LUA - GARBAGE COLLECTION

Lua uses automatic memory management that uses garbage collection based on certain algorithms that is in-built in Lua. As a result of automatic memory management, as a developer −

- No need to worry about allocating memory for objects.
- No need to free them when no longer needed except for setting it to nil.

Lua uses a garbage collector that runs from time to time to collect dead objects when they are no longer accessible from the Lua program.

All objects including tables, userdata, functions, thread, string and so on are subject to automatic memory management. Lua uses incremental mark and sweep collector that uses two numbers to control its garbage collection cycles namely **garbage collector pause** and **garbage collector step multiplier**. These values are in percentage and value of 100 is often equal to 1 internally.

Garbage Collector Pause

Garbage collector pause is used for controlling how long the garbage collector needs to wait, before; it is called again by the Lua's automatic memory management. Values less than 100 would mean that Lua will not wait for the next cycle. Similarly, higher values of this value would result in the garbage collector being slow and less aggressive in nature. A value of 200, means that the collector waits for the total memory in use to double before starting a new cycle. Hence, depending on the nature and speed of application, there may be a requirement to alter this value to get best performance in Lua applications.

Garbage Collector Step Multiplier

This step multiplier controls the relative speed of garbage collector to that of memory allocation in Lua program. Larger step values will lead to garbage collector to be more aggressive and it also increases the step size of each incremental step of garbage collection. Values less than 100 could often lead to avoid the garbage collector not to complete its cycle and its not generally preferred. The default value is 200, which means the garbage collector runs twice as the speed of memory allocation.

Garbage Collector Functions

As developers, we do have some control over the automatic memory management in Lua. For this, we have the following methods.

- **collectgarbage** " ***collect*** " − Runs one complete cycle of garbage collection.
- **collectgarbage** " ***count*** " − Returns the amount of memory currently used by the program in Kilobytes.
- **collectgarbage** " ***restart*** " − If the garbage collector has been stopped, it restarts it.
- **collectgarbage** " ***setpause*** " − Sets the value given as second parameter divided by 100 to the garbage collector pause variable. Its uses are as discussed a little above.
- **collectgarbage** " ***setstepmul*** " − Sets the value given as second parameter divided by 100 to the garbage step multiplier variable. Its uses are as discussed a little above.
- **collectgarbage** " ***step*** " − Runs one step of garbage collection. The larger the second argument is, the larger this step will be. The collectgarbage will return true if the triggered step was the last step of a garbage-collection cycle.
- **collectgarbage** " ***stop*** " − Stops the garbage collector if its running.

A simple example using the garbage collector example is shown below.

```
mytable = {"apple", "orange", "banana"}
```

```
print(collectgarbage("count"))

mytable = nil

print(collectgarbage("count"))

print(collectgarbage("collect"))

print(collectgarbage("count"))
```

When we run the above program, we will get the following output. Please note that this result will vary due to the difference in type of operating system and also the automatic memory management feature of Lua.

```
23.1455078125   149
23.2880859375   295
0
22.37109375     380
```

You can see in the above program, once garbage collection is done, it reduced the memory used. But, it's not mandatory to call this. Even if we don't call them, it will be executed automatically at a later stage by Lua interpreter after the predefined period.

Obviously, we can change the behavior of the garbage collector using these functions if required. These functions provide a bit of additional capability for the developer to handle complex situations. Depending on the type of memory need for the program, you may or may not use this feature. But it is very useful to know the memory usage in the applications and check it during the programming itself to avoid undesired results after deployment.