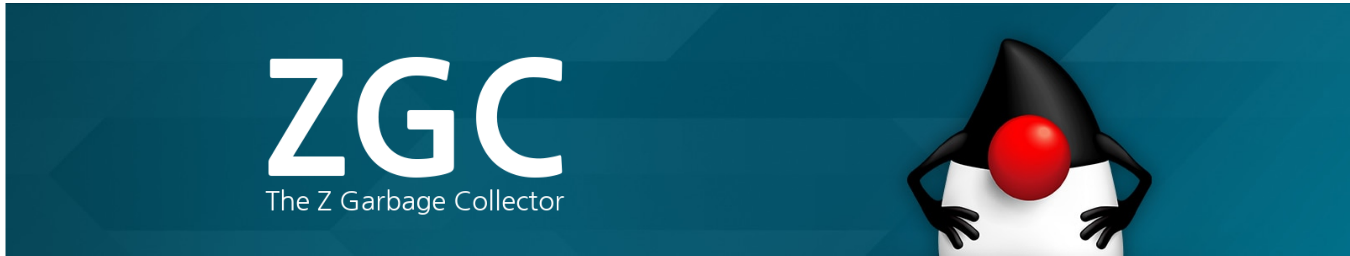
Main



The Z Garbage Collector, also known as **ZGC**, is a scalable low latency garbage collector designed to meet the following goals:

- Pause times **do not** exceed **10ms**
- Pause times **do not** increase with the heap or live-set size
- Handle heaps ranging from a **few hundred megabytes** to **multi terabytes** in size

At a glance, ZGC is:

- Concurrent
- Region-based
- Compacting
- NUMA-aware
- Using colored pointers
- Using load barriers

At its core, ZGC is a **concurrent** garbage collector, meaning all heavy lifting work is done while **Java threads continue to execute**. This greatly limits the impact garbage collection will have on your application's response time.

This OpenJDK project is sponsored by the HotSpot Group.

Download

Latest Stable: JDK 13
Latest Development: JDK 14 Early Access

Source Code

http://hg.openjdk.java.net/jdk/jdk

Talks

Oracle Code One 2019 - Slides
PL-Meetup 2019 - Slides
Jfokus 2019 - Slides | Video (21 min)
Devoxx 2018 - Slides | Video (40 min)
Oracle Code One 2018 - Slides | Video (45 min)
Jfokus 2018 - Slides | Video (45 min)
FOSDEM 2018 - Slides

Mailing List

Subscribe | Archive

Project

Members
JIRA Dashboard
JEPs 333, 351, 364, 365

Contents

- Change Log
- Supported Platforms
- Quick Start
- JVM Options
    - Overview
    - Enabling ZGC
    - Setting Heap Size
    - Setting Concurrent GC Threads
    - Returning Unused Memory to the Operating System
    - Enabling Large Pages On Linux
    - Enabling Transparent Huge Pages On Linux
    - Enabling NUMA Support
    - Enabling GC Logging

Change Log

JDK 15

- Concurrent thread stack scanning (in progress)
- Improved NUMA awareness on Linux
- <TBD>

JDK 14

- macOS support (JEP 364)
- Windows support (JEP 365)
- Support for tiny/small heaps (down to 8M)
- Support for JFR leak profiler
- Support for limited and discontiguous address space
- Parallel pre-touch (when using -XX:+AlwaysPreTouch)
- Performance improvements (clone intrinsic, etc)
- Stability improvements

JDK 13

- Increased max heap size from 4TB to 16TB
- Support for uncommitting unused memory (JEP 351)
- Support for -XX:SoftMaxHeapSIze
- Support for the Linux/AArch64 platform
- Reduced Time-To-Safepoint

JDK 12

- Support for concurrent class unloading
- Further pause time reductions

JDK 11

- Initial version of ZGC
- Does not support class unloading (using -XX:+ClassUnloading has no effect)

Supported Platforms

Platform	Supported	Since	Comment
Linux/x64	✓	JDK 11	
Linux /AArch64	✓	JDK 13	
macOS	✓	JDK 14	
Windows	✓	JDK 14	Requires Windows version 1803 (Windows 10 or Windows Server 2019) or later.

