

# JNI 與 VM 介紹

## JNI: Java 層與 C++層的接口

在雙層框架裡，上層是 Java 框架，而下層是 C/C++框架。這兩層框架之間會有密切的溝通。此時 JNI(Java Native Interface)就扮演雙方溝通的接口了。藉由 JNI 接口，可將 Java 層的基類或子類的函數實作部份挖空，而移到 JNI 層的 C 函數來實作之。例如，原來在 Java 層有個完整的 Java 類：

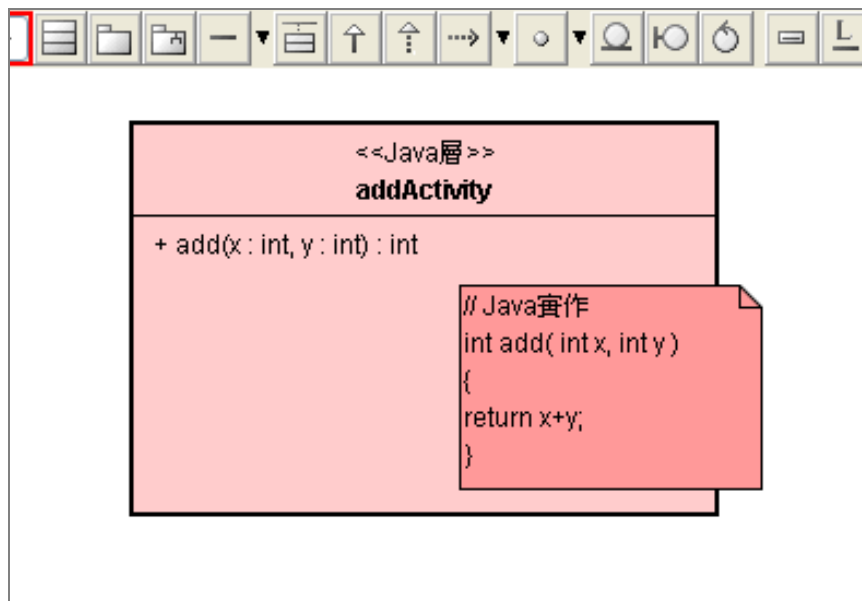


圖 1 一般的 Java 類

這是一個完整的 Java 類，其 `add()`函數裡有完整的實作(Implement)代碼。如果從這 Java 類裡移除掉 `add()`函數裡的實作代碼(就如同抽象類裡的抽象函數一般)，而成爲本地(Native)函數；然後依循 JNI 接口協定而以 C 語言來實作之。如下圖所示：

