Native Image Developer Experience

Vojin Jovanovic

@vojjov Twitter

@vjovanov GitHub

What is Special in Native Image?

Reflective accesses must be known at build time, see:

```
class Foo {};
public static void main(String[] args) throws ClassNotFoundException {
    Class.forName(args[0]);
}
```

Therefore Native Image needs reachability metadata at build time, for example:

```
{
  "name": "Foo"
}
```

Native Image tries to infer metadata entries in the code, for example:

```
Class.forName("Foo")
```

To help the user there is a metadata agent (JVMTI) that will collects reachability metadata

```
$ java -agentlib:native-image-agent=config-output-dir=META-INF/native-image/ App
```

Developer Experience in JDK 21

Imprecise errors when metadata is not specified

Overly verbose and complicated reachability metadata

- 5 JSON files for specifying what is reachable
- Quirks stemming from organic growth JDKs history

Lack of specification for metadata inference

- Semantics depends on compiler optimizations and reachability
- Differences in semantics between community and enterprise
- Unpredictable reflection behavior

Metadata agent: both too much metadata and too little metadata

```json
{
  "reflection": [
    {
      "type": "reflectively.accessed.Type",
      "fields": [{ "name": "field1"}],
      "methods": [{"name": "method1", "parameterTypes": []}],
      "allDeclaredConstructors": true,
      "allPublicConstructors": true,
      "allDeclaredFields": true,
      "allPublicFields": true,
      "allDeclaredMethods": true,
      "allPublicMethods": true,
      "unsafeAllocated": true
    }
  ],
  "jni": [
    {
      "type": "jni.accessed.Type",
      "fields": [{ "name": "field1"}],
      "methods": [{"name": "method1", "parameterTypes": []}],
      "allDeclaredConstructors": true,
      "allPublicConstructors": true,
      "allDeclaredFields": true,
      "allPublicFields": true,
      "allDeclaredMethods": true,
      "allPublicMethods": true
    }
  ],
  "resources": [
    {
      "module": "optional.module.of.a.resource",
      "glob": "path1/level*/**"
    }
  ],
  "bundles": [
    {
      "name": "fully.qualified.bundle.name",
      "locales": ["en", "de", "other_optional_locales"]
    }
  ],
  "serialization": [
    {
      "type": "serialized.Type",
      "customTargetConstructorClass": "optional.serialized.super.Type"
    }
  ]
}
```

Developer Experience Improvements in JDK 23

User-Friendly Reflection Semantics

Problem: imprecise errors when metadata is missing

1. A standard exception (e.g., `ClassNotFoundException`) is thrown (or swallowed)
2. The classes have incomplete fields and methods => semi-serialized objects

Solution: dynamic accesses must be registered or `MissingReflectionRegistrationError`

```
$ native-image --exact-reachability-metadata –jar myapp.jar        # for frameworks
$ native-image --exact-reachability-metadata=my.pkg –jar myapp.jar # for transitioning

$ ./myapp                                           # standard use
$ ./myapp -XX:MissingRegistrationReportingMode=Warn # initial run
$ ./myapp -XX:MissingRegistrationReportingMode=Exit # in testing and CI
```

Note: `--exact-reachability-metadata` becomes the default in JDK 25 so timely migration is advised

User-Friendly Reflection Error Messages

Instead of having an imprecise error cause:

```
Exception in thread "main" java.lang.NoSuchMethodException: java.lang.Exception.<init>()
```

The error messages are now precise and actionable:

```
Exception in thread "main" org.graalvm.nativeimage.MissingReflectionRegistrationError: The
program tried to reflectively access method

   java.lang.Exception#<init>()

without it being registered for runtime reflection. Add 'java.lang.Exception#<init>()' to the
reflection metadata to solve this problem. See https://www.graalvm.org/latest/reference-
manual/native-image/metadata/#reflection for help.
```

This error message will get more opinionated in the near future

Metadata: `type` instead of name (#7753)

