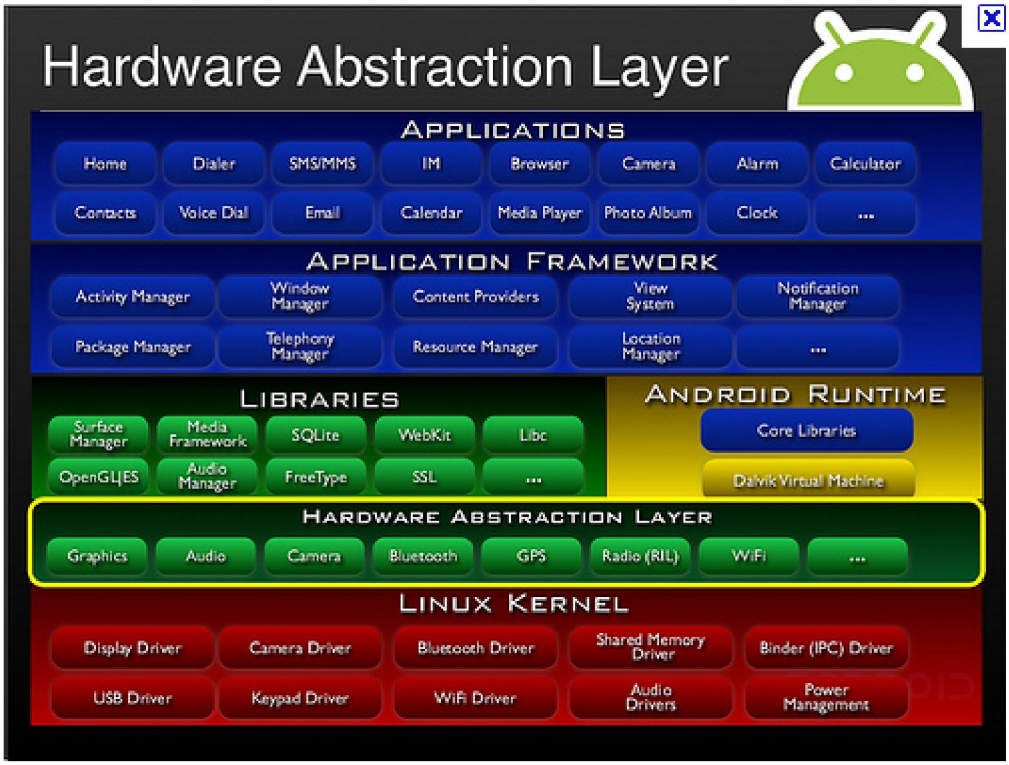
Android HAL 实例解析
华清远见讲师　刘洪涛

一、概述

本文希望通过分析台湾的 Jollen 的 mokoid 工程代码，和在 s5pc100 平台上实现过程种遇到的问题，解析 Andorid HAL 的开发方法。

二、HAL 介绍

现有 HAL 架构由 Patrick Brady (Google) 在 2008 Google I/O 演讲中提出的，如下图。



HAL **硬件抽象层** (Hardware Abstraction Layer)

Android 的 HAL 是为了保护一些硬件提供商的知识产权而提出的，是为了避开 linux 的 GPL 束缚。思路是把控制硬件的动作都放到了 Android HAL 中，而 linux driver 仅仅完成一些简单的数据交互作用，甚至把硬件寄存器空间直接映射到 user space。而 Android 是基于 Aparch 的 license，因此硬件厂商可以只提供二进制代码，所以说 Android 只是一个开放的平台，并不是一个开源的平台。也许也正是因为 Android 不遵从 GPL，所以 Greg Kroah-Hartman 才在 2.6.33 内核将 Andorid 驱动从 linux 中删除。GPL 和硬件厂商目前还是有着无法弥合的裂痕。Android 想要把这个问题处理好也是不容易的。HAL 的目的是为了把 Android framework 与 Linux kernel 完整隔开。让 Android 不至过度依赖 Linux kernel，有点像是 kernel independent 的意思，让 Android framework 的开发能在不考虑驱动程序的前提下进行发展

总结下来，Android HAL 存在的原因主要有：

1. 并不是所有的硬件设备都有标准的 linux kernel 的接口

2. KERNEL DRIVER 涉及到 GPL 的版权。某些设备制造商并不原因公开硬件驱动，所以才去用 HAL 方式绕过 GPL。

3. 针对某些硬件，Android 有一些特殊的需求

三、HAL 内容

1、HAL 主要的储存于以下目录：

（注意：HAL 在其它目录下也可以正常编译）

- libhardware_legacy/ - 旧的架构、采取链接库模块的观念进行
- libhardware/ - 新架构、调整为 HAL stub 的观念
- ril/ - Radio Interface Layer
- msm7k QUAL 平台相关

主要包含以下一些模块：Gps、Vibrator、Wifi、Copybit、Audio、Camera、Lights、Ril、Overlay 等。

2、两种 HAL 架构比较

目前存在两种 HAL 架构，位于 libhardware_legacy 目录下的“旧 HAL 架构”和位于 libhardware 目录下的“新 HAL 架构”。两种框架如下图所示。

