invokedynamic under the hood

Nadeesh T V

ORACLE India Pvt Ltd

26 Aug 2016

Outline

JVM Languages

- Languages which can run on Java Virtual Machine (**JVM**)
- Should be a valid class file containing byte codes
- More than 200+ JVM languages [1]
  - Java (**primary language**) [2]
  - JRuby (ported version of Ruby)
  - Jython (ported version of Python)
  - JavaScript
  - Groovy
  - Smalltalk
  - Scala etc...

JVM Languages

- Semantics of other JVM languages differ from Java in many ways
  - Types can be **dynamic**

```
def sum(a, b):
total = a + b
return total;
```

  - Method invocation can be different
    - Smalltalk send messages for invocation
  - **Scope, access rules** can be different

Non Java Language's Overhead

Figure: NonJava languages on JVM

Wrapper layer adds **performance overheads** while running on JVM!!!

NonJava Language's Pain Points

- NonJava compiler implementors have to write lot of code
  - JVM have to **load** lot of byte code (performance overhead)
- Some of the dynamic languages heavily use **Reflection** (performance overhead) [3]
  - **Every** invocation requires an **access check**
  - Method arguments are objects - **boxing/unboxing**

NonJava Language's Pain Points

Wrapper layer act as a black box which can affect **JIT inlining!!!**

```
void method1(boolean arg) {
  if (arg) {
    do.....
  } else {
    do...
  }
}
method1(false)
```

Types of Method Invocations

Before Java 7

- **invokestatic** - Used to call static methods

```
static void staticMethod() {
}

A.staticMethod();
```

- Byte code of above invocation will be

```
invokestatic #7 //Method staticMethod:()V
```

Types of Method Invocations

Before Java 7

- **invokevirtual** - Used to call instance methods

```
void instanceMethod() {
}

    obj.instanceMethod();
```

- Byte code of above invocation will be

```
invokevirtual #4    // Method instanceMethod:()V
```

Types of Method Invocations

Before Java 7

- **invokeinterface** - Use interface reference to call methods

```
List l = new ArrayList();
l.add("1");
```

- Byte code of above invocation will be

```
invokeinterface #10, 1
//InterfaceMethod java/util/List.add:(Ljava/lang/Object)Z
```

Types of Method Invocations

Before Java 7

- **invokespecial** - Used to call private, constructors, super methods

```
private void privateMethod() {
}

    obj.privateMethod();
```

- Byte code of above invocation will be

```
invokespecial #5    // Method privateMethod:()V
```

Types of Method Invocations

Before Java 7

Instruction	Receiver	Dispatch
invokestatic	no	no
invokevirtual	yes	yes
invokeinterface	yes	yes
invokespecial	yes	no

Figure: Summary of JVM bytecode invocations [2]

- **Receiver** - Object in oop
- **Dispatch** - Method searching and linking

- Started in **2005**
- Feature Completed in **2011**
- Released in **JDK 7**

Key Features

- New byte code - **invokedynamic**
- New unit of behavior - **methodhandle**

invokedynamic

5 byte instruction

opcode	index1	index2	unused1 (0)	unused2 (0)

Key Features

- No **receiver**
- **Custom dispatching** under **user control**
- Dynamic call sites can be **relinked** dynamically

Linking of a Dynamic Callsite (DC) to a method (One time activity)

