Google

Hermetic Java™ for OpenJDK discussion

Self-contained high performance Java^{TM} executable images

Native Images for Java: Existing Approaches, Project Leyden, ...

Graal Native Image

- Graal native image compiles Java code ahead of time to executable images (as standalone executables or shared libraries)
  - Include application classes, dependency classes, and statically linked JDK natives
  - Include a substrate VM for runtime with memory management, thread scheduling, etc
  - Closed-world: allows advanced optimizations

Project Leyden

- Project Leyden is aimed to address some of the Java's long-term pain points
  - Slow startup time
  - Slow to reach runtime peak performance
  - Large footprint
- A static image derived from an application
  - Standalone program for running the application
  - Can contain class metadata, initial Java heap with populated Java objects, compiled code, auxiliary data, etc
- A closed world
  - Only load classes from the static image

CRaC (Coordinated Restore at Checkpoint)

- CRaC - checkpoint and restore for Java program
  - New standard API to notify checkpoint and restore events
  - Smaller image
  - Checkpoint and restore safety

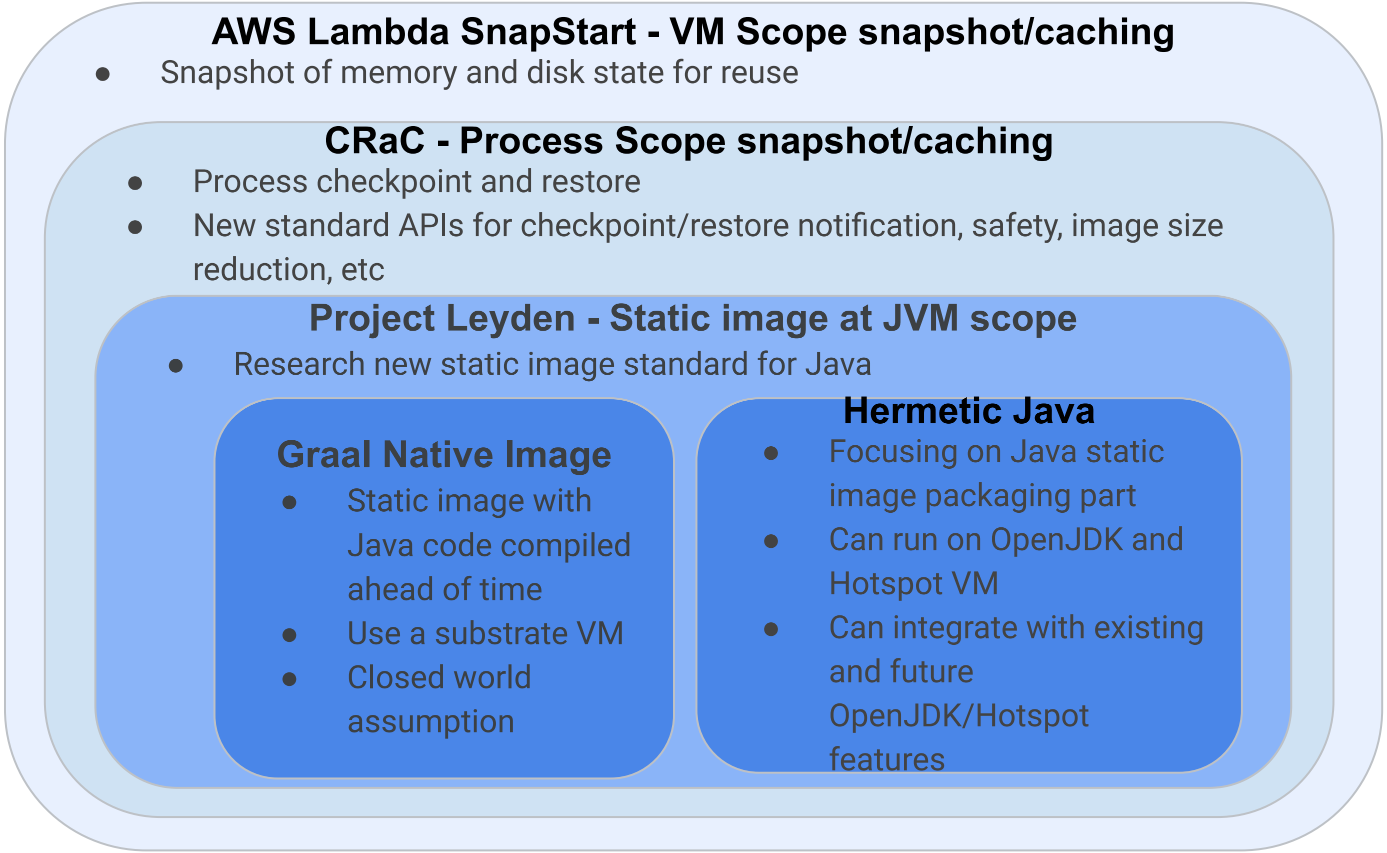
AWT Lambda SnapStart

- Lambda SnapStart (blog)
  - Makes use of Firecracker MicroVM snapshot (github repo)
  - Bypass usual Init phase when using a cached snapshot in subsequent invocations

Our Proposal - Hermetic Java

- Address Java application packaging and deployment issues
