READ MODE · ANSWER FIRST

ZGC: How It Works

An interactive, visual explanation of ZGC: How It Works, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

The whole story in 6 lines

ZGC: How It Works becomes easier to reason about when every stage is connected as one system.

1. Page-based layout enables per-page evacuation and concurrent compaction without whole-heap STW.
2. Colored pointers let the GC communicate state through every reference without touching the object header.
3. Multi-mapping lets any colored pointer resolve to the correct physical page regardless of which metadata bit is set.
4. The load barrier is the only synchronization point between mutator threads and the GC. It is the beating heart of ZGC.
5. Marking flips pointer colors from Remapped to Marked. The load barrier ensures mutator threads cooperate.
6. Relocation selects sparse pages, moves live objects to new pages, and lets the barrier fix references lazily.

STAGE 1 OF 8

Setup

Welcome. ZGC is the low-latency garbage collector shipped with modern JVMs. It can handle multi-terabyte heaps while keeping pause times under one millisecond. This explainer walks through exactly how it achieves that.

A ZPage is a contiguous chunk of memory that holds Java objects. ZGC divides the entire heap into these pages instead of using one big slab. Think of them like numbered containers in a warehouse.

A colored pointer is a 64-bit object reference that embeds metadata directly in the pointer bits. Instead of storing mark state in the object header, ZGC stores it in the reference itself. This is the key innovation.

A load barrier is a small piece of code the JVM injects every time your application loads an object reference. If the reference has a stale color, the barrier fixes it on the spot. This is how ZGC works concurrently without stopping your threads.

A forwarding table records where an object has been relocated. When the collector moves an object from one page to another, it writes the old-to-new address mapping into this table. The load barrier reads from it to self-heal stale pointers.

Over the next six stages we will trace the full ZGC cycle: heap pages, colored pointers, multi-mapped memory, the load barrier, concurrent marking, and concurrent relocation. Let us start with how memory is organized.

STAGE 2 OF 8

Heap Pages

02 HEAP PAGES

The JVM starts with a heap divided into ZPages. There are three sizes: small pages are 2 MB, medium pages are 32 MB, and large pages are multiples of 2 MB reserved for huge objects. This per-page organization is what makes concurrent relocation possible.

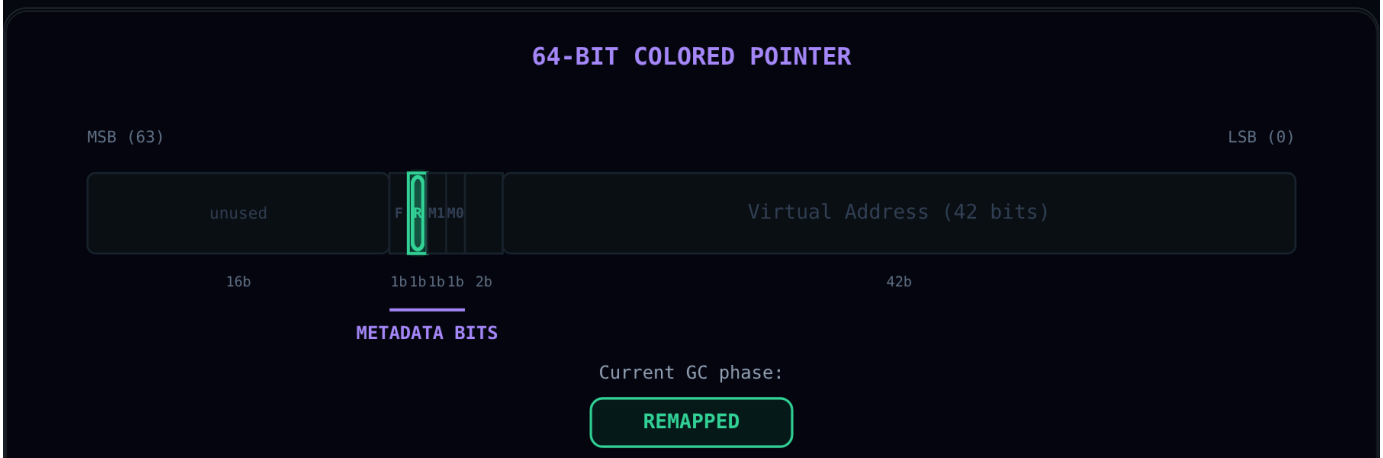
Small pages hold objects up to 256 KB. They are the most common page type and the majority of your allocations land here. Watch the objects fill the page from left to right.

