J.N.I.
Développement d’un WRAPPER JAVA

1. Introduction

Comme nous le savons, l'une des principales forces de Java est sa portabilité - ce qui signifie qu'une fois que nous écrivons et compilons le code, le résultat de ce processus est un bytecode indépendant de la plate-forme.

En d'autres termes, ce code peut être exécuté sur n'importe quelle machine ou appareil capable d'exécuter une machine virtuelle Java, et il fonctionnera de manière aussi transparente que nous pouvons l'espérer.

Cependant, il arrive que nous ayons besoin d'utiliser du code compilé nativement pour une architecture spécifique.

Plusieurs raisons peuvent justifier l'utilisation d'un code natif :

La nécessité de gérer certains matériels
Amélioration des performances d'un processus très exigeant
une bibliothèque existante que nous voulons réutiliser au lieu de la réécrire en Java.

Pour ce faire, le JDK introduit un pont entre le bytecode qui s'exécute dans notre JVM et le code natif (généralement écrit en C ou C++).

Cet outil s'appelle Java Native Interface. Dans cet article, nous verrons comment écrire du code avec cet outil.

2. Fonctionnement

2.1. Méthodes natives : la JVM rencontre le code compilé

Java fournit le mot-clé **native** qui est utilisé pour indiquer que l'implémentation de la méthode sera fournie par un code natif.

Normalement, lors de la création d'un programme exécutable natif, nous pouvons choisir d'utiliser des bibliothèques statiques ou partagées :

Bibliothèques statiques
toutes les bibliothèques binaires seront incluses dans notre exécutable au cours du processus de liaison. Ainsi, nous n'aurons plus besoin des bibliothèques, mais cela augmentera la taille de notre fichier exécutable.
Librairies partagées

l'exécutable final ne contient que des références aux librairies, et non le code lui-même. Il faut que l'environnement dans lequel nous exécutons notre exécutable ait accès à tous les fichiers des librairies utilisées par notre programme.

C'est ce dernier point qui est logique pour JNI, car nous ne pouvons pas mélanger du bytecode et du code compilé nativement dans le même fichier binaire.

Par conséquent, notre librairie partagée conservera le code natif séparément dans son fichier .so/.dll/.dylib (en fonction du système d'exploitation que nous utilisons) au lieu de faire partie de nos classes.

Le mot-clé **native** transforme notre méthode en une sorte de méthode abstraite :

```
private native void aNativeMethod();
```

La principale différence est qu'au lieu d'être implémentée par une autre classe Java, elle sera implémentée dans une bibliothèque partagée native séparée.

Un tableau contenant des pointeurs en mémoire vers l'implémentation de toutes nos méthodes natives sera construit afin qu'elles puissent être appelées à partir de notre code Java.

2.2. Composants nécessaires

Voici une brève description des composants clés que nous devons prendre en compte. Nous les expliquerons plus en détail dans la suite de cet article

- Code Java - nos classes. Elles comprendront **au moins une méthode native**.
- Code natif - la logique réelle de nos méthodes natives, généralement codée en C ou C++.
- Fichier d'en-tête JNI - ce fichier d'en-tête pour C/C++ (include/jni.h dans le répertoire JDK). Comprend toutes les définitions des éléments JNI que nous pouvons utiliser dans nos programmes natifs.
- Compilateur C/C++ - nous pouvons choisir entre GCC, Clang, Visual Studio ou tout autre compilateur capable de générer une bibliothèque partagée native pour notre plateforme.

2.3. Éléments JNI dans le code (Java et C/C++)

Éléments Java :

- Mot-clé "**native**" - comme nous l'avons déjà vu, toute méthode marquée comme native doit être implémentée dans une **librairie partagée native**.
- **System.loadLibrary(String libname)** - méthode statique qui charge en mémoire une ibliothèque partagée depuis le système de fichiers et met ses fonctions exportées à la disposition de notre code Java.

Éléments C/C++ (nombre d'entre eux sont définis dans le fichier jni.h)

- JNIEXPORT - marque la fonction dans la bibliothèque partagée comme exportable afin qu'elle soit incluse dans la table des fonctions et que JNI puisse la trouver.

- JNICALL - combiné à JNIEXPORT, il garantit que nos méthodes sont disponibles pour le cadre JNI.
- JNIEnv - une structure contenant des méthodes que nous pouvons utiliser dans notre code natif pour accéder à des éléments Java
- JavaVM - une structure qui nous permet de manipuler une JVM en cours d'exécution (ou même d'en démarrer une nouvelle) en y ajoutant des threads, en la détruisant, etc...

