eBook Gratuit

APPRENEZ Go

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#go

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: go

It is an unofficial and free Go ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Go.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec Go

Remarques

Go est un open-source, compilé, langage typé statiquement dans la tradition de Algol et C. Il possède des caractéristiques telles que la collecte des ordures, le typage structurel limité, les caractéristiques de sécurité de la mémoire, et facile à utiliser CSP programmation concurrente de style.

Versions

La dernière version majeure est en gras ci-dessous. L'historique complet des versions peut être trouvé ici .

Version	Date de sortie
1.8.3	2017-05-24
1.8.0	2017-02-16
1.7.0	2016-08-15
1.6.0	2016-02-17
1.5.0	2015-08-19
1.4.0	2014-12-04
1.3.0	2014-06-18
1.2.0	2013-12-01
1.1.0	2013-05-13
1.0.0	2012-03-28

Examples

Bonjour le monde!

Placez le code suivant dans un nom de fichier `hello.go` :

```
package main

import "fmt"
```

```
func main() {
    fmt.Println("Hello, □□")
}
```

Cour de récréation

Lorsque `Go` est installé correctement, ce programme peut être compilé et exécuté comme ceci:

```
go run hello.go
```

Sortie:

```
Hello, □□
```

Une fois que vous êtes satisfait du code, vous pouvez le compiler en exécutant:

```
go build hello.go
```

Cela créera un fichier exécutable adapté à votre système d'exploitation dans le répertoire en cours, que vous pourrez ensuite exécuter avec la commande suivante:

Linux, OSX et autres systèmes de type Unix

```
./hello
```

les fenêtres

```
hello.exe
```

***Remarque** : Les caractères chinois sont importants car ils démontrent que les chaînes Go sont stockées en tant que tranches d'octets en lecture seule.*

FizzBuzz

Un autre exemple de programme de style "Hello World" est FizzBuzz . Voici un exemple d'implémentation FizzBuzz. Très idiomatique Allez en jeu ici.

```
package main

// Simple fizzbuzz implementation

import "fmt"

func main() {
    for i := 1; i <= 100; i++ {
        s := ""
        if i % 3 == 0 {
```

```
            s += "Fizz"
        }
        if i % 5 == 0 {
            s += "Buzz"
        }
        if s != "" {
            fmt.Println(s)
        } else {
            fmt.Println(i)
        }
    }
}
```

Cour de récréation

Liste des variables d'environnement Go

Les variables d'environnement qui affectent l'outil `go` peuvent être visualisées via la commande `go env [var ...]` :

```
$ go env
GOARCH="amd64"
GOBIN="/home/yourname/bin"
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/home/yourname"
GORACE=""
GOROOT="/usr/lib/go"
GOTOOLDIR="/usr/lib/go/pkg/tool/linux_amd64"
CC="gcc"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/tmp/go-
build059426571=/tmp/go-build -gno-record-gcc-switches"
CXX="g++"
CGO_ENABLED="1"
```

Par défaut, il imprime la liste en tant que script shell; cependant, si un ou plusieurs noms de variables sont donnés en arguments, il imprime la valeur de chaque variable nommée.

```
$go env GOOS GOPATH
linux
/home/yourname
```

Mise en place de l'environnement

Si Go n'est pas pré-installé sur votre système, vous pouvez aller sur https://golang.org/dl/ et choisir votre plateforme pour télécharger et installer Go.

Pour configurer un environnement de développement Go de base, seules quelques-unes des nombreuses variables d'environnement affectant le comportement de l'outil `go` (voir: Liste des variables d'environnement Go pour une liste complète) doivent être définies (généralement dans le fichier `~/.profile` votre shell `~/.profile` fichier, ou équivalent sur les systèmes d'exploitation de

type Unix).

GOPATH

Comme la variable d'environnement `PATH` du système, Go path est une liste de répertoires délimités par `:` (`;` sur Windows) où Go recherche les packages. Le `go get` outil télécharger les paquets dans le premier répertoire dans cette liste.

`GOPATH` est l'endroit où Go va installer les dossiers `bin` , `pkg` et `src` associés nécessaires pour l'espace de travail:

- `src` - emplacement des fichiers sources: `.go` , `.c` , `.g` , `.s`
- `pkg` - a compilé des fichiers `.a`
- `bin` - contient les fichiers exécutables construits par Go

A partir de Go 1.8, la variable d'environnement `GOPATH` aura une valeur par défaut si elle n'est pas définie. La valeur par défaut est $ HOME / aller sous Unix / Linux et% USERPROFILE% / aller sous Windows.

Certains outils supposent que `GOPATH` contiendra un seul répertoire.

GOBIN

Le répertoire bin où `go install` et `go get` va placer des fichiers binaires après la construction des paquets `main` . Généralement, cela est défini quelque part sur le système `PATH` afin que les fichiers binaires installés puissent être exécutés et découverts facilement.

GOROOT

C'est l'emplacement de votre installation Go. Il est utilisé pour trouver les bibliothèques standard. Il est très rare de devoir définir cette variable, car Go intègre le chemin de génération dans la chaîne d'outils. La configuration de `GOROOT` est nécessaire si le répertoire d'installation diffère du répertoire de construction (ou de la valeur définie lors de la création).

Accès à la documentation hors ligne

Pour une documentation complète, exécutez la commande:

```
godoc -http=:<port-number>
```

Pour un tour de Go (hautement recommandé pour les débutants dans la langue):

```
go tool tour
```

Les deux commandes ci-dessus lancent des serveurs Web avec une documentation similaire à celle trouvée en ligne ici et ici respectivement.

Pour une vérification rapide des références depuis la ligne de commande, par exemple pour fmt.Print:

```
godoc cmd/fmt Print
# or
go doc fmt Print
```

Une aide générale est également disponible en ligne de commande:

```
go help [command]
```

Courir Aller en ligne

Le terrain de jeu Go

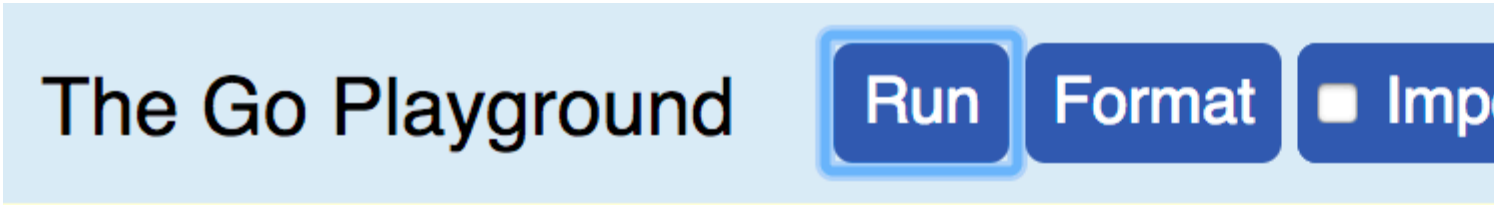
Un des outils les moins connus de Go est **The Go Playground** . Si l'on veut expérimenter Go sans le télécharger, ils peuvent facilement le faire simplement. . .

1. Visiter le terrain de jeu dans leur navigateur Web
2. Entrer leur code
3. En cliquant sur "Exécuter"

Partager votre code

The Go Playground dispose également d’outils de partage; Si un utilisateur appuie sur le bouton «Partager», un lien (comme celui-ci) sera généré et pourra être envoyé à d'autres personnes pour le tester et le modifier.

En action




```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func main() {
8          fmt.Println("Hello, playgroun
9  }
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

Chapitre 2: Analyse de fichiers CSV

Syntaxe

- csvReader: = csv.NewReader (r)
- data, err: = csvReader.Read ()

Examples

Analyse CSV simple

Considérez ces données CSV:

```
#id,title,text
1,hello world,"This is a ""blog""."
2,second time,"My
second
entry."
```

Ces données peuvent être lues avec le code suivant:

```
// r can be any io.Reader, including a file.
csvReader := csv.NewReader(r)
// Set comment character to '#'.
csvReader.Comment = '#'
for {
    post, err := csvReader.Read()
    if err != nil {
        log.Println(err)
        // Will break on EOF.
        break
    }
    fmt.Printf("post with id %s is titled %q: %q\n", post[0], post[1], post[2])
}
```

Et produire:

```
post with id 1 is titled "hello world": "This is a \"blog\"."
post with id 2 is titled "second time": "My\nsecond\nentry."
2009/11/10 23:00:00 EOF
```

Terrain de jeux: https://play.golang.org/p/d2E6-CGGIe .

Lire Analyse de fichiers CSV en ligne: https://riptutorial.com/fr/go/topic/5818/analyse-de-fichiers-csv

Chapitre 3: Analyse des arguments et des drapeaux de ligne de commande

Examples

Arguments de ligne de commande

L'argument d'analyse de ligne de commande est Go est très similaire aux autres langages. Dans votre code, vous accédez simplement à la partie des arguments où le premier argument sera le nom du programme lui-même.

Exemple rapide:

```
package main

import (
    "fmt"
    "os"
)

func main() {

    progName := os.Args[0]
    arguments := os.Args[1:]

    fmt.Printf("Here we have program '%s' launched with following flags: ", progName)

    for _, arg := range arguments {
        fmt.Printf("%s ", arg)
    }

    fmt.Println("")
}
```

Et la sortie serait:

```
$ ./cmd test_arg1 test_arg2
Here we have program './cmd' launched with following flags: test_arg1 test_arg2
```

Chaque argument est juste une chaîne. Dans le paquetage `os` , il ressemble à `var Args []string` : `var Args []string`

Les drapeaux

Aller bibliothèque standard fournit un `flag` package qui aide à analyser les indicateurs transmis au programme.

Notez que le package d' `flag` ne fournit pas d' `flag` habituels de style GNU. Cela signifie que les indicateurs à plusieurs lettres doivent être lancés avec un tiret unique comme ceci: `-exampleflag` ,

pas ceci: `--exampleflag` . Les drapeaux de type GNU peuvent être utilisés avec un paquetage tiers.

```
package main

import (
    "flag"
    "fmt"
)

func main() {

    // basic flag can be defined like this:
    stringFlag := flag.String("string.flag", "default value", "here comes usage")
    // after that stringFlag variable will become a pointer to flag value

    // if you need to store value in variable, not pointer, than you can
    // do it like:
    var intFlag int
    flag.IntVar(&intFlag, "int.flag", 1, "usage of intFlag")

    // after all flag definitions you must call
    flag.Parse()

    // then we can access our values
    fmt.Printf("Value of stringFlag is: %s\n", *stringFlag)
    fmt.Printf("Value of intFlag is: %d\n", intFlag)

}
```

`flag` aide le message pour nous:

```
$ ./flags -h
Usage of ./flags:
  -int.flag int
        usage of intFlag (default 1)
  -string.flag string
        here comes usage (default "default value")
```

Appel avec tous les drapeaux:

```
$ ./flags -string.flag test -int.flag 24
Value of stringFlag is: test
Value of intFlag is: 24
```

Appel avec des drapeaux manquants:

```
$ ./flags
Value of stringFlag is: default value
Value of intFlag is: 1
```

Lire Analyse des arguments et des drapeaux de ligne de commande en ligne: https://riptutorial.com/fr/go/topic/4023/analyse-des-arguments-et-des-drapeaux-de-ligne-de-commande

Chapitre 4: Boucles

Introduction

En tant que l'une des fonctions les plus élémentaires de la programmation, les boucles constituent une partie importante de presque tous les langages de programmation. Les boucles permettent aux développeurs de définir certaines parties de leur code à répéter à travers un certain nombre de boucles appelées itérations. Cette rubrique traite de l'utilisation de plusieurs types de boucles et d'applications de boucles dans Go.

Examples

Boucle de base

`for` est la seule instruction de boucle en cours, donc une implémentation de boucle de base pourrait ressembler à ceci:

```
// like if, for doesn't use parens either.
// variables declared in for and if are local to their scope.
for x := 0; x < 3; x++ { // ++ is a statement.
    fmt.Println("iteration", x)
}

// would print:
// iteration 0
// iteration 1
// iteration 2
```

Pause et Continuation

Sortir de la boucle et continuer à la prochaine itération est également pris en charge dans Go, comme dans de nombreuses autres langues:

```
for x := 0; x < 10; x++ { // loop through 0 to 9
    if x < 3 { // skips all the numbers before 3
        continue
    }
    if x > 5 { // breaks out of the loop once x == 6
        break
    }
    fmt.Println("iteration", x)
}

// would print:
// iteration 3
// iteration 4
// iteration 5
```

Les instructions `break` et `continue` acceptent en outre une étiquette facultative, utilisée pour

identifier les boucles externes à cibler avec l'instruction:

```
OuterLoop:
for {
    for {
        if allDone() {
            break OuterLoop
        }
        if innerDone() {
            continue OuterLoop
        }
        // do something
    }
}
```

Boucle conditionnelle

Le `for` mot - clé est également utilisé pour les boucles conditionnelles, traditionnellement `while` boucles dans d' autres langages de programmation.

```
package main

import (
    "fmt"
)

func main() {
    i := 0
    for i < 3 { // Will repeat if condition is true
        i++
        fmt.Println(i)
    }
}
```

jouer sur le terrain de jeu

Va sortir:

```
1
2
3
```

boucle infinie:

```
for {
    // This will run until a return or break.
}
```

Différentes formes de For Loop

Forme simple utilisant une variable:

```
for i := 0; i < 10; i++ {
```

```
    fmt.Print(i, " ")
}
```

En utilisant deux variables (ou plus):

```
for i, j := 0, 0; i < 5 && j < 10; i, j = i+1, j+2 {
    fmt.Println(i, j)
}
```

Sans utiliser l'instruction d'initialisation:

```
i := 0
for ; i < 10; i++ {
    fmt.Print(i, " ")
}
```

Sans expression de test:

```
for i := 1; ; i++ {
    if i&1 == 1 {
        continue
    }
    if i == 22 {
        break
    }
    fmt.Print(i, " ")
}
```

Sans expression d'incrément:

```
for i := 0; i < 10; {
    fmt.Print(i, " ")
    i++
}
```

Lorsque toutes les trois expressions d'initialisation, de test et d'incrémentation sont supprimées, la boucle devient infinie:

```
i := 0
for {
    fmt.Print(i, " ")
    i++
    if i == 10 {
        break
    }
}
```

Voici un exemple de boucle infinie avec compteur initialisé à zéro:

```
for i := 0; ; {
    fmt.Print(i, " ")
    if i == 9 {
        break
```

```
    }
    i++
}
```

Lorsque seule l'expression de test est utilisée (agit comme une boucle while typique):

```
i := 0
for i < 10 {
    fmt.Print(i, " ")
    i++
}
```

En utilisant uniquement l'expression d'incrémentation:

```
i := 0
for ; ; i++ {
    fmt.Print(i, " ")
    if i == 9 {
        break
    }
}
```

Itérer sur une plage de valeurs en utilisant un index et une valeur:

```
ary := [5]int{0, 1, 2, 3, 4}
for index, value := range ary {
    fmt.Println("ary[", index, "] =", value)
}
```

Itérer sur une plage en utilisant juste l'index:

```
for index := range ary {
    fmt.Println("ary[", index, "] =", ary[index])
}
```

Itérer sur une plage en utilisant juste l'index:

```
for index, _ := range ary {
    fmt.Println("ary[", index, "] =", ary[index])
}
```

Itérer sur une plage en utilisant simplement la valeur:

```
for _, value := range ary {
    fmt.Print(value, " ")
}
```

Itérer sur une plage en utilisant la clé et la valeur pour la carte (peut ne pas être dans l'ordre):

```
mp := map[string]int{"One": 1, "Two": 2, "Three": 3}
for key, value := range mp {
```

```
    fmt.Println("map[", key, "] =", value)
}
```

Itérer sur une plage en utilisant simplement la touche pour la carte (peut ne pas être dans l'ordre):

```
for key := range mp {
    fmt.Print(key, " ") //One Two Three
}
```

Itérer sur une plage en utilisant simplement la touche pour la carte (peut ne pas être dans l'ordre):

```
for key, _ := range mp {
    fmt.Print(key, " ") //One Two Three
}
```

Itérer sur une plage en utilisant juste la valeur pour la carte (peut ne pas être dans l'ordre):

```
for _, value := range mp {
    fmt.Print(value, " ") //2 3 1
}
```

Itérer sur une plage pour les canaux (quitte si le canal est fermé):

```
ch := make(chan int, 10)
for i := 0; i < 10; i++ {
    ch <- i
}
close(ch)

for i := range ch {
    fmt.Print(i, " ")
}
```

Itérer sur une plage pour la chaîne (donne les points de code Unicode):

```
utf8str := "B = \u00b5H" //B = µH
for _, r := range utf8str {
    fmt.Print(r, " ") //66 32 61 32 181 72
}
fmt.Println()
for _, v := range []byte(utf8str) {
    fmt.Print(v, " ") //66 32 61 32 194 181 72
}
fmt.Println(len(utf8str)) //7
```

comme vous le voyez, `utf8str` a 6 runes (points de code Unicode) et 7 octets.

Boucle chronométrée

```
package main
```

```
import(
    "fmt"
    "time"
)

func main() {
    for _ = range time.Tick(time.Second * 3) {
        fmt.Println("Ticking every 3 seconds")
    }
}
```

Lire Boucles en ligne: https://riptutorial.com/fr/go/topic/975/boucles

Chapitre 5: Brancher

Introduction

Go fournit un mécanisme de plug-in qui peut être utilisé pour lier dynamiquement un autre code Go lors de l'exécution.

A partir de Go 1.8, il n'est utilisable que sous Linux.

Examples

Définir et utiliser un plugin

```
package main

import "fmt"

var V int

func F() { fmt.Printf("Hello, number %d\n", V) }
```

Cela peut être construit avec:

```
go build -buildmode=plugin
```

Et puis chargé et utilisé depuis votre application:

```
p, err := plugin.Open("plugin_name.so")
if err != nil {
    panic(err)
}

v, err := p.Lookup("V")
if err != nil {
    panic(err)
}

f, err := p.Lookup("F")
if err != nil {
    panic(err)
}

*v.(*int) = 7
f.(func())() // prints "Hello, number 7"
```

Exemple tiré de *The State of Go 2017* .

Lire Brancher en ligne: https://riptutorial.com/fr/go/topic/9150/brancher

Chapitre 6: Canaux

Introduction

Un canal contient des valeurs d'un type donné. Les valeurs peuvent être écrites sur un canal et en être lues, et elles circulent à l'intérieur du canal dans l'ordre du premier entré, premier sorti. Il existe une distinction entre les canaux tamponnés, qui peuvent contenir plusieurs messages, et les canaux non tamponnés, ce qui est impossible. Les canaux sont généralement utilisés pour communiquer entre les goroutines, mais sont également utiles dans d'autres circonstances.

Syntaxe

- make (chan int) // crée un canal sans tampon
- make (chan int, 5) // crée un canal en mémoire tampon d'une capacité de 5
- close (ch) // ferme un canal "ch"
- ch <- 1 // écrit la valeur de 1 dans un canal "ch"
- val: = <-ch // lit une valeur du canal "ch"
- val, ok: = <-ch // syntaxe alternative; ok est un bool indiquant si le canal est fermé

Remarques

Un canal contenant la structure vide `make(chan struct{})` indique clairement à l'utilisateur qu'aucune information n'est transmise sur le canal et qu'elle est utilisée uniquement pour la synchronisation.

En ce qui concerne les canaux non tamponnés, une écriture de canal bloquera jusqu'à ce qu'une lecture correspondante se produise à partir d'une autre goroutine. La même chose est vraie pour un blocage de lecture de canal en attendant un écrivain.

Examples

Utiliser la gamme

Lors de la lecture de plusieurs valeurs d'un canal, l'utilisation de la `range` est un modèle courant:

```
func foo() chan int {
    ch := make(chan int)

    go func() {
        ch <- 1
        ch <- 2
        ch <- 3
        close(ch)

    }()
```

```
    return ch
}

func main() {
    for n := range foo() {
        fmt.Println(n)
    }

    fmt.Println("channel is now closed")
}
```

Cour de récréation

Sortie

```
1
2
3
channel is now closed
```

Des délais d'attente

Les canaux sont souvent utilisés pour implémenter les délais d'attente.

```
func main() {
    // Create a buffered channel to prevent a goroutine leak. The buffer
    // ensures that the goroutine below can eventually terminate, even if
    // the timeout is met. Without the buffer, the send on the channel
    // blocks forever, waiting for a read that will never happen, and the
    // goroutine is leaked.
    ch := make(chan struct{}, 1)

    go func() {
        time.Sleep(10 * time.Second)
        ch <- struct{}{}
    }()

    select {
    case <-ch:
        // Work completed before timeout.
    case <-time.After(1 * time.Second):
        // Work was not completed after 1 second.
    }
}
```

Coordination des goroutines

Imaginez une goroutine avec un processus en deux étapes, où le thread principal doit travailler entre chaque étape:

```
func main() {
    ch := make(chan struct{})
    go func() {
        // Wait for main thread's signal to begin step one
        <-ch
```

```
        // Perform work
        time.Sleep(1 * time.Second)

        // Signal to main thread that step one has completed
        ch <- struct{}{}

        // Wait for main thread's signal to begin step two
        <-ch

        // Perform work
        time.Sleep(1 * time.Second)

        // Signal to main thread that work has completed
        ch <- struct{}{}
    }()

    // Notify goroutine that step one can begin
    ch <- struct{}{}

    // Wait for notification from goroutine that step one has completed
    <-ch

    // Perform some work before we notify
    // the goroutine that step two can begin
    time.Sleep(1 * time.Second)

    // Notify goroutine that step two can begin
    ch <- struct{}{}

    // Wait for notification from goroutine that step two has completed
    <-ch
}
```

Tamponné vs non tamponné

```
func bufferedUnbufferedExample(buffered bool) {
    // We'll declare the channel, and we'll make it buffered or
    // unbuffered depending on the parameter `buffered` passed
    // to this function.
    var ch chan int
    if buffered {
        ch = make(chan int, 3)
    } else {
        ch = make(chan int)
    }

    // We'll start a goroutine, which will emulate a webserver
    // receiving tasks to do every 25ms.
    go func() {
        for i := 0; i < 7; i++ {
            // If the channel is buffered, then while there's an empty
            // "slot" in the channel, sending to it will not be a
            // blocking operation. If the channel is full, however, we'll
            // have to wait until a "slot" frees up.
            // If the channel is unbuffered, sending will block until
            // there's a receiver ready to take the value. This is great
            // for goroutine synchronization, not so much for queueing
            // tasks for instance in a webserver, as the request will
```

```
            // hang until the worker is ready to take our task.
            fmt.Println(">", "Sending", i, "...")
            ch <- i
            fmt.Println(">", i, "sent!")
            time.Sleep(25 * time.Millisecond)
        }
        // We'll close the channel, so that the range over channel
        // below can terminate.
        close(ch)
    }()

    for i := range ch {
        // For each task sent on the channel, we would perform some
        // task. In this case, we will assume the job is to
        // "sleep 100ms".
        fmt.Println("<", i, "received, performing 100ms job")
        time.Sleep(100 * time.Millisecond)
        fmt.Println("<", i, "job done")
    }
}
```

aller au terrain de jeux

Blocage et déblocage des canaux

Par défaut, la communication sur les canaux est synchronisée; Lorsque vous envoyez une valeur, il doit y avoir un récepteur. Sinon, vous aurez `fatal error: all goroutines are asleep - deadlock!` comme suit:

```
package main

import "fmt"

func main() {
    msg := make(chan string)
    msg <- "Hey There"
    go func() {
        fmt.Println(<-msg)
    }()
}
```

Il existe une solution: utiliser des canaux tamponnés:

```
package main

import "fmt"
import "time"

func main() {
    msg :=make(chan string, 1)
    msg <- "Hey There!"
    go func() {
        fmt.Println(<-msg)
    }()
    time.Sleep(time.Second * 1)
}
```

En attendant que le travail se termine

Une technique courante pour utiliser des canaux consiste à créer un certain nombre de travailleurs (ou de consommateurs) à lire depuis le canal. L'utilisation d'un sync.WaitGroup est un moyen simple d'attendre que ces travailleurs aient fini de fonctionner.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    numPiecesOfWork := 20
    numWorkers := 5

    workCh := make(chan int)
    wg := &sync.WaitGroup{}

    // Start workers
    wg.Add(numWorkers)
    for i := 0; i < numWorkers; i++ {
        go worker(workCh, wg)
    }

    // Send work
    for i := 0; i < numPiecesOfWork; i++ {
        work := i % 10 // invent some work
        workCh <- work
    }

    // Tell workers that no more work is coming
    close(workCh)

    // Wait for workers to finish
    wg.Wait()

    fmt.Println("done")
}

func worker(workCh <-chan int, wg *sync.WaitGroup) {
    defer wg.Done() // will call wg.Done() right before returning

    for work := range workCh { // will wait for work until workCh is closed
        doWork(work)
    }
}

func doWork(work int) {
    time.Sleep(time.Duration(work) * time.Millisecond)
    fmt.Println("slept for", work, "milliseconds")
}
```

Lire Canaux en ligne: https://riptutorial.com/fr/go/topic/1263/canaux

Chapitre 7: cgo

Examples

Cgo: tutoriel Premiers pas

Quelques exemples pour comprendre le flux de travail de l'utilisation des liaisons Go C

Quelle

Dans Go, vous pouvez appeler les programmes et fonctions C en utilisant cgo . De cette façon, vous pouvez facilement créer des liaisons C vers d'autres applications ou bibliothèques fournissant une API C.

Comment

Tout ce que vous avez à faire est d'ajouter un `import "C"` au début de votre programme Go **juste** après avoir inclus votre programme C:

```
//#include <stdio.h>
import "C"
```

Avec l'exemple précédent, vous pouvez utiliser le package `stdio` dans Go.

Si vous avez besoin d'utiliser une application qui se trouve dans votre même dossier, vous utilisez la même syntaxe que dans C (avec le `"` au lieu de `<>`)

```
//#include "hello.c"
import "C"
```

> **IMPORTANT** : Ne **laissez pas de nouvelle ligne entre les** instructions **`import "C"`** **`include`** **et** **`import "C"`** ou vous obtiendrez ce type d'erreurs lors de la génération:

```
# command-line-arguments
could not determine kind of name for C.Hello
could not determine kind of name for C.sum
```

L'exemple

Dans ce dossier, vous pouvez trouver un exemple de liaisons C. Nous avons deux bibliothèques "C" très simples appelées `hello.c` :

```
//hello.c
#include <stdio.h>
```

```
void Hello(){
    printf("Hello world\n");
}
```

Cela imprime simplement "hello world" dans la console et `sum.c`

```
//sum.c
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}
```

... qui prend 2 arguments et retourne sa somme (ne pas l'imprimer).

Nous avons un programme `main.go` qui utilisera ces deux fichiers. Nous les importons d'abord comme nous l'avons déjà mentionné:

```
//main.go
package main

/*
  #include "hello.c"
  #include "sum.c"
*/
import "C"
```

Bonjour le monde!

Nous sommes maintenant prêts à utiliser les programmes C dans notre application Go. Essayons d'abord le programme Hello:

```
//main.go
package main

/*
  #include "hello.c"
  #include "sum.c"
*/
import "C"


func main() {
    //Call to void function without params
    err := Hello()
    if err != nil {
        log.Fatal(err)
    }
}

//Hello is a C binding to the Hello World "C" program. As a Go user you could
//use now the Hello function transparently without knowing that it is calling
//a C function
func Hello() error {
    _, err := C.Hello()    //We ignore first result as it is a void function
```

```
    if err != nil {
        return errors.New("error calling Hello function: " + err.Error())
    }

    return nil
}
```

Maintenant, exécutez le programme main.go en utilisant `go run main.go` pour obtenir l'impression du programme C: "Hello world!". Bien joué!

Somme des ints

Rendons-le un peu plus complexe en ajoutant une fonction qui résume ses deux arguments.

```
//sum.c
#include <stdio.h>

int sum(int a, int b) {
  return a + b;
}
```

Et nous l'appellerons depuis notre précédente application Go.

```
//main.go
package main

/*
#include "hello.c"
#include "sum.c"
*/
import "C"

import (
    "errors"
    "fmt"
    "log"
)

func main() {
    //Call to void function without params
    err := Hello()
    if err != nil {
        log.Fatal(err)
    }

    //Call to int function with two params
    res, err := makeSum(5, 4)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("Sum of 5 + 4 is %d\n", res)
}

//Hello is a C binding to the Hello World "C" program. As a Go user you could
//use now the Hello function transparently without knowing that is calling a C
//function
```

```
func Hello() error {
    _, err := C.Hello() //We ignore first result as it is a void function
    if err != nil {
        return errors.New("error calling Hello function: " + err.Error())
    }

    return nil
}

//makeSum also is a C binding to make a sum. As before it returns a result and
//an error. Look that we had to pass the Int values to C.int values before using
//the function and cast the result back to a Go int value
func makeSum(a, b int) (int, error) {
    //Convert Go ints to C ints
    aC := C.int(a)
    bC := C.int(b)

    sum, err := C.sum(aC, bC)
    if err != nil {
        return 0, errors.New("error calling Sum function: " + err.Error())
    }

    //Convert C.int result to Go int
    res := int(sum)

    return res, nil
}
```

Regardez la fonction "makeSum". Il reçoit deux paramètres `int` qui doivent être convertis en C `int` avant en utilisant la fonction `C.int` . De plus, le retour de l'appel nous donnera un C `int` et une erreur en cas de problème. Nous devons convertir la réponse C en un int de Go en utilisant `int()` .

Essayez de lancer notre application go en utilisant `go run main.go`

```
$ go run main.go
Hello world!
Sum of 5 + 4 is 9
```

Générer un binaire

Si vous essayez une construction, vous pourriez avoir plusieurs erreurs de définition.

```
$ go build
# github.com/sayden/c-bindings
/tmp/go-build329491076/github.com/sayden/c-bindings/_obj/hello.o: In function `Hello':
../../go/src/github.com/sayden/c-bindings/hello.c:5: multiple definition of `Hello'
/tmp/go-build329491076/github.com/sayden/c-
bindings/_obj/main.cgo2.o:/home/mariocaster/go/src/github.com/sayden/c-bindings/hello.c:5:
first defined here
/tmp/go-build329491076/github.com/sayden/c-bindings/_obj/sum.o: In function `sum':
../../go/src/github.com/sayden/c-bindings/sum.c:5: multiple definition of `sum`
/tmp/go-build329491076/github.com/sayden/c-
bindings/_obj/main.cgo2.o:/home/mariocaster/go/src/github.com/sayden/c-bindings/sum.c:5: first
defined here
collect2: error: ld returned 1 exit status
```

L'astuce consiste à se référer directement au fichier principal lors de l'utilisation de `go build` :

```
$ go build main.go
$ ./main
Hello world!
Sum of 5 + 4 is 9
```

> Rappelez-vous que vous pouvez donner un nom au fichier binaire en utilisant `-o` flag `go build -o my_c_binding main.go`

J'espère que vous avez apprécié ce tutoriel.

Lire cgo en ligne: https://riptutorial.com/fr/go/topic/6125/cgo

Chapitre 8: cgo

Examples

Appel de la fonction C à partir de Go

Cgo permet la création de packages Go qui appellent du code C.
Pour utiliser `cgo` écrivez un code Go normal qui importe un pseudo-package "C". Le code Go peut alors faire référence à des types tels que `C.int` ou des fonctions telles que `C.Add` .
L'importation de "C" est immédiatement précédée d'un commentaire, ce commentaire, appelé préambule, est utilisé comme en-tête lors de la compilation des parties C du package.
Notez qu'il ne doit y avoir aucune ligne vide entre le commentaire `cgo` et l'instruction import.
Notez que l' `import "C"` ne peut pas être regroupée avec d'autres importations dans une instruction d'importation "factorisée" entre parenthèses. Vous devez écrire plusieurs instructions d'importation, comme:

```
import "C"
import "fmt"
```

Et c'est un bon style d'utiliser le relevé d'importation factorisé, pour d'autres importations, comme:

```
import "C"
import (
    "fmt"
    "math"
)
```

Exemple simple utilisant `cgo` :

```
package main

//int Add(int a, int b){
//    return a+b;
//}
import "C"
import "fmt"

func main() {
    a := C.int(10)
    b := C.int(20)
    c := C.Add(a, b)
    fmt.Println(c) // 30
}
```

Ensuite, `go build` et exécutez le résultat:

```
30
```

Pour construire des paquets `cgo` , utilisez simplement `go build` ou `go install` normalement. L' `go`

`tool` reconnaît l'importation spéciale `"C"` et utilise automatiquement `cgo` pour ces fichiers.

Fil C et Go dans toutes les directions

Appeler le code C de Go

```
package main

/*
// Everything in comments above the import "C" is C code and will be compiles with the GCC.
// Make sure you have a GCC installed.

int addInC(int a, int b) {
    return a + b;
}
 */
import "C"
import "fmt"

func main() {
       a := 3
       b := 5

       c := C.addInC(C.int(a), C.int(b))

       fmt.Println("Add in C:", a, "+", b, "=", int(c))
}
```

Code d'appel de C de C

```
package main

/*
static inline int multiplyInGo(int a, int b) {
    return go_multiply(a, b);
}
 */
import "C"
import (
       "fmt"
)

func main() {
       a := 3
       b := 5

       c := C.multiplyInGo(C.int(a), C.int(b))

       fmt.Println("multiplyInGo:", a, "*", b, "=", int(c))
}

//export go_multiply
func go_multiply(a C.int, b C.int) C.int {
       return a * b
}
```

Traiter les pointeurs de fonction

```
package main

/*
int go_multiply(int a, int b);

typedef int (*multiply_f)(int a, int b);
multiply_f multiply;

static inline init() {
    multiply = go_multiply;
}

static inline int multiplyWithFp(int a, int b) {
    return multiply(a, b);
}
 */
import "C"
import (
       "fmt"
)

func main() {
       a := 3
       b := 5
       C.init(); // OR:
       C.multiply = C.multiply_f(go_multiply);

       c := C.multiplyWithFp(C.int(a), C.int(b))

       fmt.Println("multiplyInGo:", a, "+", b, "=", int(c))
}

//export go_multiply
func go_multiply(a C.int, b C.int) C.int {
       return a * b
}
```

Types de conversion, structures d'accès et arithmétique de pointeur

De la documentation officielle Go:

```
// Go string to C string
// The C string is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CString(string) *C.char

// Go []byte slice to C array
// The C array is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CBytes([]byte) unsafe.Pointer

// C string to Go string
func C.GoString(*C.char) string

// C data with explicit length to Go string
func C.GoStringN(*C.char, C.int) string
```

```
// C data with explicit length to Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte
```

Comment l'utiliser:

```
func go_handleData(data *C.uint8_t, length C.uint8_t) []byte {
       return C.GoBytes(unsafe.Pointer(data), C.int(length))
}

// ...

goByteSlice := []byte {1, 2, 3}
goUnsafePointer := C.CBytes(goByteSlice)
cPointer := (*C.uint8_t)(goUnsafePointer)

// ...

func getPayload(packet *C.packet_t) []byte {
       dataPtr := unsafe.Pointer(packet.data)
       // Lets assume a 2 byte header before the payload.
       payload := C.GoBytes(unsafe.Pointer(uintptr(dataPtr)+2), C.int(packet.dataLength-2))
       return payload
}
```

Lire cgo en ligne: https://riptutorial.com/fr/go/topic/6455/cgo

Chapitre 9: Chaîne

Introduction

Une chaîne est en effet une tranche d'octets en lecture seule. Dans Go, un littéral de chaîne contiendra toujours une représentation UTF-8 valide de son contenu.

Syntaxe

- variableName: = "Hello World" // déclare une chaîne
- variableName: = `Hello World` // déclare une chaîne littérale brute
- variableName: = "Bonjour" + "Monde" // concatène les chaînes
- sous-chaîne: = "Hello World" [0: 4] // récupère une partie de la chaîne
- lettre: = "Hello World" [6] // obtient un caractère de la chaîne
- fmt.Sprintf ("% s", "Hello World") // formate une chaîne

Examples

Type de chaîne

Le type de `string` vous permet de stocker du texte, qui est une série de caractères. Il existe plusieurs façons de créer des chaînes. Une chaîne littérale est créée en écrivant le texte entre guillemets doubles.

```
text := "Hello World"
```

Parce que les chaînes Go prennent en charge UTF-8, l'exemple précédent est parfaitement valide. Les chaînes contiennent des octets arbitraires, ce qui ne signifie pas nécessairement que chaque chaîne contiendra un code UTF-8 valide, mais les chaînes de caractères contiendront toujours des séquences UTF-8 valides.

La valeur zéro des chaînes est une chaîne vide `""` .

Les chaînes peuvent être concaténées à l'aide de l'opérateur `+` .

```
text := "Hello " + "World"
```

Les chaînes peuvent également être définies en utilisant les backticks `` ` `` . Cela crée un littéral de chaîne brut, ce qui signifie que les caractères ne seront pas échappés.

```
text1 := "Hello\nWorld"
text2 := `Hello
World`
```

Dans l'exemple précédent, `text1` échappe au caractère `\n` qui représente une nouvelle ligne alors

que `text2` contient directement le nouveau caractère de ligne. Si vous comparez `text1 == text2` le résultat sera `true` .

Cependant, `text2 := `Hello\nWorld`` n'échapperait pas au caractère `\n` , ce qui signifie que la chaîne contient le texte `Hello\nWorld` sans nouvelle ligne. Ce serait l'équivalent de taper `text1 := "Hello\\nWorld"` .

Formatage du texte

Le package `fmt` implémente des fonctions pour imprimer et formater du texte à l'aide de *verbes de* format. Les verbes sont représentés avec un signe de pourcentage.

Verbes généraux:

```
%v    // the value in a default format
      // when printing structs, the plus flag (%+v) adds field names
%#v   // a Go-syntax representation of the value
%T    // a Go-syntax representation of the type of the value
%%    // a literal percent sign; consumes no value
```

Booléen:

```
%t    // the word true or false
```

Entier:

```
%b    // base 2
%c    // the character represented by the corresponding Unicode code point
%d    // base 10
%o    // base 8
%q    // a single-quoted character literal safely escaped with Go syntax.
%x    // base 16, with lower-case letters for a-f
%X    // base 16, with upper-case letters for A-F
%U    // Unicode format: U+1234; same as "U+%04X"
```

Composants à virgule flottante et complexes:

```
%b    // decimalless scientific notation with exponent a power of two,
      // in the manner of strconv.FormatFloat with the 'b' format,
      // e.g. -123456p-78
%e    // scientific notation, e.g. -1.234456e+78
%E    // scientific notation, e.g. -1.234456E+78
%f    // decimal point but no exponent, e.g. 123.456
%F    // synonym for %f
%g    // %e for large exponents, %f otherwise
%G    // %E for large exponents, %F otherwise
```

Chaîne et tranche d'octets (traitées de manière équivalente avec ces verbes):

```
%s    // the uninterpreted bytes of the string or slice
%q    // a double-quoted string safely escaped with Go syntax
%x    // base 16, lower-case, two characters per byte
```

```
%X    // base 16, upper-case, two characters per byte
```

Aiguille:

```
%p    // base 16 notation, with leading 0x
```

En utilisant les verbes, vous pouvez créer des chaînes concaténant plusieurs types:

```
text1 := fmt.Sprintf("Hello %s", "World")
text2 := fmt.Sprintf("%d + %d = %d", 2, 3, 5)
text3 := fmt.Sprintf("%s, %s (Age: %d)", "Obama", "Barack", 55)
```

La fonction `Sprintf` formate la chaîne dans le premier paramètre en remplaçant les verbes par la valeur des valeurs dans les paramètres suivants et renvoie le résultat. Comme `Sprintf` , la fonction `Printf` également mais au lieu de retourner le résultat, elle imprime la chaîne.

paquet de chaînes

- strings.Contains

  ```
  fmt.Println(strings.Contains("foobar", "foo")) // true
  fmt.Println(strings.Contains("foobar", "baz")) // false
  ```

- strings.HasPrefix

  ```
  fmt.Println(strings.HasPrefix("foobar", "foo")) // true
  fmt.Println(strings.HasPrefix("foobar", "baz")) // false
  ```

- strings.HasSuffix

  ```
  fmt.Println(strings.HasSuffix("foobar", "bar")) // true
  fmt.Println(strings.HasSuffix("foobar", "baz")) // false
  ```

- strings.Join

  ```
  ss := []string{"foo", "bar", "bar"}
  fmt.Println(strings.Join(ss, ", ")) // foo, bar, baz
  ```

- strings.Replace

  ```
  fmt.Println(strings.Replace("foobar", "bar", "baz", 1)) // foobaz
  ```

- strings.Split

  ```
  s := "foo, bar, bar"
  fmt.Println(strings.Split(s, ", ")) // [foo bar baz]
  ```

- strings.ToLower

  ```
  fmt.Println(strings.ToLower("FOOBAR")) // foobar
  ```

- `strings.ToUpper`

```
fmt.Println(strings.ToUpper("foobar")) // FOOBAR
```

- `strings.TrimSpace`

```
fmt.Println(strings.TrimSpace("  foobar  ")) // foobar
```

Plus: https://golang.org/pkg/strings/ .

Lire Chaîne en ligne: https://riptutorial.com/fr/go/topic/9666/chaine

Chapitre 10: Client HTTP

Syntaxe

- resp, err: = http.Get (url) // Crée une requête HTTP GET avec le client HTTP par défaut. Une erreur non nulle est renvoyée si la demande échoue.
- resp, err: = http.Post (url, bodyType, body) // Crée une requête HTTP POST avec le client HTTP par défaut. Une erreur non nulle est renvoyée si la demande échoue.
- resp, err: = http.PostForm (url, values) // Crée une requête HTTP POST avec le client HTTP par défaut. Une erreur non nulle est renvoyée si la demande échoue.

Paramètres

Paramètre	Détails
resp	Une réponse de type `*http.Response` à une requête HTTP
se tromper	Une `error` S'il n'est pas nul, il représente une erreur qui s'est produite lors de l'appel de la fonction.
URL	Une URL de type `string` pour faire une requête HTTP.
bodyType	Le type MIME de type `string` de la charge utile du corps d'une requête POST.
corps	Un `io.Reader` (implémente `Read()`) qui sera lu jusqu'à ce qu'une erreur soit atteinte pour être soumis en tant que charge utile du corps d'une requête POST.
valeurs	Une carte clé-valeur de type `url.Values` . Le type sous-jacent est une `map[string][]string` .

Remarques

Il est important de `defer resp.Body.Close()` après chaque requête HTTP ne `defer resp.Body.Close()` pas d'erreur non NULL, sinon les ressources seront fuites.

Examples

GET de base

Effectuez une requête GET de base et imprimez le contenu d'un site (HTML).

```
package main

import (
```

```
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    resp, err := http.Get("https://example.com/")
    if err != nil {
        panic(err)
    }

    // It is important to defer resp.Body.Close(), else resource leaks will occur.
    defer resp.Body.Close()

    data, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        panic(err)
    }

    // Will print site contents (HTML) to output
    fmt.Println(string(data))
}
```

GET avec des paramètres d'URL et une réponse JSON

Une demande pour les 10 publications les plus récentes de StackOverflow en utilisant l'API Stack Exchange.

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
)

const apiURL = "https://api.stackexchange.com/2.2/posts?"

// Structs for JSON decoding
type postItem struct {
    Score int    `json:"score"`
    Link  string `json:"link"`
}

type postsType struct {
    Items []postItem `json:"items"`
}

func main() {
    // Set URL parameters on declaration
    values := url.Values{
        "order": []string{"desc"},
        "sort":  []string{"activity"},
        "site":  []string{"stackoverflow"},
    }

    // URL parameters can also be programmatically set
    values.Set("page", "1")
```

```
    values.Set("pagesize", "10")

    resp, err := http.Get(apiURL + values.Encode())
    if err != nil {
        panic(err)
    }

    defer resp.Body.Close()

    // To compare status codes, you should always use the status constants
    // provided by the http package.
    if resp.StatusCode != http.StatusOK {
        panic("Request was not OK: " + resp.Status)
    }

    // Example of JSON decoding on a reader.
    dec := json.NewDecoder(resp.Body)
    var p postsType
    err = dec.Decode(&p)
    if err != nil {
        panic(err)
    }

    fmt.Println("Top 10 most recently active StackOverflow posts:")
    fmt.Println("Score", "Link")
    for _, post := range p.Items {
        fmt.Println(post.Score, post.Link)
    }
}
```

Demande de délai avec un contexte

1.7+

La temporisation d'une requête HTTP avec un contexte peut être accomplie avec uniquement la bibliothèque standard (pas la sous-position) dans 1.7+:

```
import (
    "context"
    "net/http"
    "time"
)

req, err := http.NewRequest("GET", `https://example.net`, nil)
ctx, _ := context.WithTimeout(context.TODO(), 200 * time.Milliseconds)
resp, err := http.DefaultClient.Do(req.WithContext(ctx))
// Be sure to handle errors.
defer resp.Body.Close()
```

Avant 1.7

```
import (
    "net/http"
    "time"
```

```
    "golang.org/x/net/context"
    "golang.org/x/net/context/ctxhttp"
)

ctx, err := context.WithTimeout(context.TODO(), 200 * time.Milliseconds)
resp, err := ctxhttp.Get(ctx, http.DefaultClient, "https://www.example.net")
// Be sure to handle errors.
defer resp.Body.Close()
```

Lectures complémentaires

Pour plus d'informations sur le package de `context` , voir Contexte .

Demande PUT d'objet JSON

Ce qui suit met à jour un objet User via une requête PUT et imprime le code d'état de la requête:

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "net/http"
)

type User struct {
    Name  string
    Email string
}

func main() {
    user := User{
        Name:  "John Doe",
        Email: "johndoe@example.com",
    }

    // initialize http client
    client := &http.Client{}

    // marshal User to json
    json, err := json.Marshal(user)
    if err != nil {
        panic(err)
    }

    // set the HTTP method, url, and request body
    req, err := http.NewRequest(http.MethodPut, "http://api.example.com/v1/user",
bytes.NewBuffer(json))
    if err != nil {
        panic(err)
    }

    // set the request header Content-Type for json
    req.Header.Set("Content-Type", "application/json; charset=utf-8")
    resp, err := client.Do(req)
    if err != nil {
```

```
        panic(err)
    }

    fmt.Println(resp.StatusCode)
}
```

Lire Client HTTP en ligne: https://riptutorial.com/fr/go/topic/1422/client-http

Chapitre 11: Compilation croisée

Introduction

Le compilateur Go peut produire des binaires pour de nombreuses plates-formes, c'est-à-dire des processeurs et des systèmes. Contrairement à la plupart des autres compilateurs, il n'ya pas d'exigence spécifique à la compilation croisée, elle est aussi simple à utiliser que la compilation normale.

