

原 深入浅出 - Android系统移植与平台开发 (八) - HAL Stub框架分析

标签: [android](#) [平台](#) [module](#) [struct](#) [methods](#)

2012-10-15 20:18 14471人阅读 评论(18) 收藏 举报

分类: [Android移植 \(59\)](#)

版权声明：本文为博主原创文章，未经博主允许不得转载。

1. HAL Stub框架分析

HAL stub的框架比较简单，三个结构体、两个常量、一个函数，简称321架构，它的定义在：

@hardware/libhardware/include/hardware/hardware.h

@hardware/libhardware/hardware.c

```
[cpp] ❏ ❏ ❏ ❏  
01.  /*  
02.  每一个硬件都通过hw_module_t来描述，我们称之为一个硬件对象。你可以去“继承”这个hw_module_t，然后扩展自己的属性，硬件对象必须定义为一个固定的名字：HMI，即：Hardware Module Information的简写，每一个硬件对象里都封装了一个函数指针open用于打开该硬件，我们理解为硬件对象的open方法，open调用后返回这个硬件对应的Operation interface。  
03.  */  
04.  struct hw_module_t {  
05.      uint32_t tag;           // 该值必须声明为HARDWARE_MODULE_TAG  
06.      uint16_t version_major; // 主版本号  
07.      uint16_t version_minor; // 次版本号  
08.      const char *id;        // 硬件id名，唯一标识module  
09.      const char *name;      // 硬件module名字  
10.      const char *author;    // 作者  
11.      struct hw_module_methods_t* methods; // 指向封装有open函数指针的结构体  
12.      void* dso;              // module's dso  
13.      uint32_t reserved[32-7]; // 128字节补齐  
14.  };  
15.  
16.  /*  
17.  硬件对象的open方法描述结构体，它里面只有一个元素：open函数指针
```

```

18.  */
19.  struct hw_module_methods_t{
20.      // 只封装了open函数指针
21.      int (*open)(const struct hw_module_t* module, const char * id,
22.                  struct hw_device_t** device);
23.  };
24.
25.  /*
26.  硬件对象hw_module_t的open方法返回该硬件的Operation interface, 它由hw_device_t结构体来描述, 我们称之为: 该硬件的操作接口
27.  */
28.  struct hw_device_t{
29.      uint32_t tag;           // 必须赋值为HARDWARE_DEVICE_TAG
30.      uint32_t version;      // 版本号
31.      struct hw_module_t* module; // 该设备操作属于哪个硬件对象, 可以看成硬件操作接口与硬件对象的联系
32.      uint32_t reserved[12];  // 字节补齐
33.      int (*close)(struct hw_device_t* device); // 该设备的关闭函数指针, 可以看做硬件的close方法
34.  };

```

上述三个结构之间关系紧密, 每个硬件对象由一个hw_module_t来描述, 只要我们拿到了这个硬件对象, 就可以调用它的open方法, 返回这个硬件对象的硬件操作接口, 然后就可以通过这些硬件操作接口来间接操作硬件了。只不过, open方法被struct hw_module_methods_t结构封装了一次, 硬件操作接口被hw_device_t封装了一次而已。

那用户程序如何才能拿到硬件对象呢?

答案是通过硬件id名来拿。

我们来看下321架构里的: 两个符号常量和一个函数:

```

[cpp]
01. // 这个就是HAL Stub对象固定的名字
02. #define HAL_MODULE_INFO_SYM          HMI
03. // 这是字符串形式的名字
04. #define HAL_MODULE_INFO_SYM_AS_STR   "HMI"
05. //这个函数是通过硬件名来获得硬件HAL Stub对象
06. int hw_get_module(const char *id, const struct hw_module_t **module);

```

当用户调用hw_get_module函数时, 第一个参数传硬件id名, 那么这个函数会从当前系统注册的硬件对象里查找传递过来的id名对应的硬件对象, 然后返回之。

从调用者的角度, 我们基本上没有什么障碍了, 那如何注册一个硬件对象呢?

很简单, 只需要声明一个结构体即可, 看下面这个Led Stub注册的例子:

```

[cpp]
01. const struct led_module_t HAL_MODULE_INFO_SYM = {
02.     common: {    // 初始化父结构hw_module_t成员
03.         tag: HARDWARE_MODULE_TAG,
04.         version_major: 1,
05.         version_minor: 0,
06.         id: LED_HARDWARE_MODULE_ID,
07.         name: "led HAL Stub",
08.         author: "farsight",
09.         methods: &led_module_methods,
10.     },
11.     // 扩展属性放在这儿
12. };

