Validate Day03

Created @October 31, 2021 11:44 AM

Introduction

평가 진행 시 다음과 같은 사항들을 지켜주십시오.

- 상호 간 존중하는 마음으로 예의를 지키며 평가를 진행합니다.
- 평가 진행 중 발생한 기능 장애들은 평가자와 함께 문제점을 밝혀내도록 하십시오. 이 과정에서 토론과 논의가 오갈 수 있습니다.
- 동료가 이해한 것과 다르게 이해할 수도 있다는 점을 배려하십시오. 열린 마음으로 평가에 임하는 것을 권장합니다.

Guidelines

반드시 제출 시 이용되었던 Git 레포지토리를 기준으로 평가를 진행해야 합니다.

평가 진행 시 악의가 있는 별칭 등으로 결과를 조작하지는 않았는지 꼼꼼히 확인하십시오. 그리고 평가 진행 시 이용될 수 있는 스크립트에 대해서도 크로스 체크를 통해 별 문제가 없다는 것을 확인하여 당황스러운 상황을 피할 수 있도록 하십시오.

빈 레포지토리, (서브젝트에서 의도되지 않은) 기능 상에 문제가 있는 프로그램, 설명할 수 없는 코드가 포함된 치팅 등의 상황에서는 **FAIL**을 부여합니다. 특히 치팅에 대해선 경위와 사유를 듣고, 다음에도 같은 상황이 벌어지지 않도록 장려해주십시오.

General

- 제출된 레포지토리가 **gofmt**와 **golangci-lint**를 만족하는지 하십시오.

```
gofmt -l [filename]
golanci-lint run [filename]
```

- 제출된 레포지토리에 go.sum이 없거나 변조되어 모듈이 정상 작동하지 않을 수 있습니다. **go mod tidy**를 이용하여 모듈 정리를 진행 했음에도 문제가 있다면, **FAIL**입니다.
- 모든 기능들은 서브젝트에 명시된 *_test.go 파일을 이용하여 제출 이전에 검증되어 있어야 합니다. 테스트 파일이 잘 작성되어 있는지 확인하십시오. 테스트 케이스가 빈약하다면 평가자가 테스트 케이스를 요구할 수 있으며, 제시된 테스트 케이스가 정상적으로 작동하지 않는다면 **FAIL**입니다.

✅ **YES**

✅ **No**

Validation

1) ex00

- 다음과 같은 디렉토리 구조를 가졌는지 하십시오. 한 개의 파일이라도 누락되어선 안 됩니다. 피평가자는 go.sum 파일이 왜 필요한지 설명할 수 있어야 합니다.

```
ex00
├── go.mod
├── main.go
└── mr
    ├── funcs.go
    ├── mr.go
    └── mr_test.go
```

✅ **YES**

✅ **No**

- Map, Reduce 메서드 다음과 같이 함수 원형을 만족해야 합니다. int나 string을 리턴하면 안 됩니다.

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

✅ **YES** ✅ **No**

- Map, Reduce 함수외에 다른 파일이나 내용이 수정되었는지 확인하십시오.
- mr.go 파일에는 "fmt"를 제외한 다른 패키지가 포함되지 않아야 합니다.

✅ **YES** ✅ **No**

- main.go 파일을 다음과 같이 수정하고 **go build**를 하여 빌드가 잘 되어 실행파일 이름이 ex00으로 나오는지 확인하십시오.
- ex00을 실행하였을 때 결과가 동일하게 나오는지 확인하십시오.

```
package main

import (
  "ex00/mr"
  "fmt"
)

func main() {
  l := mr.NewLinkedList()
  l.Append("1")
  l.Append(2)
  l.Append("3")
  l.Append(4)
  l.Append("5")
  l.Print()

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

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

✅ **YES** ✅ **No**

- mr_test.go 파일을 **go test**로 실행 하십시오. 테스트 케이스가 부족하다고 생각되면 테스트 케이스를 더 요구할 수 있습니다.

✅ **YES** ✅ **No**

2) ex01

- 다음과 같은 디렉토리 구조를 가졌는지 하십시오. 한 개의 파일이라도 누락되어선 안 됩니다. 피평가자는 go.sum 파일이 왜 필요한지 설명할 수 있어야 합니다.

