JEP 416: Reimplement Core Reflection with Method Handles

Owner	Mandy Chung
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	18
Component	core-libs / java.lang:reflect
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	M
Duration	M
Reviewed by	Alan Bateman, John Rose
Endorsed by	John Rose
Created	2021/04/26 22:41
Updated	2024/03/07 18:49
Issue	8266010

Summary

Reimplement `java.lang.reflect.Method`, `Constructor`, and `Field` on top of `java.lang.invoke` method handles. Making method handles the underlying mechanism for reflection will reduce the maintenance and development cost of both the `java.lang.reflect` and `java.lang.invoke` APIs.

Non-Goals

It is not a goal to make any change to the `java.lang.reflect` API. This is solely an implementation change.

Motivation

Core reflection has two internal mechanisms for invoking methods and constructors. For fast startup, it uses native methods in the HotSpot VM for the first few invocations of a specific reflective method or constructor object. For better peak performance, after a number of invocations it generates bytecode for the reflective operation and uses that in subsequent invocations.

For field access, core reflection uses the internal `sun.misc.Unsafe` API.

With the `java.lang.invoke` method-handle API introduced in Java 7, there are altogether three different internal mechanisms for reflective operations:

- VM native methods,
- Dynamically generated bytecode stubs for `Method::invoke` and `Constructor::newInstance`, along with `Unsafe` field access for `Field::get` and `set`, and
- Method handles.

When we update `java.lang.reflect` and `java.lang.invoke` to support new language features, such as those envisioned in Project Valhalla, we must modify all three code paths, which is costly. In addition, the current implementation relies on special treatment by the VM of the generated bytecode, which is wrapped in subclasses of `jdk.internal.reflect.MagicAccessorImpl`:

- Accessibility is relaxed so that these classes can access inaccessible fields and methods of other classes,
- Verification is disabled to work around JLS §6.6.2 in order to support reflection on `Object::clone`, and
- A non-well-behaved class loader is used to work around some security and compatibility issues.

Description

Reimplement `java.lang.reflect` on top of method handles as the common underlying reflective mechanism of the platform by replacing the bytecode-generating implementations of `Method::invoke`, `Constructor::newInstance`, `Field::get`, and `Field::set`.

The new implementation performs direct invocations of the method handles for specific reflective objects. We use the VM's native reflection mechanism only during early VM startup, before the method-handle mechanism is initialized. That happens soon after `System::initPhase1` and before `System::initPhase2`, after which we switch to using method handles exclusively. This benefits Project Loom by reducing the use of native stack frames.

For optimal performance, `Method`, `Constructor`, and `Field` instances should be held in `static final` fields so that they can be constant-folded by the JIT. When that is done, microbenchmarks show that the performance of the new implementation is significantly faster than the old implementation, by 43–57%.

When `Method`, `Constructor`, and `Field` instances are held in non-constant fields (e.g., in a non-`final` field or an array element), microbenchmarks show some performance degradation. The performance of field accesses is significantly slower than the old implementation, by 51–77%, when `Field` instances cannot be constant-folded.

This degradation may, however, not have much effect on the performance of real-world applications. We ran several serialization and deserialization benchmarks using real-world libraries and found no degradation in

- A custom JSON serialization and deserialization benchmark using Jackson,
- An XStream converter type benchmark, or
- A Kryo field serializer benchmark.

We will continue to explore opportunities to improve performance, for example by refining the bytecode shapes for field access to enable concrete `MethodHandle`s and `VarHandle`s to be reliably optimized by the JIT regardless of whether the receiver is constant.

The new implementation will reduce the cost of upgrading reflection support for new language features and, further, will allow us to simplify the HotSpot VM by removing the special treatment of `MagicAccessorImpl` subclasses.

Alternatives

Alternative 1: Do nothing

Retain the existing core reflection implementation to avoid any compatibility risk. The dynamic bytecode generated for core reflection would remain at class file version 49, and the VM would continue to treat such bytecode specially.

We reject this alternative because

- The cost of updating `java.lang.reflect` and `java.lang.invoke` to support Project Valhalla's primitive classes and generic specialization would be high,
- Additional special rules in the VM would likely be needed to support new language features within the limitation of the old class-file format, and
- Project Loom would need to find a way to handle the introduction of native stack frames by core reflection.

Alternative 2: Upgrade to a new bytecode library

Replace the bytecode writer used by core reflection to use a new bytecode library that evolves together with the class-file format, but otherwise retain the existing core reflection implementation and continue to treat dynamically-generated reflection bytecode specially.

This alternative has lower compatibility risk than what we propose above, but it is still a sizeable amount of work and it still has the first and last disadvantages of the first alternative.

