一、 FFMPEG 中 MPEG2 TS 流解码的流程分析

说道具体的音频或者视频格式，一上来就是理论，那是国内混资历的所谓教授的做为，对于我们，不合适，还是用自己的方式理解这些晦涩不已的理论吧。

其实 MPEG2 是一族协议，至少已经成为 ISO 标准的就有以下几部分:

ISO/IEC－13818－1：系统部分；
ISO/IEC－13818－2： 视频编码格式；
ISO/IEC－13818－3：音频编码格式；
ISO/IEC－13818－4：一致性测试；
ISO/IEC－13818－5： 软件部分；
ISO/IEC－13818－6：数字存储媒体命令与控制；
ISO/IEC－13818－7：高级音频编码；
ISO/IEC－13818－8： 系统解码实时接口；

我不是很想说实际的音视频编码格式，毕竟协议已经很清楚了，我主要想说说这些部分怎么组合起来在实际应用中工作的。

第一部分(系统部分)很重要，是构成以 MPEG2 为基础的应用的基础. 很绕口，是吧，我简单解释一下:比如 DVD 实际上是以系统部分定义 的 PS 流为基础，加上版权管理等其他技术构成的。而我们的故事主角，则是另外一种流格式，TS 流，它在现阶段最大的应用是在数字电视节目 的传输与存储上，因此，你可以理解 TS 实际上是一种传输协议，与实际传输的负载关系不大，只是在 TS 中传输了音频，视频或者其他数据。先说一下为什么会有这两种格式的出现，PS 适用于没有损耗的环境下面存储，而 TS 则适用于可能出现损耗或者错误的各种物理网络环境,比如你 在公交上看到的电视,很有可能就是基于 TS 的 DVB-T 的应用:)

我们再来看 MPEG2 协议中的一些概念，为理解代码做好功课:

- **ES(Elementary Stream)**:

wiki 上说“An elementary stream (ES) is defined by MPEG communication protocol is usually the output of an audio or video encoder”

恩，很简单吧，就是编码器编出的一组数据，可能是音频的，视频的，或者其他数据。说到着，其实可以对编码器的流程思考一下，无非是执行：采样，量化，编码这 3 个步骤中的编码而已(有些设备可能会包含前面的采样和量化)。 关于视频编码的基本理论，还是请参考其它的资料。

- **PES(Packetized Elementary Stream):**

wiki 上说“allows an Elementary stream to be divided into packets”

其实可以理解成，把一个源源不断的数据(音频，视频或者其他)流，打断成 一段一段，以便处理.

- **TS(Transport Stream)**:
- **PS(Program Stream):**

这两个上面已经有所提及，后面会详细分析 TS,我对 PS 格式兴趣不大.

步入正题

才进入正题，恩，看来闲话太多了:(,直接看 Code.

前面说过，TS 是一种传输协议，因此，对应到 FFmpeg,可以认为他是一种封装格式。

因此，对应的代码应该先去 libavformat 里面找，很容易找到，就是 mpegts.c:)。还是逐步看过来:

```
[libavformat/utils.c]
int av_open_input_file(AVFormatContext **ic_ptr, const char *filename,
                       AVInputFormat *fmt,
                       int buf_size,
                       AVFormatParameters *ap)
{
    int err, probe_size;
    AVProbeData probe_data, *pd = &probe_data;
    ByteIOContext *pb = NULL;
    pd->filename = "";
    if (filename)
        pd->filename = filename;
    pd->buf = NULL;
    pd->buf_size = 0;
    ////////////////////////////////////////////////////////////////////////////
    【1】这段代码其实是为了针对不需要 Open 文件的容器 Format 的探测，其实就是使用
    AVFMT_NOFILE 标记的容器格式单独处理，现在只有使用了该标记的 Demuxer 很少，
    只有 image2_demuxer，rtsp_demuxer，因此我们分析 TS 时候可以不考虑这部分
    ////////////////////////////////////////////////////////////////////////////
    if (!fmt) {
        /* guess format if no file can be opened */
        fmt = av_probe_input_format(pd, 0);
    }
    /* Do not open file if the format does not need it. XXX: specific
       hack needed to handle RTSP/TCP */
    if (!fmt || !(fmt->flags & AVFMT_NOFILE)) {
        /* if no file needed do not try to open one */
        ////////////////////////////////////////////////////////////////////
        【2】这个函数似乎很好理解，无非是带缓冲的 IO 的封装，不过我们既然到此了
        ，不妨跟踪下去，看看别人对带缓冲的 IO 操作封装的实现:)
        ////////////////////////////////////////////////////////////////////
        if ((err=url_fopen(&pb, filename, URL_RDONLY)) < 0) {
            goto fail;
        }
        if (buf_size > 0) {
            url_setbufsize(pb, buf_size);
        }
        for(probe_size= PROBE_BUF_MIN; probe_size<=PROBE_BUF_MAX && !fmt; probe_size<<=1){
            int score= probe_size < PROBE_BUF_MAX ? AVPROBE_SCORE_MAX/4 : 0;
            /* read probe data */
            pd->buf= av_realloc(pd->buf, probe_size + AVPROBE_PADDING_SIZE);
            ////////////////////////////////////////////////////////////////
            【3】真正将文件读入到 pd 的 buffer 的地方，实际上最终调用 FILE protocol
            的 file_read(),将内容读入到 pd 的 buf，具体代码如果有兴趣可以自己跟踪
            ////////////////////////////////////////////////////////////////
            pd->buf_size = get_buffer(pb, pd->buf, probe_size);
            memset(pd->buf+pd->buf_size, 0, AVPROBE_PADDING_SIZE);
            if (url_fseek(pb, 0, SEEK_SET) < 0) {
                url_fclose(pb);
                if (url_fopen(&pb, filename, URL_RDONLY) < 0) {
                    pb = NULL;
                    err = AVERROR(EIO);
                    goto fail;
                }
            }
            ////////////////////////////////////////////////////////////////
            【4】此时的 pd 已经有了需要分析的原始文件，只需要查找相应容器 format
            的 Tag 比较，以判断读入的究竟为什么容器格式，这里
            ////////////////////////////////////////////////////////////////
            /* guess file format */
            fmt = av_probe_input_format2(pd, 1, &score);
        }
        av_freep(&pd->buf);
    }
    /* if still no format found, error */
    if (!fmt) {
```

