MODUL XIII
PACKAGE ADDITIONAL

Pertemuan : 13
Waktu : 8 x 60 menit (Online)

1.1 Tujuan Modul III

Setelah mahasiswa mempelajari materi ini, diharapkan dapat :

1. Memahami package yang sering/familiar di golang.

1.2 Landasan Teori

1.2.1 Reflect

Reflection adalah teknik untuk inspeksi variabel, mengambil informasi dari variabel tersebut atau bahkan memanipulasinya. Cakupan informasi yang bisa didapatkan lewat reflection sangat luas, seperti melihat struktur variabel, tipe, nilai pointer, dan banyak lagi.

Go menyediakan package `reflect`, berisikan banyak sekali fungsi untuk keperluan reflection. Di bab ini, kita akan belajar tentang dasar penggunaan package tersebut.

Dari banyak fungsi yang tersedia di dalam package tersebut, ada 2 fungsi yang paling penting untuk diketahui, yaitu `reflect.ValueOf()` dan `reflect.TypeOf()`.

- Fungsi `reflect.ValueOf()` akan mengembalikan objek dalam tipe `reflect.Value`, yang berisikan informasi yang berhubungan dengan nilai pada variabel yang dicari
- Sedangkan `reflect.TypeOf()` mengembalikan objek dalam tipe `reflect.Type`. Objek tersebut berisikan informasi yang berhubungan dengan tipe data variabel yang dicari

1.2.1.1 Mencari Tipe Data & Value Menggunakan Reflect

Dengan reflection, tipe data dan nilai variabel dapat diketahui dengan mudah. Contoh penerapannya bisa dilihat pada kode berikut.

```
package main

import "fmt"
import "reflect"

func main() {
    var number = 23
    var reflectValue = reflect.ValueOf(number)

    fmt.Println("tipe  variabel :", reflectValue.Type())

    if reflectValue.Kind() == reflect.Int {
        fmt.Println("nilai variabel :", reflectValue.Int())
    }
}
```

```
novalagung:belajar-golang $ go run bab28.go
tipe  variabel : int
nilai variabel : 23
novalagung:belajar-golang $
```

Fungsi `reflect.valueOf()` memiliki parameter yang bisa menampung segala jenis tipe data. Fungsi tersebut mengembalikan objek dalam tipe `reflect.Value`, yang berisikan informasi mengenai variabel yang bersangkutan.

Objek `reflect.Value` memiliki beberapa method, salah satunya `Type()`. Method ini mengembalikan tipe data variabel yang bersangkutan dalam bentuk `string`.

Statement reflectValue.Int() menghasilkan nilai int dari variabel number. Untuk menampilkan nilai variabel reflect, harus dipastikan dulu tipe datanya. Ketika tipe data adalah int, maka bisa menggunakan method Int(). Ada banyak lagi method milik struct reflect.Value yang bisa digunakan untuk pengambilan nilai dalam bentuk tertentu, contohnya: reflectValue.String() digunakan untuk mengambil nilai string, reflectValue.Float64() untuk nilai float64, dan lainnya.

Perlu diketahui, fungsi yang digunakan harus sesuai dengan tipe data nilai yang ditampung variabel. Jika fungsi yang digunakan berbeda dengan tipe data variabelnya, maka akan menghasilkan error. Contohnya pada variabel menampung nilai bertipe `float64`, maka tidak bisa menggunakan method `String()`.

Diperlukan adanya pengecekan tipe data nilai yang disimpan, agar pengambilan nilai bisa tepat. Salah satunya bisa dengan cara seperti kode di atas, yaitu dengan mengecek dahulu apa jenis tipe datanya menggunakan method `Kind()`, setelah itu diambil nilainya dengan method yang sesuai.

