> source: 《Go语言学习笔记》
> https://studygolang.gitbook.io/learn-go-with-tests/go-ji-chu/install-go

- Hello
- Refactor code
- switch statement
- integer
- for loop and benchmarks
- array and slice - 代码解析 - 为什么需要预先定义切片的长度：
    - refactor the code
    - test with empty slices
    - refactor the test code
        - 创建切片
        - 添加元素
        - 切片操作
        - 遍历切片
        - 切片的容量和长度
- struct, method and interface
    - create a struct called `Rectangle`
    - now we want to add new type struct `circle`
        - Go 语言中的解决方案
        - 测试代码
        - 总结
    - interface
        - 为什么使用接口
        - 接口的定义和实现
        - 使用接口重构测试
        - 多态性
        - 例子
    - 一个Go的小知识
        - 顶级函数
        - 匿名函数
        - 区别
    - 列表驱动测试
    - refactor the test code to make it more clear
        - 1. 明确测试用例
        - 2. 使用命名字段
        - 3. 错误信息的改进
        - 4. 使用 `t.Run` 改善测试输出
        - 5. 列表驱动的测试
        - 总结
- pointer and error
    - refactor the code
    - alias
    - add Withdraw function
    - refactor test code

- withdraw with insufficient amount
- 我还是不理解go语言中“实现了一个接口中的方法，就是实现了这个接口” 能举例详细解释一下吗
  - 接口的定义
  - 类型实现接口
  - 使用接口
  - 总结
- refactor the test code
- we find duplicate in the error information
- fmt.Sprintf

Hello

> source: https://studygolang.gitbook.io/learn-go-with-tests/go-ji-chu/hello-world#zai-ci-hui-dao-hello-world

hello.go

```
package main

import "fmt"

const englishHello = "Hello, "

func Hello(name string) string {
    return englishHello + name
}

func main() {
    fmt.Println(Hello("Bruce"))
}
```

hello_test.go

```
package main

import "testing"

func TestHello(t *testing.T) {
    got := Hello("Bruce")
    want := "Hello, Bruce"

    if got != want {
        t.Errorf("got '%q want '%q", got, want)
    }
}
```

`const` make sure that we don't write duplicate "Hello, " if we need

Refactor code

We require that when we enter empty string, we print "Hello, World"

```
package main

import "fmt"

const englishHello = "Hello, "

func Hello(name string) string {
    if name == "" {
        return englishHello + "World"
    }
    return englishHello + name
}

func main() {
    fmt.Println(Hello("Bruce"))
}


package main

import "testing"

func TestHello(t *testing.T) {
    assertCorrectHello := func(t *testing.T, got, want string) {
        t.Helper()
        if got != want {
            t.Errorf("got '%q' want '%q'", got, want)
        }
    }

    t.Run("this is testing for hello with name", func(t *testing.T) {
        got := Hello("Bruce")
        want := "Hello, Bruce"

        assertCorrectHello(t, got, want)
    })

    t.Run("this is testing for hello with empty string", func(t
*testing.T) {
        got := Hello("")
        want := "Hello, World"

        assertCorrectHello(t, got, want)
    })
```

```
}
```

`t.Helper()` used for display the line number of error if there is

We add another parameter into `func Hello` which is `language`

```
package main

import "fmt"

const englishHello = "Hello, "
const spanishHello = "Hola, "
const spanish = "Spanish"

func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }
    if language == spanish {
        return spanishHello + name
    }
    return englishHello + name
}

func main() {
    fmt.Println(Hello("Bruce", "English"))
}
package main

import "testing"

func TestHello(t *testing.T) {
    assertCorrectHello := func(t *testing.T, got, want string) {
        t.Helper()
        if got != want {
            t.Errorf("got '%q' want '%q'", got, want)
        }
    }

    //3.9 add Spanish cases
    t.Run("this is testing for Spanish hello", func(t *testing.T) {
        got := Hello("Jose", "Spanish")
        want := "Hola, Jose"
        assertCorrectHello(t, got, want)
    })

    t.Run("this is testing for hello with name", func(t *testing.T) {
        got := Hello("Bruce", "English")
        want := "Hello, Bruce"

        assertCorrectHello(t, got, want)
```

```
    })

    t.Run("this is testing for hello with empty string", func(t
*testing.T) {
        got := Hello("", "English")
        want := "Hello, World"

        assertCorrectHello(t, got, want)
    })

}
```

switch statement

using `switch` and refactor the code

```
package main

import "fmt"

const englishHello = "Hello, "
const spanishHello = "Hola, "
const frenchHello = "Bonjour, "
const spanish = "Spanish"

// func Hello(name string, language string) string {
//  if name == "" {
//      name = "World"
//  }
//  if language == spanish {
//      return spanishHello + name
//  }
//  return englishHello + name
// }

// 3.13 refactor code

func Hello(name string, language string) string {
    if name == "" {
        name = "World"
    }
    return greetingPrefix(language) + name
}
func greetingPrefix(language string) (prefix string) {
    switch language {
    case "French":
        prefix = frenchHello
    case "Spanish":
        prefix = spanishHello
```

```
    case "English":
        prefix = englishHello
    }
    return
}
func main() {
    fmt.Println(Hello("Bruce", "English"))
}
```

```
package main

import "testing"

func TestHello(t *testing.T) {
    assertCorrectHello := func(t *testing.T, got, want string) {
        t.Helper()
        if got != want {
            t.Errorf("got '%q' want '%q'", got, want)
        }
    }

    //3.9 add Spanish cases
    t.Run("this is testing for Spanish hello", func(t *testing.T) {
        got := Hello("Jose", "Spanish")
        want := "Hola, Jose"
        assertCorrectHello(t, got, want)
    })

    t.Run("this is testing for hello with name", func(t *testing.T) {
        got := Hello("Bruce", "English")
        want := "Hello, Bruce"

        assertCorrectHello(t, got, want)
    })

    t.Run("this is testing for hello with empty string", func(t
*testing.T) {
        got := Hello("", "English")
        want := "Hello, World"

        assertCorrectHello(t, got, want)
    })

    t.Run("this is testing for French hello", func(t *testing.T) {
        got := Hello("Louis", "French")
        want := "Bonjour, Louis"

        assertCorrectHello(t, got, want)
    })

}
```

