ภาวะพร้อมกัน (Concurrency)

โปรแกรมขนาดใหญ่มักจะประกอบด้วยโปรแกรมขนาดเล็กหลายตัว อย่างเช่น เว็บเซิร์ฟเวอร์ที่ต้องจัดการรีเควสท์ จากเบราเซอร์และส่ง HTML กลับไป ตอนที่ต้องจัดการกับรีเควสท์ แต่ละตัวมันก็เหมือนกับโปรแกรมเล็กๆ ตัวนึง
ในทางอุดมคติโปรแกรมแบบนี้จะสามารถรันแต่ละส่วนพร้อมกันได้ (กรณีที่เว็บเซิร์ฟเวอร์จัดการรีเควสท์หลายตัว) การทำให้งานแต่ละงานสามารถทำในเวลาเดียวกันได้นี่แหละเราเรียกว่า concurrency ซึ่ง Go มันโคตรจะสนับสนุนการทำ concurrency โดยใช้สิ่งที่เรียกว่า goroutines และ channels

โกรูทีน (Goroutines)

โกรูทีนเป็นฟังก์ชันที่สามารถทำงานได้ในเวลาเดียวกันกับฟังก์ชันอื่นได้ วิธีสร้างโกรูทีนก็แค่ใส่คำว่า `go` นำหน้าการเรียกใช้งานฟังก์ชัน

```
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)
}
```

โปรแกรมนี้มีโกรูทีนสองตัว ตัวแรกมันซ่อนอยู่ ซึ่งก็คือฟังก์ชัน main นั่นเอง ส่วนตัวที่สองมันถูกสร้างตอนที่เราเรียก `go f(0)` ปกติแล้วเวลาเราเรียกฟังก์ชันโปรแกรมมันจะทำงานและคืนค่าผลลัพธ์ของฟังก์ชันก่อนทำคำสั่งในบรรทัดถัดไป แต่โกรูทีนจะคืนค่ากลับมาทันทีและทำบรรทัดถัดไปโดยไม่ต้องรอให้ฟังก์ชันทำงานจบ เป็นที่มาว่าทำไมต้องเรียก `Scanln` เพราะถ้าไม่เรียกโปรแกรมก็จะจบการทำงานไปเลยโดยไม่มีโอกาสที่จะได้พิมพ์ค่าตัวเลขออกทางหน้าจอ

โกรูทีนมันเบา (lightweight) และเราสามารถสร้างมันขึ้นมาเป็นพันๆ ได้โดยง่าย เราสามารถแก้โปรแกรมให้รัน 10 โกรูทีน ได้ตามนี้

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

เราอาจจะเห็นเหมือนว่าโปรแกรมมันรันโกรูทีนตามลำดับแทนที่จะรันพร้อมกัน ทีนี้เราลองมาหน่วงเวลามันซักหน่อยด้วยคำสั่ง `time.Sleep` และ `rand.Intn`

```
package main

import (
    "fmt"
    "time"
    "math/rand"
)

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}
func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

`f` พิมพ์ตัวเลขจาก 0 ถึง 10, แล้วก็รอ 0 - 250 มิลลิวินาทีในแต่ละรอบ นั่นแน่! เห็นแล้วใช่มั้ยว่าโกรูทีน มันรันพร้อมกัน

แชนแนล (Channels)

แชนแนลจัดวิธีที่จะทำให้โกรูทีน คุยกันและทำงานประสานจังหวะ (synchronize) กันได้ มาดูตัวอย่างการใช้แชนแนลกันเลย

