
    .read_probe     = flv_probe,
    .read_header    = flv_read_header,
    .read_packet    = flv_read_packet,
    .read_seek      = flv_read_seek,
    .read_close     = flv_read_close,
```

```
    .extensions     = "flv",
    .priv_class     = &flv_class,
};
```

从 ff_flv_demuxer 的定义中可以看出，read_close()指向的函数是 flv_read_close()函数，代码如下：

```
//chapter3/3.6.help.txt
static int flv_read_close(AVFormatContext * s)
{
    int i;
    FLVContext * flv = s->priv_data;
    for (i = 0; i < FLV_STREAM_TYPE_NB; i++)
        av_freep(&flv->new_extradata[i]);
    return 0;
}
```

从 flv_read_close()的定义可以看出，该函数释放了 FLVContext 中的 new_extradata 数组中每个元素指向的内存。

avformat_free_context()是一个 FFmpeg 的 API 函数，用于释放一个 AVFormatContext。

avio_close()是一个 FFmpeg 的 API 函数，用于关闭和释放 AVIOContext。它的声明位于 libavformat/avio.h 文件，代码如下：

```
//chapter3/3.6.help.txt
/**
 * Close the resource accessed by the AVIOContext s and free it.
 * This function can only be used if s was opened by avio_open().
 * 只有在使用 avio_open()函数打开的情况，才可以调用 avio_close()函数
 * The internal buffer is automatically flushed before closing the
 * resource. 内部缓冲区在关闭资源前被自动冲刷
 *
 * @return 0 on success, an AVERROR < 0 on error.
 * @see avio_closep
 */
int avio_close(AVIOContext * s);
```

avio_close()函数的定义位于 libavformat/aviobuf.c 文件，代码如下：

```
//chapter3/3.6.help.txt
int avio_close(AVIOContext * s)
{
    AVIOInternal * internal;
```

```
    URLContext *h;

    if (!s)
        return 0;

    avio_flush(s);
    internal = s->opaque;
    h        = internal->h;

    av_freep(&s->opaque);
    av_freep(&s->buffer);
    if (s->write_flag)
        av_log(s, AV_LOG_VERBOSE, "Statistics: %d seeks, %d writeouts\n", s->seek_count,
s->writeout_count);
    else
        av_log(s, AV_LOG_VERBOSE, "Statistics: %"PRId64" Bytes read, %d seeks\n", s->
Bytes_read, s->seek_count);
    av_opt_free(s);

    avio_context_free(&s);

    return ffurl_close(h);
}
```

从源代码可以看出，avio_close()函数按照顺序执行了以下几步：

(1) 调用 avio_flush()函数强制清除缓存中的数据。

(2) 调用 av_freep()函数释放 AVIOContext 中的 buffer。

(3) 调用 avio_context_free()函数释放 AVIOContext 结构体。

(4) 调用 ffurl_close()函数关闭并且释放 URLContext。

ffurl_close()和 ffurl_closep()是 FFmpeg 内部的两个函数，它们的声明位于 libavformat/url.h 文件，代码如下：

```
//chapter3/3.6.help.txt
/**
 * Close the resource accessed by the URLContext h, and free the
 * memory used by it. Also set the URLContext pointer to NULL.
 *
 * @return a negative value if an error condition occurred, 0
 * otherwise
 */
int ffurl_closep(URLContext **h);
int ffurl_close(URLContext *h);
```

其实这两个函数是等同的，ffurl_close()函数的定义位于 libavformat/avio.c 文件，代

码如下：

```
//chapter3/3.6.help.txt
int ffurl_close(URLContext * h)
{
    return ffurl_closep(&h);
}
```

可见 ffurl_close()函数调用了 ffurl_closep()函数，代码如下：

```
//chapter3/3.6.help.txt
int ffurl_closep(URLContext ** hh)
{
    URLContext * h = * hh;
    int ret = 0;
    if (!h)
        return 0;     /* can happen when ffurl_open fails */

    if (h->is_connected && h->prot->url_close)
        ret = h->prot->url_close(h);
#if CONFIG_NETWORK
    if (h->prot->flags & URL_PROTOCOL_FLAG_NETWORK)
        ff_network_close();
#endif
    if (h->prot->priv_data_size) {
        if (h->prot->priv_data_class)
            av_opt_free(h->priv_data);
        av_freep(&h->priv_data);
    }
    av_freep(hh);
    return ret;
}
```

可以看出，它主要完成了两步工作：

(1) 调用 URLProtocol 的 url_close()函数。

(2) 调用 av_freep()函数释放 URLContext 结构体。

例如文件协议的结构体变量的定义如下：

```
//chapter3/3.6.help.txt
URLProtocol ff_file_protocol = {
    .name                = "file",
    .url_open            = file_open,
    .url_read            = file_read,
    .url_write           = file_write,
```

```
    .url_seek             = file_seek,
    .url_close            = file_close,
    .url_get_file_handle = file_get_handle,
    .url_check            = file_check,
    .priv_data_size       = sizeof(FileContext),
    .priv_data_class      = &file_class,
};
```

从 ff_file_protocol 中可以看出，url_close()函数指向了 file_close()函数，而 file_close()函数的定义，代码如下：

```
//chapter3/3.6.help.txt
static int file_close(URLContext * h)
{
    FileContext * c = h->priv_data;
    return close(c->fd);
}
```

由此可见，file_close()函数最终调用了系统函数 close()关闭了文件指针。至此 avio_close()函数的流程执行完毕。

3. 解码器相关函数

使用 FFmepg 进行解码的主要相关步骤：首先使用 avcodec_alloc_context3()函数分配编解码上下文，接着使用 avcode_parameter_to_context()函数将码流中的编解码信息复制到编解码器上下文结构体(AVCodecContex)中，然后根据编解码器信息使用 avcodec_find_decoder()函数查找相应的解码器或使用 avcodec_find_decoder_by_name()函数根据解码器名称查找指定的解码器，再使用 avcodec_open2()函数打开解码器并关联到 AVCodecContex，然后使用 avcodec_send_packet()函数向解码器发送数据包或使用 avcodec_receive_frame()函数接收解码后的帧，最后使用 avcodec_close()函数和 avcodec_free_context()函数关闭解码器并释放上下文。FFmpeg 与解码操作相关的主要函数如下。

(1) avcodec_alloc_context3()：分配解码器上下文。

(2) avcodec_find_decoder()：根据 ID 查找解码器。

(3) avcodec_find_decoder_by_name()：根据解码器名字查找解码器。

(4) avcodec_open2()：打开编解码器。

(5) avcodec_decode_video2()：解码一帧视频数据。

(6) avcodec_decode_audio4()：解码一帧音频数据。

(7) avcodec_send_packet()：发送编码数据包。

(8) avcodec_receive_frame()：接收解码后数据。

(9) avcodec_free_context()：释放解码器上下文，包含了 avcodec_close()函数。

(10) avcodec_close()：关闭解码器。

1）avcodec_alloc_context3()函数

avcodec_alloc_context3()函数用于分配解码器上下文，并将其字段设置为默认值，应使用 avcodec_free_context()函数释放相应的资源，函数声明如下：

```
//chapter3/3.6.help.txt
/**
 * Allocate an AVCodecContext and set its fields to default values. The
 * resulting struct should be freed with avcodec_free_context().
 *
 * @param codec if non-NULL, allocate private data and initialize defaults
 *                for the given codec. It is illegal to then call avcodec_open2()
 *                with a different codec.
 *                If NULL, then the codec-specific defaults won't be initialized,
 *                which may result in suboptimal default settings (this is
 *                important mainly for encoders, e.g. libx264).
 *
 * @return An AVCodecContext filled with default values or NULL on failure.
 */
AVCodecContext * avcodec_alloc_context3(const AVCodec * codec);
```

参数 codec 如果非 NULL，则对于给定的编解码器分配私有数据并初始化默认值，然后使用不同的编解码器调用 avcodec_open2()函数是非法的。如果为 NULL，则不会初始化特定于编解码器的默认值，这可能会导致次优的默认设置（这对于一些编码器非常重要，例如 libx264）。成功时返回用默认值填充的 AVCodecContext，失败时返回 NULL。

2）avcodec_open2()函数

avcodec_open2()函数用于打开编解码器，并将编码器上下文和编码器进行关联，函数声明如下：

```
//chapter3/3.6.help.txt
/**
 * Initialize the AVCodecContext to use the given AVCodec. Prior to using this
 * function the context has to be allocated with avcodec_alloc_context3().
 * *
 * @warning This function is not thread safe! :注意该函数非线程安全
 *
 * @note Always call this function before using decoding routines (such as
 * @ref avcodec_receive_frame()).
 * @param avctx The context to initialize.
 * @param codec The codec to open this context for. If a non-NULL codec has been previously
passed to avcodec_alloc_context3() or for this context, then this parameter MUST be either NULL
or equal to the previously passed codec.
 * @param options A dictionary filled with AVCodecContext and codec-private options. On
return this object will be filled with options that were not found.
```

