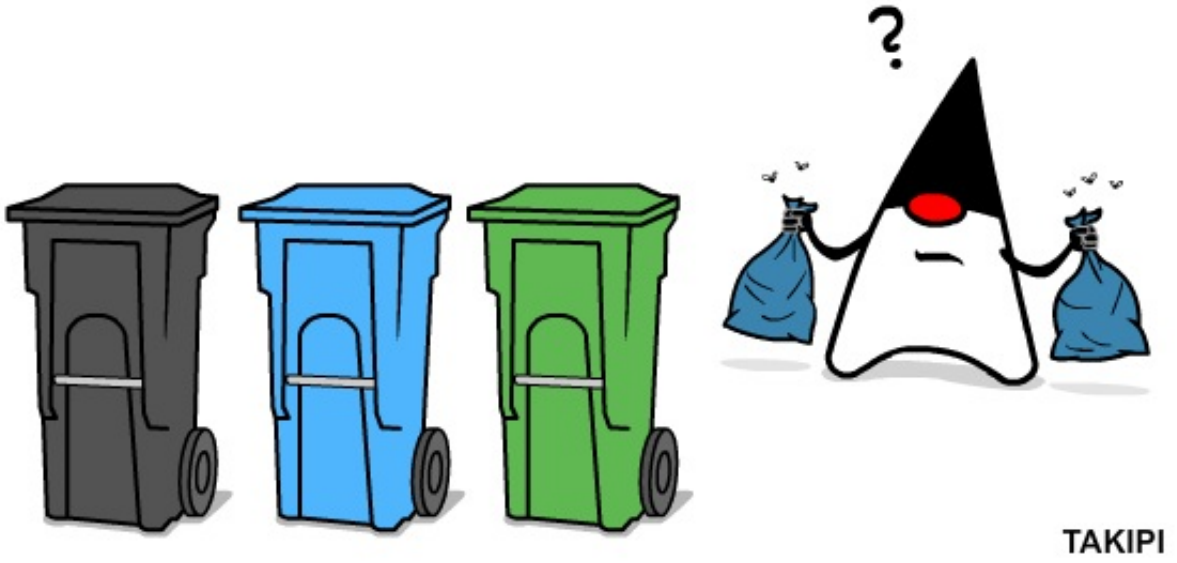
Garbage Collectors – Serial vs. Parallel vs. CMS vs. G1 (and what's new in Java 8)

By Tal Weiss — September 3, 2014 — 7 Comments



The 4 Java Garbage Collectors – How the Wrong Choice Dramatically Impacts Performance

The year is 2014 and there are two things that still remain a mystery to most developers – Garbage collection and understanding the opposite sex. Since I don't know much about the latter, I thought I'd take a whack at the former, especially as this is an area that has seen some major changes and improvements with Java 8, especially with the removal of the PermGen and some new and exciting optimizations (more on this towards the end).

When we speak about garbage collection, the vast majority of us know the concept and employ it in our everyday programming. Even so, there's much about it we don't understand, and that's when things get painful. One of the biggest misconceptions about the JVM is that it has one garbage collector, where in fact it provides four different ones, each with its own unique advantages and disadvantages. The choice of which one to use isn't automatic and lies on your shoulders and the differences in throughput and application pauses can be dramatic.

What's common about these four garbage collection algorithms is that they are generational, which means they split the managed heap into different segments, using the age-old assumptions that most objects in the heap are short lived and should be recycled quickly. As this too is a well-covered area, I'm going to jump directly into the different algorithms, along with their pros and their cons.

1. The Serial Collector

The serial collector is the simplest one, and the one you probably won't be using, as it's mainly designed for single-threaded environments (e.g. 32 bit or Windows) and for small heaps. This collector freezes all application threads whenever it's working, which disqualifies it for all intents and purposes from being used in a server environment.

How to use it: You can use it by turning on the -XX:+UseSerialGCJVM argument,

2. The Parallel / Throughput collector

Next off is the Parallel collector. This is the JVM's default collector. Much like its name, its biggest advantage is that is uses multiple threads to scan through and compact the heap. The downside to the parallel collector is that it will stop application threads when performing either a minor or full GC collection. The parallel collector is best suited for apps that can tolerate application pauses and are trying to optimize for lower CPU overhead caused by the collector.

