Pratique de la programmation en Go

Andrew Gerrand

adg@golang.org

Traduction en français

xavier.mehaut@gmail.com

Qu’est-ce que Go?

Go est un langage de programmation généraliste.

Les points forts de Go sont les suivants:

- Met l’accent sur la simplicité ; facile à apprendre
- Mémoire bien gérée ; facile à utiliser
- Code compilé rapide; *comparable au C*
- Support natif de la concurrence; code plus simple à écrire
- Typage statique
- Bibliothèque standard importante
- Auto-documenté (et bien documenté)
- Libre et Open Source (licence BSD)

Google

Cette présentation

Cette présentation couvre le développement complet d'une application web simple.
Il y a beaucoup à couvrir aussi irons-nous assez vite.

Si vous êtes débutants en Go, il se peut que certains points de syntaxe vous échappent. La chose la plus importante est d'avoir une idée de ce que le programme fait, plutôt que de comment il le fait exactement.

Ces transparents (en anglais) sont disponibles à l'adresse suivante :

http://wh3rd.net/practical-go/

Le code source complet et autres babioles sont disponibles sur le référentiel `git` suivant :

http://github.com/nf/goto

Twitter stuff:

```
#golang hashtag
@go_nuts (c'est moi!)
```

Google

Ecrivons un programme Go

goto: un raccourciceur (*shortener*) d'URL

Goto est un service web (HTTP) qui fait deux choses:

- Quand on lui fournit une adresse longue , *goto* retourne une version raccourcie de cette adresse:

  ```
  http://maps.google.com/maps?f=q&source=
  s_q&hl=en&geocode=&q=tokyo&sll=37.0625,
  95.677068&sspn=68.684234,65.566406&ie=U
  TF8&hq=&hnear=Tokyo,+Japan&t=h&z=9
  ```

  devient

  ```
  http://goo.gl/UrcGq
  ```

- Quand une requête courte est envoyée, *Goto* redirige l'utilisateur vers l'URL originale, la longue.

Google

Les structures de données

Goto associe (map) des URLs courtes à des URLs longues. Pour stocker cette association en mémoire, nous pouvons utiliser un dictionnaire.

Le type dictionnaire en Go vous permet d'associer des valeurs de n'importe quel type* vers des valeurs également de n'importe quel type.

Les dictionnaires doivent être initialisés avec la fonction native `make` :

```
m := make(map[int]string)
m[1] = "Un"

u := m[1]
// u == "Un"

v, present := m[2]
// v == "", present == false
```

(les clefs doivent pouvoir être triées avec ==, ie en anglais comparable)*

Google

Les structures de données (2)

Nous spécifions le type `URLStore` en tant que `map`, une structure de données de base de Go :

```
type URLStore map[string]string
m := make(URLStore)
```

Pour stocker l'association de `http://goto/a` vers `http://google.com/` dans `m`:

```
m["a"] = "http://google.com/"

url := m["a"] // url == http://google.com/
```

Attention : le type `map` en Go n'est pas sécurisé d'un point de vue accès concurrent, cad si plusieurs *threads* (tâches) tentent d'y accéder en même temps.

Goto acceptera plusieurs requêtes de manière concurrente, ainsi nous devrons rendre notre type `URLStore` sécurisé pour pouvoir y accéder à partir de plusieurs threads (tâches).

Google

Ajout d'un verrou

Pour protéger un dictionnaire d'une modification pendant une opération de lecture, nous devons ajouter un verrou (*lock*) à la structure de données.

En changeant la définition du type , nous pouvons transformer `URLStore` en un type structure à deux champs :

- Le dictionnaire
- Un `RWMutex` du package `sync`

```
import "sync"
type URLStore struct {
    urls map[string]string
    mu sync.RWMutex
}
```

Un `RWMutex` possède deux verrous : un pour le consommateur, un pour le producteur. Plusieurs clients peuvent prendre le verrou en lecture simultanément, mais un seul client peut prendre le verrou en écriture (à l'exclusion de tous les consommateurs).

Google

Les méthodes d'accès (get, set)

Nous devons maintenant intéragir avec `URLStore` à travers des méthodes d'accès (`Set` et `Get`).

