Tutoriel de programmation Go

Par Brice Colucci

Date de publication : 25 décembre 2010

Cet article est un tutoriel d'introduction aux bases du langage de programmation Go.

Cet article est une traduction provenant de **http://golang.org/doc/go_tutorial.html** qui est sous licence Creative Commons Attribution 3.0.

N'hésitez pas à commenter cet article ! **Commentez** ♪

I - Introduction

Cet article est un tutoriel d'introduction aux bases du langage de programmation Go. Ce tutoriel est destiné aux programmeurs familiarisés avec le C ou le C++. Ce n'est pas un guide pas-à-pas (débutant) du langage. Pour le moment, ce qui se rapprocherait le plus de ce type de document est la spécification du langage. Après avoir lu ce tutoriel, vous pourrez parcourir « Le Go effectif » qui va plus loin dans la manière d'utiliser le langage. Des diapositives sont également disponibles pour un cours en 3 jours (en anglais) : **Jour 1**, **jour 2** et **jour 3**.

Ce tutoriel se déroule suivant une série de programmes modestes illustrant les points clés du langage. Tous les programmes fonctionnent (au moment de l'écriture de ce tutoriel) et sont versionnés dans le dossier /doc/progs/.

Les sources des programmes sont détaillées point par point.

II - Hello, World

Débutons d'une manière classique :

```
1. package main
2.
3. import fmt "fmt" // Package implementing formatted I/O.
4.
5. func main() {
6.   fmt.Printf("Hello, world; or Καλημέρα κόσμε; or ##### ##\n")
7. }
```

Chaque fichier source Go déclare de quel package il fait partie en utilisation une instruction de package. Le fichier peut aussi importer d'autres packages afin d'utiliser leurs fonctionnalités. Ce programme importe le package fmt pour avoir accès à notre bon vieil ami fmt.Printf.

Les fonctions sont introduites avec le mot clé **func**. La principale « fonction principale » du package est le point de commencement du programme (après toute initialisation).

Les chaînes constantes peuvent contenir des caractères Unicode UTF-8. (En fait les fichiers sources Go sont définis pour être encodés en UTF-8)

Les conventions pour les commentaires sont les mêmes qu'en C++ :

```
/* ... */
// ...
```

Nous aurons plus de choses à dire plus tard sur l'affichage.

III - Les points-virgules

Vous avez peut-être remarqué que notre programme n'a pas de points virgules. En Go, le seul endroit où vous avez besoin typiquement de points-virgules est la séparation des clauses des boucles. Ils ne sont pas nécessaires après chaque instruction.

En fait, le langage formel utilise les points-virgules, autant qu'en C ou en Java, mais ils sont insérés automatiquement à la fin de chaque ligne qui ressemble à une instruction. Vous n'avez pas besoin de les taper vous-même.

Pour les détails de comment cela est réalisé, reportez-vous à la spécification du langage, mais en pratique la seule chose que vous devez savoir est que vous n'avez jamais besoin de mettre un point virgule à la fin d'une ligne (vous pouvez en mettre si vous souhaitez mettre plusieurs instructions par ligne). Vous pouvez aussi enlever le point virgule juste avant une accolade fermante.

Cette approche épure le rendu visuel du code. La seule surprise est qu'il est important de mettre l'accolade ouvrante d'une structure, comme une instruction if, sur la même ligne que le if. Si vous ne le faites pas, il y a des situations qui pourront ne pas compiler ou qui donneront un résultat erroné. Le langage force cela dans une certaine mesure.

IV - La compilation

Go est un langage compilé. Pour le moment il existe deux compilateurs. Gccgo est un compilateur Go qui utilise GCC. Il y a aussi une suite de compilateurs de noms différents pour chaque architecture :

- 6g pour le 64-bit x86 ;
- 8g pour le 32-bit x86 ;
- et d'autres…

Ces compilateurs sont significativement plus rapides, mais génèrent un code moins efficace que Gccgo. Au moment de l'écriture de cet article, ils ont aussi un système d'exécution plus robuste.

