Piscine Golang - Day03

Created @October 24, 2021 11:28 PM

요약 : 이 문서는 Golang의 Closure, Panic, Recover 내용을 담고 있습니다.

서브젝트 오류 문의 : bigpel66@icloud.com

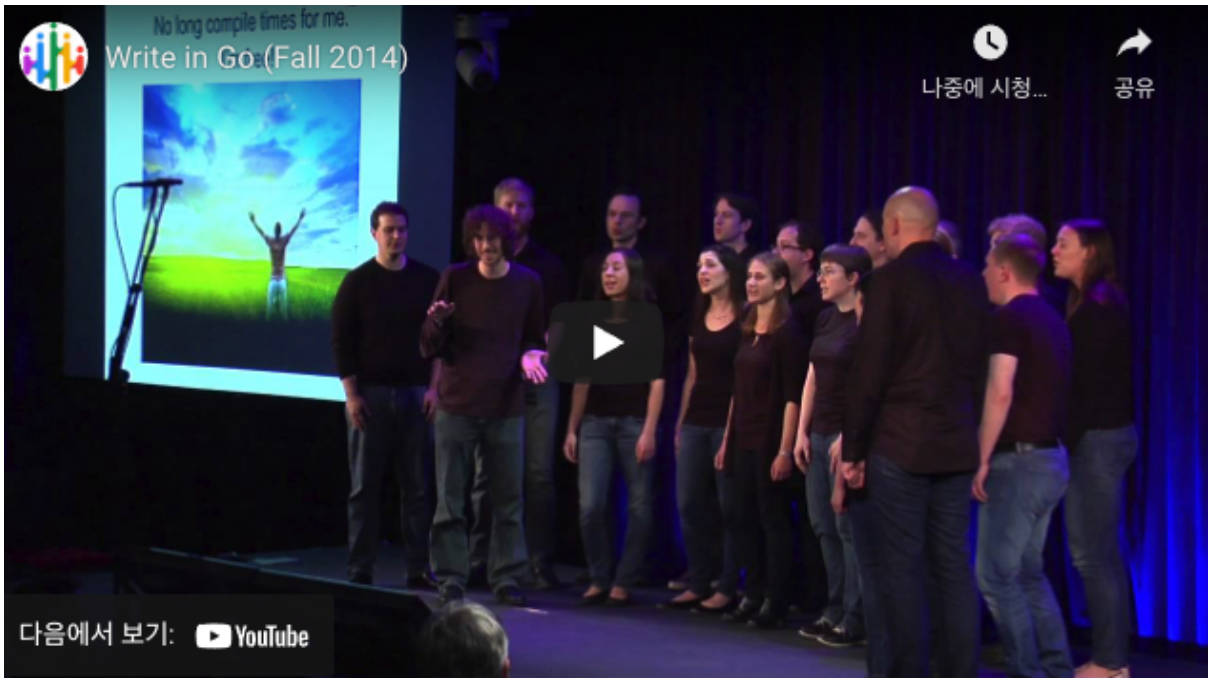
Contents

Chapter I - General Rules

- 모든 Exercise들은 Golang 1.17로 진행하며 테스트 코드와 함께 작성되어야 합니다. 작성된 테스트 코드들은 Exercise와 함께 제출되어야 합니다.
- 제출된 테스트 코드 내에 테스트 케이스는 반드시 존재해야 합니다.
- 적극적인 테스트 케이스는 권장되며, 특히 예외 케이스에 신경써야 합니다.
- 평가자는 자신이 생각했던 테스트 케이스가 존재하지 않는 경우, 피평가자에게 테스트 케이스를 요청할 수 있습니다. 이 때 피평가자의 코드가 동작하지 않으면 평가는 종료됩니다.
- 테스트 진행은 "testing" 패키지를 이용해야 하며, "testing"과 "github.com/stretchr/testify/assert" 둘 중 하나를 일관성 있게 사용해야 합니다.
- 테스트 진행 시 testing 객체를 이용한 벤치마킹 역시 적극 권장됩니다.
- Exercise의 제출은 동료 평가 및 Moulinette 채점으로 **PASS** 혹은 **FAIL**이 결정됩니다.
- Exercise들은 gofmt를 이용하여 포맷팅이 되어 있어야 합니다. 동료 평가 진행 시, 제출된 Exercise가 포맷팅이 되어 있지 않다면 **FAIL**입니다.
- Exercise들은 golangci-lint를 이용했을 때, 별다른 문제가 없다는 것이 검증되어야 합니다. golangci-lint에서 문제가 있는 부분이 발견되면 **FAIL**입니다.
- 동료 평가 진행을 위한 주석은 적극 허용됩니다. 각 함수 및 변수들에게 적용된 주석은 Golang에서 정식으로 지원하는 godoc을 이용하여 Document로 만들 수 있습니다.
- 컴파일은 **go build**를 이용하며, 이를 위해 **go mod**를 먼저 이해해야 합니다. **go mod**를 이용하지 않고 제출된 Exercise 역시 **FAIL**입니다.
- 제공된 서브젝트를 철저히 파악하여 Golang의 기초 지식을 모두 얻을 수 있도록 하십시오.
- 서브젝트에서 요구하는 내용들이 짧더라도, 이 내용들을 이해하고 예상한대로 작동되도록 시간을 쏟는 것은 굉장히 가치 있는 행동입니다. 다양한 시도를 권장합니다.
- *Go ? Ahead ! Write in Go.*