La méthode `Get` prend le verrou en lecture avec `mu.RLock`, et retourne une string comme URL. Si la clef est présente dans le dictionnaire, la valeur zéro pour le type `string` (une chaîne vide) sera retournée.

```
func (s *URLStore) Get(key string) string {
  s.mu.RLock()
  url := s.urls[key]
  s.mu.RUnlock()
  return url
}
```

Google

Les méthodes d'accès (get, set) (2)

La méthode `Set` prend un verrou en écriture et remet à jour le dictionnaire avec l'URL. Si la clef est déjà présente, `Set` retourne un booléan `false` et le dictionnaire n'est pas mis à jour (plus tard, nous utiliserons ce comportement pour garantir que chaque URL possède une valeur unique)

```
func (s *URLStore) Set(key, url string) bool {
   s.mu.Lock()
   _, present := s.urls[key]
   if present {
     s.mu.Unlock()
     return false
   }
   s.urls[key] = url
   s.mu.Unlock()
   return true
}
```

Google

Defer : un aparté

Une instruction `defer` ajoute un appel de fonction à une pile. La pile des appels sauvegardés est traitée à la fin de l'exécution de la méthode englobante. Le `defer` est habituellement utilisé en vue de simplifier l'écriture des fonctions qui doivent exécuter des opérations de nettoyage.

Par exemple, cette fonction va afficher "Bonjour" et ensuite "monde" :

```
func foo() {
   defer fmt.Println("monde")
   fmt.Println("Bonjour")
}
```

Nous pouvons utiliser le `defer` afin de simplifier les méthodes `Get` et `Set`.
Il y a beaucoup plus à savoir au sujet du `defer`. Voir "Defer, Panic, and Recover" pour une discussion plus approfondie sur le sujet.

Google

Les méthodes d'accès (get, set) (3)

En utilisant `defer`, la méthode `Get` permet d'éviter l'utilisation de la variable `url` locale et renvoie la valeur `map` directement:

```
func (s *URLStore) Get(key string) string {
   s.mu.RLock()
   defer s.mu.RUnlock()
   return s.urls[key]
}
```

Et la logique pour le `Set` devient alors plus clair:

```
func (s *URLStore) Set(key, url string) bool {
   s.mu.Lock()
   defer s.mu.Unlock()
   _, present := s.urls[key]
   if present {
     return false
   }
   s.urls[key] = url
   return true
}
```

Google

Une fonction d'initialisation

La structure `URLStore` contient un champ `map`, ce dernier devant être initialisé avec `make` avant de pouvoir être utilisé.

```
type URLStore struct {
   urls map[string]string
   mu sync.RWMutex
}
```

Go ne possède pas de constructeurs. A la place, nous respectons la convention d'écrire une fonction nommée `NewXXX` qui renvoie une instance initialisée de ce type.

```
func NewURLStore() *URLStore {
   return &URLStore{
     urls: make(map[string]string),
   }
}
```

Google

Utilisation de URLStore

Création d'une instance:

```
s := NewURLStore()
```

Stockage d'une URL via sa clef :

```
if s.Set("a", "http://google.com") {
   // success
}
```

Récupération d'une URL par la clef :

```
if url := s.Get("a"); url != "" {
   // redirect to url
} else {
   // key not found
}
```

Google

Raccourcicement d'URLs

Nous possédons déjà la méthode `Get` pour récupérer les URLs. Créons maintenant la méthode `Put` qui prend une URL, la stocke grâce à sa clef correspondante, et qui renvoie la clef.

```
func (s *URLStore) Put(url string) string {
   for {
     key := genKey(s.Count())
    if s.Set(key, url) {
              return key
     }
   }
   panic("shouldn't get here")
}
func (s *URLStore) Count() int {
     s.mu.RLock()
     defer s.mu.RUnlock()
     return len(s.urls)
}
```

La fonction `genKey` prend un entier et renvoie une clef alphanumérique correspondante :

```
func genKey(n int) string {
   /* implementation omitted */
}
```

Google

Le serveur HTTP

Le package Go `http` fournit l'infrastructure nécessaire pour exécuter des requêtes HTTP.

```
package main

import ( "fmt" "http" )

func Hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, world!")
}

func main() {
  http.HandleFunc("/", Hello)
  http.ListenAndServe(":8080", nil)
}
```

Google

Gestionnaires HTTP (handlers)

Notre programme possèdera deux *handlers* HTTP :

- `Redirect`, qui redirige les requêtes URL courtes et
- `Add`, qui gère la réception de nouvelles URLs.

La fonction `HandleFunc` est utilisée pour les prendre en compte au moyen du package `http`.

```
func main() {
   http.HandleFunc("/", Redirect) http.HandleFunc("/add", Add)
   http.ListenAndServe(":8080", nil)
}
```

Les requêtes vers `/add` seront traitées par le gestionnaire `Add`.
Toutes les autres requêtes seront traitées par le gestionnaire `Redirect`.

