

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]
1.  /**
2.   * Create and initialize a AVIOContext for accessing the
3.   * resource indicated by url.
4.   * @note When the resource indicated by url has been opened in
5.   * read+write mode, the AVIOContext can be used only for writing.
6.   *
7.   * @param s Used to return the pointer to the created AVIOContext.
8.   * In case of failure the pointed to value is set to NULL.
9.   * @param url resource to access
10.  * @param flags flags which control how the resource indicated by url
11.  * is to be opened
12.  * @param int_cb an interrupt callback to be used at the protocols level
13.  * @param options A dictionary filled with protocol-private options. On return
14.  * this parameter will be destroyed and replaced with a dict containing options
15.  * that were not found. May be NULL.
16.  * @return >= 0 in case of success, a negative value corresponding to an
17.  * AVERROR code in case of failure
18.  */
19.  int avio_open2(AVIOContext **s, const char *url, int flags,
20.                 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。



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]
1.  int avio_open(AVIOContext **s, const char *filename, int flags)
2.  {
3.      return avio_open2(s, filename, flags, NULL, NULL);
4.  }
```

avio_open2()

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

```
[cpp]
1.  int avio_open2(AVIOContext **s, const char *filename, int flags,
2.               const AVIOInterruptCB *int_cb, AVDictionary **options)
3.  {
4.      URLContext *h;
5.      int err;
6.
7.
8.      err = ffurl_open(&h, filename, flags, int_cb, options);
9.      if (err < 0)
10.         return err;
11.      err = ffio_fdopen(s, h);
12.      if (err < 0) {
13.          ffurl_close(h);
14.          return err;
15.      }
16.      return 0;
17.  }
```

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]
1.  typedef struct URLProtocol {
2.      const char *name;
3.      int      (*url_open)( URLContext *h, const char *url, int flags);
4.      /**
5.       * This callback is to be used by protocols which open further nested
6.       * protocols. options are then to be passed to ffurl_open()/ffurl_connect()
7.       * for those nested protocols.
8.       */
9.      int      (*url_open2)(URLContext *h, const char *url, int flags, AVDictionary **options);
10.
11.
12.      /**
13.       * Read data from the protocol.
14.       * If data is immediately available (even less than size), EOF is
15.       * reached or an error occurs (including EINTR), return immediately.
16.       * Otherwise:
17.       * In non-blocking mode, return AVERRROR(EAGAIN) immediately.
18.       * In blocking mode, wait for data/EOF/error with a short timeout (0.1s),
19.       * and return AVERRROR(EAGAIN) on timeout.
20.       * Checking interrupt_callback, looping on EINTR and EAGAIN and until
21.       * enough data has been read is left to the calling function; see
22.       * retry_transfer_wrapper in avio.c.
23.       */
24.      int      (*url_read)( URLContext *h, unsigned char *buf, int size);
25.      int      (*url_write)(URLContext *h, const unsigned char *buf, int size);
26.      int64_t  (*url_seek)( URLContext *h, int64_t pos, int whence);
27.      int      (*url_close)(URLContext *h);
28.      struct URLProtocol *next;
29.      int (*url_read_pause)(URLContext *h, int pause);
30.      int64_t (*url_read_seek)(URLContext *h, int stream_index,
31.                              int64_t timestamp, int flags);
32.      int (*url_get_file_handle)(URLContext *h);
33.      int (*url_get_multi_file_handle)(URLContext *h, int **handles,
34.                                       int *numhandles);
35.      int (*url_shutdown)(URLContext *h, int flags);
36.      int priv_data_size;
37.      const AVClass *priv_data_class;
38.      int flags;
39.      int (*url_check)(URLContext *h, int mask);
40.  } 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]
1.  URLProtocol ff_file_protocol = {
2.      .name           = "file",
3.      .url_open        = file_open,
4.      .url_read         = file_read,
5.      .url_write        = file_write,
6.      .url_seek         = file_seek,
7.      .url_close        = file_close,
8.      .url_get_file_handle = file_get_handle,
9.      .url_check        = file_check,
10.     .priv_data_size    = sizeof(FileContext),
11.     .priv_data_class    = &file_class,
12. };
```

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]
1.  URLProtocol ff_librtmp_protocol = {
2.      .name           = "rtmp",
3.      .url_open        = rtmp_open,
4.      .url_read         = rtmp_read,
5.      .url_write        = rtmp_write,
6.      .url_close        = rtmp_close,
7.      .url_read_pause   = rtmp_read_pause,
8.      .url_read_seek    = rtmp_read_seek,
9.      .url_get_file_handle = rtmp_get_file_handle,
10.     .priv_data_size    = sizeof(LibRTMPContext),
11.     .priv_data_class    = &librtmp_class,
12.     .flags              = URL_PROTOCOL_FLAG_NETWORK,
13. };
```

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

