



Heap Optimization for Go Systems

Nishant Roy Tech Lead @ Pinterest Dec 2020



About Me



Nishant Roy

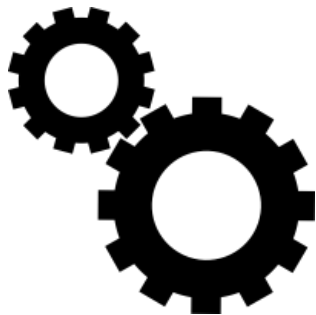
**Ads Serving Architecture
@ Pinterest**

Georgia Tech '17

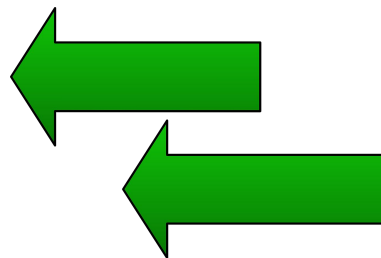
Brunch & Basketball 

How does memory management in Go work?

Automated



Concurrent



How does GC impact performance?

**Attempts to limit
CPU usage**



**GC steals
resources**

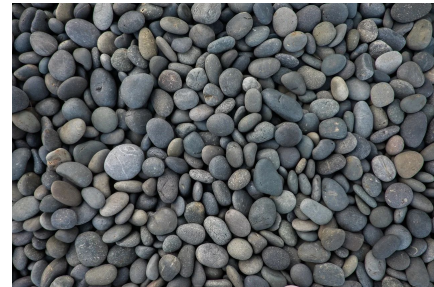


What causes GC pressure?

Scanning the heap



Number of heap objects



How to determine if GC is the problem?

- High tail latency
- Set the GODEBUG environment variable to see GC stats
 - GODEBUG=gctrace=1
- Compare GC start/end times to system metrics like latency

gctrace

```
gc 2553 @8.452s 14%: 0.004+0.33+0.051 ms clock, 0.056+0.12/0.56/0.94+0.61 ms cpu, 4->4->2 MB, 5 MB goal, 12 P

gc 2553      : The 2553 GC runs since the program started
@8.452s     : Eight seconds since the program started
14%        : Fourteen percent of the available CPU so far has been spent in GC

// wall-clock
0.004ms     : STW           : Write-Barrier - Wait for all Ps to reach a GC safe-point.
0.33ms      : Concurrent   : Marking
0.051ms     : STW           : Mark Term      - Write Barrier off and clean up.

// CPU time
0.056ms     : STW           : Write-Barrier
0.12ms      : Concurrent   : Mark - Assist Time (GC performed in line with allocation)
0.56ms      : Concurrent   : Mark - Background GC time
0.94ms      : Concurrent   : Mark - Idle GC time
0.61ms      : STW           : Mark Term

4MB         : Heap memory in-use before the Marking started
4MB         : Heap memory in-use after the Marking finished
2MB         : Heap memory marked as live after the Marking finished
5MB         : Collection goal for heap memory in-use after Marking finished

// Threads
12P         : Number of logical processors or threads used to run Goroutines.
```

Source:
[ArdenLabs](#)

How to profile heap usage?

- Built-in tools to study heap usage

runtime.MemStats - Statistics about the memory allocator

pprof - System profile visualizer

MemStats

```
// HeapObjects is the number of allocated heap objects.
```

```
HeapObjects uint64
```

```
// HeapAlloc is bytes of allocated heap objects.
```

```
HeapAlloc uint64
```

```
// Sys is the total bytes of memory obtained from the OS.
```

```
HeapSys uint64
```

Source:

[Go docs](#)

MemStats

```
func main() {
    PrintMemstats()

    var arr [][]int

    for i := 0; i<4; i++ {
        vec := make([]int, 0, 25000)
        overall = append(arr, vec)
        PrintMemUsage()
    }

    overall = nil
    PrintMemstats()

    runtime.GC()
    PrintMemstats()
}
```

```
func PrintMemstats() {
    var m runtime.MemStats
    runtime.ReadMemStats(&m)
    fmt.Printf("HeapAlloc = %v", (m.HeapAlloc))
    fmt.Printf("\tHeapObjects = %v", (m.HeapObjects))
    fmt.Printf("\tHeapSys = %v", (m.Sys))
    fmt.Printf("\tNumGC = %v\n", m.NumGC)
}
```

Source:

[Golang Code](#)

