Le langage Go

1ère partie

Rob Pike
r@google.com

Traduction en français
xavier.mehaut @gmail.com

(Version de Juin 2011)

Qui?

Travail effectué par une petite équipe chez Google, plus beaucoup de contributeurs du monde entier.

Contact:

- http://golang.org*: web site*
- golang-nuts@golang.org: *user discussion*
- golang-dev@golang.org: *developers*

Google

Plan du cours

1ere jour

- *Les bases*

2ème jour

- *Types, méthodes, et interfaces*

3ème jour

- *Concurrence and communication*

Ce cours se focalise sur la programmation en GO et non pas sur la genèse et conception du langage lui-même.

Google

Aperçu du cours d'aujourd'hui

- *Motivations*
- *Les bases*
  - *Du simple et du familier*
- *Les « packages » et la construction d'un programme*

Google

Motivations

Google

Pourquoi un nouveau langage?

Dans le monde qui est le nôtre , les langages actuels ne nous aident pas suffisamment :

- *Les ordinateurs sont rapides mais développer un logiciel est lent*
- *Une analyse de dépendance est nécessaire pour la rapidité et la sûreté de fonctionnement*
- *Typage trop verbeux*
- *Le ramasse-miettes et la concurrence sont trop pauvrement pris en charge*
- *Les multi-coeurs sont considérés comme des sources d eproblème et non des opportunités*

Google

Pour positiver

Notre but est rendre la programmation de nouveau fun

- *Le feeling d'un langage d'un dynamique avec la sûreté de fonctionnement d'un système typé statiquement*
- *Compiler en langage machine pour une exécution rapide*
- *Run-time temps réel qui supporte un ramasse-miettes et la concurrence*
- *Léger, système de type flexible*
- *Possède des méthodes comme un langage objet mais pas de manière conventionnelle*

Google

Ressources

Pour plus d'information, voir la documentation sur le site :

- http://golang.org/

Inclus:

- *Spécification du langage*
- *tutoriel*
- *« Effective Go »*
- *documentation des bibliothèques de code*
- *Démarrage et « how-to» s*
- *FAQs*
- *Bac à sable (exécuter Go à partir d'un navigateur)*
- *Etc…*

Google

Status : compilateurs

gc (Ken Thompson), a.k.a. 6g, 8g, 5g

- *Dérivé du modèle de compilation de Plan9*
- *Génère du code OK très rapidement*
- *Pas directement linkable avec gcc*

gccgo (Ian Taylor)

- *Architecture plus familère*
- *Gén-re un bon code mais pas aussi rapide*
- *Linkable avec gcc*

Disponible pour les architectures 32-bit, 64-bits x86 (amd64, x86-64) et ARM.

Ramasse-miettes, concurrence, etc.., implémentés.
Bibliothèques de bonne qualité et améliorées.

Google

Les bases

Google

Codons

```
package main

import "fmt"

func main() {
  fmt.Print("Hello, bonjour\n")
}
```

Google

Bases du langage

En pré-supposant votre familiarité avec des langage de type C, nous allons brosser un rapide aperçu des bases du langage.

Ce sera facile, familier et sans doute rébarbatif. Toutes mes excuses pour cela.

Les deux prochaines parties contiennent plus de matière à amusement, mais nous devons poser les bases en premier lieu.

Google

Structure lexicale

Traditionnel avec quelques détails plus modernes

Le code source est UTF-8.

Espaces muets : espace , tabulation, nouvelle ligne , retour charriot.

Les identificateurs sont des lettres et des nombres (plus '_') avec with “lettre" et “nombre" définis par Unicode.

