ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 03 Go Introduction

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

August 26, 2024

Outline

Go Language Overview

Programming in Go

Reading Assignment

- This lecture: 2,3
  - Please install VSCode and Go following the instructions on: https://docs.microsoft.com/en-us/azure/developer/go/configure-visual-studio-code
  - Clone our sample code from https://github.com/wngjia/ece573-go
- Next lecture: 2,3

Outline

Go Language Overview

Programming in Go

Go

- The Go programming language.
  - Designed at Google in 2007 to improve programming productivity in an era of multicore, networked machines and large codebases.
  - Version 1.0: March 2012
- Modernization of C for simplicity, safety, and readability.
  - Package management, garbage collection, concurrency, etc.
  - Simplified C syntax with standard tool to format code.
  - Exactly the same value semantics as C.
  - Adopt common C patterns to support array/slice and OOP.

Composition and Structural Typing

- OOP helps to handle complexities in software development by limiting the scope of the work.
- Modern OOP practices favor composition and interface-based design over deep inheritance hierarchies.
  - Avoiding use of a common base class, where changes are difficult, improves flexibility and modularity.
  - Use of interfaces encourages encapsulation and then reduces couplings between class implementations.
  - Testing becomes easier for a smaller set of classes and interfaces that depending on each other.
- Surprisingly (or not so surprisingly), many of such approaches have been widely used for system programming in C.
  - Captured by Go to provide necessary abstractions.

Comprehensibility, Memory Safety, and Performance

- Directly affect cost to develop and operate cloud software.
- Languages trade-off different between the three.
- C doesn't have much feature to learn, has the best performance, but is not quite safe for memory operations.
- C++ and Rust have the best performance with lifetime based memory management but have a steep learning curve.
- Dynamic languages like Python are too slow although they are easy to learn and have garbage collection for memory safety.
- Java achieves a good balance among the three.
- Go is somewhere near Java for the three, with less features to learn but somewhat slower.

Runtime Support

- Deploying applications on cloud benefit from a small runtime for the underlying language.
  - Need less time to download and install smaller runtimes.
  - Need less memory for the runtime in addition to what the application needs to use.
- Core C/C++ libraries are part of OS distribution and require little additional memory.
- Java and dynamic languages require to download and install a large runtime like JVM and need a lot more memory.
- Go benefits from static linking to standard C library so that it requires very little runtime support as C/C++.

Concurrency

- Concurrency makes it possible to simplify complex I/O logics and to use multiple cores.
  - A number of running threads communicate with each other via shared-memory regions and message-passing channels.
- Concurrency is not among language features for most languages designed in and before 1990's.
  - Rely on OS to provide a set of functions for accessing shared-memory regions, e.g. C/C++/Java.
  - Or not allow concurrency at all, e.g. Python and Javascript.
- Communications based on shared-memory, like locks, although intuitive apparently, are prone to misuse and error.
  - Languages like C++ and Java spend a lot of efforts to provide concurrency at higher levels through message-passing.
  - Still, this doesn't prevent developers to overlook things like locks and use them incorrectly.
- Go provides concurrency based on Communicating Sequential Processes (CSP) as part of its language features.
  - Developers are forced to give up locks and many other mechanisms and have to use message-passing channels instead.

Outline

Go Language Overview

Programming in Go

Hello World

```
// hw/hw.go
package main

import "fmt"

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

- Go uses the same entrypoint `main` as C.
  - It has to be inside `package main`
- Save the code to `hw.go` and run it via `go run hw.go`
- Language features
  - Both `//` and `/**/` work for comments
  - Use `import` instead of `#include`
  - Use `func` to define a function
  - No need to use `;`
  - `{` must be at the end of the line

Variable

```go
// swap/main.go
package main

import "fmt"

func main() {
  var a int = 1
  b := 2
  fmt.Printf("before swap: a = %d, b = %d\n", a, b)
  swap(&a, &b)
  fmt.Printf("after swap: a = %d, b = %d\n", a, b)
}
```

- A variable can be defined using `var` and then initialized.
- Or you can use `:=` to define and initialize a variable.
  - Without the need to specify a type.
  - The variable still has a type and cannot be changed.
- Usually, library names are lowercase while library functions are uppercase.

Pointer

```
// swap/swap.go
package main

func swap(pa, pb *int) {
  *pa, *pb = *pb, *pa
}
```

- Pointers `*T` are addresses to variables of type `T`
  - Allow you to change a variable outside of the current function.
  - Same as C, use `&` to take address for a variable and use `*` to refer to the variable using the pointer.
- Types can be omitted for the function parameters if they have the same type.
- Multiple variables can be assigned at the same time.

Go Module

- Since `swap` is in a different file as `main`, we cannot run this more complicated program directly.
- Use `go mod init swap` to initialize a Go module to manage multiple go files.
- Run it as `go run .`
  - You can also debug it in VSCode or other IDEs.

Array and Slice

