Les Génériques en Go

Michel Casabianca
casa@sweetohm.net

La version *1.18* du Go vient de paraître et les Génériques en sont la fonctionnalité la plus importante. Je vous propose d'en faire un tour rapide dans cet article.

Avant Go 1.18

Il a toujours été possible d'écrire du code générique en Go à l'aide du type **interface{}**. Par exemple, on peut écrire une fonction qui affiche *n* fois une valeur quelconque avec :

```
package main

import "fmt"

func Repeat(something interface{}, times int) {
        for i := 0; i < times; i++ {
                fmt.Println(something)
        }
}

func main() {
        Repeat("Hello World!", 3)
    Repeat(42, 3)
}
```

Sur le Playground

Cet exemple est particulièrement simple car la fonction `fmt.Println()` accepte tout type. Avant Go *1.18*, sa signature était : `func Println(a ...interface{}) (n int, err error)`.

D'autre part, on peut définir le type d'un argument avec une interface spécifique. Par exemple:

```
package main

import (
        "errors"
        "strconv"
)

type Failure int

func (t Failure) Error() string {
        return strconv.Itoa(int(t))
}
```

```
func PrintError(err error) {
        println("error: " + err.Error())
}

func main() {
        PrintError(errors.New("This is a test!"))
        PrintError(Failure(42))
}
```

Sur le Playground

Le type `error` est une interface qui définit la méthode `Error() string`. On peut donc envoyer n'importe quoi à la fonction `PrintError()` pourvu que ça implémente une méthode `Error()`.

Le début des ennuis

Supposons que nous voulions écrire une fonction qui renvoie le maximum de deux valeurs. Nous pouvons l'écrire, pour les entiers, comme suit :

```
package main

func Max(x, y int) int {
        if x > y {
                return x
        }
        return y
}

func main() {
        println(Max(1, 2))
}
```

Sur le Playground

Si nous voulons généraliser cette fonction à d'autres types, les interfaces ne nous sont pas d'un grand secours car aucune fonction ne définit les opérateurs de comparaison. Nous devrons donc **réécrire cette fonction pour tous les types** ! Il serait possible d'accepter en entrée le type `interface{}` mais nous devons alors faire des **assertions sur les types** et cela ne simplifierait pas les chose.

Les Generics à la rescousse

Go *1.18* implémente les *Generics*. On peut maintenant ajouter des paramètres de type (*type parameters* en anglais) à la signature d'une fonction. Pour pouvoir rendre notre fonction `Max()` générique, nous pourrions écrire :

```
package main

func Max[N int | float64](x, y N) N {
```

```
        if x > y {
                return x
        }
        return y
}

func main() {
        println(Max(1, 2))
        println(Max(1.2, 2.1))
}
```

Sur le Playground

Ainsi, avec le paramètre de type `[N int | float64]`, nous indiquons que les paramètres de la fonction sont du type `int` ou `float64`. À noter que l'on ne peut mélanger les types, donc l'appel `Max(1, 2.0)` provoque une erreur de compilation.

Le retour des interfaces

Il est aussi possible à partir de Go *1.18* de définir des interfaces comme une liste de types. Nous pourrions réécrire l'exemple précédent de la manière suivante :

```
package main

type Number interface {
        int | int16 | int32 | int64 | float32 | float64
}

func Max[N Number](x, y N) N {
        if x > y {
                return x
        }
        return y
}

func main() {
        println(Max(1, 2))
        println(Max(1.2, 2.1))
}
```

Sur le Playground

Les alias de types

Si nous définissons un alias pour un type, nous pouvons l'englober dans une liste avec le caractère `~`, comme suit :

```
package main

type Number interface {
```

```
        ~int | ~int16 | ~int32 | ~int64 | ~float32 | ~float64
}

type Truc int

func Max[N Number](x, y N) N {
        if x > y {
                return x
        }
        return y
}

func main() {
        println(Max(Truc(1), Truc(2)))
}
```

Sur le Playground

Ainsi par exemple, `~int` englobe le type `int` mais aussi tous ses alias, dont `Truc`.

Contraintes

Il peut être laborieux de définir ainsi ses propres interfaces avec des listes de types. Le package *golang.org/x/exp/constraints* en propose un certain nombre :

- **Signed** : *~int* | *~int8* | *~int16* | *~int32* | *~int64*
- **Unsigned** : *~uint* | *~uint8* | *~uint16* | *~uint32* | *~uint64* | *~uintptr*
- **Integer** : *Signed* | *Unsigned*
- **Float** : *~float32* | *~float64*
- **Ordered** : *Integer* | *Float* | *~string*
- **Complex** : *~complex64* | *~complex128*

