Project Jigsaw: Module System Quick-Start Guide

Project Jigsaw：模块系统快速入门指南

This document provides a few simple examples to get developers started with modules.

本文档提供了几个简单的示例，供开发人员使用 从 modules 开始。

The file paths in the examples use forward slashes, and the path separators are colons. Developers on Microsoft Windows should use file paths with back slashes and a semi-colon as the path separator.

示例中的文件路径使用正斜杠，路径分隔符是冒号。Microsoft Windows 上的开发人员应使用带有反斜杠和分号的文件路径作为路径分隔符。

Greetings 问候

This first example is a module named `com.greetings` that simply prints "Greetings!". The module consists of two source files: the module declaration (`module-info.java`) and the main class.

第一个示例是名为 `com.greetings` 的模块，它只打印 “Greetings！”。该模块由两个源文件组成：模块声明 （`module-info.java`） 和 主类。

By convention, the source code for the module is in a directory that is the name of the module.

按照约定，模块的源代码位于作为模块名称的目录中。

```
src/com.greetings/com/greetings/Main.java
src/com.greetings/module-info.java

$ cat src/com.greetings/module-info.java
module com.greetings { }

$ cat src/com.greetings/com/greetings/Main.java
package com.greetings;
public class Main {
    public static void main(String[] args) {
        System.out.println("Greetings!");
    }
}
```

The source code is compiled to the directory `mods/com.greetings` with the following commands:

```
$ mkdir -p mods/com.greetings

$ javac -d mods/com.greetings \
    src/com.greetings/module-info.java \
    src/com.greetings/com/greetings/Main.java
```

Now we run the example with the following command:

```
$ java --module-path mods -m com.greetings/com.greetings.Main
```

`--module-path` is the module path, its value is one or more directories that contain modules. The `-m` option specifies the main module, the value after the slash is the

class name of the main class in the module.

源代码编译到目录 mods/com.greetings 替换为以下命令： 现在，我们使用以下命令运行示例： --module-path 是模块路径，其值是一个或多个包含模块的目录。-m 选项 指定 main module，斜杠后面的值是 class 模块中 main 类的名称。

Greetings world 问候世界

This second example updates the module declaration to declare a dependency on module `org.astro`. Module `org.astro` exports the API package `org.astro`.

```
src/org.astro/module-info.java
src/org.astro/org/astro/World.java
src/com.greetings/com/greetings/Main.java
src/com.greetings/module-info.java

$ cat src/org.astro/module-info.java
module org.astro {
    exports org.astro;
}

$ cat src/org.astro/org/astro/World.java
package org.astro;
public class World {
    public static String name() {
        return "world";
    }
}

$ cat src/com.greetings/module-info.java
module com.greetings {
    requires org.astro;
}

$ cat src/com.greetings/com/greetings/Main.java
package com.greetings;
import org.astro.World;
public class Main {
    public static void main(String[] args) {
        System.out.format("Greetings %s!%n", World.name());
    }
}
```

The modules are compiled, one at a time. The `javac` command to compile module `com.greetings` specifies a module path so that the reference to module `org.astro` and the types in its exported packages can be resolved.

```
$ mkdir -p mods/org.astro mods/com.greetings

$ javac -d mods/org.astro \
    src/org.astro/module-info.java src/org.astro/org/astro/World.java

$ javac --module-path mods -d mods/com.greetings \
    src/com.greetings/module-info.java src/com.greetings/com/greetings/Main.java
```

The example is run in exactly the same way as the first example:

第二个示例更新了 module 声明以声明 对模块 `org.astro` 的依赖。模块 `org.astro` 导出 API 包 `org.astro` 的 模块被编译，一次编译一个。用于编译模块 `com.greetings` 的 `javac` 命令指定了一个模块路径，以便对模块 `org.astro` 的引用 并且可以解析其导出的包中的类型。 该示例的运行方式与第一个示例完全相同：

```
$ java --module-path mods -m com.greetings/com.greetings.Main
Greetings world!
```

Multi-module compilation 多模块编译

In the previous example then module `com.greetings` and module `org.astro` were compiled separately. It is also possible to compile multiple modules with one `javac` command:

在前面的例子中， 模块 `com.greetings` 和 `org.astro` 模块是分开编译的。也可以使用一个 `javac` 编译多个模块 命令：

```
$ mkdir mods

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

$ find mods -type f
mods/com.greetings/com/greetings/Main.class
mods/com.greetings/module-info.class
mods/org.astro/module-info.class
mods/org.astro/org/astro/World.class
```

Packaging 包装

In the examples so far then the contents of the compiled modules are exploded on the file system. For transportation and deployment purposes then it is usually more convenient to package a module as a *modular JAR*. A modular JAR is a regular JAR file that has a `module-info.class` in its top-level directory. The following example creates `org.astro@1.0.jar` and `com.greetings.jar` in directory `mlib`.

```
$ mkdir mlib

$ jar --create --file=mlib/org.astro@1.0.jar \
    --module-version=1.0 -C mods/org.astro .

$ jar --create --file=mlib/com.greetings.jar \
    --main-class=com.greetings.Main -C mods/com.greetings .

$ ls mlib
com.greetings.jar   org.astro@1.0.jar
```

In this example, then module `org.astro` is packaged to indicate that its version is `1.0`. Module `com.greetings` has been packaged to indicate that its main class is `com.greetings.Main`. We can now execute module `com.greetings` without needing to specify its main class:

```
$ java -p mlib -m com.greetings
Greetings world!
```

The command line is also shortened by using `-p` as an alternative to `--module-path`.

在到目前为止的示例中，编译后的模块的内容 在文件系统上展开。用于运输和部署 目的，那么通常将模块打包为 *一个模块化的 JAR*。模块化 JAR 是一个常规的 JAR 文件，其顶级目录中有一个 `module-info.class`。以下示例创建 `org.astro@1.0.jar` 和 `com.greetings.jar` 在 `mlib` 目录中。 在这个例子中，模块 `org.astro` 被打包以表明其版本是 `1.0`。模块 `com.greetings` 已打包，以指示其主类是 `com.greetings.Main`。我们现在可以执行模块 `com.greetings`，而无需指定其 主类： 使用 `-p` 作为 `--module-path` 的替代方案也缩短了命令行。

The `jar` tool has many new options (see `jar -help`), one of which is to print the module declaration for a module packaged as a modular JAR.

`jar` 工具有许多新选项（参见 `jar -help`），其中之一是打印打包为模块化 JAR 的模块的模块声明。

```
$ jar --describe-module --file=mlib/org.astro@1.0.jar
org.astro@1.0 jar:file:///d/mlib/org.astro@1.0.jar/!module-info.class
exports org.astro
requires java.base mandated
```

Missing requires or missing exports

缺少 requires 或缺少导出

Now let's see what happens with the previous example when we mistakenly omit the `requires` from the `com.greetings` module declaration:

```
$ cat src/com.greetings/module-info.java
module com.greetings {
    // requires org.astro;
}

$ javac --module-path mods -d mods/com.greetings \
    src/com.greetings/module-info.java src/com.greetings/com/greetings/Main.java
src/com.greetings/com/greetings/Main.java:2: error: package org.astro is not visible
```

```
    import org.astro.World;
           ^
  (package org.astro is declared in module org.astro, but module com.greetings does not read it)
1 error
```

We now fix this module declaration but introduce a different mistake, this time we omit the `exports` from the `org.astro` module declaration:

现在让我们看看前面的例子会发生什么，当我们 错误地省略了 `com.greetings` 模块声明：我们现在修复这个模块声明，但引入了一个不同的 错误，这次我们省略了 `org.astro` 模块声明：

```
$ cat src/com.greetings/module-info.java
module com.greetings {
    requires org.astro;
}
$ cat src/org.astro/module-info.java
module org.astro {
    // exports org.astro;
}

$ javac --module-path mods -d mods/com.greetings \
    src/com.greetings/module-info.java src/com.greetings/com/greetings/Main.java
$ javac --module-path mods -d mods/com.greetings \
   src/com.greetings/module-info.java src/com.greetings/com/greetings/Main.java
src/com.greetings/com/greetings/Main.java:2: error: package org.astro is not visible
    import org.astro.World;
           ^
  (package org.astro is declared in module org.astro, which does not export it)
1 error
```

Services 服务业

Services allow for loose coupling between *service consumers* modules and *service providers* modules.

