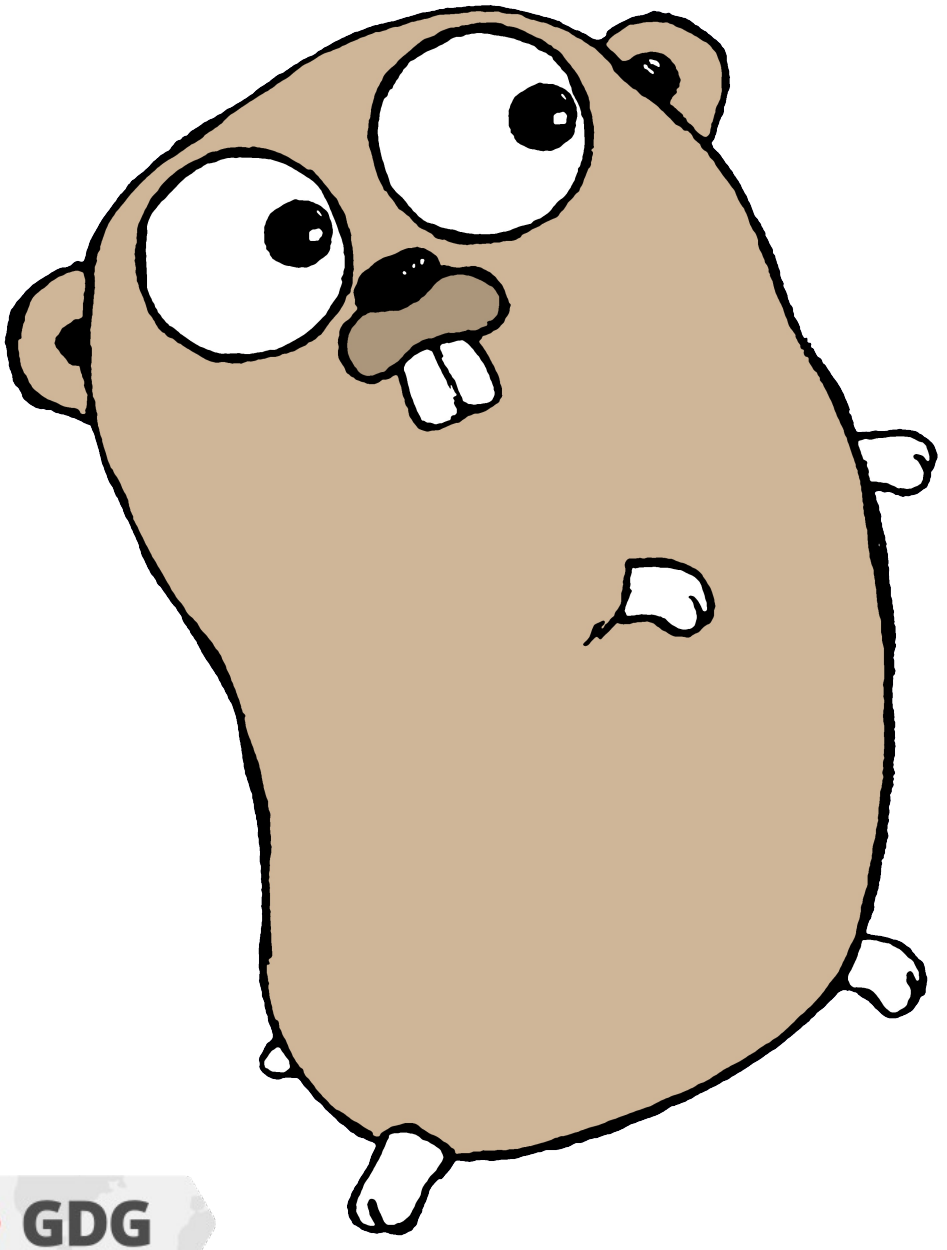
Introduction à Go

Tristan Louet

05/07/2013



Code d'Armor

UNIVERSITÉ DE ROUEN

GDG

Pourquoi un nouveau langage ?

- Les langages à typage statique sont efficaces, mais produisent un code source complexe, verbeux
- À l'inverse, les langages à typage dynamique permettent un développement facile et rapide, mais sont facilement source d'erreurs. De plus, ils sont moins efficaces, notamment pour les programmes déployés à grande échelle
- Programmation concurrentielle difficile

- « Vitesse, fiabilité, simplicité » ← choisissez-en deux

Qu'est ce que Go ?

- Un langage moderne et aux objectifs variés

- Compilation vers du langage machine natif (x86 32 et 64 bits, ARM)
- Typage statique
- Syntaxe légère (*lightweight*)
- Conçu pour la programmation concurrentielle (types primitifs et routines *built-in*)

Le classique

```
package main

import "fmt"

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

Hello, world 2.0

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello,"+r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

http://localhost:8080/world
http://localhost:8080/Lannion

Syntaxe et idiomes

new(Type) *Type

make(Type, IntegerType) Type

Pas de while, seulement for

Retour multivalué des fonctions

- Permet d'éviter les pointeurs en argument de fonction ne servant qu'à simuler ce comportement
- Permet la gestion des erreurs

panic(error) termine le programme à cause de l'erreur en paramètre

Système de typage simple

Go est statiquement typé, mais l'inférence des types évite les répétitions et allège le code

Java :

```
    SomeType v = new SomeType(...);
```

C/C++ :

```
    int i = 1;
```

Go :

i := 1 //Type int

pi := 3.142 //Type float64

hello:= "Hello, Lannion !" //Type string

Types et méthodes

Les méthodes sont définies sur les types :

