java.lang.invoke (invokedynamic)

Rémi Forax

Java comme plateforme

Java a évolué depuis la version 1.0, le language est très utilisé mais la plateforme (la machine virtuelle) l'est encore plus

On dénombre plus de 240 langages qui tourne sur la plateforme Java
Scala, Kotlin, Gosu, Rhino/Nashorn(javascript), Groovy(java?), JRuby(ruby), Jython(python), Clojure(lisp?), etc...

Les languages dynamiques

Beaucoup de ces languages sont des langages dynamiquement typé pourtant le bytecode Java est statiquement typé

- Language statiquement typé

  int x = ... int y = ... x + y est traduit en iadd

- Language dynamiquement typé

  x = ... y = ... x + y est soit iadd, ladd, fadd, dadd, String.concat(), etc

Idée !

Ajouter un nouveau bytecode

- qui permet de choisir quel code appeler à l'exécution

  => trop lent, car cela nécessite de boxer les arguments puis d'appeler un code générique
  (donc non spécifique à une opération)

- qui permet insérer des tests qui indiquera quel code appelé

  => mieux mais risque d'ajouter trop de code

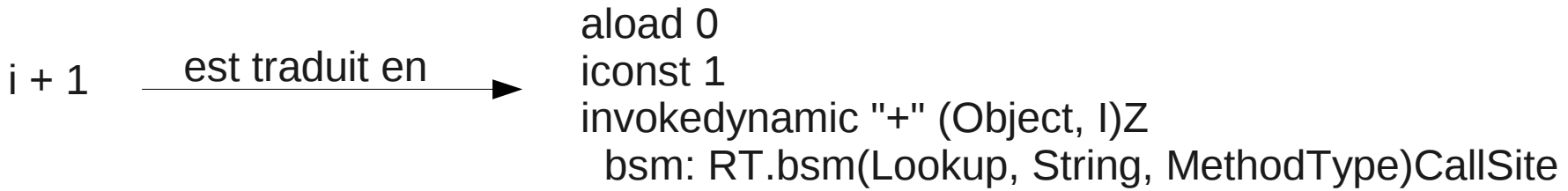
- qui permet aussi de changer/ajouter des tests si certains ne sont pas suffisant ultérieurement

  => cool, cela veut dire que tous les tests n'ont pas besoin d'être créer mais uniquement ceux pour lequel le code est utilisé

Comment ça marche ?

Lors du premier appel à invokedynamic, la VM appel la méthode de bootstrap qui doit créer un object CallSite qui contiendra un pointer de fonction vers la fonction qui sera appelé pour les appels suivant

De plus, la méthode CallSite.setTarget permet de changer le pointeur à n'importe quel moment



Pointeur de fonction

j.l.invoke.MethodHandle est un pointeur de fonction + un MethodType qui représente la signature de la fonction qui peut être appelée

une méthode est une fonction dont le type de this est le premier paramètre, une méthode statique est une fonction

MethodHandle possède deux méthodes magiques invoke() et invokeExact() qui permettent d'appeler la méthode pointée

Lookup

Contrairement à j.l.reflect.Method qui test la sécurité lors d'un appel, j.l.invoke.methodHandle test la sécurité une fois lors de la création

Un objet MethodHandles.Lookup contient les droits associés à une classe.

- MethodHandles.lookup() crée un objet Lookup avec les droits de la classe qui appel la méthode lookup()
- MethodHandles.publicLookup() est un objet lookup qui ne voit que les classes/méthodes publiques

Lookup

- find[Static][Getter|Setter] trouve un champ et le voit comme un appel de méthode getter ou setter
- findStatic trouve une méthode static
- findVirtual trouve une méthode d'une classe/interface qui nécessite un appel virtuel
- findSpecial appel un constructeur mais avec l'objet déjà construit
- findConstructor appel un constructeur
- unreflect*() prend un objet j.l.reflect et le convertit en MethodHandle

Exemple

- Un exemple de findVirtual suivi d'un invokeExact
- Le cast n'est pas un cast pour la VM mais uniquement pour le compilateur car il ne peut pas deviner le type de retour

```
public static void main(String[] args) throws Throwable {
  Lookup lookup = MethodHandles.lookup();
  MethodHandle mh = lookup.findVirtual(String.class, "charAt",
     MethodType.methodType(char.class, int.class));

  mh.invokeExact("foo", 1);  // WrongMethodTypeException

  int i = (int)mh.invokeExact("foo", 1);  // WrongMethodTypeException

  char c = (char)mh.invokeExact("foo", 1);  // Ok
  System.out.println(c); // o
}
```

2 sémantiques d'appels

mh.invokeExact(desc)