Syntaxe

- GOOS = linux GOARCH = amd64 go build

Remarques

Combinaisons de système d'exploitation et de cible d'architecture prises en charge (source)

$ GOOS	$ GOARCH
Android	bras
Darwin	386
Darwin	amd64
Darwin	bras
Darwin	arm64
libellule	amd64
freebsd	386
freebsd	amd64
freebsd	bras
linux	386
linux	amd64
linux	bras
linux	arm64
linux	ppc64

$ GOOS	$ GOARCH
linux	ppc64le
linux	mips64
linux	mips64le
netbsd	386
netbsd	amd64
netbsd	bras
openbsd	386
openbsd	amd64
openbsd	bras
plan9	386
plan9	amd64
Solaris	amd64
les fenêtres	386
les fenêtres	amd64

Examples

Compiler toutes les architectures en utilisant un Makefile

Ce Makefile recoupera et compilera les exécutables pour Windows, Mac et Linux (ARM et x86).

```
# Replace demo with your desired executable name
appname := demo

sources := $(wildcard *.go)

build = GOOS=$(1) GOARCH=$(2) go build -o build/$(appname)$(3)
tar = cd build && tar -cvzf $(1)_$(2).tar.gz $(appname)$(3) && rm $(appname)$(3)
zip = cd build && zip $(1)_$(2).zip $(appname)$(3) && rm $(appname)$(3)

.PHONY: all windows darwin linux clean

all: windows darwin linux

clean:
    rm -rf build/

##### LINUX BUILDS #####
```

```
linux: build/linux_arm.tar.gz build/linux_arm64.tar.gz build/linux_386.tar.gz
build/linux_amd64.tar.gz

build/linux_386.tar.gz: $(sources)
    $(call build,linux,386,)
    $(call tar,linux,386)

build/linux_amd64.tar.gz: $(sources)
    $(call build,linux,amd64,)
    $(call tar,linux,amd64)

build/linux_arm.tar.gz: $(sources)
    $(call build,linux,arm,)
    $(call tar,linux,arm)

build/linux_arm64.tar.gz: $(sources)
    $(call build,linux,arm64,)
    $(call tar,linux,arm64)

##### DARWIN (MAC) BUILDS #####
darwin: build/darwin_amd64.tar.gz

build/darwin_amd64.tar.gz: $(sources)
    $(call build,darwin,amd64,)
    $(call tar,darwin,amd64)

##### WINDOWS BUILDS #####
windows: build/windows_386.zip build/windows_amd64.zip

build/windows_386.zip: $(sources)
    $(call build,windows,386,.exe)
    $(call zip,windows,386,.exe)

build/windows_amd64.zip: $(sources)
    $(call build,windows,amd64,.exe)
    $(call zip,windows,amd64,.exe)
```

(Faites attention à ce que le fichier Makefile nécessite des onglets durs et non des espaces)

Compilation croisée simple avec go build

À partir du répertoire de votre projet, exécutez la commande `go build` et spécifiez la cible du système d'exploitation et de l'architecture avec les variables d'environnement `GOOS` et `GOARCH` :

Compilation pour Mac (64 bits):

```
GOOS=darwin GOARCH=amd64 go build
```

Compilation pour Windows x86:

```
GOOS=windows GOARCH=386 go build
```

Vous pouvez également définir manuellement le nom du fichier exécutable de sortie pour suivre l'architecture:

```
GOOS=windows GOARCH=386 go build -o appname_win_x86.exe
```

A partir de la version 1.7 et suivantes, vous pouvez obtenir une liste de toutes les combinaisons GOOS et GOARCH possibles avec:

```
go tool dist list
```

(ou pour faciliter la consommation de la machine, `go tool dist list -json`)

Compilation croisée en utilisant gox

Une autre solution pratique pour la compilation croisée est l'utilisation de `gox` : https://github.com/mitchellh/gox

Installation

L'installation se fait très facilement en exécutant `go get github.com/mitchellh/gox` . L'exécutable résultant est placé dans le répertoire binaire de Go, par exemple `/golang/bin` ou `~/golang/bin` . Assurez-vous que ce dossier fait partie de votre chemin afin d'utiliser la commande `gox` partir d'un emplacement arbitraire.

Usage

Dans le dossier racine du projet Go (où vous exécutez par exemple la `go build`), exécutez `gox` afin de créer tous les binaires possibles pour toute architecture (par exemple x86, ARM) et système d'exploitation (par exemple Linux, macOS, Windows).

Pour construire pour un certain système d'exploitation, utilisez par exemple `gox -os="linux"` place. L'option d'architecture pourrait également être définie: `gox -osarch="linux/amd64"` .

Exemple simple: compiler helloworld.go pour une architecture de bras sur une machine Linux

Préparez helloworld.go (trouvez ci-dessous)

```
package main

import "fmt"

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

Exécuter `GOOS=linux GOARCH=arm go build helloworld.go`

Copiez le `helloworld` généré `helloworld` (arm executable) sur votre machine cible.

Lire Compilation croisée en ligne: https://riptutorial.com/fr/go/topic/1020/compilation-croisee

Chapitre 12: Concurrence

Introduction

Dans Go, la simultanéité est obtenue grâce à l'utilisation de goroutines, et la communication entre les goroutines se fait généralement avec des canaux. Cependant, d'autres moyens de synchronisation, tels que les mutex et les groupes d'attente, sont disponibles et doivent être utilisés chaque fois qu'ils sont plus pratiques que les canaux.

Syntaxe

- aller doWork () // exécuter la fonction doWork en tant que goroutine
- ch: = make (chan int) // déclare le nouveau canal de type int
- ch <- 1 // envoi sur un canal
- value = <-ch // recevoir d'un canal

Remarques

Goroutines in Go sont similaires aux threads dans d'autres langues en termes d'utilisation. En interne, Go crée un certain nombre de threads (spécifiés par `GOMAXPROCS`), puis planifie l'exécution des goroutines sur les threads. En raison de cette conception, les mécanismes de concurrence de Go sont beaucoup plus efficaces que les threads en termes d'utilisation de la mémoire et de temps d'initialisation.

Examples

Créer des goroutines

Toute fonction peut être appelée en tant que goroutine en préfixant son invocation avec le mot `go` clé `go` :

```
func DoMultiply(x,y int) {
    // Simulate some hard work
    time.Sleep(time.Second * 1)
    fmt.Printf("Result: %d\n", x * y)
}

go DoMultiply(1,2) // first execution, non-blocking
go DoMultiply(3,4) // second execution, also non-blocking

// Results are printed after a single second only,
// not 2 seconds because they execute concurrently:
// Result: 2
// Result: 12
```

Notez que la valeur de retour de la fonction est ignorée.

Bonjour tout le monde Goroutine

single channel, single goroutine, one write, one read.

```
package main

import "fmt"
import "time"

func main() {
    // create new channel of type string
    ch := make(chan string)

    // start new anonymous goroutine
    go func() {
        time.Sleep(time.Second)
        // send "Hello World" to channel
        ch <- "Hello World"
    }()
    // read from channel
    msg, ok := <-ch
    fmt.Printf("msg='%s', ok='%v'\n", msg, ok)
}
```

Exécuter sur le terrain de jeu

Le canal `ch` est un **canal non tamponné ou synchrone** .

Le `time.Sleep` est ici pour illustrer `main()` fonction `main()` qui **attendra** sur le canal `ch` , ce qui signifie que la fonction littérale exécutée en tant que goroutine a le temps d'envoyer une valeur via ce canal: l' opérateur de réception `<-ch` va bloquer `main()` . Si ce n'était pas le cas, le goroutine serait tué lorsque `main()` quitterait et n'aurait pas le temps d'envoyer sa valeur.

En attente de goroutines

Les programmes Go se terminent lorsque la fonction `main` se termine , il est donc courant d'attendre que toutes les goroutines se terminent. Une solution courante consiste à utiliser un objet sync.WaitGroup .

```
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup // 1

func routine(i int) {
    defer wg.Done() // 3
    fmt.Printf("routine %v finished\n", i)
}

func main() {
    wg.Add(10) // 2
```

```
    for i := 0; i < 10; i++ {
        go routine(i) // *
    }
    wg.Wait() // 4
    fmt.Println("main finished")
}
```

Exécuter l'exemple dans la cour de récréation

WaitGroup utilisation dans l'ordre d'exécution:

1. Déclaration de variable globale. Le rendre global est le moyen le plus simple de le rendre visible à toutes les fonctions et méthodes.
2. Augmenter le compteur Cela doit être fait dans la base de données principale car rien ne garantit qu'une goroutine nouvellement démarrée s'exécutera avant 4 en raison des garanties du modèle de mémoire.
3. Diminuer le compteur Cela doit être fait à la sortie d'une goroutine. En utilisant un appel différé, nous nous assurons qu'il sera appelé chaque fois que la fonction se terminera , peu importe comment elle se termine.
4. Attendre que le compteur atteigne 0. Cela doit être fait dans la liste principale pour empêcher le programme de sortir avant que toutes les goroutines soient terminées.

* Les paramètres sont évalués avant de lancer un nouveau goroutine . Il est donc nécessaire de définir explicitement leurs valeurs avant `wg.Add(10)` afin que le code éventuellement paniqué n'augmente pas le compteur. En ajoutant 10 éléments au WaitGroup, il faudra attendre 10 éléments avant que `wg.Wait` ramène le contrôle à `main()` goroutine. Ici, la valeur de i est définie dans la boucle for.

Utiliser des fermetures avec des goroutines en boucle

Lorsqu'elle est en boucle, la variable de boucle (val) dans l'exemple suivant est une variable unique qui change de valeur lorsqu'elle passe sur la boucle. Par conséquent, il faut faire ce qui suit pour que chaque valeur de valeur soit transmise au goroutine:

```
for val := range values {
    go func(val interface{}) {
        fmt.Println(val)
    }(val)
}
```

Si vous deviez faire juste aller `func(val interface{}) { ... }()` sans passer val, alors la valeur de `val` sera n'importe quel val est quand les goroutines fonctionne réellement.

Une autre façon d'obtenir le même effet est la suivante:

```
for val := range values {
    val := val
    go func() {
        fmt.Println(val)
    }()
}
```

Le `val := val` étrange `val := val` crée une nouvelle variable dans chaque itération, à laquelle le goroutine accède ensuite.

Arrêt des goroutines

```
package main

import (
    "log"
    "sync"
    "time"
)

func main() {
    // The WaitGroup lets the main goroutine wait for all other goroutines
    // to terminate. However, this is no implicit in Go. The WaitGroup must
    // be explicitely incremented prior to the execution of any goroutine
    // (i.e. before the `go` keyword) and it must be decremented by calling
    // wg.Done() at the end of every goroutine (typically via the `defer` keyword).
    wg := sync.WaitGroup{}

    // The stop channel is an unbuffered channel that is closed when the main
    // thread wants all other goroutines to terminate (there is no way to
    // interrupt another goroutine in Go). Each goroutine must multiplex its
    // work with the stop channel to guarantee liveness.
    stopCh := make(chan struct{})


    for i := 0; i < 5; i++ {
        // It is important that the WaitGroup is incremented before we start
        // the goroutine (and not within the goroutine) because the scheduler
        // makes no guarantee that the goroutine starts execution prior to
        // the main goroutine calling wg.Wait().
        wg.Add(1)
        go func(i int, stopCh <-chan struct{}) {
            // The defer keyword guarantees that the WaitGroup count is
            // decremented when the goroutine exits.
            defer wg.Done()

            log.Printf("started goroutine %d", i)

            select {
            // Since we never send empty structs on this channel we can
            // take the return of a receive on the channel to mean that the
            // channel has been closed (recall that receive never blocks on
            // closed channels).
            case <-stopCh:
                log.Printf("stopped goroutine %d", i)
            }
        }(i, stopCh)
    }

    time.Sleep(time.Second * 5)
    close(stopCh)
    log.Printf("stopping goroutines")
    wg.Wait()
    log.Printf("all goroutines stopped")
}
```

Ping-pong avec deux goroutines

```
package main

import (
    "fmt"
    "time"
)

// The pinger prints a ping and waits for a pong
func pinger(pinger <-chan int, ponger chan<- int) {
    for {
        <-pinger
        fmt.Println("ping")
        time.Sleep(time.Second)
        ponger <- 1
    }
}

// The ponger prints a pong and waits for a ping
func ponger(pinger chan<- int, ponger <-chan int) {
    for {
        <-ponger
        fmt.Println("pong")
        time.Sleep(time.Second)
        pinger <- 1
    }
}

func main() {
    ping := make(chan int)
    pong := make(chan int)

    go pinger(ping, pong)
    go ponger(ping, pong)

    // The main goroutine starts the ping/pong by sending into the ping channel
    ping <- 1

    for {
        // Block the main thread until an interrupt
        time.Sleep(time.Second)
    }
}
```

Exécuter une version légèrement modifiée de ce code dans Go Playground

Lire Concurrence en ligne: https://riptutorial.com/fr/go/topic/376/concurrence

Chapitre 13: Console I / O

Examples

Lire l'entrée depuis la console

Utiliser `scanf`

> Scanf scanne le texte lu depuis l'entrée standard, stockant les valeurs successives séparées par des espaces dans des arguments successifs, déterminés par le format. Il renvoie le nombre d'éléments analysés avec succès. Si cela est inférieur au nombre d'arguments, err indiquera pourquoi. Les nouvelles lignes dans l'entrée doivent correspondre aux nouvelles lignes du format. La seule exception: le verbe% c analyse toujours la prochaine rune dans l'entrée, même s'il s'agit d'un espace (ou d'un onglet, etc.) ou d'une nouvelle ligne.

```
# Read integer
var i int
fmt.Scanf("%d", &i)

# Read string
var str string
fmt.Scanf("%s", &str)
```

Utilisation de l' `scan`

> L'analyse analyse le texte lu à partir de l'entrée standard, stockant les valeurs successives séparées par des espaces dans des arguments successifs. Les nouvelles lignes comptent comme un espace. Il renvoie le nombre d'éléments analysés avec succès. Si cela est inférieur au nombre d'arguments, err indiquera pourquoi.

```
# Read integer
var i int
fmt.Scan(&i)

# Read string
var str string
fmt.Scan(&str)
```

Utiliser `scanln`

> Sscanln est similaire à Sscan, mais arrête l'analyse à la nouvelle ligne et après le dernier élément, il doit y avoir une nouvelle ligne ou EOF.

```
# Read string
var input string
fmt.Scanln(&input)
```

En utilisant `bufio`

```
# Read using Reader
reader := bufio.NewReader(os.Stdin)
text, err := reader.ReadString('\n')

# Read using Scanner
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
    fmt.Println(scanner.Text())
}
```

Lire Console I / O en ligne: https://riptutorial.com/fr/go/topic/9741/console-i---o

Chapitre 14: Construire des contraintes

Syntaxe

- // + construire des balises

Remarques

Les balises de construction sont utilisées pour construire de manière conditionnelle certains fichiers dans votre code. Les balises de génération peuvent ignorer les fichiers que vous ne voulez pas construire, sauf si elles sont explicitement incluses, ou certaines balises de génération prédéfinies peuvent être utilisées pour qu'un fichier ne soit construit que sur une architecture ou un système d'exploitation particulier.

Les balises de génération peuvent apparaître dans n'importe quel type de fichier source (pas seulement dans Go), mais elles doivent apparaître en haut du fichier, précédées uniquement de lignes vierges et d'autres commentaires de ligne. Ces règles signifient que, dans les fichiers Go, une contrainte de génération doit apparaître avant la clause du package.

Une série de balises de construction doit être suivie d'une ligne vide.

Examples

Tests d'intégration séparés

Les contraintes de construction sont généralement utilisées pour séparer les tests unitaires normaux des tests d'intégration nécessitant des ressources externes, telles qu'une base de données ou un accès réseau. Pour ce faire, ajoutez une contrainte de génération personnalisée en haut du fichier de test:

```
// +build integration

package main

import (
    "testing"
)

func TestThatRequiresNetworkAccess(t *testing.T) {
    t.Fatal("It failed!")
}
```

Le fichier de test ne sera pas compilé dans le fichier exécutable à moins que le `go test` appel suivant `go test` soit utilisé:

```
go test -tags "integration"
```

Résultats:

```
$ go test
?       bitbucket.org/yourname/yourproject    [no test files]
$ go test -tags "integration"
--- FAIL: TestThatRequiresNetworkAccess (0.00s)
        main_test.go:10: It failed!
FAIL
exit status 1
FAIL    bitbucket.org/yourname/yourproject    0.003s
```

Optimiser les implémentations basées sur l'architecture

Nous pouvons optimiser une fonction xor simple pour les architectures qui prennent en charge les lectures / écritures non alignées en créant deux fichiers qui définissent la fonction et les préfixent avec une contrainte de construction (pour un exemple réel du code xor qui est hors de portée ici, voir `crypto/cipher/xor.go` dans la bibliothèque standard):

```
// +build 386 amd64 s390x

package cipher

func xorBytes(dst, a, b []byte) int { /* This function uses unaligned reads / writes to
optimize the operation */ }
```

et pour d'autres architectures:

```
// +build !386,!amd64,!s390x

package cipher

func xorBytes(dst, a, b []byte) int { /* This version of the function just loops and xors */ }
```

Lire Construire des contraintes en ligne: https://riptutorial.com/fr/go/topic/2595/construire-des-contraintes

Chapitre 15: Conversions de type

Examples

Conversion de type de base

Il existe deux styles de base de conversion de type dans Go:

```
// Simple type conversion
var x := Foo{}    // x is of type Foo
var y := (Bar)Foo // y is of type Bar, unless Foo cannot be cast to Bar, then compile-time
error occurs.
// Extended type conversion
var z,ok := x.(Bar)    // z is of type Bar, ok is of type bool - if conversion succeeded, z
has the same value as x and ok is true. If it failed, z has the zero value of type Bar, and ok
is false.
```

Mise en œuvre de l'interface de test

Comme Go utilise l'implémentation d'interface implicite, vous n'obtiendrez pas d'erreur de compilation si votre structure n'implémente pas une interface que vous aviez l'intention d'implémenter. Vous pouvez tester l'implémentation explicitement à l'aide de la conversion de type: tapez MyInterface interface {Thing ()}

```
type MyImplementer struct {}

func (m MyImplementer) Thing() {
    fmt.Println("Huzzah!")
}

// Interface is implemented, no error. Variable name _ causes value to be ignored.
var _ MyInterface = (*MyImplementer)nil

type MyNonImplementer struct {}

// Compile-time error - cannot case because interface is not implemented.
var _ MyInterface = (*MyNonImplementer)nil
```

Implémenter un système d'unités avec des types

Cet exemple illustre comment le système de type Go peut être utilisé pour implémenter un système d'unités.

```
package main

import (
    "fmt"
)

type MetersPerSecond float64
```

```
type KilometersPerHour float64

func (mps MetersPerSecond) toKilometersPerHour() KilometersPerHour {
    return KilometersPerHour(mps * 3.6)
}

func (kmh KilometersPerHour) toMetersPerSecond() MetersPerSecond {
    return MetersPerSecond(kmh / 3.6)
}

func main() {
    var mps MetersPerSecond
    mps = 12.5
    kmh := mps.toKilometersPerHour()
    mps2 := kmh.toMetersPerSecond()
    fmt.Printf("%vmps = %vkmh = %vmps\n", mps, kmh, mps2)
}
```

Open in Playground

Lire Conversions de type en ligne: https://riptutorial.com/fr/go/topic/2851/conversions-de-type

Chapitre 16: Cryptographie

Introduction

Découvrez comment chiffrer et déchiffrer les données avec Go. Gardez à l'esprit que ce n'est pas un cours sur la cryptographie, mais plutôt comment y parvenir avec Go.

Examples

Cryptage et décryptage

Avant-propos

Ceci est un exemple détaillé sur la façon de chiffrer et déchiffrer les données avec Go. Le code d'utilisation est raccourci, par exemple la gestion des erreurs n'est pas mentionnée. Le projet de travail complet avec gestion des erreurs et interface utilisateur se trouve sur Github ici .

Cryptage

Introduction et données

Cet exemple décrit un chiffrement et un déchiffrement complets dans Go. Pour ce faire, nous avons besoin de données. Dans cet exemple, nous utilisons notre propre `secret` structure de données:

```
type secret struct {
    DisplayName       string
    Notes             string
    Username          string
    EMail             string
    CopyMethod        string
    Password          string
    CustomField01Name string
    CustomField01Data string
    CustomField02Name string
    CustomField02Data string
    CustomField03Name string
    CustomField03Data string
    CustomField04Name string
    CustomField04Data string
    CustomField05Name string
    CustomField05Data string
    CustomField06Name string
    CustomField06Data string
}
```

Ensuite, nous voulons chiffrer un tel `secret` . L'exemple de travail complet peut être trouvé ici (lien vers Github) . Maintenant, le processus pas à pas:

Étape 1

Tout d'abord, nous avons besoin d'une sorte de mot de passe maître pour protéger le secret:
`masterPassword := "PASS"`

Étape 2

Toutes les méthodes de cryptage utilisant des octets au lieu de chaînes. Ainsi, nous construisons un tableau d'octets avec les données de notre secret.

```
secretBytesDecrypted :=
[]byte(fmt.Sprintf("%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
    artifact.DisplayName,
    strings.Replace(artifact.Notes, "\n", string(65000), -1),
    artifact.Username,
    artifact.EMail,
    artifact.CopyMethod,
    artifact.Password,
    artifact.CustomField01Name,
    artifact.CustomField01Data,
    artifact.CustomField02Name,
    artifact.CustomField02Data,
    artifact.CustomField03Name,
    artifact.CustomField03Data,
    artifact.CustomField04Name,
    artifact.CustomField04Data,
    artifact.CustomField05Name,
    artifact.CustomField05Data,
    artifact.CustomField06Name,
    artifact.CustomField06Data,
))
```

Étape 3

Nous créons du sel afin de prévenir les attaques à l'arc-en-ciel, cf. Wikipedia : `saltBytes := uuid.NewV4().Bytes()` . Ici, nous utilisons un UUID v4 qui n'est pas prévisible.

Étape 4

Maintenant, nous sommes en mesure de dériver une clé et un vecteur du mot de passe maître et du sel aléatoire, en ce qui concerne RFC 2898:

```
keyLength := 256
rfc2898Iterations := 6

keyVectorData := pbkdf2.Key(masterPassword, saltBytes, rfc2898Iterations,
(keyLength/8)+aes.BlockSize, sha1.New)
keyBytes := keyVectorData[:keyLength/8]
```

```
vectorBytes := keyVectorData[keyLength/8:]
```

Étape 5

Le mode CBC souhaité fonctionne avec des blocs entiers. Nous devons donc vérifier si nos données sont alignées sur un bloc complet. Sinon, nous devons le remplir:

```
if len(secretBytesDecrypted)%aes.BlockSize != 0 {
    numberNecessaryBlocks := int(math.Ceil(float64(len(secretBytesDecrypted)) /
float64(aes.BlockSize)))
    enhanced := make([]byte, numberNecessaryBlocks*aes.BlockSize)
    copy(enhanced, secretBytesDecrypted)
    secretBytesDecrypted = enhanced
}
```

Étape 6

Maintenant, nous créons un chiffrement AES: `aesBlockEncrypter, aesErr := aes.NewCipher(keyBytes)`

Étape 7

Nous réservons la mémoire nécessaire aux données chiffrées: `encryptedData := make([]byte, len(secretBytesDecrypted))` . Dans le cas d'AES-CBC, les données chiffrées avaient la même longueur que les données non chiffrées.

Étape 8

Maintenant, nous devrions créer le crypteur et crypter les données:

```
aesEncrypter := cipher.NewCBCEncrypter(aesBlockEncrypter, vectorBytes)
aesEncrypter.CryptBlocks(encryptedData, secretBytesDecrypted)
```

Maintenant, les données chiffrées se trouvent dans la variable `encryptedData` .

Étape 9

Les données cryptées doivent être stockées. Mais pas seulement les données: sans le sel, les données chiffrées n'ont pas pu être déchiffrées. Nous devons donc utiliser un format de fichier pour gérer cela. Ici, nous encodons les données chiffrées en base64, cf. Wikipedia :

```
encodedBytes := make([]byte, base64.StdEncoding.EncodedLen(len(encryptedData)))
base64.StdEncoding.Encode(encodedBytes, encryptedData)
```

Ensuite, nous définissons notre contenu de fichier et notre propre format de fichier. Le format ressemble à ceci: `salt[0x10]base64 content` . Tout d'abord, nous stockons le sel. Afin de marquer le

début du contenu de base64, nous stockons l'octet `10` . Cela fonctionne, car base64 n'utilise pas cette valeur. Par conséquent, nous pourrions trouver le début de base64 en recherchant la première occurrence de `10` de la fin au début du fichier.

```
fileContent := make([]byte, len(saltBytes))
copy(fileContent, saltBytes)
fileContent = append(fileContent, 10)
fileContent = append(fileContent, encodedBytes...)
```

Étape 10

Enfin, nous pourrions écrire notre fichier: `writeErr := ioutil.WriteFile("my secret.data", fileContent, 0644)` .

Décryptage

Introduction et données

En ce qui concerne le chiffrement, nous avons besoin de certaines données pour travailler. Ainsi, nous supposons que nous avons un fichier crypté et la structure mentionnée `secret` . L'objectif est de lire les données chiffrées à partir du fichier, de les déchiffrer et de créer une instance de la structure.

Étape 1

La première étape est identique au chiffrement: nous avons besoin d'une sorte de mot de passe principal pour déchiffrer le secret: `masterPassword := "PASS"` .

Étape 2

Maintenant, nous lisons les données chiffrées à partir du fichier: `encryptedFileData, bytesErr := ioutil.ReadFile(filename)` .

Étape 3

Comme mentionné précédemment, nous pourrions diviser les données de sel et les données cryptées par l'octet de délimiteur `10` , recherché en arrière de la fin au début:

```
for n := len(encryptedFileData) - 1; n > 0; n-- {
    if encryptedFileData[n] == 10 {
        saltBytes = encryptedFileData[:n]
        encryptedBytesBase64 = encryptedFileData[n+1:]
        break
    }
```

```
}
```

Étape 4

Ensuite, nous devons décoder les octets encodés en base64:

```
decodedBytes := make([]byte, len(encryptedBytesBase64))
countDecoded, decodedErr := base64.StdEncoding.Decode(decodedBytes, encryptedBytesBase64)
encryptedBytes = decodedBytes[:countDecoded]
```

Étape 5

Maintenant, nous sommes en mesure de dériver une clé et un vecteur du mot de passe maître et du sel aléatoire, en ce qui concerne RFC 2898:

```
keyLength := 256
rfc2898Iterations := 6

keyVectorData := pbkdf2.Key(masterPassword, saltBytes, rfc2898Iterations,
(keyLength/8)+aes.BlockSize, sha1.New)
keyBytes := keyVectorData[:keyLength/8]
vectorBytes := keyVectorData[keyLength/8:]
```

Étape 6

Créez un chiffrement AES: `aesBlockDecrypter, aesErr := aes.NewCipher(keyBytes)` .

Étape 7

Réservez la mémoire nécessaire pour les données déchiffrées: `decryptedData := make([]byte, len(encryptedBytes))` . Par définition, il a la même longueur que les données cryptées.

Étape 8

Maintenant, créez le décrypteur et décryptez les données:

```
aesDecrypter := cipher.NewCBCDecrypter(aesBlockDecrypter, vectorBytes)
aesDecrypter.CryptBlocks(decryptedData, encryptedBytes)
```

Étape 9

Convertissez les octets de lecture en chaîne: `decryptedString := string(decryptedData)` . Comme nous avons besoin de lignes, séparez la chaîne: `lines := strings.Split(decryptedString, "\n")` .

Étape 10

Construis un `secret` sur les lignes:

```
artifact := secret{}
artifact.DisplayName = lines[0]
artifact.Notes = lines[1]
artifact.Username = lines[2]
artifact.EMail = lines[3]
artifact.CopyMethod = lines[4]
artifact.Password = lines[5]
artifact.CustomField01Name = lines[6]
artifact.CustomField01Data = lines[7]
artifact.CustomField02Name = lines[8]
artifact.CustomField02Data = lines[9]
artifact.CustomField03Name = lines[10]
artifact.CustomField03Data = lines[11]
artifact.CustomField04Name = lines[12]
artifact.CustomField04Data = lines[13]
artifact.CustomField05Name = lines[14]
artifact.CustomField05Data = lines[15]
artifact.CustomField06Name = lines[16]
artifact.CustomField06Data = lines[17]
```

Enfin, recréez les sauts de ligne dans le champ notes: `artifact.Notes = strings.Replace(artifact.Notes, string(65000), "\n", -1)` .

Lire Cryptographie en ligne: https://riptutorial.com/fr/go/topic/10065/cryptographie

Chapitre 17: Développement pour plusieurs plates-formes avec compilation conditionnelle

Introduction

La compilation conditionnelle basée sur une plate-forme se présente sous deux formes dans Go, l'une avec les suffixes de fichier et l'autre avec les balises de génération.

Syntaxe

- Après " `// +build` ", une seule plate-forme ou une liste peut suivre
- La plate-forme peut être inversée en la précédant par `!` signe
- La liste des plates-formes séparées par un espace est ORed ensemble

Remarques

Mises en garde pour les balises de construction:

- La contrainte de `// +build` doit être placée en haut du fichier, même avant la clause de package.
- Il doit être suivi par une ligne vide pour séparer les commentaires du package.

Liste des plates-formes valides pour les balises de construction et les suffixes de fichiers
Android
Darwin
libellule
freebsd
linux
netbsd
openbsd
plan9
Solaris

Liste des plates-formes valides pour les balises de construction et les suffixes de fichiers
les fenêtres

Reportez-vous à la liste `$GOOS` dans https://golang.org/doc/install/source#environment pour obtenir la liste des plates-formes la plus récente.

Examples

Construire des balises

```
// +build linux

package lib

var OnlyAccessibleInLinux int // Will only be compiled in Linux
```

Annulez une plate-forme en la plaçant `!` avant cela:

```
// +build !windows

package lib

var NotWindows int // Will be compiled in all platforms but not Windows
```

La liste des plates-formes peut être spécifiée en les séparant par des espaces

```
// +build linux darwin plan9

package lib

var SomeUnix int // Will be compiled in linux, darwin and plan9 but not on others
```

Suffixe de fichier

Si vous nommez votre fichier `lib_linux.go` , tout le contenu de ce fichier ne sera compilé que dans des environnements Linux:

```
package lib

var OnlyCompiledInLinux string
```

Définir des comportements distincts sur différentes plates-formes

Différentes plates-formes peuvent avoir des implémentations distinctes de la même méthode. Cet exemple montre également comment les balises de construction et les suffixes de fichiers peuvent être utilisés ensemble.

Fichier `main.go` :

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World from Conditional Compilation Doc!")
    printDetails()
}
```

`details.go` :

```
// +build !windows

package main

import "fmt"

func printDetails() {
    fmt.Println("Some specific details that cannot be found on Windows")
}
```

`details_windows.go` :

```
package main

import "fmt"

func printDetails() {
    fmt.Println("Windows specific details")
}
```

Lire Développement pour plusieurs plates-formes avec compilation conditionnelle en ligne: https://riptutorial.com/fr/go/topic/8599/developpement-pour-plusieurs-plates-formes-avec-compilation-conditionnelle

Chapitre 18: Encodage Base64

Syntaxe

- func (enc * base64.Encoding) Encode (dst, src [] byte)
- func (enc * base64.Encoding) Decode (dst, src [] octet) (n int, erreur err)
- func (enc * base64.Encoding) EncodeToString (src [] byte) chaîne
- func (enc * base64.Encoding) DecodeString (s string) ([] byte, error)

Remarques

Le package `encoding/base64` contient plusieurs encodeurs intégrés . La plupart des exemples de ce document utiliseront `base64.StdEncoding` , mais tout encodeur (`URLEncoding` , `RawStdEncodign` , votre propre encodeur personnalisé, etc.) peut être remplacé.

Examples

Codage

```
const foobar = `foo bar`
encoding := base64.StdEncoding
encodedFooBar := make([]byte, encoding.EncodedLen(len(foobar)))
encoding.Encode(encodedFooBar, []byte(foobar))
fmt.Printf("%s", encodedFooBar)
// Output: Zm9vIGJhcg==
```

Cour de récréation

Encodage d'une chaîne

```
str := base64.StdEncoding.EncodeToString([]byte(`foo bar`))
fmt.Println(str)
// Output: Zm9vIGJhcg==
```

Cour de récréation

Décodage

```
encoding := base64.StdEncoding
data := []byte(`Zm9vIGJhcg==`)
decoded := make([]byte, encoding.DecodedLen(len(data)))
n, err := encoding.Decode(decoded, data)
if err != nil {
    log.Fatal(err)
}

// Because we don't know the length of the data that is encoded
```

```
// (only the max length), we need to trim the buffer to whatever
// the actual length of the decoded data was.
decoded = decoded[:n]

fmt.Printf("`%s`", decoded)
// Output: `foo bar`
```

Cour de récréation

Décoder une chaîne

```
decoded, err := base64.StdEncoding.DecodeString(`biws`)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("%s", decoded)
// Output: n,,
```

Cour de récréation

Lire Encodage Base64 en ligne: https://riptutorial.com/fr/go/topic/4492/encodage-base64

Chapitre 19: Enregistrement

Examples

Impression de base

Go dispose d'une bibliothèque de journalisation intégrée appelée `log` avec une méthode d'utilisation courante, `Print` et ses variantes. Vous pouvez importer la bibliothèque puis faire des impressions de base:

```
package main

import "log"

func main() {

    log.Println("Hello, world!")
    // Prints 'Hello, world!' on a single line

    log.Print("Hello, again! \n")
    // Prints 'Hello, again!' but doesn't break at the end without \n

    hello := "Hello, Stackers!"
    log.Printf("The type of hello is: %T \n", hello)
    // Allows you to use standard string formatting. Prints the type 'string' for %T
    // 'The type of hello is: string
}
```

Connexion au fichier

Il est possible de spécifier la destination du journal avec quelque chose qui correspond à l'interface io.Writer. Avec ça on peut se connecter au fichier:

```
package main

import (
    "log"
    "os"
)

func main() {
    logfile, err := os.OpenFile("test.log", os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
    if err != nil {
        log.Fatalln(err)
    }
    defer logfile.Close()

    log.SetOutput(logfile)
    log.Println("Log entry")
}
```

Sortie:

```
$ cat test.log
2016/07/26 07:29:05 Log entry
```

Connexion à syslog

Il est également possible de se connecter à syslog avec `log/syslog` comme ceci:

```
package main

import (
    "log"
    "log/syslog"
)

func main() {
    syslogger, err := syslog.New(syslog.LOG_INFO, "syslog_example")
    if err != nil {
        log.Fatalln(err)
    }

    log.SetOutput(syslogger)
    log.Println("Log entry")
}
```

Après l'exécution, nous pourrons voir cette ligne dans syslog:

```
Jul 26 07:35:21 localhost syslog_example[18358]: 2016/07/26 07:35:21 Log entry
```

Lire Enregistrement en ligne: https://riptutorial.com/fr/go/topic/3724/enregistrement

Chapitre 20: Envoyer / recevoir des emails

Syntaxe

- func PlainAuth (identité, nom d'utilisateur, mot de passe, chaîne hôte)
- func SendMail (chaîne addr, une authentification, de chaîne, à [] chaîne, octet msg [] octet)

Examples

Envoi d'emails avec smtp.SendMail ()

L'envoi de courrier électronique est assez simple dans Go. Il est utile de comprendre le RFC 822, qui spécifie le style dans lequel un courrier électronique doit être, le code ci-dessous envoie un courrier électronique conforme à la RFC 822.

```
package main

import (
    "fmt"
    "net/smtp"
)

func main() {
    // user we are authorizing as
    from := "someuser@example.com"

    // use we are sending email to
    to := "otheruser@example.com"

    // server we are authorized to send email through
    host := "mail.example.com"

    // Create the authentication for the SendMail()
    // using PlainText, but other authentication methods are encouraged
    auth := smtp.PlainAuth("", from, "password", host)

    // NOTE: Using the backtick here ` works like a heredoc, which is why all the
    // rest of the lines are forced to the beginning of the line, otherwise the
    // formatting is wrong for the RFC 822 style
    message := `To: "Some User" <someuser@example.com>
From: "Other User" <otheruser@example.com>
Subject: Testing Email From Go!!

This is the message we are sending. That's it!
`

    if err := smtp.SendMail(host+":25", auth, from, []string{to}, []byte(message)); err != nil
{
        fmt.Println("Error SendMail: ", err)
        os.Exit(1)
    }
    fmt.Println("Email Sent!")
}
```

Le message ci-dessus enverra un message comme celui-ci:

```
To: "Other User" <otheruser@example.com>
From: "Some User" <someuser@example.com>
Subject: Testing Email From Go!!

This is the message we are sending. That's it!
.
```

Lire Envoyer / recevoir des emails en ligne: https://riptutorial.com/fr/go/topic/5912/envoyer---recevoir-des-emails

Chapitre 21: Essai

Introduction

Go est livré avec ses propres installations de test qui ont tout ce qu'il faut pour exécuter des tests et des tests de performances. Contrairement à la plupart des autres langages de programmation, il n'est souvent pas nécessaire de disposer d'un cadre de test distinct, bien que certains existent.

Examples

Test de base

`main.go` :

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(Sum(4,5))
}

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

`main_test.go` :

```
package main

import (
    "testing"
)

// Test methods start with `Test`
func TestSum(t *testing.T) {
    got := Sum(1, 2)
    want := 3
    if got != want {
        t.Errorf("Sum(1, 2) == %d, want %d", got, want)
    }
}
```

Pour exécuter le test, utilisez simplement la commande `go test` :

```
$ go test
ok      test_app    0.005s
```

Utilisez le drapeau `-v` pour voir les résultats de chaque test:

```
$ go test -v
=== RUN   TestSum
--- PASS: TestSum (0.00s)
PASS
ok      _/tmp    0.000s
```

Utilisez le chemin d'accès `./...` pour tester les sous-répertoires de manière récursive:

```
$ go test -v ./...
ok      github.com/me/project/dir1    0.008s
=== RUN   TestSum
--- PASS: TestSum (0.00s)
PASS
ok      github.com/me/project/dir2    0.008s
=== RUN   TestDiff
--- PASS: TestDiff (0.00s)
PASS
```

Exécuter un test particulier:
S'il existe plusieurs tests et que vous souhaitez exécuter un test spécifique, vous pouvez le faire comme suit:

```
go test -v -run=<TestName> // will execute only test with this name
```

Exemple:

```
go test -v run=TestSum
```

Tests de benchmark

Si vous souhaitez mesurer les benchmarks, ajoutez une méthode de test comme celle-ci:

`sum.go` :

```
package sum

// Sum calculates the sum of two integers
func Sum(a, b int) int {
    return a + b
}
```

`sum_test.go` :

```
package sum

import "testing"

func BenchmarkSum(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = Sum(2, 3)
    }
}
```

Ensuite, pour exécuter un benchmark simple:

```
$ go test -bench=.
BenchmarkSum-8    2000000000             0.49 ns/op
ok      so/sum    1.027s
```

Tests unitaires pilotés par tableau

Ce type de test est une technique populaire pour tester avec des valeurs d'entrée et de sortie prédéfinies.

Créez un fichier appelé `main.go` avec le contenu:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(Sum(4, 5))
}

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

Après l'avoir exécuté avec, vous verrez que le résultat est `9` . Bien que la fonction `Sum` semble assez simple, il est conseillé de tester votre code. Pour ce faire, nous créons un autre fichier nommé `main_test.go` dans le même dossier que `main.go` , contenant le code suivant:

```
package main

import (
    "testing"
)

// Test methods start with Test
func TestSum(t *testing.T) {
    // Note that the data variable is of type array of anonymous struct,
    // which is very handy for writing table-driven unit tests.
    data := []struct {
        a, b, res int
    }{
        {1, 2, 3},
        {0, 0, 0},
        {1, -1, 0},
        {2, 3, 5},
        {1000, 234, 1234},
    }

    for _, d := range data {
        if got := Sum(d.a, d.b); got != d.res {
            t.Errorf("Sum(%d, %d) == %d, want %d", d.a, d.b, got, d.res)
        }
    }
```

```
}
```

Comme vous pouvez le constater, une partie des structures anonymes est créée, chacune avec un ensemble d'entrées et le résultat attendu. Cela permet de créer un grand nombre de cas de test tous au même endroit, puis de les exécuter en boucle, ce qui réduit la répétition du code et améliore la clarté.

Exemples de tests (tests d'auto-documentation)

Ce type de test s'assure que votre code compile correctement et apparaîtra dans la documentation générée pour votre projet. En plus de cela, les tests d'exemple peuvent affirmer que votre test produit une sortie correcte.

`sum.go` :

```
package sum

// Sum calculates the sum of two integers
func Sum(a, b int) int {
    return a + b
}
```

`sum_test.go` :

```
package sum

import "fmt"

func ExampleSum() {
    x := Sum(1, 2)
    fmt.Println(x)
    fmt.Println(Sum(-1, -1))
    fmt.Println(Sum(0, 0))

    // Output:
    // 3
    // -2
    // 0
}
```

Pour exécuter votre test, exécutez `go test` dans le dossier contenant ces fichiers ou placez ces deux fichiers dans un sous-dossier nommé `sum` , puis dans le dossier parent, exécutez `go test ./sum` . Dans les deux cas, vous obtiendrez une sortie similaire à celle-ci:

```
ok      so/sum    0.005s
```

Si vous vous demandez comment cela teste votre code, voici un autre exemple de fonction, qui échoue en fait au test:

```
func ExampleSum_fail() {
    x := Sum(1, 2)
    fmt.Println(x)
```

```
    // Output:
    // 5
}
```

Lorsque vous exécutez `go test` , vous obtenez la sortie suivante:

```
$ go test
--- FAIL: ExampleSum_fail (0.00s)
got:
3
want:
5
FAIL
exit status 1
FAIL    so/sum    0.006s
```

Si vous voulez voir la documentation de votre paquet de `sum` - lancez simplement:

```
go doc -http=:6060
```

et accédez à http: // localhost: 6060 / pkg / FOLDER / sum / , où *FOLDER* est le dossier contenant le package de `sum` (dans cet exemple, `so`). La documentation de la méthode sum ressemble à ceci:

Package sum

```
import "so/sum"
```

Overview
Index
Examples

Overview ▾

Package sum is a sample package for test purpos

Index ▾

func Sum(a, b int) int

Examples

Sum

Package files

sum.go

- Une fonction tearDown effectue une restauration.

C'est une bonne option lorsque vous ne pouvez pas modifier votre base de données et que vous devez créer un objet qui simule un objet apporté par la base de données ou qui doit lancer une configuration dans chaque test.

Un exemple stupide serait:

```
// Standard numbers map
var numbers map[string]int = map[string]int{"zero": 0, "three": 3}

// TestMain will exec each test, one by one
func TestMain(m *testing.M) {
    // exec setUp function
    setUp("one", 1)
    // exec test and this returns an exit code to pass to os
    retCode := m.Run()
    // exec tearDown function
    tearDown("one")
    // If exit code is distinct of zero,
    // the test will be failed (red)
    os.Exit(retCode)
}

// setUp function, add a number to numbers slice
func setUp(key string, value int) {
    numbers[key] = value
}

// tearDown function, delete a number to numbers slice
func tearDown(key string) {
    delete(numbers, key)
}

// First test
func TestOnePlusOne(t *testing.T) {
    numbers["one"] = numbers["one"] + 1

    if numbers["one"] != 2 {
        t.Error("1 plus 1 = 2, not %v", value)
    }
}

// Second test
func TestOnePlusTwo(t *testing.T) {
    numbers["one"] = numbers["one"] + 2

    if numbers["one"] != 3 {
        t.Error("1 plus 2 = 3, not %v", value)
    }
}
```

Autre exemple: préparer une base de données pour tester et effectuer un retour en arrière

```
 // ID of Person will be saved in database
personID := 12345
// Name of Person will be saved in database
personName := "Toni"
```

```
func TestMain(m *testing.M) {
    // You create an Person and you save in database
    setUp(&Person{
            ID:   personID,
            Name: personName,
            Age:  19,
        })
    retCode := m.Run()
    // When you have executed the test, the Person is deleted from database
    tearDown(personID)
    os.Exit(retCode)
}

func setUp(P *Person) {
    // ...
    db.add(P)
    // ...
}

func tearDown(id int) {
    // ...
    db.delete(id)
    // ...
}

func getPerson(t *testing.T) {
    P := Get(personID)

    if P.Name != personName {
        t.Error("P.Name is %s and it must be Toni", P.Name)
    }
}
```

Afficher la couverture du code au format HTML

Exécuter `go test` comme d'habitude, mais avec l'indicateur `coverprofile` . Utilisez `go tool` pour afficher les résultats au format HTML.

```
    go test -coverprofile=c.out
    go tool cover -html=c.out
```

Lire Essai en ligne: https://riptutorial.com/fr/go/topic/1234/essai

Chapitre 22: Exécution des commandes

Examples

Chronométrer avec interruption puis tuer

```
c := exec.Command(name, arg...)
b := &bytes.Buffer{}
c.Stdout = b
c.Stdin = stdin
if err := c.Start(); err != nil {
    return nil, err
}
timedOut := false
intTimer := time.AfterFunc(timeout, func() {
    log.Printf("Process taking too long. Interrupting: %s %s", name, strings.Join(arg, " "))
    c.Process.Signal(os.Interrupt)
    timedOut = true
})
killTimer := time.AfterFunc(timeout*2, func() {
    log.Printf("Process taking too long. Killing: %s %s", name, strings.Join(arg, " "))
    c.Process.Signal(os.Kill)
    timedOut = true
})
err := c.Wait()
intTimer.Stop()
killTimer.Stop()
if timedOut {
    log.Print("the process timed out\n")
}
```

Exécution de commande simple

```
// Execute a command a capture standard out. exec.Command creates the command
// and then the chained Output method gets standard out. Use CombinedOutput()
// if you want both standard out and standerr output
out, err := exec.Command("echo", "foo").Output()
if err != nil {
    log.Fatal(err)
}
```

Exécuter une commande, puis continuer et attendre

```
cmd := exec.Command("sleep", "5")

// Does not wait for command to complete before returning
err := cmd.Start()
if err != nil {
    log.Fatal(err)
}

// Wait for cmd to Return
err = cmd.Wait()
```

```
log.Printf("Command finished with error: %v", err)
```

Exécuter une commande deux fois

> Un Cmd ne peut pas être réutilisé après l'appel de ses méthodes Run, Output ou CombinedOutput

Exécuter une commande deux fois ***ne*** *fonctionnera* ***pas*** :

```
cmd := exec.Command("xte", "key XF86AudioPlay")
_ := cmd.Run() // Play audio key press
// .. do something else
err := cmd.Run() // Pause audio key press, fails
```

> Erreur: exec: déjà commencé

Il faut plutôt utiliser **deux** `exec.Command` **séparés** . Vous pourriez aussi avoir besoin d'un délai entre les commandes.

```
cmd := exec.Command("xte", "key XF86AudioPlay")
_ := cmd.Run() // Play audio key press
// .. wait a moment
cmd := exec.Command("xte", "key XF86AudioPlay")
_ := cmd.Run() // Pause audio key press
```

Lire Exécution des commandes en ligne: https://riptutorial.com/fr/go/topic/1097/execution-des-commandes

Chapitre 23: Expansion Inline

Remarques

L'expansion en ligne est une optimisation courante du code compilé qui privilégie les performances par rapport à la taille binaire. Il permet au compilateur de remplacer un appel de fonction par le corps même de la fonction; copier / coller efficacement du code d'un endroit à un autre au moment de la compilation. Comme le site d'appel est développé pour contenir uniquement les instructions machine générées par le compilateur pour la fonction, nous n'avons pas à exécuter un appel CALL ou PUSH (l'équivalent x86 d'une instruction GOTO ou d'un push de cadre) ou leur équivalent sur d'autres des architectures.

L'inliner prend des décisions sur l'inclusion ou non d'une fonction basée sur un certain nombre d'heuristiques, mais en général, Go s'intègre par défaut. Parce que l'inliner se débarrasse des appels de fonction, il parvient effectivement à décider où le planificateur est autorisé à préempter une goroutine.

