Scalable Web Service With Golang sesi 5

Channels

Channels

Channel merupakan sebuah mekanisme untuk *Goroutine* saling berkomunikasi dengan *Goroutine* lainnya. Maksud berkomunikasi disini adalah proses pengiriman dan pertukaran data antara *Goroutine* satu dengan *Goroutine* lainnya.

Untuk membuat sebuah *Goroutine,* kita memerlukan function *make* yang merupakan sebuah *built-in function* dari bahasa Go. Contohnya seperti pada gambar dibawah ini.

```
func main() {
  c := make(chan string)
}
```

Variable *c* pada gambar diatas merupakan sebuah variable yang memiliki tipe data *channel of string*. Maksud channel of string disini adalah sebuah *channel* yang memiliki tipe data *string*. Lalu keyword *chan* merupakan keyword untuk membuat *channel*.

Channels (Channel operators)

Kita membutuhkan operator dari *channel* agar *channel* tersebut dapat dijadikan sebagai alat untuk berkomunikasi antara *Goroutine* dengan yang lainnya. Contohnya seperti pada gambar dibawah ini.

```
func main() {
  c := make(chan string)

  //Mengirim data kepada channel
  c <- value

  //Menerima data dari channel

  result := <- c
}
```

Tanda <- merupakan sebuah operator dari *channel* untuk proses pengiriman data dari *Goroutine* satu dengan yang lainnya.

Lalu maksud dari penulisan **c <- data** pada gambar diatas adalah kita mengirimkan data kepada channel *c*.

Dan yang terakhir maksud dari penulisan **result := <-c** adalah kita menerima data dari channel *c* dan data tersebut kita simpan di dalam variable *result.* Perlu diingat disini bahwa proses pengiriman dan penerimaan data dari *channel* bersifat synchronous.

Channels (Implementing channel)

Sekarang kita akan mulai mengimplementasikan *channel* untuk proses komunikasi *Goroutine*.

Contohnya seperti pada gambar di sebelah kanan. Kita membuat sebuah *channel of string* yang disimpan pada variable *c* pada line 7.

Lalu kita membuat sebuah function bernama *introduce* yang menerima 2 parameter dengan tipe data *string* dan *channel of string*.

Function *introduce* digunakan untuk membuat pesan dengan kalimat yang kita buat menggunakan function *fmt.Sprintf* pada line 28.

Perhatikan pada line 30, penulisan c <- result maksudnya adalah kita mengirimkan nilai yang dikandung oleh variable *result* kepada channel *c*. Lalu dari 9 - 13 kita membuat 3 *Goroutine* yang dimana kita memanggil function *introduce* sebanyak 3 kali dengan keyword *go,* dan kita mengirimkan nilai dengan tipe data *string* yang berbeda-beda dan kita juga mengirimkan channel *c* .

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7    c := make(chan string)
8
9    go introduce("Airell", c)
10
11   go introduce("Nanda", c)
12
13   go introduce("Mailo", c)
14
15   msg1 := <-c
16   fmt.Println(msg1)
17
18   msg2 := <-c
19   fmt.Println(msg2)
20
21   msg3 := <-c
22   fmt.Println(msg3)
23
24   close(c)
25 }
26
27 func introduce(student string, c chan string) {
28   result := fmt.Sprintf("Hai, my name is %s", student)
29
30   c <- result
31 }
```

Channels (Implementing channel)

Lalu perhatikan pada line 15, 18, dan 21.

Karena kita membutuhkan data yang telah dikirimkan oleh function *introduce* melalui channel *c,* maka kita perlu menggunakan menggunakan operator <- c yang artinya adalah function *main* menerima data dari function *introduce* dan disimpan ke pada 3 variable yaitu *msg1*, *msg2*, dan *msg3*.

Lalu function *close* merupakan sebuah *function* yang digunakan untuk mengindikasikan bahwa sebuah channel sudah tidak digunakan untuk berkomunikasi lagi. Perlu diingat bahwa cara ini perlu dilakukan.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7    c := make(chan string)
8
9    go introduce("Airell", c)
10
11   go introduce("Nanda", c)
12
13   go introduce("Mailo", c)
14
15   msg1 := <-c
16   fmt.Println(msg1)
17
18   msg2 := <-c
19   fmt.Println(msg2)
20
21   msg3 := <-c
22   fmt.Println(msg3)
23
24   close(c)
25 }
26
27 func introduce(student string, c chan string) {
28   result := fmt.Sprintf("Hai, my name is %s", student)
29
30   c <- result
31 }
```

