

宝贵建议请发送至：wangzhenyang@itcast.cn



黑马程序员

itheima.com

-编程，始于黑马

Android 课程同步笔记

Beta 0.01 版

By 阳哥

Android-JNI-03JNI 深入

1. JNI 开发中常见错误 (★★)

1.1 动态库名称写错，或者不存在

```
static{
    System.loadLibrary("hello");
}
```

当我们在写上面代码的时候不小心将 hello 写成了 hell0。或者 libhello.so 动态库不存在，那么系统启动时会报如下异常。

```
com.itheima.hellojni AndroidRuntime FATAL EXCEPTION: main
com.itheima.hellojni AndroidRuntime java.lang.ExceptionInInitializerError
com.itheima.hellojni AndroidRuntime at java.lang.Class.newInstanceImpl(Native Method)
com.itheima.hellojni AndroidRuntime at java.lang.Class.newInstance(Class.java:1319)
com.itheima.hellojni AndroidRuntime at android.app.Instrumentation.newActivity(Instrumentation.java:1053)
com.itheima.hellojni AndroidRuntime at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:1974)
com.itheima.hellojni AndroidRuntime at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2084)
com.itheima.hellojni AndroidRuntime at android.app.ActivityThread.access$600(ActivityThread.java:130)
com.itheima.hellojni AndroidRuntime at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1195)
com.itheima.hellojni AndroidRuntime at android.os.Handler.dispatchMessage(Handler.java:99)
com.itheima.hellojni AndroidRuntime at android.os.Looper.loop(Looper.java:137)
com.itheima.hellojni AndroidRuntime at android.app.ActivityThread.main(ActivityThread.java:4745)
com.itheima.hellojni AndroidRuntime at java.lang.reflect.Method.invokeNative(Native Method)
com.itheima.hellojni AndroidRuntime at java.lang.reflect.Method.invoke(Method.java:511)
com.itheima.hellojni AndroidRuntime at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:786)
com.itheima.hellojni AndroidRuntime at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:553)
com.itheima.hellojni AndroidRuntime at dalvik.system.NativeStart.main(Native Method)
com.itheima.hellojni AndroidRuntime Caused by: java.lang.UnsatisfiedLinkError: Couldn't load hell0: findLibrary returned null
com.itheima.hellojni AndroidRuntime at java.lang.Runtime.loadLibrary(Runtime.java:365)
com.itheima.hellojni AndroidRuntime at java.lang.System.loadLibrary(System.java:535)
com.itheima.hellojni AndroidRuntime at com.itheima.hellojni.MainActivity.<clinit>(MainActivity.java:13)
```

1.2 Android.mk 配置文件写错

如果改配置文件中的某个参数名称被写错，那么我们在调用 ndk-build.cmd 命令的时候很可能报如下异常。

```
F:\heima33\JNI\JNI02\code\03JNI开发常见错误01\jni>ndk-build.cmd
F:\heima33\JNI\JNI02\code\03JNI01\jni\Android.mk:4: *** missing separator. Stop.
F:\heima33\JNI\JNI02\code\03JNI开发常见错误01\jni>
```

出现这类异常，就需要我们检查我们的 Android.mk 文件，看是否写错。

1.3 目标文件名画蛇添足导致的错误

如果我们将 Android.mk 中的目标文件名：`LOCAL_MODULE := hello` 写成了 `LOCAL_MODULE := hello.so`，那么当我们使用 ndk 进行编译的时候会出现如下错误。

```
F:\heima33\JNI\JNI02\code\03JNI开发常见错误01\jni>ndk-build.cmd
Android NDK: F:/heima33/JNI/JNI02/code/03JNI开发常见错误01/jni/Android.mk:hello
.so: LOCAL_MODULE_FILENAME must not contain a file extension
F:/heima33/JNI/JNI02//android-ndk-r9b-windows-x86/android-ndk-r9b/build/core/bui
ld-shared-library.mk:30: *** Android NDK: Aborting . Stop.
```

上面的异常告诉我们，LOCAL_MODULE 不能有文件拓展名。

1.4 源文件名写错

如果我们将 Android.mk 中的源文件名：`LOCAL_SRC_FILES := hello.c` 写成了 `LOCAL_SRC_FILES := helo.c`（少了一个单词），那么当使用 ndk 进行编译的时候会出现如下错误：

```
F:\heima33\JNI\JNI02\code\03JNI开发常见错误01\jni>ndk-build.cmd
make.exe: *** No rule to make target `F:/heima33/JNI/JNI02/code/03JNI01//jni/hel
o.c', needed by `F:/heima33/JNI/JNI02/code/03JNI01//obj/local/armeabi/objs/hello
/helo.o'. Stop.
F:\heima33\JNI\JNI02\code\03JNI开发常见错误01\jni>
```

1.5 平台使用错误

如果我们编译的时候用的是 arm 平台，但是将目标文件运行在了 x86 平台上，那么会产生如下错误。

```
12.715 1568 1568 com.itheima.hellojni dalvikvm threadid=1: thread exiting with uncaught exception (group=unsec/ze)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime FATAL EXCEPTION: main
12.715 1568 1568 com.itheima.hellojni AndroidRuntime java.lang.ExceptionInInitializerError
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at java.lang.Class.newInstanceImpl(Native Method)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at java.lang.Class.newInstance(Class.java:1319)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at android.app.Instrumentation.newActivity(Instrumentation.java:1053)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:1974)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2084)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at android.app.ActivityThread.access$600(ActivityThread.java:130)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1195)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at android.os.Handler.dispatchMessage(Handler.java:99)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at android.os.Looper.loop(Looper.java:137)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at android.app.ActivityThread.main(ActivityThread.java:4745)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at java.lang.reflect.Method.invokeNative(Native Method)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at java.lang.reflect.Method.invoke(Method.java:511)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:7086)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:553)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at dalvik.system.NativeStart.main(Native Method)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime Caused by: java.lang.UnsatisfiedLinkError: Couldn't load hello: findLibrary retu
rned null
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at java.lang.Runtime.loadLibrary(Runtime.java:365)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at java.lang.System.loadLibrary(System.java:535)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime at com.itheima.hellojni.MainActivity.<clinit>(MainActivity.java:13)
12.715 1568 1568 com.itheima.hellojni AndroidRuntime ... 15 more
```


如果想让我们编译的动态库既支持 arm 平台又支持 x86 平台，那么我们可以在工程中的 jni 目录下添加 Application.mk 文件。关于 Application.mk 的配置在本人的上一个文档中有说明，这里就不再介绍。

