LES CHANNELS DANS LE LANGAGE DE PROGRAMMATION GO

C'est quoi les channels ?

Dans ce tutoriel nous allons parler des channels. Les channels sont utilisés avec les goroutines pour envoyer des données (int, string, struct...) d'une goroutine et les recevoir dans une autre goroutine. C'est un moyen de connecter les différentes goroutines, c'est un moyen de communication et de synchronisation entre les goroutines. **La transmission des channels se fait qu'avec des goroutines**

Les channels

Déclarer un channel

Pour déclarer votre channel, vous utiliserez le mot-clé `make` avec le mot-clé `chan` suivit du type de donnée que vous souhaitez transiter.

```
ch := make(chan typeDeValeur)
```

Envoyer et récupérer un channel

Pour envoyer ou recevoir une valeur dans un channel, il faut utiliser l'opérateur `<-`.

Exemple :

```
package main

import "fmt"

func run(c chan string, name string) {
```

```
    c <- name // envoyer une valeur d'un channel
}

func main() {
    canal := make(chan string)
    go run(canal, "Hatim")
    fmt.Println(<-canal) // récupérer une valeur d'un channel
}
```

Résultat :

```
Hatim
```

Les channels sont bloquants

Les envois et les réceptions sur un channel est bloquant par défaut. Qu'est-ce que ça veut dire ? Lorsqu'une donnée est envoyée à un channel, le contrôle est bloqué dans l'instruction d'envoi jusqu'à ce qu'une autre goroutine lise depuis ce channel. De la même manière, lorsque des données sont lues sur un channel, la lecture est bloquée jusqu'à ce qu'une certaine goroutine écrit des données sur ce channel.

C'est cette propriété des channels qui permet aux goroutines de communiquer efficacement sans l'utilisation de verrous explicites ou de variables conditionnelles.

```
package main

import (
 "fmt"
 "time"
)

func run(ch chan string, name string) {
 time.Sleep(time.Second * 2)
 fmt.Println("fonction run() :", name)
 ch <- name
}

func main() {

 now := time.Now()

 ch := make(chan string)
```

```
 go run(ch, "channel 1")
 fmt.Println("fonction main() :", <-ch)

 go run(ch, "channel 2")
 fmt.Println("fonction main() :", <-ch)

 fmt.Println(time.Now().Sub(now))
}
```

Résultat :

```
fonction run() : channel 1
fonction main() : channel 1
fonction run() : channel 2
fonction main() : channel 2
4.00097503s
```

Nous avons lancé 2 fois la goroutine de la fonction `run()` avec une lecture sur le channel `ch`, nous pouvons remarquer que le temps d'exécution est de 4 secondes avec un intervalle de 2 secondes entre chaque goroutine, ce qui prouve que les channels sont bien bloquants.

Par contre ici on ne profite pas forcément de la puissance des goroutines car elles ne sont pas lancées en simultanée vu que le channel est placé entre chaque goroutine. Pour régler ce problème je vais changer l'ordre de lecture de nos channels.

```
package main

import (
    "fmt"
    "time"
)

func run(ch chan string, name string) {
    time.Sleep(time.Second * 2)
    fmt.Println("fonction run() :", name)
    ch <- name
}

func main() {
```

```
    now := time.Now()

    ch := make(chan string)

    go run(ch, "channel 1")
 go run(ch, "channel 2")

 // changement d'ordre de lecture de nos channels
    fmt.Println("fonction main() :", <-ch)
    fmt.Println("fonction main() :", <-ch)

    fmt.Println(time.Now().Sub(now))
}
```

Résultat :

```
fonction run() : channel 1
fonction main() : channel 1
fonction run() : channel 2
fonction main() : channel 2
2.000341006s
```

> **Information**
>
> La première goroutine qui aura écrit sur un channel sera la première à être lu dans notre programme.

deadlock

Par défaut, les channels sont dit **unbuffered**, ce qui signifie qu'ils n'accepteront pas de **récepteur** (chan<-) que s'il existe un **expéditeur** (<- chan) correspondant prêt à recevoir la valeur envoyée, l'inverse est aussi vrai.

Voila ce que j'entend par expéditeur et récepteur :

- **récepteur** : c'est le moment où on entre une valeur dans notre channel

- **expéditeur** : c'est le moment où on lit une valeur depuis notre channel

Dans cet exemple je vais créer un channel avec 5 récepteurs et 5 expéditeurs :

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg = sync.WaitGroup{}

func main() {

    now := time.Now()
    ch := make(chan int)

    // 5 expéditeurs
    for j := 0; j < 5; j++ {
        wg.Add(1)
        go func() {
            time.Sleep(time.Second * 2)
            i := <-ch
            fmt.Println(i)
            wg.Done()
        }()
    }

