EBook Gratuito

APPRENDIMENTO Go

Free unaffiliated eBook created from
Stack Overflow contributors.

#go

Sommario

Di

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

Capitolo 1: Iniziare con Go

Osservazioni

Go è un linguaggio open source, compilato, tipizzato staticamente nella tradizione di Algol e C. Vanta funzionalità come la garbage collection, la tipizzazione strutturale limitata, le funzionalità di sicurezza della memoria e la programmazione simultanea di stile CSP di facile utilizzo.

Versioni

L'ultima versione della versione principale è in grassetto qui sotto. La cronologia completa del rilascio può essere trovata **qui** .

Versione	Data di rilascio
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

Ciao mondo!

Inserisci il seguente codice nel nome di un file `hello.go` :

```
package main

import "fmt"

func main() {
```

```
    fmt.Println("Hello, □□")
}
```

Terreno di gioco

Quando Go è installato correttamente, questo programma può essere compilato ed eseguito in questo modo:

```
go run hello.go
```

Produzione:

```
Hello, □□
```

Una volta che sei soddisfatto del codice, può essere compilato con un eseguibile eseguendo:

```
go build hello.go
```

Questo creerà un file eseguibile appropriato per il tuo sistema operativo nella directory corrente, che potrai quindi eseguire con il seguente comando:

Linux, OSX e altri sistemi simili a Unix

```
./hello
```

finestre

```
hello.exe
```

***Nota** : i caratteri cinesi sono importanti perché dimostrano che le stringhe Go vengono memorizzate come fette di byte di sola lettura.*

FizzBuzz

Un altro esempio di programmi in stile "Hello World" è FizzBuzz . Questo è un esempio di un'implementazione di FizzBuzz. Molto idiomatico Entra in gioco qui.

```
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)
        }
    }
}
```

Terreno di gioco

Elenco delle variabili d'ambiente Go

Le variabili d'ambiente che influenzano lo strumento `go` possono essere visualizzate tramite il comando `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"
```

Di default stampa l'elenco come uno script di shell; tuttavia, se uno o più nomi di variabili vengono forniti come argomenti, stampa il valore di ciascuna variabile denominata.

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

Impostazione dell'ambiente

Se Go non è preinstallato nel tuo sistema, puoi andare su https://golang.org/dl/ e scegliere la tua piattaforma per scaricare e installare Go.

Per configurare un ambiente di sviluppo Go di base, è necessario impostare solo alcune delle numerose variabili di ambiente che influiscono sul comportamento dello strumento `go` (Vedi: Elenco delle variabili d'ambiente Go per un elenco completo) (generalmente nel file `~/.profile` della shell file o equivalente su sistemi operativi Unix).

GOPATH

Come il sistema `PATH` variabile di ambiente, percorso Go è un `:` (`;` su Windows) lista di directory in cui Go cercherà per i pacchetti delimitato. Lo strumento `go get` scaricherà anche i pacchetti nella prima directory in questo elenco.

`GOPATH` è dove Go `GOPATH` cartelle `bin` , `pkg` e `src` associate necessarie per lo spazio di lavoro:

- `src` - posizione del file di origine: `.go` , `.c` , `.g` , `.s`
- `pkg` - ha compilato i file `.a`
- `bin` - contiene i file eseguibili creati da Go

Da Go 1.8 in poi, la variabile d'ambiente `GOPATH` avrà un valore predefinito se non è impostata. Il valore predefinito è $ HOME / go su Unix / Linux e% USERPROFILE% / go su Windows.

Alcuni strumenti presumono che `GOPATH` conterrà una singola directory.

GOBIN

La directory bin dove `go install` e `go get` metterà i binari dopo aver creato i pacchetti `main` . Generalmente questo è impostato su un punto del sistema `PATH` modo che i binari installati possano essere eseguiti e rilevati facilmente.

GOROOT

Questa è la posizione della tua installazione Go. È usato per trovare le librerie standard. È molto raro dover impostare questa variabile quando Go incorpora il percorso di build nella toolchain. L'impostazione di `GOROOT` è necessaria se la directory di installazione differisce dalla directory di build (o dal valore impostato durante la creazione).

Accesso alla documentazione offline

Per la documentazione completa, eseguire il comando:

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

Per un tour di Go (altamente raccomandato per i principianti nella lingua):

```
go tool tour
```

I due comandi precedenti daranno avvio ai server web con una documentazione simile a quella che si trova online qui e qui rispettivamente.

Per un controllo rapido dei riferimenti da riga di comando, ad es. Per fmt.Print:

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

La guida generale è disponibile anche dalla riga di comando:

```
go help [command]
```

Esecuzione Vai online

Il Go Playground

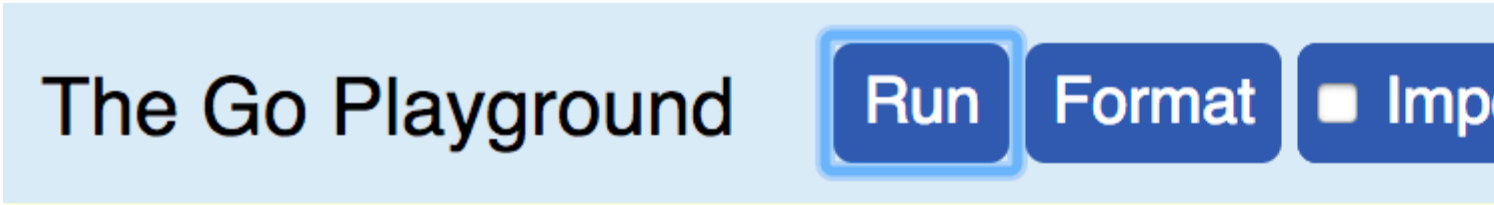
Uno strumento Go poco conosciuto è **The Go Playground** . Se si vuole sperimentare con Go senza scaricarlo, possono farlo facilmente semplicemente. . .

1. Visitando il parco giochi nel proprio browser web
2. Inserendo il loro codice
3. Cliccando su "Esegui"

Condivisione del tuo codice

Il Go Playground ha anche strumenti per la condivisione; se un utente preme il pulsante "Condividi", verrà generato un collegamento (come questo) che può essere inviato ad altre persone per testare e modificare.

In azione




```
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
```

Capitolo 2: analisi

introduzione

Go viene fornito con le proprie strutture di test che hanno tutto il necessario per eseguire test e benchmark. A differenza della maggior parte degli altri linguaggi di programmazione, spesso non è necessario un quadro di test separato, anche se alcuni esistono.

Examples

Test di 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)
    }
}
```

Per eseguire il test basta usare il comando `go test` :

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

Usa il flag `-v` per vedere i risultati di ogni test:

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

Utilizzare il percorso `./...` per verificare in modo ricorsivo le sottodirectory:

```
$ 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
```

Esegui un test particolare:
Se ci sono più test e si desidera eseguire un test specifico, può essere fatto in questo modo:

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

Esempio:

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

Test di riferimento

Se si desidera misurare parametri di riferimento, aggiungere un metodo di prova come questo:

`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)
    }
}
```

Quindi, per eseguire un semplice benchmark:

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

Test unitari basati su tabella

Questo tipo di test è una tecnica diffusa per testare valori di input e output predefiniti.

Crea un file chiamato `main.go` con il contenuto:

```
package main

import (
    "fmt"
)

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

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

Dopo averlo eseguito, vedrai che l'output è `9` . Sebbene la funzione `Sum` piuttosto semplice, è una buona idea testare il codice. Per fare ciò, creiamo un altro file chiamato `main_test.go` nella stessa cartella di `main.go` , contenente il seguente codice:

```
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)
        }
    }
}
```

Come puoi vedere, viene creata una porzione di strutture anonime, ciascuna con un insieme di input e il risultato previsto. Ciò consente di creare un gran numero di casi di test tutti insieme in un unico punto, quindi eseguiti in un ciclo, riducendo la ripetizione del codice e migliorando la chiarezza.

Test di esempio (test di auto-documentazione)

Questo tipo di test assicura che il codice venga compilato correttamente e venga visualizzato nella documentazione generata per il progetto. In aggiunta a ciò, i test di esempio possono affermare che il test produce un output corretto.

`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
}
```

Per eseguire il test, eseguire `go test` nella cartella contenente quei file OPPURE mettere questi due file in una sottocartella denominata `sum` e quindi dalla cartella padre eseguire `go test ./sum` . In entrambi i casi otterrai un risultato simile a questo:

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

Se ti stai chiedendo come questo sta testando il tuo codice, ecco un'altra funzione di esempio, che in realtà fallisce il test:

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

    // Output:
```

```
    // 5
}
```

Quando esegui il `go test` , ottieni il seguente risultato:

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

Se vuoi vedere la documentazione per il tuo pacchetto `sum` - esegui semplicemente:

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

e vai a http: // localhost: 6060 / pkg / FOLDER / sum / , dove *FOLDER* è la cartella contenente il pacchetto `sum` (in questo esempio `so`). La documentazione per il metodo sum si presenta così:

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

- Una funzione di TearDown esegue un rollback.

Questa è una buona opzione quando non è possibile modificare il database ed è necessario creare un oggetto che simuli un oggetto portato da un database o che sia necessario avviare una configurazione in ciascun test.

Un esempio stupido potrebbe essere:

```
// 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)
    }
}
```

Un altro esempio potrebbe essere la preparazione del database per testare e eseguire il rollback

```
 // 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)
    }
}
```

Visualizza la copertura del codice in formato HTML

Esegui il `go test` di `go test` normalmente, ma con la bandiera del profilo di `coverprofile` . Quindi utilizzare lo `go tool` per visualizzare i risultati in formato HTML.

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

Leggi analisi online: https://riptutorial.com/it/go/topic/1234/analisi

Capitolo 3: Analisi dei file CSV

Sintassi

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

Examples

Analisi CSV semplice

Considera questi dati CSV:

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

Questi dati possono essere letti con il seguente codice:

```
// 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])
}
```

E produrre:

```
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
```

Parco giochi: https://play.golang.org/p/d2E6-CGGIe .

Leggi Analisi dei file CSV online: https://riptutorial.com/it/go/topic/5818/analisi-dei-file-csv

Capitolo 4: Argomenti e bandiere della riga di comando di analisi

Examples

Argomenti della riga di comando

L'analisi degli argomenti della riga di comando è Go è molto simile alle altre lingue. Nel tuo codice ti basta accedere a una serie di argomenti in cui il primo argomento sarà il nome del programma stesso.

Esempio veloce:

```
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("")
}
```

E l'output sarebbe:

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

Ogni argomento è solo una stringa. Nel pacchetto `os` sembra: `var Args []string`

bandiere

Vai alla libreria standard fornisce il `flag` pacchetto che aiuta con i flag di analisi passati al programma.

Nota che il pacchetto `flag` non fornisce i normali flag stile GNU. Ciò significa che i flag di più lettere devono essere avviati con un trattino singolo come questo: `-exampleflag` , non questo: `--exampleflag` . I flag in stile GNU possono essere fatti con alcuni pacchetti di terze parti.

```
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` fa il messaggio di aiuto per noi:

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

Chiama con tutte le bandiere:

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

Chiama con bandiere mancanti:

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

Leggi Argomenti e bandiere della riga di comando di analisi online:
https://riptutorial.com/it/go/topic/4023/argomenti-e-bandiere-della-riga-di-comando-di-analisi

Capitolo 5: Array

introduzione

Le matrici sono tipi di dati specifici, che rappresentano una raccolta ordinata di elementi di un altro tipo.

In Go, gli array possono essere semplici (a volte chiamati "elenchi") o multidimensionali (ad esempio, un array di 2 dimensioni rappresenta una raccolta ordinata di matrici, che contiene elementi)

Sintassi

- var variableName [5] ArrayType // Dichiarazione di una matrice di dimensione 5.
- var variableName [2] [3] ArrayType = {{Valore1, Valore2, Valore3}, {Valore4, Valore5, Valore6}} // Dichiarazione di un array multidimensionale
- variableName: = [...] ArrayType {Value1, Value2, Value3} // Dichiara una matrice di dimensione 3 (Il compilatore conterà gli elementi dell'array per definire la dimensione)
- arrayName [2] // Ottenere il valore per indice.
- arrayName [5] = 0 // Impostazione del valore all'indice.
- arrayName [0] // Primo valore della matrice
- arrayName [len (arrayName) -1] // Ultimo valore della matrice

Examples

Creare matrici

Un array in go è una raccolta ordinata di elementi dello stesso tipo.
La notazione di base per rappresentare gli array è usare `[]` con il nome della variabile.

La creazione di un nuovo array assomiglia a `var array = [size]Type` , sostituendo le `size` con un numero (ad esempio `42` per specificare che sarà un elenco di 42 elementi) e sostituendo `Type` base al tipo di elementi che l'array può contenere (per esempio `int` o `string`)

Subito sotto è un esempio di codice che mostra il diverso modo di creare un array in 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
```

giocarci sul campo da gioco

Matrice multidimensionale

Gli array multidimensionali sono fondamentalmente array che contengono altri array come elementi.
È rappresentato come `[sizeDim1][sizeDim2]..[sizeLastDim]type` , sostituendo `sizeDim` base ai numeri corrispondenti alla lunghezza della dimensione e `type` il tipo di dati nell'array multidimensionale.

Ad esempio, `[2][3]int` rappresenta una matrice composta da **2 sottoarray** di **3 elementi tipizzati int** .
Può essere fondamentalmente la rappresentazione di una matrice di **2 righe** e **3 colonne** .

Quindi possiamo creare array di numeri di dimensioni enormi come `var values := [2017][12][31][24][60]int` per esempio se è necessario memorizzare un numero per ogni minuto dall'anno 0.

Per accedere a questo tipo di array, per l'ultimo esempio, cercando il valore di 2016-01-31 alle 19:42, si accede ai `values[2016][0][30][19][42]` (perché gli **indici di array inizia da 0** e non a 1 come giorni e mesi)

Alcuni esempi seguenti:

```
// 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"]]]]
```

Indici di matrice

È necessario accedere ai valori delle matrici utilizzando un numero che specifica la posizione del valore desiderato nell'array. Questo numero è chiamato Indice.

Gli indici iniziano da **0** e finiscono con la **lunghezza dell'array -1** .

Per accedere a un valore, devi fare qualcosa del genere: `arrayName[index]` , sostituendo "index" con il numero corrispondente al rank del valore dell'array.

Per esempio:

```
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)
```

Per impostare o modificare un valore nella matrice, il modo è lo stesso.
Esempio:

```
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]
```

Leggi Array online: https://riptutorial.com/it/go/topic/390/array

Capitolo 6: Autorizzazione JWT in Go

introduzione

I token Web JSON (JWT) sono un metodo popolare per rappresentare i reclami in modo sicuro tra due parti. Comprendere come lavorare con loro è importante quando si sviluppano applicazioni Web o interfacce di programmazione delle applicazioni.

Osservazioni

context.Context e HTTP middleware non rientrano nell'ambito di questo argomento, ma ciononostante le persone curiose e vaganti dovrebbero consultare https://github.com/goware/jwtauth , https://github.com/auth0/go-jwt- middleware e https://github.com/dgrijalva/jwt-go .

Grandi complimenti a Dave Grijalva per il suo fantastico lavoro su go-jwt.

Examples

Analisi e convalida di un token mediante il metodo di firma 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)
}
```

Produzione:

```
bar 1.4444784e+09
```

(Dalla documentazione , per gentile concessione di Dave Grijalva.)

Creazione di un token utilizzando un tipo di attestazioni personalizzato

`StandardClaim` è incorporato nel tipo personalizzato per consentire una facile codifica, analisi e convalida delle attestazioni standard.

```
tokenString := 
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.HE7fK0xOQwFEr


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)
    }
})
```

Produzione:

```
bar 15000
```

(Dalla documentazione , per gentile concessione di Dave Grijalva.)

Creazione, firma e codifica di un token JWT utilizzando il metodo di firma 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)
```

Produzione:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJuYmYiOjE0NDQ0Nzg0MDB9.u1riaD1rW97opCoAuRCTy4w58
Zk-bh7vLiRIsrpU <nil>
```

(Dalla documentazione , per gentile concessione di Dave Grijalva.)

Usando il tipo StandardClaims da solo per analizzare un token

Il tipo `StandardClaims` è progettato per essere incorporato nei tipi personalizzati per fornire funzionalità di convalida standard. Puoi usarlo da solo, ma non c'è modo di recuperare altri campi dopo l'analisi. Vedere l'esempio delle attestazioni personalizzate per l'utilizzo previsto.

```
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)
```

Produzione:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MDAwLCJpc3MiOiJ0ZXN0In0.QsODzZu3lUZMVdhbO76u3Jv02iYCvE
<nil>
```

(Dalla documentazione , per gentile concessione di Dave Grijalva.)

Analisi dei tipi di errore utilizzando i controlli 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)
```

```
}
```

Produzione:

```
Timing is everything
```

(Dalla documentazione , per gentile concessione di Dave Grijalva.)

Ottenere token dall'intestazione dell'autorizzazione 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)
    }
}
```

(Dal kit di go / kit , per gentile concessione di Peter Bourgon)

Leggi Autorizzazione JWT in Go online: https://riptutorial.com/it/go/topic/10161/autorizzazione-jwt-in-go

Capitolo 7: branching

Examples

Switch Statements

Una semplice dichiarazione di `switch` :

```
switch a + b {
case c:
    // do something
case d:
    // do something else
default:
    // do something entirely different
}
```

L'esempio sopra è equivalente a:

```
if a + b == c {
    // do something
} else if a + b == d {
    // do something else
} else {
    // do something entirely different
}
```

La clausola `default` è facoltativa e verrà eseguita se e solo se nessuno dei casi si confronta con vero, anche se non appare per ultimo, il che è accettabile. Quanto segue è semanticamente uguale al primo esempio:

```
switch a + b {
default:
    // do something entirely different
case c:
    // do something
case d:
    // do something else
}
```

Ciò potrebbe essere utile se si intende utilizzare l'istruzione `fallthrough` nella clausola `default` , che deve essere l'ultima istruzione in un caso e fa sì che l'esecuzione del programma passi al caso successivo:

```
switch a + b {
default:
    // do something entirely different, but then also do something
    fallthrough
case c:
    // do something
```

```
case d:
    // do something else
}
```

Un'espressione switch vuota è implicitamente `true` :

```
switch {
case a + b == c:
    // do something
case a + b == d:
    // do something else
}
```

Le istruzioni switch supportano una semplice istruzione simile alle istruzioni `if` :

```
switch n := getNumber(); n {
case 1:
    // do something
case 2:
    // do something else
}
```

I casi possono essere combinati in un elenco separato da virgole se condividono la stessa logica:

```
switch a + b {
case c, d:
    // do something
default:
    // do something entirely different
}
```

Se le dichiarazioni

Una semplice dichiarazione `if` :

```
if a == b {
    // do something
}
```

Si noti che non ci sono parentesi che circondano la condizione e che la parentesi graffa di apertura `{` deve essere sulla stessa linea. Quanto segue *non* verrà compilato:

```
if a == b
{
    // do something
}
```

Un'istruzione `if` che fa uso di `else` :

```
if a == b {
    // do something
} else if a == c {
    // do something else
} else {
    // do something entirely different
}
```

La documentazione di Per golang.org , "L'espressione può essere preceduta da una semplice istruzione, che viene eseguita prima che l'espressione venga valutata." Le variabili dichiarate in questa semplice istruzione sono associate all'istruzione `if` e non possono essere accessibili al di fuori di essa:

```
if err := attemptSomething(); err != nil {
    // attemptSomething() was successful!
} else {
    // attemptSomething() returned an error; handle it
}
fmt.Println(err) // compiler error, 'undefined: err'
```

Digitare istruzioni switch

Un semplice interruttore di tipo:

```
// 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
}
```

È possibile eseguire test per qualsiasi tipo, inclusi `error` , tipi definiti dall'utente, tipi di interfaccia e tipi di funzione:

```
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!")
    }
}
```

Dichiarazioni Goto

`goto` trasferisce il controllo all'istruzione con l'etichetta corrispondente all'interno della stessa funzione. L'esecuzione dell'istruzione `goto` non deve far entrare in campo le variabili che non erano già in ambito al punto del `goto` .

per esempio vedi il codice sorgente della libreria 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)
```

Dichiarazioni Break-continue

L'istruzione break, durante l'esecuzione, rende il loop corrente forzato all'uscita

pacchetto principale

```
import "fmt"

func main() {
```

```
    i:=0
    for true {
      if i>2 {
        break
        }
    fmt.Println("Iteration : ",i)
    i++
    }
}
```

L'istruzione continue, durante l'esecuzione sposta il controllo all'inizio del ciclo

```
import "fmt"

func main() {
    j:=100
    for j<110 {
     j++
     if j%2==0 {
        continue
        }
    fmt.Println("Var : ",j)
    }
}
```

Interruttore di interruzione / continuazione del circuito interno

```
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)
    }
}
```

Leggi branching online: https://riptutorial.com/it/go/topic/1342/branching

Capitolo 8: canali

introduzione

Un canale contiene valori di un determinato tipo. I valori possono essere scritti su un canale e letti da esso, e circolano all'interno del canale in ordine first-in-first-out. Vi è una distinzione tra i canali bufferizzati, che possono contenere più messaggi e canali non bufferizzati, che non possono. I canali vengono in genere utilizzati per comunicare tra le goroutine, ma sono anche utili in altre circostanze.

Sintassi

- make (chan int) // crea un canale unbuffered
- make (chan int, 5) // crea un canale bufferizzato con una capacità di 5
- chiudi (ch) // chiude un canale "ch"
- ch <- 1 // scrivi il valore di 1 in un canale "ch"
- val: = <-ch // legge un valore dal canale "ch"
- val, ok: = <-ch // sintassi alternativa; ok è un bool che indica se il canale è chiuso

Osservazioni

Un canale che tiene la struct vuota `make(chan struct{})` è un messaggio chiaro all'utente che nessuna informazione è trasmessa sul canale e che è puramente usata per la sincronizzazione.

Per quanto riguarda i canali non bufferizzati, una scrittura di canale si bloccherà fino a quando non si verificherà una lettura corrispondente da un'altra goroutine. Lo stesso vale per un blocco di lettura del canale in attesa di uno scrittore.

Examples

Usando la gamma

Quando si leggono più valori da un canale, l'utilizzo `range` è un modello comune:

```
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")
}
```

Terreno di gioco

Produzione

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

timeout

I canali sono spesso usati per implementare i timeout.

```
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.
    }
}
```

Goroutine di coordinamento

Immagina una goroutine con un processo in due passaggi, in cui il thread principale deve fare un po 'di lavoro tra ogni passaggio:

```
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
}
```

Buffered vs unbuffered

```
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")
    }
}
```

vai al parco giochi

Blocco e sblocco dei canali

Di default la comunicazione sui channcels è sincronizzata; quando invii qualche valore ci deve essere un ricevitore. Altrimenti si verificherà un `fatal error: all goroutines are asleep - deadlock!` come segue:

```
package main

import "fmt"

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

Bu c'è una soluzione uso: utilizzare i canali bufferizzati:

```
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)
}
```

Aspettando che il lavoro finisca

Una tecnica comune per l'utilizzo dei canali consiste nel creare un certo numero di lavoratori (o consumatori) da leggere dal canale. Usare un sync.WaitGroup è un modo semplice per aspettare che quei lavoratori finiscano di correre.

```
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")
}
```

Leggi canali online: https://riptutorial.com/it/go/topic/1263/canali

Capitolo 9: CGO

Examples

Cgo: Primi passi tutorial

Alcuni esempi per capire il flusso di lavoro dell'utilizzo di Go C Bindings

Che cosa

In Go puoi chiamare programmi e funzioni C usando cgo . In questo modo è possibile creare facilmente associazioni C ad altre applicazioni o librerie che forniscono API C.

Come

Tutto quello che devi fare è aggiungere una `import "C"` all'inizio del tuo programma Go **subito** dopo aver incluso il tuo programma C:

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

Con l'esempio precedente puoi usare il pacchetto `stdio` in Go.

Se hai bisogno di usare un'app che si trova nella tua stessa cartella, usi la stessa sintassi che in C (con `"` invece di `<>`)

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

> **IMPORTANTE** : **non lasciare una nuova riga tra le** istruzioni di **`include`** **e di** **`import "C"`** o si otterrà questo tipo di errori durante la compilazione:

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

L'esempio

In questa cartella puoi trovare un esempio di associazioni C. Abbiamo due "librerie" C molto semplici chiamate `hello.c` :

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

void Hello(){
```

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

Questo semplicemente stampa "ciao mondo" nella console e `sum.c`

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

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

... che accetta 2 argomenti e restituisce la sua somma (non stamparla).

Abbiamo un programma `main.go` che farà uso di questi due file. Per prima cosa li importiamo come abbiamo detto prima:

```
//main.go
package main

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

Ciao mondo!

Ora siamo pronti per utilizzare i programmi C nella nostra app Go. Proviamo prima il programma 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
}
```

Ora esegui il programma main.go usando `go run main.go` per ottenere la stampa del programma C: "Hello world!". Molto bene!

Somma di ints

Facciamo un po 'più complicato aggiungendo una funzione che riassume i suoi due argomenti.

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

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

E lo chiameremo dalla nostra precedente app 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
}
```

Dai un'occhiata alla funzione "makeSum". Riceve due parametri `int` che devono essere convertiti in C `int` prima usando la funzione `C.int` . Inoltre, il ritorno della chiamata ci darà un C `int` e un errore nel caso qualcosa sia andato storto. Abbiamo bisogno di lanciare la risposta C a int di Go utilizzando `int()` .

Prova a eseguire la nostra app di `go run main.go` utilizzando `go run main.go`

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

Generare un binario

Se provi a fare una build potresti ottenere più errori di definizione.

```
$ 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
```

Il trucco è fare riferimento al file principale direttamente quando si usa `go build` :

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

> Ricorda che puoi fornire un nome al file binario usando `-o` flag `go build -o my_c_binding main.go`

Spero che questo tutorial ti sia piaciuto.

Leggi CGO online: https://riptutorial.com/it/go/topic/6125/cgo

Capitolo 10: CGO

Examples

Chiamata funzione C da Go

Cgo consente la creazione di pacchetti Go che chiamano il codice C.
Per utilizzare `cgo` scrivere il normale codice Go che importa uno pseudo-pacchetto "C". Il codice Go può quindi fare riferimento a tipi come `C.int` o funzioni come `C.Add` .
L'importazione di "C" è immediatamente preceduta da un commento, che il commento, chiamato preambolo, viene utilizzato come intestazione durante la compilazione delle parti C del pacchetto.
Nota che non ci devono essere righe vuote tra il commento di `cgo` e la dichiarazione di importazione.
Si noti che l' `import "C"` non può essere raggruppata con altre importazioni in una dichiarazione di importazione "fattorizzata" tra parentesi. È necessario scrivere più istruzioni di importazione, ad esempio:

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

Ed è bene usare la dichiarazione importata fattorizzata, per altre importazioni, come:

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

Semplice esempio con `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
}
```

Quindi `go build` ed eseguilo, output:

```
30
```

Per creare pacchetti `cgo` , basta usare `go build` o `go install` come al solito. Lo `go tool` riconosce la speciale importazione `"C"` e utilizza automaticamente `cgo` per questi file.

Cablare il codice C in tutte le direzioni

Chiamare il codice C da 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))
}
```

Chiamare il codice Go da 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
}
```

Trattare con i puntatori di funzione

```
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
}
```

Converti tipi, strutture di accesso e aritmetica del puntatore

Dalla documentazione ufficiale di 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
```

Come usarlo:

```
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
}
```

Leggi CGO online: https://riptutorial.com/it/go/topic/6455/cgo

Capitolo 11: chiusure

Examples

Nozioni di base sulla chiusura

Una *chiusura* è una funzione presa insieme a un ambiente. La funzione è in genere una funzione anonima definita all'interno di un'altra funzione. L'ambiente è l'ambito lessicale della funzione di inclusione (l'idea di base di uno scope lessicale di una funzione sarebbe lo scopo che esiste tra le parentesi della funzione).

