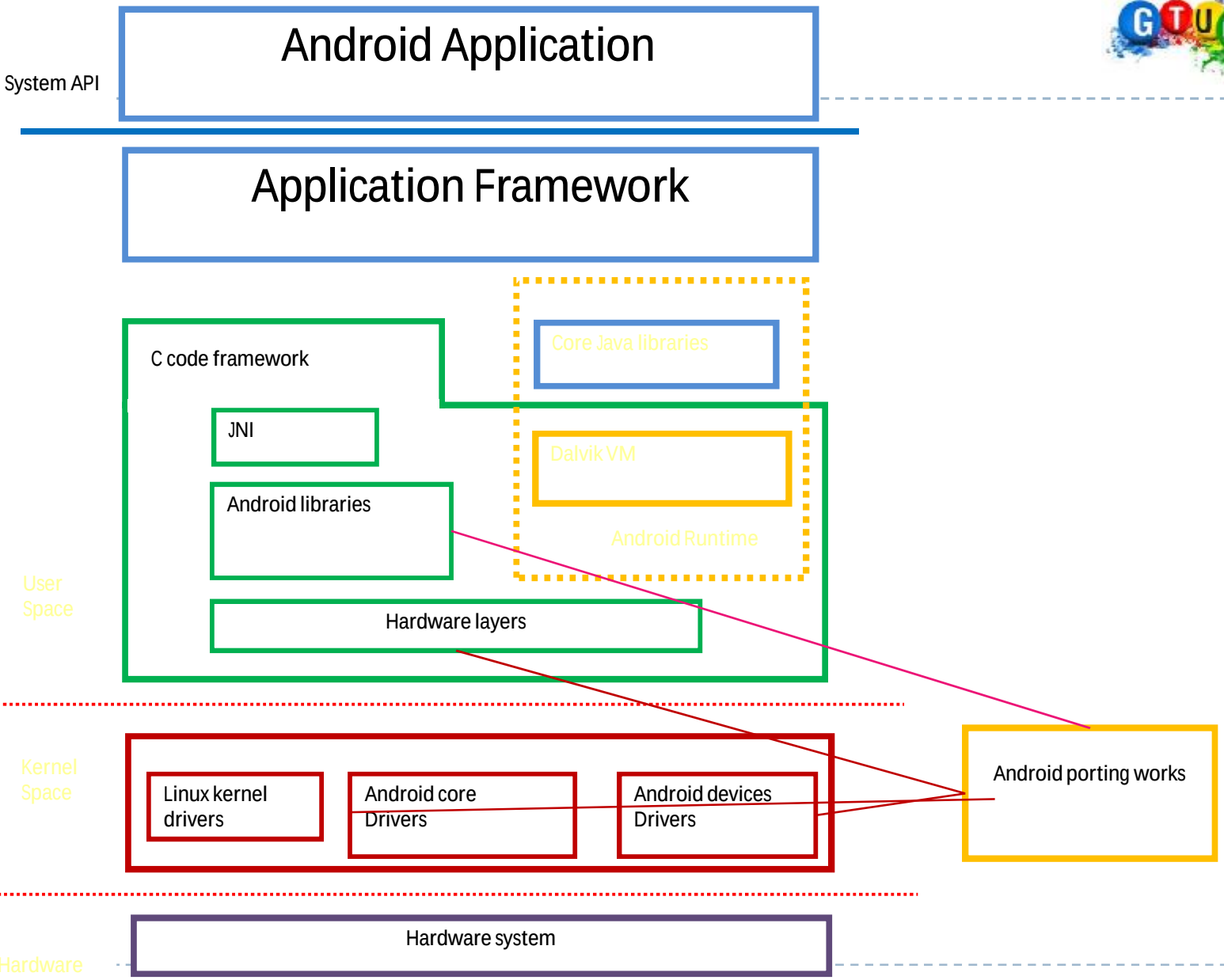

# Android技术之JNI和HAL

# 内容提纲


- jni的介绍和作用
- jni在android系统中的应用
- jni的编写和编译技术
- hal的介绍
- hal的编写技术


Android Application
System API
Application Framework
C code framework
Core Java libraries
JNI
Dalvik VM
Android libraries
Android Runtime
User Space
Hardware layers
Kernel Space
Linux kernel drivers
Android core Drivers
Android devices Drivers
Android porting works
Hardware system
Hardware system Space

