Machines Virtuelles
et bazard autour

Rémi Forax

Appel de fonction

- *caller*
  fonction contenant l'appel de fonction
- *callsite*
  site d'appel: call(declaredArgumentTypes)
- *callee*

  fonction appelée: def m(parameterTypes)

Le type des arguments et le type des paramètres n'est pas forcément le même
=> adaptation

Appel de méthode

Appel de méthode

polymorphisme/appel virtuelle

Appel de méthode est un appel de fonction avec une sélection dynamique de l'implantation en fonction des arguments

- single-dispatch

  La sélection ne dépend du type dynamique du *receiver*

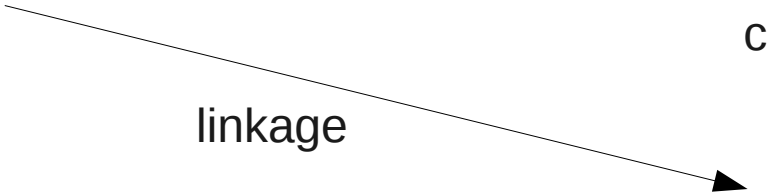
- multi-dispatch

  La sélection dépend du type dynamique de tous les arguments

Il n'est pas nécessaire d'avoir une classe pour avoir un appel de méthode

Edition de lien

```
main ([Ljava/lang/String;)V
  ...
  aload 2
  iconst 1
  invokevirtual Foo foo (I)D
```




```
class Foo {
  ...

  foo (I)D
    iload 1
    i2d
    dreturn

}
```

invokespecial constructeur, super et privé
invokestatic méthode statique
invokevirtual single dispatch
invokeinterface single dispatch
invokedynamic l'utilisateur spécifiec l'édition de lien

invoke*

Appel de fonction ou de méthode ?

- receiver
  - invokespecial, invokevirtual & invokeinterface
- receiverless
  - invokestatic, invokedynamic

invokespecial

la classe spécifiée est soit la classe du caller ou sa super classe

invokevirtual requiert une classe

invokeinterface requiert une interface

invokedynamic

il faut aussi spécifier une méthode de *bootstrap*

Single dispatch

La méthode a appelé en calculer à l'exécution en fonction de la classe du *receiver*

Idée: vtable

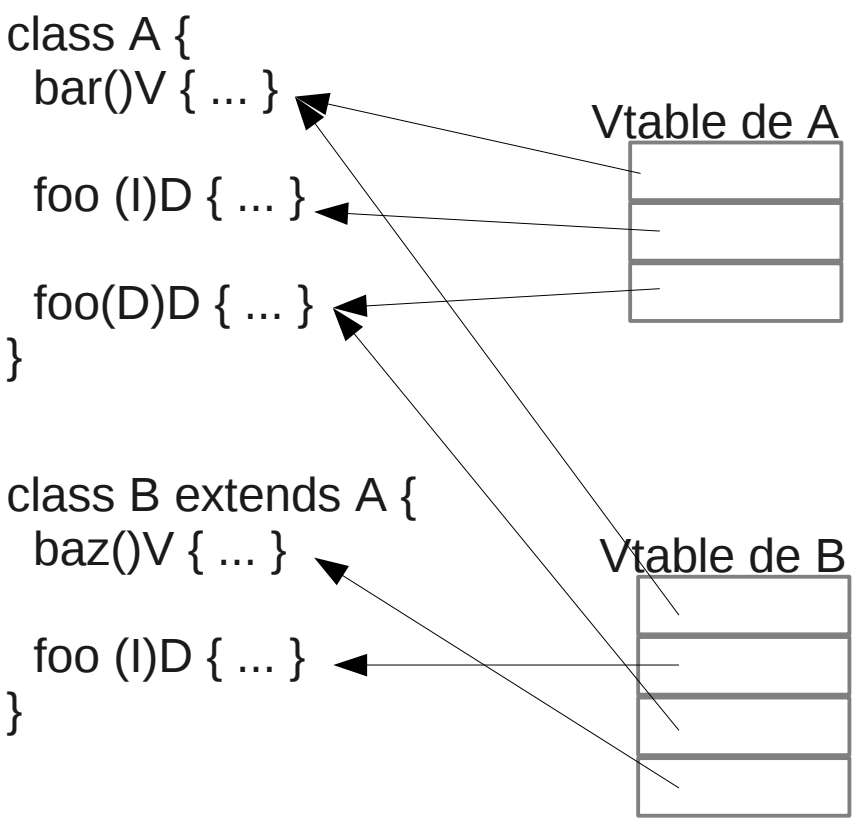
- Lors du chargement d'une classe
  - On recopie la vtable de la classe parent
  - On ajoute les méthodes propre à la classe en utilisant le même index de la vtable si la méthode redéfinie une déjà présente
- Lors de l'édition de lien, calculer l'index de l'appel