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

【2】带缓冲 IO 的封装的实现

```
[liavformat/aviobuf.c]
int url_fopen(ByteIOContext **s, const char *filename, int flags)
{
    URLContext *h;
    int err;
    err = url_open(&h, filename, flags);
    if (err < 0)
        return err;
    err = url_fdopen(s, h);
    if (err < 0) {
        url_close(h);
        return err;
    }
    return 0;
}
```

可以看到，下面的这个函数，先查找是否是 FFmpeg 支持的 protocol 的格式，如果文件名不符合，则默认是 FILE protocol 格式，很显然，这里 protocol 判断是以 URL 的方式判读的，因此基本上所有的 IO 接口函数都是 url_xxx 的形式。

在这也可以看到，FFmpeg 支持的 protocol 有：

```
/* protocols */
REGISTER_PROTOCOL (FILE, file);
REGISTER_PROTOCOL (HTTP, http);
REGISTER_PROTOCOL (PIPE, pipe);
REGISTER_PROTOCOL (RTP, rtp);
REGISTER_PROTOCOL (TCP, tcp);
REGISTER_PROTOCOL (UDP, udp);
```

而大部分情况下，如果你不指明类似 file://xxx,http://xxx 格 式，它都以 FILE protocol 来处理。

```
[liavformat/avio.c]
int url_open(URLContext **puc, const char *filename, int flags)
{
    URLProtocol *up;
    const char *p;
    char proto_str[128], *q;
    p = filename;
    q = proto_str;
    while (*p != '\0' && *p != ':') {
        /* protocols can only contain alphabetic chars */
```

```
        if (!isalpha(*p))
            goto file_proto;
        if ((q - proto_str) < sizeof(proto_str) - 1)
            *q++ = *p;
        p++;
    }
    /* if the protocol has length 1, we consider it is a dos drive */
    if (*p == '\0' || (q - proto_str) <= 1) {
    file_proto:
        strcpy(proto_str, "file");
    } else {
        *q = '\0';
    }
    up = first_protocol;
    while (up != NULL) {
        if (!strcmp(proto_str, up->name))
            ##################################################################
            很显然，此时已经知道 up，filename，flags
            ##################################################################
            return url_open_protocol (puc, up, filename, flags);
        up = up->next;
    }
    *puc = NULL;
    return AVERROR(ENOENT);
}
[libavformat/avio.c]
int url_open_protocol (URLContext **puc, struct URLProtocol *up,
                       const char *filename, int flags)
{
    URLContext *uc;
    int err;

    ###############################################################################
    【a】？ 为什么这样分配空间
    ###############################################################################
    uc = av_malloc(sizeof(URLContext) + strlen(filename) + 1);
    if (!uc) {
        err = AVERROR(ENOMEM);
        goto fail;
    }
#if LIBAVFORMAT_VERSION_MAJOR >= 53
    uc->av_class = &urlcontext_class;
#endif
    ###############################################################################
    【b】？ 这样的用意又是为什么
    ###############################################################################
    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 */
    err = up->url_open(uc, filename, flags);
    if (err < 0) {
        av_free(uc);
        *puc = NULL;
        return err;
    }
    //We must be carefull here as url_seek() could be slow, for example for
    //http
    if(    (flags & (URL_WRONLY | URL_RDWR))
       || !strcmp(up->name, "file"))
        if(!uc->is_streamed && url_seek(uc, 0, SEEK_SET) < 0)
            uc->is_streamed= 1;
    *puc = uc;
    return 0;
 fail:
    *puc = NULL;
    return err;
```

```
}
```

上面这个函数不难理解，但有些地方颇值得玩味，比如，上面给出问号的地方，你明白为什么这样 Coding 么 :) 很显然，此时 up->url_open() 实际上调用的是 file_open()[libavformat/file.c]，看完这个函数，对上面的内存分配，是否恍然大悟:)

上面只是分析了 url_open()，还没有分析 url_fdopen(s, h);这部分代码，也留给有好奇心的你了:)恩，为了追踪这个流程，走得有些远，但不是全然无用:)

