OPEN JDK

Understanding Garbage Collectors

How the default garbage collectors work

by Christine H. Flood

November 21, 2019

Garbage collection (or GC) is an automated way to reclaim for reuse memory that is no longer in use. Unlike other languages in which objects are allocated and destroyed manually, with GC, Java programmers don't need to pick up and examine each object to decide whether it is needed. Instead, the omniscient GC housekeeper process works behind the scenes quietly discarding objects that are no longer useful and tidying up what's left. This decluttering leads to an efficient program.

This article is updated and abriged from the article "The New Garbage Collectors in OpenJDK," which was originally published in *Java Magazine* on March 1, 2016

What is GC?

The JVM organizes program data into objects. Objects contain fields (data) in a managed address space called a *heap*. Imagine the Java class below, which represents a simple binary tree node.

```
class TreeNode {
    public TreeNode left, right;
    public int data;
    TreeNode(TreeNode l,  TreeNode r, int d) {
        left = l; right = r; data = d;
    }
    public void setLeft(TreeNode l) { left = l;}
    public void setRight(TreeNode r) {right = r;}
}
```

Now imagine the following operations performed on this class.

```
TreeNode left = new TreeNode(null, null, 13);
TreeNode right = new TreeNode(null, null, 19);
TreeNode root = new TreeNode(left, right, 17);
```

Here, I've created a binary tree with a root of 17, a left subnode of 13, and a right subnode of 19 (see **Figure 1**).

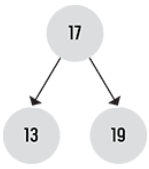

Figure 1. A three-node tree

Suppose I then replace the right subnode, leaving subnode 19 as unconnected garbage:

```
root.setRight(new TreeNode(null, null, 21));
```

This results in the situation shown in **Figure 2**.

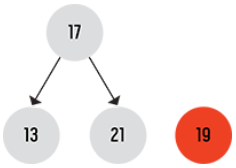

Figure 2. The same tree with one subnode replaced

As you can imagine, in the process of constructing and manipulating data structures, the heap will start to look like **Figure 3**.

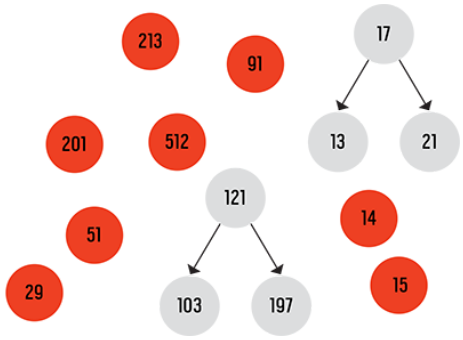

Figure 3. A heap with many unused data items in it

Compacting the data means changing its address in memory. The Java program expects to find an object at a particular address. If the garbage collector moves the object, the Java program needs to know the new location. The easiest way to do this is to stop all the Java threads, compact all the objects, update all the references to the old addresses to now point to the new addresses, and resume the Java program. However, this approach can lead to long periods (called *GC pause times*) when the Java threads aren't running.

Java programmers aren't happy when their applications aren't running. There are two popular strategies for decreasing GC pause times. The GC literature refers to them as *concurrent algorithms* (doing work while the program is running) and *parallel algorithms* (employing more threads to get the work done faster while the Java threads are stopped). The default garbage collector in JDK 8 (which can be manually specified on the command line with `-XX:+UseParallelGC`) adopts the parallel strategy. It uses many GC threads to get impressive throughput.

Parallel Garbage Collector

The parallel garbage collector segregates objects into two regions—*young* and *old*—according to how many GC cycles they have survived. Young objects are initially allocated in the young region, and the compaction step keeps them in that region until they have survived a certain number of young collections. If they live long enough, they are promoted to the old generation. The theory is that rather than pausing to collect the entire heap, which would take too long, you can collect just the part of the heap that is likely to contain short-lived objects. Eventually it will become necessary to collect the older objects as well.

In order to collect just the younger objects, the garbage collector needs to know which objects in the old generation reference objects in the young generation. The old objects need to be updated to reference the new locations for the new objects. The JVM does this by maintaining a summarization data structure called the *card table.* Whenever a reference is written into an old-generation object, the card table is marked so that during the next young GC cycle, the JVM can scan this

card looking for old-to-young references. With these references known, the parallel garbage collector is able to identify which objects to cull and which references to update. It uses multiple GC threads to get the work done faster while it has paused the program.

Garbage-First Garbage Collector

The JDK garbage collector named G1 uses both parallel and concurrent threads. It uses concurrent threads to scan the live objects while the Java program is running. It uses parallel threads to copy objects quickly and keep pause times low.

G1 divides the heap into many regions. A region might be either an old region or a young region at any time during the program run. The young regions must be collected at every GC pause, but G1 has the flexibility to collect as many or as few old regions as it predicts it can collect within the user-specified pause-time goal. This flexibility allows G1 to focus the old-object GC work on the areas of the heap that have the most garbage. It also enables G1 to tune collection pause times based on user-specified pause times.

