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

C02_c

认识JNI开发与NDK(c)

By 高煥堂

3、如何载入*.so档案

VM的角色

- 由于Android的应用层级类别都是以Java撰写的，这些Java类别转译为Dex型式的Bytecode之后，必须仰赖Dalvik虚拟机器(VM: Virtual Machine)来执行之。VM在Android平台里，扮演很重要的角色。

- 此外，在执行Java类别的过程中，如果Java类别需要与JNI本地模块沟通时，VM就会去加载JNI本地模块，然后让Java的函数顺利地调用到本地模块的函数。此时，VM扮演着桥梁的角色，让Java与本地模块能透过标准的JNI接口而相互沟通。

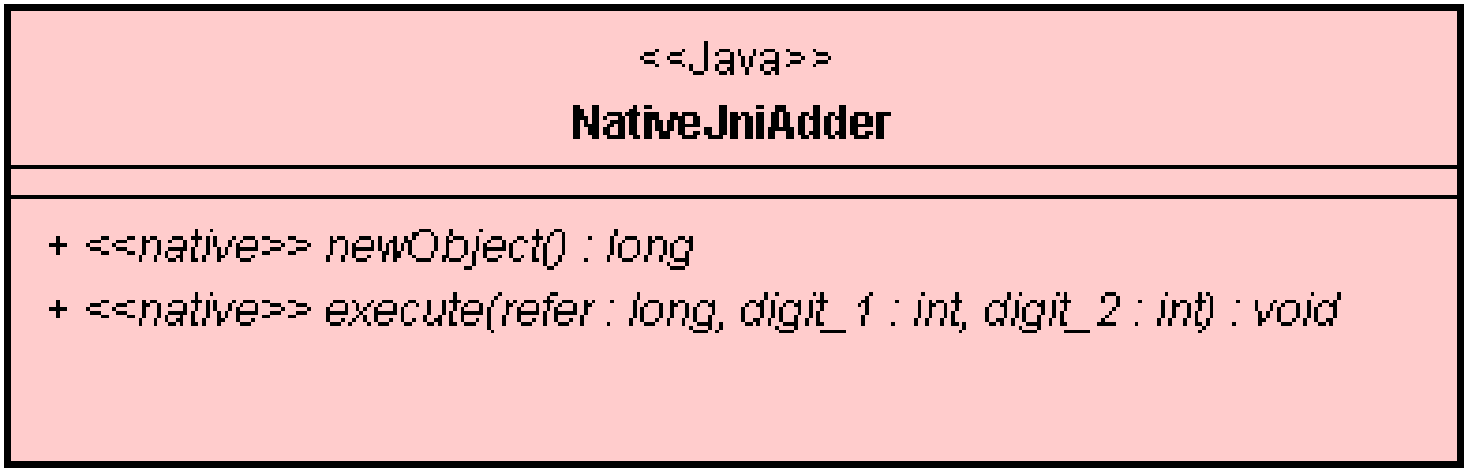
- Java层的类别是在VM上执行的，而本地模块则不是在VM上执行，那么Java程序又如何要求VM去加载(Load)所指定的C模块呢?可使用下述指令：

  System.loadLibrary(.so的檔名);*

- 例如，NativeJniAdder类别，其程序码：

NativeJniAdder.java

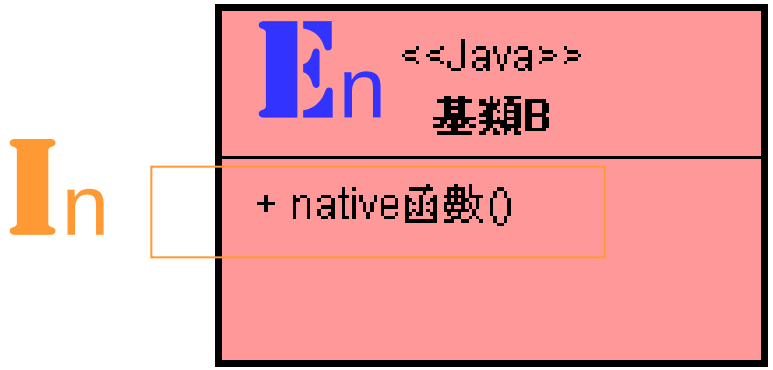
```
package com.misoo.gx06;
import android.util.Log;
public class NativeJniAdder {
  static {
    try {
      Log.i("JNI", "Trying to load libNativeJniAdder.so");
      System.loadLibrary("NativeJniAdder");
    }
    catch (UnsatisfiedLinkError ule) {
      Log.e("JNI", "Could not load libNativeJniAdder.so");
    }}
  public static native long newObject();
  public static native long execute(long refer,
                                    int digit_1, int digit_2);
}
```


<<Java>>
NativeJniAdder
+ <<native>> newObject() : long
+ <<native>> execute(refer : long, digit_1 : int, digit_2 : int) : void

```
/*  com_misoo_gx06_NativeJniAdder.c  */
#include "Adder.h"
#include "com_misoo_gx06_NativeJniAdder.h  "

JNIEXPORT jlong JNICALL
    Java_com_misoo_gx06_NativeJniAdder_newObject(JNIEnv *env,
              jclass c){
    Adder* ar = (Adder*)AdderNew();  創建一個C對象
    return (jlong)ar;
}
JNIEXPORT jlong JNICALL
    Java_com_misoo_gx06_NativeJniAdder_execute(
      JNIEnv *env, jclass c, jlong refer, jint digit_1, jint digit_2){
      Adder* pa = (Adder*)refer;  //轉成對象的指針
      long result = pa->exec(digit_1, digit_2);
      return result;
}
```

- 就要求VM去加载Android的/system/lib/libNativeJniAdder.so档案。载入*.so档之后，Java类别与*.so档就汇合起来，一起执行了。

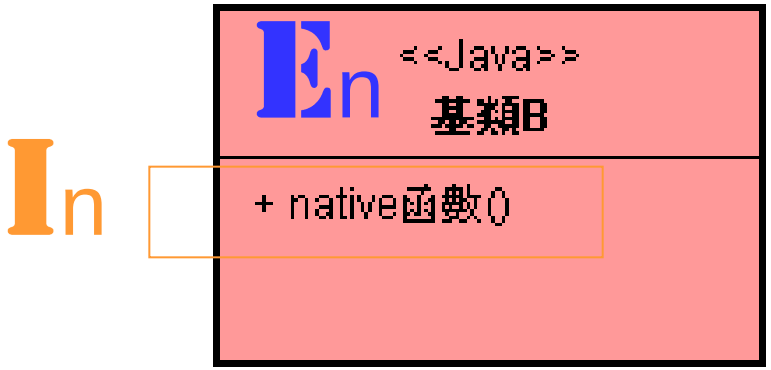


loadLibrary()进行
装配<Tn>的任务

Tn

以C档案形式，
包装本地函数的实现代码

NativeJniAdder.java



Tn

以C檔案形式,
包裝本地函数的实现代码

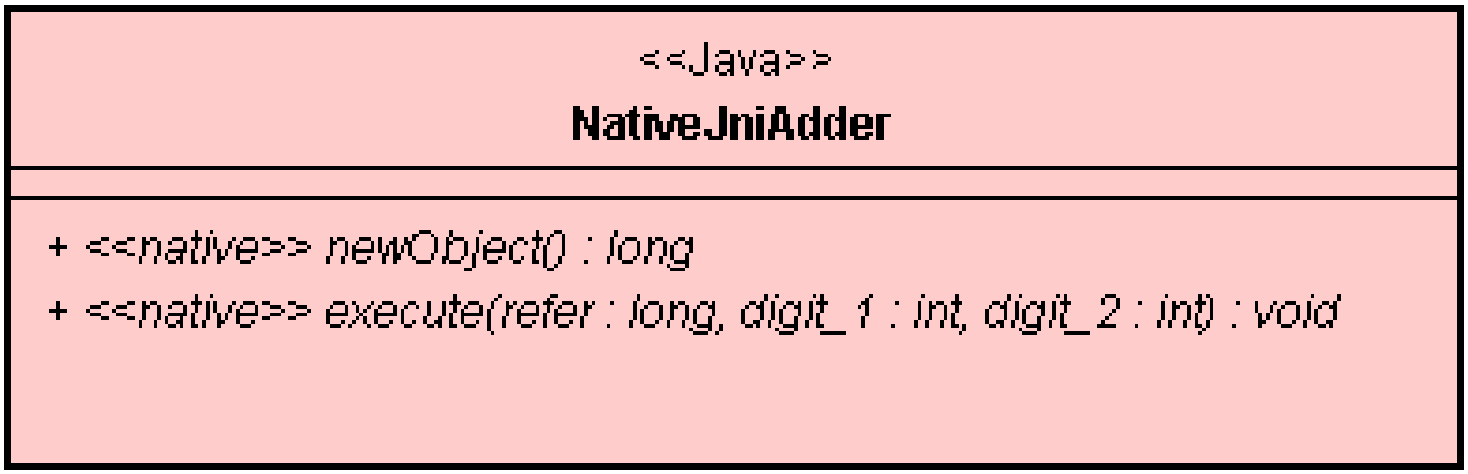
(libNativeJniAdder.so)

