**为何要用到NDK?**  
  
前 段时间，工作中需要用android的应用程序访问android的根文件系统中的文件，但是由于应用程序的权限限制，无法访问这些文件，怎么办？ 所以 这就要用到了NDK编程了，既用C/C++代码实现访问系统文件，并将其生成本地库，供android中的java代码调用，这样就可以在java代码中 通过调用C/C++编写的库的接口来实现对系统文件的访问。  
  
**下面就介绍下Android NDK的入门学习过程：**  
  
入门的最好办法就是学习Android自带的例子， 这里就通过学习Android的NDK自带的demo程序：hello-jni来达到这个目的。  
  
**一、 开发环境的搭建**  
  
a.  android的NDK开发需要在linux下进行： 因为需要把C/C++编写的代码生成能在arm上运行的.so文件，这就需要用到交叉编译环境，而交叉编译需要在linux系统下才能完成。  
  
b. 安装android-ndk开发包，这个开发包可以在google android 官网下载： 通过这个开发包的工具才能将android jni 的C/C++的代码编译成库  
  
c. android应用程序开发环境： 包括eclipse、java、 android sdk、 adt等。  
  
如何下载和安装android-ndk我这里就不啰嗦了，安装完之后，需要将android-ndk的路劲加到环境变量PATH中：

代码:

sudo gedit /etc/environment

在environment的PATH环境变量中添加你的android-ndk的安装路劲，然后再让这个更改的环境变量立即生效：

代码:

source  /etc/environment

经过了上述步骤，在命令行下敲： 

代码:

ndk-bulid

弹出如下的错误，而不是说ndk-build not found，就说明ndk环境已经安装成功了。

代码:

Android NDK: Could not find application project directory !     Android NDK: Please define the NDK\_PROJECT\_PATH variable to point to it.     /home/braincol/workspace/android/android-ndk-r5/build/core/build-local.mk:85: \*\*\* Android NDK: Aborting    .  Stop.

**二、代码的编写**  
  
**1.首先是写java代码**  
  
建立一个Android应用工程HelloJni，创建HelloJni.java文件：  
  
HelloJni.java :

代码:

/\*   \* Copyright (C) 2009 The Android Open Source Project   \*   \* Licensed under the Apache License, Version 2.0 (the "License");   \* you may not use this file except in compliance with the License.   \* You may obtain a copy of the License at   \*   \*      http://www.apache.org/licenses/LICENSE-2.0   \*   \* Unless required by applicable law or agreed to in writing, software   \* distributed under the License is distributed on an "AS IS" BASIS,   \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.   \* See the License for the specific language governing permissions and   \* limitations under the License. \*/    package com.example.hellojni;  import android.app.Activity;  import android.widget.TextView;  import android.os.Bundle;    public class HelloJni extends Activity{             /\*\* Called when the activity is first created. \*/                      @Override             public void onCreate(Bundle savedInstanceState)    {                        super.onCreate(savedInstanceState);                        /\* Create a TextView and set its content.                          \* the text is retrieved by calling a native                          \* function.         \*/                                         TextView  tv = new TextView(this);                        tv.setText( stringFromJNI() );                        setContentView(tv);             }             /\* A native method that is implemented by the               \* 'hello-jni' native library, which is packaged               \* with this application.     \*/                        public native String  stringFromJNI();              /\* This is another native method declaration that is \*not\*                \* implemented by 'hello-jni'. This is simply to show that                \* you can declare as many native methods in your Java code                \* as you want, their implementation is searched in the                \* currently loaded native libraries only the first time                \* you call them.                \*                \* Trying to call this function will result in a                \* java.lang.UnsatisfiedLinkError exception !     \*/                         public native String  unimplementedStringFromJNI();              /\* this is used to load the 'hello-jni' library on application                \* startup. The library has already been unpacked into               \* /data/data/com.example.HelloJni/lib/libhello-jni.so at                \* installation time by the package manager.                \*/                        static {                         System.loadLibrary("hello-jni");              }  }

