Tutoriel pour apprendre le langage Go par l'exemple

Par Clément Keirua

Date de publication : 17 avril 2017

Go est un langage de programmation open source conçu pour permettre de réaliser des programmes simples, rapides et fiables.

Le *Go par l'exemple* est une introduction pratique au Go avec des programmes d'exemple annotés. Regardez le **premier exemple** ou naviguez dans la liste complète ci-contre.

Commentez

I - Hello World

Notre premier programme va afficher le classique « hello world »

Voici le code source complet.

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

Exécuter le code en ligne

Pour exécuter le programme, mettez le code dans le fichier hello-world.go et lancez go run.

```
$ go run hello-world.go
hello world
```

Parfois on veut compiler pour obtenir un exécutable. Nous pouvons faire cela avec go build.

```
$ go build hello-world.go
$ ls
hello-world    hello-world.go
```

On peut ensuite exécuter le fichier binaire

```
$ ./hello-world
hello world
```

Maintenant que l'on sait exécuter et compiler des programmes en Go, apprenons-en plus sur le langage.

II - Valeurs

Go a plusieurs types de valeurs, incluant les chaînes de caractères, les entiers, les nombres flottants, les booléens, etc. Voici quelques exemples basiques.

Les chaînes de caractères, que l'on peut concaténer avec +.

```
    fmt.Println("go" + "lang")
```

Les entiers et les flottants.

```
    fmt.Println("1+1 =", 1+1)
    fmt.Println("7.0/3.0 =", 7.0/3.0)
```

Les booléens, avec les opérateurs booléens tels qu'on les attend.

```
    fmt.Println(true && false)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

Code complet

```
package main

import "fmt"
```

Exécuter le code en ligne

Code complet

```
func main() {

    fmt.Println("go" + "lang")

    fmt.Println("1+1 =", 1+1)
    fmt.Println("7.0/3.0
 =", 7.0/3.0)

    fmt.Println(true && false)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

```
$ go run values.go
golang
1+1 = 2
7.0/3.0 = 2.3333333333333335
false
true
false
```

III - Variables

En Go, les *variables* sont déclarées explicitement et utilisées par le compilateur, par exemple pour vérifier que le type de retour des appels de fonction est correct.

var déclare une ou plusieurs variables.

```
    var a string = "initial"
    fmt.Println(a)
```

On peut déclarer plusieurs variables à la fois

```
    var b, c int = 1, 2
    fmt.Println(b, c)
```

Go déduira le type des variables non initialisées

```
    var d = true
    fmt.Println(d)
```

Les variables déclarées sans être initialisées ont une *valeur nulle*. Par exemple, la valeur nulle d'un int est 0.

```
    var e int
    fmt.Println(e)
```

La syntaxe := est un raccourci pour déclarer et initialiser une variable, par exemple pour var f string = "short" ici.

```
    f := "short"
    fmt.Println(f)
}
```

Code complet

```
package main

import "fmt"

func main() {
```

Exécuter le code en ligne

Code complet

```
    var a string = "initial"
    fmt.Println(a)

    var b, c int = 1, 2
    fmt.Println(b, c)

    var d = true
    fmt.Println(d)

    var e int
    fmt.Println(e)

    f := "short"
    fmt.Println(f)
}
```

```
$ go run variables.go
initial
1 2
true
0
short
```

IV - Constantes

Go supporte les constantes de caractères, chaînes de caractères, booléens et valeurs numériques

const déclare une valeur constante.

```
const s string = "constant"

func main() {
    fmt.Println(s)
```

Le mot-clé **const** peut apparaître à chaque endroit où l'on peut mettre le mot-clé **var**

```
    const n = 500000000
```

Les expressions constantes réalisent les opérations arithmétiques avec une précision arbitraire.

```
    const d = 3e20 / n
    fmt.Println(d)
```

Une constante numérique n'a pas de type jusqu'à ce qu'on lui en donne un, par exemple via un cast explicite

```
    fmt.Println(int64(d))
```

On peut donner un type à un nombre en l'utilisant dans un contexte qui en requiert un, tel qu'une affectation ou un appel de fonction. Par exemple ici, math.Sin attend un float64.

```
    fmt.Println(math.Sin(n))
}
```

Code complet

```
package main

import "fmt"
import "math"

const s string = "constant"
```

Exécuter le code en ligne

Code complet

```
func main() {
    fmt.Println(s)

    const n = 500000000

    const d = 3e20 / n
    fmt.Println(d)

    fmt.Println(int64(d))

    fmt.Println(math.Sin(n))
}
```

```
$ go run constant.go
constant
6e+11
600000000000
-0.28470407323754404
```

V - Déclarations courtes

x := val est la version courte pour var x type = val.

```
func main() {
    x := "Hello var"
    fmt.Println(x)
}
```

Code complet

```
package main

import "fmt"

func main() {

    x := "Hello var"
    fmt.Println(x)
}
```

Exécuter le code en ligne

```
$ go run short-declarations.go
Hello var
```

VI - For

for est la seule manière de faire une boucle en Go. Voici trois types basiques de boucles for.

La plus simple, avec une unique condition.

```
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }
```

La boucle for classique, en trois étapes : initialisation/condition/incrémentation.

```
    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }
```

for sans condition va boucler indéfiniment, jusqu'à ce qu'on **break** pour en sortir, ou qu'un **return** fasse sortir de la fonction correspondante.

```
    for {
        fmt.Println("loop")
        break
    }
}
```

Code complet

```
package main

import "fmt"

func main() {

    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    for {
        fmt.Println("loop")
        break
    }
}
```

Exécuter le code en ligne

```
$ go run for.go
1
2
3
7
8
9
loop
```

Nous verrons d'autres formes de boucles **for** plus tard lorsque nous verrons le mot-clé **range**, les canaux et d'autres structures de données.

VII - If/Else

Les conditions avec **if** et **else** sont simples en Go

Voici un exemple simple.

```
    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }
```

On peut avoir un **if** sans **else**.

```
    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }
```

Une déclaration peut précéder la condition. Toute variable déclarée là est disponible dans toutes les branches.

```
    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

À noter qu'il n'y a pas besoin de parenthèses autour des conditions en Go, mais que les accolades sont requises.

Code complet

```
package main

import "fmt"

func main() {

    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    if 8%4 == 0 {
        fmt.Println("8 is
 divisible by 4")
    }

    if num := 9; num < 0 {
        fmt.Println(num, "is
 negative")
    } else if num < 10 {
        fmt.Println(num, "has 1
 digit")
    } else {
        fmt.Println(num, "has
 multiple digits")
    }
}
```

Exécuter le code en ligne

```
$ go run if-else.go
7 is odd
8 is divisible by 4
9 has 1 digit
```

Il n'y a pas de **if ternaire** en Go, donc il faut utiliser le **if** complet même pour des conditions basiques.

VIII - Switch

Les expressions Switch réalisent des conditions à travers plusieurs branches.

Voici un **switch** basique.

```
    i := 2
    fmt.Print("write ", i, " as ")
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }
```

On peut utiliser des virgules pour séparer plusieurs expressions dans un même **case**. Nous utilisons également le cas **default** optionnel dans cet exemple.

```
    switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("it's the weekend")
    default:
        fmt.Println("it's a weekday")
    }
```

switch sans expression est une autre manière de faire un if/else. Vous pouvez aussi voir que les expressions dans **case** peuvent être non constantes.

```
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("it's before noon")
    default:
        fmt.Println("it's after noon")
    }
```

Un switch sur les types **switch** compare des types et non des valeurs. Vous pouvez utiliser cela pour découvrir le type correspondant à la valeur d'une interface. Dans cet exemple, la variable t aura comme valeur le type de la variable i.

```
    whatAmI := func(i interface{}) string {
        switch t := i.(type) {
        case bool:
            return "I am a bool"
        case int:
            return "I am an int"
        default:
            return fmt.Sprintf("Can't handle type %T", t)
        }
    }
    fmt.Println(whatAmI(1))
    fmt.Println(whatAmI(true))
    fmt.Println(whatAmI("hey"))
}
```

Code complet

```
package main

import "fmt"
import "time"

func main() {

    i := 2
    fmt.Print("write ", i, "
 as ")
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }

    switch
 time.Now().Weekday() {
    case time.Saturday,
 time.Sunday:
        fmt.Println("it's the
 weekend")
    default:
```

Exécuter le code en ligne

Code complet

```
        fmt.Println("it's a
weekday")
    }

    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("it's before
noon")
    default:
        fmt.Println("it's after
noon")
    }


whatAmI := func(i interface{}) stri
        switch t := i.(type) {
        case bool:
            return "I am a bool"
        case int:
            return "I am an int"
        default:
            return
fmt.Sprintf("Can't handle type
%T", t)
        }
    }
    fmt.Println(whatAmI(1))
    fmt.Println(whatAmI(true))
    fmt.Println(whatAmI("hey"))
}
```

```
$ go run switch.go
write 2 as two
it's the weekend
it's before noon
I am an int
I am a bool
Can't handle type string
```

IX - Tableaux

En Go, un tableau est une séquence numérotée d'éléments d'une longueur donnée.

Ici, nous créons un tableau `a` qui contiendra exactement cinq `int`. Le type des éléments et la longueur font tous les deux partie du type du tableau. Par défaut, les valeurs des éléments du tableau sont nulles, c'est-à-dire 0 chez les `int`.

```
    var a [5]int
    fmt.Println("emp:", a)
```

On peut affecter la valeur à un indice particulier avec la syntaxe `array[index] = value`. On obtient sa valeur avec `array[index]`.

```
    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])
```

La fonction **len**, intégrée au langage, renvoie la longueur du tableau.

```
    fmt.Println("len:", len(a))
```

On peut utiliser cette syntaxe pour déclarer et initialiser un tableau en une ligne.

```
    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)
```

Les tableaux sont à une dimension, mais on peut les composer pour obtenir des structures de données multidimensionnelles.

```
    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

Code complet

Exécuter le code en ligne

```
package main

import "fmt"

func main() {

    var a [5]int
    fmt.Println("emp:", a)

    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])

    fmt.Println("len:", len(a))

    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

À noter que les tableaux apparaissent sous la forme [v1 v2 v3 ...] lorsqu'on les affiche avec fmt.Println.

```
$ go run arrays.go
emp: [0 0 0 0 0]
set: [0 0 0 0 100]
get: 100
len: 5
dcl: [1 2 3 4 5]
2d:  [[0 1 2] [1 2 3]]
```

En Go, on utilise les *slices* bien plus souvent que les tableaux. C'est ce que nous allons voir maintenant.

X - Slices

Les slices sont un type de données clé en Go. Ils sont bien plus puissants que les tableaux pour manipuler les séquences de données.

Contrairement aux tableaux, les slices sont typées uniquement par les éléments qu'elles contiennent (pas par le nombre d'éléments). Pour créer une slice vide de longueur non nulle, on utilise la méthode intégrée **make**. Ici, on crée une slice de string de longueur 3 (initialement de valeurs nulles).

```
func main() {

    s := make([]string, 3)
    fmt.Println("emp:", s)
```

On peut affecter et retrouver les valeurs comme avec les tableaux.

```
    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])
```

len renvoie la longueur de la slice comme prévu.

```
    fmt.Println("len:", len(s))
```

En plus de ces opérations basiques, les slices en supportent plusieurs qui les rendent plus riches que les tableaux. L'une d'elles est la méthode prédéfinie **append**, qui renvoie une slice avec une ou plusieurs nouvelles valeurs. À noter que la nouvelle slice est renvoyée par append : la slice originale, passée en paramètre, n'est pas modifiée.

```
    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)
```

On peut également copier les slices. Ici, on crée une slice c vide, de même longueur que s et on copie les valeurs de s dans c.

```
    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)
```

Les slices possèdent un opérateur « slice » avec la syntaxe slice[debut:fin]. Par exemple, l'exemple suivant crée une slice avec les éléments s[2], s[3] et s[4].

```
    l := s[2:5]
    fmt.Println("sl1:", l)
```

Ceci crée une slice jusqu'à (mais sans) s[5].

```
    l = s[:5]
    fmt.Println("sl2:", l)
```

Et ceci crée une slice à partir de (et incluant) s[2].

```
    l = s[2:]
    fmt.Println("sl3:", l)
```

On peut déclarer et initialiser une variable pour une slice en une ligne également. À la différence des tableaux, on ne précise pas la longueur dans [].

```
    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)
```

Les slices peuvent être composées en structures de données multidimensionnelles. La longueur des slices internes peut varier, contrairement aux tableaux multidimensionnels.

```
    twoD := make([][]int, 3)
```

```
    for i := 0; i < 3; i++ {
        innerLen := i + 1
        twoD[i] = make([]int, innerLen)
        for j := 0; j < innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

Code complet

```
package main

import "fmt"

func main() {

    s := make([]string, 3)
    fmt.Println("emp:", s)

    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])

    fmt.Println("len:", len(s))

    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)

    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)

    l := s[2:5]
    fmt.Println("sl1:", l)

    l = s[:5]
    fmt.Println("sl2:", l)

    l = s[2:]
    fmt.Println("sl3:", l)

    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)

    twoD := make([][]int, 3)
    for i := 0; i < 3; i++ {
        innerLen := i + 1
        twoD[i] = make([]int,
 innerLen)
        for j := 0; j <
 innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

Exécuter le code en ligne

À noter que, bien que les slices soient un type différent des tableaux, elles s'affichent de la même manière avec fmt.Println.

```
$ go run slices.go
emp: [  ]
set: [a b c]
get: c
len: 3
```

```
apd: [a b c d e f]
cpy: [a b c d e f]
sl1: [c d e]
sl2: [a b c d e]
sl3: [c d e f]
dcl: [g h i]
2d:  [[0] [1 2] [2 3 4]]
```

Vous pouvez lire ce **bon article** de l'équipe Go pour plus de détails sur le design et l'implémentation des slices en Go.

Maintenant que nous avons vu les tableaux et les slices, nous allons regarder une autre structure de données clé : les *maps*.

XI - Maps

Les *maps* sont un **type de données associatif** (parfois appelé *hashes* ou *dicts* dans d'autres langages).

Pour créer une map vide, on utilise la méthode intégrée **make** : **make(map[key-type]val-type)**.

```
    m := make(map[string]int)
```

On affecte les paires clé/valeur avec la syntaxe name[key] = val :

```
    m["k1"] = 7
    m["k2"] = 13
```

Afficher une map via Println affichera toutes les paires clé/valeur.

```
    fmt.Println("map:", m)
```

On récupère une valeur avec name[key].

```
    v1 := m["k1"]
    fmt.Println("v1: ", v1)
```

La méthode intégrée **len** renvoie le nombre de paires clé/valeur lorsqu'on l'appelle sur une map :

```
    fmt.Println("len:", len(m))
```

La méthode prédéfinie delete supprime d'une map une paire clé/valeur :

```
    delete(m, "k2")
    fmt.Println("map:", m)
```

La seconde valeur de retour optionnelle lorsque l'on récupère une valeur dans une map indique si la clé était présente. Cela peut être utilisé pour lever l'ambiguïté entre des clés manquantes et des clés de valeur nulle, comme 0 ou "". Ici, nous n'avions pas besoin de la valeur en elle-même, donc nous l'avions ignorée avec l'*identifieur vide* _.

```
    _, prs := m["k2"]
    fmt.Println("prs:", prs)
```

On peut aussi déclarer et initialiser une nouvelle map sur la même ligne avec cette syntaxe.

```
    n := map[string]int{"foo": 1, "bar": 2}
    fmt.Println("map:", n)
}
```

Code complet

```
package main

import "fmt"

func main() {

    m := make(map[string]int)

    m["k1"] = 7
    m["k2"] = 13

    fmt.Println("map:", m)

    v1 := m["k1"]
    fmt.Println("v1: ", v1)

    fmt.Println("len:", len(m))

    delete(m, "k2")
    fmt.Println("map:", m)

    _, prs := m["k2"]
    fmt.Println("prs:", prs)


 n := map[string]int{"foo": 1, "bar"
    fmt.Println("map:", n)
}
```

Exécuter le code en ligne

À noter que les maps apparaissent sous la forme `map[k:v k:v]` lorsqu'on les affiche avec `fmt.Println` :

```
$ go run maps.go
map: map[k1:7 k2:13]
v1:  7
len: 2
map: map[k1:7]
prs: false
map: map[foo:1 bar:2]
```

XII - Range

`range` itère sur les éléments de différents types de structures de données. Voyons comment utiliser **`range`** avec certaines structures de données que nous avons déjà rencontrées.

Ici on utilise **`range`** pour additionner les nombres d'une slice. Cela fonctionne de la même manière avec un tableau.

```
func main() {
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
```

`range` sur les tableaux et les slices fournit à la fois la clé et la valeur pour chaque entrée. Au-dessus, nous n'avions pas besoin de la clé, donc nous l'avions ignorée avec l'identifieur blanc `_`. On a cependant parfois besoin de la récupérer.

```
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
```

range sur une map itère sur les paires clé/valeur.

```
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
```

range sur une chaîne itère sur les points de code **Unicode**. La première valeur est l'index de l'octet de départ de la rune, et le second la rune elle-même.

```
    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

Code complet

```
package main

import "fmt"

func main() {

    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    for i, num := range nums {
        if num == 3 {

 fmt.Println("index:", i)
        }
    }


 kvs := map[string]string{"a": "appl
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n",
 k, v)
    }

    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

Exécuter le code en ligne

```
$ go run range.go
sum: 9
index: 1
a -> apple
b -> banana
0 103
1 111
```

XIII - Fonctions

Les *fonctions* sont centrales en Go. Nous allons les découvrir à travers quelques exemples différents.

Voici une fonction qui prend deux int en paramètres et renvoie leur somme, un int.

```
func plus(a int, b int) int {
```

Go a besoin de retours explicites : il ne renverra pas automatiquement la valeur de la dernière expression.

```
    return a + b
}
```

Quand vous avez plusieurs paramètres consécutifs du même type, vous pouvez vous passer des déclarations de type jusqu'au dernier, qui le déclare.

```
func plusPlus(a, b, c int) int {
    return a + b + c
}
```

On appelle une fonction comme on s'y attend, avec nom(arguments)

```
func main() {

    res := plus(1, 2)
    fmt.Println("1+2 =", res)
    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

Code complet

```
package main

import "fmt"

func plus(a int, b int) int {

    return a + b
}

func plusPlus(a, b, c int) int {
    return a + b + c
}

func main() {

    res := plus(1, 2)
    fmt.Println("1+2 =", res)

    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

Exécuter le code en ligne

```
$ go run functions.go
1+2 = 3
1+2+3 = 6
```

Les fonctions ont plusieurs autres fonctionnalités particulières. L'une d'entre elles, c'est les valeurs de retour multiples, que nous allons maintenant aborder.

XIV - Valeurs de retour multiples

Go supporte de base les *valeurs de retour multiples*. On s'en sert souvent en Go, par exemple pour renvoyer le résultat et la valeur d'erreur d'une fonction.

Le (int, int) dans cette signature de fonction montre que la fonction renvoie deux int.

