Un livre de Wikilivres.

Programmation en Go

Installation

Prérequis

Vous devez installer ou avoir installé les outils suivants :

- compilateur C et librairies standard
- GNU Bison
- **make**
- **awk**
- **ed**

Ces outils sont installés par défaut sur la plupart des systèmes Linux. De plus, il faut installer le logiciel Mercurial.(Téléchargement de Mercurial (en) (http://mercurial.selenic.com/wiki/Download) [archive])

Récupérer les sources

Lancez la commande suivante :

```
hg clone -r release https://go.googlecode.com/hg/ go
```

Cette commande crée un sous-répertoire **go** dans le répertoire courant.

Compiler go

Lancer :

```
cd go/src
./all.bash
```

La dernière commande compile le logiciel. Cela peut prendre un certain temps, et cela s'accompagne d'un défilement de commandes sur la console. A la fin de la compilation le texte suivant est affiché :

```
--- cd ../test
N known bugs; 0 unexpected bugs
---
Installed Go for linux/amd64 in /home/you/go.
Installed commands in /home/you/go/bin.
*** You need to add /home/you/go/bin to your $PATH. ***
The compiler is 6g.
```

Ce texte peut varier suivant votre architecture matérielle et le répertoire dans lequel se trouvent les sources. Il y a principalement deux architectures supportées par Go :

- 386 (compilateur 8g)
- AMD 64 bits (compilateur 6g)

Dans la suite nous supposerons que le compilateur est **6g**.

Ajouter le chemin au PATH

Editez le fichier **.bashrc** se trouvant dans votre répertoire personnel. En supposant que le répertoire de Go soit situé dans votre répertoire personnel, ajoutez les lignes suivantes à la fin de ce fichier :

```
export PATH=$PATH:$HOME/go/bin
export GOROOT=$HOME/go
```

A présent, vous pouvez tester le compilateur en compilant votre premier programme, comme nous allons le voir au chapitre suivant.

Premier

Compiler votre premier programme

Créez le fichier **hello.go** et insérez le texte suivant :

```
package main
```

```
import "fmt"

/* Un commentaire sur
   plusieurs lignes */
func main() {
    // Un commentaire sur une ligne
    fmt.Printf("Bonjour, monde!\n")
}
```

Lancez ensuite les commandes suivantes dans le répertoire où se trouve ce fichier :

```
8g hello.go
8l hello.8
```

```
./8.out
->Bonjour, monde!
```

Si ces commandes échouent c'est probablement que le chemin d'accès à Go n'est pas configuré. Dans ce cas, vérifiez votre fichier .bashrc comme indiqué au chapitre précédent, ou indiquez le chemin complet à Go dans les commandes 8g et 8l. Il se peut également que votre compilateur ne soit pas 8g mais 6g (AMD 64 bits) ; dans ce cas, remplacez 8 par 6 dans 8g 8l hello.8 et 8.out :

```
6g hello.go
6l hello.6
```

```
./6.out
->Bonjour, monde!
```

Structure

```
package main
```

Cette ligne permet de déclarer un module, en l'occurrence le module "main" qui représente notre programme.

```
import "fmt"
```

Cette ligne déclare l'utilisation du module "fmt". Les guillemets (double quotes) sont indispensables.

```
func main() {
```

Ceci déclare le point d'entrée du programme. L'accolade ouvrante définit le début d'un bloc, celui-ci se terminera par une accolade fermante. Les accolades doivent toujours être appariées.

```
fmt.Printf("Bonjour, monde!\n")
```

Cette ligne, lorsqu'elle est évaluée, affiche à l'écran la chaîne de caractères "Bonjour, monde!" suivie d'un

retour à la ligne.

Voilà, nous avons étudié en détail notre premier programme. Passons maintenant au chapitre suivant qui traite des types de base.

Les types de base

Les réels

Ce sont les nombres réels qui ici sont, bien entendu, approchés par des valeurs finies pour des raisons évidentes de limite de taille. Ces nombres sont déclarés par le type **float** qui veut dire "nombre réel à virgule flottante".