Google

Gestionnaires HTTP : Add

La fonction `Add` lit les paramètres de l'url à partir de la requête HTTP, les met (`Put`) dans un magasin (`store`), et renvoie à l'utilisateur l'URL courte correspondante.

```
func Add(w http.ResponseWriter, r *http.Request) {
   url := r.FormValue("url")
   key := store.Put(url)
   fmt.Fprintf(w, "http://localhost:8080/%s", key)
}
```

Mais qu'est-ce que ce magasin? C'est une variable globale pointant vers une instance de `URLStore`:

```
var store = NewURLStore()
```

La ligne ci-dessus peut apparaître n'importe où au niveau le plus haut d'un fichier. Elle sera évaluée à l'initialisation du programme, avant que la fonction `main` ne soit elle-même appelée.

Google

Gestionnaires HTTP : Add (2)

Quid de l'interface utilisateur ? Modifions `Add` pour afficher un texte en HTML quand aucune URL n'est fournie :

```
func Add(w http.ResponseWriter, r *http.Request) {
   url := r.FormValue("url")
   if url == "" {
     fmt.Fprint(w, AddForm)
     return
   }
   key := store.Put(url)
   fmt.Fprintf(w, "http://localhost:8080/%s", key)
}

const AddForm = `
<form method="POST" action="/add">
   URL: <input type="text" name="url">
   <input type="submit" value="Add">
</form>
`
```

Google

Gestionnaires HTTP : Redirect

La fonction `Redirect` trouve la clef dans le chemin de la requête HTTP, récupère l'URL correspondante dans le magasin et envoie un redirect HTTP à l'utilisateur. Si l'URL n'est pas trouvée, une erreur 404 "`Not found`" est envoyée à la place.

```
func Redirect(w http.ResponseWriter, r *http.Request) {
    key := r.URL.Path[1:]
    url := store.Get(key)
    if url == "" {
      http.NotFound(w, r)
      return
    }
    http.Redirect(w, r, url, http.StatusFound)
}
```

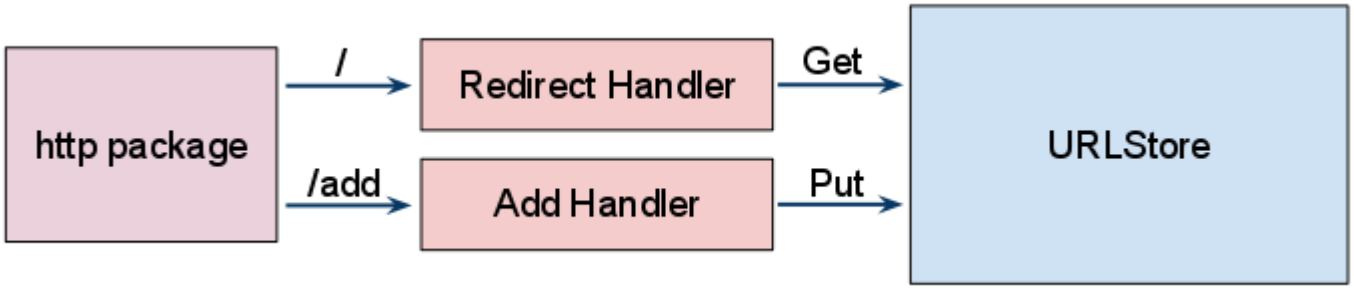
La clef est le chemin de la requête moins le premier caractère. Pour la requête "/foo", la clef sera "foo".

`http.NotFound` et `http.Redirect` des aides pour envoyer des réponses HTTP communes. La constante `http.StatusFound` représente le code HTTP 302 ("`Found`").

Google

Démonstration

Nous avons écrit moins de 100 lignes de code, et nous avons déjà une application web complète et fonctionnelle



Voir le code que nous avons écrit jusqu'ici :
https://github.com/nf/goto/tree/master/talk/code/0/

Google

Stockage persistent

Quand le processus `goto` se termine, les URLs raccourcies qui se trouvent en mémoire sont perdues.

Ceci n'est pas très utile en définitive.

Modifions quelque peu `URLStore` afin que les données soient écrites dans un fichier, et restaurées au démarrage.

Google

Les interfaces : un aparté

Les types `interface` en Go définissent un ensemble de méthodes. N'importe quel type qui implémente ces méthodes satisfait implicitement cette interface.

Une interface très fréquemment utilisée est l'interface `Writer`, spécifiée dans le package `io` :

```
type Writer interface {
  Write(p []byte) (n int, err os.Error)
}
```

De nombreux types, aussi bien dans la bibliothèque standard que dans du code Go, implémentent la méthode `Write` décrite ci-dessus, et peuvent ainsi être utilisés n'importe où du moment que `io.Writer` est attendu par l'utilisateur.