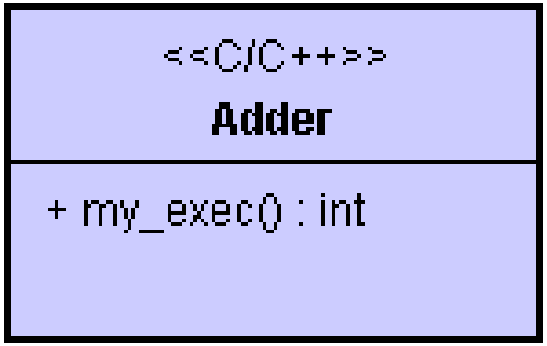
定义Adder类(Adder.h)

```
typedef struct Adder Adder;
struct Adder {
    int (*exec)(int a, int b);
};
```

撰写函数

```
struct Adder *AdderNew(){   // 構造式
      struct Adder *t
              = (Adder *)malloc(sizeof(Light));
      t->exec = my_exec;
      return (void*) t;
}
    static int my_exec( int a, int b ){
        return (a + b);
    }
```





以C档案形式，
包装本地函数的实现代码
(libNativeJniAdder.so)

将C/C++对象指针传回Java层

- 这个JNI接口定义类别含有2个函数：newObject()和execute()。
- 其中，newObject()函数诞生一个Adder对象，并且将该对象的指针传递回来给Java程序。

- 而execute()函数的refer参数，是用来让Java程序能将对象指针传进去给execute()函数，此时execute()就能藉由该指标而调用到先前newObject()函数所诞生的那个对象了。
- 典型的Java程序如下述的ac01类别：

