Go

Un langage déficient ?

Tour d’horizon

Fiche d’identité

Surnom: Golang

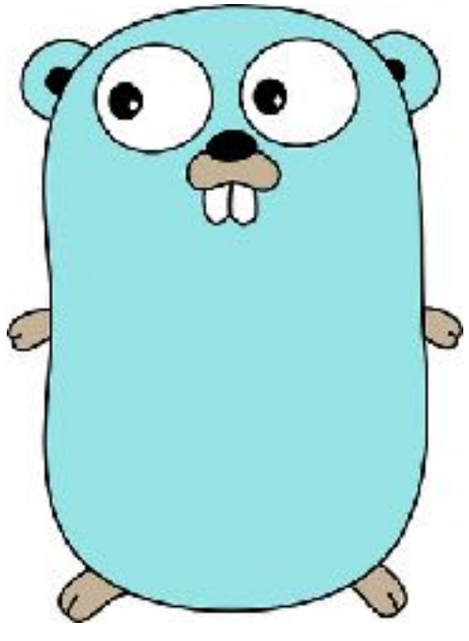
Type: compilé

Gestion de la mémoire: Gabarage collector

Paradigmes: impératif, fortement typé, structurel

Domaines: programmation système, web

Philosophie: Forte opinion, Share memory by communicating



Historique

- - 2007 Google crée le langage Go
- - Mars 2012: Version 1.0 de Go
- - 17 February 2016: Go 1.6 le compilateur Go est écrit en Go

Rob Pike

Unix Team

Ken Thompson

B Programming Language, c predecessor

Robert Griesemer

V8 engine, Google Distributed File System

Pourquoi ?

- Plus simple que C++
- Plus sûre que C
- Accessible aux développeurs backend et aux ops
- Programmation concurrent facile, efficace et fun

Quelques chiffres

- Parse a réduit son pool de serveur de 90% en migrant de Ruby à Go. (Puis a fait faillite)

- De 10 heures à quelques minutes. Temps de compilation du repo C++ de Google vs quelques minutes pour Go.

Quelques chiffres

	Clojure	Elixir	Go
Github Stars	5,120	6,445	14,994
Github Contributors	125	358	597
Github Projects	36,988	9,338	111,364
Most Popular Projects	LightTable, ClojureScript, Om	Phoenix, Ecto, Dynamo	Docker, Kubernetes, Gogs
Stackshare Stacks	112	60	502
Stackshare Fans	143	82	553
Stackoverflow Questions	11,153	1,475	14,146
Stackoverflow Followers	5,100	1,400	8,800
Original Release Year	2007	2012	2009

Go powered


InfluxDB


docker


Dropbox


mailjet


Grafana


CONSUL


Xee


SendGrid


ethereum

Les faiblesses du langage

Génériques

```
package main

import (
    "fmt"
)

type Single<T> struct{
    T value,
}

func (s Single<T>) Value() T {
    return s.Value
}

func main() {
    var single = Single<int>{T: 2}

    fmt.Println(single.Value())
}
```

Polymorphisme paramétrique

```
package main

import (
    "fmt"
)

type Single<T> struct{
    T value,
}

func (s Single<T>) Value() T {
    return s.Value
}

func main() {
    var single = Single<int>{T: 2}

    fmt.Println(single.Value())
}
```

Polymorphisme paramétrique

```
package main

import (
    "fmt"
)

type Single<T> struct{
    T value,
}

func (s Single<T>) Value() T {
    return s.Value
}

func main() {
    var single = Single<int>{T: 2}

    fmt.Println(single.Value())
}
```

FAUX

S’il te plaît Rob Pike donne nous des types génériques

~~Covariance sur les tableaux~~

Gopath

```
$GOPATH/
    bin/
        hello                 # command executable
        outyet                # command executable
    pkg/
        linux_amd64/
            github.com/golang/example/
                stringutil.a  # package object
    src/
        github.com/golang/example/
            .git/             # Git repository metadata
            hello/
                hello.go      # command source
            outyet/
                main.go       # command source
            stringutil/
                reverse.go    # package source
```

Pointer receiver

```
package main

import "fmt"

type Jedi struct {
    Name    string
    padawan string
}

func (j Jedi) SetPadawan(name string) {
    j.padawan = name
}

func main() {
    var j = Jedi{
        Name: "Obiwan",
    }

    j.SetPadawan("Anakin")

    fmt.Println(j.padawan)
}
```

Pointer receiver

```go
package main

import "fmt"

type Jedi struct {
    Name    string
    padawan string
}

func (j *Jedi) SetPadawan(name string) {
    j.padawan = name
}

func main() {
    var j = Jedi{
        Name: "Obiwan",
    }

    j.SetPadawan("Anakin")

    fmt.Println(j.padawan)
}
```

Interopérabilité avec C

```
package rand

/*
#include <stdlib.h>
*/
import "C"

func Random() int {
    return int(C.random())
}

func Seed(i int) {
    C.srandom(C.uint(i))
}
```

