Proxy 2.0 - Behind The Scene

Rémi Forax

JFokus VM Summit - Feb 2015

Me

Assistant Prof at Paris East University

JCP Expert

- Invokedynamic (JSR 292)
- Lambda (JSR 335)
- Module (JSR 376)

Java Dev

- ASM
- OpenJDK
- ...

Big Disclaimer

Don't believe Me !

Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah.

Why ?

Proxies are everywhere

Most Java frameworks use proxies

- Spring, Hibernate, Weld, etc

JDK uses Proxies

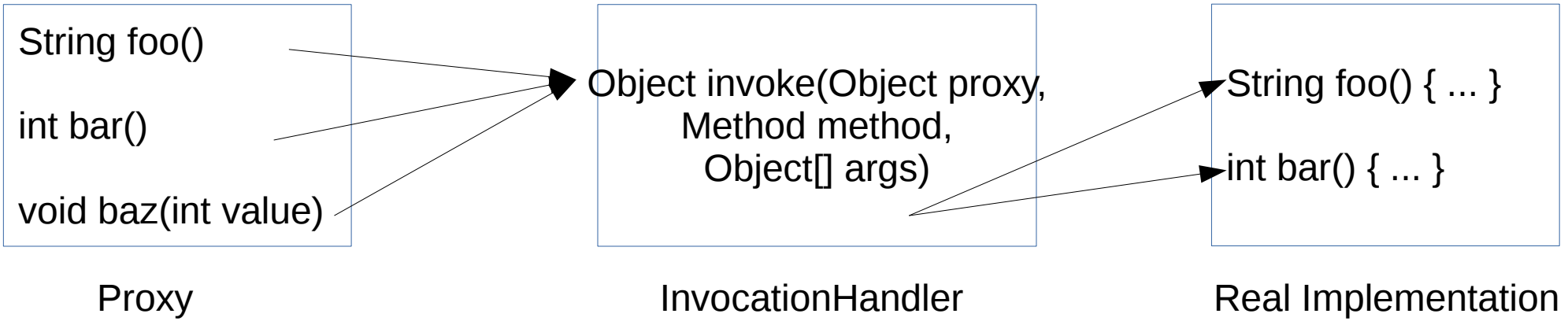
- MethodHandleProxies and Annotation support

Dynamically typed languages may use proxies

- to export script part as Java objects

java.lang.reflect.Proxy

Act as a kind of multiplexer



Use j.l.r.Method to do the de-multiplexing

j.l.r.Proxy shows its age :(

Most Frameworks do not use j.l.r.Proxy anymore
but bytecode generation

- ASM, Javassist, BCEL, etc

j.l.r.Proxy is slow

- One method called at runtime (Boxing + Profile pollution)
- Two objects, proxy + handler (Allocation + Field deref)

