The Liquid Software Company

WEB APPLICATION PROFILING 101

Yinon Avraham

ABOUT ME

Yinon Avraham

Software Architect @ JFrog

@yinonavraham

github.com/yinonavraham

orange-coding.blogspot.com

The Liquid Software Company

OVERVIEW

Go's standard library comes with built-in tools to support monitoring and profiling applications.

The goal of this talk is to give an overview and starting point for understanding and using these tools.

Disclaimer-
This talk is not about coding best practices nor security best practices.

AGENDA

- Follow along a demo server application
- See how to enable profiling
- Use the `pprof` tool to profile CPU and memory
- Use the `trace` tool to trace the execution
- Try and improve the demo application
- Expose custom operational information

Note-
- Everything is done only using the standard library
- Available at: `github.com/yinonavraham/go-profiling-demo`

The Liquid Software Company

LET'S GO

DEMO SETUP

- Simple server application in Go
- Single endpoint to download a file: `/file/{filepath}`
- Use a 1MB generated file to download
- Use `wrk` as the benchmarking tool (see: `github.com/wg/wrk`)
- Run benchmarks with 100 threads and 100 connections for 7 seconds

```
wrk -t100 -c100 -d7s http://localhost:8000/file/test-1mb
```

PPROF

- `runtime/pprof` package - for instrumentation
- `net/http/pprof` package - expose `pprof` data over HTTP
- Provides a web page with several built-in profiling information, including:
  - Memory allocations
  - Synchronization points (blocks)
  - Active goroutines
  - Locks (mutex)

PPROF HOW TO

- Register the endpoints, e.g. implicitly:

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

- Browse to the `pprof` web page (by default at):

```
/debug/pprof
```

- Analyze using the `pprof` tool, for example:

```
go tool pprof -http : \
  http://localhost:8000/debug/pprof/profile
```

