

Le langage Go

1^{ère} partie

Rob Pike

r@google.com

Traduction en français

xavier.mehaut@gmail.com

(Version de Juin 2011)

Créé par qui?

Travail effectué par une petite équipe chez Google, à laquelle se sont adjoints beaucoup de contributeurs du monde entier.

Contact:

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

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 en lui-même.

Aperçu du cours d'aujourd'hui

Motivations

Les bases

Du simple et de l'habituel

*Les « packages » et la construction d'un
programme*

Motivations



Pourquoi un nouveau langage?

Dans le monde qui est le nôtre , les langages du moment ne nous sont pas d'une aide suffisante:

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

Pour positiver

*Notre but est rendre la programmation à nouveau **fun***

- *La facilité d'un langage d'un dynamique ainsi que la sûreté de fonctionnement d'un système typé statiquement*
- *Compilation en langage machine pour une exécution plus rapide*
- *Run-time temps réel supportant un ramasse-miettes et la concurrence*
- *Léger, système de type flexible*
- *Possédant des méthodes comme dans un langage objet mais pas de manière conventionnelle*

Ressources

Pour plus d'information, voir la documentation à l'adresse suivante :

<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...*

Status des divers compilateurs

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

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

gccgo (*Ian Taylor*)

- Architecture plus commune
- Génère 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 continuellement.

Les bases



Codons

```
package main
```

```
import "fmt"
```

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

Bases du langage

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

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

Les deux prochaines parties auront plus de matière à amusement, mais nous devons poser en premier lieu les bases avant d'aller plus loin.

Structure lexicale

Traditionnelle avec quelques touches de modernité

Le code source est en UTF-8.

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

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

Commentaires:

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

Littéraux

Similaires au C mais les nombres ne requièrent aucun signe, ni marque de taille :

23

0x0FF

1.234e7

Similaires au C, mais toujours en Unicode/UTF-8.

De même, \xNN toujours 2 digits; \012 toujours 3; les deux sont des octets:

"Hello, world\n"

"\xFF" // 1 octet

"\u00FF" // 1 caractère Unicode, 2 octets en UTF-8

Chaînes de caractères:

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

Aperçu de la syntaxe

*Globalement similaire au C avec des déclarations de types inversés ,
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++ { ... }
```

NB: pas de parenthèses, mais des accolades requises

Point virgule

Les points-virgules 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.*
- NB: 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- virgule en dehors des clauses `for` et `if`

Les types numériques

Les types numériques sont des types natifs et semblent familiers:

INT	UINT		
INT8	UINT8=OCTET		
INT16	UNINT16		
INT32	UNINT32	FLOAT32	COMPLEX64
INT64	UNINT64	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 un **int32** même sur une machine 32 bits!*

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

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.

String

Le type natif `string` représente un tableau d'octets immuable, 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`.

Immuable, tout comme les `ints`. On peut réaffecter des variables mais pas éditer leur contenu.

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

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

Expressions

Globalement comme les opérateurs C

Ordre de précedence	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.

Expressions Go vs. Expressions C

Surprenant pour le développeur C:

Moins de niveaux de précédence (devrait être plus 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., requierent 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.)

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" // concaténation
// pas comme en C "a" "b"

Conversions numériques

Transformer 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)

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
0xFEEDBEEEEEEEEEEEEEEEEEEF // hexadécimal
1 << 100
```

*Il existe également des nombres flottants ou entiers idéaux;
La syntaxe des littéraux détermine le type :*

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


Expressions constantes

Les constantes à point flottant et entières peuvent être combinées à volonté, avec le type résultant qui est déterminé par le types des 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 = 0.69314718055994530941723212145817656807
const Log2E = 1/Ln2 // réciproque précis
```

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

Conséquence des nombre idéaux

Le langage permet l'utilisation de constantes sans conversion explicite si la valeur peut être convertie dans le type récepteur(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.
Exemple: ^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
```

Déclarations

Les déclarations sont introduites par un mot clef (var, const, type, func) et sont inversées en comparaison du 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.

