Android底层开发

华清远见： 刘洪涛

版权

- 华清远见嵌入式培训中心版权所有；
- 未经华清远见明确许可，不能为任何目的以任何形式复制或传播此文档的任何部分；
- 本文档包含的信息如有更改，恕不另行通知；
- 保留所有权利。

内容提纲

- Android开发技术分类
- Android 系统移植步骤
- Android 编译系统
- HAL层开发方法
- HAL开发实例解析

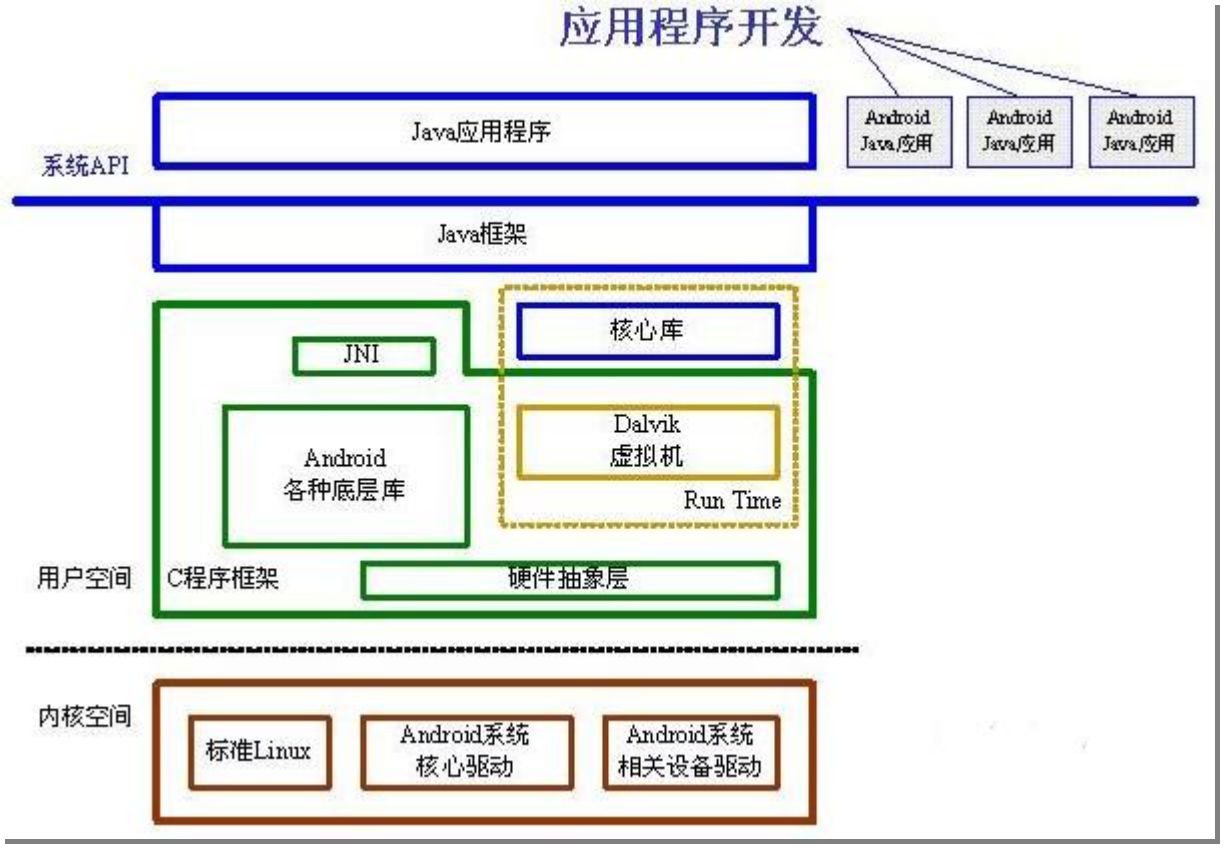
Google Android 软件架构

- Android系统架构和其操作系统一样，采用了分层的架构。从架构图看，Android系统架构分为四个层，从高层到低层分别为
  - 应用程序层、
  - 应用程序框架层、
  - 系统运行库层
  - linux核心层。

