Golang FP

Functional Programming (FP: Interrediate)

(Integrative-Generative AI Edition)

Student Price Book Center AI

คำนำ

หนังสือวิชาการ Golang Functional Programming (FP): Intermediate

การเขียนโปรแกรมเชิงฟังก์ชัน (Functional Programming: FP) กลายเป็นแนวทางสำคัญในการพัฒนาโปรแกรมยุคใหม่ที่มุ่งเน้นความกระชับ ความสามารถในการทดสอบ ความสามารถในการจัดการข้อมูลอย่างปลอดภัย และการรองรับระบบที่ซับซ้อนอย่างยั่งยืน ภาษา Go (Golang) แม้จะไม่ได้ถูกออกแบบมาให้เป็นภาษา functional โดยตรง แต่ด้วยคุณสมบัติด้าน **first-class functions**, **closures**, **goroutines**, และระบบ type ที่ยืดหยุ่น ทำให้สามารถประยุกต์แนวคิด FP เข้ามาสร้างโปรแกรมที่มีประสิทธิภาพและ maintainable ได้อย่างเต็มรูปแบบ หนังสือเล่มนี้จึงถูกจัดทำขึ้นเพื่อเป็นแนวทางเชิงลึกสำหรับผู้ที่มีพื้นฐาน FP ในระดับต้น และต้องการก้าวสู่ระดับกลาง (Intermediate) ในบริบทของ Go อย่างเป็นระบบ

หนังสือเล่มนี้แบ่งเนื้อหาออกเป็น **5 บทหลัก** (บทที่ 6–10) ครอบคลุมหัวข้อสำคัญของ FP ที่เป็นแกนกลางในการสร้างโปรแกรมที่ซับซ้อนและ reusable ได้จริงในภาษา Go โดยมุ่งเน้นทั้ง **แนวคิดเชิงทฤษฎี** และ **การประยุกต์เชิงปฏิบัติ** พร้อมตัวอย่างโปรแกรมตั้งแต่ระดับบูรณาการจนถึงระดับ Ultimate เพื่อให้ผู้อ่านสามารถนำไปต่อยอดในโปรเจกต์จริงได้อย่างมั่นใจ

เริ่มต้นที่ **บทที่ 6: Higher-Order Functions** ผู้อ่านจะได้ทำความเข้าใจฟังก์ชันระดับสูงที่สามารถรับฟังก์ชันเป็น argument และ return ฟังก์ชันเป็นผลลัพธ์ ซึ่งเป็นหัวใจสำคัญของ FP การใช้ higher-order functions ทำให้สามารถออกแบบ pipeline ของการประมวลผลข้อมูลได้อย่างชัดเจน ยืดหยุ่น และ modular นอกจากนี้ยังมีตัวอย่างการสร้าง **Super Ultimate Pipeline** พร้อม dashboard/visualization เพื่อให้เห็นภาพการทำงานเชิงระบบอย่างครบถ้วน

ต่อมา **บทที่ 7: Closures** จะเจาะลึกเทคนิคการสร้างและใช้งาน closure ใน Go เพื่อเก็บ state ภายในแบบ local โดยไม่ต้องใช้ global variables ผู้อ่านจะได้เรียนรู้การออกแบบ factory functions และเทคนิคขั้นสูงในการสร้าง CLI dashboard แบบ interactive ที่ใช้ closures เป็นแกนหลักของระบบ ซึ่งช่วยให้โปรแกรมมีความยืดหยุ่น ปลอดภัยจาก side effects และสามารถปรับขยายได้ง่าย

บทที่ 8: Recursion จะนำเสนอการใช้ฟังก์ชันแบบ recursive ในการแก้ปัญหาเชิง functional รวมถึงแนวทางการออกแบบ **tail recursion** เพื่อจัดการ stack limitation ของ Go พร้อมตัวอย่าง Fibonacci และ factorial แบบ functional รวมถึงชุดโปรเจกต์ Ultimate ที่รวม recursion เข้ากับแนวคิด FP อื่น ๆ เพื่อสร้างระบบที่มีความซับซ้อนระดับสูงในลักษณะ Master Suite

หัวข้อสำคัญถัดมาคือ **บทที่ 9: Functional Error Handling** ซึ่งถือเป็นหนึ่งในความท้าทายของการประยุกต์ FP ในภาษา Go เนื่องจาก Go ไม่มี Either/Option type โดยตรง บทนี้จะสอนการใช้ pattern แบบ FP ในการจัดการ error ผ่านการ return multiple values (value, error) และการสร้าง helper functions สำหรับ chaining เพื่อให้สามารถสร้าง pipeline ของฟังก์ชันที่รองรับ error ได้อย่างเป็นระบบ ช่วยให้โค้ดมีความ pure, readable และ maintainable

สุดท้าย **บทที่ 10: Comprehensions / Iterators** จะเปิดมุมมองใหม่ในการสร้าง flow ของข้อมูลเชิง functional ใน Go แม้จะไม่มี list comprehension แบบภาษาอื่น แต่สามารถใช้ higher-order functions เช่น map, filter, reduce ร่วมกับ **lazy evaluation ผ่าน channels และ goroutines** เพื่อสร้าง iterators แบบ functional ได้อย่างทรงพลัง บทนี้จะรวมแนวคิดทั้งหมดเข้าด้วยกันผ่านโปรเจกต์บูรณาการ Ultimate และ Super Ultimate ที่จำลองสถานการณ์การประมวลผลข้อมูลจริงขนาดใหญ่

นอกจากการอธิบายหลักการและโค้ดตัวอย่างแล้ว หนังสือเล่มนี้ยังให้ความสำคัญกับการออกแบบระบบ การเขียนโค้ดเชิง declarative และการสร้าง abstraction ที่เหมาะสม เพื่อให้ผู้อ่านสามารถนำไปใช้ในโปรเจกต์ขนาดกลางถึงใหญ่ได้จริง รูปแบบการเขียนอธิบายจะค่อยเป็นค่อยไป พร้อมโค้ดตัวอย่างที่ครบถ้วนและสามารถนำไปทดลองได้ทันที

เราหวังเป็นอย่างยิ่งว่าหนังสือ **Golang Functional Programming (FP): Intermediate** เล่มนี้จะเป็นคู่มือสำคัญสำหรับนักพัฒนาและนักวิจัยที่ต้องการยกระดับทักษะ FP ในภาษา Go จากพื้นฐานสู่การออกแบบโปรแกรมระดับสูงอย่างมีระบบ และเป็นสะพานเชื่อมสู่แนวทาง Functional Programming ขั้น Advance และ Professional ในอนาคต

ด้วยความปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

บทที่ 6

Higher-Order Functions

(Higher-Order Functions)

