[Android Jni调用浅述](http://blog.csdn.net/flydream0/article/details/7371692)

标签： [android](http://www.csdn.net/tag/android)[jni](http://www.csdn.net/tag/jni)[java](http://www.csdn.net/tag/java)[server](http://www.csdn.net/tag/server)[interface](http://www.csdn.net/tag/interface)[虚拟机](http://www.csdn.net/tag/%e8%99%9a%e6%8b%9f%e6%9c%ba)

2012-03-20 12:06 5980人阅读 [评论](http://blog.csdn.net/flydream0/article/details/7371692#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/flydream0/article/details/7371692#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/7371692)

**声明：欢迎转载，转载时请注明出处!** <http://blog.csdn.net/flydream0/article/details/7371692>

**1 简述**

    JNI是Java Native Interface的缩写，中文为JAVA本地调用。从Java1.1开始，Java Native Interface(JNI)标准成为java平台的一部分，它允许Java代码和其他语言写的代码进行交互。JNI一开始是为了本地已编译语言，尤其是C和C++而设计的，但是它并不妨碍你使用其他语言，只要调用约定受支持就可以了。

    由于Android的应用层的类都是以Java写的，这些Java类编译为Dex型式的字节码之后，必须依靠Dalvik虚拟机来运行，在Android中Dalvik虚拟机扮演很重要的角色.而Android中间件是由C/C++写的，这些C/C+＋写的组件并不是在Dalvik虚拟机上运行的。那么应用层上的Java代码又是如何与C/C＋＋写的组件之间又是如何沟通的？

**2 载入.so文件**

System.loadLibrary(\*.so文件);

在java代码中，可以通过loadLibrary要求VM装载so文件，java代码一般如下形式：

**[java]** [view plain](http://blog.csdn.net/flydream0/article/details/7371692) [copy](http://blog.csdn.net/flydream0/article/details/7371692)

1. **public** **class** jnitest {
2. **static** {
3. System.loadLibrary("jnitest");
4. }
5. //...
6. }

上述代码运行时将会在/system/lib/目录下查找libjnitest.so文件，将载入VM,这样，java代码和C组件之间就构成了联系，接下来就可以通过一些方法可以相互调用了.

**3 JNI\_OnLoad与JNI\_OnUnload**

    在Android中，当程序在java层运行System.loadLibrary("jnitest");这行代码后，程序会去载入libjnitest.so文件，与此同时，产生一个"Load"事件，这个事件触发后，程序默认会在载入的.so文件的函数列表中查找JNI\_OnLoad函数并执行，与"Load"事件相对，当载入的.so文件被卸载时，“Unload”事件被触发，此时，程序默认会去在载入的.so文件的函数列表中查找JNI\_OnUnload函数并执行，然后卸载.so文件。需要注意的是，JNI\_OnLoad与JNI\_OnUnload这两个函数在.so组件中并不是强制要求的，用户也可以不去实现，java代码一样可以调用到C组件中的函数，在接下来的章节中会讲到这点.

    之所以在C组件中去实现这两个函数（特别是JNI\_OnLoad函数），往往是做一个初始化工作或“善后”工作。可以这样认为，将JNI\_ONLoad看成是.so组件的初始化函数，当其第一次被装载时被执行（window下的dll文件也可类似的机制，在\_DLL\_Main()函数中，通过一个swith case语句来识别当前是载入还是卸载）。将JNI\_OnUnload函数看成是析构函数，当其被卸载时被调用。

    由此看来，就不难明白为什么很多jni C组件中会实现JNI\_OnLoad这个函数了。 一般情况下，在C组件中的JNI\_OnLoad函数用来实现给VM注册接口，以方便VM可以快速的找到Java代码需要调用的C函数。（此外，JNI\_OnLoad函数还有另外一个功能，那就是告诉VM此C组件使用那一个JNI版本，如果未实现JNI\_OnLoad函数，则默认是JNI 1.1版本）。

