JEP 220: Modular Run-Time Images

Author	Mark Reinhold
Owner	Alan Bateman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	9
JSR	376
Discussion	jigsaw dash dev at openjdk dot java dot net
Effort	L
Duration	L
Blocks	JEP 200: The Modular JDK JEP 261: Module System
Relates to	JEP 162: Prepare for Modularization JEP 201: Modular Source Code JEP 282: jlink: The Java Linker
Reviewed by	Alan Bateman, Alex Buckley, Chris Hegarty, Mandy Chung, Paul Sandoz
Created	2014/10/23 15:05
Updated	2017/09/22 20:16
Issue	8061971

Summary

Restructure the JDK and JRE run-time images to accommodate modules and to improve performance, security, and maintainability. Define a new URI scheme for naming the modules, classes, and resources stored in a run-time image without revealing the internal structure or format of the image. Revise existing specifications as required to accommodate these changes.

Goals

- Adopt a run-time format for stored class and resource files that:
  - Is more time- and space-efficient than the legacy JAR format, which in turn is based upon the ancient ZIP format;
  - Can locate and load class and resource files on a per-module basis;
  - Can store class and resource files from JDK modules and from library and application modules; and
  - Can be extended to accommodate additional kinds of data going forward, such as precomputed JVM data structures and precompiled native code for Java classes.
- Restructure the JDK and JRE run-time images to draw a clear distinction between files that developers, deployers, and end-users can rely upon and, when appropriate, modify, in contrast to files that are internal to the implementation and subject to change without notice.
- Provide supported ways to perform common operations such as, *e.g.*, enumerating all of the classes present in an image, which today require inspecting the internal structure of a run-time image.
- Enable the selective *de-privileging* of JDK classes that today are granted all security permissions but do not actually require those permissions.
- Preserve the existing behavior of well-behaved applications, *i.e.*, applications that do not depend upon internal aspects of JRE and JDK run-time images.

Success Metrics

Modular run-time images equivalent to the JRE, JDK, and Compact Profile images of the immediately-preceding JDK 9 build must not regress on a representative set of startup, static footprint, and dynamic footprint benchmarks.

Non-Goals

- It is not a goal to preserve all aspects of the current run-time image structure.
- It is not a goal to preserve the exact current behavior of all existing APIs.

Motivation

Project Jigsaw aims to design and implement a standard module system for the Java SE Platform and to apply that system to the Platform itself, and to the JDK. Its primary goals are to make implementations of the Platform more easily scalable down to small devices, improve the security and maintainability, enable improved application performance, and provide developers with better tools for programming in the large.

This JEP is the third of four JEPs for Project Jigsaw. The earlier JEP 200 defines the structure of the modular JDK, and JEP 201 reorganizes the JDK source code into modules. A later JEP, 261, introduces the actual module system.

Description

Current run-time image structure

The JDK build system presently produces two types of run-time images: A Java Runtime Environment (JRE), which is a complete implementation of the Java SE Platform, and a Java Development Kit (JDK), which embeds a JRE and includes development tools and libraries. (The three Compact Profile builds are subsets of the JRE.)

The root directory of a JRE image contains two directories, `bin` and `lib`, with the following content:

- The `bin` directory contains essential executable binaries, and in particular the `java` command for launching the run-time system. (On the Windows operating system it also contains the run-time system's dynamically-linked native libraries.)
- The `lib` directory contains a variety of files and subdirectories:
  - Various `.properties` and `.policy` files, most of which may be, though rarely are, edited by developers, deployers, and end users;
  - The `endorsed` directory, which does not exist by default, into which JAR files containing implementations of endorsed standards and standalone technologies may be placed;
  - The `ext` directory, into which JAR files containing extensions or optional packages may be placed;
  - Various implementation-internal data files in assorted binary formats, *e.g.*, fonts, color profiles, and time-zone data;
  - Various JAR files, including `rt.jar`, which contain the run-time system's Java class and resource files.
  - The run-time system's dynamically-linked native libraries on the Linux, macOS, and Solaris operating systems.

A JDK image includes a copy of the JRE in its `jre` subdirectory and contains additional subdirectories:

- The `bin` directory contains command-line development and debugging tools, *e.g.*, `javac`, `javadoc`, and `jconsole`, along with duplicates of the binaries in the `jre/bin` directory for convenience;

- The `demo` and `sample` directories contain demonstration programs and sample code, respectively;
- The `man` directory contains UNIX-style manual pages;
- The `include` directory contains C/C++ header files for use when compiling native code that interfaces directly with the run-time system; and
- The `lib` directory contains various JAR files and other types of files comprising the implementations of the JDK's tools, among them `tools.jar`, which contains the classes of the `javac` compiler.

