MICROOH 麦可网

Android-从程序员到架构师之路

出品人：Sundy
讲师：高焕堂（台湾）

http://www.microoh.com

C03_c

JNI：从C调用Java函数 (c)

By 高煥堂

3、How-to:
从C调用Java函数

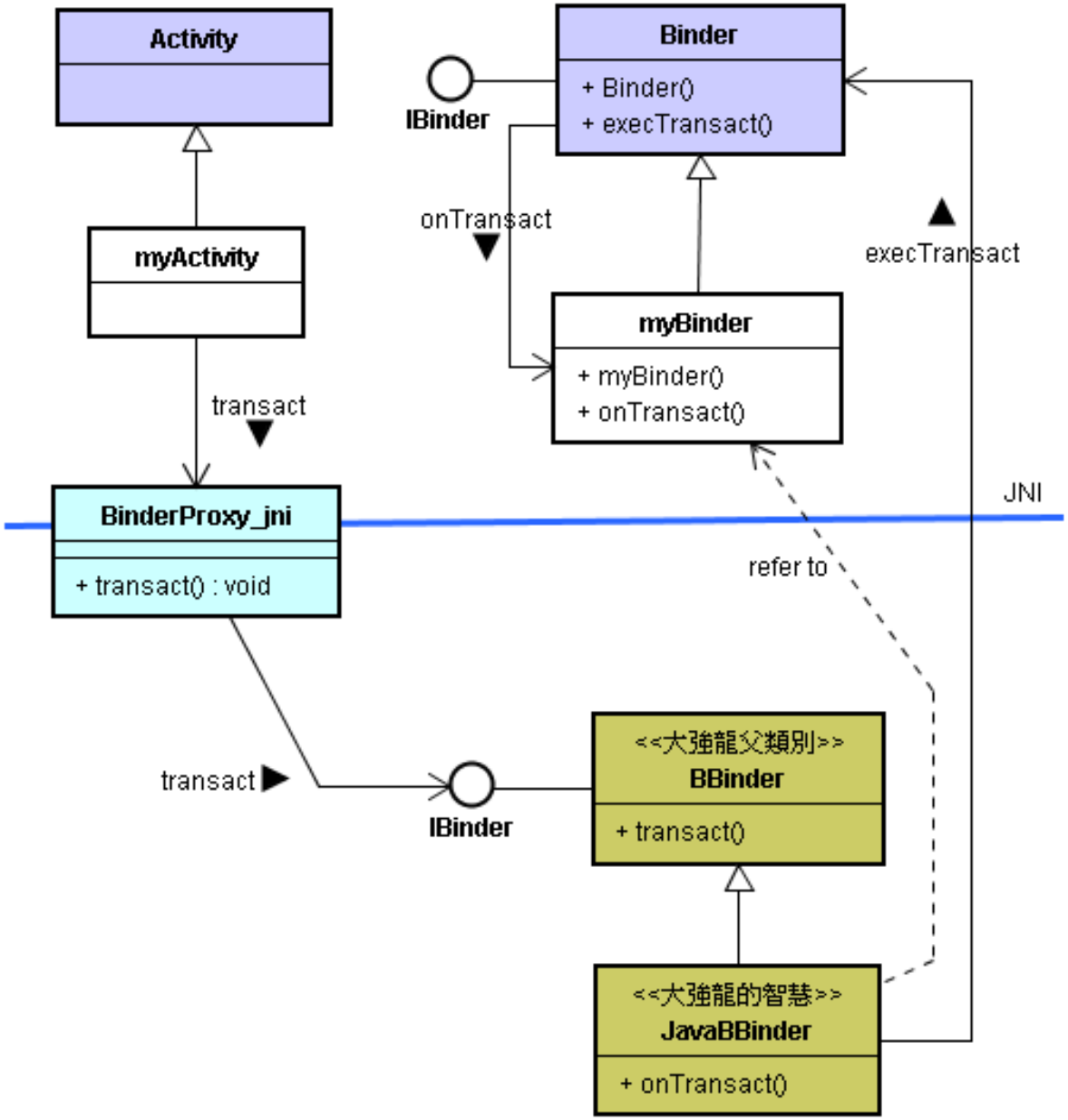
控制点的基本特性

- 如果控制点摆在本地C层，就会常常

1. 从本地C函数去调用Java函数；
2. 从本地C函数去存取Java层对象的属性值；
3. 从本地C函数去创建Java层的对象。