```
 *
 * @return zero on success, a negative value on error
 * @see avcodec_alloc_context3(), avcodec_find_decoder(), avcodec_find_encoder(),
 *       av_dict_set(), av_opt_find().
 */
int avcodec_open2(AVCodecContext *avctx, const AVCodec *codec, AVDictionary **options);
```

各个参数的含义如下。

(1) avctx：需要初始化的 AVCodecContext。

(2) codec：输入的 AVCodec。

(3) options：一些编码器选项。例如使用 libx264 编码时，preset、tune 等都可以通过该参数设置。

avcodec_open2()的源代码量非常大，但是它的调用关系非常简单，只调用了一个关键的函数，即 AVCodec 的 init()，用于初始化编解码器，它所做的工作如下所述：

(1) 为各种结构体分配内存(通过各种 av_malloc()实现)。

(2) 将输入的 AVDictionary 形式的选项设置到 AVCodecContext。

(3) 其他检查，例如检查编解码器是否处于“实验”阶段。

(4) 如果是编码器，则检查输入参数是否符合编码器的要求。

(5) 调用 AVCodec 的 init() 初始化具体的解码器。

3) avcodec_find_decoder()函数

FFmpeg 提供了两种方式查找解码器：通过 codecId 查找 avcodec_find_decoder()与通过名字查找 avcodec_find_decoder_by_name()。同样地，也提供了两种方式查找编码器：通过 codecId 查找 avcodec_find_encoder()与通过名字查找 avcodec_find_encoder_by_name()。这两个函数声明的代码如下：

```
//chapter3/3.6.help.txt
/**
 * Find a registered decoder with a matching codec ID.
 * 根据 codecID 来查找一个注册过的解码器
 * @param id AVCodecID of the requested decoder
 * @return A decoder if one was found, NULL otherwise.
 */
const AVCodec *avcodec_find_decoder(enum AVCodecID id);

/**
 * Find a registered decoder with the specified name.
 * 根据给定的名称来查找一个注册过的解码器
 * @param name name of the requested decoder
 * @return A decoder if one was found, NULL otherwise.
 */
const AVCodec *avcodec_find_decoder_by_name(const char *name);
```

avcodec_find_decoder()函数通过 AVCodecID 查找对应的解码器，如果未查找到，则返回 NULL；参数 id 是 AVCodecID 类型，表明要查找解码器的 id。avcodec_find_encoder_by_name()函数通过解码器的名称找到解码器，如果查找成功，则返回解码器信息，如果未找到，则返回 NULL；参数 name 是编解码器的名称。

函数源码位于 libavcodec/allcodecs.c 文件中，查找编解码器的过程如图 3-20 所示。

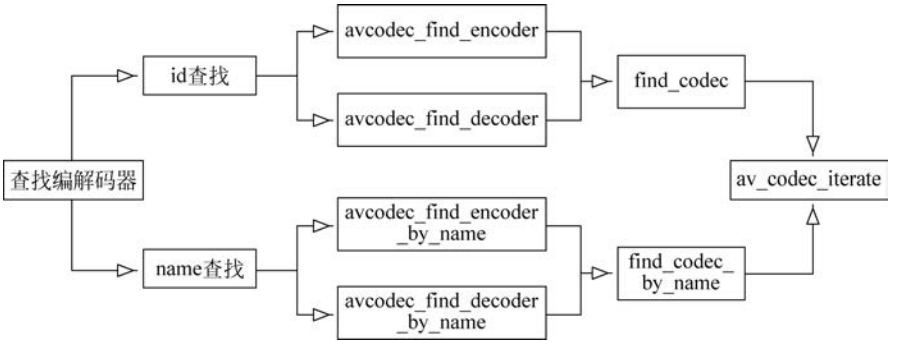

图 3-20　avcodec_find_decoder()函数的查找过程

avcodec_find_decoder()函数通过 codecId 来查找解码器，该函数的实现代码如下：

```
//chapter3/3.6.help.txt
AVCodec * avcodec_find_decoder(enum AVCodecID id)
{
    return find_codec(id, av_codec_is_decoder);
}
```

函数体只有一行代码，即调用 find_codec()函数来查找，第 1 个参数为 codecId，第 2 个参数是 int 类型，表示是否属于解码器类型，具体的代码如下：

```
//chapter3/3.6.help.txt
static AVCodec * find_codec(enum AVCodecID id, int ( * x)(const AVCodec * ))
{
    const AVCodec * p, * experimental = NULL;
    void * i = 0;

    id = remap_deprecated_codec_id(id);

    while ((p = av_codec_iterate(&i))) {
        if (!x(p))
            continue;
        if (p->id == id) {
            if (p->capabilities & AV_CODEC_CAP_EXPERIMENTAL && !experimental) {
                experimental = p;
```

```
            } else
                return (AVCodec * )p;
        }
    }

    return (AVCodec * )experimental;
}
```

可以看到，该函数主要通过 while 循环调用 av_codec_iterate()来迭代遍历，获取 AVCodec，然后用 id 判断是否相等。如果 id 相等，就返回对应的 AVCodec。下面来看 av_codec_iterate()函数的代码实现：

```
//chapter3/3.6.help.txt
AVCodec * codec_list[] = {
    NULL,
    NULL,
    NULL
};
static void av_codec_init_static(void)
{
    for (int i = 0; codec_list[i]; i++) {
        if (codec_list[i]->init_static_data)
            codec_list[i]->init_static_data((AVCodec * )codec_list[i]);
    }
}

const AVCodec * av_codec_iterate(void ** opaque)
{
    uintptr_t i = (uintptr_t) * opaque;
    const AVCodec * c = codec_list[i];

    ff_thread_once(&av_codec_static_init, av_codec_init_static);

    if (c)
        * opaque = (void * )(i + 1);

    return c;
}
```

该函数主要根据传入的 id，从 codec_list 编解码器列表中查找对应的项。

avcodec_find_decoder_by_name()函数通过名字查找解码器，代码如下：

```
//chapter3/3.6.help.txt
AVCodec * avcodec_find_decoder_by_name(const char * name)
{
    return find_codec_by_name(name, av_codec_is_decoder);
}
```

可以看到，主要调用了 find_codec_by_name()函数，第 1 个参数为解码器名字，第 2 个参数是 int 类型，表示是否属于解码器类型，具体方法如下：

```
//chapter3/3.6.help.txt
static AVCodec * find_codec_by_name(const char * name, int ( * x)(const AVCodec * ))
{
    void * i = 0;
    const AVCodec * p;

    if (!name)
        return NULL;

    while ((p = av_codec_iterate(&i))) {
        if (!x(p))
            continue;
        if (strcmp(name, p->name) == 0)
            return (AVCodec * )p;
    }

    return NULL;
}
```

可以看到，主要通过 while 循环调用 av_codec_iterate()函数来迭代遍历，获取 AVCodec，然后用名字进行比较。如果匹配成功，就返回对应的 AVCodec；否则返回 NULL。

上述是查找解码器的函数及其实现原理，查找编码器的方式与此类似，函数是 avcodec_find_encoder()和 avcodec_find_encoder_by_name()，实现原理完全相同，这两个函数声明的代码如下：

```
//chapter3/3.6.help.txt
/**
 * Find a registered encoder with a matching codec ID.
 * 根据 codecID 来查找一个注册过的编码器
 * @param id AVCodecID of the requested encoder
 * @return An encoder if one was found, NULL otherwise.
 */
const AVCodec * avcodec_find_encoder(enum AVCodecID id);
```

```
/**
 * Find a registered encoder with the specified name.
 * 根据给定的名称来查找一个注册过的编码器
 * @param name name of the requested encoder
 * @return An encoder if one was found, NULL otherwise.
 */
const AVCodec *avcodec_find_encoder_by_name(const char *name);
```

4）avcodec_free_context()函数

avcodec_alloc_context3()函数为 AVCodecContext 分配内存空间，并设置默认的值，释放需要使用 avcodec_free_context()函数，函数声明的代码如下：

```
//chapter3/3.6.help.txt
/**
 * Free the codec context and everything associated with it
 * and write NULL to the provided pointer.
 * 释放 codec context 和与之关联的 everything，并将指针置为 NULL
 */
void avcodec_free_context(AVCodecContext **avctx);
```

该函数释放 AVFormatContext 结构里的所有东西及该结构本身，并将指针置为空。

5）编解码器列表及 id

编解码器列表来自 libavcodec/allcodecs.c 文件，含有声明全局变量，这些变量都是以 ff_开头的，中间是编解码类型（如 h264、aac、wmv、pcm、vp9 等），以_encoder 结尾表示的是编码器，以_decoder 结尾表示的是解码器，代码如下：

```
//chapter3/3.6.help.txt
extern AVCodec ff_flv_encoder;
extern AVCodec ff_flv_decoder;
extern AVCodec ff_h264_decoder;
extern AVCodec ff_hevc_decoder;
extern AVCodec ff_mpeg4_encoder;
extern AVCodec ff_mpeg4_decoder;
extern AVCodec ff_msmpeg4v1_decoder;
extern AVCodec ff_msmpeg4v2_encoder;
extern AVCodec ff_msmpeg4v2_decoder;
extern AVCodec ff_msmpeg4v3_encoder;
extern AVCodec ff_msmpeg4v3_decoder;
extern AVCodec ff_vp8_decoder;
extern AVCodec ff_vp9_decoder;
extern AVCodec ff_wmv1_encoder;
extern AVCodec ff_wmv1_decoder;
```

```
extern AVCodec ff_wmv2_encoder;
extern AVCodec ff_wmv2_decoder;
extern AVCodec ff_wmv3_decoder;