服务允许*服务使用者*模块和服务*提供者*模块之间松散耦合。

This example has a service consumer module and a service provider module:

此示例具有一个服务使用者模块和一个服务提供者模块：

- module `com.socket` exports an API for network sockets. The API is in package `com.socket` so this package is exported. The API is *pluggable* to allow for alternative implementations. The service type is class `com.socket.spi.NetworkSocketProvider` in the same module and thus package `com.socket.spi` is also exported.

  module `com.socket` 导出一个用于网络套接字的 API。API 位于 `com.socket` 包中，因此会导出此包。API 是*可插拔*的，以允许 替代实现。服务类型为 class `com.socket.spi.NetworkSocketProvider` 在同一个模块中，因此 `com.socket.spi` 包也被导出。

- module `org.fastsocket` is a service provider module. It provides an implementation of `com.socket.spi.NetworkSocketProvider`. It does not export any packages.

  模块 `org.fastsocket` 是一个服务提供者 模块。它提供了 `com.socket.spi.NetworkSocketProvider` 。它不会导出任何包。

The following is the source code for module `com.socket` .

```
$ cat src/com.socket/module-info.java
module com.socket {
    exports com.socket;
    exports com.socket.spi;
    uses com.socket.spi.NetworkSocketProvider;
}

$ cat src/com.socket/com/socket/NetworkSocket.java
package com.socket;

import java.io.Closeable;
import java.util.Iterator;
import java.util.ServiceLoader;
```

```
import com.socket.spi.NetworkSocketProvider;

public abstract class NetworkSocket implements Closeable {
    protected NetworkSocket() { }

    public static NetworkSocket open() {
        ServiceLoader<NetworkSocketProvider> sl
            = ServiceLoader.load(NetworkSocketProvider.class);
        Iterator<NetworkSocketProvider> iter = sl.iterator();
        if (!iter.hasNext())
            throw new RuntimeException("No service providers found!");
        NetworkSocketProvider provider = iter.next();
        return provider.openNetworkSocket();
    }
}


$ cat src/com.socket/com/socket/spi/NetworkSocketProvider.java
package com.socket.spi;

import com.socket.NetworkSocket;

public abstract class NetworkSocketProvider {
    protected NetworkSocketProvider() { }

    public abstract NetworkSocket openNetworkSocket();
}
```

The following is the source code for module `org.fastsocket` .

```
$ cat src/org.fastsocket/module-info.java
module org.fastsocket {
    requires com.socket;
    provides com.socket.spi.NetworkSocketProvider
        with org.fastsocket.FastNetworkSocketProvider;
}

$ cat src/org.fastsocket/org/fastsocket/FastNetworkSocketProvider.java
package org.fastsocket;

import com.socket.NetworkSocket;
import com.socket.spi.NetworkSocketProvider;

public class FastNetworkSocketProvider extends NetworkSocketProvider {
    public FastNetworkSocketProvider() { }

    @Override
    public NetworkSocket openNetworkSocket() {
        return new FastNetworkSocket();
    }
}

$ cat src/org.fastsocket/org/fastsocket/FastNetworkSocket.java
package org.fastsocket;

import com.socket.NetworkSocket;

class FastNetworkSocket extends NetworkSocket {
    FastNetworkSocket() { }
    public void close() { }
}
```

For simplicity, we compile both modules together. In practice then the service consumer module and service provider modules will nearly always be compiled separately.

以下是 `com.socket` 模块的源码 . 以下是 module 的源码 `org.fast 套接字 。` 为简单起见，我们将两个模块一起编译。在实践中 服务消费者模块和服务提供者模块将 几乎总是单独编译。

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

Finally we modify our module `com.greetings` to use the API.

