JULIEN PONGE

BIO

Part 1

Demystifying invokedynamic

Learn how to use invokedynamic in your code.

The release of Java 7 brought substantial additions to the language, such as the new try-with-resources statement, multi-catch clauses, and the diamond operator. New APIs have been introduced, too, with notable examples being the fork-join framework for parallel algorithms and NIO.2 to better deal with native file system capabilities.

Another much-touted change is the introduction of a new Java Virtual Machine (JVM) bytecode instruction called invokedynamic. It was introduced to facilitate the implementation of dynamic languages on top of the JVM, which is an attractive target platform with a robust adaptive runtime that, in recent years, has seen many languages flourish. Some are ports of existing languages to the JVM (for example, Rhino, Jython, and JRuby) while some are new (for example, Groovy, Clojure, and Fantom).

Julien Ponge chats with Oracle's Stephen Chin about invokedynamic as part of Chin's NightHacking Tour across Europe.

This article is a gentle introduction to using invokedynamic in your own code. The topic in itself is fairly rich and will mostly appeal to language and middleware implementers. We will only scratch the surface of the provided APIs and possible usages, but by the end you should have a basic technical understanding of invokedynamic.

Note: The source code for the examples in this article can be downloaded here.

Invoking Methods on the JVM

The JVM has traditionally offered four instructions for invoking methods. Each points out an owner class, a name, and a signature description. As an example, given a static method void baz(int a, String s) in class foo.Bar, invoking the method requires a reference with foo/Bar as the owner, baz as the name, and (ILjava/lang/String;)V as the JVM internal representation signature. Invoking a method requires placing a *receiver* object—that is, the object on which the method shall be invoked—on the stack. The only exception is the case of static methods, which do not have a receiver.

The following are the four invoke instructions:

- invokestatic, which is used for static methods.
- invokevirtual, which is used for methods that require dynamic dispatch, that is, public and protected methods.
- invokeinterface, which is similar to invokevirtual, except that the method dispatch is based on an interface type.
- invokespecial, which is for the other types of methods: constructors and private virtual methods. The method dispatch is

PHOTOGRAPH BY MATT BOSTOCK/GETTY IMAGES

not performed based on the receiver type but rather based on the specified owner class (this is how the super reference works).

The design of the JVM makes perfect sense because it was originally created for a strongly typed language, Java. In such a language, it is sufficient to check method invocations at compile time. Hence, it is both easy and natural to produce bytecode where invocations unambiguously specify a target. Signature descriptors exactly match those of the target methods.

The Case of Dynamic Languages

Things are slightly different with loosely typed dynamic languages such as Groovy, JRuby, or Jython. In such languages, types are most often not checked at compile time. It is up to the programmer to pass a "good" type, and failing to do so results in an error at runtime. Thus, in such languages, the closest "accurate" JVM type for a method argument is more often than not java.lang.Object. It easily follows that this complicates type analysis (which has to be done mostly at runtime) and affects performance.

Dynamic languages also often support a form of *monkey-patching*, that is, the ability to add methods and fields to and remove methods and fields from existing classes. Some languages even support doing so at the instance level. For all these reasons, it is necessary to defer the resolution of classes, methods, and fields to the runtime. It is also necessary to dynamically update the resolved entities at runtime. Also, there is often a need for adapting a method invocation to a target of a different signature.

Dynamic languages on the JVM have traditionally built ad hoc runtime support on top of the JVM. This includes using wrapper type classes; building custom *inline caches* for invoking methods; interpreting, tracing, and generating specialized bytecode; or using hash tables to provide dynamic symbol resolution. In doing so, such languages generate bytecode with entry points to the runtime in the form of method calls using either of the four existing bytecode instructions for invocations.

Here Comes invokedynamic

The main problem is that of performance. The bytecode generated by dynamic languages tends to require several actual JVM method invocations in order to perform one in the dynamic language. There is extensive use of reflection and dynamic proxies, which have a performance cost. Last but not least, such code tends to target common language runtime classes and methods.

In short, because there are many different execution paths with great variance in actual types, the adaptive runtime—just-in-time (JIT) compilation—has trouble applying optimizations. Although dynamic language implementers have used clever techniques, it remains true that the JVM has issues understanding what such code does, thus hindering performance. The bytecode emitted by such languages lacks information and patterns that the JVM knows how to optimize, which are typically present in bytecode generated from Java. One can argue that this is a matter of making the JVM more aware of new patterns, but the fact that bytecode is typed also gets in the way.