Création de la vtable

La vtable est l'état de la classe B si toutes les méthodes de A étaient recopiées dans B

Les méthodes privées, static etc. Ne sont pas stockées dans la vtable

Les méthodes rédéfinies partage le même index



Single dispatch

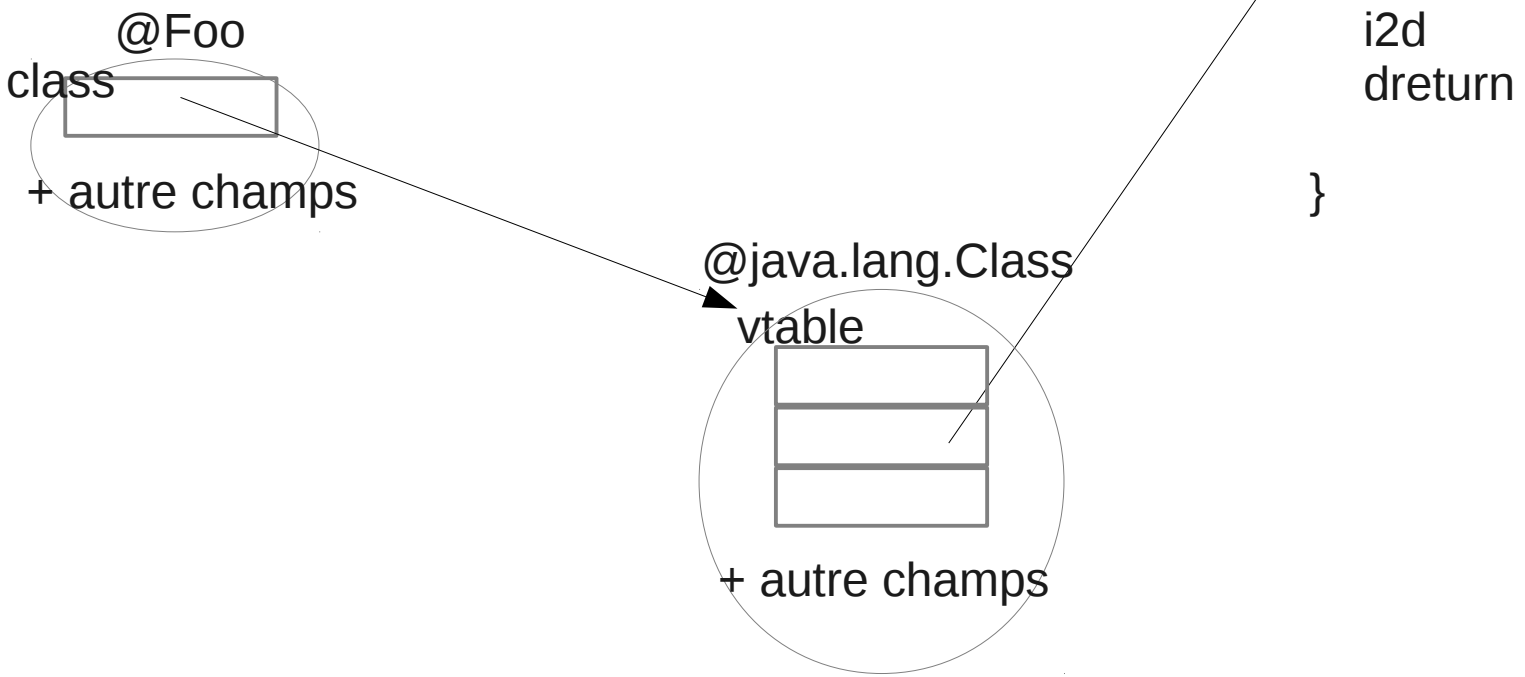
```
main ([Ljava/lang/String;)V
  ...
  aload 2
  iconst 1
  invokevirtual Foo foo (I)D
```

