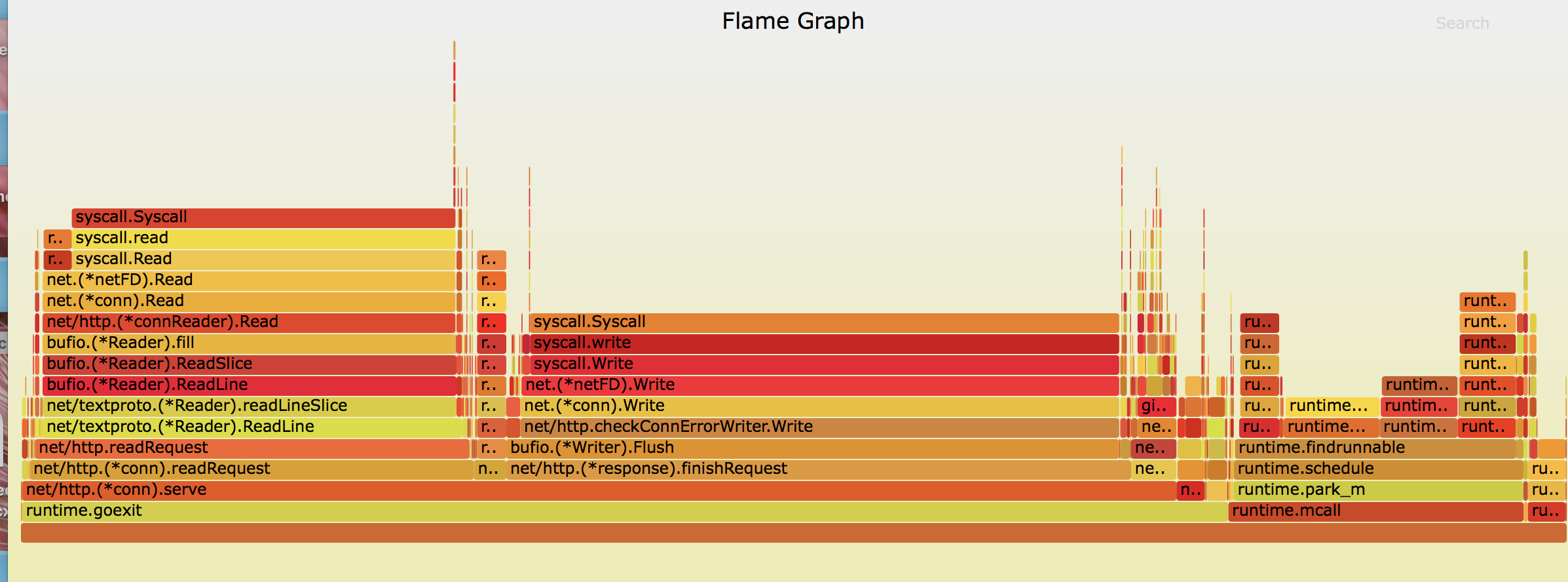
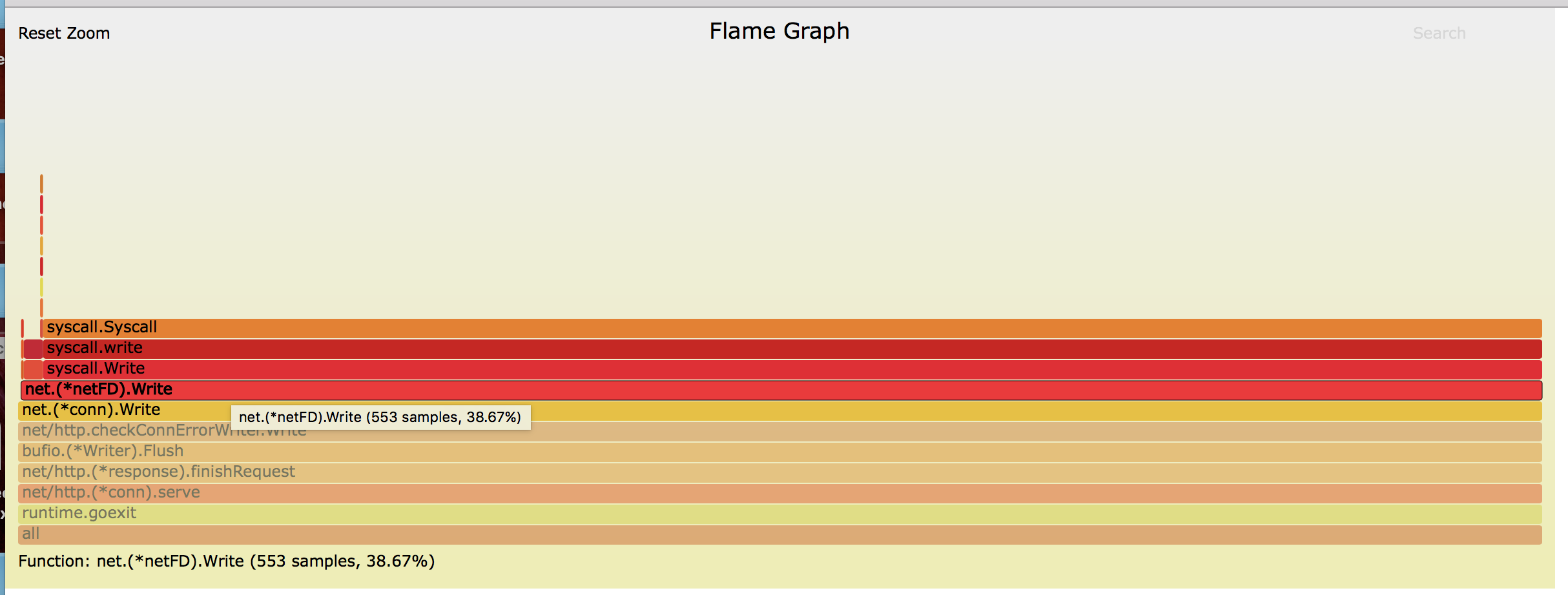
I 8: keep going “deep” in go profiling and optimization

