MATERI

Hari 12 - Goroutine & Channel

Goroutine

Sebelum mempelajari goroutine sangat disarankan untuk membaca artikel pada link ini terlebih dahulu untuk memahami apa itu thread dan concurrency

Apa itu Goroutine?

Goroutine adalah lightweight thread yang di manage oleh Go runtime. Untuk membuat sebuah Goroutine kita hanya memerlukan 2kb memori. Goroutine memiliki sifat yang asynchronous jadi tidak saling menunggu dengan Goroutine lain.

Untuk membuat goroutine yang sederhana cukup mudah. Pembuatan goroutine baru di awali dengan sintaks “go” di ikuti dengan nama method maupun fungsi. Method yang di awali dengan keyword go artinya method tersebut di anggap sebuah goroutine. berikut contoh penggunaanya:

```
package main

import (
  "fmt"
  "time"
)

func say(s string) {
  for i := 0; i < 5; i++ {
   time.Sleep(100 * time.Millisecond)
   fmt.Println(s)
  }
}
```

```
func main() {
  go say("world")
  say("hello")
}
```

jika kita run kode diatas maka outputnya bisa muncul terlebih dahulu world ataupun hello dan lain sebagainya, kenapa seperti itu karena pada function say("world") sudah menggunakan goroutine yang artinya function say("world") berjalan di thread lain yang berbeda dengan say("hello")

Channel

Channel adalah penghubung antara goroutine satu dengan goroutine lainnya. Mekanisme channel yaitu serah terima, dimana ada yang mengirim dan ada yang menerima. Jika di contohnya di dalam kehidupan nyata seperti pipa yang berisi air, dimana terdapat ujung pengirim air dan ujung sebagai penerima air.

Deklarasi Channel

Setiap channel dapat terkait dengan channel lainnya. Dan channel dapat di cetak dengan tipe data tertentu. Cara mendeklarasikan channel seperti mendeklarasikan tipe data lainnya. Tipe data nya yaitu **chan**. misalnya : chan b, artinya variable bertipe channel.

Namun untuk membuat channel baru dapat menggunakan sintaks di awal yaitu make.

```
angka :=make(chan int).
```

Secara default channel mempunyai nilai nil. Maka dari itu channel harus di keklarasikan seperti di atas yaitu menggunakan awalan **make**.

karena channel mempunyai mekanisme kirim dan menerima itu berarti kedua mekanisme tersebut harus di deklarasikan juga

Cara deklarasi memberi dan menerima nilai channel yaitu dengan tanda <-(**arrow**).

```
// untuk mengirim
a <- 10

//untuk menerima
angka := <- a
```

berikut ini contoh penggunaan channel:

```
package main

import "fmt"

func cetak(ch chan int) {
  fmt.Println("ini dari goroutine...")
  ch <- 10
}

func main() {
  angka:= make(chan int)
  go cetak(angka)
  nilai := <-angka
  fmt.Println("nilai channel integer :",
nilai)
  fmt.Println("ini dari function main...")
}
```

dapat terlihat pada contoh diatas pada function cetak terdapat sintaks ch <- 10 yang berarti channel ch mengirim nilai 10, dan dapat terlihat juga di function main terdapat sintaks nilai := <- angka, yang dimana nilai menerima data atau value dari channel angka

Menutup Channel

Ketika kita bicara channel dua arah yaitu pengirim dan penerima maka sebenarnya ada hak lain yang dapat di lakukan. Pengirim mempunyai kemampuan menutup channel untuk memberi tahu penerima bahwa tidak ada data lagi.

Penerima dapat memberikan tambahan kondisi apakah channel tersebut sudah di tutup ataupun belum.

berikut ini contohnya:

```
package main

import (
  "fmt"
)

func cetak(ch chan<- int) {
  for index := 0; index < 10; index++ {
    ch <- index
  }
  close(ch)
}

func main() {
  ch := make(chan int)

  go cetak(ch)

  for {
    data, ok := <-ch
    if ok == false {
      break
    }
    fmt.Printf("Data di terima %v\n", data)
  }
}
```

Perhatikan kode program di atas, terdapat 1 goroutine dengan nama fungsi **cetak()**, dimana fungsi tersebut melakukan perulangan sebanyak 10 kali dan di dalam perulangan tersebut data channel di berikan data. Setelah data selesai dikirim maka channel ditutup.