在我们的函数签名中，我们使用了 命名返回值（prefix string）。

这将在你的函数中创建一个名为 prefix 的变量。

它将被分配「零」值。这取决于类型，例如 int 是 0，对于字符串它是 ""。

你只需调用 return 而不是 return prefix 即可返回所设置的值。

这将显示在 Go Doc 中，所以它使你的代码更加清晰。

如果没有其他 case 语句匹配，将会执行 default 分支。

函数名称以小写字母开头。在 Go 中，公共函数以大写字母开始，私有函数以小写字母开头。我们不希望我们算法的内部结构暴露给外部，所以我们将这个功能私有化。

integer

```
package integers

// Add takes two integers and returns the sum of them
func Add(x, y int) int {
    return x + y
}

package integers

import (
    "fmt"
    "testing"
)

func TestAdder(t *testing.T) {
    sum := Add(2, 2)
    expected := 4

    if sum != expected {
        t.Errorf("expected '%d' but got '%d'", expected, sum)
    }
}

func ExampleAdd() {
    sum := Add(1, 5)
    fmt.Println(sum)
    // Output: 6
}
```

for loop and benchmarks

run `go test -bench=.`

```
package iteration

const repeatCount = 5

func Repeat(input string) (output string) {
    for i := 0; i < repeatCount; i++ {
        output += input
    }
    return
}

package iteration

import "testing"

func TestRepeat(t *testing.T) {
    got := Repeat("a")
    want := "aaaaa"

    if got != want {
        t.Errorf("expected %q but got %q", want, got)
    }
}

func BenchmarkRepeat(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Repeat("a")
    }
}
```

以下是 Go 中一些常用的字符串操作 API，以及它们的用法和功能：

1. **len(str)**：返回字符串的长度。

```
str := "Hello, 世界"
length := len(str) // length 的值为 13
```

2. **strings.Contains(s, substr)**：判断字符串 s 是否包含子串 substr。

```
import "strings"

str := "hello world"
contains := strings.Contains(str, "world") // contains 的值为 true
```

3. **strings.HasPrefix(s, prefix)** 和 **strings.HasSuffix(s, suffix)**：判断字符串 s 是否以指定的前缀或后缀开头。

```
import "strings"

str := "hello world"
startsWith := strings.HasPrefix(str, "hello") // startsWith 的值为 true
endsWith := strings.HasSuffix(str, "world")   // endsWith 的值为 true
```

4. **strings.ToUpper(s)** 和 **strings.ToLower(s)**：将字符串 s 中的所有字母转换为大写或小写。

```
import "strings"

str := "Hello, 世界"
upper := strings.ToUpper(str) // upper 的值为 "HELLO, 世界"
lower := strings.ToLower(str) // lower 的值为 "hello, 世界"
```

5. **strings.Split(s, sep)**：根据分隔符 sep 对字符串 s 进行分割，返回分割后的字符串切片。

```
import "strings"

str := "one,two,three"
parts := strings.Split(str, ",") // parts 的值为 []string{"one", "two",
"three"}
```

6. **strings.Join(strs []string, sep string)**：使用分隔符 sep 将字符串切片 strs 中的所有字符串连接起来。

```
import "strings"

parts := []string{"one", "two", "three"}
joined := strings.Join(parts, ",") // joined 的值为 "one,two,three"
```

7. **strings.Replace(s, old, new, n)**：将字符串 s 中的前 n 个 old 子串替换为 new 子串，并返回替换后的结果。

```
import "strings"

str := "hello, hello, hello"
replaced := strings.Replace(str, "hello", "world", 2) // replaced 的值为
"world, world, hello"
```

这些是一些在 Go 中常用的字符串操作 API。你可以根据需要选择合适的 API 来处理和操作字符串。

array and slice

we build a program to generate `sum` of array or slice

```
package sum

func Sum(numbers []int) (sum int) {
    for _, number := range numbers {
        sum += number
    }
    return
}

package sum

import "testing"

func TestSum(t *testing.T) {
    t.Run("collection of size 5 array", func(t *testing.T) {
        numbers := []int{1, 2, 3, 4, 5}

        got := Sum(numbers)
        want := 15

        if got != want {
            t.Errorf("got %d want %d, given %v", got, want, numbers)
        }
    })

    // t.Run("collection of any size slice", func(t *testing.T) {
    //  numbers := []int{1, 2, 3}

    //  got := Sum(numbers)
    //  want := 6

    //  if got != want {
    //      t.Errorf("got %d want %d, given %v", got, want, numbers)
    //  }
    // })
}
```

note

if you write `for number := range numbers` in the for loop, `sum` will be 10. That because it get the `index` of array not the `element` of array.

run `go test -cover` to check the test coverage

> next stage

then we want to get sum of given slices, return type should be a `slice`

write the following code

```
func TestSumAll(t *testing.T) {

    got := SumAll([]int{1,2}, []int{0,9})
    want := []int{3, 9}

    if got != want {
        t.Errorf("got %v want %v", got, want)
    }
}

func SumAll(numbers1, numbers2 []int) (output []int) {
    return
}
```

when you run, you will face following error

```
# mymod/src/sum [mymod/src/sum.test]
./sum_test.go:34:5: invalid operation: got != want (slice can only be
compared to nil)
FAIL    mymod/src/sum [build failed]
```

that is because, we cannot compare two `slice` by using `!=`, instead, we will use `reflect.DeepEqual`

```
func TestSumAll(t *testing.T) {
    got := SumAll([]int{1, 2}, []int{3, 4})

    want := []int{3, 7}

    if !reflect.DeepEqual(got, want) {
        t.Errorf("got %v want %v", got, want)
    }

}
```

but remember, `reflect.DeepEqual` is not type safe. If you change `want` to a string, you will pass the complier as well.

