Profiling Go Applications in the Right Way with Examples

Abdulsamet İLERİ · Follow
Published in Stackademic
9 min read · Sep 13

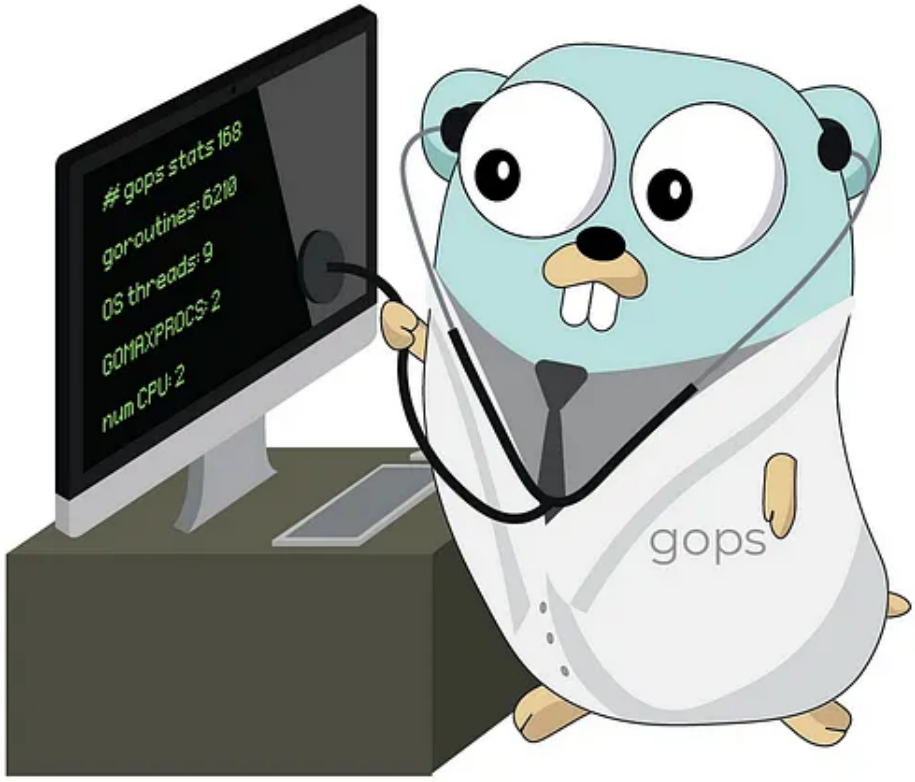


Illustration created for “A Journey With Go,” made from the original Go Gopher, created by Renee French.

What is profiling?

Profiling is an essential technique for analyzing applications and identifying bottlenecks hindering the application’s performance. It is helpful to detect which parts of the code are taking **too long to execute** or **consuming too many resources,** such as CPU and memory.

Table of Contents

- Benchmarking
  - Profiling CPU
  - Profiling Memory
- Runtime profiling
  - Profiling CPU
  - Profiling Memory
- Web profiling
  - Getting CPU Profile and tricks
  - Getting Heap Profile and tricks
  - Getting Allocation Profile and tricks

Methods of Profiling

You have **three** methods for profiling.

- **Go test** (with benchmarking).
- **Runtime profiling** with runtime/pprof.
- **Web profiling** with net/http/pprof.

Types of Profiling

- **CPU** *(collects data on the application CPU usage)*
- **Heap / Memory** *(collects data on application memory usage)*
- **Goroutine** (*identifying the functions that create the most number of goroutines*)
- **Block** (*identifying the functions that cause the most blocking*)
- **Thread** (*identifying the functions that create the most threads*)
- **Mutex** (*identifying the functions that have the most contention*)

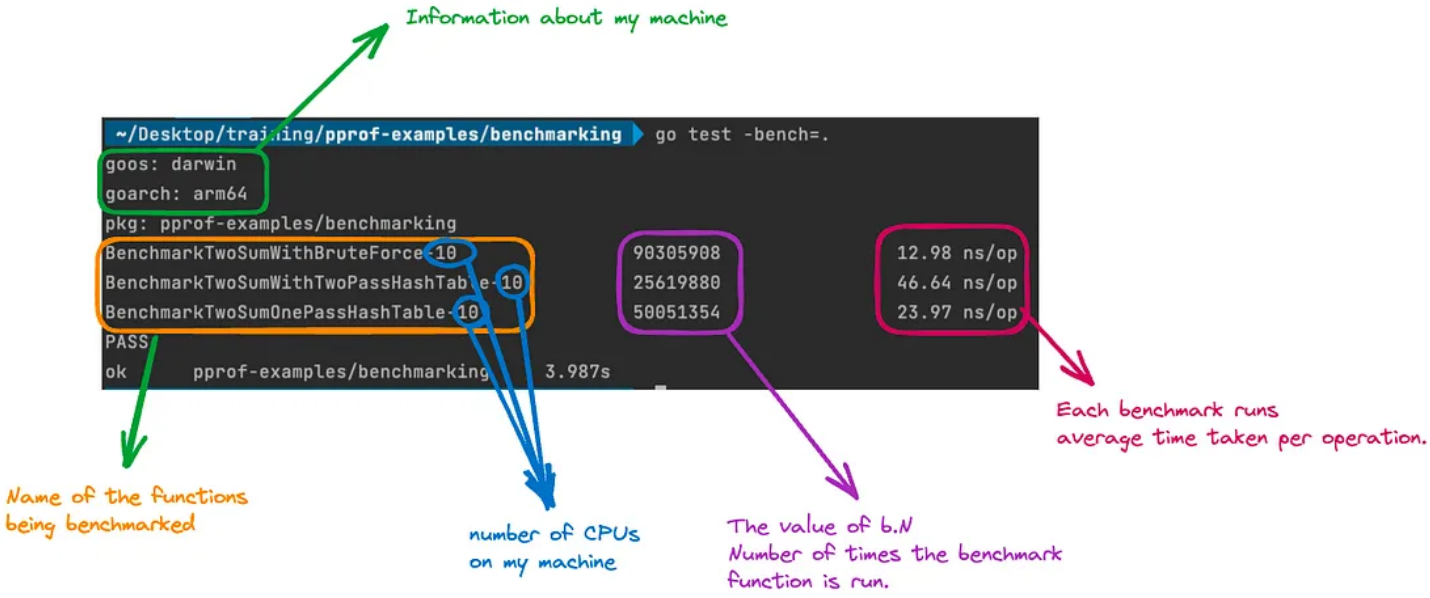
In this article, I mainly focus on CPU and Memory profiling with the abovementioned methods.

1. Benchmarking

I wanted to implement the famous Two Sum algorithm problem. I don't focus on the details about it. Let's run

```
go test -bench=.
```

-bench option runs all of the benchmark tests in our project.



Go test bench output.

Based on the output above, `TwoSumWithBruteForce` is the most efficient one compared to others. Don't forget that it depends on the function's inputs. If you give a large array, you will get different results. 😅

If you type `go help testflag` , you will see lots of flags and their explanations. Like `count, benchtime` etc. I will explain the most used ones.

- If you want to run a specific function, you can specify it as

```
go test -bench='BenchmarkTwoSumWithBruteForce'
```

- By default, benchmark functions run only once; if you want to override this behavior, you can use `count` a flag. For example,

```
go test -bench='.' -count=2
```

gives output as shown below.

```
~/Desktop/training/pprof-examples/benchmarking ▶ go test -bench='.' -count=2
goos: darwin
goarch: arm64
pkg: pprof-examples/benchmarking
BenchmarkTwoSumWithBruteForce-10                88698626                13.26 ns/op
BenchmarkTwoSumWithBruteForce-10                89610752                13.43 ns/op
BenchmarkTwoSumWithTwoPassHashTable-10          25649223                46.69 ns/op
BenchmarkTwoSumWithTwoPassHashTable-10          25512508                46.68 ns/op
BenchmarkTwoSumOnePassHashTable-10              48109047                23.87 ns/op
BenchmarkTwoSumOnePassHashTable-10              49866072                23.96 ns/op
PASS
ok      pprof-examples/benchmarking     7.576s
```

Benchmarking with count flag

- By default, Go decides how long each benchmarking operation runs. You can override this by providing. `benchtime='2s' .`

> *You can use `count` and `benchtime` flags together for better measurement of your benchmarked functions. You can also refer to this article.*
>
> *You can find source code here!*

In the real world, our functions can be complex and long. The timing is not helpful here, so we need to extract the CPU and memory profile for further analysis. You can type

```
go test -bench='.' -cpuprofile='cpu.prof' -memprofile='mem.prof'
```

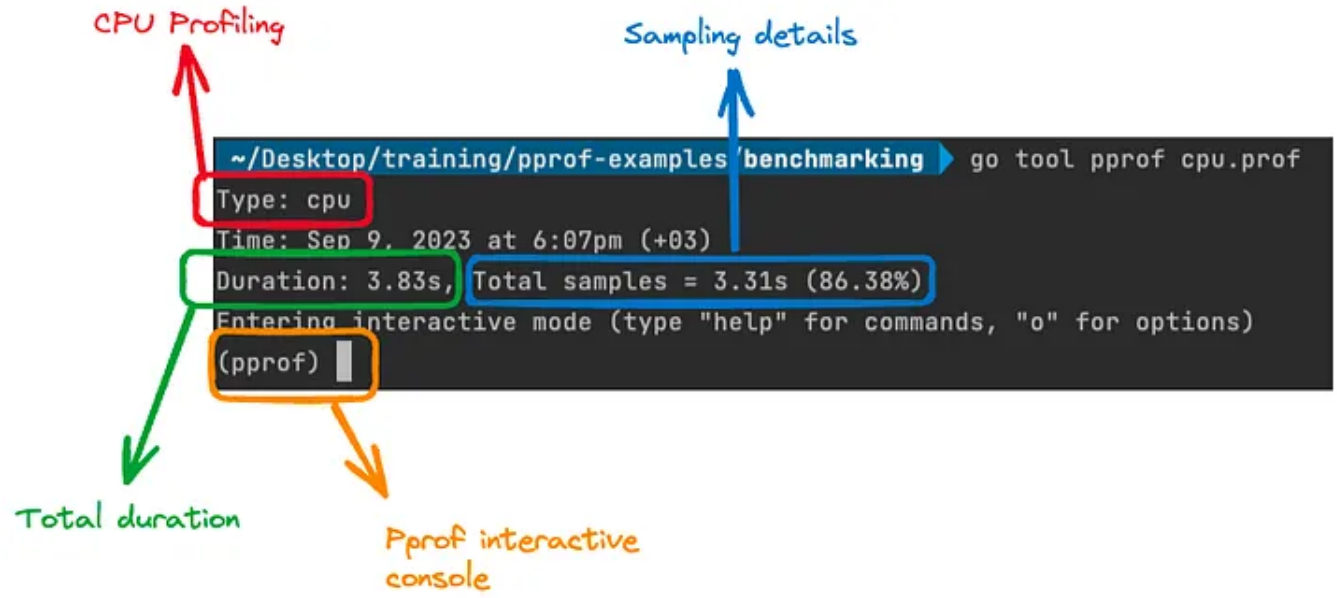
Let's analyze them using the pprof tool.

