Programmazione Concorrente

In informatica la concorrenza è una caratteristica dei sistemi di elaborazione nei quali può verificarsi che un insieme di processi o sottoprocessi (thread) computazionali sia in esecuzione nello stesso istante. L'esecuzione parallela può condurre a interazione tra processi quando viene coinvolta una risorsa condivisa.

L'esecuzione di un processo parallelo necessita di:

- un elaboratore non sequenziale;
- un linguaggio di programmazione non sequenziale.

La sequenza di istruzioni con la quale un processo accede e modifica un insieme di variabili condivise prende il nome di sezione critica.

Mutua esclusione: L'accesso a una risorsa in cui a ogni istante, al massimo un processo può accedere a quella risorsa. In particolare per avere la mutua esclusione devono essere soddisfatte le seguenti quattro condizioni:

- nessun coppia di processi può trovarsi simultaneamente nella sezione critica;
- l'accesso alla regione critica non deve essere regolato da alcuna assunzione temporale o dal numero di CPU;

nessun processo che sta eseguendo codice al di fuori della regione critica può bloccare un processo interessato a entrarvi; nessun processo deve attendere indefinitamente per poter accedere alla regione critica.

Un'errata sincronizzazione può portare al fallimento delle elaborazioni, che genera situazioni di:

- ***starvation*** (o blocco individuale): si verifica quando un processo rimane in attesa di un evento che non accadrà mai, e quindi non può portare a termine il proprio lavoro.
- ***deadlock*** (blocco multiplo): si verifica quando due o più processi rimangono in attesa di eventi che non potranno mai verificarsi a causa di condizioni cicliche nel possesso e nella richiesta di risorse.

Linguaggio di programmazione GO

Introduzione

Il linguaggio di programmazione Go è un progetto open source per rendere più produttivi i programmatori. Go è espressivo, conciso, pulito ed efficiente. Obiettivo dichiarato, rendere la programmazione veloce, produttiva e divertente. Go, assicura l'azienda di Mountain View, è in grado di offrire un'elevata velocità di compilazione. Go è un tentativo di combinare la facilità di programmazione di un linguaggio interpretato tipicamente dinamico con l'efficienza e la sicurezza di un linguaggio compilato tipicamente statico. Go è un linguaggio object- oriented sì e no. Anche se Go ha tipi e metodi e consente uno stile di programmazione orientato agli oggetti, non esiste una gerarchia dei tipi. Inoltre, la mancanza di gerarchia dei tipi dà la sensazione che gli "oggetti" in Go siano molto più leggeri rispetto ai linguaggi come C++ o Java.

Caratteristiche

Le caratteristiche del linguaggio sono:

- ***velocità*** – offre un'alta velocità di esecuzione e anche di compilazione in quanto il codice viene trasformato in binario in maniera più o meno istantanea, è possibile realizzare programmi che girano alla stessa velocità di applicazioni native in C, consente una velocità di compilazione eccellente su singola macchina e anche per grossi file binari. Il componente essenziale - che poi rende il sistema così propenso alla velocità - di ogni "Go-programma" sono le "Go-routine": processi leggeri ed efficienti a cui vengono delegati i compiti di gestire i vari sistemi e server da sviluppare;
- ***semplicità*** – presenta una sintassi semplice paragonabile a quella di un C misto a Python, fornisce un modello per la costruzione di un software che rende facile l'analisi delle dipendenze ed evita gran parte del sovraccarico dello stile C che include i file e le librerie;
- ***similarità C*** – il linguaggio si presenta come un C object-oriented molto ottimizzato. Presenta caratteristiche tipiche dei linguaggi dinamici, ma mantiene, quasi immutate, caratteristiche del C;
- ***concorrenza*** – implementata a livello nativo e grazie all'utilizzo delle "goroutine", semplici funzioni eseguite concorrentemente, possiamo scrivere programmi che sfruttano il parallelismo in tutta tranquillità, fornisce il supporto fondamentale per l'esecuzione e la comunicazione
- ***facilità*** – la gestione della memoria si basa sulla garbage collection e il feeling generale del linguaggio è da linguaggio interpretato, senza le asprezze tipiche del lavoro col compilatore di un linguaggio strettamente tipizzato.

Parole chiave

Le seguenti parole chiave sono riservate e non possono essere utilizzate come nomi di identificatori.

```
 break   func   interface   select   default
case  defer go    map    struct
chan  else  goto  package     switch
const fallthrough      if     range type
continue    for    import      return      var
```

Concorrenza GO

La concorrenza nel linguaggio e gestita con i canali e le goroutine.

Un esempio di programma Go

```
package main    //dichiarazione del package
import "fmt"    //import della standard library

func add(wait chan bool, x*int){  //dichiarazione di funzione con parametri
//un canale e un puntatore
*x++    //incrementa di uno la variabile      x
wait <- true    //invia true sul canale wait
}

func main(){
wait := make(chan bool, 1)     //crea un canale di bool con size 1
x:=0    //dichiara una variabile x inizializzata a 0
go add(wait, &x)        // goroutine add
<-wait  //attende un output su wait
x++     //incrementa x di 1
fmt.Println(x)
}
```

Non si ha race condition nell'incremento di x perchè l'input da canale è bloccante, ottenendo così una specie di semaforo. L'esecuzione di questo programma stamperà sempre 2.

