After following the tutorial in week4 after the blog, we wanted to play something on our own to see if we can do the improvement for a given go program.

I found a simple web app online. It creates a request on port 8080 and there are two index, /hello and /simple. They all do a pretty simple thing, just print on the screen with “hello world”. The /hello operation records the response of the server so the run time of these two function should be different.

First thing first, I need to configure the path for go since when I was running the code, the system keeps giving me an error “cannot find the package”. As a C++ programmer on windows (just switched to mac), we use IDE like VS. As a result, I didn’t modify the path by my own before.

After reading some article, I set my local GOPATH by type in :

$sudo nano .bash\_profile

it shows me a blank document.

Then I add these in it:

# Setting for go programming language

export GOROOT="/usr/local/go"

export GOPATH="/Users/shuyanli/Desktop/go\_project"

export PATH="/Users/shuyanli/Desktop/go\_project/bin:$PATH"

I set the local path to my desktop and now I can import the local file to the main function.

Lets scan the two “hello world” function.

The simple function is nothing but fprint a string on the screen(guess there is not too much thing we can modify)

The hello function is way more complicate. It wraps the handler and the request with some latency tracking function

The difference of the function is that the hello function can return the states.

For example, if we type in the bash

$ go run main.go -printStats

when we refresh the simple function, nothing will be shown on the screen.

When we refresh the hello function, however, we can see the actual return states:

Starting server on :8080

IncCounter: handler.received.169-231-111-73.hello.Mac-OS.Chrome = 1

RecordTimer: handler.latency.169-231-111-73.hello.Mac-OS.Chrome = 46.884µs

IncCounter: handler.received.169-231-111-73.hello.Mac-OS.Chrome = 1

RecordTimer: handler.latency.169-231-111-73.hello.Mac-OS.Chrome = 31.994µs

IncCounter: handler.received.169-231-111-73.hello.Mac-OS.Chrome = 1

RecordTimer: handler.latency.169-231-111-73.hello.Mac-OS.Chrome = 42.787µs

IncCounter: handler.received.169-231-111-73.hello.Mac-OS.Chrome = 1

RecordTimer: handler.latency.169-231-111-73.hello.Mac-OS.Chrome = 32.152µs

We want to check the load test for the index. We need the tools first:

<https://github.com/adjust/go-wrk>

correction:

<https://github.com/tsliwowicz/go-wrk>

this is the right tool to use since the previous command cannot be found, no matter how I reinstall and recompile

following this link, we got the go-wrk tool, that can check the test load of a url.

Now, we open another command window and type:

$go-wrk -d 5 <http://localhost:8080/hello>

258429 requests in 4.884710008s, 27.85MB read

Requests/sec: 52905.70

Transfer/sec: 5.70MB

Avg Req Time: 189.015µs

Fastest Request: 58.198µs

Slowest Request: 151.564736ms

Number of Errors: 0

We just get the load test result as shown above(52k request/s)

Same for the simple:

Running 5s test @ http://localhost:8080/simple

  10 goroutine(s) running concurrently

275997 requests in 4.874995884s, 29.74MB read

Requests/sec: 56614.82

Transfer/sec: 6.10MB

Avg Req Time: 176.632µs

Fastest Request: 50.034µs

Slowest Request: 13.70035ms

Number of Errors: 0

If we check the heap, we can see how actually the program do the GC

# runtime.MemStats

# Alloc = 843720

# TotalAlloc = 27504491696

# Sys = 14850360

# Lookups = 118

# Mallocs = 289111011

# Frees = 289103025

…

…

# HeapReleased = 4276224

# HeapObjects = 7986

# Stack = 1179648 / 1179648

# MSpan = 46208 / 98304

# MCache = 9600 / 16384

# BuckHashSys = 1452356

# GCSys = 503808

# OtherSys = 5439476

# NextGC = 4194304

# NumGC = 9331

we can se that the number of garbage collection and there is a list for those garbage. According to our previous experience, this part is one of the main part that we can modify.

Result we got finally:

12.61s of 13.71s total (91.98%)

Dropped 153 nodes (cum <= 0.07s)

Showing top 10 nodes out of 79 (cum >= 0.16s)

      flat  flat%   sum%        cum   cum%

     8.74s 63.75% 63.75%      8.79s 64.11%  syscall.Syscall

     1.06s  7.73% 71.48%      1.06s  7.73%  runtime.usleep

     0.76s  5.54% 77.02%      0.76s  5.54%  runtime.kevent

     0.66s  4.81% 81.84%      0.66s  4.81%  runtime.mach\_semaphore\_wait

     0.59s  4.30% 86.14%      0.59s  4.30%  runtime.mach\_semaphore\_signal

     0.56s  4.08% 90.23%      0.56s  4.08%  runtime.freedefer

     0.07s  0.51% 90.74%      0.07s  0.51%  runtime.mach\_semaphore\_timedwait

     0.07s  0.51% 91.25%      0.07s  0.51%  runtime.memclrNoHeapPointers

     0.05s  0.36% 91.61%      0.22s  1.60%  net/http.(\*chunkWriter).writeHeader

     0.05s  0.36% 91.98%      0.16s  1.17%  runtime.mallocgc

Showing top 10 nodes out of 79 (cum >= 0.16s)

      flat  flat%   sum%        cum   cum%

     8.74s 63.75% 63.75%      8.79s 64.11%  syscall.Syscall

     1.06s  7.73% 71.48%      1.06s  7.73%  runtime.usleep

     0.76s  5.54% 77.02%      0.76s  5.54%  runtime.kevent

     0.66s  4.81% 81.84%      0.66s  4.81%  runtime.mach\_semaphore\_wait

     0.59s  4.30% 86.14%      0.59s  4.30%  runtime.mach\_semaphore\_signal

     0.56s  4.08% 90.23%      0.56s  4.08%  runtime.freedefer

     0.07s  0.51% 90.74%      0.07s  0.51%  runtime.mach\_semaphore\_timedwait

     0.07s  0.51% 91.25%      0.07s  0.51%  runtime.memclrNoHeapPointers

     0.05s  0.36% 91.61%      0.22s  1.60%  net/http.(\*chunkWriter).writeHeader

     0.05s  0.36% 91.98%      0.16s  1.17%  runtime.mallocgc

Important note:

When we first run go tool pprof -seconds 5 <http://localhost:8080/debug/pprof/profile> and type in top10, we get a message that the profile is empty. We try run 10 seconds first then do the same step, but we keeps getting the same thing.

After research the next day, we figure out that (thank god) out program is actually run too “fast” so that the profile flag (we imported in the header as import \_ "net/http/pprof", don’t forget this \_ ! ) is not recording anything actually. We try to use the latest profile tools here: <https://github.com/pkg/profile>

But this is not good for a web request.

We followed this stack Overflow answer:

<http://stackoverflow.com/questions/30871691/cant-get-golang-pprof-working>

and finally find out that the only way that we can get the data is use the third terminal window

steps:

1: open window 1 and do $go run main.go

2: open the second window and do $ go-wrk -d 200 <http://localhost:8080/hello>

note that the duration should be longer then 10 seconds, otherwise there is no data in the profile.

3: open the third window simultaneously and type in $ go tool pprof -seconds 5 <http://localhost:8080/debug/pprof/profile> .

only in this way can we record the data and get the result we want to see