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

All'interno del corpo di una funzione anonima (per esempio `f`) definito all'interno di un'altra funzione (dire `g`), le variabili presenti in ambiti sia `f` e `g` sono accessibili. Tuttavia, è lo scopo di `g` che forma la parte dell'ambiente della chiusura (la parte della funzione è `f`) e di conseguenza le modifiche apportate alle variabili nello scope di `g` mantengono i loro valori (cioè l'ambiente persiste tra le chiamate a `f`) .

Considera la seguente funzione:

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

In precedenza definizione, `NaturalNumbers` ha una funzione interna `f` che `NaturalNumbers` rendimenti. All'interno di `f` , si accede alla variabile `i` definita nell'ambito di `NaturalNumbers` .

Otteniamo una nuova funzione da `NaturalNumbers` modo:

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

Ora `n` è una chiusura. È una funzione (definita da `f`) che ha anche un ambiente associato (ambito di `NaturalNumbers`).

In caso di `n` , la parte ambiente contiene solo una variabile: `i`

Poiché `n` è una funzione, può essere chiamata:

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

Come evidente dall'output precedente, ogni volta che `n` viene chiamato, incrementa `i` . `i` inizia a 0, e ogni chiamata di `n` li esegue `i++` .

Il valore di `i` viene mantenuto tra le chiamate. Cioè, l'ambiente, essendo parte della chiusura, persiste.

Chiamando di nuovo `NaturalNumbers` creerebbe e restituirà una nuova funzione. Ciò inizializzerebbe una nuova `i` entro `NaturalNumbers` . Il che significa che le forme funzionali di recente restituiti un'altra chiusura avere la stessa parte per la funzione (ancora `f`), ma un ambiente nuovo di zecca (una nuova inizializzato `i`).

```
o := NaturalNumbers()

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

Sia `n` che `o` sono chiusure contenenti la stessa parte di funzione (che dà loro lo stesso comportamento), ma ambienti diversi. Pertanto, l'uso di chiusure consente alle funzioni di accedere a un ambiente persistente che può essere utilizzato per conservare le informazioni tra le chiamate.

Un altro esempio:

```
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
```

Leggi chiusure online: https://riptutorial.com/it/go/topic/2741/chiusure

Capitolo 12: Client HTTP

Sintassi

- resp, err: = http.Get (url) // Esegue una richiesta HTTP GET con il client HTTP predefinito. Un errore non nullo viene restituito se la richiesta non riesce.
- resp, err: = http.Post (url, bodyType, body) // Fa una richiesta POST HTTP con il client HTTP predefinito. Un errore non nullo viene restituito se la richiesta non riesce.
- resp, err: = http.PostForm (url, values) // Fa una richiesta POST di form HTTP con il client HTTP predefinito. Un errore non nullo viene restituito se la richiesta non riesce.

Parametri

Parametro	Dettagli
resp	Una risposta di tipo `*http.Response` a una richiesta HTTP
sbagliare	Un `error` Se non è nulla, rappresenta un errore che si è verificato quando è stata chiamata la funzione.
url	Un URL di tipo `string` per fare una richiesta HTTP a.
tipo di corpo	Il tipo MIME di tipo `string` del carico utile del corpo di una richiesta POST.
corpo	Un `io.Reader` (implements `Read()`) che verrà letto da fino a quando non viene raggiunto un errore da inviare come carico utile del corpo di una richiesta POST.
valori	Una mappa valore-chiave di tipo `url.Values` . Il tipo sottostante è una `map[string][]string` .

Osservazioni

È importante `defer resp.Body.Close()` dopo ogni richiesta HTTP che non restituisce un errore non nullo, altrimenti le risorse saranno trapelate.

Examples

GET di base

Esegui una richiesta GET di base e stampa il contenuto di un sito (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))
}
```

OTTIENI con i parametri URL e una risposta JSON

Una richiesta per i primi 10 post StackOverflow attivi più recenti utilizzando 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)
    }
}
```

Richiesta di timeout con un contesto

1.7+

Il timeout di una richiesta HTTP con un contesto può essere eseguito solo con la libreria standard (non i sottorepos) in 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()
```

Prima dell'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()
```

Ulteriori letture

Per ulteriori informazioni sul pacchetto di `context` , vedere Contesto .

Richiesta PUT dell'oggetto JSON

Il seguente aggiorna un oggetto Utente tramite una richiesta PUT e stampa il codice di stato della richiesta:

```
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)
}
```

Leggi Client HTTP online: https://riptutorial.com/it/go/topic/1422/client-http

Capitolo 13: Codifica Base64

Sintassi

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

Osservazioni

Il pacchetto `encoding/base64` contiene diversi encoder integrati . La maggior parte degli esempi in questo documento utilizza `base64.StdEncoding` , ma qualsiasi codificatore (`URLEncoding` , `RawStdEncodign` , il proprio codificatore personalizzato, ecc.) Può essere sostituito.

Examples

Codifica

```
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==
```

Terreno di gioco

Codifica in una stringa

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

Terreno di gioco

decodifica

```
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`
```

Terreno di gioco

Decodifica di una stringa

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

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

Terreno di gioco

Leggi Codifica Base64 online: https://riptutorial.com/it/go/topic/4492/codifica-base64

Capitolo 14: Collegare

introduzione

Go fornisce un meccanismo di plugin che può essere utilizzato per collegare in modo dinamico altri codici Go in fase di runtime.

A partire da Go 1.8, è utilizzabile solo su Linux.

Examples

Definizione e utilizzo di un plugin

```
package main

import "fmt"

var V int

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

Questo può essere costruito con:

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

E poi caricato e usato dalla tua applicazione:

```
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"
```

Esempio da *The State of Go 2017* .

Leggi Collegare online: https://riptutorial.com/it/go/topic/9150/collegare

Capitolo 15: Compilazione incrociata

introduzione

Il compilatore Go può produrre binari per molte piattaforme, ovvero processori e sistemi. A differenza della maggior parte degli altri compilatori, non esiste un requisito specifico per la compilazione incrociata, è facile da usare quanto la compilazione regolare.

Sintassi

- GOOS = linux GOARCH = amd64 go build

Osservazioni

Combinazioni di target per sistema operativo e architettura supportate (fonte)

$ GOOS	$ GOARCH
androide	braccio
Darwin	386
Darwin	amd64
Darwin	braccio
Darwin	arm64
libellula	amd64
FreeBSD	386
FreeBSD	amd64
FreeBSD	braccio
linux	386
linux	amd64
linux	braccio
linux	arm64
linux	ppc64
linux	ppc64le

$ GOOS	$ GOARCH
linux	MIPS64
linux	mips64le
NetBSD	386
NetBSD	amd64
NetBSD	braccio
openbsd	386
openbsd	amd64
openbsd	braccio
plan9	386
plan9	amd64
solaris	amd64
finestre	386
finestre	amd64

Examples

Compilare tutte le architetture usando un Makefile

Questo Makefile eseguirà il cross-compile e comprimerà i file eseguibili per Windows, Mac e Linux (ARM e 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)
```

(sii prudente che Makefile abbia bisogno di schede rigide e non spazi)

Semplice compilazione incrociata con go build

Dalla directory del progetto, eseguire il comando `go build` e specificare il sistema operativo e la destinazione dell'architettura con le variabili di ambiente `GOOS` e `GOARCH` :

Compilazione per Mac (64 bit):

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

Compilazione per il processore x86 di Windows:

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

Si potrebbe anche voler impostare manualmente il nome file dell'eseguibile di output per tenere traccia dell'architettura:

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

Dalla versione 1.7 in poi è possibile ottenere un elenco di tutte le possibili combinazioni GOOS e GOARCH con:

```
go tool dist list
```

(o per un consumo più semplice della macchina, `go tool dist list -json`)

Compilazione incrociata usando gox

Un'altra soluzione conveniente per la compilazione incrociata è l'utilizzo di `gox` : https://github.com/mitchellh/gox

Installazione

L'installazione viene eseguita molto facilmente eseguendo `go get github.com/mitchellh/gox` . L'eseguibile risultante viene inserito nella directory binaria di Go, ad esempio `/golang/bin` o `~/golang/bin` . Assicurati che questa cartella sia parte del tuo percorso per poter utilizzare il comando `gox` da una posizione arbitraria.

uso

All'interno della cartella radice di un progetto Go (dove si esegue es. `go build`), eseguire `gox` per creare tutti i possibili binari per qualsiasi architettura (ad esempio x86, ARM) e il sistema operativo (ad es. Linux, macOS, Windows) che è disponibile.

Per costruire per un determinato sistema operativo, usa ad esempio `gox -os="linux"` . Anche l'opzione architettura potrebbe essere definita: `gox -osarch="linux/amd64"` .

Semplice esempio: compila helloworld.go per l'architettura arm sulla macchina Linux

Preparare helloworld.go (trova sotto)

```
package main

import "fmt"

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

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

Copia il file `helloworld` (arm eseguibile) generato nel computer di destinazione.

Leggi Compilazione incrociata online: https://riptutorial.com/it/go/topic/1020/compilazione-

incrociata

Capitolo 16: Concorrenza

introduzione

In Go, la concorrenza viene raggiunta attraverso l'uso di goroutine e la comunicazione tra le goroutine viene solitamente eseguita con i canali. Tuttavia, sono disponibili altri mezzi di sincronizzazione, come mutex e wait groups, che dovrebbero essere utilizzati ogni volta che sono più convenienti dei canali.

Sintassi

- go doWork () // esegue la funzione doWork come una goroutine
- ch: = make (chan int) // dichiara un nuovo canale di tipo int
- ch <- 1 // invio su un canale
- value = <-ch // ricezione da un canale

Osservazioni

Le goroutine in Go sono simili alle discussioni in altre lingue in termini di utilizzo. Internamente, Go crea un numero di thread (specificato da `GOMAXPROCS`) e quindi pianifica le goroutine per l'esecuzione sui thread. Grazie a questo design, i meccanismi di concorrenza di Go sono molto più efficienti dei thread in termini di utilizzo della memoria e tempo di inizializzazione.

Examples

Creazione di goroutine

Qualsiasi funzione può essere invocata come una goroutine mediante il prefisso della sua chiamata con la parola chiave `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
```

Si noti che il valore di ritorno della funzione è ignorato.

Ciao World Goroutine

canale singolo, goroutine singola, una scrittura, una lettura.

```
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)
}
```

Eseguilo sul campo da gioco

Il canale `ch` è un **canale senza buffer o sincrono** .

Il `time.Sleep` è qui per illustrare la funzione `main()` **aspetterà** sul canale `ch` , che significa che la funzione letterale eseguita come una goroutine ha il tempo di inviare un valore attraverso quel canale: l' operatore di ricezione `<-ch` bloccherà l'esecuzione di `main()` . Se così non fosse, la goroutine verrebbe uccisa quando `main()` uscisse e non avrebbe il tempo di inviare il suo valore.

In attesa di goroutine

I programmi Go terminano quando termina la funzione `main` , quindi è prassi comune aspettare che tutte le goroutine finiscano. Una soluzione comune per questo è utilizzare un oggetto 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")
}
```

Esegui l'esempio nel parco giochi

Uso WaitGroup in ordine di esecuzione:

1. Dichiarazione di variabile globale. Rendendolo globale è il modo più semplice per renderlo visibile a tutte le funzioni e i metodi.
2. Aumentare il contatore. Questo deve essere fatto nella goroutine principale perché non vi è alcuna garanzia che una goroutine appena avviata venga eseguita prima delle 4 a causa delle garanzie del modello di memoria.
3. Diminuendo il contatore. Questo deve essere fatto all'uscita di una goroutine. Usando una chiamata differita, ci assicuriamo che venga chiamata ogni volta che la funzione termina , indipendentemente da come finisce.
4. Aspettando che il contatore raggiunga 0. Questo deve essere fatto nella goroutine principale per impedire che il programma esca prima che tutte le goroutine siano finite.

* I parametri vengono valutati prima di iniziare una nuova goroutine . Quindi è necessario definire i loro valori esplicitamente prima di `wg.Add(10)` modo che il codice eventualmente panico non aumenti il contatore. Aggiungendo 10 elementi al WaitGroup, quindi attenderà 10 elementi prima che `wg.Wait` restituisca il controllo alla goroutine `main()` . Qui, il valore di i è definito nel ciclo for.

Usando chiusure con goroutine in un ciclo

Quando si trova in un ciclo, la variabile di ciclo (val) nell'esempio seguente è una variabile singola che cambia valore mentre passa sopra il ciclo. Pertanto si deve fare quanto segue per passare effettivamente ogni val di valori alla goroutine:

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

Se dovessi fare solo go `func(val interface{}) { ... }()` senza passare val, allora il valore di `val` sarà qualunque cosa valga quando le goroutine vengono effettivamente eseguite.

Un altro modo per ottenere lo stesso effetto è:

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

`val := val` dall'aspetto strano `val := val` crea una nuova variabile in ogni iterazione, a cui accede la goroutine.

Fermare le goroutine

```
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 con due goroutine

```
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)
    }
}
```

Esegui una versione leggermente modificata di questo codice in Go Playground

Leggi Concorrenza online: https://riptutorial.com/it/go/topic/376/concorrenza

Capitolo 17: Console I / O

Examples

Leggi l'input dalla console

Utilizzando `scanf`

> Scanf esegue la scansione del testo letto dall'input standard, memorizzando i successivi valori separati dallo spazio in argomenti successivi determinati dal formato. Restituisce il numero di elementi sottoposti a scansione. Se questo è inferiore al numero di argomenti, err indicherà il perché. I newline nell'input devono corrispondere a newlines nel formato. L'unica eccezione: il verbo% c esegue sempre la scansione della runa successiva nell'input, anche se si tratta di uno spazio (o scheda, ecc.) O newline.

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

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

Utilizzo della `scan`

> Scansione esegue la scansione del testo letto dallo standard input, memorizzando i successivi valori separati dallo spazio in argomenti successivi. I Newlines contano come spazio. Restituisce il numero di elementi sottoposti a scansione. Se questo è inferiore al numero di argomenti, err indicherà il perché.

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

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

Usando `scanln`

> Sscanln è simile a Sscan, ma interrompe la scansione su una nuova riga e dopo l'elemento finale deve esserci una nuova riga o EOF.

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

Usando il `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())
}
```

Leggi Console I / O online: https://riptutorial.com/it/go/topic/9741/console-i---o

Capitolo 18: Contesto

Sintassi

- digitare CancelFunc func ()
- func Contesto () Contesto
- func TODO () Contesto
- func WithCancel (parent Context) (ctx Context, annulla CancelFunc)
- func WithDeadline (Parent context, deadline time.Time) (Context, CancelFunc)
- func WithTimeout (Parent context, timeout time.Duration) (Context, CancelFunc)
- func WithValue (contesto padre, interfaccia chiave {}, interfaccia val {})

Osservazioni

Il pacchetto di `context` (in Go 1.7) o il pacchetto `golang.org/x/net/context` (Pre 1.7) è un'interfaccia per la creazione di contesti che possono essere utilizzati per trasportare valori e scadenze dell'ambito della richiesta tra i confini dell'API e tra i servizi, nonché come una semplice implementazione di detta interfaccia.

a parte: la parola "contesto" viene usata vagamente per riferirsi all'intero albero, o alle singole foglie dell'albero, ad es. il `context.Context` attuale. Valori contenuti.

Ad un livello elevato, un contesto è un albero. Le nuove foglie vengono aggiunte all'albero quando vengono costruite (un `context.Context` con un valore padre) e le foglie non vengono mai rimosse dall'albero. Qualsiasi contesto ha accesso a tutti i valori sopra di esso (l'accesso ai dati scorre solo verso l'alto), e se qualsiasi contesto viene cancellato i suoi figli vengono anche cancellati (i segnali di annullamento si propagano verso il basso). Il segnale di annullamento è implementato mediante una funzione che restituisce un canale che verrà chiuso (leggibile) quando il contesto viene cancellato; questo rende i contesti un modo molto efficace per implementare la pipeline e il pattern di concorrenza di cancellazione , o timeout.

Per convenzione, le funzioni che prendono un contesto hanno il primo argomento `ctx context.Context` . Anche se questa è solo una convenzione, è una che dovrebbe essere seguita poiché molti strumenti di analisi statica cercano specificamente questo argomento. Dal momento che Context è un'interfaccia, è anche possibile trasformare dati contestuali esistenti (i valori che passano attraverso una catena di chiamate di richieste) in un normale contesto di Go e utilizzarli in un modo compatibile con le versioni precedenti implementando solo alcuni metodi. Inoltre, i contesti sono sicuri per l'accesso simultaneo in modo da poterli usare da molte goroutine (indipendentemente dal fatto che siano in esecuzione su thread paralleli o come coroutine concorrenti) senza paura.

Ulteriori letture

- https://blog.golang.org/context

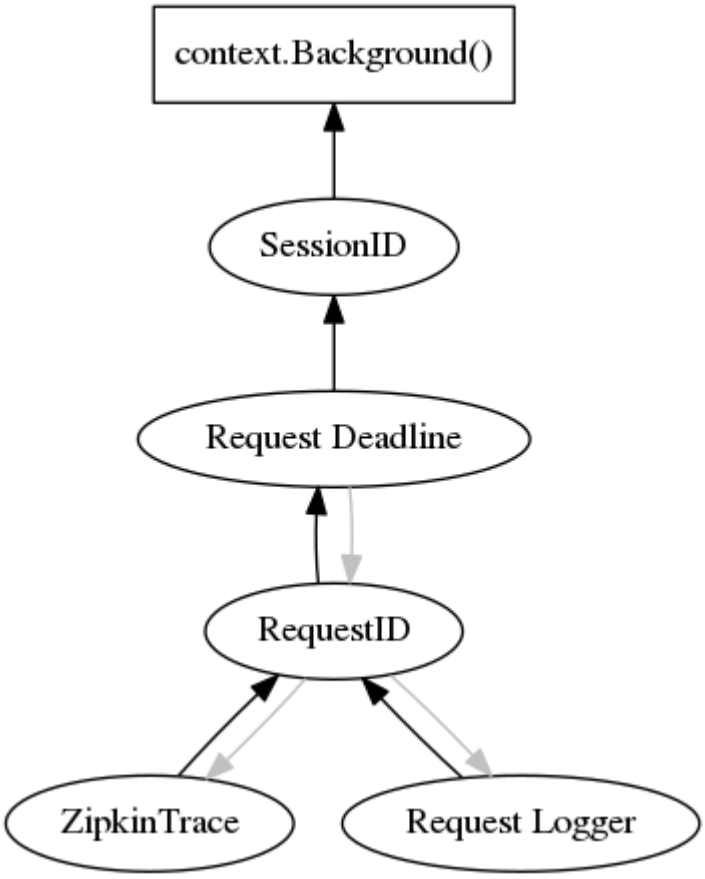
Examples

Albero del contesto rappresentato come grafico diretto

Un semplice albero di contesto (contenente alcuni valori comuni che potrebbero essere richiesti e inclusi in un contesto) creati dal codice Go come segue:

```
// 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())
```

È un albero che può essere rappresentato come un grafico diretto che assomiglia a questo:



Ogni contesto figlio ha accesso ai valori dei suoi contesti genitore, quindi l'accesso ai dati scorre verso l'alto nell'albero (rappresentato dai bordi neri). D'altra parte, i segnali di annullamento viaggiano lungo l'albero (se un contesto viene cancellato, tutti i suoi figli vengono cancellati). Il flusso del segnale di annullamento è rappresentato dai bordi grigi.

Utilizzare un contesto per annullare il lavoro

È possibile utilizzare il passaggio di un contesto con un timeout (o una funzione di annullamento) a una funzione di esecuzione prolungata per annullare il funzionamento di tali funzioni:

```
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!
    }
}
```

Leggi Contesto online: https://riptutorial.com/it/go/topic/2743/contesto

Capitolo 19: costanti

Osservazioni

Go supporta le costanti di carattere, stringa, valori booleani e numerici.

Examples

Dichiarare una costante

Le costanti sono dichiarate come variabili, ma usando la parola chiave `const` :

```
const Greeting string = "Hello World"
const Years int = 10
const Truth bool = true
```

Come per le variabili, i nomi che iniziano con una lettera maiuscola vengono esportati (*pubblici*), i nomi che iniziano con lettere minuscole non lo sono.

```
// not exported
const alpha string = "Alpha"
// exported
const Beta string = "Beta"
```

Le costanti possono essere utilizzate come qualsiasi altra variabile, ad eccezione del fatto che il valore non può essere modificato. Ecco un esempio:

```
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
```

```
}
```

Terreno di gioco

Dichiarazione di costanti multiple

È possibile dichiarare più costanti all'interno dello stesso blocco `const` :

```
const (
    Alpha = "alpha"
    Beta  = "beta"
    Gamma = "gamma"
)
```

E incrementa automaticamente le costanti con la parola chiave `iota` :

```
const (
    Zero = iota // Zero == 0
    One         // One  == 1
    Two         // Two  == 2
)
```

Per ulteriori esempi sull'utilizzo di `iota` per dichiarare le costanti, vedere Iota .

Puoi anche dichiarare più costanti usando il compito multiplo. Tuttavia, questa sintassi potrebbe essere più difficile da leggere ed è generalmente evitata.

```
const Foo, Bar = "foo", "bar"
```

Costanti tipizzate e non tipizzate

Le costanti in Go possono essere digitate o non tipizzate. Ad esempio, data la seguente stringa letterale:

```
"bar"
```

si potrebbe dire che il tipo di letterale è una `string` , tuttavia, questo non è semanticamente corretto. Invece, i letterali sono costanti di *stringa non tipizzate* . È una stringa (più correttamente, il suo *tipo predefinito* è `string`), ma non è un **valore** Go e quindi non ha alcun tipo fino a quando non viene assegnato o utilizzato in un contesto che viene digitato. Questa è una sottile distinzione, ma utile per capire.

Allo stesso modo, se assegniamo il letterale a una costante:

```
const foo = "bar"
```

Rimane non tipizzato poiché, per impostazione predefinita, le costanti non sono tipizzate. È possibile dichiararlo anche come *costante stringa digitato* :

```
const typedFoo string = "bar"
```

La differenza entra in gioco quando tentiamo di assegnare queste costanti in un contesto che ha tipo. Ad esempio, considera quanto segue:

```
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
```

Leggi costanti online: https://riptutorial.com/it/go/topic/1047/costanti

Capitolo 20: Costruisci vincoli

Sintassi

- // + crea tag

Osservazioni

I tag di costruzione vengono utilizzati per creare condizionatamente determinati file nel codice. I tag di compilazione possono ignorare i file che non si desidera creare a meno che non siano esplicitamente inclusi, oppure possono essere utilizzati alcuni tag predefiniti per costruire un file solo su una particolare architettura o sistema operativo.

I tag di compilazione possono apparire in qualsiasi tipo di file sorgente (non solo Go), ma devono apparire nella parte superiore del file, preceduti solo da righe vuote e altri commenti di riga. Queste regole significano che nei file Go deve apparire un vincolo di build prima della clausola del pacchetto.

Una serie di tag build deve essere seguita da una riga vuota.

Examples

Test di integrazione separati

I vincoli di costruzione sono comunemente usati per separare i normali test unitari dai test di integrazione che richiedono risorse esterne, come un database o un accesso di rete. Per fare ciò, aggiungi un vincolo di build personalizzato nella parte superiore del file di test:

```
// +build integration

package main

import (
    "testing"
)

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

Il file di test non verrà compilato nell'eseguibile di build a meno che non venga utilizzata la seguente chiamata di `go test` :

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

risultati:

```
$ 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
```

Ottimizza le implementazioni basate sull'architettura

Possiamo ottimizzare una semplice funzione xor solo per le architetture che supportano le letture / scritture non allineate creando due file che definiscono la funzione e prefiggendoli con un vincolo di build (per un esempio reale del codice xor che è fuori ambito qui, vedi `crypto/cipher/xor.go` nella libreria standard):

```
// +build 386 amd64 s390x

package cipher

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

e per altre architetture:

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

package cipher

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

Leggi Costruisci vincoli online: https://riptutorial.com/it/go/topic/2595/costruisci-vincoli

Capitolo 21: Crittografia

introduzione

Scopri come crittografare e decrittografare i dati con Go. Tieni presente che questo non è un corso sulla crittografia, ma piuttosto come ottenerlo con Go.

Examples

Crittografia e decrittografia

Prefazione

Questo è un esempio dettagliato su come crittografare e decodificare i dati con Go. Il codice degli usi è abbreviato, ad esempio la gestione degli errori non è menzionata. Il progetto completo di lavoro con gestione degli errori e interfaccia utente può essere trovato su Github qui .

---

crittografia

Introduzione e dati

Questo esempio descrive una crittografia e una decrittazione funzionanti complete in Go. Per fare ciò, abbiamo bisogno di un dato. In questo esempio, usiamo la nostra propria struttura dati `secret` :

```
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
}
```

Successivamente, vogliamo crittografare un tale `secret` . L'esempio funzionante completo può essere trovato qui (link a Github) . Ora, il processo passo-passo:

Passo 1

Prima di tutto, abbiamo bisogno di una specie di password principale per proteggere il segreto:
`masterPassword := "PASS"`

Passo 2

Tutti i metodi di crittografia che funzionano con byte invece di stringhe. Quindi, costruiamo un array di byte con i dati del nostro segreto.

```
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",
    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,
))
```

Passaggio 3

Creiamo un po 'di sale per prevenire gli attacchi dei tavoli arcobaleno, cf. Wikipedia : `saltBytes := uuid.NewV4().Bytes()` . Qui, usiamo un UUID v4 che non è prevedibile.

Passaggio 4

Ora, siamo in grado di ricavare una chiave e un vettore dalla password principale e dal sale casuale, per quanto riguarda 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:]
```

Passaggio 5

La modalità CBC desiderata funziona con blocchi interi. Pertanto, dobbiamo verificare se i nostri dati sono allineati a un blocco completo. In caso contrario, dobbiamo riempirlo:

```
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
}
```

Passaggio 6

Ora creiamo un codice AES: `aesBlockEncrypter, aesErr := aes.NewCipher(keyBytes)`

Passaggio 7

Ci riserviamo la memoria necessaria per i dati crittografati: `encryptedData := make([]byte, len(secretBytesDecrypted))` . Nel caso di AES-CBC, i dati crittografati avevano la stessa lunghezza dei dati non crittografati.

Passaggio 8

Ora, dovremmo creare il codificatore e crittografare i dati:

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

Ora, i dati crittografati si trovano all'interno della variabile `encryptedData` .

Passaggio 9

I dati crittografati devono essere memorizzati. Ma non solo i dati: senza il sale, i dati crittografati non potevano essere decifrati. Quindi, dobbiamo usare un qualche tipo di formato di file per gestirlo. Qui, codifichiamo i dati crittografati come base64, cf. Wikipedia :

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

Successivamente, definiamo il nostro contenuto di file e il nostro formato di file. Il formato ha questo aspetto: `salt[0x10]base64 content` . Per prima cosa, immagazziniamo il sale. Per marcare l'inizio del contenuto di base64, memorizziamo il byte `10` . Funziona, perché base64 non usa

questo valore. Pertanto, potremmo trovare l'inizio di base64 cercando la prima occorrenza di `10` dalla fine all'inizio del file.

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

Passaggio 10

Infine, potremmo scrivere il nostro file: `writeErr := ioutil.WriteFile("my secret.data", fileContent, 0644)` .

---

decrittazione

Introduzione e dati

Per quanto riguarda la crittografia, abbiamo bisogno di alcuni dati con cui lavorare. Pertanto, assumiamo di avere un file crittografato e la struttura citata `secret` . L'obiettivo è leggere i dati crittografati dal file, decodificarlo e creare un'istanza della struttura.

Passo 1

Il primo passo è identico alla crittografia: è necessario un tipo di password principale per decrittografare il segreto: `masterPassword := "PASS"` .

Passo 2

Ora, leggiamo i dati crittografati dal file: `encryptedFileData, bytesErr := ioutil.ReadFile(filename)` .

Passaggio 3

Come accennato in precedenza, potremmo dividere il sale e i dati crittografati dal byte delimitatore `10` , ricercato all'indietro dalla fine all'inizio:

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

Passaggio 4

Successivamente, dobbiamo decodificare i byte codificati base64:

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

Passaggio 5

Ora, siamo in grado di ricavare una chiave e un vettore dalla password principale e dal sale casuale, per quanto riguarda 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:]
```

Passaggio 6

Creare un codice AES: `aesBlockDecrypter, aesErr := aes.NewCipher(keyBytes)` .