1.6 C 语言中被 Java 调用的方法名写错

比如 C 语言中的方法 jstring Java_com_itheima_jnihello_MainActivity_helloC 把 helloC 写成了 heloC，那么将会报如下错误。

```
14.712 1674 1674 com.itheima.hellojni dalvikvm threadid=1: thread exiting with uncaught exception (group=0xb5ec7288)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime FATAL EXCEPTION: main
14.712 1674 1674 com.itheima.hellojni AndroidRuntime java.lang.IllegalStateException: Could not execute method of the activity
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at android.view.View$1.onClick(View.java:3591)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at android.view.View.performClick(View.java:4084)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at android.view.View$PerformClick.run(View.java:16966)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at android.os.Handler.handleCallback(Handler.java:615)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at android.os.Handler.dispatchMessage(Handler.java:92)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at android.os.Looper.loop(Looper.java:137)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at android.app.ActivityThread.main(ActivityThread.java:4745)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at java.lang.reflect.Method.invokeNative(Native Method)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at java.lang.reflect.Method.invoke(Method.java:511)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:786)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:553)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at dalvik.system.NativeStart.main(Native Method)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime Caused by: java.lang.reflect.InvocationTargetException
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at java.lang.reflect.Method.invokeNative(Native Method)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at java.lang.reflect.Method.invoke(Method.java:511)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at android.view.View$1.onClick(View.java:3586)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime ... 11 more
14.712 1674 1674 com.itheima.hellojni AndroidRuntime Caused by: java.lang.UnsatisfiedLinkError: Native method not found: com.itheima.hellojni.MainActivity.helloFromC()Ljava/lang/String;
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at com.itheima.hellojni.MainActivity.helloFromC(Native Method)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime at com.itheima.hellojni.MainActivity.click(MainActivity.java:27)
14.712 1674 1674 com.itheima.hellojni AndroidRuntime ... 11 more
```

2. 自动生成 JNI 头文件 (★★)

对于一些特殊的 Java 方法名，我们很难写出其对应的 JNI 方法名，比如：`public native String a_b_c_d()`；这时候我们可以通过我们的 JDK 工具自动生成头文件。该工具位于 JDK 中，如果我们给电脑配置 JAVA_HOME 则可以直接在命令行中使用，使用法则很简单。如下图所示：

```
C:\Users\thinkpad>javah
用法:
  javah [options] <classes>
其中, [options] 包括:
  -o <file>          输出文件 <只能使用 -d 或 -o 之一>
  -d <dir>           输出目录
  -v -verbose         启用详细输出
  -h --help -?       输出此消息
  -version            输出版本信息
  -jni                生成 JNI 样式的标头文件 <默认值>
  -force              始终写入输出文件
  -classpath <path>  从中加载类的路径
  -bootclasspath <path> 从中加载引导类的路径
<classes> 是使用其全限定名称指定的
<例如, java.lang.Object>。
```

下面演示如何使用 javah 工具生成我们 MainActivity.java 中 native 方法的头文件。

1

将命令行控制台切换到我们 MainActivity.class (注意：是.class 不是.java) 所在目录。本人的目录如下：

```
C:\Users\thinkpad>cd C:\Users\thinkpad\workspace\JNI入门\bin\classes  
C:\Users\thinkpad\workspace\JNI入门\bin\classes>
```

2

执行 javah 命令：javah com.itheima.jnihello.MainActivity，发现报了如下错误：

```
C:\Users\thinkpad\workspace\JNI入门\bin\classes>javah com.itheima.jnihello.MainActivity  
错误：无法访问android.app.Activity  
找不到android.app.Activity的类文件  
C:\Users\thinkpad\workspace\JNI入门\bin\classes>
```

Tips

：出现上面的错误时因为我用的是 jdk7 版本，jdk7 在编译 MainActivity.class 类的时候会查找其所有的父

类，但是在当前命令行中是不可能存在 MainActivity 类的父类路径的。有两种办法可以解决上述问题：

- ◆ 1) 改成 jdk6 版本
- ◆ 2) 将 native 方法定义在另外一个独立的类中

本人在这里采用第二种方法来解决该问题，因此我创建一个类，将所有的 native 方法都定义在该类中。



JNIMethod.java 代码清单如下：

```
package com.itheima.jnihello.jni;  
  
public class JNIMethod {  
    public native String a_b__c_d();  
}
```

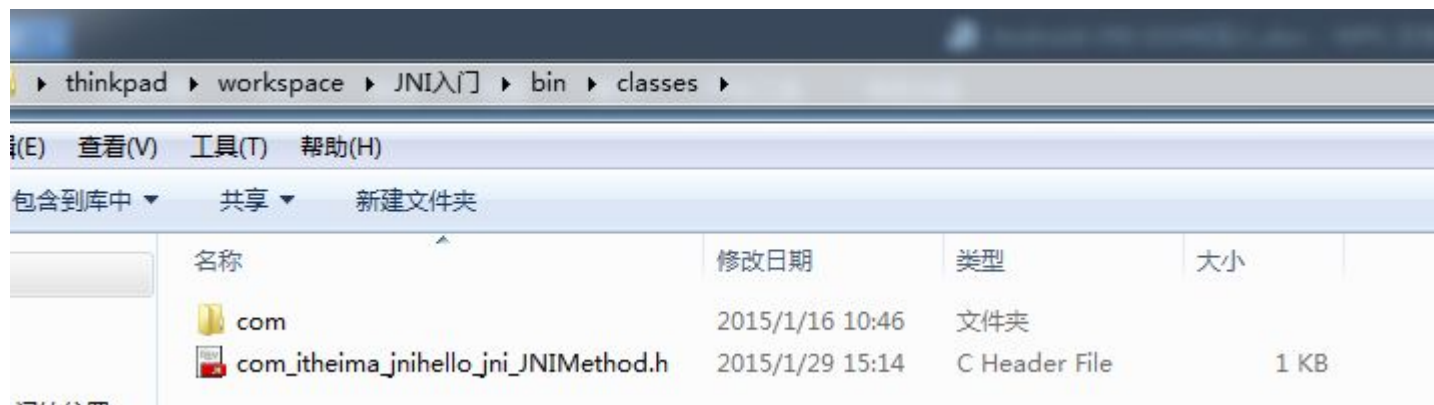
3

再次执行 javah 命令：javah com.itheima.jnihello.jni.JNIMethod

```
C:\Users\thinkpad\workspace\JNI入门\bin\classes>javah com.itheima.jnihello.jni.JNIMethod

C:\Users\thinkpad\workspace\JNI入门\bin\classes>_
```

这次发现没有报错，生成的头文件在如下位置：



4

打开生成的头文件，将生成的方法签名拷贝出来添加到我们的 hello.c 源文件中即可。

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_ithema_jnihello_jni_JNIMethod */

#ifndef _Included_com_ithema_jnihello_jni_JNIMethod
#define _Included_com_ithema_jnihello_jni_JNIMethod
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      com_ithema_jnihello_jni_JNIMethod
 * Method:     a_b_c_d
 * Signature:  ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_ithema_jnihello_jni_JNIMethod_a_1b_1_1c_1d
    (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

