How Garbage Collection Really Works

Garbage collection collects and discards dead objects?

YES /NO

- ✓ In reality, Java garbage collection is doing the opposite!
- ✓ Live objects are tracked and everything else designated garbage.

GC *Decisioning*

1. Do I really need a Low Latency

   No? Use Parallel Gc

2. Do I really need a Big Heap?

   Yes? Thing again!

3. Do I really need a Big Heap?

   No? Use small heap & Parallel GC

   Yes? Try CMS 1st

4. Is CMS performing well?

   Yes? Done!

   No? Tune it!

5. Is tuned CMS Performing well?

   Yes? Done!

   No? Tune it more!

6. Is CMS performing well now?

   Yes? Done!

   No? Do you really need such big heap?

7. Yes, I really need a Big Heap a Low Pauses!

   This is the moment where you should start considering to use G1!

Garbage-Collection Roots—The Source of All Object Trees

Every object tree must have one or more root objects. As long as the application can reach those roots, the whole tree is reachable.

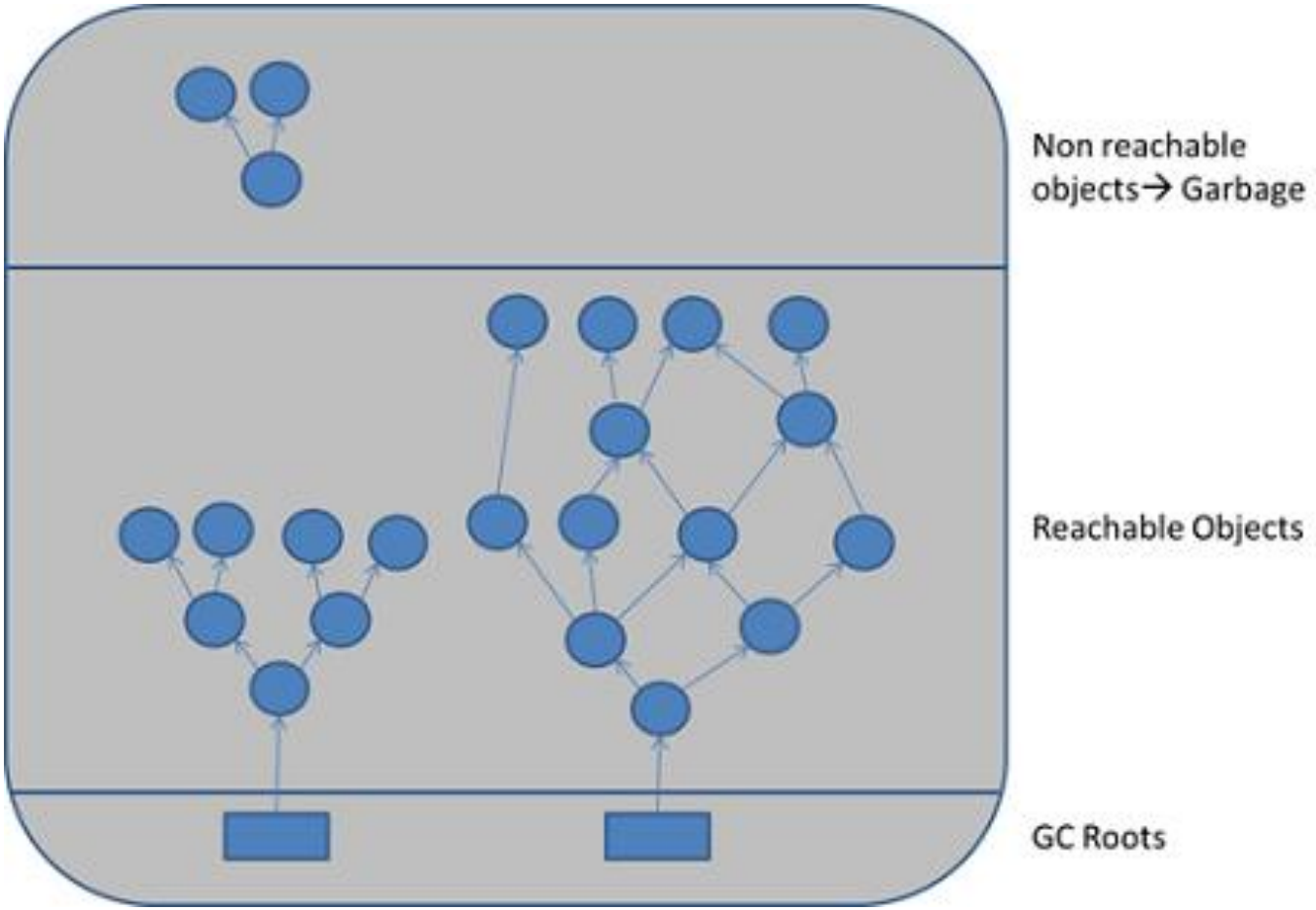
There are four kinds of GC roots in Java:

