[Android下如何通过JNI方法向上提供接口总结](http://blog.csdn.net/flydream0/article/details/7088514)

标签： [android](http://www.csdn.net/tag/android)[jni](http://www.csdn.net/tag/jni)[server](http://www.csdn.net/tag/server)[struct](http://www.csdn.net/tag/struct)[module](http://www.csdn.net/tag/module)[java](http://www.csdn.net/tag/java)

2011-12-20 21:12 9024人阅读 [评论](http://blog.csdn.net/flydream0/article/details/7088514#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/flydream0/article/details/7088514#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Android（24） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/flydream0/article/details/7088514)

参考文献:

<http://blog.csdn.net/luoshengyang/article/details/6575988>

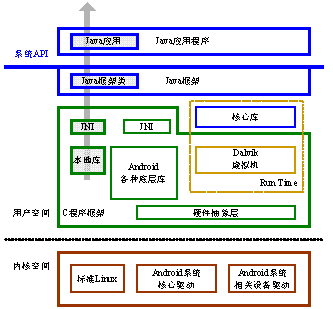
<http://www.androidmi.com/Androidkaifa/rumen/201005/633.html>

**1 什么是JNI**

JNI是Java Native Interface的缩写,即Java本地接口.从Java1.1开始，JNI标准成为Java平台的一部分，它允许java代码和用其它语言编写的代码进行交互.JNI是本地编程接口，它使得在Java虚拟机(VM)内部运行的Java代码能够与用其他编程语言(如C,C++和汇编语言)的应用程序和库进行交互操作.

在Android中提供的JNI的方式，让Java程序可以调用C语言程序。Android中很多Java类都具有native接口，这些native接口就是同本地实现，然后注册到系统中的.

JNI在Android层次结构中的作用如下图所示:



在Android中，主要的JNI代码在以下的路径中:

Android源码根目录/frameworks/base/core/jni/

这个路径中的内容将被编译成库libandroid\_runtime.so,这就是一个普通的动态库，被放置在目标系统的/system/lib目录中.

除此之外，Android还包含其他的JNI库，例如,媒体部分的JNI目录frameworks/base/media/jni/中，被编译成库libmedia\_jni.so.

JNI中的各个文件实际上就是C++的普通文件，其命名一般和支持的Java类有对应关系。这种关系是习惯上的写法，而不是强制的。

在Android中实现的JNI库，需要连接动态库libnativehelper.so.

**2 注册JNI方法**

在Android源码根目录/frameworks/base/services/jni/目录下有一个onload.cpp文件，其内容如下:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7088514) [copy](http://blog.csdn.net/flydream0/article/details/7088514)

1. /\*
2. \* Copyright (C) 2009 The Android Open Source Project
3. \*
4. \* Licensed under the Apache License, Version 2.0 (the "License");
5. \* you may not use this file except in compliance with the License.
6. \* You may obtain a copy of the License at
7. \*
8. \*      http://www.apache.org/licenses/LICENSE-2.0
9. \*
10. \* Unless required by applicable law or agreed to in writing, software
11. \* distributed under the License is distributed on an "AS IS" BASIS,
12. \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13. \* See the License for the specific language governing permissions and
14. \* limitations under the License.
15. \*/
17. #include "JNIHelp.h"
18. #include "jni.h"
19. #include "utils/Log.h"
20. #include "utils/misc.h"
22. **namespace** android {
23. **int** register\_android\_server\_AlarmManagerService(JNIEnv\* env);
24. **int** register\_android\_server\_BatteryService(JNIEnv\* env);
25. **int** register\_android\_server\_InputApplicationHandle(JNIEnv\* env);
26. **int** register\_android\_server\_InputWindowHandle(JNIEnv\* env);
27. **int** register\_android\_server\_InputManager(JNIEnv\* env);
28. **int** register\_android\_server\_LightsService(JNIEnv\* env);
29. **int** register\_android\_server\_PowerManagerService(JNIEnv\* env);
30. **int** register\_android\_server\_UsbDeviceManager(JNIEnv\* env);
31. **int** register\_android\_server\_UsbHostManager(JNIEnv\* env);
32. **int** register\_android\_server\_VibratorService(JNIEnv\* env);
33. **int** register\_android\_server\_SystemServer(JNIEnv\* env);
34. **int** register\_android\_server\_location\_GpsLocationProvider(JNIEnv\* env);
35. **int** register\_android\_server\_connectivity\_Vpn(JNIEnv\* env);
36. **int** register\_android\_server\_HelloService(JNIEnv \*env);
37. };
39. **using** **namespace** android;
41. **extern** "C" jint JNI\_OnLoad(JavaVM\* vm, **void**\* reserved)
42. {
43. JNIEnv\* env = NULL;
44. jint result = -1;
46. **if** (vm->GetEnv((**void**\*\*) &env, JNI\_VERSION\_1\_4) != JNI\_OK) {
47. LOGE("GetEnv failed!");
48. **return** result;
49. }
50. LOG\_ASSERT(env, "Could not retrieve the env!");
52. register\_android\_server\_PowerManagerService(env);
53. register\_android\_server\_InputApplicationHandle(env);
54. register\_android\_server\_InputWindowHandle(env);
55. register\_android\_server\_InputManager(env);
56. register\_android\_server\_LightsService(env);
57. register\_android\_server\_AlarmManagerService(env);
58. register\_android\_server\_BatteryService(env);
59. register\_android\_server\_UsbDeviceManager(env);
60. register\_android\_server\_UsbHostManager(env);
61. register\_android\_server\_VibratorService(env);
62. register\_android\_server\_SystemServer(env);
63. register\_android\_server\_location\_GpsLocationProvider(env);
64. register\_android\_server\_connectivity\_Vpn(env);
65. register\_android\_server\_HelloService(env);
67. **return** JNI\_VERSION\_1\_4;
68. }

