原 FFmpeg源代码简单分析：avio_open2()

2015年03月04日 14:16:41　阅读数：38499

==========

FFmpeg的库函数源代码分析文章列表：

【架构图】

FFmpeg 源代码结构图 - 解码

FFmpeg 源代码结构图 - 编码

【通用】

FFmpeg 源代码简单分析： av_register_all()

FFmpeg 源代码简单分析： avcodec_register_all()

FFmpeg 源代码简单分析：内存的分配和释放（ av_malloc()、 av_free() 等）

FFmpeg 源代码简单分析：常见结构体的初始化和销毁（ AVFormatContext， AVFrame 等）

FFmpeg 源代码简单分析： avio_open2()

FFmpeg 源代码简单分析： av_find_decoder() 和 av_find_encoder()

FFmpeg 源代码简单分析： avcodec_open2()

FFmpeg 源代码简单分析： avcodec_close()

【解码】

图解 FFMPEG 打开媒体的函数 avformat_open_input

FFmpeg 源代码简单分析： avformat_open_input()

FFmpeg 源代码简单分析： avformat_find_stream_info()

FFmpeg 源代码简单分析： av_read_frame()

FFmpeg 源代码简单分析： avcodec_decode_video2()

FFmpeg 源代码简单分析： avformat_close_input()

【编码】

FFmpeg 源代码简单分析： avformat_alloc_output_context2()

FFmpeg 源代码简单分析： avformat_write_header()

FFmpeg 源代码简单分析： avcodec_encode_video()

FFmpeg 源代码简单分析： av_write_frame()

FFmpeg 源代码简单分析： av_write_trailer()

【其它】

FFmpeg 源代码简单分析：日志输出系统（ av_log() 等）

FFmpeg 源代码简单分析：结构体成员管理系统 -AVClass

FFmpeg 源代码简单分析：结构体成员管理系统 -AVOption

FFmpeg 源代码简单分析： libswscale 的 sws_getContext()

FFmpeg 源代码简单分析： libswscale 的 sws_scale()

FFmpeg 源代码简单分析： libavdevice 的 avdevice_register_all()

FFmpeg 源代码简单分析： libavdevice 的 gdigrab

【脚本】

FFmpeg 源代码简单分析： makefile

FFmpeg 源代码简单分析： configure

【H.264】

FFmpeg 的 H.264 解码器源代码简单分析：概述

==========

本文简单分析FFmpeg中一个常用的函数avio_open2()。该函数用于打开FFmpeg的输入输出文件。avio_open2()的声明位于libavformat\avio.h文件中，如下所示。

```cpp
/**
 * Create and initialize a AVIOContext for accessing the
 * resource indicated by url.
 * @note When the resource indicated by url has been opened in
 * read+write mode, the AVIOContext can be used only for writing.
 *
 * @param s Used to return the pointer to the created AVIOContext.
 * In case of failure the pointed to value is set to NULL.
 * @param url resource to access
 * @param flags flags which control how the resource indicated by url
 * is to be opened
 * @param int_cb an interrupt callback to be used at the protocols level
 * @param options  A dictionary filled with protocol-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 in case of success, a negative value corresponding to an
 * AVERROR code in case of failure
 */
int avio_open2(AVIOContext **s, const char *url, int flags,
               const AVIOInterruptCB *int_cb, AVDictionary **options);
```

