Introduzione al linguaggio go

http://golang.org: web site

golang-nuts@golang.org: user discussion

golang-dev@golang.org: developers

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
Ricco set di librerie (packages).

Risorse utili

- Tutta la documentazione su:

http://golang.org/

In particolare:

- language specification – tutorial “A tour of go”
- library documentation - setup and how-to docs – FAQs
- playground (esecuzione Go dal browser)

Go & gophers

Chi usa go?

Tra le principali aziende che utilizzano Go (o Golang) ci sono:

- **Google**: Essendo il creatore originale del linguaggio Go, Google lo utilizza ampiamente per molti dei propri servizi e progetti interni, come Google Cloud Platform.
- **Docker**, il popolare strumento per la creazione, la distribuzione e l'esecuzione di applicazioni in containers, è scritto principalmente in Go.
- **SoundCloud**, la piattaforma di streaming audio, ha adottato Go per alcuni dei suoi servizi backend per migliorare le prestazioni e la scalabilità.
- La **BBC,** che ha utilizzato Go per alcuni dei suoi progetti interni, incluso il backend di alcune delle sue applicazioni e servizi digitali.
- **Uber**, che utilizza Go per molti dei suoi servizi backend, inclusi quelli che gestiscono le richieste degli utenti, il calcolo dei percorsi e la gestione dei pagamenti.
- **Dropbox**, che ha iniziato a utilizzare Go per migliorare le prestazioni dei suoi servizi, accelerare lo sviluppo e gestire meglio le infrastrutture distribuite.

Queste sono solo alcune delle aziende più conosciute che fanno uso del linguaggio Go nelle loro attività di sviluppo. La lista completa è in continuo aggiornamento ed è consultabile alla pagina https://go.dev/wiki/GoUsers)

… CHI USA GO (https://go.dev/wiki/GoUsers)

- **Google**
- Airbrake - blog
- Apcera - blog
- Aruba Networks - golang-nuts
- BBC Worldwide – source
- Bitbucket - source
- bitly - github blog
- Canonical - source
- Carbon Games - source
- CloudFlare - blog article
- Cloud Foundry - blog github
- CloudWalk - github
- clypd - blog
- Conformal Systems
- Crashlytics - tweet
- Cupcake
- CustomerIO - tweet
- Digital Ocean - blog
- Disqus - blog
- DNSimple - blog
- dotCloud - docker slides
- drone.io - post github
- Dropbox
- Embedly - blog
- ErrPlane - blog
- Foize - github
- Flipboard - source (job post)
- Getty Images - tweet tweet
- GitHub - blog post
- Globo.com - github article
- GOV.UK - slides github blog
- Granify - blog
- Hailo - slides
- Heroku - Go blog post
- Intel Germany (Debugger QA Team) - source
- Iron.io - source blog:"30 servers to 2" blog:"2 years of production Go"
- JelloLabs - blog
- Jimdo - tweet github
- Koding - quora
- Lincoln Loop - blog (as part of https://botbot.me/ , #go-nuts IRC logging)
- New Bamboo - blog
- MaxCDN - blog
- Microcosm - slides
- Modcloth - github
- Moovweb - github source
- Mozilla Services - github blog
- MROffice - source
- New York Times
- Novartis - g+ post
- Netflix
- Ooyala - github
- Oyster Books - blog
- Poptip job posting
- Rawstream - tweets
- Raygun - blog
- Rendered :Text - blog
- Repustate - blog
- Rounds - blog
- Secret - blog
- SendGrid - blog
- SendHub
- Shopify tweet
- SmugMug - blog
- Skimlinks blog
- SoundCloud - blog
- Sourcegraph github
- Space Monkey - blog
- Splice - tweet
- Square - blog
- StatHat - Go blog post
- Steals.com blog
- Streetspotr - tweet
- Stretchr github
- SyndicatePro - source
- Tamber - blog
- The Plant
- Thomson Reuters Eikon - github source
- VividCortex - source github
- Zynga

Un primo programma (hello.go)

```
package main

import "fmt"

func main() {

fmt.Print("Hello !\n")

}
```

Comandi per la compilazione

- Dalla linea di comando, si usa il comando go:

```
$ go run hello.go
```

Compila ed esegue il programma esempio.go

```
$ go build esempio.go
```

Compilazione del programma

Per un approfondimento v.

