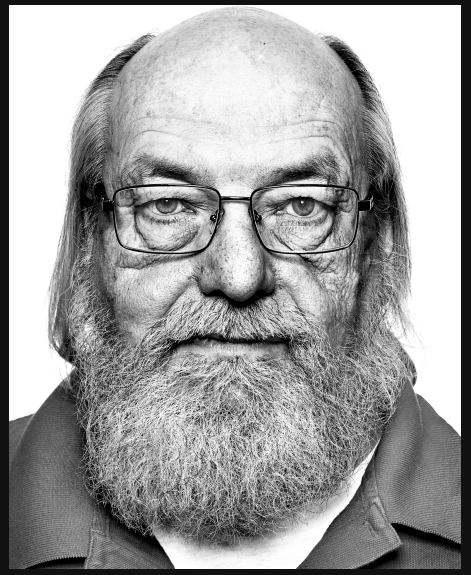
Golang

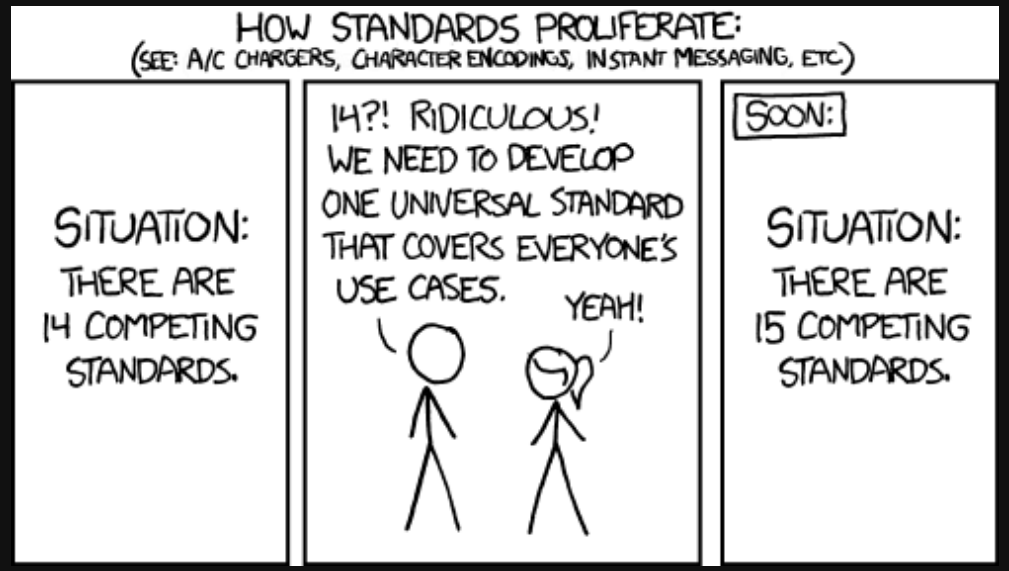
Une introduction pour DevOps - Part 1

Origines

- Né chez Google en 2007
- Principalement conçu par :



Pourquoi ?


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Pourquoi ?

- Temps de compilation du C++ trop longs
- Besoin de typage statique

```
def multiplyByFour(argsList):
  output = ????

  for arg in argsList:
    output += arg * 4

  return output
# (pour rappel, ceci est valide en python:
# argsList = ["name1", "long name1", 1, 2, 3])
```

Concrètement !

- Un langage à syntaxe "C-like", impératif
- Pas de POO ("Composition over inheritance")
- Compilé
- Statiquement typé (avec inférence de type)

Avantages

- Bibliothèque standard et des outils très complets
- Concurrence et parallélisme
- Garbage collection et sécurité mémoire
- Cross Compilation et static linking

Bibliothèque ?

Bibliothèque standard très complète (https://golang.org/pkg/) :

http, archives (tar, zip, bzip...), crypto, driver de DB, XML, JSON, hashing, testing, os...

Outils ?

Et des outils complets :

- go fmt (formatting automatique du code)
- go mod (vendoring)
- go doc
- go test
- go install ...

