

原 FFMpeg源代码简单分析：日志输出系统（av_log()等）

2015年03月14日 12:19:53 阅读数：26083

=====

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

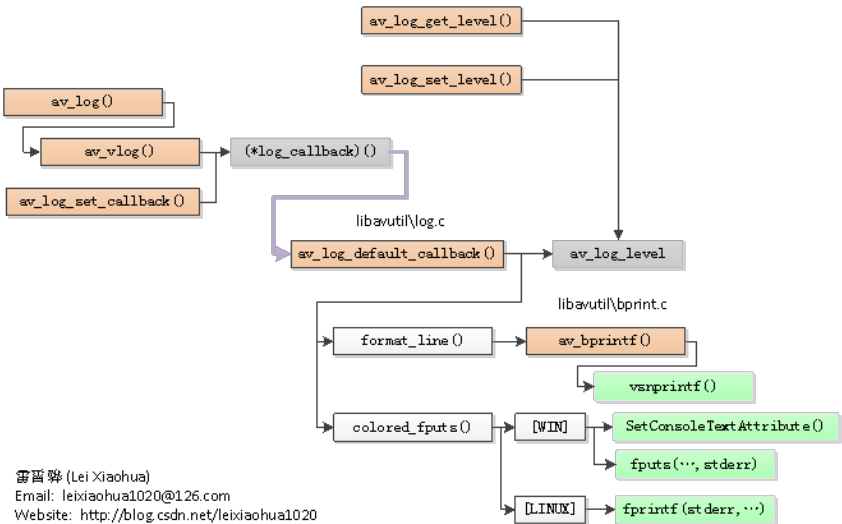
本文分析一下FFmpeg的日志（Log）输出系统的源代码。日志输出部分的核心函数只有一个：av_log()。使用av_log()在控制台输出日志的效果如下图所示。

```
[NULL @ 0047c4c0] Panic: Something went really wrong and we will crash now.
[NULL @ 0047c4c0] Fatal: Something went wrong and recovery is not possible..
[NULL @ 0047c4c0] Error: Something went wrong and cannot lesslessly be recovered

[NULL @ 0047c4c0] Warning: This may or may not lead to problems.
[NULL @ 0047c4c0] Info: Standard information.
[NULL @ 0047c4c0] Verbose: Detailed information.
[NULL @ 0047c4c0] Debug: Stuff which is only useful for libav* developers.
```

函数调用结构图

FFmpeg日志输出系统的函数调用结构图如图所示。



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

av_log()

av_log()是FFmpeg中输出日志的函数。随便打开一个FFmpeg的源代码文件，就会发现其中遍布着av_log()函数。一般情况下FFmpeg类库的源代码中是不允许使用printf()这种的函数的，所有的输出一律使用av_log()。
av_log()的声明位于libavutil\log.h，如下所示。

```
1. /**
2.  * Send the specified message to the log if the level is less than or equal
3.  * to the current av_log_level. By default, all logging messages are sent to
4.  * stderr. This behavior can be altered by setting a different logging callback
5.  * function.
6.  * @see av_log_set_callback
7.  *
8.  * @param avcl A pointer to an arbitrary struct of which the first field is a
9.  * pointer to an AVClass struct.
10. * @param level The importance level of the message expressed using a @ref
11. * lavu_log_constants "Logging Constant".
12. * @param fmt The format string (printf-compatible) that specifies how
13. * subsequent arguments are converted to output.
14. */
15. void av_log(void *avcl, int level, const char *fmt, ...) av_printf_format(3, 4);
```

这个函数的声明有两个地方比较特殊：

- (1) 函数最后一个参数是“...”。
- 在C语言中，在函数参数数量不确定的情况下使用“...”来代表参数。例如printf()的原型定义如下

```
1. int printf (const char*, ...);
```

后文中对此再作详细分析。

- (2) 它的声明后面有一个`av_printf_format(3, 4)`。有关这个地方的左右还没有深入研究，网上资料中说它的作用是按照`printf()`的格式检查`av_log()`的格式。
`av_log()`每个字段的含义如下：
`avcl`：指定一个包含`AVClass`的结构体。
`level`：log的级别
`fmt`：和`printf()`一样。