เนื้อหา

บทที่ 6: Higher-Order Functions

ในภาษา Go หนึ่งในคุณสมบัติสำคัญที่ช่วยยกระดับการเขียนโปรแกรมเชิงฟังก์ชันคือ **Higher-Order Functions (HOFs)** ฟังก์ชันประเภทนี้มีความสามารถพิเศษในการ **รับฟังก์ชันเป็น argument** หรือ **return ฟังก์ชันเป็นผลลัพธ์** ทำให้เราสามารถสร้างโค้ดที่ยืดหยุ่น modular และ reusable ได้มากยิ่งขึ้น การเรียนรู้และประยุกต์ใช้ HOFs เป็นก้าวสำคัญในการเข้าใจการเขียนโค้ดแบบ functional ที่ซับซ้อนและทรงพลังใน Go

ฟังก์ชันที่รับฟังก์ชันเป็น argument เป็นตัวอย่างที่ชัดเจนของความสามารถของ HOFs ฟังก์ชันหลักสามารถรับพฤติกรรมเฉพาะเป็น parameter แล้วนำไปประมวลผลต่อ เช่น การส่งฟังก์ชันที่ระบุเงื่อนไขการกรองข้อมูล หรือฟังก์ชันที่กำหนดการแปลงค่า เทคนิคนี้ช่วยให้โค้ดกระชับและปรับเปลี่ยนพฤติกรรมได้โดยไม่ต้องแก้ไขฟังก์ชันหลัก

อีกประเภทของ HOFs คือ **ฟังก์ชันที่ return ฟังก์ชัน** ฟังก์ชันเหล่านี้สามารถสร้างฟังก์ชันใหม่ตามพารามิเตอร์หรือบริบทที่ได้รับ ซึ่งทำให้เราสามารถสร้างโค้ดที่เป็น dynamic และปรับเปลี่ยนพฤติกรรมได้ตามต้องการ ตัวอย่างเช่น การสร้างตัวคูณหรือตัวกรองเฉพาะตัวโดยใช้ฟังก์ชัน factory pattern ซึ่งสามารถสร้างฟังก์ชันใหม่แต่ละตัวที่มีพฤติกรรมแตกต่างกันได้

การใช้ HOFs ร่วมกับ **first-class functions และ anonymous functions** ทำให้เราสามารถสร้าง **pipelines** ของฟังก์ชันได้อย่างชัดเจน โค้ดสามารถเรียงลำดับการประมวลผลข้อมูลเป็นขั้นตอนต่อเนื่อง เช่น การ map ⟶ filter ⟶ reduce โดยแต่ละขั้นตอนถูกนิยามเป็นฟังก์ชันแยก การทำ pipelines ช่วยให้ตรรกะของโปรแกรมอ่านง่าย ลดความซับซ้อน และส่งเสริมการเขียนโค้ดแบบ declarative

เทคนิค pipelines ยังช่วยให้เราสามารถ **แยกความรับผิดชอบของแต่ละฟังก์ชัน** ออกจากกัน ทำให้โค้ด modular และง่ายต่อการทดสอบ ตัวอย่างเช่น สามารถสร้าง pipeline สำหรับการแปลงค่าข้อมูล text, ตัวเลข หรือ struct โดยแต่ละขั้นตอนทำหน้าที่เฉพาะตัว ทำให้โค้ด maintainable และ reusable

นอกจากนี้ Higher-Order Functions ยังเป็นกุญแจสำคัญในการออกแบบ **DSL (Domain-Specific Language) ขนาดเล็ก** หรือ **API แบบ functional** ในโปรเจกต์ Go การออกแบบโค้ดเช่นนี้ช่วยให้ฟังก์ชันถูกเรียกใช้ได้ง่าย ปรับเปลี่ยนได้อย่างยืดหยุ่น และลดการซ้ำซ้อนของโค้ด
การเข้าใจและฝึกประยุกต์ใช้ HOFs เป็นพื้นฐานสำคัญที่จะช่วยให้ผู้พัฒนาสามารถสร้าง **โปรแกรม functional ที่ซับซ้อนและยืดหยุ่น** ได้ใน Go นอกจากนี้ยังช่วยให้โค้ดอ่านง่าย สื่อความหมายชัดเจน และรองรับการทำงานแบบ concurrent หรือ asynchronous ได้ดียิ่งขึ้น

บทนี้จะพาผู้อ่านไปเรียนรู้แนวคิด Higher-Order Functions ตั้งแต่พื้นฐานจนถึงการทำ pipelines ของฟังก์ชัน พร้อมตัวอย่างเชิงปฏิบัติ Ultimate Project ที่สามารถรันได้จริง เพื่อให้ผู้อ่านสามารถนำแนวคิดนี้ไปปรับใช้ในโปรแกรมจริงได้อย่างมั่นใจ

Higher-Order Functions

- ฟังก์ชันที่รับฟังก์ชันเป็น argument
- ฟังก์ชันที่ return ฟังก์ชัน
- การทำ pipelines ด้วยฟังก์ชัน

บทที่ 6 — Higher-Order Functions (อธิบายโดยละเอียด)
สรุปสั้น ๆ ก่อน:
Higher-Order Function (HOF) คือฟังก์ชันที่ **รับฟังก์ชันเป็นพารามิเตอร์** หรือ **คืนค่าฟังก์ชัน** (หรือทั้งสองอย่าง) — เป็นหัวใจของแนวคิด Functional Programming เพราะช่วยให้เราสร้างโค้ดที่ยืดหยุ่น, นำกลับมาใช้ใหม่ได้ และเขียนเป็นชุดของตัวแปลงข้อมูล (pipelines) ได้ง่ายขึ้น
ต่อไปนี้จะอธิบายรายละเอียด 3 หัวข้อที่คุณขอ: (1) ฟังก์ชันรับฟังก์ชันเป็น argument (2) ฟังก์ชันที่ return ฟังก์ชัน (closures, currying) และ (3) การทำ pipelines ด้วยฟังก์ชัน — **มีตัวอย่างเป็นโค้ด Go (พร้อมคำอธิบาย)**

---

1) ฟังก์ชันที่รับฟังก์ชันเป็น argument

แนวคิด: ฟังก์ชันหนึ่งรับอีกฟังก์ชันหนึ่งเข้าไปเพื่อเปลี่ยนพฤติกรรมโดยไม่ต้องแก้โค้ดภายใน — ตัวอย่างทั่วไปคือ map, filter, sort ที่รับฟังก์ชันเป็น predicate/transformer/comparator
ตัวอย่างพื้นฐาน (ไม่ใช้ generics):

```
package main

import "fmt"

// apply: รับ slice ของ int และฟังก์ชันที่แปลง int⟶int
func apply(nums []int, fn func(int) int) []int {
    out := make([]int, len(nums))
    for i, v := range nums {
        out[i] = fn(v)
    }
    return out
}

func main() {
    nums := []int{1, 2, 3, 4}
    doubled := apply(nums, func(x int) int { return x * 2 })
    fmt.Println(doubled) // [2 4 6 8]
}
```