1.1 Profiling CPU

If you type in

```
go tool pprof cpu.prof
```

and hit enter, you will have a pprof interactive console.



go tool pprof cpu.prof output

Let's take a look at the most popular ones.

- You can type `top15` to see the top 15 resource-intensive functions during the execution time. *(15 indicates the number of nodes that will displayed.)*

To be clear, let's assume that we have a `A` function.

```
func A() {
   B()              // takes 1s
   DO STH DIRECTLY  // takes 4s
   C()              // takes 6s
}
```

Flat and cumulative values are calculated as; **A=4 for flat** and **A=11 for cum** *(1s + 4s + 6s)*.

- If you want to sort based on cum value, you can type `top15 -cum` . You can also use `sort=cum` and `top15` commands respectively.
- If you want to get more verbose output in the `top` , you can specify `granularity` option. For example, if we pass `granularity=lines` , it shows the lines of the functions.

```
(pprof) granularity=lines
(pprof) top15
Showing nodes accounting for 2180ms, 65.86% of 3310ms total
Dropped 66 nodes (cum <= 16.55ms)
Showing top 15 nodes out of 115
      flat  flat%   sum%        cum   cum%
    1280ms 38.67% 38.67%     1280ms 38.67%  runtime.kevent /Users/samet.ileri/go-121/go1.21.1/src/runtime/sys_darwin.go:458
     170ms  5.14% 43.81%      170ms  5.14%  runtime.usleep /Users/samet.ileri/go-121/go1.21.1/src/runtime/sys_darwin.go:382
     110ms  3.32% 47.13%      110ms  3.32%  runtime.madvise /Users/samet.ileri/go-121/go1.21.1/src/runtime/sys_darwin.go:253
      80ms  2.42% 49.55%       80ms  2.42%  runtime.pthread_cond_wait /Users/samet.ileri/go-121/go1.21.1/src/runtime/sys_darwin.go:507
      70ms  2.11% 51.66%       70ms  2.11%  runtime.pthread_kill /Users/samet.ileri/go-121/go1.21.1/src/runtime/sys_darwin.go:177
      60ms  1.81% 53.47%      400ms 12.08%  benchmarking.BenchmarkTwoSumWithBruteForce /Users/samet.ileri/Desktop/training/pprof-examples/benchmarking/main_test.go:7
      60ms  1.81% 55.29%       60ms  1.81%  runtime.mapassign_fast64 /Users/samet.ileri/go-121/go1.21.1/src/runtime/map_fast64.go:126
      50ms  1.51% 56.80%       50ms  1.51%  runtime.mapassign_fast64 /Users/samet.ileri/go-121/go1.21.1/src/runtime/map_fast64.go:128
      50ms  1.51% 58.31%       50ms  1.51%  runtime.nextFreeFast /Users/samet.ileri/go-121/go1.21.1/src/runtime/malloc.go:889 (inline)
      50ms  1.51% 59.82%       50ms  1.51%  runtime.pthread_cond_timedwait_relative_np /Users/samet.ileri/go-121/go1.21.1/src/runtime/sys_darwin.go:517
      40ms  1.21% 61.03%       40ms  1.21%  runtime.mallocgc /Users/samet.ileri/go-121/go1.21.1/src/runtime/malloc.go:1027
      40ms  1.21% 62.24%       40ms  1.21%  runtime.mallocgc /Users/samet.ileri/go-121/go1.21.1/src/runtime/malloc.go:1180
      40ms  1.21% 63.44%       40ms  1.21%  runtime.mapassign_fast64 /Users/samet.ileri/go-121/go1.21.1/src/runtime/map_fast64.go:109
      40ms  1.21% 64.65%       40ms  1.21%  runtime.mapassign_fast64 /Users/samet.ileri/go-121/go1.21.1/src/runtime/map_fast64.go:176
      40ms  1.21% 65.86%       40ms  1.21%  runtime.memhash64 /Users/samet.ileri/go-121/go1.21.1/src/runtime/asm_arm64.s:530
(pprof)
```

top15 output with granularity lines

Thanks to this, we can identify specific lines of the function causing performance issues. 😌

- The output also shows us runtime and user-defined functions together. If you want to focus on only your functions, you can provide `hide=runtime` and execute `top15` again.

```
(pprof) hide=runtime
(pprof) top15
Active filters:
   hide=runtime
Showing nodes accounting for 1.49s, 45.02% of 3.31s total
      flat  flat%   sum%        cum   cum%
     0.62s 18.73% 18.73%      0.62s 18.73%  benchmarking.TwoSumWithTwoPassHashTable
     0.45s 13.60% 32.33%      0.45s 13.60%  benchmarking.TwoSumOnePassHashTable
     0.34s 10.27% 42.60%      0.34s 10.27%  benchmarking.TwoSumWithBruteForce
     0.06s  1.81% 44.41%      0.40s 12.08%  benchmarking.BenchmarkTwoSumWithBruteForce
     0.02s   0.6% 45.02%      0.47s 14.20%  benchmarking.BenchmarkTwoSumOnePassHashTable
         0     0% 45.02%      0.62s 18.73%  benchmarking.BenchmarkTwoSumWithTwoPassHashTable
         0     0% 45.02%      1.49s 45.02%  testing.(*B).launch
         0     0% 45.02%      1.49s 45.02%  testing.(*B).runN
```

top15 with hide option

You can reset it by typing `hide=` .

- Conversely, you can use `show` command. For example, by typing `show=TwoSum`

```
(pprof) show=TwoSum
(pprof) top
Active filters:
   show=TwoSum
Showing nodes accounting for 1.49s, 45.02% of 3.31s total
      flat  flat%   sum%        cum   cum%
     0.62s 18.73% 18.73%      0.62s 18.73%  benchmarking.TwoSumWithTwoPassHashTable
     0.45s 13.60% 32.33%      0.45s 13.60%  benchmarking.TwoSumOnePassHashTable
     0.34s 10.27% 42.60%      0.34s 10.27%  benchmarking.TwoSumWithBruteForce
     0.06s  1.81% 44.41%      0.40s 12.08%  benchmarking.BenchmarkTwoSumWithBruteForce
     0.02s   0.6% 45.02%      0.47s 14.20%  benchmarking.BenchmarkTwoSumOnePassHashTable
         0     0% 45.02%      0.62s 18.73%  benchmarking.BenchmarkTwoSumWithTwoPassHashTable
```

- If you want to focus on a specific node, you can use `focus` command. Let's focus on `TwoSumOnePassHashTable` . The output is displayed as

```
Active filters:
   focus=TwoSumOnePassHashTable
   hide=runtime
Showing nodes accounting for 0.47s, 14.20% of 3.31s total
      flat  flat%   sum%        cum   cum%
     0.45s 13.60% 13.60%      0.45s 13.60%  benchmarking.TwoSumOnePassHashTable
     0.02s   0.6% 14.20%      0.47s 14.20%  benchmarking.BenchmarkTwoSumOnePassHashTable
         0     0% 14.20%      0.47s 14.20%  testing.(*B).launch
         0     0% 14.20%      0.47s 14.20%  testing.(*B).runN
```

top output with hide and focus together

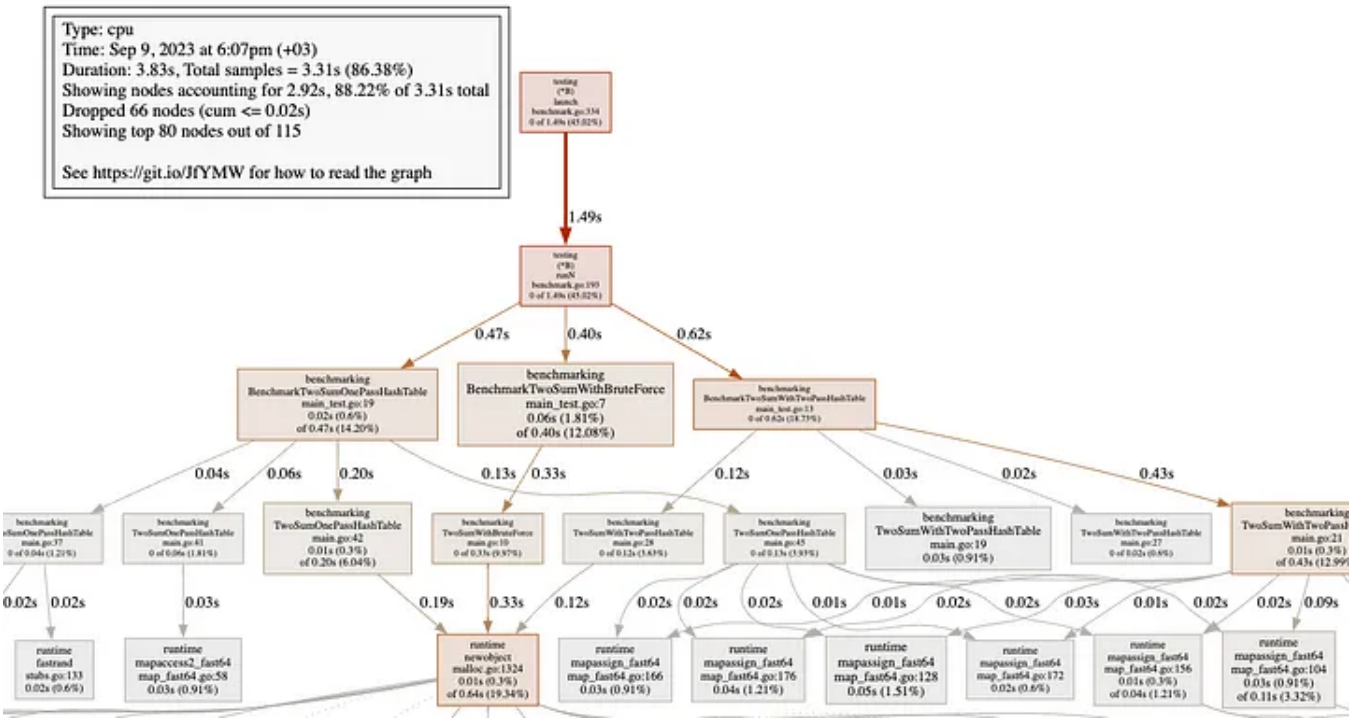
You can reset it by typing `focus=` .