```
ex01
├── go.mod
├── main.go
└── fibo
    ├── fibo.go
    └── fibo_test.go
```

✅ **YES** ✅ **No**

- fibo() 함수가 다음과 같은 함수 원형을 가지는지 확인하십시오. int16이 아닌 int를 리턴하면 안 됩니다.

```
func Fibo() func() int16
```

✅ YES ✅ No

- fibo.go 파일에는 다른 패키지가 포함되면 안 됩니다.

✅ YES ✅ No

- main.go 파일을 다음과 같이 수정하고 **go build**를 하여 빌드가 잘 되어 실행파일 이름이 ex01로 나오는지 확인하십시오.
- ex01을 실행하였을 때 결과가 동일하게 나오는지 확인하십시오. f()를 50번 반복했을 때 결과가 12065가 나와야 합니다.

```
package main

import (
  "ex01/fibo"
  "fmt"
)

func main() {
  f := fibo.Fibo()
  for i := 0; i < 50-1; i++ {
    f()
  }
  fmt.Println(f())
}
```

```
> ./ex01
12065
>
```

✅ YES ✅ No

- main.go 파일을 다음과 같이 수정하고 빌드하여 실행하였을 때 결과가 동일하게 나오는지 확인하십시오. 여러 개의 f 함수 흐름이 따로 실행되는지도 확인하십시오.

```
package main

import (
  "ex01/fibo"
  "fmt"
)

func main() {
  f1 := fibo.Fibo()
  f2 := fibo.Fibo()
  fmt.Printf("%T\n", f1())
  fmt.Println(f1())
  fmt.Println(f1())
  fmt.Println(f2())
  fmt.Println(f1())
  f3 := fibo.Fibo()
  for i := 0; i < 25; i++ {
    f3()
  }
  fmt.Println(f2())
  fmt.Println(f2())
  fmt.Println(f3())
  fmt.Println(f2())
  fmt.Println(f1())
}
```

✅ **YES**

✅ **No**

3) ex02

- 다음과 같은 디렉토리 구조를 가졌는지 하십시오. 한 개의 파일이라도 누락되어선 안 됩니다. 피평가자는 go.sum 파일이 왜 필요한지 설명할 수 있어야 합니다.

```
ex02
├── go.mod
├── main.go
```

```
└── fibo
    ├── fibo.go
    └── fibo_test.go
```

✅ YES

✅ No

- Fibo 구조체는 int16 변수 2개와 다음 Next() 함수와 같은 function value를 가지고 있어야 합니다. 평가자는 변수 이름은 달라도 되는 점을 감안하십시오.
- NewFibo() 함수 다음과 같은 함수 원형을 가져야 합니다.

```
type Fibo struct {
  first, second int16
  Next          func(*Fibo) int16
}
func NewFibo() *Fibo
```

✅ YES

✅ No

- fibo.go 파일에는 다른 패키지가 포함되면 안 됩니다.

✅ YES

✅ No

- main.go 파일을 다음과 같이 수정하고 **go build**를 하여 빌드가 잘 되어 실행파일 이름이 ex02로 나오는지 확인하십시오.
- ex02을 실행하였을 때 결과가 동일하게 나와야 합니다. f()를 50번 반복했을 때 결과가 12065가 나오는지 확인하십시오.

```
package main

import (
  "ex02/fibo"
  "fmt"
)

func main() {
  f := fibo.NewFibo()
  for i := 0; i < 50-1; i++ {
    f.Next(f)
```

```
  }
  fmt.Println(f.Next(f))
}
```

```
> ./ex02
12065
>
```

✅ **YES** ✅ **No**

- main.go 파일을 다음과 같이 수정하고 빌드하여 실행하였을 때 결과가 동일하게 나오는지 확인하십시오. 여러 개의 f 함수 흐름이 따로 실행되어야 합니다.

