Liberica JDK
Using jcmd locally, containerized, and remotely

Liberica JDK
Revision 1.0
October 17, 2023

bellsoft

BellSoft software contains open source software. Additional information about third party code is available at https://bell-sw.com/third_party_licenses. You can also get more information on how to get a copy of source code by contacting info@bell-sw.com.

THIS INFORMATION MAY CHANGE WITHOUT NOTICE. TO THE EXTENT PERMITTED BY APPLICABLE LAW, BELLSOFT PROVIDES THIS DOCUMENTATION “AS IS” WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL BELLSOFT BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF BELLSOFT IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in this document is governed by the applicable license agreement, which is not modified in any way by the terms of this notice.

Alpaquita, Liberica and BellSoft are trademarks or registered trademarks of BellSoft Corporation. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Java and OpenJDK are trademarks or registered trademarks of Oracle and/or its affiliates. Other trademarks are the property of their respective owners and are used only for identification purposes.

Contents

1. Introduction

jcmd is the command line executed with JVM diagnostic tools shipped with OpenJDK. Before the introduction of jcmd, there were multiple tools to run **live** and **postmortem** diagnostics for JVM. jcmd became a to-go utility for **live** diagnostics of JVM executed from the command line (e.g., dumping threads or inspecting JVM configuration). The postmortem diagnostics are performed with other tools.

There exists a wide range of tools in the JVM ecosystem, including graphical and web, so why use the command line tools? There are a few reasons:

- Command line tools can be executed with a shell / SSH terminal, and an SSH terminal could be the only secure way to work effectively with the JVM.
- Command line tools from JDK do not need upfront configuration and rely on OS access control for security;
- Command line tools are convenient for scripts and other means of automation you may introduce.

2. Common tasks for JCMD

jcmd displays a long list of commands (which is included at the end of the document), and a lot of them work on the low level of programming. Here is an overview of operations available via jcmd:

- **Capturing thread and heap dumps** — this is a pretty common task for Java engineers. Such operations were available long before jcmd was introduced and were utilized by means of the dedicated tools jstack and jmap.
- **Control of flight recorder** — jcmd has multiple commands to start/stop/dump JFR (JDK Flight Recorder) sessions.
- **Inspection of various aspects of JVM configuration** — with jcmd, you can check the HotSpot JVM runtime setting, system properties, and command line parameters.
- **Control of the JMX socket at runtime** — using jcmd, you can open a socket accepting JMX connection without JVM restart. This is useful if you want to use GUI tools such as Mission Control while avoiding configuring the JMX upfront.

3. Example of jcmd usage

Next, we will overview the typical jcmd workflow with examples.

Usually, jcmd is added to PATH when installing OpenJDK. If, for some reason, it does not exist in your path, you can find binary under bin at your OpenJDK installation directory.

The first thing jcmd requires is the PID of the JVM process you want to work with. There are multiple ways to get the PID of a JVM, and one of them is to list locally running JVM processes with jcmd itself.

Executing jcmd without parameters will list locally running JVMs and their respective PIDs.

```
jcmd
23876 sun.tools.jcmd.JCmd
32311 HelloJDK
```

As you see, jcmd includes itself in the list, because it is implemented in Java.

Once you've got a PID, you can list commands exposed by the JVM.

```
jcmd 32311 help
VM.unlock_commercial_features
JFR.configure
JFR.stop
JFR.start
JFR.dump
JFR.check
VM.native_memory
ManagementAgent.stop
ManagementAgent.start_local
ManagementAgent.start
VM.classloader_stats
GC.rotate_log
Thread.print
GC.class_stats
GC.class_histogram
GC.heap_dump
GC.finalizer_info
GC.heap_info
GC.run_finalization
GC.run
VM.uptime
VM.dynlibs
```

```
VM.flags
VM.system_properties
VM.command_line
VM.version
help
```

The list of commands may vary between versions of JDK.

