JVM INTERNALS

Understanding Java method invocation with invokedynamic

The invokedynamic instruction added in Java 7 makes it possible to resolve method calls dynamically at runtime.

by Ben Evans

March 12, 2021

[*Java Magazine* is pleased to republish this article from Ben Evans, published in 2017, about Java Virtual Machine internals. —*Ed*.]

In a previous article, "Mastering the mechanics of Java method invocation," I discussed four of Java's five method-invocation opcodes. These four are the bytecode representations of the standard forms of method invocation used in Java 8 and Java 9, and they are `invokevirtual`, `invokespecial`, `invokeinterface`, and `invokestatic`.

This raises the question of how the fifth opcode, `invokedynamic`, enters the picture. The short answer is that, as of Java 9, there was no direct support for `invokedynamic` in the Java language.

In fact, when `invokedynamic` was added to the runtime in Java 7, the `javac` compiler would not emit the new bytecode under any circumstances whatsoever.

As of Java 8, `invokedynamic` is used as a primary implementation mechanism to provide advanced platform features. One of the clearest and simplest examples of this use of the opcode is in the implementation of lambda expressions. To follow along with the rest of this article, you'll need to have some familiarity with how the JVM invokes methods, or you'll need to read the first article in this series.

Lambdas are object references

Before diving into how `invokedynamic` is used to enable lambdas, a brief reminder of what lambdas actually are is in order. Java has only two types of values: primitive types (such as `char` and `int`) and object references. Lambdas are obviously not primitive types, so they must be object references. Consider the following lambda:

```
public class LambdaExample {
   private static final String HELLO = "Hello
   public static void main(String[] args) thr
      Runnable r = () -> System.out.println(H
      Thread t = new Thread(r);
      t.start();
      t.join();
   }
}
```

The lambda expression is assigned to a variable of type `Runnable`. This means that the lambda evaluates to a reference to an object that has a type that is compatible with `Runnable`.

Essentially, this object's type will be some subclass of `Object` that has defined one extra method (and has no fields). The extra method is understood to be the `run()` method expected by the `Runnable` interface.

Before Java 8, such an object was represented only by an instance of a concrete anonymous class that implemented `Runnable`. In fact, in the initial prototypes of Java 8 lambdas, inner classes were used as the implementation technology.

The long-range future roadmap for the JVM could contain future versions where more sophisticated representations of lambdas could be possible. Fixing the representation to use explicit inner classes would prevent a different representation from being used by a future version of the platform. This is undesirable and so, instead, Java 8 and Java 9 use a more sophisticated technique than hardcoding inner classes. The bytecode for the previous lambda example is as follows:

```
public static void main(java.lang.String[]) t
   Code:
      0: invokedynamic #2, 0 // InvokeDynamic

      5: astore_1
      6: new #3 // class java/lang/Thread
      9: dup
      10: aload_1
      11: invokespecial #4 // Method java/lan
                                         // (L
      14: astore_2
      15: aload_2
      16: invokevirtual #5 // Method java/lan
      19: aload_2
      20: invokevirtual #6 // Method java/lan
      23: return
```

The bytecode at offset 0 indicates that some method is being called via `invokedynamic`, and the return value of that call is placed upon the stack. The rest of the bytecode in the method is a straightforward representation of the rest of the method.

How invokedynamic operates

At this point, I'll discuss some of the details of the nature of `invokedynamic` and how the opcode operates. When a class containing an `invokedynamic` instruction is loaded by the class loader, the target of the method invocation is not known ahead of time. This design differs from all other types of call sites in JVM bytecode.

For example, in the case of `invokestatic` and `invokespecial` sites, which I discussed in the previous article, the exact implementation method (referred to as the *call target*) is known at compile time. In the case of `invokevirtual` and `invokeinterface`, the call target is determined at runtime. However, the target selection is subject to the constraints of the Java language inheritance rules and type system. As a result, at least *some* call target information is known at compile time.

In contrast, `invokedynamic` is far more flexible about which method will actually be called when the opcode is dispatched. To allow for this flexibility, `invokedynamic` opcodes refer to a special attribute in the constant pool of the class that contains the dynamic invocation. This attribute contains additional information to support the dynamic nature of the call, called *bootstrap methods* (BSMs).