**4 显式注册native方法**

**4.1 显式注册的作用:**

    应用层的Java类别通过VM而调用到native函数。一般是通过VM去寻找\*.so里的native函数。如果需要连续呼叫很多次，每次都需要寻找一遍，会多花许多时间。此时，C组件开发者可以将本地函数向VM进行注册,以便能加快后续调用native函数的效率.可以这么想象一下，假设VM内部一个native函数链表，初始时是空的，在未显式注册之前此native函数链表是空的，每次java调用native函数之前会首先在此链表中查找需要查找需要调用的native函数，如果找到就直接使用，如果未找到，得再通过载入的.so文件中的函数列表中去查找，且每次java调用native函数都是进行这样的流程，因此，效率就自然会下降，为了克服这样现象，我们可以通过在.so文件载入初始化时，即JNI\_OnLoad函数中，先行将native函数注册到VM的native函数链表中去，这样一来，后续每次java调用native函数时都会在VM中的native函数链表中找到对应的函数，从而加快速度.

注：在Android 源码开发环境下，大多采用显式注册native方法.

**4.2 在Android源码开发模式下有两种方法可以实现显示注册native方法:**

**方法一: 使用JNIHelp.h头文件中定义的jniRegisterNativeMethods来实现.**

如~/WORKING\_DIRECTORY/frameworks/base/services/jni/com\_android\_server\_location\_GpsLocationProvider.cpp：

注:此文件同级目录中的其它cpp文件大多采用此种方法进行native方法显式注册.

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7371692) [copy](http://blog.csdn.net/flydream0/article/details/7371692)

1. **static** JNINativeMethod sMethods[] = {
2. /\* name, signature, funcPtr \*/
3. {"class\_init\_native", "()V", (**void** \*)android\_location\_GpsLocationProvider\_class\_init\_native},
4. {"native\_is\_supported", "()Z", (**void**\*)android\_location\_GpsLocationProvider\_is\_supported},
5. {"native\_init", "()Z", (**void**\*)android\_location\_GpsLocationProvider\_init},
6. {"native\_cleanup", "()V", (**void**\*)android\_location\_GpsLocationProvider\_cleanup},
7. {"native\_set\_position\_mode", "(IIIII)Z", (**void**\*)android\_location\_GpsLocationProvider\_set\_position\_mode},
8. {"native\_start", "()Z", (**void**\*)android\_location\_GpsLocationProvider\_start},
9. {"native\_stop", "()Z", (**void**\*)android\_location\_GpsLocationProvider\_stop},
10. {"native\_delete\_aiding\_data", "(I)V", (**void**\*)android\_location\_GpsLocationProvider\_delete\_aiding\_data},
11. {"native\_read\_sv\_status", "([I[F[F[F[I)I", (**void**\*)android\_location\_GpsLocationProvider\_read\_sv\_status},
12. {"native\_read\_nmea", "([BI)I", (**void**\*)android\_location\_GpsLocationProvider\_read\_nmea},
13. {"native\_inject\_time", "(JJI)V", (**void**\*)android\_location\_GpsLocationProvider\_inject\_time},
14. {"native\_inject\_location", "(DDF)V", (**void**\*)android\_location\_GpsLocationProvider\_inject\_location},
15. {"native\_supports\_xtra", "()Z", (**void**\*)android\_location\_GpsLocationProvider\_supports\_xtra},
16. {"native\_inject\_xtra\_data", "([BI)V", (**void**\*)android\_location\_GpsLocationProvider\_inject\_xtra\_data},
17. {"native\_agps\_data\_conn\_open", "(Ljava/lang/String;)V", (**void**\*)android\_location\_GpsLocationProvider\_agps\_data\_conn\_open},
18. {"native\_agps\_data\_conn\_closed", "()V", (**void**\*)android\_location\_GpsLocationProvider\_agps\_data\_conn\_closed},
19. {"native\_agps\_data\_conn\_failed", "()V", (**void**\*)android\_location\_GpsLocationProvider\_agps\_data\_conn\_failed},
20. {"native\_agps\_set\_id","(ILjava/lang/String;)V",(**void**\*)android\_location\_GpsLocationProvider\_agps\_set\_id},
21. {"native\_agps\_set\_ref\_location\_cellid","(IIIII)V",(**void**\*)android\_location\_GpsLocationProvider\_agps\_set\_reference\_location\_cellid},
22. {"native\_set\_agps\_server", "(ILjava/lang/String;I)V", (**void**\*)android\_location\_GpsLocationProvider\_set\_agps\_server},
23. {"native\_send\_ni\_response", "(II)V", (**void**\*)android\_location\_GpsLocationProvider\_send\_ni\_response},
24. {"native\_agps\_ni\_message", "([BI)V", (**void** \*)android\_location\_GpsLocationProvider\_agps\_send\_ni\_message},
25. {"native\_get\_internal\_state", "()Ljava/lang/String;", (**void**\*)android\_location\_GpsLocationProvider\_get\_internal\_state},
26. {"native\_update\_network\_state", "(ZIZZLjava/lang/String;Ljava/lang/String;)V", (**void**\*)android\_location\_GpsLocationProvider\_update\_network\_state },
27. };
29. **int** register\_android\_server\_location\_GpsLocationProvider(JNIEnv\* env)
30. {
31. **return** jniRegisterNativeMethods(env, "com/android/server/location/GpsLocationProvider", sMethods, NELEM(sMethods));
32. }