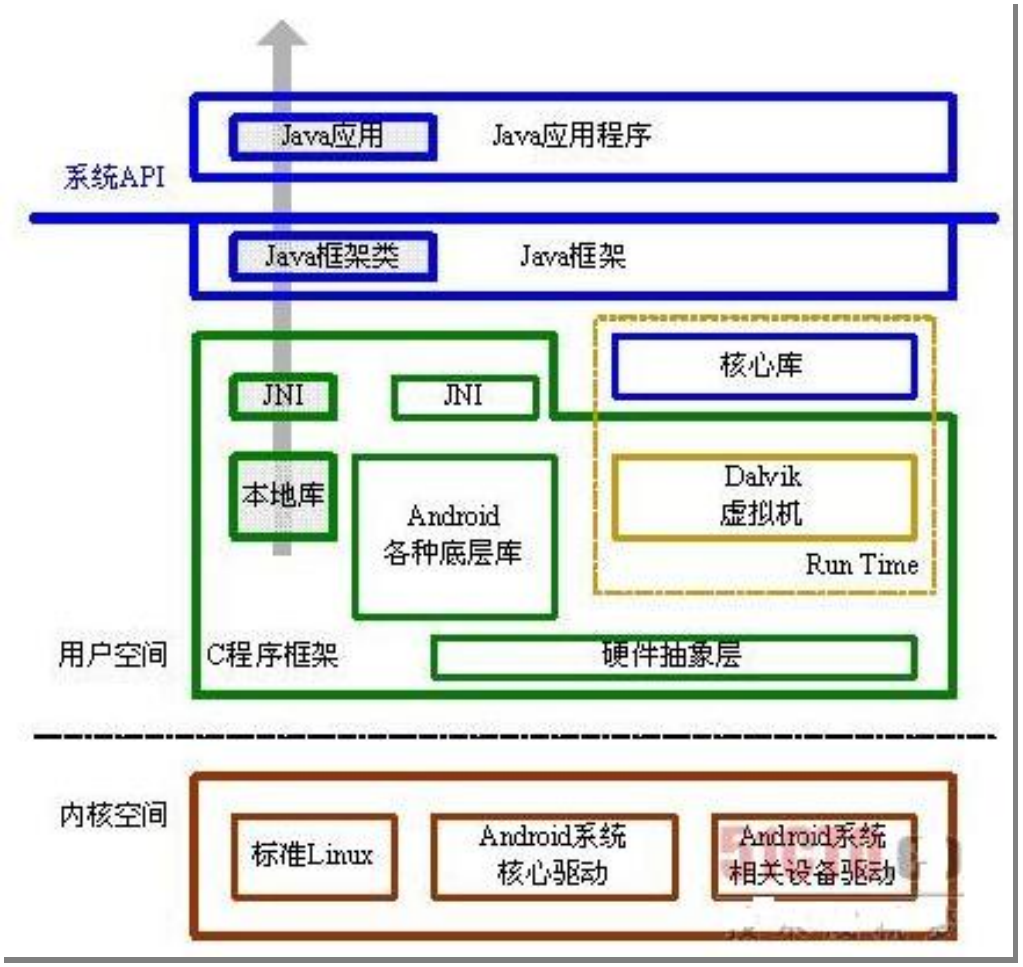
Android技术开发分类

- Android应用程序开发
- Android系统开发
- Android移植开发（硬件相关）

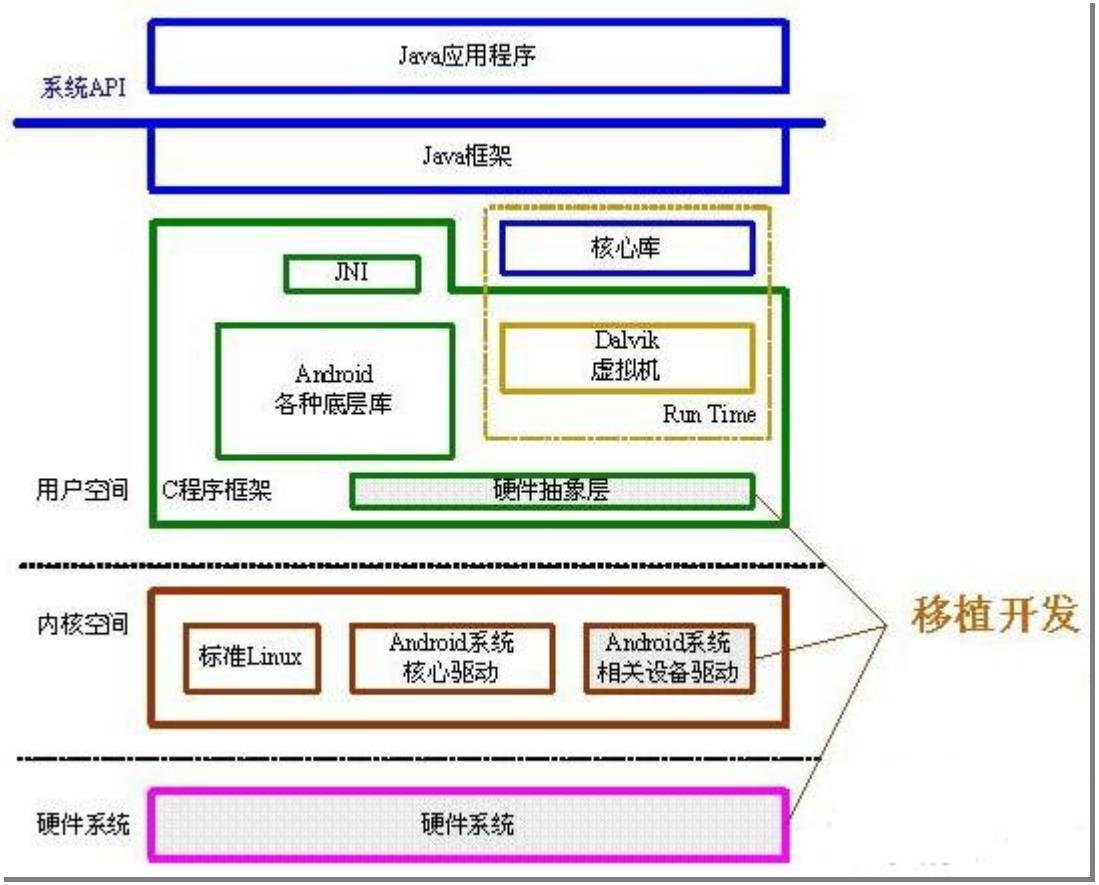
Android应用程序开发



Android系统开发



Android移植开发



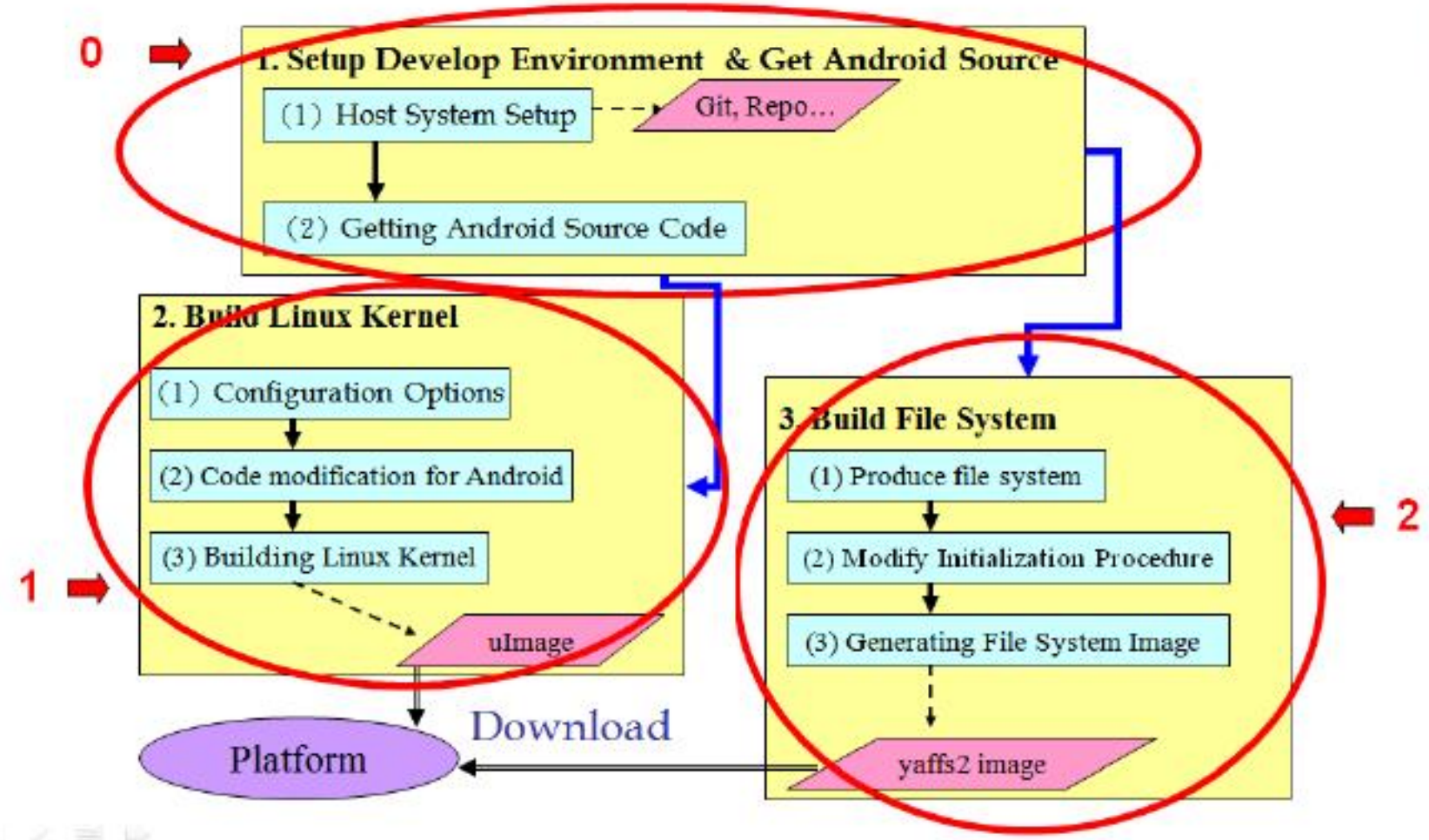
开发方式

- 基于SDK开发
  - 使用Google提供的SDK。Android的SDK中包含Android系统下层的二进制映像、模拟器及相关的工具，在Linux和Windows系统中使用Eclipse IDE环境进行Android应用程序的开发。
  - 不需要使用硬件，不需要涉及Android系统的底层，只需要了解Android系统的API。
- 基于源码
  - 可以开发Android应用程序，进行系统移植或者开发Android系统本身。

