Profiling

Profiling

The best way to make your application faster is by improving inefficient code. But how can you tell which code is inefficient?

You've got to measure it! Go has you covered with the built-in `pprof` toolset.

Types Of Profiling

- CPU Profiling
- Memory Profiling
- Blocking Profiling

CPU Profiling

- Most Common type of profiling
- Interrupts every 10ms and records the stack trace of currently running go routines

CPU profiling can be misleading as it doesn't take into account `latency`. This is due to the fact that the profiler is only taking samples of CPU usage of active go routines, and `sleeping` go routines won't be counted in the cpu profile.

Filippo Valsorda has a great video explanation here:

Fighting latency: the CPU profiler is not your ally

Memory Profiling

- Records allocations from stack trace
- By default records 1 in every 1,000 allocations (can be configured)
- Stack allocations are assumed to be free and not tracked
- Can be difficult to get a complete picture because it tracks allocations, and not use

Block Profiling

Block profiling records amount of time a go routine spent waiting for a shared resource (blocked).

Block profiling is typically used once you have feel you have eliminated cpu and memory performance issues.

A go routine is considered to be "blocked" when:

- Sending or receiving on a unbuffered channel.
- Sending to a full channel, receiving from an empty one.
- Trying to obtain a Lock on a sync.Mutex that is already locked. This can be a read or write lock.

One Profile At At Time

Profiling is not free. Profiling has a moderate, but measurable impact on program performance—especially if you increase the memory profile sample rate. Most tools will not stop you from enabling multiple profiles at once. If you enable multiple profiles at the same time, they will observe their own interactions and skew your results.

Do not enable more than one kind of profile at a time.

Rules Of Performance

1. Never guess about performance.
2. Measurements must be relevant.
3. Profile before you decide something is performance critical.
4. Test to know you are correct.

Using `pprof`

The `pprof` tool *should* always be invoked with *two* arguments.

```
$ go tool pprof /path/to/your/binary /path/to/your/profile
```

The "binary" argument *must* be the binary that produced this profile.

The "profile" argument *must* be the profile generated by this binary.

Warning: Because pprof also supports an online mode where it can fetch profiles from a running application over http, the pprof tool can be invoked without the name of your binary (issue 10863):

Do not do this or `pprof` will report your profile is empty:

```
$ go tool pprof /tmp/c.pprof
```

Generating A Profile From Benchmarks

To run a pprof against a benchmark, simply add the `-cpuprofile` flag as an argument.

In this example we will run a profile on a simple benchmark for the strings.Split function.

```
func BenchmarkSplit(b *testing.B) {
    for i := 0; i < b.N; i++ {
        split = strings.Split("one,two,three,four,five,six,seven", ",")
    }
}
```

Ø

Running The Benchmark

```
$ go test -bench=Split -cpuprofile=cpu.pprof
BenchmarkFields_Escape-8         5000000               244 ns/op
PASS
ok      github.com/gopherguides/training/advanced/profiling/src 1.475s
```

In addition to the benchmark output, two new files were created.

- cpu.pprof: The binary file that we will use with `pprof`
- x.test: where x corresponds to the name of the current directory. It's the test binary that runs the benchmarks.

Running The `pprof` Command

`pprof` is an interactive console. Typing `help` at any time will list off many of the things you can do, however, `top` is one of the more useful commands to run.

```
$ go tool pprof src.test cpu.pprof
Entering interactive mode (type "help" for commands)
(pprof) top
1.13s of 1.17s total (96.58%)
Showing top 10 nodes out of 45 (cum >= 0.61s)
      flat  flat%   sum%        cum   cum%
     0.46s 39.32% 39.32%      0.46s 39.32%  runtime.mach_semaphore_signal
     0.28s 23.93% 63.25%      1.10s 94.02%  strings.genSplit
     0.13s 11.11% 74.36%      0.13s 11.11%  runtime.heapBitsSetType
     0.12s 10.26% 84.62%      0.12s 10.26%  runtime.indexbytebody
     0.06s  5.13% 89.74%      0.20s 17.09%  strings.Count
     0.04s  3.42% 93.16%      0.04s  3.42%  runtime.mach_semaphore_wait
     0.01s  0.85% 94.02%      0.01s  0.85%  nanotime
     0.01s  0.85% 94.87%      0.01s  0.85%  runtime.mach_semaphore_timedwait
     0.01s  0.85% 95.73%      0.62s 52.99%  runtime.makeslice
     0.01s  0.85% 96.58%      0.61s 52.14%  runtime.mallocgc
```