Problem: metadata is too fine-grained

Solution: allow all reflective queries on registered types

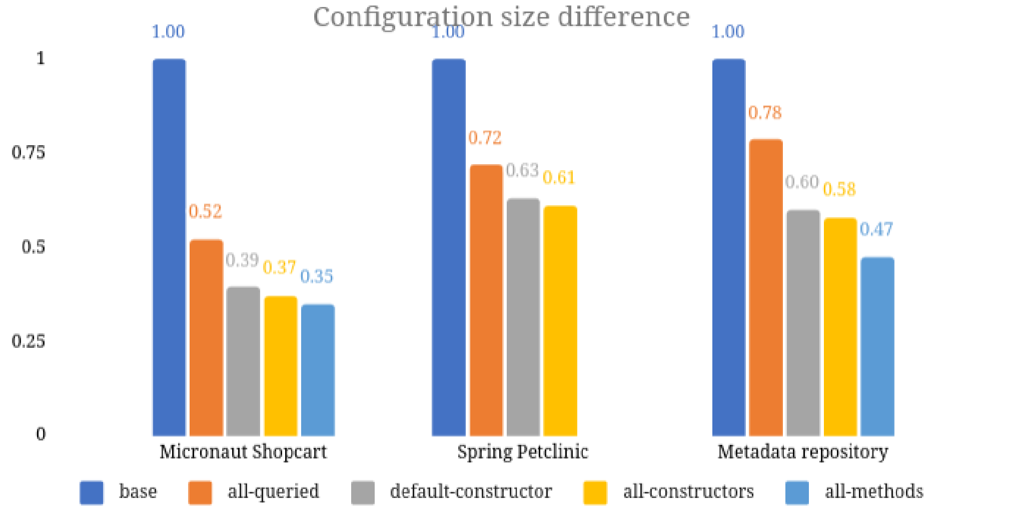
```
{
  "type": "reflectively.accessed.Type"
}
```

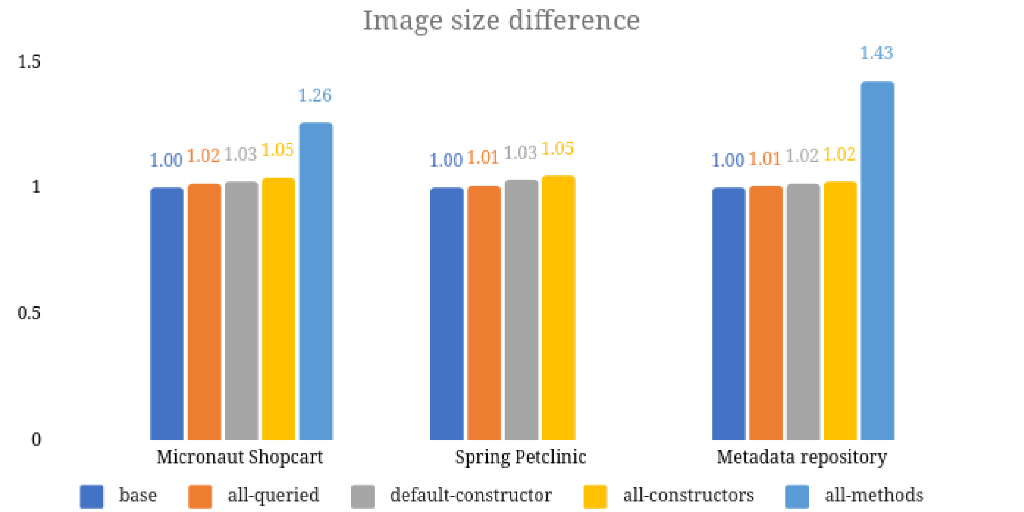
The new type entry now allows

- Fetching and loading the `java.lang.Class`
- 17 calls on `java.lang.Class`

A type does not allow

- Calling methods of the provided class
- Accessing fields of the provided class
- Unsafe allocation of the instance object





Metadata: Proxy Classes are Just Types (#7476)

Problem 1: reflection was not possible on proxy types

Problem 2: proxy classes required the separate `proxy-config.json` file

Solution: proxy is just a special kind of type specified with a list of interfaces

```
{
 "type": {
  "proxy": [ "sun.misc.SignalHandler" ]
 },
 "allDeclaredMethods": true
}
```

```
Proxy.newProxyInstance(
  classLoader, interfaces, handler)
```