ข้อดี

- แยก logic การวนลูปออกจากการประมวลผลแต่ละตัว (separation of concerns)
- ยืดหยุ่น — เปลี่ยนพฤติกรรมได้โดยไม่แตะส่วนวนลูป

ตัวอย่างระดับถัดไป — ใช้ Generics (Go 1.18+) เพื่อทำ Map ที่ใช้ได้กับทุกชนิด:

```
package main

import "fmt"

func Map[T any, R any](arr []T, fn func(T) R) []R {
    res := make([]R, len(arr))
    for i, v := range arr {
        res[i] = fn(v)
    }
```

```
    return res
}

func main() {
    s := []string{"a", "bb", "ccc"}
    lens := Map(s, func(x string) int { return len(x) })
    fmt.Println(lens) // [1 2 3]
}
```

Pattern ที่พบบ่อย

- Filter(arr, predicate func(T) bool) []T
- Reduce(arr, reducer func(acc, T) accType, init) accType
- Sort มักใช้ comparator เป็น argument (sort.Slice ใน stdlib)

---

2) ฟังก์ชันที่ return ฟังก์ชัน (Closures / Currying / Factories)

แนวคิด: ฟังก์ชันสร้างและคืนฟังก์ชันใหม่ — มักใช้สร้าง *factory* (ฟังก์ชันที่กำหนดค่าคงที่บางอย่างให้กับฟังก์ชันย่อย) หรือ *decorator* ที่ห่อหุ้มพฤติกรรม

ตัวอย่าง closure (adder):

```
package main

import "fmt"

func Adder(start int) func(int) int {
    sum := start
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    a := Adder(10)
    fmt.Println(a(5))  // 15
    fmt.Println(a(2))  // 17
}
```

อธิบาย: ฟังก์ชันที่คืนค่าจำค่า sum ในบริบทของมันไว้ — นี่คือ *closure* (มี lexical scope)

ตัวอย่าง Currying / Partial application:

```
func Multiply(a int) func(int) int {
    return func(b int) int { return a * b }
}

mul3 := Multiply(3)
fmt.Println(mul3(4)) // 12
```

ตัวอย่างจริง (HTTP middleware factory) — ฟังก์ชันที่รับ http.Handler แล้วคืน http.Handler (pattern ที่ใช้มากในเว็บ Go):

```
package main

import (
    "log"
    "net/http"
)

func LoggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        log.Printf("method=%s path=%s", r.Method, r.URL.Path)
        next.ServeHTTP(w, r)
    })
}

func Hello(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("hello"))
}

func main() {
    http.Handle("/", LoggingMiddleware(http.HandlerFunc(Hello)))
    http.ListenAndServe(":8080", nil)
}
```

ข้อควรระวังเกี่ยวกับ closures

- การจับตัวแปรจาก outer scope อาจทำให้เกิด *bug* (เช่น capture loop variable) — ตัวอย่าง common pitfall:

```
func bad() {
    funcs := make([]func() int, 3)
    for i := 0; i < 3; i++ {
        funcs[i] = func() int { return i } // ทุกฟังก์ชันจะคืนค่า 3 เมื่อเรียก
    }
    // แก้โดยจับค่าลงตัวแปรใหม่ j := i ภายใน loop
}
```

---

3) การทำ Pipelines ด้วยฟังก์ชัน

แนวคิด: เอาผลลัพธ์ของฟังก์ชันหนึ่งไปเป็นอินพุตของอีกฟังก์ชันหนึ่ง ⟶ ทำให้เราสร้าง "สายการแปรรูป" (pipeline) ของการแปลงข้อมูลได้ชัดเจนและยืดหยุ่น

การทำ composition แบบง่าย (สองฟังก์ชัน)

```
func Compose[A, B, C any](f func(B) C, g func(A) B) func(A) C {
    return func(a A) C {
        return f(g(a))
    }
}
```

ตัวอย่างใช้ Compose:

```
package main

import (
    "fmt"
    "strings"
)

func ToUpper(s string) string { return strings.ToUpper(s) }
func AddExcl(s string) string { return s + "!" }

func Compose[A, B, C any](f func(B) C, g func(A) B) func(A) C {
    return func(a A) C {
        return f(g(a))
    }
```

```
}

func main() {
    shout := Compose[string, string, string](AddExcl, ToUpper)
    fmt.Println(shout("hello")) // HELLO!
}
```

Pipeline แบบหลายฟังก์ชัน (same-type transformations)

ถ้าเรามีชุดฟังก์ชันที่รับและคืนค่าเป็นชนิดเดียวกัน เราสามารถสร้าง Chain:

```
package main

import (
    "fmt"
    "strings"
)

func Chain[T any](fns ...func(T) T) func(T) T {
    return func(x T) T {
        for _, f := range fns {
            x = f(x)
        }
        return x
    }
}

func Trim(s string) string  { return strings.TrimSpace(s) }
func Lower(s string) string { return strings.ToLower(s) }
func AddPeriod(s string) string { return s + "." }

func main() {
    pipeline := Chain[string](Trim, Lower, AddPeriod)
    fmt.Println(pipeline("  HELLO GoLang  ")) // "hello golang."
}
```

ประโยชน์ของ pipeline

- อ่านง่าย: แสดงลำดับการแปลงข้อมูลเป็นจุดเดียว

- ง่ายต่อการปรับ/เพิ่มขั้นตอน: เพิ่ม/ลบฟังก์ชันจาก chain ได้โดยไม่แตะส่วนอื่น
- ทดสอบง่าย: ทดสอบแต่ละฟังก์ชันแยกกัน

ตัวอย่าง pipeline กับ slice (map ⟶ filter ⟶ reduce)

รวมแนวคิด HOF ทั้งหมด:

```
package main

import "fmt"

func Map[T any, R any](arr []T, fn func(T) R) []R {
    res := make([]R, 0, len(arr))
    for _, v := range arr {
        res = append(res, fn(v))
    }
    return res
}

func Filter[T any](arr []T, pred func(T) bool) []T {
    res := make([]T, 0)
    for _, v := range arr {
        if pred(v) {
            res = append(res, v)
        }
    }
    return res
}

func Reduce[T any, R any](arr []T, fn func(R, T) R, init R) R {
    acc := init
    for _, v := range arr {
        acc = fn(acc, v)
    }
    return acc
}
```

```
func main() {
    nums := []int{1, 2, 3, 4, 5, 6}

    doubled := Map(nums, func(n int) int { return n * 2 })
    evens := Filter(doubled, func(n int) bool { return n%2 == 0 })
    sum := Reduce(evens, func(acc, n int) int { return acc + n }, 0)

    fmt.Println("doubled:", doubled) // [2 4 6 8 10 12]
    fmt.Println("evens:", evens)     // [2 4 6 8 10 12] (ทั้งหมดเป็นคู่)
    fmt.Println("sum:", sum)         // 42
}
```

