Method handle invocation

Method handle invocation

Method handles have several modes of invocation, involving various degrees of type checking and conversion.

Every method handle invocation takes into account two independent sources of type information. The caller (whose call site is invoking the method handle) specifies the *caller method type* implicitly by the choice of the call descriptor in the call site bytecode. The callee (i.e., the method handle being invoked) has its own *callee method type*, which is simply its `MethodHandle.type` property.

The possible modes of method handle invocation include:

- *Exact invocation:* The `invokeExact` method requires the caller and callee method types to have identical components. This method is public, safe, and signature-polymorphic.
- *Generic invocation:* The generic `invoke` method allows the caller and callee method types to differ, as long as the callee can be viewed under the caller type, using the `asType` method. As a consequence, this mode offers on-the-fly argument and return conversions. This method is public, safe, and signature-polymorphic.
- *Varargs invocation:* The overloaded `invokeWithArguments` method allow an arbitrary varargs array or list of untyped arguments to be applied to a callee, and the call succeeds as long as the callee can be viewed under an implied caller type determined as if by `MethodType.genericType`. This method is public.
- *Basic invocation:* The `invokeBasic` method requires that the caller and callee method types have corresponding components, but uses "basic type" matching, which allows any reference to match any other reference. Because there is no runtime check for matching types, this method must be used carefully. It is used by trusted code to implement the other invocation modes. This method is non-public and signature-polymorphic. The JVM implementation treats it very specially.
- *Invokedynamic invocation:* The `invokedynamic` instruction implicitly performs an exact invocation on the target of its resolved call site. (Unlike the other modes, the callee is not visible as a receiver on the JVM expression stack.) Because the system ensures that all call site targets have pre-checked callee method types, the system may use basic invocation as an implementation option.

In addition, method handles can be viewed by various transforms that change their apparent type, and hence their acceptable invocation types and modes:

- *Retyping view:* A method handle can be viewed as a range of other compatible callee method types. The view methods, `asType` and `explicitCastArguments`, are public. Allowable conversions are drawn from those natural either to the Java language (on erased types only) or to the JVM itself. The `explicitCastArguments` view allows some conversions not permitted by the `asType` view, such as narrowing primitive casts.
- *Spread view:* A method handle can be viewed as if it accepts a varargs-like array of trailing arguments. The view method, `asSpreader`, is public. The array can be of any element type.
- *Collecting view:* A method handle which accepts a trailing array argument can be viewed as if it takes those arguments positionally. Either a fixed or variable number of positional arguments are acceptable. The view methods, `asCollector` and `asVarargsCollector`, are public. The array can be of any element type.
- Other transforms, such as for argument permutation or binding, may also be considered as method handle views.

Additional implementation details of invocation paths are discussed on Method handles and invokedynamic.

Exact invocation

Currently the exact type check is a simple pointer comparison. This works because instances of `MethodType` are interned as an implementation choice, although the spec. does not require this and it could change.

In the implementation, this check is performed by the adapter method described below.

Generic invocation

Generic invoke is defined as the composition of `asType` and `invokeExact`. A number of identities are available which allow optimization.

Here is a "whiteboard dump":

```
// mt0 STATIC methodType(R, TYPEOF(a)*);   Appendix in CPCE
// cs0 STATIC new GenericCallSite(mt0)     Extra state for call site cache if any
R r = (R ) mh.invoke[Generic](a*);
lambda(mh, a*, appendix=mt0) {    Method* in CPCE, which is a Lambda Form
 mt1 = mh.type;
 if (LATER) {
   if (mt1 == mt0)
      (R ) mh.invokeBasic(a*)
   if (safeBasicTypeConv(mt1 -> mt0))
      (R ) mh.invokeBasic(a*)
   if (safeBasicTypeConv(mt1 -> mt0[R := Object]))
      (Object) mh.invokeBasic(a*) & checkCast(R )   checkcast(A)* also?
   if ?
      bigAdapter(mt1,mt0,cs0).invokeBasic(mt0,mh,a*)   cache $[2]
 }
 else  (R ) mh.asType(mt0).invokeBasic(a*)   cache $[1]
}
```

Adding a cache ($1) for `asType` allows a fast path assuming the same MH gets generically invoked many times with the same caller method type. Other simple optimizations involve recognizing exact or near-exact matches on the caller and callee method types.

Adding a cache to either method type (`mt0` or `mt1`) may allow a "big adapter" to be cached which can take into account details of the types. Call-site caching may also be useful, in case different method handles with common behavior are invoked repeatedly from the same site.