Allows reflective method invocation

The entry above both generates the proxy class and registers it for reflection

- `Proxy.getProxyClass` is basically `Class.forName` for proxies

The same type of proxy entry is used for serialization additionally reducing complexity

The `proxy-config.json` file is now obsolete

Metadata: Restrict the Expressivity for Resource Inclusion (#7487)

Problem 1: the `include` patterns are complex for the users

Problem 2: the `exclude` patterns are not composable

Solution: support a subset of glob patterns and remove the exclude patterns altogether

- The ***** pattern that matches any number of characters within the current directory
- The ****** as a name component to recursively match any number of layers of directories

```
{
  "resources": [
    { "glob": "assets/**" },
    { "glob": "images/**/*.png" },
    { "glob": "META-INF/services/app.Service" },
  ]
}
```

Runtime-Checked Metadata Conditions (#7480)

Problem: using static analysis to determine elements registered for reflection

- Changes in analysis precision affect program correctness
- Semantics changes with layered images when the base includes all types

Solution: introduce runtime checks for typeReachable and rename it to typeReached

- An element will be included into the image as if the condition was reachable
- To access the element runtime, the guarding type must be reached

```
{
  "condition": { "typeReached": "app.ConditionType" },
  "type": "app.ReflectivelyAccessedType"
}
```

Debugging Flags to estimate the impact of the change:

- `-H:+TreatAllTypeReachableConditionsAsTypeReached`
- `-H:+TrackUnsatisfiedTypeReachedConditions`

Note: `typeReached` is still not supported in the Native Image Feature API (ETA, JDK 24)

Runtime-Checked Metadata Semantics (#7480)

A type is reached immediately before the static initialization routine

A type is always reached before any of its subtypes are reached

```
class SuperType {
    static {
        // ConditionType reached (subtype reached)
    }
}
class ConditionType extends SuperType {
    static {
        // ConditionType reached (before static initializer)
    }

    public static ConditionType main() {}
}
class App {
    public static void main(String[] args) {
        // ConditionType not reached
        var clazz = ConditionType.class;
        // ConditionType not reached (ConditionType.class doesn't start class initialization)
        ConditionType.singleton();
        // ConditionType reached (already initialized)

    }
}
```

Streamline Reachability Metadata

Problem: having 5 different files to specify reachability metadata is complex

Solution: use a single file (`reachability-metadata.json`) with 5 fields

```
{
  "reflection": [{"type": "reflectively.accessed.Type"}, ...],
  "resources": [{"glob": "assets/**/*.png"}, ...],
  "bundles": [{"name": "bundle.name"}, ...],
  "serialization":[{"type": "reflectively.accessed.Type"}, ...],
  "jni": [{"type": "jni.accessed.Type"}, ...]
}
```

Only `typeReached` conditions are allowed in `reachability-metadata.json`

The legacy field name is not allowed for reflection and JNI

Documentation and Testing of new Metadata

We have simplified metadata documentation to three pages:

- Reachability Metadata
- Collect Metadata with the Tracing Agent
- Project-Author Guide: How to Support my Library or Project

Introduced a version-agnostic test suite:

- Simple snippets testing every reflective method
- No reflection in the suite
- Tests that the agent outputs the same JSON metadata

```
@Test(metadata ="""
    {
     "reflection": [
      { "type": "pkg.ForNameTestClass" }
     ]
    }""",
    expectedNoRegistrationException = MISSING_REFLECTION_ERROR)
public static void forName() throws ClassNotFoundException {
   var result = Class.forName(opaque("pkg.ForNameTestClass"));
   assertEquals(ForNameTestClass.class, result);
}
```

Run the test suite on JDK 17, JDK 21, and the latest GraalVM

Metadata Migration and Backword&Forward Compatibility

Native Image will support old files (`reflect-config.json`, etc.) for a very long time

Backport PRs for to 21 and 17 should be merged ASAP in all Native Image distributions:

- JDK 21: https://github.com/oracle/graal/pull/9670
- JDK 17: https://github.com/oracle/graal/pull/9671

Use the tool native-image-configure to convert previous files to the latest ones

```
  $ native-image-configure generate --input-dir=<legacy-files> --output-dir=<new-metadata>
```

Note: Seldom there can be failures due to conversion of `typeReachable` to `typeReached`

Important: avoid distributing the new metadata to libraries until the backport PRs are adopted

Developer Experience in JDK 23

Clear errors when metadata is not specified

Less verbose reachability metadata

- 1 JSON file for specifying what is reachable
- Still some quirks that come from organic development the long history of the JDK

Lack of specification

- Semantics depends on compiler optimizations
- Differences in semantics between community and enterprise
- Unpredictable behavior

Metadata agent produces both too much metadata and too little metadata

Can we Make Developer Experience Better?

