Android JNI - NDK

Nguyen Duc Tai
Email: tai.nguyenduc@gameloft.com
Skype: ndtaibk07t4

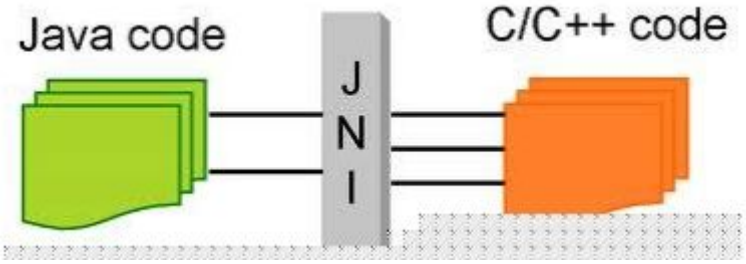
DAD Studio, 2018

CONTENT

I. Java Native Interface

Java Native Interface (JNI). The JNI is a native programming interface. It allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.

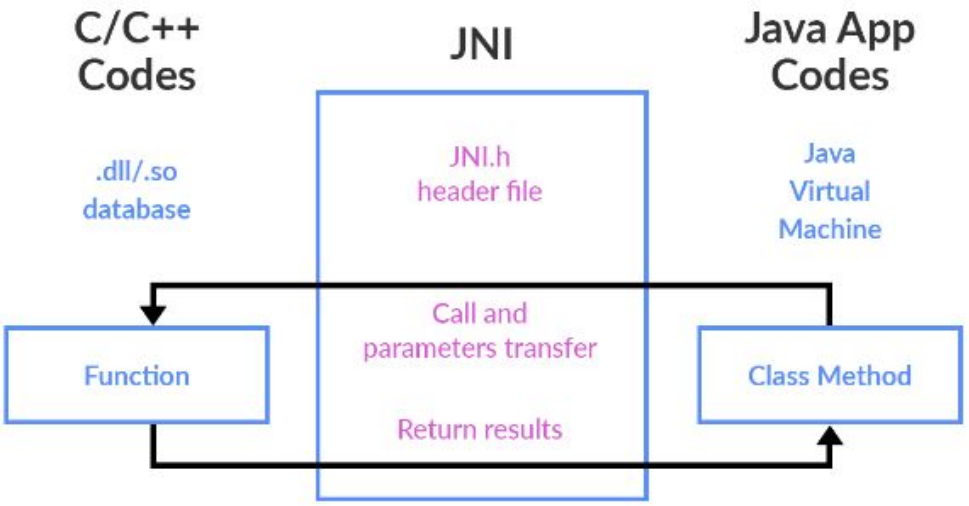
In other words, JNI is used to communicate between native code C/C++ and java and vice versa.



1. JNI Components

- **javah**: JDK tool that builds C-style header files from a given Java class that includes **native** methods. Adapts Java method signatures to native function prototypes
- **jni.h**: C/C++ header file included with the JDK that maps Java types to their native counterparts. javah automatically includes this file in the application header files