linkage

vtable_slot = 1

```
class Foo {
  ...

  foo (I)D
    iload 1
    i2d
    dreturn

}
```

a.class[vtable_slot] (a, 1)

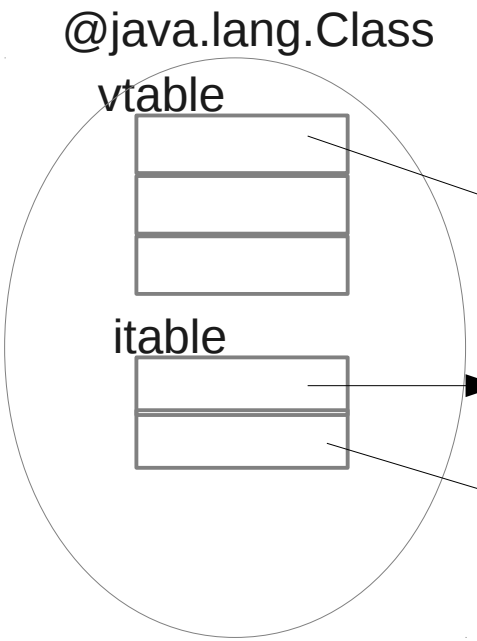


Interface

Il faut une itable par interface !

```
main ([Ljava/lang/String;)V
  ...
  aload 2
  iconst 1
  invokeinterface J (I)D
```

linkage → itable=1 slot = 1




```
interface I {
  void m();  // 0
}

interface J {
  void f();   // 0
  void m(); // 1
}

class Foo
  implements I, J {

  void m() { ... }
}
```

Morphism du callsite

- On sait que même si l'appel est virtuel bcp son monomorphique
- Monomorphique

  Une seul callee à l'exécution
- Polymorphique

  Plusieurs implantations mais comptable sur les doigts d'une main

    Bi-morphic (2 callees)
- Megamorphique

  Pleins d'implantations

  => Essayer de détecter les callsite monomorphique pour les inliner

Class Hierarchy Analysis

L'appel est forcément monomorphique:

- Si un type n'a qu'une implantation
- Si une méthode n'est jamais redéfinie

Si les classes sont chargées à l'exécution

Cela peut invalider une condition

=> il faudra dé-optimizer

Inlining cache

Idée: se souvenir de la classe du receiver, si la classe est identique alors la méthode est identique

Il faut pour cela être capable de modifier le code lors de l'exécution

- on réserve de la place pour le check + jump

=> le code peut alors être inliner !