refactor the code in `sum.go`

```
func SumAll(numbersToSum ...[]int) (output []int) {
    lengthOfNumbers := len(numbersToSum)
    output = make([]int, lengthOfNumbers)

    for i, number := range numbersToSum {
        output[i] = Sum(number)
    }
    return
}
```

这段 Go 语言的代码定义了一个名为 SumAll 的函数，其目的是接收一系列整数切片，对每个切片中的所有数字进行求和，然后返回一个新的切片，其中包含了每个输入切片的求和结果。

代码解析

```
func SumAll(numbersToSum ...[]int) (output []int) {
```

- func SumAll：定义了一个名为 SumAll 的函数。
- (numbersToSum ...[]int)：函数的参数 numbersToSum 是一个类型为 ...[]int 的可变参数，意味着这个函数可以接收多个 []int 类型的切片作为输入。... 表示该参数可以接受任意数量的切片。
- (output []int)：函数的返回类型是 []int，这是一个整数切片。函数名之后的这部分定义了一个名为 output 的命名返回值，它将用于存储函数的最终结果。

```
    lengthOfNumbers := len(numbersToSum)
```

- lengthOfNumbers := len(numbersToSum)：这行代码通过 len 函数获取 numbersToSum 中切片的数量，并将这个数量存储在 lengthOfNumbers 变量中。

```
    output = make([]int, lengthOfNumbers)
```

- output = make([]int, lengthOfNumbers)：使用 make 函数创建了一个长度等于 lengthOfNumbers 的整数切片，并将这个新切片赋值给了 output。make 函数在这里用于初始化切片，确保 output 有足够的空间来存储每个输入切片的求和结果。

```
    for i, number := range numbersToSum {
        output[i] = Sum(number)
    }
```

- for i, number := range numbersToSum：这是一个 for 循环，使用 range 关键字遍历 numbersToSum 中的每个切片。i 是当前切片的索引，number 是当前遍历到的切片。

- `output[i] = Sum(number)`：对每个 `number` 切片调用 `Sum` 函数（假设这个函数是定义好的，用于计算一个切片中所有整数的总和），并将结果赋值给 `output` 切片的对应位置。

```
    return
}
```

- `return`：由于在函数定义中已经命名了返回值 `output`，所以这里可以直接使用一个不带任何参数的 `return` 语句来返回 `output` 的值。

总的来说，`SumAll` 函数接收一组整数切片，分别计算每个切片中数字的总和，然后返回一个包含这些总和的新切片。

> Why we need `make` to define length of slice?

在 Go 语言中，切片确实是可变长度的数据结构，这意味着它们在运行时可以根据需要增长或缩小。当我们提到切片可以改变长度时，通常是指通过追加元素（使用 `append` 函数）来增加切片的长度，或者通过切片操作来减少它的长度。

然而，在某些情况下，如果我们事先知道切片将要存储的元素数量，预先为切片分配足够的容量会更有效率。这就是为什么我们需要 `make([]int, lengthOfNumbers)` 这行代码的原因。

为什么需要预先定义切片的长度：

1. **性能优化**：通过 `make` 函数预先分配切片的空间，可以避免在追加元素时频繁扩容的开销。扩容操作通常涉及分配新的更大的内存块，复制现有元素到新的内存块，然后释放旧的内存块，这些操作都是比较耗时的。

2. **明确的初始长度**：在某些情况下，我们可能会提前知道切片最终将包含多少元素。例如，在 `SumAll` 函数中，输出切片的长度显然等于输入的切片组的长度，因为我们对每个输入切片计算一个总和。在这种情况下，直接使用 `make` 函数来创建一个具有确切长度的切片更为直接和清晰。

3. **避免 nil 切片**：直接声明一个切片（如 `var output []int`）不会分配内存，这样的切片被初始化为 `nil`。对于 `nil` 切片，你必须在使用前通过 `append` 函数或分配内存来初始化。使用 `make` 可以直接跳过这个步骤，因为它保证了切片被初始化并且具有指定的长度。

总的来说，尽管切片是可变长度的，但在已知最终长度的情况下预先分配切片的空间可以提高效率并减少运行时的内存分配开销。这就是为什么即使在切片可以动态改变长度的语言中，`make` 这样的函数也仍然非常有用。

refactor the code

add all the elements in the slice except the first element

```
package sum

func Sum(numbers []int) (sum int) {
    for _, number := range numbers {
        sum += number
    }
```

```
    return
}

func SumTails(numbersToSum ...[]int) []int {
    var sums []int
    for _, numbers := range numbersToSum {
        tail := numbers[1:]
        sums = append(sums, Sum(tail))
    }

    return sums
}


func TestSumTails(t *testing.T) {
    got := SumTails([]int{1, 2}, []int{0, 9})
    want := []int{2, 9}

    if !reflect.DeepEqual(got, want) {
        t.Errorf("got %v want %v", got, want)
    }
}
```

test with empty slices

```
func TestSumTails(t *testing.T) {
    t.Run("test with regular slices", func(t *testing.T) {
        got := SumTails([]int{1, 2}, []int{0, 9})
        want := []int{2, 9}

        if !reflect.DeepEqual(got, want) {
            t.Errorf("got %v want %v", got, want)
        }
    })

    t.Run("test with empty slices", func(t *testing.T) {
        got := SumTails([]int{}, []int{0, 9})
        want := []int{0, 9}

        if !reflect.DeepEqual(got, want) {
            t.Errorf("got %v want %v", got, want)
        }
    })
}
```

we see the following error

```
panic: runtime error: slice bounds out of range [1:0] [recovered] panic: runtime error: slice bounds out of range [1:0]
```

we update the code by adding a `if` statement

```
func SumTails(numbersToSum ...[]int) []int {
    var sums []int
    for _, numbers := range numbersToSum {
        if len(numbers) == 0 {
            sums = append(sums, 0)
        } else {
            tail := numbers[1:]
            sums = append(sums, Sum(tail))
        }
    }

    return sums
}
```

refactor the test code

```
func TestSumTails(t *testing.T) {

    checkSum := func(t *testing.T, got, want []int) {
        if !reflect.DeepEqual(got, want) {
            t.Errorf("got %v want %v", got, want)
        }
    }
    t.Run("test with regular slices", func(t *testing.T) {
        got := SumTails([]int{1, 2}, []int{0, 9})
        want := []int{2, 9}

        checkSum(t, got, want)
    })

    t.Run("test with empty slices", func(t *testing.T) {
        got := SumTails([]int{}, []int{0, 9})
        want := []int{0, 9}
        checkSum(t, got, want)
    })
}
```

