转 C语言的面向对象设计 —— 对 X264/FFMPEG 架构探讨

2013年10月14日 23:49:42　阅读数：6047

类似题目的文章已经不新鲜了，这里，我仅仅总结自己的一些代码经验，结合两款在视频开发领域比较常用的开源软件探讨C语言的应用问题。

1.为什么要用C语言

曾几何时，我也不熟悉C，最早接触C的是在大学四年级，当时已经学过pascal，过二级也是pascal。接着走上了Delphi的路，多方便的软件，写写画画，程序就出来了，本科的毕业设计就是这样出来的MIS，在当时还很时髦的花哨了一阵，弄了个优秀论文。当有一天，看到别人的代码，一行行的，整齐的缩进，而略为难理解的*号，一下被这种语言难住了。于是拿着国内最流行的谭浩强的C书一阵狂读，云里雾里的。等到了国外，依然要跟那些金发碧眼的朋友一起读C语言，但是是为了解决数值运算问题，一面拼命理解难懂的外文，一面偷偷在下面看谭C，所作的作业也是求4阶矩阵乘法等等，终于有些实战机会了，当然也很气愤那调试的黑呼呼GCC界面，想着弄个Delphi的美丽调试界面该多好啊。越往后走，接触的C代码越多，认识些C的程序员也越多，终于发现那些自诩厉害的“核心”程序员，原来是用C的。记得一次项目的机会，接触一个在硅谷摸爬了20多年的美国架构师，每次开项目会议都把电脑带上，一面讨论一面敲打些什么，会议结束，代码架构已成。不是流程图，不是伪代码，不是文字，是真真实实的C代码。

直到今天，C语言虽然不是使用人数最多的语言了，但是C没有老去，在很多的核心系统代码里，依然跑的是设计精美的C，绝大多数的嵌入式开发核心库软件是C开发的，多数标准算法是基于标准C设计的。C语言以其简洁，灵活和性能优越，依然在核心软件设计师心目中有不可动摇的地位。

2.为什么要面向对象

面向对象是一种设计方法，刚开始学习C++的时候，接触了面向对象的方法，终于对以前写Delphi系统是托动的图图框框的控件有了更深入的认识。及至以后要用到Java，更是觉得面向对象的方便，容易理解分析问题。以至于到了现在，每当要写软件原型，我的第一反应就是用写字板来写Java，然后用javac来命令行编译，古老的打印调试。这样的开发方式好处多多，首先JAVA的库比较稳定单一，不如C++的庞大繁杂，也没有内存管理的头痛事务，更重要的是能够充分发挥面向对象分析，使得自己在单位时间专注做好一个类，最后把这若干的类串起来，代码已成。重构，设计模式都不过是日后的优化提炼，没有必要硬性引入。

面向过程往往被认为是一种严格的自顶向下，逐步细分的设计方式，按部就班的大规模设计分解成小的具体实现。而面向对象是基于对象模型对问题域进行描述,更加接近于人们对客观世界的认识过程。在一般的软件工程教案中,罗列了如下面向对象的好处：(1)模块化 (2)抽象(3)信息隐藏（4）弱耦合（5）强内聚（6）可重用。而这些好处来自于运用面向对象的三个基本方法：封装、继承和多态。在实际的软件工程项目中，正是因为面向对象的这些特性，这种分析方法受到广泛欢迎并且继续保持发展，最近的J2EE项目层出不穷的框架思想正是最好的例子，无论是对象注入还是POJO都给面向对象方法增添更多新的活力。

从【1】中可以看到面向过程和面向对象对同一项目分析的简单举例，从而得到结论，面向对象是以功能来划分问题，而不是步骤。在【2】的系列文章中，作者分析了如何用C语言的结构体和函数指针模拟封装，继承和多态，并用简单的实验分析了性能损失，结果是C语言模拟类的损失可以忽略不计。基于标准C面向对象的代码示例可以从LINUX，GTK等源码中看到，本文仅仅分析FFMPEG和X264的基本架构。面向对象是一种高效的分析设计方法，而C语言没有直接支持面向对象的语法，用C来模仿C++是没有必要的，在考虑用C语言构建大型项目的时候，利用面向对象设计，并且适当的构造C语法支持这样的设计思想是需要的。