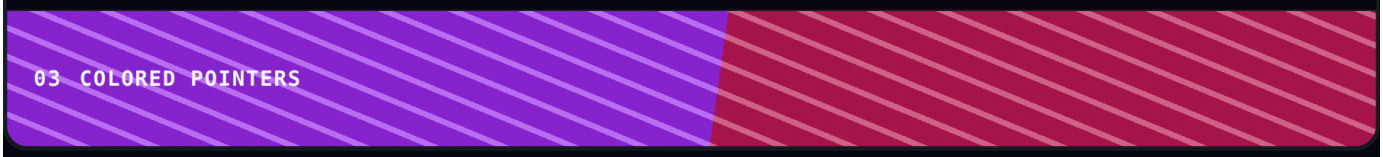
When an object exceeds 256 KB but stays under 4 MB, the allocator places it in a medium page. Medium pages are 32 MB and give larger objects room to breathe without wasting a whole large page.

Objects larger than 4 MB get their own large page. A large page is sized exactly to fit one object, always a multiple of 2 MB. Only one object ever lives in a large page. Switch the Page type control in the sidebar to see the three different page layouts.

When ZGC needs to compact the heap, it evacuates entire pages by relocating all live objects out, then reclaims the page. This page-at-a-time approach means the collector never needs to pause to compact the whole heap at once. That is the first piece of the sub-millisecond puzzle.

Colored Pointers





Now that we know how pages organize memory, let us look at how ZGC tracks object state. Every object reference in ZGC is a 64-bit colored pointer. The color is metadata stored directly in the pointer bits. No separate mark bitmap, no object header flags.

The lowest 42 bits hold the virtual address of the object. That gives ZGC a 4 TB address space. The bits above that are the metadata bits: Finalizable, Remapped, Marked1, and Marked0.

At any moment, exactly one metadata bit is the "good" color for the current GC phase. During normal execution the Remapped bit is set. When a mark cycle starts, the good color flips to Marked0 or Marked1. The load barrier checks this bit on every reference load.