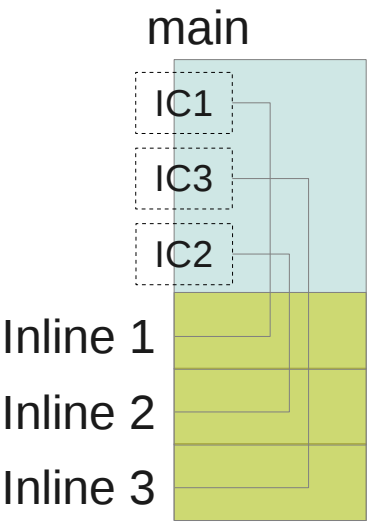
```
main ([Ljava/lang/String;)V
  ...
  aload 2
  iconst 1
  invokevirtual Foo foo (I)D
```

```
class Foo {
  void foo() { ... }
}

class Bar extends Foo {
  void foo() { ... }
}
```

```
      main ([Ljava/lang/String;)V
        ...
        // st r2, ??
        st r3, 1
        jpe r2  // nullcheck
        ld r4, [r2]  // .class

0x...A  nop
        nop

        ld r5, [r5 + 32]  // vtableSlot[foo]
        ...   // update callsite (0x...A) with [r4], [r5]
        call r5, r2, r3
        ...
```

```
main ([Ljava/lang/String;)V
  ...
  // st r2, ??
  st r3, 1
  jpe r2  // nullcheck
  ld r4, [r2]  // .class

0x...A  cmp r4, 0x...r4   // foo.class
  Jmp 0x...E

  ld r5, [r5 + 32]  // vtableSlot[foo]
  ...   // update callsite (0x...A) with [r4], [r5]
