● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#57

PERFORMANCE PROFILING WITH PPROF

PEERING INTO THE "ENGINE ROOM" OF YOUR GO APPLICATION

The pprof Philosophy

Stop Guessing, Start Measuring

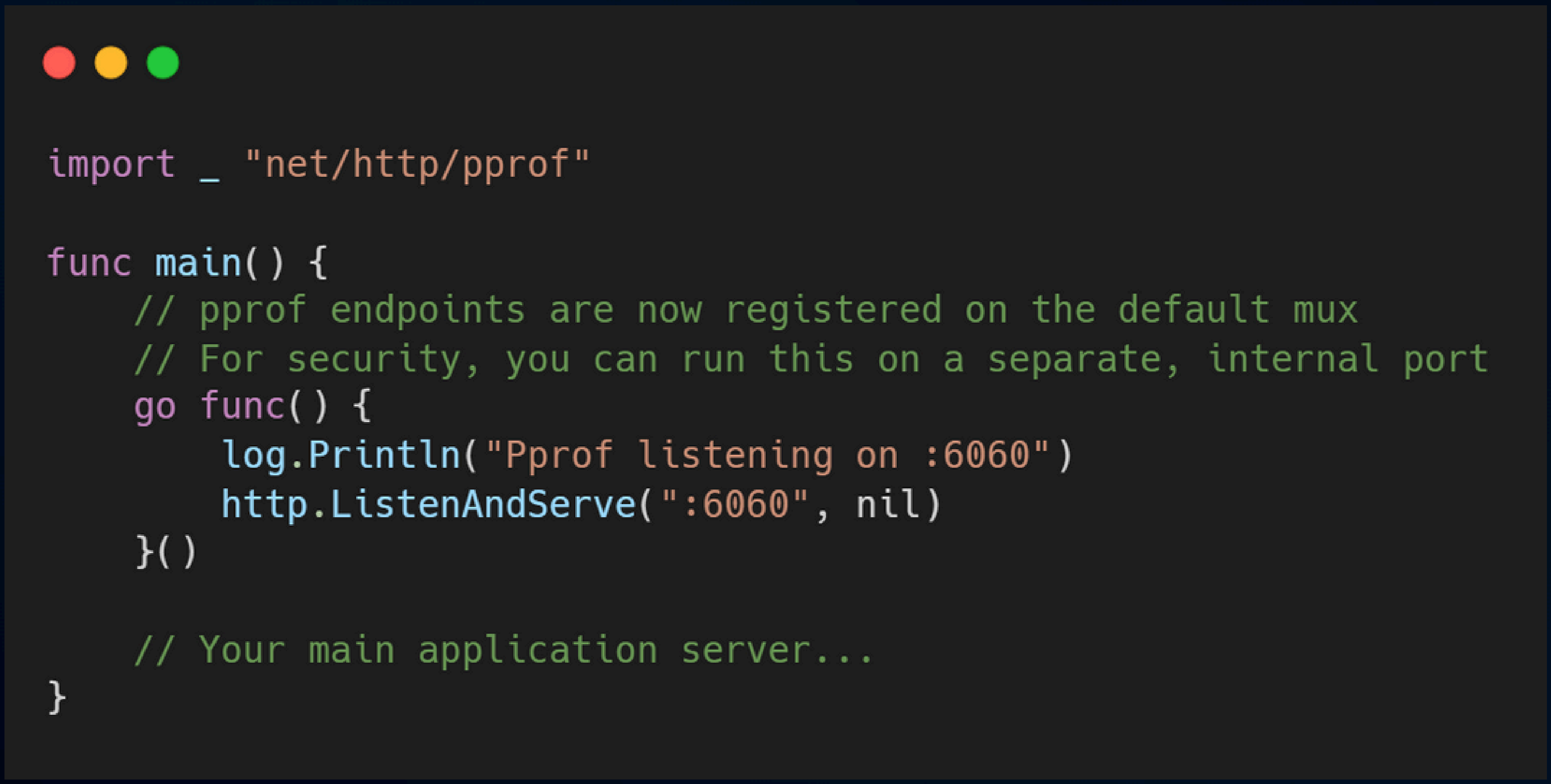
Premature optimization is the root of all evil. Before you rewrite a single line of code to make it "faster", you must profile it. Go makes this easy by including *net/http/pprof* in the standard library, allowing you to inspect a live running service without stopping it.