---

4) Patterns & Practical Uses (รวบรัด)

- **Decorator / Middleware**: ฟังก์ชันที่รับ handler และคืน handler (เว็บ, RPC, validation)
- **Factory**: คืนฟังก์ชันที่มี configuration ฝังอยู่ (เช่น Retry(times) คืนฟังก์ชัน wrapper)
- **Memoization**: คืนเวอร์ชันที่เก็บ cache ของฟังก์ชันบริสุทธิ์ (ต้องระวัง concurrency)
- **Composition libraries**: สร้างชุดฟังก์ชัน Compose/Chain/Pipe เพื่อเขียนโค้ดสั้นและชัดเจน

---

5) ข้อดี / ข้อควรระวัง (สรุป)

ข้อดี

- โค้ดยืดหยุ่น, นำกลับมาใช้ได้, อ่านเป็น flow ดี (เมื่อออกแบบดี)
- ทำ unit testing ง่าย: ทดสอบแต่ละฟังก์ชันหน่วยได้

ข้อควรระวัง

- อาจบ่อน้ำหนัก performance ถ้ามีการสร้าง closure เยอะ ๆ หรือการ allocate บ่อย ๆ (ต้องวัดก่อน optimize)
- อาจยากต่อการ debug ถ้าซ้อนฟังก์ชันหลายชั้น
- การจับตัวแปรภายนอก (closures) อาจเกิด bug หรือ memory retention ได้
- Currying/Composition แบบ generic อาจซับซ้อนในกรณีชนิดต่างกัน (ต้องออกแบบ signature ให้เหมาะ)

Higher-Order Functions (รายละเอียดเชิงลึก)

ต่อไปนี้เป็นคำอธิบายเชิงลึก (ทฤษฎี + แนวปฏิบัติใน Go) พร้อมตัวอย่างโค้ดพร้อมอธิบาย — ครอบคลุมฟีเจอร์หลัก, แบบแผน (patterns), ข้อควรระวัง, การทดสอบ และประเด็น performance/parallelism ที่สำคัญ

เบื้องต้น (สั้น)

Higher-Order Function (HOF) = ฟังก์ชันที่รับฟังก์ชันเป็นพารามิเตอร์ หรือคืนค่าฟังก์ชัน (หรือทั้งสองอย่าง)

จุดประสงค์หลัก: แยกพฤติกรรม (behavior) ออกจากการไหลของข้อมูล (control flow/data flow), ทำให้สร้าง pipeline, middleware, factory, decorator, partial application ได้ง่าย

---

1. ฟังก์ชันที่รับฟังก์ชันเป็น argument — (HOF as consumer)

แนวคิด

- แยก *what* (การแปลง/การตรวจสอบ) ออกจาก *how* (วนลูป, error handling, concurrency)
- ทำให้สร้าง Map, Filter, Reduce, Sort แบบยืดหยุ่นได้

ตัวอย่าง: Generic Map / Filter / Reduce (Go ≥ 1.18)

```
package main

import "fmt"

func Map[T any, R any](arr []T, fn func(T) R) []R {
    res := make([]R, 0, len(arr))
    for _, v := range arr {
        res = append(res, fn(v))
    }
    return res
}

func Filter[T any](arr []T, pred func(T) bool) []T {
    res := make([]T, 0)
    for _, v := range arr {
        if pred(v) {
            res = append(res, v)
        }
    }
    return res
}
```

```
func Reduce[T any, R any](arr []T, fn func(R, T) R, init R) R {
    acc := init
    for _, v := range arr {
        acc = fn(acc, v)
    }
    return acc
}

func main() {
    nums := []int{1, 2, 3, 4, 5}
    doubled := Map(nums, func(x int) int { return x * 2 })
    even := Filter(doubled, func(x int) bool { return x%2 == 0 })
    sum := Reduce(even, func(acc, n int) int { return acc + n }, 0)
    fmt.Println(doubled, even, sum) // [2 4 6 8 10] [2 4 6 8 10] 30
}
```

ข้อแนะนำ

- ถ้าเป็น hot-path (เรียกบ่อยมาก) ให้ตรวจสอบการ allocate (append, make) และการ escape ของค่าที่ถูกจับใน closures
- หากต้องการลด allocation ให้เตรียม capacity หรือเขียนเวอร์ชันที่แก้ในที่เดิม (in-place) เมื่อปลอดภัย

---

2. ฟังก์ชันที่ return ฟังก์ชัน — (Closures, Factories, Currying)

Closure / Factory

Closure = ฟังก์ชันที่ “จำ” (capture) ค่าจาก outer scope ไว้

ตัวอย่าง: Adder (stateful closure)

```
func Adder(start int) func(int) int {
    sum := start
    return func(x int) int {
        sum += x
        return sum
    }
}
```

การใช้งาน:

```
a := Adder(10)
```

```
fmt.Println(a(5))  // 15
fmt.Println(a(3))  // 18
```

Currying / Partial application

Go ไม่มี syntax currying อัตโนมัติ แต่ทำได้ด้วย closure:

```
func Multiply(a int) func(int) int {
    return func(b int) int { return a * b }
}
mul2 := Multiply(2)
fmt.Println(mul2(5)) // 10
```

Factory ตัวใช้งานจริง — Retry wrapper (support context)

```
import (
    "context"
    "time"
)

func Retry(ctx context.Context, attempts int, delay time.Duration) func(func() error) error {
    return func(fn func() error) error {
        var lastErr error
        for i := 0; i < attempts; i++ {
            if err := fn(); err == nil {
                return nil
            } else {
                lastErr = err
            }
            select {
            case <-time.After(delay):
            case <-ctx.Done():
                return ctx.Err()
            }
        }
        return lastErr
    }
}
```

ข้อดี: แยกนโยบายการ retry ออกจากงานจริง ทำให้ทดสอบง่าย

3. Composition & Pipelines (การนำฟังก์ชันมาต่อกัน)

Compose แบบทั่วไป (different types)

```
func Compose[A, B, C any](f func(B) C, g func(A) B) func(A) C {
    return func(a A) C {
        return f(g(a))
    }
}
```

ใช้งาน:

```
toUpper := func(s string) string { return strings.ToUpper(s) }
addExcl := func(s string) string { return s + "!" }
shout := Compose(addExcl, toUpper)
fmt.Println(shout("hello")) // HELLO!
```

Chain (same-type transformations) — variadic

```
func Chain[T any](fns ...func(T) T) func(T) T {
    return func(x T) T {
        for _, f := range fns {
            x = f(x)
        }
        return x
    }
}
```

