Esercitazione 3
Programmazione Concorrente nel linguaggio go

15 Novembre 2024

Concorrenza in go

Agenda:

- Concorrenza basata su **goroutine**
- Interazione basata sullo scambio di messaggi:
  - Mezzo di comunicazione: **chan**
  - Operatore di comunicazione: **<-**
  - Ricezione non deterministica messaggi in arrivo: istruzione **select**
  - Realizzazione di **guardie logiche**

goroutine

L’unità di esecuzione concorrente è la “**goroutine**”:

è una chiamata a 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.

Una goroutine **condivide lo spazio di indirizzamento** con le altre.

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...")
  …
}
```

-> tre threads: main, Isready(“tea”..), Isready(“coffee”…))

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 di un canale:

```
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 //invia il valore 1 in c (send sincrona!)
```

NB La freccia "punta" nella direzione del flusso dei messaggi.

Receive:

<variabile> = <- <canale>

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**): 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

Non esiste l'equivalente della «**join**» per attendere la terminazione di un processo.

Come imporre al padre l'attesa della terminazione di un figlio?

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

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

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

[Oppure: v. `WaitGroup` in `package sync`**]**

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

Per avere una semantica asincrona nella send è necessario utilizzare un **canale bufferizzato.**

Quindi, ad esempio:

```
const DIM = 50
c:= make(chan int, DIM)
```

il canale c è stato creato come canale bufferizzato, con capacità pari a 50 interi.

👉 Ogni send sul canale c avrà una semantica asincrona. Nel caso di canale pieno, la send sospenderà la goroutine fino a quando non verrà eseguita la prima receive su c.

Send asincrone

Esempio:

```
c := make(chan int, 50)
go func() {
      time.Sleep(60*1e9)
      x := <-c  // receive
      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>
}
```

Semantica:

Selezione **non deterministica** di un ramo con guardia valida, altrimenti attesa (se tutti i rami hanno guardia ritardata).

STRUTTURA della GUARDIA:

Nella select **non è prevista la guardia logica :** le guardie sono semplici **receive**

👉 Valutazione: solo guardie **valide** o **ritardate**; **non esiste la guardia fallita.**

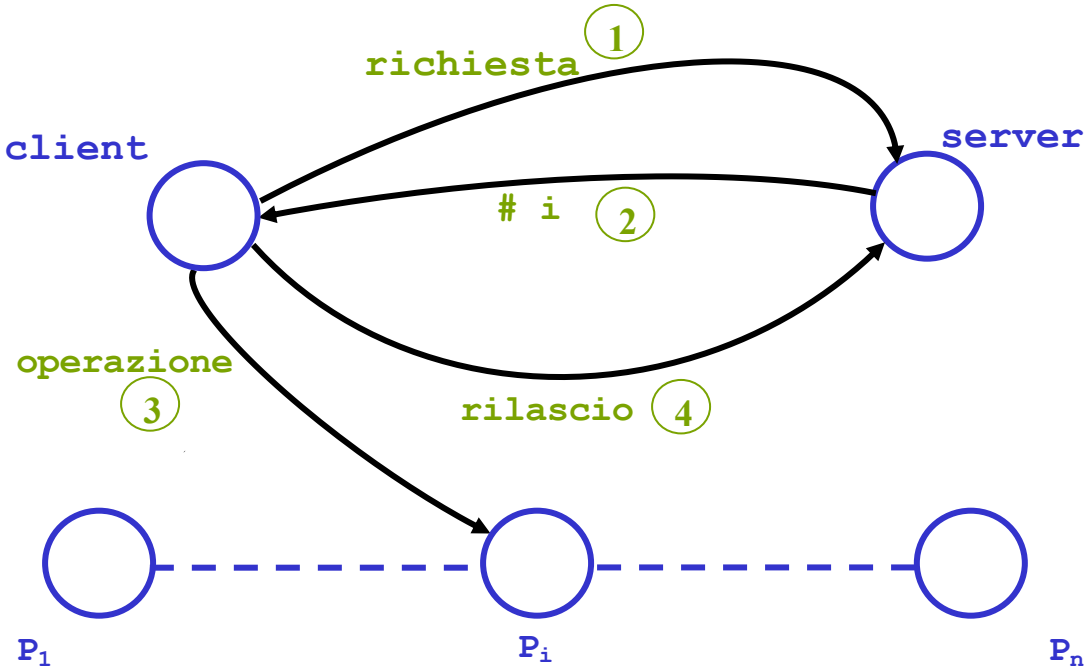
Esempio:

```
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.

Esempio: gestione di un pool di risorse equivalenti

Realizzare la gestione di un **pool di risorse equivalenti, con priorità.**



Soluzione senza guardie logiche (esempio.pool1.go)

```
// esempio.pool1.go
package main

import (
        "fmt"
        "math/rand"
        "time"
)
const MAXPROC = 100 //massimo numero di processi
const MAXRES = 5    //massimo numero di risorse nel pool

//Canali:
var richiesta = make(chan int)
var rilascio = make(chan int)
var risorsa [MAXPROC]chan int //1 chan per ogni cliente x ricez. risorsa
var done = make(chan int)
var termina = make(chan int)
```

```
func client(i int) {
        richiesta <- i
        r := <-risorsa[i]
        // uso della risorsa r:
        timeout := rand.Intn(3)
        time.Sleep(time.Duration(timeout) * time.Second)
        rilascio <- r
        done <- i //comunico al main la terminazione
}
```

```
func server(nris int, nproc int) {
        var disponibili int = nris
        var res, p, i int
        var libera [MAXRES]bool
        var sospesi [MAXPROC]bool
        var nsosp int = 0

        for i := 0; i < nris; i++ {
                libera[i] = true
        }
        for i := 0; i < nproc; i++ {
                sospesi[i] = false
        }
        // continua..
```

```
for {
        select {
        case p = <-richiesta:
                if disponibili > 0 { //allocazione della risorsa
                        for i = 0; i < nris && !libera[i]; i++ {
                        }
                        libera[i] = false
                        disponibili--
                        risorsa[p] <- i
                } else { // attesa
                        nsosp++
                        sospesi[p] = true
                }
```

```
        case res = <-rilascio:
                if nsosp == 0 {
                        disponibili++
                        libera[res] = true
                } else { // alloc. res al processo in attesa più prio.
                        for i = 0; i < nproc && !sospesi[i]; i++ {
                        }
                        sospesi[i] = false
                        nsosp--
                        risorsa[i] <- res //il proc. i ottiene res
                        }
        case <-termina: // tutti i processi clienti hanno finito
                fmt.Println("-- FINE -- ")
                done <- 1
                return
        } //fine select
  } //fine for
} //fine server
```

```
func main() {
        var cli, res int
        rand.Seed(time.Now().Unix())
        fmt.Printf("\n quanti clienti (max %d)? ", MAXPROC)
        fmt.Scanf("%d", &cli)
        fmt.Println("clienti:", cli)
        fmt.Printf("\n quante risorse (max %d)? ", MAXRES)
        fmt.Scanf("%d", &res)
        fmt.Println("risorse da gestire:", res)
        for i := 0; i < cli; i++ { //inizializzazione canali
                risorsa[i] = make(chan int)
        }
        for i := 0; i < cli; i++ { //creazione clienti
                go client(i)
        }
        go server(res, cli) //creazione server
        for i := 0; i < cli; i++ { //attesa della term. dei clienti:
                <-done
        }
        termina <- 1 //terminazione server
        <-done //attesa della terminazione del server
}
```

Esercizio 1 - Il noleggio biciclette

In un'area naturalistica è disponibile un servizio di **noleggio bici** a disposizione dei visitatori.

Il servizio dispone complessivamente di:

- **NB** biciclette tradizionali
- **NE** e-bike.

