Part 2

Demystifying invokedynamic

Learn how to master the details of invokedynamic.

JULIEN PONGE

BIO

Part 1 of this series served as a gentle introduction to invokedynamic. It showed how to obtain *method handles*, bind them to *call sites* at runtime, and then emit Java Virtual Machine (JVM) bytecode with actual invokedynamic instructions. Part 1 introduced you to invokedynamic, but the finer details of the java.lang.invoke API remain to be explored.

In particular, the method handle combinator chains that we built in Part 1 remained fairly linear. Indeed, we had a method handle to a target method, and we sometimes performed arity and type conversions to adapt a call site and a target of different types, for example, adapting a call site of type (Object, Object)Object to a method handle of type (int, int[])int.

This article shows that more-elaborate logic can be put into method handle chains. We will see that values can be filtered, conditional branching can be done, and exceptions can be caught. A very important point is that you can create a lot of programming logic by combining method handles. In effect, correctly used, the MethodHandle API is versatile enough to abstract us away from ever explicitly having to use bytecode. This is very powerful. We will present each of the corresponding combinators through simple examples and explain how they could be used in the context of real dynamic language implementations.

Dancing with the Types

The need to do type conversions quickly arises when implementing a language on the JVM, especially for Java interoperability purposes. Fortunately, doing conversions is often a simple matter of using filtering combinators.

Suppose that we have the RichInt class shown in **Listing 1**. It wraps a primitive int type and adds a square method. A full implementation of an "improved integer" in a language would, of course, provide more than the square method, but this implementation shows the basic idea.

We can obtain a method handle to square and invoke it as long as we are passing an instance of RichInt as the receiver, as shown in **Listing 2**. Wouldn't it be convenient to pass int values instead, invoke square on them, and get back an int value?

This kind of conversion can be performed using the filterArguments and filterReturnValue combinators of the MethodHandles class. filterArguments takes a position in the arguments list, followed by a variable number of method handles. Each method handle filters an argument starting from that position. filterReturnValue works similarly, albeit with a single method handle.

We perform the method handle invocation using invokeExact, which requires that the arguments and return value types exactly match those of the method handle. This is not always possible (that is, known at compile time), in which cases invoke and invokeWithArguments can be used instead. The difference is that those methods need to perform type conversions, such as boxing and unboxing primitive types. It is preferable to use invokeExact whenever possible, because it has better performance.

Creating a new instance of RichInt does the conversion from an int to a RichInt. Calling the intValue virtual method of RichInt does the opposite. The code in **Listing 3** illustrates how it is done. By

PHOTOGRAPH BY MATT BOSTOCK/GETTY IMAGES

doing so, we can treat int like a RichInt, call its square method, and return an int.

Branching with guardWithTest

Dynamic language implementers frequently need to map if/then/else conditional branching at a call site, for instance, to dispatch a method invocation to different targets based on the receiver type or depending on some argument value. The previous article showed only linear combinator chains, but fortunately, it doesn't have to be that way.

The MethodHandles class has a method called guardWithTest that is used to create a combinator with conditional branching. To do so, it takes three method handles as its input:

- A predicate method handle that implements the guard condition and returns a Boolean
- A method handle that is executed when the guard returns true
- A method handle that is executed when the guard returns false

While the guard method handle consumes the arguments that are being passed on the combinator chain, it might not consume all of them. For instance, it might evaluate its condition based only on the first parameter. The true and false branch method handles need to process all arguments, either directly or through further combinator-based adaptation.

Let's build an example in a class called Guards. We want to create a chain of method handle combinators for a call site with an (Object) String signature, where we pass an object and obtain a string representation for it. We expect specialized representations for Integer and String arguments along with a general-purpose representation for any other type. To do that, we provide three static methods (see **Listing 4**).

We will take advantage of guardWithTest to create a branching structure so that the target is any of these three methods depending on the argument class. Because there are three cases, we need two guard methods, which are shown in **Listing 5**.