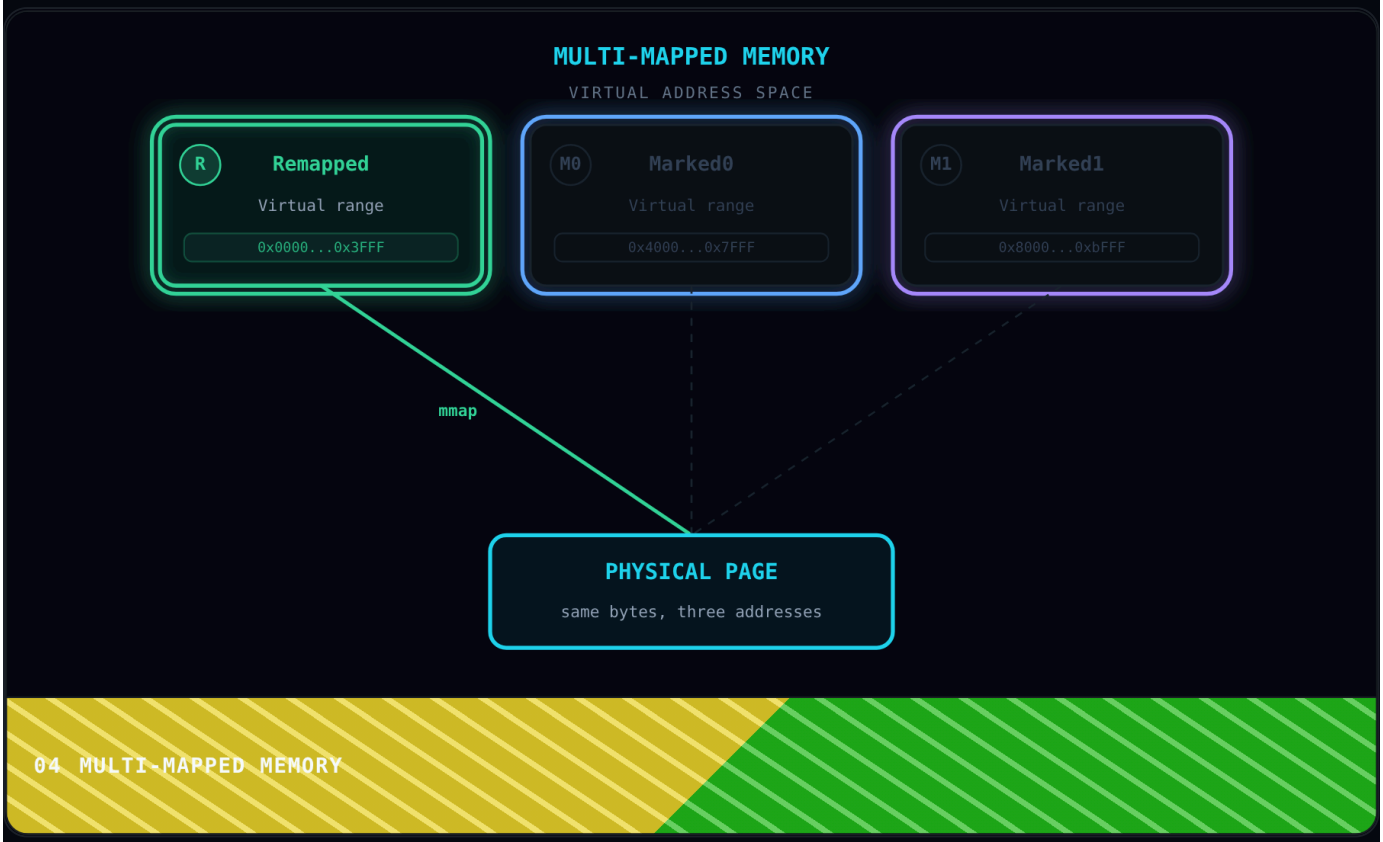
Switch the GC phase control in the sidebar. Watch which bit becomes the active color. In the Remapped phase the R bit is lit. In Marking the M0 bit is lit. In Relocating the R bit lights up again because freshly relocated references get the Remapped color.

Why alternate between Marked0 and Marked1? Because mark cycles alternate: one cycle uses M0 as the good color, the next uses M1. This avoids a separate "clear all mark bits" phase between cycles. The old marks just become stale and ignored.

The Finalizable bit is special. It marks references discovered through finalizer or phantom-reference chains. These objects are technically unreachable from normal roots but still alive until their finalizer runs. That completes the pointer anatomy. Next, let us see how ZGC makes these colored pointers work with hardware.

STAGE 4 OF 8

Multi-Mapped Memory



04 MULTI-MAPPED MEMORY

We just learned that pointers carry a color bit. But a CPU does not know about GC colors. It just dereferences the full 64-bit address. If different metadata bits produce different addresses, how does the hardware find the right memory? The answer is multi-mapping.

ZGC asks the OS to map the same physical page at three different virtual address ranges. One range corresponds to the Remapped bit, one to Marked0, and one to Marked1. All three virtual addresses land on the same physical bytes.

