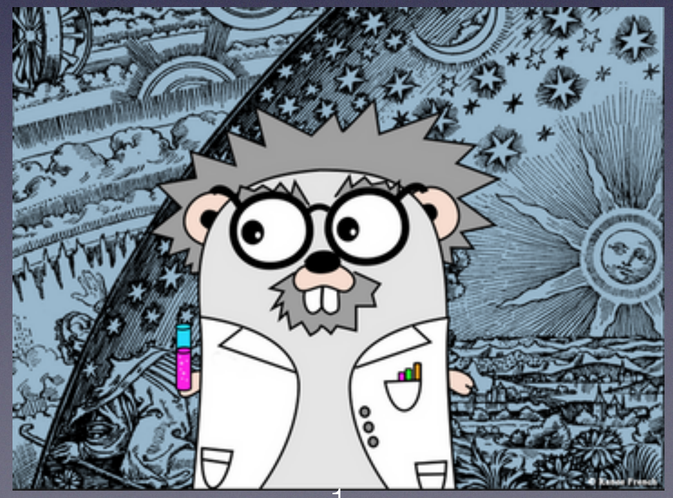
Go 1.14
Gestion des erreurs

Frédéric G. MARAND - Golang Paris Meetup - 26/02/2020



Il était une fois…

```
tar1, err := tarFromFiles(layer1Files...)
if err != nil {
 t.Fatal(err)
}

tar2, err := tarFromFiles(layer2Files...)
if err != nil {
 t.Fatal(err)
}

graph, err := newVFSGraphDriver(filepath.Join(td, "graphdriver-"))
if err != nil {
 t.Fatal(err)
}

graphID1 := stringid.GenerateRandomID()
if err := graph.Create(graphID1, "", nil); err != nil {
 t.Fatal(err)
}
```

https://github.com/docker/docker-ce/blob/129ece68573c3b2559a3894f143b9d922c551dc6/components/engine/layer/migration_test.go

