Go : collections et itération

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Go

Les collections et parcours en **Go**

- Go ne possède pas une notion unique d'*itérable* comme Python.
- Les parcours se font principalement avec la boucle `for` et le mot-clé `range`.
- Les structures les plus utilisées sont les tableaux (`array`), les tranches (`slice`), les maps (`map`) et les chaînes (`string`).

Go

Exemples de collections en **Go**

- `array` : collection de taille fixe, typée, indexée et ordonnée.
- `slice` : vue dynamique sur un tableau, typée, indexée et ordonnée.
- `map` : collection de couples clé/valeur, non ordonnée.
- `string` : séquence immuable d'octets UTF-8, parcourable caractère par caractère avec `range`.

Go

À ne pas confondre

- **Tableau** (`array`) : taille connue et incluse dans le type, par exemple `[4]int`.
- **Slice** : taille variable, syntaxe `[]int`, structure la plus utilisée pour représenter une liste.
- **Map** : accès par clé, pas par indice numérique.

Go

Tableau comparatif des collections

Type	Ordonné	Modifiable	Doublons	Syntaxe
`array`	Oui	Oui	Oui	`[n]T`
`slice`	Oui	Oui	Oui	`[]T`
`map`	Non	Oui	Clés uniques	`map[K]V`
`string`	Oui	Non	Oui	`"..."`

Go

Pour déclarer un slice

```
nombres := []int{2, 3, 8, 5}
```

Go

Pour accéder à un élément selon son indice (premier élément d'indice 0)

```
fmt.Println(nombres[1])
// affiche 3
```

Go

Pour accéder à un élément selon son indice (premier élément d'indice 0)

```
fmt.Println(nombres[1])
// affiche 3
```

Pour accéder au dernier élément

```
fmt.Println(nombres[len(nombres)-1])
// affiche 5
```

Go

Accéder à un élément via un indice inexistant génère une erreur

```
fmt.Println(nombres[10])
// panic: runtime error: index out of range
```

Go

Accéder à un élément via un indice inexistant génère une erreur

```
fmt.Println(nombres[10])
// panic: runtime error: index out of range
```

Pour déterminer la taille d'un slice

```
fmt.Println(len(nombres))
// affiche 4
```

Go

Pour extraire un sous-slice

```
fmt.Println(nombres[1:3])
// affiche [3 8]

fmt.Println(nombres[2:])
// affiche [8 5]

fmt.Println(nombres[:2])
// affiche [2 3]
```

Go

Pour tester si un élément est dans le slice

```
trouve := false
for _, valeur := range nombres {
    if valeur == 3 {
        trouve = true
        break
    }
}
fmt.Println(trouve)
// affiche true
```

Go

Pour ajouter un élément à la fin

```
nombres = append(nombres, 9)
fmt.Println(nombres)
// affiche [2 3 8 5 9]
```

Go

Pour ajouter un élément à la fin

```
nombres = append(nombres, 9)
fmt.Println(nombres)
// affiche [2 3 8 5 9]
```

Pour fusionner deux slices

```
nombres = append(nombres, []int{1, 6}...)
fmt.Println(nombres)
// affiche [2 3 8 5 1 6]
```

Go

L'instruction suivante permet de modifier l'élément d'indice 2

```
nombres[2] = 10
fmt.Println(nombres)
// affiche [2 3 10 5]
```

Go

L'instruction suivante permet de modifier l'élément d'indice 2

```
nombres[2] = 10
fmt.Println(nombres)
// affiche [2 3 10 5]
```

Mais ne permet pas d'ajouter si l'indice n'est pas dans le slice

```
nombres[4] = 4
// erreur si l'indice 4 n'existe pas
```

Go

Pour supprimer le dernier élément

```
nombres = nombres[:len(nombres)-1]
fmt.Println(nombres)
// affiche [2 3 8]
```

Go

Pour supprimer le dernier élément

```
nombres = nombres[:len(nombres)-1]
fmt.Println(nombres)
// affiche [2 3 8]
```

Pour supprimer un élément selon son indice

```
i := 1
nombres = append(nombres[:i], nombres[i+1:]...)
fmt.Println(nombres)
// affiche [2 8 5]
```

Go

Pour parcourir un slice

```
for _, valeur := range nombres {
    fmt.Println(valeur)
}
```

Go

Pour parcourir un slice

```
for _, valeur := range nombres {
    fmt.Println(valeur)
}
```

Pour afficher les éléments et leurs indices

```
for indice, valeur := range nombres {
    fmt.Println(indice, valeur)
}
```

Go

Pour parcourir dans le sens inverse

```
for i := len(nombres) - 1; i >= 0; i-- {
    fmt.Println(nombres[i])
}
```

Go

Pour parcourir dans le sens inverse

```
for i := len(nombres) - 1; i >= 0; i-- {
    fmt.Println(nombres[i])
}
```

Pour parcourir avec une boucle classique

```
for i := 0; i < len(nombres); i++ {
    fmt.Println(i, nombres[i])
}
```

Go

Fonctions et opérations utiles sur les slices

- `len(s)` : retourne le nombre d'éléments.
- `cap(s)` : retourne la capacité du slice.
- `append(s, x)` : ajoute un élément.
- `copy(dst, src)` : copie des éléments.
- `slices.Sort(s)` : trie un slice avec le package `slices`.