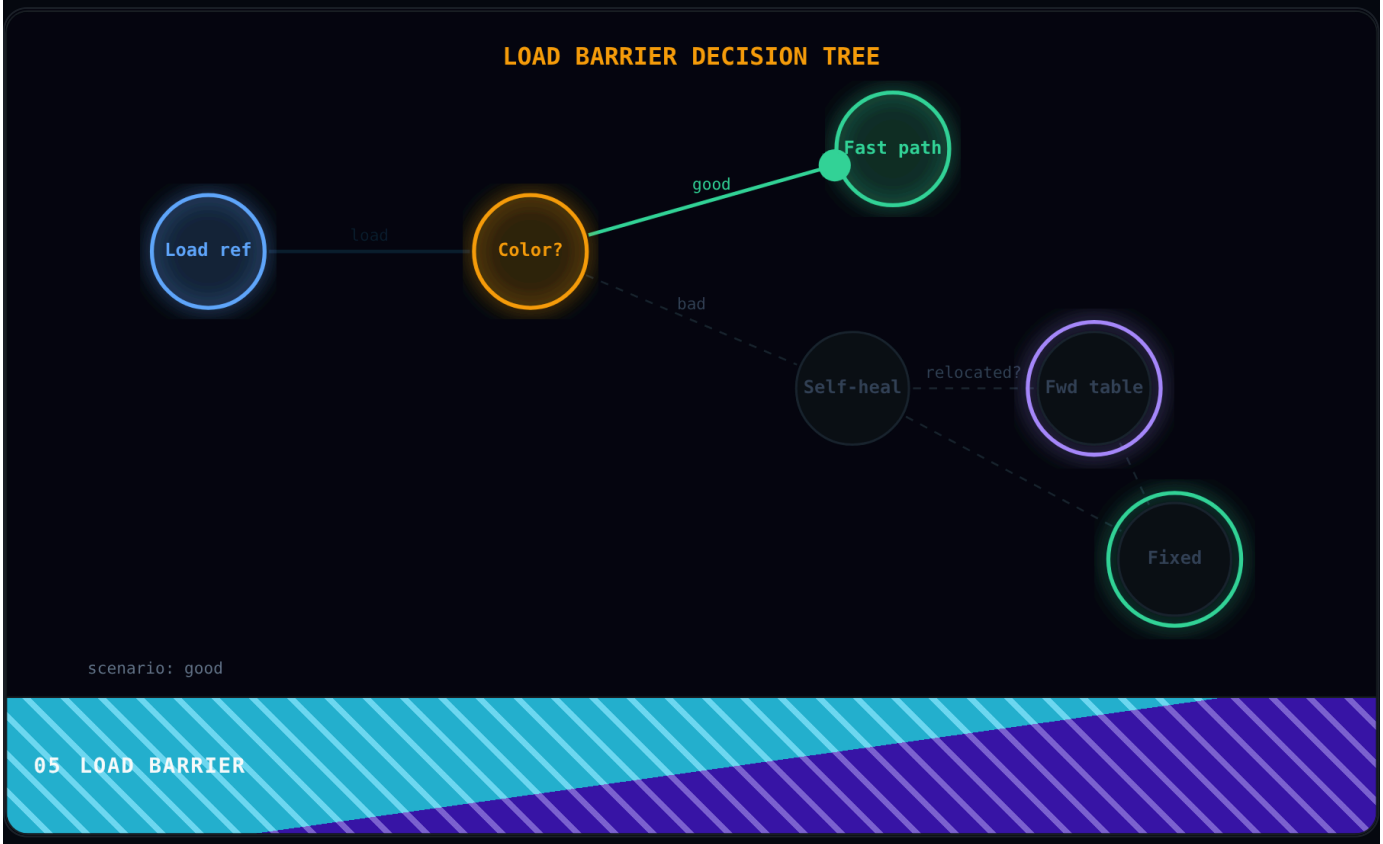
When a pointer has the Remapped bit set, its virtual address falls in the Remapped range. When the same logical object is referenced with Marked0 set, the address falls in the Marked0 range. Both resolve to the same physical location.

Try the Active color control. Switch between Remapped, Marked0, and Marked1. Watch how the highlighted virtual range changes but the arrow always points to the same physical page. The CPU follows whichever colored address it is given and always reaches the right data.

This multi-mapping trick is what makes colored pointers practical. Without it, every pointer dereference would need the metadata bits stripped before the load. With it, the hardware does the right thing automatically. The load barrier only needs to fix the color, not the address. Next we will see how that barrier actually works.

STAGE 5 OF 8

Load Barrier



05 LOAD BARRIER

We now know that pointers carry color and that multi-mapping makes them dereferenceable. But what happens when a pointer has the wrong color for the current phase? That is where the load barrier comes in. It is a tiny piece of code the JIT compiler injects after every reference load.