最后，我们修改模块 `com.greetings` 以使用 API。

```
$ cat src/com.greetings/module-info.java
module com.greetings {
    requires com.socket;
}

$ cat src/com.greetings/com/greetings/Main.java
package com.greetings;

import com.socket.NetworkSocket;

public class Main {
    public static void main(String[] args) {
        NetworkSocket s = NetworkSocket.open();
        System.out.println(s.getClass());
    }
}


$ javac -d mods/com.greetings/ -p mods $(find src/com.greetings/ -name "*.java")
```

Finally we run it:

```
$ java -p mods -m com.greetings/com.greetings.Main
class org.fastsocket.FastNetworkSocket
```

The output confirms that the service provider has been located and that it was used as the factory for the `NetworkSocket`.

最后我们运行它： 输出确认已找到服务提供商，并且 它被用作 `NetworkSocket` 的工厂。

The linker 链接器

`jlink` is the linker tool and can be used to link a set of modules, along with their transitive dependences, to create a custom modular run-time image (see JEP 220).

`jlink` 是链接器工具，可用于链接一组模块及其传递依赖项，以创建自定义的模块化运行时映像（请参阅 JEP 220）。

The tool currently requires that modules on the module path be packaged in modular JAR or JMOD format. The JDK build packages the standard and JDK-specific modules in JMOD format.

该工具当前要求将模块路径上的模块打包为模块化 JAR 或 JMOD 格式。JDK 版本将标准模块和特定于 JDK 的模块打包为 JMOD 格式。

The following example creates a run-time image that contains the module `com.greetings` and its transitive dependences:

下面的示例创建一个运行时映像，其中包含模块 `com.greetings` 及其传递依赖项：

```
jlink --module-path $JAVA_HOME/jmods:mlib --add-modules com.greetings --output greetingsapp
```

The value to `--module-path` is a PATH of directories containing the packaged modules. Replace the path separator ':' with ';' on Microsoft Windows.

`--module-path` 的值是目录的 PATH 包含打包的模块。替换路径分隔符 ': ' 在 Microsoft Windows 上带有 ';“。

`$JAVA_HOME/jmods` is the directory containing `java.base.jmod` and the other standard and JDK modules.

`$JAVA_HOME/jmods` 是包含 `java.base.jmod` 和其他标准模块和 JDK 模块。

The directory `mlib` on the module path contains the artifact for module `com.greetings`.

模块路径上的目录 `mlib` 包含模块 `com.greetings` 的工件。

The `jlink` tool supports many advanced options to customize the generated image, see `jlink --help` for more options.

`jlink` 工具支持许多高级选项来自定义生成的镜像，更多选项见 `jlink --help`。

`--patch-module`

Developers that checkout `java.util.concurrent` classes from Doug Lea's CVS will be used to compiling the source files and deploying those classes with `-Xbootclasspath/p`.

从 Doug Lea 的 CVS 中签出 `java.util.concurrent` 类的开发人员将习惯于编译源文件并使用 `-Xbootclasspath/p` 部署这些类。

`-Xbootclasspath/p` has been removed, its module replacement is the option `--patch-module` to override classes in a module. It can also be used to augment the contents of module. The `--patch-module` option is also supported by `javac` to compile code "as if" part of the module.

`-Xbootclasspath/p` 已被删除，其模块替换是选项 `--patch-module` 来覆盖模块中的类。它也可以用于增强 module 的内容。`--patch-module` 选项也被 `javac` 编译代码，以“仿佛”模块的一部分编译代码。

Here's an example that compiles a new version of `java.util.concurrent.ConcurrentHashMap` and uses it at run-time:

下面是一个编译新版本的 `java.util.concurrent.ConcurrentHashMap` 并在运行时使用它：

```
javac --patch-module java.base=src -d mypatches/java.base \
    src/java.base/java/util/concurrent/ConcurrentHashMap.java

java --patch-module java.base=mypatches/java.base ...
```

More information 更多信息

- The State of the Module System

  模块系统的状态

- JEP 261: Module System JEP 261： 模块系统
- Project Jigsaw 拼图项目

Feedback 反馈

Please send usage questions and experience reports to the jigsaw-dev list. Specific suggestions about the design of the module system should be sent to the JSR 376 Expert Group's comments list.

请将使用问题和体验报告发送至 拼图开发 列表。关于模块系统设计的具体建议 应发送到 JSR 376 专家组的评论列表 。