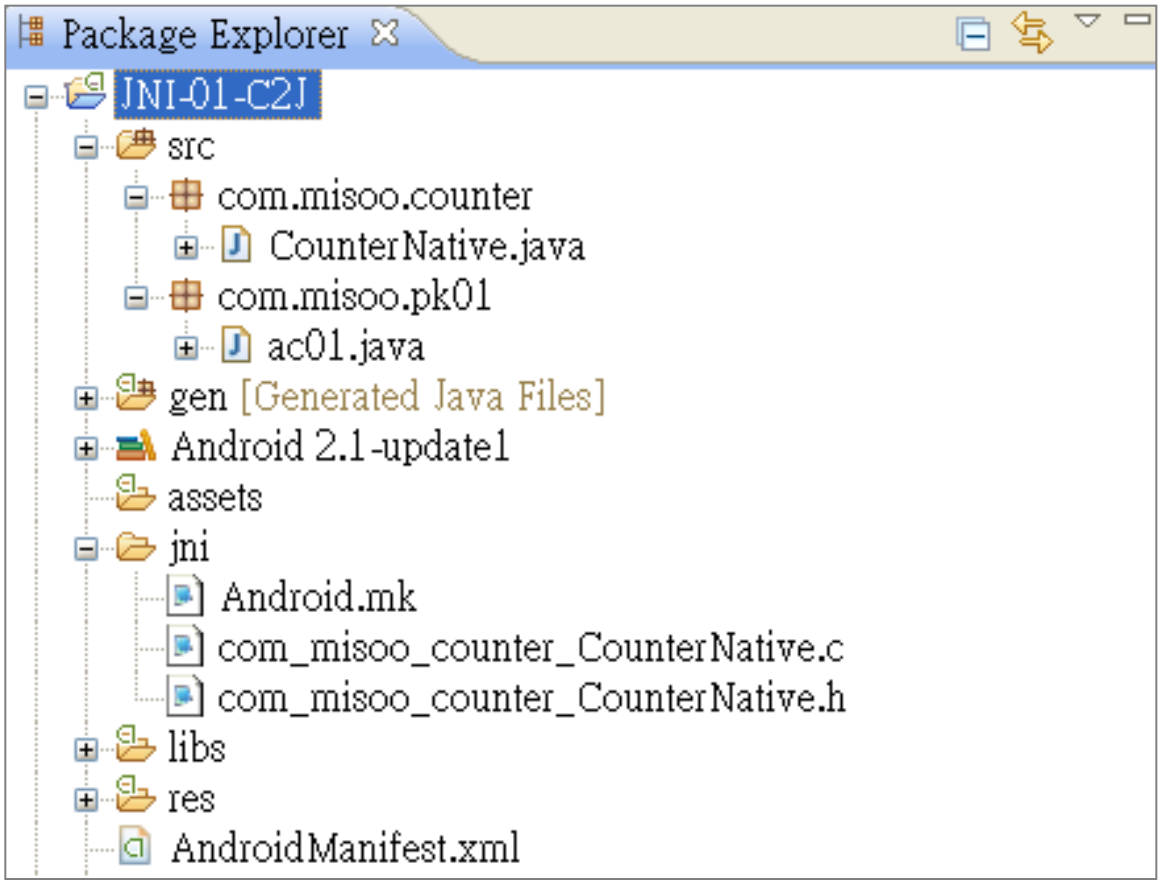
从C调用Java函数

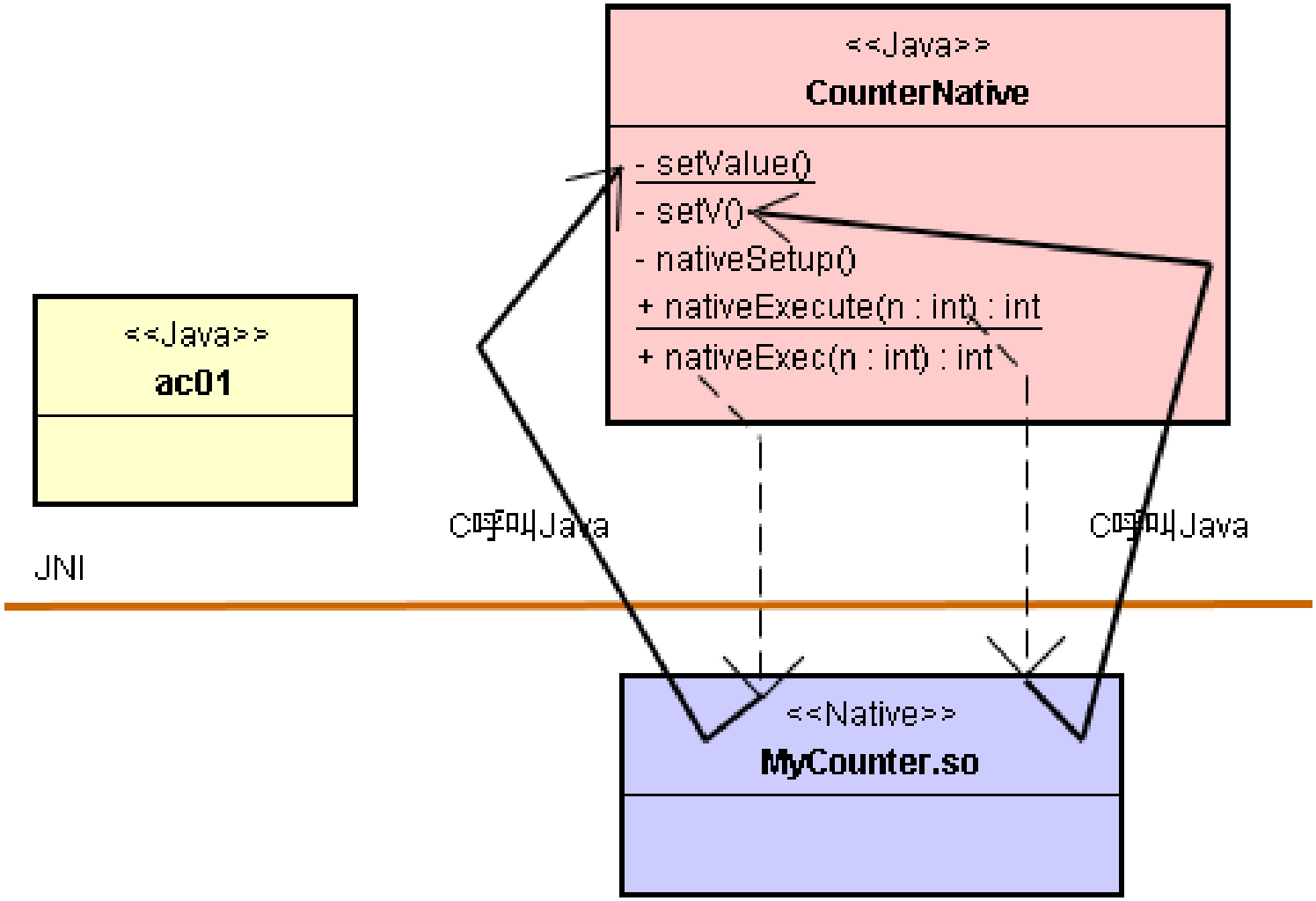
- 关于JNI，大家都知道如何从Java调用C函数。然而，在Android里，反而由C呼叫Java的情形才更具关键性。例如，Activity的跨进程沟通如下：


Activity
myActivity
transact
BinderProxy_jni
+ transact() : void
transact
IBinder
Binder
+ Binder()
+ execTransact()
IBinder
onTransact
myBinder
+ myBinder()
+ onTransact()
execTransact
JNI
refer to
<<大強龍父類別>>
BBinder
+ transact()
<<大強龍的智慧>>
JavaBBinder
+ onTransact()

- 当App里的Activity透过IBinder接口来与Service进行IPC沟通时，事实上是由Java层的Activity调用C/C++模块去进行IPC沟通，再由C模块调用Java层的Service。
- 所以，Java与C函数的双向调用都是Android平台的重要机制。