```

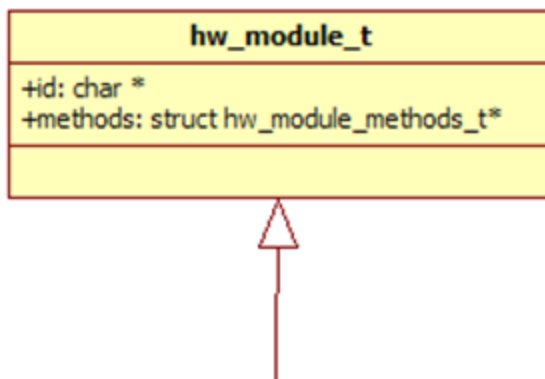
对，就这么简单，我们只需要声明一个结构体led_module_t，起名叫HAL_MODULE_INFO_SYM，也就是固定的名字：HMI，然后将这个结构体填充好就行了。led_module_t又是什么结构体类型啊？前面分析hw_module_t类型时说过，我们可以“继承”hw_module_t类型，创建自己的硬件对象，然后自己再扩展特有属性，这里的led_module_t就是“继承”的hw_module_t类型。注意，继承加上了双引号，因为在C语言里没有继承这个概念：

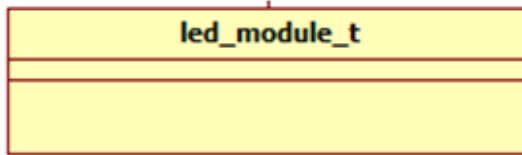
```

[cpp]
01. struct led_module_t {
02.     struct hw_module_t common;
03. };

```

结构体led_module_t封装了hw_module_t结构体，也就是说led_module_t这个新（子）结构体包含了旧（父）结构体，在新结构体里可以再扩展一些新的成员。结构体本身就具有封装特性，这不就是面向对象的封装和继承吗！为了显得专业点，我们用UML描述一下：





在上面的类图里，把hw_module_methods_t里封装的open函数指针指针写成open方法。

该open方法既：methods，自然也被子结构体给“继承”下来，我们将它初始化为led_module_methods的地址，该结构是hw_module_methods_t类型的，其声明代码如下：

```
[cpp] 01. static struct hw_module_methods_t led_module_methods = {
02.     open: led_device_open
03. };
```

简洁，我喜欢！！，它里面仅有的open成员是个函数指针，它被指向led_device_open函数：

```
[cpp] 01. static int led_device_open(const struct hw_module_t* module, const char* name,
02.     struct hw_device_t** device)
03. {
04.     struct led_device_t *led_device;
05.     LOGI("%s E ", __func__);
06.     led_device = (struct led_device_t *)malloc(sizeof(*led_device));
07.     memset(led_device, 0, sizeof(*led_device));
08.
09.     // init hw_device_t
10.     led_device->common.tag= HARDWARE_DEVICE_TAG;
11.     led_device->common.version = 0;
12.     led_device->common.module= module;
13.     led_device->common.close = led_device_close;
14.
15.     // init operation interface
16.     led_device->set_on= led_set_on;
17.     led_device->set_off= led_set_off;
18.     led_device->get_led_count = led_getcount;
19.     *device= (struct hw_device_t *)led_device;
20.
21.     if((fd=open("/dev/leds",O_RDWR))!=-1)
22.     {
23.         LOGI("open error");
```

```

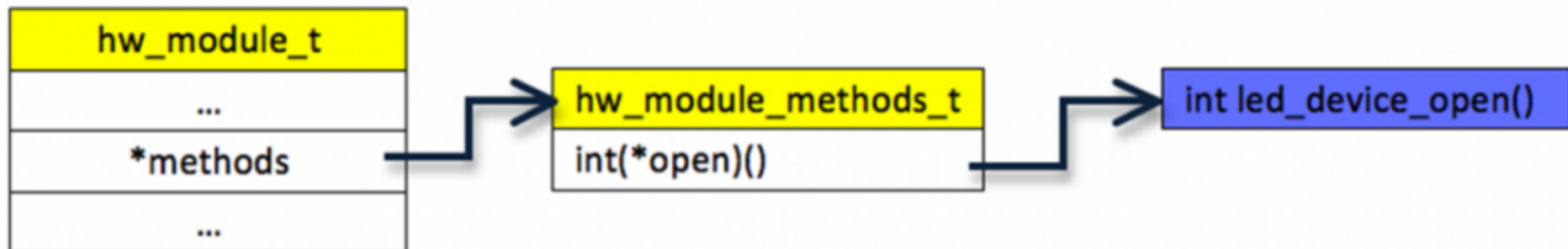
24.     return -1;
25. }else
26.     LOGI("open ok\n");
27.
28.     return 0;
29. }