Voici comment déclarer un réel :

```
// déclaration de constante
const Pi = 3.14159
// variable rayon
var rayon float32 = 1.0
```

Les lignes commençant par deux barres sont ignorées à la compilation ; ce sont ce qu'on appelle des commentaires.

Si l'on utilise ":=" à la place de "=", le type est déterminé automatiquement (inféré) et à ce moment, on peut écrire :

```
rayon := 1.0
```

Opérations arithmétiques

Les opérations suivantes sont supportées :

- addition +
- soustraction -
- multiplication *
- division /

Voici deux exemples :

```
var aire = rayon*rayon*Pi
var circonference = 2*rayon*Pi
```

```
var n = 100
var somme = 2*n*(n-1)
```

Entiers

Un entier est déclaré avec le type **int** (integer). Il existe d'autres types d'entier, notamment **uint** qui est toujours positif.

Les types **int64** et **uint64** sont de plus grande capacité car ils sont stockés sur 64 bits quelle que soit l'architecture utilisée, tandis qu' **int** et **uint** sont en général stockés sur 32 bits.

Les mêmes opérations arithmétiques s'appliquent aux entiers, avec en plus le reste de la division ou modulo (%). Exemple:

```
var reste = 821 % 100
// le résultat est le reste de la division par 100 soit 21
```

De plus, les nombres entiers et réels supportent les incrémentations (ajout de un) et décrémentations (retrancher un).

```
// Incrémentation
i++
// Décrémentation
i--
```

Chaînes de caractères

Elles correspondent au type **string**. Les chaînes peuvent être concaténées avec l'opérateur d'addition (+), en concaténant `"James "` et `"Bond"` on obtient `"James Bond"`. Les chaînes de caractères peuvent être déclarées en les mettant entre guillemets (double quotes).

Exemple :

```
var james = "James"
var nom = "Bond"
var nomComplet = james+" "+nom
```

Les chaînes de caractères sont encodées en UTF-8 dans les sources Go, et Go gère le jeu de caractères Unicode.

Booléens

Un booléen (type `bool`) ne peut prendre que deux valeurs: vrai et faux. Ces deux valeurs correspondent aux mots-clés `true` et `false`.

Nombres complexes

La fonction `cmplx()` construit un nombre complexe, tandis que `real()` et `imag()` permettent respectivement d'en obtenir les parties réelle et imaginaire.

```
cmplx(partieReelle, partieImaginaire) complex
real(complex) float
imag(complex) float
```

Les tableaux

Tableaux

Un tableau de valeurs est une série consécutive de valeurs, accessibles par leur indice entier, 0, 1, 2 3...taille-1.

En Go, on déclare un tableau en faisant précéder le type des éléments par la taille entre crochets, comme dans **[3]int**. Par exemple, pour déclarer un tableau de 3 réels :

```
var tab [4]float
```

ou, en déclarant une valeur "littérale" d'un tableau :

```
var tab = [...]float{1.0, 2.0, 3.0}
```

On peut accéder à un élément d'un tableau en faisant suivre le nom de la variable de l'indice dans le tableau, entre crochets :

```
var un =tab[0]
tab[1] = 5.1
```

Attention, les indices d'un tableau démarrent à zéro et pas à un.

On ne peut pas accéder à un élément dont l'indice dépasse la taille du tableau, moins un, ou dont l'indice est négatif. On peut obtenir la taille d'un tableau en utilisant la fonction `len ()` :

```
var taille = len(tab)
```

Fonction somme

Voici un exemple de fonction : la somme de 3 réels. Nous allons faire la somme en utilisant un tableau au lieu de passer chaque élément du tableau en paramètre :

```
func Somme(a [3]float) (somme float) {
    for _, v := range a {
        somme += v
    }
    return
}
```