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

其中jniRegisterNativeMethods和NELEM都是在头文件JNIHelp.h定义的，得:

**[plain]** [view plain](http://blog.csdn.net/flydream0/article/details/7371692) [copy](http://blog.csdn.net/flydream0/article/details/7371692)

1. #include "JNIHelp.h"

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

在Android.mk文件中得加上:

**[plain]** [view plain](http://blog.csdn.net/flydream0/article/details/7371692) [copy](http://blog.csdn.net/flydream0/article/details/7371692)

1. LOCAL\_SHARED\_LIBRARIES +=libnativehelper

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

**方法二:使用AndroidRuntime::registerNativeMethods**

如~/WORKING\_DIRECTORY/frameworks/base/media/jni/android\_media\_MediaPlayer.cpp：

注：当前目录下其它cpp文件大多采用此种方法进行显式native注册.

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7371692) [copy](http://blog.csdn.net/flydream0/article/details/7371692)

1. **static** JNINativeMethod gMethods[] = {
2. {"setDataSource",       "(Ljava/lang/String;)V",            (**void** \*)android\_media\_MediaPlayer\_setDataSource},
4. {
5. "\_setDataSource",
6. "(Ljava/lang/String;[Ljava/lang/String;[Ljava/lang/String;)V",
7. (**void** \*)android\_media\_MediaPlayer\_setDataSourceAndHeaders
8. },
10. {"setDataSource",       "(Ljava/io/FileDescriptor;JJ)V",    (**void** \*)android\_media\_MediaPlayer\_setDataSourceFD},
11. {"\_setVideoSurface",    "(Landroid/view/Surface;)V",        (**void** \*)android\_media\_MediaPlayer\_setVideoSurface},
12. {"prepare",             "()V",                              (**void** \*)android\_media\_MediaPlayer\_prepare},
13. {"prepareAsync",        "()V",                              (**void** \*)android\_media\_MediaPlayer\_prepareAsync},
14. {"\_start",              "()V",                              (**void** \*)android\_media\_MediaPlayer\_start},
15. {"\_stop",               "()V",                              (**void** \*)android\_media\_MediaPlayer\_stop},
16. {"getVideoWidth",       "()I",                              (**void** \*)android\_media\_MediaPlayer\_getVideoWidth},
17. {"getVideoHeight",      "()I",                              (**void** \*)android\_media\_MediaPlayer\_getVideoHeight},
18. {"seekTo",              "(I)V",                             (**void** \*)android\_media\_MediaPlayer\_seekTo},
19. //...
20. }
21. // This function only registers the native methods
22. **static** **int** register\_android\_media\_MediaPlayer(JNIEnv \*env)
23. {
24. **return** AndroidRuntime::registerNativeMethods(env,
25. "android/media/MediaPlayer", gMethods, NELEM(gMethods));
26. }