Note that we suppose that the value won't be null. We could have used the instanceof operator, but it might be slow with deep type hierarchies, so comparing references against getClass() (and maybe null, too) is the best bet in all cases where an instanceof check can be avoided. Our test method is shown in **Listing 6**.

We first obtain a lookup object, and then we get method handles for the target and guard methods. Because the target methods have different signatures, we need to

LISTING 1 | LISTING 2 | LISTING 3 | LISTING 4 | LISTING 5 | LISTING 6

```
public class RichInt {

 private final int value;

 public RichInt(int value) {
  this.value = value;
 }

 public int intValue() {
  return value;
 }

 public RichInt square() {
  return new RichInt(value * value);
 }

 @Override
 public String toString() {
  return "RichInt{" + "value=" + value + "}";
 }
}
```

Download all listings in this issue as text

adapt each to the call site type, which is (Object)String, using the asType combinator. This works because it is obviously possible to deal with an Integer or a String as an Object.

Remember that method handle chains are type-checked when built: failing to adapt the signatures with asType would raise an exception at runtime. Last but not least, never forget that boxing and unboxing primitive types are costly operations that, as of yet, cannot always be eliminated by the Java HotSpot VM.

Finally, we build a branching method handle using a first guardWithTest. If the object type is an Integer, the intDisplay branch is taken; otherwise, the code falls back to a second guardWithTest construct that looks for a String. It takes the stringDisplay branch or falls back to the general-purpose objectDisplay.

Running this example yields the following console output:

```
Integer = 1
String = foo
Object = true
```

In real dynamic languages, guardWithTest is often used to build inline caches with self-modifying mutable or volatile call sites. This idea comes from the days of Smalltalk. Briefly, the idea is to cache the target method handle when doing method invocations, as long as the receiver objects encountered at a call site remain of the same type. Doing this yields dramatically better performance than performing a method handle lookup for each invocation.

Inline caches can also be used in adaptive runtimes, such as JVMs. The just-in-time (JIT) compiler generates native code outside of the interpreter that makes speculative assumptions about the code being executed, such as a receiver type being of a certain type or locks never being used in a multithreaded context. Because such assumptions might be invalidated, generated native-code blocks (the "fast" path) have guards to fall back to previously optimized code or the interpreter (the "slow" path).

Showing an example of an inline cache in this article would be too lengthy, and many details of an inline cache's implementation depend on the target language's runtime specificities. Instead, you are encouraged to look at the example by Rémi Forax in the JSR 292 Cookbook project and then experiment with an adaptation for your own language's runtime.

LISTING 7

```
static class Runner extends Thread {

 final MethodHandle runHandle;
 final MethodHandle waitHandle;

 Runner(MethodHandle runHandle, MethodHandle waitHandle) {
  super();
  this.runHandle = runHandle;
  this.waitHandle = waitHandle;
 }

 @Override
 public void run() {
  while (true) {
   try {
    System.out.println((String) runHandle.invokeExact());
    waitHandle.invokeExact();
   } catch (Throwable e) {
    throw new RuntimeException(e);
   }
  }
 }
};
```

Download all listings in this issue as text

Switch Points

The SwitchPoint class is another useful addition to the language implementer toolbox when it comes to branching, especially when a target should be executed as long as "some condition" is met, forever reverting to a fallback target as long as the condition no longer holds.

A typical case would be a dynamic language in which methods in object instances can be replaced at runtime. A switch point holds for as long as the method is stable, and then it gets invalidated once it is replaced. It is equivalent to a guardWithTest-based construction, albeit sometimes more straightforward to use when many call sites depend on the same condition not changing.

Listing 7 is an example of a runner thread that takes two method

handles. The code is an infinite loop in which runHandle is a method handle of type ()String. It yields a string value whenever it is called. waitHandle is of type ()void and is supposed to put the thread to sleep for some arbitrary time.

