La Java Virtual Machine selon Julien Chable :

Référence officielle :

(cf http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html)

DESCRIPTION

Voici un article qui vous introduira aux binaires Java, afin de pouvoir optimiser ou modifier votre code et peut-être, pour vous, de créer un mini compilateur Java, un obfuscateur ou un générateur de code à la volée !

Dans un premier temps, nous nous attarderons sur la machine virtuelle Java ou JVM (Java Virtual Machine), de cette façon, la compréhension des instructions binaires et de la structure d'un fichier .class sera plus facile à aborder par la suite. Cette première partie n'est rien de plus qu'un rassemblement des spécifications de la JVM, et un cours d'introduction sur le 'byte-code' (ce que l'on pourrait appeler l'assembleur du Java). Amusez-vous bien, c'est une nouvelle dimension pour ceux qui connaissent déjà le langage Java sur le bout des doigts !

BINAIRE JAVA

Lorsque vous compilez votre code Java à l'aide de *javac* par exemple, le code résultant de cette opération est appelé ***byte-code***. Cecode compilé peut-être exécuté par n'importe quelle JVM répondantauxspécifications de SUN, de plus le format de ce code binaire estindépendant du matériel et de la plateforme sur laquelle il estexécuté. Le code peut-être, mais pas nécessairement, contenu dans un fichier (avec généralement l'extension *.class*). Le format de fichier ***class*** définit précisemment la représentation d'une classe ou d'une interface Java dans un fichier de ce type.

REVISION ET NOUVEAUTE

Les types

Le langage de programmation Java, tout comme la machine virtuelle, opère sur 2 sortes de type : les types dit 'primitif' et les types dit'référence'. De ce fait, une variable peut contenir ces 2 sortes de types.Une référence peut-être de plusieurs types : de type *classe*, de type *tableau*, et de type *interface*.

Parmi les types supportés par la JVM, nous pouvons distinguer les types *numériques*, les *booléen* et les types de *retour d'adresse* (returnAddress).

Les numériques

Les types numériques sont représentés par les types dit *intégral* et les types dit à *virgule flottante*. Les types intégrals sont :

- **byte**, valeur entière sur 8 bits signée (complément à 2) de -128 à 127 inclus.
- **short**, valeur entière sur 16 bits signée (complément à 2) de -32768 à 32767 inclus.

- **int**, valeur entière sur 32 bits signée (complément à 2) de -2147483648 à 2147483647 inclus.
- **long**, valeur entière sur 64 bits signée (complément à 2) de -9223372036854775808 à 9223372036854775807 inclus.
- **char**, valeur entière sur 16 bits non signée représentant un caractère Unicode de 0 à 65535 inclus.

Les types à virgule flottante sont:

- **float**, valeur à virgule flottante sur 32 bits à simple précision IEEE
- **long**, valeur à virgule flottante sur 64 bits à double précision IEEE

Le booléen

Le type booléen :

- **boolean**, encode une valeur de vérité *true* (vrai) ou *false* (faux)

Le retour d'adresse

Le type de *returnAddress* est un pointeur vers un des opcodes de la JVM (utilisé par les instructions *jsr, ret, et jsr_w*). **Le type *returnAddress* n'est pas directement associé au langage de programmation Java.**

ZONE DE DONNEE AU RUNTIME

La JVM définit de nombreuses zones de données pendant le runtime qui sont utilisées pendant l'exécution d'un programme. Certaines zones sont créées au démarrage de la JVM et sont détruites seulement lorsque celle-ci s'arrête. D'autres zones de données sont utilisées pour chaque Thread, elles sont créées lorsqu'il est créé, et détruites lors de sa destruction. Afin de mieux comprendre le mécanisme d'exécution des byte-codes, nous allons voir rapidement ces différents espaces de stockage dedonnées.

Le registre *pc*

La machine virtuelle supporte plusieurs threads à la fois pendant l'exécution d'un programme. Chaque thread de la machine virtuelle Java possède son propre pc (program counter).

La pile de la JVM

Chaque thread de la JVM possède une pile privée, créée en même temps que le Thread. Une pile de JVM stocke des *frames* (cf la partie sur les *frames*).

Le tas

La machine virtuelle Java possède un tas commun à tous les threads de lamachine virtuelle. La tas est l'espace mémoire depuis lequel toutes les instances de classes et les tableaux sont alloués. Le tas est créé au démarrage de la machine virtuelle Java.