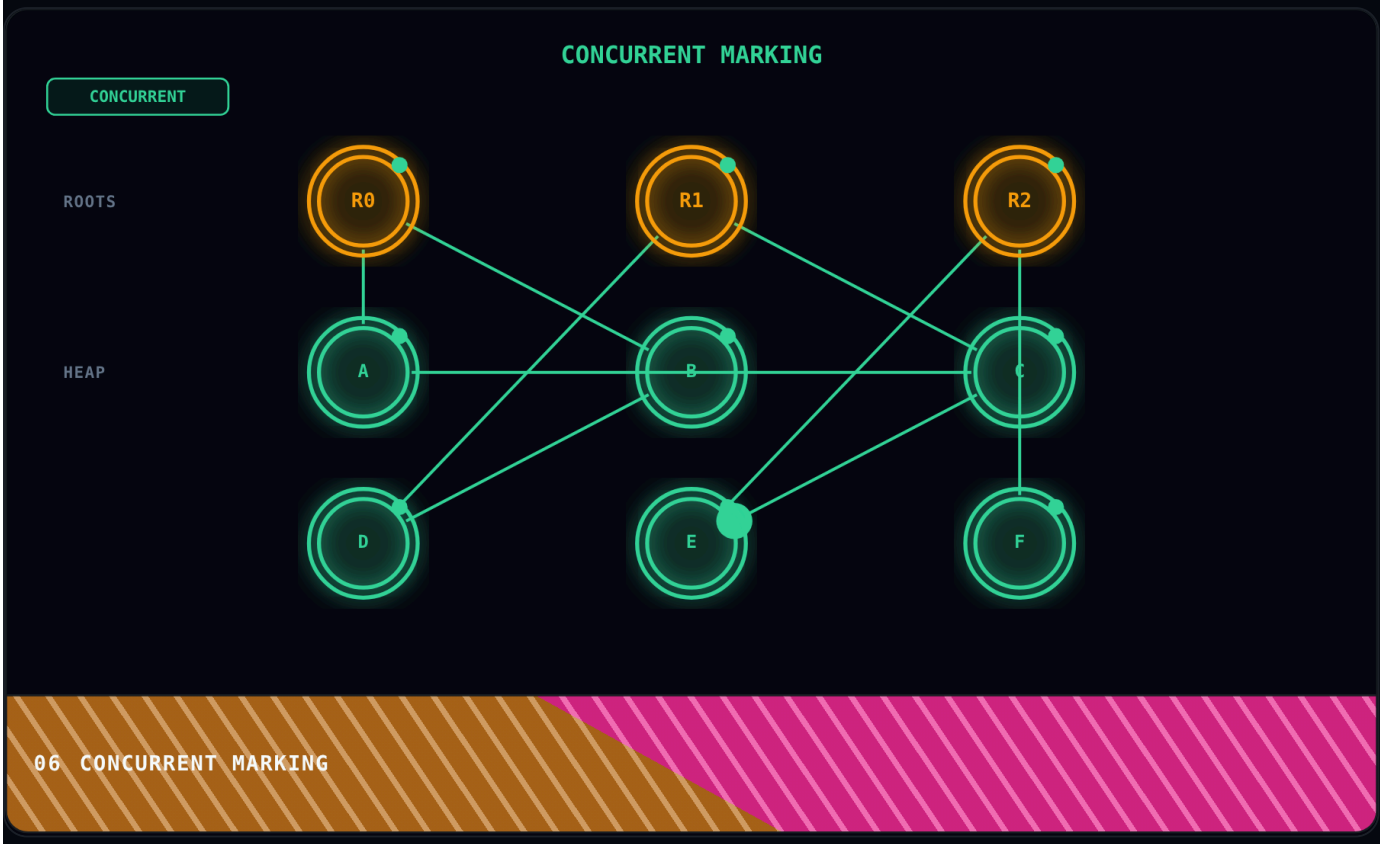
The fast path is simple. The barrier checks the metadata bits. If the pointer already has the good color for the current phase, it does nothing. This is the common case and it costs essentially one test-and-branch. Watch the reference sail through with no correction.

Now switch the Scenario control to Stale ref. The pointer has an outdated color. The barrier catches this and updates the metadata bits in place. This is called self-healing: the barrier rewrites the pointer in the field so future loads of the same field skip the slow path.

