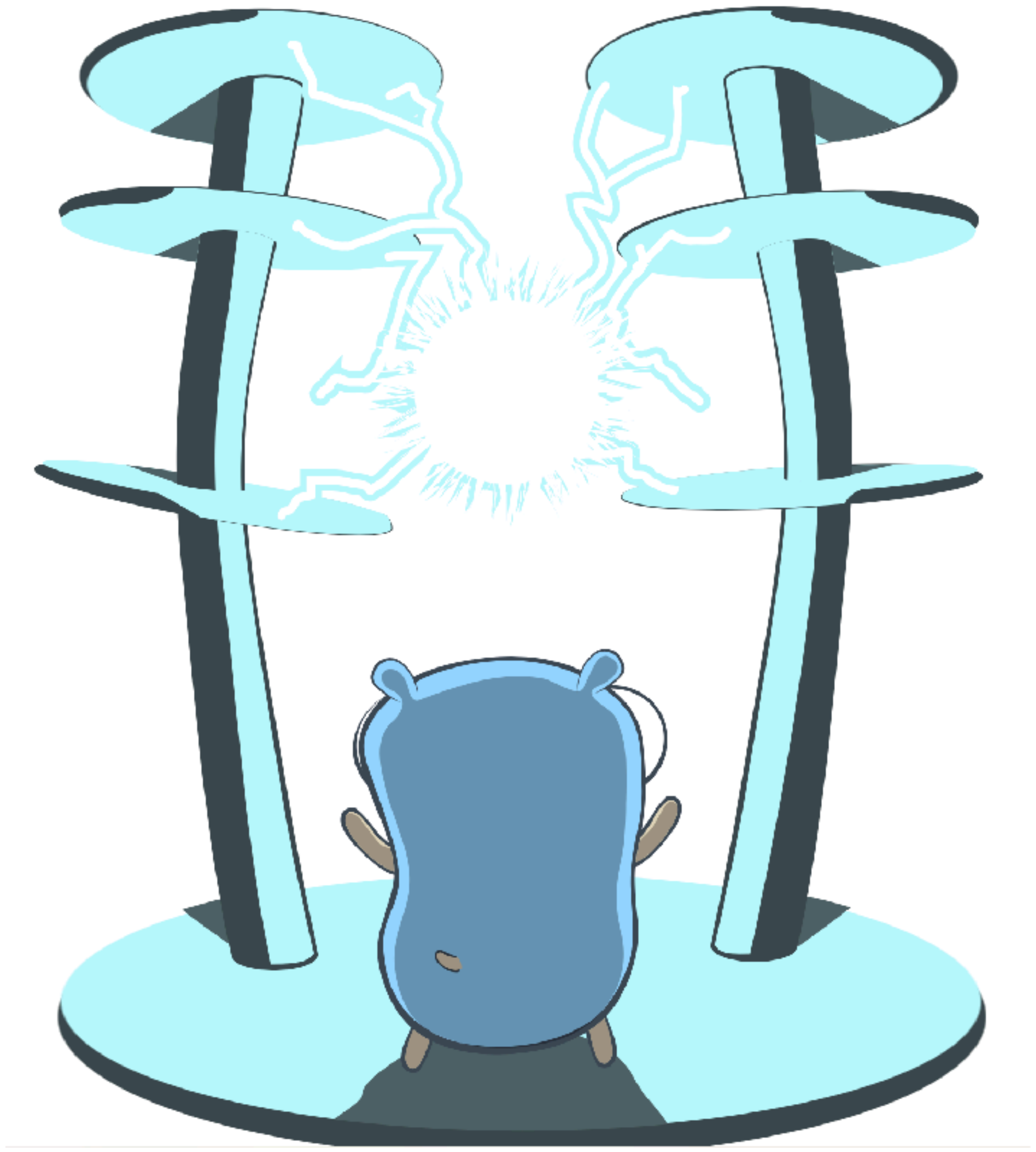
Go 1.9

Coolest Go release ever*

@idanyliuk
BCN Golang, Sep 28, 2017



*every Go release is coolest ever by definition

Performance

- As usual, most of the programs should be a bit faster
- Speedups in different libs
- GC optimizations
- Better generated code

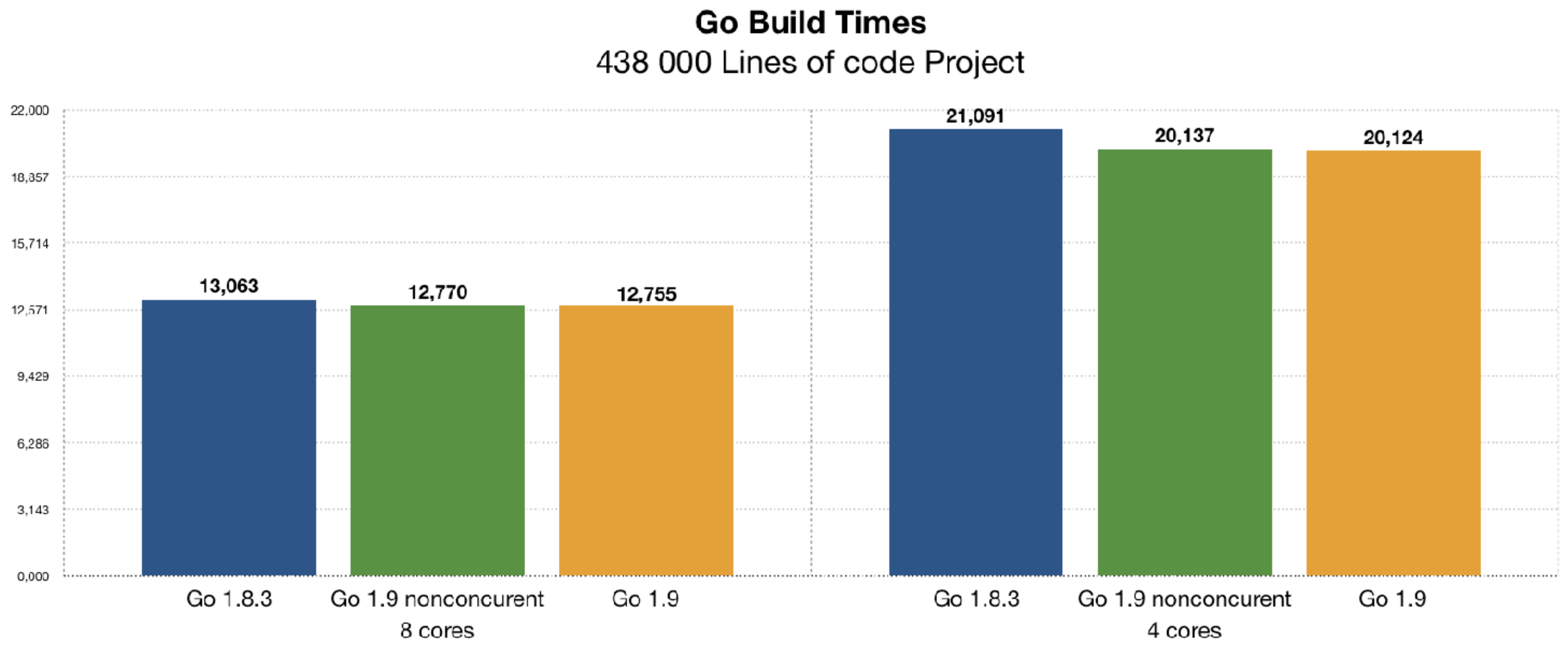
Parallel compilation

- Go has always compiled files in parallel
- In Go1.9 functions also can be processes/ compiled in parallel
- Can be disabled by:

```
export GO19CONCURRENTCOMPILATION=0
```

