KRAKOW POLAND, OCT 25 2018

GO ADVICES

Oleg Kovalov

Allegro

Twitter:
oleg_kovalov

Github: cristaloleg

Me

- Gopher for ~3 years
- Open-source contributor
- Engineer at Allegro.pl core team
- Suddenly a speaker at conf

Twitter: @oleg_kovalov
Github: @cristaloleg

Allegro

- 120 million items
- 20 million accounts
- 1.2 million events per minute
- 700 microservices
- Super fast Bigcache on a Github

Sources for level-up

- Effective Go
- CodeReviewComments
- Stdlib

Go advices

List of advices and tricks for Go \ʕ◔ϖ◔ʔ/

https://github.com/cristaloleg/go-advices

Use raw strings

```
func main() {
    value := "ʕ◔ϖ◔ʔ"
    a := "{\"key\":\"" + value + "\"}"

    b := "{" +
        "\"key\":\"" + value + "\"" +
        "}"

    c := `{"key":"` + value + `"}`

    println(a == c && b == c)
    println(c)
}
// true
// {"key":"ʕ◔ϖ◔ʔ"}
```

Strings should be human readable

Use raw strings for regex

```
func main() {
    value := "ʕ◔ϖ◔ʔ"

    a := "\"\\<h3\\>.*\\</h3\\>\""

    b := `"\<h3\>.*\</h3\>"`

    println(a)
    println(b)
}


// "\<h3\>.*\</h3\>"
// "\<h3\>.*\</h3\>"
```

REgexp means readable

Just a HTTP handler

```
type helloRequest struct {
    Name string `json:"name"`
    Toke string `json:"token"`
}

type helloResponse struct {
    Message string `json:"message"`
}

func helloHandler(...) {
    // process helloRequest/helloResponse
}

type confirmRequest struct { ... }
type confirmResponse struct { ... }
func confirmHandler(...) { ... }
```

The obvious way to declare req/resp :(

Function scope types

```
func helloHandler(...) {

    type request struct {
        Name string `json:"name"`
        Toke string `json:"token"`
    }
    type response struct {
        Message string `json:"message"`
    }

    var req request
    var resp response

    // process request/response
}
```

But we can do better :)

Anonymous types

```
func helloHandler(...) {
    type request struct {
        Name string `json:"name"`
        Toke string `json:"token"`
    }
    // process request

    var response = struct {
        Message string `json:"message"`
    }{
        Message: “yo, GoGoConf”,
    }
    // process request
}
```

or with an anonymous type :D

Don't be afraid to use defer

- Clear
  - You know what to do before an exit
- Reliable
  - No forgotten places
- Efficient
  - Performance impact is neglectable

Because someone said defer is slow

```
var mu sync.Mutex

func foo() interface{} {
    mu.Lock()

    if !flag {
        mu.Unlock()
        return nil
    }

    // do some stuff

    // oops…
    return value
}
```

Suddenly we have a locked mutex

Let’s do not omit a defer

```
var mu sync.Mutex

func foo() interface{} {
    mu.Lock()
    defer mu.Unlock()

    if !flag {
        return nil
    }

    // do some stuff

    // huh… everything is fine
    return value
}
```

And suddenly we have… no problems

What is the problem?

```
var i int64
var mu sync.RWMutex

http.HandleFunc("/add", func(...) {
     mu.Lock()
     defer mu.Unlock()
     i++

     fmt.Fprintf(w, "count: %d\r\n", i)
})

http.HandleFunc("/status", func(...) {
     mu.RLock()
     defer mu.RUnlock()

     fmt.Fprintf(w, "count: %d\r\n", i)
})
```

It’s race free.
I promise.

Slow client will kill our app

```
var i int64
var mu sync.RWMutex

http.HandleFunc("/add", func(...) {
     mu.Lock()
     defer mu.Unlock()
     i++

     fmt.Fprintf(w, "count: %d\r\n", i)
})

http.HandleFunc("/status", func(...) {
     mu.RLock()
     defer mu.RUnlock()

     fmt.Fprintf(w, "count: %d\r\n", i)
})
```

Over protected

IO operation should be cancelable or timeoutable

```
srv := &http.Server{
    Addr:           ":8080",
    Handler:        router,
    ReadTimeout:    3 * time.Second,
    WriteTimeout:   5 * time.Second,
    MaxHeaderBytes: 1 << 20,
}
srv.ListenAndServe()
```

Have a timeout for IO operations.

IO operation should be cancelable or timeoutable

```
// don't
var netClient = &http.DefaultClient{}

// do
var netClient = &http.Client{
  Timeout: 10 * time.Second,
}

// or even more pedantic

var netTransport = &http.Transport{
  Dial: (&net.Dialer{
    Timeout: 3 * time.Second,
  }).Dial,
  TLSHandshakeTimeout: 3 * time.Second,
}

netClient.Transport = netTransport
```

Omit http DefaultClient

Channel axioms

- A send to a nil channel blocks forever
- A send to a closed channel panics
- A receive from a nil channel blocks forever
- A receive from a closed channel returns the zero value immediately

Simple code, huh

```
func out() chan int {
     ch := make(chan int)
     go func() {
          var i int64
          for {
               ch <- i
               i++
          }
     }()
     return ch
}
func foo() {
     ch := out()

     for c := range ch {
          ...
     }
}
```