On déclare ici une fonction nommée Somme qui prend comme paramètre un tableau de 3 réels, et retourne un réel. La valeur de retour, en Go, vient après la liste des paramètres.

```
for _, v := range a {
```

Il s'agit d'une boucle d'itération sur chaque élément du tableau "a". Le mot-clé **for** définit une boucle d'itération, tandis que **range** définit l'intervalle d'un tableau (ici, "a")

```
_, v := range a
```

Deux variables, "_" et "v", prennent comme valeur d'itération l'indice et la valeur, respectivement, de chaque élément du tableau a. Le caractère "_" utilisé comme variable signifie que la valeur de l'indice est ignorée durant l'itération.

```
return
```

Ici, nous retournons la valeur "somme" à la fonction qui nous appelle. Ce devrait être écrit "return somme", mais comme nous avions déclaré la valeur de retour avec un nom de variable, la valeur de cette variable est automatiquement retournée. De plus, la variable est initialisée automatiquement (à zéro) ce qui fait encore une opération qui n'est pas écrite.

Les 'tranches'

Les *slices* (tranches en français), sont des tableaux au comportement différent. Disons que les tableaux 'classiques' ont une taille fixe tandis que les tranches n'ont pas de taille précise au moment de la compilation.

Les tableaux fixes sont passés par copie lors d'un appel de fonction. Donc, si votre tableau occupe 4 Mégaoctets, tout le contenu du tableau sera copié à chaque appel de fonction ! Ce peut être une source de lenteur dans votre programme, et il y a deux solutions à ce problème :

1. utiliser un pointeur sur le tableau ou,
2. utiliser une tranche.

Les tranches sont prévues pour ce cas, de plus elles sont plus pratiques que les tableaux du fait qu'elles ont une taille modifiable.

Le type tranche est défini par le type des éléments précédé par des crochets : "[]int" est une tranche d'entiers.

Pour créer une tranche il suffit de prendre un intervalle d'indices séparés par ":" entre crochets, d'une variable tableau ou d'une autre tranche :

```
var tranche []int = tab[0:3]
var autre []int = tab[:] // tab est pris en entier dans la tranche autre
```

Il est aussi possible de déclarer une tranche "littérale" :

```
var fibonacci = []int{1, 1, 2, 3, 5, 8}
```

La longueur d'une tranche s'obtient par la fonction `len()`, comme pour les tableaux. La longueur du tableau sous-jacent est appelée *capacité* et s'obtient par la fonction `cap()`.

Si l'on ne précise pas le début d'une tranche, c'est le premier élément du tableau comme dans `tab[:3]`, si on

ne précise pas la fin comme dans `tab[2:]`, c'est le dernier élément du tableau.

Fonction Somme sur une tranche

Réécrivons à présent notre fonction Somme avec les tranches :

```
func Somme (a []int) (somme int) {
    for _,v := range a {
        somme += v
    }
    return
}
```

Comme on peut le voir, la différence de syntaxe est minime, mais la différence sémantique est réelle.

La fonction append()

Cette fonction retourne la tranche formée en ajoutant après le dernier élément de la tranche argument, les autres arguments passés à la fonction.

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

Ce code affiche :

```
[1 2 3 4 5 6]
```

Allocation

On peut allouer de la mémoire pour une tranche en utilisant la fonction `make(type, longueur, capacité)` :

```
var tranche = make([]int,10,100)
```

Initialisation

Constantes

On peut initialiser une constante avec le mot-clé `const` :

```
const Pi := 3.14159
```

Variables globales

Une variable globale est déclarée par le mot-clé var, avec au minumum un premier caractère en majuscule(exemple:bb est local alors que Bb est globale). Elle est initialisée par le symbole "=" ou ":=".