Switch to Relocated. This is the heaviest case. The object was moved during relocation. The barrier looks up the forwarding table for the old page, finds the new address, and updates the reference with the new address and the good color. The forwarding table from the Setup stage is exactly what the barrier reads here.

Because the barrier fixes the pointer in the referring field, each stale reference is healed at most once. Subsequent loads find the good color and take the fast path. This amortized self-healing is why ZGC can relocate objects concurrently without stopping the world. The barrier is the single mechanism that makes all of ZGC concurrent.

Concurrent Marking



We have all the building blocks: pages, colored pointers, multi-mapping, and the load barrier. Now let us see them in action during a mark cycle. Marking answers one question: which objects are still alive?

The cycle begins with a brief stop-the-world pause called Initial Mark. This pause scans GC roots: thread stacks, static fields, JNI handles. It is extremely short because it only processes the roots, not the full graph. Watch the root nodes light up.

After the STW pause ends, application threads resume. GC worker threads begin concurrent marking. They follow references from each root, visiting every reachable object and flipping its pointer color from Remapped to the current mark color. Try changing the Root count to see more or fewer roots being traced.

While GC workers trace the graph, application threads keep running and can modify references. The load barrier cooperates: any reference loaded by the application during marking gets its color checked and, if needed, marked on the spot. This is how the marking wave and the application coexist.

A second brief STW pause called Remark handles a small set of references that changed during concurrent marking. It also processes weak references, cleaners, and the finalization queue. After remark, the mark phase is done and ZGC knows exactly which objects are alive.

The result is a complete liveness map of the heap built almost entirely concurrently. The only pauses were the two tiny bookend STW phases. Next, ZGC uses this liveness information to relocate objects and compact memory.

STAGE 7 OF 8

Concurrent Relocation

07 CONCURRENT RELOCATION

Marking told ZGC which objects are alive. Now it uses that information to compact the heap. The first step is selecting the relocation set: the pages with the lowest liveness ratios. Sparse pages waste the most memory, so they are relocated first.

For each page in the relocation set, ZGC allocates a forwarding table. This table will map every old object address to its new location. Watch the forwarding table appear next to the source page.