Passaggio 7

Riservare la memoria necessaria per i dati decrittografati: `decryptedData := make([]byte, len(encryptedBytes))` . Per definizione, ha la stessa lunghezza dei dati crittografati.

Passaggio 8

Ora crea il decodificatore e decrittografa i dati:

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

Passaggio 9

Converti i byte letti nella stringa: `decryptedString := string(decryptedData)` . Perché abbiamo bisogno di linee, dividere la stringa: `lines := strings.Split(decryptedString, "\n")` .

Passaggio 10

Costruisci un `secret` fuori dalle righe:

```
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]
```

Infine, ricrea le interruzioni di riga all'interno del campo delle note: `artifact.Notes = strings.Replace(artifact.Notes, string(65000), "\n", -1)` .

Leggi Crittografia online: https://riptutorial.com/it/go/topic/10065/crittografia

Capitolo 22: Differire

introduzione

`defer` spinge una chiamata di funzione in una lista. L'elenco delle chiamate salvate viene eseguito dopo il ritorno della funzione circostante. Defer è comunemente usato per semplificare le funzioni che eseguono varie azioni di pulizia.

Sintassi

- differire someFunc (args)
- defer func () {// code goes here} ()

Osservazioni

Il differimento funziona iniettando un nuovo stack frame (la funzione chiamata dopo la parola chiave `defer`) nello stack di chiamate sotto la funzione attualmente in esecuzione. Ciò significa che il differimento è garantito per l'esecuzione fino a quando lo stack sarà svolto (se il tuo programma si blocca o ottiene un `SIGKILL` , il rinvio non verrà eseguito).

Examples

Defer Nozioni di base

Una *dichiarazione di differimento* in Go è semplicemente una chiamata di funzione contrassegnata per essere eseguita in un secondo momento. L'istruzione di `defer` è una chiamata di funzione ordinaria preceduta dal `defer` della parola chiave.

```
defer someFunction()
```

Una funzione differita viene eseguito una volta la funzione che contiene le `defer` restituisce istruzione. La chiamata effettiva alla funzione differita si verifica quando la funzione di chiusura:

- esegue una dichiarazione di ritorno
- cade alla fine
- panico

Esempio:

```
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)
}
```

Produzione:

```
First main statement
Last main statement
Function main returned
```

Se una funzione ha più istruzioni posticipate, formano una pila. L'ultimo `defer` è il primo da eseguire dopo il ritorno della funzione di inclusione, seguito dalle successive chiamate al precedente `defer` in ordine (sotto l'esempio restituisce provocando un panico):

```
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)
}
```

Produzione:

```
First main statement
Num 3
Num 2
Num 1
panic: panic occurred

goroutine 1 [running]:
....
```

Si noti che le funzioni posticipate hanno i loro argomenti valutati al momento `defer` :

```
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)
}
```

Produzione:

```
First main statement
Num 4
Num 2
Num 1
```

Se una funzione ha denominato valori di ritorno, una funzione anonima differita all'interno di quella funzione può accedere e aggiornare il valore restituito anche dopo che la funzione è ritornata:

```
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
}
```

Infine, una dichiarazione di `defer` viene generalmente utilizzata operazioni che spesso si verificano insieme. Per esempio:

- aprire e chiudere un file
- connettersi e disconnettersi
- bloccare e sbloccare un mutex
- segna un waitgroup come fatto (`defer wg.Done()`)

Questo utilizzo garantisce il corretto rilascio delle risorse di sistema indipendentemente dal flusso di esecuzione.

```
resp, err := http.Get(url)
if err != nil {
return err
}
defer resp.Body.Close() // Body will always get closed
```

Chiamate a funzioni differite

Le chiamate a funzioni differite hanno uno scopo simile a cose come blocchi `finally` in linguaggi come Java: assicurano che alcune funzioni vengano eseguite quando la funzione esterna ritorna, indipendentemente dal fatto che si sia verificato un errore o che sia stata rilasciata una dichiarazione di ritorno in caso di più resi. Questo è utile per ripulire le risorse che devono essere chiuse come connessioni di rete o puntatori di file. La parola chiave `defer` indica una chiamata a una funzione differita, analogamente alla parola chiave `go` avvia una nuova goroutine. Come una chiamata `go` , gli argomenti delle funzioni vengono valutati immediatamente, ma a differenza di una chiamata `go` , le funzioni posticipate non vengono eseguite contemporaneamente.

```
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
```

Nota l'uso di `conn.Close()` invece di `conn.Close` : non stai passando solo una funzione, stai rimandando una *chiamata a* una funzione completa, inclusi i suoi argomenti. Le chiamate a funzioni multiple possono essere rinviate nella stessa funzione esterna e ciascuna verrà eseguita una volta nell'ordine inverso. Puoi anche differire le chiusure - non dimenticare i paren!

```
defer func(){
    // Do some cleanup
}()
```

Leggi Differire online: https://riptutorial.com/it/go/topic/2795/differire

Capitolo 23: Digita le conversioni

Examples

Conversione di base

Ci sono due stili di base per la conversione del tipo in 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.
```

Test dell'interfaccia di implementazione

Poiché Go utilizza l'implementazione implicita dell'interfaccia, non si otterrà un errore in fase di compilazione se la propria struttura non implementa un'interfaccia che si intendeva implementare. Puoi testare l'implementazione in modo esplicito usando type casting: digita l'interfaccia MyInterface {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
```

Implementare un sistema di unità con tipi

Questo esempio illustra come il sistema di tipo di Go può essere utilizzato per implementare un sistema di unità.

```
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)
}
```

Apri nel parco giochi

Leggi Digita le conversioni online: https://riptutorial.com/it/go/topic/2851/digita-le-conversioni

Capitolo 24: Esecuzione di comandi

Examples

Time out con Interrupt e poi Kill

```
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")
}
```

Esecuzione semplice di comandi

```
// 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)
}
```

Eseguendo un comando quindi continua e aspetta

```
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)
```

Esecuzione di un comando due volte

> Un Cmd non può essere riutilizzato dopo aver chiamato i suoi metodi Run, Output o CombinedOutput

L'esecuzione di un comando due volte ***non*** *funzionerà* :

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

> Errore: exec: già avviato

Piuttosto, si deve usare **due** `exec.Command` **separati** . Potrebbe anche essere necessario un certo ritardo tra i comandi.

```
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
```

Leggi Esecuzione di comandi online: https://riptutorial.com/it/go/topic/1097/esecuzione-di-comandi

Capitolo 25: Espansione in linea

Osservazioni

L'espansione in linea è un'ottimizzazione comune nel codice compilato che ha privilegiato le prestazioni rispetto alle dimensioni binarie. Permette al compilatore di sostituire una chiamata di funzione con il corpo reale della funzione; copia / incolla efficacemente il codice da un posto a un altro in fase di compilazione. Poiché il sito di chiamata è espanso per contenere solo le istruzioni della macchina che il compilatore ha generato per la funzione, non è necessario eseguire CALL o PUSH (l'equivalente x86 di un'istruzione GOTO o uno stack frame push) o il loro equivalente su altro architetture.

L'inliner prende le decisioni sull'opportunità o meno di incorporare una funzione in base a un numero di euristiche, ma in generale Vai in linea per impostazione predefinita. Dato che l'inliner si sbarazza delle chiamate di funzione, in effetti riesce a decidere dove è consentito al programmatore di anticipare una goroutine.

Le chiamate di funzione non saranno in linea se una delle seguenti condizioni è vera (ci sono anche molte altre ragioni, questa lista è incompleta):

- Le funzioni sono variadiche (ad esempio hanno `...` argomenti)
- Le funzioni hanno una "massima pelosità" superiore al budget (si recitano troppo o non possono essere analizzate per altri motivi)
- Contengono il `panic` , `recover` o `defer`

Examples

Disabilitare l'espansione in linea

L'espansione in linea può essere disabilitata con il `go:noinline` pragma. Ad esempio, se costruiamo il seguente programma semplice:

```
package main

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

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

otteniamo un output simile a questo (tagliato per la leggibilità):

```
$ 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
…
```

si noti che non vi è alcuna `CALL` al `printhello` . Tuttavia, se poi costruiamo il programma con il pragma in atto:

```
package main

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

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

L'output contiene la funzione printhello e 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
…
```

Leggi Espansione in linea online: https://riptutorial.com/it/go/topic/2718/espansione-in-linea

Capitolo 26: fette

introduzione

Una slice è una struttura di dati che incapsula una matrice in modo che il programmatore possa aggiungere tutti gli elementi necessari senza doversi preoccupare della gestione della memoria. Le fette possono essere tagliate in sottosquadri in modo molto efficiente, poiché le sezioni risultanti puntano tutte allo stesso array interno. I programmatori spesso ne approfittano per evitare la copia di array, che in genere vengono eseguiti in molti altri linguaggi di programmazione.

Sintassi

- slice: = make ([] type, len, cap) // crea una nuova slice
- slice = append (slice, item) // aggiunge un elemento ad una slice
- slice = append (slice, items ...) // aggiunge una porzione di elementi a una slice
- len: = len (slice) // ottiene la lunghezza di una fetta
- cap: = cap (slice) // ottiene la capacità di una slice
- elNum: = copy (dst, slice) // copia un contenuto di una slice in un'altra slice

Examples

Aggiungendo alla fetta

```
slice = append(slice, "hello", "world")
```

Aggiunta di due fette insieme

```
slice1 := []string{"!"}
slice2 := []string{"Hello", "world"}
slice  := append(slice1, slice2...)
```

Esegui nel Go Playground

Rimozione elementi / fette "Affettare"

Se è necessario rimuovere uno o più elementi da una sezione o se è necessario lavorare con una sottosezione di un'altra esistente; puoi usare il seguente metodo.

> Gli esempi seguenti utilizzano slice of int, ma questo funziona con tutti i tipi di slice.

Quindi, per quello, abbiamo bisogno di una fetta, da parte nostra rimuoveremo alcuni elementi:

```
slice := []int{1, 2, 3, 4, 5, 6}
// > [1 2 3 4 5 6]
```

Abbiamo bisogno anche degli indici di elementi da rimuovere:

```
// 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
```

E così possiamo "tagliare" la fetta, rimuovendo elementi indesiderati:

```
// 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]
```

Per rimuovere solo un elemento, devi solo mettere l'indice di questo elemento come primo E come ultimo indice da rimuovere, proprio così:

```
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]
```

E puoi anche rimuovere elementi dall'inizio della sezione:

```
newSlice6 := append(slice[:0], slice[last+1:]...)
// > [6]

// That can be simplified into
newSlice6 := slice[last+1:]
// > [6]
```

Puoi anche rimuovere alcuni elementi dalla fine della sezione:

```
newSlice7 := append(slice[:first], slice[first+1:len(slice)-1]...)
// > [1 2]

// That can be simplified into
newSlice7 := slice[:first]
```

```
// > [1 2]
```

Se la nuova slice deve contenere esattamente gli stessi elementi della prima, puoi usare la stessa cosa ma con l' `last := first-1` .
(Questo può essere utile nel caso in cui i tuoi indici siano precedentemente calcolati)

Lunghezza e capacità

Le fette hanno sia lunghezza che capacità. La lunghezza di una sezione è il numero di elementi *attualmente* nella sezione, mentre la capacità è il numero di elementi che la sezione *può contenere* prima di dover essere riallocata.

Quando si crea una sezione usando la funzione built-in `make()` , è possibile specificare la sua lunghezza e facoltativamente la sua capacità. Se la capacità non è specificata esplicitamente, sarà la lunghezza specificata.

```
var s = make([]int, 3, 5) // length 3, capacity 5
```

Puoi controllare la lunghezza di una sezione con la funzione `len()` integrata:

```
var n = len(s) // n == 3
```

Puoi verificare la capacità con la funzione `cap()` integrata:

```
var c = cap(s) // c == 5
```

Gli elementi creati da `make()` sono impostati sul valore zero per il tipo di elemento della slice:

```
for idx, val := range s {
    fmt.Println(idx, val)
}
// output:
// 0 0
// 1 0
// 2 0
```

Eseguilo su play.golang.org

Non è possibile accedere agli elementi oltre la lunghezza di una sezione, anche se l'indice è all'interno della capacità:

```
var x = s[3] // panic: runtime error: index out of range
```

Tuttavia, finché la capacità supera la lunghezza, è possibile aggiungere nuovi elementi senza riallocare:

```
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
```

Se si aggiunge una porzione che non ha la capacità di accettare i nuovi elementi, l'array sottostante verrà riallocato per voi con una capacità sufficiente:

```
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
```

Pertanto, è generalmente buona norma allocare una capacità sufficiente quando si crea una sezione, se si conosce lo spazio necessario per evitare riallocazioni non necessarie.

Copia dei contenuti da una sezione all'altra

Se si desidera copiare il contenuto di una sezione in una sezione inizialmente vuota, è possibile eseguire i seguenti passaggi per realizzarlo-

1. Crea la slice sorgente:

```
var sourceSlice []interface{} = []interface{}{"Hello",5.10,"World",true}
```

2. Crea la sezione di destinazione, con:

- Lunghezza = Lunghezza di sourceSlice

```
var destinationSlice []interface{} = make([]interface{},len(sourceSlice))
```

3. Ora che l'array sottostante della slice di destinazione è abbastanza grande da contenere tutti gli elementi della slice source, possiamo procedere a copiare gli elementi usando la `copy` integrata:

```
copy(destinationSlice,sourceSlice)
```

Creazione di fette

Le fette sono il modo tipico in cui i programmatori memorizzano gli elenchi di dati.

Per dichiarare una variabile slice usa `[]Type` sintassi.

```
var a []int
```

Per dichiarare e inizializzare una variabile slice in una riga, utilizzare la sintassi `[]Type{values}` .

```
var a []int = []int{3, 1, 4, 1, 5, 9}
```

Un altro modo per inizializzare una sezione è con la funzione `make` . Tre argomenti: il `Type` di sezione (o mappa), la `length` e la `capacity` .

```
a := make([]int, 0, 5)
```

Puoi aggiungere elementi alla tua nuova sezione usando `append` .

```
a = append(a, 5)
```

Controlla il numero di elementi nella tua fetta usando `len` .

```
length := len(a)
```

Controlla la capacità della tua fetta usando `cap` . La capacità è il numero di elementi attualmente allocati in memoria per la sezione. È sempre possibile aggiungere una porzione alla capacità poiché Go creerà automaticamente una fetta più grande per te.

```
capacity := cap(a)
```

È possibile accedere agli elementi in una sezione utilizzando la tipica sintassi di indicizzazione.

```
a[0]  // Gets the first member of `a`
```

Puoi anche usare un ciclo `for` su loop con `range` . La prima variabile è l'indice nell'array specificato e la seconda variabile è il valore per l'indice.

```
for index, value := range a {
    fmt.Println("Index: " + index + " Value: " + value)  // Prints "Index: 0 Value: 5" (and
continues until end of slice)
}
```

Vai al parco giochi

Filtrare una fetta

Per filtrare una porzione senza allocare una nuova matrice sottostante:

```
// 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]
```

Valore zero della fetta

Il valore zero della slice è `nil` , che ha lunghezza e capacità `0` . Una slice `nil` non ha array sottostanti. Ma ci sono anche fette di lunghezza non nulla e capacità `0` , come `[]int{}` o `make([]int, 5)[5:]` .

Qualsiasi tipo che ha valori nil può essere convertito in `nil` slice:

```
s = []int(nil)
```

Per verificare se una sezione è vuota, utilizzare:

```
if len(s) == 0 {
    fmt.Ptintf("s is empty.")
}
```

Leggi fette online: https://riptutorial.com/it/go/topic/733/fette

Capitolo 27: File I / O

Sintassi

- file, err: = os.Open (*nome*) // Apre un file in modalità di sola lettura. Un errore non nullo viene restituito se il file non può essere aperto.
- file, err: = os.Create (*nome*) // Crea o apre un file se già esiste in modalità di sola scrittura. Il file viene sovrascritto se esiste già. Un errore non nullo viene restituito se il file non può essere aperto.
- file, err: = os.OpenFile (*nome* , *flag* , *perm*) // Apre un file nella modalità specificata dai flag. Un errore non nullo viene restituito se il file non può essere aperto.
- data, err: = ioutil.ReadFile (*nome*) // Legge l'intero file e lo restituisce. Viene restituito un errore non nullo se non è possibile leggere l'intero file.
- err: = ioutil.WriteFile (*nome* , *data* , *perm*) // Crea o sovrascrive un file con i dati forniti ei bit di autorizzazione UNIX. Un errore non nullo viene restituito se il file non è stato scritto.
- err: = os.Remove (*nome*) // Elimina un file. Un errore non nullo viene restituito se il file non può essere cancellato.
- err: = os.RemoveAll (*nome*) // Elimina un file o una gerarchia di directory intera. Viene restituito un errore non nullo se il file o la directory non possono essere cancellati.
- err: = os.Rename (*oldName* , *newName*) // Rinomina o sposta un file (può trovarsi tra le directory). Un errore non nullo viene restituito se il file non può essere spostato.

Parametri

Parametro	Dettagli
nome	Un nome file o percorso di tipo stringa. Ad esempio: `"hello.txt"` .
sbagliare	Un `error` Se non è `nil` , rappresenta un errore che si è verificato quando è stata chiamata la funzione.
file	Un gestore di file di tipo `*os.File` restituito dalle funzioni relative al file del pacchetto `os` . Implementa un `io.ReadWriter` , ovvero è possibile chiamare `Read(data)` e `Write(data)` su di esso. Si noti che queste funzioni potrebbero non essere possibile chiamare in base ai flag aperti del file.
dati	Una porzione di byte (`[]byte`) che rappresenta i dati grezzi di un file.
permanente	I bit di autorizzazione UNIX utilizzati per aprire un file con tipo `os.FileMode` . Sono disponibili diverse costanti per l'utilizzo dei bit di autorizzazione.
bandiera	File flag aperti che determinano i metodi che possono essere chiamati sul gestore di file di tipo `int` . Sono disponibili diverse costanti per aiutare con l'uso delle bandiere. Sono: `os.O_RDONLY` , `os.O_WRONLY` , `os.O_RDWR` , `os.O_APPEND` , `os.O_CREATE` , `os.O_EXCL` , `os.O_SYNC` e `os.O_TRUNC` .

Examples

Leggere e scrivere su un file usando ioutil

Un semplice programma che scrive "Ciao, mondo!" per `test.txt` , legge i dati e li stampa. Dimostra semplici operazioni di I / O su file.

```
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))
}
```

Elenco di tutti i file e le cartelle nella directory corrente

```
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())
    }
}
```

Elenco di tutte le cartelle nella directory corrente

```
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())
        }
    }
}
```

Leggi File I / O online: https://riptutorial.com/it/go/topic/1033/file-i---o

Capitolo 28: Fmt

Examples

Stringer

L'interfaccia `fmt.Stringer` richiede un singolo metodo, `String() string` per essere soddisfatti. Il metodo stringa definisce il formato stringa "nativo" per quel valore, ed è la rappresentazione predefinita se il valore è fornito a una delle routine di formattazione o di stampa dei pacchetti `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 di base

Il pacchetto fmt implementa l'I / O formattato usando i *verbi di* formato:

```
%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
```

Funzioni di formattazione

Ci sono **4** tipi di funzioni principali in `fmt` e diverse varianti all'interno.

Stampare

```
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` possibile utilizzare `Fprint` , all'interno dei gestori `http` :

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

Scansione

Scansione esegue la scansione del testo letto dallo standard input.

```
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
```

Interfaccia Stringer

Qualsiasi valore che abbia un metodo `String()` implementa lo `Stringer` **inteface** `fmt`

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

Leggi Fmt online: https://riptutorial.com/it/go/topic/2938/fmt

Capitolo 29: funzioni

introduzione

Le funzioni in Go forniscono un codice organizzato e riutilizzabile per eseguire una serie di azioni. Le funzioni semplificano il processo di codifica, impediscono la logica ridondante e rendono il codice più facile da seguire. Questo argomento descrive la dichiarazione e l'utilizzo di funzioni, argomenti, parametri, dichiarazioni di ritorno e ambiti in Go.

Sintassi

- func () // tipo di funzione senza argomenti e senza valore di ritorno
- func (x int) int // accetta intero e restituisce un intero
- func (a, b int, z float32) bool // accetta 2 interi, uno mobile e restituisce un valore booleano
- func (prefisso stringa, valori ... int) // funzione "variadic" che accetta una stringa e uno o più numeri di interi
- func () (int, bool) // funzione che restituisce due valori
- func (a, b int, z float64, opt ... interface {}) (successo bool) // accetta 2 interi, un float e uno o più numeri di interfacce e restituisce il valore booleano denominato (che è già inizializzato all'interno della funzione)

Examples

Dichiarazione di base

Una semplice funzione che non accetta parametri e non restituisce alcun valore:

```
func SayHello() {
    fmt.Println("Hello!")
}
```

parametri

Una funzione può facoltativamente dichiarare una serie di parametri:

```
func SayHelloToMe(firstName, lastName string, age int) {
    fmt.Printf("Hello, %s %s!\n", firstName, lastName)
    fmt.Printf("You are %d", age)
}
```

Si noti che il tipo per `firstName` è omesso perché è identico a `lastName` .

Valori di ritorno

Una funzione può restituire uno o più valori al chiamante:

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

Il secondo valore di ritorno può anche essere l'errore 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
}
```

Devono essere annotate due cose importanti:

- La parentesi può essere omessa per un singolo valore di ritorno.
- Ogni `return` istruzione deve fornire un valore per **tutti i** valori di ritorno dichiarati.

Valori di ritorno nominati

I valori di ritorno possono essere assegnati a una variabile locale. Un vuoto `return` dichiarazione può quindi essere utilizzata per restituire i loro valori correnti. Questo è noto come ritorno *"nudo"* . Le dichiarazioni di reso nude devono essere utilizzate solo in brevi funzioni poiché danneggiano la leggibilità in funzioni più lunghe:

```
func Inverse(v float32) (reciprocal float32) {
    if v == 0 {
        return
    }
    reciprocal = 1 / v
    return
}
```

giocarci sul campo da gioco

```
//A function can also return multiple values
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

giocarci sul campo da gioco

Devono essere annotate due cose importanti:

- La parentesi attorno ai valori di ritorno è **obbligatoria** .
- Un vuoto `return` deve essere sempre fornita dichiarazione.

Funzioni e chiusure letterali

Una semplice funzione letterale, stampa `Hello!` allo stdout:

```
package main

import "fmt"

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

giocarci sul campo da gioco

Una funzione letterale, che stampa il `str` su stdout:

```
package main

import "fmt"

func main() {
    func(str string) {
        fmt.Println(str)
    }("Hello!")
}
```

giocarci sul campo da gioco

Una funzione letterale, che si chiude sulla variabile `str` :

```
package main

import "fmt"

func main() {
    str := "Hello!"
    func() {
        fmt.Println(str)
    }()
}
```

giocarci sul campo da gioco

È possibile assegnare una funzione letterale a una variabile:

```
package main

import (
    "fmt"
)

func main() {
```

```
    str := "Hello!"
    anon := func() {
        fmt.Println(str)
    }
    anon()
}
```

giocarci sul campo da gioco

Funzioni variabili

Una funzione variadica può essere chiamata con qualsiasi numero di argomenti **finali** . Questi elementi sono memorizzati in una sezione.

```
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")
}
```

giocarci sul campo da gioco

Puoi anche assegnare una sezione a una funzione variadica, con `...` :

```
func main() {
     strs := []string {"Str1", "Str2", "Str3"}

     variadic(strs...)
}
```

giocarci sul campo da gioco

Leggi funzioni online: https://riptutorial.com/it/go/topic/373/funzioni

Capitolo 30: Gestione degli errori

introduzione

In Go, le situazioni inaspettate vengono gestite utilizzando **errori** , non eccezioni. Questo approccio è più simile a quello di C, usando errno, rispetto a quello di Java o di altri linguaggi orientati agli oggetti, con i loro blocchi try / catch. Tuttavia, un errore non è un numero intero ma un'interfaccia.

Una funzione che potrebbe non riuscire in genere restituisce un **errore** come ultimo valore restituito. Se questo errore non è **nulla** , qualcosa è andato storto e il chiamante della funzione dovrebbe agire di conseguenza.

Osservazioni

Nota come in Go non si *genera* un errore. Invece, si *restituisce* un errore in caso di errore.

Se una funzione non riesce, l'ultimo valore restituito è generalmente un tipo di `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

Creazione di un valore di errore

Il modo più semplice per creare un errore è utilizzare il pacchetto `errors` .

```
errors.New("this is an error")
```

Se si desidera aggiungere ulteriori informazioni a un errore, il pacchetto `fmt` fornisce anche un utile metodo di creazione degli errori:

```
var f float64
fmt.Errorf("error with some additional information: %g", f)
```

Ecco un esempio completo in cui viene restituito l'errore da una funzione:

```
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")
}
```

Apri nel parco giochi

Creazione di un tipo di errore personalizzato

In Go, un errore è rappresentato da qualsiasi valore che può descriversi come stringa. Qualsiasi tipo che implementa l'interfaccia di `error` incorporata è un errore.

```
// The error interface is represented by a single
// Error() method, that returns a string representation of the error
type error interface {
    Error() string
}
```

L'esempio seguente mostra come definire un nuovo tipo di errore utilizzando una stringa composita letterale.

```
// 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)
}
```

Ora posso usare il mio tipo di errore personalizzato come errore:

```
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
}
```

Restituzione di un errore

In Go non si *genera* un errore. Invece, si *restituisce* un `error` in caso di errore.

```
// 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
}
```

Se il metodo restituisce più valori (e l'esecuzione può non riuscire), la convenzione standard restituisce l'errore come ultimo argomento.

```
// 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)
}
```

Gestione di un errore

Gli errori In Go possono essere restituiti da una chiamata di funzione. La convenzione è che se un metodo può fallire, l'ultimo argomento restituito è un `error` .

```
func DoAndReturnSomething() (string, error) {
    if os.Getenv("ERROR") == "1" {
        return "", errors.New("The method failed")
    }

    // The method succeeded.
    return "Success!", nil
}
```

Si utilizzano più assegnazioni di variabili per verificare se il metodo non è riuscito.

```
result, err := DoAndReturnSomething()
if err != nil {
    panic(err)
}

// This is executed only if the method didn't return an error
fmt.Println(result)
```

Se non ti interessa l'errore, puoi semplicemente ignorarlo assegnandolo a _ .

```
result, _ := DoAndReturnSomething()
fmt.Println(result)
```

Ovviamente, ignorare un errore può avere implicazioni serie. Pertanto, questo in genere non è raccomandato.

Se si dispone di più chiamate al metodo e uno o più metodi nella catena possono restituire un errore, è necessario propagare l'errore al primo livello in grado di gestirlo.

```
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
}
```

Recupero dal panico

Un errore comune è dichiarare una sezione e iniziare a richiedere gli indici da essa senza inizializzarla, il che porta a un panico "indice fuori intervallo". Il seguente codice spiega come recuperare dal panico senza uscire dal programma, che è il comportamento normale per un panico. Nella maggior parte dei casi, restituire un errore in questo modo piuttosto che uscire dal programma in preda al panico è utile solo per scopi di sviluppo o test.

```
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'uso di una funzione separata (piuttosto che la chiusura) consente il riutilizzo della stessa funzione in altre funzioni inclini al panico.

