https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md

GraalVM Native Image Compatibility and Optimization Guide

GraalVM Native Image uses a different way of executing Java programs than users of the Java HotSpot VM are used to. It distinguishes between *image build time* and *image run time*. At image build time, a static analysis finds all methods that are reachable from the entry point of the application. These (and only these) methods are then ahead-of-time compiled into the native image. Because of the different optimization model, Java programs can behave somewhat differently when compiled into a native image.

GraalVM Native Image is an optimization that reduces the memory footprint and startup time of an application. This requires a closed-world assumption, where all code is known at image build time, i.e., no new code is loaded at run time. As with most optimizations, not all applications are amenable for that optimization. If an application is not optimizable, then a so-called fallback image is generated that launches the Java HotSpot VM, i.e., requires a JDK for execution.

Class Metadata Features (Require Configuration)

The following features generally require configuration at image build time in order to use the closed-world optimization. Configuration ensures that the minimum amount of space necessary is used in the native image binary. If one of the following features is used without providing a configuration at image build time, a fallback image is generated.

Dynamic Class Loading

Any class to be accessed by name at image run time must be enumerated at image build time. For example, a call to `Class.forName("myClass”)` must have `myClass` in a configuration file. If the configuration file is used, but does not include a class that is requested for dynamic class loading, a `ClassNotFoundException` will be thrown, as if the class was not found on the class path or was inaccessible.

Reflection

This category includes listing methods and fields of a class; invoking methods and accessing fields reflectively; and using other classes in the package `java.lang.reflect`.
Individual classes, methods, and fields that should be accessible via reflection need to be known ahead-of-time. Native Image tries to resolve these elements through a static analysis that detects calls to the reflection API. Where the analysis fails the program elements reflectively accessed at run time must be specified during native image generation in a configuration file or by using `RuntimeReflection` from a `Feature`. For more details, read our documentation on reflection.
During native image generation, reflection can be used without restrictions during native image generation, for example in class initializers.

Dynamic Proxy

This category includes generating dynamic proxy classes and allocating instances of dynamic proxy classes using the `java.lang.reflect.Proxy` API. Dynamic class proxies are supported with the closed-world optimization as long as the bytecode is generated ahead-of-time. This means that the list of interfaces that define dynamic proxies needs to be known at image build time. Native Image employs a simple static analysis that intercepts calls to `java.lang.reflect.Proxy.newProxyInstance(ClassLoader, Class<?>[], InvocationHandler)` and `java.lang.reflect.Proxy.getProxyClass(ClassLoader, Class<?>[])` and tries to determine the list of interfaces automatically. Where the analysis fails the lists of interfaces can be specified in a configuration file. For more details, read our documentation on dynamic proxies.

JCA (Java Cryptography Architecture)

The JCA security services must be enabled using the option `--enable-all-security-services`. They require a custom configuration on Native Image since the JCA framework relies on reflection to achieve algorithm extensibility. For more details, read our documentation on security services.

JNI (Java Native Interface)

Native code may access Java objects, classes, methods and fields by name, in a similar way to using the reflection API in Java code. So, for the same reasons, any Java artifacts accessed by name via JNI must be specified during native image generation in a configuration file. For more details, read our JNI implementation documentation.

Alternative: In addition to JNI, GraalVM Native Image provides its own native interface that is much simpler than JNI and with lower overhead. It allows calls between Java and C, and access of C data structures from Java code. However, it does not allow access of Java data structures from C code. For more details, read our JavaDoc of the package `org.graalvm.nativeimage.c` and its subpackages.

Features Incompatible with Closed World Optimization

Some Java features are not yet supported with the closed-world optimization, and if used, lead to a fallback image.

`invokedynamic` Bytecode and Method Handles

Under the closed-world assumption, all methods that are called and their call sites must be known. `invokedynamic` and method handles can introduce calls at runtime or change the method that is invoked.
Note that `invokedynamic` use cases generated by `javac` for, e.g., Java lambda expressions and string concatenation are supported because they do not change called methods at image run time.

Serialization

Java serialization requires class metadata information in order to function, and could be supported similarly to reflection using configuration at image build time. However, Java serialization has been a persistent source of security vulnerabilities. The Java architects have announced that the existing serialization mechanism will be replaced with a new mechanism avoiding these problems in the near future. Note that this limitation extends to packages that rely on serialization, such as java.rmi.

Security Manager

The Java security manager is no longer recommended as a way to isolate less trusted code from more trusted code in the same process. This is because almost all typical hardware architectures are susceptible to side-channel attacks to access data that is restricted via the security manager. Using separate processes is now recommended for these cases.

Features That May Operate Differently in Native Image

GraalVM Native Image implements some Java features in a different way than the Java HotSpot VM.

Class Initializers

By default, classes are initialized at image run time. This ensures compatibility, but limits some optimizations. For faster startup and better peak performance, it is desirable to initialize classes at image build time. Class initialization behavior can be adjusted using the options `--initialize-at-build-time` or `--initialize-at-run-time` for specific classes and packages or for all classes. See `native-image --help` for details. Classes of the JDK class libraries are handled for you and do not need special consideration from the user.
Native image users should be aware that class initialization at image build time may break specific assumptions in existing code. For example, files loaded in a class initializer may not be in the same place at image build time as at image run time. Also, certain objects such as a file descriptors or running threads must not be stored into a native image binary. If such objects are reachable at image build time, image generation fails with an error.

Finalizers

The Java base class `java.lang.Object` defines the method `finalize()`. It is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the `finalize()` method to dispose of system resources or to perform other cleanup.
Finalizers are deprecated since Java 9. They are complicated to implement, and have badly designed semantics. For example, the finalizer can make the object reachable again by storing it in a static field. Therefore, finalizers are not invoked. We recommend replacing finalizers with weak references and reference queues for use in any Java VM.

Threads

SubstrateVM does not implement long-deprecated methods in `java.lang.Thread` such as `Thread.stop()`.

Unsafe Memory Access

Fields that are accessed using `sun.misc.Unsafe` need to be marked as such for the static analysis if classes are initialized at image build time. In most cases, that happens automatically: field offsets stored in `static final` fields are automatically rewritten from the hosted value (the field offset for the VM that the image generator is running on) to the native image value, and as part of that rewrite the field is marked as `Unsafe`-accessed. For non-standard patterns, field offsets can be recomputed manually using the annotation `RecomputeFieldValue`.

Debugging and Monitoring

Java has some optional specifications that a Java implementation can use for debugging and monitoring Java programs, including JVMTI. They help you monitor the VM at run time for events like compilation for example, which don’t occur in most native images. These interfaces are built on the assumption that Java bytecode is available at run time, which is not the case for native images built with the closed-world optimization. Because native image generates a native binary, users must use native debuggers and monitoring tools (like GDB or VTune) rather than tools targeted for Java. JVMTI and other bytecode-based tools are not supported with native image.