Commencez à développer dans OpenJDK 8 sous Ubuntu (partie 2)

PITTON Olivier
25 juillet 2013

Table des matières

Introduction

OpenJDK est un projet volumineux. Il comprend de nombreux projets différents, écrit dans plusieurs langages, principalement Java, C et C++. Cet article a pour vocation de présenter le développement dans OpenJDK en commençant par le projet JDK, puis le projet Hotspot. Nous verrons des points comme l'ajout de classes dans le JDK, l'intégration de la machine virtuelle dans Eclipse, l'apport de nouveaux paramètres VM. Je vous invite à lire mon précédent article sur l'installation d'OpenJDK 8, qui vous permettra de commencer avec un environnement stable. Cet article fait partie d'une série dédiée à OpenJDK 8.

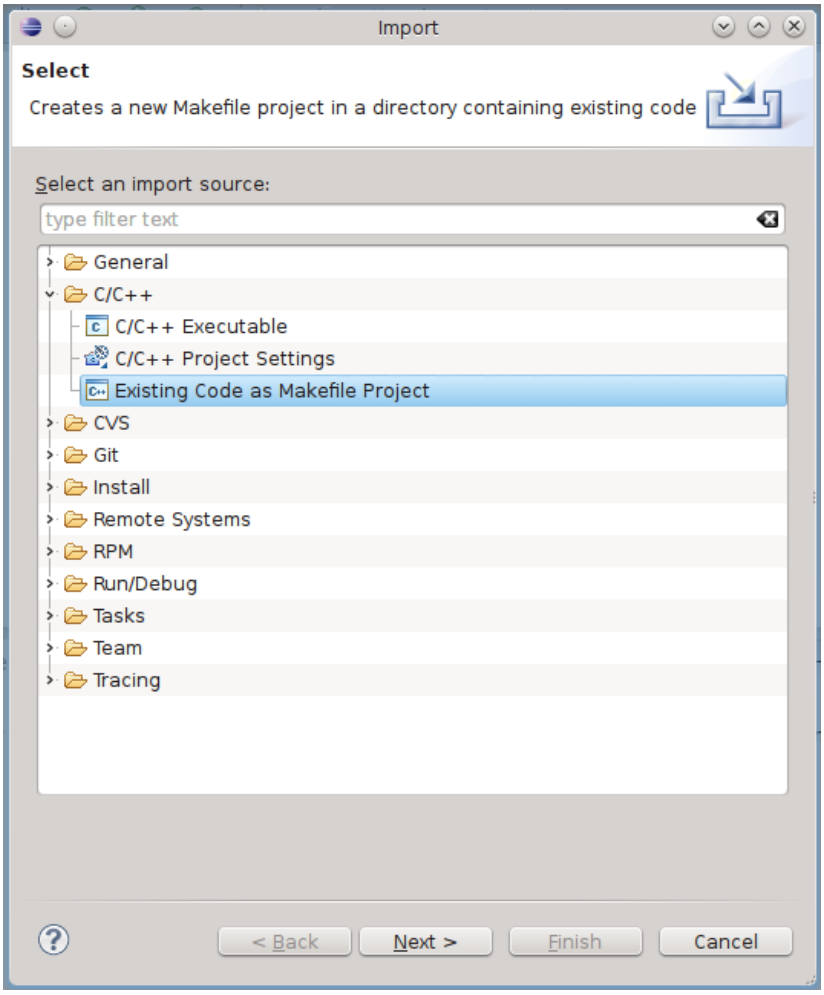
Développement dans OpenJDK

Introduction

Hotspot est la machine virtuelle Java sur laquelle repose tout programme développé dans ce langage. C'est elle qui gère les threads, les appels systèmes, les garbage collector, … et son rôle est plus que primordial. Elle est écrite en C++ et contient approximativement 250 000 lignes de code. Notre objectif va être d'intégrer la machine virtuelle Java dans Eclipse afin de faciliter le développement dans celle-ci, puis de modifier le code source de celle-ci. Nous ne verrons pas comment installer Eclipse C / C++, seulement l'intégration de la JVM. Enfin, nous verrons comment ajouter une classe dans le JDK afin de pouvoir l'utiliser comme n'importe quelle classe déjà existante. Cet article sera donc décomposé en deux parties, la première dédiée à la machine virtuelle, la seconde au jdk.

