JEP 261: Module System JEP 261： 模块系统

Authors 作者	Alan Bateman, Alex Buckley, Jonathan Gibbons, Mark Reinhold 艾伦·贝特曼、亚历克斯·巴克利、乔纳森·吉本斯、马克·莱因霍尔德
Owner 所有者	Mark Reinhold 马克·莱因霍尔德
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	XL
Duration 期间	L
Blocks 块	JEP 200: The Modular JDK JEP 200：模块化 JDK JEP 282: jlink: The Java Linker JEP 282：jlink：Java 链接器
Depends 取决于	JEP 220: Modular Run-Time Images JEP 220：模块化运行时映像 JEP 260: Encapsulate Most Internal APIs JEP 260：封装大多数内部 API
Relates to 涉及	JEP 396: Strongly Encapsulate JDK Internals by Default JEP 396：默认强封装 JDK 内部 JEP 403: Strongly Encapsulate JDK Internals JEP 403：强封装 JDK 内部
Reviewed by 校订者	Alan Bateman, Alex Buckley, Chris Hegarty, Jonathan Gibbons, Mandy Chung, Paul Sandoz Alan Bateman， Alex Buckley， Chris Hegarty， Jonathan Gibbons， Mandy Chung， Paul Sandoz
Endorsed by 通过	Brian Goetz 布莱恩·戈茨
Created 创建	2014/10/23 15:05
Updated 更新	2024/08/19 18:30
Issue 问题	8061972

Summary 总结

Implement the Java Platform Module System, as specified by JSR 376, together with related JDK-specific changes and enhancements.

实现 Java 平台模块系统，如 JSR 376 以及相关的特定于 JDK 的更改和增强功能。

Description 描述

The Java Platform Module System (JSR 376) specifies changes and extensions to the Java programming language, the Java virtual machine, and the standard Java APIs. This JEP implements that specification. As a consequence, the `javac` compiler, the HotSpot virtual machine, and the run-time libraries implement modules as a fundamental new kind of Java program component and provide for the reliable configuration and strong encapsulation of modules in all phases of development.

Java 平台模块系统 （JSR 376） 指定了对 Java 编程语言、Java 虚拟机和标准 Java API 的更改和扩展。此 JEP 实现了该规范。因此，`javac` 编译器、HotSpot 虚拟机和运行时库将模块实现为一种基本的新型 Java 程序组件，并在开发的所有阶段提供模块的可靠配置和强封装。

This JEP also changes, extends, and adds JDK-specific tools and APIs, which are outside the scope of the JSR, that are related to compilation, linking, and execution. Related changes to other tools and APIs, *e.g.*, the `javadoc` tool and the Doclet API, are the subject of separate JEPs.

此 JEP 还更改、扩展和添加特定于 JDK 的工具和 API，这些工具和 API 不在 JSR 的范围内，与编译、链接和执行相关。对其他工具和 API（ *例如* `javadoc` 工具和 Doclet API）的相关更改是单独 JEP 的主题。

This JEP assumes that the reader is familiar with the latest *State of the Module System* document and also the other Project Jigsaw JEPs:

此 JEP 假定读者熟悉最新的 *Module System* 文档的状态，以及 拼图项目 JEPs：

- 200: The Modular JDK 第 200 章：模块化 JDK
- 201: Modular Source Code 201： 模块化源代码
- 220: Modular Run-Time Images

  220：模块化运行时映像
- 260: Encapsulate Most Internal APIs

  260：封装大多数内部 API
- 282: jlink: The Java Linker

  282： jlink： Java 链接器

Phases 阶段

To the familiar phases of compile time (the `javac` command) and run time (the `java` run-time launcher) we add the notion of *link time*, an optional phase between the two in which a set of modules can be assembled and optimized into a custom run-time image. The linking tool, `jlink`, is the subject of JEP 282; many of the new command-line options implemented by `javac` and `java` are also implemented by `jlink`.

在熟悉的编译时（`javac` 命令）和运行时（`java` 运行时启动器）阶段中，我们添加了*链接时间*的概念，这是介于两者之间的一个可选阶段，在该阶段中，可以将一组模块组装并优化为自定义运行时映像。链接工具 `jlink` 是 JEP 282 的主题;由 JavaC 和 Java 实现的许多新命令行选项也是由 `JLink` 实现的。

Module paths 模块路径

The `javac`, `jlink`, and `java` commands, as well as several others, now accept options to specify various *module paths*. A module path is a sequence, each element of which is either a *module definition* or a directory containing module definitions. Each module definition is either

`javac`、`jlink` 和 `java` 命令以及其他几个命令现在接受用于指定各种*模块路径*的选项。模块路径是一个序列，其每个元素都是一个*模块定义* 或包含模块定义的目录。 每个模块定义都是 也

- A *module artifact*, *i.e.*, a modular JAR file or a JMOD file containing a compiled module definition, or else

  模块工件 ， *即*模块化 JAR 文件或包含已编译模块定义的 JMOD 文件，或者

- An *exploded-module directory* whose name is, by convention, the module's name and whose content is an "exploded" directory tree corresponding to a package hierarchy.

  一个 exploded-module 目录 ，按照约定，其名称是模块的名称，其内容是对应于 package 层次结构的 “exploded” 目录树。

In the latter case the directory tree can be a compiled module definition, populated with individual class and resource files and a `module-info.class` file at the root or, at compile time, a source module definition, populated with individual source files and a `module-info.java` file at the root.

在后一种情况下，目录树可以是已编译的模块 定义，填充了单独的类和资源文件，以及一个 `module-info.class` 文件，或者在编译时，源模块 定义，填充了单独的源文件和 `module-info.java` 文件。

A module path, like other kinds of paths, is specified by a string of path names separated by the host platform's path-separator character (`':'` on most platforms, `';'` on Windows).

与其他类型的路径一样，模块路径由一串路径名指定，这些路径名由主机平台的路径分隔符分隔（在大多数平台上为 `':'`，在 Windows 上为 `';'`）。

Module paths are very different from class paths: Class paths are a means to locate definitions of individual types and resources, whereas module paths are a means to locate definitions of whole modules. Each element of a class path is a container of type and resource definitions, *i.e.*, either a JAR file or an exploded, package-hierarchical directory tree. Each element of a module path, by contrast, is a module definition or a directory which each element in the directory is a module definition, *i.e.*, a container of type and resource definitions, *i.e.*, either a modular JAR file, a JMOD file, or an exploded module directory.

模块路径与类路径非常不同：类路径是查找单个类型和资源的定义的方法，而模块路径是查找整个模块定义的方法。类路径的每个元素都是类型和资源定义的容器， *即* 可以是 JAR 文件，也可以是松散的包分层目录树。 相比之下，模块路径的每个元素都是一个模块定义或 directory 中，该目录中的每个元素都是一个模块定义， *即*类型和资源定义的容器， *即*模块化 JAR 文件、JMOD 文件或松散的模块目录。

During the resolution process the module system locates a module by searching along several different paths, dependent upon the phase, and also by searching the compiled modules built-in to the environment, in the following order:

在解析过程中，模块系统根据阶段沿几条不同的路径搜索，并按以下顺序搜索内置到环境中的已编译模块来定位模块：

- The *compilation module path* (specified by the command-line option `--module-source-path`) contains module definitions in source form (compile time only).

  编译模块路径 （由 command-line 选项 `--module-source-path`）包含源形式的模块定义（仅限编译时）。

- The *upgrade module path* (`--upgrade-module-path`) contains compiled definitions of modules intended to be used in preference to the compiled definitions of any upgradeable modules present amongst the system modules or on the application module path (compile time and run time).

  升级模块路径 （`--upgrade-module-path`） 包含模块的编译定义，旨在优先使用系统模块中或应用程序模块路径（编译时和运行时）上存在的任何可升级模块的编译定义。

- The *system modules* are the compiled modules built-in to the environment (compile time and run time). These typically include Java SE and JDK modules but, in the case of a custom linked image, can also include library and application modules. At compile time the system modules can be overridden via the `--system` option, which specifies a JDK image from

which to load system modules.

*系统模块*是内置于 environment （编译时和运行时）。 这些通常包括 Java SE 和 JDK 模块 ，但在自定义链接映像的情况下，还可以包括库和应用程序模块。在编译时，可以通过 `--system` 选项覆盖系统模块，该选项指定要从中加载系统模块的 JDK 映像。

- The *application module path* (`--module-path`, or `-p` for short) contains compiled definitions of library and application modules (all phases). At link time this path can also contain Java SE and JDK modules.

  应用程序模块路径 （`--module-path`，或简称 `-p`）包含库和应用程序模块（所有阶段）的编译定义。在链接时，此路径还可以包含 Java SE 和 JDK 模块。

The module definitions present on these paths, together with the system modules, define the universe of *observable modules*.

这些路径上的模块定义与系统模块一起定义了*可观察模块*的范围。

When searching a module path for a module of a particular name, the module system takes the first definition of a module of that name. Version strings, if present, are ignored; if an element of a module path contains definitions of multiple modules with the same name then resolution fails and the compiler, linker, or virtual machine will report an error and exit. It is the responsibility of build tools and container applications to configure module paths so as to avoid version conflicts; it is not a goal of the module system to address the version-selection problem.

在模块路径中搜索特定名称的模块时，模块系统采用该名称的模块的第一个定义。如果存在版本字符串，则忽略版本字符串;如果模块路径的元素包含多个同名模块的定义，则解析将失败，编译器、链接器或虚拟机将报告错误并退出。构建工具和容器应用程序负责配置模块路径，以避免版本冲突;解决版本选择问题不是 Module System 的目标 。

Root modules ***根模块***

The module system constructs a module graph by resolving the transitive closure of the dependences of a set of *root modules* with respect to the set of observable modules.

模块系统通过解析一组*根模块*相对于可观察模块集的依赖关系的传递闭包来构造模块图。

When the compiler compiles code in the unnamed module, or the `java` launcher is invoked and the main class of the application is loaded from the class path into the unnamed module of the application class loader, then the *default set of root modules for the unnamed module* is computed as follows in JDK 9:

当编译器编译 unnamed 模块中的代码时，或者 `java` 启动器，并从 应用程序类加载器的 unnamed 模块中的类路径， 然后，在 JDK 9 中， *未命名模块的默认根模块集*的计算方式如下：

- The `java.se` module is a root, if it exists. If it does not exist then every `java.*` module on the upgrade module path or among the system modules that exports at least one package, without qualification, is a root.

  `java.se` 模块是根（如果存在）。如果不存在，则升级模块路径上的每个 java.* 模块或系统模块中导出至少一个包（无限定）的每个 `java.*` 模块都是根。

- Every non-`java.*` module on the upgrade module path or among the system modules that exports at least one package, without qualification, is also a root.

  升级模块路径上的每个非 java.* 模块或系统模块中导出至少一个包（不加限定）的每个非 `java.*` 模块也是一个根模块。

Update, June 2018: In JDK 11, the default set of root modules for the unnamed module changed. The default set is now computed as follows:

2018 年 6 月更新：在 JDK 11 中，默认的根模块集 未命名模块 更改 。现在，默认集的计算方式如下：

- *Every module on the upgrade module path or among the system modules that exports at least one package, without qualification, is a root.*

  升级模块路径上的每个模块或系统模块中至少导出一个软件包（不加限定）的每个模块都是一个根模块。

The `java.se` module still exists in JDK 11 and later, but it is no longer a root.