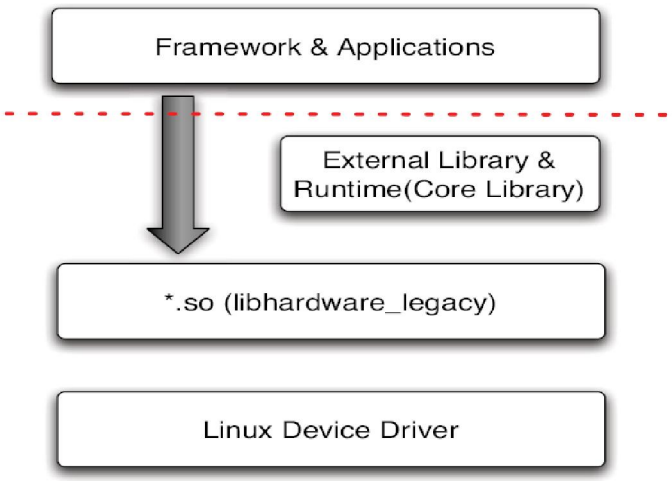


图 3.1 旧 HAL架构

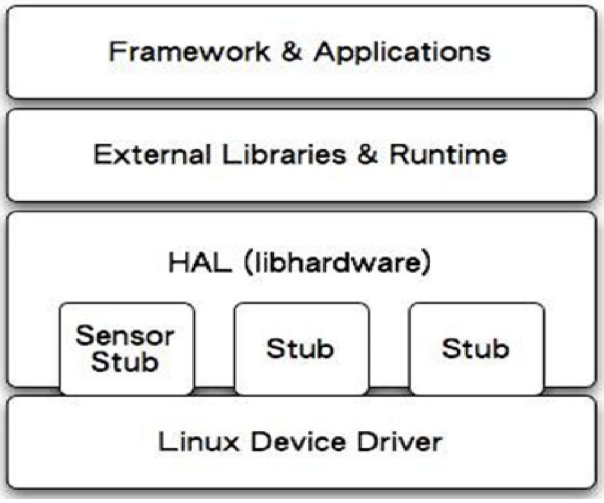


图 3.2　新 HAL 架构

libhardware_legacy 是将 *.so 文件当作 shared library 来使用，在 runtime（JNI 部份）以 direct function call 使用 HAL module。通过直接函数调用的方式，来操作驱动程序。当然，应用程序也可以不需要通过 JNI 的方式进行，直接加载 *.so （dlopen）的做法调用*.so 里的符号（symbol）也是一种方式。总而言之是没有经过封装，上层可以直接操作硬件。

现在的 libhardware 架构，就有 stub 的味道了。HAL stub 是一种代理人（proxy）的概念，stub 虽然仍是以 *.so 檔的形式存在，但 HAL 已经将 *.so 档隐藏起来了。Stub 向 HAL 提供操作函数（operations），而 runtime 则是向 HAL 取得特定模块（stub）的 operations，再 callback 这些操作函数。这种以 indirect function call 的架构，让 HAL stub 变成是一种包含关系，即 HAL 里包含了许许多多的 stub（代理人）。Runtime 只要说明类型，即 module ID，就可以取得操作函数。对于目前的 HAL，可以认为 Android 定义了 HAL 层结构框架，通过几个接口访问硬件从而统一了调用方式。

3. HAL_legacy 和 HAL 的对比

HAL_legacy：旧式的 HAL 是一个模块，采用共享库形式，在编译时会调用到。由于采用 function call 形式调用，因此可被多个进程使用，但会被 mapping 到多个进程空间中，造成浪费，同时需要考虑代码能否安全重入的问题（thread safe）。

HAL：新式的 HAL 采用 HAL module 和 HAL stub 结合形式，HAL stub 不是一个 share library，编译时上层只拥有访问 HAL stub 的函数指针，并不需要 HAL stub。上层通过 HAL module 提供的统一接口获取并操作 HAL stub，so 文件只会被 mapping 到一个进程，也不存在重复 mapping 和重入问题。

下面结合实例来分析 HAL 编程方法。

4. HAL module 架构

HAL moudle 主要分为三个结构：

```
struct hw_module_t;
struct hw_module_methods_t;
struct hw_device_t;
```

hw_module_t 定义如下：

```
typedef struct hw_module_t {
    /** tag must be initialized to HARDWARE_MODULE_TAG */
    uint32_t tag;

    /** major version number for the module */
    uint16_t version_major;

    /** minor version number of the module */
    uint16_t version_minor;

    /** Identifier of module */
    const char *id;

    /** Name of this module */
    const char *name;

    /** Author/owner/implementor of the module */
    const char *author;

    /** Modules methods */
    struct hw_module_methods_t* methods; // 硬件模块的方法

    /** module's dso */
    void* dso;

    /** padding to 128 bytes, reserved for future use */
    uint32_t reserved[32-7];

} hw_module_t;
```

硬件模块方法结构体 hw_module_methods_t 定义如下：

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
                struct hw_device_t** device);

} hw_module_methods_t;
```

只定义了一个 open 方法，其中调用的设备结构体参数 hw_device_t 定义如下：

```
typedef struct hw_device_t {
    /** tag must be initialized to HARDWARE_DEVICE_TAG */
    uint32_t tag;

    /** version number for hw_device_t */
    uint32_t version;

    /** reference to the module this device belongs to */
    struct hw_module_t* module;

    /** padding reserved for future use */
    uint32_t reserved[12];

    /** Close this device */
    int (*close)(struct hw_device_t* device);

} hw_device_t;
```

他们的继承关系如下图：

open()

\
\
\
hw_module_t
---
tag:uint32_t
, , , , ,
\
\
\
\
hw_device_t
---

close()

5HAL 使用方法

（1）Native code 通过 hw_get_module 调用获取 HAL stub：

```
hw_get_module(LED_HARDWARE_MODULE_ID, (const hw_module_t**)&module)
```

（2）通过继承 hw_module_methods_t 的 callback 来 open 设备：

```
module->methods->open(module,
```

```
            LED_HARDWARE_MODULE_ID, (struct hw_device_t**)device);
```

（3）通过继承 hw_device_t 的 callback 来控制设备：

```
sLedDevice->set_on(sLedDevice, led);
sLedDevice->set_off(sLedDevice, led);
```

6 HAL stub 编写方法

（1）定义自己的 HAL 结构体，编写头文件 led.h, hardware/hardware.h

```
struct led_module_t {
    struct hw_module_t common;
};

struct led_control_device_t {
    struct hw_device_t common;

    int fd;                 /* file descriptor of LED device */

    /* supporting control APIs go here */
    int (*set_on)(struct led_control_device_t *dev, int32_t led);
    int (*set_off)(struct led_control_device_t *dev, int32_t led);
};
```

2）设计 led.c 完成功能实现和 HAL stub 注册

（2.1）led_module_methods 继承 hw_module_methods_t，实现 open 的 callback

```
struct hw_module_methods_t led_module_methods = {
    open: led_device_open
};
```

（2.2）用 HAL_MODULE_INFO_SYM 实例 led_module_t，这个名称不可修改

tag：需要制定为 HARDWARE_MODULE_TAG

id：指定为 HAL Stub 的 module ID

methods：struct hw_module_methods_t，为 HAL 所定义的「method」

```
const struct led_module_t HAL_MODULE_INFO_SYM = {
    common: {
        tag: HARDWARE_MODULE_TAG,
        version_major: 1,
        version_minor: 0,
        id: LED_HARDWARE_MODULE_ID,
        name: "Sample LED Stub",
        author: "The Mokoid Open Source Project",
        methods: &led_module_methods,
    }
```