Leggi Gestione degli errori online: https://riptutorial.com/it/go/topic/785/gestione-degli-errori

Capitolo 31: Goroutines

introduzione

Una goroutine è un thread leggero gestito dal runtime Go.

vai f (x, y, z)

avvia una nuova goroutine in esecuzione

f (x, y, z)

La valutazione di f, x, yez avviene nella goroutine corrente e l'esecuzione di f avviene nella nuova goroutine.

Le goroutine vengono eseguite nello stesso spazio degli indirizzi, pertanto l'accesso alla memoria condivisa deve essere sincronizzato. Il pacchetto di sincronizzazione fornisce primitive utili, anche se non ne avrete bisogno molto in Go in quanto vi sono altre primitive.

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

Examples

Goroutines Programma base

```
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")
}
```

Una goroutine è una funzione che è in grado di funzionare in concomitanza con altre funzioni. Per creare una goroutine usiamo la parola chiave `go` seguita da una funzione invocazione:

```
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)
}
```

Generalmente, la chiamata di funzione esegue tutte le istruzioni all'interno del corpo della funzione e torna alla riga successiva. Ma con le goroutine torniamo immediatamente alla riga successiva in quanto non aspetta che la funzione si completi. Quindi, è stata inclusa una chiamata a una funzione `Scanln` , altrimenti il programma è stato chiuso senza stampare i numeri.

Leggi Goroutines online: https://riptutorial.com/it/go/topic/9776/goroutines

Capitolo 32: Il comando Go

introduzione

Il comando `go` è un programma a riga di comando che consente la gestione dello sviluppo di Go. Consente di costruire, eseguire e testare il codice, oltre a una serie di altre attività relative a Go.

Examples

Vai a correre

`go run` eseguirà un programma senza creare un file eseguibile. Principalmente utile per lo sviluppo. `run` eseguirà solo pacchetti il cui *nome del pacchetto* è **main** .

Per dimostrare, useremo un semplice esempio di Hello World: `main.go` :

```
package main

import fmt

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

Esegui senza compilarlo in un file:

```
go run main.go
```

Produzione:

```
Hello, World!
```

Esegui più file nel pacchetto

Se il pacchetto è **principale** e suddiviso in più file, uno deve includere gli altri file nel comando di `run` :

```
go run main.go assets.go
```

Vai a costruire

`go build` compilerà un programma in un file eseguibile.

Per dimostrare, useremo un semplice esempio di Hello World: main.go:

```
package main

import fmt

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

Compila il programma:

```
go build main.go
```

`build` crea un programma eseguibile, in questo caso: `main` o `main.exe` . È quindi possibile eseguire questo file per vedere l'output `Hello, World!` . Puoi anche copiarlo su un sistema simile che non ha installato Go, *renderlo eseguibile* ed eseguirlo lì.

Specificare OS o Architecture in build:

Puoi specificare quale sistema o architettura costruire compilando `env` before `build` :

```
env GOOS=linux go build main.go # builds for Linux
env GOARCH=arm go build main.go # builds for ARM architecture
```

Costruisci più file

Se il pacchetto è diviso in più file **e** il nome del pacchetto è **main** (ovvero *non è un pacchetto importable*), è necessario specificare tutti i file da compilare:

```
go build main.go assets.go # outputs an executable: main
```

Costruire un pacchetto

Per creare un pacchetto chiamato `main` , puoi semplicemente usare:

```
go build . # outputs an executable with name as the name of enclosing folder
```

Vai pulito

`go clean` pulirà tutti i file temporanei creati durante il richiamo `go build` su un programma. Pulirà anche i file rimasti da Makefile.

Vai a Fmt

`go fmt` formatterà il codice sorgente di un programma in un modo pulito, idiomatico, di facile lettura e comprensione. Si consiglia di utilizzare `go fmt` su qualsiasi sorgente prima di inviarlo per la visualizzazione pubblica o di impegnarsi in un sistema di controllo della versione, per facilitare la

lettura.

Per formattare un file:

```
go fmt main.go
```

O tutti i file in una directory:

```
go fmt myProject
```

Puoi anche usare `gofmt -s` (**non** `go fmt`) per tentare di semplificare qualsiasi codice che possa.

`gofmt` (**not** `go fmt`) può essere utilizzato anche per il codice refactoring. Comprende Go, quindi è più potente dell'utilizzo di una semplice ricerca e sostituzione. Ad esempio, dato questo programma (`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")
}
```

Puoi sostituire il metodo `Original` con `Refactor` con `gofmt` :

```
gofmt -r 'Original -> Refactor' -d main.go
```

Quale produrrà la 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")
 }
```

Vai a prendere

`go get` scaricare i pacchetti nominati dai percorsi di importazione, insieme alle loro dipendenze. Quindi installa i pacchetti con nome, come "vai su". Get accetta anche i flag di compilazione per

controllare l'installazione.

    vai a ottenere github.com/maknahar/phonecountry

Quando si estrae un nuovo pacchetto, ottenere crea la directory di destinazione `$GOPATH/src/<import-path>` . Se GOPATH contiene più voci, usa il primo. Allo stesso modo, installerà i binari compilati in `$GOPATH/bin` .

Durante il check-out o l'aggiornamento di un pacchetto, cerca un ramo o un tag che corrisponda alla versione di Go installata localmente. La regola più importante è che se l'installazione locale sta eseguendo la versione "go1", cerca le ricerche per un ramo o un tag chiamato "go1". Se non esiste una tale versione, recupera la versione più recente del pacchetto.

Quando si usa `go get` , il flag `-d` lo fa scaricare ma non installa il pacchetto specificato. Il flag `-u` consentirà di aggiornare il pacchetto e le sue dipendenze.

Non eseguire mai il check out o aggiornare il codice memorizzato nelle directory del fornitore.

Vai env

`go env [var ...]` stampa vai informazioni sull'ambiente.

    Di default stampa tutte le informazioni.

```
$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"
```

    Se uno o più nomi di variabili vengono forniti come argomenti, stampa il valore di ciascuna variabile denominata sulla propria riga.

```
$go env GOOS GOPATH
```

```
darwin
/Users/vikashkv/work
```

Leggi Il comando Go online: https://riptutorial.com/it/go/topic/4828/il-comando-go

Capitolo 33: immagini

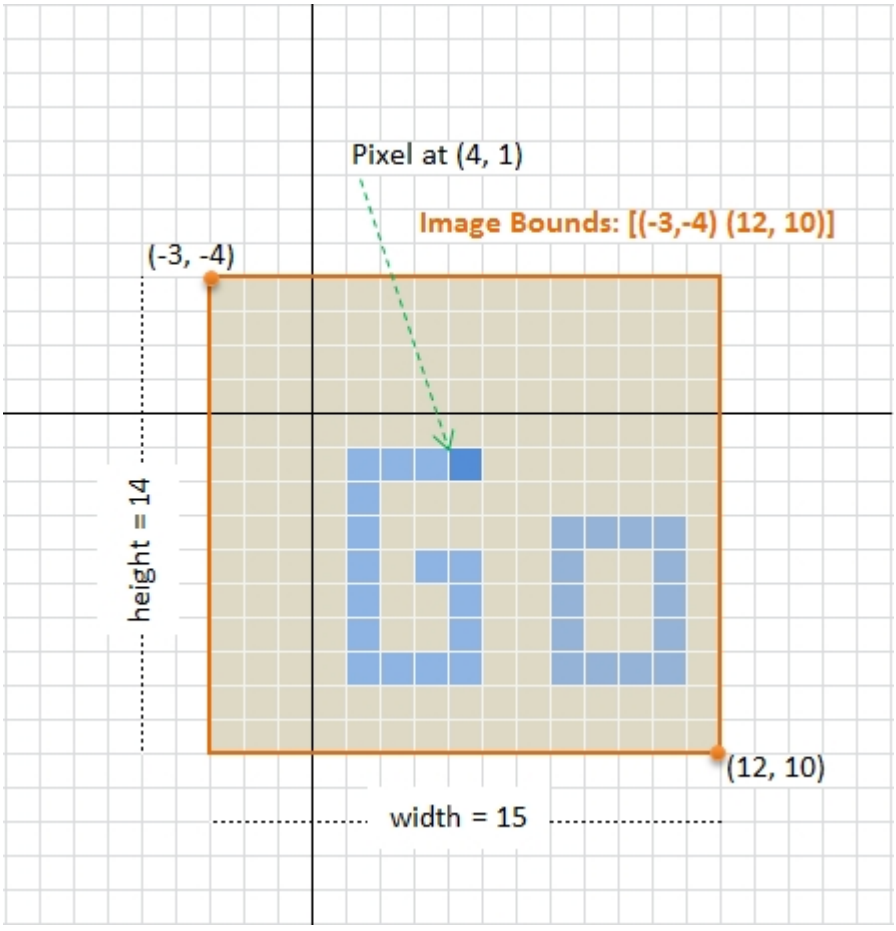
introduzione

Il pacchetto di immagini offre funzionalità di base per lavorare con immagini 2D. Questo argomento descrive diverse operazioni di base quando si lavora con immagini come la lettura e la scrittura di un particolare formato di immagine, il ritaglio, l'accesso e la modifica di *pixel* , la conversione del colore, il ridimensionamento e il filtro di base delle immagini.

Examples

Concetti basilari

Un'immagine rappresenta una griglia rettangolare di elementi dell'immagine (*pixel*). Nel pacchetto immagine , il pixel è rappresentato come uno dei colori definiti nel pacchetto immagine / colore . La geometria 2D dell'immagine è rappresentata come immagine. `image.Rectangle` , mentre `image.Point` indica una posizione sulla griglia.



La figura sopra illustra i concetti di base di un'immagine nel pacchetto. Un'immagine di dimensioni 15x14 pixel ha un *limite* rettangolare iniziato nell'angolo in *alto a sinistra* (es. Coordinata (-3, -4) nella figura sopra), e i suoi assi aumentano verso destra e verso il *basso* nell'angolo in *basso a destra* (es. Coordinata (12, 10) nella figura). Si noti che i limiti **non iniziano necessariamente da**

o contengono il punto (0,0) .

Immagine relativa al *tipo*

In `Go` , un'immagine implementa sempre la seguente `image.Image` Interfaccia immagine

```
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
}
```

in cui l'interfaccia `color.Color` è definita come

```
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)
}
```

e `color.Model` è un'interfaccia dichiarata come

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

Accesso alla dimensione e al pixel dell'immagine

Supponiamo di avere un'immagine memorizzata come `img` variabile, quindi possiamo ottenere la dimensione e il pixel dell'immagine per:

```
// 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
    }
}
```

Si noti che nel pacchetto, il valore di ciascun componente `R,G,B,A` è compreso tra `0-65535` (`0x0000 - 0xffff`), **non** `0-255` .

Caricamento e salvataggio dell'immagine

In memoria, un'immagine può essere vista come una matrice di pixel (colore). Tuttavia, quando un'immagine viene archiviata in una memoria permanente, può essere memorizzata come è (formato RAW), bitmap o altri formati di immagine con un particolare algoritmo di compressione per risparmiare spazio di archiviazione, ad esempio PNG, JPEG, GIF, ecc. Quando si carica un'immagine con un particolare formato, l'immagine deve essere *decodificata* in immagine. `image.Image` con algoritmo corrispondente. Una funzione `image.Decode` dichiarata come

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

è fornito per questo particolare utilizzo. Per poter gestire vari formati di immagine, prima di chiamare la funzione `image.Decode` , il decoder deve essere registrato tramite l' `image.RegisterFormat` Funzione `image.RegisterFormat` definita come

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

Attualmente, il pacchetto di immagini supporta tre formati di file: JPEG , GIF e PNG . Per registrare un decodificatore, aggiungere quanto segue

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

al pacchetto `main` dell'applicazione. Da qualche parte nel tuo codice (non necessario nel pacchetto `main`), per caricare un'immagine JPEG, usa i seguenti frammenti:

```
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` ...
```

Salva in PNG

Per salvare un'immagine in un formato particolare, il *codificatore* corrispondente deve essere importato esplicitamente, ad es

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

quindi un'immagine può essere salvata con i seguenti frammenti:

```
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
}
```

Se si desidera specificare un livello di compressione diverso dal livello `DefaultCompression` , creare un *encoder* , ad es

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

Salva in JPEG

Per salvare in formato `jpeg` , utilizzare quanto segue:

```
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
}
```

Salva in GIF

Per salvare l'immagine nel file GIF, utilizzare i seguenti frammenti.

```
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
}
```

Ritagliare l'immagine

La maggior parte del tipo di immagine nel pacchetto di immagini con `SubImage(r Rectangle) Image` Metodo `SubImage(r Rectangle) Image` , ad eccezione di `image.Uniform` . Sulla base di questo fatto, possiamo implementare una funzione per ritagliare un'immagine arbitraria come segue

```
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
}
```

Si noti che l'immagine ritagliata potrebbe condividere i suoi pixel sottostanti con l'immagine originale. Se questo è il caso, qualsiasi modifica all'immagine ritagliata influenzerà l'immagine originale.

Converti l'immagine a colori in scala di grigi

È sufficiente un algoritmo di elaborazione delle immagini digitali come il rilevamento dei bordi, le informazioni trasportate dall'intensità dell'immagine (ad es. Il valore della scala dei grigi). L'uso delle informazioni sul colore (canale `R, G, B`) può fornire risultati leggermente migliori, ma la complessità dell'algoritmo sarà aumentata. Pertanto, in questo caso, è necessario convertire l'immagine a colori in un'immagine in scala di grigi prima di applicare tale algoritmo.

Il seguente codice è un esempio di conversione di un'immagine arbitraria in un'immagine in scala di grigi a 8 bit. L'immagine viene recuperata dalla posizione remota utilizzando il pacchetto `net/http` , convertita in scala di grigi e infine salvata come immagine 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 conversione del colore si verifica quando si assegna pixel attraverso `Set(x, y int, c color.Color)` che è implementato in `image.go` come

```
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
}
```

in cui, `color.GrayModel` è definito in `color.go` come

```
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)}
}
```

In base ai fatti sopra riportati, l'intensità `Y` viene calcolata con la seguente formula:

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

Se vogliamo applicare diverse formule / algoritmi per convertire un colore in un'intesità, ad es

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

quindi, è possibile utilizzare i seguenti frammenti.

```
// 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)
    }
}
```

Il calcolo sopra è fatto per moltiplicazione in virgola mobile, e certamente non è il più efficiente, ma è sufficiente per dimostrare l'idea. L'altro punto è, quando si chiama `Set(x, y int, c color.Color)` con `color.Gray` come terzo argomento, il modello di colore non eseguirà la conversione del colore come si può vedere nella precedente funzione `grayModel` .

Leggi immagini online: https://riptutorial.com/it/go/topic/10557/immagini

Capitolo 34: Iniziare con Go Using Atom

introduzione

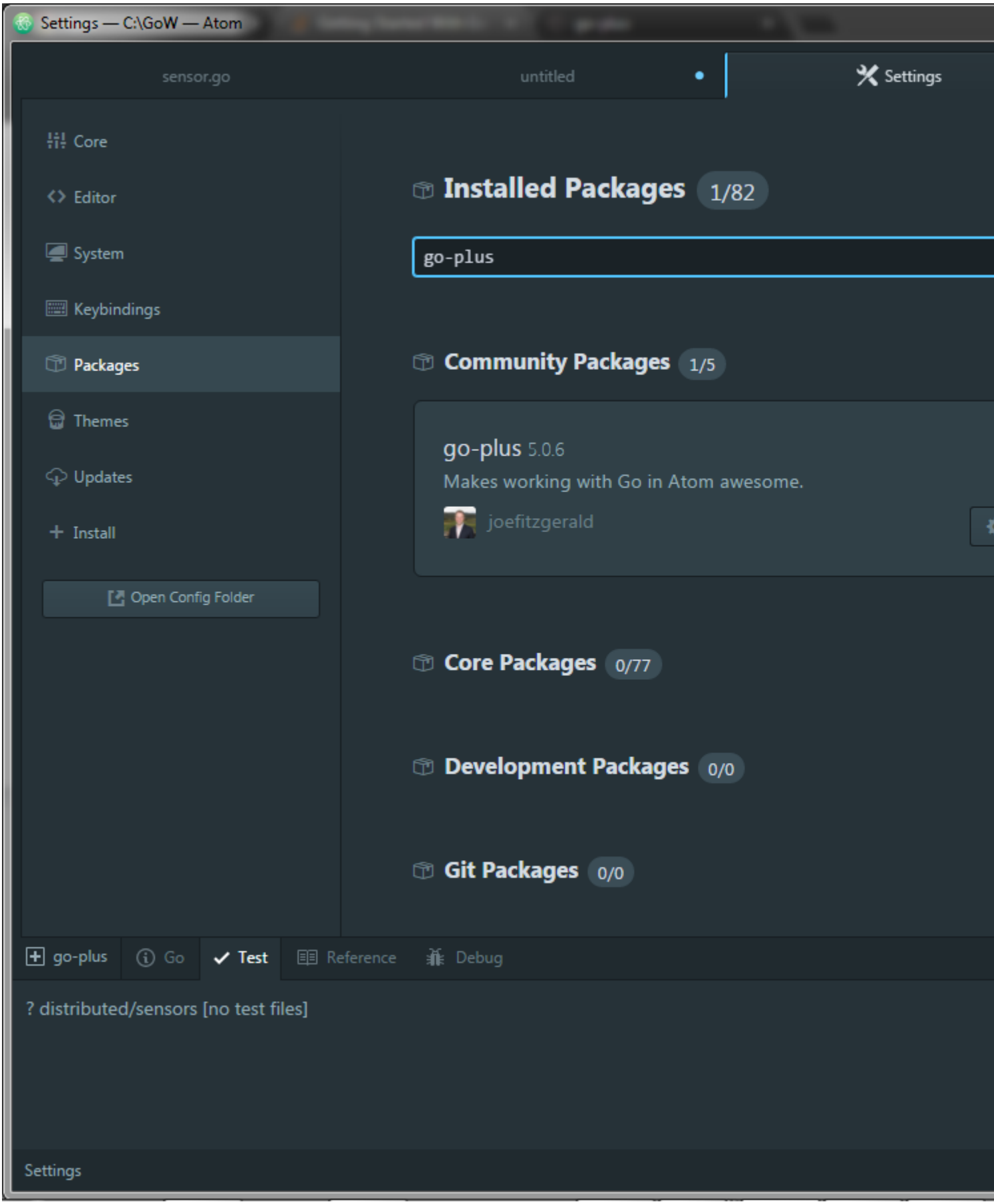
Dopo aver installato go (http://www.Scriptutorial.com/go/topic/198/getting-started-with-go) avrai bisogno di un ambiente. Un modo efficiente e gratuito per iniziare è l'utilizzo di Atom text editor (https://atom.io) e gulp. Una domanda che forse ti ha attraversato la mente è *perché usare gulp?* . Abbiamo bisogno di gulp per il completamento automatico. Iniziamo!

Examples

Ottieni, installa e installa Atom & Gulp

1. Installa Atom. Puoi prendere atomo da qui
2. Vai alle impostazioni Atom (ctrl +,). Pacchetti -> Installa pacchetto go-plus (go-plus)

Dopo aver installato go-plus in Atom:



3. Ottieni queste dipendenze usando go get o un altro gestore delle dipendenze: (apri una console ed esegui questi comandi)

   vai a get -u golang.org/x/tools/cmd/goimports

vai a get -u golang.org/x/tools/cmd/gorename

vai a get -u github.com/sqs/goreturns

vai a get -u github.com/nsf/gocode

vai a get -u github.com/alecthomas/gometalinter

vai a get -u github.com/zmb3/gogetdoc

vai a get -u github.com/rogpeppe/godef

vai a get -u golang.org/x/tools/cmd/guru

4. Installa Gulp (Gulpjs) usando npm o qualsiasi altro gestore di pacchetti (gulp-getting-started-doc):

   $ npm install --global gulp

Crea $ GO_PATH / gulpfile.js

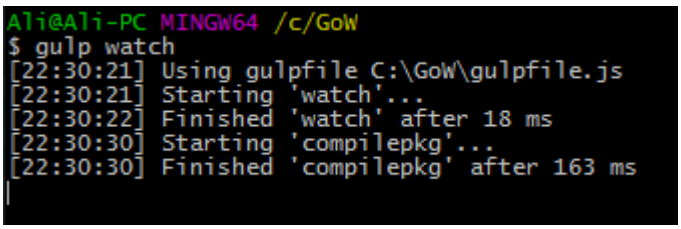
```
var gulp = require('gulp');
var path = require('path');
var shell = require('gulp-shell');

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']);
});
```

Nel codice sopra abbiamo definito un'attività *compliepkg* che verrà attivata ogni volta che un file go in goPath (src / mypackage /) o subdirectory cambia. l'attività eseguirà il comando shell go install changed_file.go

Dopo aver creato il file gulp in go path e definito l'attività aprire una riga di comando ed eseguire:

gulp watch

Vedrai qualcosa di simile ogni volta che un file cambia:

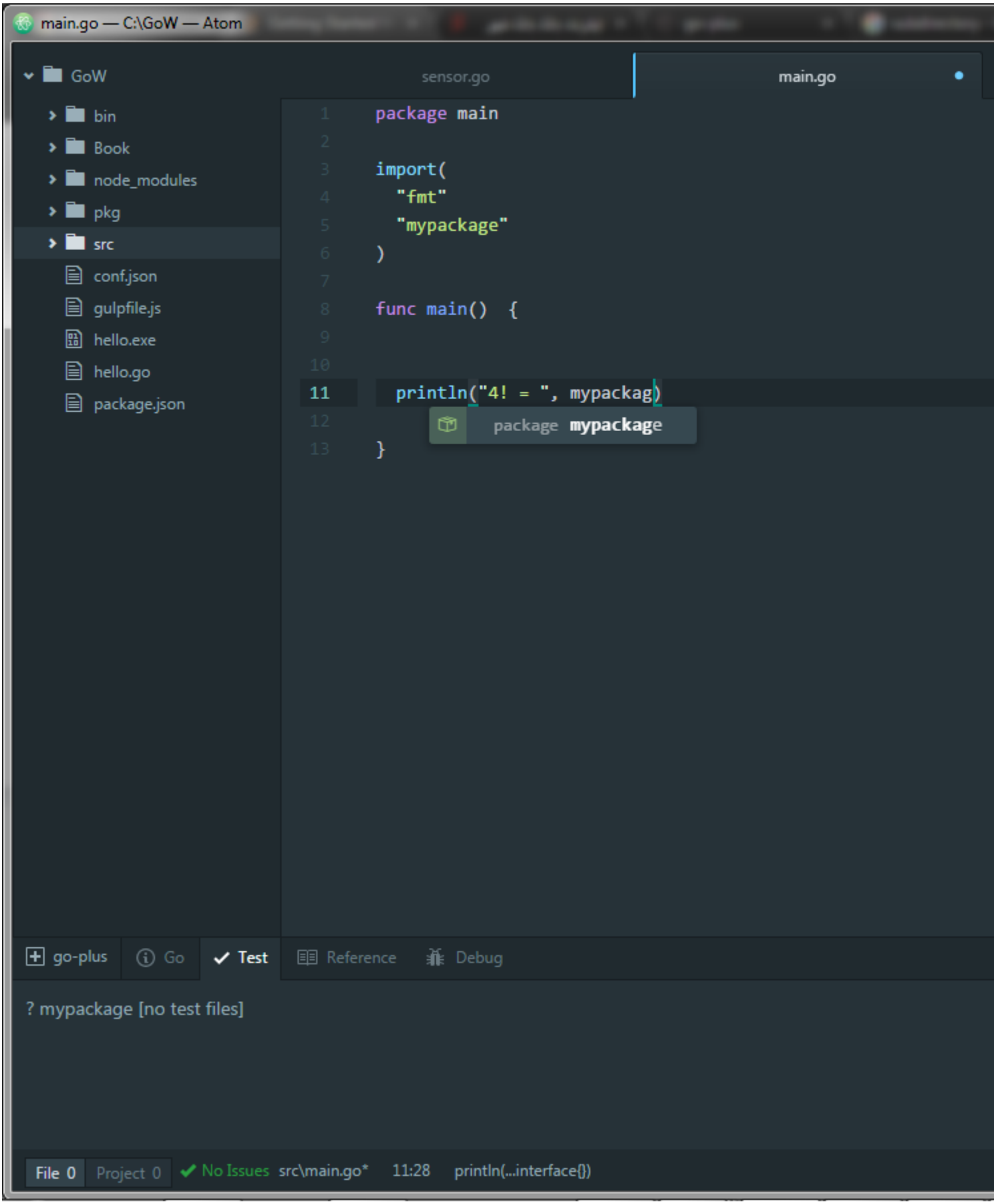


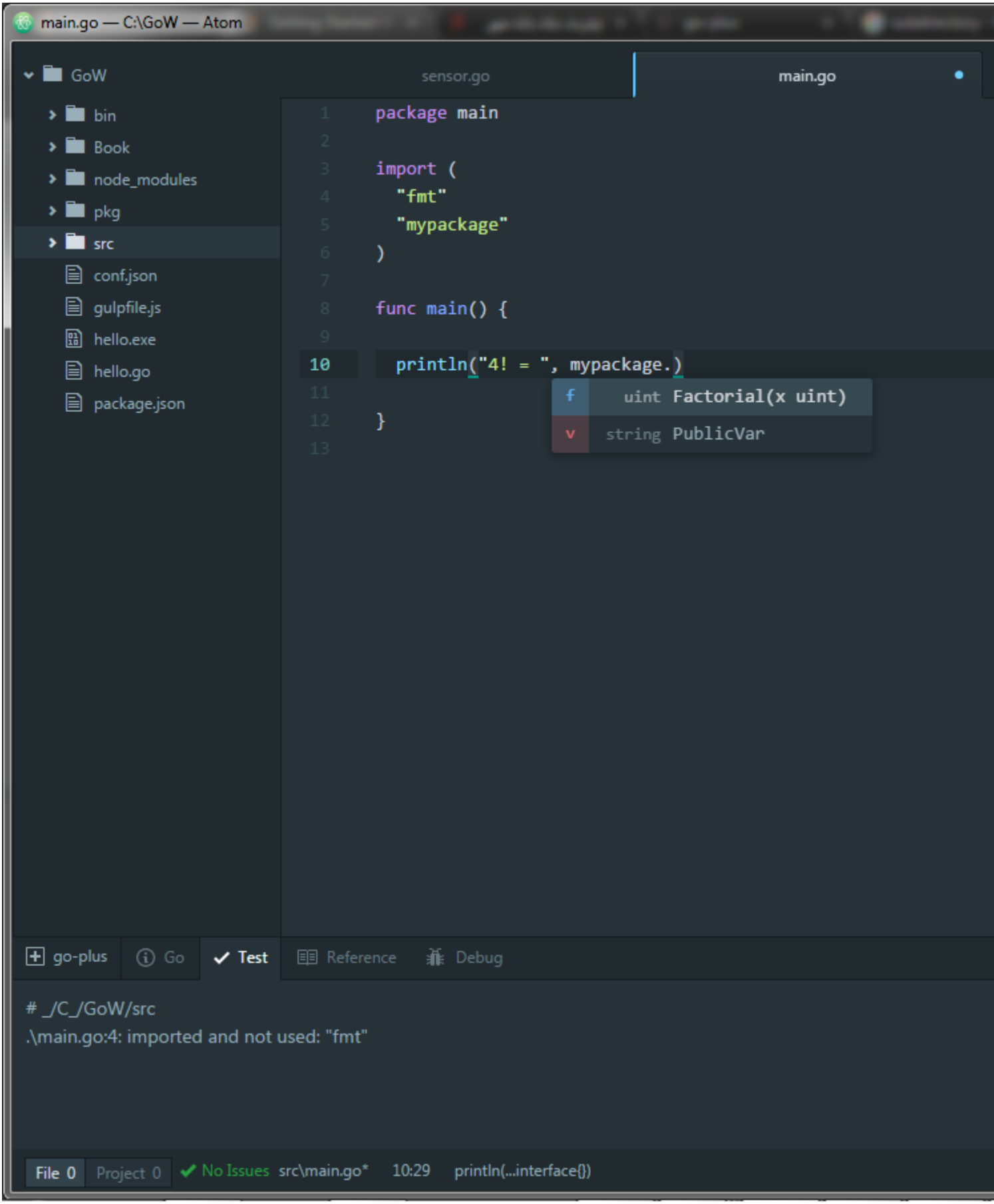
Crea $ GO_PATH / mypackage / source.go

```
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)
}
```

Creazione di $ GO_PATH / main.go

Ora puoi iniziare a scrivere il tuo codice personale con il completamento automatico usando Atom e 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
package main
import(
"fmt"
"mypackage"
)
func main() {
println("4! = ", mypackag)
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
```

Leggi Iniziare con Go Using Atom online: https://riptutorial.com/it/go/topic/8592/iniziare-con-go-using-atom

Capitolo 35: Installazione

Examples

Installa in Linux o Ubuntu

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

Puoi trovare gli elenchi delle versioni qui .