Local variables are kept alive by the stack of a thread. This is not a real object virtual reference and thus is not visible. For all intents and purposes, local variables are GC roots.

Active Java threads are always considered live objects and are therefore GC roots. This is especially important for thread local variables.

Static variables are referenced by their classes. This fact makes them de facto GC roots. Classes themselves can be garbage-collected, which would remove all referenced static variables.

JNI References are Java objects that the native code has created as part of a JNI call. Objects thus created are treated specially because the JVM does not know if it is being referenced by the native code or not.



GC roots are objects that are themselves referenced by the JVM and thus keep every other object from being garbage-collected.

Therefore, a simple Java application has the following GC roots:

Local variables in the main method
The main thread
Static variables of the main class

Marking and Sweeping Away Garbage

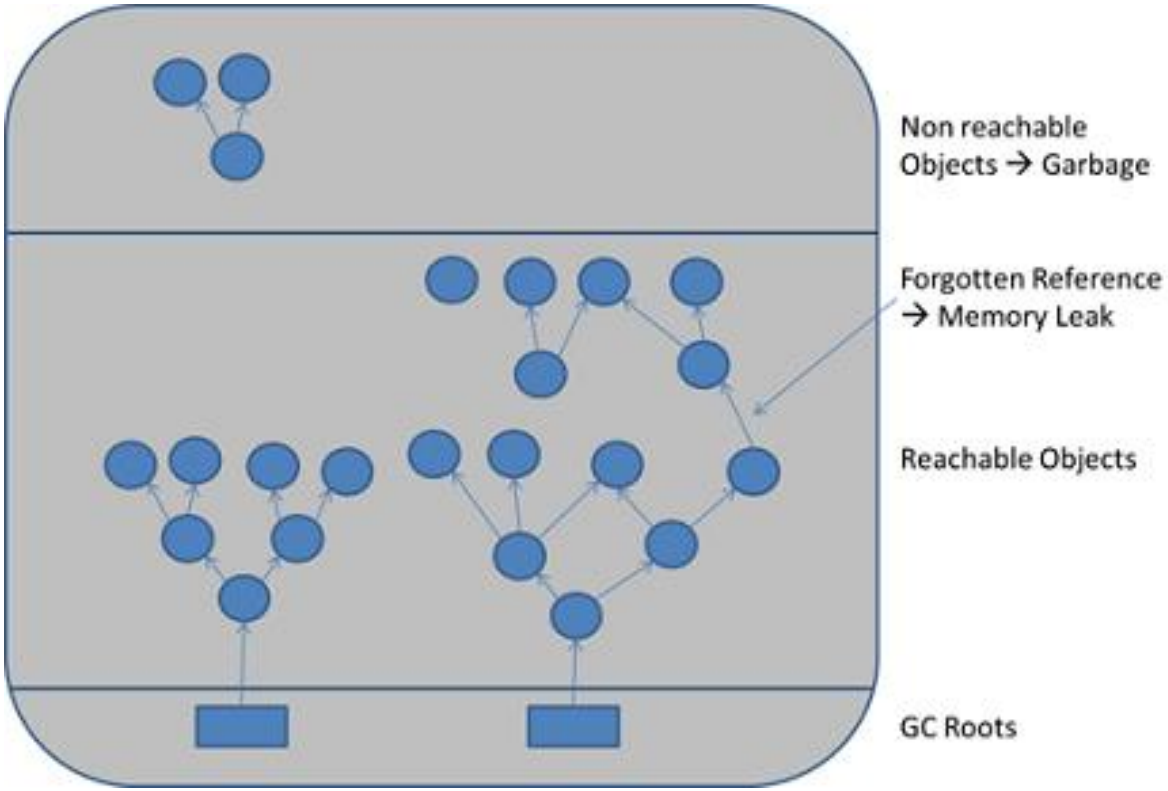
To determine which objects are no longer in use, the JVM intermittently runs what is very aptly called a mark-and-sweep algorithm

✓ The algorithm traverses all object references, starting with the GC roots, and marks every object found as alive.