https://pkg.go.dev/cmd/go

Tool di sviluppo

- Plugin per IDE “multi-linguaggio”:
  - Visual Studio
  - Eclipse

- LiteIDE: ambiente di sviluppo free

Dichiarazioni/definizioni

Ogni dichiarazione inizia con una keyword (var, const, type, func) seguita dal nome dell'oggetto dichiarato e dalle sue proprietà:

```
var i int                  //variabile i di tipo intero
var k1 float32             // k1 variabile reale
var k2 = 0.01              // k2 variabile reale con v.i. =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
```

:= "short declaration"

Soltanto all'interno del corpo di funzioni le dichiarazioni del tipo:

```
var v = value
```

Possono essere espresse in modo più sintetico nella forma:

```
v := value
```

Il tipo è quello della costante **`value`**.

Assegnamento

- L'assegnamento di valori a variabili si esprime nella forma seguente:

```
a=b
```

- E' possibile anche l'assegnamento multiplo:

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

- Le funzioni possono ritornare più di un risultato (vedi dopo), pertanto:

```
nbytes, error := Write(buf)
```

if

```
if <Cond> { <istruzione 1> } else {istruzione 2}
```

Oppure:

```
if <C1> {
<istruzione 1>
} else if <C2> {
<istruzione 2>
} else {<istruzione 3>}
```

Esempi if:

```
if x < 5 { less() }
```

```
if x < 5 {
        less()
} else if x == 5 {
        equal()
} else { more()}
```

- Possibilità di inizializzazione nella condizione (il punto e virgola funge da separatore:

```
 if v := f(); v < 10 {
      fmt.Printf("%d less than 10\n", v)
  } else {
      fmt.Printf("%d not less than 10\n", v)
}
```

for

Sintassi:

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

E' possibile omettere gli argomenti del for:

```
for ;; { fmt.Printf("ciclo infinito") }
```

Oppure anche:

```
for { fmt.Printf("ciclo infinito") }
```

- Possibilità di usare assegnamenti multipli:

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

switch

Simile a quello del C, ma più espressivo. Come nel C:

```
switch a {
        case 0: fmt.Printf("0")
        default: fmt.Printf("non-zero")
   }
```

Diversamente dal C:

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

Espressioni di vario tipo, etichette specificate da espressioni

Funzioni

Dichiarazione tramite la keyword **func:**

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

Una funzione può restituire 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 vale 3, b vale 7
   ...
    _, c := vals()   //scarto il primo risultato
   ...
}
```

Tipi di dato strutturati: array

Sintassi:

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

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

```
X:= len(ar)

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

Possibilità di definire slice come sottovettori:

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

struct

- Affinità con il C:

```
type Point struct {
      x, y float64
  }
var p Point
p.x = 7
p.y = 23.4
var pp *Point = new(Point) //alloc. dinamica
*pp = p
 pp.x = 3.14  // equivalente a(*pp).x
```

NB: Non c'è la notazione "->" per i puntatori a struct, la dereferenziazione è **automatica**.

Struttura dei programmi 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 (dal quale parte l'esecuzione); il codice di ogni package è costituito da un insieme di funzioni, che a loro volta possono riferire nomi (funzioni, tipi, variabili, costanti) esportati da altri package, usando la sintassi: "packagename.Itemname"
- L'eseguibile è costruito linkando l'insieme dei package utilizzati.

Package: importazione/esportazione

- **Importazione** di nomi definiti in un package:

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

- **Esportazione**: i nomi esportabili: iniziano con una maiuscola:

```
package pippo

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

Un programma semplice

```
package main
import "fmt" // il programma usa package "fmt" (libreria I/O)
const hello = "Hello, world !\n"
func main() {
    fmt.Print(hello)
}
```

Package: esempio

```
package pigreco
import "math"
var Pi float64
func init() {// init inizializza la var globale Pi
    Pi = 4*math.Atan(1)
}
============================
package main
import (
"fmt"
"pigreco"
)
var twoPi = 2*pigreco.Pi
func main() {
    fmt.Printf("2*Pi = %g\n", twoPi)
}
```

Concorrenza in go

goroutine

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

è una 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

- **Goroutine=thread**? In generale, 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 → 1 goroutine/thread)

👉 la goroutine può essere estremamente **leggera**

creazione goroutine

Sintassi : `go <invocazione funzione>`

- Esempio

