# **G1 GC**

The first thing that catches your eye when considering G1 is the change of approach to **heap organization**.

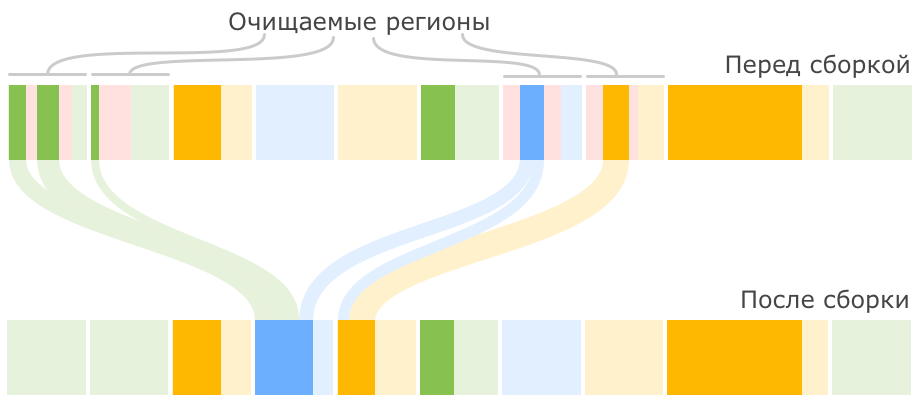
Memory is divided into many regions of the same size. The size of these regions depends on the total size of the heap and is chosen by default to be no larger than 2048, usually ranging from 1 to 32 MB. The only exception is the so-called humongous regions, which are created by combining regular regions to accommodate very large objects.

A grid of squares with different colored squares

Description automatically generated

Small builds are performed periodically to clean up the younger generation and move objects to Survivor regions, or to upgrade them to the older generation and move them to Tenured.

Cleaning is performed not on the entire generation, but only on the part of regions that the builder can clean without exceeding the desired time. In this case, it chooses for cleaning those regions in which, in its opinion, the largest amount of garbage has accumulated and cleaning of which will bring the greatest result. **Garbage First**.



Any object larger than half a region is considered huge and is treated in a special way:

* It is never moved between regions.
* It may be removed as part of a marking cycle or a full garbage collection.
* A region occupied by a bulky object is never occupied by anyone else, even if there is space left in the region.

**Configuring**

-XX:MaxGCPauseMillis=?

-XX:G1HeapRegionSize=?

**Pros and Cons**

G1 builder is more accurate at predicting pause sizes than CMS and better at distributing builds over time to prevent long application stalls, especially for large heap sizes. At the same time, it also lacks some of the other drawbacks of CMS, for example, it does not fragment memory.

The price for G1's advantages is CPU resources, which it uses to perform quite a large part of its work in parallel with the main program.

# **ZGC**

From Java 15 is production-ready garbage collector.

Purpose of ZGC:

* Keep Stop-The-World pauses to less than one millisecond.
* Ensure that pauses do not increase with heap size, number of live objects, or number of root references.
* Support heaps up to 16 TB in size.

Compared to the Garbage-First collector, ZGC is also a region-based collector; however, it is not generational. It improves upon G1 by achieving concurrent compaction with the introduction of two core techniques, read barriers and coloured pointers.

A reference represents the position of a byte in the virtual memory. However, we don’t necessarily have to use all bits of a reference to do that – **some bits can represent properties of the reference**. That’s what we call reference coloring.