Le richieste dei clienti possono essere di **3 tipi**:

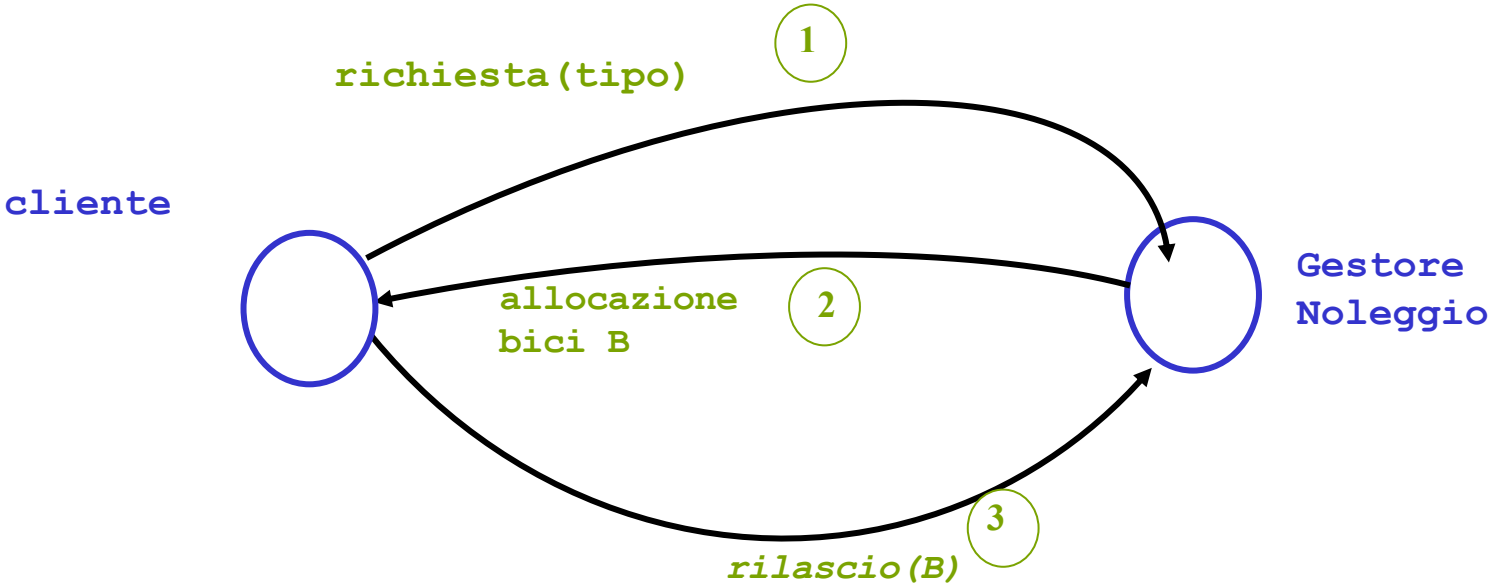
1. **BT**: richiesta di una bici tradizionale
2. **EB**: richiesta di una e-bike
3. **FLEX**: richiesta flessibile, ovvero:

   viene richiesta una e-bike:

   1. se c'è un'e-bike disponibile, il cliente ottiene l'e-bike.
   2. se non ci sono e-bike disponibili, e ce n'è almeno una tradizionale, il cliente ottiene una bici tradizionale;
   3. se non ci sono bici disponibili (di nessun tipo), il cliente attende un'e-bike.

Realizzare un'applicazione concorrente nel linguaggio go, nella quale ogni cliente e il gestore del noleggio siano rappresentati da **goroutine** distinte.

Impostazione



Impostazione

Quali e quante goroutine?

- 1 Server (gestore delle bici)
- N Client (uno per ogni visitatore)

Quali e quanti canali?