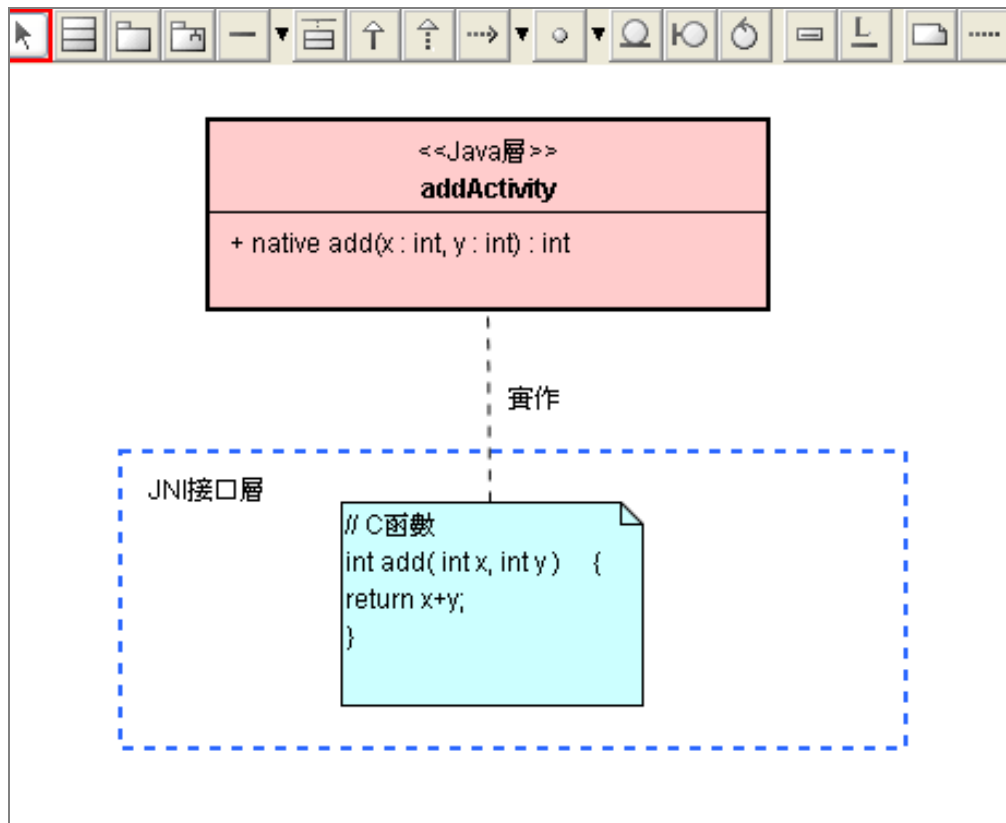


圖 2 以 C 語言來是做 Java 類的函數

這個 `add()` 函數仍然是 Java 類的一部分，只是它是用 C 語言來實作而已。為什麼要將 Java 類的 `add()` 函數挖空呢？其主要的理由是：Java 代碼執行速度較慢，而 C 代碼執行速度快。然而 Java 代碼可以跨平台，而 C 代碼與本地平台設備息息相關，所以稱之為本地(Native)代碼。