```
func vals() (int, int) {
    return 3, 7
}
```

Les valeurs de retour peuvent également être nommées. Cela peut être utile pour aider à documenter le rôle de chaque valeur de retour. Ces noms peuvent être référencés dans le corps de la fonction, comme n'importe quelle autre variable. Un retour nu avec un simple **return** utilisera les valeurs de ces variables comme résultat.

```
func splitPrice(price float32) (dollars, cents int) {
    dollars = int(price)
    cents = int((price - float32(dollars)) * 100.0)
    return
}
```

Un cas d'usage, c'est pour modifier une valeur de retour à l'intérieur d'une clause **defer** :

```
func getFileSize() (file_size int64, had_error bool) {
    f, err := os.Open("/tmp/dat")
    if err != nil {
        return 0, true
    }
    defer func() {
        if err := f.Close(); err != nil {
            had_error = true
        }
    }()

    fi, err := f.Stat()
    if err != nil {
        return 0, true
    }
    return fi.Size(), false
}

func main() {
```

Ici nous utilisons les deux valeurs de retour différentes de l'appel avec une *affectation multiple*.

```
    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)
```

Si vous voulez seulement un sous-ensemble des valeurs de retour, vous pouvez utiliser l'identifieur blanc _.

```
    _, c := vals()
    fmt.Println(c)

    dollars, cents := splitPrice(12.42)
    fmt.Println(dollars, cents)
    file_size, had_error := getFileSize()
    fmt.Println(file_size, had_error)
}
```

Code complet

```
package main

import "fmt"

func vals() (int, int) {
    return 3, 7
}

func
 splitPrice(price float32) (dollars,
 cents int) {
    dollars = int(price)

 cents = int((price - float32(dollar
    return
}
```

Exécuter le code en ligne

Code complet

```
func
 getFileSize() (file_size int64,
 had_error bool) {
    f, err := os.Open("/tmp/
dat")
    if err != nil {
        return 0, true
    }
    defer func() {
        if err := f.Close();
 err != nil {
            had_error = true
        }
    }()

    fi, err := f.Stat()
    if err != nil {
        return 0, true
    }
    return fi.Size(), false
}

func main() {

    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    _, c := vals()
    fmt.Println(c)

    dollars, cents :=
 splitPrice(12.42)
    fmt.Println(dollars, cents)

    file_size, had_error :=
 getFileSize()
    fmt.Println(file_size,
 had_error)
}
```

```
$ go run multiple-return-values.go
3
7
7
12 42
0 true
```

Accepter un nombre variable d'arguments est une autre possibilité sympa des fonctions en Go. Nous allons regarder cela maintenant.

XV - Fonctions variadiques

Les **fonctions variadiques** peuvent être appelées avec n'importe quel nombre d'arguments. Par exemple `fmt.Println` est une fonction variadique courante.

Voici une fonction qui va prendre un nombre arbitraire d'`int` comme arguments.

```
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
```

```
    fmt.Println(total)
}

func main() {
```

Les fonctions variadiques peuvent être appelées de la manière habituelle, avec des arguments individuels.

```
    sum(1, 2)
    sum(1, 2, 3)
```

Si vous avez déjà plusieurs arguments dans une slice, vous pouvez les appliquer à la fonction variadique en utilisant **func**(slice...) comme ceci.

```
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

Code complet

```
package main

import "fmt"

func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {

    sum(1, 2)
    sum(1, 2, 3)

    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

Exécuter le code en ligne

```
$ go run variadic-functions.go
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

Un autre aspect clé des fonctions en Go, c'est la possibilité d'en faire des fermetures, ce que nous allons regarder ensuite.

XVI - Fermetures

Go supporte les **fonctions anonymes**, qui peuvent former une **closure** (fermeture ou clôture en français.) Les fonctions anonymes sont utiles lorsque l'on veut définir des fonctions à la volée, sans avoir à les nommer.

Cette fonction intSeq renvoie une autre fonction, qui est définie anonymement dans le corps de intSeq. La fonction retournée *embarque* la variable i pour former une clôture.

```
func intSeq() func() int {
    i := 0
    return func() int {
        i += 1
        return i
    }
}
```

```
func main() {
```

On appelle intSeq, et assigne le résultat (une fonction) à nextInt. Cette fonction capture sa propre valeur de i, qui sera mise à jour à chaque fois que l'on appellera nextInt.

```
    nextInt := intSeq()
```

Voyez l'effet de la fermeture en appelant nextInt plusieurs fois.

```
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())
```

Pour confirmer que l'état est unique à cette fonction particulière, créez et testez-en une nouvelle.

```
    newInts := intSeq()
    fmt.Println(newInts())
}
```

Code complet

```
package main

import "fmt"

func intSeq() func() int {
    i := 0
    return func() int {
        i += 1
        return i
    }
}

func main() {

    nextInt := intSeq()

    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    newInts := intSeq()
    fmt.Println(newInts())
}
```

Exécuter le code en ligne

```
$ go run closures.go
1
2
3
1
```

La dernière fonctionnalité des fonctions que nous allons regarder pour le moment, c'est la récursivité.

XVII - Récursivité

Go supporte la **récursivité**. Voici l'exemple classique de la factorielle.

Cette fonction fact s'appelle elle-même jusqu'à ce qu'elle atteigne le cas de base, fact(0).

```
func fact(n int) int {
    if n == 0 {
        return 1
```

```
    }
    return n * fact(n-1)
}
func main() {
    fmt.Println(fact(7))
}
```

Code complet

```
package main

import "fmt"

func fact(n int) int {
    if n == 0 {
        return 1
    }
    return n * fact(n-1)
}

func main() {
    fmt.Println(fact(7))
}
```

Exécuter le code en ligne

```
$ go run recursion.go
5040
```

XVIII - Pointeurs

Go supporte les **pointeurs**, ce qui permet de passer des références vers des valeurs dans les programmes.

Nous allons voir comment fonctionnent les pointeurs, par opposition aux valeurs, dans deux fonctions : zeroval et zeroptr. zeroval a un paramètre, donc les arguments lui sont passés par valeur. zeroval aura une copie de ival distincte de celle de la fonction appelante.

```
func zeroval(ival int) {
    ival = 0
}
```

zeroptr a cette fois-ci un paramètre *int, c'est-à-dire un pointeur sur un int. Le *iptr dans le corps de la fonction permet de *déréférencer* le pointeur de son adresse mémoire pour obtenir la valeur à cette adresse. Assigner une valeur à un pointeur déréférencé change la valeur à l'adresse référencée.

```
func zeroptr(iptr *int) {
    *iptr = 0
}
func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)
```

La syntaxe &i donne l'adresse mémoire de i, c'est-à-dire un pointeur vers i.

```
    zeroptr(&i)
    fmt.Println("zeroptr:", i)
```

On peut afficher les pointeurs également.

```
    fmt.Println("pointer:", &i)
}
```

Code complet

```go
package main

import "fmt"

func zeroval(ival int) {
    ival = 0
}

func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    zeroptr(&i)
    fmt.Println("zeroptr:", i)

    fmt.Println("pointer:", &i)
}
```

Exécuter le code en ligne

zeroval ne change pas le i dans main, mais zeroptr le fait, car elle a une référence vers l'adresse mémoire de cette variable.

```
$ go run pointers.go
initial: 1
zeroval: 1
zeroptr: 0
pointer: 0x42131100
```

XIX - Structures

Les *structures* en Go sont des collections de champs typés. Elles sont utiles pour regrouper les données.

Cette structure person a des champs name et age.

```go
type person struct {
    name string
    age  int
}

func main() {
```

Cette syntaxe crée une nouvelle structure.

```go
    fmt.Println(person{"Bob", 20})
```

On peut nommer les champs lorsqu'on initialise une structure.

```go
    fmt.Println(person{name: "Alice", age: 30})
```

Les champs non spécifiés seront de valeur nulle.

```go
    fmt.Println(person{name: "Fred"})
```

Un préfixe & fournit un pointeur sur la structure.

```
    fmt.Println(&person{name: "Ann", age: 40})
```

On accède aux champs de la structure avec un point.

```
    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)
```

On peut aussi utiliser le point avec des pointeurs de structure : les pointeurs sont automatiquement déréférencés.

```
    sp := &s
    fmt.Println(sp.age)
```

Les structures sont mutables.

```
    sp.age = 51
    fmt.Println(sp.age)
}
```

Code complet

```
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {


 fmt.Println(person{"Bob", 20})


 fmt.Println(person{name: "Alice",
 age: 30})


 fmt.Println(person{name: "Fred"})


 fmt.Println(&person{name: "Ann",
 age: 40})

    s := person{name: "Sean",
 age: 50}
    fmt.Println(s.name)

    sp := &s
    fmt.Println(sp.age)

    sp.age = 51
    fmt.Println(sp.age)
}
```

Exécuter le code en ligne

```
$ go run structs.go
{Bob 20}
{Alice 30}
{Fred 0}
&{Ann 40}
Sean
50
51
```

XX - Méthodes

Go supporte des méthodes définies dans les structures.

Cette méthode area a un *type receveur* *rect.

```
func (r *rect) area() int {
    return r.width * r.height
}
```

Les méthodes peuvent être définies soit pour un pointeur, soit pour une valeur de type receveur. Voici un exemple avec une valeur.

```
func (r rect) perim() int {
    return 2*r.width + 2*r.height
}

func main() {
    r := rect{width: 10, height: 5}
```

Ici on appelle les deux méthodes définies dans notre structure.

```
    fmt.Println("area: ", r.area())
    fmt.Println("perim:", r.perim())
```

Go gère automatiquement la conversion entre valeur et pointeur pour les appels de méthode. On peut vouloir utiliser un type receveur pointeur pour éviter la copie lors des appels de méthode ou pour modifier l'objet fourni.

```
    rp := &r
    fmt.Println("area: ", rp.area())
    fmt.Println("perim:", rp.perim())
}
```

Code complet

```
package main

import "fmt"

type rect struct {
    width, height int
}

func (r *rect) area() int {
    return r.width * r.height
}

func (r rect) perim() int {

  return 2*r.width + 2*r.height
}

func main() {
    r := rect{width: 10,
 height: 5}

    fmt.Println("area: ",
 r.area())
    fmt.Println("perim:",
 r.perim())

    rp := &r
    fmt.Println("area: ",
 rp.area())
```

Exécuter le code en ligne

Code complet

```
        fmt.Println("perim:",
    rp.perim())
}
```

```
$ go run methods.go
area:  50
perim: 30
area:  50
perim: 30
```

Nous allons ensuite aborder les mécanismes pour regrouper et nommer les ensembles de méthodes liés : les interfaces.

XXI - Interfaces

Les interfaces sont des collections nommées de signatures de méthodes.

Voici une interface simple pour une forme géométrique.

```
type geometry interface {
    area() float64
    perim() float64
}
```

Pour notre exemple, nous allons implémenter cette interface sur les types rect et circle.

```
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}
```

Pour implémenter une interface en Go, il suffit d'en implémenter toutes les méthodes. Ici, nous implémentons geometry chez rect.

```
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}
```

Voici maintenant l'implémentation pour les circle.

```
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

Si une variable a une interface, alors on peut appeler les méthodes de l'interface. Voici une fonction générique qui utilise ceci pour travailler avec n'importe quelle geometry.

```
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}
```

```
func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}
```

Mes structures circle et rect implémentent toutes deux les interfaces de geometry, on peut donc utiliser des instances de ces structures en argument de measure.

```
    measure(r)
    measure(c)
}
```

Code complet

```
package main

import "fmt"
import "math"

type geometry interface {
    area() float64
    perim() float64
}

type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}

func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {

  return 2*r.width + 2*r.height
}

func (c circle) area() float64 {

    return math.Pi * c.radius *
 c.radius
}
func (c circle)
 perim() float64 {
    return 2 * math.Pi *
 c.radius
}

func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{width: 3,
 height: 4}
    c := circle{radius: 5}

    measure(r)
    measure(c)
}
```

Exécuter le code en ligne

```
$ go run interfaces.go
{3 4}
12
14
{5}
78.53981633974483
```

```
31.41592653589793
```

Pour en apprendre plus sur les interfaces en Go, vous pouvez lire ce **bon article**.

XXII - Erreurs

En Go, il est idiomatique de communiquer les erreurs via une valeur de retour explicite et séparée. Cela contraste avec les exceptions utilisées dans des langages comme Java ou Ruby, et ce qu'on retrouve en C, où l'on renvoie une valeur parfaite, et parfois un code d'erreur. L'approche en Go fait qu'il est facile de voir quelles fonctions renvoient un code d'erreur et de les gérer en utilisant les mêmes constructions du langage utilisées pour n'importe quelle autre tâche sans erreur.

Par convention, les erreurs sont la dernière valeur de retour et ont pour type error, une interface du langage.

```
func f1(arg int) (int, error) {
    if arg == 42 {
```

errors.New construit une valeur d'erreur basique avec le message d'erreur donné.

```
        return -1, errors.New("can't work with 42")

    }
```

La valeur nil pour l'erreur indique qu'il n'y a pas eu d'erreur.

```
    return arg + 3, nil
}
```

Il est possible d'utiliser des types d'erreurs sur mesure en implémentant la méthode Error() sur ces types. Voici une variante de l'exemple ci-dessus qui utilise un type sur mesure pour représenter explicitement un argument d'erreur.

```
type argError struct {
    arg  int
    prob string
}
func (e *argError) Error() string {
    return fmt.Sprintf("%d - %s", e.arg, e.prob)
}
```

Dans ce cas, nous utilisons la syntaxe &argError pour construire l'objet et fournir les valeurs des deux champs arg et prob.

```
func f2(arg int) (int, error) {
    if arg == 42 {
        return -1, &argError{arg, "can't work with it"}
    }
    return arg + 3, nil
}
```

Les deux boucles ci-dessous testent chacune de nos fonctions qui renvoient des erreurs. Notez que l'utilisation d'un test d'erreur inline dans le if est idiomatique en code Go.

```
func main() {
    for _, i := range []int{7, 42} {
        if r, e := f1(i); e != nil {
            fmt.Println("f1 failed:", e)
        } else {
            fmt.Println("f1 worked:", r)
        }
```

```
    }
    for _, i := range []int{7, 42} {
        if r, e := f2(i); e != nil {
            fmt.Println("f2 failed:", e)
        } else {
            fmt.Println("f2 worked:", r)
        }
    }
```

Si vous voulez utiliser les données d'une erreur personnalisée, vous devrez récupérer l'erreur comme une instance du type d'erreur sur mesure en explicitant le type à utiliser.

```
    _, e := f2(42)
    if ae, ok := e.(*argError); ok {
        fmt.Println(ae.arg)
        fmt.Println(ae.prob)
    }
}
```

Code complet

```
package main

import "errors"
import "fmt"

func f1(arg int) (int, error) {
    if arg == 42 {

        return -1,
 errors.New("can't work with
 42")

    }

    return arg + 3, nil
}

type argError struct {
    arg  int
    prob string
}

func (e *argError)
 Error() string {
    return fmt.Sprintf("%d -
 %s", e.arg, e.prob)
}

func f2(arg int) (int, error) {
    if arg == 42 {


  return -1, &argError{arg, "can't
 work with it"}
    }
    return arg + 3, nil
}

func main() {

    for _,
 i := range []int{7, 42} {
        if r, e := f1(i); e !
= nil {
            fmt.Println("f1
 failed:", e)
        } else {
            fmt.Println("f1
 worked:", r)
```

Exécuter le code en ligne

Code complet

```
            }
        }
        for _,
    i := range []int{7, 42} {
            if r, e := f2(i); e !
    = nil {
                fmt.Println("f2
     failed:", e)
            } else {
                fmt.Println("f2
     worked:", r)
            }
        }

        _, e := f2(42)
        if ae, ok := e.(*argError);
     ok {
            fmt.Println(ae.arg)
            fmt.Println(ae.prob)
        }
    }
```

```
$ go run errors.go
f1 worked: 10
f1 failed: can't work with 42
f2 worked: 10
f2 failed: 42 - can't work with it
42
can't work with it
```

Regardez ce **bon article** sur le blog de Go pour plus d'informations sur la gestion d'erreurs.

XXIII - Goroutines

Une goroutine est un thread d'exécution léger.

Supposons que l'on ait un appel de fonction f(s). Normalement, on l'appellerait comme ceci, et son exécution serait synchrone.

```
    f("direct")
```

Pour appeler la fonction dans une goroutine, on utilise go f(s). Cette nouvelle goroutine va s'exécuter de manière concurrente avec la fonction appelée.

```
    go f("goroutine")
```

On peut aussi démarrer une goroutine pour un appel de fonction anonyme.

```
    go func(msg string) {
        fmt.Println(msg)
    }("going")
```

Nos deux appels de fonction s'exécutent de manière asynchrone dans des goroutines séparées, donc l'exécution échoue ici. Ce Scanln demande que l'on presse une touche pour sortir du programme.

```
    var input string
    fmt.Scanln(&input)
    fmt.Println("done")
}
```

Code complet

```
package main

import "fmt"

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":",
 i)
    }
}

func main() {

    f("direct")

    go f("goroutine")

    go func(msg string) {
        fmt.Println(msg)
    }("going")

    var input string
    fmt.Scanln(&input)
    fmt.Println("done")
}
```

Exécuter le code en ligne

Lorsque l'on lance ce programme, on voit la sortie de la fonction bloquante en premier, puis les exécutions croisées des deux goroutines. Cet entrelacement reflète le fait que les deux goroutines sont exécutées de manière concurrente.

```
$ go run goroutines.go
direct : 0
direct : 1
direct : 2
goroutine : 0
going
goroutine : 1
goroutine : 2
<enter>
done
```

Ensuite nous allons regarder un complément aux goroutines : les canaux.

XXIV - Canaux

Les canaux (channels) sont des tuyaux qui relient des goroutines concurrentes. On peut envoyer des valeurs d'une goroutine à une autre via un canal.

On crée un nouveau canal avec **make(chan val-type)**. Les canaux sont typés avec les valeurs qu'ils transportent.

```
    messages := make(chan string)
```

On envoie une valeur dans un canal avec la syntaxe canal <-. Ici on envoie "ping" dans le canal messages que l'on a créé précédemment, depuis une nouvelle goroutine.

```
    go func() { messages <- "ping" }()
```

La syntaxe <-canal reçoit une valeur depuis un canal. Ici on affiche le message "ping" envoyé dans le précédent code.