After reading lots of documents, I found there is a very powerful tool than the .svg graph, that is called FrameGraph. It’s also a svg format graph but it’s more convenient and obvious for the observer to see the percentage of all the runtime system:



We can click each function, and zoom in to see the details. All the functions/processes blocks in the bottom position is the root of the blocks in the top

As we can see here, it also shows the sample numbers and the percentage it cost for the whole process(5 seconds here.

Note:

If we want to use go-torch for the svg, as well as update the profile after modification, we need to cd to the go-torch folder, for example:

cd $GOPATH/src/github.com/uber/go-torch

then go-torch --seconds 5 http://localhost:9090/debug/pprof/profile

The first graph is the pprof profile states I already improved in last two weeks. We can see that almost all the time is spent in XXXX.read and XXX.write. I read some answer online and realize that this is good, since read and write is mainly handled by the runtime. There is nothing we can really modify about these parts. It’s already the fastest. Even though it looks like read and write take majority of time, it doesn’t mean it takes majority of CPU. It’s just means we are blocking up I/O. The program spent most of the time waiting for the I/O reading and writing some data, waiting for the network and so on.

So, what I want to do Is keep zoom in to my handler and see where my time has been spent.

Using the tool in we found two weeks ago, where we generated the prof.cpu file and stats.test file by type in :

go test -bench . -benchmem -cpuprofile prof.cpu

that generates the benchmark result, and

go tool pprof stats.test prof.cpu

that shows the prof for AddTagsToName function.

 flat  flat%   sum%        cum   cum%

     630ms 30.88% 30.88%      630ms 30.88%  runtime.mach\_semaphore\_signal

     390ms 19.12% 50.00%     1240ms 60.78%  runtime.mallocgc

     210ms 10.29% 60.29%      210ms 10.29%  runtime.makeslice

     180ms  8.82% 69.12%      880ms 43.14%  runtime.growslice

      90ms  4.41% 73.53%     1870ms 91.67%  github.com/prashantv/go\_profiling\_talk\_slow/stats.addTagsToName