在本地的 C 代碼裡，可以創建 C++ 類的對象，並調用其函數。如下圖：

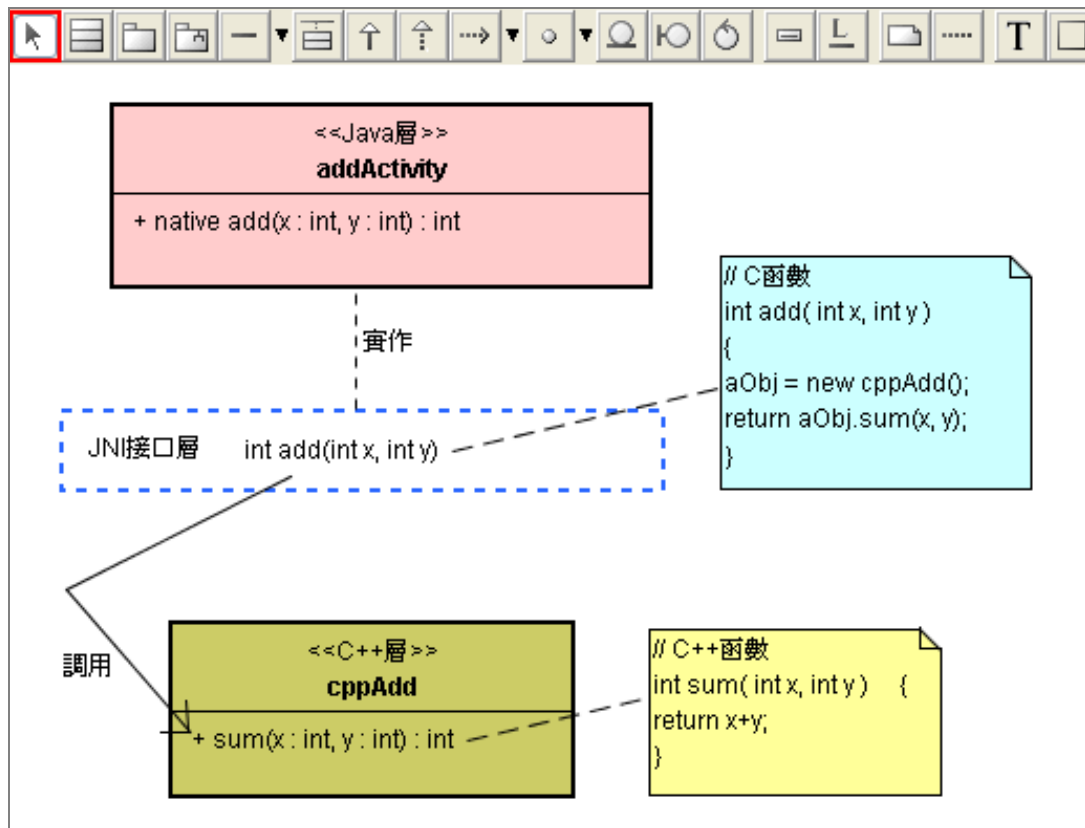


圖 3 Java 類透過本地程序來調用 C++函數

此圖可以簡潔地表示如下：

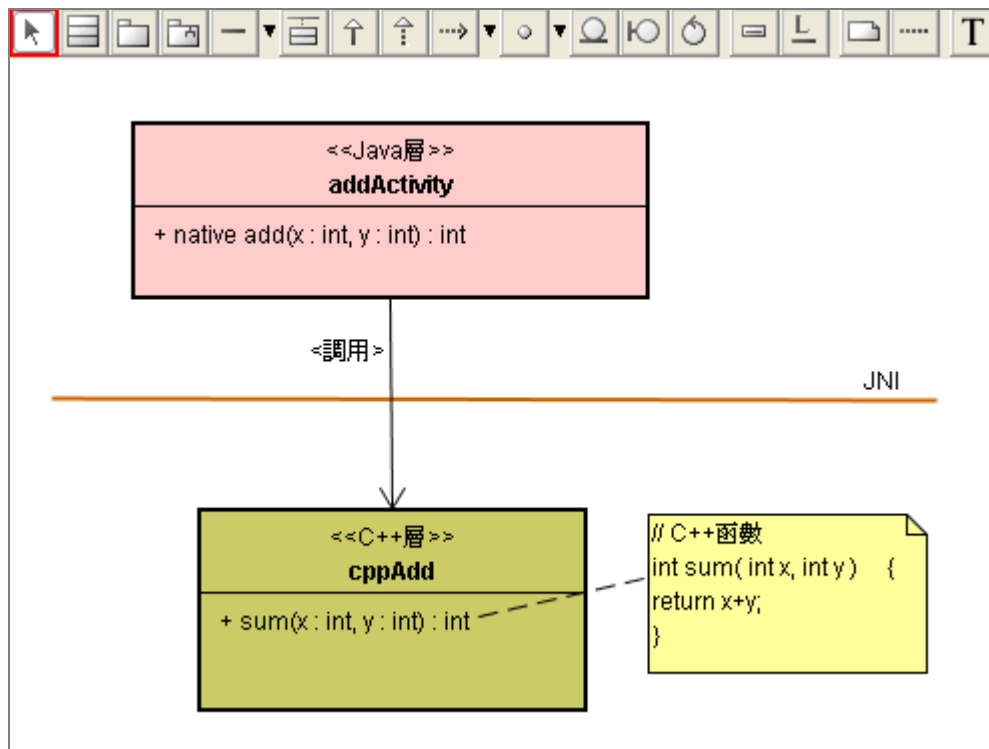


圖 4 JNI 是銜接 Java 層與 C++層的接口

藉由 JNI 接口，就能讓 Java 類與 C++類互相溝通起來了。這也是 Android 雙層框架的重要基礎機制。如下圖所示：

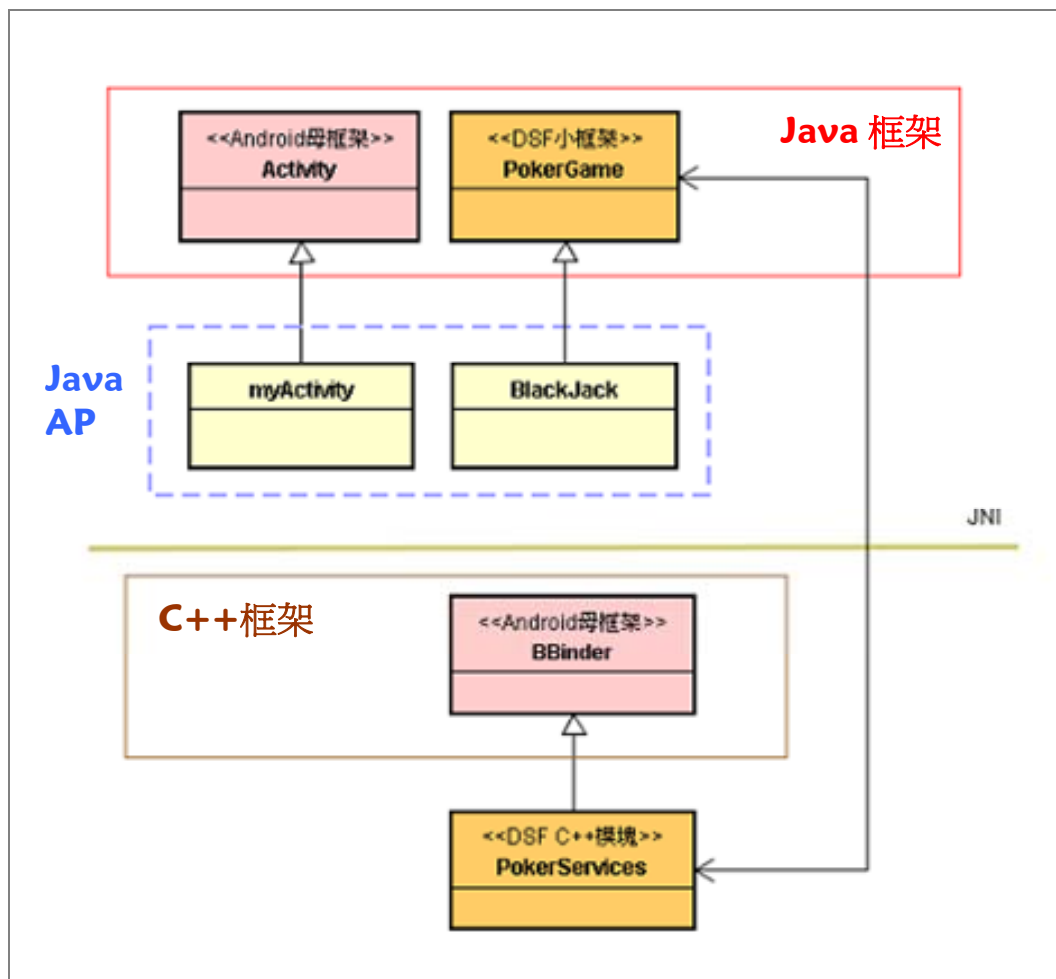


圖 5 JNI 接口是 Android 的雙層框架幕後的重要支柱

## 認識 JNI 與 VM 之關係

### 如何載入\*.so檔案

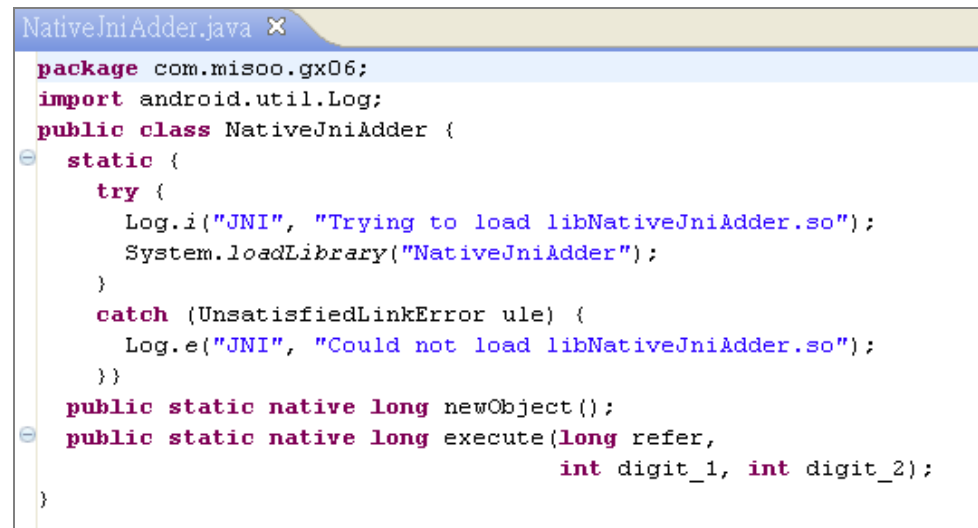
由於Android的應用層級類別都是以Java撰寫的，這些Java類別轉譯為Dex型式的Bytecode之後，必須仰賴Dalvik虛擬機器(VM: Virtual Machine)來執行之。VM在Android平台裡，扮演很重要的角色。