Parallel compilation

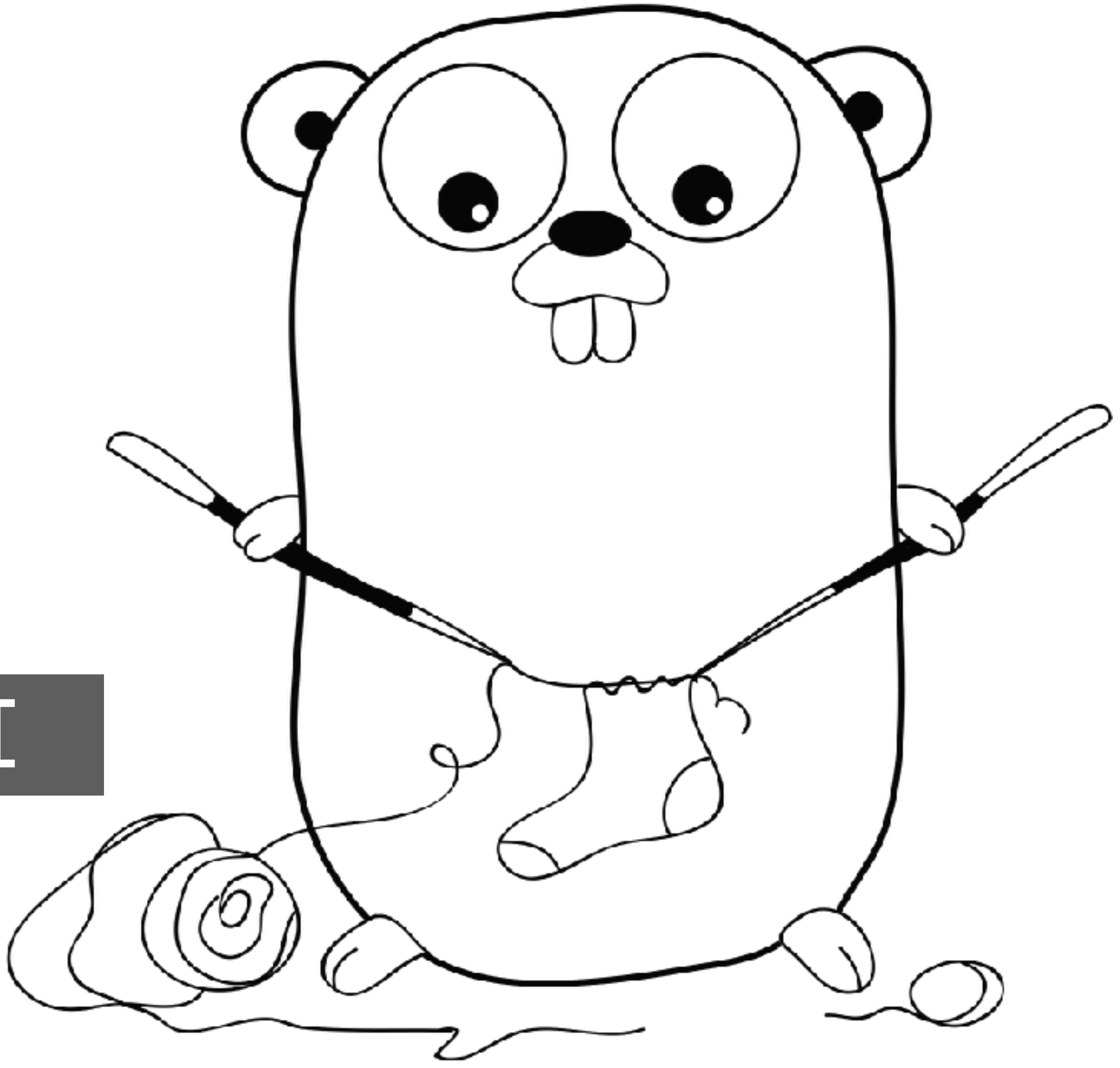
- Benefit depends on the width and hight of your packages
- For "many packages/not so many functions" gain is small



Type Aliases

- For refactoring large codebases
- T1 denotes **the same type** as T2
- Don't use it for anything else

```
type OldAPI = NewPackage.API
```



./... ignores vendor/ 🤘

- No more:

```
go test $(go list ./... | grep -v vendor)
```

- Just run:

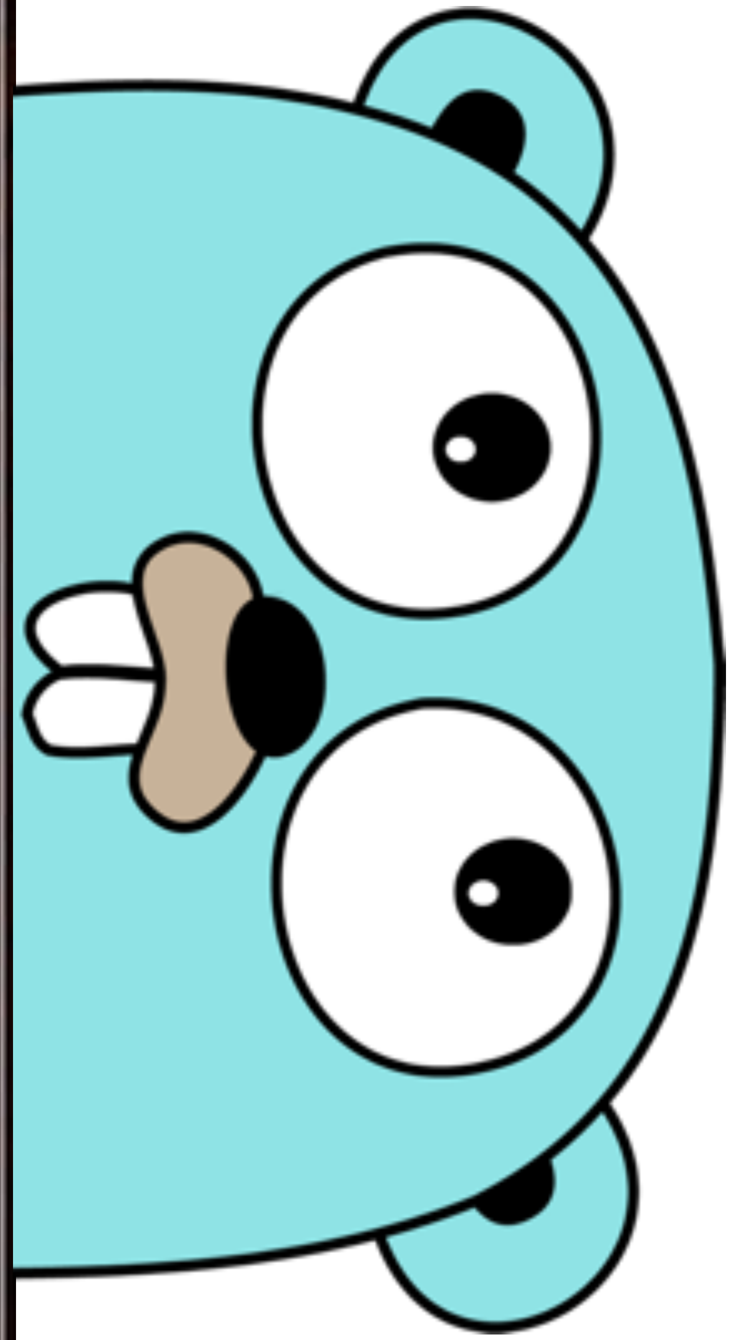
```
go test ./...
```

- If you need to also test vendor/:

```
go test ./vendor/...
```

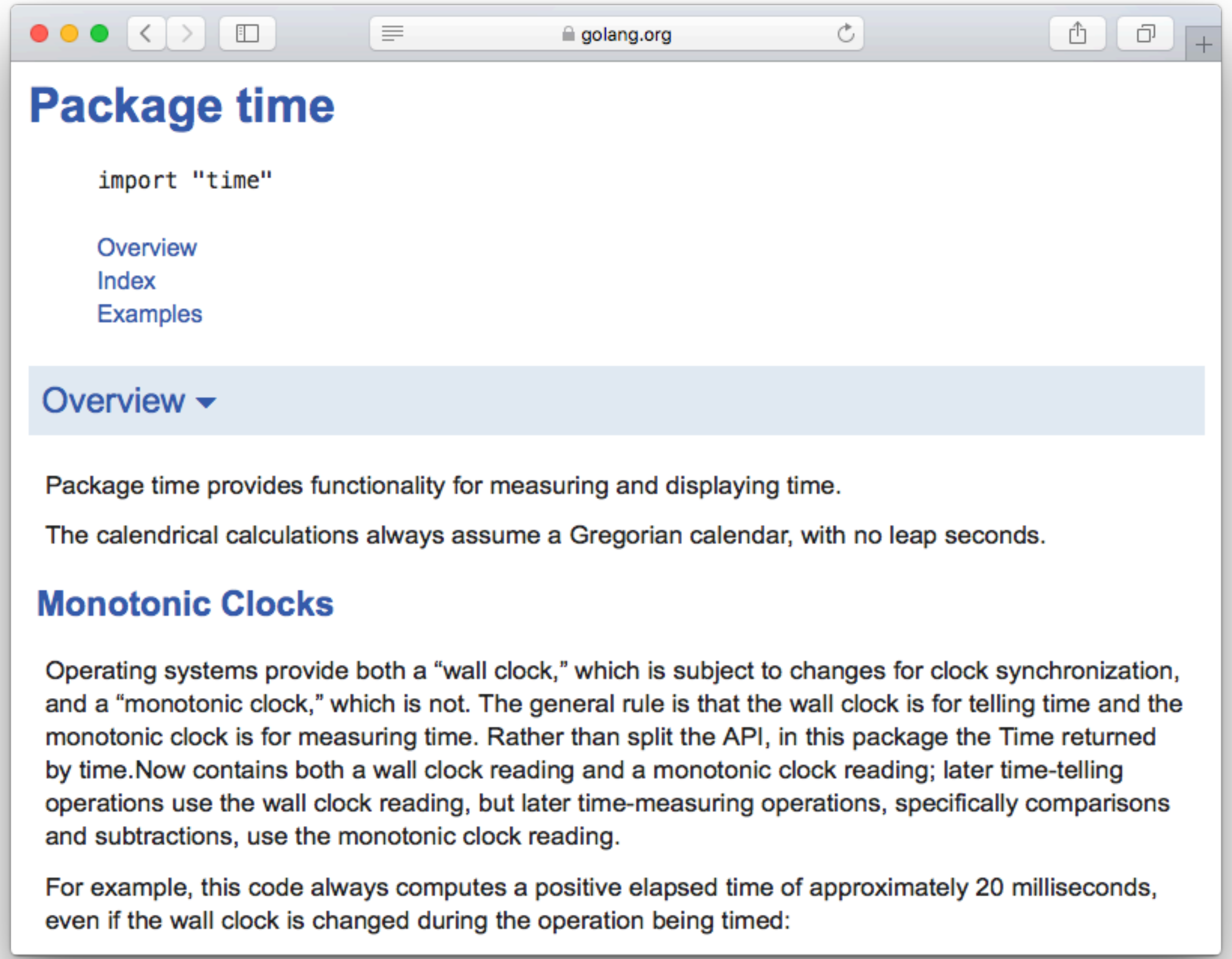
Monotonic time support

I DON'T ALWAYS USE TIME-OUTS...
BUT WHEN I DO, I USE THE MONOTONIC CLOCK.
memegenerator.net