Channels (Implementing channel)

Karena function *main* juga merupakan sebuah *Goroutine*, maka syntax pada gambar di kanan merupakan contoh dari proses komunikasi antara Goroutine *main* dengan Goroutine *introduce* melalui *channel*.

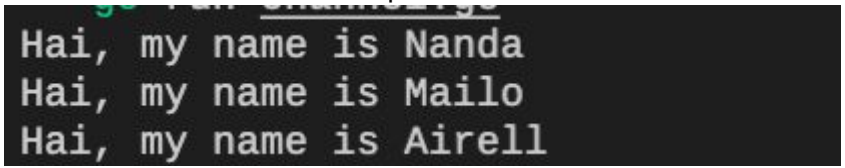
Lalu kita membuat 3 variable untuk penerimaan data dari channel *c* pada function *main* karena kita membuat 3 *Goroutine* yang dimana di tiap *Goroutine* tersebut mengirimkan sebuah data.

Kemudian jika kita jalankan pada terminal kita, maka hasilnya dapat berbeda-beda urutannya. Contoh gambar hasilnya dapat dilihat pada halaman selanjutnya.

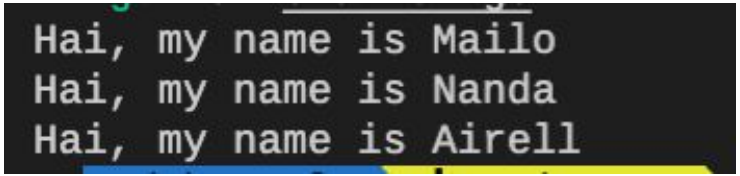
```go
package main

import "fmt"

func main() {

  c := make(chan string)

  go introduce("Airell", c)

  go introduce("Nanda", c)

  go introduce("Mailo", c)

  msg1 := <-c
  fmt.Println(msg1)

  msg2 := <-c
  fmt.Println(msg2)

  msg3 := <-c
  fmt.Println(msg3)

  close(c)
}

func introduce(student string, c chan string) {
  result := fmt.Sprintf("Hai, my name is %s", student)

  c <- result
}
```

Channels (Implementing channel)

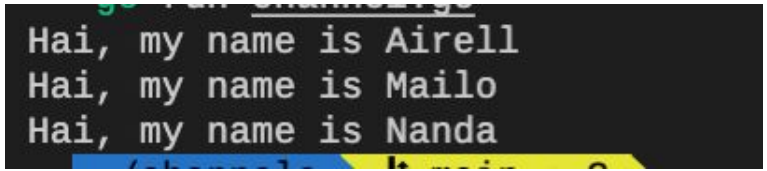
Eksekusi pertama



Eksekusi kedua



Eksekusi ketiga



Channels (Implementing channel #4)

Jika kita lihat pada pada contoh hasil pada halaman sebelumnya, di tiap eksekusi memiliki menampilkan hasil yang berbeda-beda. Ini terjadi karena seperti yang pernah kita bahas pada materi sebelumnya bahwa, *Goroutine* bekerja secara asynchronous yang dimana *Goroutine* satu dengan yang lainnya tidak akan saling menunggu.

Dan *Goroutine* merupakan cara agar kita dapat membuat concurrency pada bahasa Go, dan dalam concurrency, kita tidak akan tahu mana yang akan selesai tereksekusi terlebih dahulu. Ini lah yang menjadi penyebab mengapa di tiap eksekusi menampilkan hasil yang berbeda-beda.