avio_open2()函数参数的含义如下：

s：函数调用成功之后创建的AVIOContext结构体。

url：输入输出协议的地址（文件也是一种“广义”的协议，对于文件来说就是文件的路径）。

flags：打开地址的方式。可以选择只读，只写，或者读写。取值如下。

AVIO_FLAG_READ：只读。

AVIO_FLAG_WRITE：只写。

AVIO_FLAG_READ_WRITE：读写。

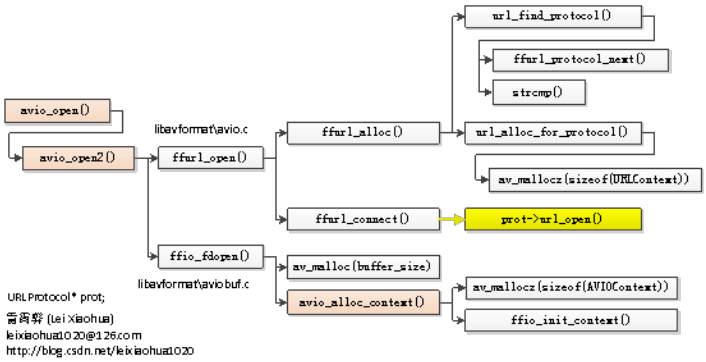
int_cb：目前还没有用过。

options：目前还没有用过。

该函数最典型的例子可以参考： 最简单的基于FFMPEG的视频编码器（YUV编码为H.264）

函数调用结构图

首先贴出来最终分析得出的函数调用结构图，如下所示。



单击查看更清晰的图片

avio_open()

有一个和avio_open2()“长得很像”的函数avio_open()，应该是avio_open2()的早期版本。avio_open()比avio_open2()少了最后2个参数。而它前面几个参数的含义和avio_open2()是一样的。从源代码中可以看出，avio_open()内部调用了avio_open2()，并且把avio_open2()的后2个参数设置成了NULL，因此它的功能实际上和avio_open2()是一样的。avio_open()源代码如下所示。

```cpp
int avio_open(AVIOContext **s, const char *filename, int flags)
{
    return avio_open2(s, filename, flags, NULL, NULL);
}
```

avio_open2()

下面看一下avio_open2()的源代码，位于libavformat\aviobuf.c文件中。

```cpp
int avio_open2(AVIOContext **s, const char *filename, int flags,
               const AVIOInterruptCB *int_cb, AVDictionary **options)
{
    URLContext *h;
    int err;


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

从avio_open2()的源代码可以看出，它主要调用了2个函数：ffurl_open()和ffio_fdopen()。其中ffurl_open()用于初始化URLContext，ffio_fdopen()用于根据URLContext初始化AVIOContext。URLContext中包含的URLProtocol完成了具体的协议读写等工作。AVIOContext则是在URLContext的读写函数外面加上了一层“包装”（通过retry_transfer_wrapper()函数）。

URLProtocol和URLContext

在查看ffurl_open()和ffio_fdopen()函数之前，首先查看一下URLContext和URLProtocol的定义。这两个结构体在FFmpeg的早期版本的SDK中是定义在头文件中可以直接使用的。但是近期的FFmpeg的SDK中已经找不到这两个结构体的定义了。FFmpeg把这两个结构体移动到了源代码的内部，变成了内部结构体。
URLProtocol的定义位于libavformat\url.h，如下所示。

```cpp
typedef struct URLProtocol {
    const char *name;
    int     (*url_open)( URLContext *h, const char *url, int flags);
    /**
     * This callback is to be used by protocols which open further nested
     * protocols. options are then to be passed to ffurl_open()/ffurl_connect()
     * for those nested protocols.
     */
    int     (*url_open2)(URLContext *h, const char *url, int flags, AVDictionary **options);


    /**
     * Read data from the protocol.
     * If data is immediately available (even less than size), EOF is
     * reached or an error occurs (including EINTR), return immediately.
     * Otherwise:
     * In non-blocking mode, return AVERROR(EAGAIN) immediately.
     * In blocking mode, wait for data/EOF/error with a short timeout (0.1s),
     * and return AVERROR(EAGAIN) on timeout.
     * Checking interrupt_callback, looping on EINTR and EAGAIN and until
     * enough data has been read is left to the calling function; see
     * retry_transfer_wrapper in avio.c.
     */
    int     (*url_read)( URLContext *h, unsigned char *buf, int size);
    int     (*url_write)(URLContext *h, const unsigned char *buf, int size);
    int64_t (*url_seek)( URLContext *h, int64_t pos, int whence);
    int     (*url_close)(URLContext *h);
    struct URLProtocol *next;
    int (*url_read_pause)(URLContext *h, int pause);
    int64_t (*url_read_seek)(URLContext *h, int stream_index,
                             int64_t timestamp, int flags);
    int (*url_get_file_handle)(URLContext *h);
    int (*url_get_multi_file_handle)(URLContext *h, int **handles,
                                     int *numhandles);
    int (*url_shutdown)(URLContext *h, int flags);
    int priv_data_size;
    const AVClass *priv_data_class;
    int flags;
    int (*url_check)(URLContext *h, int mask);
} URLProtocol;
```

从URLProtocol的定义可以看出，其中包含了用于协议读写的函数指针。例如：
url_open()：打开协议。
url_read()：读数据。
url_write()：写数据。
url_close()：关闭协议。
每种具体的协议都包含了一个URLProtocol结构体，例如：
FILE协议（“文件”在FFmpeg中也被当做一种协议）的结构体ff_file_protocol的定义如下所示（位于libavformat\file.c）。