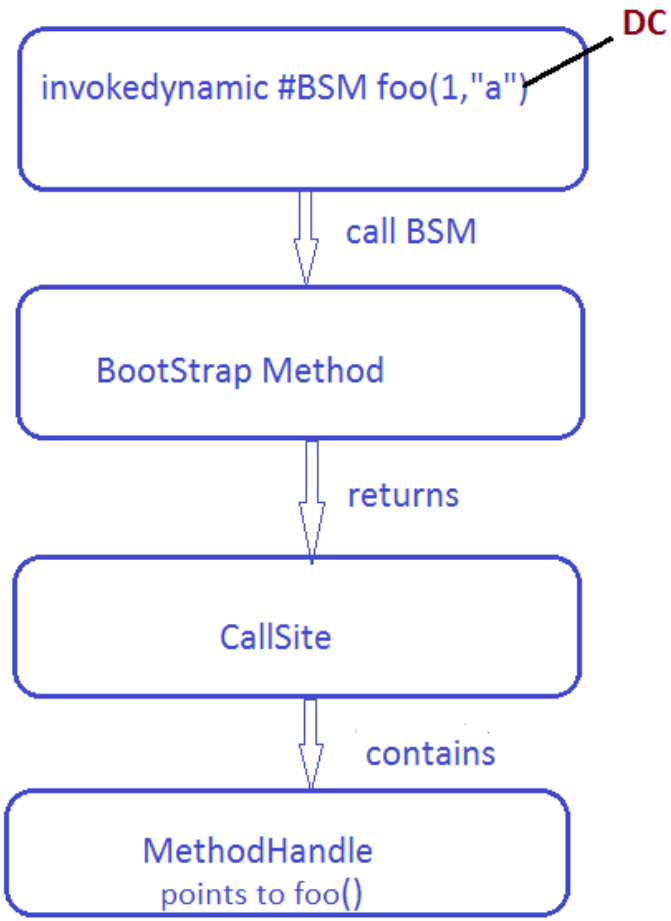


Figure: Flow of BootStrapping [4]

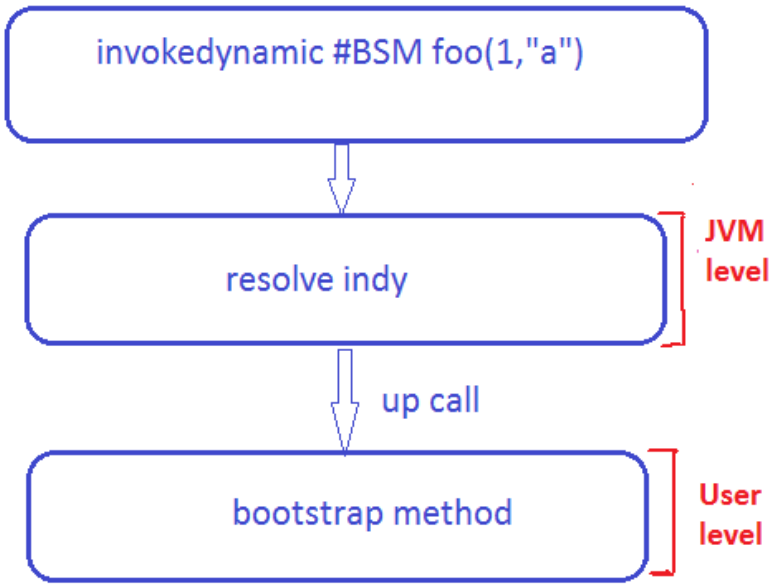
BootStrap Method

Accept 3 Compulsory arguments

(void, int, String)

(Lookup obj, String any, MethodType type,....)

Byte code generator's responsibility to link indy to BSM



Linking under user control !!!

BootStrapping

2nd time invocation

(No bootstrapping/linking)