Channels (Channel with anonymous function)

Sekarang kita akan mengubah codingan kita dari halaman sebelumnya. Kita akan menggunakan *range loop* untuk melooping data-data yang kita simpan pada sebuah *slice of string* sekaligus memanggil 3 *Goroutine*. Contohnya seperti pada gambar di sebelah kanan.

Sekarang function *introduce* kita ganti dengan sebuah function *anonymous* yang kita letakkan di dalam range loop.

Lalu untuk proses penerimaan datanya, kita membuat function bernama *print* yang akan menerima data melalui channel *c.* Perhatikan pada line 28, ketika kita tidak ingin menampung sebuah kiriman data dari *channel*, maka kita bisa langsung menuliskan operator nya secara langsung <- c.

Jika kita jalankan maka hasilnya akan sama seperti sebelumnya, yaitu di tiap eksekusi hasilnya akan berbeda-beda.

```go
package main

import "fmt"

func main() {

  c := make(chan string)

  students := []string{"Airell", "Mailo", "Indah"}

  for _, v := range students {
    go func(student string) {
      fmt.Println("Student", student)
      result := fmt.Sprintf("Hai, my name is %s", student)
      c <- result
    }(v)
  }

  for i := 1; i <= 3; i++ {
    print(c)
  }

  close(c)
}


func print(c chan string) {
  fmt.Println(<-c)
}
```

Channels (Channel directions)

Ketika kita menggunakan *channel* sebagai sebuah parameter dari sebuah *function,* kita dapat menentukan apakah *channel* tersebut digunakan untuk menerima data saja, mengirim data saja, ataupun menerima sekaligus mengirim data.

Channel directions ini memiliki sifat yang opsional, dan digunakan untuk kepentingan *type-safety*. Gambar dibawah ini merupakan penjelasan dari *channel direction*.

Parameter Syntax	**Detail**
c chan string	parameter c dapat digunakan untuk mengirim dan menerima data
c chan <- string	parameter c hanya dapat digunakan untuk mengirim data
c <- chan string	parameter c hanya dapat digunakan untuk menerima data

Channels (Channel directions)

Sekarang mari kita terapkan *channel direction* pada codingan kita yang sebelumnya. Untuk sekarang kita akan menggunakan kembali function *introduce*. Contohnya seperti pada gambar di sebelah kanan.

Perhatikan pada line 22, karena parameter *channel* pada function *print* hanya digunakan untuk menerima data, maka penulisan parameternya kita ubah menjadi **c <- chan string**.

Lalu pada line 26, karena parameter *channel* pada function *introduce* hanya digunakan untuk mengirim data, maka penulisan parameternya kita ubah menjadi **c chan <- string**.

Perlu diingat kembali disini bahwa penggunaan *channel direction* adalah opsional.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7    c := make(chan string)
8
9    students := []string{"Airell", "Mailo", "Indah"}
10
11   for _, v := range students {
12     go introduce(v, c)
13   }
14
15   for i := 1; i <= 3; i++ {
16     print(c)
17   }
18
19   close(c)
20 }
21
22 func print(c <-chan string) {
23   fmt.Println(<-c)
24 }
25
26 func introduce(student string, c chan<- string) {
27   result := fmt.Sprintf("Hai, my name is %s", student)
28   c <- result
29 }
```

Channels (Buffered vs unbuffered channel)

Pada dasarnya, channel bersifat *unbuffered* atau tidak di buffer di memori. *Channel* yang kita gunakan pada halaman-halaman sebelumnya merupakan *unbuffered channel* yang dimana proses penerimaan dan pengiriman data bersifat *synchronous*.

Lain halnya dengan *unbuffered channel* yang dimana kita dapat menentukkan kapasitas dari *buffer* nya, dan selama jumlah data yang dikirim melalui *unbuffered channel* tidak melebihi kapasitasnya, maka proses pengiriman data akan bersifat asynchronous.

