

Verbose garbage collection logs

Garbage collection (GC) reclaims used memory in the Java™ object heap for reuse. During cleanup of the heap, the verbose GC logs, when enabled, capture information about the different GC operations that are involved in the GC cycles. GC operations aim to reorganize or reclaim memory.

Verbose GC logs contain information about GC operations to assist with the following actions:

- Tuning GC and improving application performance.
- Troubleshooting GC operations and policies. For example, analyzing long pauses, or determining how free memory is divided in the Java object heap before and after a GC cycle.

Verbose GC logs, when enabled, begin capturing information as soon as GC is initialized.

To help you visualize and analyze the GC, you can feed verbose GC log files into various diagnostic tools and interfaces. Examples include tools such as [Garbage Collection and Memory Visualizer \(GCMV\)](#) and online services such as [GCEasy](#).

For examples of log output, including guidance on how to analyze the logs, see [Log examples](#).

A further diagnostic step is to run one or more traces on GC activity by using the `-Xtgc` option. Trace output provides more granular information to help diagnose GC problems or perform finer tuning.

How to generate a verbose GC log

You can enable verbose GC logs by specifying the `-verbose:gc` standard option when you start your application. For more information, see [standard command-line options](#).

The output of `-verbose:gc` is printed to STDERR by default. To print the log output to a file, append the `-Xverbosegclog` option. You can also use this option to print to a succession of files, where each file logs a specified number of GC cycles.

Verbose GC log contents and structure

The verbose GC logs are printed in XML format and consist of the following sections:

- A summary of your GC configuration, which is captured in the `<initialized>` XML element.
- Information about the GC cycles that ran, including GC operations and GC increments.

For definitions of GC cycles and operations, see [Garbage collection](#). For definitions of GC increments, see [GC increments and interleaving](#).

The logs record when GC cycles and their increments start and end, and list the GC operations that run within these increments to manage or reclaim memory. You can also determine which type of event triggered the cycle or increment, and the amount of memory available to your application before and after processing.

Initialization

The log begins by recording the configuration of the Eclipse OpenJ9™ runtime virtual environment (VM) and details of the GC configuration(GC). The configuration is recorded by using child elements of the `<initialized>` element, for example:

```
<initialized id="1" timestamp="2020-10-18T13:27:07.691">
  <attribute name="gcPolicy" value="-Xgcpolicy:gencon" />
  <attribute name="maxHeapSize" value="0x40000000" />
  <attribute name="initialHeapSize" value="0x40000000" />
  <attribute name="compressedRefs" value="true" />
  <attribute name="compressedRefsDisplacement" value="0x0" />
  <attribute name="compressedRefsShift" value="0x0" />
  <attribute name="pageSize" value="0x1000" />
  <attribute name="pageType" value="not used" />
  <attribute name="requestedPageSize" value="0x1000" />
  <attribute name="requestedPageType" value="not used" />
  <attribute name="gcthreads" value="4" />
  <attribute name="gcthreads Concurrent Mark" value="1" />
  <attribute name="packetListSplit" value="1" />
  <attribute name="cacheListSplit" value="1" />
  <attribute name="splitFreeListSplitAmount" value="1" />
  <attribute name="numaNodes" value="0" />
  <system>
    <attribute name="physicalMemory" value="100335456256" />
    <attribute name="numCPUs" value="28" />
    <attribute name="architecture" value="amd64" />
    <attribute name="os" value="Linux" />
    <attribute name="osVersion" value="3.10.0-1160.el7.x86_64" />
  </system>
  <vmargs>
    <vmarg name="-Dfile.encoding=bestfit936" />
    ...
    <vmarg name="-Xms1024m" />
    <vmarg name="-Xmx1024m" />
    ...
    <vmarg name="-Xverbosegclog:verbosegc.xml" />
    ...
  </vmargs>
</initialized>
```

```
</vmargs>
</initialized>
```

The first set of `<attribute>` elements records the configuration of the garbage collector, such as the GC policy type, configuration of the Java object heap, and the number of threads that are used for garbage collection. For example, the `GCThreads` attribute records that the garbage collector is configured to use four threads.

The `<system>` section records information about the operating system and available hardware, such as the physical memory, number of CPUs, and operating system type and version. In the example, the VM is running on Linux® amd64 V3.10 and has access to 28 CPUs and over 100 GB.

The `<vmargs>` section records any VM configuration [command-line options](#) (VM arguments) that are specified. The following types of options are recorded:

- non-standard [JVM -X options](#) and [JVM -XX options](#). In the example output, the log records the location of the file that contains VM options and definitions as `java/perffarm/sdks/011_j9_x64_linux-20201014/sdk/lib/options.default`. The verbose GC log option is set to `-Xverbosegclog:verbosegc.xml` to write the verbose GC log output to an XML file. The initial and maximum Java object heap sizes are both set to 1024 KB by using the `-Xms` and `-Xmx` options.
- [system property options](#). In the example output, the system property `file.encoding` is set to `bestfit936` to force the GBK converter to follow unicode 2.0 rather than 3.0 standards.

These command-line options can be set by using the command line, or by passing a manifest file, options file, or environment variable to the VM.

After the configurations are recorded in the Initialization section, the verbose GC log begins recording GC activities and details.