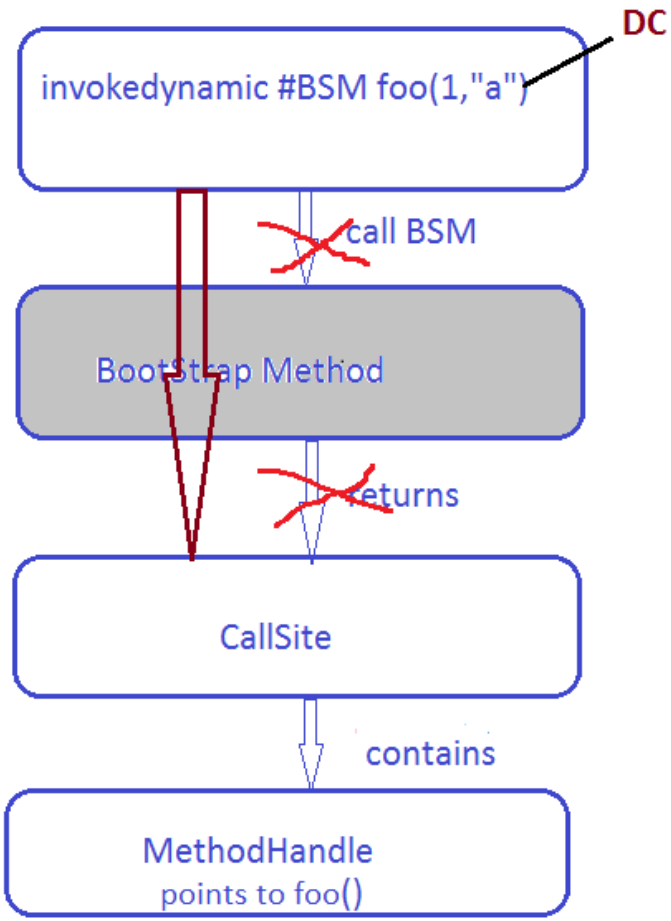


Figure: Flow of BootStrapping [4]

BootStrap Method - Example

```
Runnable r = () -> System.out.println("");
```

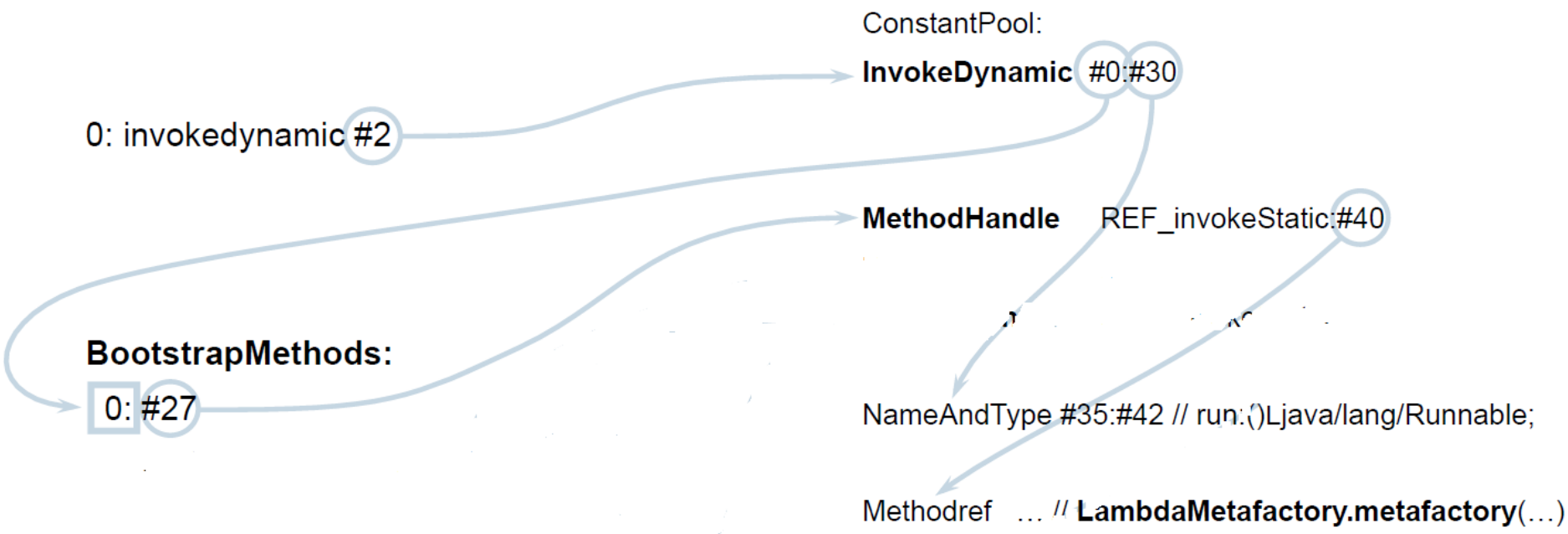
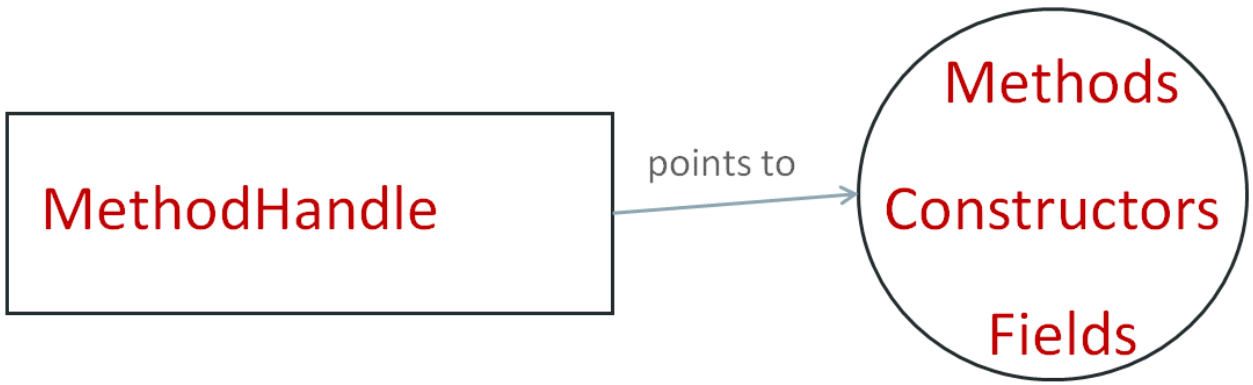


