InvokeBinder

Fluent Programming for MethodHandles

InvokeBinder

The Missing java.lang.invoke API

Me

- "Charles Oliver Nutter" <headius@headius.com>
- @headius
- Red Hat, JBoss, “Research and Prototyping” (nee Polyglot)

MethodHandle

Function and field pointers

Argument and return value manipulation

Flow control and exception handling

And the JVM optimizes it away

(Eventually…sometimes…)

So it’s all good, right?

```
MethodHandle nativeTarget = MethodHandles.findVirtual(targetClass, targetName, targetType);

// handle return value
if (nativeReturn == long.class) {
   MethodHandle returnFilter = MethodHandles.insertArguments(
                                          MethodHandles.findStatic(RubyFixnum.class,
                                              “newFixnum",
                                              MethodType.methodType(RubyFixnum.class, Ruby.class, long.class)),
                                          0,
                                          runtime);
    nativeTarget = MethodHandles.filterReturnValue(nativeTarget,
                                                   returnFilter);
```

Problem #1: Verbosity

```
MethodHandle nativeTarget = findVirtual(targetClass, targetName, targetType);

// handle return value
if (nativeReturn == long.class) {
    // native integral type, produce a Fixnum
    MethodHandle returnFilter = insertArguments(
                                        findStatic(RubyFixnum.class,
                                            “newFixnum",
                                            methodType(RubyFixnum.class, Ruby.class, long.class)),
            0,
            runtime);
    nativeTarget = filterReturnValue(nativeTarget,
                                    returnFilter);
...
```

- insert
  - drop
    - cast
      - invoke ⟵ Have to start at the target and work back

Problem #2:
Composition in reverse

```
MethodHandle nativeTarget = findVirtual(targetClass, targetName, targetType);

// handle return value
if (nativeReturn == long.class
        || nativeReturn == short.class
        || nativeReturn == char.class
        || nativeReturn == int.class
        || nativeReturn == long.class) {
    // native integral type, produce a Fixnum
    nativeTarget = explicitCastArguments(nativeTarget,
                                         methodType(long.class, targetArgs));
    MethodHandle returnFilter = insertArguments(
                                        findStatic(RubyFixnum.class,
                                            “newFixnum",
                                            methodType(RubyFixnum.class, Ruby.class, long.class)),
            0,
            runtime);
    nativeTarget = filterReturnValue(nativeTarget,
                                     returnFilter);
...
```

Problem #3:
MethodType wrangling

```
public Object tryFinally(MethodHandle target, MethodHandle post) throws Throwable {
    try {
        return target.invoke();
    } finally {
        post.invoke();
    }
}
```

ETOOMUCHCODE

try/finally

1. Like javac, finally path must be duplicated
2. Normal path invokes post, returns result of body
3. Exceptional path drops return, invokes post, re-raises exception
4. Now do this entirely on call stack, with no temp vars

```
public Object tryFinally(MethodHandle target, MethodHandle post) throws Throwable {
    try {
        return target.invoke();
    } finally {
        post.invoke();
    }
}
```

```
MethodHandle exceptionHandler = Binder
        .from(target.type().insertParameterTypes(0, Throwable.class).changeReturnType(void.class))
        .drop(0)
        .invoke(post);

MethodHandle rethrow = Binder
        .from(target.type().insertParameterTypes(0, Throwable.class))
        .fold(exceptionHandler)
        .drop(1, target.type().parameterCount())
        .throwException();

target = MethodHandles.catchException(target, Throwable.class, rethrow);

// if target returns a value, we must return it regardless of post
MethodHandle realPost = post;
if (target.type().returnType() != void.class) {
    // modify post to ignore return value
    MethodHandle newPost = Binder
            .from(target.type().insertParameterTypes(0, target.type().returnType()).changeReturnType(void.class))
            .drop(0)
            .invoke(post);

    // fold post into an identity chain that only returns the value
    realPost = Binder
            .from(target.type().insertParameterTypes(0, target.type().returnType()))
            .fold(newPost)
            .drop(1, target.type().parameterCount())
            .identity();
}

return MethodHandles.foldArguments(realPost, target);
```

