

原 FFMpeg源代码简单分析：内存的分配和释放（av_malloc()、av_free()等）

2015年03月03日 15:24:48 阅读数：27941

=====

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](#)

=====

本文简单记录一下FFmpeg中内存操作的函数。

内存操作的常见函数位于libavutil\mem.c中。本文记录FFmpeg开发中最常使用的几个函数：av_malloc(), av_realloc(), av_mallocz(), av_calloc(), av_free(), av_freep()。

av_malloc()

av_malloc()是FFmpeg中最常见的内存分配函数。它的定义如下。

```

1. #define FF_MEMORY_POISON 0x2a
2.
3. #define ALIGN (HAVE_AVX ? 32 : 16)
4.
5. static size_t max_alloc_size= INT_MAX;
6.
7. void *av_malloc(size_t size)
8. {
9.     void *ptr = NULL;
10.    #if CONFIG_MEMALIGN_HACK
11.        long diff;
12.    #endif
13.
14.
15.    /* let's disallow possibly ambiguous cases */
16.    if (size > (max_alloc_size - 32))
17.        return NULL;
18.
19.
20.    #if CONFIG_MEMALIGN_HACK
21.        ptr = malloc(size + ALIGN);
22.        if (!ptr)
23.            return ptr;
24.        diff = ((~(long)ptr)&(ALIGN - 1)) + 1;
25.        ptr = (char *)ptr + diff;
26.        ((char *)ptr)[-1] = diff;
27.    #elif HAVE_POSIX_MEMALIGN
28.        if (size) //OS X on SDK 10.6 has a broken posix_memalign implementation
29.            if (posix_memalign(&ptr, ALIGN, size))
30.                ptr = NULL;
31.    #elif HAVE_ALIGNED_MALLOC
32.        ptr = _aligned_malloc(size, ALIGN);
33.    #elif HAVE_MEMALIGN
34.    #ifndef __DJGPP__
35.        ptr = memalign(ALIGN, size);
36.    #else
37.        ptr = memalign(size, ALIGN);
38.    #endif
39.    /* Why 64?
40.     * Indeed, we should align it:
41.     *   on 4 for 386
42.     *   on 16 for 486
43.     *   on 32 for 586, PPro - K6-III
44.     *   on 64 for K7 (maybe for P3 too).
45.     * Because L1 and L2 caches are aligned on those values.
46.     * But I don't want to code such logic here!
47.     */
48.    /* Why 32?
49.     * For AVX ASM. SSE / NEON needs only 16.
50.     * Why not larger? Because I did not see a difference in benchmarks ...
51.     */
52.    /* benchmarks with P3
53.     * memalign(64) + 1      3071, 3051, 3032
54.     * memalign(64) + 2      3051, 3032, 3041
55.     * memalign(64) + 4      2911, 2896, 2915
56.     * memalign(64) + 8      2545, 2554, 2550
57.     * memalign(64) + 16     2543, 2572, 2563
58.     * memalign(64) + 32     2546, 2545, 2571
59.     * memalign(64) + 64     2570, 2533, 2558
60.     *
61.     * BTW, malloc seems to do 8-byte alignment by default here.
62.     */
63.    #else
64.        ptr = malloc(size);
65.    #endif
66.    if(!ptr && !size) {
67.        size = 1;
68.        ptr= av_malloc(1);
69.    }
70.    #if CONFIG_MEMORY_POISONING
71.    if (ptr)
72.        memset(ptr, FF_MEMORY_POISON, size);
73.    #endif
74.    return ptr;
75. }

```

如果不考虑上述代码中的一大堆宏定义（即类似CONFIG_MEMALIGN_HACK这类的宏都采用默认值0），av_malloc()的代码可以简化成如下形式。

```
[cpp]
1. void *av_malloc(size_t size)
2. {
3.     void *ptr = NULL;
4.     /* let's disallow possibly ambiguous cases */
5.     if (size > (max_alloc_size - 32))
6.         return NULL;
7.     ptr = malloc(size);
8.     if(!ptr && !size) {
9.         size = 1;
10.        ptr= av_malloc(1);
11.    }
12.    return ptr;
13. }
```

可以看出，此时的av_malloc()就是简单的封装了系统函数malloc()，并做了一些错误检查工作。

关于size_t

size_t 这个类型在FFmpeg中多次出现，简单解释一下其作用。size_t是为了增强程序的可移植性而定义的。不同系统上，定义size_t可能不一样。它实际上就是unsigned int。

为什么要内存对齐？

FFmpeg内存分配方面多次涉及到“内存对齐”（memory alignment）的概念。

这方面内容在IBM的网站上有一篇文章，讲的挺通俗易懂的，在此简单转述一下。

程序员通常认为内存就是一个字节数组，每次可以一个字节存取内存。例如在C语言中使用char *指代“一块内存”，Java中使用byte[]指代一块内存。如下所示。



但那实际上计算机处理器却不是这样认为的。处理器相对比较“懒惰”，它会以2字节，4字节，8字节，16字节甚至32字节来存取内存。例如下图显示了以4字节为单位读写内存的处理器“看待”上述内存的方式。



