



Programmazione Concorrente in Go

Gianluca Maguolo
DEI Unipd
gianluca.maguolo@phd.unipd.it



Storia di Go

- Go è un linguaggio di programmazione sviluppato nel 2007 all'interno di Google
- Linguaggio estremamente semplice e leggibile
- Gestione della concorrenza molto intuitiva



Installare Go (Linux)

- Scaricare la cartella .zip da <https://golang.org/doc/install>
- Da terminale, entrare nella cartella in cui si è salvato il file
- Estrarre file con: `tar -C /usr/local -xzf go1.15.6.linux-amd64.tar.gz`
- Aggiungere local/go/bin al proprio PATH: `export PATH=$PATH:/usr/local/go/bin`
- `sudo snap install go --classic` (teoricamente rischiosa, a me ha funzionato)
- Se tutto è andato a buon fine l'istruzione `go version` mostrerà la versione che avete installato



Installare Go (Windows e Mac)

- Scaricate Go da <https://golang.org/doc/install>
- Eseguite il file (doppio click) e seguite le istruzioni
- Aprite il prompt dei comandi e digitate “go version” per controllare di aver installato correttamente Go



Editor di testo

Scaricate un editor di testo dove potrete scrivere il vostro codice. Alcuni esempi sono:

- Notepad++ (Windows)
- Notepadqq (usato negli esempi)
- Sublime Text
- Goland
- Visual Studio

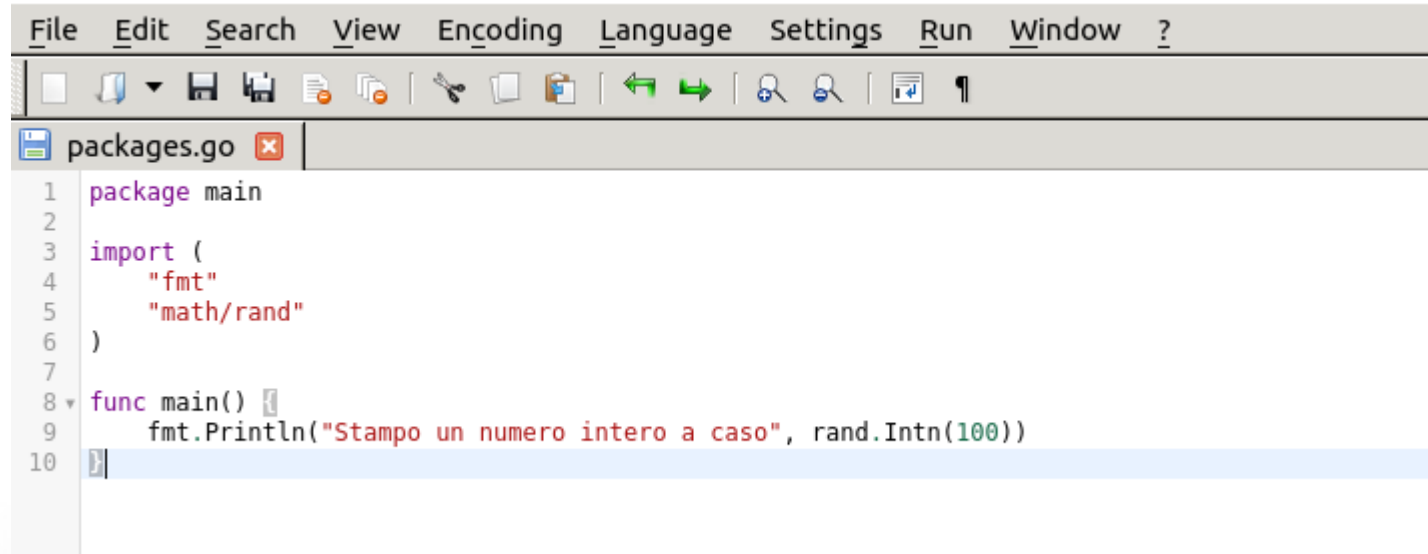


Perché GO

- Estremamente adatto alla programmazione concorrente
- Livelli di efficienza simili a C, C++, C#...
- Meno uso di memoria rispetto a C#.
- Usano attualmente GO nel loro codice: Google, Dropbox, Gitlab, Netflix, Twitch, Uber, Docker...