Analyzing a thread dump is a common procedure for getting a quick diagnosis. You can execute the `Thread.print` command for that.

```
jcmd 32311 Thread.print
32311:
2021-10-06 20:25:43
Full thread dump OpenJDK 64-Bit Server VM (25.302-b08 mixed mode):
"Attach Listener" #9 daemon prio=9 os_prio=0 tid=0x00007f34c4001000 nid=0x5dbc
waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"Service Thread" #8 daemon prio=9 os_prio=0 tid=0x00007f35040d1000 nid=0x7e44
runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"C1 CompilerThread2" #7 daemon prio=9 os_prio=0 tid=0x00007f35040c3800
nid=0x7e43 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"C2 CompilerThread1" #6 daemon prio=9 os_prio=0 tid=0x00007f35040c2000
nid=0x7e42 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"C2 CompilerThread0" #5 daemon prio=9 os_prio=0 tid=0x00007f35040bf000
nid=0x7e41 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"Signal Dispatcher" #4 daemon prio=9 os_prio=0 tid=0x00007f35040bc000
nid=0x7e40 runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"Finalizer" #3 daemon prio=8 os_prio=0 tid=0x00007f3504088000 nid=0x7e3f in
Object.wait() [0x00007f34efaf9000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x000000076e208ee0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:144)
        - locked <0x000000076e208ee0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:165)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:216)
...
```

Another typical action is dumping a heap of JVM.

```
jcmd 32311 GC.heap_dump `pwd`/dump.hprof
32311:
Heap dump file created
```

The command above will create a "dump.hprof" file in the current directory. You can open this file with VisualVm or Eclipse Memory Analyzer Tool for detailed analysis.

Notice that we provided the absolute path (via pwd command). The relative path is interpreted from the working directory of JVM, which is not the best place for heap dump creation.

If you struggle to recall arguments for a specific command, you can use built-in help.

```
jcmd 32311 help GC.heap_dump
32311:
GC.heap_dump
Generate a HPROF format dump of the Java heap.
Impact: High: Depends on Java heap size and content. Request a full GC unless
the '-all' option is specified.
Permission: java.lang.management.ManagementPermission(monitor)
Syntax : GC.heap_dump [options] <filename>
Arguments:
        filename :  Name of the dump file (STRING, no default value)
Options: (options must be specified using the <key> or <key>=<value> syntax)
        -all : [optional] Dump all objects, including unreachable objects
(BOOLEAN, false)
        -gz : [optional] If specified, the heap dump is written in gzipped
format using the given compression level. 1 (recommended) is the fastest, 9 the
strongest compression. (INT, 1)
```

An excellent addition in OpenJDK 11 and above is the `-gz` option that helps to make a compressed heap dump. Heap dumps compression is usually effective, saving both space and disk IO.

When developing and running Java applications on your desktop, jcmd works excellent. The main issue is running jcmd as the same user as the JVM you are connecting to. jcmd and other JDK tools work seamlessly on all supported platforms.

But you often have to run software in the containers or on some kind of remote host. jcmd is essential for ad hoc diagnostics of Java applications, and the ability to use it in a variety of circumstances is vital.

Running a JVM in a container is pretty standard these days, so let's see how jcmd deals with containers.

jcmd and containerized JVM

If you get a fresh Liberica JDK 17 image and use it with docker, jcmd will very likely work as expected (you will need to run jcmd as root). Although it was not the case not so long ago, JDK made tremendous progress with the native support of Linux containers over the last few years.

But let's say something goes wrong, and you need to take a thread dump. What's next?

What if jcmd does not work?

To solve some common issues, check a few things first.

Are you using Linux? Linux containers run only on Linux OS. If you utilize macOS or Windows, your container is actually running in Linux VM under hypervisor on your host OS. jcmd uses IPC and cannot access processes running under different OS.

Are you running jcmd as root? jcmd should be able to connect to the target JVM using IPC to function. That means it should either run under the same uid or as root.

What version of Java are you running? When you execute jcmd from the console, you are using the system default JDK installation. Verify its version with java `-version` command. You need to have JDK 11 or higher on your host to work with containers. Your container could use an older version of JDK, although jcmd will still be able to see it.

What version of Linux kernel do you have? You need at least a 4.1 version of the Linux kernel because it is the release in which some container-related data in procfs was introduced, and it is essential for the functionality of jcmd.