go mod를 이용하기 전에 GOPATH, GOROOT를 알아보십시오.

Chapter II - Preamble



https://www.youtube.com/watch?v=LJvEIjRBSDA

```
The schedule's tight on the cluster tonight
클러스터에서 해야 할 일로 쉴 틈 없을 오늘밤

So I parallelized my code
그래서 코드를 병렬로 작동하게 작성했어

All those threads and continuations
그 모든 스레드와 컨티뉴에이션들로

My head's going to explode
머리는 폭발하기 일보직전이야

And all that boilerplate
심지어 그 모든 형식적인 코드들

That FactoryBuilderAdapterDelegateImpl
그 모든 "괴상한디자인패턴의복잡다단한구현"

Seems unjustified
뭔가 이상한 것 같아

Give me something simple
간단하게 해결할 수 없을까
```

```
Don't write in Scheme
스킴은 쓰지마

Don't write in C
C로 짜지마

No more pointers that I forget to free()
free() 하길 깜빡한 포인터여 이제 그만 안녕

Java's verbose, Python's too slow
자바는 장황하고, 파이썬은 느려터졌단 걸

It's time you know
이제 깨달을 때가 왔어

Write in Go! Write in Go!
Go로 짜! Go로 짜!

No inheritance anymore
상속 문법이여 이제 그만 안녕

Write in Go! Write in Go!
Go로 짜! Go로 짜!

There's no do or while, just for
do도 while도 없어, 오직 for뿐

I don't care what your linters say
당신의 린터가 뭐라고 말하든 상관없어

I've got tools for that
내 툴이 그걸 해결해줄테니

The code never bothered me anyway
코드는 더 이상 날 괴롭힐 수 없어

It's funny how some features Make every change seem small
재미있는 특징들이 모든 변경점을 작아보이게 해

And the errors that once slowed me Don't get me down at all
발목잡던 에러들은 전혀 날 다운되게 하지 않아

It's time to see what Go can do
자 이제 Go가 뭘 할 수 있는지 알아볼까

'Cause is seems too good to be true
'왜냐면 믿기 어렵도록 너무 좋아보이니까

No long compile times for me.
더 이상 기나긴 컴파일 타임은 없어

I'm free!
난 자유야!

Write in Go! Write in Go!
Go로 짜! Go로 짜!

Kiss your pointer math goodbye
```

```
포인터 산술과 작별의 키스를 해

Write in Go! Write in Go!
Go로 짜! Go로 짜!

Time to give GC a try
GC가 정리하게 두자고

I don't care if my structures stay On the heap or stack
난 상관 안해 내 구조체가 힙이나 스택 영역에 남아 있어도

My program spawns its goroutines without a sound
내 프로그램은 깔끔하게 고루틴들을 생성하고

Control is spiraling through buffered channels all around
사방의 모든 버퍼드 채널을 통해 실행흐름이 제어돼

I don't remember why I ever once subclassed
내가 왜 옛날엔 서브 클래스를 생성했는지 기억도 안 나

I'm never going back, My tests all build and pass!
다신 돌아가지 않을 꺼야, 내 테스트는 빌드도되고 패스도돼!

Write in Go! Write in Go!
Go로 짜! Go로 짜!

You won't use Eclipse anyomore
넌 이클립스를 다시는 안 쓰게될 거야

Write in Go! Write in Go!
Go로 짜! Go로 짜!

Who cares what Boost is for?
부스트 라이브러리가 알 게 뭐야?

I don't care what the tech lead say
난 상관 안해 기술 책임자가 뭐라 말하든

I'll rewrite it all!
내가 전부 다시 짤꺼야!

Writing code never bothered me anyway
코드 짜기는 더 이상 날 괴롭힐 수 없으니까
```