List konstanta tipe data dan method yang bisa digunakan dalam refleksi di Go bisa dilihat di [reflect package - reflect]

1.2.1.2 Mencari Tipe Data & Value Menggunakan Reflect

Reflect bisa digunakan untuk mengambil informasi semua property variabel objek cetakan struct, dengan catatan property-property tersebut bermodifier public. Langsung saja kita praktekan, siapkan sebuah struct bernama `student`.

```
type student struct {
    Name  string
    Grade int
}
```

Buat method baru untuk struct tersebut, dengan nama method `getPropertyInfo()`. Method ini berisikan kode untuk mengambil dan menampilkan informasi tiap property milik struct `student`.

```
func (s *student) getPropertyInfo() {
    var reflectValue = reflect.ValueOf(s)

    if reflectValue.Kind() == reflect.Ptr {
        reflectValue = reflectValue.Elem()
    }

    var reflectType = reflectValue.Type()

    for i := 0; i < reflectValue.NumField(); i++ {
        fmt.Println("nama      :", reflectType.Field(i).Name)
        fmt.Println("tipe data :", reflectType.Field(i).Type)
```

```
            fmt.Println("nilai     :",
reflectValue.Field(i).Interface())
            fmt.Println("")
        }
    }
```

Terakhir, lakukan uji coba method di fungsi `main`.

```
func main() {
    var s1 = &student{Name: "wick", Grade: 2}
    s1.getPropertyInfo()
}
```

Didalam method `getPropertyInfo` terjadi beberapa hal. Pertama objek `reflect.Value` dari variabel `s` diambil. Setelah itu dilakukan pengecekan apakah variabel objek tersebut merupakan pointer atau tidak (bisa dilihat dari `if reflectValue.Kind() == reflect.Ptr`, jika bernilai `true` maka variabel adalah pointer). jika ternyata variabel memang pointer, maka perlu diambil objek reflect aslinya dengan cara memanggil method `Elem()`.

Setelah itu, dilakukan perulangan sebanyak jumlah property yang ada pada struct `student`. Method `NumField()` akan mengembalikan jumlah property publik yang ada dalam struct.

Di tiap perulangan, informasi tiap property struct diambil berurutan dengan lewat method `Field()`. Method ini ada pada tipe `reflect.Value` dan `reflect.Type`.

- `reflectType.Field(i).Name` akan mengembalikan nama property
- `reflectType.Field(i).Type` mengembalikan tipe data property
- `reflectValue.Field(i).Interface()` mengembalikan nilai property dalam bentuk `interface{}`

Pengambilan informasi property, selain menggunakan indeks, bisa diambil berdasarkan nama field dengan menggunakan method `FieldByName()`.

1.2.1.3 Pengaksesan Informasi Method Variabel Objek

Informasi mengenai method juga bisa diakses lewat reflect, syaratnya masih sama seperti pada pengaksesan proprerty, yaitu harus bermodifier public.

Pada contoh dibawah ini informasi method `SetName` akan diambil lewat reflection. Siapkan method baru di struct `student`, dengan nama `SetName`.

```
func (s *student) SetName(name string) {
    s.Name = name
}
```

Buat contoh penerapannya di fungsi `main`.

```
func main() {
    var s1 = &student{Name: "john wick", Grade: 2}
    fmt.Println("nama :", s1.Name)

    var reflectValue = reflect.ValueOf(s1)
    var method = reflectValue.MethodByName("SetName")
    method.Call([]reflect.Value{
        reflect.ValueOf("wick"),
    })

    fmt.Println("nama :", s1.Name)
}
```

Pada kode di atas, disiapkan variabel `s1` yang merupakan instance struct `student`. Awalnya property `Name` variabel tersebut berisikan string `"john wick"`.

Setelah itu, refleksi nilai objek tersebut diambil, refleksi method `SetName` juga diambil. Pengambilan refleksi method dilakukan menggunakan `MethodByName` dengan argument adalah nama method yang diinginkan, atau bisa juga lewat indeks method-nya (menggunakan `Method(i)`).

Setelah refleksi method yang dicari sudah didapatkan, `Call()` dipanggil untuk eksekusi method.