Les appels de fonction ne seront pas mis en ligne si l'une des conditions suivantes est vraie (il y a également beaucoup d'autres raisons, cette liste est incomplète):

- Les fonctions sont variadiques (par exemple, elles ont `...` args)
- Les fonctions ont un "maximum de cheveux" supérieur au budget (elles se répètent trop ou ne peuvent pas être analysées pour une autre raison)
- Ils contiennent la `panic` , `recover` ou `defer`

Examples

Désactiver l'extension en ligne

L'expansion en ligne peut être désactivée avec le pragma `go:noinline` . Par exemple, si nous construisons le programme simple suivant:

```
package main

func printhello() {
    println("Hello")
}

func main() {
    printhello()
}
```

nous obtenons une sortie qui ressemble à ceci (ajustée pour la lisibilité):

```
$ go version
go version go1.6.2 linux/amd64
$ go build main.go
```

```
$ ./main
Hello
$ go tool objdump main
TEXT main.main(SB) /home/sam/main.go
        main.go:7       0x401000        64488b0c25f8ffffff      FS MOVQ FS:0xfffffff8, CX
        main.go:7       0x401009        483b6110                CMPQ 0x10(CX), SP
        main.go:7       0x40100d        7631                    JBE 0x401040
        main.go:7       0x40100f        4883ec10                SUBQ $0x10, SP
        main.go:8       0x401013        e8281f0200              CALL runtime.printlock(SB)
        main.go:8       0x401018        488d1d01130700          LEAQ 0x71301(IP), BX
        main.go:8       0x40101f        48891c24                MOVQ BX, 0(SP)
        main.go:8       0x401023        48c744240805000000      MOVQ $0x5, 0x8(SP)
        main.go:8       0x40102c        e81f290200              CALL runtime.printstring(SB)
        main.go:8       0x401031        e89a210200              CALL runtime.printnl(SB)
        main.go:8       0x401036        e8851f0200              CALL runtime.printunlock(SB)
        main.go:9       0x40103b        4883c410                ADDQ $0x10, SP
        main.go:9       0x40103f        c3                      RET
        main.go:7       0x401040        e87b9f0400              CALL
runtime.morestack_noctxt(SB)
        main.go:7       0x401045        ebb9                    JMP main.main(SB)
        main.go:7       0x401047        cc                      INT $0x3
        main.go:7       0x401048        cc                      INT $0x3
        main.go:7       0x401049        cc                      INT $0x3
        main.go:7       0x40104a        cc                      INT $0x3
        main.go:7       0x40104b        cc                      INT $0x3
        main.go:7       0x40104c        cc                      INT $0x3
        main.go:7       0x40104d        cc                      INT $0x3
        main.go:7       0x40104e        cc                      INT $0x3
        main.go:7       0x40104f        cc                      INT $0x3
…
```

noter qu'il n'y a pas `CALL` à `printhello` . Cependant, si nous construisons ensuite le programme avec le pragma en place:

```
package main

//go:noinline
func printhello() {
    println("Hello")
}

func main() {
    printhello()
}
```

La sortie contient la fonction printhello et un `CALL main.printhello` :

```
$ go version
go version go1.6.2 linux/amd64
$ go build main.go
$ ./main
Hello
$ go tool objdump main
TEXT main.printhello(SB) /home/sam/main.go
        main.go:4       0x401000        64488b0c25f8ffffff      FS MOVQ FS:0xfffffff8, CX
        main.go:4       0x401009        483b6110                CMPQ 0x10(CX), SP
        main.go:4       0x40100d        7631                    JBE 0x401040
        main.go:4       0x40100f        4883ec10                SUBQ $0x10, SP
```

```
        main.go:5       0x401013        e8481f0200              CALL runtime.printlock(SB)
        main.go:5       0x401018        488d1d01130700          LEAQ 0x71301(IP), BX
        main.go:5       0x40101f        48891c24                MOVQ BX, 0(SP)
        main.go:5       0x401023        48c744240805000000      MOVQ $0x5, 0x8(SP)
        main.go:5       0x40102c        e83f290200              CALL runtime.printstring(SB)
        main.go:5       0x401031        e8ba210200              CALL runtime.printnl(SB)
        main.go:5       0x401036        e8a51f0200              CALL runtime.printunlock(SB)
        main.go:6       0x40103b        4883c410                ADDQ $0x10, SP
        main.go:6       0x40103f        c3                      RET
        main.go:4       0x401040        e89b9f0400              CALL
runtime.morestack_noctxt(SB)
        main.go:4       0x401045        ebb9                    JMP main.printhello(SB)
        main.go:4       0x401047        cc                      INT $0x3
        main.go:4       0x401048        cc                      INT $0x3
        main.go:4       0x401049        cc                      INT $0x3
        main.go:4       0x40104a        cc                      INT $0x3
        main.go:4       0x40104b        cc                      INT $0x3
        main.go:4       0x40104c        cc                      INT $0x3
        main.go:4       0x40104d        cc                      INT $0x3
        main.go:4       0x40104e        cc                      INT $0x3
        main.go:4       0x40104f        cc                      INT $0x3

TEXT main.main(SB) /home/sam/main.go
        main.go:8       0x401050        64488b0c25f8ffffff      FS MOVQ FS:0xfffffff8, CX
        main.go:8       0x401059        483b6110                CMPQ 0x10(CX), SP
        main.go:8       0x40105d        7606                    JBE 0x401065
        main.go:9       0x40105f        e89cffffff              CALL main.printhello(SB)
        main.go:10      0x401064        c3                      RET
        main.go:8       0x401065        e8769f0400              CALL
runtime.morestack_noctxt(SB)
        main.go:8       0x40106a        ebe4                    JMP main.main(SB)
        main.go:8       0x40106c        cc                      INT $0x3
        main.go:8       0x40106d        cc                      INT $0x3
        main.go:8       0x40106e        cc                      INT $0x3
        main.go:8       0x40106f        cc                      INT $0x3
…
```

Lire Expansion Inline en ligne: https://riptutorial.com/fr/go/topic/2718/expansion-inline

Chapitre 24: Fermetures

Examples

Les bases de la fermeture

Une *fermeture* est une fonction associée à un environnement. La fonction est généralement une fonction anonyme définie dans une autre fonction. L'environnement est la portée lexicale de la fonction englobante (l'idée fondamentale d'une portée lexicale d'une fonction serait la portée qui existe entre les accolades de la fonction.)

```
func g() {
    i := 0
    f := func() { // anonymous function
        fmt.Println("f called")
    }
}
```

Dans le corps d'une fonction anonyme (disons `f`) définie dans une autre fonction (disons `g`), les variables présentes dans les portées de `f` et de `g` sont accessibles. Cependant, c'est la portée de `g` qui constitue la partie de l'environnement de la fermeture (la partie fonction est `f`) et, par conséquent, les modifications apportées aux variables de la portée de `g` conservent leurs valeurs (l'environnement persiste entre les appels à `f`) .

Considérons la fonction ci-dessous:

```
func NaturalNumbers() func() int {
    i := 0
    f:= func() int { // f is the function part of closure
        i++
        return i
    }
    return f
}
```

Dans la définition ci-dessus, `NaturalNumbers` a une fonction interne `f` que retourne `NaturalNumbers` . À l'intérieur de `f` , la variable `i` définie dans le cadre de `NaturalNumbers` est en cours d'accès.

Nous obtenons une nouvelle fonction de `NaturalNumbers` comme ceci:

```
n := NaturalNumbers()
```

Maintenant `n` est une fermeture. C'est une fonction (définie par `f`) qui possède également un environnement associé (portée de `NaturalNumbers`).

Dans le cas de `n` , la partie environnement ne contient qu'une variable: `i`

Puisque `n` est une fonction, on peut l'appeler:

```
fmt.Println(n()) // 1
fmt.Println(n()) // 2
fmt.Println(n()) // 3
```

Comme il ressort de la sortie ci-dessus, chaque fois que `n` est appelé, il incrémente `i` . `i` commence à 0 et chaque appel à `n` exécute `i++` .

La valeur de `i` est conservée entre les appels. En d'autres termes, l'environnement, qui fait partie de la fermeture, persiste.

Appeler `NaturalNumbers` nouveau créerait et renverrait une nouvelle fonction. Cela initialiserait un nouveau `i` dans `NaturalNumbers` . Ce qui signifie que la fonction nouvellement retournée forme une autre fermeture ayant le même rôle pour la fonction (toujours `f`) mais un nouvel environnement (un `i` nouvellement initialisé).

```
o := NaturalNumbers()

fmt.Println(n()) // 4
fmt.Println(o()) // 1
fmt.Println(o()) // 2
fmt.Println(n()) // 5
```

Les deux `n` et `o` sont des fermetures contenant la même partie de fonction (ce qui leur donne le même comportement), mais des environnements différents. Ainsi, l'utilisation des fermetures permet aux fonctions d'accéder à un environnement persistant pouvant être utilisé pour conserver les informations entre les appels.

Un autre exemple:

```
func multiples(i int) func() int {
    var x int = 0
    return func() int {
        x++
        // paramenter to multiples (here it is i) also forms
        // a part of the environment, and is retained
        return x * i
    }
}

two := multiples(2)
fmt.Println(two(), two(), two()) // 2 4 6

fortyTwo := multiples(42)
fmt.Println(fortyTwo(), fortyTwo(), fortyTwo()) // 42 84 126
```

Lire Fermetures en ligne: https://riptutorial.com/fr/go/topic/2741/fermetures

Chapitre 25: Fichier I / O

Syntaxe

- fichier, err: = os.Open (*name*) // Ouvre un fichier en mode lecture seule. Une erreur non nulle est renvoyée si le fichier n'a pas pu être ouvert.
- fichier, err: = os.Create (*name*) // Crée ou ouvre un fichier s'il existe déjà en mode écriture seule. Le fichier est écrasé s'il existe déjà. Une erreur non nulle est renvoyée si le fichier n'a pas pu être ouvert.
- fichier, err: = os.OpenFile (*name* , *flags* , *perm*) // Ouvre un fichier dans le mode spécifié par les indicateurs. Une erreur non nulle est renvoyée si le fichier n'a pas pu être ouvert.
- data, err: = ioutil.ReadFile (*name*) // Lit l'intégralité du fichier et le renvoie. Une erreur non nulle est renvoyée si le fichier entier n'a pas pu être lu.
- err: = ioutil.WriteFile (*nom* , *données* , *perm*) // Crée ou écrase un fichier avec les données fournies et les bits d'autorisation UNIX. Une erreur non nulle est renvoyée si le fichier n'a pas pu être écrit.
- err: = os.Remove (*name*) // Supprime un fichier. Une erreur non nulle est renvoyée si le fichier n'a pas pu être supprimé.
- err: = os.RemoveAll (*name*) // Supprime une hiérarchie de fichiers ou de répertoires entiers. Une erreur non nulle est renvoyée si le fichier ou le répertoire n'a pas pu être supprimé.
- err: = os.Rename *(oldName, newName)* // Renomme ou déplace un fichier (peut être à travers les répertoires). Une erreur non nulle est renvoyée si le fichier n'a pas pu être déplacé.

Paramètres

Paramètre	Détails
prénom	Un nom de fichier ou un chemin de type chaîne. Par exemple: `"hello.txt"` .
se tromper	Une `error` S'il n'est pas `nil` , il représente une erreur qui s'est produite lors de l'appel de la fonction.
fichier	Un gestionnaire de fichiers de type `*os.File` renvoyé par les fonctions associées au fichier de package `os` . Il implémente un `io.ReadWriter` , ce qui signifie que vous pouvez appeler `Read(data)` et `Write(data)` sur celui-ci. Notez que ces fonctions peuvent ne pas pouvoir être appelées en fonction des drapeaux ouverts du fichier.
Les données	Une tranche d'octets (`[]byte`) représentant les données brutes d'un fichier.
permanente	Les bits d'autorisation UNIX utilisés pour ouvrir un fichier de type `os.FileMode` . Plusieurs constantes sont disponibles pour vous aider à utiliser les bits de permission.

Paramètre	Détails
drapeau	Drapeaux ouverts de fichiers qui déterminent les méthodes pouvant être appelées sur le gestionnaire de fichiers de type `int` . Plusieurs constantes sont disponibles pour vous aider à utiliser les indicateurs. Ce sont: `os.O_RDONLY` , `os.O_WRONLY` , `os.O_RDWR` , `os.O_APPEND` , `os.O_CREATE` , `os.O_EXCL` , `os.O_SYNC` et `os.O_TRUNC` .

Examples

Lire et écrire dans un fichier en utilisant ioutil

Un programme simple qui écrit "Hello, world!" to `test.txt` , lit les données et les imprime. Montre des opérations simples d'E / S de fichier.

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    hello := []byte("Hello, world!")

    // Write `Hello, world!` to test.txt that can read/written by user and read by others
    err := ioutil.WriteFile("test.txt", hello, 0644)
    if err != nil {
        panic(err)
    }

    // Read test.txt
    data, err := ioutil.ReadFile("test.txt")
    if err != nil {
        panic(err)
    }

    // Should output: `The file contains: Hello, world!`
    fmt.Println("The file contains: " + string(data))
}
```

Liste de tous les fichiers et dossiers du répertoire en cours

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    files, err := ioutil.ReadDir(".")
    if err != nil {
        panic(err)
```

```
    }

    fmt.Println("Files and folders in the current directory:")

    for _, fileInfo := range files {
        fmt.Println(fileInfo.Name())
    }
}
```

Liste de tous les dossiers du répertoire en cours

```
package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    files, err := ioutil.ReadDir(".")
    if err != nil {
        panic(err)
    }

    fmt.Println("Folders in the current directory:")

    for _, fileInfo := range files {
        if fileInfo.IsDir() {
            fmt.Println(fileInfo.Name())
        }
    }
}
```

Lire Fichier I / O en ligne: https://riptutorial.com/fr/go/topic/1033/fichier-i---o

Chapitre 26: Fmt

Examples

Raidisseur

L'interface `fmt.Stringer` nécessite une seule méthode, `String() string` à satisfaire. La méthode string définit le format de chaîne "natif" pour cette valeur et constitue la représentation par défaut si la valeur est fournie à l'une des routines de formatage ou d'impression des packages `fmt` .

```
package main

import (
    "fmt"
)

type User struct {
    Name  string
    Email string
}

// String satisfies the fmt.Stringer interface for the User type
func (u User) String() string {
    return fmt.Sprintf("%s <%s>", u.Name, u.Email)
}

func main() {
    u := User{
        Name:  "John Doe",
        Email: "johndoe@example.com",
    }

    fmt.Println(u)
    // output: John Doe <johndoe@example.com>
}
```

Playground

Fmt de base

Le package fmt implémente les E / S formatées en utilisant les *verbes de* format:

```
%v    // the value in a default format
%T    // a Go-syntax representation of the type of the value
%s    // the uninterpreted bytes of the string or slice
```

Fonctions de format

Il y a **4** types de fonctions principaux dans `fmt` et plusieurs variations à l'intérieur.

Impression

```
fmt.Print("Hello World")        // prints: Hello World
fmt.Println("Hello World")      // prints: Hello World\n
fmt.Printf("Hello %s", "World") // prints: Hello World
```

Sprint

```
formattedString := fmt.Sprintf("%v %s", 2, "words") // returns string "2 words"
```

Fprint

```
byteCount, err := fmt.Fprint(w, "Hello World") // writes to io.Writer w
```

`Fprint` peut être utilisé, à l'intérieur des gestionnaires `http` :

```
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello %s!", "Browser")
}   // Writes: "Hello Browser!" onto http response
```

Balayage

Scan analyse le texte lu depuis l'entrée standard.

```
var s string
fmt.Scanln(&s) // pass pointer to buffer
// Scanln is similar to fmt.Scan(), but it stops scanning at new line.
fmt.Println(s) // whatever was inputted
```

Interface Stringer

Toute valeur qui a une méthode `String()` implémente l' **interface** `fmt Stringer`

```
type Stringer interface {
        String() string
}
```

Lire Fmt en ligne: https://riptutorial.com/fr/go/topic/2938/fmt

Chapitre 27: Goroutines

Introduction

Une goroutine est un thread léger géré par le runtime Go.

aller f (x, y, z)

commence un nouveau goroutine en cours d'exécution

f (x, y, z)

L'évaluation de f, x, y et z se produit dans le goroutine actuel et l'exécution de f se produit dans le nouveau goroutine.

Les Goroutines s'exécutent dans le même espace d'adressage, de sorte que l'accès à la mémoire partagée doit être synchronisé. Le package de synchronisation fournit des primitives utiles, bien que vous n'en ayez pas besoin beaucoup dans Go car il existe d'autres primitives.

Référence: https://tour.golang.org/concurrency/1

Examples

Programme de base de Goroutines

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

Une goroutine est une fonction capable de fonctionner simultanément avec d'autres fonctions. Pour créer une goroutine, nous utilisons le mot `go` clé `go` suivi d'une invocation de fonction:

```
package main

import "fmt"
```

```
func f(n int) {
  for i := 0; i < 10; i++ {
    fmt.Println(n, ":", i)
  }
}

func main() {
  go f(0)
  var input string
  fmt.Scanln(&input)
}
```

En général, l'appel de fonction exécute toutes les instructions à l'intérieur du corps de la fonction et retourne à la ligne suivante. Mais, avec les goroutines, nous retournons immédiatement à la ligne suivante, car elle n'attend pas la fin de la fonction. Ainsi, un appel à une fonction `Scanln` est inclus, sinon le programme a été quitté sans imprimer les chiffres.

Lire Goroutines en ligne: https://riptutorial.com/fr/go/topic/9776/goroutines

Chapitre 28: gueule

Introduction

Gob est une méthode de sérialisation spécifique à Go. Il prend en charge tous les types de données Go, à l'exception des canaux et des fonctions. Gob code également les informations de type dans la forme sérialisée, ce qui les différencie de XML, par exemple, c'est qu'elles sont beaucoup plus efficaces.

L'inclusion d'informations de type rend le codage et le décodage assez robustes aux différences entre le codeur et le décodeur.

Examples

Comment encoder des données et écrire dans un fichier avec gob?

```
package main

import (
    "encoding/gob"
    "os"
)

type User struct {
    Username string
    Password string
}

func main() {

    user := User{
        "zola",
        "supersecretpassword",
    }

    file, _ := os.Create("user.gob")

    defer file.Close()

    encoder := gob.NewEncoder(file)

    encoder.Encode(user)

}
```

Comment lire les données d'un fichier et les décoder avec go?

```
package main

import (
    "encoding/gob"
```

```
    "fmt"
    "os"
)

type User struct {
    Username string
    Password string
}

func main() {

    user := User{}

    file, _ := os.Open("user.gob")

    defer file.Close()

    decoder := gob.NewDecoder(file)

    decoder.Decode(&user)

    fmt.Println(user)

}
```

Comment encoder une interface avec gob?

```
package main

import (
    "encoding/gob"
    "fmt"
    "os"
)

type User struct {
    Username string
    Password string
}

type Admin struct {
    Username string
    Password string
    IsAdmin  bool
}

type Deleter interface {
    Delete()
}

func (u User) Delete() {
    fmt.Println("User ==> Delete()")
}

func (a Admin) Delete() {
    fmt.Println("Admin ==> Delete()")
}

func main() {
```

```
    user := User{
        "zola",
        "supersecretpassword",
    }

    admin := Admin{
        "john",
        "supersecretpassword",
        true,
    }

    file, _ := os.Create("user.gob")

    adminFile, _ := os.Create("admin.gob")

    defer file.Close()

    defer adminFile.Close()

    gob.Register(User{}) // registering the type allows us to encode it

    gob.Register(Admin{}) // registering the type allows us to encode it

    encoder := gob.NewEncoder(file)

    adminEncoder := gob.NewEncoder(adminFile)

    InterfaceEncode(encoder, user)

    InterfaceEncode(adminEncoder, admin)

}

func InterfaceEncode(encoder *gob.Encoder, d Deleter) {

    if err := encoder.Encode(&d); err != nil {
        fmt.Println(err)
    }

}
```

Comment décoder une interface avec gob?

```
package main

import (
    "encoding/gob"
    "fmt"
    "log"
    "os"
)

type User struct {
    Username string
    Password string
}

type Admin struct {
```

```
    Username string
    Password string
    IsAdmin  bool
}

type Deleter interface {
    Delete()
}

func (u User) Delete() {
    fmt.Println("User ==> Delete()")
}

func (a Admin) Delete() {
    fmt.Println("Admin ==> Delete()")
}

func main() {

    file, _ := os.Open("user.gob")

    adminFile, _ := os.Open("admin.gob")

    defer file.Close()

    defer adminFile.Close()

    gob.Register(User{}) // registering the type allows us to encode it

    gob.Register(Admin{}) // registering the type allows us to encode it

    var admin Deleter

    var user Deleter

    userDecoder := gob.NewDecoder(file)

    adminDecoder := gob.NewDecoder(adminFile)

    user = InterfaceDecode(userDecoder)

    admin = InterfaceDecode(adminDecoder)

    fmt.Println(user)

    fmt.Println(admin)

}

func InterfaceDecode(decoder *gob.Decoder) Deleter {

    var d Deleter

    if err := decoder.Decode(&d); err != nil {
        log.Fatal(err)
    }

    return d

}
```

Lire gueule en ligne: https://riptutorial.com/fr/go/topic/8820/gueule

Chapitre 29: Images

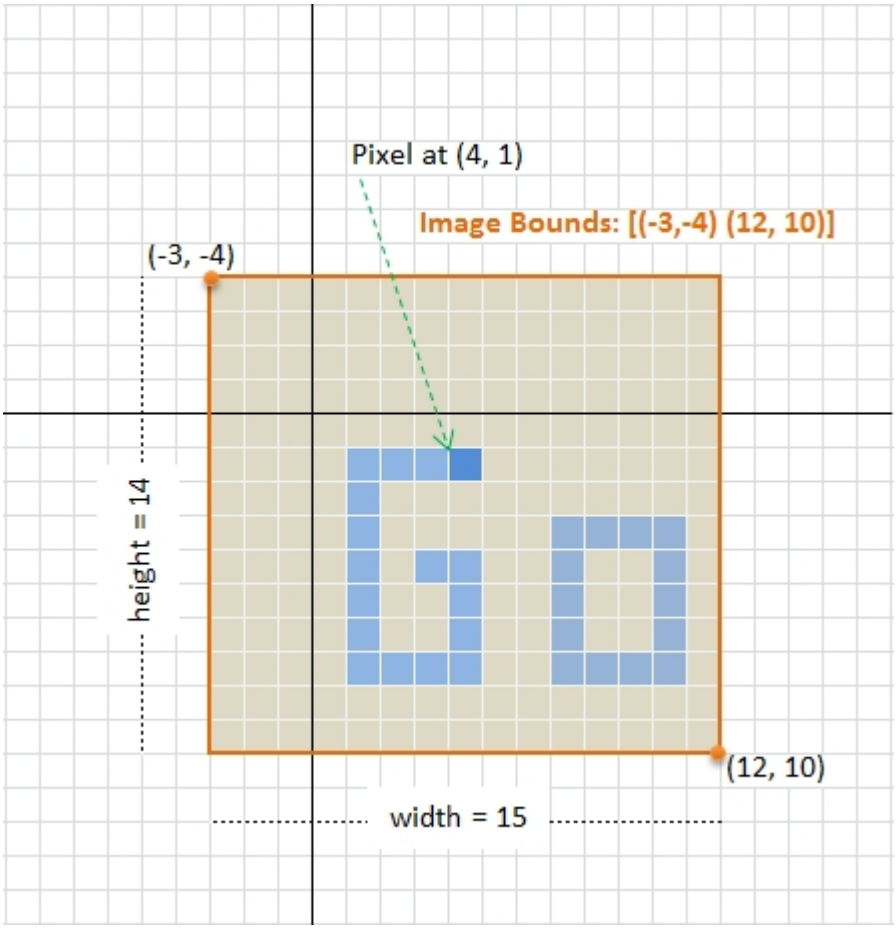
Introduction

Le package d' image fournit des fonctionnalités de base pour travailler avec une image 2D. Cette rubrique décrit plusieurs opérations de base lors de l'utilisation d'images telles que la lecture et l'écriture d'un format d'image particulier, le recadrage, l'accès et la modification de *pixels* , la conversion de couleurs, le redimensionnement et le filtrage d'image de base.

Examples

Concepts de base

Une image représente une grille rectangulaire d'éléments d'image (*pixel*). Dans le package d' image , le pixel est représenté comme l'une des couleurs définies dans le package image / couleur . La géométrie 2D de l'image est représentée sous la forme `image.Rectangle` , tandis que `image.Point` indique une position sur la grille.



La figure ci-dessus illustre les concepts de base d'une image dans le package. Une image de taille 15x14 pixels a des *limites* rectangulaires au *coin supérieur gauche* (par exemple, coordonnée (-3, -4) dans la figure ci-dessus), et ses axes augmentent vers le *bas et le* coin *droit* (par exemple, coordonnée (12, 10) sur la figure). Notez que les limites **ne partent pas**

nécessairement du point (0,0) .

Type lié à l'image

Dans `Go` , une image implémente toujours l'interface suivante `image.Image`

```
type Image interface {
    // ColorModel returns the Image's color model.
    ColorModel() color.Model
    // Bounds returns the domain for which At can return non-zero color.
    // The bounds do not necessarily contain the point (0, 0).
    Bounds() Rectangle
    // At returns the color of the pixel at (x, y).
    // At(Bounds().Min.X, Bounds().Min.Y) returns the upper-left pixel of the grid.
    // At(Bounds().Max.X-1, Bounds().Max.Y-1) returns the lower-right one.
    At(x, y int) color.Color
}
```

dans lequel l'interface `color.Color` est définie comme

```
type Color interface {
    // RGBA returns the alpha-premultiplied red, green, blue and alpha values
    // for the color. Each value ranges within [0, 0xffff], but is represented
    // by a uint32 so that multiplying by a blend factor up to 0xffff will not
    // overflow.
    //
    // An alpha-premultiplied color component c has been scaled by alpha (a),
    // so has valid values 0 <= c <= a.
    RGBA() (r, g, b, a uint32)
}
```

et `color.Model` est une interface déclarée comme

```
type Model interface {
    Convert(c Color) Color
}
```

Accéder à la dimension de l'image et au pixel

Supposons que nous ayons une image stockée en tant que variable `img` , alors nous pouvons obtenir la dimension et l'image en pixel par:

```
// Image bounds and dimension
b := img.Bounds()
width, height := b.Dx(), b.Dy()
// do something with dimension ...

// Corner co-ordinates
top := b.Min.Y
left := b.Min.X
bottom := b.Max.Y
right := b.Max.X
```

```
// Accessing pixel. The (x,y) position must be
// started from (left, top) position not (0,0)
for y := top; y < bottom; y++ {
    for x := left; x < right; x++ {
        cl := img.At(x, y)
        r, g, b, a := cl.RGBA()
        // do something with r,g,b,a color component
    }
}
```

Notez que dans le package, la valeur de chaque composant `R,G,B,A` est comprise entre `0-65535` (`0x0000 - 0xffff`) et **non** `0-255` .

Chargement et sauvegarde de l'image

En mémoire, une image peut être vue comme une matrice de pixel (couleur). Cependant, lorsqu'une image est stockée dans un stockage permanent, elle peut être stockée telle quelle (format RAW), Bitmap ou d'autres formats d'image avec un algorithme de compression particulier pour économiser de l'espace de stockage, par exemple PNG, JPEG, GIF, etc. avec un format particulier, l'image doit être *décodée* en `image.Image` avec l'algorithme correspondant. Une fonction `image.Decode` déclarée comme

```
func Decode(r io.Reader) (Image, string, error)
```

est fourni pour cet usage particulier. Pour pouvoir traiter divers formats d'image, avant d'appeler la fonction `image.Decode` , le décodeur doit être enregistré via la fonction `image.RegisterFormat` définie comme suit:

```
func RegisterFormat(name, magic string,
    decode func(io.Reader) (Image, error), decodeConfig func(io.Reader) (Config, error))
```

Actuellement, le package d'image prend en charge trois formats de fichier: JPEG , GIF et PNG . Pour enregistrer un décodeur, ajoutez ce qui suit

```
import _ "image/jpeg"    //register JPEG decoder
```

au package `main` l'application. Quelque part dans votre code (pas nécessaire dans `main` package `main`), pour charger une image JPEG, utilisez les extraits suivants:

```
f, err := os.Open("inputimage.jpg")
if err != nil {
    // Handle error
}
defer f.Close()

img, fmtName, err := image.Decode(f)
if err != nil {
    // Handle error
}

// `fmtName` contains the name used during format registration
```

```
// Work with `img` ...
```

Enregistrer dans PNG

Pour enregistrer une image dans un format particulier, l' *encodeur* correspondant doit être importé explicitement, c.-à-d.

```
import "image/png"    //needed to use `png` encoder
```

Une image peut alors être enregistrée avec les extraits suivants:

```
f, err := os.Create("outimage.png")
if err != nil {
    // Handle error
}
defer f.Close()

// Encode to `PNG` with `DefaultCompression` level
// then save to file
err = png.Encode(f, img)
if err != nil {
    // Handle error
}
```

Si vous souhaitez spécifier un niveau de compression autre que le niveau `DefaultCompression` , créez un *encodeur* , par exemple

```
enc := png.Encoder{
    CompressionLevel: png.BestSpeed,
}
err := enc.Encode(f, img)
```

Enregistrer au format JPEG

Pour enregistrer au format `jpeg` , utilisez ce qui suit:

```
import "image/jpeg"

// Somewhere in the same package
f, err := os.Create("outimage.jpg")
if err != nil {
    // Handle error
}
defer f.Close()

// Specify the quality, between 0-100
// Higher is better
opt := jpeg.Options{
    Quality: 90,
}
err = jpeg.Encode(f, img, &opt)
if err != nil {
```

```
    // Handle error
}
```

Enregistrer dans GIF

Pour enregistrer l'image dans un fichier GIF, utilisez les extraits suivants.

```
import "image/gif"

// Samewhere in the same package
f, err := os.Create("outimage.gif")
if err != nil {
    // Handle error
}
defer f.Close()

opt := gif.Options {
    NumColors: 256,
    // Add more parameters as needed
}

err = gif.Encode(f, img, &opt)
if err != nil {
    // Handle error
}
```

Image recadrée

La plupart des types d' image dans le package d' image ayant la `SubImage(r Rectangle) Image` image `SubImage(r Rectangle) Image` , sauf `image.Uniform` . Sur cette base, nous pouvons implémenter une fonction pour recadrer une image arbitraire comme suit

```
func CropImage(img image.Image, cropRect image.Rectangle) (cropImg image.Image, newImg bool) {
    //Interface for asserting whether `img`
    //implements SubImage or not.
    //This can be defined globally.
    type CropableImage interface {
        image.Image
        SubImage(r image.Rectangle) image.Image
    }

    if p, ok := img.(CropableImage); ok {
        // Call SubImage. This should be fast,
        // since SubImage (usually) shares underlying pixel.
        cropImg = p.SubImage(cropRect)
    } else if cropRect = cropRect.Intersect(img.Bounds()); !cropRect.Empty() {
        // If `img` does not implement `SubImage`,
        // copy (and silently convert) the image portion to RGBA image.
        rgbaImg := image.NewRGBA(cropRect)
        for y := cropRect.Min.Y; y < cropRect.Max.Y; y++ {
            for x := cropRect.Min.X; x < cropRect.Max.X; x++ {
                rgbaImg.Set(x, y, img.At(x, y))
            }
        }
        cropImg = rgbaImg
        newImg = true
```

```
    } else {
        // Return an empty RGBA image
        cropImg = &image.RGBA{}
        newImg = true
    }

    return cropImg, newImg
}
```

Notez que l'image recadrée peut partager ses pixels sous-jacents avec l'image d'origine. Si tel est le cas, toute modification de l'image recadrée affectera l'image d'origine.

Convertir une image couleur en niveaux de gris

Certains algorithmes de traitement d'images numériques, tels que la détection des contours, les informations portées par l'intensité de l'image (valeur d'échelle de gris) sont suffisants. L'utilisation des informations de couleur (canal `R, G, B`) peut fournir un résultat légèrement meilleur, mais la complexité de l'algorithme sera accrue. Ainsi, dans ce cas, nous devons convertir l'image couleur en image en niveaux de gris avant d'appliquer un tel algorithme.

Le code suivant est un exemple de conversion d'image arbitraire en image en niveaux de gris de 8 bits. L'image est extraite de l'emplacement distant en utilisant `net/http` package `net/http` , convertie en niveaux de gris et finalement enregistrée en tant qu'image PNG.

```
package main

import (
    "image"
    "log"
    "net/http"
    "os"

    _ "image/jpeg"
    "image/png"
)

func main() {
    // Load image from remote through http
    // The Go gopher was designed by Renee French. (http://reneefrench.blogspot.com/)
    // Images are available under the Creative Commons 3.0 Attributions license.
    resp, err := http.Get("http://golang.org/doc/gopher/fiveyears.jpg")
    if err != nil {
        // handle error
        log.Fatal(err)
    }
    defer resp.Body.Close()

    // Decode image to JPEG
    img, _, err := image.Decode(resp.Body)
    if err != nil {
        // handle error
        log.Fatal(err)
    }
    log.Printf("Image type: %T", img)

    // Converting image to grayscale
```

```
    grayImg := image.NewGray(img.Bounds())
    for y := img.Bounds().Min.Y; y < img.Bounds().Max.Y; y++ {
        for x := img.Bounds().Min.X; x < img.Bounds().Max.X; x++ {
            grayImg.Set(x, y, img.At(x, y))
        }
    }

    // Working with grayscale image, e.g. convert to png
    f, err := os.Create("fiveyears_gray.png")
    if err != nil {
        // handle error
        log.Fatal(err)
    }
    defer f.Close()

    if err := png.Encode(f, grayImg); err != nil {
        log.Fatal(err)
    }
}
```

La conversion de couleur se produit lors de l'attribution de pixels via `Set(x, y int, c color.Color)` implémentée dans `image.go` as

```
func (p *Gray) Set(x, y int, c color.Color) {
    if !(Point{x, y}.In(p.Rect)) {
        return
    }

    i := p.PixOffset(x, y)
    p.Pix[i] = color.GrayModel.Convert(c).(color.Gray).Y
}
```

dans lequel `color.GrayModel` est défini dans `color.go` comme

```
func grayModel(c Color) Color {
    if _, ok := c.(Gray); ok {
        return c
    }
    r, g, b, _ := c.RGBA()

    // These coefficients (the fractions 0.299, 0.587 and 0.114) are the same
    // as those given by the JFIF specification and used by func RGBToYCbCr in
    // ycbcr.go.
    //
    // Note that 19595 + 38470 + 7471 equals 65536.
    //
    // The 24 is 16 + 8. The 16 is the same as used in RGBToYCbCr. The 8 is
    // because the return value is 8 bit color, not 16 bit color.
    y := (19595*r + 38470*g + 7471*b + 1<<15) >> 24

    return Gray{uint8(y)}
}
```

Sur la base des faits ci-dessus, l'intensité `Y` est calculée avec la formule suivante:

> Luminance: Y = 0,299 **R** + 0,587 **G** + 0,114 **B**

Si nous voulons appliquer différentes formules / algorithmes pour convertir une couleur en une intésité, par exemple

> Moyenne: Y = (**R** + **G** + **B**) / 3
> Luma: Y = 0,2126 **R** + 0,7152 **G** + 0,0722 **B**
> Lustre: Y = (min (**R** , **G** , **B**) + max (**R** , **G** , **B**)) / 2

Ensuite, les extraits suivants peuvent être utilisés.

```
// Converting image to grayscale
grayImg := image.NewGray(img.Bounds())
for y := img.Bounds().Min.Y; y < img.Bounds().Max.Y; y++ {
    for x := img.Bounds().Min.X; x < img.Bounds().Max.X; x++ {
        R, G, B, _ := img.At(x, y).RGBA()
        //Luma: Y = 0.2126*R + 0.7152*G + 0.0722*B
        Y := (0.2126*float64(R) + 0.7152*float64(G) + 0.0722*float64(B)) * (255.0 / 65535)
        grayPix := color.Gray{uint8(Y)}
        grayImg.Set(x, y, grayPix)
    }
}
```

Le calcul ci-dessus est effectué par multiplication en virgule flottante, et n'est certainement pas le plus efficace, mais il suffit pour démontrer l'idée. L'autre point est, lorsque vous appelez `Set(x, y int, c color.Color)` avec `color.Gray` comme troisième argument, le modèle de couleur n'effectuera pas la conversion de couleur comme on peut le voir dans la fonction `grayModel` précédente.

Lire Images en ligne: https://riptutorial.com/fr/go/topic/10557/images

Chapitre 30: Installation

Examples

Installer sous Linux ou Ubuntu

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential git curl wget
$ wget https://storage.googleapis.com/golang/go<versions>.gz
```

Vous pouvez trouver les listes de versions ici .

```
# To install go1.7 use
$ wget https://storage.googleapis.com/golang/go1.7.linux-amd64.tar.gz


# Untar the file
$ sudo tar -C /usr/local -xzf go1.7.linux-amd64.tar.gz
$ sudo chown -R $USER:$USER /usr/local/go
$ rm go1.5.4.linux-amd64.tar.gz
```

Mise `$GOPATH` jour de `$GOPATH`

```
$ mkdir $HOME/go
```

Ajoutez les deux lignes suivantes à la fin du fichier ~ / .bashrc

```
export GOPATH=$HOME/go
export PATH=$GOPATH/bin:/usr/local/go/bin:$PATH
```

```
$ nano ~/.bashrc
  export GOPATH=$HOME/go
  export PATH=$GOPATH/bin:/usr/local/go/bin:$PATH

$ source ~/.bashrc
```

Maintenant, vous êtes prêt à aller tester votre version avec:

```
$ go version
go version go<version> linux/amd64
```

Lire Installation en ligne: https://riptutorial.com/fr/go/topic/5776/installation

Chapitre 31: Installation

Remarques

Téléchargement Go

Visitez la liste des téléchargements et trouvez l'archive appropriée pour votre système d'exploitation. Les noms de ces téléchargements peuvent être un peu cryptés pour les nouveaux utilisateurs.

Les noms sont au format go [version]. [Système d'exploitation] - [architecture]. [Archive]

Pour la version, vous voulez choisir le plus récent disponible. Celles-ci devraient être les premières options que vous voyez.

Pour le système d'exploitation, cela s'explique assez, sauf pour les utilisateurs de Mac, où le système d'exploitation s'appelle "darwin". Ceci est nommé d'après la partie open-source du système d'exploitation utilisé par les ordinateurs Mac .

Si vous exécutez une machine 64 bits (la plus courante sur les ordinateurs modernes), la partie "architecture" du nom de fichier doit être "amd64". Pour les machines 32 bits, ce sera "386". Si vous êtes sur un périphérique ARM comme un Raspberry Pi, vous voudrez "armv6l".

Pour la partie "archive", les utilisateurs Mac et Windows ont deux options, car Go fournit des programmes d'installation pour ces plates-formes. Pour Mac, vous voulez probablement "pkg". Pour Windows, vous voulez probablement "msi".

Donc, par exemple, si je suis sur une machine Windows 64 bits et que je veux télécharger Go 1.6.3, le téléchargement que je souhaite s'appellera:

```
go1.6.3.windows-amd64.msi
```

Extraire les fichiers téléchargés

Maintenant que nous avons téléchargé une archive Go, nous devons l'extraire quelque part.

Mac et Windows

Les installateurs étant fournis pour ces plates-formes, l'installation est facile. Il suffit de lancer le programme d'installation et d'accepter les paramètres par défaut.

Linux

Il n'y a pas d'installation pour Linux, donc plus de travail est nécessaire. Vous devriez avoir

téléchargé un fichier avec le suffixe ".tar.gz". Ceci est un fichier d'archive, similaire à un fichier ".zip". Nous devons l'extraire. Nous allons extraire les fichiers Go vers `/usr/local` car c'est l'emplacement recommandé.

Ouvrez un terminal et changez de répertoire à l'endroit où vous avez téléchargé l'archive. C'est probablement dans `Downloads` . Sinon, remplacez le répertoire de la commande suivante de manière appropriée.

```
cd Downloads
```

Maintenant, exécutez la commande suivante pour extraire l'archive dans `/usr/local` , en remplaçant `[filename]` par le nom du fichier que vous avez téléchargé.

```
tar -C /usr/local -xzf [filename].tar.gz
```

Définition des variables d'environnement

Il y a encore une étape à franchir avant de commencer à développer. Nous devons définir des variables d'environnement, informations que les utilisateurs peuvent modifier pour donner aux programmes une meilleure idée de la configuration de l'utilisateur.

les fenêtres

Vous devez définir le `GOPATH` , qui est le dossier que vous allez utiliser. Aller travailler

Vous pouvez définir des variables d'environnement via le bouton "Variables d'environnement" de l'onglet "Avancé" du panneau de configuration "Système". Certaines versions de Windows fournissent ce panneau de contrôle via l'option "Paramètres système avancés" du panneau de configuration "Système".

Le nom de votre nouvelle variable d'environnement devrait être "GOPATH". La valeur doit être le chemin complet vers un répertoire dans lequel vous développerez le code Go. Un dossier appelé "go" dans votre répertoire utilisateur est un bon choix.

Mac

Vous devez définir le `GOPATH` , qui est le dossier que vous allez utiliser. Aller travailler

Modifiez un fichier texte nommé ".bash_profile", qui doit se trouver dans votre répertoire utilisateur, et ajoutez la nouvelle ligne suivante à la fin, en remplaçant `[work area]` par un chemin complet vers un répertoire dans lequel vous souhaitez travailler. ".bash_profile" n'existe pas, créez-le. Un dossier appelé "go" dans votre répertoire utilisateur est un bon choix.

```
export GOPATH=[work area]
```

Linux

Comme Linux n'a pas d'installation, cela demande un peu plus de travail. Nous devons montrer au terminal où se trouvent le compilateur Go et les autres outils, et nous devons définir le `GOPATH` , qui est un dossier que vous allez utiliser.

Editez un fichier texte nommé ".profile", qui devrait se trouver dans votre répertoire utilisateur, et ajoutez la ligne suivante à la fin, en remplaçant `[work area]` par un chemin complet vers un répertoire dans lequel vous souhaitez travailler. .profile "n'existe pas, créez-le. Un dossier appelé "go" dans votre répertoire utilisateur est un bon choix.

Ensuite, sur une autre nouvelle ligne, ajoutez ce qui suit à votre fichier ".profile".

```
export PATH=$PATH:/usr/local/go/bin
```

Fini!

Si les outils Go ne sont toujours pas disponibles dans le terminal, essayez de fermer cette fenêtre et d'ouvrir une nouvelle fenêtre de terminal.

Examples

Exemple .profile ou .bash_profile

```
# This is an example of a .profile or .bash_profile for Linux and Mac systems
export GOPATH=/home/user/go
export PATH=$PATH:/usr/local/go/bin
```

Lire Installation en ligne: https://riptutorial.com/fr/go/topic/6213/installation

Chapitre 32: Interfaces

Remarques

Les interfaces dans Go ne sont que des ensembles de méthodes fixes. Un type implémente *implicitement* une interface si son ensemble de méthodes est un sur-ensemble de l'interface. *Il n'y a pas de déclaration d'intention.*

Examples

Interface simple

Dans Go, une interface n'est qu'un ensemble de méthodes. Nous utilisons une interface pour spécifier un comportement d'un objet donné.

```
type Painter interface {
    Paint()
}
```

Le type d'implémentation **n'a pas besoin de** déclarer qu'il implémente l'interface. Il suffit de définir les méthodes de la même signature.

```
type Rembrandt struct{}

func (r Rembrandt) Paint() {
    // use a lot of canvas here
}
```

Maintenant, nous pouvons utiliser la structure comme interface.

```
var p Painter
p = Rembrandt{}
```

Une interface peut être satisfaite (ou implémentée) par un nombre arbitraire de types. De plus, un type peut implémenter un nombre arbitraire d'interfaces.

```
type Singer interface {
     Sing()
}

type Writer interface {
     Write()
}

type Human struct{}

func (h *Human) Sing() {
    fmt.Println("singing")
}
```

```
func (h *Human) Write() {
    fmt.Println("writing")
}



type OnlySinger struct{}
func (o *OnlySinger) Sing() {
    fmt.Println("singing")
}
```

Ici, `Human` structure `Human` satisfait à la fois l'interface `Singer` et `Writer` , mais la structure `OnlySinger` ne satisfait que l'interface `Singer` .

Interface vide

Il existe un type d'interface vide, qui ne contient aucune méthode. Nous le déclarons comme `interface{}` . Ceci ne contient aucune méthode, donc chaque `type` satisfait. L'interface vide peut donc contenir n'importe quelle valeur de type.

```
var a interface{}
var i int = 5
s := "Hello world"

type StructType struct {
    i, j int
    k string
}


// all are valid statements
a = i
a = s
a = &StructType{1, 2, "hello"}
```

Le cas d'utilisation le plus courant des interfaces est de s'assurer qu'une variable prend en charge un ou plusieurs comportements. En revanche, le principal cas d'utilisation de l'interface vide est de définir une variable pouvant contenir n'importe quelle valeur, quel que soit son type concret.

Pour récupérer ces valeurs comme leurs types originaux, il suffit de faire

```
i = a.(int)
s = a.(string)
m := a.(*StructType)
```

ou

```
i, ok := a.(int)
s, ok := a.(string)
m, ok := a.(*StructType)
```

`ok` indique si l' `interface a` est convertible en type donné. S'il est impossible de jeter `ok` sera `false` .

Valeurs d'interface

Si vous déclarez une variable d'une interface, il peut stocker n'importe quel type de valeur qui implémente les méthodes déclarées par l'interface!

Si nous déclarons `h` de l' `interface Singer` , il peut stocker une valeur de type `Human` ou `OnlySinger`. Cela est dû au fait qu'ils implémentent tous des méthodes spécifiées par l'interface `Singer` .

```
var h Singer
h = &human{}

h.Sing()
```

Détermination du type sous-jacent de l'interface

Au départ, il peut parfois être utile de savoir quel type sous-jacent vous avez été transmis. Cela peut être fait avec un commutateur de type. Cela suppose que nous ayons deux structures:

```
type Rembrandt struct{}

func (r Rembrandt) Paint() {}

type Picasso struct{}

func (r Picasso) Paint() {}
```

Cela implémente l'interface Painter:

```
type Painter interface {
    Paint()
}
```

Ensuite, nous pouvons utiliser ce commutateur pour déterminer le type sous-jacent:

```
func WhichPainter(painter Painter) {
    switch painter.(type) {
    case Rembrandt:
        fmt.Println("The underlying type is Rembrandt")
    case Picasso:
        fmt.Println("The underlying type is Picasso")
    default:
        fmt.Println("Unknown type")
    }
}
```

Vérification au moment de la compilation si un type satisfait à une interface

Les interfaces et les implémentations (types implémentant une interface) sont "détachées". Il est donc légitime de vérifier à la compilation si un type implémente une interface.

Une façon de demander au compilateur de vérifier que le type `T` implémente l'interface `I` est en essayant une affectation en utilisant la valeur zéro pour `T` ou pointeur sur `T` , selon le cas. Et nous

pouvons choisir d'affecter à l' identificateur vide pour éviter les déchets inutiles:

```
type T struct{}

var _ I = T{}       // Verify that T implements I.
var _ I = (*T)(nil) // Verify that *T implements I.
```

Si `T` ou `*T` n'implémente pas `I` , ce sera une erreur de compilation.

Cette question apparaît également dans la FAQ officielle: Comment puis-je garantir que mon type satisfait une interface?

Commutateur de type

Les commutateurs de type peuvent également être utilisés pour obtenir une variable correspondant au type de la casse:

```
func convint(v interface{}) (int,error) {
    switch u := v.(type) {
    case int:
        return u, nil
    case float64:
        return int(u), nil
    case string:
        return strconv(u)
    default:
        return 0, errors.New("Unsupported type")
    }
}
```

Type Assertion

Vous pouvez accéder au type de données réel de l'interface avec Type Assertion.

