16

Fonctions anonymes et closures

Les fonctions anonymes constituent un élément fondamental de la programmation en Go. Contrairement aux fonctions nommées, elles sont définies sans identificateur et peuvent être assignées à des variables, passées en paramètre ou retournées par d'autres fonctions.

Une closure est une fonction anonyme qui capture des variables de son environnement lexical. L'environnement lexical désigne l'ensemble des variables accessibles au point où la closure est définie. Cette capacité de capture a des implications importantes sur la gestion mémoire et les performances.

Toute fonction anonyme en Go utilise le même mécanisme interne qu'une closure, même si elle ne capture aucune variable. L'analyse du code compilé révèle que le compilateur génère systématiquement la même représentation : une structure contenant un pointeur vers le code et un pointeur vers le contexte. La distinction entre *fonction anonyme* et *closure* n'existe donc que dans la sémantique du code source : une fonction anonyme devient une closure dès lors qu'elle référence des variables externes, mais les deux partagent la même implémentation sous-jacente. Par commodité, le reste de ce chapitre utilisera le terme *closure* pour désigner indifféremment les fonctions anonymes avec ou sans capture.

16.1. Mécanisme de capture des variables

Fonctions anonymes : des fonctions à part entière

Hors optimisations du compilateur (notamment l'inlining), une fonction anonyme ou une closure est une fonction à part entière. Elle possède sa propre stack frame lors de son exécution, suit les mêmes conventions d'appel que les fonctions nommées, et peut avoir ses propres variables locales.

Cette indépendance crée un problème fondamental : comment une closure peut-elle accéder aux variables de son environnement lexical alors qu'elle s'exécute dans une stack frame différente de celle de sa fonction englobante ?

Considérons cet exemple :

```
func externe() func(int) int {
    x := 42
    return func(nb int) int {
        return x + nb  // Comment accéder à x ?
    }
}

runClosure := externe()
runClosure(1)
```

Au moment d'exécuter `runClosure(1)`, la fonction `externe` a déjà terminé son exécution. Sa stack frame n'existe plus, et pourtant la closure doit pouvoir accéder à `x`. De plus, même si la fonction `externe` était encore active, les deux fonctions auraient des stack frames distinctes qui ne pourraient pas partager directement des variables locales.

Le compilateur génère pour chaque fonction anonyme un bloc d'instructions assembleur distinct et autonome. Ce code n'est pas inséré au milieu de la fonction englobante, mais existe comme une fonction à part entière dans le binaire final. En désassemblant le code de l'exemple précédent, la closure apparaît sous le nom `main.externe.func1` :

```
0000000001067e80 <_main.externe.func1>:
 1067e80: 48 03 42 08                    addq  0x8(%rdx), %rax
 1067e84: c3                             retq
```

Cette fonction possède sa propre adresse en mémoire et peut être appelée comme n'importe quelle autre fonction.

Isolement des stack frames

Chaque appel de fonction crée une nouvelle stack frame contenant :

- les paramètres de la fonction ;
- les variables locales ;
- l'adresse de retour ;
- les registres sauvegardés.

Ces stack frames sont isolées les unes des autres. Une fonction ne peut pas accéder directement aux variables locales d'une autre fonction, même si cette dernière l'a appelée. C'est un principe fondamental de la sécurité mémoire et de l'organisation du programme.

Pour permettre à une closure d'accéder aux variables de son environnement, Go doit contourner cet isolement. Il dispose de deux stratégies : la capture par valeur et la cap-

ture par référence. Mais avant de détailler ces stratégies, il faut comprendre comment Go organise le partage de ces variables.

Structure d'une closure

Au niveau machine, une closure n'est pas simplement un pointeur de fonction. Elle est représentée par une structure contenant deux éléments :

- un pointeur vers le code de la fonction ;
- un pointeur vers un contexte contenant les variables capturées.

Cette structure occupe 16 octets sur une architecture 64 bits (deux pointeurs de 8 octets) ou 8 octets sur architecture 32 bits (deux pointeurs de 4 octets).

En écrivant `runClosure := externe()` Go crée cette structure en mémoire. Le premier champ pointe vers le code compilé de la fonction anonyme, le second vers une zone mémoire (le contexte) qui stocke les variables capturées.

Lors de l'appel `runClosure(1)`, la closure reçoit implicitement un pointeur vers son contexte comme paramètre supplémentaire, transmis selon la convention d'appel (via registre sur x86-64). Ce mécanisme est totalement transparent pour le développeur : il n'apparaît ni dans la signature de la fonction ni dans l'appel, mais permet à la closure d'accéder aux variables de son environnement sans violer l'isolement des stack frames.

Son contexte est une structure dont les champs correspondent aux variables capturées. Point important : certaines variables sont stockées directement dans cette structure (capture par valeur), d'autres y sont stockées sous forme de pointeurs (capture par référence).

Le compilateur identifie les variables capturées lors du type checking et les ajoute à la liste `Cvars` de la closure. Après l'analyse d'échappement, la fonction `capturevars` détermine pour chacune le mode de capture selon les critères établis précédemment.

Considérons cet exemple :

```
func exemple() func() {
    a := 4
    b := 4
    return func() {
        fmt.Println(a, b)
        b = 8
    }
}
```

Le compilateur analyse les usages de `a` et `b` :

- `a` n'est jamais réassigné → capture par valeur ;
- `b` est réassigné → capture par référence.

La structure de contexte ressemble alors à :

```
type context struct {
    a int   // valeur copiée directement
    b *int  // pointeur vers b sur le heap
}
```

Cette représentation en Go illustre l'organisation mémoire du contexte, mais cette structure n'existe pas explicitement dans le code compilé : elle décrit simplement la disposition des données en mémoire une fois le programme assemblé.

Comme toute allocation, la structure de la closure et le bloc de contexte sont soumis à l'analyse d'échappement : ils peuvent être alloués sur la stack si la closure ne survit pas à la fonction englobante, ou sur le heap si elle est échappée.

Dans le code de la closure, accéder à ces variables a des coûts différents :

- pour `a` : lecture directe depuis `ctx.a` (un déréférencement du pointeur `ctx`) ;
- pour `b` : lecture via `*ctx.b` (deux déréférencements : `ctx` puis le pointeur `b`).

L'instruction `b = 8` dans la closure se traduit donc en réalité par `*ctx.b = 8`, avec deux déréférencements et un accès au heap, là où un accès normal à une variable locale serait direct sur la stack.

Cette organisation explique pourquoi les variables capturées par référence sont plus coûteuses : elles nécessitent une allocation sur le heap, un déréférencement supplémentaire et peuvent créer des conflits d'accès en environnement concurrent. L'impact de ces mécanismes sur les performances est analysé en détail dans la section sur les benchmarks (voir l'annexe Section 15, Impact de la capture de variables par les closures).

Capture par valeur

La capture par valeur consiste à copier la valeur de la variable dans la structure de la closure. La closure travaille sur sa propre copie, indépendante de la variable originale.

```
func exemple() func() int {
    n := 10
    return func() int {
```