Packages

Questo programma Go carica i package **fmt** e **math/rand**. Poi stampa un numero intero.



```
1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6 )
7
8 func main() {
9     fmt.Println("Stampo un numero intero a caso", rand.Intn(100))
10 }
```

>> Stampo un numero intero a caso 81

Variabili

Le variabili vanno dichiarate prima di usarle. L'inizializzazione però avviene in automatico se non la si specifica. Ecco due esempi di codice con variabili inizializzate e non.

La sintassi “:=” si può usare come alternativa per dichiarare variabili nelle funzioni.

```
variabiliNOinizializzate.go
1 package main
2
3 import "fmt"
4
5 var a,b,c bool
6
7 func main() {
8     var i int
9     fmt.Println(a, b, c, i)
10 }
```

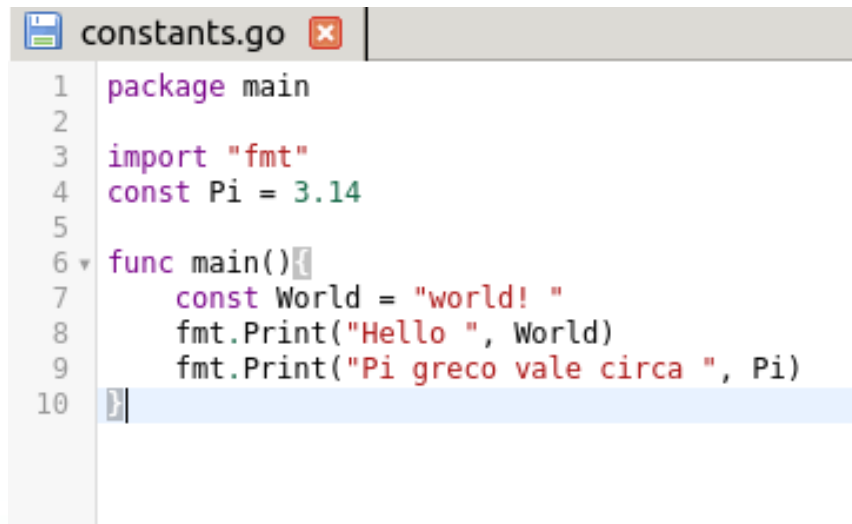
>> false, false, false, 1

```
variabiliPRIMAinizializzate.go
1 package main
2
3 import "fmt"
4
5 var a,b,c bool = true, false, true
6
7 func main() {
8     var i int = 8
9     fmt.Println(a, b, c, i)
10 }
```

>> true, false, true, 8

Costanti

Le costanti sono come le variabili, ma non possono essere modificate e vanno dichiarate con la keyword `const`.



```
1 package main
2
3 import "fmt"
4 const Pi = 3.14
5
6 func main(){
7     const World = "world! "
8     fmt.Print("Hello ", World)
9     fmt.Print("Pi greco vale circa ", Pi)
10 }
```