Figure: Lambda expression byte code details

Image adapted from [4]

MethodHandle (MH)

"MethodHandle is a typed, directly executable reference to an underlying method, constructor, field"



MH can be considered as a **function/field pointer**

Lookup

- Contain **access permission** of a class
- Should share only with **trusted parties**
- Used in **creating** MethodHandle
- Two types - **publicLookup and lookup**

In case of indy, JVM will pass lookup object to BSM

MethodHandle - Usage

- Create using
  - **MethodHandles.Lookup** (in Java)
  - **CONSTANT_MethodHandle** (in byte code)

```
MethodHandles.Lookup lookup = MethodHandles.publicLookup()

MethodHandle m = lookup.findStatic(String.class,"valueOf",
    MethodType.methodType(String.class, char.class));
String output = (String) m.invokeExact('c');
```

MethodHandle : Invocation

- **invokeExact**
  - Argument type should be **exact match**
  - If no match, throw **WrongMethodTypeException**
- **invoke (invokeGeneric)**
  - Argument should be **exact match/type converted** match (boxing/unboxing, casting)
  - If no match, throw **WrongMethodTypeException**

MethodHandle - Lookup Failing Case

```
class B {
    private void m() {}

}

public class A {
public static void main(String[] args) throws Throwable{
  MethodHandle m = MethodHandles.lookup().findSpecial(
    B.class,
    "m",
    MethodType.methodType(void.class), B.class);
        }
}
```

