Go: een snelle introductie

Auteur: Daniël Heres
Datum: Januari 2010

3,2,1 Go!

"Weer een nieuwe taal!" zul je wel denken. De keuze in programmeertalen is groot en de gelijkenis vaak ook. Toch is er een ding dat vaak overeind blijft: het gebruik van talen met statische types (system language). Deze talen zijn vaak een stuk sneller omdat er bijvoorbeeld met 32-bits typen gerekend kan worden terwijl bij de meeste dynamische talen er met 64-bit floats gerekend wordt of omdat een array uit één type kan bestaan. Een ander groot verschil tussen dynamische en statische talen is vaak de complexiteit van een taal: talen als C++ kunnen heel complex worden met hun klassensysteem. Daardoor kunnen eenvoudigere talen (vooral wanneer het aantal bestanden toeneemt) veel sneller gecompileerd worden. Dat blijkt vooral handig handig bij het ontwikkelen van programma's en bij JIT-(Just In Time) compileren. Go is een taal niet echt als doel heeft een nieuwe syntax door te voeren (zoals veel nieuwe dynamische talen) maar de snelheid van talen met statische types te combineren met het gemak van dynamische talen en een supersnelle compilatie. Deze taal wordt ontwikkeld door Google. De drijfveer van Google is duidelijk: enerzijds webapplicaties maken met veel meer mogelijkheden die op elk platform kunnen draaien en daarnaast snel afgeleverd worden en anderzijds sneller ontwikkelen van grote systemen door snelle compilatie. Go maakt een goede indruk als een alternatief voor system languages en JavaScript.

Deze gids is gemaakt voor mensen die al wat programmeerervaring hebben en snel basisconcepten van deze nieuwe taal willen leren.

Commentaar

Commentaar is hetzelfde als veel andere talen.

```
/* Dit is commentaar
over meerdere
regels */
// Commentaar voor één regel
```

Pakketten

De eerste regel van Go moet een pakketnaam beschrijven. In een Go project moet er minstens één pakket zijn met de naam main en die moet een functie met de naam "main" bevatten.

```
package kabouters // Andere pakketten in project kunnen dit kabouterpakket importeren

import "fmt" // Pakket bekend door compiler

import "./mutsen" // Pakket beschikbaar in map
```

Variabelen en constanten

Het declareren van variabelen en typen gaat in Go behoorlijk anders dan in de meeste talen. Typen worden achteraf gedeclareerd, na de namen. Dit is duidelijk te zien in het volgende voorbeeld.

```
var a int // Variabele a heeft type int. Let op: een puntkomma als einde voor een statement is voor Go niet noodzakelijk (mag wel)

var b, c, d, e int32 = 5, 7, 10, 4  // b, c, d en e hebben allemaal type int32, een regel mogen meerdere variabelen declareren. Let op int32 is niet hetzelfde als int

const r int = 5 // e is een constante

var f = 5 // Dit kan bij numerieke types, types worden automatisch toegewezen

var g string = "Go go go!"

var (
   h int;
   i string;
   j int32 = 45;
   k, l, m = "GOOG", 43, 1.0
) // Gebruik van haakjes is heel handig bij declareren van veel variabelen, puntkomma's zijn verplicht als scheidingsteken! Haakjes kunnen ook bij sleutelwoorden const en type gebruikt worden.

const (
  Maandag, Dinsdag, Woensdag = 1, 2, 3;
  Donderdag, Vrijdag, Zaterdag = 4, 5, 6;
)
```

```
//Let op, dit mag alléén binnen functies!
func functies() {
    n := 4 // Dit is een speciale korte dynamische declaratie voor variabelen

    o := functie0() // Korte notatie is ook erg handig bij aanroepen functies

    p, q, r := 5.7, 100, "Kort"
}
```

Merk op dat er aandacht besteed is aan het zo kort mogelijk opschrijven van meerdere variabelen.
De laatste constanten kunnen bijvoorbeeld door een programma verkleind worden tot

```
const(a,b,c,d,e,f=9,7,10,8,5,2)
```

Dat is natuurlijk geen toeval. Grote programma's moeten zo snel mogelijk over een netwerk verstuurd worden, daar helpt een korte notatie natuurlijk goed bij! Hier zal ook bij de volgende concepten goed over nagedacht zijn!