✓ All of the heap memory that is not occupied by marked objects is reclaimed. It is simply marked as free, essentially swept free of unused objects.

Garbage collection is intended to remove the cause for classic memory leaks: unreachable-but-not-deleted objects in memory.

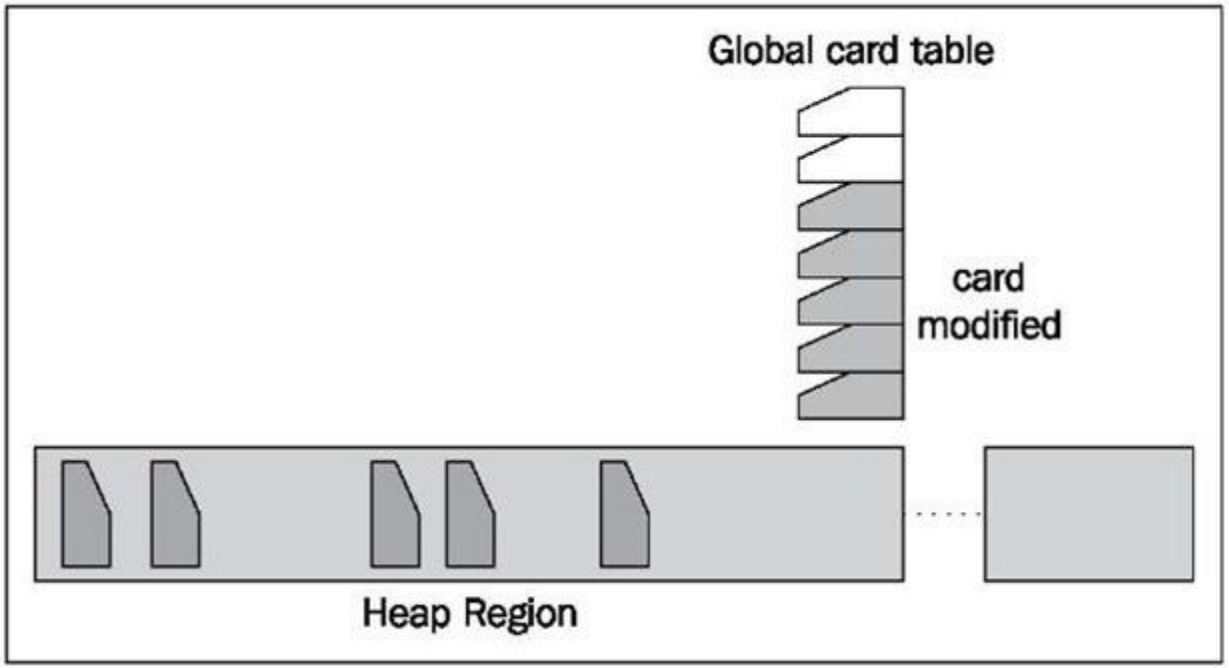
However, It’s possible to hav’e unused objects that are still reachable by an application because the developer simply forgot to dereference them. Such objects cannot be garbage-collected.



When objects are no longer referenced directly or indirectly by a GC root, they will be removed

Global Card Table

Each Region is in turn broken down into 512 bytes pieces called cards. For each card, there is one entry in the global card table.



This association helps to track which cards are modified (also known as "remembered set"), concentrating its collection and compaction activity first on the areas of the heap that are likely to be full of reclaimable objects, thus improving its efficiency.

G1 uses a pause prediction model to meet user-defined pause time targets. This helps minimize pauses that occur with the mark and sweep collector, and should show good performance improvements with long running applications.