Understanding Top

As you can see, 94% of our time was spent in `strings.genSplit`, if we look at the source code you will see it is just a wrapper for `strings.genSplit`:

```
// Split slices s into all substrings separated by sep and returns a slice of
// the substrings between those separators.
// If sep is empty, Split splits after each UTF-8 sequence.
// It is equivalent to SplitN with a count of -1.
func Split(s, sep string) []string { return genSplit(s, sep, 0, -1) }
```

Listing A Call

To investigate further, we can `list` the command:

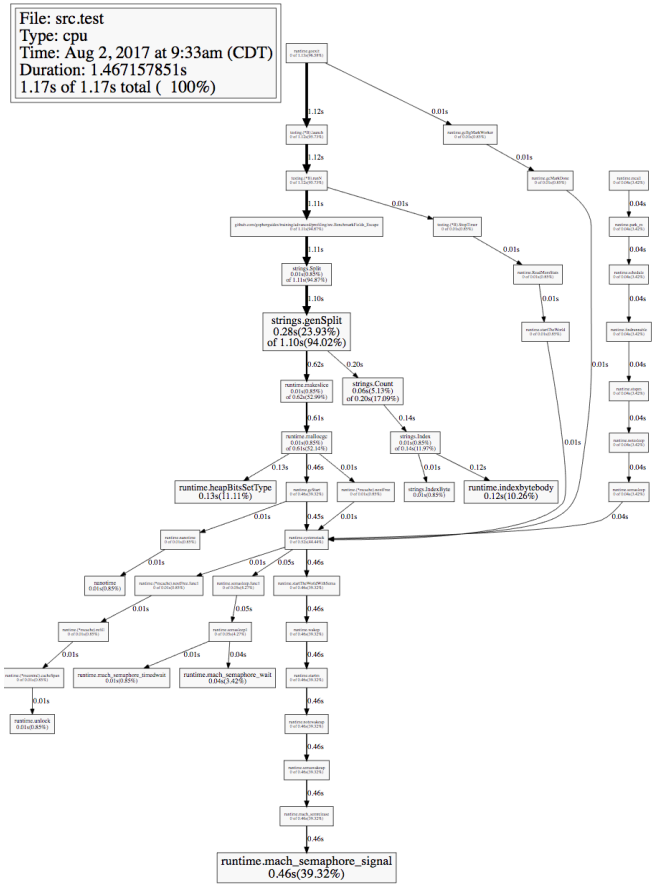
```
(pprof) list strings.genSplit
```

It's a large output, so here are a few lines from it:

```
ROUTINE ======================== strings.genSplit in /usr/local/Cellar/go/1.8.3/libexec/src/strings
     280ms      1.10s (flat, cum) 94.02% of Total
         .          .    228:   return -1
         .          .    229:}
         .          .    230:
         .          .    231:// Generic split: splits after each instance of sep,
         .          .    232:// including sepSave bytes of sep in the subarrays.
      10ms       10ms    233:func genSplit(s, sep string, sepSave, n int) []string {
      10ms       10ms    234:   if n == 0 {
         .          .    235:           return nil
         .          .    236:   }
         .          .    237:   if sep == "" {
         .          .    238:           return explode(s, n)
         .          .    239:   }
         .          .    240:   if n < 0 {
         .      200ms    241:           n = Count(s, sep) + 1
         .          .    242:   }
         .          .    243:   c := sep[0]
         .          .    244:   start := 0
         .      620ms    245:   a := make([]string, n)
         .          .    246:   na := 0
                                                                                                    Ø
```

Using Web

You can also see this in a visual format by using the `web` command. This will open up a browser and show visually where the time was spent.



split svg profile

Web Dependency

Using the web command required Graphviz to be installed on your machine. If you see this error you likely don't have it installed.

```
(pprof) web
Failed to execute dot. Is Graphviz installed? Error: exec: "dot": executable file not found in $PAT
```

Running A Memory Profile (alloc)

The `--alloc_objects` flag tells you how many objects were allocated.

```
$ go test -bench=Split -memprofile=mem.pprof
BenchmarkSplit-8         5000000               239 ns/op
PASS
ok      github.com/gopherguides/training/advanced/profiling/src 1.455s
```

```
$ go tool pprof --alloc_objects src.test mem.pprof
Entering interactive mode (type "help" for commands)
(pprof) top
5997184 of 5997184 total (  100%)
      flat  flat%   sum%        cum   cum%
   5997184   100%   100%    5997184   100%  strings.genSplit
         0     0%   100%    5997184   100%  github.com/gopherguides/training/advanced/profiling/src
         0     0%   100%    5997184   100%  runtime.goexit
         0     0%   100%    5997184   100%  strings.Split
         0     0%   100%    5997184   100%  testing.(*B).launch
         0     0%   100%    5997184   100%  testing.(*B).runN
```