终于来到了【4】，我们来看 MPEG TS 格式的侦测过程，这其实才是我们今天的主角

4. MPEG TS 格式的探测过程

```
[liavformat/mpegts.c]
static int mpegts_probe(AVProbeData *p)
{
#if 1
    const int size= p->buf_size;
    int score, fec_score, dvhs_score;
#define CHECK_COUNT 10
    if (size < (TS_FEC_PACKET_SIZE * CHECK_COUNT))
        return -1;
    score     = analyze(p->buf, TS_PACKET_SIZE     *CHECK_COUNT, TS_PACKET_SIZE, NULL);
    dvhs_score   = analyze(p->buf, TS_DVHS_PACKET_SIZE     *CHECK_COUNT, TS_DVHS_PACKET_SIZE, NULL);
    fec_score= analyze(p->buf, TS_FEC_PACKET_SIZE*CHECK_COUNT, TS_FEC_PACKET_SIZE, NULL);
//    av_log(NULL, AV_LOG_DEBUG, "score: %d, dvhs_score: %d, fec_score: %d \n", score, dvhs_score, fec_score);
// we need a clear definition for the returned score otherwise things will become messy sooner or later
    if      (score > fec_score && score > dvhs_score && score > 6) return AVPROBE_SCORE_MAX + score      - CHECK_COUNT;
    else if(dvhs_score > score && dvhs_score > fec_score && dvhs_score > 6) return AVPROBE_SCORE_MAX + dvhs_score   -
CHECK_COUNT;
    else if(                fec_score > 6) return AVPROBE_SCORE_MAX + fec_score - CHECK_COUNT;
    else                                   return -1;
#else
    /* only use the extension for safer guess */
    if (match_ext(p->filename, "ts"))
        return AVPROBE_SCORE_MAX;
    else
        return 0;
#endif
}
```

之所以会出现 3 种格式，主要原因是：TS 标准是 188Bytes,而小日本自己又弄了一个 192Bytes 的 DVH-S 格式，第三种的 204Bytes 则是在 188Bytes 的基础上，加上 16Bytes 的 FEC(前向纠错).

```
static int analyze(const uint8_t *buf, int size, int packet_size, int *index)
{
    int stat[packet_size];
    int i;
    int x=0;
    int best_score=0;
    memset(stat, 0, packet_size*sizeof(int));

    ////////////////////////////////////////////////////////////////////////////
    由于查找的特定格式至少 3 个 Bytes，因此，至少最后 3 个 Bytes 不用查找
    ////////////////////////////////////////////////////////////////////////////
    for(x=i=0; i<size-3; i++){
        ////////////////////////////////////////////////////////////////////////
        参看后面的协议说明
        ////////////////////////////////////////////////////////////////////////
        if(buf[i] == 0x47 && !(buf[i+1] & 0x80) && (buf[i+3] & 0x30)){
            stat[x]++;
            if(stat[x] > best_score){
                best_score= stat[x];
                if(index) *index= x;
            }
        }
        x++;
        if(x == packet_size) x= 0;
```

```
    }
    return best_score;
}
```

这个函数简单说来，是在 size 大小的 buf 中，寻找满足特定格式，长度为 packet_size 的 packet 的个数，显然，返回的值越 大越可能是相应的格式(188/192/204)，其中的这个特定格式，其实就是协议的规定格式：

Syntax	No. of bits	Mnemonic
transport_packet(){		
sync_byte	8	bslbf
transport_error_indicator	1	bslbf
payload_unit_start_indicator	1	bslbf
transport_priority	1	bslbf
PID	13	uimsbf
transport_scrambling_control	2	bslbf
adaptation_field_control	2	bslbf
continuity_counter	4	uimsbf
if(adaptation_field_control=='10' || adaptation_field_control=='11'){		
adaptation_field()		
}		
if(adaptation_field_control=='01' || adaptation_field_control=='11') {		
for (i=0;i<N;i++){		
data_byte	8	bslbf
}		
}		
}		

其中的 sync_byte 固定为 0x47,即上面的: buf[i] == 0x47

由于 transport_error_indicator 为 1 的 TS Packet 实际有错误，表示携带的数据无意义，这样的 Packet 显然没什么 意义，因此: !(buf[i+1] & 0x80)

对于 adaptation_field_control，如果为取值为 0x00,则表示为未来保留，现在不用，因此：buf[i+3] & 0x30

这就是 MPEG TS 的侦测过程，很简单吧:)

后面我们分析如何从 mpegts 文件中获取 stream 的过程，待续.....