```
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("I'm waiting...")
  …
}
```

-> 3 goroutine concorrenti: main, Isready(“tea”..), Isready(“coffee”…))

Interazione: canali

*"**Do not communicate by sharing memory. Instead, share memory by communicating**. "*

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)
```

Canale: caratteristiche

Proprietà del canale in go:

- **Simmetrico/asimmetrico**, permette la comunicazione:
  - 1-1,
  - 1-molti
  - molti-molti
  - molti-1

-Comunicazione **sincrona** e **asincrona**

-**bidirezionale**, **monodirezionale**

-**Oggetto tipato**

Definizione:

```
var ch chan <tipo>  // <tipo> dei messaggi
```

Canale: inizializzazione

Una volta definito, ogni canale 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`**.

Canale: uso

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

Send:

<canale> <- <messaggio>

Esempio:

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

NB La freccia è orientata nella direzione del flusso dei messaggi.

Receive:

<variabile> = <- <canale>

Riceve un messaggio dal <canale> e lo assegna alla <variabile>. Se il canale è vuoto, la receive è sospensiva (semantica bloccante).

Esempi:

```
v = <-c  // riceve un valore da c, da assegnare
         a v
<-c      // riceve un messaggio che viene scartato

i := <-c // riceve un messaggio, 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.

NB una receive da un canale non inizializzato (`nil`) è bloccante.

Esempio

```
func partenza(ch chan<- int) {
      for i := 0; ; i++ { ch <- i } // invia
}
func arrivo(ch <-chan int) {
      for { fmt.Println(<-ch) } // ricevi e stampa
  }


...
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 *join*)?

👉Uso un **canale dedicato** alla sincronizzazione:

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

func main() {
go  figlio
<-done // attesa figlio
}
```

👉 In alternativa all'uso di un canale dedicato, nel package sync esiste la possibilità di definire gruppi di goroutine (waiting groups, wg), per i quali il padre può invocare **`wg.Wait().`**

Funzioni & 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 }

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

Chiusura canale: close

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 è vero

altrimenti è falso (il sender lo ha chiuso).

Esempio:

```
for {
        v, ok := <-ch
        if ok {fmt.Println(v) }else {break}
}
```

range

La clausola

`range <canale>`

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

Esempio:

```
for v := range ch { fmt.Println(v) }
```

equivale a:

```
for {
        v, ok := <-ch
        if ok {fmt.Println(v) }else {break}
}
```

Send asincrone

Creazione di un **canale bufferizzato** di capacità 50:

```
c := make(chan int, 50)
go func() {
      time.Sleep(60*1e9)  // attesa di 60 secondi
      x := <-c
      fmt.Println(”ricevuto", x)
       }()
  fmt.Println("sending", 10)
  c <- 10 // non è sospensiva!
  fmt.Println(”inviato", 10)
```

Output:

```
sending 10     (subito)
inviato 10     (subito)
ricevuto 10    (dopo 60 secondi)
```

Comandi con guardia: select

Select è un'istruzione di controllo analoga al comando con guardia alternativo.

Sintassi:

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

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

STRUTTURA della GUARDIA:

Nella select le guardie sono semplici receive (o send): il linguaggio go **non prevede la guardia logica**. ➔ guardie **valide** o **ritardate**. (Ovvero: una guardia non può fallire.)

Esempio:

```
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)
}
```

Possibilità di un ramo **default**, sempre valido.

Guardia logica

Nella select le guardie sono receive (o send): il linguaggio go non prevede la 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
}
```

When(condizione, ch) ritorna:

- il canale ch se la condizione è vera
- nil se la condizione è falsa

Uso di when

Esempio produttori/consumatori (v. procons_sync.go):

```
var pronto_prod = make(chan int)
var pronto_cons = make(chan int)
var dati = make(chan int)
var DATI_CONS [MAXPROC]chan int
var contatore int = 0
...
select {
case x:= <-when(contatore < N, pronto_prod):
         contatore++
         msg = <-dati      // ricezione messaggio da inserire
         <inserimento msg nel buffer >
case x := <-when(contatore > 0, pronto_cons):
         contatore--
         <estrazione msg dal buffer >
         DATI_CONS[x] <- msg //consegna messaggio a consumatore
..
}
```