Commentaires:

```
/* Ceci est un commentaire; pas d’imbrication*/
// En voilà un autre.
```

Google

Litéraux

Similaire au C mais les nombres requièrent aucun signe ni marque de taille :

```
23
0x0FF
1.234e7
```

Similaire au C, mais Unicode/UTF-8. Egalement \xNN toujours 2 digits; \012 toujours 3; les deux sont des octets:

```
"Hello, world\n"
"\xFF" // 1 octet
"\u00FF" // 1 caractère Unicode, 2 octets de UTF-8
```

Chaînes de caractères:

```
`\n\.abc\t\` == "\\n\\.abc\\t\\"
```

Google

Aperçu de la syntaxe

Principalement similaire au C avec des types et déclarations renversés, plus des mots clefs pour introduire le type de déclaration

```
var a int
var b, c *int // notez la  difference avec le C
var d []int
type S struct { a, b int }
```

Les structures de contrôles sont familières :

```
if a == b { return true } else { return false }
for i = 0; i < 10; i++ { ... }
```

Note: pas de parenthèses, mais des accolades requises

Google

Point virgule

Les points virgule terminent les déclarations mais :

- *lexer les insert automatiquement **à la fin d'une ligne** si le token précédent la fin de ligne est une fin de règle grammaticale.*
- *Note: bien plus propre et simple que les règles javascript!*

Ainsi aucun point virgule n'est nécessaire dans ce programme :

```
package main
const three = 3
var i int = three
func main() { fmt.Printf("%d\n", i) }
```

En pratique, un code en Go ne possède quasiment aucun point virgure en dehors des clauses fo et if

Google

Les types numériques

Numeric types are built in, will be familiar:

int	uint		
int8	Uint8=byte		
int16	uint16		
Int32	uint32	Float32	Complex64
int64	uint64	Float64	complex128

*Auquels on peut ajouter **uintptr**, un entier suffisamment grand pour stocker un pointeur.*

*Ce sont tous des types distincts ; **int** n'est pas pas un **int32** même sur une machine 32 bits!*

Pas de conversion implicite (mais ne paniquez pas ☺).

Google

Bool

Le type booléen usuel, bool, avec comme valeurs uniques true et false (constantes prédéfinies).

*La déclaration **if** utilise des expressions booléennes.*

Les pointeurs et les entiers ne sont pas des booléens.

Google

String

Le type natif string représente un tableau d'octets immutable , c'est à dire du texte. Les chaînes de caractères sont de taille délimitée non nulles.

Les chaînes de caractères littérales sont de type string.

Immutable, tout comme les ints. On peut réaffecterdes variables mais pas éditer leur contenu.

Comme 3 est toujours 3, "bonjour" reste "bonjour".

Le langage Go possède un bon support de manipulation de chaînes.

Google

Expressions

Globalement comme les opérateurs C

Ordre de précédence	Opérateurs	Commentaires
5	* / % << >> & &^	&^ est « bit clear »
4	+ - | ^	^ est « xor »
3	== != < <= > >=	
2	&&	
1	||	

Les opérateurs qui sont également unaires : & ! * + - ^ (plus <- pour les communications).

L'opérateur unaire ^ est complément.

Google

Expressions Go vs. Expressions C

Surprenant pour le développeur C:

Moins de niveaux de précédence (devrait être simple)
*^ à la place de ~ (c'est le **ou exclusif** binaire fait unaire)*
++ et – ne sont pas des opérateurs expression
(`x++` est une déclaration , pas une expression;
`*p++` est `(*p)++` pas `*(p++)`)
&^ est nouveau; pratique dans les expressions constantes
<< >> etc. requiert un entier non signé de décalage

Non surprenant:
Les opérateurs d'affectation fonctionnent comme prévus : += <<= &^= etc.
Les expressions ressemblent généralement à celles en C (indexation, Appel de fonctions, etc.)