  call r5, r2, r3
0x...B  ...
  ret

0x...E  // inline the code of Foo::foo (specialized !) here
  ...
  goto 0x...B
```

```
class Foo {
  void foo() { ... }
}

class Bar extends Foo {
  void foo() { ... }
}
```



Inlining cache : conclusion

On sépare un slowpath et un fastpath

- on protège le fastpath avec un guard
- on espère que le gaurd sera toujours vrai

Marche mieux si l'architecture est tiered

évite de générer du code à la fin des méthodes !

Si l'appel est polymorphe, on peut utiliser plusieurs guards (pas trop !)

Ne marche pas si le code l'appel est megamorphic

JSR 292

invokedynamic them all

Idée !

Ajouter un nouveau bytecode (invokedynamic)

- 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

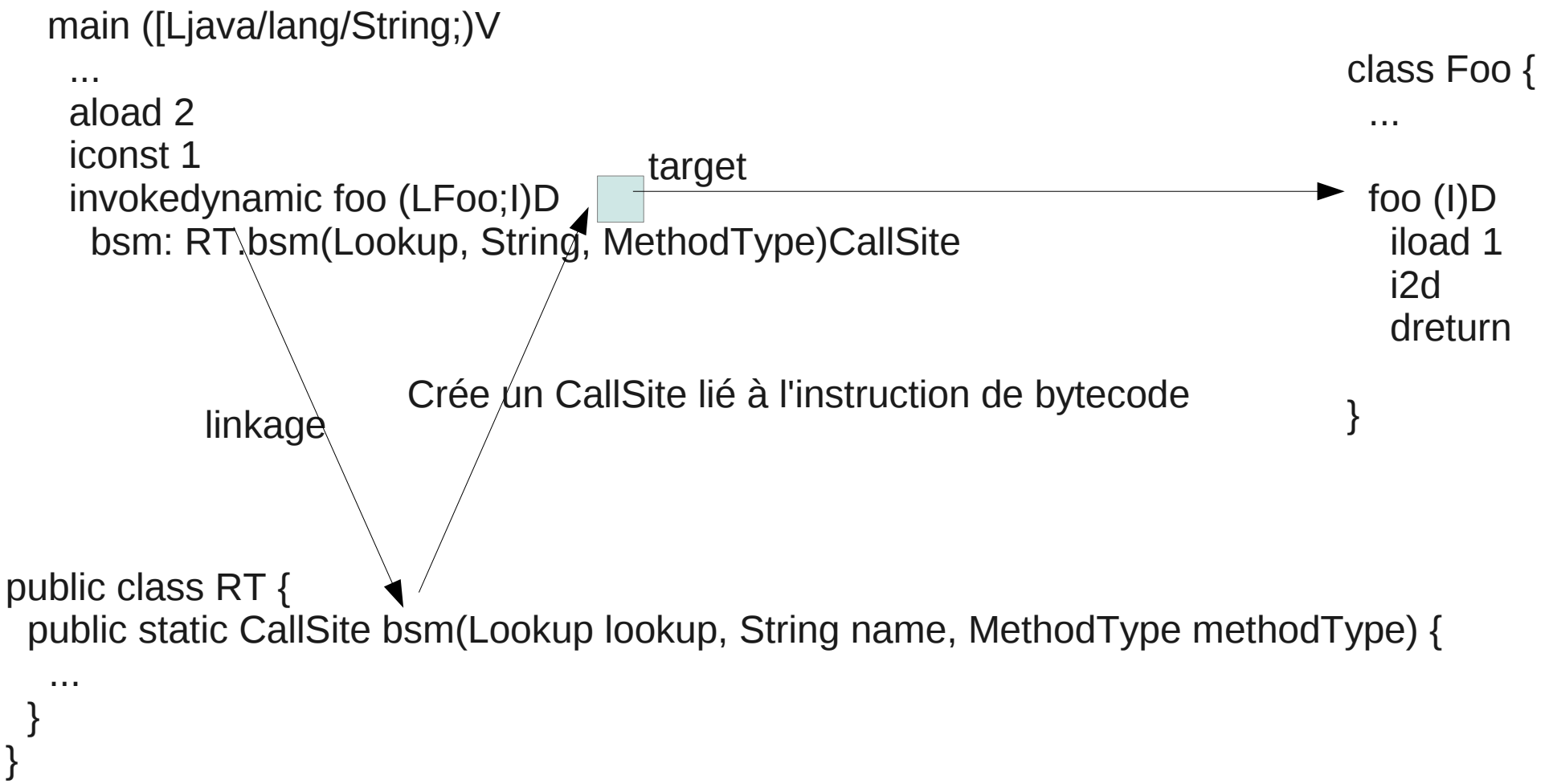
Comment ça marche ?