      90ms  4.41% 77.94%      460ms 22.55%  strings.Join

      80ms  3.92% 81.86%       80ms  3.92%  runtime.mach\_semaphore\_wait

      80ms  3.92% 85.78%       80ms  3.92%  runtime.mapaccess2\_faststr

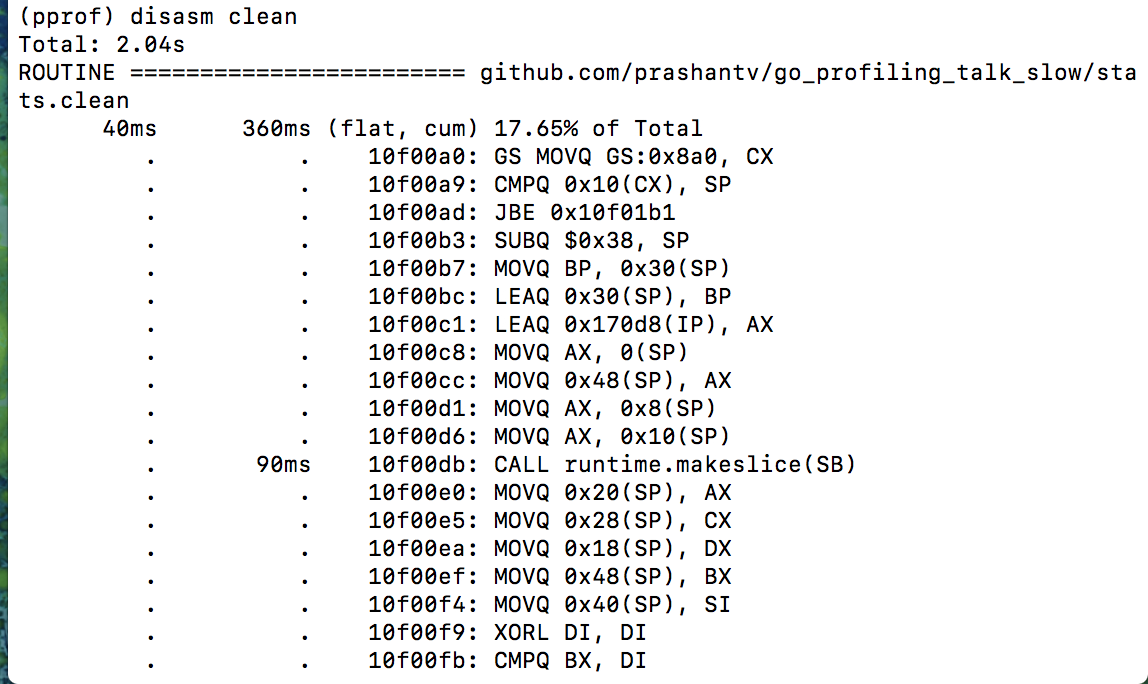
      50ms  2.45% 88.24%      250ms 12.25%  runtime.slicebytetostring