3. JNI 的值传递 (★★★)

本章我们通过一个案例来介绍几种常见的 JNI 值传递场景。

1

创建一个新的 Android 工程《JNI 值传递》。

2

在工程中创建 `com.itheima.jnipassdata.DataProvider` 类。在该类中定义三个 native 方法。代码清单如下：

```
public class DataProvider {  
    /**  
     * 模拟用 C 语言计算一些负责算法  
     * @param x  
     * @param y  
     * @return  
     */  
    public native int add(int x,int y);  
    /**  
     * 模拟用 C 语言加密字符串  
     * @param str  
     * @return  
     */  
    public native String encideString(String str);  
    /**  
     * 模拟用 C 语言进行一些图像算法  
     * @param colorArray  
     * @return  
     */  
    public native int[] changeColor(int[] colorArray);  
}
```

3

使用 javah 工具生成头文件

```
C:\Users\thinkpad>cd C:\Users\thinkpad\workspace\JNI值传递\bin\classes  
C:\Users\thinkpad\workspace\JNI值传递\bin\classes>javah com.itheima.jnipassdata.  
DataProvider  
C:\Users\thinkpad\workspace\JNI值传递\bin\classes>
```

生成的头文件在相对于命令行的当前目录下。

4

在工程中创建 jni 目录，在改 jni 目录中创建 Hello.c 文件，将上一步生成的方法拷贝到 Hello.c 文件中，并实现里面的方法。Hello.c 代码清单如下。

```
#include <stdio.h>
#include <malloc.h>
#include "jni.h"
//把 java 的 string 转化成 c 的字符串
char* Jstring2CStr(JNIEnv* env, jstring jstr)
{
    char* rtn = NULL;
    jclass clsstring = (*env)->FindClass(env,"java/lang/String");
    //String
    jstring strencode = (*env)->NewStringUTF(env,"GB2312"); // "gb2312"
    jmethodID mid = (*env)->GetMethodID(env,clsstring, "getBytes",
    "(Ljava/lang/String;)[B"); //getBytes(Str);
    jbyteArray barr=
    (jbyteArray)(*env)->CallObjectMethod(env,jstr,mid,strencode); //
    String .getBytes("GB2312");
    jsize alen = (*env)->GetArrayLength(env,barr);
    jbyte* ba = (*env)->GetByteArrayElements(env,barr,JNI_FALSE);
    if(alen > 0)
    {
        rtn = (char*)malloc(alen+1); // "\0"
        memcpy(rtn,ba,alen);
        rtn[alen]=0;
    }
    (*env)->ReleaseByteArrayElements(env,barr,ba,0); //释放内存空间
    return rtn;
}

jint Java_com_ithiema_jnipassdata_DataProvider_add
(JNIEnv* env, jobject obj, jint x, jint y){
    return x+y;
};

jstring Java_com_ithiema_jnipassdata_DataProvider_encodeString
(JNIEnv * env, jobject obj, jstring jstr){
    char* str = Jstring2CStr(env,jstr);
    char* hello = "hello";
    strcat(str,hello);
    //下面的两种写法是一样的效果
    //return ((*env)).NewStringUTF(env,str);
    return (*env)->NewStringUTF(env,str);
};

jintArray Java_com_ithiema_jnipassdata_DataProvider_changeColor
```



```
(JNIEnv * env, jobject obj, jintArray jarray){
    int size = (*env)->GetArrayLength(env,jarray);
    int* arr = (*env)->GetIntArrayElements(env,jarray,0);
    int i;
    for(i=0;i<size;i++){
        arr[i]+=10;
    }
    return jarray;
}
```

5

使用 NDK 工具将上面的 C 代码编译成动态库文件，首先得在工程的 jni 目录下添加 Android.mk 和 Application.mk 文件

6

在 MainActivity 类中调用 C 语言，MainActivity.java 代码清单如下。

```
public class MainActivity extends Activity {
    static{
        System.loadLibrary("hello");
    }
    private DataProvider provider = new DataProvider();
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void click1(View view){
        int x=1;
        int y=2;
        int result = provider.add(x, y);
        Toast.makeText(this, "计算结果为="+result, 0).show();
    }
    public void click2(View view){
        String str = "我是 Java 语言";
        String result = provider.encodeString(str);
        Toast.makeText(this, "字符串编码结果为="+result, 0).show();
    }
    public void click3(View view){
        int[] colorArray={1,2,3,4,5,6};
        int[] result = provider.changeColor(colorArray);
        Toast.makeText(this, "整型数组计算结果为="+Arrays.toString(result),
0).show();
    }
}
```

布局文件比较简单，就不再给出。运行上面代码，效果图如下：



3.1 让 C 语言输出 Log 日志

让我们的 C 语言也能在 LogCat 中输出一些信息是一个很简单但也很常见的实际需求。为了方便演示我们直接在本章节中创建的工程中演示如何让 C 语言打印 LogCat 日志。

1

在 Android.mk 中添加如下属性：

```
LOCAL_LDLIBS := -llog
```

2

在 C 源码头部引入如下头文件和宏定义

```
#include <android/log.h>
#define LOG_TAG "System.out"
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG, LOG_TAG, __VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)
```

3

在 C 代码中可以将 LOGD () 或者 LOGI () 函数当做 C 语言中的 printf () 函数使用。

这里我将 add 方法添加添加了一条日志输入。

```
jint Java_com_ithiema_jnipassdata_DataProvider_add
(JNIEnv* env, jobject obj, jint x, jint y){
    LOGI("x+y=%d",x+y);
    return x+y;
}
```

4

重新编译 hello.so 动态库，既然我们修改了 Android.mk 和 C 源码文件，那么一定要记得重新使用 NDK 编译 C 语言为动态库，否则日志不会输入而且也不报异常。

再次部署程序到模拟器，调用 add 方法，发现在 LogCat 中成功输出了如下信息：

Tag	Text
System.out	x+y=3

4. 案例-调用美图秀秀的动态库 (★★)

在美图秀秀 1.0 版本的时候程序员对其代码并没有加密以及反反编译等处理，因此我们可以将其 apk 反编译出来。

里面关于图形的核心算法都是通过 so 库来实现的，我们可以拿过来直接使用。

声明：本文档中使用的美图秀秀只用于学习交流 Android 技术，禁止用于其他目的！

美图秀秀 1.0 版本下载地址：<http://pan.baidu.com/s/1bnAjLGN>

1

将下载好的 mtxx.apk 进行反编译，将反编译好的资源作为原料备用。关于如何反编译美图秀秀请见本文档最后一章。

Tips：解压好的目录打开 lib 包，发现只有 armeabi 一个文件夹，说明**该 so 文件只能在 arm 架构的 CPU 上运行**。

2

创建一个新的 Android 工程《黑马美图秀秀》。将反编译好的 libmtime-jni.so 拷贝到工程的 libs->armeabi (该目录需要手动创建) 目录下。通过反编译的 jar 包发现美图秀秀的 jni 方法都定义在 JNI.java 类中，我们用 jd-gui 工具将反编译的 jar 包打开，找到 JNI.java，然后拷贝其中的 native 方法到我们的工程中。本人的工程目录结构如下所示：