MemStats

```
$ go run main.go
```

HeapAlloc = 106392	HeapObjects = 133	HeapSys = 69928960	NumGC = 0
HeapAlloc = 312528	HeapObjects = 142	HeapSys = 69928960	NumGC = 0
HeapAlloc = 517928	HeapObjects = 150	HeapSys = 69928960	NumGC = 0
HeapAlloc = 723112	HeapObjects = 158	HeapSys = 71631096	NumGC = 0
HeapAlloc = 928400	HeapObjects = 164	HeapSys = 71631096	NumGC = 0
HeapAlloc = 928736	HeapObjects = 170	HeapSys = 71631096	NumGC = 0
HeapAlloc = 112032	HeapObjects = 153	HeapSys = 71958776	NumGC = 1

pprof

```
go tool pprof [options] http://localhost:6060/debug/pprof/heap
```

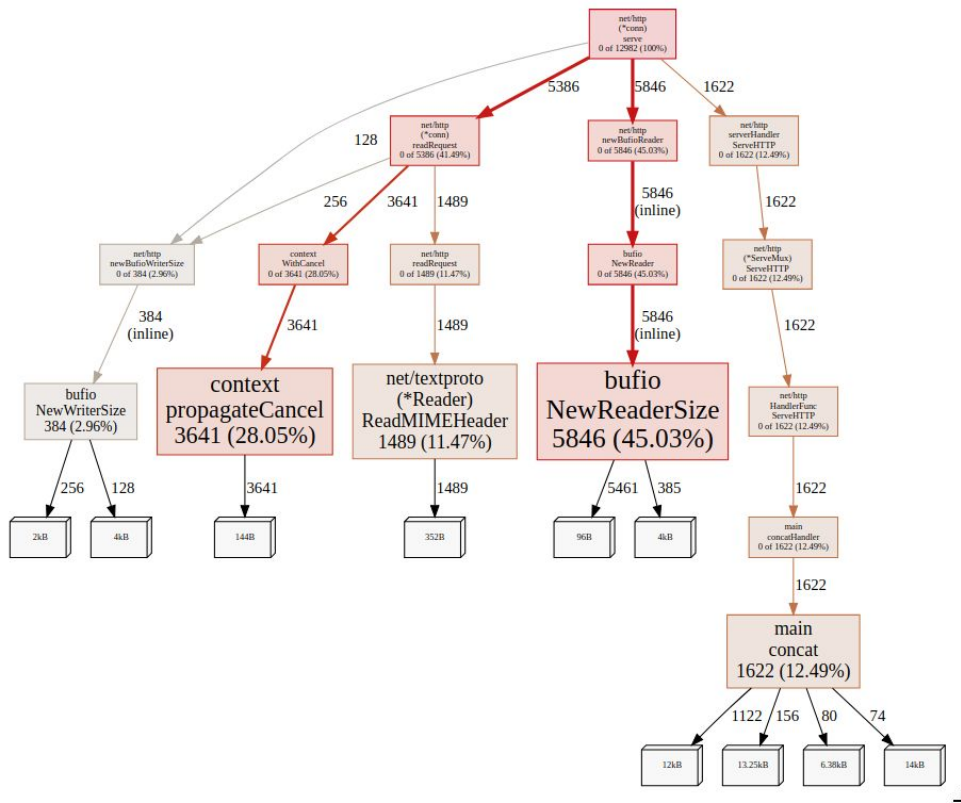
```
// Available options
```

```
-inuse_space      Display in-use memory size  
-inuse_objects    Display in-use object counts  
-alloc_space      Display allocated memory size  
-alloc_objects    Display allocated object counts
```

```
go tool pprof -http=localhost:<port> /path/to/profile.pb.gz
```

Source:
[Go docs](#)

pprof



Source:
matoski.com

pprof

(pprof) list createCatalogMap

Total: 132263423

ROUTINE ===== <CODE_PATH>

105268459 105268459 (flat, cum) 79.59% of Total

. 63815675 233: product := BuildProduct(productID, productPrice, productSellerID)

. . 234: if productPrice < minProductPrice {

. . 235: minProductPrice = productPrice

. . 236: }

. 20726392 237: catalogListing := catalogs.CreateListing(product, contextFeatures)

. . 238:

. . 239: // Create listing key by encoding productID, sellerID, and catalog version

. . 240: catalogListingKey := catalogs.CreateListingKey(productID, sellerID, catalogListing.GetVersion())

20726392 20726392 241: catalogMap[catalogListingKey] = catalogListing

. . 242: return catalogMap

How can we limit the impact of GC?