Google

Les interfaces : un aparté (2)

En fait, nous avons déjà utilisé `io.Writer` dans notre *handler (gestionnaire)* HTTP :

```
func Add(w http.ResponseWriter, r *http.Request) {
   ...
   fmt.Fprintf(w, "http://localhost:8080/%s", key)
}
```

La fonction `Fprintf` attend un `io.Writer` en tant que premier argument :

```
func Fprintf(w io.Writer, format string, a
...interface{}) (n int, error os.Error)
```

Parce que `http.ResponseWriter` implémente la méthode `Write`, `w` pouvons être passé à `Fprint` en tant que `io.Writer`.

Google

Stockage persistent : gob

Comment pouvons-nous stocker `URLStore` sur le disque dur?

La package Go `gob` s'occupe de sérialiser et désérialiser des structures de données Go. (Similaire au "pickle" de Pyhton ou à la sérialisation en Java)

La package `gob` possède deux fonctions `NewEncoder` et `NewDecoder` qui encapsulent des valeurs `io.Writer` et `io.Reader`.

Les objets résultants `Encoder` et `Decoder` fournissent les méthodes `Encode` et `Decode` pour écrire et lire des structures de données Go.

Google

Stockage persistent : URLStore

Créons un nouveau type de données, `record`, qui décrit comment une simple paire *clef/url* peut être stockée dans un fichier.

```
type record struct {
   Key,
   URL string
}
```

La méthode `save` écrit une clef donnée et son `url` sur le disque en tant que structure encodée sous forme de `gob` :

```
func (s *URLStore) save(key, url string) os.Error {
   e := gob.NewEncoder(s.file)
   return e.Encode(record{key, url})
}
```

Mais qu'est-ce que `s.file`?

Google

Stockage persistent : URLStore (2)

Le nouveau champ `file` de `URLStore` (de type `*os.File`) sera un *handle* vers un fichier ouvert qui peut être utilisé en lecture et en écriture:

```
Type URLStore struct {
   urls map[string]string
   mu sync.RWMutex
   file *os.File
}
```

La fonction `NewURLStore` prend maintenant un argument `filename`, ouvre le fichier, et enregistre une valeur `*os.File` dans le champ `file` :

```
func NewURLStore(filename string) *URLStore {
   s := &URLStore{urls: make(map[string]string)}
   f, err := os.Open(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
   if err != nil {
     log.Fatal("URLStore:", err)
   }
   s.file = f
   return s
 }
```

Google

Stockage persistent : URLStore (3)

La nouvelle méthode `load` va se positionner (`Seek`) au début du fichier, lire et enfin décoder chaque enregistrement (*record*), puis écrire les données dans le dictionnaire (`map`) en utilisant la méthode `Set` :

```
func (s *URLStore) load() os.Error {
   if _, err := s.file.Seek(0, 0); err != nil {
     return err
   }
   d := gob.NewDecoder(s.file)
   var err os.Error
   for err == nil {
     var r record
     if err = d.Decode(&r); err == nil {
              s.Set(r.Key, r.URL)
     }
   }
   if err == os.EOF {
     return nil
   }
   return err
}
```

Google

Stockage persistent : URLStore (4)

Nous ajoutons maintenant un appel à `load` à la fonction constructeur:

```
func NewURLStore(filename string) *URLStore {
  s := &URLStore{urls: make(map[string]string)}
  f, err := os.Open(filename,
  os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
  if err != nil {
   log.Fatal("URLStore:", err)
  }
  s.file = f if err := s.load(); err != nil {
   log.Println("URLStore:", err)
  }
  return s
}
```

Google

Stockage persistent : URLStore (5)

Et sauver (`save`) chaque nouvelle URL comme elle est avec `Put` :

```
func (s *URLStore) Put(url string) string {
  for {
   key := genKey(s.Count())
   if s.Set(key, url) {
         if err := s.save(key, url); err != nil {
               log.Println("URLStore:", err)
         }
         return key
   }
  }
  panic("shouldn't get here")
}
```

Google

Stockage persistent

Finalement, nous devons spécifier un nom de fichier au moment de l'instantiation de URLStore:

```
var store = NewURLStore("store.gob")
```

Démonstration

Récupérer le code sur git :

https://github.com/nf/goto/tree/master/talk/code/1/

Ces transparents (en anglais) sont disponibles à l'adresse suivante :

http://wh3rd.net/practical-go/

Google

Un point de discorde

Considérons la situation pathologique suivante :

- De nombreux clients tentent d'ajouter en même temps une URL
- Même si nous avons essayé de mettre à jour le dictionnaire de manière concurrente, plusieurs écritures sur disque peuvent s'effectuer simultanément. Dépendant des caractéristiques de votre OS, cela peut causer une corruption de la bdd
- Même si l'écriture ne provoque pas de collision, chaque client doit attendre que ses données soient écrites sur le disque avant que le Put soit exécuté
- Par conséquent, sur des systèmes fortement chargés côté I/O, les clients attendront plus longtemps que nécessaire que leur requête d'ajout soit prise en compte.