These are the most common reasons jcmd could not be working.

What if you still need to use jcmd?

There are several options available in this situation.

Run jcmd inside the container. This is the most straightforward way. If you can open the terminal in a container and run jcmd there, it will work. The caveat here is that jcmd could be missing in the container environment.

Run jcmd on the host Linux VM. If you are on Windows or macOS, try opening the terminal in the Linux VM hosting your container and run jcmd there.

Execute command remotely with JMX. It is possible to execute a jcmd command remotely with JMX. While it requires upfront configuration, it could be a good option once you have finished with local development and start moving things to the target platform.

4. JVM attach protocol under Linux

There are two protocols implemented in jcmd — discovery and commands.

Discovery

Once started, JVM automatically creates a file in /tmp/hsperfdata_/ (id does the same for non-Linux OSes, though the path may be different).

Older versions of jcmd were scanning /tmp/hsperfdata to get the list of PIDs. That does not work well with containers as /tmp in the container may not be the same as /tmp of the host. Starting with JDK 11, jcmd scans the list of processes and checks the presence of the hsperfdata file for each process (using respective user and pid). Thanks to the new approach, jcmd running under root can now see JVM from any user regardless of container boundaries.

Commands

jcmd uses a Unix domain socket to establish a connection to the target JVM. JVM should open a socket for jcmd to connect. jcmd may trigger creating a socket by sending a QUIT signal to target JVM (your JVM will not shut down). Socket is created under the current directory of target JVM (or under /tmp) and named .java_pid.

That socket uses a simple text protocol, jcmd sends a command and receives some output and status code back.

As often occurs with Linux containers, things are not what they seem. The same JVM process may have different PIDs (e.g., PID 1 in a container and some 12345 on host OS). This was fixed for jcmd in JDK 11, and now the container boundary is not a barrier for this tool.

5. jattach — a light alternative to jcmd

jattach is a compact tool written in C that implements the protocol described above.

jattach is very handy if jcmd is not available for some reason, such as you have to use stripped-down JRE instead of JDK. Unlike jcmd, which you cannot copy as a binary, jattach is a standalone binary of just a few kilobytes. It is easy to download and run when you need it.

With `jattach "jcmd …"` you can enjoy the full functionality of jcmd from JDK. Note that you need to input the whole command starting with jcmd in quotes.

