MethodHandles

Patch name: methn.patch

Calling Sequences

Method handle calling sequence is a variation of the virtual calling sequence. (The gory details are documented elsewhere.) The caller sets up the call as if the method handle were the receiver of a virtual call (which is nominally `invoke`).

Control is transferred to a location fetched from the method handle. This location is the method handle's `_call handler_`. The call handler serves for both interpreted and compiled calls. It is called directly from compiled code, and indirectly (after a load with a short negative displacement) from the interpreter.

The call handler is generally a stub of some sort, but may eventually be (in direct cases) the compiled entry point of the target method. The method handle class reserves a word (non-op) of storage which the JVM manages for the call handler.

The code of the call handler (whether interpreted or compiled) is responsible for guarding against signature mismatch. The caller passes the intended `Method`, in the same register used for the inline cache handshake (RAX, GS on SPARC).

If the call handler detects an unexpected type (via simple pointer comparison) it throws a `WrongMethodTypeException`, which is a close cousin to a `ClassCastException`.

For the simplest method handles (direct, internal type DMH), the interpreted side of the call handler loads the method handle's `methodOop` (stored in another field of the method handle instance) and sets up an interpreted call. The compiled side simply branches to the method's compiled entry point.

```
simplest call to a non-constant method handle

    ; callSite:
    ;     set #callSiteType, CHECK
    ;     load (MH + #handler), TEM
    ;     call (TEM)
    ---
    ; callHandler:
    ;     cmp (MH + #type), CHECK
    ;     jump,ne wrongMethodType
    ;     load (MH + #method), METHOD
    ;     load (METHOD + #compiledEntry), TEM
    ;     jump (TEM)
    ---
    ; compiledEntry:
    ;     ...
```

Thus, a compiled call to a non-constant method handle can be completed in a small handful of instructions, including about four memory references and two indirect branches.

Varieties of Method Handle

The simplest kind of method handle is the `direct` method handle (class `java.dyn.hotspot.DMH`). This method handle contains type, handler, and method fields. If the type exactly matches the normal type of the target method, then the handler can branch directly to the method's entry point. The method handle type may differ slightly from the target method's normal type, as long as the difference in types can be bridged by simple value checks with no data motion. When the method handle is created, a handler is fitted to it that performs any required value checks, usually narrowing reference conversions.

A direct method handle performs one specific kind of data motion: It must shift the arguments so as to discard the method handle, and promote the first argument after the method handle to the receiver position. This is called popping the method handle. It is easy to do this in the interpreted calling sequence, but requires motion for compiled sequences.

A pass-through method handle performs value checks (with no data motion, the same kind as above) and delegates control to another (arbitrary) method handle. The combination of a pass-through method handle with a direct one collapses to a direct one.

A method handle can target an abstract method. The handler is a variation of the vtable or itable stubs. In the case of an interface call the itable stub requires the interface as a parameter.

A bound method handle is a variation of a direct method handle. An instance of a bound method handle (class `java.dyn.hotspot.BMH`) has an extra "value" field to hold an initial argument to apply to the method. Like the direct kind, a bound method handle passes control to a specific concrete method. (There is no need for viable or viable dispatch because the receiver is pre-bound.) The call handler inserts the predetermined object in the position of receiver; this is the position previously occupied by the method handle itself. For compiled calling sequences, this can be done easily by loading the value field to overwrite the method handle. For interpreted calling sequences, the receiver value can be stored down into the argument list.

Optimizations
When a method is compiled, if the compiler detects that method handles have been created for it, it should compile one or more entry points for those method handles, and relink them to point to those entry points. A variation of normal "verified entry point" (VEP) structure can be used for method handles also, with the verification being against the required method type, rather than against the receiver’s class.

It is likely that we will want to compile alternate entry points in affected methods for cases like these:

- full reference type erasure, such as `(String,int)void => (Object,int)void`
- full erasure with boxing, such as `(String,int)void => (Object,Object)Object`
- boxing with varargs, such as `(String,int)void => (Object...)Object`
- bound method handles, perhaps with any of the above type variations

The optimization is reasonably self-limiting, as long as such alternate entry points are compiled only when the relevant method handles have actually been created.

If a method handle at a call site is a compile-time constant, the type check can be removed, and the method itself inlined. This is the case for `invokedynamic` instructions.