

# Le langage Go

## 3ème partie

*Rob Pike*

[r@google.com](mailto:r@google.com)

Traduction en français, adaptation

[xavier.mehaut@gmail.com](mailto:xavier.mehaut@gmail.com)

*(Version de Juin 2011)*

# Sommaire de la journée

*Exercices*

*des questions?*

*Concurrence et communication*

*goroutines*

*channels*

*problématiques de la concurrence*

# Exercice

Des questions?

Pour un serveur HTTP trivial avec de multiples générateurs, voir

<http://golang.org/src/pkg/http/triv.go>

# Concurrence et communication :

## Les goroutines



# Les goroutines

*Terminologie:*

*Il existe plusieurs termes pour “des choses qui fonctionnent de manière concurrente” – processus, tâche, coroutine, thread POSIX, thread NPTL, processus léger, ... - mais leur signification à toutes est légèrement différente.*

*Aucune ne correspond exactement à comment Go traite la concurrence.*

*Ainsi nous allons introduire un nouveau terme : les **goroutines***

# définition

*Une goroutine est une fonction ou une méthode Go s'exécutant de manière concurrente dans le même espace d'adressage que les autres goroutines. Un programme qui tourne consiste donc en une ou plusieurs goroutines.*

*Ce n'est pas identique à un thread, coroutine, processus, ....  
C'est une goroutine.*

*NB : la concurrence et le parallélisme sont deux concepts différents.*

# Démarrer une goroutine

*Invoke a function or method and say go:*

```
func IsReady(what string, minutes int64) {  
    time.Sleep(minutes * 60*1e9) // Unités en nanosecs.  
    fmt.Println(what, "est prêt")  
}
```

```
go IsReady("thé", 6)  
go IsReady("café", 2)  
fmt.Println("J'attends...")
```

*affiche à l'écran:*

```
J'attends... (right away)  
Le café est prêt (2 minutes plus tard)  
Le thé est prêt tea (6 minutes plus tard)
```

# Quelques faits simples

*Goroutines ne coûtent pas grand chose.*

*Goroutines exit by returning from their top-level function, or just falling off the end..*

*Goroutines ne peuvent s'exécuter de manière concurrente que sur un seul processeur en partageant la mémoire.*

*Vous n'avez pas à vous occuper de la taille de la pile!*



# Les piles (stacks)

*En gccgo, au moins jusqu'à présent, les goroutines sont des pthreads. En 6g, ce sont des threads multiplexés, qui sont ainsi bien plus légers.*

*Dans les deux mondes, les piles sont petites (quelques kO) et croissent selon le besoin. Ainsi les goroutines utilisent peu de mémoire, vous pouvez en avoir de nombreuses, et elles peuvent avoir dynamiquement des piles énormes en taille.*

*Le programmeur ne devrait pas se préoccuper de la taille de la pile, c'est Go qui le fait à sa place.*

# L'ordonnancement des tâches (scheduling)

*Les Goroutines sont multiplexées. Quand une goroutine exécute un appel système bloquant, les autres goroutines ne sont pas bloquées.*

*Il y a des plans de faire de même pour les goroutines liées au CPU à un certain degré, mais pour l'instant, si vous voulez du parallélisme niveau utilisateur en 6g\*, vous devez initialiser la variable shell*

*GOMAXPROCS ou appeler `runtime.GOMAXPROCS(n)`.*

GOMAXPROCS indique au scheduler d'exécution combien de goroutines s'exécutant dans l'espace utilisateur (userspace) peuvent tourner à la fois, idéalement sur différents coeurs.