Le stockage d'objet dans le tas est géré par un système de gestion de stockage automatique (connu sous le nom de *garbage collector*), de ce fait les objets ne sont jamais désalloués explicitement.

Zone de méthodes

La JVM possède une zone de méthodes qui est partagée parmi tous les threads de celle-ci. Cette zone stocke les structures par classe. Chaque structure comprend la *constant pool*, les champs, les données des méthodes et le code pour les méthodes et les constructeurs, incluant les méthodes spéciales utilisées pour l'initialisation des classes et des instances.

La *constant pool* du runtime

La constant pool est une représentation par classe ou par interface de la table constant_pool d'un fichier *class* (nous reviendrons ultérieurement dessus). Elle contient plusieurs sortes de constantes, allant du litéral numérique connu à la compilation, jusqu'aux références de méthodes ou de champs qui doivent être résolues pendant le runtime.
Une constant pool est allouée dans la *zone des méthodes*, pour chaque classe ou interface créée par la machine virtuelle Java.

La pile des méthodes natives

Nous ne nous attarderons pas sur cette zone de données, le sujet dépassant l'objectif de cet article.

Les *frames*

Une frame est utilisée pour stocker des données et des résultats partiels, tout comme pour effectuer les liaisons dynamiques (dynamic linking), retourner les valeurs des méthodes, et dispatcher les exceptions.

Une nouvelle frame est créée chaque fois qu'une méthode est invoquée, et est détruite lorsque l'invocation se termine. Une frame est allouée sur la pile de la thread qui a créée cette frame. Ainsi chaque frame possède son propre tableau de variables locales, sa propre pile d'opérandes, et une référence vers la constant pool de la classe de la méthode courante.

Seulement une frame, la frame de la méthode qui s'exécute, est active à un moment donné. La frame est identifiée comme la *frame courante*, et sa méthode est identifiée comme étant la *méthode courante*. La classe dans laquelle la méthode courante est définie s'appelle la *classe courante*. Lorsque l'on parlera d'opérations sur les variables locales et la pile des opérandes, ces opérations s'effectueront toujours sur ceux de la frame courante.

Une frame cesse d'être courante si sa méthode invoque une autre méthode ou si la méthode se termine (normalement ou pas ->exception). Quand une méthode est invoquée, une nouvelle frame est créée et devient courante. Lors du retour d'une méthode, la frame courante passe le résultat de son invocation de méthode (la valeur de retour de la méthode), à la frame précédente, si elle existe. La frame courante est alors détruite et la frame précédente redevient la frame courante.

Les variables locales

Chaque frame contient un tableau de variables connu sous le nom de *variables locales*. Une variable locale simple peut contenir un type boolean, byte, char,short, int, float, reference, ou returnAddress. Une paire de variables locales (soit 2 variables simples) peut contenir un type long ou double.

Les variables locales sont adressées par indexation. L'index de la première variable locale est 0. L'intervalle d'un index est donc [0;taille du tableau des variables locales - 1]. *Une valeur de type long ou de type double occupe 2 variables locales* ***consécutives****.*

La JVM utilise les variables locales pour passer des paramètres lors de l'invocation de méthode. Lors de l'invocation d'une méthode de classe (méthode statique, cf *Définitions* en fin d'article), tous les paramètres sont passés dans des variables locales consécutives en partant de la variable locale d'index 0. Lors de l'invocation d'une méthode d'instance (cf *Définitions* enfin d'article), la variable locale d'index 0 est toujours utilisée pour passer une référence à l'objet à partir duquel cette instance de méthode a été invoquée (***this*** en Java). Tous les autres paramètres sont passés dans les variables locales consécutives en partant de l'index 1.

La pile des opérandes

Chaque frame contient une pile (Last In First Out) connue sous le nom de *pile des opérandes*. La pile des opérandes est vide quand la frame est créée. La JVM fournit des instructions pour charger des constantes ou des valeurs, depuis les variables locales ou les champs, sur la pile des opérandes. D'autres instructions prennent des opérandes de la pile des opérandes, effectuent une opération sur celles-ci, et remettent le résultat sur la pile des opérandes. Comme nous l'avons vu juste au-dessus avec les variables locales, la pile des opérandes est aussi utilisée pour préparer les paramètres à passer aux méthodes, et pour recevoir les résultats de celles-ci.

Par exemple, l'instruction *iadd* additionne 2 valeurs de type *int*.Elle requiert que les valeurs int devant être additionnées, soit lesdeux valeurs au sommet de la pile des opérandes. Elles auront étéplacées ici par de précédentes instructions (action sur la pile ditde'push', par exemple l'instruction *iload*). Les deux valeurssont alors retirées de la pile (action dit de 'pop'), puisadditionnées, et leur somme est placée au sommet de la pile desopérandes (de nouveau un 'push').