开发方式（cont.）

- 基于源代码的Android开发，所涉及的知识体系包含以下一些内容
  - Linux操作系统的基础知识；
  - Linux内核知识（C语言）；
  - Linux驱动程序知识（C语言）；
  - 处理器（ARM）技术
  - Android底层库（C语言、C++）；
  - 音频、视频和多媒体（C语言、C++、Java）；
  - 电话部分（C语言、C++、Java）；
  - 蓝牙、Wifi、定位系统（C语言、C++、Java）；
  - 传感器系统（C语言、C++、Java）。

Android 系统移植步骤



Android 硬件系统要求

Feature	Minimum Requirement	Notes
Chipset	ARM-based	For the first release, Android is primarily targeted towards mobile handsets and portions of the platform, such as Dalvik VM graphics processing, currently assume an ARM architecture.
Memory	128 MB RAM; 256 MB Flash External	Android can boot and run in configurations with less memory, but it isn't recommended.
Storage	Mini or Micro SD	Not necessary for basic bring up, but recommended.
Primary Display	QVGA TFT LCD or larger, 16-bit color or better	The current Android interface targets a touch-based HVGA resolution display with a touch-interface no smaller than 2.8 inches in size. However, smaller displays will suffice for initial porting.
Navigation Keys	5-way navigation with 5 application keys, power, camera and volume controls	

Android 硬件系统要求 (cont.)

Camera	2MP CMOS	Not required for basic bring up.
USB	Standard mini-B USB interface	Android uses the USB interface for flashing the device system images and debugging a running device.
Bluetooth	1.2 or 2.0	Not required for initial bring up.

Android Linux kernel 特性

- 从Linux 2.6.33版本开始，Google智能手机操作系统Android核心代码将被删除。
- 目前Android 2.2 froyo使用的Linux kernel版本为2.6.32
- 在Linux 2.6.32版本上Android添加了如下几个模块
- 在kernel/drivers/misc目录
  - binder.c{ binder.h}
  - logger.c {logger.h}
  - lowmemorykiller.c
  - ram_console.c
  - timed_gpio.c {timed_gpio.h}
  - timed_output.c {timed_output.h}
  - Kernel_debugger.c {include\linux\Kernel_debugger.h}
  - Uid_stat.c {include\linux\Uid_stat.h}
  - Pmem.c {linux/android_pmem.h}

Android Linux kernel 特性(cont.)

- drivers/rtc/alarm.c
- kernel/power 目录
  - wakelock.c {include\linux\Wakelock.h}
  - userwakelock.c{include\linux\Wakelock.h}
  - earlysuspend.c
  - consoleearlysuspend.c
  - fbearlysuspend.c
- mm/ashmem.c {linux/ashmem.h}
- drivers/input/misc 目录
  - Gpio_axis.c
  - Gpio_event.c {Gpio_event.h}
  - Gpio_input.c
  - Gpio_matrix.c
  - Gpio_output.c
  - Keychord.c {Keychord.h}

Android Linux kernel 特性(cont.)

- drivers/input/keyreset.c {keyreset.h}
- drivers/switch 目录
  - switch_class.c
  - switch_gpio.c {switch.h}
- drivers/usb/gadge 目录
  - Android.c {include/linux/usb/android.h}
  - Android_aid.h
  - f_adb.c f_adb.h
  - f_mass_storage.c
  - f_mass_storage.h
- drivers\net 目录
  - Pppolac.c
  - Pppopns.c
  - Ppp_mppe.c

Android 特有驱动

- Ashmem
  - 匿名共享内存驱动
- Logger
  - 轻量级的log驱动
- Binder驱动(Binder driber)
  - 基于OpenBinder驱动，为Android平台提供IPC的支持
- 电源管理(Android Power Management)
  - 轻量级的的电源管理，为嵌入式系统做了优化，包括earlysuspend等驱动
- Low Memory Killer
  - 在缺少内存的情况下，杀死进程
- Android PMEM
  - 物理内存驱动

Android 内核移植过程

- 下载kernel:

git clone git://android.git.kernel.org/projects/kernel/common.git/summary

- 修改kernel config

```
#
# Android
#
# CONFIG_ANDROID_GADGET is not set
# CONFIG_ANDROID_RAM_CONSOLE is not set
CONFIG_ANDROID_POWER=y
CONFIG_ANDROID_POWER_STAT=y
CONFIG_ANDROID_LOGGER=y
# CONFIG_ANDROID_TIMED_GPIO is not set
CONFIG_ANDROID_BINDER_IPC=y
```

Android内核移植过程(CONT.)

