La concurrence en Go

Dmitry Vyukov
DVyukov@google.com

Traduction en français
xavier.mehaut @gmail.com

(Version de Aout 2011)

Modèle de programmation

Principalement CSP/π-calaculus (pas vraiment formalisé): *goroutines+channels*

Devise de la concurrence en Go:

“Ne pas communiquer par mémoire partagée ; à la place, Do not communicate by sharing memory; instead, partager la mémoire en communiquant"

+ mémoire partagée à part entière

Google

Goroutines

Penser **threads** (mais pas d'opération *join*)

```
go foo()

go logger.Printf("Hello, %s!", who)

go func() {
    logger.Printf("Hello, %s!", who)
    ...
}()
```

Google

Channels

Fondamentalement des queues FIFO bloquantes limitées en taille et typées.

```
//synchronous chan of ints
c := make(chan int)

//buffered chan of pointers to
  Request
c := make(chan *Request, 100)
```

Google

Channels : citoyens de première classe

Vous pouvez les passer comme des arguments de fonction, les stocker dans des containers, les passer via des channels, etc…

De plus, les channels ne sont pas liés à des goroutines. Plusieurs goroutines peuvent envoyer/recevoir des données d'un seul channel.

Google

Channels : entrées/sorties

```
func foo(c chan int) {
  c <- 0
  <-c
}


func bar(c <-chan int) {
  <-c
}


func baz(c chan<- int) {
  c <- 0
}
```

Google

Channels : fermeture

```
//producteur
func producer(c chan *Work) {
    defer close(c)
    for {
        work, ok := getWork()
        if !ok { return }
        c <- work
     }
}

// consommateur
for msg := range c {
    process(msg)
}
```

Google

Pourquoi pas de *join* dans les goroutines

Habituellement, on a besoin de retourner une valkeur de toute façon.

```
c := make(chan int, N)

// fork
for i := 0; i < N; i++ {
  go func() {
    result := ...
    c <- result
  }()
}

// join
sum := 0
for i := 0; i < N; i++ {
  sum += <-c
}
```

Google

Select

Select fait un choix pseudo-aléatoire duquel un ensemble de communications possibles procède :

```
select {
   case c1 <- foo:
   case m := <-c2:
     doSomething(m)
   case m := <-c3:
     doSomethingElse(m)
   default:
     doDefault()
}
```

Google

Select : send/recv non bloquants

```
reqChan := make(chan *Request, 100)

httpReq := parse()
select {
  case reqChan <- httpReq:
  default:
    reply(httpReq, 503)
}
```

Google

Select: timeouts

```
select {
   case c <- foo:
   case <-time.After(1e9):
}
```

Google

Exemple : le barbier

```
var sièges = make(chan Customer, 4)

func barbier() {
  for {
    c := <-sièges
    // raser c
  }
}
go barbier()

func (c Client) raser() {
  select {
   case sièges <- c:
   default:
  }
}
```

C’est aussi simple que cela!

Google

Exemple : *pooling (scrutation)* de ressources

Q: fonctionnalité de pooling générale
Tout le monde a dans son code quelque part un bon mécanisme de pooling pour un type d'interface donné? Ou juste quelque chose que quelqu'un pourrait adapter et abstraire? Je suis sûr que des drivers de base de données ont des mécanismes de ce genre, des idées?

C'est un peu effrayant non?

Google

Exemple : *pooling* de ressources (suite)

The solution is just

```
pool := make(chan *Resource, 100)

Put with [nonblocking] send.
Get with [nonblocking] recv.
```

Aussi simple que cela encore une fois!

Ah, ca marche comme un charme.
Go rend les choses simples et sympa, sans oublier la puissance!
Merci…

Google

Exemple : programmation orientée acteurs

```
type ReadReq struct {
  key string
  ack chan<- string
}

type WriteReq struct {
  key, val string
}

c := make(chan interface{})

go func() {
  m := make(map[string]string)
  for {
    switch r := (<-c).(type) {
    case ReadReq:
      r.ack <- m[r.key]
    case WriteReq:
      m[r.key] = r.val
    }
  }
}()
```

Exemple : programmation orientée acteurs (suite)

```
c <- WriteReq{"foo", "bar"}

ack := make(chan string)
c <- ReadReq{"foo", ack}
fmt.Printf("Got", <-ack)
```

Aussi simple que cela(*ter repetita*)!

Google

Exemple : pool de threads

[Cette page est laissée intentionnellement blanche]

Google

Pourquoi est-ce du *CSP(Hoare*) pur ne va pas?

Allocation de mémoire dans un style CSP:

```
ack1 := make(chan *byte)
Malloc <- AllocReq{10, ack1}


ack2 := make(chan *byte)
Malloc <- AllocReq{20, ack2}


obj1 := <-ack1

obj2 := <-ack2
```

WTF??1!