j.l.r.Proxy is outdated

- No support of default methods
- Funky support of j.l.Object public methods

Proxy 2.0
linking / runtime call separation

Proxy 2.0

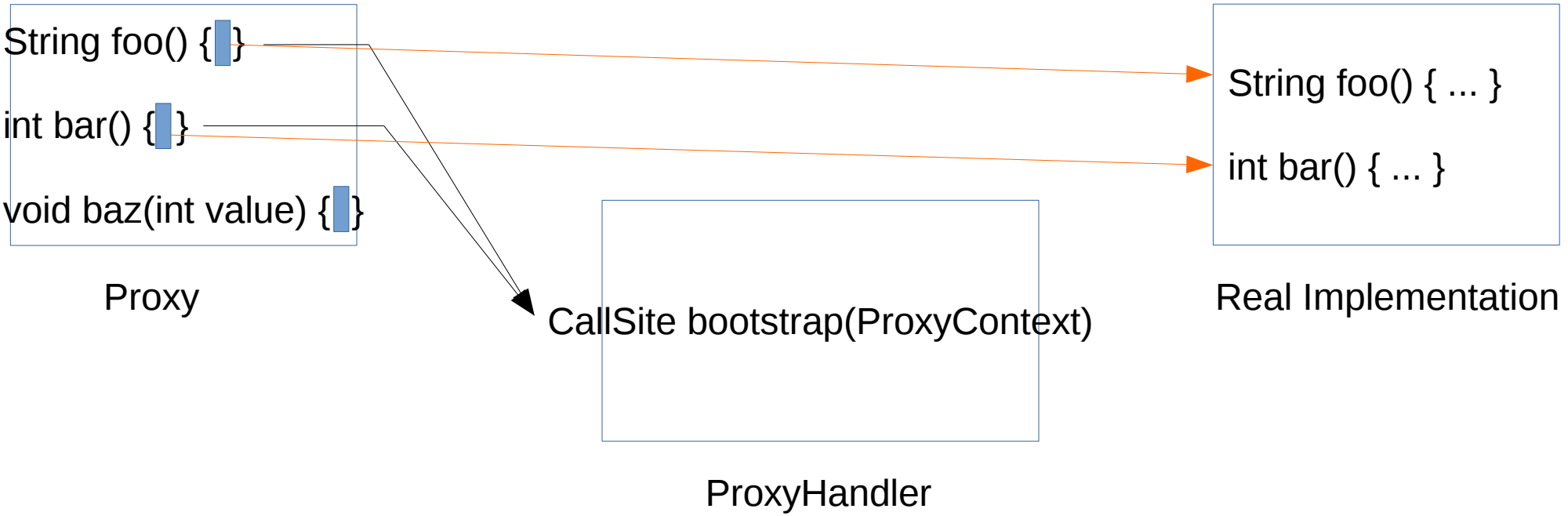
linking / runtime call separation

ProxyHandler: Linking a la invokedynamic

Use MethodHandle for invocation

Proxy2

As invokedynamic but at callee side



Use MethodHandle to link to the target method

Live Coding !

Hello Proxy

```
public interface Hello {
  public String message(String message, String user);

  public static void main(String[] args) {
    ProxyFactory<Hello> factory = Proxy2.createAnonymousProxyFactory(Hello.class,
      new ProxyHandler.Default() {
        public CallSite bootstrap(ProxyContext context) throws Throwable {
          System.out.println("bootstrap method " + context.method());
          System.out.println("bootstrap type " + context.type());
          MethodHandle target =
              methodBuilder(context.type())
                .dropFirstParameter()
                .thenCall(publicLookup(), String.class.getMethod("concat", String.class));
          return new ConstantCallSite(target);
        }
      });
    Hello simple = factory.create();
    System.out.println(simple.message("hello ", "proxy"));
    System.out.println(simple.message("hello ", "proxy 2"));
  }
}
```

Delegation

```
public interface Delegation {
  public void println(String message);

  public static void main(String[] args) {
    ProxyFactory<Delegation> factory =
      Proxy2.createAnonymousProxyFactory(Delegation.class,
        new Class<?>[] { PrintStream.class },
        new ProxyHandler.Default() {
          public CallSite bootstrap(ProxyContext context) throws Throwable {
            MethodHandle target =
              com.headius.invokebinder.Binder
                .from(context.type())
                .dropFirst()
                .invokeVirtual(publicLookup(), "println");
            return new ConstantCallSite(target);
          }
        });

    Delegation hello = factory.create(System.out);
    hello.println("hello proxy2");
  }
}
```

Intercept

```
public interface Intercept {
  public static void intercept(int v1, int v2) {
    System.out.println("intercepted " + v1 + " " + v2);
  }

  public static void main(String[] args) {
    ProxyFactory<IntBinaryOperator> factory =
      Proxy2.createAnonymousProxyFactory(IntBinaryOperator.class,
        new Class<?>[] { IntBinaryOperator.class },
        new ProxyHandler.Default() {
          public CallSite bootstrap(ProxyContext context) throws Throwable {
            MethodHandle target =
              methodBuilder(context.type())
                .dropFirstParameter()
                .before(b -> b
                    .dropFirstParameter()
                    .thenCall(publicLookup(), Intercept.class.getMethod("intercept", int.class, int.class)))
                .thenCall(publicLookup(), context.method());
            return new ConstantCallSite(target);
          }
        });

    IntBinaryOperator op = (a, b) -> a + b;
    IntBinaryOperator op2 = factory.create(op);
    System.out.println(op2.applyAsInt(1, 2));
  }
}
```