```
    msg := <-messages
    fmt.Println(msg)
}
```

Code complet

```
package main

import "fmt"

func main() {


 messages := make(chan string)

    go func() {
 messages <- "ping" }()

    msg := <-messages
    fmt.Println(msg)
}
```

Exécuter le code en ligne

Lorsque l'on lance le programme, le message **"ping"** est correctement passé d'une goroutine à l'autre via notre canal.

```
$ go run channels.go
ping
```

Par défaut, l'envoi et la réception de messages sont bloqués jusqu'à ce que l'envoyeur et le receveur soient prêts. Cette propriété nous a permis d'attendre le message **"ping"** à la fin de notre programme, sans utiliser une autre forme de synchronisation.

XXV - Canaux avec buffer

Par défaut les canaux n'ont pas de buffer, ce qui signifie qu'ils acceptent uniquement les envois (**chan <-**) s'il y a un receveur correspondant (**<- chan**) prêt à recevoir la valeur envoyée. Les *canaux avec buffer* acceptent un nombre limité de valeurs sans receveur correspondant pour ces valeurs.

Ici on crée avec **make** un canal avec buffer, qui accumule jusqu'à deux chaînes de caractères.

```
func main() {

    messages := make(chan string, 2)
```

Comme ce canal a un buffer, nous pouvons envoyer ces deux valeurs dans le canal sans une réception concurrente correspondante.

```
    messages <- "buffered"
    messages <- "channel"
```

Ensuite, on peut recevoir ces deux valeurs comme d'habitude.

```
    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

Code complet

```
package main

import "fmt"

func main() {


 messages := make(chan string, 2)

    messages <- "buffered"
```

Exécuter le code en ligne

Code complet

```
    messages <- "channel"

    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

```
$ go run channel-buffering.go
buffered
channel
```

XXVI - Synchronisation des canaux

On peut utiliser les canaux pour synchroniser l'exécution à travers des goroutines. Voici un exemple qui utilise une réception bloquante pour attendre qu'une goroutine se termine.

Voici la fonction qui va tourner dans une goroutine. Le canal done sera utilisé pour notifier à une autre goroutine que le travail de cette fonction est terminé.

```
func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second)
    fmt.Println("done")
```

on envoie une valeur pour notifier qu'on a terminé.

```
    done <- true
}
```

On démarre la goroutine, en lui fournissant le canal à notifier.

```
func main() {

    done := make(chan bool, 1)
    go worker(done)
```

On bloque jusqu'à ce qu'on ait reçu une notification sur ce canal.

```
    <-done
}
```

Code complet

```
package main

import "fmt"
import "time"

func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second)
    fmt.Println("done")

    done <- true
}

func main() {

    done := make(chan bool, 1)
    go worker(done)

    <-done
}
```

Exécuter le code en ligne

```
$ go run channel-synchronization.go
working...done
```

Si vous enlevez la ligne `<- done`, le programme se terminera avant que `worker` ait même commencé.

XXVII - Direction des canaux

Lorsqu'on utilise des canaux comme paramètres de fonctions, on peut spécifier si un canal est censé uniquement envoyer ou recevoir des valeurs. Cette spécificité augmente la sécurité du typage du programme.

Cette fonction `ping` accepte seulement un canal qui envoie des valeurs. On aurait une erreur de compilation si on essayait de recevoir sur ce canal.

```
func ping(pings chan<- string, msg string) {
    pings <- msg
}
```

La fonction `pong` accepte un canal pour recevoir (`pings`) et un second pour les envois (`pongs`).

```
func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

Code complet

```
package main

import "fmt"

func ping(pings chan<- string,
 msg string) {
    pings <- msg
}

func pong(pings <-chan string,
 pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}

func main() {

 pings := make(chan string, 1)

 pongs := make(chan string, 1)
    ping(pings, "passed
 message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

Exécuter le code en ligne

```
$ go run channel-directions.go
passed message
```

XXVIII - Select

Le select de go permet d'attendre plusieurs opérations sur les canaux. Combiner les goroutines et les canaux est une fonctionnalité puissante de Go.

Pour notre exemple, nous ferons des select à travers deux canaux.

```
func main() {

    c1 := make(chan string)
    c2 := make(chan string)
```

Chaque canal va recevoir une valeur après un certain temps, pour simuler une opération bloquante (par ex. un appel RPC) qui s'exécute dans une goroutine concurrente.

```
    go func() {
        time.Sleep(time.Second * 1)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "two"
    }()
```

Nous utilisons **select** pour attendre ces deux valeurs simultanément, en affichant chacune d'elles dès son arrivée.

```
    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

Code complet

```
package main

import "time"
import "fmt"

func main() {

    c1 := make(chan string)
    c2 := make(chan string)

    go func() {

 time.Sleep(time.Second * 1)
        c1 <- "one"
    }()
    go func() {

 time.Sleep(time.Second * 2)
        c2 <- "two"
    }()

    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:

 fmt.Println("received", msg1)
        case msg2 := <-c2:
```

Exécuter le code en ligne

Code complet

```
    fmt.Println("received", msg2)
            }
        }
    }
```

Nous recevons les valeurs "one" puis "two" comme attendu.

```
$ time go run select.go
received one
received two
```

À noter que le temps total d'exécution est seulement de ~2 secondes, car les deux Sleeps s'exécutent de manière concurrente.

```
real    0m2.245s
```

XXIX - Timeouts

Les timemouts (dépassement du temps d'exécution alloué) sont importants pour des programmes qui se connectent à des ressources externes ou qui doivent limiter leur temps d'exécution. Implémenter des timeout en Go est facile et élégant grâce aux canaux et à **select**.

Pour notre exemple, supposons que l'on exécute un appel externe qui renvoie son résultat sur le canal c1 après 2 s.

```
    c1 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        c1 <- "result 1"
    }()
```

Voici le **select** qui implémente le timeout. res := <-c1 attend le résultat et <-Time.After attend qu'une valeur soit envoyée après 1 s. Comme **select** traite la première réception, nous prenons le cas du timeout si l'opération prend plus de temps qu'il ne lui en est alloué.

```
    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println("timeout 1")
    }
```

Si nous permettons un plus grand timeout de 3 s, alors la réception depuis c2 va réussir et nous afficherons le résultat.

```
    c2 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "result 2"
    }()
    select {
    case res := <-c2:
        fmt.Println(res)
    case <-time.After(time.Second * 3):
        fmt.Println("timeout 2")
    }
}
```

Code complet

```
package main
```

Exécuter le code en ligne

Code complet

```
import "time"
import "fmt"

func main() {

    c1 := make(chan string, 1)
    go func() {

 time.Sleep(time.Second * 2)
        c1 <- "result 1"
    }()

    select {
    case res := <-c1:
        fmt.Println(res)

  case <-time.After(time.Second * 1)
        fmt.Println("timeout 1")
    }

    c2 := make(chan string, 1)
    go func() {

 time.Sleep(time.Second * 2)
        c2 <- "result 2"
    }()
    select {
    case res := <-c2:
        fmt.Println(res)

  case <-time.After(time.Second * 3)
        fmt.Println("timeout 2")
    }
}
```

Ce programme montre que la première opération, trop longue, est arrêtée et que la seconde réussit.

```
$ go run timeouts.go
timeout 1
result 2
```

Utiliser ce modèle de timeout avec **select** nécessite de communiquer les résultats à travers les canaux. C'est une bonne idée en général, car d'autres fonctionnalités importantes de Go sont basées sur les canaux et sur **select**. Nous allons aborder par la suite les timers et tickers.

XXX - Opérations non bloquantes sur les canaux

Les envois et réceptions basiques sur les canaux sont bloquants. Cependant, on peut utiliser **select** avec une clause **default** pour implémenter des envois et réceptions *non bloquants*, et même des **select** non bloquants sur plusieurs canaux.

Voici une réception non bloquante. Si une valeur est disponible sur messages, alors **select** va prendre le cas <-messages avec cette valeur. Sinon, il prendra immédiatement le cas **default**.

```
    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    default:
        fmt.Println("no message received")
    }
```

Un envoi non bloquant marche de manière similaire.

```
    msg := "hi"
    select {
    case messages <- msg:
        fmt.Println("sent message", msg)
    default:
        fmt.Println("no message sent")
    }
```

On peut utiliser plusieurs **case** au-dessus de la clause **default** pour implémenter un **select** non bloquant multiple. Ici, on essaie de recevoir des messages de manière non bloquante sur les canaux messages et signals simultanément.

```
    select {
    case msg := <-messages:
        fmt.Println("received message", msg)
    case sig := <-signals:
        fmt.Println("received signal", sig)
    default:
        fmt.Println("no activity")
    }
}
```

Code complet

```
package main

import "fmt"

func main() {

 messages := make(chan string)
    signals := make(chan bool)

    select {
    case msg := <-messages:
        fmt.Println("received
 message", msg)
    default:
        fmt.Println("no message
 received")
    }

    msg := "hi"
    select {
    case messages <- msg:
        fmt.Println("sent
 message", msg)
    default:
        fmt.Println("no message
 sent")
    }

    select {
    case msg := <-messages:
        fmt.Println("received
 message", msg)
    case sig := <-signals:
        fmt.Println("received
 signal", sig)
    default:
        fmt.Println("no
 activity")
    }
}
```

Exécuter le code en ligne

```
$ go run non-blocking-channel-operations.go
no message received
no message sent
no activity
```

XXXI - Fermer des canaux

Fermer un canal indique que nous n'enverrons plus de valeurs dessus. Cela peut être utile pour indiquer au receveur qu'on a terminé.

Dans cet exemple, nous allons utiliser un canal jobs pour communiquer le travail à faire de main() à une autre goroutine. Quand nous n'aurons plus de travail à envoyer, nous fermerons le canal jobs avec **close**.

```
func main() {
    jobs := make(chan int, 5)
    done := make(chan bool)
```

Voici la goroutine qui traite les jobs. Elle reçoit de manière répétée dans j des valeurs de jobsavec j, more := <-jobs. Dans cette forme particulière de réception à deux valeurs, la valeur de more sera **false** si jobs a été fermée et que toutes les valeurs ont déjà été reçues. Nous utilisons cela pour notifier au canal done lorsque nous avons traité tous les jobs.

```
    go func() {
        for {
            j, more := <-jobs
            if more {
                fmt.Println("received job", j)
            } else {
                fmt.Println("received all jobs")
                done <- true
                return
            }
        }
    }()
```

Cela envoie trois jobs à travers le canal jobs, puis le ferme.

```
    for j := 1; j <= 3; j++ {
        jobs <- j
        fmt.Println("sent job", j)
    }
    close(jobs)
    fmt.Println("sent all jobs")
```

On attend le worker en utilisant l'approche de **synchronisation** vue plus tôt.

```
    <-done
}
```

Code complet

```
package main

import "fmt"

func main() {
    jobs := make(chan int, 5)
    done := make(chan bool)

    go func() {
        for {
            j, more := <-jobs
            if more {

 fmt.Println("received job", j)
            } else {

 fmt.Println("received all
 jobs")
```

Exécuter le code en ligne

Code complet

```
                done <- true
                return
            }
        }
    }()

    for j := 1; j <= 3; j++ {
        jobs <- j
        fmt.Println("sent job",
 j)
    }
    close(jobs)
    fmt.Println("sent all jobs")

    <-done
}
```

```
$ go run closing-channels.go
sent job 1
received job 1
sent job 2
received job 2
sent job 3
received job 3
sent all jobs
received all jobs
```

L'idée des canaux fermés amène naturellement à notre exemple suivant : utiliser **range** sur des canaux.

XXXII - Range sur des canaux

Dans un exemple **précédent**, nous avons vu comment **for** et **range** permettent d'itérer sur des structures de données simples. On peut également utiliser cette syntaxe pour itérer sur des valeurs reçues depuis un canal.

Nous allons itérer sur deux valeurs dans le canal `queue`.

```
    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)
```

Ce **range** itère sur chaque élément à mesure qu'il est reçu dans `queue`. Comme nous avons fermé le canal plus haut, l'itération se termine après réception des deux éléments. Si nous ne l'avions pas fermé, nous bloquerions sur la réception d'une troisième valeur dans la boucle.

```
    for elem := range queue {
        fmt.Println(elem)
    }
}
```

Code complet

```
package main

import "fmt"

func main() {


 queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)
```

Exécuter le code en ligne

Code complet
```
    for elem := range queue {
        fmt.Println(elem)
    }
}
```

```
$ go run range-over-channels.go
one
two
```

Cet exemple montre également comment il est possible de fermer un canal non vide tout en continuant à recevoir les valeurs restantes.

XXXIII - Timers

On veut souvent exécuter du code Go à un certain point du futur, ou de manière répétée selon un intervalle de temps. Les fonctionnalités intégrées *timer* et *ticker* rendent ces deux tâches faciles. Nous allons d'abord aborder les timers, et ensuite les **tickers**.

Les timers représentent un événement unique dans le futur. On précise combien de temps on veut attendre, et il fournit un canal au quel sera notifié à ce moment-là. Ce timer attendra 2 secondes.

```
    timer1 := time.NewTimer(time.Second * 2)
```

Le `<-timer1.C` bloque sur le canal `C` du timer jusqu'à ce qu'il envoie une valeur indiquant que le timer a expiré.

```
    <-timer1.C
    fmt.Println("Timer 1 expired")
```

Pour simplement attendre, on peut utiliser `time.Sleep`. Mais les timers peuvent être utiles, car on peut les stopper avant qu'ils expirent. Voici un exemple :

```
    timer2 := time.NewTimer(time.Second)
    go func() {
        <-timer2.C
        fmt.Println("Timer 2 expired")
    }()
    stop2 := timer2.Stop()
    if stop2 {
        fmt.Println("Timer 2 stopped")
    }
}
```

Code complet
```
package main

import "time"
import "fmt"

func main() {

    timer1 :=
 time.NewTimer(time.Second * 2)

    <-timer1.C
    fmt.Println("Timer 1
 expired")

    timer2 :=
 time.NewTimer(time.Second)
    go func() {
        <-timer2.C
```

Exécuter le code en ligne

Code complet

```
        fmt.Println("Timer 2
  expired")
    }()
    stop2 := timer2.Stop()
    if stop2 {
        fmt.Println("Timer 2
  stopped")
    }
}
```

Le premier timer va expirer ~2 s après que l'on démarre le programme, mais le second devrait être stoppé avant qu'il ait pu expirer.

```
$ go run timers.go
Timer 1 expired
Timer 2 stopped
```

XXXIV - Tickers

Les **timers** servent quand on veut faire quelque chose dans le futur. Les *tickers* servent à faire quelque chose de manière répétée à intervalles réguliers. Voici un exemple d'un ticker qui se répète jusqu'à ce qu'on l'arrête.

Les tickers utilisent un mécanisme similaire à celui des timers : un canal pour envoyer des valeurs. Ici, nous utilisons la méthode intégrée **range** sur le canal pour itérer sur les valeurs lorsqu'elles arrivent, toutes les 500 ms.

```
    ticker := time.NewTicker(time.Millisecond * 500)
    go func() {
        for t := range ticker.C {
            fmt.Println("Tick at", t)
        }
    }()
```

Les tickers peuvent être arrêtés comme les timers. Lorsqu'un ticker est arrêté, il ne recevra plus de valeurs sur son canal. Nous arrêterons le nôtre après 1600 ms.

```
    time.Sleep(time.Millisecond * 1600)
    ticker.Stop()
    fmt.Println("Ticker stopped")
}
```

Code complet

Exécuter le code en ligne

```
package main

import "time"
import "fmt"

func main() {

    ticker :=
 time.NewTicker(time.Millisecond * 5
    go func() {
        for t := range
 ticker.C {
            fmt.Println("Tick
 at", t)
        }
    }()


 time.Sleep(time.Millisecond * 1600)
    ticker.Stop()
    fmt.Println("Ticker
 stopped")
```

Code complet
```
}
```

Quand on lance ce programme, le ticker devrait envoyer trois notifications avant qu'on ne l'arrête.

```
$ go run tickers.go
Tick at 2012-09-23 11:29:56.487625 -0700 PDT
Tick at 2012-09-23 11:29:56.988063 -0700 PDT
Tick at 2012-09-23 11:29:57.488076 -0700 PDT
Ticker stopped
```

XXXV - Worker Pools

Dans cet exemple, nous allons regarder comment implémenter un worker pool en utilisant des goroutines et des canaux.

Voici notre worker, dont nous allons lancer plusieurs instances concurrentes. Ces workers recevront du travail dans le canal `jobs` et enverront le résultat correspondant dans `results`. Nous ferons une pause d'une seconde par tâche pour simuler une opération complexe.

```
func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "processing job", j)
        time.Sleep(time.Second)
        results <- j * 2
    }
}

func main() {
```

Afin d'utiliser notre pool de workers, nous devons leur envoyer du travail et collecter leurs résultats. On crée deux canaux pour cela.

```
    jobs := make(chan int, 100)
    results := make(chan int, 100)
```

On démarre trois workers, initialement bloqués, car ils n'ont pas de jobs pour le moment.

```
    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }
```

Ici on envoie neuf travaux, puis fermons ce canal pour indiquer que c'est tout le travail que nous avons.

```
    for j := 1; j <= 9; j++ {
        jobs <- j
    }
    close(jobs)
```

Finalement, nous collectons les résultats du travail.

```
    for a := 1; a <= 9; a++ {
        <-results
    }
}
```

Code complet
```
package main

import "fmt"
```

Exécuter le code en ligne

Code complet

```
import "time"

func worker(id int,
 jobs <-chan int,
 results chan<- int) {
    for j := range jobs {
        fmt.Println("worker",
 id, "processing job", j)
        time.Sleep(time.Second)
        results <- j * 2
    }
}

func main() {

    jobs := make(chan int, 100)

 results := make(chan int, 100)

    for w := 1; w <= 3; w++ {
        go worker(w, jobs,
 results)
    }

    for j := 1; j <= 9; j++ {
        jobs <- j
    }
    close(jobs)

    for a := 1; a <= 9; a++ {
        <-results
    }
}
```

Notre programme montre, au cours de son exécution, les neuf jobs qui s'exécutent, traités par les différents workers. Le programme prend seulement environ 3 secondes à s'exécuter, bien qu'il traite environ 9 secondes de travail, car il y a trois workers qui travaillent de manière concurrente.

```
$ time go run worker-pools.go
worker 1 processing job 1
worker 2 processing job 2
worker 3 processing job 3
worker 1 processing job 4
worker 2 processing job 5
worker 3 processing job 6
worker 1 processing job 7
worker 2 processing job 8
worker 3 processing job 9

real    0m3.149s
```

XXXVI - Limitation de débit

Limiter le débit est un important mécanisme pour contrôler l'utilisation des ressources et maintenir la qualité de service. Go supporte élégamment la limitation de débit avec les goroutines, les canaux et les **tickers**.

