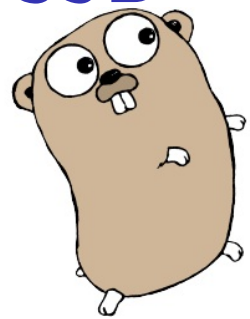
El lenguaje de programación Go
Introducción a la Concurrencia
UNPSJB - 2012



Defossé Nahuel

5 de noviembre de 2012

Introducción

Estructura del lenguaje

Instalación

Herramientas

Hola Mundo

Algunas caractrerísticas

Primitivas de Concurrencia
- Gorutinas
- Canales
- Select
- Grupos de espera
- Futures
- Diferidos

Más información

Introducción

- Google creó el lenguaje de programación Go debido a que:
  - La capacidad de los procesadores crecieron enormemente, pero el software no se ejecuta más rápido.
  - El manejo de dependecias de C es arcaico
  - La fuga de los programadores de lenguajes estáticos tipados como C++ o Java hacia los dinámicos como Python o JavaScript.
  - Características como `garbage collection` y computación paralela están ausentes en los lenguajes de programación de sistemas.
  - La aprición de muchos *cores* generó preocupación y confusión.

Estructura del lenguaje

Go es un lenguaje de programación *compilado* desarrollado por Google orientado a la concurrencia.

- Su sintaxis está basada en C, con algunas siplificaciones:
  - Todo programa en Go comienzan los la cláusula `package`
  - No se utilizan los `;` (punto y coma) para separar sentencias
  - Las cláusulas de condición no llevan paréntesis
  - Las variables se definen con la cláusula `var` y el tipo doc se ubica después de la variable (Ej: `var entero int`).
  - Las funciones se definen con `func` y su tipo se define después de sus argumentos doc
  - Tiene tipos de dato `int`, `complex`, `float`, `uint` y las constes `true`, `false` y `nil`. doc
  - Existe la clausula const para reemplazar los `#define`

Algunas diferencias con C

- Permite return con varios valores, generalmente usado para indicar errores y valores de retorno.

```
func x() {
    return "", nil
}
```

- Se pueden definir variables de manera automática utilizando el operador :=

```
a := 3 // En vez de var a int; a = 3
```

- Los arreglos llevan el tamaño al princpio

```
1 var arreglo [3] int [1 2 3]
2 arreglo2 := [...]string{"Hola", "mundo", "Go!"}
```

Algunas diferencias con C cont.

- Existe la cláusula `range` que nos permite recorrer arreglos o cadenas de manera sencilla.

```
1         for pos, char := range "abcd" {
2           fmt.Printf("Caracter %c empieza en la pos %d\n", char, pos)
3         }
```

- Para instalar Go en Ubuntu Wiki de Ubuntu

```
$ sudo add-apt-repository ppa:gophers/go
$ sudo apt-get update
$ sudo apt-get install golang-stable
```

- Para instalación en windows
  Descargar del Sitio Oficial y descomprimir en C: \go y agregar esa ruta a `GOROOT` y c: \go \bin a `PATH`

Compilación en Go

Los archivos fuentes de go tiene la extensión `.go` y se corren con la siguiente linea

```
$ go run miprograma.go
```

Para compilar el archivo para su distribución se utiliza `compile`

```
$ go build miprograma.go
```

Mi primer programa en Go

```go
// Mi primer programa en go

package main

import "fmt"

func main() {

  fmt.Printf("Hola mundo")
}
```

src

Algunas sentencias

1. Las sentencias `if` se escriben como:

```
a := 1 // Asignacio'n con autotipo
if a > 3 {
    fmt.Printf("a es mayor a 3")
}
```

2. La sentencia `for` se puede escribir como

```
for i:=0; i<10;i++{
    fmt.Printf("i = %d", i)
}
```

Mapa

Un mapa es una asociación clave valor.

```
1  // testgo project main.go
2  package main
3
4  import (
5  "fmt"
6  )
7
8  func main() {
9
10     mapa := make(map[string]int)
11     cadenas := []string{"uno", "dos", "tres", "dos", "cuatro"}
12     for _, cadena := range cadenas{
13         if _, ok := mapa[cadena]; ok {
14             //fmt.Printf("Ya existe\n")
15             mapa[cadena] += 1
16         } else {
17             //fmt.Printf("No existe\n")
18             mapa[cadena] = 1
19         }
20     }
21     for k, v := range mapa {
22         fmt.Printf("La cadena %-10s se encontro %d veces\n", k, v)
23     }
24 }
```