这段代码很简单，注释也很清晰，这里只提两点：  
  
a:  
static{   
System.loadLibrary("hello-jni");   
}  
  
表 明程序开始运行的时候会加载hello-jni, static区声明的代码会先于onCreate方法执行。如果你的程序中有多个类，而且如果 HelloJni这个类不是你应用程序的入口，那么hello-jni（完整的名字是libhello-jni.so）这个库会在第一次使用 HelloJni这个类的时候加载。  
  
b:   
public native String stringFromJNI();   
public native String unimplementedStringFromJNI();   
  
  
可以看到这两个方法的声明中有 native 关键字， 这个关键字表示这两个方法是本地方法，也就是说这两个方法是通过本地代码（C/C++）实现的，在java代码中仅仅是声明。  
  
用eclipse编译该工程，生成相应的.class文件，这步必须在下一步之前完成，因为生成.h文件需要用到相应的.class文件。  
  
**2. 编写相应的C/C++代码**  
  
刚开始学的时候，有个问题会让人很困惑，相应的C/C++代码如何编写，函数名如何定义？ 这里讲一个方法，利用javah这个工具生成相应的.h文件，然后根据这个.h文件编写相应的C/C++代码。  
  
**2.1 生成相应.h文件：**  
  
就拿我这的环境来说，首先在终端下进入刚刚建立的HelloJni工程的目录：

代码:

braincol@ubuntu:~$ cd workspace/android/NDK/hello-jni/

ls查看工程文件

代码:

braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ls  AndroidManifest.xml  assets  bin  default.properties  gen  res  src

可以看到目前仅仅有几个标准的android应用程序的文件（夹）。  
首先我们在工程目录下建立一个jni文件夹：

代码:

braincol@ubuntu:~/workspace/android/NDK/hello-jni$ mkdir jni  braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ls  AndroidManifest.xml  assets  bin  default.properties  gen  jni  res  src

下面就可以生成相应的.h文件了：

代码:

braincol@ubuntu:~/workspace/android/NDK/hello-jni$ javah -classpath bin -d jni com.example.hellojni.HelloJni

-classpath bin：表示类的路劲  
-d jni: 表示生成的头文件存放的目录  
com.example.hellojni.HelloJni 则是完整类名  
  
这一步的成功要建立在已经在 bin/com/example/hellojni/  目录下生成了 HelloJni.class的基础之上。  
  
现在可以看到jni目录下多了个.h文件：

代码:

braincol@ubuntu:~/workspace/android/NDK/hello-jni$ cd jni/  braincol@ubuntu:~/workspace/android/NDK/hello-jni/jni$ ls  com\_example\_hellojni\_HelloJni.h

我们来看看com\_example\_hellojni\_HelloJni.h的内容：  
  
com\_example\_hellojni\_HelloJni.h :

代码:

/\* DO NOT EDIT THIS FILE - it is machine generated \*/ #include <jni.h> /\* Header for class com\_example\_hellojni\_HelloJni \*/ #ifndef \_Included\_com\_example\_hellojni\_HelloJni #define \_Included\_com\_example\_hellojni\_HelloJni #ifdef \_\_cplusplusextern "C" { #endif /\*   \* Class:     com\_example\_hellojni\_HelloJni   \* Method:    stringFromJNI   \* Signature: ()Ljava/lang/String;   \*/  JNIEXPORT jstring JNICALL Java\_com\_example\_hellojni\_HelloJni\_stringFromJNI  (JNIEnv \*, jobject); /\*   \* Class:     com\_example\_hellojni\_HelloJni   \* Method:    unimplementedStringFromJNI   \* Signature: ()Ljava/lang/String; \*/ JNIEXPORT jstring JNICALL Java\_com\_example\_hellojni\_HelloJni\_unimplementedStringFromJNI  (JNIEnv \*, jobject); #ifdef \_\_cplusplus} #endif #endif