Nous allons d'abord regarder une limitation de débit simple. Supposons que l'on veuille limiter notre gestion des requêtes entrantes. Nous allons servir ces requêtes depuis un canal du même nom.

```
    requests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        requests <- i
    }
    close(requests)
```

Ce canal limiter va recevoir une valeur toutes les 200 millisecondes. C'est le régulateur de notre système de limitation du débit.

```
    limiter := time.Tick(time.Millisecond * 200)
```

En bloquant sur une réception du canal limiter avant de servir chaque requête, on se limite à une requête toutes les 200 ms.

```
    for req := range requests {
        <-limiter
        fmt.Println("request", req, time.Now())
    }
```

On pourrait vouloir permettre de courtes augmentations du nombre de requêtes, tout en préservant la limitation globale. On peut pour cela mettre un buffer sur le canal limitant. Ce canal burstyLimiter va permettre de recevoir jusqu'à trois événements.

```
    burstyLimiter := make(chan time.Time, 3)
```

On remplit le canal pour représenter l'augmentation considérée.

```
    for i := 0; i < 3; i++ {
        burstyLimiter <- time.Now()
    }
```

Toutes les 200 ms nous essaierons d'ajouter une nouvelle valeur dans burstyLimiter, jusqu'à sa limite de trois.

```
    go func() {
        for t := range time.Tick(time.Millisecond * 200) {
            burstyLimiter <- t
        }
    }()
```

Maintenant on simule cinq nouvelles requêtes entrantes. Les trois premières bénéficieront des possibilités de sur-traitement de burstyLimiter.

```
    burstyRequests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        burstyRequests <- i
    }
    close(burstyRequests)
    for req := range burstyRequests {
        <-burstyLimiter
        fmt.Println("request", req, time.Now())
    }
}
```

Code complet

```
package main

import "time"
import "fmt"

func main() {


 requests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        requests <- i
    }
    close(requests)
```

Exécuter le code en ligne

Code complet

```
    limiter :=
time.Tick(time.Millisecond * 200)

    for req := range requests {
        <-limiter
        fmt.Println("request",
req, time.Now())
    }

    burstyLimiter := make(chan
time.Time, 3)

    for i := 0; i < 3; i++ {
        burstyLimiter <-
time.Now()
    }

    go func() {
        for t := range
time.Tick(time.Millisecond * 200) {
            burstyLimiter <- t
        }
    }()


burstyRequests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        burstyRequests <- i
    }
    close(burstyRequests)
    for req := range
burstyRequests {
        <-burstyLimiter
        fmt.Println("request",
req, time.Now())
    }
}
```

À l'exécution du programme, on voit que la première série de requêtes est traitée toutes les 200 ms, comme désiré.

```
$ go run rate-limiting.go
request 1 2012-10-19 00:38:18.687438 +0000 UTC
request 2 2012-10-19 00:38:18.887471 +0000 UTC
request 3 2012-10-19 00:38:19.087238 +0000 UTC
request 4 2012-10-19 00:38:19.287338 +0000 UTC
request 5 2012-10-19 00:38:19.487331 +0000 UTC
```

Dans la seconde série, les trois premières requêtes sont servies immédiatement à cause de l'augmentation autorisée, puis on sert les deux suivantes avec un délai de ~200 ms chacune.

```
request 1 2012-10-19 00:38:20.487578 +0000 UTC
request 2 2012-10-19 00:38:20.487645 +0000 UTC
request 3 2012-10-19 00:38:20.487676 +0000 UTC
request 4 2012-10-19 00:38:20.687483 +0000 UTC
request 5 2012-10-19 00:38:20.887542 +0000 UTC
```

XXXVII - Compteurs atomiques

Le mécanisme principal pour gérer les états en Go, c'est la communication à travers les canaux. Nous avons vu ceci par exemple avec les **worker pools**. Il y a quelques autres options pour gérer les états ceci dit. Ici, nous allons utiliser le package sync/atomic pour des compteurs atomiques auxquels accèdent plusieurs goroutines.

Nous utiliserons un entier non signé pour représenter notre compteur (toujours positif).

```
    var ops uint64 = 0
```

Pour simuler des mises à jour concurrentes, nous commençons avec cinquante goroutines qui incrémentent le compteur environ une fois par milliseconde.

```
    for i := 0; i < 50; i++ {
        go func() {
            for {
```

Pour incrémenter atomiquement le compteur, nous utilisons AddUint64, en lui donnant l'adresse mémoire de notre compteur ops avec la syntaxe &.

```
                atomic.AddUint64(&ops, 1)
```

On fait continuer la goroutine.

```
                runtime.Gosched()
            }
        }()
    }
```

On attend une seconde pour permettre aux goroutines de travailler, et au compteur de s'incrémenter.

```
    time.Sleep(time.Second)
```

Afin d'utiliser le compteur de manière sûre alors qu'il est toujours mis à jour par les autres goroutines, on extrait une copie de la valeur courante dans opsFinal via LoadUint64. Comme plus haut, nous devons donner à cette fonction l'adresse mémoire &ops depuis laquelle chercher la valeur.

```
    opsFinal := atomic.LoadUint64(&ops)
    fmt.Println("ops:", opsFinal)
}
```

Code complet

```
package main

import "fmt"
import "time"
import "sync/atomic"
import "runtime"

func main() {

    var ops uint64 = 0

    for i := 0; i < 50; i++ {
        go func() {
            for {


 atomic.AddUint64(&ops, 1)


 runtime.Gosched()
            }
        }()
    }

    time.Sleep(time.Second)

    opsFinal :=
atomic.LoadUint64(&ops)
```

Exécuter le code en ligne

Code complet

```
        fmt.Println("ops:",
    opsFinal)
}
```

Lancer le programme montre qu'environ 40 000 opérations ont été exécutées.

```
$ go run atomic-counters.go
ops: 40200
```

Ensuite nous regarderons les mutex, un autre outil pour gérer les états.

XXXVIII - Mutex

Dans l'exemple précédent, nous avons vu comment gérer des compteurs d'états simples avec des opérations atomiques. Pour des états plus compliqués, on peut utiliser un **mutex** pour accéder de manière sûre à des données à travers plusieurs goroutines.

Pour notre exemple, l'état `state` sera une map.

```
    var state = make(map[int]int)
```

Ce `mutex` va synchroniser l'accès à `state`.

```
    var mutex = &sync.Mutex{}
```

Pour comparer l'approche avec des mutex avec une autre que nous verrons plus tard, `ops` va compter le nombre d'opérations réalisées avec l'état.

```
    var ops int64 = 0
```

Ici on lance cent goroutines pour exécuter des lectures répétées sur l'état.

```
    for r := 0; r < 100; r++ {
        go func() {
            total := 0
            for {
```

À chaque lecture, on sélectionne une clé à laquelle on souhaite accéder, on bloque le mutex avec `Lock()` pour s'assurer un accès exclusif à l'état, on lit la valeur de la clé choisie, on débloque le mutex, puis on incrémente le compteur.

```
                key := rand.Intn(5)
                mutex.Lock()
                total += state[key]
                mutex.Unlock()
                atomic.AddInt64(&ops, 1)
```

Pour nous assurer que cette goroutine ne prend pas toutes les ressources, on rend la main explicitement après chaque opération avec `runtime.Gosched()`. Le programmateur gère normalement automatiquement ceci, par ex. après les opérations sur les canaux et pour les appels bloquants comme `time.Sleep`, mais dans ce cas, on doit le faire manuellement.

```
                runtime.Gosched()
            }
        }()
    }
```

On démarre également dix goroutines pour simuler des écritures, de la même manière que pour les écritures.

```
    for w := 0; w < 10; w++ {
        go func() {
            for {
                key := rand.Intn(5)
                val := rand.Intn(100)
                mutex.Lock()
                state[key] = val
                mutex.Unlock()
                atomic.AddInt64(&ops, 1)
                runtime.Gosched()
            }
        }()
    }
```

On fait travailler les dix goroutines sur les `state` et `mutex` pendant une seconde.

```
    time.Sleep(time.Second)
```

On rapporte le nombre total d'opérations réalisées.

```
    opsFinal := atomic.LoadInt64(&ops)
    fmt.Println("ops:", opsFinal)
```

Avec un verrou final sur le mutex de `state`, on peut connaître l'état final.

```
    mutex.Lock()
    fmt.Println("state:", state)
    mutex.Unlock()
}
```

Code complet

```
package main

import (
    "fmt"
    "math/rand"
    "runtime"
    "sync"
    "sync/atomic"
    "time"
)

func main() {

    var
 state = make(map[int]int)

    var mutex = &sync.Mutex{}

    var ops int64 = 0

    for r := 0; r < 100; r++ {
        go func() {
            total := 0
            for {

                key :=
 rand.Intn(5)
                mutex.Lock()
                total +=
 state[key]
                mutex.Unlock()

 atomic.AddInt64(&ops, 1)
```

Exécuter le code en ligne

Code complet

```
            runtime.Gosched()
                }
            }()
        }

        for w := 0; w < 10; w++ {
            go func() {
                for {
                    key :=
rand.Intn(5)
                    val :=
rand.Intn(100)
                    mutex.Lock()
                    state[key] = val
                    mutex.Unlock()

                    atomic.AddInt64(&ops, 1)

                    runtime.Gosched()
                }
            }()
        }

        time.Sleep(time.Second)

        opsFinal :=
atomic.LoadInt64(&ops)
        fmt.Println("ops:",
opsFinal)

        mutex.Lock()
        fmt.Println("state:", state)
        mutex.Unlock()
    }
```

Lancer le programme montre qu'on a exécuté environ 3 500 000 d'opérations sur state, synchronisées par un mutex.

```
$ go run mutexes.go
ops: 3598302
state: map[1:38 4:98 2:23 3:85 0:44]
```

Ensuite nous verrons comment implémenter la même gestion d'états avec uniquement des goroutines et des canaux.

XXXIX - Goroutines à états

Dans l'exemple précédent, nous avons utilisé un verrou explicite avec des mutex pour synchroniser l'accès à l'état partagé à travers plusieurs goroutines. Une autre option consiste à utiliser les fonctionnalités natives de synchronisation des goroutines pour obtenir le même résultat. Cette approche avec les canaux s'aligne avec l'idée du Go de partager la mémoire en communiquant et en ayant chaque morceau de donnée dans exactement une goroutine.

Dans cet exemple, notre état sera contenu dans une unique goroutine. Cela va garantir que la donnée n'est jamais corrompue par des accès concurrents. Afin de lire ou écrire dans cet état, les autres goroutines vont envoyer des messages à la goroutine propriétaire et recevoir les réponses correspondantes. Ces structures readOp et writeOp encapsulent ces requêtes et une manière pour la goroutine propriétaire de répondre.

```
type readOp struct {
    key  int
    resp chan int
}
type writeOp struct {
    key  int
```

```
    val  int
    resp chan bool
}

func main() {
```

Comme précédemment, nous allons compter le nombre d'opérations réalisées.

```
    var ops int64 = 0
```

Les canaux reads et writes seront utilisés par les autres goroutines pour envoyer des demandes de lecture et d'écriture.

```
    reads := make(chan *readOp)
    writes := make(chan *writeOp)
```

Voici la goroutine qui possède l'état state, qui est une map comme dans l'exemple précédent, mais qui est privée grâce à la goroutine à état. Cette goroutine fait un select de manière répétée sur les canaux reads et writes, et répond aux requêtes à mesure qu'elles arrivent. Une réponse est exécutée en réalisant tout d'abord l'opération demandée, puis en envoyant une valeur sur le canal de réponse resp pour indiquer la réussite (et la valeur désirée dans le cas d'une lecture).

```
    go func() {
        var state = make(map[int]int)
        for {
            select {
            case read := <-reads:
                read.resp <- state[read.key]
            case write := <-writes:
                state[write.key] = write.val
                write.resp <- true
            }
        }
    }()
```

On démarre cent goroutines pour faire des lectures sur la goroutine à état, grâce au canal reads. Chaque lecture nécessite de construire un objet readOp, l'envoyer à travers le canalreads, puis recevoir le résultat à travers le canal resp fourni.

```
    for r := 0; r < 100; r++ {
        go func() {
            for {
                read := &readOp{
                    key:  rand.Intn(5),
                    resp: make(chan int)}
                reads <- read
                <-read.resp
                atomic.AddInt64(&ops, 1)
            }
        }()
    }
```

On démarre dix écritures également, selon la même approche.

```
    for w := 0; w < 10; w++ {
        go func() {
            for {
                write := &writeOp{
                    key:  rand.Intn(5),
                    val:  rand.Intn(100),
                    resp: make(chan bool)}
                writes <- write
                <-write.resp
```

```
                atomic.AddInt64(&ops, 1)
            }
        }()
    }
```

On laisse les goroutines travailler pendant une seconde.

```
    time.Sleep(time.Second)
```

Enfin, on capture et rapporte le nombre d'opérations.

```
    opsFinal := atomic.LoadInt64(&ops)
    fmt.Println("ops:", opsFinal)
}
```

Code complet

```
package main

import (
    "fmt"
    "math/rand"
    "sync/atomic"
    "time"
)

type readOp struct {
    key  int
    resp chan int
}
type writeOp struct {
    key  int
    val  int
    resp chan bool
}

func main() {

    var ops int64 = 0

    reads := make(chan *readOp)

 writes := make(chan *writeOp)

    go func() {
        var
 state = make(map[int]int)
        for {
            select {
            case
 read := <-reads:
                read.resp <-
 state[read.key]
            case
 write := <-writes:

 state[write.key] = write.val

 write.resp <- true
            }
        }
    }()

    for r := 0; r < 100; r++ {
        go func() {
            for {
                read := &readOp{
                    key:
 rand.Intn(5),
```

Exécuter le code en ligne

Code complet

```
resp: make(chan int)}
                reads <- read
                <-read.resp

atomic.AddInt64(&ops, 1)
            }
        }()
    }

    for w := 0; w < 10; w++ {
        go func() {
            for {

write := &writeOp{
                    key:
rand.Intn(5),
                    val:
rand.Intn(100),

resp: make(chan bool)}
                writes <- write
                <-write.resp

atomic.AddInt64(&ops, 1)
            }
        }()
    }

    time.Sleep(time.Second)

    opsFinal :=
atomic.LoadInt64(&ops)
    fmt.Println("ops:",
opsFinal)
}
```

Notre programme montre que la gestion d'état par goroutines réalise environ 800 000 opérations par seconde.

```
$ go run stateful-goroutines.go
ops: 807434
```

Pour ce cas particulier, l'approche avec les goroutines était un peu plus complexe qu'avec les mutex. Elle peut être utile dans certains cas néanmoins, par exemple lorsque vous avez d'autres canaux impliqués ou quand la gestion de tels mutex serait source d'erreurs. Vous devriez utiliser l'approche qui vous parait la plus naturelle, et qui vous permet d'écrire le programme le plus juste possible.

XL - Tri

Le package sort implémente le tri pour les types de base et ceux définis par l'utilisateur. Regardons d'abord le tri pour les types du langage.

Les méthodes de tri sont spécifiques à un type ; voici un exemple pour les chaînes de caractères. À noter que le tri se fait sur place, donc la slice fournie est modifiée et la fonction n'en renvoie pas de nouvelle.

```
    strs := []string{"c", "a", "b"}
    sort.Strings(strs)
    fmt.Println("Strings:", strs)
```

Un exemple de tri sur les int.

```
    ints := []int{7, 2, 4}
    sort.Ints(ints)
```

```
    fmt.Println("Ints:   ", ints)
```

On peut également utiliser sort pour tester si une slice est déjà triée.

```
    s := sort.IntsAreSorted(ints)
    fmt.Println("Sorted: ", s)
}
```

Code complet

```
package main

import "fmt"
import "sort"

func main() {


 strs := []string{"c", "a", "b"}
    sort.Strings(strs)
    fmt.Println("Strings:",
 strs)

    ints := []int{7, 2, 4}
    sort.Ints(ints)
    fmt.Println("Ints:   ",
 ints)

    s :=
 sort.IntsAreSorted(ints)
    fmt.Println("Sorted: ", s)
}
```

Exécuter le code en ligne

Notre programme affiche les slices de chaînes de caractères et d'entiers triées, puis **true** comme résultat de notre test AreSorted.

```
$ go run sorting.go
Strings: [a b c]
Ints:    [2 4 7]
Sorted:  true
```

XLI - Tri par des fonctions

Parfois on veut trier une collection selon un autre critère que l'ordre naturel. Par exemple, imaginons que l'on veuille trier des chaînes par leur longueur plutôt que par ordre alphabétique. Voici un exemple de tri sur mesure en Go.

Afin de trier selon une méthode personnalisée en Go, nous avons besoin d'un type correspondant. Ici, on crée le type ByLength, qui est simplement un alias pour le type intégré []string.

```
type ByLength []string
```

Nous implémentons sort.Interface - Len, Less, et Swap - sur notre type, afin de pouvoir utiliser la fonction générique Sort du package sort. Len et Swap seront généralement similaires à travers les types, et Less va implémenter la logique de notre méthode de tri. Dans notre cas, nous voulons trier par ordre de longueur de chaîne croissante, donc on utilise **len**(s[i]) et **len**(s[j]) ici.

```
func (s ByLength) Len() int {
    return len(s)
}
func (s ByLength) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}
func (s ByLength) Less(i, j int) bool {
```

```
    return len(s[i]) < len(s[j])
}
```

Avec tout ceci en place, nous pouvons maintenant implémenter notre tri sur mesure en convertissant la slice de départ fruits en ByLength, et ensuite utiliser la méthode sort.Sort sur cette slice typée.

```
func main() {
    fruits := []string{"peach", "banana", "kiwi"}
    sort.Sort(ByLength(fruits))
    fmt.Println(fruits)
}
```

Code complet

```
package main

import "sort"
import "fmt"

type ByLength []string

func (s ByLength) Len() int {
    return len(s)
}
func (s ByLength) Swap(i,
 j int) {
    s[i], s[j] = s[j], s[i]
}
func (s ByLength) Less(i,
 j int) bool {
    return len(s[i]) < len(s[j])
}

func main() {

 fruits := []string{"peach", "banana
    sort.Sort(ByLength(fruits))
    fmt.Println(fruits)
}
```

Exécuter le code en ligne

Exécuter notre programme montre une liste triée par longueur de chaîne, comme désiré.

```
$ go run sorting-by-functions.go
[kiwi peach banana]
```

En suivant ce modèle, qui consiste à créer un type personnalisé, implémenter les trois méthodes de l'interface sur ce type, et ensuite appeler sort.Sort sur une collection de ce type personnalisé, nous pouvons trier des slices selon n'importe quel critère.