- A self-contained static image created at build time - combine launcher executable, JDK runtime and JAR
    - Application and JDK runtime environment are packaged in the image, including
        - Application and library classes, resources, JNI natives etc
        - Launcher executable, hotspot JVM and needed JDK libraries
    - Image starts with an ELF executable (Java launcher) at the beginning - executable image
        - Can work with other executables that are not affected by appended external data
    - Currently experimented on Linux only
    - JAR content can be examined and extracted by standard `jar` tool
    - Self-contained image works well with closed-world assumption; Allow dynamic loading external classes when necessary

Technical Landscape



AWS Lambda SnapStart - VM Scope snapshot/caching

- Snapshot of memory and disk state for reuse

CRaC - Process Scope snapshot/caching

- Process checkpoint and restore
- New standard APIs for checkpoint/restore notification, safety, image size reduction, etc

Project Leyden - Static image at JVM scope

- Research new static image standard for Java

Graal Native Image

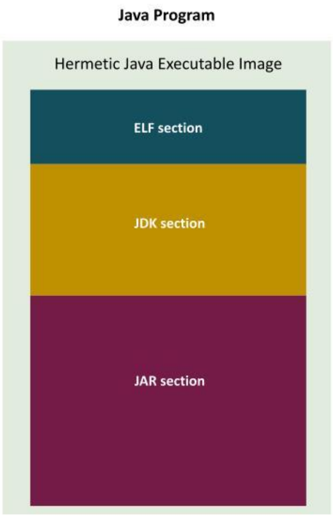
- Static image with Java code compiled ahead of time
- Use a substrate VM
- Closed world assumption

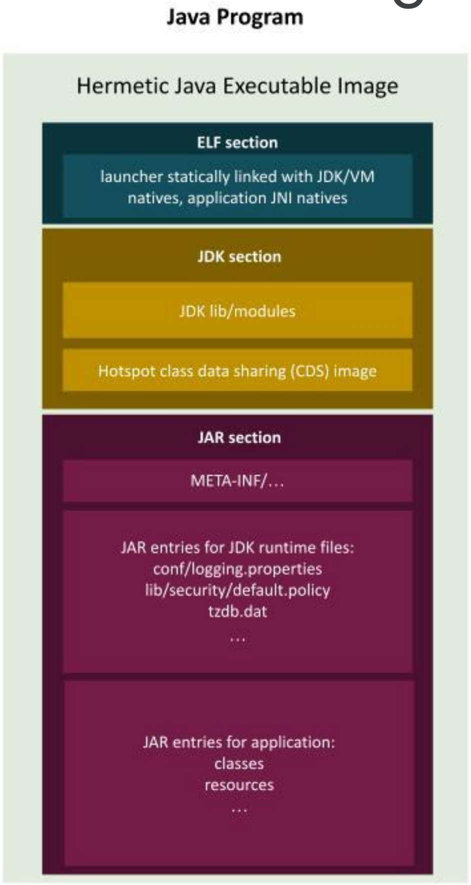
Hermetic Java

- Focusing on Java static image packaging part
- Can run on OpenJDK and Hotspot VM
- Can integrate with existing and future OpenJDK/Hotspot features