The Liquid Software Company

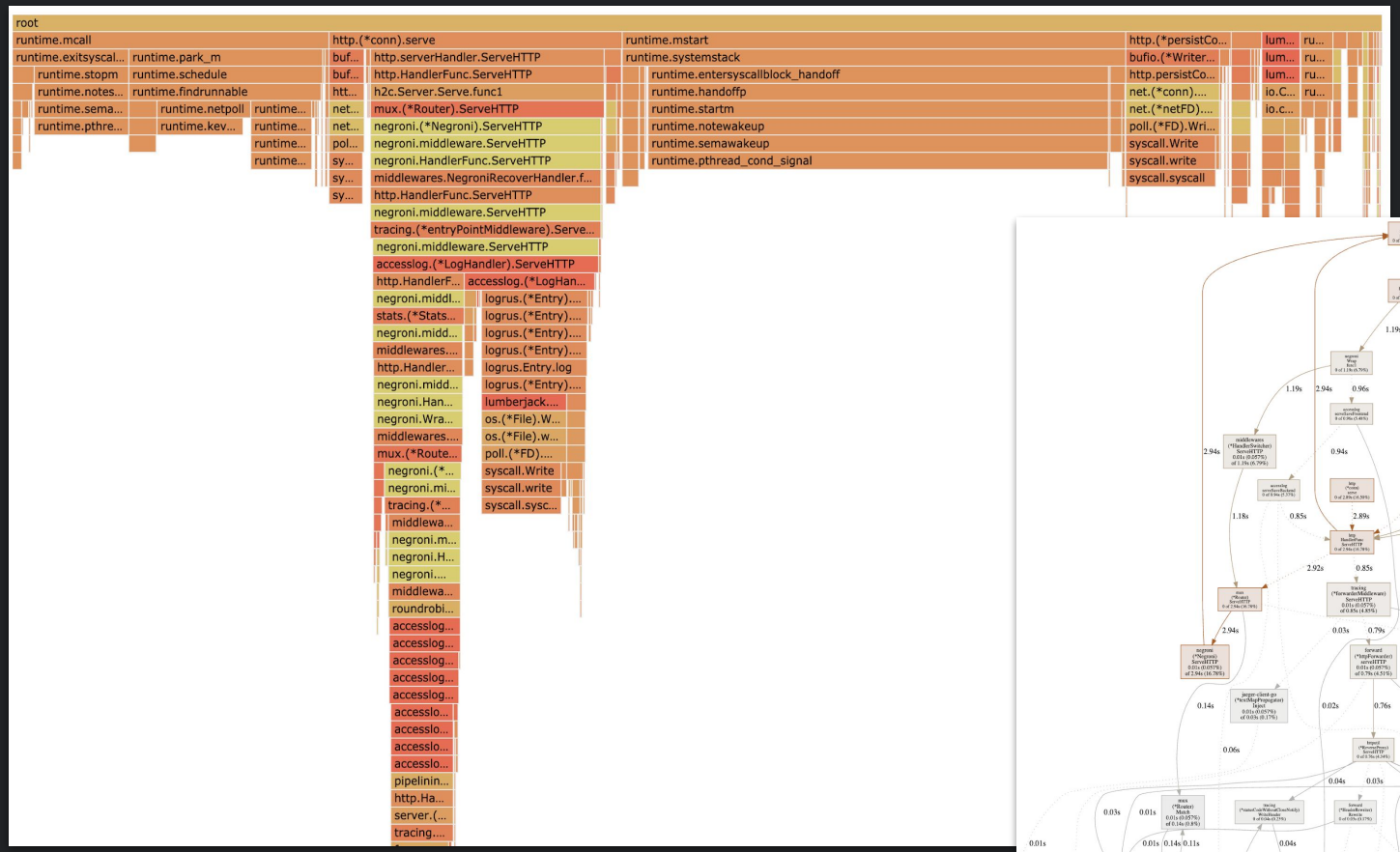
CPU & MEMORY PROFILING

CPU PROFILING

`/debug/pprof/profile`

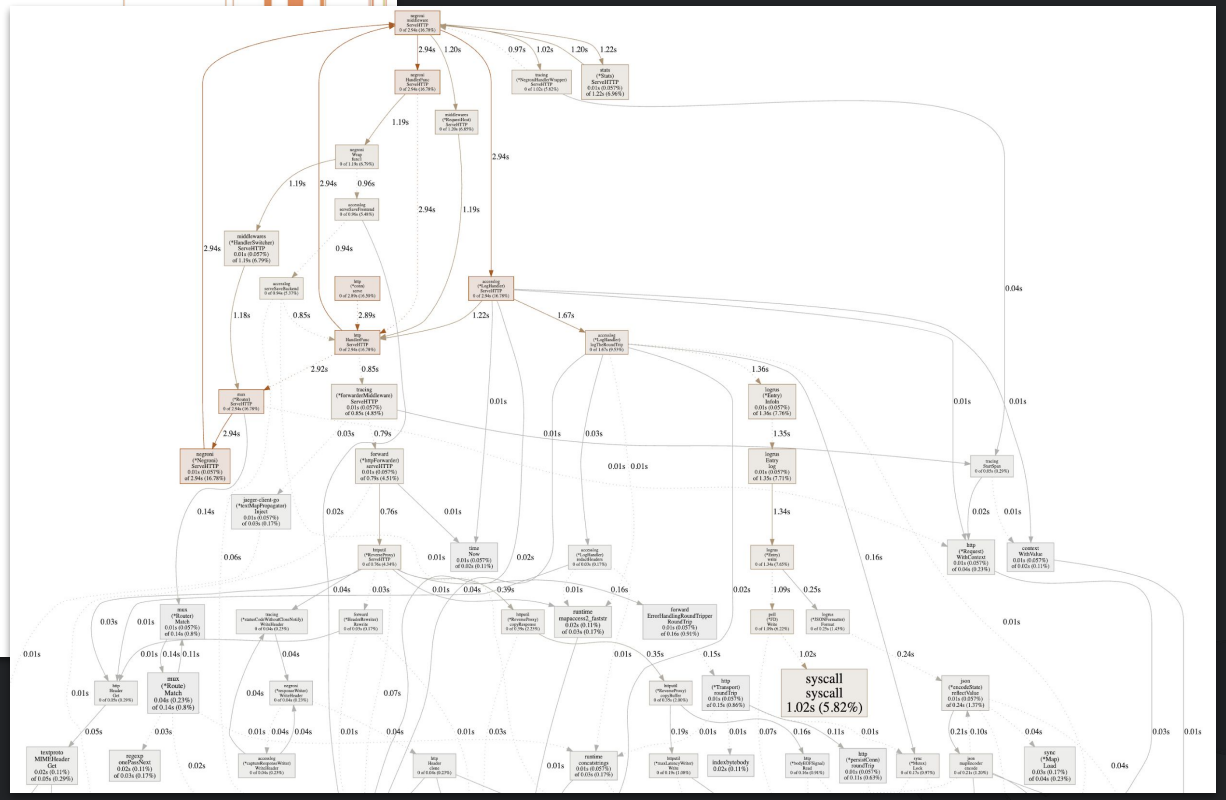
- Provides information on "hot" paths
- Call stack sample is taken every 10 ms (default)
- Sums the CPU time every sampled function spends
- Has some performance impact (non-neglectable), but only on-demand