Voir ici : https://golang.org/cmd/go/

Go fmt

- Formatage automatique du code :
  - Aide à la standardisation de l'apparence du code
  - Évite les commits à "+800" parce que quelqu'un a reformaté dans son IDE
  - Permet aux autres outils go de manipuler le code

Go mod

- Vendoring :
  - Permet de s'affranchir de l'ancienne architecture (avec le $GOPATH)
  - Plus besoin de go get
  - Les dépendances sont définies dans un seul fichier "go.mod", et importées dans le code, "par fichier"

Go doc

- Documentation :
  - Est-ce que vous documentez vos scripts ?
  - Possibilité de récupérer de la documentation avec :

```
$ go doc [Symbol]
```

Go test

- Les tests :
  - Est-ce que vous testez vos scripts ?
  - Il suffit de placer ses tests dans des fichiers "[nom_fichier]_test.go", et de nommer les fonctions de test "Test_[nom_fonction]".
  - Lancer les tests avec :

```
$ go test ./
```

Go install

- Installation des paquets :
  - Même chose qu'un package manager classique type npm, cargo, pip...

```
$ go install https://github.com/FiloSottile/age
```

Garbage collection / Sécurité mémoire

- Pas de gestion directe des allocations mémoire
- Contrôle plus fin possible :
  - On a accès aux pointeurs, ou valeurs.

```
type Example struct{}

func newExample() Example{
  return Example{}
}

func newExamplePointer() *Example{
```

Exemple :

```go
package main
import "fmt"

func main() {
    message := "Hello Cosmos"

    // Pointer to string
    var pMessage *string


    // pMessage points to addr of message
    pMessage = &message
    fmt.Println("Message = ", *pMessage)
    fmt.Println("Message Address = ", pMessage)

    // Change message using pointer de-referencing
```

```
$ go run cosmiclearn.go
Message = Hello Cosmos
Message Address = 0xc04203e1d0
Message = Hello Universe
Message Address = 0xc04203e1d0
```

Garbage collection / Sécurité mémoire

- Pas de gestion directe des allocations mémoire
- Contrôle plus fin possible
- Sécurité mémoire : attention, les nil pointer exceptions (panics en Go) sont possibles !

Cross compilation & static linking

- Possibilité de compiler pour "n'importe quelle" plateforme. Lister les targets possibles :

```
$ go tool dist list
```

Cross compilation & static linking

- Les binaires sont "self contained", "statically linked" : aucune dépendance au runtime
- Pour choisir une plateforme, il suffit de définir une variable d'environnement GOARCH pour l'architecture, et GOOS pour la plateforme :

```
$ GOARCH=amd64 GOOS=linux go build ./...
```

Architecture d'un programme

- Un programme Go s'organise en packages
- Il y a une fonction main, dans le package main, qui sert à lancer le programme
- Un package est un "dossier" dans le code source qui sert à organiser le code

Packages

- Pour déclarer un package :

```
package main
```

- Pour utiliser un package :

```
import "MonModule/main"
```

Packages

- Visibilité :
  - Publique : commence avec une majuscule

```
func MyFunc(){}
```

  - Privée : minuscule

```
func myFunc(){}
```

Le code !

- Déclarer une variable
- Définir une fonction
- Définir une méthode
- Les types de données
  - Les types "value"
  - Les types "headers"
- Les slices

Déclarer une variable

- Déclaration simple, puis initialisation

```
var maChaine string
maChaine = "test"
```

- Déclaration et initialisation en une seule ligne

```
maChaine := "test"
```

Déclarer une fonction

- Mot clé "func"
- Nom de fonction
- Argument Type, séparés par des parenthèses
- Retours

```
func maFunc(arg1 string, arg2 string) (string, err){}
```

Pour les types identiques, on peut aussi écrire :

Définir une méthode

- Il faut ajouter un champ qu'on appelle "receiver"
- Attention au receiver ! Pointeur, ou valeur !

```
type Example struct{}
func (ex *Example) maFunc(arg1, arg2 string) (string, error){}
```

Les types de données

- Go a deux grand "types" de données :
  - Types "value"
  - Types "headers"
- Types "value"
  - Ce sont des types qui désignent directement la

Les types de données

- Types "headers"
  - Ce sont des types qui comportent des références vers les valeurs décrites.
  - Exemple : Les string. C'est un groupe de deux valeurs : une adresse sur le tas, et une longueur.
  - Autre exemple : Les slices.

Les slices

- Ce sont des "vues", sur des tableaux.
- C'est un type qui a trois valeurs :
  - Un pointeur vers le premier élément du tableau
  - Une longueur (length)
    - Nombre d'éléments auxquels on a accès depuis le pointeur vers le premier élément. On l'obtient avec len(maSlice)

```go
package main
import (
  "fmt"
)
func main() {
  orig:=[]string{"one", "two", "three", "four"}
  sl1:=test[0:2]

  sl2:=test[1:3]
  modifSlice(sl1, 1)
  fmt.Printf("Ma slice 2: %v", sl2)
}

func modifSlice(sl []string, idx int){
  sl[idx] = "modified"
}
```

Vrai tableau ?

Est-ce qu'on peut faire de "vrais" tableaux, et pas des slices ?

