Abridged Tour of Go

Wayne Chang

Jun 1st, 2018

Table of Contents

These slides have coverage of:

- Language Overview & Examples
- Data Types
- Control Flow
- Goroutines and Concurrency
- Standard Libraries
- References and Resources

glhf!

Introduction

Go is a statically-typed compiled language similar to C, with memory safety, straightforward data types, a strong standard library, concurrency primitives.

Statically-Typed

- Types are checked at compile time, as opposed to runtime.

Memory Safety

- The language tracks and manages list lengths, data sizing, and r/w access for you.

Concurrency Primitives

- Go's concurrency handling is heavily influenced by communicating sequential processes (CSP), which is based on message passing via channels.

How to Go

First, install go by following https://golang.org/doc/install.

```
$ go version
go version go1.10.2 linux/amd64
$ cd hello
$ ls
main.go
$ go build
$ ls
hello main.go
$ ./hello
Hello, World!
```

Hello, World!

```go
// Similar to namespaces in C#
package main

// The `fmt` library is now available for use
import "fmt"

// Standard function signature for entry
func main() {
    fmt.Println("Hello, world!")
}
```

add()

```
package main

import "fmt"

// Takes int x and y, returns sum
func add(x, y int) int {
    return x + y
}

func main() {
    x, y := 2, 5
    /* Same as
     *     var x int = 2
     *     var y int = 5
     */
    fmt.Println(x, "plus", y, "is", add(x, y))
}
```

Primitive Data Types

- boolean (`bool`)
- string
- numeric
  - uint8/uint16/uint32/uint64
  - int8/int16/int32/int64
  - float32/float64
  - complex64/complex128
  - byte (uint8)
  - rune (int32)
  - uint (32 or 64 bits)
  - int (same size as uint)

Simple Structs

```
type Person struct {
    Name string
    Age int
}

func (p Person) Introduce() string {
    return p.Name + " is " + p.Age  // e.g., "Fred is 51"
}

func (p *Person) IncrementAge() {
    p.Age++
}

func main() {
    p := Person{Name: "Fred", Age: 51}
    fmt.Println(p.Introduce())
}
```

Interfaces

```
type Person struct { Name string }

func (p Person) Introduce() string {
    return "I am " + p.Name
}

type Being interface {
    Introduce() string
}

func hello(b Being) {
    fmt.Println(b.Introduce())
}

// ...later
    hello(Person{"John"})
```

Slice

```go
// Effectively the same
var s []int = make([]int, 0)
sP := []int{}

fmt.Println(s, sP)  // [] []

s = append(s, 1, 2, 3, 4, 5)
fmt.Println(s)  // [1, 2, 3, 4, 5]
fmt.Println(s[1:])  // [2, 3, 4, 5]
fmt.Println(s[:3])  // [1, 2, 3]
```

Slice Internals

```
+-----------+---------+---------+
+ ptr *Elem | len int | cap int |
+-----------+---------+---------+
```

- ptr: Pointer to the actual data
- len: How many elements are active
- cap: Size of ptr's referred memory allocation size

Maps

```
// Effectively the same
var m map[string]int = make(map[string]int, 0)
mP := map[string]int{}

fmt.Println(m, mP)  // map[] map[]

m["abe"] = 30
m["sally"] = 32
fmt.Println(m)  // map[abe:30 sally:32]
fmt.Println(m["abe"])  // 30
age, ok := m["jeff"]
fmt.Println(age, ok)  // 0 false
```

If Statements

```
a, b := 1, 2
if a < b {
    fmt.Println("a is less than b")
} else if b < a {
    fmt.Println("b is less than a")
} else {
    fmt.Println("a and b are equal")
}
```

If Statements w/Declaration

```
func transfer(balance, amount int) (int, error) {
    if balance < amount {
        return 0, fmt.Errorf("Insufficient funds.")
    }
    return balance - amount, nil
}
// ...later
    if nb, err := transfer(b, amt); err != nil {
        // handle error
    }
```

Compare to C:

```
    int code = do_thing();
    if (code != 0) {
        handle_error(code);
    }
```

Basic Loops

```
for {
    fmt.Println("Breaking Infinite Loop")
    break
}

for {
    fmt.Println("Infinite Loop")
}

// Count to 10
for i := 1; i <= 10; i++ {
    fmt.Println(i)
}
```

