Golang e modelli di concorrenza

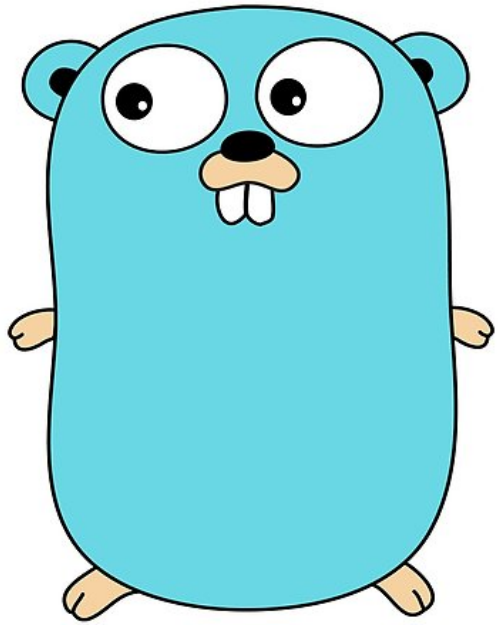
Leonardo Gonfiantini

27 Ottobre 2022

Relatore: Delzanno Giorgio

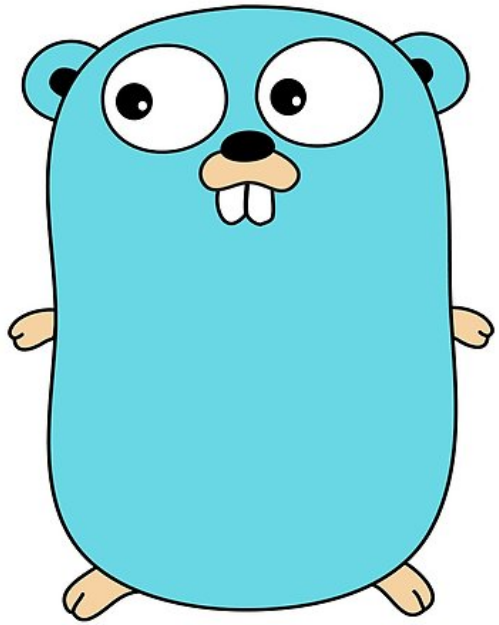
Golang

- linguaggio compilato e tipato staticamente, creato da Google
- sintatticamente simile al C ma con:
  - memory safety
  - garbage collection
  - structural typing
  - CSP-style concurreny
- alte prestazioni per il networking e multiprocessing
- open-source



Golang

- linguaggio compilato e tipato staticamente, creato da Google
- sintatticamente simile al C ma con:
  - memory safety
  - garbage collection
  - structural typing
  - CSP-style concurreny
- alte prestazioni per il networking e multiprocessing
- open-source

