Garbage Collection

Garbage Collection - Java

Garbage collection allows new objects to be created without explicitly defining their memory allocation and deallocation by reclaiming memory for reuse. The garbage collectors intention is to allow for faster development with less boilerplate code and memory leaks. However, in Java sometimes too many ojects are created and reclaimed, causing issues.

To reallocate memory, the Java garbage collector tracks live objects and designates everything else as garbage.

**The Heap**

The heap is the area of memory used for dynamic allocation, given to the environement by the OS. All objects (classes, statics and code) are put in the heap and since the heap is allocated in advance:

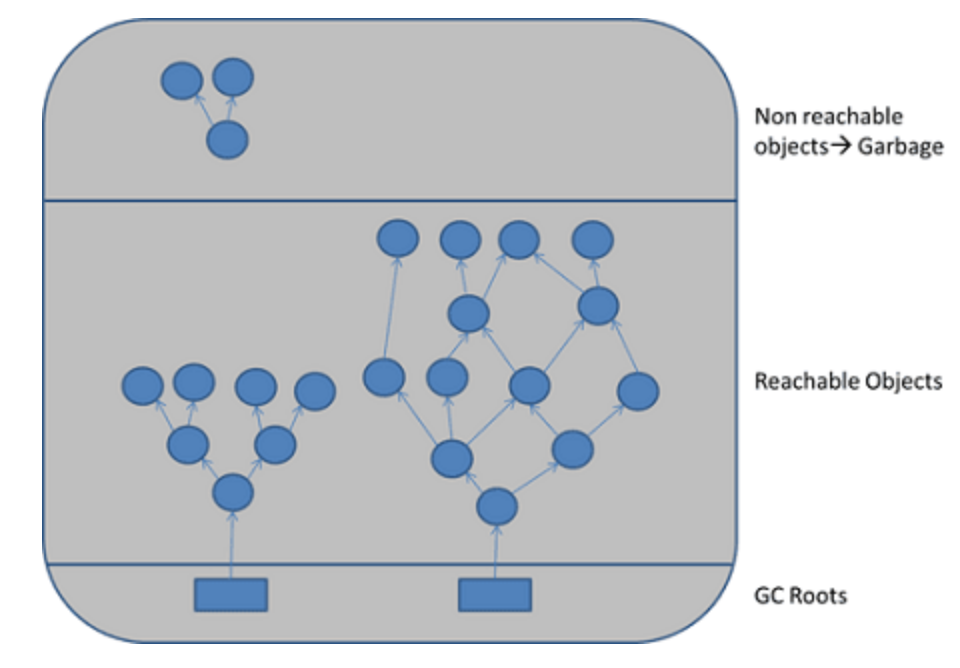
* Object creation is faster since it there is no global synchonisation with the OS for each object
* Object deletion simply assigns the memory for future object allocation, nothing is actually deleted from the memory and given back to the OS

**Object Tree**

Every object tree must have a root, so long as the application can reach all the roots all objects will be avaliable. Garbage Collection roots are always reachable, and there are four types:

* Local variable - kept alive by stack of thread
* Active Java threads - always considered live
* Static variables - referenced by class, so GC roots unless class collected
* JNI references - objects which are created by native codeas part of JNI call, treated special as JVM doesn’t know if being referenced by native code or not

GC roots are referenced by the JVM so keep every connected object from being garbage collected.



A simple java application has the following GC roots:

* Local variables in main method
* Main thread
* Static variables of the main class

**Collecting Garbage**

JVM intermittently runs a mark-and-sweep algorithm

* Traverse all object references starting with GC roots and marking every object which is found alive
* All of heap memory which is not occupied by marked objects is reclaimed

Garbage collection stops unused and unreachable objects from staying in the memory and taking up space.

**Performance**

The garabge collector counts the number of live objects, therefore ts performance is quicker the less objects there are. Since GC pauses the application running, in larger programs it can quickly lead to scalability problems.

**Fragmented Memory**

Repeated memory allocation and reclaimation by JVM leads to memory fragmentation, casuing a couple issues:

* Reduced allocation speed, JVM must find a suitably sized block
* Allocation errors when there are no suitably sized blocks

To solve these issues, JVM performs a memory defragmentation after each GC cycle. This defrag is done by compacting (moving all live objects to the end of a heap).

While compactors use mutliple cpus, the application still has to be paused. When applications have several gigabytes of data, compaction can be several seconds. Therefore to reduce this impact, compaction is split up into serveral smaller increments, and only once a certain level of fragmentation is reached.

Solving Garabge Collection Pause Time - Java

There are three different ways garbage collection can be run:

* Serial - Suspends appication and executes mark and sweep algorithm in single thread
* Parallel - Utilise mutli-core cpus so that garbage collection can be performed in parallel allowing for quicker execution
* Cocurrent - Run the garbage collection cocurrently with the application execution, allowing for most of the work to be done while the application is running, then suspending the applcation for only a small period of time to apply the results

Cocurrent garbage collection, while speeding up the application response time, does complicate the simple mark and sweep alogrithm:

* GC roots are marked as alive - all threads are suspended
* All reachable objects are marked - application still running (application can still create new objects)
* Any new objects created by the applicationa are marked as alive