Hermetic Java Overview

Anatomy of Hermetic Java Executable Image





Java Program
Hermetic Java Executable Image
ELF section
launcher statically linked with JDK/VM natives, application JNI natives
JDK section
JDK lib/modules
Hotspot class data sharing (CDS) image
JAR section
META-INF/...
JAR entries for JDK runtime files:
conf/logging.properties
lib/security/default.policy
tzdb.dat
...
JAR entries for application:
classes
resources
...

- No external JDK runtime files required
- ELF section can support other executable formats that allow appending external data
- Platform independent image format

Why Hermetic Java? - Benefits of Single Executable Image

Simplify deployment of applications in both traditional and cloud environments

- No need to specify required JDK version for deployment
- No need to install required JDK runtime on target platform

Eliminate JDK version skew issue - ensure hermeticity

- The JDK being tested within the image is the one used in production
- No untested combination of application and JDK binaries

Ensure binary compatibility with JDK runtime

- Ahead-of-time compiled code (AOT)
- Class Data Sharing (CDS) archive

Why Hermetic Java? - Unique Benefits Comparing to Alternatives

Require no explicit runtime extraction

- Execution in place
- Works in different environments
  - Desktop, cloud instances, devices, etc
- Avoid headaches caused by temp file system space issue, etc.

Alternative: Package JDK as is, extract JDK at image download/install time or at startup time.

Smaller static footprint

- Only contain application and needed JDK runtime
  - Potential image size optimizations allowed at image build time, e.g. jlink produce minimum runtime

Alternative: Process-based or container/vm-based snapshot/resume

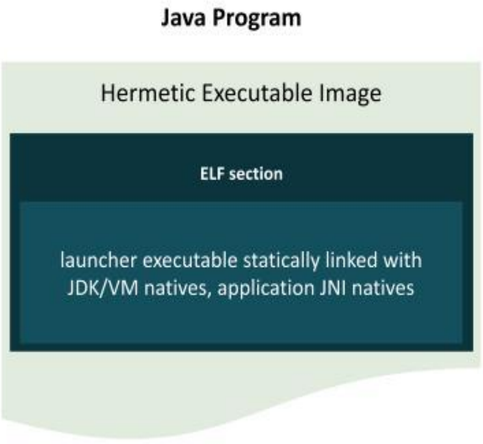
OpenJDK and Hotspot VM based solution

- G1 GC, c1/c2 compiler, etc
- Can work with JDK module system and jlink, etc

How JNI Natives Are Supported with Hermetic Java?

ELF Section In Hermetic Java Image

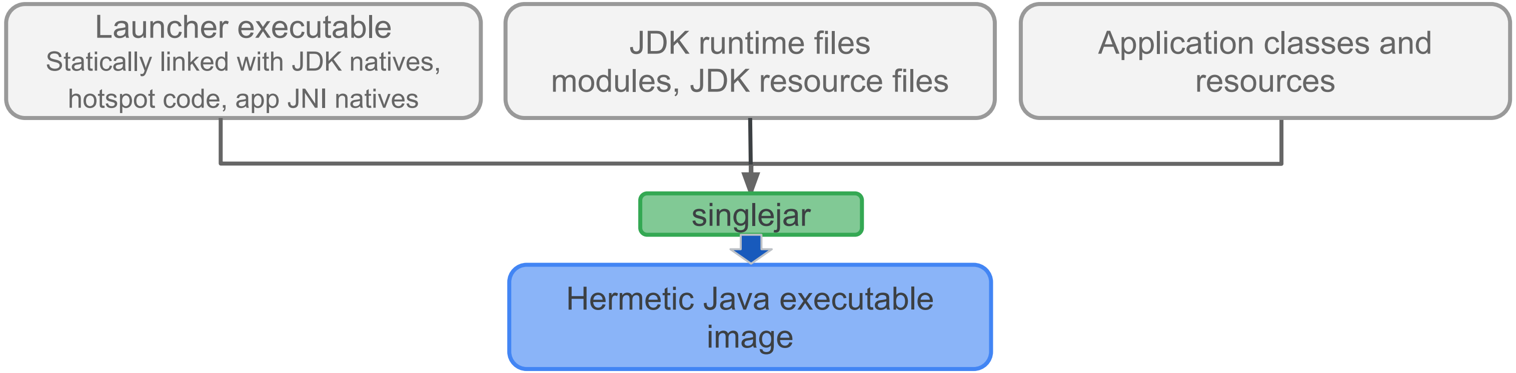
- Located at the beginning of the hermetic Java execution image
- Contain an ELF file with standard ELF format
  - Launcher executable
  - Statically linked with all VM and JNI native code
- Image can be loaded and executed as an ELF binary
- Can be processed by `readelf`, `objdump`, etc
- Debugging works normally, e.g. with `gdb, lldb`


