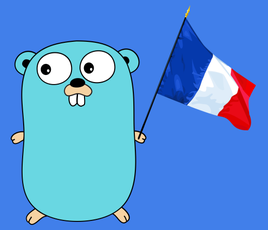
# 3 Erreurs Courantes en Go

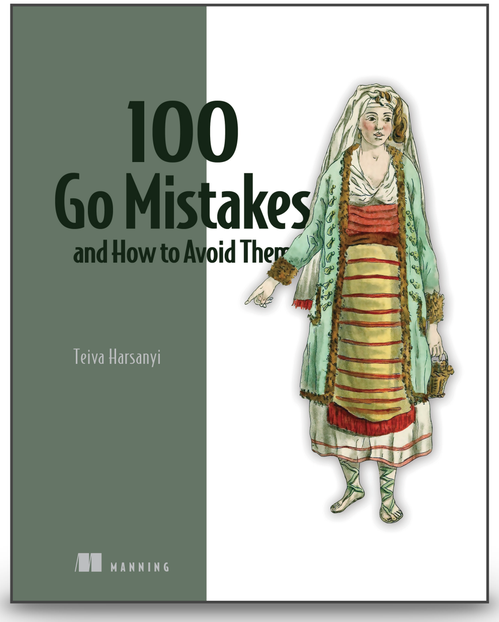
Golang Paris



10/01/2023

Teiva Harsanyi

Software Engineer at Docker 🐳



# #1
# La gestion d’un statut

```go
const (
  StatusSuccess  = "success"
  StatusErrorFoo = "error_foo"
  StatusErrorBar = "error_bar"
)

func f() error {
  var status string
  defer notify(status)
  defer incrementCounter(status)

  if err := foo(); err != nil {
    status = StatusErrorFoo
    return err
  }

  if err := bar(); err != nil {
    status = StatusErrorBar
    return err
  }

  status = StatusSuccess
  return nil
}
```

empty string “”

Quel est le problème ? ⏳

⚠️ Si l’on exécute cette function, le statut passé à notify et incrementCounter sera toujours le même: “” (empty string)

=> Les arguments passés à une fonction defer sont évalués **immédiatement**

@teivah

```go
const (
  StatusSuccess  = "success"
  StatusErrorFoo = "error_foo"
  StatusErrorBar = "error_bar"
)

func f() error {
  var status string
  defer notify(status)
  defer incrementCounter(status)

  if err := foo(); err != nil {
    status = StatusErrorFoo
    return err
  }

  if err := bar(); err != nil {
    status = StatusErrorBar
    return err
  }

  status = StatusSuccess
  return nil
}
```

## ✅ Solution 1: Si l'on peut changer notify et incrementCounter

```go
func f() error {
  var status string
  defer notify(&status)
  defer incrementCounter(&status)

  if err := foo(); err != nil {
    status = StatusErrorFoo
    return err
  }

  if err := bar(); err != nil {
    status = StatusErrorBar
    return err
  }

  status = StatusSuccess
  return nil
}
```

```go
const (
  StatusSuccess  = "success"
  StatusErrorFoo = "error_foo"
  StatusErrorBar = "error_bar"
)

func f() error {
  var status string
  defer notify(status)
  defer incrementCounter(status)

  if err := foo(); err != nil {
    status = StatusErrorFoo
    return err
  }

  if err := bar(); err != nil {
    status = StatusErrorBar
    return err
  }

  status = StatusSuccess
  return nil
}
```

## ✅ Solution 2: En utilisant une closure

```go
i := 0
j := 0
defer func(i int) {
  fmt.Println(i, j)
}(i)
i++
j++
```

```
0 1
```

@teivah

```go
const (
  StatusSuccess  = "success"
  StatusErrorFoo = "error_foo"
  StatusErrorBar = "error_bar"
)

func f() error {
  var status string
  defer notify(status)
  defer incrementCounter(status)

  if err := foo(); err != nil {
    status = StatusErrorFoo
    return err
  }

  if err := bar(); err != nil {
    status = StatusErrorBar
    return err
  }

  status = StatusSuccess
  return nil
}
```

## ✅ Solution 2: En utilisant une closure

```go
func f() error {
  var status string
  defer func() {
    notify(status)
    incrementCounter(status)
  }()

  if err := foo(); err != nil {
    status = StatusErrorFoo
    return err
  }

  if err := bar(); err != nil {
    status = StatusErrorBar
    return err
  }

  status = StatusSuccess
  return nil
}
```