```
type MyFloat float64

func (m MyFloat) Abs() float64 {
    f := float64(m)
    if f < 0 {
        return -f
    }
    return f
}
i := MyFloat(-42) //Type MyFloat
i.Abs() // == 42.0
```

Interfaces

Les interfaces spécifient un comportement
Elles font cela en définissant un ensemble de méthodes :

```
type Abser interface {
    Abs() float64
}
```

Un type qui implémente ces méthodes implémente alors implicitement l'interface :

```
func PrintAbs(a Abser) {
    Fmt.Printf("Absolute value : %2.f\n", a.Abs())
}
PrintAbs(MyFloat(-10))
```

Il n'y a aucune déclaration d'implémentation (implements en Java)

Un exemple pratique d'interface

Tiré du package io de la bibliothèque standard

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Il y a beaucoup d'implémentations de Writer dans la bibliothèque standard. Nous avons déjà vu un exemple :

```
func handler(w http.ResponseWriter, r ...) {
    fmt.Fprint(w, "Hello !")
}
```

fmt.Fprint prend un io.Writer en premier argument
http.ResponseWriter implémente la méthode Write

C'est pourquoi on peut utiliser la fonction Fprint, du package fmt, afin d'écrire la réponse dans le http.ResponseWriter.
fmt ne connaît rien de http, mais ça fonctionne.

Programmation concurrentielle

Sous un système UNIX, on a des **processus** connectés entre eux par des **tubes** (pipes) :

```
ps -aux | wc -l
```

Chaque processus est un outil conçu pour faire une seule chose et pour le faire bien.

L'analogie en Go se fait en utilisant des **goroutines** connectées entre elles par des **channels**.

Goroutines

Les goroutines sont assimilables à des threads.

Elles partagent de la mémoire, cependant sont bien moins coûteuses :

- Piles réduites à la création, segmentées
- Plusieurs goroutines par thread système

Démarrer une goroutine s'effectue simplement avec le mot clé **go**

```
i := pivot(s)
go sort(s[:i])
go sort(s[i:])
```

Channels

Les **channels** sont un type primitif assimilable au **tube** UNIX.

Ils permettent de :

- Communiquer
- Synchroniser

L'opérateur <- permet d'envoyer et de recevoir des valeurs :

```
func compute(ch chan int) {
    ch <- someFunction()
}
func main() {
    ch:= make(chan int)
    go compute(ch)
    result:= <-ch
}
```

Synchronisation

Intéressons nous à l'exemple de triage de slice :

Nous pouvons utiliser un channel pour synchroniser les goroutines :

```
func doSort(s []int, done chan bool) {
    sort(s)
    done <- true
}
func main() {
    ...
    done := make(chan bool)
    i := pivot(s)
    go doSort(s[:i], done)
    go doSort(s[i:], done)
    <- done
    <- done
}
```

Communication

Habituellement, lorsqu'il y a un grand nombre de tâches à accomplir, on utilise des Workers qui vont traiter les tâches de manière concurrencielle.

Voici la manière idiomatique de le faire en Go :

```
type Task struct {
    //...
}
func worker(in, out chan *Task) {
    for {
        t := <- in
        process(t)
        out <- t
    }
}
func main() {
    pending, done := make(chan *Task), make(chan *Task)
    //Remplir pending
    for i := 0; i < 10; i++ {
        go worker(pending, done)
    }
    //Lire done
}
```

Philosophie Go

- Les goroutines permettent d'avoir l'efficacité de la programmation concurrentielle (et potentiellement parallèle)
- Elles permettent un raisonnement simple : on écrit une fonction destinée à être une goroutine de sorte qu'elle effectue sont travail, puis on connecte les goroutines via des channels
- Elles produisent un code plus simple à lire et à maintenir

Une phrase résume parfaitement l'esprit de Go dans le modèle de la programmation concurentielle :

« Ne communiquez pas en partageant de la mémoire,

À la place, partagez la mémoire en communicant »

Bibliothèque

- Une bibliothèque standard riche, stable et cohérente
- Plus de 150 packages
- De plus en plus de projets externes
  - Gorilla toolkit
  - SDL/OpenGL bindings
  - Oauth
  - Mysql, MongoDB et SQLite3
  - Etc...

Open Source

- Développement commencé en 2007 chez Google
- Libéré sous licence BSD-style en Novembre 2009
- Depuis, plus de 130 contributeurs externes à Google ont soumit plus de 1000 modifications au noyau de Go
- Environ 10 employés Google travaillent sur Go à plein temps
- Deux personnes externes à Google sont "*committeurs*" sur le projet

Informations de 2011

Un exemple pratique

Exemple d'application réelle transférée du C++ vers Go :

http://www.angio.net/pi/

Pi-Search parse Pi à la recherche d'un nombre (une sous-chaîne de caractères) dans les 200 premiers millions de chiffres après la virgule.

Benchmarks :

Version	Ligne de code	Nombre de requêtes / sec (moyenne)	Temps moyen par requête (ms)
C++	765	45.76	21.851
Go 1	350	607.67	1.646
Go 1.1	350	823.89	-

Pour aller plus loin

Si vous êtes intéressés par Go, faîtes un tour sur le site officiel :

http://golang.org/

Ou sur le blog des développeurs :

http://blog.golang.org/

Sources

- Cette présentation est largement inspirée de celle donnée par Andrew Gerrand, développeur Go chez Google Sydney, lors de la I/O 2011

- PiSearch :

  http://da-data.blogspot.fr/2013/05/improving-pi-searchers-speed-by-moving.html
- Go documentation

  http://golang.org/doc/
- Vidéos de conférences par les développeurs Go
- Le blog de Dave Cheney

  http://dave.cheney.net/