- Each OS has two clocks - "*wall*" and "*monotonic*"
- *Wall* clock - for telling time
- *Monotonic* clock - for measuring time
- If time changes (sync, "leap second"), time.Duration() before Go 1.9 could return wrong measurement

Monotonic time support



Package time

```
import "time"
```

Overview
Index
Examples

Overview ▾

Package time provides functionality for measuring and displaying time.

The calendrical calculations always assume a Gregorian calendar, with no leap seconds.

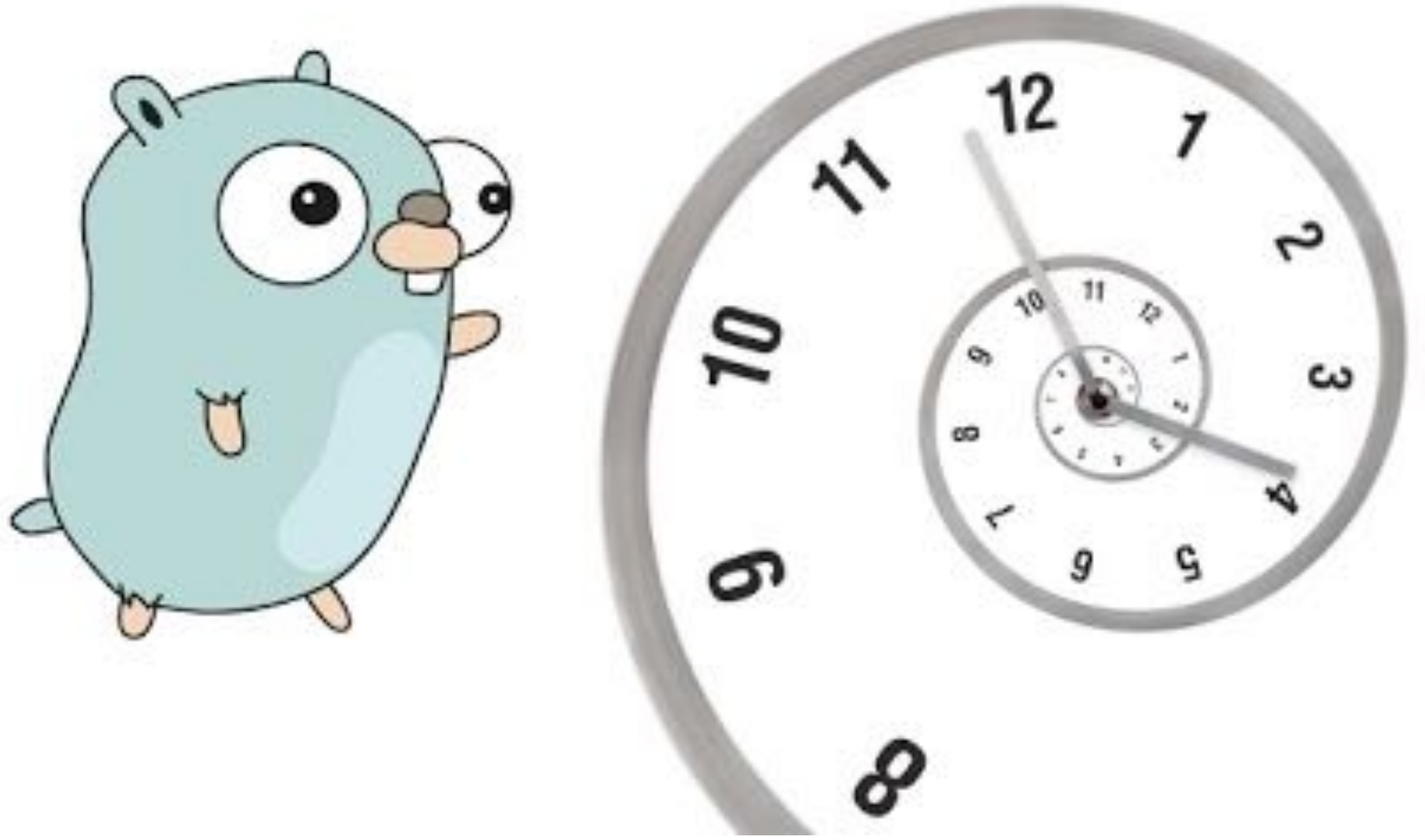
Monotonic Clocks

Operating systems provide both a “wall clock,” which is subject to changes for clock synchronization, and a “monotonic clock,” which is not. The general rule is that the wall clock is for telling time and the monotonic clock is for measuring time. Rather than split the API, in this package the Time returned by time.Now contains both a wall clock reading and a monotonic clock reading; later time-telling operations use the wall clock reading, but later time-measuring operations, specifically comparisons and subtractions, use the monotonic clock reading.

For example, this code always computes a positive elapsed time of approximately 20 milliseconds, even if the wall clock is changed during the operation being timed:

- CloudFlare wrote a blog post of how this absence of monotonic clock support in Go causes serious outage.
- Rationale of using monotonic clocks are well described in the GoDoc for time package in Go 1.9
- What is cool: there is no change in API
- Go will use right clock for the right task

Monotonic time support



Don't use Time.String() for comparison:

```
now := time.Now()
now.String()
```

Go 1.8

2017-09-24 20:05:54.356882078 +0200 CEST

Go 1.9

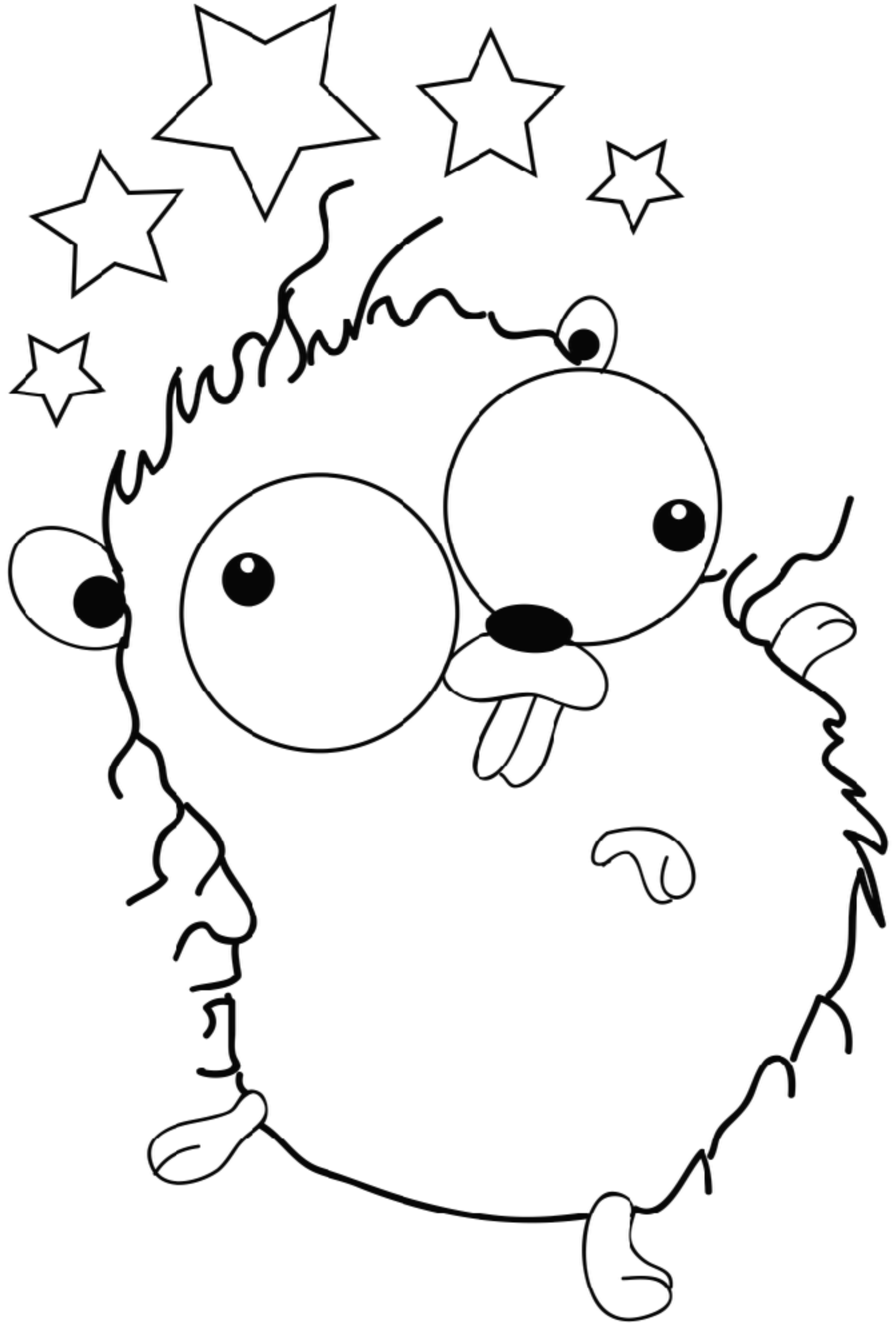
2017-09-24 20:05:38.755165304 +0200 CEST **m=+0.000259428**

