       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](http://blog.csdn.net/luoshengyang/article/details/6573809)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](http://blog.csdn.net/luoshengyang/article/details/6575988)

1. #define LOG\_TAG "HelloService"
2. #include "jni.h"
3. #include "JNIHelp.h"
4. #include "android\_runtime/AndroidRuntime.h"
5. #include <utils/misc.h>
6. #include <utils/Log.h>
7. #include <hardware/hardware.h>
8. #include <hardware/hello.h>
9. #include <stdio.h>

接着定义hello\_init、hello\_getVal和hello\_setVal三个JNI方法：

[view plain](http://blog.csdn.net/luoshengyang/article/details/6575988)

1. **namespace** android
2. {
3. /\*在硬件抽象层中定义的硬件访问结构体，参考<hardware/hello.h>\*/
4. **struct** hello\_device\_t\* hello\_device = NULL;
5. /\*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\*/
6. **static** **void** hello\_setVal(JNIEnv\* env, jobject clazz, jint value) {
7. **int** val = value;
8. LOGI("Hello JNI: set value %d to device.", val);
9. **if**(!hello\_device) {
10. LOGI("Hello JNI: device is not open.");
11. **return**;
12. }
14. hello\_device->set\_val(hello\_device, val);
15. }
16. /\*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\*/
17. **static** jint hello\_getVal(JNIEnv\* env, jobject clazz) {
18. **int** val = 0;
19. **if**(!hello\_device) {
20. LOGI("Hello JNI: device is not open.");
21. **return** val;
22. }
23. hello\_device->get\_val(hello\_device, &val);
25. LOGI("Hello JNI: get value %d from device.", val);
27. **return** val;
28. }
29. /\*通过硬件抽象层定义的硬件模块打开接口打开硬件设备\*/
30. **static** **inline** **int** hello\_device\_open(**const** hw\_module\_t\* module, **struct** hello\_device\_t\*\* device) {
31. **return** module->methods->open(module, HELLO\_HARDWARE\_MODULE\_ID, (**struct** hw\_device\_t\*\*)device);
32. }
33. /\*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;
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/HelloService", 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。