Method handles and invokedynamic

JSR 292 introduces a flexible `invokedynamic` instruction which is bound to a user-defined graph of method handles.

Sub-topics

- Bound method handles — A bound method handle is a small struct containing one or more saved argument values, plus a target method handle which will eventually receive those arguments.
- Direct method handles — A direct method handle represents a particular named method, field, or constructor, with no transformations.
- Method handle invocation — Method handles have several modes of invocation, involving various degrees of type checking and conversion.
    - Deconstructing MethodHandles

Method handle fundamentals

Abstractly, a method handle is simply a type and some behavior that conforms to that type. As befits an object-oriented system, the behavior may include data.

Concretely, a method handle can refer to any JVM method, field, or constructor, or else it can be a transform of any previously specified method handle. Transforms include partial application (binding), filtering, and various forms of argument shuffling.

The method handle's type is expressed as a sequence of zero or more parameter types, and an optional return type (or the non-type void). Concretely, this is a `MethodType` reference, and can be extracted from any method handle using `MethodHandle.type`.

The behavior is what happens when the method handle is invoked, using the method `MethodHandle.invokeExact`. The special capability of method handles is that `invokeExact` accepts any number of any type of arguments, and can return any type or void. A regular `invokevirtual` instruction performs this. (It is rewritten secretly to `invokehandle`, as discussed below, but this can be ignored except by HotSpot implementors.)

Uniquely to method handles, the `invokevirtual` instruction can specify any structurally valid type signature, and the call site will link. Technically, we say that `invokeExact` is *signature polymorphic*. Practically speaking, when linking such a call site, the JVM must be ready to deal with any type signature, which means it will have to generate adapters of various sorts. From the user's point of view, a method handle is a magic thing which can wrap and/or invoke any method, of any type.

Concretely, the behavior of a method handle depends on a object called a `LambdaForm`, which is a low-level description of step-by-step operations. A method handle's lambda form is stored in its `form` field, just as its type is stored in its `type` field.

A method handle's lambda form may ignore the method handle completely and do something context-independent, like throw an exception or return zero. More generally, it can consult the method handle for information. For example, it can examine the method handle's return type and convert some value to that type before returning it.

More interestingly, if a method handle's class is a subclass which contains additional data fields, the lambda form can refer to those fields as it executes.

Since method handles express behavior more than state, their fields are typically immutable. But, method handles can easily be *bound* to arbitrary Java objects, producing closures.

The "basic type" system

In order to implement signature polymorphism more simply, method handles internally operate in terms of *basic types*. A basic type is a JVM type in which many inconvenient distinctions have been "erased", so that the remaining distinctions (such as reference vs. primitive and int vs. long) can be attended to.

For starters, in the basic type system, all 32-bit types except `float` are erased to simple `int`. If a byte value is required somewhere, it must be masked down from a full int. Thus, there are only four primitive types to worry about.

Under basic typing rules, all reference types are represented by `java.lang.Object`. Thus, there are a total of five basic types, represented by their JVM signature characters: L, I, J, F, D. To these we add V for the non-type void.

In the bulk of Java code, the full type system is in force. In order to name reference types, a system of class loaders and type constraints must be consulted and honored. From perspective of the JSR 292 runtime, this type system is a complex mix of names and scopes. Inside the runtime, using basic types there are no names to worry about, except `Object` and other types on the boot class path.

If a reference of a narrower type is required somewhere, an explicit checkcast must be issued before the reference is used. In fact, the checkcast is in general a call to `Class.cast`, with the specialized type being a constant `Class` reference rather than a symbolic reference name.

Normally, all extra conversions (such as int to byte and `Object` to a named reference type) disappear in the optimizer, which keeps track of full type information from context.

Lambda form basics

In brief, a lambda form is a classic lambda expression with zero or more formal parameters, plus zero or more body expressions. The types of parameters and expression values are drawn from the basic type system.

Each expression is simply the application of a method handle to zero or more arguments. Each argument is either a constant value or a previously specified parameter or expression value.

When a lambda form is used as a method handle behavior, the first parameter (`a0`) is always the method handle itself. (But there are other uses for lambda forms.)

When a method handle is invoked, after any initial type checking, the JVM executes the lambda form of the method handle to complete the method handle invocation. This leads to some bootstrapping challenges, since the lambda form executes by evaluating additional method handle invocations.

