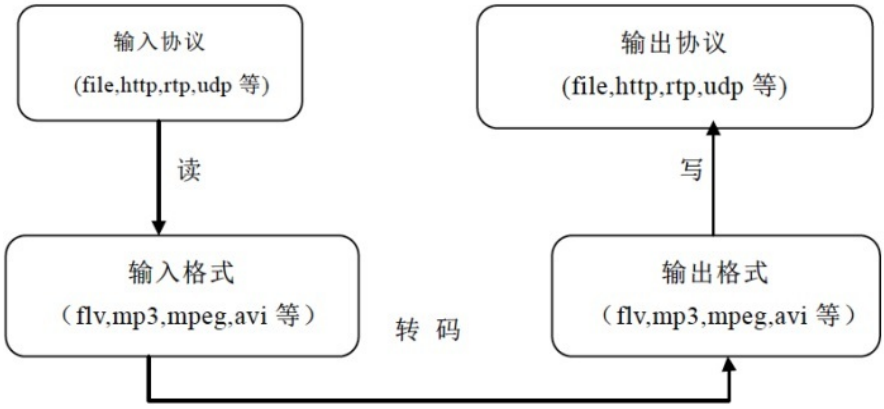
转 FFmpeg架构之I/O模块分析

2013年10月16日 16:10:04　阅读数：9416

注意：这篇转载的文章比较早，写得很清晰，但是新版的ffmpeg的很多数据结构的名字已经改了。因此只能作参考。（例如ByteIOContext已经改名为AVIOContext）

1概述

ffmpeg项目的数据IO部分主要是在libavformat库中实现，某些对于内存的操作部分在libavutil库中。数据IO是基于文件格式（Format）以及文件传输协议(Protocol)的，与具体的编解码标准无关。 ffmpeg工程转码时数据IO层次关系如图所示：



ffmpeg 转码数据 IO 流程

对于上面的数据IO流程，具体可以用下面的例子来说明，我们从一个http服务器获取音视频数据，格式是flv的，需要通过转码后变成avi格式，然后通过udp协议进行发布。

其过程就如下所示：

1、读入http协议数据流，根据http协议获取真正的文件数据（去除无关报文信息）；

2、根据flv格式对数据进行解封装；

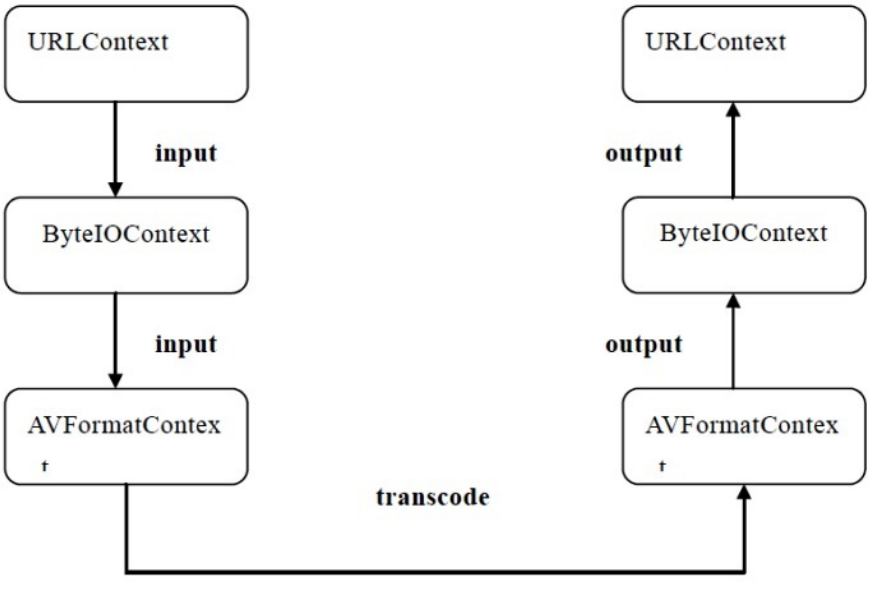
3、读取帧进行转码操作；

4、按照目标格式avi进行封装；

5、通过udp协议发送出去。

2相关数据结构介绍

在libavformat库中与数据IO相关的数据结构主要有URLProtocol、URLContext、ByteIOContext、AVFormatContext等，各结构之间的关系如图所示。



libavformat 库中 IO 相关数据结构之间的关系

1、URLProtocol结构

表示广义的输入文件，该结构体提供了很多的功能函数，每一种广义的输入文件（如：file、pipe、tcp、rtp等等）对应着一个URLProtocol结构，在av_register_all()中将该结构