上面代码中的JNIEXPORT 和 JNICALL 是jni的宏，在android的jni中不需要，当然写上去也不会有错。  
  
从上面的源码中可以看出这个函数名那是相当的长啊。。。。 不过还是很有规律的， 完全按照：java\_pacakege\_class\_mathod 形式来命名。  
  
也就是说：  
  
Hello.java中 stringFromJNI() 方法对应于 C/C++中的 Java\_com\_example\_hellojni\_HelloJni\_stringFromJNI() 方法  
  
HelloJni.java中的 unimplementedStringFromJNI() 方法对应于 C/C++中的 Java\_com\_example\_hellojni\_HelloJni\_unimplementedStringFromJNI() 方法  
  
注意下其中的注释：

代码:

Signature: ()Ljava/lang/String;

()Ljava/lang/String;  
  
()表示函数的参数为空（这里为空是指除了JNIEnv \*, jobject 这两个参数之外没有其他参数，JNIEnv\*, jobject是所有jni函数必有的两个参数，分别表示jni环境和对应的java类（或对象）本身），  
  
Ljava/lang/String; 表示函数的返回值是java的String对象。  
  
**2.2 编写相应的.c文件：**  
  
hello-jni.c :

代码:

/\*   \* Copyright (C) 2009 The Android Open Source Project   \*   \* Licensed under the Apache License, Version 2.0 (the "License");   \* you may not use this file except in compliance with the License.   \* You may obtain a copy of the License at   \*   \*      http://www.apache.org/licenses/LICENSE-2.0   \*   \* Unless required by applicable law or agreed to in writing, software   \* distributed under the License is distributed on an "AS IS" BASIS,   \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.   \* See the License for the specific language governing permissions and   \* limitations under the License.   \*   \*/   #include <string.h>#include <jni.h> /\* This is a trivial JNI example where we use a native method   \* to return a new VM String. See the corresponding Java source   \* file located at:   \*   \*   apps/samples/hello-jni/project/src/com/example/HelloJni/HelloJni.java   \*/   jstringJava\_com\_example\_hellojni\_HelloJni\_stringFromJNI( JNIEnv\* env, jobject thiz ){         return (\*env)->NewStringUTF(env, "Hello from JNI !"); }

这 里只是实现了Java\_com\_example\_hellojni\_HelloJni\_stringFromJNI方法， 而 Java\_com\_example\_hellojni\_HelloJni\_unimplementedStringFromJNI 方法并没有实现， 因为在HelloJni.java中只调用了stringFromJNI()方法，所以unimplementedStringFromJNI()方法没 有实现也没关系，不过建议最好还是把所有java中定义的本地方法都实现了，写个空函数也行啊。。。有总比没有好。  
  
Java\_com\_example\_hellojni\_HelloJni\_stringFromJNI() 函数只是简单的返回了一个内容为 "Hello from JNI !" 的jstring对象（对应于java中的String对象）。  
  
hello-jni.c文件已经编写好了，现在可以把com\_example\_hellojni\_HelloJni.h文件给删了，当然留着也行，只是我还是习惯把不需要的文件给清理干净了。  
  
**3. 编译hello-jni.c 生成相应的库**  
  
**3.1 编写Android.mk文件**  
  
在jni目录下（即hello-jni.c 同级目录下）新建一个Android.mk文件，Android.mk文件是Android 的 makefile文件，内容如下：

代码:

# Copyright (C) 2009 The Android Open Source Project # # Licensed under the Apache License, Version 2.0 (the "License"); # you may not use this file except in compliance with the License. # You may obtain a copy of the License at # #      http://www.apache.org/licenses/LICENSE-2.0 # # Unless required by applicable law or agreed to in writing, software # distributed under the License is distributed on an "AS IS" BASIS, # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. # See the License for the specific language governing permissions and # limitations under the License. # LOCAL\_PATH := $(call my-dir) include $(CLEAR\_VARS) LOCAL\_MODULE    := hello-jni LOCAL\_SRC\_FILES := hello-jni.c include $(BUILD\_SHARED\_LIBRARY)