The root directory of a JDK image, or of a JRE image that is not embedded in a JDK image, also contains various `COPYRIGHT`, `LICENSE` and `README` files and also a `release` file that describes the image in terms of simple key/value property pairs, *e.g.*,

```
JAVA_VERSION="1.9.0"
OS_NAME="Linux"
OS_VERSION="2.6"
OS_ARCH="amd64"
```

New run-time image structure

The present distinction between JRE and JDK images is purely historical, a consequence of an implementation decision made late in the development of the JDK 1.2 release and never revisited. The new image structure eliminates this distinction: A JDK image is simply a run-time image that happens to contain the full set of development tools and other items historically found in the JDK.

A modular run-time image contains the following directories:

- The `bin` directory contains any command-line launchers defined by the modules linked into the image. (On Windows it continues to contain the run-time system's dynamically-linked native libraries.)
- The `conf` directory contains the `.properties`, `.policy`, and other kinds of files intended to be edited by developers, deployers, and end users, which were formerly found in the `lib` directory or subdirectories thereof.
- The `lib` directory on Linux, macOS, and Solaris contains the run-time system's dynamically-linked native libraries, as it does today. These files, named `libjvm.so` or `libjvm.dylib`, may be linked against by programs that embed the run-time system. A few other files in this directory are also intended for external use, including `src.zip` and `jexec`.
- All other files and directories in the `lib` directory must be treated as private implementation details of the run-time system. They are not intended for external use and their names, format, and content are subject to change without notice.
- The `legal` directory contains the legal notices for the modules linked into the image, grouped into one subdirectory per module.
- A full JDK image contains, additionally, the `demo`, `man`, and `include` directories, as it does today. (The `samples` directory was removed by JEP 298.)

The root directory of a modular run-time image also contains the `release` file, which is generated by the build system. To make it easy to tell which modules are present in a run-time image the `release` file includes a new property, `MODULES`, which is a space-separated list of the names of those modules. The list is topologically ordered according to the modules' dependence relationships, so the `java.base` module is always first.

Removed: The endorsed-standards override mechanism

The endorsed-standards override mechanism allowed implementations of newer versions of standards maintained outside of the Java Community Process, or of standalone APIs that are part of the Java SE Platform yet continue to evolve independently, to be installed into a run-time image.

The endorsed-standards mechanism was defined in terms of a path-like system property, `java.endorsed.dirs`, and a default value for that property, `$JAVA_HOME/lib/endorsed`. A JAR file containing a newer implementation of an endorsed standard or standalone API can be installed into a run-time image by placing it in one of the directories named by the system property, or by placing it in the default `lib/endorsed` directory if the system property is not defined. Such JAR files are prepended to the JVM's bootstrap class path at run time, thereby overriding any definitions stored in the run-time system itself.

A modular image is composed of modules rather than JAR files. Going forward, endorsed standards and standalone APIs are supported in modular form only, via the concept of *upgradeable modules*. We have therefore removed the endorsed-standards override mechanism, including the `java.endorsed.dirs` system property and the `lib/endorsed` directory. To help identify any existing uses of this mechanism the compiler and the launcher now fail if this system property is set, or if the `lib/endorsed` directory exists.

Removed: The extension mechanism

The extension mechanism allowed JAR files containing APIs that extend the Java SE Platform to be installed into a run-time image so that their contents are visible to every application that is compiled with or runs on that image.

The mechanism was defined in terms of a path-like system property, `java.ext.dirs`, and a default value for that property composed of `$JAVA_HOME/lib/ext` and a platform-specific system-wide directory (*e.g*, `/usr/java/packages/lib/ext` on Linux). It worked in much the same manner as the endorsed-standards mechanism except that JAR files placed in an extension directory were loaded by the run-time environment's *extension class loader*, which is a child of the bootstrap class loader and the parent of the *system class loader*, which actually loads the application to be run from the class path. Extension classes therefore could not override the JDK classes loaded by the bootstrap loader but they were loaded in preference to classes defined by the system loader and its descendants.

The extension mechanism was introduced in JDK 1.2, which was released in 1998, but in modern times we have seen little evidence of its use. This is not surprising, since most Java applications today place the libraries that they need directly on the class path rather than require that those libraries be installed as extensions of the run-time system.

It is technically possible, though awkward, to continue to support the extension mechanism in the modular JDK. To simplify both the Java SE Platform and the JDK we have removed the extension mechanism, including the `java.ext.dirs` system property and the `lib/ext` directory. To help identify any existing uses of this mechanism the compiler and the launcher now fail if this system property is set, or if the `lib/ext` directory exists. The compiler and the launcher ignore the platform-specific system-wide extension directory by default, but if the `-XX:+CheckEndorsedAndExtDirs` command-line option is specified then they fail if that directory exists and is not empty.