Chapter III

Exercise 00 : Map Reduce

Aa Name	Tags
Turn-in directory	ex00
Files to turn in	mr/mr.go , mr/mr_test.go , mr/func.go , go.mod, main.go
Module name	ex00
Allowed packages	"fmt" , "strconv"

여러분은 LinkedList에 적용할 수 있는 Map 그리고 Reduce 함수를 구현하면서, 함수 타입 변수(Function Value)와 함수 리터럴(Function Literal)에 대해 공부할 것입니다.

mr.go에는 아래와 같이 LinkedList 구조체 및 메서드, 그 내부를 이루는 Node 구조체 및 메서드가 있습니다.

```
// mr/mr.go
package mr

import (
  "fmt"
)

type Any interface{}

type Node struct {
  Value Any
  Next  *Node
}

func NewNode(value Any) *Node {
  return &Node{value, nil}
}

func (n *Node) Insert(value Any) {
  tmp := n.Next
  n.Next = NewNode(value)
  if n.Next != nil {
    n.Next.Next = tmp
  }
}

func (n *Node) Remove() {
  if n.Next != nil {
    n.Value = n.Next.Value
    n.Next = n.Next.Next
```

```
  } else {
    n.Value = nil
    n.Next = nil
  }
}

type LinkedList struct {
  Head *Node
  Tail *Node
}

func NewLinkedList() *LinkedList {
  return &LinkedList{nil, nil}
}

func (l *LinkedList) Append(value Any) {
  if l.Head == nil {
    l.Head = NewNode(value)
    l.Tail = l.Head
  } else {
    l.Tail.Insert(value)
    l.Tail = l.Tail.Next
  }
}

func (l *LinkedList) Print() {
  for tmp := l.Head; tmp != nil; tmp = tmp.Next {
    fmt.Println(tmp.Value)
  }
  fmt.Println()
}

func (l *LinkedList) Map(f func(value Any) Any) (ret *LinkedList) {
  //...
}

func (l *LinkedList) Reduce(f func(value1, value2 Any) Any) (ret Any) {
  //...
}
```

아래와 같은 main.go 파일을 이용해보면, 그림과 같은 결과를 얻을 수 있습니다.

```
// main.go
package main

import "ex00/mr"

func main() {
  l := mr.NewLinkedList()
  l.Append("1")
  l.Append(2)
  l.Append("3")
  l.Append(4)
```

```
    l.Print()
    l.Head.Next.Next.Remove()
    l.Print()
}
```

Node에는 임의의 타입인 Value를 저장할 수 있지만, 이번 챕터에서는 int와 string 타입만을 Node구조체의 Value로 사용할 것입니다.

함수형 프로그래밍에는 Map과 Reduce라는 개념이 있습니다.

- Map은 Iterable 속성을 가진 데이터에 특정 함수를 적용한 동일한 크기의 데이터를 새롭게 만들어 냅니다.
- Map에서 적용할 수 있는 함수는 LinkedList의 원소 하나를 받아서 연산을 적용한 값을 리턴합니다.
- Reduce는 Iterable한 데이터에 특정 함수를 적용하여 크기가 1인 데이터를 결과로 얻어냅니다.
- Reduce에서 적용할 수 있는 함수는 LinkedList의 원소를 2개 받아서 연산을 적용한 값을 리턴합니다. 2개를 인자로 받고 1개의 값을 리턴하기 때문에 결과적으로 모든 원소를 돌면서 적용했을 때 결과의 크기가 1이 됩니다.
- Map과 Reduce에서 사용할 수 있는 함수들은 데이터가 동일하다면 데이터가 정렬된 순서에 상관없이 항상 동일한 결과를 내야합니다.