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"
```

Var commonalisé

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`.

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 nom/type vue précédemment par rapport au C)

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

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

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

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  
)
```

Iota

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

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

Raccourcis: 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  
)
```


Types

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

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

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

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

Nous reviendrons sur les fonctions également un peu plus tard.

New

*La fonction native `new` alloue de la mémoire. La syntaxe est similaire à celle d'une fonction, avec des types comme arguments. 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.*

Affectations

L'affectation est simple et habituelle:

```
a = b
```

Mais les affectations multiples fonctionnent également:

```
x, y, z = f1(), f2(), f3()
```

```
a, b = b, a // swap
```

Les fonctions peuvent retourner des valeurs multiples :

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

Les structures de contrôles

Similaire au C, mais différents sur certains points.

*Go possède des **if**, **for** et **switch** (plus un encore 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.*

Forme des structures de contrôle

*Les instructions **if** et **switch** seront décrites dans les transparents suivants . Ils peuvent posséder une ou deux expressions.*

*La boucle **for** peut posséder une ou trois expressions :
un seul élément correspond au **while** du C:*

```
for a {}
```

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

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

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

If

La forme de base est familière :

```
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**.*

For

La forme de base est classique:

```
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  
{ ... }
```

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  
}
```


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]; { ... }
```

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**!*

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 vous y 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  
}
```

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())
```

Fonctions avec résultats variables

Les paramètres résultat sont des variables concrètes 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  
}
```

Le retour vide

Finally, a return without expression returns the values of the return variables.

Two versions of `MySqrt` :

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

```
func MySqrt(f float64) (v float64, ok bool) {  
    if f < 0 { return } // cas d'erreur  
    return math.Sqrt(f), true  
}
```

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.*

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, ...

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.

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...



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() }
```

Les littéraux fonction

Comme en C, les fonctions ne peuvent pas être déclarées à l'intérieur des fonctions – mais les littéraux 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)  
    }  
}
```

Les littéraux fonction sont des « *closures* »

Les littéraux 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)

Construction d'un programme



Packages

Un programme est construit comme un “package” qui peut utiliser les 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.

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"           // utilise le nom par  
                        // défaut
```

```
import myFmt "fmt" // utilise le nom myFmt
```

- *Des déclarations zéro ou plus globales ou "package-level".*

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)
}
```

Main et main.main

Chaque programme Go contient un package appelé `main` et sa fonction associée `main` tout comme en C, C++ est 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.

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 { //longueur des arguments de la 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)
```

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.

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 dépendance des packages garantit un ordre d'exécution correcte.

L'initialisation est toujours une seule tâche.

Exemple d'initialisation

```
package transcendental
import "math"
var Pi float64
func init() {
    Pi = 4*math.Atan(1) // la fct init calcule 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*



La construction des package et programme

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

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.

Tests

Pour tester un package, écrire un ensemble de fichiers sources Go appartenant au même package ; donner des 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 avoir 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.

Un exemple de test

Quelques morceaux de code intéressants venant de `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)
        }
    }
}
```

Test : gotest

```
% ls
Makefile fmt.a fmt_test.go format.go print.go # ...
% gotest # par default, 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
%
```

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)
    }
}
```

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
...
%
```

Bibliothèques de code

- Les bibliothèques sont juste des packages.
- L'ensemble des bibliothèques est modeste mais croissant fortement.
- 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
Regex	Expressions régulières	Compile, Match
Template	HTML, etc...	Parse, Execute
Bytes	Tableaux d'octets	Compare, Buffer

Un peu plus au sujet de `fmt`

Le package `fmt` contient des noms de fonction familiers en avec la première lettre en majuscule:

`Printf` - affiche sur la sortie standard

`Sprintf` - retourne une chaîne 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)
```

Chacun alternative produit le même résultat : "1 2 3.5\n"

Bibliothèque **documentation**

Le code source contient des commentaires.

Un utilitaire en ligne de commande ou l'outil web mis à disposition extrait ces commentaires :

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

Commandes:

```
% godoc fmt
```

```
% godoc fmt Printf
```


Exercices



Exercices : 1^{er} 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 connaissez pas encore les `struct` ; pouvez trouver une façon de sauver l'état sans les globales?) Mais à la place de l'addition, créer une opération écrite 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.

Prochaine leçon

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

A domain!