此外，在執行Java類別的過程中，如果Java類別需要與C組件溝通時，VM就會去載入C組件，然後讓Java的函數順利地呼叫到C組件的函數。此時，VM扮演著橋樑的角色，讓Java與C組件能透過標準的JNI介面而相互溝通。

應用層級的Java類別是在虛擬機器(VM: Virtual Machine)上執行的，而C組件不是在VM上執行，那麼Java程式又如何要求VM去載入(Load)所指定的C組件呢？可使用下述指令：

`System.loadLibrary(*.so 的檔名);`

例如，在上一節的範例裡的 `NativeJniAdder` 類別，其程式碼：



```
NativeJniAdder.java x
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);
}
```

就要求 VM 去載入 Android 的 `/system/lib/libNativeJniAdder.so` 檔案。再來看看 Android 框架裡所提供的 `MediaPlayer.java` 類別，內含指令：

```
public class MediaPlayer{
    static {
        System.loadLibrary("media_jni");
    }
    .....
}
```

這要求 VM 去載入 Android 的 `/system/lib/libmedia_jni.so` 檔案。載入 `*.so` 檔之後，Java 類別與 `*.so` 檔就匯合起來，一起執行了。

## 如何撰寫\*.so的入口函數

### JNI\_OnLoad()與JNI\_OnUnload()函數之用途

當 VM 執行到 `System.loadLibrary()` 函數時，首先會去執行 C 組件裡的 `JNI_OnLoad()` 函數。它的用途有二：

1. 告訴 VM 此 C 組件使用那一個 JNI 版本。如果你的 `*.so` 檔沒有提供 `JNI_OnLoad()` 函數，VM 會默認該 `*.so` 檔是使用最老的 JNI 1.1 版本。由於

新版的 JNI 做了許多擴充，如果需要使用 JNI 的新版功能，例如 JNI 1.4 的 `java.nio.ByteBuffer`，就必須藉由 `JNI_OnLoad()` 函數來告知 VM。

2. 由於 VM 執行到 `System.loadLibrary()` 函數時，就會立即先呼叫 `JNI_OnLoad()`，所以 C 組件的開發者可以藉由 `JNI_OnLoad()` 來進行 C 組件內的初期值之設定 (Initialization)。