Oui ! Comme ça :

```
vraiTableau:=[2]int{1,8}
Pôle Universitaire
```

Définir son propre type

- On peut définir des types qui contiennent plusieurs champs, à la manière des classes.
- Attention, les types n'ont (presque toujours) que de la donnée à l'intérieur, pas de méthodes !
- Exemple :

Définir son propre type

On peut aussi définir des alias de types:

- types interchangeables :

```
type T = string
```

- nouveaux types :

```
type T string
```

Les "zero-value"

- En go, l'initialisation des variables est automatique si elle n'est pas explicite
- Chaque type a une "zero-value", qui correspond à la valeur par défaut du type
- Exemples : 0 pour int, false pour bool, etc.
- Fonctionne pour tous types y compris structs, et "types header" !

Lecture des arguments

On peut lire les arguments passés à un programme par une commande comme :

```
$ go run main.go "image.png"
```

Lecture des arguments

Pour celà, il faut importer le package "os", et récupérer la slice de string "Args" :

```
argsWithProg := os.Args
argsWithoutProg := os.Args[1:]
arg := os.Args[3]
```

Premier programme !

- Dossier du projet

```
$ mkdir hello && cd hello
```

Premier programme !

- Initialisation d'un module : création du fichier go.mod

```
$ go mod init hello
```

Premier programme !

- Déclaration du package, et fonction main

```
package main
func main(){}
```

Premier programme !

- Imports et appel

```
package main
import "fmt"
func main(){
  fmt.Println("Hello, World!")
}
```

On itère !

Est-ce qu'on pourrait faire exécuter une commande système à notre programme ?

Essayons avec la commande "date", pour afficher la... date.

On itère !

Le package os/exec :

https://golang.org/pkg/os/exec/

- Le type Cmd:

```
Cmd represents an external command being prepared or run.
A Cmd cannot be reused after calling its Run, Output or CombinedOutput r
```

- La fonction Command:

```
func Command(name string, arg ...string)
```

- La méthode Output

```
func (c *Cmd) Output() ([]byte, error)
```

- Revenons à notre exemple

```
package main
import "fmt"
func main() {
  fmt.Println("Hello, World!")
}
```

- On modifie la fonction : Création de Cmd

```
package main
import "fmt"

func main(){
  out, err := exec.Command("date").Output()
  fmt.Println("Hello, World!")
}
```

- Gestion des erreurs :

```
package main
import "fmt"
func main(){

  out, err := exec.Command("date").Output()
  fmt.Println("Hello, World!")
}
```

Ce code ne compile pas !

- Affichage du message de sortie :

```
package main
import (
  "fmt"
  "log"
  "os/exec"
)

func main(){
  out, err := exec.Command("date").Output()
  if err!=nil {
    log.Fatal(err)
  }
  fmt.Printf("The date is: %s\n", out)
}
```

Pourquoi ça fonctionne ?

- "date" fonctionne parce qu'elle est disponible sous Windows et sur les systèmes type Unix.

Comment faire pour une commande qui dépend de l'OS ?

Réponse : la cross-compilation, et les instructions de compilation

- Comment spécifier au compilateur qu'on aimerait compiler tel code pour tel OS ?
  - En C par exemple, on peut mettre des headers.
  - En Go, on a plusieurs méthodes : noms de fichiers et annotations.

Instructions compilateur et noms de fichiers

- Il suffit de nommer ses fichiers avec un suffixe qui est le target de la plateforme pour laquelle on veut compiler.
- On a la liste des plateformes avec la commande donnée précédemment.

```
$ go tool dist list
```

- Exemple de structure

Fichier Linux (getCommand_linux.go) :

```
func GetCommand() []string{
return []string{"ifconfig", "-a"}
}
```

Fichier Windows (getCommand_windows.go) :

```
func GetCommand() []string{
return []string{"ipconfig", "/all"}
}
```

Fichier main (tronqué) :

```
cmd := GetCommand()
out,err := exec.Command(cmd[0], cmd[1:]...).Output()
```

- Instructions compilateur et annotations :
  - Il est également possible d'utiliser des annotations pour les instructions.
  - C'est plus flexible, et plus "propre".

- Instructions compilateur et annotations :
  - Supposons que l'on ait 4 targets différents à couvrir : MacOS, Windows (32bits), Openbsd, Linux.
  - Plutôt que de faire 4 fichiers, on factorise et on utilise des annotations comme ceci :

Fichier Unix (getCommand_unix.go) :

```
// +build linux darwin openbsd
func GetCommand() []string{
  return []string{"ifconfig", "-a"}
}
```

Fichier Windows (getCommand_windows.go):

```
// +build windows,386
func GetCommand() []string{
  return []string{"ipconfig", "/all"}
}
```

Fichier main: inchangé.

Référence : https://golang.org/cmd/go/#hdr-Build_constraints