The introduction of invoke dynamic to the JVM specification is recognition of the fact that dynamic languages bring value to the Java ecosystem and, as such, the JVM needs to provide better support for them. This new bytecode instruction is able to defer the invocation target resolution to some runtime logic, and it provides support for dynamically changing call site targets. It also has support in the JVM internals so that it can be better optimized by the JIT compiler.

Method Handles

Method handles are analogous to the function pointers found in languages such as C in the sense that they point to a target and can be used to invoke it. Method handles can be obtained and manipulated by using the new java.lang.invoke API.

Getting and invoking. Listing 1 is an example in which we obtain a method handle to the Boolean startsWith(String) instance method of the java.lang.String class and then invoke it.

We first need an object of type MethodHandles.Lookup to search for targets that include virtual methods, static methods, special methods, constructors, or field accessors.

A lookup object is dependent on the invocation context of a call site. As an example, a lookup made in a class A cannot see the private methods of a class B. When invoking the lookup method in **Listing 1**, we obtain a lookup object that respects the accessibility rules from within the main method of Sample1. The preferred way to look up only public methods is to call MethodHandles.publicLookup.

Once this is done, we can call the findVirtual method. The first argument is the class in which to look for the method. The second argument is the method name,

while the third one is an instance of MethodType. Here, the method that we are looking for returns a Boolean and takes a string as a parameter.

Note that we actually pass *three* arguments to build the type: the return type followed by the argument types. Indeed, startsWith is a virtual method, so its first call argument must be the *receiver* type, which here is a string, too.

Finally, we perform two invocations by printing whether *Java* and *Groovy* start with *J*.

java.lang.reflect integration. Just looking at the example in **Listing 1**, you could think that method handles can be used in place of reflective method invocations. Worse, you could think that java.lang.invoke is a possible replacement for reflection. This is far from the truth, because both APIs complement each other.

You need to know the exact signatures of the elements to be looked up using MethodHandles.Lookup. There is no way to look for the methods, constructors, and fields of a class just by their names without knowing their precise signatures. To do that, you should simply stick to the well-known reflective APIs in java.lang.reflect, and then use either of the unreflect methods found in MethodHandles.Lookup that provide bridging between reflection and method handles. This is especially useful when writing dynamic applications where you need to look up elements by name, but the exact signatures are not known.

- The example in **Listing 1** with method handles can be equivalently coded as shown in **Listing 2**, in which the target method handle is found using reflection.

If we were to stop our explorations of invokedynamic here, there would not be many differences between an invocation using reflection and one using a method handle. Still, it should be noted that every reflective invocation induces verifications at the JVM level. This is not the case with method handles, because those verifications happen only when method handles are built.

Combinators. While method handles can directly map to class methods, it is possible to build more-elaborate method handles by combining several of them. This is useful in many situations, such as when there is a need to adapt a call site and a target of different types, to process arguments and return values, or to provide conditional branching. In this regard, guards are an especially useful building block for constructing more-complex call-site logic as is found in inline caches.

LISTING 1 LISTING 2 LISTING 3

```
package sample1;

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodHandles.Lookup;
import static java.lang.invoke.MethodType.methodType;

public class Sample1 {
 public static void main(String... args) throws Throwable {
  Lookup lookup = MethodHandles.lookup();

  MethodHandle startsWith = lookup.findVirtual(
   String.class, "startsWith",
       methodType(boolean.class, String.class));
  // Prints "true"
  System.out.println(startsWith.invokeWithArguments(
   "Java", "J"));
  // Prints "false"
  System.out.println(startsWith.invokeWithArguments(
   "Groovy", "J"));
 }}
```

Download all listings in this issue as text

Let's start with a simple example. In **Listing 2**, we obtained a method handle to the startsWith method of the String class. Now, in **Listing 3**, let's instead build a method handle that checks whether a string given as a receiver starts with *J*.

To do this, we need to perform a form of *partial function application* that binds the first invocation argument of the method to the constant value *J*. The java.lang.invoke.MethodHandles class provides many *combinators*, that is, methods to derive new method handles from existing ones. In our case, we take advantage of insertArguments to bind the element at index 1. This method works as follows: it takes a method handle, an index where arguments will be inserted, and a variable arguments list of values to insert.