Intercept + exception

```
public interface Intercept {
  public static void intercept(int v1, int v2) {
    throw null;
  }

  public static void main(String[] args) {
      Proxy2.createAnonymousProxyFactory(IntBinaryOperator.class,
        new Class<?>[] { IntBinaryOperator.class },
        new ProxyHandler.Default() {
          public CallSite bootstrap(ProxyContext context) throws Throwable {
            MethodHandle target =
              methodBuilder(context.type())
                .dropFirstParameter()
                .before(b -> b
                    .dropFirstParameter()
                    .thenCall(publicLookup(), Intercept.class.getMethod("intercept", int.class, int.class)))
                .thenCall(publicLookup(), context.method());
            return new ConstantCallSite(target);
          }
        });

    ...
  }
}
```

```
Exception in thread "main" java.lang.NullPointerException
at Intercept.intercept(Intercept.java:17)
at Intercept.main(Intercept.java:39)
```

No proxy !

Proxy 2.0

ProxyHandler act as a linker

- **override**(Method)
  - Called if an implementation already exists (method of j.l.Object + default method)
- **bootstrap**(ProxyContext)
  - ProxyContext = Method to implement + MethodType of the callsite

- Fields are stored inside the proxy
  - ProxyFactory<T> createAnonymousProxyFactory(Class<T> type, Class<?>[] fieldTypes, ProxyHandler handler)

  and inserted before calling the method handle

Class is generated using ASM

Unsafe.defineAnonymousClass

To support private interface

Host class = interface

Security by taking a Lookup object as first parameter

... createAnonymousProxyFactory(Lookup lookup, ...

check that the interface is visible from the lookup.

Look Ma, no classloader !

Use constant pool patching to inject j.l.r.Method

Constant Pool patching

defineAnonymousClass(Class<?> hostClass,
byte[] bytecode, Object[] **patches**)

- array with the same size as the constant pool array
- Replace a String constant by a live Object

ldc or invokedynamic (bootstrap constants) to get access to the live objects

Very useful, not unsafe,
why not adding a new overload of ClassLoader.defineClass() that allow patching ??

Hidden Proxy

Generate proxy in package java.lang.invoke

Use annotations

- Hidden
  - mv.visitAnnotation("Ljava/lang/invoke/LambdaForm$Hidden;", true);
- ForceInline
  - mv.visitAnnotation("Ljava/lang/invoke/ForceInline;", true);

Problem => ProxyHandler must be visible from bootstrap classloader (from java.lang.invoke)

- The infamous NoClassDefFoundError !

Hidden Proxy

Generate proxy in package java.lang.invoke

Use annotations

- Hidden
  - mv.visitAnnotation("Ljava/lang/invoke/LambdaForm$Hidden;", true);
- ForceInline
  - mv.visitAnnotation("Ljava/lang/invoke/ForceInline;", true);

Problem => ProxyHandler must be visible from bootstrap classloader (from java.lang.invoke)

Inject ProxyHandler::bootstrap as a 'constant' in the constant pool

MethodBuilder

My own InvokeBinder

- Can do before/after using lambdas
  - => AOP/interceptors
- implementation uses lambdas too

- LambdaForm and findVirtual ?
  - No inlining cache ?? (not yet ???)
  - => Roll my own inlining cache
  - but the fallback is visible from the user POV :(

Inlining cache with asCollector

```
class InliningCacheCallSite extends MutableCallSite {
  ...
  private final MethodHandle mh;
  private int kindOfTypeCounter;   // the access is racy but we don't care !

  InliningCacheCallSite(MethodHandle mh) {
      super(mh.type());
      this.mh = mh;
      setTarget(FALLBACK.bindTo(this)
          .asCollector(Object[].class, mh.type().parameterCount())
          .asType(mh.type()));
  }

  private static MethodHandle fallback(InliningCacheCallSite callsite, Object[] args)
                                                                    throws Throwable {
    if (callsite.kindOfTypeCounter++ == MAX_KIND_OF_TYPE) {
      callsite.setTarget(mh);
    } else {
      MethodHandle test = insertArguments(CLASS_CHECK, 1, receiver.getClass());
      callsite.setTarget(guardWithTest(test, mh, callsite.getTarget()));
    }
    return mh.invokeWithArguments(args);
  }
}
```

Oops !