As shown in **Figure 4**, G1 will freely compact objects into new regions.

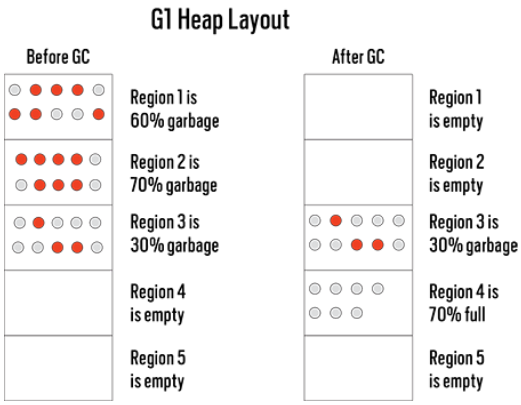

Figure 4. Before and after a G1 run. Regions 1 and 2 are compacted into region 4. New objects may be allocated to fill region 4. Region 3 is untouched because there would be too much copying work (70 percent) for too little space reclamation (30 percent).

G1 knows how much data is live in each region and the approximate time it takes to copy that live data. If the user is interested in minimal pause times, G1 can choose to evacuate only a few regions. If the user is not worried about pause times or has stated a fairly large pause-time goal, G1 might choose to include more regions.

G1 must maintain a card table data structure so that it can collect only young regions. It also must maintain a record for each old region that other old regions have references to. This data structure is called an *into remembered set*.

The downside of specifying small pause times is that G1 might not be able to keep up with the program allocation rate, in which case it will eventually give up and fall back to a full stop-the-world GC mode. This means that both the scanning and the copying work are done while the Java threads are stopped. Note that if the GC can't meet the pause-time goal with partial collections, then a full GC is guaranteed to exceed the allocated time.

In sum, G1 is a good overall collector that balances throughput and pause-time constraints.

Shenandoah Garbage Collector

The Shenandoah garbage collector is an OpenJDK project that became part of the part of OpenJDK 12 distribution and is being back-ported to JDK 8 and 11. It uses the same region-based heap layout as G1 and employs the same concurrent scanning threads to calculate the amount

of live data in each region. It differs in the way it handles the compaction stage.

Shenandoah compacts the data concurrently. As a consequence, Shenandoah doesn't need to limit the number of regions it collects in order to minimize application pause times.

Shenandoah compacts the data concurrently. (The sharp-eyed among you will have noticed that this means it might need to move objects around while the application is trying to read them or write to them; don't worry—I'll come to that in a second.) As a consequence, Shenandoah doesn't need to limit the number of regions it collects in order to minimize application pause times. Instead it picks all the most fruitful regions—that is, regions that have very few live objects or, conversely, a lot of dead space. The only steps that introduce pauses are those associated with certain bookkeeping tasks performed at the beginning and end of scanning.

The key difficulty with Shenandoah's concurrent copying is that the GC threads doing the copying work and the Java threads accessing the heap need to agree on an object's address. This address might be stored in several places, and the update to the address must appear to happen simultaneously. Like most thorny problems in computer science, the solution is to add a level of indirection.

Objects are allocated with extra space for an indirection pointer. When the Java threads access the object, they first read the indirection pointer to see whether the object has moved. When the garbage collector moves an object, it updates the indirection pointer to point to the new location. New objects are allocated with an indirection pointer that points to themselves. Only when an object is copied during GC will the indirection pointer point to somewhere else.

This indirection pointer is not free. It has a cost in both space and time to read the pointer and find the current location of the object. These costs are less than you might think. Spacewise, Shenandoah does not need the off-heap data structures used to support partial collections like the card table and the into remembered sets. Timewise, there are various strategies to eliminate read barriers. The optimizing JIT compiler can realize that the program is accessing an immutable field, such as an array size. It's correct in those cases to read either the old or the new copy of the object so no indirection read is required. In addition, if the Java program reads multiple fields from the same object, the JIT may recognize this and remove the subsequent reads of the forwarding pointer.

If the Java program writes to an object that Shenandoah is copying, a race condition occurs. This is solved by having the Java threads cooperate with the GC threads. If the Java threads are about to write to an object that has been targeted for copying, the Java thread will first copy the object to its own allocation area, check to see that it was the first to copy the object, and then perform the write. If the GC thread copied the object first, then the Java thread can unwind its allocation and use the GC copy.

Shenandoah eliminates the need to pause during the copying of live objects, thus providing much shorter pause times.

Conclusion

Since this article was first published, GC in the JDK has evolved in new ways. Other articles in this issue detail some of those changes, which can be best understood in light of the GC mechanics in this article. —*Ed.*

Also in This Issue

Christine H. Flood

Christine H. Flood is a principal software engineer for the Java platform at Red Hat, where she works on garbage collection.

Share this Page