@teivah

# #2
# L’implementation d’un cache

## Quel est le problème ? ⏳

⚠️ L'itération sur customers utilise une seule et unique variable customer avec une adresse fixe

```go
type Customer struct {
  ID      string
  Balance float64
}

type Store struct {
  m map[string]*Customer
}

func (s *Store) storeCustomers(customers []Customer) {
  for _, customer := range customers {
    s.m[customer.ID] = &customer
  }
}

func main() {
  s := Store{m: make(map[string]*Customer)}
  s.storeCustomers([]Customer{
    {ID: "1", Balance: 10},
    {ID: "2", Balance: -10},
    {ID: "3", Balance: 0},
  })
  // ...
}
```

```go
fmt.Printf("%p\n", &customer)
```

```
0xc000096020
0xc000096020
0xc000096020
```

```go
for k, v := range s.m {
  fmt.Printf("key=%v, value=%#v\n", k, v)
}
```

```
key=1, value=&main.Customer{ID:"3", Balance:0}
key=2, value=&main.Customer{ID:"3", Balance:0}
key=3, value=&main.Customer{ID:"3", Balance:0}
```

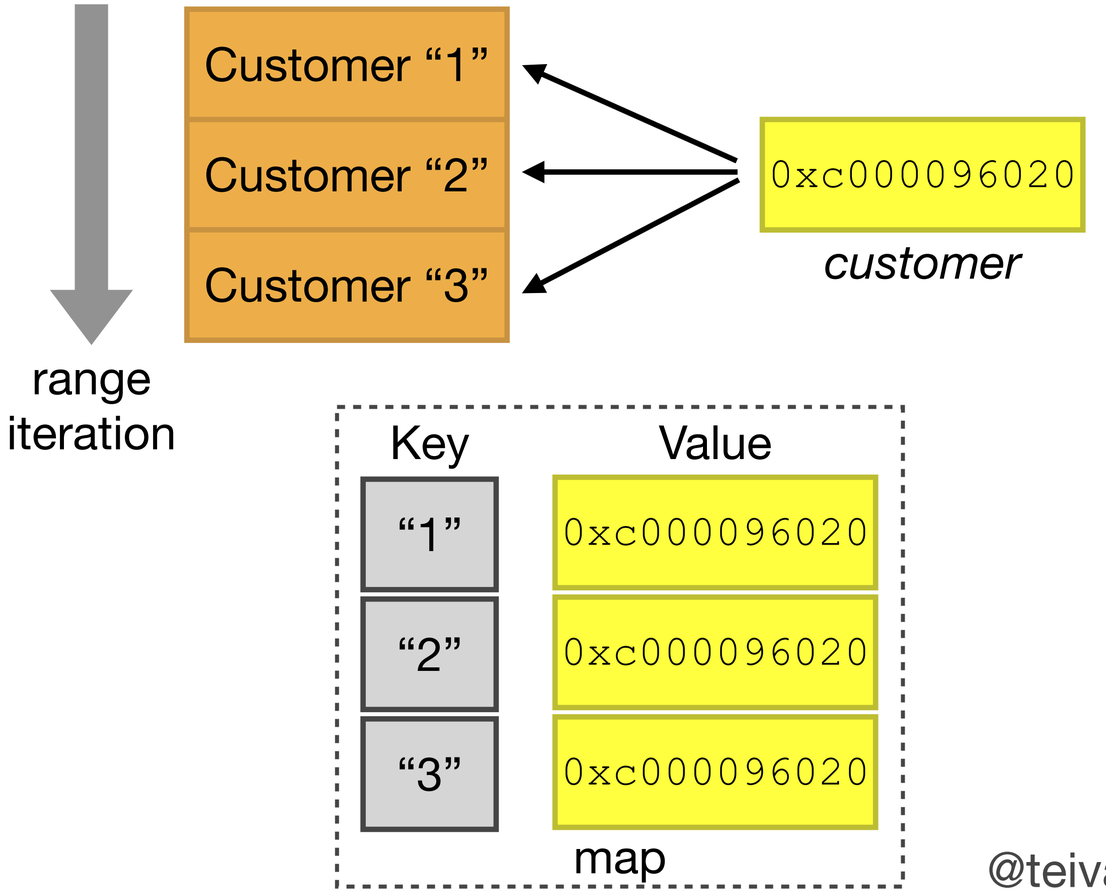
✅ ❌

@teivah

```go
type Customer struct {
  ID      string
  Balance float64
}

type Store struct {
  m map[string]*Customer
}

func (s *Store) storeCustomers(customers []Customer) {
  for _, customer := range customers {
    s.m[customer.ID] = &customer
  }
}

func main() {
  s := Store{m: make(map[string]*Customer)}
  s.storeCustomers([]Customer{
    {ID: "1", Balance: 10},
    {ID: "2", Balance: -10},
    {ID: "3", Balance: 0},
  })
  // ...
}
```