在 Go 语言中，切片是一种非常灵活和强大的数据结构，用于处理序列化的数据。以下是一些常见的切片操作：

创建切片

1. **使用 `make` 函数**：可以使用 `make` 函数创建一个具有指定类型、长度和容量的切片。例如：`s := make([]int, 10, 15)` 创建了一个长度为 10，容量为 15 的整数切片。

2. **字面量**：可以通过直接指定元素来创建切片。例如：`s := []int{1, 2, 3}` 创建了一个整数切片，包含元素 1、2、3。

添加元素

1. **使用 `append` 函数**：`append` 函数用于向切片末尾添加一个或多个元素。如果切片容量不足以容纳新元素，`append` 会自动扩展切片。例如：`s = append(s, 4)` 向切片 `s` 添加了元素 4。

切片操作

1. **切片**：可以通过指定开始和结束索引来创建一个切片的切片（子切片）。例如：`sub := s[1:3]` 创建了一个新的切片 `sub`，包含 `s` 切片的第 2 个和第 3 个元素。

2. **删除元素**：Go 没有直接删除切片元素的函数，但可以通过切片操作和 `append` 来实现。例如，要从切片 `s` 中删除第 `i` 个元素：`s = append(s[:i], s[i+1:]...)`。

遍历切片

1. **使用 `for` 循环**：可以通过 `for` 循环和索引来遍历切片中的每个元素。例如：

```
for i := 0; i < len(s); i++ {
    fmt.Println(s[i])
}
```

2. **使用 `range` 关键字**：`range` 关键字提供了一种更简洁的方式来遍历切片，它可以返回索引和该索引处的值。例如：

```
for index, value := range s {
    fmt.Println(index, value)
}
```

切片的容量和长度

- **长度**：切片的长度是切片中元素的数量。使用 `len(s)` 函数获取。
- **容量**：切片的容量是从切片的第一个元素开始到其底层数组元素末尾的元素数量。使用 `cap(s)` 函数获取。

通过这些操作，你可以灵活地处理 Go 中的切片数据。切片是 Go 语言中非常核心的数据结构之一，熟练掌握这些操作对于编写高效和简洁的 Go 代码至关重要。

struct, method and interface

let's build a program to get the `perimeter` and `area` of rectangle

```
package method
```

```
func Perimeter(width, height float64) float64 {
    return (width + height) * 2
}

func Area(width, height float64) float64 {
    return width * height
}


package method

import "testing"

func TestPerimeter(t *testing.T) {
    got := Perimeter(10.0, 10.0)
    want := 40.0

    if got != want {
        t.Errorf("got %.2f want %.2f", got, want)
    }
}

func TestArea(t *testing.T) {
    got := Area(10.0, 10.0)
    want := 100.00

    if got != want {
        t.Errorf("got %.2f want %.2f", got, want)
    }
}
```

create a struct called `Rectangle`

```
package method

type Rectangle struct {
    Width  float64
    Height float64
}

func Perimeter(rectangle Rectangle) float64 {
    return (rectangle.Height + rectangle.Width) * 2
}

func Area(rectangle Rectangle) float64 {
    return rectangle.Height * rectangle.Width
}
```

```
package method

import "testing"

func TestPerimeter(t *testing.T) {
    rectangle := Rectangle{10.0, 10.0}
    got := Perimeter(rectangle)
    want := 40.0

    if got != want {
        t.Errorf("got %.2f want %.2f", got, want)
    }
}

func TestArea(t *testing.T) {
    rectangle := Rectangle{12.0, 6.0}
    got := Area(rectangle)
    want := 72.0

    if got != want {
        t.Errorf("got %.2f want %.2f", got, want)
    }
}
```

now we want to add new type struct `circle`

有些编程语言中我们可以这样做：

```
func Area(circle Circle) float64 { ... }
func Area(rectangle Rectangle) float64 { ... }
```

but not in Go

so instead, we write

```
package method

import "math"

type Rectangle struct {
    Width  float64
    Height float64
}

type Circle struct {
    Radius float64
}
```

```
func Perimeter(rectangle Rectangle) float64 {
    return (rectangle.Height + rectangle.Width) * 2
}

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}


package method

import "testing"

func TestPerimeter(t *testing.T) {
    rectangle := Rectangle{10.0, 10.0}
    got := Perimeter(rectangle)
    want := 40.0

    if got != want {
        t.Errorf("got %.2f want %.2f", got, want)
    }
}

func TestArea(t *testing.T) {
    t.Run("test rectangles", func(t *testing.T) {
        rectangle := Rectangle{12.0, 6.0}
        got := rectangle.Area()
        want := 72.0

        if got != want {
            t.Errorf("got %.2f want %.2f", got, want)
        }
    })

    t.Run("test circles", func(t *testing.T) {
        circle := Circle{10}
        got := circle.Area()
        want := 314.1592653589793

        if got != want {
            t.Errorf("got %.2f want %.2f", got, want)
        }
    })
}
```

在一些编程语言中，你可以通过函数重载来为不同类型定义相同名称的函数。然而，在 Go 语言中，**不支持函数重载**。这意味着你不能仅仅因为参数类型不同就定义两个同名的函数。因此，在 Go 语言中处理类似情况时，我们采用的方法有所不同。

