Memory Management

Table of Contents

JVM Internals

Java Virtual Machine provides an environment that allows java applications to run in a platform-independent manner. JVM handles the execution of complied Java bytecode, memory management and interaction with native system resources. The JVM has several internal components, including class loading, memory management and execution.

Components of JVM

- Class Loader Subsystem
  The class loader is responsible for dynamically loading Java classes into memory at runtime. It reads the .class files containing bytecode and loads them into the JVM's memory.
- Runtime Data Areas
  The JVM organizes memory into several runtime data areas. These memory areas are created when the JVM starts and are used during the execution of Java programs
  1. Method Area
     Stores metadata about classes, such as class structures (field and method data), constant pool information, method bytecode, and static variables.
  2. Heap
     This is the area where java objects are dynamically allocated. The heap is managed by the garbage collector to free memory occupied by unreachable objects.
  3. Stack
     Java stack that stores local variables, method call information, and partial results.
  4. Program Counter (PC) Register
     PC register keeps track of the address of the next bytecode instruction to execute for that thread.
  5. Native Method Stack
     Stores the state of native (non-Java) method invocations. These are methods written in other languages like C or C++ and executed using the **Java Native Interface (JNI)**. When Java interacts with native libraries, such as through system calls or custom native code, the native method stack holds the state of these calls.

Comparison of Different Garbage Collectors

Aspect	Parallel Garbage Collector	G1 (Garbage First) Garbage Collector	Z Garbage Collector (ZGC)
Basic Concept	Multi-threaded garbage collection, optimizing for throughput.	Region-based garbage collection, optimizing for low-pause times.	Highly concurrent garbage collection, aiming for very short pauses (sub-10ms).
Type	Stop-the-world, multi-threaded garbage collection.	Mostly concurrent, region-based collection with occasional stop-the-world pauses.	Fully concurrent garbage collection with minimal stop-the-world phases.
Targeted Workload	Applications where high throughput is prioritized, and pause times can be longer.	Applications requiring predictable low-pause times, suitable for mixed workloads.	Large-scale applications requiring extremely low pause times, even with very large heaps (multi-terabyte).
Heap Structure	Monolithic heap divided into young and old generations.	Heap divided into multiple regions, with both young and old generations split into regions.	Heap divided into regions but without generational distinctions.
Young Generation Collection	Parallel Minor GC using multiple threads.	Concurrent region-based collection, collecting the young generation from regions first.	Fully concurrent collection with no stop-the-world phases, even for young generation objects.
Old Generation Collection	Parallel Major GC using multiple threads, with stop-the-world pauses.	Concurrent Mark-and-Sweep for old regions; regions with the most garbage are collected first.	Fully concurrent old generation collection, only minimal stop-the-world phases for root scanning.
GC Pauses	Long stop-the-world pauses, but high throughput.	Predictable, shorter pauses based on	Extremely short pauses, typically

		configured pause-time goals.	under 10ms, even for large heaps.
Throughput	High throughput due to parallel processing, but long pauses affect latency-sensitive applications.	Moderate throughput, balanced with low-pause times, designed for applications that need a mix of both.	High throughput with minimal impact on latency, designed for ultra-low-latency applications.
Pause Time Goals	Focused on maximizing throughput, so pause times can be longer (e.g., 100ms+).	Tunable pause time goals, generally aiming for short pauses (e.g., 50ms or less).	Aiming for sub-10ms pause times, regardless of heap size.
Garbage Collection Phases	Stop-the-world for both young and old generations, with multi-threaded collection.	Concurrent, with incremental stop-the-world phases for certain stages, but overall region-based collection reduces pause times.	Fully concurrent for most phases, including marking, sweeping, and compacting, with very short stop-the-world phases only during root scanning.
Scalability	Suitable for small to medium-sized heaps (a few GBs).	Scalable to large heaps (hundreds of GBs) with mixed workloads.	Designed for extremely large heaps, scaling up to multi-terabyte heaps.
Memory Overhead	Moderate memory overhead due to parallel threads.	Slightly higher memory overhead because of region-based structures.	Higher memory overhead due to metadata tracking for concurrency, but scales efficiently.
Use Case	High-throughput applications where latency is not critical (e.g., batch processing).	Applications requiring predictable latency and balanced throughput (e.g., web servers, database applications).	Large-scale, latency-sensitive applications with huge memory footprints (e.g., cloud services, financial systems).

Memory Management Best Practices

1. Avoid Creating Unnecessary Objects
   Reuse objects wherever possible instead of creating new ones unnecessarily. For example, use a `StringBuilder` for string concatenation inside loops instead of `String` objects to avoid creating multiple immutable objects.
2. Use Primitive Types Instead of Wrapper Classes
   Use primitive types (`int`, `long`, `boolean`) instead of their wrapper classes (`Integer`, `Long`, `Boolean`) when possible to avoid unnecessary object creation and overhead.
3. Be Mindful of Memory Leaks
   Ensure objects are not unintentionally retained in memory by static references, collections, or long-lived objects. A common example is forgetting to remove objects from collections (like `ArrayList` or `HashMap`) when they're no longer needed. Best practice is to use weak references or `WeakHashMap` when appropriate, especially for caching.
4. Explicitly Nullify Unused References
   Set unused object references to `null` when they're no longer needed, especially for long-lived objects in scopes with large memory allocations (like instance variables).
5. Optimize Data Structures
   Choose the right data structures based on your use case. For instance, avoid using `HashMap` or `ArrayList` if the number of entries is small or constant. Rather, consider using `EnumSet` or `EnumMap` when working with enums instead of general-purpose collections like `HashSet` or `HashMap`.
6. Use the Right Size for Data Structures
   Initialize collections (e.g., `ArrayList`, `HashMap`) with the right initial size when you know the size upfront to avoid resizing overhead.
7. Profile and Monitor Memory Usage
   Use JVM monitoring tools such as JVisualVM, JConsole, or Java Flight Recorder to profile memory usage, identify memory leaks, and optimize memory consumption.