上述的存取单位的大小称之为内存存取粒度。

下面看一个实例，分别从地址0，和地址1读取4个字节到寄存器。

从程序员的角度来看，读取方式如下图所示。



而2字节存取粒度的处理器的读取方式如下图所示。



可以看出2字节存取粒度的处理器从地址0读取4个字节一共读取2次；从地址1读取4个字节一共读取了3次。由于每次读取的开销是固定的，因此从地址1读取4字节的效率有所下降。

4字节存取粒度的处理器的读取方式如下图所示。



可以看出4字节存取粒度的处理器从地址0读取4个字节一共读取1次；从地址1读取4个字节一共读取了2次。从地址1读取的开销比从地址0读取多了一倍。由此可见内存不对齐对CPU的性能是有影响的。

av_realloc()

av_realloc()用于对申请的内存的大小进行调整。它的定义如下。

```
[cpp]
1. void *av_realloc(void *ptr, size_t size)
2. {
3.     #if CONFIG_MEMALIGN_HACK
4.         int diff;
5.     #endif
6.
7.
8.     /* let's disallow possibly ambiguous cases */
9.     if (size > (max_alloc_size - 32))
10.        return NULL;
11.
12.
13.     #if CONFIG_MEMALIGN_HACK
14.         //FIXME this isn't aligned correctly, though it probably isn't needed
15.         if (!ptr)
16.             return av_malloc(size);
17.         diff = ((char *)ptr)[-1];
18.         av_assert0(diff>0 && diff<=ALIGN);
19.         ptr = realloc((char *)ptr - diff, size + diff);
20.         if (ptr)
21.             ptr = (char *)ptr + diff;
22.         return ptr;
23.     #elif HAVE_ALIGNED_MALLOC
24.         return _aligned_realloc(ptr, size + !size, ALIGN);
25.     #else
26.         return realloc(ptr, size + !size);
27.     #endif
28. }
```

默认情况下（CONFIG_MEMALIGN_HACK这些宏使用默认值0）的代码：

```
[cpp]
1. void *av_realloc(void *ptr, size_t size)
2. {
3.     /* let's disallow possibly ambiguous cases */
4.     if (size > (max_alloc_size - 32))
5.        return NULL;
6.     return realloc(ptr, size + !size);
7. }
```

可以看出av_realloc()简单封装了系统的realloc()函数。

av_mallocz()

av_mallocz()可以理解为av_malloc()+zeromemory。代码如下。

```
[cpp]
1. void *av_mallocz(size_t size)
2. {
3.     void *ptr = av_malloc(size);
4.     if (ptr)
5.         memset(ptr, 0, size);
6.     return ptr;
7. }
```

从源代码可以看出av_mallocz()中调用了av_malloc()之后，又调用memset()将分配的内存设置为0。

av_calloc()



av_calloc()则是简单封装了av_mallocz()，定义如下所示。

```
[cpp]
1. void *av_calloc(size_t nmemb, size_t size)
2. {
3.     if (size <= 0 || nmemb >= INT_MAX / size)
4.         return NULL;
5.     return av_mallocz(nmemb * size);
6. }
```



从代码中可以看出，它调用av_mallocz()分配了nmemb*size个字节的内存。

av_free()

av_free()用于释放申请的内存。它的定义如下。

```
[cpp]    
1. void av_free(void *ptr)  
2. {  
3.     #if CONFIG_MEMALIGN_HACK  
4.         if (ptr) {  
5.             int v= ((char *)ptr)[-1];  
6.             av_assert0(v>0 && v<=ALIGN);  
7.             free((char *)ptr - v);  
8.         }  
9.     #elif HAVE_ALIGNED_MALLOC  
10.        _aligned_free(ptr);  
11.    #else  
12.        free(ptr);  
13.    #endif  
14. }
```



默认情况下（CONFIG_MEMALIGN_HACK这些宏使用默认值0）的代码：

```
[cpp]    
1. void av_free(void *ptr)  
2. {  
3.     free(ptr);  
4. }
```

可以看出av_free()简单的封装了free()。

av_freep()

av_freep()简单封装了av_free()。并且在释放内存之后将目标指针设置为NULL。

```
[cpp]    
1. void av_freep(void *arg)  
2. {  
3.     void **ptr = (void **)arg;  
4.     av_free(*ptr);  
5.     *ptr = NULL;  
6. }
```

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/41176777>

文章标签：[ffmpeg](#) [malloc](#) [free](#) [calloc](#) [内存分配](#)

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