`java.se` 模块仍然存在于 JDK 11 及更高版本中，但它不再是根模块。

Otherwise, the default set of root modules depends upon the phase:

否则，默认的根模块集取决于阶段：

- At compile time it is usually the set of modules being compiled (more on this below);

  在编译时，它通常是正在编译的模块集（更多内容见下文）；

- At link time it is empty; and

  在链接时，它是空的;和

- At run time it is the application's main module, as specified via the `--module` (or `-m` for short) launcher option.

  在运行时，它是应用程序的主模块，通过 `--module` （或简称 `-m`） 启动器选项。

It is occasionally necessary to add modules to the default root set in order to ensure that specific platform, library, or service-provider modules will be present in

the resulting module graph. In any phase the option

有时需要将模块添加到默认根集，以确保特定的平台、库或 service-provider 模块将出现在生成的模块图中。在任何阶段，选项

```
--add-modules <module>(,<module>)*
```

where `<module>` is a module name, adds the indicated modules to the default set of root modules. This option may be used more than once.

其中 `<module>` 是模块名称，将指示的模块添加到默认的根模块集。此选项可以多次使用。

As a special case at run time, if `<module>` is `ALL-DEFAULT` then the default set of root modules for the unnamed module, as defined above, is added to the root set. This is useful when the application is a container that hosts other applications which can, in turn, depend upon modules not required by the container itself.

作为运行时的特殊情况，如果 `<module>` 为 `ALL-DEFAULT`，则未命名模块的默认根模块集（如上定义）将添加到根集中。 当应用程序是托管其他应用程序的容器时，这非常有用，而这些应用程序又可能依赖于容器本身不需要的模块。

As a further special case at run time, if `<module>` is `ALL-SYSTEM` then all system modules are added to the root set, whether or not they are in the default set. This is sometimes needed by test harnesses. This option will cause many modules to be resolved; in general, `ALL-DEFAULT` should be preferred.

作为运行时的进一步特殊情况，如果 `<module>` 是 `ALL-SYSTEM`，则所有系统模块都将添加到根集中，无论它们是否在默认集中。 测试框架有时需要这样做。此选项将导致解析许多模块;通常，应首选 `ALL-DEFAULT`。

As a final special case, at both run time and link time, if `<module>` is `ALL-MODULE-PATH` then all observable modules found on the relevant module paths are added to the root set. `ALL-MODULE-PATH` is valid at both compile time and run time. This is provided for use by build tools such as Maven, which already ensure that all modules on the module path are needed. It is also a convenient means to add automatic modules to the root set.

作为最后的特殊情况，在运行时和链接时，如果 `<module>` 是 `ALL-MODULE-PATH`， 则在相关模块路径上找到的所有可观察模块都将添加到根集中。`ALL-MODULE-PATH` 在编译时和运行时都有效。这是供 Maven 等构建工具使用的，这些工具已经确保需要模块路径上的所有模块。这也是将 automatic modules 添加到根集的便捷方法。

Limiting the observable modules

限制 observable 模块

It is sometimes useful to limit the observable modules for, *e.g.*, debugging, or to reduce the number of modules resolved when the main module is the unnamed module defined by the application class loader for the class path. The `--limit-modules` option can be used, in any phase, to do this. Its syntax is:

有时，限制可观察模块用于调试等目的，或者当主模块是应用程序类加载器为类路径定义的未命名模块时，减少解析的模块数量是有用的。可以在任何阶段使用 `--limit-modules` 选项来执行此作。它的语法是：

```
--limit-modules <module>(,<module>)*
```

where `<module>` is a module name. The effect of this option is to limit the observable modules to those in the transitive closure of the named modules plus the main module, if any, plus any further modules specified via the `--add-modules` option.

其中 `<module>` 是模块名称。此选项的效果是将可观察模块限制为命名模块加上主模块（如果有）以及通过 `--add-modules` 选项指定的任何其他模块的传递闭包中的模块。

(The transitive closure computed for the interpretation of the `--limit-modules` option is a temporary result, used only to compute the limited set of observable modules. The resolver will be invoked again in order to compute the actual module graph.)

（为解释 `--limit-modules` 选项是一个临时结果，仅用于计算有限的可观察模块集。将再次调用 resolver 以计算实际的 module graph.）

Increasing readability 提高可读性

When testing and debugging it is sometimes necessary to arrange for one module to read some other module, even though the first module does not depend upon the second via a `requires` clause in its module declaration. This may be needed to, *e.g.*, enable a module under test to access the test harness itself, or to access libraries related to the harness. The `--add-reads` option can be used, at both compile time and run time, to do this. Its syntax is:

在测试和调试时，有时需要安排一个模块读取其他模块，即使第一个模块不通过其模块声明中的 `requires` 子句依赖于第二个模块。例如 ， *可能需要*启用待测模块以访问 测试测试工具本身，或访问与工具相关的库。 这 `--add-reads` 选项可以在编译时和运行时使用。它的语法是：

```
--add-reads <source-module>=<target-module>
```

where `<source-module>` and `<target-module>` are module names.

其中 `<source-module>` 和 `<target-module>` 是模块名称。

The `--add-reads` option can be used more than once. The effect of each instance is to add a readability edge from the source module to the target module. This is, essentially, a command-line form of a `requires` clause in a module declaration, or an invocation of an unrestricted form of the `Module::addReads` method. As a consequence, code in the source module will be able to access types in a package of the target module at both compile time and run time if that package is exported via an `exports` clause in the source module's declaration, an invocation of the `Module::addExports` method, or an instance of the `--add-exports` option

(defined below). Such code will, additionally, be able to access types in a package of the target module at run time if that module is declared to be open or if that package is opened via an opens clause in the source module's declaration, an invocation of the `Module::addOpens` method, or an instance of the `--add-opens` option (also defined below).

`--add-reads` 选项可以多次使用。每个实例的效果是从源模块添加可读性边缘 添加到目标模块中。 这实质上是 `requires` 子句，或者调用 `Module：：addReads` 方法的无限制形式。因此，如果该包是通过源模块声明中的 `exports` 子句、`Module：：addExports` 方法的调用或 `--add-exports` 选项的实例（定义如下）导出的，则源模块中的代码将能够在编译时和运行时访问目标模块的包中的类型。此外，如果目标模块被声明为打开的，或者该包是通过源模块声明中的 `opens` 子句打开的，调用 `Module：：addOpens` 方法 ，或者 `--add-opens` 选项的实例（也在下面定义），则此类代码将能够在运行时访问目标模块的包中的类型。

If, for example, a test harness injects a white-box test class into the `java.management` module, and that class extends an exported utility class in the (hypothetical) `testng` module, then the access it requires can be granted via the option

例如，如果测试框架将白盒测试类注入到 `java.management` 模块，并且该类扩展了（假设的）`testng` 模块中导出的实用程序类，那么它所需的访问权限可以通过选项

```
--add-reads java.management=testng
```

As a special case, if the `<target-module>` is `ALL-UNNAMED` then readability edges will be added from the source module to all present and future unnamed modules, including that corresponding to the class path. This allows code in modules to be tested by test frameworks that have not, themselves, yet been converted to modular form.

作为特殊情况，如果 `<target-module>` 是 `ALL-UNNAMED`，则可读性边缘将从源模块添加到所有当前和将来的未命名模块，包括与类路径对应的模块。这允许模块中的代码由测试框架测试，这些框架本身尚未转换为模块化形式。

Breaking encapsulation 打破封装

It is sometimes necessary to violate the access-control boundaries defined by the module system, and enforced by the compiler and virtual machine, in order to allow one module to access some of the unexported types of another module. This may be desirable in order to, *e.g.*, enable white-box testing of internal types, or to expose unsupported internal APIs to code that has come to depend upon them. The `--add-exports` option can be used, at both compile time and run time, to do this. Its syntax is:

有时需要违反由模块系统定义并由编译器和虚拟机强制执行的访问控制边界，以便允许一个模块访问另一个模块的某些未导出类型。这可能是可取的，例如， 启用内部类型的白盒测试，或公开 Unsupported 内部 API 来编写依赖于它们的代码。 这 `--add-exports` 选项可以在编译时和运行时来执行此作。它的语法是：

```
--add-exports &lt;source-module&gt;/&lt;package&gt;=&lt;target-module&gt;(,&lt;target-module&gt;)*
```

where `<source-module>` and `<target-module>` are module names and `<package>` is the name of a package.

其中 `<source-module>` 和 `<target-module>` 是模块名称， `<package>` 是包的名称。

The `--add-exports` option can be used more than once, but at most once for any particular combination of source module and package name. The effect of each instance is to add a qualified export of the named package from the source module to the target module. This is, essentially, a command-line form of an `exports` clause in a module declaration, or an invocation of an unrestricted form of the `Module::addExports` method. As a consequence, code in the target module will be able to access public types in the named package of the source module if the target module reads the source module, either via a `requires` clause in its module declaration, an invocation of the `Module::addReads` method, or an instance of the `--add-reads` option.

`--add-exports` 选项可以多次使用，但对于源模块和包名称的任何特定组合，最多使用一次。每个实例的效果是将命名包的限定导出从源模块添加到目标模块。这实质上是模块中 `exports` 子句的命令行形式 声明，或调用不受限制的 `Module：：addExports` 方法 。因此，如果目标模块读取源模块，则目标模块中的代码将能够访问源模块的命名包中的公共类型，无论是通过其模块声明中的 `requires` 子句、调用 `Module：：addReads` 方法还是 `--add-reads` 选项。

If, for example, the module `jmx.wbtest` contains a white-box test for the unexported `com.sun.jmx.remote.internal` package of the `java.management` module, then the access it requires can be granted via the option

例如，如果模块 `jmx.wbtest` 包含未导出的 `com.sun.jmx.remote.internal` 包的白盒测试。 `java.management` 模块，则可以通过选项

```
--add-exports java.management/com.sun.jmx.remote.internal=jmx.wbtest
```

As a special case, if the `<target-module>` is `ALL-UNNAMED` then the source package will be exported to all unnamed modules, whether they exist initially or are created later on. Thus access to the `sun.management` package of the `java.management` module can be granted to all code on the class path via the option

作为特殊情况，如果 `<target-module>` 是 `ALL-UNNAMED`，则 源码包会被导出到所有未命名的模块中，无论它们是否 最初存在或稍后创建。 因此，访问 `java.management` 模块的 `sun.management` 软件包可以通过选项

```
--add-exports java.management/sun.management=ALL-UNNAMED
```

The `--add-exports` option enables access to the public types of a specified package. It is sometimes necessary to go further and enable access to all non-public elements via the `setAccessible` method of the core reflection API. The `--add-opens` option can be used, at run time, to do this. It has the same syntax as

the `--add-exports` option:

`--add-exports` 选项允许访问指定包的公有类型。有时需要更进一步，通过核心反射 API 的 `setAccessible` 方法启用对所有非公共元素的访问。 这 `--add-opens` 选项可以在运行时执行此作。它与 `--add-exports` 选项具有相同的语法：

```
    --add-opens <source-module>/<package>=<target-module>(,<target-module>)*
```

where `<source-module>` and `<target-module>` are module names and `<package>` is the name of a package.

其中 `<source-module>` 和 `<target-module>` 是模块名称， `<package>` 是包的名称。

The `--add-opens` option can be used more than once, but at most once for any particular combination of source module and package name. The effect of each instance is to add a qualified open of the named package from the source module to the target module. This is, essentially, a command-line form of an `opens` clause in a module declaration, or an invocation of an unrestricted form of the `Module::addOpens` method. As a consequence, code in the target module will be able to use the core reflection API to access all types, public and otherwise, in the named package of the source module so long as the target module reads the source module.