举例说明


Package Explorer
JNI-01-C2J
src
com.misoo.counter
CounterNative.java
com.misoo.pk01
ac01.java
gen [Generated Java Files]
Android 2.1-update1
assets
jni
Android.mk
com_misoo_counter_CounterNative.c
com_misoo_counter_CounterNative.h
libs
res
AndroidManifest.xml


<<Java>>
CounterNative
- setValue()
- setV()
- nativeSetup()
+ nativeExecute(n : int) : int
+ nativeExec(n : int) : int
<<Java>>
ac01
C呼叫Java
C呼叫Java
JNI
<<Native>>
MyCounter.so

- 在CounterNative类别里，有3个本地函数：静态(static)的nativeExecute()和一般的nativeSetup()及nativeExec()。
- 其中，静态nativeExecute()会调用Java层的一般的setV()函数；而一般的nativeExec()会调用Java层的静态setValue()函数。

```
// ac01.java
// .......
public class ac01 extends Activity implements OnClickListener {
   private CounterNative cn;

  @Override public void onCreate(Bundle savedInstanceState){
    //.......
    cn = new CounterNative();
  }
@Override public void onClick(View v) {
            switch(v.getId()){
            case 101:
              cn.nativeExec(10);            break;
            case 102:
              CounterNative.nativeExecute(11);          break;
            case 103:  finish();
              break;
}}}
```

- 指令：cn = **new** CounterNative();

- 其调用CounterNative()建构函数。执行到nativeSetup()函数，转而调用本地C函数：

  com_misoo_counter_CounterNative_nativeSetup()

- 这个函数只负责将m_class、m_object、m_static_mid和m_mid储存在C模块的静态区域里而已。

- 执行指令：　cn.nativeExec(10);

- 就呼叫C函数：nativeExec()，计算出sum值之后，透过VM的CallVoidMethod()函数而调用到目前Java对象的setValue()函数，把sum值传入Java层，并显示出来。

```
// CounterNative.java
// .........
public class CounterNative {
    private static Handler h;
    static {
                System.loadLibrary("MyCounter");
            }
    public CounterNative(){
        h = new Handler(){
            public void handleMessage(Message msg) {
                        ac01.ref.setTitle(msg.obj.toString());
        }};
        nativeSetup();
    }
    private static void setValue(int value){
        String str = "Value(static) = " + String.valueOf(value);
        Message m = h.obtainMessage(1, 1, 1, str);
        h.sendMessage(m);
    }
```

```
        private void setV(int value){
            String str = "Value = " + String.valueOf(value);
            Message m = h.obtainMessage(1, 1, 1, str);
            h.sendMessage(m);
         }
        private native void nativeSetup();
        public native static void nativeExecute(int n);
        public native void nativeExec(int n);
}
```

- ac01调用CounterNative类的建构函数，此函数诞生了一个Handler对象，并且调用本地的nativeSetup()函数。
- 随后，ac01将调用静态的nativeExecute()函数，此函数则反过来调用Java层一般的setV()函数。
- 接着，ac01调用一般的nativeExec()函数，此函数则反过来呼叫Java层的静态setValue()函数。

留意：目前该行代码正由那一个线程所执行

- 请记得，在学习Android时，从第一秒钟就持着优雅的素养：对于每一行代码，都必须能准确而正确地说出来，目前该行代码正由那一个线程(Thread)所执行的。

```
/*  com.misoo.counter.CounterNative.c  */
#include "com_misoo_counter_CounterNative.h"
jclass    m_class;
jobject   m_object;
jmethodID m_mid_static, m_mid;

JNIEXPORT void JNICALL
Java_com_misoo_counter_CounterNative_nativeSetup
  (JNIEnv *env, jobject thiz) {
   jclass clazz = (*env)->GetObjectClass(env, thiz);
   m_class = (jclass)(*env)->NewGlobalRef(env, clazz);
   m_object = (jobject)(*env)->NewGlobalRef(env, thiz);
   m_mid_static
        = (*env)->GetStaticMethodID(env, m_class, "setValue", "(I)V");
   m_mid = (*env)->GetMethodID(env, m_class, "setV", "(I)V");
   return;
 }
```

```
JNIEXPORT void JNICALL
Java_com_misoo_counter_CounterNative_nativeExecute
  (JNIEnv *env, jclass clazz, jint n) {
    int i, sum = 0;
    for(i=0; i<=n; i++)   sum+=i;
    (*env)->CallVoidMethod(env, m_object, m_mid, sum);
    return;
 }

JNIEXPORT void JNICALL
Java_com_misoo_counter_CounterNative_nativeExec
  (JNIEnv *env, jobject thiz, jint n)  {
    int i, sum = 0;
    for(i=0; i<=n; i++)  sum+=i;
    (*env)->CallStaticVoidMethod(env, m_class, m_mid_static, sum);
    return;
 }
```