Problem #4:
Complicated forms

The Problems

- Verbosity
- Composition in reverse
- MethodType must match exactly all the way through
- Many common patterns are complicated to represent
- **Argument manipulation is by absolute offsets**
- **Varargs and argument collection often need fixed size**
- **Transforms can’t be easily shared across signatures**

The Solution

InvokeBinder

https://github.com/headius/invokebinder
com.headius:invokebinder

com.headius.invokebinder

Binder

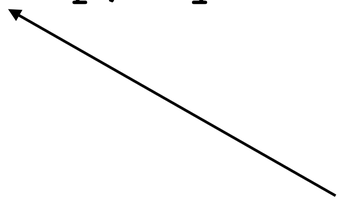
- Binder.from(type) produces a new Binder (which is immutable)
- Binder#drop, insert, permute, etc act in call-forward direction
- Binder#fold, filter, catchException, etc take additional handles
- Endpoints: invoke*, set/getField, arraySet/Get, constant, identity, nop, branch (guardWithTest)
- Utilities for larger constructs

Static Method

```
String value1 = System.getProperty("foo");

MethodHandle m1 = lookup
        .findStatic(System.class, "getProperty",
                    MethodType.methodType(String.class, String.class));

MethodHandle m2 = Binder.from(String.class, String.class)
        .invokeStatic(lookup, System.class, "getProperty");
```



Lookup can also be passed into Binder#withLookup, eliminating this param

Static Field Get

```
PrintStream out1 = System.out;

MethodHandle m3 = lookup
        .findStaticGetter(System.class, "out", PrintStream.class);

MethodHandle m4 = Binder.from(PrintStream.class)
        .getStatic(lookup, System.class, "out");
```

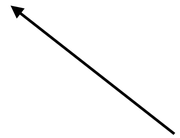
Instance Field Set

```
class MyStruct {
    public String name;
}

MyStruct ms = new MyStruct();

MethodHandle m5 = lookup
        .findSetter(MyStruct.class, "name", String.class);

MethodHandle m6 = Binder.from(void.class, MyStruct.class, String.class)
        .setField(lookup, "name");
```



No target type; Binder knows it already!

Deleting Args

```
MethodHandle m9 = lookup
        .findStatic(Demo1.class, "twoArgs",
                MethodType.methodType(String.class, String.class, String.class));
m9 = MethodHandles.dropArguments(m9, 2, String.class);

MethodHandle m10 = Binder.from(String.class, String.class, String.class,
String.class)
        .drop(2)
        .invokeStatic(lookup, Demo1.class, "twoArgs");
m10.invoke("one", "two", "three"); // => "[one,two]"
```

Permute Args

```
MethodHandle m11 = lookup
        .findStatic(Demo1.class, "twoArgs",
                MethodType.methodType(String.class, String.class, String.class));
m11 = MethodHandles.permuteArguments(
        m11,
        MethodType.methodType(String.class, String.class, String.class, int.class),
        1, 0);

MethodHandle m12 = Binder.from(String.class, String.class, String.class, int.class)
        .permute(1, 0)
        .invokeStatic(lookup, Demo1.class, "initials");

m12.invoke("one", "two", 3); // => "[two,one]"
```

Fold

```
MethodHandle m13 = lookup
        .findStatic(Demo1.class, "threeArgs",
                MethodType.methodType(String.class, String.class,
                                      String.class, String.class));
MethodHandle combiner = lookup
        .findStatic(Demo1.class, "initials",
                MethodType.methodType(String.class, String.class,
String.class));
m13 = MethodHandles.foldArguments(m13, combiner);

MethodHandle m14 = Binder.from(String.class, String.class, String.class)
        .fold(
                Binder
                        .from(String.class, String.class, String.class)
                        .invokeStatic(lookup, Demo1.class, "initials")
        )
        .invokeStatic(lookup, Demo1.class, "threeArgs");

m14.invoke("Charles", "Nutter"); // => ["CN", "Charles", "Nutter"]
```