*\*gccgo utilise toujours un thread par goroutine.*

# Concurrence et communication: les canaux (channels)



# Les channels en Go

*A moins que two goroutines ne communiquent, elles ne peuvent se coordonner.*

*Go possède un type spécifique appelé channel (canal) qui fournit des mécanismes de communication et de synchronisation.*

*Il existe également des structures de contrôle spéciales qui sont basées sur les channels pour rendre la programmation concurrente plus simple.*

*NB: J'ai donné une conf en 2006 sur ce sujet mais en un autre langage*  
<http://www.youtube.com/watch?v=HmxnCEa8Ctw>

# Le type channel

*Dans sa plus simple forme, le type ressemble à ceci :*

```
chan elementType
```

*Avec une valeur de ce type, vous pouvez envoyer et recevoir des items de `elementType`.*

*Les channels sont des types de référence, ce qui signifie que si vous affectez une variable `chan` à une autre variable, les deux variables accèdent au même canal de communication. Ceci signifie que vous utilisez un `make` pour en allouer un.*

```
var c = make(chan int)
```

# L'opérateur de communication : <-

*La flèche pointe dans la direction du flow de données.*

*En tant qu'opérateur binaire, <- envoie la valeur vers la droite du canal sur la gauche :*

```
c := make(chan int)
c <- 1 // envoie 1 sur c
```

*Comme opérateur unaire, <- reçoit des données d'un canal (channel) :*

```
v = <-c // reçoit une valeur de c, l'affecte
<-c // reçoit une valeur, sans la stocker
i := <-c // reçoit une valeur, initialise i
```

# Sémantique

*Par défaut, la communication est synchrone (nous reparlerons des communications asynchrones plus tard)*

*Ceci signifie donc:*

- 1. Une opération envoi sur un channel bloque l'exécution jusqu'au moment où le récepteur est de nouveau disponible sur ce channel*
- 2. Une opération de réception est bloquée jusqu'au moment où l'envoyeur est disponible sur ce même channel*

*La communication est par conséquent une forme de synchronisation : deux goroutines échangent des données à travers un canal et synchronise de facto les deux tâches au moment de la communication.*

# Injectons des données

```
func pump(ch chan int) {  
    for i := 0; ; i++ { ch <- i }  
}  
ch1 := make(chan int)  
go pump(ch1)  
fmt.Println(<-ch1) // affiche 0
```

*Maintenant nous démarrons une boucle de réception*

```
func suck(ch chan int) {  
    for { fmt.Println(<-ch) }  
}  
go suck(ch1) // des tonnes de chiffres apparaissent
```



# Fonctions retournant un channel

*Dans l'exemple précédent, `pump` était comme un générateur de valeurs. Packageons la dans une fonction retournant un channel de valeurs.*

```
func pump() chan int {  
    ch := make(chan int)  
    go func() {  
        for i := 0; ; i++ { ch <- i }  
    }()  
    return ch  
}  
stream := pump()  
fmt.Println(<-stream) // prints 0
```

*“une fonction retournant un channel” est un idiome important.*

# Des fonctions *channel* partout

*J'évite de répéter des exemples illustres que vous pouvez trouver partout. Ici un ensemble de liens intéressants :*

- 1. Les nombres premiers; dans la spécification du langage et également dans ce tutoriel*
- 2. Un papier de Doug McIlroy :*

<http://plan9.bell-labs.com/who/rsc/thread/squint.pdf>

*Une version Go de ce programme fameux est disponible dans les suites de tests :*

<http://golang.org/test/chan/powser1.go>

# Range et channels

*La clause `range` des boucles `for` acceptent les channels comme opérandes, dans quel cas le `for` boucle sur toutes la valeurs recues du channel. Nous pouvons réécrire l'exemple ainsi:*

```
func suck(ch chan int) {  
    go func() {  
        for v := range ch { fmt.Println(v) }  
    }()  
}  
suck(pump()) // ne bloque plus maintenant
```

# Fermer un channel

*Comment est-ce que le `range` sait quand le channel arrête d'envoyer des données? L'émetteur envoie la commande native `close`:*

```
close(ch)
```

*Le récepteur teste si l'émetteur a fermé le canal en utilisant le "comma ok":*

```
val, ok := <-ch
```

*Le résultat est `(value, true)` tant qu'il ya des valeurs disponibles ;  
Une fois que le canal est fermé et vidé, le résultat est `(zero, false)`.*