- **il server riceve le richieste su:**
  - **1 canale richiesta: molti a uno per la richiesta(tipo), dove tipo** $\in \{BT, EB, FLEX\}$
- **il server riceve i rilasci su:**
  - **1 canale rilascio:molti a uno per il rilascio(bici), dove bici** $\in \{BT, EB\}$
- **a seguito di una richiesta, ogni client attende il messaggio di allocazione:**
  - **N canali risorsa (1 per ogni cliente) : (Server->Client):**
- **canali per la sincronizzazione padre-figli**
  - **done: attesa della terminazione delle goroutine (goroutine>main)**
  - **termina: messaggio di terminazione al server (main ->server)**

Struttura del Server

```
var richiesta = make(chan req)  // quale tipo per req?
var rilascio = make(chan bici)   //quale tipo per bici?
var risorsa [MAXPROC]chan bici
var done = make(chan int)
var termina = make(chan int)

func server() {
<var locali: bici disponibili, ecc.>
for {
    select {
          case r = <-richiesta:
                <gestione della richiesta di allocazione di una risorsa, in base al tipo di
richiesta pervenuta: FLEX, EB, BT>
          case b = <-rilascio:
                 < rilascio della bici b>
          case <-termina: // quando tutti i processi clienti hanno finito, il server termina
                    done <- 1   //comunicazione al main: il server sta terminando
                    return
   } } }
```

Struttura del Client

```
func client(r req, id int) {
        var b bici
        richiesta <- r   //invio richiesta -  r indica il tipo: BT, EB o FLEX
        b = <-risorsa[id] //allocazione della bici del tipo b: BT o EB
        <uso della risorsa>
        rilascio <- b  //restituzione della bici
        done <- .. //comunicazione al main: il client sta terminando
}
```

Struttura del main

```
func main() {
        //inizializzazione canali
        ...
        go server()

        for i := 0; i < Nclienti; i++ {
                <definizione tipo richiesta per il cliente i>
                go client(r, i)
        }

        for i := 0; i < Nclienti; i++ { // attesa terminazione clienti
                <-done
        }
        termina <- 1 //terminazione server
        <-done   // attesa terminazione server
}
```

Uso di «guardie logiche» in go

Guardia logica: realizzazione

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
}
```

la funzione **`when(condizione, ch)`** ritorna:

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

* ricordiamo che **una receive da un canale con valore nil è sospensiva**

Esempio: pool di risorse equivalenti senza priorità (esempio.pool2.go)

```
func server(nris int) {
        var disponibili int = nris
        var res, p, i int
        var libera [MAXRES]bool
        for i := 0; i < nris; i++ { libera[i] = true }
        for {
             select {
                 case p = <-when(disponibili > 0, richiesta):
                          for i = 0; i < nris && !libera[i]; i++ {}
                          libera[i] = false
                          disponibili--
                          risorsa[p] <- i
                 case res = <-rilascio:
                          disponibili++
                          libera[res] = true
                 case <-termina: // quando tutti i processi clienti hanno finito
                          done <- 1
                          return
        }
}}
```

Esempio: pool di risorse equivalenti

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

Possibile implementazione del comando con guardia alternativo

```
var cc chan int
cc=make(chan int, 10);
cc<-1
select {
case msg:= <-when(Cond1, canale1):
                                    <blocco di istruzioni 1>
case msg:= <-when(Cond2, canale2):
                                    <blocco di istruzioni 2>
..
case msg:= <-when(CondN, canaleN):
                                    <blocco di istruzioni N>
case msg:=<-when((!Cond1)&&(!Cond2)&&…&&(!CondN),cc):
                                    break  // tutte le guardie fallite
}
```

NB ipotizziamo che canale1, .. canaleN siano tutti dello stesso tipo, altrimenti occorrerebbe definire una funzione “when“ per ogni canale di tipo diverso

Esercizio 2 - Il noleggio biciclette (con le guardie logiche)

Risolvere l'esercizio 1 utilizzando le «guardie logiche» mediante un'opportuna funzione «when».