(Optimizations may be tuned based on experiments with branch frequencies, detection of equivalent MHs (especially DMHs), or cache miss data. Interned vs. copied DMHs could have the usual opposing effects on footprint and cache specificity.)

Varargs invocation

discuss

Basic invocation

discuss

Implementing method handle invocation

Internally to HotSpot (in `rewriter.cpp`) method handle invocations of the exact and generic modes are rewritten to use a special instruction called `invokehandle`. (This instruction in many ways is parallel to `invokedynamic`.) It resolves to an adapter method pointer and an appendix, both of which are recorded as results of the resolution. The appendix (if not null) is always pushed after the explicit arguments to `invoke` or `invokeExact`, and is therefore available for use by the adapter method (which may therefore take an extra trailing argument).

The resolution is done via a call to trusted Java code, to a method called `MethodHandleNatives.linkMethod`. As with `linkCallSite`, the JVM passes all resolved constant pool references to `linkMethod`, and receives back a coordinated pair of values, a `MemberName` and an `Object`. After unpacking, these are plugged into the CPCE as the adapter and appendix.

The same degrees of freedom apply to `invokehandle` CPCE entries as apply to `invokedynamic` CPCE entries, and similar optimization opportunities apply.

There is one major difference from `invokedynamic`: Many `invokehandle` instructions can share a single CPCE entry, if they all have the same signature and method name ("invokeExact" vs. "invoke").

Adapter method for `invokeExact`

The standard semantics of an `invokevirtual` of `MethodHandle.invokeExact` are simple. The signature for the call site (which may contain any mix of any references and primitive types) is resolved (once, at link time) into a `MethodType`. Every time the method handle is invoked, the method type is checked against the type of the method handle being invoked. (Since the method handle is a computed value, it can of course be different every time, and the type does not necessarily match.) If the two types differ in any way, a `WrongMethodTypeException` is thrown. Otherwise, the method handle is invoked on the given types, and returns a value of the type expected by the caller.

An adapter method for `invokeExact` is correspondingly simple. It merely performs a method type check and then calls `invokeBasic`. The appendix is a reference to the resolved `MethodType`, required to make the type check.

Here is an example:

```
LambdaForm(a0:L,a1:L,a2:L)=>{
    t3:V=Invokers.checkExactType(a0:L,a2:L);
    t4:L=MethodHandle.invokeBasic(a0:L,a1:L);
    t4:L}
```

The leading argument `a0` is a method handle. The trailing argument `a2` is the method type appendix, computed when the call site is resolved. The middle argument `a1` is the sole argument to the method handle.

First the subroutine `Invokers.checkExactType` is called on the method handle and the appendix, extracting the type from the method handle and comparing it with the static call site type. If no exception is thrown, control returns to the adapter method. No value is returned, and the name `t3` has a pseudo-type of void.

(Since the appendix represents the static call site type, and the type of the method handle is the dynamically acceptable type, this could be viewed as a simple dynamic type check.)

Next, `invokeBasic` is used to jump into the method handle (which is now known to be completely safe for this call). The result comes back, briefly named `t4`, and is returned to the caller.

Adapter method for generic `invoke`

Method handles also support a more complex invocation mode, which can perform type conversions on individual arguments and return values, and even group arguments into varargs arrays.

As with `invokeExact` the resolution of such a call site is carried out by a call to `MethodHandleNatives.linkMethod`. In this case, the trusted Java code must return a more flexible and complex adapter method.

Here is an example:

```
LambdaForm(a0:L,a1:L,a2:I,a3:L)=>{
    t4:L=Invokers.checkGenericType(a0:L,a3:L);
    t5:I=MethodHandle.invokeBasic(t4:L,a3:L,a0:L,a1:L,a2:I);
    t5:I}
```

As before, `a0` is the method handle and the trailing `a3` is a method type. Here, there are two regular arguments, a reference and an int.

As before, the first job is to check the type, and this is done by `Invokers.checkGenericType`. Unlike the simpler check, this routine returns a value. (The routine can also throw `WrongMethodTypeException` if necessary.)

The value returned from `checkGenericType` is in fact a method handle `t4`. This method handle is immediately invoked on the original arguments, *plus* th e original method handle `a0`, *plus* the desired call site type `a3` (not in that order).

Depending on the match or mismatch between the type of `a0` and the call site type `a3`, the actual behavior of `t4` could be very simple (basically another `in vokeBasic`) or very complex (an arity change with type conversions. That's up to the runtime.