体初始化为一个链表，表头为avio.c里的URLProtocol *first_protocol = NULL;保存所有支持的输入文件协议，该结构体的定义如下：

```cpp
typedef struct URLProtocol
{
const char *name;
int (*url_open)(URLContext *h, const char *url, int flags);
int (*url_read)(URLContext *h, unsigned char *buf, int size);
int (*url_write)(URLContext *h, const unsigned char *buf, int size);
int64_t (*url_seek)(URLContext *h, int64_t pos, int whence);
int (*url_close)(URLContext *h); struct URLProtocol *next;
int (*url_read_pause)(URLContext *h, int pause);
int64_t (*url_read_seek)(URLContext *h, int stream_index,
        int64_t timestamp, int flags);
int (*url_get_file_handle)(URLContext *h);
int priv_data_size;
const AVClass *priv_data_class;
int flags;
int (*url_check)(URLContext *h, int mask);
} URLProtocol;
```

注意到，URLProtocol是一个链表结构，这是为了协议的统一管理，ffmpeg项目中将所有的用到的协议都存放在一个全局变量first_protocol中，协议的注册是在av_register_all中完成的，新添加单个协议可以调用av_register_protocol2函数实现。而协议的注册就是将具体的协议对象添加至first_protocol链表的末尾。 URLProtocol在各个具体的文件协议中有一个具体的实例，如在file协议中定义为：

```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,
};
```

2、URLContext结构

URLContext提供了与当前打开的具体的文件协议（URL）相关数据的描述，在该结构中定义了指定当前URL（即filename项）所要用到的具体的URLProtocol，即：提供了一个在URLprotocol链表中找到具体项的依据，此外还有一些其它的标志性的信息，如flags， is_streamed等。它可以看成某一种协议的载体。其结构定义如下：

```cpp
typedef struct URLContext
{
const AVClass *av_class; ///< information for av_log(). Set by url_open().
struct URLProtocol *prot;
int flags;
int is_streamed; //< true if streamed (no seek possible), default = false * int max_packet_size;
void *priv_data;
char *filename; //< specified URL
int is_connected;
} URLContext;
```

那么ffmpeg依据什么信息初始化URLContext？然后又是如何初始化URLContext的呢？在打开一个URL时，全局函数ffurl_open会根据filename的前缀信息来确定URL所使用的具体协议，并为该协议分配好资源，再调用ffurl_connect函数打开具体协议，即调用协议的url_open，调用关系如下：

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

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

static int init_input(AVFormatContext *s， const char *filename)

int avio_open(AVIOContext **s， const char *filename， int flags)

int ffurl_open(URLContext **puc， const char *filename， int flags)

int ffurl_alloc(URLContext **puc， const char *filename， int flags)

static int url_alloc_for_protocol (URLContext **puc， struct URLProtocol *up， const char *filename， int flags)

浅蓝色部分的函数完成了URLContext函数的初始化，URLContext使ffmpeg外所暴露的接口是统一的，而不是对于不同的协议用不同的函数，这也是面向对象思维的体现。在此结构中还有一个值得说的是priv_data项，这是结构的一个可扩展项，具体协议可以根据需要添加相应的结构，将指针保存在这就行。

3、AVIOContext结构

AVIOContext（即：ByteIOContext）是由URLProtocol和URLContext结构扩展而来，也是ffmpeg提供给用户的接口，它将以上两种不带缓冲的读取文件抽象为带缓冲