Lambda forms are described in detail elsewhere:http://wiki.jvmlangsummit.com/Lambda_Forms:_IR_for_Method_Handles

Lambda forms will be introduced by example as various behaviors are described.

Lambda form optimization

There is one more indirection in lambda form execution which allows the system to optimize itself: A lambda form has a field called `vmentry` which (at long last) provides a `Method*` pointer for the JVM to jump into, in order to evaluate the lambda form.

(Note: Since Java cannot directly represent JVM metadata pointers, this `vmentry` is actually of type `MemberName`, which is a low-level wrapper for a `Method*`. So there is one more indirection after all, to hide the metadata.)

When a lambda form is first created, this `vmentry` pointer is initialized to a method called the lambda form interpreter, which can execute any lambda form. (Actually it has a thin wrapper which is specialized to the arity and basic types of the arguments.) The lambda form interpreter is very simple and slow. After it executes a given lambda form a few dozen times, the interpreter fetches or generates bytecode for the lambda form, which is customized (at least partially) to the lambda form body. In the steady state, all "hot" method handles and their "hot" lambda forms have bytecode generated, and eventually JIT-compiled.

Thus, in the steady state, a hot method handle is executed without the lambda form interpreter. The low-level JVM steps are as follows:

- Fetch `MethodHandle.form`.
- Fetch `LambdaForm.vmentry`.
- Fetch `MemberName.vmtarget`, a hidden `Method*` pointer.
- Fetch `Method::from_compiled_entry`.
- Jump to optimized code.

As noted elsewhere, if the method handle (or if the lambda form or the member name) is a compile-time constant, all the usual inlining can be done.

Invokedynamic

As defined in the JVMS, `invokedynamic` consists of a name, a method type signature, and bootstrap specifier.

The caller-visible behavior of the instruction is defined only by the type signatures, which determines exactly which types of arguments and return values are shuffled through the stack.

The actual behavior of the instruction is determined when the instruction is first executed. As with the other invoke instructions, the `LinkResolver` modules handles setup operations performed on first execution.

For `invokedynamic`, the bootstrap specifier is resolved into a method handle and zero or more extra constant arguments. (These are all drawn from the constant pool.) The name and signature are pushed on the stack, along with the extra arguments and a `MethodHandles.Lookup` parameter to reify the requesting class, and the bootstrap method handle is invoked.

(This appeal to a user-specified method may seem startling, but to the JVM is it not much more complex than the `ClassLoader` operations which must be performed to locate a new class's bytecodes.)

When the bootstrap method returns, it presents a `CallSite` object to the JVM runtime. This call site contains a method handle, which (in the end) determines the exact behavior of the linked `invokedynamic` instruction. Since method handle can do just about anything, the `invokedynamic` instruction, after linking, is a fully general virtual machine instruction.

(The alert reader will wonder why the bootstrap method doesn't just return a method handle. The answer is that some call sites can, potentially, be bound over time to a succession of *different* method handles. This gives Java programmers the a low-level code-patching technique similar to that used by the JVM to manage monomorphic and polymorphic virtual call sites.)

Invokedynamic implementation

Because each invokedynamic instruction links (in general) to a different call site, the constant pool cache must contain a separate entry for each invokedynamic instruction. (Other invoke instructions can share CP cache entries, if they use the same symbolic reference in the constant pool.)

A CP cache entry ("CPCE"), when resolved, has one or two words of metadata and/or offset information.

For invokedynamic, a resolved CPCE contains a `Method*` pointer to a concrete *adapter* method providing the exact behavior of the call. There is also a reference parameter associated with the call site called the *appendix*, which is stored in the `resolved_references` array for the CPCE.

The method is called an adapter because (generally speaking) it shuffles arguments, extracts a target method handle from the call site, and invokes the method handle.

The extra reference parameter is called the appendix because it is appended to the argument list when the invokedynamic instruction is executed.

Typically the appendix is the `CallSite` reference produced by the bootstrap method, but the JVM does not care about this. As long as the adapter method in the CPCE knows what to do with the appendix stored with the CPCE, all is well.