Running A Memory Profile (alloc) (cont.)

```
(pprof) list strings.genSplit
Total: 5997184
ROUTINE ======================== strings.genSplit in /usr/local/Cellar/go/1.8.3/libexec/src/strings
   5997184    5997184 (flat, cum)   100% of Total
         .          .    240:   if n < 0 {
         .          .    241:           n = Count(s, sep) + 1
         .          .    242:   }
         .          .    243:   c := sep[0]
         .          .    244:   start := 0
   5997184    5997184    245:   a := make([]string, n)
         .          .    246:   na := 0
         .          .    247:   for i := 0; i+len(sep) <= len(s) && na+1 < n; i++ {
         .          .    248:           if s[i] == c && (len(sep) == 1 || s[i:i+len(sep)] == sep) {
         .          .    249:                   a[na] = s[start : i+sepSave]
         .          .    250:                   na++
```

Running A Memory Profile (alloc)

The `--inuse_objects` flag tells you how many objects are still in use.

```
$ go tool pprof --inuse_objects src.test mem.pprof
Entering interactive mode (type "help" for commands)
(pprof) top
0 of 0 total (    0%)
      flat  flat%   sum%        cum   cum%
         0     0%     0%          0     0%  github.com/gopherguides/training/advanced/profiling/src
         0     0%     0%          0     0%  runtime.goexit
         0     0%     0%          0     0%  strings.Split
         0     0%     0%          0     0%  strings.genSplit
         0     0%     0%          0     0%  testing.(*B).launch
         0     0%     0%          0     0%  testing.(*B).runN
```

Due to the benchmark being very small, we actually have no measurable in use objects. We will revisit this command later in the module.

Exercise

- Write a benchmark for the strings.ToUpper function.
- Uppercase the following string in the benchmark: `Education is what remains after one has forgotten what one has learned in school.`
- Using the pprof tool, generate a cpu profile
- Use the following commands: `top`, `list`, `web`
- Extra Credit - Name who the quote is from.

Solution

```
$ go test -bench=ToUpper -cpuprofile=cpu.pprof
BenchmarkToUpper-8       2000000               858 ns/op
PASS
ok      github.com/gopherguides/training/advanced/profiling/src 2.582s
```

```
package profiling

import (
    "strings"
    "testing"
)

var upper string

func BenchmarkToUpper(b *testing.B) {
    for i := 0; i < b.N; i++ {
        upper = strings.ToUpper("Education is what remains after one has forgotten what one has lea
    }
}
                                                                                                  Ø
```

Quote is from Albert Einstein

Profiling Live Programs

Using net/http/pprof, we can grab live profiles from running programs via an http endpoint.

You will need to import the package with an underscore (_), since we are importing it for side effects, and not direct use.

```
import _ "net/http/pprof"
```

Once you have the program running, you can visit the http://localhost:8080/debug/pprof/ and explore the dashboard.

Demo App

```
// super simple email regex
const emailRegex = `(\w+)\@(\w+)\.[a-zA-Z]`

func main() {
    http.HandleFunc("/", handler)
    log.Printf("listening on localhost:8080")
    log.Fatal(http.ListenAndServe("localhost:8080", nil))
}

func handler(w http.ResponseWriter, r *http.Request) {
    email := r.URL.Path[1:]

    if isEmail(email) {
        log.Printf("%s looks like an email\n", email)
        fmt.Fprintf(w, "%s looks like an email", email)
        return
    }
    log.Printf("%s doesn't look like an email\n", email)
    fmt.Fprintf(w, "%s doesn't look an email", email)
}

func isEmail(email string) bool {
    re := regexp.MustCompile(emailRegex)
    return re.MatchString(email)
}
                                                                    Ø
```

Grabbing A Live Profile

```
$ go tool pprof -seconds 5 http://localhost:8080/debug/pprof/profile

Fetching profile from http://localhost:8080/debug/pprof/profile?seconds=5
Please wait... (5s)
Saved profile in /Users/corylanou/pprof/pprof.localhost:8080.samples.cpu.001.pb.gz
Entering interactive mode (type "help" for commands)
(pprof)
```

The Profile

```
(pprof) top
profile is empty
```

What happened?