```
interfaceVariable.(DataType)
```

Exemple de struct `MyType` qui implémente l'interface `Subber` :

```
package main

import (
    "fmt"
)

type Subber interface {
    Sub(a, b int) int
}

type MyType struct {
    Msg string
}

//Implement method Sub(a,b int) int
func (m *MyType) Sub(a, b int) int {
```

```
    m.Msg = "SUB!!!"

    return a - b;
}

func main() {
    var interfaceVar Subber = &MyType{}
    fmt.Println(interfaceVar.Sub(6,5))
    fmt.Println(interfaceVar.(*MyType).Msg)
}
```

Sans `.(*MyType)` nous ne pourrions pas accéder à `Msg` Field. Si nous essayons `interfaceVar.Msg` il affichera une erreur de compilation:

```
interfaceVar.Msg undefined (type Subber has no field or method Msg)
```

Aller des interfaces d'un aspect mathématique

En mathématiques, en particulier la *théorie des ensembles* , nous avons une collection de choses appelée *ensemble* et nous les appelons *éléments* . Nous montrons un ensemble avec son nom comme A, B, C, ... ou explicitement en mettant son membre sur la notation entre accolades: {a, b, c, d, e}. Supposons que nous ayons un élément arbitraire x et un ensemble Z. La question clé est: "Comment pouvons-nous comprendre que x est membre de Z ou non?". Mathématicien répond à cette question avec un concept: **Propriété caractéristique** d'un ensemble. *Caractéristique La propriété* d'un ensemble est une expression qui décrit l'ensemble. Par exemple, nous avons un ensemble de *nombres naturels* qui est {0, 1, 2, 3, 4, 5, ...}. Nous pouvons décrire cet ensemble avec cette expression: {a_n | a_0 = 0, a_n = a_{n-1} +1}. Dans la dernière expression a_0 = 0, a_n = a_{n-1} +1 est la propriété caractéristique de l'ensemble des nombres naturels. **Si nous avons cette expression, nous pouvons construire cet ensemble complètement** . Soit décrire l'ensemble des *nombres pairs* de cette manière. Nous savons que cet ensemble est composé des nombres suivants: {0, 2, 4, 6, 8, 10, ...}. En un coup d'œil, nous comprenons que tous ces nombres sont aussi un *nombre naturel* , en d'autres termes, *si nous ajoutons des conditions supplémentaires à la propriété caractéristique des nombres naturels, nous pouvons construire une nouvelle expression décrivant cet ensemble* . On peut donc décrire avec cette expression: {n | n est un membre de nombres naturels *et* le rappel de n sur 2 est zéro}. Maintenant, nous pouvons créer un filtre qui obtient la propriété caractéristique d'un ensemble et filtrer certains éléments souhaités pour renvoyer des éléments de notre ensemble. Par exemple, si nous avons un filtre de nombres naturels, à la fois les nombres naturels et les nombres pairs peuvent passer ce filtre, mais si nous avons un filtre de nombres pairs, certains éléments comme 3 et 137871 ne peuvent pas passer le filtre.

La définition de l'interface dans Go est comme définir la propriété caractéristique et le mécanisme d'utilisation de l'interface car un argument d'une fonction est comme un filtre qui détecte que l'élément est un membre de notre ensemble souhaité ou non. Décrivons cet aspect avec le code:

```
type Number interface {
    IsNumber() bool // the implementation filter "meysam" from 3.14, 2 and 3
}
```

```
type NaturalNumber interface {
    Number
    IsNaturalNumber() bool // the implementation filter 3.14 from 2 and 3
}

type EvenNumber interface {
    NaturalNumber
    IsEvenNumber() bool // the implementation filter 3 from 2
}
```

La propriété caractéristique de `Number` est constituée de toutes les structures qui ont la méthode `IsNumber` . Pour `NaturalNumber` toutes les méthodes `IsNaturalNumber` méthodes `IsNumber` et `IsNaturalNumber` et enfin, pour `EvenNumber` toutes les méthodes `IsEvenNumber` méthodes `IsNumber` , `IsNaturalNumber` et `IsEvenNumber` . Grâce à cette interprétation de l'interface, nous pouvons facilement comprendre que l' `interface{}` ne possédant aucune propriété caractéristique, acceptez tous les types (car elle ne possède pas de filtre pour distinguer les valeurs).

Lire Interfaces en ligne: https://riptutorial.com/fr/go/topic/1221/interfaces

Chapitre 33: Iota

Introduction

Iota permet de déclarer des constantes numériques à partir d'une valeur de départ qui augmente de manière monotone. Iota peut être utilisé pour déclarer des masques de bits souvent utilisés dans la programmation système et réseau et d'autres listes de constantes avec des valeurs associées.

Remarques

L'identifiant `iota` est utilisé pour attribuer des valeurs à des listes de constantes. Lorsque iota est utilisé dans une liste, il commence par une valeur de zéro et s'incrémente d'une valeur pour chaque valeur de la liste de constantes. Il est réinitialisé pour chaque mot clé `const` . Contrairement aux énumérations des autres langues, iota peut être utilisé dans des expressions (par exemple, `iota + 1`), ce qui permet une plus grande flexibilité.

Examples

Utilisation simple de iota

Pour créer une liste de constantes - attribuez une valeur `iota` à chaque élément:

```
const (
  a = iota // a = 0
  b = iota // b = 1
  c = iota // c = 2
)
```

Pour créer une liste de constantes de manière raccourcie, affectez la valeur `iota` au premier élément:

```
const (
  a = iota // a = 0
  b        // b = 1
  c        // c = 2
)
```

Utilisation de iota dans une expression

`iota` peut être utilisé dans des expressions, il peut donc également être utilisé pour affecter des valeurs autres que de simples entiers incrémentés à partir de zéro. Pour créer des constantes pour les unités SI, utilisez cet exemple de Effective Go :

```
type ByteSize float64
```

```
const (
    _           = iota // ignore first value by assigning to blank identifier
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

Valeurs de saut

La valeur de `iota` est toujours incrémentée pour chaque entrée d'une liste de constantes, même si iota n'est pas utilisé:

```
const ( // iota is reset to 0
    a = 1 << iota  // a == 1
    b = 1 << iota  // b == 2
    c = 3          // c == 3  (iota is not used but still incremented)
    d = 1 << iota  // d == 8
)
```

il sera également incrémenté même si aucune constante n'est créée, ce qui signifie que l'identifiant vide peut être utilisé pour ignorer complètement les valeurs:

```
const (
  a = iota // a = 0
  _        // iota is incremented
  b        // b = 2
)
```

Le premier bloc de code a été extrait de Go Spec (CC-BY 3.0).

Utilisation de iota dans une liste d'expressions

Comme `iota` est incrémenté après chaque `ConstSpec` , les valeurs de la même liste d'expressions auront la même valeur pour `iota` :

```
const (
    bit0, mask0 = 1 << iota, 1<<iota - 1  // bit0 == 1, mask0 == 0
    bit1, mask1                           // bit1 == 2, mask1 == 1
    _, _                                  // skips iota == 2
    bit3, mask3                           // bit3 == 8, mask3 == 7
)
```

Cet exemple provient de Go Spec (CC-BY 3.0).

Utilisation de iota dans un masque

Iota peut être très utile lors de la création d'un masque de bits. Par exemple, pour représenter

l'état d'une connexion réseau qui peut être sécurisée, authentifiée et / ou prête, nous pouvons créer un masque comme celui-ci:

```
const (
    Secure = 1 << iota // 0b001
    Authn              // 0b010
    Ready              // 0b100
)

ConnState := Secure|Authn // 0b011: Connection is secure and authenticated, but not yet Ready
```

Utilisation de iota dans const

Ceci est une énumération pour la création de const. Le compilateur Go démarre iota à partir de 0 et s'incrémente d'une unité pour chaque constante suivante. La valeur est déterminée au moment de la compilation plutôt que lors de l'exécution. Pour cette raison, nous ne pouvons pas appliquer iota aux expressions qui sont évaluées au moment de l'exécution.

Programme pour utiliser iota dans const

```
package main

import "fmt"

const (
    Low = 5 * iota
    Medium
    High
)

func main() {
    // Use our iota constants.
    fmt.Println(Low)
    fmt.Println(Medium)
    fmt.Println(High)
}
```

Essayez-le dans Go Playground

Lire Iota en ligne: https://riptutorial.com/fr/go/topic/2865/iota

Chapitre 34: JSON

Syntaxe

- func Marshal (v interface {}) ([] byte, error)
- erreur func Unmarshal (data [] byte, v interface {})

Remarques

Le package `"encoding/json"` Package json implémente le codage et le décodage des objets JSON dans `Go` .

Les types dans JSON avec leurs types concrets correspondants dans Go sont:

Type JSON	Go Type de béton
booléen	bool
Nombres	float64 ou int
chaîne	chaîne
nul	néant

Examples

Encodage JSON de base

`json.Marshal` du package `"encoding/json"` encode une valeur sur JSON.

Le paramètre est la valeur à encoder. Les valeurs renvoyées sont un tableau d'octets représentant l'entrée codée JSON (en cas de succès) et une erreur (en cas d'échec).

```
decodedValue := []string{"foo", "bar"}

// encode the value
data, err := json.Marshal(decodedValue)

// check if the encoding is successful
if err != nil {
    panic(err)
}

// print out the JSON-encoded string
// remember that data is a []byte
fmt.Println(string(data))
// "["foo","bar"]"
```

Cour de récréation

Voici quelques exemples de base de codage pour les types de données intégrés:

```
var data []byte

data, _ = json.Marshal(1)
fmt.Println(string(data))
// 1

data, _ = json.Marshal("1")
fmt.Println(string(data))
// "1"

data, _ = json.Marshal(true)
fmt.Println(string(data))
// true

data, _ = json.Marshal(map[string]int{"London": 18, "Rome": 30})
fmt.Println(string(data))
// {"London":18,"Rome":30}
```

Cour de récréation

Encoder des variables simples est utile pour comprendre comment fonctionne l'encodage JSON dans Go. Cependant, dans la réalité, vous allez probablement encoder des données plus complexes stockées dans des structures .

Décodage JSON de base

`json.Unmarshal` du package `"encoding/json"` décode une valeur JSON dans la valeur indiquée par la variable donnée.

Les paramètres sont la valeur à décoder en `[]bytes` et une variable à utiliser comme stockage pour la valeur désérialisée. La valeur renvoyée est une erreur (en cas d'échec).

```
encodedValue := []byte(`{"London":18,"Rome":30}`)

// generic storage for the decoded JSON
var data map[string]interface{}

// decode the value into data
// notice that we must pass the pointer to data using &data
err := json.Unmarshal(encodedValue, &data)

// check if the decoding is successful
if err != nil {
    panic(err)
}

fmt.Println(data)
map[London:18 Rome:30]
```

Cour de récréation

Remarquez que dans l'exemple ci-dessus, nous connaissions à l'avance le type de la clé et la valeur. Mais ce n'est pas toujours le cas. En fait, dans la plupart des cas, le JSON contient des types de valeurs mixtes.

```
encodedValue := []byte(`{"city":"Rome","temperature":30}`)

// generic storage for the decoded JSON
var data map[string]interface{}

// decode the value into data
if err := json.Unmarshal(encodedValue, &data); err != nil {
    panic(err)
}

// if you want to use a specific value type, we need to cast it
temp := data["temperature"].(float64)
fmt.Println(temp) // 30
city := data["city"].(string)
fmt.Println(city) // "Rome"
```

Cour de récréation

Dans le dernier exemple ci-dessus, nous avons utilisé une carte générique pour stocker la valeur décodée. Nous devons utiliser une `map[string]interface{}` car nous savons que les clés sont des chaînes, mais nous ne connaissons pas le type de leurs valeurs à l'avance.

C'est une approche très simple, mais elle est aussi extrêmement limitée. Dans le monde réel, vous décoderiez généralement un JSON dans un type de `struct` personnalisé .

Décodage des données JSON à partir d'un fichier

Les données JSON peuvent également être lues à partir de fichiers.

Supposons que nous ayons un fichier appelé `data.json` avec le contenu suivant:

```
[
    {
      "Name" : "John Doe",
      "Standard" : 4
    },
    {
      "Name" : "Peter Parker",
      "Standard" : 11
    },
    {
      "Name" : "Bilbo Baggins",
      "Standard" : 150
    }
]
```

L'exemple suivant lit le fichier et décode le contenu:

```
package main
```

```
import (
    "encoding/json"
    "fmt"
    "log"
    "os"
)

type Student struct {
    Name     string
    Standard int `json:"Standard"`
}

func main() {
    // open the file pointer
    studentFile, err := os.Open("data.json")
    if err != nil {
        log.Fatal(err)
    }
    defer studentFile.Close()

    // create a new decoder
    var studentDecoder *json.Decoder = json.NewDecoder(studentFile)
    if err != nil {
        log.Fatal(err)
    }

    // initialize the storage for the decoded data
    var studentList []Student

    // decode the data
    err = studentDecoder.Decode(&studentList)
    if err != nil {
        log.Fatal(err)
    }

    for i, student := range studentList {
        fmt.Println("Student", i+1)
        fmt.Println("Student name:", student.Name)
        fmt.Println("Student standard:", student.Standard)
    }
}
```

Le fichier `data.json` doit se trouver dans le même répertoire que le programme exécutable Go. Lisez la documentation d'E / S sur les fichiers Go pour plus d'informations sur l'utilisation des fichiers dans Go.

Utiliser des structures anonymes pour le décodage

Le but de l'utilisation de structures anonymes est de décoder uniquement les informations qui nous intéressent sans que notre application ne soit gaspillée avec des types utilisés uniquement dans une seule fonction.

```
jsonBlob := []byte(`
  {
    "_total": 1,
    "_links": {
      "self":
```

```
"https://api.twitch.tv/kraken/channels/foo/subscriptions?direction=ASC&limit=25&offset=0",
      "next":
"https://api.twitch.tv/kraken/channels/foo/subscriptions?direction=ASC&limit=25&offset=25"
    },
    "subscriptions": [
      {
        "created_at": "2011-11-23T02:53:17Z",
        "_id": "abcdef0000000000000000000000000000000000",
        "_links": {
          "self": "https://api.twitch.tv/kraken/channels/foo/subscriptions/bar"
        },
        "user": {
          "display_name": "bar",
          "_id": 123456,
          "name": "bar",
          "staff": false,
          "created_at": "2011-06-16T18:23:11Z",
          "updated_at": "2014-10-23T02:20:51Z",
          "logo": null,
          "_links": {
            "self": "https://api.twitch.tv/kraken/users/bar"
          }
        }
      }
    ]
  }
`)

var js struct {
    Total int `json:"_total"`
    Links struct {
        Next string `json:"next"`
    } `json:"_links"`
    Subs []struct {
        Created string `json:"created_at"`
        User    struct {
            Name string `json:"name"`
            ID   int    `json:"_id"`
        } `json:"user"`
    } `json:"subscriptions"`
}

err := json.Unmarshal(jsonBlob, &js)
if err != nil {
    fmt.Println("error:", err)
}
fmt.Printf("%+v", js)
```

Sortie: `{Total:1 Links:{Next:https://api.twitch.tv/kraken/channels/foo/subscriptions?direction=ASC&limit=25&offset=25} Subs:[{Created:2011-11-23T02:53:17Z User:{Name:bar ID:123456}}]}`

Cour de récréation

Pour le cas général, voir aussi:
http://stackoverflow.com/documentation/go/994/json/4111/encoding-decoding-go-structs

Configuration des champs de structure JSON

Prenons l'exemple suivant:

```
type Company struct {
    Name     string
    Location string
}
```

Masquer / Ignorer certains champs

Pour exporter les `Revenue` et les `Sales` , mais les masquer du codage / décodage, utilisez `json:"-"` ou renommez la variable pour commencer par une lettre minuscule. Notez que cela empêche la variable d'être visible en dehors du package.

```
type Company struct {
    Name     string `json:"name"`
    Location string `json:"location"`
    Revenue  int    `json:"-"`
    sales    int
}
```

Ignorer les champs vides

Pour empêcher que `Location` ne soit inclus dans le JSON lorsqu'il est défini sur sa valeur zéro, ajoutez `,omitempty` à la balise `json` .

```
type Company struct {
    Name     string `json:"name"`
    Location string `json:"location,omitempty"`
}
```

Exemple dans un terrain de jeu

Structures de marshaling avec champs privés

En tant que bon développeur, vous avez créé la structure suivante avec les champs exportés et non exportés:

```
type MyStruct struct {
    uuid string
    Name string
}
```

Exemple dans Playground: https://play.golang.org/p/Zk94Il2ANZ

Maintenant, vous voulez que `Marshal()` cette structure en JSON valide pour le stockage dans quelque chose comme etcd. Cependant, depuis que `uuid` n'est pas exporté, `json.Marshal()` ignore. Que faire? Utilisez une structure anonyme et l'interface `json.MarshalJSON()` ! Voici un exemple:

```
type MyStruct struct {
    uuid string
```

```
    Name string
}

func (m MyStruct) MarshalJSON() ([]byte, error {
    j, err := json.Marshal(struct {
        Uuid string
        Name string
    }{
        Uuid: m.uuid,
        Name: m.Name,
    })
    if err != nil {
            return nil, err
    }
    return j, nil
}
```

Exemple dans Playground: https://play.golang.org/p/Bv2k9GgbzE

Encodage / décodage à l'aide des structures Go

Supposons que nous ayons la `struct` suivante qui définit un type de `City` :

```
type City struct {
    Name string
    Temperature int
}
```

Nous pouvons encoder / décoder les valeurs City en utilisant le package `encoding/json` .

Tout d'abord, nous devons utiliser les métadonnées Go pour indiquer à l'encodeur la correspondance entre les champs struct et les clés JSON.

```
type City struct {
    Name string `json:"name"`
    Temperature int `json:"temp"`
    // IMPORTANT: only exported fields will be encoded/decoded
    // Any field starting with a lower letter will be ignored
}
```

Pour garder cet exemple simple, nous déclarons une correspondance explicite entre les champs et les clés. Cependant, vous pouvez utiliser plusieurs variantes des métadonnées `json:` comme expliqué dans les documents .

IMPORTANT: seuls les champs exportés (champs avec nom de base) seront sérialisés / désérialisés. Par exemple, si vous nommez le champ ***t** empérature* il sera ignoré même si vous définissez l' `json` métadonnées.

Codage

Pour encoder une structure `City` , utilisez `json.Marshal` comme dans l'exemple de base:

```
// data to encode
city := City{Name: "Rome", Temperature: 30}

// encode the data
bytes, err := json.Marshal(city)
if err != nil {
    panic(err)
}

fmt.Println(string(bytes))
// {"name":"Rome","temp":30}
```

Cour de récréation

Décodage

Pour décoder une structure `City` , utilisez `json.Unmarshal` comme dans l'exemple de base:

```
// data to decode
bytes := []byte(`{"name":"Rome","temp":30}`)

// initialize the container for the decoded data
var city City

// decode the data
// notice the use of &city to pass the pointer to city
if err := json.Unmarshal(bytes, &city); err != nil {
    panic(err)
}

fmt.Println(city)
// {Rome 30}
```

Cour de récréation

Lire JSON en ligne: https://riptutorial.com/fr/go/topic/994/json

Chapitre 35: JWT Authorization in Go

Introduction

Les jetons Web JSON (JWT) sont une méthode populaire pour représenter des revendications de manière sécurisée entre deux parties. Comprendre comment travailler avec eux est important lors du développement d'applications Web ou d'interfaces de programmation d'applications.

Remarques

context.Context et le middleware HTTP sortent du cadre de ce sujet, mais néanmoins, les curieux et les esprits errants devraient consulter https://github.com/goware/jwtauth , https://github.com/auth0/go-jwt- middleware , et https://github.com/dgrijalva/jwt-go .

Bravo à Dave Grijalva pour son incroyable travail sur go-jwt.

Examples

Analyse et validation d'un jeton à l'aide de la méthode de signature HMAC

```
// sample token string taken from the New example
tokenString :=
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJuYmYiOjE0NDQ0Nzg0MDB9.u1riaD1rW97opCoAuRCTy4w5
Zk-bh7vLiRIsrpU"

// Parse takes the token string and a function for looking up the key. The latter is
especially
// useful if you use multiple keys for your application.  The standard is to use 'kid' in the
// head of the token to identify which key to use, but the parsed token (head and claims) is
provided
// to the callback, providing flexibility.
token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
    // Don't forget to validate the alg is what you expect:
    if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
        return nil, fmt.Errorf("Unexpected signing method: %v", token.Header["alg"])
    }

    // hmacSampleSecret is a []byte containing your secret, e.g. []byte("my_secret_key")
    return hmacSampleSecret, nil
})

if claims, ok := token.Claims.(jwt.MapClaims); ok && token.Valid {
    fmt.Println(claims["foo"], claims["nbf"])
} else {
    fmt.Println(err)
}
```

Sortie:

```
bar 1.4444784e+09
```

(Extrait de la documentation , avec la permission de Dave Grijalva.)

Création d'un jeton à l'aide d'un type de revendications personnalisé

`StandardClaim` est intégré au type personnalisé pour faciliter l'encodage, l'analyse et la validation des revendications standard.

```
tokenString :=
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.HE7fK0xOQwFE


type MyCustomClaims struct {
    Foo string `json:"foo"`
    jwt.StandardClaims
}

// sample token is expired.  override time so it parses as valid
at(time.Unix(0, 0), func() {
    token, err := jwt.ParseWithClaims(tokenString, &MyCustomClaims{}, func(token *jwt.Token)
(interface{}, error) {
        return []byte("AllYourBase"), nil
    })

    if claims, ok := token.Claims.(*MyCustomClaims); ok && token.Valid {
        fmt.Printf("%v %v", claims.Foo, claims.StandardClaims.ExpiresAt)
    } else {
        fmt.Println(err)
    }
})
```

Sortie:

```
bar 15000
```

(Extrait de la documentation , avec la permission de Dave Grijalva.)

Création, signature et codage d'un jeton JWT à l'aide de la méthode de signature HMAC

```
// Create a new token object, specifying signing method and the claims
// you would like it to contain.
token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
    "foo": "bar",
    "nbf": time.Date(2015, 10, 10, 12, 0, 0, 0, time.UTC).Unix(),
})

// Sign and get the complete encoded token as a string using the secret
tokenString, err := token.SignedString(hmacSampleSecret)

fmt.Println(tokenString, err)
```

Sortie:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJuYmYiOjE0NDQ0Nzg0MDB9.u1riaD1rW97opCoAuRCTy4w58
```

```
Zk-bh7vLiRIsrpU <nil>
```

(Extrait de la documentation , avec la permission de Dave Grijalva.)

Utiliser le type StandardClaims seul pour analyser un jeton

Le type `StandardClaims` est conçu pour être intégré dans vos types personnalisés afin de fournir des fonctionnalités de validation standard. Vous pouvez l'utiliser seul, mais il n'y a aucun moyen de récupérer d'autres champs après l'analyse. Voir l'exemple de revendications personnalisées pour l'utilisation prévue.

```
mySigningKey := []byte("AllYourBase")

// Create the Claims
claims := &jwt.StandardClaims{
    ExpiresAt: 15000,
    Issuer:    "test",
}

token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
ss, err := token.SignedString(mySigningKey)
fmt.Printf("%v %v", ss, err)
```

Sortie:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.QsODzZu3lUZMVdhbO76u3Jv02iYCvE
<nil>
```

(Extrait de la documentation , avec la permission de Dave Grijalva.)

Analyse des types d'erreur à l'aide des contrôles de bitfield

```
// Token from another example.  This token is expired
var tokenString =
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.HE7fK0xOQwFEr


token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
    return []byte("AllYourBase"), nil
})

if token.Valid {
    fmt.Println("You look nice today")
} else if ve, ok := err.(*jwt.ValidationError); ok {
    if ve.Errors&jwt.ValidationErrorMalformed != 0 {
        fmt.Println("That's not even a token")
    } else if ve.Errors&(jwt.ValidationErrorExpired|jwt.ValidationErrorNotValidYet) != 0 {
        // Token is either expired or not active yet
        fmt.Println("Timing is everything")
    } else {
        fmt.Println("Couldn't handle this token:", err)
    }
} else {
    fmt.Println("Couldn't handle this token:", err)
```

```
}
```

Sortie:

```
Timing is everything
```

(Extrait de la documentation , avec la permission de Dave Grijalva.)

Obtenir un jeton de l'en-tête d'autorisation HTTP

```
type contextKey string

const (
    // JWTTokenContextKey holds the key used to store a JWT Token in the
    // context.
    JWTTokenContextKey contextKey = "JWTToken"

    // JWTClaimsContextKey holds the key used to store the JWT Claims in the
    // context.
    JWTClaimsContextKey contextKey = "JWTClaims"
)

// ToHTTPContext moves JWT token from request header to context.
func ToHTTPContext() http.RequestFunc {
    return func(ctx context.Context, r *stdhttp.Request) context.Context {
        token, ok := extractTokenFromAuthHeader(r.Header.Get("Authorization"))
        if !ok {
            return ctx
        }

        return context.WithValue(ctx, JWTTokenContextKey, token)
    }
}
```

(À partir du kit / kit , gracieuseté de Peter Bourgon)

Lire JWT Authorization in Go en ligne: https://riptutorial.com/fr/go/topic/10161/jwt-authorization-in-go

Chapitre 36: La gestion des erreurs

Introduction

Dans Go, les situations inattendues sont traitées à l'aide d' **erreurs** et non d'exceptions. Cette approche est plus similaire à celle de C, en utilisant errno, qu'à celle de Java ou d'autres langages orientés objet, avec leurs blocs try / catch. Cependant, une erreur n'est pas un entier mais une interface.

Une fonction susceptible d'échouer renvoie généralement une **erreur** comme dernière valeur de retour. Si cette erreur n'est pas **nulle** , quelque chose ne va pas et l'appelant de la fonction doit agir en conséquence.

Remarques

Notez comment dans Go vous ne *soulevez* pas d'erreur. Au lieu de cela, vous *renvoyez* une erreur en cas d'échec.

Si une fonction peut échouer, la dernière valeur renvoyée est généralement un type d' `error` .

```
// This method doesn't fail
func DoSomethingSafe() {
}

// This method can fail
func DoSomething() (error) {
}

// This method can fail and, when it succeeds,
// it returns a string.
func DoAndReturnSomething() (string, error) {
}
```

Examples

Créer une valeur d'erreur

Le moyen le plus simple de créer une erreur consiste à utiliser le package d' `errors` .

```
errors.New("this is an error")
```

Si vous souhaitez ajouter des informations supplémentaires à une erreur, le package `fmt` fournit également une méthode de création d'erreur utile:

```
var f float64
fmt.Errorf("error with some additional information: %g", f)
```

Voici un exemple complet, où l'erreur est renvoyée par une fonction:

```
package main

import (
    "errors"
    "fmt"
)

var ErrThreeNotFound = errors.New("error 3 is not found")

func main() {
    fmt.Println(DoSomething(1)) // succeeds! returns nil
    fmt.Println(DoSomething(2)) // returns a specific error message
    fmt.Println(DoSomething(3)) // returns an error variable
    fmt.Println(DoSomething(4)) // returns a simple error message
}

func DoSomething(someID int) error {
    switch someID {
    case 3:
        return ErrThreeNotFound
    case 2:
        return fmt.Errorf("this is an error with extra info: %d", someID)
    case 1:
        return nil
    }

    return errors.New("this is an error")
}
```

Open in Playground

Créer un type d'erreur personnalisé

Dans Go, une erreur est représentée par toute valeur pouvant se décrire comme une chaîne. Tout type qui implémente l'interface d' `error` intégrée est une erreur.

```
// The error interface is represented by a single
// Error() method, that returns a string representation of the error
type error interface {
    Error() string
}
```

L'exemple suivant montre comment définir un nouveau type d'erreur à l'aide d'un littéral composé de chaînes.

```
// Define AuthorizationError as composite literal
type AuthorizationError string

// Implement the error interface
// In this case, I simply return the underlying string
func (e AuthorizationError) Error() string {
    return string(e)
}
```

Je peux maintenant utiliser mon type d'erreur personnalisé comme erreur:

```
package main

import (
    "fmt"
)

// Define AuthorizationError as composite literal
type AuthorizationError string

// Implement the error interface
// In this case, I simply return the underlying string
func (e AuthorizationError) Error() string {
    return string(e)
}

func main() {
    fmt.Println(DoSomething(1)) // succeeds! returns nil
    fmt.Println(DoSomething(2)) // returns an error message
}

func DoSomething(someID int) error {
    if someID != 1 {
        return AuthorizationError("Action not allowed!")
    }

    // do something here

    // return a nil error if the execution succeeded
    return nil
}
```

Renvoyer une erreur

Dans Allez vous ne *soulevez* pas d'erreur. Au lieu de cela, vous *renvoyez* une `error` en cas d'échec.

```
// This method can fail
func DoSomething() error {
    // functionThatReportsOK is a side-effecting function that reports its
    // state as a boolean. NOTE: this is not a good practice, so this example
    // turns the boolean value into an error. Normally, you'd rewrite this
    // function if it is under your control.
    if ok := functionThatReportsOK(); !ok {
        return errors.New("functionThatReportsSuccess returned a non-ok state")
    }

    // The method succeeded. You still have to return an error
    // to properly obey to the method signature.
    // But in this case you return a nil error.
    return nil
}
```

Si la méthode retourne plusieurs valeurs (et que l'exécution peut échouer), la convention standard consiste à renvoyer l'erreur comme dernier argument.

```
// This method can fail and, when it succeeds,
// it returns a string.
func DoAndReturnSomething() (string, error) {
    if os.Getenv("ERROR") == "1" {
        return "", errors.New("The method failed")
    }

    s := "Success!"

    // The method succeeded.
    return s, nil
}

result, err := DoAndReturnSomething()
if err != nil {
    panic(err)
}
```

Gérer une erreur

Les erreurs In Go peuvent être renvoyées par un appel de fonction. La convention est que si une méthode peut échouer, le dernier argument renvoyé est une `error` .

```
func DoAndReturnSomething() (string, error) {
    if os.Getenv("ERROR") == "1" {
        return "", errors.New("The method failed")
    }

    // The method succeeded.
    return "Success!", nil
}
```

Vous utilisez plusieurs affectations de variables pour vérifier si la méthode a échoué.

```
result, err := DoAndReturnSomething()
if err != nil {
    panic(err)
}

// This is executed only if the method didn't return an error
fmt.Println(result)
```

Si l'erreur ne vous intéresse pas, vous pouvez simplement l'ignorer en l'attribuant à _ .

```
result, _ := DoAndReturnSomething()
fmt.Println(result)
```

Bien sûr, ignorer une erreur peut avoir de graves conséquences. Par conséquent, ceci n'est généralement pas recommandé.

Si vous avez plusieurs appels de méthode et qu'une ou plusieurs méthodes de la chaîne peuvent renvoyer une erreur, vous devez propager l'erreur au premier niveau capable de la gérer.

```
func Foo() error {
```

```
    return errors.New("I failed!")
}

func Bar() (string, error) {
    err := Foo()
    if err != nil {
        return "", err
    }

    return "I succeeded", nil
}

func Baz() (string, string, error) {
    res, err := Bar()
    if err != nil {
        return "", "", err
    }

    return "Foo", "Bar", nil
}
```

Récupérer de la panique

Une erreur courante est de déclarer une tranche et de commencer à demander des index sans l'initialiser, ce qui conduit à une panique "d'index hors de portée". Le code suivant explique comment se remettre de la panique sans quitter le programme, ce qui est le comportement normal d'une panique. Dans la plupart des cas, le fait de renvoyer une erreur de cette manière plutôt que de quitter le programme en panique n'est utile que pour le développement ou les tests.

```
type Foo struct {
    Is []int
}

func main() {
    fp := &Foo{}
    if err := fp.Panic(); err != nil {
        fmt.Printf("Error: %v", err)
    }
    fmt.Println("ok")
}

func (fp *Foo) Panic() (err error) {
    defer PanicRecovery(&err)
    fp.Is[0] = 5
    return nil
}

func PanicRecovery(err *error) {

    if r := recover(); r != nil {
        if _, ok := r.(runtime.Error); ok {
             //fmt.Println("Panicing")
             //panic(r)
             *err = r.(error)
        } else {
            *err = r.(error)
        }
    }
```

```
}
```

L'utilisation d'une fonction distincte (plutôt que la fermeture) permet de réutiliser la même fonction dans d'autres fonctions sujettes à la panique.

Lire La gestion des erreurs en ligne: https://riptutorial.com/fr/go/topic/785/la-gestion-des-erreurs

Chapitre 37: La vente

Remarques

Le service de vente est une méthode qui garantit que tous les packages tiers que vous utilisez dans votre projet Go sont cohérents pour tous ceux qui développent pour votre application.

Lorsque votre package Go importe un autre package, le compilateur vérifie normalement `$GOPATH/src/` pour le chemin du projet importé. Toutefois , si votre paquet contient un dossier nommé `vendor` , le compilateur vérifiera dans ce dossier en *premier.* Cela signifie que vous pouvez importer des paquets d'autres parties dans votre propre référentiel de codes, sans avoir à modifier leur code.

Vendoring est une fonctionnalité standard de Go 1.6 et versions ultérieures. Dans Go 1.5, vous devez définir la variable d'environnement `GO15VENDOREXPERIMENT=1` pour permettre la vente.

Examples

Utiliser govendor pour ajouter des packages externes

Govendor est un outil utilisé pour importer des packages tiers dans votre référentiel de code d'une manière compatible avec la vente de golang.

Dites par exemple que vous utilisez un package tiers `bosun.org/slog` :

```
package main

import "bosun.org/slog"

func main() {
    slog.Infof("Hello World")
}
```

Votre structure de répertoire peut ressembler à:

```
$GOPATH/src/
├── github.com/me/helloworld/
|   ├── hello.go
├── bosun.org/slog/
|   ├── ... (slog files)
```

Cependant, une personne qui clone `github.com/me/helloworld` peut ne pas avoir `$GOPATH/src/bosun.org/slog/` , ce qui provoque l'échec de *leur* compilation en raison de paquets manquants.

L'exécution de la commande suivante à l'invite de commandes permet de récupérer tous les packages externes de votre package Go et de regrouper les bits requis dans un dossier fournisseur:

```
govendor add +e
```

Cela demande à govendor d'ajouter tous les packages externes dans votre référentiel actuel.

La structure des répertoires de votre application ressemblerait maintenant à:

```
$GOPATH/src/
├── github.com/me/helloworld/
|   ├── vendor/
|   |   ├── bosun.org/slog/
|   |   |   ├── ... (slog files)
|   ├── hello.go
```

et ceux qui clonent votre dépôt vont également récupérer les paquets tiers requis.

Utiliser la corbeille pour gérer ./vendor

`trash` est un outil de vente minimaliste que vous configurez avec le fichier `vendor.conf` . Cet exemple est pour la `trash` elle-même:

```
# package
github.com/rancher/trash

github.com/Sirupsen/logrus                      v0.10.0
github.com/urfave/cli                           v1.18.0
github.com/cloudfoundry-incubator/candiedyaml   99c3df8
https://github.com/imikushin/candiedyaml.git
github.com/stretchr/testify                     v1.1.3
github.com/davecgh/go-spew                      5215b55
github.com/pmezard/go-difflib                   792786c
golang.org/x/sys                                a408501
```

La première ligne sans commentaire est le package que nous gérons ./vendor pour (note: cela peut être littéralement n'importe quel paquet dans votre projet, pas seulement le paquet racine).

Les lignes commentées commencent par `#` .

Chaque ligne non vide et sans commentaire répertorie une dépendance. Seul le package "root" de la dépendance doit être répertorié.

Après le nom du package va la version (commit, tag ou branche) et éventuellement l'URL du référentiel de package (par défaut, il est déduit du nom du package).

Pour remplir votre répertoire ./vendor, vous devez avoir le fichier `vendor.conf` dans le répertoire en cours et exécuter simplement:

```
$ trash
```

La corbeille va cloner les bibliothèques vendues dans `~/.trash-cache` (par défaut), extraire les versions demandées, copier les fichiers dans `./vendor` dir et **élaguer les paquets et les fichiers de test non importés** . Cette dernière étape permet de maintenir votre budget ./vendor à un

niveau raisonnable et de gagner de la place dans votre dépôt de projet.

Note: comme des ordures v0.2.5 est disponible pour Linux et Mac OS, et ne supporte que git pour récupérer les paquets, comme git est le plus populaire, mais nous travaillons sur l' ajout de tous les autres qui `go get` soutien.

Utilisez golang / dep

golang / dep est un outil prototype de gestion des dépendances. Bientôt un outil de versioning officiel. État actuel **Alpha** .

Usage

Obtenez l'outil via

```
$ go get -u github.com/golang/dep/...
```

L'utilisation typique sur un nouveau repo pourrait être

```
$ dep init
$ dep ensure -update
```

Pour mettre à jour une dépendance vers une nouvelle version, vous pouvez exécuter

```
$ dep ensure github.com/pkg/errors@^0.8.0
```

Notez que les formats de fichier manifeste et verrou **ont maintenant été finalisés** . Ceux-ci resteront compatibles même si l'outil change.

vendor.json utilisant l'outil Govendor

```
# It creates vendor folder and vendor.json inside it
govendor init

# Add dependencies in vendor.json
govendor fetch <dependency>

# Usage on new repository
# fetch depenencies in vendor.json
govendor sync
```

Exemple vendor.json

```
{

"comment": "",
"ignore": "test",
"package": [
    {
        "checksumSHA1": "kBeNcaKk56FguvPSUCEaH6AxpRc=",
```

```
        "path": "github.com/golang/protobuf/proto",
        "revision": "2bba0603135d7d7f5cb73b2125beeda19c09f4ef",
        "revisionTime": "2017-03-31T03:19:02Z"
    },
    {
        "checksumSHA1": "1DRAxdlWzS4U0xKN/yQ/fdNN7f0=",
        "path": "github.com/syndtr/goleveldb/leveldb/errors",
        "revision": "8c81ea47d4c41a385645e133e15510fc6a2a74b4",
        "revisionTime": "2017-04-09T01:48:31Z"
    }
],
"rootPath": "github.com/sample"

}
```

Lire La vente en ligne: https://riptutorial.com/fr/go/topic/978/la-vente

Chapitre 38: Le contexte

Syntaxe

- tapez CancelFunc func ()
- func Background () Contexte
- func TODO () Contexte
- func WithCancel (contexte parent) (contexte ctx, annuler CancelFunc)
- func WithDeadline (contexte parent, heure limite.Time) (Contexte, CancelFunc)
- func WithTimeout (contexte parent, timeout time.Duration) (Context, CancelFunc)
- func WithValue (Contexte parent, interface clé {}, interface val {})

Remarques

Le paquet de `context` (dans Go 1.7) ou le paquet `golang.org/x/net/context` (Pre 1.7) est une interface permettant de créer des contextes pouvant être utilisés pour transporter des valeurs et des délais de requête entre les limites des API et entre les services. comme simple implémentation de ladite interface.

à part: le mot "contexte" est généralement utilisé pour désigner l'arbre entier ou des feuilles individuelles dans l'arbre, par exemple. les valeurs `context.Context` réelles.

À un niveau élevé, un contexte est un arbre. De nouvelles feuilles sont ajoutées à l'arborescence lors de leur construction (un `context.Context` avec une valeur parente) et les feuilles ne sont jamais supprimées de l'arborescence. Tout contexte a accès à toutes les valeurs au-dessus (l'accès aux données ne circule que vers le haut) et si un contexte est annulé, ses enfants sont également annulés (les signaux d'annulation sont propagés vers le bas). Le signal d'annulation est implémenté au moyen d'une fonction qui renvoie un canal qui sera fermé (lisible) lorsque le contexte est annulé; Cela rend les contextes très efficaces pour implémenter le modèle de concurrence de pipeline et d'annulation , ou les délais d'attente.

Par convention, les fonctions qui prennent un contexte ont le premier argument `ctx context.Context` . Bien qu'il ne s'agisse que d'une convention, il convient de la suivre, car de nombreux outils d'analyse statique recherchent spécifiquement cet argument. Étant donné que Context est une interface, il est également possible de transformer des données contextuelles existantes (valeurs transmises lors d'une chaîne d'appels) en un contexte Go normal et de les utiliser de manière rétrocompatible en implémentant quelques méthodes. De plus, les contextes sont sécurisés pour un accès simultané, de sorte que vous pouvez les utiliser depuis de nombreuses goroutines (qu'elles s'exécutent sur des threads parallèles ou des coroutines simultanées) sans crainte.

Lectures complémentaires

- https://blog.golang.org/context

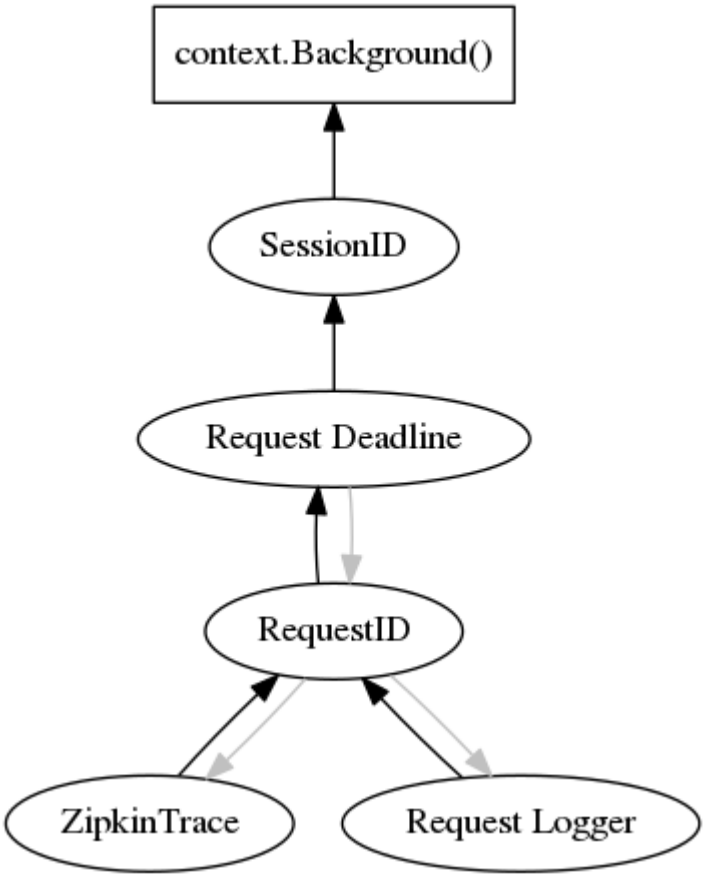
Examples

Arbre de contexte représenté sous forme de graphe orienté

Une simple arborescence de contexte (contenant des valeurs communes pouvant être définies et incluses dans un contexte), construite à partir du code Go, comme suit:

```
// Pseudo-Go
ctx := context.WithValue(
    context.WithDeadline(
        context.WithValue(context.Background(), sidKey, sid),
        time.Now().Add(30 * time.Minute),
    ),
    ridKey, rid,
)
trCtx := trace.NewContext(ctx, tr)
logCtx := myRequestLogging.NewContext(ctx, myRequestLogging.NewLogger())
```

Est-ce un arbre qui peut être représenté comme un graphique dirigé qui ressemble à ceci:



Chaque contexte enfant a accès aux valeurs de ses contextes parents, de sorte que l'accès aux données circule vers le haut dans l'arborescence (représentée par des bords noirs). Les signaux d'annulation, en revanche, descendent dans l'arbre (si un contexte est annulé, tous ses enfants sont également annulés). Le flux du signal d'annulation est représenté par les bords gris.

Utiliser un contexte pour annuler le travail

Passer un contexte avec un timeout (ou avec une fonction d'annulation) à une fonction longue peut être utilisé pour annuler le fonctionnement des fonctions:

```
ctx, _ := context.WithTimeout(context.Background(), 200*time.Millisecond)
for {
    select {
    case <-ctx.Done():
        return ctx.Err()
    default:
        // Do an iteration of some long running work here!
    }
}
```

Lire Le contexte en ligne: https://riptutorial.com/fr/go/topic/2743/le-contexte

Chapitre 39: Le Go Command

Introduction

La commande `go` est un programme en ligne de commande permettant de gérer le développement de Go. Il permet de créer, d'exécuter et de tester du code, ainsi que diverses autres tâches liées à Go.

Examples

Aller courir

`go run` lance un programme sans créer de fichier exécutable. Principalement utile pour le développement. `run` n'exécutera que les packages dont le *nom de package* est **principal** .

Pour démontrer, nous allons utiliser un exemple simple Hello World `main.go` :

```
package main

import fmt

func main() {
    fmt.Println("Hello, World!")
}
```

Exécuter sans compiler dans un fichier:

```
go run main.go
```

Sortie:

```
Hello, World!
```

Exécuter plusieurs fichiers dans un package

Si le package est **principal** et divisé en plusieurs fichiers, il faut inclure les autres fichiers dans la commande `run` :

```
go run main.go assets.go
```

Aller construire

`go build` compilera un programme dans un fichier exécutable.

Pour démontrer, nous allons utiliser un exemple simple Hello World main.go:

```
package main

import fmt

func main() {
    fmt.Println("Hello, World!")
}
```

Compilez le programme:

```
go build main.go
```

`build` crée un programme exécutable, dans ce cas: `main` ou `main.exe` . Vous pouvez alors exécuter ce fichier pour voir le résultat `Hello, World!` . Vous pouvez également le copier sur un système similaire sur lequel Go n'est pas installé, le *rendre exécutable* et l'exécuter.

Spécifiez le système d'exploitation ou l'architecture dans la génération:

Vous pouvez spécifier quel système ou architecture construire en modifiant l' `env` avant la `build` :

```
env GOOS=linux go build main.go # builds for Linux
env GOARCH=arm go build main.go # builds for ARM architecture
```

Construire plusieurs fichiers

Si votre package est divisé en plusieurs fichiers **et que** le nom du package est **principal** (c'est-à-dire *qu'il ne s'agit pas d'un package importable*), vous devez spécifier tous les fichiers à générer:

```
go build main.go assets.go # outputs an executable: main
```

Construire un package

Pour construire un paquet appelé `main` , vous pouvez simplement utiliser:

```
go build . # outputs an executable with name as the name of enclosing folder
```

Aller propre

`go clean` nettoiera tous les fichiers temporaires créés lors de l'appel de `go build` sur un programme. Il va également nettoyer les fichiers laissés par Makefiles.

Aller fmt

`go fmt` formatera le code source d'un programme de manière soignée et idiomatique, facile à lire et à comprendre. Il est recommandé d'utiliser `go fmt` sur n'importe quelle source avant de la

soumettre au public ou de l'engager dans un système de contrôle de version, afin de faciliter sa lecture.

Pour formater un fichier:

```
go fmt main.go
```

Ou tous les fichiers d'un répertoire:

```
go fmt myProject
```

Vous pouvez également utiliser `gofmt -s` (**pas** `go fmt`) pour tenter de simplifier le code possible.

`gofmt` (**not** `go fmt`) peut également être utilisé pour refactoriser le code. Il comprend Go, il est donc plus puissant que d'utiliser une simple recherche et de remplacer. Par exemple, étant donné ce programme (`main.go`):

```
package main

type Example struct {
    Name string
}

func (e *Example) Original(name string) {
    e.Name = name
}

func main() {
    e := &Example{"Hello"}
    e.Original("Goodbye")
}
```

Vous pouvez remplacer la méthode `Original` par `Refactor` avec `gofmt` :

```
gofmt -r 'Original -> Refactor' -d main.go
```

Qui produira le diff:

```
-func (e *Example) Original(name string) {
+func (e *Example) Refactor(name string) {
     e.Name = name
 }

 func main() {
     e := &Example{"Hello"}
-    e.Original("Goodbye")
+    e.Refactor("Goodbye")
 }
```

Va chercher

`go get` télécharge les paquets nommés par les chemins d’importation, avec leurs dépendances. Il

installe ensuite les paquets nommés, comme "go install". Get accepte également les indicateurs de construction pour contrôler l'installation.