Intégration d'Hotspot dans Eclipse

L'intégration d'Hotspot dans Eclipse est simple. Lancez Eclipse et dans le *Package Explorer* faites *Clic droit -> Import* . Dans la sélection choisissez, *Existing Code as Makefile Project*, comme sur l'image ci-dessous.



Sélectionner le répertoire d'Hotspot, et non celui du projet OpenJDK, donc */home/<name>/Interne/jdk8/hotspot* et laisser les paramètres par défaut. Nous ne chercherons pas à compiler Hotspot depuis Eclipse.

Une fois cela réalisé, l'import du projet se termine. Eclipse va alors se mettre à indexer la totalité des fichiers de la machine virtuelle.

Le projet se décompose en trois grands répertoires :

1. *src* : contient les sources de la machine virtuelle
2. *test* : contient les sources des tests de la machine virtuelle
3. *make* : contient l'ensemble des fichiers de compilation de la machine virtuelle

Nous travaillerons uniquement dans le dossier *src*.
Les implémentations des garbage collector se trouvent dans le dossier *src/share/vm/gc_implementation*.

Rentrons maintenant dans le vif du sujet.

Ajouter un paramètre VM

Les paramètres VM sont un ensemble d'arguments permettant de paramétrer à souhaite Hotspot. Par exemple le paramètre -XX:+PrintGCDetails permet de donner des informations sur la mémoire utilisée par les GC. Notre objectif dans cette partie va être d'ajouter un nouveau paramètre dans la machine virtuelle afin de délimiter notre code. Dès lors que nous voudrons ajouter ou modifier du code existant, nous nous assurerons que notre paramètre est bien activé, afin d'éviter d'éventuelles effets de bord ou oubli.

Vous allez voir qu'ajouter un paramètre est très simple. Ouvrez le fichier *globals.hpp*, situé dans le répertoire *src/share/vm/runtime/*.

Selon le type de compilation (debug, product, …) les paramètres ont des visibilités différentes. Cela est réalisé durant la compilation grâce à un ensemble de macros détaillées ci-dessous. La déclaration d'un nouveau paramètre se fait toujours par le biais de l'une de ces macros.

Les paramètres peuvent avoir quatre types distincts :

1. intx : Un entier
2. uintx : Un long
3. ccstr : Alias pour une chaîne de caractère constante (const char*)
4. bool : Un booléen

La déclaration d'un nouveau paramètre se fait grâce à l'une des macros suivantes :

1. product : Le paramètre est toujours accessible et modifiable.
2. develop : Le paramètre est accessible et modifiable en mode développement et est une constante en mode production.
3. experimental : Le paramètre est toujours accessible. Cette macro sert à tester de nouvelles fonctionnalités pendant le développement. Pour activer les paramètres de type experimental, il faut lancer la VM avec le paramètre -XX:+UnlockExperimentalVMOptions. Sans cela, il ne sera pas accessible.
4. lp64_product : Le paramètre sera toujours une constante dans une version 32 bits de la VM.
5. notproduct : Le paramètre est accessible et modifiable en mode développement et n'existe pas en mode production.
6. diagnostic : Le paramètre est toujours accessible. Cette macro sert pour diagnostiquer des bugs et s'assurer de la qualité de la machine virtuelle. Pour activer les paramètres de type diagnostic, il faut lancer la VM avec le paramètre -XX:+UnlockDiagnosticVMOptions. Sans cela, il ne sera pas accessible.
7. manageable : Le paramètre est toujours accessible et modifiable. De plus, il peut être dynamiquement modifier durant l'exécution de la VM, par le biais de la JConsole ou JMX (via com.sun.management.HotSpotDiagnosticMXBean).

