ฟังก์ชัน(function)

ฟังก์ชัน(function) คือส่วนของโค้ดที่เราเขียนแยกออกมา โดยอาจจะไม่มีการรับค่าพารามิเตอร์ หรือ รับหลายๆตัวก็ได้ แล้วเมื่อฟังก์ชันทำงานเสร็จ อาจจะไม่มีการส่งผลลัพธ์ออกมา หรือ ถ้ามี โก นั้นก็สามารถมีการส่งค่ากลับออกมาได้หลายค่า เราอาจจะมองฟังก์ชันเป็น กล่องดำ ก็ได้คือ

เราไม่จำเป็นต้องรู้ว่าภายในตัวฟังก์ชันที่เราจะเรียกใช้ทำงานยังไง แค่รู้ว่าจำเป็นต้องใส่อินพุตพารามิเตอร์อะไรเข้าไป ถึงจะได้ผลลัพธ์ตามที่ต้องการออกมา เราได้ลองสร้างฟังก์ชันกันมาก่อนแล้ว คือฟังก์ชัน main ที่เป็นส่วนของโค้ดหลักที่จะถูกเรียกใช้งานเมื่อโปรแกรมทำงาน

```
func main() {}
```

ต่อไปมาดูกันว่าเราจะเขียนฟังก์ชันขึ้นมาใช้เองอีกได้ยังไงบ้าง

สร้างฟังก์ชันใหม่ขึ้นใช้เอง

จากโค้ดนี้ในบทที่ 6

```
func main() {
    xs := []float64{98,93,77,82,83}
    total := 0.0
    for _, v := range xs {
        total += v
    }
    fmt.Println(total / float64(len(xs)))
}
```

โปรแกรมนี้ทำการคำนวณค่าเฉลี่ยของตัวเลขที่อยู่ในตัวแปร xs การหาค่าเฉลี่ยแบบนี้ก็อยู่ในรูปแบบที่ถูกเอาไปใช้แก้ปัญหาอื่นๆด้วย ดังนั้นจึงควรแยกออกไปเป็นฟังก์ชันใหม่ดีกว่า เราจะสร้างฟังก์ชัน average โดยจะรับค่าข้อมูลแบบ สไลซ์ ของ float64 เข้ามา และ ให้ผลลัพธ์กลับไป 1 ค่าที่เป็นประเภทข้อมูลแบบ float64 ให้เราเพิ่มโค้ดต่อไปนี้ก่อนฟังก์ชัน main

```
func average(xs []float64) float64 {
    panic("Not Implemented")
}
```

จะเห็นว่าเราสร้างฟังก์ชันได้โดยใช้คีย์เวิร์ด func, ตามด้วยชื่อฟังก์ชัน ส่วนพารามิเตอร์อยู่ในวงเล็บหลังจากชื่อฟังก์ชัน ในรูปแบบของ ชื่อ ตามด้วยประเภทข้อมูล

```
name type, name type, …
```

ซึ่งจะไม่มี หรือ มีหลายๆตัวได้ ฟังก์ชัน average นี้รับแค่ 1 พารามิเตอร์คือลิสต์ของตัวเลขที่ต้องการหาค่าเฉลี่ย โดยเราตั้งชื่อให้ว่า xs หลังจากวงเว็บของรายการพารามิเตอร์ เราจะใส่ประเภทข้อมูลที่ฟังก์ชันนี้ต้องการส่งกลับ ส่วนประกอบของฟังก์ชันที่ว่ามาได้แก่ ชื่อ , พารามิเตอร์ , ประเภทข้อมูลส่งกลับ จะเป็นตัวบ่งบอกว่าเป็นฟังก์ชันไหน หรือเรียกว่า function signature