> allez chercher github.com/maknahar/phonecountry

Lors de la récupération d'un nouveau package, get crée le répertoire cible `$GOPATH/src/<import-path>` . Si le GOPATH contient plusieurs entrées, get utilise le premier. De même, il installera des binaires compilés dans `$GOPATH/bin` .

Lors de l'extraction ou de la mise à jour d'un package, recherchez une branche ou une balise correspondant à la version de Go installée localement. La règle la plus importante est que si l'installation locale exécute la version "go1", recherchez les branches ou les balises nommées "go1". Si aucune version de ce type n'existe, elle récupère la version la plus récente du package.

Lors de l'utilisation de `go get` , le drapeau `-d` télécharge mais n'installe pas le paquet donné. L'indicateur `-u` permettra de mettre à jour le paquet et ses dépendances.

Obtenir ne vérifie jamais ou met à jour le code stocké dans les répertoires du fournisseur.

Aller env

`go env [var ...]` imprime des informations sur l'environnement.

> Par défaut, il imprime toutes les informations.

```
$go env
```

```
GOARCH="amd64"
GOBIN=""
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/vikashkv/work"
GORACE=""
GOROOT="/usr/local/Cellar/go/1.7.4_1/libexec"
GOTOOLDIR="/usr/local/Cellar/go/1.7.4_1/libexec/pkg/tool/darwin_amd64"
CC="clang"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -Qunused-arguments -fmessage-length=0 -fdebug-prefix-map=/var/folders/xf/t3j24fjd2b7bv8c9gdr_0mj80000gn/T/go-build785167995=/tmp/go-build -gno-record-gcc-switches -fno-common"
CXX="clang++"
CGO_ENABLED="1"
```

> Si un ou plusieurs noms de variables sont donnés en arguments, il imprime la valeur de chaque variable nommée sur sa propre ligne.

```
$go env GOOS GOPATH
```

```
darwin
/Users/vikashkv/work
```

Lire Le Go Command en ligne: https://riptutorial.com/fr/go/topic/4828/le-go-command

Chapitre 40: Lecteurs

Examples

Utilisation de bytes.Reader pour lire une chaîne

Une implémentation de l'interface `io.Reader` peut être trouvée dans le paquet d' `bytes` . Il permet d'utiliser une tranche d'octets comme source pour un lecteur. Dans cet exemple, la tranche d'octets est extraite d'une chaîne, mais il est plus probable qu'elle ait été lue depuis une connexion de fichier ou de réseau.

```
message := []byte("Hello, playground")

reader := bytes.NewReader(message)

bs := make([]byte, 5)
n, err := reader.Read(bs)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("Read %d bytes: %s", n, bs)
```

Aller au terrain de jeu

Lire Lecteurs en ligne: https://riptutorial.com/fr/go/topic/7000/lecteurs

Chapitre 41: Les constantes

Remarques

Go prend en charge les constantes de caractères, de chaînes, de valeurs booléennes et numériques.

Examples

Déclarer une constante

Les constantes sont déclarées comme des variables, mais en utilisant le mot-clé `const` :

```
const Greeting string = "Hello World"
const Years int = 10
const Truth bool = true
```

Comme pour les variables, les noms commençant par une majuscule sont exportés (*public*), les noms commençant par une minuscule ne le sont pas.

```
// not exported
const alpha string = "Alpha"
// exported
const Beta string = "Beta"
```

Les constantes peuvent être utilisées comme toute autre variable, sauf que la valeur ne peut pas être modifiée. Voici un exemple:

```
package main

import (
    "fmt"
    "math"
)

const s string = "constant"

func main() {
    fmt.Println(s) // constant

    // A `const` statement can appear anywhere a `var` statement can.
    const n = 10
    fmt.Println(n)                          // 10
    fmt.Printf("n=%d is of type %T\n", n, n) // n=10 is of type int

    const m float64 = 4.3
    fmt.Println(m) // 4.3

    // An untyped constant takes the type needed by its context.
    // For example, here `math.Sin` expects a `float64`.
    const x = 10
```

```
    fmt.Println(math.Sin(x)) // -0.5440211108893699
}
```

Cour de récréation

Déclaration de constantes multiples

Vous pouvez déclarer plusieurs constantes dans le même bloc `const` :

```
const (
    Alpha = "alpha"
    Beta  = "beta"
    Gamma = "gamma"
)
```

Et incrémenter automatiquement les constantes avec le mot-clé `iota` :

```
const (
    Zero = iota // Zero == 0
    One         // One  == 1
    Two         // Two  == 2
)
```

Pour plus d'exemples d'utilisation de `iota` pour déclarer des constantes, voir Iota .

Vous pouvez également déclarer plusieurs constantes à l'aide de l'affectation multiple. Cependant, cette syntaxe peut être plus difficile à lire et elle est généralement évitée.

```
const Foo, Bar = "foo", "bar"
```

Constantes typées et non typées

Les constantes dans Go peuvent être saisies ou non typées. Par exemple, compte tenu du littéral de chaîne suivant:

```
"bar"
```

on pourrait dire que le type du littéral est `string` , cependant, ceci n'est pas sémantiquement correct. Au lieu de cela, les littéraux sont *des constantes de chaîne non typées* . C'est une chaîne (plus correctement, son *type par défaut* est `string`), mais ce n'est pas une **valeur** Go et n'a donc pas de type tant qu'elle n'est pas affectée ou utilisée dans un contexte tapé. C'est une distinction subtile, mais utile à comprendre.

De même, si nous affectons le littéral à une constante:

```
const foo = "bar"
```

Il reste non typé puisque, par défaut, les constantes ne sont pas typées. Il est possible de le déclarer comme une *chaîne typée constante* également:

```
const typedFoo string = "bar"
```

La différence entre en jeu lorsque nous tentons d'attribuer ces constantes dans un contexte de type. Par exemple, considérez les éléments suivants:

```
var s string
s = foo      // This works just fine
s = typedFoo // As does this

type MyString string
var mys MyString
mys = foo      // This works just fine
mys = typedFoo // cannot use typedFoo (type string) as type MyString in assignment
```

Lire Les constantes en ligne: https://riptutorial.com/fr/go/topic/1047/les-constantes

Chapitre 42: Les fonctions

Introduction

Les fonctions dans Go fournissent un code organisé et réutilisable pour effectuer un ensemble d'actions. Les fonctions simplifient le processus de codage, empêchent la logique redondante et facilitent le suivi du code. Cette rubrique décrit la déclaration et l'utilisation des fonctions, des arguments, des paramètres, des instructions de retour et des étendues dans Go.

Syntaxe

- func () // type de fonction sans arguments et sans valeur de retour
- func (x int) int // accepte un entier et renvoie un entier
- func (a, b int, z float32) bool // accepte 2 entiers, un flottant et renvoie un booléen
- func (chaîne de préfixe, valeurs ... int) // Fonction "variadic" qui accepte une chaîne et un ou plusieurs nombres entiers
- func () (int, bool) // fonction renvoyant deux valeurs
- func (a, b int, z float64, opt ... interface {}) (succès bool) // accepte 2 entiers, un flottant et un ou plusieurs nombres d'interfaces et renvoie une valeur booléenne nommée (qui est déjà initialisée à l'intérieur de la fonction)

Examples

Déclaration de base

Une fonction simple qui n'accepte aucun paramètre et ne renvoie aucune valeur:

```
func SayHello() {
    fmt.Println("Hello!")
}
```

Paramètres

Une fonction peut éventuellement déclarer un ensemble de paramètres:

```
func SayHelloToMe(firstName, lastName string, age int) {
    fmt.Printf("Hello, %s %s!\n", firstName, lastName)
    fmt.Printf("You are %d", age)
}
```

Notez que le type pour `firstName` est omis car il est identique à `lastName` .

Valeurs de retour

Une fonction peut renvoyer une ou plusieurs valeurs à l'appelant:

```
func AddAndMultiply(a, b int) (int, int) {
    return a+b, a*b
}
```

La deuxième valeur de retour peut également être l'erreur var:

```
import errors

func Divide(dividend, divisor int) (int, error) {
    if divisor == 0 {
        return 0, errors.New("Division by zero forbidden")
    }
    return dividend / divisor, nil
}
```

Deux choses importantes doivent être notées:

- La parenthèse peut être omise pour une seule valeur de retour.
- Chaque déclaration de `return` doit fournir une valeur pour **toutes les** valeurs de retour déclarées.

Valeurs de retour nommées

Les valeurs de retour peuvent être affectées à une variable locale. Une déclaration de `return` vide peut alors être utilisée pour renvoyer leurs valeurs actuelles. Ceci est connu comme *"nu"* retour. Les instructions de retour naked ne doivent être utilisées que dans des fonctions courtes car elles nuisent à la lisibilité dans les fonctions plus longues:

```
func Inverse(v float32) (reciprocal float32) {
    if v == 0 {
        return
    }
    reciprocal = 1 / v
    return
}
```

jouer sur le terrain de jeu

```
//A function can also return multiple values
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

jouer sur le terrain de jeu

Deux choses importantes doivent être notées:

- La parenthèse autour des valeurs de retour est **obligatoire** .
- Une déclaration de `return` vide doit toujours être fournie.

Fonctions littérales et fermetures

Une fonction littérale simple, impression `Hello!` à stdout:

```
package main

import "fmt"

func main() {
    func(){
        fmt.Println("Hello!")
    }()
}
```

jouer sur le terrain de jeu

Une fonction littérale, imprimant l'argument `str` à stdout:

```
package main

import "fmt"

func main() {
    func(str string) {
        fmt.Println(str)
    }("Hello!")
}
```

jouer sur le terrain de jeu

Une fonction littérale, fermant la variable `str` :

```
package main

import "fmt"

func main() {
    str := "Hello!"
    func() {
        fmt.Println(str)
    }()
}
```

jouer sur le terrain de jeu

Il est possible d'affecter une fonction littérale à une variable:

```
package main

import (
    "fmt"
```

```
)

func main() {
    str := "Hello!"
    anon := func() {
        fmt.Println(str)
    }
    anon()
}
```

jouer sur le terrain de jeu

Fonctions variadiques

Une fonction variadic peut être appelée avec un nombre quelconque d'arguments de **fin** . Ces éléments sont stockés dans une tranche.

```
package main

import "fmt"

func variadic(strs ...string) {
     // strs is a slice of string
     for i, str := range strs {
         fmt.Printf("%d: %s\n", i, str)
     }
}

func main() {
     variadic("Hello", "Goodbye")
     variadic("Str1", "Str2", "Str3")
}
```

jouer sur le terrain de jeu

Vous pouvez également donner une tranche à une fonction variadic, avec . . . :

```
func main() {
     strs := []string {"Str1", "Str2", "Str3"}

     variadic(strs...)
}
```

jouer sur le terrain de jeu

Lire Les fonctions en ligne: https://riptutorial.com/fr/go/topic/373/les-fonctions

Chapitre 43: Les méthodes

Syntaxe

- func (t T) exampleOne (i int) (n int) {return i} // cette fonction recevra une copie de struct
- func (t * T) exampleTwo (i int) (n int) {return i} // cette méthode recevra le pointeur sur struct et pourra le modifier

Examples

Méthodes de base

Les méthodes dans Go sont comme les fonctions, sauf qu'elles ont un *récepteur* .

Habituellement, le récepteur est une sorte de structure ou de type.

```
package main

import (
    "fmt"
)

type Employee struct {
    Name string
    Age  int
    Rank int
}

func (empl *Employee) Promote() {
    empl.Rank++
}

func main() {

    Bob := new(Employee)

    Bob.Rank = 1
    fmt.Println("Bobs rank now is: ", Bob.Rank)
    fmt.Println("Lets promote Bob!")

    Bob.Promote()

    fmt.Println("Now Bobs rank is: ", Bob.Rank)

}
```

Sortie:

```
Bobs rank now is:  1
Lets promote Bob!
Now Bobs rank is:  2
```

Méthodes de chaînage

Avec les méthodes de golang, vous pouvez utiliser la méthode "chaînant" en passant le pointeur à la méthode et en renvoyant le pointeur sur la même structure comme ceci:

```
package main

import (
    "fmt"
)

type Employee struct {
    Name string
    Age  int
    Rank int
}

func (empl *Employee) Promote() *Employee {
    fmt.Printf("Promoting %s\n", empl.Name)
    empl.Rank++
    return empl
}

func (empl *Employee) SetName(name string) *Employee {
    fmt.Printf("Set name of new Employee to %s\n", name)
    empl.Name = name
    return empl
}

func main() {

    worker := new(Employee)

    worker.Rank = 1

    worker.SetName("Bob").Promote()

    fmt.Printf("Here we have %s with rank %d\n", worker.Name, worker.Rank)

}
```

Sortie:

```
Set name of new Employee to Bob
Promoting Bob
Here we have Bob with rank 2
```

Incrémenter-décrémenter les opérateurs comme arguments dans les méthodes

Bien que Go prenne en charge les opérateurs ++ et - et que le comportement soit presque similaire à c / c ++, les variables avec de tels opérateurs ne peuvent pas être passées en argument pour fonctionner.

```
    package main
```

```
import (
    "fmt"
)

func abcd(a int, b int) {
 fmt.Println(a," ",b)
}
func main() {
    a:=5
    abcd(a++,++a)
}
```

Sortie: erreur de syntaxe: inattendu ++, attend une virgule ou)

Lire Les méthodes en ligne: https://riptutorial.com/fr/go/topic/3890/les-methodes

Chapitre 44: Les variables

Syntaxe

- var x int // déclare la variable x avec le type int
- var s string // déclare la variable s avec le type string
- x = 4 // définir la valeur x
- s = "foo" // Définit la valeur s
- y: = 5 // déclare et définit y inférant son type à int
- f: = 4.5 // déclare et définit f en inférant son type à float64
- b: = "bar" // déclare et définit b en inférant son type sur string

Examples

Déclaration de base des variables

Go est un langage de type statique, ce qui signifie que vous devez généralement déclarer le type des variables que vous utilisez.

```
// Basic variable declaration. Declares a variable of type specified on the right.
// The variable is initialized to the zero value of the respective type.
var x int
var s string
var p Person // Assuming type Person struct {}

// Assignment of a value to a variable
x = 3

// Short declaration using := infers the type
y := 4

u := int64(100)    // declare variable of type int64 and init with 100
var u2 int64 = 100 // declare variable of type int64 and init with 100
```

Affectation de variables multiples

Dans Go, vous pouvez déclarer plusieurs variables en même temps.

```
// You can declare multiple variables of the same type in one line
var a, b, c string

var d, e string = "Hello", "world!"

// You can also use short declaration to assign multiple variables
x, y, z := 1, 2, 3

foo, bar := 4, "stack" // `foo` is type `int`, `bar` is type `string`
```

Si une fonction renvoie plusieurs valeurs, vous pouvez également affecter des valeurs aux

variables en fonction des valeurs de retour de la fonction.

```
func multipleReturn() (int, int) {
    return 1, 2
}

x, y := multipleReturn() // x = 1, y = 2

func multipleReturn2() (a int, b int) {
    a = 3
    b = 4
    return
}

w, z := multipleReturn2() // w = 3, z = 4
```

Identifiant vide

Go va générer une erreur quand il y a une variable non utilisée, afin de vous encourager à écrire un meilleur code. Cependant, il existe des situations où vous n'avez pas besoin d'utiliser une valeur stockée dans une variable. Dans ces cas, vous utilisez un "identifiant vide" _ pour affecter et ignorer la valeur attribuée.

Un identificateur vide peut recevoir une valeur de n'importe quel type et est le plus souvent utilisé dans des fonctions renvoyant plusieurs valeurs.

Valeurs de retour multiples

```
func SumProduct(a, b int) (int, int) {
    return a+b, a*b
}

func main() {
    // I only want the sum, but not the product
    sum, _ := SumProduct(1,2) // the product gets discarded
    fmt.Println(sum) // prints 3
}
```

Utiliser la `range`

```
func main() {

    pets := []string{"dog", "cat", "fish"}

    // Range returns both the current index and value
    // but sometimes you may only want to use the value
    for _, pet := range pets {
        fmt.Println(pet)
    }

}
```

Vérification du type d'une variable

Il existe des situations où vous ne savez pas quel type est une variable lorsqu'elle est renvoyée par une fonction. Vous pouvez toujours vérifier le type d'une variable en utilisant `var.(type)` si vous n'êtes pas sûr du type:

```
x := someFunction() // Some value of an unknown type is stored in x now

switch x := x.(type) {
    case bool:
        fmt.Printf("boolean %t\n", x)             // x has type bool
    case int:
        fmt.Printf("integer %d\n", x)             // x has type int
    case string:
        fmt.Printf("pointer to boolean %s\n", x) // x has type string
    default:
        fmt.Printf("unexpected type %T\n", x)     // %T prints whatever type x is
}
```

Lire Les variables en ligne: https://riptutorial.com/fr/go/topic/674/les-variables

Chapitre 45: Meilleures pratiques sur la structure du projet

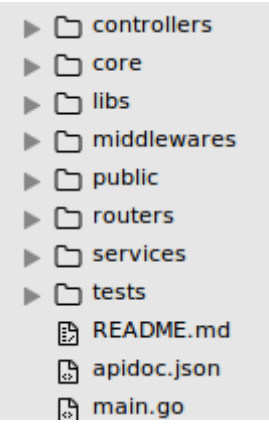
Examples

API Restfull Projects avec Gin

> Gin est un framework web écrit en Golang. Il dispose d'une API de type martini avec des performances bien supérieures, jusqu'à 40 fois plus rapides. Si vous avez besoin de performance et de bonne productivité, vous allez adorer le Gin.

Il y aura 8 paquets + main.go

1. contrôleurs
2. coeur
3. libs
4. middlewares
5. Publique
6. routeurs
7. prestations de service
8. des tests
9. main.go



---

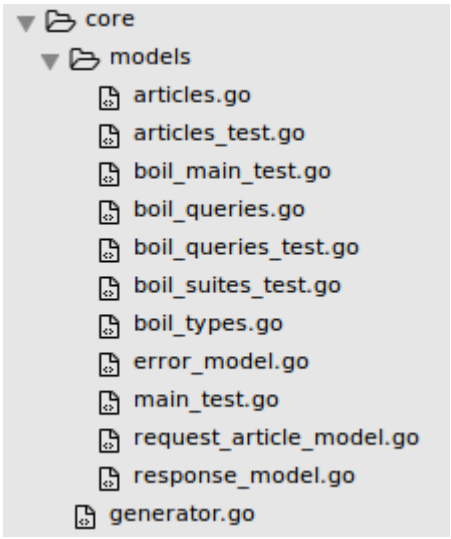
contrôleurs

Le package de contrôleurs stockera toute la logique API. Quelle que soit votre API, votre logique se produira ici

---

coeur

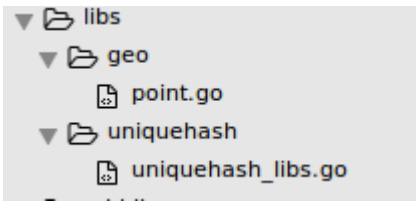
Le package principal stockera tous vos modèles créés, ORM, etc.



---

libs

Ce paquet stockera toute bibliothèque utilisée dans les projets. Mais uniquement pour les bibliothèques créées / importées manuellement, qui ne sont pas disponibles lors de l'utilisation des commandes `go get package_name` . Pourrait être votre propre algorithme de hachage, graphe, arbre, etc.

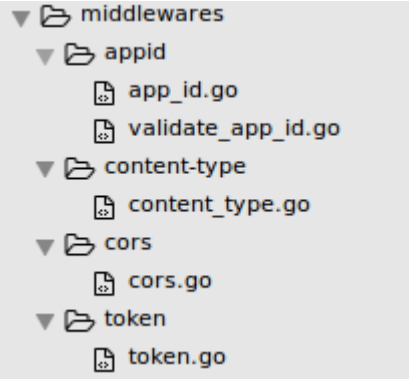


---

middlewares

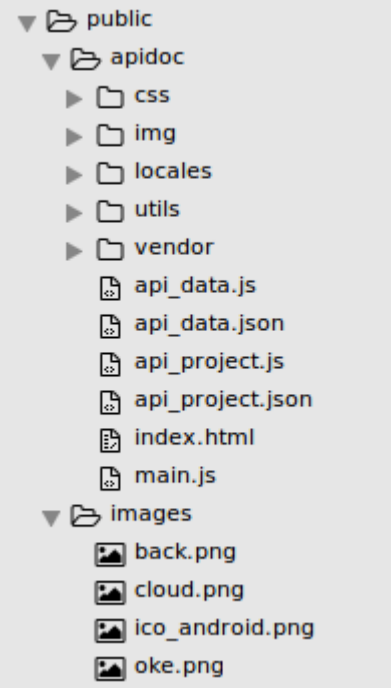
Ce paquet stocke tous les middleware utilisés dans le projet, peut être la création / validation de

cors, device-id, auth etc



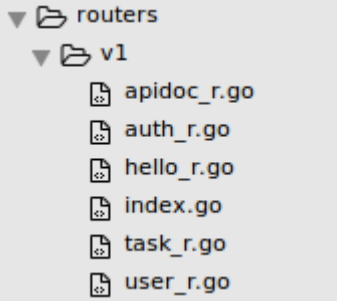
Publique

Ce pacakge stockera tous les fichiers publics et statiques, par exemple html, css, javascript, images, etc.



routeurs

Ce paquet stockera toutes les routes dans votre API REST.



Voir exemple de code pour attribuer les routes.

auth_r.go

```
import (
    auth "simple-api/controllers/v1/auth"
    "gopkg.in/gin-gonic/gin.v1"
)

func SetAuthRoutes(router *gin.RouterGroup) {

/**
 * @api {post} /v1/auth/login Login
 * @apiGroup Users
 * @apiHeader {application/json} Content-Type Accept application/json
 * @apiParam {String} username User username
 * @apiParam {String} password User Password
 * @apiParamExample {json} Input
 *    {
 *      "username": "your username",
 *        "password"     : "your password"
 *    }
 * @apiSuccess {Object} authenticate Response
 * @apiSuccess {Boolean} authenticate.success Status
 * @apiSuccess {Integer} authenticate.statuscode Status Code
 * @apiSuccess {String} authenticate.message Authenticate Message
 * @apiSuccess {String} authenticate.token Your JSON Token
 * @apiSuccessExample {json} Success
 *    {
 *        "authenticate": {
 *               "statuscode": 200,
 *              "success": true,
 *           "message": "Login Successfully",
 *              "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRyc

 *            }
 *      }
 * @apiErrorExample {json} List error
 *    HTTP/1.1 500 Internal Server Error
 */

    router.POST("/auth/login" , auth.Login)
}
```

Si vous voyez, la raison pour laquelle je sépare le gestionnaire est de nous aider à gérer chaque routeur. Je peux donc créer des commentaires sur l'API, qui avec apidoc généreront cela dans la documentation structurée. Ensuite, je vais appeler la fonction dans index.go dans le package actuel

index.go

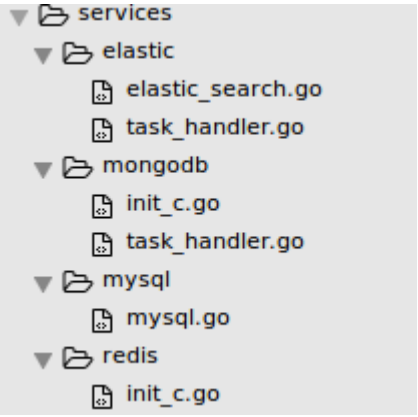
```
package v1


import (
    "gopkg.in/gin-gonic/gin.v1"
    token "simple-api/middlewares/token"
```

```
    appid "simple-api/middlewares/appid"
)
func InitRoutes(g *gin.RouterGroup)  {
    g.Use(appid.AppIDMiddleWare())
    SetHelloRoutes(g)
    SetAuthRoutes(g) // SetAuthRoutes invoked
    g.Use(token.TokenAuthMiddleWare())  //secure the API From this line to bottom with JSON
Auth
    g.Use(appid.ValidateAppIDMiddleWare())
    SetTaskRoutes(g)
    SetUserRoutes(g)
}
```

---

prestations de service

Ce paquet stockera toutes les configurations et tous les paramètres utilisés dans un projet à partir de n'importe quel service utilisé, pouvant être mongodb, redis, mysql, elasticsearch, etc.



---

main.go

L'entrée principale de l'API. Toute configuration concernant les paramètres de l'environnement de développement, les systèmes, le port, etc. sera configurée ici.

Exemple:
main.go

```
package main
import (
    "fmt"
    "net/http"
    "gopkg.in/gin-gonic/gin.v1"
    "articles/services/mysql"
    "articles/routers/v1"
    "articles/core/models"
)

var router  *gin.Engine;

func init() {
    mysql.CheckDB()
```

```
    router = gin.New();
    router.NoRoute(noRouteHandler())
    version1:=router.Group("/v1")
    v1.InitRoutes(version1)

}

func main() {
    fmt.Println("Server Running on Port: ", 9090)
    http.ListenAndServe(":9090",router)
}

func noRouteHandler() gin.HandlerFunc{
    return  func(c *gin.Context) {
    var statuscode     int
    var message        string         = "Not Found"
    var data         interface{} = nil
    var listError [] models.ErrorModel = nil
    var endpoint    string = c.Request.URL.String()
    var method        string = c.Request.Method

    var tempEr models.ErrorModel
    tempEr.ErrorCode     = 4041
    tempEr.Hints         = "Not Found !! \n Routes In Valid. You enter on invalid
Page/Endpoint"
    tempEr.Info            = "visit http://localhost:9090/v1/docs to see the available routes"
    listError             = append(listError,tempEr)
    statuscode             = 404
    responseModel := &models.ResponseModel{
        statuscode,
        message,
        data,
        listError,
        endpoint,
        method,
    }
    var content gin.H = responseModel.NewResponse();
    c.JSON(statuscode,content)
    }
}
```

ps: Chaque code de cet exemple provient de différents projets

---

voir des exemples de projets sur github

Lire Meilleures pratiques sur la structure du projet en ligne:
https://riptutorial.com/fr/go/topic/9463/meilleures-pratiques-sur-la-structure-du-projet

Chapitre 46: mgo

Introduction

mgo (prononcé comme mango) est un pilote MongoDB pour le langage Go qui implémente une sélection riche et bien testée de fonctionnalités sous une API très simple suivant les idiomes Go standard.

Remarques

Documentation API

[https://gopkg.in/mgo.v2][1]

Examples

Exemple

```
package main

import (
        "fmt"
        "log"
        "gopkg.in/mgo.v2"
        "gopkg.in/mgo.v2/bson"
)

type Person struct {
        Name string
        Phone string
}

func main() {
        session, err := mgo.Dial("server1.example.com,server2.example.com")
        if err != nil {
                panic(err)
        }
        defer session.Close()

        // Optional. Switch the session to a monotonic behavior.
        session.SetMode(mgo.Monotonic, true)

        c := session.DB("test").C("people")
        err = c.Insert(&Person{"Ale", "+55 53 8116 9639"},
                   &Person{"Cla", "+55 53 8402 8510"})
        if err != nil {
                log.Fatal(err)
        }

        result := Person{}
        err = c.Find(bson.M{"name": "Ale"}).One(&result)
        if err != nil {
```

```
                log.Fatal(err)
        }

        fmt.Println("Phone:", result.Phone)
}
```

Lire mgo en ligne: https://riptutorial.com/fr/go/topic/8898/mgo

Chapitre 47: Middleware

Introduction

In Go Middleware peut être utilisé pour exécuter du code avant et après la fonction de gestionnaire. Il utilise la puissance des interfaces à fonction unique. Peut être introduit à tout moment sans affecter les autres middleware. Pour Ex: la journalisation de l'authentification peut être ajoutée aux étapes ultérieures du développement sans perturber le code existant.

Remarques

La **signature du middleware** doit être (http.ResponseWriter, * http.Request), c'est-à-dire du type **http.handlerFunc** .

Examples

Fonction de gestionnaire normale

```
func loginHandler(w http.ResponseWriter, r *http.Request) {
            // Steps to login
}


func main() {
    http.HandleFunc("/login", loginHandler)
    http.ListenAndServe(":8080", nil)
}
```

Middleware Calcule le temps requis pour que handlerFunc s'exécute

```
// logger middlerware that logs time taken to process each request
func Logger(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        startTime := time.Now()
        h.ServeHttp(w,r)
        endTime := time.Since(startTime)
        log.Printf("%s %d %v", r.URL, r.Method, endTime)
    })
}

func loginHandler(w http.ResponseWriter, r *http.Request) {
            // Steps to login
}


func main() {
    http.HandleFunc("/login", Logger(loginHandler))
    http.ListenAndServe(":8080", nil)
}
```

Middleware CORS

```
func CORS(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        origin := r.Header.Get("Origin")
        w.Header().Set("Access-Control-Allow-Origin", origin)
        if r.Method == "OPTIONS" {
            w.Header().Set("Access-Control-Allow-Credentials", "true")
            w.Header().Set("Access-Control-Allow-Methods", "GET,POST")

            w.RespWriter.Header().Set("Access-Control-Allow-Headers", "Content-Type, X-CSRF-
Token, Authorization")
            return
        } else {
            h.ServeHTTP(w, r)
        }
    })
}

func main() {
    http.HandleFunc("/login", Logger(CORS(loginHandler)))
    http.ListenAndServe(":8080", nil)
}
```

Auth Middleware

```
func Authenticate(h http.Handler) http.Handler {
    return CustomHandlerFunc(func(w *http.ResponseWriter, r *http.Request) {
        // extract params from req
        // post params | headers etc
        if CheckAuth(params) {
            log.Println("Auth Pass")
            // pass control to next middleware in chain or handler func
            h.ServeHTTP(w, r)
        } else {
            log.Println("Auth Fail")
            // Responsd Auth Fail
        }
    })
}
```

Gestionnaire de récupération pour empêcher le serveur de tomber en panne

```
func Recovery(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request){
        defer func() {
            if err := recover(); err != nil {
                // respondInternalServerError
            }
        }()
        h.ServeHTTP(w , r)
    })
}
```

Lire Middleware en ligne: https://riptutorial.com/fr/go/topic/9343/middleware

Chapitre 48: Modèles

Syntaxe

- t, err: = template.Parse (`{{.MyName .MyAge}}`)
- t.Execute (os.Stdout, struct {MyValue, MyAge string} {"John Doe", "40.1"})

Remarques

Golang fournit des paquets comme:

1. `text/template`
2. `html/template`

mettre en œuvre des modèles basés sur des données pour générer des sorties textuelles et HTML.

Examples

Valeurs de sortie de la variable struct à la sortie standard à l'aide d'un modèle de texte

```
package main

import (
    "log"
    "text/template"
    "os"
)

type Person struct{
    MyName string
    MyAge int
}

var myTempContents string= `
This person's name is : {{.MyName}}
And he is {{.MyAge}} years old.
`

func main() {
    t,err := template.New("myTemp").Parse(myTempContents)
    if err != nil{
        log.Fatal(err)
    }
    myPersonSlice := []Person{ {"John Doe",41},{"Peter Parker",17} }
    for _,myPerson := range myPersonSlice{
        t.Execute(os.Stdout,myPerson)
    }
```

```
}
```

Cour de récréation

Définition de fonctions pour appeler depuis un template

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "text/template"
)

var requestTemplate string = `
{{range $i, $url := .URLs}}
{{ $url }} {{(status_code $url)}}
{{ end }}`

type Requests struct {
    URLs []string
}

func main() {
    var fns = template.FuncMap{
        "status_code": func(x string) int {
            resp, err := http.Head(x)
            if err != nil {
                return -1
            }
            return resp.StatusCode
        },
    }

    req := new(Requests)
    req.URLs = []string{"http://godoc.org", "http://stackoverflow.com", "http://linux.org"}

    tmpl := template.Must(template.New("getBatch").Funcs(fns).Parse(requestTemplate))
    err := tmpl.Execute(os.Stdout, req)
    if err != nil {
        fmt.Println(err)
    }
}
```

Ici, nous utilisons notre fonction définie `status_code` pour obtenir le code d'état de la page Web directement à partir du modèle.

Sortie:

```
http://godoc.org 200

http://stackoverflow.com 200

http://linux.org 200
```

Lire Modèles en ligne: https://riptutorial.com/fr/go/topic/1402/modeles

Chapitre 49: Mutex

Examples

Verrouillage mutex

Le verrouillage mutex dans Go vous permet de vous assurer qu'un seul goroutine à la fois possède un verrou:

```
import "sync"

func mutexTest() {
    lock := sync.Mutex{}
    go func(m *sync.Mutex) {
        m.Lock()
        defer m.Unlock()   // Automatically unlock when this function returns
        // Do some things
    }(&lock)

    lock.Lock()
    // Do some other things
    lock.Unlock()
}
```

L'utilisation d'un `Mutex` vous permet d'éviter les conditions de concurrence, les modifications simultanées et d'autres problèmes associés à plusieurs routines simultanées fonctionnant sur les mêmes ressources. Notez que `Mutex.Unlock()` peut être exécuté par n'importe quelle routine, pas seulement la routine qui a obtenu le verrou. Notez également que l'appel à `Mutex.Lock()` n'échouera pas si une autre routine contient le verrou; il bloquera jusqu'à ce que le verrou soit libéré.

Astuce: chaque fois que vous transmettez une variable Mutex à une fonction, transmettez-la toujours en tant que pointeur. Sinon, une copie est faite de votre variable, ce qui va à l'encontre du but du Mutex. Si vous utilisez une ancienne version de Go (<1.7), le compilateur ne vous avertira pas de cette erreur!

Lire Mutex en ligne: https://riptutorial.com/fr/go/topic/2607/mutex

Chapitre 50: Panique et récupérer

Remarques

Cet article suppose la connaissance de Defer Basics

Pour la gestion des erreurs ordinaires, lisez la rubrique sur la gestion des erreurs

Examples

Panique

Une panique interrompt le flux d'exécution normal et quitte la fonction en cours. Tous les appels différés seront alors exécutés avant que le contrôle ne soit transmis à la fonction suivante de la pile. La fonction de chaque pile sortira et lancera des appels différés jusqu'à ce que la panique soit gérée à l'aide d'une `recover()` différée `recover()` ou jusqu'à ce que la panique atteigne `main()` et termine le programme. Si cela se produit, l'argument fourni à paniquer et une trace de pile seront imprimés sur `stderr` .

```
package main

import "fmt"

func foo() {
    defer fmt.Println("Exiting foo")
    panic("bar")
}

func main() {
    defer fmt.Println("Exiting main")
    foo()
}
```

Sortie:

```
Exiting foo
Exiting main
panic: bar


goroutine 1 [running]:
panic(0x128360, 0x1040a130)
    /usr/local/go/src/runtime/panic.go:481 +0x700
main.foo()
    /tmp/sandbox550159908/main.go:7 +0x160
main.main()
    /tmp/sandbox550159908/main.go:12 +0x120
```

Il est important de noter que la `panic` acceptera tout type comme paramètre.

Récupérer

Récupérer comme son nom l'indique peut tenter de se remettre d'une `panic` . La récupération *doit* être tentée dans une instruction différée car le flux d'exécution normal a été arrêté. L'instruction de `recover` doit apparaître *directement* dans l'enceinte de la fonction différée. Les instructions de récupération dans les fonctions appelées par des appels de fonctions différés ne seront pas honorées. L'appel `recover()` renverra l'argument fourni à la panique initiale, si le programme est en train de paniquer. Si le programme n'est pas en train de paniquer, `recover()` renverra `nil` .

```
package main

import "fmt"

func foo() {
    panic("bar")
}

func bar() {
    defer func() {
        if msg := recover(); msg != nil {
            fmt.Printf("Recovered with message %s\n", msg)
        }
    }()
    foo()
    fmt.Println("Never gets executed")
}

func main() {
    fmt.Println("Entering main")
    bar()
    fmt.Println("Exiting main the normal way")
}
```

Sortie:

```
Entering main
Recovered with message bar
Exiting main the normal way
```

Lire Panique et récupérer en ligne: https://riptutorial.com/fr/go/topic/4350/panique-et-recuperer

Chapitre 51: Paquets

Examples

Initalisation des colis

Package peut avoir des méthodes d' `init` qui ne sont exécutées **qu'une seule fois** avant main.

```
package usefull

func init() {
    // init code
}
```

Si vous souhaitez simplement exécuter l'initialisation du package sans faire référence à quoi que ce soit, utilisez l'expression d'importation suivante.

```
import _ "usefull"
```

Gestion des dépendances de package

Un moyen courant de télécharger des dépendances Go consiste à utiliser la commande `go get <package>` , qui enregistre le paquet dans le répertoire global / shared `$GOPATH/src` . Cela signifie qu'une seule version de chaque package sera liée à chaque projet qui l'inclut en tant que dépendance. Cela signifie également que lorsqu'un nouveau développeur déploie votre projet, il va `go get` la dernière version de chaque dépendance.

Cependant, vous pouvez garder l'environnement de construction cohérent en attachant toutes les dépendances d'un projet dans le répertoire `vendor/` . Conserver les dépendances vendues avec le référentiel de votre projet vous permet de gérer les versions de dépendance par projet et de fournir un environnement cohérent pour votre génération.

Voici à quoi ressemblera la structure de votre projet:

```
$GOPATH/src/
├── github.com/username/project/
|   ├── main.go
|   ├── vendor/
|   |   ├── github.com/pkg/errors
|   |   ├── github.com/gorilla/mux
```

Utiliser un nom de paquet et de dossier différent

Il est parfaitement correct d'utiliser un nom de package autre que le nom du dossier. Si nous le faisons, nous devons toujours importer le package en fonction de la structure du répertoire, mais après l'importation, nous devons nous référer au nom utilisé dans la clause du package.

Par exemple, si vous avez un dossier `$GOPATH/src/mypck` , et que nous y avons un fichier `a.go` :

```
package apple

const Pi = 3.14
```

Utiliser ce package:

```
package main

import (
    "mypck"
    "fmt"
)

func main() {
    fmt.Println(apple.Pi)
}
```

Même si cela fonctionne, vous devriez avoir une bonne raison de dévier le nom du package du nom du dossier (ou cela peut devenir une source de malentendu et de confusion).

A quoi ça sert?

Simple. Un nom de package est un identifiant Go:

```
identifier = letter { letter | unicode_digit } .
```

Ce qui permet d'utiliser des lettres Unicode dans les identifiants, par exemple `αβ` est un identifiant valide dans Go. Les noms de dossiers et de fichiers ne sont pas gérés par Go mais par le système d'exploitation, et différents systèmes de fichiers ont des restrictions différentes. Il existe en fait de nombreux systèmes de fichiers qui ne permettraient pas à tous les identificateurs Go valides de devenir des noms de dossiers. Vous ne pourriez donc pas nommer vos packages autrement que ce que la spécification de langage autoriserait.

Ayant la possibilité d'utiliser des noms de paquetages différents de leurs dossiers, vous avez la possibilité de nommer vraiment vos paquets en fonction des spécifications du langage, quel que soit le système d'exploitation et de fichiers sous-jacent.

Importation de paquets

Vous pouvez importer un seul package avec l'instruction:

```
import "path/to/package"
```

ou regrouper plusieurs importations ensemble:

```
import (
    "path/to/package1"
    "path/to/package2"
```

```
)
```

Cela va chercher dans les correspondants `import` chemins à l' intérieur du `$GOPATH` pour `.go` fichiers et vous permet d' accéder aux noms exportés par `packagename.AnyExportedName` .

Vous pouvez également accéder aux packages locaux à l'intérieur du dossier en cours en préfaçant les packages avec `./` . Dans un projet avec une structure comme celle-ci:

```
project
├── src
│   ├── package1
│   │   └── file1.go
│   └── package2
│       └── file2.go
└── main.go
```

Vous pouvez appeler cela dans `main.go` pour importer le code dans `file1.go` et `file2.go` :

```
import (
    "./src/package1"
    "./src/package2"
)
```

Étant donné que les noms de paquetages peuvent entrer en conflit dans différentes bibliothèques, vous souhaiterez peut-être attribuer un nom à un seul paquet. Vous pouvez le faire en préfixant votre instruction import avec le nom que vous souhaitez utiliser.

```
import (
    "fmt" //fmt from the standardlibrary
    tfmt "some/thirdparty/fmt" //fmt from some other library
)
```

Cela vous permet d'accéder à l'ancien paquetage `fmt` en utilisant `fmt.*` Et le dernier paquetage `fmt` en utilisant `tfmt.*` .

Vous pouvez également importer le package dans le propre espace de noms afin de pouvoir vous référer aux noms exportés sans le `package.` préfixe utilisant un seul point comme alias:

```
import (
    . "fmt"
)
```

L'exemple ci-dessus importe `fmt` dans l'espace de noms global et vous permet d'appeler, par exemple, `Printf` directement: Playground

Si vous importez un paquet mais n'utilisez aucun de ses noms exportés, le compilateur Go imprimera un message d'erreur. Pour contourner cela, vous pouvez définir l'alias sur le trait de soulignement:

```
import (
    _ "fmt"
```

```
)
```

Cela peut être utile si vous n'accédez pas directement à ce paquet mais que vous avez besoin de ses fonctions `init` pour les exécuter.

Remarque:

Comme les noms de package sont basés sur la structure des dossiers, toute modification des noms de dossier et des références d'importation (y compris la casse) provoquera une erreur de compilation "collision à l'importation insensible à la casse" sous Linux et OS-X. et corriger (le message d'erreur est un peu mystérieux pour les simples mortels car il essaie de transmettre le contraire - la comparaison a échoué en raison de la sensibilité à la casse).

ex: "path / to / Package1" vs "path / to / package1"

Exemple en direct: https://github.com/akamai-open/AkamaiOPEN-edgegrid-golang/issues/2

Lire Paquets en ligne: https://riptutorial.com/fr/go/topic/401/paquets

Chapitre 52: Plans

Introduction

Les cartes sont des types de données utilisés pour stocker des paires clé-valeur non ordonnées, de sorte que rechercher la valeur associée à une clé donnée est très efficace. Les clés sont uniques. La structure de données sous-jacente se développe au besoin pour accueillir de nouveaux éléments, de sorte que le programmeur n'a pas besoin de s'inquiéter de la gestion de la mémoire. Ils sont similaires à ce que d'autres langages appellent des tables de hachage, des dictionnaires ou des tableaux associatifs.

Syntaxe

- var mapName map [KeyType] ValueType // déclare une carte
- var mapName = map [KeyType] ValueType {} // déclare et attribue une carte vide
- var mapName = map [KeyType] ValueType {key1: val1, key2: val2} // déclare et attribue une carte
- mapName: = make (mappe [KeyType] ValueType) // déclare et initialise la carte de taille par défaut
- mapName: = make (mappe [KeyType] ValueType, longueur) // déclare et initialise la carte de taille de *longueur*
- mapName: = map [KeyType] ValueType {} // déclare automatiquement et attribue une map vide avec: =
- mapName: = map [KeyType] ValueType {key1: value1, key2: value2} // déclare automatiquement et assigne une carte avec: =
- value: = mapName [clé] // Récupère la valeur par clé
- value, hasKey: = mapName [clé] // Récupère la valeur par clé, 'hasKey' est 'true' si la clé existe dans la carte
- mapName [clé] = valeur // Définir la valeur par clé

Remarques

Go fournit un type de `map` intégré qui implémente une *table de hachage* . Les cartes sont le type de données associatif intégré de Go (également appelé *hachage* ou *dictionnaires* dans d'autres langues).

Examples

Déclarer et initialiser une carte

Vous définissez une carte à l'aide du mot-clé `map` , suivi des types de ses clés et de ses valeurs:

```
// Keys are ints, values are ints.
var m1 map[int]int // initialized to nil
```

```
// Keys are strings, values are ints.
var m2 map[string]int // initialized to nil
```

Les cartes sont des types de référence et, une fois définies, elles ont une *valeur* `nil` . Écrit sur des cartes nulles paniquera et les lectures renverront toujours la valeur zéro.

Pour initialiser une carte, utilisez la fonction `make` :

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

Avec la forme de `make` deux paramètres, il est possible de spécifier une capacité d'entrée initiale pour la carte, en remplaçant la capacité par défaut:

```
m := make(map[string]int, 30)
```

Alternativement, vous pouvez déclarer une carte, l'initialiser à sa valeur zéro, puis lui attribuer une valeur littérale plus tard, ce qui aide si vous regroupez la structure en json, produisant ainsi une carte vide au retour.

```
m := make(map[string]int, 0)
```

Vous pouvez également créer une carte et définir sa valeur initiale entre accolades (`{}`).

```
var m map[string]int = map[string]int{"Foo": 20, "Bar": 30}

fmt.Println(m["Foo"]) // outputs 20
```

Toutes les instructions suivantes entraînent la liaison de la variable à la même valeur.

```
// Declare, initializing to zero value, then assign a literal value.
var m map[string]int
m = map[string]int{}

// Declare and initialize via literal value.
var m = map[string]int{}

// Declare via short variable declaration and initialize with a literal value.
m := map[string]int{}
```

Nous pouvons également utiliser un *littéral de carte* pour créer une nouvelle carte avec des paires clé / valeur initiales .

Le type de clé peut être de tout type comparable ; Cela exclut notamment les fonctions, les cartes et les tranches . Le type de valeur peut être n'importe quel type, y compris les types personnalisés ou l' `interface{}` .

```
type Person struct {
    FirstName string
    LastName  string
```

```
}

// Declare via short variable declaration and initialize with make.
m := make(map[string]Person)

// Declare, initializing to zero value, then assign a literal value.
var m map[string]Person
m = map[string]Person{}

// Declare and initialize via literal value.
var m = map[string]Person{}

// Declare via short variable declaration and initialize with a literal value.
m := map[string]Person{}
```

Créer une carte

On peut déclarer et initialiser une carte dans une seule instruction en utilisant un *littéral composite* .