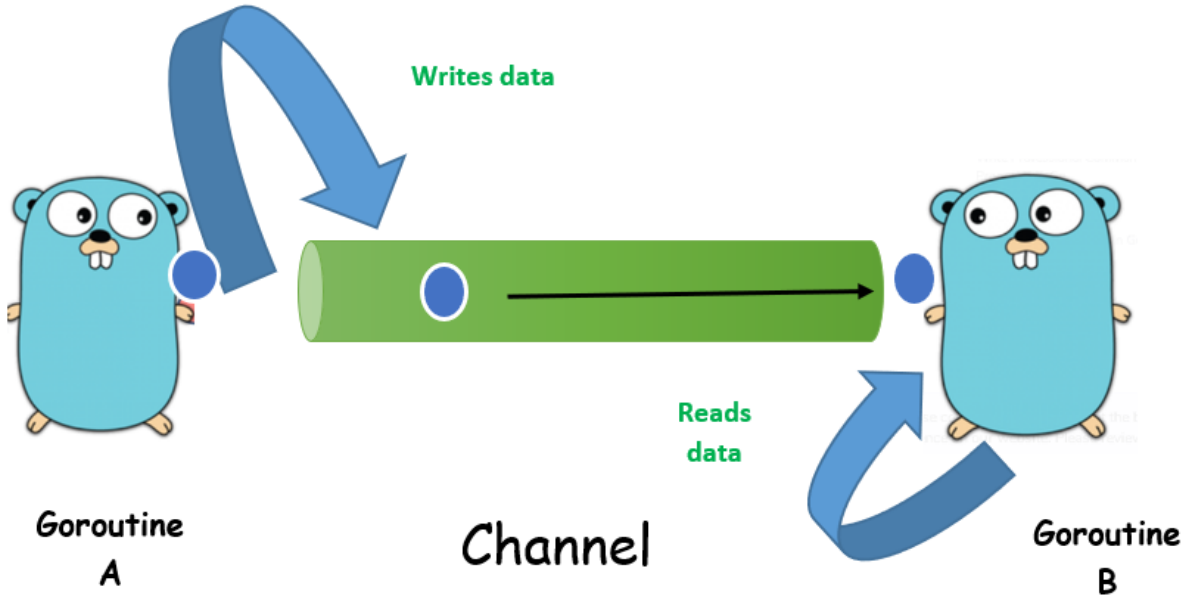


CSP - Communicating sequential processes

"Don't communicate in the form of shared memory. Instead, share memory through communication."

Generalmente nei linguaggi come C, C++ e Java la comunicazione tra thread avviene utilizzando la memoria condivisa, questo Go lo puo fare, ma puo anche utilizzare i modelli di concorrenza del CSP tramite l'utilizzo dei canali.

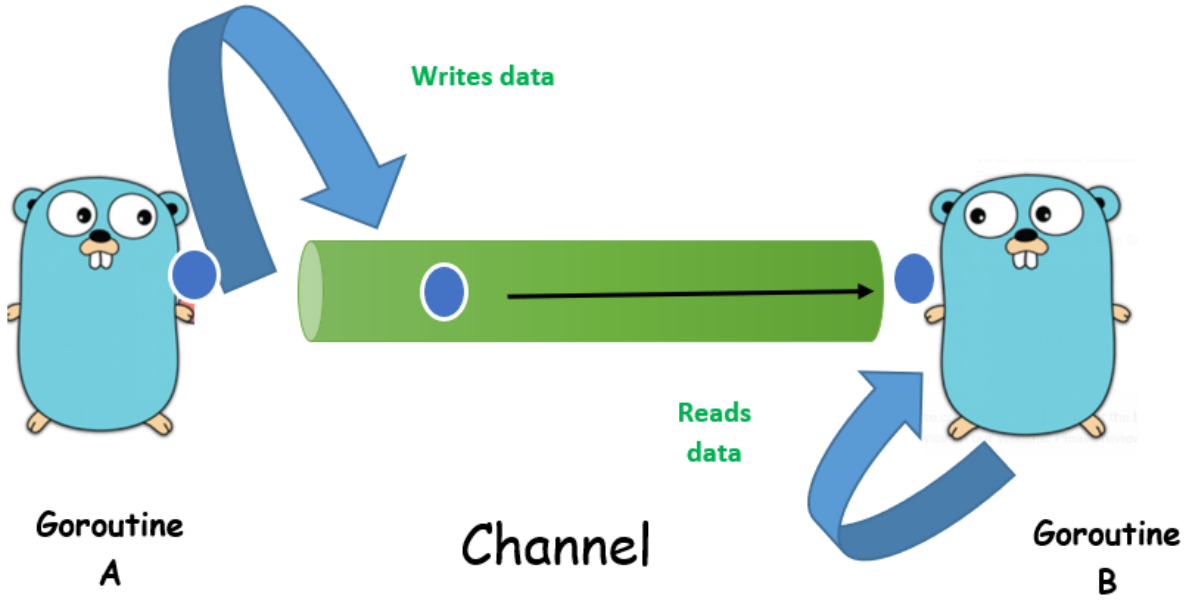
Channel



I canali funzionano come dei tubi alla quale diversi thread (**goroutine**), possono collegarsi, la comunicazione e' bidirezionale per default, il che significa che possiamo ricevere e inviare messaggi dallo stesso canale.