`--add-opens` 选项可以多次使用，但对于源模块和包名称的任何特定组合，最多使用一次。每个实例的效果是将命名包的限定打开从源模块添加到目标模块。这实质上是模块声明中 `opens` 子句的命令行形式，或者是对 `Module：：addOpens` 方法的不受限制形式的调用。因此，只要目标模块读取源模块，目标模块中的代码将能够使用核心反射 API 访问源模块的命名包中的所有类型，包括公共和其他类型。

Open packages are indistinguishable from non-exported packages at compile time, so the `--add-opens` option may not be used in that phase.

在编译时，打开的包与未导出的包没有区别，因此在该阶段可能不使用 `--add-opens` 选项。

> **The `--add-exports` and `--add-opens` options must be used with great care. You can use them to gain access to an internal API of a library module, or even of the JDK itself, but you do so at your own risk: If that internal API is changed or removed then your library or application will fail.**
>
> **必须非常小心地使用 `--add-exports` 和 `--add-opens` 选项。您可以使用它们来访问库模块的内部 API，甚至 JDK 本身，但这样做的风险由您自己承担：如果该内部 API 被更改或删除，那么您的库或应用程序将失败。**

***Patching module content** 修补模块内容*

When testing and debugging it is sometimes useful to replace selected class files or resources of specific modules with alternate or experimental versions, or to provide entirely new class files, resources, and even packages. This can be done via the `--patch-module` option, at both compile time and run time. Its syntax is:

在测试和调试时，有时将特定模块的选定类文件或资源替换为替代版本或实验版本，或者提供全新的类文件、资源甚至包是有用的。这可以通过 `--patch-module` 选项在编译时和运行时完成。它的语法是：

```
    --patch-module <module>=<file>(<pathsep><file>)*
```

where `<module>` is a module name, `<file>` is the filesystem path name of a module definition, and `<pathsep>` is the host platform's path-separator character.

其中 `<module>` 是模块名称，`<file>` 是模块定义的文件系统路径名称，`<pathsep>` 是主机平台的路径分隔符。

The `--patch-module` option can be used more than once, but at most once for any particular module name. The effect of each instance is to change how the module system searches for a type in the specified module. Before it checks the actual module, whether part of the system or defined on a module path, it first checks, in order, each module definition specified to the option. A patch path names a sequence of module definitions but it is not a module path, since it has leaky, class-path-like semantics. This allows a test harness, *e.g.*, to inject multiple tests into the same package without having to copy all of the tests into a single directory.

`--patch-module` 选项可以多次使用，但对于任何特定的模块名称，最多只能使用一次。每个实例的效果是更改模块系统在指定模块中搜索类型的方式。在检查实际模块之前，无论是系统的一部分还是在模块路径上定义，它首先按顺序检查为选项指定的每个模块定义。patch path 命名了一系列模块定义，但它不是模块路径，因为它具有泄漏的、类似类路径的语义。这允许测试工具（ *例如* ）将多个测试注入到同一个包中，而不必将所有测试复制到单个目录中。

The `--patch-module` option cannot be used to replace `module-info.class` files. If a `module-info.class` file is found in a module definition on a patch path then a warning will be issued and the file will be ignored.

`--patch-module` 选项不能用于替换 `module-info.class` 文件。 如果在补丁路径的模块定义中找到 `module-info.class` 文件，则将发出警告并忽略该文件。

If a package found in a module definition on a patch path is not already exported or opened by that module then it will, still, not be exported or opened. It can be exported or opened explicitly via either the reflection API or the `--add-exports` or `--add-opens` options.

如果在 patch 路径上的模块定义中找到的包尚未被该模块导出或打开，则该模块仍不会导出或打开该包。它可以通过反射 API 或 `--add-exports` 或 `--add-opens` 选项显式导出或打开。

The `--patch-module` option replaces the `-Xbootclasspath:/p` option, which has been removed (see below).

`--patch-module` 选项替换了已删除的 `-Xbootclasspath：/p` 选项（请参阅下文）。

The `--patch-module` option is intended only for testing and debugging. Its use in production settings is strongly discouraged.

`--patch-module` 选项仅用于测试和调试。强烈建议不要在生产环境中使用它。

Compile time 编译时

The `javac` compiler implements the options described above, as applicable to compile time: `--module-source-path`, `--upgrade-module-path`, `--system`, `--module-path`, `--add-modules`, `--limit-modules`, `--add-reads`, `--add-exports`, and `--patch-module`.

`javac` 编译器实现上述选项，适用于编译时：`--module-source-path`、 `--upgrade-module-path`， `--system`， `--module-path`， `--add-modules`， `--limit-modules`、`--add-reads`、`--add-exports` 和 `--patch-module`。

The compiler operates in one of three modes, each of which implements additional options.

编译器在三种模式之一下运行，每种模式都实现其他选项。

- *Legacy mode* is enabled when the compilation environment, as defined by the `-source`, `-target`, and `--release` options, is less than or equal to 8. None of the modular options described above may be used.

  当编译环境（由 `-source`、`-target` 和 `--release` 选项定义）小于或等于 8 时，将启用*传统模式* 。不能使用上述任何模块化选项。

In legacy mode the compiler behaves in essentially the same way as it does in JDK 8.

在 legacy 模式下，编译器的行为方式与 JDK 8 中的行为方式基本相同。

- *Single-module mode* is enabled when the compilation environment is 9 or later and the `--module-source-path` option is not used. The other modular options described above may be used; the existing options `-bootclasspath`, `-Xbootclasspath`, `-extdirs`, `-endorseddirs`, and `-XXuserPathsFirst` may not be used.

  当编译环境为 9 及以上，且不使用 `--module-source-path` 选项时，开启*单模块模式* 。可以使用上述其他模块化选项;现有选项 `-bootclasspath`、`-Xbootclasspath`、`-extdirs`、 `-endorseddirs` 和 `-XXuserPathsFirst` 不能使用。

Single-module mode is used to compile code organized in a traditional package-hierarchical directory tree. It is the natural replacement for simple uses of legacy mode of the form

单模块模式用于编译在传统包分层目录树中组织的代码。它是 legacy 模式的简单使用的自然替代品

```
$ javac -d classes -classpath classes -sourcepath src Foo.java
```

If a module descriptor in the form of a `module-info.java` or `module-info.class` file is specified on the command line, or is found on the source path or the class path, then source files will be compiled as members of the module named by that descriptor and that module will be the sole root module. Otherwise if the `--module <module>` option is present then source files will be compiled as members of `<module>`, which will be the root module. Otherwise source files will be compiled as members of the unnamed module, and the root modules will be computed as described above.

如果 `module-info.java` 或 `module-info.class` 文件在命令行上指定，或者在源路径或类路径上找到，则源文件将被编译为由该描述符命名的模块的成员，并且该模块将成为唯一的根模块。否则，如果存在 `--module <module>` 选项，则源文件将被编译为 `<module>` 的成员。 这将是根模块。 否则，将编译源文件 作为未命名模块的成员，并且根模块将被计算 如上所述 。

It is possible to put arbitrary classes and JAR files on the class path in this mode, but that is not recommended since it amounts to treating those classes and JAR files as part of the module being compiled.

在此模式下，可以将任意类和 JAR 文件放在类路径上，但不建议这样做，因为这相当于将这些类和 JAR 文件视为正在编译的模块的一部分。

- *Multi-module mode* is enabled when the compilation environment is 9 or later and the `--module-source-path` option is used. The existing `-d` option to name the output directory must also be used; the other modular options described above may be used; the existing options `-bootclasspath`, `-Xbootclasspath`, `-extdirs`, `-endorseddirs`, and `-XXuserPathsFirst` may not be used.

  当编译环境为 9 及以上，且使用 `--module-source-path` 选项时，开启*多模块模式* 。 现有的 还必须使用 `-d` 选项来命名输出目录;另一个 可以使用上述模块化选项;现有选项 `-bootclasspath`、`-Xbootclasspath`、`-extdirs`、`-endorseddirs` 和 `-XXuserPathsFirst` 不能使用。

Multi-module mode is used to compile one or more modules, whose source code is laid out in exploded-module directories on the module source path. In this mode the module membership of a type is determined by the position of its source file in the module source path, so each source file specified on the command line must exist within an element of that path. The set of root modules is the set of modules for which at least one source file is specified.

多模块模式用于编译一个或多个模块，其源代码布局在模块源路径上的 exploded-module 目录中。在此模式下，类型的模块成员身份由其源文件在模块源路径中的位置确定，因此在命令行上指定的每个源文件必须存在于该路径的元素中。根模块集是至少指定了一个源文件的模块集。

In contrast to the other modes, in this mode an output directory must be specified via the `-d` option. The output directory will be structured as an element of a module path, *i.e.*, it will contain exploded-module directories which themselves contain class and resource files. If the compiler finds a module on the module source path but cannot find the source file for some type in that module then it will search the output directory for the corresponding class file.

与其他模式相反，在此模式下，必须通过 `-d` 选项指定输出目录。output 目录将被构建为模块路径的一个元素， *即* ，它将包含爆炸模块目录，这些目录本身包含类和资源文件。如果编译器在模块源路径上找到模块，但在该模块中找不到某种类型的源文件，则它将在输出目录中搜索相应的类文件。

In large systems the source code for a particular module may be spread across several different directories. In the JDK itself, *e.g.*, the source files for a module may be found in any one of the directories `src/<module>/share/classes`, `src/<module>/<os>/classes`, or `build/gensrc/<module>`, where `<os>` is the name of the target operating system. To express this in a module source path while preserving module identities we allow each element of such a path to use braces ({ and }) to enclose commas-separated lists of alternatives and a single asterisk (*) to stand for the module name. The module source path for the JDK can then be written as

在大型系统中，特定模块的源代码可能分布在几个不同的目录中。 在 JDK 本身中， *例如* ，模块的源文件可以在 `src/<module>/share/classes`、`src/<module>/<os>/classes` 中的任何一个目录中找到，或者 `build/gensrc/<module>`，其中 `<os>` 是目标作系统的名称。为了在模块源路径中表达这一点，同时保留模块标识，我们允许此类路径的每个元素使用大括号（{ 和 }） 将逗号分隔的替代项列表括起来，并使用一个星号 （*） 来表示模块名称。然后，可以将 JDK 的模块源路径编写为

```
{src/*/{share,<os>}/classes,build/gensrc/*}
```

In both of the modular modes the compiler will, by default, generate various warnings related to the module system; these may be disabled via the option `-Xlint:-module`. More precise control of these warnings is available via the `exports`, `opens`, `requires-automatic`, and `requires-transitive-automatic` keys for the `-Xlint` option.

在这两种 modular 模式下，默认情况下，编译器将生成与 module system 相关的各种警告;这些可以通过选项 `-Xlint：-module` 来禁用。可以通过 `exports`、`opens`、`requires-automatic` 和 `requires-transitive-automatic` 键。

The new option `--module-version <version>` may be used to specify the version strings of the modules being compiled.

新选项 `--module-version <version>` 可用于指定正在编译的模块的版本字符串。

Class-file attributes ***类文件属性***

A JDK-specific class-file attribute, `ModuleTarget`, optionally records the target operating system and architecture of the module descriptor that contains it. Its format is:

特定于 JDK 的类文件属性 `ModuleTarget` 可以选择记录目标作系统和包含它的模块描述符的体系结构。其格式为：

```
ModuleTarget_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 os_arch_index; // index to a CONSTANT_utf8_info structure
}
```

The UTF-8 string in the constant pool at `os_arch_index` has the format `<os>-<arch>`, where `<os>` is typically one of `linux`, `macos`, `solaris`, or `windows`, and `<arch>` is typically one of `x86`, `amd64`, `sparcv9`, `arm`, or `aarch64`.

