

Your 100 GB JVM Needs More Than 100 GB

Oct 25, 2026 · @Chandrakanth Anne

A JVM started with `-Xmx100g` can still be killed for using too much memory. `-Xmx` caps only the Java heap, but the kernel counts the whole process. Here is what sits outside the heap, how big it gets, and how to budget for it.

Two ways to run out of memory

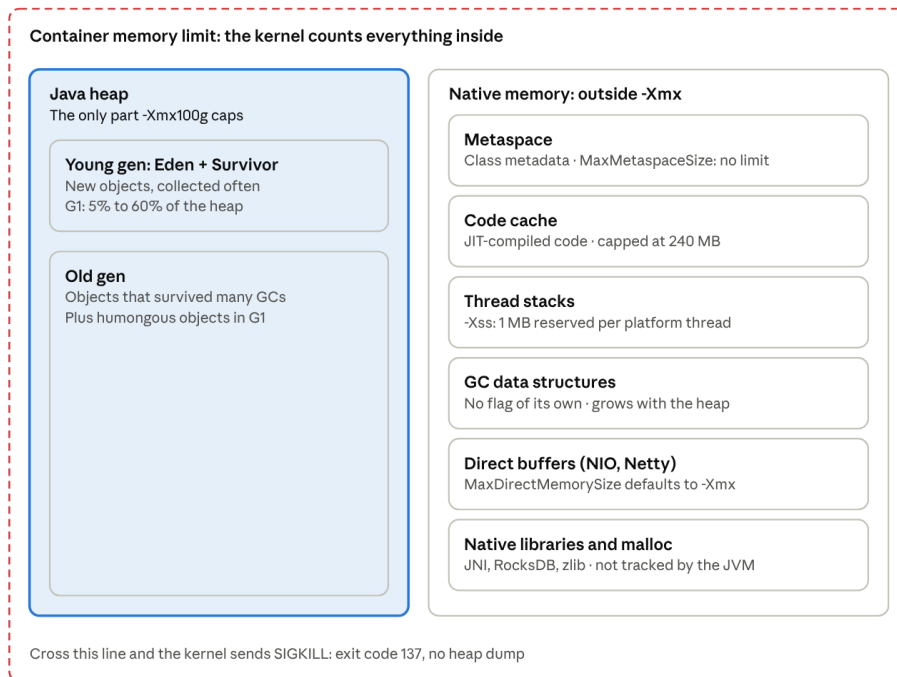
A Java `OutOfMemoryError` means the JVM could not satisfy an allocation. The detail message identifies the affected region: Java heap space, Metaspace, direct buffers or another resource. For heap exhaustion, the JVM can log a stack trace and, when configured, write a heap dump.

A container memory-limit OOM kill happens when cgroup memory use exceeds its limit. The kernel sends `SIGKILL`; Kubernetes typically reports `OOMKilled` with exit code 137, and the JVM has no opportunity to write a stack trace or heap dump. Exit code 137 alone only establishes `SIGKILL`, so confirm the termination reason and container events. Heap dashboards often look healthy right up to the kill.

`-Xmx` limits heap growth. It does not cap native memory or the container's total memory use.

Where the memory goes

`-Xmx` caps only the heap; the kernel counts the whole process



JVM process memory · the heap and six native regions

Only the shaded box obeys `-Xmx`. The native regions also contribute to memory use; their limits and resident footprint depend on the runtime, platform and workload. The diagram shows a typical HotSpot configuration, and the boxes are not drawn to scale.

Outside the heap

Native-memory limits differ by region. The following figures describe a typical 64-bit HotSpot configuration with tiered compilation; verify the flags on the JDK, collector and platform you deploy.

- **Metaspace:** class metadata. No limit unless you set `-XX:MaxMetaspaceSize`.
- Code cache: JIT-compiled code. Commonly capped at 240 MB with tiered compilation; the default depends on the runtime configuration.
- Thread stacks: commonly around 1 MB reserved per platform thread on 64-bit Linux. Pages become resident as stacks grow. Thread creation is constrained by native memory and OS or container limits.
- GC data structures: grow with the heap and workload. G1's marking, region and remembered-set structures consume additional memory; measure their contribution for your collector and JDK.
- Direct buffers (NIO, Netty): HotSpot's default NIO direct-buffer limit is commonly derived from the maximum heap. With `-Xmx100g` it can therefore permit roughly another 100 GiB unless capped. Some libraries also allocate native memory outside that limit.
- Native libraries (RocksDB, compression, crypto): their allocations may not appear in HotSpot Native Memory Tracking, and each library needs its own limits and observability.