/* audio codecs */
extern AVCodec ff_aac_encoder;
extern AVCodec ff_aac_decoder;
extern AVCodec ff_ac3_encoder;
extern AVCodec ff_ac3_decoder;
extern AVCodec ff_amrnb_decoder;
extern AVCodec ff_amrwb_decoder;
extern AVCodec ff_ape_decoder;
extern AVCodec ff_eac3_encoder;
extern AVCodec ff_eac3_decoder;
extern AVCodec ff_flac_encoder;
extern AVCodec ff_flac_decoder;
extern AVCodec ff_mp3_decoder;
extern AVCodec ff_opus_encoder;
extern AVCodec ff_opus_decoder;
extern AVCodec ff_truehd_encoder;
extern AVCodec ff_truehd_decoder;
extern AVCodec ff_vorbis_encoder;
extern AVCodec ff_vorbis_decoder;
extern AVCodec ff_wmav1_encoder;
extern AVCodec ff_wmav1_decoder;
extern AVCodec ff_wmav2_encoder;
extern AVCodec ff_wmav2_decoder;

/* PCM codecs */
extern AVCodec ff_pcm_alaw_encoder;
extern AVCodec ff_pcm_alaw_decoder;
extern AVCodec ff_pcm_mulaw_encoder;
extern AVCodec ff_pcm_mulaw_decoder;
extern AVCodec ff_pcm_s8_encoder;
extern AVCodec ff_pcm_s8_decoder;
extern AVCodec ff_pcm_s16le_encoder;
extern AVCodec ff_pcm_s16le_decoder;
extern AVCodec ff_pcm_s16le_planar_encoder;
extern AVCodec ff_pcm_s16le_planar_decoder;
extern AVCodec ff_pcm_s24le_encoder;
extern AVCodec ff_pcm_s24le_decoder;
extern AVCodec ff_pcm_s32le_encoder;
extern AVCodec ff_pcm_s32le_decoder;
extern AVCodec ff_pcm_s64le_encoder;
extern AVCodec ff_pcm_s64le_decoder;
extern AVCodec ff_pcm_u8_encoder;
```

```
extern AVCodec ff_pcm_u8_decoder;

/* ADPCM codecs */
extern AVCodec ff_adpcm_ima_wav_encoder;
extern AVCodec ff_adpcm_ima_wav_decoder;
extern AVCodec ff_adpcm_ms_encoder;
extern AVCodec ff_adpcm_ms_decoder;

/* subtitles */
extern AVCodec ff_ssa_encoder;
extern AVCodec ff_ssa_decoder;
extern AVCodec ff_ass_encoder;
extern AVCodec ff_ass_decoder;
extern AVCodec ff_movtext_encoder;
extern AVCodec ff_movtext_decoder;
extern AVCodec ff_pgssub_decoder;
extern AVCodec ff_sami_decoder;
extern AVCodec ff_srt_encoder;
extern AVCodec ff_srt_decoder;
extern AVCodec ff_subrip_encoder;
extern AVCodec ff_subrip_decoder;
extern AVCodec ff_ttml_encoder;
extern AVCodec ff_webvtt_encoder;
extern AVCodec ff_webvtt_decoder;

/* external libraries */
extern AVCodec ff_aac_at_encoder;
extern AVCodec ff_aac_at_decoder;
extern AVCodec ff_ac3_at_decoder;
extern AVCodec ff_amr_nb_at_decoder;
extern AVCodec ff_eac3_at_decoder;
extern AVCodec ff_mp3_at_decoder;
extern AVCodec ff_libaom_av1_encoder;
extern AVCodec ff_libdav1d_decoder;
extern AVCodec ff_libfdk_aac_encoder;
extern AVCodec ff_libfdk_aac_decoder;
extern AVCodec ff_libmp3lame_encoder;
extern AVCodec ff_libopencore_amrnb_decoder;
extern AVCodec ff_libopencore_amrwb_decoder;
extern AVCodec ff_libopus_encoder;
extern AVCodec ff_libopus_decoder;
extern AVCodec ff_libshine_encoder;
extern AVCodec ff_libspeex_encoder;
extern AVCodec ff_libspeex_decoder;
extern AVCodec ff_libvorbis_encoder;
extern AVCodec ff_libvorbis_decoder;
```

```
extern AVCodec ff_libvpx_vp8_encoder;
extern AVCodec ff_libvpx_vp8_decoder;
extern AVCodec ff_libvpx_vp9_encoder;
extern AVCodec ff_libvpx_vp9_decoder;
extern AVCodec ff_libwebp_encoder;
extern AVCodec ff_libx264_encoder;
extern AVCodec ff_libx265_encoder;

/* hwaccel hooks only, so prefer external decoders */
extern AVCodec ff_av1_decoder;
extern AVCodec ff_av1_cuvid_decoder;
extern AVCodec ff_av1_qsv_decoder;
extern AVCodec ff_libopenh264_encoder;
extern AVCodec ff_libopenh264_decoder;
extern AVCodec ff_h264_cuvid_decoder;
extern AVCodec ff_h264_nvenc_encoder;
extern AVCodec ff_h264_omx_encoder;
extern AVCodec ff_h264_qsv_encoder;
extern AVCodec ff_h264_v4l2m2m_encoder;
extern AVCodec ff_h264_vaapi_encoder;
extern AVCodec ff_h264_videotoolbox_encoder;
extern AVCodec ff_hevc_mediacodec_decoder;
extern AVCodec ff_hevc_nvenc_encoder;
extern AVCodec ff_hevc_qsv_encoder;
extern AVCodec ff_hevc_v4l2m2m_encoder;
extern AVCodec ff_hevc_vaapi_encoder;
extern AVCodec ff_hevc_videotoolbox_encoder;
extern AVCodec ff_mp3_mf_encoder;
extern AVCodec ff_mpeg4_cuvid_decoder;
extern AVCodec ff_mpeg4_mediacodec_decoder;
extern AVCodec ff_mpeg4_omx_encoder;
extern AVCodec ff_mpeg4_v4l2m2m_encoder;
extern AVCodec ff_vp9_cuvid_decoder;
extern AVCodec ff_vp9_mediacodec_decoder;
extern AVCodec ff_vp9_qsv_decoder;
extern AVCodec ff_vp9_vaapi_encoder;
extern AVCodec ff_vp9_qsv_encoder;
```

编解码器 id 是个枚举类型 AVCodecId，其定义位于 libavcodec/codec_id.h 头文件中，原文件中有几百个枚举项，笔者这里只列出一些重要的枚举项，代码如下：

```
//chapter3/3.6.help.txt
enum AVCodecID {
    AV_CODEC_ID_NONE,
```

```
    /* video codecs */
    AV_CODEC_ID_MPEG1VIDEO,
    AV_CODEC_ID_MPEG2VIDEO, //< preferred ID for MPEG-1/2 video decoding
    AV_CODEC_ID_MPEG4,
    AV_CODEC_ID_MSMPEG4V1,
    AV_CODEC_ID_MSMPEG4V2,
    AV_CODEC_ID_MSMPEG4V3,
    AV_CODEC_ID_WMV1,
    AV_CODEC_ID_WMV2,
    AV_CODEC_ID_FLV1,
    AV_CODEC_ID_H264,
    AV_CODEC_ID_PNG,
    AV_CODEC_ID_VC1,
    AV_CODEC_ID_WMV3,
    AV_CODEC_ID_AVS,
    AV_CODEC_ID_JPEG2000,
    AV_CODEC_ID_GIF,
    AV_CODEC_ID_VP8,
    AV_CODEC_ID_VP9,
    AV_CODEC_ID_HEVC,
#define AV_CODEC_ID_H265 AV_CODEC_ID_HEVC
    AV_CODEC_ID_VVC,
#define AV_CODEC_ID_H266 AV_CODEC_ID_VVC
    AV_CODEC_ID_AV1,

    /* various PCM "codecs" */
    AV_CODEC_ID_PCM_S16LE = 0x10000,
    AV_CODEC_ID_PCM_S16BE,
    AV_CODEC_ID_PCM_S8,
    AV_CODEC_ID_PCM_U8,
    AV_CODEC_ID_PCM_MULAW,
    AV_CODEC_ID_PCM_ALAW,
    AV_CODEC_ID_PCM_S32LE,
    AV_CODEC_ID_PCM_S24LE,
    AV_CODEC_ID_PCM_S24DAUD,
    AV_CODEC_ID_PCM_S16LE_PLANAR,
    AV_CODEC_ID_PCM_F32LE,
    AV_CODEC_ID_PCM_F64LE,
    AV_CODEC_ID_PCM_S8_PLANAR,
    AV_CODEC_ID_PCM_S24LE_PLANAR,
    AV_CODEC_ID_PCM_S32LE_PLANAR,
    AV_CODEC_ID_PCM_S64LE = 0x10800,
    AV_CODEC_ID_PCM_F16LE,
    AV_CODEC_ID_PCM_F24LE,
```

```
    /* various ADPCM codecs */
    AV_CODEC_ID_ADPCM_IMA_QT = 0x11000,
    AV_CODEC_ID_ADPCM_IMA_WAV,
    AV_CODEC_ID_ADPCM_MS,

    AV_CODEC_ID_ADPCM_AFC = 0x11800,
    AV_CODEC_ID_ADPCM_IMA_OKI,

    /* AMR */
    AV_CODEC_ID_AMR_NB = 0x12000,
    AV_CODEC_ID_AMR_WB,

    /* audio codecs */
    AV_CODEC_ID_MP2 = 0x15000,
    AV_CODEC_ID_MP3, //preferred ID for decoding MPEG audio layer 1, 2 or 3
    AV_CODEC_ID_AAC,
    AV_CODEC_ID_AC3,
    AV_CODEC_ID_VORBIS,
    AV_CODEC_ID_WMAV1,
    AV_CODEC_ID_WMAV2,
    AV_CODEC_ID_FLAC,
    AV_CODEC_ID_APE,
    AV_CODEC_ID_EAC3,
    AV_CODEC_ID_TRUEHD,
    AV_CODEC_ID_OPUS,