- If you want to get details of the function, you can use `list` command. For example, I want to get details about `TwoSumWithTwoPassHashTable` function. Type `list TwoSumWithTwoPassHashTable`

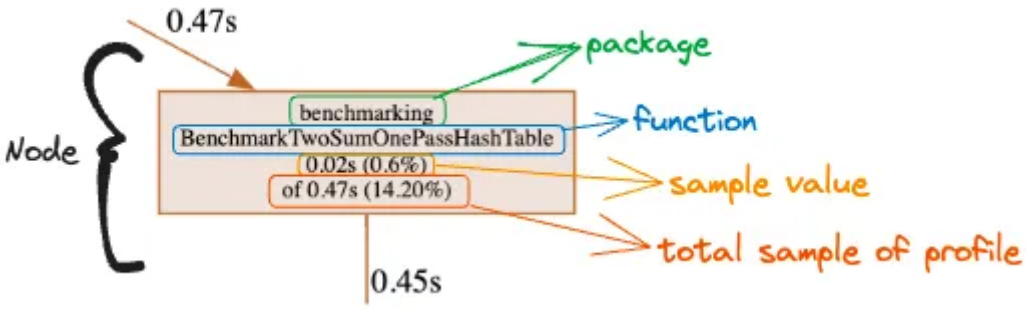
```
     60ms      620ms (flat, cum) 18.73% of Total
     10ms       10ms     18:func TwoSumWithTwoPassHashTable(nums []int, target int) []int {
     30ms       30ms     19:   indexByNums := make(map[int]int)
     10ms       10ms     20:   for i := 0; i < len(nums); i++ {
     10ms      430ms     21:           indexByNums[nums[i]] = i
        .          .     22:   }
        .          .     23:
        .          .     24:   for i := 0; i < len(nums); i++ {
        .          .     25:           complement := target - nums[i]
        .          .     26:
        .       20ms     27:           if val, ok := indexByNums[complement]; ok && val != i {
        .      120ms     28:                   return []int{i, val}
        .          .     29:           }
        .          .     30:   }
        .          .     31:
        .          .     32:   return nil
        .          .     33:}
```

`list TwoSumWithTwoPassHashTable` output

- If you want to see a graphical representation of the call stack, you can type `web` .



web output



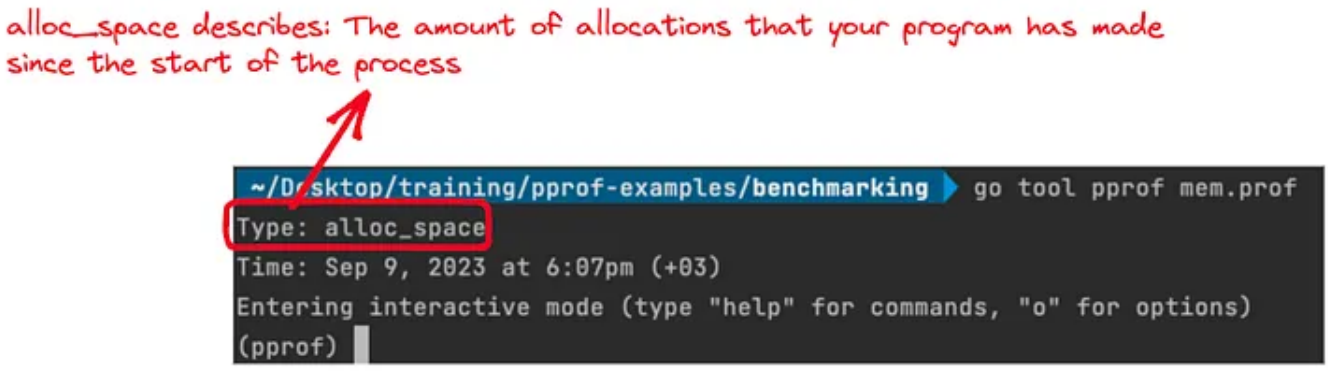
Node detail

I will give more details about reading the graphs in the following sections.

- You can also type `gif` or `pdf` to share profiling data in the corresponding format with someone. 😃

1.2 Profiling Memory

If you type in `go tool pprof mem.prof` and hit enter



go tool pprof mem.prof output

```
(pprof) top10
Showing nodes accounting for 2451.04MB, 99.90% of 2453.61MB total
Dropped 13 nodes (cum <= 12.27MB)
      flat  flat%   sum%        cum   cum%
 1327.52MB 54.10% 54.10%  1327.52MB 54.10%  benchmarking.TwoSumWithBruteForce
  741.01MB 30.20% 84.31%   741.01MB 30.20%  benchmarking.TwoSumOnePassHashTable
  382.51MB 15.59% 99.90%   382.51MB 15.59%  benchmarking.TwoSumWithTwoPassHashTable
         0     0% 99.90%   741.01MB 30.20%  benchmarking.BenchmarkTwoSumOnePassHashTable
         0     0% 99.90%  1327.52MB 54.10%  benchmarking.BenchmarkTwoSumWithBruteForce
         0     0% 99.90%   382.51MB 15.59%  benchmarking.BenchmarkTwoSumWithTwoPassHashTable
         0     0% 99.90%  2451.04MB 99.90%  testing.(*B).launch
         0     0% 99.90%  2451.04MB 99.90%  testing.(*B).runN
```

top10 output

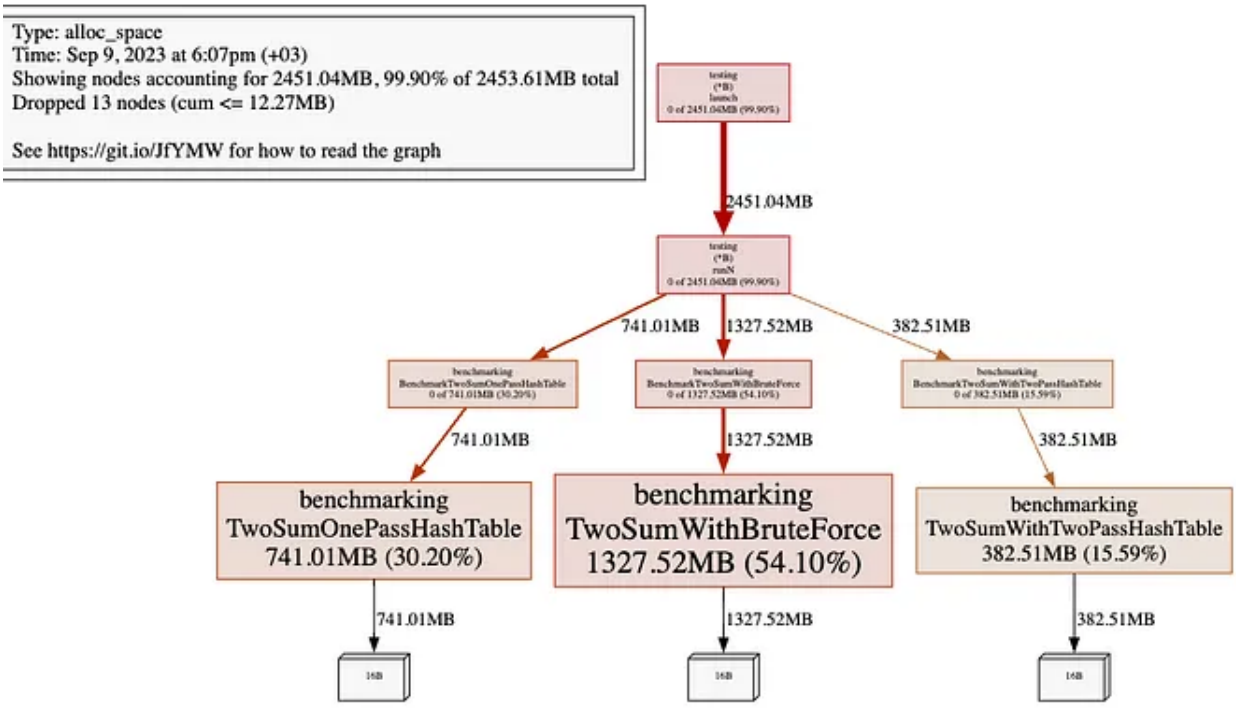
> *Note that flat and cum are the same thing mentioned above but only different measurements* (ms in CPU, MB in Memory profiling).

- list command:

```
1.30GB     1.30GB (flat, cum) 54.10% of Total
      .          .      6:func TwoSumWithBruteForce(nums []int, target int) []int {
      .          .      7:    for i := 0; i < len(nums); i++ {
      .          .      8:            for j := i + 1; j < len(nums); j++ {
      .          .      9:                    if nums[j] == target-nums[i] {
1.30GB     1.30GB     10:                            return []int{i, j}
      .          .     11:                    }
      .          .     12:            }
      .          .     13:    }
      .          .     14:    return nil
      .          .     15:}
```

list TwoSumWithBruteForce output

- web command:



web command output

We can use all the commands mentioned in the CPU profiling section.