Utilisation du type automatique Déclaration de variable courte:

```
mapIntInt := map[int]int{10: 100, 20: 100, 30: 1000}
mapIntString := map[int]string{10: "foo", 20: "bar", 30: "baz"}
mapStringInt := map[string]int{"foo": 10, "bar": 20, "baz": 30}
mapStringString := map[string]string{"foo": "one", "bar": "two", "baz": "three"}
```

Le même code, mais avec des types de variables:

```
var mapIntInt = map[int]int{10: 100, 20: 100, 30: 1000}
var mapIntString = map[int]string{10: "foo", 20: "bar", 30: "baz"}
var mapStringInt = map[string]int{"foo": 10, "bar": 20, "baz": 30}
var mapStringString = map[string]string{"foo": "one", "bar": "two", "baz": "three"}
```

Vous pouvez également inclure vos propres structures dans une carte:

Vous pouvez utiliser des types personnalisés comme valeur:

```
// Custom struct types
type Person struct {
  FirstName, LastName string
}

var mapStringPerson = map[string]Person{
  "john": Person{"John", "Doe"},
  "jane": Person{"Jane", "Doe"}}
mapStringPerson := map[string]Person{
  "john": Person{"John", "Doe"},
  "jane": Person{"Jane", "Doe"}}
```

Votre struct peut également être la *clé* de la carte:

```
type RouteHit struct {
    Domain string
```

```
    Route  string
}

var hitMap = map[RouteHit]int{
  RouteHit{"example.com","/home"}: 1,
  RouteHit{"example.com","/help"}: 2}
hitMap := map[RouteHit]int{
  RouteHit{"example.com","/home"}: 1,
  RouteHit{"example.com","/help"}: 2}
```

Vous pouvez créer une carte vide simplement en ne saisissant aucune valeur entre crochets `{}` .

```
mapIntInt := map[int]int{}
mapIntString := map[int]string{}
mapStringInt := map[string]int{}
mapStringString := map[string]string{}
mapStringPerson := map[string]Person{}
```

Vous pouvez créer et utiliser une carte directement sans avoir à l'assigner à une variable. Cependant, vous devrez spécifier la déclaration et le contenu.

```
// using a map as argument for fmt.Println()
fmt.Println(map[string]string{
  "FirstName": "John",
  "LastName": "Doe",
  "Age": "30"})

// equivalent to
data := map[string]string{
  "FirstName": "John",
  "LastName": "Doe",
  "Age": "30"}
fmt.Println(data)
```

Valeur zéro d'une carte

La valeur zéro d'une `map` est `nil` et a une longueur de `0` .

```
var m map[string]string
fmt.Println(m == nil) // true
fmt.Println(len(m) ==0) // true
```

Une carte `nil` n'a pas de clé et les clés ne peuvent pas être ajoutées. Une carte `nil` se comporte comme une carte vide si elle est lue mais provoque une panique à l'exécution si elle est écrite.

```
var m map[string]string

// reading
m["foo"] == "" // true. Remember "" is the zero value for a string
_, ok = m["foo"] // ok == false

// writing
m["foo"] = "bar" // panic: assignment to entry in nil map
```

Vous ne devriez pas essayer de lire ou d'écrire sur une carte de valeur zéro. Au lieu de cela, initialisez la carte (avec `make` ou affectation) avant de l'utiliser.

```
var m map[string]string
m = make(map[string]string) // OR m = map[string]string{}
m["foo"] = "bar"
```

Itérer les éléments d'une carte

```
import fmt

people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for key, value := range people {
  fmt.Println("Name:", key, "Age:", value)
}
```

Notez que lors d'une itération sur une carte avec une boucle d'intervalle, l'ordre d'itération n'est pas spécifié et n'est pas garanti pour être identique d'une itération à l'autre.

Vous pouvez également ignorer les clés ou les valeurs de la carte, si vous souhaitez simplement saisir des touches ou saisir des valeurs.

Itérer les clés d'une carte

```
people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for key, _ := range people {
  fmt.Println("Name:", key)
}
```

Si vous cherchez simplement les clés, car elles sont la première valeur, vous pouvez simplement déposer le trait de soulignement:

```
for key := range people {
  fmt.Println("Name:", key)
}
```

Notez que lors d'une itération sur une carte avec une boucle d'intervalle, l'ordre d'itération n'est pas spécifié et n'est pas garanti pour être identique d'une itération à l'autre.

Supprimer un élément de carte

La `delete` fonction intégrée supprime l'élément avec la clé spécifiée sur une carte.

```
people := map[string]int{"john": 30, "jane": 29}
fmt.Println(people) // map[john:30 jane:29]

delete(people, "john")
fmt.Println(people) // map[jane:29]
```

Si la `map` est `nil` ou qu'il n'ya pas d'élément de ce type, `delete` n'a aucun effet.

```
people := map[string]int{"john": 30, "jane": 29}
fmt.Println(people) // map[john:30 jane:29]

delete(people, "notfound")
fmt.Println(people) // map[john:30 jane:29]

var something map[string]int
delete(something, "notfound") // no-op
```

Compter les éléments de la carte

La fonction intégrée `len` renvoie le nombre d'éléments dans une `map`

```
m := map[string]int{}
len(m) // 0

m["foo"] = 1
len(m) // 1
```

Si une variable pointe vers une carte `nil` , alors `len` renvoie 0.

```
var m map[string]int
len(m) // 0
```

Accès simultané aux cartes

Les cartes in go ne sont pas sûres pour la concurrence. Vous devez prendre un verrou pour lire et écrire dessus si vous y accédez simultanément. Généralement, la meilleure option consiste à utiliser `sync.RWMutex` car vous pouvez avoir des verrous en lecture et en écriture. Cependant, un `sync.Mutex` pourrait également être utilisé.

```
type RWMap struct {
    sync.RWMutex
    m map[string]int
}

// Get is a wrapper for getting the value from the underlying map
func (r RWMap) Get(key string) int {
    r.RLock()
    defer r.RUnlock()
    return r.m[key]
}
```

```
// Set is a wrapper for setting the value of a key in the underlying map
func (r RWMap) Set(key string, val int) {
    r.Lock()
    defer r.Unlock()
    r.m[key] = val
}

// Inc increases the value in the RWMap for a key.
//   This is more pleasant than r.Set(key, r.Get(key)++)
func (r RWMap) Inc(key string) {
    r.Lock()
    defer r.Unlock()
    r.m[key]++
}

func main() {

    // Init
    counter := RWMap{m: make(map[string]int)}

    // Get a Read Lock
    counter.RLock()
    _ = counter.["Key"]
    counter.RUnlock()

    // the above could be replaced with
    _ = counter.Get("Key")

    // Get a write Lock
    counter.Lock()
    counter.m["some_key"]++
    counter.Unlock()

    // above would need to be written as
    counter.Inc("some_key")
}
```

Le compromis entre les fonctions d'encapsulation se situe entre l'accès public de la carte sous-jacente et l'utilisation correcte des verrous appropriés.

Création de cartes avec des tranches en tant que valeurs

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

L'accès à une clé inexistante renverra une tranche nulle en tant que valeur. Puisque les tranches nuls agissent comme des tranches de longueur zéro lorsqu'elles sont utilisées avec `append` ou d'autres fonctions intégrées, vous n'avez normalement pas besoin de vérifier si une clé existe:

```
// m["key1"] == nil && len(m["key1"]) == 0
m["key1"] = append(m["key1"], 1)
// len(m["key1"]) == 1
```

La suppression d'une clé de la carte ramène la clé à une tranche nulle:

```
delete(m, "key1")
```

```
// m["key1"] == nil
```

Vérifier l'élément dans une carte

Pour obtenir une valeur de la carte, il suffit de faire quelque chose comme: 00

```
value := mapName[ key ]
```

Si la carte contient la clé, elle renvoie la valeur correspondante.
Sinon, il renvoie la valeur zéro du type de valeur de la carte (`0` si la carte des valeurs `int` , `""` si la carte des valeurs de `string` ...)

```
m  := map[string]string{"foo": "foo_value", "bar": ""}
k  := m["foo"]  // returns "foo_value" since that is the value stored in the map
k2 := m["bar"]  // returns "" since that is the value stored in the map
k3 := m["nop"]  // returns "" since the key does not exist, and "" is the string type's zero
value
```

Pour différencier les valeurs vides des clés inexistantes, vous pouvez utiliser la seconde valeur renvoyée de l'accès à la carte (en utilisant la `value, hasKey := map["key"]` like `value, hasKey := map["key"]`).

Cette seconde valeur est `boolean` et sera:

- `true` lorsque la valeur est dans la carte,
- `false` lorsque la carte ne contient pas la clé donnée.

Regardez l'exemple suivant:

```
value, hasKey = m[ key ]
if hasKey {
    // the map contains the given key, so we can safely use the value
    // If value is zero-value, it's because the zero-value was pushed to the map
} else {
    // The map does not have the given key
    // the value will be the zero-value of the map's type
}
```

Itérer les valeurs d'une carte

```
people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for _, value := range people {
  fmt.Println("Age:", value)
}
```

Notez que lors d'une itération sur une carte avec une boucle d'intervalle, l'ordre d'itération n'est

pas spécifié et n'est pas garanti pour être identique d'une itération à l'autre.

Copier une carte

Comme les tranches, les cartes contiennent des **références** à une structure de données sous-jacente. Donc, en assignant sa valeur à une autre variable, seule la référence sera transmise. Pour copier la carte, il est nécessaire de créer une autre carte et de copier chaque valeur:

```
// Create the original map
originalMap := make(map[string]int)
originalMap["one"] = 1
originalMap["two"] = 2

// Create the target map
targetMap := make(map[string]int)

// Copy from the original map to the target map
for key, value := range originalMap {
  targetMap[key] = value
}
```

Utiliser une carte comme un ensemble

Certaines langues ont une structure native pour les ensembles. Pour créer un ensemble dans Go, il est recommandé d'utiliser une carte du type valeur de l'ensemble dans une structure vide (`map[Type]struct{}`).

Par exemple, avec des chaînes:

```
// To initialize a set of strings:
greetings := map[string]struct{}{
    "hi":    {},
    "hello": {},
}

// To delete a value:
delete(greetings, "hi")

// To add a value:
greetings["hey"] = struct{}{}

// To check if a value is in the set:
if _, ok := greetings["hey"]; ok {
    fmt.Println("hey is in greetings")
}
```

Lire Plans en ligne: https://riptutorial.com/fr/go/topic/732/plans

Chapitre 53: Pointeurs

Syntaxe

- pointeur: = & variable // obtient le pointeur de la variable
- variable: = * pointeur // obtient la variable du pointeur
- * pointeur = valeur // définir la valeur de la variable via le pointeur
- pointeur: = new (Struct) // récupère le pointeur de la nouvelle structure

Examples

Pointeurs de base

Go prend en charge les pointeurs , vous permettant de transmettre des références aux valeurs et aux enregistrements de votre programme.

```
package main

import "fmt"

// We'll show how pointers work in contrast to values with
// 2 functions: `zeroval` and `zeroptr`. `zeroval` has an
// `int` parameter, so arguments will be passed to it by
// value. `zeroval` will get a copy of `ival` distinct
// from the one in the calling function.
func zeroval(ival int) {
    ival = 0
}

// `zeroptr` in contrast has an `*int` parameter, meaning
// that it takes an `int` pointer. The `*iptr` code in the
// function body then _dereferences_ the pointer from its
// memory address to the current value at that address.
// Assigning a value to a dereferenced pointer changes the
// value at the referenced address.
func zeroptr(iptr *int) {
    *iptr = 0
}
```

Une fois ces fonctions définies, vous pouvez effectuer les opérations suivantes:

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

    zeroval(i)
    fmt.Println("zeroval:", i) // zeroval: 1
    // `i` is still equal to 1 because `zeroval` edited
    // a "copy" of `i`, not the original.

    // The `&i` syntax gives the memory address of `i`,
    // i.e. a pointer to `i`. When calling `zeroptr`,
```

```
    // it will edit the "original" `i`.
    zeroptr(&i)
    fmt.Println("zeroptr:", i) // zeroptr: 0

    // Pointers can be printed too.
    fmt.Println("pointer:", &i) // pointer: 0x10434114
}
```

Essayez ce code

Pointeur v. Méthodes de valeur

Méthodes de pointeur

Les méthodes de pointage peuvent être appelées même si la variable n'est pas elle-même un pointeur.

Selon le Go Spec ,

> . . . une référence à une méthode sans interface avec un récepteur de pointeur utilisant une valeur adressable prendra automatiquement l'adresse de cette valeur: `t.Mp` est équivalent à `(&t).Mp` .

Vous pouvez voir cela dans cet exemple:

```
package main

import "fmt"

type Foo struct {
    Bar int
}

func (f *Foo) Increment() {
    f.Bar += 1
}

func main() {
    var f Foo

    // Calling `f.Increment` is automatically changed to `(&f).Increment` by the compiler.
    f = Foo{}
    fmt.Printf("f.Bar is %d\n", f.Bar)
    f.Increment()
    fmt.Printf("f.Bar is %d\n", f.Bar)

    // As you can see, calling `(&f).Increment` directly does the same thing.
    f = Foo{}
    fmt.Printf("f.Bar is %d\n", f.Bar)
    (&f).Increment()
    fmt.Printf("f.Bar is %d\n", f.Bar)
}
```

Joue-le

Méthodes de valeur

Comme pour les méthodes de pointage, les méthodes de valeur peuvent être appelées même si la variable n'est pas elle-même une valeur.

Selon le Go Spec ,

> . . . une référence à une méthode sans interface avec un récepteur de valeur utilisant un pointeur déréférencera automatiquement ce pointeur: `pt.Mv` est équivalent à `(*pt).Mv` .

Vous pouvez voir cela dans cet exemple:

```
package main

import "fmt"

type Foo struct {
    Bar int
}

func (f Foo) Increment() {
    f.Bar += 1
}

func main() {
    var p *Foo

    // Calling `p.Increment` is automatically changed to `(*p).Increment` by the compiler.
    // (Note that `*p` is going to remain at 0 because a copy of `*p`, and not the original
`*p` are being edited)
    p = &Foo{}
    fmt.Printf("(*p).Bar is %d\n", (*p).Bar)
    p.Increment()
    fmt.Printf("(*p).Bar is %d\n", (*p).Bar)

    // As you can see, calling `(*p).Increment` directly does the same thing.
    p = &Foo{}
    fmt.Printf("(*p).Bar is %d\n", (*p).Bar)
    (*p).Increment()
    fmt.Printf("(*p).Bar is %d\n", (*p).Bar)
}
```

Joue-le

Pour en savoir plus sur les méthodes de pointeur et de valeur, consultez la section Spécification de la méthode sur les valeurs de la méthode , ou consultez la section Mise en oeuvre effective concernant Pointers v. Values .

Note 1: La parenthèse (`()`) autour de `*p` et `&f` avant que les sélecteurs comme `.Bar` soient `.Bar` à des fins de regroupement et doivent être conservés.

Note 2: Bien que les pointeurs puissent être convertis en valeurs (et vice-versa) lorsqu'ils sont les récepteurs d'une méthode, ils ne *sont* pas *convertis automatiquement entre eux lorsqu'ils sont des arguments à l'intérieur d'une fonction.*

Dereferencing Pointers

Les pointeurs peuvent être **déréférencés** en ajoutant un astérisque * avant un pointeur.

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
}

func main() {
    c := new(Person) // returns pointer
    c.Name = "Catherine"
    fmt.Println(c.Name) // prints: Catherine
    d := c
    d.Name = "Daniel"
    fmt.Println(c.Name) // prints: Daniel
    // Adding an Asterix before a pointer dereferences the pointer
    i := *d
    i.Name = "Ines"
    fmt.Println(c.Name) // prints: Daniel
    fmt.Println(d.Name) // prints: Daniel
    fmt.Println(i.Name) // prints: Ines
}
```

Les tranches sont des pointeurs vers des segments de tableau

Les tranches sont des **pointeurs** vers les tableaux, avec la longueur du segment et sa capacité. Ils se comportent comme des pointeurs et en attribuant leur valeur à une autre tranche, ils attribueront l'adresse mémoire. Pour **copier** une valeur de tranche dans une autre, utilisez la fonction de **copie** intégrée: `func copy(dst, src []Type) int` (renvoie la quantité d'éléments copiés).

```
package main

import (
    "fmt"
)

func main() {
    x := []byte{'a', 'b', 'c'}
    fmt.Printf("%s", x)       // prints: abc
    y := x
    y[0], y[1], y[2] = 'x', 'y', 'z'
    fmt.Printf("%s", x)       // prints: xyz
    z := make([]byte, len(x))
    // To copy the value to another slice, but
    // but not the memory address use copy:
    _ = copy(z, x)            // returns count of items copied
```

```
    fmt.Printf("%s", z)       // prints: xyz
    z[0], z[1], z[2] = 'a', 'b', 'c'
    fmt.Printf("%s", x)       // prints: xyz
    fmt.Printf("%s", z)       // prints: abc
}
```

Pointeurs simples

```
func swap(x, y *int) {
  *x, *y = *y, *x
}

func main() {
  x := int(1)
  y := int(2)
  // variable addresses
  swap(&x, &y)
  fmt.Println(x, y)
}
```

Lire Pointeurs en ligne: https://riptutorial.com/fr/go/topic/1239/pointeurs

Chapitre 54: Pool de mémoire

Introduction

sync.Pool stocke un cache des éléments alloués mais inutilisés pour une utilisation ultérieure, évitant ainsi le désabonnement de la mémoire pour les collections fréquemment modifiées, et permettant une réutilisation efficace et sécurisée de la mémoire. Il est utile de gérer un groupe d'éléments temporaires partagés entre des clients concurrents d'un package, par exemple une liste de connexions de base de données ou une liste de tampons de sortie.

Examples

sync.Pool

En `sync.Pool` structure `sync.Pool` , nous pouvons regrouper des objets et les réutiliser.

```
package main

import (
    "bytes"
    "fmt"
    "sync"
)

var pool = sync.Pool{
    // New creates an object when the pool has nothing available to return.
    // New must return an interface{} to make it flexible. You have to cast
    // your type after getting it.
    New: func() interface{} {
        // Pools often contain things like *bytes.Buffer, which are
        // temporary and re-usable.
        return &bytes.Buffer{}
    },
}

func main() {
    // When getting from a Pool, you need to cast
    s := pool.Get().(*bytes.Buffer)
    // We write to the object
    s.Write([]byte("dirty"))
    // Then put it back
    pool.Put(s)

    // Pools can return dirty results

    // Get 'another' buffer
    s = pool.Get().(*bytes.Buffer)
    // Write to it
    s.Write([]bytes("append"))
    // At this point, if GC ran, this buffer *might* exist already, in
    // which case it will contain the bytes of the string "dirtyappend"
    fmt.Println(s)
    // So use pools wisely, and clean up after yourself
```

```
    s.Reset()
    pool.Put(s)

    // When you clean up, your buffer should be empty
    s = pool.Get().(*bytes.Buffer)
    // Defer your Puts to make sure you don't leak!
    defer pool.Put(s)
    s.Write([]byte("reset!"))
    // This prints "reset!", and not "dirtyappendreset!"
    fmt.Println(s)
}
```

Lire Pool de mémoire en ligne: https://riptutorial.com/fr/go/topic/4647/pool-de-memoire

Chapitre 55: Pools de travailleurs

Examples

Pool de travailleurs simple

Une implémentation simple du pool de travail:

```
package main

import (
    "fmt"
    "sync"
)

type job struct {
    // some fields for your job type
}

type result struct {
    // some fields for your result type
}

func worker(jobs <-chan job, results chan<- result) {
    for j := range jobs {
        var r result
        // do some work
        results <- r
    }
}

func main() {
    // make our channels for communicating work and results
    jobs := make(chan job, 100) // 100 was chosen arbitrarily
    results := make(chan result, 100)

    // spin up workers and use a sync.WaitGroup to indicate completion
    wg := sync.WaitGroup
    for i := 0; i < runtime.NumCPU; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            worker(jobs, results)
        }()
    }

    // wait on the workers to finish and close the result channel
    // to signal downstream that all work is done
    go func() {
        defer close(results)
        wg.Wait()
    }()

    // start sending jobs
    go func() {
        defer close(jobs)
```

```
        for {
            jobs <- getJob()   // I haven't defined getJob() and noMoreJobs()
            if noMoreJobs() {  // they are just for illustration
                break
            }
        }
    }()

    // read all the results
    for r := range results {
        fmt.Println(r)
    }
}
```

File d'attente avec le pool de travail

Une file d'attente qui gère un pool de travail, utile pour effectuer des tâches telles que le traitement en arrière-plan sur les serveurs Web:

```
package main

import (
  "fmt"
  "runtime"
  "strconv"
  "sync"
  "time"
)

// Job - interface for job processing
type Job interface {
  Process()
}

// Worker - the worker threads that actually process the jobs
type Worker struct {
  done             sync.WaitGroup
  readyPool        chan chan Job
  assignedJobQueue chan Job

  quit chan bool
}

// JobQueue - a queue for enqueueing jobs to be processed
type JobQueue struct {
  internalQueue     chan Job
  readyPool         chan chan Job
  workers           []*Worker
  dispatcherStopped sync.WaitGroup
  workersStopped    sync.WaitGroup
  quit              chan bool
}

// NewJobQueue - creates a new job queue
func NewJobQueue(maxWorkers int) *JobQueue {
  workersStopped := sync.WaitGroup{}
  readyPool := make(chan chan Job, maxWorkers)
  workers := make([]*Worker, maxWorkers, maxWorkers)
  for i := 0; i < maxWorkers; i++ {
```

```
    workers[i] = NewWorker(readyPool, workersStopped)
  }
  return &JobQueue{
    internalQueue:     make(chan Job),
    readyPool:         readyPool,
    workers:           workers,
    dispatcherStopped: sync.WaitGroup{},
    workersStopped:    workersStopped,
    quit:              make(chan bool),
  }
}

// Start - starts the worker routines and dispatcher routine
func (q *JobQueue) Start() {
  for i := 0; i < len(q.workers); i++ {
    q.workers[i].Start()
  }
  go q.dispatch()
}

// Stop - stops the workers and sispatcher routine
func (q *JobQueue) Stop() {
  q.quit <- true
  q.dispatcherStopped.Wait()
}

func (q *JobQueue) dispatch() {
  q.dispatcherStopped.Add(1)
  for {
    select {
    case job := <-q.internalQueue: // We got something in on our queue
      workerChannel := <-q.readyPool // Check out an available worker
      workerChannel <- job           // Send the request to the channel
    case <-q.quit:
      for i := 0; i < len(q.workers); i++ {
        q.workers[i].Stop()
      }
      q.workersStopped.Wait()
      q.dispatcherStopped.Done()
      return
    }
  }
}

// Submit - adds a new job to be processed
func (q *JobQueue) Submit(job Job) {
  q.internalQueue <- job
}

// NewWorker - creates a new worker
func NewWorker(readyPool chan chan Job, done sync.WaitGroup) *Worker {
  return &Worker{
    done:             done,
    readyPool:        readyPool,
    assignedJobQueue: make(chan Job),
    quit:             make(chan bool),
  }
}

// Start - begins the job processing loop for the worker
func (w *Worker) Start() {
```

```
  go func() {
    w.done.Add(1)
    for {
      w.readyPool <- w.assignedJobQueue // check the job queue in
      select {
      case job := <-w.assignedJobQueue: // see if anything has been assigned to the queue
        job.Process()
      case <-w.quit:
        w.done.Done()
        return
      }
    }
  }()
}

// Stop - stops the worker
func (w *Worker) Stop() {
  w.quit <- true
}

//////////////// Example /////////////////

// TestJob - holds only an ID to show state
type TestJob struct {
  ID string
}

// Process - test process function
func (t *TestJob) Process() {
  fmt.Printf("Processing job '%s'\n", t.ID)
  time.Sleep(1 * time.Second)
}

func main() {
  queue := NewJobQueue(runtime.NumCPU())
  queue.Start()
  defer queue.Stop()

  for i := 0; i < 4*runtime.NumCPU(); i++ {
    queue.Submit(&TestJob{strconv.Itoa(i)})
  }
}
```

Lire Pools de travailleurs en ligne: https://riptutorial.com/fr/go/topic/4182/pools-de-travailleurs

Chapitre 56: Premiers pas avec Go en utilisant Atom

Introduction

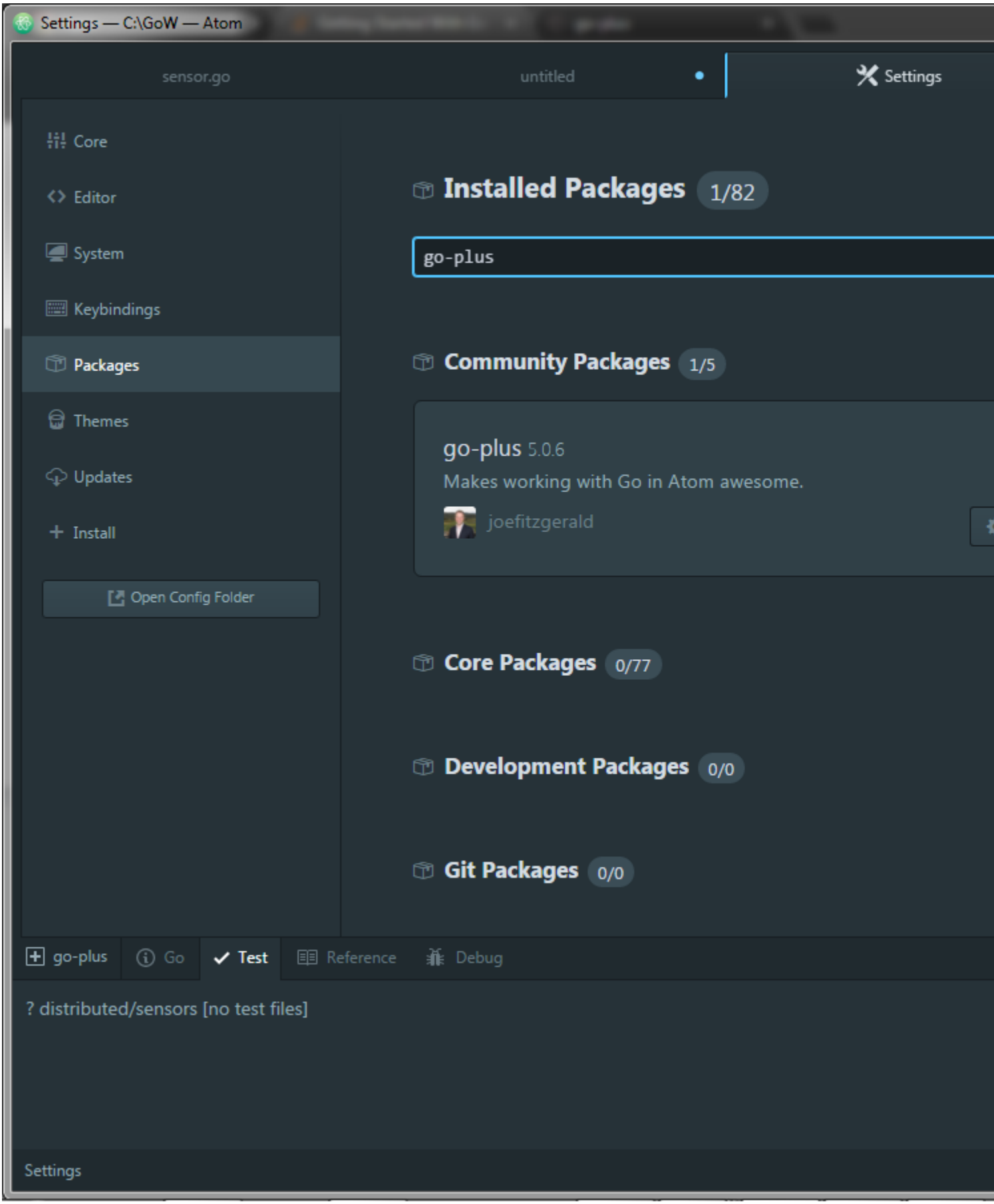
Après l'installation, allez (http://www.riptutorial.com/go/topic/198/getting-started-with-go) vous aurez besoin d'un environnement. Un moyen efficace et gratuit de commencer consiste à utiliser l'éditeur de texte Atom (https://atom.io) et gulp. Une question qui vous a peut-être traversé l'esprit est la suivante: *pourquoi utiliser gulp?* .Nous avons besoin de gulp pour l'auto-complétion. Commençons!

Examples

Obtenir, installer et installer Atom & Gulp

1. Installez Atom. Vous pouvez obtenir un atome d' ici
2. Accédez aux paramètres Atom (ctrl +,). Packages -> Installer le package go-plus (go-plus)

Après l'installation de go-plus dans Atom:



3. Obtenez ces dépendances en utilisant go get ou un autre gestionnaire de dépendances: (ouvrez une console et exécutez ces commandes)

   go get -u golang.org/x/tools/cmd/goimports

allez chercher -u golang.org/x/tools/cmd/gorename

aller chercher -u github.com/sqs/goreturns

aller chercher -u github.com/nsf/gocode

aller chercher -u github.com/alecthomas/gometalinter

go get -u github.com/zmb3/gogetdoc

aller chercher -u github.com/rogpeppe/godef

allez chercher -u golang.org/x/tools/cmd/guru

4. Installez Gulp (Gulpjs) en utilisant npm ou tout autre gestionnaire de paquets (gulp-getting-started-doc):

   $ npm install --global gulp

Créez $ GO_PATH / gulpfile.js

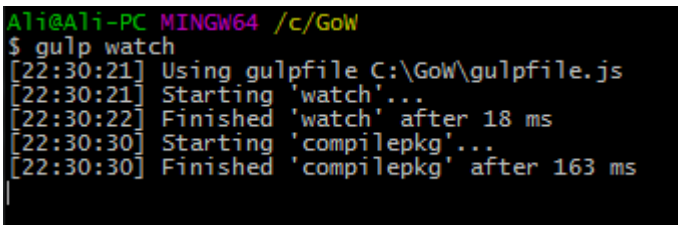
```
var gulp = require('gulp');
var path = require('path');
var shell = require('gulp-shell');

var goPath = 'src/mypackage/**/*.go';


gulp.task('compilepkg', function() {
  return gulp.src(goPath, {read: false})
    .pipe(shell(['go install <%= stripPath(file.path) %>'],
      {
          templateData: {
            stripPath: function(filePath) {
              var subPath = filePath.substring(process.cwd().length + 5);
              var pkg = subPath.substring(0, subPath.lastIndexOf(path.sep));
              return pkg;
            }
          }
      })
    );
});

gulp.task('watch', function() {
  gulp.watch(goPath, ['compilepkg']);
});
```

Dans le code ci-dessus, nous avons défini une tâche *compliepkg* qui sera déclenchée chaque fois qu'un fichier go dans goPath (src / mypackage /) ou des sous-répertoires change. la tâche lancera la commande shell go install changed_file.go

Après avoir créé le fichier gulp dans le chemin go et défini la tâche, ouvrez une ligne de commande et exécutez:

gulp montre

Vous allez voir quelque chose comme ça chaque fois que des modifications de fichiers:

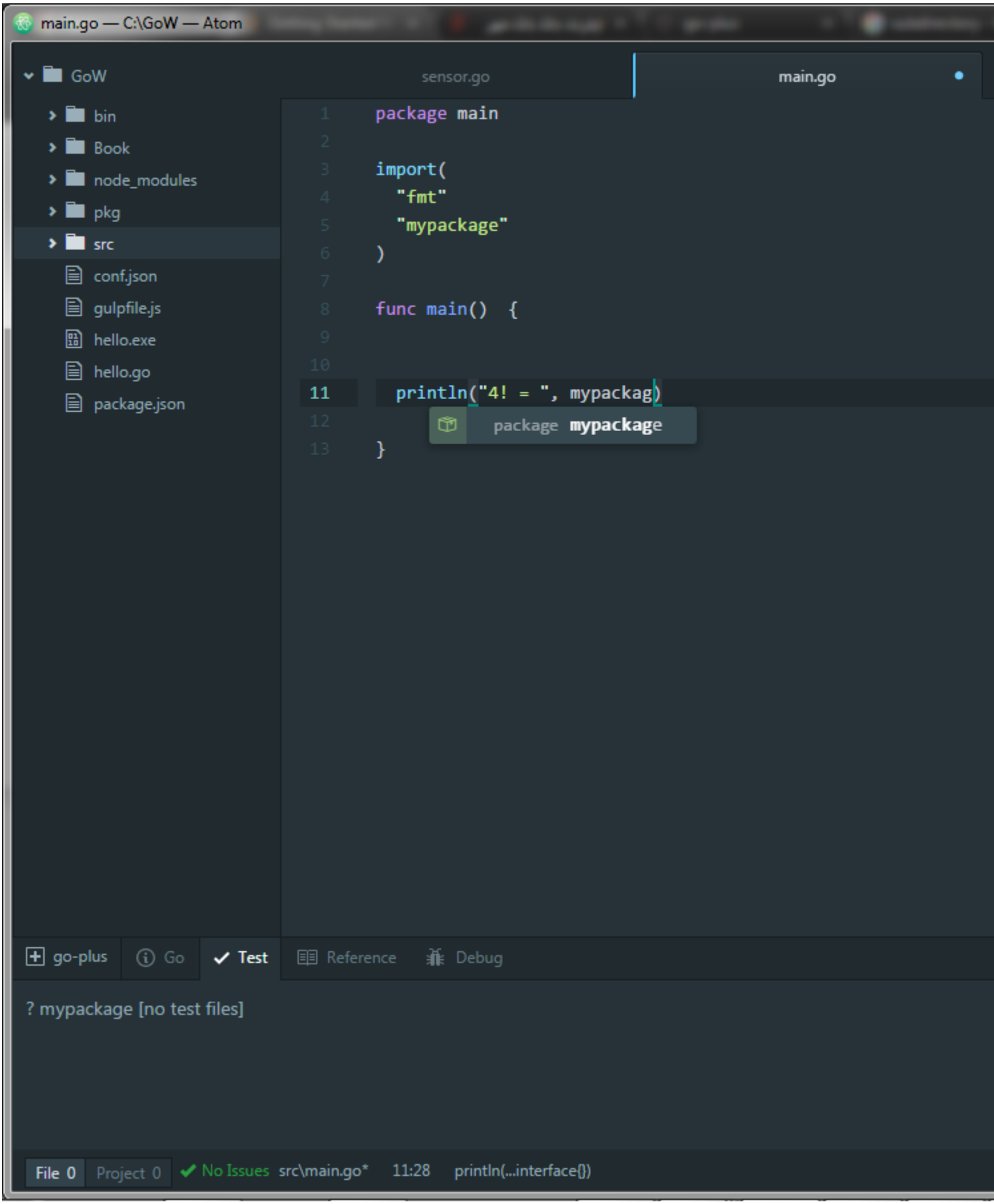


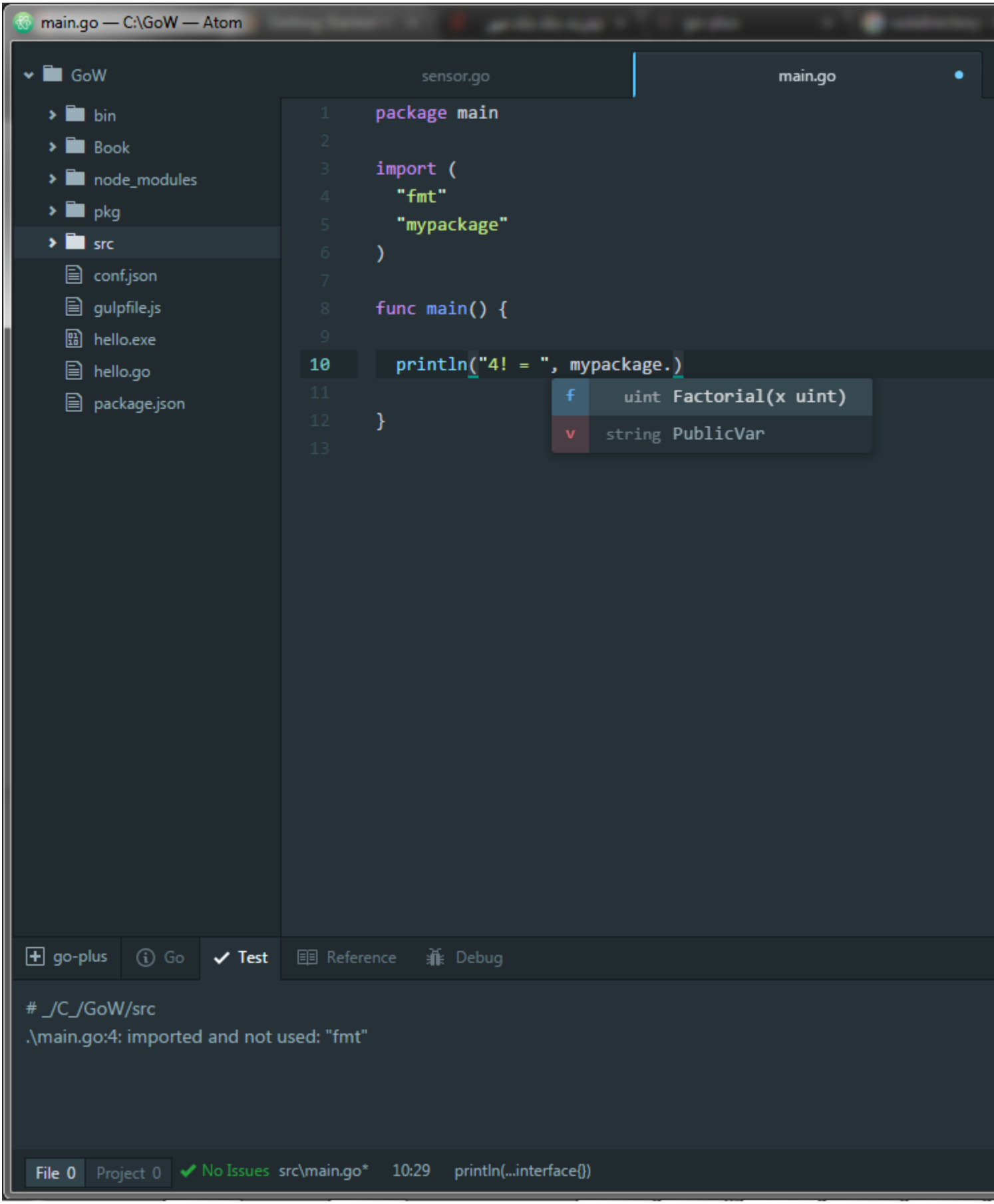
Créez $ GO_PATH / mypackage / source.go

```
package mypackage

var PublicVar string = "Hello, dear reader!"

//Calculates the factorial of given number recursively!
func Factorial(x uint) uint {
    if x == 0 {
        return 1
    }
    return x * Factorial(x-1)
}
```

Créer $ GO_PATH / main.go

Maintenant, vous pouvez commencer à écrire votre propre code go avec auto-complétion en utilisant Atom et Gulp:


main.go — C:\GoW — Atom
GoW
sensor.go
main.go
bin
Book
node_modules
pkg
src
conf.json
gulpfile.js
hello.exe
hello.go
package.json
1 package main
2
3 import(
4 "fmt"
5 "mypackage"
6)
7
8 func main() {
9
10
11 println("4! = ", mypackag)
12
13 }
package mypackage
go-plus
Go
Test
Reference
Debug
? mypackage [no test files]
File 0
Project 0
No Issues
src\main.go*
11:28
println(...interface{})




```
package main

import (
    "fmt"
```

```
    "mypackage"
)

func main() {

    println("4! = ", mypackage.Factorial(4))

}
```

```
Ali@Ali-PC MINGW64 /c/GoW
$ go run src/main.go
4! =  24
```

Lire Premiers pas avec Go en utilisant Atom en ligne:
https://riptutorial.com/fr/go/topic/8592/premiers-pas-avec-go-en-utilisant-atom

Chapitre 57: Profilage avec go tool pprof

Remarques

Pour en savoir plus sur les profils, rendez-vous sur le blog go .

Examples

Cpu de base et profilage de mémoire

Ajoutez le code suivant dans votre programme principal.

```
var cpuprofile = flag.String("cpuprofile", "", "write cpu profile `file`")
var memprofile = flag.String("memprofile", "", "write memory profile to `file`")

func main() {
    flag.Parse()
    if *cpuprofile != "" {
        f, err := os.Create(*cpuprofile)
        if err != nil {
            log.Fatal("could not create CPU profile: ", err)
        }
        if err := pprof.StartCPUProfile(f); err != nil {
            log.Fatal("could not start CPU profile: ", err)
        }
        defer pprof.StopCPUProfile()
    }
    ...
    if *memprofile != "" {
        f, err := os.Create(*memprofile)
        if err != nil {
            log.Fatal("could not create memory profile: ", err)
        }
        runtime.GC() // get up-to-date statistics
        if err := pprof.WriteHeapProfile(f); err != nil {
            log.Fatal("could not write memory profile: ", err)
        }
        f.Close()
    }
}
```

Après cela, **construisez** le programme go s'il est ajouté dans main `go build main.go` . Exécutez le programme principal avec les indicateurs définis dans le code `main.exe -cpuprofile cpu.prof -memprof mem.prof` . Si le profilage est effectué pour les cas de test, exécutez les tests avec les mêmes indicateurs. `go test -cpuprofile cpu.prof -memprofile mem.prof`

Mémoire de base

```
var memprofile = flag.String("memprofile", "", "write memory profile to `file`")

func main() {
```

```
    flag.Parse()
    if *memprofile != "" {
        f, err := os.Create(*memprofile)
        if err != nil {
            log.Fatal("could not create memory profile: ", err)
        }
        runtime.GC() // get up-to-date statistics
        if err := pprof.WriteHeapProfile(f); err != nil {
            log.Fatal("could not write memory profile: ", err)
        }
        f.Close()
    }
}
```

```
go build main.go
main.exe -memprofile mem.prof
go tool pprof main.exe mem.prof
```

Définir le taux de profil CPU / bloc

```
// Sets the CPU profiling rate to hz samples per second
// If hz <= 0, SetCPUProfileRate turns off profiling
runtime.SetCPUProfileRate(hz)

// Controls the fraction of goroutine blocking events that are reported in the blocking
profile
// Rate = 1 includes every blocking event in the profile
// Rate <= 0 turns off profiling
runtime.SetBlockProfileRate(rate)
```

Utilisation de repères pour créer un profil

Pour un colis non-principaux, ainsi que principale, au **lieu d'ajouter** des **drapeaux** à l' **intérieur du code,** écrire des **repères** dans le paquet de test, par exemple:

```
func BenchmarkHello(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fmt.Sprintf("hello")
    }
}
```

Ensuite, lancez le test avec l'indicateur de profil

    go test -cpuprofile cpu.prof -bench =.

Et les tests seront exécutés et créer un fichier prof avec le nom de fichier cpu.prof (dans l'exemple ci-dessus).

Accéder au fichier de profil

une fois qu'un fichier prof a été généré, on peut accéder au fichier prof à l'aide des **outils go** :

aller outil pprof cpu.prof

Cela entrera dans une interface de ligne de commande pour explorer le `profile`

Les commandes courantes incluent:

```
(pprof) top
```

répertorie les principaux processus en mémoire

```
(pprof) peek
```

Répertorie tous les processus, utilisez *regex* pour affiner la recherche.

```
(pprof) web
```

Ouvre un graphique (au format svg) du processus.

Un exemple de la commande `top` :

```
69.29s of 100.84s total (68.71%)
Dropped 176 nodes (cum <= 0.50s)
Showing top 10 nodes out of 73 (cum >= 12.03s)
      flat  flat%   sum%        cum   cum%
    12.44s 12.34% 12.34%     27.87s 27.64%  runtime.mapaccess1
    10.94s 10.85% 23.19%     10.94s 10.85%  runtime.duffcopy
     9.45s  9.37% 32.56%     54.61s 54.16%  github.com/tester/test.(*Circle).Draw
     8.88s  8.81% 41.36%      8.88s  8.81%  runtime.aeshashbody
     7.90s  7.83% 49.20%     11.04s 10.95%  runtime.mapaccess1_fast64
     5.86s  5.81% 55.01%      9.59s  9.51%  github.com/tester/test.(*Circle).isCircle
     5.03s  4.99% 60.00%      8.89s  8.82%  github.com/tester/test.(*Circle).openCircle
     3.14s  3.11% 63.11%      3.14s  3.11%  runtime.aeshash64
     3.08s  3.05% 66.16%      7.85s  7.78%  runtime.mallocgc
     2.57s  2.55% 68.71%     12.03s 11.93%  runtime.memhash
```

Lire Profilage avec go tool pprof en ligne: https://riptutorial.com/fr/go/topic/7748/profilage-avec-go-tool-pprof

Chapitre 58: Programmation orientée objet

Remarques

L'interface ne peut pas être implémentée avec les récepteurs de pointeurs car `*User` n'est pas un `User`

Examples

Structs

Go prend en charge les types définis par l'utilisateur sous la forme de structures et d'alias de type. Les structures sont des types composites, les composants de données constituant le type de structure sont appelés *champs* . un champ a un type et un nom qui doit être unqiue.

```
package main

type User struct {
    ID uint64
    FullName string
    Email    string
}

func main() {
    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
    }

    fmt.Println(user) // {1 Zelalem Mekonen zola.mk.27@gmail.com}
}
```

c'est aussi une syntaxe légale pour définir des structures

```
type User struct {
    ID uint64
    FullName, Email string
}

user := new(User)

user.ID = 1
user.FullName = "Zelalem Mekonen"
user.Email = "zola.mk.27@gmail.com"
```

Structures intégrées

Comme une structure est aussi un type de données, elle peut être utilisée comme un champ anonyme, la structure externe peut accéder directement aux champs de la structure intégrée

même si la structure provient d'un package différent. Ce comportement permet de dériver tout ou partie de votre implémentation à partir d'un autre type ou d'un ensemble de types.

```
package main

type Admin struct {
    Username, Password string
}

type User struct {
    ID uint64
    FullName, Email string
    Admin // embedded struct
}

func main() {
    admin := Admin{
        "zola",
        "supersecretpassword",
    }

    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
        admin,
    }

    fmt.Println(admin) // {zola supersecretpassword}

    fmt.Println(user) // {1 Zelalem Mekonen zola.mk.27@gmail.com {zola supersecretpassword}}

    fmt.Println(user.Username) // zola

    fmt.Println(user.Password) // supersecretpassword
}
```

Les méthodes

En Go une méthode est

> une fonction qui agit sur une variable d'un certain type, appelée le récepteur

le receveur peut être n'importe quoi, non seulement des `structs` mais même une `function` , les types d'alias pour les types intégrés tels que `int` , `string` , `bool` peuvent avoir une méthode, une exception à cette règle est que les `interfaces` interface est une définition abstraite et une méthode est une implémentation, essayant de générer une erreur de compilation.

En combinant des `structs` et des `methods` vous pouvez obtenir un équivalent proche d'une `class` en programmation orientée objet.

une méthode dans Go a la signature suivante

```
func (name receiverType) methodName(paramterList) (returnList) {}
```

```
package main
```

```
type Admin struct {
    Username, Password string
}

func (admin Admin) Delete() {
    fmt.Println("Admin Deleted")
}

type User struct {
    ID uint64
    FullName, Email string
    Admin
}

func (user User) SendEmail(email string) {
    fmt.Printf("Email sent to: %s\n", user.Email)
}

func main() {
    admin := Admin{
        "zola",
        "supersecretpassword",
    }

    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
        admin,
    }

    user.SendEmail("Hello") // Email sent to: zola.mk.27@gmail.com

    admin.Delete() // Admin Deleted
}
```

Pointeur Vs Value Receiver

le récepteur d'une méthode est généralement un pointeur pour des raisons de performances car nous ne ferions pas de copie de l'instance, comme ce serait le cas dans le récepteur de valeur, ceci est particulièrement vrai si le type de récepteur est une structure. une autre raison pour que le récepteur soit un pointeur serait de pouvoir modifier les données pointées par le récepteur.

un récepteur de valeur est utilisé pour éviter la modification des données contenues dans le récepteur, un récepteur de vaule peut provoquer une baisse de performance si le récepteur est une structure volumineuse.

```
package main

type User struct {
    ID uint64
    FullName, Email string
}

// We do no require any special syntax to access field because receiver is a pointer
func (user *User) SendEmail(email string) {
```

```
    fmt.Printf("Sent email to: %s\n", user.Email)
}

// ChangeMail will modify the users email because the receiver type is a ponter
func (user *User) ChangeEmail(email string) {
    user.Email = email;
}

func main() {
    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
    }

    user.SendEmail("Hello") // Sent email to: zola.mk.27@gmail.com

    user.ChangeEmail("zola@gmail.com")

    fmt.Println(user.Email) // zola@gmail.com
}
```

Interface et polymorphisme

Les interfaces permettent de spécifier le comportement d'un objet, si quelque chose peut le faire, il peut être utilisé ici. une interface définit un ensemble de méthodes, mais ces méthodes ne contiennent pas de code car elles sont abstraites ou l'implémentation est laissée à l'utilisateur de l'interface. Contrairement à la plupart des langages orientés objet, les interfaces peuvent contenir des variables dans Go.