`os_arch_index` 处常量池中的 UTF-8 字符串的格式为 `<os>-<arch>`，其中 `<os>` 通常是 `linux`、`macos`、 `solaris` 或 `windows`，`<arch>` 通常是 `x86`、`amd64`、 `sparcv9`、`arm` 或 `aarch64`。

Packaging: Modular JAR files

打包：模块化 JAR 文件

The `jar` tool can be used without change to create modular JAR files, since a modular JAR file is just a JAR file with a `module-info.class` file in its root directory.

jar 工具无需更改即可用于创建 modular JAR 文件 ，因为 modular JAR 文件只是一个 JAR 文件，其根目录中有一个 `module-info.class` 文件。

The `jar` tool implements the following new options to allow the insertion of additional information into module descriptors as modules are packaged:

`jar` 工具实现了以下新选项，以允许在打包模块时将其他信息插入模块描述符：

- `--main-class=<class-name>`, or `-e <class-name>` for short, causes `<class-name>` to be recorded in the `module-info.class` file as the class containing the module's `public static void main` entry point. (This is not a new option; it already records the main class in the JAR file's manifest.)

  `--main-class=<class-name>`，或简称 `-e <class-name>`，导致 `<class-name>` 记录在 `module-info.class` 文件中，作为包含模块的 `public static void` 主入口点的类。（这不是一个新选项;它已经在 JAR 文件的清单中记录了主类。

- `--module-version=<version>` causes `<version>` to be recorded in the `module-info.class` file as the module's version string.

  `--module-version=<version>` 导致 `<version>` 被记录在 `module-info.class` 文件作为模块的版本字符串。

- `--hash-modules=<pattern>` causes hashes of the content of the specific modules that depend upon this module, in a particular set of observable modules, to be recorded in the `module-info.class` file for later use in the validation of dependencies. Hashes are only recorded for modules whose names match the regular expression `<pattern>`. If this option is used then the `---module-path` option, or `-p` for short, must also be used to specify the set of observable modules for the purpose of computing the modules that depend upon this module.

  `--hash-modules=<pattern>` 将依赖于此模块的特定模块的内容的哈希值记录在一组特定的可观察模块中，以记录在 `module-info.class` 文件中 供以后在依赖项验证中使用。 哈希值仅是 recorded （对于名称与正则表达式匹配的模块） < 模式>。 如果使用此选项，则 `---module-path` option（或简称 `-p`）也必须用于指定可观察模块的集合，以便计算依赖于该模块的模块。

- `--describe-module`, or `-d` for short, displays the module descriptor, if any, of the specified JAR file.

  `--describe-module`（或简称 `-d`）显示指定 JAR 文件的模块描述符（如果有）。

The `jar` tool's `--help` option can be used to show a complete summary of its command-line options.

`jar` 工具的 `--help` 选项可用于显示其命令行选项的完整摘要。

Two new JDK-specific JAR-file manifest attributes are defined to correspond to the `--add-exports` and `--add-opens` command-line options:

定义了两个新的 JDK 特定的 JAR 文件清单属性，以对应于 `--add-exports` 和 `--add-opens` 命令行选项：

- `Add-Exports: <module>/<package>( <module>/<package>)*`
- `Add-Opens: <module>/<package>( <module>/<package>)*`

The value of each attribute is a space-separated list of slash-separated module-name/package-name pairs. A `<module>/<package>` pair in the value of an `Add-Exports` attribute has the same meaning as the command-line option `--add-exports <module>/<package>=ALL-UNNAMED`. A `<module>/<package>` pair in the value of an `Add-Opens` attribute has the same meaning as the command-line option `--add-opens <module>/<package>=ALL-UNNAMED`.

每个属性的值都是以斜杠分隔的 module-name/package-name 对的空格分隔列表。`Add-Exports` 属性值中的 `<module>/<package>` 对与命令行选项 `--add-exports <module>/<package>=ALL-UNNAMED` 具有相同的含义。 一个 `Add-Opens` 属性值中的 `<module>/<package>` 对与命令行选项 `--add-opens <module>/<package>=ALL-UNNAMED` 具有相同的含义。

Each attribute can occur at most once, in the main section of a `MANIFEST.MF` file. A particular pair can be listed more than once. If a specified module was not resolved, or if a specified package does not exist, then the corresponding pair is ignored. These attributes are interpreted only in the main executable JAR file of an application, i.e., in the JAR file specified to the `-jar` option of the Java run-time launcher; they are ignored in all other JAR files.

每个属性最多可以出现一次，在 清单。MF 文件。特定对可以多次列出。如果未解析指定的模块，或者指定的包不存在，则忽略相应的对。这些属性仅在应用程序的主可执行 JAR 文件中解释，即在 Java 运行时启动器的 `-jar` 选项指定的 JAR 文件中解释;它们在所有其他 JAR 文件中都将被忽略。

Packaging: JMOD files 打包：JMOD 文件

The new JMOD format goes beyond JAR files to include native code, configuration files, and other kinds of data that do not fit naturally, if at all, into JAR files. JMOD files are used to package the modules of the JDK itself; they can also be used by developers to package their own modules, if desired.

新的 JMOD 格式超越了 JAR 文件，包括本机代码、配置文件和其他类型的数据，这些数据自然不适合 JAR 文件。JMOD 文件用于打包 JDK 本身的模块;如果需要，开发人员还可以使用它们来打包自己的模块。

JMOD files can be used at compile time and link time, but not at run time. To support them at run time would require, in general, that we be prepared to extract and link native-code libraries on-the-fly. This is feasible on most platforms, though it can be very tricky, and we have not seen many use cases that require this capability, so for simplicity we have chosen to limit the utility of JMOD files in this release.

JMOD 文件可以在编译时和链接时使用，但不能在运行时使用。为了在运行时支持它们，通常需要我们准备好动态提取和链接本机代码库。这在大多数平台上都是可行的，尽管它可能非常棘手，而且我们还没有看到很多需要此功能的用例，因此为了简单起见，我们选择在此版本中限制 JMOD 文件的实用性。

A new command-line tool, `jmod`, can be used to create, manipulate, and examine JMOD files. Its general syntax is:

新的命令行工具 `jmod` 可用于创建、作和检查 JMOD 文件。它的一般语法是：

```
$ jmod (create|extract|list|describe|hash) <options> <jmod-file>
```

For the `create` subcommand, `<options>` can include the `--main-class`, `--module-version`, `--hash-modules`, and `---module-path` options described above for the `jar` tool, and also:

对于 `create` 子命令，`<options>` 可以包含 `--main-class`、 `--module-version`、`--hash-modules` 和 `---module-path` 选项，以及：

- `--class-path <path>` specifies a class path whose content will be copied into the resulting JMOD file.

`--class-path <path>` 指定一个类路径，其内容将被复制到生成的 JMOD 文件中。

- `--cmds <path>` specifies one or more directories containing native commands to be copied.

  `--cmds <path>` 指定一个或多个包含要复制的本机命令的目录。

- `--config <path>` specifies one or more directories containing configuration files to be copied.

  `--config <path>` 指定一个或多个包含要复制的配置文件的目录。

- `--exclude <pattern-list>` specifies files to be excluded, where `<pattern-list>` is a comma-separated list of patterns of the form `<glob-pattern>`, `glob:<glob-pattern>`, or `regex:<regex-pattern>`.

  `--exclude <pattern-list>` 指定要排除的文件，其中 `<pattern-list>` 是格式为逗号分隔的模式列表 `<glob-pattern>`、`glob:<glob-pattern>` 或 `regex:<regex-pattern>`。

- `--header-files <path>` specifies one or more directories containing C and C++ header files to be copied.

  `--header-files <path>` 指定要复制的一个或多个包含 C 和 C++ 头文件的目录。

- `--legal-notices <path>` specifies one or more directories containing legal notices to be copied.

  `--legal-notices <path>` 指定一个或多个包含要复制的法律声明的目录。

- `--libs <path>` specifies one or more directories containing native libraries to be copied.

  `--libs <path>` 指定一个或多个包含要复制的本机库的目录。

- `--man-pages <path>` specifies one or more directories containing manual pages to be copied.

  `--man-pages <path>` 指定一个或多个包含要复制的手册页的目录。

- `--target-platform <os>-<arch>` specifies the target operating system and architecture, to be recorded in the `ModuleTarget` attribute of the `module-info.class` file.

  `--target-platform <os>-<arch>` 指定要记录在 `module-info.class` 文件的 `ModuleTarget` 属性中的目标作系统和体系结构。

The `extract` subcommand accepts a single option, `--dir`, to indicate the directory into which the content of the specified JMOD file should be written. The directory will be created if it does not exist. If this option is not present then the content will be extracted into the current directory.

`extract` 子命令接受一个选项 `--dir`，以指示应将指定 JMOD 文件的内容写入的目录。如果目录不存在，则将创建该目录。如果此选项不存在，则内容将被提取到当前目录中。

The `list` subcommand lists the content of the specified JMOD file; the `describe` subcommand displays the module descriptor of the specified JMOD file, in the same format as the `--describe-module` options of the `jar` and `java` commands. These subcommands accept no options.

`list` 子命令列出指定 JMOD 文件的内容;这 `describe` 子命令显示指定 JMOD 文件的模块描述符，其格式与 `jar` 和 `java` 命令。这些子命令不接受任何选项。

The `hash` subcommand can be used to hash an existing set of JMOD files. It requires both the `--module-path` and `--hash-modules` options.

`hash` 子命令可用于对一组现有的 JMOD 文件进行哈希处理。它需要 `--module-path` 和 `--hash-modules` 选项。

The `jmod` tool's `--help` option can be used to show a complete summary of its command-line options.

`jmod` 工具的 `--help` 选项可用于显示其命令行选项的完整摘要。

***Link time** 链接时间*

The details of the command-line linking tool, `jlink`, are described in JEP 282. At a high level its general syntax is:

命令行链接工具 `jlink` 的详细信息在 JEP 282。在高层次上，它的一般语法是：

```
$ jlink <options> ---module-path <modulepath> --output <path>
```

where the `---module-path` option specifies the set of observable modules to be considered by the linker and the `--output` option specifies the path of the directory that will contain the resulting run-time image. The other `<options>` can include the `---limit-modules` and `---add-modules` options, described above, as well as additional linker-specific options.

其中 `---module-path` 选项指定链接器要考虑的可观察模块集，`--output` 选项指定将包含生成的运行时映像的目录的路径。其他 `<options>` 可以包括 `---limit-modules` 和 `---add-modules` 选项，以及其他特定于 linker 的选项。

The `jlink` tool's `--help` option can be used to show a complete summary of its command-line options.

`jlink` 工具的 `--help` 选项可用于显示其命令行选项的完整摘要。

***Run time** 运行时间*

The HotSpot virtual machine implements the options described above, as applicable to run time: `--upgrade-module-path`, `--module-path`, `--add-modules`, `--limit-modules`, `--add-reads`, `--add-exports`, `--add-opens`, and `--patch-module`. These options can be passed to the command-line launcher, `java`, and also to the JNI invocation API.