# JNI的介绍


- 矛盾体：java应用程序如何与linux底层交互？

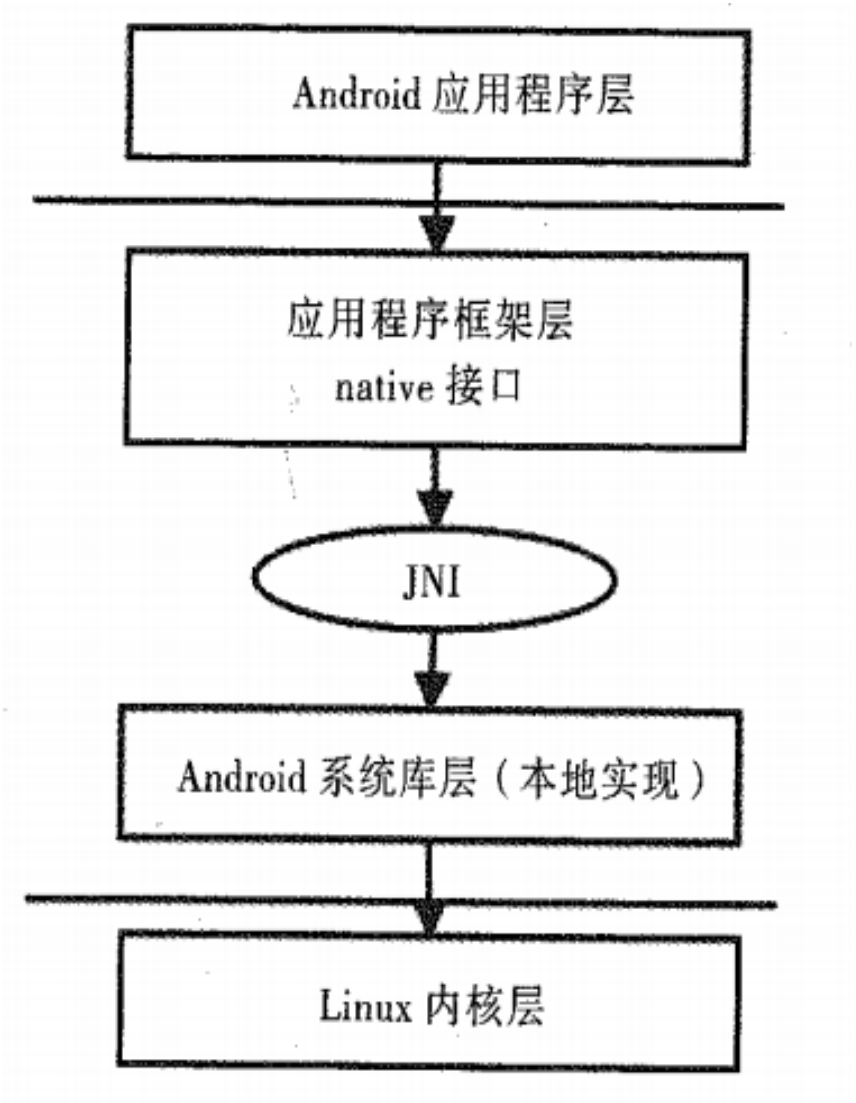
  android实现了很好的分层机制，从而使得开发者开发更加专注，但是android的应用层和框架层使用的开发语言为java，Java是一种平台无关性的语言，平台对于上层的java代码来说是透明的，而android是基于linux的一个操作系统，linux只提供了c/c++或者是汇编的接口给开发，因此，为了使得android系统运行正常，必须想办法把java语言与c/c++连接起来，这个连接器螺丝就是jni技术

# jni的概念

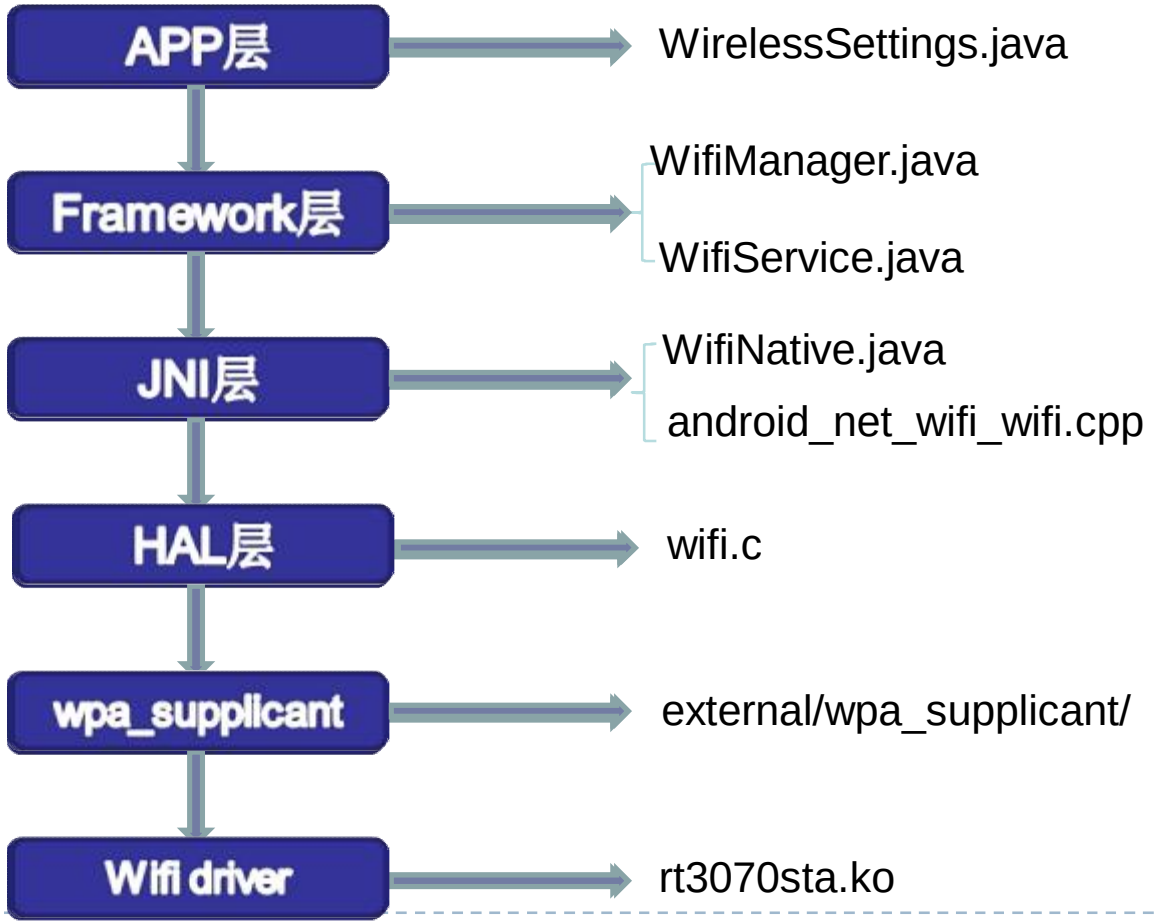

- Java 本机接口（Java Native Interface (JNI)）是一个本机编程接口，JNI允许Java代码使用以其它语言（譬如 C 和 C++）编写的代码和代码库

