在上两篇文章中，我们介绍了如何为 Android 系统的硬件编写驱动程序，包括如何在 Linux 内核空间实现内核驱动程序和在用户空间实现硬件抽象层接口。实现这两者的目的是为了向更上一层提供硬件访问接口，即为 Android 的 Application Frameworks 层提供硬件服务。我们知道，Android 系统的应用程序是用 Java 语言编写的，而硬件驱动程序是用 C 语言来实现的，那么，Java 接口如何去访问 C 接口呢？众所周知，Java 提供了 JNI 方法调用，同样，在 Android 系统中，Java 应用程序通过 JNI 来调用硬件抽象层接口。在这一篇文章中，我们将介绍如何为 Android 硬件抽象层接口编写 JNI 方法，以便使得上层的 Java 应用程序能够使用下层提供的硬件服务。

一. 参照在 Ubuntu 上为 Android 增加硬件抽象层（HAL）模块访问 Linux 内核驱动程序一文，准备好硬件抽象层模块，确保 Android 系统镜像文件 system.img 已经包含 hello.default 模块。

二. 进入到 frameworks/base/services/jni 目录，新建 com_android_server_HelloService.cpp 文件：

USER-NAME@MACHINE-NAME:~/Android$ cd frameworks/base/services/jni

USER-NAME@MACHINE-NAME:~/Android/frameworks/base/services/jni$ vi com_android_server_HelloService.cpp

在 com_android_server_HelloService.cpp 文件中，实现 JNI 方法。注意文件的命令方法，com_android_server 前缀表示的是包名，表示硬件服务 HelloService 是放在 frameworks/base/services/java 目录下的 com/android/server 目录的，即存在一个命令为 com.android.server.HelloService 的类。这里，我们暂时略去 HelloService 类的描述，在下一篇文章中，我们将回到 HelloService 类来。简单地说，HelloService 是一个提供 Java 接口的硬件访问服务类。

首先是包含相应的头文件：

view plain

```
1.  #define LOG_TAG "HelloService"
2.  #include "jni.h"
3.  #include "JNIHelp.h"
4.  #include "android_runtime/AndroidRuntime.h"
5.  #include <utils/misc.h>
```

```
#include <utils/Log.h>
#include <hardware/hardware.h>
#include <hardware/hello.h>
#include <stdio.h>
```

接着定义 hello_init、hello_getVal 和 hello_setVal 三个 JNI 方法：

view plain

```
namespace android
{
    /*在硬件抽象层中定义的硬件访问结构体，参考<hardware/hello.h>*/
        struct hello_device_t* hello_device = NULL;
    /*通过硬件抽象层定义的硬件访问接口设置硬件寄存器 val 的值*/
        static void hello_setVal(JNIEnv* env, jobject clazz, jint value) {
        int val = value;
        LOGI("Hello JNI: set value %d to device.", val);
        if(!hello_device) {
            LOGI("Hello JNI: device is not open.");
            return;
        }

        hello_device->set_val(hello_device, val);
    }
        /*通过硬件抽象层定义的硬件访问接口读取硬件寄存器 val 的值*/
    static jint hello_getVal(JNIEnv* env, jobject clazz) {
        int val = 0;
        if(!hello_device) {
            LOGI("Hello JNI: device is not open.");
            return val;
        }
        hello_device->get_val(hello_device, &val);

        LOGI("Hello JNI: get value %d from device.", val);

        return val;
    }
        /*通过硬件抽象层定义的硬件模块打开接口打开硬件设备*/
    static inline int hello_device_open(const hw_module_t* module, struct he
llo_device_t** device) {
        return module->methods->open(module, HELLO_HARDWARE_MODULE_ID, (stru
ct hw_device_t**)device);
    }
        /*通过硬件模块 ID 来加载指定的硬件抽象层模块并打开硬件*/
```

```
34.     static jboolean hello_init(JNIEnv* env, jclass clazz) {
35.         hello_module_t* module;
36.
37.         LOGI("Hello JNI: initializing......");
38.         if(hw_get_module(HELLO_HARDWARE_MODULE_ID, (const struct hw_module_t
    **)&module) == 0) {
39.             LOGI("Hello JNI: hello Stub found.");
40.             if(hello_device_open(&(module->common), &hello_device) == 0) {
41.                 LOGI("Hello JNI: hello device is open.");
42.                 return 0;
43.             }
44.             LOGE("Hello JNI: failed to open hello device.");
45.             return -1;
46.         }
47.         LOGE("Hello JNI: failed to get hello stub module.");
48.         return -1;
49.     }
50.         /*JNI 方法表*/
51.     static const JNINativeMethod method_table[] = {
52.         {"init_native", "()Z", (void*)hello_init},
53.         {"setVal_native", "(I)V", (void*)hello_setVal},
54.         {"getVal_native", "()I", (void*)hello_getVal},
55.     };
56.         /*注册 JNI 方法*/
57.     int register_android_server_HelloService(JNIEnv *env) {
58.             return jniRegisterNativeMethods(env, "com/android/server/HelloSe
    rvice", method_table, NELEM(method_table));
59.     }
60. };
```

注意，在 hello_init 函数中，通过 Android 硬件抽象层提供的 hw_get_module 方法来加载模块 ID 为 HELLO_HARDWARE_MODULE_ID 的硬件抽象层模块，其中，HELLO_HARDWARE_MODULE_ID 是在<hardware/hello.h>中定义的。Android 硬件抽象层会根据 HELLO_HARDWARE_MODULE_ID 的值在 Android 系统的/system/lib/hw 目录中找到相应的模块，然后加载起来，并且返回 hw_module_t 接口给调用者使用。在 jniRegisterNativeMethods 函数中，第二个参数的值必须对应 HelloService 所在的包的路径，即 com.android.server.HelloService。

三. 修改同目录下的 onload.cpp 文件，首先在 namespace android 增加 register_android_server_HelloService 函数声明：

namespace android {

.....

int register_android_server_HelloService(JNIEnv *env);

};

在 JNI_onLoad 增加 register_android_server_HelloService 函数调用：

extern "C" jint JNI_onLoad(JavaVM* vm, void* reserved)

{

.....

register_android_server_HelloService(env);

.....

}

这样，在 Android 系统初始化时，就会自动加载该 JNI 方法调用表。

四. 修改同目录下的 Android.mk 文件，在 LOCAL_SRC_FILES 变量中增加一行：

LOCAL_SRC_FILES:= \

com_android_server_AlarmManagerService.cpp \

com_android_server_BatteryService.cpp \

com_android_server_InputManager.cpp \

com_android_server_LightsService.cpp \

com_android_server_PowerManagerService.cpp \

com_android_server_SystemServer.cpp \

com_android_server_UsbService.cpp \

com_android_server_VibratorService.cpp \

com_android_server_location_GpsLocationProvider.cpp \

com_android_server_HelloService.cpp /

onload.cpp

五. 编译和重新找亿 system.img：

USER-NAME@MACHINE-NAME:~/Android$ mmm frameworks/base/services/jni

USER-NAME@MACHINE-NAME:~/Android$ make snod

这样，重新打包的 system.img 镜像文件就包含我们刚才编写的 JNI 方法了，也就是我们可以通过 Android 系统的 Application Frameworks 层提供的硬件服务 HelloService 来调用这些 JNI 方法，进而调用低层的硬件抽象层接口去访问硬件了。

前面提到，在这篇文章中，我们暂时忽略了 HelloService 类的实现，在下一篇文章中，我们将描述如何实现硬件服务 HelloService，敬请关注。