As a corner case, if the appendix value is null, it is not pushed at all, and the adapter method must *not* expect the extra argument. The adapter method in this case could be a permanently linked reference to a static method with a signature consistent with the `invokedynamic` instruction. This would in effect turn the `invokedynamic` into a simple `invokestatic`. Many other such strength reduction optimizations are possible.

Linkage handshake

The adapter `Method*` pointer for an invokedynamic CPCE is not chosen by the JVM, but rather by trusted Java code. The same is true of the appendix reference.

In fact, the JVM does not directly invoke the bootstrap method. Instead, the JVM calls a HotSpot-specific method `MethodHandleNatives.linkCallSite` with the resolved bootstrap specifier information. (Other JVM implementations do not necessarily use this handshake.) The `linkCallSite` method performs the steps demanded by the JSR 292 bootstrap rules, and returns two coordinated values, the adapter method and its appendix.

Since Java cannot represent a raw `Method*` pointer, the method is wrapped in a private Java type called `MemberName`, akin to a Java mirror for a `Klass*`. The appendix is a simple `Object` reference (or null). After a little unpacking, these are plugged into the CPCE.

Adapter method for `invokedynamic`

In general, the adapter method is a specially generated method created on the fly by the JSR 292 runtime. It is generated from a lambda form which computes the current call site target and invokes that target. The lambda form takes leading parameters corresponding to the arguments stacked for the `invokedynamic` instruction, i.e., those required by the method signature of the instruction. The lambda form also takes a trailing appendix argument (if relevant). It then performs whatever actions required by the bootstrap method and its call site.

Here is an example of an adapter method, taken from an actual application:

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

Here the invokedynamic instruction takes two arguments, a double `a0` and a reference `a1`, and returns a reference `t4`. The appendix trails along at the end, in `a2`.

The body of the lambda form extracts a method handle target from the appendix using the subroutine `Invokers.getCallSiteTarget`. The method handle is bound to `t3`, and then immediately invoked on the two leading arguments, `a0` and `a1`.

As may be seen by inspecting the Java code, `getCallSiteTarget` expects to get a non-null `CallSite` argument. If this were to fail, it would mean that the trusted Java code has a bug in it, since the trusted code is responsible for returning to the JVM a consistent pair of adapter and appendix.

The special non-public routine `MethodHandle.invokeBasic` is an unchecked version of `MethodHandle.invokeExact`. It differs in two ways from `invokeExact`. First, it does not check that its callee has a type which (exactly) matches the types at the call site. (For better or worse, it will never throw `WrongMethodTypeException`.)

Second, it allows loose typing of its arguments and return value, according to the *basic type* scheme used in the JSR 292 runtime. (See above.)

Example execution sequence for `invokedynamic`

The target of the invokedynamic instruction's call site can be any method handle. In the simplest case it could be a direct method handle connecting the method containing the invokedynamic instruction to some other Java language method.

Here is an example of the sequence of events and stack frames that would make such a connection, from a method IndyUser.m1 to a target method LibraryCls.m2:

(indyUser.m1)	(LF adapter for indy)	(LF method for DMH)	LibraryCls.m2)
1.2D "3" > invokedynamic foo(DL)L			
1.2D "3" push CPC.appendix			
1.2D "3" (CS) > jump to CPC.method			
...	a0:1.2D a1:"3" a2: (CS) (CS) > invokestatic Invokers.getCallSiteTarget		
...	a0:1.2D a1:"3" a2: (CS) t3: (CS.target) (CS.target) 1.2D "3" > invokevirtual MH.invokeBasic (DL)		
...	...	a0: (CS.target) a1:1.2D as:"3" (CS.target) > invokestatic DMH.internalMemberName	
...	...	a0: (CS.target) a1:1.2D as:"3" t3: (MN) 1.2D "3" (MN) > invokestatic MH.linkToStatic (DL+L)L	

...	...	1.2D "3" (MN) > pop MN	
...	...	1.2D "3" > jump to MN.method	
...	...	...	10:1.3D I1:"3" code for LibraryCls.m2 (DL)L

There are two internal stack frames, one for the adapter bound to the invokedynamic call site, and one which handles invocations for the target method handle.

The special methods internalMemberName and linkToStatic are explained on the page about direct method handles.

More low-level details are explained on the page about invocation of method handles.