JEP 220: Modular Run-Time Images

JEP 220：模块化运行时映像

Author 作者	Mark Reinhold 马克·莱因霍尔德
Owner 所有者	Alan Bateman 艾伦·贝特曼
Type 类型	Feature 特征
Scope 范围	SE
Status 地位	Closed / Delivered 已关闭 / 已交付
Release 释放	9
JSR	376
Discussion 讨论	jigsaw dash dev at openjdk dot java dot net
Effort 努力	L
Duration 期间	L
Blocks 块	JEP 200: The Modular JDK JEP 200：模块化 JDK
Relates to 涉及	JEP 261: Module System JEP 261： 模块系统 JEP 162: Prepare for Modularization JEP 162：为模块化做准备 JEP 201: Modular Source Code JEP 201：模块化源代码 JEP 282: jlink: The Java Linker JEP 282：jlink：Java 链接器
Reviewed by 校订者	Alan Bateman, Alex Buckley, Chris Hegarty, Mandy Chung, Paul Sandoz 艾伦·贝特曼、亚历克斯·巴克利、克里斯·赫加蒂、钟曼迪、保罗·山德士
Created 创建	2014/10/23 15:05
Updated 更新	2017/09/22 20:16
Issue 问题	8061971

Summary 总结

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.

重组 JDK 和 JRE 运行时映像以适应模块并提高性能、安全性和可维护性。定义一个新的 URI 方案，用于命名存储在运行时映像中的模块、类和资源，而不显示映像的内部结构或格式。根据需要修改现有规范以适应这些更改。

Goals 目标

- Adopt a run-time format for stored class and resource files that:

  对存储的类和资源文件采用运行时格式，该格式：

  - Is more time- and space-efficient than the legacy JAR format, which in turn is based upon the ancient ZIP format;

    比传统的 JAR 格式更节省时间和空间，而传统的 JAR 格式又基于古老的 ZIP 格式；

JDK (..., 23, 24, 25)

JDK (..., 23、 24、 25)

JDK Updates JDK 更新
JMC 江铃汽车
Jigsaw 拼图
Kona 科纳
Kulla 库拉
Lanai 拉奈岛
Leyden 莱 顿
Lilliput 小人 国
Locale Enhancement 区域设置增强
Loom 织布机
Memory Model Update 内存模型更新
Metropolis 都市
Multi-Language VM 多语言 VM
Nashorn 纳斯霍恩
New I/O 新 I/O
OpenJFX OpenJFX 公司
Panama 巴拿马
Penrose 彭罗斯
Port: AArch32 端口: AArch32
Port: AArch64 端口: AArch64
Port: BSD 端口: BSD
Port: Haiku 港口: Haiku
Port: Mac OS X 端口: Mac OS X
Port: MIPS 端口: MIPS
Port: Mobile 端口: 移动
Port: PowerPC/AIX 端口: PowerPC/AIX
Port: RISC-V 端口: RISC-V
Port: s390x 端口: s390x
SCTP SCTP 系列
Shenandoah 谢南多厄
Skara 人群
Sumatra 苏门答腊岛
Tsan TSA 餐厅
Valhalla 瓦尔哈拉
Verona 维罗纳
VisualVM 可视化虚拟机
Wakefield 维克菲尔德
Zero 零
ZGC 中关村

- Can locate and load class and resource files on a per-module basis;

  可以按模块查找和加载类和资源文件;

- Can store class and resource files from JDK modules and from library and application modules; and

  可以存储 JDK 模块以及库和应用程序模块中的类和资源文件;和

- Can be extended to accommodate additional kinds of data going forward, such as precomputed JVM data structures and precompiled native code for Java classes.

  可以扩展以适应将来的其他类型的数据，例如预计算的 JVM 数据结构和 Java 类的预编译本机代码。

- 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.

  重新构建 JDK 和 JRE 运行时映像，以明确区分开发人员、部署人员和最终用户可以依赖并在适当时进行修改的文件，这与实现内部的文件形成鲜明对比，这些文件可能会更改，恕不另行通知。

- 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.

  提供支持的方法来执行常见*作，例如*，枚举映像中存在的所有类，这些作现在需要检查运行时映像的内部结构。

- Enable the selective *de-privileging* of JDK classes that today are granted all security permissions but do not actually require those permissions.