Pipeline with slices (map⟶filter⟶reduce) — compose style

คุณสามารถประกอบ functions ให้เป็น pipeline (อ่านง่าย) เช่น:

```
pipeline := Chain[string](strings.TrimSpace, strings.ToLower, func(s string) string { return s + "." 
})
fmt.Println(pipeline("  HELLO GoLang  ")) // "hello golang."
```

Streaming pipelines (channels + goroutines) — example แบบ cancelable

ประยุกต์ HOF กับ concurrency — ดีสำหรับ stream processing / ETL:

```
package main

import (
    "context"
    "fmt"
```

```
    "time"
)

func gen(ctx context.Context, nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for _, n := range nums {
            select {
            case <-ctx.Done():
                return
            case out <- n:
            }
        }
    }()
    return out
}

func mul(ctx context.Context, in <-chan int, factor int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for v := range in {
            select {
            case <-ctx.Done():
                return
            case out <- v * factor:
            }
        }
    }()
    return out
}

func filterEven(ctx context.Context, in <-chan int) <-chan int {
```

```
    out := make(chan int)
    go func() {
        defer close(out)
        for v := range in {
            if v%2 == 0 {
                select {
                case <-ctx.Done():
                    return
                case out <- v:
                }
            }
        }
    }()
    return out
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
    defer cancel()

    src := gen(ctx, 1, 2, 3, 4, 5, 6)
    stage1 := mul(ctx, src, 2)
    stage2 := filterEven(ctx, stage1)

    sum := 0
    for v := range stage2 {
        sum += v
    }
    fmt.Println("sum:", sum) // sum of doubled evens
}
```

ประเด็นสำคัญ: ต้องมี ctx หรือ mechanism เพื่อยกเลิก (cancel) goroutine เพื่อหลีกเลี่ยง goroutine leak

---

4. Error handling ใน pipeline

หลาย pipeline ต้องส่งค่าพร้อม error — pattern ที่ปลอดภัยคือส่ง struct { V T; Err error } แทนการ panic:

```
type Item[T any] struct {
    V   T
    Err error
}
```

ขั้นตอนแต่ละ stage ตรวจสอบ Err ก่อนประมวลผล และอาจหยุดหรือข้าม

---

5. Memoization / Caching (พร้อม concurrency safety)

Simple memoize (not concurrent)

```
func MemoizeIntToInt(fn func(int) int) func(int) int {
    cache := map[int]int{}
    return func(n int) int {
        if v, ok := cache[n]; ok {
            return v
        }
        v := fn(n)
        cache[n] = v
        return v
    }
}
```

Concurrent-safe memoize (generic)

```
import "sync"

func Memoize[T comparable, R any](fn func(T) R) func(T) R {
    var mu sync.RWMutex
    cache := map[T]R{}
    return func(k T) R {
        mu.RLock()
        if v, ok := cache[k]; ok {
            mu.RUnlock()
            return v
        }
        mu.RUnlock()
```

```
        v := fn(k)
        mu.Lock()
        cache[k] = v
        mu.Unlock()
        return v
    }
}
```

ข้อควรระวัง

- Cache เติบโตถ้าไม่มี eviction ⟶ memory leak
- สำหรับ production ให้ใช้ LRU, TTL หรือ lib ที่ออกแบบมาสำหรับ cache (ristretto, groupcache, bigcache ฯลฯ)

---

6. Performance / Memory considerations

- **Escape analysis**: closures ที่จับตัวแปรอาจทำให้ค่าถูกย้ายไป heap — เพิ่ม GC pressure *วิธีลด*: หลีกเลี่ยงการจับตัวแปรใหญ่ ๆ; ให้ตัวแปรเป็น local copy ภายใน loop หากต้อง capture
- **Allocation**: Map/Filter/Chain ที่สร้าง slice ใหม่บ่อย ๆ จะ allocate — ใช้ preallocated slice หรือทำ in-place เมื่อปลอดภัย
- **Inlining**: ฟังก์ชันขนาดเล็กมักจะ inline — ถ้าไม่ inline จะมี overhead call
- **Interface{} boxing**: หลีกเลี่ยงใช้ interface{} ใน hot paths — ใช้ generics แทน
- **Benchmarking**: ใช้ testing.B กับ go test -bench . และ -benchmem เพื่อตรวจ allocation

ตัวอย่าง benchmark (โครงสร้าง):

```
func BenchmarkMap(b *testing.B) {
    nums := make([]int, 1000)
    for i := 0; i < 1000; i++ { nums[i] = i }
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        _ = Map(nums, func(x int) int { return x * 2 })
    }
}
```

---

7. Testing HOFs — แนวทางปฏิบัติ

- ใช้ **table-driven tests** (input, function, expected)

- สำหรับ wrapper/Retry ให้สร้าง fake function ที่ควบคุมผลลัพธ์ (fail N ครั้งแล้ว succeed) เพื่อทดสอบ logic
- ทดสอบ concurrency: ใช้ -race (go test -race) เพื่อตรวจ race condition
- ทดสอบ pipelines: mock source/ sink และใช้ deterministic inputs

ตัวอย่าง test สำหรับ Retry:

```
func TestRetry(t *testing.T) {
    ctx := context.Background()
    attempt := 0
    fn := func() error {
        attempt++
        if attempt < 3 { return errors.New("fail") }
        return nil
    }
    r := Retry(ctx, 5, time.Millisecond*1)
    if err := r(fn); err != nil {
        t.Fatalf("expected nil, got %v", err)
    }
    if attempt != 3 { t.Fatalf("expected 3 attempts, got %d", attempt) }
}
```

---

8. Common pitfalls (และวิธีแก้)

8.1 Capturing loop variable (classic bug)

ผิด:

```
func bad() {
    funcs := make([]func() int, 3)
    for i := 0; i < 3; i++ {
        funcs[i] = func() int { return i } // captures i ⟶ ทั้งหมดคืนค่า 3
    }
}
```

ถูก:

```
for i := 0; i < 3; i++ {
    j := i
    funcs[i] = func() int { return j } // capture j ที่เป็น local copy
}
```

8.2 Goroutine closure inside loop

ผิด:

```
for _, v := range items {
    go func() {
        fmt.Println(v) // v ถูกเปลี่ยนไปเมื่อ loop ดำเนินต่อ
    }()
}
```

ถูก:

```
for _, v := range items {
    go func(v T) {
        fmt.Println(v)
    }(v)
}
```

8.3 Calling nil function ⟶ panic

ตรวจสอบก่อนเรียก: if fn != nil { fn() }

8.4 Overuse of closures ⟶ allocation

ถ้าฟังก์ชันสร้าง closure ใน tight loop อาจเกิด allocation สูง — profile ก่อน optimize