“Make everything as simple as possible, but not simpler.” -- Albert Einstein

“I have only made this letter longer because I have not had the time to make it shorter." -- Blaise Pascal

Streamline Reachability Metadata (#9679)

```
"reflection": [
  {
    "type": "reflectively.accessed.Type",
    "fields": [{ "name": "field1"}],
    "allDeclaredFields": true,
    "allPublicFields": true,
    "methods": [{"name": "method1", "parameterTypes": []}],
    "allDeclaredConstructors": true,
    "allPublicConstructors": true,
    "allDeclaredMethods": true,
    "allPublicMethods": true,
    "unsafeAllocated": true
  }
]
```

Always include with type?

```
"allMethods": true
```

Always true?

```
"resources": [
  {
    "module": "optional.module.of.a.resource",
    "glob": "path1/level*/**"
  }
]
```

Hmm, a JAR tells itself to include a subset of itself?

Streamline Reachability Metadata (#9679)

```
"bundles": [
  {
    "name": "path1/level*/**"
  }
]
```

= reflection + resources

```
"jni": [
  {
    "type": "jni.accessed.Type",
    "fields": [{ "name": "field1"}],
    "allDeclaredFields": true,
    "allPublicFields": true,
    "methods": [{"name": "method1", "parameterTypes": []}],
    "allDeclaredConstructors": true,
    "allPublicConstructors": true,
    "allDeclaredMethods": true,
    "allPublicMethods": true,
  }
]
```

same syntax as reflection

Streamline Reachability Metadata (#9679)

```
"serialization": [
  {
    "type": "serialized.Type",
    "customTargetConstructorClass": "optional.serialized.super.Type"
  }
]
```

Just reflection

Generate them all

The final result (hopefully):

```
{
  "reflection": [
    {
      "type": "reflectively.accessed.Type",
      "methods": [{"name": "method1", "parameterTypes": []}],
      "allMethods": true,
      "jniAccessible": true,
      "serializable": true
    }
  ]
}
```

Reachability Metadata: Consider Different Formats

Yaml:

```yaml
reflection:
  - type: reflectively.accessed.Type
    methods:
      - name: method1
        parameterTypes: []
    allMethods: true
    jniAccessible: true
    serializable: true
```

Toml:

```toml
[[reflection]]
type = "reflectively.accessed.Type"
allMethods = true
jniAccessible = true
serializable = true

[[reflection.methods]]
name = "method1"
parameterTypes = [ ]
```

Our own DSL:

```
[condition.Type] reflectively.queried.Type {serializable=true}
[condition.Type] !reflectively.accessed.Type {jniAccessible=true}
```

A Java API that corresponds to the language above but simpler than the Feature API

Reachability Metadata Specified In Code (#9679

Explore the simplified metadata with annotations

For querying a type

```
@ReflectivelyQueries("reflectively.queried.Type")
class ConditionType {}
```

For calling methods on a type

```
@ReflectivelyAccesses("reflectively.called.Type")
class ConditionType {}
```

Or for JNI accessing Java elements:

```
@ReflectivelyAccesses(reflectively.called.Type.class)
native Class<?> nativeCall()
```

Allow Bulk Inclusion of Whole Libraries

☺ 👍 53 🎉 1 ❤️ 2

First introduce the experimental flags for bulk inclusion:

- `-H:IncludeAllMetadataForModule=<module>`
- `-H:IncludeAllMetadataForClasspath=<path>`
- `-H:IncludeAllMetadataForPackage=<package>`
- `-H:+IncludeAllMetadata`

Perform experiments on image size, RSS, and startup time

Consider including glob-like wildcards

```
{
  "type": "reflectively.accessed.*"
}
```

Include all from a single package

```
{
  "type": "reflectively.accessed.**"
}
```

Include all from subpackages

Yes, this requires classpath scanning, but just for a small subset of the classpath

Specify Metadata Proofs and Implement them on Bytecode

Native Image proves some metadata entries from the user code

```
Class<?> clazz = Class.forName("app.Type");
Constructor<?> constructor = clazz.getConstructor();
return constructor.newInstance();
```

Great, less metadata, we want that!