    /* subtitle codecs */
    AV_CODEC_ID_DVD_SUBTITLE = 0x17000,
    AV_CODEC_ID_DVB_SUBTITLE,
    AV_CODEC_ID_SSA,
    AV_CODEC_ID_MOV_TEXT,
    AV_CODEC_ID_SRT,

    AV_CODEC_ID_MICRODVD   = 0x17800,
    AV_CODEC_ID_SAMI,
    AV_CODEC_ID_SUBRIP,
    AV_CODEC_ID_WEBVTT,
    AV_CODEC_ID_ASS,
    AV_CODEC_ID_TTML,

    AV_CODEC_ID_TTF = 0x18000,
};
```

6）核心解码函数

FFmpeg 中提供了 avcodec_send_frame()和 avcodec_receive_packet()函数，这两个函

数用于编码，此外，avcodec_send_packet()和 avcodec_receive_frame()函数用于解码。

查到了对应的编解码器之后，通过 avcodec_open2()函数打开编解码器，然后就可以做真正的编解码工作了，例如解码包括几个相关的函数：avcodec_decode_video2()、avcodec_decode_audio4()、avcodec_send_packet()和 avcodec_receive_frame()。

这些函数在 FFmpeg 的 libavcodec 模块中，旧版本提供了 avcodec_decode_video2()作为视频解码函数，avcodec_decode_audio4()作为音频解码函数。在 FFmpeg 3.1 版本新增了 avcodec_send_packet()与 avcodec_receive_frame()作为音视频解码函数。后来，在 3.4 版本把 avcodec_decode_video2()和 avcodec_decode_audio4()标记为过时 API，版本变更描述信息，代码如下：

```
//chapter3/3.6.help.txt
//FFmpeg 3.1
2016-04-21 - 7fc329e - lavc 57.37.100 - avcodec.h
Add a new audio/video encoding and decoding API with decoupled input
and output -- avcodec_send_packet(), avcodec_receive_frame(),
avcodec_send_frame() and avcodec_receive_packet().

//FFmpeg 3.4
2017-09-26 - b1cf151c4d - lavc 57.106.102 - avcodec.h
Deprecate AVCodecContext.refcounted_frames. This was useful for deprecated
API only (avcodec_decode_video2/avcodec_decode_audio4). The new decode APIs
(avcodec_send_packet/avcodec_receive_frame) always work with reference
counted frames.
```

视频解码接口 avcodec_decode_video2 和 avcodec_decode_audio4 音频解码在新版本中被标记为过时的(deprecated)，对两个接口进行了合并，使用统一的 API。将音视频解码步骤分为以下两步。

(1) avcodec_send_packet()：发送编码数据包。

(2) avcodec_receive_frame()：接收解码后的数据。

avcodec_send_packet 的函数原型，代码如下：

```
int avcodec_send_packet(AVCodecContext *avctx, const AVPacket *avpkt);
```

参数说明如下。

(1) AVCodecContext *avctx：视频解码的上下文信息，包含解码器。

(2) const AVPacket *avpkt：编码的音视频帧数据。

(3) 返回值：如果成功，则返回 0，否则返回错误码(负数)。

通常情况下，在解码的最后环节，需要将空指针(NULL)传递给 avpkt 参数。例如平时看一些视频，播放时经常会少最后几帧，很多情况就是因为没有处理好缓冲帧的问题。FFmpeg 内部会缓冲几帧，要想取出来就需要将空的 AVPacket(NULL)传递进去。

avcodec_receive_frame 的函数原型，代码如下：

```
int avcodec_receive_frame(AVCodecContext *avctx, AVFrame *frame);
```

参数说明如下。

(1) AVCodecContext *avctx：视频解码的上下文，包含解码器。

(2) AVFrame *frame：解码后的音视频帧数据。

(3) 返回值：如果成功，则返回 0，否则返回错误码(负数)。

需要注意的是，解码后的图像空间由函数内部申请，程序员只需分配 AVFrame 对象空间。如果每次调用 avcodec_receive_frame()函数都传递同一个对象，则接口内部会判断空间是否已经分配，如果没有分配，则会在函数内部为图片分配内存空间。

avcodec_send_packet()和 avcodec_receive_frame()的函数调用关系并不一定是一对一的，例如一些音频数据的一个 AVPacket 中包含了 1s 的音频，调用一次 avcodec_send_packet()函数之后，可能需要调用 25 次 avcodec_receive_frame()函数才能获取全部的解码音频数据，所以要做如下处理，伪代码如下：

```
//chapter3/3.6.help.txt
int ret = avcodec_send_packet(codec, pkt);
if (ret != 0)
{
    return;
}

while( avcodec_receive_frame(codec, frame) == 0)
{
    //读取一帧音频或者视频
    //处理解码后的音视频 frame
}
```

avcodec_send_packet()和 avcodec_receive_frame()函数的声明位于 libavcodec/avcodec.h 文件中，函数声明的代码如下：

```
//chapter3/3.6.help.txt
/**
 * Supply raw packet data as input to a decoder.
 * 给解码器提供原始的音视频包
 * @param avctx codec context :编解码器上下文
 * @param[in] avpkt The input AVPacket. Usually, this will be a single video
 *                  frame, or several complete audio frames.
 * 音视频包,通常是一个完整的视频帧或几个完整的音频帧
 * @return 0 on success, otherwise negative error code:
 * 如果成功,则返回 0,否则返回负数(代表错误码)
```

```
 *       AVERROR(EAGAIN):   input is not accepted in the current state - user
 *                          must read output with avcodec_receive_frame()
 *       AVERROR_EOF:       the decoder has been flushed, and no new packets
 *       AVERROR(EINVAL):   codec not opened, it is an encoder, or requires flush
 *       AVERROR(ENOMEM):   failed to add packet to internal queue, or similar
 */
int avcodec_send_packet(AVCodecContext *avctx, const AVPacket *avpkt);


/**
 * Return decoded output data from a decoder.
 * 返回解码后的音视频裸数据,例如 YUV 或 PCM
 * @param avctx codec context :编解码器上下文
 * @param frame This will be set to a reference-counted video or audio
 *              frame (depending on the decoder type) allocated by the
 *              decoder. Note that the function will always call
 *              av_frame_unref(frame) before doing anything else.
 * 注意:这个函数在做任何工作之前,总是调用 av_frame_unref()函数
 * @return
 *       0: success, a frame was returned:返回 0 表示成功
 *       AVERROR(EAGAIN):   output is not available in this state
 *       AVERROR_EOF:       the decoder has been fully flushed
 *       AVERROR(EINVAL):   codec not opened, or it is an encoder
 *       AVERROR_INPUT_CHANGED: current decoded frame has changed parameters
 */
int avcodec_receive_frame(AVCodecContext *avctx, AVFrame *frame);
```

由描述可知，avcodec_send_packet()函数负责把 AVpacket 数据包发送给解码器，avcodec_receive_frame()函数则从解码器取出 AVFrame 数据。如果返回 0，则代表解码成功；如果返回 EAGAIN，则代表当前状态无可输出的数据；如果返回 EOF，则代表到达文件流结尾；如果返回 INVAL，则代表解码器未打开或者当前打开的是编码器；如果返回 INPUT_CHANGED，则代表输入参数发生变化，例如 width 和 height 改变了。avcodec_send_packet()和 avcodec_receive_frame()函数的解码流程如图 3-21 所示。

avcodec_send_packet()函数位于 libavcodec/decode.c 文件中，具体的代码如下：

```
//chapter3/3.6.help.txt
int avcodec_send_packet(AVCodecContext *avctx, const AVPacket *avpkt)
{
    AVCodecInternal *avci = avctx->internal;
    int ret;
    //判断解码器是否打开,是否为解码器
    if (!avcodec_is_open(avctx) || !av_codec_is_decoder(avctx->codec))
        return AVERROR(EINVAL);
    if (avctx->internal->draining)
```

