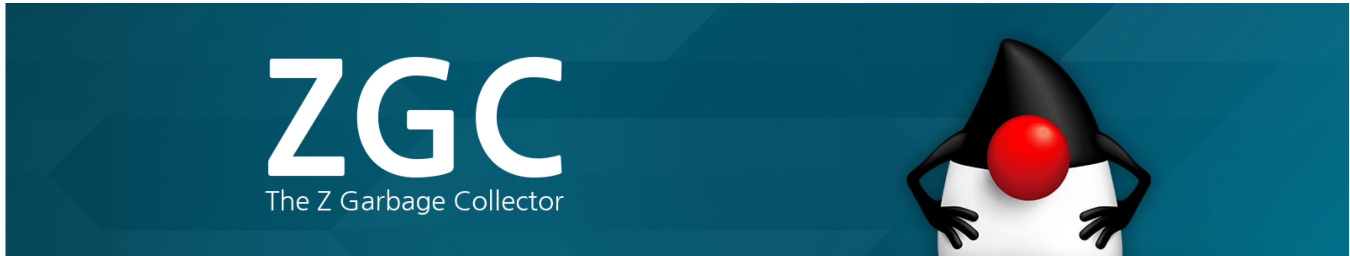
Main



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

Stable: JDK 12 (Linux/x64)
Development: JDK 13 Early Access (Linux/x64)

Source Code

Stable: http://hg.openjdk.java.net/jdk/jdk
Development: http://hg.openjdk.java.net/zgc/zgc

Talks

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

JIRA Dashboard
JEP 351
JEP 333
Members

Contents

- Change Log
- Supported Platforms
- Download Binary
- Download and Build from Source
- Quick Start
- JVM Options
  - Enabling ZGC
  - Setting Heap Size
  - Setting Concurrent GC Threads
  - Enabling Large Pages
  - Enabling Transparent Huge Pages
  - Enabling NUMA Support
  - Enabling GC Logging

Change Log

JDK 13 (Planned for September 2019)

- 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 (Released March 2019)

- Support for concurrent class unloading
- Further pause time reductions

JDK 11 (Released September 2018)

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

Supported Platforms

ZGC is currently available on **Linux/x64** and **Linux/AArch64** (starting with JDK 13). Support for other platforms will be added in the future.

Download Binary

Download JDK 11, JDK 12, or JDK 13 Early Access or **Linux/x64**.

Download and Build from Source

The source code can be found in the latest stable (or development) repository. Just clone, configure and make.

```
$ hg clone http://hg.openjdk.java.net/jdk/jdk
$ cd jdk
$ sh configure
$ make images
```

If you are building early versions of JDK 11 (versions 11.0.0, 11.0.1 or 11.0.2), you need to supply the configure option `--with-jvm-features=zgc` to enable building of ZGC. This option is no longer needed (ZGC is built by default) as of JDK 11.0.3 and JDK 12.

This will build a complete JDK for you. On Linux, the root directory of the new JDK will be found here:

```
./build/linux-x86_64-normal-server-release/images/jdk
```

And the Java launcher will be found in its usual place, here:

```
./build/linux-x86_64-normal-server-release/images/jdk/bin/java
```

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

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.

Enabling Large Pages

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

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`.