8. product_rw : Le paramètre est toujours accessible et modifiable. De plus, il peut être dynamiquement modifier durant l'exécution de la VM. Cela ressemble à manageable sauf que l'intérêt de ce type de paramètre est pour un usage privé.

Il existe deux autres macros spécifiques develop_pd et product_pd dont nous ne détaillerons pas l'intérêt ici. Dans la majorité des cas, ce sont les macros product, develop, diagnostic et experimental qui sont intéressantes.

Pour nos modifications, nous nous servirons d'un paramètre de type product et booléen.

```
product(bool, TutorialGC, false, "Used for this tutorial") \
```

Les paramètres à donner permettent de spécifier respectivement, le type, le nom, la valeur par défaut et la description du paramètre. La valeur par défaut sera faux, nous forçant à activer le paramètre en utilisant -XX:+TutorialGC. Ainsi, nos modifications ne seront pas prises en compte lors d'un lancement classique.

N'oubliez pas d'ajouter des "\" pour chaque ligne vide et à la fin de la déclaration du paramètre, comme cela est fait dans le code existant.

Afin de valider ce que nous avons modifié, recompiler le projet et exécuter un programme Java quelconque (un Hello World suffit) et utilisant le paramètre -XX:+TutorialGC.

Si le paramètre a mal été ajouté, ou que vous n'avez pas recompilé correctement, l'erreur suivante s'affichera :

```
Unrecognized VM option 'TutorialGC'
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
```

Ajout de classes dans le JDK

Dans cette partie de l'article, nous allons voir comment ajouter notre propre classe dans le JDK et la lier à la compilation. Lors de la création des images, notre classe sera alors directement intégrée aux JARs et sera accessible comme toutes les autres classes, telle que System ou String.

Dans de nombreux projets, un ensemble de classes utilitaires est toujours développé afin de faciliter le développement. La bibliothèque Apache Commons Lang se retrouve très régulièrement et embarque un nombre impressionnant de classes utilitaires. Notre objectif va être d'intégrer la classe SystemUtils d'Apache dans le JDK. J'ai choisi cette classe car elle ne possède que peu de dépendance externe, donc ne nécessite que très peu de modifications en vue de son intégration.

Il faut bien comprendre que si nous voulons ajouter une "vraie" classe, il faut faire attention à ceux à quoi elle est liée et surtout la liste des classes de la bibliothèque qu'elle utilise. Si une classe utilise cinq autres classes du projet Apache Commons Lang, alors nous devons intégrer les cinq autres, qui peuvent elles aussi dépendre d'autres classes et ainsi de suite. On se retrouve alors à ajouter une partie entière de la bibliothèque pour une seule classe à la base. Enfin, si vous désirez ajouter et coder votre propre classe, la façon de faire est absolument identique à la différence que vous n'aurez aucune dépendance, sauf sur votre propre code et les classes du JDK, ce qui ne pose aucun problème.

Tout d'abord, voyons les principaux répertoires qui vont nous intéresser :

1. **jdk8/jdk/src/share/classes** : Contient toutes les classes Java du JDK
2. **jdk8/jdk/src/share/native** : Contient le code natif (écrit en C) des classes Java. Nous ne servirons pas de ce répertoire, mais il est positif de savoir où il se trouve. Lorsque vous faites appel à *System.gc()*, vous voulez lancer le garbage collector. Pour cela, il faut réaliser un appel vers la machine virtuelle HotSpot, donc lier du code Java à la VM. Ce code de liaison, réalisé grâce au mot-clé native, est fait dans ce répertoire.
3. **jdk8/jdk/make/java/java** : Contient les fichiers de propriétés pour la compilation des classes Java. La déclaration des classes à compiler, des fichiers C, des méthodes natives, … Tout cela se fait ici.

Nous allons donc ajouter nos classes dans *jdk8/jdk/src/share/classes*, puis déclarer que nous devons les compiler et les ajouter comme les autres dans le JDK, grâce aux fichiers situés dans *jdk8/jdk/make/java/java*.