CPU PROFILE DIAGRAMS



Flame Graph

Call Graph



The Liquid Software Company

MEMORY PROFILING (sampling)

`/debug/pprof/heap`

Memory allocations of live objects

`/debug/pprof/allocs`

All past memory allocations

- Collected by sampling, based on the GC information
- Helps to identify suspects for GC exhaustion
- Use the pprof tool to analyze

GOROUTINES STACK TRACES

`/debug/pprof/goroutine`

All current goroutines

`/debug/pprof/threadcreate`

Goroutines which led to the creation of new OS threads

`/debug/pprof/block`

Goroutines blocking on synchronization primitives (inc. channels)

`/debug/pprof/mutex`

Goroutines which are holders of contended mutexes

The Liquid Software Company

EXECUTION TRACING

TRACE TOOL

- Used to trace the execution of a running application
- Provides visual information on:
  - Goroutines Scheduling
  - CPU Utilization
  - Heap Memory Allocation
  - GC

TRACE HOW TO

1. Collect trace information for e.g. 5 seconds:

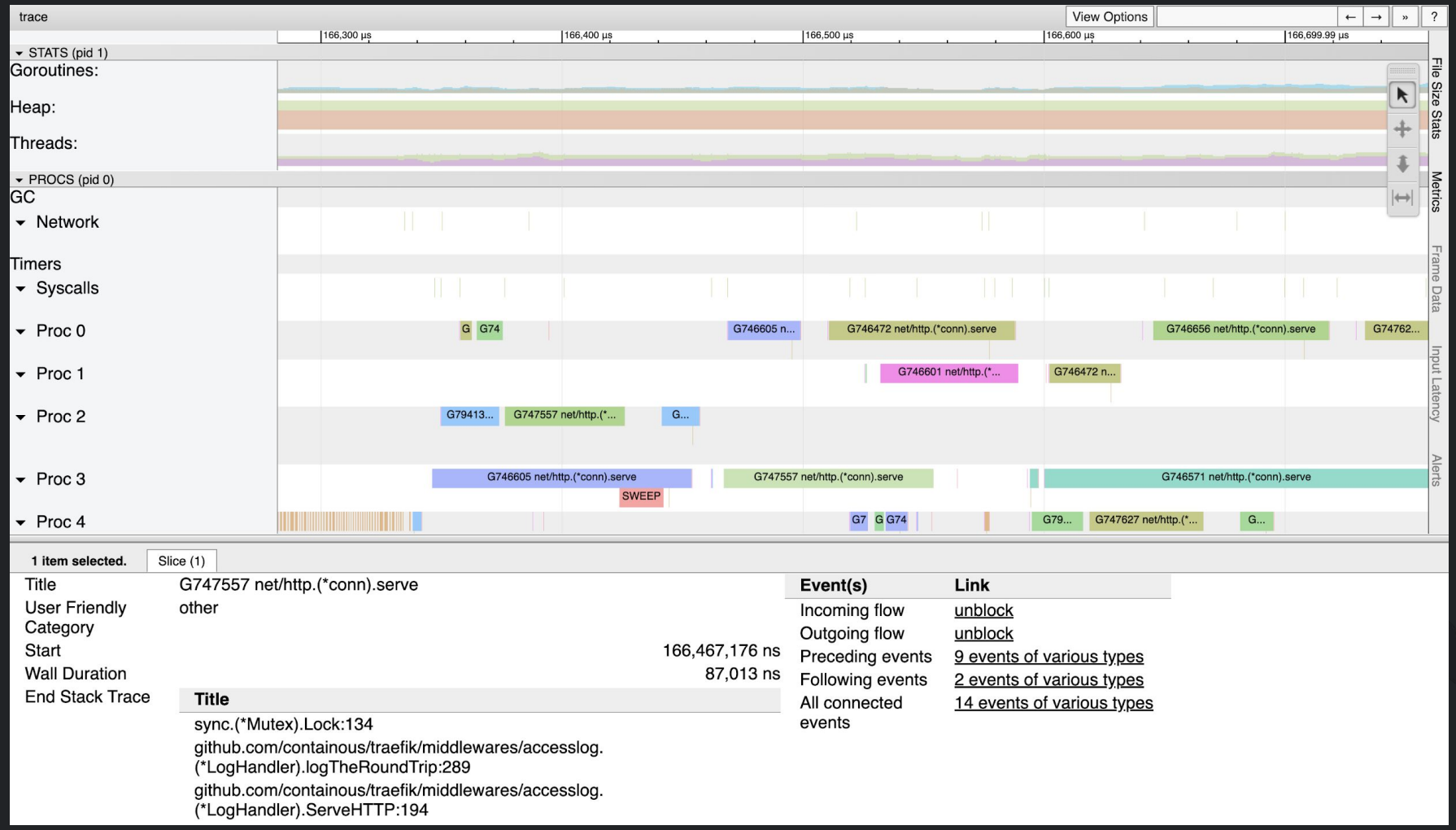
   ```
   GET /debug/pprof/trace?seconds=5
   ```