Google

Pourquoi est-ce du *CSP*pur ne va pas? (suite)

Certains codes ne sont pas différents!

```
var UidReq = make(chan chan uint64)
go func() {

  seq := uint64(0)
  for {
    c := <-UidReq
    c <- seq
    seq++
  }
}()

ack := make(chan uint64)
UidReq <- ack
uid := <-ack
```

Google

Pourquoi est-ce du *CSP* pur ne va pas? (suite 2)

“Les passage de message (*message passing*) est aisé à implémenter. Mais tout est transformé en programmation distribuée alors” (c) Joseph Seigh

- Surcharge aditionnelle
- Latence aditionnelle
- Complexité superflue (asynchronisme, réordination, …)
- Equilibrage de charge
- Contrôle de surcharge
- Difficulté à débugger

Google

La mémoire partagée à la rescousse!

```
var seq = uint64(0)
...
uid := atomic.AddUint64(&seq, 1)
```

Simple, rapide, pas de latence supplémentaire, pas de chausse-trappe, pas de surcharge.

Google

Les primitives de mémoire partagée

```
sync.Mutex
sync.RWMutex
sync.Cond
sync.Once
sync.WaitGroup

runtime.Semacquire/Semrelease

atomic.CompareAndSwap/Add/Load
```

Google

Mutexes

sync.Mutex est en fait un sémaphore binaire coopératif – pas de propriétaire, pas de récursivité.

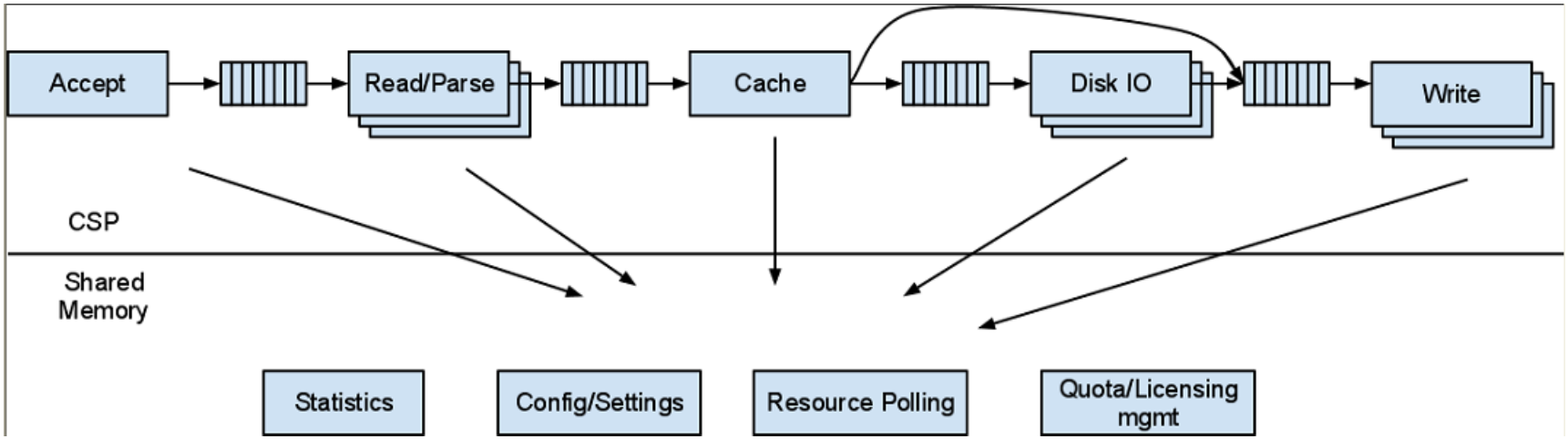
```
mtx.Lock()
go func() {
    ...
    mtx.Unlock()
  }
mtx.Lock()

func foo() {
  mtx.Lock()
  defer mtx.Unlock()
  ...
}
```

Google

Schéma général

90% de CSP au niveau le plus haut
+10% de mémoire partagée aux niveaux les plus bas



Google

Détecteur de course pour Go

Actuellement en développement dans Google MSK (pas d'obligation de livrer actuellement).

L'idée est de fournir un support général pour les outils d'analyse dynamique pour les compilateur.runtime/bibliothèques Go.

Et puis d'y attacher la tehcnologie toute-puissante *ThreadSanitizer*.

Si/quand livré sur la branche principale de gestion de conf, l'utilisateur aura juste à spécifier un seul flag de compilateur pour l'autoriser.

Google

Evolutivité (scalability)

Le but de Go est de supporter une concurrence grain fin évolutive.
Mais ce n'est pas [encore] le cas.

Il a été soumis environ CL liés à l'évolutivité. Toutes les primitives de synchronisation ont été réécrites (mutex, semaphore, once, etc), il reste a améliorer l'évolutivité des chan/select, de l'allocation de mémoire, la gestion de la pile (*stack mgmt*), etc.

"J'ai juste exécuté un benchmark réaliste fourni par quelqu'un utilisant mgo avec la release r59 du compialteur, et cela a pris environ 5 sur 20 , sans changement dans le code."

Toujours besoin d'améliorer l'ordonnanceur (scheduler) de goroutines.

Google

Merci!

Google