JSR 292 backport

(Rémi Forax)

University Paris-East

Interactive talk

- Ask your question when you want

- Just remember :
- Use only verbs understandable by a 4 year old kid
- Use any technical words you want

A stupid Idea ?

- JSR 292 exists because dynamic language needs a VM support
- Backport == dynamic language primitive without VM support
- Try and see !

Dev : backport state

- A first prototype
  - Just after the spec released (19 may 2008)
    - IKVM POC was the first
  - Before John's code
- Error in how Invokedynamic works
- Too much byte-codes at call site
- Available here :

  https://jvm-language-runtime.googlecode.com/svn/trunk/invokedynamic

What's the plan

- A new prototype from stratch
  - Using John's code (API part: **java.dyn**)
  - Better, better, better
  - In mlvm repository,
    keep in sync with other devs

Anatomy

- Emulation of 3 parts (3 patchs)
  - Anonymous klass (anonk)
  - Method Handles (meth)
  - Invokedynamic (indy)

Anonymous Klass

- Allow to create a class
  - without a classloader
  - without a name (use a generated name)
- Allow to patch an existing class by changing constant pool items

Anonymous Klass

- Allow to create a class
  - without a classloader
  - without a name (use a generated name)
  - Act as an inner-class of a host class
- Allow to patch an existing class by changing constant pool items

Anonymous Klass

- Anonymous name
  - Use unlikely name
- Access to host class members
  - Rewrite field access using
    - Unsafe.get*/put* or reflection
- Constant pool patch
  - Decode & replace constant pool value
    (backport already half implemented in John's code)

Anonymous Klass API

- 2 main scenarii:
  - Load an anonymous class

```
AnonymousClassLoader loader =
  new AnonymousClassLoader(Test.class);
byte[] array = AnonymousClassLoader.readClassFile(A.class);
Class<?> anonymousClass = loader.loadClass(array);
```

  - Parse a class, create a patch, load a patched class

```
AnonymousClassLoader loader =
      new AnonymousClassLoader(Test.class);
ConstantPoolParser parser = new ConstantPoolParser(A.class);
final ConstantPoolPatch patch = parser.createPatch();
parser.parse(...);
Class<?> anonymousClass = loader.loadClass(patch);
```

Class parsing/patching

- ConstantPoolVisitor
  - visit*(int index, byte tag,...)
- ConstantPoolPatch
  - put*(int index, byte tag, etc.)

```
AnonymousClassLoader loader =
    new AnonymousClassLoader(Test.class);
ConstantPoolParser parser = new ConstantPoolParser(A.class);
final ConstantPoolPatch patch = parser.createPatch();
parser.parse(new ConstantPoolVisitor(){
    public void visitConstantString(int idx,byte tag,String name,int nameIdx) {
        if ("hello".equals(name)) {
            patch.putConstantValue(index,tag,"hello patch");
        }
});
Class<?> anonymousClass = loader.loadClass(patch);
```

Backport Anonymous Klass

- 2 different issues :
  - Which classloader should be used ?
    - The major pain point
  - How to inject the (*patched*) bytecode ?
    - Security problem

Anonymous Klass Loading

- Lot of dynamic languages control already use classloader, so ask the Language runtime

- Use the hostClass classloader
  - Call defineClass (protected) directly by reflection

Backport invokedynamic

- Intercept class loading to modify bytecode
- Two ways to know if a class is «dynamic» or not
  - Class is registered at runtime using Linkage.registerBootstrapMethod
  - BootstrapInvokeDynamic class attribute
    - Need only to read class header

Trap classloading

- Use your own classloader
  - Ask to the language runtime
  - Generate lot of corner cases
    - security, serialization, etc.
- Use java.lang.instrumention (1.5) and create your own agent
  - Not used in many language runtime, why ?
- Enhance during class generation

At compile time

- Enhancement during class generation
- More time for a more precise analysis
  - Allow to use static analysis
- No classloding problem
  - Not true, current spec requires dynamic code creation (cf method handle adapter)

For language implementor

- 2 different backends
  - JSR 292 backend
  - JSR 292 backport backend
- So backport can't be easily enabled/disabled at runtime depending on the VM version
  - Not very cool

Bytecode enhancer

- Use ASM, fast
  (thanks again eugene)
- Can perform static analysis
  - not in one pass
  - not that fast
- remove StackMap at runtime
  - Fast but unsecure

Bytecode stub

- Replace all invokeinterface on java.dyn.Dynamic by this code

```
class MyCaller {
    private static final CallSite site1 = (...);
    private static final MethodHandle bootstrapMethod = (...);
    Object myMethod(Object x, Object y, int z) {
        // x . invokedynamic["alpha",(Object,int)Object] (y, z)
        MethodHandle method1 = site1.target;
        if (method1 == null)
            return bootstrapMethod.invoke(x, site1, {y, z});
        return method1.invoke(y, z);
    }
```

Slow path/Fast path

- Branches of the if should be optimized differently

```
Object myMethod(Object x, Object y, int z) {
        // x . invokedynamic["alpha",(Object,int)Object] (y, z)
        MethodHandle method1 = site1.target;
        if (method1 == null) // slow path
            return bootstrapMethod.invoke(x, site1, {y, z});
        // fast path
        return method1.invoke(y, z);
    }
```

Calling the method handle

- Method handle must be inserted in place of the receiver, not on top of the stack
- Method handle is an interface, erase signature

```
Object myMethod(Object x, Object y, int z) {
    // x . invokedynamic["alpha",(Object,int)Object] (y, z)
    MethodHandle method1 = site1.target;
    if (method1 == null)
        return bootstrapMethod.invoke(x, site1, {y, z});
    return ((jdk.dyn.mh.OOI)method1).invoke(y, z);
}
```