ac01.java

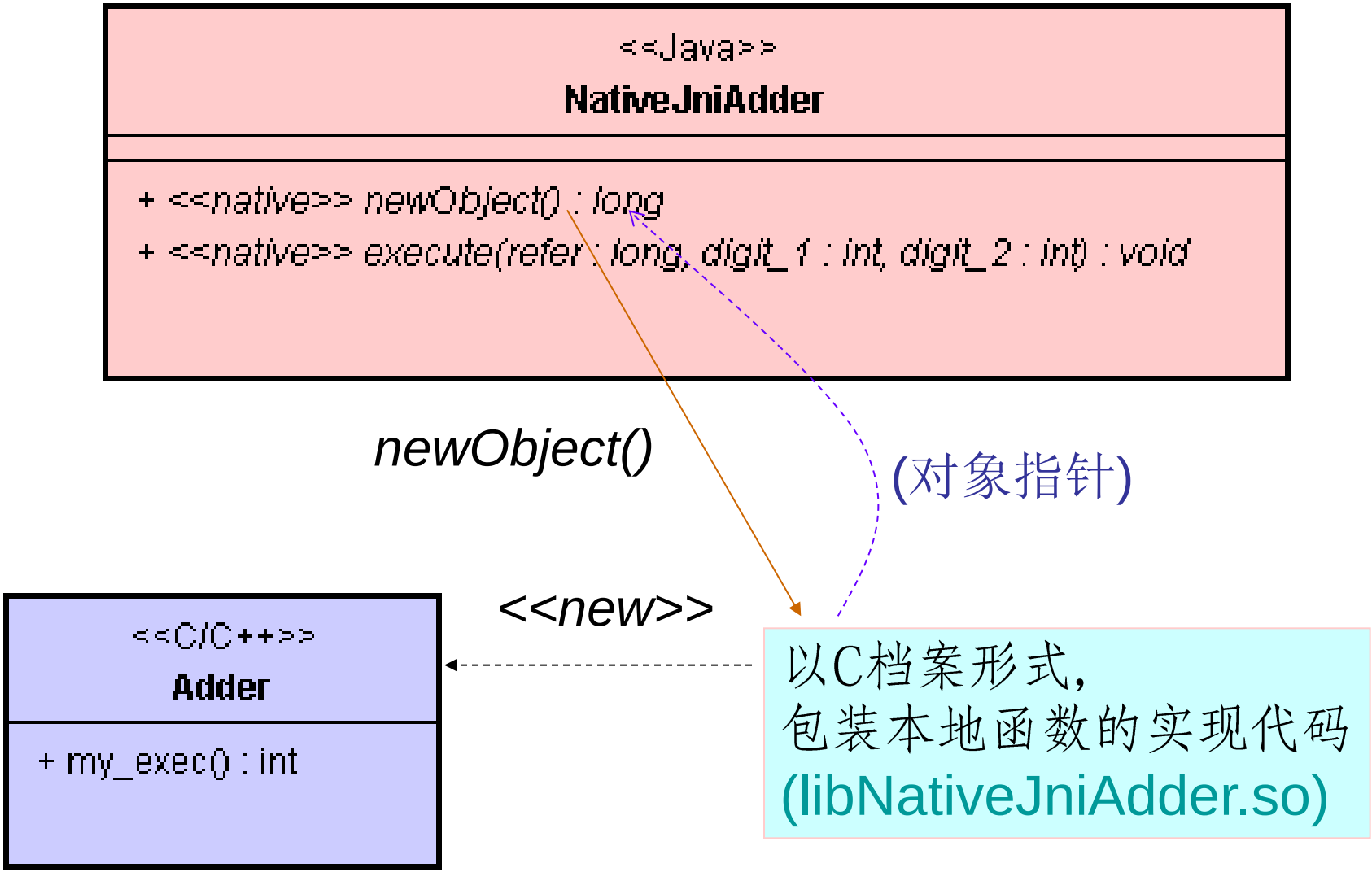
```
package com.misoo.gx06;
import android.app.Activity;
import android.os.Bundle;

public class ac01  extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
       //-------- 給c組件端測試之用 ----------------
       int a = 1, b = 0;
       long refer = NativeJniAdder.newObject();
       int cs = (int)NativeJniAdder.execute(refer, a, b);
       int sum = cs % 10;  int carry = cs / 10;
       String carry_sum_str = String.valueOf(carry)
                              + String.valueOf(sum);
       setTitle("&[" + carry_sum_str +"]&");
       //---------------------------------------
      }
}
```

- 在这ac01.java类别里，指令：

  long refer = NativeJniAdder.newObject();