On peut affecter le résultat d'une fonction à une variable globale, comme ceci:

```
var (
    HOME = os.Getenv("HOME")
    USER = os.Getenv("USER")
    GOROOT = os.Getenv("GOROOT")
)
```

Variables locales

On déclare une variable local avec *var* suivi du nom de la variable puis de son type :

```
var x float
```

On peut initialiser une variable avec le signe égal :

```
x = 2.17
```

Ou bien combiner les deux:

```
var x float = 2.17
```

On peut *inférer* le type tout en affectant la variable :

```
var x := 2.17 // automatiquement float
```

Enfin, on peut omettre *var*

```
y := 3.1415
```

La fonction init

La fonction **init()** est une fonction spéciale. Chaque fonction portant ce nom dans chaque module sera appelée à l'initialisation du programme (une seule fois) et ce, dans l'ordre d'import des modules. Un seul thread est utilisé lors de cette phase d'initialisation pour l'exécution des goroutines. (qui feront l'objet d'un autre chapitre)

Pointeurs et structures

Pointeurs

Les pointeurs sont un type important en Go. Nous les avons à peine évoqués dans un autre chapitre. Un pointeur est un peu comme l'adresse pointant sur une valeur, on peut retrouver cette valeur et la modifier à l'aide d'un pointeur.

Un pointeur est déclaré en faisant précéder le type de donnée pointée par une étoile (*int est un pointeur sur un entier). On prend l'adresse pointant sur une donnée en la faisant précéder de **&**:

```
var i = 5
var j *int = &i // pointeur sur i
```

Ensuite on peut récupérer la valeur sur laquelle on pointe en précédant le pointeur par une étoile:

```
var k = *j // k vaut 5
```

On peut modifier la valeur pointée avec ***p=v**:

```
*j = 6
```

Mais, maintenant, i vaut également 6, car j correspond à l'adresse de i! On peut également utiliser += pour ajouter une valeur:

```
*j+=10
k=*j // k vaut également 16 maintenant
```

Le pointeur 'nil'

Le mot-clé **nil** représente un pointeur qui ne pointe sur aucune valeur licite. Cela peut représenter la valeur initiale d'un pointeur qui ne pointe vers 'rien'.

Les types structurés

On peut déclarer un *alias* vers un type en Go en utilisant le mot-clé **type** suivi du type défini, par exemple:

```
type Vecteur3D [3]float
var vec Vecteur3D
```

On peut déclarer un type structuré ou structure, en utilisant le mot-clé **struct**:

```
type Intervalle struct {
    debut int
    fin   int
}
```

On peut déclarer une valeur littérale de structure comme ceci:

```
intervalle := Intervalle{0,3}
inter2 := Intervalle{fin:5}
```

Mais en général on utilise un pointeur sur une structure, pour ne pas la passer par valeur mais par référence:

```
inter  := new(Intervalle)
inter2 := &Intervalle{0,5}
```

On accède ensuite à chaque champ d'une structure ou de son pointeur en faisant suivre la valeur d'un point suivi du nom du champ:

```
inter.fin += inter.debut
```

Les tableaux associatifs

Déclaration

Le type tableau associatif représente un tableau dont on accède aux éléments par un indice. À la différence d'un tableau itératif, l'indice peut être autre chose qu'un entier, et les éléments peuvent être non-consécutifs.

```
var Constantes = map[string] float {
   "Pi": 3.14159,
   "e":  2.71,
   "g":  9.81,
}
```

On accède ainsi à Pi en écrivant:

```
var Pi := Constantes["Pi"]
```

Types supportés pour les indices

On peut définir un indice comme étant un type numérique, une chaîne de caractères, un type pointeur ou une interface. Les structures, tableaux et tranches ne peuvent être utilisés comme indices d'un tableau associatif.

Vérifier l'existence d'une entrée

On peut vérifier l'existence d'une entrée en utilisant la syntaxe suivante:

```
valeur, ok := Constantes["g"]
```

Dans ce cas, "ok" est de type booléen.

Supprimer une entrée

```
Constantes["kB"] = 0, false
```

permet de supprimer "kB" des entrées du tableau, même si cette entrée n'existe pas.