```
[cpp]
1.  URLProtocol ff_udp_protocol = {
2.      .name           = "udp",
3.      .url_open        = udp_open,
4.      .url_read         = udp_read,
5.      .url_write        = udp_write,
6.      .url_close        = udp_close,
7.      .url_get_file_handle = udp_get_file_handle,
8.      .priv_data_size    = sizeof(UDPContext),
9.      .priv_data_class    = &udp_context_class,
10.     .flags              = URL_PROTOCOL_FLAG_NETWORK,
11. };
```

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]
1.  typedef struct URLContext {
2.      const AVClass *av_class;    /**< information for av_log(). Set by url_open(). */
3.      struct URLProtocol *prot;
4.      void *priv_data;
5.      char *filename;             /**< specified URL */
6.      int flags;
7.      int max_packet_size;        /**< if non zero, the stream is packetized with this max packet size */
8.      int is_streamed;            /**< true if streamed (no seek possible), default = false */
9.      int is_connected;
10.     AVIOInterruptCB interrupt_callback;
11.     int64_t rw_timeout;          /**< maximum time to wait for (network) read/write operation completion, in mcs */
12. } 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。

```

1. int ffurl_open(URLContext **puc, const char *filename, int flags,
2.               const AVIOInterruptCB *int_cb, AVDictionary **options)
3. {
4.     int ret = ffurl_alloc(puc, filename, flags, int_cb);
5.     if (ret < 0)
6.         return ret;
7.     if (options && (*puc)->prot->priv_data_class &&
8.         (ret = av_opt_set_dict((*puc)->priv_data, options)) < 0)
9.         goto fail;
10.    if ((ret = av_opt_set_dict(*puc, options)) < 0)
11.        goto fail;
12.    ret = ffurl_connect(*puc, options);
13.    if (!ret)
14.        return 0;
15. fail:
16.    ffurl_close(*puc);
17.    *puc = NULL;
18.    return ret;
19. }

```

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。

```

1. int ffurl_alloc(URLContext **puc, const char *filename, int flags,
2.               const AVIOInterruptCB *int_cb)
3. {
4.     URLProtocol *p = NULL;
5.
6.
7.     if (!first_protocol) {
8.         av_log(NULL, AV_LOG_WARNING, "No URL Protocols are registered. "
9.             "Missing call to av_register_all()?\n");
10.    }
11.
12.
13.    p = url_find_protocol(filename);
14.    if (p)
15.        return url_alloc_for_protocol(puc, p, filename, flags, int_cb);
16.
17.
18.    *puc = NULL;
19.    if (av_strstart(filename, "https:", NULL))
20.        av_log(NULL, AV_LOG_WARNING, "https protocol not found, recompile with openssl or gnutls enabled.\n");
21.    return AERROR_PROTOCOL_NOT_FOUND;
22. }

```

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。

```

1. #define URL_SCHEME_CHARS \
2.     "abcdefghijklmnopqrstuvwxyz" \
3.     "ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
4.     "0123456789+-.\"
5.
6. static struct URLProtocol *url_find_protocol(const char *filename)
7. {
8.     URLProtocol *up = NULL;
9.     char proto_str[128], proto_nested[128], *ptr;
10.    size_t proto_len = strspn(filename, URL_SCHEME_CHARS);
11.
12.
13.    if (filename[proto_len] != ':' &&
14.        (filename[proto_len] != '/' || !strchr(filename + proto_len + 1, ':')) ||
15.        is_dos_path(filename))
16.        strcpy(proto_str, "file");
17.    else
18.        av_strlcpy(proto_str, filename,
19.                    FFMIN(proto_len + 1, sizeof(proto_str)));
20.
21.
22.    if ((ptr = strchr(proto_str, ',')))
23.        *ptr = '\0';
24.    av_strlcpy(proto_nested, proto_str, sizeof(proto_nested));
25.    if ((ptr = strchr(proto_nested, '+')))
26.        *ptr = '\0';
27.
28.
29.    while (up = ffurl_protocol_next(up)) {
30.        if (!strcmp(proto_str, up->name))
31.            break;
32.        if (up->flags & URL_PROTOCOL_FLAG_NESTED_SCHEME &&
33.            !strcmp(proto_nested, up->name))
34.            break;
35.    }
36.
37.
38.    return up;
39. }

```

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。这么一张一消，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。