# Quel est le problème ? ⏳

⚠️ L'itération sur customers utilise une seule et unique variable customer avec une adresse fixe




@teivah

# ✅ Solution 1: Variable shadowing

```go
type Customer struct {
  ID      string
  Balance float64
}

type Store struct {
  m map[string]*Customer
}

func (s *Store) storeCustomers(customers []Customer) {
  for _, customer := range customers {
    s.m[customer.ID] = &customer
  }
}

func main() {
  s := Store{m: make(map[string]*Customer)}
  s.storeCustomers([]Customer{
    {ID: "1", Balance: 10},
    {ID: "2", Balance: -10},
    {ID: "3", Balance: 0},
  })
  // ...
}
```

```go
type Customer struct {
  ID      string
  Balance float64
}

type Store struct {
  m map[string]*Customer
}

func (s *Store) storeCustomers(customers []Customer) {
  for _, customer := range customers {
    customer := customer
    s.m[customer.ID] = &customer
  }
}

func main() {
  s := Store{m: make(map[string]*Customer)}
  s.storeCustomers([]Customer{
    {ID: "1", Balance: 10},
    {ID: "2", Balance: -10},
    {ID: "3", Balance: 0},
  })
  // ...
}
```

# ✅ Solution 2: Utilisation de l'index

```go
type Customer struct {
  ID      string
  Balance float64
}

type Store struct {
  m map[string]*Customer
}

func (s *Store) storeCustomers(customers []Customer) {
  for _, customer := range customers {
    s.m[customer.ID] = &customer
  }
}

func main() {
  s := Store{m: make(map[string]*Customer)}
  s.storeCustomers([]Customer{
    {ID: "1", Balance: 10},
    {ID: "2", Balance: -10},
    {ID: "3", Balance: 0},
  })
  // ...
}
```

```go
type Customer struct {
  ID      string
  Balance float64
}

type Store struct {
  m map[string]*Customer
}

func (s *Store) storeCustomers(customers []Customer) {
  for i := range customers {
    customer := &customers[i]
    s.m[customer.ID] = customer
  }
}

func main() {
  s := Store{m: make(map[string]*Customer)}
  s.storeCustomers([]Customer{
    {ID: "1", Balance: 10},
    {ID: "2", Balance: -10},
    {ID: "3", Balance: 0},
  })
  // ...
}
```

# #3
# Mettre à jour un âge

```go
type Customer struct {
  mutex sync.RWMutex
  id    string
  age   int
}

func (c *Customer) UpdateAge(age int) error {
  c.mutex.Lock()
  defer c.mutex.Unlock()

  if age < 0 {
    return fmt.Errorf("age is negative: %v", c)
  }

  c.age = age
  return nil
}

func (c *Customer) String() string {
  c.mutex.RLock()
  defer c.mutex.RUnlock()

  return fmt.Sprintf("id %s, age %d", c.id, c.age)
}
```