Google

Exemples

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y + 1 && <-ch > 0
x &^ 7 // x avec les  3 bits de poids faible
  vidés
fmt.Printf("%5.2g\n", 2*math.Sin(PI/8))
7.234/x + 2.3i

"hello, " + "world" // concatenation
                    // pas comme en C "a" "b"
```

Google

Conversions numériques

Convertir une valeur numérique d'un type à un autre est une conversion, avec une syntaxe similaire à un appel de fonction :

```
uint8(intVar) // tronque à la taille du type récepteur
int(float64Var) // tronque la fraction
float64(intVar) // convertit en float64
```

*Il existe également des conversion de ou vers **string** :*

```
string(0x1234) // == "\u1234"
string(sliceOfBytes) // octets -> octets
string(sliceOfInts) // ints -> Unicode/UTF-8
[]byte("abc") // octets -> octets
[]int("□ □ □ ") // Unicode/UTF-8 -> ints
```

(Les slices sont liées aux tableaux vus plus tard)

Google

Constantes

Les constantes numériques sont des “nombres idéaux” : pas de taille ni de signe, et même plus pas de terminaison en L ou U ou UL .

```
077 // octal
0xFEEDBEEEEEEEEEEEEEEEEEEEEF // hexadécimal
1 << 100
```

Il existe également des nombres flottants ou entiers idéaux;

La syntaxe des litéraux détermine le type :

```
1.234e5 // point flottant
1e2 // point flottant
3.2i // point flottant imaginaire (nombre complexe)
100 // entier
```

Google

Expressions constantes

Les constantes à point flottant et entières peuvent être combinées à volonté, avec comme type résultant déterminé par le types de constantes utilisées.

Les opérations elles-mêmes dépendent du type :

```
2*3.14 // point flottant: 6.28
3./2 // point flottant : 1.5
3/2 // entier: 1
3+2i // complexe: 3.0+2.0i
// haute précision:
const Ln2 = 0.693147180559945309417232121458176568075
const Log2E = 1/Ln2 // réciproque précis
```

La représentation est “assez grande" (1024 bits pour le moment).

Google

Conséquence des nombre idéaux

Le langage permet l'utilisation de constantes sans conversion explicite si la valeur peut être converti dans le type réceptacle(il n'y a pas de conversion nécessaire ; la valeur est OK):

```
var million int = 1e6 // la syntaxe flottante est OK
math.Sin(1)
```

Les constantes doivent être convertibles dans leur type.

Example: ^0 est -1 qui n'est pas dans l'intervalle 0-255.

```
uint8(^0) // faux: -1 ne peut être converti en entier
  non signé
^uint8(0) // OK
uint8(350) // faux: 350 ne peut être converti en entier
uint8(35.0) // OK: 35
uint8(3.5) // faux: 3.5 ne peut être converti en entier
```

Google

Déclarations

*Les déclarations sont introduites par un mot clef (var,*const*, type, func) et sont inversées si on compare au C :*

```
var i int
const PI = 22./7.
type Point struct { x, y int }
func sum(a, b int) int { return a + b }
```

Pourquoi sont elles inversées? Exemple précédent:

```
var p, q *int
```

Peut être lu ainsi

```
Les variables p et q sont des pointeurs sur entier
```

Plus clair non ? Presque du langage naturel.

Une autre raison sera vue ultérieurement.

Google

Var

*Les déclarations de variable sont introduites par le mot clef **var**.*

Elles peuvent contenir un type ou une expression d'initialisation typante ; l'un ou l'autre obligatoire!

```
var i int
var j = 365.245
var k int = 0
var l, m uint64 = 1, 2
var nanoseconds int64 = 1e9 // constante float64!
var inter, floater, stringer = 1, 2.0, "hi"
```

Google

Var distribué

Il est pénible de taper var tout le temps! Nous pouvons regrouper toutes les variables de la façon suivante :

```
var (
    i int
    j = 356.245
    k int = 0
    l, m uint64 = 1, 2
    nanoseconds int64 = 1e9
    inter, floater, stringer = 1, 2.0, "hi"
)
```

Cette règle s'applique aux `const, type, var` *mais pas à* `func`*.*

Google

La « déclaration courte » :=

Dans le corps des fonctions (seulement), les déclarations de la forme

```
var v = value
```

peuvent être raccourcis en

```
v := value
```

(une autre raison pour l'inversion name/type vue précédemment par rapport au C)

Le type est ce que la valeur est (pour les nombres idéaux, on a int, float64 et complex128 respectivement)

```
a, b, c, d, e := 1, 2.0, "three", FOUR, 5e0i
```

Ce type de déclaration est beaucoup utilisé et sont disponibles dans des endroits comme des initialiseurs de boucles.

Google

Les constantes

*Les constantes sont déclarées par le mot clef **const**.*

Elles doivent avoir une “expression constante”, évaluée à la compilation, comme initialiseur et peuvent être typées de manière optionnelle.