5.渐入佳境

恩，前面的基础因该已近够了，有点像手剥洋葱头的感觉，我们来看看针对 MPEG TS 的相应解析过程。我们后面的代码，主要集中在[libavformat/mpegts.c]里面，毛爷爷说：集中优势兵力打围歼，恩，开始吧，蚂蚁啃骨头。

```
static int mpegts_read_header(AVFormatContext *s,
                              AVFormatParameters *ap)
{
    MpegTSContext *ts = s->priv_data;
    ByteIOContext *pb = s->pb;
    uint8_t buf[1024];
    int len;
    int64_t pos;
    .....
    /* read the first 1024 bytes to get packet size */
    ########################################################################
    【1】有了前面分析缓冲 IO 的经历，下面的代码就不是什么问题了:)
```

```
    ###################################################################
    pos = url_ftell(pb);
    len = get_buffer(pb, buf, sizeof(buf));
    if (len != sizeof(buf))
        goto fail;
     ###################################################################
     【2】前面侦测文件格式时候其实已经知道 TS 包的大小了，这里又侦测一次，其实
    有些多余，估计是因为解码框架的原因，已近侦测的包大小没能从前面被带过来，
    可见框架虽好，却也会带来或多或少的一些不利影响
    ###################################################################
    ts->raw_packet_size = get_packet_size(buf, sizeof(buf));
    if (ts->raw_packet_size <= 0)
        goto fail;
    ts->stream = s;
    ts->auto_guess = 0;

    if (s->iformat == &mpegts_demuxer) {
        /* normal demux */

        /* first do a scaning to get all the services */
        url_fseek(pb, pos, SEEK_SET);
        ###################################################################
         【3】
        ###################################################################
        mpegts_scan_sdt(ts);
    ###################################################################
         【4】
        ###################################################################
        mpegts_set_service(ts);
    ###################################################################
         【5】
        ###################################################################
        handle_packets(ts, s->probesize);
        /* if could not find service, enable auto_guess */

        ts->auto_guess = 1;

#ifdef DEBUG_SI
        av_log(ts->stream, AV_LOG_DEBUG, "tuning done\n");
#endif
        s->ctx_flags |= AVFMTCTX_NOHEADER;
    } else {
        ......
    }

    url_fseek(pb, pos, SEEK_SET);
    return 0;
 fail:
    return -1;
}
```

这里简单 说一下 MpegTSContext *ts，从上面可以看到，其实这是为了解码不同容器格式所使用的私有数据，只有在相应的诸如 mpegts.c 文件才可以使用的，这样，增加了这个库的模块化，而模块化的最大好处，则在于把问题集中到了一个很小的有限区域里面，如果你自己构造程序 时候，不妨多参考其基本思想--这样的化，你之后的代码，还有你之后的生活，都将轻松许多。

【3】【4】其实调用的是同一个函 数：mpegts_open_section_filter()

我们来看看意欲何为。

```
static
MpegTSFilter *mpegts_open_section_filter(MpegTSContext *ts, unsigned int pid,
                                         SectionCallback *section_cb,
                                         void *opaque,
                                         int check_crc)
{
    MpegTSFilter *filter;
    MpegTSSectionFilter *sec;
```

```
#ifdef DEBUG_SI
    av_log(ts->stream, AV_LOG_DEBUG, "Filter: pid=0x%x\n", pid);
#endif
    if (pid >= NB_PID_MAX || ts->pids[pid])
        return NULL;
    filter = av_mallocz(sizeof(MpegTSFilter));
    if (!filter)
        return NULL;
    ts->pids[pid] = filter;
    filter->type = MPEGTS_SECTION;
    filter->pid = pid;
    filter->last_cc = -1;
    sec = &filter->u.section_filter;
    sec->section_cb = section_cb;
    sec->opaque = opaque;
    sec->section_buf = av_malloc(MAX_SECTION_SIZE);
    sec->check_crc = check_crc;
    if (!sec->section_buf) {
        av_free(filter);
        return NULL;
    }
    return filter;
}
```

要完全明白这部分代码，其实需要分析作者对数据结构的定义：

依次为：

```
    struct MpegTSContext;
            |
            V
    struct MpegTSFilter;
            |
            V
+---------------+---------------+
|                               |
V                               V
MpegTSPESFilter     MpegTSSectionFilter
```

其实很简单，就是 struct MpegTSContext；中有 NB_PID_MAX(8192)个 TS 的 Filter，而每个 struct MpegTSFilter 可能是 PES 的 Filter 或者 Section 的 Filter。

我们先说为什么是 8192,在前面的分析中：

给出过 TS 的语法结构：

```
Syntax                          No. of bits     Mnemonic
transport_packet(){
    sync_byte                   8               bslbf
    transport_error_indicator   1               bslbf
    payload_unit_start_indicator 1              bslbf
    transport_priority          1               bslbf
    PID                         13              uimsbf
    transport_scrambling_control 2              bslbf
    adaptation_field_control    2               bslbf
    continuity_counter          4               uimsbf
    if(adaptation_field_control=='10'
      || adaptation_field_control=='11'){
```

```
        adaptation_field()
    }
    if(adaptation_field_control=='01'
      || adaptation_field_control=='11') {
          for (i=0;i<N;i++){
              data_byte    8        bslbf
        }
    }
}
```

而 8192,则是 2^13=8192(PID)的最大数目，而为什么会有 PES 和 Section 的区分，请参考 ISO/IEC-13818-1,我实在不太喜欢重复已有的东西.