Several features associated with the extension mechanism were retained, since they are useful in their own right:

- The `Class-Path` manifest attribute, which specifies JAR files required by another JAR file;
- The `{Specification,Implementation}-{Title,Version,Vendor}` manifest attributes, which specify package and JAR-file version information;
- The `Sealed` manifest attribute, which seals a package or a JAR file; and
- The extension class loader itself, though it is now known as the *platform class loader*.

Removed: `rt.jar` and `tools.jar`

The class and resource files previously stored in `lib/rt.jar`, `lib/tools.jar`, `lib/dt.jar`, and various other internal JAR files are now stored in a more efficient format in implementation-specific files in the `lib` directory. The format of these files is not specified and is subject to change without notice.

The removal of `rt.jar` and similar files leads to three distinct problems:

1. Existing standard APIs such as the `ClassLoader::getSystemResource` method return `URL` objects to name class and resource files inside the run-time image. For example, when run on JDK 8 the code

   ```
   ClassLoader.getSystemResource("java/lang/Class.class");
   ```

   returns a `jar` URL of the form

   ```
   jar:file:/usr/local/jdk8/jre/lib/rt.jar!/java/lang/Class.class
   ```

   which, as can be seen, embeds a `file` URL to name the actual JAR file within the run-time image. The `getContent` method of that URL object can be used to retrieve the content of the class file, via the built-in protocol handler for the `jar` URL scheme.

   A modular image does not contain any JAR files, so URLs of the above form make no sense. The specifications of `getSystemResource` and related methods, fortunately, do not require the URL objects returned by these methods actually to use the JAR scheme. They do, however, require that it be possible to load the content of a stored class or resource file via these URL objects.

2. The `java.security.CodeSource` API and security-policy files use URLs to name the locations of code bases that are to be granted specified permissions. Components of the run-time system that require specific permissions are currently identified in the `lib/security/java.policy` file via `file` URLs. The elliptic-curve cryptography provider, *e.g.*, is identified as

   ```
   file:${java.home}/lib/ext/sunec.jar
   ```

   which, obviously, has no meaning in a modular image.

3. IDEs and other kinds of development tools require the ability to enumerate the class and resource files stored in a run-time image, and to read their contents. Today they often do this directly by opening and reading `rt.jar` and similar files. This is, of course, not possible with a modular image.

New URI scheme for naming stored modules, classes, and resources

To address the above three problems a new URL scheme, `jrt`, can be used to name the modules, classes, and resources stored in a run-time image without revealing the internal structure or format of the image.

A `jrt` URL is a hierarchical URI, per RFC 3986, with the syntax

```
jrt:/[$MODULE[/$PATH]]
```

where `$MODULE` is an optional module name and `$PATH`, if present, is the path to a specific class or resource file within that module. The meaning of a `jrt` URL depends upon its structure:

- `jrt:/$MODULE/$PATH` refers to the specific class or resource file named `$PATH` within the given `$MODULE`.
- `jrt:/$MODULE` refers to all of the class and resource files in the module `$MODULE`.
- `jrt:/` refers to the entire collection of class and resource files stored in the current run-time image.

These three forms of `jrt` URLs address the above problems as follows:

1. APIs that presently return `jar` URLs now return `jrt` URLs. The above invocation of `ClassLoader::getSystemResource`, *e.g.*, now returns the URL

   ```
   jrt:/java.base/java/lang/Class.class
   ```

   A built-in protocol handler for the `jrt` scheme ensures that the `getContent` method of such URL objects retrieves the content of the named class or resource file.

2. Security-policy files and other uses of the `CodeSource` API can use `jrt` URLs to name specific modules for the purpose of granting permissions. The elliptic-curve cryptography provider, *e.g.*, can now be identified by the `jrt` URL

   ```
   jrt:/jdk.crypto.ec
   ```

   Other modules that are currently granted all permissions but do not actually require them can trivially be de-privileged, *i.e.*, given precisely the permissions they require.

3. A built-in NIO FileSystem provider for the `jrt` URL scheme ensures that development tools can enumerate and read the class and resource files in a run-time image by loading the FileSystem named by the URL `jrt:/`, as follows:

   ```
   FileSystem fs = FileSystems.getFileSystem(URI.create("jrt:/"));
   byte[] jlo = Files.readAllBytes(fs.getPath("modules", "java.base",
                                              "java/lang/Object.class"));
   ```

   The top-level `modules` directory in this filesystem contains one subdirectory for each module in the image. The top-level packages directory contains one subdirectory for each package in the image, and that subdirectory contains a symbolic link to the subdirectory for the module that defines that package.