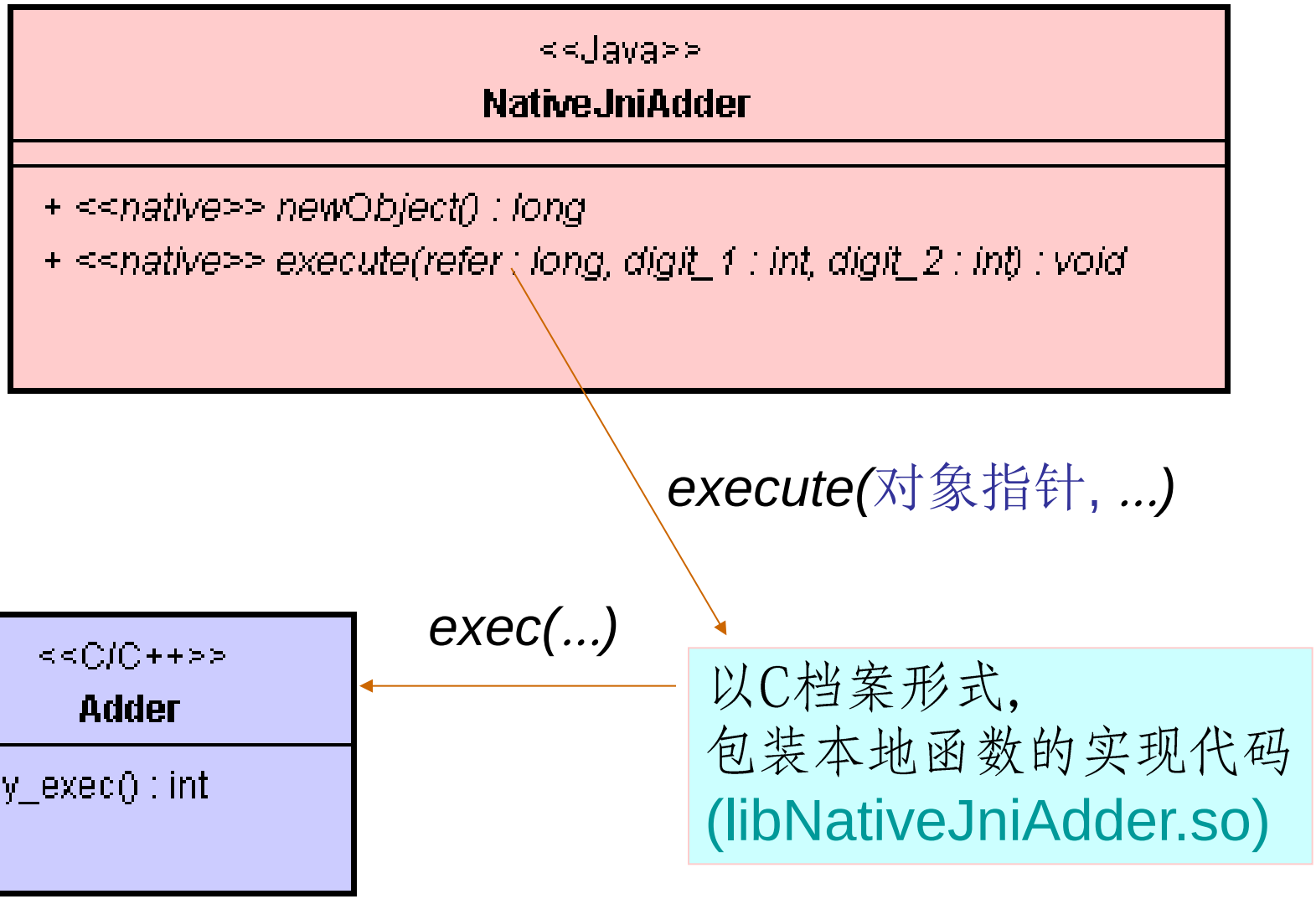
- newObject()诞生一个对象，将C/C++对象指针传回Java层。


<<Java>>
NativeJniAdder
+ <<native>> newObject() : long
+ <<native>> execute(refer : long, digit_1 : int, digit_2 : int) : void
newObject()
(对象指针)
<<new>>
<<C/C++>>
Adder
+ my_exec() : int
以C档案形式,
包装本地函数的实现代码
(libNativeJniAdder.so)

- 刚才newObject()诞生一个对象，由refer储存newObject()传回来的对象指针。指令：

  int cs = (int)NativeJniAdder.execute(refer, a, b);

- 将refer传进去给execute()函数。


<<Java>>
NativeJniAdder
+ <<native>> newObject() : long
+ <<native>> execute(refer : long, digit_1 : int, digit_2 : int) : void
execute(对象指针, ...)
exec(...)
<<C/C++>>
Adder
+ my_exec() : int
以C档案形式,
包装本地函数的实现代码
(libNativeJniAdder.so)

結語

- VM调用<Tn>本地函数时，将 Java层对象指针(pointer)传给<Tn>。
- 配上<Tn>之后，<Tn>可以将C/C++对象指针回传到Java层。
- 由于这些Java和C代码都在同一个进程里执行，所以指针都是可以互传的。

~ Continued ~