```
    /* supporting APIs go here. */
};
```

（2.3）open 是一个必须实现的 callback API，负责申请结构体空间，填充信息，注册具体操作 API 接口,打开 Linux 驱动。

由于存在多重继承关系，只需对子结构体 hw_device_t 对象申请空间即可。

```
int led_device_open(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device)
{
        struct led_control_device_t *dev;
        dev = (struct led_control_device_t *)malloc(sizeof(*dev));
        memset(dev, 0, sizeof(*dev));
        dev->common.tag = HARDWARE_DEVICE_TAG;
        dev->common.version = 0;
        dev->common.module = module;
        dev->common.close = led_device_close;
        dev->set_on = led_on;
        dev->set_off = led_off;
        *device = &dev->common;
        /*
          * Initialize Led hardware here.
          */
         dev->fd = open(LED_DEVICE, O_RDONLY);
        if (dev->fd < 0)
            return -1;

        led_off(dev, LED_C608);
        led_off(dev, LED_C609);
success:
        return 0;
}
```

（2.4）填充具体 API 操作代码

```
int led_on(struct led_control_device_t *dev, int32_t led)
{
        int fd;
        LOGI("LED Stub: set %d on.", led);
        fd = dev->fd;
        switch (led) {
                case LED_C608:
                        ioctl(fd, 1, &led);
                        break;
                case LED_C609:
```

```
                ioctl(fd, 1, &led);
                break;
        default:
                return -1;
    }
return 0;
}

int led_off(struct led_control_device_t *dev, int32_t led)
{
    int fd;
    LOGI("LED Stub: set %d off.", led);
    fd = dev->fd;
    switch (led) {
        case LED_C608:
                ioctl(fd, 2, &led);
                break;
        case LED_C609:
                ioctl(fd, 2, &led);
                break;
        default:
                return -1;
    }
    return 0;
}
```

8. Android HAL 是如何被调用的

Android 对硬件的调用， google 推荐使用 HAL 的方式进行调用，对于 Andriod HAL 的写法，可以参考 android 源码里的 hardware 目录下几个模块的模版。

在看 HAL 的编写方法的过程中，会发现整个模块貌似没有一个入口。一般说来模块都要有个入口，比如应用程序有 main 函数，可以为加载器进行加载执行，dll 文件有 dllmain ，而对于我们自己写的动态链接库，我们可以对库中导出的任何符号进行调用。

问题来了，Android 中的 HAL 是比较具有通用性的，需要上层的函数对其进行加载调用，Android 的 HAL 加载器是如何实现对不同的 Hardware Module 进行通用性的调用的呢？

带着这个疑问查看 Android 源码，会发现 Android 中实现调用 HAL 是通过 hw_get_module 实现的。

```
int hw_get_module(const char *id, const struct hw_module_t **module);
```

这是其函数原型，id 会指定 Hardware 的 id ,这是一个字符串,比如 sensor 的 id 是

#define SENSORS_HARDWARE_MODULE_ID "sensors" ，如果找到了对应的 hw_module_t 结构体，会将其指针放入 *module 中。看看它的实现

```
    /* Loop through the configuration variants looking for a module */
    for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
        if (i < HAL_VARIANT_KEYS_COUNT) {
 // 获取 ro.hardware/ro.product.board/ro.board.platform/ro.arch 等 key 的值。
            if (property_get(variant_keys[i], prop, NULL) == 0) {
                continue;
            }
            snprintf(path, sizeof(path), "%s/%s.%s.so",
                    HAL_LIBRARY_PATH, id, prop);
              // 如果开发板叫做 mmdroid, 那么这里的 path 就是 system/lib/hw/sensor.mmdroid.so
        } else {
            snprintf(path, sizeof(path), "%s/%s.default.so",
                    HAL_LIBRARY_PATH, id);// 默认会加载 /system/lib/hw/sensor.default.so
         }
        if (access(path, R_OK)) {
            continue;
        }
        /* we found a library matching this id/variant */
```

```
            break;
        }
        status = -ENOENT;
        if (i < HAL_VARIANT_KEYS_COUNT+1) {
            /* load the module, if this fails, we're doomed, and we should not try
             * to load a different variant. */
            status = load(id, path, module);// 调用 load 函数打开动态链接库
        }
```

获取了动态链接库的路径之后，就会调用 load 函数打开它，下面会打开它。

奥秘在 load 中

```
static int load(const char *id,
        const char *path,
        const struct hw_module_t **pHmi)
{
    int status;
    void *handle;
    struct hw_module_t *hmi;

    /*
     * load the symbols resolving undefined symbols before
     * dlopen returns. Since RTLD_GLOBAL is not or'd in with
     * RTLD_NOW the external symbols will not be global
     */
```

```
    handle = dlopen(path, RTLD_NOW);// 打开动态库
    if (handle == NULL) {
        char const *err_str = dlerror();
        LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
        status = -EINVAL;
        goto done;
    }

    /* Get the address of the struct hal_module_info. */
    const char *sym = HAL_MODULE_INFO_SYM_AS_STR;// 被定义为了“ HMI ”
    hmi = (struct hw_module_t *)dlsym(handle, sym);// 查找“ HMI ”这个导出符号，并获取其地址
    if (hmi == NULL) {
        LOGE("load: couldn't find symbol %s", sym);
        status = -EINVAL;
        goto done;
    }
/* Check that the id matches */
// 找到了 hw_module_t 结构 !!!
    if (strcmp(id, hmi->id) != 0) {
        LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
        status = -EINVAL;
```

```
            goto done;
        }
        hmi->dso = handle;
        /* success */
        status = 0;
        done:
        if (status != 0) {
            hmi = NULL;
            if (handle != NULL) {
                dlclose(handle);
                handle = NULL;
            }
        } else {
            LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
                    id, path, *pHmi, handle);
        }
    // 凯旋而归
        *pHmi = hmi;
         return status;
}
```

从上面的代码中，会发现一个很奇怪的宏 HAL_MODULE_INFO_SYM_AS_STR ，它直接被定义为了 #define HAL_MODULE_INFO_SYM_AS_STR "HMI" ，为何根据它就能从动态链接库中找到这个 hw_module_t 结构体呢？我们查看一下我们用到的 hal 对应的 so 就可以了，在 linux 中可以使用 readelf XX.so －s 查看。

Symbol table '.dynsym' contains 28 entries:

```
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000594     0 SECTION LOCAL  DEFAULT    7
     2: 00001104     0 SECTION LOCAL  DEFAULT   13
     3: 00000000     0 FUNC    GLOBAL DEFAULT  UND ioctl
     4: 00000000     0 FUNC    GLOBAL DEFAULT  UND strerror
     5: 00000b84     0 NOTYPE  GLOBAL DEFAULT  ABS __exidx_end
     6: 00000000     0 OBJECT  GLOBAL DEFAULT  UND __stack_chk_guard
     7: 00000000     0 FUNC    GLOBAL DEFAULT  UND
__aeabi_unwind_cpp_pr0
     8: 00000000     0 FUNC    GLOBAL DEFAULT  UND __errno
     9: 00001188     0 NOTYPE  GLOBAL DEFAULT  ABS _bss_end__
    10: 00000000     0 FUNC    GLOBAL DEFAULT  UND malloc
    11: 00001188     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start__
    12: 00000000     0 FUNC    GLOBAL DEFAULT  UND
__android_log_print
    13: 00000b3a     0 NOTYPE  GLOBAL DEFAULT  ABS __exidx_start
    14: 00000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail
    15: 00001188     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_end__
    16: 00001188     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
    17: 00000000     0 FUNC    GLOBAL DEFAULT  UND memset
    18: 00000000     0 FUNC    GLOBAL DEFAULT  UND __aeabi_uidiv
```

```
    19: 00001188      0 NOTYPE  GLOBAL DEFAULT   ABS __end__
    20: 00001188      0 NOTYPE  GLOBAL DEFAULT   ABS _edata
    21: 00001188      0 NOTYPE  GLOBAL DEFAULT   ABS _end
    22: 00000000      0 FUNC    GLOBAL DEFAULT   UND open
    23: 00080000      0 NOTYPE  GLOBAL DEFAULT   ABS _stack
    24: 00001104    128 OBJECT  GLOBAL DEFAULT    13 HMI
    25: 00001104      0 NOTYPE  GLOBAL DEFAULT    13 __data_start
    26: 00000000      0 FUNC    GLOBAL DEFAULT   UND close
    27: 00000000      0 FUNC    GLOBAL DEFAULT   UND free
```

从上面中，第 24 个符号，名字就是“ HMI ”，对应于 hw_module_t 结构体。再去对照一下 HAL 的代码。

```
/*
  * The COPYBIT Module
  */
struct copybit_module_t HAL_MODULE_INFO_SYM = {
    common: {
        tag: HARDWARE_MODULE_TAG,
        version_major: 1,
        version_minor: 0,
        id: COPYBIT_HARDWARE_MODULE_ID,
        name: "QCT MSM7K COPYBIT Module",
        author: "Google, Inc.",
```

```
        methods: &copybit_module_methods
    }
};
```

这里定义了一个名为 HAL_MODULE_INFO_SYM 的 copybit_module_t 的结构体，common 成员为 hw_module_t 类型。注意这里的 HAL_MODULE_INFO_SYM 变量必须为这个名字，这样编译器才会将这个结构体的导出符号变为“ HMI ”，这样这个结构体才能被 dlsym 函数找到！

综上，我们知道了 andriod HAL 模块也有一个通用的入口地址，这个入口地址就是 HAL_MODULE_INFO_SYM 变量，通过它，我们可以访问到 HAL 模块中的所有想要外部访问到的方法。

1，源代码和目标位置
源代码：/hardware/libhardware 目录，该目录的目录结构如下：
/hardware/libhardware/hardware.c 编译成 libhardware.so，目标位置为/system/lib 目录
/hardware/libhardware/include/hardware 目录下包含如下头文件：
hardware.h 通用硬件模块头文件
copybit.h copybit 模块头文件
gralloc.h gralloc 模块头文件
lights.h 背光模块头文件
overlay.h overlay 模块头文件
qemud.h qemud 模块头文件
sensors.h 传感器模块头文件
/hardware/libhardware/modules 目录下定义了很多硬件模块
这些硬件模块都编译成 xxx.xxx.so，目标位置为 /system/lib/hw 目录

2，HAL 层的实现方式
JNI-> 通用硬件模块 -> 硬件模块 -> 内核驱动接口
具体一点：JNI->libhardware.so->xxx.xxx.so->kernel
具体来说：android frameworks 中 JNI 调用/hardware/libhardware/hardware.c 中定义的 hw_get_module 函数来获取硬件模块，然后调用硬件模块中的方法，硬件模块中的方法直接调用内核接口完成相关功能

3, 通用硬件模块 (libhardware.so)
(1) 头文件为：/hardware/libhardware/include/hardware/hardware.h
头文件中主要定义了通用硬件模块结构体 hw_module_t，声明了 JNI 调用的接口函数 hw_get_module
hw_module_t 定义如下：

```
typedef struct hw_module_t {
    /** tag must be initialized to HARDWARE_MODULE_TAG */
```

```
    uint32_t tag;

    /** major version number for the module */
    uint16_t version_major;

    /** minor version number of the module */
    uint16_t version_minor;

    /** Identifier of module */
    const char *id;

    /** Name of this module */
    const char *name;

    /** Author/owner/implementor of the module */
    const char *author;

    /** Modules methods */
    struct hw_module_methods_t* methods; // 硬件模块的方法

    /** module's dso */
    void* dso;

    /** padding to 128 bytes, reserved for future use */
    uint32_t reserved[32-7];

} hw_module_t;
```

硬件模块方法结构体 hw_module_methods_t 定义如下：

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
            struct hw_device_t** device);

} hw_module_methods_t;
```

只定义了一个 open 方法，其中调用的设备结构体参数 hw_device_t 定义如下：

```
typedef struct hw_device_t {
    /** tag must be initialized to HARDWARE_DEVICE_TAG */
    uint32_t tag;

    /** version number for hw_device_t */
    uint32_t version;

    /** reference to the module this device belongs to */
    struct hw_module_t* module;
```

```
        /** padding reserved for future use */
        uint32_t reserved[12];

        /** Close this device */
        int (*close)(struct hw_device_t* device);

} hw_device_t;
```

hw_get_module 函数声明如下：

```
int hw_get_module(const char *id, const struct hw_module_t **module);
```

