Overview of Go

DD1396
Ric Glassey
glassey@kth.se

Concepts

Investigate the basic features of Go

Motivation for Go

Basic tool set

Tour of language

Concurrency model

Motivation and Features

Topic

“No **major systems language** has appeared in over a decade”

- Computers faster; software engineering no faster
- Dependency management is important
- Rebellion against cumbersome type systems
- Popular systems languages lack garbage collection and parallel computation
- Emergence of multicore has completely changed computer architecture and programming models

Goals of Go

It must work at **scale**

- inspired by large scale server work at Google

It must be **modern**, but **familiar**

- built-in garbage collection
- built-in concurrency
- roughly C-like

It must be **clear**

- Syntax, semantics, dependencies

Key features

Built-in Concurrency and concurrency management

Statically typed language with type inference

Fast compilation

Remote package management

Garbage collection

Composition over inheritance

Inevitable Hello World

```go
package main

import "fmt"

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

Observations: package

```
package main
```

- No classes or objects
- Packages are the basic application building block
- Programs are run from the **`main`** package
- Cannot have multiple mains in a package

Observations: import

```
import “fmt”
```

- Access to system and 3rd party packages
- Contains types, functions, etc
- Circular dependencies are not permitted
- Packages are identified by a string name
- Standard libraries live at the project root folder
- Less classpath misery *java.blah.more.what.something.whereami.*;*

Observations: functions

func main( ) { ...

- Function declaration
- Same format for any function
- Regular functions
- Anonymous functions
- Methods of structs/types
- compare to *public static void main (String[] args) { ...*

Observations: main

func main( ) { …

No return type mentioned

No return value used

Command line arguments require **os** package

- No *main(String[] args) { …*
- **http://golang.org/pkg/os**

```go
package main

import (
    // parentheses for multiple imports
    "fmt"
    "os"
)

func main() {
    fmt.Println(os.Args)
}
```

Observations: Braces, not spaces

Use of braces / brackets should be the same as most of C-style languages

There are very few semi-colons visible

Automatic insertion during compile time

Must be a new line after an opening brace {

- Go on...try to break the rules :-)

Observations: Println

```
fmt.Println("Hello world")
```

- Capital letters mean “exported” from package
  - Think “public access” modifier
- Lowercase indicate “unexported”
  - Think “private access” modifier
  - Cannot be directly accessed

Go tool chain

For this short course, all you require:

```
$ go run
$ go fmt
$ go test
```

Other tools to check out:

```
$ go help [command]
$ go build
$ go get
```

Demo: Go toolchain

BlueGo?

Sadly it does not exist

- Any programming environment will do:
- Text editor (**Atom** and **VS Code** have Go plugin support
- Go command line tools
- IDE (**Intellij** and **Eclipse**) have Go plugin support
- **GoLand** as a dedicated environment from JetBrains

Tour of Go Syntax

Types and Variables

```
func main() {
    // `var` declares 1 or more variables.
    var a string = "initial"

    // You can declare multiple variables at once.
    var b, c int = 1, 2

    // Go will infer the type of initialized variables.
    var d = true

    // Variables declared without initialization and zero-valued
    var e int

    // The `:=` syntax is shorthand for declaring and initializing
    f := "hello"
}
```

Control Structures (if/else)

```
func main() {
    if 7 % 2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    // A statement can precede conditionals
    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

Iteration (for)

```
func main() {
    // The most basic type, with a single condition.
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    // A classic initial/condition/after `for` loop.
    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    // `for` without a condition will loop repeatedly
    for {
        fmt.Println("loop")
    }
}
```

Data Structures

Arrays

```
func main() {
    // By default an array is zero-valued, fixed length
    var a [5]int

    a[4] = 100
    fmt.Println(a[4])
    fmt.Println(len(a))

    b := [5]int{1, 2, 3, 4, 5}

    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
}
```

Slices

```
func main() {
    // Unlike arrays, slices can grow
    s := make([]string, 3)

    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println(len(s))

    // Append returns a new slice value, hence assignment
    s = append(s, "d")
    s = append(s, "e", "f")

    // slice expressions [low:high]
    l := s[2:5]
    l = s[:5]
    l = s[2:]
}
```