2. The interaction scheme



II. Android NDK

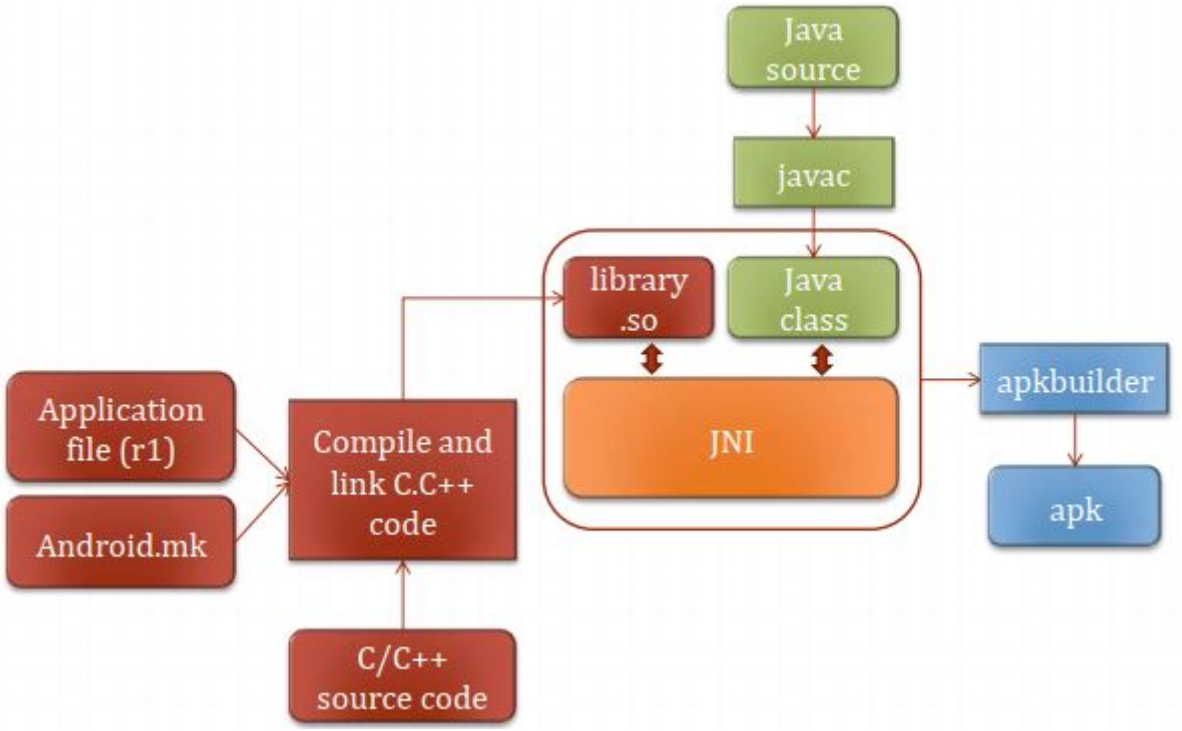
1. What is NDK?

- The Android NDK is a toolset that lets you implement parts of your app using native code languages such as C and C++. and Starting from android sdk 1.5
- Cross-compiler, linker, what you need to build for ARM, x86, etc
- NDK provides a way to bundle lib.so into your APK
  More information about Android NDK at https://developer.android.com/ndk/guides/

2. Why use Android NDK?

- Reuse C++ source code
- Improve performance of application
- Be able to access system without JVM (Java Virtual Machine)

3. JNI Inside NDK



III. JNI World

1. Loading Native Libraries

Native code is usually compiled into a shared library and loaded before the native methods can be loaded.

```
static {
   System.loadLibrary("native-lib");
}
```

2. Native method (Calling C/C++ methods)

Native method is method and declared with ***native*** keyword in java (either an instance method or a class method) whose implementation is written in another programming language such as C/C++ (this lesson mentions C/C++ programming language)

Example

```
public native String stringFromJNI();        //instance method
public native static void voidFromJNI();     //class method
```

In order to associate native method with code be written in C/C++. You will declare implement code in C/C++ side (.c or .cpp file) and must use `#include <jni.h>`

❖ **Systax**

```
JNIEXPORT return_data_type JNICALL Java_<packageName>_<className>_methodName(
        JNIEnv *env,
        [jobject instance] or [jclass type],
        parameter list);
```

All native method implementation accepts two standard parameters:

- **`JNIEnv *env:`** is a pointer that points to another pointer pointing to a function table (array of pointer). Each entry in this function table points to a JNI function. These are the functions we are going to use for type conversion
- The second argument is different depending on whether the native method is a static method or an instance method:
    - Instance method: It will be a **`jobject`** argument which is a reference to the object on which the method is invoked
    - Static method: It will be a **`jclass`** argument which is a reference to the class in which the method is define

Note: if native code (C/C++ side) be written in .cpp file , you must declare prototype method with **`extern "C"`** *or* **`extern "C" {....}`**

```
extern "C" JNIEXPORT return_data_type JNICALL
Java_<packageName>_<className>_methodName(
        JNIEnv *env, [jobject instance] or [jclass type], parameter list);
```

or

```
extern "C"
{
 JNIEXPORT return_data_type JNICALL Java_<packageName>_<className>_methodName(
        JNIEnv *env, [jobject instance] or [jclass type], parameter list);
}
```

Suppose **`MainActivity`** class in **`com.example.jni`** package declare native method:

Example 1**: `public native String stringFromJNI(int num);`**

```
extern "C" JNIEXPORT jstring JNICALL
Java_com_example_jni_MainActivity_stringFromJNI
(JNIEnv *env, jobject instance, jint n)
{

    // TODO
    std::string hello = "Hello from C++ " + std::to_string(n);
    return env->NewStringUTF(hello.c_str());
}
```

Example 2: `public static native int addFromJNI(int a, int b);`

```
extern "C" JNIEXPORT jint JNICALL
Java_com_example_jni_MainActivity_addFromJNI(
JNIEnv *env, jclass type, jint a, jint b)
{
    // TODO
    return a + b;
}
```

In example above, We can find some data types `jint, jstring,..` What are they? They are native C types that map Java types. Called type conversion

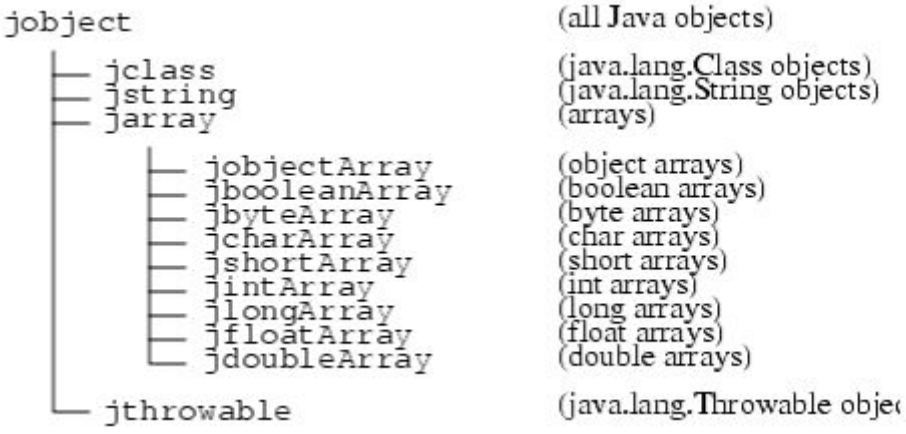
3. Type Conversion

❖ Primitive Types and Native Equivalents

Java Type	Native Type	Description
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits
void	void	N/A

❖ Reference Types

The JNI includes a number of reference types that correspond to different kinds of Java objects. JNI reference types are organized in the hierarchy shown in:



❖ String Conversion

JNIEnv *env will be the argument to use where we will find the type conversion methods. There are a lot of methods related to strings:

- java.lang.String to C string: GetStringChars (Unicode format), GetStringUTFChars (UTF-8 format)
- java.lang.String to C string: NewString (Unicode format), NewStringUTF (UTF-8 format)
- Release memory on C string: ReleaseStringChars, ReleaseStringUTFChars

Example: native method: `public native void sayHelloTo(String name);`

```
extern "C"
JNIEXPORT void JNICALL
Java_com_example_jni_MainActivity_sayHelloTo(JNIEnv *env, jobject instance,
jstring name) {
   const char *pName = env->GetStringUTFChars(name, 0);     //1

   std::cout << "Hello " << name << std::endl;

   env->ReleaseStringUTFChars(name, pName);                 //2
}
```

`//1` : returns a pointer to an array of chars representing the string in UTF-8 encoding (without making a copy)

`//2:` When we are not making a copy of the string, calling ReleaseStringUTFChars prevents the memory area used by the string to stay "pinned". If the data was copied, we need to call ReleaseStringUTFChars to free the memory which is not used anymore

Another example about String above:

```
extern "C" JNIEXPORT jstring JNICALL
Java_com_example_jni_MainActivity_stringFromJNI
(JNIEnv *env, jobject instance, jint n)
{
   // TODO
   std::string hello = "Hello from C++ " + std::to_string(n);
   return env->NewStringUTF(hello.c_str());
}
```

❖ Array Conversion

We just focus on primitive arrays only since they are different from objects arrays in JNI. Arrays are represented in JNI by the jarray reference type and its "subtypes" such as jintArray ⇒ A jarray is not a C array!