参数 id 为模块标识，定义在/hardware/libhardware/include/hardware 目录下的硬件模块头文件中，

参数 module 是硬件模块地址，定义了/hardware/libhardware/include/hardware/hardware.h 中

(2)hardware.c 中主要是定义了 hw_get_module 函数如下：

```
#define HAL_LIBRARY_PATH "/system/lib/hw"
static const char *variant_keys[] = {
    "ro.hardware",
    "ro.product.board",
    "ro.board.platform",
    "ro.arch"
};
static const int HAL_VARIANT_KEYS_COUNT =
    (sizeof(variant_keys)/sizeof(variant_keys[0]));

int hw_get_module(const char *id, const struct hw_module_t **module)
{
    int status;
    int i;
    const struct hw_module_t *hmi = NULL;
    char prop[PATH_MAX];
    char path[PATH_MAX];
    for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++)
    {
        if (i < HAL_VARIANT_KEYS_COUNT)
        {
            if (property_get(variant_keys[i], prop, NULL) == 0)
            {
                continue;
            }
            snprintf(path, sizeof(path), "%s/%s.%s.so",
                HAL_LIBRARY_PATH, id, prop);
            }
```

```
                else
                {
                    snprintf(path, sizeof(path), "%s/%s.default.so",
                             HAL_LIBRARY_PATH, id);
                }
                if (access(path, R_OK))
                {
                    continue;
                }
            /* we found a library matching this id/variant */
                break;
            }

            status = -ENOENT;
            if (i < HAL_VARIANT_KEYS_COUNT+1) {
        /* load the module, if this fails, we're doomed, and we should not try
             * to load a different variant. */
                status = load(id, path, module);
            }

            return status;
}
```

从源代码我们可以看出，hw_get_module 完成的主要工作是根据模块 id 寻找硬件模块动态连接库地址，然后调用 load 函数去打开动态连接库

四、mokoid 工程代码下载与结构分析

1、mokid 项目概述

modkoid 工程提供了一个 LedTest 示例程序，是台湾的 Jollen 用于培训的。对于理解 android 层次结构、Hal 编程方法都非常有意义。

2、下载方法

#svn checkout http://mokoid.googlecode.com/svn/trunk/mokoid-read-only

3、结构分析

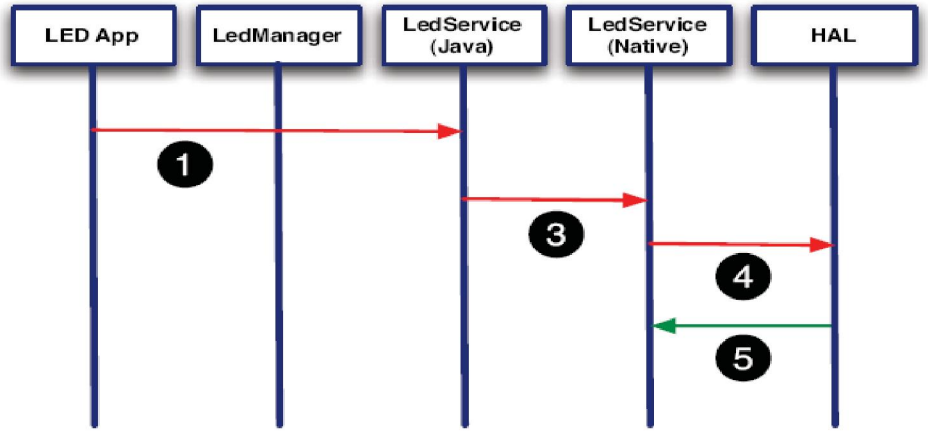
```
|-- Android.mk
|-- apps          //两种应用测试方法
|     |-- Android.mk
|     |-- LedClient       //直接调用 service 来调用 jni
|     |     |-- AndroidManifest.xml
|     |     |-- Android.mk
|     |     `-- src
|     |           `-- com
|     |                 `-- mokoid
|     |                       `-- LedClient
|     |                             `-- LedClient.java        //第 1 种方式应用程序实现代码
```

```
|   `-- LedTest              //通过 manager 来调用 jni
|       |-- AndroidManifest.xml
|       |-- Android.mk
|       `-- src
|           `-- com
|               `-- mokoid
|                   `-- LedTest
|                       |-- LedSystemServer.java    //开启了一个后台 service，下文会有解释
|                       `-- LedTest.java       //第 2 种方式应用程序实现代码
|-- dma6410xp     //这个目录可以不要
|   |-- AndroidBoard.mk
|   |-- AndroidProducts.mk
|   |-- BoardConfig.mk
|   |-- dma6410xp.mk
|   |-- init.dma6410xp.rc
|   |-- init.goldfish.sh
|   `-- init.rc
|-- frameworks         //框架代码
|   |-- Android.mk
|   `-- base
|       |-- Android.mk
|       |-- core
|       |   `-- java
|       |       `-- mokoid
|       |           `-- hardware
|       |               |-- ILedService.aidl
|       |               `-- LedManager.java        //实现了 Manager，给第 2 种方法用
|       `-- service
|           |-- Android.mk
|           |-- com.mokoid.server.xml
|           |-- java
|           |   `-- com
|           |       `-- mokoid
|           |           `-- server
|           |               `-- LedService.java       //Framework service 代码
|           `-- jni
|               |-- Android.mk
|               `-- com_mokoid_server_LedService.cpp    //jni 代码
|-- hardware
|   |-- Android.mk
|   |-- libled
|   |   |-- Android.mk
|   |   `-- libled.c
|   `-- modules
```

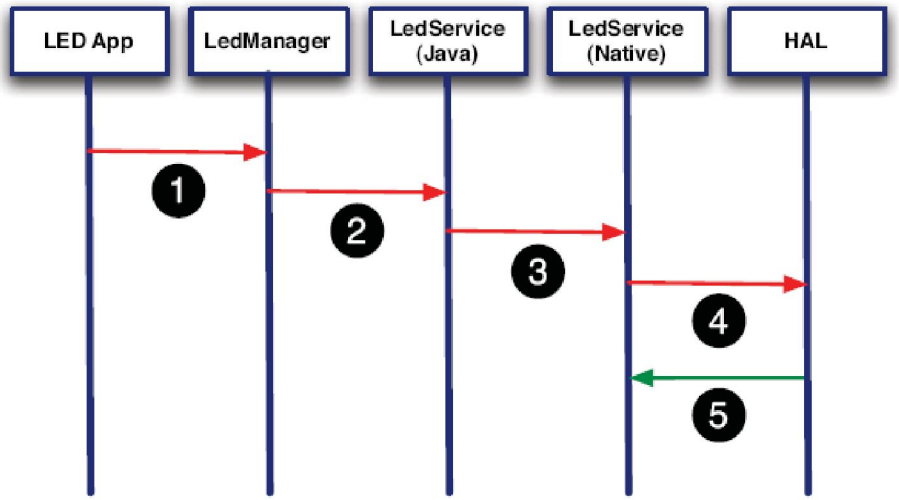
```
|           |-- Android.mk
|           |-- include
|           |    `-- mokoid
|           |         `-- led.h
|           `-- led
|                |-- Android.mk
|                `-- led.c          //led stub  硬件控制代码
`-- README.txt
```

