# Debugger Support

## Enumerating Threads

Virtual threads are just objects in the heap, there may be millions of them. There is no API support in the debugger API for enumerating all virtual threads.

JDI VirtualThreads.allThreads() (and JDWP VirtualMachine/AllThreads) enumerates all live platform threads, virtual threads are not enumerated unless the JDWP agent is started with the includevirtualthreads option (see below).

For thread dumps and troubleshooting purposes the debugger can invoke the HotSpotDiagnosticMXBean.dumpThreads API in the target VM to generate a thread dump. This can be used as stop-gap solution until there is better support for finding virtual threads in the debugger APIs.

## Thread groups

Virtual threads are not members of a thread group.

JDI ThreadGroupReference::threads (and JDWP ThreadGroupReference/Children) enumerate all live platform threads in a group, virtual threads are not enumerated.

## ThreadStart/ThreadEnd events

JDI ThreadStartEvent/ThreadDeathEvents are sent for all threads when enabled. This may impact performance if there are a huge number of virtual threads.

JDWP EventRequest/Set defines a new *PlatformThreadsOnly* filter that can be used when requesting THREAD_START and THREAD_END events. This allows these events to be filtered for virtual threads so they are not sent to the front-end/debugger.

JDI ThreadStartRequest/ThreadDeathRequest define a new method to control whether thread start/end events are sent for all threads or only platform threads.

## IsVirtual

JDI ThreadReference defines isVirtual() to test if a thread is a virtual thread.

JDWP ThreadReference/IsVirtual is the equivalent.

## Testing if target VM supports virtual threads

The recommend way is to check if the target VM is version 19 or later. There is no way to determine if *--enable-preview* was set, but this is also unnecessary. The JDK 19 debug agent will still execute commands that are virtual thread specific, even if the JVM was not run with --enable-preview.

## Not Supported

The following are not currently supported for virtual threads:

- JDI ThreadReference.stop
- JDI ThreadReference.popFrame
- JDI ThreadReference.forceEarlyReturn
- JDI StackFrame.setValue() has limited support. It can be used on the topmost frame when suspended due to certain events, such as breakpoint and single step. Details are still being refined.

## JDWP agent options

As a temporary solution to allow existing debuggers to work with virtual threads, the JDWP agent will track virtual threads so they can be enumerated for debuggers that want to enumerate all virtual threads. The option that controls this behavior is:

| Option Name and Value | Description | Default |
|---|---|---|
| includevirtualthreads=y\|n | List of all threads includes virtual threads as well as platform threads | n |
| ~~notifyvthreads=y\|n~~ | ~~send THREAD_START/END events for all virtual threads~~ | ~~y~~ |

- *includevirtualthreads* control whether or not virtual threads are included in the list of threads returned by JDWP *VirtualThread.GetAllThreads*. This flag can be turned on for compatibility purposes. This flag may eventually go away and the debug agent will operate as if it was set to 'n'. Note that the debug agent will only return virtual threads that are created after the debugger attaches. If the debugger detaches, the debug agent will forget about all discovered virtual threads, and the list will once again be empty when the debugger attaches again.
- [*notifyvthreads* has been removed. To disable *THREAD_START* and *THREAD_END* events for vthreads, use the *PlatformThreadsOnly* filter as described in this section] *notifyvthreads* controls whether or not *THREAD_START* and *THREAD_END* events are sent for virtual threads. The

default is to send them. The purpose is to allow debugger writers to use *notifyvthreads=n* to see how the debugger works without getting these events. This is meant as a short term convenience flag rather than having to set the *PlatformThreadsOnly* filter as described above. Once support for *notifyvthreads=n* is gone, if debuggers don't want *THREAD_START* and *THREAD_END* events for virtual threads, they will need to set the *PlatformThreadsOnly* filter. When using the *PlatformThreadsOnly* filter, the debugger can learn about virtual threads when events are delivered for them, such as a *BreakpointEvent*. Since no *THREAD_END* event will be sent for the virtual thread, the debugger will then need to figure out how to "forget" about the virtual threads it is tracking when they have terminated. This could possibly be done with a *ThreadDeathRequest* that filters on the virtual thread. This would need to be done for each virtual thread being tracked by the debugger. Please note that *notifyvthreads=y* will not override any *PlatformThreadsOnly* filter that is in place.

## Issues with too many virtual threads hitting breakpoints

Each running virtual thread is mounted on a carrier thread. There are a limited number of carrier threads (usually defaulting to the number of available cores). When a virtual thread hits a debugger breakpoint, it pins its carrier thread, preventing any other virtual thread from running on it. If you have an application with a large number of virtual threads, and you setup a breakpoint that many virtual threads will hit, you can wind up in a situation where every carrier thread is running a virtual thread that is at a breakpoint. No virtual threads will make progress when this happens. Those that are mounted are all at breakpoints, and those that unmounted have no carrier thread to run on. If you resume one of the virtual threads, when it yields it will allow unmounted virtual thread to run, which likely itself will then hit this same breakpoint, leaving back to having all virtual threads being stuck.

If you single step in one of these virtual threads that are at a breakpoint, single stepping will work until you step over a call that can lead to the virtual thread yielding. At this point the virtual thread is umounted and a new one can run on the carrier thread (once again, likely quickly hitting the same breakpoint). It will appear that the single step has failed to ever complete. In reality it is waiting for the yielding virtual thread to be remounted and start running again. This can't happen until a carrier thread is freed up. Thus you'll find that you need to start resuming various virtual threads that are at breakpoints until eventually one of them yields AND its carrier thread is used to mount and run the single stepping virtual thread. There's no telling how many virtual threads at breakpoints you will need to resume before this happens.

It is possible to avoid getting into this situation. The first thing to do is to setup the breakpoint so that it suspends all threads rather than just the event thread. This way you'll never have more than one virtual thread hitting the breakpoint at the same time. If you do this, you also need to make sure to setup the debugger's single stepping "thread resumption policy" to resume all threads, rather than just the single stepping thread. If you take these two steps, breakpoints and single stepping should work much better when dealing with a large number of virtual threads that can hit the same breakpoint.

## Links

Project Loom Early Access builds

Java Debug Interface (JDI)

JDWP protocol details

JVM Tool Interface

Java Native Interface

Slides from March 24, 2021 meeting with IntelliJ and Eclipse maintainers

Sample application (uses Helidon MP configured to run each service in its own virtual thread)

Eclipse Bug 527000 tracks adding support for virtual threads