>> Hello world Pi greco vale circa 3.14



Slices

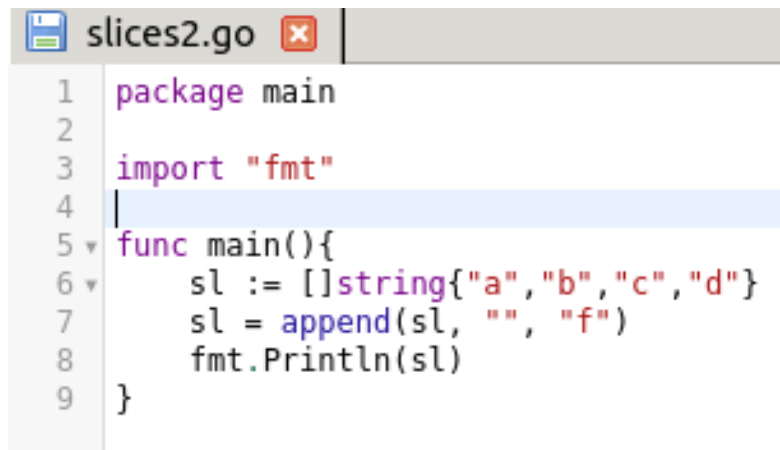
Gli slice sono sequenze di elementi dello stesso tipo (circa degli array). Il codice sotto mostra esempi di come creare e modificare uno slice. La numerazione parte da 0! Una volta creato, uno slice ha capienza fissata e ogni assegnazione oltre la capienza ritorna errore.

```
slices1.go x
1 package main
2
3 import "fmt"
4
5
6 func main(){
7     sl := []string{"a","b","c","d"}
8     fmt.Println("una parte dello slice: ",sl[1:3])
9     sl[2] = "e"
10    fmt.Println("lo slice modificato: ",sl)
11    fmt.Println("lunghezza dello slice :",len(sl))
12    fmt.Println("capienza dello slice: ",cap(sl))
13 }
```

```
>> una parte dello slice: [b c]
>> lo slice modificato: [a e c d]
>> lunghezza dello slice: 4
>> capienza dello slice: 4
```

Slices

Per aggiungere elementi a uno slice, bisogna usare la funzione `append`.

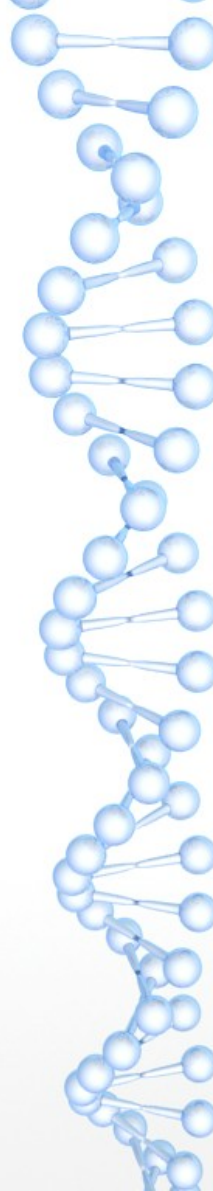


```
slices2.go x
1 package main
2
3 import "fmt"
4
5 func main(){
6     sl := []string{"a","b","c","d"}
7     sl = append(sl, "", "f")
8     fmt.Println(sl)
9 }
```

>> a b c d f

Range

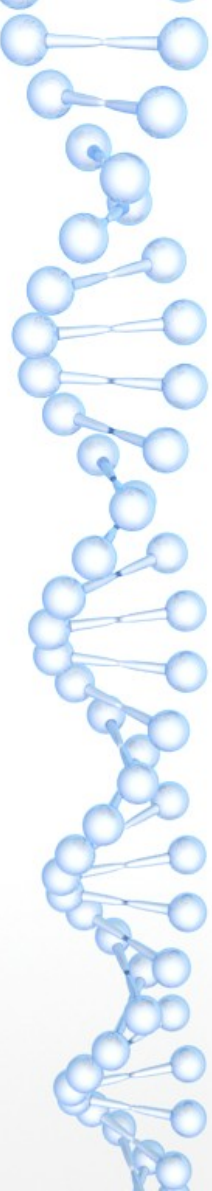
La keyword range permette di iterare lungo gli elementi di una struttura dati



```
range.go
1 package main
2
3 import "fmt"
4
5 func main(){
6     sl := []string{"a","b","c","d"}
7     for i := range sl{
8         fmt.Print(i, ", ")
9     }
10 }
```

>> a, b, c, d,

Funzioni



```
functionsMultipleOutputs.go
1 package main
2
3 import "fmt"
4 import "math"
5
6 func maxmin(x float64, y float64) (float64, float64) {
7     M := math.Max(x,y)
8     m := math.Min(x,y)
9     return m, M
10 }
11
12 func main(){
13     m, M := maxmin(18,9)
14     fmt.Print("I numeri ordinati sono ", m, " ", M)
15 }
```

