JEP: Post-mortem crash analysis with jcmd

JDK-8328351

Kevin Walls

HotSpot Serviceability

Java Platform Group, Oracle

October, 2025

Serviceability Tools

The ability to monitor, observe, debug, and troubleshoot

- `JDK/bin/jcmd`
  - Local attach, commands for monitoring and troubleshooting
    - Thread.print, GC.heap_dump, Compiler.CodeHeap_Analytics, …
- JMX
  - Local and remote. Gather stats and interact. JConsole, JMC.
- “Debugging”
  - JDI/JVMTI. Underneath your IDE.
- Logging
- JFR
  - Live monitoring/recording

JFR: JDK Flight Recorder

- Event recording, analogous to the aircraft device
- Saves event history for time period leading up to a problem
- Very low overhead, designed for production environments
- Built-in events for JVM, JDK, and OS
- Helpful for discovering latencies
- APIs for producing and consuming events

Live Serviceability Tools

- There is overlap, information available via multiple routes.
  - Your entry point may vary
  - Duplication may not always help.
  - Are features discoverable?

- Rich area for the JDK/JVM
  - It's a virtual machine, should expect such features.

When things go wrong

Crashes

Not a crash (for this discussion):

```
Exception in thread "main"
java.lang.RuntimeException: ...problem...
        at App.run(App.java:16)
        at App.main(App.java:23)
```

This is a crash:

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00007f55c434f238 (sent by handshake
timeout handler), pid=640414, tid=640414
#
# JRE version: Java(TM) SE Runtime Environment (26.0) (fastdebug
build 26-internal-2025-09-20-1621136.kwalls...)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (fastdebug 26-
internal-2025-09-20-...
# Problematic frame:
# C  [libc.so.6+0x87238]  __futex_abstimed_wait_common+0xc8
#
# Core dump will be written. Default location: ...
```


Crash dumps

Things may look less rich at this point

- In extreme scenarios...
- There is somewhat less help than live:
  - hs_err_pidXXX.log
    - Written by JVM
    - Text file
  - Core file or Minidump (Windows)
    - Written by the OS
    - Process memory image and other context

Native debuggers

- Tools (e.g. gdb, WinDbg, …) that understand native programs
  - Read and interpret the core file
  - Understand libraries, symbols, addresses. Examine code, memory, etc…
- Our Virtual Machine is not the machine they understand
  - Can explain the native JVM
  - Cannot give a Java-level view of application state
- Scripting:
  - Extract a method name or symbol, OK.
  - Heap dump…?
    - Heroic efforts. Maintenance.

```
(gdb) backtrace
...
#12 0x00007fe1e4546735 in
oopFactory::new_objArray (...)
...
#14 0x00007fe1cbba9f5a in ?? ()
#15 0x0000000000000099 in ?? ()
#16 0x00007fe1cbba9ee2 in ?? ()
#17 0xfffffffffffffff7 in ?? ()
#18 0x00007fe19c4013fd in ?? ()
...
```

Don't we already have this kind of help?

Serviceability Agent (SA)

- Exposes the Java context where a native debugger does not
- Live or from a core file.
  - Thread dumps, object inspection, heap dump...
- Niche adoption?
  - Invocation changed significantly over the years
    - launcher scripts, binaries, or just find sa-jdi.jar and invoke
    - `jstack` with an SA option, or `jhsdb` with a jstack option
  - “Experimental and unsupported” label