可见【3】【4】,就是挂载了两个 Section 类型的过滤器,其实在 TS 的两种负载中，section 是 PES 的元数据，只有先解析了 section,才能进一步解析 PES 数据，因此先挂上 section 的过滤器。

挂载上了两种 section 过滤器，如下：

PID	Section Name	Callback
SDT_PID(0x0011)	ServiceDescriptionTable	sdt_cb
PAT_PID(0x0000)	ProgramAssociationTable	pat_cb

既 然自是挂上 Callback，自然是在后面的地方使用，因此，我们还是继续

【5】处的代码看看是最重要的地方了，简单看来：

```
handle_packets()
   |
   +->read_packet()
   |
   +->handle_packet()
      |
      +->write_section_data()
```

read_packet()很简 单，就是去找 sync_byte(0x47),而看来 handle_packet()才会是我们真正因该关注的地方了:)

这个函数很重要，我们贴出代码，以备分析：

```
/* handle one TS packet */
static void handle_packet(MpegTSContext *ts, const uint8_t *packet)
{
    AVFormatContext *s = ts->stream;
    MpegTSFilter *tss;
    int len, pid, cc, cc_ok, afc, is_start;
    const uint8_t *p, *p_end;

    ############################################################
    获取该包的 PID
    ############################################################
    pid = AV_RB16(packet + 1) & 0x1fff;
    if(pid && discard_pid(ts, pid))
        return;
    ############################################################
    是否是 PES 或者 Section 的开头(payload_unit_start_indicator)
    ############################################################
```

```
is_start = packet[1] & 0x40;
tss = ts->pids[pid];

############################################################
ts->auto_guess 此时为 0，因此不考虑下面的代码
############################################################
if (ts->auto_guess && tss == NULL && is_start) {
   add_pes_stream(ts, pid, -1, 0);
   tss = ts->pids[pid];
}
if (!tss)
   return;

############################################################
代码说的很清楚，虽然检查，但不利用检查的结果
############################################################
/* continuity check (currently not used) */
cc = (packet[3] & 0xf);
cc_ok = (tss->last_cc < 0) || ((((tss->last_cc + 1) & 0x0f) == cc));
tss->last_cc = cc;

############################################################
跳到 adaptation_field_control
############################################################
/* skip adaptation field */
afc = (packet[3] >> 4) & 3;
p = packet + 4;
if (afc == 0) /* reserved value */
   return;
if (afc == 2) /* adaptation field only */
   return;
if (afc == 3) {
   /* skip adapation field */
   p += p[0] + 1;
}

############################################################
p 已近到达 TS 包中的有效负载的地方
############################################################
/* if past the end of packet, ignore */
p_end = packet + TS_PACKET_SIZE;
if (p >= p_end)
   return;

ts->pos47= url_ftell(ts->stream->pb) % ts->raw_packet_size;

if (tss->type == MPEGTS_SECTION) {
   if (is_start) {
      ############################################################
      针对 Section，符合部分第一个字节为 pointer field，该字段如果为 0,
      则表示后面紧跟着的是 Section 的开头，否则是某 Section 的 End 部分和
      另一 Section 的开头，因此，这里的流程实际上由两个值 is_start
      (payload_unit_start_indicator)和 len(pointer field)一起来决定
      ############################################################
      /* pointer field present */
      len = *p++;
      if (p + len > p_end)
         return;
      if (len && cc_ok) {
         ############################################################
         1).is_start == 1
           len > 0
         负载部分由 A Section 的 End 部分和 B Section 的 Start 组成，把 A 的
         End 部分写入
         ############################################################
         /* write remaining section bytes */
         write_section_data(s, tss,
                            p, len, 0);
         /* check whether filter has been closed */
```

```
            if (!ts->pids[pid])
                return;
        }
        p += len;
        if (p < p_end) {
            ##########################################################
            2).is_start == 1
              len > 0
            负载部分由 A Section 的 End 部分和 B Section 的 Start 组成，把 B 的
            Start 部分写入
            或者：
            3).
              is_start == 1
              len == 0
            负载部分仅是一个 Section 的 Start 部分，将其写入
            ##########################################################
            write_section_data(s, tss,
                               p, p_end - p, 1);
        }
    } else {
        if (cc_ok) {
            ##########################################################
            4).is_start == 0
            负载部分仅是一个 Section 的中间部分部分，将其写入
            ##########################################################
            write_section_data(s, tss,
                               p, p_end - p, 0);
        }
    }
} else {
    ###########################################################
    如果是 PES 类型，直接调用其 Callback，但显然，只有 Section 部分
    解析完成后才可能解析 PES
    ###########################################################
    tss->u.pes_filter.pes_cb(tss,
                             p, p_end - p, is_start);
}
}
```

write_section_data()函数则反 复收集 buffer 中的数据，指导完成相关 Section 的重组过程，然后调用之前注册的两个 section_cb:

后面我们将 分析之前挂在的两个 section_cb，待续.....