Again we will use the JNIEnv *env parameter to access the type conversion methods:

- Get<DataType>ArrayRegion: copies the contents of primitive arrays to a preallocated C buffer. Good to use when the size of the array is known
- Get<DataType>ArrayElements: gets a pointer to the content of the primitive array

- New<DataType>Array: to create an array specifying a length
- Release<DataType>ArrayElements: release arrays

We are going to see an example of how to read a Java primitive array in the native code - Sum a int array: `public native int sumArray(int []array);`

```
extern "C"
JNIEXPORT jint JNICALL
Java_com_example_jni_MainActivity_sumArray(JNIEnv *env, jobject instance,
jintArray array) {
    jint *pArray = env->GetIntArrayElements(array, NULL);

    jint sum = 0;
    jint length = env->GetArrayLength(array);
    for(int i = 0; i < length; i++)
    {
        sum += *(pArray + i);
    }
    env->ReleaseIntArrayElements(array, pArray, 0);
    return sum;
}
```

4. Calling Java methods

❖ **JNI_OnLoad method:** invoke when the native library loaded. It is the right and safe place to register the native methods before their execution.

```
JNIEnv *g_env;

JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    if (vm->GetEnv(reinterpret_cast<void**>(&g_env), JNI_VERSION_1_6)!= JNI_OK)
    {
        return -1;
    }
    return JNI_VERSION_1_6;
}
```

The purpose of method above is initialization `g_env` variable

In order to call a java method from native code (C/C++). You will do 3 steps below:

- **Find class**: find class that contains the method.
  - Non-static method, you find method through instance of class

    ```
    jclass _class = g_env->GetObjectClass(object);
    ```

  - Static method, you find method through class name and the package contains itself

    ```
    jclass _class = g_env->FindClass("package_name/class_name");
    ```

- **Get Method:** get the method from class that found above. It need 3 parameters: class, method name and parameter type with return type

- Non-static method

```
jmethodID methodID = g_env->GetMethodID(_class,
                        "method_name",
                        "(parameter type list)return_data_type");
```

- Static method, you find method through class name and the package contains itself

```
jmethodID methodID = g_env->GetStaticMethodID(_class,
                        "method_name",
                        "(parameter type list)return_data_type");
```

- **Call method:** call method need parameters: class or instance, methodId and parameter list if have. Depend on method type (static, non-static) and return data type that you must choose a function correctly. You can find at original link at: https://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/functions.html

  I will show you some methods:

Non-static method	Static method
GetMethodID	GetStaticMethodID
CallObjectMethod	CallStaticObjectMethod
CallByteMethod	CallStaticByteMethod
CallIntMethod	CallStaticIntMethod
CallLongMethod	CallStaticLongMethod
CallFloatMethod	CallStaticFloatMethod
CallDoubleMethod	CallStaticDoubleMethod
CallVoidMethod	CallStaticVoidMethod
...	...

❖ **Type Signatures**

The JNI uses the Java VM's representation of type signatures.

Java Type	Type Signatures
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
void	V
String	Ljava/lang/String;
type[]	[type

For example, the Java method: long func (int n, String s, int[] arr);

has the following type signature: (ILjava/lang/String;[I)J

After that, I will give some example about native code call java method (both static and non-static method). Suppose **`MainActivity`** class in **`com.example.jni`**

```
public void nonStaticMethod(String name){
   System.out.println("nonStatic method: " + name );
}
```

`nonStaticMethod:`

- is non-static method
- return data type: void
- Parameter: String

so you can call from native code:

```
void CallNonStaticMethodFromJava(jobject object, jstring name)
{
   jclass _class = g_env->GetObjectClass(object);
   jmethodID methodID = g_env->GetMethodID(_class, "nonStaticMethod",
"(Ljava/lang/String;)V");
   g_env->CallVoidMethod(object, methodID, name);
}
```

To use `CallNonStaticMethodFromJava`, you have to call it inside native method, that makes sure have parameter Example:

```
extern "C"
JNIEXPORT void JNICALL
Java_com_example_jni_MainActivity_sayHelloTo(JNIEnv *env, jobject instance,
jstring name) {
   CallNonStaticMethodFromJava(instance, name);
}
```

Other example:

```
public static String staticMethod(String name){
   System.out.println("staticMethod: " + name);
   return "Hello JNI";
}
```

staticMethod:

- is static method
- return data type: String
- Parameter: String

so you can call from native code:

```
void CallStaticMethodFromJava(jstring name)
{
   jclass _class = g_env->FindClass("com/example/jni/MainActivity");
   jmethodID methodID = g_env->GetStaticMethodID(_class, "staticMethod",
"(Ljava/lang/String;)Ljava/lang/String;");
   jstring value = (jstring)g_env->CallStaticObjectMethod(_class, methodID,
name);
   std::cout << value << std::endl;
}
```

5. Accessing Java member variables

The JNI provides functions that native methods use to get and set Java member variables. You can get and set both instance and class member variables. Similar to accessing methods, you use one set of JNI functions to access instance member variables and another set of JNI functions to access class member variables.

Our example: give Person class

```
package com.example.jni;

public class Person {
   private String name;
   private static int sCount;

   Person(String name){
       this.name = name;
       sCount++;
   }

   public String getName(){
       return name;
   }
```

```
    public static int getCount(){
        return sCount;
    }
    //We will change value of name and sCount in accessFields method
    public native void accessFields();
}
```

And use:

```
Person person = new Person("Tran Van Nam");
System.out.println("ame: " + person.getName() + " " + Person.getCount());
person.accessFields();
System.out.println("Name: " + person.getName() + " " + Person.getCount());
```

We will write native code of `accessFields` method to change value of variables in Person class

To get and set Java member variables from a native language method, you must obtain the identifier for that member variable from its class, name, and type signature.

- **Get Field ID**: take 3 parameters: instance or class, variable name, type signature. Example

```
//for instance variable
jfieldID fieldId = env->GetFieldID(_class, "name", "Ljava/lang/String;");

//for class variable
jfieldID  fieldId = env->GetStaticFieldID(_class, "sCount", "I");
```

- **Get Value:** get value of variable, it takes 2 parameters: instance or class and fieldID. Example

```
//for instance variable
jstring  name = (jstring)env->GetObjectField(instance, fieldId);

//for class variable
jint count = env->GetStaticIntField(_class, fieldId);
```

- **Set Value**: set new value for variable, it takes 3 parameters: instance or class, fieldID and new value. Example:

```
//for instance variable
env->SetObjectField(instance, fieldId, env->NewStringUTF("Nguyen Van Tan"));

//for class variable
env->SetStaticIntField(_class, fieldId, 100);
```

You can find more methods to Get/Set for variable at original link at:

https://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/functions.html

I will show you some methods:

Instance variable	Class variable
GetFieldID	GetStaticFieldID
GetObjectField	GetStaticObjectField
GetBooleanField	GetStaticBooleanField
GetByteField	GetStaticByteField
GetCharField	GetStaticCharField
GetShortField	GetStaticShortField
...	...
SetObjectField	SetStaticObjectField
SetBooleanField	SetStaticBooleanField
SetByteField	SetStaticByteField
SetCharField	SetStaticCharField
SetShortField	SetStaticShortField
...	...

And here is implementation of native method: `accessFields`

```
extern "C"
JNIEXPORT void JNICALL
Java_com_example_jni_Person_accessFields(JNIEnv *env, jobject instance) {
   jclass _class = env->GetObjectClass(instance);
   jfieldID fieldId = env->GetFieldID(_class, "name", "Ljava/lang/String;");
   //Set and Get instance variable
   //get name
   jstring  name = (jstring)env->GetObjectField(instance, fieldId);
   const char* pName = env->GetStringUTFChars(name, NULL);
   std::cout << "Get name field from native code: "  << pName << std::endl;
   //set value for name field
   env->SetObjectField(instance, fieldId, env->NewStringUTF("Nguyen Van Tan"));
   //Set and Get class variable
   fieldId = env->GetStaticFieldID(_class, "sCount", "I");
   jint count = env->GetStaticIntField(_class, fieldId);
   std::cout << "Get count field from native code: "  << count << std::endl;
   env->SetStaticIntField(_class, fieldId, 100);
}
```

About Interacting with Java from the Native Side, please check more at:

http://leo.ugr.es/elvira/devel/Tutorial/Java/native1.1/implementing/index.html