(마지막 문장은 Reduce(Sum)은 가능하지만 Reduce(Sub)는 원소의 순서가 바뀌면 결과가 달라지기 때문에 사용이 불가능하다는 말입니다.)

다음과 같은 mr/funcs.go와 main.go를 실행하였을 때, 동일한 결과를 얻을 수 있도록 mr.go에 Map과 Reduce 메서드를 작성하세요

```
// mr/funcs.go
package mr

import "strconv"

func SumInt(value1, value2 Any) Any {
  var ret1, ret2 int
  switch v1 := value1.(type) {
  case int:
    ret1 = int(v1)
  case string:
    ret1, _ = strconv.Atoi(v1)
  }
  switch v2 := value2.(type) {
  case int:
    ret2 = int(v2)
  case string:
    ret2, _ = strconv.Atoi(v2)
  }
  return ret1 + ret2
}

func AddOneInt(value Any) (ret Any) {
  switch v := value.(type) {
  case int:
    ret = v + 1
  case string:
    vi, _ := strconv.Atoi(v)
    ret = strconv.Itoa(vi + 1)
  }
  return
}

func AddOneString(value Any) (ret Any) {
  switch v := value.(type) {
  case int:
    ret, _ = strconv.Atoi(strconv.Itoa(v) + "1")
  case string:
    ret = v + "1"
  }
  return
}
```

```
// main.go
package main

import (
  "ex00/mr"
```

```
  "fmt"
)

func main() {
  l := mr.NewLinkedList()
  l.Append("1")
  l.Append(2)
  l.Append("3")
  l.Append(4)
  l.Print()

  l2 := l.Map(mr.AddOneInt)
  l2.Print()

  l3 := l.Map(mr.AddOneString)
  l3.Print()

  fmt.Println(l.Reduce(mr.SumInt))
  fmt.Println(l2.Reduce(mr.SumInt))
  fmt.Println(l3.Reduce(mr.SumInt))
}
```

Map과 Reduce 메서드는 다음과 같은 함수의 원형을 가집니다.

```
func (l *LinkedList) Map(f func(value Any) Any) (ret *LinkedList)
func (l *LinkedList) Reduce(f func(value1, value2 Any) Any) (ret Any)
```

mr/funcs.go와 mr/mr.go에 Map과 Reduce 메서드를 제외하고는 다른 내용은 수정되어서는 안됩니다!

> 💡 만약 이 문제를 c로 풀었으면 어땠을지 생각해보십시오. Map, Reduce라는 함수의 원형을 파악하기도 어려웠을 것입니다. 다음 두 글을 참고하여 Golang에서 변수의 형식을 이름-타입, 함수의 형식을 이름-인풋-아웃풋 형식을 채택한 이유를 생각해보십시오.

Clockwise/Spiral Rule

There is a technique known as the ``Clockwise/Spiral Rule'' which enables any C programmer to parse in their head any C declaration!

http://c-faq.com/decl/spiral.anderson.html

Go's Declaration Syntax

Rob Pike 7 July 2010 Newcomers to Go wonder why the declaration syntax is different from the tradition established in the C family. In this post we'll compare the two approaches and

https://go.dev/blog/declaration-syntax

Chapter IV

Copy of Exercise 01 : Closure Fibonacci!

Aa Name	Tags
Turn-in directory	ex01
Files to turn in	fibo/fibo.go , fibo/fibo_test.go , go.mod , main.go
Module name	ex01
Allowed packages	"fmt"

Golang에서는 함수 타입 변수를 이용할 수 있습니다. 이는 함수의 실행 결과 값을 변수에 담는 것과는 다르며, C 언어에서 함수 포인터를 이용한다는 개념처럼 함수 자체를 변수로 취급할 수 있습니다.

만들어야 할 프로그램은 피보나치이며 다음과 같은 코드를 실행했을 때, 주어진 그림처럼 출력 결과를 얻을 수 있도록 Fibo 함수를 정의해야 합니다. 함수의 원형은 스스로 판단하여 정의해야 하고, 클로저를 이용해야 합니다.

```
package main

import (
  "ex01/fibo"
  "fmt"
)

func main() {
  f := fibo.Fibo()
  fmt.Printf("%T\n", f())
  fmt.Println(f())
  fmt.Println(f())
  fmt.Println(f())
  fmt.Println(f())
  fmt.Println(f())
}
```

💡 x라는 함수가 있고, x 함수는 y라는 함수를 반환한다고 해보자. 반환 받은 y가 x 함수의 지역 변수에 접근하는 경우 y라는 함수를 클로저라고 부릅니다.

