Weekly assignment week 6 and week 7.

Almost finishing the stackImpect research. I will post another study note based on that. In a nutshell, it’s not pretty useful in doing profiling, but it’s a good tool for checking and comparing the overall performance of the program.

Beside stackImpect:

Found an adder function in go online.

Involve both parsing, data structure and http request, as well as handler. This could be a good source to practice how to use benchmark, how to change the structure of the code after doing the profiling.

Writing a benchmark:

func BenchmarkHandleStructAdd(b \*testing.B) {

r := request(b, "/?first=20&second=30")

for i := 0; i < b.N; i++ {

rw := httptest.NewRecorder()

handleStructAdd(rw, r)

}

}

using b \*testing.B (I might need to do some reading on testing.B, since I have no idea how it work. Instead, I simply use the conclusion that testing.B is for the benchmark generating), we run our to-test-function b.N times.

Run the benchmark:

ShuyanmatoMacBook-Pro:simple\_adder shuyanli$ go test -run=xxx -bench=. | tee bench0

BenchmarkHandleStructAdd-8     500000       32886 ns/op

PASS

It shows the time the program spent per iteration.

We also generate a file called bench0, we can then compare the performace of each result of the benchmark

Using the pprof tools

ShuyanmatoMacBook-Pro:simple\_adder shuyanli$ go tool pprof adder.test cpu.out

Entering interactive mode (type "help" for commands)

(pprof) top10

2800ms of 2870ms total (97.56%)

Dropped 31 nodes (cum <= 14.35ms)

Showing top 10 nodes out of 82 (cum >= 30ms)

      flat  flat%   sum%        cum   cum%

    1700ms 59.23% 59.23%     1700ms 59.23%  syscall.Syscall

     480ms 16.72% 75.96%      480ms 16.72%  runtime.mach\_semaphore\_wait

     470ms 16.38% 92.33%      470ms 16.38%  runtime.mach\_semaphore\_signal

     150ms  5.23% 97.56%      150ms  5.23%  runtime.usleep

         0     0% 97.56%     2090ms 72.82%  \_/Users/shuyanli/Desktop/adder.BenchmarkHandleStructAdd

         0     0% 97.56%     2090ms 72.82%  \_/Users/shuyanli/Desktop/adder.handleStructAdd