Quick Start

If you're trying out ZGC for the first time, start by using the following GC options:

```
-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx<size> -Xlog:gc
```

For more detailed logging, use the following options:

```
-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx<size> -Xlog:gc*
```

See below for more information on these and additional options.

JVM Options

Overview

The following JVM options can be used with ZGC:

General GC Options	ZGC Options	ZGC Diagnostic Options (-XX: +UnlockDiagnosticVMOptions)

-XX:MinHeapSize, -Xms -XX:InitialHeapSize, -Xms -XX:MaxHeapSize, -Xmx -XX:SoftMaxHeapSize -XX:ConcGCThreads -XX:ParallelGCThreads -XX:SoftRefLRUPolicyMSPerMB	-XX:ZAllocationSpikeTolerance -XX:ZCollectionInterval -XX:ZFragmentationLimit -XX:ZMarkStackSpaceLimit -XX:ZPath -XX:ZUncommit -XX:ZUncommitDelay	-XX:ZProactive -XX:ZStatisticsInterval -XX:ZVerifyForwarding -XX:ZVerifyMarking -XX:ZVerifyObjects -XX:ZVerifyRoots -XX:ZVerifyViews

Enabling ZGC

Use the `-XX:+UnlockExperimentalVMOptions -XX:+UseZGC` options to enable ZGC.

Setting Heap Size

The most important tuning option for ZGC is setting the max heap size (`-Xmx<size>`). Since ZGC is a concurrent collector a max heap size must be selected such that, 1) the heap can accommodate the live-set of your application, and 2) there is enough headroom in the heap to allow allocations to be serviced while the GC is running. How much headroom is needed very much depends on the allocation rate and the live-set size of the application. In general, the more memory you give to ZGC the better. But at the same time, wasting memory is undesirable, so it's all about finding a balance between memory usage and how often the GC needs to run.

Setting Concurrent GC Threads

The second tuning option one might want to look at is setting the number of concurrent GC threads (`-XX:ConcGCThreads=<number>`). ZGC has heuristics to automatically select this number. This heuristic usually works well but depending on the characteristics of the application this might need to be adjusted. This option essentially dictates how much CPU-time the GC should be given. Give it too much and the GC will steal too much CPU-time from the application. Give it too little, and the application might allocate garbage faster than the GC can collect it.

NOTE! In general, if low latency (i.e. low application response time) is important for you application, then *never* over-provision your system. Ideally, your system should never have more than 70% CPU utilization.

Returning Unused Memory to the Operating System

By default, ZGC uncommits unused memory, returning it to the operating system. This is useful for applications and environments where memory footprint is a concern. This feature can be disabled using `-XX:-ZUncommit`. Furthermore, memory will not be uncommitted so that the heap size shrinks below the minimum heap size (`-Xms`). This means this feature will be implicitly disabled if the minimum heap size (`-Xms`) is configured to be equal to the maximum heap size (`-Xmx`).

An uncommit delay can be configured using `-XX:ZUncommitDelay=<seconds>` (default is 300 seconds). This delay specifies for how long memory should have been unused before it's eligible for uncommit.

NOTE! On Linux, uncommitting unused memory requires `fallocate(2)` with `FALLOC_FL_PUNCH_HOLE` support, which first appeared in kernel version **3.5** (for tmpfs) and **4.3** (for hugetlbfs).

Enabling Large Pages On Linux

Configuring ZGC to use large pages will generally yield better performance (in terms of throughput, latency and start up time) and comes with no real disadvantage, except that it's slightly more complicated to setup. The setup process typically requires root privileges, which is why it's not enabled by default.

Large pages are also known as "huge pages" on Linux/x86 and have a size of 2MB.

Let's assume you want a 16G Java heap. That means you need 16G / 2M = 8192 huge pages.