```cpp
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,
};
```

在使用FILE协议进行读写的时候，调用url_open()实际上就是调用了file_open()函数，这里限于篇幅不再对file_open()的源代码进行分析。file_open()函数实际上调用了系统的打开文件函数open()。同理，调用url_read()实际上就是调用了file_read()函数；file_read()函数实际上调用了系统的读取文件函数read()。url_write()，url_seek()等函数的道理都是一样的。

LibRTMP协议的结构体ff_librtmp_protocol的定义如下所示（位于libavformat\librtmp.c）。

```cpp
URLProtocol ff_librtmp_protocol = {
    .name                = "rtmp",
    .url_open            = rtmp_open,
    .url_read            = rtmp_read,
    .url_write           = rtmp_write,
    .url_close           = rtmp_close,
    .url_read_pause      = rtmp_read_pause,
    .url_read_seek       = rtmp_read_seek,
    .url_get_file_handle = rtmp_get_file_handle,
    .priv_data_size      = sizeof(LibRTMPContext),
    .priv_data_class     = &librtmp_class,
    .flags               = URL_PROTOCOL_FLAG_NETWORK,
};
```

UDP协议的结构体ff_udp_protocol的定义如下所示（位于libavformat\udp.c）。

```cpp
URLProtocol ff_udp_protocol = {
    .name                = "udp",
    .url_open            = udp_open,
    .url_read            = udp_read,
    .url_write           = udp_write,
    .url_close           = udp_close,
    .url_get_file_handle = udp_get_file_handle,
    .priv_data_size      = sizeof(UDPContext),
    .priv_data_class     = &udp_context_class,
    .flags               = URL_PROTOCOL_FLAG_NETWORK,
};
```

上文中简单介绍了URLProtocol结构体。下面看一下URLContext结构体。URLContext的定义也位于libavformat\url.h，如下所示。

```cpp
typedef struct URLContext {
    const AVClass *av_class;    /**< information for av_log(). Set by url_open(). */
    struct URLProtocol *prot;
    void *priv_data;
    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 */
} URLContext;
```

从代码中可以看出，URLProtocol结构体是URLContext结构体的一个成员。由于还没有对URLContext结构体进行详细研究，有关该结构体的代码不再做过多分析。

ffurl_open()

前文提到AVIOContext中主要调用了2个函数：ffurl_open()和ffio_fdopen()。其中ffurl_open()用于初始化URLContext，ffio_fdopen()用于根据URLContext初始化AVIOContext。下面首先看一下初始化URLContext的函数ffurl_open()。
ffurl_open()的函数定义位于libavformat\avio.c中，如下所示。

```cpp
int ffurl_open(URLContext **puc, const char *filename, int flags,
               const AVIOInterruptCB *int_cb, AVDictionary **options)
{
    int ret = ffurl_alloc(puc, filename, flags, int_cb);
    if (ret < 0)
        return ret;
    if (options && (*puc)->prot->priv_data_class &&
        (ret = av_opt_set_dict((*puc)->priv_data, options)) < 0)
        goto fail;
    if ((ret = av_opt_set_dict(*puc, options)) < 0)
        goto fail;
    ret = ffurl_connect(*puc, options);
    if (!ret)
        return 0;
fail:
    ffurl_close(*puc);
    *puc = NULL;
    return ret;
}
```

从代码中可以看出，ffurl_open()主要调用了2个函数：ffurl_alloc()和ffurl_connect()。ffurl_alloc()用于查找合适的URLProtocol，并创建一个URLContext；ffurl_connect()用于打开获得的URLProtocol。

ffurl_alloc()

ffurl_alloc()的定义位于libavformat\avio.c中，如下所示。