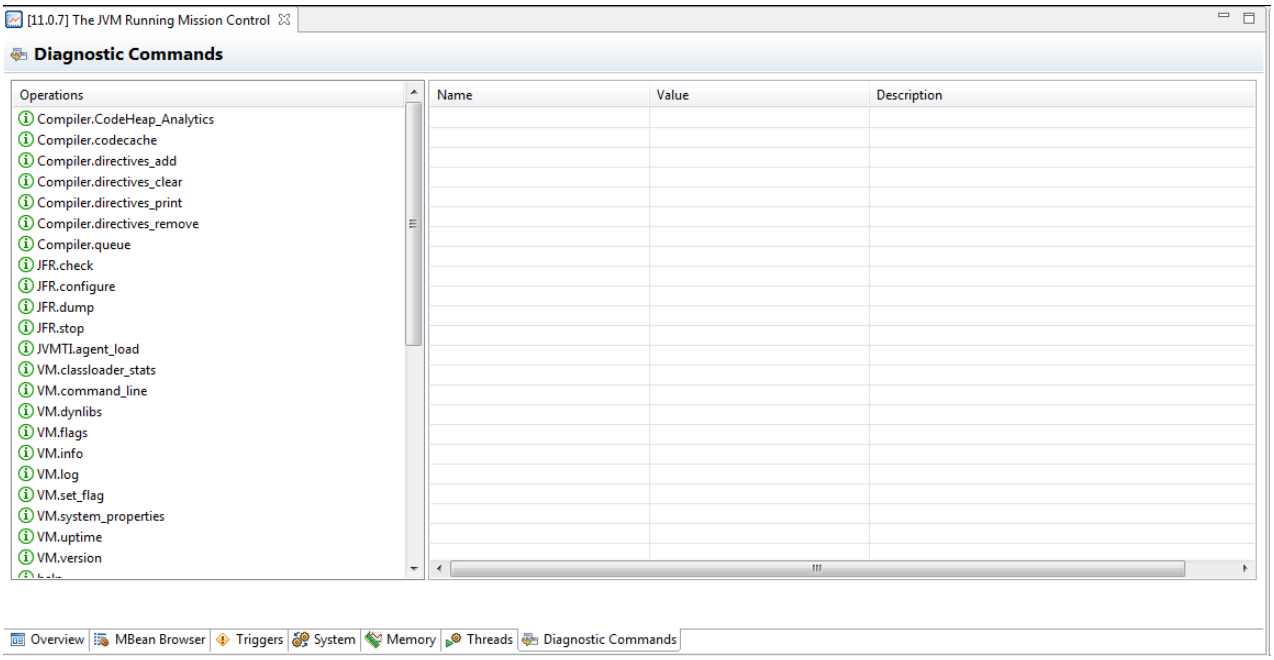
6. Sending jcmd commands via JMX

We mentioned the lack of need for upfront configuration as one of the advantages of jcmd. Sometimes though, JMX access is available while terminal access is not.

In this case, you cannot use jcmd, but precisely the same list of commands is available via JVM.

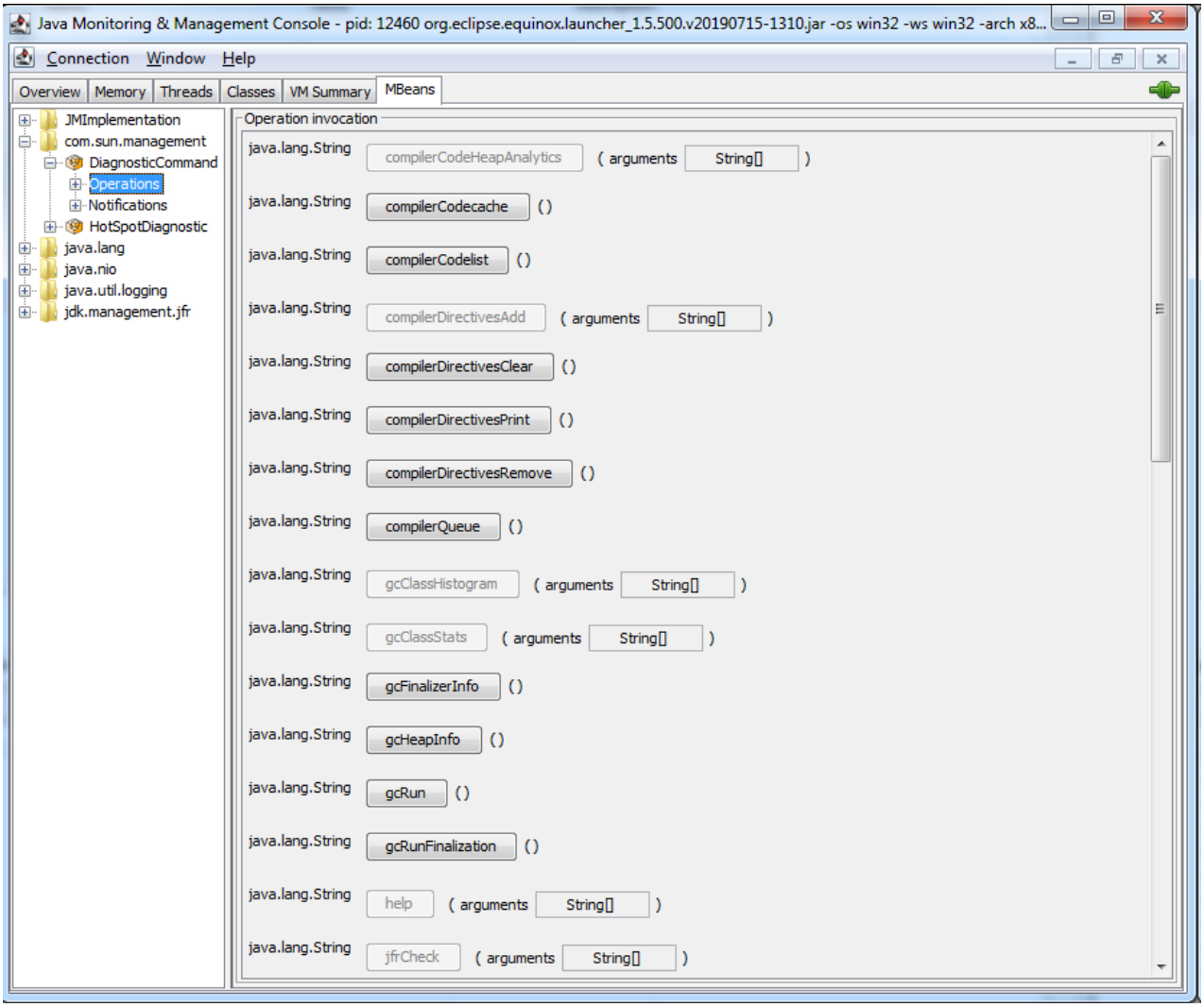
We suggest using Liberica Mission Control.

