Web Programing

Lab 2 : Introduction to Go

Content

- Introduction to clickup / trello
- Getting started with git
- Setting up github
- Pointers
- Structs
- slice
- maps

Next Class

- Methods and interface
- Concurrency
- Using Go for Web Application

objective

- Getting to know project management tool
- Getting to know about version control systems
- Introduction to real world tools for project organization and management
- Know different data types in go programing language

Trello / clickup

- Clickup / trelo are examples of project management tool for monitoring and managing your projects and the team member

  - Faster collaboration
  - Easier delegation
  - Accurate project tracking
  - Seamless communication

  - Quick file-sharing
  - Time tracking
  - Quick reporting

- https://app.clickup.com

- https://trello.com

Version control / git

- Version control is a system that records changes to a file or files over time so that you can recall specific version later
- The major difference between Git and any other VCS is the way Git thinks about its data.
- Git thinks of its data more like a series of snapshots
- every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
- **Github / Gitlab / Bitbucket** are git repository hosting service with additional features to manage and collaborate on project
- Read more about git https://git-scm.com/book/en/v2

Installing git

- For debian / ubuntu
  - apt-get install git
- For mac and windows
  - Download and install the git package from https://git-scm.com/downloads
- Create account on gitlab / github

Git introduction

- features of git
  - Nearly Every Operation Is Local
  - Git Has Integrity
  - Git Generally Only Adds Data to its database
- Git has 3 states
  - Modified : files have been changed
  - Staged : marked those files to be included in the next snapshot of git
  - Committed : saved in the git database
- Git has 3 sections
  - Working directory
  - Staging area
  - Git repository / directory (.git)

Git basic commands

- git --version
- git config --global user.name "John Doe"
- git config --global user.email johndoe@example.com

- Git init
- git add *.go | git add file1.go | git add *
- git status
- git commit -m 'first commit'

Git basic commands

- Git remote -v
- git remote add origin2 https://github.com/test/testproject
- git push
- git branch -b
- git pull
- git clone https://gitlab.com/natget21/golab
- git log
- git checkout
- Git fetch

pointers

- A pointer holds the memory address of a value.
- The type *T is a pointer to a T value.
- Its zero value is nil.
- The & operator generates a pointer to its operand.
- The * operator denotes the pointer's underlying value.

```
package main
import "fmt"
func main() {
    i, j := 42, 2701

    p := &i         // point to i
    fmt.Println(*p) // read i through the pointer
    *p = 21         // set i through the pointer
    fmt.Println(i)  // see the new value of i

    p = &j         // point to j
    *p = *p / 37   // divide j through the pointer
    fmt.Println(j) // see the new value of j
}


// *p =21 is known as "dereferencing" or
"indirecting".
```

structs

- A struct is a collection of fields.
- Struct fields are accessed using a dot.
- You can list a subset of fields by using the Name: syntax.

```
type Vertex struct {
    X int
    Y int
}

v = Vertex{Y: 1} //only Y is assigned
```

```
package main
import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1, 2}
    v.X = 4              // p := &v
    fmt.Println(v)       // (*p).X = 100
}                        // ~ p.x = 100

//outputs {4,2}
```

arrays

- An array has a fixed size. And can not be resized
- [n]T is an array of n values of type T.

```go
package main

import "fmt"

func main() {
    var a [2]string
    a[0] = "Hello"
    a[1] = "World"
    //a[2] = "World"
    fmt.Println(a[0], a[1])
    fmt.Println(a)

    primes := [6]int{2, 3, 5, 7, 11, 13}
    fmt.Println(primes)
}
```

Slices

- Slices are dynamically-sized, flexible view into the elements of an array
- A slice does not store any data, it just describes a section of an underlying array.
- Changing the elements of a slice modifies the corresponding elements of its underlying array.

```
package main
import "fmt"

func main() {
    names :=
[4]string{"John","Paul","George","Ringo"}
    fmt.Println(names)

    a := names[0:2]
    b := names[1:3]
    fmt.Println(a, b)

    b[0] = "MAX"
    fmt.Println(a, b)
    fmt.Println(names)
}
```