Use instead:

```
now.Format()
```

sync.Map

- Concurrent map that solves **specific case of cache contention**:
  - high-performance (ns makes a diff)
  - stable keys
  - many CPU cores (16 and more)
- In other cases, map+RWLock mutex is generally better

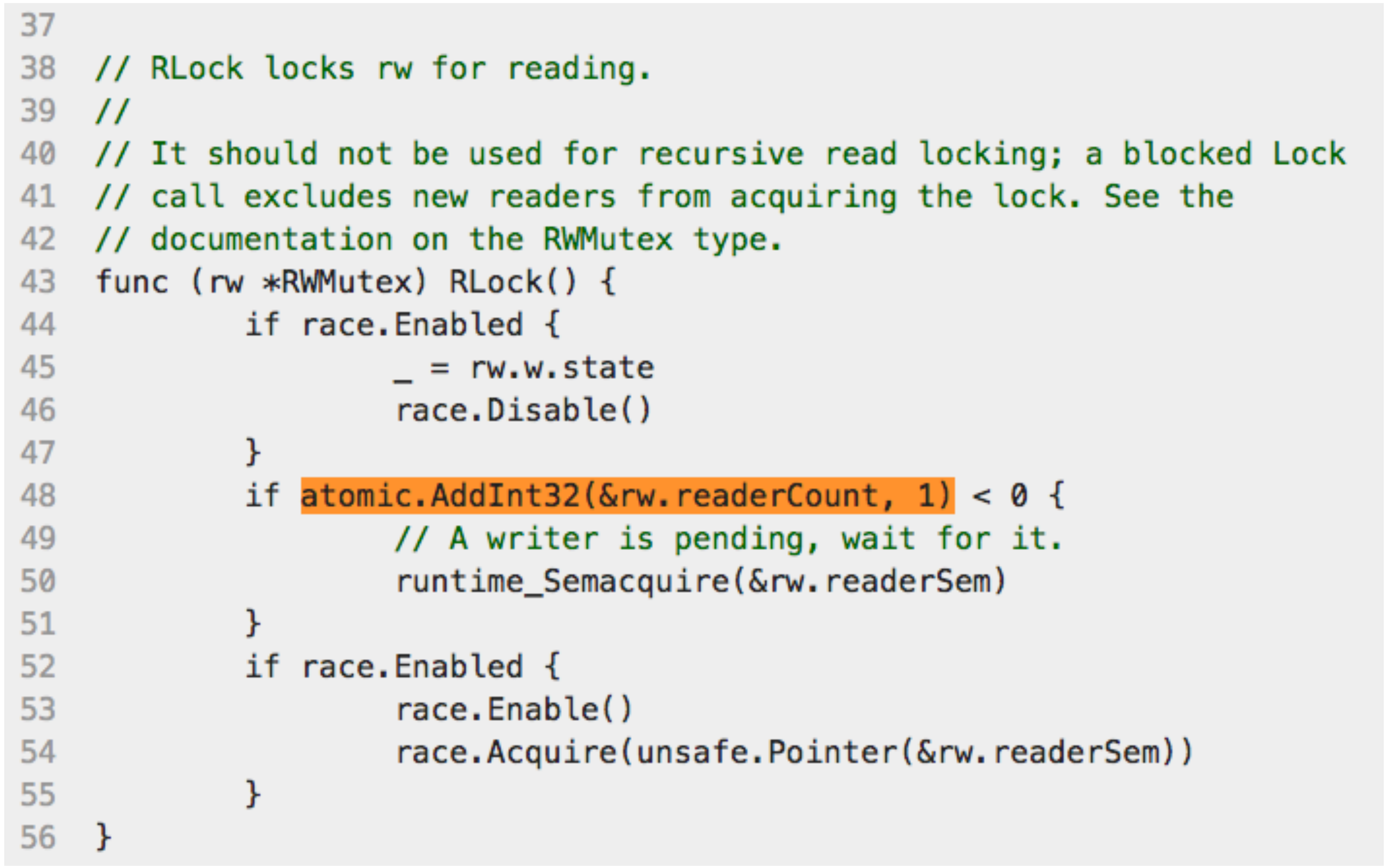


Cache Contention

sync.Map

- You use **encoding/json**...
- ...encoding/json uses **reflect**
- ...reflect uses sync.**RWMutex**
- ...RWMutex uses **atomic.AddInt32** to update readers counter
- ...each reader needs to **invalidate L2 cache and transfer value** from other core cache
- L2 cache transfer is around **40ns** on modern CPU
- O(1) task becomes **O(N)** by number of cores = **cache contention**