Pile début -> instruction -> Pile fin
..., val1, val2 -> xadd -> ..., result

Préparation aux instructions de la JVM

Une instruction de la machine virtuelle Java consiste en un ***opcode*** sur1 byte (1 octet) spécifiant l'opération à effectuer, suivie par zéro ouplusieurs opérandes servant d'arguments ou de données qui serontutilisés par l'opération. Plusieurs instructions n'ont pas d'opérandeet consistent donc simplement en un opcode.

Voici un pseudo-code de l'exécution du byte-code :

```
do {
  fetch an opcode;
  if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

Le nombre, le type et la taille des opérandes sont déterminés par l'opcode.

Les types de la JVM

La plupart des instructions de la JVM 'encodent' les informations de type à propos de l'opération qu'elles effectuent dans leur nom. Par exemple, l'instruction *iload* charge le contenu d'une variable locale, qui doit être un int, au dessus de la pile des opérandes. L'instruction *fload* fait de même pour une value de type float. Plusieurs instructions peuvent avoir la même fonction, mais des opcodes différents; par exemple, c'est le cas de *iload, fload, dload, ...*, tous chargent une variable locale sur la pile des opérandes.

De cette façon, pour la majorité des instructions typées, le type de l'instruction est représenté explicitement dans le nom de l'opcode par une lettre : ***i*** pour une opération sur un int, ***l*** pour une opération sur un long, ***s*** pour une opération sur un short, ***b*** pour une opération sur un byte, ***c*** pour une opération pour un char, ***f*** pour une opération pour un float, ***d*** pour une opération sur un double, et enfin ***a*** pour une opération sur une référence.

Quelques instructions pour lesquelles le type n'est pas ambigue (c'est à dire qu'un seul type est autorisé, par exemple les opcodes *putfield, jsr, ...*) n'ont pas de lettre spécifique dans leur nom.

La longueur de 1 byte (1 octet -> 8 bits) d'un opcode empêche d'avoir un jeu d'instructions supérieur à 256 opcodes. Par conséquent ce faible nombre d'instructions réduit le nombre de types supportés pour certaines opérations (on ne va pas trouver toutes les instructions du jeu d'instructions pour chaque type de données). En d'autres termes, le jeu n'est pas orthogonal, des instructions supplémentaires peuvent être utilisées pour convertir les types de données non supportés en types de données supportés. Cela peut réduire les performances dans les codes de personnes non initiées, mais cela est nécessaire pour garder un jeu d'instructions et des fichiers *.class* compacts.

Le tableau suivant résume le support des types des instructions de la JVM. Une instruction spécifique, avec l'information de type, est construite en remplaçant le T dans la colonne des opcodes par la lettre du type de la colonne. Si la colonne de type pour certains modèles d'instructions est blanche, cela signifie qu'il n'existe pas de support pour ce type concernant cette opération. Par exemple, il y a une instruction de chargement (Tload) pour le type int (iload), mais pas pour le type byte.

La plupart des opérations ne supportent pas les types *boolean, byte, char, et short*. Par conséquent, les valeurs de ces types, sont implicitement converties en type *int* à la compilation ou à l'exécution, et ensuite exécutées par des instructions de type *int*.

Voir les spécifications pour plus de détails.

Opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

Certaines instructions de la JVM comme *pop* et *swap* opèrent sur la pile des opérandes sans faire attention à leur type; cependant, de telles instructions sont contraintes d'être utilisées avec des valeurs d'une certaine catégorie, donnée dans le tableau ci-dessous :

Type	***Type calculé***	***Categorie***
`boolean`	`int`	1
`byte`	`int`	1
`char`	`int`	1
`short`	`int`	1
`int`	`int`	1
`float`	`float`	1
`reference`	`reference`	1
`returnAddress`	`returnAddress`	1
`long`	`long`	2
`double`	`double`	2

Par exemple, les instructions *pop et dup* sont utilisées pour les types de la catégorie 1, et les instructions *pop2 et dup2* sont utilisées pour les types de la catégorie 2. On comprendra aisément que les types ayant une taille de 64 bits (8 octets) appartiennent à la catégorie 2.

Charger et stocker des instructions

Les instructions de chargement et de stockage transfèrent les valeurs entreles variables locales et la pile d'opérandes d'une frame de la JVM :

- Charge une variable locale sur la pile des opérandes : **iload, iload_N, lload, lload_N, fload, fload_N, dload, dload_N, aload, aload_N**.