---

9. Idiomatic patterns in real projects

- **HTTP middleware** (logger, auth, recovery) — HOF returning http.Handler
- **Functional options**: สร้างตัวเลือกการตั้งค่าด้วย type Option func(*Config) (pattern ยอดนิยมใน Go)
- **Retry / Backoff / Circuit breaker** — ใช้ factory ให้คืน wrapper fn
- **Streaming ETL** — ใช้ pipeline channels + context
- **Validation pipeline** — หลาย validator ถูกต่อกัน, แต่ละ validator เป็น func(T) error

ตัวอย่าง Functional Options (สั้น):

```
type Config struct { Timeout time.Duration; Retries int }
type Option func(*Config)

func WithTimeout(d time.Duration) Option { return func(c *Config) { c.Timeout = d } }
func New(opts ...Option) *Config {
    c := &Config{Timeout: time.Second, Retries: 3}
    for _, o := range opts { o(c) }
    return c
```

}

---

10. Practical checklist — ก่อนใช้ HOF ใน production

1. Is the function *pure* or side-effectful? ถ้าผลลัพธ์ขึ้นกับ external state จัดการให้ชัดเจน
2. Tested? — มี unit tests + benchmarks
3. Memory: ตรวจ profiling (pprof) for allocations and goroutine leaks
4. Cancellation: ถ้าใช้ goroutine/stream ต้อง support context/cancel
5. Concurrency: ถ้าแชร์ cache/state ต้อง thread-safe (sync.Mutex / sync.Map)
6. Observability: add logging/metrics in wrapper functions (middleware)

ฟังก์ชันที่รับฟังก์ชันเป็น argument (Functions as Arguments)

ฟังก์ชันที่รับฟังก์ชันเป็น argument (Functions as Arguments) เป็นแนวคิดสำคัญใน **Functional Programming** ของภาษา **Golang** ที่ช่วยให้โค้ดยืดหยุ่น, นำกลับมาใช้ซ้ำได้, และลดการเขียนโค้ดซ้ำซ้อน โดยเฉพาะเมื่อเราทำงานกับข้อมูลที่ต้องการ **การประมวลผลหลายรูปแบบ** แต่ไม่อยากเขียนฟังก์ชันหลายตัวที่คล้ายกัน

□ แนวคิดพื้นฐาน

ใน Go ฟังก์ชันเป็น **first-class citizen** หมายความว่า:

- ฟังก์ชันสามารถเก็บในตัวแปรได้
- ฟังก์ชันสามารถส่งเป็น argument ไปยังอีกฟังก์ชันได้
- ฟังก์ชันสามารถคืนค่าฟังก์ชันได้

ดังนั้น เราสามารถสร้างฟังก์ชันทั่วไปที่ **“รับฟังก์ชันอื่น”** เพื่อควบคุมพฤติกรรมการทำงานภายในได้ เช่น การกรองข้อมูล, การแปลงข้อมูล, หรือการจัดการกับอัลกอริทึมที่เปลี่ยนได้

□ โครงสร้างพื้นฐาน

```
func FunctionName(arg1 Type1, f func(param ParamType) ReturnType) ReturnType {
    // ใช้ f ภายใน
}
```

เช่น:

- f คือ parameter ที่เป็น “ฟังก์ชัน”
- ฟังก์ชันนี้สามารถถูกเรียกภายในได้เหมือนฟังก์ชันปกติ

□ ตัวอย่างพื้นฐานที่ 1: ฟังก์ชัน apply

```
package main
```

```
import "fmt"

// apply รับฟังก์ชัน f และตัวเลข x จากนั้นเรียก f(x)
func apply(x int, f func(int) int) int {
    return f(x)
}

func square(n int) int {
    return n * n
}

func main() {
    result := apply(5, square) // ส่งฟังก์ชัน square เข้าไป
    fmt.Println("Square of 5 =", result)

    // ส่ง anonymous function เข้าไปก็ได้
    double := apply(7, func(n int) int {
        return n * 2
    })
    fmt.Println("Double of 7 =", double)
}
```

คำอธิบาย

- apply เป็นฟังก์ชันทั่วไป ที่รับ x int และ f func(int) int
- f สามารถเป็นฟังก์ชันชื่อปกติ (square) หรือ anonymous function
- สิ่งนี้ช่วยให้ apply เป็น **ฟังก์ชันระดับสูง (higher-order)** ที่ควบคุม “พฤติกรรม” ผ่าน argument

ผลการรัน

```
Square of 5 = 25
Double of 7 = 14
```

ตัวอย่างเชิงลึกที่ 2: ฟังก์ชัน filter

```
package main
```

```
import "fmt"

// filter: รับ slice และฟังก์ชัน predicate แล้วกรองค่า
func filter(nums []int, predicate func(int) bool) []int {
    var result []int
    for _, n := range nums {
        if predicate(n) {
            result = append(result, n)
        }
    }
    return result
}

func main() {
    numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

    // กรองเลขคู่
    evens := filter(numbers, func(n int) bool {
        return n%2 == 0
    })
    fmt.Println("Evens:", evens)

    // กรองเลขที่มากกว่า 5
    greaterThanFive := filter(numbers, func(n int) bool {
        return n > 5
    })
    fmt.Println("Greater than 5:", greaterThanFive)
}
```

คำอธิบาย

- filter ไม่รู้ล่วงหน้าว่าจะกรองแบบไหน ⟶ มอบหมายให้ predicate (ฟังก์ชัน) เป็นคนตัดสิน
- ช่วยให้โค้ด reusable มาก — ไม่ต้องสร้าง filterEven, filterOdd, filterGreaterThan หลายตัว

ผลการรัน

```
Evens: [2 4 6 8 10]
Greater than 5: [6 7 8 9 10]
```

□ **ประโยชน์ในเชิง Functional Programming**

1. □ **ลดโค้ดซ้ำซ้อน** — สามารถส่งพฤติกรรมเป็นฟังก์ชัน แทนที่จะคัดลอกโค้ด
2. □ **เพิ่มความยืดหยุ่น** — ฟังก์ชันเดียวทำงานได้หลายรูปแบบ เพียงเปลี่ยนฟังก์ชัน argument
3. ♻□ **Composability** — นำฟังก์ชันหลายตัวมาต่อกันเป็น pipeline ได้ (ซึ่งเราจะเจอในหัวข้อ “pipelines”)

□ **ตัวอย่างประยุกต์จริง: การประมวลผลข้อมูลแบบ dynamic**

```
package main

import (
    "fmt"
    "strings"
)

func process(text string, processor func(string) string) string {
    return processor(text)
}

func main() {
    input := "Go Functional Programming"

    upper := process(input, strings.ToUpper)
    fmt.Println("Upper:", upper)

    lower := process(input, strings.ToLower)
    fmt.Println("Lower:", lower)

    replaceSpaces := process(input, func(s string) string {
        return strings.ReplaceAll(s, " ", "_")
    })
    fmt.Println("Replace spaces:", replaceSpaces)
}
```