Execute Mission Control, connect to the remote process with JMX Console and switch to the **Diagnostics Commands** tab.



As you see, the very same list of commands from `jcmd` is present there.

Commands are exposed as MBean operations, so they are available for other tools too.



The "typical" JConsole. Operations are available under `com.sun.management/DiagnosticCommand` MBean.

Unfortunately, JConsole cannot deal with argument types, so only the operations without parameters can be invoked.

SJK is another open source tool for JVM diagnostics. It is capable of invoking any MBean operations through a JMX connection. If you need to invoke the jcmd command from the terminal on a remote JVM, you can use SJK.

```
java -jar sjk.jar jcmd -s host:port help
The following commands are available:
VM.unlock_commercial_features
JFR.configure
JFR.stop
JFR.start
JFR.dump
```

```
JFR.check
VM.native_memory
VM.classloader_stats
GC.rotate_log
Thread.print
GC.class_stats
GC.class_histogram
GC.finalizer_info
GC.heap_info
GC.run_finalization
GC.run
VM.uptime
VM.dynlibs
VM.flags
VM.system_properties
VM.command_line
VM.version
help
```

For more information about a specific command use `help <command>`.

7. Accessing dump files

Commands that produce files do that relative to the JVM working directory. If a JVM is running in a container, then the container's FS will be used. If you need to take a heap dump of a JVM running in a container, it is better to use a path mounted from the host as a filename for the dump file.

If no good mount is available, you can fall back to /proc/PID/root/ mount point to access the container's file system from the host. Note that it will not help if the container is running in a VM or remote host.

8. A complete list of jcmd commands

While the list of commands supported by jcmd is long, many are more useful for JVM engineers than application developers. In any case, we would like to point out all of them.

`Compiler.CodeHeap_Analytics` (available since JDK 11)

Dumps detailed information about the code produced by the JIT compiler, including addresses of compiled code blocks.

`Compiler.codecache` (available since JDK 11)

Prints code cache layout and bounds.

`Compiler.codelist` (available since JDK 17)

Prints all compiled methods in the code cache that are alive.

`Compiler.directives_[add, clear, print, remove]` (available since JDK 11)

Helps you to edit the JIT compiler option on flight.

`Compiler.perfmap` (available since JDK 17)

Native Linux profilers expect debug symbol information for binary to be available in a specific format under /tmp/perf-.map. Without symbol file addresses from the stack, a trace cannot be translated to meaningful method names. With JDK 17 release, that functionality is built into JVM and exposed via jcmd.

`Compiler.queue` (available since JDK 11)

Prints methods queued for compilation.

`GC.class_histogram`

This command walks heap content (beware of Stop-the-World pause here) and dumps statistics aggregated by classes. It is valuable for spotting obvious memory hogs without firing up a full-featured heap analyzer tool.

`GC.class_stats` (discontinued after JDK 11)

Similar to the command above, but shows more statistics.

`GC.finalizer_info`

Prints info about heap objects pending for finalization.

`GC.heap_dump`

Dumps content of JVM heap into a file for further analysis.