The 32 GB cliff

With the usual 8-byte object alignment, HotSpot compressed ordinary object pointers address roughly 32 GiB: $2^{32} \times 8$ bytes. Above the supported range, heap references use 8 bytes instead of 4. The exact boundary depends on alignment and runtime configuration; do not treat `-Xmx31g` and `-Xmx32g` as a universal switch. Check `UseCompressedOops` in the runtime flags.

A heap just beyond that boundary can hold less application data than a smaller heap with compressed references. Cross it deliberately and measure the live set. On supported JDK 25 configurations, `-XX:+UseCompactObjectHeaders` can reduce object-header overhead to 8 bytes. JEP 519 cites about 22% lower heap use in one SPECjbb2015 configuration; that result is workload-specific. Header compression and ordinary object-reference compression are separate mechanisms.

-Xmx is a ceiling, not a reservation

The JVM commits heap memory only as the heap grows. A test JVM started with `-Xmx100g` had committed just 640 MB. A sizing mistake therefore shows up at peak traffic, not at deploy time.

- For workloads that need predictable resident capacity, consider setting `-Xms` equal to `-Xmx` with `-XX:+AlwaysPreTouch`. This touches the initial heap at startup and exposes some capacity shortfalls earlier, at the cost of startup time and immediate memory use; it does not preallocate future native memory.
- **Don't rely on `-XX:MaxRAMPercentage` alone.** Native memory follows threads, classes and libraries, not heap size. A fixed percentage is too little on small containers and wasteful on large ones.
- If Guaranteed QoS fits the workload, set equal CPU and memory requests and limits for every container. This improves protection during node-pressure eviction, but does not prevent a container from being OOM-killed when it exceeds its own memory limit.

A budget for a 100 GB heap

The following illustrative budget totals about 116 GiB for a 100 GiB G1 heap. It is a worked example, not a sizing rule: replace each line with peak measurements and consistent units. JVM flags such as `-Xmx100g` use binary units (GiB).

- Java heap: 100 GiB
- G1 data structures: 3 GiB
- Metaspace, capped: 1 GiB
- Code cache: 0.25 GiB
- Thread stacks, 1,000 threads: 1 GiB
- Direct buffers, capped: 4 GiB
- Native libraries and malloc: 2 GiB
- Headroom: 5 GiB
- **Container limit: about 116 GiB**

Measure it

Native Memory Tracking (NMT) shows what each JVM region has committed. Enable it at startup; Oracle documents a 5–10% overhead, so use staging or a canary.

```
java -XX:NativeMemoryTracking=summary ...
jcmd <pid> VM.native_memory summary scale=MB
```

Compare NMT with process RSS and cgroup usage, but do not expect them to match: committed memory is not the same as resident memory, and NMT omits some native allocations and mappings. Investigate large or growing gaps with OS-level tools. In production, monitor cgroup memory usage, limit, events and working set alongside heap usage.

Should it be one 100 GB JVM?

A single large JVM can make sense when the workload needs one shared in-memory dataset. Several smaller JVMs can retain compressed references and reduce individual heap dumps and failure scope, but add coordination and data-duplication costs. GC pause behavior depends on the collector and workload.

For a large heap with strict pause objectives, evaluate ZGC. It is designed for sub-millisecond pauses independent of heap size, but that design goal is not an end-to-end latency guarantee. ZGC uses its own pointer representation rather than compressed ordinary object pointers, so assess its memory, CPU and throughput trade-offs with representative load.

Checklist

1. Pick a side of the 32 GB line on purpose.
2. Evaluate equal -Xms and -Xmx with AlwaysPreTouch against startup and memory goals.
3. Cap Metaspace and direct memory.
4. Write the native-memory budget down.
5. Run NMT in staging and compare its total with RSS.
6. Alert on cgroup memory usage and events, not just the heap.

What's the biggest gap you've seen between -Xmx and a JVM's real memory use, and what caused it?

Sources

- [JEP 519: Compact Object Headers](#), OpenJDK
- [Native Memory Tracking, JDK 25 Virtual Machine Guide](#), Oracle
- JVM defaults quoted (code cache, Metaspace, compressed references at 100 GB, G1 settings) were read from -XX:+PrintFlagsFinal on OpenJDK 21.