Java Program
Hermetic Executable Image
ELF section
launcher executable statically linked with JDK/VM natives, application JNI natives

JDK Static Linking

- Build on top of existing OpenJDK work - becomes a complete solution for static linking for JDK
  - JDK-8005716: Enhance JNI specification to allow static JNI libraries
  - JDK-8136556: Add the ability to perform static builds of MacOSX x64 binaries
  - JDK-8232748: Build static versions of certain JDK libraries
- Support both dynamic and static linking with the same set of `.o` files
  - Use weak symbols to detect static linking
  - Remove dynamic linking assumptions in JDK and hotspot VM code

Singlejar - Packaging Tool



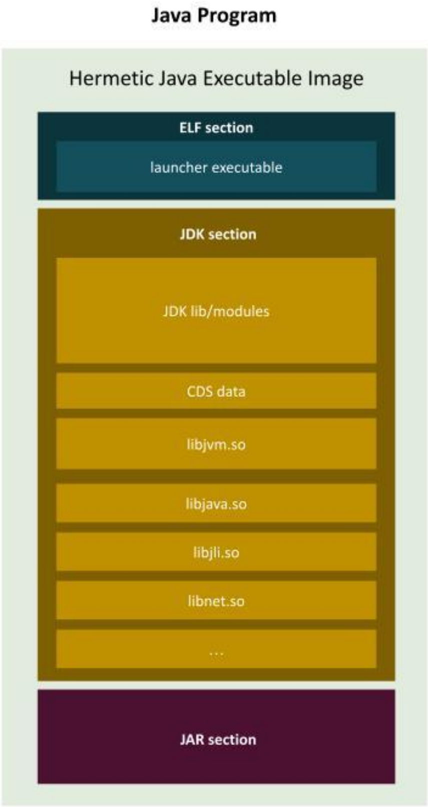
Launcher executable
Statically linked with JDK natives, hotspot code, app JNI natives
JDK runtime files
modules, JDK resource files
Application classes and resources
singlejar
Hermetic Java executable image

- JDK binary provides both `.so` and `.a` for JVM and JDK native code
- Application can build hermetic Java image as a post build process
  - Use pre-built statically linked standard launcher
  - Or, statically link JDK/VM .a static libraries with custom launcher
- Build hermetic Java image using singlejar
  - Enhanced with hermetic packaging support

Enhanced JDK Built-in Library/Agent Support

- Support uniquely defined `JNI_OnLoad_<lib_name>|JNI_OnUnload_<lib_name>|Agent_OnUnload_<agent_name>|Agent_OnAttach_<agent_name>` by default
  - Non-builtin application JNI libraries can continue use `JNI_OnLoad|JNI_OnUnload|Agent_OnLoad|Agent_OnUnload|Agent_OnAttach`
- ClassLoader and agent support are enhanced to support built-in native/agent libraries transparently
  - Lookup using unique `Agent_On(Un)Load/Attach<_agent_name>` first, fallback to conventional naming

Alternative Approach - What about dynamic Linking?