XLII - Panic

Un **panic** signifie typiquement que quelque chose s'est mal passé et de manière inattendue. On l'utilise principalement pour faire échouer rapidement suite à des erreurs qui ne devraient normalement pas arriver, ou que nous ne pouvons pas traiter.

Nous allons utiliser panic dans les exemples de ce site pour traiter les cas imprévus. Ce programme-ci est le seul du site qui va paniquer durant son déroulement normal.

```
func main() {

    panic("a problem")
```

Une utilisation courante de **panic**, c'est d'abandonner si une fonction renvoie une valeur d'erreur que nous ne savons (ou ne voulons) pas traiter. Voici un exemple de panique si on a une erreur inattendue lors de la création d'un nouveau fichier.

```
    _, err := os.Create("/tmp/file")
    if err != nil {
        panic(err)
    }
}
```

Code complet

```
package main

import "os"

func main() {

    panic("a problem")

    _, err := os.Create("/tmp/
file")
    if err != nil {
        panic(err)
    }
}
```

Exécuter le code en ligne

Ce programme va causer un panic, qui affiche un message d'erreur, une trace d'exécution, et quitte le programme en renvoyant une valeur de statut non nulle.

```
$ go run panic.go
panic: a problem

goroutine 1 [running]:
main.main()
    /.../panic.go:12 +0x47
...
exit status 2
```

À noter que contrairement à certains langages qui utilisent des exceptions pour la gestion de nombreuses erreurs, en Go il est idiomatique d'utiliser des valeurs de retour qui indiquent le statut d'erreur dès que possible.

XLIII - Defer

Defer est utilisé pour s'assurer qu'un appel de fonction est réalisé en dernier à la fin de l'exécution d'un programme, en général pour libérer les ressources. On utilise souvent **defer** là où on aurait utilisé ensure et finally dans d'autres langages.

Supposons que nous voulions créer un fichier, y écrire et le fermer quand on aura terminé. Voici comment nous pourrions le faire avec **defer**.

```
func main() {
```

Immédiatement après avoir obtenu un objet de type fichier avec createFile, on repousse la fermeture de ce fichier avec closeFile. Cette fonction sera exécutée à la fin de la fonction où se trouve l'appel (ici, main), c'est-à-dire après que writeFile a terminé.

```
    f := createFile("/tmp/defer.txt")
    defer closeFile(f)
    writeFile(f)
}
func createFile(p string) *os.File {
```

```
    fmt.Println("creating")
    f, err := os.Create(p)
    if err != nil {
        panic(err)
    }
    return f
}
func writeFile(f *os.File) {
    fmt.Println("writing")
    fmt.Fprintln(f, "data")

}
func closeFile(f *os.File) {
    fmt.Println("closing")
    f.Close()
}
```

Code complet

```
package main

import "fmt"
import "os"

func main() {

    f := createFile("/tmp/
defer.txt")
    defer closeFile(f)
    writeFile(f)
}

func
 createFile(p string) *os.File {
    fmt.Println("creating")
    f, err := os.Create(p)
    if err != nil {
        panic(err)
    }
    return f
}

func writeFile(f *os.File) {
    fmt.Println("writing")
    fmt.Fprintln(f, "data")

}

func closeFile(f *os.File) {
    fmt.Println("closing")
    f.Close()
}
```

Exécuter le code en ligne

Le programme confirme que le fichier est fermé après avoir été écrit.

```
$ go run defer.go
creating
writing
closing
```

XLIV - Fonctions sur les collections

On a souvent besoin que les programmes réalisent des opérations sur des collections de données, comme sélectionner tous les éléments qui correspondent à un prédicat, ou associer les éléments à une autre collection selon une fonction donnée.

Dans certains langages il est idiomatique d'utiliser des structures de données et des algorithmes **génériques**. Go ne supporte pas les types génériques. En Go, il est courant de fournir des fonctions sur les collections spécialement pour un type donné, lorsque c'est spécifiquement nécessaire pour le programme.

Voici quelques exemples de fonctions sur les collections pour des slices de strings. Vous pouvez utiliser ces exemples pour concevoir vos propres fonctions. À noter que dans certains cas, il peut être plus clair d'utiliser le code de manipulation de collections inline, plutôt que créer et appeler des fonctions auxiliaires.

Renvoie le premier index où se trouve la chaîne cible t, ou -1 si aucune correspondance n'est trouvée.

```
func Index(vs []string, t string) int {
    for i, v := range vs {
        if v == t {
            return i
        }
    }
    return -1
}
```

Renvoie **true** si la chaîne cible t est dans la slice.

```
func Include(vs []string, t string) bool {
    return Index(vs, t) >= 0
}
```

Renvoie **true** si une des chaînes de la slice satisfait le prédicat f.

```
func Any(vs []string, f func(string) bool) bool {
    for _, v := range vs {
        if f(v) {
            return true
        }
    }
    return false
}
```

Renvoie **true** si toutes les chaînes de la slice satisfont le prédicat f.

```
func All(vs []string, f func(string) bool) bool {
    for _, v := range vs {
        if !f(v) {
            return false
        }
    }
    return true
}
```

Renvoie une nouvelle slice contenant toutes les chaînes de la slice qui satisfont le prédicat f.

```
func Filter(vs []string, f func(string) bool) []string {
    vsf := make([]string, 0)
    for _, v := range vs {
        if f(v) {
            vsf = append(vsf, v)
        }
    }
    return vsf
}
```

Renvoie une nouvelle slice contenant le résultat de l'application de la fonction f à chacune des chaînes de la slice de départ.

```
func Map(vs []string, f func(string) string) []string {
    vsm := make([]string, len(vs))
    for i, v := range vs {
        vsm[i] = f(v)
    }
    return vsm
}

func main() {
```

Maintenant, on essaye nos différentes fonctions sur les collections.

```
    var strs = []string{"peach", "apple", "pear", "plum"}

    fmt.Println(Index(strs, "pear"))

    fmt.Println(Include(strs, "grape"))
    fmt.Println(Any(strs, func(v string) bool {
        return strings.HasPrefix(v, "p")
    }))
    fmt.Println(All(strs, func(v string) bool {
        return strings.HasPrefix(v, "p")
    }))
    fmt.Println(Filter(strs, func(v string) bool {
        return strings.Contains(v, "e")
    }))
```

Les exemples ci-dessus ont tous utilisé des fonctions anonymes, mais vous pouvez également utiliser des fonctions nommées (à condition qu'elles soient du bon type).

```
    fmt.Println(Map(strs, strings.ToUpper))
}
```

Code complet

```
package main

import "strings"
import "fmt"

func Index(vs []string,
 t string) int {
    for i, v := range vs {
        if v == t {
            return i
        }
    }
    return -1
}

func Include(vs []string,
 t string) bool {
    return Index(vs, t) >= 0
}

func Any(vs []string,
 f func(string) bool) bool {
    for _, v := range vs {
        if f(v) {
            return true
        }
    }
    return false
}

func All(vs []string,
 f func(string) bool) bool {
    for _, v := range vs {
        if !f(v) {
```

Exécuter le code en ligne

Code complet

```
            return false
        }
    }
    return true
}

func Filter(vs []string,
 f func(string) bool) []string {
    vsf := make([]string, 0)
    for _, v := range vs {
        if f(v) {
            vsf = append(vsf, v)
        }
    }
    return vsf
}

func Map(vs []string,
 f func(string) string) []string {

 vsm := make([]string, len(vs))
    for i, v := range vs {
        vsm[i] = f(v)
    }
    return vsm
}

func main() {

    var
 strs = []string{"peach", "apple", "

 fmt.Println(Index(strs, "pear"))

 fmt.Println(Include(strs, "grape"))

 fmt.Println(Any(strs, func(v string
        return
 strings.HasPrefix(v, "p")
    }))

 fmt.Println(All(strs, func(v string
        return
 strings.HasPrefix(v, "p")
    }))

 fmt.Println(Filter(strs, func(v str
        return
 strings.Contains(v, "e")
    }))

    fmt.Println(Map(strs,
 strings.ToUpper))

}
```

```
$ go run collection-functions.go
2
false
true
false
[peach apple pear]
[PEACH APPLE PEAR PLUM]
```

XLV - Fonctions sur les chaînes

Le package strings de la bibliothèque standard fournit de nombreuses fonctions utiles pour la manipulation des chaînes de caractères. En voici un aperçu.

On crée un alias sur fmt.Println vers un nom plus court, car on va beaucoup s'en servir ensuite.

```
var p = fmt.Println

func main() {
```

Voici un exemple des fonctions disponibles dans strings. Notez que ce sont toutes des fonctions du package, pas des méthodes sur l'objet string lui-même. Cela signifie que vous devez passer la chaîne en question comme premier argument des fonctions.

```
    p("Contains:  ", s.Contains("test", "es"))
    p("Count:     ", s.Count("test", "t"))
    p("HasPrefix: ", s.HasPrefix("test", "te"))
    p("HasSuffix: ", s.HasSuffix("test", "st"))
    p("Index:     ", s.Index("test", "e"))
    p("Join:      ", s.Join([]string{"a", "b"}, "-"))
    p("Repeat:    ", s.Repeat("a", 5))
    p("Replace:   ", s.Replace("foo", "o", "0", -1))
    p("Replace:   ", s.Replace("foo", "o", "0", 1))
    p("Split:     ", s.Split("a-b-c-d-e", "-"))
    p("ToLower:   ", s.ToLower("TEST"))
    p("ToUpper:   ", s.ToUpper("test"))
    p()
```

Vous pourrez trouver plus de fonctions dans la documentation du package **strings**.

Cela ne fait pas partie de strings, mais il est utile de mentionner qu'on peut obtenir la longueur d'une chaîne avec **len** et qu'on peut accéder à un caractère en particulier via son index.

```
    p("Len: ", len("hello"))
    p("Char:", "hello"[1])
}
```

Code complet

```
package main

import s "strings"
import "fmt"

var p = fmt.Println

func main() {

    p("Contains:  ",
 s.Contains("test", "es"))
    p("Count:     ",
 s.Count("test", "t"))
    p("HasPrefix: ",
 s.HasPrefix("test", "te"))
    p("HasSuffix: ",
 s.HasSuffix("test", "st"))
    p("Index:     ",
 s.Index("test", "e"))
    p("Join:      ",
 s.Join([]string{"a", "b"}, "-"))
    p("Repeat:    ",
 s.Repeat("a", 5))
```

Exécuter le code en ligne

Code complet

```
    p("Replace:    ",
  s.Replace("foo", "o", "0", -1))
    p("Replace:    ",
  s.Replace("foo", "o", "0", 1))
    p("Split:      ", s.Split("a-
b-c-d-e", "-"))
    p("ToLower:    ",
  s.ToLower("TEST"))
    p("ToUpper:    ",
  s.ToUpper("test"))
    p()

    p("Len: ", len("hello"))
    p("Char:", "hello"[1])
}
```

```
$ go run string-functions.go
Contains:   true
Count:      2
HasPrefix:  true
HasSuffix:  true
Index:      1
Join:       a-b
Repeat:     aaaaa
Replace:    f00
Replace:    f0o
Split:      [a b c d e]
toLower:    test
ToUpper:    TEST

Len:  5
Char: 101
```

XLVI - Formatage de chaînes

Go propose un excellent support du formatage façon printf. Voici quelques exemples de formatages courants.

Go propose plusieurs « verbes » d'affichage, conçus pour formater des valeurs générales en Go. Par exemple, ceci affiche une instance de notre structure point.

```
    p := point{1, 2}
    fmt.Printf("%v\n", p)
```

Si la valeur est une **struct**, la variable %+v va inclure les noms des champs.

```
    fmt.Printf("%+v\n", p)
```

La variante %#v affiche une représentation en syntaxe Go de la valeur, c'est-à-dire le bout de code qui produirait cette valeur.

```
    fmt.Printf("%#v\n", p)
```

Pour afficher le type d'une valeur, on utilise %T.

```
    fmt.Printf("%T\n", p)
```

Formater des booléens est simple.

```
    fmt.Printf("%t\n", true)
```

Il y a beaucoup d'options pour formater des entiers. %d est le formatage standard, en base 10.

```
fmt.Printf("%d\n", 123)
```

%b fournit une représentation binaire.

```
fmt.Printf("%b\n", 14)
```

%c affiche le caractère correspondant à l'entier donné.

```
fmt.Printf("%c\n", 33)
```

%x renvoie l'encodage hexadécimal.

```
fmt.Printf("%x\n", 456)
```

Il y a également plusieurs options de formatage pour les float. Pour du formatage basique décimal, utilisez %f.

```
fmt.Printf("%f\n", 78.9)
```

%e et %E formatent les float en (des versions légèrement modifiées de) notation scientifique.

```
fmt.Printf("%e\n", 123400000.0)
fmt.Printf("%E\n", 123400000.0)
```

Pour afficher des chaînes basiques, utilisez %s.

```
fmt.Printf("%s\n", "\"string\"")
```

Pour des chaînes avec double-quote comme en code Go, utilisez %q.

```
fmt.Printf("%q\n", "\"string\"")
```

Comme avec les entiers vus plus tôt %x affiche la chaîne en base 16, avec deux caractères de sortie par octet d'entrée.

```
fmt.Printf("%x\n", "hex this")
```

Pour afficher une représentation d'un pointeur, utilisez %p.

```
fmt.Printf("%p\n", &p)
```

Lorsqu'on formate des nombres, on veut souvent contrôler la largeur et la précision du chiffre en sortie. Pour spécifier la largeur d'un entier, on met un nombre après le % dans le verbe. Par défaut le résultat est justifié et complété avec des espaces.

```
fmt.Printf("|%6d|%6d|\n", 12, 345)
```

On peut aussi préciser la largeur des floats, mais on peut cette fois-ci également préciser la largeur de la précision avec la syntaxe width.precision.

```
fmt.Printf("|%6.2f|%6.2f|\n", 1.2, 3.45)
```

Pour justifier à gauche, ajoutez -.

```
    fmt.Printf("|%-6.2f|%-6.2f|\n", 1.2, 3.45)
```

Vous pouvez aussi vouloir contrôler la largeur lorsque vous formatez des chaînes, en particulier pour contrôler qu'elles s'alignent comme dans des tableaux. Voici un exemple justifié à droite :

```
    fmt.Printf("|%6s|%6s|\n", "foo", "b")
```

Pour justifier à gauche, on utilise -, comme pour les nombres.

```
    fmt.Printf("|%-6s|%-6s|\n", "foo", "b")
```

Pour le moment nous avons vu Printf, qui affiche la chaîne dans os.Stdout. Sprintf formate et renvoie la chaîne sans l'afficher.

```
    s := fmt.Sprintf("a %s", "string")
    fmt.Println(s)
```

On peut formater et afficher dans des io.Writers autres que os.Stdout avec Fprintf.

```
    fmt.Fprintf(os.Stderr, "an %s\n", "error")
}
```

Code complet

```
package main

import "fmt"
import "os"

type point struct {
    x, y int
}

func main() {

    p := point{1, 2}
    fmt.Printf("%v\n", p)

    fmt.Printf("%+v\n", p)

    fmt.Printf("%#v\n", p)

    fmt.Printf("%T\n", p)

    fmt.Printf("%t\n", true)

    fmt.Printf("%d\n", 123)

    fmt.Printf("%b\n", 14)

    fmt.Printf("%c\n", 33)

    fmt.Printf("%x\n", 456)

    fmt.Printf("%f\n", 78.9)


 fmt.Printf("%e\n", 123400000.0)

 fmt.Printf("%E\n", 123400000.0)


 fmt.Printf("%s\n", "\"string\"")


 fmt.Printf("%q\n", "\"string\"")
```

Exécuter le code en ligne

Code complet

```
    fmt.Printf("%x\n", "hex
 this")

    fmt.Printf("%p\n", &p)

    fmt.Printf("|%6d|
%6d|\n", 12, 345)

    fmt.Printf("|%6.2f|
%6.2f|\n", 1.2, 3.45)

    fmt.Printf("|%-6.2f|
%-6.2f|\n", 1.2, 3.45)

    fmt.Printf("|%6s|
%6s|\n", "foo", "b")

    fmt.Printf("|%-6s|
%-6s|\n", "foo", "b")

    s := fmt.Sprintf("a
 %s", "string")
    fmt.Println(s)

    fmt.Fprintf(os.Stderr, "an
 %s\n", "error")
}
```

```
$ go run string-formatting.go
{1 2}
{x:1 y:2}
main.point{x:1, y:2}
main.point
true
123
1110
!
1c8
78.900000
1.234000e+08
1.234000E+08
"string"
"\"string\""
6865782074686973
0x42135100
|    12|   345|
|  1.20|  3.45|
|1.20  |3.45  |
|   foo|     b|
|foo   |b     |
a string
an error
```

XLVII - Expressions régulières

Go offre de base un support des **expressions régulières** (ou *regex*, pour *regular expression*). Voici quelques exemples de tâches courantes sur les regex en Go.