```
        return AVERROR_EOF;
    if (avpkt && !avpkt->size && avpkt->data)
        return AVERROR(EINVAL);

    av_packet_unref(avci->buffer_pkt);                          //先解除包引用
    if (avpkt && (avpkt->data || avpkt->side_data_elems)) {     //判断非空
        ret = av_packet_ref(avci->buffer_pkt, avpkt);           //增加包引用
        if (ret < 0)
            return ret;
    }
    //则 packet 发送到 bitstream 滤波器
    ret = av_bsf_send_packet(avci->bsf, avci->buffer_pkt);
    if (ret < 0) {//如果失败了,则解除包引用
        av_packet_unref(avci->buffer_pkt);
        return ret;
    }
    if (!avci->buffer_frame->buf[0]) {//调用 decode_receive_frame_internal
        ret = decode_receive_frame_internal(avctx, avci->buffer_frame);
        if (ret < 0 && ret != AVERROR(EAGAIN) && ret != AVERROR_EOF)
            return ret;
    }

    return 0;
}
```

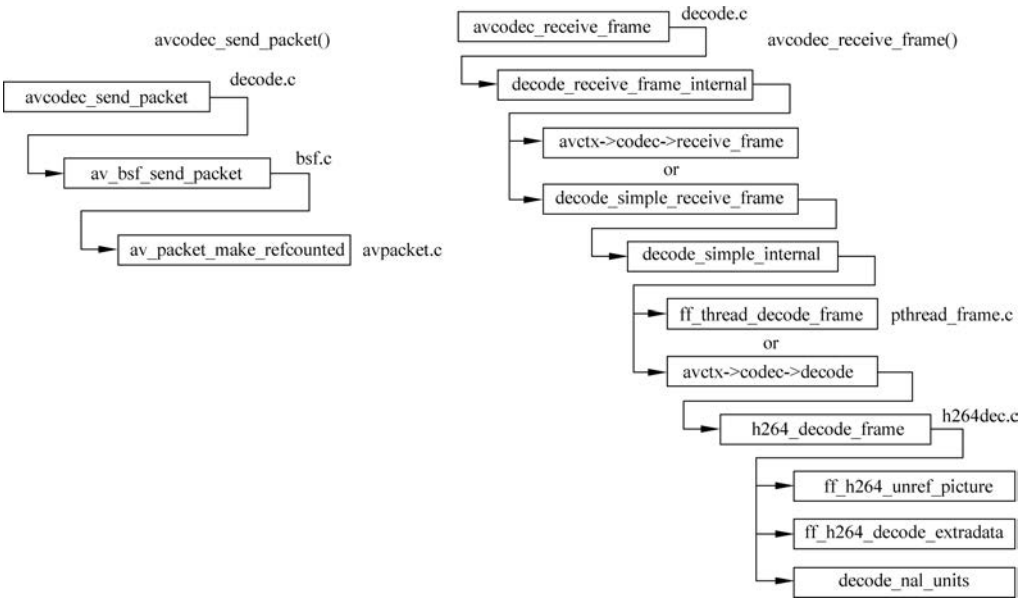

图 3-21　avcodec_send_packet()和 avcodec_receive_frame()函数

可以看到，在该函数内部调用 av_bsf_send_packet()函数把 packet 发送给 bitstream 滤波器，代码位于 libavcodec/bsf.c 文件中，具体的代码如下：

```
//chapter3/3.6.help.txt
int av_bsf_send_packet(AVBSFContext *ctx, AVPacket *pkt)
{
    AVBSFInternal *bsfi = ctx->internal;
    int ret;

    if (!pkt || IS_EMPTY(pkt)) {
        bsfi->eof = 1;
        return 0;
    }
    if (bsfi->eof) {
        return AVERROR(EINVAL);
    }
    if (!IS_EMPTY(bsfi->buffer_pkt))
        return AVERROR(EAGAIN);
    //申请 AVPacket 内存，并复制数据
    ret = av_packet_make_refcounted(pkt);
    if (ret < 0)
        return ret;
    //把 pkt 指针赋值给 bsfi->buffer_pkt
    av_packet_move_ref(bsfi->buffer_pkt, pkt);

    return 0;
}
```

可以看到，av_bsf_send_packet()函数内部调用 av_packet_make_refcounted()来申请 AVPacket 内存，并复制数据，然后调用 av_packet_move_ref()函数把 pkt 指针赋值给 bsfi->buffer_pkt。

avcodec_receive_frame()函数位于 libavcodec/decode.c 文件中，主要调用内部函数 decode_receive_frame_internal()实现解码，具体的代码如下：

```
//chapter3/3.6.help.txt
int avcodec_receive_frame(AVCodecContext *avctx, AVFrame *frame)
{
    AVCodecInternal *avci = avctx->internal;
    int ret, changed;

    av_frame_unref(frame);
    //判断解码器是否打开，是否为解码器
    if (!avcodec_is_open(avctx) || !av_codec_is_decoder(avctx->codec))
        return AVERROR(EINVAL);
```

```
    if (avci->buffer_frame->buf[0]) {
        av_frame_move_ref(frame, avci->buffer_frame);
    } else {
        ret = decode_receive_frame_internal(avctx, frame);
        if (ret < 0)
            return ret;
    }
    if (avctx->codec_type == AVMEDIA_TYPE_VIDEO) {
        ret = apply_cropping(avctx, frame);
        if (ret < 0) {
            av_frame_unref(frame);
            return ret;
        }
    }

    ...

    return 0;
}
```

decode_receive_frame_internal()函数用于判断是否支持 avctx->codec->receive_frame,如果支持就调用 avctx->codec->receive_frame()函数进行解码,否则调用 decode_simple_receive_frame()函数进行解码,具体的代码如下:

```
//chapter3/3.6.help.txt
static int decode_receive_frame_internal(AVCodecContext *avctx, AVFrame *frame)
{
    AVCodecInternal *avci = avctx->internal;
    int ret;

    if (avctx->codec->receive_frame) {
        ret = avctx->codec->receive_frame(avctx, frame);
        if (ret != AVERROR(EAGAIN))
            av_packet_unref(avci->last_pkt_props);
    } else {
        ret = decode_simple_receive_frame(avctx, frame);
    }
    ...

    /* free the per-frame decode data */
    av_buffer_unref(&frame->private_ref);
    return ret;
}
```

decode_simple_receive_frame()函数又调用 decode_simple_internal()内部函数进行解

码，代码如下：

```
//chapter3/3.6.help.txt
static int decode_simple_receive_frame(AVCodecContext *avctx, AVFrame *frame)
{
    int ret;
    int64_t discarded_samples = 0;

    while (!frame->buf[0]) {
        if (discarded_samples > avctx->max_samples)
            return AVERROR(EAGAIN);
        ret = decode_simple_internal(avctx, frame, &discarded_samples);
        if (ret < 0)
            return ret;
    }

    return 0;
}
```

可以看出，decode_simple_internal()函数是解码器的核心函数。在真实应用场景中，需要循环调用 avcodec_receive_frame()函数，直到返回 EAGAIN。通过判断是否需要特定线程进行解码，如果需要就调用 ff_thread_decode_frame()函数，否则调用 avctx->codec->decode 指向的解码函数，具体的代码如下：

```
//chapter3/3.6.help.txt
/*
 * The core of the receive_frame_wrapper for the decoders implementing
 * the simple API. 解码器的核心解码函数
 * Certain decoders might consume partial packets without
 * returning any output, so this function needs to be called in a loop until it 有可能没有任何输出的情况下，已经消费了一些输入的音视频包，所以需要循环调用该函数
 * returns EAGAIN.
 **/
static inline int decode_simple_internal(AVCodecContext *avctx,
                                         AVFrame *frame,
                                         int64_t *discarded_samples)
{
    AVCodecInternal   *avci = avctx->internal;
    DecodeSimpleContext *ds = &avci->ds;
    AVPacket            *pkt = ds->in_pkt;
    int got_frame, actual_got_frame;
    int ret;

    if (!pkt->data && !avci->draining) {//判断包非空
        av_packet_unref(pkt);
```

```
            ret = ff_decode_get_packet(avctx, pkt);
            if (ret < 0 && ret != AVERROR_EOF)
                return ret;
        }
        if (avci->draining_done)
            return AVERROR_EOF;

        if (!pkt->data &&
            !(avctx->codec->capabilities & AV_CODEC_CAP_DELAY ||
              avctx->active_thread_type & FF_THREAD_FRAME))
            return AVERROR_EOF;

        got_frame = 0;

         //判断是否需要特定线程进行解码
        if (HAVE_THREADS && avctx->active_thread_type & FF_THREAD_FRAME) {
            ret = ff_thread_decode_frame(avctx, frame, &got_frame, pkt);
        } else {
            ret = avctx->codec->decode(avctx, frame, &got_frame, pkt);

            ...
        }

        ...

        return ret < 0 ? ret : 0;
    }
```

7）核心编码函数

在 FFmpeg 中，avcodec_send_frame()和 avcodec_receive_packet()函数用于编码，函数原型如下（详见注释信息）：

```
//chapter3/3.6.help.txt
//int avcodec_send_frame(AVCodecContext *avctx, const AVFrame *frame);
//int avcodec_receive_packet(AVCodecContext *avctx, AVPacket *avpkt);

/**
* Supply a raw video or audio frame to the encoder. Use avcodec_receive_packet()  to retrieve
buffered output packets.
* 向编码器提供原始视频或音频帧.使用 avcodec_receive_packet()检索缓冲输出数据包
* @param avctx      codec context:编解码器上下文参数
* @param[in] frame AVFrame containing the raw audio or video frame to be encoded. 原始音频或
视频帧:AVFrame
*                   Ownership of the frame remains with the caller, and the
*                   encoder will not write to the frame. The encoder may create
```

```
 *                    a reference to the frame data (or copy it if the frame is
 *                    not reference-counted).
 *                    It can be NULL, in which case it is considered a flush
 *                    packet.  This signals the end of the stream. If the encoder
 *                    still has packets buffered, it will return them after this
 *                    call. Once flushing mode has been entered, additional flush
 *                    packets are ignored, and sending frames will return
 *                    AVERROR_EOF.
```

帧的所有权仍然属于调用方，并且编码器不会写入帧.编码器可能会创建对帧数据的引用(如果帧未计算引用).它可以为 NULL,在这种情况下,它被视为"刷新数据包".这标志着流的结束.如果编码器仍有缓冲的数据包,则它将在此之后返回它们.一旦进入冲洗模式,额外冲洗数据包被忽略,发送帧将返回 AVERROR_EOF.