# jni在android系统中的应用

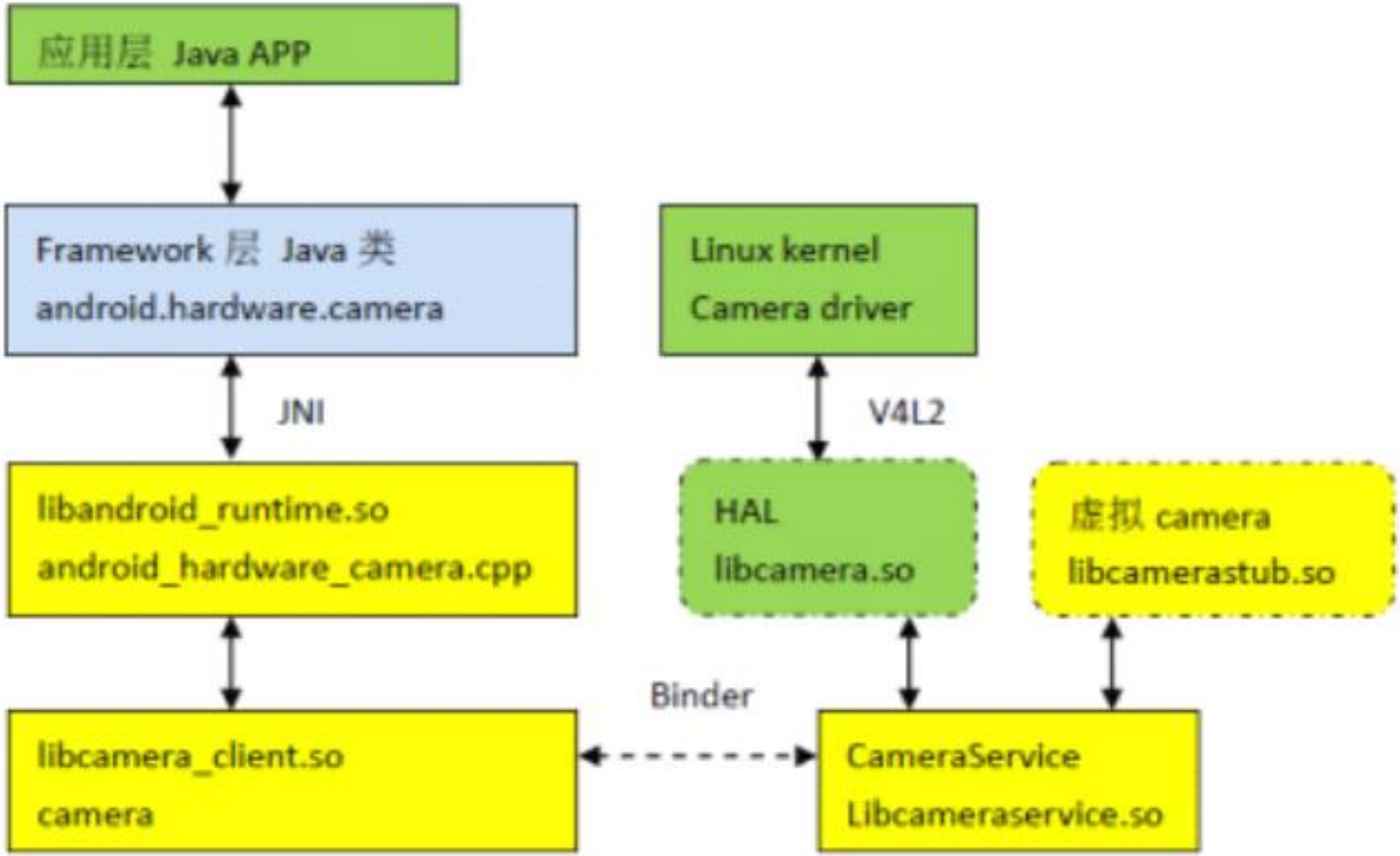
- jni在android系统中得到了大量的运用，比如wifi，3G，camera，g-sensor等等，只要是涉及到平台硬件，都需要用到jni