`GC.heap_info`

Prints details of heap memory spaces.

`GC.run`

Same as java.lang.System.gc()

`GC.run_finalization`

Same as java.lang.System.runFinalization()

`GC.rotate_log` (discontinued after JDK 8)

Triggers GC log rotation. Requires specific GC log configuration.

`JFR.*`

jcmd allows you to control JDK Flight Recorder sessions. You can start and stop recording sessions and dump results to file from the terminal with jcmd. JDK has another tool — jfr, which can help analyze the content of JFR recording without ever leaving the terminal.

`JVMTI.*` (available since JDK 11)

This command could be used to enable JVMTI profiling agent without JVM restart.

`ManagementAgent.*`

A handy set of commands, which enables the JMX port without restarting JVM. Typically, JMX port should be configured via JVM start up command. With jcmd, you can start the JMX port when you need it and start using JMX-based tools such as Mission Control, even if the JVM is not configured upfront.

`Thread.print`

Produces a JVM thread dump.

`VM.cds` (available since JDK 17) CDS (Class Data Sharing) is an old Java feature. Usually, shared class data archives are generated at JVM exit, if all required XX options were specified at the start. jcmd offers an alternative way to create such archives.

`VM.classloader_stats`

Prints statistics about all ClassLoaders.

`VM.classloaders` (available since JDK 11)

Prints a hierarchy of classloaders. Optionally can include a list of classes. This command could be very helpful for troubleshooting the "A cannot be cast to A" problem.

`VM.command_line`

Prints all arguments of JVM start command.

`VM.dynlibs`

Prints list of loaded native dynamic libraries.

`VM.events` (available since JDK 17)

Prints latest VM events. Events are produced by various JVM subsystems and may be useful for troubleshooting JVM issues.

`VM.flags`

Prints effective values of HotSpot JVM XX options.

`VM.info` (available since JDK 11)

Prints details similar to hserror crash dump, but without crashing the JVM.

`VM.log` (available since JDK 11)

This command allows changing the configuration for JVM logging.

`VM.metaspace` (available since JDK 11)

This command can be used to retrieve information about metaspace configuration and content.

`VM.native_memory`

This command works together with JVM native memory tracking. Tracking should be enabled on JVM startup.

`VM.print_touched_methods` (available since JDK 11)

Prints all methods that have ever been touched during the lifetime of this JVM. Requires

`-XX:+LogTouchedMethodsoptions`.

`VM.set_flag` (available since JDK 11)

The command allows you to modify manageable XX options of Hotspot JVM.

`VM.stringtableand` and `VM.symboltable` (available since JDK 11)

Prints statistics for string and symbol tables.

`VM.system_properties`

Dumps properties from `java.lang.System.getProperties()`.

`VM.systemdictionary` (available since JDK 11)

Prints statistics for system dictionaries per class loader.

`VM.unlock_commercial_features` (discontinued after JDK 8)

Rudimentary option for compatibility with older JDK. Noop in OpenJDK.

`VM.uptime`

Prints uptime of JVM.

`VM.version`

Prints JVM version details.

`PerfCounter.print`

This is a "secret" option, not listed in help, and not available via JVM. This command prints the content of the perf counter accumulated by the JVM. Unlike other jcmd commands, it does not require connecting to the JVM. Perf counters are exposed by JVM via /tmp/hsperfdata_/ file.

9. Considerations

jcmd is a simple tool with a command line interface. You can use it for troubleshooting tasks, such as taking thread and heap dumps, introspecting JVM configuration, controlling flight recorder, and some others.

Starting with JDK 11, jcmd is built with excellent support for Linux containers. In the case of using the containers, there are few key points to keep in mind:

- If you are on Windows or macOS, containers are running on Linux VM, and you have to run jcmd on the same VM for it to work.
- You usually need to run jcmd as root.
- jcmd from the host can see JVM processes in the container, but if launched within the container, jcmd will not see processes outside its environment.

If you do not have jcmd installed on the host or in the container, jattach is a reliable and lightweight drop-in replacement that can execute the same commands.

Liberica JDK
Using jcmd locally, containerized, and remotely

bell//soft