GC cycles

The start of a GC cycle is recorded by the `<cycle-start>` XML element. The trigger for the start of a GC cycle is captured in a preceding element to the `<cycle-start>` element. A GC cycle or GC increment is triggered for one of the following reasons:

- an allocation failure occurs. Allocation failures occur when a request for memory fails because the Java object heap does not have enough memory available. The element `<af-start>` logs an allocation failure trigger.
- a memory threshold is reached. Memory threshold values, which set the conditions for triggering certain types of GC cycles or increments, are defined by the policy type and configuration options. For more information about the particular elements or attributes

that are used to record a memory threshold trigger, see specific policies and cycles in [Log examples](#).

The following XML structure is an example of the verbose GC logs that are generated from the Generational Concurrent GC policy (`-Xgcpolicy:gencon`). In this example, the lines are indented to help illustrate the flow and attributes and some child elements are omitted for clarity:

```
<exclusive-start/>

  <af-start/>

    <cycle-start/>

      <gc-start>

        <mem-info>

          <mem/>

        </mem-info>

      </gc-start>

      <allocation-stats/>

        <gc-op/>

      <gc-end>

        <mem-info>

          <mem/>

        </mem-info>

      </gc-end>

    <cycle-end/>

    <allocation-satisfied/>

  <af-end/>

<exclusive-end/>
```

Some elements serve as markers for starting and ending parts of the GC cycle and do not contain child elements, while other elements do contain child elements. In this example, the `<af-start/>`, `<cycle-start/>`, `<cycle-end/>`, `<allocation-satisfied/>`, and `<af-end/>` XML elements are empty and contain only attributes. All other XML elements contain child XML elements, which are omitted from this simplified example. For detailed examples of log output for a specific cycle, see [Log examples](#)).

GC increments and interleaving

Some GC cycle types are run in piecemeal blocks of operations called GC increments. Using GC increments reduces pause times by enabling blocks of operations or operation steps to interleave with operations or operation steps from other types of cycle.

For example, consider the garbage collector for the `gencon` policy, which uses partial cycles and global cycles. The partial cycle consists of just 1 GC operation, scavenge, that runs on the *nursery* area during a stop-the-world (STW) pause. However, the `gencon` global cycle, which runs when the *tenure* area is close to full, is split into three increments. The initial and final global cycle increments run during a relatively short STW pause. The intermediate global cycle increment, which consists of the majority of the GC cycle's work, runs its GC operations concurrently.


Splitting the global cycle operations into these increments reduces pause times by running most of the GC operations concurrently with application threads. The `gencon` global cycle's concurrent increment is paused when a `gencon` partial GC cycle is triggered and resumes when the partial cycle, or multiple partial cycles, complete. In this way, a global cycle can progress while other types of cycle are run by pausing and resuming the concurrent work. In some policies, concurrent operations are split further into multiple concurrent increments for better control of progress of the concurrent operation.

You can see this interleaving of the increments in the verbose GC log. The following table illustrates how the interleaving of the `gencon` policy's partial scavenge and global cycles appears in the logs. Line numbers of an example `gencon` policy's verbose GC log are displayed, alongside columns that show the status of each cycle that is recorded in the logs. (for clarity, not all GC activities are listed):

Table showing how the ``gencon`` policy's global and partial scavenge cycles, which interleave with each other, are recorded in an example log.

Example log line number	<code>`gencon`</code> global GC cycle status recorded in log	<code>`gencon`</code> partial GC cycle status recorded in log
1-87	Initialization section of the logs	
87-51676	-	series of partial scavenge cycles start and finish
51677	global cycle's trigger and target logged	-
51680	STW pause starts	-

Example log line number	`gencon` global GC cycle status recorded in log	`gencon` partial GC cycle status recorded in log
51683	global cycle starts	-
51684	STW pause ends	-
518685	blank line in logs. (Concurrent increment runs)	-
51686	(concurrent increment paused)	STW pause starts
51690	(concurrent increment paused)	partial scavenge cycle starts
51691	(concurrent increment paused)	partial scavenge increment runs
51730	(concurrent increment paused)	partial cycle ends
51733	(concurrent increment resumes)	STW pause ends
51734	blank line in logs. (Concurrent increment resumes)	-
51735	STW pause starts	-
51741	final global increment logged	-
51793	global cycle ends	-
51795	STW pause ends	-

 **Note:** Zero, one, or multiple GC cycles might run between the start and end of a `gencon` global GC cycle.

The XML elements and attribute values that define operations and increments of a particular cycle are specific to the policy and type of cycle. To follow how the different cycle's increments interleave in a log, you can locate the elements and attributes that record the increments and operations that belong to a particular type of cycle. For example, for the `gencon` policy, you can locate the start of the intermediate, concurrent increment of the global cycle by searching for the `<concurrent-kickoff>` element.

For more information about the XML elements and attribute values that are used for a particular type of cycle for a particular policy, and examples of log output, see [Log examples](#).

You can determine the GC increments and operations that are associated with a particular *instance* of a cycle by using the `contextid` and `id` attributes:

1. Determine the ID of the GC cycle: find the value of the `id` attribute of the `<cycle-start>` element that denotes the start of the GC cycle. Note: the `id` attribute increases incrementally with each GC event.
2. Search for the `contextid` attribute values that match the GC cycle's ID. All GC increments, operations, and concurrent events that are associated with a particular cycle have a `contextid` attribute whose value matches the GC cycle's ID.