□ จุดเด่น

- ใช้ฟังก์ชันจาก standard library (strings.ToUpper) ได้เลย เพราะ type match
- สามารถใช้ anonymous function สำหรับ behavior พิเศษได้

□ สรุปสำคัญ

หัวข้อ	รายละเอียด
ฟังก์ชันเป็น Argument	ส่งพฤติกรรมเข้าไปควบคุมฟังก์ชันระดับสูง
Anonymous Function	ใช้สร้างพฤติกรรมเฉพาะได้แบบ inline
Reusability	ฟังก์ชันเดียวใช้งานซ้ำได้ในหลายกรณี
Composability	ต่อฟังก์ชันเข้าด้วยกันเพื่อสร้าง pipeline

ชุดตัวอย่าง **“ฟังก์ชันที่รับฟังก์ชันเป็น argument”** ในภาษา **Golang**
ซึ่งผมจะแบ่งออกเป็น 2 กลุ่ม ได้แก่

- □ **ตัวอย่างพื้นฐาน** จำนวน 3 โปรแกรม (Basic Examples)
- □ **ตัวอย่างแนวประยุกต์** จำนวน 3 โปรแกรม (Applied / Real-world Examples)

แต่ละโปรแกรมจะมี □ โครงสร้างโฟลเดอร์, □ คำอธิบายโค้ดแบบละเอียด, และ □ ผลการรัน

□ A. ตัวอย่างพื้นฐาน (Basic) — 3 โปรแกรม

□ โปรแกรมที่ 1: Apply Function

ฟังก์ชันที่รับฟังก์ชันเข้ามาเพื่อ “ประมวลผลตัวเลข” ที่ส่งเข้าไป

□ โครงสร้าง

```
go-functional-arg-basic-1/
└── main.go
```

□ main.go

```
package main

import "fmt"

// apply รับค่า x และฟังก์ชัน f จากนั้นเรียก f(x)
func apply(x int, f func(int) int) int {
	return f(x)
}
```

```
// square ยกกำลังสอง
func square(n int) int {
	return n * n
}

func main() {
	// ส่งฟังก์ชัน square เข้าไป
	result1 := apply(5, square)
	fmt.Println("Square of 5 =", result1)

	// ส่ง anonymous function เข้าไป
	result2 := apply(7, func(n int) int {
		return n * 2
	})
	fmt.Println("Double of 7 =", result2)
}
```

คำอธิบาย

- apply เป็น **higher-order function** ที่รับฟังก์ชัน f เป็น argument
- square คือฟังก์ชันธรรมดา
- main ส่งทั้งฟังก์ชันชื่อและ anonymous function

ผลการรัน

```
Square of 5 = 25
Double of 7 = 14
```

โปรแกรมที่ 2: Filter Slice

ใช้ฟังก์ชันเป็น argument เพื่อกำหนด “เงื่อนไขการกรอง” ของ slice

โครงสร้าง

```
go-functional-arg-basic-2/
└── main.go
```

main.go

```
package main

import "fmt"
```

```
// filter กรองค่าจาก slice ตาม predicate
func filter(nums []int, predicate func(int) bool) []int {
	var result []int
	for _, n := range nums {
		if predicate(n) {
			result = append(result, n)
		}
	}
	return result
}

func main() {
	numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

	// กรองเลขคู่
	evens := filter(numbers, func(n int) bool {
		return n%2 == 0
	})
	fmt.Println("Evens:", evens)

	// กรองเลขมากกว่า 5
	gt5 := filter(numbers, func(n int) bool {
		return n > 5
	})
	fmt.Println("Greater than 5:", gt5)
}
```

คำอธิบาย

- filter ไม่รู้ว่าต้องกรองอะไร ⟶ delegate ให้ predicate ตัดสิน
- ผู้ใช้สามารถส่งเงื่อนไขแบบ inline ได้เลย

ผลการรัน

```
Evens: [2 4 6 8 10]
Greater than 5: [6 7 8 9 10]
```

□ โปรแกรมที่ 3: Text Processor

ใช้ฟังก์ชันเป็น argument เพื่อกำหนด “วิธีการประมวลผล string”

□ โครงสร้าง

```
go-functional-arg-basic-3/
└── main.go
```

□ main.go

```
package main

import (
	"fmt"
	"strings"
)

// process รับ string และฟังก์ชัน processor เพื่อแปลงค่า
func process(text string, processor func(string) string) string {
	return processor(text)
}

func main() {
	input := "Go Functional Programming"

	upper := process(input, strings.ToUpper)
	fmt.Println("Upper:", upper)

	lower := process(input, strings.ToLower)
	fmt.Println("Lower:", lower)

	replaced := process(input, func(s string) string {
		return strings.ReplaceAll(s, " ", "_")
	})
	fmt.Println("Replace spaces:", replaced)
}
```

□ คำอธิบาย

- ใช้ฟังก์ชันจาก standard library (strings.ToUpper, ToLower) ได้ทันที

- process เป็นฟังก์ชันกลางที่ไม่ต้องรู้ logic การแปลง string

ผลการรัน

Upper: GO FUNCTIONAL PROGRAMMING

Lower: go functional programming

Replace spaces: Go_Functional_Programming

B. ตัวอย่างแนวประยุกต์ (Applied) — 3 โปรแกรม

โปรแกรมที่ 4: Sorting with Custom Comparators

ใช้ฟังก์ชันเป็น argument เพื่อ “กำหนดเงื่อนไขการเรียง” ได้เอง

โครงสร้าง

go-functional-arg-advanced-1/

└── main.go

main.go

```
package main

import (
	"fmt"
	"sort"
)

// sortBy ใช้ comparator function เพื่อกำหนดการเรียง
func sortBy(data []int, comparator func(a, b int) bool) {
	sort.Slice(data, func(i, j int) bool {
		return comparator(data[i], data[j])
	})
}

func main() {
	numbers := []int{5, 1, 8, 3, 9, 2}

	// เรียงจากน้อยไปมาก
	asc := make([]int, len(numbers))
	copy(asc, numbers)
```

```
        sortBy(asc, func(a, b int) bool {
                return a < b
        })
        fmt.Println("Ascending:", asc)

        // เรียงจากมากไปน้อย
        desc := make([]int, len(numbers))
        copy(desc, numbers)
        sortBy(desc, func(a, b int) bool {
                return a > b
        })
        fmt.Println("Descending:", desc)
}
```

คำอธิบาย

- sortBy ใช้ sort.Slice และ comparator ที่ผู้ใช้ส่งเข้ามา
- สามารถเปลี่ยนพฤติกรรมได้โดยไม่ต้องสร้างฟังก์ชันใหม่