BSMs are a key part of `invokedynamic`, and every `invokedynamic` call site has a constant pool entry for a corresponding BSM. To allow the association of a BSM to a specific `invokedynamic` call site, a new entry type, also called `InvokeDynamic`, has been added to the class file format as of Java 7.

The call site of the `invokedynamic` instruction is said to be *unlaced* at class loading time. The BSM is called to determine what method should actually be called, and the resulting `CallSite` object will then be *laced* into the call site.

In the simplest case, that of a `ConstantCallSite`, as soon as the lookup has been done once, it will not need to be repeated. Instead, the target of the call site will be directly called on all future invocations without any further work. This means that the call site is now stable and is, therefore, friendly to other JVM subsystems, such as the just-in-time (JIT) compiler.

For this mechanism to work efficiently, the JDK must contain suitable types to represent the call site, the BSMs, and other parts of the implementation. Java's original core reflection types are capable of representing methods and types. However, the API dates from the very early days of the Java platform and has several aspects that make it a less-than-ideal choice.

For example, reflection predates both collections and generics. As a result, method signatures are represented by `Class[]` in the Reflection API. This can be cumbersome and error-prone, and it is hampered by the verbose nature of Java's array syntax. It is further complicated by the need to manually box and unbox primitive types and to work around the possibility of void methods.

Method handles to the rescue

Instead of forcing the programmer to deal with these issues, Java 7 introduced a new API, called `MethodHandles`, to represent the necessary abstractions. The core of this API is the package `java.lang.invoke` and especially the class `MethodHandle`. Instances of this type provide the ability to call a method, and they are directly executable. They are dynamically typed according to their parameter and return types, which provides as much type safety as possible, given the dynamic way in which they are used. The API is needed for `invokedynamic`, but it can also be used alone, in which case it can be considered a modern, safe alternative to reflection.

To get a handle for a method, the method must be looked up via a *lookup context*. The usual way to get a context is to call the static helper method `MethodHandles.lookup()`. This method returns a lookup context based on the currently executing method. From this context, you can obtain method handles by calling one of the `find*()` methods, such as `findVirtual()` or `findConstructor()`.

One important difference between method handles and reflection is that lookup contexts return only methods that were accessible from the scope where the lookup object was created. There is no way to subvert this; there is no equivalent of the `setAccessible()` back door that is present in reflection. This means that method handles are safe to use under *all* circumstances, including using them with a security manager.

However, care must be taken, as the access control check has been moved to method-lookup time. This means that a lookup context can hand out references to private methods that were visible to the lookup but are not necessarily visible at the time when the method handle is invoked.

To solve the problems of representing method signatures, the `MethodHandles` API also includes the `MethodType` class. This is a simple immutable type that has some very useful properties and does the following:

- Represents the type signature of a method
- Consists of the return type followed by the argument types

- Does not include the "receiver type" or name of the method
- Is designed to remove the `Class[]` problem from core reflection

In addition, instances of it are immutable.

With this API, signatures of methods are represented as instances of `MethodType`, and there is no need to create a new type to model each possible signature. New instances are created from a simple factory method, for example:

```
// toString()
MethodType mtToString =
   MethodType.methodType(String.class);

// A setter method
MethodType mtSetter =
   MethodType.methodType(void.class, Object.c

// compare() from Comparator<String>
MethodType mtStringComparator =
   MethodType.methodType(int.class, String.cl
```

Once you have created a signature object, it can be used (along with a method name) to look up a method handle, as in the following example to get a method handle on `toString()`:

```
public MethodHandle getToStringHandle() {
      MethodHandle mh = null;
      MethodType mt = MethodType.methodType(S
      MethodHandles.Lookup lk = MethodHandles
      try {
             mh = lk.findVirtual(getClass(),
      } catch (NoSuchMethodException | Illega
            throw new AssertionError().initCau
      }
      return mh;
}
```

The handle can then be invoked in a similar way to a reflective call. A receiver object must be supplied for instance methods, and the invocation code must deal with the possibility of a coarse-grained exception.

```
MethodHandle mh = getToStringMH();
try {
      mh.invoke(this, null);
} catch (Throwable e) {
      e.printStackTrace();
}
```

The concept of a BSM should now be clear: When program control reaches an `invokedynamic` call site for the first time, the associated BSM is called. The BSM returns a call site object containing a method handle to the method that will actually be bound into the call site. For this mechanism to function correctly with static typing, the BSM must return a handle to a method of the correct method signature.