HotSpot 虚拟机实现上述适用于运行时的选项：`--upgrade-module-path`、`--module-path`、`--add-modules`、`--limit-modules`、`--add-reads`、`--add-exports`、`--add-opens` 和 `--patch-module` 的 API API 这些选项可以传递给命令行启动器 `java` 以及 JNI 调用 API。

An additional option specific to this phase and supported by the launcher is:

特定于此阶段并受启动器支持的另一个选项是：

- `--module <module>`, or `-m <module>` for short, specifies the main module of a modular application. This will be the default root module for the purpose of constructing the application's initial module graph. If the main module's descriptor does not indicate a main class then the syntax `<module>/<class>` can be used, where `<class>` names the class that contains the application's `public static void main` entry point.

  `--module <module>`，或简称 `-m <module>`，指定模块化应用程序的主模块。这将是用于构建应用程序的初始模块图的默认根模块。如果主模块的描述符没有指示主类，则可以使用语法 `<module>/<class>`，其中 `<class>` 命名包含应用程序的 `public static void` 主入口点的类。

Additional diagnostic options supported by the launcher include:

启动器支持的其他诊断选项包括：

- `--list-modules` displays the names and version strings of the observable modules and then exits, in the same manner as `java --version`.

  `--list-modules` 显示可观察模块的名称和版本字符串，然后退出，方式与 `java --version` 相同。

- `--describe-module <module>`, or `-d <module>` for short, displays the module descriptor of the specified module, in the same format as the `jar -d` option and the `jmod describe` subcommand, and then exits.

  `--describe-module <module>`，或简称 `-d <module>`，以与 `jar -d` 选项和 `jmod describe` 子命令相同的格式显示指定模块的模块描述符，然后退出。

- `--validate-modules` validates all the observable modules, checking for conflicts and other potential errors, and then exits.

  `--validate-modules` 验证所有可观察的模块，检查冲突和其他潜在错误，然后退出。

- `--dry-run` initializes the virtual machine and loads the main class but does not invoke the main method; this is useful for validating the configuration of the module system.

  `--dry-run` 初始化虚拟机并加载 main 类，但不调用 main 方法;这对于验证 Module System 的配置很有用。

- `--show-module-resolution` causes the module system to describe its activities as it constructs the initial module graph.

  `--show-module-resolution` 使模块系统在构建初始模块图时描述其活动。

- `-Dsun.reflect.debugModuleAccessChecks` causes a thread dump to be shown whenever an access check in the `java.lang.reflect` API fails with an `IllegalAccessException` or an `InaccessibleObjectException`. This is useful for debugging when the underlying reason for a failure is hidden because the exception is caught and not re-thrown.

  `-Dsun.reflect.debugModuleAccessChecks` 每当 `java.lang.reflect` API 中的访问检查失败并显示 `IllegalAccessException` 或 `InaccessibleObjectException` 时，都会显示线程转储。当失败的根本原因被隐藏时，这非常有用，因为异常被捕获而不是重新引发。

- `-Xlog:module+[load|unload][=[debug|trace]]` causes the VM to log debug or trace messages as modules are defined and changed in the run-time module graph. These options generate voluminous output during startup.

  `-Xlog:module+[load|unload][=[debug|trace]]` 导致 VM 在运行时模块图中定义和更改模块时记录调试或跟踪消息。这些选项在启动过程中生成大量输出。

- `-verbose:module` is a shorthand for `-Xlog:module+load -Xlog:module+unload`.

  `-verbose: module` 是 `-Xlog:module+load -Xlog:module+unload` .

- `-Xlog:init=debug` causes a stack trace to be displayed if the initialization of the module system fails.

  `-Xlog: init=debug` 会导致模块系统初始化失败时显示堆栈跟踪。

- `--version`, `--show-version`, `--help`, and `--help-extra` display the same information and work in the same way as the existing `-version`, `-show-version`, `-help`, and `-Xhelp` options, respectively, except that they write the help text to the standard output stream rather than the standard error stream.

  `--version`、`--show-version`、`--help` 和 `--help-extra` 显示 相同的信息和工作方式与现有的 `-version`、`-show-version`、`-help` 和 `-Xhelp` 选项，不同之处在于它们将帮助文本写入标准输出流而不是标准错误流。

The stack traces generated for exceptions at run time have been extended to include, when present, the names and version strings of relevant modules. The detail strings of exceptions such as `ClassCastException`, `IllegalAccessException`, and `IllegalAccessError` have also been updated to include module information.

在运行时为异常生成的堆栈跟踪已扩展为包括相关模块的名称和版本字符串（如果存在）。异常的详细信息字符串，例如 `ClassCastException`、 `IllegalAccessException` 和 `IllegalAccessError` 也已更新，以包含模块信息。

The existing `-jar` option has been enhanced so that if the manifest file of the JAR file being launched contains a `Launcher-Agent-Class` attribute then the JAR file is launched as both an application and as an agent for that application. This allows `java -jar foo.jar` to be used in place of the more verbose `java -javaagent:foo.jar -jar foo.jar`.

现有的 `-jar` 选项已得到增强，因此，如果正在启动的 JAR 文件的清单文件包含 `Launcher-Agent-Class` 属性，则 JAR 文件将作为应用程序和 agent 的代理。这允许使用 `java -jar foo.jar` 代替更详细的 `java -javaagent:foo.jar -jar foo.jar` .

Relaxed strong encapsulation

松弛的强封装

In this release the strong encapsulation of some of the JDK's packages is relaxed by default, as permitted by the Java SE 9 Platform Specification. This relaxation is controlled at run time by a new launcher option, `--illegal-access`, which works as follows:

在此版本中，某些 JDK 软件包的强封装是 默认情况下是 relaxed 的， Java SE 9 平台规范允许 。 此松弛在运行时由新的 launcher 选项 `--illegal-access`，其工作原理如下：

- `--illegal-access=permit` opens each package in each module in the run-time image to code in all unnamed modules, *i.e.*, to code on the class path, if that package existed in JDK 8. This enables both static access, *i.e.*, by compiled bytecode, and deep reflective access, via the platform's various reflection APIs.

  `--illegal-access=permit` 打开运行时映像中每个模块中的每个包，以在所有未命名模块中编码， *即*在类路径上编码（如果该包存在于 JDK 8 中）。这既支持静态访问（ *即*通过编译的字节码），也可以通过平台的各种反射 API 实现深度反射访问。

  The first reflective-access operation to any such package causes a warning to be issued, but no warnings are issued after that point. This single warning describes how to enable further warnings. This warning cannot be suppressed.

  对任何此类包的第一次反射访问作都会导致发出警告，但在该点之后不会发出警告。此单个警告描述如何启用更多警告。无法抑制此警告。

  This mode is the default in JDK 9. It will be phased out in a future release and, eventually, removed.

  此模式是 JDK 9 中的默认模式。它将在未来发行版中逐步淘汰，并最终被删除。

- `--illegal-access=warn` is identical to `permit` except that a warning message is issued for each illegal reflective-access operation.

  `--illegal-access=warn` 与 `permit` 相同，只是为每个非法反射访问作发出警告消息。

- `--illegal-access=debug` is identical to `warn` except both a warning message and a stack trace are issued for each illegal reflective-access operation.

  `--illegal-access=debug` 与 `warn` 相同，只是会为每个非法反射访问作发出警告消息和堆栈跟踪。

- `--illegal-access=deny` disables all illegal-access operations except for those enabled by other command-line options, e.g., `--add-opens`.

  `--illegal-access=deny` 禁用所有非法访问作，但其他命令行选项（例如 `--add-opens`）启用的作除外。

  This mode will become the default in a future release.

  此模式将成为未来发行版中的默认模式。

When deny becomes the default illegal-access mode then `permit` will likely remain supported for at least one release, so that developers can continue to migrate their code. The `permit`, `warn`, and debug modes will, over time, be removed, as will the `--illegal-access` option itself. (For launch-script compatibility the unsupported modes will most likely just be ignored, after issuing a warning to that effect.)

当 deny 成为默认的非法访问模式时，`permit` 可能会至少在一个版本中保持受支持状态，以便开发人员可以继续迁移其代码。 随着时间的推移，`permit`、`warn` 和 debug 模式将被删除，`--illegal-access` 选项本身也将被删除。（为了 launch-script 兼容性，在发出警告后，很可能会忽略不支持的模式。

The default mode, `--illegal-access=permit`, is intended to make you aware when you have code on the class path that reflectively accesses some JDK-internal API at least once. To prepare for the future you can use the `warn` or debug modes to learn about all such accesses. For each library or framework on the class path that requires illegal access you have two options:

默认模式 `--illegal-access=permit` 旨在让您知道类路径上的代码何时反射性地访问某些 JDK 内部 API 至少一次。为了将来做好准备，您可以使用 `warn` 或 debug 模式来了解所有此类访问。对于类路径上需要非法访问的每个库或框架，您有两个选项：

- If the component's maintainers have already released a new, fixed version that no longer uses JDK-internal APIs then you can consider upgrading to that version.

  如果组件的维护者已经发布了不再使用 JDK 内部 API 的新修复版本，那么您可以考虑升级到该版本。

- If the component still needs to be fixed then we encourage you to contact its maintainers and ask them to replace their use of JDK-internal APIs with proper exported APIs, if available.

  如果组件仍需要修复，那么我们鼓励您联系其维护者，并要求他们用适当的导出 API（如果可用）替换他们对 JDK 内部 API 的使用。

If you must continue to use a component that requires illegal access then you can eliminate the warning messages by using one or more `--add-opens` options to open just those internal packages to which access is required.

如果必须继续使用需要非法访问的组件，则可以使用一个或多个 `--add-opens` 来消除警告消息 选项以打开需要访问的内部包。

To verify that your application is ready for the future, run it with `--illegal-access=deny` along with any necessary `--add-opens` options. Any remaining illegal-access errors will most likely be due to static references from compiled code to JDK-internal APIs. You can identify those by running the `jdeps` tool with the `--jdk-internals` option. (The run-time system does not issue warnings for illegal static-access operations because that would require deep VM changes and degrade performance.)

要验证您的应用程序是否已为将来做好准备，请使用 `--illegal-access=deny` 以及任何必要的 `--add-opens` 选项。任何剩余的非法访问错误很可能是由于从编译代码到 JDK 内部 API 的静态引用造成的。您可以通过运行带有 `--jdk-internals` 选项的 `jdeps` 工具来识别这些值。（运行时系统不会对非法静态访问作发出警告，因为这需要对 VM 进行深度更改并降低性能。

The warning message issued when an illegal reflective-access operation is detected has the following form:

检测到非法反射访问作时发出的警告消息采用以下形式：

```
WARNING: Illegal reflective access by $PERPETRATOR to $VICTIM
```

where: 哪里：

- $PERPETRATOR is the fully-qualified name of the type containing the code that invoked the reflective operation in question plus the code source (i.e., JAR-file path), if available, and

  $PERPETRATOR 是类型的完全限定名称，其中包含调用相关反射作的代码加上代码源（即 JAR 文件路径）（如果可用），以及

- $VICTIM is a string that describes the member being accessed, including the fully-qualified name of the enclosing type

  $VICTIM 是一个字符串，用于描述正在访问的成员，包括封闭类型的完全限定名称

In the default mode, `--illegal-access=permit`, at most one of these warning messages will be issued, accompanied by additional instructive text. Here is an example, from running Jython:

在默认模式下 `--illegal-access=permit`，最多会发出一条警告消息，并附有额外的指导性文本。下面是一个运行 Jython 的示例：

```
$ java -jar jython-standalone-2.7.0.jar
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by jnr.posix.JavaLibCHelper (file:/tmp/jython-standalone-2.7.0.jar) to method sun.nio.ch.S
WARNING: Please consider reporting this to the maintainers of jnr.posix.JavaLibCHelper
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Jython 2.7.0 (default:9987c746f838, Apr 29 2015, 02:25:11)
[OpenJDK 64-Bit Server VM (Oracle Corporation)] on java9
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
```

The run-time system makes a best-effort attempt to suppress duplicate warnings for the same $PERPETRATOR and $VICTIM.