由此可见，`av_log()`和`printf()`的不同主要在于前面多了两个参数。其中第一个参数指定该log所属的结构体，例如`AVFormatContext`、`AVCodecContext`等等。第二个参数指定log的级别，源代码中定义了如下几个级别。

```
[cpp]
1.  /**
2.   * Print no output.
3.   */
4.  #define AV_LOG_QUIET    -8
5.
6.  /**
7.   * Something went really wrong and we will crash now.
8.   */
9.  #define AV_LOG_PANIC     0
10.
11. /**
12.  * Something went wrong and recovery is not possible.
13.  * For example, no header was found for a format which depends
14.  * on headers or an illegal combination of parameters is used.
15.  */
16. #define AV_LOG_FATAL     8
17.
18. /**
19.  * Something went wrong and cannot losslessly be recovered.
20.  * However, not all future data is affected.
21.  */
22. #define AV_LOG_ERROR    16
23.
24. /**
25.  * Something somehow does not look correct. This may or may not
26.  * lead to problems. An example would be the use of '-vstrict -2'.
27.  */
28. #define AV_LOG_WARNING  24
29.
30. /**
31.  * Standard information.
32.  */
33. #define AV_LOG_INFO     32
34.
35. /**
36.  * Detailed information.
37.  */
38. #define AV_LOG_VERBOSE  40
39.
40. /**
41.  * Stuff which is only useful for libav* developers.
42.  */
43. #define AV_LOG_DEBUG    48
```

从定义中可以看出，随着严重程度逐渐下降，一共包含如下级别：`AV_LOG_PANIC`，`AV_LOG_FATAL`，`AV_LOG_ERROR`，`AV_LOG_WARNING`，`AV_LOG_INFO`，`AV_LOG_VERBOSE`，`AV_LOG_DEBUG`。每个级别定义的数值代表了严重程度，数值越小代表越严重。默认的级别是`AV_LOG_INFO`。此外，还有一个级别不输出任何信息，即`AV_LOG_QUIET`。

当前系统存在着一个“Log级别”。所有严重程度高于该级别的Log信息都会输出出来。例如当前的Log级别是`AV_LOG_WARNING`，则会输出`AV_LOG_PANIC`，`AV_LOG_FATAL`，`AV_LOG_ERROR`，`AV_LOG_WARNING`级别的信息，而不会输出`AV_LOG_INFO`级别的信息。可以通过`av_log_get_level()`获得当前Log的级别，通过另一个函数`av_log_set_level()`设置当前的Log级别。

av_log_get_level(), av_log_set_level()

`av_log_get_level()`的定义如下所示

```
[cpp]
1.  /**
2.   * Get the current log level
3.   *
4.   * @see lavu_log_constants
5.   *
6.   * @return Current log level
7.   */
8.  int av_log_get_level(void);
```

可以通过`av_log_set_level()`设置当前Log的级别。

```
[cpp]
1. /**
2.  * Set the log level
3.  *
4.  * @see lavu_log_constants
5.  *
6.  * @param level Logging level
7.  */
8. void av_log_set_level(int level);
```

上述两个函数的定义十分的简单，如下所示。

```
[cpp]
1. int av_log_get_level(void)
2. {
3.     return av_log_level;
4. }
```

```
[cpp]
1. void av_log_set_level(int level)
2. {
3.     av_log_level = level;
4. }
```

从代码中可以看出，以上两个函数主要操作了一个静态全局变量av_log_level。该变量用于存储当前系统Log的级别。它的定义如下所示。

```
[cpp]
1. static int av_log_level = AV_LOG_INFO;
```

