Understanding Garbage Collection in Go

Garbage collection is a mechanism Go developers use to find memory space that is allocated recently but is no longer needed, hence the need to deallocate them to create a clean slate so that further allocation can be done on the same space or so that memory can be reused.

If this process is done automatically, without any intervention of the programmer, it is called *automatic garbage collection*.

The term garbage essentially means *unused* or objects that are created in the memory and are no longer needed, which can be seen as nothing more than garbage, hence a candidate for wiping out from the memory.

## Why Do Developers Need Garbage Collection

Because memory is still a costly space and must be cleaned periodically to make space for other programs to execute (or for the same program to work efficiently), a cluttered memory with a lot of unused elements can create havoc in the long run.

## Issue with Garbage Collection in C/C++

The C programming language has the **free()** function and C++ has the **delete** operator, which are specifically used for the purpose of clean-up and garbage collection procedures.

The **free()** function of the C library helps to release blocks of memory typically allocated by **malloc**, **calloc**, or **realloc** functions.

The **delete** operator in C++ is often used with a destructor to deallocate objects in memory created with the **new** keyword. The **delete** operator can also be used to free-up pointers or an array as well.

But the point is that this freeing up of memory space in C/C++ requires explicit and conscious invocation.

Any miscalculation or forgetfulness can be a breeding place for potential problems now or later.

This is the reason why C/C++ programmers are very meticulous about memory allocation and are cautious about thier use and reuse of variables.

## Garbage Collection in Go and Golang

*The GC runs concurrently with mutator threads, is type accurate (aka precise), allows multiple. GC thread to run in parallel.*

*It is a concurrent mark and sweep that uses a write barrier. It is non-generational and non-compacting.*

*Allocation is done using size segregated per P allocation areas to minimize fragmentation while eliminating locks in the common case.*

## Memory Statistics in Go and Golang

The Go standard library has a host of functions to peek at memory statistics runtime.

The runtime package offers some key **struct** types that can be used to gather memory info at runtime. One of them is called **MemStats**.

This can be used to get feedback on the statistics of the memory allocator.

Some of the key fields of **MemStats** type and what they refer to are as follows. Note that all of these are declared as 64-bit unsigned **int**:

type MemStats struct {

Alloc uint64

TotalAlloc uint64

Mallocs uint64

Frees uint64

Sys uint64

...

}

* **Alloc**: It represents bytes of allocated heap objects. The bytes increase as more objects are created and decrease as they are deallocated.
* **TotalAlloc**: It keeps track of the total number of bytes allocated in the heap objects; however, the number of bytes does not get adjusted as memory gets deallocated through the garbage collector.
* **Sys**: It represents total bytes of memory obtained from the Operating System.
* **Mallocs and Frees**: The **malloc** represents the total count of heap objects allocated and **Frees** represents the total number of heap objects deallocated. Therefore, the count of live objects is always **Mallocs – Frees**.
* **HeapAlloc:** which represent bytes of allocated heap objects
* **HeapSys:** bytes of heap memory obtained from OS
* **HeapIdle:** bytes of unused heap spans
* **HeapInuse:** bytes of used heap span
* **StackAlloc:** bytes of allocated stack memory
* **StackSys:** bytes of stack memory obtained from OS
* **StackIdle:** bytes of unused stack memory
* **StackInuse**: bytes of used stack memory

## Example of Garbage Collection in Go and Golang

Let us write some simple Go code to get the memory statistics of a running program

The point here is to illustrate how to extract memory information.

Getting a memory snapshot after a certain interval – and then comparing and investigating the result – will reveal how garbage collection works behind the scenes.

package main

import (

"fmt"

"math/rand"

"runtime"

"time"

)

func main() {

var ms runtime.MemStats

printMemStat(ms)

//----------------------------------

// you can write any code here

//----------------------------------

intArr := make([]int, 900000)

for i := 0; i < len(intArr); i++ {

intArr[i] = rand.Int()

}

//------------------------------------

time.Sleep(5 \* time.Second)

printMemStat(ms)

}

func printMemStat(ms runtime.MemStats) {

runtime.ReadMemStats(&ms)

fmt.Println("--------------------------------------")

fmt.Println("Memory Statistics Reporting time: ", time.Now())

fmt.Println("--------------------------------------")

fmt.Println("Bytes of allocated heap objects: ", ms.Alloc)

fmt.Println("Total bytes of Heap object: ", ms.TotalAlloc)

fmt.Println("Bytes of memory obtained from OS: ", ms.Sys)

fmt.Println("Count of heap objects: ", ms.Mallocs)

fmt.Println("Count of heap objects freed: ", ms.Frees)

fmt.Println("Count of live heap objects", ms.Mallocs-ms.Frees)

fmt.Println("Number of completed GC cycles: ", ms.NumGC)

fmt.Println("--------------------------------------")

}

The expected output of running this code in your integrated development environment (IDE) or code editor would be:

-------------------------------------------------------

Memory Statistics Reporting time: 2022-04-14 17:43:11.048224903 +0530 IST m=+0.000264317

-------------------------------------------------------

Bytes of allocated heap objects: 89432

Total bytes of Heap object: 89432

Bytes of memory obtained from OS: 8211472

Count of heap objects: 180

Count of heap objects freed: 3

Count of live heap objects 177

NumGC is the number of completed GC cycles: 0

-------------------------------------------------------

-------------------------------------------------------

Memory Statistics Reporting time: 2022-04-14 17:43:16.072656121 +0530 IST m=+5.024695581

-------------------------------------------------------

Bytes of allocated heap objects: 7285832

Total bytes of Heap object: 7301992

Bytes of memory obtained from OS: 17189648

Count of heap objects: 227

Count of heap objects freed: 47

Count of live heap objects 180

NumGC is the number of completed GC cycles: 1

-------------------------------------------------------

There is a way to get even more detailed info about the Go garbage collector using the following command while running the program above:

GODEBUG=gctrace=1 go run memprog1.go

The output will be something like this:

gc 9 @0.126s 2%: 0.10+0.73+0.012 ms clock, 0.40+0.48/0.68/0.060+0.048 ms cpu, 4->4->0 MB, 5 MB goal, 4 P

Here, in the numbers **4->4->0**, the first number represents heap size prior to garbage collection, the second number represents heap size after garbage collection, and the last number is the live heap object count.