Génération de bytecode Java

michel.chilowicz@univ-mlv.fr

Sous licence Creative Commons By-NC-SA

Photo par Linda Hoover (licence CC-By-NC)

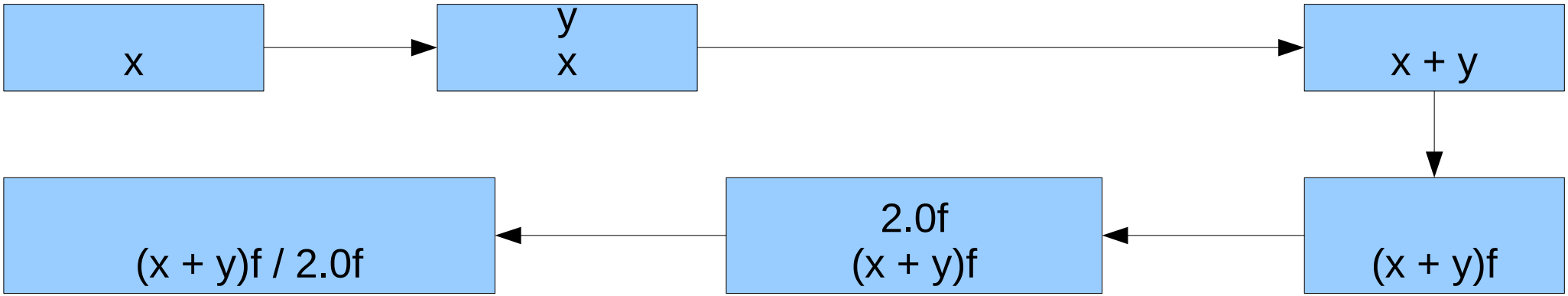
Modèle de machine à pile

- Les opérandes à manipuler sont placées sur une pile
- Une instruction
  - dépile zéro, un ou plusieurs opérandes ;
  - réalise son opération ;
  - puis empile son résultat (zéro, un ou plusieurs mots).
- En bytecode Java, opérande = mot de 32 bits
  les opérandes long et double requièrent 2 cases de pile
- Autres exemples de langages à pile :
  - langage RPN (calculatrices HP)
  - langage de description de page PostScript
  - langage Forth

Exemples d'opérations sur pile

- Calculons la moyenne de deux variables locales 0 et 1 :
  1)iload_0 : on empile la variable locale 0
  2)iload_1 : on empile la variable locale 1
  3)iadd : on dépile 2 éléments et on empile leur somme
  4)i2f : on dépile 1 élément int et on empile sa conversion en float
  5)fconst_2 : on empile la constante 2.0f
  6)fdiv : on divise deux éléments dépilés et on empile le résultat

Variables locales
0:x
1:y



Anatomie d'une classe Java

- Nombre magique : 0xCAFEBABE
- Version de format de classe
- Réservoir à constantes
- Drapeaux d'accès de la classe (statique, publique, abstraite...)
- Super-classe (@référence de classe dans le réservoir)
- Interfaces implantées par la classe (idem)
- Champs de la classe
- Méthodes de la classe
- Attributs de la classe

➔ http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

Réservoir à constantes

- Constantes d'une classe stockées dans une section spécifique : le réservoir à constantes
- Types de constantes du réservoir :
  - String en UTF-8
  - Entier 32 bits en grand-boutisme
  - Flottant 32 bits IEEE 754
  - Long 64 bits en grand-boutisme
  - Double 64 bits IEEE 754
  - Référence de classe : @String désignant la classe
  - Référence de String : @String
  - Référence de champ : @référence de classe, @descripteur de nom et type
  - Référence de méthode : @référence de classe, @descripteur de nom et type
  - Référence d'interface : @référence de classe, @descripteur de nom et type
  - Descripteur de nom et type : @String pour le nom, @String pour le type

Descripteurs de type

- Descripteurs pour les types primitifs
  - boolean : Z, byte : B, char : C, short : S, int : I, long : J , float : F, double : D
- Descripteurs pour les classes :
  - Classe : *Lpackage/subpackage/.../MyClass ;*
  - Tableau : préfixe [(exemple : double[][] : *[[D*)
- Descripteurs pour les méthodes : *(typearg1typearg2...)typeretour*
  - ➔ Exemple : méthode long[] f(int i, Class[] classes) *(I[Ljava/lang/Class;)[J*