```
func main() {
 c := make(chan int, 3)
}
```

Gambar diatas merupakan cara untuk membuat *unbuffered channel* dengan memberikan jumlah kapasitas buffer pada argumen kedua dari function *make.*

UnBuffered Channel (Buffered vs unbuffered channel)

Agar lebih jelas, mari kita coba melihat perbedaan antara *buffered* dengan *unbuffered channel*. Pertama-tama kita akan memulai dulu dari buffered channel.

Perhatikan pada gambar dikanan, variable c1 pada line 63 merupakan sebuah *unbuffered channel*. Lalu kita membuat sebuah *Goroutine* berupa *function anonymous* pada line 65 - line 69.

Ketika *Goroutine* tersebut dijalankan, maka text yang terdapat pada line 66 akan ditampilkan dan setelah *Goroutine* tersebut mengirimkan data melalui *channel* nya, maka text pada line 68 akan ditampilkan.

Kemudian pada line 72, kita menggunakan function *time.Sleep* untuk memberikan jeda selama 2 detik terhadap penerimaan data melalui *channel* pada line 75.

Kemudian pada line 79, kita juga memberikan jeda selama 1 detik sebelum function *main* menghentikan eksekusinya.

```
54  package main
55
56  import (
57    "fmt"
58    "time"
59  )
60
61  func main() {
62
63    c1 := make(chan int)
64
65    go func(c chan int) {
66      fmt.Println("func goroutine starts sending data into the channel")
67      c <- 10
68      fmt.Println("func goroutine after sending data into the channel")
69    }(c1)
70
71    fmt.Println("main goroutine sleeps for 2 seconds")
72    time.Sleep(time.Second * 2)
73
74    fmt.Println("main goroutine starts receiving data")
75    d := <-c1
76    fmt.Println("main goroutine received data:", d)
77
78    close(c1)
79    time.Sleep(time.Second)
80  }
```

UnBuffered Channel (Buffered vs unbuffered channel)

```
main goroutine sleeps for 2 seconds
func goroutine starts sending data into the channel
main goroutine starts receiving data
main goroutine received data: 10
func goroutine after sending data into the channel
```

Kemudian ketika di eksekusi, maka hasilnya akan seperti pada gambar di atas.

Perhatikan hasilnya, tulisan berupa **func goroutine after sending data into the channel** seharusnya ditampilkan setelah tulisan **func goroutine starts sending data into the channel,** tetapi faktanya tulisan **func goroutine after sending data into the channel** ditampilkan diakhir.

Hal ini terjadi karena syntax apapun yang berada dibawah proses pengiriman data melalui *unbuffered channel* akan tertahan hingga datanya diterima oleh *Goroutine* lainnya.

Jika kita perhatikan pada gambar di halaman sebelumnya, proses penerimaan data pada function *main* tertahan selama 2 detik sehingga. Karena syntax untuk menampilkan tulisan **func goroutine after sending data into the channel** berada dibawah proses pengiriman data, maka dari itu tulisan tersebut ditampilkan setelah datanya di terima pada function *main*.

Buffered Channel (Buffered vs unbuffered channel)

Sekarang kita akan mulai menggunakan *Buffered Channel.* Perhatikan pada line 92, variable *c1* merupakan sebuah *buffered channel* dengan jumlah kapasitas 3.

Lalu pada line 94 - 103, terdapat sebuah *Goroutine* yang mengunakan looping untuk mengirimkan data melalui *channel* sebanyak 5 kali. Kemudian pada line 106, kita menggunakan *time.Sleep* kembali untuk memberikan jeda terhadap proses penerimaan data pada line 108 - 110.

Terdapat hal baru saat ini yaitu *range loop* yang terdapat pada line 108-110. Agar lebih mudah dalam menerima data yang banyak dari *channel*, maka kita dapat melakukan *range loop* yang akan melakukan looping sebanyak data yang akan diterima dari channel pengirim (line 97).

Range loop terhadap sebuah *channel* tersebut sama saja seperti kita membuat variable *c1* sebanyak 5 kali.