3. Hello World JNI

Comment fonctionne JNI dans la pratique.
Dans ce tutoriel, nous utiliserons C++ comme langage natif et G++ comme compilateur et éditeur de liens.
Nous pouvons utiliser tout autre compilateur de notre choix, mais voici comment installer G++ sur Ubuntu/Debian, Windows et MacOS :

Ubuntu Linux - dans un terminal exécuter la commande

sudo apt-get install build-essential

Windows - Installer MinGW
MacOS - lancez la commande "g++" dans un terminal et s'il n'est pas encore présent, il l'installera.

3.1. Création de la classe Java

Commençons par créer notre premier programme JNI en implémentant un "Hello World" classique.

HelloWorldJNI.java

Pour commencer, nous créons la classe Java suivante qui
inclut la méthode native qui effectuera le travail :

```
/**
 *
 * @author florian
*/
package sayHelloJNI;

public class HelloWorldJNI {
    static {
        System.loadLibrary("native");
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        new HelloWorldJNI().sayHello();
        // aFaire : code logique de l'application
    }

    public native void sayHello();
}
```

Comme nous pouvons le voir, nous chargeons la bibliothèque partagée dans un bloc statique.

```
System.loadLibrary("<bibliothèquePartagée>");
```

Cela garantit qu'elle sera prête lorsque nous en aurons besoin et à partir de n'importe quel endroit.
NB : Celle-ci devra s'appeler lib<bibliothèquePartagée>.so
dans cet exemple libnative.so

Alternativement, dans ce programme trivial, nous pourrions charger la bibliothèque juste avant d'appeler notre méthode native parce que nous n'utilisons la bibliothèque native nulle part ailleurs.

3.2. Implémentation d'une méthode en C

Nous devons maintenant créer l'implémentation de notre méthode native en C++.

En C++, la définition et l'implémentation sont généralement stockées dans les fichiers .h et .cpp respectivement.
Tout d'abord, pour créer la définition de la méthode, nous devons utiliser le **drapeau -h du compilateur Java**. Il convient de noter que pour les versions antérieures à Java 9, nous devons utiliser l'outil javah au lieu de la commande **javac -h** :

```
javac -h . HelloWorldJNI.java
```

Cela génère un fichier sayHelloJNI_HelloWorldJNI.h avec toutes les méthodes natives incluses dans la classe passée en paramètre, dans ce cas, une seule :

sayHelloJNI_HelloWorldJNI.h

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

#ifndef _Included_sayHelloJNI_HelloWorldJNI
#define _Included_sayHelloJNI_HelloWorldJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     sayHelloJNI_HelloWorldJNI
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_sayHelloJNI_HelloWorldJNI_sayHello
  (JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

Comme nous pouvons le voir, le nom de la fonction est a**utomatiquement généré** en utilisant le nom pleinement qualifié du **paquetage**, de la **classe** et de la **méthode**.

Il est également intéressant de noter que deux paramètres sont transmis à notre fonction :

- un pointeur sur le JNIEnv actuel
- l'objet Java auquel la méthode est attachée, l'instance de notre classe HelloWorldJNI.

Nous devons maintenant créer un **nouveau fichier .cpp** pour l'**implémentation de la fonction sayHello**. C'est là que nous effectuerons les actions qui impriment "Hello World" sur la console.

Nous nommerons notre **fichier .cpp avec le même nom que le fichier .h** contenant l'en-tête et ajouterons ce code pour implémenter la fonction native :

sayHelloJNI_HelloWorldJNI.cpp

```
#include "sayHelloJNI_HelloWorldJNI.h"
#include <iostream>


JNIEXPORT void JNICALL Java_sayHelloJNI_HelloWorldJNI_sayHello
  (JNIEnv* env, jobject thisObject) {
    std::cout << "Hello from C++ !!" << std::endl;
}
```

3.3. Compilation et liaison

À ce stade, tous les éléments dont nous avons besoin sont en place et sont reliés entre eux.

Nous devons construire notre bibliothèque partagée à partir du code C++ et l'exécuter !

Compilation (pour la plateforme)

Pour ce faire, nous devons utiliser le compilateur G++, sans oublier d'**inclure les en-têtes JNI de notre installation Java JDK**.