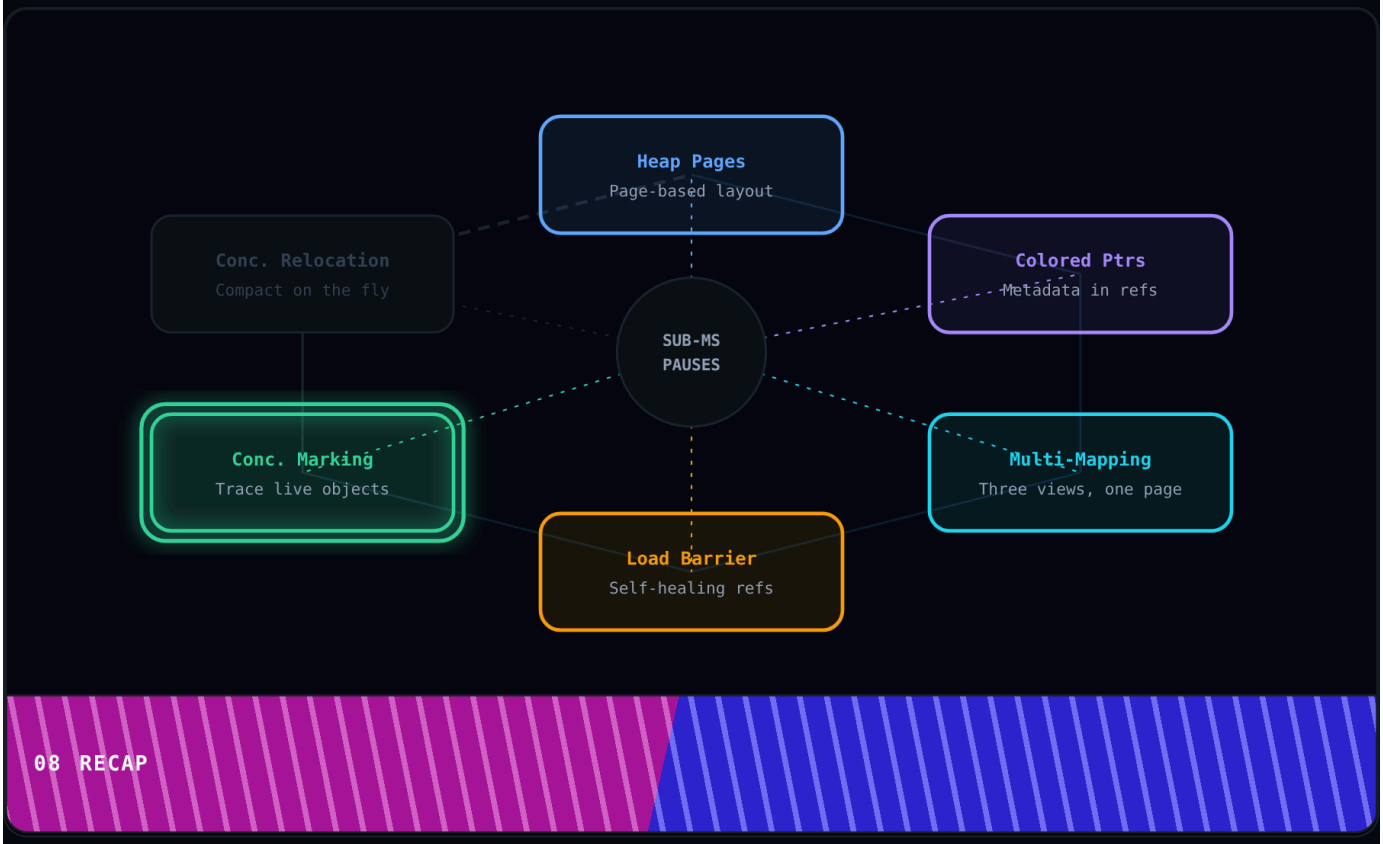
GC worker threads begin moving live objects from the source page to a fresh target page. Each moved object gets an entry in the forwarding table. The application keeps running during this entire process. Switch to Dense in the sidebar to see what happens when a page has high liveness: fewer objects are dead, so less space is reclaimed.

When an application thread loads a reference to an object that was relocated, the load barrier kicks in. It detects the stale color, looks up the forwarding table, finds the new address, and updates the pointer in place. This is the self-healing we saw in Stage 4, now applied at scale across the entire relocation set.

After all live objects are evacuated, the source page is reclaimed. Its forwarding table stays alive until the next mark cycle remaps all remaining stale references. Then the forwarding table is freed too.

That completes the ZGC cycle. Pages are selected, objects are moved concurrently, stale pointers self-heal through the barrier, and empty pages are reclaimed. The whole process runs alongside your application with only two sub-millisecond pauses for root scanning. Let us step back and see the full picture.

Recap



08 RECAP

We started with Heap Pages. ZGC divides the heap into small, medium, and large pages. This per-page organization means the collector can evacuate one page at a time instead of stopping the world to compact everything.

Then we learned about Colored Pointers. Every object reference carries metadata bits that encode the GC phase. This removes the need for mark bitmaps or object header flags and lets the collector communicate state through every reference in the heap.

Multi-Mapped Memory makes colored pointers hardware-friendly. The same physical page is mapped at three virtual addresses, one per color. The CPU follows any colored address and always reaches the right data.

The Load Barrier is the synchronization engine. Every reference load checks the pointer color. If it is stale, the barrier fixes it in place. Self-healing means each stale pointer is corrected at most once, keeping the overhead small.

Concurrent Marking traces the live object graph almost entirely in the background. Two tiny STW pauses bookend the traversal. The load barrier ensures application threads cooperate without being stopped.

Concurrent Relocation selects sparse pages, moves live objects to fresh pages, and records the new addresses in forwarding tables. The load barrier heals stale pointers on the fly. Empty pages are reclaimed without any pause.

Together these six mechanisms form a single cohesive design. Colored pointers carry state, multi-mapping makes them dereferenceable, the load barrier keeps everything consistent, and the page-based layout enables incremental compaction. That is how ZGC delivers sub-millisecond pause times on heaps up to 16 terabytes.