Tips :我们使用了美图秀秀的 JNI 源码, 那么我们的 JNI.java 名字以及其所在的包名必须严格跟原美图秀秀保持一致, 不然程序从 so 动态库中是找不到目标方法的。原因很简单, 因为 C 中方法名是根据 JNI.java 中的方法全限定名生成的。

3 编写 activity_main.xml 布局文件, 布局文件清单如下:

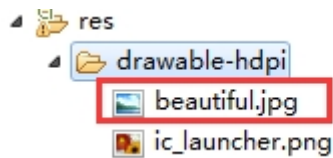
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/iv"
    />

    <Button
        android:layout_gravity="right"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Lomo"
        android:onClick="click"
    />

</LinearLayout>
```

4 在工程的 res->drawable-hdpi 目录下放入一张图片。



5

编写 MainActivity.java 代码，在改类中实现核心方法

```

public class MainActivity extends Activity {
    private ImageView iv;
    private Bitmap bitmap;
    static{
        System.loadLibrary("mtimage-jni");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        bitmap = BitmapFactory.decodeResource(getResources(),
R.drawable.beautiful);
        iv = (ImageView) findViewById(R.id.iv);
        iv.setImageBitmap(bitmap);
    }
    public void click(View view){
        JNI jni = new JNI();
        //获取 Bitmap 的宽和高
        int width = bitmap.getWidth();
        int height = bitmap.getHeight();
        //创建一个整型数组，存放每个像素点
        int[] pixels = new int[width*height];
        /**
         * 获取 bitmap 的像素数组
         * 第一个参数是 int[]
         * 第二个参数是第获取一个像素的偏移量，这里是 0，也就是从第一个像素开始获取
         * 第三个参数每行获取多少个像素，当然 bitmap 的宽度就是每行的像素个数
         * 第四个参数、第五个参数 分别是开始获取像素的 x、y 坐标，这个坐标是相对于 bitmap
本身来说的，因此是 0,0
         * 第六、七个参数分别是每行、每列获取像素的个数
         */
        bitmap.getPixels(pixels, 0, width, 0, 0, width, height);
        //调用美图秀秀 API
    }
}

```

```
jni.StyleLomoB(pixels, width, height);  
    //重新根据 pixels 数组创建一个 Bitmap 对象  
    Bitmap newBitmap = Bitmap.createBitmap(pixels, width, height,  
bitmap.getConfig());  
    iv.setImageBitmap(newBitmap);  
}  
}
```

6 运行上面打代码，效果如下：

◆ 1) 处理前



◆ 2) 处理后

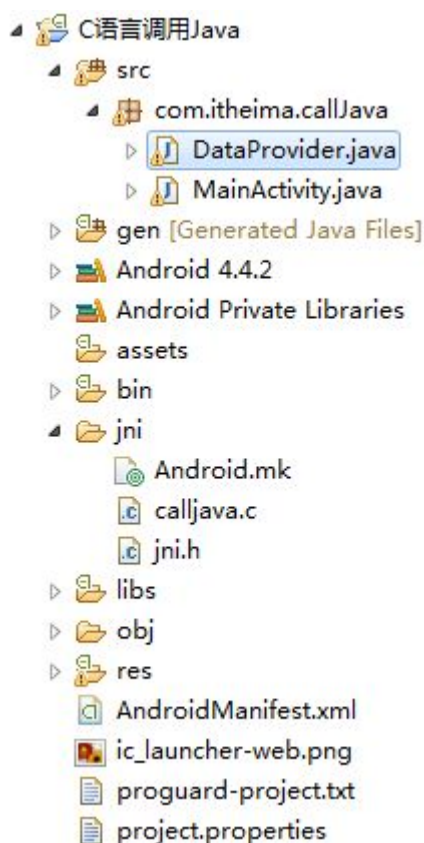


5. C 语言调用 Java 代码 (★★★)

之前我们学习的 JNI 都是 Java 代码调用 C 代码，本章节中演示 C 代码如何调用 Java 代码。

1

创建一个新的 Android 工程《C 语言调用 Java》，在工程中创建 jni 目录，在改目录下放置 Android.mk 和 jni.h 文件（从老工程中拷贝）。工程目录结构如下：



2

编写 DataProvider.java 文件，在该文件中编写 native 方法，DataProvider.java 代码清单如下：

```
public class DataProvider {  
    //C 代码调用该方法发送短信  
    public void methodInJava(){  
        System.out.println("我是 Java 中的方法，我被调用了");  
        SmsManager manager = SmsManager.getDefault();  
        manager.sendTextMessage("5556", null, "hello I'm from Java", null, null);  
    }  
    //C 代码调用该方法传递一个字符串，获取一个字符串  
    public String methodInJava2(String str){  
        return "hello:"+str;  
    }  
}
```

```

}
//C 代码调用一个空参返回值为 void 的实例方法
public void methodInJava3(){
    System.out.println("我是 Java 中的方法 3，我被调用了");
}
//C 语言调用静态方法
public static void methodInJava4(){
    System.out.println("我是静态方法。被调用了。");
}
//定义四个 native 方法，这些方法在 MainActivity 中被调用，这些方法在 C 代码中回调上
面的 Java 代码
public native void callCMethod();
public native String callCMethod2(String str);
public native void callCMethod3();
public native void callCMethod4();
}

```

3

在 jni 中创建 calljava.c 文件，在该文件中实现调用 java 的方法。代码清单如下。

```

#include <jni.h>
#include <stdio.h>
void Java_com_itheima_callJava_DataProvider_callCMethod(JNIEnv * env, jobject
obj){
    //类似 java 的反射，获取 java 对象
    jclass clazz = (*env)->FindClass(env,"com/itheima/callJava/DataProvider");
    //根据方法签名获取目标方法
    jmethodID methodID = (*env)->GetMethodID(env,clazz,"methodInJava","()V");
    //调用目标方法
    (*env)->CallVoidMethod(env,obj,methodID);
}
jstring Java_com_itheima_callJava_DataProvider_callCMethod2(JNIEnv * env,
jobject obj,jstring str){
    jclass clazz = (*env)->FindClass(env,"com/itheima/callJava/DataProvider");
    jmethodID methodID =
(*env)->GetMethodID(env,clazz,"methodInJava2","(Ljava/lang/String;)Ljava/lang
/String;");
    jstring jstr = (*env)->CallObjectMethod(env,obj,methodID,str);
    return jstr;
}
void Java_com_itheima_callJava_MainActivity_callCMethod3(JNIEnv * env, jobject
obj){
    jclass clazz = (*env)->FindClass(env,"com/itheima/callJava/DataProvider");

```