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

```
aload 2
iconst 1
invokedynamic foo (LFoo;I)D
  bsm: RT.bsm(Lookup, String, MethodType)CallSite
```

bootstrap

L'appel à la méthode de bootstrap se fait une seul fois lors du premier appel



Bootstrap (suite)

- La méthode de boostrap est appelée avec en paramètre
  - Un Lookup qui représente la classe du caller
  - Un nom (String) qui est le nom indiqué par l'opcode invokedynamic
  - Un MethodType qui représente la signature (type de retour + types des paramètres) de l'opcode invokedynamic
- Le pointeur de fonction devra pointer vers une fonction (le callee) ayant la même signature exactement !

Update à postériori

De plus, si on garde une référence sur l'objet CallSite, la méthode CallSite.setTarget permet de changer le pointeur du site d'appel à n'importe quel moment

Cela permet par exemple de changer le test comme dans un inlining cache

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 teste la sécurité à chaque appel, j.l.invoke.methodHandle teste 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 avec la visibilité public

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

Un exemple de findStatic avec la visibilité de la classe courante

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

Appel du pointeur de fonction

MethodHandle.invoke et MethodHandle.invokeExact sont des méthodes traitées spécialement par le compilateur

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)

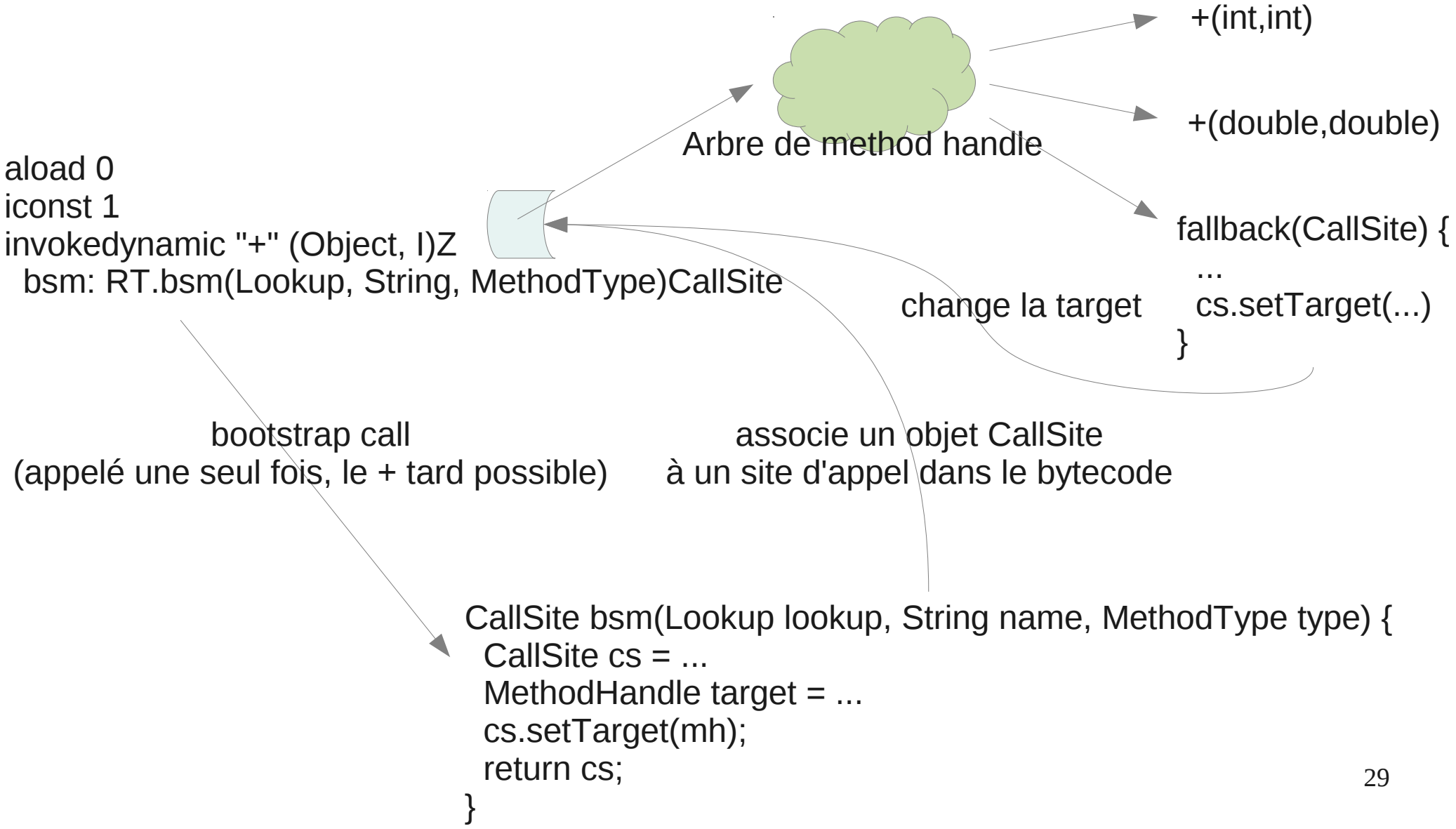
- essaye de faire les conversions sans perte et boxing/unboxing
  - si mh est marqué avec asVarargsCollector(), fait un appel varargs
- appel peut être lent, dépend des conversions