Ceci teste si un motif **"p([a-z]+)ch"** correspond à une chaîne.

```
    match, _ := regexp.MatchString("p([a-z]+)ch", "peach")
    fmt.Println(match)
```

Au-dessus, nous avons utilisé un motif directement, mais pour d'autres tâches, il faut la compiler en une structure Regexp optimisée.

```
    r, _ := regexp.Compile("p([a-z]+)ch")
```

Plusieurs méthodes sont disponibles. Voici le même test que celui vu plus tôt.

```
    fmt.Println(r.MatchString("peach"))
```

Ceci trouve l'association pour la regex.

```
    fmt.Println(r.FindString("peach punch"))
```

Ceci trouve également la première correspondance, mais renvoie les indices de début et fin de la correspondance, au lieu de renvoyer le texte correspondant.

```
    fmt.Println(r.FindStringIndex("peach punch"))
```

La variante Submatch inclut des informations sur les associations au niveau du pattern entier, ainsi que les sous-correspondances à l'intérieur du motif. Par exemple ici, cela renverra des informations à la fois sur p([a-z]+)ch et ([a-z]+).

```
    fmt.Println(r.FindStringSubmatch("peach punch"))
```

De la même manière, cela donne des informations sur les indices de correspondance globale et les sous-correspondances.

```
    fmt.Println(r.FindStringSubmatchIndex("peach punch"))
```

Les variantes All de ces fonctions s'appliquent à toutes les correspondances du texte d'entrée, pas juste la première. Par exemple pour trouver toutes les correspondances d'une regex.

```
    fmt.Println(r.FindAllString("peach punch pinch", -1))
```

Ces variantes All sont également disponibles pour toutes les autres méthodes vues plus haut.

```
    fmt.Println(r.FindAllStringSubmatchIndex("peach punch pinch", -1))
```

Fournir un entier non négatif en second argument à ces fonctions limite le nombre de résultats.

```
    fmt.Println(r.FindAllString("peach punch pinch", 2))
```

Nos exemples ci-dessus avaient comme arguments des chaînes de caractères et avaient des noms comme MatchString. On peut également fournir des arguments []byte et oublier le String des noms des fonctions.

```
    fmt.Println(r.Match([]byte("peach")))
```

Lorsqu'on crée des constantes avec des expressions régulières, on peut utiliser la variation MustCompile de Compile. Un simple Compile ne marchera pas, car il a deux valeurs de retour.

```
    r = regexp.MustCompile("p([a-z]+)ch")
    fmt.Println(r)
```

Le package regexp peut également être utilisé pour remplacer des sous-ensembles de chaînes avec d'autres valeurs.

```
    fmt.Println(r.ReplaceAllString("a peach", "<fruit>"))
```

La variante Func permet de transformer les textes de correspondance selon une fonction donnée.

```
    in := []byte("a peach")
    out := r.ReplaceAllFunc(in, bytes.ToUpper)
    fmt.Println(string(out))
}
```

Code complet

```
package main

import "bytes"
import "fmt"
import "regexp"

func main() {

    match, _ :=
 regexp.MatchString("p([a-
z]+)ch", "peach")
    fmt.Println(match)

    r, _ :=
 regexp.Compile("p([a-z]+)ch")


 fmt.Println(r.MatchString("peach"))


 fmt.Println(r.FindString("peach
 punch"))


 fmt.Println(r.FindStringIndex("peac
 punch"))


 fmt.Println(r.FindStringSubmatch("p
 punch"))


 fmt.Println(r.FindStringSubmatchInd
 punch"))


 fmt.Println(r.FindAllString("peach
 punch pinch", -1))


 fmt.Println(r.FindAllStringSubmatch
 punch pinch", -1))


 fmt.Println(r.FindAllString("peach
 punch pinch", 2))


 fmt.Println(r.Match([]byte("peach")

    r =
 regexp.MustCompile("p([a-
z]+)ch")
    fmt.Println(r)


 fmt.Println(r.ReplaceAllString("a
 peach", "<fruit>"))
```

Exécuter le code en ligne

Code complet

```
    in := []byte("a peach")
    out := r.ReplaceAllFunc(in,
 bytes.ToUpper)
    fmt.Println(string(out))
}
```

```
$ go run regular-expressions.go
true
true
peach
[0 5]
[peach ea]
[0 5 1 3]
[peach punch pinch]
[[0 5 1 3] [6 11 7 9] [12 17 13 15]]
[peach punch]
true
p([a-z]+)ch
a <fruit>
a PEACH
```

Pour une référence complète sur les expressions régulières en Go, regardez la documentation du package **regexp**.

XLVIII - JSON

Go fournit un support pour l'encodage et le décodage en JSON, à la fois pour les types intégrés, mais aussi pour les types sur mesure.

Nous allons utiliser ces deux structures pour démontrer l'encodage et le décodage vers des types de données personnalisés plus bas.

```
type Response1 struct {
    Page   int
    Fruits []string
}
type Response2 struct {
    Page   int      `json:"page"`
    Fruits []string `json:"fruits"`
}

func main() {
```

Nous allons commencer par encoder des types de données de base vers une chaîne JSON. Voici quelques exemples pour des valeurs atomiques.

```
    bolB, _ := json.Marshal(true)
    fmt.Println(string(bolB))

    intB, _ := json.Marshal(1)
    fmt.Println(string(intB))

    fltB, _ := json.Marshal(2.34)
    fmt.Println(string(fltB))

    strB, _ := json.Marshal("gopher")
    fmt.Println(string(strB))
```

Et voici quelques exemples pour les slices et les maps, qui encodent vers des tableaux et objets JSON comme attendu.

```
    slcD := []string{"apple", "peach", "pear"}
    slcB, _ := json.Marshal(slcD)
    fmt.Println(string(slcB))
```

```
    mapD := map[string]int{"apple": 5, "lettuce": 7}
    mapB, _ := json.Marshal(mapD)
    fmt.Println(string(mapB))
```

Le package JSON peut automatiquement encoder vos types de données personnalisés. Il inclura seulement les champs exportés dans la sortie encodée et utilisera par défaut ces noms comme clés JSON.

```
    res1D := &Response1{
        Page:   1,
        Fruits: []string{"apple", "peach", "pear"}}
    res1B, _ := json.Marshal(res1D)
    fmt.Println(string(res1B))
```

Vous pouvez utiliser des tags sur les déclarations de champs de structures pour personnaliser l'encodage des noms de clé JSON. Regardez la définition de Response2 au-dessus pour voir un exemple de tels tags.

```
    res2D := &Response2{
        Page:   1,
        Fruits: []string{"apple", "peach", "pear"}}
    res2B, _ := json.Marshal(res2D)
    fmt.Println(string(res2B))
```

Maintenant regardons le décodage des données JSON vers des valeurs. Voici un exemple pour une structure de données générique.

```
    byt := []byte(`{"num":6.13,"strs":["a","b"]}`)
```

Nous devons fournir une variable dans laquelle le package JSON peut décoder les données. Cette map[string]interface{} contiendra une map avec string comme type de valeurs arbitraire.

```
    var dat map[string]interface{}
```

Voici le décodage, accompagné d'un test d'erreurs.

```
    if err := json.Unmarshal(byt, &dat); err != nil {
        panic(err)
    }
    fmt.Println(dat)
```

Pour utiliser les données dans la map décodée, nous devons les convertir dans le type approprié. Par exemple ici, on convertit num en float64.

```
    num := dat["num"].(float64)
    fmt.Println(num)
```

Accéder aux données imbriquées nécessite une série de casts.

```
    strs := dat["strs"].([]interface{})
    str1 := strs[0].(string)
    fmt.Println(str1)
```

Nous pouvons aussi décoder du JSON dans des types de données personnalisés. Ceci a également l'avantage d'assurer une sécurité au niveau des types de données, et d'éliminer le besoin de vérifier les types lorsqu'on accède aux données décodées.

```
    str := `{"page": 1, "fruits": ["apple", "peach"]}`
    res := Response2{}
    json.Unmarshal([]byte(str), &res)
    fmt.Println(res)
    fmt.Println(res.Fruits[0])
```

Dans les exemples ci-dessus, nous avons toujours utilisé des octets et des chaînes comme données intermédiaires entre les données et leur représentation JSON sur la sortie standard. On peut également streamer l'encodage directement dans un os.Writer comme os.Stdout, ou même au sein d'une réponse HTTP.

```
    enc := json.NewEncoder(os.Stdout)
    d := map[string]int{"apple": 5, "lettuce": 7}
    enc.Encode(d)
}
```

Code complet

```
package main

import "encoding/json"
import "fmt"
import "os"

type Response1 struct {
    Page   int
    Fruits []string
}
type Response2 struct {
    Page   int
  `json:"page"`

 Fruits []string `json:"fruits"`
}

func main() {

    bolB, _ :=
 json.Marshal(true)
    fmt.Println(string(bolB))

    intB, _ := json.Marshal(1)
    fmt.Println(string(intB))

    fltB, _ :=
 json.Marshal(2.34)
    fmt.Println(string(fltB))

    strB, _ :=
 json.Marshal("gopher")
    fmt.Println(string(strB))


 slcD := []string{"apple", "peach",
    slcB, _ :=
 json.Marshal(slcD)
    fmt.Println(string(slcB))


 mapD := map[string]int{"apple": 5,
    mapB, _ :=
 json.Marshal(mapD)
    fmt.Println(string(mapB))

    res1D := &Response1{
        Page:   1,

 Fruits: []string{"apple", "peach",
    res1B, _ :=
 json.Marshal(res1D)
    fmt.Println(string(res1B))

    res2D := &Response2{
        Page:   1,

 Fruits: []string{"apple", "peach",
    res2B, _ :=
 json.Marshal(res2D)
```

Exécuter le code en ligne

Code complet

```
    fmt.Println(string(res2B))


 byt := []byte(`{"num":6.13,"strs":
["a","b"]}`)

    var
 dat map[string]interface{}

    if err :=
 json.Unmarshal(byt, &dat);
 err != nil {
        panic(err)
    }
    fmt.Println(dat)

    num := dat["num"].(float64)
    fmt.Println(num)

    strs := dat["strs"].
([]interface{})
    str1 := strs[0].(string)
    fmt.Println(str1)

    str := `{"page": 1,
 "fruits": ["apple", "peach"]}`
    res := Response2{}

 json.Unmarshal([]byte(str), &res)
    fmt.Println(res)
    fmt.Println(res.Fruits[0])

    enc :=
 json.NewEncoder(os.Stdout)

 d := map[string]int{"apple": 5, "le
    enc.Encode(d)
}
```

```
$ go run json.go
true
1
2.34
"gopher"
["apple","peach","pear"]
{"apple":5,"lettuce":7}
{"Page":1,"Fruits":["apple","peach","pear"]}
{"page":1,"fruits":["apple","peach","pear"]}
map[num:6.13 strs:[a b]]
6.13
a
{1 [apple peach]}
apple
{"apple":5,"lettuce":7}
```

Nous avons couvert les bases du JSON en Go ici, mais regardez l'article sur **JSON et le Go** et la documentation **du package JSON** pour en savoir plus.

XLIX - Dates

Go fournit un support complet des fonctions usuelles de gestion du temps et des durées. Voici quelques exemples.

Commençons par récupérer l'heure actuelle.

```
    now := time.Now()
    p(now)
```

On peut construire une structure time en fournissant année, mois, jour, etc. Les time sont toujours associés à un lieu, c'est-à-dire à un fuseau horaire.

```
    then := time.Date(
        2009, 11, 17, 20, 34, 58, 651387237, time.UTC)
    p(then)
```

On peut extraire les différentes valeurs des composantes du temps comme on s'y attend.

```
    p(then.Year())
    p(then.Month())
    p(then.Day())
    p(then.Hour())
    p(then.Minute())
    p(then.Second())
    p(then.Nanosecond())
    p(then.Location())
```

Une méthode Weekday est également disponible pour connaître le jour de la semaine (lundi-dimanche).

```
    p(then.Weekday())
```

Ces méthodes comparent deux instances de time, pour savoir si la première est arrivée respectivement avant, après ou au même moment que la seconde.

```
    p(then.Before(now))
    p(then.After(now))
    p(then.Equal(now))
```

La méthode Sub renvoie une durée qui représente l'intervalle écoulé entre les dates contenues dans deux instances de time.

```
    diff := now.Sub(then)
    p(diff)
```

On peut calculer la longueur de la durée dans plusieurs unités.

```
    p(diff.Hours())
    p(diff.Minutes())
    p(diff.Seconds())
    p(diff.Nanoseconds())
```

On peut aussi utiliser Add pour avancer à une date selon une certaine durée, ou reculer avec -.

```
    p(then.Add(diff))
    p(then.Add(-diff))
}
```

Code complet

```
package main

import "fmt"
import "time"

func main() {
    p := fmt.Println

    now := time.Now()
    p(now)

    then := time.Date(
```

Exécuter le code en ligne

Code complet

```
    2009, 11, 17, 20, 34, 58, 65138723
  time.UTC)
    p(then)

    p(then.Year())
    p(then.Month())
    p(then.Day())
    p(then.Hour())
    p(then.Minute())
    p(then.Second())
    p(then.Nanosecond())
    p(then.Location())

    p(then.Weekday())

    p(then.Before(now))
    p(then.After(now))
    p(then.Equal(now))

    diff := now.Sub(then)
    p(diff)

    p(diff.Hours())
    p(diff.Minutes())
    p(diff.Seconds())
    p(diff.Nanoseconds())

    p(then.Add(diff))
    p(then.Add(-diff))
}
```

```
$ go run time.go
2012-10-31 15:50:13.793654 +0000 UTC
2009-11-17 20:34:58.651387237 +0000 UTC
2009
November
17
20
34
58
651387237
UTC
Tuesday
true
false
false
25891h15m15.142266763s
25891.25420618521
1.5534752523711128e+06
9.320851514226677e+07
93208515142266763
2012-10-31 15:50:13.793654 +0000 UTC
2006-12-05 01:19:43.509120474 +0000 UTC
```

Ensuite, nous allons regarder l'idée liée au temps qui se rapporte au temps Unix.

L - Temps UNIX

Un besoin courant des programmes, c'est d'obtenir le nombre de secondes, millisecondes ou nanosecondes depuis l'**heure Unix**. Voici comment le faire en Go.

On utilise time.Now avec Unix ou UnixNano pour obtenir le temps écoulé depuis l'heure Unix en secondes ou en nanosecondes.

```
    now := time.Now()
```

```
    secs := now.Unix()
    nanos := now.UnixNano()
    fmt.Println(now)
```

À noter qu'il n'y a pas de UnixMillis, donc pour obtenir des millisecondes, il faut faire la division soi-même à partir des nanosecondes.

```
    millis := nanos / 1000000
    fmt.Println(secs)
    fmt.Println(millis)
    fmt.Println(nanos)
```

On peut aussi convertir des secondes ou des nanosecondes depuis l'heure Unix.

```
    fmt.Println(time.Unix(secs, 0))
    fmt.Println(time.Unix(0, nanos))
}
```

Code complet

```
package main

import "fmt"
import "time"

func main() {

    now := time.Now()
    secs := now.Unix()
    nanos := now.UnixNano()
    fmt.Println(now)

    millis := nanos / 1000000
    fmt.Println(secs)
    fmt.Println(millis)
    fmt.Println(nanos)


 fmt.Println(time.Unix(secs, 0))
    fmt.Println(time.Unix(0,
 nanos))
}
```

Exécuter le code en ligne

```
$ go run epoch.go
2012-10-31 16:13:58.292387 +0000 UTC
1351700038
1351700038292
1351700038292387000
2012-10-31 16:13:58 +0000 UTC
2012-10-31 16:13:58.292387 +0000 UTC
```

Ensuite, nous verrons une autre tâche liée au temps : parser et formater des dates.

LI - Formatage et analyse de dates

Go supporte le formatage et le parsing de temps via des motifs.

Voici un exemple basique de formatage de temps selon la RFC3339, qui utilise la constante correspondante.

```
    t := time.Now()
    p(t.Format(time.RFC3339))
```

On parse le temps en utilisant la même constante dans Format.

```
    t1, e := time.Parse(
        time.RFC3339,
        "2012-11-01T22:08:41+00:00")
    p(t1)
```

Format et Parse utilisent des motifs d'exemples. En général, vous utiliserez une constante de time pour cela, mais vous pouvez aussi fournir votre propre formatage. Le formatage doit utiliser le temps de référence Mon Jan 2 15:04:05 MST 2006 pour montrer le motif avec lequel formater/parser un temps ou une chaîne. Le temps d'exemple doit être exactement comme ceci : l'année 2006, 15 pour l'heure, lundi pour le jour de la semaine, etc.

```
    p(t.Format("3:04PM"))
    p(t.Format("Mon Jan _2 15:04:05 2006"))
    p(t.Format("2006-01-02T15:04:05.999999-07:00"))
    form := "3 04 PM"
    t2, e := time.Parse(form, "8 41 PM")
    p(t2)
```

Pour des représentations purement numériques, vous pouvez également utiliser des fonctions standard de formatage en extrayant les composantes de temps nécessaires.

```
    fmt.Printf("%d-%02d-%02dT%02d:%02d:%02d-00:00\n",
        t.Year(), t.Month(), t.Day(),
        t.Hour(), t.Minute(), t.Second())
```

Parse renvoie une erreur qui explique le problème lorsque le format d'entrée est mal formé.

```
    ansic := "Mon Jan _2 15:04:05 2006"
    _, e = time.Parse(ansic, "8:41PM")
    p(e)
}
```

Code complet

```
package main

import "fmt"
import "time"

func main() {
    p := fmt.Println

    t := time.Now()
    p(t.Format(time.RFC3339))

    t1, e := time.Parse(
        time.RFC3339,

  "2012-11-01T22:08:41+00:00")
    p(t1)

    p(t.Format("3:04PM"))
    p(t.Format("Mon Jan _2
 15:04:05 2006"))

 p(t.Format("2006-01-02T15:04:05.999
    form := "3 04 PM"
    t2, e := time.Parse(form, "8
 41 PM")
    p(t2)

    fmt.Printf("%d-%02d-%02dT
%02d:%02d:%02d-00:00\n",
        t.Year(), t.Month(),
 t.Day(),
        t.Hour(), t.Minute(),
 t.Second())
```

Exécuter le code en ligne

Code complet

```
    ansic := "Mon Jan _2 15:04:05 2006"
    _, e = time.Parse(ansic, "8:41PM")
    p(e)
}
```

```
$ go run time-formatting-parsing.go
2014-04-15T18:00:15-07:00
2012-11-01 22:08:41 +0000 +0000
6:00PM
Tue Apr 15 18:00:15 2014
2014-04-15T18:00:15.161182-07:00
0000-01-01 20:41:00 +0000 UTC
2014-04-15T18:00:15-00:00
parsing time "8:41PM" as "Mon Jan _2 15:04:05 2006": ...
```

LII - Nombres aléatoires

Le package math/rand de Go permet la génération de nombres **pseudoaléatoires**.