Our test method will exercise the thread in two phases:

- runHandle will return "a" every 500 milliseconds for about 5 seconds.
- Then runHandle will return "b" every 1,200 milliseconds afterward.

To do that, we first define the static methods in **Listing 8**, which will later be used as method handle targets. Our test method is shown in **Listing 9**.

As usual, we first construct method handles to our target methods. We then instantiate two switch points: one for the runner target and one for the waiting target.

A SwitchPoint gives us a method handle by invoking the guardWithTest method. It is very similar to directly using a guardWithTest combinator, as we previously did, except that it takes only two method handles. The first will be executed as long as the switch point is valid; the second is executed as soon as the switch point is no longer valid.

No explicit guard code is executed each time the SwitchPoint is called. The same semantics could be achieved using guardWithTest with a guard that evaluates against some reference object or expression.

We can then start the thread, passing it the method handles for the switch point targets. The thread is then put to sleep for five seconds, after which we invalidate the two switch points using the invalidateAll static method of the SwitchPoint class. At this point, "a" gets printed every 1,200 milliseconds until the application is terminated.

Invalidating a set of switch points causes all of them to switch their target branches. This is immediately reflected across concurrent threads with volatile semantics. While this is a potentially costly operation, it can be more effective to use a SwitchPoint in combination with a MutableCallSite than to rely on a VolatileCallSite just to ensure that changes get propagated as soon as the invalidation happens. Again, the choice is best made based on performance analysis against the invalidation rate of your call sites.

The choice of a SwitchPoint versus a guardWithTest construct also depends on how invalidation is done. In situations in which the invalidation depends only on the receiver type or arguments, a guardWithTest is a good choice. Invalidation is a local call-time decision and applies for, say, inline cache constructs.

Now, consider the case of a dynamic language in which methods can be modified not just on class definitions but also on objects. A good example is the Ruby programming language, in which you can define a class as follows:

```
class Foo
  def bar
    "bar"
  end
end

foo = Foo.new
puts foo.bar # prints bar
```

LISTING 8 LISTING 9

```
static String a() {
 return "a";
}

static String b() {
 return "b";
}

static void slow() {
 try {
  Thread.sleep(1200);
 } catch (InterruptedException e) {
  throw new RuntimeException(e);
 }
}

static void fast() {
 try {
  Thread.sleep(500);
 } catch (InterruptedException e) {
  throw new RuntimeException(e);
 }
}
```

Download all listings in this issue as text

This calls method bar on object foo, which is an instance of class Foo. Ruby also allows you to redefine bar on the foo instance:

```
def foo.bar
  "[bar]"
end
puts foo.bar # prints [bar]
```

In such cases, there are many call sites that can depend on a given method to remain valid, such as calls to bar in the previous example. Invalidation happens when the instance method is changed. This could be done with guardWithTest and an instance-wide condition as a guard, but SwitchPoint is much better. Indeed, all call sites can be invalidated at once under volatile semantics and, more interestingly, the overhead of a SwitchPoint is virtually zero for as long as it does not need to be invalidated.

Not So Exceptional Exceptions

Method handle targets can throw exceptions, just like any Java method. Because call sites are dynamically bound at runtime, it is likely that you will find yourself in a situation in which the target raises an exception, but the bytecode surrounding the invokedynamic instructions makes no assumption about that exception to be thrown.

The MethodHandles class provides a combinator called catchException that can mimic a try/catch construction at the method-handle level. It takes a method handle that is the regular target, an exception class, and a method handle to be invoked as a catch block equivalent. Similarly, there is a throwException combinator in the MethodHandles class that can be used to throw exceptions at a call site.

