Theory: Garbage Collector

7 minutes 0 / 4 problems solved

Every time our program creates new objects, we need a chunk of physical memory (RAM) on our computer to store them. Memory is allocated on the heap of the Java Virtual Machine every time we use the `new` keyword. However, once our program has finished using the information, the corresponding memory can be used for something else, such as storing new information.

Garbage Collector (known as **GC**) is a part of JVM that frees up this memory in runtime so that it can be reused for other purposes. Automatized memory management protects Java programmers from errors and memory leaks. For instance, in C or C++, where memory allocation and deallocation is handled manually, the part of the storage stays sometimes reserved, even though a program will never use it again.

§1. Ways for requesting JVM to run GC

Garbage collection is performed automatically while a program is running, JVM takes all the dirty work for itself. In most cases, Java developers do not need to think about how GC works and how to customize it. However, in modern high-load applications, this knowledge can be useful.

There are two ways to request GC to perform the job:

- calling `System.gc()`
- calling `Runtime.getRuntime().gc()`

Developers are not supposed to run the garbage collector, so we can't be sure that any one of these methods will actually invoke GC. For the same reason, we recommend to use them only in a test environment!

For the sake of better understanding, we are going to invoke the first method.

§2. Garbage collection example: a horde of hamsters

Let's look at a case where we create several hamster objects to waste computer memory and then request GC to clear the memory from hamsters. We will go into details and explanations further.

Current topic:

Garbage Collector

Topic depends on:

Stage 3
Objects

Table of contents:

↑ Garbage Collector

§1. Ways for requesting JVM to run GC

§2. Garbage collection example: a horde of hamsters

§3. Conclusion

Feedback & Comments

```
class Hamster {
    private int id;

    public Hamster(int id) {
        this.id = id;
    }
}

public class GCExample {



    private static void printUsedMemory() {


Runtime r = Runtime.getRuntime(); // it allows to get memory information f
or JVM

        long usedMemory = r.totalMemory() - r.freeMemory();

        System.out.println("Used memory (bytes): " + usedMemory);

    }



    public static void main(String[] args) {

        printUsedMemory();



        for (int i = 0; i < 1_000_000; i++) {

            new Hamster(i);

        }



        printUsedMemory();



        System.gc(); // Requesting JVM for running GC



        printUsedMemory();

    }

}
```

If we compile and run the program, our output should look as follows:

```
Used memory (bytes): 1197456
Used memory (bytes): 4489984
Used memory (bytes): 712360
```

The numbers you see will probably be different. Let's delve into the code to understand what's happening.

In our `main` method, we first print how much memory we are using at the very start of the program. In our case, it is **1197456** bytes. Then we create a horde of hamsters to waste our memory. Every time we instantiate objects using `new`, we consume some of the available allocated memory. It increases our used memory up to **4489984** bytes. However, once we come back to the `main` method and call `System.gc()`, we notice that our used memory drops to **712360** bytes. The Java garbage collector notices that the program doesn't use our created hamsters once we exit the method scope. Therefore GC frees the corresponding memory and allows other objects to use it in the program.

If we remove the call to `System.gc()`, we should see the following output:

```
Used memory (bytes): 1197448
Used memory (bytes): 4783616
Used memory (bytes): 4783616
```

We see from the 3rd output line that the memory has not been freed at the time of printing. However, we do not need to worry about this as the Java runtime will automatically trigger `System.gc()` behind the scenes and clean up the used memory after the program has finished executing.

§3. Conclusion

We've considered that garbage collector cleans memory from unused objects. We invoke it manually, but then GC works on its own. It automatically handles the unused memory after the program has finished executing. It is not recommended to invoke GC manually in the code using `System.gc()` or `Runtime.getRuntime().gc()`. Both methods do not even guarantee that JVM will invoke the GC.

Report a typo

69 users liked this theory. **4** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion