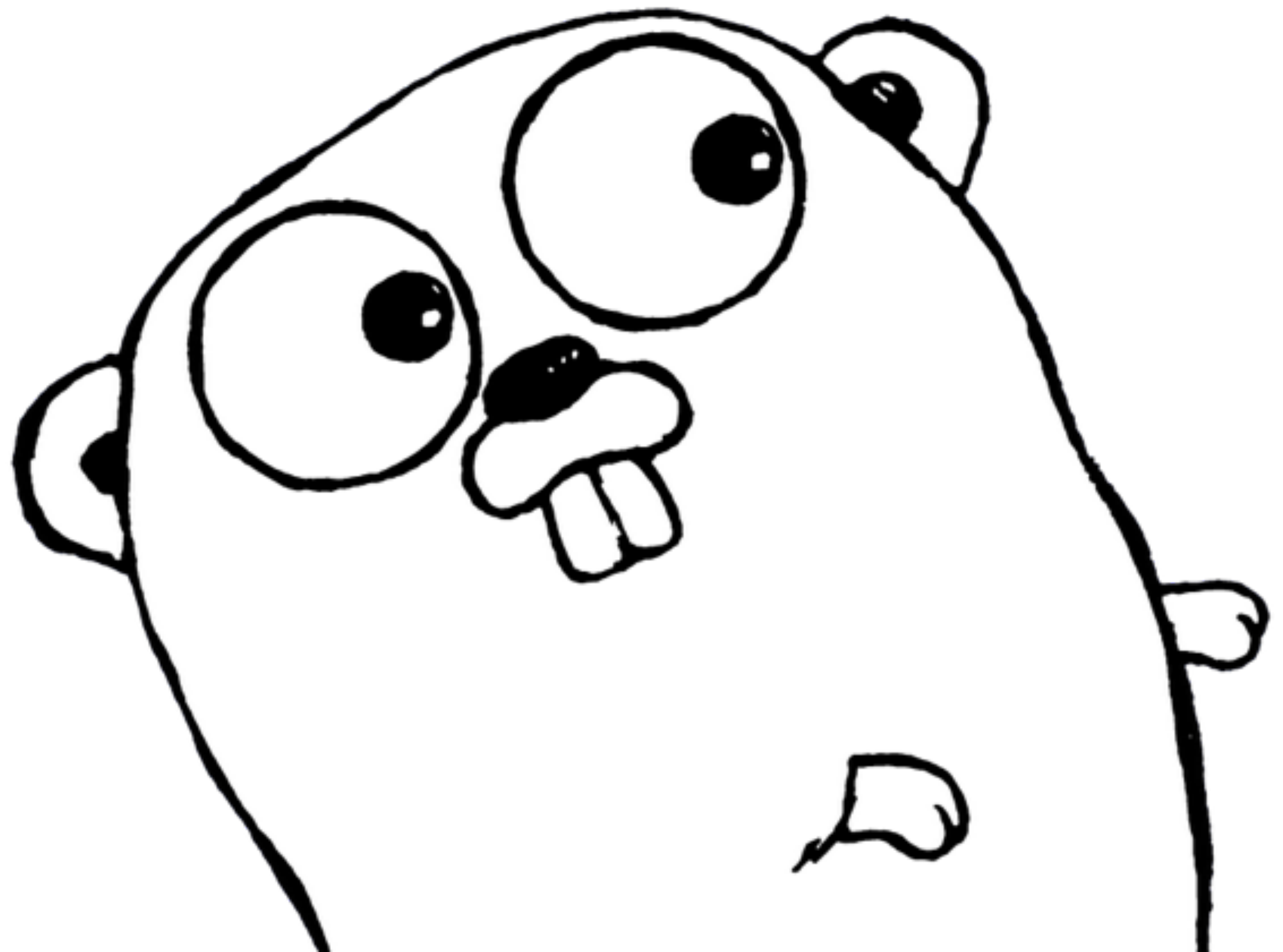
Successful Go program design

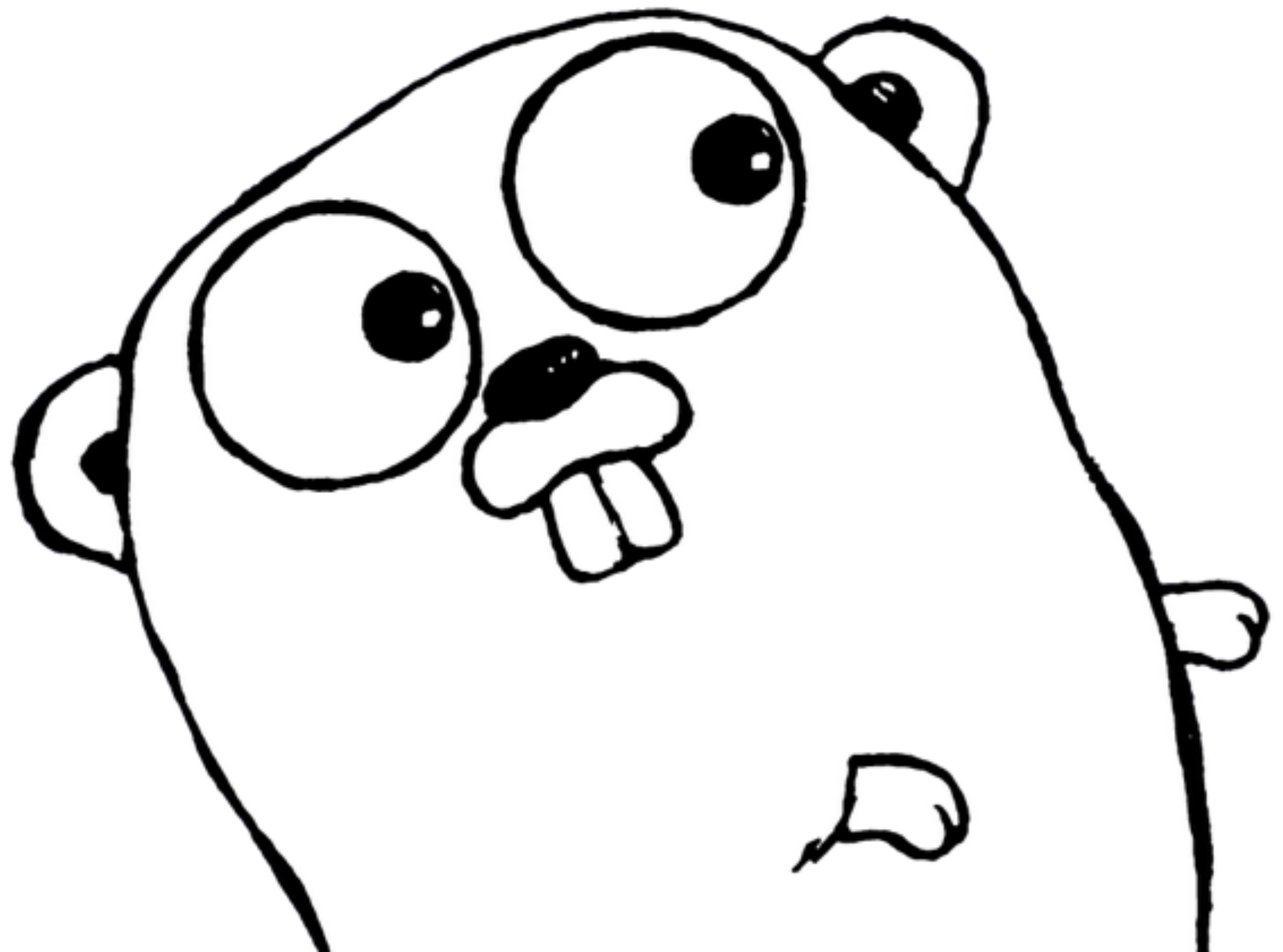
Six years on



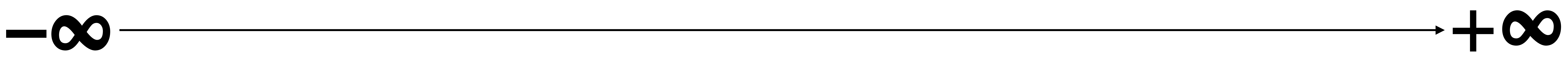
HAIL
ROBOTS

Successful Go program design

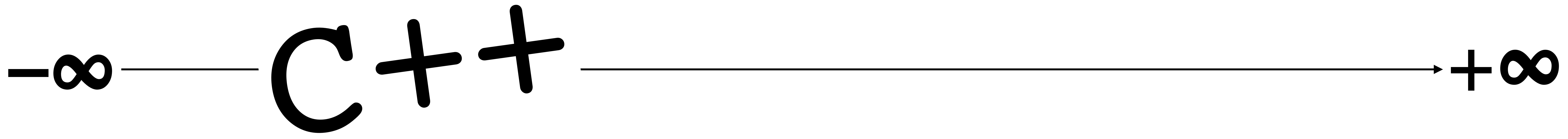
Six years on



My background



My background

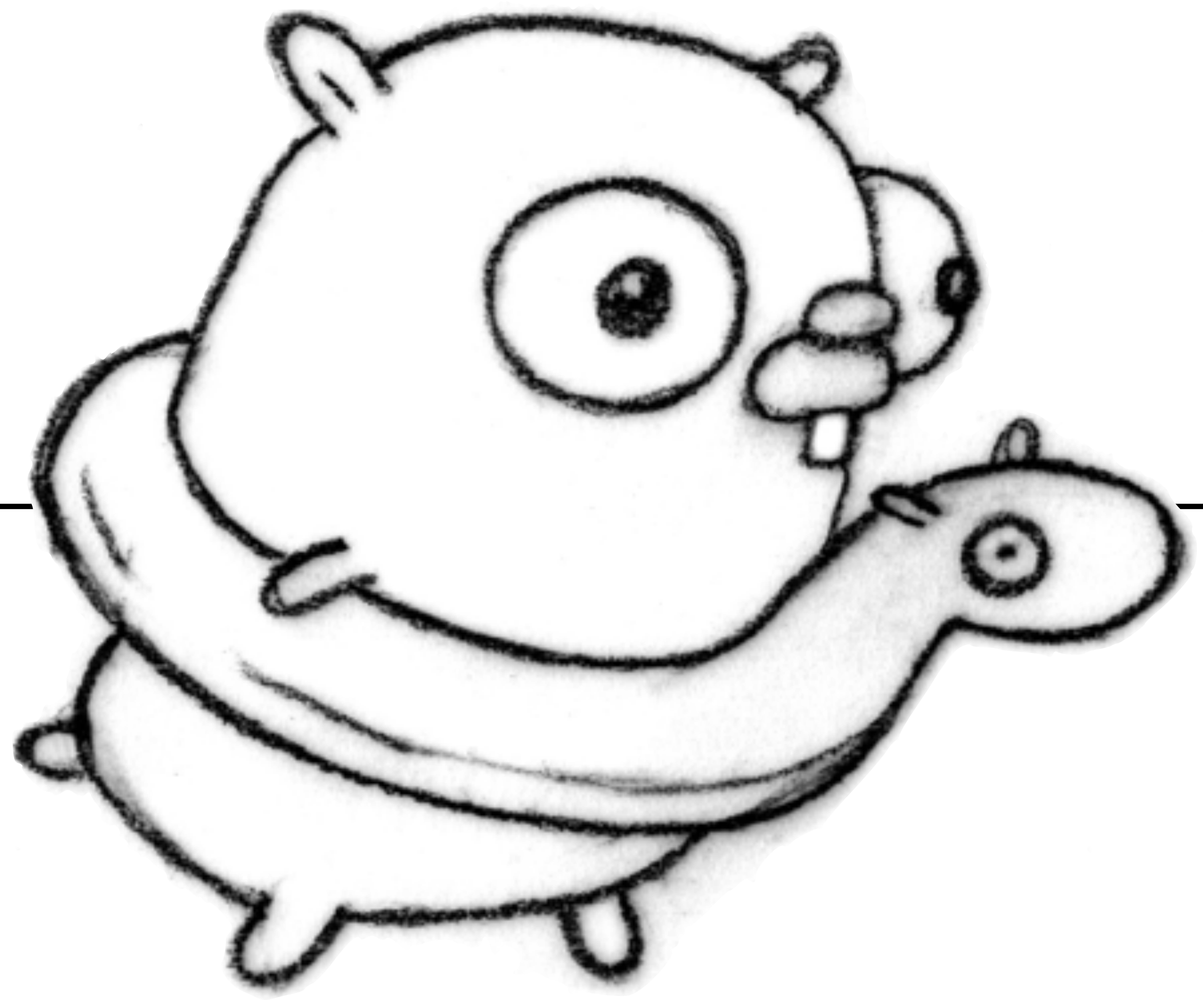


My background

$$\begin{tikzcd}
-\infty \arrow[r, dash] & \text{C++} \arrow[r, dash] & 2009 \arrow[r] & +\infty
\end{tikzcd}$$

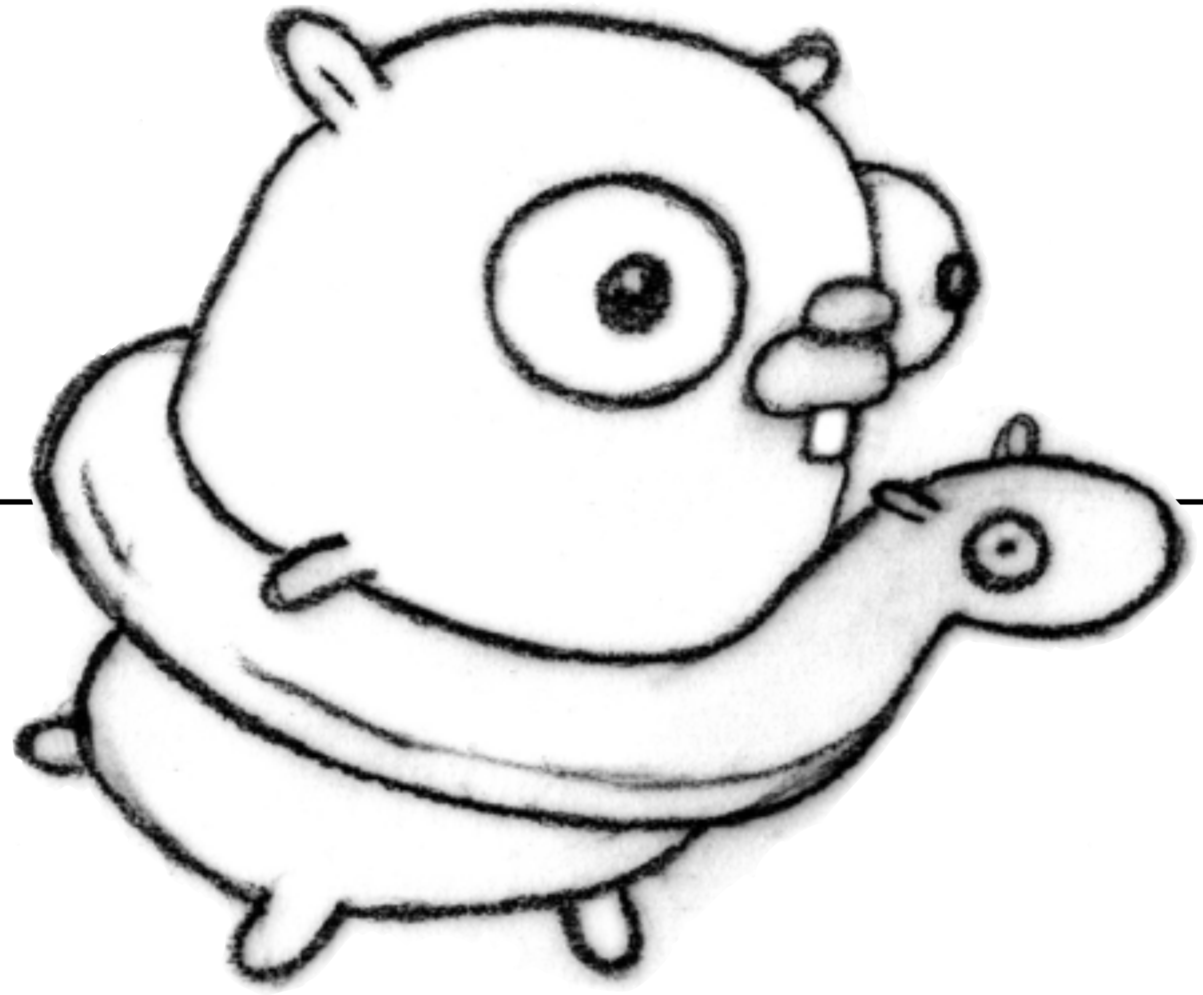
My background

−∞ —— C++ —— 2009 ——



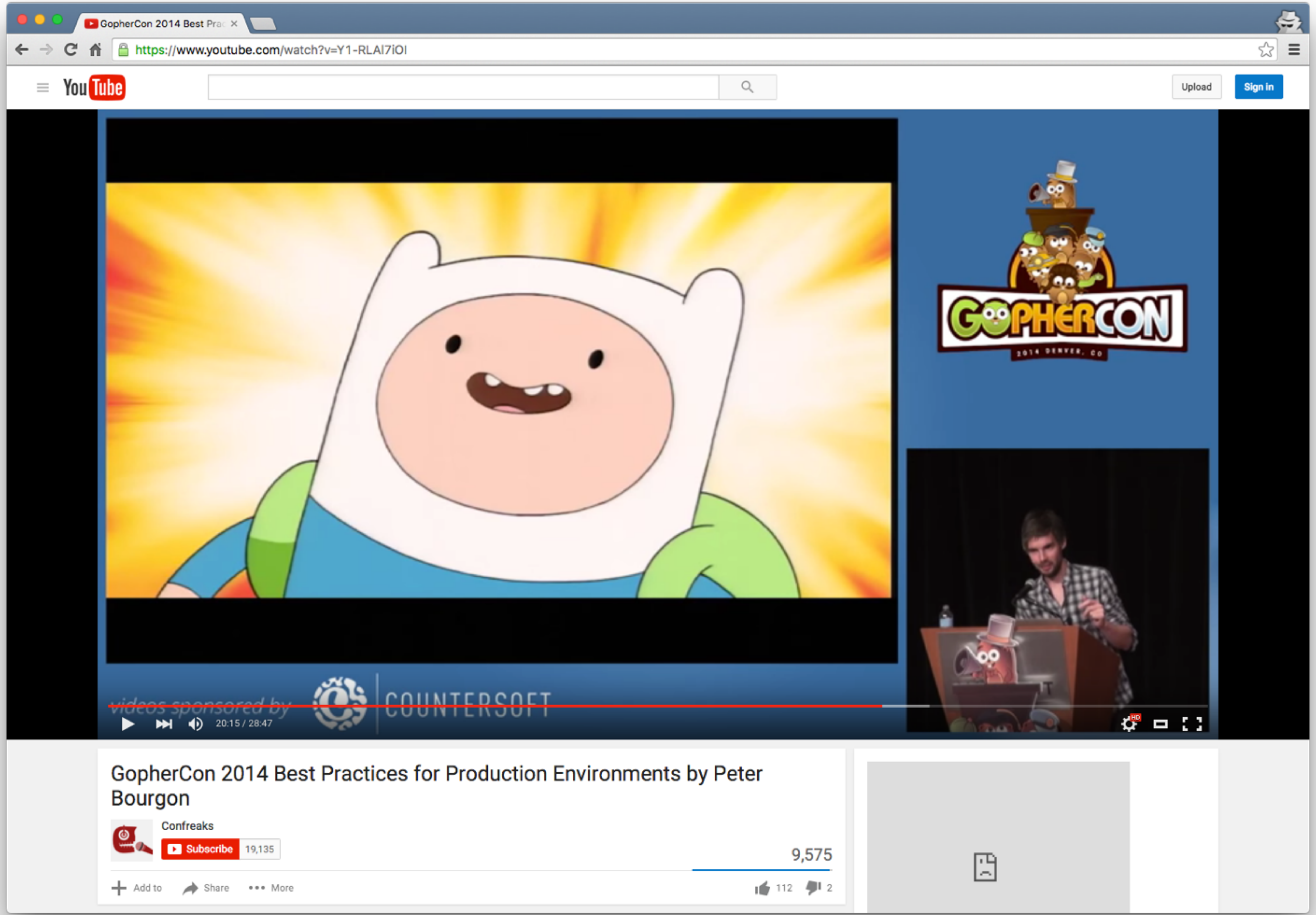
→ +∞

My background

−∞ 2009  +∞

My background

- github.com/peterbourgon/**diskv**
- developers.soundcloud.com/blog/**go-at-soundcloud**
- github.com/soundcloud/**roshi**
- github.com/weaveworks/**scope**
- github.com/**go-kit**/kit


GopherCon 2014 Best Practices for Production Environments by Peter Bourgon

Confreaks

Subscribe 19,135

9,575

Dev environment

Repo structure

Formatting and style

Configuration

Logging and telemetry

Validation and testing

Dependency management ᕕ(ᐛ)ᕗ

Build and deploy

Dev environment

Repo structure

Formatting and style

Configuration

Logging and telemetry

Validation and testing

Dependency management ᕕ(ᐛ)ᕗ

Build and deploy

1. Dev environment

Dev environment

- $GOPATH
  - Single global $GOPATH — still the easiest/best
  - Per-project $GOPATH — OK for binaries, see **getgb.io**
  - Two-entry $GOPATH — OK for strict internal/external separation
- Put $GOPATH/bin in your $PATH

Dev environment

- $GOPATH
  - Single global $GOPATH — still the easiest/best
  - Per-project $GOPATH — OK for binaries, see **getgb.io**
  - Two-entry $GOPATH — OK for strict internal/external separation
- Put $GOPATH/bin in your $PATH

TOP TIP

2. Repo structure

Repo structure

- Private/internal — go nuts: own GOPATH, custom build tools, etc.
- Public/OSS — please play nice with **go get**
- Command || library — base dir + subdirs for other packages
- Command && library — which is primary? Optimize for use…

Repo structure

```
github.com/peterbourgon/foo/
    main.go
    main_test.go
    handlers.go
    handlers_test.go
    compute.go
    compute_test.go
    lib/
        foo.go
        foo_test.go
        bar.go
        bar_test.go
```

Repo structure

```
github.com/peterbourgon/foo/
    main.go
    main_test.go
    handlers.go
    handlers_test.go                package main
    compute.go
    compute_test.go
    lib/
        foo.go
        foo_test.go                 package foo
        bar.go
        bar_test.go
```

Repo structure

```
github.com/peterbourgon/foo/
    main.go
    main_test.go
    handlers.go
    handlers_test.go
    compute.go
    compute_test.go
    lib/
        foo.go
        foo_test.go
        bar.go
        bar_test.go
```

package main

package foo

TOP TIP

Repo structure

```
github.com/peterbourgon/foo/
    main.go
    main_test.go
    handlers.go
    handlers_test.go
    compute.go
    compute_test.go
    lib/
        foo.go
        foo_test.go
        bar.go
        bar_test.go
```

$\leftarrow$ package main

$\leftarrow$ package foo

github.com/tsenart/**vegeta**

Repo structure

```
github.com/peterbourgon/foo/
    foo.go
    foo_test.go
    bar.go
    bar_test.go
    cmd/
        foo/
            main_test.go
            handlers.go
            handlers_test.go
            compute.go
            compute_test.go
```

Repo structure

```
github.com/peterbourgon/foo/
    foo.go
    foo_test.go
    bar.go
    bar_test.go
    cmd/
        foo/
            main_test.go
            handlers.go
            handlers_test.go
            compute.go
            compute_test.go
```

package foo

package main

Repo structure

```
github.com/peterbourgon/foo/
    foo.go
    foo_test.go
    bar.go
    bar_test.go
    cmd/
        foo/
            main_test.go
            handlers.go
            handlers_test.go
            compute.go
            compute_test.go
```

package foo

package main

github.com/constabulary/**gb**

3. Formatting and style

Formatting and style

- Go has strong opinions — abide by them
- Format (gofmt) on save — no excuses
- github.com/golang/go/wiki/CodeReviewComments
    - bit.ly/**GoCodeReview**
- talks.golang.org/2014/names.slide
    - bit.ly/**GoNames**

Formatting and style

- Go has strong opinions — abide by them
- Format (gofmt) on save — no excuses
- github.com/golang/go/wiki/CodeReviewComments
  - bit.ly/**GoCodeReview**
- talks.golang.org/2014/names.slide
  - bit.ly/**GoNames**

TOP TIP

4. Configuration

Configuration

- Configuration bridges environment and process domains
- **Make it explicit!**
- package flag — though I wish it were less esoteric...
- os.Getenv — too subtle, too implicit; avoid
- Env vars + flags — see the value, but **document in usage!**

Configuration

- Configuration bridges environment and process domains
- **Make it explicit!**
- package flag — though I wish it were less esoteric…
- os.Getenv — too subtle, too implicit; avoid

TOP TIP

- Env vars + flags — see the value, but **document in usage!**

Example program

```
package main

import (
    "log"

    "github.com/peterbourgon/foo/common"
)

func main() {
    log.Print(common.HelloWorld)
}
```

Package naming

```
package main

import (
    "log"

    "github.com/peterbourgon/foo/consts"
)

func main() {
    log.Print(consts.HelloWorld)
}
```

Package naming

```go
package main

import (
    "log"

    "github.com/peterbourgon/foo/greetings"
)

func main() {
    log.Print(greetings.HelloWorld)
}
```

Package naming

```
package main

import (
    "log"

    "github.com/peterbourgon/foo/greetings"
)

func main() {
    log.Print(greetings.HelloWorld)
}
```

TOP TIP

Dot import

```
package main

import (
    "log"

    . "github.com/peterbourgon/foo/greetings"
)

func main() {
    log.Print(HelloWorld)
}
```

Dot import

```
package main

import (
    "log"

    ~~. "github.com/peterbourgon/foo/greetings"~~
)

func main() {
    log.Print(HelloWorld)
}
```

Dot import

```
package main

import (
    "log"

    ~~. "github.com/peterbourgon/foo/greetings"~~
)

func main() {
    log.Print(HelloWorld)
}
```

TOP TIP

Flags

```
var stdout = flag.Bool("stdout", false, "log to stdout")

func init() {
    flag.Init()
}

func main() {
    if *stdout {
        log.SetOutput(os.Stdout)
    }
    log.Print(greetings.HelloWorld)
}
```

Flags

```
func main() {
    var stdout = flag.Bool("stdout", false, "log to stdout")
    flag.Init()

    if *stdout {
        log.SetOutput(os.Stdout)
    }
    log.Print(greetings.HelloWorld)
}
```

Flags

```
func main() {
    var stdout = flag.Bool("stdout", false, "log to stdout")
    flag.Init()

    if *stdout {
        log.SetOutput(os.Stdout)
    }
    log.Print(greetings.HelloWorld)
}
```

TOP TIP

http://bit.ly/**GoFlags**

Construction

```
func main() {
    var (
        stdout = flag.Bool("stdout", false, "log to stdout")
        fooKey = flag.String("fooKey", "", "access key for foo")
    )
    flag.Init()

    foo, err := newFoo(*fooKey)
    if err != nil {
        log.Fatal(err)
    }
    defer foo.close()
```

Construction

```
foo, err := newFoo(
    *fooKey,
    bar,
    baz,
    100 * time.Millisecond,
    nil,
)
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Construction

```
cfg := fooConfig{}
cfg.Bar = bar
cfg.Baz = baz
cfg.Period = 100 * time.Millisecond
cfg.Output = nil

foo, err := newFoo(*fooKey, cfg)
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Construction

```
cfg := fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
    Output: nil,
}
foo, err := newFoo(*fooKey, cfg)
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Construction

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
    Output: nil,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Construction

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
    Output: nil,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

TOP TIP

Usable defaults

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
    Output: nil,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Usable defaults

```
func (f *foo) process() {
    if f.Output != nil {
        fmt.Fprintf(f.Output, "beginning\n")
    }
    // ...
}
```

Usable defaults

```
func (f *foo) process() {
    fmt.Fprintf(f.Output, "beginning\n")
    // ...
}
```

Usable defaults

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
    Output: ioutil.Discard,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Usable defaults

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
    Output: ioutil.Discard,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

TOP TIP

Smart constructors

```
func newFoo(..., cfg fooConfig) *foo {
    if cfg.Output == nil {
        cfg.Output = ioutil.Discard
    }
    // ...
}
```

Smart constructors

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Smart constructors

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

TOP TIP

Cross-referential components

```
type bar struct {
    baz *baz
    // ...
}

type baz struct {
    bar *bar
    // ...
}
```

Cross-referential components

```
type bar struct {
    baz *baz
    // ...
}

type baz struct {
    bar *bar
    // ...
}
```

```
bar := newBar(...)
baz := newBaz(...)

bar.baz = baz
baz.bar = bar

// :(
```

Cross-referential components

- Combine
- Split
- Externalize communication

Combine

```
type bar struct {
    baz *baz
    // ...
}

type baz struct {
    bar *bar
    // ...
}
```

→

```
type barbaz struct {
    // ...
}
```

Split

```go
type bar struct {
    a *atom
    monad
    // ...
}

type baz struct {
    atom
    m *monad
    // ...
}
```

$\longrightarrow$

```go
a := &atom{...}
m := newMonad(...)

bar := newBar(a, m, ...)
baz := newBaz(a, m, ...)
```

Split

```
type bar struct {
    a *atom
    monad
    // ...
}

type baz struct {
    atom
    m *monad
    // ...
}
```

→

```
a := &atom{...}
m := newMonad(...)

bar := newBar(a, m, ...)
baz := newBaz(a, m, ...)
```

Externalize communication

```go
type bar struct {
    toBaz chan<- event
    // ...
}

type baz struct {
    fromBar <-chan event
    // ...
}
```

$\rightarrow$

```go
c := make(chan event)

bar := newBar(c, ...)
baz := newBaz(c, ...)
```

Externalize communication

```
type bar struct {
    toBaz chan<- event
    // ...
}

type baz struct {
    fromBar <-chan event
    // ...
}
```

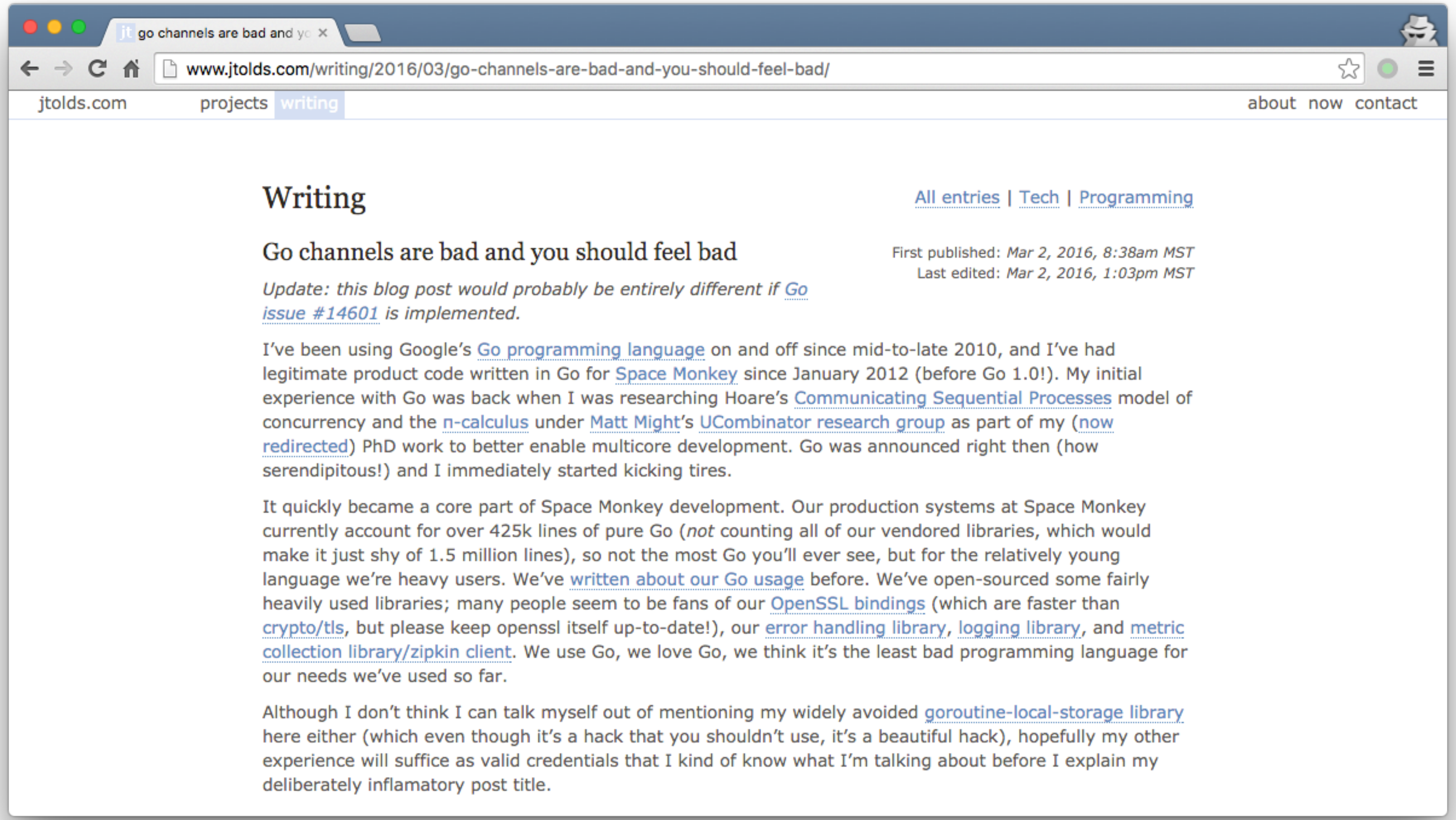
$\rightarrow$

```
c := make(chan event)

bar := newBar(c, ...)
baz := newBaz(c, ...)
```

X. Concurrency patterns

Channels are bad?


go channels are bad and yo

www.jtolds.com/writing/2016/03/go-channels-are-bad-and-you-should-feel-bad/

jtolds.com projects writing about now contact

Writing

All entries | Tech | Programming

Go channels are bad and you should feel bad

First published: *Mar 2, 2016, 8:38am MST*
Last edited: *Mar 2, 2016, 1:03pm MST*

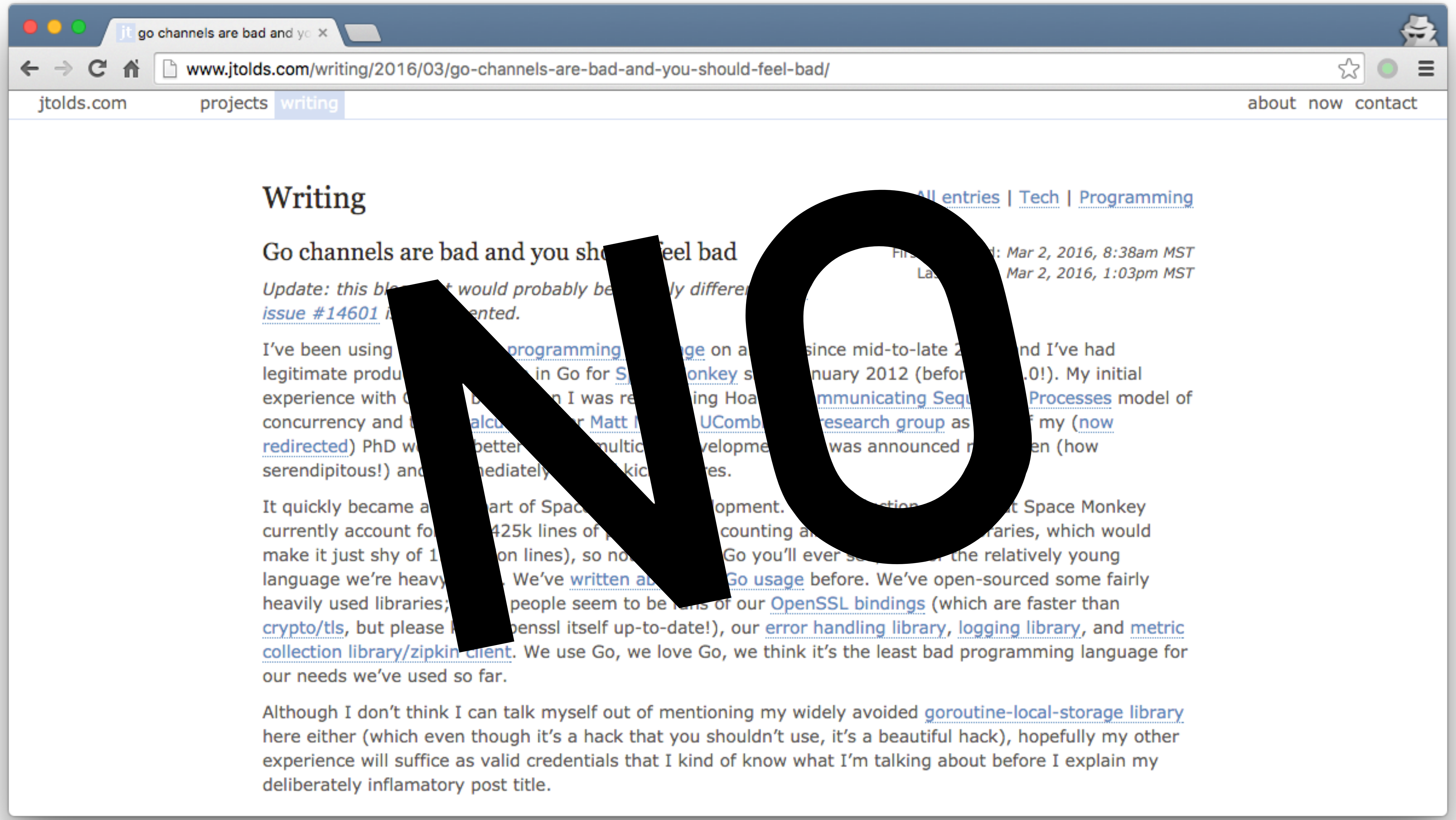
Update: this blog post would probably be entirely different if Go issue #14601 is implemented.

I've been using Google's Go programming language on and off since mid-to-late 2010, and I've had legitimate product code written in Go for Space Monkey since January 2012 (before Go 1.0!). My initial experience with Go was back when I was researching Hoare's Communicating Sequential Processes model of concurrency and the π-calculus under Matt Might's UCombinator research group as part of my (now redirected) PhD work to better enable multicore development. Go was announced right then (how serendipitous!) and I immediately started kicking tires.

It quickly became a core part of Space Monkey development. Our production systems at Space Monkey currently account for over 425k lines of pure Go (*not* counting all of our vendored libraries, which would make it just shy of 1.5 million lines), so not the most Go you'll ever see, but for the relatively young language we're heavy users. We've written about our Go usage before. We've open-sourced some fairly heavily used libraries; many people seem to be fans of our OpenSSL bindings (which are faster than crypto/tls, but please keep openssl itself up-to-date!), our error handling library, logging library, and metric collection library/zipkin client. We use Go, we love Go, we think it's the least bad programming language for our needs we've used so far.

Although I don't think I can talk myself out of mentioning my widely avoided goroutine-local-storage library here either (which even though it's a hack that you shouldn't use, it's a beautiful hack), hopefully my other experience will suffice as valid credentials that I kind of know what I'm talking about before I explain my deliberately inflamatory post title.

Channels are bad?


go channels are bad and yo
www.jtolds.com/writing/2016/03/go-channels-are-bad-and-you-should-feel-bad/
jtolds.com
projects
writing
about
now
contact
NO

Writing

All entries | Tech | Programming

Go channels are bad and you sh eel bad

Fir : Mar 2, 2016, 8:38am MST
La Mar 2, 2016, 1:03pm MST

Update: this bl t would probably be ly differe
issue #14601 i ented.

I've been using programming ge on a since mid-to-late 2 nd I've had
legitimate produ in Go for S onkey s nuary 2012 (befor .0!). My initial
experience with I was re ing Hoa mmunicating Seq Processes model of
concurrency and alcu r Matt UComb esearch group as my (now
redirected) PhD w better multic velopme was announced en (how
serendipitous!) an ediately kic res.

It quickly became a art of Spac opment. ction t Space Monkey
currently account fo 425k lines of counting a aries, which would
make it just shy of 1 on lines), so no Go you'll ever s the relatively young
language we're heavy We've written a Go usage before. We've open-sourced some fairly
heavily used libraries; people seem to be s of our OpenSSL bindings (which are faster than
crypto/tls, but please penssl itself up-to-date!), our error handling library, logging library, and metric
collection library/zipkin client. We use Go, we love Go, we think it's the least bad programming language for
our needs we've used so far.

Although I don't think I can talk myself out of mentioning my widely avoided goroutine-local-storage library
here either (which even though it's a hack that you shouldn't use, it's a beautiful hack), hopefully my other
experience will suffice as valid credentials that I kind of know what I'm talking about before I explain my
deliberately inflamatory post title.

Channels are fine

- Sharing memory between goroutines — use a mutex
- Orchestrating goroutines — use channels
- "Channels orchestrate; mutexes serialize."
  - **go-proverbs**.github.io

Good uses for a channel

```
semaphore := make(chan struct{}, 3)
for i := 0; i < 1000; i++ {
    go func() {
        semaphore <- struct{}{}
        defer func() { <-semaphore }()
        // process
    }()
}
```

Good uses for a channel

```go
resultc := make(chan int, n)

// Scatter
for i := 0; i < n; i++ {
    go func() {
        resultc <- process()
    }()
}

// Gather
for i := 0; i < n; i++ {
    fmt.Println(<-resultc)
}
```

Good uses for a channel

```
func (f *foo) set(k, v string) {
    f.setc <- setReq{k, v}
}

func (f *foo) get(k string) string {
    req := getReq{k, make(chan string)}
    f.getc <- req
    return <-req.res
}

func (f *foo) stop() {
    close(f.quitc)
}
```

```
func (f *foo) loop() {
    for {
        select {
        case req := <-f.setc:
            f.m[req.k] = req.v

        case req := <-f.getc:
            req.res <- f.m[req.k]

        case <-f.quitc:
            return
        }
    }
}
```

Good uses for a channel

```go
func (f *foo) set(k, v string) {
    f.actionc <- func() {
        f.m[k] = v
    }
}

func (f *foo) get(k string) (v string) {
    done := make(chan struct{})
    f.actionc <- func() {
        v = f.m[k]
        close(done)
    }
    <-done
    return v
}
```

```go
func (f *foo) loop() {
    for {
        select {
        case fn := <-f.actionc:
            fn()

        case <-f.quitc:
            return
        }
    }
}
```

Good uses for a channel

```go
func (f *foo) set(k, v string) {
    f.actionc <- func() {
        f.m[k] = v
    }
}

func (f *foo) get(k string) (v string) {
    done := make(chan struct{})
    f.actionc <- func() {
        v = f.m[k]
        close(done)
    }
    <-done
    return v
}
```

TOP TIP

```go
func (f *foo) loop() {
    for {
        select {
        case fn := <-f.actionc:
            fn()

        case <-f.quitc:
            return
        }
    }
}
```

Bad uses for a channel

```
type foo struct {
    m     map[string]string
    setc  chan setReq
    getc  chan getReq
    quitc chan struct{}
}
```

Bad uses for a channel

```
type foo struct {
    m   map[string]string
    mtx sync.RWMutex
}
```

Bad uses for a channel

```
func iterator() (<-chan string) {
    // ...
}
```

Bad uses for a channel

```
func iterator(cancel <-chan struct{}) (<-chan string) {
    // ...
}
```

Bad uses for a channel

```
func iterator() (results <-chan string, cancel chan<- struct{}) {
    // ...
}
```

Bad uses for a channel

```
func iterator(results chan<- string, cancel <-chan struct{}) {
    // ...
}
```

Bad uses for a channel

```
func iterator(f func(item) error) {
    // ...
}
```

Bad uses for a channel

```
func iterator(f func(item) error) {
    // ...
}
```

TOP TIP

Construction

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Be explicit

```go
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Be explicit

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

DEPENDENCIES

TOP TIP

MAKE DEPENDENCIES EXPLICIT

TOP TIP

Dependencies

```
func (f *foo) process() {
    fmt.Fprintf(f.Output, "beginning\n")
    result := f.Bar.compute()
    log.Printf("bar: %v", result)
    // ...
}
```

Dependencies

```
func (f *foo) process() {
    fmt.Fprintf(f.Output, "beginning\n")
    result := f.Bar.compute()
    log.Printf("bar: %v", result)
    // ...
}
```

Dependencies

Not a dependency

Dependency

Dependency

```
func (f *foo) process() {
    fmt.Fprintf(f.Output, "beginning\n")
    result := f.Bar.compute()
    log.Printf("bar: %v", result)
    // ...
}
```

Dependencies

```
func (f *foo) process() {
    fmt.Fprintf(f.Output, "beginning\n")
    result := f.Bar.compute()
    f.Logger.Printf("bar: %v", result)
    // ...
}
```

Dependencies

TOP TIP

```
func (f *foo) process() {
    fmt.Fprintf(f.Output, "beginning\n")
    result := f.Bar.compute()
    f.Logger.Printf("bar: %v", result)
    // ...
}
```

Dependencies

```
foo, err := newFoo(*fooKey, fooConfig{
    Bar:    bar,
    Baz:    baz,
    Period: 100 * time.Millisecond,
    Logger: log.NewLogger(dst, ...),
})
if err != nil {
    log.Fatal(err)
}
defer foo.close()
```

Dependencies

```
func newFoo(..., cfg fooConfig) *foo {
    if cfg.Output == nil {
        cfg.Output = ioutil.Discard
    }
    if cfg.Logger == nil {
        cfg.Logger = log.NewLogger(ioutil.Discard, ...)
    }
    // ...
}
```

TOP TIP

MAKE DEPENDENCIES EXPLICIT

TOP TIP

5. Logging and instrumentation

Logging

- More expensive than you think
- Actionable info only — read by humans or consumed by machines
- Avoid many levels — info+debug is fine
- Use structured logging — key=val
- Loggers are dependencies, not globals!

Instrumentation

- Cheaper than you think
- Instrument every significant component of your system
    - Resource — Utilization, Saturation, Error count (USE, Brendan Gregg)
    - Endpoint — Request rate, Error rate, Duration (RED, Tom Wilkie)
- Use Prometheus
- Metrics are dependencies, not globals!

Logging and instrumentation

- blog.raintank.io/logs-and-metrics-and-graphs-oh-my
  - bit.ly/**GoLogsAndMetrics**
- peter.bourgon.org/blog/2016/02/07/logging-v-instrumentation.html
  - bit.ly/**GoLoggingVsInstrumentation**

Global state

- log.Print uses a fixed, global log.Logger
- http.Get uses a fixed, global http.Client
- database/sql uses a fixed, global driver registry
- func init exists only to have side effects on package-global state

Global state

- log.Print uses a fixed, global log.Logger
- http.Get uses a fixed, global http.Client
- database/sql uses a fixed, global driver registry
- func init exists only to have side effects on package-global state

Eliminate implicit global deps

```go
func foo() {
    resp, err := http.Get("http://zombo.com")
    // ...
}
```

Eliminate implicit global deps

```
func foo(client *http.Client) {
    resp, err := client.Get("http://zombo.com")
    // ...
}
```

Eliminate implicit global deps

```
func foo(doer Doer) {
    req, _ := http.NewRequest("GET", "http://zombo.com", nil)
    resp, err := doer.Do(req)
    // ...
}
```

Eliminate global state

```
var registry = map[string]*http.Client{}

func init() {
    registry["default"] = &http.Client{}
}

func main() {
    if cond {
        registry[key] = otherClient
    }
    // ...
    exec(driver)
}
```

```
func exec(driver string) {
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    // ...
}
```

Eliminate global state

```
func init() {
    registry["default"] = &http.Client{}
}

func main() {
    var registry = map[string]*http.Client{}
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    exec(driver)
}
```

```
func exec(driver string) {
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    // ...
}
```

Eliminate global state

```go
func init() {
    //
}

func main() {
    registry := map[string]*http.Client{}
    registry["default"] = &http.Client{}
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    exec(driver)
}
```

```go
func exec(driver string) {
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    // ...
}
```

Eliminate global state

```
func init() {
    //
}

func main() {
    registry := map[string]*http.Client{
        "default": &http.Client{},
    }
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    exec(driver)
}
```

```
func exec(driver string) {
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    // ...
}
```

Eliminate global state

TOP TIP

```
func main() {
    registry := map[string]*http.Client{
        "default": &http.Client{},
    }
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    exec(driver)
}
```

```
func exec(driver string) {
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    // ...
}
```

Eliminate global state

```
func main() {
    registry := map[string]*http.Client{
        "default": &http.Client{},
    }
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    exec(driver)
}
```

```
func exec(driver string) {
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    // ...
}
```

Eliminate global state

```go
func main() {
    registry := map[string]*http.Client{
        "default": &http.Client{},
    }
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    exec(driver, registry)
}
```

```go
func exec(
    driver string,
    registry map[string]*http.Client,
) {
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    // ...
}
```

Eliminate global state

```
func main() {
    registry := map[string]*http.Client{
        "default": &http.Client{},
    }
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    exec(driver, registry)
}
```

```
func exec(
    client *http.Client,
) {
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    // ...
}
```

Eliminate global state

```
func main() {
    registry := map[string]*http.Client{
        "default": &http.Client{},
    }
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    exec(driver, registry)
}
```

```
func exec(
    client *http.Client,
) {
    // ...
}
```