Version Ubuntu /Debian :

```
g++ -c -fPIC -I${JAVA_HOME}/include -I${JAVA_HOME}/include/linux
         -I/usr/lib/jvm/java-11-openjdk-amd64/include/
         -I/usr/lib/jvm/java-11-openjdk-amd64/include/linux/
         sayHelloJNI_HelloWorldJNI.cpp
         -o sayHelloJNI_HelloWorldJNI.O
```

JNI.H se trouve en /usr/lib/jvm/java-11-openjdk-amd64/include/
JNI_VM.H se trouve en /usr/lib/jvm/java-11-openjdk-amd64/include/linux/

version Windows:

g++ -c -I%JAVA_HOME%\include -I%JAVA_HOME%\include\win32
com_baeldung_jni_HelloWorldJNI.cpp -o com_baeldung_jni_HelloWorldJNI.o

Version MacOS;

g++ -c -fPIC -I${JAVA_HOME}/include -I${JAVA_HOME}/include/darwin
com_baeldung_jni_HelloWorldJNI.cpp -o com_baeldung_jni_HelloWorldJNI.o

Bibliothèque (Linux : SO Windows DLL)
Une fois le code compilé pour notre plateforme dans le fichier sayHelloJNI_HelloWorldJNI.O, nous devons l'inclure dans une nouvelle bibliothèque partagée.
Le nom que nous décidons de lui donner est l'argument passé à la méthode **System.loadLibrary**.

Nous avons nommé la nôtre "**native**", et nous la chargerons lors de l'exécution de notre code Java.

L'éditeur de liens G++ lie ensuite les fichiers objets C++ à notre bibliothèque pontée.

Version Ubuntu/Debian

```
g++ -shared -fPIC -o libnative.so sayHelloJNI_HelloWorldJNI.O -lc
```

⇒ crée le fichier **libnative.so**

Version Windows:

g++ -shared -o native.dll com_baeldung_jni_HelloWorldJNI.o -Wl,--add-stdcall-alias

version MacOS:

g++ -dynamiclib -o libnative.dylib com_baeldung_jni_HelloWorldJNI.o -lc

4. Exécuter

Nous pouvons maintenant exécuter notre programme à partir de la ligne de commande. Cependant, nous devons **ajouter le chemin complet du répertoire contenant la bibliothèque** que nous venons de générer. Ainsi, Java saura où chercher nos librairies natives :

```
java -cp . -Djava.library.path=/NATIVE_SHARED_LIB_FOLDER
sayHelloJNI.HelloWorldJNI

/NATIVE_SHARED_LIB_FOLDER :   répertoire qui contien libnative.so
                              dans notre exemple :
                              = /home/florian/devoloppement/__W__/sayHelloJNI/
```

soit

```
florian@mayall-II-1:~/devoloppement/__W__$
java -cp .
-Djava.library.path=/home/florian/devoloppement/__W__/sayHelloJNI/
sayHelloJNI.HelloWorldJNI
```

5. Utilisation des fonctionnalités avancées de JNI

Dire bonjour, c'est bien, mais ce n'est pas très utile. En général, nous aimerions échanger des données entre le code Java et le code C++ et gérer ces données dans notre programme.

4.1. Ajouter des paramètres à nos méthodes natives

Nous allons ajouter quelques paramètres à nos méthodes natives.

Créons deux nouvelles méthodes native appelées
`sumIntegers`
`sayHelloToMe`
utilisant des paramètres et des retours de types différents :

HelloWorldJNI.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this
license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Other/File.java to edit this template
 */

/**
 *
 * @author florian
 */
package sayHelloJNI;
public class HelloWorldJNI {
    static {
        System.loadLibrary("native");
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        HelloWorldJNI huhu = new HelloWorldJNI();
        long res;
        huhu.sayHello();
        res = huhu.sumIntergers(10, 4);
        System.out.print(res);
        huhu.sayHelloToMe("floflo", true);
    }
    //=========================================================
    // NATIVES
    //=========================================================
    public native void sayHello();
    private native long sumIntergers(int a, int b);
    private native String sayHelloToMe(String myName, boolean isMale);

}
```

Ensuite, répétez la procédure pour créer un nouveau fichier .h avec "javac -h" comme nous l'avons fait précédemment.

```
    javac -h . HelloWorldJNI.java
```

pour generer **sayHelloJNI_HelloWorldJNI.h**
et compiler **HelloWorldJNI.class**

sayHelloJNI_HelloWorldJNI.h

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

#ifndef _Included_sayHelloJNI_HelloWorldJNI
#define _Included_sayHelloJNI_HelloWorldJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     sayHelloJNI_HelloWorldJNI
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_sayHelloJNI_HelloWorldJNI_sayHello
  (JNIEnv *, jobject);

/*
 * Class:     sayHelloJNI_HelloWorldJNI
 * Method:    sumIntergers
 * Signature: (II)J
 */
JNIEXPORT jlong JNICALL Java_sayHelloJNI_HelloWorldJNI_sumIntergers
  (JNIEnv *, jobject, jint, jint);

/*
 * Class:     sayHelloJNI_HelloWorldJNI
 * Method:    sayHelloToMe
 * Signature: (Ljava/lang/String;Z)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_sayHelloJNI_HelloWorldJNI_sayHelloToMe
  (JNIEnv *, jobject, jstring, jboolean);

#ifdef __cplusplus
}
#endif
#endif
```