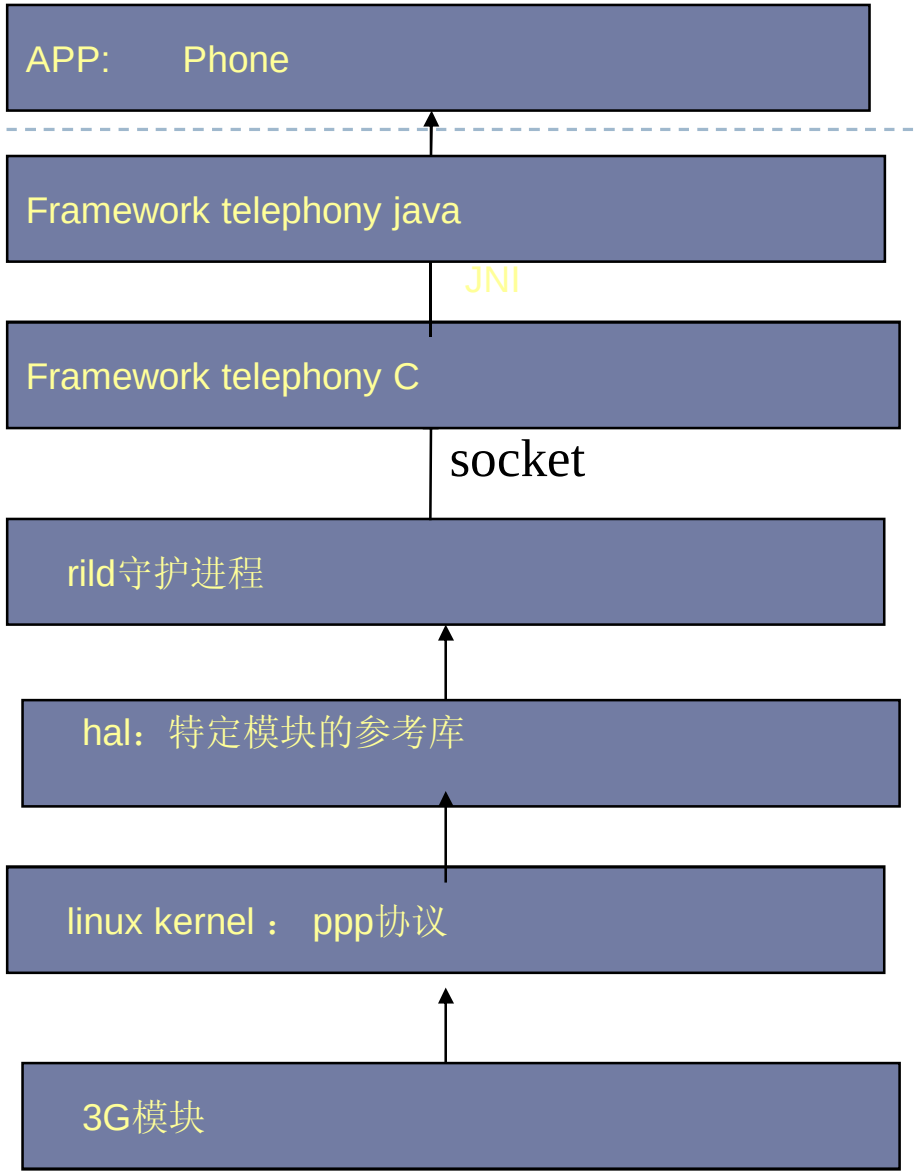
# wifi框架


APP层
WirelessSettings.java
Framework层
WifiManager.java
WifiService.java
JNI层
WifiNative.java
android_net_wifi_wifi.cpp
HAL层
wifi.c
wpa_supplicant
external/wpa_supplicant/
Wifi driver
rt3070sta.ko

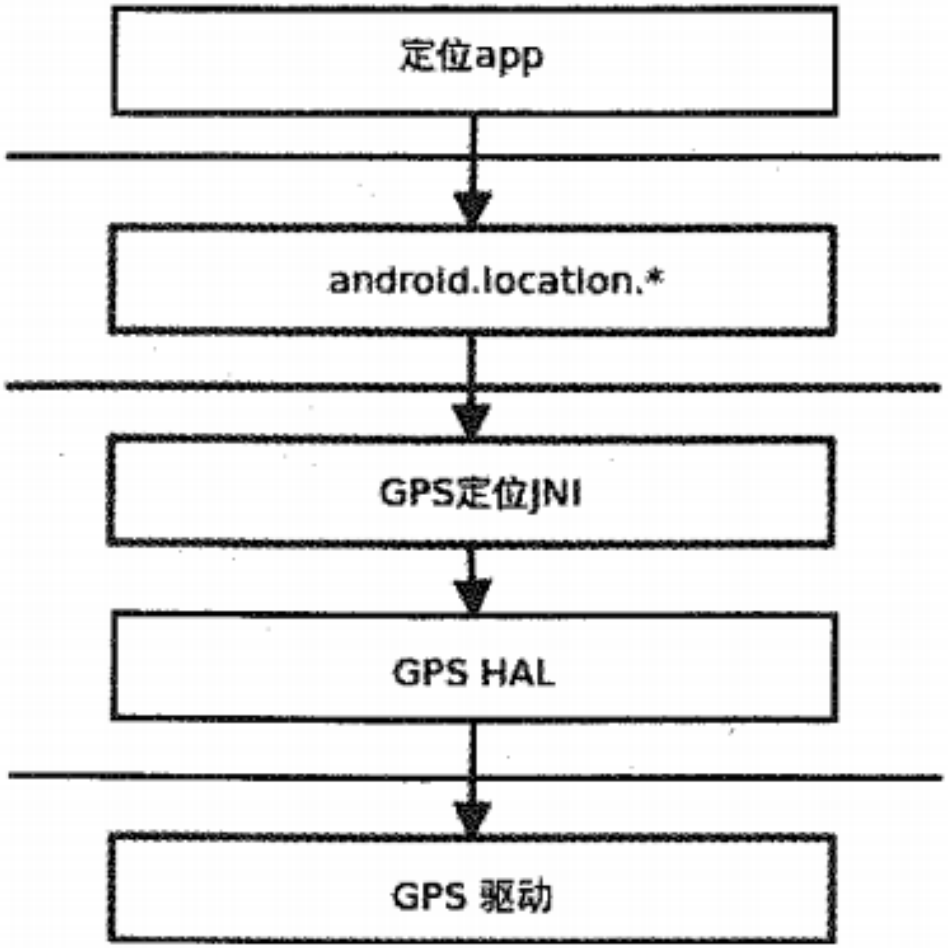
# 摄像头的框架


应用层 Java APP
Framework 层 Java 类
android.hardware.camera
JNI
libandroid_runtime.so
android_hardware_camera.cpp
libcamera_client.so
camera
Linux kernel
Camera driver
V4L2
HAL
libcamera.so
虚拟 camera
libcamerastub.so
Binder
CameraService
Libcameraservice.so

# 3G框架


APP: Phone
Framework telephony java
JNI
Framework telephony C
socket
rild守护进程
hal：特定模块的参考库
linux kernel ： ppp协议
3G模块

# GPS框架



# jni的编写


- Java调用c/c++的框架：

  Java代码 --> jni代码 --> c/c++代码

- 写一个简单的Java app 来调用 C/C++代码打印“HelloWorld！”的过程。这个过程由下面这几个步骤组成：

1.编写java代码，创建一个类（HelloWorld.java）声明native method。

2，编写jni文件

3，将jni实现文件编译成一个native lib

# 第一步：编写java代码，创建一个类（HelloWorld.java），并且使用**本地方法**。

```
public class HelloActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        this.fsSayHello();
    }
    static {
        System.loadLibrary("hello_jni"); //notice
    }
 private static native int fsSayHello(); //notice declare
}
```