The Strategy:
Profiling in Go is "Sampling-based". It doesn't track every single instruction (which would be too slow); instead, it pauses the program hundreds of times per second to see what it's doing. This gives you a statistically accurate map of performance with minimal overhead.

Enabling the Profiler

Exposing the Endpoints

In a web service, enabling *pprof* is as simple as a single blank import. This automatically registers several hidden routes under */debug/pprof/*. In production, you should keep these endpoints protected or internal, but always accessible.



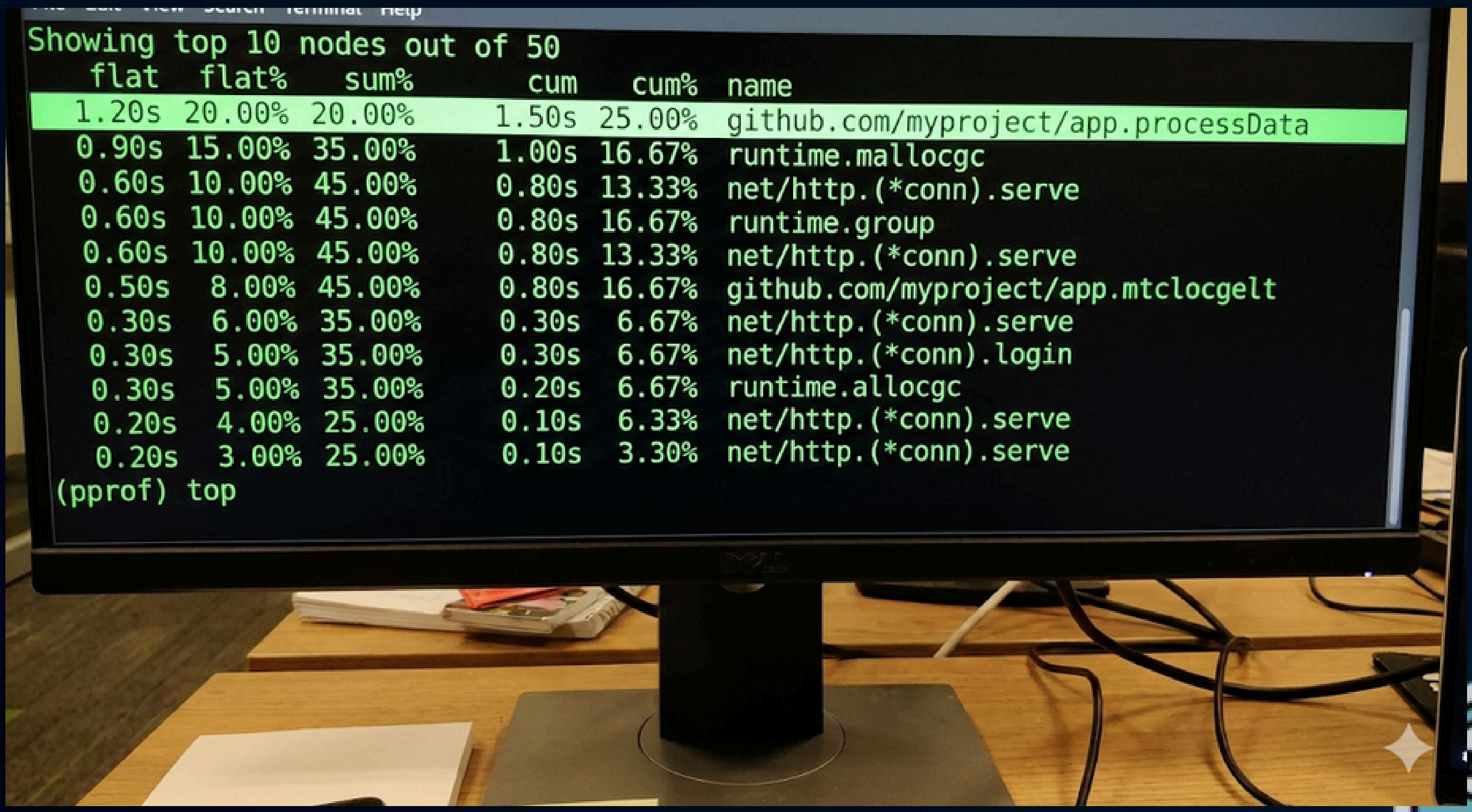
```go
import _ "net/http/pprof"

func main() {
    // pprof endpoints are now registered on the default mux
    // For security, you can run this on a separate, internal port
    go func() {
        log.Println("Pprof listening on :6060")
        http.ListenAndServe(":6060", nil)
    }()

    // Your main application server...
}
```