สุดท้ายหลังจากกำหนด signature ของฟังก์ชันแล้วก็จะตามด้วยวงเล็บปีกกาที่จะเป็นกลุ่มข้องโค้ดการทำงานของฟังก์ชันนี้หรือเรียกว่า body ของฟังก์ชัน จากโค้ดที่เห็นในบอดี้ของฟังก์ชันนี้เรามีการเรียกฟังก์ชันชื่อ panic อยู่ ซึ่งตอนนี้จะทำให้เกิด run time error ขึ้น (เดี๋ยวเราดูเรื่อง panic กันอีกทีท้ายๆบทนี้) การเขียนการทำงานของฟังก์ชันที่ยากๆ มักจะไม่เอามารวมไว้ในที่เดียว แต่เราจะแตกย่อยเป็นฟังก์ชันเล็กๆ แล้วเอากลุ่มฟังก์ชันย่อยที่สร้าง มาสร้างฟังก์ชันที่ซับซ้อนขึ้นมาอีกที

ทีนี้เราเอาโค้ดที่เคยเขียนใน main ย้ายมาอยู่ที่ฟังก์ชัน average ได้แบบนี้

```
func average(xs []float64) float64 {
    total := 0.0
    for _, v := range xs {
        total += v
    }
    return total / float64(len(xs))
}
```

เราเปลี่ยนโค้ดตรง fmt.Println เป็น return แทน เพราะเราไม่ได้ต้องการให้ฟังก์ชันแสดงอะไรออกไปที่หน้าจอ แต่เราจะใช้ return เพื่อส่งค่ากลับไปให้กับโค้ดจุดที่เรียกใช้ฟังก์ชันแทน การใช้ return จะทำให้การทำงานของฟังก์ชันหยุดลงทันที และส่งค่าที่กำหนดหลัง return ออกไป โค้ดใน main เราก็จะแก้ให้เป็นดังนี้

```
func main() {
    xs := []float64{98,93,77,82,83}
    fmt.Println(average(xs))
}
```

เมื่อรัน ก็ควรจะได้ผลลัพธ์แบบเดิมกับโค้ดก่อนหน้านี้ มีสิ่งที่ควรจำไว้ก็คือ
ชื่อของตัวแปรที่ส่งให้กับฟังก์ชัน ไม่จำเป็นต้องเป็นชื่อเหมือนกับพารามิเตอร์ตอนที่เราสร้างฟังก์ชัน
ตัวอย่างเช่นเราสามารถเขียนแบบนี้ได้

```
func main() {
    someOtherName := []float64{98,93,77,82,83}
    fmt.Println(average(someOtherName))
}
```

แล้วโปรแกรมเราก็ควรจะทำงานได้เหมือนเดิม โค้ดในตัวฟังก์ชันเอง ไม่สามารถเข้าใช้งานตัวแปรที่ถูกสร้างในฟังก์ชันต้นทางที่เรียกใช้ฟังก์ชันได้ เช่นแบบนี้ทำไม่ได้

```
func f() {
    fmt.Println(x)
}
func main() {
    x := 5
    f()
}
```

เราควรจะให้ฟังก์ชันส่งข้อมูลที่จำเป็นไปแทน ผ่านพารามิเตอร์ แบบนี้

```
func f(x int) {
    fmt.Println(x)
}
func main() {
    x := 5
    f(x)
}
```

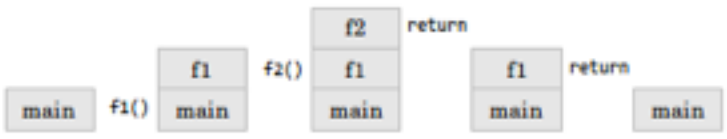
หรือเลือกที่จะประกาศตัวแปรไว้ภายนอกฟังก์ชันทั้งคู่แบบนี้

```
var x int = 5
func f() {
    fmt.Println(x)
}
func main() {
    f()
}
```

กลไกการทำงานของฟังก์ชัน เวลาฟังก์ชันหนึ่งเรียกใช้อีกฟังก์ชันหนึ่ง จะทำงานกันแบบ stack เช่นเรามีโค้ดแบบนี้

```
func main() {
    fmt.Println(f1())
}
func f1() int {
    return f2()
}
func f2() int {
    return 1
}
```