Tot slot, de numerieke types zijn:

int	uint	float
int8	uint8 of byte	
int16	uint16	
int32	uint32	float32
int64	uint64	float64

Assignments

Laten we meteen maar met een paar voorbeelden beginnen!

```
a = x

b, c = functie1(), 5 // Meerdere assignments, we zien in het volgende waar dat heel handig voor is

d, e = e, d // Wisselen van variabelen! Hiervoor zijn bij minder expressieve talen drie regels voor nodig
```

If

Voor een if-statement worden geen haakjes gebruikt. De rest is gelijk aan de meeste andere talen.

```
import "fmt" // importeert pakket fmt
```

```
func main() {
   if x<a {
      klein()
   } else if x>a { // Let op, else en else if moeten op zelfde regel als sluitende accolade!
      groot()
   } else {
      normaal()
   }

   if v:=s(); v<10 { // v wordt gedeclareerd, let op de puntkomma!
       fmt.Printf("Go " + v) // Geeft Go plus uitvoer van s in console weer
   }
}
```

Operators

Voorrang	
6	/ % << >> & &^
5	+ - | ^
4	== != < <= > >=
3	<- (Deze operator wordt gebruikt voor communicatie tussen Goroutines)
2	&&
1	||

For

Een for loop gebruikt ook geen haakjes. Een for loop heeft twee verschillende modussen: één met één element en één met drie elementen.

```
for a<5 {} // Dit is gelijk aan een while statement in andere talen

for {} // while(true)

for a:=0; ; a++{} // while(true) en een counter!

for a:=0; a<10; a++ {} // Een gewone for-loopt

for i := range x {} // Probeer deze op verschillende types! Let op: werkt niet voor integers zoals bijvoorbeeld bij programmeertaal Python.
```

Switch

Switches zijn grotendeels gelijk aan andere talen met een paar belangrijke verschillen: alle types kunnen gebruikt worden en er is geen break nodig (waarmee vaak fouten worden gemaakt), het switch statement stopt standaard zodra er een kloppende voorwaarde is.

```
switch f(10){
   case 100: Print("Vertienvoudigd!")
   case 10: Print("Gelijk!")
   case 2,3,5: Print("Nu, dit is wezenlijk verrassend!")
   default: Print("Standaard!")
}

switch f:=f(10);{ // Je kan ook meerdere variabelen of declareren!
   case f>100: Print("Erg groot");// Expressies in een switch statement
   case f>10: Print("Aardig groot") fallthrough; // Gaat door naar volgende case, ook als hij waar is
   case f==20: Print("Aardig groot en ook nog 20!")
}
```

Functies

Kenmerkend voor Go zijn de notatie van parameters en de mogelijkheid om meerdere waardes te "returnen". Een functie wordt gedeclareerd met "func".

```
/* Meerdere waardes kunnen voor het registreren van fouten gebruikt worden */
func Wortel(parameter int) (float, bool) {
   if parameter > 0 { return math.Sqrt(parameter), true }
   return 0, false
}

func Konijn(wortels int) (genoeg bool) { // Als je parameters benoemt kun je ze gebruiken
   if wortels > 10 { genoeg = true; }
   return genoeg
}

func Konijn(wortels int) (genoeg bool) {
   if wortels > 10 { genoeg = true; }
   return // genoeg wordt "gereturned", kleiner dan tien -> false
}

func FunctieInFunctie(i int) (int) (
   g := func(j int) (int) {
      h := j * i + 1;
      return h
   }
   return g(i +5) + 5;
}

func doeDitEnDat(){ // void in meeste talen
   x = x +5
}
```