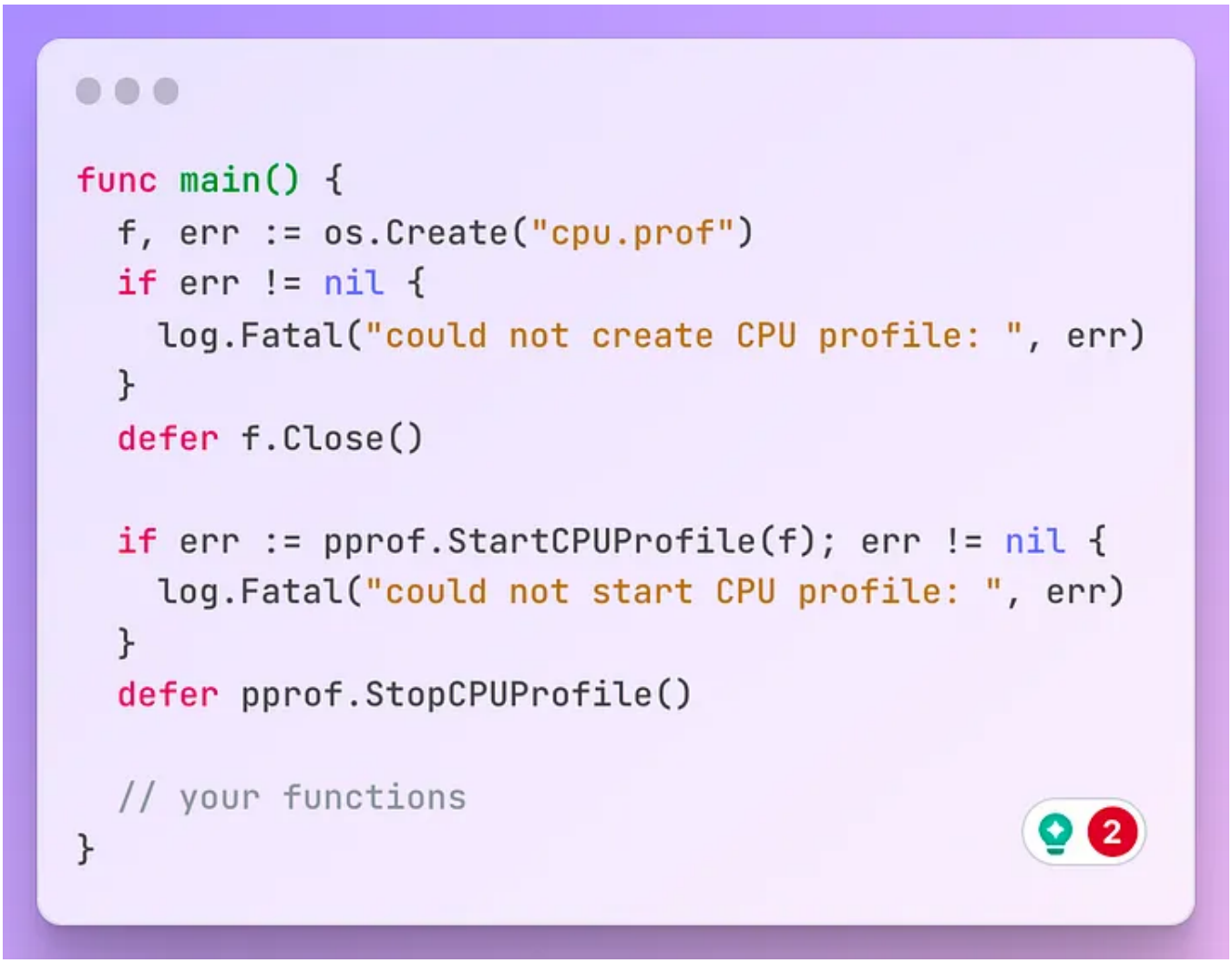
Let's take a look at another method, runtime/pprof. 🚀

2- Runtime profiling with runtime/pprof.

Benchmark tests are useful for an isolated function's performance but are insufficient to understand the whole picture. Here runtime/pprof shines 💠.

2.1 Profiling CPU

In benchmark tests, CPU and memory profiling are supported built-in. If you need to add CPU profiling support to your application, you must enable it first.



```go
func main() {
  f, err := os.Create("cpu.prof")
  if err != nil {
    log.Fatal("could not create CPU profile: ", err)
  }
  defer f.Close()

  if err := pprof.StartCPUProfile(f); err != nil {
    log.Fatal("could not start CPU profile: ", err)
  }
  defer pprof.StopCPUProfile()

  // your functions
}
```

If you run `go run .` here, you will see the generated `cpu.prof` file. You can analyze it `go tool pprof cpu.prof` as mentioned in the benchmarking section.

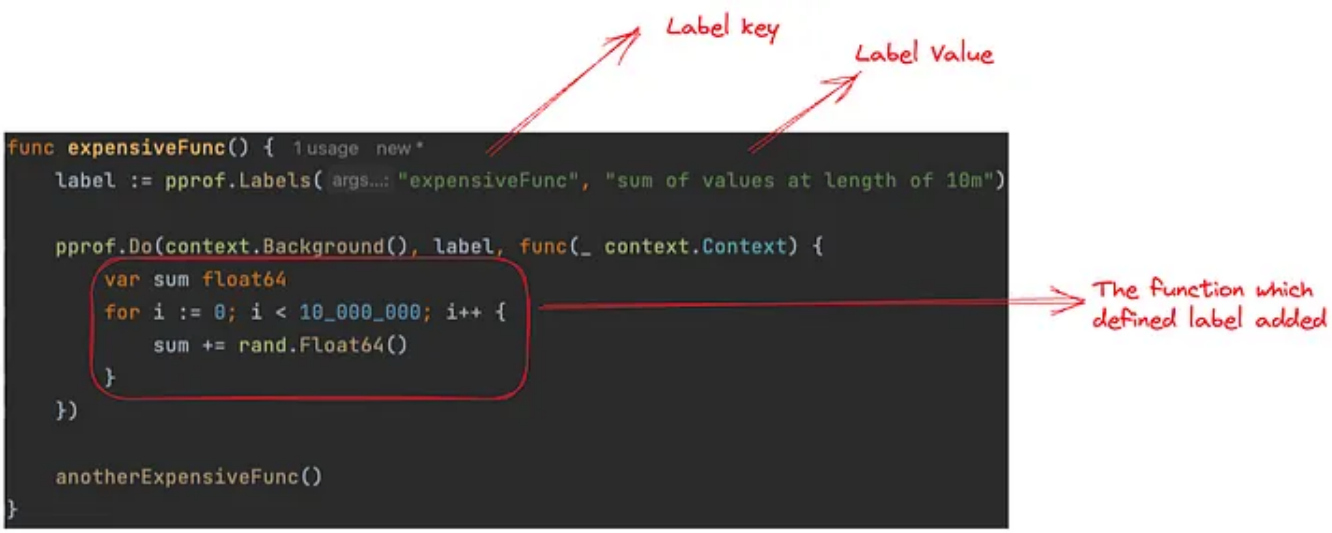
In this section, I want to introduce to you one of my favorite features is `pprof.Labels`. T*his feature is available only on CPU and goroutine profiling*.

If you want to add a label/labels to a specific function, you can use `pprof.Do` func.

```go
pprof.Do(ctx, pprof.Labels("label-key", "label-value"), func(ctx context.Contex
    // execute labeled code
```

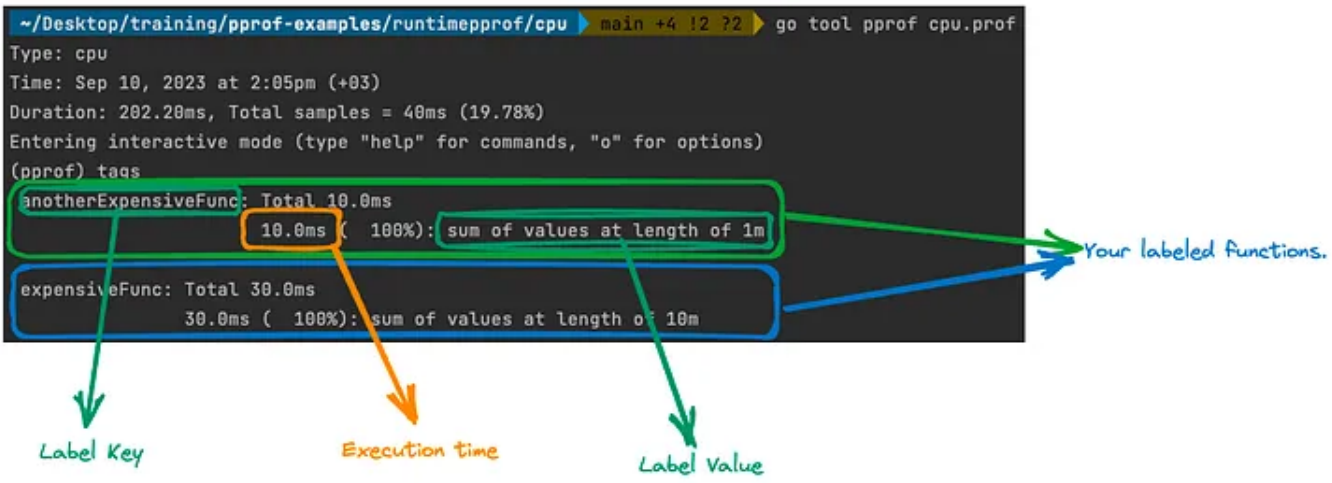
```
    })
```

For example,



Adding a label to a specific function

In the pprof interactive console, type `tags` . It shows your labeled functions with useful information.



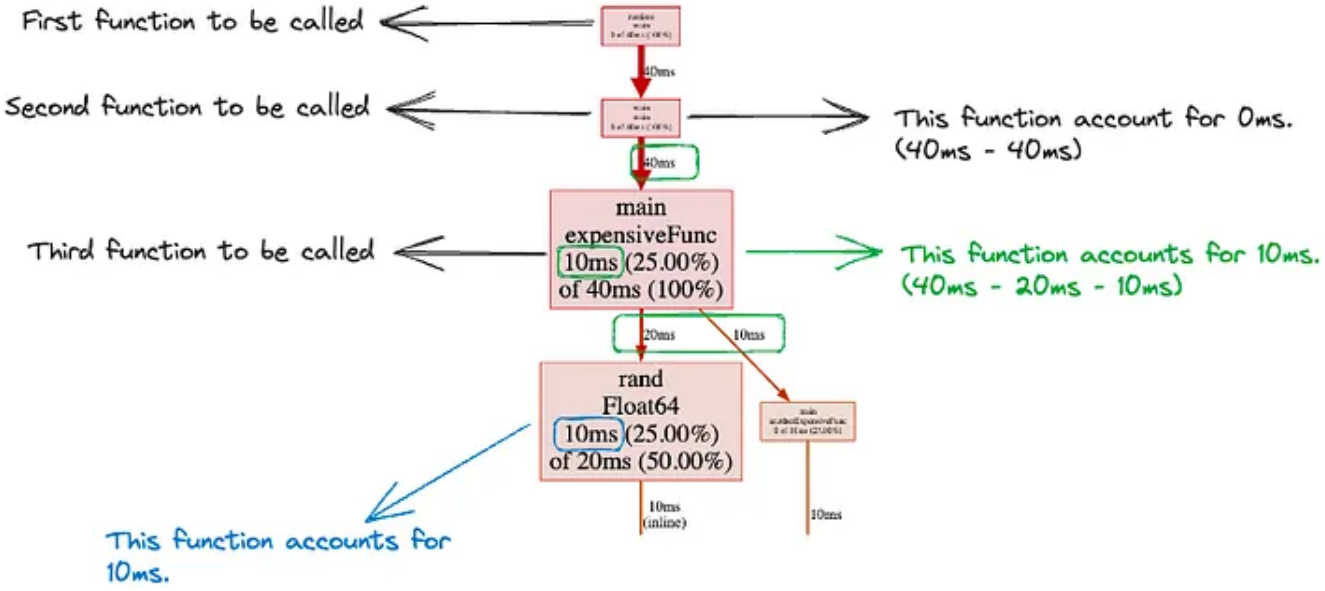
tags output

You can do lots of things with labels. You can get more information from this article.