```
const Pi = 22./7.
const AccuratePi float64 = 355./113
const beef, two, parsnip = "meat", 2, "veg"

const (
    Monday, Tuesday, Wednesday = 1, 2, 3
    Thursday, Friday, Saturday = 4, 5, 6
)
```

Google

Iota

Les constantes peuvent utiliser le compteur iota, qui démarre à 0 dans chaque bloc et qui s'incrémente à chaque point virgule implicite (fin de ligne).

```
const (
    Monday = iota // 0
    Tuesday = iota // 1
)
```

Raccorcis: les types et l'expressions précédents peuvent être répétés.

```
const (
    loc0, bit0 uint32 = iota, 1<<iota  // 0, 1
    loc1, bit1                         // 1, 2
    loc2, bit2                         // 2, 4
)
```

Google

Types

*Les types sont introduits par le mot clef **type**.*

Nous reviendrons plus précisément sur les types plus tard mais voici tout de même quelques exemples :

```
type Point struct {
    x, y, z float64
    name string
}

type Operator func(a, b int) int
type SliceOfIntPointers []*int
```

Nous reviendrons sur les fonctions également un peu plus tard.

Google

New

La fonction native new alloue de la mémoire. La syntaxe est similaire à celle d'une fonction, avec des types comme arguments similaires au c++. Retourne un pointeur sur un objet alloué.

```
var p *Point = new(Point)
v := new(int) // v a le type *int
```

Nous verrons plus tard comment construire des slices et autres.

*Il n'y a pas de **delete** ni de **free** ; Go possède un ramasse-miettes.*

Google

Affectations

L'affectation est simple et familière :

```
a = b
```

Mais les affectations multiples fonctionnent aussi :

```
x, y, z = f1(), f2(), f3()
a, b = b, a // swap
```

Les fonctions peuvent retourner des valeurs multiples :

```
nbytes, error := Write(buf)
```

Google

Les structures de contrôles

Similaire au C, mais différents sur certains points.

*Go possède des **if**, **for** et **switch** (plus un de plus que nous verrons plus tard).*

Comme indiqué précédemment, pas de parenthèses ni accolades obligatoires.

*Les **if**, **for** et **switch** acceptent tous des déclarations d'initialisation.*

Google

Forme des structures de contrôle

*Les déclarations **if** et **switch** seront décrites dans les slides suivants . Ils peuvent avoir une ou deux expressions.*

*La boucle **for** peut posséder une ou trois expressions :*

*un seule élément correspond au **while** du C:*

```
for a {}
```

*trois élements correspond au **for** du C :*

```
for a;b;c {}
```

Dans aucune de ces formes, un élement ne peut être vide.

Google

If

La forme de base est familière, Basic form is familiar :

```
if x < 5 { less() }
if x < 5 { less() } else if x == 5 { equal() }
```

L'initialisation d'une déclaration est autorisée ; requiert un point virgule.