Go 语言中的解决方案

在提供的代码示例中，我们引入了两个结构体：`Rectangle` 和 `Circle`，它们代表了矩形和圆形。为了计算这些形状的面积，我们使用了 Go 语言中的**方法**（Method）。

- **Rectangle 和 Circle 结构体**：定义了矩形和圆形的基本属性。`Rectangle` 有 `Width` 和 `Height` 两个字段，而 `Circle` 有一个 `Radius` 字段。
- **Perimeter 函数**：这是一个普通的函数，计算一个矩形的周长。
- **Area 方法**：我们为 `Circle` 和 `Rectangle` 类型分别定义了 `Area` 方法。注意这里的方法定义使用了 `(c Circle)` 和 `(r Rectangle)` 这样的接收器（Receiver）语法，使得 `Area` 方法与 `Circle` 和 `Rectangle` 类型关联起来。这种方式类似于其他语言中的面向对象方法，允许我们对不同的类型执行相同名称的操作。

测试代码

- **TestPerimeter**：这个测试函数检查 `Perimeter` 函数是否能正确计算矩形的周长。
- **TestArea**：这个测试函数包含了两个子测试，分别测试 `Rectangle` 和 `Circle` 类型的 `Area` 方法。通过调用形状的 `Area` 方法，我们能够计算并验证不同形状的面积计算是否正确。

总结

在 Go 语言中，我们通过为结构体定义方法来提供特定于类型的功能，而不是依赖函数重载。这种方式提供了一种清晰且灵活的方法来处理不同类型的操作，同时保持了代码的简洁性和易于理解。通过结构体方法，我们能够在类型上下文中封装相关的操作，这是一种非常强大且广泛使用的 Go 语言特性。

interface

we want to refactor the code in `test.go` so we add an interface called `shape`

```
type Shape interface {
    Area() float64
}
```

```
func TestArea(t *testing.T) {

    checkArea := func(t *testing.T, shape Shape, want float64) {
        t.Helper()
        got := shape.Area()
        if got != want {
            t.Errorf("got %.2f want %.2f", got, want)
        }
    }
    t.Run("test rectangles", func(t *testing.T) {
        rectangle := Rectangle{12, 6}
        checkArea(t, rectangle, 72.0)
    })
```

```
    t.Run("test circles", func(t *testing.T) {
        circle := Circle{10}
        checkArea(t, circle, 314.1592653589793)
    })
}
```

在 Go 语言中，接口是一种非常强大的工具，它允许你定义一组方法签名（但不实现它们），任何实现了这些方法的类型都隐式地实现了该接口。这提供了一种方式，使得你可以编写函数或方法，它们可以接受任何具有特定行为的类型，而不需要知道这些类型的具体实现细节。这就是所谓的**多态**。

在你的例子中，`Shape` 接口被用来定义所有“形状”必须实现的行为。在这个场景里，这个行为就是 `Area()` 方法，它返回形状的面积。

为什么使用接口

1. **类型安全**：通过接口，我们可以确保只有实现了 `Area()` 方法的类型才能被 `checkArea` 函数接受。这在编译时提供了类型安全保证，如果传递了一个未实现 `Area()` 方法的类型，编译器会报错。
2. **代码解耦**：接口允许 `checkArea` 函数与具体的形状实现（如 `Rectangle` 或 `Circle`）解耦。这意味着你可以添加更多实现了 `Shape` 接口的形状类型，而不需要修改 `checkArea` 函数的代码。
3. **代码复用**：通过将 `checkArea` 函数编写为接受 `Shape` 接口类型的参数，你可以避免为每种形状编写重复的测试代码。不管是矩形、圆形还是未来可能添加的任何形状，只要它们实现了 `Shape` 接口，就可以重用相同的 `checkArea` 函数进行测试。

接口的定义和实现

接口的定义很简单，只需要列出组成该接口的方法：

```
type Shape interface {
    Area() float64
}
```

任何具有 `Area() float64` 方法的类型都隐式地实现了 `Shape` 接口。这意味着你不需要在类型定义中显式声明它实现了哪个接口。

在你的代码中，`Rectangle` 和 `Circle` 类型都有一个返回 `float64` 的 `Area()` 方法，所以它们都隐式地实现了 `Shape` 接口。

使用接口重构测试

重构后的 `TestArea` 函数创建了一个名为 `checkArea` 的辅助函数，它接受一个 `Shape` 类型的参数。由于 `Rectangle` 和 `Circle` 都实现了 `Shape` 接口，它们都可以传递给 `checkArea` 函数。这种方式使得测试更加灵活和通用，减少了重复代码，并提高了测试代码的可维护性。

总之，接口在 Go 语言中是实现多态、提高代码的灵活性和可维护性的关键。通过定义和使用接口，你可以编写出更加通用、解耦和类型安全的代码。

是的，你的理解是正确的。在 Go 语言中，接口是一种类型，它定义了一组方法签名，但不实现这些方法。任何类型，只要实现了接口中所有的方法，那么这个类型就隐式地实现了该接口。

在你的代码中，`Shape` 接口定义了一个方法 `Area()`，这个方法返回一个 `float64` 类型的值。`Rectangle` 和 `Circle` 结构体都实现了 `Area()` 方法，因此它们都隐式地实现了 `Shape` 接口。

当你定义一个 `Shape` 类型的变量时，这个变量可以存储任何实现了 `Shape` 接口的类型的实例。这意味着，这个变量可以是 `Rectangle` 的实例，也可以是 `Circle` 的实例。

多态性

当你调用接口类型变量的 `Area()` 方法时（比如 `shape.Area()`），Go 语言会根据变量实际存储的类型，调用对应类型的 `Area()` 方法实现。这种能力称为多态性。多态允许你编写更加通用和灵活的代码，因为你可以编写操作接口类型的函数或方法，而不用关心具体的实现细节。