         0     0% 97.56%      160ms  5.57%  bytes.(\*Buffer).ReadFrom

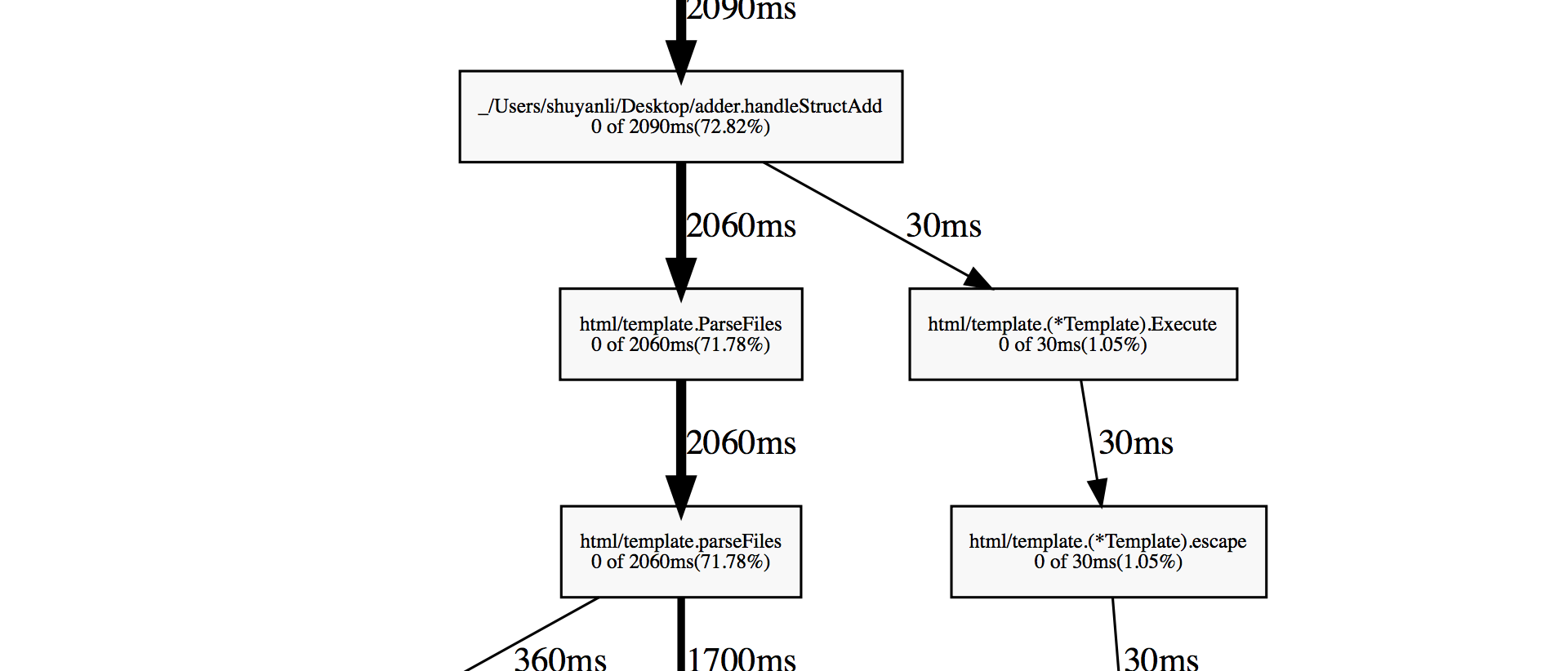
         0     0% 97.56%       30ms  1.05%  html/template.(\*Template).Execute

         0     0% 97.56%      360ms 12.54%  html/template.(\*Template).Parse

         0     0% 97.56%       30ms  1.05%  html/template.(\*Template).escape

we can see that the handleStructAd, Parse spent top10 of the time. Since this is our function, we want to minimize the time consumption and memory allocation.

Check the .svg diagram:



we can see that ParseFIles are taking the dominating time. We could do this once the program begins and just execute it once the API handler gets the request.

Lets check the code:

In handleStructadd function:

|  |
| --- |
|  |

m := struct{ a, b int }{one, two}

structSum := add{Sum: m.a + m.b}

t, err := template.ParseFiles("template.html")

if err != nil {

http.Error(w, err.Error(), 500)

}

err = t.Execute(&html, structSum)

We define the input, wait for the request, and then wait for the error panic. As a result, we are doing this iteration over and over again. Let’s try do this function before doing all these things:

var templates = template.Must(template.ParseFiles("template.html"))

we add this one line of code so that the templates will be a globel variable. We don’t have to do it over and over again. We implement this simple cache so that the template will not be rendered 50 thousand times.

Let’s check the performace:

ShuyanmatoMacBook-Pro:simple\_adder shuyanli$ go test -run=xxx -bench=. | tee bench0

BenchmarkHandleStructAdd-8     500000      12764 ns/op

PASS

Compare to the previous performance:

ShuyanmatoMacBook-Pro:simple\_adder shuyanli$ go test -run=xxx -bench=. | tee bench0

BenchmarkHandleStructAdd-8     500000       32886 ns/op

PASS

We can draw one conclusion that caching the program, especially the top used, hotspot variables, can significantly reduce the time consumption. The reason is that most of the repeatition of the loop will waste lots of time by calling useless function. Most top-used variable need to be assigned only one time

Check the new pprof cpu profiling top10:

ShuyanmatoMacBook-Pro:simple\_adder shuyanli$ go tool pprof adder.test cpu.out

Entering interactive mode (type "help" for commands)

(pprof) top10

680ms of 1400ms total (48.57%)