When we were running the profile, there was no actual load on the server, so nothing was recorded

Adding Load

We can add some load to the server with the go-wrk tool.

```
go get github.com/adjust/go-wrk
```

Creating load:

```
go-wrk -d 30 http://localhost:8080/john@smith.com
```

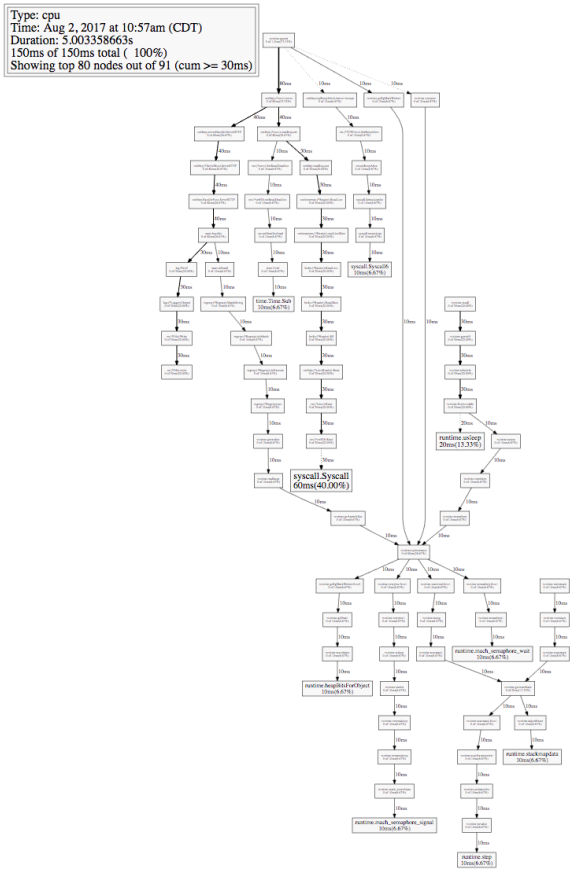
Now, while that is working, we can grab another profile:

```
$  go tool pprof -seconds 5 http://localhost:8080/debug/pprof/profile
```

Let's use the `web` command to look at the results.

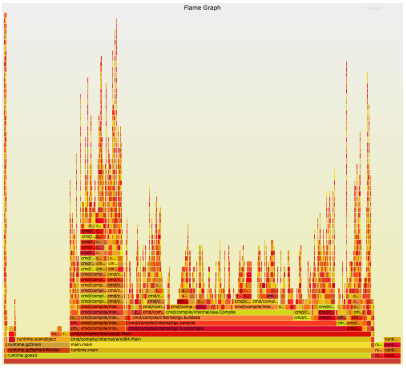
Inspecting Results Via Web

Wow, this is a lot of information, and hard to follow. Let's move on to flame graphs and see if that helps us understand better what is going on.