Remember Set(RSet):-

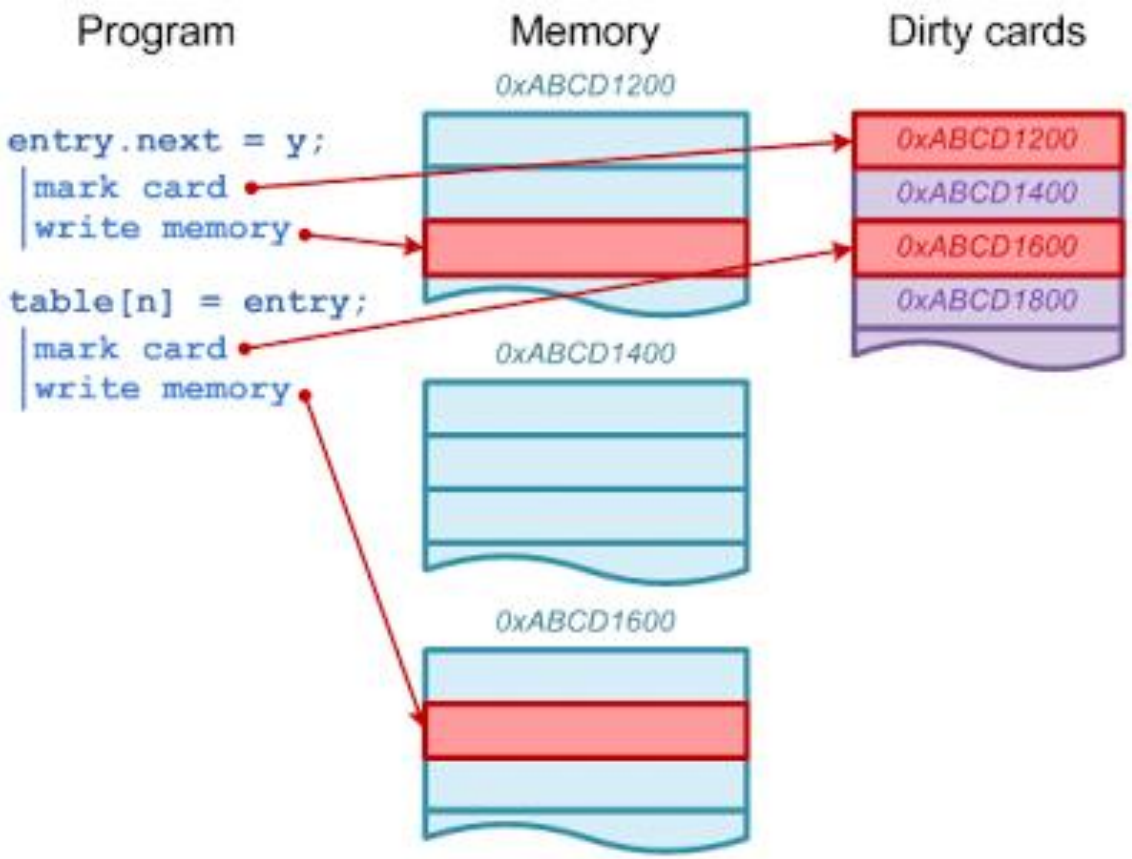
The RSet is like a card table for each region. It records references from other regions to objects in this region.

A regions’s Remember Set consists of card addresses.

These cards contain locations of references that point into this region.

Dirty cards write-barrier (card marking)

Principle of dirty card write-barrier is very simple. Each time when program modifies reference in memory, it should mark modified memory page as dirty. There is a special card table in JVM and each 512 byte page of memory has associated byte in card table



Young space collection algorithm

Before start collecting live objects, JVM should find all root references.

Root references for minor GC are:

- ✓ references from stack and
- ✓ all references from old space.

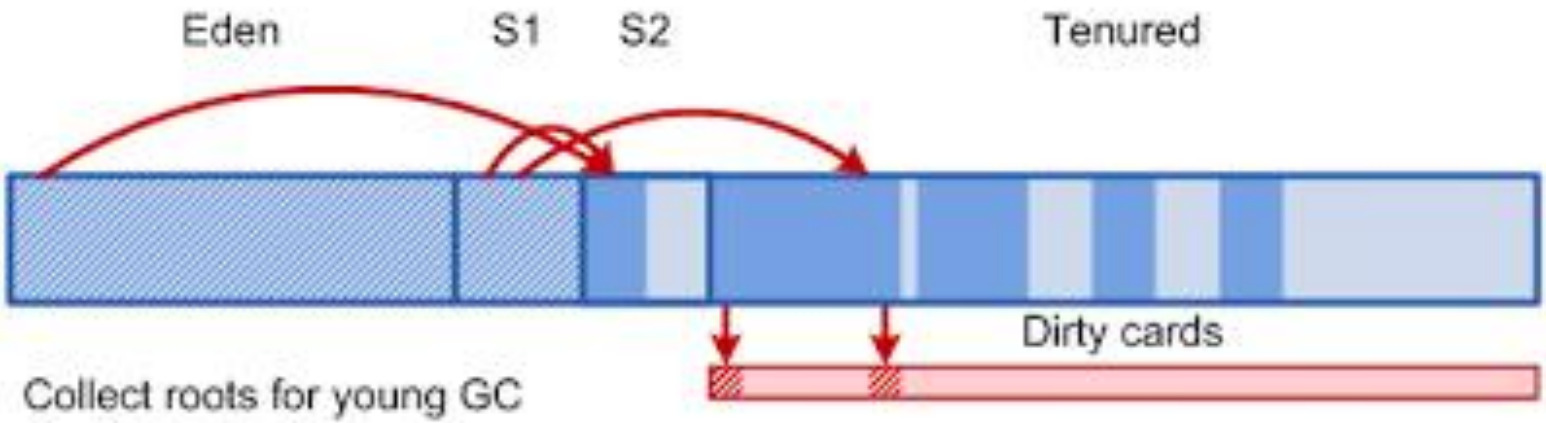
Normally collection of all reference from old space will require scanning through all objects in old space. That is why we need write-barrier.