Showing top 10 nodes out of 160 (cum >= 210ms)

      flat  flat%   sum%        cum   cum%

     300ms 21.43% 21.43%      300ms 21.43%  runtime.mach\_semaphore\_signal

     150ms 10.71% 32.14%      510ms 36.43%  runtime.mallocgc

      40ms  2.86% 35.00%       40ms  2.86%  runtime.memmove

      40ms  2.86% 37.86%      140ms 10.00%  runtime.newarray

      30ms  2.14% 40.00%      270ms 19.29%  reflect.Value.call

      30ms  2.14% 42.14%       30ms  2.14%  runtime.freedefer

      30ms  2.14% 44.29%       30ms  2.14%  runtime.mapiternext

      20ms  1.43% 45.71%       80ms  5.71%  html/template.htmlReplacer

      20ms  1.43% 47.14%      270ms 19.29%  net/http/httptest.(\*ResponseRecorder).writeHeader

      20ms  1.43% 48.57%      210ms 15.00%  net/http/httptest.cloneHeader

(pprof)

clearly, our most time consuming functions is no longer on the top 10 list

checking the detail by tpying list Func\_name:

ROUTINE ======================== \_/Users/shuyanli/Desktop/adder.handleStructAdd in /Users/shuyanli/Desktop/adder/adder.go

      10ms      1.10s (flat, cum) 78.57% of Total

         .          .     14:

         .          .     15:var templates = template.Must(template.ParseFiles("template.html"))

         .          .     16:

         .          .     17:func handleStructAdd(w http.ResponseWriter, r \*http.Request) {

         .          .     18:

         .       90ms     19: var html bytes.Buffer

         .          .     20: first, second := r.FormValue("first"), r.FormValue("second")

         .       10ms     21: one, err := strconv.Atoi(first)

         .          .     22: if err != nil {

         .          .     23: http.Error(w, err.Error(), 500)

         .          .     24: }

         .          .     25: two, err := strconv.Atoi(second)

         .          .     26: if err != nil {

         .          .     27: http.Error(w, err.Error(), 500)

         .          .     28: }

         .          .     29: m := struct{ a, b int }{one, two}

         .          .     30: structSum := add{Sum: m.a + m.b}

         .          .     31:

         .      590ms     32: err = templates.Execute(&html, structSum)

         .          .     33:

         .          .     34: if err != nil {

         .          .     35: http.Error(w, err.Error(), 500)

         .          .     36: }

         .       90ms     37: w.Header().Set("Content-Type", "text/html; charset=utf-8")

      10ms      320ms     38: w.Write([]byte(html.String()))

         .          .     39:}

         .          .     40:

         .          .     41:func main() {

         .          .     42:

         .          .     43: http.HandleFunc("/struct", handleStructAdd)

(pprof)

we can see that calling the Execute takes majority of the time in this function. Checking details of the execute:

  0      590ms (flat, cum) 42.14% of Total

         .          .    112:// If an error occurs executing the template or writing its output,

         .          .    113:// execution stops, but partial results may already have been written to

         .          .    114:// the output writer.

         .          .    115:// A template may be executed safely in parallel.

         .          .    116:func (t \*Template) Execute(wr io.Writer, data interface{}) error {

         .       30ms    117: if err := t.escape(); err != nil {

         .          .    118: return err

         .          .    119: }

         .      560ms    120: return t.text.Execute(wr, data)

         .          .    121:}

         .          .    122:

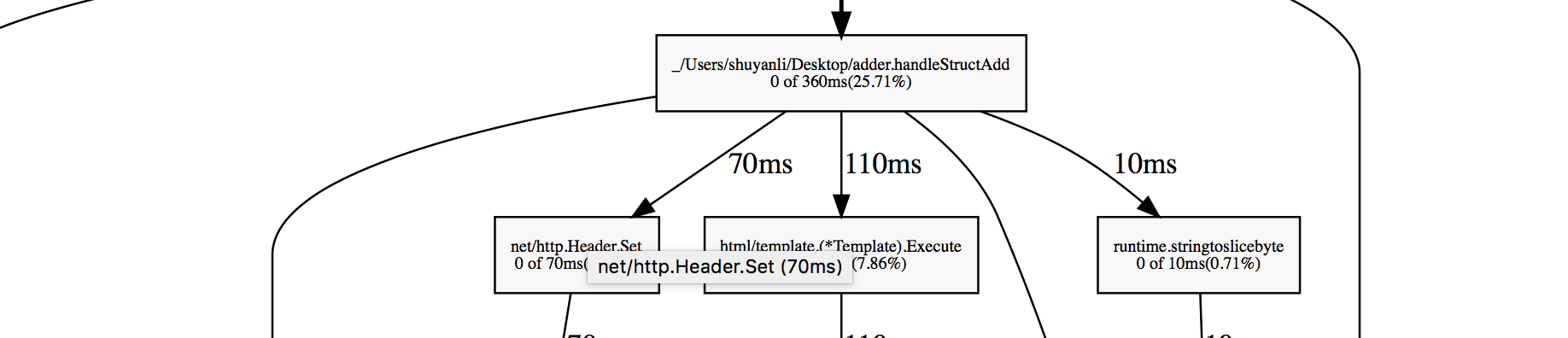
         .          .    123:// ExecuteTemplate applies the template associated with t that has the given

         .          .    124:// name to the specified data object and writes the output to wr.

         .          .    125:// If an error occurs executing the template or writing its output,

now it involve return, by calling another Execute, that is a template pointer type.

I don’t know why this function take so much time. We should restart our profiling by checking the GC in runtime.mallocgc function:



which means that simply checking the time consuming will not fix the problem. We need to find out what slow down the GC process. According to our previous work, memory allocation number is the most significant thing we need to reduce.

Add the memmalloc flag in the benchmark. We can see the memory allocation number:

ShuyanmatoMacBook-Pro:simple\_adder shuyanli$ go test -bench=.

BenchmarkHandleStructAdd-8     500000       2758 ns/op     1528 B/op       21 allocs/op

We can see that there are 21 allocation numbers in our program, which is huge.

We have two files here, cpu.out and mem.out, that is the binary for cpu and memory performance.

lets see the top 10 results:

ShuyanmatoMacBook-Pro:simple\_adder shuyanli$ go tool pprof --alloc\_space adder.test mem0.out

Entering interactive mode (type "help" for commands)

(pprof) top10

701.15MB of 745.65MB total (94.03%)

Dropped 13 nodes (cum <= 3.73MB)

Showing top 10 nodes out of 32 (cum >= 13MB)

      flat  flat%   sum%        cum   cum%

  362.12MB 48.56% 48.56%   362.12MB 48.56%  net/textproto.MIMEHeader.Set

89.51MB 12.00% 60.57%   745.65MB   100%  \_/Users/shuyanli/Desktop/adder.BenchmarkHandleStructAdd

   51.01MB  6.84% 67.41%   635.14MB 85.18%  \_/Users/shuyanli/Desktop/adder.handleStructAdd

      45MB  6.04% 73.44%       45MB  6.04%  html/template.htmlReplacer

   44.50MB  5.97% 79.41%    44.50MB  5.97%  runtime.makemap

   34.50MB  4.63% 84.04%   132.51MB 17.77%  reflect.Value.call

      25MB  3.35% 87.39%       25MB  3.35%  reflect.unsafe\_New

      21MB  2.82% 90.21%   167.51MB 22.46%  text/template.(\*Template).execute

   15.50MB  2.08% 92.29%       24MB  3.22%  reflect.MakeSlice

13MB  1.74% 94.03%       13MB  1.74%  runtime.stringtoslicebyte

lets check our number one memory consumption function:

pprof) list MIMEHeader.Set

Total: 745.65MB

ROUTINE ======================== net/textproto.MIMEHeader.Set in /usr/local/go/src/net/textproto/header.go

       7MB   186.56MB (flat, cum) 25.02% of Total

         .          .     17:

         .          .     18:// Set sets the header entries associated with key to

         .          .     19:// the single element value. It replaces any existing

         .          .     20:// values associated with key.

         .          .     21:func (h MIMEHeader) Set(key, value string) {

   48.56MB   362.12MB     22: h[CanonicalMIMEHeaderKey(key)] = []string{value}

         .          .     23:}

         .          .     24:

         .          .     25:// Get gets the first value associated with the given key.