MethodBuilder

My own InvokeBinder

- Can do before/after using lambdas
  - => AOP/interceptors
- implementation use lambdas too

- LambdaForm and findVirtual ?
  - No inlining cache ?? (not yet ???)
  - => Roll my own inlining cache
  - but the fallback is visible from the user POV :(
  - => Use **foldArguments** + **exactInvoker** instead !

foldArguments + exactInvoker

```
class InliningCacheCallSite extends MutableCallSite {
  ...
  private final MethodHandle mh;
  private int kindOfTypeCounter;   // the access is racy but we don't care !

  InliningCacheCallSite(MethodHandle mh) {
      super(mh.type());
      this.mh = mh;
      MethodType type = mh.type();
      setTarget(foldArguments(exactInvoker(type), FALLBACK.bindTo(this)
          .asType(methodType(MethodHandle.class, type.parameterType(0)))));
  }

  private static MethodHandle fallback(InliningCacheCallSite callsite, Object receiver)
                                                                    throws Throwable {
     if (callsite.kindOfTypeCounter++ == MAX_KIND_OF_TYPE) {
       callsite.setTarget(mh);
       return mh;
     }
     MethodHandle test = insertArguments(CLASS_CHECK, 1, receiver.getClass());
     callsite.setTarget(guardWithTest(test, mh, callsite.getTarget()));
     return mh;
  }
}
```

FoldArguments + exactInvoker

Avoid to call invokeWithArguments so the fallback method is not on the stack anymore when the target method is called

- A kind of tailcall !

It also solves the security issue raised by Jochen

- The fallback code is not part of the stack

Retrofit to Java 7

Used with current JavaEE implementations ?

Unsafe.defineAnonymousClass was introduced in 1.7

But MethodBuilder is full of lambdas :(

Retrofit to Java 7

Used with current JavaEE implementations ?

Unsafe.defineAnonymousClass was introduced in 1.7

But MethodBuilder is full of lambdas :(

- only need to emulate lambdas !
  - and Lambda objects are just proxies !

Lambda MetaFactory using Proxy 2.0

```
public static CallSite metafactory(Lookup lookup, String name, MethodType type,
                                   MethodType sig, MethodHandle impl, MethodType reifiedSig) throws ... {
    Class<?>[] capturedTypes = type.parameterArray();
    MethodHandle endPoint = impl.asType(reifiedSig.insertParameterTypes(0, capturedTypes));
    MethodHandle mh = Proxy2.createAnonymousProxyFactory(lookup, type,
        new ProxyHandler() {
            ...
            public boolean override(Method method) {
                return Modifier.isAbstract(method.getModifiers());
            }
            public CallSite bootstrap(ProxyContext context) throws Throwable {
                MethodHandle target = MethodHandles.dropArguments(endPoint, 0, Object.class);
                return new ConstantCallSite(target.asType(context.type()));
            }
        });
    if (type.parameterCount() == 0) {
        return new ConstantCallSite(MethodHandles.constant(type.returnType(), mh.invoke()));
    }
    return new ConstantCallSite(mh);
}
```

Java/Javascript bridge

```
var NIL = {
    size: function() { return 0 },
    forEach: function(consumer) { }
}

function nil() {
  return NIL
}

function cons(value, next) {
    return {
        value: value,
        next: next,
        size: function() { return 1 + next.size() },
        forEach: function(consumer) {
            consumer.accept(value)
            next.forEach(consumer)
        }
    }
}
```

```
interface FunList extends Bridge {
    int size();
    void forEach(IntConsumer consumer);
}

interface FunListFactory extends Bridge {
    FunList cons(int value, FunList next);
    FunList nil();
}

interface Bridge {
    public ScriptObjectMirror __getSOM__();
}
```