- 移植 Drivers
  - Connectivity
    - Bluetooth
    - GPS
    - Wi-Fi
  - Display Drivers
  - Input Devices
    - Keymaps and Keyboard
  - Lights
  - Multimedia
    - Audio
    - Camera/Video
  - Power Management
  - Sensors
  - Telephony
    - Radio Interface Layer
    - SIM Toolkit Application (STK)

获得 Android 源代码

- The source is approximately 2.1GB in size. You will need 6GB free to complete the build.
- Installing Repo
- $ curl http://android.git.kernel.org/repo > ~/bin/repo
- $ chmod a+x ~/bin/repo
- $ mkdir working-directory-name
- $ cd working-directory-name
- $ repo init -u git://android.git.kernel.org/platform/manifest.git -b froyo

Android 编译系统

- 几个重要的makefile：

  Android.mk

  AndroidProducts.mk

  target_<os>-.mk, host_-.mk and -.mk

  BoardConfig.mk

  buildspec.mk

Android 编译系统（cont.）

Android.mk 是module和package的makefile，每个module和package目录下都会有这么一个文件。

AndroidProducts.mk为设定product配置，一个product表示一个特定的版本，Android通过编译不同的product来产生不同的软件配置内容。

BoardConfig.mk是为product主板做设定，设置一些Board的参数如kernel的传递参数等。

*-.mk是针对选择的操作系统和CPU架构，进行相关设定。

buidspec.mk是位于source跟目录下为编译之外的设定，如可以选择要产生的product，平台，额外的module/package等。

添加本地程序和库的方法

- Android中增加本地的程序或者库，这些程序和库与它们所在的路径没有关系，只和它们的Makefile：Android.mk文件有关系。
- Android.mk具有统一的写法，主要包含了一些系统公共的宏。
- 选项可以参考以下文件：
  - build/core/config.mk
- 默认的值在下文件中定义：
  - build/core/base_rules.mk
- 在一个Android.mk中也可以生成多个可执行程序、动态库或者静态库。

添加本地程序和库的方法(cont.)

- 编译可执行程序的Android.mk 示例

```
# Test Exe

LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
  main.c
LOCAL_MODULE:= test_exe

#LOCAL_C_INCLUDES :=
#LOCAL_STATIC_LIBRARIES :=
#LOCAL_SHARED_LIBRARIES :=

include $(BUILD_EXECUTABLE)
```

添加本地程序和库的方法(cont.)

- 编译静态库的Android.mk

```
# Test Static lib

LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
  helloworld.c

LOCAL_MODULE:= libtest_static

#LOCAL_C_INCLUDES :=
#LOCAL_STATIC_LIBRARIES :=
#LOCAL_SHARED_LIBRARIES :=

include $(BUILD_STATIC_LIBRARY)
```

添加本地程序和库的方法(cont.)

- 编译动态库的Android.mk

```
# Test shared lib

LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
  helloworld.c

LOCAL_MODULE:= libtest_shared

TARGET_PRELINK_MODULE := false

#LOCAL_C_INCLUDES :=
#LOCAL_STATIC_LIBRARIES :=
#LOCAL_SHARED_LIBRARIES :=

include $(BUILD_SHARED_LIBRARY)
```

添加本地程序和库的方法(cont.)

- 可执行程序、动态库和静态库生成的文件在分别在：
  - out/target/product/*/obj/EXECUTABLE
  - out/target/product/*/obj/STATIC_LIBRARY
  - out/target/product/*/obj/SHARED_LIBRARY
- 其目标的文件夹分别为：
  - XXX_intermediates
  - XXX_shared_intermediates
  - XXX_static_intermediates
- XXX为每个模块中LOCAL_MODULE所定义的名字。

添加本地程序和库的方法(cont.)