Calling the method handle

- 2 solutions :
  - Use an adapter method
  - Perform a static analysis to insert before invokeinterface

```
Object myMethod(Object x, Object y, int z) {

    ALOAD_1      # push x
    ALOAD_2      # push y
    ILOAD_3      # push z
    INVOKE_INTERFACE Dynamic, "alpha", (Object,int)Object

    ...
```

Use an adapter method

- Adapter change argument order, can be done without adapter if one/no argument
  - One adapter by call site erased signature
  - 2 method calls before the real method

```
    // x . invokedynamic["alpha",(Object,int)Object] (y, z)

        GET_STATIC_FIELD site1
        GET_FIELD target
        DUP
        IF_NULL label

       INVOKESTATIC adapter (LObject;ILMethodHandle;)LObject;
        POP                          # remove the receiver
        GOTO end

label: # slow path
```

Insert a preamble

```
   GET_STATIC_FIELD site1     # preamble
    GET_FIELD target
    DUP
    ISTORE_0                  # insert a new local variable

    ALOAD_1      # push x (receiver)
    ALOAD_2      # push y
    ILOAD_3      # push z

    ILOAD_0
    IF_NULL label

    CHECKCAST Lmh/OIO;
    INVOKE_INTERFACE "invoke"(LObject;I)LObject;
    POP          # remove the receiver
    GOTO end

label:
    GET_STATIC_FIELD site1
    LDC caller_class
    INVOKE_STATIC "uncommonAdapter"(LObject;LObject;I...)LObject;
```

Uncommon adapter methods

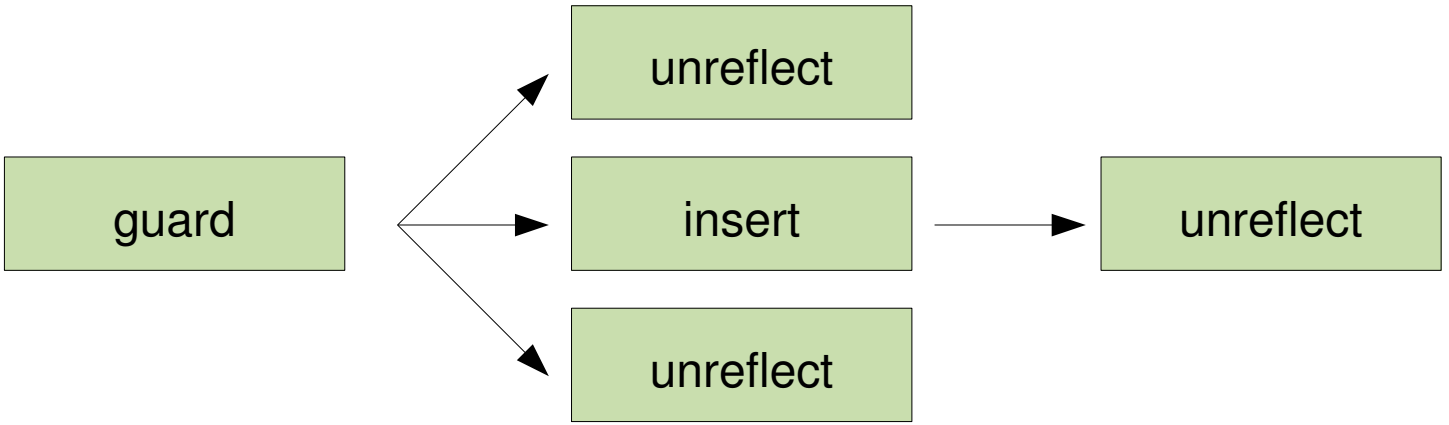
- Box arguments and call bootstrap method
- Shared! only one method by erased signature/bootstrap method
- Spread in more than one class
  - Some signatures pre-exist
- Expect few bootstrap methods
  - not more than 4/5
    - one for uni-dispatch, one for multi-dispatch, one for native call

Manipulating Method Handle

- MethodHandles defines static helpers
  - MH Unreflect(reflect.Method)
  - MH insertArgument(MH,Object)
  - MH convertArguments(MT,MH)
  - MH guardsWithTest(MH,MH,MH)
  - Etc.
- Can be combined

Manipulating Method Handle

- Avoid to create one method call by adaptation
  - chain of bytecode generator (with optim)



- Reify to one code when CallSite.setTarget()
- Allow user method handle

Inject bytecode : define class

- Unsafe backport :
  - Unsafe.defineClass()
- Agnostic backport :
  - Trap unlikely name using instrumentation agent
  - Call method Classloader.defineClass() using reflection (to bypass security)
    - Applet (only works with signed applet and next gen plugin java 6 update 10)
  - Ask the Language runtime

Open issues

- More than one backport :
  - Agnostic VM backport
  - sun.misc.Unsafe based backport ?
- Backport anonk patch to 1.5, 1.6 VM ?
- More optimization
  - Is lazy allocation of CallSite allowed ?

Lazy allocation of CallSite

- Avoid to retain unecessary objets
- Semantic not exactly the same than the spirit (John) of JSR292

- My opinion, JSR292 should not require the same CallSite object for a call site

sun.misc.Unsafe

- exists in : Hotspot, IBM VM, Apple VM, CACAO, IKVM at least.
- Bypass reflection API and security checks
- Cool methods :
- defineClass()
- staticFieldBase/staticFieldOffset/ get*/put*/putVolatile*/putOrdered*