  启用对 JDK 类的选择性*取消特权*，这些类目前被授予了所有安全权限，但实际上并不需要这些权限。

- 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.

  保留行为良好的应用程序的现有行为， *即*不依赖于 JRE 和 JDK 运行时映像的内部方面的应用程序。

Success Metrics 成功指标

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.

等效于 JRE、JDK 和 紧接在前面的 JDK 9 构建的 Compact Profile 映像不得在一组具有代表性的启动、静态占用空间和动态占用空间基准上回归。

Non-Goals 非目标

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

  保留当前运行时映像结构的所有方面并不是一个目标。

- It is not a goal to preserve the exact current behavior of all existing APIs.

  保留所有现有 API 的确切当前行为并不是目标。

Motivation 赋予动机

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.

Jigsaw 项目旨在为 Java SE 平台设计和实现一个标准模块系统，并将该系统应用于平台本身和 JDK。其主要目标是使平台的实现更容易扩展到小型设备，提高安全性和可维护性，提高应用程序性能，并为开发人员提供更好的大型编程工具。

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.

此 JEP 是 Project Jigsaw 的四个 JEP 中的第三个。 较早的 JEP 200 定义了模块化 JDK 的结构，并且 JEP 201 将 JDK 源代码重新组织为模块。后来的 JEP 261 介绍了实际的模块系统。

Description 描述

Current run-time image structure

当前运行时映像结构

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.)

JDK 构建系统目前生成两种类型的运行时映像：一个 Java 运行时环境 （JRE） 是 Java SE 平台和 Java 开发工具包 （JDK），其中嵌入了 JRE，并包含开发工具和库。 （这三个 Compact Profile 构建是 JRE 的子集。

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

JRE 镜像的根目录包含两个目录，`bin` 和 `lib` 中，内容如下：

- 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.)

  `bin` 目录包含基本的可执行二进制文件，特别是用于启动运行时系统的 `java` 命令。（在 Windows 作系统上，它还包含运行时系统的动态链接本机库。

- The `lib` directory contains a variety of files and subdirectories:

  `lib` 目录包含各种文件和子目录：

  - Various `.properties` and `.policy` files, most of which may be, though rarely are, edited by developers, deployers, and end users;

    各种 `.properties` 和 `.policy` 文件，其中大多数可能由开发人员、部署人员和最终用户编辑，但很少编辑;

  - The `endorsed` directory, which does not exist by default, into which JAR files containing implementations of endorsed standards and standalone technologies may be placed;

    背书目录（默认情况下不存在）放入 哪些 JAR 文件包含 可以放置认可的标准和独立技术 ;

  - The `ext` directory, into which JAR files containing extensions or optional packages may be placed;

ext 目录，其中包含 可以放置扩展或可选包；

- Various implementation-internal data files in assorted binary formats, *e.g.*, fonts, color profiles, and time-zone data;

  各种二进制格式的各种实现内部数据文件， *例如*字体、颜色配置文件和时区数据;

- Various JAR files, including `rt.jar`, which contain the run-time system's Java class and resource files.

  各种 JAR 文件，包括 `rt.jar`，其中包含运行时系统的 Java 类和资源文件。

- The run-time system's dynamically-linked native libraries on the Linux, macOS, and Solaris operating systems.

  运行时系统在 Linux、macOS 和 Solaris 作系统上的动态链接本机库。

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

JDK 映像在其 `jre` 子目录中包含 JRE 的副本，并包含其他子目录：

- 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;

  为方便起见，`bin` 目录包含命令行开发和调试工具， *例如* `javac`、`javadoc` 和 `jconsole`，以及 `jre/bin` 目录中的二进制文件副本;

- The `demo` and `sample` directories contain demonstration programs and sample code, respectively;

  `demo` 和 `sample` 目录分别包含演示程序和示例代码;

- The `man` directory contains UNIX-style manual pages;

  `man` 目录包含 UNIX 样式的手册页;

- The `include` directory contains C/C++ header files for use when compiling native code that interfaces directly with the run-time system; and

  `include` 目录包含 C/C++ 头文件，用于编译直接与运行时系统交互的本机代码;和

- 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.