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

其中AndroidRuntime::registerNativeMethods是在头文件android\_runtime/AndroidRuntime.h中定义，使用时得:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7371692) [copy](http://blog.csdn.net/flydream0/article/details/7371692)

1. #include "android\_runtime/AndroidRuntime.h"

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

且得在Android.mk文件中加上:

**[plain]** [view plain](http://blog.csdn.net/flydream0/article/details/7371692) [copy](http://blog.csdn.net/flydream0/article/details/7371692)

1. LOCAL\_SHARED\_LIBRARIES += \
2. libandroid\_runtime

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

有关函数命名，请参考博客内另一篇文章：[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)。

注：以上两种显式注册native方法都只是适用于Android源码开发环境下，至于在Android NDK环境下如何显式注册native方法，暂时还没有研究过，且此两种方法目前NDK还不支持,NDK开发模式下一般采用隐式注册native函数，即接下来要讲的内容.

**5 隐式注册native方法**

前面第3节已经讲到，JNI\_OnLoad和JNI\_OnUnload函数并不是强制要求实现的，在这种情况下，就相当于在载入.so文件时，没有了初始化函数，既然没有了初始化函数，那显式注册native方法也行不通了。那这个时候又该如何让应用层的java代码调用下层的C函数呢?

    幸运地是，即使我们不使用任何代码做native函数显式注册，应用层的java代码在调用native函数时，也会采用默认的方法到转入的.so文件中的函数列表中查找对应的native 函数，只不过，这个native函数与java类型中声明的native函数的名字之前有一种默认的对应关系。如:

java类的native成员函数:public native int socket\_send(int cmdid,String argus);默认会在.so文件中的函数列表中查找jint JNICALL Java\_com\_hase\_bclm\_bclm\_socket\_1send(JNIEnv \*env, jobject obj, jint cmdid, jstring argus);函数，一旦找到此对应的函数，VM就会将此native函数自动注册到VM内部的native函数链表中，以便加快后续相同jni调用.

可接下来的问题是，作为码农的我们，又是如何知道java native成员函数对应着C组件中的native 函数的名字呢?简单地函数名字也许我们能搞定，复杂一点的就要傻眼了。同样幸运地是，JDK提供了一个javah工具，可以用这个工具来自动通过.class文件来生成C组件的头文件.

比如你用java写了一个xxx.java文件，里边的java类型里面声明了一些native成员函数，此xxx.java文件编译后会生成xxx.class文件，那么就在你生成的xxx.class包所在目录输入命令行:

**[plain]** [view plain](http://blog.csdn.net/flydream0/article/details/7371692) [copy](http://blog.csdn.net/flydream0/article/details/7371692)

1. $javah -jni com.packagename.yourclassname

就会在当前目录下生成一个头文件。

比如：

javah -jni com.test.example

则在当前目录下生成 com\_test\_example.h头文件.

当前目录下的com结构为:com/test/example.class

再将此头文件拷贝到你的C组件工程内，实现其中声明的native函数即可.

之前在显式注册native函数的相关章节中已经说明，显式注册是将native函数添加到VM内部的native函数链表中，以加快后续jni调用的效率，其实在隐性native 注册时，每一次执行某个jni调用时，VM在.so函数列表中找到对应的native函数后，同样也会将其注册到VM内部的native函数链表中，由此看到，隐式native注册的方法除了第一次执某个jni调用时会稍微速度慢点外，后续同样的调用就会直接在VM内部的native函数链表中找到对应的native函数，这样看来，显式与隐式注册native方法，其实效率相差无几.

OK,到此结束!