Un panorama des parseurs YAML en Go

Michel Casabianca
casa@sweetohm.net

Je me propose de réaliser dans cet article un panorama des parseurs YAML en Go. Pour chacun d'entre eux, je fournis une brève description, une évaluation et un exemple de code.

Vous pouvez télécharger ici une archive avec les sources des exemples.

À noter que les noms des parseurs ont été changés par mes soins dans la mesure où tous ces parseurs s'appellent **goyaml** :o)

Dans les codes d'exemple, nous parserons le fichier YAML suivant :

```
foo: 1
bar:
  - one
  - two
```

Dans chacun des sources, nous extraierons et afficherons le premier élément de la liste se trouvant sous la clé *foo* du tableau associatif au premier niveau.

Goyaml

- Homepage: http://gopkg.in/yaml.v2.
- Documentation: http://godoc.org/gopkg.in/yaml.v2.

Ce parseurs est très pratique pour charger des fichiers de configuration. Pour ce faire, il suffit de déclarer une struct ayant la structure du fichier YAML à charger et de passer une référence à cette structure au parseur lors du chargement. Le parseur va alors peupler la structure avec les données du fichier YAML.

Ce parseur est probablement la meilleure option pour charger des fichiers de configuration YAML.

Exemple

```
package main
```

```
import (
    "fmt"
    "gopkg.in/yaml.v2"
    "io/ioutil"
    "os"
)

type Config struct {
    Foo string
    Bar []string
}

func main() {
    filename := os.Args[1]
    var config Config
    source, err := ioutil.ReadFile(filename)
    if err != nil {
        panic(err)
    }
    err = yaml.Unmarshal(source, &config)
    if err != nil {
        panic(err)
    }
    fmt.Printf("Value: %#v\n", config.Bar[0])
}
```

Pour exécuter ce fichier d'exemple, on pourra taper la commande suivante :

```
go run src/goyaml.go src/test.yml
```

Gypsy

- Homepage: https://github.com/kylelemons/go-gypsy.
- Documentation: https://godoc.org/github.com/kylelemons/go-gypsy/yaml.

Gypsy est un parseur bas niveau, ce qui veut dire que le parsing d'un fichier renverra un arbre de noeuds qu'il vous faudra explorer *à la main* (comme dans l'exemple suivant) :

Exemple

```
package main

import (
    "fmt"
    "github.com/kylelemons/go-gypsy/yaml"
    "os"
)

func nodeToMap(node yaml.Node) (yaml.Map) {
    m, ok := node.(yaml.Map)
```

```
        if !ok {
            panic(fmt.Sprintf("%v is not of type map", node))
        }
        return m
    }

    func nodeToList(node yaml.Node) (yaml.List) {
        m, ok := node.(yaml.List)
        if !ok {
            panic(fmt.Sprintf("%v is not of type list", node))
        }
        return m
    }

    func main() {
        filename := os.Args[1]
        file, err := yaml.ReadFile(filename)
        if err != nil {
            panic(err)
        }
        value := nodeToList(nodeToMap(file.Root)["bar"])[0]
        fmt.Printf("Value: %#v\n", value)
    }
```

Pour exévuter cet exemple, on pourra taper la commande suivante :

```
    go run src/gypsy.go src/test.yml
```

Bug

Le parseurs Gypsy ne parse pas le fichier YAML suivant (à cause du niveau d'indentation de la liste qui est le même que celui de la map) :

```
    foo: 1
    bar:
    - one
    - two
```

Beego

- Homepage: https://github.com/beego/goyaml2.
- Documentation: http://godoc.org/github.com/beego/goyaml2.

Beego est un outil de build, mais il inclut un parseur YAML bas niveau. Ce parseur implémente un mapping très naturel :

- Une *map* YAML donne en Go *map[string]interface{}*.
- Une *liste* YAML donne en Go *[]interface{}*.
- et ainsi de suite...

Exemple

```
package main

import (
    "fmt"
    "github.com/beego/goyaml2"
    "os"
)

func nodeToMap(node interface{}) map[string]interface{} {
    m, ok := node.(map[string]interface{})
    if !ok {
        panic(fmt.Sprintf("%v is not of type map", node))
    }
    return m
}

func nodeToList(node interface{}) []interface{} {
    m, ok := node.([]interface{})
    if !ok {
        panic(fmt.Sprintf("%v is not of type list", node))
    }
    return m
}

func main() {
    filename := os.Args[1]
    file, err := os.Open(filename)
    if err != nil {
        panic(err)
    }
    object, err := goyaml2.Read(file)
    if err != nil {
        panic(err)
    }
    value := nodeToList(nodeToMap(object)["bar"])[0]
    fmt.Printf("Value: %#v\n", value)
}
```

Golly

- Homepage: http://github.com/tav/golly.
- Documentation: http://godoc.org/github.com/tav/golly/yaml.

Ce parseur fait partie d'une bibliothèque Go généraliste. Elle ne peut parser que des dictionnaires et son API est un peu retorse.

Exemple

```
package main
```

```
import (
    "fmt"
    "github.com/tav/golly/yaml"
    "os"
)

func main() {
    filename := os.Args[1]
    data, err := yaml.ParseFile(filename)
    if err != nil {
        panic(err)
    }
    list, ok := data.GetList("bar")
    if !ok {
        panic("Not OK!")
    }
    value := list[0].String
    fmt.Printf("Value: %#v\n", value)
}
```

Simple YAML

- Homepage: https://github.com/smallfish/simpleyaml.
- Documentation: http://godoc.org/github.com/smallfish/simpleyaml.

C'est une surcouche à GoYAML pour simplifier le parsing de fichiers YAML. On n'a pas à décrire la structure du document, on accède directement au contenu par exploration de l'arbre. Cependant, ce qu'on économise d'un côté (la description de la structure du document), on le perd de l'autre (code pour explorer l'arbre et gérer les erreurs)...

Exemple

```
package main

import (
        "fmt"
        "github.com/smallfish/simpleyaml"
        "io/ioutil"
        "os"
)

func main() {
        filename := os.Args[1]
        source, err := ioutil.ReadFile(filename)
        if err != nil {
                panic(err)
        }
        yaml, err := simpleyaml.NewYaml(source)
        if err != nil {
                panic(err)
        }
        bar, err := yaml.Get("bar").GetIndex(0).String()
```

```
        if err != nil {
                panic(err)
        }
        fmt.Printf("Value: %#v\n", bar)
}
```

Zombiezen

- Homepage: https://bitbucket.org/zombiezen/yaml.
- Documentation: Pas de documentation connue.

Le corps de la struct `Parser` est un commentaire TODO... Ne semble donc pas encore opérarionnel.