   And save the output to a file (e.g. using: `curl <url> -o trace.out`)

2. Use the `trace` tool to open a web browser:

   ```
   go tool trace trace.out
   ```

TRACE EXAMPLE



The Liquid Software Company

The Liquid Software Company

CUSTOM PROFILES

CUSTOM PROFILES

- Anyone can add custom defined profiles
- Usually used to track resources and identify leaks
- There are some requirements from the managed resource - read the `runtime/pprof` package's GoDoc

CUSTOM PROFILES HOW TO

```
import "runtime/pprof"

var myProfile = pprof.NewProfile("my.profile")

func New() *Resource {
   r := &Resource{}
   myProfile.Add(r, 1)
   return r
}

func (r *Resource) Close() {
   myProfile.Remove(r)
}
```

The Liquid Software Company

The Liquid Software Company

EXPOSE OPERATIONAL INFORMATION

EXPVAR PACKAGE

- Provides information on exposed application variables
- Predefined variables: command line, memory statistics
- Supports adding custom variables
- Useful for monitoring operational information of a running application

EXPVAR HOW TO

```
import "expvar" // omitted other required imports
var calls expvar.Int
func main() {
   expvar.Publish("hello.calls", &calls)
   http.HandleFunc("/hello", ServeHello)
   log.Fatal(http.ListenAndServe(":7777", nil))
}
func ServeHello(w http.ResponseWriter, req *http.Request) {
   calls.Add(1)
   fmt.Fprintf(w, "Hello JFrog!")
}
```

The Liquid Software Company

EXPVAR EXAMPLE

```
GET /debug/vars
  {
    "cmdline": ["/path/to/myapp"],
    "memstats": {
      "Alloc": 7891752,
      "PauseTotalNs": 316582784,
      "NumGC": 670,

      ...
    },
    "hello.calls": 4
  }
```

The Liquid Software Company

The Liquid Software Company

SUMMARY TIME

CONCLUSION

- Make the `debug` endpoints (`vars`, `pprof`, etc.) available at runtime
- Use the `expvar` package to expose applicative information and metrics
- **DON'T** use `net/http` package's `DefaultServeMux`, create one your own explicitly add debug endpoints
- **MUST** restrict access to the `debug` endpoints

  e.g. put behind authorization, or use non-public IP & port

The Liquid Software Company

THANK YOU!