Web Programing

Lab 3: Introduction to Go

Content

- Methods and interface
- Concurrency
- Using Go for Web Application
- HTTP Request/Response

Next Class

- Templating and Routing

Overview

- Understand how methods and interface work on go
- Understand how go handles concurrency
- Write basic Go web server program and explain the parts involved such as http and template packages; HandleFunc, multiplexer, ListenAndServe, FileServer
- Integrate Bootstrap with Go

Methods

- Go does not have classes. However, you can define methods on types.
- A method is a function with a special receiver argument.
- The receiver appears in its own argument list between the func keyword and the method name
- the Abs method has a receiver of type Vertex named v

```
package main
import ("fmt""math")

type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(v.Abs())
}
```

Methods

- You can declare a method on non-struct types
- You cannot declare a method with a receiver whose type is defined in another package (which includes the built-in types such as int).

```
package main
import ("fmt""math")

type MyFloat float64
//can not define new methods on non-local
type

func (f MyFloat) Abs() float64 {
    if f < 0 {return float64(-f)}
    return float64(f)
}

func main() {
    f := MyFloat(-math.Sqrt2)
    fmt.Println(f.Abs())
}
```

Methods - Pointer receivers

- Methods with pointer receivers can modify the value to which the receiver points
- With a value receiver, the Scale method operates on a copy of the original Vertex value.
- The Scale method must have a pointer receiver to change the Vertex value declared in the main function
- Go interprets the statement v.Scale(10) as (&v).Scale(10) since the Scale method has a pointer receiver.

```
package main
import ("fmt""math")

type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex{3, 4}
    v.Scale(10)
    fmt.Println(v.Abs())
}
```

Methods - value / Pointer receiver?

- There are two reasons to use a pointer receiver.
  - The method can modify the value that its receiver points to.
  - To avoid copying the value on each method call.

- In general, all methods on a given type should have either value or pointer receivers, but not a mixture of both

Interface

- An interface type is defined as a set of method signatures.
- A value of interface type can hold any value that implements those methods.
- A type implements an interface by implementing its methods. There is no explicit declaration of intent, no "implements" keyword.

```
type Abser interface {
    Abs() float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}

    a = f  // a MyFloat implements Abser
    a = &v // a *Vertex implements Abser
    a = v  // a vertex does not implement abser

    fmt.Println(a.Abs())
}

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {X, Y float64}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Interface values

- interface values can be thought of as a tuple of a value and a concrete type
- A nil interface value holds neither value nor concrete type.
- The interface type that specifies zero methods is known as the empty interface

```
type I interface {M()}

type Z interface {} // empty interface

type F float64

func (f F) M() {
    fmt.Println(f)
}

func main() {
    var i I
    i = F(math.Pi)
    // (3.1415926, main.F)
    var t *F
    i = t
    //(<nil>, *main.T)
    var c I
    //nill interface
}
```

Interface values

- A type assertion provides access to an interface value's underlying concrete value.

- A type switch is a construct that permits several type assertions in series

```
switch v := i.(type) {
case T:
    // here v has type T
case S:
    // here v has type S
default:
    // no match; here v has the same type
as i
}
```

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"

    s := i.(string)
    fmt.Println(s)

    s, ok := i.(string)
    fmt.Println(s, ok)

    f, ok := i.(float64)
    fmt.Println(f, ok)

    f = i.(float64) // panic
    fmt.Println(f)
}
```

Activity

- Add a string interface that converts and array of 4 integers / string to IP Address format

  {1,2,3,4} => “1.2.3.4”

Go routine / channels / Select

- A goroutine is a lightweight thread managed by the Go runtime.
- Channels are a typed conduit through which you can send and receive values with the channel operator, <-.
- Like maps and slices, channels must be created before use
- A sender can close a channel to indicate that no more values will be sent.
- The select statement lets a goroutine wait on multiple communication operations

```go
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

Go routine / channels / Select

```go
package main

import "fmt"

func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // send sum to c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c // receive from c

    fmt.Println(x, y, x+y)
}
```

```go
package main

import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

Go for web

- Package http provides HTTP client and server implementations
- HTTP Request Multiplexer: matches the URL of incoming requests against a list of registered paths and calls the associated handler for the path whenever a match is found

```
mux := http.NewServeMux()
```

- The handler function (index) should have the following signature

```
func(w http.ResponseWriter, r *http.Request)
```

- The http.ListenAndServe function is used to start the server at specific address and port
- The http.FileServer function in the following code creates a handler that will serve files from a given directory

lab

01 starting and setting up web server

02 using template

03 passing data to template

04 serving static files

05 using bootstrap

https://gitlab.com/natget21/go_lab_3

Assignment / Activity

- Create a 2 page website (make it your own page) using bootstrap / materialui
    - Gallery
    - Tables
    - Modals
    - Sliders ….
    - Try to make it responsive