- ne marche que si mh.type() et le descripteur sont égaux
- appel très rapide garantie (cmp + call)

mh.invoke(desc)

- essaye de faire les conversions sans perte et boxing/unboxing
  - si mh est marqué avec asVarargsCollector(), fait un appel varargs
- appel peut être lent, dépend des conversions

Exemple

- Un appel en utilisant invoke() fait les adaptations si besoin
- Si les descripteurs match, invoke doit être aussi efficace que invokeExact

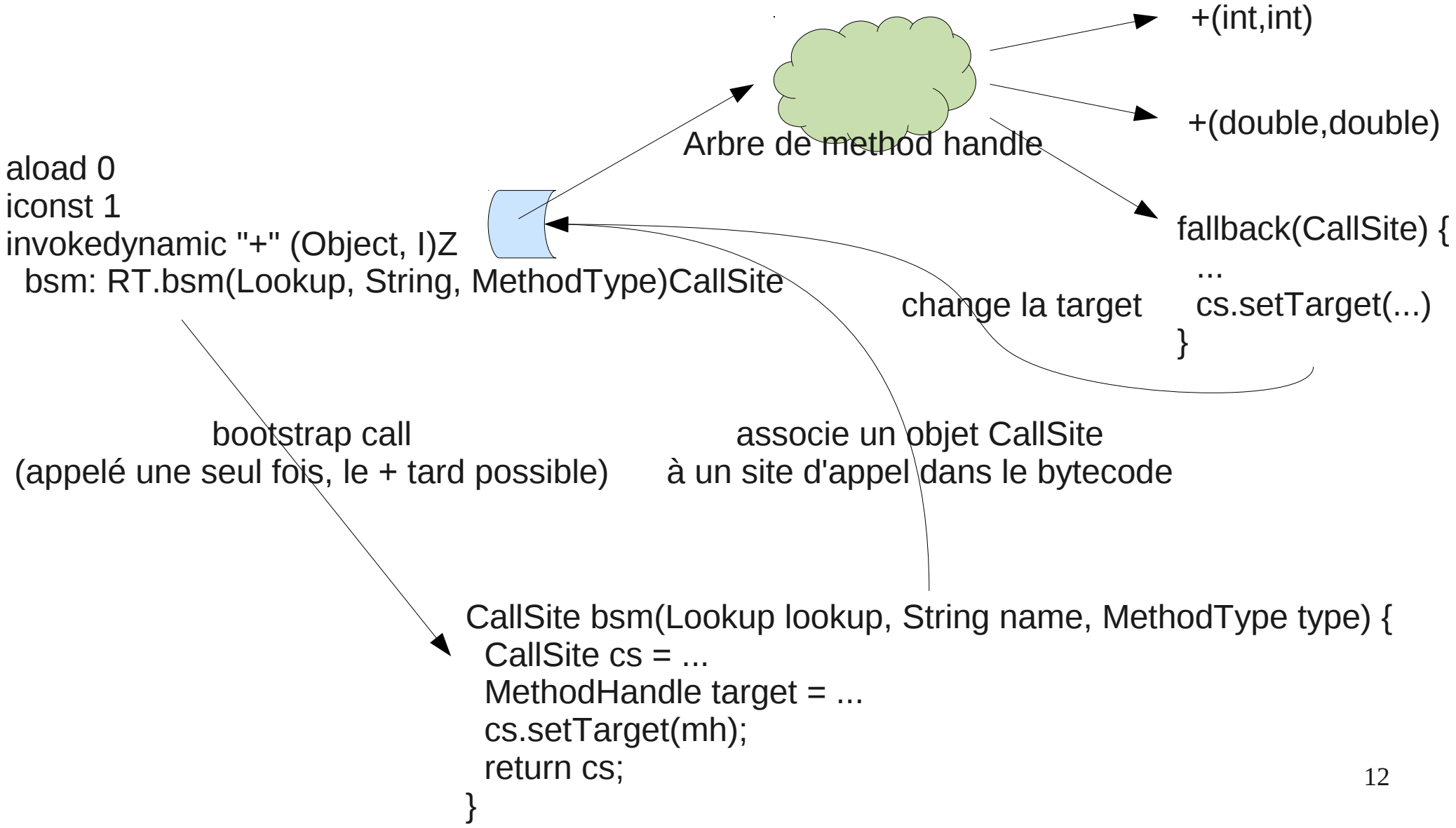
```
public static void main(String[] args) throws Throwable {
  Lookup lookup = MethodHandles.lookup();
  MethodHandle mh = lookup.findVirtual(String.class, "charAt",
      MethodType.methodType(char.class, int.class));

  mh.invoke("foo", 1);   // ok

  int i = (int)mh.invoke("foo", 1);   // ok

  char c = (char)mh.invoke("foo", 1);  // ok
  System.out.println(c); // o
}
```