  `lib` 目录包含各种 JAR 文件和其他类型的 文件中包含 JDK 工具的实现，其中包括 `tools.jar`，其中包含 `javac` 编译器的类。

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.*,

JDK 映像或未嵌入 JDK 映像中的 JRE 映像的根目录还包含各种 `COPYRIGHT`、`LICENSE` 和 `README` 文件，以及一个发布文件，该文件根据简单的键/值属性对描述图像， *例如,*

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

New run-time image structure

新的运行时映像结构

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.

目前 JRE 和 JDK 映像之间的区别纯粹是历史性的，这是在 JDK 1.2 版本开发后期做出的实现决策的结果，从未重新审视。新的映像结构消除了这种区别：JDK 映像只是一个运行时映像，它恰好包含一整套开发工具和 JDK 中历史上的其他项目。

A modular run-time image contains the following directories:

模块化运行时映像包含以下目录：

- 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.)

  `bin` 目录包含由链接到映像中的模块定义的任何命令行启动器。（在 Windows 上，它继续包含运行时系统的动态链接本机库。

- 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.

  `conf` 目录包含 `.properties`、`.policy` 和供开发人员、部署人员和最终用户编辑的其他类型的文件，这些文件以前位于 `lib` 目录或其子目录中。

- 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`.

  Linux、macOS 和 Solaris 上的 `lib` 目录包含运行时系统的动态链接本机库，就像现在一样。这些文件名为 `libjvm.so` 或 `libjvm.dylib`，可能由嵌入运行时系统的程序链接。此目录中的其他一些文件也供外部使用，包括 `src.zip` 和 `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.

  `lib` 目录中的所有其他文件和目录必须被视为运行时系统的私有实现详细信息。它们不供外部使用，其名称、格式和内容如有更改，恕不另行通知。

- The `legal` directory contains the legal notices for the modules linked into the image, grouped into one subdirectory per module.

  法律目录包含链接到映像中的模块的法律声明，每个模块分组到一个子目录中。

- A full JDK image contains, additionally, the `demo`, `man`, and `include` directories, as it does today. (The `samples` directory was removed by JEP 298.)

  完整的 JDK 映像还包含 `demo`、`man` 和 `include` 目录，就像现在一样。（`samples` 目录已被 JEP 298 删除。

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.

模块化运行时镜像的根目录还包含 release 文件，该文件由构建系统生成。为了便于区分运行时映像中存在哪些模块， 发布文件包含一个新属性 MODULES， 该属性是这些模块名称的空格分隔列表。该列表根据模块的依赖关系进行拓扑排序，因此 java.base 模块始终排在第一位。

Removed: The endorsed-standards override mechanism

已删除：endorsed-standards 覆盖机制

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.

认可标准覆盖机制允许在 Java 社区进程之外维护的较新版本的标准实现，或者将作为 Java SE 平台的一部分但继续独立发展的独立 API 的实现安装到运行时映像中。

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.

认可标准机制是根据路径式系统属性 java.endorsed.dirs 和该属性的默认值 $JAVA_HOME/lib/endorsed 定义的。包含认可的标准或独立 API 的较新实现的 JAR 文件可以安装到运行时映像中，方法是将其放置在 system 属性命名的目录之一中，或者将其放置在默认的 lib/endorsed 中 directory （如果未定义 system 属性）。 此类 JAR 文件是 在运行时附加到 JVM 的引导类路径，从而 覆盖运行时系统本身中存储的任何定义。

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.

模块化映像由模块组成，而不是 JAR 文件。展望未来，通过*可升级模块*的概念，仅以模块化形式支持认可的标准和独立 API。因此，我们删除了 endorsed-standards 覆盖机制，包括 java.endorsed.dirs 系统属性和 lib/endorsed 目录。 为了帮助确定此机制的任何现有用途， 编译器，如果设置了此系统属性，或者如果 lib/endorsed 目录存在。

Removed: The extension mechanism

删除：扩展机构

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.

扩展机制允许将包含扩展 Java SE 平台的 API 的 JAR 文件安装到运行时映像中，以便使用该映像编译或在该映像上运行的每个应用程序都能看到其内容。

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.