下面我们看一下H.264编码的时候libx264的Log输出的示例：

```
[[libx264 @ 0041c4c0] using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE4.2 AVX
[[libx264 @ 0041c4c0] profile Main, level 2.1
[[libx264 @ 0041c4c0] frame= 0 QP=32.07 NAL=3 Slice:I Poc:0 I:510 P:0 SKI
P:0 size=9321 bytes
[[libx264 @ 0041c4c0] frame= 1 QP=39.48 NAL=2 Slice:P Poc:10 I:0 P:224 SKI
P:206 size=305 bytes
[[libx264 @ 0041c4c0] frame= 2 QP=40.47 NAL=2 Slice:B Poc:6 I:0 P:85 SKI
P:425 size=131 bytes
[[libx264 @ 0041c4c0] frame= 3 QP=41.47 NAL=0 Slice:B Poc:2 I:0 P:87 SKI
P:423 size=127 bytes
[[libx264 @ 0041c4c0] frame= 4 QP=41.71 NAL=0 Slice:B Poc:4 I:0 P:82 SKI
P:428 size=141 bytes
[[libx264 @ 0041c4c0] frame= 5 QP=41.71 NAL=0 Slice:B Poc:8 I:0 P:58 SKI
P:452 size=109 bytes
[[libx264 @ 0041c4c0] frame= 6 QP=36.10 NAL=2 Slice:P Poc:22 I:0 P:288 SKI
P:222 size=675 bytes
[[libx264 @ 0041c4c0] frame= 7 QP=39.05 NAL=2 Slice:B Poc:16 I:0 P:164 SKI
P:346 size=261 bytes
[[libx264 @ 0041c4c0] frame= 8 QP=41.00 NAL=0 Slice:B Poc:12 I:0 P:85 SKI
P:425 size=155 bytes
[[libx264 @ 0041c4c0] frame= 9 QP=40.71 NAL=0 Slice:B Poc:14 I:0 P:113 SKI
```

下面回到av_log()函数的源代码。它的源代码位于libavutil/log.c，如下所示。

```
[cpp]
1. void av_log(void* avcl, int level, const char *fmt, ...)
2. {
3.     AVClass* avc = avcl ? *(AVClass **) avcl : NULL;
4.     va_list vl;
5.     va_start(vl, fmt);
6.     if (avc && avc->version >= (50 << 16 | 15 << 8 | 2) &&
7.         avc->log_level_offset_offset && level >= AV_LOG_FATAL)
8.         level += *(int *) (((uint8_t *) avcl) + avc->log_level_offset_offset);
9.     av_vlog(avcl, level, fmt, vl);
10.    va_end(vl);
11. }
```

首先来提一下C语言函数中“...”参数的含义。与它相关还涉及到以下4个部分：

- (1) va_list变量
- (2) va_start()
- (3) va_arg()
- (4) va_end()

va_list是一个指向函数的参数的指针。va_start()用于初始化va_list变量。va_arg()用于返回可变参数。va_end()用于结束可变参数的获取。有关它们的用法可以参考一个小demo，如下所示。

```
[cpp]
1. #include <stdio.h>
2. #include<stdarg.h>
3. void fun(int a,...){
4.     va_list pp;
5.     va_start(pp,a);
6.     do{
7.         printf("param =%d\n",a);
8.         a=va_arg(pp,int);//使 pp 指向下一个参数,将下一个参数的值赋给变量 a
9.     }
10.    while (a!=0);//直到参数为 0 时停止循环
11. }
12. void main(){
13.     fun(20,40,60,80,0);
14. }
```

有关这方面的知识很难用简短的语言描述清楚，因此不再详述。av_log()的源代码中，在va_start()和va_end()之间，调用了另一个函数av_vlog()。

av_vlog()

av_vlog()是一个FFmpeg的API函数。它的声明位于libavutilLog.h中，如下所示。

```
[cpp]
1. /**
2.  * Send the specified message to the log if the level is less than or equal
3.  * to the current av_log_level. By default, all logging messages are sent to
4.  * stderr. This behavior can be altered by setting a different logging callback
5.  * function.
6.  * @see av_log_set_callback
7.  *
8.  * @param avcl A pointer to an arbitrary struct of which the first field is a
9.  * pointer to an AVClass struct.
10. * @param level The importance level of the message expressed using a @ref
11. * lavu_log_constants "Logging Constant".
12. * @param fmt The format string (printf-compatible) that specifies how
13. * subsequent arguments are converted to output.
14. * @param vl The arguments referenced by the format string.
15. */
16. void av_vlog(void *avcl, int level, const char *fmt, va_list vl);
```