```
if v := f(); v < 10 {
  fmt.Printf("%d less than 10\n", v)
} else {
  fmt.Printf("%d not less than 10\n", v)
}
```

Utile avec des fonctions à multivariables :

```
if n, err = fd.Write(buf); err != nil { ... }
```

Des conditions manquantes signifient ***true****, ce qui n'est pas très utile dans ce contexte mais utile dans les* ***for*** *et* ***switch****.*

Google

For

La forme de base est familière :

```
for i := 0; i < 10; i++ { ... }
```

La condition manquante signifie true :

```
for ;; { fmt.Printf("looping forever") }
```

Mais vous pouvez éliminer les points virgule également:

```
for { fmt.Printf("Mine! ") }
```

Ne pas oublier les affectations multi-variables :

```
for i,j := 0,N; i < j; i,j = i+1,j-1
{...}
```

Google

Switch

*Les **switchs** sont globalement similaires à ceux du C.*

Mais il existe des différences syntaxiques et sémantiques :
- les expressions n'ont pas besoin d'être constantes ou bien entières
- pas d'échappement automatique
- à la place, la dernière déclaration peut être **`fallthrough`**
- les cas multiples peuvent être séparés par des virgules

```
switch count%7 {
    case 4,5,6: error()
    case 3: a *= v; fallthrough
    case 2: a *= v; fallthrough
    case 1: a *= v; fallthrough
    case 0: return a*v
}
```

Google

Switch (2)

Les switchs en Go sont bien plus performants qu'en C. La forme familière :

```
switch a {
    case 0: fmt.Printf("0")
    default: fmt.Printf("non-zero")
}
```

Les expressions peuvent être de n'importe quel type et une expression manquante signifie true. Résultat : la chaîne if-else avec un switch donne :

```
a, b := x[i], y[j]
switch {
    case a < b: return -1
    case a == b: return 0
    case a > b: return 1
}
```

or

```
switch a, b := x[i], y[j]; { ... }
```

Google

Break, continue, etc...

Les déclarations break et continue fonctionnent comme en C.

On peut également spécifier un label pour sortir de la structure de contrôle :

```
Loop: for i := 0; i < 10; i++ {
  switch f(i) {
   case 0, 1, 2: break Loop
  }
  g(i)
}
```

*Oui Ken (Thomson), il y a un **goto**!*

Google

Fonctions

Les fonctions sont introduites par le mot clef ***func****.*

Le type de retour, s'il y en a, vient après les paramètres dans la déclaration. Le ***return*** *se comporte comme vous l'attendez.*

```
func square(f float64) float64 { return f*f }
```

Une fonction peut retourner des valeurs multiples. Si c'est le cas, les types de retour sont entre parenthèses.

```
func MySqrt(f float64) (float64, bool) {
  if f >= 0 { return math.Sqrt(f), true }
  return 0, false
}
```

Google

L'identificateur blank

*Qu'arrive-t-il si vous vous souciez seulement de la première valeur retournée **MySqrt**? Toujours besoin de prendre en compte la seconde valeur?*

Solution: l'identificateur blank, _ (underscore).

```
// Ne pas se soucier de la 2ème valeur
// booléenne renvoyée par MySqrt.
val, _ = MySqrt(foo())
```

Google

Fonctions avec résultats variables

Les paramètres résultat sont des variables concrêtes que vous pouvez utiliser si vous les nommez :

```
func MySqrt(f float64) (v float64, ok bool) {
  if f >= 0 { v,ok = math.Sqrt(f), true }
  else { v,ok = 0,false }
  return v,ok
}
```

*Les variables résultat sont initialisées à zéro (0,*0.0*, false, etc. selon le type) :*

```
func MySqrt(f float64) (v float64, ok bool) {
  if f >= 0 { v,ok = math.Sqrt(f), true }
   return v,ok
}
```

Google

Le retour vide

Finallement, un retour sans expression retourne les valeurs existantes des variables de retour.

Deux versions de plus de MySqrt :

```
func MySqrt(f float64) (v float64, ok bool) {
  if f >= 0 { v,ok = math.Sqrt(f), true }
  return // must be explicit
}

func MySqrt(f float64) (v float64, ok bool) {
  if f < 0 { return } // error case
  return math.Sqrt(f),true
}
```

Google

Quid au sujet du zéro?

Toute la mémoire en Go est initialisée. Toutes les variables sont initialisées au moment de l'exécution de leur déclaration.

Sans une expression d'initialisation, la valeur "zéro" du type concerné est utilisée.

La boucle

