The Go Programming Language

Jinuk Kim (rein@ncsoft.com)

Server Platform Team, Studio 7, NCSOFT

2010. 05. 26

Why go?

- **Simple**, concise syntax
- **Reduced type** system
- **Fast** code and fast build for the new era of system programming
- Safe type systems (**static**-typing) / memory system (**GC**)
- Concurrency: Using sets of *lightweight communicating processes*

Hello, world

```go
package main
import "fmt"
func main() {
    fmt.Println("Hello, world")
}
```

Types

- int, float, ... : *machine friendly* types
- int8, uint16, uint32, ... : explicitly sized
- string: *immutable* string of byte(=uint8) sequence
- map[key] value: dictionary
- [size] *type*: array
- arrays can be sliced using array[low : high] without copy

Variable Declarations

- **var** b:int = 2 *// explicit type declarations*
- **var** a = 1 *// integer type => int, int8, int16, uint32, ...*
- c := “3” *// implicit declarations, type take from expression*
- **var** p *string = &c *// pointer to string*
- **var** a [16] int *// array*
- months := map[int] string { 1:”Jan”, 2:”Feb”, ... } *// dictionary*

Functions

- **func** add(a, b int) int { **return** a + b }
- **func** getPairs(index int) (int, int) { ... *// can return multiple values*
- **type** Op **func** (int, int) int *// type for a function which takes 2 integer arguments and returns integer value*
- All arguments are **passed by "value" except** *slices, maps, channels*
  - *Watch out for copy-construction overhead; Use slice*

Control Structures

- **if** x > 0 { ... } *// Mandatory braces*
- **for** i := 0; i < N; i++ { ... } *// C/C++ for*
- **for** i < N { ... } *// while(i < N)*
- **for** { ... } *// for(; ;)*
- **for** _, value := **range** map[string] int { init list... } { ... } *// Pythonic*
- Go also has **continue**, **break**, **goto**, ...

Control Structures

- Enhanced switch statement

```
switch { // replacement for if-else-if ...
  case ‘0’ <= c && c <= ‘9’: return c - ‘0’
  case ‘a’ <= c && c <= ‘f’: return c - ‘a’ + 10
} // no need for ‘break’. no automatic fall-through
switch c { // matches comma separated list
  case ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’: return c - ‘A’ + 10
}
```

Example: array-reversing

- Array reversing functions : call by *value* vs. call by *reference*

```
func reverse(a [10] int) {
    for i := 0; i < 5; i++ { a[i], a[9-i] = a[9-i], a[i] }
}
func reverse2(a[ ] int) {
    l := len(a) // len( ) returns size of slice or array
    for i := 0; i < l/2; i++ { a[i], a[l-i-1] = a[l-i-1], a[i] }
}
```

Example: array-reversing

```
func main() {
    a := [10] int { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
    fmt.Println("before reverse():", a);
    reverse(a) // array, passed by value
    fmt.Println("after reverse():", a) // a is not reversed
    fmt.Println("before reverse2():", a)
    reverse2(a[0:len(a)]) // slice, passed by reference
    fmt.Println("after reverse2():", a) // a has been reversed
}
```

User Defined Types

- type Point struct { X, Y float; } // type decl.
- p := Point{3, 4} // initialization
- func (p *Point) Translate(x, y float) { p.X += x; p.Y += y } // method
- func (p Point) String( ) string { // Print* functions use this method

```
        return fmt.Sprintf(“(x: %f, y: %f)”, p.x, p.y))
  }
```

Memory Allocation

- Two allocation primitives: **new(T), make(T**, arg**)**
- new(T) returns pointer for T type. Allocated memory area is zeroed
- For complex type, use “constructor” and composite literal, like

```
func NewPoint(x, y int) *Point {
  return &Point{x, y} // We can return the address of local variable
}
```

Memory Allocation

- **make(T**, arg**)** is for *arrays, maps, channels*
- These data types need internal-initialization
- Returns T type value, not T* type value
- eg) **var** p *[ ] int := &make([ ] int, 8) *// slice referring array of 8 elements*
- cf) **var** p *[ ] int = new([ ] int) *// refers array of length 0 (=nil)*
  *p = make([ ] int, 8, 8) *// type, length, (optional) capacity*

Interfaces

- Go has no *class*, no *inheritance*, nor *template*, ... but has “**interface**”
- **type** SomeInterfaceName **interface** { SomeInterfaceFunction(arg) rv;}
- To implement the interface, just define the methods in interface
  - **type** Magnitude **interface** { Abs( ) float; }
    func (p *Point) Abs( ) float { **return** math.Sqrt(p.X*p.X + p.Y*p.Y) }

Example: sort.interface

-
  ```
  type sort.interface interface { // interface from sort package
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
  }
  ```

-
  ```
  type IntSlice [ ] int  // define type to bind methods
  func (p IntSlice) Len( ) int { return len(p) }
  func (p IntSlice) Less(i, j int) bool { return p[ i ] < p[ j ] }
  func (p IntSlice) Swap(i, j int) { p[ i ], p[ j ] = p[ j ], p[ i ]
  ```

Concurrency Primitives

- Shared communication using **‘channels’**
  - Acts as **type-safe** UNIX-*pipe* like object
  - *Synchronization* + value-exchange
- Lightweight, shared-memory processes called **‘goroutines’**
  - *Little overhead* : a few stack space + alpha
  - Can be **multiplexed** over multiple OS-threads

Concurrency Primitives: channel

- Channel can be created bufferer/unbuffered

  -
    ```
    intUnbufferedChan := make(chan int) // same as make(chan int, 0)
    fileBufferedChan := make(chan *os.File, 100) // buffered channel
    ```
- Send to channel: intUnbufferedChannel <- 100
- Receive from the channel: x <- intUnbufferedChannel
- Channel is iteratable && can be passed as argument

Concurrency Primitives: goroutine

- Runs arbitrary functions in parallel

  ```
  go list.Sort( ) // run function in parallel
  go func ( ) { blah blah } ( ) // define and run anonymous function
  ```

- Go DONOT wait(or join) for forked go-routines
- Use channel to join, or to signal the go-routines

Example: producer-consumer

-
  ```
  channel := make(chan *string, 8)
  ```

-
  ```
  go func (ch chan *string) { // producer, anonymous function
      // ...make string
      ch <- &madeStr
  } (channel)
  ```

-
  ```
  go func (ch chan *string) { // consumer, anonymous function
      pStr <- ch; fmt.Println(*pStr) // consume; print string
  } (channel)
  ```

TODO

- Create examples for built-in types
- Build library package
- Multiplexing on channel-list
- Build simple server application: regex based query classifier?
- Performance test
- ...