从声明中可以看出，av_vlog()和av_log()的参数基本上是一模一样的。唯一的不同在于av_log()中的“...”变成了av_vlog()中的va_list。

av_vlog()的定义位于libavutilLog.c中，如下所示。

```
[cpp]
1. void av_vlog(void* avcl, int level, const char *fmt, va_list vl)
2. {
3.     void (*log_callback)(void*, int, const char*, va_list) = av_log_callback;
4.     if (log_callback)
5.         log_callback(avcl, level, fmt, vl);
6. }
```

从定义中可以看出，av_vlog()简单调用了一个函数指针av_log_callback。av_log_callback是一个全局静态变量，定义如下所示。

```
[cpp]
1. static void (*av_log_callback)(void*, int, const char*, va_list) =
2.     av_log_default_callback;
```

从代码中可以看出，av_log_callback指针默认指向一个函数av_log_default_callback()。av_log_default_callback()即FFmpeg默认的Log函数。需要注意的是，这个Log函数是可以自定义的。按照指定的参数定义一个自定义的函数后，可以通过FFmpeg的另一个API函数av_log_set_callback()设定为Log函数。

av_log_set_callback()的声明如下所示。

```
[cpp]
1. /**
2.  * Set the logging callback
3.  *
4.  * @note The callback must be thread safe, even if the application does not use
5.  * threads itself as some codecs are multithreaded.
6.  *
7.  * @see av_log_default_callback
8.  *
9.  * @param callback A logging function with a compatible signature.
10. */
11. void av_log_set_callback(void (*callback)(void*, int, const char*, va_list));
```

从声明中可以看出，需要指定一个参数为(void*, int, const char*, va_list)，返回值为void的函数作为Log函数。

av_log_set_callback()的定义很简单，做了一个函数指针赋值的工作，如下所示。

```
[cpp]
1. void av_log_set_callback(void (*callback)(void*, int, const char*, va_list))
2. {
3.     av_log_callback = callback;
4. }
```

例如，我们可以指定一个my_logoutput()函数作为Log的输出函数，就可以将Log信息输出到文本中（而不是屏幕上）。

```
[cpp]
1. void my_logoutput(void* ptr, int level, const char* fmt, va_list vl){
2.     FILE *fp = fopen("my_log.txt", "a+");
3.     if(fp){
4.         vfprintf(fp, fmt, vl);
5.         fflush(fp);
6.         fclose(fp);
7.     }
8. }
```

编辑好函数之后，使用av_log_set_callback()函数设置该函数为Log输出函数即可。

```
[cpp]
1. av_log_set_callback(my_logoutput);
```

av_log_default_callback()

下面我们看一下FFmpeg的默认Log输出函数av_log_default_callback()。它的定义如下。

```
[cpp]
1. void av_log_default_callback(void* ptr, int level, const char* fmt, va_list vl)
2. {
3.     static int print_prefix = 1;
4.     static int count;
5.     static char prev[LINE_SZ];
6.     AVBPrint part[4];
7.     char line[LINE_SZ];
8.     static int isatty;
9.     int type[2];
10.    unsigned tint = 0;
11.
12.    if (level >= 0) {
13.        tint = level & 0xff00;
14.        level &= 0xff;
15.    }
16.
17.    if (level > av_log_level)
18.        return;
19.    #if HAVE_PTHREADS
20.    pthread_mutex_lock(&mutex);
21.    #endif
22.
23.    format_line(ptr, level, fmt, vl, part, &print_prefix, type);
24.    snprintf(line, sizeof(line), "%s%s%s", part[0].str, part[1].str, part[2].str, part[3].str);
25.
26.    #if HAVE_ISATTY
27.    if (!isatty)
28.        isatty = isatty(2) ? 1 : -1;
29.    #endif
30.
31.    if (print_prefix && (flags & AV_LOG_SKIP_REPEATED) && !strcmp(line, prev) &&
32.        *line && line[strlen(line) - 1] != '\r'){
33.        count++;
34.        if (isatty == 1)
35.            fprintf(stderr, "    Last message repeated %d times\r", count);
36.        goto end;
37.    }
38.    if (count > 0) {
39.        fprintf(stderr, "    Last message repeated %d times\n", count);
40.        count = 0;
41.    }
42.    strcpy(prev, line);
43.    sanitize(part[0].str);
44.    colored_fputs(type[0], 0, part[0].str);
45.    sanitize(part[1].str);
46.    colored_fputs(type[1], 0, part[1].str);
47.    sanitize(part[2].str);
48.    colored_fputs(av_clip(level >> 3, 0, 6), tint >> 8, part[2].str);
49.    sanitize(part[3].str);
50.    colored_fputs(av_clip(level >> 3, 0, 6), tint >> 8, part[3].str);
51. end:
52.    av_bprint_finalize(part+3, NULL);
53.    #if HAVE_PTHREADS
54.    pthread_mutex_unlock(&mutex);
55.    #endif
56. }
```

