Fine-tuning Performance of Java Applications

J. NANAVATI

SEMCOM, Vallabh Vidyanagar, Gujarat, India

Email: jaynanavati@gmail.com
Telephone: +91-9924471070

Abstract - The interpretation of the term 'Performance' of Java applications may vary. This paper primarily discusses problems of crawling of application and too much memory consumption, and fine-tuning performance of Java applications by altering parameters such as changing the code, finding the behaviour of the application in terms of the ratio of young to old objects, tuning the JVM accordingly and altering the GC parameters.

Keywords: Java Application Performance, Fine-tuning performance, OutOfMemoryError, JVM, GC

"(Received October 23rd 2019 / Accepted November 23rd, 2019)"

1. Introduction

The Java Virtual Machine has two primary jobs:

1. To Execute Code
2. To Manage Memory

Memory management by JVM includes acquiring memory from the operating system, managing memory in heap and stack, compaction of the heap and removal of garbaged objects.

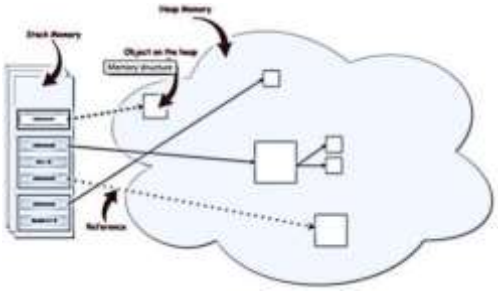

Figure 1 - Memory Management by JVM

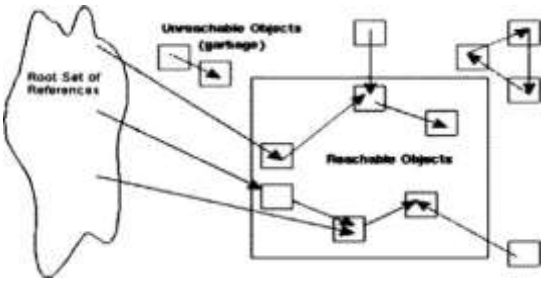

Figure 2 - Garbage Collection by JVM

Objects inside the square are reachable from the thread root set, while objects outside the square are not.

The garbage collection process takes place as follows:

1. The root set is traced to identify objects which are not referenced at all.
2. The garbaged objects from step-1 are placed in the finalizer.
3. The finalize() method for each of the garbaged object is executed.
4. Memory is freed up.

2. Generational Garbage Collection

Generational garbage collection scheme is used in the Java 2 VMs. The Java Heap is separated into two regions:

- New Objects
- Old Objects

The New Objects region is further divided into three smaller regions:

1. Eden, where objects are allocated.
2. Survivor semi-spaces: From and To.

In the Eden area, the track of memory allocated to objects is kept with the help of pointer increment.

In case the Eden area is full, the reachability test is performed by the GC and all the live objects are copied from the Eden to the To region.

Further, To becomes From i.e. the labels on the regions are interchanged. Now, the objects are there in the From area.

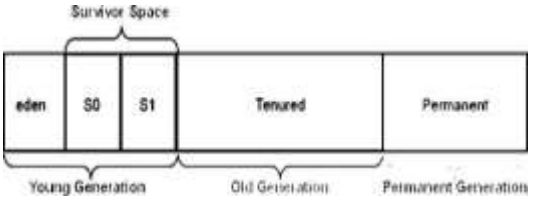

Figure 3 - Generational garbage collection scheme

Objects get created in New generation and then move to Survivor Spaces (SS) at every GC run. If these objects succeed in survival for long (enough to be considered old), they move to the Tenured generation.

The number of times an object needs to survive GC cycles to be considered old enough, can be configured.

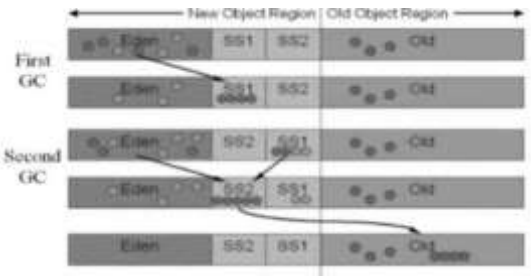

Figure 4 – GC Cycles

By default, Java has 2 separate threads for GC, one each for young (minor GC) and old generation (major GC). Garbage among the young generation is cleaned up by the minor GC whereas the major GC cleans up the garbage in the old generation. The JVM increases the current memory to facilitate creation of new object in case the major GC too fails to free required memory. This whole cycle can go on till the current memory reaches the MaxMemory for the JVM (default is 64MB for client JVM), after which JVM throws OutOfMemory Error.

3. JVM Process Memory

Managed Heap (Java Heap, PERM, Code Cache) + Native HEAP + Thread Memory <= 2GB (on windows)

Here,

Java Heap: This part of the memory is used when you create new java objects.

PERM: For reflective calls etc.