3.FFMPEG架构分析

FFMPEG是目前被应用最广泛的编解码软件库，支持多种流行的编解码器，它是C语言实现的，不仅被集成到各种PC软件，也经常被移植到多种嵌入式设备中。使用面向对象的办法来设想这样一个编解码库，首先让人想到的是构造各种编解码器的类，然后对于它们的抽象基类确定运行数据流的规则，根据算法转换输入输出对象。

在实际的代码，将这些编解码器分成encoder/decoder，muxer/demuxer和device三种对象，分别对应于编解码，输入输出格式和设备。在main函数的开始，就是初始化这三类对象。在avcodec_register_all中，很多编解码器被注册，包括视频的H.264解码器和X264编码器等，

```
REGISTER_DECODER (H264, h264);
REGISTER_ENCODER (LIBX264, libx264);
```

找到相关的宏代码如下

```
#define REGISTER_ENCODER(X,x) { \
extern AVCodecx##_encoder; \
if(CONFIG_##X##_ENCODER)  avcodec_register(&x##_encoder); }
#define REGISTER_DECODER(X,x) { \
extern AVCodec x##_decoder;\
if(CONFIG_##X##_DECODER)  avcodec_register(&x##_decoder); }
```

这样就实际在代码中根据CONFIG_##X##_ENCODER这样的编译选项来注册libx264_encoder和h264_decoder，注册的过程发生在avcodec_register(AVCodec *codec)函数中，实际上就是向全局链表first_avcodec中加入libx264_encoder、h264_decoder特定的编解码器，输入参数AVCodec是一个结构体，可以理解为编解码器的基类，其中不仅包含了名称，id等属性，而且包含了如下函数指针，让每个具体的编解码器扩展类实现。

```
int (*init)(AVCodecContext *);
int (*encode)(AVCodecContext *, uint8_t *buf, int buf_size,void *data);
int (*close)(AVCodecContext *);
```

```
int (*decode)(AVCodecContext *, void *outdata, int*outdata_size,
const uint8_t *buf, int buf_size);
void (*flush)(AVCodecContext *);
```

继续追踪libx264，也就是X264的静态编码库，它在FFMPEG编译的时候被引入作为H.264编码器。在libx264.c中有如下代码

```
AVCodec libx264_encoder = {
.name = "libx264",
.type = CODEC_TYPE_VIDEO,
.id = CODEC_ID_H264,
.priv_data_size = sizeof(X264Context),
.init = X264_init,
.encode = X264_frame,
.close = X264_close,
.capabilities = CODEC_CAP_DELAY,
.pix_fmts = (enum PixelFormat[]) { PIX_FMT_YUV420P,PIX_FMT_NONE },
.long_name = NULL_IF_CONFIG_SMALL("libx264 H.264 / AVC/ MPEG-4 AVC / MPEG-4 part 10"),
};
```

这里具体对来自AVCodec得属性和方法赋值。其中

```
.init = X264_init,
.encode = X264_frame,
.close = X264_close,
```

将函数指针指向了具体函数，这三个函数将使用libx264静态库中提供的API，也就是X264的主要接口函数进行具体实现。pix_fmts定义了所支持的输入格式，这里4：2：0

```
PIX_FMT_YUV420P,   ///< planar YUV 4:2:0, 12bpp, (1 Cr & Cbsample per 2x2 Y samples)
```

上面看到的X264Context封装了X264所需要的上下文管理数据，

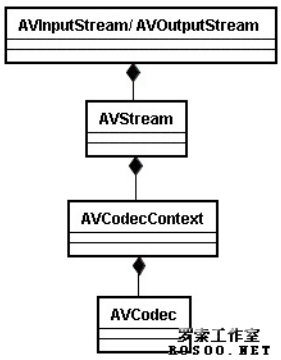
```
typedef struct X264Context {
x264_param_t params;
x264_t *enc;
x264_picture_t pic;
AVFrame out_pic;
} X264Context;
```

它属于结构体AVCodecContext的void*priv_data变量，定义了每种编解码器私有的上下文属性，AVCodecContext也类似上下文基类一样，还提供其他表示屏幕解析率、量化范围等的上下文属性和rtp_callback等函数指针供编解码使用。