To get back to the lambda expression example I gave earlier, you can think of the `invokedynamic` opcode as representing a call to some sort of platform factory method for a lambda expression. The actual body of the lambda has been transformed into a private static method on the class where the lambda is defined.

```
private static void lambda$main$0();
   Code:
      0: getstatic #7           // Field
                                       // jav
      3: ldc #9                   // String H
      5: invokevirtual #10 // Method
                                       // jav
                                       // (Lj
      8: return
```

The lambda factory will return an instance of some type that implements `Runnable`, and the `run()` method of that type will call back to this private method when the lambda is executed.

Using `javap -v` to look inside the constant pool shows the following entry:

```
#2 = InvokeDynamic #0:#40 //
#0:run:()Ljava/lang/Runnable;
```

Looking at the BSM section of the class file shows the factory that is being called.

```
BootstrapMethods:
0: #37 REF_invokeStatic
java/lang/invoke/LambdaMetafactory.metafactor
andles$Lookup;Ljava/lang/String;Ljava/lang/in
nvoke/MethodType;Ljava/lang/invoke/MethodHand
ype;)Ljava/lang/invoke/CallSite;
Method arguments:
#38 ()V
#39 REF_invokeStatic optjava/LambdaExample.la
#38 ()V
```

This output refers to a static factory method, called `metafactory()`, on the `LambdaMetafactory` implementation class in `java.lang.invoke`. This is a BSM that will create the linkage bytecode at runtime if the lambda is ever created. The metafactory code takes in a lookup object and the method types to ensure static type safety, along with a method handle pointing at the private static method containing the lambda method body.

It returns a call site that will lace in the lambda body if it is ever called.

```
public static CallSite metafactory(
            MethodHandles.Lookup caller,
            String invokedName,
            MethodType invokedType,
            MethodType samMethodType,
            MethodHandle implMethod,
            MethodType instantiatedMethodType
                        throws LambdaConversi
            AbstractValidatingLambdaMetafacto
            mf = new InnerClassLambdaMetafact
                        caller, invokedType,
                        invokedName, samMetho
                        implMethod, instantia
                        false, EMPTY_CLASS_AR
            mf.validateMetafactoryArgs();
            return mf.buildCallSite();
}
```

The current implementation uses a private metafactory that will still create an inner class per lambda, but the classes are dynamically created and are never written to disk. This means that the implementation mechanism could change with a future release of Java, and any existing lambdas will be able to take advantage of the new mechanism.

In Java 8 and Java 9, the implementation based on the `InnerClassLambdaMetafactory` class makes use of a slightly modified version of the ASM bytecode manipulation library that ships in the package `jdk.internal.org.objectweb.asm`.

This implementation creates dynamic classes to represent the implementing type of a lambda, while at the same time future-proofing the implementation and maintaining JIT-friendliness.

It makes use of the simplest case—call sites that are looked up once and cannot change thereafter. These are represented by instances of `ConstantCallSite`, which I discussed earlier. More complex cases are possible, including call sites that can change or even have semantics similar to volatile variables. These cases are harder to handle and quickly become very complex, but they provide the greatest amount of dynamic flexibility available to the platform.

The previous example of lambda expressions shows how the `invokedynamic` opcode relaxes a key part of the static type system and makes flexible runtime dispatch possible.

Conclusion

While `invokedynamic` might not be a part of Java that most developers are exposed to very often, the Java ecosystem has evolved significantly through its addition. Future versions of Java may well introduce further advances in JVM technology, and many of these techniques would be impossible without the advent of `invokedynamic` and the reimagining of method execution that it represents.

Dig deeper

- Mastering the mechanics of Java method invocation
- Behind the scenes: How do lambda expressions really work in Java?
- Package java.lang.invoke

Ben Evans

Ben Evans (@kittylyst) is a Java Champion and Principal Engineer at New Relic. He has written five books on programming, including *Optimizing Java* (O'Reilly). Previously he was a founder of jClarity (acquired by Microsoft) and a member of the Java SE/EE Executive Committee.