Passons à la pratique. Télécharger les sources du projet à l'adresse suivante : http://commons.apache.org/proper/commons-lang/download_lang.cgi. Pour cet article,

j'ai utilisé la version 3.1.

Puisque nous désirons ajouter une classe utilitaire, nous placerons celle-ci dans le package java.util. Cela n'est en rien une nécessité, mais un choix personnel. Copier les classes SystemUtils et JavaVersion, situées dans *src/main/java/org/apache/commons/lang3* dans le répertoire *jdk8/jdk/src/share/classes/java/util*. Nous devons ajouter l'énumération JavaVersion, car elle est utilisée dans SystemUtils.

Ces deux classes étaient situées dans le package org.apache.commons.lang3, or nous les avons déclaré dans java.util. Nous devons donc éditer les deux classes et remplacer le code suivant :

```
package org.apache.commons.lang3;
```

par celui-ci :

```
package java.util;
```

Une fois cela fait, allez dans le répertoire *jdk8/jdk/make/java/java*. Voici une description des principaux fichiers de compilation :

1. **FILES_java.gmk** : Contient la liste des classes Java à compiler. Cette liste n'est pas complète, vous pourrez trouver d'autres fichiers comme celui-ci pour certaines implémentations Sun par exemple.
2. **FILES_c.gmk** : Contient la liste des fichier c à compiler. Les fichiers contiennent le code des méthodes déclarées natives dans une classe. Par exemple la méthode Runtime.gc() est native, vous trouverez donc le code C associée dans le fichier Runtime.c avec comme nom de méthode "Java_java_lang_Runtime_gc".
3. **Exportedfiles.gmk** : Contient la liste des classes Java pour lesquelles le compilateur doit générer un header c (.h). Ce header généré permet d'appeler des méthodes définies dans un autre fichier. Par exemple, vous pouvez importer le header associé à Runtime, pour faire exécuter des méthodes natives de cette classe.
4. **mapfile-vers** : Contient la liste des méthodes natives de toutes les classes Java. Lorsque vous déclarez une méthode native, celle-ci doit se retrouver dans ce fichier.

Puisque nos deux classes ne contiennent pas de méthodes natives, nous avons uniquement besoin de les déclarer dans le fichier FILES_java.gmk. Ajoutez deux lignes, sur le même modèle que les autre classes, pour nos deux nouvelles classes, soit java/util/SystemUtils.java et java/util/JavaVersion.java. N'oubliez pas d'ajouter des "\" à la fin de chaque ligne.

Une fois cela fait, il ne vous reste plus qu'à compiler le JDK.

Pour tester que l'ajout fonctionne bien, il vous suffit de créer une classe dans laquelle vous appelez des méthodes ou des constantes de la classe SystemUtils.

```
import java.util.SystemUtils;

public class Test {

        public static void main(String[] args) {
                System.out.println(SystemUtils.JAVA_VM_VERSION);
        }
}
```

Sur ma machine, le résultat affiché est "25.0-b39" correspondant bien à la version compilée, affichée grâce à l'option "java -version" et me donnant comme affichage :

openjdk version "1.8.0-internal"
OpenJDK Runtime Environment (build 1.8.0-internal-olivier_2013_06_30_00_34-b00)
OpenJDK 64-Bit Server VM (build 25.0-b39, mixed mode)

Conclusion

Nous avons vu comment intégrer la machine virtuelle Hotspot dans l'IDE Eclipse CDT, l'ajout de paramètres dans celle-ci ainsi que l'ajout de classes dans le JDK. Des explications ont été fournies sur le fonctionnement de ces parties du projet OpenJDK ainsi qu'une description des différents fichiers et répertoires utiles au développement. Il ne vous reste plus qu'à modifier le code et à vous amuser à découvrir l'intérieur d'une plateforme utilisée par des millions d'utilisateurs quotidiennement.