# Range sur un channel à la main

*Un for range sur un channel tel que :*

```
for value := range <-ch {  
    use(value)  
}
```

*est équivalent à :*

```
for {  
    value, ok := <-ch  
    if !ok {  
        break  
    }  
    use(value)  
}
```

# Close

*Les point clefs:*

- Seulement l'émetteur doit appeler close.*
- Seulement le récepteur peut demander si un channel a été fermé*
- Peut seulement demander tant qu'il est en réception de valeurs (évite les courses).*

*Appeler close seulement quand c'est nécessaire pour signaler au récepteur qu'aucune autre valeur n'arrivera .*

*La plupart du temps, close n'est pas nécessaire ; ce n'est pas analogue à la fermeture d'un fichier.*

*Channels sont récupérés par le ramasse-miette quoiqu'il en soit.*

# Bidirectionnalité des channels

*Dans sa plus simple expression, un channel est non bufferisé (synchronous) et peut être utilisé pour émettre ou recevoir une donnée.*

*Un type channel peut être annoté pour spécifier qu'il est en lecture ou en écriture seule :*

```
var recvOnly <-chan int  
var sendOnly chan<- int
```

# Bidirectionnalité des channels (2)

*Tous les channels sont créés bidirectionnels, mais nous pouvons les affecter à des variables unidirectionnelles. Utile par exemple dans les fonctions pour la sûreté de fonctionnement :*

```
func sink(ch <-chan int) {  
    for { <-ch }  
}  
func source(ch chan<- int) {  
    for { ch <- 1 }  
}
```

```
c := make(chan int) // bidirectionnel  
go source(c)  
go sink(c)
```



# Channels synchrones

*Les channels synchrones sont non bufferisés. L'envoi ne s'effectue pas tant que le récepteur n'a pas accepté la valeur émise.*

```
c := make(chan int)
go func() {
    time.Sleep(60*1e9)
    x := <-c
    fmt.Println("reçu", x)
}()

fmt.Println("en train d'émettre", 10)
c <- 10
fmt.Println("envoyé", 10)
```

*sortie:*

```
en train d'émettre 10 (arrive immédiatement)
envoyé 10 (60s plus tard, ces deux lignes apparaissent)
reçu 10
```

# Channels asynchrones

*Un channel bufferisé et asynchrone est créé en indiquant à `make` le nombre d'éléments dans le buffer (tampon).*

```
c := make(chan int, 50)
go func() {
    time.Sleep(60*1e9)
    x := <-c
    fmt.Println("reçu", x)
}()

fmt.Println("en train d'émettre", 10)
c <- 10
fmt.Println("envoyé", 10)
```

**Output:**

```
en train d'émettre 10 (arrive immédiatement)
envoyé 10 (maintenant)
reçu 10 (60s plus tard)
```

# Le buffer ne fait pas partie du type

*Notez que la taille du buffer, ou même son existence, ne fait pas partie du type du channel, seulement de la valeur concrète.*

*Ce code est par conséquent légal, quoique dangereux :*

```
buf = make(chan int, 1)
unbuf = make(chan int)
buf = unbuf
unbuf = buf
```

*Mettre en tampon est une propriété de la valeur, pas du type de la valeur.*

# Select

Select est une structure de contrôle en Go analogue à l'instruction *switch*. *Chaque cas doit former une communication, soit recevoir, soit émettre une donnée.*

```
ci, cs := make(chan int), make(chan string)
select {
case v := <-ci:
fmt.Printf("received %d from ci\n", v)
case v := <-cs:
fmt.Printf("received %s from cs\n", v)
}
```

Select *exécute un des branchements au hasard. Si aucun cas n'est exécutable, il bloque l'exécution jusqu'à ce qu'un cas valable soit déclenché. Une clause default est toujours exécutable.*