Problem: the proofs are based on compiler optimizations: unpredictable and unspecifiable

Solution: Start from scratch and specify what can be proven and why

Challenge: find a sweet spot between complexity and the number of covered cases

Extra: do it as a bytecode transformation so we can use it with the metadata agent

Status: we have the intra-procedural semantics implemented, now the hard part

Improve Metadata Agent Accuracy

Given the metadata to the right, what is the agent output:

```
{ "type": "Foo" }
```

```
public static void main(String[] args) throws Exception {
    Class.forName(opaque("Foo"));
    Class.forName("Bar")
         .getMethod("baz").invoke(null);
}
```

Currently, the answer is

```
 {"type": "Foo"}
 {
   "type": "Bar"
   "methods": [{"name": "baz", "parameterTypes": []}]
 }
```

Already exists in the metadata

Can be inferred in the code

In addition, the agent will often introduce some JDK classes to the mix

Solution: the agent should output differential metadata and use the bytecode metadata inference

Towards Native Image Semantics on HotSpot

Assuming previous is done, how hard is it to have Native Image semantics on HotSpot?

- We have a clear specification for reflective operations
- We have bytecode-to-bytecode metadata inference

Benefits of having Native Image semantics on HotSpot

- Quick development turnaround
- Easy debugging with JDWP
- Can selectively allow run-time class definition (e.g., in tests)

Implemented prototype for run-time class definition and reflection

- ~150 lines of code in the JDK
- Directly using the reflection metadata module from Native Image
- The metadata agent implementation almost for free

Towards Better Native Image Developer Experience

Precise errors when the community adopts it

Simple metadata in a single file that keeps small binaries

Clear specification for metadata inference

- Semantics does not depend on compiler optimizations and reachability
- No differences in semantics between community and enterprise
- Predictable reflection behavior

Metadata agent produces exact metadata

- Differential agent operation
- Bytecode metadata inference

```
{
  "reflection": [
    {
      "type": "reflectively.accessed.Type",
      "methods": [{"name": "method1", "parameterTypes": []}],
      "allMethods": true,
      "jni" = true,
      "serializable" = true
    }
  ]
}
```

Native Build Tools and the Reachability Metadata Repository

Toolchain support for GraalVM in Gradle and Maven

- Automatic downloads of GraalVM in build tools

Improved build time via native image layers

Complete support for JUnit testing

Complete support for Mockito and other testing frameworks

Automatically test each new library version in the metadata repository

Metadata repository release monthly, snapshot release after every PR

Building most popular 500 libraries to find ones with missing metadata

- Contribute metadata to the cornerstone libraries

Thank you!

Minimize Build-Time Initialization of the JDK (#7488)

Problem: Native Image currently initializes large portions of the JDK at build time

- It requires user configuration even if they do not use initialization at build time
- It requires hardly-maintainable substitutions that introduce subtle bugs

Solution: Initialize only classes that are necessary for the Native Image runtime

Migration: can affect code that uses build-time initialization

Rolled out in 4 releases:

- The **experimental release:** behind an experimental flag, used by the frameworks
- The **preview release:** behind an API flag that is hinted to the users.
- The release where this feature becomes the **default** but can be switched off.
- The release where the previous functionality is **removed**.

Inspecting the Reachability Metadata (#7482)

Problem: no information about the elements that were included for reachability metadata

- Hard to know how the element got included
- Impossible to write tests without running actual programs

Solution: output all reachability metadata in JSON files

- Add a field `reasons` of type array to each element in metadata

```
{
  "reasons": [
    "app.Feature#beforeAnalysis",
    "jar://<path>/app.jar!META-INF/native-image/lib/reflect-config.json"
  ],
  "name": "sun.misc.Unsafe"
}
```

- Provide methods in the Feature API for inspecting the metadata?