Go

Exercice 1

Étant donné le slice suivant :

```
nombres := []int{2, 7, 2, 1, 3, 9, 2, 4, 2}
```

Go

Exercice 1

Étant donné le slice suivant :

```
nombres := []int{2, 7, 2, 1, 3, 9, 2, 4, 2}
```

Écrire un programme Go qui permet de supprimer l'avant-dernière occurrence du chiffre 2.
Le résultat attendu : `[2 7 2 1 3 9 4 2]`

Go

Une solution possible

```
nombres := []int{2, 7, 2, 1, 3, 9, 2, 4, 2}
compteur := 0
indiceASupprimer := -1

for i := len(nombres) - 1; i >= 0; i-- {
    if nombres[i] == 2 {
        compteur++
        if compteur == 2 {
            indiceASupprimer = i
            break
        }
    }
}

if indiceASupprimer != -1 {
    nombres = append(nombres[:indiceASupprimer], nombres[
       indiceASupprimer+1:]...)
}
fmt.Println(nombres)
```

Go

Pour créer un slice vide

```
var nombres []int
// slice nil

nombres2 := []int{}
// slice vide mais non nil
```

Go

Pour créer un slice vide

```
var nombres []int
// slice nil

nombres2 := []int{}
// slice vide mais non nil
```


Pour créer un slice avec make

```
notes := make([]float64, 0)
notesAvecTaille := make([]float64, 10)
```

Go

Exercice

Écrire un programme Go qui :

1. demande à l'utilisateur de remplir un slice de notes ; la saisie s'arrête si la valeur n'est pas comprise entre 0 et 20
2. affiche le max, le min et la moyenne des notes.

Go

Tableau en Go

- Un tableau possède une taille fixe.
- La taille fait partie du type : `[3]int` et `[4]int` sont deux types différents.
- En pratique, on utilise plus souvent les slices.

Go

Pour déclarer et initialiser un tableau

```
var tab [4]int
fmt.Println(tab)
// affiche [0 0 0 0]

tab2 := [4]int{2, 3, 8, 5}
fmt.Println(tab2)
```

Go

Pour déclarer et initialiser un tableau

```
var tab [4]int
fmt.Println(tab)
// affiche [0 0 0 0]

tab2 := [4]int{2, 3, 8, 5}
fmt.Println(tab2)
```


Laisser Go déduire la taille

```
tab3 := [...]int{2, 3, 8, 5}
fmt.Println(len(tab3))
// affiche 4
```

Go

Pour parcourir un tableau

```
for i, valeur := range tab3 {
    fmt.Println(i, valeur)
}
```

Go

Slice multi-dimensions ≡ matrice

- Slice de slices.
- Chaque sous-slice correspond à une ligne de la matrice.
- Les lignes peuvent avoir des tailles différentes.

Go

Pour créer une matrice 2x2 et 3x3

```
matrice2 := [][]int{
    {1, 2},
    {3, 4},
}

matrice3 := [][]int{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
}
```

Go

Pour accéder à un élément

```
fmt.Println(matrice3[0][0])
// affiche 1
```

Go

Pour accéder à un élément

```
fmt.Println(matrice3[0][0])
// affiche 1
```

Pour parcourir la matrice ligne par ligne

```
for _, ligne := range matrice3 {
    fmt.Println(ligne)
}
```

Go

Pour parcourir la matrice élément par élément

```
for i := range matrice3 {
    for j := range matrice3[i] {
        fmt.Print(matrice3[i][j], " ")
    }
}
```

Go

Map en Go

- Une map associe des clés à des valeurs.
- Les clés doivent être comparables : `string`, `int`, `bool`...
- L'ordre de parcours d'une map n'est pas garanti.