# Sémantique du Select

*Rapide résumé:*

- *Chaque cas doit représenter une communication*
- *Tous les branchements du select sont exécutés*
- *Toutes les expressions à envoyer sont évaluées*
- *Si une communication peut être entreprise, elle l'est; sinon elle est ignorée*
- *Si de multiples cas son possibles, une est sélectionnée au hasard. Les autre ne s'exécutent pas.*
- *Autrement:*
  - *S'il existe une clause default, elle s'exécute*
  - *S'il n'y a pas de default, le select bloque jusqu'à ce qu'une communication soit démarrée. Il n'y a pas réévaluation des channels ou des valeurs.*

# Un générateur de bits aléatoire

*Idiot mais illustratif :*

```
c := make(chan int)
go func() {
    for {
        fmt.Println(<-c)
    }
}()
for {
    select {
        case c <- 0: //pas d'instruction, pas d'échappement
        case c <- 1:
    }
}
```

*affiche 0 1 1 0 0 1 1 1 0 1 ...*

# Tester la communication

*Est-ce qu'une communication peut s'effectuer sans blocage?*

Select *avec un default* peut nous le dire:

```
select {  
  case v := <-ch:  
    fmt.Println("reçu", v)  
  default:  
    fmt.Println("ch n'est pas prêt à recevoir")  
}
```

*La clause par défaut s'exécute si aucun autre cas ne convient, ainsi c'est l'idiome pour une réception non bloquante ; un envoi non bloquant est une variante évidente;*

# Timeout

*Est-ce qu'une communication peut être un succès après un interval de temps donné?  
Le package `time` contient la fonction `After` pour cela:*

```
func After(ns int64) <-chan int64
```

*Il renvoie une valeur (le temps courant) sur le channel retourné après un temps spécifié..*

*L'utiliser dans `select` pour implémenter le timeout:*

```
select {
    case v := <-ch:
        fmt.Println("reçu", v)
    case <-time.After(30*1e9):
        fmt.Println("timed out après 30 secondes")
}
```



# Multiplexing

*Les channels sont des valeurs de première classe, ce qui signifie qu'elles peuvent elles-mêmes être envoyées sur des channels comme toute autre valeur.*

*Cette propriété rend très simple l'écriture d'un multiplexeur puisque le client peut fournir, à côté de sa requête, le channel sur lequel répondre.*

```
chanOfChans := make(chan chan int)
```

*Ou plus typiquement*

```
type Reply struct { ... }
type Request struct {
    arg1, arg2 someType
    replyc chan *Reply
}
```

# Serveur de multiplexage

```
type request struct {  
    a, b int  
    replyc chan int  
}
```

```
Type binOp func(a, b int) int
```

```
func run(op binOp, req *request) {  
    req.replyc <- op(req.a, req.b)  
}
```

```
func server(op binOp, service <-chan *request) {  
    for {  
        req := <-service // les requêtes arrivent ici  
        go run(op, req) // n'attend pas op  
    }  
}
```

# Démarrer le serveur

*Utiliser une “fonction retournant un channel” pour créer un channel vers un nouveau serveur.*

```
func startServer(op binOp) chan<- *request {  
    service := make(chan *request)  
    go server(op, req)  
    return service  
}
```

```
adderChan := startServer(  
    func(a, b int) int { return a + b }  
)
```

# Le client

*Un exemple similaire est expliqué plus en détail dans le tutoriel, mais il y a une variante :*

```
func (r *request) String() string {  
    return fmt.Sprintf("%d+%d=%d",  
        r.a, r.b, <-r.replyc)  
}  
req1 := &request{7, 8, make(chan int)}  
req2 := &request{17, 18, make(chan int)}
```

*Les requêtes sont prêtes ; on les envoie*

```
adderChan <- req1  
adderChan <- req2
```

*On peut récupérer les résultats dans n'importe quel ordre ; r.replyc demuxes:*

```
fmt.Println(req2, req1)
```