Formatage de texte

Formatage de types de base

Nous allons utiliser le module "fmt" comme expliqué au chapitre Premier exemple.

fmt.Printf()

Cette fonction très simple prend un nombre variable d'arguments: le premier est une chaîne [de format] et les suivants sont les arguments à formater.

Dans la chaîne de format, le caractère "%" a une signification spéciale:

- suivi d'un "c", il implique la présence d'un octet ou d'un entier et le formate en caractère,
- suivi d'un "d", il implique la présence d'un entier et le formate en décimal,
- suivi d'un "f", il implique la présence d'un nombre décimal à virgule flottante,
- suivi d'un "s", il implique la présence d'une chaîne,
- suivi d'un "x", il implique la présence d'un entier et le formate en hexadécimal,
- suivi d'un "g", il implique la présence d'un réel,
- suivi d'un "v", il formate de manière automatique n'importe quel type,
- suivi d'un "T", il affiche le type de la valeur fournie.

Enfin, la chaîne "%%" est remplacée par un seul "%" à la sortie.

La fonction **fmt.Sprintf** a le même type d'entrée mais retourne la chaîne formatée au lieu de l'afficher.

```
fmt.Printf("Hello %d, %x\n", 123, 237)
=> Hello 123, ed
```

Exemple

```
type T struct {
   a int
   b float
   c string
}
t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", Constantes)
```

affiche:

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string] float{"Pi":3.14159, "e":2.71, "g":9.81}
```

fmt.Println()

Cette fonction formate de manière automatique tout type passé comme paramètre. La contrepartie **fmt.Sprintln()** retourne la chaîne formatée au lieu de l'afficher. Cela revient au même que d'utiliser le formatage par défaut "%v".

String()

On peut définir la méthode "String()" pour un type quelconque et alors, Go l'utilisera pour formater la structure dans le formatage par défaut ("%v").

La boucle for

La forme 'tant que'

```
for {
for condition {
```

Dans ces deux formes, la boucle se répète tant que la condition est évaluée vraie, ou de manière inconditionnelle. Dans les deux cas, cela peut mener à une boucle infinie qui prend de plus en plus de ressources système.

En revanche, il est possible de couper l'exécution d'une boucle:

break

Cette commande interrompt la boucle et en sort.

continue

Cette commande interrompt la boucle et réitère. Si l'on parcourt un tableau, on réitère sur un nouvel élément, sinon on réévalue la condition de boucle.

return

Return fait sortir de la fonction en cours; si une valeur est fournie à return, celle-ci est retournée à l'appelant.

Itérer sur les éléments d'un tableau

```
for index, value := range tableau {
```

Cette forme a déjà été étudiée dans le chapitre Les tableaux. Elle permet également d'itérer sur les éléments d'une chaîne de caractères Unicode (voir le chapitre Unicode et Go) et sur ceux d'un tableau associatif.

Itération plus générale

```
for initialisation; condition; iteration {...}
```

Par exemple la ligne suivante compte de 0 à 9:

```
for i:=0;i<10;i++ {fmt.Println(i)}
```

If et switch

Alternatives: if/else

La forme if/else permet de tester une condition, d'exécuter un bloc d'instructions si cette condition est vraie (le booléen 'true'), et un autre bloc si la condition est fausse.

```
if condition {
  condition vraie
} else {
  condition fausse
}
```

Par exemple, pour calculer une valeur absolue:

```
func Abs (x int) int {
  if x < 0 {
    return -x
  }
  return x
}
```

Switch

```
switch {
case condition: instructions
case condition: instructions
default: instructions
}
```

On peut ainsi réaliser plusieurs alternatives. On peut également réaliser un "switch" sur une valeur donnée:

```
signe:="Bélier"
switch signe {
case "Bélier","Gémeaux": return "Vous allez être riche"
case "Taureau","Sagittaire": return "Vous allez épouser une belle personne"
default: return "Je ne sais pas trop"
}
```

Détecter un type

Une variable de type **interface {}** peut contenir une valeur de type quelconque. Il est possible de tester que le type de cette valeur est T en exécutant:

```
v, ok = valeur.(T)
```

Il est aussi possible de le faire avec switch:

```
var valeur interface {}
switch t := valeur.(type) {
default:
   fmt.Printf("type inattendu %T", t)
case bool:
   fmt.Printf("booléen %T\n", t)
case int:
   fmt.Printf("entier %T\n", t)
case *bool:
   fmt.Printf("pointeur vers booléen %T\n", *t)
case *int:
   fmt.Printf("pointeur vers entier %T\n", *t)
}
```

Alternative avec initialisation

```
if initialisation; condition {
switch initialisation; condition {
```

Exemple:

```
if err := file.Chmod("0664"); err {
  //erreur dans chmod
}
```

Conditions

Conditions simples

Il est possible d'utiliser les opérateurs de comparaison suivants:

```
== != < <= > >=
```

"==" est l'opérateur égal à, différent de l'opérateur d'affectation "=". "!=" signifie 'différent de'.

Ces opérateurs fonctionnent pour les nombres entiers, flottants et pour les chaînes de caractères.

Opérateurs logiques

- **&&**, ET logique,
- ||, OU logique,
- **!**, négation logique.

Ces opérateurs agissent sur les valeurs de type booléen.

Opérateurs bit à bit

- **&**, ET logique bit à bit,
- |, OU logique bit à bit,
- ^, OU EXCLUSIF bit à bit,
- ~, complément binaire ou négation bit à bit.

À ces opérateurs il faut ajouter;

- << décalage binaire vers la gauche,
- >> décalage à droite.

Ces opérateurs agissent sur les entiers non signés: **uint, uint8, uint16, uint32, uint64, byte**.

Il est possible de combiner ces opérateurs avec une affectation de leur résultat comme avec **a += b**, par exemple:

```
a <<= 2
b ^= a & 0xffffffff
```

Fonctions

Nombre variable de paramètres

Le dernier paramètre d'une fonction peut être suivi de **...type** qui définit un nombre variable de paramètres.

```
func Min(a ...int) int {
  if len(a)==0 {
    return 0
  }
  min := a[0]
  for _, v := range a {
    if v < min {
```

```
      min = v
    }
  }
  return min
}
```

Ensuite, on peut appeler cette fonction de la manière suivante:

```
x := Min(1, 3, 2, 0) // zéro
```

On peut également passer un tableau à la place des arguments avec la syntaxe **t...**:

```
tab := []int{1,3,2,0}
x := Min(tab...)
```

Valeur de retour nommée

On peut omettre le nom de la variable de retour grâce à ce nom:

```
func Somme(a []int) (somme int) {
  for _,v := range a {
    somme += v
  }
  return
}
```

Valeurs de retour multiples

On peut retourner deux valeurs ou plus avec la syntaxe suivante:

```
func SommeEtMoyenne (a []float) (float,float) {
  somme := 0.0
  for _, x := range a {
    somme += x
  }
  return somme,somme/len(a)
}
somme, moyenne := SommeEtMoyenne(a)
```

Defer

```
defer fonction(....)
```

Entraîne l'exécution différée d'une commande à la sortie de la fonction courante. Par exemple, juste après avoir ouvert un fichier, on peut différer sa fermeture par:

```
defer fichier.Close()
```

ce qui fait que le fichier sera fermé quoi qu'il arrive avant de quitter la fonction. Les commandes enregistrées par 'defer' sont exécutées dans l'ordre inverse de leur déclaration: la dernière déclarée sera la première exécutée.

Fonctions anonymes

Les fonctions anonymes sont un des types de base de Go. On peut donc stocker une fonction dans un tableau et dans n'importe quelle variable dont le type est défini comme étant:

```
func (arguments, ...) retour
```

On déclare une fonction littérale par la syntaxe suivante :

```
func (arguments, ...) retour { corps de la fonction }
```

Interfaces

Définition d'une interface

Une interface est un ensemble de *méthodes*, c'est-à-dire des fonctions qu'il est possible d'implémenter pour un type défini. En voici un exemple, une interface de tri:

```
type sort.Interface interface {
  Len() int
  Less(i, j int) bool
  Swap(i, j int)
}
```

Cette interface permet de trier n'importe quelle structure de données consécutives. Il suffit de définir les méthode Len(), Less() et Swap() et nous pourrons appeler sort() sur cette structure de données.

Nous pouvons par exemple définir cette interface pour un tableau d'entiers:

```
type Sequence []int

func (s Sequence) Len() int {
   // longueur de s
   return len(s)
}
func (s Sequence) Less(i, j int) bool {
   // retourne vrai si l'élément d'indice i est inférieur à l'élément d'indice j
   return s[i] < s[j]
}
func (s Sequence) Swap(i, j int) {
   // échange deux éléments
   s[i], s[j] = s[j], s[i]
}
```

Ainsi la syntaxe fait précéder le nom de la méthode du type défini entre parenthèses. Nous pouvons

maintenant trier une séquence en faisant:

```
var seq Sequence
sort.Sort(seq)
```

Note sur les pointeurs

Lorsque l'on implémente une méthode sur une structure, il est d'usage de déclarer la méthode sur un type pointeur de cette structure, pour éviter le passage par valeur:

```
type Foo struct {...}
func (foo *Foo) String() string {return ...}
```

Héritage d'interfaces

En incluant un nom de type précédé d'une étoile dans une structure, sans le nommer, la structure hérite de toutes les interfaces de cette structure en les lui déléguant.

```
type Tache struct {
   Commande string
   *log.Logger
}
```

Ensuite on appelle

```
tache.Log()
```

qui est délégué au membre log.Logger de la structure.

Héritage multiple

Il est possible d'hériter de manière multiple comme dans l'exemple suivant:

```
type ReaderWriter struct {
  *io.Reader
  *io.Writer
}
```

Constructeur

Il est alors d'usage de définir un constructeur, si nécessaire, pour la structure Tache, que l'on nommera "NewTache":

```
func NewTache(commande string, loggueur *log.Logger) *Tache {
   return &Tache{commande, loggueur}
}
```

Modules

Importer un module

Pour importer un seul module:

```
import "module"
```

Pour en importer plusieurs:

```
import (
   "module"
   "module2"
   ...
)
```

Un nom de module est le chemin relatif du répertoire contenant le module. Par exemple, dans le répertoire de go on trouve "src/pkg/container/vector" qui est importé comme "container/vector". Une fois le module importé, on s'y réfère dans le source comme "vector" tout court.

Il est indispensable de noter le nom des modules entre guillemets.

Conventions

Les seuls noms exportés sont les noms commençant par une majuscules: noms de fonctions, de types, de méthodes, de constantes, de variables... Il est d'usage de ne pas utiliser le tiret de soulignement (underscore) mais de capitaliser les noms à exporter en "CamelCase". Utiliser des noms simples, notamment pour les interfaces: une interface comportant la méthode Print s'appellera Printer, Write deviendra Writer, Read deviendra Reader, etc...

Enfin, l'utilitaire **gofmt** permet de formater un code source selon les recommendations des concepteurs du langage Go.

Panic et recover

Panic

Lorsqu'une erreur survient dans un appel de fonction, le meilleur comportement est en général de signaler l'erreur par un code retour. Mais parfois une erreur est irréversible et doit entraîner l'arrêt du programme. Dans ce cas, il convient d'appeler la fonction **panic()**. Cette fonction prend un seul argument de type quelconque.

Exemple:

```
var user = os.Getenv("USER")
```

```
func init() {
   if user == "" {
       // le programme se termine abruptement
       panic("pas de valeur pour $USER")
   }
}
```

Recover

Cette fonction permet de récupérer une exception lancée par "panic". Elle s'utilise en général avec defer:

```
func serveur(canal <-chan *Tache) {
   for tache := range canal {
       go faitEnSecurite(tache)
   }
}

func faitEnSecurite(tache *Tache) {
   defer func() {
       if err := recover(); err != nil {
           log.Println("tache echouee:", err)
       }
   }()
   faire(tache)
}
```

Ici, la fonction "faitEnSecurite" applique au résultat de "faire()" la fonction déclarée avec defer, qui appelle **recover()** et vérifie si une erreur a eu lieu. L'exception est à ce moment totalement levée et il n'y a rien d'autre à faire pour continuer l'exécution normale du programme.

Goroutines

Définition

Une goroutine est un fil d'éxécution tournant en parallèle avec d'autres goroutines à l'intérieur du même espace d'adressage (le processus).

Les goroutines permettent de paralléliser des tâches en les répartissant sur plusieurs coeurs ou processeurs. Le mot clé **go** permet de lancer un appel de fonction en une goroutine, et de ne pas attendre le résultat:

```
go list.Sort() // trie une liste en parallèle
```

Il est courant d'utiliser une fonction littérale pour appeler une goroutine:

```
go func(arguments, ...) {
  commandes...
} (paramètres, ...)
```

Les canaux (de communication)

Un canal est un type de base de Go, et se déclare en plaçant **chan** avant le type de ses éléments. On peut créer un canal avec make:

```
c := make(chan int)
```

Ensuite on peut envoyer et recevoir des données du type précisé, et la première donnée reçue est la première donnée envoyée. (*First In First Out*)

On dit qu'on envoie une donnée à travers un canal avec **canal<-valeur** et qu'on reçoit une donnée d'un canal avec = **<-canal**.

Dans cet exemple, on lance une goroutine et on attend sa fin en recevant le signal qu'elle va envoyer:

```
go func() {
   list.Sort()
   c <- 1  // Envoie un signal, la valeur importe peu
}()
blahBlahPendantUnMoment()
<-c   // Attend la fin du tri de la liste à la réception du signal
```

Canaux avec tampon

On peut créer un canal avec un tampon en fournissant la taille du tampon en argument de make().

Lorsque le canal n'a pas de tampon, la goroutine qui enfile une valeur doit attendre qu'une autre goroutine dépile cette valeur. En revanche, avec un tampon, la goroutine qui enfile une donnée attend seulement que le tampon dispose d'une case vide pour y mettre la donnée.

Dans l'exemple suivant, MaxTaches tâches peuvent se dérouler en parallèle:

```
var sem = make(chan int, MaxTaches)

func handle(r *Requete) {
   sem <- 1    // attend que sem soit disponible
   process(r)  // cela peut prendre un certain temps.
   <-sem       // cela permet à la prochaine tâche de démarrer
}

func Serve(file chan *Requete) {
   for {
       requete := <-file
       go handle(requete)  // lance la goroutine
   }
}
```

La syntaxe suivante permet de savoir si une case est disponible dans le canal:

```
v,ok = <- canal
```

Select

Select est une structure de contrôle similaire à switch. Elle permet de gérer une multitude d'envois et de

réceptions à travers des canaux différents.

Voici sa syntaxe:

```
select {
case canal <- valeur: commandes...
case var = <-canal: ...
case var := <-canal: ...
default: ...
}
```

Les différentes alternatives sont évaluées dans l'ordre de leur écriture, la première réception ou émission dans un canal déclenche l'exécution des commandes correspondant à l'alternative. Si aucune de ces opérations ne réussit, le cas par défaut est exécuté, sinon l'opération bloque jusqu'à la réception ou l'envoi d'une donnée.

Le choix entre les différentes alternatives est basé sur un processus pseudo-aléatoire.

Enfin, une structure **select** sans aucune alternative bloque éternellement.

Définir le nombre de processeurs

En définissant la variable d'environnement GOMAXPROCS comme étant égale au moins au nombre de processeurs, ou en appelant runtime.GOMAXPROCS() avec ce nombre, on peut répartir les goroutines sur les différents cœurs ou processeurs sur un ordinateur particulier.