Making your Go go go Faster

Bryan Boreham, Director of Engineering, Weaveworks

@bboreham

Weaveworks
What does Weave do?

Weave lets devops iterate faster with:

- observability & monitoring
- continuous delivery
- container networks & firewalls

Kubernetes is our #1 platform

Deep Integration with Kubernetes
(Other orchestrators coming soon!)
Hi, I’m Bryan Boreham

At Weaveworks, I work on system visualisation, observability & monitoring, CI/CD
I also contribute to Container Network Interface, Kubernetes, Prometheus

Program optimisation is my video-game.
Who is working with...

• Go
• Prometheus
• Weaveworks
What I will cover

• **How** to drill into the perf of your Go code
• **When** to look at the perf of your Go code
• Some **patterns** to look out for
• Things that matter more than you might **think**
The three most important things in software optimisation
Measure,
Measure,
Measure.
Measure big things
Measure all the time
Measure in detail

![Jaeger UI Screenshot]

**authfe: GET /api/prom/api/v1/query_range**

Trace Start: December 21, 2017 9:56 PM  
Duration: 47.48s  
Services: 3  
Depth: 6  
Total Spans: 19

<table>
<thead>
<tr>
<th>Service &amp; Operation</th>
<th>0ms</th>
<th>11.87s</th>
<th>23.74s</th>
<th>35.61s</th>
</tr>
</thead>
<tbody>
<tr>
<td>authfe: GET /api/prom/api/v1/query_range</td>
<td>2.6ms</td>
<td>2.08ms</td>
<td>2.07ms</td>
<td>2.07ms</td>
</tr>
<tr>
<td>users-api /users.Users/lookupUsingToken</td>
<td>3.04s</td>
<td>816.75ms</td>
<td>845.07ms</td>
<td>681.89ms</td>
</tr>
<tr>
<td>quierer /grpc.HTTP/Handle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>quierer getMetricNameChunks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>quierer QueryPages</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>quierer DynamoDB.Query</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>quierer DynamoDB.Update</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jaeger - Distributed tracing https://jaegertracing.io
OK, now Profiling

Basic instructions: [http://blog.golang.org/profiling-go-programs](http://blog.golang.org/profiling-go-programs)

$ go test -cpuprofile=cpu.out
$ go tool pprof cpu.out

import _ "net/http/pprof"
What’s going on here?

```
$ go tool pprof -top -cum cpu.out

<table>
<thead>
<tr>
<th>flat</th>
<th>flat%</th>
<th>sum%</th>
<th>cum</th>
<th>cum%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>13.12s</td>
<td>65.67%</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0.4%</td>
<td>5.85s</td>
<td>29.28%</td>
</tr>
<tr>
<td>0.61s</td>
<td>3.05%</td>
<td>3.45%</td>
<td>4.89s</td>
<td>24.47%</td>
</tr>
<tr>
<td>0.46s</td>
<td>2.30%</td>
<td>5.76%</td>
<td>4.66s</td>
<td>23.32%</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>5.76%</td>
<td>4.63s</td>
<td>23.17%</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>5.76%</td>
<td>4.53s</td>
<td>22.67%</td>
</tr>
<tr>
<td>1.46s</td>
<td>7.31%</td>
<td>13.06%</td>
<td>3.58s</td>
<td>17.92%</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>13.06%</td>
<td>3.38s</td>
<td>16.92%</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>13.06%</td>
<td>3.22s</td>
<td>16.12%</td>
</tr>
<tr>
<td>0.02s</td>
<td>0.1%</td>
<td>13.16%</td>
<td>3.22s</td>
<td>16.12%</td>
</tr>
</tbody>
</table>
```

