

User, Sys and Real Times in GC Log

by Grzegorz Mirek  MVB · Sep. 28, 17 · Java Zone · Tutorial

Have you ever wondered what `user`, `sys`, and `real` times in a GC log mean? Well, I have, a couple of times at least. Let's say that we have the following line after Full GC entry:

```
[Times: user=4.21 sys=0.03, real=0.75 secs]
```

To find out what they represent, we should check the UNIX command `time` first. `time` is a utility that executes the command that you put after it and prints time statistics. For example, `time ps` will show running processes together with the info about how much time it took to invoke `ps`.

```
1  94880 ttys017    0:00.19 -bash
2  3574  ttys018     0:00.03 -bash
3  3850  ttys019     0:00.09 -bash
4
5  real  0m0.066s
6  user  0m0.005s
7  sys   0m0.053s
```

As you can see, the output also contains `real`, `user`, and `sys` times. The manual for the `time` command gives a little bit more information about the terms:

These statistics consist of (i) the elapsed real time between invocation and termination, (ii) the user CPU time, and (iii) the system CPU time

- `real` : The user-perceived time it took to execute the command from the start to the end of the call, including time slices used by other processes and the time when our process is blocked (e.g. I/O waiting). The synonyms are wall clock time and elapsed real time.
- `user` : The CPU time that was spent on non-kernel (user-mode) code only within the given process — so other processes and the time when our process is blocked are not included.
- `sys` : The CPU time spent on the kernel (system) code only within the given process. Similarly to `user` time,

other processes and the time when our process is blocked are not included.

When real Time Is Lower Than user + sys Time

Let's look at the GC log example from the top:

```
[Times: user=4.21 sys=0.03, real=0.75 secs]
```

What it basically says is that, in reality, from the user perspective, it took 0.75 seconds to complete. Kernel code was executed for 0.03 seconds and non-kernel code took 4.21 seconds. How is it possible that the `real` elapsed time is lower than `user` time? Well, it's because the `user` code was executed by multiple threads and the statistics print the **sum** of all threads working on it. So basically, multiple GC threads were involved.

When real Time Is Equals user + sys Time

That's a typical situation when the Serial Garbage Collector is used. Because it runs only one GC thread, `user` + `sys` time will be more or less similar to the `real` time.

When real Time Is Greater Than user + sys Times

In certain circumstances, you might see `real` time to be greater than `user` + `sys` time. If it happens often and the differences are substantial, then it might mean that there is either a lack of CPU or there is heavy I/O activity in the system. Lack of CPU is when the application doesn't get enough CPU cycles to run because they are shared among too many processes. The solution might be to add more CPU cores or to reduce the number of the most CPU-consuming processes. Heavy I/O is well explained in this [LinkedIn article](#).

Like This Article? Read More From DZone



When Application Threads Can Be Stopped: Safepoints



Questioning Status Quo - 'Serial GC not for serious applications'



Real-Time OCR for Mobile Apps With RTR SDK



Free DZone Refcard Java Containerization

Topics: JAVA GC , REAL TIME , JAVA , USER TIME , SYS TIME , JAVA PERFORMANCE , TUTORIAL

Published at DZone with permission of Grzegorz Mirek , DZone MVB. [See the original article here.](#) Opinions expressed by DZone contributors are their own.

Building Java 6-8 Libraries for JPMS in Gradle

by Tomasz Linkowski · May 04, 19 · Java Zone · Tutorial

Find out how to use Gradle to build Java 6-8 libraries that support JPMS (Java Platform Module System) by providing Java 9 `module-info.class`.

Introduction

If you need an introduction to JPMS itself, check out this nice overview.

This post is primarily targeted at Java library maintainers. Any such maintainer has to make a choice of which JDK to target:

- Targeting the newest JDKs (JDK 11, or the newly released JDK 12) provides developers and users with access to new APIs and more functionality.
- However, it prevents the library from being used by users who are stuck on older JDKs.
 - And those older JDKs are still **very** popular, taking ~95 percent share in 2018, and predicted to take ~90 percent in 2019. Especially the popularity of JDK 8 (> 80 percent share) makes it a de-facto standard for now.