En résumé


+(int,int)
+(double,double)
Arbre de method handle
aload 0
iconst 1
invokedynamic "+" (Object, I)Z
bsm: RT.bsm(Lookup, String, MethodType)CallSite
fallback(CallSite) {
...
cs.setTarget(...)
}
change la target
bootstrap call
(appelé une seul fois, le + tard possible)
associe un objet CallSite
à un site d'appel dans le bytecode
CallSite bsm(Lookup lookup, String name, MethodType type) {
CallSite cs = ...
MethodHandle target = ...
cs.setTarget(mh);
return cs;
}

Combinateurs

Il est possible de créer des MethodHandles à partir de MethodHandle

Methodes sur un MethodHandle

asType(), bindTo(), asCollector()/asSpreader()/asVarargsCollector, invokeWithArguments()

Methodes statique sur MethodHandles

guardWithTest(), dropArguments/insertArguments(), filterArguments()/filterReturnValue(), constant(), etc.

Convertions

Permet d'adapter un method handle à une signature particulière

- MH.asType()
  - Conversions sans perte
    - Cast, primitive, vers void, etc
- MHs.explicitCastArguments()
  - Autre conversions
    - double -> int, void -> Object, etc

Exemple

asType() permet de faire l'adaptation à une signature particulière

```
public static void main(String[] args) throws Throwable {
  Lookup lookup = MethodHandles.lookup();
  MethodHandle mh = lookup.findVirtual(String.class, "charAt",
     MethodType.methodType(char.class, int.class));

  mh.invokeExact("foo", 1);   // WrongMethodTypeException

  MethodHandle mh2 = mh.asType(
     mh.type().changeReturnType(void.class));

  mh2.invokeExact("foo", 1);   // ok
}
```

Curryfication

- Un pointeur de fonction à plusieurs arguments peut être vu comme un fonction de fonction avec moins d'argument + les valeurs des arguments
  - MH.bindTo(Object)
  - MHs.insertArguments(MH, int, Object... args)
- Si on bind le receveur d'un appel virtuel alors l'appel n'est plus virtuel

Exemple

Mhs.insertArguments() permet remplacer certain arguments par des constantes

```
public static void main(String[] args) throws Throwable {
  Lookup lookup = MethodHandles.lookup();
  MethodHandle mh = lookup.findVirtual(String.class, "charAt",
     MethodType.methodType(char.class, int.class));

  MethodHandle mh2 = MethodHandles.insertArguments(mh, 0, "foobar");

  char c = (char)mh2.invokeExact(3);  // un seul argument
  System.out.println(c); // b
}
```

Conversion varargs

- asCollector() permet de voir un method handle prenant en paramètre un tableau comme un method handle à plusieurs arguments
- asSpreader() permet de voir un method handle à plusieurs argument comme un method handle prenant en paramètre un tableau
- asVarargsCollector() marque le method handle comme un varargs pour que invoke() mettent les arguments dans un tableau

Exemple

asCollector() met les arguments dans un tableau, asSpreader le contraire.

```
public static void main(String[] args) throws Throwable {
  Lookup lookup = MethodHandles.lookup();
  MethodHandle mh = lookup.findVirtual(String.class, "charAt",
      MethodType.methodType(char.class, int.class));

  MethodHandle mh2 = mh.asSpreader(Object[].class, 2);
  char c = (char)mh2.invokeExact(new Object[]{ "foo", 0});
  System.out.println(c);  // f

  MethodHandle mh3 = mh2.asCollector(Object[].class, 2);
  c = (char)mh3.invokeExact((Object)"foo", (Object)0);
  System.out.println(c);  // f

  MethodHandle mh4 = mh2.asVarargsCollector(Object[].class);
  c = (char)mh4.invoke("foo", 0);
  System.out.println(c);  // f
}
```