This output shows CPU profiling data using the `go tool pprof` command with the `-top` and `-cum` flags. The table lists the top functions by cumulative time percentage, along with their flat time percentages, cumulative times, and cumulative percentage of total time. The functions are from packages such as `weaveworks/cortex/pkg/querier` and `weaveworks/cortex/pkg/chunk`. The functions include methods like `Query`, `chunksToMatrix`, `mallocgc`, `Samples`, `FetchChunkData`, `Decode`, `scanobject`, `gcBgMarkWorker`, `gcBgMarkWorker.func2`, and `gcDrain`. The percentages and times indicate which functions are the most time-consuming in the program.
```plaintext
$ go tool pprof -top -cum cpu.out

<table>
<thead>
<tr>
<th>flat</th>
<th>flat%</th>
<th>sum%</th>
<th>cum</th>
<th>cum%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>13.12s</td>
<td>65.67%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>weaveworks/cortex/pkg/querier.(*chunkQuerier).Query</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0.4%</td>
<td>5.85s</td>
<td>29.28%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>weaveworks/cortex/pkg/chunk.chunksToMatrix</td>
<td></td>
</tr>
<tr>
<td>0.61s</td>
<td>3.05%</td>
<td>3.45%</td>
<td>4.89s</td>
<td>24.47%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>runtime.mallocgc</td>
<td></td>
</tr>
<tr>
<td>0.46s</td>
<td>2.30%</td>
<td>5.76%</td>
<td>4.66s</td>
<td>23.32%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>weaveworks/cortex/pkg/chunk.(*Chunk).Samples</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>5.76%</td>
<td>4.63s</td>
<td>23.17%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>weaveworks/cortex/pkg/chunk.(*Cache).FetchChunkData</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>5.76%</td>
<td>4.53s</td>
<td>22.67%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>weaveworks/cortex/pkg/chunk.(*Chunk).Decode</td>
<td></td>
</tr>
<tr>
<td>1.46s</td>
<td>7.31%</td>
<td>13.06%</td>
<td>3.58s</td>
<td>17.92%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>runtime.scanobject</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>13.06%</td>
<td>3.38s</td>
<td>16.92%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>runtime.gcBgMarkWorker</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>13.06%</td>
<td>3.22s</td>
<td>16.12%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>runtime.gcBgMarkWorker.func2</td>
<td></td>
</tr>
<tr>
<td>0.02s</td>
<td>0.1%</td>
<td>13.16%</td>
<td>3.22s</td>
<td>16.12%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>runtime.gcDrain</td>
<td></td>
</tr>
</tbody>
</table>
```
Garbage Collection, visualised

go_memstats_heap_alloc_bytes{job="kubernetes-apiservers"}
CPU memory architecture

(Not to scale)
Caches in use

CPU

Logic

Level 1 Cache

Level 2 Cache

RAM

RAM
After GC has run
Memory Profile

```none
$ go tool pprof -alloc_objects -top -cum mem.profile

<table>
<thead>
<tr>
<th></th>
<th>Flat</th>
<th>Flat%</th>
<th>Cum</th>
<th>Cum%</th>
</tr>
</thead>
<tbody>
<tr>
<td>.../cortex/pkg/chunk.(*Store).Get</td>
<td>0</td>
<td>0%</td>
<td>85063816</td>
<td>80.84%</td>
</tr>
<tr>
<td>.../cortex/chunk.(*Chunk).ExternalKey</td>
<td>45679033</td>
<td>43%</td>
<td>61078016</td>
<td>58.05%</td>
</tr>
<tr>
<td>.../cortex/chunk.ByKey.Less</td>
<td>0</td>
<td>0%</td>
<td>56523148</td>
<td>53.72%</td>
</tr>
<tr>
<td>sort.Sort</td>
<td>0</td>
<td>0%</td>
<td>56523148</td>
<td>53.72%</td>
</tr>
<tr>
<td>.../cortex/chunk.(*Chunk).Decode</td>
<td>1818786</td>
<td>1.7%</td>
<td>22562147</td>
<td>21.44%</td>
</tr>
<tr>
<td>encoding/json.(*decodeState).unmarshal</td>
<td>0</td>
<td>0%</td>
<td>19784133</td>
<td>18.80%</td>
</tr>
<tr>
<td>encoding/json.(*decodeState).object</td>
<td>3227746</td>
<td>3.1%</td>
<td>19456448</td>
<td>18.49%</td>
</tr>
<tr>
<td>fmt.Sprintf</td>
<td>15398983</td>
<td>14.6%</td>
<td>15401714</td>
<td>14.64%</td>
</tr>
</tbody>
</table>
```

