Deconstructing MethodHandles

May 2014
@PaulSandoz (markdown source)

The document deconstructs the current implementation of `MethodHandle` in OpenJDK.

Note: the Java documentation of `MethodHandle` provides a more detailed and thorough description of interoperable `MethodHandle` behaviour.

MethodHandles are cunning:

> *Blackadder: I have come up with a plan so cunning you could stick a tail on it and call it a weasel.*

The general philosophy is to leverage a few key intrinsic mechanisms of HotSpot, perform most of the heavy lifting in Java code, and let the runtime compiler "have-at-it" and inline that code.

Deconstruction

This section deconstructs how exact invocation of MethodHandles are compiled, linked and executed using a specific example of setting the value of a non-static field. Examples of byte code and inlining traces are obtained by compiling and executing a `MethodHandle`-based `jmh` micro-benchmark.

Despite the name a `MethodHandle` can reference an underlying static or instance field of class. In the OpenJDK implementation invocations of such handles result in a corresponding call to a method on `sun.misc.Unsafe`, after appropriate safety checks have been performed. For example, if the field is a reference type (a non-primitive type) marked as volatile then the method `Unsafe.putObjectVolatile` will be invoked.

If such a reference to a `MethodHandle` is held in a static final field then the runtime should be able to constant fold invocations on that reference and what it holds when inlining occurs. In such cases, perhaps surprisingly, the generated machine code can be competitive with direct invocation of methods on `Unsafe` or `get/putfield` byte codes instructions.

Note: It is straightforward to extend the `MethodHandle` implementation to support handles for relaxed, lazy and compare-and-set atomic operations by invoking the appropriate method on `Unsafe`, `putObject`, `putOrderedObject` and `compareAndSwapObject` respectively.

A `MethodHandle` giving write access to a non-static field, "vfield" say of type `Value`, on a class, `Receiver` say, can be obtained as follows:

```
MethodHandles.Lookup lookup = ...
MethodHandler setterOfValueOnReceiver =
    lookup.findSetter(Receiver.class, "vfield", Value.class);
```

An exact invocation of that `MethodHandle` will then set the value of the field "vfield" on an instance of `Receiver`:

```
Receiver r = ...
Value v = ...
setterOfValueOnReceiver.invokeExact(r, v);
```

Note that the receiver instance is passed as the first parameter to the `invokeExact` method. The receiver provides the base location from which to access the value of field, "vfield", it holds (since there is no direct l-value and pass by reference of fields, or array elements, supported in Java a pair of receiver and value is necessary).

The method `MethodHandle.invokeExact` is declared as a polymorphic signature method:

```
public final native @PolymorphicSignature Object invokeExact(Object... args) throws Throwable;
```

When javac compiles invocations to signature polymorphic methods it uses a symbolic type descriptor as the method signature, which is derived from actual parameter and return types of the caller and not the method signature of the method declaration.

An example of an invokevirtual instruction is shown as follows:

```
13: invokevirtual #12    // Method java/lang/invoke/MethodHandle.invokeExact
                            :(Lvarmh/VolatileSetAndGetTest$Receiver;Lvarmh/VolatileSetAndGetTest$Value;)V
```

Notice that the method signature accepts two arguments, a instance of a class `Receiver` and class `Value`, and returns void, this signature is referred to as the symbolic type descriptor.

An inlining trace of such an invocation, when the handle is constant folded, is as follows:

```
@ 13   java.lang.invoke.LambdaForm$MH/363164801::invokeExact_MT (15 bytes)   inline (hot)
  @ 2   java.lang.invoke.Invokers::checkExactType (30 bytes)   inline (hot)
    @ 11   java.lang.invoke.MethodHandle::type (5 bytes)   accessor
  @ 11   java.lang.invoke.LambdaForm$MH/717088119::putObjectVolatileFieldCast (32 bytes)   inline (hot)
    @ 1   java.lang.invoke.DirectMethodHandle::fieldOffset (9 bytes)   inline (hot)
    @ 6   java.lang.invoke.DirectMethodHandle::checkBase (7 bytes)   inline (hot)
      @ 1   java.lang.Object::getClass (0 bytes)   (intrinsic)
    @ 13   java.lang.invoke.DirectMethodHandle::checkCast (9 bytes)   inline (hot)
      @ 5   java.lang.invoke.DirectMethodHandle$Accessor::checkCast (9 bytes)   inline (hot)
        @ 5   java.lang.Class::cast (27 bytes)   inline (hot)
          @ 6   java.lang.Class::isInstance (0 bytes)   (intrinsic)
    @ 28   sun.misc.Unsafe::putObjectVolatile (0 bytes)   (intrinsic)
```

The invocation is comprised of two stages. The first stage performs an efficient run-time safety check to determine if the symbolic type descriptor encoded at the call site exactly matches the method type descriptor of the `MethodHandle`. The second stage performs the write access, in this case to a volatile field.