```
# 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
```

Aggiorna `$GOPATH`

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

Aggiungi le seguenti due righe alla fine del file ~ / .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
```

Ora sei pronto per andare, prova la tua versione go usando:

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

Leggi Installazione online: https://riptutorial.com/it/go/topic/5776/installazione

Capitolo 36: Installazione

Osservazioni

Scarica Go

Visita l' elenco dei download e trova l'archivio giusto per il tuo sistema operativo. I nomi di questi download possono essere un po 'criptici per i nuovi utenti.

I nomi sono nel formato go [versione]. [Sistema operativo] - [architettura]. [Archivio]

Per la versione, si desidera scegliere la più recente disponibile. Queste dovrebbero essere le prime opzioni che vedi.

Per il sistema operativo, questo è abbastanza auto-esplicativo tranne per gli utenti Mac, dove il sistema operativo è chiamato "darwin". Questo nome prende il nome dalla parte open-source del sistema operativo utilizzato dai computer Mac .

Se si esegue una macchina a 64 bit (che è la più comune nei computer moderni), la parte "architettura" del nome del file deve essere "amd64". Per le macchine a 32 bit, sarà "386". Se sei su un dispositivo ARM come un Raspberry Pi, ti consigliamo "armv6l".

Per la parte "archivio", gli utenti Mac e Windows hanno due opzioni perché Go fornisce programmi di installazione per tali piattaforme. Per Mac, probabilmente vuoi "pkg". Per Windows, probabilmente si desidera "msi".

Quindi, ad esempio, se sono su una macchina Windows a 64 bit e voglio scaricare Go 1.6.3, il download che desidero sarà denominato:

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

Estrazione dei file di download

Ora che abbiamo scaricato un archivio Go, dobbiamo estrarlo da qualche parte.

Mac e Windows

Poiché gli installer sono forniti per queste piattaforme, l'installazione è semplice. Basta eseguire il programma di installazione e accettare i valori predefiniti.

Linux

Non esiste un programma di installazione per Linux, quindi è necessario un po 'più di lavoro. Dovresti aver scaricato un file con il suffisso ".tar.gz". Questo è un file di archivio, simile a un file

".zip". Dobbiamo estrarlo. Estraiamo i file Go in `/usr/local` perché è la posizione consigliata.

Apri un terminale e cambia le directory nel punto in cui hai scaricato l'archivio. Questo è probabilmente nei `Downloads` . In caso contrario, sostituire la directory nel seguente comando in modo appropriato.

```
cd Downloads
```

Ora, esegui quanto segue per estrarre l'archivio in `/usr/local` , sostituendo `[filename]` con il nome del file che hai scaricato.

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

Impostazione delle variabili d'ambiente

C'è ancora un passo da fare prima di essere pronto per iniziare a sviluppare. Abbiamo bisogno di impostare variabili d'ambiente, che sono informazioni che gli utenti possono modificare per dare ai programmi un'idea migliore della configurazione dell'utente.

finestre

È necessario impostare `GOPATH` , che è la cartella in cui si andrà a lavorare.

È possibile impostare le variabili di ambiente tramite il pulsante "Variabili d'ambiente" nella scheda "Avanzate" del pannello di controllo "Sistema". Alcune versioni di Windows forniscono questo pannello di controllo tramite l'opzione "Impostazioni di sistema avanzate" all'interno del pannello di controllo "Sistema".

Il nome della nuova variabile di ambiente dovrebbe essere "GOPATH". Il valore dovrebbe essere il percorso completo di una directory in cui verrà sviluppato il codice Go. Una cartella chiamata "go" nella directory dell'utente è una buona scelta.

Mac

È necessario impostare `GOPATH` , che è la cartella in cui si andrà a lavorare.

Modifica un file di testo chiamato ".bash_profile", che dovrebbe essere nella tua directory utente, e aggiungi la seguente nuova riga alla fine, sostituendo `[work area]` con un percorso completo per una directory che vorresti fare Vai a lavorare. Se ".bash_profile" non esiste, crealo. Una cartella chiamata "go" nella tua directory utente è una buona scelta.

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

Linux

Perché Linux non ha un programma di installazione, richiede un po 'più di lavoro. Abbiamo bisogno di mostrare il terminale in cui sono presenti il compilatore Go e altri strumenti, e dobbiamo

impostare `GOPATH` , che è una cartella in cui dovrai lavorare.

Modifica un file di testo chiamato ".profile", che dovrebbe essere nella tua directory utente, e aggiungi la seguente riga alla fine, sostituendo `[work area]` con un percorso completo per una directory che vorresti fare Vai a lavorare. Se " .profile "non esiste, crealo. Una cartella chiamata "go" nella tua directory utente è una buona scelta.

Quindi, su un'altra nuova riga, aggiungi quanto segue al tuo file ".profile".

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

Finito!

Se gli strumenti Go non sono ancora disponibili nel terminale, prova a chiudere la finestra e ad aprire una nuova finestra di terminale.

Examples

Esempio .profile o .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
```

Leggi Installazione online: https://riptutorial.com/it/go/topic/6213/installazione

Capitolo 37: interfacce

Osservazioni

Le interfacce in Go sono solo set di metodi fissi. Un tipo implementa *implicitamente* un'interfaccia se il suo set di metodi è un superset dell'interfaccia. *Non c'è una dichiarazione di intenti.*

Examples

Interfaccia semplice

In Go, un'interfaccia è solo un insieme di metodi. Usiamo un'interfaccia per specificare un comportamento di un dato oggetto.

```
type Painter interface {
    Paint()
}
```

Il tipo di implementazione **non deve** dichiarare che sta implementando l'interfaccia. È sufficiente definire i metodi della stessa firma.

```
type Rembrandt struct{}

func (r Rembrandt) Paint() {
    // use a lot of canvas here
}
```

Ora possiamo usare la struttura come interfaccia.

```
var p Painter
p = Rembrandt{}
```

Un'interfaccia può essere soddisfatta (o implementata) da un numero arbitrario di tipi. Inoltre un tipo può implementare un numero arbitrario di interfacce.

```
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")
}
```

Qui, la struttura `Human` soddisfa sia l'interfaccia `Singer` che quella `Writer` , ma la struttura `OnlySinger` soddisfa solo l'interfaccia `Singer` .

Interfaccia vuota

C'è un tipo di interfaccia vuoto, che non contiene metodi. Lo dichiariamo come `interface{}` . Questo non contiene metodi, quindi ogni `type` soddisfa. Quindi l'interfaccia vuota può contenere qualsiasi valore di tipo.

```
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"}
```

Il caso d'uso più comune per le interfacce è garantire che una variabile supporti uno o più comportamenti. Al contrario, il caso d'uso principale per l'interfaccia vuota è definire una variabile che può contenere qualsiasi valore, indipendentemente dal suo tipo concreto.

Per riportare questi valori come i loro tipi originali, dobbiamo solo farlo

```
i = a.(int)
s = a.(string)
m := a.(*StructType)
```

o

```
i, ok := a.(int)
s, ok := a.(string)
m, ok := a.(*StructType)
```

`ok` indica se l' `interface a` è convertibile in un determinato tipo. Se non è possibile il cast `ok` sarà `false` .

Valori dell'interfaccia

Se dichiari una variabile di un'interfaccia, può memorizzare qualsiasi tipo di valore che implementa i metodi dichiarati dall'interfaccia!

Se dichiariamo `h interface Singer` , può memorizzare un valore di tipo `Human` o `OnlySinger`. Ciò è dovuto al fatto che tutti implementano i metodi specificati dall'interfaccia `Singer` .

```
var h Singer
h = &human{}

h.Sing()
```

Determinazione del tipo sottostante dall'interfaccia

In go a volte può essere utile sapere quale tipo di sottofondo ti è stato passato. Questo può essere fatto con un interruttore di tipo. Questo presuppone che abbiamo due strutture:

```
type Rembrandt struct{}

func (r Rembrandt) Paint() {}

type Picasso struct{}

func (r Picasso) Paint() {}
```

Che implementano l'interfaccia di Painter:

```
type Painter interface {
    Paint()
}
```

Quindi possiamo usare questa opzione per determinare il tipo sottostante:

```
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")
    }
}
```

Controllo in fase di compilazione se un tipo soddisfa un'interfaccia

Interfacce e implementazioni (tipi che implementano un'interfaccia) sono "distaccati". Quindi è una domanda legittima come verificare in fase di compilazione se un tipo implementa un'interfaccia.

Un modo per chiedere al compilatore di verificare che il tipo `T` implementa l'interfaccia `I` tentando un'assegnazione usando il valore zero per `T` o puntatore a `T` , a seconda dei casi. E potremmo

scegliere di assegnare all'identificatore vuoto per evitare inutili immondizie:

```
type T struct{}

var _ I = T{}       // Verify that T implements I.
var _ I = (*T)(nil) // Verify that *T implements I.
```

Se `T` o `*T` non implementa `I` , sarà un errore in fase di compilazione.

Questa domanda appare anche nelle FAQ ufficiali: come posso garantire che il mio tipo soddisfi un'interfaccia?

Digitare interruttore

Gli switch di tipo possono essere utilizzati anche per ottenere una variabile che corrisponda al tipo del caso:

```
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")
    }
}
```

Asserzione di tipo

È possibile accedere al tipo di dati reale dell'interfaccia con Asserzione di tipo.

```
interfaceVariable.(DataType)
```

Esempio di struct `MyType` che implementa l'interfaccia `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)
}
```

Senza `.(*MyType)` non saremmo in grado di accedere a `Msg` Field. Se proviamo `interfaceVar.Msg` mostrerà l'errore di compilazione:

```
interfaceVar.Msg undefined (type Subber has no field or method Msg)
```

Vai Interfacce da un Aspetto Matematico

In matematica, specialmente *Set Theory* , abbiamo una collezione di cose che è chiamata *set* e chiamiamo quelle cose come *elementi* . Mostriamo un set con il suo nome come A, B, C, ... o esplicitamente con il suo membro su notazione brace: {a, b, c, d, e}. Supponiamo di avere un elemento arbitrario x e un insieme Z, la domanda chiave è: "Come possiamo capire che x è membro di Z o no?". Matematica risposta a questa domanda con un concetto: **Proprietà caratteristica** di un insieme. *La proprietà caratteristica* di un insieme è un'espressione che descrive completamente il set. Ad esempio abbiamo impostato *Natural Numbers* che è {0, 1, 2, 3, 4, 5, ...}. Possiamo descrivere questo insieme con questa espressione: {a_n | $a_0 = 0$, $a_n = a_{n-1} +1$}. Nell'ultima espressione $a_0 = 0$, $a_n = a_{n-1} +1$ è la proprietà caratteristica dell'insieme di numeri naturali. **Se abbiamo questa espressione, possiamo costruire completamente questo set** . Lascia che descriva l'insieme di *numeri pari* in questo modo. Sappiamo che questo set è composto da questi numeri: {0, 2, 4, 6, 8, 10, ...}. Con un'occhiata capiamo che tutti questi numeri sono anche un *numero naturale* , in altre parole *se aggiungiamo alcune condizioni extra alla proprietà caratteristica dei numeri naturali, possiamo costruire una nuova espressione che descrive questo insieme* . Quindi possiamo descrivere con questa espressione: {n | n è un membro di numeri naturali *e* il promemoria di n su 2 è zero}. Ora possiamo creare un filtro che ottiene la proprietà caratteristica di un set e filtra alcuni elementi desiderati per restituire elementi del nostro set. Ad esempio, se abbiamo un filtro di numeri naturali, entrambi i numeri naturali e quelli pari possono passare questo filtro, ma se abbiamo un filtro di numero pari, alcuni elementi come 3 e 137871 non possono passare il filtro.

La definizione dell'interfaccia in Go è come definire la proprietà caratteristica e il meccanismo di utilizzo dell'interfaccia come argomento di una funzione è come un filtro che rileva che l'elemento è un membro del set desiderato o meno. Descrivi questo aspetto con il codice:

```
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 proprietà caratteristica di `Number` è tutte le strutture che hanno il metodo `IsNumber` , per `NaturalNumber` tutti quelli che hanno i metodi `IsNumber` e `IsNaturalNumber` e infine `EvenNumber` è tutti i tipi che hanno `IsNumber` , `IsNaturalNumber` e `IsEvenNumber` . Grazie a questa interpretazione dell'interfaccia, possiamo facilmente capire che poiché `interface{}` non ha alcuna proprietà caratteristica, accetta tutti i tipi (perché non ha alcun filtro per distinguere tra valori).

Leggi interfacce online: https://riptutorial.com/it/go/topic/1221/interfacce

Capitolo 38: Invia / ricevi email

Sintassi

- func PlainAuth (identità, nome utente, password, stringa host) Aut
- func SendMail (stringa addr, un errore Auth, da stringa, a [] stringa, msg [] byte)

Examples

Invio di email con smtp.SendMail ()

L'invio di e-mail è piuttosto semplice in Go. Aiuta a capire la RFC 822, che specifica lo stile di cui ha bisogno l'e-mail, il codice sottostante invia una e-mail conforme a 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!")
}
```

Quanto sopra invierà un messaggio simile al seguente:

```
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!
.
```

Leggi Invia / ricevi email online: https://riptutorial.com/it/go/topic/5912/invia---ricevi-email

Capitolo 39: Iota

introduzione

Iota fornisce un modo per dichiarare le costanti numeriche da un valore iniziale che cresce monotonicamente. Iota può essere utilizzato per dichiarare maschere di bit che vengono spesso utilizzate nella programmazione di sistemi e reti e altri elenchi di costanti con valori correlati.

Osservazioni

L'identificatore `iota` viene utilizzato per assegnare valori agli elenchi di costanti. Quando iota viene utilizzato in un elenco, inizia con un valore pari a zero e aumenta di uno per ogni valore nell'elenco di costanti e viene reimpostato su ogni parola chiave `const` . A differenza delle enumerazioni di altre lingue, iota può essere utilizzato nelle espressioni (ad esempio `iota + 1`) che consente una maggiore flessibilità.

Examples

Semplice utilizzo di iota

Per creare un elenco di costanti - assegnare un valore `iota` a ciascun elemento:

```
const (
  a = iota // a = 0
  b = iota // b = 1
  c = iota // c = 2
)
```

Per creare un elenco di costanti in modo abbreviato, assegna il valore `iota` al primo elemento:

```
const (
  a = iota // a = 0
  b        // b = 1
  c        // c = 2
)
```

Usare iota in un'espressione

`iota` può essere utilizzato nelle espressioni, quindi può anche essere utilizzato per assegnare valori diversi dai semplici numeri interi incrementali a partire da zero. Per creare costanti per unità SI, utilizzare questo esempio di 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
)
```

Saltare valori

Il valore di `iota` viene comunque incrementato per ogni voce in un elenco costante anche se iota non viene utilizzato:

```
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
)
```

verrà anche incrementato anche se non viene creata alcuna costante, il che significa che l'identificatore vuoto può essere utilizzato per saltare interamente i valori:

```
const (
  a = iota // a = 0
  _        // iota is incremented
  b        // b = 2
)
```

Il primo blocco di codice è stato preso da Go Spec (CC-BY 3.0).

Uso di iota in un elenco di espressioni

Poiché `iota` viene incrementato dopo ogni `ConstSpec` , i valori all'interno dello stesso elenco di espressioni avranno lo stesso valore per lo `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
)
```

Questo esempio è stato preso da Go Spec (CC-BY 3.0).

Uso di iota in una maschera di bit

Iota può essere molto utile quando si crea una maschera di bit. Ad esempio, per rappresentare lo stato di una connessione di rete che può essere sicura, autenticata e / o pronta, potremmo creare una maschera di bit simile alla seguente:

```
const (
    Secure = 1 << iota // 0b001
    Authn              // 0b010
    Ready              // 0b100
)

ConnState := Secure|Authn // 0b011: Connection is secure and authenticated, but not yet Ready
```

Uso di iota in const

Questa è un'enumerazione per la creazione di cost. Il compilatore inizia iota da 0 e incrementa di uno per ogni costante successiva. Il valore è determinato in fase di compilazione anziché in fase di esecuzione. Per questo motivo non possiamo applicare iota alle espressioni che vengono valutate in fase di esecuzione.

Programma per utilizzare iota in 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)
}
```

Provalo in Go Playground

Leggi Iota online: https://riptutorial.com/it/go/topic/2865/iota

Capitolo 40: JSON

Sintassi

- func Marshal (v interface {}) ([] byte, errore)
- func Errore Unmarshal (data [] byte, v interfaccia {})

Osservazioni

Il pacchetto `"encoding/json"` Package json implementa la codifica e la decodifica degli oggetti JSON in `Go` .

I tipi in JSON insieme ai tipi di calcestruzzo corrispondenti in Go sono:

Tipo JSON	Vai tipo di cemento
booleano	bool
numeri	float64 o int
stringa	stringa
nullo	zero

Examples

Codifica JSON di base

`json.Marshal` dal pacchetto `"encoding/json"` codifica un valore per JSON.

Il parametro è il valore da codificare. I valori restituiti sono una matrice di byte che rappresentano l'input codificato JSON (in caso di successo) e un errore (in caso di errore).

```
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"]"
```

Terreno di gioco

Ecco alcuni esempi di base di codifica per i tipi di dati incorporati:

```
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}
```

Terreno di gioco

La codifica di variabili semplici è utile per capire come funziona la codifica JSON in Go. Tuttavia, nel mondo reale, probabilmente codificherete dati più complessi memorizzati nelle strutture .

Decodifica JSON di base

`json.Unmarshal` dal pacchetto `"encoding/json"` decodifica un valore JSON nel valore indicato dalla variabile data.

I parametri sono il valore da decodificare in `[]bytes` e una variabile da utilizzare come memoria per il valore de-serializzato. Il valore restituito è un errore (in caso di errore).

```
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]
```

Terreno di gioco

Si noti come nell'esempio precedente abbiamo saputo in anticipo sia il tipo di chiave che il valore.

Ma questo non è sempre il caso. Di fatto, nella maggior parte dei casi il JSON contiene tipi di valori misti.

```
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"
```

Terreno di gioco

Nell'ultimo esempio sopra abbiamo usato una mappa generica per memorizzare il valore decodificato. Dobbiamo utilizzare un'interfaccia `map[string]interface{}` perché sappiamo che le chiavi sono stringhe, ma non conosciamo il tipo dei loro valori in anticipo.

Questo è un approccio molto semplice, ma è anche estremamente limitato. Nel mondo reale, generalmente decodificare un JSON in un tipo di `struct` definito dall'utente .

Decodifica dei dati JSON da un file

I dati JSON possono anche essere letti dai file.

Supponiamo di avere un file chiamato `data.json` con il seguente contenuto:

```
[
    {
      "Name" : "John Doe",
      "Standard" : 4
    },
    {
      "Name" : "Peter Parker",
      "Standard" : 11
    },
    {
      "Name" : "Bilbo Baggins",
      "Standard" : 150
    }
]
```

L'esempio seguente legge il file e decodifica il contenuto:

```
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)
    }
}
```

Il file `data.json` deve essere nella stessa directory del programma eseguibile Go. Leggi la documentazione di I / O sul file Go per ulteriori informazioni su come lavorare con i file in Go.

Utilizzo di strutture anonime per la decodifica

L'obiettivo con l'utilizzo di strutture anonime è quello di decodificare solo le informazioni che ci interessano senza sporcare la nostra app con tipi che vengono utilizzati solo in una singola funzione.

```
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)
```

Output: `{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}}]}`

Terreno di gioco

Per il caso generale vedi anche:
http://stackoverflow.com/documentation/go/994/json/4111/encoding-decoding-go-struct

Configurazione dei campi struct JSON

Considera il seguente esempio:

```
type Company struct {
    Name     string
    Location string
}
```

Nascondi / ignora determinati campi

Per esportare `Revenue` e `Sales` , ma nasconderli dalla codifica / decodifica, usa `json:"-"` o rinomina la variabile per iniziare con una lettera minuscola. Si noti che questo impedisce alla variabile di essere visibile all'esterno del pacchetto.

```
type Company struct {
    Name     string `json:"name"`
    Location string `json:"location"`
    Revenue  int    `json:"-"`
    sales    int
}
```

Ignora campi vuoti

Per evitare che `Location` venga incluso nel JSON quando è impostato sul suo valore zero, aggiungi `,omitempty` al tag `json` .

```
type Company struct {
    Name     string `json:"name"`
    Location string `json:"location,omitempty"`
}
```

Esempio nel parco giochi

Strutture di marshalling con campi privati

Come bravo sviluppatore hai creato la seguente struttura con campi sia esportati che non esportati:

```
type MyStruct struct {
    uuid string
    Name string
}
```

Esempio in Playground: https://play.golang.org/p/Zk94Il2ANZ

Ora vuoi `Marshal()` questa struttura in JSON valido per l'archiviazione in qualcosa come ecc. Tuttavia, poiché `uuid` non viene esportato, `json.Marshal()` salta. Cosa fare? Usa una struttura anonima e l'interfaccia `json.MarshalJSON()` ! Ecco un esempio:

```
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
}
```

Esempio in Playground: https://play.golang.org/p/Bv2k9GgbzE

Codifica / decodifica usando le strutture di Go

Supponiamo di avere la seguente `struct` che definisce un tipo di `City` :

```
type City struct {
    Name string
    Temperature int
}
```

Possiamo codificare / decodificare i valori della città usando il pacchetto `encoding/json` .

Prima di tutto, dobbiamo usare i metadati Go per dire al codificatore la corrispondenza tra i campi struct e le chiavi 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
}
```

Per mantenere semplice questo esempio, dichiareremo una corrispondenza esplicita tra i campi e le chiavi. Tuttavia, è possibile utilizzare diverse varianti del `json:` metadati come spiegato nei documenti .

IMPORTANTE: solo i campi esportati (campi con il nome maiuscolo) verranno serializzati / deserializzati. Ad esempio, se si *nomina* il campo ***t*** *emperature* , verrà ignorato anche se si impostano i metadati `json` .

Codifica

Per codificare una struttura di `City` , usa `json.Marshal` come nell'esempio di 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}
```

Terreno di gioco

decodifica

Per decodificare una struttura di `City` , usa `json.Unmarshal` come nell'esempio di 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}
```

Terreno di gioco

Leggi JSON online: https://riptutorial.com/it/go/topic/994/json

Capitolo 41: lettori

Examples

Utilizzo di bytes.Reader per leggere da una stringa

`io.Reader` dell'interfaccia `io.Reader` può essere trovata nel pacchetto `bytes` . Permette di usare una slice slice come sorgente per un Reader. In questo esempio la slice di byte è presa da una stringa, ma è più probabile che sia stata letta da un file o da una connessione di rete.

```
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)
```

Vai al parco giochi

Leggi lettori online: https://riptutorial.com/it/go/topic/7000/lettori

Capitolo 42: Loops

introduzione

Essendo una delle funzioni di base della programmazione, i loop sono un elemento importante in quasi tutti i linguaggi di programmazione. I loop consentono agli sviluppatori di impostare alcune parti del loro codice per ripetere attraverso un numero di cicli che vengono definiti iterazioni. Questo argomento riguarda l'utilizzo di più tipi di loop e applicazioni di loop in Go.

Examples

Loop di base

`for` è l'unica istruzione di loop in go, quindi l'implementazione di un ciclo di base potrebbe assomigliare a questa:

```
// 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
```

Rompere e continuare

L'interruzione del ciclo e il passaggio alla successiva iterazione sono supportati anche in Go, come in molte altre lingue:

```
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
```

Le istruzioni `break` e `continue` accettano inoltre un'etichetta facoltativa, utilizzata per identificare i loop esterni per il target con l'istruzione:

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

Ciclo condizionale

La parola chiave `for` viene anche utilizzata per i loop condizionali, tradizionalmente `while` loop si trovano in altri linguaggi di programmazione.

```
package main

import (
    "fmt"
)

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

giocarci sul campo da gioco

Produrrà:

```
1
2
3
```

ciclo infinito:

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

Diverse forme di loop

Forma semplice usando una variabile:

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

Utilizzando due variabili (o più):

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

Senza usare la dichiarazione di inizializzazione:

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

Senza un'espressione test:

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

Senza espressione di incremento:

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

Quando vengono rimosse tutte e tre le espressioni di inizializzazione, test e incremento, il ciclo diventa infinito:

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

Questo è un esempio di loop infinito con contatore inizializzato con zero:

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

Quando viene utilizzata solo l'espressione di test (si comporta come un tipico ciclo while):

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

Usando solo l'espressione di incremento:

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

Iterare su un intervallo di valori usando indice e valore:

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

Iterare su un intervallo usando solo indice:

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

Iterare su un intervallo usando solo indice:

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

Iterare su un intervallo usando il valore giusto:

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

Iterare su un intervallo utilizzando la chiave e il valore per la mappa (potrebbe non essere in ordine):

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

Iterare su un intervallo utilizzando solo il tasto per la mappa (potrebbe non essere in

ordine):

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

Iterare su un intervallo utilizzando solo il tasto per la mappa (potrebbe non essere in ordine):

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

Iterare su un intervallo utilizzando solo il valore per la mappa (potrebbe non essere in ordine):

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

Iterare su un intervallo per i canali (esce se il canale è chiuso):

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

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

Iterare su un intervallo per stringa (fornisce punti codice 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
```

come vedi `utf8str` ha 6 rune (punti codice Unicode) e 7 byte.

Ciclo temporizzato

```
package main

import(
    "fmt"
```

```
    "time"
)

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

Leggi Loops online: https://riptutorial.com/it/go/topic/975/loops

Capitolo 43: Mappe

introduzione

Le mappe sono tipi di dati utilizzati per l'archiviazione di coppie chiave-valore non ordinate, in modo che la ricerca del valore associato a una determinata chiave sia molto efficiente. Le chiavi sono uniche. La struttura dei dati sottostanti cresce come necessario per accogliere nuovi elementi, quindi il programmatore non deve preoccuparsi della gestione della memoria. Sono simili a quelli che altri linguaggi chiamano tabelle hash, dizionari o array associativi.

Sintassi

- var mapName map [KeyType] ValueType // dichiara una mappa
- var mapName = map [KeyType] ValueType {} // dichiara e assegna una mappa vuota
- var mapName = map [KeyType] ValueType {key1: val1, key2: val2} // dichiara e assegna una mappa
- mapName: = make (map [KeyType] ValueType) // dichiara e inizializza la mappa delle dimensioni di default
- mapName: = make (map [KeyType] ValueType, length) // dichiara e inizializza la mappa delle dimensioni della *lunghezza*
- mapName: = map [KeyType] ValueType {} // auto-dichiara e assegna una mappa vuota con: =
- mapName: = map [KeyType] ValueType {key1: value1, key2: value2} // auto-dichiara e assegna una mappa con: =
- valore: = mapName [chiave] // Ottieni valore per chiave
- value, hasKey: = mapName [chiave] // Ottieni valore per chiave, 'hasKey' è 'true' se la chiave esiste nella mappa
- mapName [chiave] = valore // Imposta valore per chiave

Osservazioni

Go fornisce un tipo di `map` integrato che implementa una *tabella hash* . Le mappe sono tipi di dati associativi integrati di Go (chiamati anche *hash* o *dizionari* in altre lingue).

Examples

Dichiarazione e inizializzazione di una mappa

Definisci una mappa usando la `map` parole chiave, seguita dai tipi delle sue chiavi e dai suoi valori:

```
// 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
```

Le mappe sono tipi di riferimento e, una volta definiti, hanno un *valore zero pari a* `nil` . Scrive su zero le mappe andranno nel panico e le letture restituiranno sempre il valore zero.

Per inizializzare una mappa, usa la funzione `make` :

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

Con la forma a due parametri di `make` , è possibile specificare una capacità di immissione iniziale per la mappa, ignorando la capacità predefinita:

```
m := make(map[string]int, 30)
```

In alternativa, è possibile dichiarare una mappa, inizializzarla sul suo valore zero e quindi assegnarvi un valore letterale in seguito, il che aiuta se si effettua il marshalling della struct in json producendo quindi una mappa vuota al momento del ritorno.