例如，在 Android 的 `/system/lib/libmedia_jni.so` 檔案裡，就提供了 `JNI_OnLoad()` 函數，其程式碼片段為：

```

#define LOG_NDEBUG 0
#define LOG_TAG "MediaPlayer-JNI"
.....
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* env = NULL;
    jint result = -1;

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
    assert(env != NULL);
    if (register_android_media_MediaPlayer(env) < 0) {
        LOGE("ERROR: MediaPlayer native registration failed\n");
        goto bail;
    }
    if (register_android_media_MediaRecorder(env) < 0) {
        LOGE("ERROR: MediaRecorder native registration failed\n");
        goto bail;
    }
    if (register_android_media_MediaScanner(env) < 0) {
        LOGE("ERROR: MediaScanner native registration failed\n");
        goto bail;
    }
    if (register_android_media_MediaMetadataRetriever(env) < 0) {
        LOGE("ERROR: MediaMetadataRetriever native registration failed\n");
        goto bail;
    }
    /* success -- return valid version number */
    result = JNI_VERSION_1_4;
bail:
    return result;
}
// KTHXBYE
```

此函數回傳JNI\_VERSION\_1\_4值給VM，於是VM知道了其所使用的JNI版本了。此外，它也做了一些初期的動作(可呼叫任何本地函數)，例如指令：

```
if (register_android_media_MediaPlayer(env) < 0) {  
    LOGE("ERROR: MediaPlayer native registration failed\n");  
    goto bail;  
}
```

就將此組件提供的各個本地函數(Native Function)登記到VM裡，以便能加快後續呼叫本地函數之效率。

JNI\_OnUnload()函數與JNI\_OnLoad()相對應的。在載入C組件時會立即呼叫JNI\_OnLoad()來進行組件內的初期動作；而當VM釋放該C組件時，則會呼叫JNI\_OnUnload()函數來進行善後清除動作。當VM呼叫JNI\_OnLoad()或JNI\_Unload()函數時，都會將VM的指標(Pointer)傳遞給它們，其參數如下：

```
jint JNI_OnLoad(JavaVM* vm, void* reserved) {  
    .....  
}  
jint JNI_OnUnload(JavaVM* vm, void* reserved){  
    .....  
}
```

在JNI\_OnLoad()函數裡，就透過VM之指標而取得JNIEnv之指標值，並存入env指標變數裡，如下述指令：

```
jint JNI_OnLoad(JavaVM* vm, void* reserved){  
    JNIEnv* env = NULL;  
    jint result = -1;  
  
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {  
        LOGE("ERROR: GetEnv failed\n");  
        goto bail;  
    }  
}
```

由於 VM 通常是多執行緒(Multi-threading)的執行環境。每一個執行緒在呼叫JNI\_OnLoad()時，所傳遞進來的 JNIEnv 指標值都是不同的。為了配合這種多執行緒的環境，C 組件開發者在撰寫本地函數時，可藉由 JNIEnv 指標值之不同而避免執行緒的資料衝突問題，才能確保所寫的本地函數能安全地在 Android 的多執行緒 VM 裡安全地執行。基於這個理由，當在呼叫 C 組件的函數時，都會將 JNIEnv 指標值傳遞給它，如下：

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)  
{  
    JNIEnv* env = NULL;  
    .....  
}
```

```

        if (register_android_media_MediaPlayer(env) < 0) {
            .....
        }
    }
}

```

這 `JNI_OnLoad()` 呼叫 `register_android_media_MediaPlayer(env)` 函數時，就將 `env` 指標值傳遞過去。如此，在 `register_android_media_MediaPlayer()` 函數就能藉由該指標值而區別不同的執行緒，以便化解資料衝突的問題。

例如，在 `register_android_media_MediaPlayer()` 函數裡，可撰寫下述指令：

```

        if ((*env)->MonitorEnter(env, obj) != JNI_OK) {
            .....
        }
    }
}

```

查看是否已經有其他執行緒進入此物件，如果沒有，此執行緒就進入該物件裡執行了。還有，也可撰寫下述指令：

```

        if ((*env)->MonitorExit(env, obj) != JNI_OK) {
            .....
        }
    }
}

```

查看是否此執行緒正在此物件內執行，如果是，此執行緒就會立即離開。

## registerNativeMethods() 函數之用途

應用層級的 Java 類別透過 VM 而呼叫到本地函數。一般是仰賴 VM 去尋找 \*.so 裡的本地函數。如果需要連續呼叫很多次，每次都需要尋找一遍，會多花許多時間。此時，組件開發者可以自行將本地函數向 VM 進行登記。例如，在 Android 的 /system/lib/libmedia\_jni.so 檔案裡的程式碼片段如下：