```
for i := 0; i < 5; i++ {
  var v int
  fmt.Printf("%d ", v)
  v = 5
}
```

affichera 0 0 0 0 0.

La valeur zéro dépend du type : numeric 0; boolean false; empty string ""; nil pointer, map, slice, channel; zeroed struct, etc.

Google

Defer

L'instruction ***defer*** *exécute une fonction (ou méthode) quand la fonction appelante se termine. Les arguments sont évalués à l'endroit où le defer est déclaré ; l'appel de fonction survient au retour de la fonction.*

```
func data(fileName string) string {
  f := os.Open(fileName)
  defer f.Close()
  contents := io.ReadAll(f)
  return contents
}
```

Utile pour fermer des descripteurs de fichier, délocker des mutex, …

Google

Une invocation de fonction par **defer**

Chaque defer qui s'exécute met dans une file un appel de fonction à exécuter ultérieurement, dans l'ordre LIFO, ainsi

```
func f() {
  for i := 0; i < 5; i++ {
   defer fmt.Printf("%d ", i)
  }
}
```

affiche 4 3 2 1 0.

Vous pouvez fermer tous les descripteurs de fichier ou délocker tous les mutex à la fin de la fonction.

Google

Tracer avec un **defer**

```
func trace(s string) { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

func a() {
   trace("a")
   defer untrace("a")
   fmt.Println("in a")
}
func b() {
   trace("b")
   defer untrace("b")
   fmt.Println("in b")
   a()
}
func main() { b() }
```

Mais vous pouvez le faire plus proprement...

Google

Args sont évalués maintenant, **defer** plus tard

```
func trace(s string) string {
   fmt.Println("entering:", s)
   return s
}
func un(s string) {
   fmt.Println("leaving:", s)
}
func a() {
   defer un(trace("a"))
   fmt.Println("in a")
}
func b() {
   defer un(trace("b"))
   fmt.Println("in b")
   a()
}
func main() { b() }
```

Google

Les litéraux fonction

Comme en C, les fonctions ne peuvent pas être déclarées à l'intérieur des fonctions – mais les litéraux fonction peuvent être affectés à des variables.

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

Google

Les litéraux fonction sont des « *closures* »

Les litéraux fonction sont en fait des "closures".

```
func adder() (func(int) int) {
  var x int
  return func(delta int) int {
   x += delta
   return x
  }
}
f := adder()
fmt.Print(f(1))
fmt.Print(f(20))
fmt.Print(f(300))
```

affiche 1 21 321 - x s'accumulant dans f(delta)

Google

Construction d’un programme

Google

Packages

Un programme est construit comme un “package” qui peut utiliser des facilités d’autres packages.

Un programme Go est créé par l’assemblage d’un ensemble de “packages”.

Un package peut être construit à partir de plusieurs fichiers source.

Les noms dans les packages importés sont accédés via un “qualified identifier":

```
packagename.Itemname.
```

Google

Structure d’un fichier source

Chaque fichier source contient:

- Une clause package; ce nom est le nom par défaut utilisé par les packages qui importent

```
package fmt
```

- Un ensemble optionnel de déclarations d’import

```
import "fmt" // use default name
import myFmt "fmt" // use the name myFmt
```

- Des declarations zéro ou plus globales ou "package-level“.

Google

Un package simple fichier

```
package main // Ce fichier fait partie du package
             // “main”

import "fmt" // ce fichier utilise le package "fmt“

const hello = "Hello, Bonjour\n”

func main() {
   fmt.Print(hello)
}
```

Google

Main et main.main

Chaque programme Go contient un package appelé `main` *et sa fonction associée* `main`*tout comme en C, C++ ets la fonction de démarrage du programme.*

La fonction `main.main` *ne prends pas d'argument et ne retourne pas de valeur. Le programme sort immédiatement et avec succès quand* `main.main` *se termine.*

Google

Le package *OS*

Le package `os` fournit `Exit` et les accès aux entrées/sorties fichier, lignes de commande, etc…