```
jmethodID methodID = (*env)->GetMethodID(env,clazz,"methodInJava3","()V");
jobject o = (*env)->AllocObject(env,clazz);
(*env)->CallVoidMethod(env,o,methodID);
}
void Java_com_itheima_callJava_DataProvider_callCMethod4(JNIEnv * env, jobject
obj){
    jclass clazz = (*env)->FindClass(env,"com/itheima/callJava/DataProvider");
    jmethodID methodID =
(*env)->GetStaticMethodID(env,clazz,"methodInJava4","()V");
    (*env)->CallStaticVoidMethod(env,clazz,methodID);
}
```

4

使用 NDK 工具，将 calljava.c 编译成动态库文件。（NDK 的使用在上一篇文档中有详细的介绍，这里就不再说明）

5

在 MainActivity.java 中调用 C 语言，代码清单如下：

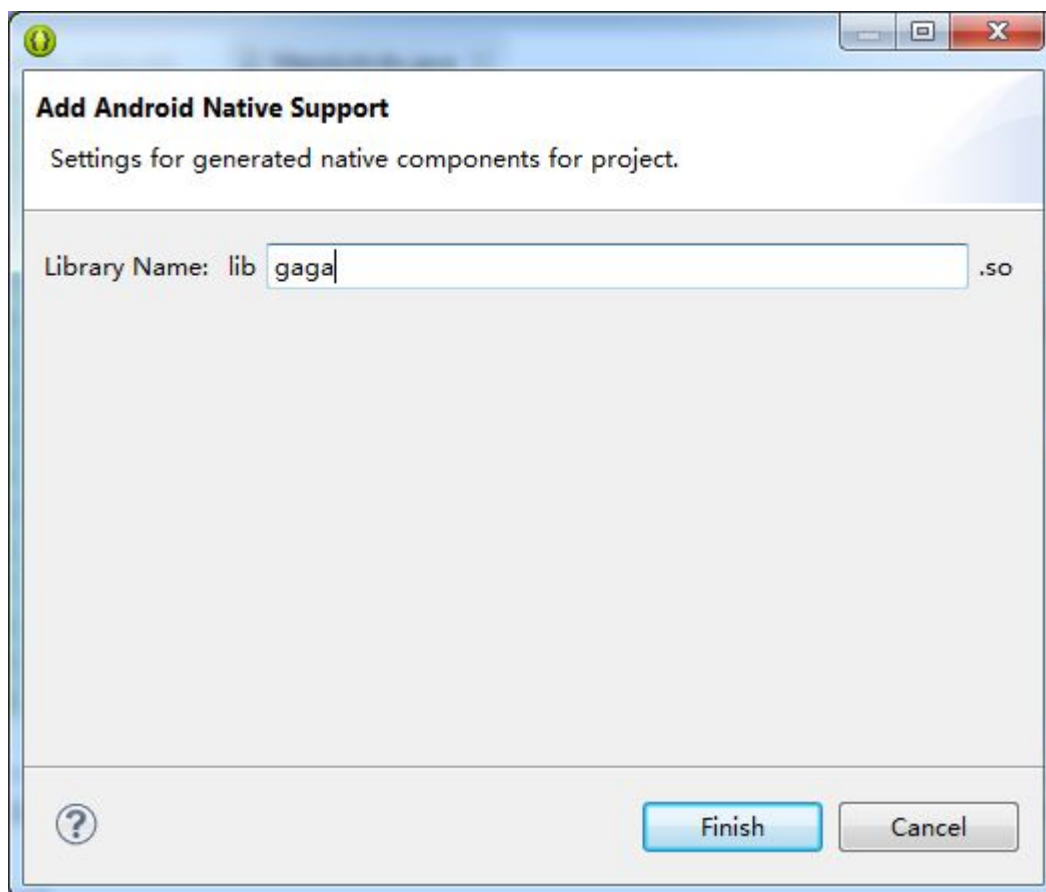
```
public class MainActivity extends Activity {
    static {
        System.loadLibrary("calljava");
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void click(View v) {
        new DataProvider().callCMethod();
    }
    public void click2(View view) {
        String result = new DataProvider().callCMethod2("ddd");
        Toast.makeText(this, result, 1).show();
    }
    public void click3(View v) {
        callCMethod3();
    }
    public void click4(View view) {
        new DataProvider().callCMethod4();
    }
    public native void callCMethod3();
}
```

布局文件比较简单这里就不再给出。

6. 用 C++ 实现 JNI (★★)

C++ 语言是面向对象的编程语言，源于 C 语言，部分语法通用。本章节中主要介绍如何使用 C++ 完成简单的 JNI 开发。

1 创建一个新 Android 工程《cpp 实现 jni》，创建 jni 包，在该包下创建 JNI.java 文件，在该类中写 naive 方法。因为我们是 C++ 项目，因此需要给当前工程添加 native Support。右键点击项目，在弹出的对话框中选择 Android Tools，然后选择 Add native Support，弹出如下对话框：



这个名字是 C++ 源文件名，因为没有实际的业务意义一次我们这里就随便输入一个了。

在工程中创建 com.itheima.cppjni.jni 包，在该包下创建 JNI.java，JNI.java 代码清单如下：

```
public class JNI {  
    public native void helloFormCPP();  
}
```

2

使用 javah 工具生成头文件，并将生成的头文件拷贝到工程根目录下的 jni 目录中。

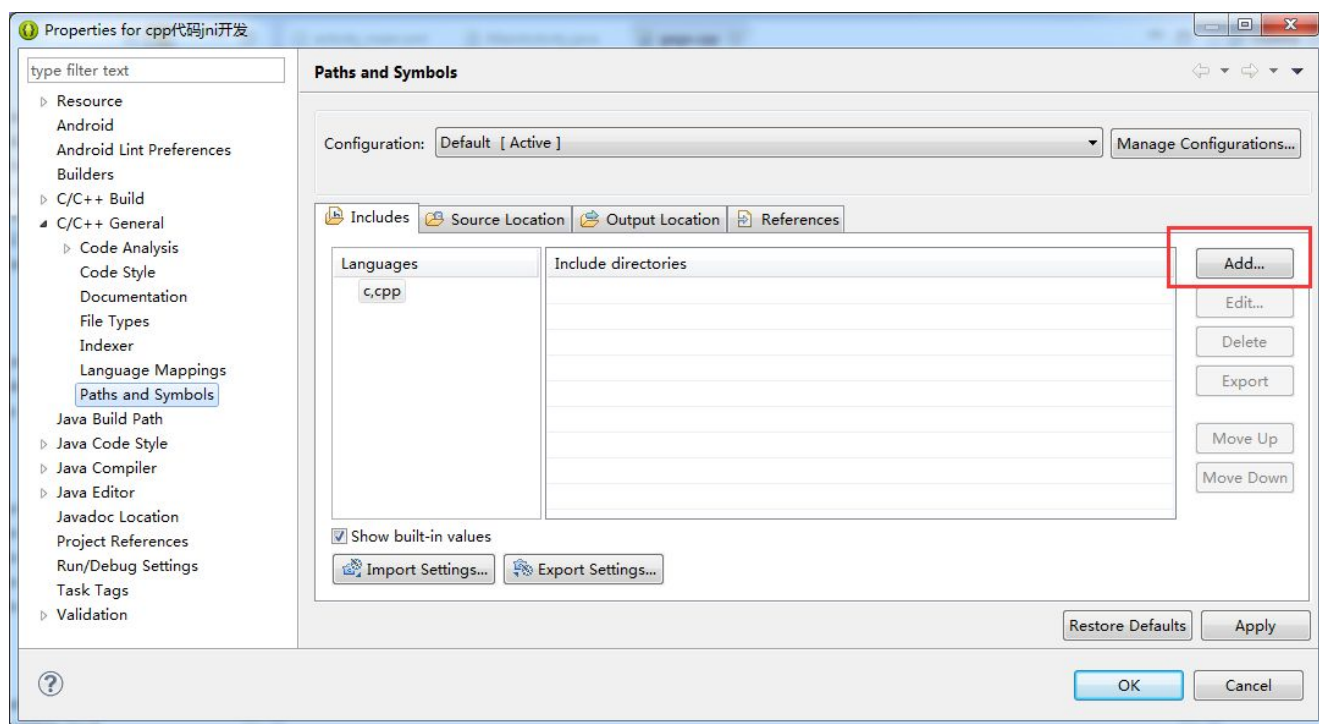
```
C:\Users\thinkpad\workspace\cpp代码jni开发\bin\classes>javah com.itheima.cppjni.  
jni.JNI
```