```cpp
int ffurl_alloc(URLContext **puc, const char *filename, int flags,
                const AVIOInterruptCB *int_cb)
{
    URLProtocol *p = NULL;


    if (!first_protocol) {
        av_log(NULL, AV_LOG_WARNING, "No URL Protocols are registered. "
                                     "Missing call to av_register_all()?\n");
    }


    p = url_find_protocol(filename);
    if (p)
       return url_alloc_for_protocol(puc, p, filename, flags, int_cb);


    *puc = NULL;
    if (av_strstart(filename, "https:", NULL))
        av_log(NULL, AV_LOG_WARNING, "https protocol not found, recompile with openssl or gnutls enabled.\n");
    return AVERROR_PROTOCOL_NOT_FOUND;
}
```

从代码中可以看出，ffurl_alloc()主要调用了2个函数：url_find_protocol()根据文件路径查找合适的URLProtocol，url_alloc_for_protocol()为查找到的URLProtocol创建URLContext。

url_find_protocol()

先来看一下url_find_protocol()函数，定义如下所示。

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

static struct URLProtocol *url_find_protocol(const char *filename)
{
    URLProtocol *up = NULL;
    char proto_str[128], proto_nested[128], *ptr;
    size_t proto_len = strspn(filename, URL_SCHEME_CHARS);


    if (filename[proto_len] != ':' &&
        (filename[proto_len] != ',' || !strchr(filename + proto_len + 1, ':')) ||
        is_dos_path(filename))
        strcpy(proto_str, "file");
    else
        av_strlcpy(proto_str, filename,
                   FFMIN(proto_len + 1, sizeof(proto_str)));


    if ((ptr = strchr(proto_str, ',')))
        *ptr = '\0';
    av_strlcpy(proto_nested, proto_str, sizeof(proto_nested));
    if ((ptr = strchr(proto_nested, '+')))
        *ptr = '\0';


    while (up = ffurl_protocol_next(up)) {
        if (!strcmp(proto_str, up->name))
            break;
        if (up->flags & URL_PROTOCOL_FLAG_NESTED_SCHEME &&
            !strcmp(proto_nested, up->name))
            break;
    }


    return up;
}
```

url_find_protocol()函数表明了FFmpeg根据文件路径猜测协议的方法。该函数首先根据strspn()函数查找字符串中第一个“非字母或数字”的字符的位置，并保存在proto_len中。一般情况下，协议URL中都是包含“:”的，比如说RTMP的URL格式是“rtmp://xxx…”，UDP的URL格式是“udp://…”，HTTP的URL格式是“http://...”。因此，一般情况下proto_len的数值就是“:”的下标（代表了“:”前面的协议名称的字符的个数，例如rtmp://的proto_len为4）。
接下来函数将filename的前proto_len个字节拷贝至proto_str字符串中。
PS：

这个地方比较纠结，源代码中av_strlcpy()函数的第3个参数size写的字符串的长度是（proto_len+1），但是查了一下av_strlcpy()的定义，发现该函数至多拷贝（size-1）个字符。这么一涨一消，最终还是拷贝了proto_len个字节。例如RTMP协议就拷贝了“rtmp”，UDP协议就拷贝了“udp”。

av_strlcpy()是FFMpeg的一个工具函数，声明位于libavutil\avstring.h，如下所示。

```cpp
/**
 * Copy the string src to dst, but no more than size - 1 bytes, and
 * null-terminate dst.
 *
 * This function is the same as BSD strlcpy().
 *
 * @param dst destination buffer
 * @param src source string
 * @param size size of destination buffer
 * @return the length of src
 *
 * @warning since the return value is the length of src, src absolutely
 * _must_ be a properly 0-terminated string, otherwise this will read beyond
 * the end of the buffer and possibly crash.
 */
size_t av_strlcpy(char *dst, const char *src, size_t size);
```