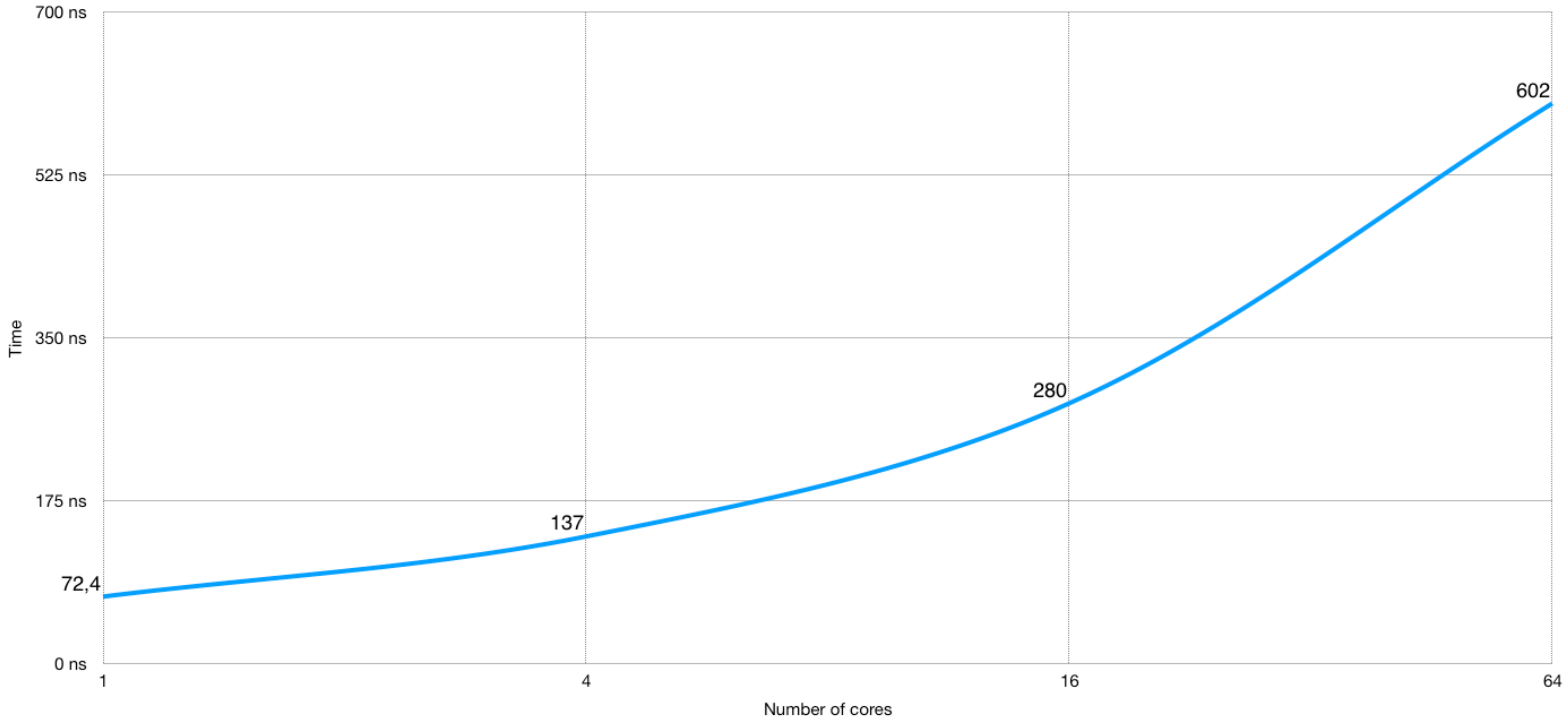
sync.Map



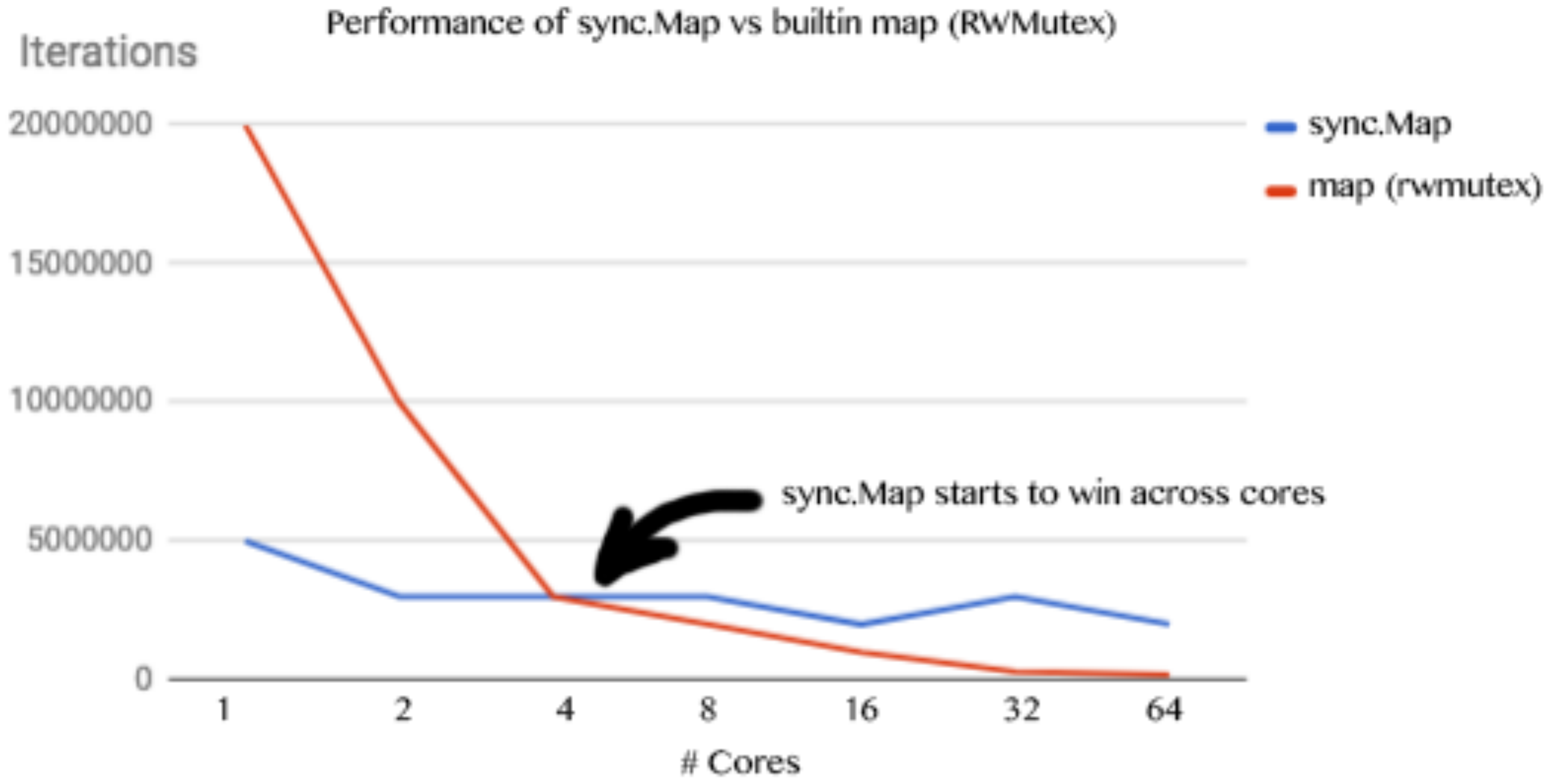
```go

// RLock locks rw for reading.
//
// It should not be used for recursive read locking; a blocked Lock
// call excludes new readers from acquiring the lock. See the
// documentation on the RWMutex type.
func (rw *RWMutex) RLock() {
	if race.Enabled {
		_ = rw.w.state
		race.Disable()
	}
	if atomic.AddInt32(&rw.readerCount, 1) < 0 {
		// A writer is pending, wait for it.
		runtime_Semacquire(&rw.readerSem)
	}
	if race.Enabled {
		race.Enable()
		race.Acquire(unsafe.Pointer(&rw.readerSem))
	}
}
```

sync.Map

Benchmark from original issue on RWMutex's cache contention:



sync.Map



Performance of sync.Map vs builtin map (RWMutex)
Iterations
20000000
15000000
10000000
5000000
0
1
2
4
8
16
32
64
# Cores
sync.Map
map (rwmutex)
sync.Map starts to win across cores

Map

```
m := make(map[string]int64)

m["key"] = 42

val, ok := m["key"]

delete(m, "key")
```

sync.Map

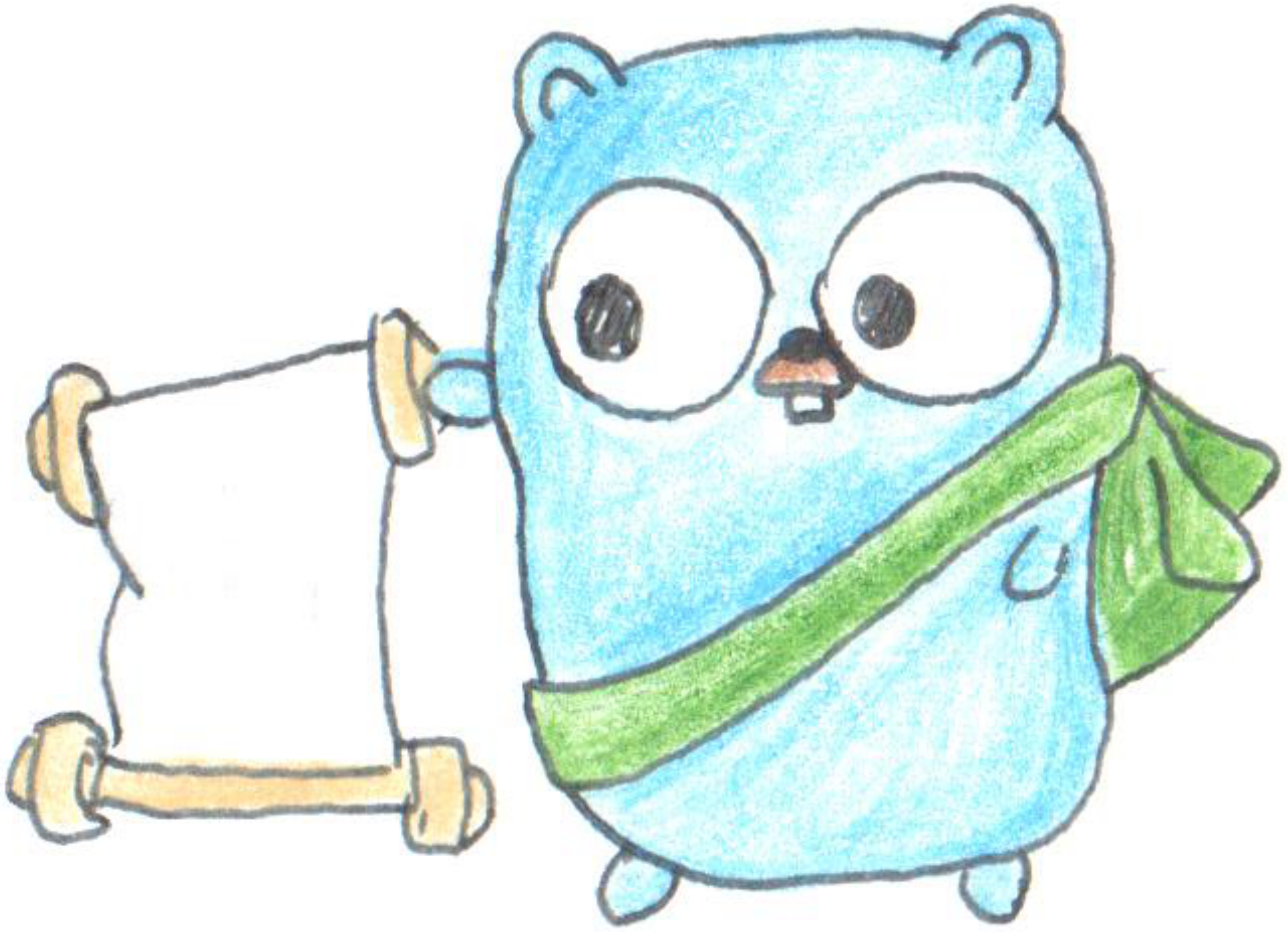
```
var m sync.Map

m.Store("key", 42)

val, ok := m.Load("key")

m.Delete("key")

val, ok := m.LoadOrStore("key")
m.Range(func(k, v interface{}) bool
{
  fmt.Println("key", k, "val", v)
  return true
})
```