```
src/hotspot/share/runtime/thread.hpp:
class Thread: …
```

```
src/jdk.hotspot.agent/…/Thread.java:
public class Thread …
```

- SA duplicates in Java, that which the JVM does natively
- Significant burden to keep the SA in sync with JVM. e.g. scanning first half of 2025:
  - >=10 JVM changes causing test failures, causing more work
  - >=14 Other SA fixes, or changes that involve an SA update

Tool duplication can be confusing

- Multiple routes:
  - `jcmd` on live JVM and SA on live JVM have overlap.
    - e.g. Live thread dump from `jcmd` (native VM) or SA
    - Which do we recommend, and maintain?

- Mixed message for users and ourselves
  - Issue for development, maintenance, documentation.
  - Get me a Thread dump...
  - Where's the Thread dump code...

- jcmd has become the go-to VM tool
  - after earlier inventing separate launchers jstack, jmap, etc..

We should have jcmd on core files

- `jcmd PID Thread.print` ..etc...
  - We should have this for crashes: `jcmd corefile Thread.print`
  - Many jcmds relevant on a crash dump
  - jcmd commands are added as needs arise
    - Namespace of commands

- We have code that understands the JVM
  - The JVM already understands itself
  - jcmd attaches live, VM's response is to invoke Diagnostic Commands natively in itself
    - Native JVM code exists for Thread dumper, Heap dumper, etc...

- Unification:
  - User interface: jcmd for live or deceased JVMs
  - Code: avoid duplication and JVM sync/maintenance issues

Process Revival

Native debuggers “open” or “interpret” core files, but…

- To enable jcmd live and on core files
  - We want to run the existing native JVM code on a core file, unchanged
- We need
  - the JVM code loaded
  - the JVM memory loaded
  - find and execute diagnostic code
    - With restored data from process, everything will make sense…

```
$ jcmd core.734636 Thread.print
…
"Reference Handler" #15 [734649] daemon prio=10 os_prio=0 cpu=-0.00ms elapsed=26.31s tid=0x00007fd72816e410 nid=734649 waiting on
condition  [0x00007fd6f71f0000]
   java.lang.Thread.State: RUNNABLE
Thread: 0x00007fd72816e410  [0xb35b9] State: _running _at_poll_safepoint 0
   JavaThread state: _thread_blocked
        at java.lang.ref.Reference.waitForReferencePendingList(java.base@26-internal/Native Method)
        at java.lang.ref.Reference.processPendingReferences(java.base@26-internal/Reference.java:241)
        at java.lang.ref.Reference$ReferenceHandler.run(java.base@26-internal/Reference.java:203)
```

Revival Details

- jcmd (Java) creates new live native helper process
- Map in memory from core file
- Load JVM - at same address as previously loaded
  - Fix JVM load address (like “prelink” on some Linux)
- Call a new JVM helper function to reset some JVM state
  - References to system libraries: e.g. libc time functions
- A few other JVM modifications to accommodate revival
  - GC before Heap Dump? No!
  - Use GC worker threads for Heap Dump? No!

- Call known JVM diagnostic code
  - `DCmd::parse_and_execute`

Limitations

- JVM code works, can inspect itself, Java heap, etc...
- **Must** use exact same JVM binary as crash
  - As per regular core file analysis
- Not planned to run Java code
  - jcmds that call Java, not available
    - Some (most?) of which not relevant after a crash
- JVM is loaded in a live process, but not "running"
  - Application and VM Threads do not exist in OS terms
    - e.g. Compiler threads, GC workers
  - But VM diagnostic command code can inspect memory
    - Including Java heap and thread stacks

Other features

- Transported core files
  - `jcmd` must be given the correct JVM library
  - Enable support to have sensible conversations

- Revival has fewer requirements than regular native debugger
  - But not a direct replacement for gdb, WinDbg, ...

- `hs_err_pidXXX.log` provides native backtrace of crash

- Discoverability
  - jcmd help

Future work

- Everything is in-progress
  - Working on tidying up to share more widely
- Additional jcmd:
  - VM.inspect to examine arbitrary Java/JVM Objects
- Will approximate SA functionality with lower maintenance
  - Not direct command-for-command replacement

Post-mortem crash analysis with jcmd

- https://openjdk.org/jeps/8328351

- Goals:
  - Make the troubleshooting of crashed JVMs as familiar and productive as troubleshooting live JVMs
  - Enable post-mortem diagnostics on Linux and Windows.
  - Reduce the future cost of JDK maintenance by focusing serviceability work on jcmd

Thank you


ORACLE


ORACLE