Le polymorphisme est l'essence même de la programmation orientée objet: la capacité à traiter des objets de types différents de manière uniforme tant qu'ils adhèrent à la même interface. Les interfaces Go offrent cette fonctionnalité de manière très directe et intuitive

```
package main

type Runner interface {
    Run()
}

type Admin struct {
    Username, Password string
}

func (admin Admin) Run() {
    fmt.Println("Admin ==> Run()");
}

type User struct {
    ID uint64
    FullName, Email string
}

func (user User) Run() {
    fmt.Println("User ==> Run()")
}
```

```
// RunnerExample takes any type that fullfils the Runner interface
func RunnerExample(r Runner) {
    r.Run()
}

func main() {
    admin := Admin{
        "zola",
        "supersecretpassword",
    }

    user := User{
        1,
        "Zelalem Mekonen",
        "zola.mk.27@gmail.com",
    }

    RunnerExample(admin)

    RunnerExample(user)

}
```

Lire Programmation orientée objet en ligne: https://riptutorial.com/fr/go/topic/8801/programmation-orientee-objet

Chapitre 59: Protobuf in Go

Introduction

Protobuf ou Protocol Buffer code et décode les données afin que différentes applications ou modules écrits dans des langages différents puissent échanger le grand nombre de messages rapidement et de manière fiable sans surcharger le canal de communication. Avec protobuf, la performance est directement proportionnelle au nombre de messages que vous avez tendance à envoyer. Il compresse le message pour l'envoyer dans un format binaire sérialisé en fournissant vos outils pour encoder le message à la source et le décoder à la destination.

Remarques

Il y a deux étapes d'utilisation de **protobuf** .

1. Vous devez d'abord compiler les définitions de tampon de protocole
2. Importez les définitions ci-dessus, avec la bibliothèque de support dans votre programme.

Prise en charge de gRPC

Si un fichier proto spécifie les services RPC, protoc-gen-go peut être chargé de générer du code compatible avec gRPC (http://www.grpc.io/) . Pour ce faire, passez le paramètre `plugins` à protoc-gen-go; la manière habituelle est de l'insérer dans l'argument --go_out pour protoc:

```
protoc --go_out=plugins=grpc:. *.proto
```

Examples

Utiliser Protobuf avec Go

Le message que vous souhaitez sérialiser et envoyer que vous pouvez inclure dans un fichier **test.proto** , contenant

```
package example;

enum FOO { X = 17; };

message Test {
  required string label = 1;
  optional int32 type = 2 [default=77];
  repeated int64 reps = 3;
  optional group OptionalGroup = 4 {
    required string RequiredField = 5;
  }
}
```

Pour compiler la définition du tampon de protocole, exécutez le protocole avec le paramètre --

go_out défini dans le répertoire vers lequel vous voulez envoyer le code Go.

```
protoc --go_out=. *.proto
```

Pour créer et jouer avec un objet Test à partir du package exemple,

```
package main

import (
    "log"

    "github.com/golang/protobuf/proto"
    "path/to/example"
)

func main() {
    test := &example.Test {
        Label: proto.String("hello"),
        Type:  proto.Int32(17),
        Reps:  []int64{1, 2, 3},
        Optionalgroup: &example.Test_OptionalGroup {
            RequiredField: proto.String("good bye"),
        },
    }
    data, err := proto.Marshal(test)
    if err != nil {
        log.Fatal("marshaling error: ", err)
    }
    newTest := &example.Test{}
    err = proto.Unmarshal(data, newTest)
    if err != nil {
        log.Fatal("unmarshaling error: ", err)
    }
    // Now test and newTest contain the same data.
    if test.GetLabel() != newTest.GetLabel() {
        log.Fatalf("data mismatch %q != %q", test.GetLabel(), newTest.GetLabel())
    }
    // etc.
}
```

Pour transmettre des paramètres supplémentaires au plug-in, utilisez une liste de paramètres séparés par des virgules, séparée du répertoire de sortie par deux points:

```
protoc --go_out=plugins=grpc,import_path=mypackage:. *.proto
```

Lire Protobuf in Go en ligne: https://riptutorial.com/fr/go/topic/9729/protobuf-in-go

Chapitre 60: Ramification

Examples

Instructions de commutation

Une simple déclaration de `switch` :

```
switch a + b {
case c:
    // do something
case d:
    // do something else
default:
    // do something entirely different
}
```

L'exemple ci-dessus est équivalent à:

```
if a + b == c {
    // do something
} else if a + b == d {
    // do something else
} else {
    // do something entirely different
}
```

La clause `default` est facultative et sera exécutée si et seulement si aucun des cas ne se compare à true, même si cela ne semble pas le dernier, ce qui est acceptable. Ce qui suit est sémantiquement identique au premier exemple:

```
switch a + b {
default:
    // do something entirely different
case c:
    // do something
case d:
    // do something else
}
```

Cela peut être utile si vous avez l'intention d'utiliser l'instruction `fallthrough` dans la clause `default` , qui doit être la dernière instruction d'un cas et provoque l'exécution du programme au cas suivant:

```
switch a + b {
default:
    // do something entirely different, but then also do something
    fallthrough
case c:
    // do something
```

```
case d:
    // do something else
}
```

Une expression de commutateur vide est implicitement `true` :

```
switch {
case a + b == c:
    // do something
case a + b == d:
    // do something else
}
```

Les instructions switch prennent en charge une instruction simple similaire aux instructions `if` :

```
switch n := getNumber(); n {
case 1:
    // do something
case 2:
    // do something else
}
```

Les cas peuvent être combinés dans une liste séparée par des virgules s'ils partagent la même logique:

```
switch a + b {
case c, d:
    // do something
default:
    // do something entirely different
}
```

Si déclarations

Une simple déclaration `if` :

```
if a == b {
    // do something
}
```

Notez qu'il n'y a pas de parenthèses entourant la condition et que l'accolade ouvrante `{` doit être sur la même ligne. Les éléments suivants *ne* seront *pas* compilés:

```
if a == b
{
    // do something
}
```

Une déclaration `if` utilisant `else` :

```
if a == b {
    // do something
} else if a == c {
    // do something else
} else {
    // do something entirely different
}
```

Selon la documentation de golang.org , "l'expression peut être précédée d'une simple instruction, qui s'exécute avant que l'expression soit évaluée." Les variables déclarées dans cette instruction simple sont étendues à l'instruction `if` et ne sont pas accessibles en dehors de celle-ci:

```
if err := attemptSomething(); err != nil {
    // attemptSomething() was successful!
} else {
    // attemptSomething() returned an error; handle it
}
fmt.Println(err) // compiler error, 'undefined: err'
```

Relevés de type

Un simple commutateur de type:

```
// assuming x is an expression of type interface{}
switch t := x.(type) {
case nil:
    // x is nil
    // t will be type interface{}
case int:
    // underlying type of x is int
    // t will be int in this case as well
case string:
    // underlying type of x is string
    // t will be string in this case as well
case float, bool:
    // underlying type of x is either float or bool
    // since we don't know which, t is of type interface{} in this case
default:
    // underlying type of x was not any of the types tested for
    // t is interface{} in this type
}
```

Vous pouvez tester tout type, y compris l' `error` , les types définis par l'utilisateur, les types d'interface et les types de fonction:

```
switch t := x.(type) {
case error:
    log.Fatal(t)
case myType:
    fmt.Println(myType.message)
case myInterface:
    t.MyInterfaceMethod()
case func(string) bool:
    if t("Hello world?") {
```

```
        fmt.Println("Hello world!")
    }
}
```

Goto déclarations

Une `goto` transfère le contrôle à l'instruction avec l'étiquette correspondante dans la même fonction. L'exécution de l' `goto` ne doit pas entraîner la portée des variables qui n'étaient pas déjà présentes au point du `goto` .

Par exemple, consultez le code source de la bibliothèque standard: https://golang.org/src/math/gamma.go :

```
    for x < 0 {
        if x > -1e-09 {
            goto small
        }
        z = z / x
        x = x + 1
    }
    for x < 2 {
        if x < 1e-09 {
            goto small
        }
        z = z / x
        x = x + 1
    }

    if x == 2 {
        return z
    }

    x = x - 2
    p = (((((x*_gamP[0]+_gamP[1])*x+_gamP[2])*x+_gamP[3])*x+_gamP[4])*x+_gamP[5])*x + _gamP[6]
    q =
((((((x*_gamQ[0]+_gamQ[1])*x+_gamQ[2])*x+_gamQ[3])*x+_gamQ[4])*x+_gamQ[5])*x+_gamQ[6])*x +
_gamQ[7]
    return z * p / q

small:
    if x == 0 {
        return Inf(1)
    }
    return z / ((1 + Euler*x) * x)
```

Déclarations de rupture

L'instruction break, lors de l'exécution, oblige la boucle courante à quitter

paquet principal

```
import "fmt"

func main() {
    i:=0
```

```
    for true {
      if i>2 {
        break
        }
    fmt.Println("Iteration : ",i)
    i++
    }
}
```

L'instruction continue, à l'exécution, déplace le contrôle au début de la boucle

```
import "fmt"

func main() {
    j:=100
    for j<110 {
     j++
     if j%2==0 {
        continue
        }
    fmt.Println("Var : ",j)
    }
}
```

Casser / continuer la boucle à l'intérieur du commutateur

```
import "fmt"

func main() {
    j := 100

loop:
    for j < 110 {
        j++

        switch j % 3 {
        case 0:
            continue loop
        case 1:
            break loop
        }

        fmt.Println("Var : ", j)
    }
}
```

Lire Ramification en ligne: https://riptutorial.com/fr/go/topic/1342/ramification

Chapitre 61: Réflexion

Remarques

Les `reflect` sont une excellente référence. Dans la programmation informatique générale, la **réflexion** est la capacité d'un programme à **examiner** la structure et le comportement de **lui** - **même** à l' **`runtime`** .

Basé sur son système de `static type` strict, Go lang a des règles (lois de réflexion)

Examples

Utilisation de reflet de base.Valeur

```
import "reflect"

value := reflect.ValueOf(4)

// Interface returns an interface{}-typed value, which can be type-asserted
value.Interface().(int) // 4

// Type gets the reflect.Type, which contains runtime type information about
// this value
value.Type().Name() // int

value.SetInt(5) // panics -- non-pointer/slice/array types are not addressable

x := 4
reflect.ValueOf(&x).Elem().SetInt(5) // works
```

Structs

```
import "reflect"

type S struct {
    A int
    b string
}

func (s *S) String() { return s.b }

s := &S{
    A: 5,
    b: "example",
}

indirect := reflect.ValueOf(s) // effectively a pointer to an S
value := indirect.Elem()       // this is addressable, since we've derefed a pointer

value.FieldByName("A").Interface() // 5
```

```
value.Field(2).Interface()         // "example"

value.NumMethod()    // 0, since String takes a pointer receiver
indirect.NumMethod() // 1

indirect.Method(0).Call([]reflect.Value{})              // "example"
indirect.MethodByName("String").Call([]reflect.Value{}) // "example"
```

Tranches

```
import "reflect"

s := []int{1, 2, 3}

value := reflect.ValueOf(s)

value.Len()                // 3
value.Index(0).Interface() // 1
value.Type().Kind()        // reflect.Slice
value.Type().Elem().Name() // int

value.Index(1).CanAddr()   // true -- slice elements are addressable
value.Index(1).CanSet()    // true -- and settable
value.Index(1).Set(5)

typ := reflect.SliceOf(reflect.TypeOf("example"))
newS := reflect.MakeSlice(typ, 0, 10) // an empty []string{} with capacity 10
```

reflète.Value.Elem ()

```
import "reflect"

// this is effectively a pointer dereference

x := 5
ptr := reflect.ValueOf(&x)
ptr.Type().Name() // *int
ptr.Type().Kind() // reflect.Ptr
ptr.Interface()   // [pointer to x]
ptr.Set(4)        // panic

value := ptr.Elem() // this is a deref
value.Type().Name() // int
value.Type().Kind() // reflect.Int
value.Set(4)        // this works
value.Interface()   // 4
```

Type de valeur - le package "reflète"

reflect.TypeOf peut être utilisé pour vérifier le type de variables lors de la comparaison

```
package main
```

```
import (
    "fmt"
    "reflect"
)
type Data struct {
 a int
}
func main() {
    s:="hey dude"
    fmt.Println(reflect.TypeOf(s))

    D := Data{a:5}
    fmt.Println(reflect.TypeOf(D))

}
```

Sortie:
chaîne
donnée principale

Lire Réflexion en ligne: https://riptutorial.com/fr/go/topic/1854/reflexion

Chapitre 62: Reporter

Introduction

Une instruction de `defer` pousse un appel de fonction sur une liste. La liste des appels enregistrés est exécutée après le retour de la fonction environnante. Defer est couramment utilisé pour simplifier les fonctions qui effectuent diverses actions de nettoyage.

Syntaxe

- différer someFunc (args)
- reporter func () {// le code va ici} ()

Remarques

Defer fonctionne en injectant un nouveau frame de pile (la fonction appelée après le mot-clé `defer`) dans la pile d'appels située sous la fonction en cours d'exécution. Cela signifie que le renvoi est garanti tant que la pile sera déroulée (si votre programme plante ou obtient un `SIGKILL` , le différé ne sera pas exécuté).

Examples

Différer les bases

Une *déclaration différée* dans Go est simplement un appel de fonction marqué pour être exécuté ultérieurement. L'instruction de report est un appel de fonction ordinaire préfixé par le mot-clé `defer` .

```
defer someFunction()
```

Une fonction différée est exécutée une fois que la fonction qui contient la déclaration de `defer` est retournée. L'appel réel à la fonction différée se produit lorsque la fonction englobante:

- exécute une déclaration de retour
- tombe la fin
- panique

Exemple:

```
func main() {
    fmt.Println("First main statement")
    defer logExit("main") // position of defer statement here does not matter
    fmt.Println("Last main statement")
}
```

```
func logExit(name string) {
    fmt.Printf("Function %s returned\n", name)
}
```

Sortie:

```
First main statement
Last main statement
Function main returned
```

Si une fonction comporte plusieurs instructions différées, elles forment une pile. Le dernier `defer` est le premier à s'exécuter après le retour de la fonction englobante, suivi des appels suivants au `defer` précédent dans l'ordre (l'exemple ci-dessous renvoie en provoquant une panique):

```
func main() {
    defer logNum(1)
    fmt.Println("First main statement")
    defer logNum(2)
    defer logNum(3)
    panic("panic occurred")
    fmt.Println("Last main statement") // not printed
    defer logNum(3) // not deferred since execution flow never reaches this line
}

func logNum(i int) {
    fmt.Printf("Num %d\n", i)
}
```

Sortie:

```
First main statement
Num 3
Num 2
Num 1
panic: panic occurred

goroutine 1 [running]:
....
```

Notez que les fonctions différées ont leurs arguments évalués au moment `defer` exécute:

```
func main() {
    i := 1
    defer logNum(i) // deferred function call: logNum(1)
    fmt.Println("First main statement")
    i++
    defer logNum(i) // deferred function call: logNum(2)
    defer logNum(i*i) // deferred function call: logNum(4)
    return // explicit return
}

func logNum(i int) {
    fmt.Printf("Num %d\n", i)
}
```

Sortie:

```
First main statement
Num 4
Num 2
Num 1
```

Si une fonction a des valeurs de retour nommées, une fonction anonyme différée au sein de cette fonction peut accéder à la valeur renvoyée et la mettre à jour même après le retour de la fonction:

```
func main() {
    fmt.Println(plusOne(1)) // 2
    return
}

func plusOne(i int) (result int) { // overkill! only for demonstration
    defer func() {result += 1}() // anonymous function must be called by adding ()

    // i is returned as result, which is updated by deferred function above
    // after execution of below return
    return i
}
```

Enfin, une déclaration `defer` est généralement utilisée pour des opérations qui se produisent souvent ensemble. Par exemple:

- ouvrir et fermer un fichier
- connecter et déconnecter
- verrouiller et déverrouiller un mutex
- marquer un groupe d'attente comme fait (`defer wg.Done()`)

Cette utilisation garantit une libération correcte des ressources du système, quel que soit le flux d'exécution.

```
resp, err := http.Get(url)
if err != nil {
return err
}
defer resp.Body.Close() // Body will always get closed
```

Appels de fonction différés

Appels de fonction différée jouent un rôle semblable à des choses comme `finally` des blocs dans des langages comme Java: ils assurent que certaines fonction sera exécutée lorsque la fonction retourne externe, quel que soit le cas d' une erreur ou qui renvoient déclaration a été frappé dans les cas avec plusieurs retours. Ceci est utile pour nettoyer les ressources qui doivent être fermées comme les connexions réseau ou les pointeurs de fichiers. Le mot-clé `defer` indique un appel de fonction différé, de la même manière que le mot `go` clé `go` initiant une nouvelle goroutine. Comme un `go` appel, des arguments sont évalués immédiatement, mais contrairement à un `go` appel, les fonctions différées ne sont pas exécutées simultanément.

```
func MyFunc() {
    conn := GetConnection()    // Some kind of connection that must be closed.
    defer conn.Close()        // Will be executed when MyFunc returns, regardless of how.
    // Do some things...
    if someCondition {
        return                // conn.Close() will be called
    }
    // Do more things
}// Implicit return - conn.Close() will still be called
```

Notez l'utilisation de `conn.Close()` au lieu de `conn.Close` - vous ne faites pas que passer une fonction, vous différerez un *appel de* fonction complet, y compris ses arguments. Plusieurs appels de fonction peuvent être différés dans la même fonction externe et chacun sera exécuté une fois dans l'ordre inverse. Vous pouvez également reporter les fermetures - n'oubliez pas les parens!

```
defer func(){
    // Do some cleanup
}()
```

Lire Reporter en ligne: https://riptutorial.com/fr/go/topic/2795/reporter

Chapitre 63: Sélectionner et canaux

Introduction

Le mot-clé `select` fournit une méthode simple pour travailler avec des canaux et effectuer des tâches plus avancées. Il est fréquemment utilisé à plusieurs fins: - Gestion des délais d'attente. - Lorsqu'il y a plusieurs canaux à lire, le sélecteur lit au hasard un canal qui contient des données. - Fournir un moyen facile de définir ce qui se passe si aucune donnée n'est disponible sur un canal.

Syntaxe

- sélectionnez {}
- sélectionnez {case true:}
- select {case incomingData: = <-someChannel:}
- sélectionnez {default:}

Examples

Simple Select Travailler avec des canaux

Dans cet exemple, nous créons une goroutine (une fonction s'exécutant dans un thread séparé) qui accepte un paramètre `chan` , et qui boucle simplement, envoyant chaque fois des informations dans le canal.

Dans le `main` , nous avons une `for` boucle et une `select` . Le `select` bloquera le traitement jusqu'à ce que l'une des instructions de `case` devienne vraie. Ici nous avons déclaré deux cas; la première est lorsque l'information passe par le canal, et l'autre si aucun autre cas ne se produit, appelé par `default` .

```
// Use of the select statement with channels (no timeouts)
package main

import (
    "fmt"
    "time"
)

// Function that is "chatty"
// Takes a single parameter a channel to send messages down
func chatter(chatChannel chan<- string) {
    // Clean up our channel when we are done.
    // The channel writer should always be the one to close a channel.
    defer close(chatChannel)

    // loop five times and die
    for i := 1; i <= 5; i++ {
        time.Sleep(2 * time.Second) // sleep for 2 seconds
        chatChannel <- fmt.Sprintf("This is pass number %d of chatter", i)
    }
```

```
}

// Our main function
func main() {
    // Create the channel
    chatChannel := make(chan string, 1)

    // start a go routine with chatter (separate, non blocking)
    go chatter(chatChannel)

    // This for loop keeps things going while the chatter is sleeping
    for {
        // select statement will block this thread until one of the two conditions below is
met
        // because we have a default, we will hit default any time the chatter isn't chatting
        select {
        // anytime the chatter chats, we'll catch it and output it
        case spam, ok := <-chatChannel:
            // Print the string from the channel, unless the channel is closed
            // and we're out of data, in which case exit.
            if ok {
                fmt.Println(spam)
            } else {
                fmt.Println("Channel closed, exiting!")
                return
            }
        default:
            // print a line, then sleep for 1 second.
            fmt.Println("Nothing happened this second.")
            time.Sleep(1 * time.Second)
        }
    }
}
```

Essayez-le sur le terrain de jeu Go!

Utilisation de select avec timeouts

Donc, ici, j'ai supprimé les boucles `for` et mis un **délai** d' **attente** en ajoutant un deuxième `case` à la `select` qui retourne après 3 secondes. Parce que la `select` attend que ANY cas soit vrai, le deuxième `case` déclenche, puis notre script se termine, et `chatter()` n'a même jamais une chance de se terminer.

```
// Use of the select statement with channels, for timeouts, etc.
package main

import (
    "fmt"
    "time"
)

// Function that is "chatty"
//Takes a single parameter a channel to send messages down
func chatter(chatChannel chan<- string) {
    // loop ten times and die
    time.Sleep(5 * time.Second) // sleep for 5 seconds
    chatChannel<- fmt.Sprintf("This is pass number %d of chatter", 1)
}
```

```
// out main function
func main() {
    // Create the channel, it will be taking only strings, no need for a buffer on this
project
    chatChannel := make(chan string)
    // Clean up our channel when we are done
    defer close(chatChannel)

    // start a go routine with chatter (separate, no blocking)
    go chatter(chatChannel)

    // select statement will block this thread until one of the two conditions below is met
    // because we have a default, we will hit default any time the chatter isn't chatting
    select {
    // anytime the chatter chats, we'll catch it and output it
    case spam := <-chatChannel:
        fmt.Println(spam)
    // if the chatter takes more than 3 seconds to chat, stop waiting
    case <-time.After(3 * time.Second):
        fmt.Println("Ain't no time for that!")
    }
}
```

Lire Sélectionner et canaux en ligne: https://riptutorial.com/fr/go/topic/3539/selectionner-et-canaux

Chapitre 64: Serveur HTTP

Remarques

`http.ServeMux` fournit un multiplexeur qui appelle les gestionnaires pour les requêtes HTTP.

Les alternatives au multiplexeur de bibliothèque standard incluent:

- Gorilla Mux

Examples

HTTP Hello World avec serveur personnalisé et mux

```
package main

import (
    "log"
    "net/http"
)

func main() {

    // Create a mux for routing incoming requests
    m := http.NewServeMux()

    // All URLs will be handled by this function
    m.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, world!"))
    })

    // Create a server listening on port 8000
    s := &http.Server{
        Addr:    ":8000",
        Handler: m,
    }

    // Continue to process new requests until an error occurs
    log.Fatal(s.ListenAndServe())
}
```

Appuyez sur `Ctrl` + `C` pour arrêter le processus.

Bonjour le monde

La façon habituelle de commencer à écrire des serveurs Web dans golang consiste à utiliser le module `net/http` bibliothèque standard.

Il y a aussi un tutoriel pour cela ici .

Le code suivant l'utilise également. Voici l'implémentation de serveur HTTP la plus simple

possible. Il répond `"Hello World"` à toute requête HTTP.

Enregistrez le code suivant dans un fichier `server.go` dans vos espaces de travail.

```
package main

import (
    "log"
    "net/http"
)

func main() {
    // All URLs will be handled by this function
    // http.HandleFunc uses the DefaultServeMux
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, world!"))
    })

    // Continue to process new requests until an error occurs
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Vous pouvez exécuter le serveur en utilisant:

```
$ go run server.go
```

Ou vous pouvez compiler et exécuter.

```
$ go build server.go
$ ./server
```

Le serveur écoutera le port spécifié (`:8080`). Vous pouvez le tester avec n'importe quel client HTTP. Voici un exemple avec `cURL` :

```
curl -i http://localhost:8080/
HTTP/1.1 200 OK
Date: Wed, 20 Jul 2016 18:04:46 GMT
Content-Length: 13
Content-Type: text/plain; charset=utf-8

Hello, world!
```

Appuyez sur `Ctrl` + `C` pour arrêter le processus.

Utiliser une fonction de gestionnaire

`HandleFunc` enregistre la fonction de gestionnaire pour le modèle donné dans le serveur mux (routeur).

Vous pouvez passer à définir une fonction anonyme, comme nous l'avons vu dans l'exemple de base de *Hello World* :

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, world!")
}
```

Mais on peut aussi passer un type `HandlerFunc` . En d'autres termes, nous pouvons passer toute fonction qui respecte la signature suivante:

```
func FunctionName(w http.ResponseWriter, req *http.Request)
```

Nous pouvons réécrire l'exemple précédent en passant la référence à un `HandlerFunc` précédemment défini. Voici l'exemple complet:

```
package main

import (
    "fmt"
    "net/http"
)

// A HandlerFunc function
// Notice the signature of the function
func RootHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, "Hello, world!")
}

func main() {
    // Here we pass the reference to the `RootHandler` handler function
    http.HandleFunc("/", RootHandler)
    panic(http.ListenAndServe(":8080", nil))
}
```

Bien entendu, vous pouvez définir plusieurs gestionnaires de fonctions pour différents chemins.

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func FooHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, "Hello from foo!")
}

func BarHandler(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, "Hello from bar!")
}

func main() {
    http.HandleFunc("/foo", FooHandler)
    http.HandleFunc("/bar", BarHandler)

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Voici la sortie en utilisant `cURL` :

```
➜  ~ curl -i localhost:8080/foo
HTTP/1.1 200 OK
Date: Wed, 20 Jul 2016 18:23:08 GMT
Content-Length: 16
Content-Type: text/plain; charset=utf-8

Hello from foo!

➜  ~ curl -i localhost:8080/bar
HTTP/1.1 200 OK
Date: Wed, 20 Jul 2016 18:23:10 GMT
Content-Length: 16
Content-Type: text/plain; charset=utf-8

Hello from bar!

➜  ~ curl -i localhost:8080/
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Wed, 20 Jul 2016 18:23:13 GMT
Content-Length: 19

404 page not found
```

Créer un serveur HTTPS

Générer un certificat

Pour pouvoir exécuter un serveur HTTPS, un certificat est nécessaire. La génération d'un certificat auto-signé avec `openssl` se fait en exécutant cette commande:

```
openssl req -x509 -newkey rsa:4096 -sha256 -nodes -keyout key.pem -out cert.pem -subj
"/CN=example.com" -days 3650`
```

Les paramètres sont:

- `req` Utiliser l'outil de demande de certificat
- `x509` Crée un certificat auto-signé
- `newkey rsa:4096` Crée une nouvelle clé et un nouveau certificat en utilisant les algorithmes RSA avec une longueur de clé de `4096` bits.
- `sha256` Force les algorithmes de hachage SHA256 que les principaux navigateurs considèrent comme sécurisés (en 2017)
- `nodes` Désactive la protection par mot de passe pour la clé privée. Sans ce paramètre, votre serveur doit vous demander le mot de passe à chaque démarrage.
- `keyout` le fichier où écrire la clé
- `out` Nomme le fichier où écrire le certificat
- `subj` Définit le nom de domaine pour lequel ce certificat est valide
- `days` Combien de jours faut-il pour que ce certificat soit valide? `3650` sont environ. 10 années.

Remarque: Un certificat auto-signé peut être utilisé, par exemple, pour des projets internes, le débogage, les tests, etc. Tout navigateur sur ce site mentionnera que ce certificat n'est pas sûr. Pour éviter cela, le certificat doit être signé par une autorité de certification. Surtout, ce n'est pas disponible gratuitement. Une exception est le mouvement "Let's Encrypt": https://letsencrypt.org

Le code nécessaire

Vous pouvez gérer la configuration TLS pour le serveur avec le code suivant. `cert.pem` et `key.pem` sont votre certificat et votre clé SSL, générés avec la commande ci-dessus.

```
package main

import (
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, world!"))
    })

    log.Fatal(http.ListenAndServeTLS(":443","cert.pem","key.pem", nil))
}
```

Répondre à une demande HTTP à l'aide de modèles

Les réponses peuvent être écrites sur un `http.ResponseWriter` utilisant des modèles dans Go. Cela s'avère être un outil pratique si vous souhaitez créer des pages dynamiques.

(Pour savoir comment fonctionnent les modèles dans Go, visitez la page Documentation des modèles de version .)

Continuez avec un exemple simple pour utiliser le `html/template` pour répondre à une requête HTTP:

```
package main

import(
    "html/template"
    "net/http"
    "log"
)

func main(){
    http.HandleFunc("/",WelcomeHandler)
    http.ListenAndServe(":8080",nil)
}

type User struct{
    Name string
    nationality string //unexported field.
}
```

```
func check(err error){
    if err != nil{
        log.Fatal(err)
    }
}

func WelcomeHandler(w http.ResponseWriter, r *http.Request){
    if r.Method == "GET"{
        t,err := template.ParseFiles("welcomeform.html")
        check(err)
        t.Execute(w,nil)
    }else{
        r.ParseForm()
        myUser := User{}
        myUser.Name = r.Form.Get("entered_name")
        myUser.nationality = r.Form.Get("entered_nationality")
        t, err := template.ParseFiles("welcomeresponse.html")
        check(err)
        t.Execute(w,myUser)
    }
}
```

Où, le contenu de

1. `welcomeform.html` sont:

```
<head>
    <title> Help us greet you </title>
</head>
<body>
    <form method="POST" action="/">
        Enter Name: <input type="text" name="entered_name">
        Enter Nationality: <input type="text" name="entered_nationality">
        <input type="submit" value="Greet me!">
    </form>
</body>
```

1. `welcomeresponse.html` sont:

```
<head>
    <title> Greetings, {{.Name}} </title>
</head>
<body>
    Greetings, {{.Name}}.<br>
    We know you are a {{.nationality}}!
</body>
```

Remarque:

1. Assurez-vous que les fichiers `.html` sont dans le bon répertoire.
2. Lorsque `http://localhost:8080/` peut être visité après le démarrage du serveur.
3. Comme on peut le voir après avoir soumis le formulaire, le *champ de* nationalité non *exporté* de la structure n'a pas pu être analysé par le package de modèle, comme prévu.

Servir du contenu avec ServeMux

Un simple serveur de fichiers statique ressemblerait à ceci:

```
package main

import (
    "net/http"
)

func main() {
    muxer := http.NewServeMux()
    fileServerCss := http.FileServer(http.Dir("src/css"))
    fileServerJs := http.FileServer(http.Dir("src/js"))
    fileServerHtml := http.FileServer(http.Dir("content"))
    muxer.Handle("/", fileServerHtml)
    muxer.Handle("/css", fileServerCss)
    muxer.Handle("/js", fileServerJs)
    http.ListenAndServe(":8080", muxer)
}
```

Gestion de la méthode http, accès aux chaînes de requête et au corps de la requête

Voici un exemple simple de certaines tâches courantes liées au développement d'une API, en différenciant la méthode HTTP de la requête, en accédant aux valeurs de chaîne de requête et en accédant au corps de la requête.

Ressources

- Interface http.Handler
- http.ResponseWriter
- http.Request
- Constantes de méthode et d'état disponibles

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
)

type customHandler struct{}

// ServeHTTP implements the http.Handler interface in the net/http package
func (h customHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    // ParseForm will parse query string values and make r.Form available
    r.ParseForm()

    // r.Form is map of query string parameters
    // its' type is url.Values, which in turn is a map[string][]string
    queryMap := r.Form
```

```
    switch r.Method {
    case http.MethodGet:
        // Handle GET requests
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(fmt.Sprintf("Query string values: %s", queryMap)))
        return
    case http.MethodPost:
        // Handle POST requests
        body, err := ioutil.ReadAll(r.Body)
        if err != nil {
            // Error occurred while parsing request body
            w.WriteHeader(http.StatusBadRequest)
            return
        }
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(fmt.Sprintf("Query string values: %s\nBody posted: %s", queryMap,
body)))
        return
    }

    // Other HTTP methods (eg PUT, PATCH, etc) are not handled by the above
    // so inform the client with appropriate status code
    w.WriteHeader(http.StatusMethodNotAllowed)
}

func main() {
    // All URLs will be handled by this function
    // http.Handle, similarly to http.HandleFunc
    // uses the DefaultServeMux
    http.Handle("/", customHandler{})

    // Continue to process new requests until an error occurs
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Échantillon curl sortie:

```
$ curl -i 'localhost:8080?city=Seattle&state=WA' -H 'Content-Type: text/plain' -X GET
HTTP/1.1 200 OK
Date: Fri, 02 Sep 2016 16:36:24 GMT
Content-Length: 51
Content-Type: text/plain; charset=utf-8

Query string values: map[city:[Seattle] state:[WA]]%

$ curl -i 'localhost:8080?city=Seattle&state=WA' -H 'Content-Type: text/plain' -X POST -d
"some post data"
HTTP/1.1 200 OK
Date: Fri, 02 Sep 2016 16:36:35 GMT
Content-Length: 79
Content-Type: text/plain; charset=utf-8

Query string values: map[city:[Seattle] state:[WA]]
Body posted: some post data%

$ curl -i 'localhost:8080?city=Seattle&state=WA' -H 'Content-Type: text/plain' -X PUT
HTTP/1.1 405 Method Not Allowed
Date: Fri, 02 Sep 2016 16:36:41 GMT
Content-Length: 0
```

```
Content-Type: text/plain; charset=utf-8
```

Lire Serveur HTTP en ligne: https://riptutorial.com/fr/go/topic/756/serveur-http

Chapitre 65: Signaux OS

Syntaxe

- func Notify (c chan <- os.Signal, sig ... os.Signal)

Paramètres

Paramètre	Détails
c chan <- os.Signal	`channel` réception spécifiquement du type `os.Signal` ; facilement créé avec `sigChan := make(chan os.Signal)`
sig ... os.Signal	Liste des types `os.Signal` à capturer et envoyer sur ce `channel` . Voir https://golang.org/pkg/syscall/#pkg-constants pour plus d'options.

Examples

Assigner des signaux à un canal

Souvent, vous aurez des raisons de vous arrêter lorsque votre programme se fera arrêter par le système d'exploitation et prendra des mesures pour préserver l'état ou nettoyer votre application. Pour ce faire, vous pouvez utiliser le package `os/signal` de la bibliothèque standard. Vous trouverez ci-dessous un exemple simple d'affectation de tous les signaux du système à un canal, puis de la réaction à ces signaux.

```
package main

import (
    "fmt"
    "os"
    "os/signal"
)

func main() {
    // create a channel for os.Signal
    sigChan := make(chan os.Signal)

    // assign all signal notifications to the channel
    signal.Notify(sigChan)

    // blocks until you get a signal from the OS
    select {
    // when a signal is received
    case sig := <-sigChan:
        // print this line telling us which signal was seen
        fmt.Println("Received signal from OS:", sig)
    }
}
```

Lorsque vous exécutez le script ci-dessus, il créera un canal, puis bloquera jusqu'à ce que ce canal reçoive un signal.

```
$ go run signals.go
^CReceived signal from OS: interrupt
```

Le `^C` ci-dessus est la commande clavier `CTRL+C` qui envoie le signal `SIGINT` .

Lire Signaux OS en ligne: https://riptutorial.com/fr/go/topic/4497/signaux-os

Chapitre 66: SQL

Remarques

Pour obtenir une liste des pilotes de base de données SQL, consultez l'article officiel du wiki de Go *SQLDrivers* .

Les pilotes SQL sont importés et préfixés par _ , de sorte qu'ils *ne* sont disponibles que pour le pilote.

Examples

Interroger

Cet exemple montre comment interroger une base de données avec `database/sql` , en prenant comme exemple une base de données MySql.

```
package main

import (
    "log"
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    dsn := "mysql_username:CHANGEME@tcp(localhost:3306)/dbname"

    db, err := sql.Open("mysql", dsn)
    if err != nil {
        log.Fatal(err)
    }
    defer sql.Close()

    rows, err := db.Query("select id, first_name from user limit 10")
    if err != nil {
        log.Fatal(err)
    }
    defer rows.Close()

    for rows.Next() {
        var id int
        var username string
        if err := rows.Scan(&id, &username); err != nil {
            log.Fatal(err)
        }
        fmt.Printf("%d-%s\n", id, username)
    }
}
```

MySQL

Pour activer MySQL, un pilote de base de données est nécessaire. Par exemple, github.com/go-sql-driver/mysql .

```
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

Ouvrir une base de données

L'ouverture d'une base de données est spécifique à la base de données, il existe des exemples pour certaines bases de données.

Sqlite 3

```
file := "path/to/file"
db_, err := sql.Open("sqlite3", file)
if err != nil {
    panic(err)
}
```

MySql

```
dsn := "mysql_username:CHANGEME@tcp(localhost:3306)/dbname"
db, err := sql.Open("mysql", dsn)
if err != nil {
    panic(err)
}
```

MongoDB: connecter et insérer et supprimer et mettre à jour et interroger

```
package main

import (
    "fmt"
    "time"

    log "github.com/Sirupsen/logrus"
    mgo "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)

var mongoConn *mgo.Session

type MongoDB_Conn struct {
    Host string `json:"Host"`
    Port string `json:"Port"`
    User string `json:"User"`
    Pass string `json:"Pass"`
    DB   string `json:"DB"`
}

func MongoConn(mdb MongoDB_Conn) (*mgo.Session, string, error) {
    if mongoConn != nil {
```

```
        if mongoConn.Ping() == nil {
            return mongoConn, nil
        }
    }
    user := mdb.User
    pass := mdb.Pass
    host := mdb.Host
    port := mdb.Port
    db := mdb.DB
    if host == "" || port == "" || db == "" {
        log.Fatal("Host or port or db is nil")
    }
    url := fmt.Sprintf("mongodb://%s:%s@%s:%s/%s", user, pass, host, port, db)
    if user == "" {
        url = fmt.Sprintf("mongodb://%s:%s/%s", host, port, db)
    }
    mongo, err := mgo.DialWithTimeout(url, 3*time.Second)
    if err != nil {
        log.Errorf("Mongo Conn Error: [%v], Mongo ConnUrl: [%v]",
            err, url)
        errTextReturn := fmt.Sprintf("Mongo Conn Error: [%v]", err)
        return &mgo.Session{}, errors.New(errTextReturn)
    }
    mongoConn = mongo
    return mongoConn, nil
}

func MongoInsert(dbName, C string, data interface{}) error {
    mongo, err := MongoConn()
    if err != nil {
        log.Error(err)
        return err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    err = collection.Insert(data)
    if err != nil {
        return err
    }
    return nil
}

func MongoRemove(dbName, C string, selector bson.M) error {
    mongo, err := MongoConn()
    if err != nil {
        log.Error(err)
        return err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    err = collection.Remove(selector)
    if err != nil {
        return err
    }
    return nil
}

func MongoFind(dbName, C string, query, selector bson.M) ([]interface{}, error) {
    mongo, err := MongoConn()
    if err != nil {
        return nil, err
```

```
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    result := make([]interface{}, 0)
    err = collection.Find(query).Select(selector).All(&result)
    return result, err
}

func MongoUpdate(dbName, C string, selector bson.M, update interface{}) error {
    mongo, err := MongoConn()
    if err != nil {
        log.Error(err)
        return err
    }
    db := mongo.DB(dbName)
    collection := db.C(C)
    err = collection.Update(selector, update)
    if err != nil {
        return err
    }
    return nil
}
```

Lire SQL en ligne: https://riptutorial.com/fr/go/topic/1273/sql

Chapitre 67: Structs

Introduction

Les structures sont des ensembles de différentes variables regroupés. La structure elle-même n'est qu'un *package* contenant des variables et les rendant facilement accessibles.

Contrairement à C, les structures de Go peuvent être associées à des méthodes. Cela leur permet également de mettre en œuvre des interfaces. Cela rend les structures de Go similaires aux objets, mais il manque (probablement intentionnellement) des fonctionnalités majeures connues dans les langages orientés objet tels que l'héritage.

Examples

Déclaration de base

Une structure de base est déclarée comme suit:

```
type User struct {
    FirstName, LastName string
    Email               string
    Age                 int
}
```

Chaque valeur est appelée un champ. Les champs sont généralement écrits un par ligne, le nom du champ précédant son type. Des champs consécutifs du même type peuvent être combinés, comme `FirstName` et `LastName` dans l'exemple ci-dessus.

Champs exportés et non exportés (privés et publics)

Les champs de structure dont les noms commencent par une lettre majuscule sont exportés. Tous les autres noms sont non exportés.

```
type Account struct {
    UserID      int    // exported
    accessToken string // unexported
}
```

Les champs non exportés ne sont accessibles que par code dans le même package. En tant que tel, si vous accédez à un champ depuis un *autre* package, son nom doit commencer par une lettre majuscule.

```
package main

import "bank"

func main() {
```

```
    var x = &bank.Account{
        UserID: 1,          // this works fine
        accessToken: "one", // this does not work, since accessToken is unexported
    }
}
```

Cependant, à partir du package de `bank` , vous pouvez accéder à UserId et à accessToken sans problème.

La `bank` paquets pourrait être implémentée comme ceci:

```
package bank

type Account struct {
    UserID int
    accessToken string
}

func ProcessUser(u *Account) {
    u.accessToken = doSomething(u) // ProcessUser() can access u.accessToken because
                                   // it's defined in the same package
}
```

Composition et enrobage

La composition offre une alternative à l'héritage. Une structure peut inclure un autre type par nom dans sa déclaration:

```
type Request struct {
    Resource string
}

type AuthenticatedRequest struct {
    Request
    Username, Password string
}
```

Dans l'exemple ci-dessus, `AuthenticatedRequest` contiendra quatre membres publics: `Resource` , `Request` , `Username` et `Password` .

Les structures composites peuvent être instanciées et utilisées de la même manière que les structures normales:

```
func main() {
    ar := new(AuthenticatedRequest)
    ar.Resource = "example.com/request"
    ar.Username = "bob"
    ar.Password = "P@ssw0rd"
    fmt.Printf("%#v", ar)
}
```

jouer sur le terrain de jeu

Enrobage

Dans l'exemple précédent, `Request` est un champ incorporé. La composition peut également être obtenue en intégrant un type différent. Ceci est utile, par exemple, pour décorer une structure avec plus de fonctionnalités. Par exemple, en continuant avec l'exemple Resource, nous voulons une fonction qui formate le contenu du champ Resource pour le préfixer avec `http://` ou `https://` . Nous avons deux options: créer les nouvelles méthodes sur AuthenticatedRequest ou l' **incorporer** à partir d'une structure différente:

```
type ResourceFormatter struct {}

func(r *ResourceFormatter) FormatHTTP(resource string) string {
    return fmt.Sprintf("http://%s", resource)
}
func(r *ResourceFormatter) FormatHTTPS(resource string) string {
    return fmt.Sprintf("https://%s", resource)
}


type AuthenticatedRequest struct {
    Request
    Username, Password string
    ResourceFormatter
}
```

Et maintenant, la fonction principale pourrait faire ce qui suit:

```
func main() {
    ar := new(AuthenticatedRequest)
    ar.Resource = "www.example.com/request"
    ar.Username = "bob"
    ar.Password = "P@ssw0rd"

    println(ar.FormatHTTP(ar.Resource))
    println(ar.FormatHTTPS(ar.Resource))

    fmt.Printf("%#v", ar)
}
```

Regardez ce que `AuthenticatedRequest` a une structure incorporée `ResourceFormatter` .

Mais l'inconvénient est que vous ne pouvez pas accéder à des objets en dehors de votre composition. Donc, `ResourceFormatter` ne peut pas accéder aux membres de `AuthenticatedRequest` .

jouer sur le terrain de jeu

Les méthodes

Les méthodes Struct sont très similaires aux fonctions:

```
type User struct {
    name string
```

```
}

func (u User) Name() string {
    return u.name
}

func (u *User) SetName(newName string) {
    u.name = newName
}
```

La seule différence est l'ajout du récepteur de méthode. Il peut être déclaré soit comme une instance du type, soit comme un pointeur sur une instance du type. Étant donné que `SetName()` modifie l'instance, le récepteur doit être un pointeur afin d'effectuer un changement permanent dans l'instance.

Par exemple:

```
package main

import "fmt"

type User struct {
    name string
}

func (u User) Name() string {
    return u.name
}

func (u *User) SetName(newName string) {
    u.name = newName
}

func main() {
    var me User

    me.SetName("Slim Shady")
    fmt.Println("My name is", me.Name())
}
```

Aller au terrain de jeu

Structure anonyme

Il est possible de créer une structure anonyme:

```
data := struct {
    Number int
    Text   string
} {
    42,
    "Hello world!",
}
```

Exemple complet:

```
package main

import (
    "fmt"
)

func main() {
    data := struct {Number int; Text string}{42, "Hello world!"} // anonymous struct
    fmt.Printf("%+v\n", data)
}
```

jouer sur le terrain de jeu

Mots clés

Les champs de structure peuvent être associés à des balises. Ces balises peuvent être lues par le package de `reflect` pour obtenir des informations personnalisées spécifiées sur un champ par le développeur.

```
struct Account {
    Username      string `json:"username"`
    DisplayName   string `json:"display_name"`
    FavoriteColor string `json:"favorite_color,omitempty"`
}
```

Dans l'exemple ci-dessus, les balises sont utilisées pour modifier les noms de clé utilisés par le package `encoding/json` lors du marshaling ou de la suppression de JSON.

Bien que la balise puisse être n'importe quelle valeur de chaîne, il est conseillé d'utiliser des `key:"value"` séparées par des espaces `key:"value"` paires `key:"value"` :

```
struct StructName {
    FieldName int `package1:"customdata,moredata" package2:"info"`
}
```

Les balises struct utilisées avec le package `encoding/xml` et `encoding/json` sont utilisées dans la bibliothèque standard.

Faire des copies de structure

Une structure peut simplement être copiée en utilisant l'affectation.

```
type T struct {
    I int
    S string
}

// initialize a struct
t := T{1, "one"}

// make struct copy
u := t // u has its field values equal to t
```

```
if u == t { // true
    fmt.Println("u and t are equal") // Prints: "u and t are equal"
}
```

Dans le cas ci-dessus, `'t'` et «u» sont maintenant des objets séparés (valeurs de structure).

Comme `T` ne contient aucun type de référence (tranches, carte, canaux), ses champs, `t` et `u` ci-dessus peuvent être modifiés sans se toucher.

```
fmt.Printf("t.I = %d, u.I = %d\n", t.I, u.I) // t.I = 100, u.I = 1
```

Cependant, si `T` contient un type de référence, par exemple:

```
type T struct {
    I  int
    S  string
    xs []int // a slice is a reference type
}
```

Ensuite, une simple copie par affectation copiera également la valeur du champ de type de tranche dans le nouvel objet. Cela se traduirait par deux objets différents faisant référence au même objet de tranche.

```
// initialize a struct
t := T{I: 1, S: "one", xs: []int{1, 2, 3}}

// make struct copy
u := t // u has its field values equal to t
```

Étant donné que les deux u et t se réfèrent à la même tranche dans leur champ, xs mettant à jour une valeur dans la tranche d'un objet reflète le changement dans l'autre.

```
// update a slice field in u
u.xs[1] = 500

fmt.Printf("t.xs = %d, u.xs = %d\n", t.xs, u.xs)
// t.xs = [1 500 3], u.xs = [1 500 3]
```

Par conséquent, des précautions supplémentaires doivent être prises pour s'assurer que cette propriété de type référence ne produit pas de comportement involontaire.

Pour copier des objets ci-dessus par exemple, une copie explicite du champ de la tranche peut être effectuée:

```
// explicitly initialize u's slice field
u.xs = make([]int, len(t.xs))
// copy the slice values over from t
copy(u.xs, t.xs)

// updating slice value in u will not affect t
u.xs[1] = 500
```

```
fmt.Printf("t.xs = %d, u.xs = %d\n", t.xs, u.xs)
// t.xs = [1 2 3], u.xs = [1 500 3]
```

Struct Literals

Une valeur d'un type de structure peut être écrite en utilisant un *littéral struct* qui spécifie des valeurs pour ses champs.

```
type Point struct { X, Y int }
p := Point{1, 2}
```

L'exemple ci-dessus spécifie chaque champ dans le bon ordre. Ce qui n'est pas utile, car les programmeurs doivent se rappeler les champs exacts dans l'ordre. Plus souvent, une structure peut être initialisée en listant certains ou tous les noms de champs et leurs valeurs correspondantes.