```

1. /**
2.  * Copy the string src to dst, but no more than size - 1 bytes, and
3.  * null-terminate dst.
4.  *
5.  * This function is the same as BSD strlcpy().
6.  *
7.  * @param dst destination buffer
8.  * @param src source string
9.  * @param size size of destination buffer
10. * @return the length of src
11. *
12. * @warning since the return value is the length of src, src absolutely
13. * _must_ be a properly 0-terminated string, otherwise this will read beyond
14. * the end of the buffer and possibly crash.
15. */
16. 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]    
1. static inline int is_dos_path(const char *path)  
2. {  
3.     #if HAVE_DOS_PATHS  
4.         if (path[0] && path[1] == ':')  
5.             return 1;  
6.     #endif  
7.     return 0;  
8. }
```

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]    
1. URLProtocol *ffurl_protocol_next(const URLProtocol *prev)  
2. {  
3.     return prev ? prev->next : first_protocol;  
4. }
```

url_alloc_for_protocol()

url_alloc_for_protocol()的定义位于libavformatlavio.c中，如下所示。

[cpp]  

```
1. static int url_alloc_for_protocol(URLContext **puc, struct URLProtocol *up,
2.                                const char *filename, int flags,
3.                                const AVIOInterruptCB *int_cb)
4. {
5.     URLContext *uc;
6.     int err;
7.
8.
9.     #if CONFIG_NETWORK
10.    if (up->flags & URL_PROTOCOL_FLAG_NETWORK && !ff_network_init())
11.        return AERROR(EIO);
12.    #endif
13.    if ((flags & AVIO_FLAG_READ) && !up->url_read) {
14.        av_log(NULL, AV_LOG_ERROR,
15.             "Impossible to open the '%s' protocol for reading\n", up->name);
16.        return AERROR(EIO);
17.    }
18.    if ((flags & AVIO_FLAG_WRITE) && !up->url_write) {
19.        av_log(NULL, AV_LOG_ERROR,
20.             "Impossible to open the '%s' protocol for writing\n", up->name);
21.        return AERROR(EIO);
22.    }
23.    uc = av_mallocz(sizeof(URLContext) + strlen(filename) + 1);
24.    if (!uc) {
25.        err = AERROR(ENOMEM);
26.        goto fail;
27.    }
28.    uc->av_class = &ffurl_context_class;
29.    uc->filename = (char *)&uc[1];
30.    strcpy(uc->filename, filename);
31.    uc->prot      = up;
32.    uc->flags     = flags;
33.    uc->is_streamed = 0; /* default = not streamed */
34.    uc->max_packet_size = 0; /* default: stream file */
35.    if (up->priv_data_size) {
36.        uc->priv_data = av_mallocz(up->priv_data_size);
37.        if (!uc->priv_data) {
38.            err = AERROR(ENOMEM);
39.            goto fail;
40.        }
41.        if (up->priv_data_class) {
42.            int proto_len = strlen(up->name);
43.            char *start = strchr(uc->filename, ',');
44.            *(const AVClass **)uc->priv_data = up->priv_data_class;
45.            av_opt_set_defaults(uc->priv_data);
46.            if (!strncmp(up->name, uc->filename, proto_len) && uc->filename + proto_len == start){
47.                int ret = 0;
48.                char *p = start;
49.                char sep = **p;
50.                char *key, *val;
51.                p++;
52.                while (ret >= 0 && (key = strchr(p, sep)) && p < key && (val = strchr(key+1, sep))){
53.                    *val = *key = 0;
54.                    ret = av_opt_set(uc->priv_data, p, key+1, 0);
55.                    if (ret == AERROR_OPTION_NOT_FOUND)
56.                        av_log(uc, AV_LOG_ERROR, "Key '%s' not found.\n", p);
57.                    *val = *key = sep;
58.                    p = val+1;
59.                }
60.                if (ret < 0 || p != key){
61.                    av_log(uc, AV_LOG_ERROR, "Error parsing options string %s\n", start);
62.                    av_freep(&uc->priv_data);
63.                    av_freep(&uc);
64.                    err = AERROR(EINVAL);
65.                    goto fail;
66.                }
67.                memmove(start, key+1, strlen(key));
68.            }
69.        }
70.    }
71.    if (int_cb)
72.        uc->interrupt_callback = *int_cb;
73.
74.
75.    *puc = uc;
76.    return 0;
77. fail:
78.    *puc = NULL;
79.    if (uc)
80.        av_freep(&uc->priv_data);
81.    av_freep(&uc);
82.    #if CONFIG_NETWORK
83.    if (up->flags & URL_PROTOCOL_FLAG_NETWORK)
84.        ff_network_close();
85.    #endif
86.    return err;
87. }
```