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

onload.cpp文件上部分为注册函数的声明，下部分为调用各种注册函数，而这些注册函数就是JNI方法的注册函数! 正有通过这些注册函数，上层才有可能调用注册的JNI方法.

这些注册函数是由同目录下的其他.cpp文件中实现,如上面的register\_android\_server\_HelloService(env)这个函数是在com\_android\_service\_HelloService.cpp文件中实现的.那么编译器又是如何知道这点的呢? 答案当然是Android.mk这个文件，打开这个文件，其内容如下:

**[html]** [view plain](http://blog.csdn.net/flydream0/article/details/7088514) [copy](http://blog.csdn.net/flydream0/article/details/7088514)

1. LOCAL\_PATH:= $(call my-dir)
2. include $(CLEAR\_VARS)
4. LOCAL\_SRC\_FILES:= \
5. com\_android\_server\_AlarmManagerService.cpp \
6. com\_android\_server\_BatteryService.cpp \
7. com\_android\_server\_InputApplicationHandle.cpp \
8. com\_android\_server\_InputManager.cpp \
9. com\_android\_server\_InputWindowHandle.cpp \
10. com\_android\_server\_LightsService.cpp \
11. com\_android\_server\_PowerManagerService.cpp \
12. com\_android\_server\_SystemServer.cpp \
13. com\_android\_server\_UsbDeviceManager.cpp \
14. com\_android\_server\_UsbHostManager.cpp \
15. com\_android\_server\_VibratorService.cpp \
16. com\_android\_server\_location\_GpsLocationProvider.cpp \
17. com\_android\_server\_connectivity\_Vpn.cpp \
18. com\_android\_server\_HelloService.cpp \
19. onload.cpp
21. LOCAL\_C\_INCLUDES += \
22. $(JNI\_H\_INCLUDE) \
23. frameworks/base/services \
24. frameworks/base/core/jni \
25. external/skia/include/core
27. LOCAL\_SHARED\_LIBRARIES := \
28. libandroid\_runtime \
29. libcutils \
30. libhardware \
31. libhardware\_legacy \
32. libnativehelper \
33. libsystem\_server \
34. libutils \
35. libui \
36. libinput \
37. libskia \
38. libgui \
39. libusbhost
41. ifeq ($(WITH\_MALLOC\_LEAK\_CHECK),true)
42. LOCAL\_CFLAGS += -DMALLOC\_LEAK\_CHECK
43. endif
45. LOCAL\_MODULE:= libandroid\_servers
47. include $(BUILD\_SHARED\_LIBRARY)

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

在LOCAL\_SRC\_FILE中给出了所有实现文件(cpp文件)的路径，因此编译就能找到各个注册函数对应的实现文件了.

接下来让我们来看看其中一个注册函数的具体实现过程是如何的，比如:register\_android\_server\_HelloService(env),打开com\_android\_service\_HelloService.cpp文件,其下有注册函数的实现代码，如下:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7088514) [copy](http://blog.csdn.net/flydream0/article/details/7088514)

1. **int** register\_android\_server\_HelloService(JNIEnv \*env) {
2. **return** jniRegisterNativeMethods(env, "com/android/server/HelloService", method\_table, NELEM(method\_table));
3. }

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

其中jniRegisterNativeMethods为注册JNI方法函数，此函数在JNI方法使用中非常重要，此函数的第二个参数为对应着java类即HelloService.java的文件名，第三个参数为注册的方法表:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7088514) [copy](http://blog.csdn.net/flydream0/article/details/7088514)

1. /\*JNI方法表\*/
2. **static** **const** JNINativeMethod method\_table[] = {
3. {"init\_native", "()Z", (**void**\*)hello\_init},
4. {"setVal\_native", "(I)V", (**void**\*)hello\_setVal},
5. {"getVal\_native", "()I", (**void**\*)hello\_getVal},
6. };

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

接下来就是方法表内各个接口的实现代码了.