运行时系统会尽最大努力尝试禁止显示相同$PERPETRATOR 和$VICTIM 的重复警告。

An extended example 扩展示例

Suppose we have an application module, `com.foo.bar`, which depends upon a library module, `com.foo.baz`. If we have the source code for both modules in the module-path directory `src`:

假设我们有一个应用程序模块 `com.foo.bar`，它依赖于库模块 `com.foo.baz`。如果我们在 module-path 目录 `src` 中有这两个模块的源代码：

```
src/com.foo.bar/module-info.java
src/com.foo.bar/com/foo/bar/Main.java
src/com.foo.baz/module-info.java
src/com.foo.baz/com/foo/baz/BazGenerator.java
```

then we can compile them, together:

然后我们可以一起编译它们：

```
$ javac --module-source-path src -d mods $(find src -name '*.java')
```

The output directory, `mods`, is a module-path directory containing exploded, compiled definitions of the two modules:

输出目录 mods 是一个 module-path 目录，包含两个模块的分解、编译定义：

```
mods/com.foo.bar/module-info.class
mods/com.foo.bar/com/foo/bar/Main.class
mods/com.foo.baz/module-info.class
mods/com.foo.baz/com/foo/baz/BazGenerator.class
```

Assuming that the com.foo.bar.Main class contains the application's entry point, we can run these modules as-is:

假设 com.foo.bar.Main 类包含应用程序的入口点，我们可以按原样运行这些模块：

```
$ java -p mods -m com.foo.bar/com.foo.bar.Main
```

Alternatively, we can package them up into modular JAR files:

或者，我们可以将它们打包成模块化的 JAR 文件：

```
$ jar --create -f mlib/com.foo.bar-1.0.jar \
      --main-class com.foo.bar.Main --module-version 1.0 \
      -C mods/com.foo.bar .
$ jar --create -f mlib/com.foo.baz-1.0.jar \
      --module-version 1.0 -C mods/com.foo.baz .
```

The mlib directory is a module-path directory containing the packaged, compiled definitions of the two modules:

mlib 目录是一个 module-path 目录，包含两个模块的打包、编译定义：

```
$ ls -l mlib
-rw-r--r-- 1501 Sep  6 12:23 com.foo.bar-1.0.jar
-rw-r--r-- 1376 Sep  6 12:23 com.foo.baz-1.0.jar
```

We can now run the packaged modules directly:

我们现在可以直接运行打包的模块：

```
$ java -p mlib -m com.foo.bar
```

jtreg enhancements

JTREG 增强功能

The jtreg test harness supports a new declarative tag, @modules, to express a test's dependences upon the modules in the system being tested. It takes a series of space-separated arguments, each of which can be of the form

jtreg 测试框架支持新的声明性标签 @modules，表示测试对被测系统中模块的依赖性。它采用一系列以空格分隔的参数，每个参数都可以采用以下形式

- <module>, where <module> is a module name, to indicate that the specified module must be present;

  <module>，其中 <module> 是模块名称，表示必须存在指定的模块;

- <module>/<package>, to indicate that the specified module must be present and the specified package must be exported to the test's module; or

  <module>/<package>，表示指定的模块必须存在，并且必须将指定的包导出到测试的模块;或

- <module>/<package>:<flag>, to indicate that the specified module must be present and, if the flag is open then the specified package must be opened to the test's module, or else if the flag is +open then the specified package must be both exported and opened to the test's module.

  <module>/<package>：<flag>，表示指定的模块必须存在，如果标志为 open，则必须向测试的模块打开指定的包，否则如果标志为 +open 然后，必须将指定的包导出并打开到 test 的模块。

A default set of @modules arguments, which will be used for all tests in a directory hierarchy that do not include such a tag, can be specified as the value of the modules property in a TEST.ROOT file or in any TEST.properties file.

可以将一组默认的 @modules 参数指定为 TEST 中 modules 属性的值，该参数将用于目录层次结构中不包含此类标记的所有测试 。ROOT 文件或任何 TEST.properties 文件。

The existing @compile tag accepts a new option, /module=<module>. This has the effect of invoking javac with the --module <module> option, defined above, to compile the specified classes as members of the indicated module.

现有的 @compile 标记接受新选项 /module=<module>。这具有使用 --module <module> 调用 javac 的效果 选项，将指定的类编译为 指示的模块。

Class loaders 类加载器

The Java SE Platform API historically specified two class loaders: The *bootstrap class loader*, which loads classes from the bootstrap class path, and the *system class loader*, which is the default delegation parent for new class loaders and, typically, the class loader used to load and start the application. The specification does not mandate the concrete types of either of these class loaders, nor their precise delegation relationship.

Java SE 平台 API 历史上指定了两个类加载器： *Bootstrap 类加载器* （从 Bootstrap 类路径加载类）和*系统类加载器* （新类加载器的默认委托父级，通常用于加载和启动应用程序的类加载器）。该规范没有规定这些类加载器的具体类型，也没有规定它们的确切委托关系。

The JDK has, since the 1.2 release, implemented a three-level hierarchy of class loaders, where each loader delegates to the next:

自 1.2 版本以来，JDK 实现了类加载器的三级层次结构，其中每个加载器都委托给下一个加载器：

- The application class loader, an instance of `java.net.URLClassLoader`, loads classes from the class path and is installed as the system class loader unless an alternate system loader is specified via the system property `java.system.class.loader`.

  应用程序类加载器，即 `java.net.URLClassLoader` 从类路径加载类，并且是 作为系统类加载器安装，除非有备用系统 loader 是通过 system 属性指定的 `java.system.class.loader` 中。

- The extension class loader, also an instance of `URLClassLoader`, loads classes available via the extension mechanism and, also, some resources and service providers built-in to the JDK. (This loader is not mentioned explicitly in the Java SE Platform API Specification.)

  扩展类加载器也是 `URLClassLoader` 的一个实例，它加载通过扩展机制可用的类，以及 JDK 中内置的一些资源和服务提供者。 （Java SE 平台 API 规范中未明确提及此加载程序。

- The bootstrap class loader, which is implemented solely within the virtual machine and is represented by `null` in the `ClassLoader` API, loads classes from the bootstrap class path.

  引导类加载器，仅在虚拟机中实现，在 `ClassLoader` 中由 `null` 表示 API，从 bootstrap 类路径加载类。

JDK 9 retains this three-level hierarchy, in order to preserve compatibility, while making the following changes to implement the module system:

JDK 9 保留了这个三级层次结构，以保持兼容性，同时进行了以下更改以实现模块系统：

- The application class loader is no longer an instance of `URLClassLoader` but, rather, of an internal class. It is the default loader for named modules that are neither Java SE nor JDK modules.

  应用程序类加载器不再是 `URLClassLoader` 的 URL，而是内部类。它是既不是 Java SE 也不是 JDK 模块的命名模块的默认加载程序。

- The extension class loader is no longer an instance of `URLClassLoader` but, rather, of an internal class. It no longer loads classes via the extension mechanism, which was removed by JEP 220. It does, however, define selected Java SE and JDK modules, about which more below. In its new role this loader is known as the *platform class loader*, it is available via the new `ClassLoader::getPlatformClassLoader` method, and it will be required by the Java SE Platform API Specification.

  扩展类加载器不再是 `URLClassLoader` 的 URL，而是内部类。 它不再是 加载类，该扩展机制已被 JEP 220 页。但是，它确实定义了选定的 Java SE 和 JDK 模块，下面将详细介绍。在其新角色中，此加载器称为 *平台类加载器* ，它可以通过新的 `ClassLoader::getPlatformClassLoader` 方法 ，并且 Java SE 平台 API 规范将要求这样做。

- The bootstrap class loader is implemented in both library code and within the virtual machine, but for compatibility it is still represented by `null` in the `ClassLoader` API. It defines the core Java SE and JDK modules.

  引导类加载器在库代码和虚拟机中都实现，但为了兼容性，它在 `ClassLoader` API 中仍由 `null` 表示。它定义了核心 Java SE 和 JDK 模块。

The platform class loader is retained not only for compatibility but, also, to improve security. Types loaded by the bootstrap class loader are implicitly granted all security permissions (`AllPermission`), but many of these types do not actually require all permissions. We have *de-privileged* modules that do not require all permissions by defining them to the platform loader rather than the bootstrap class loader, and by granting them the permissions they actually need in the default security policy file. The Java SE and JDK modules defined to the platform class loader are:

保留平台类加载器不仅是为了兼容性，也是为了提高安全性。引导类加载器加载的类型被隐式授予所有安全权限 （`AllPermission`），但 其中许多类型实际上并不需要所有权限。 我们有 *取消* 不需要所有权限的模块，方法是将它们定义为平台加载器而不是引导程序类加载器，并在默认安全策略文件中授予它们实际需要的权限。为平台类加载器定义的 Java SE 和 JDK 模块是：

```
java.activation*           jdk.accessibility
java.compiler*             jdk.charsets
java.corba*                jdk.crypto.cryptoki
java.scripting             jdk.crypto.ec
java.se                    jdk.dynalink
java.se.ee                 jdk.incubator.httpclient
java.security.jgss         jdk.internal.vm.compiler*
java.smartcardio           jdk.jsobject
java.sql                   jdk.localedata
java.sql.rowset            jdk.naming.dns
java.transaction*          jdk.scripting.nashorn
java.xml.bind*             jdk.security.auth
java.xml.crypto            jdk.security.jgss
java.xml.ws*               jdk.xml.dom
java.xml.ws.annotation*    jdk.zipfs
```

(An asterisk, `'*'`, in these lists indicates an upgradeable module.)

（这些列表中的星号 “*” 表示可升级的模块。

JDK modules that provide tools or export tool APIs are defined to the application class loader:

提供工具或导出工具 API 的 JDK 模块被定义到应用程序类加载器中：

```
jdk.aot                     jdk.jdeps
jdk.attach                  jdk.jdi
jdk.compiler                jdk.jdwp.agent
jdk.editpad                 jdk.jlink
jdk.hotspot.agent           jdk.jshell
jdk.internal.ed             jdk.jstatd
jdk.internal.jvmstat        jdk.pack
jdk.internal.le             jdk.policytool
jdk.internal.opt            jdk.rmic
jdk.jartool                 jdk.scripting.nashorn.shell
jdk.javadoc                 jdk.xml.bind*
jdk.jcmd                    jdk.xml.ws*
jdk.jconsole
```

All other Java SE and JDK modules are defined to the bootstrap class loader:

所有其他 Java SE 和 JDK 模块都定义到引导类加载器中：

```
java.base                   java.security.sasl
java.datatransfer           java.xml
java.desktop                jdk.httpserver
java.instrument             jdk.internal.vm.ci
java.logging                jdk.management
java.management             jdk.management.agent
java.management.rmi         jdk.naming.rmi
java.naming                 jdk.net
java.prefs                  jdk.sctp
java.rmi                    jdk.unsupported
```

The three built-in class loaders work together to load classes as follows:

三个内置类加载器协同工作以加载类，如下所示：

- The application class loader first searches the named modules defined to all of the built-in loaders. If a suitable module is defined to one of these loaders then that loader will load the class. If a class is not found in a named module defined to one of these loaders then the application class loader delegates to its parent. If a class is not found by its parent then the application class loader searches the class path. Classes found on the class path are loaded as members of this loader's unnamed module.