这里有一种例外，那就是文件路径。“文件”在FFmpeg中也是一种“协议”，并且前缀是“file”。也就是标准的文件路径应该是“file://...”格式的。但是这太不符合我们一般人的使用习惯，我们一般是不会在文件路径前面加上“file”协议名称的。所以该函数采取的方法是：一旦检测出来输入的URL是文件路径而不是网络协议，就自动向proto_str中拷贝“file”。

其中判断文件路径那里有一个很复杂的if()语句。根据我的理解，“||”前面的语句用于判断是否是相对文件路径，“||”后面的语句用于判断是否是绝对路径。判断绝对路径的时候用到了一个函数is_dos_path()，定义位于libavformat\os_support.h，如下所示。

```cpp
static inline int is_dos_path(const char *path)
{
#if HAVE_DOS_PATHS
    if (path[0] && path[1] == ':')
        return 1;
#endif
    return 0;
}
```

注意“&&”优先级低于“==”。如果文件路径第1个字符不为空（一般情况下是盘符）而且第2个字符为“:”，就认为它是绝对文件路径。
此外url_find_protocol()函数中还涉及到一个函数ffurl_protocol_next()。该函数用于获得下一个URLProtocol（所有的URLProtocol在FFmpeg初始化注册的时候形成一个链表结构）。ffurl_protocol_next()代码极其简单，如下所示。

```cpp
URLProtocol *ffurl_protocol_next(const URLProtocol *prev)
{
    return prev ? prev->next : first_protocol;
}
```

url_alloc_for_protocol()

url_alloc_for_protocol()的定义位于libavformat\avio.c中，如下所示。

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


#if CONFIG_NETWORK
    if (up->flags & URL_PROTOCOL_FLAG_NETWORK && !ff_network_init())
        return AVERROR(EIO);
#endif
    if ((flags & AVIO_FLAG_READ) && !up->url_read) {
        av_log(NULL, AV_LOG_ERROR,
               "Impossible to open the '%s' protocol for reading\n", up->name);
        return AVERROR(EIO);
    }
    if ((flags & AVIO_FLAG_WRITE) && !up->url_write) {
        av_log(NULL, AV_LOG_ERROR,
               "Impossible to open the '%s' protocol for writing\n", up->name);
        return AVERROR(EIO);
    }
    uc = av_mallocz(sizeof(URLContext) + strlen(filename) + 1);
    if (!uc) {
        err = AVERROR(ENOMEM);
        goto fail;
    }
    uc->av_class = &ffurl_context_class;
    uc->filename = (char *)&uc[1];
    strcpy(uc->filename, filename);
    uc->prot            = up;
    uc->flags           = flags;
    uc->is_streamed     = 0; /* default = not streamed */
    uc->max_packet_size = 0; /* default: stream file */
    if (up->priv_data_size) {
        uc->priv_data = av_mallocz(up->priv_data_size);
        if (!uc->priv_data) {
            err = AVERROR(ENOMEM);
            goto fail;
        }
        if (up->priv_data_class) {
            int proto_len= strlen(up->name);
            char *start = strchr(uc->filename, ',');
            *(const AVClass **)uc->priv_data = up->priv_data_class;
            av_opt_set_defaults(uc->priv_data);
            if(!strncmp(up->name, uc->filename, proto_len) && uc->filename + proto_len == start){
                int ret= 0;
                char *p= start;
                char sep= *++p;
                char *key, *val;
                p++;
                while(ret >= 0 && (key= strchr(p, sep)) && p<key && (val = strchr(key+1, sep))){
                    *val= *key= 0;
                    ret= av_opt_set(uc->priv_data, p, key+1, 0);
                    if (ret == AVERROR_OPTION_NOT_FOUND)
                        av_log(uc, AV_LOG_ERROR, "Key '%s' not found.\n", p);
                    *val= *key= sep;
                    p= val+1;
                }
                if(ret<0 || p!=key){
                    av_log(uc, AV_LOG_ERROR, "Error parsing options string %s\n", start);
                    av_freep(&uc->priv_data);
                    av_freep(&uc);
                    err = AVERROR(EINVAL);
                    goto fail;
                }
                memmove(start, key+1, strlen(key));
            }
        }
    }
    if (int_cb)
        uc->interrupt_callback = *int_cb;