- Stocke une valeur depuis la pile des opérandes vers une variable locale : **istore, istore_N, lstore, lstore_N, fstore, fstore_N, dstore, dstore_N, astore, astore_N**.
- Charge une constante sur la pile des opérandes : **bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, iconst_N, lconst_N, fconst_N, dconst_N**.
- Avoiraccèsà plus de variables locales en utilisant un index plus grand, oua uneopérande immédiate plus large (par exemple, avoir un long au lieu de unint) : **wide**.

Les instructions qui accèdent aux champs des objects et aux éléments des tableaux, transfèrent aussi des données depuis et vers la pile des opérandes.

Le format des instructions avec des lettres génériques N (par exemple, *iload_N*) dénote des familles d'instructions (avec les membres iload_0, iload_1,iload_2 et iload_3 dans le cas de iload_N). De telles familles d'instructions sont des spécialisations d'une instruction générique (iload) qui ne prend qu'un paramètre. Pour les instructions génériques, l'opérande est implicite et l'instruction *iload_0* signifie la même chose que *iload*.

Les instructions arithmétiques

Les instructions arithmétiques calculent un résultat, ce sont typiquement des fonctions qui prennent 2 valeurs sur la pile des opérandes, et qui mettent le résultat au-dessus de cette même pile une fois l'opération effectuée. Il existe 2 types principaux d'instructions arithmétiques : ceux opérant sur des valeurs entières et ceux opérant sur des valeurs à virgule flottante. Comme nous l'avons vu un peu plus haut, il n'existe pas de support direct pour une arithmétique entière sur les types *byte, short, char, et boolean*; ces opérations sont gérées par les instructions opérant sur le type *int*. Les instructions arithmétiques sont les suivantes :

- Addition : **iadd, ladd, fadd, dadd**.
- Soustraction : **isub, lsub, fsub, dsub**.
- Multiplication : **imul, lmul, fmul, dmul**.
- Division : **idiv, ldiv, fdiv, ddiv**.
- Reste (modulo) : **irem, lrem, frem, drem**.
- Négation : **ineg, lneg, fneg, dneg**.
- Shift (décalage de bit) : **ishl, ishr, iushr, lshr, lshl, lushr**.
- OR (bit à bit) : **ior, lor**.
- AND (bit à bit) : **iand, land**.
- XOR (bit à bit) : **ixor, lxor**.
- Incrémentation : **iinc**.
- Comparaison : **dcmpg, dcmpl, fcmpg, fcmpl, lcmp**.

Les instructions entière et à virgule flottante diffèrent aussi dans leurcomportement pour ce qui est de l'overflow et la division parzéro. Les opérations sur les types entiers ne renvoient pas d'overflow; la seuleerreur pour ces opérations (idiv, ldiv, irem, lrem) est la division par zéro qui lance un ArithmeticException (cf le paragraphe sur lesexceptions en fin d'article).

Les comparaisons sur les valeurs de type *long (lcmp)* effectue une comparaison signée. Les comparaisons sur les types à virgule flottante (*dcmpg, dcmpl, fcmpg, fcmpl*) sont effectuées en utilisant la comparaison IEEE 754.

Instructions de conversion de types

Les instructions de conversion de types permettent la conversion entre les différents types numériques de la JVM. Celles-ci peuvent être utilisées pour implémenter une conversion explicite. La conversion d'une valeur d'un type vers un autre type possédant un intervalle de valeurs plus petit, nécessite une conversion explicite. Par exemple la conversion d'un *long* vers un *int*.

Voici les différentes conversions réalisables (implicites et explicites) par la JVM :

La JVM supporte directement les conversions d'élargissement d'intervalle suivantes :

- *int* vers long, float ou double
- *long* vers float ou double
- *float* vers double

Les instructions de conversion numerique d'élargissement sont **i2l, i2f, i2d, l2d, et f2d**. La signification de ces opcodes est relativement simple à comprendre. Par exemple, l'instruction *i2f* convertie une valeur de type int vers une valeur de type float.

A noter que la conversion numérique d'élargissement est automatique pour le langage de programmation Java (le programmeur n'a pas besoin de le faire explicitement). Egalement, la conversion n'existe pas pour les types byte, char, et short vers le type int. Car comme spécifié plus haut, les valeurs de type byte, char, et short sont élargies intrinsèquement vers le type int, par une conversion implicite.(Rappel: la conversion d'un type numérique vers un booléen n'est pasautorisé en Java.)

La JVM supporte également les conversions numériques de rétrécissement d'intervalles :

- *int* vers byte, short ou char
- *long* vers int