例子

假设你有以下的代码：

```
var shape Shape

shape = Rectangle{Width: 10, Height: 5}
fmt.Println(shape.Area()) // 这将调用 Rectangle 的 Area 方法

shape = Circle{Radius: 5}
fmt.Println(shape.Area()) // 这将调用 Circle 的 Area 方法
```

在这个例子中，首先 `shape` 被赋值为一个 `Rectangle` 实例，因此 `shape.Area()` 会调用 `Rectangle` 类型的 `Area()` 方法。然后，`shape` 被重新赋值为一个 `Circle` 实例，因此 `shape.Area()` 会调用 `Circle` 类型的 `Area()` 方法。这就是接口如何实现多态性和类型抽象的一个简单示例。

通过使用接口，你可以编写更加灵活和可重用的代码，而不必关心具体的实现细节。这是 Go 语言设计中的一个非常强大和核心的概念。

一个Go的小知识

在 Go 语言中，函数可以通过两种不同的方式定义：作为**顶级函数**或者作为**匿名函数**（有时也称为闭包）。在你提供的例子中，`TestArea` 是一个顶级函数，而 `checkArea` 是一个匿名函数。让我们来讨论一下这两种方式的区别：

顶级函数

顶级函数是独立定义的，它不属于任何结构体或接口。顶级函数可以在包的任何地方被直接调用，只要调用者导入了该函数所在的包。顶级函数的定义格式如下：

```
func FunctionName(parameters) returnType {
    // 函数体
}
```

例如，`TestArea` 就是一个顶级函数。它可以在相同包内的其他函数中直接通过名称调用。

匿名函数

匿名函数，顾名思义，是没有名称的函数。它们可以在需要函数类型的变量或参数的地方定义和使用。匿名函数通常用于实现局部的、一次性的逻辑，也可以用来定义闭包。匿名函数的定义格式如下：

```
func(parameters) returnType {
    // 函数体
}
```

你可以立即调用匿名函数，或者将它赋值给一个变量。在你的例子中，`checkArea` 是一个赋值给变量的匿名函数。

区别

1. **名称和作用域**：
    - 顶级函数有一个名称，这个名称在其所在的包内是唯一的。顶级函数的作用域是整个包。
    - 匿名函数没有名称，它的作用域被限制在定义它的代码块内。
2. **使用场景**：
    - 顶级函数通常用于定义可以在多个地方重复使用的逻辑。
    - 匿名函数常用于定义一次性的逻辑，特别是作为回调函数，或者在需要将函数作为参数传递给其他函数时。匿名函数还可以捕获和使用其定义所在作用域内的变量，这使得它们在定义闭包时非常有用。
3. **生命周期**：
    - 顶级函数的生命周期与其所在的包一致。
    - 匿名函数的生命周期取决于它们被赋值给的变量或者它们的使用方式。如果匿名函数被赋值给全局变量，它们的生命周期可以很长；如果赋值给局部变量，则随着局部变量的销毁而结束。

在实际编程中，根据函数的用途和预期的作用域来选择适当的函数定义方式是很重要的。顶级函数适合于那些可能在包的多个地方使用的通用逻辑，而匿名函数适合于实现局部的、特定的逻辑处理，尤其是当这些逻辑需要访问外部作用域中的变量时。

列表驱动测试

let's refactor the code

```
func TestArea(t *testing.T) {

    areaTest := []struct {
        shape Shape
        want  float64
    }{
```

```
        {Rectangle{12, 6}, 72.0},
        {Circle{10.0}, 314.1592653589793},
    }

    for _, tt := range areaTest {
        got := tt.shape.Area()
        if got != tt.want {
            t.Errorf("got %.2f want %.2f", got, tt.want)
        }
    }
}
```

add `Triangle` struct

```
type Triangle struct {
    Base   float64
    Height float64
}

func (t Triangle) Area() float64 {
    return t.Height * t.Base * 0.5
}
```

refactor the test code to make it more clear

```
func TestArea(t *testing.T) {

    areaTest := []struct {
        name    string
        shape   Shape
        hasArea float64
    }{
        {name: "Rectangle1", shape: Rectangle{Width: 12, Height: 6},
hasArea: 70.0},
        {name: "Rectangle", shape: Rectangle{Width: 12, Height: 6},
hasArea: 72.0},
        {name: "Circle", shape: Circle{Radius: 10}, hasArea:
314.1592653589793},
        {name: "Triangle", shape: Triangle{Base: 12, Height: 6}, hasArea:
36.0},
    }

    for _, tt := range areaTest {
        // using tt.name from the case to use it as the `t.Run` test name
        t.Run(tt.name, func(t *testing.T) {
            got := tt.shape.Area()
            if got != tt.hasArea {
```

```
                t.Errorf("%#v got %.2f want %.2f", tt.shape, got,
tt.hasArea)
            }
        })

    }
}
```

in this way, the code will print following error

```
➜  method git:(main) ✗ go test
--- FAIL: TestArea (0.00s)
    --- FAIL: TestArea/Rectangle1 (0.00s)
        method_test.go:33: method.Rectangle{Width:12, Height:6} got 72.00
want 70.00
FAIL
exit status 1
FAIL    mymod/src/method        0.102s
```

we can clearly locate which `name` test case is wrong

这段代码是对之前的测试函数进行的重构，主要目的是为了提高测试的可读性和易于维护性。下面将逐步解释这个重构过程：

1. 明确测试用例

原先的测试用例直接使用了结构体实例和期望值，这样虽然简洁，但是当测试用例很多时，可读性会下降，特别是对于那些不太熟悉代码的人来说，很难快速理解测试用例代表的含义。

为了解决这个问题，重构后的代码为每个测试用例增加了 `name` 字段，用一个字符串明确指出了这个测试用例是什么，例如 `"Rectangle"`、`"Circle"` 和 `"Triangle"`。这样一来，即便测试用例很多，每个测试用例也都有了明确的标识，提高了可读性。