# Démontage

*Dans l'exemple multiplexé, le serveur tourne à jamais.*

*Pour l'éteindre proprement, il faut notifier le channel. Ce serveur possède la même fonctionnalité mais avec un channel.*

```
func server(op binOp, service <-chan *request,
  quit <-chan bool) {
  for {
    select {
      case req := <-service:
        go run(op, req) // don't wait for it
      case <-quit:
        return
    }
  }
}
```

# Démarrer le serveur

*Le reste du code est quasiment le même, avec un channel supplémentaire :*

```
func startServer(op binOp) (service chan<-
    *request, quit chan<- bool) {
    service = make(chan *request)
    quit = make(chan bool)
    go server(op, service, quit)
    return service, quit
}

adderChan, quitChan := startServer(
    func(a, b int) int { return a + b }
)
```

# Arrêter le client

*Le client n'est pas affecté jusqu'à ce qu'il soit à arrêter le serveur:*

```
req1 := &request{7, 8, make(chan int)}  
req2 := &request{17, 18, make(chan int)}  
adderChan <- req1  
adderChan <- req2  
fmt.Println(req2, req1)
```

*Tout est accompli, on signale au serveur de sortir :*

```
quitChan <- true
```

# Chaînage

```
package main
import ("flag"; "fmt")
var nGoroutine = flag.Int("n", 100000, "combien")
func f(left, right chan int) { left <- 1 + <-right }
func main() {
    flag.Parse()
    leftmost := make(chan int)
    var left, right chan int = nil, leftmost
    for i := 0; i < *nGoroutine; i++ {
        left, right = right, make(chan int)
        go f(left, right)
    }
    right <- 0 // bang!
    x := <-leftmost // attend la terminaison
    fmt.Println(x) // 100000
}
```



# Exemple : channel comme cache de buffer

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)
func server() {
    for {
        b := <-serverChan           // attend qqe chose à faire
        process(b)                  // Met la requête dans le buffer
        select {
            case freeList <- b:      // reutilise le buffer si place
            default:                 // autrement ne fait rien.
        }
    }
}
func client() {
    for {
        var b *Buffer
        select {
            case b = <-freeList:     // en prend un si disponible
            default: b = new(Buffer) // alloue si pas
        }
        load(b)                     // lit la prochaine requête dans b
        serverChan <- b             // envoie une requête vers le serveur
    }
}
```

# Concurrence



# Problématiques de la concurrence

*Beaucoup bien sûr, mais Go essaye de prendre soin d'elles. Les opérations d'envoi et de réception sont en Go atomiques. L'instruction Select est définie de manière très précise et bien implémentée, ...*

*Mais les goroutines s'exécutent dans un espace partagé, le réseau de communication peut très bien être dans un état d'étreinte fatale, les debuggers multi-threadés ne sont pas encore au point, etc...*

*Que faire alors?*

# Go vous donne des primitives

*Ne programmez pas de la même façon que vous programmeriez en c, C++ et même java.*

*Les channels vous offre et la synchronisation et la communication, et cela rend la programmation très puissante et rigoureuse si vous savez bien les utiliser.*

*La règle est la suivante:*

- *Ne communiquez pas par mémoire partagée*
- *A la place, partagez la mémoire en communiquant!*

*L'acte de communiquer garantit la synchronization!*

# Le modèle

*Par exemple, utiliser un channel pour envoyer des données à une goroutine serveur dédiée. Si seulement une goroutine à la fois possède un pointeur vers les données, il n'y a pas de problématiques de concurrence.*

*C'est le modèle que nous prônons pour programmer des serveurs, au moins. C'est la bonne vieille approche « une tâche thread par client », généralisée – et en utilisation depuis les années 80. Et ça marche très bien.*

*Ma conf précédente (mentionnée auparavant) décline cette idée plus en profondeur.*

# Le modèle mémoire

*Les détails qui tuent au sujet de la synchronisation et de la mémoire partagée sont décrits ici :*

[http://golang.org/doc/go\\_mem.html](http://golang.org/doc/go_mem.html)

*Mais vous aurez rarement besoin de les comprendre si vous suivez notre approche.*