> *pprof also has a great web interface. It allows us to analyze profiling data using various visualizations.*

Type `go tool pprof -http=:6060 cpu.prof` , `localhost:6060` will be opened. *(To be more clear, I removed pprof.Labels)*

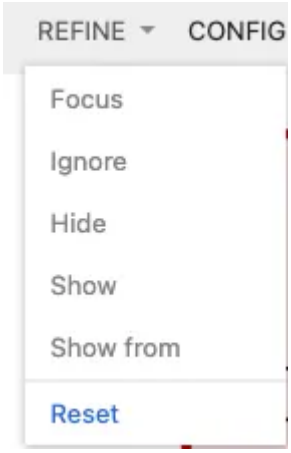
Let's dive into graph representation.



Graph for CPU analysis

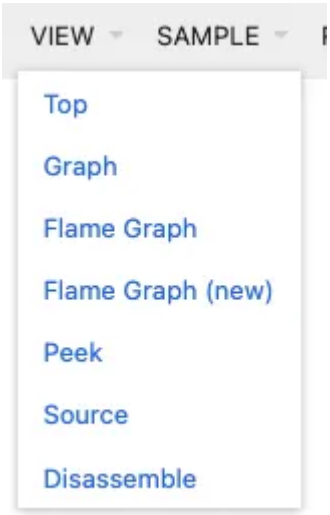
Node color, font size, edge weight, and many more have different meanings. Please refer to it here. This visualization allows us to identify and fix performance issues more easily.

After clicking a node in the graph, you can refine it. It allows us to filter the visualization based on our choice. I showed some of the above *(focus, hide, etc.)*.



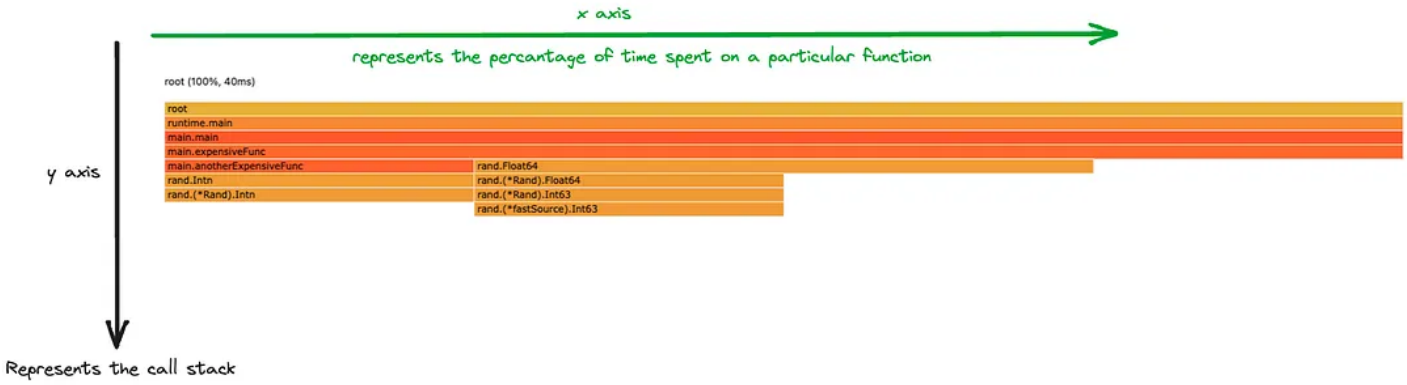
Refine options

You can also see other visualization options.



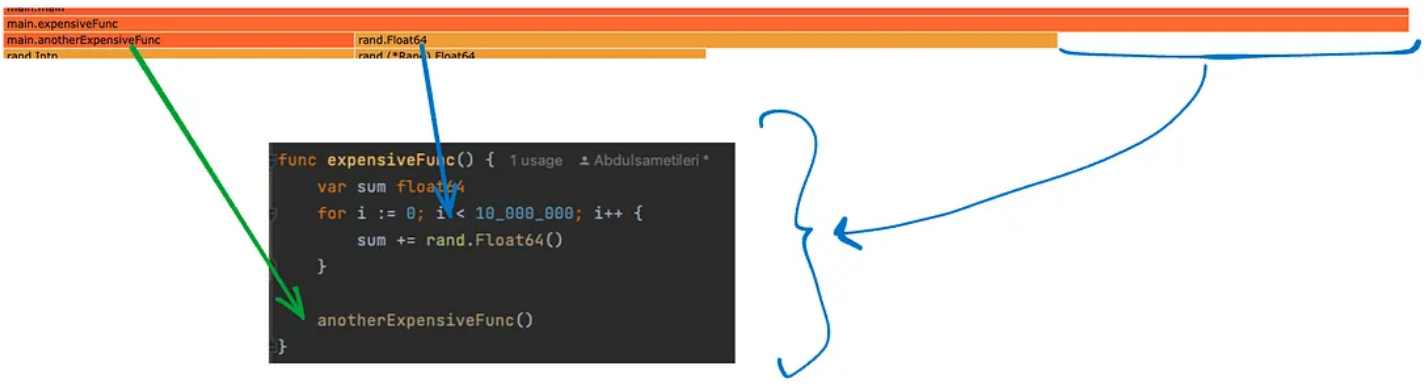
View options

I showed peek and source *(as the list command),* so I want to introduce you to Flame Graph. Flame graphs provide a high-level view of where time is spent in your code.



Flame Graph

Each function is represented by a colored rectangle where the width of the rectangle is proportional to the amount of time spent in that function.



> *You can find the source code here!*

2.2 Profiling Memory

If you need to add memory profiling support to your application, you must enable it first.

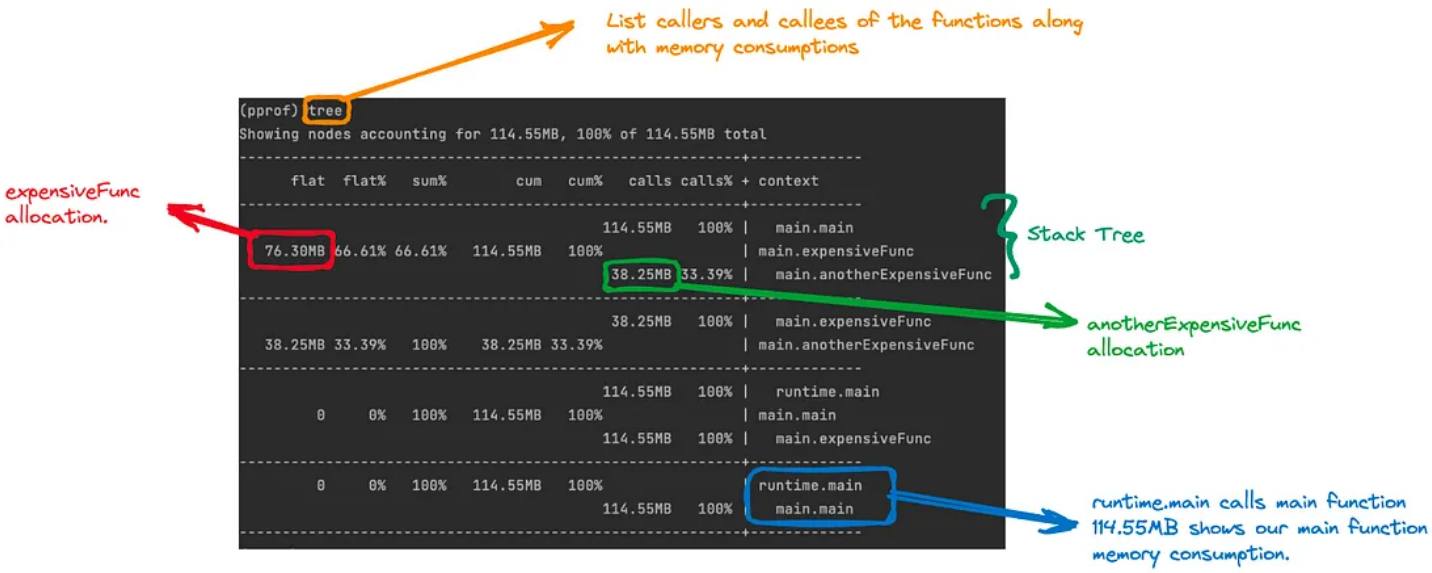
> *You can find the source code here.*

```go
func main() {
  f, err := os.Create("mem.prof")
  if err != nil {
    log.Fatal("could not create memory profile: ", err)
  }
  defer f.Close()

  runtime.GC() // invoke gc, in order to get up-to-date statistics

  // your function

  if err := pprof.WriteHeapProfile(f); err != nil {
    log.Fatal("could not write memory profile: ", err)
  }
}
```

If you run `go run .` here, you will see the generated `mem.prof` file. You can analyze it `go tool pprof mem.prof` as mentioned in the benchmarking section.

I want to introduce to you two more useful commands `tree` and `peek`.

- The `tree` shows all callers and callees of the execution flow.



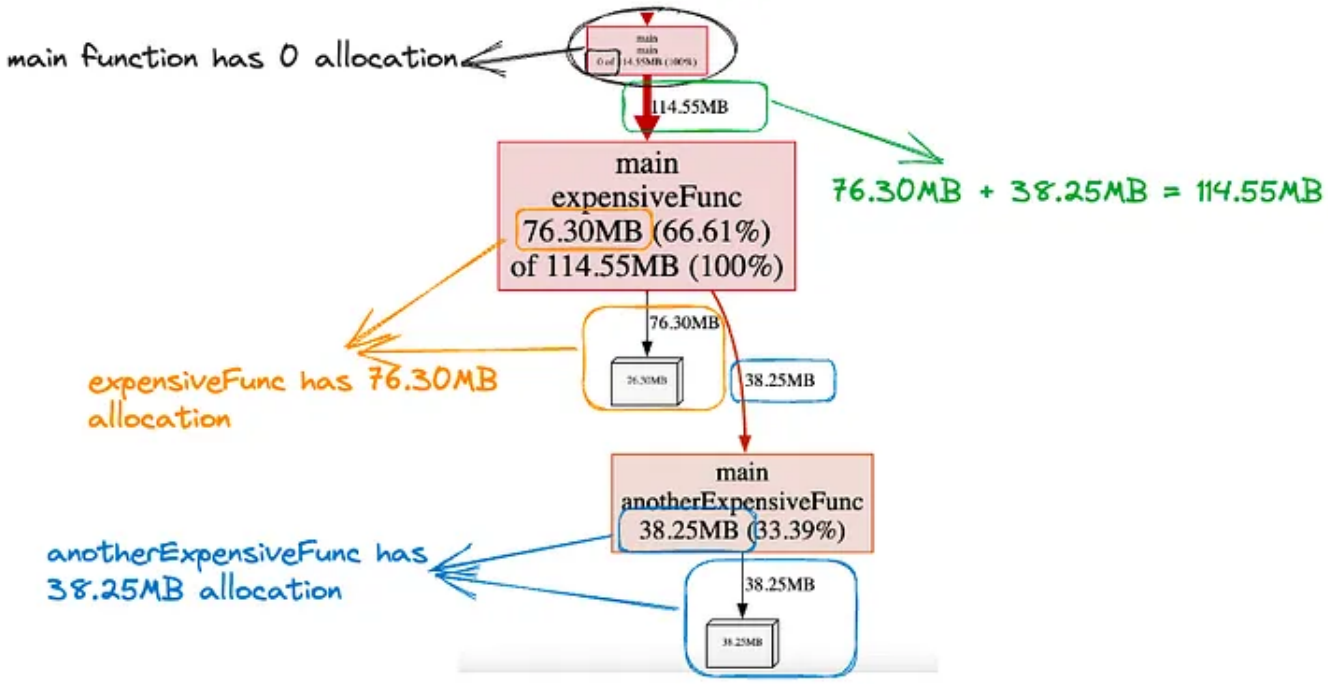
tree output

This allows us to identify the execution flow and spot the objects that consume the most memory. *(Don't forget to use* `granularity=lines` *It provides a more readable format.)*