PProf

- Profile files now contain symbol information
- Means - no need to keep binaries

```
go tool pprof cpu.prof
```

- Super useful for profiling remote servers or cross compiled apps



Profile Labels

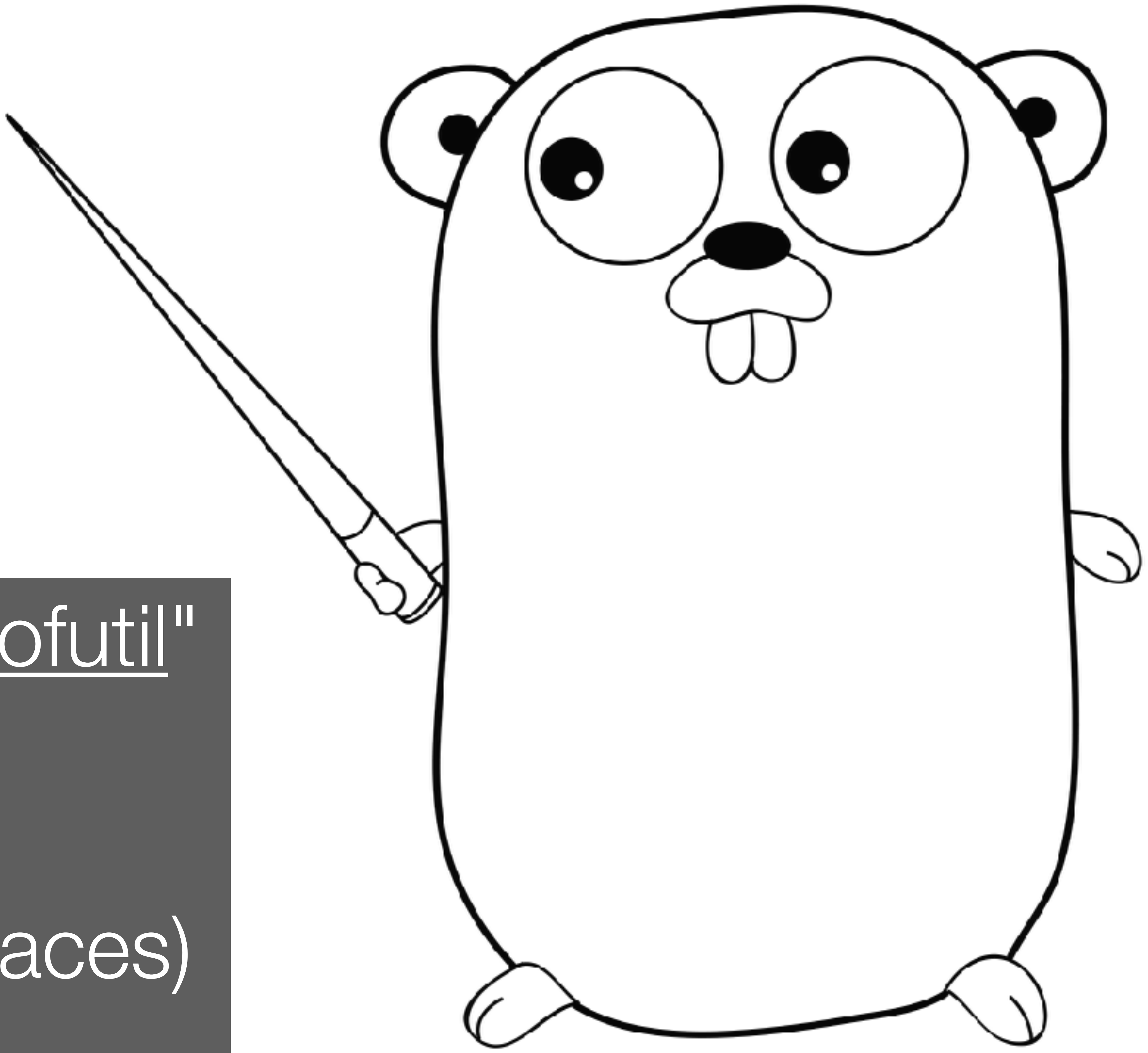
- New feature in profile - custom labels
- Adds label to functions you profiling

```
l := pprof.Labels("ext", "zip")
pprof.Do(ctx, l,
    func(ctx context.Context) {
        myFunc(ctx, args)
    }
)
```

Profile Labels

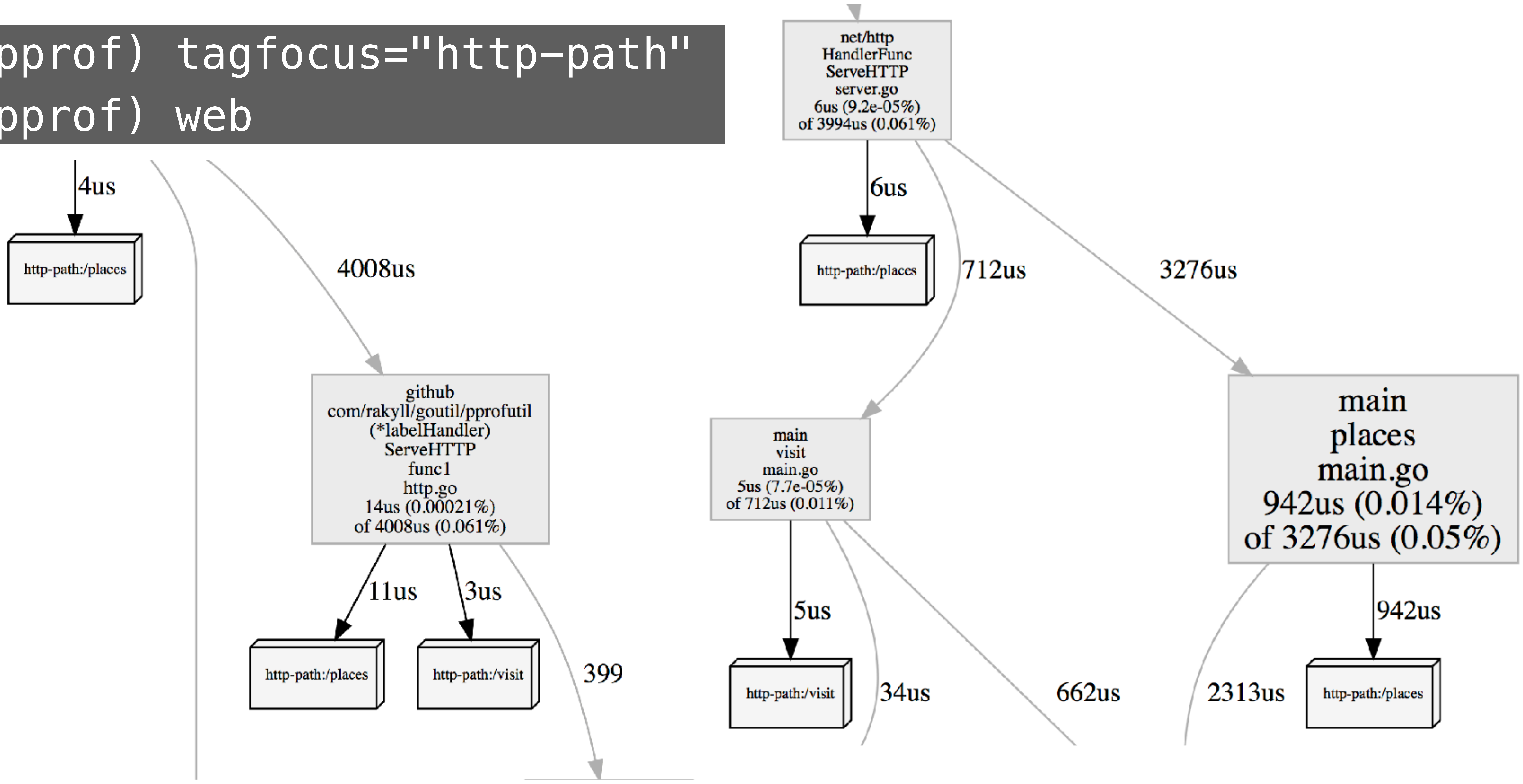
- https://rakyll.org/profiler-labels/
- Example: pprofutil package,
- wrapper for http.Handler
- Adds "http-path" labels to each request

```
import "github.com/rakyll/goutil/pprofutil"

http.Handle("/places",
    pprofutil.LabelHandlerFunc(places)
    )
```