Maps

```
func main() {
    // To create an empty map, use the builtin `make`
    m := make(map[string]int)

    m["k1"] = 7
    m["k2"] = 13
    fmt.Println("map:", m)

    v1 := m["k1"]

    // The builtin `len` returns the number of key/value pairs
    fmt.Println("len:", len(m))

    delete(m, "k2")

    // The optional second return value indicates key presence
    _, prs := m["k2"]

    n := map[string]int{"foo": 1, "bar": 2}
}
```

Functions

Declaring Function

```
func plus(a int, b int) int {
    // Go requires explicit returns, i.e. it won't
    // automatically return the value of the last
    // expression.
    return a + b
}

func plusPlus(a, b, c int) int {
  // Omit the type name for the like-typed parameters
    return a + b + c
}

func main() {
    // Call a function just as you'd expect
    res := plus(1, 2)
    res = plusPlus(1, 2, 3)
}
```

Return values

```
// The `(int, int)` in this function signature shows that
// the function returns 2 `int`s.
func vals() (int, int) {
    return 3, 7
}

func main() {
    // Here we use the 2 different return values from the
    // call with multiple assignment.
    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    // If you only want a subset of the returned values,
    // use the blank identifier `_`.
    _, c := vals()
    fmt.Println(c)
}
```

Variadic parameters

```go
// We can express an unknown number of parameters of the same type
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {
    sum(1, 2)
    sum(1, 2, 3)
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

Pointers, Structs & Methods

Pointers

Pointers reference a location in memory where a value is stored

Consider this example with no pointers, only passing by value

What happens to x?

```
func zero(x int) {
    x = 0
}

func main() {
    x := 5
    zero(x)
    fmt.Println(x)
}
```

Pointers

***type** is used to declare a pointer

- For example ***int**

&name is used to find the address of a variable

What happens now?

```
func zeroval(ival int) {
    ival = 0
}

func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    // The &i syntax gives the memory address of i
    zeroptr(&i)
    fmt.Println("zeroptr:", i)
}
```

Structs

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}
```

```go
func main() {
    // Create a new struct
    bob := person{"Bob", 20}

    // Name the fields when initializing a struct
    alice := person{name: "Alice", age: 30}

    // Omitted fields will be zero-valued
    fred := person{name: "Fred"}

    // An `&` prefix yields a pointer to the struct
    fred_ptr := &fred

    // Access struct fields with a dot.
    sara := person{name: "Sara", age: 50}
    fmt.Println(sara.name)

    // Structs are mutable
    sara.age = 51
    fmt.Println(sara.age)
}
```

Pass by value, pass by pointer/reference

```
func upperCaseByValue(p person) {
    // creates a copy of the struct and calls it p
    p.name = strings.ToUpper(p.name)
}

func upperCaseByPointer(p *person) {
    // creates a reference to the struct (alice) and calls it p
    p.name = strings.ToUpper(p.name)
}

func main() {
    alice := person{"Alice", 20}

    upperCaseByValue(alice)
    fmt.Println(alice) // prints {Alice 20}

    upperCaseByPointer(&alice)
    fmt.Println(alice) // prints {ALICE 20}
}
```

Methods for structs

```
type rectangle struct {
    width, height int
}

// This area method has a receiver type of *rect
func (r *rectangle) area() int {
    return r.width * r.height
}

func main() {
    r := rectangle{width: 10, height: 5}
    fmt.Println("area of rectangle: ", r.area())
}
```

Concurrency in Go

Goroutines

A function that is capable of running concurrently with other functions

Use the **go** keyword followed by a function

```
package main

import "fmt"

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
    }
}

func main() {
    // create goroutine with f()
    go f(0)
    var input string
    fmt.Scanln(&input)
}
```

Demo: Goroutines

```go
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

Creating 10 concurrent threads

Demo: Non-determinism

Communication by channel

```
package main

import "fmt"

func main() {

    // Create a shared channel
    messages := make(chan string)

    // Send a value into a channel
    go func() { messages <- "ping" } ()

    // Receives a value over the channel
    msg := <-messages
    fmt.Println(msg)
 }
```

Concept Review

Concept review

Go makes concurrency part of the language syntax

Goroutines and channels

Language is not too far removed from Java/Python

Opinionated design choices

Recommended Reading

Go go-to-guide

- https://yourbasic.org/golang

Go website - http://golang.org