- If you want to see a specific function execution flow, you can use `peek` command. For example `peek expensiveFunc` shows

```
(pprof) peek expensiveFunc
Showing nodes accounting for 114.55MB, 100% of 114.55MB total
----------------------------------------------------------+-------------
      flat  flat%   sum%        cum   cum%   calls calls% + context
----------------------------------------------------------+-------------
                                          114.55MB   100% |   main.main
   76.30MB 66.61% 66.61%   114.55MB   100%                | main.expensiveFunc
                                           38.25MB 33.39% |   main.anotherExpensiveFunc
----------------------------------------------------------+-------------
```

- You can also use the pprof web interface for memory analysis. Type `go tool pprof -http=:6060 mem.prof` , `localhost:6060` will be opened.

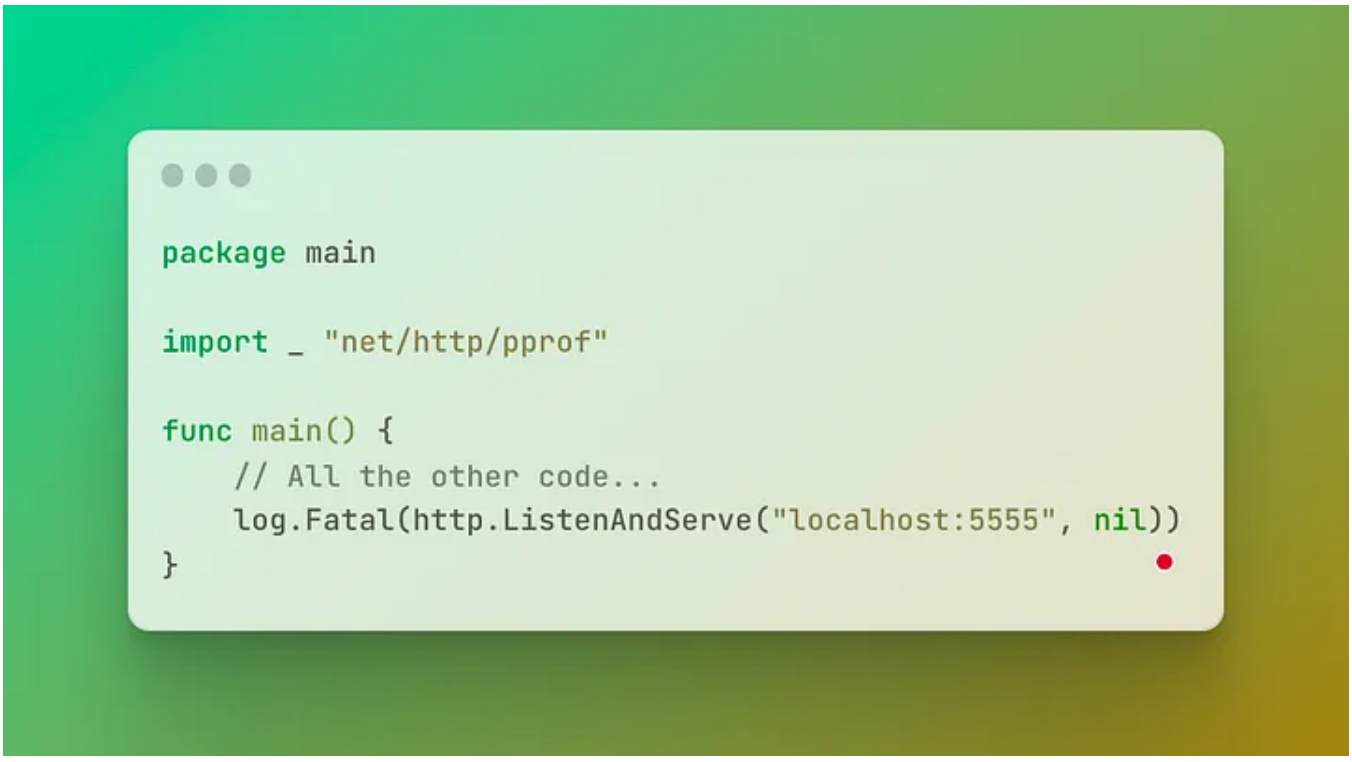


Graph for memory analysis

> *You can find the source code here!*

3. Web profiling with net/http/pprof

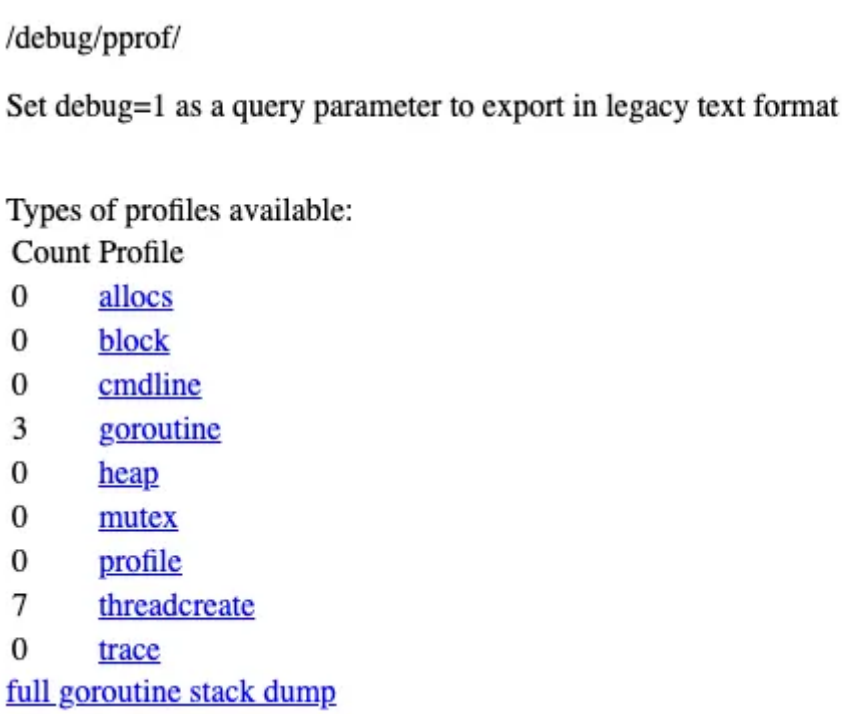
The **runtime/pprof** package offers a lower-level interface for our Go program's performance. On the other hand, **net/http/pprof** provides a higher-level interface for profiling. It allows us to collect our program profiling information through HTTP 🐓 All you can do is:



```go
package main

import _ "net/http/pprof"

func main() {
    // All the other code...
    log.Fatal(http.ListenAndServe("localhost:5555", nil))
}
```

Adding net HTTP pprof

If you type `localhost:5555/debug/pprof` , you will see all available profiles on your browser *if you are not using stdlib, check fiber, gin, or echo pprof implementations.*



debug/pprof view

All other usages and parameters are documented here. Let's review popular ones.

Getting CPU profile and Tricks

```
go tool pprof http://localhost:5555/debug/pprof/profile?seconds=30
```

During the CPU profiling, be aware

`runtime.mallogc` that → means you can optimize the number of small heap allocations.

`syscall.Read` or `syscall.Write` that → The application spends a significant amount of time in Kernel mode. You can try to do IO/buffering.

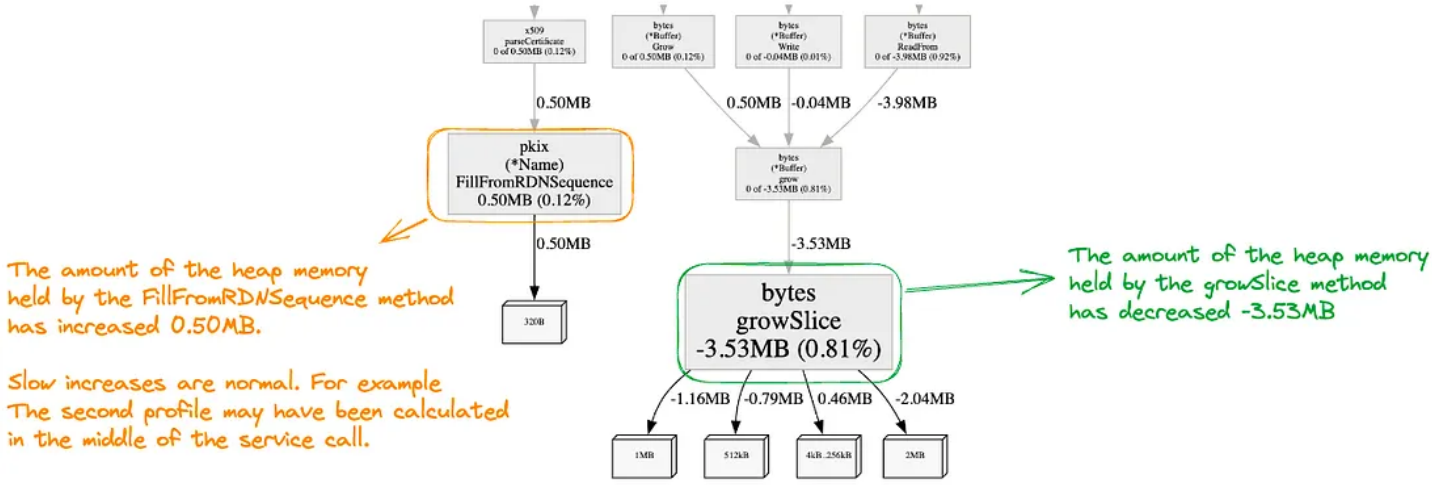
Getting Heap (a sampling of live objects' memory allocations) profile and Tricks

```
go tool pprof http://localhost:5555/debug/pprof/heap

go tool pprof http://localhost:5555/debug/pprof/heap?gc=1
```

Personally, I loved playing with GC parameters for diagnosing problems. For example, if your application has memory leak issues, you can do the following:

- Trigger GC *(Go to /debug/pprof/heap?gc=1 in your browser)*
- Download heap data; *give the downloaded file name -> file1*
- Wait for seconds/minutes.
- Trigger another GC *(Go to /debug/pprof/heap?gc=1 in your browser)*
- Download another heap of data; *give the downloaded file name -> file2*
- Compare using the diff_base

> *go tool pprof -http=:6060 -diff_base file2 file1*



diff_base output

Getting Allocation (a sampling of all past memory allocations) profile and Tricks

```
go tool pprof http://localhost:5555/debug/pprof/allocs
```

During the allocs profiling, you can do it

- If you saw `bytes.growSlice` , you should think about using `sync.Pool` .
- If you saw your functions, check whether you define fixed capacity in your slice or maps.