```
C:\Users\thinkpad\workspace\cpp代码jni开发\bin\classes>_
```

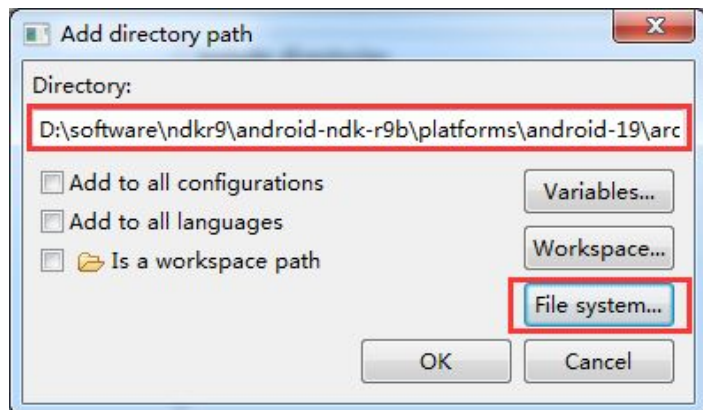
3

为了让编译器提示 C++ 语言，我们需要给工程添加 C++ 库。

右键点击工程，选择 Properties，在弹出的对话框中选择 C/C++ General->Paths And Symbols，弹出如下对话框：



点击图中红色框中的 Add 按钮，在弹出的对话框（如下图）中点击 File system, 然后选择 ndk 安装目录，选择如下目录：
D:\software\ndkr9\android-ndk-r9b\platforms\android-19\arch-arm\usr\include



然后点击 OK。

Tips：上面目录中红色的是本人的 ndk 所在目录，大家找到自己的 ndk 所在目录即可。然后随便选择一个 platform 即可，但是推荐大家选择一个高版本的平台。

4

在 gaga.cpp 中引入上面生成的头文件。同时编辑 gaga.cpp，代码清单如下：

```
#include <jni.h>
#include "com_itheima_cppjni_jni_JNI.h"
JNIEXPORT jstring JNICALL Java_com_itheima_cppjni_jni_JNI_helloFromCPP
(JNIEnv * env, jobject obj){
    return env->NewStringUTF("gaga from cpp");
};
```

5

编写 MainActivity.java,在该方法中加载动态库

```
public class MainActivity extends Activity {
    static{
        System.loadLibrary("gaga");
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void click(View view){
        JNI jni = new JNI();
        String helloFromCPP = jni.helloFromCPP();
        Toast.makeText(this, helloFromCPP, 1).show();
    }
}
```

6

将上面工程部署到一个 arm 架构的模拟器上。在部署的时候观察控制台，发现控制台输出如下信息

```
**** Build of configuration Default for project cpp 代码 jni 开发 ****
```

```
D:\software\ndkr9\android-ndk-r9b\ndk-build.cmd all
```

```
C:\Users\thinkpad\workspace\cpp 代码 jni 开发>rem This is a Windows cmd.exe script
used to invoke the NDK-specific GNU Make executable
```



```
C:\Users\thinkpad\workspace\cpp 代码 jni 开发>call
"D:\software\ndkr9\android-ndk-r9b\find-win-host.cmd" NDK_WIN_HOST
Android NDK: WARNING: APP_PLATFORM android-19 is larger than
android:minSdkVersion 8 in ./AndroidManifest.xml
[armeabi] Compile++ thumb: gaga <= gaga.cpp
[armeabi] StaticLibrary : libstdc++.a
[armeabi] SharedLibrary : libgaga.so
[armeabi] Install      : libgaga.so => libs/armeabi/libgaga.so

**** Build Finished ****
```

Tips：通过控制台，我们发现当我们的工程添加本地支持以后，当我们在部署的时候 eclipse 会自动的完成动态库的编译工作。而且我们还发现在生成动态之前先生成了 libstdc++.a 静态库然后才生成了动态库。

上面代码运行效果如下图所示：



7. 案例-锅炉压力监测 (★★★)

需求：硬件设备可以监测锅炉的压力，监测代码逻辑是用 C 语言编写。客户端用 java 代码每一秒调用一次 C 语言，

以获取锅炉的压力值，然后将锅炉的压力值以动态柱形图的形式显示在手机客户端。

Tips：分析上面的需求，我们需要使用 jni 技术让 Java 和 C 代码通信。在 C 语言端我们可以调用随机函数模拟锅炉压力的动态变化。在 Java 端，我们可以自定义一个 View 对象显示我们的锅炉压力。

1 创建一个 Android 工程《JNI 锅炉压力检测》，在该工程中创建 jni 目录，将 Android.mk、jni.h 从其他工程中拷贝到该目录下。

2 在 jni 目录下创建一个 C 源文件 pressure.c，代码清单如下：

```
#include <jni.h>
#include <stdio.h>
#include <stdlib.h>
//定义一个返回值为 int 型的函数，返回一个 100 以内的函数值 rand()是 C 语言的一个随机函数
int getPressure(){
    return rand()%100;
}
jint Java_com_itheima_jniPressure_MainActivity_getPressure(){
    return getPressure();
}
```

3 将 pressure.c 编译成动态库文件

4 在 MainActivity 的同一个包目录下创建一个自定义控件类 MyView 继承 View 类。

```
public class MyView extends View {
    int top = 100;
    public MyView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    public MyView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }
    Paint paint = new Paint();
    public MyView(Context context) {
        super(context);
        paint.setColor(Color.RED);
        paint.setTextSize(18);
    }
}
```