    // 5 récepteurs
    for j := 0; j < 5; j++ {
        wg.Add(1)
        go func() {
            time.Sleep(time.Second * 2)
            ch <- 50

            wg.Done()
        }()
    }

    wg.Wait()

    fmt.Println(time.Now().Sub(now))
}
```

Résultat :

```
50
50
50
50
50
2.000827906s
```

Ici nous avons autant de récepteurs que d'expéditeurs, donc aucune erreur dans notre programme, mais maintenant je vais rajouter un récepteur en plus.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg = sync.WaitGroup{}

func main() {

    now := time.Now()
    ch := make(chan int)

    // 5 expéditeur
    for j := 0; j < 5; j++ {
        wg.Add(1)
        go func() {
            time.Sleep(time.Second * 2)
            i := <-ch
            fmt.Println(i)
            wg.Done()
        }()
    }

    // 6 récepteurs
    for j := 0; j < 6; j++ {
        wg.Add(1)
        go func() {
            time.Sleep(time.Second * 2)
            ch <- 50

            wg.Done()
        }()
    }

    wg.Wait()
```

```
    fmt.Println(time.Now().Sub(now))
}
```

Erreur :

```
50
50
50
50
50
fatal error: all goroutines are asleep - deadlock!
```

D'après le résultat nous pouvons remarquer que les 5 premières goroutines se sont bien exécutées mais ce n'est pas le cas pour la dernière goroutine. Car cette dernière n'a aucun expéditeur d'où l'erreur `deadlock`.

buffered channels

Pour éviter le problème d'avant il est possible de rendre nos channels buffered, c'est-à-dire de posséder autant de récepteurs que la taille du buffer de notre channel, par exemple un channel avec une taille de buffer de 30 peut posséder 30 récepteurs.

Pour rendre notre channel buffered, il suffit d'indiquer la longueur de notre buffer comme second argument de notre canal.

```
ch := make(chan type-de-valeur, taille_du_buffer)
```

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg = sync.WaitGroup{}
```

```go
func main() {

    now := time.Now()
    const size int = 10
    ch := make(chan int, size) // channel avec un buffer de taille 10

    // 5 expéditeurs
    for j := 0; j < 5; j++ {
        wg.Add(1)
        go func() {
            time.Sleep(time.Second * 2)
            i := <-ch
            fmt.Println(i)
            wg.Done()
        }()
    }

    // 10 récepteurs
    for j := 0; j < size; j++ {
        wg.Add(1)
        go func() {
            time.Sleep(time.Second * 2)
            ch <- 50
            wg.Done()
        }()
    }

    wg.Wait()

    fmt.Println(time.Now().Sub(now))
}
```

Résultat :

```
50
50
50
50
50
2.00064941s
```

Itération dans un channel

Il est possible d'itérer sur un channel buffered en utilisant le mot clé `range`. Je vais volontairement provoquer une erreur en itérant sur une channel buffered qui possède moins de récepteurs que la taille du buffer.

```go
package main

    import (
    "fmt"
    "time"
)

func main() {

    ch := make(chan string, 2) // buffer de taille 2

    go func() {
        ch <- "test" // 1 seul récepteur
    }()

    for elem := range ch {
        fmt.Println(elem)
    }

}
```

Erreur :

```
test
fatal error: all goroutines are asleep - deadlock!
```

Alors, déjà on peut remarquer qu'on arrive à lire notre première valeur de notre channel, mais juste après nous avons un deadlock. L'erreur vient du fait qu'on est en train de lire sur un channel avec un buffer de taille 2, sauf que ne nous n'avons rentré qu'une seule valeur dans notre channel d'où l'erreur.

Pour nous prémunir de cette erreur, il faut indiquer à notre compilateur qu'il n'est pas nécessaire de lire la suite du buffer de notre channel, cela est possible avec la fonction `close()`.

```go
package main

import (
 "fmt"
 "time"
)
```

```
func main() {

 ch := make(chan string, 2) // buffer de taille 2

 go func() {
  defer close(ch) // on indique à notre compilateur qu'on a finit d'écrire sur le cha
  ch <- "test"
 }()

 for elem := range ch {
  fmt.Println(elem)
 }

}
```

Résultat :

```
test
```

Il existe aussi un autre moyen d'itérer sur un channel, mais avant de vous montrez le code il faut savoir qu'un **channel renvoie à la fois sa valeur mais aussi un booléen** qui vous indique si la valeur a été envoyée sur le channel.

```
package main

import (
 "fmt"
)

func main() {
 ch := make(chan string, 2) // buffer de taille 2

 go func() {
  defer close(ch) // on indique à notre compilateur qu'on a finit d'écrire sur le cha
  ch <- "test"
 }()

 for true {
  if elem, ok := <-ch; ok == true { // est ce que le chanel possède encore un récepte
   fmt.Println(elem, ok)
  } else {
   break
  }
 }
}
```

Résultat :

```
test
```