Quel est le problème ? ⏳

fmt.Stringer interface:

```go
type Stringer interface {
  String() string
}
```

❌ Deadlock

@teivah

# ✅ Solution: Locker au bon moment

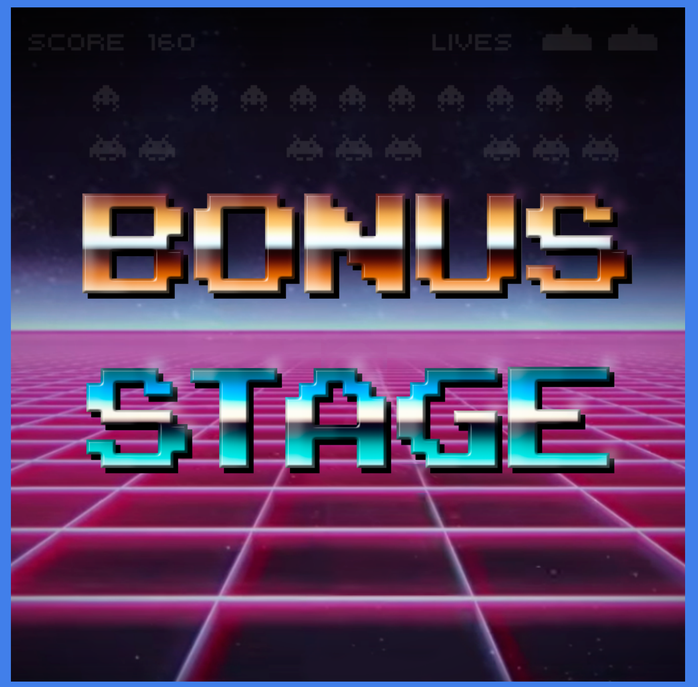
```go
type Customer struct {
  mutex sync.RWMutex
  id    string
  age   int
}

func (c *Customer) UpdateAge(age int) error {
  c.mutex.Lock()
  defer c.mutex.Unlock()

  if age < 0 {
    return fmt.Errorf("age is negative: %v", c)
  }

  c.age = age
  return nil
}

func (c *Customer) String() string {
  c.mutex.RLock()
  defer c.mutex.RUnlock()

  return fmt.Sprintf("id %s, age %d", c.id, c.age)
}
```

```go
type Customer struct {
  mutex sync.RWMutex
  id    string
  age   int
}

func (c *Customer) UpdateAge(age int) error {
  if age < 0 {
    return fmt.Errorf("age is negative: %v", c)
  }

  c.mutex.Lock()
  defer c.mutex.Unlock()

  c.age = age
  return nil
}

func (c *Customer) String() string {
  c.mutex.RLock()
  defer c.mutex.RUnlock()

  return fmt.Sprintf("id %s, age %d", c.id, c.age)
}
```

@teivah

# Conclusion

- On peut utiliser une closure pour *retarder* l'évaluation d'une variable qu'on passe à defer
- Attention lorsqu'on manipule des pointers avec une range loop
- Attention aux effets de bord avec du string formatting


SCORE 160
LIVES
BONUS
STAGE

Teiva Harsanyi
Apr 16, 2018 · 9 min read · Listen

# Good Code vs Bad Code in Golang

Recently, I was asked to detail what makes a good code or a bad code in **Golang**. I found this exercice very interesting. Actually, interesting enough to write a post about that. To illustrate my answer, I have taken a concrete use cases I faced in the Air Traffic Management (ATM) domain. The project is available in Github.

## Context

First, few words to explain the context of the implementation.

Eurocontrol is the organization managing the air traffic across Europe countries. The common network for exchanging data between Eurocontrol and an Air Navigation Service Provider (ANSP) is called AFTN. This network is mainly used to exchange two different message types: ADEXP and ICAO messages. Each message type has its own syntax but in terms of semantic, both types are equivalent (more or less). Given the context, **performance** must be a key element for the implementation.

This project has to provide two implementations for parsing ADEXP messages (ICAO is not managed in the frame of this exercise) based on Go:

- A bad implementation (package name: bad)
- A refactored implementation (package name: good)

Val Deleplace
May 29, 2018 · 9 min read · Listen

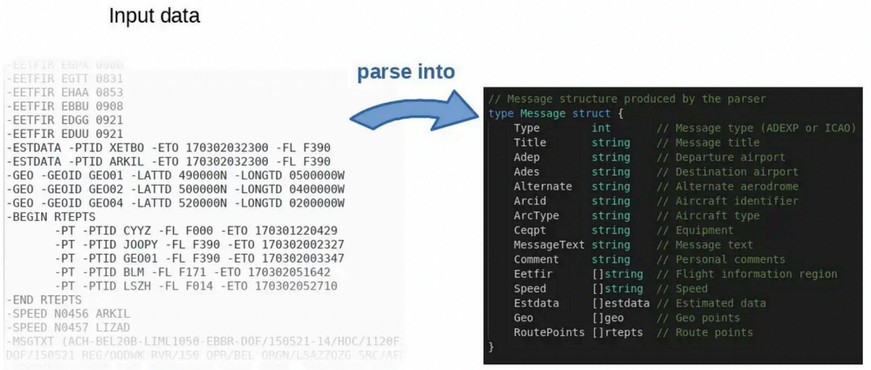
# Go code refactoring : the 23x performance hunt

A few weeks ago, I read an article called "Good Code vs Bad Code in Golang" where the author guides us step-by-step through the refactoring of an actual business use case.

The article focuses on turning "bad code" into "good code": more idiomatic, more legible, leveraging the specifics of the go language. But it also insists on **performance** being an important aspect of the project. This triggered my curiosity: let's dig in!

. . .

The program basically reads an input file, and parses each line to populate an object in memory.




The author not only published the source on Github, he also wrote an idiomatic benchmark. This was a really great idea, like an invitation to tweak the code and reproduce the measurements with the command:

- N’ayez pas honte de vos erreurs
- Ce qui compte au final, c’est notre capacité à **apprendre** de nos erreurs


Thank you