Slices

- The default is zero for the low bound and the length of the slice for the high bound.
  - a[0:10]
  - a[:10]
  - a[0:]
  - a[:]

- The length of a slice is the number of elements it contains.
  - len(s)
- The capacity of a slice is the number of elements in the underlying array, counting from the first element in the slice.
  - cap(s)
- The zero value of a slice is nil. And nil slice has a length and capacity of 0

Slices

- Slices can be created with the built-in make function.
- this is how you create dynamically sized arrays.
- The make function allocates a zeroed array and returns a slice that refers to that array
- To specify a capacity, pass a third argument to make

```
package main
import "fmt"
func main() {
    //a := make([]int,0, 5) len=0 cap=5
    b := make([]int, 5)
    printSlice("b", b)

    c := b[:2]
    printSlice("c", c)

    d := c[2:5]
    printSlice("d", d)
}
func printSlice(s string, x []int) {
    fmt.Printf("%s len=%d cap=%d %v\n",s,
len(x), cap(x), x)
}
```

Slices

- It is common to append new elements to a slice, and so Go provides a built-in append function.
- If the backing array of s is too small to fit all the given values a bigger array will be allocated.

```
package main
import "fmt"
func main() {
    var s []int
    printSlice(s)

    s = append(s, 0)
    printSlice(s)

    s = append(s, 1, 2, 3)
    printSlice(s)
}
func printSlice(s []int) {
    fmt.Printf("len=%d cap=%d %v\n",
len(s), cap(s), s)
}
```

Range

- The range form of the for loop iterates over a slice or map.
- When ranging over a slice, two values are returned for each iteration. The first is the index, and the second is a copy of the element at that index.
- You can skip the index or value by assigning to _.(**Blank identifier**)

```
package main

import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64,
128}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }

     for _, value := range pow {
          fmt.Printf("%d\n", value)
     }

}
```

Maps

- A map maps keys to values.
- The zero value of a map is nil. A nil map has no keys, nor can keys be added.
- The make function returns a map of the given type, initialized and ready for use.
- Map literals are like struct literals, but the keys are required.

```
package main
import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m2 = map[string]Vertex{
    "Bell Labs": {40.68433, -74.39967},
    "Google":    {37.42202, -122.08408},
}

func main() {
      fmt.Println(m2)
      m1 := make(map[string]Vertex)
      m1["new"]=Vertex{37.42202, -122.08408}
      fmt.Println(m1["new"])
}
```

Mutating Maps

```
package main
import "fmt"

func main() {
    m := make(map[string]int)

    m["Answer"] = 42
    fmt.Println("The value:", m["Answer"])

    m["Answer"] = 48
    fmt.Println("The value:", m["Answer"])

    delete(m, "Answer")
    fmt.Println("The value:", m["Answer"])

    v, ok := m["Answer"]
    fmt.Println("The value:", v, "Present?", ok)
}
```

Function values

- Function values may be used as function arguments and return values.
- Go functions may be closures.
- A closure is a function value that references variables from outside its body

```
package main
import "fmt"

func compute(fn func(float64, float64) float64) float64
{
    return fn(3, 4)
}
//function used as argument and return value

func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {

    hypot := func(x, y float64) float64 {
        return math.Sqrt(x*x + y*y)
    }
    fmt.Println(compute(hypot))

    pos := adder()
    for i := 0; i < 10; i++ {
        fmt.Println(
            pos(i),
        )
    }
}
```

Assignment

- Write a function that combines two lists by alternatingly taking elements, e.g. [a,b,c], [1,2,3] → [a,1,b,2,c,3]

- Implement a fibonacci function which returns a fibonacci sequence (a fibonacci sequence is sequence where the sum of the two previous numbers is the next number in the sequence 0, 1, 1, 2, 3, 5, ...)

Lab 2

end