Data yang di tampilkan dengan infinity loop(perulangan terus-menerus). Penerima channel akan memeriksa apakah channel sudah di tutup dengan identifikasi data masih ada ataupun tidak.

Channel Buffer

channel secara ***default*** di lakukan secara **unbufferd**, artinya hanya data yang di kirim satu per satu melalui **channel**. Sedangkan dengan channel buffer kita dapat melakukan pengiriman data lebih daru satu. unbuffered bukan hanya pengirim saja yang di block, melainkan penerimanya juga di block ketika data sebelumnya belum selesai. dengan menggunakan channel buffer ini memungkinkan kita mengirim dan menerima banyak permintaan.

Membuat Channel Buffer

Untuk membuat channel buffer harus menentukan jumlah kapasitas data yang bisa di lakukan secara bersamaan dengan format :

Contohnya:

```
ch := make(chan int, 1)
```

Perhitungan kapasitas channel seperti array yaitu di mulai dari 0. Ketika kita mengisi nilai buffer yaitu 2 maka akan ada proses yang dapat di lakukan bersamaan sejumlah 2. Namun akan lebih baik ketika kita memberikan kapasitas nilai buffer sama dengan jumlah data yang ingin di kirim.

Mari kita buat contoh penggunaan channel buffered yang sederhana.

```
package main

import (
  "fmt"
)

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

  ch <- 6
  ch <- 7
  ch <- 5

  fmt.Println(<-ch)
  fmt.Println(<-ch)
  fmt.Println(<-ch)
}
```

Channel Select

Select adalah proses untuk menvalidasi kondisi ketika mempunyai lebih dari satu channel dalam satu goroutine yang jalan. Bisa di katakan select pemilihan terhadap channel. Ketika menggunakan 2 channel konsep sudah di ketahui bahwa proses nya kirim dan menerima. Select berperan untuk memilih channel yang di terima di dalam goroutine. Cara kerjanya seperti ini, asumsikan kita punya 2 channel yang jalan, nah ketika data sudah terkirim beberapa data, tugas select ini memilih penerimaan channel yang jalan. Cara kerja select sebenarnya sama seperti suatu kondisi switch, dimana switch digunakan untuk melakukan seleksi terhadap kondisi.

Penerapan Select

Program berikut merupakan contoh sederhana penerapan select dalam channel. Dipersiapkan 2 buah goroutine, satu untuk pencarian rata-rata, dan satu untuk nilai tertinggi. Hasil operasi di masing-masing goroutine dikirimkan ke fungsi main() via channel (ada dua channel). Di fungsi main() sendiri, data tersebut diterima dengan memanfaatkan keyword select.

Ok, langsung saja kita praktek. Pertama, siapkan 2 fungsi yang sudah dibahas di atas. Fungsi pertama digunakan untuk mencari rata-rata, dan fungsi kedua untuk penentuan nilai tertinggi dari sebuah slice.

```
package main

import "fmt"

func getAverage(numbers []int, ch chan
float64) {
  var sum = 0
  for _, e := range numbers {
    sum += e
  }
  ch <- float64(sum) / float64(len(numbers))
}

func getMax(numbers []int, ch chan int) {
  var max = numbers[0]
  for _, e := range numbers {
    if max < e {
      max = e
    }
  }
  ch <- max
}
```

Kedua fungsi tersebut dijalankan sebagai goroutine. Di akhir blok masing-masing fungsi, hasil kalkulasi dikirimkan via channel yang sudah dipersiapkan, yaitu ch1 untuk menampung data rata-rata, ch2 untuk data nilai tertinggi.

Ok lanjut, buat implementasinya pada fungsi main().

```
  fmt.Println("numbers :", numbers)

  var ch1 = make(chan float64)
  go getAverage(numbers, ch1)

  var ch2 = make(chan int)
  go getMax(numbers, ch2)

  for i := 0; i < 2; i++ {
    select {
    case avg := <-ch1:
      fmt.Printf("Avg \t: %.2f \n", avg)
    case max := <-ch2:
      fmt.Printf("Max \t: %d \n", max)
    }
  }
}
```

Pada kode di atas, pengiriman data pada channel ch1 dan ch2 dikontrol menggunakan select. Terdapat 2 buah case kondisi penerimaan data dari kedua channel tersebut.