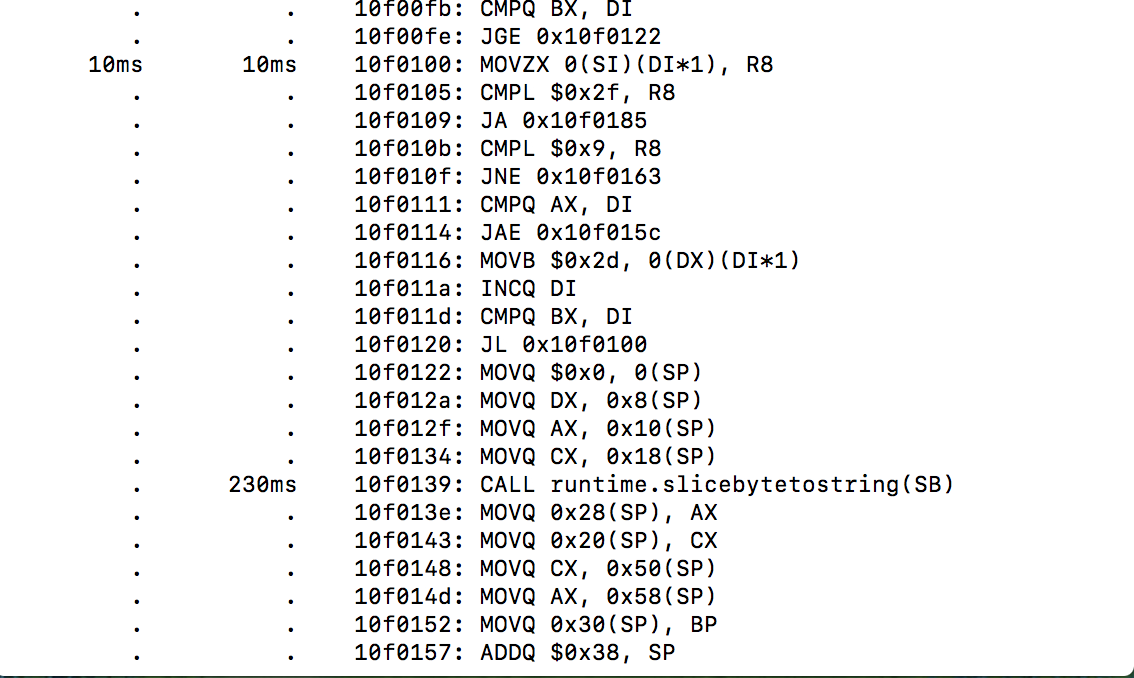
      40ms  1.96% 90.20%      360ms 17.65%  github.com/prashantv/go\_profiling\_talk\_slow/stats.clean

We can see here that we didn’t call this growslice,makeslice and slicebytetostring function, but it takes lots of time, so where does this function come from? Seems like they all do something with the slice. So I’m guessing they all have some relation in our functions

New feature: disassembling

After reading the document of the profiling, we found out that we can actually disassemble the function by typing disasm function\_name in pprof to check the bytecode of the function





We already modified our clean function before by writing our own char replacement function instead of using the library. We can see that after we disassemble our clean function, we have a makeslice function that takes 90 ms as well as a slicebytetostring function that takes 230ms.

We don’t know why these two functions take so much time, but this could be our entrance for the next move.

Using list addTagsToName, we keep checking this function

 90ms      1.87s (flat, cum) 91.67% of Total

         .          .     36:

         .          .     37:func addTagsToName(name string, tags map[string]string) string {

         .          .     38: // The format we want is: host.endpoint.os.browser

         .          .     39: // if there's no host tag, then we don't use it.

         .          .     40: var keyOrder []string

      10ms       20ms     41: if \_, ok := tags["host"]; ok {

         .       90ms     42: keyOrder = append(keyOrder, "host")

         .          .     43: }

         .      150ms     44: keyOrder = append(keyOrder, "endpoint", "os", "browser")

         .          .     45:

         .          .     46: parts := []string{name}

      20ms       20ms     47: for \_, k := range keyOrder {

      20ms       90ms     48: v, ok := tags[k]

         .          .     49: if !ok || v == "" {

         .          .     50: parts = append(parts, "no-"+k)

         .          .     51: continue

         .          .     52: }

      20ms      0.88s     53: parts = append(parts, clean(v))

         .          .     54: }

         .          .     55:

      10ms      470ms     56: return strings.Join(parts, ".")

         .          .     57:}

         .          .     58:

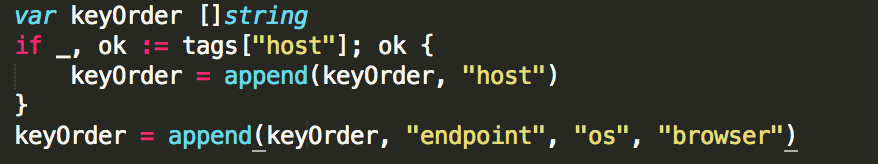
         .          .     59:

         .          .     60:

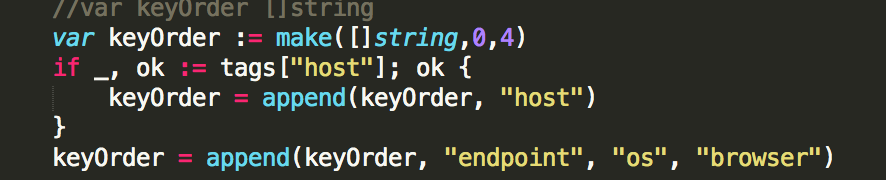
         .          .     61:// clean takes a string that may contain special characters, and replaces these

recall that we already modified the clean function so that now it takes much less memory and time. The return function takes 470ms, but for now I have no idea how to optimize this. The third largest occupation of our function is the append(keyOrder, “endpoint”, “os”, “browser”) function. Looks like we can try to make this better.

