Initiation à Go

Devoxx Paris, 19/04/2018

Bonjour 

Christopher Moreau

- **https://github.com/oupo42**

Eric Lefevre-Ardant

- **@elefevre**

Démarrage

Pour suivre cet atelier, vous pouvez commencer à

- **installer Go**
- **positionner la variable GOPATH**

Cloner notre repo Github pour les instructions d'installation et les exercices :

https://github.com/leboncoin/atelier-initiation-golang

...ou copier une des clefs USB.

Au menu ce soir

- About Go
- Java vs. Go
- Coder un Hello, World
- Coder un serveur HTTP
- Coder un client HTTP
- Coder avec des goroutines et des channels

Origines de Go

- Développé par une équipe d'ingénieurs chez Google
- Développé pour rendre l'environnement de travail plus productif
- Sortie de la version 1.0 en Mars 2012

Introduction

Le langage Go est

- Compilé
- Statiquement typé
- Garbage collecté
- Concurrent (Concurrence != Parallélisme)

Les outils

Go:

- **go build**
- **go run**
- **go test**
- **go fmt**
- **go get**

GoDoc:

- **Extrait et génère de la documentation pour les packages Go**
- **Similaire à JavaDoc**

Java vs. Go

Java

```
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

Go

```
package main

import "fmt"

func main() {
  fmt.Println("Hello, world!")
}
```

Ce que ça apporte (par rapport à Java)

- Les goroutines
- Les channels
- Valeurs de retour multiples
- Pas de machine virtuelle
- Un seul binaire

Ce qui suit va vous étonner

Les trucs bizarres quand on vient du Java

Pas de génériques

- Les collections de base sont fournies
- Elles sont instanciables pour tous les types d'objets
- Au pire, on a l'équivalent de `Object` en Java : `interface{}`

```
var arrayOfMyInterfaces [2]myInterface
var sliceOfStrings []string
var mapOfAnything map[string]interface{}
var channelOfInts chan int
```

Les exceptions

- Il n'y en a pas !
- A la place, on retourne une instance de Error
- Et on checke la valeur de retour.
- Souvent.
- Très souvent.
- Très, très souvent.

Les ~~exceptions~~ Errors

```
func f() (int, error) {
  return 0, errors.New("failed :-(")
}

func main() {
  result, err := f()
  if err != nil {
    fmt.Printf("It doesn't work: %s\n", err)
    return
  }

  fmt.Printf("Result: %d\n", result)
}
```

Pas d’héritage de classe

- *Composition over inheritance*
- Les champs anonymes ressemblent un peu à de l’héritage

```
type english struct {}

func (e *english) speak() { fmt.Println("Hello") }

type american struct {
  english
}

func main() {
  sp := &american{}
  sp.speak()
}
```

Les paramètres sont passés par valeur

- ...mais on a souvent besoin de les passer par pointeur
- c'est explicite dans le code
- il n'est pas toujours évident de savoir quand c'est nécessaire

Les paramètres sont passés par valeur

```
type english struct { sentence string }

func updateValue(en english) { en.sentence = "value" }

func updatePointer(en *english) { en.sentence = "pointer" }

func main() {
    e := english{}

    updateValue(e)
    fmt.Println(e.sentence) // -> <blank line>

    updatePointer(&e)
    fmt.Println(e.sentence) // -> “pointer”
}
```

L’équivalent d’ArrayList est le *slice*

```
func updateCell(s []string) {
  s[0] = "world"
}

func main() {
  strs := []string{}

  strs = append(strs, "hello")
  fmt.Println(strs) // “[hello]”

  updateCell(strs)
  fmt.Println(strs) // “[world]”
}
```

Et aussi...

- Pas de surcharge de fonction (mêmes paramètres)
- Accessibilité par capitalisation (`Speak()` / `speak()`)
- Pas de programmation fonctionnelle
- Compilateur très strict

Are you ready?

- On va écrire un serveur HTTP

Serveur HTTP

- Pour associer une route à une fonction

```
func HandleFunc(pattern string, handler func(ResponseWriter,
*Request))
```

- Pour écouter sur un numéro de port et bloquer la goroutine principale