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]
1. int ffurl_connect(URLContext *uc, AVDictionary **options)
2. {
3.     int err =
4.         uc->prot->url_open2 ? uc->prot->url_open2(uc,
5.                                                 uc->filename,
6.                                                 uc->flags,
7.                                                 options) :
8.         uc->prot->url_open(uc, uc->filename, uc->flags);
9.     if (err)
10.        return err;
11.    uc->is_connected = 1;
12.    /* We must be careful here as ffurl_seek() could be slow,
13.     * for example for http */
14.    if ((uc->flags & AVIO_FLAG_WRITE) || !strcmp(uc->prot->name, "file"))
15.        if (!uc->is_streamed && ffurl_seek(uc, 0, SEEK_SET) < 0)
16.            uc->is_streamed = 1;
17.    return 0;
18. }
```

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]
1. #define IO_BUFFER_SIZE 32768
2.
3.
4. int ffio_fdopen(AVIOContext **s, URLContext *h)
5. {
6.     uint8_t *buffer;
7.     int buffer_size, max_packet_size;
8.
9.
10.    max_packet_size = h->max_packet_size;
11.    if (max_packet_size) {
12.        buffer_size = max_packet_size; /* no need to bufferize more than one packet */
13.    } else {
14.        buffer_size = IO_BUFFER_SIZE;
15.    }
16.    buffer = av_malloc(buffer_size);
17.    if (!buffer)
18.        return AVERROR(ENOMEM);
19.
20.
21.    *s = avio_alloc_context(buffer, buffer_size, h->flags & AVIO_FLAG_WRITE, h,
22.                           (void*)ffurl_read, (void*)ffurl_write, (void*)ffurl_seek);
23.    if (!*s) {
24.        av_free(buffer);
25.        return AVERROR(ENOMEM);
26.    }
27.    (*s)->direct = h->flags & AVIO_FLAG_DIRECT;
28.    (*s)->seekable = h->is_streamed ? 0 : AVIO_SEEKABLE_NORMAL;
29.    (*s)->max_packet_size = max_packet_size;
30.    if (h->prot) {
31.        (*s)->read_pause = (int (*)(void *, int))h->prot->url_read_pause;
32.        (*s)->read_seek = (int64_t (*)(void *, int, int64_t, int))h->prot->url_read_seek;
33.    }
34.    (*s)->av_class = &ffio_url_class;
35.    return 0;
36. }
```

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]
1.  /**
2.   * Allocate and initialize an AVIOContext for buffered I/O. It must be later
3.   * freed with av_free().
4.   *
5.   * @param buffer Memory block for input/output operations via AVIOContext.
6.   *       The buffer must be allocated with av_malloc() and friends.
7.   * @param buffer_size The buffer size is very important for performance.
8.   *       For protocols with fixed blocksize it should be set to this blocksize.
9.   *       For others a typical size is a cache page, e.g. 4kb.
10.  * @param write_flag Set to 1 if the buffer should be writable, 0 otherwise.
11.  * @param opaque An opaque pointer to user-specific data.
12.  * @param read_packet A function for refilling the buffer, may be NULL.
13.  * @param write_packet A function for writing the buffer contents, may be NULL.
14.  *       The function may not change the input buffers content.
15.  * @param seek A function for seeking to specified byte position, may be NULL.
16.  *
17.  * @return Allocated AVIOContext or NULL on failure.
18.  */
19.  AVIOContext *avio_alloc_context(
20.      unsigned char *buffer,
21.      int buffer_size,
22.      int write_flag,
23.      void *opaque,
24.      int (*read_packet)(void *opaque, uint8_t *buf, int buf_size),
25.      int (*write_packet)(void *opaque, uint8_t *buf, int buf_size),
26.      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]
1.  AVIOContext *avio_alloc_context(
2.      unsigned char *buffer,
3.      int buffer_size,
4.      int write_flag,
5.      void *opaque,
6.      int (*read_packet)(void *opaque, uint8_t *buf, int buf_size),
7.      int (*write_packet)(void *opaque, uint8_t *buf, int buf_size),
8.      int64_t (*seek)(void *opaque, int64_t offset, int whence))
9.  {
10.     AVIOContext *s = av_mallocz(sizeof(AVIOContext));
11.     if (!s)
12.         return NULL;
13.     ffio_init_context(s, buffer, buffer_size, write_flag, opaque,
14.         read_packet, write_packet, seek);
15.     return s;
16. }
```

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]
1. int ffio_init_context(AVIOContext *s,
2.     unsigned char *buffer,
3.     int buffer_size,
4.     int write_flag,
5.     void *opaque,
6.     int (*read_packet)(void *opaque, uint8_t *buf, int buf_size),
7.     int (*write_packet)(void *opaque, uint8_t *buf, int buf_size),
8.     int64_t (*seek)(void *opaque, int64_t offset, int whence))
9. {
10.     s->buffer = buffer;
11.     s->orig_buffer_size =
12.     s->buffer_size = buffer_size;
13.     s->buf_ptr = buffer;
14.     s->opaque = opaque;
15.     s->direct = 0;
16.
17.
18.     url_resetbuf(s, write_flag ? AVIO_FLAG_WRITE : AVIO_FLAG_READ);
19.
20.
21.     s->write_packet = write_packet;
22.     s->read_packet = read_packet;
23.     s->seek = seek;
24.     s->pos = 0;
25.     s->must_flush = 0;
26.     s->eof_reached = 0;
27.     s->error = 0;
28.     s->seekable = seek ? AVIO_SEEKABLE_NORMAL : 0;
29.     s->max_packet_size = 0;
30.     s->update_checksum = NULL;
31.
32.
33.     if (!read_packet && !write_flag) {
34.         s->pos = buffer_size;
35.         s->buf_end = s->buffer + buffer_size;
36.     }
37.     s->read_pause = NULL;
38.     s->read_seek = NULL;
39.
40.
41.     return 0;
42. }
```