Voici comment compiler et exécuter notre programme. Avec 6g :

```
$ 6g helloworld.go
$ 6l helloworld.6
$ 6.out
Hello, world
$
```

Avec Gccgo c'est plus traditionnel :

```
$ gccgo helloworld.go
$ a.out
Hello, world
$
```

V - Un programme echo

Prochaine étape, voici une version de l'utilitaire UNIX echo :

```
1. package main
2.
3. import (
4.   "os"
5.   "flag" // parseur d'options de ligne de commande
6. )
7.
8. var omitNewline = flag.Bool("n", false, "don't print final newline")
9.
10. const (
11.   Space = " "
12.   Newline = "\n"
13. )
14.
15. func main() {
16.   flag.Parse()  // scanne la liste des arguments
17.   var s string = ""
18.
19.   for i := 0; i < flag.NArg(); i++ {
20.     if i > 0 {
```

```
21.       s += Space
22.     }
23.     s += flag.Arg(i)
24.   }
25.
26.   if !*omitNewline {
27.     s += Newline
28.   }
29.
30.   os.Stdout.WriteString(s)
31. }
```

Ce programme est court, mais réalise un bon nombre de nouvelles choses. Dans le dernier exemple, nous avons vu que **func** introduit une fonction. Les mots clés **var**, **const** et **type** (pas encore utilisé) introduisent aussi des déclarations, comme le fait **import**. Notez que l'on peut grouper les déclarations d'un même genre dans une liste entre parenthèses, une déclaration par ligne comme le montrent les lignes 3-6 et 10-13. Mais il n'est pas nécessaire de faire ainsi, on aurait pu écrire :

```
const Space = " "
const Newline = "\n"
```

Ce programme importe le package « os » afin d'accéder à sa variable Stdout, de type *os.File. L'instruction **import** est en fait une déclaration : dans sa forme générale, comme utilisée dans notre programme « hello world », il nomme l'identifieur (fmt) qui sera utilisé pour accéder aux membres du package importé depuis le fichier (« fmt »), trouvé dans le répertoire courant ou dans un répertoire standard. Dans ce programme nous avons utilisé le nom implicite du package. Par défaut, les packages sont importés en utilisant le nom défini par le package importé qui bien sûr est par convention le nom du fichier lui-même. Notre programme « hello world » peut donc simplement dire :

```
import "fmt"
```

Vous pouvez spécifier vos propres noms pour l'importation si vous le désirez, mais cela est seulement nécessaire si vous avez besoin de résoudre des conflits de nommages.

À l'aide de os.Stdout nous pouvons utiliser sa méthode **WriteString** pour afficher une chaîne de caractères.

À l'aide du package **flag**, la ligne 8 crée une variable globale pour capturer la valeur de l'argument -n du programme echo. La variable omitNewline est de type *bool, un pointeur sur un booléen.

Dans le **main.main**, nous parsons les arguments (ligne 16) et ensuite, nous créons une chaîne de caractères locale que nous utiliserons pour construire la sortie.

L'instruction de déclaration a la forme :

```
var s string = "";
```

C'est le mot clé **var**, suivi par le nom de la variable, suivi par son type, suivi par le signe égal et une valeur initiale pour la variable.

Go essaie d'être concis et cette déclaration peut être écourtée. Puisqu'une chaîne de caractères constante est de type string, nous n'avons pas besoin de le dire au compilateur. Nous pouvons écrire :

```
var s = "";
```

Ou nous pouvons encore faire plus concis et écrire :

```
s := "";
```

L'opérateur := est beaucoup utilisé en Go pour représenter une déclaration « initialisante ». Il y en a une dans la clause du for de la ligne suivante :

```
for i := 0; i < flag.NArg(); i++ {
```

Le package **flag** a parsé les arguments qui sont accessibles via une liste itérable.