Pour remédier à ce problème, nous devrions découpler le `Put` du processus de sauvegarde.

Google

Les goroutines : un aparté

Une *goroutine* est une tâche légère (thread) gérée par le *runtime* Go.

Les goroutines sont lancées par l'instruction `go`. Ce code exécute `foo` et `far` de manière concurrente :

```
go foo()
bar()
```

La fonction `foo` s'éxécute dans une *goroutine* nouvellement créée, tandis que `bar` tourne dans la *goroutine* principale.

La mémoire est partagée entre les goroutines, comme dans la plupart des modèles multi-tâches.

Les *goroutines* sont bien mois coûteuses à créer que les *threads* (tâches) des systèmes d'exploitation sous-jacents!

Google

Les canaux (channels) : un aparté

Un *channel* est un conduit comme un pipeunix, à travers lequel vous pouvez envoyer des valeurs typées. Il fournit de nombreuses possibilités algorithmiques intéressantes.

Comme les dictionnaires (map), les channels doivent être initialisés avec make :

```
ch := make(chan int) // un channel d'entiers
```

La communication est exprimée en utilisant l'opérateur channel <- :

```
ch <- 7    // envoie l'entier 7 sur le channel
i := <-ch  // reçoit un entier à partir du channel
```

Les données transitent toujours dans la direction de la flèche.

Google

Les channels : un aparté (2)

Communiquer entre goroutines :

```
func sum(x, y int, c chan int) {
        c <- x + y
}

func main() {
        c := make(chan int)
        go sum(24, 18, c)
        fmt.Println(<-c)
}
```

Le channel envoie/reçoit typiquement des blocks jusqu'à ce qu l'autre côté soit prêt. Les channels peuvent être bufferisés ou non. Les envois vers un channel bufferisé ne bloquera l'exécution de la goroutine à moins que que le buffer ne soit plein.

Les channels bufferisés sont initialisés en spécifiant la taille du buffer (tampon) comme second argument de la fonction `make`:

```
ch := make(chan int, 10)
```

Voir les posts sur le blog "Share Memory by Communicating" et "Timing out, moving on" pour une discussion détaillée sur les goroutines et les channels.

Google

Sauvegarder de manière séparée

A la place de créer un appel de fonction pour sauvegarder chaque enregistrement (*record*) sur le disque, `Put` peut envoyer un enregistrement sur le channel bufferisé (communication asynchrone) :

```
type URLStore struct {
   urls map[string]string
   mu sync.RWMutex
   save chan record
}

func (s *URLStore) Put(url string) string {
   for {
     key := genKey(s.Count())
     if s.Set(key, url) {
               s.save <- record{key, url}
               return key
     }
   }
   panic("shouldn't get here")
}
```

Google

Sauvegarder de manière séparée (2)

De l'autre côté du channel `save`, nous devons avoir un récepteur.

Cette nouvelle méthode `saveLoop` s'exécutera dans une *goroutine* séparée, et recevra des valeurs de type *record* puis les sauvera dans un fichier .

```
func (s *URLStore) saveLoop(filename string) {
   f, err := os.Open(filename,os.O_WRONLY|os.O_CREATE|os.O_APPEND,0644)
   if err != nil {
     log.Fatal("URLStore:", err)
   }
   e := gob.NewEncoder(f)
   for {
     r := <-s.save
     if err := e.Encode(r); err != nil {
             log.Println("URLStore:", err)
     }
   }
}
```

Google

Sauvegarder de manière séparée (3)

Nous avons besoin de modifier la fonction `NewURLStore` afin de lancer la *goroutine* `saveLoop` (et supprimer le code d'ouverture de fichier désormais non caduque):

```
const saveQueueLength = 1000

func NewURLStore(filename string) *URLStore {
    s := &URLStore{
        urls: make(map[string]string),
        save: make(chan record, saveQueueLength),
    }
    if err := s.load(filename); err != nil {
        log.Println("URLStore:", err)
    }
    go s.saveLoop(filename)
    return s
}
```