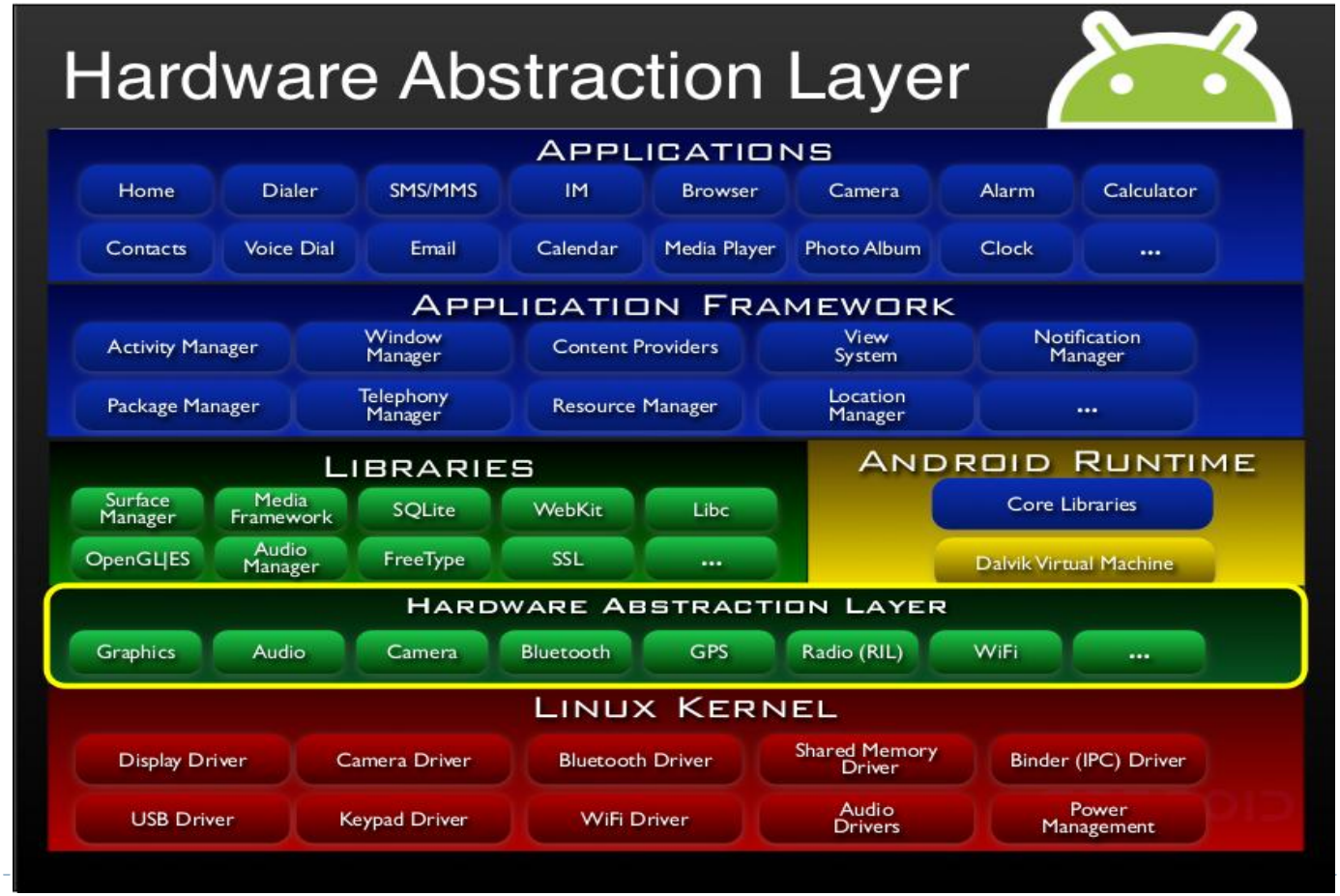
- 如果编译主机的：可执行程序，动态库，静态库
  - include $(BUILD_HOST_EXECUTABLE)
  - include $(BUILD_HOST_SHARED_LIBRARY)
  - include $(BUILD_HOST_STATIC_LIBRARY)

添加本地程序和库的方法(cont.)

- 在Android文件中，可以指定最后的目标安装路径，使用下面两个宏指定：
  - LOCAL_MODULE_PATH和LOCAL_UNSTRIPPED_PATH
  - 分别代表最终模块的目标路径和没有经过符号剥离的目标路径
- 增加以下可以安装到不同的文件系统：
  - LOCAL_MODULE_PATH := $(TARGET_ROOT_OUT)
  - LOCAL_UNSTRIPPED_PATH := $(TARGET_ROOT_OUT_UNSTRIPPED)
- 不同的文件系统使用下面宏进行选择：
  - TARGET_ROOT_OUT：
  - 表示根文件系统out/target/product/generic/root
  - TARGET_OUT：
  - 表示system文件系统out/target/product/generic/system
  - TARGET_OUT_DATA：
  - 表示data文件系统out/target/product/generic/data

HAL 概念

- Android 的 HAL（Hardware Abstract Layer硬件抽象层）是Google因应厂商「希望不公开源码」的要求下，所推出的新观念，其架构如下图。



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

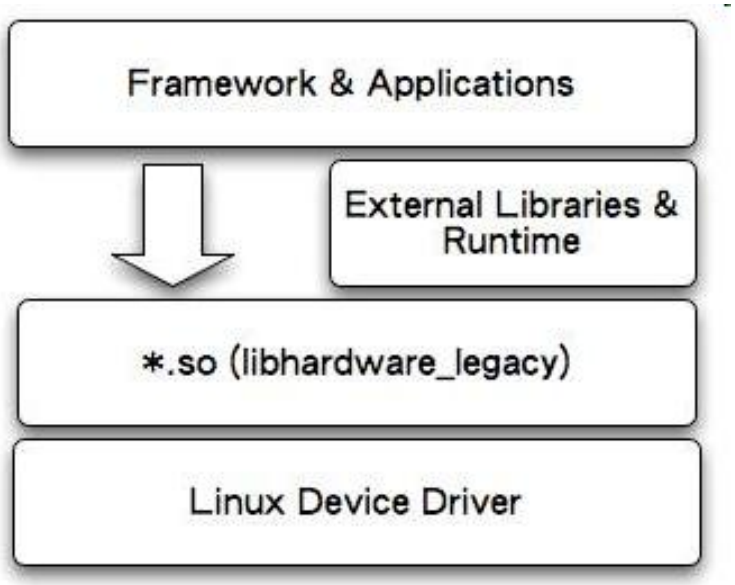
- 现有HAL架构由Patrick Brady (Google) 在2008 Google I/O演讲中提出，从上面架构图我们知道，HAL 的目的是为了把 Android framework 与 Linux kernel 完整「隔开」。让 Android 不至过度依赖 Linux kernel，有点「kernel independent」的意思。
- 因为各厂商需要开发不开源代码的驱动程序模组要求下所规划的架构与概念要求下所規劃的架構與觀念
- 但是目前的HAL架构抽象程度还不足
- 需要变动框架来整合HAL模组