The other combinators can remove elements; permute ele-

ments; wrap arguments to or unwrap arguments from an array; perform type conversions; and more. While it is beyond the scope of this article to provide a complete overview of each of these, you will discover some commonly used ones if you keep reading.

Example of call site and target adaptation. You will often run into the problem of adapting call sites and targets when working with invokedynamic. Suppose that you have a method printAll declared as shown in **Listing 4**.

Clearly, printAll is of type (String, Object[])void. In Java, invocations of this method would call sites such as those shown in **Listing 5**.

Now suppose that we have call sites expecting something different, such as a method that takes a fixed number of parameters and no prefix string. Clearly, the signatures do not match. A fix would be to introduce an adaptation method such as that shown in **Listing 6**.

Such an adaptation is simple to perform using method handles and combinators, as shown in **Listing 7**.

First, target is a direct method handle to the printAll static method. We use the bindTo combinator to bind the first argument to a value. In the case of a static method, this is effectively the first argument, while in the case of an instance method, the object would be the receiver.

Then, the asCollector combinator wraps arguments into a collecting array. Here, we bind four arguments into an array of Object instances. Using the code in **Listing 8**, we can check that the resulting method handle is of type (Object,Object,Object,Object)void and that invoking it with arguments of the expected arity works.

Performance-wise, you should use invokeExact rather than invokeWithArguments whenever you can. invokeExact requires the argument types to be exactly like those of the method handle type descriptor, and it dispatches the call much faster. In contrast, invokeWithArguments performs type checks and conversions.

A classic optimization for virtual methods is to build *inline caches* where the target method handle is cached as long as the receiver type remains stable. You should limit the cache depth in case the call site happens to be megamorphic; otherwise, you would create too many method handles arranged in a chain of guarded tests, which further degrades the performance figures.

Bootstrapping

At this point, you should have a basic understanding of what method handles are. We can now turn to the bootstrapping side of invokedynamic.

CallSite objects. Call sites need to be bound at runtime using the java.lang.invoke API. To do that, each invokedynamic instruction refers to a *bootstrap method* that is executed the first time the call site invocation is executed.

The bootstrap method must return an instance of java.lang.invoke.CallSite. As the name suggests, such objects represent a call site, and they are bound to a target method handle, which may, in turn, be a chain of combinators.

The CallSite can be thought of as the container for the MethodHandle. Once it has been bound, a call site always references the same CallSite instance. Although the CallSite itself will refer to the same program point throughout its lifecycle, call sites may allow their target method handle to be redefined, providing a way to swap out old code with new code on the fly, without regenerating the bytecode for an entire method.

The java.lang.invoke package offers three concrete implementations of the abstract CallSite class, and you can subclass them, too:

- ConstantCallSite is for call sites whose target method handle never changes.
- MutableCallSite has object field semantics, which means that the target method handle can be changed.
- VolatileCallSite is similar to MutableCallSite but with volatile reference semantics.

If you need a call site where the target can be changed, choosing between MutableCallSite and VolatileCallSite is a matter of understanding the Java memory model and weighing the consequences of choosing one versus the other. In multithreaded environments, a *volatile call site target*

LISTING 4 LISTING 5 LISTING 6 LISTING 7 LISTING 8

```
static void printAll(String prefix, Object... values) {
 for (Object value : values) {
  System.out.println(prefix + value);
 }
}
```

Download all listings in this issue as text

change will be immediately visible to all threads, while with ordinary field semantics, thread caching occurs and requires explicit synchronization (see the static syncAll method in MutableCallSite). Of course, there are performance implications, too.

Bootstrap methods. invokedynamic bootstrap instructions are bound by invoking a static method that returns a CallSite object that must take at least three parameters:

- A MethodHandles.Lookup object that is bound to the classes visible in context of the call site
- A symbolic name as a string
- A MethodType that corresponds to the type that is expected by the call site (this is often useful for performing a final asType() combinator invocation to ensure that method handles match)

Other parameters can be passed as extra arguments, with the only constraint being that they must be constant values, which means they can be written to the *constant pool* of a class bytecode representation. This is the case for primitive types, strings, class references, and method handles.

KEY FOR LAMBDAS

Java 8 is poised to take advantage of invokedynamic as a way to support lambdas.

