GRAALVM

GraalVM: Native Images in Containers

Put Java apps into containers, run them as native apps, and get faster startup times and lower memory overhead.

by Oleg Šelajev

GraalVM is a high-performance virtual machine for running programs in different languages. Currently, it supports JVM languages such as Java, Scala, Kotlin, and Groovy. It also supports JavaScript and Node.js, Ruby, R, Python, and the native languages that use LLVM. It's a very versatile project. However, one GraalVM capability is perhaps the most exciting for cloud deployments and the containers world. GraalVM can compile the JVM bytecode to the native executable or a shared library ahead of time in a way in which the resulting binary does not depend on the JVM for the execution.

This executable can be placed as a standalone application in a container and started really, really fast. Besides the quick startup time, GraalVM native images have low runtime memory overhead, which makes them even more attractive for use in the cloud.

Getting Started

Let's start at the beginning and create a GraalVM native image from an example application. First, you need a GraalVM distribution; download one from the GraalVM website. Both the community edition and the enterprise edition can create native images.

Unpack the archive and set `$GRAALVM_HOME` to point to the GraalVM directory; you can also point `$GRAALVM_HOME/bin` (or `$GRAALVM_HOME/Contents/Home/bin` on macOS) to the path for convenience. Once this is done, the utility for producing native images, called native-image, is available to you. Check the setup with `$GRAALVM_HOME/bin/native-image –version`.

Let's use GraalVM on a small example application. Clone `https://github.com/graalvm/graalvm-demos/` and navigate to the `native-list-dir` directory. There you'll find the `ListDir.java` class, which is a simple utility that traverses the filesystem and prints some information about what it finds. The code is straightforward:

```
public class ListDir { public static void main(Stri
         throws java.io.IOException {
    String root = ".";
    if(args.length > 0) {
        root = args[0];
    }
    System.out.println("Walking path: " + Paths.get
    long[] size = {0};
    long[] count = {0};
    try (Stream<Path> paths = Files.walk(Paths.get(
        paths.filter(Files::isRegularFile).forEach(
            File f = p.toFile();
            size[0] += f.length();
```

```
            count[0] += 1;
        });
    }
    System.out.println("Total: " +
        count[0] + " files, total size = " + size[0
}
```

Compile this code to a `.class` file, because native-image operates on the bytecode level, which allows it to support other JVM languages, too.

After running `javac ListDir.java`, run `native-image ListDir`.

You also can use native-image on a collection of JAR files; you just need to specify the classpath and the main class for the executable. The native-image utility will analyze your application, statically determine which other classes it uses (both in your dependencies and the JDK library), and create a map of reachable classes and method calls. It does this analysis statically and depends on a "closed universe" premise—making sure that all bytecode files ever to be executed in the resulting executable are present at the native image generation time.

One important feature of GraalVM native images is that the generation process can evaluate the static initializers of classes at generation time and store the preinitialized data structures in the resulting image heap.

Moments after the analysis, you can find a `listdir` file. For me, on macOS, it's a native macOS executable. It is linked to the operating system libraries directly without the JVM.

The file itself is a few megabytes. It contains the sample program compiled ahead of time and the JDK classes it uses, such as the `java.lang` classes or `Exception` classes, which can be thrown at any time. However, even with all the required classes, the size of these native executables is often smaller than the full distribution of the JDK that would otherwise be needed to run the program.

Let's run the Java version and the native binary and then time the execution using the UNIX time command in the Java directory:

```
$ time java ListDir
Walking path: .
Total: 7 files, total size = 8366834 bytes
java ListDir  0.22s user 0.06s system 51% cpu 0.555
```

Now let's run the Graal native implementation:

```
$ time ./listdir
Walking path: .
Total: 7 files, total size = 8366834 bytes
./listdir  0.00s user 0.00s system 66% cpu 0.011 to
```

You can see they produce the same result, and although the time of the Java version isn't bad, the time used by the native image version is almost zero.

One important feature of GraalVM native images is that the generation process can evaluate the static initializers of classes at generation time and store the preinitialized data structures in the resulting image heap. It's a configurable option, but it's useful for shaving off the last milliseconds of startup time.

This design, however, poses an interesting challenge for native images: What if the program you try to compile ahead of time uses production instance initialization in the class initializers, for example, creating thread pools, opening files, or mapping memory? It would not make any sense to perform these actions during the image generation phase, which

usually won't be done in the production environment but on a continuous integration server, for example. The native-image utility will back down and refuse to compile your app if class initializers perform actions that don't make sense at image generation time. And you'd need to configure which classes should be initialized at runtime by using the `--delay-class-initialization-to-runtime=classname`, list option.