说明nativeSetup()函数的内容

- 上述的nativeSetup()函数之定义：

```
JNIEXPORT void JNICALL
Java_com_misoo_counter_CounterNative_nativeSetup
        (JNIEnv *env, jobject thiz)  {
            //……..
    }
```

第2個參數是什麼?

- 其中的第2个参数thiz就是Java层目前对象的参考(Reference)。所谓目前对象就是正在调用此本地函数的Java层对象。例如，在此范例里，就是CounterNative类的对象参考。

- 指令：

  ```
  jclass clazz = (*env)->GetObjectClass(env, thiz);
  ```

- 向VM(Virtual Machine)询问这thiz所参考对象的类(即CounterNative类别)。

- 由于这class是这本地函数的区域(Local)变量，当此函数执行完毕后，这个class变量及其所参考的值都会被删除。因此，使用指令：

```
m_class = (jclass)(*env)->NewGlobalRef(env, clazz);
```

- 来将区域型的class参考转换为全域(Global)型的参考，并将此全域参考存入到这本地C模块的全域变数m_class里。

- 如此，当函数执行完毕后，这个m_class变量及其所参考的值都不会被删除掉。同理，thiz也是区域变量，函数执行完毕，这个thiz及其值都会被删除。因此，使用指令：

```
m_object = (jobject)(*env)->NewGlobalRef(env, thiz);
```

- 将区域型的class参考转换为全域(Global)型的参考，并将此全域参考存入到这本地C模块的全域变数m_object里。

- 接着，指令：

```
m_mid_static = (*env)->GetStaticMethodID(env,
                    m_class, "setValue", "(I)V");
```

- 这要求VM去取得m_class所参考的类(就是CounterNative类)的setValue()函数的ID值。并将此ID值存入到这本地C模块的全域变数m_mid_static里。

- 同理，指令：

```
m_mid = (*env)->GetMethodID(env, m_class,
                            "setV", "(I)V");
```

- 这找到CounterNative类的setV()函数的ID，并将此ID值存入到这本地C模块的全域变数m_mid里。

- 由于m_class和m_object两者都是参考(Reference)，其必须透过VM的NewGlobalRef()来转换出全域性的参考。至于m_static_mid和m_mid则是一般的整数值，直接储存于静态变量里即可了。

说明nativeExecute()和nativeExec()函数的内容

- 于此，这nativeSetup()函数已经将m_class、m_object、m_static_mid和m_mid储存妥当了，准备好让后续调用nativeExecute()和nativeExec()函数时能使用之。

- 例如：

```
JNIEXPORT void JNICALL
Java_com_misoo_counter_CounterNative_nativeExecute
            (JNIEnv *env, jclass clazz, jint n)
  {
    // .............
    (*env)->CallVoidMethod(env, m_object, m_mid,  sum);
  }
```

- 这个m_object正指向Java层的目前对象，而m_mid则是其setV()函数的ID。
- 依据这两项资料，就能透过VM的CallVoidMethod()函数而调用到目前Java对象的setV()函数，而把数据传送到Java层。

Summary：How To

- 拿目前对象指针换取它的类(目前类)ID：

```
jclass clazz = (*env)->GetObjectClass(env, thiz);
```

- 拿目前类ID换取某函数ID：

```
m_mid = (*env)->GetMethodID(env, m_class, "setV",
                            "(I)V");
```

- 依据类ID和函数ID，调用这指定的类里的指定的函数：

```
(*env)->CallVoidMethod(env, m_object, m_mid, sum);
```

~ Continued ~