av_log_default_callback()的代码是比较复杂的。其实如果我们仅仅是希望把Log信息输出到屏幕上，远不需要那么多代码，只需要简单打印一下就可以了。av_log_default_callback()之所以会那么复杂，主要是因为他还包含了很多的功能，比如说根据Log级别的不同将输出的文本设置成不同的颜色等等。下图显示了不同级别的Log不同的背景颜色。



下面看一下av_log_default_callback()的源代码大致的流程：

- (1) 如果输入参数level大于系统当前的日志级别av_log_level，表明不需要做任何处理，直接返回。
- (2) 调用format_line()设定Log的输出格式。
- (3) 调用colored_fputs()设定Log的颜色。

format_line(), av_log_format_line()

format_line()用于设定Log的输出格式。它本身并不是一个FFmpeg的API，但是FFmpeg有一个API函数av_log_format_line()调用了这个函数。av_log_format_line()的声明如下所示。

```
[cpp]
1.  /**
2.   * Format a line of log the same way as the default callback.
3.   * @param line      buffer to receive the formatted line
4.   * @param line_size size of the buffer
5.   * @param print_prefix used to store whether the prefix must be printed;
6.   *                   must point to a persistent integer initially set to 1
7.   */
8.  void av_log_format_line(void *ptr, int level, const char *fmt, va_list vl,
9.                          char *line, int line_size, int *print_prefix);
```

av_log_format_line()的定义如下所示。

```
[cpp]
1.  void av_log_format_line(void *ptr, int level, const char *fmt, va_list vl,
2.                          char *line, int line_size, int *print_prefix)
3.  {
4.      AVBPrint part[4];
5.      format_line(ptr, level, fmt, vl, part, print_prefix, NULL);
6.      snprintf(line, line_size, "%s%s%s%s", part[0].str, part[1].str, part[2].str, part[3].str);
7.      av_bprint_finalize(part+3, NULL);
8.  }
```

从代码中可以看出，首先声明了一个AVBPrint类型的数组，其中包含了4个元素；接着调用format_line()设定格式；最后将设置格式后的AVBPrint数组中的4个元素连接起来。

在这里遇到了一个结构体AVBPrint，它的定义位于libavutil/bprint.h，如下所示。

```

1.  /**
2.   * Buffer to print data progressively
3.   *
4.   * The string buffer grows as necessary and is always 0-terminated.
5.   * The content of the string is never accessed, and thus is
6.   * encoding-agnostic and can even hold binary data.
7.   *
8.   * Small buffers are kept in the structure itself, and thus require no
9.   * memory allocation at all (unless the contents of the buffer is needed
10.  * after the structure goes out of scope). This is almost as lightweight as
11.  * declaring a local "char buf[512]".
12.  *
13.  * The length of the string can go beyond the allocated size: the buffer is
14.  * then truncated, but the functions still keep account of the actual total
15.  * length.
16.  *
17.  * In other words, buf->len can be greater than buf->size and records the
18.  * total length of what would have been to the buffer if there had been
19.  * enough memory.
20.  *
21.  * Append operations do not need to be tested for failure: if a memory
22.  * allocation fails, data stop being appended to the buffer, but the length
23.  * is still updated. This situation can be tested with
24.  * av_bprint_is_complete().
25.  *
26.  * The size_max field determines several possible behaviours:
27.  *
28.  * size_max = -1 (= UINT_MAX) or any large value will let the buffer be
29.  * reallocated as necessary, with an amortized linear cost.
30.  *
31.  * size_max = 0 prevents writing anything to the buffer: only the total
32.  * length is computed. The write operations can then possibly be repeated in
33.  * a buffer with exactly the necessary size
34.  * (using size_init = size_max = len + 1).
35.  *
36.  * size_max = 1 is automatically replaced by the exact size available in the
37.  * structure itself, thus ensuring no dynamic memory allocation. The
38.  * internal buffer is large enough to hold a reasonable paragraph of text,
39.  * such as the current paragraph.
40.  */
41.  typedef struct AVBPrint {
42.      FF_PAD_STRUCTURE(1024,
43.      char *str;          /**< string so far */
44.      unsigned len;       /**< length so far */
45.      unsigned size;      /**< allocated memory */
46.      unsigned size_max;  /**< maximum allocated memory */
47.      char reserved_internal_buffer[1];
48.  )
49.  } AVBPrint;