该机制是根据路径式系统属性 `java.ext.dirs` 定义的，该属性的默认值由 `$JAVA_HOME/lib/ext` 和特定于平台的系统范围目录（ *例如* ，Linux 上的 `/usr/java/packages/lib/ext`）组成。它的工作方式与认可标准机制大致相同，只是放置在扩展目录中的 JAR 文件由运行时环境的*扩展类加载器加载，该扩展类加载器*是引导类加载器的子类和*系统类加载*器的父类，它实际上加载要从类路径运行的应用程序。因此，扩展类无法覆盖引导加载程序加载的 JDK 类，但它们的加载优先于系统加载程序及其后代定义的类。

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.

扩展机制是在 1998 年发布的 JDK 1.2 中引入的，但在现代，我们几乎没有看到使用它的证据。这并不奇怪，因为现在大多数 Java 应用程序都将它们需要的库直接放在类路径上，而不是要求将这些库作为运行时系统的扩展安装。

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.

从技术上讲，继续支持模块化 JDK 中的扩展机制是可能的，尽管这很尴尬。为了简化 Java SE 平台和 JDK，我们删除了扩展机制，包括 `java.ext.dirs` 系统属性和 `lib/ext` 目录中。为了帮助识别此机制的任何现有用途，如果设置了此系统属性，或者 `lib/ext` 目录存在，则编译器和 Launcher 现在会失败。默认情况下，编译器和 Launcher 会忽略特定于平台的系统范围扩展目录，但如果指定了 `-XX：+CheckEndorsedAndExtDirs` 命令行选项，则如果该目录存在且不为空，则它们将失败。

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

保留了与扩展机制相关的几个功能，因为它们本身就很有用：

- The `Class-Path` manifest attribute, which specifies JAR files required by another JAR file;

  `Class-Path` 清单属性，用于指定另一个 JAR 文件所需的 JAR 文件;

- The `{Specification,Implementation}-{Title,Version,Vendor}` manifest attributes, which specify package and JAR-file version information;

  manifest `{Specification,Implementation}-{Title,Version,Vendor}` 属性，用于指定软件包和 JAR 文件版本信息;

- The `Sealed` manifest attribute, which seals a package or a JAR file; and

  `Sealed` 清单属性，用于密封包或 JAR 文件;和

- The extension class loader itself, though it is now known as the *platform* class loader.

  扩展类加载器本身，尽管它现在被称为 *platform* 类加载器 。

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

已删除：`rt.jar` 和 `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.

以前存储在 `lib/rt.jar` 中的类和资源文件， `lib/tools.jar`、`lib/dt.jar` 和各种其他内部 JAR 文件现在以更高效的格式存储在 `lib` 目录中特定于实现的文件中。这些文件的格式未指定，如有更改，恕不另行通知。

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

删除 `rt.jar` 和类似文件会导致三个不同的问题：

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

   现有的标准 API，例如 `ClassLoader::getSystemResource` 方法返回 `URL` 对象来命名运行时映像中的类和资源文件。例如，在 JDK 8 上运行时，代码

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

   returns a `jar` URL of the form

   返回格式为

   ```
   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.

   可以看出，它嵌入了一个文件 URL，用于在运行时映像中命名实际的 JAR 文件。`getContent` 方法可用于通过 `jar` URL 方案的内置协议处理程序检索类文件的内容。

   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.

   模块化映像不包含任何 JAR 文件，因此上述形式的 URL 没有意义。`getSystemResource` 的规范 和相关方法，幸运的是，不需要这些方法返回的 URL 对象实际上使用 JAR 方案。但是，它们确实要求能够通过这些 URL 对象加载存储的类或资源文件的内容。

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

   `java.security.CodeSource` API 和 security-policy 文件使用 URL 来命名要被授予指定权限的代码库的位置。运行时系统中需要特定权限的组件当前通过 `file` URL 在 `lib/security/java.policy` 文件中标识。 *例如* ，椭圆曲线密码学提供程序标识为

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

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

显然，这在模块化图像中没有意义。

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.

   IDE 和其他类型的开发工具需要能够枚举存储在运行时映像中的类和资源文件，并读取其内容。今天，他们通常通过打开和读取 `rt.jar` 和类似文件来直接执行此作。当然，这对于模块化映像是不可能的。

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

用于命名存储的模块、类和资源的新 URI 方案

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.

为了解决上述三个问题，可以使用新的 URL 方案 `jrt` 来命名存储在运行时图像中的模块、类和资源，而无需透露图像的内部结构或格式。

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

`jrt` URL 是符合 RFC 3986 的分层 URI，其语法为