First assign at least 16G (8192 pages) of memory to the pool of huge pages. The "at least" part is important, since enabling the use of large pages in the JVM means that not only the GC will try to use these for the Java heap, but also that other parts of the JVM will try to use them for various internal data structures (code heap, marking bitmaps, etc). In this example we will therefore reserve 9216 pages (18G) to allow for 2G of non-Java heap allocations to use large pages.

Configure the system's huge page pool to have the required number pages (requires root privileges):

```
$ echo 9216 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

Note that the above command is not guaranteed to be successful if the kernel can not find enough free huge pages to satisfy the request. Also note that it might take some time for the kernel to process the request. Before proceeding, check the number of huge pages assigned to the pool to make sure the request was successful and has completed.

```
$ cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
9216
```

NOTE! If you're using a **Linux kernel >= 4.14**, then the next step (where you mount a hugetlbfs filesystem) can be skipped. However, if you're using an older kernel then ZGC needs to access large pages through a hugetlbfs filesystem.

Mount a hugetlbfs filesystem (requires root privileges) and make it accessible to the user running the JVM (in this example we're assuming this user has 123 as its uid).

```
$ mkdir /hugepages
$ mount -t hugetlbfs -o uid=123 nodev /hugepages
```

Now start the JVM using the `-XX:+UseLargePages` option.

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xms16G -Xmx16G -XX:
+UseLargePages ...
```

If there are more than one accessible hugetlbfs filesystem available, then (and only then) do you also have to use `-XX:ZPath` to specify the path to the filesystems you want to use. For example, assume there are multiple accessible hugetlbfs filesystems mounted, but the filesystem you specifically want to use it mounted on `/hugepages`, then use the following options.

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xms16G -Xmx16G -XX:
+UseLargePages -XX:ZPath=/hugepages ...
```

NOTE! The configuration of the huge page pool and the mounting of the hugetlbfs file system is not persistent across reboots, unless adequate measures are taken.

Enabling Transparent Huge Pages On Linux

An alternative to using explicit large pages (as described above) is to use transparent huge pages. Use of transparent huge pages is usually **not** recommended for latency sensitive applications, because it tends to cause unwanted latency spikes. However, it might be worth experimenting with to see if/how your workload is affected by it. But be aware, your mileage may vary.

Note that using ZGC with transparent huge pages enabled requires **Linux kernel >= 4.7**.

Use the following options to enable transparent huge pages in the VM:

```
-XX:+UseLargePages -XX:+UseTransparentHugePages
```

These options tell the JVM to issue `madvise(..., MADV_HUGEPAGE)` calls for memory it mapps, which is useful when using transparent huge pages in *madvise* mode.

To enable transparent huge pages you also need to configure the kernel, by enabling the *madvise* mode.

```
$ echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
```

and

```
$ echo advise > /sys/kernel/mm/transparent_hugepage/shmem_enabled
```

See the kernel documentation for more information.

Enabling NUMA Support

ZGC has basic NUMA support, which means it will try it's best to direct Java heap allocations to NUMA-local memory. This feature is **enabled by default**. However, it will automatically be disabled if the JVM detects that it's bound to a sub-set of the CPUs in the system. In general, you don't need to worry about this setting, but if you want to explicitly override the JVM's decision you can do so by using the `-XX:+UseNUMA` or `-XX:-UseNUMA` options.

When running on a NUMA machine (e.g. a multi-socket x86 machine), having NUMA support enabled will often give a noticeable performance boost.

Enabling GC Logging

GC logging is enabled using the following command-line option:

```
-Xlog:<tag set>,[<tag set>, ...]:<log file>
```

For general information/help on this option:

```
-Xlog:help
```

To enable basic logging (one line of output per GC):

```
-Xlog:gc:gc.log
```

To enable GC logging that is useful for tuning/performance analysis:

```
-Xlog:gc*:gc.log
```

Where `gc*` means log all tag combinations that contain the gc tag, and `:gc.log` means write the log to a file named `gc.log`.