Instructions de bytecode Java

- Une instruction = 1 octet
  - En pratique jusqu'à 256 instructions (certains octets réservés)
  - Nombreuses variantes d'instructions pour éviter l'usage de paramètres :
    - iconst_m1 : charge l'entier -1 sur la pile
    - iconst_0 : charge l'entier 0 sur la pile
    - ...
  - Certaines instructions nécessitent d'être suivies d'octets pour leurs paramètres :
    - get_field (0xb4) suivi de deux octets a et b pour spécifier le champ d'indice a << 8 + b (élément de haut de pile = référence de l'instance)
- Notation utilisée :
  op <param1> <param2> … [operande1, operande2, ...]

Méthode Java

- Drapeaux d'accès (public, private, protected, static, final, synchronized, native, abstract, strict)
- Nom et descripteur (types des arguments, type de retour)
- Tableau d'attributs :
  - Code : bytecode Java (ssi non abstract ou native)
    - Taille de la pile
    - Taille de la table de variables locales
    - Tableau de bytecode Java
    - Table d'exception handlers : spécifie où sauter lorsqu'une exception d'un certain type est levée entre deux offsets de bytecode
  - Exceptions : clause throws
  - SourceFile : chemin vers le fichier original
  - LineNumberTable : correspondance entre bytecode et source original (debug)
  - LocalVariableTable : table des noms des variables locales dans le source original
  - Runtime{Visible, Invisible}Annotations : table des annotations
  - ...

Désassemblage avec javap -c

```
public class PrintArg0
{
	public static void main(String[] args)
	{
		try {
			System.out.println(args[0]);
		} catch (IndexOutOfBoundsException e)
		{
			System.err.println(
				"args[0] is not defined");
		}
	}
}
```

```
Compiled from "PrintArg0.java"
public class PrintArg0 extends java.lang.Object{
public PrintArg0();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   getstatic   #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   aload_0
   4:   iconst_0
   5:   aaload
   6:   invokevirtual   #3; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   9:   goto  21
   12: astore_1
   13: getstatic   #5; //Field java/lang/System.err:Ljava/io/PrintStream;
   16: ldc   #6; //String args[0] is not defined
   18: invokevirtual   #3; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   21: return
  Exception table:
   from   to  target type
     0     9    12   Class java/lang/IndexOutOfBoundsException
}
```

Chargement de constantes

- Famille const [0x01-0x0f] :
  - aconst_null : empile la référence nulle
  - iconst_{m1, 0, 1, 2, 3, 4, 5} : empile une constante entière
  - lconst_{0,1} : empile 0L ou 1L
  - fconst_{0,1,2} : empile 0.0f, 1.0f ou 2.0f
  - dconst_{0,1} : empile les doubles 0.0 ou 1.0
- Famille ldc [0x12-0x14] :
  - ldc #index : empile une constante du réservoir
  - ldc_w #index1 #index2 : empile la constante d'indice index1 << 8 + index2 du réservoir
  - ldc2_w #index1 #index2 : empile la constante (double ou long) d'indice index1 << 8 + index2 du réservoir
- Manipulation inutile des offsets de réservoir avec un assembleur

Manipulation de variables locales

- Table de variables locales maintenue pour chaque appel de fonction dans la pile d'appel
  - La variable #0 désigne le 1er argument de la méthode, la variable #1 le 2ème argument...
  - Pour les méthodes non-statiques, la variable #0 est *this*
- iload, lload, fload, dload et aload <index> chargent un entier, un long, un float, un long ou une référence dans la pile
- Variantes avec indices 0, 1, 2, 3 incorporés (iload_0, iload_1, …)
- Un long ou un double occupent 2 emplacements de variable locale ou sur la pile
- {i, l, f, d, a}store suivi de l'indice de variable pour stocker l'élément de haut de pile dans la table des variables
- Variantes avec indices incorporés *store_0, *store_1, *store_2, *store_3

Manipulation de tableaux

- {i,l,f,d,a,b,c,s}aload [arrayref, index] : chargement sur la pile d'un primitif ou référence
- {i,l,f,d,a,b,c,s}astore [arrayref, index, value] : mise à jour d'une cellule d'un tableau
- Exemple : affecter une cellule de matrice d'entier a[x][y] = z
  - aload_0 : charge a
  - aload_1 : charge x
  - aaload : empile la référence de a[x]
  - aload_2 : charge y
  - aload_3 : charge z
  - iastore : met à jour la cellule de y de a[x] avec la valeur z