```

AVBPrint的注释代码很长，不再详细叙述。在这里只要知道他是用于打印字符串的缓存就可以了。它的名称BPrint的意思应该就是“Buffer to Print”。其中的str存储了将要打印的字符串。

format_line()函数的定义如下所示。


```

1. static void format_line(void *avcl, int level, const char *fmt, va_list vl,
2.                          AVBPrint part[4], int *print_prefix, int type[2])
3. {
4.     AVClass* avc = avcl ? (AVClass **) avcl : NULL;
5.     av_bprint_init(part+0, 0, 1);
6.     av_bprint_init(part+1, 0, 1);
7.     av_bprint_init(part+2, 0, 1);
8.     av_bprint_init(part+3, 0, 65536);
9.
10.    if(type) type[0] = type[1] = AV_CLASS_CATEGORY_NA + 16;
11.    if (*print_prefix && avc) {
12.        if (avc->parent_log_context_offset) {
13.            AVClass** parent = *(AVClass ***) (((uint8_t *) avcl) +
14.                                                avc->parent_log_context_offset);
15.            if (parent && *parent) {
16.                av_bprintf(part+0, "[%s @ %p] ",
17.                           (*parent)->item_name(parent), parent);
18.                if(type) type[0] = get_category(parent);
19.            }
20.        }
21.        av_bprintf(part+1, "[%s @ %p] ",
22.                   avc->item_name(avcl), avcl);
23.        if(type) type[1] = get_category(avcl);
24.
25.        if (flags & AV_LOG_PRINT_LEVEL)
26.            av_bprintf(part+2, "[%s] ", get_level_str(level));
27.    }
28.
29.    av_vbprintf(part+3, fmt, vl);
30.
31.    if(*part[0].str || *part[1].str || *part[2].str || *part[3].str) {
32.        char lastc = part[3].len && part[3].len <= part[3].size ? part[3].str[part[3].len - 1] : 0;
33.        *print_prefix = lastc == '\n' || lastc == '\r';
34.    }
35. }

```

从代码中可以看出，其分别处理了AVBPrint类型数组part的4个元素。由此我们也可以看出FFmpeg一条Log可以分成4个组成部分。在这里涉及到几个与AVBPrint相关的函数，由于篇幅的关系，不再分析它们的源代码，仅仅列出它们的定义：

初始化AVBPrint的函数av_bprint_init()。

```

1. /**
2.  * Init a print buffer.
3.  *
4.  * @param buf      buffer to init
5.  * @param size_init initial size (including the final 0)
6.  * @param size_max  maximum size;
7.  *                  0 means do not write anything, just count the length;
8.  *                  1 is replaced by the maximum value for automatic storage;
9.  *                  any large value means that the internal buffer will be
10.                 reallocated as needed up to that limit; -1 is converted to
11.                 UINT_MAX, the largest limit possible.
12.  *                  Check also AV_BPRINT_SIZE_* macros.
13.  */
14. void av_bprint_init(AVBPrint *buf, unsigned size_init, unsigned size_max);

```

向AVBPrint添加一个字符串的函数av_bprintf()。

```

1. /**
2.  * Append a formatted string to a print buffer.
3.  */
4. void av_bprintf(AVBPrint *buf, const char *fmt, ...) av_printf_format(2, 3);

```

向AVBPrint添加一个字符串的函数av_vbprintf()，注意与av_bprintf()的不同在于其第3个参数不一样。

```

1. /**
2.  * Append a formatted string to a print buffer.
3.  */
4. void av_vbprintf(AVBPrint *buf, const char *fmt, va_list vl_arg);