```
 *
 *                    For audio:
 *                    If AV_CODEC_CAP_VARIABLE_FRAME_SIZE is set, then each frame
 *                    can have any number of samples.
 * 如果设置了 AV_CODEC_CAP_VARIABLE_FRAME_SIZE,则每个帧可以有任意数量的样本
 *                    If it is not set, frame->nb_samples must be equal to
 *                    avctx->frame_size for all frames except the last.
 * 如果未设置,frame->nb_samples 必须等于 avctx->frame 除最后一帧外的所有帧的大小
 *                    The final frame may be smaller than avctx->frame_size.
 * 最终帧可能小于 avctx->frame_size
 * @return 0 on success, otherwise negative error code:
 * 如果成功,则返回 0,否则返回错误码(负数)
 *      AVERROR(EAGAIN):   input is not accepted in the current state - user
 *                         must read output with avcodec_receive_packet() (once
 *                         all output is read, the packet should be resent, and
 *                         the call will not fail with EAGAIN).
 * 当前状态下不接收输入 -- 用户必须使用 avcodec_receive_packet()读取输出
 * (一旦读取所有输出,应重新发送数据包,并且在使用 EAGAIN 时,就不会失败)
 *      AVERROR_EOF:       the encoder has been flushed, and no new frames can
 *        be sent to it.编码器已刷新,不可以给它再发送新帧
 *      AVERROR(EINVAL):   codec not opened, it is a decoder, or requires flush
                               编解码器未打开,它是解码器,或需要刷新
 *      AVERROR(ENOMEM):   failed to add packet to internal queue, or similar
 *      other errors: legitimate encoding errors
 *无法将数据包添加到内部队列,或类似操作的其他错误:合法编码错误
 */
int avcodec_send_frame(AVCodecContext *avctx, const AVFrame *frame);

/**
 * Read encoded data from the encoder.
 * 从编码器读取编码数据
 * @param avctx codec context :编解码器上下文信息
 * @param avpkt This will be set to a reference-counted packet allocated by the encoder. Note
that the function will always call
```

```
 *                 av_packet_unref(avpkt) before doing anything else.
 * 这将被设置为编码器分配的参考计数数据包.
 * 需要注意,该函数将始终调用 av_packet_unref(avpkt),然后执行其他操作
 * @return 0 on success, otherwise negative error code:
 * 如果成功,则返回 0, 否则返回错误码(负数)
 *       AVERROR(EAGAIN):   output is not available in the current state - user
 *             must try to send input.输出在当前状态下不可用,用户必须尝试再次输入
 *       AVERROR_EOF:        the encoder has been fully flushed, and there will be   no more
output packets :编码器已完全刷新,将不再有输出数据包
 *       AVERROR(EINVAL):   codec not opened, or it is a decoder
 *       other errors: legitimate encoding errors
 */
int avcodec_receive_packet(AVCodecContext *avctx, AVPacket *avpkt);
```

对于编码,需要调用 avcodec_send_frame()函数为编码器提供包含未压缩音频或视频的 AVFrame。avcodec_send_frame() 函数负责将未压缩的 AVFrame 音视频数据传给编码器。成功后,它将返回一个带有压缩帧的 AVPacket。重复此调用,直到返回 AVERROR(EAGAIN)或错误。AVERROR(EAGAIN)返回值意味着需要新的输入数据才能返回新的输出。在这种情况下,继续发送输入。对于每个输入帧/数据包,编解码器通常会返回 1 个输出帧/数据包,但也可以是 0 或大于 1。avcodec_receive_packet()函数负责返回保存在 AVPacket 中的压缩数据,即编码数据。

FFmpeg 的音视频编码的函数执行流程如图 3-22 所示。

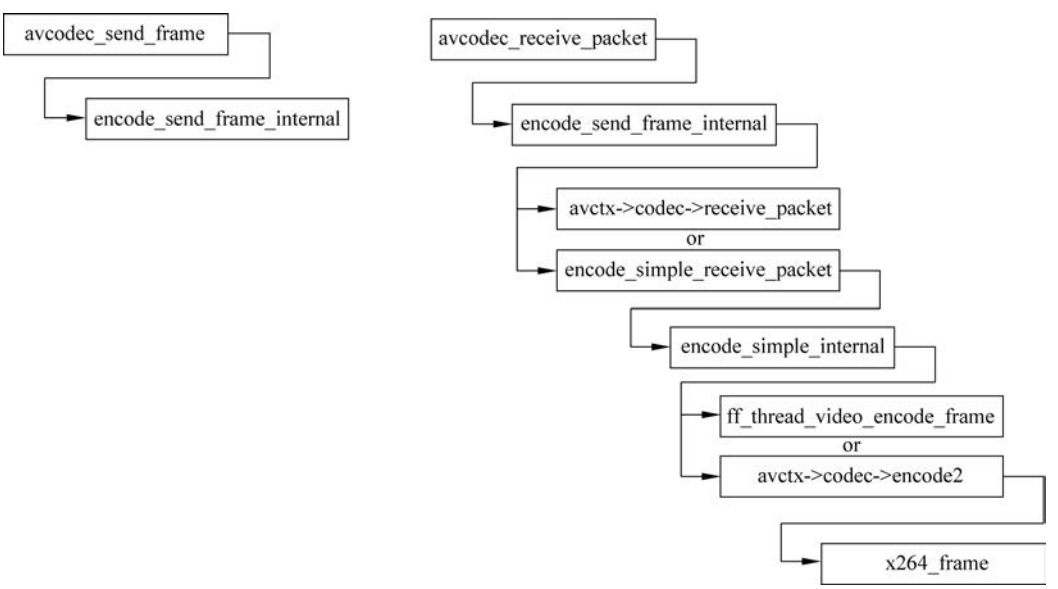

图 3-22　avcodec_send_frame()和 avcodec_receive_packet()函数

avcodec_send_frame()函数位于 libavcodec/encode.c 文件中，首先判断编码器有没打开及是否为编码器类型，然后调用 internal 函数执行具体操作，代码如下：

```
//chapter3/3.6.help.txt
int avcodec_send_frame(AVCodecContext *avctx, const AVFrame *frame)
{
    AVCodecInternal *avci = avctx->internal;
    int ret;
    //判断编码器有没有打开，是否为编码器
    if (!avcodec_is_open(avctx) || !av_codec_is_encoder(avctx->codec))
        return AVERROR(EINVAL);
    if (avci->draining)
        return AVERROR_EOF;
    if (avci->buffer_frame->data[0])
        return AVERROR(EAGAIN);
    if (!frame) {
        avci->draining = 1;
    } else {
        ret = encode_send_frame_internal(avctx, frame);
        if (ret < 0)
            return ret;
    }

    if (!avci->buffer_pkt->data && !avci->buffer_pkt->side_data) {
        ret = encode_receive_packet_internal(avctx, avci->buffer_pkt);
        if (ret < 0 && ret != AVERROR(EAGAIN) && ret != AVERROR_EOF)
            return ret;
    }

    return 0;
}
```

avcodec_receive_packet()函数首先判断编码器是否打开及是否为编码器类型，然后调用 encode_receive_packet_internal()函数执行具体操作，代码如下：

```
//chapter3/3.6.help.txt
int avcodec_receive_packet(AVCodecContext *avctx, AVPacket *avpkt)
{
    AVCodecInternal *avci = avctx->internal;
    int ret;
    av_packet_unref(avpkt);
    //判断编码器是否打开，是否为编码器
    if (!avcodec_is_open(avctx) || !av_codec_is_encoder(avctx->codec))
        return AVERROR(EINVAL);
```

```
    if (avci->buffer_pkt->data || avci->buffer_pkt->side_data) {
        av_packet_move_ref(avpkt, avci->buffer_pkt);
    } else {
        ret = encode_receive_packet_internal(avctx, avpkt);
        if (ret < 0)
            return ret;
    }

    return 0;
}
```

encode_receive_packet_internal()函数首先检测视频的宽和高、像素格式，然后判断使用 receive_packet 还是 encode_simple_receive_packet 执行编码操作。encode_simple_receive_packet()函数比较简单，主要调用 encode_simple_internal()函数。encode_simple_internal()函数首先判断 frame，如果 frame 为空，则调用 ff_encode_get_frame()取出一帧未压缩的数据，然后判断使用 ff_thread_video_encode_frame 还是 avctx->codec->encode2 执行真正的编码操作。最终会调用相应的编码器(libx264、libmp3lame)来执行真正的编码工作，例如 libx264 编码一帧图像会调用 X264_frame()函数。

由于 avcodec_encode_video2()函数已经过时，所以内部提供了 compat_encode()函数兼容旧版本，具体的代码如下：

```
//chapter3/3.6.help.txt
int avcodec_encode_video2(AVCodecContext *avctx,
                                                AVPacket *avpkt,
                                                const AVFrame *frame,
                                                int *got_packet_ptr)
{
    int ret = compat_encode(avctx, avpkt, got_packet_ptr, frame);

    if (ret < 0)
        av_packet_unref(avpkt);

    return ret;
}
```

而 compat_encode()函数内部其实调用了 avcodec_send_frame()和 avcodec_receive_packet()这两个函数进行编码，具体代码如下：

```
//chapter3/3.6.help.txt
static int compat_encode(AVCodecContext *avctx, AVPacket *avpkt,
                         int *got_packet, const AVFrame *frame)
{
    AVCodecInternal *avci = avctx->internal;
```