```
// Une version de echo(1)
package main
import (
    "fmt"
    "os"
)
func main() {
    if len(os.Args) < 2 { // length of argument slice
            os.Exit(1)
    }
    for i := 1; i < len(os.Args); i++ {
            fmt.Printf("arg %d: %s\n", i, os.Args[i])
    }
} // fin == os.Exit(0)
```

Google

Visibilité globale et package

A l'intérieur d'un package, toutes les variables, fonctions, types et constantes globales sont visibles de tous les fichiers source du package.

Pour les clients (importers) du package, les noms doivent être en majuscule pour être visibles : : variables, fonctions, types, constantes, plus les méthodes ainsi que les champs de structure pour les variables et les types globaux.

```
const hello = "you smell"       // visibilité package
const Hello = "you smell nice" // visibilité globale
const _Bye = "stinko!"          // _ n'est pas majuscule
```

Très différent du C/C++ : pas d'extern, ni static, ni private, ni public.

Google

Initialisation

Deux façons d'initialiser les variables globales avant l'exécution de `main.main`*:*

1) Une déclaration globale avec un initialiseur

2) A l'intérieur d'une fonction init()

La dépendence des packages garantit un ordre d'excution correcte.

L'initialisation est toujours une seule tâche.

Google

Example d'initialisation

```
package transcendental
import "math"
var Pi float64
func init() {
   Pi = 4*math.Atan(1) // init function computes Pi
}
================
package main
import (
   "fmt"
   "transcendental"
)
var twoPi = 2*transcendental.Pi // decl calcule twoPi
func main() {
   fmt.Printf("2*Pi = %g\n", twoPi)
}
================
Output: 2*Pi = 6.283185307179586
```

Google

La construction des package et programme

Pour construire un programme, les packages et les fichiers à l'intérieurs de ceux-ci, doivent être compilés dans l'ordre correct. Les dépendances de package déterminent l'ordre dans lequel les packages à construire.

A l'intérieur d'un package, les fichiers source doivent être tous compilés ensemble. Le package est compilé comme une unité, et de manière conventionelle chaque répertoire contient un package. En ignorant les tests ,

```
cd mypackage
6g *.go
```

Habituellement, nous utilisons make ; des outils spécifiques à Go sont en cours de préparation.

Google

Construire le package « fmt »

```
% pwd
/Users/r/go/src/pkg/fmt
% ls
Makefile fmt_test.go format.go print.go # ...
% make # écrit à la main mais trivial
% ls
Makefile _go_.6 _obj fmt_test.go format.go print.go #
   ...
% make clean; make
...
```

Les objets sont placés dans le sous-répertoire _obj.

Les makefiles sont écrits en utilisant des *“helpers” appelés Make.pkg, etc… voir les sources.*

Google

Tests

Pour tester un package, écrire un ensemble de fichiers sources Go appartenant au même package ; donner les noms des fichiers de la forme :

`*_test.go`.

A l'intérieur de ces fichiers, les fonctions globales avec des noms commençant par `Test[^a-z]` *seront exécutées par l'outil de test* `gotest`. *Ces fonctions doivent une signature équivalente à*

```
func TestXxx(t *testing.T)
```

Le package testing fournit le support pour le log, le benchmarking, et le reporting d'erreur.

Google

Un exemple de test

Interesting pieces from fmt_test.go:

```
package fmt // package est fmt, pas main
import (
   "testing"
)
func TestFlagParser(t *testing.T) {
   var flagprinter flagPrinter
   for i := 0; i < len(flagtests); i++ {
     tt := flagtests[i]
     s := Sprintf(tt.in, &flagprinter)
     if s != tt.out {
            t.Errorf("Sprintf(%q, &flagprinter) => %q,"
            + " want %q", tt.in, s, tt.out)
     }
   }
}
```

Google

Test : gotest

