Université Nice Sophia Antipolis

Jean-Yves Tigli, Stéphane Lavirotte
Polytech'Nice – Sophia
SI3
2017-2018

TD n°10
Mise en œuvre de JNI sur un cas concret

Nous allons ici traiter un exemple concret d'application de JNI. Il va nous permettre ici d'accéder à des informations système qui ne sont pas accessibles directement depuis Java (ce dernier étant « OS indépendant »).

1 Récupérer l'identifiant de processus d'une JVM (processus ID)

Imaginons que nous souhaitions récupérer le processus ID de la `jvm` qui exécute notre programme Java. Malheureusement, il n'existait pas d'interface Java qui soit plateforme indépendante et qui garantisse que cela fonctionne dans toutes les implémentations de jvm (cela n'a été introduit qu'à partir de Java 9, il en aura fallu du temps…).

Vous pouviez éventuellement récupérer une information intéressante via l'appel à `ManagementFactory.getRuntimeMXBean().getName()`. C'est simple, et marche probablement sur toutes les implémentations de `jvm`. Sous Linux ou Windows, la valeur retournée est du type `12345@hostname` (`12345` étant l'identifiant du processus java). Mais attention, suivant la documentation, il n'y a pas de garantie pour cette valeur :

« *Returns the name representing the running Java virtual machine. The returned name string can be any arbitrary string and a Java virtual machine implementation can choose to embed platform-specific useful information in the returned name string. Each running virtual machine could have a different name.* »

Exercice n°1:

Ecrivez une première méthode qui fournisse le numéro de processus ID à l'aide de `RunTimeMXBean`.

A partir de Java 9, la nouvelle API des processus est : `long pid = ProcessHandle.current().getPid();`.
https://docs.oracle.com/javase/9/docs/api/java/lang/ProcessHandle.html#pid--

Nous allons fournir cette même information, de manière fiable, à travers un appel natif. Pour toute version antérieure à Java 9, c'était la seule solution pour fournir cette information de manière fiable.

Exercice n°2:

A l'aide de l'appel système adapté à l'OS que vous utilisez (`GetProcessId` sous Windows et `getpid` en Posix), fabriquez au moins deux bibliothèques dynamiques natives (`.so`, `.dll` ou `.dylib`) qui fourniront cette fonctionnalité à Java. Ecrivez un programme Java utilisant la bonne bibliothèque selon l'OS et affichera l'information du numéro de processus de la `jvm`. Vérifiez que vous affichez la bonne valeur à l'aide de la commande `ps` sous Unix ou de la commande `PscView` sous Windows.

Plus généralement, s'il vous manque un élément système qui vous serait utile en Java, il vous suffit avec JNI de l'ajouter comme une nouvelle méthode utilisable.

2 Fork Posix d'une JVM : impossible n'est pas JNI

Il est possible d'exécuter un processus externe depuis Java, mais Java ne permet pas non plus de faire le `fork` d'une `jvm`. Nous allons utiliser cette fonctionnalité et ainsi rendre possible de dupliquer une `jvm` par le `fork` Posix.

Exercice n°3:

Ajouter à votre bibliothèque dynamique une fonction qui permettre d'appeler `fork` depuis un code Java.

Exercice n°4:

A l'aide des deux fonctions que vous venez de rendre accessible en Java, essayer de refaire les exercices 1 et 2 du TD 3, mais en les codant depuis Java.

TD n°10
Mise en œuvre de JNI sur un cas concret

Vous pourrez aussi consulter le projet `jnr` sur `github` qui propose de disposer de l'interface Posix En Java. Si vous consulter les codes suivants, vous verrez bien que, pour disposer de `fork` en Java, il n'est pas possible de le réaliser avec une interface de programmation purement Java (exemple `JavaPosix`), mais bien en appelant la fonction à partir de la librairie C (exemple `BaseNativePosix`).