Questo permette alle goroutine di sincronizzarsi non utilizzando lock o condition variable.

Channel



I canali funzionano come dei tubi alla quale diversi thread (**goroutine**), possono collegarsi, la comunicazione e' bidirezionale per default, il che significa che possiamo ricevere e inviare messaggi dallo stesso canale.

Questo permette alle goroutine di sincronizzarsi non utilizzando lock o condition variable.

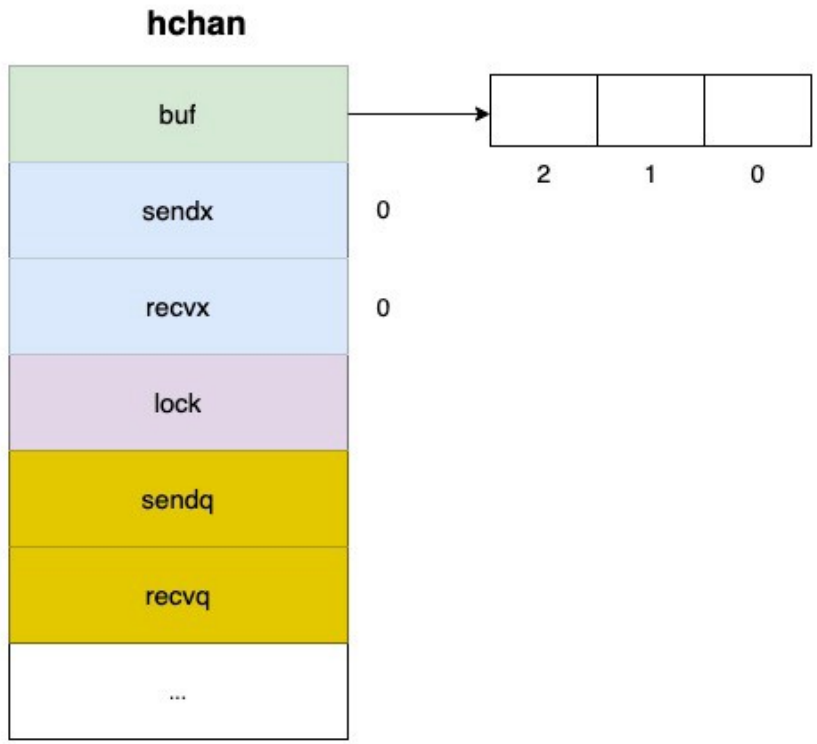
Channel - dichiarazione

```
//unbuffered channel
ch := make(ch int) //init
ch <- 42 //inviamo 42

n := <- ch
//riceviamo dal canale 42

//nel caso di buffered channel
ch := make(ch int, 10)
ch <- 42 ch <- 12 ch <- 3

fmt.Println(<- ch, <- ch, <- ch)
//output = 42 12 3
```



Channel - channel vs mutex

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

type Ball struct {
    hit int
}

func player(name string, table chan *Ball, wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 0; i < 10; i++ {
        ball := <-table
        ball.hit++
        fmt.Println(name, ball.hit)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}

func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    table := make(chan *Ball, 1)

    go player("playerA", table, &wg)
    go player("playerB", table, &wg)

    table <- &Ball{}
    wg.Wait()

}
```

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

type Ball struct {
    hit int
    sync.Mutex
}

func player(name string, b *Ball, wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 0; i < 10; i++ {
        b.Lock()
        b.hit++
        fmt.Println(name, b.hit)
        b.Unlock()
        time.Sleep(100 * time.Millisecond)
    }
}

func main() {
    var b Ball
    wg := sync.WaitGroup{}
    wg.Add(2)

    go player("playerA", &b, &wg)
    go player("playerB", &b, &wg)

    wg.Wait()

}
```

Channel - pro e contro

Mutex e channel sono strumenti diversi, i mutex sono utili per accedere ad una risorsa in modo sequenziale e performante, i channel per gestire la comunicazione tra diverse goroutine.

Quando usare i channel?