출력 결과는 정확히 지켜져야 하며, 테스트 파일에는 Fibonacci의 다양한 케이스를 검증하여 정상적으로 작동하는지 확인하십시오. 오버플로우가 발생할 수도 있는 점을 유의하며, 별도의 처리 없이 오버플로우된 값을 그대로 사용할 수 있어야 합니다.

Chapter V

Exercise 02 : Closure-Like Fibonacci!

Aa Name	Tags
Turn-in directory	ex02
Files to turn in	fibo/fibo.go , fibo/fibo_test.go , go.mod , main.go
Module name	ex02
Allowed packages	"fmt"

클로저라는 개념이 굉장히 생소했을 것입니다. 이것을 이해하기 위해 당신은 많은 고민을 했습니다. 보다 정확한 이해를 위해, 클로저처럼 동작하지만 클로저 없이 동작할 수 있도록 이전 문제를 다시 구현해보기로 했습니다.

함수가 생성된 시점으로 볼 때, 자신의 스코프가 아닌 다른 함수 스코프에 존재하는 지역 변수를 사용하게 되었을 때 클로저라고 했습니다. 클로저를 사용하지 않고 문제를 해결해야 합니다. 따라서 평가 시에 클로저 사용 여부가 동료에 의해 확인될 것입니다.

다음과 같은 코드를 실행했을 때, 주어진 그림처럼 출력 결과를 얻을 수 있어야 합니다.

```
package main

import (
  "ex02/fibo"
  "fmt"
)

func main() {
  f := fibo.NewFibo()
  fmt.Printf("%T\n", f)
  fmt.Printf("%T\n", f.Next(f))
  fmt.Println(f.Next(f))
  fmt.Println(f.Next(f))
  fmt.Println(f.Next(f))
  fmt.Println(f.Next(f))
  fmt.Println(f.Next(f))
  fmt.Println(f.Next(f))
}
```

클로저 없이 클로저처럼 동작하도록 만들기 위해선 별도의 구조체가 필요합니다. 따라서 위와 같은 실행 결과를 얻기 위해서 Fibo 구조체와 NewFibo라는 생성자 함수를 정의하십시오. 많은 고민을 해보십시오.

출력 결과는 정확히 지켜져야 하며, 테스트 파일에는 Fibonacci의 다양한 케이스를 검증하여 정상적으로 작동하는지 확인하십시오. 오버플로우가 발생할 수도 있는 점을 유의하며, 별도의 처리 없이 오버플로우된 값을 그대로 사용할 수 있어야 합니다.

당신이 구조체와 구조체가 담고 있는 내용을 이용하여 클로저를 구현할 수 있게 된다면, 다양한 언어에서도 이를 활용할 수 있으며 클로저를 사용할 때 보다 깊은 이해를 바탕으로 다양한 시도를 할 수 있을 것입니다. Awesome! Golang에서 변수처럼 쓰는 함수는 C 언어에서 함수 포인터를 이용하면 되니, C 언어에서도 이를 이용할 수 있을 것입니다.

Chapter VI

Exercise 03 : Let's build a Freak Calculator...

Aa Name	Tags
Turn-in directory	ex03
Files to turn in	calc/calculator.go , calc/calculator_test.go , go.mod , main.go
Module name	ex03
Allowed packages	"fmt" , "math"

당신이 만들어야 할 Calculator 구조체는 아래와 같이 선언되어야 합니다.

```
type Calculator struct {
  f func(a, b int) int
  cnt int8
}
```

- Calculator는 f라는 함수를 가집니다. f 함수는 2개의 int를 매개변수로 사용하며, int를 반환합니다.
- Calculator는 cnt라는 int8 타입의 변수를 가집니다.
- Calculator의 연산 함수를 실행할 수 있도록 Run 메서드를 정의해야 합니다.
- Calculator의 연산을 수행 횟수를 확인할 수 있도록 Count 메서드를 정의해야 합니다.
- Calculator를 생성할 수 있도록 함수 f를 인자로 받는 NewCalculator 생성자 함수를 정의해야 합니다.