Lets look back to the code.

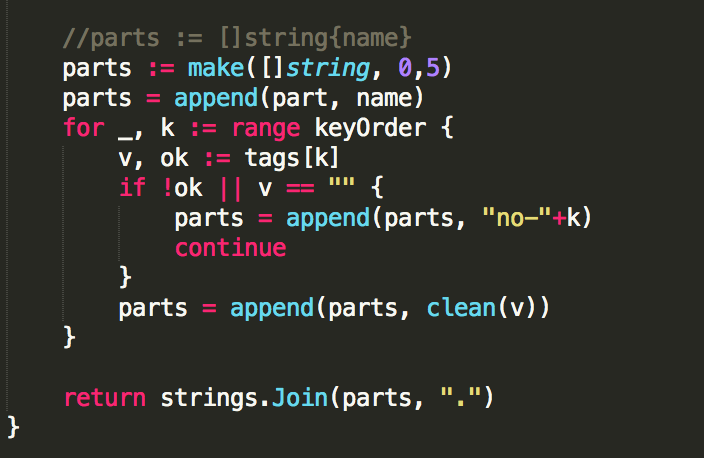


we first create a slice called KeyOrder, which in this case is an empty slice that doesn’t have any capacity and length. We then use append to add the host string to this slice. Now there is a re-allocation since we allocate a new slice, copy over all the elements (actually there is nothing we can copy form keyOrder) to the keyOrder. Since there is no capacity for this slice, we actually create a new slice and copy everything there. Then we call the append again to append the slice. In other words, we did 2 more allocation, which was a waste of time and space. We can actually improve this part of the code.

The solution is easy: we just give them initial capacity when we define them. 

This is how we modify this. Since we already know that we need four string here, so that the capacity is 4. But initially we want the slice to be empty so the length is 0.

Same thing for all the slice definition in this function:



We comment the previous version, and replace it with two line definition with 0 length and 5 capacity. We add the name(string) to it.

Lets see how the function work:

ShuyanmatoMacBook-Pro:stats shuyanli$ go test -bench . -benchmem -cpuprofile prof.cpu

BenchmarkAddTagsToName-8    3000000       386 ns/op     144 B/op       10 allocs/op

WOW!!!!!

Reminder from previous report:

169-231-98-53:stats shuyanli$ go test -bench . -benchmem -cpuprofile prof.cpu

BenchmarkAddTagsToName-8     500000       1214 ns/op     220 B/op       17 allocs/op

and now we have:

ShuyanmatoMacBook-Pro:stats shuyanli$ go test -bench . -benchmem -cpuprofile prof.cpu

BenchmarkAddTagsToName-8    3000000       580 ns/op     144 B/op       10 allocs/op

We can see that the time consumption reduce from 1.2us to 0.58 us and the allocation drops from 17 to 10. That’s a huge improvement in performance. I didn’t even expect that will happens.

