Le langage de programmation Go

Stéphane Bortzmeyer
<stephane+enac@bortzmeyer.org>

19 avril 2011

Exposé libre

Pourquoi parler de Go ?

1. Situé sur un créneau où il y a peu de langages concurrents,
2. Robert Griesemer, Rob Pike, Ken Thompson, Ian Taylor, Google...

Positionnement de Go

Il existe des zillions de langages de programmation...

Certaines familles sont très encombrées (langages fonctionnels : Haskell, Erlang, famille ML...)

Go est sur le créneau des langages « systèmes et réseaux », où il n'y a guère que C et C++ (et Erlang ?). La précédente tentative sérieuse de détrôner C était D (peu de succès).

Go est donc positionné comme langage de relativement bas niveau. Ce n'est pas Python. Ce n'est pas Haskell.

Au commencement, était Hello, World

```
package main

import ("flag"; "fmt"; "os";)

func main() {
   if flag.NArg() != 1 {
      fmt.Printf("Usage: program name\n");
      os.Exit(1);
   }
   fmt.Printf("Hello, %s\n", flag.Arg(0));
}
```


Points importants du langage

1. Parallélisme (les *goroutines*),
2. Inférence de types,
3. Pas orienté objet (mais une fonction peut être attachée à un type),
4. Pas d'exceptions (mais les fonctions peuvent renvoyer plusieurs valeurs et il existe un mécanisme de *panic/recover*),
5. Vues (*slices*) sur les tableaux,
6. Et bien d'autres choses...

Parallélisme

Une des principales raisons de la conception de Go, selon les auteurs.

Toute fonction peut être exécutée en parallèle avec le mot-clé go :

```
go coffee_machine(chan_machine);
```

Très pratique pour les serveurs réseaux. Ici, un serveur echo :

```
for {   // ever...
        conn, error := listener.AcceptTCP();
        if error != nil {
             ...
             go handle(conn);
        }
        // Prêt pour le client suivant
```


Canaux de communication

Les goroutines communiquent par des canaux **typés**.

```
func coffee_machine(c chan bool) {
   // Do something
   c <- true;
}
// Dans une autre goroutine:
result <- chan_machine;
```

Autre possibilités sur les canaux

- Les canaux sont synchrones par défaut mais on peut les rendre asynchrones en indiquant la taille du tampon,
- On peut, avec `select`, écouter sur plusieurs canaux en même temps.

Inférence de types

Go est typé mais il n'est pas indispensable de déclarer les variables.

```
conn, error := net.Dial("tcp", "", "whois.nic.fr:43");
```

Et `conn` sera de type `net.Conn`, sans avoir eu besoin de le déclarer, car c'est ce que renvoie `net.Dial`. Ce type sera vérifié :

```
whois.go:30: invalid operation: conn + 4 (type net.Conn + int)
```

Pas orienté-objet

Ce n'est plus à la mode.

Mais on peut attacher une fonction à un type.

```
// Définition
func (t *Twitter) FriendsTimeline() (string, os.Error) {
...
// Utilisation
tw := twitter.NewTwitter(*username, *password);
timeline, error := tw.FriendsTimeline();
os.Stdout.WriteString(timeline);
```


Pas d'exceptions

Les auteurs les voient comme une structure de contrôle dangereuse.

Mais :

- Fonctions multi-valuées :

```
file, status := os.Open(filename, os.O_RDONLY, 0)
if status != nil {
      fmt.Printf("Cannot open \"%s\": %s\n", filename, status.String())
}
```

- Mécanisme *panic/recover*, moins général :

```
func clean() {
      status := recover()
      /* Do something */
...
defer clean()
...
panic("Something is horribly wrong")
```

Une vue sur le tableau

Les tableaux sont assez rigides et fixes.

Mais il y a les vues (*slices*) !

En Go, on utilise presque toujours une vue et pas le tableau sous-jacent.

```
binary.BigEndian.PutUint16(result[4:6], packet.Qdcount)
// result est le tableau, result[4:6] une vue (pas une copie)
```


Interfaces

Go, comme Java, a des interfaces : un ensemble de fonctions que doit mettre en œuvre un type, pour pouvoir être utilisé.

Très utilisé dans les bibliothèques. Exemple dans `io` :

```
/* Un type qui a une fonction d'écriture. Cela peut être un fichier,
une prise réseau, une chaîne de caractères, etc */
type Writer interface {
    Write(p []byte) (n int, err os.Error)
}
/* Et ensuite : */
func NewBufferedWriter(wr io.Writer) *BufferedWriter
/* Permet de ''bufferiser'' tout io.Writer */
```

La bibliothèque standard

Elle est très riche :

- Crypto
- Expressions rationnelles
- Réseau (et protocoles comme HTTP)
- XML
- Grands entiers
- mais pas encore de moyens standards de parler à un SGBD
- et pas grand'chose encore question interface utilisateur...

Exemple bibliothèque : XML

Accès aux données du VéloStar de Rennes, KR (Keolis-Rennes). Le paquetage XML utilisé la **réflexion** de Go :

```
type Station struct {
Id              int
        ...
Bikesavailable int
...
type ApiKR struct {
Request string
Answer  Answer
}
...
result := ApiKR{}
error := xml.Unmarshal(buffer, &result)
fmt.Printf("Bikes available: %d\n",
          result.Answer.Data.Station.Bikesavailable)
```

Autres bibliothèques

`http://godashboard.appspot.com/package`

Qualité variable...

Déjà un programme standard d'installation, goinstall

Mises en œuvre

- gc, le compilateur natif original. Pour Linux, FreeBSD et MacOS X. Génère du code pour amd64, i386 et ARM. Les goroutines sont mises sur des fils du système sous-jacent, selon la demande (il n'y a pas de correspondance univoque).
- gccgo, désormais inclus dans gcc (depuis la 4.6).
- erGo, écrit en Go, non libre, sur Windows.

Problèmes et limites

- Nom incherchable dans Google,
- Langage (et environnement) pas encore stabilisé, tout change tout le temps (il y a même un programme spécial pour mettre à jour les sources, gofix),
- Manque encore de bibliothèques **stables** pour des choses comme l'accès aux SGBD,
- Langage de bas niveau (et c'est fait exprès), donc on n'essaie pas de masquer la machine,
- Les chaînes de caractères sont en fait des chaînes d'octets (pénible pour Unicode, où on doit travailler avec des tableaux d'entiers).

Idiosyncrasies

- Variable non utilisée ? C'est interdit...

```
hello.go:10: name declared and not used
```

- Allocation mémoire avec `make` :

```
value := make([]byte, 1024)
```

Un serveur de noms en Go

Grong `http://github.com/bortzmeyer/grong` est un serveur de noms faisant autorité.

Il est séparé en deux parties, un **frontal** qui analyse les requêtes entrantes et génère des requêtes sortantes correctes. Et un **dorsal** qui produit la réponse à partir de la requête. Plusieurs dorsaux différents sont possibles comme un serveur AS112 ou bien un serveur qui renvoie au client l'adresse IP du résolveur.

Implémentation de Grong

Une fonction d'écoute pour TCP et une pour UDP, toutes les deux lancées en goroutines.

Chacune lance ensuite une goroutine par requête (UDP) ou par connexion (TCP).

Avec les goroutines, écrire un serveur réseau est un plaisir.

Autres exemples de DNS en Go

[Développés à SIDN]

1. GoDNS, une bibliothèque `https://github.com/miekg/godns`
2. Funkensturm, un relais DNS (capable par exemple de signer à la volée) `http://www.miek.nl/blog/archives/2011/01/23/funkensturm_a_versatile_dns_proxy/index.html`