```
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:

其中 `$MODULE` 是可选的模块名称，`$PATH` （如果存在）是该模块中特定类或资源文件的路径。`jrt` URL 的含义取决于其结构：

- `jrt:/$MODULE/$PATH` refers to the specific class or resource file named `$PATH` within the given `$MODULE`.

  `jrt:/$MODULE/$PATH` 是指给定 `$MODULE` 中名为 `$PATH` 的特定类或资源文件。

- `jrt:/$MODULE` refers to all of the class and resource files in the module `$MODULE`.

  `jrt:/$MODULE` 引用模块 `$MODULE` 中的所有类和资源文件。

- `jrt:/` refers to the entire collection of class and resource files stored in the current run-time image.

  `jrt:/` 是指存储在当前运行时映像中的类和资源文件的整个集合。

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

这三种形式的 `jrt` URL 解决了上述问题，如下所示：

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

   当前返回 `jar` URL 的 API 现在返回 `jrt` URL。 *例如* ， 上述对 `ClassLoader::getSystemResource` ， 的调用现在返回 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.

`jrt` 方案的内置协议处理程序可确保 `getContent` 方法检索命名类或资源文件的内容。

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

   安全策略文件和 `CodeSource` API 的其他用途可以使用 捷睿用于命名特定模块以授予权限的 URL。 *例如* ，椭圆曲线加密提供程序现在可以通过 `jrt` URL 来识别

   ```
   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.

   当前被授予所有权限但实际上并不需要它们的其他模块可以很容易地被取消特权，*即*恰好获得它们所需的权限。

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:

   用于 `jrt` URL 的内置 NIO FileSystem 提供程序 方案确保开发工具可以枚举和读取 类和资源文件，方法是在运行时映像中加载 由 URL `jrt:/` 命名的 FileSystem，如下所示：

   ```
   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.

   此文件系统中的顶级 `modules` 目录包含映像中每个模块的一个子目录。顶级软件包 directory 包含镜像中每个包的一个子目录， 并且该子目录包含指向该子目录的符号链接 对于定义该软件包的模块。

   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`.

   对于支持 JDK 9 代码开发但本身在 JDK 8 上运行的工具，此文件系统提供程序的副本适合在 JDK 8 上使用，将放在 `lib` 中 目录中的 JDK 9 运行时映像的 `jrt-fs.jar`。

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

（`jrt` URL 协议处理程序不返回第二种和第三种形式的 URL 的任何内容。

Build-system changes 构建系统更改

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

构建系统使用 Java 链接器 （JEP 282） 生成上述新的运行时映像格式。

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

我们终于借此机会将 images/j2sdk-image 重命名为 和 images/j2re-image 目录分别设置为 images/jdk 和 images/jre。

Minor specification changes

微小的规格更改

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:

在 JDK 8 中实现的 JEP 162 进行了大量更改，以便为 Java SE 平台和 JDK 做好准备，以便在此处和相关 JEP 中描述的模块化工作。这些更改包括删除规范性规范语句，这些语句要求在运行时映像的 `lib` 目录中查找某些配置文件，因为这些文件现在位于 `conf` 目录中。作为 Java SE 8 的一部分，大多数具有此类声明的仅限 SE 的 API 都进行了修订，但在 Java SE 和 EE 平台之间共享的一些 API 仍然包含此类声明：

- `javax.xml.stream.XMLInputFactory` specifies `${java.home}/lib/stax.properties` (JSR 173).

  `javax.xml.stream.XMLInputFactory` 指定 `${java.home}/lib/stax.properties` （JSR 173） 的

- `javax.xml.ws.spi.Provider` specifies `${java.home}/lib/jaxws.properties` (JSR 224).