The `MethodHandler.invokeExact` invocation is intrinsically linked (see section "Linking of `invokeExact` invocations") to the static method `invokeExact_MT`, on a dynamically generated class, the byte code of which is:

```
static void invokeExact_MT(java.lang.Object, java.lang.Object, java.lang.Object, java.lang.Object);
  descriptor: (Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/Object;)V
  flags: ACC_STATIC
  Code:
    stack=3, locals=4, args_size=4
       0: aload_0
       1: aload_3
       2: invokestatic  #16                 // Method java/lang/invoke/Invokers.checkExactType:(Ljava/lang
/Object;Ljava/lang/Object;)V
       5: aload_0
       6: checkcast     #18                 // class java/lang/invoke/MethodHandle
       9: aload_1
      10: aload_2
      11: invokevirtual #21                 // Method java/lang/invoke/MethodHandle.invokeBasic:(Ljava/lang
/Object;Ljava/lang/Object;)V
      14: return
```

The parameters for `invokeExact_MT` are as follows:

- the `MethodHandle` instance, `setterOfValueOnReceiver`;
- the parameters `(r, v)` passed to the `invokeExact` method; and finally
- the call site's symbolic type descriptor, appended when the call site is linked.

Notice that at this point the reference parameter types are erased to `Object`, thus the class declaring this static method can be shared for invocations with different method type descriptors that erase to the same signature.

This method first performs the method type descriptor check and if that fails an exception is thrown, otherwise it is safe to proceed as it is known the parameter types and return type of the call site are correct and exactly match. Next, the `invokeBasic` on the `MethodHandle` instance is invoked with the same parameters passed to the `invokeExact` method.

The invocation of `invokeBasic` is intrinsically linked to the static method `putObjectVolatileFieldCast` on a dynamically generated class corresponding to the compiled lambda form of the `MethodHandle`. The `vmentry` field of the `LambdaForm` of the `MethodHandle` is the `MemberName` that characterizes the method `putObjectVolatileFieldCast`, the byte code of which is:

```
static void putObjectVolatileFieldCast(java.lang.Object, java.lang.Object, java.lang.Object);
  descriptor: (Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/Object;)V
  flags: ACC_STATIC
  Code:
    stack=5, locals=7, args_size=3
       0: aload_0
       1: invokestatic  #16                 // Method java/lang/invoke/DirectMethodHandle.fieldOffset:(Ljava
/lang/Object;)J
       4: lstore_3
       5: aload_1
       6: invokestatic  #20                 // Method java/lang/invoke/DirectMethodHandle.checkBase:(Ljava/lang
/Object;)Ljava/lang/Object;
       9: astore        5
      11: aload_0
      12: aload_2
      13: invokestatic  #24                 // Method java/lang/invoke/DirectMethodHandle.checkCast:(Ljava/lang
/Object;Ljava/lang/Object;)Ljava/lang/Object;
      16: astore        6
      18: ldc           #26                 // String CONSTANT_PLACEHOLDER_0 <<sun.misc.Unsafe@77669660>>
      20: checkcast     #28                 // class sun/misc/Unsafe
      23: aload         5
      25: lload_3
      26: aload         6
      28: invokevirtual #32                 // Method sun/misc/Unsafe.putObjectVolatile:(Ljava/lang/Object;
JLjava/lang/Object;)V
      31: return
```

The first parameter is the `MethodHandle` instance and the subsequent parameters are those, `(r, v)`, passed to the `invokeExact` method. The `MethodHandle` instance is a direct handle that holds the field offset to be used with the invocation of `Unsafe.putObjectVolatile` at the end of this method. Before that invocation:

- a safety check is performed to ensure the receiver instance is not `null`; and
- a cast check of the value instance to an instance of the value (field) type is performed to ensure the runtime compiler has sufficient information to perform type profiling. Note that this is not required for type safety since such a safety check was already performed by the `invokeExact_MT` method; observe that the type of the receiver instance does not require a cast to an instance of the receiver type.

Linking of invokeExact invocations

Invocations of `MethodHandler.invokeExact` are intrinsically linked via an up call from the VM to a Java method that returns a `MemberName` characterizing the Java method to be linked to. This up-called Java method, statically known to the VM, is `MethodHandleNatives.linkMethod`:

```
static MemberName linkMethod(Class<?> callerClass, int refKind,
                             Class<?> defc, String name, Object type,
                             Object[] appendixResult)
```

The "type" parameter is either an instance of `MethodType` or `String` corresponding to the symbolic type descriptor.

The “appendixResult” is used to return an optional extra parameter that must match the last parameter of the method characterized by the `MemberName`, and will be permanently appended at the linked call site.

On invocation of `invokeExact` the VM stack already contains three values and the VM will add one additional parameter onto the call stack such that the parameters are as follows:

- the `MethodHandle` instance, `setterOfValueOnReceiver`;
- the parameters `(r, v)`; and finally the additional parameter that is
- the first element of the "appendixResult", which is the call site's symbolic type descriptor.

Acknowledgements

Thanks to John Rose for providing useful comments and feedback.