- HelloActivity类声明一个本地 fsSayHello()方法。还有个static初始化函数
- native方法的声明：
  - 和一般java程序语言的声明中有不同之处，如fsSayHello()。一个native方法声明必须包含native修饰符

# 第二步：构建jni函数，并且编译成本地库


```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
        jint result = -1;
        JNIEnv* env = NULL;
        LOGI("JNI_OnLoad");
        if (vm->GetEnv((void **)&env, JNI_VERSION_1_4) != JNI_OK) {
            LOGE("ERROR: GetEnv failed");
            goto bail;
        }
        if (registerNatives(env) != JNI_TRUE) {
            LOGE("ERROR: registerNatives failed");
            goto bail;
        }
        result = JNI_VERSION_1_4;
    bail:
        return result;
}
```

- 当android的VM执行到程序里的System.loadLibrary(“xx.so”)函数的时候，首先会执行xx.so中的JNI_OnLoad()函数，所以我们需要在native lib中编写JNI_OnLoad()函数，
- JNI_OnLoad的写法其实不用自己记住，可以参考android自带的例子：./development/samples/SimpleJNI/jni/native.cpp

接下来就要实现registerNatives(env)

```
        static JNINativeMethod gMethods[] = {
                {"fsSayHello", "()I" , (void*)hello_printf},
        };
        static const char *classPathName = "com/farsight/HelloActivity ";
        static int registerNatives(JNIEnv* env){
            jclass clazz;
            clazz = env->FindClass(classPathName);
            if (clazz == NULL) {
                return JNI_FALSE;}
            if (env->RegisterNatives(clazz, gMethods, sizeof(gMethods) /
sizeof(gMethods[0])) < 0) {
                return JNI_FALSE;
            }
            return JNI_TRUE;
        }
```

- 几个注意点：
- classPathName:指的是声明fsSayHello()所在的类，
- env->RegisterNatives：向VM(即AndroidRuntime)登记gMethods[]表格所含的本地函数

env->RegisterNatives的几个参数：

clazz：告诉VM，clazz类中使用了gMethods中的本地方法

gMethods：定义java语言和本地语言(c/c++)的映射关系，"()Z"中，"()" 表示参数，Z则代表返回值为void。

sizeof(gMethods) / sizeof(gMethods[0])：gMethods数组元素个数

```
typedef struct {
    const char* name;
    const char* signature;
    void*       fnPtr;
} JNINativeMethod;
```

| Java 的类型 | JNI 的类型 | 对应的字母 | Java 的类型 | JNI 的类型 | 对应的字母 |
|---|---|---|---|---|---|
| boolean | jboolean | Z | long | jlong | J |
| byte | jbyte | B | float | jfloat | F |
| char | jchar | C | double | jdouble | D |
| short | jshort | S | object | jobject | L |
| int | jint | I | void | void | V |

# 第三步：编写c/c++代码

实现c语言的hello_printf()，这个就是linux工程师的强项了，学习c语言第一天干的就是这个事

```
static jint hello_printf(JNIEnv *env, jobject thiz)
{
        LOGI(">>>>jni test : hello world<<<<<\n");
        return 0;
}
```