```
anim := gif.GIF{LoopCount: nframes}
```

Les champs omis sont définis sur la valeur zéro pour son type.

Remarque: **Les deux formes ne peuvent pas être mélangées dans le même littéral.**

Structure vide

Une structure est une séquence d'éléments nommés, appelés champs, chacun ayant un nom et un type. La structure vide n'a pas de champs, comme cette structure vide anonyme:

```
var s struct{}
```

Ou comme ce type de structure vide nommée:

```
type T struct{}
```

La chose intéressante à propos de la structure vide est que sa taille est zéro (essayez The Go Playground):

```
fmt.Println(unsafe.Sizeof(s))
```

Ceci imprime `0` , donc la structure vide elle-même ne prend pas de mémoire. c'est donc une bonne option pour quitter canal, comme (essayez The Play Playground):

```
package main

import (
    "fmt"
    "time"
)

func main() {
```

```
    done := make(chan struct{})
    go func() {
        time.Sleep(1 * time.Second)
        close(done)
    }()

    fmt.Println("Wait...")
    <-done
    fmt.Println("done.")
}
```

Lire Structs en ligne: https://riptutorial.com/fr/go/topic/374/structs

Chapitre 68: Tableaux

Introduction

Les tableaux sont un type de données spécifique, représentant une collection ordonnée d'éléments d'un autre type.

Dans Go, les tableaux peuvent être simples (parfois appelés "listes") ou multidimensionnels (comme par exemple, un tableau à 2 dimensions représente une collection ordonnée de tableaux, qui contient des éléments).

Syntaxe

- var variableName [5] ArrayType // Déclarer un tableau de taille 5.
- var NomVariable [2] [3] ArrayType = {{Valeur1, Valeur2, Valeur3}, {Valeur4, Valeur5, Valeur6}} // Déclaration d'un tableau multidimensionnel
- variableName: = [...] ArrayType {Value1, Value2, Value3} // Déclarer un tableau de taille 3 (Le compilateur comptera les éléments du tableau pour définir la taille)
- arrayName [2] // Obtention de la valeur par index.
- arrayName [5] = 0 // Définition de la valeur à l'index.
- arrayName [0] // Première valeur du tableau
- arrayName [len (arrayName) -1] // Dernière valeur du tableau

Examples

Créer des tableaux

Un tableau en cours est une collection ordonnée d'éléments de même type.
La notation de base pour représenter les tableaux consiste à utiliser `[]` avec le nom de la variable.

Créer un nouveau tableau ressemble à `var array = [size]Type` , en remplaçant `size` par un nombre (par exemple `42` pour spécifier qu'il s'agira d'une liste de 42 éléments) et en remplaçant `Type` par le type des éléments que le tableau peut contenir (for exemple `int` ou `string`)

Juste en dessous se trouve un exemple de code montrant la manière différente de créer un tableau dans Go.

```
// Creating arrays of 6 elements of type int,
// and put elements 1, 2, 3, 4, 5 and 6 inside it, in this exact order:
var array1 [6]int = [6]int {1, 2, 3, 4, 5, 6} // classical way
var array2 = [6]int {1, 2, 3, 4, 5, 6} // a less verbose way
var array3 = [...]int {1, 2, 3, 4, 5, 6} // the compiler will count the array elements by
itself

fmt.Println("array1:", array1) // > [1 2 3 4 5 6]
fmt.Println("array2:", array2) // > [1 2 3 4 5 6]
fmt.Println("array3:", array3) // > [1 2 3 4 5 6]
```

```
// Creating arrays with default values inside:
zeros := [8]int{}               // Create a list of 8 int filled with 0
ptrs := [8]*int{}               // a list of int pointers, filled with 8 nil references (
<nil> )
emptystr := [8]string{}         // a list of string filled with 8 times ""

fmt.Println("zeroes:", zeros)      // > [0 0 0 0 0 0 0 0]
fmt.Println("ptrs:", ptrs)         // > [<nil> <nil> <nil> <nil> <nil> <nil> <nil> <nil>]
fmt.Println("emptystr:", emptystr) // > [       ]
// values are empty strings, separated by spaces,
// so we can just see separating spaces


// Arrays are also working with a personalized type
type Data struct {
    Number int
    Text   string
}

// Creating an array with 8 'Data' elements
// All the 8 elements will be like {0, ""} (Number = 0, Text = "")
structs := [8]Data{}

fmt.Println("structs:", structs) // > [{0 } {0 } {0 } {0 } {0 } {0 } {0 } {0 }]
// prints {0 } because Number are 0 and Text are empty; separated by a space
```

jouer sur le terrain de jeu

Tableau multidimensionnel

Les tableaux multidimensionnels sont essentiellement des tableaux contenant d'autres tableaux en tant qu'éléments.
Il est représenté comme type `[sizeDim1][sizeDim2]..[sizeLastDim]type` , en remplaçant `sizeDim` par des nombres correspondant à la longueur de la dimension, et `type` le type de données dans le tableau multidimensionnel.

Par exemple, `[2][3]int` représente un tableau composé de **2 sous-tableaux** de **3 éléments int** .
Il peut s'agir essentiellement de la représentation d'une matrice de **2 lignes** et de **3 colonnes** .

Donc, nous pouvons créer des tableaux de nombres de dimensions énormes comme les `var values := [2017][12][31][24][60]int` par exemple si vous devez enregistrer un nombre pour chaque minute depuis l'année 0.

Pour accéder à ce type de tableau, dans le dernier exemple, en recherchant la valeur de 2016-01-31 à 19:42, vous accéderez aux `values[2016][0][30][19][42]` (car les **index de tableau commence à 0** et pas à 1 comme jours et mois)

Quelques exemples suivants:

```
// Defining a 2d Array to represent a matrix like
// 1 2 3     So with 2 lines and 3 columns;
// 4 5 6
var multiDimArray := [2/*lines*/][3/*columns*/]int{ [3]int{1, 2, 3}, [3]int{4, 5, 6} }
```

```
// That can be simplified like this:
var simplified := [2][3]int{{1, 2, 3}, {4, 5, 6}}

// What does it looks like ?
fmt.Println(multiDimArray)
// > [[1 2 3] [4 5 6]]

fmt.Println(multiDimArray[0])
// > [1 2 3]    (first line of the array)

fmt.Println(multiDimArray[0][1])
// > 2          (cell of line 0 (the first one), column 1 (the 2nd one))
```

```
// We can also define array with as much dimensions as we need
// here, initialized with all zeros
var multiDimArray := [2][4][3][2]string{}

fmt.Println(multiDimArray);
// Yeah, many dimensions stores many data
// > [[[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]]
//   [[[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]]
```

```
// We can set some values in the array's cells
multiDimArray[0][0][0][0] := "All zero indexes"  // Setting the first value
multiDimArray[1][3][2][1] := "All indexes to max"  // Setting the value at extreme location

fmt.Println(multiDimArray);
// If we could see in 4 dimensions, maybe we could see the result as a simple format

// > [[["All zero indexes" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]]
//   [[[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" ""]]]
//    [[["" ""] ["" ""]] [["" ""] ["" ""]] [["" ""] ["" "All indexes to max"]]]]
```

Index de tableau

Les valeurs de tableaux doivent être accessibles à l'aide d'un numéro spécifiant l'emplacement de la valeur souhaitée dans le tableau. Ce numéro s'appelle Index.

Les index commencent à **0** et finissent à la **longueur du tableau -1** .

Pour accéder à une valeur, vous devez faire quelque chose comme ceci: `arrayName[index]` , en remplaçant "index" par le nombre correspondant au rang de la valeur dans votre tableau.

Par exemple:

```
var array = [6]int {1, 2, 3, 4, 5, 6}

fmt.Println(array[-42]) // invalid array index -1 (index must be non-negative)
fmt.Println(array[-1]) // invalid array index -1 (index must be non-negative)
fmt.Println(array[0]) // > 1
fmt.Println(array[1]) // > 2
fmt.Println(array[2]) // > 3
fmt.Println(array[3]) // > 4
fmt.Println(array[4]) // > 5
fmt.Println(array[5]) // > 6
fmt.Println(array[6]) // invalid array index 6 (out of bounds for 6-element array)
fmt.Println(array[42]) // invalid array index 42 (out of bounds for 6-element array)
```

Pour définir ou modifier une valeur dans le tableau, le chemin est le même.
Exemple:

```
var array = [6]int {1, 2, 3, 4, 5, 6}

fmt.Println(array) // > [1 2 3 4 5 6]

array[0] := 6
fmt.Println(array) // > [6 2 3 4 5 6]

array[1] := 5
fmt.Println(array) // > [6 5 3 4 5 6]

array[2] := 4
fmt.Println(array) // > [6 5 4 4 5 6]

array[3] := 3
fmt.Println(array) // > [6 5 4 3 5 6]

array[4] := 2
fmt.Println(array) // > [6 5 4 3 2 6]

array[5] := 1
fmt.Println(array) // > [6 5 4 3 2 1]
```

Lire Tableaux en ligne: https://riptutorial.com/fr/go/topic/390/tableaux

Chapitre 69: Temps

Introduction

Le package Go `time` fournit des fonctionnalités pour mesurer et afficher le temps.

Ce paquet fournit une structure `time.Time` , permettant de stocker et de faire des calculs aux dates et heures.

Syntaxe

- time.Date (2016, time.December, 31, 23, 59, 59, 999, time.UTC) // initialise
- date1 == date2 // retourne `true` quand les 2 sont le même moment
- date1! = date2 // renvoie `true` lorsque les 2 sont des moments différents
- date1.Avant (date2) // retourne `true` lorsque le premier est strictement avant le second
- date1.After (date2) // renvoie `true` lorsque le premier est strictement après le second

Examples

Heure de retour.Temps Zéro Valeur lorsque la fonction a une erreur

```
const timeFormat = "15 Monday January 2006"

func ParseDate(s string) (time.Time, error) {
    t, err := time.Parse(timeFormat, s)
    if err != nil {
        // time.Time{} returns January 1, year 1, 00:00:00.000000000 UTC
        // which according to the source code is the zero value for time.Time
        // https://golang.org/src/time/time.go#L23
        return time.Time{}, err
    }
    return t, nil
}
```

Analyse du temps

Si vous avez une date stockée sous forme de chaîne, vous devrez l'analyser. Utiliser le `time.Parse` .

```
//           time.Parse(   format    , date to parse)
date, err := time.Parse("01/02/2006",  "04/08/2017")
if err != nil {
    panic(err)
}

fmt.Println(date)
// Prints 2017-04-08 00:00:00 +0000 UTC
```

Le premier paramètre est la disposition dans laquelle la chaîne stocke la date et le deuxième paramètre est la chaîne contenant la date. `01/02/2006` est la même chose que de dire que le format est `MM/DD/YYYY` .

La mise en page définit le format en indiquant comment l'heure de référence, définie comme étant `Mon Jan 2 15:04:05 -0700 MST 2006` serait interprétée si c'était la valeur; il sert d'exemple du format d'entrée. La même interprétation sera alors apportée à la chaîne d'entrée.

Vous pouvez voir les constantes définies dans le package de temps pour savoir comment écrire la chaîne de présentation, mais notez que les constantes ne sont pas exportées et ne peuvent pas être utilisées en dehors du package de temps.

```
const (
    stdLongMonth             // "January"
    stdMonth                 // "Jan"
    stdNumMonth              // "1"
    stdZeroMonth             // "01"
    stdLongWeekDay           // "Monday"
    stdWeekDay               // "Mon"
    stdDay                   // "2"
    stdUnderDay              // "_2"
    stdZeroDay               // "02"
    stdHour                  // "15"
    stdHour12                // "3"
    stdZeroHour12            // "03"
    stdMinute                // "4"
    stdZeroMinute            // "04"
    stdSecond                // "5"
    stdZeroSecond            // "05"
    stdLongYear              // "2006"
    stdYear                  // "06"
    stdPM                    // "PM"
    stdpm                    // "pm"
    stdTZ                    // "MST"
    stdISO8601TZ             // "Z0700"  // prints Z for UTC
    stdISO8601SecondsTZ      // "Z070000"
    stdISO8601ShortTZ        // "Z07"
    stdISO8601ColonTZ        // "Z07:00" // prints Z for UTC
    stdISO8601ColonSecondsTZ // "Z07:00:00"
    stdNumTZ                 // "-0700"  // always numeric
    stdNumSecondsTz          // "-070000"
    stdNumShortTZ            // "-07"    // always numeric
    stdNumColonTZ            // "-07:00" // always numeric
    stdNumColonSecondsTZ     // "-07:00:00"
)
```

Temps de comparaison

Parfois, vous aurez besoin de connaître, avec des objets à 2 dates, s'il y a des correspondances avec la même date, ou de trouver quelle date est après l'autre.

Dans **Go** , il y a 4 façons de comparer les dates:

- `date1 == date2` , retourne `true` quand les 2 sont le même moment
- `date1 != date2` , renvoie `true` lorsque les 2 sont des moments différents

- `date1.Before(date2)` , renvoie `true` lorsque le premier est strictement avant le second
- `date1.After(date2)` , renvoie `true` lorsque le premier est strictement après le second

  ATTENTION: Lorsque les 2 temps à comparer sont les mêmes (ou correspondent à la même date), les fonctions `After` et `Before` renverront `false` , car une date n'est ni avant ni après elle-même

  - `date1 == date1` , retourne `true`
  - `date1 != date1` , retourne `false`
  - `date1.After(date1)` , retourne `false`
  - `date1.Before(date1)` , retourne `false`

  CONSEILS: Si vous avez besoin de savoir si une date est antérieure ou égale à une autre, il suffit de combiner les 4 opérateurs

  - `date1 == date2 && date1.After(date2)` , renvoie `true` lorsque la date1 est postérieure ou égale à date2
    ou en utilisant `! (date1.Before(date2))`
  - `date1 == date2 && date1.Before(date2)` , retourne `true` lorsque date1 est avant ou égale date2 ou utilise `!(date1.After(date2))`

Quelques exemples pour voir comment utiliser:

```
// Init 2 dates for example
var date1 = time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
var date2 = time.Date(2017, time.July, 25, 16, 22, 42, 123, time.UTC)
var date3 = time.Date(2017, time.July, 25, 16, 22, 42, 123, time.UTC)

bool1 := date1.Before(date2) // true, because date1 is before date2
bool2 := date1.After(date2) // false, because date1 is not after date2

bool3 := date2.Before(date1) // false, because date2 is not before date1
bool4 := date2.After(date1) // true, because date2 is after date1

bool5 := date1 == date2 // false, not the same moment
bool6 := date1 == date3 // true, different objects but representing the exact same time

bool7 := date1 != date2 // true, different moments
bool8 := date1 != date3 // false, not different moments

bool9 := date1.After(date3) // false, because date1 is not after date3 (that are the same)
bool10:= date1.Before(date3) // false, because date1 is not before date3 (that are the same)

bool11 := !(date1.Before(date3)) // true, because date1 is not before date3
bool12 := !(date1.After(date3)) // true, because date1 is not after date3
```

Lire Temps en ligne: https://riptutorial.com/fr/go/topic/8860/temps

Chapitre 70: Texte + HTML Templating

Examples

Modèle d'élément unique

Notez l'utilisation de `{{.}}` Pour générer l'élément dans le modèle.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

func main() {
    const (
        letter = `Dear {{.}}, How are you?`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    tmpl.Execute(os.Stdout, "Professor Jones")
}
```

Résulte en:

```
Dear Professor Jones, How are you?
```

Modèle de plusieurs articles

Notez l'utilisation de `{{range .}}` Et `{{end}}` pour parcourir la collection.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

func main() {
    const (
        letter = `Dear {{range .}}{{.}}, {{end}} How are you?`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
```

```
    }

    tmpl.Execute(os.Stdout, []string{"Harry", "Jane", "Lisa", "George"})
}
```

Résulte en:

```
Dear Harry, Jane, Lisa, George,  How are you?
```

Modèles avec logique personnalisée

Dans cet exemple, une carte de fonction nommée `funcMap` est fournie au modèle via la méthode `Funcs()` , puis invoquée dans le modèle. Ici, la fonction `increment()` est utilisée pour contourner le manque de fonction inférieure ou égale dans le langage de modélisation. Notez dans la sortie comment le dernier élément de la collection est géré.

A `-` au début `{{-` ou end `-}}` est utilisé pour couper les espaces et peut être utilisé pour rendre le modèle plus lisible.

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

var funcMap = template.FuncMap{
    "increment": increment,
}

func increment(x int) int {
    return x + 1
}

func main() {
    const (
        letter = `Dear {{with $names := .}}
        {{- range $i, $val := $names}}
            {{- if lt (increment $i) (len $names)}}
                {{- $val}}, {{else -}} and {{$val}}{{end}}
        {{- end}}{{end}}; How are you?`
    )

    tmpl, err := template.New("letter").Funcs(funcMap).Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    tmpl.Execute(os.Stdout, []string{"Harry", "Jane", "Lisa", "George"})
}
```

Résulte en:

```
Dear Harry, Jane, Lisa, and George; How are you?
```

Modèles avec des structures

Notez comment les valeurs de champs sont obtenues en utilisant `{{.FieldName}}` .

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

type Person struct {
    FirstName string
    LastName  string
    Street    string
    City      string
    State     string
    Zip       string
}

func main() {
    const (
        letter = `------------------------------
{{range .}}{{.FirstName}} {{.LastName}}
{{.Street}}
{{.City}}, {{.State}} {{.Zip}}

Dear {{.FirstName}},
    How are you?

------------------------------
{{end}}`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    harry := Person{
        FirstName: "Harry",
        LastName:  "Jones",
        Street:    "1234 Main St.",
        City:      "Springfield",
        State:     "IL",
        Zip:       "12345-6789",
    }

    jane := Person{
        FirstName: "Jane",
        LastName:  "Sherman",
        Street:    "8511 1st Ave.",
        City:      "Dayton",
        State:     "OH",
        Zip:       "18515-6261",
    }

    tmpl.Execute(os.Stdout, []Person{harry, jane})
}
```

Résulte en:

```
------------------------------
Harry Jones
1234 Main St.
Springfield, IL 12345-6789

Dear Harry,
    How are you?

------------------------------
Jane Sherman
8511 1st Ave.
Dayton, OH 18515-6261

Dear Jane,
    How are you?

------------------------------
```

Modèles HTML

Notez les différentes importations de packages.

```
package main

import (
    "fmt"
    "html/template"
    "os"
)

type Person struct {
    FirstName string
    LastName  string
    Street    string
    City      string
    State     string
    Zip       string
    AvatarUrl string
}

func main() {
    const (
        letter = `<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>
{{range .}}
<tr>
<td><img src="{{.AvatarUrl}}"></td>
<td>{{.FirstName}} {{.LastName}}</td>
<td>{{.Street}}, {{.City}}, {{.State}} {{.Zip}}</td>
</tr>
{{end}}
</table></body></html>`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
```

```
        fmt.Println(err.Error())
    }

    harry := Person{
        FirstName: "Harry",
        LastName:  "Jones",
        Street:    "1234 Main St.",
        City:      "Springfield",
        State:     "IL",
        Zip:       "12345-6789",
        AvatarUrl: "harry.png",
    }

    jane := Person{
        FirstName: "Jane",
        LastName:  "Sherman",
        Street:    "8511 1st Ave.",
        City:      "Dayton",
        State:     "OH",
        Zip:       "18515-6261",
        AvatarUrl: "jane.png",
    }

    tmpl.Execute(os.Stdout, []Person{harry, jane})
}
```

Résulte en:

```
<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td><img src="harry.png"></td>
<td>Harry Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td><img src="jane.png"></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>
```

Comment les modèles HTML empêchent l'injection de code malveillant

Tout d'abord, voici ce qui peut arriver lorsque le `text/template` est utilisé pour HTML. Notez la propriété `FirstName` de Harry).

```
package main

import (
    "fmt"
    "html/template"
    "os"
)
```

```
type Person struct {
    FirstName string
    LastName  string
    Street    string
    City      string
    State     string
    Zip       string
    AvatarUrl string
}

func main() {
    const (
        letter = `<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>
{{range .}}
<tr>
<td><img src="{{.AvatarUrl}}"></td>
<td>{{.FirstName}} {{.LastName}}</td>
<td>{{.Street}}, {{.City}}, {{.State}} {{.Zip}}</td>
</tr>
{{end}}
</table></body></html>`
    )

    tmpl, err := template.New("letter").Parse(letter)
    if err != nil {
        fmt.Println(err.Error())
    }

    harry := Person{
        FirstName: `Harry<script>alert("You've been hacked!")</script>`,
        LastName:  "Jones",
        Street:    "1234 Main St.",
        City:      "Springfield",
        State:     "IL",
        Zip:       "12345-6789",
        AvatarUrl: "harry.png",
    }

    jane := Person{
        FirstName: "Jane",
        LastName:  "Sherman",
        Street:    "8511 1st Ave.",
        City:      "Dayton",
        State:     "OH",
        Zip:       "18515-6261",
        AvatarUrl: "jane.png",
    }

    tmpl.Execute(os.Stdout, []Person{harry, jane})
}
```

Résulte en:

```
<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td><img src="harry.png"></td>
```

```
<td>Harry<script>alert("You've been hacked!")</script> Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td><img src="jane.png"></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>
```

L'exemple ci-dessus, s'il est accessible depuis un navigateur, entraînerait l'exécution du script et la génération d'une alerte. Si, au lieu de cela, le `html/template` était importé à la place de `text/template` , le script serait désinfecté en toute sécurité:

```
<html><body><table>
<tr><th></th><th>Name</th><th>Address</th></tr>

<tr>
<td><img src="harry.png"></td>
<td>Harry&lt;script&gt;alert(&#34;You&#39;ve been hacked!&#34;)&lt;/script&gt; Jones</td>
<td>1234 Main St., Springfield, IL 12345-6789</td>
</tr>

<tr>
<td><img src="jane.png"></td>
<td>Jane Sherman</td>
<td>8511 1st Ave., Dayton, OH 18515-6261</td>
</tr>

</table></body></html>
```

Le second résultat serait déformé lorsqu'il est chargé dans un navigateur, mais ne provoquerait pas l'exécution d'un script potentiellement malveillant.

Lire Texte + HTML Templating en ligne: https://riptutorial.com/fr/go/topic/3888/texte-plus-html-templating

Chapitre 71: Tranches

Introduction

Une tranche est une structure de données qui encapsule un tableau afin que le programmeur puisse ajouter autant d'éléments que nécessaire sans avoir à se soucier de la gestion de la mémoire. Les tranches peuvent être découpées en sous-tranches très efficacement, puisque les tranches résultantes pointent toutes vers le même tableau interne. Les programmeurs vont souvent en profiter pour éviter de copier des tableaux, ce qui se fait généralement dans de nombreux autres langages de programmation.

Syntaxe

- slice: = make ([] type, len, cap) // crée une nouvelle tranche
- slice = ajouter (tranche, élément) // ajouter un élément à une tranche
- slice = append (tranche, éléments ...) // ajoute une tranche d'éléments à une tranche
- len: = len (slice) // obtient la longueur d'une tranche
- cap: = cap (tranche) // obtient la capacité d'une tranche
- elNum: = copy (dst, slice) // copie le contenu d'une tranche sur une autre tranche

Examples

Aider à trancher

```
slice = append(slice, "hello", "world")
```

Ajouter deux tranches ensemble

```
slice1 := []string{"!"}
slice2 := []string{"Hello", "world"}
slice  := append(slice1, slice2...)
```

Courir dans le terrain de jeu Go

Suppression d'éléments / tranches "Slicing"

Si vous devez supprimer un ou plusieurs éléments d'une tranche ou si vous devez travailler avec une sous-tranche d'un autre existant; vous pouvez utiliser la méthode suivante.

> Les exemples suivants utilisent slice of int, mais cela fonctionne avec tous les types de tranche.

Donc, pour cela, nous avons besoin d'une tranche, à partir de laquelle nous allons supprimer certains éléments:

```
slice := []int{1, 2, 3, 4, 5, 6}
// > [1 2 3 4 5 6]
```

Nous avons également besoin des index d'éléments à supprimer:

```
// index of first element to remove (corresponding to the '3' in the slice)
var first = 2

// index of last element to remove (corresponding to the '5' in the slice)
var last = 4
```

Et nous pouvons "découper" la tranche en supprimant les éléments indésirables:

```
// keeping elements from start to 'first element to remove' (not keeping first to remove),
// removing elements from 'first element to remove' to 'last element to remove'
// and keeping all others elements to the end of the slice
newSlice1 := append(slice[:first], slice[last+1:]...)
// > [1 2 6]

// you can do using directly numbers instead of variables
newSlice2 := append(slice[:2], slice[5:]...)
// > [1 2 6]

// Another way to do the same
newSlice3 := slice[:first + copy(slice[first:], slice[last+1:])]
// > [1 2 6]

// same that newSlice3 with hard coded indexes (without use of variables)
newSlice4 := slice[:2 + copy(slice[2:], slice[5:])]
// > [1 2 6]
```

Pour supprimer un seul élément, il suffit de mettre l'index de cet élément en tant que premier ET en tant que dernier index à supprimer, comme cela:

```
var indexToRemove = 3
newSlice5 := append(slice[:indexToRemove], slice[indexToRemove+1:]...)
// > [1 2 3 5 6]

// hard-coded version:
newSlice5 := append(slice[:3], slice[4:]...)
// > [1 2 3 5 6]
```

Et vous pouvez également supprimer des éléments au début de la tranche:

```
newSlice6 := append(slice[:0], slice[last+1:]...)
// > [6]

// That can be simplified into
newSlice6 := slice[last+1:]
// > [6]
```

Vous pouvez également supprimer certains éléments de la fin de la tranche:

```
newSlice7 := append(slice[:first], slice[first+1:len(slice)-1]...)
```

```
// > [1 2]

// That can be simplified into
newSlice7 := slice[:first]
// > [1 2]
```

Si la nouvelle tranche doit contenir exactement les mêmes éléments que le premier, vous pouvez utiliser la même chose mais avec `last := first-1` .
(Cela peut être utile si vos index sont calculés précédemment)

Longueur et capacité

Les tranches ont à la fois la longueur et la capacité. La longueur d'une tranche correspond au nombre d'éléments *présents* dans la tranche, tandis que la capacité correspond au nombre d'éléments que la tranche *peut contenir* avant de devoir être réallouée.

Lors de la création d'une tranche à l'aide de la fonction `make()` intégrée, vous pouvez spécifier sa longueur et éventuellement sa capacité. Si la capacité n'est pas explicitement spécifiée, il s'agira de la longueur spécifiée.

```
var s = make([]int, 3, 5) // length 3, capacity 5
```

Vous pouvez vérifier la longueur d'une tranche avec la fonction `len()` intégrée:

```
var n = len(s) // n == 3
```

Vous pouvez vérifier la capacité avec la fonction `cap()` intégrée:

```
var c = cap(s) // c == 5
```

Les éléments créés par `make()` sont définis sur la valeur zéro pour le type d'élément de la tranche:

```
for idx, val := range s {
    fmt.Println(idx, val)
}
// output:
// 0 0
// 1 0
// 2 0
```

Exécutez-le sur play.golang.org

Vous ne pouvez pas accéder aux éléments au-delà de la longueur d'une tranche, même si l'index est dans la capacité:

```
var x = s[3] // panic: runtime error: index out of range
```

Cependant, tant que la capacité dépasse la longueur, vous pouvez ajouter de nouveaux éléments sans réallouer:

```
var t = []int{3, 4}
s = append(s, t) // s is now []int{0, 0, 0, 3, 4}
n = len(s) // n == 5
c = cap(s) // c == 5
```

Si vous ajoutez à une tranche qui n'a pas la capacité d'accepter les nouveaux éléments, la baie sous-jacente sera réallouée pour vous avec une capacité suffisante:

```
var u = []int{5, 6}
s = append(s, u) // s is now []int{0, 0, 0, 3, 4, 5, 6}
n = len(s) // n == 7
c = cap(s) // c > 5
```

Il est donc généralement recommandé d'allouer une capacité suffisante lors de la première création d'une tranche, si vous savez combien d'espace vous aurez besoin pour éviter des réaffectations inutiles.

Copier le contenu d'une tranche vers une autre tranche

Si vous souhaitez copier le contenu d'une tranche dans une tranche initialement vide, vous pouvez suivre les étapes suivantes pour y parvenir.

1. Créez la tranche source:

```
var sourceSlice []interface{} = []interface{}{"Hello",5.10,"World",true}
```

2. Créez la tranche de destination avec:

- Longueur = Longueur de sourceSlice

```
var destinationSlice []interface{} = make([]interface{},len(sourceSlice))
```

3. Maintenant que le tableau sous-jacent de la tranche de destination est suffisamment grand pour accueillir tous les éléments de la tranche source, nous pouvons procéder à la copie des éléments en utilisant la `copy` intégrée:

```
copy(destinationSlice,sourceSlice)
```

Créer des tranches

Les tranches sont la façon typique dont les programmeurs vont stocker les listes de données.

Pour déclarer une variable slice, utilisez la syntaxe `[]Type` .

```
var a []int
```

Pour déclarer et initialiser une variable de tranche dans une ligne, utilisez la syntaxe `[]Type{values}` .

```
var a []int = []int{3, 1, 4, 1, 5, 9}
```

Une autre façon d'initialiser une tranche consiste à `make` fonction `make` . Il y a trois arguments: le `Type` de la tranche (ou la carte), la `length` et la `capacity` .

```
a := make([]int, 0, 5)
```

Vous pouvez ajouter des éléments à votre nouvelle tranche en utilisant `append` .

```
a = append(a, 5)
```

Vérifiez le nombre d'éléments dans votre tranche en utilisant `len` .

```
length := len(a)
```

Vérifiez la capacité de votre tranche à l'aide du `cap` . La capacité correspond au nombre d'éléments actuellement alloués à la mémoire pour la tranche. Vous pouvez toujours ajouter à une tranche à sa capacité car Go créera automatiquement une tranche plus grande pour vous.

```
capacity := cap(a)
```

Vous pouvez accéder aux éléments d'une tranche en utilisant une syntaxe d'indexation classique.

```
a[0]  // Gets the first member of `a`
```

Vous pouvez également utiliser une boucle `for` sur des tranches avec `range` . La première variable est l'index dans le tableau spécifié et la deuxième variable est la valeur de l'index.

```
for index, value := range a {
    fmt.Println("Index: " + index + " Value: " + value)  // Prints "Index: 0 Value: 5" (and
continues until end of slice)
}
```

Aller au terrain de jeu

Filtrer une tranche

Pour filtrer une tranche sans allouer un nouveau tableau sous-jacent:

```
// Our base slice
slice := []int{ 1, 2, 3, 4 }
// Create a zero-length slice with the same underlying array
tmp := slice[:0]

for _, v := range slice {
  if v % 2 == 0 {
    // Append desired values to slice
    tmp = append(tmp, v)
  }
```

```
}

// (Optional) Reassign the slice
slice = tmp // [2, 4]
```

Valeur zéro de la tranche

La valeur zéro de la tranche est `nil` , ce qui a la longueur et la capacité `0` . Une tranche `nil` n'a pas de tableau sous-jacent. Mais il y a aussi des tranches non nul de longueur et de la capacité `0` , comme `[]int{}` ou `make([]int, 5)[5:]` .

Tout type ayant des valeurs nulles peut être converti en tranche `nil` :

```
s = []int(nil)
```

Pour tester si une tranche est vide, utilisez:

```
if len(s) == 0 {
    fmt.Ptintf("s is empty.")
}
```

Lire Tranches en ligne: https://riptutorial.com/fr/go/topic/733/tranches

Chapitre 72: Valeurs nulles

Remarques

Une chose à noter - les types qui ont une valeur nulle non nulle comme les chaînes, les ints, les flottants, les bools et les structures ne peuvent pas être définis sur nil.

Examples

Valeurs de base zéro

Les variables dans Go sont toujours initialisées, que vous leur donniez une valeur de départ ou non. Chaque type, y compris les types personnalisés, a une valeur de zéro pour laquelle il est défini s'il n'y a pas de valeur.

```
var myString string      // "" - an empty string
var myInt int64          // 0 - applies to all types of int and uint
var myFloat float64      // 0.0 - applies to all types of float and complex
var myBool bool          // false
var myPointer *string    // nil
var myInter interface{}  // nil
```

Cela s'applique également aux cartes, aux tranches, aux canaux et aux types de fonctions. Ces types seront initialisés à zéro. Dans les tableaux, chaque élément est initialisé à la valeur zéro de son type respectif.

Plus de valeurs zéro complexes

Dans les tranches, la valeur zéro est une tranche vide.

```
var myIntSlice []int    // [] - an empty slice
```

Utilisez `make` pour créer une tranche remplie de valeurs, toutes les valeurs créées dans la tranche sont définies sur la valeur zéro du type de la tranche. Par exemple:

```
myIntSlice := make([]int, 5)    // [0, 0, 0, 0, 0] - a slice with 5 zeroes
fmt.Println(myIntSlice[3])
// Prints 0
```

Dans cet exemple, `myIntSlice` est une tranche `int` contenant 5 éléments qui sont tous à 0 car c'est la valeur zéro pour le type `int` .

Vous pouvez également créer une tranche avec `new` , cela créera un pointeur sur une tranche.

```
myIntSlice := new([]int)        // &[] - a pointer to an empty slice
*myIntSlice = make([]int, 5)    // [0, 0, 0, 0, 0] - a slice with 5 zeroes
fmt.Println((*myIntSlice)[3])
```

```
// Prints 0
```

Remarque: les pointeurs de tranche ne prennent pas en charge l'indexation, vous ne pouvez donc pas accéder aux valeurs à l'aide de `myIntSlice[3]` , mais plutôt `(*myIntSlice)[3]` .

Struct Zero Values

Lors de la création d'une structure sans l'initialiser, chaque champ de la structure est initialisé à sa valeur zéro respective.

```
type ZeroStruct struct {
    myString string
    myInt    int64
    myBool   bool
}

func main() {
    var myZero = ZeroStruct{}
    fmt.Printf("Zero values are: %q, %d, %t\n", myZero.myString, myZero.myInt, myZero.myBool)
    // Prints "Zero values are: "", 0, false"
}
```

Valeurs de tableau zéro

Comme pour le blog Go :

> Les tableaux n'ont pas besoin d'être initialisés explicitement; la valeur zéro d'un tableau est un tableau prêt à l'emploi dont les éléments sont eux-mêmes mis à zéro

Par exemple, `myIntArray` est initialisé avec la valeur zéro de `int` , qui est 0:

```
var myIntArray [5]int     // an array of five 0's: [0, 0, 0, 0, 0]
```

Lire Valeurs nulles en ligne: https://riptutorial.com/fr/go/topic/6069/valeurs-nulles

Chapitre 73: Valeurs nulles

Examples

Explication

Des valeurs nulles ou une initialisation nulle sont simples à mettre en œuvre. Venant de langages comme Java, il peut sembler compliqué que certaines valeurs puissent être `nil` alors que d'autres ne le sont pas. En résumé de la valeur zéro: la spécification du langage de programmation Go :

> Les pointeurs, les fonctions, les interfaces, les tranches, les canaux et les cartes sont les seuls types pouvant être nuls. Les autres sont initialisés avec des chaînes fausses, nulles ou vides en fonction de leurs types respectifs.

Si une fonction vérifie certaines conditions, des problèmes peuvent survenir:

```
func isAlive() bool {
    //Not implemented yet
    return false
}
```

La valeur zéro sera fausse avant l'implémentation. Les tests unitaires dépendant du retour de cette fonction peuvent donner des faux positifs / négatifs.

Une solution typique consiste à renvoyer une erreur idiomatique dans Go:

```
package main

import "fmt"

func isAlive() (bool, error) {
    //Not implemented yet
    return false, fmt.Errorf("Not implemented yet")
}

func main() {
    _, err := isAlive()
    if err != nil {
        fmt.Printf("ERR: %s\n", err.Error())
    }

}
```

jouer sur le terrain de jeu

Lorsque vous retournez à la fois une structure et une erreur, vous avez besoin d'une structure utilisateur pour le retour, ce qui n'est pas très élégant. Il y a deux contre-options:

- Travailler avec des interfaces: renvoyer nil en renvoyant une interface.
- Travailler avec des pointeurs: un pointeur **peut** être `nil`

Par exemple, le code suivant renvoie un pointeur:

```
func(d *DB) GetUser(id uint64) (*User, error) {
    //Some error ocurred
    return nil, err
}
```

Lire Valeurs nulles en ligne: https://riptutorial.com/fr/go/topic/6379/valeurs-nulles

Chapitre 74: XML

Remarques

Bien que de nombreuses utilisations du package `encoding/xml` incluent le marshaling et le unmarshaling sur une `struct` Go, il convient de noter qu'il ne s'agit pas d'un mapping direct. La documentation du package indique:

> Le mappage entre des éléments XML et des structures de données est intrinsèquement vicié: un élément XML est une collection de valeurs anonymes dépendant de l'ordre, tandis qu'une structure de données est une collection de valeurs nommées indépendantes de l'ordre.

Pour des paires clé-valeur simples, non ordonnées, l'utilisation d'un encodage différent tel que Gob ou JSON peut être plus adaptée. Pour les données ordonnées ou les flux de données basés sur des événements / rappels, XML peut être le meilleur choix.

Examples

Décodage / désarchivage de base d'éléments imbriqués avec des données

Les éléments XML nichent souvent, ont des données dans des attributs et / ou des données de caractères. La manière de capturer ces données consiste à utiliser respectivement `,attr` et `,chardata` pour ces cas.

```
var doc = `
<parent>
  <child1 attr1="attribute one"/>
  <child2>and some cdata</child2>
</parent>
`

type parent struct {
    Child1 child1 `xml:"child1"`
    Child2 child2 `xml:"child2"`
}

type child1 struct {
    Attr1 string `xml:"attr1,attr"`
}

type child2 struct {
    Cdata1 string `xml:",cdata"`
}

func main() {
    var obj parent
    err := xml.Unmarshal([]byte(doc), &obj)
    if err != nil {
        log.Fatal(err)
    }
```

```
    fmt.Println(obj.Child2.Cdata1)

}
```

Playground

Lire XML en ligne: https://riptutorial.com/fr/go/topic/1846/xml

Chapitre 75: YAML

Examples

Création d'un fichier de configuration au format YAML

```
import (
    "io/ioutil"
    "path/filepath"

    "gopkg.in/yaml.v2"
)

func main() {
    filename, _ := filepath.Abs("config/config.yml")
    yamlFile, err := ioutil.ReadFile(filename)
    var config Config
    err = yaml.Unmarshal(yamlFile, &config)
    if err != nil {
        panic(err)
    }
    //env can be accessed from config.Env
}

type Config struct {
    Env                 string `yaml:"env"`
}

//config.yml should be placed in config/config.yml for example, and needs to have the
following line for the above example:
//env: test
```

Lire YAML en ligne: https://riptutorial.com/fr/go/topic/2503/yaml

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec Go	4444, alejosocorro, Alexander, Amitay Stern, Andrej Bencic, Andrii Abramov, burfl, Burhan Ali, cat, Cody Gustafson, Community, David G., Dmitri Goldring, Feckmore, Florian Hämmerle, Franck Dernoncourt, Gerep, Greg Bray, hellyale, Hunter, James Taylor, Jared Hooper, Jon Chan, Katamaritaco, Mark Henderson, Matt, mbb, MegaTom, mmlb, mnoronha, mohan08p, Nir, nix, nouney, patterns, Pavel Nikolov, ProfNandaa, Quentin Skousen, Radouane ROUFID, Rahul Nair, RamenChef, raulsntos, Sam Whited, seriousdev, Simone Carletti, skunkmb, sztanpet, Tanmay Garg, Topo, Unapiedra, Vikash, Xavier Nicollet
2	Analyse de fichiers CSV	Ainar-G
3	Analyse des arguments et des drapeaux de ligne de commande	Ingve, Pavel Kazhevets, Sam Whited
4	Boucles	1lann, burfl, Community, ivan73, jayantS, Jon Chan, mgh, MohamedAlaa, RamenChef, Sam Whited, Steven Maude, Thomas Gerot
5	Brancher	Sam Whited
6	Canaux	Chris Lucas, Howl, Jeremy, Kwarrtz, metmirr, RamenChef, Rodolfo Carvalho, Zoyd
7	cgo	MaC, Vojtech Kane
8	Chaîne	Ainar-G, NatNgs, raulsntos
9	Client HTTP	1lann, dmportella, Lanzafame, Sam Whited, SommerEngineering
10	Compilation croisée	Jordan, Katamaritaco, mbb, mohan08p, RamenChef, Riley Guerin, SH', Siu Ching Pong -Asuka Kenji-, SommerEngineering, sztanpet, Zoyd
11	Concurrence	Chris Lucas, Community, Florian Hämmerle, flyingfinger, Grzegorz Żur, Harshal Sheth, Ilya, Inanc Gumus, Kyle Brandt,

		Nathan Osman, Roland Illig, Ryan Kelln, Tim S. Van Haren, VonC, zianwar, Zoyd
12	Console I / O	Abhilekh Singh
13	Construire des contraintes	4444, RamenChef, Sam Whited, seriousdev
14	Conversions de type	Adrian, Florian Hämmerle
15	Cryptographie	SommerEngineering
16	Développement pour plusieurs plates-formes avec compilation conditionnelle	ecem
17	Encodage Base64	Nathan Osman, RamenChef, Sam Whited
18	Enregistrement	Grzegorz Żur, Jon Chan, Nathan Osman, Pavel Kazhevets, Sam Whited
19	Envoyer / recevoir des emails	Utahcon
20	Essai	Adrian, Ankit Deshpande, Harshal Sheth, ivan.sim, Jared Ririe, Nathan Osman, Omid, Pavel Nikolov, Rodolfo Carvalho, seriousdev, Toni Villena, Zoyd
21	Exécution des commandes	Krzysztof Kowalczyk, Kyle Brandt, Nevermore
22	Expansion Inline	Sam Whited
23	Fermetures	abhink
24	Fichier I / O	1lann, Andres Kütt, greatwolf, Grzegorz Żur, koblas, noisewaterphd, Quentin Skousen, Sam Whited
25	Fmt	Lanzafame, Nevermore, Sam Whited
26	Goroutines	mohan08p
27	gueule	zola
28	Images	putu
29	Installation	sadlil

30	Interfaces	Cody Roseborough, dotctor, Francis Norton, Grzegorz Żur, icza, Ingve, meysam, Mike, ptman, sadlil, Sam Whited, Wendy Adi
31	Iota	4444, Florian Hämmerle, Ingve, mohan08p, Sam Whited, Wojciech Kazior, Zoyd
32	JSON	Dmitry Udod, Joe, Jon Chan, Kyle Brandt, Nathan Osman, RamenChef, Sam Whited, shayan, Simone Carletti, sztanpet, Tanmay Garg, Utahcon
33	JWT Authorization in Go	AniSkywalker
34	La gestion des erreurs	bowsersenior, elevine, Elijah Sarver, Florian Hämmerle, groob, Ingve, Joe, Kin, Paul Hankin, Quentin Skousen, Sam Whited, Simone Carletti, Sridhar, Surreal Dreams, Vervious, Zoyd
35	La vente	Abhilekh Singh, Boris Le Méec, burfl, Dmitri Goldring, Ivan Mikushin, Mark Henderson, Martin Campbell, Michael, Sam Whited, Vardius
36	Le contexte	Ingaz, Sam Whited
37	Le Go Command	ganesh kumar, Harshal Sheth, Ingve, Lanzafame, Mayank Patel , Nevermore, Quentin Skousen, Sam Whited, theflametrooper, Vikash
38	Lecteurs	Mike Houston
39	Les constantes	Pavel Nikolov, RamenChef, Sam Whited, Simone Carletti
40	Les fonctions	Boris Le Méec, Dmytro Sadovnychyi, Grzegorz Żur, jayantS, LeoTao, Nathan Osman, nouney, palestamp, RamenChef, Right leg, Thomas Gerot
41	Les méthodes	ganesh kumar, Pavel Kazhevets
42	Les variables	Community, FredMaggiowski, Jon Chan, Simone Carletti
43	Meilleures pratiques sur la structure du projet	Iman Tumorang
44	mgo	Florian Hämmerle, Sourabh
45	Middleware	Ankit Deshpande
46	Modèles	Pavel Kazhevets, RamenChef, Tanmay Garg
47	Mutex	Adrian, Prutswonder

48	Panique et récupérer	JunLe Meng, Kaedys, Kristoffer Sall-Storgaard, Sam Whited
49	Paquets	dmportella, Grzegorz Żur, icza, Michael, Nathan Osman, RadicalFish, RamenChef, skunkmb, tkausl
50	Plans	Abhay, abhink, Amitay Stern, Brendan, burfl, chowey, Chris Lucas, cizixs, Community, creker, Dair, Dmitri Goldring, gbulmer , Hugo, James, JepZ, Joe, Kaedys, Kamil Kisiel, Kyle Brandt, Mark Henderson, matt.s, Milo Christiansen, NatNgs, Oleg Sklyar, radbrawler, RamenChef, Roland Illig, Sam Whited, seh, Simone Carletti, skunkmb, Surreal Dreams, Vojtech Kane, Zoyd , Zyerah
51	Pointeurs	David Hoelzer, Jon Chan, Joost, Mal Curtis, metmirr, Nevermore, skunkmb
52	Pool de mémoire	Elijah Sarver, Grzegorz Żur, Kenny Grant
53	Pools de travailleurs	burfl, photoionized, seriousdev
54	Premiers pas avec Go en utilisant Atom	Ali M, Danny Chen, Katamaritaco
55	Profilage avec go tool pprof	mbb, Nevermore, radbrawler
56	Programmation orientée objet	Davyd Dzhahaiev, Sam Whited, zola
57	Protobuf in Go	mohan08p
58	Ramification	burfl, Community, ganesh kumar, Ingve, nk2ge5k
59	Réflexion	ganesh kumar, mammothbane, radbrawler
60	Reporter	abhink, Adrian, Sam Whited, Vikash
61	Sélectionner et canaux	Harshal Sheth, Kaedys, RamenChef, Sam Whited, Utahcon
62	Serveur HTTP	Chief, frigo americain, Jon Erickson, Kin, Nathan Osman, rogerdpack, Sam Whited, Sascha, seriousdev, Simone Carletti, SommerEngineering, Tanmay Garg, Zhinkk
63	Signaux OS	Community, Sam Whited, Utahcon
64	SQL	Adrian, artamonovdev, bernardn, Francesco Pasa, Nevermore, Sam Whited, Sascha, Tanmay Garg, wrfly
65	Structs	abhink, Amitay Stern, Anthony Atkinson, Blixt, burfl, cizixs,

		Community, FredMaggiowski, Howl, Ingve, Kin, MaC, Mark Henderson, matt.s, mohan08p, Nathan Osman, nouney, Patrick, Quentin Skousen, radbrawler, RamenChef, Roland Illig, Simone Carletti, sunkuet02, Vojtech Kane, Wojciech Kazior
66	Tableaux	NatNgs, nouney, Noval Agung Prayogo, Sam Whited
67	Temps	Lanzafame, NatNgs, raulsntos
68	Texte + HTML Templating	Stephen Rudolph
69	Tranches	1lann, Benjamin Kadish, burfl, cizixs, Grzegorz Żur, Guillaume, Jared Hooper, Joost, Jukurrpa, Kyle Brandt, Mark Henderson, NatNgs, RamenChef, Simone Carletti, skunkmb, Tanmay Garg, Zoyd
70	Valeurs nulles	Harshal Sheth, raulsntos, Surreal Dreams
71	XML	ivarg, Sam Whited
72	YAML	Nathan Osman, Orr, Sam Whited