Google

Un aparté : les *flags* de ligne de commande

La package Go `flag` permet de créer de manière aisée des flags de ligne de commande. Utilisons le pour remplacer les constantes de notre code.

```
import (
	"flag"
	"fmt"
	"http"
)
```

Nous créons tout d'abord des variables globales qui renferment les valeurs *flag* :

```
var (
	listenAddr  = flag.String("http", ":8080", "http listen address")
	dataFile    = flag.String("file", "store.gob", "data store file name")
	hostname    = flag.String("host", "localhost:8080", "host name and port")
)
```

Google

Un aparté : les *flags* de ligne de commande (2)

Nous pouvons ensuite ajouter `flag.Parse()` à la fonction principale (`main`), et instancier `URLStore` après avoir parsé les `flags` (une fois que l'on connait la valeur des `*dataFile`).

```
var store *URLStore func main() {
    flag.Parse()
    store = NewURLStore(*dataFile)
    http.HandleFunc("/", Redirect)
    http.HandleFunc("/add", Add)
    http.ListenAndServe(*listenAddr, nil)
}
```

Et substituer `*hostname` dans le *handler* `Add` :

```
fmt.Fprintf(w, "http://%s/%s", *hostname, key)
```

Google

Démonstration

Voir le code écrit jusqu'ici:

https://github.com/nf/goto/tree/master/talk/code/2/

Les transparents sont disponibles (en anglais) à l'adresse suivante :

http://wh3rd.net/practical-go/

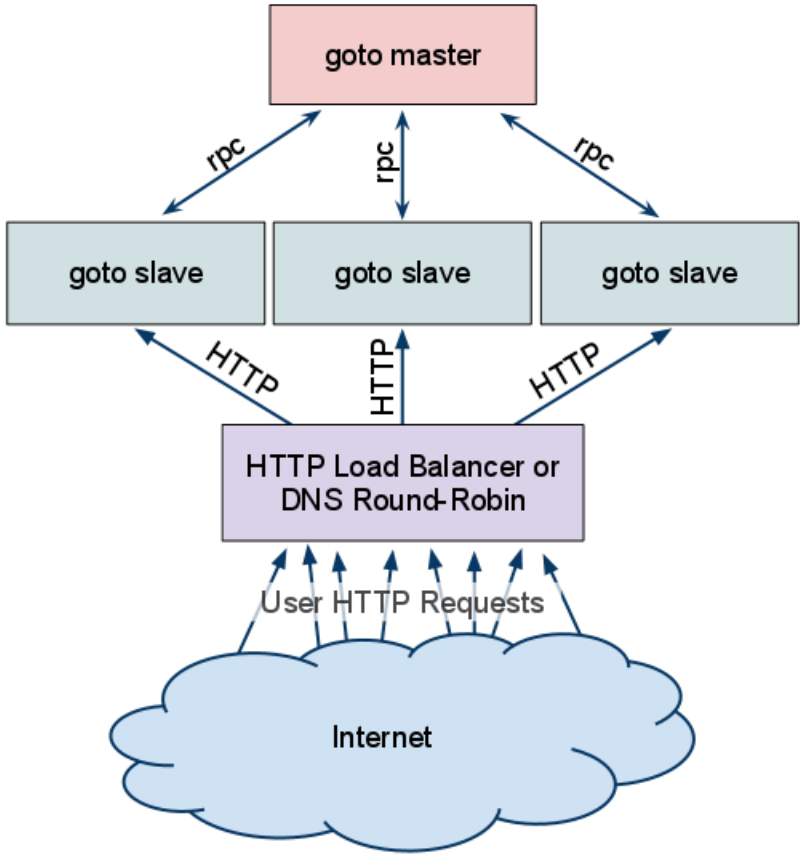
Google

Un point supplémentaire

Jusqu'à présent, nous avons un programme qui fonctionne comme un processus simple. Mais un processus unique tournant sur une machine ne peut traiter beaucoup de requêtes simultanées.

Un raccourciceur d'URL typiquement gèrent traditionnellement plus de requêtes de type *Redirects* (lecture) que de requêtes *Adds* (écriture).

Par conséquent, nous pouvons créer un nombre arbitraire d'esclaves en lecture seule et un cache pour traiter les requête de type *Get*, puis passer les requêtes de type *Put* au maître.