```
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)
}
```

จากตัวอย่างนี้จะพิมพ์ “ping” ออกมาเรื่อยๆ (กด enter เพื่อให้จบการทำงาน) แชนแนลประกาศโดยใช้คำว่า `chan` ตามด้วยประเภทของสิ่งที่จะส่งเข้าไปในแชนแนล (ในตัวอย่างเราส่ง string) เครื่องหมาย `<-` (ลูกศรชี้ไปทางซ้าย) ใช้เพื่อส่งและรับข้อความบนแชนแนล `c <- “ping”` หมายถึงส่ง `“ping”` เข้าไปในแชนแนล ส่วนการรับข้อความจะเขียนในรูป `msg := <- c` ซึ่งหมายถึง รับข้อความจากแชนแนล และเก็บข้อความไว้ใน `msg` ที่จริงการพิมพ์ข้อความออกหน้าจอจะลบบรรทัดก่อนหน้าออกแล้วเขียนแค่ `fmt.Println(<-c)` ก็ได้นะ

การใช้แชนแนลแบบนี้จะประสานจังหวะของโกรูทีนสองตัว เมื่อ pinger พยายามที่จะส่งข้อความบนแชนแนลมันก็จะรอจนกว่า printer พร้อมที่จะรับข้อความ (เราเรียกอาการแบบนี้ว่า blocking) มาลองเพิ่มตัวส่งข้อความอีกตัวเข้าไปในโปรแกรมแล้วดูซิว่าจะเกิดอะไรขึ้น เราเพิ่มฟังก์ชันเข้าไป

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

แล้วแก้ฟังก์ชัน main อีกซักหน่อย

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

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

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

ตอนนี้โปรแกรมก็จะสลับกับพิมพ์ข้อความ “ping” และ “pong”

ทิศทางของแชนแนล (Channel Direction)

เราสามารถระบุได้ว่าแชนแนลนี้จะให้ทำเฉพาะรับหรือส่ง เช่นฟังก์ชัน pinger เราสามารถเขียนเป็น

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

ตอนนี้เราจะทำได้แค่ส่งข้อความเข้าไปใน c เท่านั้น ถ้าเราพยายามที่จะรับข้อมูลจาก c มันจะเกิดความผิดพลาดขึ้นตอนที่เราคอมไพล์ เราสามารถเปลี่ยน printer ได้โดยใช้วิธีคล้ายๆกัน

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

แชนแนลที่ไม่ได้ระบุทิศทางจะสามารถใช้งานได้ทั้งสองทิศทาง ซึ่งแชนแนลแบบสองทิศทางสามารถส่งเข้าฟังก์ชันที่รับแชนแนลแบบส่งอย่างเดียว หรือ รับอย่างเดียวก็ได้

Select

Go มี statement พิเศษชื่อ `select` ซึ่งทำงานคล้ายกับ `switch` แต่ใช้กับแชนแนล

```
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)
}
```

โปรแกรมนี้จะพิมพ์ “from 1” ทุกสองวินาที และ “from 2” ทุกสามวินาที `select` จะเลือกแชนแนลแรกที่พร้อมจะรับจากมัน (หรือส่งมาให้มัน) ถ้ามีมากกว่าหนึ่งแชนแนลที่พร้อม มันก็จะสุ่มเอาอันไหนก็ได้ ถ้าไม่มีแชนแนลไหนพร้อมเลย มันก็จะรอจนกว่าจะมีแชนแนลพร้อมขึ้นมาซักตัว

`select` ถูกใช้บ่อยในการทำ timeout

```
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")
}
```

time.After จะสร้างแชนแนลขึ้นมาและส่งไปตอนที่ครบกำหนดเวลา (เราไม่สนใจที่จะเก็บเวลาก็เลยไม่ต้องประกาศตัวแปรมาเก็บค่ามัน) นอกจากนี้เรายังสามารถระบุกรณี default ได้ด้วย

```
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")
}
```

กรณี default นี้จะเกิดทันทีที่ไม่มีแชนแนลใดพร้อมเลย

Buffered Channels

เราสามารถส่งพารามิเตอร์ตัวที่สองเข้าไปในฟังก์ชัน make ตอนที่สร้างแชนแนลได้ ดังนี้

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

เราจะได้ buffered channel ที่มีความจุเป็น 1 โดยปกติแล้วแชนแนลจะเป็น synchronous แต่ละฝั่งของแชนแนลจะรอจนกว่าอีกฝั่งจะพร้อม ซึ่ง buffered channel จะต่างออกไป โดย buffered channel จะเป็น asynchronous แต่ละฝั่งจะไม่รอกันจนกว่าแชนแนลจะเต็ม

ปัญหาท้าทาย

- ลองเขียนแชนแนลแบบระบุทิศทางดูหน่อย มันเขียนยังไงแล้วนะ?
- เขียนฟังก์ชัน Sleep ขึ้นมาเองเลย โดยใช้ time.After นะ
- buffered channel มันคืออะไรอะ อธิบายหน่อย แล้วถ้าเราจะสร้างมันให้มีความจุซัก 20 เราจะทำได้ยังไง?