         .          .     26:// It is case insensitive; CanonicalMIMEHeaderKey is used

         .          .     27:// to canonicalize the provided key.

(pprof)

We found the place to optimize, we will get back to this in the future.

==================================week 9 ===========================

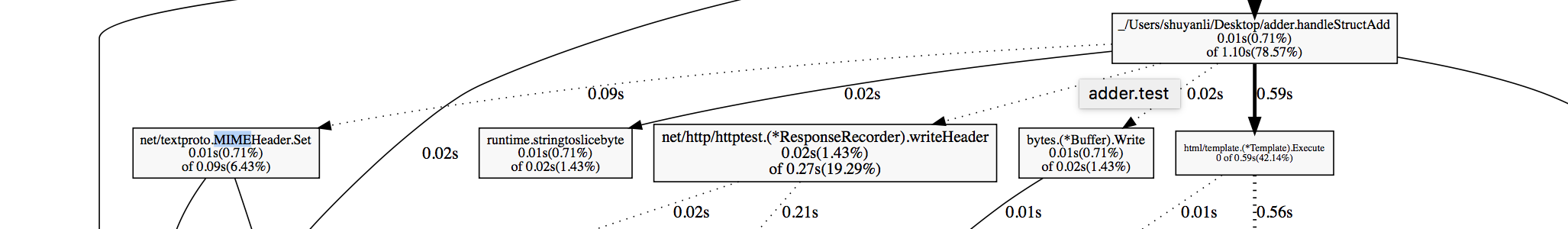
As promised, I go back to the place that stopped us last time.

After reading the document, I found the description for the function:

func (h [MIMEHeader](https://golang.org/pkg/net/textproto/#MIMEHeader)) Set(key, value [string](https://golang.org/pkg/builtin/#string))

Set sets the header entries associated with key to the single element value. It replaces any existing values associated with key.

Since this is a library (package) we call, we need to find out where we actually use this function



We can see that handleStructAdd calls this function.

We should focus back to our handleStructAddfunction.

Let’s check its memory consumption:

ROUTINE ======================== \_/Users/shuyanli/Desktop/adder.handleStructAdd in /Users/shuyanli/Desktop/adder/adder.go

   51.01MB   635.14MB (flat, cum) 85.18% of Total

         .          .     14:

         .          .     15:var templates = template.Must(template.ParseFiles("template.html"))

         .          .     16:

         .          .     17:func handleStructAdd(w http.ResponseWriter, r \*http.Request) {

         .          .     18:

   51.01MB    51.01MB     19: var html bytes.Buffer

         .          .     20: first, second := r.FormValue("first"), r.FormValue("second")

         .          .     21: one, err := strconv.Atoi(first)

         .          .     22: if err != nil {

         .          .     23: http.Error(w, err.Error(), 500)

         .          .     24: }

         .          .     25: two, err := strconv.Atoi(second)

         .          .     26: if err != nil {

         .          .     27: http.Error(w, err.Error(), 500)

         .          .     28: }

         .          .     29: m := struct{ a, b int }{one, two}

         .          .     30: structSum := add{Sum: m.a + m.b}

         .          .     31:

         .   170.51MB     32: err = templates.Execute(&html, structSum)

         .          .     33:

         .          .     34: if err != nil {

         .          .     35: http.Error(w, err.Error(), 500)

         .          .     36: }

         .          .     37: w.Header().Set("Content-Type", "text/html; charset=utf-8")

         .   199.56MB     38: w.Write([]byte(html.String()))

         .   214.06MB     39:}

         .          .     40:

         .          .     41:func main() {

         .          .     42:

         .          .     43: http.HandleFunc("/struct", handleStructAdd)

         .          .     44: log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))

(pprof)

there are 4 places that dominate the program. I don’t know why a right bracket consum memory. I guess they refer to Header().Set and Write().

Accoring to our previous profiling experience, write function is the least thing we want to modify since it is already well implemented by the designer of the language.