- https://github.com/jnr/jnr-posix/blob/master/src/main/java/jnr/posix/JavaPOSIX.java#L246
- https://github.com/jnr/jnr-posix/blob/master/src/main/java/jnr/posix/BaseNativePOSIX.java#L453

3 Autres exemples d'utilisation de bibliothèques natives depuis un langage

Pour étendre les fonctionnalités proposées d'un langage, il est nécessaire de faire appel à du code natif, comme nous avons pu le voir précédemment avec `fork` et `getppid`.

Dans le cas où vous souhaitez utiliser des bibliothèques pour faciliter le développement d'une application, deux possibilités s'offrent à vous : utiliser une bibliothèque intégralement écrite dans le langage que vous utilisez ou bien utiliser une bibliothèque qui utilise une bibliothèque native.

Bien entendu, si une bibliothèque existe dans le langage utilisé pour le développement, il est souvent préférable d'utiliser celle-ci (si elle fournit les bonnes fonctionnalités et a été éprouvée).

Mais il est aussi possible de disposer de bibliothèques développées et éprouvées dans un langage natif et que l'on souhaite rendre disponibles dans un autre langage. Cette technique de wrapping existe non seulement pour Java à l'aide de JNI (il existe aussi JNA), mais bien entendu pour d'autres langages (C#, Node.js, Python, …).

Voici quelques exemples pour rendre accessible :

- les données d'une centrale inertielle en Node.js : https://github.com/lavirott/nodeimu
- les donnes du port GPIO de la Raspberry Pi en Java : https://github.com/mattjlewis/pigpioj
- le calcul scientifique en Python (NymPy) : http://www.numpy.org/
- …

Pour finir de vous convaincre que cette approche n'est pas spécifique à Java, voici les premières lignes de la documentation pour étendre Python avec du code C/C++ (la doc pointée est pour 2.7, mais cela est bien entendu vrai pour 3.x). https://docs.python.org/2/extending/extending.html : "*Extending Python with C or C++*"

"*It is quite easy to add new built-in modules to Python, if you know how to program in C. Such extension modules can do two things that can't be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.*

To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header "Python.h".

The compilation of an extension module depends on its intended use as well as on your system setup; details are given in later chapters."

S'il fallait encore vous convaincre, vous constatez que ce sont bien les mêmes principes et mécanismes que ceux que nous avons vu avec JNI.

4 Lancer une JVM depuis C

Vous avez vu en cours le fait de pouvoir exécuter une `jvm` depuis une application native pour piloter le lancement de vos programmes Java :

TD n°10
Mise en œuvre de JNI sur un cas concret

Exercice n°5:

Reproduisez le lancement de la `jvm` depuis un programme C en utilisant l'interface prévue pour cela par JNI.

Exercice n°6:

Exécutez une `jvm` à l'aide de l'interface Posix `execXX()` que nous avons vu lors des TDs précédents.

Comparez les deux méthodes.

5 Votre JVM étendue, plus qu'une JVM

Dans le code en C écrit pour lancer une jvm vous pouvez remarquer deux étapes. Une première consistant à créer une machine virtuelle java, (Cf. `JNI_CreateJavaVM`) et une seconde consistant à chercher et lancer la méthode main de la classe principale (Cf. `FindClass` et `GetStaticMethodID`).
Entre ces deux vous avez donc tout loisir d'inspecter l'environnement de votre programme java (Cf. JNIEnv *env).
A partir d'un tel constat, de multiples applications sont alors envisageables dans le domaine de la sécurité.

Exercice n°7:

Imaginez et illustrez l'introduction d'un cheval de Troie inoffensif dans votre code (ex. création d'un fichier pour marquer l'originalité de votre jvm, surcharge d'une méthode du code java lancé, …). Faites preuve d'imagination et illustrez votre proposition.

Exercice n°8:

Que pourrions-nous imaginer pour que seuls les programmes java dotés d'une « clef » dans leur code puisse se lancer avec votre jvm (et pas d'autres d'ailleurs). Illustrez votre proposition.