Autre combinateurs

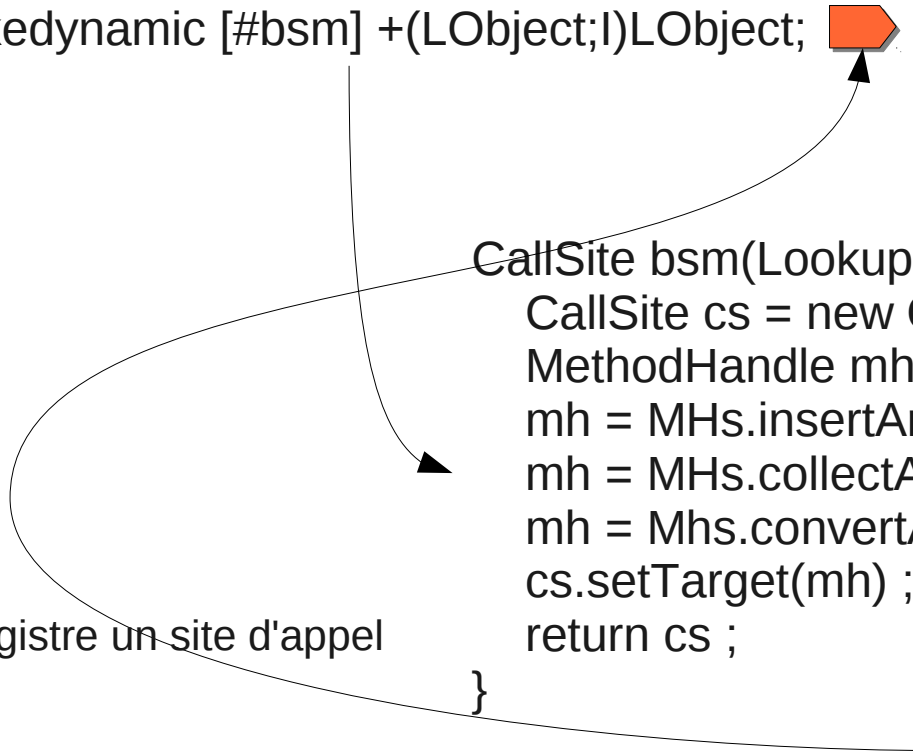
dans MethodHandles:

- guardWithTest
- dropArguments
- filterArguments/filterReturnValue
- foldArgument
- permuteArgument
- constant/identity
- ...

Implantons un inlining cache ?

La méthode de bootstrap installe une méthode générique de test

```
aload 1
iconst 1
invokedynamic [#bsm] +(LObject;I)LObject;
```



```
CallSite bsm(Lookup lookup, String name, MethodType type) {
    CallSite cs = new CallSite(type);
    MethodHandle mh = #fallback(CallSite, Object[]);
    mh = MHs.insertArguments(mh, 0, cs);
    mh = MHs.collectArguments(mh, type.pCount(), type.generic());
    mh = Mhs.convertArguments(mh, type);
    cs.setTarget(mh) ;
    return cs ;
}
```

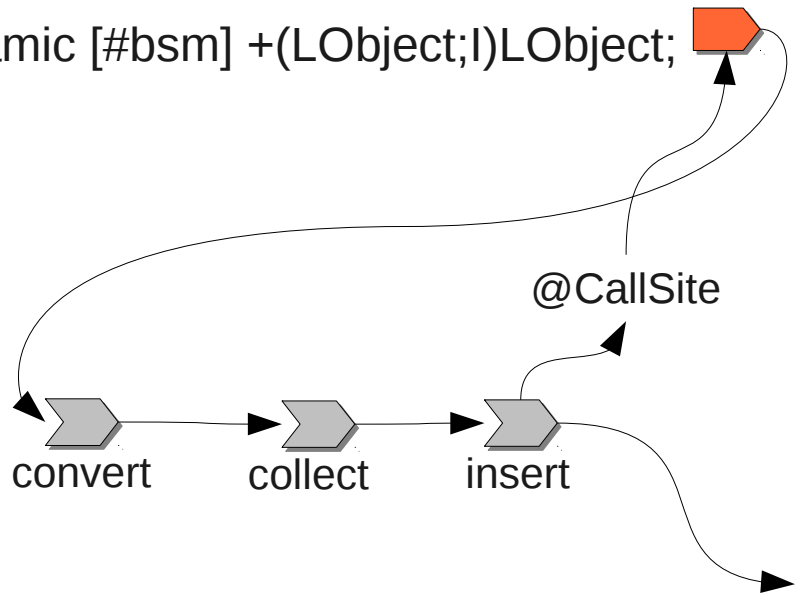
Enregistre un site d'appel


```
Object fallback(CallSite cs, Object[] args) { ... }
```

Implantons un inlining cache ?

La méthode générique installe un test/guard

```
aload 1
iconst 1
invokedynamic [#bsm] +(LObject;I)LObject;
```




```
Object fallback(CallSite cs, Object[] args) {
  MethodType type = type(cs, args);
  MH mh = findMH(type, cs.type());
  MH test = findTest(type, mh.type());
  MH mh = MHs.guardWithTest(test,
    mh,
    cs.getTarget());
  cs.setTarget(mh);
  return mh.invokeWithArguments(args);
}
```

Implantons un inlining cache ?

L'arbre est stable juqu'à ce qu'une nouvelle classe soit découverte

```
aload 1
iconst 1
invokedynamic [#bsm] +(LObject;I)LObject;
```