二、 mpegts.c 文件分析

1 综述

ffmpeg 框架对应 MPEG-2 TS 流的解析的代码在 mpegts.c 文件中，该文件有两个解复用的实例：mpegts_demuxer 和 mpegtsraw_demuxer，mpegts_demuxer 对应的真实的 TS 流格式，也就是机顶盒直接处理的 TS 流，本文主要分析和该种格式相关 的代码；mpegtsraw_demuxer 这个格式我没有遇见过，本文中不做分析。本文针对的 ffmpeg 的版本是 0.5 版本。

2 mpegts_demuxer 结构分析

AVInputFormat mpegts_demuxer = {au

"mpegts", //demux 的名称

NULL_IF_CONFIG_SMALL("MPEG-2 transport stream format"),// 如果定义了 CONFIG_SMALL 宏，该域返回 NULL，也就是取消 long_name 域的定义。

```
    sizeof(MpegTSContext),//每个 demuxer 的结构的私有域的大小
    mpegts_probe,//检测是否是 TS 流格式
    mpegts_read_header,//下文介绍
    mpegts_read_packet,//下文介绍
    mpegts_read_close,//关闭 demuxer
    read_seek,//下文介绍
    mpegts_get_pcr,//下文介绍
    .flags = AVFMT_SHOW_IDS|AVFMT_TS_DISCONT,//下文介绍
};
```

该结构通过 av_register_all 函数注册到 ffmpeg 的主框架中，通过 mpegts_probe 函数来检测是否是 TS 流格式，然后通过 mpegts_read_header 函数找到一路音频流和一路视频流（注意：在该函数中没有找全所有的音频流和视频流），最后调用 mpegts_read_packet 函数将找到的音频流和视频流数据提取出来，通过主框架推入解码器。

3 mpegts_probe 函数分析

mpegts_probe 被 av_probe_input_format2 调用，根据返回的 score 来判断那种格式的可能性最大。 mpegts_probe 调用了 analyze 函数，我们先分析一下 analyze 函数。

```
static int analyze(const uint8_t *buf, int size, int packet_size, int *index){
    int stat[TS_MAX_PACKET_SIZE];//积分统计结果
    int i;
    int x=0;
    int best_score=0;
    memset(stat, 0, packet_size*sizeof(int));
    for(x=i=0; i<size-3; i++){
        if(buf[i] == 0x47 && !(buf[i+1] & 0x80) && (buf[i+3] & 0x30)){
            stat[x]++;
            if(stat[x] > best_score){
                best_score= stat[x];
                if(index) *index= x;
            }
        }
        x++;
        if(x == packet_size) x= 0;
    }
    return best_score;
}
```

analyze 函数的思路：

buf[i] == 0x47 && !(buf[i+1] & 0x80) && (buf[i+3] & 0x30)是 TS 流同步开始的模式，0x47 是 TS 流同步的标志，(buf[i+1] & 0x80 是传输错误标志，buf[i+3] & 0x30 为 0 时表示为 ISO/IEC 未来使用保留，目前不存在这样的值。记该模式为“TS 流同步模式”

stat 数组变量存储的是“TS 流同步模式”在某个位置出现的次数。

analyze 函数扫描检测数据，如果发现该模式，则 stat[i%packet_size]++(函数中的 x 变量就是用来取模运算的，因为当 x==packet_size 时，x 就被归零)，扫描完后，自然是同步位置的累加值最大。

mpegts_probe 函数通过调用 analyze 函数来得到相应的分数，ffmpeg 框架会根据该分数判断是否是对应的格式，返回对应的 AVInputFormat 实例。

4 mpegts_read_header 函数分析

下文中省略号代表的是和 mpegtsraw_demuxer 相关的代码，暂不涉及。

```
static int mpegts_read_header(AVFormatContext *s,
                              AVFormatParameters *ap)
{
    MpegTSContext *ts = s->priv_data;
```

```
    ByteIOContext *pb = s->pb;
    uint8_t buf[5*1024];
    int len;
    int64_t pos;
    ......
    //保存流的当前位置，便于检测操作完成后恢复到原来的位置,
    //这样在播放的时候就不会浪费一段流。
    pos = url_ftell(pb);
    //读取一段流来检测 TS 包的大小
    len = get_buffer(pb, buf, sizeof(buf));
    if (len != sizeof(buf))
        goto fail;
    //得到 TS 流包的大小，通常是 188bytes,我目前见过的都是 188 个字节的。
    //TS 包的大小有三种：
    //1) 通常情况下的 188 字节
    //2)日本弄了个 192Bytes 的 DVH-S 格式
    //3)在 188Bytes 的基础上，加上 16Bytes 的 FEC(前向纠错),也就是 204bytes
    ts->raw_packet_size = get_packet_size(buf, sizeof(buf));
    if (ts->raw_packet_size <= 0)
        goto fail;
    ts->stream = s;

    //auto_guess = 1, 则在 handle_packet 的函数中只要发现一个 PES 的 pid 就
    //建立该 PES 的 stream
    //auto_guess = 0, 则忽略。
    //auto_guess 主要作用是用来在 TS 流中没有业务信息时，如果被设置成了 1 的话，
    //那么就会将任何一个 PID 的流当做媒体流建立对应的 PES 数据结构。
    //在 mpegts_read_header 函数的过程中发现了 PES 的 pid，但
    //是不建立对应的流,只是分析 PSI 信息。
    //相关的代码见 handle_packet 函数的下面的代码：
    // tss = ts->pids[pid];
    //if (ts->auto_guess && tss == NULL && is_start) {
    //    add_pes_stream(ts, pid, -1, 0);
    //    tss = ts->pids[pid];
    //}
    ts->auto_guess = 0;
    if (s->iformat == &mpegts_demuxer) {
        /* normal demux */
        /* first do a scaning to get all the services */
        url_fseek(pb, pos, SEEK_SET);
        //挂载解析 SDT 表的回调函数到 ts->pids 变量上，
        //这样在 handle_packet 函数中根据对应的 pid 找到对应处理回调函数。
        mpegts_open_section_filter(ts, SDT_PID, sdt_cb, ts, 1);
        //同上，只是挂上 PAT 表解析的回调函数
        mpegts_open_section_filter(ts, PAT_PID, pat_cb, ts, 1);
        //探测一段流，便于检测出 SDT，PAT，PMT 表
        handle_packets(ts, s->probesize);
        /* if could not find service, enable auto_guess */

        //打开 add pes stream 的标志，这样在 handle_packet 函数中发现了 pes 的
        //pid，就会自动建立该 pes 的 stream。
        ts->auto_guess = 1;
        dprintf(ts->stream, "tuning done\n");
        s->ctx_flags |= AVFMTCTX_NOHEADER;
    } else {
        ......
    }
    //恢复到检测前的位置。
    url_fseek(pb, pos, SEEK_SET);
    return 0;
 fail:
    return -1;
}
```