2. 使用命名字段

重构后的代码在创建结构体实例时使用了命名字段，比如 `Rectangle{Width: 12, Height: 6}`，这比原先的 `Rectangle{12, 6}` 更加清晰。使用命名字段的好处是，即使不熟悉 `Rectangle` 结构体定义的人也能快速理解每个值的含义。

3. 错误信息的改进

在测试失败时，重构后的代码使用了 `%#v` 格式化符，它会打印出结构体的类型和字段值，这样的错误信息比原先仅包含数值的错误信息更加详细，帮助开发者快速定位问题。

4. 使用 `t.Run` 改善测试输出

通过为每个测试用例调用 `t.Run` 并传入用例的 `name`，测试框架在报告测试结果时会包含这个名称。这样一来，即使在一个包含很多测试用例的大型测试中，一旦某个测试失败，也能够迅速知道是哪个具体的测试用例出了问题，而不需要手动检查所有测试用例。

此外，使用 `t.Run` 还允许你通过 `go test -run TestArea/Rectangle` 这样的命令单独运行某个测试用例，这在调试时非常有用。

5. 列表驱动的测试

重构后的测试代码采用了列表驱动的方式，将所有测试用例组织在一个切片中。这种方式使得添加新的测试用例变得非常简单，只需往切片中添加一个新元素即可。同时，这也让测试代码更加结构化，易于维护。

总结

这次的重构主要是为了提高测试代码的可读性和易维护性。通过为测试用例命名、使用命名字段、改善错误信息、使用 `t.Run` 和采用列表驱动的测试，代码变得更加清晰易懂。这对于保证代码质量、便于团队协作以及未来的维护工作都非常重要。

pointer and error

we define a program to represent wallet

pointer.go

```
package pointer

type Wallet struct {
    balance int
}

func (w Wallet) Deposit(amount int) {
    w.balance += amount
}

func (w Wallet) Balance() int {
    return w.balance
}
```

pointer_test.go

```
package pointer

import "testing"

func TestWallet(t *testing.T) {
    wallet := Wallet{}

    wallet.Deposit(10)
```

```
    got := wallet.Balance()
    want := 10

    if got != want {
        t.Errorf("got %d want %d", got, want)
    }

}
```

we run `go test` and find out following

```
--- FAIL: TestWallet (0.00s)
    pointer_test.go:14: got 0 want 10
```

可以看出两个 balance 的地址是不同的。因此，当我们在代码中更改 balance 的值时，我们处理的是来自测试的副本。因此，balance 在测试中没有被改变。

```
func (w *Wallet) Deposit(amount int) {
    w.balance += amount
}

func (w *Wallet) Balance() int {
    return w.balance
}
```

不同之处在于，接收者类型是 *Wallet 而不是 Wallet，你可以将其解读为「指向 wallet 的指针」。

尝试重新运行测试，它们应该可以通过了。

refactor the code

we use `Bitcoin` instead of `int`

```
type Bitcoin int

type Wallet struct {
    balance Bitcoin
}

func (w *Wallet) Deposit(amount Bitcoin) {
    w.balance += amount
}

func (w *Wallet) Balance() Bitcoin {
    return w.balance
}
```

```
func TestWallet(t *testing.T) {

    wallet := Wallet{}

    wallet.Deposit(Bitcoin(10))

    got := wallet.Balance()

    want := Bitcoin(10)

    if got != want {
        t.Errorf("got %d want %d", got, want)
    }
}
```

alias

we want to print the balance of wallet in `%s` format

so we need to create method on the `type alias`

```
type Stringer interface {
    String() string
}
```

we create alias method on this method from package `fmt`

```
func (b Bitcoin) String() string {
    return fmt.Sprintf("%d BTC", b)
}
```

then we can use `%s` in test.go

```
    if got != want {
        t.Errorf("got %s want %s", got, want)
    }
```

add Withdraw function

update on test.go

```
package pointer
```

```
import (
    "testing"
)

func TestWallet(t *testing.T) {
    t.Run("test for Deposit", func(t *testing.T) {
        wallet := Wallet{}

        wallet.Deposit(Bitcoin(10))

        got := wallet.Balance()
        want := Bitcoin(10)

        if got != want {
            t.Errorf("got %s want %s", got, want)
        }
    })

    t.Run("test for Withdraw", func(t *testing.T) {
        wallet := Wallet{}

        wallet.Deposit(Bitcoin(10))

        wallet.Withdraw(Bitcoin(10))

        got := wallet.Balance()

        want := Bitcoin(0)

        if got != want {
            t.Errorf("got %s want %s", got, want)
        }
    })

}
```

```
func (w *Wallet) Withdraw(amount Bitcoin) {
    w.balance -= amount
}
```

refactor test code

```
package pointer

import (
    "testing"
)
```

```
func TestWallet(t *testing.T) {

    assertBalance := func(t *testing.T, wallet Wallet, want Bitcoin) {
        t.Helper()
        got := wallet.Balance()

        if got != want {
            t.Errorf("got %s want %s", got, want)
        }
    }
    t.Run("test for Deposit", func(t *testing.T) {
        wallet := Wallet{}

        wallet.Deposit(Bitcoin(10))

        assertBalance(t, wallet, Bitcoin(10))
    })

    t.Run("test for Withdraw", func(t *testing.T) {
        wallet := Wallet{balance: Bitcoin(10)}

        wallet.Withdraw(Bitcoin(10))

        assertBalance(t, wallet, Bitcoin(0))
    })

}
```

withdraw with insufficient amount

test.go

```
t.Run("Withdraw with insufficient funds", func(t *testing.T) {
        startingBalance := Bitcoin(20)
        withdrawAmount := Bitcoin(100)
        wallet := Wallet{balance: startingBalance}
        err := wallet.Withdraw(withdrawAmount)

        assertBalance(t, wallet, startingBalance)

        if err == nil {
            t.Errorf("want an error but didn't")
        }
    })
```

pointer.go

```
func (w *Wallet) Withdraw(amount Bitcoin) error {
```

```
    if amount > w.balance {
        return errors.New("你他妈没这么多钱")
    }
    w.balance -= amount
    return nil
}
```