  应用程序类加载器首先搜索为所有内置加载器定义的命名模块。如果为这些加载器之一定义了合适的模块，则该加载器将加载该类。如果在定义为这些加载器之一的命名模块中找不到类，则应用程序类加载器将委托给其父类。如果类未被其父类找到，则应用程序类加载器将搜索类路径。在 class path 上找到的类将作为此 loader 的 unnamed 模块的成员加载。

- The platform class loader searches the named modules defined to all of the built-in loaders. If a suitable module is defined to one of these loaders then that loader will load the class. (The platform class loader can, consequently, now delegate to the application class loader, which can be useful when a module on the upgrade module path depends upon a module on the application module path.) If a class is not found in a named module defined to one of these loaders then the platform class loader delegates to its parent.

  平台类加载器搜索为所有内置加载器定义的命名模块。如果为这些加载器之一定义了合适的模块，则该加载器将加载该类。（因此，平台类加载器现在可以委托给应用程序类加载器，当升级模块路径上的模块依赖于应用程序模块路径上的模块时，这可能很有用。如果在定义为这些加载器之一的命名模块中找不到类，则平台类加载器将委托给其父类。

- The bootstrap class loader searches the named modules defined to itself. If a class is not found in a named module defined to the bootstrap loader then the bootstrap class loader searches the files and directories added to the bootstrap class path via the `-Xbootclasspath/a` option. Classes found on the bootstrap class path are loaded as members of this loader's unnamed module.

  引导类加载器搜索定义为 本身。 如果在定义为 bootstrap 加载程序，然后引导类加载程序搜索文件 以及通过 `-Xbootclasspath/a` 选项。在 bootstrap 类路径上找到的类将作为此加载程序的 unnamed 模块的成员加载。

The application and platform class loaders delegate to their respective parent loaders in order to ensure that the bootstrap class path is still searched when a class is not found in a module defined to one of the built-in loaders.

应用程序和平台类加载器委托给各自的父加载器，以确保在定义为内置加载器之一的模块中找不到类时，仍会搜索引导程序类路径。

Removed: Bootstrap class-path options

已删除：Bootstrap 类路径选项

In earlier releases the `-Xbootclasspath` option allows the default bootstrap class path to be overridden, and the `-Xbootclasspath/p` option allows a sequence of files and directories to be prepended to the default path. The computed value of this path is reported via the JDK-specific system property `sun.boot.class.path`.

在早期版本中，`-Xbootclasspath` 选项允许覆盖默认引导程序类路径，而 `-Xbootclasspath/p` 选项允许在默认路径前面添加一系列文件和目录。此路径的计算值通过特定于 JDK 的系统属性 `sun.boot.class.path` 报告。

With the module system in place the bootstrap class path is empty by default, since bootstrap classes are loaded from their respective modules. The `javac`

compiler only supports the `-Xbootclasspath` option in legacy mode, the `java` launcher no longer supports either of these options, and the system property `sun.boot.class.path` has been removed.

有了模块系统，引导类路径默认为空，因为引导类是从各自的模块加载的。`javac` 编译器仅支持传统模式下的 `-Xbootclasspath` 选项，`java` 启动程序不再支持这两个选项中的任何一个，并且系统属性 `sun.boot.class.path` 已被删除。

The compiler's `--system` option can be used to specify an alternate source of system modules, as described above, and its `-release` option can be used to specify an alternate platform version, as described in JEP 247 (Compile for Older Platform Versions). At run time the `--patch-module` option, mentioned above, can be used to inject content into modules in the initial module graph.

如上所述，编译器的 `--system` 选项可用于指定系统模块的备用源，其 `-release` 选项 可用于指定备用平台版本，如 JEP 247（针对较旧的平台版本编译）。 在运行时， `--patch-module` 选项可用于将内容注入初始模块图中的模块。

A related option, `-Xbootclasspath/a`, allows files and directories to be appended to the default bootstrap class path. This option, and the related API in the `java.lang.instrument` package, is sometimes used by instrumentation agents, so for compatibility it is still supported at run time. Its value, if specified, is reported via the JDK-specific system property `jdk.boot.class.path.append`. This option can be passed to the command-line launcher, `java`, and also to the JNI invocation API.

相关选项 `-Xbootclasspath/a` 允许将文件和目录附加到默认引导程序类路径。此选项以及 `java.lang.instrument` 包中的相关 API 有时由检测代理使用，因此为了兼容性，它在运行时仍受支持。如果指定了该值，则通过 JDK 特定的系统属性 `jdk.boot.class.path.append` 报告。此选项可以传递给命令行启动器 `java` 以及 JNI 调用 API。

Testing 测试

Many existing tests were affected by the introduction of the module system. In JDK 9 the `@modules` tag, described above, was added to the unit and regression tests as needed, and tests that used the `-Xbootclasspath/p` option or assumed that the system class loader is a `URLClassLoader` were updated.

许多现有的测试都受到模块系统引入的影响。在 JDK 9 中，上述 `@modules` 标记被添加到 单元和回归测试，以及使用 `-Xbootclasspath/p` 选项，或者假设系统类加载器是 `URLClassLoader` 已更新。

There is, of course, an extensive set of unit tests for the module system itself. In the JDK 9 source forest the run-time tests are in the test/jdk/modules directory of the `jdk` repository and the runtime/modules directory of the `hotspot` repository; the compile-time tests are in the tools/javac/modules directory of the `langtools` repository.

当然，对于模块系统，有一组广泛的单元测试 本身。 在 JDK 9 源林中，运行时测试位于 test/jdk/modules 目录和 runtime/modules 目录; 编译时测试位于 tools/javac/modules 中 `langtools` 存储库的目录。

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 primary risks of this proposal are ones of compatibility due to changes to existing language constructs, APIs, and tools.

此提案的主要风险是由于对现有语言结构、API 和工具的更改而导致的兼容性风险。

Changes due primarily to the introduction of the Java Platform Module System (JSR 376) include:

主要由于引入 Java 平台模块系统 （JSR 376） 包括：

- Applying the `public` modifier to an API element no longer guarantees that the element will be everywhere accessible. Accessibility now depends also upon whether the package containing that element is exported or opened by its defining module, and whether that module is readable by the module containing the code that is attempting to access it. For example, code of the following form might not work correctly:

  将 `public` 修饰符应用于 API 元素不再保证该元素在所有位置都可访问。可访问性现在还取决于包含该元素的包是否由其定义模块导出或打开，以及该模块是否可由包含尝试访问该模块的代码的模块读取。例如，以下形式的代码可能无法正常工作：

  ```
  Class<?> c = Class.forName(...);
  if (Modifier.isPublic(c.getModifiers()) {
      // Assume that c is accessible
  }
  ```

- If a package is defined in both a named module and on the class path then the package on the class path will be ignored. The class path can, therefore, no longer be used to augment packages that are built into the environment. The `javax.transaction` package, *e.g.*, is defined by the `java.transaction` module, so the class path will not be searched for types in that package. This restriction is important to avoid splitting packages across class loaders and across modules. At compile time and run time the upgrade module path can be used to upgrade modules that are built-in into

the environment. The `--patch-module` option can be used for other ad-hoc patching.

如果在命名模块和类路径中都定义了包，则将忽略类路径上的包。因此，类路径不能再用于扩充内置于环境中的软件包。 *例如* ， `javax.transaction` 包由 `java.transaction` 模块定义，因此类路径不会 搜索该包中的类型。 此限制很重要以避免跨类加载器和跨模块拆分包。 在编译时和运行时，升级模块路径可用于 升级环境中内置的模块。 这 `--patch-module` 选项可用于其他临时修补。

- The `ClassLoader::getResource*` methods can no longer be used to locate JDK-internal resources other than class files. Module-private non-class resources can be read via the `Class::getResource*` methods, the `Module::getResourceAsStream` method or via the `jrt:` URL scheme and filesystem defined in JEP 220.

  `ClassLoader：：getResource*` 方法不能再用于查找类文件以外的 JDK 内部资源。模块私有非类资源可以通过 `Class：：getResource*` 读取 方法、`Module：：getResourceAsStream` 方法或通过 `jrt：` JEP 220 中定义的 URL 方案和文件系统。

- The `java.lang.reflect.AccessibleObject::setAccessible` method cannot be used to gain access to public members of packages that are not exported or opened by their defining modules, or to non-public members of packages that are not opened by their defining modules; in either case, an `InaccessibleObjectException` will be thrown. If a framework library, such as a serializer, needs access to such members at run time then the relevant packages must be opened to the framework module by declaring the containing module to be open, by declaring that package to be open, or by opening that package via the `--add-opens` command-line option.

  该方法 `java.lang.reflect.AccessibleObject::setAccessible` 不能用于获取对 public 的访问权限 未通过其定义导出或打开的包的成员 modules 打开的 Modules 的 API 中，或者发送给未由 他们的定义模块;在任何一种情况下，`InaccessibleObjectException` 将被抛出。 如果框架 库（如序列化程序）需要在运行时访问此类成员 time 时，必须向框架打开相关软件包 module，通过声明包含的模块 to be open，通过声明 该软件包要打开，或者通过 `--add-opens` 命令行选项。

- JVM TI agents can no longer instrument Java code that runs early in the startup of the run-time environment. The `ClassFileLoadHook` event, in particular, is no longer sent during the primordial phase. The `VMStart` event, which signals the beginning of the start phase, is only posted after the the VM is initialized to the point where it can load classes in modules other than `java.base`. Two new capabilities, `can_generate_early_class_hook_events` and `can_generate_early_vmstart`, can be added by agents that are carefully written to handle events early in VM initialization. More details can be found in the updated description of the class file load hook event and the start event.

  JVM TI 代理无法再检测在运行时环境启动早期运行的 Java 代码。 类 `FileLoadHook` 特别是 event，在 primordial 阶段不再发送。 `VMStart` 事件表示开始阶段的开始，只有在 VM 初始化到可以在 `java.base` 以外的模块中加载类之后，才会发布该事件。两项新功能， `can_generate_early_class_hook_events` 以及 `can_generate_early_vmstart`，可以由精心编写的代理添加，以便在 VM 初始化的早期处理事件。更多详细信息可以在 class file load hook 事件的更新描述中找到 和 start 事件 。

- The == syntax in security policy files has been revised to augment the permissions granted to standard and JDK modules rather than override them. Applications that override other aspects of the JDK's default policy file therefore do not need to copy the default permissions granted to the standard and JDK modules.

  安全策略文件中的 == 语法已修订，以增强授予标准和 JDK 模块的权限，而不是覆盖它们。因此，覆盖 JDK 默认策略文件的其他方面的应用程序不需要复制授予 standard 和 JDK 模块的默认权限。

Modules that define Java EE APIs, or APIs primarily of interest to Java EE applications, have been deprecated and will be removed in a future release. They are not resolved by default for code on the class path:

定义 Java EE API 的模块或 Java EE 应用程序主要感兴趣的 API 已弃用，并将在将来发行版中删除。默认情况下，不会为 class path 上的代码解析它们：

- The default set of root modules for the unnamed module is based, in JDK 9, upon the `java.se` module rather than the `java.se.ee` module. Thus, by default, code in the unnamed module will not have access to APIs in the following modules:

  在 JDK 9 中，未命名模块的默认根模块集基于 `java.se` 模块，而不是 `java.se.ee` 模块。 因此，默认情况下，未命名模块中的代码不会有 访问以下模块中的 API：

  ```
  java.activation
  java.corba
  java.transaction
  java.xml.bind
  java.xml.ws
  java.xml.ws.annotation
  ```

  This is an intentional, if painful, choice, driven by two goals:

  这是一个有意为之的选择，尽管是痛苦的，但由两个目标驱动：

  - To avoid unnecessary conflicts with popular libraries that define types in some of the same packages. The widely-used `jsr305.jar`,

e.g., defines annotation types in the `javax.annotation` package, which is also defined by the `java.xml.ws.annotation` module.