Looping Over Slices

```
a := []string{"alpha", "beta", "charlie", "delta"}
for idx, v := range a {
    fmt.Println(idx, v)
}
/* Output:
0: alpha
1: beta
2: charlie
3: delta
*/
```

Looping Over Maps

```
m := map[string]string{"a": "alpha", "b": "beta",
                       "c": "charlie", "d": "delta"}
for k, v := range m {
    fmt.Println(k, v)
}
/* Output (order NOT guaranteed):
a: alpha
b: beta
c: charlie
d: delta
*/
```

Channels

Channels are pipes that send data one way.

```
// create a channel with buffer of 2
var c chan string = make(chan string, 2)
c <- "hey"
c <- "how are you?"
fmt.Println(<-c)  // "hey"
fmt.Println(<-c)  // "how are you?"
fmt.Println(<-c)  // block forever
```

Multiplexing Channels

`select` can be used to wait for new values across many channels.

```
for {
    select {
        case msg := <-ch1:
            fmt.Println("On ch1:", msg)
        case msg := <-ch2:
            fmt.Println("On ch2:", msg)
    }
}
```

The `time` library provides special time-related channels:

```
<-time.After(5*time.Second)
fmt.Println("5 seconds have passed")
```

Goroutines

Goroutines can be used as greenthreads to run code concurrently.

```
func count(i *int) {
    // production code should have better locking
    for {
        <-time.After(1*time.Second)
        *i++
    }
}

func main() {
    i := 0
    go count(&i)
    for {
        <-time.After(5*time.Second)
        fmt.Println("count is", i)
    }
}
```

STL: net/http

```
package main
import "net/http"

func EchoURI(w http.ResponseWriter, r *http.Request) {
    w.Write(r.URL.RequestURI())
}

func main() {
    http.HandleFunc("/", EchoURI)
    http.ListenAndServe(":8000", nil)
}
```

From a shell:

```
$ go run&
$ curl http://localhost:8000/?test=example
/?test=example
```

STL: encoding/json

```
// {"username": "admin", "password": "hunter2"}
type LoginRequest struct {
    Username string `json:"username"`
    Password string `json:"password"`
}

func Login(w http.ResponseWriter, r *http.Request) {
    decoder := json.NewDecoder(r.Body)
    payload := LoginRequest{}
    err := decoder.Decode(&payload)
    if err != nil {
        handleError(err)
        return
    }
    DoLogin(payload.Username, payload.Password)
}
```

STL: time formats

```
ts := time.Now().Format(
    "Mon Jan 2 15:04:05 -0700 MST 2006")
fmt.Println("The datetime is", ts)
// "The datetime is Fri Jun 1 17:45:04 -0400 EDT 2018"

ts = time.Now().Format("Monday January 2, 3:04 PM MST")
fmt.Println("The datetime is", ts)
// "The datetime is Friday June 1, 5:45 PM EDT"

// parsing
const longForm = "Jan 2, 2006 at 3:04pm (MST)"
t, _ := time.Parse(longForm,
    "Feb 3, 2013 at 7:54pm (PST)")
```

STL: time arithmetic

Knows about leap years, leap seconds, month days, etc.

```
func (t Time) Add(d Duration) Time
func (t Time) AddDate(years, months, days int) Time
func (t Time) Sub(u Time) Duration

now := time.Now()
tomorrow := now.Add(24 * time.Hour)
tomorrow.Sub(now)  // 24 hours as Duration
inTwoMonths := now.AddDate(0, 2, 0)
```

STL: io

Powerful abstractions to prevent useless copying.

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
type Writer interface {
        Write(p []byte) (n int, err error)
}
type Seeker interface {
        Seek(offset int64, whence int) (int64, error)
}
type ReadWriteSeeker interface {
        Reader
        Writer
        Seeker
}
```

References

- Go Spec https://golang.org/ref/spec#Types
- Tour of Go https://tour.golang.org/welcome/1
- The Go Programming Language, Donovan, Kernighan https://www.gopl.io/
- Effective Go https://golang.org/doc/effective_go.html
- Go By Example https://gobyexample.com

EOF