Filter

```
MethodHandle m15 = lookup
        .findStatic(Demo1.class, "twoArgs",
                MethodType.methodType(String.class, String.class,
                                      String.class));
MethodHandle filter = lookup
        .findStatic(Demo1.class, "upcase",
                MethodType.methodType(String.class, String.class));
m15 = MethodHandles.filterArguments(m15, 0, filter, filter);

MethodHandle m16 = Binder.from(String.class, String.class, String.class)
        .filter(0,
                 Binder.from(String.class, String.class)
                       .invokeStatic(lookup, Demo1.class, "upcase")
        )
        .invokeStatic(lookup, Demo1.class, "twoArgs");
m16.invoke("hello", "world"); // => ["HELLO", "WORLD"]
```

Boolean Branch

```
MethodHandle m21 = Binder.from(String.class, int.class, String.class)
        .branch(
                Binder.from(boolean.class, int.class, String.class)
                        .drop(1)
                        .invokeStatic(lookup, Demo1.class, "upOrDown"),

                Binder.from(String.class, int.class, String.class)
                        .drop(0)
                        .invokeStatic(lookup, Demo1.class, "upcase"),

                Binder.from(String.class, int.class, String.class)
                        .drop(0)
                        .invokeStatic(lookup, Demo1.class, "downcase")
        );

m21.invoke(1, "MyString"); // => "MYSTRING"
m21.invoke(0, "MyString"); // => "mystring"
```

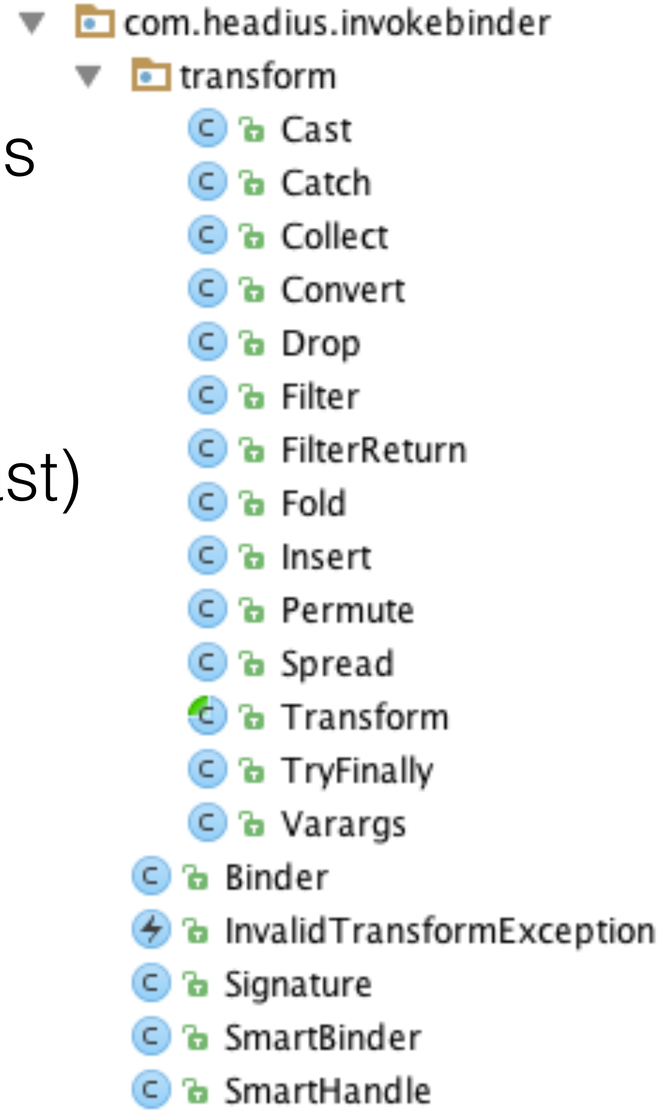
Debugging!

```
handle = Binder
         .from(String.class, Object.class, double.class, String.class)
         .dropFirst(2)
         .printType()
         .insert(1, "world")
         .invoke(target);
```