Go

Pour déclarer et initialiser une map

```
ages := map[string]int{
    "Ali": 23,
    "Sara": 31,
}
fmt.Println(ages)
```

Go

Pour déclarer et initialiser une map

```
ages := map[string]int{
    "Ali": 23,
    "Sara": 31,
}
fmt.Println(ages)
```


Pour créer une map vide

```
ages := make(map[string]int)
ages["Ali"] = 23
```

Go

Pour accéder à une valeur

```
fmt.Println(ages["Ali"])
// affiche 23
```

Go

Pour accéder à une valeur

```
fmt.Println(ages["Ali"])
// affiche 23
```

Pour tester l'existence d'une clé

```
age, existe := ages["Sara"]
if existe {
    fmt.Println(age)
}
```

Go

Pour modifier ou ajouter une valeur

```
ages["Ali"] = 24
ages["Nora"] = 19
```

Go

Pour modifier ou ajouter une valeur

```
ages["Ali"] = 24
ages["Nora"] = 19
```

Pour supprimer une entrée

```
delete(ages, "Ali")
```

Go

Pour modifier ou ajouter une valeur

```
ages["Ali"] = 24
ages["Nora"] = 19
```

Pour supprimer une entrée

```
delete(ages, "Ali")
```

Pour parcourir une map

```
for cle, valeur := range ages {
    fmt.Println(cle, valeur)
}
```

Go

Attention à l'ordre des maps

- Une map Go est volontairement parcourue dans un ordre non garanti.
- Si un ordre est nécessaire, il faut extraire les clés, les trier, puis parcourir selon ces clés.
- Cette différence est importante pour les affichages, les tests et la génération de fichiers.

Go

Chaînes de caractères en Go

- Une chaîne est immuable : on ne modifie pas directement un caractère.
- `len(s)` retourne un nombre d'octets, pas forcément un nombre de caractères.
- `range` permet de parcourir les runes Unicode.

Go

Pour déclarer une chaîne

```
texte := "Bonjour"
fmt.Println(texte)
```

Go

Pour déclarer une chaîne

```
texte := "Bonjour"
fmt.Println(texte)
```

Pour accéder à un octet

```
texte := "Go"
fmt.Println(texte[0])
// affiche 71, le code ASCII de G
```

Go

Pour parcourir les caractères avec range

```
texte := "été"
for indice, r := range texte {
    fmt.Println(indice, string(r))
}
```

Go

Pour parcourir les caractères avec range

```
texte := "été"
for indice, r := range texte {
    fmt.Println(indice, string(r))
}
```

Pour transformer en slice de runes

```
runes := []rune("été")
runes[0] = 'é'
texte = string(runes)
fmt.Println(texte)
```

Go

Variable, valeur et référence

- En Go, l'affectation copie une valeur.
- Un tableau est copié entièrement lors d'une affectation.
- Un slice contient un pointeur vers un tableau sous-jacent : deux slices peuvent partager les mêmes données.
- Une map est un type référence : une affectation partage la même structure interne.

Go

Copie avec les tableaux

```
a := [3]int{1, 2, 3}
b := a
b[0] = 99
fmt.Println(a) // [1 2 3]
fmt.Println(b) // [99 2 3]
```

Go

Copie avec les tableaux

```
a := [3]int{1, 2, 3}
b := a
b[0] = 99
fmt.Println(a) // [1 2 3]
fmt.Println(b) // [99 2 3]
```


Partage avec les slices

```
a := []int{1, 2, 3}
b := a
b[0] = 99
fmt.Println(a) // [99 2 3]
```

Go

Copier réellement un slice

```
a := []int{1, 2, 3}
b := make([]int, len(a))
copy(b, a)
b[0] = 99
fmt.Println(a) // [1 2 3]
fmt.Println(b) // [99 2 3]
```

Go

Copier réellement un slice

```
a := []int{1, 2, 3}
b := make([]int, len(a))
copy(b, a)
b[0] = 99
fmt.Println(a) // [1 2 3]
fmt.Println(b) // [99 2 3]
```


Partage avec les maps

```
m1 := map[string]int{"a": 1}
m2 := m1
m2["a"] = 99
fmt.Println(m1["a"])
// affiche 99
```

Go

Bonnes pratiques

- Utiliser les slices pour les listes de taille variable.
- Utiliser les tableaux uniquement lorsque la taille fixe a un sens métier ou technique.
- Tester l'existence d'une clé dans une map avec la forme `valeur, ok := m[cle]`.
- Ne jamais supposer l'ordre de parcours d'une map.
- Être vigilant avec le partage de données entre slices.

Go

Mini TP

Écrire un programme Go qui gère une petite liste de produits :

1. chaque produit possède un nom et un prix
2. stocker les produits dans un slice
3. afficher tous les produits avec leur indice
4. calculer le prix total
5. construire une map qui associe le nom du produit à son prix
6. rechercher un produit par son nom.

Go

Point de départ possible

```
type Produit struct {
    Nom  string
    Prix float64
}

produits := []Produit{
    {Nom: "Clavier", Prix: 29.90},
    {Nom: "Souris", Prix: 14.90},
    {Nom: "Ecran", Prix: 159.90},
}
```