下面介绍被 mpegts_read_header 直接或者间接调用的几个函数：mpegts_open_section_filter，

handle_packets，handle_packet

5 mpegts_open_section_filter 函数分析

这个函数可以解释 mpegts.c 代码结构的精妙之处，PSI 业务信 息表的处理都是通过该函数挂载到 MpegTSContext 结构的 pids 字段上的。这样如果你想增加别的业务信息的表处理函数只要通过 这个函数来挂载即可，体现了软件设计的著名的“开闭”原则。下面分析一下他的代码。

```
static MpegTSFilter *mpegts_open_section_filter(MpegTSContext *ts, unsigned int pid,
                                                SectionCallback *section_cb, void *opaque,
                                                int check_crc)
{
    MpegTSFilter *filter;
    MpegTSSectionFilter *sec;
    dprintf(ts->stream, "Filter: pid=0x%x\n", pid);
    if (pid >= NB_PID_MAX || ts->pids[pid])
        return NULL;
    //给 filter 分配空间，挂载到 MpegTSContext 的 pids 上
    //就是该实例
    filter = av_mallocz(sizeof(MpegTSFilter));
    if (!filter)
        return NULL;
    //挂载 filter 实例
    ts->pids[pid] = filter;
    //设置 filter 相关的参数，因为业务信息表的分析的单位是段，
    //所以该 filter 的类型是 MPEGTS_SECTION
    filter->type = MPEGTS_SECTION;

    //设置 pid
    filter->pid = pid;
    filter->last_cc = -1;
    //设置 filter 回调处理函数
    sec = &filter->u.section_filter;
    sec->section_cb = section_cb;
    sec->opaque = opaque;
    //分配段数据处理的缓冲区，调用 handle_packet 函数后会调用
    //write_section_data 将 ts 包中的业务信息表的数据存储在这儿，
    //直到一个段收集完成才交付上面注册的回调函数处理。
    sec->section_buf = av_malloc(MAX_SECTION_SIZE);
    sec->check_crc = check_crc;
    if (!sec->section_buf) {
        av_free(filter);
        return NULL;
    }
    return filter;
}
```

6 handle_packets 函数分析

handle_packets 函数在两个地方被调用，一个是 mpegts_read_header 函数中，另外一个是 mpegts_read_packet 函数中，被 mpegts_read_header 函数调用是用来搜索 PSI 业务信息，nb_packets 参数为探测的 ts 包的个数；在 mpegts_read_packet 函数中被调用用来搜索补充 PSI 业务信息和 demux PES 流，nb_packets 为 0，0 不是表示处理的包的个数为 0。

```
static int handle_packets(MpegTSContext *ts, int nb_packets)
{
    AVFormatContext *s = ts->stream;
    ByteIOContext *pb = s->pb;
    uint8_t packet[TS_PACKET_SIZE];
    int packet_num, ret;
    //该变量指示一次 handle_packets 处理的结束。
    //在 mpegts_read_packet 被调用的时候，如果发现完一个 PES 的包，则
    // ts->stop_parse = 1，则当前分析结束。
    ts->stop_parse = 0;
    packet_num = 0;
    for(;;) {
```

```
        if (ts->stop_parse>0)
            break;
        packet_num++;
        if (nb_packets != 0 && packet_num >= nb_packets)
            break;
        //读取一个 ts 包，通常是 188bytes
        ret = read_packet(pb, packet, ts->raw_packet_size);
        if (ret != 0)
            return ret;
        handle_packet(ts, packet);
    }
    return 0;
}
```