Google

Faire de URLStore un service RPC

La package Go `rpc` fournit une manière pratique de faire des appels de fonction à travers une connection réseau. Etant donné une valeur, `rpc` va exposer au réseau les méthodes qui correspondent à la signature de cette fonction :

```
func (t T) Name(args *ArgType, reply *ReplyType) os.Error
```

Pour faire de `URLStore` un service RPC , nous avons besoin de modifier `Put` et `Get` afin qu'elles correspondent respectivement aux signatures de fonction suivante :

```
func (s *URLStore) Get(key, url *string) os.Error
func (s *URLStore) Put(url, key *string) os.Error
```

Et, bien entendu, nous avons besoin de changer les sites d'appel pour appeler ces fonctions de manière appropriée.

Ces changements sont décrits en entier par quelques transparents à la fin de cette présentation. Ils sont été omis ici pour des questions de temps.

Google

Faire de URLStore un service RPC (2)

Ajouter un `flag` ligne de commande pour autoriser le serveur RPC :

```
var rpcEnabled = flag.Bool("rpc", false, "enable RPC server")
```

Et ensuite `Register` l'`URLStore` avec le package `rpc` puis initialiser le handler `RPC-over-HTTP` avec `HandleHTTP`.

```
func main() {
    flag.Parse()
    store = URLStore(*dataFile)
    if *rpcEnabled {
        rpc.RegisterName("Store", store)
        rpc.HandleHTTP()
    }
    ... (set up http)
}
```

Google

ProxyStore

Maintenant que nous avons rendu `URLStore` disponible en tant que service RPC, nous pouvons construire un autre type qui transfère les requêtes vers le serveur RPC.

Nous l'appellerons `ProxyStore`:

```
type ProxyStore struct {
    client *rpc.Client
}

func NewProxyStore(addr string) *ProxyStore {
    client, err := rpc.DialHTTP("tcp", addr)
    if err != nil {
        log.Println("ProxyStore:", err)
    }
    return &ProxyStore{client: client}
}
```

Google

ProxyStore (2)

Ces méthodes `Get` et `Put` transmettent les requêtes directement au serveur RPC :

```
func (s *ProxyStore) Get(key, url *string) os.Error {
    return s.client.Call("Store.Get", key, url)
}

func (s *ProxyStore) Put(url, key *string) os.Error {
  return s.client.Call("Store.Put", url, key)
}
```

Mais il y a quelque chose qui manque : l'esclave doit mettre en cache les données du maître, sinon il n'y aura aucun bénéfice à la manoeuvre.

Google

Un ProxyStore incluant un cache

Nous avons déjà défini la structure de données parfaite pour mettre en cache ces données, l' `URLStore`.
Ajoutons une instance d'`URLStore` à `ProxyStore`:

```
type ProxyStore struct {
    urls *URLStore
    client *rpc.Client
}

func NewProxyStore(addr string) *ProxyStore {
    client, err := rpc.DialHTTP("tcp", addr)
    if err != nil {
            log.Println("ProxyStore:", err)
    }
    return &ProxyStore{urls: NewURLStore(""), client: client}
}
```

(et nous devons aussi modifier l' URLStore afin qu'il n'essaye pas d'écrire ou de lire sur/à partir du disque si un nom de fichier vide est renvoyé)

Google

Un ProxyStore incluant un cache (2)

La méthode `Get` devra tout d'abord vérifier si la clef est dans le cache. Si elle l'est, `Get` devra retourner la valeur en cache. Sinon, elle devra faire un appel RPC, et mettre à jour le cache local avec ce résultat.

```
func (s *ProxyStore) Get(key, url *string) os.Error {
  if err := s.urls.Get(key, url); err == nil {
   return nil
  }
  if err := s.client.Call("Store.Get", key, url); err
  != nil {
    return err
  }
  s.urls.Set(key, url)
  return nil
}
```

Google

Un ProxyStore incluant un cache (3)

La méthode `Put` a seulement besoin de mettre à jour le cache quand elle exécute avec succès un `Put` RPC.

```
func (s *ProxyStore) Put(url, key *string)
os.Error {
    if err := s.client.Call("Store.Put",
                            url,key); err != nil{
        return err
    }
    s.urls.Set(key, url)
    return nil
}
```

Google

Intégrer le ProxyStore

Maintenant nous voulons pouvoir utiliser `ProxyStore` avec un front-end Web à la place de `URLStore`.

