FREE eBook

LEARNING garbage-collection

#garbage-collection

Table of Contents

About

Chapter 1: Getting started with garbage-collection

Remarks

Garbage Collection (GC) is a way of automatically reclaiming memory that is occupied by objects that are no longer needed by a program. This is in contrast with manual memory management where the programmer explicitly specifies which objects should be deallocated and returned to memory. Good GC-strategies can be more efficient than manual memory management, but it may depend on the type of software.

The main advantages of garbage collection are:

- It frees the programmer from having to do doing manual memory management.
- It avoids certain difficult-to-find bugs that may arise from manual memory management (e.g. dangling pointers, double freeing, certain types of memory leaks).
- Languages that use garbage collection are usually less complex.

The main disadvantages are:

- Garbage collection has some overhead compared to manual memory management.
- It can potentially impact performance, especially when garbage collection is triggered at undesirable moments.
- It is indeterministic, the programmer doesn't know when garbage collection is done and if objects are freed or not.

Most 'newer' programming languages have garbage collection built-in, for example Java, C#, .NET, Ruby and JavaScript. Older languages like C and C++ do not have garbage collection although there are implementations available with garbage collection. There are also languages that allow you to use a combination of garbage collection and manual memory management, for example Modula-3 and Ada.

Garbage collection strategies differ but many use a (variation of) the mark-and-sweep approach. In the mark phase all accessible objects are found and marked. In the sweep phase the heap is scanned for inaccessible and unmarked objects which are then cleaned up. Modern garbage collectors also use a generational approach where two or more object allocation regions (generations) are kept. The youngest generation contains the newest allocated objects and is cleaned more often. Objects that 'survive' for a certain timespan are promoted to an older generation.

Many languages with GC allow programmers to fine-tune it (see for example the Java 8 Virtual Machine Garbage Collection Tuning Guide or the .Net Garbage Collection documentation)

Examples

Introduction

Objects become eligible for garbage collection (GC) if they are no longer reachable by the main entry point(s) in a program. GC is usually not performed explicitly by the user, but to let the GC know an object is no longer needed a developer can:

Dereference / assign null

```
someFunction {
     var a = 1;
     var b = 2;
     a = null; // GC can now free the memory used for variable a
     ...
} // local variable b not dereferenced but will be subject to GC when function ends
```

Use weak references

Most languages with GC allow you to create weak references to an object which do not count as a reference for the GC. If there are only weak references to an object and no strong (normal) references, then the object is eligible for GC.

```
WeakReference wr = new WeakReference(createSomeObject());
```

Note that after this code it is dangerous to use the target of the weak reference without checking if the object still exists. Beginner-programmers sometimes make the mistake of using code like this:

```
if wr.target is not null {
    doSomeAction(wr.target);
}
```

This can cause problems because GC may have been invoked after the null check and before the execution of doSomeAction. It's better to first create a (temporary) strong reference to the object like this:

```
Object strongRef = wr.target;
if strongRef is not null {
    doSomeAction(strongRef);
}
strongRef = null;
```

Enabling verbose gc logging in Java

Normally the jvm's garbage collection (gc) is transparent to the user (developer/engineer).

GC tuning is normally not required unless the user faces a memory leak or has an application that requires large amount of memory - both of which eventually lead to an out-of-memory exception which compels the user to look into the problem.

The first step is typically to increase the memory (either the heap or the perm-gen/meta-space depending on whether its due to load at runtime or the libary base of the application is large or there is a leak in the classloading or thread-handling mechanism). But whenever that is not feasible, the next step is to try to understand what is going wrong.

If one wants just the snapshot at a particular instant in time, then the `jstat` utility that is part of the jdk would suffice.

However for a more detailed understanding, it is helpful to have a log containing the snapshot of the heap before and after each gc event. For that the user has to enable verbose gc logging by using the `-verbose:gc` as part of the jvm startup parameters and including `-XX:+PrintGCDetails` and `-XX:+PrintGCTimeStamp` flags.