ผลการรัน

```
Ascending: [1 2 3 5 8 9]
Descending: [9 8 5 3 2 1]
```

โปรแกรมที่ 5: Data Transformer Pipeline

ใช้ฟังก์ชัน argument เพื่อ “ประกอบขั้นตอนประมวลผลข้อมูล” ที่ยืดหยุ่น

โครงสร้าง

```
go-functional-arg-advanced-2/
└── main.go
```

main.go

```
package main

import (
        "fmt"
        "strings"
)

// transformSlice ใช้ฟังก์ชัน transformer เพื่อแปลงแต่ละ element
```

```
func transformSlice(data []string, transformer func(string) string) []string {
	var result []string
	for _, s := range data {
		result = append(result, transformer(s))
	}
	return result
}

func main() {
	data := []string{" go ", "Functional", " Programming "}

	trimmed := transformSlice(data, strings.TrimSpace)
	fmt.Println("Trimmed:", trimmed)

	upper := transformSlice(data, func(s string) string {
		return strings.ToUpper(strings.TrimSpace(s))
	})
	fmt.Println("Upper:", upper)

	withPrefix := transformSlice(data, func(s string) string {
		return ">> " + strings.TrimSpace(s)
	})
	fmt.Println("With Prefix:", withPrefix)
}
```

คำอธิบาย

- เราไม่จำเป็นต้องสร้างหลายฟังก์ชัน เช่น trimAll, upperAll, prefixAll
- ส่งฟังก์ชัน transformer เข้าไปเพื่อกำหนดพฤติกรรม

ผลการรัน

Trimmed: [go Functional Programming]

Upper: [GO FUNCTIONAL PROGRAMMING]

With Prefix: [>> go >> Functional >> Programming]

โปรแกรมที่ 6: Logger with Custom Formatters

ใช้ฟังก์ชัน argument เพื่อ “ปรับรูปแบบข้อความ log” ได้อย่างอิสระ

โครงสร้าง

```
go-functional-arg-advanced-3/
└── main.go
```

main.go

```
package main

import (
	"fmt"
	"strings"
	"time"
)

// logMessage ใช้ formatter function ในการจัดรูปแบบข้อความ
func logMessage(msg string, formatter func(string) string) {
	formatted := formatter(msg)
	fmt.Println(formatted)
}

func main() {
	// 1. Logger ปกติ
	logMessage("System started", func(s string) string {
		return s
	})

	// 2. Logger พร้อม Timestamp
	logMessage("Database connected", func(s string) string {
		return fmt.Sprintf("[%s] %s", time.Now().Format(time.RFC3339), s)
	})

	// 3. Logger แบบ Uppercase + Prefix
	logMessage("critical error", func(s string) string {
		return "ALERT: " + strings.ToUpper(s)
	})
}
```

□ คำอธิบาย

- logMessage ไม่ต้องรู้ว่าจะ log แบบไหน ⟶ delegate ให้ formatter
- ผู้ใช้กำหนดรูปแบบข้อความได้อย่างยืดหยุ่น

□ ผลการรัน (ตัวอย่าง)

System started

[2025-10-05T14:22:13+07:00] Database connected

ALERT: CRITICAL ERROR

□ สรุปภาพรวม

ประเภท	โปรแกรม	แนวคิดหลัก
พื้นฐาน 1	Apply Function	ฟังก์ชันกลางควบคุม behavior
พื้นฐาน 2	Filter Slice	ส่งเงื่อนไขการกรองแบบ dynamic
พื้นฐาน 3	Text Processor	ส่งฟังก์ชันมาตรฐาน/anonymous เพื่อแปลง string
ประยุกต์ 4	Sorting	ส่ง comparator เพื่อกำหนดการเรียง
ประยุกต์ 5	Transformer Pipeline	ประกอบขั้นตอนการประมวลผลข้อมูล
ประยุกต์ 6	Logger	ปรับรูปแบบ log ได้อิสระ

ฟังก์ชันที่ return ฟังก์ชัน (Functions that return functions)

เป็นหัวข้อสำคัญใน **Functional Programming ของภาษา Go (Golang)**
ซึ่งช่วยให้เราสามารถสร้างฟังก์ชันที่ “ปรับแต่งพฤติกรรม” ได้แบบไดนามิก, ทำงานแบบ **factory functions**, หรือสร้าง **closures** ที่เก็บสถานะไว้ภายในได้

□ แนวคิดหลัก

ในภาษา Go:

- ฟังก์ชันเป็น **First-Class Citizen** ⟶ สามารถส่งเข้า, และ **คืนค่าออกมา** ได้
- ฟังก์ชันสามารถ **คืนค่าเป็นอีกฟังก์ชัน** ได้ ⟶ เรียกว่า “higher-order function (returning function)”
- สามารถใช้ **closure** เพื่อให้ฟังก์ชันที่ return ออกมา “จดจำค่า state ภายใน” ได้

□ โครงสร้างทั่วไป

```
func outer(...) func(...) ReturnType {
    // อาจมีตัวแปรภายใน
```

```
    return func(...) ReturnType {
        // ทำงานบางอย่าง
    }
}
```

เช่น

```
func makeMultiplier(factor int) func(int) int {
    return func(n int) int {
        return n * factor
    }
}
```

☐ ตัวอย่างพื้นฐานที่ 1: ฟังก์ชัน Factory

```
package main

import "fmt"

// makeMultiplier คืนฟังก์ชันที่คูณด้วย factor ที่กำหนด
func makeMultiplier(factor int) func(int) int {
        return func(n int) int {
                return n * factor
        }
}

func main() {
        double := makeMultiplier(2) // คืนฟังก์ชันคูณ 2
        triple := makeMultiplier(3) // คืนฟังก์ชันคูณ 3

        fmt.Println("5 x 2 =", double(5))
        fmt.Println("7 x 3 =", triple(7))
}
```

☐ อธิบาย

- makeMultiplier คืนฟังก์ชันที่มีการ **“จดจำค่า factor”** ไว้
- double และ triple เป็น “ฟังก์ชันใหม่” ที่มีพฤติกรรมเฉพาะ

ผลการรัน

5 x 2 = 10

7 x 3 = 21

ตัวอย่างพื้นฐานที่ 2: Closure เก็บ State

```
package main

import "fmt"

// counter คืนฟังก์ชันที่เก็บ state ภายใน และเพิ่มค่าทุกครั้งที่ถูกเรียก
func counter() func() int {
	count := 0
	return func() int {
		count++
		return count
	}
}

func main() {
	c1 := counter()
	fmt.Println(c1()) // 1
	fmt.Println(c1()) // 2
	fmt.Println(c1()) // 3

	c2 := counter() // คนละตัวกับ c1
	fmt.Println(c2()) // 1
}
```

อธิบาย

- count เป็นตัวแปรภายในของ counter
- ฟังก์ชันที่ return ออกมาจะ “จดจำ” ตัวแปรนี้ (closure)
- แต่ละ counter() คืนฟังก์ชันที่มี state เป็นของตัวเอง

ผลการรัน

1

2