Defer

Defer is een statement die als functie heeft om de executie van een statement te verplaatsen naar het einde van een functie. De volgorde van executie is LIFO, het laatste

statement wordt als eerste uitgevoerd. Dit kan handig zijn om meerdere functies te groeperen, bijvoorbeeld bij IO.

```
func main(element InfoElement)(Info) {
  o := open(element)
  defer o.Sluiten() // Verplaatst zich naar onderen

  p := o.Info() // Sluit is nog niet uitgevoerd!
  // Nu pas!
  return p
}
```

Arrays

```
var x = [4]int{2, 8, 19, 30} // Maak een array met lengte 4 en waardes 2, 8, 19 en 30
var y = [4]int{2, 9} // Array met lengte 4, met waardes 2, 9, 0, 0
var z = [100]int{20:7, 65:8,99:2} // Array met lengte 100. Index 20 heeft waarde 7, 65 waarde 8 enz. De rest heeft de waarde 0.

var a = [4]string{"bas", "bink", "bobo"} // Index 3 is empty string
var b = [7]MijnEigenType{} // Eigen types kunnen ook, volgt later!

func Arrays() {

   c := [2]string{"Go", "ne"}

   for i := range x {
      fmt.Print(i) // 0, 1, 2, 3
      fmt.Print("\n")

   for _, j := range x {  // Een array heeft keys (index) en waardes, indien je keys niet nodig hebt gebruik je een laag streepje om alleen de waardes op te vragen!
      fmt.Print(j) // 2, 8, 19, 30
      fmt.Print("\n")
   }

   for i := range x {
      x[i]  = i
   } // Maakt {0, 1, 2, 3}

   x[2] = 0

   fmt.Print(x[1:len(x)]) // Print alles behalve de eerste. len(x) rekent lengthe array uit.

   fmt.Print([3]int{7, 9, 9}) // Nieuwe array

}
```

Slices

Slices zijn vrij uniek voor Go. Slices zijn eigenlijk verwijzingen naar arrays met een variabele lengte. Omdat slices verwijzingen zijn hoef je niet met pointers te werken.

```
var a = []int{} // Integer slice
var b = []string{} // String slice

var x = make([]int, 0, 100) // Deze kan je tot 100 herschalen

func ToevoegenAanSlice(i int, sl []int) []int {
  if len(sl) == cap(sl) { return sl }
  n := len(sl);
  sl = sl[0:n+1]; // Verleng met 1
  sl[n] = i;
  return sl
}
```

Interfaces

Interfaces zijn in Go niks anders dan functies met een ontvanger. De ontvanger komt voor de naam van de functie.

```
func (o *ObjectType) FunctieNaam() (aanroep int) returnwaarde int {
}
// Dus
func (v *Vector2D) Optellen() (w *Vector2D) {
   v.x = v.x + w.x
   v.y = v.y + w.y
}
```

We zien bij het onderwerp "types" hoe we interfaces in functies kunnen gebruiken.

Types

In Go kan je verschillende soorten types aanmaken. Types zijn niet hetzelfde als types in andere programmeertalen.

Soort type	**Eigenschap**
Type van een al bestaand type	Nieuwe naam voor type, mogelijk uitbreiden met methodes.
Interface	Beschrijft interfaces.
Struct	Vormt structuur van geheugen (velden en methoden). Lijkt een beetje op een klasse uit andere talen maar heeft niet precies dezelfde functie. Je kan instanties van dit type aanmaken.