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
  - ……

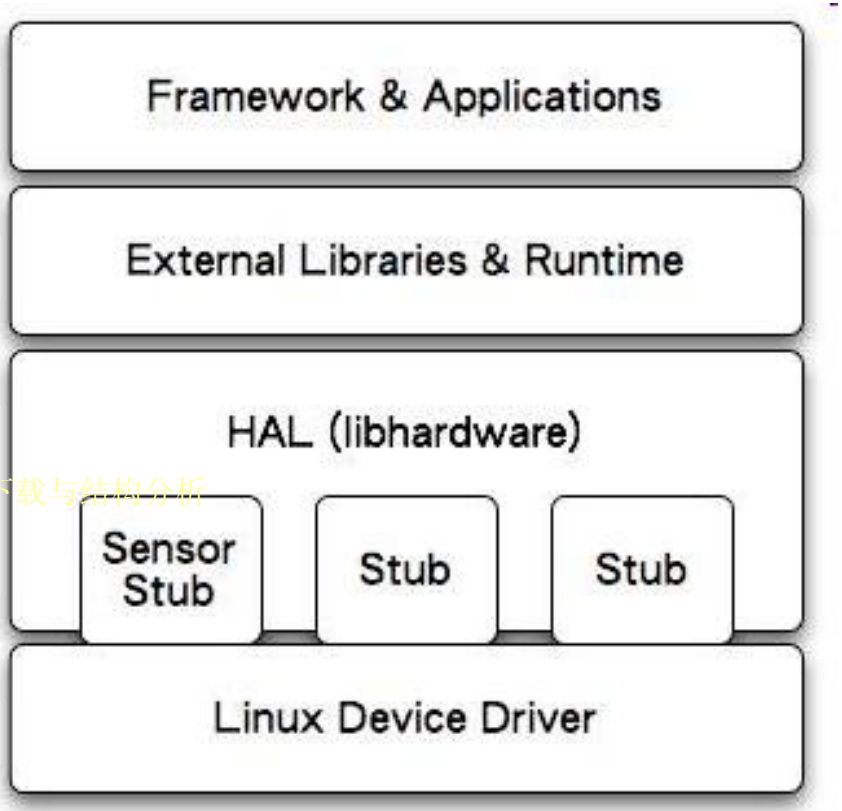
旧的HAL 架构(libhardware_legacy)