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= \
        com.farsight.hello.cpp
LOCAL_SHARED_LIBRARIES := \
        libutils
LOCAL_C_INCLUDES += \
        $(JNI_H_INCLUDE)
LOCAL_MODULE:= libhello_jni
LOCAL_PRELINK_MODULE := false
include $(BUILD_SHARED_LIBRARY)
```

源码都已经写好了，需要一个编译规则，把c/cpp编译一个so，可以用Android..mk，可以参考development/samples/SimpleJNI/jni/Android.mk

# hal的介绍

- 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存在的原因


- 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HAL方式绕过GPL。
- 3、 针对某些硬件，Android有一些特殊的需求。

# HAL 简介及现状分析

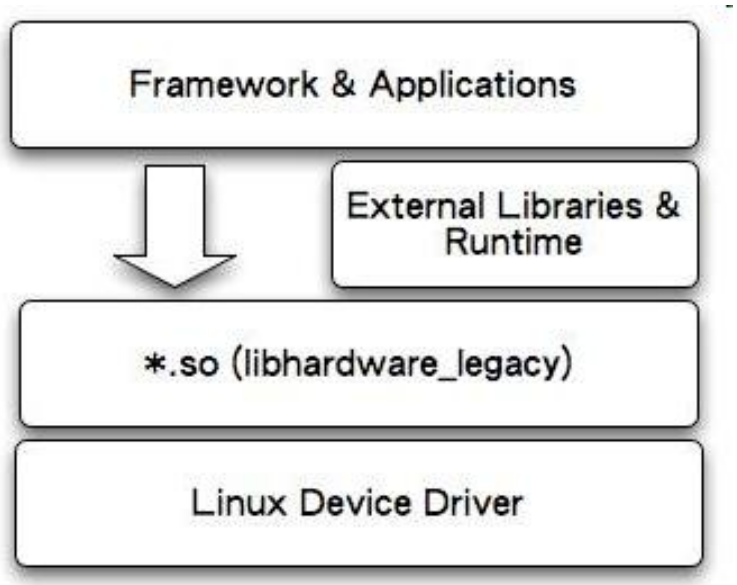

- 现有HAL架构由Patrick Brady (Google) 在2008 Google I/O演讲中提出，HAL 的目的是为了把 Android framework 与 Linux kernel 完整「隔开」。让 Android 不至过度依赖 Linux kernel，有点「kernel independent」的意思。
- 因为各厂商需要开发不开源代码的驱动程序模组要求下所规划的架构与概念要求下所规划的架构与观念，但是目前的HAL架构抽象程度还不足需要变动框架来整合HAL模组

# HAL内容


- HAL 主要的实作储存于以下目录：
  1. libhardware_legacy/ - 旧的架构、采取链接库模块的观念进行
  2. libhardware/ - 新架构、调整为 HAL stub 的观念
  3. ril/ - Radio Interface Layer
  4.msm7k  QUAL平台相关
- 主要包含一下一些模块：
  - Gps
  - Vibrator
  - Wifi
  - Uevent
  - Copybit
  - Audio