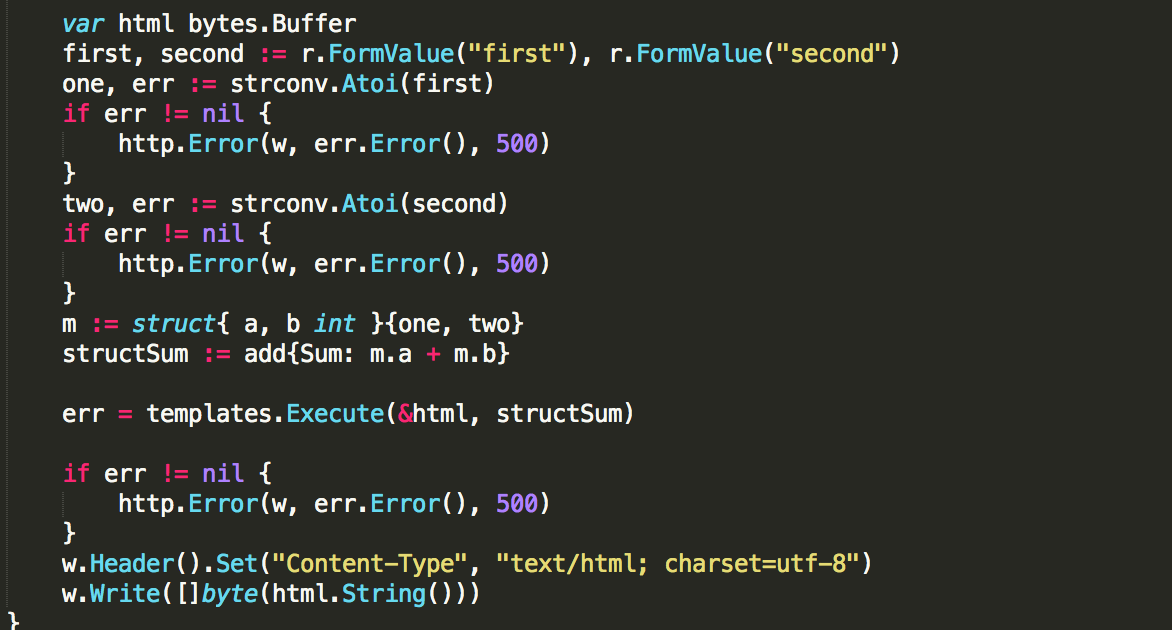
Checking the execute function:

**func (\*Template)**[**Execute**](https://golang.org/src/html/template/template.go?s=3389:3453#L106)

func (t \*[Template](https://golang.org/pkg/html/template/#Template)) Execute(wr [io](https://golang.org/pkg/io/).[Writer](https://golang.org/pkg/io/#Writer), data interface{}) [error](https://golang.org/pkg/builtin/#error)

Execute applies a parsed template to the specified data object, writing the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel.

As we can see, the Execute function will write the output to io.Writer.

Back to our code:

**Lets find out the algorithm here:**

var html bytes.Buffer

err = templates.Execute(&html, structSum)

w.Header().Set("Content-Type", "text/html; charset=utf-8")

w.Write([]byte(html.String()))

Here, we create a Buffer called html, then we pass this buffer to Execute to write in the output through IO, we then create a header and create template strings. Then it calls write function to pass the buffer to a string and write them to a slice.

We can see here that the code actually generates lots of intermediate objects. Like create an A, pass A to B, generate a C and pass B to C. One intermediate objects means more memory allocation, and it means a possible GC, which means that the system need to do extra work for the program.

Let’s get rid of all these intermediate objects:

We don’t need the buffer, and we don’t need to call IO.write twice, instead, we simply pass in the IO.w in the Execute function so that we can get the output directly.

Optimization:

Let’s check the performance:

ShuyanmatoMacBook-Pro:simple\_adder shuyanli$ go test -bench=.

BenchmarkHandleStructAdd-8     500000       2890 ns/op     1384 B/op       17 allocs/op

PASS

ShuyanmatoMacBook-Pro:simple\_adder shuyanli$ go test -bench=.

BenchmarkHandleStructAdd-8     500000       2758 ns/op     1528 B/op       21 allocs/op

Let’s compare it to the previous results:

We can see that the allocation number decrease from 21 to 17.