如hello\_setVal函数的实现:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7088514) [copy](http://blog.csdn.net/flydream0/article/details/7088514)

1. /\*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\*/
2. **static** **void** hello\_setVal(JNIEnv\* env, jobject clazz, jint value) {
3. **int** val = value;
4. LOGI("Hello JNI: set value %d to device.", val);
5. **if**(!hello\_device) {
6. LOGI("Hello JNI: device is not open.");
7. **return**;
8. }
10. hello\_device->set\_val(hello\_device, val);
11. }

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

方法列表中的hello\_init的实现代码中展现了如何调用下层HAL提供的接口, 还记得上一章: [Android中HAL如何向上层提供接口总结](http://blog.csdn.net/flydream0/article/details/7086273) 一文中描述HAL是如何向上层提供接口的吗?这个hello\_init函数的实现就是典型的调用HAL提供的初始化接口的例子,下面见hello\_init这个函数的实现代码:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7088514) [copy](http://blog.csdn.net/flydream0/article/details/7088514)

1. /\*通过硬件抽象层定义的硬件模块打开接口打开硬件设备\*/
2. **static** **inline** **int** hello\_device\_open(**const** hw\_module\_t\* module, **struct** hello\_device\_t\*\* device) {
3. **return** module->methods->open(module, HELLO\_HARDWARE\_MODULE\_ID, (**struct** hw\_device\_t\*\*)device);
4. }
5. /\*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\*/
6. **static** jboolean hello\_init(JNIEnv\* env, jclass clazz) {
7. hello\_module\_t\* module;
9. LOGI("Hello JNI: initializing......");
10. **if**(hw\_get\_module(HELLO\_HARDWARE\_MODULE\_ID, (**const** **struct** hw\_module\_t\*\*)&module) == 0) {
11. LOGI("Hello JNI: hello Stub found.");
12. **if**(hello\_device\_open(&(module->common), &hello\_device) == 0) {
13. LOGI("Hello JNI: hello device is open.");
14. **return** 0;
15. }
16. LOGE("Hello JNI: failed to open hello device.");
17. **return** -1;
18. }
19. LOGE("Hello JNI: failed to get hello stub module.");
20. **return** -1;
21. }

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

上述的module->methods->open这个open函数就是HAL提供的接口，其函数原型在hardware.h头文件中有定义，只能返回struct hw\_device\_t类型的指针,而在JNI方法中，我们关心的是struct hello\_device\_t，只有通过struct hello\_device\_t，我们才能获取其所有的成员函数（接HAL提供的接口函数），由于struct hello\_device\_t的第一个成员就是struct hw\_device\_t类型的数据，因此在这里可以将获取的struct hw\_device\_t强制转化为struct hello\_device\_t来用。还没有明白过来的，建议回过头去看上一篇文章:[Android中HAL如何向上层提供接口总结](http://blog.csdn.net/flydream0/article/details/7086273) .

**3 方法列表说明**

关于static const JNINativeMethod method\_table[]方法列表的原型如下:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7088514) [copy](http://blog.csdn.net/flydream0/article/details/7088514)

1. **typedef** **struct** {
2. **const** **char**\* name;
3. **const** **char**\* signature;
4. **void**\* fnPtr;
5. } JNINativeMethod;

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

Andoird 中使用了一种不同传统Java JNI的方式来定义其native的函数。其中很重要的区别是Andorid使用了一种Java 和 C 函数的映射表数组，并在其中描述了函数的参数和返回值。这个数组的类型就是JNINativeMethod，见上述定义.

第一个变量name是Java中函数的名字。第二个变量signature，用字符串是描述了函数的参数和返回值.第三个变量fnPtr是函数指针，指向C函数。

其中比较难以理解的是第二个参数，例如:

"()V"

"(II)V"

"(Ljava/lang/String;Ljava/lang/String;)V"

实际上这些字符是与函数的参数类型一一对应的。

"()" 中的字符表示参数，后面的则代表返回值。例如"()V" 就表示void Func();

"(II)V" 表示 void Func(int, int);

具体的每一个字符的对应关系如下  
字符 Java类型    C类型

V      void              void  
Z       jboolean     boolean  
I        jint                int  
J       jlong            long  
D      jdouble       double  
F      jfloat            float  
B      jbyte            byte  
C      jchar           char  
S      jshort          short

数组则以"["开始，用两个字符表示

[I       jintArray      int[]  
[F     jfloatArray    float[]  
[B     jbyteArray    byte[]  
[C    jcharArray    char[]  
[S    jshortArray   short[]  
[D    jdoubleArray double[]  
[J     jlongArray     long[]  
[Z    jbooleanArray boolean[]

上面的都是基本类型。如果Java函数的参数是class，则以"L"开头，以";"结尾.中间是用"/" 隔开的包及类名。而其对应的C函数名的参数则为jobject. 一个例外是String类，其对应的类为jstring

Ljava/lang/String;   String jstring  
Ljava/net/Socket;   Socket jobject

如果JAVA函数位于一个嵌入类，则用$作为类名间的分隔符。

例如 "(Ljava/lang/String;Landroid/os/FileUtils$FileStatus;)Z"