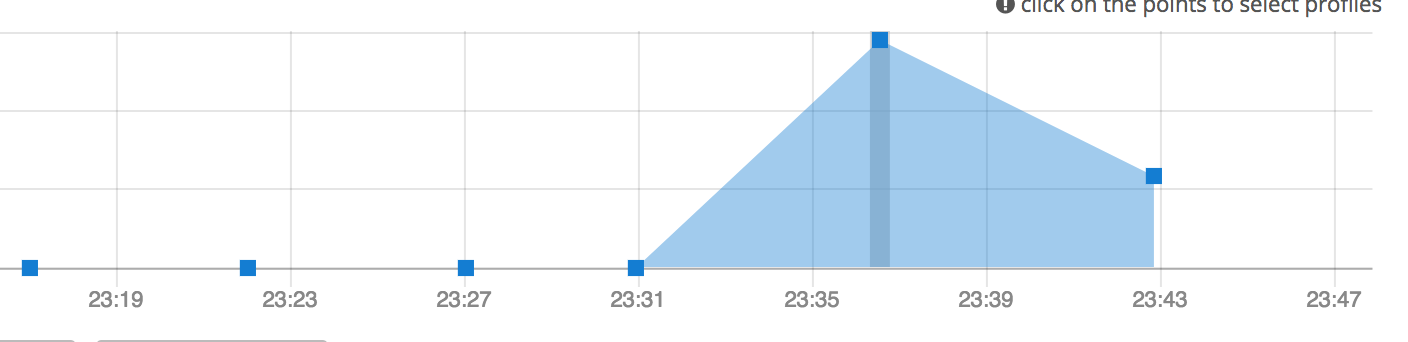


figure: cpu performance

we run our original program without this modification for 5 mins and then run the modified program for 5 mins, we can clearly see that according to the stackImpect report ,the program has a better performace.

Checking the list of the addTagsToName:

  190ms      1.97s (flat, cum) 93.36% of Total

         .          .     37:func addTagsToName(name string, tags map[string]string) string {

         .          .     38: // The format we want is: host.endpoint.os.browser

         .          .     39: // if there's no host tag, then we don't use it.

         .          .     40:

         .          .     41: //var keyOrder []string

      10ms       10ms     42: keyOrder := make([]string,0,4)

         .       30ms     43: if \_, ok := tags["host"]; ok {

      10ms       10ms     44: keyOrder = append(keyOrder, "host")

         .          .     45: }

         .          .     46: keyOrder = append(keyOrder, "endpoint", "os", "browser")

         .          .     47:

         .          .     48: //parts := []string{name}

         .       10ms     49: parts := make([]string,0,5)

      10ms       10ms     50: parts = append(parts, name)

      30ms       30ms     51: for \_, k := range keyOrder {

         .      150ms     52: v, ok := tags[k]

         .          .     53: if !ok || v == "" {

         .          .     54: parts = append(parts, "no-"+k)

         .          .     55: continue

         .          .     56: }

     120ms      960ms     57: parts = append(parts, clean(v))

         .          .     58: }

         .          .     59:

      10ms      760ms     60: return strings.Join(parts, ".")

         .          .     61:}

Now we can see all the append function has significantly reduced its time consumption. The return function that return a concatenate of two strings still takes most of the time. Since we already modify the slice append function, it proves that trying to modify the function in the library is possible. We might don’t need to modify the library itself, but the way how we define and invoke these functions.

I’m guessing here that this join function will still create a new string slice object so that it will waste lots of memory and time.

Solution:

<https://stackoverflow.com/questions/1760757/how-to-efficiently-concatenate-strings-in-go>

I found a very useful tips here:

In Go, string is a primitive type, it's readonly, every manipulation to it will create a new string.

So, if we want to concatenate strings many times without knowing the length of the resulting string, what's the best way to do it?

The naive way would be:

s := ""

for i := 0; i < 1000; i++ {

s += getShortStringFromSomewhere()

}

return s

but that does not seem very efficient.

Answer:

The best way is to use the [bytes](http://golang.org/pkg/bytes/) package. It has a [Buffer](http://golang.org/pkg/bytes/#Buffer) type which implements [io.Writer](http://golang.org/pkg/io/#Writer).

package main

import (

"bytes"

"fmt"

)

func main() {

var buffer bytes.Buffer

for i := 0; i < 1000; i++ {

buffer.WriteString("a")

}

fmt.Println(buffer.String())

}

This does it in O(n) time.

Another answer suggests that using copy() is more efficient. However, after I read some documents, it states that the bytes.Buffer should do basically the same as the copy (with some extra bookkeeping I guess) and the speed isn't that different. So I'd use buffer.WriteString. The difference being that the buffer starts with 0 bytes so it has to reallocate (this make it seem a little slower I guess). Easier to use, though.

One tip from other programmer:

buffer.Write (bytes) is 30% faster than buffer.WriteString. [useful if you can get the data as []byte]

we can combine all these and try to modify our program.