```
package main

import (
  "ex02/fibo"
  "fmt"
)

func main() {
  f1 := fibo.NewFibo()
  f2 := fibo.NewFibo()
  fmt.Printf("%T\n", f1)
  fmt.Printf("%T\n", f1.Next(f1))
  fmt.Println(f1.Next(f1))
  fmt.Println(f1.Next(f1))
  fmt.Println(f2.Next(f2))
  fmt.Println(f1.Next(f1))
  f3 := fibo.NewFibo()
  for i := 0; i < 25; i++ {
    f3.Next(f3)
  }
  fmt.Println(f2.Next(f2))
  fmt.Println(f2.Next(f2))
  fmt.Println(f3.Next(f3))
  fmt.Println(f2.Next(f2))
  fmt.Println(f1.Next(f1))
}
```

✅ YES ✅ No

- fibo_test.go 파일을 **go test**로 실행 하십시오. 테스트 케이스가 부족하다고 생각되면 테스트 케이스를 더 요구할 수 있습니다.

✅ YES ✅ No

4) ex03

- 다음과 같은 디렉토리 구조를 가졌는지 하십시오. 한 개의 파일이라도 누락되어선 안 됩니다. 피평가자는 go.sum 파일이 왜 필요한지 설명할 수 있어야 합니다.

```
ex02
├── go.mod
├── main.go
└── calc
    ├── calculator.go
    └── calculator_test.go
```

✅ YES ✅ No

- Run 메서드는 다음과 같은 함수 원형을 가져야 합니다.
- Count 메서드는 다음과 같은 함수 원형을 가져야 합니다.
- NewCalculator 함수는 다음과 같은 함수 원형을 가져야 합니다.

```
func (c *Calculator) Run(a, b int) (ret int)
func (c *Calculator) Count() int8
func NewCalculator(f func(a, b int) int) *Calculator
```

✅ YES ✅ No

- calculator.go 파일에는 "math" 패키지 제외한 다른 패키지가 포함되면 안 됩니다.

✅ YES ✅ No

- calculator의 Run 메서드 안에서 panic을 일으키는지 확인하십시오. 제시된 문구와 일치해야 합니다.
- 이 패닉을 처리하기 위해서 어떤 방법을 사용했는지 답할 수 있어야 합니다.

```
panic("cnt")
```

✅ YES ✅ No

- main.go 파일을 다음과 같이 수정하고 **go build**를 하여 빌드가 잘 되어 실행파일 이름이 ex03로 나오는지 확인하십시오.
- ex03을 실행하였을 때 아래와 같은 결과가 나와야 합니다. 특히 0으로 나눴을 때 MaxInt가 나오는지도 확인하십시오.

```
package main

import (
  "ex03/calc"
  "fmt"
)

func main() {
  c := calc.NewCalculator(func(a, b int) int {
    return a / b
  })
  fmt.Println(c.Run(3, 0))
}
```

```
> ./ex03
9223372036854775807
>
```

✅ YES ✅ No

- main.go 파일을 다음과 같이 수정하고 빌드하여 실행하였을 때 결과가 동일하게 나오는지 확인하십시오. 계산 결과에 대해서는 overflow가 발생해야 합니다.

```
package main

import (
  "ex03/calc"
  "fmt"
)

func main() {
  c := calc.NewCalculator(func(a, b int) int {
    return a + b
  })
  fmt.Println(c.Run(9223372036854775807, 1))
  fmt.Println(c.Count())
}
```

```
> ./ex03
-9223372036854775808
1
>
```

✅ YES ✅ No

- main.go 파일을 go test로 실행 하십시오. 테스트 케이스가 부족하다고 생각되면 테스트 케이스를 더 요구할 수 있습니다.

```
package main

import (
  "ex03/calc"
  "fmt"
)

func main() {
```

```
  c := calc.NewCalculator(func(a, b int) int {
    return a + b
  })
  for i := 0; i < 128; i++ {
    c.Run(0, 0)
  }
  fmt.Println(c.Count())
}
```

✅ **YES** ✅ **No**

- calculator_test.go 파일을 **go test**를 통해서 실행해보세요. 테스트 케이스가 부족하다고 생각되면 테스트 케이스를 더 요구할 수 있습니다.

✅ **YES** ✅ **No**

5) ex04

- 다음과 같은 디렉토리 구조를 가졌는지 하십시오. 한 개의 파일이라도 누락되어선 안 됩니다. 피평가자는 go.sum 파일이 왜 필요한지 설명할 수 있어야 합니다.

```
ex02
├── go.mod
├── main.go
└── print
    ├── printer.go
    └── printer_test.go
```

✅ **YES** ✅ **No**

- Wrapper 메서드가 다음과 같은 함수 원형을 가지는지 확인하십시오.