Throws IllegalAccessException

MethodHandle - Advantages

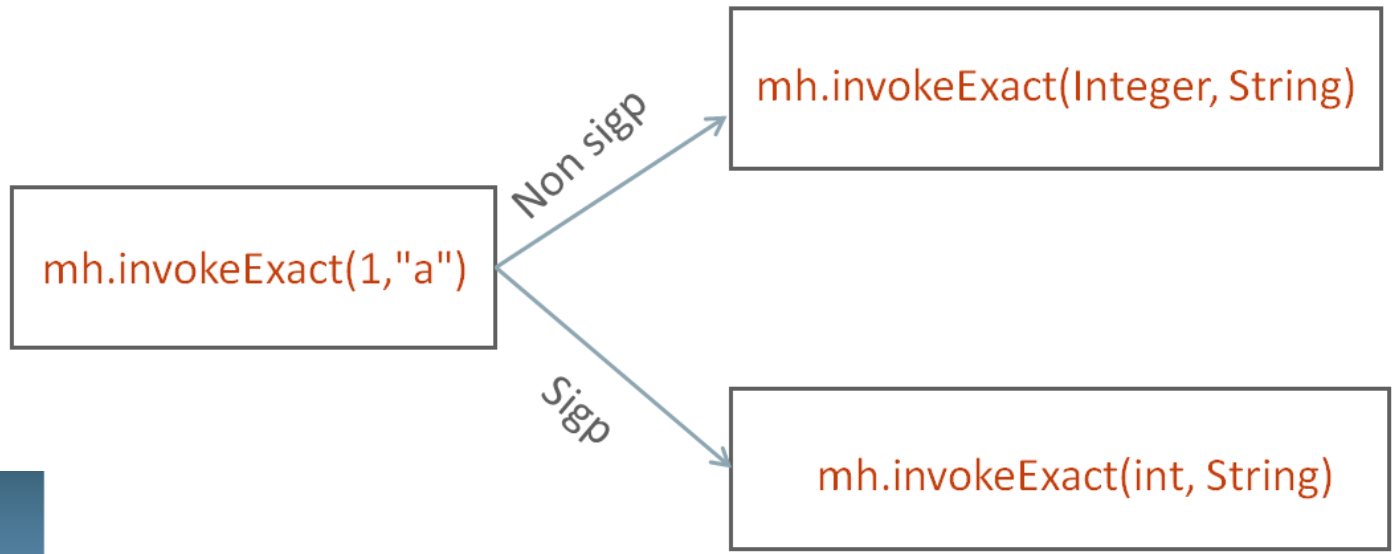
- **Access check only during creation of MH**
- **No more access check during usage**
- No more boxing/unboxing overhead due to signature polymorphic invocation

Signature Polymorphism

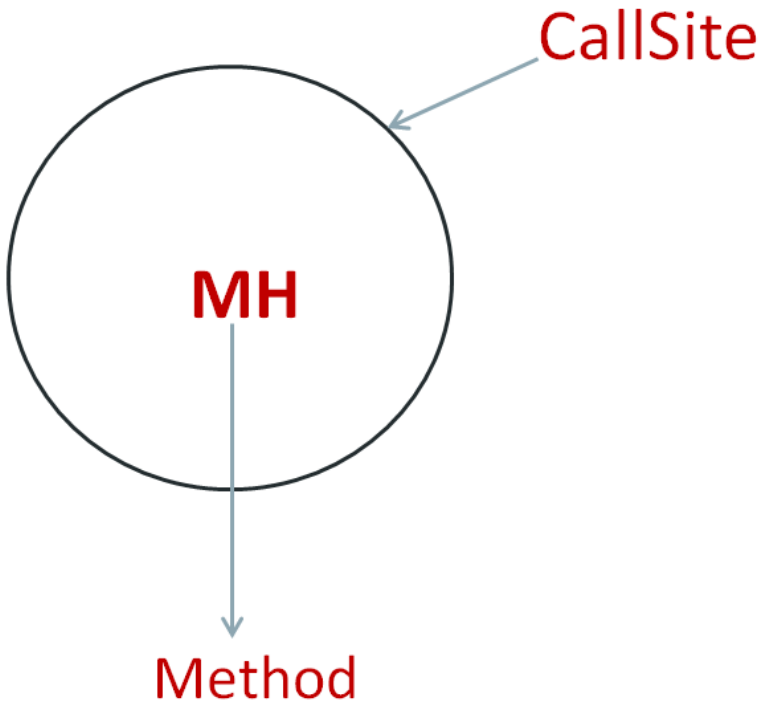
```
public final native @PolymorphicSignature
Object invokeExact(Object... args) throws Throwable;
```

"A signature polymorphic method is one which can operate with any of a wide range of call signatures and return types."