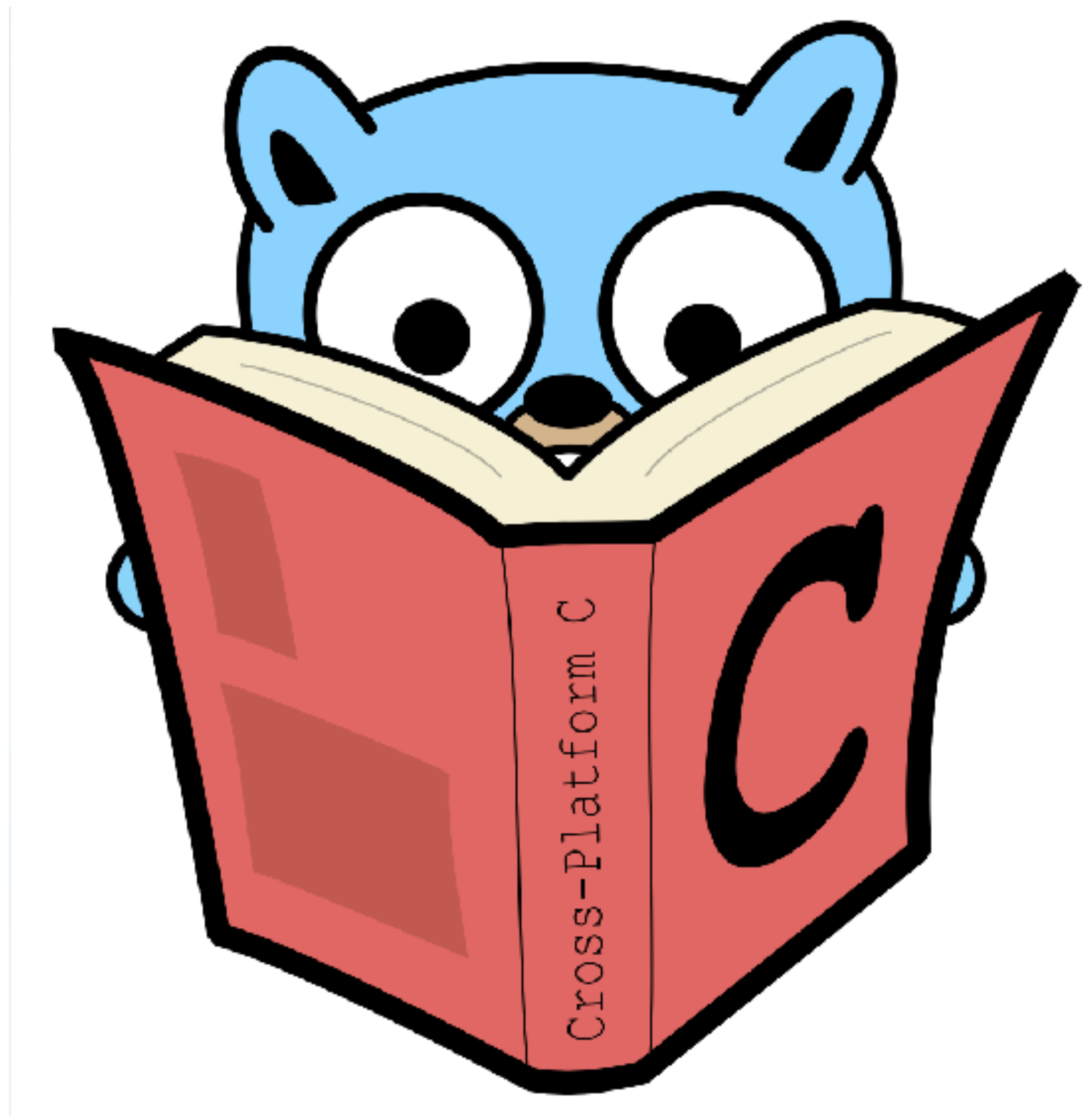
Profile Labels

```
(pprof) tagfocus="http-path"
(pprof) web
```