- passare la ownership del dato
- distribuire unita di lavoro (**worker**)
- comunicare risultati asincroni

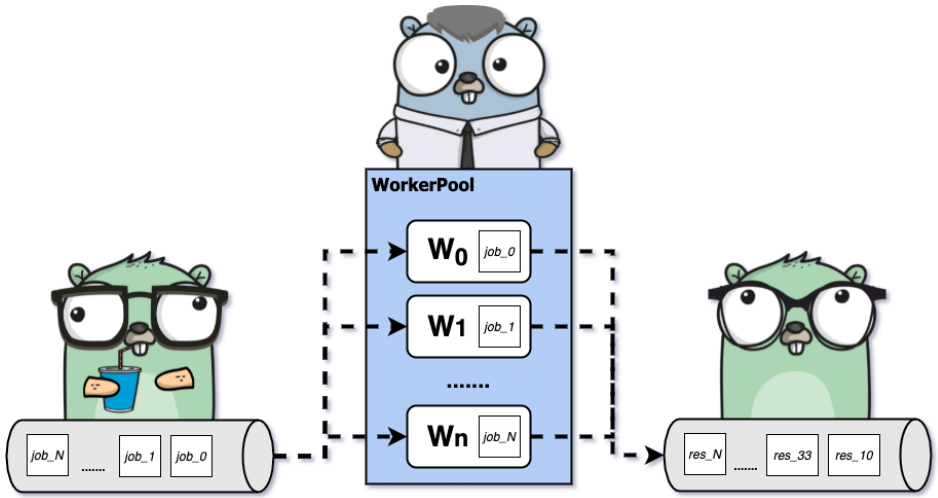
Channel - pro e contro

Mutex e channel sono strumenti diversi, i mutex sono utili per accedere ad una risorsa in modo sequenziale e performante, i channel per gestire la comunicazione tra diverse goroutine.

Quando usare i channel?

- passare la ownership del dato
- distribuire unita di lavoro (**worker**)
- comunicare risultati asincroni

Worker pattern



La worker pool viene utilizzata per assegnare un numero n di Tasks ad un numero m di Workers. Le task dovranno stare in una queue, quando un Worker finisce una task controlla se nella queue sono presenti altre task, se si, procedera con la nuova task altrimenti aspettera.

Worker pattern - implementazione

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

//il canale viene usato solo per ricevere il messaggio nel caso di jobs
//il canale viene usato solo per inviare il messaggio nel caso di results
func worker(id int, jobs <-chan int, results chan<- int, wg *sync.WaitGroup) {
    defer wg.Done()
    for j := range jobs {
        fmt.Println("worker", id, " => job num:", j)
        time.Sleep(time.Second)
        results <- j * 42
    }
}

func main() {

    wg := sync.WaitGroup{}
    wg.Add(2)

    job := make(chan int, 10)
    result := make(chan int, 10)

    for w := 1; w <= 2; w++ {
        go worker(w, job, result, &wg)
    }

    for j:= 1; j <= 9; j++ {
        job <- j
    }

    close(job) //chiudo il canale

    wg.Wait() //aspetto che i worker abbiano finito

    for r := 1; r <= 9; r++ {
        res := <-result
        fmt.Println("result for number", r, "=>", res)
    }

    close(result)
}
```

```
worker 2  => job num: 1
worker 1  => job num: 2
worker 2  => job num: 4
worker 1  => job num: 3
worker 2  => job num: 5
worker 1  => job num: 6
worker 1  => job num: 7
worker 2  => job num: 8
worker 2  => job num: 9
result for number 1 => 84
result for number 2 => 42
result for number 3 => 168
result for number 4 => 126
result for number 5 => 252
result for number 6 => 210
result for number 7 => 336
result for number 8 => 294
result for number 9 => 378
```

Pooling pattern

Il Pool pattern mostra come utilizzare un canale per creare una pool di risorse condivise che possono essere utilizzate da diverse goroutine singolarmente.
Quando una goroutine ha bisogno di una risorsa dalla pool, viene acquisita, usata, e ritornata alla pool.

Pooling pattern - implementazione

Riferimenti