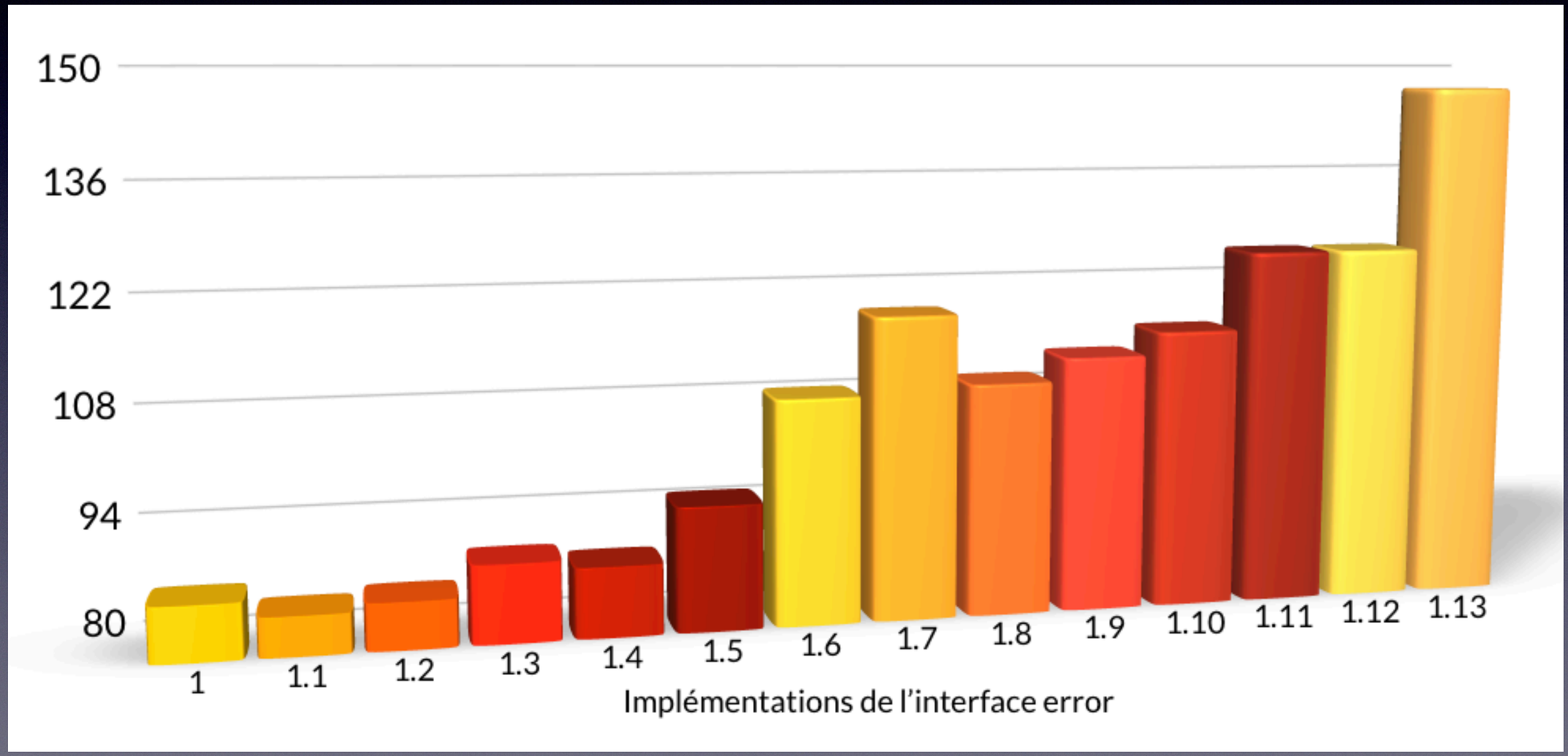
L’interface error

- Seul type interface fourni par le runtime Go
- Scope univers: pas d’import explicite
- Une seule méthode: `Error() string`
  - Hyper-simple à inclure dans autant de types qu’on veut
  - Possibilité de valeurs d’erreur significatives
- Convention d’utilisation: `nil` == pas d’erreur

Helpers

- `errors.New(text string) error`
- `fmt.Errorf(format string, a …interface{}) error`

De plus en plus populaire



« Errors are values.
Don’t just check errors, handle them gracefully. »

- Rob Pike - Go Proverbs
https://osinet.fr/go/builtins/yt-pike2015-errors

Go 2: les essais (1)

- `check` / `handle` (2018):
  - supprimer les `err != nil` en revenant directement
  - comme un `defer`, mais dédié aux résultats de `check`
- Verdict: syntaxe trop lourde, confusion entre handle et defer
  => Next!

Go 2: les essais (2)

- `try` / `defer` (2019):
  - même but: supprimer les `err != nil` en revenant directement
  - `try` au lieu de `check`
  - retours unifiés dans `defer`, plus besoin de `handle`.
- Verdict: la communauté n'a pas aimé. Que faire ?
- Russ Cox sur le sujet: https://osinet.fr/go/builtins/yt-cox2019-erreurs

« Errors are values.
Don’t just check errors, handle them gracefully. » (bis)

- Rob Pike - Go Proverbs
https://osinet.fr/go/builtins/yt-pike2015-errors

Le grand emballage

- Utiliser: prendre l’erreur là où elle survient et la renvoyer
- L’appelant rajoute son contexte, et la renvoie enrichie
- Et ainsi de suite
- C’est l’emballage (*error wrapping*)

Standardisation Go 1.13

- Fonction `errors.Unwrap(err error) error`
  - Renvoie l'erreur déballée si possible, `nil` sinon
  - En appelant la méthode `Unwrap()` des erreurs, sans interface associée
- `errors.Is(err, target error) bool`
  - Déballe jusqu'à `nil` ou `err==target` (`reflectlite`)
- `errors.As(err error, target interface{}) bool`
  - Tente une assertion de type en boucle de déballage

Mais pourquoi ?

- Dans le modèle classique, on ne peut comparer que l'erreur initiale
- Avec l'emballage/déballage, le test peut s'appliquer à l'erreur emballée sur n'importe quel niveau de déballage
- => Si on cible Go >= 1.13, toujours remplacer les tests et assertions de type par ces fonctions, moins bavardes

Helper

- `fmt.Errorf("%w", err)`
  - Applique `Unwrap` avant de faire le rendu
  - Donc ces erreurs supportent `errors.As` et `errors.Is`

Go 1.14 et plus

- Et la stack ?
  - Pendant le cycle de dév, on pouvait l'afficher, retiré en 1.13, pas réintroduit en 1.14
  - Porte ouverte pour l'avenir avec le type `errors.Frame`
  - https://osinet.fr/go/builtins/go-formatage-erreurs
- Et l'interface `Unwrapper` ?
  - Désaccord sur le nom: suggestion d'utiliser xerrors.Wrapper du paquet golang.org/x/xerrors

Spécificités Go 1.14

- `strconv` :
  - Le type `strconv.NumError` implémente `Unwrap() error`
  - Possibilité d'utiliser `errors.Is` et `errors.As` pour comparer à `errors.ErrRange` et `errors.ErrSyntax`

Et pour en savoir plus…

- 400 pages de Go en profondeur
- 160 listings

Au programme:

- Go: pour qui, pourquoi ?
- Syntaxe
- Types élémentaires
- Types composites
- Données
- Expressions
- Instructions
- Builtins
- Organisation du code
- Programmation concurrente