    *puc = uc;
    return 0;
fail:
    *puc = NULL;
    if (uc)
        av_freep(&uc->priv_data);
    av_freep(&uc);
#if CONFIG_NETWORK
    if (up->flags & URL_PROTOCOL_FLAG_NETWORK)
        ff_network_close();
#endif
    return err;
}
```

url_alloc_for_protocol()完成了以下步骤：首先，检查输入的URLProtocol是否支持指定的flag。比如flag中如果指定了AVIO_FLAG_READ，则URLProtocol中必须包含url_read()；如果指定了AVIO_FLAG_WRITE，则URLProtocol中必须包含url_write()。在检查无误之后，接着就可以调用av_mallocz()为即将创建的URLContext分配内存了。接下来基本上就是各种赋值工作，在这里不再详细记录。

ffurl_connect()

ffurl_connect()用于打开获得的URLProtocol。该函数的定义位于libavformat\avio.c中，如下所示。

```cpp
int ffurl_connect(URLContext *uc, AVDictionary **options)
{
    int err =
        uc->prot->url_open2 ? uc->prot->url_open2(uc,
                                                  uc->filename,
                                                  uc->flags,
                                                  options) :
        uc->prot->url_open(uc, uc->filename, uc->flags);
    if (err)
        return err;
    uc->is_connected = 1;
    /* We must be careful here as ffurl_seek() could be slow,
     * for example for http */
    if ((uc->flags & AVIO_FLAG_WRITE) || !strcmp(uc->prot->name, "file"))
        if (!uc->is_streamed && ffurl_seek(uc, 0, SEEK_SET) < 0)
            uc->is_streamed = 1;
    return 0;
}
```

该函数最重要的函数就是它的第一句：URLProtocol中是否包含url_open2()？如果包含的话，就调用url_open2()，否则就调用url_open()。

url_open()本身是URLProtocol的一个函数指针，这个地方根据不同的协议调用的url_open()具体实现函数也是不一样的，例如file协议的url_open()对应的是file_open()，而file_open()最终调用了_wsopen()，_sopen()（Windows下）或者open()（Linux下，类似于fopen()）这样的系统中打开文件的API函数；而libRTMP的url_open()对应的是rtmp_open()，而rtmp_open()最终调用了libRTMP的API函数RTMP_Init()，RTMP_SetupURL()，RTMP_Connect() 以及RTMP_ConnectStream()。

ffio_fdopen()

ffio_fdopen()使用已经获得的URLContext初始化AVIOContext。它的函数定义位于libavformat\aviobuf.c中，如下所示。

```cpp
#define IO_BUFFER_SIZE 32768


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


    max_packet_size = h->max_packet_size;
    if (max_packet_size) {
        buffer_size = max_packet_size; /* no need to bufferize more than one packet */
    } else {
        buffer_size = IO_BUFFER_SIZE;
    }
    buffer = av_malloc(buffer_size);
    if (!buffer)
        return AVERROR(ENOMEM);