...
Memory profile options

- `-inuse_space` - bytes allocated but not freed
- `-inuse_objects` - count of objects allocated but not freed
- `-alloc_space` - bytes allocated, including those freed
- `-alloc_objects` - count of objects allocated
- `-memprofilerate` - how often samples are taken
Avoidance strategies

- Reuse
- Reduce
- Recycle
### Anecdote: Decompressor

```bash
$ go tool pprof -alloc_space -top -cum mem.out
```

<table>
<thead>
<tr>
<th>flat</th>
<th>flat%</th>
<th>sum%</th>
<th>cum</th>
<th>cum%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>1529.93MB</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>chunk.BenchmarkDecode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>weaveworks/cortex/chunk/chunk_test.go</td>
<td></td>
</tr>
<tr>
<td>1442.37MB</td>
<td>94.25%</td>
<td>94.25%</td>
<td>1442.37MB</td>
<td>94.25%</td>
</tr>
<tr>
<td>.../vendor/github.com/golang/snappy/decode.go</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```go
snappy.NewReader(r) → sync.Pool
```
“Reuse” impact
func (cs ByKey) Less(i, j int) bool {
    return cs[i].ExternalKey() < cs[j].ExternalKey()
}

func (c *Chunk) ExternalKey() string {
}

Compare data directly
“Reduce” impact
Stack vs Heap

var x int

var y = make([]int, n)
Stack vs Heap
func BenchmarkOne(b *testing.B) {
    var buf io.Writer = &bytes.Buffer{}

    for i := 0; i < b.N; i++ {
        var data = []byte("hello")
        buf.Write(data)
    }
}
var buf io.Writer = &bytes.Buffer{}
for i := 0; i < b.N; i++ {
    var data = []byte("hello")
    buf.Write(data)
}

$ go test -bench=. -benchmem
BenchmarkOne    30000000   47.2 ns/op    27 B/op    1 allocs/op
Memory Profile

```go
var buf io.Writer = &bytes.Buffer{}
for i := 0; i < b.N; i++ {
    var data = []byte("hello")
    buf.Write(data)
}

$ go test -bench=. -memprofile=mem.out
$ go tool pprof -alloc_objects -top -cum mem.out

<table>
<thead>
<tr>
<th>flat</th>
<th>flat%</th>
<th>sum%</th>
<th>cum</th>
<th>cum %</th>
<th>cum%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0%</td>
<td>9830599</td>
<td>100%</td>
<td>testing.(*B).launch</td>
</tr>
<tr>
<td>9830550</td>
<td>100%</td>
<td>100%</td>
<td>9830565</td>
<td>100%</td>
<td>BenchmarkOne</td>
</tr>
</tbody>
</table>
```
$ go tool pprof -alloc_objects -list=BenchmarkOne mem.out

9830550 9830565 (flat, cum) 100% of Total

9: func BenchmarkOne(b *testing.B) {
  var buf io.Writer = &bytes.Buffer{}
  for i := 0; i < b.N; i++ {
    var data = []byte("hello")
    buf.Write(data)
  }
}
Escape Analysis

```go
for i := 0; i < b.N; i++ {
    var data = []byte("hello")
    buf.Write(data)
}
```

$ go test -gcflags '-m -m'

```
test.go:13:27: ([]byte)("hello") escapes to heap
test.go:13:27: from data (assigned) at ./one_test.go:13:7
```

```
test.go:13:27: from buf.Write(data) (parameter to indirect call)
at test.go:14
```
Which kinds of things escape?

- Address is passed out of a function
- Parameters of indirect calls
- Passed to a chan or a goroutine, or defer
- Others...
  - Arguments of recursive calls
  - Added to a slice or map
  - Passed to `panic()`
  - Too large for stack
Action points

• Measure your system
  – If CPU is high, look at profile
  – If GC is high, look at memory allocations

• It’s always memory allocations. 😊

• Avoid via:
  – Stack instead of heap
  – Different algorithm
  – Pooled objects
Thanks! Questions?

We are hiring!
Engineers in Berlin & SF

weave.works/hiring