```

我们可以瞄一眼av_bprintf()的定义，位于libavutil/bprint.c，如下所示。

```
[cpp]
1. void av_bprintf(AVBPrint *buf, const char *fmt, ...)
2. {
3.     unsigned room;
4.     char *dst;
5.     va_list vl;
6.     int extra_len;
7.
8.     while (1) {
9.         room = av_bprint_room(buf);
10.        dst = room ? buf->str + buf->len : NULL;
11.        va_start(vl, fmt);
12.        extra_len = vsnprintf(dst, room, fmt, vl);
13.        va_end(vl);
14.        if (extra_len <= 0)
15.            return;
16.        if (extra_len < room)
17.            break;
18.        if (av_bprint_alloc(buf, extra_len))
19.            break;
20.    }
21.    av_bprint_grow(buf, extra_len);
22. }
```

可以看出av_bprintf()实际上调用了系统的vsnprintf()完成了相应的功能。

看完以上几个与AVBPrint相关函数之后，就可以来看一下format_line()的代码了。例如，part[0]对应的是目标结构体的父结构体的名称（如果父结构体存在的话）；其打印格式形如“[%s @ %p]”，其中前面的“%s”对应父结构体的名称，“%p”对应其所在的地址。part[1]对应的是目标结构体的名称；其打印格式形如“[%s @ %p]”，其中前面的“%s”对应本结构体的名称，“%p”对应其所在的地址。part[2]用于输出Log的级别，这个字符串只有在flag中设置AV_LOG_PRINT_LEVEL的时候才能打印。part[3]则是打印原本传送进来的文本。将format_line()函数处理后得到的4个字符串连接起来，就可以得到一条完整的Log信息。下面图显示了flag设置AV_LOG_PRINT_LEVEL后的打印出来的Log的格式。

默认情况下不设置flag打印出来的格式如下所示。

colored_fputs()

colored_fputs()函数用于将输出的文本“上色”并且输出。在这里有一点需要注意：Windows和Linux下控制台程序上色的方法是不一样的。Windows下是通过SetConsoleTextAttribute()方法给控制台中的文本上色；Linux下则是通过添加一些ANSI控制码完成上色。

Linux下控制台文字上色的方法

Linux下控制台颜色是通过添加专用数字来选择的。这些数字夹在“\e[”和“m”之间。如果指定一个以上的数字，则用分号将它们分开。

举几个例子：

- (1) 第一个数字(31)为前景颜色(红色)；第二个数字为(42)背景颜色(绿色)

```
[plain]
1. echo -e "\e[31;42m"
```

- (2) 使用“\e[0m”序列将颜色重新设置为正常值

```
[plain]
1. echo -e "\e[0m" 或 echo -e "\033[0m"
```

- (3) 颜色对应关系如下所示：

```
\e[30m -- \e[37m 设置前景色(字体颜色)
echo -e "\e[30m" 灰色
echo -e "\e[31m" 红色
echo -e "\e[32m" 绿色
echo -e "\e[33m" 黄色
echo -e "\e[34m" 蓝色
echo -e "\e[35m" 紫色
echo -e "\e[36m" 淡蓝色
echo -e "\e[37m" 白色
```

```
\e[40m -- \e[47m 设置背景色
```

```
echo -e "\e[40m" 灰色
echo -e "\e[41m" 红色
echo -e "\e[42m" 绿色
echo -e "\e[43m" 黄色
echo -e "\e[44m" 蓝色
echo -e "\e[45m" 紫色
echo -e "\e[46m" 淡蓝色
echo -e "\e[47m" 白色
```

具体到编程中，printf() 颜色设置示例代码如下所示。

```
[cpp]
1. int main()
2. {
3.     printf("\e[31m Hello World. \e[0m \n"); // 红色字体
4.     return 0;
5. }
```

Windows下控制台文字上色的方法

Windows下控制台颜色是通过SetConsoleTextAttribute()函数完成的。SetConsoleTextAttribute()函数的原型如下所示。

```
[cpp]
1. BOOL SetConsoleTextAttribute(HANDLE hConsoleOutput, WORD wAttributes);
```