Par exemple, rand.Intn renvoie un int n, tel que 0 <= n < 100.

```
    fmt.Print(rand.Intn(100), ",")
    fmt.Print(rand.Intn(100))
    fmt.Println()
```

rand.Float64 renvoie un float64 f, tel que 0.0 <= f < 1.0.

```
    fmt.Println(rand.Float64())
```

On peut s'en servir pour générer des float aléatoires dans d'autres intervalles, par exemple 5.0 <= f < 10.0.

```
    fmt.Print((rand.Float64()*5)+5, ",")
    fmt.Print((rand.Float64() * 5) + 5)
    fmt.Println()
```

Le générateur de nombres par défaut est déterministe, donc il produit la même séquence de nombres à chaque fois par défaut. Pour produire des séquences variées, il faut lui fournir une donnée qui change. À noter que ce n'est pas une technique sûre pour les nombres aléatoires qui doivent être secrets : utilisez crypto/rand pour ceux-là.

```
    s1 := rand.NewSource(time.Now().UnixNano())
    r1 := rand.New(s1)
```

On appelle les méthodes de l'instance de rand.Rand obtenue comme les autres méthodes du package.

```
    fmt.Print(r1.Intn(100), ",")
    fmt.Print(r1.Intn(100))
    fmt.Println()
```

Si on fournit deux fois la même donnée à deux générateurs, il produiront la même séquence de nombres aléatoires.

```
    s2 := rand.NewSource(42)
    r2 := rand.New(s2)
    fmt.Print(r2.Intn(100), ",")
    fmt.Print(r2.Intn(100))
    fmt.Println()
    s3 := rand.NewSource(42)
    r3 := rand.New(s3)
    fmt.Print(r3.Intn(100), ",")
```

```
    fmt.Print(r3.Intn(100))
}
```

Code complet

```
package main

import "time"
import "fmt"
import "math/rand"

func main() {


 fmt.Print(rand.Intn(100), ",")
    fmt.Print(rand.Intn(100))
    fmt.Println()

    fmt.Println(rand.Float64())


 fmt.Print((rand.Float64()*5)+5, ","

 fmt.Print((rand.Float64() * 5) + 5)
    fmt.Println()

    s1 :=
 rand.NewSource(time.Now().UnixNano(
    r1 := rand.New(s1)

    fmt.Print(r1.Intn(100), ",")
    fmt.Print(r1.Intn(100))
    fmt.Println()

    s2 := rand.NewSource(42)
    r2 := rand.New(s2)
    fmt.Print(r2.Intn(100), ",")
    fmt.Print(r2.Intn(100))
    fmt.Println()
    s3 := rand.NewSource(42)
    r3 := rand.New(s3)
    fmt.Print(r3.Intn(100), ",")
    fmt.Print(r3.Intn(100))
}
```

Exécuter le code en ligne

```
$ go run random-numbers.go
81,87
0.6645600532184904
7.123187485356329,8.434115364335547
0,28
5,87
5,87
```

Regardez la documentation du package **math/rand** pour une référence concernant les autres nombres aléatoires que propose Go.

LIII - Extraction de nombres

Parser des nombres à partir d'une chaîne est une tâche simple, mais courante dans de nombreux programmes. Voici comment le faire en Go.

Le package `strconv` de la bibliothèque standard fournit ces fonctionnalités.

```
import "strconv"
import "fmt"

func main() {
```

Avec ParseFloat, le 64 indique le nombre de bits de précision à utiliser.

```
    f, _ := strconv.ParseFloat("1.234", 64)
    fmt.Println(f)
```

Pour ParseInt, le 0 signifie qu'on cherche à déduire la base à partir de la chaîne. 64 demande à faire rentrer le résultat dans 64 bits.

```
    //

    i, _ := strconv.ParseInt("123", 0, 64)
    fmt.Println(i)
```

ParseInt reconnaît les nombres formatés en hexadécimal.

```
    d, _ := strconv.ParseInt("0x1c8", 0, 64)
    fmt.Println(d)
```

Une fonction ParseUint est également disponible.

```
    u, _ := strconv.ParseUint("789", 0, 64)
    fmt.Println(u)
```

Atoi est une fonction pratique pour extraire facilement un entier en base 10.

```
    k, _ := strconv.Atoi("135")
    fmt.Println(k)
```

Les fonctions de parsing renvoient une erreur lorsque les données d'entrée ne conviennent pas.

```
    _, e := strconv.Atoi("wat")
    fmt.Println(e)
}
```

Code complet

```
package main

import "strconv"
import "fmt"

func main() {

    f, _ :=
 strconv.ParseFloat("1.234", 64)
    fmt.Println(f)

    //

    i, _ :=
 strconv.ParseInt("123", 0, 64)
    fmt.Println(i)

    d, _ :=
 strconv.ParseInt("0x1c8", 0, 64)
    fmt.Println(d)

    u, _ :=
 strconv.ParseUint("789", 0, 64)
    fmt.Println(u)

    k, _ := strconv.Atoi("135")
    fmt.Println(k)
```

Exécuter le code en ligne

Code complet

```
    _, e := strconv.Atoi("wat")
    fmt.Println(e)
}
```

```
$ go run number-parsing.go
1.234
123
456
789
135
strconv.ParseInt: parsing "wat": invalid syntax
```

Ensuite nous allons regarder une autre tâche de parsing courante : les URL.

LIV - Analyse d'URL

Les URL fournissent une **manière uniforme de localiser des ressources**. Voici comment analyser les URL en Go.

Nous allons analyser cette URL d'exemple, qui inclut un protocole (*scheme* en anglais), des informations d'authentification, un hôte, un port, un chemin, des paramètres de requêtes et un fragment de requête.

```
func main() {

    s := "postgres://user:pass@host.com:5432/path?k=v#f"
```

On parse l'URL et on s'assure qu'il n'y a pas d'erreur.

```
    u, err := url.Parse(s)
    if err != nil {
        panic(err)
    }
```

On accède facilement au protocole.

```
    fmt.Println(u.Scheme)
```

User contient toutes les informations d'authentification ; on appelle Username et Password dessus pour les valeurs individuelles.

```
    fmt.Println(u.User)
    fmt.Println(u.User.Username())
    p, _ := u.User.Password()
    fmt.Println(p)
```

Hostcontient à la fois le nom d'hôte et le port s'il est présent. Il faut utiliser SplitHostPort pour les extraire.

```
    fmt.Println(u.Host)
    host, port, _ := net.SplitHostPort(u.Host)
    fmt.Println(host)
    fmt.Println(port)
```

Ici on extrait le chemin (path) et le fragment (ce qui est après le #).

```
    fmt.Println(u.Path)
    fmt.Println(u.Fragment)
```

Pour obtenir les paramètres de requête dans une chaîne au format k=v, on utilise RawQuery. On peut aussi extraire les paramètres de requête dans une **map**. Les maps de paramètres de requêtes ont comme clés des strings et comme valeurs des slices de strings, donc il faut accéder à l'index 0 si on veut seulement la première valeur.

```
    fmt.Println(u.RawQuery)
    m, _ := url.ParseQuery(u.RawQuery)
    fmt.Println(m)
    fmt.Println(m["k"][0])
}
```

Code complet

```
package main

import "fmt"
import "net"
import "net/url"

func main() {

    s := "postgres://
user:pass@host.com:5432/path?
k=v#f"

    u, err := url.Parse(s)
    if err != nil {
        panic(err)
    }

    fmt.Println(u.Scheme)

    fmt.Println(u.User)

 fmt.Println(u.User.Username())
    p, _ := u.User.Password()
    fmt.Println(p)

    fmt.Println(u.Host)
    host, port, _ :=
 net.SplitHostPort(u.Host)
    fmt.Println(host)
    fmt.Println(port)

    fmt.Println(u.Path)
    fmt.Println(u.Fragment)

    fmt.Println(u.RawQuery)
    m, _ :=
 url.ParseQuery(u.RawQuery)
    fmt.Println(m)
    fmt.Println(m["k"][0])
}
```

Exécuter le code en ligne

Notre programme d'analyse d'URL montre les différents morceaux que nous avons extraits.

```
$ go run url-parsing.go
postgres
user:pass
user
pass
host.com:5432
host.com
5432
/path
f
k=v
map[k:[v]]
v
```

LV - Empreinte SHA1

Les **empreintes SHA1** sont fréquemment utilisées pour calculer de courtes chaînes permettant d'identifier des chaînes ou des gros objets binaires. Par exemple, le **système de gestion de versions git system** utilise SHA1 pour identifier les fichiers et répertoires versionnés. Voici comment calculer des empreintes SHA1 en Go.

Go implémente plusieurs fonctions de hachage dans plusieurs packages crypto/*.

```
import "crypto/sha1"
import "fmt"

func main() {
    s := "sha1 this string"
```

Le modèle pour générer une empreinte, c'est sha1.New(), sha1.Write(bytes), puis sha1.Sum([]byte{}). Ici, on commence avec un nouveau hachage.

```
    h := sha1.New()
```

Write attend un tableau d'octets (bytes). Si vous avez une chaîne, utilisez []byte(s) pour forcer le type.

```
    h.Write([]byte(s))
```

Ceci récupère le résultat du hachage dans une slice d'octets. L'argument à Sum peut être ajouté à une slice d'octets existante. En général, ce n'est pas nécessaire.

```
    bs := h.Sum(nil)
```

On affiche souvent les valeurs SHA-1 en hexadécimal, par exemple pour les commits Git. Utilisez le formatage avec %x pour convertir un résultat de hachage sous forme de chaîne hexadécimale.

```
    fmt.Println(s)
    fmt.Printf("%x\n", bs)
}
```

Code complet

```
package main

import "crypto/sha1"
import "fmt"

func main() {
    s := "sha1 this string"

    h := sha1.New()

    h.Write([]byte(s))

    bs := h.Sum(nil)

    fmt.Println(s)
    fmt.Printf("%x\n", bs)
}
```

Exécuter le code en ligne

Le programme calcule l'empreinte et l'affiche de manière lisible en hexadécimal.

```
$ go run sha1-hashes.go
sha1 this string
cf23df2207d99a74fbe169e3eba035e633b65d94
```

Vous pouvez calculer des empreintes de la même manière qu'ici avec d'autres méthodes de hachage. Par exemple, pour utiliser MD5, il faut importer crypto/md5 et utiliser md5.New().

Si vous avez besoin d'empreintes sûres pour de la cryptographie, vous devriez vous renseigner au sujet de la **force des différents algorithmes** !

LVI - Encodage Base64

Go propose un support natif pour l'encodage et le décodage en **base64**.

Cette syntaxe importe le package encoding/base64 avec le nom b64 au lieu du nom par défaut base64. Cela nous fait gagner un peu de place en dessous.

```
import b64 "encoding/base64"
import "fmt"

func main() {
```

Voici la chaîne que l'on va encoder et décoder.

```
    data := "abc123!?$*&()'-=@~"
```

Go supporte à la fois le format base64 standard et celui compatible avec les URL. Voici comment encoder avec l'encodeur standard. Il a besoin d'un []byte donc on va faire un cast de notre string vers ce type.

```
    sEnc := b64.StdEncoding.EncodeToString([]byte(data))
    fmt.Println(sEnc)
```

Le décodage peut renvoyer une erreur, que l'on peut vérifier si vous ne savez pas déjà si l'entrée est correctement formée.

```
    sDec, _ := b64.StdEncoding.DecodeString(sEnc)
    fmt.Println(string(sDec))
    fmt.Println()
```

Ceci encode et décode selon le format base64 compatible avec les URL.

```
    uEnc := b64.URLEncoding.EncodeToString([]byte(data))
    fmt.Println(uEnc)
    uDec, _ := b64.URLEncoding.DecodeString(uEnc)
    fmt.Println(string(uDec))
}
```

Code complet

```
package main

import b64 "encoding/base64"
import "fmt"

func main() {

    data := "abc123!?$*&()'-=@~"

    sEnc :=
 b64.StdEncoding.EncodeToString([]by
    fmt.Println(sEnc)

    sDec, _ :=
 b64.StdEncoding.DecodeString(sEnc)
    fmt.Println(string(sDec))
```

Exécuter le code en ligne

Code complet

```
    fmt.Println()

    uEnc :=
  b64.URLEncoding.EncodeToString([]by
    fmt.Println(uEnc)
    uDec, _ :=
  b64.URLEncoding.DecodeString(uEnc)
    fmt.Println(string(uDec))
}
```

La chaîne est encodée un peu différemment avec les deux encodeurs (`+` à la fin au lieu de `-`), mais elles se décodent vers la chaîne de départ comme désiré.

```
$ go run base64-encoding.go
YWJjMTIzIT8kKiYoKSctPUB+
abc123!?$*&()'-=@~

YWJjMTIzIT8kKiYoKSctPUB-
abc123!?$*&()'-=@~
```

LVII - Lire des fichiers

La lecture et l'écriture dans les fichiers sont des tâches de base nécessaires dans de nombreux programmes en Go. Nous allons tout d'abord regarder comment lire des fichiers.

La lecture de fichiers nécessite de vérifier les erreurs pour la plupart des appels. Cette fonction va nous permettre de simplifier la gestion d'erreurs par la suite.

```
func check(e error) {
    if e != nil {
        panic(e)
    }
}
```

Peut-être une des tâches de lecture les plus simples : extraire le contenu entier d'un fichier en mémoire.

```
func main() {
    dat, err := ioutil.ReadFile("/tmp/dat")
    check(err)
    fmt.Print(string(dat))
```

On veut souvent plus de contrôle sur comment et quelles parties d'un fichier sont lues. Pour ces tâches, on commence par ouvrir le fichier avec `Open` pour obtenir un `os.File` que l'on va manipuler.

```
    f, err := os.Open("/tmp/dat")
    check(err)
```

Le code ci-dessous lit quelques octets au début du fichier. il permet d'en lire jusqu'à cinq, mais regardez combien sont effectivement lus.

```
    b1 := make([]byte, 5)
    n1, err := f.Read(b1)
    check(err)
    fmt.Printf("%d bytes: %s\n", n1, string(b1))
```

On peut aussi se déplacer avec `Seek` à une position connue dans le fichier, et lire à partir de là avec `Read`.

```
    o2, err := f.Seek(6, 0)
    check(err)
    b2 := make([]byte, 2)
```

```
    n2, err := f.Read(b2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n2, o2, string(b2))
```

Le package io fournit des fonctions qui peuvent être utiles pour la lecture de fichiers. Par exemple, les lectures comme celles ci-dessus peuvent être réalisées de manière plus robuste avec ReadAtLeast.

```
    o3, err := f.Seek(6, 0)
    check(err)
    b3 := make([]byte, 2)
    n3, err := io.ReadAtLeast(f, b3, 2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n3, o3, string(b3))
```

Il n'y a pas de fonction de retour au début telle que rewind, mais on peut le faire avec Seek(0, 0).

```
    _, err = f.Seek(0, 0)
    check(err)
```

Le package bufio implémente un lecteur avec buffer qui peut être utile à la fois pour ses performances sur les petites lectures et par les méthodes additionnelles qu'il propose.

```
    r4 := bufio.NewReader(f)
    b4, err := r4.Peek(5)
    check(err)
    fmt.Printf("5 bytes: %s\n", string(b4))
```

On ferme le fichier quand on a terminé. En général, on le programme automatiquement après l'ouverture avec **defer**.

```
    f.Close()

}
```

Code complet

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "os"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {

    dat, err :=
 ioutil.ReadFile("/tmp/dat")
    check(err)
    fmt.Print(string(dat))

    f, err := os.Open("/tmp/
dat")
    check(err)

    b1 := make([]byte, 5)
    n1, err := f.Read(b1)
    check(err)
```

Exécuter le code en ligne

Code complet

```
    fmt.Printf("%d bytes: %s\n",
n1, string(b1))

    o2, err := f.Seek(6, 0)
    check(err)
    b2 := make([]byte, 2)
    n2, err := f.Read(b2)
    check(err)
    fmt.Printf("%d bytes @ %d:
%s\n", n2, o2, string(b2))

    o3, err := f.Seek(6, 0)
    check(err)
    b3 := make([]byte, 2)
    n3, err := io.ReadAtLeast(f,
b3, 2)
    check(err)
    fmt.Printf("%d bytes @ %d:
%s\n", n3, o3, string(b3))

    _, err = f.Seek(0, 0)
    check(err)

    r4 := bufio.NewReader(f)
    b4, err := r4.Peek(5)
    check(err)
    fmt.Printf("5 bytes:
%s\n", string(b4))

    f.Close()

}
```

```
$ echo "hello" > /tmp/dat
$ echo "go" >>   /tmp/dat
$ go run reading-files.go
hello
go
5 bytes: hello
2 bytes @ 6: go
2 bytes @ 6: go
5 bytes: hello
```

Maintenant on va aborder l'écriture de fichiers.

LVIII - Écrire dans des fichiers

Écrire dans des fichiers en Go se fait de manière similaire à ce que nous avons vu pour la lecture.

Pour commencer, voici comment écrire une chaîne (ou simplement des octets) dans un fichier.

```
    d1 := []byte("hello\ngo\n")
    err := ioutil.WriteFile("/tmp/dat1", d1, 0644)
    check(err)
```

Pour des écritures plus granulaires, on ouvre un fichier pour y écrire.

```
    f, err := os.Create("/tmp/dat2")
    check(err)
```

Il est idiomatique de reporter la fermeture avec `Close` immédiatement après l'ouverture avec **`defer`**.

```
    defer f.Close()
```

On peut écrire des slices d'octets comme on s'y attend avec Write.

```
    d2 := []byte{115, 111, 109, 101, 10}
    n2, err := f.Write(d2)
    check(err)
    fmt.Printf("wrote %d bytes\n", n2)
```

WriteString est également disponible.

```
    n3, err := f.WriteString("writes\n")
    fmt.Printf("wrote %d bytes\n", n3)
```

Appelez Sync pour forcer l'écriture.

```
    f.Sync()
```

bufio fournit des writers avec buffer en complément des readers que nous avons vus plus tôt.

```
    w := bufio.NewWriter(f)
    n4, err := w.WriteString("buffered\n")
    fmt.Printf("wrote %d bytes\n", n4)
```

Utilisez Flush pour vous assurer que toutes les opérations avec buffer ont été appliquées au writer sous-jacent.

```
    w.Flush()

}
```

Code complet

```
package main

import (
    "bufio"
    "fmt"
    "io/ioutil"
    "os"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {

    d1 := []byte("hello\ngo\n")
    err := ioutil.WriteFile("/
tmp/dat1", d1, 0644)
    check(err)

    f, err := os.Create("/tmp/
dat2")
    check(err)

    defer f.Close()


 d2 := []byte{115, 111, 109, 101, 10
    n2, err := f.Write(d2)
    check(err)
    fmt.Printf("wrote %d
 bytes\n", n2)
```

Exécuter le code en ligne

Code complet

```
    n3, err :=
f.WriteString("writes\n")
    fmt.Printf("wrote %d
bytes\n", n3)

    f.Sync()

    w := bufio.NewWriter(f)
    n4, err :=
w.WriteString("buffered\n")
    fmt.Printf("wrote %d
bytes\n", n4)

    w.Flush()

}
```

Essayez d'exécuter le code d'écriture dans les fichiers.

```
$ go run writing-files.go
wrote 5 bytes
wrote 7 bytes
wrote 9 bytes
```

Puis vérifiez le contenu des fichiers écrits.

```
$ cat /tmp/dat1
hello
go
$ cat /tmp/dat2
some
writes
buffered
```

Ensuite, nous verrons comment appliquer certaines idées que nous avons vues pour l'I/O sur les fichiers au flux stdin et stdout.

LIX - Filtres de ligne

Un *filtre de ligne* est un type de programme courant qui lit des entrées sur stdin, les traite, puis affiche un résultat dérivé sur stdout. grep et sed sont des exemples de filtres de ligne.

Voici un filtre de ligne en Go qui met en majuscules tout le texte d'entrée. Vous pouvez utiliser ce modèle pour écrire vos filtres de lignes en Go.

On encadre os.Stdin qui n'a pas de buffer, avec un scanner qui en a un. Il nous fournit en outre une méthode pratique Scan qui avance la lecture jusqu'à la marque suivante, c'est-à-dire ici la ligne suivante dans le scanner par défaut.