Handling Special Cases

There are a few more things that require configuration at native image generation time. The most obvious is, perhaps, reflection. Java code can inspect the class data, load additional classes, or invoke methods using the Reflection API. Because the Reflection API allows fully dynamic access to the classes and objects, static analysis cannot resolve all classes that must be included in the native image. This doesn't mean GraalVM native images cannot process any code that uses reflection. You just need to list ahead of time the classes and methods that will be used reflectively. The format of the configuration is a JSON file listing the classes and the files. Imagine you have two classes like the following, where one calls into the other via reflection:

```
package org.example;
class ReflectionTarget {
    public String greet();
}
```

and

```
import java.lang.reflect.Method;
public class Main {

    public static void main(String[] args) throws E
        System.out.println(
            getResult(Class.forName("org.example.Re
    }
    private static Object getResult(Class<?> klass)
        Method method = klass.getDeclaredMethod("gr
        return method.invoke(
                klass.getDeclaredConstructor().newIn
    }
}
```

To compile them as a native image, you provide the following JSON file and specify it on the command line using the `-H:ReflectionConfigurationFiles=` command-line parameter:

```
[
  {
  "name" : "org.example.ReflectionTarget",
  "methods" : [
     {
       "name" : "<init>",
       "parameterTypes" : []
     },
     {
       "name" : "greet",
       "parameterTypes" : []
     }
  }
]
```

This file specifies which classes, methods, and constructors will be accessed reflectively. In a similar manner, you would typically need to configure Java Native Interface (JNI) access if the application you are compiling to a native image uses JNI.

You might imagine that providing such a configuration could become annoying, especially if the code that uses reflection is not yours but

comes from a dependency. In such a case, you can use the configuration javaagent that GraalVM provides. Run your application with the agent attached, and it will record all uses of reflection, JNI, and anything else that you need to configure for the native image:

```
$ /path/to/graalvm/bin/java \
    -agentlib:native-image-agent=trace-output=/path/
```

You can run it multiple times, producing different trace files to ensure that all relevant code paths are executed at least once and the native-image utility has the full picture of the code you want to run.

You can run the tracing agent when you execute tests. Tests usually cover the most important code path. (If not, perhaps you should correct that first.) When traces are collected, you can turn them into a native-image configuration file:

```
$ native-image --tool:native-image-configure
$ native-image-configure process-trace \
  --output-dir=/path/to/config-dir/ /path/to/trace-
```

The commands above will process the trace file and output the required configuration JSON files: `jni-config.json`, `reflect-config.json`, `proxy-config.json`, and `resource-config.json`.

After this preparation, using the generated configuration is pretty straightforward. The following command would take the configuration into account:

```
$ native-image -H:ConfigurationFileDirectories=/path
```

Another important configuration option to know is the `--allow-incomplete-classpath` option. Java applications often check for the existence of a class on the classpath and behave differently based on its availability. The classic example of such behavior is perhaps the logging configuration, which might state that if the `logback` library classes are available, then configure `logback`; otherwise, check for `log4j2` and configure it if it's available; if it's not, then fall back to `log4j`, and so on. How can native-image—which requires that all the classes be present for the analysis and which eagerly follows all the code paths—deal with such code? The answer is simple: By default, it currently refuses to compile code that uses this pattern, but if you explicitly say that an incomplete classpath is not a problem, it can compile the code without including those code paths.

There are plenty of configurable options that influence the behavior of the native-image generation, such as the ones we looked at earlier. As a developer, you have access to all the configurations that can make GraalVM native images successfully process more programs.

Performance

Let's consider the performance of native images. You saw in the earlier example that a native image can start in milliseconds. What about its throughput capabilities? After all, you know that just-in-time (JIT) compilers typically target peak performance rather than the speed of the startup or warm-up. Native images are not sluggish, but a warmed-up JIT compiler would be preferable performance-wise for long-running workloads. With this in mind, let's take a look at a sample Netty-based web service application.

First, build and produce the native image binary with the following commands:

```
$ mvn clean package
$ native-image -jar target/netty-svm-httpserver-full
   -H:ReflectionConfigurationResources=\
netty_reflection_config.json \
   -H:Name=netty-svm-http-server \
   --delay-class-initialization-to-runtime=\
io.netty.handler.codec.http.HttpObjectEncoder \
   -Dio.netty.noUnsafe=true
```

Now you can start the native file and check how fast it works for a single request and for some load.

For this test, I used the wrk2 benchmarking tool to generate the load and measure the latencies of the responses from the service. On my MacBook I ran the following, which specifies 2 threads and 100 simultaneous connections to keep a stable request rate of 2,000 per second for 30 seconds:

```
$ wrk -t2 -c100 -d30s -R2000 http://localhost:8080/
```

Here are the results. I'll first show the bytecode version of the Netty sample application and then the native version.