No-op, just prints out current type

```
MethodHandle handle = Binder
        .from(void.class, String[].class)
        .tryFinally(post)
        .invokeStatic(LOOKUP, BinderTest.class, "setZeroToFoo");
```

- Binder aggregates MethodType, list of transforms
- While building
  - MethodType is transformed (and verified or cast)
  - Transform stack pushes down a transform
- At endpoint transform stack is played in reverse
  - And verified or cast (perhaps redundant)



```java
/**
 * Filter incoming arguments, starting at the given index, replacing
 * each with the result of calling the associated function in the
 * given list.
 *
 * @param index     the index of the first argument to filter
 * @param functions the array of functions to transform the arguments
 * @return a new Binder
 */
public Binder filter(int index, MethodHandle... functions) {
    return new Binder(this, new Filter(index, functions));
}
```

```
public class Filter extends Transform {

    private final int index;
    private final MethodHandle[] functions;

    public Filter(int index, MethodHandle... functions) {
        this.index = index;
        this.functions = functions;
    }

    public MethodHandle up(MethodHandle target) {
        return MethodHandles.filterArguments(target, index, functions);
    }

    public MethodType down(MethodType type) {
        for (int i = 0; i < functions.length; i++) {
            type = type.changeParameterType(index + i, functions[i].type().returnType());
        }
        return type;
    }

    public String toString() {
        return "fold args from " + index + " with " + Arrays.toString(functions);
    }
}
```

BUT WAIT THERE’S MORE!

Argument juggling sucks

If I need to change signature, all my offsets break

```
MethodHandle handle2 = Binder
        .from(Subjects.StringIntegerIntegerIntegerString.type())
        .collect(1, 3, Integer[].class)
        .invoke(Subjects.StringIntegersStringHandle);
```

Need a representation of an argument list…not just types

Signature

- Combines MethodType with argument names
- Perform argument manipulations by name, not offset
- Regex-based operations: collect, drop, permute
- Conversion: signature1.to(signature2) => permute offsets
- Type checking, pretty toString, other niceties

```
Signature sig = Signature
        .returning(String.class)
        .asFold(Object.class);

assertEquals(Object.class, sig.type().returnType());
```

```
Signature sig = Signature
        .returning(String.class)
        .appendArg("obj", Object.class)
        .appendArg("num", int.class)
        .insertArg("num", "flo", float.class);

assertEquals("(Object obj, float flo, int num)String", sig.toString());
```

```
private static final Signature stringObjectInt = Signature
        .returning(String.class)
        .appendArg("obj", Object.class)
        .appendArg("num", int.class);
```

```
Signature sig = stringObjectInt
        .appendArg("flo", float.class)
        .appendArg("dub", double.class)
        .permute("obj", "dub");

assertEquals("(Object obj, double dub)String", sig.toString());
```

```
int[] permuteInts = stringObjectInt
        .appendArg("flo", float.class)
        .appendArg("dub", double.class)
        .to(stringObjectInt);

assertArrayEquals(new int[] {0, 1}, permuteInts);
```

```
int[] permuteInts = stringObjectInt
        .appendArg("flo", float.class)
        .appendArg("dub", double.class)
        .to(".*o.*");

assertArrayEquals(new int[] {0, 2}, permuteInts);

permuteInts = stringObjectInt
        .appendArg("flo", float.class)
        .appendArg("dub", double.class)
        .to("num", "dub");

assertArrayEquals(new int[] {1, 3}, permuteInts);
```

```
Signature sig = stringObjectInt
        .appendArg("flo", float.class)
        .appendArg("dub", double.class)
        .exclude("obj", "dub");

assertEquals("(int num, float flo)String", sig.toString());
```

```
public static final Signature StringIntegerIntegerIntegerString = Signature
        .returning(String.class)
        .appendArg("a", String.class)
        .appendArg("b1", Integer.class)
        .appendArg("b2", Integer.class)
        .appendArg("b3", Integer.class)
        .appendArg("c", String.class);

Signature oldSig = Subjects.StringIntegerIntegerIntegerString;
Signature newSig = oldSig.collect("bs", "b.*");

assertEquals(Integer[].class, newSig.argType(1));
assertEquals("bs", newSig.argName(1));
assertEquals(3, newSig.argCount());
assertEquals("c", newSig.argName(2));
```