  `javax.xml.ws.spi.Provider` 指定 `${java.home}/lib/jaxws.properties` （JSR 224） 的。

- `javax.xml.soap.MessageFactory`, and related classes, specify `${java.home}/lib/jaxm.properties` (JSR 67).

  `javax.xml.soap.MessageFactory` 和相关类中，指定 `${java.home}/lib/jaxm.properties` （JSR 67） 的

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

在 Java SE 9 中，这些语句不再强制要求 `lib` 目录。

Testing 测试

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.

一些现有的测试直接使用了运行时映像内部（ *例如* `rt.jar`） 或引用不再存在的系统属性（ *例如* `java.ext.dirs`）。这些测试已修复。

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.

包含此处描述的更改的早期访问版本在整个模块系统开发过程中可用。强烈建议 Java 社区的成员针对这些版本测试他们的工具、库和应用程序，以帮助识别兼容性问题。

Risks and Assumptions 风险和假设

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

该提案的主要风险是兼容性风险，总结如下：

- 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.

  如上所述，JDK 映像不再包含 `jre` 子目录。假定存在该目录的现有代码可能无法正常工作。

- 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.

  JDK 和 JRE 镜像不再包含文件 `lib/rt.jar`、 `lib/tools.jar`、`lib/dt.jar` 和其他内部 JAR 文件，如上所述。假定存在这些文件的现有代码可能无法正常工作。

- 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.

  如上所述，系统属性 `java.endorsed.dirs` 和 `java.ext.dirs` 不再定义。假定这些属性具有非 `null` 值的现有代码可能无法正常工作。

- 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.

  运行时系统的动态链接本机库始终位于 `lib` 目录中，Windows 除外;在 Linux 和 Solaris 版本中，它们以前放在 `lib/$ARCH` 子目录中。这是可以支持多个 CPU 架构的映像的残余，这不再是必需的。

- 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.

  `src.zip` 文件现在位于 `lib` 目录而不是顶级目录中，并且此文件现在包含映像中每个模块的一个目录。读取此文件的 IDE 和其他工具将需要更新。

- 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.

  如上所述，将 URL 对象返回给运行时映像中的名称类和资源文件的现有标准 API 现在返回 `jrt` URL。期望这些 API 返回 `jar` 的现有代码 URL 可能无法正常工作。

- The internal system property `sun.boot.class.path` has been removed. Existing code that depends upon this property might not work correctly.

  已删除内部系统属性 `sun.boot.class.path`。依赖于此属性的现有代码可能无法正常工作。

- 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`.

  JDK 映像中的类和资源文件，这些文件以前在 `lib/tools.jar`，并且仅在该文件添加到类路径时可见，现在可以通过系统类加载器或在某些情况下通过 bootstrap 类加载器查看。包含这些文件的模块未在应用程序类路径中提及， *即*在系统属性 `java.class.path` 的值中。

- 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.

  以前在 `lib/dt.jar` 中找到且仅在该文件添加到类路径时可见的类和资源文件现在通过引导类加载器可见，并且同时存在于 JRE 和 JDK 中。

- 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.

  以前在 `lib` 目录中找到的配置文件（包括安全策略文件）现在位于 `conf` 中 目录。检查或作这些文件的现有代码 可能需要更新。

- 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.

  某些现有包中类型的定义类加载器已更改。对这些类型的类加载器做出假设的现有代码可能无法正常工作。JEP 261 中列举了具体更改。其中一些更改是包含 API 和工具的组件模块化方式的结果。这种组件的类历史上分为 `rt.jar` 和 `tools.jar`，但现在所有这些类都位于单个模块中。

- 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.

  JRE 映像中的 `bin` 目录包含一些以前只能在 JDK 映像中找到的命令，即 `appletviewer`、`idlj`、 `jrunscript` 和 `jstatd`.与前一项一样，这些更改是包含 API 和工具的组件模块化方式的结果。

Dependences 依赖项

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.

此 JEP 是 Project Jigsaw 的四个 JEP 中的第三个。它依赖于 JEP 201，JEP 201 将 JDK 源码重新组织成模块，并升级构建系统以编译模块。它还依赖于在 JEP 162 中完成的早期准备工作，并在 JDK 8 中实现。