Jika eksekusi method diikuti pengisian parameter, maka parameternya harus ditulis dalam bentuk array `[]reflect.Value` berurutan sesuai urutan deklarasi parameter-nya. Dan nilai yang dimasukkan ke array tersebut harus dalam bentuk `reflect.Value` (gunakan `reflect.ValueOf()` untuk pengambilannya).

```
[]reflect.Value{
    reflect.ValueOf("wick"),
}
```

1.2.2 Goroutine

Goroutine mirip dengan thread thread, tapi sebenarnya bukan. Sebuah *native thread* bisa berisikan sangat banyak goroutine. Mungkin lebih pas kalau goroutine disebut sebagai **mini thread**. Goroutine sangat ringan, hanya dibutuhkan sekitar **2kB** memori saja untuk satu buah goroutine. Eksekusi goroutine bersifat *asynchronous*, menjadikannya tidak saling tunggu dengan goroutine lain.

Note : Karena goroutine sangat ringan, maka eksekusi banyak goroutine bukan masalah. Akan tetapi jika jumlah goroutine sangat banyak sekali (contoh 1 juta goroutine dijalankan pada komputer dengan RAM terbatas), memang proses akan jauh lebih cepat selesai, tapi memory/RAM pasti bengkak.

Goroutine merupakan salah satu bagian paling penting dalam *concurrent programming* di Go. Salah satu yang membuat goroutine sangat istimewa adalah eksekusi-nya dijalankan di multi core processor. Kita bisa tentukan berapa banyak core yang aktif, makin banyak akan semakin cepat.

1.2.2.1 Penerapan Goroutine

Untuk menerapkan goroutine, proses yang akan dieksekusi sebagai goroutine harus dibungkus kedalam sebuah fungsi. Pada saat pemanggilan fungsi tersebut, ditambahkan keyword `go` didepannya, dengan itu goroutine baru akan dibuat dengan tugas adalah menjalankan proses yang ada dalam fungsi tersebut.