```
% ls
Makefile fmt.a fmt_test.go format.go print.go # ...
% gotest # par defaut, fait tous les  *_test.go
PASS
wally=% gotest -v fmt_test.go
=== RUN fmt.TestFlagParser
--- PASS: fmt.TestFlagParser (0.00 seconds)
=== RUN fmt.TestArrayPrinter
--- PASS: fmt.TestArrayPrinter (0.00 seconds)
=== RUN fmt.TestFmtInterface
--- PASS: fmt.TestFmtInterface (0.00 seconds)
=== RUN fmt.TestStructPrinter
--- PASS: fmt.TestStructPrinter (0.00 seconds)
=== RUN fmt.TestSprintf
--- PASS: fmt.TestSprintf (0.00 seconds) # plus un peu plus
PASS
%
```

Google

Un exemple de benchmark

Benchmarks ont une signature

```
func BenchmarkXxxx(b *testing.B)
```

Et on boucle sur b.N; le package testing fait le reste .

Ci dessous un exemple de fichier fmt_test.go:

```
package fmt // package est fmt, pas main
import (
   "testing"
)
func BenchmarkSprintfInt(b *testing.B) {
   for i := 0; i < b.N; i++ {
     Sprintf("%d", 5)
   }
}
```

Google

Benchmarking : gotest

```
% gotest -bench="." # expression régulière
fmt_test.BenchmarkSprintfEmpty   5000000 310 ns/op
fmt_test.BenchmarkSprintfString  2000000 774 ns/op
fmt_test.BenchmarkSprintfInt     5000000 663 ns/op
fmt_test.BenchmarkSprintfIntInt  2000000 969 ns/op
...
%
```

Google

Bibliothèques de code

- Les bibliothèques sont juste des packages.
- L'ensemble des bibliothèques est modeste mais croissant.
- Des exemples:

Package	But	Exemples
fmt	I/O formatées	Printf, Scanf
os	Interface OS	Open, read, Write
strconv	Nombres<-> chaines	Atoi, Atof, Itoa
Io	I/O génériques	Copy, Pipe
Flag	Flags: --help etc…	Bool, String
Log	Log des évènements	Logger, Printf
Regexp	Expressions régulières	Compile, Match
Template	HTML, etc…	Parse, Execute
Bytes	Tableaux d'octets	Compare, Buffer

Un peut plus au sujet de fmt

Le package fmt contient des noms familiers en avec la poremière lettre en majuscule:

Printf – affiche sur la sortie standard

Sprintf – retourne une chaine de caractères

Fprintf – écrit sur os.Stderr etc. (demain)

Mais aussi

Print, Sprint, Fprint – pas de formatage

Println, Sprintln, Fprintln – pas de formatage, ajoute des espaces, final \n

```
fmt.Printf("%d %d %g\n", 1, 2, 3.5)
fmt.Print(1, " ", 2, " ", 3.5, "\n")
fmt.Println(1, 2, 3.5)
```

Chacune produit le même résultat : "1 2 3.5\n"

Google

Bibliothèque **documentation**

Le code source contient des commentaires.

La ligne de commande ou l'outil web extrait ces commentaires :

Link: http://golang.org/pkg/

Command:

```
% godoc fmt
% godoc fmt Printf
```

Google

Exercices

Google

Exercices : 1er jour

Initialiser l'environnement - voir
http://golang.org/doc/install.html

Vous connaissez tous les suites de fibonacci.
Ecrire un package pour les implémenter. Il devrait y avoir une fonction pour récupérer la valeur (vous ne connsaissez pas encore les struct ; pouvez trouver une façon de sauver l'état sans les globales?)
Mais à la place de l'addition, faire une opération fournie sous forme de fonction par l'utilisateur. Entiers, Flottants? Chaines de caractères? C'est à votre convenance!

Ecrire une petit test gotest pour votre package.

Google

Prochaine leçon

- *Les types composite*
- *Les méthodes*
- *Les interfaces*

Google

A demain!

Google