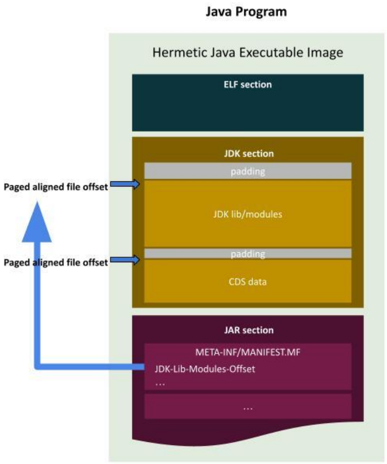


- Potential glibc RFE: `dlopen` of in-memory `ET_DYN` or `ET_EXEC` object
  - Use file embedded DSOs
  - Proof-of-concept prototype
- Debugging symbol issues with embedded DSOs
  - Existing tools such as `perf` assume ELF header starts at the beginning of an ELF file
  - Cannot map symbol files to prebuilt DSOs that are embedded in the executable image

Executable Image with Embedded JDK Runtime Files

JDK Section In Hermetic Java Image

- Located between the ELF section and JAR section
- Contains JDK files that require page alignmer for start offset (required by `mmap`)
  - `lib/modules`
  - CDS archive
- The start position of the files in the section ar padded to be page alignmed

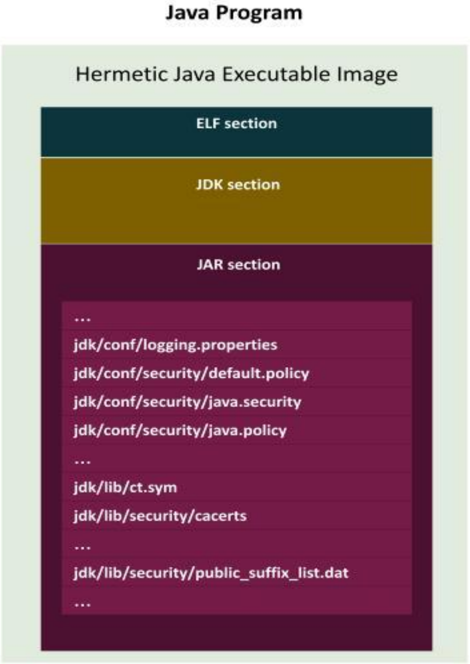


JDK Section (continued)

- JDK/Hotspot is enhanced to access file (hermetic Java executable image) embedded `modules` and CDS archive
- Files in JDK section are unaffected by updating the JAR content
  - Contents cannot be read or extracted by standard `Jar` tool
  - Protected from unexpected modification

JAR Section and JDK Resource Files

- JDK resources files are packaged as regular JAR file entries inside the image JAR section


Java Program
Hermetic Java Executable Image
ELF section
JDK section
JAR section
...
jdk/conf/logging.properties
jdk/conf/security/default.policy
jdk/conf/security/java.security
jdk/conf/security/java.policy
...
jdk/lib/ct.sym
jdk/lib/security/cacerts
...
jdk/lib/security/public_suffix_list.dat
...

java.home

- `System.getProperty(“java.home”)`
  - Traditional Java returns JDK directory path
  - Hermetic Java returns path to the execution image
- A new `JavaHome` class
  - Provide uniform APIs for accessing JDK resources in both conventional and hermetic Java modes
  - Use zip file system provider for accessing hermetic Java image packaged JDK resources

```
Path resource = JavaHome.getJDKResource(...)
```

Java Invocation

- Traditional JAR file name:

  `app.jar`

```
bin/java <JVM options>
-cp app.jar MainClass
<app options>
```

- Hermetic JAR image name:

  `hermeticApp.jar`

```
hermeticApp.jar <JVM
options> run <app
options>
```

Summary

- Hermetic Java provides a package solution with self-contained static image including launcher executable, JDK runtime and Java application
  - Packaged by singlejar
  - Image is an executable JAR file
  - Simplify deployment
- May propose via JEP process
  - Welcome any initial feedback for contributing in OpenJDK