回到main函数，可以看到完成了各类编解码器，输入输出格式和设备注册以后，将进行上下文初始化和编解码参数读入，然后调用av_encode（）函数进行具体的编解码工作。根据该函数的注释一路查看其过程：

1. 输入输出流初始化。
2. 根据输入输出流确定需要的编解码器，并初始化。
3. 写输出文件的各部分

重点关注一下step2和3，看看怎么利用前面分析的编解码器基类来实现多态。大概查看一下这段代码的关系，发现在FFMPEG里，可以用类图来表示大概的编解码器组合。



可以参考【3】来了解这些结构的含义（见附录）。在这里会调用一系列来自utils.c的函数，这里的avcodec_open（）函数，在打开编解码器都会调用到，它将运行如下代码：

```
avctx->codec = codec;
avctx->codec_id = codec->id;
avctx->frame_number = 0;
if(avctx->codec->init){
```

ret =avctx->codec->init(avctx);
进行具体适配的编解码器初始化，而这里的avctx->codec->init(avctx)就是调用AVCodec中函数指针定义的具体初始化函数，例如X264_init。

在avcodec_encode_video（）和avcodec_encode_audio（）被output_packet（）调用进行音视频编码，将同样利用函数指针avctx->codec->encode（）调用适配编码器的编码函数，如X264_frame进行具体工作。

从上面的分析，我们可以看到FFMPEG怎么利用面向对象来抽象编解码器行为，通过组合和继承关系具体化每个编解码器实体。设想要在FFMPEG中加入新的解码器H265，要做的事情如下：
1. 在config编译配置中加入CONFIG_H265_DECODER
2. 利用宏注册H265解码器
3. 定义AVCodec 265_decoder变量，初始化属性和函数指针
4. 利用解码器API具体化265_decoder的init等函数指针

完成以上步骤，就可以把新的解码器放入FFMPEG，外部的匹配和运行规则由基类的多态实现了。

4. X264架构分析

X264是一款从2004年有法国大学生发起的开源H.264编码器，对PC进行汇编级代码优化，舍弃了片组和多参考帧等性能效率比不高的功能来提高编码效率，它被FFMPEG作为引入的.264编码库，也被移植到很多DSP嵌入平台。前面第三节已经对FFMPEG中的X264进行举例分析，这里将继续结合X264框架加深相关内容的了解。

查看代码前，还是思考一下对于一款具体的编码器，怎么面向对象分析呢？对熵编码部分对不同算法的抽象，还有帧内或帧间编码各种估计算法的抽象，都可以作为类来构建。

在X264中，我们看到的对外API和上下文变量都声明在X264.h中，API函数中，关于辅助功能的函数在common.c中定义
```
void x264_picture_alloc( x264_picture_t *pic, int i_csp, int i_width, inti_height );
void x264_picture_clean( x264_picture_t *pic );
int x264_nal_encode( void *, int *, int b_annexeb, x264_nal_t *nal );
```
而编码功能函数定义在encoder.c
```
x264_t *x264_encoder_open   ( x264_param_t * );
int     x264_encoder_reconfig( x264_t *, x264_param_t * );
int     x264_encoder_headers( x264_t *, x264_nal_t **, int* );
int     x264_encoder_encode ( x264_t *, x264_nal_t **, int*, x264_picture_t *, x264_picture_t * );
void    x264_encoder_close  ( x264_t * );
```
在x264.c文件中，有程序的main函数，可以看作做API使用的例子，它也是通过调用X264.h中的API和上下文变量来实现实际功能。