为避免与定义 类型。 广泛使用的 *例如* ， `jsr305.jar` 在 `javax.annotation` 包，该包也由 `java.xml.ws.annotation` 模块。

- To make it easier for existing application servers to migrate to JDK 9. Application servers often override the content of one or more of these modules, and in the near term they are most likely to do so by continuing to place the necessary non-modular JAR files on the class path. If these modules were resolved by default then the maintainers of application servers would have to take awkward actions to exclude them in order to override them.

  使现有应用程序服务器更容易迁移到 JDK 9。应用程序服务器通常会覆盖这些模块中的一个或多个内容，并且在短期内，它们最有可能通过继续将必要的非模块化 JAR 文件放在类路径上来实现此目的。如果这些模块默认被解析，那么应用服务器的维护者将不得不采取尴尬的作来排除它们，以便覆盖它们。

These modules are still part of JDK 9. Code on the class path can be granted access to one or more of these modules, as needed, via the `--add-modules` option.

这些模块仍然是 JDK 9 的一部分。可以根据需要，通过 `--add-modules` 选项向类路径上的代码授予对一个或多个模块的访问权限。

The run-time behavior of some Java SE APIs has changed, though in ways that continue to honor their existing specifications:

某些 Java SE API 的运行时行为已更改，但其方式仍遵循其现有规范：

- The application and platform class loaders are no longer instances of the `java.net.URLClassLoader` class, as noted above. Existing code that invokes `ClassLoader::getSystemClassLoader` and blindly casts the result to `URLClassLoader`, or does the same thing with the parent of that class loader, might not work correctly.

  如上所述，应用程序和平台类加载器不再是 `java.net.URLClassLoader` 类的实例。调用 `ClassLoader::getSystemClassLoader` 并盲目地将结果强制转换为 `URLClassLoader` 或对该类加载器的父级执行相同作的现有代码可能无法正常工作。

- Some Java SE types have been de-privileged and are now loaded by the platform class loader rather than the bootstrap class loader, as noted above. Existing custom class loaders that delegate directly to the bootstrap class loader might not work correctly; they should be updated to delegate to the platform class loader, which is easily available via the new `ClassLoader::getPlatformClassLoader` method.

  如上所述，某些 Java SE 类型已被取消特权，现在由平台类加载器而不是引导类加载器加载。直接委托给引导程序类加载器的现有自定义类加载器可能无法正常工作;应该更新它们以委托给平台类加载器，这可以通过新 `ClassLoader::getPlatformClassLoader` 方法轻松获得。

- Instances of `java.lang.Package` created by the built-in class loaders for packages in named modules do not have specification or implementation versions. In previous releases this information was read from the manifest of `rt.jar`. Existing code that expects the `Package::getSpecification*` or `Package::getImplementation*` methods always to return non-null values might not work correctly.

  由内置类加载器为命名模块中的包创建的 `java.lang.Package` 实例没有规范或实现版本。在以前的版本中，此信息是从 `rt.jar` 的清单中读取的。 需要 `Package：：getSpecification*` 或 `Package：：getImplementation*` 方法始终返回非 null 值可能无法正常工作。

There are several source-incompatible Java SE API changes:

有几项源代码不兼容的 Java SE API 更改：

- The `java.lang` package includes two new top-level classes, `Module` and `ModuleLayer`. The `java.lang` package is implicitly imported on demand (*i.e.*, `import java.lang.*`). If code in an existing source file imports some other package on demand, and that package declares a `Module` or `ModuleLayer` type, and the existing code refers to that type, then the file will not compile without change.

  `java.lang` 包包括两个新的顶级类 `Module`。 和 `ModuleLayer` 的 `ModuleLayer` 进行匹配。`java.lang` 包是按需隐式导入的（ *即* `import java.lang.*`）。如果现有源文件中的代码按需导入其他一些包，并且该包声明了 `Module` 或 `ModuleLayer` 类型，并且现有代码引用了该类型，则文件将在不更改的情况下无法编译。

- The `java.lang.instrument.Instrumentation` interface declares two new abstract methods, `redefineModule` and `isModifiableModule`. This interface is not intended to be implemented outside of the `java.instrument` module. If there are external implementations then they will not compile on JDK 9 without change.

  该 `java.lang.instrument.Instrumentation` 接口声明了两个新的抽象方法，`redefineModule` 和 `isModifiableModule`。 这 接口不打算在 `java.instrument` 模块。如果存在外部实现，则它们不会在不更改的情况下在 JDK 9 上进行编译。

- The five-parameter `transform` method declared in the `java.lang.instrument.ClassFileTransformer` interface is now a default method. The interface now also declares a new `transform` method that makes the relevant `java.lang.reflect.Module` object available to the

transformer when instrumenting classes at load time. Existing compiled code will continue to run, but existing source code that uses the existing five-parameter transform method as a functional interface will no longer compile.

在 `java.lang.instrument.ClassFileTransformer` interface 现在是默认方法。该接口现在还声明了一个新的转换 该方法，该方法使相关的 `java.lang.reflect.Module` 对象在加载时对 transformer 可用。现有的编译代码将继续运行，但使用现有的五参数转换方法作为功能接口的现有源代码将不再编译。

Finally, changes due to revisions to JDK-specific APIs and tools include:

最后，由于对特定于 JDK 的 API 和工具的修订而导致的更改包括：

- Most of the JDK's internal APIs are inaccessible by default at compile time, as described in JEP 260. Existing code that compiled against these APIs with warnings in previous releases will no longer compile. A workaround is to break encapsulation via the `--add-exports` option, defined above.

  默认情况下，JDK 的大多数内部 API 在编译时都是不可访问的，如 JEP 260 中所述。现有代码 针对这些 API 进行编译，并在以前的版本中发出警告 不再编译。 解决方法是通过 `--add-exports` 选项。

- Selected critical internal APIs in the `sun.misc` and `sun.reflect` packages have been moved to the `jdk.unsupported` module, as described in JEP 260. Non-critical internal APIs in these packages, such as `sun.misc.BASE64{De,En}coder`, have been either moved or removed.

  在 `sun.misc` 和 `sun.reflect` 中选定的关键内部 API 软件包已移至 `jdk.unsupported` 模块，如 JEP 260 中所述。这些软件包中的非关键内部 API（如 `sun.misc.BASE64{De，En}coder`）已被移动或删除。

- If a security manager is present then the run-time permission `accessSystemModules` is required in order to access JDK-internal resources via the `ClassLoader::getResource*` or `Class::getResource*` methods; in previous releases, permission to read the file `${java.home}/lib/rt.jar` was required.

  如果存在安全管理器，则运行时权限 `accessSystemModules` 才能通过 `ClassLoader：：getResource*` 或 `Class：：getResource*` 方法;在以前的发行版中，需要读取文件 `${java.home}/lib/rt.jar` 的权限。

- The `-Xbootclasspath` and `-Xbootclasspath/p` options have been removed, as noted above. At compile time, the new `--release` option can be used to specify an alternate platform version (see JEP 247). At run time, the new `--patch-module` option, described above, can be used to inject content into system modules.

  如上所述，`-Xbootclasspath` 和 `-Xbootclasspath/p` 选项已被删除。在编译时，新的 `--release` 选项 可用于指定备用平台版本（请参阅 JEP 247）。在运行时，上述新的 `--patch-module` 选项可用于将内容注入系统模块。

- The JDK-specific system property `sun.boot.class.path` has been removed, since the bootstrap class path is empty by default. Existing code that uses this property might not work correctly.

  已删除特定于 JDK 的系统属性 `sun.boot.class.path`，因为默认情况下引导类路径为空。使用此属性的现有代码可能无法正常工作。

- The JDK-specific annotation `@jdk.Exported`, introduced by JEP 179, has been removed since the information it conveys is now recorded in the `exports` declarations of module descriptors. We have seen no evidence of this annotation being used by tools outside of the JDK.

  特定于 JDK 的注释 `@jdk`。导出 ，引入 JEP 179 已被删除，因为它传达的信息现在记录在模块描述符的 `exports` 声明中。我们没有看到 JDK 以外的工具使用此注释的证据。

- The `META-INF/services` resource files previously found in `rt.jar` and other internal artifacts are not present in the corresponding system modules, since service providers and dependences are now declared in module descriptors. Existing code that scans for such files might not work correctly.

  之前在 `rt.jar` 中找到的 `META-INF/services` 资源文件 和其他内部工件不存在于相应的 系统模块，因为服务提供商和依赖项现在是 在模块描述符中声明。 扫描此类文件可能无法正常工作。

- The JDK-specific system property `file.encoding` can be set on the command line via the `-D` option, as before, but it must specify a charset defined in the base module. If any other charset is specified then the run-time system will fail to start. Existing launch scripts that specify such charsets might not work correctly.

  像以前一样，可以通过 `-D` 选项在命令行上设置 JDK 特定的系统属性 `file.encoding`，但它必须指定在基本模块中定义的字符集。如果指定了任何其他字符集，则运行时系统将无法启动。指定此类字符集的现有启动脚本可能无法正常工作。

- The `com.sun.tools.attach` API can no longer be used, by default, to attach an agent to the current process or an ancestor of the current process. Such attachment operations can be enabled by setting the system property `jdk.attach.allowAttachSelf` on the command line.

  默认情况下，不能再使用 `com.sun.tools.attach` API 将代理附加到当前进程或当前进程的上级。可以通过在命令行上设置系统属性 `jdk.attach.allowAttachSelf` 来启用此类附件作。

- The dynamic loading of JVM TI agents will be disabled by default in a future release. To prepare for that change we recommend that applications that allow dynamic agents start using the option `-XX:+EnableDynamicAgentLoading` to enable that loading explicitly. The option `-XX:-EnableDynamicAgentLoading` disables dynamic agent loading.

  默认情况下，JVM TI 代理的动态加载将在 未来版本。 为了准备这一变化，我们建议允许动态代理程序的应用程序开始使用选项 `-XX:+EnableDynamicAgentLoading` 以显式启用该加载。该选项 `-XX:-EnableDynamicAgentLoading` 将禁用动态代理加载。

Dependences 依赖项

JEP 200 (The Modular JDK) originally defined the modules present in the JDK in an XML document, as an interim measure. This JEP moved those definitions to proper module descriptors, *i.e.*, `module-info.java` and `module-info.class` files, and the `modules.xml` file in the root source-code repository was removed.

JEP 200（模块化 JDK） 最初在 XML 文档中定义了 JDK 中存在的模块，作为临时措施。这个 JEP 将这些定义移动到适当的模块描述符中， *即* `module-info.java` 和 `module-info.class` 文件，以及 `modules.xml` 根源代码存储库中的文件已删除。

The initial implementation of JEP 220 (Modular Run-Time Images) in JDK 9 used a custom build-time tool to construct JRE and JDK images. This JEP replaced that tool with the `jlink` tool.

JEP 220 的初始实现（模块化运行时映像） 在 JDK 9 中使用自定义构建时工具来构建 JRE 和 JDK 图像。 这个 JEP 用 `jlink` 工具替换了那个工具。

Modular JAR files can also be Multi-Release JAR files, per JEP 238.

模块化 JAR 文件也可以是多版本 JAR 文件，每个 JEP 238。