New memory consumption:

 51.01MB   635.14MB (flat, cum) 85.18% of Total

         .          .     14:

         .          .     15:var templates = template.Must(template.ParseFiles("template.html"))

         .          .     16:

         .          .     17:func handleStructAdd(w http.ResponseWriter, r \*http.Request) {

         .          .     18: //get rid of the intermediate objects.

   51.01MB    51.01MB     19: //instead of using a buffer, we pass the IO.write directly to templates.Execute()

         .          .     20:

         .          .     21:

         .          .     22: //var html bytes.Buffer

         .          .     23: first, second := r.FormValue("first"), r.FormValue("second")

         .          .     24: one, err := strconv.Atoi(first)

         .          .     25: if err != nil {

         .          .     26: http.Error(w, err.Error(), 500)

         .          .     27: }

         .          .     28: two, err := strconv.Atoi(second)

         .          .     29: if err != nil {

         .          .     30: http.Error(w, err.Error(), 500)

         .          .     31: }

         .   170.51MB     32: m := struct{ a, b int }{one, two}

         .          .     33: structSum := add{Sum: m.a + m.b}

         .          .     34:

         .          .     35: //err = templates.Execute(&html, structSum)

         .          .     36: err = templates.Execute(w, structSum)

         .          .     37:

         .   199.56MB     38: if err != nil {

         .   214.06MB     39: http.Error(w, err.Error(), 500)

         .          .     40: }

         .          .     41:

         .          .     42: //get rid of intermediate objects

         .          .     43: //w.Header().Set("Content-Type", "text/html; charset=utf-8")

         .          .     44: //w.Write([]byte(html.String()))

Now, the Error hander takes lots of memory in our main function.

(pprof) list Error

Total: 745.65MB

(pprof) list http.Error

Total: 745.65MB

The system didn’t show me the detail of the error. I assume that the error function cannot be modified anymore since handling an error has nothing to optimize.

Checking the performance again:

362.12MB 48.56% 48.56%   362.12MB 48.56%  runtime.mapassign

   89.51MB 12.00% 60.57%   745.65MB   100%  \_/Users/shuyanli/Desktop/adder.BenchmarkHandleStructAdd

   51.01MB  6.84% 67.41%   635.14MB 85.18%  \_/Users/shuyanli/Desktop/adder.handleStructAdd

      45MB  6.04% 73.44%       45MB  6.04%  html/template.htmlReplacer

   44.50MB  5.97% 79.41%    44.50MB  5.97%  runtime.makemap

   34.50MB  4.63% 84.04%   132.51MB 17.77%  reflect.Value.call

      25MB  3.35% 87.39%       25MB  3.35%  reflect.unsafe\_New

      21MB  2.82% 90.21%   167.51MB 22.46%  text/template.(\*Template).execute

   15.50MB  2.08% 92.29%       24MB  3.22%  reflect.MakeSlice

      13MB  1.74% 94.03%       13MB  1.74%  runtime.stringtoslicebyte

now the mapassign function dominate the whole program. Checking the details of it:

(pprof) list mapassign

Total: 745.65MB

ROUTINE ======================== runtime.mapassign in /usr/local/go/src/runtime/hashmap.go

  362.12MB   362.12MB (flat, cum) 48.56% of Total

         .          .    502:

         .          .    503: alg := t.key.alg

         .          .    504: hash := alg.hash(key, uintptr(h.hash0))

         .          .    505:

         .          .    506: if h.buckets == nil {

  362.12MB   362.12MB    507: h.buckets = newarray(t.bucket, 1)

         .          .    508: }

         .          .    509:

         .          .    510:again:

         .          .    511: bucket := hash & (uintptr(1)<<h.B - 1)