X264最重要的记录上下文数据的结构体x264_t定义在common.h中，它包含了从线程控制变量到具体的SPS、PPS、量化矩阵、cabac上下文等所有的H.264编码相关变量。其中包含如下的结构体
```
x264_predict_t      predict_16x16[4+3];
x264_predict_t      predict_8x8c[4+3];
x264_predict8x8_t   predict_8x8[9+3];
x264_predict_t      predict_4x4[9+3];
x264_predict_8x8_filter_t predict_8x8_filter;

x264_pixel_function_t pixf;
x264_mc_functions_t   mc;
x264_dct_function_t   dctf;
x264_zigzag_function_t zigzagf;
x264_quant_function_t quantf;
x264_deblock_function_t loopf;
```
跟踪查看可以看到它们或是一个函数指针，或是由函数指针组成的结构，这样的用法很想面向对象中的interface接口声明。这些函数指针将在x264_encoder_open（）函数中被初始化，这里的初始化首先根据CPU的不同提供不同的函数实现代码段，很多与可能是汇编实现，以提高代码运行效率。其次把功能相似的函数集中管理，例如类似intra16的4种和intra4的九种预测函数都被用函数指针数组管理起来。

x264_encoder_encode（）是负责编码的主要函数，而其内包含的x264_slice_write（）负责片层一下的具体编码，包括了帧内和帧间宏块编码。在这里，cabac和cavlc的行为是根据h->param.b_cabac来区别的，分别运行x264_macroblock_write_cabac（）和x264_macroblock_write_cavlc（）来写码流，在这一部分，功能函数按文件定义归类，基本按照编码流程图运行，看起来更像面向过程的写法，在已经初始化了具体的函数指针，程序就一直按编码过程的逻辑实现。如果从整体架构来看，x264利用这种类似接口的形式实现了弱耦合和可重用，利用x264_t这个贯穿始终的上下文，实现信息封装和多态。

本文大概分析了FFMPEG/X264的代码架构，重点探讨用C语言来实现面向对象编码，虽不至于强行向C++靠拢，但是也各有实现特色，保证实用性。值得规划C语言软件项目所借鉴。

【参考文献】

1.“ 用例子说明面向对象和面向过程的区别 ”
2. liyuming1978，“ liyuming1978 的专栏 ”
3. “FFMpeg框架代码阅读”

附录：节选自【3】

3. 当前muxer/demuxer的匹配
在FFmpeg的文件转换过程中，首先要做的就是根据传入文件和传出文件的后缀名[FIXME]匹配合适的demuxer和muxer。匹配上的demuxer和muxer都保存在如下所示，定义在ffmpeg.c里的全局变量file_iformat和file_oformat中：
static AVInputFormat *file_iformat;
static AVOutputFormat *file_oformat;

3.1 demuxer匹配
在libavformat\utils.c中的staticAVInputFormat *av_probe_input_format2(
AVProbeData *pd, int is_opened, int *score_max)函数用途是根据传入的probe data数据，依次调用每个demuxer的read_probe接口，来进行该demuxer是否和传入的文件内容匹配的判断。其调用顺序如下：
void parse_options(int argc, char **argv, const OptionDef *options,
void (* parse_arg_function)(const char *));
static void opt_input_file(const char *filename)
int av_open_input_file(……)
AVInputFormat*av_probe_input_format(AVProbeData *pd,
int is_opened)
static AVInputFormat*av_probe_input_format2(……)
opt_input_file函数是在保存在const OptionDef options[]数组中，用于
void parse_options(int argc, char **argv, const OptionDef *options)中解析argv里的
“-i” 参数，也就是输入文件名时调用的。

3.2 muxer匹配
与demuxer的匹配不同，muxer的匹配是调用guess_format函数，根据main() 函数的argv里的输出文件后缀名来进行的。
void parse_options(int argc, char **argv, const OptionDef *options,
void (* parse_arg_function)(const char *));
void parse_arg_file(const char *filename)
static void opt_output_file(const char *filename)
AVOutputFormat *guess_format(constchar *short_name,
const char *filename,
const char *mime_type)

3.3 当前encoder/decoder的匹配
在main()函数中除了解析传入参数并初始化demuxer与muxer的parse_options( )函数以外，其他的功能都是在av_encode( )函数里完成的。
在libavcodec\utils.c中有如下二个函数:
AVCodec *avcodec_find_encoder(enum CodecID id)
AVCodec *avcodec_find_decoder(enum CodecID id)
他们的功能就是根据传入的CodecID，找到匹配的encoder和decoder。