```
    AVPacket user_pkt;
    int ret;
    * got_packet = 0;
    //检测视频的 pixel_format、width、height
    if (frame && avctx->codec->type == AVMEDIA_TYPE_VIDEO) {
        if (frame->format == AV_PIX_FMT_NONE)
            av_log(avctx, AV_LOG_WARNING, "format is not set\n");
        if (frame->width == 0 || frame->height == 0)
            av_log(avctx, AV_LOG_WARNING, "width or height is not set\n");
    }
    //调用 avcodec_send_frame()发送 frame
    ret = avcodec_send_frame(avctx, frame);
    if (ret == AVERROR_EOF)
        ret = 0;
    else if (ret == AVERROR(EAGAIN)) {
        return AVERROR_Bug;
    } else if (ret < 0)
        return ret;

    av_packet_move_ref(&user_pkt, avpkt);
    while (ret >= 0) {
        //调用 avcodec_receive_packet()接收 packet
        ret = avcodec_receive_packet(avctx, avpkt);
        if (ret < 0) {
            if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF)
                ret = 0;
            goto finish;
        }
        …
    }

finish:
    if (ret < 0)
        av_packet_unref(&user_pkt);

    return ret;
}
```

和 avcodec_encode_video2()函数一样，avcodec_encode_audio2()函数已经过时，内部也提供了 compat_encode()函数兼容旧版本，具体的代码如下：

```
//chapter3/3.6.help.txt
int avcodec_encode_audio2(AVCodecContext * avctx,
                                                AVPacket * avpkt,
                                                const AVFrame * frame,
```

```
                                                    int * got_packet_ptr)
{
    int ret = compat_encode(avctx, avpkt, got_packet_ptr, frame);

    if (ret < 0)
        av_packet_unref(avpkt);

    return ret;
}
```

4. 内存分配相关函数

内存操作的常见函数位于 libavutil\mem.c 文件中，FFmpeg 开发中最常使用的几个内存操作函数包括 av_malloc()、av_realloc()、av_mallocz()、av_calloc()、av_free()和 av_freep()等。

1）av_malloc()函数

av_malloc()是 libavutil 模块中的函数，用于给对象分配内存块，并且是内存对齐的，该函数声明的代码如下：

```
//chapter3/3.6.help.txt
/**
 * Allocate a memory block with alignment suitable for all memory accesses
 * (including vectors if available on the CPU).以内存对齐方式分配内存
 * @param size Size in Bytes for the memory block to be allocated
 * @return Pointer to the allocated block, or `NULL` if the block cannot
 *         be allocated
 * @see av_mallocz()
 */
void * av_malloc(size_t size) av_malloc_attrib av_alloc_size(1);
```

该函数用于分配一个适合所有内存访问的对齐方式的内存块(包括向量，如果在 CPU 上可用)。参数 size 表示要分配的内存块的大小(以字节为单位)。如果成功，则返回指向已分配块的指针，否则返回 NULL。

av_malloc_attrib 是一个宏定义，如果是在编译器 GCC 3.1 及以上版本的情况下，则需给方法 av_malloc 增加属性__attribute__((malloc))，该属性用于指示编译器按照 malloc 函数来对待，并且可以对其实施相应的优化措施，该宏的代码如下：

```
//chapter3/3.6.help.txt
  //@def av_malloc_attrib
  //函数属性表示类似 malloc 的函数
/**
 * @def av_malloc_attrib
```

```
 * Function attribute denoting a malloc-like function.
 */
#if AV_GCC_VERSION_AT_LEAST(3,1)
    #define av_malloc_attrib __attribute__((__malloc__))
#else
    #define av_malloc_attrib
#endif
```

av_alloc_size(1)也是一个宏定义，如果是在编译器 GCC 4.3 及以上版本的情况下，则需给函数增加一个属性__attribute__((alloc_size(1)))，告知编译器 av_malloc(size_t size)函数的第 1 个参数，即 size 是要分配的空间大小。

av_malloc()是 FFmpeg 中最常见的内存分配函数，它的定义代码如下：

```
//chapter3/3.6.help.txt
void *av_malloc(size_t size)
{
    void *ptr = NULL;

    /* let's disallow possibly ambiguous cases:禁止有歧义的情况 */
    if (size > (max_alloc_size - 32))
        return NULL;

#if HAVE_POSIX_MEMALIGN //处理内存对齐
    if (size) //OS X on SDK 10.6 has a broken posix_memalign implementation
    if (posix_memalign(&ptr, ALIGN, size))
        ptr = NULL;
#elif HAVE_ALIGNED_MALLOC
    ptr = _aligned_malloc(size, ALIGN);
#elif HAVE_MEMALIGN
#ifndef __DJGPP__
    ptr = memalign(ALIGN, size);
#else
    ptr = memalign(size, ALIGN);
#endif

#else
    ptr = malloc(size);
#endif
    if(!ptr && !size) {
        size = 1;
        ptr= av_malloc(1);
    }
#if CONFIG_MEMORY_POISONING
    if (ptr)
```

```
        memset(ptr, FF_MEMORY_POISON, size);
#endif
    return ptr;
}
```

可以看到，av_malloc()简单地封装了系统函数 malloc()，并做了一些错误检查工作，以内存对齐的方式调用 memalign()或 malloc()函数来分配指定大小的内存块。

2）malloc()函数

malloc()函数用来动态地分配内存空间(C 语言动态内存分配及变量存储类别)，其函数原型如下：

```
void * malloc (size_t size);
```

参数 size 为需要分配的内存空间的大小，以字节(Byte)为单位。如果分配成功，则返回指向该内存的地址，否则返回 NULL。

该函数在堆区分配一块指定大小的内存空间，用来存放数据。这块内存空间在函数执行完成后不会被初始化，它们的值是未知的。如果希望在分配内存的同时进行初始化，则可以使用 calloc()函数。由于申请内存空间时可能会失败，所以需要自行判断是否申请成功，再进行后续操作。如果 size 的值为 0，则返回值会因标准库实现的不同而不同，可能是 NULL，也可能不是，但返回的指针不应该再次被引用。

size_t 这种类型在 FFmpeg 中多次出现，简单解释一下其作用。size_t 是为了增强程序的可移植性而定义的。在不同的系统上，定义 size_t 可能不一样。它实际上是 unsigned int。

注意：malloc()函数的返回值类型是 void *，void 并不是说没有返回值或者返回空指针，而是返回的指针类型未知，所以在使用 malloc() 时通常需要进行强制类型转换，将 void 指针转换成具体的类型，例如 char *ptr = (char *)malloc(10)。

下面列举一个动态内存分配的案例，代码如下：

```
//chapter3/3.6.help.txt
#include <stdio.h>  /* printf, scanf, NULL */
#include <stdlib.h>  /* malloc, free, rand, system */
int main ()
{
    int i,n;
    char * buffer;
    printf ("输入字符串的长度:");
    scanf ("%d", &i);
```

```
    buffer = (char*)malloc(i+1);          //字符串最后包含 \0
    if(buffer==NULL) exit(1);             //判断是否分配成功
    //随机生成字符串
    for(n=0; n<i; n++)
        buffer[n] = rand()%26+'a';
    buffer[i]='\0';
    printf("随机生成的字符串为%s\n",buffer);
    free(buffer);                         //释放内存空间
    system("pause");
    return 0;
}
```

该程序会生成一个指定长度的字符串，并用随机生成的字符填充。字符串的长度受限于可用内存的长度。

可能的运行结果如下：

```
输入字符串的长度:20
随机生成的字符串为 pcqdhueeaxlnlf23firc
```

3）内存对齐简介

posix_memalign()函数在大多数情况下，编译器和C库会自动处理对齐问题。POSIX标明了通过malloc( )、calloc( )和realloc( )返回的地址对于任何的C类型来讲都是对齐的。在Linux系统中，这些函数返回的地址在32位系统是以8字节为边界对齐的，在64位系统是以16字节为边界对齐的。有时，对于更大的边界，例如页面，程序员需要动态地对齐。虽然动机是多种多样的，但最常见的是I/O缓存的对齐或者其他的软件对硬件的交互，因此，POSIX 1003.1d提供了一个叫作posix_memalign( )的函数。

下面用一个例子来解释内存对齐，看下面的小程序，理论上，32位系统下，int占4字节，char占1字节，那么将它们放到一个结构体中应该占4+1=5字节，但是实际上，通过运行程序得到的结果是8字节，这就是由内存对齐所导致的，代码如下：