Android 的 HAL 的实现需要通过 JNI(Java Native Interface)，JNI 简单来说就是 java 程序可以调用 C/C++写的动态链接库，这样的话，HAL 可以使用 C/C++语言编写，效率更高。在 Android 下访问 HAL 大致有以下两种方式：

（1）Android 的 app 可以直接通过 service 调用.so 格式的 jni



（2）经过 Manager 调用 service



上面两种方法应该说是各有优缺点，第一种方法简单高效，但不正规。第二种方法实

现起来比较复杂，但更符合目前的 Android 框架。第二种方法中，LegManager 和 LedService（java）在两个进程中，需要通过进程通讯的方式来通讯。

mokoid 工程中实现了上述两种方法。下面将详细介绍这两种方法的实现原理。

4、第一种方法：直接调用 service 方法的实现过程

下面分析第一种方法中，各层的关键代码。

（1）HAL 层

一般来说 HAL moudle 需要涉及的是三个关键结构体：

```
struct hw_module_t;
struct hw_module_methods_t;
struct hw_device_t;
```

下面结合代码说明这 3 个结构的用法。部分代码经过修改，后面的章节会给出修改的原因。

文件：mokoid-read-only/hardware/modules/include/mokoid/led.h

```
struct led_module_t {
    struct hw_module_t common;
};
//HAL 规定不能直接使用 hw_module_t 结构，因此需要做这么一个继承。

struct led_control_device_t {
//自定义的一个针对 Led 控制的结构，包含 hw_device_t 和支持的 API 操作
    struct hw_device_t common;
    /* attributes */
    int fd;   //可用于具体的设备描述符
    /* supporting control APIs go here */
    int (*set_on)(struct led_control_device_t *dev, int32_t led);
    int (*set_off)(struct led_control_device_t *dev, int32_t led);
};
#define LED_HARDWARE_MODULE_ID "led"
//定义一个 MODULE_ID，HAL 层可以根据这个 ID 找到我们这个 HAL stub
```

文件：mokoid-read-only/hardware/modules/led/led.c

```
view plaincopy to clipboardprint?
#define LOG_TAG "MokoidLedStub"
#include <hardware/hardware.h>
#include <fcntl.h>
#include <errno.h>
```

```
#include <cutils/log.h>
#include <cutils/atomic.h>
//#include <mokoid/led.h>
#include "../include/mokoid/led.h"
/******************************************************************************/
int fd;              //硬件 led 的设备描述符 。你也可以用 led_control_device_t 结构中定义的
fd
#define GPG3DAT2_ON 0x4800                          //ioctl 控制命令
#define GPG3DAT2_OFF 0x4801
int led_device_close(struct hw_device_t* device)
{
    struct led_control_device_t* ctx = (struct led_control_device_t*)device;
    if (ctx) {
        free(ctx);
    }
    close(fd);
    return 0;
}
int led_on(struct led_control_device_t *dev, int32_t led)
{
    LOGI("LED Stub: set %d on.", led);
    ioctl(fd,GPG3DAT2_ON,NULL);                 //控制 Led 亮灭，和硬件相关
    return 0;
}
int led_off(struct led_control_device_t *dev, int32_t led)
{
    LOGI("LED Stub: set %d off.", led);
    ioctl(fd,GPG3DAT2_OFF,NULL);
    return 0;
}
static int led_device_open(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device)
{
    struct led_control_device_t *dev;
    dev = (struct led_control_device_t *)malloc(sizeof(*dev));
    memset(dev, 0, sizeof(*dev));
    dev->common.tag =   HARDWARE_DEVICE_TAG;
    dev->common.version = 0;
    dev->common.module = module;
    dev->common.close = led_device_close;
    dev->set_on = led_on;           //实例化支持的操作
    dev->set_off = led_off;
    *device = &dev->common;       //将实例化后的 led_control_device_t 地址返回给 jni
层 ,这样 jni 层就可以直接调用 led_on、led_off、led_device_close 方法了。
```

```
    if((fd=open("/dev/led",O_RDWR))==-1)          //打开硬件设备
    {
        LOGE("LED open error");
    }
    else
        LOGI("open ok");
success:
    return 0;
}
static struct hw_module_methods_t led_module_methods = {
    open: led_device_open
};
const struct led_module_t HAL_MODULE_INFO_SYM = {
//定义这个对象等于向系统注册了一个 ID 为 LED_HARDWARE_MODULE_ID 的 stub。注意这里 HAL_MODULE_INFO_SYM 的名称不能改。(HMI)
    common: {
        tag: HARDWARE_MODULE_TAG,
        version_major: 1,
        version_minor: 0,
        id: LED_HARDWARE_MODULE_ID,
        name: "Sample LED Stub",
        author: "The Mokoid Open Source Project",
        methods: &led_module_methods,   //实现了一个 open 的方法供 jni 层调用，
                                        //从而实例化 led_control_device_t
    }
    /* supporting APIs go here */
};
```