More stdlib changes

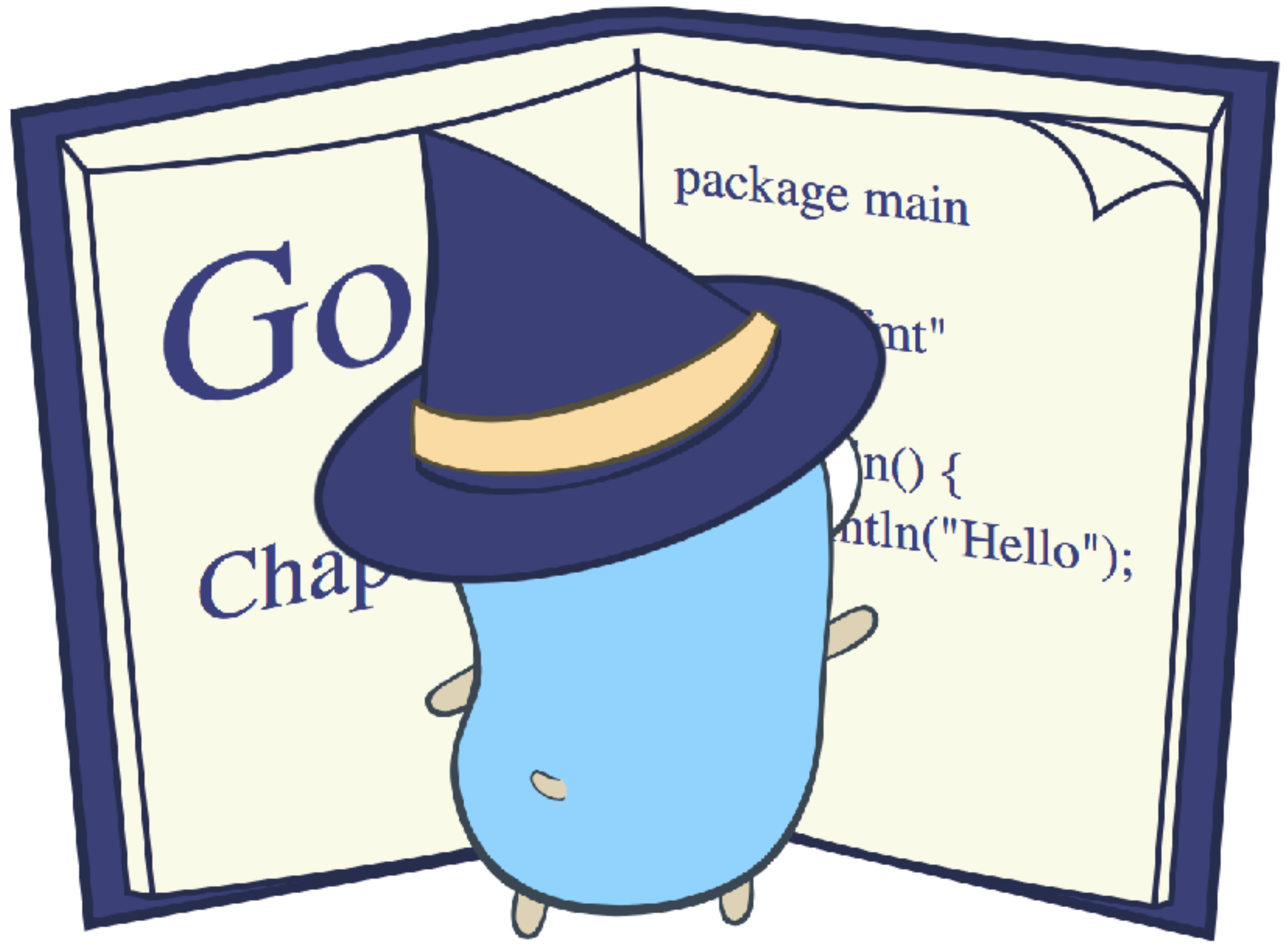
math/bits



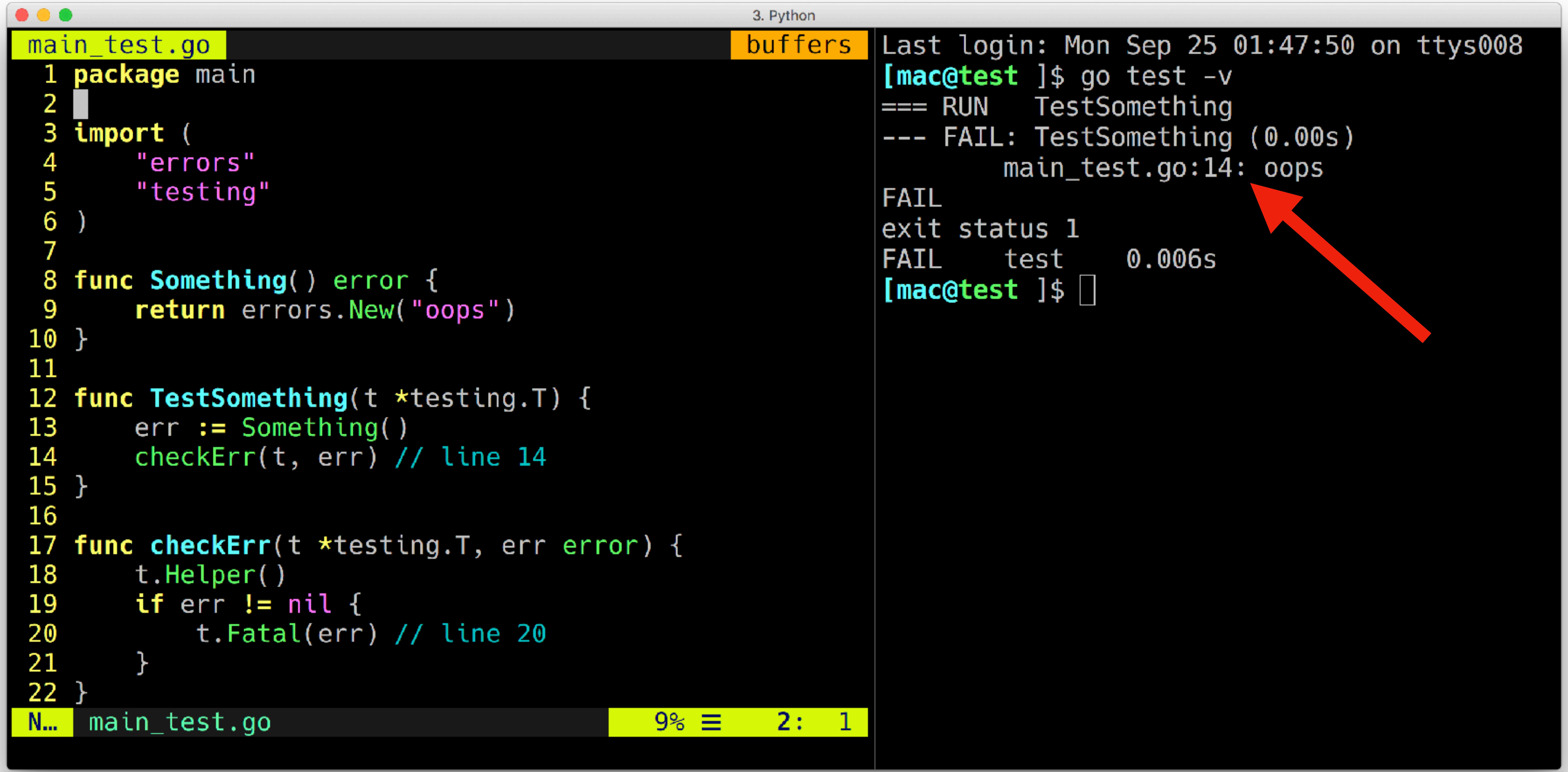
- New package with bit manipulation functions
- Highly optimized for different CPU architectures
  - Rotate bits
  - Count zeroes / ones
  - Reverse bits/bytes
- If you're asked on interview how to reverse bits, use math/bits :)

Test helpers functions

- In testing package:
  - **(*T).Helper()**
  - **(*B).Helper()**
- Marks functions as helper, so it's skipped when reports file:line information in log.



Test helpers functions

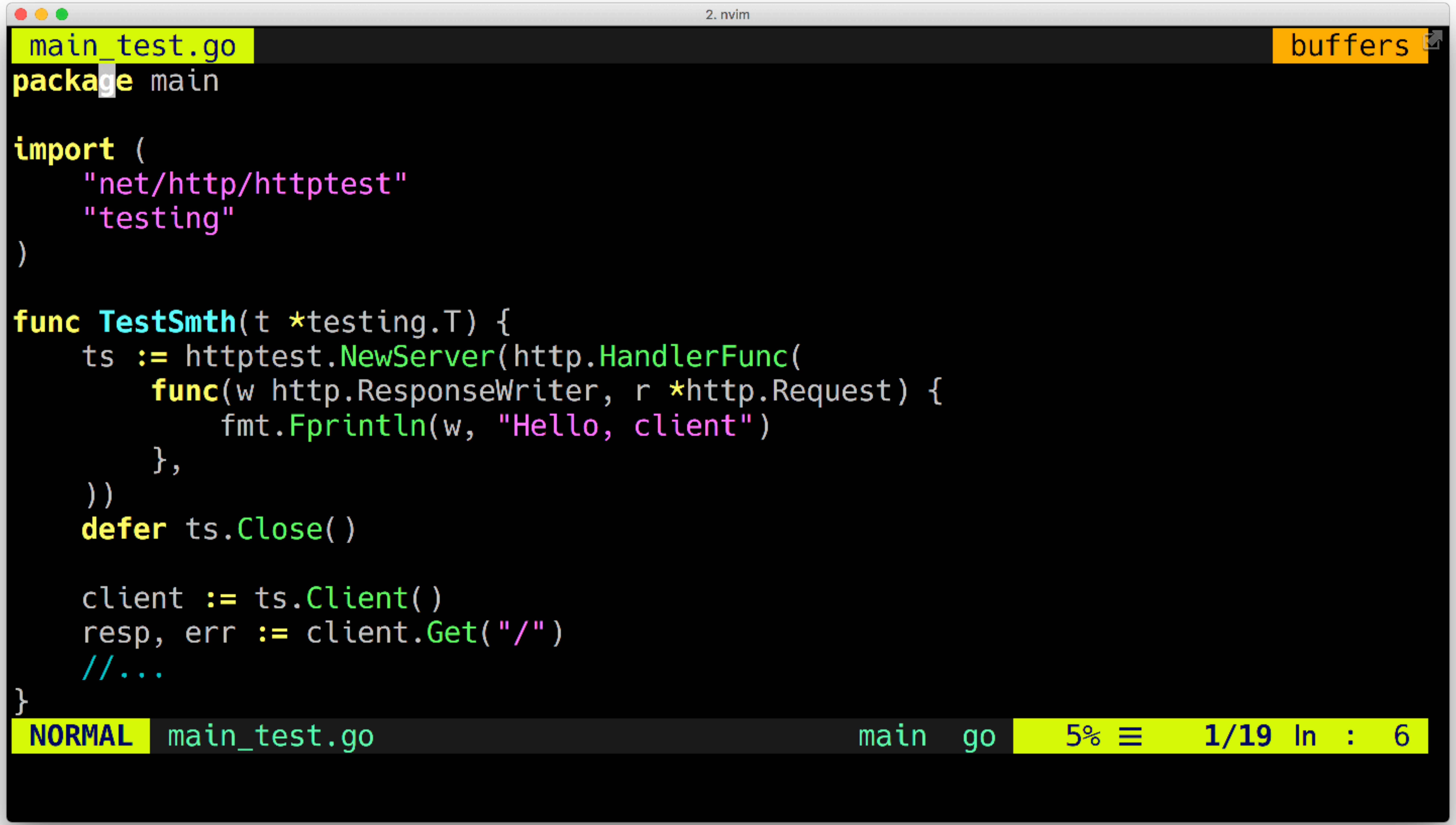


3. Python

```
main_test.go                                   buffers
 1 package main
 2
 3 import (
 4     "errors"
 5     "testing"
 6 )
 7
 8 func Something() error {
 9     return errors.New("oops")
10 }
11
12 func TestSomething(t *testing.T) {
13     err := Something()
14     checkErr(t, err) // line 14
15 }
16
17 func checkErr(t *testing.T, err error) {
18     t.Helper()
19     if err != nil {
20         t.Fatal(err) // line 20
21     }
22 }
N… main_test.go                          9% ≡   2:  1
```

```
Last login: Mon Sep 25 01:47:50 on ttys008
[mac@test ]$ go test -v
=== RUN   TestSomething
--- FAIL: TestSomething (0.00s)
        main_test.go:14: oops
FAIL
exit status 1
FAIL    test    0.006s
[mac@test ]$
```

httptest.Server.Client()



2. nvim

main_test.go buffers

```go
package main

import (
	"net/http/httptest"
	"testing"
)

func TestSmth(t *testing.T) {
	ts := httptest.NewServer(http.HandlerFunc(
		func(w http.ResponseWriter, r *http.Request) {
			fmt.Fprintln(w, "Hello, client")
		},
	))
	defer ts.Close()

	client := ts.Client()
	resp, err := client.Get("/")
	//...
}
```

NORMAL main_test.go main go 5% ≡ 1/19 ln : 6

Go 1.9 is cool

- Upgrade now!
- Amazing artwork by @egonlibre and Olga Shalakhina

Thank you