- 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。



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

HAL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);
```

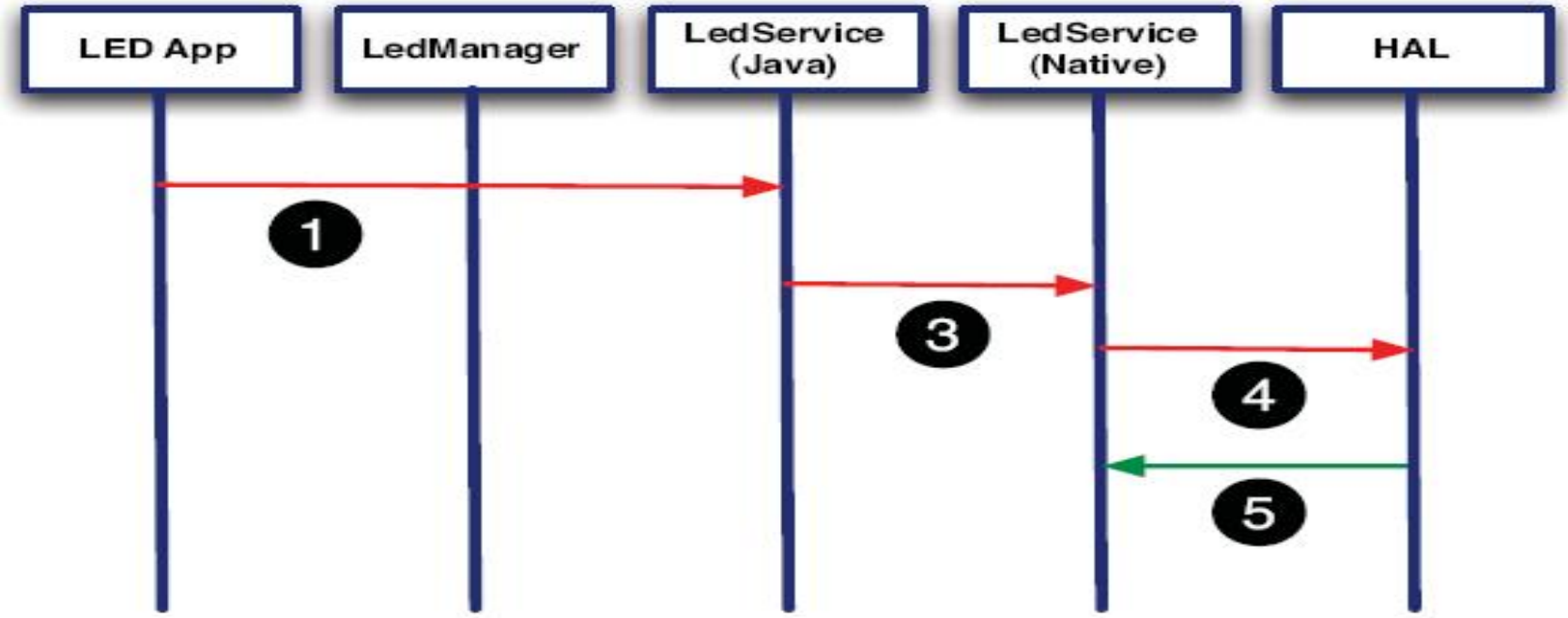
mokoid 工程代码下载与结构分析

- **mokid项目概述**
- modkoid工程提供了一个LedTest示例程序，对于理解android层次结构、Hal编程方法都非常有意义。
- **下载方法**

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

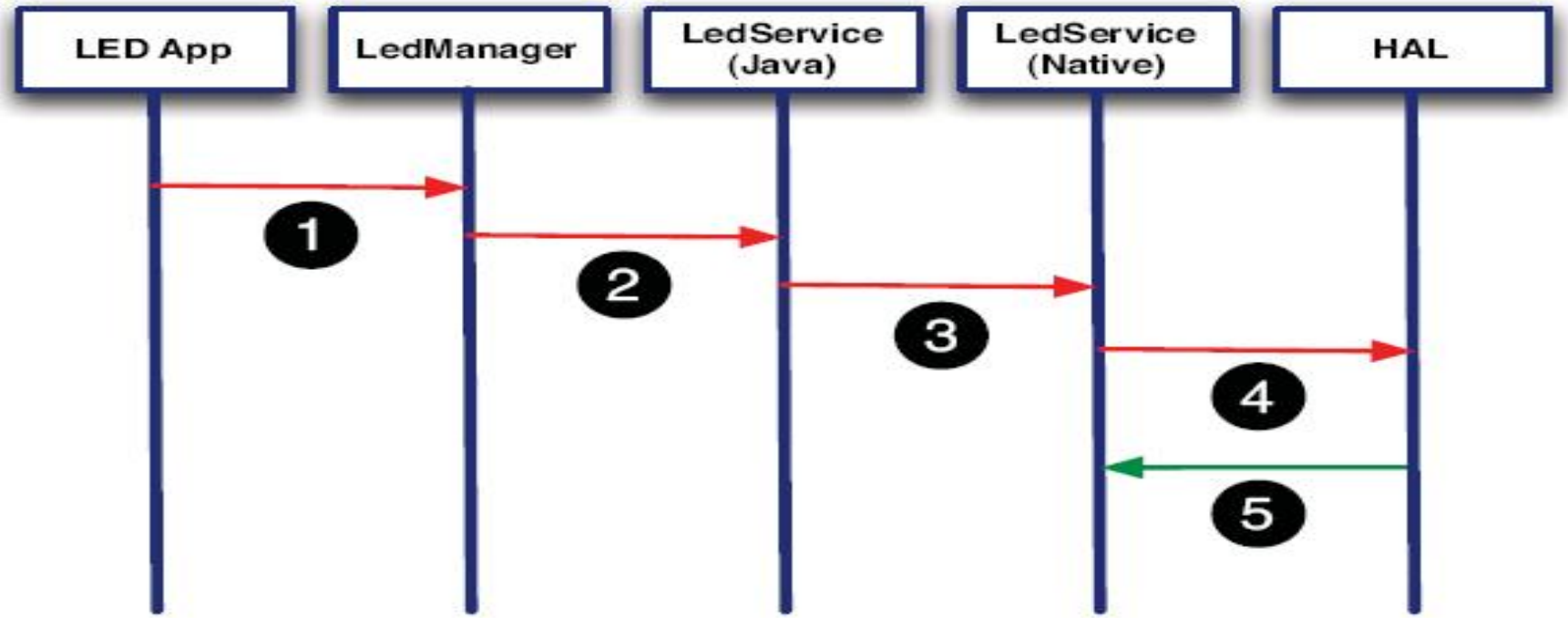
两种实现方式

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



两种实现方式（cont.）

- （2）经过Manager调用service



功能实现和HAL stub注册

- 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. 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. */
};
```

功能实现和HAL stub注册（cont.）

- 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。

```
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
    ……
}
```

功能实现和HAL stub注册(cont.)

- 填充具体API操作代码

```
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);
    return 0;
}
```

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"
```

详细的代码讲解及实验过程

- 参见：
  - http://blog.csdn.net/hongtao_liu/archive/2010/12/07/6060734.aspx#FeedBack

谢谢！

刘洪涛
lht@farsight.com.n