   For tools that support the development of code for JDK 9 but which themselves run on JDK 8, a copy of this filesystem provider suitable for use on JDK 8 is placed in the `lib` directory of JDK 9 run-time images, in a file named `jrt-fs.jar`.

(The `jrt` URL protocol handler does not return any content for URLs of the second and third forms.)

Build-system changes

The build system produces the new run-time image format described above, using the Java linker (JEP 282).

We took the opportunity here, finally, to rename the `images/j2sdk-image` and `images/j2re-image` directories to `images/jdk` and `images/jre`, respectively.

Minor specification changes

JEP 162, implemented in JDK 8, made a number of changes to prepare the Java SE Platform and the JDK for the modularization work described here and in related JEPs. Among those changes were the removal of normative specification statements that require certain configuration files to be looked up in the `lib` directory of run-time images, since those files are now in the `conf` directory. Most of the SE-only APIs with such statements were so revised as part of Java SE 8, but some APIs shared across the Java SE and EE Platforms still contain such statements:

- `javax.xml.stream.XMLInputFactory` specifies `${java.home}/lib/stax.properties` (JSR 173).
- `javax.xml.ws.spi.Provider` specifies `${java.home}/lib/jaxws.properties` (JSR 224).
- `javax.xml.soap.MessageFactory`, and related classes, specify `${java.home}/lib/jaxm.properties` (JSR 67).

In Java SE 9, these statements no longer mandate the `lib` directory.

Testing

Some existing tests made direct use of run-time image internals (*e.g.*, `rt.jar`) or refer to system properties (*e.g.*, `java.ext.dirs`) that no longer exist. These tests have been fixed.

Early-access builds containing the changes described here were available throughout the development of the module system. Members of the Java community were strongly encouraged to test their tools, libraries, and applications against these builds to help identify compatibility issues.

Risks and Assumptions

The central risks of this proposal are ones of compatibility, summarized as follows:

- A JDK image no longer contains a `jre` subdirectory, as noted above. Existing code that assumes the existence of that directory might not work correctly.
- JDK and JRE images no longer contain the files `lib/rt.jar`, `lib/tools.jar`, `lib/dt.jar`, and other internal JAR files, as noted above. Existing code that assumes the existence of these files might not work correctly.
- The system properties `java.endorsed.dirs` and `java.ext.dirs` are no longer defined, as noted above. Existing code that assumes these properties to have non-`null` values might not work correctly.
- The run-time system's dynamically-linked native libraries are always in the `lib` directory, except on Windows; in Linux and Solaris builds they were previously placed in the `lib/$ARCH` subdirectory. That was a vestigial remnant of images that could support multiple CPU architectures, which is no longer a requirement.
- The `src.zip` file is now in the `lib` directory rather than the top-level directory, and this file now includes one directory for each module in the image. IDEs and other tools that read this file will need to be updated.
- Existing standard APIs that return `URL` objects to name class and resource files inside the run-time image now return `jrt` URLs, as noted above. Existing code that expects these APIs to return `jar` URLs might not work correctly.
- The internal system property `sun.boot.class.path` has been removed. Existing code that depends upon this property might not work correctly.
- Class and resource files in a JDK image that were previously found in `lib/tools.jar`, and visible only when that file was added to the class path, are now visible via the system class loader or, in some cases, the bootstrap class loader. The modules containing these files are not mentioned in the application class path, *i.e.*, in the value of the system property `java.class.path`.
- Class and resource files previously found in `lib/dt.jar` and visible only when that file was added to the class path are now visible via the bootstrap class loader and present in both the JRE and the JDK.
- Configuration files previously found in the `lib` directory, including the security policy file, are now located in the `conf` directory. Existing code that examines or manipulates these files may need to be updated.
- The defining class loader of the types in some existing packages has changed. Existing code that makes assumptions about the class loaders of these types might not work correctly. The specific changes are enumerated in JEP 261. Some of these changes are a consequence of the way in which components that contain both APIs and tools were modularized. The

classes of such a component were historically split between `rt.jar` and `tools.jar`, but now all such classes are in a single module.

- The `bin` directory in a JRE image contains a few commands that were previously found only in JDK images, namely `appletviewer`, `idlj`, `jrunscript`, and `jstatd`. As with the previous item, these changes are a consequence of the way in which components that contain both APIs and tools were modularized.

Dependences

This JEP is the third of four JEPs for Project Jigsaw. It depends upon JEP 201, which reorganized the JDK source code into modules and upgraded the build system to compile modules. It also depends upon earlier preparatory work done in JEP 162, implemented in JDK 8.