Eliminate global state

```
func main() {
    registry := map[string]*http.Client{
        "default": &http.Client{},
    }
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    exec(client)
}
```

```
func exec(
    client *http.Client,
) {
    // ...
}
```

Eliminate global state

```go
func main() {
    registry := map[string]*http.Client{
        "default": &http.Client{},
    }
    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    exec(client)
}
```

```go
func exec(client *http.Client) {
   // ...
}
```

Eliminate global state

```go
func main() {
    client := &http.DefaultClient{}


    // ...
    if cond {
        registry[key] = otherClient
    }
    // ...
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    exec(client)
}
```

```go
func exec(client *http.Client) {
   // ...
}
```

Eliminate global state

```
func main() {
    client := &http.DefaultClient{}


    // ...
    if cond {
        client = otherClient
    }
    // ...
    client := registry[driver]
    if client == nil {
        client = registry["default"]
    }
    exec(client)
}
```

```
func exec(client *http.Client) {
   // ...
}
```

Eliminate global state

```
func main() {
    client := &http.DefaultClient{}


    // ...
    if cond {
        client = otherClient
    }
    // ...





    exec(client)
}
```

```
func exec(client *http.Client) {
   // ...
}
```

Eliminate global state

```go
func main() {
    client := &http.DefaultClient{}
    // ...
    if cond {
        client = otherClient
    }
    // ...
    exec(client)
}
```

```go
func exec(client *http.Client) {
   // ...
}
```

6. Testing

Testing

- Testing is programming — nothing special
- package testing continues to be well-suited to the task
- TDD/BDD packages bring new, unfamiliar DSLs and structures
- You already have a language for writing tests — called Go

Design for testing

- Write code in functional style
- Take dependencies explicitly, as parameters
- Avoid depending on or mutating global state!
- Make heavy use of interfaces

Design for testing

```
func process(db *database) (result, error) {
    rows, err := db.Query("SELECT foo")
    if err != nil {
        return result{}, err
    }
    defer rows.Close()
    var r result
    if err := rows.Scan(&r); err != nil {
        return result{}, err
    }
    return r, nil
}
```

```
func main() {
    db := newDatabase()
    r, err := process(db)
}
```

Design for testing

```
func process(db *database) (result, error) {
    rows, err := db.Query("SELECT foo")
    if err != nil {
        return result{}, err
    }
    defer rows.Close()
    var r result
    if err := rows.Scan(&r); err != nil {
        return result{}, err
    }
    return r, nil
}
```

```
func main() {
    db := newDatabase()
    r, err := process(db)
}

type queryer interface {
    Query(s string) (rows, error)
}
```

Design for testing

```
func process(q queryer) (result, error) {
    rows, err := db.Query("SELECT foo")
    if err != nil {
        return result{}, err
    }
    defer rows.Close()
    var r result
    if err := rows.Scan(&r); err != nil {
        return result{}, err
    }
    return r, nil
}
```

```
func main() {
    db := newDatabase()
    r, err := process(db)
}

type queryer interface {
    Query(s string) (rows, error)
}
```

Design for testing

```go
type fakeQueryer struct{}

func (q fakeQueryer) Query(s string) (rows, error) {
    return []row{"fakerow"}, nil
}
```

Design for testing

```
func TestProcess(t *testing.T) {
    q := fakeQueryer{}
    have, err := process(q)
    if err != nil {
        t.Fatal(err)
    }
    want := result{"fakedata"} // or whatever
    if want != have {
        t.Errorf("process: want %v, have %v", want, have)
    }
}
```

Design for testing

```
func process(q queryer) (result, error) {
    rows, err := db.Query("SELECT foo")
    if err != nil {
        return result{}, err
    }
    defer rows.Close()
    var r result
    if err := rows.Scan(&r); err != nil {
        return result{}, err
    }
    return r, nil
}
```

```
func main() {
    db := newDatabase()
    r, err := process(db)
}

type queryer interface {
    Query(s string) (rows, error)
}
```

TOP TIP

Design for testing

TOP TIP

```
func process(q queryer) (result, error) {
    rows, err := db.Query("SELECT foo")
    if err != nil {
        return result{}, err
    }
    defer rows.Close()
    var r result
    if err := rows.Scan(&r); err != nil {
        return result{}, err
    }
    return r, nil
}
```

```
func main() {
    db := newDatabase()
    r, err := process(db)
}

type queryer interface {
    Query(s string) (rows, error)
}
```

7. Dependency management

Dependency management

- Vendoring is still the solution
- GO15VENDOREXPERIMENT is the future — use it
- The tools have gotten a lot better

Dependency management

- github.com/FiloSottile/**gvt** — minimal, copies manually
- github.com/dpw/**vendetta** — minimal, via git submodules
- github.com/Masterminds/**glide** — maximal, manifest + lock file
- github.com/constabulary/**gb** — go tool replacement for binaries

Dependency management

- github.com/FiloSottile/**gvt** — minimal, copies manually

TOP TIP

- github.com/dpw/**vendetta** — minimal, via git submodules
- github.com/Masterminds/**glide** — maximal, manifest + lock file
- github.com/constabulary/**gb** — go tool replacement for binaries

Caveat for libraries… !

- Dependency management is a concern of the binary author
- Libraries with vendored deps are very difficult to use
- In general, **libraries should not vendor dependencies**
- If your library has hermetically-sealed deps — proceed with caution

Caveat for libraries… !

- Dependency management is a concern of the binary author
- Libraries with vendored deps are very difficult to use
- In general, **libraries should not vendor dependencies**
- If your library has hermetically-sealed deps — proceed with caution

TOP TIP

8. Build and deploy

Build

- Prefer go install to go build
- If you produce a binary, your responsibilities have grown
  - Don't be afraid of new approaches to manage complexity — **gb**
- Since Go 1.5 cross-compilation is built-in — no need for extra tools

Deploy

- We have it relatively easy
- If you deploy in containers — FROM scratch
- Think carefully before choosing a platform or orchestration system
- An elegant monolith is very productive

Deploy

- We have it relatively easy
- If you deploy in containers — FROM scratch

TOP TIP

- Think carefully before choosing a platform or orchestration system
- An elegant monolith is very productive

Summary

Top Tips

- Put $GOPATH/bin in your $PATH
- Name github.com/yourname/foo/lib as "package foo"
- Name things well — bit.ly/GoNames
- Avoid using os.Getenv by itself for configuration
- Name packages for what they provide, not what they contain

Top Tips

- Never use the dot import
- Define and scope flags in func main
- Use struct literal initialization
- Avoid nil checks with no-op implementations
- Make the zero value useful, especially with config objects

Top Tips

- Consider modeling actor pattern (for/select) as a single chan of funcs
- Model iterators as functions that take callbacks
- MAKE DEPENDENCIES EXPLICIT
- Loggers are dependencies
- Init smells really, really bad

Top Tips

- Define client-side interfaces to represent consumer contracts
- Take dependencies as interfaces
- Use gvt, vendetta, glide, or gb to manage vendoring for your binary
- Probably don't use vendoring for your library
- If you deploy in containers — FROM scratch

Go kit

A toolkit for microservices

github.com/go-kit/kit

1 year · 41 contributors

weaveworks

http://weave.works

Thank you! Questions?

@peterbourgon