```
m := make(map[string]int, 0)
```

Puoi anche creare una mappa e impostare il suo valore iniziale con parentesi graffe (`{}`).

```
var m map[string]int = map[string]int{"Foo": 20, "Bar": 30}

fmt.Println(m["Foo"]) // outputs 20
```

Tutte le seguenti istruzioni fanno sì che la variabile sia associata allo stesso valore.

```
// 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{}
```

Possiamo anche usare una *mappa letterale* per creare una nuova mappa con alcune coppie chiave / valore iniziali .

Il tipo di chiave può essere di qualsiasi tipo comparabile ; in particolare, questo esclude funzioni, mappe e sezioni . Il tipo di valore può essere di qualsiasi tipo, inclusi i tipi personalizzati o 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{}
```

Creare una mappa

Si può dichiarare e inizializzare una mappa in una singola istruzione usando un *letterale composito* .

Utilizzo del tipo automatico Breve dichiarazione variabile:

```
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"}
```

Lo stesso codice, ma con tipi Variabili:

```
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"}
```

Puoi anche includere le tue strutture in una mappa:

Puoi usare i tipi personalizzati come valore:

```
// 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"}}
```

La tua struttura può anche essere la *chiave* per la mappa:

```
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}
```

Puoi creare una mappa vuota semplicemente non inserendo alcun valore tra parentesi `{}` .

```
mapIntInt := map[int]int{}
mapIntString := map[int]string{}
mapStringInt := map[string]int{}
mapStringString := map[string]string{}
mapStringPerson := map[string]Person{}
```

È possibile creare e utilizzare direttamente una mappa, senza la necessità di assegnarla a una variabile. Tuttavia, dovrai specificare sia la dichiarazione che il contenuto.

```
// 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)
```

Valore zero di una mappa

Il valore zero di una `map` è `nil` e ha una lunghezza pari a `0` .

```
var m map[string]string
fmt.Println(m == nil) // true
fmt.Println(len(m) ==0) // true
```

Una mappa `nil` non ha chiavi e non è possibile aggiungere le chiavi. Una mappa `nil` si comporta come una mappa vuota se letta da ma provoca un panico di runtime se viene scritta.

```
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
```

Non dovresti provare a leggere o scrivere su una mappa a valore zero. Invece, inizializza la

mappa (con `make` o assignment) prima di usarla.

```
var m map[string]string
m = make(map[string]string) // OR m = map[string]string{}
m["foo"] = "bar"
```

Iterazione degli elementi di una mappa

```
import fmt

people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for key, value := range people {
  fmt.Println("Name:", key, "Age:", value)
}
```

Nota che durante l'iterazione su una mappa con un loop di intervallo, l'ordine di iterazione non è specificato e non è garantito che sia lo stesso da una iterazione alla successiva.

Puoi anche scartare le chiavi o i valori della mappa, se stai cercando di afferrare le chiavi o semplicemente di prendere i valori.

Iterazione delle chiavi di una mappa

```
people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for key, _ := range people {
  fmt.Println("Name:", key)
}
```

Se stai solo cercando le chiavi, dato che sono il primo valore, puoi semplicemente eliminare il carattere di sottolineatura:

```
for key := range people {
  fmt.Println("Name:", key)
}
```

Nota che durante l'iterazione su una mappa con un loop di intervallo, l'ordine di iterazione non è specificato e non è garantito che sia lo stesso da una iterazione alla successiva.

Eliminazione di un elemento della mappa

La funzione integrata di `delete` rimuove l'elemento con la chiave specificata da una mappa.

```
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]
```

Se la `map` è `nil` o non esiste alcun elemento, l' `delete` non ha alcun effetto.

```
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
```

Conteggio degli elementi della mappa

La funzione built-in `len` restituisce il numero di elementi in una `map`

```
m := map[string]int{}
len(m) // 0

m["foo"] = 1
len(m) // 1
```

Se una variabile punta a una mappa `nil` , allora `len` restituisce 0.

```
var m map[string]int
len(m) // 0
```

Accesso simultaneo di mappe

Le mappe in go non sono sicure per la concorrenza. È necessario prendere un lucchetto per leggere e scrivere su di essi se si accederà a loro in contemporanea. In genere, l'opzione migliore è utilizzare `sync.RWMutex` perché è possibile avere blocchi di lettura e scrittura. Tuttavia, è anche possibile utilizzare un `sync.Mutex` .

```
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")
}
```

Il compromesso tra le funzioni del wrapper è tra l'accesso pubblico della mappa sottostante e l'uso corretto dei blocchi appropriati.

Creazione di mappe con sezioni come valori

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

L'accesso a una chiave inesistente restituirà una fetta nil come valore. Poiché le porzioni nil si comportano come sezioni a lunghezza zero quando vengono utilizzate con `append` o altre funzioni integrate di solito non è necessario verificare se esiste una chiave:

```
// m["key1"] == nil && len(m["key1"]) == 0
m["key1"] = append(m["key1"], 1)
// len(m["key1"]) == 1
```

L'eliminazione di una chiave dalla mappa riporta la chiave su una fetta nullo:

```
delete(m, "key1")
// m["key1"] == nil
```

Controlla l'elemento in una mappa

Per ottenere un valore dalla mappa, devi solo fare qualcosa del tipo: 00

```
value := mapName[ key ]
```

Se la mappa contiene la chiave, restituisce il valore corrispondente.
In caso contrario, restituisce valore zero del tipo di valore della mappa (`0` se mappa di valori `int` , `""` se mappa di valori `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
```

Per distinguere tra valori vuoti e chiavi inesistenti, è possibile utilizzare il secondo valore restituito dell'accesso alla mappa (utilizzando il `value, hasKey := map["key"]` simile `value, hasKey := map["key"]`).

Questo secondo valore è `boolean` digitato e sarà:

- `true` quando il valore è nella mappa,
- `false` quando la mappa non contiene la chiave indicata.

Guarda il seguente esempio:

```
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
}
```

Iterazione dei valori di una mappa

```
people := map[string]int{
  "john": 30,
  "jane": 29,
  "mark": 11,
}

for _, value := range people {
  fmt.Println("Age:", value)
}
```

Nota che durante l'iterazione su una mappa con un loop di intervallo, l'ordine di iterazione non è specificato e non è garantito che sia lo stesso da una iterazione alla successiva.

Copia una mappa

Come le sezioni, le mappe contengono **riferimenti** a una struttura dati sottostante. Quindi assegnando il suo valore a un'altra variabile, verrà passato solo il riferimento. Per copiare la mappa, è necessario creare un'altra mappa e copiare ogni valore:

```
// 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
}
```

Usare una mappa come set

Alcune lingue hanno una struttura nativa per insiemi. Per creare un set in Go, è consigliabile utilizzare una mappa dal tipo di valore del set su una struttura vuota (`map[Type]struct{}`).

Ad esempio, con le stringhe:

```
// 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")
}
```

Leggi Mappe online: https://riptutorial.com/it/go/topic/732/mappe

Capitolo 44: metodi

Sintassi

- func (t T) exampleOne (i int) (n int) {return i} // questa funzione riceverà una copia di struct
- func (t * T) exampleTwo (i int) (n int) {return i} // questo metodo riceverà il puntatore a struct e sarà in grado di modificarlo

Examples

Metodi di base

I metodi in Go sono come le funzioni, tranne che hanno un *ricevitore* .

Di solito il ricevitore è una sorta di struttura o tipo.

```
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)

}
```

Produzione:

```
Bobs rank now is:  1
Lets promote Bob!
Now Bobs rank is:  2
```

Metodi di concatenamento

Con i metodi in golang puoi fare il metodo "concatenare" il puntatore che passa al metodo e restituire il puntatore alla stessa struttura in questo modo:

```
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)

}
```

Produzione:

```
Set name of new Employee to Bob
Promoting Bob
Here we have Bob with rank 2
```

Operatori di decremento dell'incremento come argomenti in Metodi

Sebbene Go supporti ++ e - operatori e il comportamento sia quasi simile a c / c ++, le variabili con tali operatori non possono essere passate come argomento per funzionare.

```
    package main

    import (
        "fmt"
```

```
)

func abcd(a int, b int) {
 fmt.Println(a," ",b)
}
func main() {
    a:=5
    abcd(a++,++a)
}
```

Output: errore di sintassi: inaspettato ++, in attesa di virgola o)

Leggi metodi online: https://riptutorial.com/it/go/topic/3890/metodi

Capitolo 45: MgO

introduzione

mgo (pronunciato come mango) è un driver MongoDB per il linguaggio Go che implementa una ricca e ben collaudata selezione di funzionalità sotto un'API molto semplice che segue gli idiomi Go standard.

Osservazioni

Documentazione API

[https://gopkg.in/mgo.v2][1]

Examples

Esempio

```
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)
}
```

Leggi MgO online: https://riptutorial.com/it/go/topic/8898/mgo

Capitolo 46: middleware

introduzione

In Go Middleware può essere utilizzato per eseguire il codice prima e dopo la funzione del gestore. Usa la potenza delle interfacce a singola funzione. Può essere introdotto in qualsiasi momento senza influire sugli altri middleware. Ad esempio: la registrazione dell'autenticazione può essere aggiunta nelle fasi successive dello sviluppo senza interferire con il codice esistente.

Osservazioni

La **firma del middleware** dovrebbe essere (http.ResponseWriter, * http.Request) cioè del tipo **http.handlerFunc** .

Examples

Funzione Handler normale

```
func loginHandler(w http.ResponseWriter, r *http.Request) {
            // Steps to login
}


func main() {
    http.HandleFunc("/login", loginHandler)
    http.ListenAndServe(":8080", nil)
}
```

Middleware Calcolare il tempo richiesto per l'esecuzione di handlerFunc

```
// 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
        }
    })
}
```

Recovery Handler per evitare il crash del server

```
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)
    })
}
```

Leggi middleware online: https://riptutorial.com/it/go/topic/9343/middleware

Capitolo 47: Migliori pratiche sulla struttura del progetto

Examples

Restfull Projects API with Gin

> Gin è una struttura web scritta in Golang. È dotato di un'API di tipo martini con prestazioni molto migliori, fino a 40 volte più veloci. Se hai bisogno di prestazioni e buona produttività, amerai il Gin.

Ci saranno 8 pacchetti + main.go

1. controllori
2. nucleo
3. libs
4. middleware
5. pubblico
6. router
7. Servizi
8. test
9. main.go

---

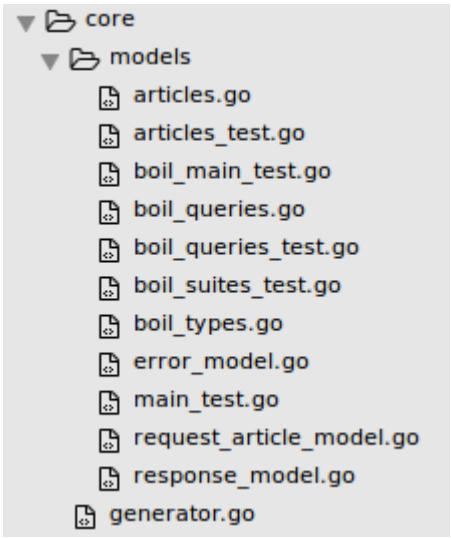
controllori

Il pacchetto Controller memorizzerà tutta la logica dell'API. Qualunque sia la tua API, la tua logica accadrà qui

---

nucleo

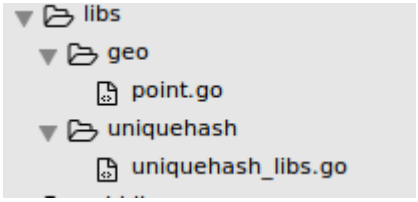
Il pacchetto principale memorizzerà tutti i modelli creati, ORM, ecc



---

libs

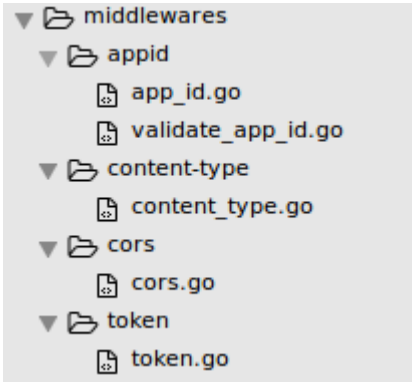
Questo pacchetto memorizzerà qualsiasi libreria utilizzata nei progetti. Ma solo per la libreria creata / importata manualmente, che non è disponibile quando si usa `go get package_name` comandi `go get package_name` . Potrebbe essere il tuo algoritmo di hashing, il grafico, l'albero ecc.



---

middleware

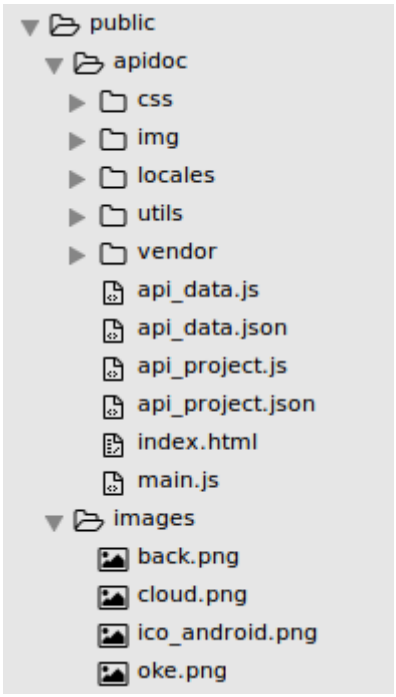
Questo pacchetto memorizza tutti i middleware utilizzati nel progetto, potrebbe essere la creazione / validazione di cors, id-dispositivo, auth ecc



---

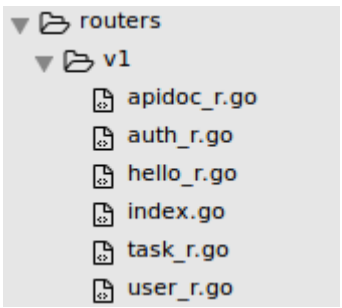
pubblico

Questo pacakge memorizzerà tutti i file pubblici e statici, potrebbe essere html, css, javascript, immagini, ecc



router

Questo pacchetto memorizzerà tutte le rotte nella tua API REST.



Vedi codice di esempio su come assegnare i percorsi.

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)
}
```

Se vedi, il motivo per cui separo il gestore è, per facilitare noi a gestire ogni router. Quindi posso creare commenti sull'API, che con apidoc lo genererà in una documentazione strutturata. Quindi chiamerò la funzione in index.go nel pacchetto corrente

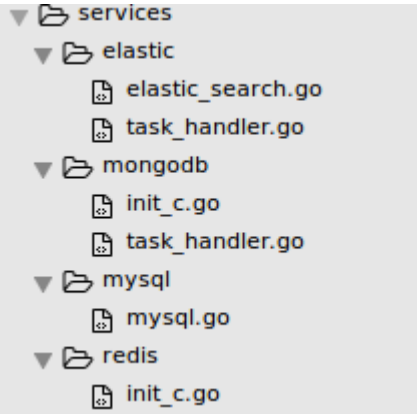
index.go

```
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)
}
```

Servizi

Questo pacchetto memorizzerà qualsiasi configurazione e impostazione da utilizzare nel progetto da qualsiasi servizio utilizzato, potrebbe essere mongodb, redis, mysql, elasticsearch, ecc.



---

main.go

L'ingresso principale dell'API. Qui verrà configurata qualsiasi configurazione relativa alle impostazioni dell'ambiente, ai sistemi, alla porta, ecc.

Esempio:
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: ogni codice in questo esempio proviene da diversi progetti

---

vedi esempi di progetti su github

Leggi Migliori pratiche sulla struttura del progetto online:
https://riptutorial.com/it/go/topic/9463/migliori-pratiche-sulla-struttura-del-progetto

Capitolo 48: Modelli

Sintassi

- t, err: = template.Parse (`{{.MyName .MyAge}}`)
- t.Execute (os.Stdout, struct {MyValue, MyAge string} {"John Doe", "40.1"})

Osservazioni

Golang offre pacchetti come:

1. `text/template`
2. `html/template`

implementare modelli basati sui dati per generare output testuali e HTML.

Examples

I valori di output della variabile struct su Output standard utilizzano un modello di testo

```
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)
    }
}
```

Terreno di gioco

Definizione delle funzioni per chiamare dal modello

```
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)
    }
}
```

Qui usiamo la nostra funzione `status_code` definita per ottenere il codice di stato della pagina web direttamente dal modello.

Produzione:

```
http://godoc.org 200

http://stackoverflow.com 200

http://linux.org 200
```

Leggi Modelli online: https://riptutorial.com/it/go/topic/1402/modelli

Capitolo 49: mutex

Examples

Mutex Locking

Il blocco di Mutex in Go ti consente di assicurarti che solo una goroutine alla volta abbia un lucchetto:

```
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'utilizzo di un `Mutex` consente di evitare condizioni di competizione, modifiche simultanee e altri problemi associati a più routine simultanee che operano sulle stesse risorse. Si noti che `Mutex.Unlock()` può essere eseguito da qualsiasi routine, non solo dalla routine che ha ottenuto il blocco. Si noti inoltre che la chiamata a `Mutex.Lock()` non fallirà se un'altra routine mantiene il blocco; bloccherà fino a quando il blocco non verrà rilasciato.

Suggerimento: ogni volta che si passa una variabile Mutex a una funzione, passarla sempre come puntatore. Altrimenti viene fatta una copia della variabile, che sconfigge lo scopo del Mutex. Se stai usando una versione Go precedente (<1.7), il compilatore non ti avviserà di questo errore!

Leggi mutex online: https://riptutorial.com/it/go/topic/2607/mutex

Capitolo 50: Pacchi

Examples

Initalizzazione del pacchetto

Il pacchetto può avere metodi `init` che vengono eseguiti **solo una volta** prima di main.

```
package usefull

func init() {
    // init code
}
```

Se si desidera eseguire l'inizializzazione del pacchetto senza fare riferimento a nulla, utilizzare la seguente espressione di importazione.

```
import _ "usefull"
```

Gestire le dipendenze del pacchetto

Un metodo comune per scaricare le dipendenze di Go è utilizzando il comando `go get <package>` , che salverà il pacchetto nella directory globale / condivisa `$GOPATH/src` . Ciò significa che una singola versione di ciascun pacchetto sarà collegata a ciascun progetto che la include come dipendenza. Ciò significa anche che quando un nuovo sviluppatore distribuisce il tuo progetto, `go get` l'ultima versione di ciascuna dipendenza.

Tuttavia è possibile mantenere coerente l'ambiente di costruzione, collegando tutte le dipendenze di un progetto nella directory del `vendor/` . Mantenere le dipendenze vendute e il repository del progetto consente di eseguire il controllo delle versioni per progetto e fornire un ambiente coerente per la build.

Ecco come sarà la struttura del tuo progetto:

```
$GOPATH/src/
├── github.com/username/project/
|   ├── main.go
|   ├── vendor/
|   |   ├── github.com/pkg/errors
|   |   ├── github.com/gorilla/mux
```

Usando un diverso nome di pacchetto e cartella

Va perfettamente bene usare un nome di pacchetto diverso dal nome della cartella. Se lo facciamo, dobbiamo ancora importare il pacchetto in base alla struttura della directory, ma dopo l'importazione dobbiamo farvi riferimento con il nome che abbiamo usato nella clausola del pacchetto.

Ad esempio, se hai una cartella `$GOPATH/src/mypck` , e in essa abbiamo un file `a.go` :

```
package apple

const Pi = 3.14
```

Utilizzando questo pacchetto:

```
package main

import (
    "mypck"
    "fmt"
)

func main() {
    fmt.Println(apple.Pi)
}
```

Anche se questo funziona, dovresti avere una buona ragione per deviare il nome del pacchetto dal nome della cartella (o potrebbe diventare fonte di incomprensioni e confusione).

A cosa serve questo?

Semplice. Un nome pacchetto è un identificatore di Go:

```
identifier = letter { letter | unicode_digit } .
```

Che consente di utilizzare lettere unicode negli identificatori, ad esempio `αβ` è un identificatore valido in Go. I nomi delle cartelle e dei file non sono gestiti da Go ma dal sistema operativo e diversi sistemi di file hanno restrizioni diverse. Esistono in realtà molti file system che non consentirebbero tutti gli identificatori di Go validi come nomi di cartelle, quindi non si sarebbe in grado di assegnare un nome ai pacchetti che altrimenti le specifiche della lingua consentirebbero.

Avendo la possibilità di utilizzare diversi nomi di pacchetti rispetto alle loro cartelle di contenimento, hai la possibilità di nominare realmente i tuoi pacchetti che cosa consente la specifica della lingua, indipendentemente dal sistema operativo e file sottostante.

Importazione di pacchetti

Puoi importare un singolo pacchetto con l'istruzione:

```
import "path/to/package"
```

o raggruppare più importazioni insieme:

```
import (
    "path/to/package1"
    "path/to/package2"
)
```

Verrà visualizzato nei percorsi di `import` corrispondenti all'interno di `$GOPATH` per i file `.go` e consente di accedere ai nomi esportati tramite `packagename.AnyExportedName` .

Puoi anche accedere ai pacchetti locali all'interno della cartella corrente facendo precedere i pacchetti con `./` . In un progetto con una struttura come questa:

```
project
├── src
│   ├── package1
│   │   └── file1.go
│   └── package2
│       └── file2.go
└── main.go
```

Puoi chiamarlo in `main.go` per importare il codice in `file1.go` e `file2.go` :

```
import (
    "./src/package1"
    "./src/package2"
)
```

Dato che i nomi dei pacchetti possono collidere in diverse librerie, è possibile che si desideri creare un alias di un pacchetto con un nuovo nome. Puoi farlo anteponendo la tua dichiarazione di importazione con il nome che desideri utilizzare.

```
import (
    "fmt" //fmt from the standardlibrary
    tfmt "some/thirdparty/fmt" //fmt from some other library
)
```

Questo ti permette di accedere al precedente pacchetto `fmt` usando `fmt.*` E il secondo pacchetto `fmt` usando `tfmt.*` .

È inoltre possibile importare il pacchetto nel proprio spazio dei nomi, in modo da poter fare riferimento ai nomi esportati senza il `package.` prefisso utilizzando un singolo punto come alias:

```
import (
    . "fmt"
)
```

L'esempio precedente importa `fmt` nello spazio dei nomi globale e consente di chiamare, ad esempio, direttamente `Printf` : Playground

Se importi un pacchetto ma non usi nessuno dei suoi nomi esportati, il compilatore Go stamperà un messaggio di errore. Per aggirare questo problema, puoi impostare l'alias sul carattere di sottolineatura:

```
import (
    _ "fmt"
)
```

Questo può essere utile se non si accede direttamente a questo pacchetto, ma è necessario eseguire le sue funzioni `init` .

Nota:

Poiché i nomi dei pacchetti si basano sulla struttura delle cartelle, qualsiasi modifica nei nomi delle cartelle e dei riferimenti di importazione (inclusa la distinzione tra maiuscole e minuscole) causerà un errore di compilazione "collisione di importazione senza distinzione tra maiuscole e minuscole" in Linux e OS-X, che è difficile da tracciare e aggiusta (il messaggio di errore è piuttosto criptico per i comuni mortali poiché cerca di esprimere il contrario - che, il confronto non è riuscito a causa della distinzione tra maiuscole e minuscole).

ex: "percorso / su / Pacchetto1" vs "percorso / su / pacchetto1"

Esempio dal vivo: https://github.com/akamai-open/AkamaiOPEN-edgegrid-golang/issues/2

Leggi Pacchi online: https://riptutorial.com/it/go/topic/401/pacchi

Capitolo 51: Panico e Recupero

Osservazioni

Questo articolo presuppone la conoscenza di Defer Basics

Per la normale gestione degli errori, leggere l' argomento sulla gestione degli errori

Examples

Panico

Un panico blocca il normale flusso di esecuzione ed esce dalla funzione corrente. Qualsiasi chiamata differita verrà quindi eseguita prima che il controllo venga passato alla successiva funzione superiore nello stack. Ogni funzione dello stack verrà chiusa ed eseguirà le chiamate differite fino a quando il panico verrà gestito utilizzando un `recover()` differito `recover()` , o fino a quando il panico raggiunge `main()` e termina il programma. Se ciò si verifica, l'argomento fornito per il panico e una traccia dello stack verranno stampati su `stderr` .

```
package main

import "fmt"

func foo() {
    defer fmt.Println("Exiting foo")
    panic("bar")
}

func main() {
    defer fmt.Println("Exiting main")
    foo()
}
```

Produzione:

```
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
```

È importante notare che il `panic` accetterà qualsiasi tipo come parametro.

Recuperare

Recuperare come suggerisce il nome, può tentare di riprendersi dal `panic` . Il recupero *deve* essere tentato in una dichiarazione posticipata poiché il normale flusso di esecuzione è stato interrotto. La dichiarazione di `recover` deve apparire *direttamente* all'interno del recinto della funzione posticipata. Recuperare le affermazioni in funzioni chiamate da chiamate con funzioni differite non sarà rispettato. La chiamata a `recover()` restituirà l'argomento fornito al panico iniziale, se il programma è attualmente in panico. Se il programma non è attualmente in panico, `recover()` restituirà `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")
}
```

Produzione:

```
Entering main
Recovered with message bar
Exiting main the normal way
```

Leggi Panico e Recupero online: https://riptutorial.com/it/go/topic/4350/panico-e-recupero

Capitolo 52: Piscine di lavoratori

Examples

Piscina semplice lavoratore

Una semplice implementazione del pool di lavoro:

```
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)
    }
}
```

Job Queue con Worker Pool

Una coda di lavoro che mantiene un pool di worker, utile per fare cose come l'elaborazione in background nei server 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)})
  }
}
```

Leggi Piscine di lavoratori online: https://riptutorial.com/it/go/topic/4182/piscine-di-lavoratori

Capitolo 53: Pooling di memoria

introduzione

sync.Pool memorizza una cache di elementi allocati ma non utilizzati per uso futuro, evitando la perdita di memoria per le collezioni modificate frequentemente e consentendo un riutilizzo efficiente della memoria in modo thread-safe. È utile gestire un gruppo di elementi temporanei condivisi tra client concorrenti di un pacchetto, ad esempio un elenco di connessioni di database o un elenco di buffer di output.

Examples

sync.Pool

Utilizzando la struttura `sync.Pool` possiamo riunire oggetti e riutilizzarli.

```
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)
}
```

Leggi Pooling di memoria online: https://riptutorial.com/it/go/topic/4647/pooling-di-memoria

Capitolo 54: Profilazione usando go tool pprof

Osservazioni

Per ulteriori informazioni sui programmi go visitare il blog go .

Examples

CPU di base e profilo di memoria

Aggiungi il seguente codice nel tuo programma principale.