How to shoot yourself into the foot

Don't close in-channel

```
func out() chan int {
     ch := make(chan int)
     go func() {
          var i int64
          for {
               ch <- i  // *BANG*
               i++
          }
     }()
     return ch
}
func foo() {
     ch := out()
     // read from ch a bit and...
     close(ch) // and make a panic, probably
}
```

BANG

Specify a channel direction

```
func out() <-chan int {
     ch := make(chan int)
     go func() {
          var i int64
          for {
               ch <- i    // just send, right?
               i++
          }
     }()
     return ch
}
func foo() {
     ch := out()
     // read from ch a bit and...
     close(ch) // hah, compilation error
     // invalid operation: close(ch)
     // (cannot close receive-only channel)
}
```

One small arrow for developer
One giant leap for safety

Not so common for-loop

```
import "sync"

func main() {

     var greetOnce sync.Once

     for i := range [10]struct{}{} {

          println(i)

          greetOnce.Do(func() {
               print("hey, there!")
          })

     }
}

// 0hey, there!123456789
```

Don’t use it :)

Do something once with sync.Once

```
import "sync"

func main() {

     var greetOnce sync.Once

     for i := range [10]struct{}{} {

          println(i)

          greetOnce.Do(func() {
               print("hey, there!")
          })

     }
}

// 0hey, there!123456789
```

No comments

Avoid a global state

```
package cache

var keyValue map[string][]byte

func Get(k string) ([]byte, bool) {
     return keyValue[k]
}

func Set(k string, v []byte) {
     keyValue[k] = v
}

package service

func foo() {
     cache.Set(“conf”, “GoGoConf”)
}
```

Yeah, it works

But how to make it different?

```
package cache

var CacheType string

var keyValue map[string][]byte
var redisClient *redis.Client

func Get(k string) ([]byte, bool) {

     switch CacheType {
     case “inmemory”:
          return keyValue[k]
     case “redis”:
          return redisClient.Get(k)
     ...
     }
}
```

Oh..that’s hard

Let’s make it simpler

```
package inmemory

type Cache struct {
     data map[string][]byte
}

func New() *Cache {...}

func (c*Cache) Get(k string) ([]byte, bool) {...}
func (c*Cache) Set(k string, value []byte) {...}

package service

func foo() {
     c :=      inmemory.New()
     c.Set(“best conf”, “GoGoConf”)
}
```

And it’s
Easy to test

Let’s make it changable

```
package cache

type Cache interface {
     Get(k string) (value []byte, ok bool)
     Set(k string, value []byte)
}

package inmemory

type Cache struct {
     data map[string][]byte
}
func (c*Cache) Get(k string) ([]byte, bool) {...}
func (c*Cache) Set(k string, value []byte) {...}

package redis

// ...
```

Agile™

Unmarshal JSON

```
body := `{
     “id”: “i-am-so-random-woah”,
     “timestamp”: 123456789,
     “data”: {
          “sub_params”: {
               “oh_please_stop”: [{
                    ...
               }]
          },
          “tet_another”: {...},
     }
}`

var m map[string]interface{}

json.Unmarshal([]byte(body), &m) // slow :(

// and we need only id & timestamp
```

JSON from a production

We can make it type-safe

```
type Body struct {
     ID        string           `json:"id"`
     Timestamp int              `json:"timestamp"`
     Data      json.RawMessage  `json:"data"`
}

// encoding/json.go
// type RawMessage []byte

var m Body

json.Unmarshal([]byte(body), &m) // fast :)

timestamp := m.Timestamp
```

Lazy Unmarshalling

Table-driven tests

```
func TestSomething(t *testing.T) {
     testCases := []struct{
          name string
          a,b int64
          res int64
     }{
          {“simple case”, 1, 2, 3},
          {“less simple”, 3, 3, 23},
          {“omg”, 42, 78, 30307},
     }
     for _, tc := range testCases {

          t.Logf(“test: %s”, tc.name)

          res := foo(tc.a, tc.b)
          if res != tc.res {
               t.Errorf(“want %v, got %v, res, tc.res)
          }
     }
```

Table & test

Order independent test

```
func TestSomething(t *testing.T) {
     testCases := map[name]struct{
          a,b int64
          res int64
     }{
          “simple case”: {1, 2, 3},
          “less simple”: {3, 3, 23},
          “omg”: {42, 78, 30307},
     }
     for name, tc := range testCases {

          t.Logf(“test: %s”, name)

          res := foo(tc.a, tc.b)
          if res != tc.res {
               t.Errorf(“want %v, got %v, res, tc.res)
          }
     }
}
```

Hardening

Use build tags in tests

```
package service_test

var iTests = flag.Bool("integration", false, "<docs>")

func TestSomething(t *testing.T) {
     if !*iTests {
          t.Skip(“skipping test because of”)
     }

     if service.IsMeaningful() != 42 {
          t.Errorf(“oh no!”)
     }
}

// And run: go test ./... -args integration
```

And we need to edit every test like that

Use build tags in tests

```
// +build integration

package service_test

func TestSomething(t *testing.T) {
     if service.IsMeaningful() != 42 {
          t.Errorf(“oh no!”)
     }
}


// And run: go test --tags integration ./...
```

Voilà! ®

Use static code analysis tools

- golint
- go vet (enabled by default in Go 1.11)
- gometalinter
- golanci-lint
- go-critic 

That’s all folks

Thank you
Questions?

Twitter: @oleg_kovalov
Github: @cristaloleg