>> I numeri ordinati sono 9 18

Strutture

Una struttura è un oggetto che comprende diversi campi. E' possibile definirla con la keyword `type` e si può creare semplicemente usando il suo nome.



```
structures.go
1 package main
2
3 import "fmt"
4
5 type Books struct {
6     title string
7     author string
8 }
9
10 func changeTitle(b Books, title string){
11     b.title = title
12 }
13
14 func main(){
15     b := Books {"La Commedia", "Dante Alighieri"}
16     changeTitle(b, "La Vita Nuova")
17     fmt.Print(b.title)
18 }
```

>> La commedia

Puntatori come argomenti delle funzioni

Per passare una variabile come puntatore a una funzione, questa deve essere preceduta dal simbolo &, e nella definizione di funzione deve essere indicata dal simbolo *.



```
structures.go
1 package main
2
3 import "fmt"
4
5 type Books struct {
6     title string
7     author string
8 }
9
10 func changeTitle(b *Books, title string){
11     b.title = title
12 }
13
14 func main(){
15     b := Books {"La Commedia", "Dante Alighieri"}
16     changeTitle(&b, "La Vita Nuova")
17     fmt.Print(b.title)
18 }
```

>> La Vita Nuova

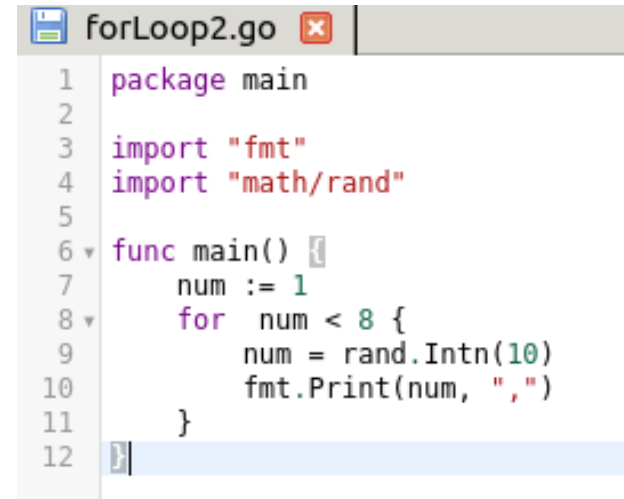
Cicli for e while

I cicli for e while si fanno usando la stessa keyword: "for".



```
1 package main
2
3 import "fmt"
4
5
6 func main() {
7     for i:= 1; i < 10; i++ {
8         fmt.Print(i, ",")
9     }
10 }
```

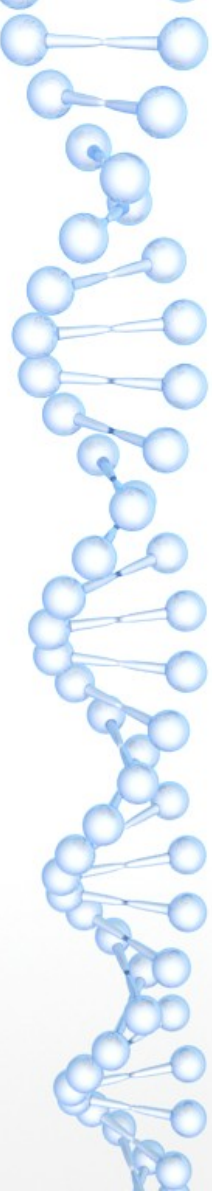
>> 1,2,3,4,5,6,7,8,9



```
1 package main
2
3 import "fmt"
4 import "math/rand"
5
6 func main() {
7     num := 1
8     for num < 8 {
9         num = rand.Intn(10)
10        fmt.Print(num, ",")
11    }
12 }
```

>> 1,7,7,9

Condizione if



```
if.go
1 package main
2
3 import "fmt"
4 import "math/rand"
5
6 func main() {
7     num := rand.Intn(10)
8     if num < 8 {
9         fmt.Print(num, " minore di otto")
10    } else if num > 8 {
11        fmt.Print(num, " maggiore di otto")
12    } else {
13        fmt.Print(num, " uguale a otto")
14    }
15 }
```