Opérations arithmétiques et binaires

- Opérations arithmétiques binaires {i,l,f,d}{add, sub, mul, div, rem} [value1, value2]
- Opération unaire de négation : {i,l,f,d}neg [value1]
- Opérateur de décalage de bits : {i,l}sh{l,r} [value1, value2]
- Opérateur de décalage à droite >>> {i,l}ushr [value1, value2]
- Opérateurs sur champs de bits :{i,l}{and, or, xor} [value1, value2]
- iinc <index> <const> : incrémente la variable locale entière #index de const
- {i, l, f, d}2{i, l, f, d} [value] : conversion inter-types primitifs long, float et double
- i2{b, c, s} [value] : un entier peut être converti en byte, char ou short
- lcmp [value1, value2] : compare 2 longs value1 et value2 (résultat : entier -1, 0 ou 1)
- {f,d}cmpl [value1, value2] : comparaison de 2 flottants IEEE 754 (-1 si au moins un NaN)
- {f,d}cmpg [value1, value2] : idem mais 1 si au moins un NaN

Instructions de saut

- Toutes les instructions de saut sont suivies de deux octets i1 et i2 indiquant l'offset de saut i1 << 8 + i2
- En pratique, on utilise un langage assembleur intermédiaire avec des labels (convertis ensuite en offsets)
- Saut par comparaison d'un entier avec 0 : if{eq,ne,lt,ge,gt,le}
- Saut par comparaison de deux valeurs entières de la pile : if_icmp{eq,ne,lt,ge,gt,le}
- Saut par test de nullité d'une référence : ifnull, ifnonnull
- Saut par comparaison de deux références de la pile : if_acmp{eq,ne}
- Saut non-conditionnel : goto
- lookupswitch et tableswitch permettent de gérer des switch-case ainsi que des recherches dichotomiques sur un entier

Création d'objets et manipulation de champs

- new <@class> : crée une instance de classe
- newarray <primtype> [len] : crée un nouveau tableau de longueur len d'éléments primitifs
- anewarray <@class> [len] : crée un tableau de références de longueur len pointant vers des instances de type classe
- multianewarray <@class> <n> [dim1len, dim2len...] : crée un tableau de références à n dimensions
- getstatic <@field> : récupère le contenu d'un champ statique
- putstatic <@field> [value] : place la valeur spécifiée dans le champ statique
- getfield <@field> [ref] : récupère le contenu d'un champ d'une instance
- putfield <@field> [ref, value] : affecte le contenu d'un champ d'une instance

Création d'objets et manipulation de champs

- new <@class> : crée une instance de classe
- newarray <primtype> [len] : crée un nouveau tableau de longueur len d'éléments primitifs
- anewarray <@class> [len] : crée un tableau de références de longueur len pointant vers des instances de type classe
- multianewarray <@class> <n> [dim1len, dim2len...] : crée un tableau de références à n dimensions
- getstatic <@field> : récupère le contenu d'un champ statique
- putstatic <@field> [value] : place la valeur spécifiée dans le champ statique
- getfield <@field> [ref] : récupère le contenu d'un champ d'une instance
- putfield <@field> [ref, value] : affecte le contenu d'un champ d'une instance

Invocation de méthodes

- Les types d'invocation :
  - Invocation de méthode statique
  - Invocation spéciale : utilisée pour appeler un constructeur, une méthode privée ou une méthode d'un ancêtre
  - Invocation de méthode virtuelle : appel d'une méthode normale sur une instance
  - Invocation de méthode d'interface
  - Invocation dynamique (JSR292) : appel avec connaissance du type d'objet seulement à l'exécution
- Instructions :
  - invoke{static, special, virtual, interface, dynamic} <@class/method> [ref, arg1, arg2, …] (ref n'est pas demandé pour static)

Instantiation d'objet

- Étapes de l'instantiation d'un objet :
  - Création de l'instance avec *new chemin/Classe*
  - Appel d'un des constructeurs de l'objet avec invokespecial chemin/Classe/<init>(types)V
- Exemple : new URL(« http://igm.univ-mlv.fr/ »)
  - new java/net/URL
  - dup (duplique la référence pour le constructeur)
  - ldc « http://igm.univ-mlv.fr » (charge la constante du réservoir)
  - invokespecial java/net/URL/<init>(Ljava/lang/String;)V
  - astore 1 (on stocke la référence dans la variable #1)