Thank you for reading so far 💛 . All feedback is welcome 🙏 Special thanks to Emre Odabas for the review and Mert Bulut for encouraging me.

Useful Links

1- pprof Github Readme

2- Profiling Go Programs by Russ Cox

3- pprof man page

4- GopherCon 2019: Dave Cheney — Two Go Programs, Three Different Profiling Techniques

5- GopherCon 2021: Felix Geisendörfer — Go Profiling and Observability from Scratch

6- GopherConAU 2019 — Alexander Else — Profiling a go service in production

7- Practical Go Lessons Profiling Chapter

Stackademic

Thank you for reading until the end. Before you go:

- *Please consider **clapping** and **following** the writer!* 👏
- *Follow us on **Twitter(X)**, **LinkedIn**, and **YouTube**.*
- *Visit **Stackademic.com** to find out more about how we are democratizing free programming education around the world.*

Golang Go Pprof Memory Leak Cpu

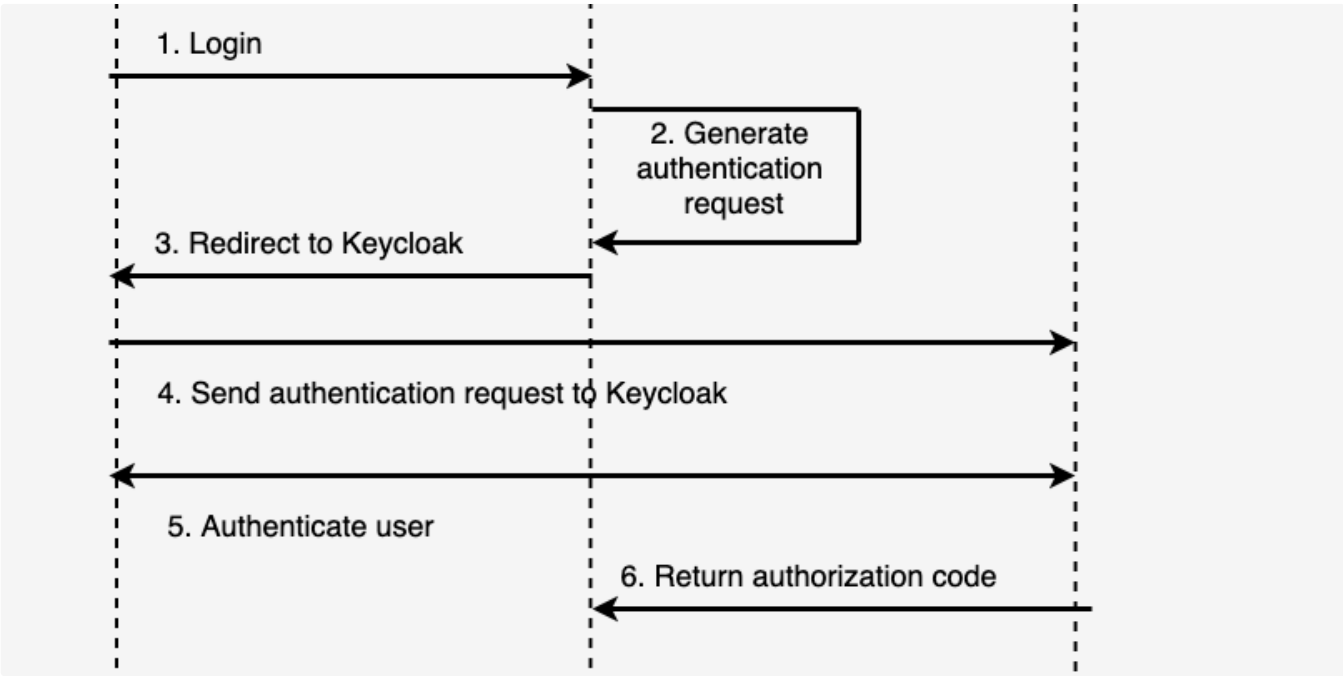
Follow

Written by Abdulsamet İLERİ

400 Followers · Writer for Stackademic

Software Engineer @Trendyol

More from Abdulsamet İLERİ and Stackademic



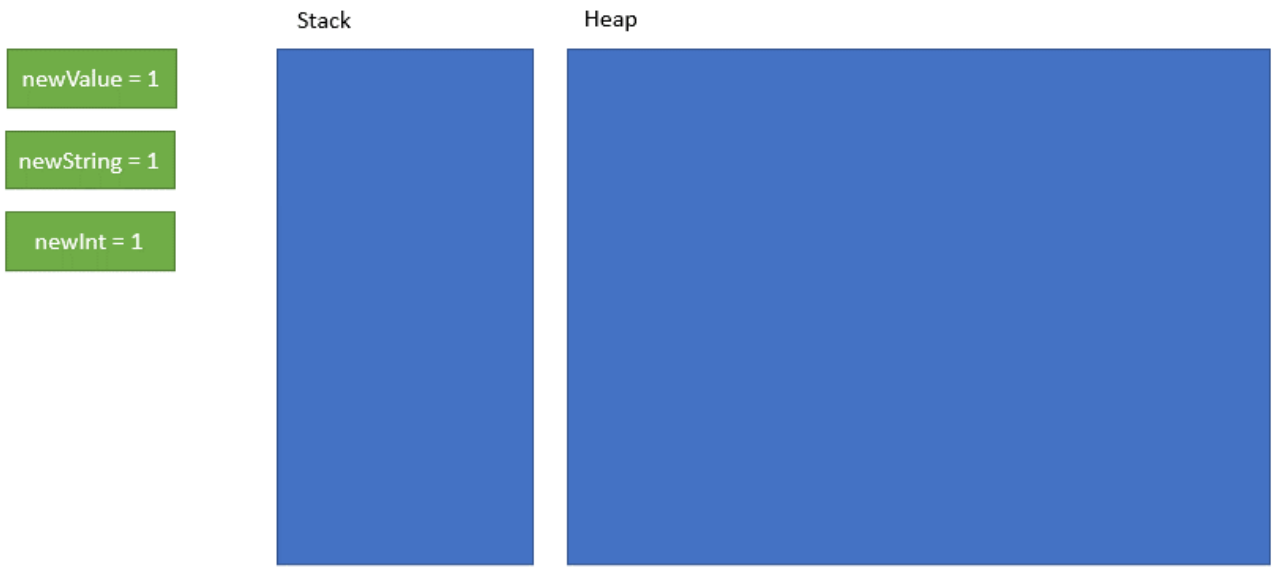
Abdulsamet İLERİ

Introduction to Keycloak

Keycloak is an open-source identity and access management tool with a focus on modern applications such as single-page applications, mobile...

6 min read · Aug 22, 2021

191 5



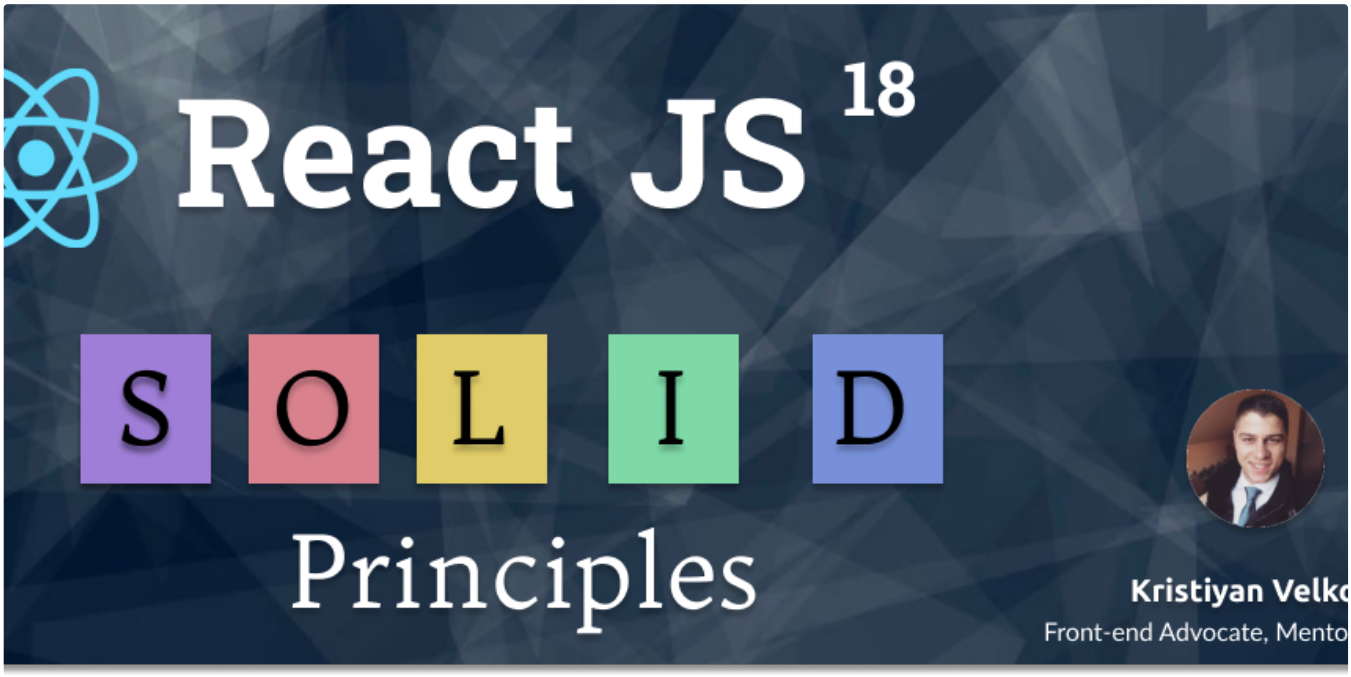
Berkay Haberal in Stackademic

How Java Memory Works?