For those who would like to pro-actively profile their application, there are also tools such as `jvisualvm` that is also part of the jdk through which they can gain insight into the applications behaviour.

Below is a sample program, the gc configuration and the verbose-gc log output :

```
package com.example.so.docs.gc.logging;

import java.util.Arrays;
import java.util.Random;

public class HelloWorld {

    public static void main(String[] args) {
        sortTest();
    }

    private static void sortTest() {
        System.out.println("HelloWorld");

        int count = 3;
        while(count-- > 0) {
            int size = 1024*1024;
            int[] numbers = new int[size];
            Random random = new Random();
            for(int i=0;i<size;i++) {
                numbers[i] = random.nextInt(size);
            }

            Arrays.sort(numbers);
        }
        System.out.println("Done");

    }


}
```

GC Options :

```
-server -verbose:gc  -XX:+PrintGCDetails -XX:+PrintGCTimeStamps  -Xmx10m  -XX:-
PrintTenuringDistribution  -XX:MaxGCPauseMillis=250 -Xloggc:/path/to/logs/verbose_gc.log
```

Output :

```
Java HotSpot(TM) 64-Bit Server VM (25.72-b15) for windows-amd64 JRE (1.8.0_72-b15), built on
Dec 22 2015 19:16:16 by "java_re" with MS VC++ 10.0 (VS2010)
Memory: 4k page, physical 6084464k(2584100k free), swap 8130628k(3993460k free)
CommandLine flags: -XX:InitialHeapSize=10485760 -XX:MaxGCPauseMillis=250 -
XX:MaxHeapSize=10485760 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:-
PrintTenuringDistribution -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:-
UseLargePagesIndividualAllocation -XX:+UseParallelGC
0.398: [GC (Allocation Failure) [PSYoungGen: 483K->432K(2560K)] 4579K->4536K(9728K), 0.0012569
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.400: [GC (Allocation Failure) [PSYoungGen: 432K->336K(2560K)] 4536K->4440K(9728K), 0.0008121
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.401: [Full GC (Allocation Failure) [PSYoungGen: 336K->0K(2560K)] [ParOldGen: 4104K-
>294K(5632K)] 4440K->294K(8192K), [Metaspace: 2616K->2616K(1056768K)], 0.0056202 secs] [Times:
user=0.00 sys=0.00, real=0.01 secs]
0.555: [GC (Allocation Failure) [PSYoungGen: 41K->0K(2560K)] 4431K->4390K(9728K), 0.0004678
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.555: [GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] 4390K->4390K(9728K), 0.0003490
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.556: [Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] [ParOldGen: 4390K-
>293K(5632K)] 4390K->293K(8192K), [Metaspace: 2619K->2619K(1056768K)], 0.0060187 secs] [Times:
user=0.00 sys=0.00, real=0.01 secs]
Heap
 PSYoungGen      total 2560K, used 82K [0x00000000ffd00000, 0x0000000100000000,
0x0000000100000000)
  eden space 2048K, 4% used [0x00000000ffd00000,0x00000000ffd14938,0x00000000fff00000)
  from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)
  to   space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)
 ParOldGen       total 5632K, used 4389K [0x00000000ff600000, 0x00000000ffb80000,
0x00000000ffd00000)
  object space 5632K, 77% used [0x00000000ff600000,0x00000000ffa49670,0x00000000ffb80000)
 Metaspace       used 2625K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 282K, capacity 386K, committed 512K, reserved 1048576K
```

Below are few useful links on GC:

1. An archived page explaining gc concepts (jdk7)
2. G1 Collector Tutorial
3. Useful VM Options
4. JDK 5 - GC Ergonomics (concepts are still relevant)
5. JDK 6 Tuning (concepts are still relevant)

Read Getting started with garbage-collection online: https://riptutorial.com/garbage-collection/topic/8171/getting-started-with-garbage-collection

Credits

S. No	Chapters	Contributors
1	Getting started with garbage-collection	Community, Ravindra HV, THelper