Les instructions en Go diffèrent en de nombreux points par rapport au C. Premièrement, le for est la seule boucle, il n'y a pas de while ou de do. Deuxièmement, il n'y a pas de parenthèses autour de la clause, mais des accolades autour du corps sont obligatoires. Il en va de même pour les instructions if et switch. D'autres exemples viendront illustrer d'autres manières d'écrire ces instructions.

Le corps de la boucle construit la chaîne de caractères s par concaténation (en utilisant +=) des arguments séparés par des espaces. Après la boucle, si l'argument -n n'est pas présent, le programme concatène une nouvelle ligne. Enfin, le résultat est affiché.

Notez que **main.main** est une fonction qui ne retourne rien (sans type de retour). Elle est définie ainsi. Arriver à la fin de la fonction **main.main** signifie que celle-ci s'est bien exécutée ou « success ». Si vous souhaitez signaler un retour d'erreur, faites ainsi :

```
os.Exit(1)
```

Le package **os** contient d'autres fonctionnalités essentielles pour débuter. os.Arg est une tranche (slice) utilisée par le package **flag** pour accéder aux arguments de ligne de commande.

VI - Un interlude sur les types

Go possède des types familiers tels que **int** et **float** qui représentent des valeurs de tailles appropriées pour la machine. Mais Go définit également des types de tailles explicites comme **int8**, **float64** et bien d'autres plus les entiers non signés (unsigned) tels que **uint**, **uint32,** etc. Ce sont des types distincts, même si **int** et **int32** sont tous les deux d'une taille de 32 bits, ils ne sont pas le même type. Il y a aussi un synonyme de l'octet pour le type unit8 qui est le type élémentaire des chaînes de caractères.

En parlant de chaînes de caractères, c'est un type à part entière également. Les chaînes de caractères sont non modifiables. Ce ne sont pas simplement des tableaux d'octets. Une fois que vous avez construit une chaîne, vous ne pouvez pas la modifier. Vous pouvez modifier bien entendu une variable de type chaîne en la réaffectant à une autre valeur. Cet extrait de code provenant de strings.go est valide :

```
s := "hello"
if s[1] != 'e' { os.Exit(1) }
s = "good bye"
var p *string = &s
*p = "ciao"
```

Voici maintenant des instructions non valides, car ils tentent de modifier la valeur d'une chaîne :

```
s[0] = 'x';
(*p)[1] = 'y';
```

En termes C++, les chaînes Go sont semblables à des chaînes constantes (const string) tandis que les pointeurs sur des chaînes sont analogues à des références sur des chaînes constantes (string const &).

Oui, il y a des pointeurs. Cependant Go simplifie un peu leur utilisation.

Les tableaux sont déclarés de cette manière :

```
var arrayOfInt [10]int;
```

La taille d'un tableau est une partie de son type. Cependant, on peut déclarer une variable « tranche » à laquelle on peut assigner un pointeur vers n'importe quel tableau du même type d'éléments. Une tranche à la forme a[début : fin], ce qui représente le sous-tableau indexé par début jusqu'à fin - 1. Les tranches ressemblent beaucoup aux tableaux, mais n'ont pas de taille explicite ([] au lieu de [10]) et ils référencent un segment d'un tableau régulier. Plusieurs tranches peuvent partager des données si celles-ci proviennent du même tableau. Plusieurs tableaux ne peuvent pas partager des données.

Les tranches sont bien plus communes dans les programmes Go que les tableaux réguliers. Elles sont plus flexibles, ont des références sémantiques et sont efficaces. Ce qui leur manque est le contrôle précis du layout de stockage d'un tableau régulier. Si vous souhaitez avoir une centaine d'éléments d'un tableau dans votre structure, vous devriez utiliser un tableau régulier.

Lorsque vous voulez passer un tableau à une fonction, vous devez pratiquement toujours déclarer le paramètre formel comme étant une tranche. Lorsque vous appelez la fonction, passer l'adresse du tableau et Go créera (efficacement) une tranche et passera celle-ci.

En utilisant les tranches, on peut écrire cette fonction (de sum.go) :

```
func sum(a []int) int {
  s := 0
  for i := 0; i < len(a); i++ {
    s += a[i]
  }
  return s
}
```

et l'invoquer de cette manière :

```
s := sum(&[3]int{1,2,3}) // une tranche du tableau est passée à sum
```

Notez ici comme le type de retour (int) est défini pour **sum()** en le plaçant après la liste des paramètres. L'expression [3]int{1,2,3} (un type suivi par une expression entre accolades) est le constructeur d'une valeur. Dans ce cas-ci un tableau de trois entiers. En mettant un & devant, on passe ainsi l'adresse d'une instance unique de cette valeur. Nous passons le pointeur à **sum()** (implicitement) en créant une tranche.

Si vous êtes en train de créer un tableau régulier, mais que voulez que le compilateur compte le nombre d'éléments pour vous, utilisez… comme taille du tableau :

```
s := sum(&[...]int{1,2,3});
```

En pratique, sauf si vous êtes méticuleux concernant le layout de stockage d'une structure, une tranche (en utilisant des crochets vides et pas de &) est tout ce dont vous avez besoin :

```
s := sum([]int{1,2,3});
```

Il y a aussi les maps, que vous pouvez initialiser ainsi :

```
m := map[string]int{"one":1, "two":2}
```

La fonction **len()**, qui retourne un nombre d'éléments, fait sa première apparition dans **sum**. Elle fonctionne sur les chaînes, les tableaux, les tranches, les maps et les channels.

Soit dit en passant, on peut également utiliser range dans les boucles sur les chaînes, les tableaux, les tranches, les maps et les channels. Au lieu d'écrire :

```
for i := 0; i < len(a); i++ { ... }
```

pour boucler sur les éléments d'une tranche (ou d'une map ou…), on pourrait écrire :

```
for i, v := range a { ... }
```

Cela assigne i en index et v en valeur pour chaque élément successif de la cible du range.

VII - Un interlude sur les allocations

La plupart des types en Go sont des valeurs. Si vous avez un entier ou une structure ou un tableau, un assignement copie le contenu de l'objet. Pour allouer une nouvelle variable, utilisez **new()** qui retourne un pointeur sur le stockage alloué.

```
type T struct { a, b int }
var t *T = new(T);
```

ou plus simplement

```
t := new(T);
```

Certains types (les maps, les tranches et les channels) ont des références sémantiques. Si vous modifiez le contenu d'une tranche ou d'une map, les autres variables référençant les mêmes valeurs verront la modification. Pour ces trois types, utilisez la fonction **make() :**

```
m := make(map[string]int);
```

Cette instruction initialise une nouvelle map prête à stocker des entrées. Si vous voulez simplement déclarer une map faîtes ainsi :

```
var m map[string]int;
```

Cela crée une référence nulle qui ne peut pas recevoir d'affectation. Pour utiliser la map vous devez d'abord initialiser la référence en utilisant **make()** ou en l'assignant à partir d'une map existante.

Notez ici que **new**(T) retourne un type *Talors que **make**(T) retourne un type T. Si par erreur vous allouez à la référence d'un objet avec **new()**, vous recevrez un pointeur sur une référence nulle. C'est équivalent à déclarer une variable non initialisée et à prendre son adresse.

VIII - Un interlude sur les constantes

Bien que les entiers ont différentes tailles en Go, les entiers constants ont une taille fixe. Il n'existe pas de constantes comme 0LL ou 0x0UL. Au lieu d'avoir différentes tailles, les entiers constants sont évalués comme des valeurs de grande précision qui peuvent déborder uniquement lorsqu'elles sont assignées à une variable entière d'une précision inférieure.

```
const hardEight = (1 << 100) >> 97 // correct
```

Il existe des nuances qui méritent quelques retours sur les spécifications du langage, en voici des illustrations :

```
var a uint64 = 0  // a est de type uint64, valeur 0
a := uint64(0)    // équivalent; utilise une "conversion"
i := 0x1234       // i a le type entier par défaut
var j int = 1e6   // correct - 1000000 est représentable en entier
x := 1.5          // a flottant
```

```
i3div2 := 3/2     // division entière - le résultat est 1
f3div2 := 3./2.   // division flottante - le résultat est 1.5
```

Les conversions ne fonctionnent que pour de simples cas comme la conversion d'entiers de leur signe ou de leur taille et entre les entiers et les flottants plus quelques cas particuliers. Il n'existe pas en Go de conversion numérique automatique ou autre conversion du même genre autre que donner une taille fixe à une constante ainsi que d'assigner un type à une variable.

IX - Un package d'entrée/sortie (IO)

Nous allons maintenant étudier un simple package afin de gérer les entrées-sorties sur des fichiers. En somme, les interfaces : Ouvrir, fermer, lire et écrire. Voici la première partie du fichier file.go :

```
1. package file
2.
3. import (
4.   "os"
5.   "syscall"
6. )
7.
8. type File struct {
9.   fd int // le numéro d'identification du fichier (file descriptor number)
10.   name string // le nom du fichier à l'ouverture
11. }
```

Ces premières lignes déclarent le nom du package **file** et importent deux packages. Le package os gère l'abstraction du système d'exploitation afin de donner une vue générique sur les fichiers et bien plus. Dans cet exemple nous utiliserons ses utilitaires de capture des erreurs et reproduirons les rudiments de sa gestion des entrées-sorties sur les fichiers.

Le deuxième élément est un package externe de bas niveau fournissant une interface primitive aux appels du système d'exploitation.

Ensuite, nous avons la définition d'un type. Le mot clé keyword introduit une déclaration de type, dans ce cas une structure de données appelée **File**. Pour rendre les choses un peu plus intéressantes, notre fichier (File) inclut le nom du fichier auquel le descripteur de fichier fait référence.

Du fait que File commence par une majuscule, le type est accessible depuis un package externe. En Go, la règle concernant la visibilité des informations est simple : si le nom (d'un type, d'une fonction, d'une méthode, d'une constante, d'une variable, etc.) commence par une majuscule, les utilisateurs d'un package externe pourront y accéder. Autrement, ce qui a été déclaré n'est visible qu'à l'intérieur de son propre package. C'est plus qu'une convention, la règle est strictement appliquée par le compilateur. En Go, le terme désignant les éléments accessibles depuis des packages externes est « exported ».

Dans le cas de **File**, tous les champs sont en minuscules et donc invisibles aux utilisateurs. Mais nous allons rapidement donner quelques méthodes exportées.

D'abord, voici une fonction pour créer un fichier (type File) :

```
func newFile(fd int, name string) *File {
  if fd < 0 {
    return nil
  }
  return &File{fd, name}
}
```

Cette fonction retourne un pointeur sur une nouvelle structure de type File. Ce code utilise la notion Go de « composé littéral », analogue à celles utilisées pour créer des maps et des tableaux. Pour allouer un nouvel objet de type File, nous aurions également pu écrire :

```
n := new(File)
n.fd = fd
n.name = name
return n
```

Mais pour de simples structures comme File, il est plus simple de retourner l'adresse d'un composé littéral, comme cela a été fait à la ligne 21.

On peut utiliser la fonction **newFile()** pour créer des variables de type *File bien connues :

```
var (
  Stdin  = newFile(0, "/dev/stdin")
  Stdout = newFile(1, "/dev/stdout")
  Stderr = newFile(2, "/dev/stderr")
)
```

La fonction **newFile()** n'est pas exportée, car elle est interne. La bonne fonction exportée à utiliser est Open :

```
func Open(name string, mode int, perm uint32) (file *File, err os.Error) {
  r, e := syscall.Open(name, mode, perm)
  if e != 0 {
    err = os.Errno(e)
  }
  return newFile(r, name), err
}
```

Il y a un certain nombre de nouvelles choses dans ces quelques lignes. Premièrement, Open retourne plusieurs valeurs. Un File et une erreur (vous en saurez plus sur les erreurs dans un moment). Nous déclarons un retour de plusieurs valeurs dans une liste de déclarations entre parenthèses. Syntaxiquement, c'est comme une seconde liste de paramètres. La fonction **syscall()**.Open retourne également plusieurs valeurs que l'on récupère à la ligne 31. À cette ligne, on déclare r et e pour capturer ces valeurs, toutes les deux de type int (vous pouvez jeter un oil au package **syscall**). Finalement la ligne 35 retourne deux valeurs : un pointeur sur le fichier et une erreur. Si syscall.Open échoue, le descripteur de fichier r sera négatif et **newFile()** retournera nil.