**LOCAL\_PATH := $(call my-dir)**  
  
一个Android.mk 文件首先必须定义好LOCAL\_PATH变量。它用于在开发树中查找源文件。在这个例子中，宏函数’my-dir’, 由编译系统提供，用于返回当前路径（即包含Android.mk file文件的目录）。  
  
**include $( CLEAR\_VARS)**  
  
CLEAR\_VARS由编译系统提供，指定让GNU MAKEFILE为你清除许多LOCAL\_XXX变量（例如 LOCAL\_MODULE, LOCAL\_SRC\_FILES, LOCAL\_STATIC\_LIBRARIES, 等等...),   
除LOCAL\_PATH 。这是必要的，因为所有的编译控制文件都在同一个GNU MAKE执行环境中，所有的变量都是全局的。  
  
**LOCAL\_MODULE := hello-jni**  
  
编译的目标对象，LOCAL\_MODULE变量必须定义，以标识你在Android.mk文件中描述的每个模块。名称必须是唯一的，而且不包含任何空格。  
  
注意：编译系统会自动产生合适的前缀和后缀，换句话说，一个被命名为'hello-jni'的共享库模块，将会生成'libhello-jni.so'文件。   
  
重要注意事项：   
  
如果你把库命名为‘libhello-jni’，编译系统将不会添加任何的lib前缀，也会生成 'libhello-jni.so'，这是为了支持来源于Android平台的源代码的Android.mk文件，如果你确实需要这么做的话。  
  
**LOCAL\_SRC\_FILES := hello-jni.c**  
  
LOCAL\_SRC\_FILES变量必须包含将要编译打包进模块中的C或C++源代码文件。注意，你不用在这里列出头文件和包含文件，因为编译系统将会自动为你找出依赖型的文件；仅仅列出直接传递给编译器的源代码文件就好。   
  
注意，默认的C++源码文件的扩展名是’.cpp’. 指定一个不同的扩展名也是可能的，只要定义LOCAL\_DEFAULT\_CPP\_EXTENSION变量，不要忘记开始的小圆点（也就是’.cxx’,而不是’cxx’）  
  
**include $(BUILD\_SHARED\_LIBRARY)**  
  
BUILD\_SHARED\_LIBRARY 表示编译生成共享库，是编译系统提供的变量，指向一个GNU Makefile脚本，负责收集自从上次调用 'include $(CLEAR\_VARS)'以来，定义在LOCAL\_XXX变量中的所有信息，并且决定编译什么，如何正确地去做。还 有 BUILD\_STATIC\_LIBRARY变量表示生成静态库：lib$(LOCAL\_MODULE).a， BUILD\_EXECUTABLE 表 示生成可执行文件。  
  
**3.2 生成.so共享库文件**

代码:

braincol@ubuntu:~/workspace/android/NDK/hello-jni/jni$ cd ..  braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ls  AndroidManifest.xml  assets  bin  default.properties  gen  jni  libs  obj  res  src  braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ndk-build  Gdbserver      : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver  Gdbsetup       : libs/armeabi/gdb.setup  Install        : libhello-jni.so => libs/armeabi/libhello-jni.so

可以看到已经正确的生成了libhello-jni.so共享库了， 我们去 libs/armeabi/ 目录下看看：

代码:

braincol@ubuntu:~/workspace/android/NDK/hello-jni$ cd libs/  braincol@ubuntu:~/workspace/android/NDK/hello-jni/libs$ ls  armeabi  braincol@ubuntu:~/workspace/android/NDK/hello-jni/libs$ cd armeabi/  braincol@ubuntu:~/workspace/android/NDK/hello-jni/libs/armeabi$ ls  gdbserver  gdb.setup  libhello-jni.so

**4. 在eclipse重新编译HelloJni工程，生成apk**  
  
eclipse中刷新下HelloJni工程，重新编译生成apk，libhello-jni.so共享库会一起打包在apk文件内。  
  
在模拟器中看看运行结果：  
  