split svg profile

Flame Graphs

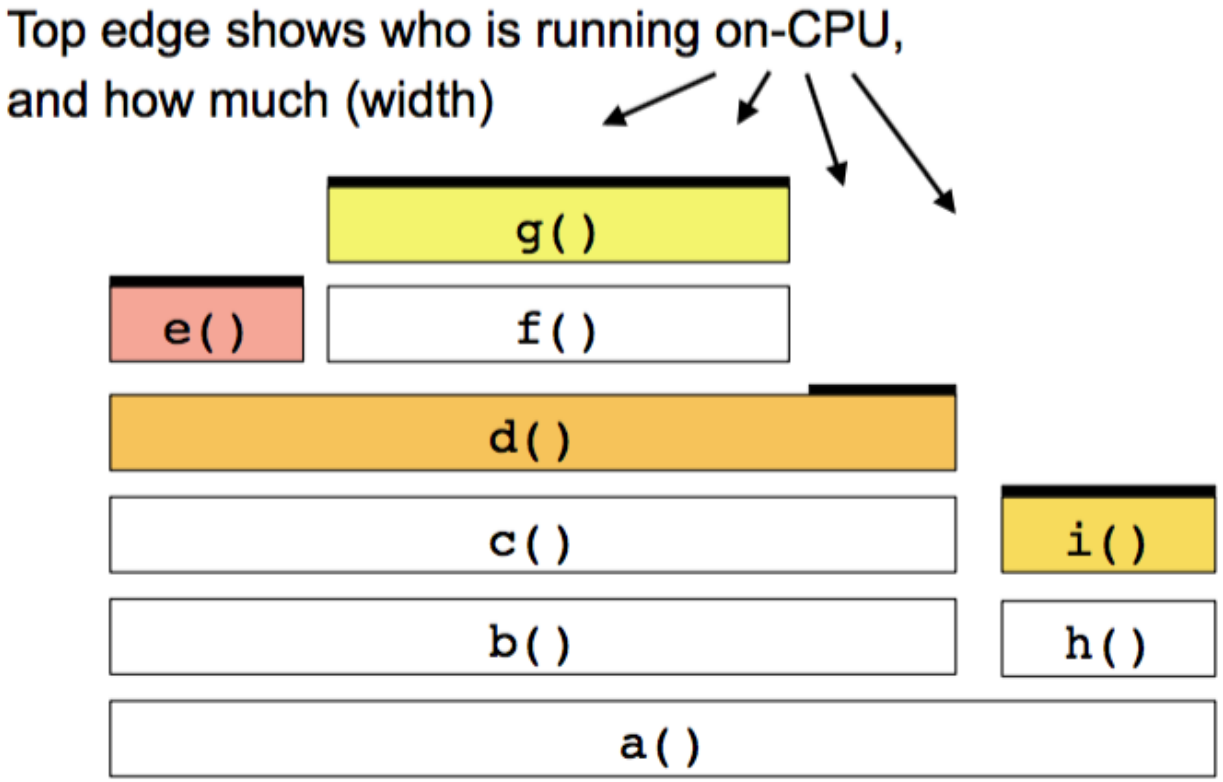


"The x-axis shows the stack profile population, sorted alphabetically (it is not the passage of time), and the y-axis shows stack depth. Each rectangle represents a stack frame. The wider a frame is is, the more often it was present in the stacks. The top edge shows what is on-CPU, and beneath it is its ancestry. The colors are usually not significant, picked randomly to differentiate frames."

- *x axis* alphabetical stack sort, to maximize merging.
- *y axis* stack depth.

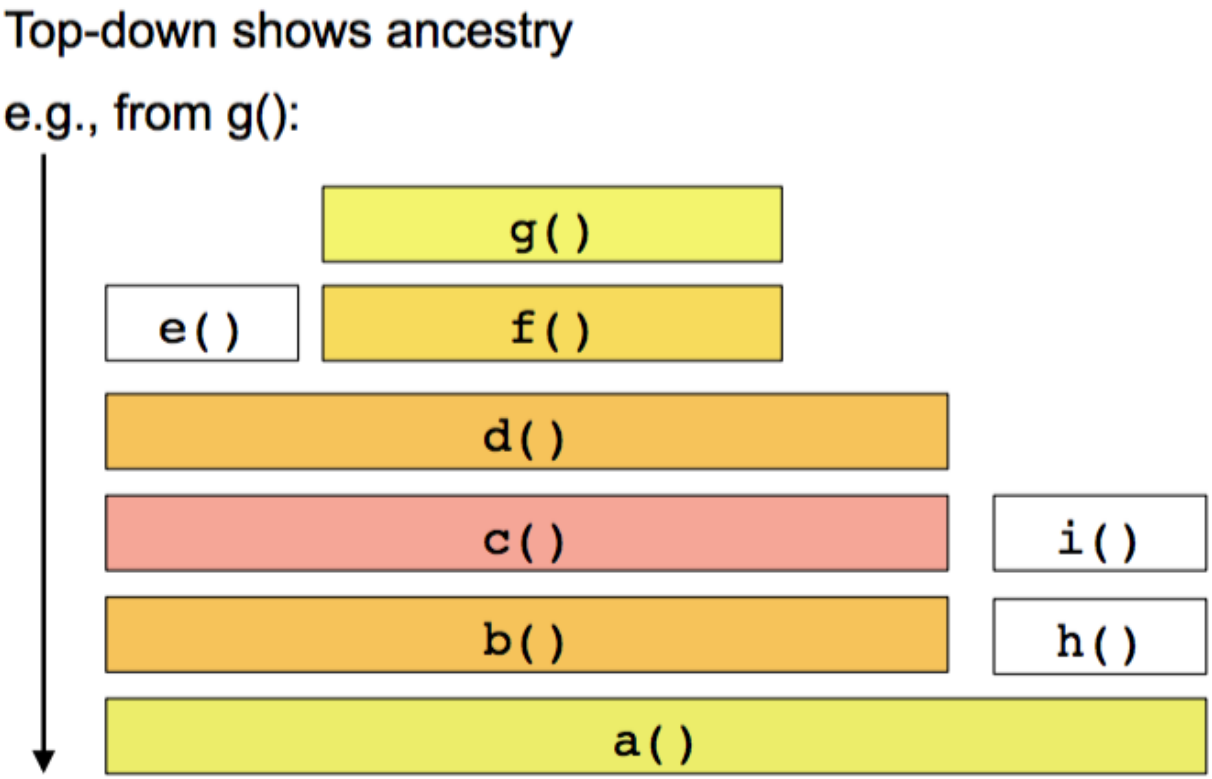
http://www.brendangregg.com/flamegraphs.html