```
func (p *Printer) Wrapper(f func(w io.Writer)) func(w io.Writer)
```

✅ YES ✅ No

- printer.go 파일에는 "fmt"와 "io"를 제외한 다른 패키지가 포함되면 안 됩니다.

✅ YES ✅ No

- main.go 파일을 다음과 같이 수정하고 **go build**를 하여 빌드가 잘 되어 실행파일 이름이 ex04로 나오는지 확인하십시오.
- ex04을 실행하였을 때 결과가 동일하게 나오는지 확인하십시오. Wrapper로 감싼 후 Print()를 4번 호출하여 fix되게 만들고 그후 또 Print()를 4번 호출한 후 fix cnt가 2가 출력되는지 확인하십시오.
- 다시 Wrapper를 통해서 출력할 일을 set하고 나면 fix 횟수가 초기화 되어야 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.Print(os.Stdout)
  p.Print(os.Stdout)
  p.Print(os.Stdout)
  p.Print(os.Stdout)
  p.Print(os.Stdout)
  p.Print(os.Stdout)
  p.Print(os.Stdout)
  p.SetPrint(p.Wrapper(print.Printing))
  p.Print(os.Stdout)
  p.Print(os.Stdout)
  p.Print(os.Stdout)
  p.Print(os.Stdout)
}
```

```
> ./ex04
printer1: Waiting...
printer1: Waiting...
printer1: Waiting...
printer1: printer is fixed 1 times
printer1: Waiting...
printer1: Waiting...
printer1: Waiting...
printer1: Waiting...
printer1: printer is fixed 2 times
printer1: Waiting...
printer1: Waiting...
printer1: Printing...
printer1: Printing...
printer1: printer is fixed 1 times
printer1: Printing...
printer1: Printing...
> 
```

✅ **YES** ✅ **No**

- printer_test.go 파일을 **go test**로 실행 하십시오. 테스트 케이스가 부족하다고 생각되면 테스트 케이스를 더 요구할 수 있습니다.

✅ **YES** ✅ **No**

- printer_test.go 파일에서는 Wrapper 함수를 이용하지 않았을 때와 Wrapper 함수를 이용했을 때, 총 2가지 경우를 모두 테스트 코드로 작성해야 합니다. 이를 통해 Printing과 Waiting 함수를 직접 수정하는 치팅을 구분 할 수 있어야 합니다. 아래 코드와 정확히 일치하지 않아도 됩니다. 제 기능만 한다면...

```
func TestWrapperPrinter(t *testing.T) {
  t.Run("printer panic1", func(t *testing.T) {
      ass := assert.New(t)
      defer func() {
        if r := recover(); r != nil {
          ass.Equal(r, "machine broken")
        }
      }()
      var buf bytes.Buffer
      p := NewPrinter("printer1")
      p.Print(&buf)
      p.Print(&buf)
```

```
      p.SetPrint(Printing)
      p.Print(&buf)
      p.Print(&buf)
      expected := "Waiting...\nWaiting...\nPrinting...\npanic: machine broken\n"
      ass.Equal(expected, buf.String())
    })

  t.Run("wrapper 1", func(t *testing.T) {
      ass := assert.New(t)
      var buf bytes.Buffer
      p := NewPrinter("printer1")
      p.SetPrint(p.Wrapper(Waiting))
      p.Print(&buf)
      p.Print(&buf)
      p.SetPrint(p.Wrapper(Printing))
      p.Print(&buf)
      p.Print(&buf)
      expected := `printer1: Waiting...
printer1: Waiting...
printer1: Printing...
printer1: printer is fixed 1 times
printer1: Printing...
`
      ass.Equal(expected, buf.String())
    })
}
```

✅ **YES**

✅ **No**

- printer_test.go 파일에서 "bytes" 패키지을 import하여 다음과 같은 식으로 활용했는지 확인하십시오. 만일 활용하지 않았고 평가자가 모르는 것 같으면 bytes.Buffer의 존재에 대해 알려주십시오! 이는 평가 항목이 아니지만, 동료 학습을 위한 정보 제공으로 간주합니다. 피평가자가 bytes.Buffer를 활용했고 여태까지 모든 평가 항목을 만족했으면, 까짓거 아웃스탠딩 주는 건 어떠십니까?

```
buf := bytes.Buffer{}
p := NewPrinter("printer1")
p.Print(&buf)

if buf.String() != want {
//...
}
```

Ratings

✅ OK

⭐ Outstanding project

🗂️ Empty work

💾 Invalid program

👀 Cheat

💣 Crash