    *s = avio_alloc_context(buffer, buffer_size, h->flags & AVIO_FLAG_WRITE, h,
                            (void*)ffurl_read, (void*)ffurl_write, (void*)ffurl_seek);
    if (!*s) {
        av_free(buffer);
        return AVERROR(ENOMEM);
    }
    (*s)->direct = h->flags & AVIO_FLAG_DIRECT;
    (*s)->seekable = h->is_streamed ? 0 : AVIO_SEEKABLE_NORMAL;
    (*s)->max_packet_size = max_packet_size;
    if(h->prot) {
        (*s)->read_pause = (int (*)(void *, int))h->prot->url_read_pause;
        (*s)->read_seek  = (int64_t (*)(void *, int, int64_t, int))h->prot->url_read_seek;
    }
    (*s)->av_class = &ffio_url_class;
    return 0;
}
```

ffio_fdopen()函数首先初始化AVIOContext中的Buffer。如果URLContext中设置了max_packet_size，则将Buffer的大小设置为max_packet_size。如果没有设置的话（似乎大部分URLContext都没有设置该值），则会分配IO_BUFFER_SIZE个字节给Buffer。IO_BUFFER_SIZE取值为32768。

avio_alloc_context()

ffio_fdopen()接下来会调用avio_alloc_context()初始化一个AVIOContext。avio_alloc_context()本身是一个FFmpeg的API函数。它的声明位于libavformat\avio.h中，如下所示。

```cpp
/**
 * Allocate and initialize an AVIOContext for buffered I/O. It must be later
 * freed with av_free().
 *
 * @param buffer Memory block for input/output operations via AVIOContext.
 *        The buffer must be allocated with av_malloc() and friends.
 * @param buffer_size The buffer size is very important for performance.
 *        For protocols with fixed blocksize it should be set to this blocksize.
 *        For others a typical size is a cache page, e.g. 4kb.
 * @param write_flag Set to 1 if the buffer should be writable, 0 otherwise.
 * @param opaque An opaque pointer to user-specific data.
 * @param read_packet  A function for refilling the buffer, may be NULL.
 * @param write_packet A function for writing the buffer contents, may be NULL.
 *        The function may not change the input buffers content.
 * @param seek A function for seeking to specified byte position, may be NULL.
 *
 * @return Allocated AVIOContext or NULL on failure.
 */
AVIOContext *avio_alloc_context(
                  unsigned char *buffer,
                  int buffer_size,
                  int write_flag,
                  void *opaque,
                  int (*read_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int (*write_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int64_t (*seek)(void *opaque, int64_t offset, int whence));
```

avio_alloc_context()看上去参数很多，但实际上并不复杂。先简单解释一下它各个参数的含义：

buffer：AVIOContext中的Buffer。

buffer_size：AVIOContext中的Buffer的大小。

write_flag：设置为1则Buffer可写；否则Buffer只可读。

opaque：用户自定义数据。

read_packet()：读取外部数据，填充Buffer的函数。

write_packet()：向Buffer中写入数据的函数。

seek()：用于Seek的函数。

该函数成功执行的话则会返回一个创建好的AVIOContext。

下面看一下avio_alloc_context()的定义，位于libavformat\aviobuf.c，如下所示。

```cpp
AVIOContext *avio_alloc_context(
                  unsigned char *buffer,
                  int buffer_size,
                  int write_flag,
                  void *opaque,
                  int (*read_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int (*write_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int64_t (*seek)(void *opaque, int64_t offset, int whence))
{
    AVIOContext *s = av_mallocz(sizeof(AVIOContext));
    if (!s)
        return NULL;
    ffio_init_context(s, buffer, buffer_size, write_flag, opaque,
                  read_packet, write_packet, seek);
    return s;
}
```

该函数代码很简单：首先调用av_mallocz()为AVIOContext分配一块内存空间，然后基本上将所有输入参数传递给ffio_init_context()。

ffio_init_context()

ffio_init_context()的定义如下。

```cpp
int ffio_init_context(AVIOContext *s,
                  unsigned char *buffer,
                  int buffer_size,
                  int write_flag,
                  void *opaque,
                  int (*read_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int (*write_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int64_t (*seek)(void *opaque, int64_t offset, int whence))
{
    s->buffer      = buffer;
    s->orig_buffer_size =
    s->buffer_size = buffer_size;
    s->buf_ptr     = buffer;
    s->opaque      = opaque;
    s->direct      = 0;


    url_resetbuf(s, write_flag ? AVIO_FLAG_WRITE : AVIO_FLAG_READ);


    s->write_packet    = write_packet;
    s->read_packet     = read_packet;
    s->seek            = seek;
    s->pos             = 0;
    s->must_flush      = 0;
    s->eof_reached     = 0;
    s->error           = 0;
    s->seekable        = seek ? AVIO_SEEKABLE_NORMAL : 0;
    s->max_packet_size = 0;
    s->update_checksum = NULL;