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]
1. int ffurl_read(URLContext *h, unsigned char *buf, int size)
2. {
3.     if (!(h->flags & AVIO_FLAG_READ))
4.         return AVERROR(EIO);
5.     return retry_transfer_wrapper(h, buf, size, 1, h->prot->url_read);
6. }
```

该函数先判断了一下输入的URLContext是否支持“读”操作，接着调用了函数：retry_transfer_wrapper()。

如果我们看ffurl_write()的代码，如下所示。

```
[cpp]
1. int ffurl_write(URLContext *h, const unsigned char *buf, int size)
2. {
3.     if (!(h->flags & AVIO_FLAG_WRITE))
4.         return AVERROR(EIO);
5.     /* avoid sending too big packets */
6.     if (h->max_packet_size && size > h->max_packet_size)
7.         return AVERROR(EIO);
8.
9.
10.    return retry_transfer_wrapper(h, (unsigned char *)buf, size, size, (void*)h->prot->url_write);
11. }
```

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。

```

1. static inline int retry_transfer_wrapper(URLContext *h, uint8_t *buf,
2.                                     int size, int size_min,
3.                                     int (*transfer_func)(URLContext *h,
4.                                                         uint8_t *buf,
5.                                                         int size))
6. {
7.     int ret, len;
8.     int fast_retries = 5;
9.     int64_t wait_since = 0;
10.
11.
12.     len = 0;
13.     while (len < size_min) {
14.         if (ff_check_interrupt(&h->interrupt_callback))
15.             return AVERROR_EXIT;
16.         ret = transfer_func(h, buf + len, size - len);
17.         if (ret == AVERROR(EINTR))
18.             continue;
19.         if (h->flags & AVIO_FLAG_NONBLOCK)
20.             return ret;
21.         if (ret == AVERROR(EAGAIN)) {
22.             ret = 0;
23.             if (fast_retries) {
24.                 fast_retries--;
25.             } else {
26.                 if (h->rw_timeout) {
27.                     if (!wait_since)
28.                         wait_since = av_gettime_relative();
29.                     else if (av_gettime_relative() > wait_since + h->rw_timeout)
30.                         return AVERROR(EIO);
31.                 }
32.                 av_usleep(1000);
33.             }
34.         } else if (ret < 1)
35.             return (ret < 0 && ret != AVERROR_EOF) ? ret : len;
36.         if (ret)
37.             fast_retries = FFMAX(fast_retries, 2);
38.         len += ret;
39.     }
40.     return len;
41. }

```

从代码中可以看出，它的核心实际上是调用了—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>

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/41199947>

文章标签： [FFmpeg](#) [AVIOContext](#) [IO](#) [源代码](#)

个人分类： [FFMPEG](#)

所属专栏： [FFmpeg](#)

此PDF由spygg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com