- Lower the number of objects on heap
- Reduce the rate of object allocation
- Optimize data structures for minimal memory usage

Reduce long-living heap objects

- Create objects on-demand
- Be mindful of pointers
- Strings and byte arrays are also pointers!

Impact of removing strings

(pprof) list createCatalogMap

Total: 132263423

ROUTINE ===== <CODE_PATH>

105268459 105268459 (flat, cum) 79.59% of Total

. 63815675 233: product := BuildProduct(productID, productPrice, productSellerID)

. . 234: if productPrice < minProductPrice {

. . 235: minProductPrice = productPrice

. . 236: }

. 20726392 237: catalogListing := catalogs.CreateListing(product, contextFeatures)

. . 238:

. . 239: // Create listing key by encoding productID, sellerID, and catalog version

. . 240: catalogListingKey := catalogs.CreateListingKey(productID, sellerID, catalogListing.GetVersion())

20726392 20726392 241: catalogMap[catalogListingKey] = catalogListing

. . 242: return catalogMap

Impact of removing strings

(pprof) list createCatalogMap

Total: 106261986

```
ROUTINE ===== <CODE_PATH>
 34768  84576835 (flat, cum) 79.59% of Total
    .   63815675   233: product := BuildProduct(productID, productPrice, productSellerID)
    .           .   234: if productPrice < minProductPrice {
    .           .   235:   minProductPrice = productPrice
    .           .   236: }
    .   20726392   237: catalogListing := catalogs.CreateListing(product, contextFeatures)
    .           .   238:
    .           .   239: structKey := CatalogKeyStruct{
    .           .   240:   ProductID:    productID,
    .           .   241:   SellerID:    productSellerID,
    .           .   242:   CatalogVersion: catalogListing.GetVersion(),
    .           .   243: }
 34768  34768     244: catalogMap[structKey] = catalogListing
    .           .   245: return catalogMap
```

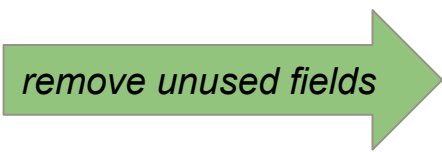
Reduce the rate of allocation

- **Object pooling**
 - *Warning: Can cause memory leaks if not used properly*

Clean up unused data fields

64 bytes

```
type BadObject struct {  
    A    bool  
    B    int64  
    C    int32  
    D    bool  
    E    int32  
    F    bool  
    G    int32  
    H    bool  
    I    int64    // unused  
    J    bool     // unused  
    K    int32    // unused  
    L    int64    // unused  
}
```



remove unused fields

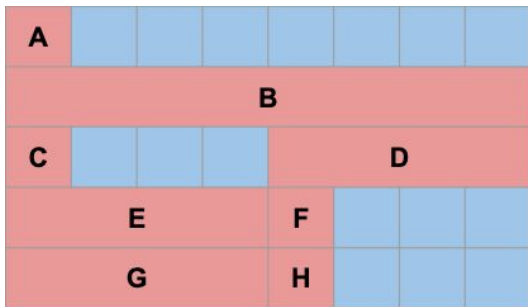
40 bytes

```
type GoodObject struct {  
    A    bool  
    B    int64  
    C    int32  
    D    bool  
    E    int32  
    F    bool  
    G    int32  
    H    bool  
}
```

Reorder fields for proper data alignment

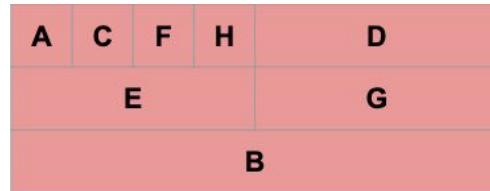
```
type BadObject struct {  
    A    bool  
    B    int64  
    C    int32  
    D    bool  
    E    int32  
    F    bool  
    G    int32  
    H    bool  
}
```

40 bytes



```
type GoodObject struct {  
    A    bool  
    C    bool  
    F    bool  
    H    bool  
    D    int32  
    E    int32  
    G    int32  
    B    int64  
}
```

24 bytes



reordering fields

Conclusion

- **Go GC is great, but not always perfect**
- **Go has tools to find the problem**
- **Optimizing for GC can significantly improve performance for heavy use cases!**



Thank You!

Nishant Roy Tech Lead @ Pinterest Dec 2020