- invoke/invokeExact are signature polymorphic



CallSite (CS)



- MethodHandle(MH) **wrapped** into CallSite
- CS **relink/retarget** to different MH **dynamically** (setTarget)
- CS is **transparent** to JIT and can optimistically **inline** through it

Summary

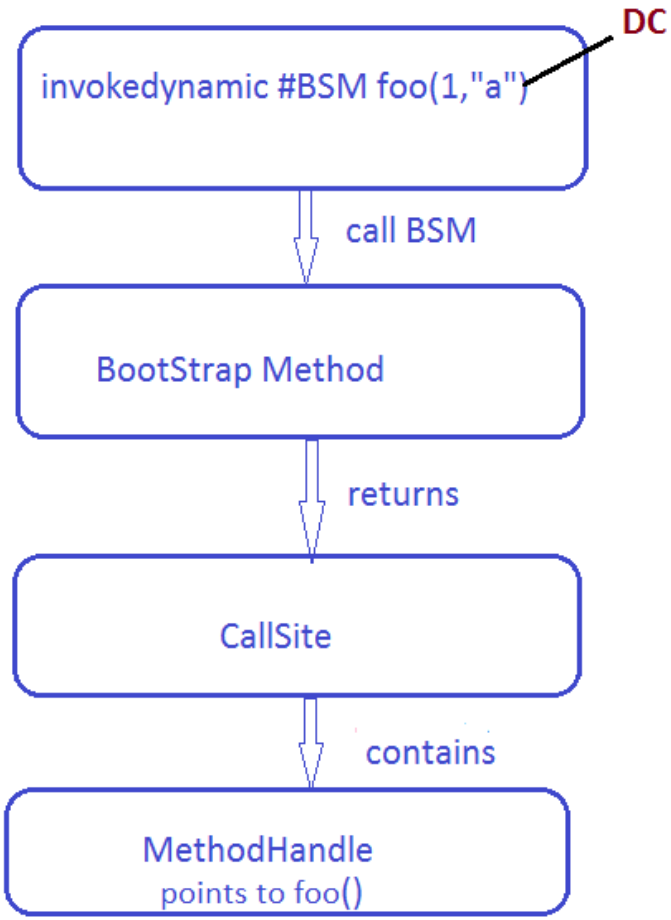


Figure: Flow of BootStrapping [4]

Acknowledgement

I would like to thank Vladimir Ivanov, Jamsheed CM (Oracle, Hotspot team) for helping me in understanding invokedynamic.

References

- 1. "A Renaissance VM: One Platform, Many Languages", JRose, JVMLS 2011
- 2. http://cr.openjdk.java.net/ jrose/pres/200910-VMIL.pdf
- 3."Invokedynamic for Mere Mortals", David Buck, JavaOne 2015
- 4."http://cr.openjdk.java.net/ vlivanov/talks/2015-IndyDeepDive.pdf", VIvanov
- 5."http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html", CNutter

Thank you

Types

- **Static** - types determined at compile time

```
int x ;
x = 2;
x = "hi" ; // Error
```

- **Dynamic** - types determined at run time

```
x = 2;
x = "hi" ; // Works
```

Types

- **Weak** - types can be mixed

```
$a=10;
$b="a";
$c=$a.$b;
print $c; #10a

$a=10;
$b="20a";
$c=$a+$b;
print $c; #30
```

Types

- **Strong** - No implicit type conversion

```
x = 2;
y =  2 + "hi" ; // Error
```

References
1. "http://www.rubyfleebie.com/ruby-is-dynamically-and-strongly-typed/"
2."http://www.i-programmer.info/programming/theory/1469-type-systems-demystified-part2-weak-vs-strong.html"