```

@Override
protected void onDraw(Canvas canvas) {
    /**
     * 绘制一个矩形区域
     * 第一、二、三、四个参数分别代表绘图区域离画布左、上、右、下的距离。这四个参数
     确定了矩形的大小也确定了
     */
    canvas.drawRect(60, top, 90, 150, paint);
    canvas.drawText("当前压力值: "+(150-top),60, 170, paint );

    super.onDraw(canvas);
}
//根据压力转换成绘图区域的 top 值
public void setPressure(int pressure){
    top = 150-pressure;
    if (pressure<30) {
        paint.setColor(Color.GREEN);
    }else if (pressure<60) {
        paint.setColor(Color.YELLOW);
    }else {
        paint.setColor(Color.RED);
    }
}
}

```

5

编写 MainActivity.java 代码，在该代码中实现 Java 端的核心方法。

```

public class MainActivity extends Activity {
    //加载动态库
    static{
        System.loadLibrary("pressure");
    }
    //声明一个 Timer 和 TimerTask,用于定时任务的处理
    private Timer timer;
    private TimerTask task;
    private MyView view;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //创建自定义控件对象
        view = new MyView(this);
        //将自定义控件对象作为显示对象，而没有使用布局文件
        setContentView(view);
    }
}

```

```
//创建定时任务
timer = new Timer();
task = new TimerTask() {

    @Override
    public void run() {
        view.setPressure(getPressure());
        //通过子线程通知控件重绘
        view.postInvalidate();
    }
};
//每 500 毫秒执行一次，第二个参数是延时执行，这里为 0 也就是第一次会立即执行
timer.schedule(task, 0, 500);
}
//调用 native 方法，获取压力值
public native int getPressure();
}
```

6

将上面工程部署到 arm 架构的模拟器上。运行结果如下图所示。

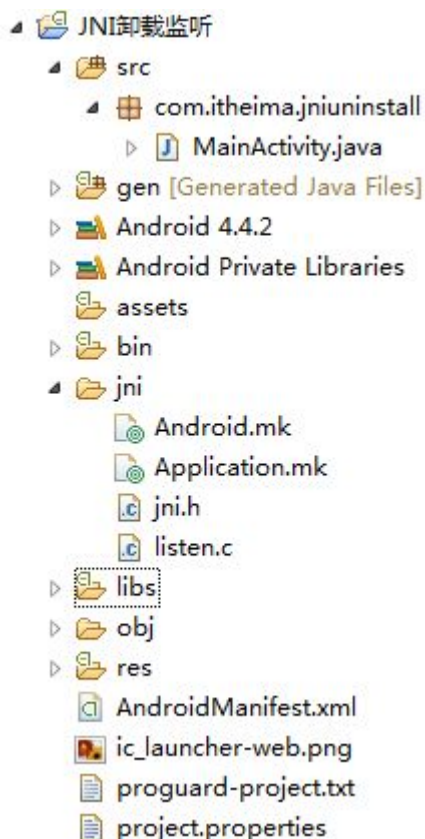


8. 案例-监听应用程序的卸载 (★★★)

需求：当我们的 apk 安装在 Android 手机上后，我们可以在其后台开启个 C 语言编写的死循环，C 语言编写的程序跟我们的应用不在同一个进程中，因此当我们的应用软件被卸载的时候，C 语言可以监测到。监测原理就是访问 `/data/data/{包名}` 文件是否存在，如果不存在显然是被删除了。

Tips：考虑到我们上面的案例跟这个案例使用的知识点差不多，都是使用 Java 语言调用 C 语言。因此这里就不再一步一步的演示如何创建工程。这里只给出核心的 C 语言代码。

本人工程目录结构如下所示：



这里只给出 listen.c 源文件清单，核心方法是这个文件实现的。在 MainActivity.java 中加载该动态库，并在

onCreate 方法中调用我们的 C 语言函数即可。listen.c 代码清单如下所示：

```
#include <jni.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/inotify.h>
#include <sys/stat.h>
#include <android/log.h>
#define LOG_TAG "System.out"
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG, LOG_TAG, __VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)
void Java_com_itheima_jniuninstall_MainActivity_listen(
    JNIEnv * env, jobject obj) {
    pid_t pid = fork(); //叉子
    if (pid == 0) { //当前是分叉出来的进程。
        int isStop = 1;
```

```
while (isStop) {
    //监视当前应用程序的包名文件夹是否存在如果不存在了就是被卸载了。
    FILE* file; //文件的指针
    file = fopen("/data/data/com.itheima.jniuninstall", "r");
    if (file == NULL) {
        //被卸载了。
        LOGI("uninstalled\n");
        //开启一个网页了。
        // int execlp(char *pathname, char *arg0, arg1, .., NULL);