Binder

SmartBinder

Signature

SmartBinder

- All operations from Binder, but Signature-aware
- Variable-length argument lists
- Regex-based transforms
- Result is a SmartHandle (MethodHandle + Signature)

```
MethodHandle StringIntegersStringHandle = Binder
        .from(String.class, String.class, Integer[].class, String.class)
        .invokeStaticQuiet(LOOKUP, Subjects.class, "stringIntegersString");
```

```
MethodHandle StringIntegersStringHandle = Binder
        .from(String.class, String.class, Integer[].class, String.class)
        .invokeStaticQuiet(LOOKUP, Subjects.class, "stringIntegersString");

Signature oldSig = Subjects.StringIntegerIntegerIntegerString;

SmartHandle handle = SmartBinder
        .from(oldSig)
        .collect("bs", "b.*")
        .invoke(Subjects.StringIntegersStringHandle);
```

```
Signature oldSig = Subjects.StringIntegerIntegerIntegerString;

SmartHandle handle = SmartBinder.from(oldSig)
        .drop("b1")
        .drop("b2")
        .drop("b3")
        .insert(1, "bs", new Integer[]{1, 2, 3})
        .invoke(Subjects.StringIntegersStringHandle);
```

```
MethodHandle target = Subjects.concatHandle();
MethodHandle filter = MethodHandles.insertArguments(Subjects.concatHandle(), 1, "goodbye");
MethodHandle handle = SmartBinder
        .from(String.class, arrayOf(“arg1", “arg2”), String.class, String.class)
        .filter("arg.*", filter)
        .invoke(target).handle();
```

```
Signature.returning(IRubyObject.class)
         .appendArg("context", ThreadContext.class)
         .appendArg("caller", IRubyObject.class)
         .appendArg("self", IRubyObject.class)
         .appendArg("arg0", IRubyObject.class)
         .appendArg("arg1", IRubyObject.class)
         .appendArg("arg2", IRubyObject.class)
```

```
SmartBinder.from(site.signature)
        .permute("context", "arg.*")
```

```
MethodHandle stringInt = Binder
        .from(String.class, int.class)
        .invokeStaticQuiet(LOOKUP, Integer.class, “toString");

MethodHandle handle = SmartBinder
        .from(String.class, "i", int.class)
        .fold("s", stringInt)
        .dropLast()
        .identity()
        .handle();

assertEquals(MethodType.methodType(String.class, int.class), handle.type());
assertEquals("15", (String)handle.invokeExact(15));
```

Status

- Binder supports nearly all j.l.i.MethodHandle(s) operations
- Signature has a wide array of transforms
- SmartBinder has what I needed

TODO

- Analyze transform chain to emit more efficient handles
- Fill out missing functionality, tests, docs
- More utility forms
- Generate bytecode or .java callable for non-indy
- “Interpreted” [Smart]Binder#call like LFs for non-bytecode-loading
- More extensive debugging (current transform stack, visual graph)

Thank you!

- "Charles Oliver Nutter" <headius@headius.com>
- @headius
- https://github.com/headius/invokebinder
- Maven: com.headius:invokebinder