```

#define LOG_NDEBUG 0
#define LOG_TAG "MediaPlayer-JNI"
.....
static JNINativeMethod gMethods[] = {
    {"setDataSource", "(Ljava/lang/String;)V",
     (void *)android_media_MediaPlayer_setDataSource},
    {"setDataSource", "(Ljava/io/FileDescriptor;JJ)V",
     (void *)android_media_MediaPlayer_setDataSourceFD},
    {"prepare", "()V", (void *)android_media_MediaPlayer_prepare},
    {"prepareAsync", "()V", (void *)android_media_MediaPlayer_prepareAsync},
    {"_start", "()V", (void *)android_media_MediaPlayer_start},
    {"_stop", "()V", (void *)android_media_MediaPlayer_stop},
    {"getVideoWidth", "()I", (void *)android_media_MediaPlayer_getVideoWidth},
    {"getVideoHeight", "()I", (void *)android_media_MediaPlayer_getVideoHeight},
    {"seekTo", "(I)V", (void *)android_media_MediaPlayer_seekTo},
    {"_pause", "()V", (void *)android_media_MediaPlayer_pause},
    {"isPlaying", "()Z", (void *)android_media_MediaPlayer_isPlaying},
    {"getCurrentPosition", "()I", (void *)android_media_MediaPlayer_getCurrentPosition},
}

```



```

{"getDuration",    "()I",  (void *)android_media_MediaPlayer_getDuration},
{"_release",      "()V",  (void *)android_media_MediaPlayer_release},
{"_reset",        "()V",  (void *)android_media_MediaPlayer_reset},
{"setAudioStreamType", "(I)V",
                    (void *)android_media_MediaPlayer_setAudioStreamType},
{"setLooping",    "(Z)V",  (void *)android_media_MediaPlayer_setLooping},
{"setVolume",     "(FF)V",  (void *)android_media_MediaPlayer_setVolume},
{"getFrameAt",    "(I)Landroid/graphics/Bitmap;",
                    (void *)android_media_MediaPlayer_getFrameAt},
{"native_setup",  "(Ljava/lang/Object;)V",
                    (void *)android_media_MediaPlayer_native_setup},
{"native_finalize", "()V",  (void *)android_media_MediaPlayer_native_finalize},
};
.....
static int register_android_media_MediaPlayer(JNIEnv *env){
    .....
    return AndroidRuntime::registerNativeMethods(env,
        "android/media/MediaPlayer", gMethods, NELEM(gMethods));
}
.....
//
jint JNI_OnLoad(JavaVM* vm, void* reserved){
    .....
    if (register_android_media_MediaPlayer(env) < 0) {
        LOGE("ERROR: MediaPlayer native registration failed\n");
        goto bail;
    }
    .....
}

```

當VM載入libmedia\_jni.so檔案時，就呼叫JNI\_OnLoad()函數。接著，JNI\_OnLoad()呼叫register\_android\_media\_MediaPlayer()函數。此時，就呼叫到AndroidRuntime::registerNativeMethods()函數，向VM(即AndroidRuntime)登記gMethods[]表格所含的本地函數了。簡而言之，registerNativeMethods()函數的用途有二：

1. 更有效率去找到函數。
2. 可在執行期間進行抽換。由於 gMethods[]是一個<名稱，函數指標>對照表，在程式執行時，可多次呼叫 registerNativeMethods()函數來更換本地函數之指標，而達到彈性抽換本地函數之目的。

由於JNI是Android用來融合Java與C/C++程式的關鍵技術，而它的幕後使用了極多的物件導向(即面向對象)觀念和技術。一樣地，AIDL是Android用來建立跨進程的IPC溝通的關鍵技術，它的幕後也使用了極多的物件導向觀念和技術。

~~ END ~~