        execlp("am", "am", "start", "-a", "android.intent.action.VIEW", "-d", "http://ww
w.baidu.com", NULL);
        isStop = 0;
    } else {
        //没有被卸载
        LOGI("haha huode henhao \n");
    }
    //让线程休眠 2 秒 在 C 语言中是秒为单位的不是毫秒
    sleep(2);
}
}
```

Tips：上面的 execlp 函数用于调用本地系统（Android 系统）命令打开一个浏览器，访问一个指定的 URL。但是经过我的测试发现在低版本模拟器（2.3）上该功能可是使用但是在高版本模拟器（4.3）上不可以使用。



9. apk 的反编译（★）

有一款叫安卓逆向助手软件反编译 apk 十分方便。这里给大家介绍的反编译方法就是基于这款软件的。

安卓逆向助手下载地址：<http://pan.baidu.com/s/1eQkvlvW>

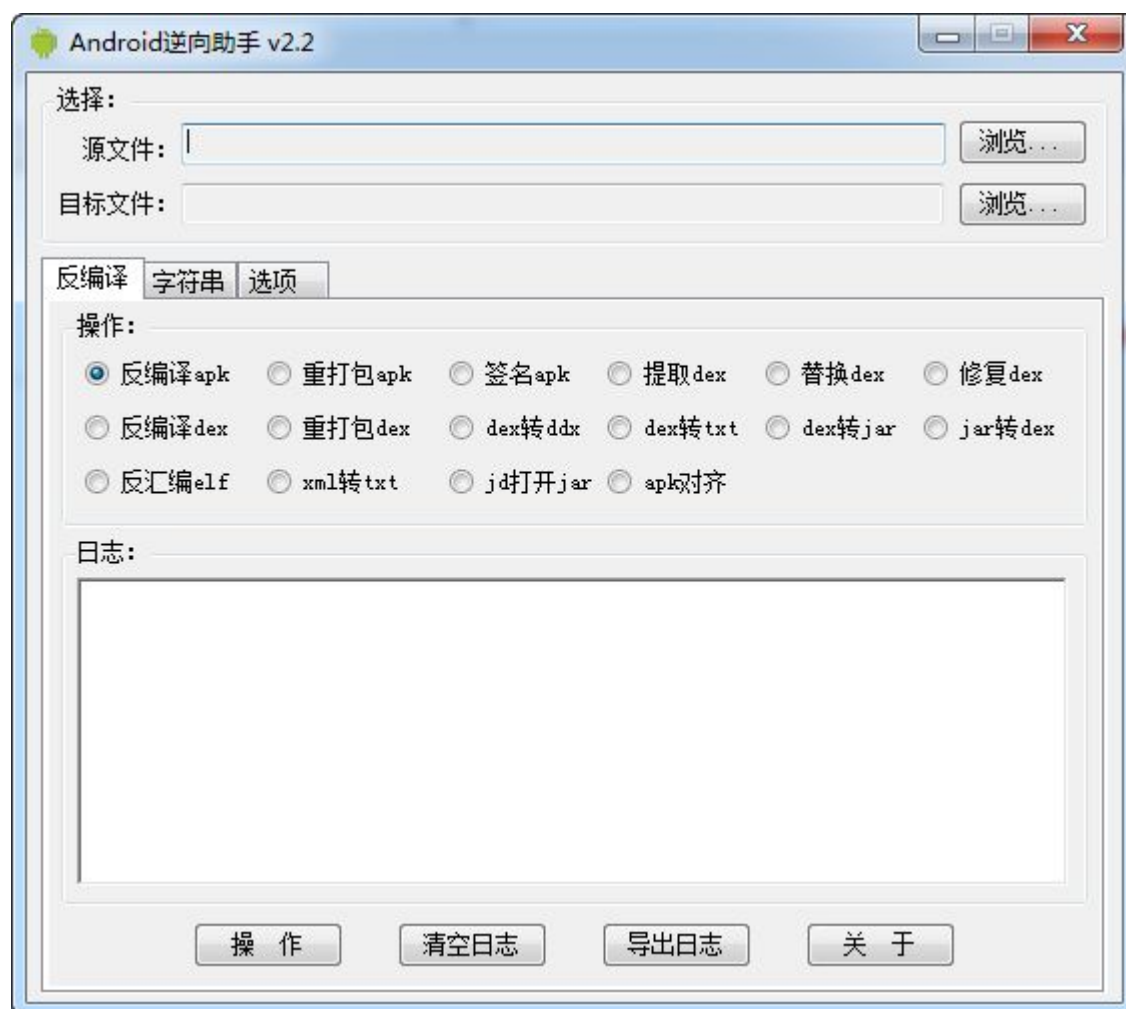
1

将下载好的 rar 包解压缩以后目录结构如下（内置的广告被我删除后的）

 lib	2014/5/19 17:01	文件夹	
 Android逆向助手.exe	2014/5/12 14:04	应用程序	207 KB

Tips：lib 目录存放都是用 java 写的核心反编译逻辑，必须跟 exe 文件放在同一个目录下。

打开 Android 逆向助手.exe，如下图所示：



2

选择源文件，并且选择（也是默认的选择）反编译 apk，我们找到 mtxx.apk 的路径，然后点击操作。

在 mtxx.apk 目录下生成了一个 mtxx 文件夹，打开该文件，目录结构如下图所示：

assets	2015/1/30 9:40	文件夹	
lib	2015/1/30 9:40	文件夹	
original	2015/1/30 9:40	文件夹	
res	2015/1/30 9:40	文件夹	
smali	2015/1/30 9:40	文件夹	
AndroidManifest.xml	2015/1/30 9:40	XML 文件	3 KB
apktool.yml	2015/1/30 9:40	YML 文件	1 KB

在上面操作后打开 lib 目录可以找到美图秀秀的动态库文件，但是我们还需要找到其 java 代码。显然美图秀秀用

smali 算法反编译了。那么我们接着下一步。



3

在 Android 逆向助手.exe 中打开源文件，选择提取 dex 点击执行。



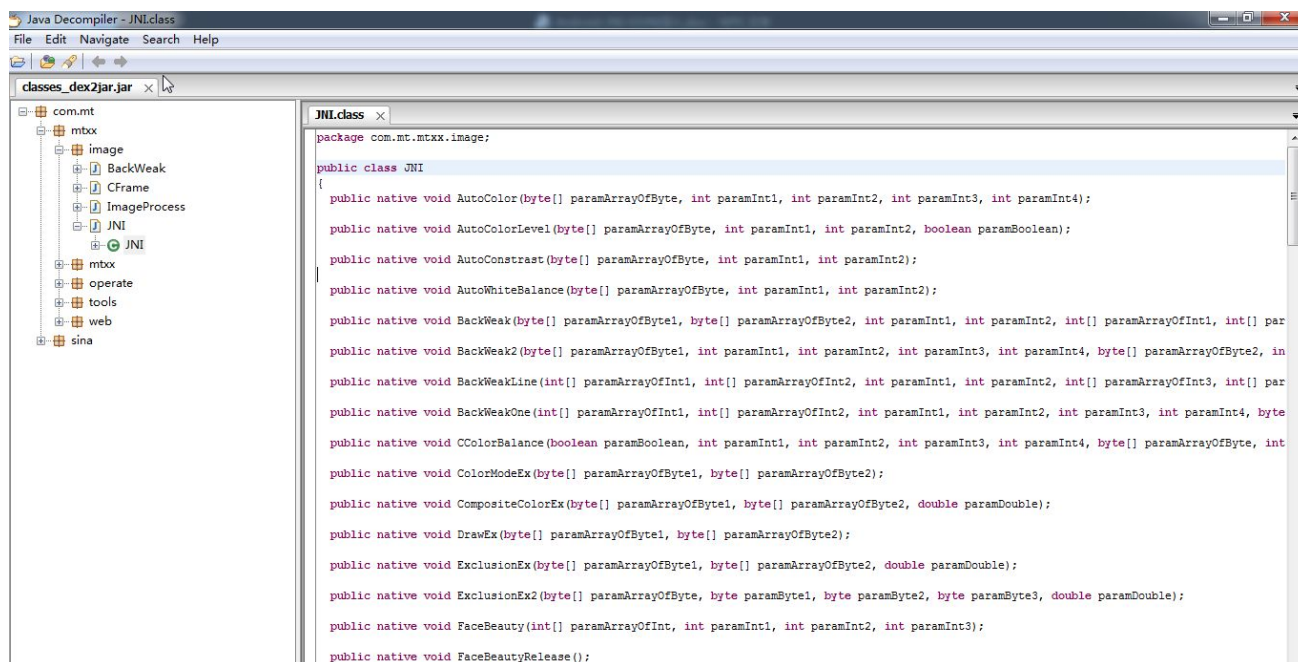
这时候在目标文件夹下生成了 dex 文件



4

最后在 Android 逆向助手.exe 中选择 dex 转 jar 选项。在源文件中选择上一步生成的 classes.dex 文件，然后点击执行（这个过程大概需要几秒的等待时间）。这时候该软件会自动将我们生成的 jar 文件用 jd-gui 工具打开。

打开效果如下所示：



至此，本文档完！

2015 年 1 月 31 日 星期六 1:37:39
北京市通州区星湖园温泉度假酒店