我还是不理解go语言中“实现了一个接口中的方法，就是实现了这个接口”能举例详细解释一下吗

在 Go 语言中，接口（Interface）是一种类型，它定义了一组方法签名，但不实现这些方法。当某个类型为接口中所有的方法提供了实现，我们就说这个类型实现了该接口。这种机制是隐式的，意味着你不需要像在一些其他语言中那样显式声明一个类型实现了某个接口。

接口的定义

假设我们有一个接口 `Speaker`，它定义了一个方法 `Speak()`：

```
type Speaker interface {
    Speak() string
}
```

这个 `Speaker` 接口需要一个 `Speak()` 方法，但并没有指定这个方法应该如何实现。

类型实现接口

接下来，我们定义一个 `Dog` 类型，并为它提供一个 `Speak()` 方法的实现：

```
type Dog struct{}

func (d Dog) Speak() string {
    return "Woof!"
}
```

尽管我们没有显式声明 `Dog` 类型实现了 `Speaker` 接口，但由于 `Dog` 类型提供了 `Speak()` 方法，它就隐式地满足了 `Speaker` 接口。因此，我们可以说 `Dog` 类型实现了 `Speaker` 接口。

使用接口

现在我们可以编写一个接受 `Speaker` 接口类型参数的函数，并在其中调用 `Speak()` 方法：

```
func MakeSound(s Speaker) {
    fmt.Println(s.Speak())
}
```

由于 Dog 类型实现了 Speaker 接口，我们可以创建一个 Dog 的实例并将其传递给 MakeSound 函数：

```
func main() {
    dog := Dog{}
    MakeSound(dog) // 输出: Woof!
}
```

总结

- 当一个类型（如 Dog）为接口（如 Speaker）中定义的所有方法（在这个例子中是 Speak() 方法）提供了实现，我们就说这个类型实现了该接口。
- 这种机制是隐式的，无需特殊语法来声明实现关系。
- 接口使得我们可以编写更通用和灵活的代码。在上述例子中，MakeSound 函数可以接受任何实现了 Speaker 接口的类型，不仅仅是 Dog 类型。这提高了代码的可重用性和模块化。

Go 语言通过这种隐式接口实现机制，提供了一种简洁而强大的方式来实现多态和解耦，使得代码更加灵活和可维护。

refactor the test code

```
    assertError := func(t *testing.T, got error, want string) {
        if got == nil {
            t.Fatal("want an error but didn't")
        }

        if got.Error() != want {
            t.Errorf("got %s want %s", got, want)
        }
    }

        t.Run("Withdraw with insufficient funds", func(t *testing.T) {
        startingBalance := Bitcoin(20)
        withdrawAmount := Bitcoin(100)
        wallet := Wallet{balance: startingBalance}
        err := wallet.Withdraw(withdrawAmount)

        assertBalance(t, wallet, startingBalance)

        assertError(t, err, "你他妈没这么多钱")
    })
```

我们已经介绍了 t.Fatal。如果它被调用，它将停止测试。这是因为我们不希望对返回的错误进行更多断言。如果没有这个，测试将继续进行下一步，并且因为一个空指针而引起 panic。

这样子改了代码，我们就能查出.go文件有没有报错并且包的错误信息对不对\

we find duplicate in the error information

so we declare a variable

here is the final code

.go

```go
package pointer

import (
    "errors"
    "fmt"
)

var InsufficientFundsError = errors.New("你他妈没这么多钱")

type Wallet struct {
    balance Bitcoin
}

type Bitcoin int

func (w *Wallet) Deposit(amount Bitcoin) {
    w.balance += amount
}

func (w *Wallet) Balance() Bitcoin {
    return w.balance
}

func (w *Wallet) Withdraw(amount Bitcoin) error {
    if amount > w.Balance() {
        return InsufficientFundsError
    }
    w.balance -= amount
    return nil
}

func (b Bitcoin) String() string {
    return fmt.Sprintf("%d BTC", b)
}
```

test.go

```go
package pointer

import (
    "testing"
)

func TestWallet(t *testing.T) {
```

```
    assertBalance := func(t *testing.T, wallet Wallet, want Bitcoin) {
        t.Helper()
        got := wallet.Balance()

        if got != want {
            t.Errorf("got %s want %s", got, want)
        }
    }
    t.Run("test for Deposit", func(t *testing.T) {
        wallet := Wallet{}

        wallet.Deposit(Bitcoin(10))

        assertBalance(t, wallet, Bitcoin(10))
    })

    t.Run("test for Withdraw", func(t *testing.T) {
        wallet := Wallet{balance: Bitcoin(10)}

        wallet.Withdraw(Bitcoin(10))

        assertBalance(t, wallet, Bitcoin(0))
    })

    assertError := func(t *testing.T, got error, want error) {
        if got == nil {
            t.Fatal("want an error but didn't")
        }

        if got != want {
            t.Errorf("got %s want %s", got, want)
        }
    }
    t.Run("Withdraw with insufficient funds", func(t *testing.T) {
        startingBalance := Bitcoin(20)
        withdrawAmount := Bitcoin(100)
        wallet := Wallet{balance: startingBalance}
        err := wallet.Withdraw(withdrawAmount)

        assertBalance(t, wallet, startingBalance)

        assertError(t, err, InsufficientFundsError)
    })

}
```

fmt.Sprintf

use below code in test file

```
func TestString(t *testing.T) {
    wallet := Wallet{balance: Bitcoin(20)}

    myBalance := wallet.balance

    fmt.Println(myBalance)
}
```

you will see

```
20 BTC
```

fmt.Sprintf 是 Go 语言标准库中 fmt 包提供的一个函数，它可以根据格式字符串和参数生成一个新的字符串，但不会输出到控制台，而是将格式化后的字符串作为返回值。