  - Camera
  - Lights
  - Ril
  - Overlay
  - ……

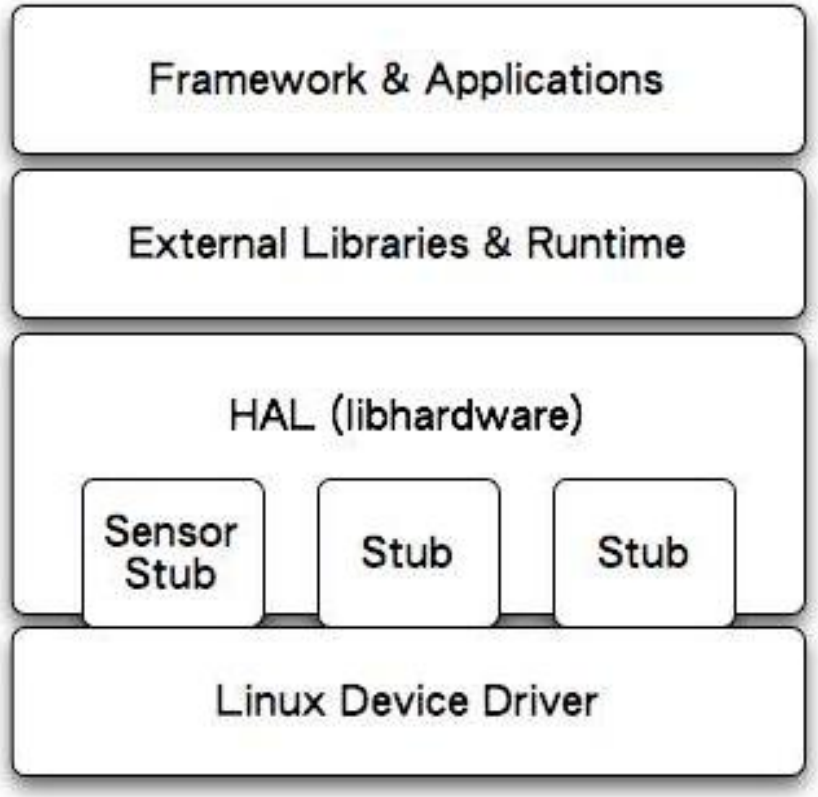
# 旧的HAL 架构(libhardware_legacy)

- libhardware_legacy 作法，是传统的「module」方式，也就是将 *.so 文件当做「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 檔（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。



# 新的HAL 架构(libhardware)

- 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。



# 目前HAL的解决方案

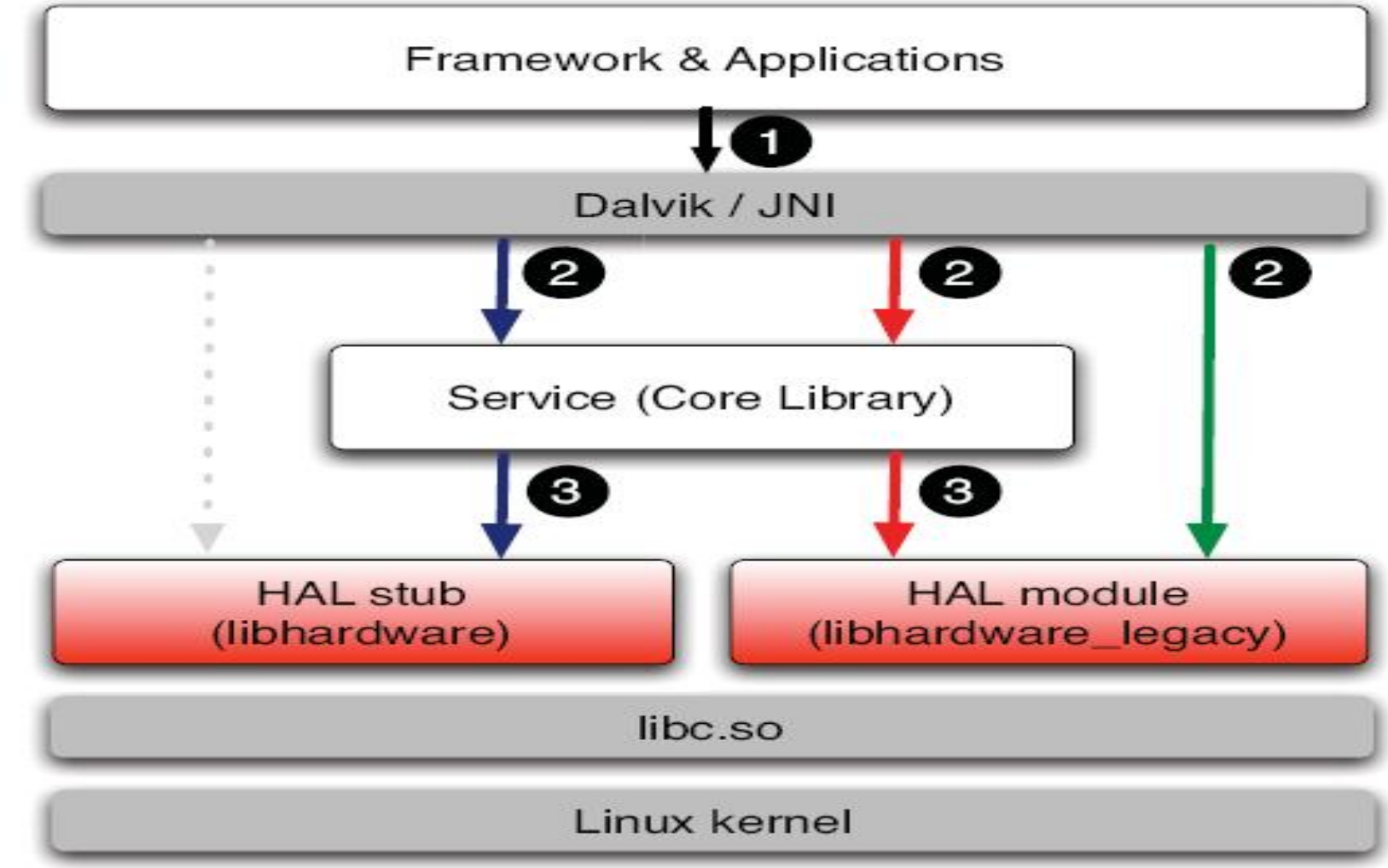

- 旧的(legeacy)HAL
  - 在hardware目录的libhardware_legacy
  - 我们称为HAL module
- 新的HAL
  - 在hardware的libhardware下
  - 称之为HAL stub

# 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。

# Android HAL 架构



# HAL module架构

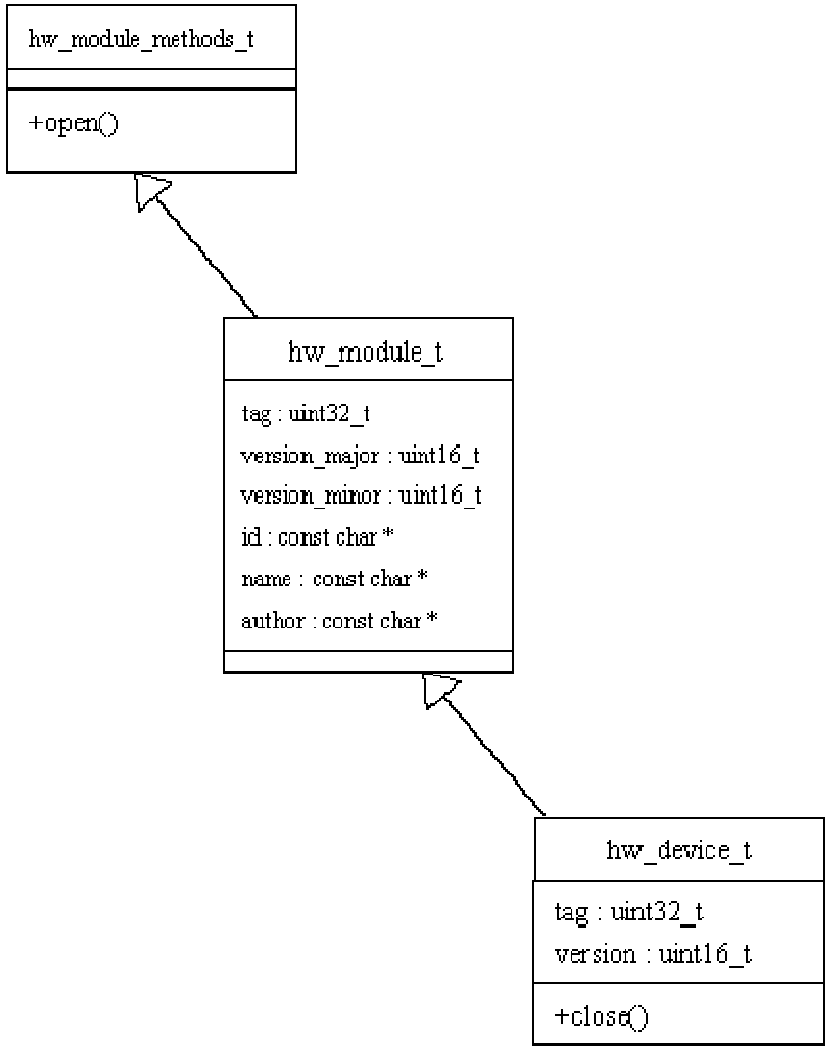
- HAL moudle主要分为三个结构：

  struct hw_module_t;

  struct hw_module_methods_t;

  struct hw_device_t;

- 定义在hardware.h文件里面
- 他们的继承关系如右图：



# hal的编写技术


- 实验框架：
  - App --> jni --> hal -->操作硬件
- 步骤的简单介绍：
  - 1，在jni实验的基础上，在app（HelloActivity.java）中加入操作硬件的本地函数。
  - 2，在jni文件中，加入hal的接口，从而可以调用hal的代码，与hal关联起来
  - 3，按照hal规范，构建hal 模块文件
  - 4，编写操作操作某个硬件的代码，我们只做简单的，实现打开/dev/led

# 第一步：修改HelloActivity.java中加入操作硬件的本地函数