All objects in young space have been created (or relocated) since last reset of write-barrier, so non-dirty pages cannot have references into young space. This means we can scan only object in dirty pages.

Once initial reference set is collected, dirty cards are reset and JVM starts coping of live objects from Eden and one of survivor spaces into other survivor space.

JVM only need to spend time on live objects.

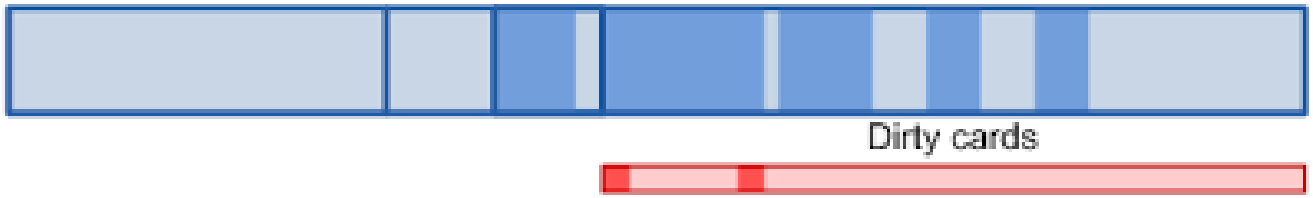
Relocating of object also requires updating of references pointing to it.



Collect roots for young GC

- Clean cards
- Recursive copy of live objects (only live objects are traversed)

While JVM is updating references to relocated object, memory pages get marked again, so we can be sure what on next young GC only dirty pages has references to young space.



Time of young GC

Young space collection happens during stop-the-world pause (all non-GC-related threads in JVM are suspended). Wall clock time of stop-the-world pause is very important factor for applications (especially applications requiring fast response time). Parallel execution affects wall clock time of pause but not work effort to be done.

Let's summarize components of young GC pause. Total pause time can be written as:
$T_{young} = T_{stack\_scan} + T_{card\_scan} + T_{old\_scan} + T_{copy}$;
there T_{young} is total time of young GC pause, T_{stack_scan} is time to scan root in stacks, T_{card_scan} is time to scan card table to find dirty pages, T_{old_scan} is time to scan roots in old space and T_{copy} is time to copy live objects (1). Thread stack are usually very small, so major factors affecting time of young GC
is T_{card_scan} , T_{old_scan} and T_{copy}.

GC attacks by Linux

IO Starvation

-> Symptom : GC log shows "low user time, low system time, long pause"

-> Cause : GC threads stuck in kernal waiting for IO, usually due to journal commits or FS flush of changes by gzip of log rolling

Memory starvation:

-> Symptom : GC log shows "low user time, high system time, long pause"

-> Cause : Memory pressure triggers swapping or scanning for free memory

Solutions for GC-attacks

IO Starvation

-> Strategy : Even out workload to disk drives
(flush every 5s rather than 30s)

sysctl -w vm.dirty_writeback_centisecs = 500

sysctl -w vm.dirty_expire_centisecs = 500

Memory Starvation

-> Strategy : Pre-allocate memory to JVM heap and protect it against swapping or scanning

-> Turn on -XX:+AlwaysPreTouch option in JVM

-> sysctl -w vm.swappiness=0 to protect heap and anonymous memory

-> JVM start up has 2 second delay to allocate all memory (17GB!!)