Capturing a CPU Profile

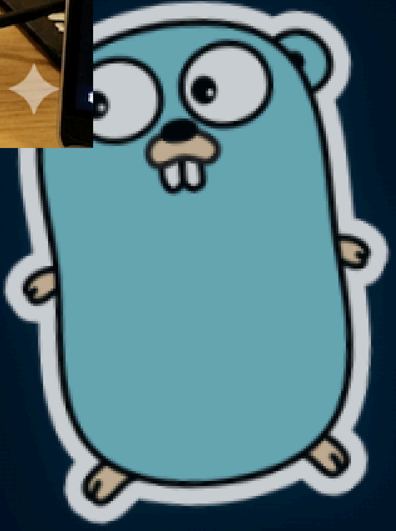
Where is the Time Going?

The CPU profile is your best friend for finding "hot" functions. You can use the *go tool pprof* command to capture a window of your app's life. It records the call stack every time the CPU is busy, helping you identify functions that are doing too much math, string manipulation, or JSON parsing.

```
# Capture 30 seconds of CPU data from a running app
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30
```



```
Showing top 10 nodes out of 50
      flat  flat%   sum%        cum   cum%  name
     1.20s 20.00% 20.00%      1.50s 25.00%  github.com/myproject/app.processData
     0.90s 15.00% 35.00%      1.00s 16.67%  runtime.mallocgc
     0.60s 10.00% 45.00%      0.80s 13.33%  net/http.(*conn).serve
     0.60s 10.00% 45.00%      0.80s 16.67%  runtime.group
     0.60s 10.00% 45.00%      0.80s 13.33%  net/http.(*conn).serve
     0.50s  8.00% 45.00%      0.80s 16.67%  github.com/myproject/app.mtclocgelt
     0.30s  6.00% 35.00%      0.30s  6.67%  net/http.(*conn).serve
     0.30s  5.00% 35.00%      0.30s  6.67%  net/http.(*conn).login
     0.30s  5.00% 35.00%      0.20s  6.67%  runtime.allocgc
     0.20s  4.00% 25.00%      0.10s  6.33%  net/http.(*conn).serve
     0.20s  3.00% 25.00%      0.10s  3.30%  net/http.(*conn).serve
(pprof) top
```



Memory Profiling (Heap)

Tracking Allocations and Leaks

High memory usage in Go usually comes from one of two things: keeping objects alive longer than needed (leaks) or creating too many temporary objects (allocation pressure). The Heap profile shows you exactly which lines of code are allocating the most memory.

Seniors look at "In-use memory" to find leaks, but they look at "Allocated memory" to find performance bottlenecks. Even if memory is being garbage collected, high allocation rates force the GC to run more often, slowing down the whole system.