Flame Graph Interpretation (1/3)



http://www.slideshare.net/brendangregg/java-performance-analysis-on-linux-with-flame-graphs

Flame Graph Interpretation (2/3)

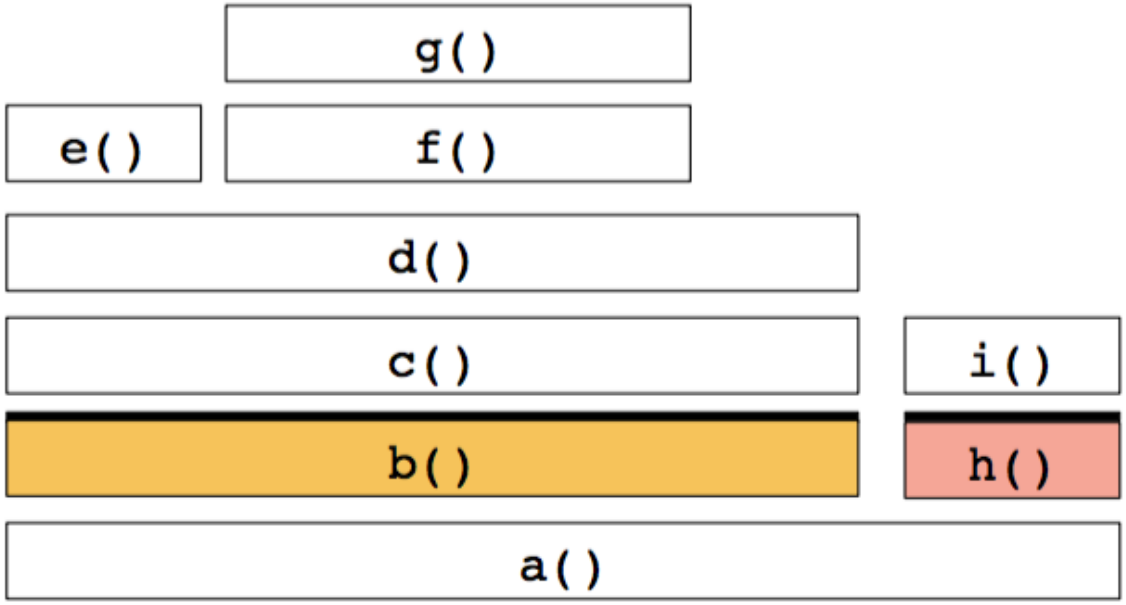


http://www.slideshare.net/brendangregg/java-performance-analysis-on-linux-with-flame-graphs

Flame Graph Interpretation (3/3)

Widths are proportional to presence in samples

e.g., comparing b() to h() (incl. children)



http://www.slideshare.net/brendangregg/java-performance-analysis-on-linux-with-flame-graphs

Using Go-torch

Flame graphs can consume data from many sources, including `pprof` (and `perf(1)`).

Uber has open sourced a tool call `go-torch` which automates the process, assuming you have the `/debug/pprof` endpoint, or you can feed it an existing profile.

https://github.com/uber/go-torch

DEMO

Get needed support scripts

```
$ go get -u -v github.com/uber/go-torch
$ mkdir -p $GOPATH/src/github.com/brendangregg
$ cd $GOPATH/src/github.com/brendangregg
$ git clone https://github.com/brendangregg/FlameGraph.git
$ export PATH=$PATH:$GOPATH/src/github.com/brendangregg/FlameGraph
```

Run the torch command:

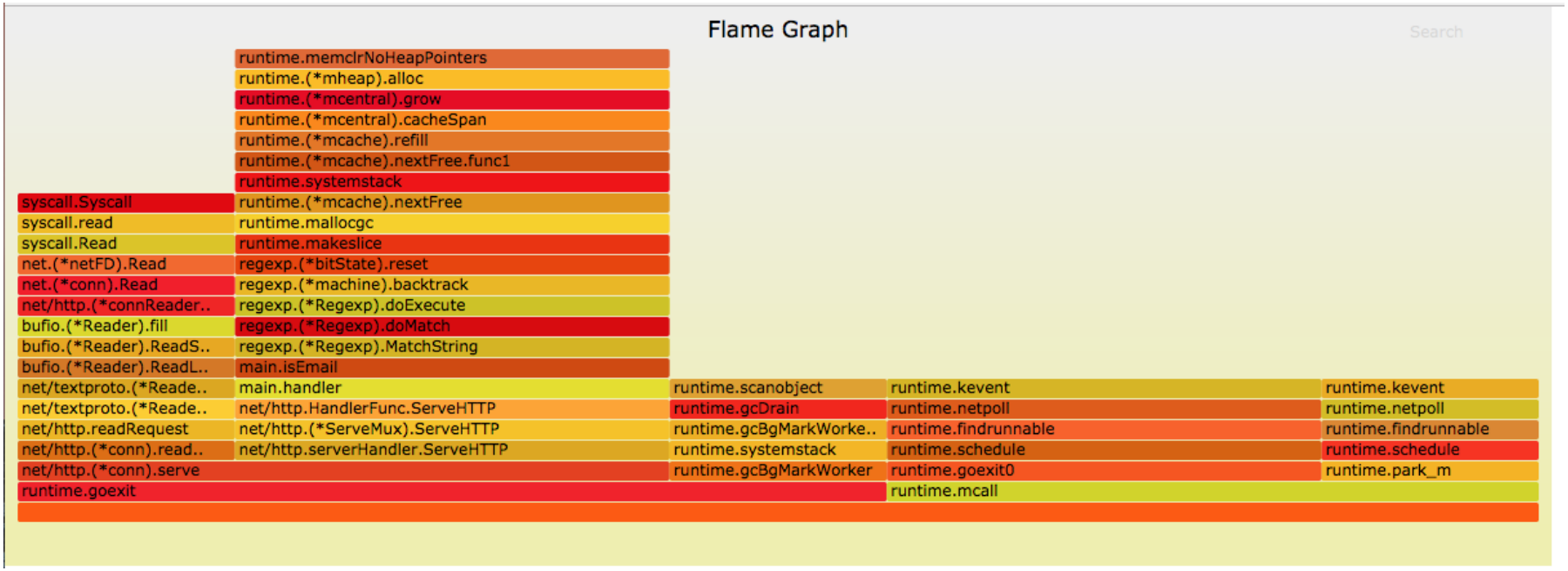
```
$ go-torch -t 5

INFO[11:14:35] Run pprof command: go tool pprof -raw -seconds 5 http://localhost:8080/debug/pprof/p
INFO[11:14:41] Writing svg to torch.svg
```

Inspecting Torch Output

The width represents relative time in a call. The wider the column, the more time spent in that call.

The colors and the order on the graph are random and hold no value other than making it easy to distinguish one call from another.