（2）JNI 层

文件 mokoid-read-only/frameworks/base/service/jni/com/mokoid/server/LedService.cpp

```
struct led_control_device_t *sLedDevice = NULL;

static jboolean mokoid_setOn(JNIEnv* env, jobject thiz, jint led)
{
    LOGI("LedService JNI: mokoid_setOn() is invoked.");

    if (sLedDevice == NULL) {
        LOGI("LedService JNI: sLedDevice was not fetched correctly.");
        return -1;
    } else {
        return sLedDevice->set_on(sLedDevice, led);//调用 hal 层的注册的方法
    }
}
```

```
static jboolean mokoid_setOff(JNIEnv* env, jobject thiz, jint led)
{
	LOGI("LedService JNI: mokoid_setOff() is invoked.");


	if (sLedDevice == NULL) {
		LOGI("LedService JNI: sLedDevice was not fetched correctly.");
		return -1;
	} else {
		return sLedDevice->set_off(sLedDevice, led); //调用 hal 层的注册的方法
	}
}

/** helper APIs */
static inline int led_control_open(const struct hw_module_t* module,
		struct led_control_device_t** device) {
	return module->methods->open(module,
			LED_HARDWARE_MODULE_ID, (struct hw_device_t**)device);
//这个过程非常重要，jni 通过 LED_HARDWARE_MODULE_ID 找到对应的 stub
}

static jboolean mokoid_init(JNIEnv *env, jclass clazz)
{
	led_module_t* module;
	 LOGI("jni init----------.");
if (hw_get_module(LED_HARDWARE_MODULE_ID, (const hw_module_t**)&module) ==
0) {
//根据 LED_HARDWARE_MODULE_ID 找到 hw_module_t，参考 hal 层的实现
		LOGI("LedService JNI: LED Stub found.");
		if (led_control_open(&module->common, &sLedDevice) == 0) {
	//通过 hw_module_t 找到 led_control_device_t
			LOGI("LedService JNI: Got Stub operations.");
			return 0;
		}
	}

	LOGE("LedService JNI: Get Stub operations failed.");
	return -1;
}

/*
 * Array of methods.
* Each entry has three fields: the name of the method, the method
```

```
 * signature, and a pointer to the native implementation.
 */
static const JNINativeMethod gMethods[] = {
    { "_init",          "()Z",    (void *)mokoid_init },//Framework 层调用_init 时促发
    { "_set_on",     "(I)Z", (void *)mokoid_setOn },
    { "_set_off",     "(I)Z", (void *)mokoid_setOff },
};
/*
*JNINativeMethod 是 jni 层注册的方法，Framework 层可以使用这些方法
*_init 、_set_on、_set_off 是在 Framework 中调用的方法名称，函数的类型及返回值如下：
*（）Z    无参数      返回值为 bool 型
* (I)Z    整型参数   返回值为 bool 型
*/
static int registerMethods(JNIEnv* env) {
        static const char* const kClassName =
      "com/mokoid/server/LedService";//注意：必须和你 Framework 层的 service 类名相同
        jclass clazz;
       /* look up the class */
        clazz = env->FindClass(kClassName);
        if (clazz == NULL) {
            LOGE("Can't find class %s\n", kClassName);
            return -1;
        }
    /* register all the methods */
        if (env->RegisterNatives(clazz, gMethods,
                        sizeof(gMethods) / sizeof(gMethods[0])) != JNI_OK)
        {
            LOGE("Failed registering methods for %s\n", kClassName);
            return -1;
        }
    /* fill out the rest of the ID cache */
        return 0;
}
jint JNI_OnLoad(JavaVM* vm, void* reserved) {//Framework 层加载 jni 库时调用
        JNIEnv* env = NULL;
        jint result = -1;
        LOGI("JNI_OnLoad LED");
            if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
            LOGE("ERROR: GetEnv failed\n");
            goto fail;
        }
            assert(env != NULL);
        if (registerMethods(env) != 0) { //注册你的 JNINativeMethod
            LOGE("ERROR: PlatformLibrary native registration failed\n");
```

```
            goto fail;
        }
        /* success -- return valid version number */
        result = JNI_VERSION_1_4;
fail:
        return result;
}
```

（3）service （属于 Framework 层）
文件：frameworks/base/service/java/com/mokoid/server/LedService.java

```
package com.mokoid.server;
import android.util.Config;
import android.util.Log;
import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.RemoteException;
import android.os.IBinder;
import mokoid.hardware.ILedService;
public final class LedService extends ILedService.Stub {
//对于这种直接模式不需要进程通讯，所以可以不加 extends ILedService.Stub，此处加上
主要是为了后面的第二种模式.
    static {
        System.load("/system/lib/libmokoid_runtime.so");//加载 jni 的动态库
    }
    public LedService() {
        Log.i("LedService", "Go to get LED Stub...");
    _init();
    }
    /*
     * Mokoid LED native methods.
     */
    public boolean setOn(int led) {
        Log.i("MokoidPlatform", "LED On");
    return _set_on(led);
    }
    public boolean setOff(int led) {
        Log.i("MokoidPlatform", "LED Off");
    return _set_off(led);
    }
    private static native boolean _init();                  //声明 jni 库可以提供的方法
    private static native boolean _set_on(int led);
    private static native boolean _set_off(int led);
```

```
}
```

（4）APP 测试程序 （属于 APP 层）

文件：apps/LedClient/src/com/mokoid/LedClient/LedClient.java

```
package com.mokoid.LedClient;
import com.mokoid.server.LedService;// 导入 Framework 层的 LedService
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class LedClient extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Call an API on the library.
    LedService ls = new LedService();   //实例化 LedService
    ls.setOn(1);                        //通过 LedService 提供的方法，控制底层硬件
    ls.setOff(2);

        TextView tv = new TextView(this);
        tv.setText("LED 1 is on. LED 2 is off.");
        setContentView(tv);
    }
}
```

5、第二种方法：经过 Manager 调用 service

HAL、JNI 两层和第一种方法一样，所以后面只分析其他的层次。

（1）Manager （属于 Framework 层）

APP 通过这个 Manager 和 service 通讯。

文件：mokoid-read-only /frameworks/base/core/java/mokoid/hardware/LedManager.java

```
package mokoid.hardware;
import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.Parcelable;
import android.os.ParcelFileDescriptor;
import android.os.Process;
import android.os.RemoteException;
import android.os.Handler;
```

```
import android.os.Message;
import android.os.ServiceManager;
import android.util.Log;
import mokoid.hardware.ILedService;

/*
 * Class that lets you access the Mokoid LedService.
 */
public class LedManager
{
    private static final String TAG = "LedManager";
    private ILedService mLedService;
    public LedManager() {
        mLedService= ILedService.Stub.asInterface(ServiceManager.getService("led"));

/*  这一步是关键，利用 ServiceManager 获取到 LedService，从而调用它提供的方法。这要求 LedService 必须已经添加到了 ServiceManager 中，这个过程将在 App 中的一个 service 进程中完成。  */

    if (mLedService != null) {
            Log.i(TAG, "The LedManager object is ready.");
    }
    }
    public boolean LedOn(int n) {
        boolean result = false;
        try {
            result = mLedService.setOn(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOn:", e);
        }
        return result;
    }
    public boolean LedOff(int n) {
        boolean result = false;
        try {
            result = mLedService.setOff(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOff:", e);
        }
        return result;
    }
}
```