Before we move on to the performence things, we need to learn that what is really going on in the background of JVM (Java Virtual Machine)...

4 min read · Jul 30

635 6



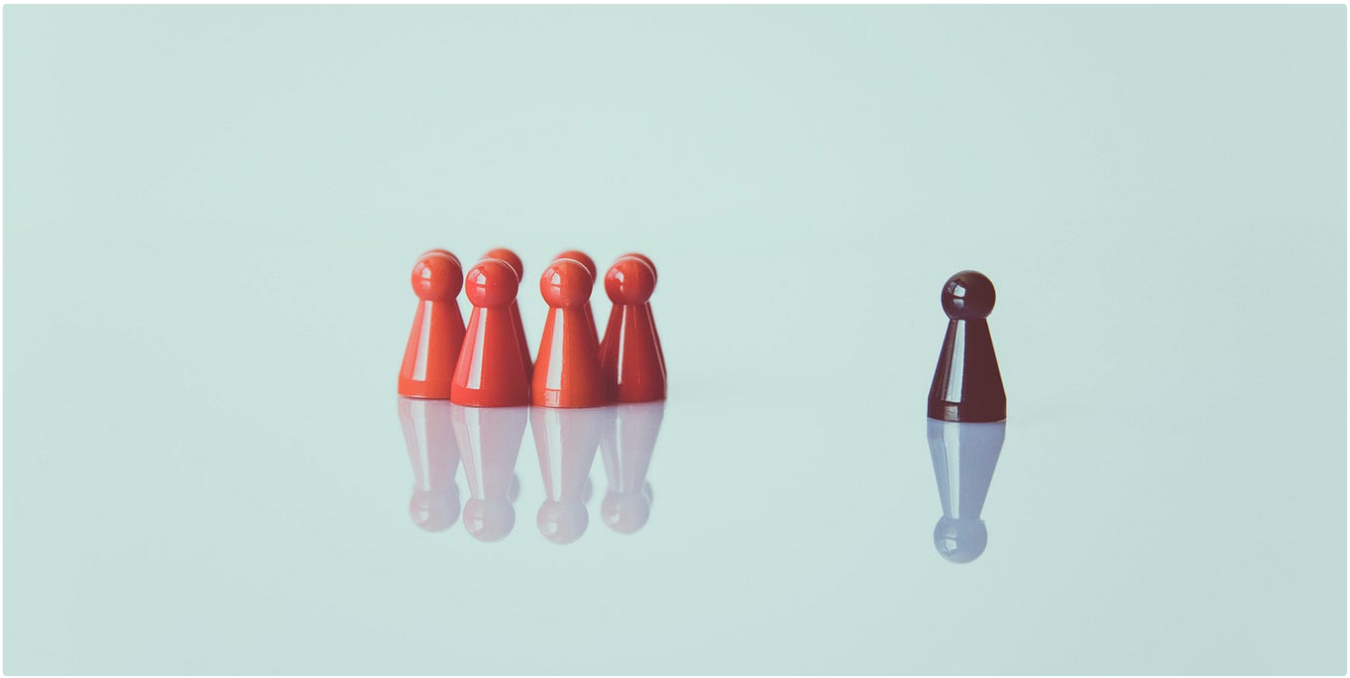
Kristiyan Velkov in Stackademic

React JS—Mastering React JS SOLID Principles

What are React JS SOLID principles?

✨ · 11 min read · Jun 26

643 9



Abdulsamet İLERİ in ITNEXT

Let’s implement a basic leader election algorithm using Go with RPC 🚀

Motivation

5 min read · Aug 7

306

See all from Abdulsamet İLERİ

See all from Stackademic

Recommended from Medium

Pascal Allen

Streaming Server-Sent Events With Go

This publication demonstrates how to stream server-sent events over HTTP with Go.

4 min read · Jul 19

27

Igor Carvalho

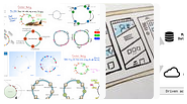
Go lang: From 0 to Employed

Go lang foundations :: Complex types :: part 1

9 min read · Jun 5

21 2

Lists



General Coding Knowledge

20 stories · 356 saves

Now in AI: Handpicked by Better Programming

266 stories · 153 saves

Mou Sam Dahal

Handle Errors In Go Like A Pro.

Errors? Manage them at Scale? Debug at runtime?

9 min read · May 11

111 1

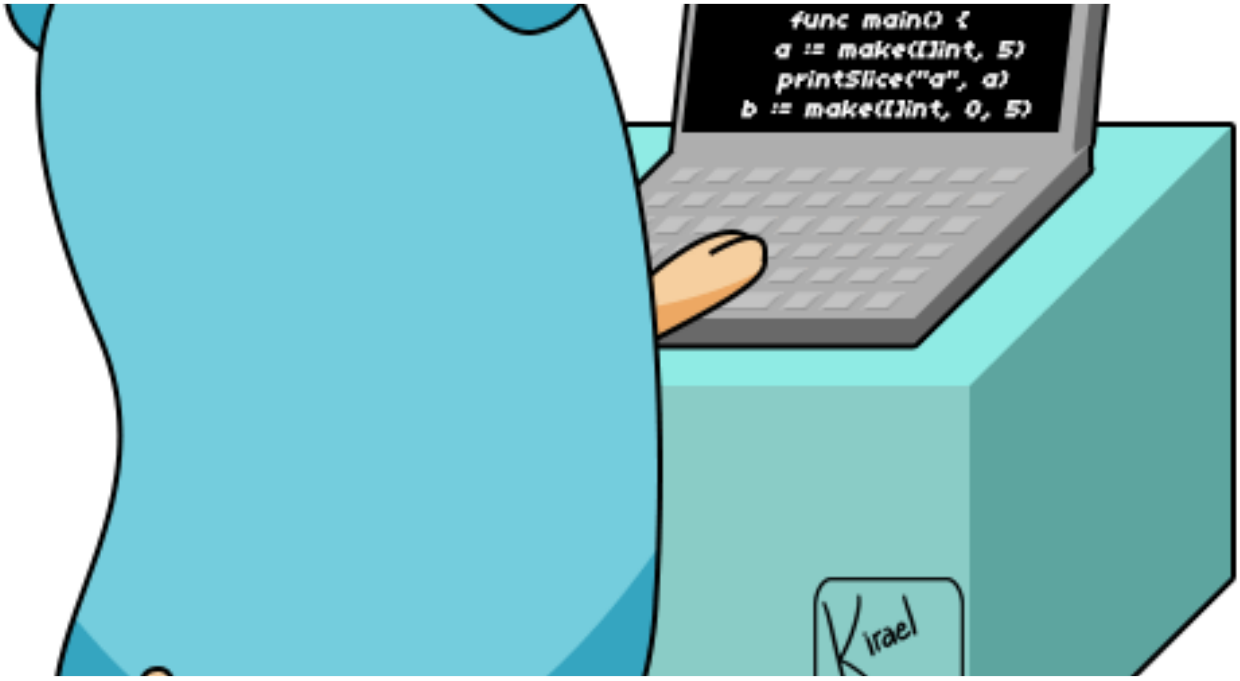
Matt Bentley in Better Programming

Why You Should Be Using Event Storming

A breakdown of the benefits of using Event Storming for solution design and a simple guide to facilitating your Event Storming workshops

✨ · 10 min read · Sep 13

635 5



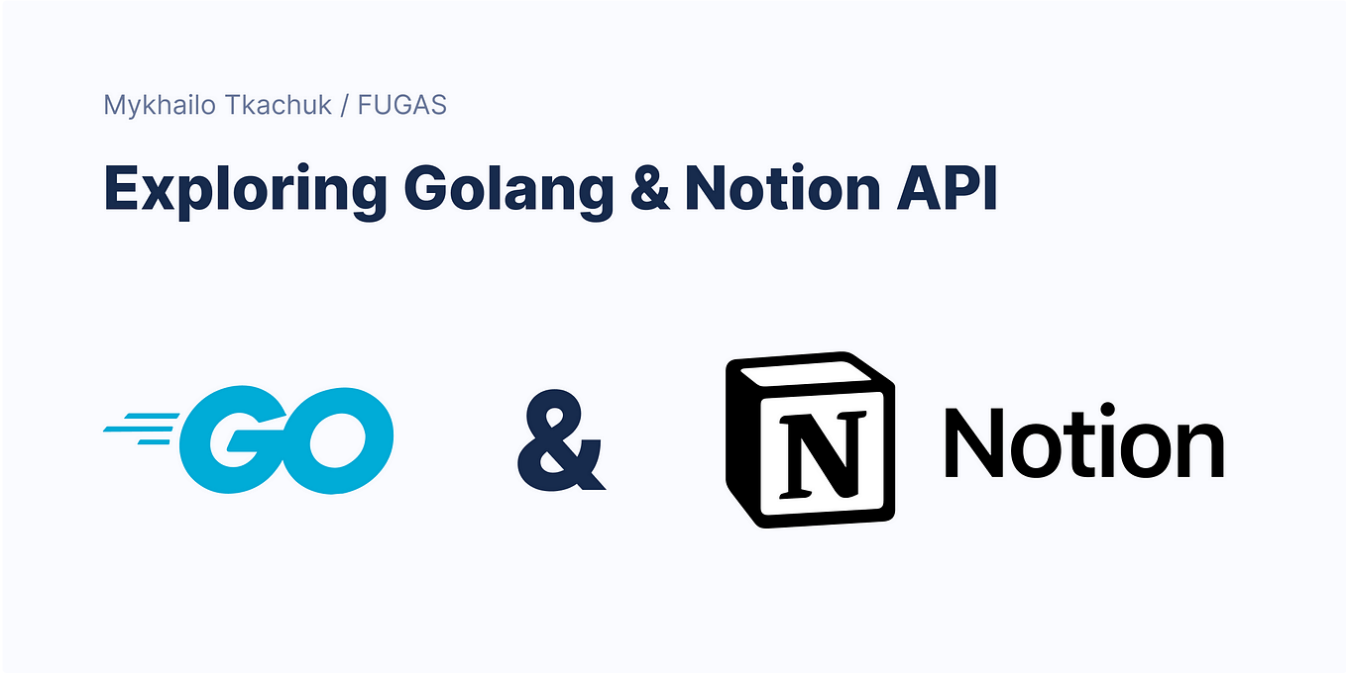
Marco Rosner in Stackademic

Go servers benchmark: Echo, Fiber, and Gin

As part of my return to the tech industry after my sabbatical, I started to learn Golang and in my first project (Simple API), I realized...

4 min read · Sep 5

43



Mykhailo Tkachuk

Exploring Golang and Notion API

Welcome to this guide, where we'll embark on a journey to explore the endless possibilities that arise when Golang and the Notion API...

6 min read · Jul 20

4

See more recommendations