```
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()
    }
}
```

dopo di ciò **costruisci** il programma go se aggiunto in main `go build main.go` Eseguire il programma principale con flag definiti nel codice `main.exe -cpuprofile cpu.prof -memprof mem.prof` . Se la profilazione viene eseguita per i casi di test, eseguire i test con gli stessi flag `go test -cpuprofile cpu.prof -memprofile mem.prof`

Profiling di memoria di 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
```

Imposta la velocità del profilo CPU / blocco

```
// 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)
```

Utilizzo dei benchmark per creare un profilo

Per i pacchetti non principali e principali, **invece di aggiungere i flag all'interno del codice** , scrivere **benchmark** nel pacchetto di test, ad esempio:

```
func BenchmarkHello(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fmt.Sprintf("hello")
    }
}
```

Quindi esegui il test con il flag del profilo

> vai test -cpuprofile cpu.prof -bench =.

E i benchmark verranno eseguiti e creare un file prof con nome file cpu.prof (nell'esempio precedente).

Accesso al file di profilo

una volta generato un file prof, è possibile accedere al file prof usando gli **strumenti go** :

    go tool pprof cpu.prof

Questo entrerà in un'interfaccia a riga di comando per esplorare il `profile`

I comandi più comuni includono:

```
(pprof) top
```

elenca i processi più importanti in memoria

```
(pprof) peek
```

Elenca tutti i processi, usa *regex* per restringere la ricerca.

```
(pprof) web
```

Apre un grafico (in formato svg) del processo.

Un esempio del comando in `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
```

Leggi Profilazione usando go tool pprof online: https://riptutorial.com/it/go/topic/7748/profilazione-usando-go-tool-pprof

Capitolo 55: Programmazione orientata agli oggetti

Osservazioni

L'interfaccia non può essere implementata con i ricevitori puntatore perché `*User` non è `User`

Examples

Structs

Go supporta i tipi definiti dall'utente sotto forma di strutture e digita alias. le strutture sono tipi compositi, le parti componenti dei dati che costituiscono il tipo di struttura sono chiamate *campi* . un campo ha un tipo e un nome che deve essere 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}
}
```

questa è anche una sintassi legale per definire le strutture

```
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"
```

Strutture incorporate

poiché una struct è anche un tipo di dati, può essere usata come un campo anonimo, la struct

esterna può accedere direttamente ai campi della struct incorporata anche se la struct proviene da un pacchetto diffrent. questo comportamento fornisce un modo per ricavare parte o tutta l'implementazione da un altro tipo o da un insieme di tipi.

```
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
}
```

metodi

In Go a method is

> una funzione che agisce su una variabile di un certo tipo, chiamata ricevitore

il ricevitore può essere qualsiasi cosa, non solo `structs` ma anche una `function` , i tipi di alias per i tipi built-in come `int` , `string` , `bool` possono avere un metodo, un'eccezione a questa regola è che le `interfaces` (discusse più tardi) non possono avere metodi, poiché l'interfaccia è una definizione astratta e un metodo è un'implementazione, provando a generare un errore di compilazione.

combinando le `structs` e i `methods` è possibile ottenere un equivalente eqivalente di una `class` nella programmazione orientata agli oggetti.

un metodo in Go ha la seguente firma

```
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
}
```

Pointer Vs Value receiver

il ricevitore di un metodo è di solito un puntatore per il motivo delle prestazioni perché non faremo una copia dell'istanza, come nel caso del ricevitore di valore, ciò è particolarmente vero se il tipo di destinatario è una struttura. un motivo in più per fare in modo che il ricevitore digiti un puntatore sarebbe in modo da poter modificare i dati a cui punta il ricevitore.

un ricevitore di valore viene utilizzato per evitare la modifica dei dati contenuti nel ricevitore, un ricevitore di vaule può causare un colpo di prestazioni se il ricevitore è una struttura di grandi dimensioni.

```
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
}
```

Interfaccia e polimorfismo

Le interfacce forniscono un modo per specificare il comportamento di un oggetto, se qualcosa può farlo, allora può essere usato qui. un'interfaccia definisce un insieme di metodi, ma questi metodi non contengono codice in quanto sono astratti o l'implementazione è lasciata all'utente dell'interfaccia. a differenza della maggior parte delle interfacce linguistiche orientate agli oggetti possono contenere variabili in Go.

Il polimorfismo è l'essenza della programmazione orientata agli oggetti: la capacità di trattare oggetti di tipi diversi in modo uniforme purché aderiscano alla stessa interfaccia. Le interfacce Go forniscono questa funzionalità in modo molto diretto e intuitivo

```
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)

}
```

Leggi Programmazione orientata agli oggetti online:
https://riptutorial.com/it/go/topic/8801/programmazione-orientata-agli-oggetti

Capitolo 56: Protobuf in Go

introduzione

Protobuf o Protocol Buffer codifica e decodifica i dati in modo che diverse applicazioni o moduli scritti in lingue diverse possano scambiare il gran numero di messaggi in modo rapido e affidabile senza sovraccaricare il canale di comunicazione. Con protobuf, la performance è direttamente proporzionale al numero di messaggi che si tende ad inviare. Comprime il messaggio per inviare un formato binario serializzato fornendo gli strumenti per codificare il messaggio all'origine e decodificarlo alla destinazione.

Osservazioni

Ci sono due passaggi per usare **protobuf** .

1. Per prima cosa devi compilare le definizioni del buffer del protocollo
2. Importa le definizioni di cui sopra, con la libreria di supporto nel tuo programma.

Supporto gRPC

Se un file di proto specifica i servizi RPC, è possibile istruire protoc-gen-go per generare codice compatibile con gRPC (http://www.grpc.io/) . Per fare ciò, passa il parametro `plugins` a protoc-gen-go; il solito modo è di inserirlo nell'argomento --go_out per proteggere:

```
protoc --go_out=plugins=grpc:. *.proto
```

Examples

Usare Protobuf con Go

Il messaggio che si desidera serializzare e inviare che è possibile includere in un file **test.proto** , contenente

```
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;
  }
}
```

Per compilare la definizione del buffer del protocollo, eseguire protoc con il parametro --go_out

impostato sulla directory a cui si desidera inviare il codice Go.

```
protoc --go_out=. *.proto
```

Per creare e giocare con un oggetto Test dal pacchetto di esempio,

```
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.
}
```

Per passare parametri extra al plug-in, utilizzare un elenco di parametri separati da virgole separati dalla directory di output di due punti:

```
protoc --go_out=plugins=grpc,import_path=mypackage:. *.proto
```

Leggi Protobuf in Go online: https://riptutorial.com/it/go/topic/9729/protobuf-in-go

Capitolo 57: puntatori

Sintassi

- pointer: = & variable // ottiene il puntatore dalla variabile
- variabile: = * puntatore // ottiene la variabile dal puntatore
- * pointer = value // imposta il valore dalla variabile attraverso il puntatore
- pointer: = new (Struct) // ottiene il puntatore della nuova struttura

Examples

Puntatori di base

Go supporta i puntatori , consentendo di passare riferimenti a valori e record all'interno del programma.

```
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
}
```

Una volta definite queste funzioni, puoi fare quanto segue:

```
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
}
```

Prova questo codice

Puntatore v. Metodi di valore

Metodi di puntamento

I metodi di puntatore possono essere chiamati anche se la variabile non è essa stessa un puntatore.

Secondo la specifica Go ,

> . . . un riferimento a un metodo non-interface con un ricevitore puntatore che usa un valore indirizzabile prenderà automaticamente l'indirizzo di quel valore: `t.Mp` è equivalente a `(&t).Mp` .

Puoi vedere questo in questo esempio:

```
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)
}
```

Gioca

Metodi di valore

Analogamente ai metodi puntatori, i metodi di valore possono essere chiamati anche se la variabile non è essa stessa un valore.

Secondo la specifica Go ,

> . . . un riferimento a un metodo non-interface con un ricevitore di valore che utilizza un puntatore automaticamente `pt.Mv` quel puntatore: `pt.Mv` è equivalente a `(*pt).Mv` .

Puoi vedere questo in questo esempio:

```
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)
}
```

Gioca

Per ulteriori informazioni sui metodi di puntatore e valore, visitare la sezione Vai specifiche su Valori metodo o consultare la sezione Effettiva Vai su Puntatori v. Valori .

Nota 1: La parentesi (`()`) attorno a `*p` e `&f` prima di selettori come `.Bar` sono lì per scopi di raggruppamento e devono essere mantenuti.

Nota 2: Sebbene i puntatori possano essere convertiti in valori (e viceversa) quando sono i ricevitori di un metodo, non vengono convertiti automaticamente a vicenda quando sono

argomenti all'interno di una funzione.

Puntatori di dereferenziamento

I puntatori possono essere **dereferenziati** aggiungendo un asterisco * prima di un puntatore.

```
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
}
```

Le fette sono puntatori ai segmenti dell'array

Le slice sono **puntatori** agli array, con la lunghezza del segmento e la sua capacità. Si comportano come puntatori e assegnando il loro valore a un'altra fetta assegneranno l'indirizzo di memoria. Per **copiare** un valore di slice su un altro, utilizzare la funzione di **copia** incorporata: `func copy(dst, src []Type) int` (restituisce la quantità di elementi copiati).

```
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
}
```

Puntatori semplici

```
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)
}
```

Leggi puntatori online: https://riptutorial.com/it/go/topic/1239/puntatori

Capitolo 58: Registrazione

Examples

Stampa di base

Go ha una libreria di registrazione incorporata nota come `log` con un metodo di `Print` comunemente usato e le sue varianti. È possibile importare la libreria, quindi eseguire alcune operazioni di stampa di 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
}
```

Registrazione su file

È possibile specificare la destinazione del registro con qualcosa che soddisfi l'interfaccia io.Writer. Con ciò possiamo accedere al file:

```
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")
}
```

Produzione:

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

Registrazione su syslog

È anche possibile accedere a syslog con `log/syslog` come questo:

```
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")
}
```

Dopo l'esecuzione potremo vedere quella riga in syslog:

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

Leggi Registrazione online: https://riptutorial.com/it/go/topic/3724/registrazione

Capitolo 59: Riflessione

Osservazioni

I documenti di `reflect` sono un ottimo riferimento. Nella programmazione generale del computer, la **riflessione** è la capacità di un programma di **esaminare** la struttura e il comportamento di **se stesso** in **`runtime`** di **`runtime`** .

Basato sul suo sistema di `static type` rigoroso, Go lang ha alcune regole (leggi di riflessione)

Examples

Base reflect.Value Usage

```
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"
```

fette

```
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
```

reflect.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
```

Tipo di valore - il pacchetto "riflette"

reflect.TypeOf può essere utilizzato per verificare il tipo di variabili durante il confronto

```
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))

}
```

Produzione :
stringa
main.Data

Leggi Riflessione online: https://riptutorial.com/it/go/topic/1854/riflessione

Capitolo 60: Segnali OS

Sintassi

- func Notifica (c chan <- os.Signal, sig ... os.Signal)

Parametri

Parametro	Dettagli
c chan <- os.Signal	Ricevere il `channel` specificatamente di tipo `os.Signal` ; facilmente creato con `sigChan := make(chan os.Signal)`
sig ... os.Signal	Elenco di tipi `os.Signal` per catturare e inviare questo `channel` . Vedi https://golang.org/pkg/syscall/#pkg-constants per ulteriori opzioni.

Examples

Assegnazione di segnali a un canale

Spesso si avrà motivo di prendere quando al programma viene richiesto di fermarsi dal sistema operativo e intraprendere alcune azioni per preservare lo stato o pulire l'applicazione. Per fare ciò è possibile utilizzare il pacchetto `os/signal` dalla libreria standard. Di seguito è riportato un semplice esempio di assegnazione di tutti i segnali dal sistema a un canale, e quindi come reagire a tali segnali.

```
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)
    }
}
```

Quando si esegue lo script precedente, verrà creato un canale e quindi bloccato fino a quando quel canale non riceve un segnale.

```
$ go run signals.go
^CReceived signal from OS: interrupt
```

Il `^C` sopra è il comando da tastiera `CTRL+C` che invia il segnale `SIGINT` .

Leggi Segnali OS online: https://riptutorial.com/it/go/topic/4497/segnali-os

Capitolo 61: Seleziona e Canali

introduzione

La parola chiave `select` fornisce un metodo semplice per lavorare con i canali ed eseguire attività più avanzate. Viene spesso utilizzato per diversi scopi: - Gestione dei timeout. - Quando ci sono più canali da cui leggere, la selezione leggerà a caso da un canale che ha dati. - Fornire un modo semplice per definire cosa succede se non ci sono dati disponibili su un canale.

Sintassi

- seleziona {}
- seleziona {caso vero:}
- seleziona {case incomingData: = <-someChannel:}
- seleziona {predefinito:}

Examples

Semplice Seleziona Lavorare con i canali

In questo esempio creiamo una goroutine (una funzione che gira in un thread separato) che accetta un parametro `chan` e semplicemente loop, inviando ogni volta informazioni nel canale.

Nel `main` abbiamo un ciclo `for` e una `select` . La `select` bloccherà l'elaborazione fino a quando una delle affermazioni `case` diventerà vera. Qui abbiamo dichiarato due casi; il primo è quando le informazioni arrivano attraverso il canale, e l'altra è se non si verificano altri casi, che è noto come `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)
        }
    }
}
```

Provalo sul Go Playground!

Usando select con timeouts

Quindi, qui ho rimosso i cicli `for` , e ho fatto un **timeout** aggiungendo un secondo `case` alla `select` che ritorna dopo 3 secondi. Poiché la `select` aspetta solo che ANY case sia vero, il secondo `case` attiva, e quindi il nostro script termina e `chatter()` non ha nemmeno la possibilità di finire.

```
// 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!")
    }
}
```

Leggi Seleziona e Canali online: https://riptutorial.com/it/go/topic/3539/seleziona-e-canali

Capitolo 62: Server HTTP

Osservazioni

`http.ServeMux` fornisce un multiplexer che chiama i gestori per le richieste HTTP.

Le alternative al multiplexer di libreria standard includono:

- Gorilla Mux

Examples

HTTP Hello World con server personalizzato e 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())
}
```

Premi `Ctrl` + `C` per interrompere il processo.

Ciao mondo

Il modo tipico per iniziare a scrivere server web in golang è usare il modulo standard `net/http` libreria.

C'è anche un tutorial per questo qui .

Anche il seguente codice lo usa. Ecco l'implementazione del server HTTP più semplice possibile.

Risponde `"Hello World"` a qualsiasi richiesta HTTP.

Salva il seguente codice in un file `server.go` nelle tue aree di lavoro.

```
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))
}
```

È possibile eseguire il server utilizzando:

```
$ go run server.go
```

O puoi compilare ed eseguire.

```
$ go build server.go
$ ./server
```

Il server ascolterà la porta specificata (`:8080`). Puoi testarlo con qualsiasi client HTTP. Ecco un esempio con `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!
```

Premi `Ctrl` + `C` per interrompere il processo.

Utilizzando una funzione di gestore

`HandleFunc` registra la funzione di gestione per il modello specificato nel server mux (router).

Puoi passare definire una funzione anonima, come abbiamo visto nell'esempio base di *Hello World* :

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, world!")
```

```
}
```

Ma possiamo anche passare un tipo `HandlerFunc` . In altre parole, possiamo passare qualsiasi funzione che rispetti la seguente firma:

```
func FunctionName(w http.ResponseWriter, req *http.Request)
```

Possiamo riscrivere l'esempio precedente passando il riferimento a un `HandlerFunc` precedentemente definito. Ecco l'esempio completo:

```
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))
}
```

Naturalmente, è possibile definire diversi gestori di funzioni per percorsi diversi.

```
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))
}
```

Ecco l'output usando `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
```

Crea un server HTTPS

Genera un certificato

Per eseguire un server HTTPS, è necessario un certificato. La generazione di un certificato autofirmato con `openssl` viene eseguita eseguendo questo comando:

```
openssl req -x509 -newkey rsa:4096 -sha256 -nodes -keyout key.pem -out cert.pem -subj
"/CN=example.com" -days 3650`
```

I parametri sono:

- `req` Usa lo strumento di richiesta del certificato
- `x509` Crea un certificato autofirmato
- `newkey rsa:4096` Crea una nuova chiave e certificato utilizzando gli algoritmi RSA con una lunghezza della chiave di `4096` bit
- `sha256` Forza gli algoritmi di hashing SHA256 che i principali browser considerano sicuri (nell'anno 2017)
- `nodes` Disabilita la protezione con password per la chiave privata. Senza questo parametro, il server ha dovuto chiederti la password ogni volta che si avvia.
- `keyout` Nomi il file in cui scrivere la chiave
- `out` Nomi il file in cui scrivere il certificato
- `subj` Definisce il nome di dominio per il quale questo certificato è valido
- `days` Fow quanti giorni dovrebbe essere valido questo certificato? `3650` sono ca. 10 anni.

Nota: è possibile utilizzare un certificato autofirmato, ad esempio per progetti interni, debug, test,

ecc. Qualsiasi browser esterno menzionerà che questo certificato non è sicuro. Per evitare ciò, il certificato deve essere firmato da un'autorità di certificazione. Per lo più, questo non è disponibile gratuitamente. Un'eccezione è il movimento "Let's Encrypt": https://letsencrypt.org

Il codice Go necessario

È possibile gestire configurare TLS per il server con il seguente codice. `cert.pem` e `key.pem` sono il certificato e la chiave SSL, che sono stati generati con il comando precedente.

```
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))
}
```

Risposta a una richiesta HTTP utilizzando i modelli

Le risposte possono essere scritte su un `http.ResponseWriter` utilizzando i modelli in Go. Questo si rivela uno strumento utile se desideri creare pagine dinamiche.

(Per sapere come funzionano i modelli in Go, visita la pagina Documentazione di Go Templates .)

Continuando con un semplice esempio per utilizzare il `html/template` per rispondere a una richiesta 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)
    }
}
```

Dove, il contenuto di

1. `welcomeform.html` sono:

```
<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` sono:

```
<head>
    <title> Greetings, {{.Name}} </title>
</head>
<body>
    Greetings, {{.Name}}.<br>
    We know you are a {{.nationality}}!
</body>
```

Nota:

1. Assicurati che i file `.html` siano nella directory corretta.
2. Quando `http://localhost:8080/` può essere visitato dopo l'avvio del server.
3. Come si può vedere dopo aver inviato il modulo, il campo della nazionalità non *esportato* della struct non può essere analizzato dal pacchetto di template, come previsto.

Fornire contenuti utilizzando ServeMux

Un semplice file server statico sarebbe simile a questo:

```
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)
}
```

Gestire il metodo http, accedere alle stringhe di query e al corpo della richiesta

Ecco un semplice esempio di alcune attività comuni relative allo sviluppo di un'API, che distinguono tra il metodo HTTP della richiesta, l'accesso ai valori della stringa di query e l'accesso al corpo della richiesta.

risorse

- interfaccia http.Handler
- http.ResponseWriter
- http.Request
- Metodo disponibile e costanti di stato

```
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))
}
```

Risultato di arricciatura del campione:

```
$ 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
```

Leggi Server HTTP online: https://riptutorial.com/it/go/topic/756/server-http

Capitolo 63: sputo

introduzione

Gob è un metodo di serializzazione specifico Go. ha supporto per tutti i tipi di dati Go tranne canali e funzioni. Gob codifica anche le informazioni sul tipo nella forma serializzata, ciò che lo rende diverso dall'esempio XML è che è molto più efficiente.

L'inclusione delle informazioni sul tipo rende la codifica e la decodifica abbastanza affidabili rispetto alle differenze tra encoder e decodificatore.

Examples

Come codificare i dati e scrivere su file con 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)

}
```

Come leggere i dati dal file e decodificarli con 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)

}
```

Come codificare un'interfaccia con 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)
    }

}
```

Come decodificare un'interfaccia con 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

}
```

Leggi sputo online: https://riptutorial.com/it/go/topic/8820/sputo

Capitolo 64: SQL

Osservazioni

Per un elenco dei driver del database SQL, consultare l'articolo Go wiki ufficiale *SQLDrivers* .

I driver SQL vengono importati e preceduti da _ , in modo che siano disponibili *solo* per il driver.

Examples

Interrogazione

Questo esempio mostra come interrogare un database con `database/sql` , prendendo come esempio un database 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

Per abilitare MySQL, è necessario un driver di database. Ad esempio github.com/go-sql-

driver/mysql .

```
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

Aprire un database

L'apertura di un database è specifica del database, qui ci sono esempi per alcuni database.

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: connetti e inserisci e rimuovi e aggiorna e richiedi

```
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
}
```

Leggi SQL online: https://riptutorial.com/it/go/topic/1273/sql

Capitolo 65: Stringa

introduzione

Una stringa è in effetti una porzione di byte di sola lettura. In Go a string letteral conterrà sempre una rappresentazione UTF-8 valida del suo contenuto.

Sintassi

- variableName: = "Hello World" // dichiara una stringa
- variableName: = `Hello World` // dichiara una stringa letterale non elaborata
- variableName: = "Hello" + "World" // concatena le stringhe
- sottostringa: = "Hello World" [0: 4] // ottiene una parte della stringa
- letter: = "Hello World" [6] // ottiene un carattere della stringa
- fmt.Sprintf ("% s", "Hello World") // formatta una stringa

Examples

Tipo di stringa

Il tipo di `string` consente di memorizzare il testo, che è una serie di caratteri. Esistono diversi modi per creare stringhe. Una stringa letterale viene creata scrivendo il testo tra virgolette.

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

Poiché le stringhe Go supportano UTF-8, l'esempio precedente è perfettamente valido. Le stringhe contengono byte arbitrari che non significano necessariamente che ogni stringa conterrà UTF-8 valido, ma i valori letterali stringa avranno sempre sequenze UTF-8 valide.

Il valore zero delle stringhe è una stringa vuota `""` .

Le stringhe possono essere concatenate usando l'operatore `+` .

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

Le stringhe possono anche essere definite usando i backtick `` ` `` . Questo crea un letterale stringa grezzo che significa che i caratteri non saranno sfuggiti.

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

Nell'esempio precedente, `text1` sfugge al carattere `\n` che rappresenta una nuova riga mentre `text2` contiene direttamente il carattere della nuova riga. Se confronti `text1 == text2` il risultato sarà `true` .

Tuttavia, `text2 := `Hello\nWorld`` non sfuggirebbe al carattere `\n` , il che significa che la stringa contiene il testo `Hello\nWorld` senza una nuova riga. Sarebbe l'equivalente di digitare `text1 := "Hello\\nWorld"` .

Formattazione del testo

`fmt` funzioni di `fmt` implementa per stampare e formattare il testo usando i *verbi di* formato. I verbi sono rappresentati con un segno di percentuale.

Verbi generali:

```
%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
```

booleano:

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

Numero intero:

```
%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"
```

Costituenti a virgola mobile e complessi:

```
%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
```

Stringa e fetta di byte (trattate in modo equivalente con questi verbi):

```
%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
```

Pointer:

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

Utilizzando i verbi, puoi creare stringhe concatenando più tipi:

```
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 funzione `Sprintf` formatta la stringa nel primo parametro sostituendo i verbi con il valore dei valori nei parametri successivi e restituisce il risultato. Come `Sprintf` , anche la funzione `Printf` formatta, ma invece di restituire il risultato, stampa la stringa.

pacchetto di stringhe

- 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
```

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

Leggi Stringa online: https://riptutorial.com/it/go/topic/9666/stringa

Capitolo 66: Structs

introduzione

Le strutture sono insiemi di varie variabili imballate insieme. La struct stessa è solo un *pacchetto* contenente variabili e rendendole facilmente accessibili.

A differenza di C, le strutture di Go possono avere metodi collegati a loro. Permette anche loro di implementare interfacce. Ciò rende le strutture di Go simili agli oggetti, ma sono (probabilmente intenzionalmente) mancanti di alcune delle principali funzionalità conosciute nei linguaggi orientati agli oggetti come l'ereditarietà.

Examples

Dichiarazione di base

Una struttura di base è dichiarata come segue:

```
type User struct {
    FirstName, LastName string
    Email               string
    Age                 int
}
```

Ogni valore è chiamato un campo. I campi sono solitamente scritti uno per riga, con il nome del campo che precede il suo tipo. Campi consecutivi dello stesso tipo possono essere combinati, come `FirstName` e `LastName` nell'esempio precedente.

Campi Esportati vs. Non Esportati (Privati vs Pubblico)

Vengono esportati i campi Struct i cui nomi iniziano con una lettera maiuscola. Tutti gli altri nomi non sono stati esportati.

```
type Account struct {
    UserID      int    // exported
    accessToken string // unexported
}
```

I campi non esportati possono essere raggiunti solo dal codice all'interno dello stesso pacchetto. Pertanto, se si accede sempre a un campo da un pacchetto *diverso* , il suo nome deve iniziare con una lettera maiuscola.

```
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
    }
}
```

Tuttavia, dall'interno `bank` pacchetto `bank` , è possibile accedere a UserId e accessToken senza problemi.

La `bank` pacchetti potrebbe essere implementata in questo modo:

```
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
}
```

Composizione e incorporamento

La composizione fornisce un'alternativa all'eredità. Una struct può includere un altro tipo per nome nella sua dichiarazione:

```
type Request struct {
    Resource string
}

type AuthenticatedRequest struct {
    Request
    Username, Password string
}
```

Nell'esempio sopra, `AuthenticatedRequest` conterrà quattro membri pubblici: `Resource` , `Request` , `Username` e `Password` .

Le strutture composite possono essere istanziate e utilizzate allo stesso modo delle normali strutture:

```
func main() {
    ar := new(AuthenticatedRequest)
    ar.Resource = "example.com/request"
    ar.Username = "bob"
    ar.Password = "P@ssw0rd"
    fmt.Printf("%#v", ar)
}
```

giocarci sul campo da gioco

Incorporare

Nell'esempio precedente, `Request` è un campo incorporato. La composizione può anche essere ottenuta inserendo un tipo diverso. Questo è utile, ad esempio, per decorare una struttura con più funzionalità. Ad esempio, continuando con l'esempio di risorsa, vogliamo una funzione che formatta il contenuto del campo Risorsa per prefissarlo con `http://` o `https://` . Abbiamo due opzioni: crea i nuovi metodi su AuthenticatedRequest o **incorporalo** da una diversa struttura:

```
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
}
```

E ora la funzione principale potrebbe fare quanto segue:

```
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)
}
```

Guarda che `AuthenticatedRequest` ha una struttura incorporata `ResourceFormatter` .

Ma il rovescio della medaglia è che non puoi accedere ad oggetti al di fuori della tua composizione. Quindi `ResourceFormatter` non può accedere ai membri da `AuthenticatedRequest` .

giocarci sul campo da gioco

metodi

I metodi di costruzione sono molto simili alle funzioni:

```
type User struct {
    name string
}
```

```
func (u User) Name() string {
    return u.name
}

func (u *User) SetName(newName string) {
    u.name = newName
}
```

L'unica differenza è l'aggiunta del ricevitore del metodo. Può essere dichiarato come un'istanza del tipo o un puntatore a un'istanza del tipo. Poiché `SetName()` l'istanza, il ricevitore deve essere un puntatore per effettuare una modifica permanente nell'istanza.

Per esempio:

```
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())
}
```

Vai al parco giochi

Struttura anonima

È possibile creare una struttura anonima:

```
data := struct {
    Number int
    Text   string
} {
    42,
    "Hello world!",
}
```

Esempio completo:

```
package main
```

```
import (
    "fmt"
)

func main() {
    data := struct {Number int; Text string}{42, "Hello world!"} // anonymous struct
    fmt.Printf("%+v\n", data)
}
```

giocarci sul campo da gioco

tag

I campi Struct possono avere tag associati a loro. Questi tag possono essere letti dal pacchetto `reflect` per ottenere informazioni personalizzate specificate su un campo dallo sviluppatore.

```
struct Account {
    Username      string `json:"username"`
    DisplayName   string `json:"display_name"`
    FavoriteColor string `json:"favorite_color,omitempty"`
}
```

Nell'esempio precedente, i tag vengono utilizzati per modificare i nomi delle chiavi utilizzati dal pacchetto di `encoding/json` quando si effettua il marshalling o l'unmarshaling JSON.