```

led_device_open函数的功能：

- Ø 分配硬件设备操作结构体led_device_t，该结构体描述硬件操作行为
- Ø 初始化led_device_t的父结构体hw_device_t成员
- Ø 初始化led_device_t中扩展的操作接口
- Ø 打开设备，将led_device_t结构体以父结构体类型返回（面向对象里的多态）

hw_module_t与hw_module_methods_t及硬件open函数的关系如下：



我们来看下led_device_t和其父结构体hw_device_t的关系：

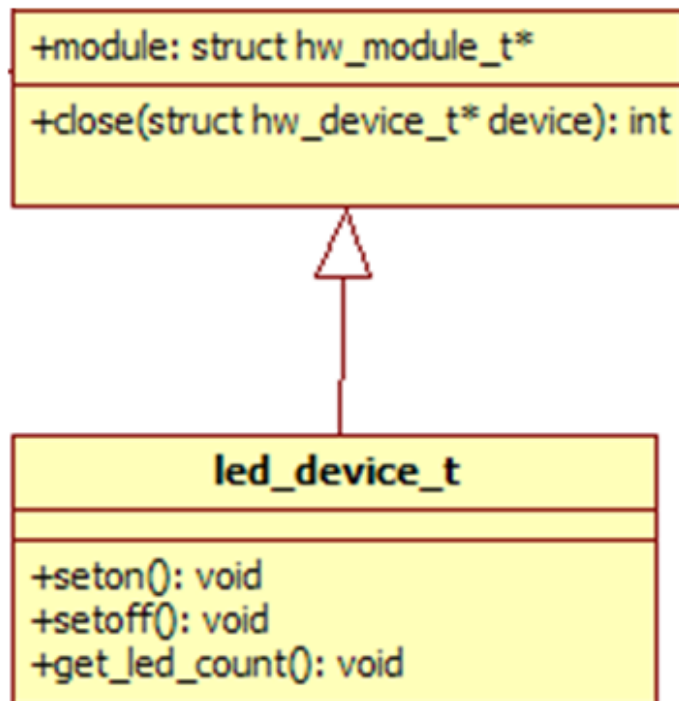
```

[cpp]
01. struct led_device_t {
02.     struct hw_device_t common; // led_devict_t的父结构，它里面只封装了close方法
03.     // 下面三个函数指针是子结构led_device_t对父结构hw_device_t的扩展，可以理解为子类扩展了父类增加了三个方法
04.     int (*getcount_led)(struct led_device_t *dev);
05.     int (*set_on)(struct led_device_t *dev);
06.     int (*set_off)(struct led_device_t *dev);
07. };

```





用UML类图来表示：





由类图可知，`led_device_t`扩展了三个接口：`seton()`，`setoff()`，`get_led_count()`。

那么剩下的工作就是实现子结构中新扩展的三个接口了：

```
[cpp]    
01. static int led_getcount(struct led_control_device_t*dev)
02. {
03.     LOGI("led_getcount");
04.     return 4;
05. }
06.
07. static int led_set_on(struct led_control_device_t *dev)
08. {
09.     LOGI("led_set_on");
10.     ioctl(fd,GPG3DAT2_ON,NULL);
11.     return 0;
12. }
13.
14. static int led_set_off(struct led_control_device_t*dev)
15. {
16.     LOGI("led_set_off");
17.     ioctl(fd,GPG3DAT2_OFF,NULL);
```

```
18.     return 0;
19. }
```

这三个接口函数直接和底层驱动打交道去控制硬件，具体驱动部分我们不去讲，那是另外一个体系了。

总结一下：

我们有一个硬件id名，通过这个id调用hw_get_module(char*id, struct hw_module_t **module),这个函数查找注册在当前系统中与id对应的硬件对象并返回之，硬件对象里有个通过hw_module_methods_t结构封装的open函数指针，回调这个open函数，它返回封装有硬件操作接口的led_device_t结构体，这样我们可以通过这个硬件接口去间接的访问硬件了。

在这个过程中hw_get_module返回的是子结构体类型led_module_t，虽然函数的第二个参数类型为hw_module_t的父类型，这里用到了面向对象里的多态的概念。

下面还有一个问题我们没有解决，为什么我们声明了一个名字为HMI结构体后，它就注册到了系统里？hw_get_module函数怎么找到并返回led_module_t描述的硬件对象的？