Java bytecode version:

```
Running 30s test @ http://127.0.0.1:8080/
  2 threads and 100 connections
  Thread calibration: mean lat.: 1.386ms, sampling
  Thread calibration: mean lat.: 1.362ms, sampling
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.30ms  573.88us   3.34ms   65.01%
    Req/Sec     1.05k   181.18     1.67k    78.84%
  59802 requests in 30.00s, 5.70MB read
Requests/sec:   1993.21
Transfer/sec:    194.65KB
```

Native image version, which produces a very similar result:

```
$ wrk -t2 -c100 -d30s -R2000 http://127.0.0.1:8080/
Running 30s test @ http://127.0.0.1:8080/
  2 threads and 100 connections
  Thread calibration: mean lat.: 1.196ms, sampling
  Thread calibration: mean lat.: 2.788ms, sampling
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.43ms  715.90us   5.78ms   70.34%
    Req/Sec     1.07k     1.37k    5.55k    89.40%
  58898 requests in 30.01s, 5.62MB read
Requests/sec:   1962.88
Transfer/sec:    191.69KB
```

[Note the output for both results has been slightly truncated to fit the page. —*Ed.*]

This is not a rigorous benchmark, of course, but these numbers demonstrate that for a short span, a native image can show similar performance to the JDK version of an application.

If you want even better throughput of native images, you can consider the Oracle GraalVM enterprise version of GraalVM, which is a proprietary product and includes additional performance enhancements. For native images, it includes profile-guided optimizations among other optimizations, which means you can build an instrumented image, gather the profile data by applying load, and then build the final image with performance optimizations that are tailored to the application's specific needs. This brings performance almost to the levels of the warmed-up JIT.

Memory Consumption

Let's talk about memory consumption. One of the typical complaints about using the JVM for serverless workloads is that it takes quite a bit of memory, even for a one-off task such as processing an individual request. (If you're interested in how GraalVM native images perform in that regard, the previous Netty application on my machine consumes 30 MB of memory total—including the heap.)

A native image also has garbage collection. It runs your program and collects unused objects at runtime to create the illusion of infinite memory. This is not new; any JVM does the same. Moreover, the JVM usually offers you a choice of garbage collection algorithms tuned for low latencies, minimal CPU consumption, or anything in between.

The garbage collector in native images is not the one you run in the JVM. Rather, it's a special implementation of a garbage collector written in Java that is a nonparallel generational scavenger. For simplicity, you can think of it as a somewhat simpler implementation of the parallel garbage collector that is the default in JDK 8. It splits the heap into generations; new objects are created in the so-called eden (see **Figure 1**) and then they are either collected or promoted to the old generation.

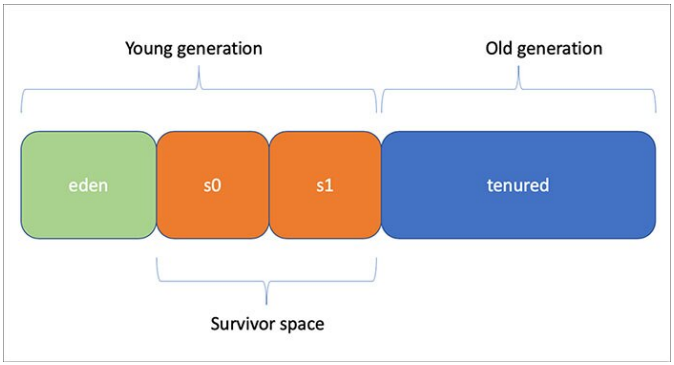

Figure 1. Garbage collector in native images

You can tune the garbage collector options for native images. Typically, you might want to adjust the maximum heap size. You can configure it with the `-Xmx` command-line parameter. If you'd like to better understand the garbage collector patterns of your native image, you can use the `-R:+PrintGC` or `-R:+VerboseGC` flags to get a summary of the garbage collector information before and after each collection. Native images often require less memory; one reason is that they do not need or include the machinery to load new classes dynamically, store their metadata for possible reflection, or compile them at runtime.

Conclusion

All in all, GraalVM native images offer a great opportunity to run Java applications in containers without loading the Java runtime. They also offer almost instantaneous startup and very low runtime memory overhead. This can be very important for cloud deployments where you want to autoscale your services or you have compute and memory constraints, such as in a function as a service (FaaS) environment.

Native images are an experimental feature of GraalVM, and today you can find applications that won't work with them out of the box. But many nontrivial apps work, and there are frameworks that accept GraalVM native images as a deployment target to simplify their usage. If you're deploying your apps in containers and you value startup performance and low runtime memory consumption, you'll likely find GraalVM native images very useful.

Also in This Issue

Oleg Šelajev

Oleg Šelajev (@shelajev) is a developer advocate at Oracle Labs, working on GraalVM—the high-performance embeddable polyglot virtual machine. He organizes the VirtualJUG, the online Java User Group, and a GDG chapter in Tartu, Estonia. In his spare time, he is pursuing a PhD in dynamic system updates and code evolution. He became a Java Champion in 2017.

Share this Page