Our example is inspired by the *safe navigation operator* (?.) found in the Groovy programming language, which can be used to avoid the infamous NullPointerException when chaining method calls. Indeed, if a method in the middle of the chain returns null, the next call raises a NullPointerException. Placed as a method invocation operator, the safe navigation operator avoids this issue and returns null instead:

```
def mrbeanPhone =
person?.friends["mrbean"]?.phone
```

If friends is null, or if there is no "mrbean" entry, mrbeanPhone is null. The same code with just "." could raise a NullPointerException in such cases.

Let's build runtime support for a similar construction in which we return a value when the regular target throws an exception. Consider the target method shown in **Listing 10**.

LISTING 10 LISTING 11 / LISTING 12

```
static String stringAndHashCode(String prefix, Object obj) {
  return new StringBuilder(prefix)
        .append(obj.toString())
        .append(" #")
        .append(obj.hashCode())
        .toString();
}
```

Download all listings in this issue as text

Obviously, passing null as the obj parameter will raise NullPointerException. In such cases, we would like to return a different representation using the target method handle shown in **Listing 11**, which works with any call site signature as long as the arguments are being wrapped into an array. We can leverage catchException as shown in **Listing 12**.

The method handle to stringAndHashCode is straightforward. The one to npeHandler requires two adaptations using intermediate combinators: one to collect all arguments into an array of two arguments (asCollector) and then a call site type adaptation. Once this is done, catchException yields to us a method handle that properly routes a method call dispatch to stringAndHashCode,

unless a NullPointerException is raised. Executing the program gives the following output:

```
>>> 666 #666
>>> Hey! #2245797
[null] -> [>>> , null]
[null] -> [null, null]
```

The interesting point is that exception handling and throwing can be done using combinators. This does not require the corresponding bytecode-level support, which is, incidentally, more complicated, because you need to delimit a try lock and specify an exception type and a jump to a catch block. Multiple catch clauses require such constructs to be nested in bytecode. By contrast, a combinator-based solution comes up with simple bytecode and a composition of functions.

HANDLING LOGIC

Elaborate logic can be put into method handle chains; for example, values can be filtered, conditional branching can be done, and exceptions can be caught.

The question of performance is worth raising. Why not use a guardWithTest to check for a null value instead of waiting for an exception to be thrown? As usual, your mileage varies depending on how often you expect exceptions to be raised. As their name suggests, exceptions should be rare. In such cases, you can gamble on a fast dispatch to the target method handle and a slower dispatch in the rare cases in which an exception happens to be thrown. Always benchmark your solutions.

Conclusion

This article concludes a series on invokedynamic. We covered portions of the java.lang.invoke APIs that allow more-elaborate logic to be made out of method handles. You should now be ready to explore more of invokedynamic by yourself. This is a fairly low-level API, but it comes with everything you need for supporting late binding while not confusing the JVM when it comes to adaptive optimizations.

Last but not least, higher-level libraries, such as the Dynalink linker, can greatly help by providing some common runtime plumbing logic. Indeed, developing runtime support for a dynamic language using the java.lang.invoke API is anything but trivial. Dynalink provides ready-to-use building blocks such as ready-made and efficient guarded invocations, type conversions, overloaded method resolution, variable arguments handling, and more. Dynalink also offers a metaobject protocol library that can greatly facilitate the interoperability between Java and Dynalink-powered JVM languages such as Oracle's Nashorn.

Will you contribute to the invokedynamic support of an existing dynamic language for the JVM? Will you design your own language? Will you make fun hacks with invokedynamic? March on—the rest is history.

Acknowledgements. *The author would like to thank Marcus Lagergren for his thoughtful and positive feedback during the review process.* `</article>`

LEARN MORE

- John R. Rose, "Bytecodes Meet Combinators: invokedynamic on the JVM"
- "JooFlux: Hijacking Java 7 InvokeDynamic to Support Live Code Modifications" by Julien Ponge and Frédéric Le Mouël
- "JSR 292 Cookbook," a collection of common invokedynamic patterns by Rémi Forax
- Dynalink: Dynamic Linker Framework on the JVM by Attila Szegedi