```
// slice/slice.go
package main

import "fmt"

func main() {
  var a [10]int
  s := make([]int, 0)
  for i := 0; i < 10; i++ {
    a[i] = i
    s = append(s, i*i)
  }
  for i, val := range s {
    fmt.Printf("s[%d]=%d=%d*%d\n", i, val, a[i], a[i])
  }
}
```

- Arrays like `a`, as those in C/C++/Java, are of fixed size.
- Slices like `s` are more flexible.
  - Use `make` to create a slice with initial size.
  - Use `append` to append an element to the end.
- Use `[]` to access elements using 0-based indices.

for Loops

```
for i := 0; i < 10; i++ {
  a[i] = i
  s = append(s, i*i)
}
for i, val := range s {
  fmt.Printf("s[%d]=%d=%d*%d\n", i, val, a[i], a[i])
}
```

- The most simple `for` loops use three statements
  `for initialization; condition; postcondition`
  - Similar to C/C++/Java but no parentheses
  - You'll need to use `i++` instead of `++i`
- The range `for` loops allow to obtain both the index and the element at the same time.
- Use `break` to exit the loop.
- Use `continue` to exit the current iteration.

More for Loops

```
// a while loop
for condition {
  ...
}
// an infinite loop
for {
  ...
}
```

- There is no `while` or `do while` loop in Go. Every loop is a `for` loop.

What is a slice?

```
func assign() {
  a := []int{0, 1, 2, 3, 4}
  b := a

  b[0] = 100

  fmt.Printf("after assign: a=%v, b=%v\n", a, b)
}
```

- A slice stores the address of the first element and the number of elements.
  - A memory area is allocated from the heap to store the elements.
  - No, you don't need to call `malloc`, `free`, etc. like in C or other languages.
  - `[]` will be able to check if the index is out of bound or not.
- Assignment `=` will only copy the address and the length so now `a` and `b` refer to the same memory area.

Copy a Slice

```
func mycopy() {
  a := []int{0, 1, 2, 3, 4}

  b := make([]int, len(a))
  copy(b, a)

  b[0] = 100

  fmt.Printf("after copy: a=%v, b=%v\n", a, b)
}
```

- The copy function is able to make a copy of the slice so that you can have two slices referring to two separated memory areas.

Be Careful with Append

```
func myappend() {
  a := []int{100}

  // don't do this
  for i := 0; i < 10; i++ {
    b := a
    a = append(a, i)
    b[0]++
    fmt.Printf("append %d: a=%v, b=%v\n", i, a, b)
  }
}
```

- `append` may or may not need to reallocate the memory area used by a slice when appending a new elements.
  - This behavior is the same as the `realloc` function in C.
- `a` and `b` could sometimes use the same memory area and sometime not.
  - Once `append` is called, don't reuse a slice assigned from the original slice.

Slicing a Slice

```
func slicing() {
  a := []int{0, 1, 2, 3, 4}
  b := a[1:3]
  c := a[:len(a)-1]
  d := a[2:]

  fmt.Printf("a=%v, b=%v, c=%v, d=%v\n", a, b, c, d)
}
```

- Use `[begin:end]` to slicing a slice.
  - Half close half open (`begin` included, `end` excluded).
  - `begin = 0` if omitted, `end = len()` if omitted.
  - No negative indices like in Python.
- Slicing is essentially pointer arithmetics in C so all the slices `a`, `b`, `c`, `d` now share the same memory area.
  - What if we change `a[2]` to `100`? `b[1]`, `c[2]`, and `d[0]` will all change to `100`
  - If we `append` to `a` later, We should not use `b`, `c`, and `d` any more!

Branches

```go
// rand/rand.go
package main

import (
  "fmt"
  "math/rand"
)

func main() {
  d := rand.Float64()
  if d < 0.4 {
    fmt.Println("Win!")
  } else if d > 0.6 {
    fmt.Println("Lose!")
  } else {
    fmt.Println("Tie!")
  }
}
```

- Similar to C/C++/Java but no parentheses.
  - Recall that `{` must be at the end of the line
  - If there is an `else`, then `}` must be on the same line as well.

Map

```
// map/map.go
package main

import (
  "fmt"
)

func main() {
  months := make(map[string]int)
  months["Jan"] = 1
  months["Feb"] = 2
  fmt.Printf("Jan is month %d.\n", months["Jan"])
  ...
```

- `map[K]V` allows to search for a value using a key.
  - A hash table as in most other languages.
  - `K` is the key type, don't use `float32`/`float64`.
  - `V` is the value type, can be anything.
- Use `[]` to insert key/value pairs and search for values.

Map Membership Testing

```
fmt.Printf("Input a name: ")
var name string
fmt.Scanf("%s", &name)
index, ok := months[name]
if !ok {
  fmt.Printf("Unknown month %v.\n", name)
} else {
  fmt.Printf("%v is month %d.\n", name, index)
}
```

- When searching for values, `[]` returns an extra result optionally.
  - The first one is the value, if the key exists.
  - The second one indicates if the key exists or not.

Summary

- Why Go?
  - A modern language created for cloud computing.
- Tutorials can be found at `https://go.dev/doc/tutorial/`
- Use the Go Playground `https://go.dev/play/`