的读取和写入，为用户提供带缓冲的读取和写入操作。数据结构定义如下：

```cpp
typedef struct {
unsigned char *buffer; /**< Start of the buffer. */
int buffer_size; /**< Maximum buffer size */
unsigned char *buf_ptr; /**< Current position in the buffer */
unsigned char *buf_end;
void *opaque; /关联URLContext
int (*read_packet)(void *opaque, uint8_t *buf, int buf_size);
int (*write_packet)(void *opaque, uint8_t *buf, int buf_size);
int64_t (*seek)(void *opaque, int64_t offset, int whence);
int64_t pos;
int must_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;
int (*read_pause)(void *opaque, int pause)
int64_t (*read_seek)(void *opaque, int stream_index, int64_t timestamp, int flags);
int seekable;
} AVIOContext;
```

结构简单的为用户提供读写容易实现的四个操作，read_packet write_packet read_pause read_seek，极大的方便了文件的读取，四个函数在加了缓冲机制后被中转到，URLContext指向的实际的文件协议读写函数中。 下面给出0.8版本中是如何将AVIOContext的读写操作中转到实际文件中的。 在avio_open（）函数中调用了ffio_fdopen（）函数完成了对AVIOContex的初始化，其调用过程如下：

int avio_open(AVIOContext **s， const char *filename， int flags)

ffio_fdopen(s， h); //h是URLContext指针 ffio_init_context(*s， buffer， buffer_size， h->flags & AVIO_FLAG_WRITE， h， (void*)

ffurl_read，(void*)ffurl_write，(void*)ffurl_seek)

蓝色部分的函数调用完成了对AVIOContext的初始化，在初始化的过程中，将AVIOContext的read_packet、write_packet、seek分别初始化为：ffurl_read ffurl_write ffurl_seek，而这三个函数又将具体的读写操作中转为：h->prot->url_read、h->prot->url_write、h->prot->url_seek，另外两个变量初始化时也被相应的中转，如下：

(*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;

所以，可以简要的描述为：AVIOContext的接口口是加了缓冲后的URLProtocol的函数接口。

在aviobuf.c中定义了一系列关于ByteIOContext这个结构体的函数，如下 put_xxx系列：

```cpp
put_xxx系列：
void put_byte(ByteIOContext *s， int b);
void put_buffer(ByteIOContext *s， const unsigned char *buf， int size);
void put_le64(ByteIOContext *s， uint64_t val);
void put_be64(ByteIOContext *s， uint64_t val);
void put_le32(ByteIOContext *s， unsigned int val);
void put_be32(ByteIOContext *s， unsigned int val);
void put_le24(ByteIOContext *s， unsigned int val);
void put_be24(ByteIOContext *s， unsigned int val);
void put_le16(ByteIOContext *s， unsigned int val);
void put_be16(ByteIOContext *s， unsigned int val);
void put_tag(ByteIOContext *s， const char *tag);
get_xxx系列：
int get_buffer(ByteIOContext *s， unsigned char *buf， int size);
int get_partial_buffer(ByteIOContext *s， unsigned char *buf， int size);
int get_byte(ByteIOContext *s);
unsigned int get_le24(ByteIOContext *s);
unsigned int get_le32(ByteIOContext *s);
uint64_t get_le64(ByteIOContext *s);
unsigned int get_le16(ByteIOContext *s);
char *get_strz(ByteIOContext *s， char *buf， int maxlen);
unsigned int get_be16(ByteIOContext *s);
unsigned int get_be24(ByteIOContext *s);
unsigned int get_be32(ByteIOContext *s);
uint64_t get_be64(ByteIOContext *s);
```

这些put_xxx及get_xxx函数是用于从缓冲区buffer中写入或者读取若干个字节，对于读写整型数据，分别实现了大端和小端字节序的版本。而缓冲区buffer中的数据又是从何而来呢，有一个fill_buffer的函数，在fill_buffer函数中调用了ByteIOContext结构的read_packet接口。在调用put_xxx函数时，并没有直接进行真正写入操作，而是先缓存起来，直到缓存达到最大限制或调用flush_buffer函数对缓冲区进行刷新，才使用write_packet函数进行写入操作。

原文地址：

文章标签： ffmpeg IO 源代码 分析

个人分类： FFMPEG

所属专栏： FFmpeg