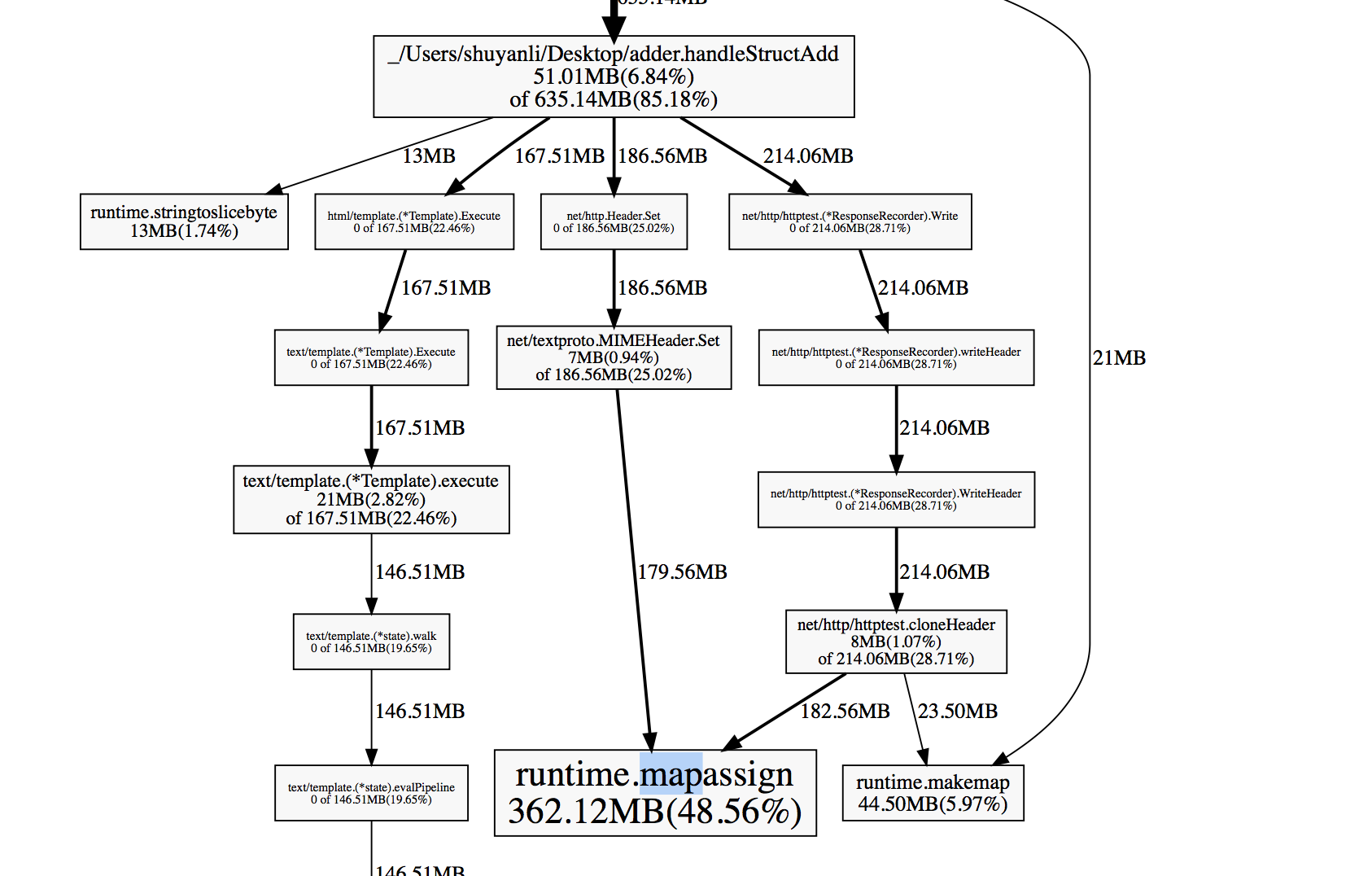
         .          .    512: if h.growing() {

(pprof)

We can actually see that all the memory come from creating a new array.

We need to find out where did we use this function.

Checking the web svg file:



We can see that handleStructAdd function calls this mapassign twice, first in setting the header, another is write the response to header. Which means that anything that calls the header will indirectly call this mapassign function.

MIME is a package we used. The memory is consumed during the map allocation. If we want to modify this, we need to change the data structure of this function. Instead of using map, we can try using slice[] int. We can try writing our own function instead of using package I guess?