3. The CMS Collector

Following up on the parallel collector is the CMS collector ("concurrent-mark-sweep"). This algorithm uses multiple threads ("concurrent") to scan through the heap ("mark") for unused objects that can be recycled ("sweep"). This algorithm will enter "stop the world" (STW) mode in two cases: when initializing the initial marking of roots (objects in the old generation that are reachable from thread entry points or static variables) and when the application has changed the state of the heap while the algorithm was running concurrently, forcing it to go back and do some final touches to make sure it has the right objects marked.

The biggest concern when using this collector is encounteringpromotion failures which are instances where a race condition occurs between collecting the young and old generations. If the collector needs to promote young objects to the old generation, but hasn't had enough time to make space clear it, it will have to do so first which will result in a full STW collection – the very thing this CMS collector was meant to prevent. To make sure this doesn't happen you would either increase the size of the old generation (or the entire heap for that matter) or allocate more background threads to the collector for him to compete with the rate of object allocation.

Another downside to this algorithm in comparison to the parallel collector is that it uses more CPU in order to provide the application with higher levels of continuous throughput, by using multiple threads to perform scanning and collection. For most long-running server applications which are adverse to application freezes, that's usually a good trade off to make. Even so, this algorithm is not on by default. You have to specifyXX:+USeParNewGC to actually enable it. If you're willing to allocate more CPU resources to avoid application pauses this is the collector you'll probably want to use, assuming that your heap is less than 4Gb in size. However, if it's greater than 4GB, you'll probably want to use the last algorithm – the G1 Collector.

4. The G1 Collector

The Garbage first collector (G1) introduced in JDK 7 update 4 was designed to better support heaps larger than 4GB. The G1 collector utilizes multiple background threads to scan through the heap that it divides into regions, spanning from 1MB to 32MB (depending on the size of your heap). G1 collector is geared towards scanning those regions that contain the most garbage objects first, giving it its name (Garbage first). This collector is turned on using the –XX:+UseG1GC flag.

This strategy the chance of the heap being depleted before background threads have finished scanning for unused objects, in which case the collector will have to stop the application which will result in a STW collection. The G1 also has another advantage that is that it compacts the heap on-the-go, something the CMS collector only does during full STW collections.

Large heaps have been a fairly contentious area over the past few years with many developers moving away from the single JVM per machine model to more micro-service, componentized architectures with multiple JVMs per machine. This has been driven by many factors including the desire to isolate different application parts, simplifying deployment and avoiding the cost which would usually come with reloading application classes into memory (something which has actually been improved in Java 8).

Even so, one of the biggest drivers to do this when it comes to the JVM stems from the desire to avoid those long "stop the world" pauses (which can take many seconds in a large collection) that occur with large heaps. This has also been accelerated by container technologies like Docker that enable you to deploy multiple apps on the same physical machine with relative ease.

Java 8 and the G1 Collector

Another beautiful optimization which was just out with Java 8 update 20 for is the G1 Collector String deduplication. Since strings (and their internal char[] arrays) takes much of our heap, a new optimization has been made that enables the G1 collector to identify strings which are duplicated more than once across your heap and correct

them to point into the same internal char[] array, to avoid multiple copies of the same string from residing inefficiently within the heap. You can use the -XX:+UseStringDeduplicationJVM argument to try this out.

Java 8 and PermGen

One of the biggest changes made in Java 8 was removing the permgen part of the heap that was traditionally allocated for class meta-data, interned strings and static variables. This would traditionally require developers with applications that would load significant amount of classes (something common with apps using enterprise containers) to optimize and tune for this portion of the heap specifically. This has over the years become the source of many OutOfMemory exceptions, so having the JVM (mostly) take care if it is a very nice addition. Even so, that in itself will probably not reduce the tide of developers decoupling their apps into multiple JVMs.

Each of these collectors is configured and tuned differently with a slew of toggles and switches, each with the potential to increase or decrease throughput, all based on the specific behavior of your app. We'll delve into the key strategies of configuring each of these in our next posts.

In the meanwhile, what are the things you're most interested in learning about regarding the differences between the different collectors? Hit me up in the comments section 🙂

Additional reading –

1. A really great in-depth review of the G1 Collector on InfoQ.

2. Java performance – The definitive guide. My favorite book on Java performance.

3. More about String deduplication on the CodeCentric blog.