Intéroperabilité avec C

- Pas de cross compilation
- Système basé sur des commentaires
- Chaque appel à C coute 20ns au minimum

Pas de contrôle de la mémoire

- Pas de temps réel
- Pas d’optimisation des performances possibles

Vendoring

Workflow par Google pour Google

- Utilisation de la branche master du repo distant
- Pas de sémantique versioning
- Versioning complet du GOPATH pour sauvegarde

Vendoring

Glide à la rescousse

- Gestion du sémantique versioning
- Freeze des versions avec un glide.lock
- Alias de namespace
- Gestion des privates repositories

Inférence de types

```go
package main

import (
    "fmt"
)

type Type1 struct {
    Prop1 int
}

type Type2 string

func println(t1 Type1, t2 Type2) {
    fmt.Println(t1)
    fmt.Println(t2)
}

func main() {
    t1 := Type1{
        Prop1: 1,
    }

    var t2 = "hello"

    println(t1, t2) // compile error
}
```

Inférence de types

```go
package main

import (
    "fmt"
)

type Type1 struct {
    Prop1 int
}

type Type2 string

func println(t1 Type1, t2 Type2) {
    fmt.Println(t1)
    fmt.Println(t2)
}

func main() {
    t1 := Type1{
        Prop1: 1,
    }

    var t2 Type2 = "hello"
    println(t1, t2) // {1} hello
```

Les atouts du langage

Go routines

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

Channels

```go
package main

import "fmt"

func main() {

    // We'll iterate over 2 values in the `queue` channel.
    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)

    // This `range` iterates over each element as it's
    // received from `queue`. Because we `close`d the
    // channel above, the iteration terminates after
    // receiving the 2 elements.
    for elem := range queue {
        fmt.Println(elem)
    }
}
```

Concurrence

```
package main

import "fmt"
import "time"

func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second)
    fmt.Println("done")

    // Send a value to notify that we're done.
    done <- true
}

func main() {
    done := make(chan bool, 1)
    go worker(done)

    <-done
}
```

Multiple valeurs de retour

```go
package main

import "fmt"

func vals() (int, int) {
    return 3, 7
}

func main() {
    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    _, c := vals()
    fmt.Println(c)
}
```

Defer

```go
package main

import "fmt"

func main() {
    defer fmt.Println("world")

    fmt.Println("hello")
}
```

```
hello
world
```

Interfaces implicites

```go
package main

import (
    "fmt"
)

type Namer interface {
    Name() string
}

type Poney struct {
    name string
}

func (p Poney) Name() string {
    return p.name
}

func printName(n Namer) {
    fmt.Println(n.Name())
}

func main() {
    var p = Poney{
        name: "Fauche le vent",
    }

    printName(&p) //Fauche le vent
}
```

Marshalling

```go
package main

import (
    "encoding/json"
    "fmt"
)

type Kitty struct {
    ID     int
    Name   string
    Age    int
    secret string
}

func main() {
    var k = Kitty{
        Name:   "Garfield",
        Age:    5,
        secret: "Je suis un baron de la drogue.",
    }

    var data, _ = json.Marshal(&k)

    fmt.Println(string(data)) // {"ID":0,"Name":"Garfield","Age":5}
}
```

Marshalling

```go
package main

import (
    "encoding/json"
    "fmt"
)

type Kitty struct {
    ID     int    `json:",omitempty"`
    Name   string `json:"name"`
    Age    int    `json:"-"`
    secret string
}

func main() {
    var k = Kitty{
        Name:   "Garfield",
        Age:    5,
        secret: "Je suis un baron de la drogue.",
    }

    var data, _ = json.Marshal(&k)

    fmt.Println(string(data)) // {"name":"Garfield"}
}
```

Compilation ultra rapide

Cross compilation

- Distribuer facilement les binaires via la CI
- Créer des conteneur Docker via OSX
- Compilation plus lente
- Pas de binding C

Un seul type de boucles

```go
package main

import (
    "fmt"
)

func main() {
    animals := []string{
        "cat",
        "dog",
        "unicorn",
    }

    for _, animal := range animals {
        fmt.Println(animal)
    }

    for i := 0; i < len(animals); i++ {
        fmt.Println(animals[i])
    }

    i := 0
    for i < len(animals) {
        fmt.Println(animals[i])
        i++
    }
}
```

Import explicite

Déterminer facilement la provenance d'une dépendance

```
package main

import (
    "net/http"

    "github.com/gorilla/mux"
)

func handler(w http.ResponseWriter, r *http.Request) {
    return
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", handler)
    http.Handle("/", r)
}
```

Performances

- Faible consommation de mémoire
- Utilisation de tous les coeurs du CPU
- Rapidité d’execution proche de celle de Java

Godoc

Tooling

- Code linting
- Code generation
- Testing
- Profiling

Demo

Go

Un langage ~~déficient~~ d'efficient !