```
84  package main
85
86  import (
87    "fmt"
88    "time"
89  )
90
91  func main() {
92    c1 := make(chan int, 3)
93
94    go func(c chan int) {
95      for i := 1; i <= 5; i++ {
96        fmt.Printf("func goroutine #%d starts sending data into the channel\n", i)
97        c <- i
98        fmt.Printf("func goroutine #%d after sending data into the channel\n", i)
99      }
100
101     close(c)
102
103   }(c1)
104
105   fmt.Println("main goroutine sleeps 2 seconds")
106   time.Sleep(time.Second * 2)
107
108   for v := range c1 { // v  = <- c1
109     fmt.Println("main goroutine received value from channel:", v)
110   }
111 }
```

Buffered Channel (Buffered vs unbuffered channel)

```
main goroutine sleeps 2 seconds
func goroutine #1 starts sending data into the channel
func goroutine #1 after sending data into the channel
func goroutine #2 starts sending data into the channel
func goroutine #2 after sending data into the channel
func goroutine #3 starts sending data into the channel
func goroutine #3 after sending data into the channel
func goroutine #4 starts sending data into the channel
main goroutine received value from channel: 1
main goroutine received value from channel: 2
main goroutine received value from channel: 3
main goroutine received value from channel: 4
func goroutine #4 after sending data into the channel
func goroutine #5 starts sending data into the channel
func goroutine #5 after sending data into the channel
main goroutine received value from channel: 5
```

Gambar diatas merupakan hasil eksekusi dari penggunaan *buffered channel* pada halaman sebelumnya. Bisa kita lihat tulisan func goroutine #1 after sending data into the channel dan tulisan lainnya yang mengandung kata after langsung ditampilkan setelah proses pengiriman data. Hal ini terjadi karena kita menggunakan *buffered channel* dengan kapasitas 3.

Lalu perhatikan kembali, proses pengiriman data terhenti ketika pengiriman data sudah melibihi kapasitas dari kapasitas buffer yang kita berikan sebanyak 3 buffer.

Jika pengiriman data sudah melibihi kapasitas, maka proses penerimaan data akan dieksekusi hingga paling tidak sudah ada data yang diterima. Kemudian setelah proses penerimaan data telah selesai dan jika masih ada data yang dikirimkan, maka proses pengiriman data akan dimulai kembali secara asynchronous.

Channel (Select)

Select merupakan sebuah fitur pada bahasa Go yang memungkinkan kita untuk dapat menggunakan lebih dari satu channel untuk proses komunikasi antara *Goroutine* satu dengan yang lainnya.

Perhatikan pada gambar disebelah kanan. Kita membuat 2 channel yang terdapat pada variable *c1* dan *c2*.

Lalu kemudian kita membuat 2 *Goroutine* menggunakan function *anonymous*.

```
120  func main() {
121
122    c1 := make(chan string)
123    c2 := make(chan string)
124
125    go func() {
126      time.Sleep(2 * time.Second)
127
128      c1 <- "Hello!"
129    }()
130
131    go func() {
132      time.Sleep(1 * time.Second)
133
134      c2 <- "Salut!"
135    }()
136
137    for i := 1; i <= 2; i++ {
138      select {
139      case msg1 := <-c1:
140        fmt.Println("Received", msg1)
141      case msg2 := <-c2:
142        fmt.Println("Received", msg2)
143      }
144    }
145
146  }
```

Channel (Select)

Kemudian pada line 137-143 kita melakukan looping sebanyak jumlah *channel* yang telah kita buat yaitu 2. Lalu kita menggunakan *select* didalam looping nya. Berikut merupakan penjelasan dari penggunana select pada gambar di sebelah kanan:

- Kondisi case *msg1* akan terpenuhi ketika terdapat peneriman data dari channel *c1,* yang kemudian akan ditampung oleh variable *msg1*.
- Kondisi case *msg2* akan terpenuhi ketika terdapat peneriman data dari channel *c2,* yang kemudian akan ditampung oleh variable *msg2*.

Ketika dijalankan pada terminal, maka hasilnya akan seperti pada gambar dibawah.