So, the latter is rightly a *deciding factor* for many library maintainers. For example, vavr 1.0 was intended to target JDK 11, but will ultimately target JDK 8.

Still, it's advisable to add *some* support for JPMS in the hope that it will see wide adoption in future (I'd say 5+ years from now). Stephen Colebourne describes three options here:

1. Do nothing (not recommended).
2. Minimum: add an `Automatic-Module-Name` entry in your `MANIFEST.MF` file.
3. Optimum: add a `module-info.class` targeting JDK 9+ while providing all the remaining classes targeting JDK 6-8*.

Here, we'll delve into how to achieve option three (the optimum).

**I write about JDK 6-8 (and not e.g. JDK 5-8) because, in JDK 11, `javac`'s `--release` option is limited to range 6-11.*

Justification

Before we delve into "how," though, let's skim over the "why."

Why is it worth bothering with JPMS at all? Primarily because JPMS:

- Provides strong encapsulation
- Prevents introducing split packages
- Ensures faster class loading

To sum up, JPMS is *really cool* (more here), and it's in our best interest to encourage its adoption!

So, I encourage the maintainers of Java 6-8 libraries to make the most of JPMS:

- For themselves, by compiling `module-info.java` against the JDK 6-8 classes of its module and against other modules
- For their users, by providing `module-info.class` for the library to work well on `module-path`

Possible Behavior

Location of `module-info.java`

There are two places where `module-info.java` can be located:

1. With all the other classes, in `src/main/java`
2. In a separate "source set," e.g. in `src/main/java9`.

I prefer option 1 because it just seems more natural.

Location of `module-info.class`

There are two places where `module-info.class` can end up:

1. In the root output directory, with all the other classes
2. In `META-INF/versions/9` (Multi-Release JAR, AKA MRJAR)

Having read a post on MRJARs by Cédric Champeau, I'm rather suspicious of MRJARs, and so I prefer option 1.

Note, however, that Gunnar Morling reports having had some issues with option 1. On the other hand, I hope that 1.5 years from the release of JDK 9, all major libraries are already patched to properly handle `module-info.class`.

Example Libraries Per Build Tool

This section contains a few examples of libraries that provide `module-info.class` while targeting JDK 6-8.

Ant

- Lombok (JDK 6 main + JDK 9 `module-info.class`)

Maven

- ThreeTen-extra (JDK 8 main + JDK 9 `module-info.class`)
- Google Gson — not released yet (JDK 6 main + JDK 9 `module-info.class`)
- SLF4J — not released yet (JDK 6 main + JDK 9 `module-info.class` in `META-INF/versions/9`)

Note that Maven Compiler Plugin provides an example of how to provide such support.

Gradle

I haven't found any popular libraries that provide such support using Gradle (please comment if you know any). I only know of vavr trying to do this (#2230).

Existing Approaches in Gradle

ModiText

ModiText (by Gunnar Morling) and its Gradle plugin (by Serban Iordache) have some really cool features. In essence, ModiText generates `module-info.class` without the use of `javac`, based on a special notation or directly from `module-info.java`.

However, in the case of direct generation from `module-info.java`, ModiText effectively duplicates what `javac` does while introducing issues of its own (e.g. #90). That's why I feel it's not the best tool here.

Badass Jar Plugin

Serban Iordache also created a Gradle plugin that lets one "seamlessly create modular jars that target a Java release before 9."

It looks quite nice, however:

- In order to build the proper JAR and validate `module-info.java`, the Gradle build has to be run twice
- It doesn't use `javac`'s `--release` option, which guarantees that only the right APIs are referenced
- It doesn't use `javac` to compile `module-info.java`.