Code Cache: Contains JIT code and hotspot code.
Thread Memory = Thread Stack Size*No. of threads.
Native Heap: Used for native allocations.
Thread Memory: Used for thread allocations.

4. Problems

The following problems are often seen during execution of a Java application:

1. Crawling application
2. Too much memory consumption

The number and size of the live objects that are in the JVM at any given point of time governs the memory footprint of the application. This can be either due to valid objects that are required to stay in memory, or because programmer forgot to remove the reference to unwanted objects (typically known as 'Memory leaks' in the context of Java.

As the memory footprint exceeds the threshold limit, the JVM throws the java.lang.OutOfMemoryError.

4.1 Java.lang.OutOfMemoryError

Can occur due to 3 possible reasons:

1. JavaHeap space low to create new objects. Increase by -Xmx

(java.lang.OutOfMemoryError: Java heap space).
java.lang.OutOfMemoryError: Java heap space

MaxHeap=30528 KB TotalHeap=30528 KB FreHeap=170 KB UsedHeap=30357 KB

2. Permanent Generation low. Increase by XX:MaxPermSize=256m

(java.lang.OutOfMemoryError: PermGen space)
java.lang.OutOfMemoryError: PermGen space

MaxHeap=65088 KB TotalHeap=17616 KB Fr

eeHeap=9692 KB UsedHeap=7923 KB

3. java.lang.OutOfMemoryError: Out of swap space ...

5. Solutions

- For memory leak issues,
  - Change the code.
  - Define how fast your application code has to be, e.g., by specifying a maximum response time for all API calls or the number of records that you want to import within a specified time frame. After you've done that, you can measure which parts of your application are too slow and need to be improved. You can take a look at your code and start with the part that looks suspicious or where you feel that it might create problems. Or you use a profiler and get detailed information about the behavior and performance of each part of your code.
  - Use primitive data types instead of objects of wrapper classes.
  - Use StringBuilder to concatenate Strings programmatically.
- Find the behavior of your app in terms of the ratio of young to old objects, and then tune the JVM accordingly.
  - -ms, -Xms : sets the initial heap size (young and tenured generation ONLY, NOT Permanent)

    If the application starts with a large memory footprint, then you should set the initial heap to a large value so that the JVM does not consume cycles to keep expanding the heap.
  - -mx, -Xmx : sets the maximum heap size (young and tenured gen ONLY,NOT Perm) (default: 64mb)

This is the most frequently tuned parameter to suit the max memory requirements of the app. A low value overworks the GC so that it frees space for new objects to be created, and may lead to OOM.

A very high value can starve other apps and induce swapping. Hence, Profile the memory requirements to select the right value.

  - -XX:PermSize=256 -XX:MaxPermSize=256m
  - MaxPermSize default value (32MB for -client and 64MB for -server)
  - Tune this to increase the Permanent generation max size.
- Fine-tune with GC parameters.

  - -Xminf [0-1], -XX:MinHeapFreeRatio [0-100] : sets the percentage of minimum free heap space - controls heap expansion rate

  - -Xmaxf [0-1], -XX:MaxHeapFreeRatio [0-100] : sets the percentage of maximum free heap space - controls when the VM will return unused heap memory to the OS

  - -XX:NewRatio : sets the ratio of the old and new generations in the heap. A NewRatio of 5 sets the ratio of new to old at 1:5, making the new generation occupy 1/6th of the overall heap defaults: client 8, server 2
  - -XX:SurvivorRatio : sets the ratio of the survisior survivor space to the eden in the new object area. A SurvivorRatio of 6 sets the ratio of the three spaces to 1:1:6, making each survivor space 1/8th of the new object region

6. Conclusion

There can be various bottlenecks for the entire application, and JVM may be one of the reasons. However, other reasons such as JVM not tuned optimally to suit your application, Memory

leakages, JNI issues etc may also can not be ruled out. They need to be diagnosed, analyzed and then fixed.

References

[1] Java Performance Tuning, Jack Shirazi p. 67-75 ISBN: 978-0-596-00377-7

[2] Java Performance: The Definitive Guide, Scott Oaks, p. 18-24 ISBN: 978-4-149-35845-7

[3] Java Performance Companion, Charlie Hunt, ISBN: 978-0-133-79682-7

[4] Systems Performance: Enterprise and the Cloud, Brendan Gregg, ISBN: 978-1-333- 9009-4

[5] Java Performance and Scalability: A Quantitative Approach,Henry H. Liu, ISBN:

[6] The Well-Grounded Java Developer, Benjamin J Evans, Martijn Verburg, ISBN: 978-1- 617-29006-0

[7] Java Performance, Binu John, Charlie Hunt, ISBN: 978-0-137-14252-1

[8] https://www.javatpoint.com/Garbage-Collection (Last visited on 6/4/2019)

[9] https://www.journaldev.com/4098/java-heap-space-vs-stack-memory (Last visited on 6/4/2019)

[10] https://www.yourkit.com/docs/kb/sizes.jsp (Last visited on 8/4/2019)