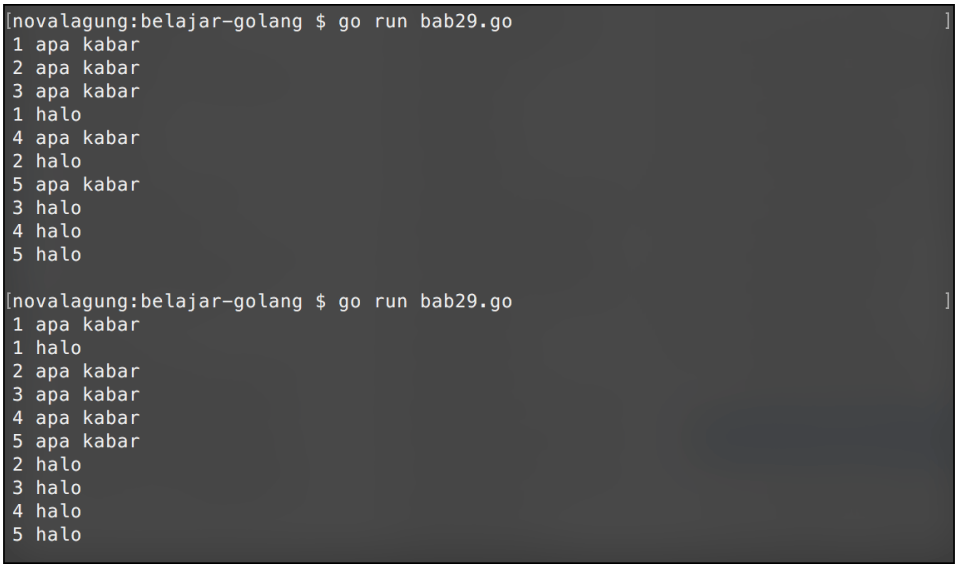
Berikut merupakan contoh implementasi sederhana tentang goroutine. Program di bawah ini menampilkan 10 baris teks, 5 dieksekusi dengan cara biasa, dan 5 lainnya dieksekusi sebagai goroutine baru.

```
package main

import "fmt"
import "runtime"

func print(till int, message string) {
    for i := 0; i < till; i++ {
        fmt.Println((i + 1), message)
    }
}

func main() {
    runtime.GOMAXPROCS(2)

    go print(5, "halo")
    print(5, "apa kabar")

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

Pada kode di atas, Fungsi `runtime.GOMAXPROCS(n)` digunakan untuk menentukan jumlah core yang diaktifkan untuk eksekusi program.

Pembuatan goroutine baru ditandai dengan keyword `go`. Contohnya pada statement `go print(5, "halo")`, di situ fungsi `print()` dieksekusi sebagai goroutine baru.

Fungsi `fmt.Scanln()` mengakibatkan proses jalannya aplikasi berhenti di baris itu (**blocking**) hingga user menekan tombol enter. Hal ini perlu dilakukan karena ada kemungkinan waktu selesainya eksekusi goroutine `print()` lebih lama dibanding waktu selesainya goroutine utama `main()`, mengingat bahwa keduanya sama-sama asynchronous. Jika itu terjadi, goroutine yang belum selesai secara paksa dihentikan prosesnya karena goroutine utama sudah selesai dijalankan.



Bisa dilihat di output, tulisan `"halo"` dan `"apa kabar"` bermunculan selang-seling. Ini disebabkan karena statement `print(5, "halo")` dijalankan sebagai goroutine baru, menjadikannya tidak saling tunggu dengan `print(5, "apa kabar")`.

Pada gambar di atas, program dieksekusi 2 kali. Hasil eksekusi pertama berbeda dengan kedua, penyebabnya adalah karena kita menggunakan 2 prosesor. Goroutine mana yang dieksekusi terlebih dahulu tergantung kedua prosesor tersebut.

1.2.2.2 Penggunaan Fungsi runtime.GOMAXPROCS()

Fungsi ini digunakan untuk menentukan jumlah core atau processor yang digunakan dalam eksekusi program.

Jumlah yang diinputkan secara otomatis akan disesuaikan dengan jumlah asli *logical processor* yang ada. Jika jumlahnya lebih, maka dianggap menggunakan sejumlah prosesor yang ada.

1.2.3 Random Generator

Random Number Generator (RNG) merupakan sebuah perangkat (bisa software, bisa hardware) yang menghasilkan data deret/urutan angka yang sifatnya acak.

RNG bisa berupa hardware yang murni bisa menghasilkan data angka acak, atau bisa saja sebuah pseudo-random yang menghasilkan deret angka-angka yang **terlihat acak** tetapi sebenarnya tidak benar-benar acak, yang deret angka tersebut sebenarnya merupakan hasil kalkulasi algoritma deterministik dan probabilitas. Jadi untuk pseudo-random ini, asalkan kita tau *state*-nya maka kita akan bisa menebak hasil deret angka random-nya.

Dalam per-randoman-duniawi terdapat istilah **seed** atau titik mulai (*starting point*). Seed ini digunakan oleh RNG dalam peng-generate-an angka random di tiap urutannya.

Sedikit ilustrasi mengenai korelasi antara seed dengan RNG, agar lebih jelas.

- Dimisalkan saya menggunakan seed yaitu angka `10`, maka ketika fungsi RNG dijalankan untuk pertama kalinya, output angka yang dihasilkan pasti `5221277731205826435`. Angka random tersebut pasti *fix* dan akan selalu menjadi output angka random pertama yang dihasilkan, ketika seed yang digunakan adalah angka `10`.
- Misalnya lagi, fungsi RNG di-eksekusi untuk ke-dua kalinya, maka angka random kedua yang dihasilkan adalah pasti `3852159813000522384`. Dan seterusnya.
- Misalkan lagi, fungsi RNG di-eksekusi lagi, maka angka random ketiga pasti `8532807521486154107`.
- Jadi untuk seed angka `10`, akan selalu menghasilkan angka random ke-1: `5221277731205826435`, ke-2: `3852159813000522384`, ke-3 `8532807521486154107`. Meskipun fungsi random dijalankan di program yang berbeda, di waktu yang berbeda, di environment yang berbeda, jika seed adalah `10` maka deret angka random yang dihasilkan pasti sama seperti contoh di atas.

1.2.3.1 Package math/rand

Di Go terdapat sebuah package yaitu `math/rand` yang isinya banyak sekali API untuk keperluan penciptaan angka random. Package ini mengadopsi **PRNG** atau *pseudo-random* number generator. Deret angka

random yang dihasilkan sangat tergantung dengan angka **seed** yang digunakan.

Cara menggunakan package ini sangat mudah, yaitu cukup import `math/rand`, lalu set seed-nya, lalu panggil fungsi untuk generate angka random-nya. Lebih jelasnya silahkan cek contoh berikut.

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    rand.Seed(10)
    fmt.Println("random ke-1:", rand.Int()) // 5221277731205826435
    fmt.Println("random ke-2:", rand.Int()) // 3852159813000522384
    fmt.Println("random ke-3:", rand.Int()) // 8532807521486154107
}
```