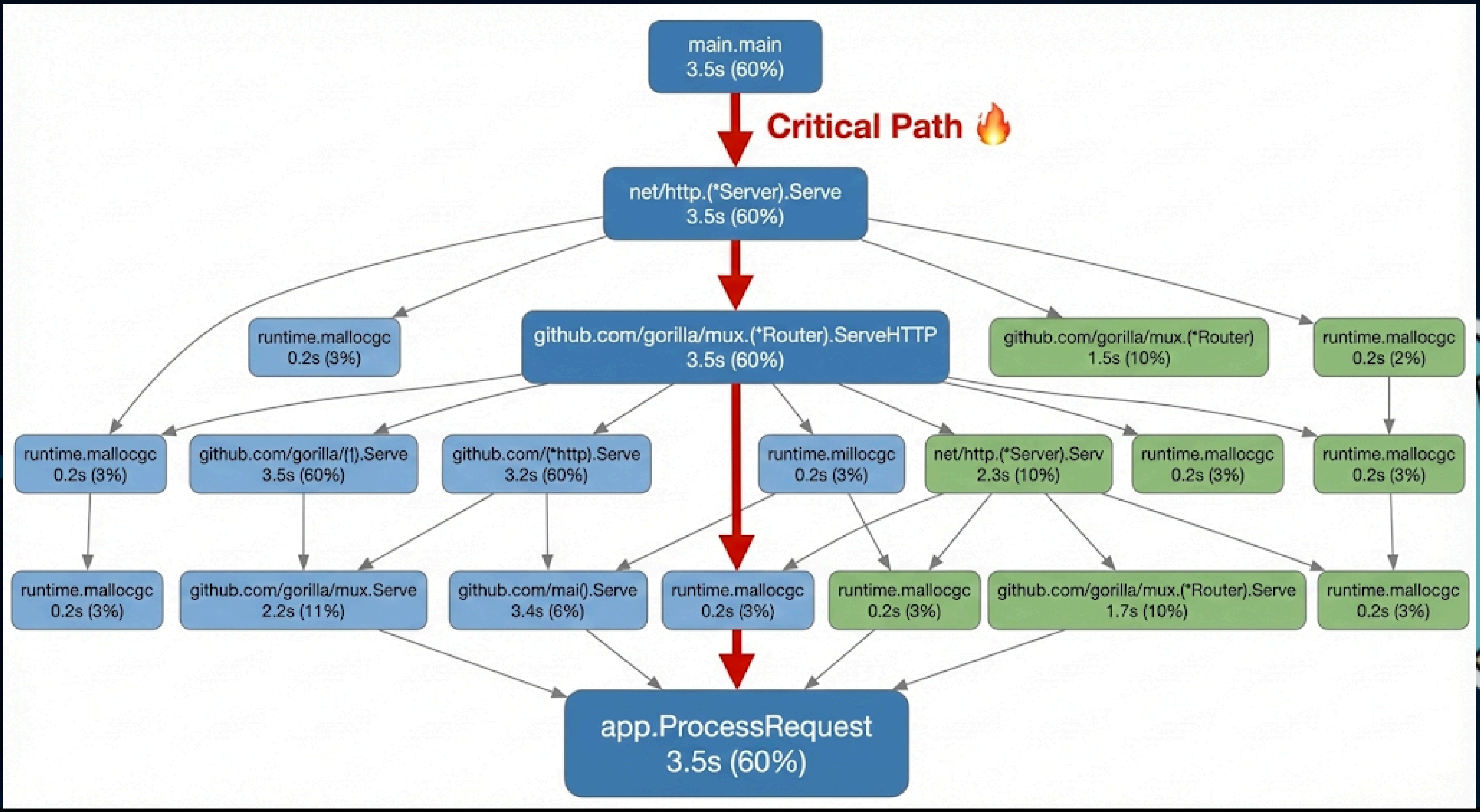
Visualizing with the Web UI

The Interactive Graph

Reading text output in a terminal is hard. The *-http* flag in *pprof* launches a local web server that generates an interactive "Call Graph". Larger boxes represent functions that consume more resources, and thicker arrows show the most common execution paths.

The Strategy:
Follow the "Red Arrows". These represent the "Critical Path" of your application. If you can shrink the boxes on this path, you get the biggest performance ROI for your effort.

```
# Launch the interactive web UI using the captured profile
go tool pprof -http=:8080 cpu.pb.gz
```



Finding Goroutine Leaks

The Goroutine Dump

Is your service slowly getting slower? You might have a goroutine leak. If you start a goroutine that waits on a channel that is never closed, that goroutine stays in memory forever.

The Fix:
Check the */debug/pprof/goroutine?debug=1* endpoint. If you see thousands of goroutines stuck in the same function, you’ve found your leak. A healthy app should have a stable number of goroutines that correlates with its traffic.

Block and Mutex Profiling

Solving Concurrency Bottlenecks

Sometimes your CPU usage is low, but your app is still slow. This is often "Contention." The Block and Mutex profiles show you where goroutines are waiting for a lock or a channel.

The Strategy:
If you see high contention on a Mutex, it's time to reduce the time the lock is held, use a *sync.RWMutex* for many readers, or shard your data so multiple goroutines aren't fighting for the same lock.

```go
// You must enable these manually in your code to collect data
runtime.SetBlockProfileRate(1)
runtime.SetMutexProfileFraction(1)
```

Summary:

- **pprof**: Use it to visualize CPU, Memory, and Concurrency.
- **Standard Lib**: It’s built-in; no need for heavy external agents.
- **Heap vs. CPU**: Use Heap to find leaks; use CPU to find slowness.
- **Visuals**: Use the *-http* flag to see the call graph and flame graphs.

Question: When your app is slow, do you reach for the logs first or the profiler? Why? 👇