- Kondisi case avg := <-ch1 akan terpenuhi ketika ada penerimaan data dari channel ch1, yang kemudian akan ditampung oleh variabel avg.
- Kondisi case max := <-ch2 akan terpenuhi ketika ada penerimaan data dari channel ch2, yang kemudian akan ditampung oleh variabel max.

Karena ada 2 buah channel, maka perlu disiapkan perulangan 2 kali sebelum penggunaan keyword select.

WaitGroup

Jika di artikan bahasa Indonesia Wait dan Group bisa di artikan menunggu kelompok. Nah kelompok yang di maksud yaitu gorotuine. Sehingga **WaitGroup** adalah mekanisme digolang yang berfungsi untuk melakukan sinkronisasi antara goroutine. Beda dengan **Channel** ya, kalau channel digunakan untuk komunikasi antar goroutine.

Masih ingat mengenai ketika awal belajar goroutine, dimana kita menggunakan fungsi **sleep** untuk menunggu goroutine.

Nah dengan mneggunakan waitgroup kita tidak perlu lagi menggakan fungsi sleep.

Berikut ini manfaat menggunakan goroutine.

- Menambahkan Goroutine yang jalan.
- Menunggu / Menunda Goroutine Jalan.
- Dan mengakiri Goroutine yang jalan.

Kenapa harus menggunakan WaitGroup ?

Ketika kita ingin melakukan proses secara concurency maka harus satu persatu proses selesai di dalam goroutine, entah goroutine itu berjalan berapa kali. Maka solusi pertama mengunakan **sleep**. Namun dengan menggunakan sleep sama saja kita memaksakan goroutine berhenti dalam waktu tertentu. Hal ini tentu akan membuat program lebih lambat.

Sebagai contoh di bawah ini program yang menerapkan penggunaan sleep untuk menggunakan goroutine.

```
package main

import (
  "fmt"
  "time"
)

func printText(text string){
  for i:=0; i<5; i++{
    fmt.Println(text)
  }
}

func main(){
  go printText("Halo")
  go printText("Dunia")
  time.Sleep(500 * time.Millisecond)
}
```

Jika di lihat dari hasil terhadap nilai sudah benar, namun jika kita lihat dari penggunaan kode yang benar dan kecepatan data hal ini tidak di rekomendasikan. Karena Data di cetak dengan jeda waktu 500 millisecond.

Sekarang jika udah mengetahui penggunaan di atas dan di rasa kurang baik maka saat menggunakan waitGroup.

Cara menggunakan WaitGroup

- Wait(), digunakan untuk menunggu proses berjalan.
- Done(), digunakan untuk menandai bahwa semua goroutine sudah selesai.

Yuk langsung saja kita bikin contohnya.

Contoh kode berikut ini mengadopsi dari kode sebelumnya, namun di rubah ke bentuk WaitGroup.

```
package main

import (
  "fmt"
  "sync"
)

func printText(text string, wg *sync.WaitGroup){
  for i:=0; i<5; i++{
    fmt.Println(text)
  }
  wg.Done()
}

func main(){
  var wg sync.WaitGroup

  wg.Add(1)
  go printText("Halo", &wg)

  wg.Add(1)
  go printText("Dunia", &wg)

  wg.Wait()
}
```

perhatikan kode diatas, dengan memanfaatkan sync.WaitGroup kita tidak perlu untuk menyisipkan time.Sleep lagi agar semua goroutine dapat terlihat hasil printnya.

Referensi Tulisan:

- https://dev.to/fncolon/apa-itu-thread-dan-concurrency-dan-perbedaan-multiprocessing-multiprogramming-dengan-multithreading-1n7k
- https://kodingin.com/golang-goroutine/
- https://kodingin.com/golang-channel/
- https://kodingin.com/golang-buffered-channel/
- https://kodingin.com/golang-select-channel/
- https://kodingin.com/golang-waitgroup/
- https://dasarpemrogramangolang.novalagung.com/A-goroutine.html
- https://dasarpemrogramangolang.novalagung.com/A-channel.html
- https://dasarpemrogramangolang.novalagung.com/A-buffered-channel.html
- https://dasarpemrogramangolang.novalagung.com/A-channel-select.html
- https://dasarpemrogramangolang.novalagung.com/A-channel-range-close.html
- https://tour.golang.org/concurrency/1
- https://tour.golang.org/concurrency/2
- https://tour.golang.org/concurrency/3
- https://tour.golang.org/concurrency/4
- https://tour.golang.org/concurrency/5
- https://tour.golang.org/concurrency/6