Type van een type

```
type Dag int
type Kracht float
type Mensen []string
```

Interface

```
type Honden interface { // Specificatie interface
   Rennen(h *Hond) // *Voer betekent verwijzing naar object met type Voer
   Eten(h *Hond) bool
}

func (h *Hond) Rennen() { // Interface, kan ook zonder specificatie
                          // *Hond is recievertype, kan voor elk type
   h.ren = true
}

func (h *Hond) Eten(v Voer) bool { // Nog een interface
   if (h.ren || v == PrutUitBlik) {
      return false
   }
   return true
}
```

Struct

```
type Hond struct {
   x,y int
   ren bool
}

type Voer int;

const (
   Brokken Voer = iota // iota is een enumerator. Standaard 0, 1, 2 enzovoort
   Vlees
   PrutUitBlik
)

func main(){

   monty := &Hond{5, 2, false} // Verwijzing naar hond
   monty.Rennen()
   fmt.Print(monty.Eten(Brokken)) // Wat wordt dit?
}
```

Goroutines

Programmeurs vinden multithreading vaak lastig. Go heeft het uitvoeren van meerdere code in dezelfde geheugenruimte in de taal ingebouwd om het een stukje makkelijker te maken. Een Goroutine wordt gestart door het sleutelwoord "go".

```
func zegHoi() {
   fmt.Print("Hoi")
}

func main() {
  go zegHoi() // zegHoi wordt uitgevoerd en gaat meteen naar volgende regel
  time.Sleep(1000000000) // Nanoseconden
```

```
}
```

We kunnen ook een anonieme functie gebruiken.

```
func main() {
   go func() {
      fmt.Print("hoi")
   }() // Let op, functie moet worden aangeroepen!
   time.Sleep(1000000000)
}
```

Buiten dat het geen zin heeft om een aparte Goroutine te openen om naar een console te schrijven is er iets anders: er is helemaal geen garantie dat de Goroutines worden uitgevoerd.
Daarvoor zijn Channels in het leven geroepen.

Channels

Channels worden gebruikt om te wachten en om informatie uit te wisselen, bijvoorbeeld over de status van een berekening.

```
var quit = make(chan bool) // Maakt een channel met communicatietype bool

func main() {
   go func() {
      fmt.Print("hoi")
      quit <- true // stuur waarde true naar channel quit
   }()
   x := <-quit // Wacht op quit en sla waarde op in x, gebruik "<-quit" om niks op te slaan!
   fmt.Print(x) // Geeft output true in console
}
```

We kunnen dat zo vaak als we willen communiceren met een Goroutine.

```
var comm = make(chan int)

func main() {
   go func() {
      i := 0
      for {
         time.Sleep(1000000000)

comm <- i // Stuur nummer van berekening door naar channel comm
         i++
      }
   }()
   for {
      x := <-comm
      fmt.Printf("%d \n", x)
   }
```

```
}
```

We kunnen meerdere Goroutines parallel uitvoeren. Om meerdere channels aan te maken gebruiken we
***channelnaam* := make(chan *Type*, *aantal*).** De volgende code simuleert een hond met 4 koppen die elk 1 seconde nodig hebben om een brok op te eten.

```
const Cores = 4

type Hond struct {
   x,y int
   ren bool
   stuks int
}

func (h *Hond) Brok(c chan bool) {
   time.Sleep(1000000000)
   h.stuks = h.stuks - 1
   fmt.Printf("Brok opgegeten, nog %d \n", h.stuks)
   c<-true
}

func (h *Hond) EetBrokken(brokken int) {
   h.stuks = brokken
   c := make(chan bool, Cores)
   j := brokken / Cores
   for k:=0; k<j; k++ { // We willen (brokken / Cores) keer met 4 koppen eten
      for i:= 0; i<Cores; i++ {
         go h.Brok(c)
      }
      for i:= 0; i<Cores; i++ {
         <-c
      }
   }
   nogtedoen := h.stuks // Als aantal brokken niet te delen is door 4 moeten we nog een
aantal brokken eten
   if nogtedoen==0 {return}
   for i:=0; i<nogtedoen; i++ {
      go h.Brok(c)
   }

   for i:= 0; i<nogtedoen; i++ {
      <-c
   }

}

func main(){
  rex := &Hond{7, 9, true, 0}
  rex.EetBrokken(50)
}
```