Java/Javascript bridge

```
interface FunList extends Bridge {
    int size();
    void forEach(IntConsumer consumer);
}
interface FunListFactory extends Bridge {
    FunList cons(int value, FunList next);
    FunList nil();
}

public static void main(String[] args) throws ... {
    ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
    try(Reader reader = Files.newBufferedReader(Paths.get("demo8/funlist.js"))) {
        engine.eval(reader);
    }
    ScriptObjectMirror global = (ScriptObjectMirror)engine.eval("this");
    FunListFactory f = bridge(FunListFactory.class, global);
    FunList list = f.cons(1, f.cons(2, f.cons(3, f.nil())));
    System.out.println(list.size());
    list.forEach(System.out::println);
}
```

Live Coding !

With j.l.r.Proxy

```
static Object unwrap(Object o) {
    if (o instanceof Bridge)
        return ((Bridge)o).__getSOM__();
    return o; }
```

```
static Object wrap(Class<?> returnType, Object o) {
    if (o instanceof ScriptObjectMirror)
        return createBridge(returnType, (ScriptObjectMirror)o);
    return o; }
```

```
private static Object createBridge(Class<?> type, ScriptObjectMirror mirror) {
    return type.cast(Proxy.newProxyInstance(type.getClassLoader(), new Class<?>[] { type },
            (Object proxy, Method method, Object[] args) -> {
                String name = method.getName();
                if (name.equals("__getSOM__")) {
                    return mirror;
                }
                if (name.startsWith("get")) {
                    String property = propertyName(name);
                    return wrap(method.getReturnType(), mirror.getMember(property));
                }
                if (name.startsWith("set")) {
                    String property = propertyName(name);
                    mirror.setMember(property, unwrap(args[0]));
                    return null;
                }
                if (args != null) Arrays.setAll(args, i -> unwrap(args[i]));
                return wrap(method.getReturnType(), mirror.callMember(name, args));
            }));
}
```

With Proxy2

```
public CallSite bootstrap(ProxyContext context) throws Throwable {
    Method method = context.method();
    String name = method.getName();
    MethodHandle target;
    if (name.equals("__getSOM__")) {
         target = methodBuilder(context.type())
                .dropFirstParameter()
                .thenCallIdentity();
    } else {
        if (name.startsWith("get")) {
            String property = propertyName(name);
            target = methodBuilder(context.type())
                .dropFirstParameter()
                .convertReturnTypeTo(Object.class)
                .insertValueAt(1, String.class, property)
                .compose(Object.class, b -> b.thenCallMethodHandle(GET_MEMBER))
                .thenCallMethodHandle(WRAP.bindTo(method.getReturnType()));
        } else {
            ...
        }
    }
```

With Proxy2

```
public CallSite bootstrap(ProxyContext context) throws Throwable {
    if (name.equals("__getSOM__")) {
        ...
    } else {
        if (name.startsWith("set")) {
            String property = propertyName(name);
            target = methodBuilder(context.type())
                .dropFirstParameter()
                .convertTo(void.class, ScriptObjectMirror.class, Object.class)
                .insertValueAt(1, String.class, property)
                .filter(2, Object.class, b -> b.thenCallMethodHandle(UNWRAP))
                .thenCallMethodHandle(SET_MEMBER);
        } else {
            int argumentCount = method.getParameterCount();
            target = methodBuilder(context.type())
                .dropFirstParameter()
                .convertReturnTypeTo(Object.class)
                .insertValueAt(1, String.class, name)
                .filterLastArguments(argumentCount, Object.class, Object.class,
                    b -> b.thenCallMethodHandle(UNWRAP))
                .boxLastArguments(argumentCount)
                .compose(Object.class, b -> b.thenCallMethodHandle(CALL_MEMBER))
                .thenCallMethodHandle(WRAP.bindTo(method.getReturnType()));
        } } }
```

Wrap up

Proxy 2.0

https://github.com/forax/proxy2

Modernized version of j.l.r.Proxy

ProxyHandler acts as a linker

Support methods of j.l.Object & default methods

Transparent and fast

Time to start a JEP ?