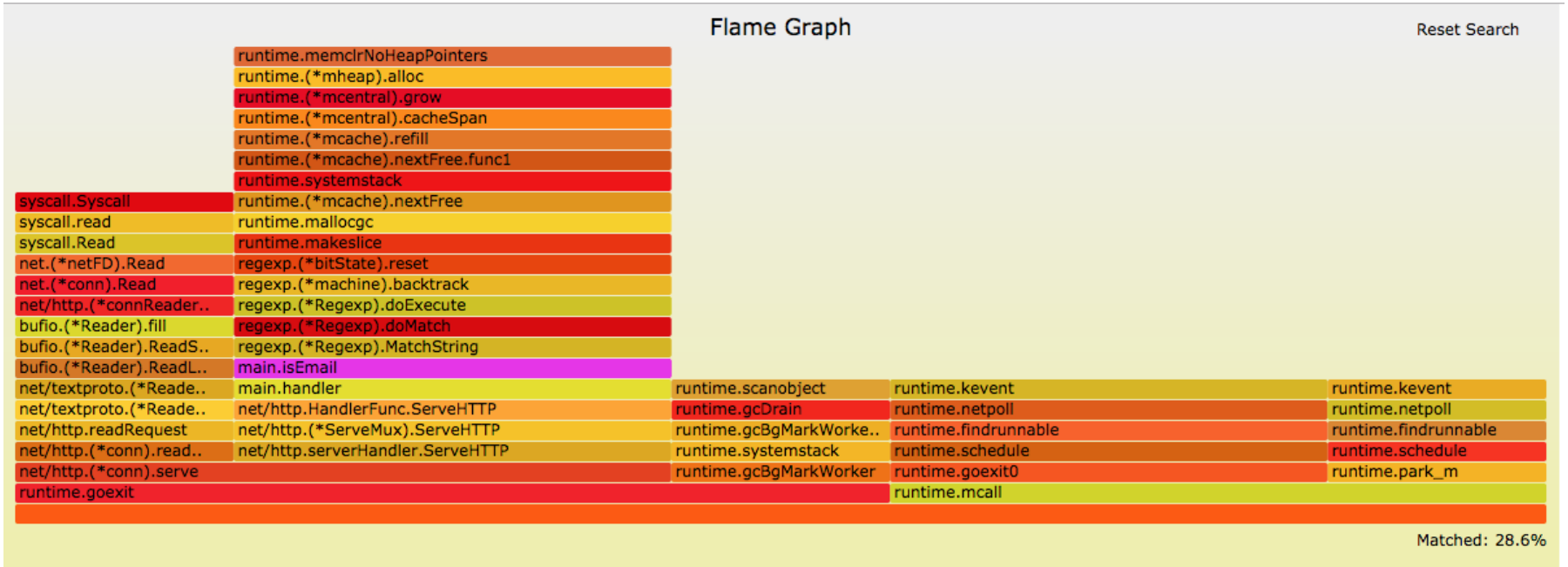


torch svg

Searching Flame Graphs

There is a `search` input box that is very hard to see in the upper right hand side.

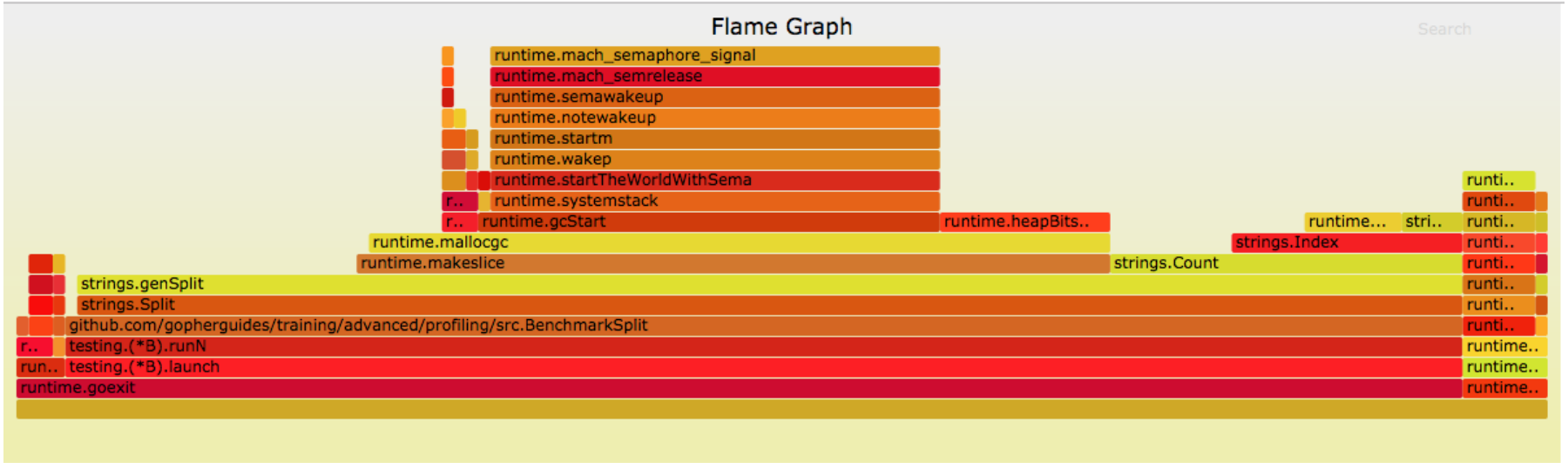
Using it we can search for the `isEmail` function and highlight that part of the flame graph.



Generating From Existing Profiles

We can also use the output from our benchmark profile to generate flame graphs as well.

```
$ go-torch --binaryname src.test -b cpu.pprof
INFO[11:24:37] Run pprof command: go tool pprof -raw src.test cpu.pprof
INFO[11:24:37] Writing svg to torch.svg
```



torch svg

Exercise

- Using the output of the benchmarks on the previous `ToUpper` exercise, generate a flame graph.