```
func ListenAndServe(addr string, handler Handler) error
```

Serveur HTTP

leboncoin ❤ gin

```
app := initApp(apiHandler)
_ = ginx.ListenAndServe(app, ":6903")
if err != nil {
    log.WithError(err).Error("Server stopped")
}


func initApp(apiHandler apiHandler) *gin.Engine {
    app := gin.New()
    app.DELETE("/v1/users", apiHandler.delete)

    return app
}
```

Let’s code

1. Partie 1 : Hello World
2. Partie 2 : Serveur HTTP
    a. Sortie dans la console sur un appel
    b. Retour d’un body
    c. Paramètre de path
    d. Lecture d’un body

https://github.com/leboncoin/atelier-initiation-golang/

Ce qu’on a vu jusqu’à maintenant

1. La sortie dans la console
2. Les tests
3. La lib net/http
4. La manipulation de Request et ResponseWriter

Up next

- goroutines
- channels

Les goroutines

- comme des threads, en plus léger
- font tourner une fonction de façon concurrente à l'appelant
- échange de données par channel
- on démarre avec `go myFunction(params)`

Les goroutines

```
func say(s string) {
    for i := 0; i < 5; i++ {
      time.Sleep(100 * time.Millisecond)
      fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

Les channels

- on y écrit des données
- on y lit des données
- c'est bloquant
- structure `chan`
- opérateur `<-`

Les channels

```
c := make(chan int)

c <- 1 // envoie une valeur

i := <-c  // lit la valeur et l'assigne à une variable
fmt.Printf("result : %d\n", i)
```

Les channels

goroutines + channels

=

Let’s code

- Troisième partie
  - appel Get
  - lecture du body de la réponse
  - envoi d’un body
- Quatrième partie
  - appel d’un serveur (local)
  - plusieurs appels, les uns après les autres
  - appels dans des goroutines
  - récupération des résultats dans un channel

A retenir

- Le Go est simple à écrire et à lire
- Le Go est particulièrement utile pour écrire beaucoup de services HTTP
- Les Goroutines et les channels sont un progrès

Annexes : slides candidats à jeter

Les mots-clefs pour l’accessibilité

- Il n'y en a pas !
- Se base sur la capitalisation de la première lettre

```
type english struct{}
func (e *english) speak() { fmt.Println("Hello package-only") }

type English struct{}
func (e *English) Speak() { fmt.Println("Hello exported") }
```

L’appel d’une fonction sur une struct nil

- Ca n’échoue pas !
- Enfin… pas toujours

```
type english struct{
  sentence string
}
func (e *english) speak() { fmt.Println("Hello") }
func (e *english) speakSentence() { fmt.Println(e.sentence) }

func main() {
  var e *english     // e est nil
  e.speak()          // -> pas de NullPointer !
  e.speakSentence() // "panic: runtime error: invalid memory
address or nil pointer dereference"
}
```

La programmation fonctionnelle

- Il n'y en a pas !
- Sauf les fonctions qui sont de première classe

```
func httpHandler(w http.ResponseWriter, r *http.Request) {
    ...
}

http.HandleFunc("/", httpHandler)
```

func **HandleFunc**

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

Validation à la compilation

- Il y en a plein !
- Le compilateur est strict
- Très strict
- Trop strict
- C'est une feature

```
package main
import  "fmt"
func main() {
//    fmt.Println("Hello, Devoxx")
}

$ go build
prog.go:2:8: imported and not used: "fmt"
```

Pas de surcharge de fonction

```
func f(a int) {}
func f(b string) {}

prog.go:6:10: f redeclared in this block
    previous declaration at prog.go:5:10
```

Implémentation d'interface par duck typing

```
type speaker interface {
  speak()
}

type english struct{}

func (e *english) speak() { fmt.Println("Hello") }

func main() {
  var sp speaker

  sp = &english{}
  sp.speak()
}
```

Comment écrire un client HTTP

- package `net/http`
- fonctions utilitaires
  - `func Get(url string)`
  - `func Post(url string, contentType string, body io.Reader)`
- `func NewRequest(method, url string, body io.Reader)`