Développement Go: Les types de données

Stéphane Karraz
ESGI - semaine thématique 2021

Introduction

Dans un programme, nous interagissons avec la mémoire de notre machine à travers nos variables.

Il existe deux grands types de variables: les variables primaires ayant des tailles fixes, et les variables composites.

Types primaires

bool

Il s’agit d’un booléen, dont les valeurs ne peuvent être que true ou false

var res bool = true

res2 := false

int

Représente un nombre entier en mémoire.

var nb int = 12

Il existe les types sous-jacents suivants:

int8, int16, int32, int64

Le nombre en suffixe correspond aux nombres d’octets occupés en mémoire.

Il existe le type rune qui est un alias du type int32 et qui représente un symbole en unicode.

uint

Représente un nombre entier non-signé: uniquement des nombres positifs ou nulles.

var uint nb = 62

Comme pour les int, nous avons aussi des types sous-jacents:

uint8, uint16, uint32, uint64

Il existe le type byte qui est un alias du type uint8

float32 et float64

Pour représenter un nombre décimal, nous avons ces deux types qui permettent différents niveau de précision selon la place prise en mémoire.

var nb float32 = 6.4

complex64 et complex128

Représente des types de nombres scalaires, par exemple (2+3i)

var z complex128 = cmplx.Sqrt(-5 + 12i)

Comme pour les float, selon la précision nécessaire, on pourra utiliser un type plutôt que l’autre.

Variables composites

Types non référencés

En Go, il existe deux types composites non référencés: les tableaux (array) et les structures.

Cela veut dire que lorsque vous passez ces types de variables en paramètre d’une fonction, le contenu sera dupliqué et passé en copie comme les variables primaires.

Structures

Cela correspond à une collection de champs de types différents. On peut par exemple créer une structure pour représenter l'état d'un jeu: la partie est-elle terminée ? Combien y a t'il de joueurs ? Quels sont les noms des joueurs ?...

Exemple:

```
type game struct {
	isEnd		bool
	nbPlayers	int
	playersNames	[]string
}

var mygame game
fmt.Println(mygame)
```

Structures - 2

Même si cela ne correspond pas à des classes d'objet, nous pouvons tout de même écrire du code orienté objet dans une certaine mesure.

Il est possible de créer des méthodes liés à une structure, et nous pouvons aussi faire de la composition d'objet même si l'héritage n'existe pas.

Pour déclarer une méthode sur notre structure game:

```
func (myGame *game) addPlayer(name string) {
    myGame.playersNames = append(myGame.playersNames, name)
    myGame.nbPlayers++
}
```

Ici, myGame représente le mot "self" que vous connaissez en JS ou en Python.

Tableaux

Un tableau est une suite de valeurs en mémoire du même type.

On les déclare comme ceci:

var myArray [len]Type

Ou encore

myArray := [len]Type{value1, value2 ...}

Ici, “len” est la taille du tableau correspondant au nombre de valeurs d’un certain Type. Exemple:

myNumbers := [5]int{1, 2, 3, 4, 5}

Tableaux - 2

Un tableau aura une taille fixe en mémoire.

Nous ne pouvons réassigner un autre tableau d’une taille différente à une variable correspondant déjà à un tableau même s’il est du même type.

Il n’est pas non plus possible de rajouter un élément au-delà de la capacité du tableau.

Les références

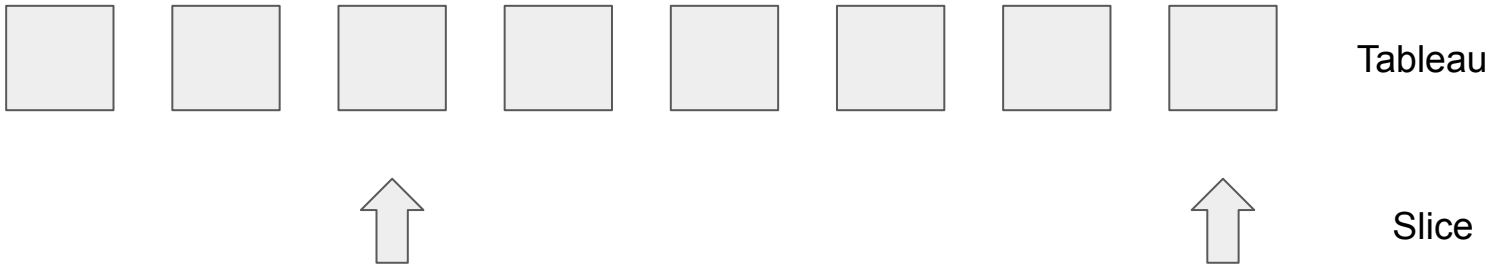
Il existe différents types de référence en Go:

- les slices
- les pointeurs
- les maps
- les fonctions
- les channels

Les slices

Pour déclarer un tableau dont l'espace mémoire est dynamique, on utilise les slice.