其中2个参数的含义如下所示：

hConsoleOutput：指向控制台的句柄。

wAttributes：文本属性。

hConsoleOutput可以选择以下3种句柄：

STD_INPUT_HANDLE：

标准输入的句柄

STD_OUTPUT_HANDLE：

标准输出的句柄

STD_ERROR_HANDLE：

标准错误的句柄

wAttributes可以控制前景色和背景色：

FOREGROUND_BLUE：

字体颜色：蓝

FOREGROUND_GREEN：

字体颜色：绿

FOREGROUND_RED：

字体颜色：红

FOREGROUND_INTENSITY：

前景色高亮显示

BACKGROUND_BLUE：

背景颜色：蓝

BACKGROUND_GREEN：

背景颜色：绿

BACKGROUND_RED：

背景颜色：红

BACKGROUND_INTENSITY

背景色高亮显示

控制台文本上色demo代码如下所示。

```

1.  /**
2.   * 雷霄骅 Lei Xiaohua
3.   * leixiaohua1020@126.com
4.   * http://blog.csdn.net/leixiaohua1020
5.   */
6.  #include <stdio.h>
7.  #include <windows.h>
8.
9.
10. int main()
11. {
12.     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED);
13.     printf("red\n");
14.     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_GREEN);
15.     printf("green\n");
16.     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_BLUE);
17.     printf("blue\n");
18.     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED|FOREGROUND_GREEN);
19.     printf("red+green=yellow\n");
20.     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED|FOREGROUND_BLUE);
21.     printf("red+blue=purple\n");
22.     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_GREEN|FOREGROUND_BLUE);
23.     printf("green+blue=cyan\n");
24.     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED|BACKGROUND_GREEN);
25.     printf("Add background\n");
26.
27.
28.     return 0;
29. }

```

程序的运行结果如下图所示。



colored_fputs()源代码

下面看一下colored_fputs()函数的源代码。

```

1. static void colored_fputs(int level, int tint, const char *str)
2. {
3.     int local_use_color;
4.     if (!*str)
5.         return;
6.
7.     if (use_color < 0)
8.         check_color_terminal();
9.
10.    if (level == AV_LOG_INFO/8) local_use_color = 0;
11.    else local_use_color = use_color;
12.
13.    #if defined(_WIN32) && !defined(__MINGW32CE__) && HAVE_SETCONSOLETEXTATTRIBUTE
14.        if (local_use_color)
15.            SetConsoleTextAttribute(con, background | color[level]);
16.        fputs(str, stderr);
17.        if (local_use_color)
18.            SetConsoleTextAttribute(con, attr_orig);
19.    #else
20.        if (local_use_color == 1) {
21.            fprintf(stderr,
22.                "\033[%d;3%dm%s\033[0m",
23.                (color[level] >> 4) & 15,
24.                color[level] & 15,
25.                str);
26.        } else if (tint && use_color == 256) {
27.            fprintf(stderr,
28.                "\033[48;5;%dm\033[38;5;%dm%s\033[0m",
29.                (color[level] >> 16) & 0xff,
30.                tint,
31.                str);
32.        } else if (local_use_color == 256) {
33.            fprintf(stderr,
34.                "\033[48;5;%dm\033[38;5;%dm%s\033[0m",
35.                (color[level] >> 16) & 0xff,
36.                (color[level] >> 8) & 0xff,
37.                str);
38.        } else
39.            fputs(str, stderr);
40.    #endif
41.
42. }

```

从colored_fputs()的源代码中可以看出如下流程：

首先判定根据宏定义系统的类型，如果系统类型是Windows，那么就调用SetConsoleTextAttribute()方法设定控制台文本的颜色，然后调用fputs()将Log记录输出到stderr（注意不是stdout）；如果系统类型是Linux，则通过添加特定字符串的方式设定控制台文本的颜色，然后将Log记录输出到stderr。

至此FFmpeg的日志输出系统的源代码就分析完毕了。

雷霄骅

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/44243155>

文章标签： [FFmpeg](#) [源代码](#) [日志](#) [log](#) [av_log](#)

个人分类： [FFMPEG](#)

所属专栏： [FFmpeg](#)

此PDF由spyyg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com