Testing

Comprehensive testing will ensure that the implementation is robust and compatible with existing behavior. Performance testing will ensure that there are no unexpected significant performance regressions compared to the current implementation. We will encourage developers using early-access builds to test as many libraries and frameworks as possible in order to help us identify any behavior or performance regressions.

Baseline

```
Benchmark                                        Mode  Cnt    Score   Error  Units
ReflectionSpeedBenchmark.constructorConst        avgt   10   68.049 ± 0.872  ns/op
ReflectionSpeedBenchmark.constructorPoly         avgt   10   94.132 ± 1.805  ns/op
ReflectionSpeedBenchmark.constructorVar          avgt   10   64.543 ± 0.799  ns/op
ReflectionSpeedBenchmark.instanceFieldConst      avgt   10   35.361 ± 0.492  ns/op
ReflectionSpeedBenchmark.instanceFieldPoly       avgt   10   67.089 ± 3.288  ns/op
ReflectionSpeedBenchmark.instanceFieldVar        avgt   10   35.745 ± 0.554  ns/op
ReflectionSpeedBenchmark.instanceMethodConst     avgt   10   77.925 ± 2.026  ns/op
ReflectionSpeedBenchmark.instanceMethodPoly      avgt   10   96.094 ± 2.269  ns/op
ReflectionSpeedBenchmark.instanceMethodVar       avgt   10   80.002 ± 4.267  ns/op
ReflectionSpeedBenchmark.staticFieldConst        avgt   10   33.442 ± 2.659  ns/op
ReflectionSpeedBenchmark.staticFieldPoly         avgt   10   51.918 ± 1.522  ns/op
ReflectionSpeedBenchmark.staticFieldVar          avgt   10   33.967 ± 0.451  ns/op
ReflectionSpeedBenchmark.staticMethodConst       avgt   10   75.380 ± 1.660  ns/op
ReflectionSpeedBenchmark.staticMethodPoly        avgt   10   93.553 ± 1.037  ns/op
ReflectionSpeedBenchmark.staticMethodVar         avgt   10   76.728 ± 1.614  ns/op
```

New implementation

```
Benchmark                                        Mode  Cnt    Score   Error  Units
ReflectionSpeedBenchmark.constructorConst        avgt   10   32.392 ± 0.473  ns/op
ReflectionSpeedBenchmark.constructorPoly         avgt   10  113.947 ± 1.205  ns/op
ReflectionSpeedBenchmark.constructorVar          avgt   10   76.885 ± 1.128  ns/op
ReflectionSpeedBenchmark.instanceFieldConst      avgt   10   18.569 ± 0.161  ns/op
ReflectionSpeedBenchmark.instanceFieldPoly       avgt   10   98.671 ± 2.015  ns/op
ReflectionSpeedBenchmark.instanceFieldVar        avgt   10   54.193 ± 3.510  ns/op
ReflectionSpeedBenchmark.instanceMethodConst     avgt   10   33.421 ± 0.406  ns/op
ReflectionSpeedBenchmark.instanceMethodPoly      avgt   10  109.129 ± 1.959  ns/op
ReflectionSpeedBenchmark.instanceMethodVar       avgt   10   90.420 ± 2.187  ns/op
ReflectionSpeedBenchmark.staticFieldConst        avgt   10   19.080 ± 0.179  ns/op
ReflectionSpeedBenchmark.staticFieldPoly         avgt   10   92.130 ± 2.729  ns/op
ReflectionSpeedBenchmark.staticFieldVar          avgt   10   53.899 ± 1.051  ns/op
ReflectionSpeedBenchmark.staticMethodConst       avgt   10   35.907 ± 0.456  ns/op
ReflectionSpeedBenchmark.staticMethodPoly        avgt   10  102.895 ± 1.604  ns/op
ReflectionSpeedBenchmark.staticMethodVar         avgt   10   82.123 ± 0.629  ns/op
```

Risks and Assumptions

Code that depends upon highly implementation-specific and undocumented aspects of the existing implementation may be impacted. To mitigate this compatibility risk, as a workaround you can enable the old implementation via `-Djdk.reflect.useDirectMethodHandle=false`.

- Code that inspects the internal generated reflection classes (i.e., subclasses of `MagicAccessorImpl`) will no longer work and must be updated.
- Method-handle invocation may consume more resources than the old core reflection implementation. Such invocation involves calling multiple Java methods to ensure that the declaring class of a member is initialized prior to access, and thus may require more stack space for the necessary execution frames. This may result in a `StackOverflowError` or, if a `StackOverflowError` is thrown while initializing a class, then a `NoClassDefFoundError`.
- We will remove the old core reflection implementation in a future release. The `-Djdk.reflect.useDirectMethodHandle=false` workaround will stop working at that point.