- *float* vers int ou long
- *double* vers int, long ou float

Les instructions de conversion numérique de 'rétrécissement' sont **i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, et d2f**. Les règles de conversion sont celles du langage de programmation Java.Une conversion de ce type peut entraîner un résultat ayant un signe différent de la valeur initiale, et/ou une perte de précision, mais cela reste pour des cas particuliers. Veuillez vous reporter aux spécifications pour de plus amples informations sur les erreurs de conversion.

Manipulation et création d'objet

Bien que les instances de classe et les tableaux soient des objets, la JVM crée et manipule les instances de classe et les tableaux de façon distincte avec un jeu d'instructions propre à chacun :

- Création d'une nouvelle instance de classe : **new**.
- Création d'un nouveau tableau : **newarray, anewarray, multianewarray**.
- Accès aux champs d'une classe (champs statiques, aussi appelés variables de classe) et les champs d'instance de classe (champs non statiques, aussi appelés variables d'instance) : **getfield, putfield, getstatic, putstatic**
- Chargement d'un élément d'un tableau sur la pile des opérandes : **baload, caload, saload, iaload, laload, faload, daload, aaload**.
- Stocker une valeur depuis la pile des opérandes en tant qu'élement de tableau : **bastore, castore, sastore, iastore, lastore, faastore, dastore, aastore**.
- Obtenir la longueur d'un tableau : **arraylength**
- Vérifier les propriétés d'instances de classe ou de tableaux : **instanceof, checkcast**.

Les instructions de gestion de la pile des opérandes

Un certain nombre d'instructions est fourni pour la manipulation directe de la pile des opérandes : **pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2, swap**.

Les instructions de controle

Les instructions de branchements conditionnels ou inconditionnels permettent à la JVM de continuer de s'exécuter avec une instruction différente de celle suivant l'instruction de branchement. Les instructions de branchement sont :

- Branchement conditionnel : **ifeq,iflt, ifle, ifne,ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne,if_icmplt,if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne**.
- Branchement conditionnel composé : **tableswitch, lookupswitch**.
- Branchement inconditionnel : **goto, goto_w, jsr, jsr_w, ret**.

Toutes ces comparaisons sont effectuées en tenant compte du signe.

Instructions d'invocation de méthode et retour

Les instructions suivantes invoquent une méthode :

- **Invokevirtual** invoque une méthode d'une instance d'un objet, en appelant la bonne méthode virtuelle de l'objet. Ceci est le comportement normal dulangage de programmation Java.
- **invokeinterface** invoque une méthode qui est implémentée par une interface, en cherchant lesméthodes implémentées par cet objet pour trouver la méthode appropriée.
- **Invokespecial** invoque une instance qui requiert un traitement special, c'est à dire une méthode d'initialisation d'instance, une méthode privée, ou une méthode de la super classe.
- **invokestatic** invoque une méthode de classe (statique) dans la classe nommée.

Les instructions de retour de méthodes, qui sont distinguables par le type de retour, sont **ireturn** *(utilisée pour retourner des valeurs de type boolean, byte, char, short, ou int)*, **lreturn, freturn, dreturn, et areturn**. De plus, l'instruction *return* est utilisée pour retourner depuis des méthodes déclarées void, des méthodes d'initialisation d'instance, et des méthodes d'initialisation de classe ou d'interface.

Lancement d'exception

Une exception est lancée 'programmaticalement' en utilisant l'instruction **athrown**. Les exceptions peuvent aussi être lancées par d'autres instructions de la JVM lorsqu'une condition inhabituelle est détectée (par exemple, la division par zéro).

Définitions :

Méthode d'instance : une sous-routine ou une fonction appartenant à l'objet courant. Les méthodes font toujours parties d'une classe en Java, vous ne pouvez pas avoir de méthodes seules comme le permet le C ou le C++. Une méthode d'instance a accès à toutes les variables de l'instance, à toutes les autres méthodes d'instances, et aux méthodes et variables statiques.

Méthode de classe :une méthode statique dans une classe. Elle n'a pas accès aux variables d'instance, seulement aux variables statiques de cette classe. De plus, elle ne peut pas invoquer de méthode d'instance (non statique) à moins qu'elle ne possède une référence vers cet objet.