```
public class HelloActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        this.fsSayHello();
        this.fsLedInit();  //新增
    }
    static {
        System.loadLibrary("hello_jni"); //notice
    }
    private static native int fsSayHello(); //notice declare
    private static native int fsLedInit(); //notice declare //新增
}
```

# 第二步：修改jni文件，实现本地函数 fsLedInit();同时加入hal的接口

A,添加fsFbInit的c语言映射函数

```
static JNINativeMethod gMethods[] = {
   {"fsSayHello", "()I" , (void*)hello_printf},
   {"fsLedInit", "()I", (void*)fs_led_init},
};
```

# B，实现fs_led_init()，用来调用一个open函数，从而打开设备

```
static struct led_hal_control_device_t *sFsLedDevice = 0;
static struct led_hal_module_t* sFsLedModule=0;
    /* convenient api for open device*/
    static inline int fs_led_control_open(const struct
hw_module_t* module,
            struct led_hal_control_device_t** device) {
            LOGI("fs_led_control_open");
            return module->methods->open(module,
                    LED_HAL_HARDWARE_MODULE_ID, (struct
hw_device_t**)device);
    }
```

```
static jint fs_led_init(JNIEnv *env, jobject thiz)
{
struct led_hal_module_t const *module;
if (hw_get_module(LED_HAL_HARDWARE_MODULE_ID,
(const hw_module_t**)&module) == 0) {
sFsLedModule = (struct led_hal_module_t *)module;
fs_led_control_open(&sFsLedModule->common,
&sFsLedDevice);
}
        LOGI("fs_led_init success");
        return 0;
}
```

这里引进了几个hal的常用接口：

struct led_hal_module_t： hal层的代码都是通过模块的形式存在，而一个模块对应一个XX_module_t，这个结构体需要自己定义,并且初始化，淡定，定义很简单

hw_get_module(); 加载hal模块的api，不用自己实现，(module指向变量HAL_MODULE_INFO_SYM)

struct led_hal_control_device_t；每个模块中都会对硬件进行操作，而每个设备对应一个XX_control_device_t,需要自己定义，并实现，同样淡定

我们的目的就是要open一个设备，open()函数就是最终用来打开设备的函数

} 所以我们接下来就是要定义和实现以上几个接口

A, 声明struct ;ed_hal_module_t等几个结构体

```
} struct  led_hal_module_t {//自己定义
        struct hw_module_t common; //第一个成员必须是hw_module_t类型
   };
struct  led_hal_control_device_t{ // 用于控制
    struct hw_device_t common; //不好意思，第一个成员也必须是struct hw_device_t类型
   }
```

} #define LED_HAL_HARDWARE_MODULE_ID "led_hal_sample"

这个宏很重要，hal以so的形式存在，那么so的名字中就包含fs_hal_sample，所以不是瞎定义的

B,初始化struct  led_hal_module_t几个结构体：

1）构建HAL_MODULE_INFO_SYM变量

```
const struct led_hal_module_t HAL_MODULE_INFO_SYM = {
common: {
}                tag: HARDWARE_MODULE_TAG, //必须是这个值
}                version_major: 1, //
}                version_minor: 0, //没有太多讲究
}                id: LED_HAL_HARDWARE_MODULE_ID,
}               name: "fs HAL sample module",
}                author: "shenzhen farsight",
}                methods: &led_hal_module_methods,    }
}  };
```

每个模块中必须要有一个以HAL_MODULE_INFO_SYM为名字的结构体变量，至于这个变量的结构体　类型名由开发人员自己定，比如：fs_fb_module_t为自己定义，并且模块中有变量：

```
struct  fs_fb_module_t HAL_MODULE_INFO_SYM
    ;
```

构建fs_fb_module_methods

```
static struct hw_module_methods_t
        led_hal_module_methods = {
}               open: led_device_open
}       };
```

```
#define SAMPLE_DEVICE_NAME "/dev/led"
static int fd = -1;
static int led_device_close(struct hw_device_t* device){
if (device) {
}                   free(device);
}          }
}          close(fd);
}          return 0;
}  }
```

```
static int led_device_open(const struct hw_module_t* module,
    const char* name,struct hw_device_t** device){
    struct led_hal_control_device_t *hal_ctl_dev;
    hal_ctl_dev = (struct
    led_hal_control_device_t*)malloc(sizeof(*hal_ctl_dev));
    memset(hal_ctl_dev, 0, sizeof(*hal_ctl_dev));
    hal_ctl_dev->common.tag=HARDWARE_DEVICE_TAG;
    hal_ctl_dev->common.version = 0;
    hal_ctl_dev->common.module= module;
    hal_ctl_dev->common.close = led_device_close;
}
```

```
} if((fd = open(SAMPLE_DEVICE_NAME, O_RDWR))==-1){
}                               exit(1);
}           }else{
}                       LOGI("open ok\n");
}           }
}                       return 0;
}           }
```

linux应用中，要对某个硬件操作，必须用open()将设备打开，因此可以封装open()

- int (*led_device_open)(const struct hw_module_t* module, const char* id, struct hw_device_t** device);
- 介绍一下这三个参数：

struct hw_module_t* module：模块对象，可以理解为一个module代表一个so

const char* name ： 一般对设备操作有控制和传输数据，其实可以通过这个name来判断是来控制设备还是与设备进行数据传输

struct hw_device_t** device， 面向对象的思想，可以理解为一个真正设备的形象代言人，包含设备属性和功能

# 第四步：Android.mk的编写

- LOCAL_PATH := $(call my-dir)
- include $(CLEAR_VARS)
- LOCAL_PRELINK_MODULE := false
- LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
- LOCAL_SHARED_LIBRARIES := liblog
- LOCAL_SRC_FILES := led_hal_sample.c
- LOCAL_MODULE := led_hal_sample.default
- include $(BUILD_SHARED_LIBRARY)

- 为什么要命名 led_hal_sample.default呢，因为hw_get_module的时候会到/system/lib/hw去搜索相应的so文件，一般命名方式为 xx.ro.hardware.so, xx.default.so

# HAL Module 获取

- hardware.c
- int hw_get_module(const char *id, const struct hw_module_t **module)
- 1. 查找module patch

```
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);
    }
```

# HAL Module 获取 (cont.)


- 2. 载入 so

```
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);
}
```

- 3. Open so

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

# HAL Module 获取 (cont.)


- 4. 获取symbol

```
/* Get the address of the struct hal_module_info. */
    const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
    hmi = (struct hw_module_t *)dlsym(handle, sym);
    if (hmi == NULL) {
        LOGE("load: couldn't find symbol %s", sym);
        status = -EINVAL;
        goto done;
    }
```

# HAL Module 获取 (cont.)


- hardware.h

```
#define HAL_MODULE_INFO_SYM         HMI
/**
 * Name of the hal_module_info as a string
 */
#define HAL_MODULE_INFO_SYM_AS_STR  "HMI"
```