7 handle_packet 函数分析

可以说 handle_packet 是 mpegts.c 代码的核心，所有的其他代码都是为这 个函数准备的。

在调用该函数之前先调用 read_packet 函数获得一个 ts 包（通常是 188bytes），然后传给该函 数，packet 参数就是 TS 包。

```
static int handle_packet(MpegTSContext *ts, const uint8_t *packet)
{
    AVFormatContext *s = ts->stream;
    MpegTSFilter *tss;
    int len, pid, cc, cc_ok, afc, is_start;
    const uint8_t *p, *p_end;
    int64_t pos;
    //从 TS 包获得包的 PID。
    pid = AV_RB16(packet + 1) & 0x1fff;
    if(pid && discard_pid(ts, pid))
        return 0;
    is_start = packet[1] & 0x40;
    tss = ts->pids[pid];
    //ts->auto_guess 在 mpegts_read_header 函数中被设置为 0，
    //也就是说在 ts 检测过程中是不建立 pes stream 的。
    if (ts->auto_guess && tss == NULL && is_start) {
        add_pes_stream(ts, pid, -1, 0);
        tss = ts->pids[pid];
    }
    //mpegts_read_header 函数调用 handle_packet 函数只是处理 TS 流的
    //业务信息，因为并没有为对应的 PES 建立 tss，所以 tss 为空，直接返回。
    if (!tss)
        return 0;
    /* continuity check (currently not used) */
    cc = (packet[3] & 0xf);
    cc_ok = (tss->last_cc < 0) || ((((tss->last_cc + 1) & 0x0f) == cc));
    tss->last_cc = cc;
    /* skip adaptation field */
    afc = (packet[3] >> 4) & 3;
    p = packet + 4;
    if (afc == 0) /* reserved value */
        return 0;
    if (afc == 2) /* adaptation field only */
        return 0;
    if (afc == 3) {
        /* skip adapation field */
        p += p[0] + 1;
    }
    /* if past the end of packet, ignore */
    p_end = packet + TS_PACKET_SIZE;
    if (p >= p_end)
        return 0;
    pos = url_ftell(ts->stream->pb);
    ts->pos47= pos % ts->raw_packet_size;
    if (tss->type == MPEGTS_SECTION) {
        //表示当前的 TS 包包含一个新的业务信息段
        if (is_start) {
```

```
            //获取 pointer field 字段,
            //新的段从 pointer field 字段指示的位置开始
            len = *p++;
            if (p + len > p_end)
                return 0;
            if (len && cc_ok) {
                //这个时候 TS 的负载有两个部分构成：
                //1)从 TS 负载开始到 pointer field 字段指示的位置;
                //2)从 pointer field 字段指示的位置到 TS 包结束

                //1)位置代表的是上一个段的末尾部分。
                //2)位置代表的新的段开始的部分。
                //下面的代码是保存上一个段末尾部分数据，也就是
                //1)位置的数据。
                write_section_data(s, tss,
                                   p, len, 0);
                /* check whether filter has been closed */
                if (!ts->pids[pid])
                    return 0;
            }
            p += len;
            //保留新的段数据，也就是 2)位置的数据。
            if (p < p_end) {
                write_section_data(s, tss,
                                   p, p_end - p, 1);
            }
        } else {
            //保存段中间的数据。
            if (cc_ok) {
                write_section_data(s, tss,
                                   p, p_end - p, 0);
            }
        }
    } else {
        int ret;
        //正常的 PES 数据的处理
        // Note: The position here points actually behind the current packet.
        if ((ret = tss->u.pes_filter.pes_cb(tss, p, p_end - p, is_start,
                                            pos - ts->raw_packet_size)) < 0)
            return ret;
    }
    return 0;
}
```

8 write_section_data 函数分析

PSI 业务信息表在 TS 流中是以段为单位传输的。

```
static void write_section_data(AVFormatContext *s, MpegTSFilter *tss1,
                               const uint8_t *buf, int buf_size, int is_start)
{
    MpegTSSectionFilter *tss = &tss1->u.section_filter;
    int len;
    //buf 中是一个段的开始部分。
    if (is_start) {
        //将内容复制到 tss->section_buf 中保存
        memcpy(tss->section_buf, buf, buf_size);
        //tss->section_index 段索引。
        tss->section_index = buf_size;
        //段的长度，现在还不知道，设置为-1
        tss->section_h_size = -1;
        //是否到达段的结尾。
        tss->end_of_section_reached = 0;
    } else {
        //buf 中是段中间的数据。
        if (tss->end_of_section_reached)
            return;
        len = 4096 - tss->section_index;
        if (buf_size < len)
            len = buf_size;
```

wish happy everyday!

```
        memcpy(tss->section_buf + tss->section_index, buf, len);
        tss->section_index += len;
    }
    //如果条件满足，计算段的长度
    if (tss->section_h_size == -1 && tss->section_index >= 3) {
        len = (AV_RB16(tss->section_buf + 1) & 0xfff) + 3;
        if (len > 4096)
            return;
        tss->section_h_size = len;
    }
    //判断段数据是否收集完毕，如果收集完毕，调用相应的回调函数处理该段。
    if (tss->section_h_size != -1 && tss->section_index >= tss->section_h_size) {
        tss->end_of_section_reached = 1;
        if (!tss->check_crc ||
            av_crc(av_crc_get_table(AV_CRC_32_IEEE), -1,
                   tss->section_buf, tss->section_h_size) == 0)
            tss->section_cb(tss1, tss->section_buf, tss->section_h_size);
    }
}
```