>> 1 minore di 8

Defer

La keyword “defer” posticipa l’esecuzione di un comando alla fine della porzione di codice.



```
defer.go
1 package main
2
3 import "fmt"
4
5 func main() {
6     defer fmt.Println("primo")
7
8     fmt.Println("secondo")
9 }
```

>> secondo

>> primo



Concorrenza in go

- La concorrenza in Go viene eseguita tramite quelle che si chiamano goroutine, che di fatto sono dei thread.
- Le goroutine vengono avviate da un altro thread usando la parola chiave go

Goroutine

Quando il main finisce, di default non si aspetta che finiscano gli altri thread!



```
goroutine.go
1 package main
2
3 import "fmt"
4 import "time"
5
6 func writeTwoTimes(s string) {
7     for i:=0; i<2; i++){
8         time.Sleep(100 * time.Millisecond)
9         fmt.Print(s)
10    }
11 }
12
13 func main() {
14     go writeTwoTimes("T ")
15     writeTwoTimes("M ")
16 }
```

>> T M M

Channels

- Un channel è la struttura che permette a due goroutines di inviarsi dati.
- Si crea un channel usando la funzione `make`.
- Per inviare o ricevere dati da un channel si usa il simbolo `<-`.
- Un thread si blocca fino a che l'istruzione di inviare o ricevere un elemento non si è completata.
- Se si vuole ricevere un elemento da un channel vuoto, il thread attende fino a che non viene inserito un elemento nel channel.
- Da un channel chiuso si riceve l'elemento zero della classe.
- Se si chiedono due output, il secondo è un flag che dice se il channel è aperto.

```
channels1.go
1 package main
2
3 import "fmt"
4 import "time"
5
6
7 func main(){
8     // unbuffered channel of ints
9     ic := make(chan int)
10    go func(){
11        ic <- 3
12        ic <- 4
13        close(ic)
14    }()
15    time.Sleep(time.Second)
16    receivedNumber1 := <-ic
17    fmt.Print(receivedNumber1, " ")
18    receivedNumber2 := <-ic
19    fmt.Print(receivedNumber2, " ")
20    receivedNumber3, flag := <-ic
21    fmt.Print(receivedNumber3, " ")
22    fmt.Print(flag)
23 }
```

>> 3, 4, 0, false

Buffered Channels

I channel possono essere buffered o unbuffered.

Nel primo caso, in ogni channel c'è spazio per un solo elemento.

Nel secondo caso la capienza va settata quando si crea il buffer.

In un channel bufferd l'inserimento degli elementi si blocca solo se il thread trova il buffer pieno.

channelsUnbuffered.go	channels1	channelsUnbuffered.go	channels1
<pre>1 package main 2 3 import "fmt" 4 import "time" 5 6 7 func main(){ 8 ic := make(chan int) 9 go func(){ 10 ic <- 3 11 ic <- 4 12 fmt.Print("finito ,") 13 close(ic) 14 }() 15 time.Sleep(time.Second) 16 receivedNumber1 := <-ic 17 fmt.Print(receivedNumber1, " ,") 18 receivedNumber2 := <-ic 19 fmt.Print(receivedNumber2, " ,") 20 }</pre>		<pre>1 package main 2 3 import "fmt" 4 import "time" 5 6 7 func main(){ 8 ic := make(chan int, 2) 9 go func(){ 10 ic <- 3 11 ic <- 4 12 fmt.Print("finito ,") 13 close(ic) 14 }() 15 time.Sleep(time.Second) 16 receivedNumber1 := <-ic 17 fmt.Print(receivedNumber1, " ,") 18 receivedNumber2 := <-ic 19 fmt.Print(receivedNumber2, " ,") 20 }</pre>	

>> 3, finito, 4,

>> finito, 3, 4,

Select

- La keyword select permette di aspettare il primo a finire tra un gruppo di thread.
- Nel caso in cui il channel che da cui si vuol ricevere un input sia vuoto, l'opzione default permette di proseguire comunque senza fermarsi.
- A destra, sono riportate le tre possibili stampe dello script, a seconda dei valori assunti da rand.Intn(5).

```
func returnNumber(num int, ic chan int){  
    time.Sleep(time.Duration(rand.Intn(5))*time.Second)  
    ic <- num  
}  
  
func main(){  
    rand.Seed(time.Now().UnixNano())  
    c1 := make(chan int)  
    c2 := make(chan int)  
    go returnNumber(1, c1)  
    go returnNumber(2, c2)  
    time.Sleep(time.Duration(rand.Intn(5))*time.Second)  
    select{  
        case x := <- c1:  
            fmt.Print(x)  
        case x := <- c2:  
            fmt.Print(x)  
        default:  
            fmt.Print("nessun input")  
    }  
}
```

>> nessun input

>> 1

>> 2

Select

Una alternativa al codice della slide precedente è il seguente.