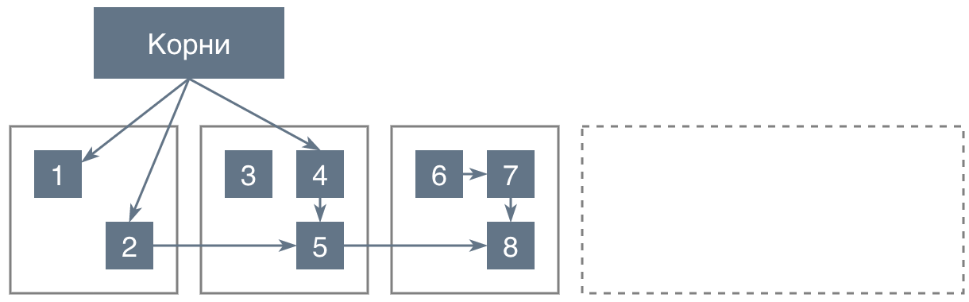
A diagram of a computer code

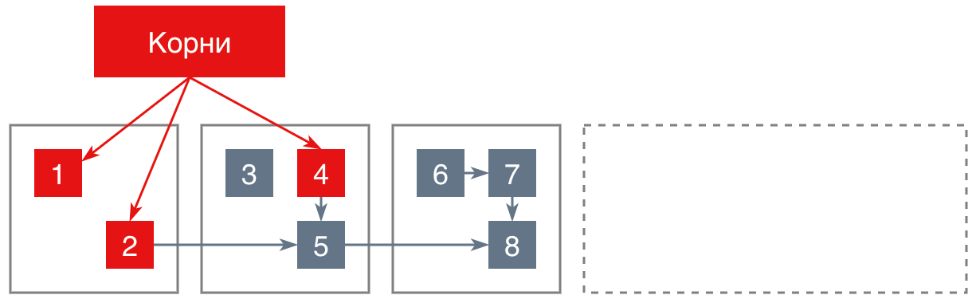
Description automatically generated with medium confidence

**Searching for living objects**

The first stage of the collector's work is to paint the signs for the reachable objects.

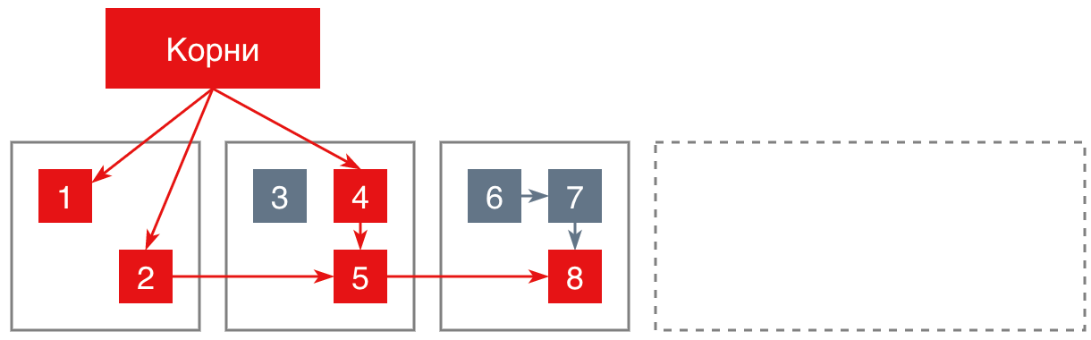
**Pause Mark Start Phase**





After all the objects reachable from the roots are found, the application operation is resumed and the **second phase begins**, in which the **collector continues to search for live objects reachable from the found roots**, but already in a concurrent mode.

After the end of the **concurrent phase**, there is again an STW pause in which the ZGC handles various special cases. In particular, soft- and weak-references.



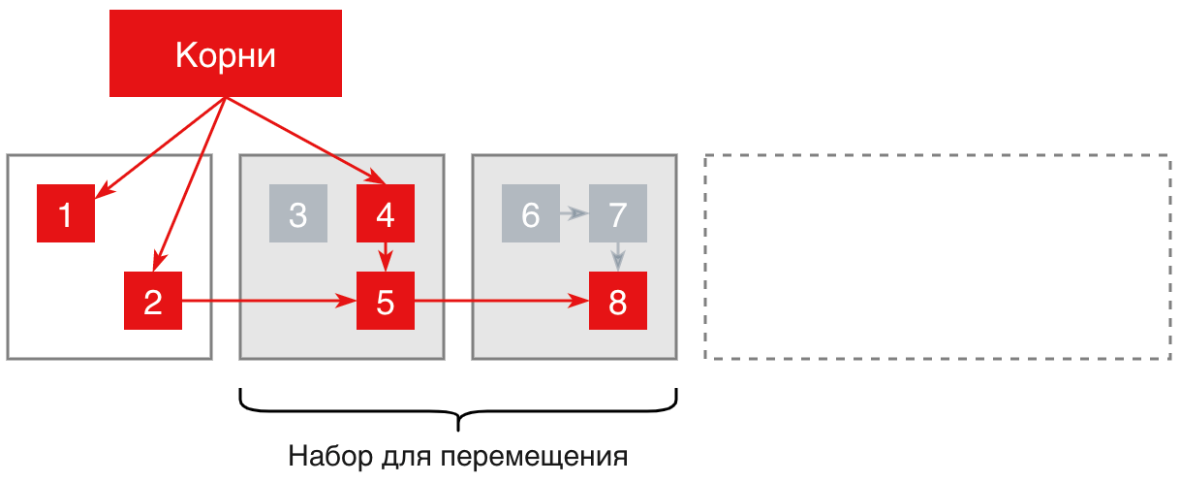
**Relocation**

The next phase of ZGC is moving objects to defragment and free up memory. This phase is also divided into several phases:

**Concurrent Prepare for Relocate**

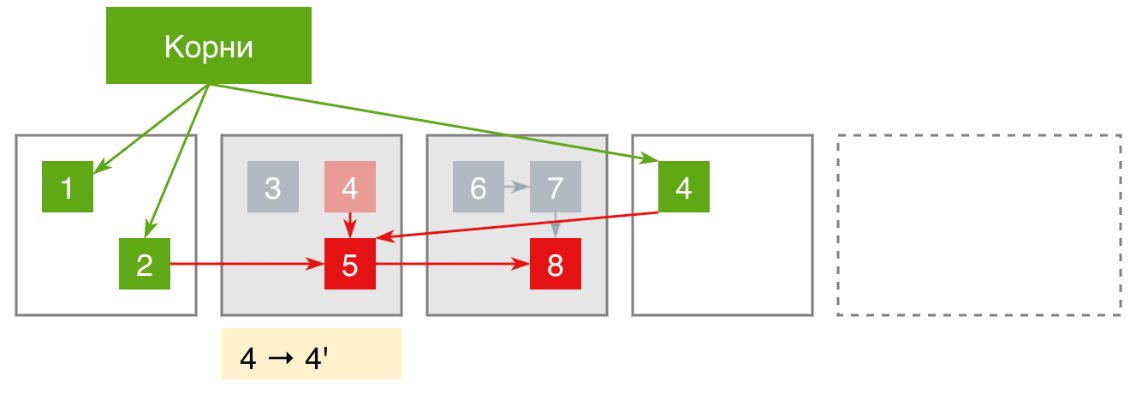
As part of this activity, the collector defines memory blocks from which objects are to be relocated. These blocks are included in the so-called relocation set.

The relocation set includes blocks that contain enough dead objects to be useful for relocation:



**Pause Relocate Start**

If an object is reachable by a pointer from the root and is to be moved, it is moved to a new memory block, the root pointer to it is painted and the assembler remembers the correspondence between the old and new address of the moved object in special **forwarding tables**. Such tables are maintained separately for each block in memory outside the heap.



**Concurrent Remap**

After the end of the previous phase, all objects are moved to the target memory regions and the only thing that separates us from the final target picture are the hovering pointers to objects in the freed memory regions (they remain red on the diagram).

To correct all such pointers, we need to make another traversal of the object graph, traversing all pointers and redirecting them to new addresses according to the redirection tables. But ZGC does not perform this phase at once within the current build cycle, but combines it with the Concurrent Mark phase of the next build cycle.

**Pros and Cons**

In practice, ZGC does indeed achieve sub-millisecond pauses, including on very large heaps, which is very good for applications sensitive to even short delays in query processing.

Also to keep pauses at the sub-millisecond level and prevent full ZGC garbage collections requires a sufficient number of free memory regions. If your application is very actively creating new objects, it will likely need a slightly larger heap than the same G1.

**Shenandoah GC**

The Shenandoah garbage collector, like the ZGC, has the goal of reducing pause times on large heaps.

It is also a region-based non generational collector that uses similar principles to ZGC but follows a different implementation strategy. I

nstead of coloured pointers, Shenandoah makes use of Brooks pointers, for allowing concurrent compaction of memory. The main idea behind a Brooks pointer is that each object has an additional reference field that always points to the current location of the object.

**A diagram of a diagram

Description automatically generated**

A classic ”stop-the-world” compaction would then stop application threads so it could update all references to the old ”from-space” object, to the current ”tospace” reference.

However, with Brooks pointers, we no longer need to stop the application to update all references leading to the ”from-space” object. Every object now has an additional reference field (forward pointer) that points to the object itself, or, as soon as the object gets copied to a new location, to that new location.