NewCalculator 함수의 반환 값과 각 메서드들의 리시버가 값인지 포인터인지 명확히 확인하십시오. 뒤죽박죽 난잡한 정의가 되어선 안 됩니다. 하나의 방향으로 통일해야 하며, 왜 그 방향으로 정했는지 명확히 설명할 수 있어야 합니다.

아래의 코드가 정상적으로 동작해야 합니다.

```
package main

import (
  "ex03/calc"
  "fmt"
)

func main() {
  c := calc.NewCalculator(func(a, b int) int {
    return a + b
  })
  fmt.Println(c.Run(3, 2))
  fmt.Println(c.Count())
}
```

```
> ./ex03
5
1
>
```

NewCalculator 함수의 매개변수로 사용될 수 있는 함수들은 Add, Sub, Mul, Div, Mod 등의 기능을 하는 func (int,int) int 시그니처를 가진 모든 함수를 대상으로 하며, 함수 리터럴로 인자를 넣어 테스트하게 됩니다.

Calculator의 필드들 이용할 때 2가지 사항에 대해서 고려되어야 합니다.

- f 함수로 Div 혹은 Mod와 같은 기능을 이용할 때, 0으로 나누게 되면 그 결과 값은 MaxInt가 되어야 합니다.
- cnt 값이 오버플로우가 되었을 때는 panic을 유발해야 하며, 그 내용은 "cnt"가 되어야 합니다.

당신이 만든 Calculator는 panic으로 졸도하더라도 스스로 복구될 수 있어야 합니다. 발생될 수 있는 panic은 구분지어야 하며, cnt를 이용한 panic이 발생했을 때는 복구 루틴에서 cnt 필드를 0으로 초기화해야 합니다. panic을 구분지어야 한다는 것은 제시된 2개의 panic을 따로 처리할 수 있어야 하는 것을 의미합니다. 계산 결과와 cnt는 매우 정상적인 값을 유지해야 합니다.

💡 defer의 defer를 이용해보십시오. 좋은 패턴은 아니지만 defer와 recover의 depth를 능숙하게 이용할 수 있도록 깊은 이해를 만들어야 합니다. 무엇보다 2개의 panic을 동시에 처리할 수 있어야 합니다!

cnt 필드 외의 계산 과정에서 발생하는 오버플로우에 대해서 처리하지 마십시오. 테스트 파일을 작성하여 자신이 만든 panic과 복구 루틴을 적절히 검증하고, main 함수에서 다양한 상황을 제시해보십시오.

Chapter VII

Exercise 04 : What is Wrapping?

Aa Name	Tags
Turn-in directory	ex04
Files to turn in	print/printer.go , print/printer_test.go , go.mod , main.go
Module name	ex04
Allowed packages	"fmt" , "io" , "os"

지난 몇 챕터를 이용하여 클로저를 학습했습니다. 클로저는 상태를 유지하면서 다른 동작을 추가하는 Wrapping 형태의 구조에서 유용하게 사용될 수 있습니다. panic을 하기 전에 감싸질 함수를 defer로 호출해둔다면, 해당 함수를 감싸 놓은 클로저를 이용하여 복구 루틴을 적용할 수도 있습니다!

아래의 주어진 코드는 print/print.go 파일의 내용입니다. print 패키지에 선언된 Printer 구조체는 이름, 출력해야 할 일, 출력한 횟수를 가지고 있고, NewPrinter라는 생성자 함수를 통해 이름과 출력해야할 일을 할당할 수 있습니다. Printer 구조체는 몇가지 메서드를 갖고 있는데, Print 메서드를 통해 출력을 진행하고 SetPrint 메서드로 출력할 일도 바꿀 수 있습니다. 그리고 Printing과 Waiting 함수는 Printer 구조체의 출력해야 할 일로 사용될 수 있습니다. (함수의 원형이 같지 않습니까?)

```
package print

import (
  "fmt"
  "io"
  "os"
)

type Printer struct {
  Name    string
  doPrint func(w io.Writer)
  cnt     int
}

func NewPrinter(name string) *Printer {
  return &Printer{name, Waiting, 0}
}

func (p *Printer) Print(w io.Writer) {
  p.cnt++
  defer p.doPrint(w)
  if p.cnt == 4 {
```

```
    panic("machine broken")
  }
}

func (p *Printer) SetPrint(f func(w io.Writer)) {
  p.doPrint = f
}

func Printing(w io.Writer) {
  fmt.Fprintln(w, "Printing...")
}

func Waiting(w io.Writer) {
  fmt.Fprintln(w, "Waiting...")
}
```