Sticking to the example in **Listing 8**, which obtained a method handle for the printAll method, the code in **Listing 9** would be a possible call site bootstrap method.

This method returns a nonmodifiable call site, where the prefix string is bound to ">>>" and the remaining arguments are collected as variable arguments. The resulting call site is of type (Object[])void. Note that because a string is a constant, we could also obtain the prefix as an extra parameter, and the value would depend on what is referenced from the call site in the bytecode. In this case, the bootstrap method would be as shown in **Listing 10**.

It is easy to test the bootstrap method to verify that it works as intended. Indeed, call sites provide a dynamicInvoker method that returns a method handle. Thus, you can use it as we've previously seen (see **Listing 11**).

Emitting Bytecode with invokedynamic Instructions

We are now ready to assemble the final pieces and actually put invokedynamic into practice. While the java.lang.invoke API can be

LISTING 9 LISTING 10 LISTING 11 LISTING 12

```
public static CallSite bootstrap(Lookup lookup, String name,
                MethodType type) throws Throwable {
  MethodHandle target = lookup.findStatic(
    Sample3.class, "printAll",
      methodType(void.class, String.class, Object[].class))
      .bindTo(">>> ")
      .asVarargsCollector(Object[].class)
      .asType(type);
  return new ConstantCallSite(target);
}
```

Download all listings in this issue as text

used as is, it remains to be seen how it can be leveraged directly from bytecode instructions.

As of Java 7, there is *no* language construct that translates into invokedynamic instructions at the bytecode level. This might change in Java 8 because some of the support for lambdas is likely to be implemented with invokedynamic.

In order to bootstrap a call site and leverage a method handle, we need to use a third-party library to write bytecode. The ASM library is our weapon of choice, because it helps in reading, writing, and transforming bytecode.

Another option for simpler cases is to use a tool such as Indify that can replace some marked call sites with an invokedynamic call, but leveraging ASM leaves no magic behind.

Let's consider the code in **Listing 12**.

The code generates a sample3.Caller class as a subclass of java.lang.Object. It has a main method that loads two strings and then calls a console:print function of type (Object, Object)void that is bootstrapped using the static method that we defined earlier. The symbolic name for the function can be anything, but it is common to normalize names and separate portions using a colon, as in property:get, property:set, and so on.

We can run this method, as shown in **Listing 13**, and see that the invokedynamic instruction is correctly bound at runtime. We can also check the generated bytecode with the javap decompiler that comes as part of the JDK, as shown in **Listing 14**.

BETTER SUPPORT

The introduction of invokedynamic to the JVM specification is recognition of the fact that dynamic languages bring value to the Java ecosystem and, as such, the JVM needs to provide better support for them.

Conclusion

This article introduced invoke dynamic, a new instruction backed with runtime API support for facilitating the implementation and execution of dynamic languages on top of the JVM. The java.lang.invoke API provides a rich set of operations to adapt a call site to a target that might be of a different signature type.

Existing dynamic languages for the JVM (for example, Groovy and JRuby) are starting to support invoke dynamic. In the long run, this should reduce the footprint of their runtime support code because maintaining Java 6 backward compatibility will become less of an issue. Fresh language implementations, such as Oracle's Nashorn (JavaScript), Rémi Forax' PHP.reboot, or the present author's yet-to-be-released Golo language, are based on invokedynamic from the get-go. Java 8 is poised to take advantage of invokedynamic as a way to efficiently implement the support of lambdas. Finally, derivative usages—such as the JooFlux research project, which provides dynamic code replacement and aspect-oriented programming—are starting to appear.

While invokedynamic is mostly useful to language and middleware implementers, let nothing restrain your creativity! Are you thinking of derivative usages? Are you thinking about creating yet another language? In any case, there is no harm in trying, and who knows what you might come up with.

Acknowledgements. The author would like to thank Marcus Lagergren for his very constructive feedback. `</article>`

LISTING 13 **LISTING 14**

```
$ java -classpath code/build/classes sample3.Caller
>>> World
>>> Hello
```

Download all listings in this issue as text

LEARN MORE

- "Bytecodes Meet Combinators: invokedynamic on the JVM"
- "JooFlux: Hijacking Java 7 InvokeDynamic to Support Live Code Modifications"
- "JSR 292 Cookbook"
- "Dynalink: Dynamic Linker Framework for Languages on the JVM"