因为 LedService 和 LedManager 在不同的进程，所以要考虑到进程通讯的问题。Manager

通过增加一个 aidl 文件来描述通讯接口。

文件：mokoid-read-only/frameworks/base/core/java/mokoid/hardware/ILedService.aidl

```
package mokoid.hardware;
interface ILedService
{
    boolean setOn(int led);
    boolean setOff(int led);
}
//系统的 aidl 工具会将 ILedService.aidl 文件 ILedService.java 文件，实现了 ILedService
```

（2）SystemServer （属于 APP 层）

文件：mokoid-read-only/apps/LedTest/src/com/mokoid/LedTest/LedSystemServer.java

```
package com.mokoid.LedTest;
import com.mokoid.server.LedService;
import android.os.IBinder;
import android.os.ServiceManager;
import android.util.Log;
import android.app.Service;
import android.content.Context;
import android.content.Intent;

public class LedSystemServer extends Service {
//注意这里的 Service 是 APP 中的概念，代表一个后台进程。注意区别和 Framework 中的 service 的概念。
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    public void onStart(Intent intent, int startId) {
        Log.i("LedSystemServer", "Start LedService...");

    /* Please also see SystemServer.java for your interests. */
    LedService ls = new LedService();
        try {
          ServiceManager.addService("led", ls);      // 将 LedService 添加到 ServiceManager 中
        } catch (RuntimeException e) {
            Log.e("LedSystemServer", "Start LedService failed.");
        }
    }
}
```

（3）APP 测试程序（属于 APP 层）

文件：mokoid-read-only/apps/LedTest/src/com/mokoid/LedTest/LedTest.java

```
package com.mokoid.LedTest;
import mokoid.hardware.LedManager;
import com.mokoid.server.LedService;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;
import android.widget.Button;
import android.content.Intent;
import android.view.View;

public class LedTest extends Activity implements View.OnClickListener {
    private LedManager mLedManager = null;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Start LedService in a seperated process.
        startService(new Intent("com.mokoid.systemserver"));//开启后台进程
        Button btn = new Button(this);
        btn.setText("Click to turn LED 1 On");
     btn.setOnClickListener(this);
        setContentView(btn);
    }
    public void onClick(View v) {
        // Get LedManager.
        if (mLedManager == null) {
        Log.i("LedTest", "Creat a new LedManager object.");
        mLedManager = new LedManager();   //实例化 Framework 层中的 Manager
        }
        if (mLedManager != null) {
        Log.i("LedTest", "Got LedManager object.");
     }
        /** Call methods in LedService via proxy object
          * which is provided by LedManager.
          */
        mLedManager.LedOn(1);
        TextView tv = new TextView(this);
        tv.setText("LED 1 is On.");
        setContentView(tv);
    }
}
```

五、实验中需要注意的问题
将下载后的源码放到你的 android 源码目录下，然后编译系统。本实验用的 android 版本为 2.1。实验的过程中大致出现过以下几个问题：

1、目标系统中没有生成 LedClient.apk 或 LedTest.apk 应用程序
编译完成后，没有在目标系统的 system/app/目录下找到 LedClient.apk 或 LedTest 应用程序。只有通过单独编译 LedClient 或 LedTest 才能在目标目录中生成。方法如下：

#mmm mokoid-read-only/apps/LedTest/

检查原因后发现 mokoid-read-only/apps/LedTest/Android.mk

LOCAL_MODULES_TAGS :=user

而我们的 s5pc100 系统在配置时 tapas 时选择的是 eng，所以没有装载到目标系统

所以修改 LedTest 和 LedClient 的 Android.mk

LOCAL_MODULES_TAGS :=user eng

再次编译即可自动装载到目标系统/system/app/目录下。

2、启动后没有图标，找不到应用程序
目标系统启动后找不到两个应用程序的图标。仔细阅读 logcat 输出的信息发现：

E/PackageManager(2717): Package com.mokoid.LedClient requires unavailable shared library com.mokoid.server; failing!

原因是找不到 com.mokoid.server。检查 mokoid-read-only/frameworks/base/Android.mk 发现系统将 LedManager 和 LedService 编译成 mokoid.jar 库文件。为了让应用程序可以访问到这个库，需要通过 com.mokoid.server.xml 来设定其对应关系。解决方法：拷贝 com.mokoid.server.xml 到目标系统的 system/etc/permissions/目录下

此时两个应用的程序的图标都正常出现了。

3、提示找不到 JNI_OnLoad
按照以前的实验加入下列代码：

```
static int registerMethods(JNIEnv* env) {
    static const char* const kClassName ="com/mokoid/server/LedService";
    jclass clazz;
        /* look up the class */
```

```
        clazz = env->FindClass(kClassName);
        if (clazz == NULL) {
            LOGE("Can't find class %s\n", kClassName);
            return -1;
        }
            /* register all the methods */
        if (env->RegisterNatives(clazz, gMethods,
                        sizeof(gMethods) / sizeof(gMethods[0])) != JNI_OK)
        {
            LOGE("Failed registering methods for %s\n", kClassName);
            return -1;
        }
    /* fill out the rest of the ID cache */
        return 0;
}
/*
 * This is called by the VM when the shared library is first loaded.
 */
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
        JNIEnv* env = NULL;
        jint result = -1;
        LOGI("JNI_OnLoad LED");
        if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
            LOGE("ERROR: GetEnv failed\n");
            goto fail;
        }
        assert(env != NULL);
        if (registerMethods(env) != 0) {
            LOGE("ERROR: PlatformLibrary native registration failed\n");
            goto fail;
        }
        /* success -- return valid version number */
        result = JNI_VERSION_1_4;
fail:
        return result;
}
```

4、需要针对你的目标平台修改 HAL 的 Makefile
修改 mokoid-read-only/hardware/modules/led/Android.mk

```
LOCAL_MODULE := led.default
```

5、在 eclipse 中编译不了 LedSystemServer.java
原因是程序中要用到 ServiceManager.addService，这需要系统权限。

解决方法可以把应用程序放入 Android 源码中编译，并确保以下两点：

(1) 在应用程序的 AndroidManifest.xml 中的 manifest 节点中加入 android:sharedUserId="android.uid.system"这个属性。

(2)修改 Android 加入 LOCAL_CERTIFICATE := platform.
当然：mokoid 工程源码中已经做了这些。

（3）经过 Manager 调用 service