Puisque les deux implémentent les mêmes méthodes `Get` et `Put`, nous pouvons spécifier une interface pour généraliser leur comportement :

```
type Store interface {
  Put(url, key *string) os.Error
  Get(key, url *string) os.Error
}
```

Notre variable globale `store` désormais peut être du type `Store` :

```
var store Store
```

Google

Intégrer le ProxyStore (2)

Notre fonction `main` peut instancier soit une `URLStore` soit un `ProxyStore` cela dépend du `flag` ligne de commande entré :

```
var masterAddr = flag.String("master", "", "RPC master address")

func main() {
   flag.Parse()
   if *masterAddr != "" {
     // we are a slave
     store = NewProxyStore(*masterAddr)
   } else {
     // we are the master
     store = NewURLStore(*dataFile)
   } ...
}
```

Le reste du front-end (façade) continue de travailler comme précédemment. Il n'a pas besoin d'être au courant de l'interface `Store`.

Google

Démonstration finale

Nous pouvons désormais lancer un maître et plusieurs esclaves, et effectuer des tests de stress sur les esclaves.

Voir le programme complet :

https://github.com/nf/goto/tree/master/talk/code/3/

Les transparents en anglais sont disponibles à l'adresse suivante :

http://wh3rd.net/practical-go/

Google

Exercices pour le lecteur (ou l'auditeur)

Bien que ce programme fait ce pour quoi il a été conçu, il existe quelques moyens supplémentaires de l'améliorer :

- L'esthétique : l'interface utilisateur pourrait être bien plus sympathique. Voir le Wiki Codelab à golang.org pour des détails sur l'utilisation du package Go `template`.
- La fiabilité: les connexions maître/esclave pourraient être plus fiables. Si la connexion tombe, le client devrait réessayer de se connecter. Une goroutine "dialer" pourrait gérer cela.
- L'épuisement des ressources : au fur et à mesuree que la base de données grossit, la taille de la mémoire peut devenir problématique. Ceci pourrait être résolu en répartissant les ressources sur divers serveurs.
- La suppression: en support de la suppression des URL raccourcies, les intéractions entre le maître et les esclaves nécessiterait d'être plus efficaces.

Google

Des ressources Go

- **http://golang.org/** - la page officielle de Go.
  - Beaucoup de documentation (lire les specs du langage!!!)
  - Tutoriels (et les codelabs, et codewalks),
  - Le bac-à-sable Go (playground) (écrire, compiler, exécuter du code Go à partir d'un navigateur web)
  - Et bien plus...
- **http://blog.golang.org/** - le blog officiel de Go.
- **http://godashboard.appspot.com/package** - des bibliothèques Go écrites par la communauté.
- **http://groups.google.com/group/golang-nuts** - la mailing-list de Go
- **#go-nuts** on **irc.freenode.net** – aide en temps réel pour Go.

Google

Des questions?

Andrew Gerrand

adg@golang.org

http://wh3rd.net/practical-go/

Google

Des transparents additionnels…

Google

Faire de URLStore un service RPC (3)

Pour faire de `URLStore` un service RPC, nous avons besoin de modifier les méthodes `Get` et `Put` pour les rendre `rpc` compatibles. Les signatures de ces fonctions changent, et retournent maintenant une valeur `os.Error`.

La méthode `Get` peut retourner une erreur explicite quand la clef fournie n'est pas trouvée :

```
func (s *URLStore) Get(key, url *string) os.Error {
	s.mu.RLock()
	defer s.mu.RUnlock()
	if u, ok := s.urls[*key]; ok {
		*url = u return nil
	}
	return os.NewError("key not found")
}
```

Au delà de la signature de fonction, la méthode `Put` ne change pratiquement pas dans le code réel (pas montré ici) :

```
func (s *URLStore) Put(url, key *string) os.Error
```

Google

Faire de URLStore un service RPC (4)

A son tour, le handler `HTTP` doit être modifié pour se mettre au diapason des changement sur `URLStore`.

Le handler `Redirect` retourne maintenant une chaine d'erreur fournie par `URLStore`:

```
func Redirect(w http.ResponseWriter, r *http.Request) {
   key := r.URL.Path[1:]
   var url string
   if err := store.Get(&key, &url); err != nil {
     http.Error(w, err.String(), http.StatusInternalServerError)
     return
   }
   http.Redirect(w, r, url, http.StatusFound)
}
```

Google

Faire de URLStore un service RPC (5)

Le *handler* `Add` change de manière conséquente de la même façon :

```
func Add(w http.ResponseWriter, r *http.Request) {
  url := r.FormValue("url")
  if url == "" {
   fmt.Fprint(w, AddForm)
   return
  }
  var key string
  if err := store.Put(&url, &key); err != nil {
   http.Error(w, err.String(),
  http.StatusInternalServerError)    return
  }
  fmt.Fprintf(w, "http://%s/%s", *hostname, key)
}
```

Google

FIN ☺

Google