```
//chapter3/3.6.help.txt
//32位系统
#include<stdio.h>
struct{
    int x;
    char y;
} s;
int main()
{
    printf("%d\n", sizeof(s)),; //输出8
    return 0;
}
```

现代计算机中内存空间都是按照字节(Byte)划分的,从理论上讲似乎对任何类型的变量的访问都可以从任何地址开始,但是实际的计算机系统对基本类型数据在内存中存放的位置有限制,它们会要求这些数据的首地址的值是某个数 k(通常它为 4 或 8)的倍数,这就是所谓的内存对齐。

尽管内存是以字节为单位的,但是大部分处理器并不是按字节块来存取内存的。它一般会以双字节、4 字节、8 字节、16 字节甚至 32 字节为单位来存取内存,将上述这些存取单位称为内存存取粒度。

现在考虑 4 字节存取粒度的处理器取 int 类型变量(32 位系统),该处理器只能从地址为 4 的倍数的内存开始读取数据。

(1) 平台原因(移植原因):不是所有的硬件平台都能访问任意地址上的任意数据的;某些硬件平台只能在某些地址处取某些特定类型的数据,否则会抛出硬件异常。

(2) 性能原因:数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于,为了访问未对齐的内存,处理器需要进行两次内存访问,而对齐的内存访问仅需要一次访问。

假如没有内存对齐机制,数据可以任意存放,现在一个 int 变量存放在从地址 1 开始的连续 4 字节地址中,该处理器去取数据时,要先从 0 地址开始读取第 1 个 4 字节块,剔除不想要的字节(0 地址),然后从地址 4 开始读取下一个 4 字节块,同样剔除不要的数据(5、6、7 地址),最后留下的两块数据合并放入寄存器。这需要做很多工作。

现在有了内存对齐,int 类型数据只能存放在按照对齐规则的内存中,例如 0 地址开始的内存。那么现在该处理器在取数据时一次性就能将数据读出来了,而且不需要做额外的操作,从而提高了效率。

内存对齐规则如下。

(1) 基本类型的对齐值就是其 sizeof 值。

(2) 结构体的对齐值就是其成员的最大对齐值。

(3) 编译器可以设置一个最大对齐值,类型的实际对齐值是该类型的对齐值与默认对齐值取最小值得来。

4) av_mallocz()函数

av_mallocz()函数可以理解为 av_malloc()+ZeroMemory,函数声明的代码如下:

```
//chapter3/3.6.help.txt
/* 分配一个适合所有内存访问的对齐方式的内存块
   *(包括向量,如果在 CPU 上可用)并将所有字节归零.
   * @param size 要分配的内存块的大小(以字节为单位)
   * @return 指向已分配块的指针,如果无法分配,则为`NULL`
   * @参见 av_malloc()
 */
/**
 * Allocate a memory block with alignment suitable for all memory accesses
 * (including vectors if available on the CPU) and zero all the Bytes of the
```

```
 * block.
 *
 * @param size Size in Bytes for the memory block to be allocated
 * @return Pointer to the allocated block, or `NULL` if it cannot be allocated
 * @see av_malloc()
 */
void *av_mallocz(size_t size) av_malloc_attrib av_alloc_size(1);
```

从源代码可以看出 av_mallocz()中调用了 av_malloc()之后，又调用 memset()将分配的内存设置为 0，代码如下：

```
//chapter3/3.6.help.txt
void *av_mallocz(size_t size)
{
    void *ptr = av_malloc(size);            //使用 av_malloc 分配内存
    if (ptr)
        memset(ptr, 0, size);               //将分配的内存的所有字节置为 0
    return ptr;
}
```

memset()函数用来初始化内存，函数声明的代码如下：

```
//头文件：#include<string.h>
void *memset(void *str, int c, size_t n)
```

该函数的功能是将字符 c(一个无符号字符)复制到参数 str 所指向的字符串的前 n 个字符。它是在一段内存块中填充某个给定的值，是对较大的结构体或数组进行清零操作的一种非常快的方法。

函数非常简单，只用于初始化，但是需要注意的是 memset 赋值时是按字节赋值的，是将参数化成二进制之后填入一字节。例如想通过 memset(a,100,sizeof a)给 int 类型的数组赋值，给第 1 字节的是 100，转换成二进制就是 0110 0100，而 int 有 4 字节，也就是说，一个 int 被赋值为 0110 0100,0110 0100,0110 0100,0110 0100，对应的十进制是 1 684 300 900，根本不是想要赋的值 100。

memset 赋值时可以是任何值，例如任意字符都是可以的，初始化成 0 是最常用的。int 类型一般赋值 0 或－1，其他的值都不行。

总之，为地址 str 开始的 n 字节赋值 c，注意：是逐字节赋值，str 开始的 n 字节中的每字节都赋值为 c。

(1) 若 str 指向 char 型地址，则 value 可为任意字符值。

(2) 若 str 指向非 char 型，如 int 型地址，要想赋值正确，value 的值只能是－1 或 0，因为－1 和 0 转化成二进制后每位都是一样的，假如 int 型占 4 字节，则－1＝0XFFFFFFFF，0＝0X00000000。

例如给数组 A 赋值－1，代码如下：

```
int A[2];
memset(A, -1, sizeof A);
```

5）av_calloc()函数

av_calloc()函数简单封装了 av_mallocz()，函数定义的代码如下：

```
//chapter3/3.6.help.txt
void *av_calloc(size_t nmemb, size_t size)
{
    if (size <= 0 || nmemb >= INT_MAX / size)
        return NULL;
    return av_mallocz(nmemb * size);
}
```

可以看出，它调用 av_mallocz()函数分配了 nmemb * size 字节的内存(*代表乘)。

6）av_free()函数

av_free()用于释放申请的内存，它的定义代码如下：

```
//chapter3/3.6.help.txt
void av_free(void *ptr)
{
#if CONFIG_MEMALIGN_HACK
    if (ptr) {
        int v= ((char *)ptr)[-1];
        av_assert0(v>0 && v<=ALIGN);
        free((char *)ptr - v);
    }
#elif HAVE_ALIGNED_MALLOC
    _aligned_free(ptr);
#else
    free(ptr);
#endif
}
```

默认情况下(CONFIG_MEMALIGN_HACK 这些宏使用的默认值为 0)的代码如下：

```
//chapter3/3.6.help.txt
void av_free(void *ptr)
{
    free(ptr);
}
```

可以看出，av_free()只是简单地封装了 free()函数。

7）av_freep()函数

av_freep()函数封装了av_free()函数，并且在释放内存之后将目标指针设置为NULL，函数的定义代码如下：

```
//chapter3/3.6.help.txt
void av_freep(void *arg)
{
    void **ptr = (void **)arg;
    av_free(*ptr);
    *ptr = NULL;
}
```

3.7 Ubuntu下编译并运行解封装案例

在“3.1 FFmpeg的读者入门案例”中使用的是Qt开发工具，在Windows 10环境下调试成功。现在将代码文件ffmpeganalysestreams.cpp复制到Ubuntu系统中，使用的编译命令如下：

```
//chapter3/3.7.help.txt
gcc -o ffmpeganalysestreams ffmpeganalysestreams.cpp -I
/root/ffmpeg-5.0.1/install5/include/ -L /root/ffmpeg-5.0.1/install5/lib/
-lavcodec -lavformat -lavutil
```

编译成功后会生成可执行文件ffmpeganalysestreams，然后将hello.mp4文件复制到Ubuntu同路径下，如图3-23所示。运行该程序即可，如图3-24所示。

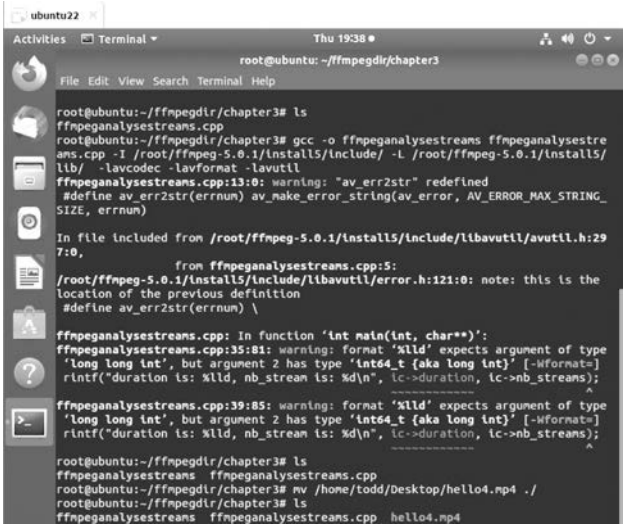

图3-23 avcodec_send_frame()和avcodec_receive_packet()函数

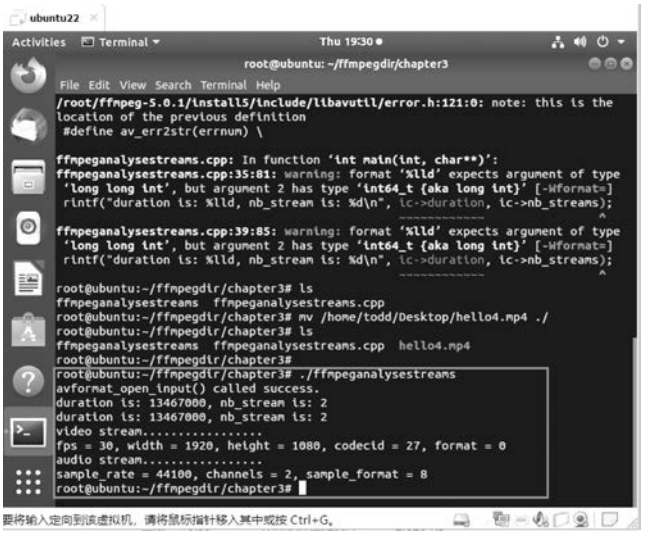

图 3-24　avcodec_send_frame()和 avcodec_receive_packet()函数

由于 Linux 下编译这些 C++文件的命令几乎完全相同，后续章节中不再重复，这里先给出比较完整的 Linux 系统中编译 FFmpeg 案例的命令，具体命令如下：

```
//chapter3/3.7.help.txt
gcc -o hello xxx.cpp -I /root/ffmpeg-5.0.1/install5/include/ -L /root/ffmpeg-5.0.1/install5/lib/ -lavcodec -lavformat -lavutil -lswscale -lswresample -lavfilter -lstdc++
```

注意：如果用到 C++的特性及库函数，则编译时需要 C++链接库-lstdc++。