Gorutinas

Una gorutina es una función (o sentencia) que se ejecuta de manera concurrente y comparten el mismo espacio de direcciones. Si es posible se traduce en un hilo del sistema operativo.

```
package main
import "fmt"
func f(id int){
    fmt.Printf("Soy la gorutina %d", id)
}
func main(){
    for i:=0; i < 10, i++ {
        go f()
    }
}
```

Channel

Los canales son similares a las colas de mensaes y permiten la comunicación entre código concurrentes.
Se crean con la sentencia `make` y tienen un tipo de datos específico.
La sintaxis es muy sencilla:

```
canal <- "Hola mundo" // Poner en el canal
a <- canal  // Tomar del canal
```

Productor Consumidor con Gorutinas y Channels

```
1  package main;
2  import ("fmt"; "time"; "math/rand")
3
4  var canal = make(chan int)
5  //var listo = make(chan bool, 1)
6
7  func prod(){
8      for i:=0; i<100; i++{
9          // Producir un item
10         canal <- rand.Int() // Entero aleatorio
11     }
12
13 }
14 func cons(){
15    cantidad := 1
16    for {
17       entero := <-canal
18       fmt.Printf("Recibi %.2d %d\n", cantidad, entero)
19       cantidad++
20    }
21
22 }
23 func main(){
24     go prod()
25     go cons()
26     // Esperar a que terminen
```

Select

Select es similar a la cláusula `switch` pero permite hacer polling sobre varios canales de comunicación.

```
1         var c, c1, c2, c3 chan int
2         var i1, i2 int
3         select {
4         case i1 = <-c1:
5          print("received ", i1, " from c1\n")
6         case c2 <- i2:
7          print("sent ", i2, " to c2\n")
8         case i3, ok := (<-c3):  // same as: i3, ok := <-c3
9          if ok {
10         print("received ", i3, " from c3\n")
11         } else {
12         print("c3 is closed\n")
13         }
14        default:
15         print("no communication\n")
16        }
```

Locks

Dentro del pauete ‘‘`sync`’’ se encuentra una colección de funciones de bloqueo.

```
func (*Mutex) Lock
func (*Mutex) Unlock
```

Están implementadas con channels y se limitan a trabajo con recursos compartidos.

WaitGroups

Los grupos de espera son un conjunto de gorutinas que se deben esperar para poder continuar con la ejecución. Están en el paquete `sync`.

```
var grupo sync.WaitGroups
grupo.Add(1) // Agrega una gorutina
grupo.Done() // Termina una rutina
grupo.Wait() // Espera a que todas hallan hecho Done
```

Ver códgo

Futures

Un *future* es la promesa de un cálculo que se efectuará cuándo sea necesario su resultado. Es una aplicación de lo que se conoce como evaluación perezosa (lazy evaluation).

```
1  func InvertirMatrizFuture(mat Matrix){
2      future := make(chan Matrix) // Canal para recibir resultado
3      // Se lanza el c'alculo en una gorutina
4      go func () { future <- InvertirMatriz(mat)}
5      return future // Retornamos el canal
6  }
```

Futures Cont.

Para poder utilizar los resultados generados por una promesa o *future*

```
1  func InverseProduct (a Matrix, b Matrix) {
2      a_inv_future := InvertirMatrizFuture(a);
3      b_inv_future := InvertirMatrizFuture(b);
4      a_inv := <-a_inv_future;
5      b_inv := <-b_inv_future;
6      return Product(a_inv, b_inv);
7  }
```

Diferidos

Go permite programar la llamada a una función cuando un bloque de función termina mediante la cláusula `defer`.

```
1  package main
2  import "fmt"
3
4  func AlgoUtil(){
5      fmt.Printf("2 Realizo algo util...\n")
6  }
7
8  func Funcion() int {
9      defer fmt.Printf("1\n")
10     defer AlgoUtil()
11     defer fmt.Printf("3\n")
12     defer fmt.Printf("4\n")
13     return 1
14 }
15
16 func main(){
17     fmt.Printf("Retorno: %d\n", Funcion())
18 }
```

Más información

- Página de Go
- Effective Go
- Especificación del Lenguaje
- Patrones de Concurrencia con Go