Créez maintenant le fichier .cpp correspondant avec l'implémentation des nouvelles méthodes C++ :

sayHelloJNI_HelloWorldJNI.cpp

```
#include "sayHelloJNI_HelloWorldJNI.h"
#include <iostream>
/*
 * Class:     sayHelloJNI_HelloWorldJNI
 * Method:    sayHello
 * Signature: (II)J
 */
JNIEXPORT void JNICALL Java_sayHelloJNI_HelloWorldJNI_sayHello
  (JNIEnv* env, jobject thisObject) {
    std::cout << "Hello from C++ !!" << std::endl;
}
/*
 * Class:     sayHelloJNI_HelloWorldJNI
 * Method:    sumIntergers
 * Signature: (II)J
 */
JNIEXPORT jlong JNICALL
Java_sayHelloJNI_HelloWorldJNI_sumIntergers
  (JNIEnv* env, jobject thisObject, jint a, jint b)
{
    std::cout << "C++: The numbers received are : " << a << " and
" << b << std::endl;
    return (long)a + (long)b;
}
/*
 * Class:     sayHelloJNI_HelloWorldJNI
 * Method:    sayHelloToMe
 * Signature: (Ljava/lang/String;Z)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL
Java_sayHelloJNI_HelloWorldJNI_sayHelloToMe
  (JNIEnv* env, jobject thisObject, jstring nom, jboolean isMale)
{
    const char* nameCharPointer = env->GetStringUTFChars(nom,
NULL);
    std::string title;

    if(isMale) {
        title = "M. ";
    }
    else {
        title = "Mme ";
    }

    std::string fullName = title + nameCharPointer;
    return env->NewStringUTF(fullName.c_str());
}
```

Nous avons utilisé le pointeur *env de type JNIEnv pour accéder aux méthodes fournies par l'instance d'environnement JNI.

JNIEnv nous permet, dans ce cas, de passer des chaînes de caractères Java dans notre code C++ et d'en ressortir sans nous soucier de l'implémentation.

Nous pouvons vérifier l'équivalence des types Java et des types C JNI dans la documentation officielle d'Oracle.

Pour tester notre code, nous devons répéter toutes les étapes de compilation de l'exemple HelloWorld précédent.

Compilation / Link

```
g++ -c -fPIC -I${JAVA_HOME}/include -I${JAVA_HOME}/include/linux -I/usr/lib/jvm/java-11-openjdk-amd64/include/  -I/usr/lib/jvm/java-11-openjdk-amd64/include/linux/   sayHelloJNI_HelloWorldJNI.cpp -o sayHelloJNI_HelloWorldJNI.O

g++ -shared -fPIC -o libnative.so sayHelloJNI_HelloWorldJNI.O -lc
```

Exécution

```
florian@mayall-II-1:~/devoloppement/__W__$

java -cp . -Djava.library.path=/home/florian/devoloppement/__W__/sayHelloJNI/ sayHelloJNI.HelloWorldJNI
```

5.2. Utilisation d'objets et appel de méthodes Java à partir du code natif

Dans ce dernier exemple, nous allons voir comment nous pouvons manipuler des objets Java dans notre code C++ natif.

Nous commencerons par créer une nouvelle classe UserData que nous utiliserons pour stocker des informations sur l'utilisateur :

```
================================
=   Methode pàp creation bibli  =
================================
driver vmk80xx

g++ -c libK8055.cpp -o libK8055.o -lusb
g++ -o libK8055exe libK8055.o -lusb
sudo ./libK8055exe

g++ -c -fPIC -I${JAVA_HOME}/include -I${JAVA_HOME}/include/linux -
I/usr/lib/jvm/java-11-openjdk-amd64/include/  -I/usr/lib/jvm/java-
11-openjdk-amd64/include/linux/ lib_cards_K8055_K8055JNI.cpp -o
lib_cards_K8055_K8055JNI.o -lusb

g++ -shared  -o libK8055JNI.so lib_cards_K8055_K8055JNI.o


sudo java -cp .
-Djava.library.path=/home/florian/devoloppement/__W__/lib/cards/K8
055/  Test
```

Sudo car pas accèsà vmk80xx (voir pour un chmod 777)