Autres instructions de bytecode

- Lancement d'une exception avec athrow [ref] (le bas de la pile est effacé)
- Retour d'une méthode avec {i, l, f, d, a}return [ref]
- Cast d'une référence avec checkcast <@class> [ref] : remplace la référence en haut de la pile par une référence castée sinon lève un ClassCastException.
- instanceof <@class> [ref] : empile 1 si ref est une instance de class, 0 sinon.
- arraylength [ref] : récupère la longueur d'un tableau
- monitor{enter, exit} [ref] : acquiert ou libère un moniteur sur un objet
- Instructions de manipulation de pile :
  - dup [value] : duplique l'élément de haut de pile (dup2 pour 2 éléments)
  - pop [value] : enlève l'élément value du haut de la pile (pop2 pour 2 éléments)
  - swap [value] : échange les 2 éléments en haut de pile

Récapitulatif d'instructions essentielles... pour IR2012

- Chargement de constantes int et String : ldc
- Chargement et sauvegarde de variables locales int (entier et booléen) et référence (vers String) : *{i,a}load*, *{i,a}store*
- Les fonctions sont des méthodes statiques privées d'une classe appelées par *invokestatic*
- Il faudra récupérer System.out (getstatic) et appeler println (invokevirtual)
- Opérations sur les int : *iadd*, *isub*, *imul*, *idiv*, *ineg*
- Saut conditionnel pour booléen faux (entier 0) : *ifeq*
- Retour d'un entier (*ireturn*) ou d'une référence de String (*areturn*)

print « Hello World IR 2012 »

```
.class public HelloWorld
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
.limit locals 10
.limit stack 10
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Hello World IR2"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return
.end method
```

Nombre de Fibonacci en Jasmin

```
def int fibo(int n) :
    if n < 2 :
        return 1
    return fibo(n-1)
    + fibo (n-2)

print(fibo(10)) ;
```

```
class public Fibonacci
.super java/lang/Object

.method public static fibo(I)I
.limit locals 1
.limit stack 3
iload 0
ldc 2
if_icmpge recursivecall
ldc 1
ireturn
recursivecall:
iload 0
ldc 1
isub
dup
invokestatic Fibonacci/fibo(I)I
swap
ldc 1
isub
invokestatic Fibonacci/fibo(I)I
iadd
ireturn
.end method
```

```
.method public static print(I)V
.limit locals 1
.limit stack 2
getstatic java/lang/System.out
Ljava/io/PrintStream;
iload 0
invokevirtual
java/io/PrintStream/println(I)V
return
.end method

.method public static
main([Ljava/lang/String;)V
.limit locals 1
.limit stack 1
ldc 10
invokestatic Fibonacci/fibo(I)I
invokestatic Fibonacci/print(I)V
return
.end method
```

Vérification de classes

- Du bytecode incorrect peut corrompre la JVM
- Des vérifications sont nécessaires :
  - Pour ne pas dépasser les limites de la pile
  - Assurer une cohérence des types des opérandes avec les opérateurs
  - Avoir des arguments valides pour chaque opération
  - Garantir la validité des offsets atteints par saut (dans la même méthode, au début d'une instruction)
  - Respecter la visibilité des champs
  - ...
- Suivi des différents chemins d'exécution pour vérifier l'état de la pile
- Sanction : lancement de java.lang.VerifyError au chargement de la classe

Exercices en Jasmin

- Écrire une méthode calculant la factorielle d'un entier (que l'on exprimera par un long)
  - En récursif
  - En itératif
- Réaliser un jeu de divination de nombre :
  - Tirage au sort d'un nombre *n* aléatoire
  - Boucle de lecture sur l'entrée standard d'entier *i*
    - Si i != n, on affiche le résultat de la comparaison
    - Si i == n, la partie est terminée

➔ Référence Jasmin : https://www.vmth.ucdavis.edu/incoming/Jasmin/jvmref.html

ASM

- Framework permettant la manipulation de bytecode Java
  - Génération « ex-nihilo » de bytecode Java
  - Modification de bytecode existant
- Utilise le design-pattern de visiteur
  - *ClassReader* lit du bytecode Java qu'il traduit en appel de visites sur un visiteur
  - *ClassWriter* est un visiteur dont les visites produisent du bytecode