Cela peut se représenter comme deux curseurs sur un tableau, indiquant qu'on utilise une partie de ce tableau située entre le curseur du début et le curseur de fin.



Slices - 2

Le type d’une slice correspond au types des éléments du tableau sous-jacent.

On peut le déclarer ainsi:

var slice [ ]Type

Le compilateur va alors traduire ça en interne par la création d’un tableau d’une taille fixe, puis une slice par-dessus ayant une longueur de 0 et une capacité de 0.

On va pouvoir rajouter des éléments grâce à la fonction append:

```
var slice [ ]int
slice = append(slice, 2)
```

Slices - 3

Il existe une fonction nommée make nous permettons d’initialiser une slice avec une certaine taille et capacité:

make([]TYPE, length, capacity)

length correspond au nombre de valeurs présente à l’initialisation

capacity correspond à l’espace supplémentaire disponible pour accueillir de nouvelles valeurs avant de devoir allouer un nouvelle espace en mémoire pour avoir la place nécessaire.

Pointeurs

C’est simplement une variable ayant pour valeur une adresse en mémoire d’une autre variable.

On peut récupérer cette adresse à l’aide de **&**

ptr := &variable

Si la variable est de type “Type”, le pointeur sera un *Type:

```
var nb int := 5
ptr := &nb // ptr a pour type *int
```

On peut accéder à la valeur correspondant en mémoire grâce au déréférencement:

```
fmt.Println(*ptr)       // 5
```

Pointeurs - 2

Si vous voulez créer un nouveau pointeur sans créer de variable préalable, en Go cela est possible:

ptr := new(Type)

Ainsi, nous avons alloué l’espace nécessaire au stockage d’un int, et avons attribué l’adresse de cet espace à notre pointeur. Exemple:

```
ptr := new(int)
*ptr = 25
fmt.Println(*ptr)     // 25
```

Maps

Correspond à un ensemble clef-valeur.

On peut les déclarer en indiquant le type des clefs et le type des valeurs:
var myPlayersScores map[string]int

On peut aussi initialiser une map grâce à make:
myPlayersScores := make(map[string]int)

Maps - 2

Ajouter une valeur:
myPlayersScores[“me”] = 9000

Récupérer une valeur:
v := myPlayersScores[“me”]

Supprimer une valeur:
delete(myPlayersScores, “me”)

Maps - 3

À la différence des tableaux, il s’agit d’un type de référence. Ainsi lorsqu’on écrit qu’une map est égale à une autre, on ne va pas dupliquer de valeur: les deux maps correspondront à la même zone mémoire, et modifier l’un affectera donc l’autre.

Fonctions

Il est tout à fait possible d'utiliser un type de fonction comme paramètre. Ainsi il est possible de passer une fonction comme n'importe quel autre type de variable, permettant facilement d'écrire du code selon le paradigme fonctionnel.

On peut aussi déclarer une fonction dans le scope d'une autre et l'attribuer à une variable:

```
func main() {
	add := func(x, y int) int {
	return x + y
	}
	fmt.Println(add(1, 2))
}
```

Exercices

Players list

Créer un programme demandant d’abord le nom, puis le prénom et enfin le pseudo d’un joueur pour l’ajouter à une partie. La structure d’un player devra aussi comporter un score et ses points de vie. À chaque nouvel ajout de joueur, un résumé de la partie devra s’afficher:

Il y a 2 joueurs.
Le joueur 1 s’appelle AAA bbb, il porte le pseudo CcCcC et possède un score de 0.
Le joueur 2 s’appelle UUUUU bcd, il porte le pseudo NaNa possède un score de 0.

Game start

Compléter le programme précédent pour demander s'il on veut ajouter un nouveau joueur ou si le jeu peut commencer.

S'il y a un joueur au moins, on peut démarrer la partie.

But du jeu:
Un nombre est tiré aléatoirement entre 0 et 100, le but du jeu est de le deviner à tour de rôle. Si le nombre proposé par un joueur est incorrect, le jeu répond "Trop grand" ou "Trop petit" et passe au joueur suivant, si le joueur a deviné correctement un score lui est attribué en fonction du nombre de tentative au total. Un rappel des scores est affichés, puis un nouveau numéro est tiré et la partie continue en repartant du gagnant. Une partie se joue en 5 numéros.

High Scores

Sauvegarder les meilleurs scores dans un fichier local.

Si un fichier de high scores existe au lancement du jeu, on doit afficher un premier message listant les scores présents avant de démarrer l’enregistrement des joueurs.

Lorsqu’une partie se termine, ce fichier doit être mis à jour en conséquence.