Garbage Collection Optimization for High-Throughput and Low-Latency Java Applications

The following combination of options enabled an I/O intensive application with a large cache and mostly short-lived objects after initialization to achieve 60ms pause times for the 99.9th percentile latency:

-server -Xms40g -Xmx40g -XX:MaxDirectMemorySize=4096m -XX:PermSize=256m -XX:MaxPermSize=256m -XX:NewSize=6g -XX:MaxNewSize=6g -XX:+UseParNewGC -XX:MaxTenuringThreshold=2 -XX:SurvivorRatio=8 -XX:+UnlockDiagnosticVMOptions -XX:ParGCCardsPerStrideChunk=32768 -XX:+UseConcMarkSweepGC -XX:CMSParallelRemarkEnabled -XX:+ParallelRefProcEnabled -XX:+CMSClassUnloadingEnabled -XX:CMSInitiatingOccupancyFraction=80 -XX:+UseCMSInitiatingOccupancyOnly -XX:+AlwaysPreTouch -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime -XX:-OmitStackTraceInFastThrow.

-XX:+PrintTenuringDistribution -- print byte counts of objects in age groups of the survivor spaces for YoungGen

-XX:+PrintPromotionFailure -- show information about failures to move objects from pools for younger objects to pools for older ones

-XX:+UseBiasedLocking
Enables biased locking feature.

"Biased locking is good for un-contended locks, but is worth disabling (-XX:-UseBiasedLocking).

if there is contention (monitor with -XX:+PrintSafepointStatistics -XX:+PrintGCApplicationStoppedTime which let's you see safepoint statistics for stopped times)"

-XX:+UseStringDeduplication
(G1GC, minimum Java 8u20)

-XX:+PrintStringDeduplicationStatistics

-XX:-OmitStackTraceInFastThrow
to turn off the ability of the JVM to throw fast stackless exceptions

```
[GC concurrent-string-deduplication, 2893.3K->2672.0B(2890.7K), avg 97.3%, 0.0175148 secs]
  [Last Exec: 0.0175148 secs, Idle: 3.2029081 secs, Blocked: 0/0.0000000 secs]
    [Inspected:          96613]
      [Skipped:              0(  0.0%)]
      [Hashed:           96598(100.0%)]
      [Known:                2(  0.0%)]
      [New:              96611(100.0%)   2893.3K]
    [Deduplicated:         96536( 99.9%)   2890.7K( 99.9%)]
      [Young:                0(  0.0%)      0.0B(  0.0%)]
      [Old:              96536(100.0%)   2890.7K(100.0%)]
  [Total Exec: 452/7.6109490 secs, Idle: 452/776.3032184 secs, Blocked: 11/0.0258406 secs]
    [Inspected:         27108398]
      [Skipped:              0(  0.0%)]
      [Hashed:         26828486( 99.0%)]
      [Known:            19025(  0.1%)]
      [New:            27089373( 99.9%)    823.9M]
    [Deduplicated:     26853964( 99.1%)    801.6M( 97.3%)]
      [Young:            4732(  0.0%)    171.3K(  0.0%)]
      [Old:            26849232(100.0%)    801.4M(100.0%)]
  [Table]
    [Memory Usage: 2834.7K]
    [Size: 65536, Min: 1024, Max: 16777216]
    [Entries: 98687, Load: 150.6%, Cached: 415, Added: 252375, Removed: 153688]
    [Resize Count: 6, Shrink Threshold: 43690(66.7%), Grow Threshold: 131072(200.0%)]
    [Rehash Count: 0, Rehash Threshold: 120, Hash Seed: 0x0]
    [Age Threshold: 3]
  [Queue]
    [Dropped: 0]
```

These are the results after running the app for 10 minutes. As we can see String Deduplication was executed 452 times and "deduplicated" 801.6 MB Strings. String Deduplication inspected 27 000 000 Strings.

When we compare memory consumption from Java 7 with the standard Parallel GC to Java 8u20 with the G1 GC and enabled String Deduplication the heap dropped approximatley 50%:

Summarizing Remembered Sets

The option `-XX:+G1SummarizeRSetStats` can be used to provide a window into the total number of RSet coarsenings to help determine if concurrent refinement threads are able to handle the updated buffers.

-XX:G1SummarizeRSetStatsPeriod=n
This option summarizes RSet statistics every *n*th GC pause