Fungsi `rand.Seed()` digunakan untuk penentuan nilai seed. Fungsi `rand.Int()` digunakan untuk generate angka random dalam bentuk numerik bertipe `int`. Fungsi `rand.Int()` ini setiap kali dipanggil akan menghasilkan angka berbeda, tapi pasti hasilnya akan selalu tetap jika mengacu ke deret.

- Angka random ke-1 akan selalu `5221277731205826435`
- Angka random ke-2 akan selalu `3852159813000522384`
- Angka random ke-3 akan selalu `8532807521486154107`
- Dan seterusnya ...

1.2.3.2 Unique Seed

Lalu bagaimana cara agar angka yang dihasilkan selalu berbeda setiap kali dipanggil? Apakah harus set ulang seed-nya? Jangan, karena kalau seed di-set ulang maka urutan deret random akan berubah. Seed hanya perlu di set sekali di awal. Lha, terus bagaimana?

Jadi begini, setiap kali `rand.Int()` dipanggil, hasilnya itu selalu berbeda, tapi sangat bisa diprediksi jika kita tau seed-nya, dan ini adalah masalah besar. Nah, ada cara agar angka random yang dihasilkan tidak

berulang-ulang selalu contoh di-atas, caranya adalah menggunakan angka yang *unique*/unik sebagai seed, contohnya seperti angka unix nano dari waktu sekarang.

Coba modifikasi program dengan kode berikut, lalu jalankan ulang. Jangan lupa meng-import package `time` ya.

1.2.3.3 Random Tipe Data Numerik Lainnya

Di dalam package `math/rand`, ada banyak fungsi untuk generate angka random. Fungsi `rand.Int()` hanya salah satu dari fungsi yang tersedia di dalam package tersebut, yang gunanya adalah menghasilkan angka random bertipe `int`.

Selain itu, ada juga `rand.Float32()` yang menghasilkan angka random bertipe `float32`. Ada juga `rand.Uint32()` yang menghasilkan angka random bertipe *unsigned* int, dan lainnya.

lebih detailnya silakan merujuk ke https://golang.org/pkg/math/rand/

1.2.3.4 Angka Random Index Tertentu

Gunakan fungsi `rand.Intn(n)` untuk mendapatkan angka random pada indeks ke `n`. Dengan ini tidak perlu memanggil `rand.Int()` tiga kali untuk mendapatkan angka random ke-tiga, melainkan cukup gunakan `rand.Intn(2)` (indeks dari 0 ya).

1.2.3.5 Random Tipe Data String

Untuk menghasilkan data random string, ada banyak cara yang bisa digunakan, salah satunya adalah dengan memanfaatkan alfabet dan hasil random numerik.

```
	var letters =
[]rune("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")

	func randomString(length int) string {
	    b := make([]rune, length)
	    for i := range b {
	        b[i] = letters[rand.Intn(len(letters))]
	    }
	    return string(b)
```

```
}
```

Dengan fungsi di atas kita bisa dengan mudah meng-generate string random dengan panjang karakter yang sudah ditentukan, misal `randomString(10)` akan menghasilkan random string 10 karakter.

1.3 Praktikum

1.3.1 Latihan Praktikum

None

1.3.2 Tugas Praktikum

None