Nous pouvons alors utiliser la *contrainte constraints.Ordered* comme suit :

```
package main

import "golang.org/x/exp/constraints"

func Max[N constraints.Ordered](x, y N) N {
        if x > y { return x }
        return y
}

func main() {
        println(Max("abc", "def"))
}
```

Sur le Playground

D'autre part, Go *1.18* définit deux autres contraintes :

- **any** qui est identique à `interface{}`
- **comparable** pour les types qui peuvent être comparés avec les opérateurs == et `!=`

Instantiation

Il est possible de passer le type d'argument lors de l'appel d'une fonction générique. On pourra par exemple faire l'appel :

```
m := Max[int](1, 2)
```

L'expression `Max[int]` est une *instantiation* de la fonction générique `Max`. Elle fixe les types des paramètres. On peut par exemple écrire :

```
MaxFloat := Max[float64]
m := MaxFloat(1.0, 2.0)
```

La fonction `MaxFloat` est maintenant une fonction non générique qui n'accepte que des paramètres de type `float`.

Types avec paramètres de type

Supposons que nous voulions faire la somme des valeurs des éléments d'une liste. Nous pourrions écrire, avec la liste chaînée standard du Go :

```
package main

import "container/list"

func main() {
        list := &list.List{}
        list.PushBack(1)
        list.PushBack(2)
        list.PushBack(3)
        sum := 0
        for e := list.Front(); e != nil; e = n.Next() {
                sum += e.Value
        }
        println(sum)
}
```

Sur le Playground

Cela ne compile pas car on ne peut faire une somme avec le type `interface{}` qui est celui de la valeur des éléments d'une liste : `src/list.go:12:3: invalid operation: sum += e.Value (mismatched types int and any)`.

Utiliser le type `interface{}` ou `any` est ennuyeux car nous devons caster les valeurs pour pouvoir les utiliser. Il y a bien sûr une solution à base de Generics. Voici une implémentation

minimaliste de liste avec des génériques :

```
package main

type Element[T any] struct {
        Next  *Element[T]
        Value T
}

type List[T any] struct {
        Front *Element[T]
        Last  *Element[T]
}

func (l *List[T]) PushBack(value T) {
        node := &Element[T]{
                Next:  nil,
                Value: value,
        }
        if l.Front == nil {
                l.Front = node
                l.Last = node
        } else {
                l.Last.Next = node
                l.Last = node
        }
}

func main() {
        list := &List[int]{}
        list.PushBack(1)
        list.PushBack(2)
        list.PushBack(3)
        sum := 0
        for n := list.Front; n != nil; n = n.Next {
                sum += n.Value
        }
        println(sum)
}
```

Sur le Playground

Dans ce code nous avons ajouté des paramètres de type aux définitions des types, comme dans `Element[T any]`. Cette notation indique que nous définissons un type `Element` qui contient le type `T` qui peut être quelconque. Nous pouvons alors utiliser ces valeurs sans avoir à les caster.

Il est important de noter que nous avons fixé le type de la liste lors de l'instanciation :

```
list := &List[int]{}
```

Nous avons ainsi indiqué que notre liste contient des `int` et nous pouvons alors les manipuler comme tels.

Inférence de type

Nous avons vu que nous pouvons fixer le type des paramètres lors d'un appel à une fonction générique avec :

```
m := Max[int](1, 2)
```

Dans ce cas, le compilateur sait le type des paramètres parce qu'on lui indique. Mais lorsque nous écrivons :

```
m := Max(1, 2)
```

Le compilateur **infère le type des paramètres** de la fonction générique de celui des arguments lors de l'appel. Ce type d'inférence est appelé en anglais *function argument type inference*. Cependant, il est parfois impossible d'inférer le type de la valeur de retour, comme pour la fonction :

```
func NewT[T any]() *T {
    ...
}
```

Il faudra alors aider le compilateur en procédant à l'*instanciation* de la fonction avant l'appel :

```
t := NewT[int]()
```

Quand utiliser les Generics ?

La première recommandation est de ne **jamais définir des contraintes avant d'écrire le code**. Cela peut sembler séduisant d'anticiper et de commencer par définir des contraintes, mais c'est inutile.

Le cas d'usage des génériques est la factorisation de **code identique dupliqué avec plusieurs types**. C'est une alternative préférables à l'utilisation du type `interface{}` pour des questions de performance, d'occupation mémoire et de simplicité du code. Les génériques sont ainsi tout indiquées pour les structures de données, comme les *listes chaînées* ou les *arbres binaires* par exemple.

Conclusion

Les génériques sont la grande nouveauté du Go *1.18* qui est la release qui a amené le plus de changements depuis que le Go est Open Source. Cependant, cette fonctionnalité n'a pas été encore assez utilisée en production par un grand nombre d'utilisateurs et doit donc être **utilisée avec précaution**, et bien sûr largement couverte de tests.

Generics Gopher