```
    scanner := bufio.NewScanner(os.Stdin)
```

Text renvoie la zone de texte courante, ici une ligne du texte d'entrée.

```
    for scanner.Scan() {

        ucl := strings.ToUpper(scanner.Text())
```

On affiche la ligne en majuscules.

```
        fmt.Println(ucl)
    }
```

On teste les erreurs durant notre Scan. Un caractère de fin de fichier est attendu et si l'on n'en trouve pas, il est reporté par Scan comme une erreur.

```
    if err := scanner.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "error:", err)
        os.Exit(1)
    }
}
```

Code complet

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

func main() {

    scanner :=
 bufio.NewScanner(os.Stdin)

    for scanner.Scan() {

        ucl :=
 strings.ToUpper(scanner.Text())

        fmt.Println(ucl)
    }

    if err := scanner.Err();
 err != nil {

 fmt.Fprintln(os.Stderr, "error:",
 err)
        os.Exit(1)
    }
}
```

Exécuter le code en ligne

Pour essayer notre filtre de ligne, commencez par créer un fichier avec quelques lignes en minuscules.

```
$ echo 'hello'   > /tmp/lines
$ echo 'filter' >> /tmp/lines
```

Puis utilisez le filtre de ligne pour les mettre en majuscules.

```
$ cat /tmp/lines | go run line-filters.go
HELLO
FILTER
```

LX - Arguments de ligne de commande

Les **arguments de ligne de commande** sont une manière courante de paramétrer l'exécution des programmes. Par exemple, go run hello.go envoie les arguments run et hello.go au programme go.

os.Args donne accès aux arguments bruts de la ligne de commande. À noter que la première valeur de cette slice est le chemin du programme, et os.Args[1:] contient les arguments du programme.

```
    argsWithProg := os.Args
    argsWithoutProg := os.Args[1:]
```

On peut accéder aux arguments individuels par leur indice.

```
    arg := os.Args[3]
    fmt.Println(argsWithProg)
    fmt.Println(argsWithoutProg)
    fmt.Println(arg)
}
```

Code complet

```
package main

import "os"
import "fmt"

func main() {

    argsWithProg := os.Args
    argsWithoutProg :=
 os.Args[1:]

    arg := os.Args[3]

    fmt.Println(argsWithProg)
    fmt.Println(argsWithoutProg)
    fmt.Println(arg)
}
```

Exécuter le code en ligne

Pour expérimenter avec les arguments de ligne de commande, il vaut mieux d'abord compiler un exécutable avec `go build`.

```
$ go build command-line-arguments.go
$ ./command-line-arguments a b c d
[./command-line-arguments a b c d]
[a b c d]
c
```

Ensuite, nous regarderons des techniques plus avancées de traitement avec les flags.

LXI - Options de ligne de commande

Les options de **ligne de commande** sont une manière courante de préciser des options pour des programmes en ligne de commande. Par exemple, dans `wc -l`, le `-l` est une option.

Go fournit un package `flag` qui permet d'analyser des options simples. Nous utiliserons ce package pour implémenter notre programme d'exemple.

```
import "flag"
import "fmt"

func main() {
```

Des déclarations d'options simples sont disponibles pour les chaînes de caractères, les entiers et les booléens. Ici on déclare une option `word` avec comme valeur par défaut **"foo"**, et une courte description. La fonction `flag.String` renvoie un pointeur sur la chaîne (pas une chaîne directement). Nous verrons comment utiliser ce pointeur après.

```
    wordPtr := flag.String("word", "foo", "une chaine")
```

On déclare des options `numb` et `fork`, en utilisant une approche similaire.

```
    numbPtr := flag.Int("numb", 42, "un int")
    boolPtr := flag.Bool("fork", false, "un booleen")
```

Il est également possible de déclarer l'option en utilisant une variable déjà déclarée ailleurs dans le programme. À noter que l'on doit passer un pointeur à la fonction de déclaration d'option.

```
    var svar string
    flag.StringVar(&svar, "svar", "bar", "a string var")
```

Une fois que toutes les options sont déclarées, il faut appeler flag.Parse() pour exécuter l'analyse des paramètres de ligne de commande.

```
    flag.Parse()
```

Ici, on va afficher les options analysées ainsi que les éventuels paramètres restants. À noter qu'il faut déréférencer les pointeurs, par ex. avec *wordPtr pour obtenir les valeurs réelles des options.

```
    fmt.Println("word:", *wordPtr)
    fmt.Println("numb:", *numbPtr)
    fmt.Println("fork:", *boolPtr)
    fmt.Println("svar:", svar)
    fmt.Println("tail:", flag.Args())
}
```

Code complet

```
package main

import "flag"
import "fmt"

func main() {

    wordPtr :=
flag.String("word", "foo", "une
chaine")

    numbPtr :=
flag.Int("numb", 42, "un int")
    boolPtr :=
flag.Bool("fork", false, "un
booleen")

    var svar string

flag.StringVar(&svar, "svar", "bar"
string var")

    flag.Parse()


fmt.Println("word:", *wordPtr)

fmt.Println("numb:", *numbPtr)

fmt.Println("fork:", *boolPtr)
    fmt.Println("svar:", svar)
    fmt.Println("tail:",
flag.Args())
}
```

Exécuter le code en ligne

Pour expérimenter avec les options du programme en ligne de commande, il est préférable de le compiler :

```
#ligne de commande, le plus simple est de le compiler
```

et d'exécuter le binaire ensuite.

```
$ go build command-line-flags.go
```

Essayez le programme en donnant tout d'abord des valeurs pour toutes les options.

```
$ ./command-line-flags -word=opt -numb=7 -fork -svar=flag
word: opt
numb: 7
fork: true
svar: flag
tail: []
```

Notez que si vous ne précisez pas certaines options, elles prennent automatiquement leurs valeurs par défaut.

```
$ ./command-line-flags -word=opt
word: opt
numb: 42
fork: false
svar: bar
tail: []
```

Des arguments restants peuvent être fournis après toutes les options.

```
$ ./command-line-flags -word=opt a1 a2 a3
word: opt
...
tail: [a1 a2 a3]
```

Notez que le package flag a besoin que toutes les options apparaissent avant les arguments « positionnels ». Sinon, les options sont traitées comme des arguments positionnels.

```
$ ./command-line-flags -word=opt a1 a2 a3 -numb=7
word: opt
numb: 42
fork: false
svar: bar
tail: [a1 a2 a3 -numb=7]
```

Utilisez les options -h ou --help pour obtenir une aide générée automatiquement au sujet des paramètres de ligne de commande.

```
$ ./command-line-flags -h
Usage of ./command-line-flags:
  -fork=false: a bool
  -numb=42: an int
  -svar="bar": a string var
  -word="foo": a string
```

Si vous fournissez une option qui n'était pas précisée au package flag, le programme affichera un message d'erreur puis le texte d'aide à nouveau.

```
$ ./command-line-flags -wat
flag provided but not defined: -wat
Usage of ./command-line-flags:
...
```

Nous allons ensuite aborder les variables d'environnement, une autre manière courante de paramétrer les programmes.

LXII - Variables d'environnement

Les **variables d'environnement** sont un mécanisme universel pour fournir des **informations de configuration à des programmes Unix**. Regardons comment créer, récupérer et lister des variables d'environnement.

Pour créer une paire clé/valeur, on utilise os.Setenv. Pour récupérer une valeur à partir d'une clé, il y a os.Getenv. Elle renvoie une chaîne vide si la clé n'est pas présente dans l'environnement.

```
func main() {
    os.Setenv("FOO", "1")
    fmt.Println("FOO:", os.Getenv("FOO"))
    fmt.Println("BAR:", os.Getenv("BAR"))
```

os.Environ permet de lister toutes les paires clé/valeur de variables d'environnement. Elle renvoie une slice de chaînes de caractères sous la forme CLE=valeur. On peut obtenir les clés et valeurs avec strings.Split. Ici, on affiche toutes les clés.

```
    fmt.Println()
    for _, e := range os.Environ() {
        pair := strings.Split(e, "=")
        fmt.Println(pair[0])
    }
}
```

Code complet

```
package main

import "os"
import "strings"
import "fmt"

func main() {

    os.Setenv("FOO", "1")
    fmt.Println("FOO:",
 os.Getenv("FOO"))
    fmt.Println("BAR:",
 os.Getenv("BAR"))

    fmt.Println()
    for _, e := range
 os.Environ() {
        pair :=
 strings.Split(e, "=")
        fmt.Println(pair[0])
    }
}
```

Exécuter le code en ligne

Le programme montre que nous choisissons une valeur pour FOO, mais que BAR est vide.

```
$ go run environment-variables.go
FOO: 1
BAR:
```

La liste des clés de l'environnement dépendra de votre machine.

```
TERM_PROGRAM
PATH
SHELL
...
```

Si nous affectons BAR dans l'environnement avant de lancer le programme, il récupérera sa valeur.

```
$ BAR=2 go run environment-variables.go
FOO: 1
BAR: 2
...
```

LXIII - Lancer des processus

Parfois nos programmes Go doivent lancer d'autres processus non Go. Par exemple, la mise en forme de syntaxe de ce site est implémentée par un processus **pygmentize** lancé depuis un programme Go. Regardons quelques exemples de lancement de programmes depuis Go.

Nous allons commencer par une simple commande qui ne prend ni arguments ni entrée, et qui affiche simplement quelque chose sur la sortie standard. La fonction helper exec.Command crée un objet pour représenter ce processus externe.

```
    dateCmd := exec.Command("date")
```

.Output est un autre helper qui gère les cas courants d'exécution de commande, en attendant qu'elle se termine et en récupérant sa sortie. S'il n'y a pas eu d'erreur, dateOut contiendra les octets avec les informations de date.

```
    //
    dateOut, err := dateCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> date")
    fmt.Println(string(dateOut))
```

Ensuite nous allons regarder un exemple un peu plus complexe, où l'on transfère des données au processus externe sur son entrée standard stdin, et collecte les résultats de sa sortie stdout.

```
    grepCmd := exec.Command("grep", "hello")
```

Ici, on récupère explicitement les canaux d'entrée/sortie, démarre le processus, envoie des données d'entrée, lit la sortie correspondante, et finalement attend que le processus se termine.

```
    grepIn, _ := grepCmd.StdinPipe()
    grepOut, _ := grepCmd.StdoutPipe()
    grepCmd.Start()
    grepIn.Write([]byte("hello grep\ngoodbye grep"))
    grepIn.Close()
    grepBytes, _ := ioutil.ReadAll(grepOut)
    grepCmd.Wait()
```

Nous n'avons pas fait les contrôles d'erreurs dans les exemples ci-dessus, mais vous pouvez utiliser le motif habituel **if** err **!= nil** pour chacun d'entre eux. De plus, on ne collecte les résultats qu'à travers StdoutPipe, mais on pourrait utiliser la sortie d'erreur (stderr) de la même manière via StderrPipe.

```
    fmt.Println("> grep hello")
    fmt.Println(string(grepBytes))
```

À noter que lorsqu'on lance des commandes, nous devons fournir une commande et un tableau d'arguments explicitement délimités (et non une unique chaîne comme sur la ligne de commande). Si vous voulez lancer une commande avec une chaîne, vous pouvez utiliser l'option -c de bash :

```
    lsCmd := exec.Command("bash", "-c", "ls -a -l -h")
    lsOut, err := lsCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> ls -a -l -h")
    fmt.Println(string(lsOut))
}
```

Code complet

```
package main

import "fmt"
import "io/ioutil"
import "os/exec"

func main() {

    dateCmd :=
 exec.Command("date")

    //

    dateOut, err :=
 dateCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> date")
    fmt.Println(string(dateOut))

    grepCmd :=
 exec.Command("grep", "hello")

    grepIn, _ :=
 grepCmd.StdinPipe()
    grepOut, _ :=
 grepCmd.StdoutPipe()
    grepCmd.Start()
    grepIn.Write([]byte("hello
 grep\ngoodbye grep"))
    grepIn.Close()
    grepBytes, _ :=
 ioutil.ReadAll(grepOut)
    grepCmd.Wait()

    fmt.Println("> grep hello")

 fmt.Println(string(grepBytes))

    lsCmd :=
 exec.Command("bash", "-c", "ls
 -a -l -h")
    lsOut, err := lsCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> ls -a -l -h")
    fmt.Println(string(lsOut))
}
```

Exécuter le code en ligne

Les programmes lancés retournent la même sortie, comme si nous les avions lancés depuis la ligne de commande.

```
$ go run spawning-processes.go
> date
Wed Oct 10 09:53:11 PDT 2012

> grep hello
hello grep
> ls -a -l -h
drwxr-xr-x  4 mark 136B Oct 3 16:29 .
drwxr-xr-x 91 mark 3.0K Oct 3 12:50 ..
-rw-r--r--  1 mark 1.3K Oct 3 16:28 spawning-processes.go
```

LXIV - Exécuter des processus

Dans l'exemple précédent, nous avions regardé comment **lancer des processus externes**. On fait ça quand on a besoin d'un processus externe accessible à un processus Go qui tourne. Parfois, on veut complètement remplacer le processus Go en cours par un autre (peut-être non Go). Pour cela, nous allons utiliser l'implémentation Go de la fonction classique **exec**.

Pour cet exemple, nous allons lancer ls. Go a besoin d'un chemin absolu vers l'exécutable que l'on veut lancer, donc on va utiliser exec.LookPath pour le localiser (il est probablement situé dans /bin/ls).

```
    binary, lookErr := exec.LookPath("ls")
    if lookErr != nil {
        panic(lookErr)
    }
```

Exec a besoin d'une slice pour les arguments (et non une grosse chaîne). On va donner à ls quelques arguments courants. À noter que le premier argument doit être le nom du programme.

```
    args := []string{"ls", "-a", "-l", "-h"}
```

Exec a également besoin d'un ensemble de **variables d'environnement**. Ici, on va lui fournir notre environnement actuel.

```
    env := os.Environ()
```

Voici enfin l'appel à syscall.Exec. Si l'appel réussit, l'exécution de notre processus va s'arrêter et sera remplacé par celui de /bin/ls -a -l -h. S'il y a une erreur, nous aurons une valeur de retour.

```
    execErr := syscall.Exec(binary, args, env)
    if execErr != nil {
        panic(execErr)
    }
}
```

Code complet

```
package main

import "syscall"
import "os"
import "os/exec"

func main() {

    binary, lookErr :=
 exec.LookPath("ls")
    if lookErr != nil {
        panic(lookErr)
    }

    args := []string{"ls", "-
a", "-l", "-h"}

    env := os.Environ()

    execErr :=
 syscall.Exec(binary, args, env)
    if execErr != nil {
        panic(execErr)
    }
}
```

Exécuter le code en ligne

Quand on lance notre programme, il est remplacé par ls.

```
$ go run execing-processes.go
total 16
drwxr-xr-x  4 mark 136B Oct 3 16:29 .
drwxr-xr-x 91 mark 3.0K Oct 3 12:50 ..
-rw-r--r--  1 mark 1.3K Oct 3 16:28 execing-processes.go
```

Notez que Go ne propose pas la fonction Unix classique fork. En général, ce n'est pas un problème, car lancer des goroutines, des processus, et exécuter des processus couvre la plupart des cas d'usage de fork.

LXV - Signaux

Parfois, on voudrait que les programmes en Go gèrent intelligemment les **signaux Unix**. Par exemple, on peut vouloir qu'un serveur s'éteigne gracieusement lorsqu'il reçoit le signal SIGTERM, ou qu'un outil en ligne de commande arrête de traiter les entrées lorsqu'il reçoit un SIGINT. Voici comment gérer les signaux en Go avec des canaux.

Go signale les travaux de notification en envoyant des valeurs os.Signal sur un canal. Nous allons créer un canal pour recevoir ces notifications. Nous allons aussi en faire un pour nous notifier lorsque le programme se termine.

```
    sigs := make(chan os.Signal, 1)
    done := make(chan bool, 1)
```

signal.Notify enregistre le canal fourni pour recevoir les notifications sur les signaux précisés.

```
    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
```

Cette goroutine exécute une réception bloquante des signaux. Quand elle se terminera, elle l'affichera et notifiera au programme qu'il peut prendre fin.

```
    go func() {
        sig := <-sigs
        fmt.Println()
        fmt.Println(sig)
        done <- true
    }()
```

Le programme va attendre ici jusqu'à ce qu'il reçoive le signal attendu (comme indiqué par la goroutine au-dessus, qui envoie une valeur sur done), puis se terminera.

```
    fmt.Println("awaiting signal")
    <-done
    fmt.Println("exiting")
}
```

Code complet

```
package main

import "fmt"
import "os"
import "os/signal"
import "syscall"

func main() {

    sigs := make(chan
 os.Signal, 1)
    done := make(chan bool, 1)

    signal.Notify(sigs,
 syscall.SIGINT,
 syscall.SIGTERM)
```

Exécuter le code en ligne

Code complet
```
    go func() {
        sig := <-sigs
        fmt.Println()
        fmt.Println(sig)
        done <- true
    }()

    fmt.Println("awaiting
 signal")
    <-done
    fmt.Println("exiting")
}
```

Quand on lance ce programme, il va bloquer en attendant un signal. En faisant ctrl-C (que le terminal montre comme ^C), on peut envoyer un signal SIGINT, qui amène le programme à afficher interrupt et à prendre fin.

```
$ go run signals.go
awaiting signal
^C
interrupt
exiting
```

LXVI - Sortie

On utilise os.Exit pour quitter immédiatement avec un statut donné.

Les **defer** ne seront *pas* exécutés lorsque l'on utilise os.Exit, donc ce fmt.Println ne sera jamais appelé.

```
    defer fmt.Println("!")
```

On quitte avec un statut 3.

```
    os.Exit(3)
}
```

Contrairement au C par exemple, Go n'utilise pas un entier comme valeur de retour de main pour indiquer le statut de sortie. Si vous voulez indiquer une valeur de sortie non nulle il faut utiliser os.Exit.

Code complet
```
package main

import "fmt"
import "os"

func main() {

    defer fmt.Println("!")

    os.Exit(3)
}
```

Exécuter le code en ligne

Si vous lancez exit.go en utilisant go run, le code de statut sera récupéré et affiché par go.

```
$ go run exit.go
exit status 3
```

En compilant et en exécutant le binaire, vous pouvez voir le statut dans le terminal.

```
$ go build exit.go
$ ./exit
```

```
$ echo $?
3
```

Notez que le! de notre programme n'a jamais été affiché.

LXVII - Remerciements

Nous tenons à remercier **Winjerome** pour la mise au gabarit et **Claude Leloup** pour la relecture orthographique.