    if (!read_packet && !write_flag) {
        s->pos     = buffer_size;
        s->buf_end = s->buffer + buffer_size;
    }
    s->read_pause = NULL;
    s->read_seek  = NULL;


    return 0;
}
```

可以看出，这个函数的工作就是各种赋值，不算很有“技术含量”，不再详述。

ffurl_read()，ffurl_write()，ffurl_seek()

现在我们再回到ffio_fdopen()，会发现它初始化AVIOContext的结构体的时候，首先将自己分配的Buffer设置为该AVIOContext的Buffer；然后将URLContext作为用户自定义数据（对应AVIOContext的opaque变量）提供给该AVIOContext；最后分别将3个函数作为该AVIOContext的读，写，跳转函数：ffurl_read()，ffurl_write()，ffurl_seek()。下面我们选择一个ffurl_read()看看它的定义。
ffurl_read()的定义位于libavformat\avio.c，如下所示。

```cpp
int ffurl_read(URLContext *h, unsigned char *buf, int size)
{
    if (!(h->flags & AVIO_FLAG_READ))
        return AVERROR(EIO);
    return retry_transfer_wrapper(h, buf, size, 1, h->prot->url_read);
}
```

该函数先判断了一下输入的URLContext是否支持“读”操作，接着调用了一个函数：retry_transfer_wrapper()。
如果我们看ffurl_write()的代码，如下所示。

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


    return retry_transfer_wrapper(h, (unsigned char *)buf, size, size, (void*)h->prot->url_write);
}
```

会发现他也调用了同样的一个函数retry_transfer_wrapper()。唯一的不同在于ffurl_read()调用retry_transfer_wrapper()的时候，最后一个参数是URLProtocol的url_read()，而ffurl_write()调用retry_transfer_wrapper()的时候，最后一个参数是URLProtocol的url_write()。
下面我们看一下retry_transfer_wrapper()的定义，位于libavformat\avio.c，如下所示。

```cpp
static inline int retry_transfer_wrapper(URLContext *h, uint8_t *buf,
                                         int size, int size_min,
                                         int (*transfer_func)(URLContext *h,
                                                              uint8_t *buf,
                                                              int size))
{
    int ret, len;
    int fast_retries = 5;
    int64_t wait_since = 0;


    len = 0;
    while (len < size_min) {
        if (ff_check_interrupt(&h->interrupt_callback))
            return AVERROR_EXIT;
        ret = transfer_func(h, buf + len, size - len);
        if (ret == AVERROR(EINTR))
            continue;
        if (h->flags & AVIO_FLAG_NONBLOCK)
            return ret;
        if (ret == AVERROR(EAGAIN)) {
            ret = 0;
            if (fast_retries) {
                fast_retries--;
            } else {
                if (h->rw_timeout) {
                    if (!wait_since)
                        wait_since = av_gettime_relative();
                    else if (av_gettime_relative() > wait_since + h->rw_timeout)
                        return AVERROR(EIO);
                }
                av_usleep(1000);
            }
        } else if (ret < 1)
            return (ret < 0 && ret != AVERROR_EOF) ? ret : len;
        if (ret)
            fast_retries = FFMAX(fast_retries, 2);
        len += ret;
    }
    return len;
}
```

从代码中可以看出，它的核心实际上是调用了一个名称为transfer_func()的函数。而该函数就是retry_transfer_wrapper()的第四个参数。该函数实际上是对URLProtocol的读写操作中的错误进行了一些“容错”处理，可以让数据的读写更加的稳定。

avio_alloc_context()执行完毕后，ffio_fdopen()函数的工作就基本完成了，avio_open2()的工作也就做完了。

雷霄骅 (Lei Xiaohua)
leixiaohua1020@126.com
http://blog.csdn.net/leixiaohua1020

文章标签： FFmpeg AVIOContext IO 源代码

个人分类： FFMPEG

所属专栏： FFmpeg