杀鸡取卵找HAL Stub

如果要知道为什么通过声明结构体就将HALStub注册到系统中，最好的方法是先知道怎么样通过hw_get_module_t来找到注册的硬件对象。

我们分析下hw_get_module函数的实现：

@hardware/libhardware/hardware.c

```
[cpp]    
01. static const char *variant_keys[] = {
02.     "ro.hardware",
03.     "ro.product.board",
04.     "ro.board.platform",
05.     "ro.arch"
06. };
07. // 由上面定义的字符串数组可知，HAL_VARIANT_KEYS_COUNT的值为4
08. struct constint HAL_VARIANT_KEYS_COUNT = (sizeof(variant_keys)/sizeof(variant_keys[0]));
09.
10. int hw_get_module(const char *id, const struct hw_module_t **module){
11.     // 调用3个参数的hw_get_module_by_class函数
12.     return hw_get_module_by_class(id, NULL, module);
13. }
14.
15. int hw_get_module_by_class(const char *class_id, const char *inst,
16. const struct hw_module_t **module){
17.     int status;
```

```

18.     int i;
19.     // 声明一个hw_module_t指针变量hmi
20.     const struct hw_module_t *hmi = NULL;
21.     char prop[PATH_MAX];
22.     char path[PATH_MAX];
23.     char name[PATH_MAX];
24.     // 由前面调用函数可知，inst = NULL，执行else部分，将硬件id名拷贝到name数组里
25.     if(inst)
26.         snprintf(name, PATH_MAX, "%s.%s", class_id, inst);
27.     else
28.         strcpy(name, class_id, PATH_MAX);
29.     // i 循环5次
30.     for(i=0; i<HAL_VARIANT_KEYS_COUNT+1; i++){
31.         if(i<HAL_VARIANT_KEYS_COUNT){
32.             // 从系统属性里依次查找前面定义的4个属性的值，找其中一个后，执行后面代码，找不到，进入else部分执行
33.             if(property_get(variant_keys[i], prop, NULL) == 0){
34.                 continue;
35.             }
36.             // 找到一个属性值prop后，拼写path的值为：/vendor/lib/hw/硬件id名.prop.so
37.             snprintf(path, sizeof(path), "%s/%s.%s.so",
38.                 HAL_LIBRARY_PATH2, name, prop);
39.             if(access(path, R_OK) == 0) break; // 如果path指向有效的库文件，退出for循环
40.             // 如果vendor/lib/hw目录下没有库文件，查找/system/lib/hw目录下有没有：硬件id名.prop.so的库文件
41.             snprintf(path, sizeof(path), "%s/%s.%s.so",
42.                 HAL_LIBRARY_PATH1, name, prop);
43.             If(access(path, R_OK) == 0) break;
44.         } else {
45.             // 如果4个系统属性都没有定义，则使用默认的库名：/system/lib/hw/硬件id名.default.so
46.             snprintf(path, sizeof(path), "%s/%s.default.so",
47.                 HAL_LIBRARY_PATH1, name);
48.             If(access(path, R_OK) == 0) break;
49.         }
50.     }
51.     status = -ENOENT;
52.     if(i<HAL_VARIANT_KEYS_COUNT+1){
53.         status = load(class_id, path, module); // 难道是要加载前面查找到的so库？？
54.     }
55.     return status;
56. }
57.
58. static int load(const char *id, coundst char *path, const struct hw_module_t **pHmi){
59.     void *handle;
60.     struct hw_module_t * hmi;

```



```

61. // 通过dlopen打开so库
62. handle = dlopen(path, RTLD_NOW);
63. // sym的值为"HMI", 这个名字还有印象吗?
64. const char * sym = HAL_MODULE_INFO_SYM_AS_STR;
65. // 通过dlsym从打开的库里查找"HMI"这个符号, 如果在so代码里有定义的函数名或变量名为HMI, dlsym返回其地址hmi, 将该地址转化成hw_module_t类型,
    即, 硬件对象, 这招够狠, "杀鸡取卵"
66. hmi = (struct hw_module_t *)dlsym(handle, sym);
67. // 判断找到的硬件对象的id是否和要查找的id名一致, 不一致出错退出
68. // 取了卵还要验证下是不是自己要的"卵"
69. if(strcmp(id, hmi->) != 0){
70.     // 出错退出处理
71. }
72. // 将库的句柄保存到hmi硬件对象的dso成员里
73. hmi->dso = handle;
74. // 将硬件对象地址送给load函数者, 最终将硬件对象返回到了hw_get_module的调用者
75. *pHmi = hmi;
76. // 成功返回
77. }

```

通过上面代码的注释分析可知, 硬件对象声明的结构体代码被编译成了so库, 由于该结构体声明为const类型, 被so库包含在其静态代码段里, 要找到硬件对象, 首先要找到其对应的so库, 再通过dlopen, dlsym这种“杀鸡取卵”的方式找到硬件对象, 当然这儿的: “鸡”是指: so库, “卵”既: 硬件对象led_module_t结构。

在声明结构体led_module_t时, 其名字统一定义为了HMI, 而这么做的目的就是为了通过dlsym来查找led HAL Stub源码生成的so库里的"HMI"符号。现在很明了, 我们写的HAL Stub代码最终要编译so库文件, 并且库文件名为: led.default.so (当然可以设置四个系统属性之一来指定名字为: led.属性值.so), 并且库的所在目录为: /system/lib/hw/。

现在底层的实现部分基本上吃透了, 现在我们把目光放到调用者上, 根据本章开头介绍可知, 上层调用本地代码要使用JNI技术, 我们先来恶补下JNI的知识吧。