À propos des erreurs : la librairie os inclut une notion générale d'une erreur. Il est judicieux d'utiliser cette facilité dans nos interfaces. Dans la fonction Open, on convertit un entier Unix errnd en un type os.Errno, qui implémente os.Error.

Maintenant que nous pouvons créer des fichiers, nous pouvons écrire des méthodes pour eux. Pour déclarer une méthode sur un type, nous définissons une fonction avec une variable de réception explicite du même type, placée entre parenthèses avant le nom de la fonction. Voici quelques méthodes de *File, chacune d'entre elles déclare une variable de réception file.

```
1. func (file *File) Close() os.Error {
2.   if file == nil {
3.     return os.EINVAL
4.   }
5.   e := syscall.Close(file.fd)
6.   file.fd = -1 // ainsi il ne pourra pas être fermé une deuxième fois
7.   if e != 0 {
8.     return os.Errno(e)
9.   }
10.   return nil
11. }
12.
13. func (file *File) Read(b []byte) (ret int, err os.Error) {
14.   if file == nil {
15.     return -1, os.EINVAL
16.   }
```

```
17.   r, e := syscall.Read(file.fd, b)
18.   if e != 0 {
19.     err = os.Errno(e)
20.   }
21.   return int(r), err
22. }
23.
24. func (file *File) Write(b []byte) (ret int, err os.Error) {
25.   if file == nil {
26.     return -1, os.EINVAL
27.   }
28.   r, e := syscall.Write(file.fd, b)
29.   if e != 0 {
30.     err = os.Errno(e)
31.   }
32.   return int(r), err
33. }
34.
35. func (file *File) String() string {
36.   return file.name
37. }
```

Il n'existe pas de this implicite et la variable de réception doit être utilisée pour accéder aux membres de la structure. Les méthodes ne sont pas déclarées dans la définition de la structure elle-même. La déclaration de struct définie seulement les données membres. En fait, des méthodes peuvent être créées pour pratiquement n'importe quel type que vous nommez, comme un entier ou un tableau. Nous verrons un exemple avec les tableaux par la suite.

La méthode **String()** est appelée ainsi à cause d'une convention d'affichage que nous décrirons plus tard.

Les méthodes utilisent la variable publique os.EINVAL pour retourner une erreur Unix de code EINVAL (la version de os.Error). La librairie **os** définit un ensemble d'erreurs standards comme celle-ci.

Maintenant nous pouvons utiliser notre nouveau package :

```
1. package main
2.
3. import (
4.   "./file"
5.   "fmt"
6.   "os"
7. )
8.
9. func main() {
10.   hello := []byte("hello, world\n")
11.   file.Stdout.Write(hello)
12.   f, err := file.Open("/does/not/exist", 0, 0)
13.   if f == nil {
14.     fmt.Printf("can't open file; err=%s\n", err.String())
15.     os.Exit(1)
16.   }
17. }
```

Le « ./ » dans « ./File » indique au compilateur de bien utiliser notre package au lieu d'un autre présent dans le répertoire d'installation des packages. (Il faut au préalable compiler file.go).

Maintenant nous pouvons compiler et exécuter le programme :

```
$ 6g file.go                      # compile le package file
$ 6g helloworld3.go               # compile le package principal
$ 6l -o helloworld3 helloworld3.6  # lien - pas besoin de mentionner "file"
$ helloworld3
hello, world
can't open file; err=No such file or directory
$
```