Again, I feel it's not the right tool here.

JpmsGradlePlugin

This is my most recent find: JpmsGradlePlugin by Axel Howind.

The plugin does some nice things (e.g. excluding `module-info.java` from `javadoc` task), however:

- it too doesn't use `javac`'s `--release` option,
- it doesn't support Java modularity fully (e.g. module patching),
- it doesn't feel mature enough (code hard to follow, non-standard behavior like calling `javac` directly).

Proposed Approach in Gradle

Gradle Script

Initially, I wanted to do this by adding a custom source set. However, it turned out that such an approach would

introduce unnecessary configurations and tasks, while what we really need is only one extra task, "hooked" properly into the build lifecycle.

As a result, I came up with the following:

1. Configure `compileJava` to:
 - `exclude module-info.java` ,
 - use `--release 6/7/8` option.
2. Add a new `JavaCompile` task named `compileModuleInfoJava` and configure it to:
 - Include only `module-info.java` ,
 - Use `--release 9` option,
 - Use the classpath of `compileJava` as `--module-path *` ,
 - Use the destination directory of `compileJava` * ,
 - Depend on `compileJava` * .
3. Configure `classes` task to depend on `compileModuleInfoJava` .

The above, expressed as a Gradle script in Groovy DSL, can be found in this Stack Overflow answer of mine.

These three steps are necessary for `compileModuleInfoJava` to see classes compiled by `compileJava` . Otherwise, `javac` wouldn't be able to compile `module-info.java` due to unresolved references. Note that in such configuration, every class is **compiled just once (unlike with the recommended Maven Compiler Plugin configuration).*

Unfortunately, such configuration:

- Is not easily reusable across repositories,
- Doesn't support Java modularity fully

Gradle Modules Plugin

Finally, there's a plugin (Gradle Modules Plugin) that adds full support for JPMS to Gradle (created by the authors of *Java 9 Modularity*, Sander Mak and Paul Bekker).

This plugin only lacks support for the scenario described in this post. Therefore, I decided to:

- File a feature request with this plugin: #72
- Provide a Pull Request with a **complete** implementation of #72 (as a "proof of concept"): #73

I tried hard to make these high-quality contributions. The initial feedback was very welcome (even Mark Reinhold liked this!). Thank you!

Now, I'm patiently waiting for further feedback (and potential improvement requests) before the PR can be

(hopefully) merged.

Summary

In this post, I've shown how to build Java 6-8 libraries with Gradle so that `module-info.java` is compiled to JDK 9 format (JPMS support), while all the other classes are compiled to JDK 6-8 format.

I've also recommended using Gradle Modules Plugin for such configuration (as soon as my PR gets merged and a new plugin version gets released).

EDIT (Apr 4, 2019): Gradle Modules Plugin v1.5.0, which includes my PR, has been released!

Like This Article? Read More From DZone



How to Create Modular Jars That Target a Java Release Before 9



How to Bootstrap Your Open-Source Projects




Gradle Goodness: Running a Single Test



Free DZone Refcard Java Containerization

Topics: JAVA, GRADLE, GROOVY, JDK, JPMS, OPEN-SOURCE, LIBRARIES

Published at DZone with permission of Tomasz Linkowski . [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.

Introduction to Bridge Design Pattern [Video]

by Ram N · May 03, 19 · Java Zone · Tutorial

In the video below, we take a closer look at a bridge design pattern in Java. Let's get started!



Like This Article? Read More From DZone



Factory Design Pattern [Video]



Front Controller Design Pattern [Video]



Chain of Responsibility Design Pattern [Video]



**Free DZone Refcard
Java Containerization**

Topics: DESIGN PATTERN, BRIDGE DESIGN PATTERN, DESIGN PATTERN TUTORIAL, DESIGN PATTERNS FOR BEGINNERS, JAVA, VIDEO, TUTORIAL

Opinions expressed by DZone contributors are their own.