프린터는 출력을 할 때 어떤 곳이든 출력할 수 있어야 합니다. 종이에 출력할 수도 있고, 네트워크를 통해서 다른곳에 출력할 수도 있고, 모니터에 연결되어 있다면 모니터 화면에 출력할 수도 있겠죠? 출력해야할 목적지가 추상화 되어있다면 이것이 가능해집니다. 우리의 프린터는 io.Writer 인터페이스를 사용해서 이를 해결했습니다. (Printing() 함수와 Waiting() 함수의 매개변수를 보세요!) 이를 통해 테스트코드 만들기도 수월해 질 겁니다.

print 패키지를 이용하는 main 함수는 아래와 같은데, 주어진 print 패키지를 그대로 이용하여 실행해보면 그림과 같은 결과를 얻는 것을 확인할 수 있습니다.

```
package main

import (
  "ex04/print"
  "os"
)

func main() {
  p := print.NewPrinter("printer1")
  p.Print(os.Stdout)
  p.Print(os.Stdout)
  p.SetPrint(print.Printing)
  p.Print(os.Stdout)
  p.Print(os.Stdout)
}
```

```
> ./ex04
Waiting...
Waiting...
Printing...
Printing...
panic: machine broken

goroutine 1 [running]:
ex04/print.(*Printer).Print(0x1400006df48, {0x104ddc200, 0x1400000e018})
```

그렇다면 주어진 main 함수를 조금 바꿔서 아래처럼 이용해보십시오. 코드를 실행했을 때, 주어진 그림과 같은 형태의 출력을 얻을 수 있어야 합니다. 당신이 만든 Printer는 조금 특이해서 Printing을 하는 기계인지 Waiting을 하는 기계인지 직접 정해줘야 하고, 4번째 출력마다 고장이 나서 panic을 유발하도록 되어 있기 때문에 그 때마다 Printer가 스스로 복구되어야 합니다. 특히 SetPrint 함수는 사람이 Printer를 직접 건드린 것으로 인식하기 때문에 Printer가 스스로 복구된 횟수를 0으로 갱신하게 됩니다. 이 점들을 유의해서 Wrapper라는 메서드를 직접 정의하십시오.

Wrapper 메서드는 클로저입니다. 클로저는 다른 스코프의 지역 변수를 이용한다는 점을 명심하여 고친 횟수와 연결 지어야 합니다.

```
package main

import (
  "ex04/print"
  "os"
)

func main() {
  p := print.NewPrinter("printer1")
  p.SetPrint(p.Wrapper(print.Waiting))
  p.Print(os.Stdout)
  p.Print(os.Stdout)
  p.SetPrint(p.Wrapper(print.Printing))
  p.Print(os.Stdout)
  p.Print(os.Stdout)
}
```

```
> ./ex04
printer1: Waiting...
printer1: Waiting...
printer1: Printing...
printer1: printer is fixed 1 times
printer1: Printing...
>
```

작성된 함수는 print 패키지 내에 존재해야 합니다. 주의할 점으로는 위 실행 결과를 만들기 위해 아래 코드처럼 Printing과 Waiting 함수를 직접 조작할 수도 있습니다. 하지만 Printing 함수와 Waiting 함수는 조작하지 않고, 직접 정의한 Wrapper 함수만 활용해야 합니다. Wrapper 함수를 이용해서 어떤 Printer가 출력하고 있는지 알려줄 수 있어야 하고, panic이 일어났을 때 복구까지 할 수 있어야 합니다.

```
func Printing(w io.Writer, name string) {
  fmt.Fprintln(w, name + ": " + "Printing...")
}

func Waiting(w io.Writer, name string) {
  fmt.Fprintln(w, name + ": " + "Waiting...")
}
```

특히 print/printer_test.go 내에서 작성된 Wrapper에 대한 테스트는 Wrapper 함수를 이용했을 때와 이용하지 않았을 때를 모두 검증하도록 작성해서 치팅 여부를 보일 수 있어야 합니다.