เวลาโปรแกรมทำงานจะเริ่มจาก main และมี stack ของการเรียกฟังก์ชันแสดงออกมาเป็นรูปได้แบบนี้



ทุกครั้งที่เรียกฟังก์ชันอื่นจะเกิดการเก็บข้อมูลในขอบเขตของฟังก์ชันลง stack ซ้อนทับลงไปที่ฟังก์ชันที่เป็นคนเรียก และ ก็ซ้อนทับไปอีกชั้นเมื่อมีการเรียกต่อไปอีกฟังก์ชัน เมื่อมีการ return จะจบการทำงานของฟังก์ชัน ก็จะเอาข้อมูลของฟังก์ชันที่ทำงานจบแล้วออกไปจาก stack
Go นั้นสามารถกำหนดชื่อให้ตัวแปรให้กับค่าที่ต้องการส่งกลับได้ เช่น

```
func f2() (r int) {
    r = 1
    return
}
```

โดยเราสามารถกำหนดค่าให้ตัวแปรนี้แทนการ return ค่าตรงๆได้

การส่งค่ากลับหลายค่า

Go สามารถส่งค่ากลับได้หลายค่า ตัวอย่างเช่น

```
func f() (int, int) {
    return 5, 6
}

func main() {
    x, y := f()
}
```

จากโค้ดนี้จะเห็นว่าถ้าต้องการทำให้ส่งกลับได้หลายค่า ตรงส่วนของประเภทข้อมูลของค่าที่ส่งกลับ เราจะมีวงเล็บครอบก่อนด้วย โดยในนั้นก็คือลิสต์ของประเภทข้อมูลที่จะส่งกลับ คั่นด้วย , ในส่วนของการเรียกใช้แล้วเราตัวแปรมารับค่าที่ส่งกลับแบบหลายค่า ก็ต้องกำหนดตัวแปรหลายตัวคั่นตัว , ทางด้านซ้ายของเครื่องหมาย := หรือ = เช่นกัน

การส่งกลับหลายๆค่าใน Go มักจะถูกเอาไปใช้กับการส่งข้อมูลความผิดพลาดของการทำงานของฟังก์ชันออกมาพร้อมกับค่าผลลัพธ์ มักจะอยู่ในรูปแบบ x, err := f() หรือใช้ข้อมูลแบบ boolean ช่วยเช่น x, ok := f()

ฟังก์ชันแบบรับพารามิเตอร์ได้ โดยไม่จำกัดจำนวน (Variadic Functions)

การรับค่าแบบนี้เป็นรูปแบบพิเศษ ตัวอย่างที่เราได้ใช้กันไปแล้วก็อย่างเช่น fmt.Println ที่เราสามารถส่งไปกี่ค่าก็ได้ ที่นี้ถ้าเราจะสร้างขึ้นมาเองให้รับแบบหลายค่าแบบนี้ได้บ้าง ทำได้ตามตัวอย่างนี้

```
func add(args ...int) int {
    total := 0
    for _, v := range args {
        total += v
    }
    return total
}
func main() {
    fmt.Println(add(1,2,3))
}
```

โดยประเภทของพารามิเตอร์ เราจะใส่ ... เข้าไปหน้าประเภทข้อมูล ข้อบังคับอย่างหนึ่งคือพารามิเตอร์แบบนี้ ต้องเป็นลำดับสุดท้ายของฟังก์ชันเท่านั้น เอาไปอยู่ก่อนหน้าพารามิเตอร์อื่นไม่ได้ ตอนเรียกใช้งาน เราสามารถส่งค่าให้พารามิเตอร์นี้หลายๆค่า หรือไม่ส่งเลยก็ได้

นี่คือตัวอย่างของ signature ของฟังก์ชัน Println ที่รับค่าหลายค่า

```
func Println(a ...interface{}) (n int, err error)
```

โดยค่าที่รับเป็นประเภทข้อมูลแบบ interface{} เป็นประเภทพิเศษที่เราจะดูกันอีกทีในบทที่ 9
นอกจากการส่งค่าพารามิเตอร์ทีละค่าแล้ว เรายังสามารถใช้ข้อมูลแบบ slices ไปได้โดยเติม ... ด้านหลังตัวแปร slice ตอนเรียกใช้ฟังก์ชัน

```
func main() {
    xs := []int{1,2,3}
    fmt.Println(add(xs...))
}
```

Closure

closure คือฟังก์ชันที่สร้างขึ้นมาโดยไม่จำเป็นต้องมีชื่อ และสามารถกำหนดให้กับตัวแปรได้ หรือ จะส่งเป็นข้อมูลเข้าออกจากฟังก์ชันอื่นได้ ตัวอย่างเราสร้าง function ขึ้นมาภายในตัวฟังก์ชันได้ เช่น

```
func main() {
    add := func(x, y int) int {
        return x + y
    }
    fmt.Println(add(1,1))
}
```

จากโค้ด add เป็นตัวแปรโลคอล ที่เรากำหนด closure ให้ จะเห็นว่าตรงนี้เราประกาศฟังก์ชันโดยใช้ แค่ func ตามด้วยลิสต์ของพารามิเตอร์ และ ประเภทข้อมูลส่งกลับ แล้วก็ตามด้วย body ของฟังก์ชันเลย ซึ่งตัวแปร add จะมีประเภทข้อมูลเป็น func(int, int) int จะเห็นว่าเวลาเราพูดถึงประเภทข้อมูลของตัวแปรที่เก็บฟังก์ชัน จะบอกแค่ ประเภทข้อมูลของพารามิเตอร์และประเภทข้อมูลที่ส่งกลับ ไม่ได้สนใจชื่อของมัน เช่น add บอกว่าเป็นฟังก์ชันที่รับ int สองค่า และส่งกลับค่า int
จุดสำคัญของ closure หรือการประกาศฟังก์ชันภายในฟังก์ชันนี้ก็คือในขอบเขตของตัวฟังก์ชัน closure จะเรียกใช้ตัวแปรโลคอลภายในฟังก์ชันที่สร้างมันขึ้นมาได้ด้วย เช่น

```
func main() {
    x := 0
    increment := func() int {
        x++
        return x
    }
    fmt.Println(increment())
    fmt.Println(increment())
}
```

จะได้ผลลัพธ์ออกมาเป็น

1
2

ฟังก์ชัน increment ถูกสร้างขึ้นมาภายในฟังก์ชัน main โดยทำการเพิ่มค่าให้กับตัวแปร x ที่ถูกประกาศไว้ภายในฟังก์ชัน main เช่นกัน

อีกวิธีหนึ่งในการใช้ closure คือเราจะเขียนฟังก์ชัน ที่ส่งค่าออกมาได้เป็นฟังก์ชันอื่น ซึ่งเมื่อเอามาเรียกใช้จะยังคงรักษาค่าของตัวแปรโลคอลของฟังก์ชันที่สร้างมันมา ทำให้เราเอามาใช้สร้างการ generate ลำดับของตัวเลขได้ เช่นตัวอย่างนี้เราจะสร้างฟังก์ชัน ที่ generate ตัวเลขคู่ออกมา

```
func makeEvenGenerator() func() uint {
    i := uint(0)
    return func() (ret uint) {
        ret = i
        i += 2
        return
    }
}
func main() {
    nextEven := makeEvenGenerator()
    fmt.Println(nextEven()) // 0
    fmt.Println(nextEven()) // 2
    fmt.Println(nextEven()) // 4
}
```

ในฟังก์ชัน makeEvenGenerator เองนั้นกำหนดตัวแปร i และค่าเริ่มต้นเอาไว้เป็น 0 และทำการ return closure ฟังก์ชันออกมา โดยที่ตัว closure ฟังก์ชันจะเอาตัวแปร i มาบวกเพิ่มไปทีละ 2 แล้วส่งกลับ เมื่อเราเรียก makeEvenGenerator() แล้วกำหนดให้ตัวแปร nextEven แล้วเอา nextEven ไปเรียกใช้ ก็จะได้ค่าเลขคู่ออกมา ที่เกิดจากการจำสถานะของค่า i ของฟังก์ชัน makeEvenGenerator นั่นเอง

ฟังก์ชันเรียกตัวเอง

รูปแบบสุดท้ายในการเรียกฟังก์ชันของบทนี้ก็คือ ฟังก์ชันเรียกตัวเอง ตัวอย่างของการใช้งานฟังก์ชันเรียกตัวเองเช่น ฟังก์ชันการคำนวณค่า factorial

```
func factorial(x uint) uint {
    if x == 0 {
        return 1
    }
    return x * factorial(x-1)
}
```

จะเห็นว่าภายในฟังก์ชัน factorial จะมีการเรียก factorial ใช้งานตรงจุดที่มีการ return ซึ่งเป็นการเรียกตัวเองของฟังกีชัน factorial เพื่อทำความเข้าในการทำงานของฟังก์ชันเรียกตัวเอง ลองไล่ดูการทำงานเมื่อเราเรียกใช้ factorial(2):

- ตรวจว่า x == 0 หรือไม่? ไม่ (เพราะตอนนี้ x คือ 2)
- หาค่าของ factorial ของ x - 1
    - ตรวจว่า x == 0 หรือไม่? ไม่ (เพราะตอนนี้ x คือ 1)
    - หาค่าของ factorial ของ x - 1
        - ตรวจว่า x == 0 หรือไม่? ใช่ ส่งค่า 1 กลับไป
    - ส่งค่า 1 * 1
- ส่งค่า 2 * 1

ทั้ง Closure และฟังก์ชันเรียกตัวเองเป็นเทคนิคที่ทรงพลังมากเพราะเป็นรูปแบบที่ทำให้เราเขียนโปรแกรมในลักษณะเชิงฟังก์ชันได้ (Functional Programming) แม้ว่าคนส่วนใหญ่จะยังคงคิดว่าการเขียนโปรแกรมในลักษณะนี้จะยากในการทำความเข้าใจกว่าการใช้การวนซ้ำด้วย for, การเช็คเงื่อนไขด้วย if, ตัวแประ และ การเรียกฟังก์ชันธรรมดาก็ตาม

Defer, Panic & Recover

โก มีคำสั่งพิเศษที่ชื่อว่า defer ซึ่งจะใช้กำหนดฟังก์ชันที่จะถูกเรียกใช้งาน เมื่อฟังก์ชันหลักที่ครอบ defer อยู่ทำงานเสร็จ ลองดูตัวอย่างต่อไปนี้

```
package main

import "fmt"

func first() {
    fmt.Println("1st")
}
func second() {
    fmt.Println("2nd")
}
func main() {
    defer second()
    first()
}
```

โปรแกรมนี้จะพิมพ์ 1st ตามด้วย 2nd เพราะว่าเราสั่ง defer แล้วตามด้วยการเรียก second() ทำให้ second() ถูกเรียกเมื่อฟังก์ชัน main() ทำงานเสร็จแล้วนั่นคือทำ first() เสร็จก่อน เมื่อเรียกลำดับจะเห็นว่า second() จะทำงานลำดับสุดท้ายของ main() เสมอแบบนี้

```
func main() {
    first()
    second()
}
```

defer มักจะถูกเอาไปใช้กับการคือค่าของทรัพยากรระบบเมื่อใช้งานเสร็จ ตัวอย่างเช่นเราทำการเปิดไฟล์ข้อมูลมาใช้งาน ต้องแน่ใจว่าสุดท้ายเมื่อเลิกใช้ไฟล์นั้นแล้วต้องปิดไฟล์นั้น เราจะใช้ defer มาช่วยได้ดังนี้

```
f, _ := os.Open(filename)
defer f.Close()
```

การใช้ defer ช่วยจัดการมีประโยชน์หลัก 3 อย่างดังนี้ (1) โค้ดที่สั่งปิดเราจะเห็นใกล้ๆกับตอนเปิด ทำให้ง่ายในการทำความเข้าใจ (2) ถึงแม้ว่าฟังก์ชันเราจะมีเงื่อนไขในการส่งค่ากลับหลายแบบ แต่การปิดไฟล์จะถูกเรียกเสมอ และ (3) ฟังก์ชันที่กำหนดให้ defer จะถูกเรียกเสมอแม้ว่าเกิดกรณี panic ขึ้นมาระหว่างรันโปรแกรม

Panic & Recover

บทที่แล้วเราได้เห็นกรณีเกิดข้อผิดพลาดขนาดรัน ซึ่งระบบจะเรียกฟังก์ชัน panic เพื่อส่งข้อความบอกข้อผิดพลาดออกมา เราสามารถจัดการกับ panic ที่เกิดขึ้นระหว่างโปรแกรมทำงานได้ โดยใช้ฟังก์ชัน recover ซึ่ง recover จะหยุดการ panic แล้วส่งค่ากลับไปให้กับจุดที่เกิด panic เราจะลองแกล้งทำให้เกิด panic ขึ้นเพื่อดูตัวอย่างการใช้ recover ดังนี้

```
package main

import "fmt"

func main() {
    panic("PANIC")
    str := recover()
    fmt.Println(str)
}
```

แต่การเขียน recover เอาไว้หลัง panic แบบนี้จะไม่เกิดอะไรขึ้นเพราะเมื่อเรียก panic จะทำให้โปรแกรมหยุดการทำงานลง ดังนั้นเราจะเรียก recover ผ่าน defer แทนดังนี้

```
package main

import "fmt"

func main() {
    defer func() {
        str := recover()
        fmt.Println(str)
    }()
    panic("PANIC")
}
```

panic นั้นโดยทั่วไปเราจะใช้ระบุข้อผิดพลาดที่เกิดขึ้นจากโปรแกรมเมอร์ (เช่นการพยายามเข้าถึงข้อมูลของอาเรย์นอกขอบเขตที่กำหนด, การลืมกำหนดค่าเริ่มต้นให้กับข้อมูลแบบ แมพ, ฯลฯ) หรือ เงื่อนไขพิเศษอื่นๆ ที่ยากในการที่จะ recover (เพราะแบบนี้เลยเรียกข้อผิดพลาดว่า “panic”)

ปัญหาท้ายบท

- sum คือฟังก์ชันที่รับค่าข้อมูลแบบ สไลซ์ ของตัวเลข และหาผลรวมทั้งหมดของตัวเลขที่อยู่ในสไลซ์ จงเขียนฟังก์ชัน signature ของ sum ที่รองรับการทำงานแบบนี้
- จงเขียนฟังก์ชันที่รับเลขจำนวนเต็ม และ ส่งค่ากลับสองค่า คือ ค่าหารสองของจำนวนนั้น และ true ถ้าจำนวนนั้นเป็นเลขคู่ หรือ false ถ้าจำนวนนั้นเป็นเลขคี่ เช่น half(1) ให้ส่งค่า (0, false) และ half(2) ควรจะส่งค่า (1, true) กลับไป
- จงเขียนฟังก์ชันที่รับค่าพารามิเตอร์ 1 ค่าแต่เป็นแบบไม่จำกัด (variadic parameter) เพื่อหาค่าที่มากที่สุดของลิสต์ตัวเลขที่ส่งมา
- จากฟังก์ชัน makeEvenGenerator ตามตัวอย่าง จงเขียนฟังก์ชัน makeOddGenerator เพื่อให้สร้างลำดับของเลขคี่ออกมา
- ลำดับของฟิโบนัสชีนั้นถูกนิยามไว้ดังนี้ fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2) จงเขียนฟังก์ชันแบบเรียกซ้ำเพื่อคำนวณค่าของ fib(n)
- อะไรคือ defer, panic และ recover และ เราจะแก้ไขข้อผิดพลาด panic ที่เกิดขณะรันโปรแกรมได้อย่างไร