Al posto di default, si aspetta per un periodo di tempo fissato e in caso si va avanti seguendo un caso prestabilito.

```
select2.go
1 package main
2
3 import "fmt"
4 import "time"
5 import "math/rand"
6
7 func returnNumber(num int, ic chan int){
8     time.Sleep(time.Duration(rand.Intn(5))*time.Second)
9     ic <- num
10 }
11
12 func main(){
13     rand.Seed(time.Now().UnixNano())
14     c1 := make(chan int)
15     c2 := make(chan int)
16     go returnNumber(1, c1)
17     go returnNumber(2, c2)
18     select{
19         case x := <- c1:
20             fmt.Print(x)
21         case x := <- c2:
22             fmt.Print(x)
23         case <-time.After(time.Duration(rand.Intn(5))*time.Second):
24             fmt.Print("nessun input")
25     }
26 }
```

>> 1

>> 2

>> nessun input

Data races

- Con data race si intende la potenziale modifica contemporanea di un oggetto da parte di due thread.
- In Go, per evitare che accada basta mettere l'oggetto in un buffered channel di capienza 1.
- In questo modo, ogni thread che voglia modificarlo (O LEGGERLO) deve prima acquisirlo, e poi reinserirlo nel channel

```
dataRaces.go
1 package main
2
3 import "fmt"
4 import "time"
5
6 func increaseNumber(ic chan int){
7     num := <- ic
8     num ++
9     ic <- num
10 }
11
12 func main(){
13     c1 := make(chan int,1)
14     c1 <- 0
15     // altro code...
16     go increaseNumber(c1)
17     go increaseNumber(c1)
18
19     time.Sleep(time.Second)
20     num2 := <- c1
21     fmt.Print(num2)
22 }
```

>> 2

WaitGroup

- Per aspettare che un gruppo di thread finisca l'esecuzione prima di passare oltre, si usa WaitGroup
- La funzione add aggiunge all'oggetto un numero di thread da aspettare
- La funzione Done comunica che il thread in questione è arrivato al punto di sincronizzazione
- La funzione Wait mette in pausa il thread finché tutti i thread in questione non hanno raggiunto il punto di sincronizzazione.
- L'oggetto wg deve essere passato come puntatore!!!

```
wait.go
1 package main
2
3 import "fmt"
4 import "sync"
5
6 func increaseNumber(ic chan int, wg *sync.WaitGroup){
7     num := <- ic
8     num ++
9     ic <- num
10    wg.Done()
11 }
12
13 func main(){
14     c1 := make(chan int,1)
15     c1 <- 0
16     // altro code...
17     var wg sync.WaitGroup
18     wg.Add(2)
19     go increaseNumber(c1, &wg)
20     go increaseNumber(c1, &wg)
21
22     wg.Wait()
23     num2 := <- c1
24     fmt.Print(num2)
25 }
```

>> 2

Sincronizzare thread



```
sync.go
1 package main
2
3 import "fmt"
4 import "sync"
5
6 func someThread(wg *sync.WaitGroup){
7     fmt.Print("prima, ")
8     wg.Done()
9     wg.Wait()
10    fmt.Print("dopo, ")
11 }
12
13 func main(){
14     var wg sync.WaitGroup
15     wg.Add(3)
16     go someThread(&wg)
17     go someThread(&wg)
18
19     fmt.Print("prima, ")
20     wg.Done()
21     wg.Wait()
22     fmt.Print("dopo, ")
23 }
```

>> prima, prima, prima, dopo, dopo, dopo,