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

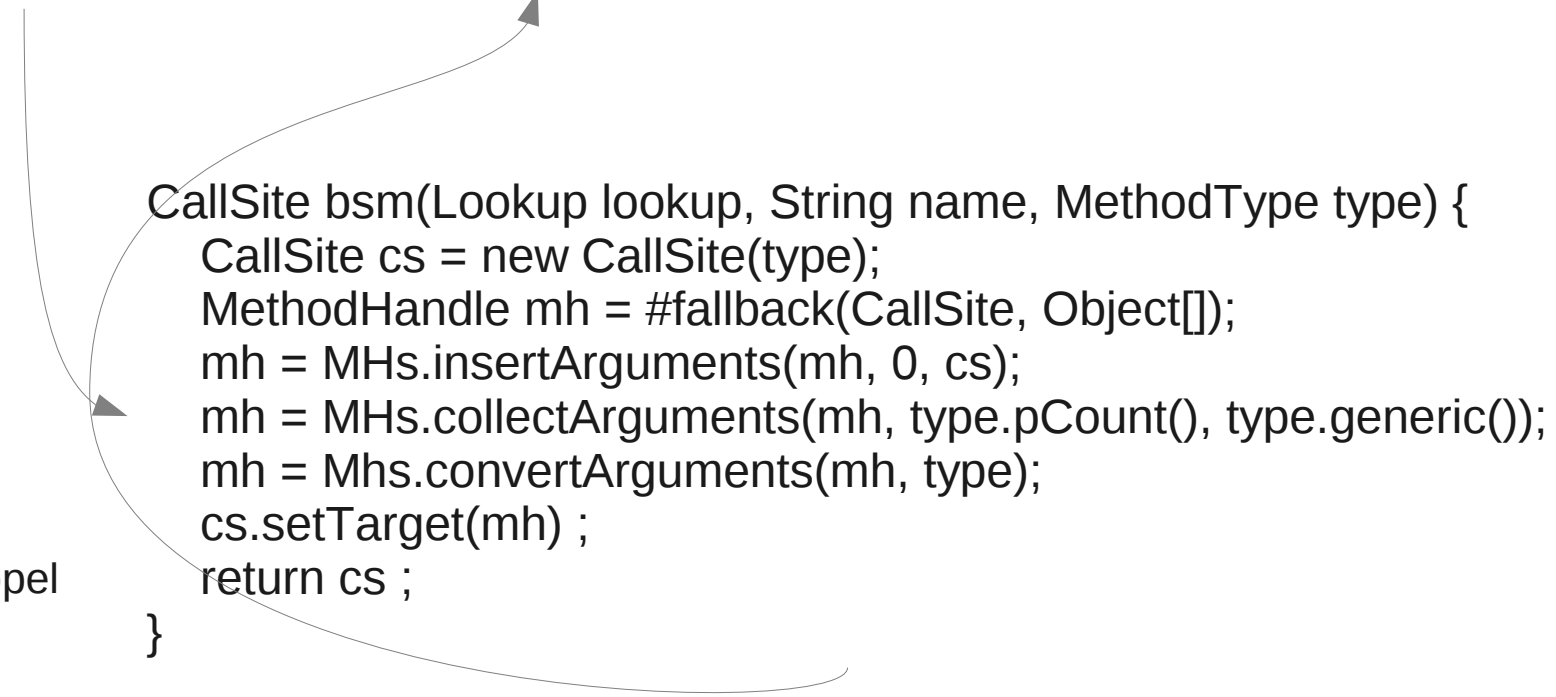
dans MethodHandles:

- 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 ;
}
```

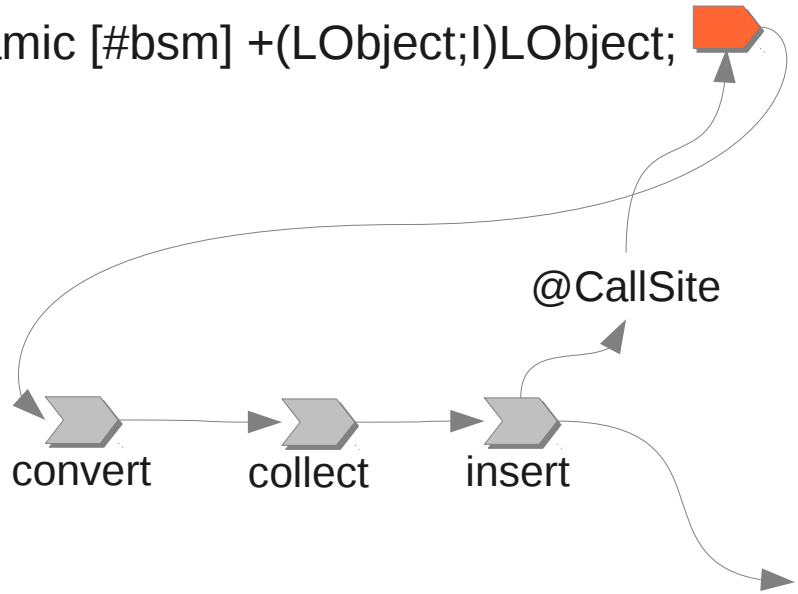
Enregistre un site d'appel


```
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;
```