Mentre il tag può essere qualsiasi valore di stringa, è consigliabile utilizzare la `key:"value"` separata dallo spazio `key:"value"` coppie `key:"value"` :

```
struct StructName {
    FieldName int `package1:"customdata,moredata" package2:"info"`
}
```

I tag struct utilizzati con il pacchetto `encoding/xml` e `encoding/json` sono usati in tutta la libarary standard.

Creare copie struct.

Una struttura può essere semplicemente copiata usando l'assegnazione.

```
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"
```

```
}
```

Nel caso precedente, `'t'` e 'u' sono ora oggetti separati (valori struct).

Poiché `T` non contiene alcun tipo di riferimento (fette, mappa, canali) come i suoi campi, `t u` sopra possono essere modificati senza influire l'un l'altro.

```
fmt.Printf("t.I = %d, u.I = %d\n", t.I, u.I) // t.I = 100, u.I = 1
```

Tuttavia, se `T` contiene un tipo di riferimento, ad esempio:

```
type T struct {
    I  int
    S  string
    xs []int // a slice is a reference type
}
```

Quindi una semplice copia per assegnazione dovrebbe copiare il valore del campo del tipo di sezione anche nel nuovo oggetto. Ciò risulterebbe in due oggetti diversi che si riferiscono allo stesso oggetto fetta.

```
// 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
```

Poiché sia u che t si riferiscono alla stessa porzione attraverso il loro campo xs, l'aggiornamento di un valore nella porzione di un oggetto rifletterebbe il cambiamento nell'altro.

```
// 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]
```

Pertanto, è necessario prestare la massima attenzione per garantire che questa proprietà del tipo di riferimento non produca comportamenti indesiderati.

Ad esempio, per copiare sopra oggetti, è possibile eseguire una copia esplicita del campo slice:

```
// 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 letterali

Un valore di un tipo di struct può essere scritto usando una *struct literal* che specifica i valori per i suoi campi.

```
type Point struct { X, Y int }
p := Point{1, 2}
```

L'esempio precedente specifica ogni campo nel giusto ordine. Che non è utile, perché i programmatori devono ricordare i campi esatti in ordine. Più spesso, una struttura può essere inizializzata elencando alcuni o tutti i nomi dei campi e i loro valori corrispondenti.

```
anim := gif.GIF{LoopCount: nframes}
```

I campi omessi sono impostati sul valore zero per il suo tipo.

Nota: **le due forme non possono essere mescolate nello stesso valore letterale.**

Struttura vuota

Una struct è una sequenza di elementi denominati, chiamati campi, ognuno dei quali ha un nome e un tipo. La struttura vuota non ha campi, come questa struttura anonima vuota:

```
var s struct{}
```

O come questo tipo di struct vuoto chiamato:

```
type T struct{}
```

La cosa interessante della struttura vuota è che, la sua dimensione è zero (prova The Go Playground):

```
fmt.Println(unsafe.Sizeof(s))
```

Questo stampa `0` , quindi la struttura vuota non prende memoria. quindi è una buona opzione per uscire dal canale, come (prova The Go 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.")
}
```

Leggi Structs online: https://riptutorial.com/it/go/topic/374/structs

Capitolo 67: Sviluppo per piattaforme multiple con compilazione condizionale

introduzione

La compilazione condizionale basata su piattaforma è disponibile in due formati in Go, uno con suffissi file e l'altro con tag build.

Sintassi

- Dopo " `// +build` ", può seguire una singola piattaforma o un elenco
- La piattaforma può essere ripristinata precedendola con `!` cartello
- Elenco di piattaforme separate dallo spazio sono ORed insieme

Osservazioni

Avvertenze per i tag di costruzione:

- Il `// +build` constraint deve essere posizionato all'inizio del file, anche prima della clausola del pacchetto.
- Deve essere seguito da una riga vuota per separare i commenti del pacchetto.

Elenco di piattaforme valide per entrambi i tag di costruzione e i suffissi dei file
androide
Darwin
libellula
FreeBSD
linux
NetBSD
openbsd
plan9
solaris
finestre

Fare riferimento all'elenco `$GOOS` in https://golang.org/doc/install/source#environment per l'elenco

delle piattaforme più aggiornato.

Examples

Costruisci tag

```
// +build linux

package lib

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

Annulla una piattaforma posizionando `!` prima di cio:

```
// +build !windows

package lib

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

L'elenco delle piattaforme può essere specificato separandole con spazi

```
// +build linux darwin plan9

package lib

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

Suffisso file

Se dai il nome al tuo file `lib_linux.go` , tutto il contenuto di quel file sarà compilato solo in ambienti Linux:

```
package lib

var OnlyCompiledInLinux string
```

Definire comportamenti separati in piattaforme diverse

Diverse piattaforme possono avere implementazioni separate dello stesso metodo. Questo esempio illustra anche come possono essere utilizzati insieme tag di costruzione e suffissi di file.

File `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")
}
```

Leggi Sviluppo per piattaforme multiple con compilazione condizionale online: https://riptutorial.com/it/go/topic/8599/sviluppo-per-piattaforme-multiple-con-compilazione-condizionale

Capitolo 68: Tempo

introduzione

Il pacchetto Go `time` fornisce funzionalità per misurare e visualizzare il tempo.

Questo pacchetto fornisce una struttura `time.Time` . `time.Time` , permettendo di memorizzare e fare calcoli su date e orari.

Sintassi

- time.Date (2016, time.December, 31, 23, 59, 59, 999, time.UTC) // initialize
- date1 == date2 // restituisce `true` quando i 2 sono nello stesso momento
- date1! = date2 // restituisce `true` quando i 2 sono momenti diversi
- date1.Before (date2) // restituisce `true` quando il primo è rigorosamente prima del secondo
- date1.After (date2) // restituisce `true` quando il primo è rigorosamente dopo il secondo

Examples

Return time. Time Zero Value quando la funzione ha un errore

```
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
}
```

Analisi del tempo

Se hai una data memorizzata come stringa, dovrai analizzarla. Usa `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
```

Il primo parametro è il layout in cui la stringa memorizza la data e il secondo parametro è la

stringa che contiene la data. `01/02/2006` a dire che il formato è `MM/DD/YYYY` .

Il layout definisce il formato mostrando come il tempo di riferimento, definito come `Mon Jan 2 15:04:05 -0700 MST 2006` sarebbe interpretato se fosse il valore; serve come esempio del formato di input. La stessa interpretazione verrà quindi apportata alla stringa di input.

È possibile visualizzare le costanti definite nel pacchetto orario per sapere come scrivere la stringa di layout, ma si noti che le costanti non vengono esportate e non possono essere utilizzate al di fuori del time package.

```
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"
)
```

Confronto del tempo

A volte è necessario conoscere, con 2 oggetti date, se ci sono corrispondenti alla stessa data o trovare la data successiva all'altra.

In **Go** , ci sono 4 modi per confrontare le date:

- `date1 == date2` , restituisce `true` quando i 2 sono nello stesso momento
- `date1 != date2` , restituisce `true` quando 2 sono di momento diverso
- `date1.Before(date2)` , restituisce `true` quando il primo è rigorosamente prima del secondo
- `date1.After(date2)` , restituisce `true` quando il primo è rigorosamente dopo il secondo

ATTENZIONE: quando i 2 tempi di confronto sono uguali (o corrispondono alla stessa data esatta), le funzioni `After` e `Before` restituiranno `false` , poiché una data non è né prima né dopo se stessa

- `date1 == date1` , restituisce `true`
- `date1 != date1` , restituisce `false`
- `date1.After(date1)` , restituisce `false`
- `date1.Before(date1)` , restituisce `false`

SUGGERIMENTI: Se hai bisogno di sapere se una data è precedente o uguale a un'altra, devi solo combinare i 4 operatori

- `date1 == date2 && date1.After(date2)` , restituisce `true` quando date1 è successiva o uguale a date2
  o usando `! (date1.Before(date2))`
- `date1 == date2 && date1.Before(date2)` , restituisce `true` quando date1 è precedente o uguale a date2 o utilizza `!(date1.After(date2))`

Alcuni esempi per vedere come usare:

```
// 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
```

Leggi Tempo online: https://riptutorial.com/it/go/topic/8860/tempo

Capitolo 69: Text + HTML Templating

Examples

Modello di oggetto singolo

Si noti l'uso di `{{.}}` Per stampare l'elemento all'interno del modello.

```
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")
}
```

Risultati in:

```
Dear Professor Jones, How are you?
```

Modello di articoli multipli

Nota l'uso di `{{range .}}` e `{{end}}` per passare in rassegna la collezione.

```
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"})
}
```

Risultati in:

```
Dear Harry, Jane, Lisa, George,  How are you?
```

Modelli con logica personalizzata

In questo esempio, una mappa di funzioni denominata `funcMap` viene fornita al modello tramite il metodo `Funcs()` e quindi richiamata all'interno del modello. Qui, la funzione `increment()` viene usata per aggirare la mancanza di una funzione inferiore o uguale nel linguaggio dei template. Nota nell'output come viene gestito l'elemento finale nella raccolta.

A `-` all'inizio `{{-` o end `-}}` viene usato per tagliare gli spazi bianchi e può essere usato per rendere il modello più leggibile.

```
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"})
}
```

Risultati in:

```
Dear Harry, Jane, Lisa, and George; How are you?
```

Modelli con strutture

Nota come vengono ottenuti i valori di campo usando `{{.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})
}
```

Risultati in:

```
------------------------------
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?

------------------------------
```

Modelli HTML

Annota la diversa importazione del pacchetto.

```
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})
}
```

Risultati in:

```
<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>
```

In che modo i modelli HTML prevengono l'iniezione di codice dannoso

Innanzitutto, ecco cosa può accadere quando `text/template` viene utilizzato per HTML. Nota la proprietà di Harry's `FirstName`).

```
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})
}
```

Risultati in:

```
<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'esempio precedente, se si accede da un browser, comporterebbe lo script in esecuzione e un avviso generato. Se, invece, il `text/template` `html/template` fosse importato invece di `text/template` , lo script sarebbe stato sanificato in modo sicuro:

```
<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>
```

Il secondo risultato sembrerebbe confuso quando caricato in un browser, ma non comporterebbe un'esecuzione potenzialmente maligna dello script.

Leggi Text + HTML Templating online: https://riptutorial.com/it/go/topic/3888/text-plus-html-templating

Capitolo 70: Valori zero

Osservazioni

Una cosa da notare - i tipi che hanno un valore zero non zero come stringhe, int, float, bool e structs non possono essere impostati su zero.

Examples

Valori zero di base

Le variabili in Go sono sempre inizializzate se gli dai un valore iniziale o no. Ogni tipo, inclusi i tipi personalizzati, ha un valore zero su cui sono impostati se non viene fornito un valore.

```
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
```

Questo vale anche per mappe, sezioni, canali e tipi di funzioni. Questi tipi verranno inizializzati a zero. Negli array, ogni elemento viene inizializzato sul valore zero del rispettivo tipo.

Più valori zero complessi

A fette il valore zero è una fetta vuota.

```
var myIntSlice []int    // [] - an empty slice
```

Utilizzare `make` per creare una slice popolata con valori, tutti i valori creati nella slice sono impostati sul valore zero del tipo della slice. Per esempio:

```
myIntSlice := make([]int, 5)    // [0, 0, 0, 0, 0] - a slice with 5 zeroes
fmt.Println(myIntSlice[3])
// Prints 0
```

In questo esempio, `myIntSlice` è una slice `int` che contiene 5 elementi che sono tutti 0 perché è il valore zero per il tipo `int` .

Puoi anche creare una sezione con una `new` , questo creerà un puntatore a una sezione.

```
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
```

Nota: i puntatori Slice non supportano l'indicizzazione, quindi non puoi accedere ai valori usando `myIntSlice[3]` , invece devi farlo come `(*myIntSlice)[3]` .

Struct Zero Values

Quando si crea una struct senza inizializzarla, ogni campo della struct viene inizializzato al rispettivo valore zero.

```
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"
}
```

Array Zero Values

Come da blog Go :

> Gli array non devono essere inizializzati in modo esplicito; il valore zero di un array è un array pronto all'uso i cui elementi sono a loro volta azzerati

Ad esempio, `myIntArray` viene inizializzato con il valore zero di `int` , che è 0:

```
var myIntArray [5]int     // an array of five 0's: [0, 0, 0, 0, 0]
```

Leggi Valori zero online: https://riptutorial.com/it/go/topic/6069/valori-zero

Capitolo 71: Valori zero

Examples

Spiegazione

I valori zero o l'inizializzazione zero sono semplici da implementare. Provenendo da lingue come Java può sembrare complicato che alcuni valori possano essere `nil` mentre altri no. In sintesi da Zero Value: The Go Programming Language Specification :

> Puntatori, funzioni, interfacce, sezioni, canali e mappe sono gli unici tipi che possono essere nulli. Il resto viene inizializzato su false, zero o stringhe vuote in base ai rispettivi tipi.

Se una funzione che controlla alcune condizioni, potrebbero sorgere problemi:

```
func isAlive() bool {
    //Not implemented yet
    return false
}
```

Il valore zero sarà falso anche prima dell'implementazione. Test unitari dipendenti dal ritorno di questa funzione potrebbero dare falsi positivi / negativi.

Una tipica soluzione è anche restituire un errore, che è idiomatico in 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())
    }

}
```

giocarci sul campo da gioco

Quando si restituisce sia una struttura che un errore, è necessaria una struttura utente per il reso, che non è molto elegante. Ci sono due contro-opzioni:

- Lavorare con le interfacce: restituire nil restituendo un'interfaccia.
- Lavora con i puntatori: un puntatore **può** essere `nil`

Ad esempio, il seguente codice restituisce un puntatore:

```
func(d *DB) GetUser(id uint64) (*User, error) {
    //Some error ocurred
    return nil, err
}
```

Leggi Valori zero online: https://riptutorial.com/it/go/topic/6379/valori-zero

Capitolo 72: variabili

Sintassi

- var x int // declare la variabile x con tipo int
- var s string // dichiara la variabile s con tipo string
- x = 4 // definisce il valore x
- s = "pippo" // definisce il valore s
- y: = 5 // dichiara e definisce y inferendo il suo tipo a int
- f: = 4.5 // dichiara e definisce f inferendo il suo tipo a float64
- b: = "bar" // dichiara e definisce b inferendo il suo tipo a stringa

Examples

Dichiarazione delle variabili di base

Go è un linguaggio tipizzato staticamente, il che significa che in genere devi dichiarare il tipo di variabili che stai utilizzando.

```
// 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
```

Assegnazione di variabili multiple

In Go, puoi dichiarare più variabili allo stesso tempo.

```
// 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`
```

Se una funzione restituisce più valori, è anche possibile assegnare valori alle variabili in base ai

valori di ritorno della funzione.

```
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
```

Identificatore vuoto

Go genererà un errore quando è presente una variabile non utilizzata, al fine di incoraggiarti a scrivere codice migliore. Tuttavia, ci sono alcune situazioni in cui davvero non è necessario utilizzare un valore memorizzato in una variabile. In questi casi, si utilizza un "identificativo vuoto" _ per assegnare e scartare il valore assegnato.

Ad un identificatore vuoto può essere assegnato un valore di qualsiasi tipo ed è più comunemente utilizzato in funzioni che restituiscono più valori.

Più valori di ritorno

```
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
}
```

Usando 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)
    }

}
```

Controllo del tipo di una variabile

Ci sono alcune situazioni in cui non si è sicuri di quale sia il tipo di variabile quando viene restituito da una funzione. Puoi sempre controllare il tipo di una variabile usando `var.(type)` se non sei sicuro di quale tipo sia:

```
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
}
```

Leggi variabili online: https://riptutorial.com/it/go/topic/674/variabili

Capitolo 73: Vendoring

Osservazioni

La vendita è un metodo per garantire che tutti i pacchetti di terze parti che utilizzi nel tuo progetto Go siano coerenti per tutti coloro che si sviluppano per la tua applicazione.

Quando il tuo pacchetto Go importa un altro pacchetto, il compilatore normalmente controlla `$GOPATH/src/` per il percorso del progetto importato. Tuttavia, se il pacchetto contiene una cartella denominata `vendor` , il compilatore controllerà in quella cartella *prima.* Ciò significa che puoi importare pacchetti di altre parti all'interno del tuo repository di codice, senza dover modificare il loro codice.

La vendita è una funzione standard in Go 1.6 e versioni successive. In Go 1.5, è necessario impostare la variabile di ambiente di `GO15VENDOREXPERIMENT=1` per abilitare la vendita.

Examples

Utilizza il govendor per aggiungere pacchetti esterni

Govendor è uno strumento che viene utilizzato per importare pacchetti di terze parti nel proprio repository di codice in un modo compatibile con la distribuzione di golang.

Supponiamo ad esempio che tu stia utilizzando un pacchetto di terze parti `bosun.org/slog` :

```
package main

import "bosun.org/slog"

func main() {
    slog.Infof("Hello World")
}
```

La struttura della directory potrebbe essere simile a:

```
$GOPATH/src/
├── github.com/me/helloworld/
|   ├── hello.go
├── bosun.org/slog/
|   ├── ... (slog files)
```

Comunque qualcuno che cloni `github.com/me/helloworld` potrebbe non avere una `$GOPATH/src/bosun.org/slog/` , causando *il* fallimento della *loro* compilazione a causa di pacchetti mancanti.

L'esecuzione del seguente comando al prompt dei comandi acquisirà tutti i pacchetti esterni dal pacchetto Go e comprimerà i bit necessari in una cartella del fornitore:

```
govendor add +e
```

Questo istruisce il govendor ad aggiungere tutti i pacchetti esterni nel tuo attuale repository.

La struttura della directory dell'applicazione ora sarà simile a:

```
$GOPATH/src/
├── github.com/me/helloworld/
|   ├── vendor/
|   |   ├── bosun.org/slog/
|   |   |   ├── ... (slog files)
|   ├── hello.go
```

e quelli che clonano il tuo repository prenderanno anche i pacchetti necessari di terze parti.

Usare il cestino per gestire ./vendor

`trash` è uno strumento di vendita minimalista che puoi configurare con il file `vendor.conf` . Questo esempio è per il `trash` stesso:

```
# 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 prima riga non commentata è il pacchetto che stiamo gestendo ./vendor for (nota: questo può essere letteralmente qualsiasi pacchetto nel tuo progetto, non solo quello di root).

Le righe commentate iniziano con `#` .

Ogni riga non vuota e non commentata elenca una dipendenza. È necessario elencare solo il pacchetto "root" della dipendenza.

Dopo che il nome del pacchetto diventa la versione (commit, tag o branch) e opzionalmente l'URL del repository del pacchetto (per impostazione predefinita, viene dedotto dal nome del pacchetto).

Per popolare la tua directory ./vendor, devi avere il file `vendor.conf` nella directory corrente ed eseguire semplicemente:

```
$ trash
```

Il cestino clonerà le librerie vendute in `~/.trash-cache` (per impostazione predefinita), `~/.trash-cache` checkout delle versioni richieste, copierà i file nella directory `./vendor` e `./vendor` **pacchetti non importati e i file di test** . Questo ultimo passaggio mantiene il tuo ./vendor snello e cattivo e

aiuta a risparmiare spazio nel repository del tuo progetto.

Nota: dal v0.2.5 il cestino è disponibile per Linux e macOS e supporta solo git per recuperare i pacchetti, dato che git è il più popolare, ma stiamo lavorando per aggiungere tutti gli altri che `go get` supporti.

Usa golang / dep

golang / dep è un prototipo di strumento di gestione delle dipendenze. Presto sarà uno strumento di versioning ufficiale. Stato attuale **Alfa** .

uso

Ottieni lo strumento tramite

```
$ go get -u github.com/golang/dep/...
```

L'uso tipico su un nuovo repository potrebbe essere

```
$ dep init
$ dep ensure -update
```

Per aggiornare una dipendenza a una nuova versione, è possibile eseguire

```
$ dep ensure github.com/pkg/errors@^0.8.0
```

Si noti che i formati manifest e lock **sono ora stati finalizzati** . Questi rimarranno compatibili anche se lo strumento cambia.

vendor.json utilizzando lo strumento 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
```

Esempio 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"

}
```

Leggi Vendoring online: https://riptutorial.com/it/go/topic/978/vendoring

Capitolo 74: XML

Osservazioni

Mentre molti usi del pacchetto `encoding/xml` includono il marshalling e l'unmarshaling su una `struct` Go, vale la pena notare che non si tratta di una mappatura diretta. La documentazione del pacchetto afferma:

> La mappatura tra elementi XML e strutture dati è intrinsecamente imperfetta: un elemento XML è una raccolta dipendente dall'ordine di valori anonimi, mentre una struttura dati è una raccolta indipendente dall'ordine di valori denominati.

Per coppie di valori-chiave semplici, non ordinate, usare una codifica diversa come Gob o JSON può essere una soluzione migliore. Per dati ordinati o flussi di dati basati su eventi / richiamate, XML può essere la scelta migliore.

Examples

Decodifica di base / annullamento della memoria di elementi nidificati con dati

Gli elementi XML spesso nidificano, hanno dati in attributi e / o come dati di carattere. Il modo per acquisire questi dati è usando `,attr` e `,chardata` rispettivamente per questi casi.

```
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

Leggi XML online: https://riptutorial.com/it/go/topic/1846/xml

Capitolo 75: YAML

Examples

Creazione di un file di configurazione in formato 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
```

Leggi YAML online: https://riptutorial.com/it/go/topic/2503/yaml

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con 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	analisi	Adrian, Ankit Deshpande, Harshal Sheth, ivan.sim, Jared Ririe, Nathan Osman, Omid, Pavel Nikolov, Rodolfo Carvalho, seriousdev, Toni Villena, Zoyd
3	Analisi dei file CSV	Ainar-G
4	Argomenti e bandiere della riga di comando di analisi	Ingve, Pavel Kazhevets, Sam Whited
5	Array	NatNgs, nouney, Noval Agung Prayogo, Sam Whited
6	Autorizzazione JWT in Go	AniSkywalker
7	branching	burfl, Community, ganesh kumar, Ingve, nk2ge5k
8	canali	Chris Lucas, Howl, Jeremy, Kwarrtz, metmirr, RamenChef, Rodolfo Carvalho, Zoyd
9	CGO	MaC, Vojtech Kane
10	chiusure	abhink
11	Client HTTP	1lann, dmportella, Lanzafame, Sam Whited, SommerEngineering
12	Codifica Base64	Nathan Osman, RamenChef, Sam Whited
13	Collegare	Sam Whited

14	Compilazione incrociata	Jordan, Katamaritaco, mbb, mohan08p, RamenChef, Riley Guerin, SH', Siu Ching Pong -Asuka Kenji-, SommerEngineering, sztanpet, Zoyd
15	Concorrenza	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
16	Console I / O	Abhilekh Singh
17	Contesto	Ingaz, Sam Whited
18	costanti	Pavel Nikolov, RamenChef, Sam Whited, Simone Carletti
19	Costruisci vincoli	4444, RamenChef, Sam Whited, seriousdev
20	Crittografia	SommerEngineering
21	Differire	abhink, Adrian, Sam Whited, Vikash
22	Digita le conversioni	Adrian, Florian Hämmerle
23	Esecuzione di comandi	Krzysztof Kowalczyk, Kyle Brandt, Nevermore
24	Espansione in linea	Sam Whited
25	fette	1lann, Benjamin Kadish, burfl, cizixs, Grzegorz Żur, Guillaume, Jared Hooper, Joost, Jukurrpa, Kyle Brandt, Mark Henderson, NatNgs, RamenChef, Simone Carletti, skunkmb, Tanmay Garg, Zoyd
26	File I / O	1lann, Andres Kütt, greatwolf, Grzegorz Żur, koblas, noisewaterphd, Quentin Skousen, Sam Whited
27	Fmt	Lanzafame, Nevermore, Sam Whited
28	funzioni	Boris Le Méec, Dmytro Sadovnychyi, Grzegorz Żur, jayantS, LeoTao, Nathan Osman, nouney, palestamp, RamenChef, Right leg, Thomas Gerot
29	Gestione degli errori	bowsersenior, elevine, Elijah Sarver, Florian Hämmerle, groob, Ingve, Joe, Kin, Paul Hankin, Quentin Skousen, Sam Whited, Simone Carletti, Sridhar, Surreal Dreams, Vervious, Zoyd
30	Goroutines	mohan08p
31	Il comando Go	ganesh kumar, Harshal Sheth, Ingve, Lanzafame, Mayank Patel , Nevermore, Quentin Skousen, Sam Whited, theflametrooper,

		Vikash
32	immagini	putu
33	Iniziare con Go Using Atom	Ali M, Danny Chen, Katamaritaco
34	Installazione	sadlil
35	interfacce	Cody Roseborough, dotctor, Francis Norton, Grzegorz Żur, icza, Ingve, meysam, Mike, ptman, sadlil, Sam Whited, Wendy Adi
36	Invia / ricevi email	Utahcon
37	Iota	4444, Florian Hämmerle, Ingve, mohan08p, Sam Whited, Wojciech Kazior, Zoyd
38	JSON	Dmitry Udod, Joe, Jon Chan, Kyle Brandt, Nathan Osman, RamenChef, Sam Whited, shayan, Simone Carletti, sztanpet, Tanmay Garg, Utahcon
39	lettori	Mike Houston
40	Loops	1lann, burfl, Community, ivan73, jayantS, Jon Chan, mgh, MohamedAlaa, RamenChef, Sam Whited, Steven Maude, Thomas Gerot
41	Mappe	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
42	metodi	ganesh kumar, Pavel Kazhevets
43	MgO	Florian Hämmerle, Sourabh
44	middleware	Ankit Deshpande
45	Migliori pratiche sulla struttura del progetto	Iman Tumorang
46	Modelli	Pavel Kazhevets, RamenChef, Tanmay Garg
47	mutex	Adrian, Prutswonder
48	Pacchi	dmportella, Grzegorz Żur, icza, Michael, Nathan Osman, RadicalFish, RamenChef, skunkmb, tkausl

49	Panico e Recupero	JunLe Meng, Kaedys, Kristoffer Sall-Storgaard, Sam Whited
50	Piscine di lavoratori	burfl, photoionized, seriousdev
51	Pooling di memoria	Elijah Sarver, Grzegorz Żur, Kenny Grant
52	Profilazione usando go tool pprof	mbb, Nevermore, radbrawler
53	Programmazione orientata agli oggetti	Davyd Dzhahaiev, Sam Whited, zola
54	Protobuf in Go	mohan08p
55	puntatori	David Hoelzer, Jon Chan, Joost, Mal Curtis, metmirr, Nevermore, skunkmb
56	Registrazione	Grzegorz Żur, Jon Chan, Nathan Osman, Pavel Kazhevets, Sam Whited
57	Riflessione	ganesh kumar, mammothbane, radbrawler
58	Segnali OS	Community, Sam Whited, Utahcon
59	Seleziona e Canali	Harshal Sheth, Kaedys, RamenChef, Sam Whited, Utahcon
60	Server HTTP	Chief, frigo americain, Jon Erickson, Kin, Nathan Osman, rogerdpack, Sam Whited, Sascha, seriousdev, Simone Carletti, SommerEngineering, Tanmay Garg, Zhinkk
61	sputo	zola
62	SQL	Adrian, artamonovdev, bernardn, Francesco Pasa, Nevermore, Sam Whited, Sascha, Tanmay Garg, wrfly
63	Stringa	Ainar-G, NatNgs, raulsntos
64	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
65	Sviluppo per piattaforme multiple con compilazione condizionale	ecem
66	Tempo	Lanzafame, NatNgs, raulsntos

67	Text + HTML Templating	Stephen Rudolph
68	Valori zero	Harshal Sheth, raulsntos, Surreal Dreams
69	variabili	Community, FredMaggiowski, Jon Chan, Simone Carletti
70	Vendoring	Abhilekh Singh, Boris Le Méec, burfl, Dmitri Goldring, Ivan Mikushin, Mark Henderson, Martin Campbell, Michael, Sam Whited, Vardius
71	XML	ivarg, Sam Whited
72	YAML	Nathan Osman, Orr, Sam Whited