Introduzione al linguaggio GO

Return

Indice

Introduzione

- Linguaggio sviluppato da google (fine 2009)
- Linguaggio concorrente con primitive per lo scambio di messaggi:
  - canali
  - send sincrone/asincrone
  - Comandi con guardia
- Sintassi C-like
- Modulare (ma non OO)
- Tipi statici
- Garbage collection

- Numerosi packages aggiuntivi
- Documentazione su http://golang.org
    - Specifiche del linguaggio
    - Documentazione librerie
    - Sandbox

Hello.go

```
package main

import "fmt"

func main(){
    fmt.Print("Hello!\n")
}
```

compilazione e eseguzione

```
#compila e esegue il programma
$ go run hello.go

#compila il programma
$ go build hello.go
```

Linguaggio

Dichiaraizioni

```
var i int                               //variabile i di tipo intero
var k1 float32                          //k1 variabile reale
var k2 = 0.01                           //k2 variabile reale = 0.01
const PI = 22./7.                       //costante PI=3.14
type Point struct {x, y int}            //tipo Point
func sum(a, b int) int {return a+b}     //funzione somma
```

Short declaration

Soltanto all'interno del corpo di funzioni, le dichiarazioni del tipo: `var v = value`, posson essere espresse in modo sintentico nella forma: `v := value`, il tipo è quello della costante `value`.

Assegnamento

```
a=b
```

È possibile l'assegnamento multiplo

```
x,y,z = f1(), f2(), f3()
a,b = b,a //swap
```

if

```
if <Condizione1> {
    <istruzione1>
} else if <C2> {
    <istruzione2>
} else {
    <istruzione3>
}
```

Si può inizializzare nella condizione

```
if v:=f(); v<10 {
    ...
} else {
    ...
}
```

for

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

È possibile omettere gli argomenti del for:

```
for ;; {...}    //ciclo infinito
for {...}       //ciclo infinito
```

È possibile usare assegnamenti multipli

```
for i,j :=0,N; i<j; i,j = i+1,j-1 {...}
```

Switch

```
switch a {
    case 0: ...
    default: ...
}
```

O anche

```
switch a,b := x[i], y[j] {
    case a < b: return -1
    case a == b: return 0
    case a > b: return 1
}
```

Funzioni

Tramite la keyword `func`

```
func square(f float64) float64{
    return f*f
}
```

È possibile ritornare più di un valore

```
func MySqrt(f float64) (float64, bool){
    if f>=0{
        return math.Sqrt(f), true
    } else {
        return 0, false
    }
}
```

Funzioni con risultati multipli: esempio

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

func main(){
    a, b :=vals()   //a = 3, b = 7
    _, c :=vals()   //scarto il primo risultato, c = 7
}
```

Array

```
var ar [10]int
```

- ar è un array di 10 interi (valori iniziali 0)
- Per ottenere la dimensione di un vettore: `len`

```
x:=len(ar)
```

anche nel for

```
for i:=0; i<len(ar); i++{
    ar[i]=i*i
}
```

È possibile definire slice come sottovettori:

```
A:=ar[2:8] //A riferimento al sottovettore
```

Struct

```
type Point struct {
    x, y float64
}

var p Point
p.x = 7
p.y = 23.4
var pp *Point = new(Point)  //allocazione dinamica
*pp = p
pp.x = 3.14     // equivalente a (*pp).x
```

Struttura di un programma GO

- Un programma è composto da un insieme di moduli detti package ognuno definito da uno o più file sorgenti.
- Ogni programma contiene almeno il package `main`. Il codice di ongi package è costituito da un insieme di funzioni, che a loro volta possono riferire nomi esportati da altri package, usando la sintassi: `packagename.itemname`
- L'eseguibile è costruito linkando l'insieme dei package utilizzati.

Package

- Importazione di nomi definiti di un package

```
import "pippo"
...
pippo.Util()
```

- I nomi esportabili iniziano con una maiuscola

```
package pippo

const GLO = 100              //esportata
var priv float64 = 0,99      //privata
func Util(){...}             //esportata
```

Concorrenza in GO

Goroutine

L'unità di esecuzione concorrente è la **goroutine**

- Funzione che esegue concorrentemente ad altre goroutine nello stesso spazio di indirizzamento.
- Un programma go in esecuzione è costituito da una o più goroutine concorrenti
- In generale ci sono più goroutine per thread di SO.
    - Il supporto runtime di go schedula le goroutine sui thread disponibili
    - La costante di ambiente `GOMAXPROCS` determina il numero di thread utilizzati (se `GOMAXPROCS`=1 $\rarr$ 1goroutine/thread)
- La goroutine può essere estremamente leggera

creazione goroutine

```
go <invocazione funzione>
```

```
func IsReady(what string, minutes int64){
    time.Sleep(minutes*60*1e9)  //unità nanosecondi
    fmt.Println(what, "is ready")
}

func main(){
    go IsReady("tea", 6)
    go IsReady("coffee", 2)
```

```
    fmt.Println("Waiting...")
    ...
}
```

- 3 goroutine concorrenti: main, IsReady("tea", ...) e IsReady("coffee", ...)

canali

L'interazione tra processi può essere espressa tramite comunicazione attraverso **canali**

Il canale permette sia la comunicazione che la sincronizzazione tra goroutines.

I canali sono oggetti di prima classe:

```
var C chan int
C = <espressione>
Op(C)
```

Caratteristiche

- Simmetrico/Asimmetrico, permette comunicazioni:
  - 1-1
  - 1-molti
  - molti-molti
  - molti-1
- Comunicazione sincrona e asincrona
- Bidirezionale, Monodirezionale
- Oggetto tipato

Definizione

```
var ch chan <Type> // <Type> tipo del messaggio
```

Inizializzazione

Una volta definito va inizializzato:

```
var C1, C2 chan bool
C1 = make(chan bool)        // canale non bufferizzato: send sincrone
C2 = make(chan bool, 100)   // canale bufferizzato: send asincrone
```

Oppure

```

```

```
C1 := make(chan bool)
C2 := make(chan bool, 100)
```

Il valore di un canale non inizializzato è la costante `nil`

Uso

L'operatore di comunicazione `<-` permette di esprimere sia send che receive:

```
//Send <canale> <- <messaggio>

c := make(chan int)     // c non bufferizzato
c <- 1                  // send il valore 1 in c
```

```
//Receive <variabile> = <- <canale>
v = <- c    // riceve un valore da c, da assegnare a v
<- c        // riceve un messaggio che viene scartato
i := <- c   // riceve un essaggio, il cui valore inizializza i
```

Semantica

Default: canali non bufferizzati, pertanto la comunicazione è sincrona. Quindi:

1. La send blocca il processo mittente in attesa che il destinatario esegua la receive
2. La receive blocca il processo destinatario in attesa che il mittente esegua la send

In questo caso la comunicazione è una forma di sincronizzazione tra goroutines concorrenti.

Una receive da un canale non inizializzato (`nil`) è bloccante. (si spacca il programma)

```
func partenza(ch chan <- int){
    for i:=0; ; i++{
        ch <- i //invia
    }
}

func arrivo(ch <- chan int){
    for {
        fmt.Println(<-ch) //riceve
    }
}

func main(){
    ch1 := make(chan int)
    go partenza(ch1)
    go arrivo(ch1)
}
```

Sincronizzazione padre-figlio

Come imporre al padre l'attesa della terminazione di un figlio (tipo la join)?

Uno un **canale dedicato** alla sincronizzazione:

```
var done = make(chan bool)
func figlio(){
    ...
    done <- true
}

func main(){
    go figlio
    <- done //attesa figlio, perchè la receive è bloccante
}
```

In alternativa, nel package sync esiste la possibilità di utilizzare un **WaitGroup**:

```
import "sync"

var wg sync.WaitGroup

func figlio(){
    defer wg.Done()
    ...
}

func main(){
    wg.Add(1)
    go figlio()
    wg.Wait()
}
```

Funzioni e canali

Una funzione può restituire un canale:

```
func partenza() chan int{
    ch := make(chan int)
    go func() {
        for i := 0; ; i++ {ch <- i}
    }()
    return ch
}

func main(){
```

```
    stream := partenza()    //stream è un canale int
    fmt.Println(<-stream)   //stampa il primo messaggio: 0
}
```

Chiusura canale

Un canale può essere chiuso (dal sender) tramite close: `close(ch)`

Il destinatario può verificare se il canale è chiuso nel modo seguente:

```
msg, ok := <- ch
```

Se il canale è ancora aperto, `ok = true`

Range

La clausola `range <canale>`

nel for ripete la receive dal canale specificato fino a che il canale non viene chiuso.

```
for v := range ch {fmt.Println(v)} // per ogni v (ricevuto dal canale)
finchè il canale è aperto, stampo v
```

equivale a:

```
for {   // ciclo infinito
    v, ok := <- ch  // v ricevuto dal canale, se ok è vero il canale è
aperto
    if ok {fmt.Println(v)} else {break} // se il canale è aperto stampo v
altrimenti spacco
}
```

Send asincrone

Canale bufferizzato di capacità 50

```
func main() {
    c := make(chan int, 50) // Canale bufferizzato

    go func() {
        Sleep(60 * 1e9) // Sleep di 60 secondi
        x := <-c
        fmt.Println("ricevuto", x)
    }()
```

```
    fmt.Println("sending", 10)
    c <- 10 // Non è bloccante grazie al buffer
    fmt.Println("inviato", 10)
    ...
}
```

```
#output:
sending 10 # subito
inviato 10 # subito
# sleep di 60 secondi
ricevuto 10
```

Comandi con guardia

select

È una istruzione di controllo analoga al comando con guardia alternativo.

```
select{
    case <guardia1>:
        <istruzioni1>
    case <guardia2>:
        <istruzioni2>
    ...
    case <guardiaN>
        <istruzioniN>
}
```

selezione non deterministica di un ramo con guardia valida, altrimenti attesa.

Nelle select le guardie sono semplici receive o send, il linguaggio go non prevede la guardia logica $\rarr$ guardie valide o ritardate.

```
ci, cs := make(chan int), make(chan string)
select{
    case v := <- ci:
        fmt.Printf("ricevuto %d da ci\n", v)
    case v := <- cs:
        fmt.Printf("ricevuto %s da cs\n", v)
}
```

C'è possibilità di un ramo `default`, sempre valido.

Guardia logica

Possiamo costruire le guardie logiche tramite una funzione che restituisce un canale:

```
func when(b bool, c chan int) chan int{
    if !b{
        return nil
    }
    return c
}
```