在av_encode( )函数的开头，首先初始化各个AVInputStream和AVOutputStream，然后分别调用上述二个函数，并将匹配上的encoder与decoder分别保存在:
AVInputStream->AVStream *st->AVCodecContext *codec->struct AVCodec*codec与
AVOutputStream->AVStream *st->AVCodecContext *codec->struct AVCodec*codec变量。

4. 其他主要数据结构
4.1 AVFormatContext
AVFormatContext是FFMpeg格式转换过程中实现输入和输出功能、保存相关数据的主要结构。每一个输入和输出文件，都在如下定义的指针数组全局变量中有对应的实体。
static AVFormatContext *output_files[MAX_FILES];
static AVFormatContext *input_files[MAX_FILES];
对于输入和输出，因为共用的是同一个结构体，所以需要分别对该结构中如下定义的iformat或oformat成员赋值。
struct AVInputFormat *iformat;

struct AVOutputFormat *oformat;
对一个AVFormatContext来说，这二个成员不能同时有值，即一个AVFormatContext不能同时含有demuxer和muxer。在main( )函数开头的parse_options( )函数中找到了匹配的muxer和demuxer之后，根据传入的argv参数，初始化每个输入和输出的AVFormatContext结构，并保存在相应的output_files和input_files指针数组中。在av_encode( )函数中，output_files和input_files是作为函数参数传入后，在其他地方就没有用到了。

4.2 AVCodecContext
保存AVCodec指针和与codec相关数据，如video的width、height，audio的sample rate等。AVCodecContext中的codec_type，codec_id二个变量对于encoder/decoder的匹配来说，最为重要。
enum CodecType codec_type; /* see CODEC_TYPE_xxx */
enum CodecID codec_id; /* seeCODEC_ID_xxx */

如上所示，codec_type保存的是CODEC_TYPE_VIDEO，CODEC_TYPE_AUDIO等媒体类型，codec_id保存的是CODEC_ID_FLV1，CODEC_ID_VP6F等编码方式。

以支持flv格式为例，在前述的av_open_input_file(……) 函数中，匹配到正确的AVInputFormat demuxer后，通过av_open_input_stream( )函数中调用AVInputFormat的read_header接口来执行flvdec.c中的flv_read_header()函数。在flv_read_header( )函数内，根据文件头中的数据，创建相应的视频或音频AVStream，并设置AVStream中AVCodecContext的正确的codec_type值。codec_id值是在解码过程中flv_read_packet( )函数执行时根据每一个packet头中的数据来设置的。

4.3 AVStream
AVStream结构保存与数据流相关的编解码器，数据段等信息。比较重要的有如下二个成员：
AVCodecContext *codec; /**< codec context */
void *priv_data;
其中codec指针保存的就是上节所述的encoder或decoder结构。priv_data指针保存的是和具体编解码流相关的数据，如下代码所示，在ASF的解码过程中，priv_data保存的就是ASFStream结构的数据。
AVStream *st;
ASFStream *asf_st;
… …
st->priv_data = asf_st;

4.4 AVInputStream/ AVOutputStream
根据输入和输出流的不同，前述的AVStream结构都是封装在AVInputStream和AVOutputStream结构中，在av_encode( )函数中使用。AVInputStream中还保存的有与时间有关的信息。AVOutputStream中还保存有与音视频同步等相关的信息。

4.5 AVPacket
AVPacket结构定义如下，其是用于保存读取的packet数据。
typedef struct AVPacket {
int64_t pts; ///<presentation time stamp in time_base units
int64_t dts; ///<decompression time stamp in time_base units
uint8_t *data;
int size;
int stream_index;
int flags;
int duration; ///<presentation duration in time_base units (0 if not available)
void (*destruct)(struct AVPacket *);
void *priv;
int64_t pos; ///< byteposition in stream, -1 if unknown
} AVPacket;

在av_encode()函数中，调用AVInputFormat的(*read_packet)(struct AVFormatContext *, AVPacket *pkt)接口，读取输入文件的一帧数据保存在当前输入AVFormatContext的AVPacket成员中。

原文：http://jmvc.blog.sohu.com/120705757.html

(jmvc)

文章标签： ffmpeg x264 架构 设计 c语言

个人分类： FFMPEG 视频编码

所属专栏： FFmpeg