Costrutti sulla concorrenza

- ch := make(chan string) crea un nuovo canale ch ha tipo string
- <-c legge un valore dal canale c.
- ch<- <valore> inserisce il valore c nel canale ch.
- go f(<valore1>, <valore2>...) genera una nuova goroutine f. Un programma Go termina quando termina il main, quindi eventuali goroutine che stanno ancora eseguendo possono essere terminate improvvisamente

Goroutines

Una goroutine è una funzione che è in grado di funzionare contemporaneamente 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)
}
```

Con una goroutine torniamo immediatamente alla riga successiva e non aspettiamo che la funzione venga completata. Questo è il motivo per cui Scanln è stata inclusa la chiamata alla funzione; senza di esso il programma uscirebbe prima di avere la possibilità di stampare tutti i numeri.

Le goroutine sono leggere e possiamo facilmente crearne migliaia. Possiamo modificare il nostro programma per eseguire 10 goroutine facendo così:

```
func main() {
  for i := 0; i < 10; i++ {
    go f(i)
  }
  var input string
  fmt.Scanln(&input)
}
```

Canali

I canali forniscono un modo per due goroutine di comunicare tra loro e sincronizzare la loro esecuzione. Ecco un programma di esempio che utilizza i canali:

```
package main

import (  "fmt"
  "time"
)

func pinger(c chan string) {
  for i := 0; ; i++ {
    c <- "ping"
  }
}

func printer(c chan string) {
  for {
    msg := <- c
    fmt.Println(msg)
    time.Sleep(time.Second * 1)
  }
}

func main() {
  var c chan string = make(chan string)

  go pinger(c)
  go printer(c)

  var input string
  fmt.Scanln(&input)
}
```

Questo programma stamperà "ping" per sempre Un tipo di canale è rappresentato con la parola chiave `chan` seguita dal tipo di cose che vengono trasmesse sul canale (in questo caso stiamo passando stringhe). L' `<-` operatore (freccia sinistra) viene utilizzato per inviare e ricevere messaggi sul canale. `c <- "ping"` significa inviare `"ping"`. `msg := <- c` significa ricevere un messaggio e memorizzarlo in `msg`. La `fmt` potrebbe anche essere scritta in questo modo: `fmt.Println(<-c)` .

L'uso di un canale come questo sincronizza le due goroutine. Quando pingertenta di inviare un messaggio sul canale, aspetterà che `printer` sia pronto per ricevere il messaggio. (questo è noto come blocco) Aggiungiamo un altro mittente al programma e vediamo cosa succede.

```
func ponger (c chan string) {
  per i: = 0; ; i ++ {
    c <- "pong"
  }
}

func main () {
  var c chan string = make (chan string)

  go pinger (c)
  go ponger (c)
  go printer (c)

  var input string
  fmt.Scanln (&input)
}
```

Ora il programma stampa a turno "ping" e "pong".

Direzione del canale

Possiamo specificare una direzione su un tipo di canale limitandolo quindi a inviare o ricevere. Ad esempio, la firma della funzione di pinger può essere cambiata in questo modo:

```
func pinger (c chan <- string)
```

Ora da c si può solo inviare. Il tentativo di ricevere da c provocherà un errore del compilatore. Allo stesso modo possiamo cambiare printer in questo modo:

```
func printer (c <-chan string)
```

Un canale che non ha queste restrizioni è noto come bidirezionale. Un canale bidirezionale può essere passato a una funzione che richiede canali di sola ricezione o solo ricezione, ma non il contrario.

Select

Go ha una dichiarazione speciale chiamata select che funziona come un switch ma per i canali:

```
func main() {
  c1 := make(chan string)
  c2 := make(chan string)

  go func() {
    for {
      c1 <- "from 1"
      time.Sleep(time.Second * 2)
    }
  }()

  go func() {
    for {
      c2 <- "from 2"
      time.Sleep(time.Second * 3)
    }
  }()

  go func() {
    for {
      select {
      case msg1 := <- c1:
        fmt.Println(msg1)
      case msg2 := <- c2:
        fmt.Println(msg2)
      }
    }
  }()

  var input string
  fmt.Scanln(&input)}
```

`select` seleziona il primo canale che è pronto e riceve da esso (o invia ad esso). Se più di uno dei canali è pronto, seleziona in modo casuale da quale ricevere. Se nessuno dei canali è pronto, l'istruzione viene bloccata finché non si diventa disponibili.

Possiamo anche specificare un defaultcaso:

```
select {
case msg1 := <- c1:
  fmt.Println("Message 1", msg1)
case msg2 := <- c2:
  fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
  fmt.Println("timeout")
default:
  fmt.Println("nothing ready")
}
```

Il caso predefinito avviene immediatamente se nessuno dei canali è pronto.

Canali bufferizzati

È anche possibile passare un secondo parametro alla funzione make quando si crea un canale:

```
package main

import "fmt"

func main() {
       ch := make(chan int,2)
       ch<-3
       ch <- 2

       go fmt.Println(<-ch)
       go fmt.Println(<-ch)
       var input string
       fmt.Scanln(&input)}
```

Un canale bufferizzato è asincrono

La routine che invia si blocca solo quando è stato copiato il valore nel buffer; se il buffer è pieno questo significa aspettare fino a quando il ricevente ha ricevuto il valore. I riceventi si bloccano sempre quando ci sono dati da ricevere

Sitografia

https://www.ionos.it/digitalguide/server/ know-how/golang/

https://www.golang-book.com/books/intro/

https://www.tutorialspoint.com/go/

http://www.dacrema.com/Informatica/Processo.htm

http://www.edutecnica.it/informatica/thread/thread.htm

https://mos4tfa.wordpress.com/teoria/ipc/corse-critiche/

https://www.disi.unige.it/person/DoderoG/SistOp/lu_sem.htm