Introduction à YAML

Michel Casabianca
casa@sweetohm.net

Cet article est une introduction à YAML, un langage permettant de représenter des données structurées, comme le ferait XML par exemple, mais de manière plus naturelle et moins verbeuse. On y verra une description de la syntaxe de YAML ainsi que des exemples en Java, Python et Go.

An english version is available here.

On trouvera une archive ZIP avec cet article au format PDF ainsi que les exemples à l'adresse : http://www.sweetohm.net/arc/introduction-yaml.zip.

Qu'est ce que YAML ?

Le nom YAML veut dire "YAML Ain't Markup Language", soit "YAML n'est pas un langage de balises". Si cela met d'emblée des distances avec XML, cela ne nous dit pas ce qu'est YAML. YAML est, d'après sa spécification, un langage de sérialisation de données conçu pour être lisible par des humains et travaillant bien avec les langage de programmation modernes pour les tâches de tous les jours.

Concrètement, on pourrait noter la liste des ingrédients pour un petit déjeuner de la manière suivante :

```
- croissants
- chocolatines
- jambon
- oeufs
```

Ceci est un fichier YAML valide qui représente une liste de chaînes de caractères. Pour s'en convaincre, nous pouvons écrire le script Python suivant qui parse le fichier, dont le nom est passé sur la ligne de commande, et affiche le résultat :

```
#!/usr/bin/env python
# encoding: UTF-8

import sys
import yaml

print yaml.load(open(sys.argv[1]))
```

Ce script produira le résultat suivant :

```
['croissants', 'chocolatines', 'jambon', 'oeufs']
```

Ce qui veut dire que le résultat de ce parsing est une liste Python contenant les chaînes de caractères appropriées ! Le parseur est donc capable de restituer des structures de données *naturelles* du langage utilisé pour le parsing.

Écrivons maintenant un compte à rebours :

```
- 3
- 2
- 1
- 0
```

Nous obtenons le résultat suivant :

```
[3, 2, 1, 0]
```

C'est toujours une liste, mais le parseur a reconnu chaque élément comme étant un entier et non une simple chaîne de caractères comme dans l'exemple précédent. Le parseur est donc capable de distinguer des types de données tels que chaînes de caractères et entiers, nombres à virgule flottante et dates. On utilise pour ce faire une syntaxe naturelle : *3* est reconnu comme un entier alors que *croissants* ne l'est pas car ce dernier ne peut être converti ni en entier, ni en nombre à virgule flottante, ni en tout autre type reconnu par YAML. Pour forcer YAML à interpréter *3* comme une chaîne de caractères, on peut l'entourer de guillemets.

YAML peut aussi reconnaître des tableaux associatifs, ainsi on pourrait noter une commande de petit déjeuner de la manière suivante :

```
croissants: 30
chocolatines: 30
jambon: 0
oeufs: 0
```

Qui sera chargé de la manière suivante :

```
{'chocolatines': 30, 'croissants': 30, 'jambon': 0, 'oeufs': 0}
```

En combinant les types de données de base dans les collections reconnues par YAML, on peut représenter quasiment toute structure de données. D'autre part, la représentation textuelle de ces données est très lisible et quasiment naturelle.

Il est aussi possible de réaliser l'opération inverse, à savoir sérialiser des structures de données en mémoire sous forme de texte. Dans l'exemple suivant, nous écrivons sur la sortie standard un `Dictionnaire` Python :

```
#!/usr/bin/env python
# encoding: UTF-8

import yaml

recette = {
    'nom': 'sushi',
    'ingredients': ['riz', 'vinaigre', 'sucre', 'sel', 'thon', 'saumon'],
    'temps de cuisson': 10,
    'difficulte': 'difficile'
}

print yaml.dump(recette)
```

Qui produira la sortie suivante :

```
difficulte: difficile
ingredients: [riz, vinaigre, sucre, sel, thon, saumon]
nom: sushi
temps de cuisson: 10
```

Syntaxe de base

Après cette brève introduction, voici une description plus exhaustive de la syntaxe YAML.

Scalaires

Les scalaires sont l'ensemble des types YAML qui ne sont pas des collections (liste ou tableau associatif). Ils peuvent être représentés par une liste de caractères Unicode. Voici une liste des scalaires reconnus par les parseurs YAML :

Chaîne de caractères

Voici un exemple :

```
- Chaîne
- "3"
- Chaîne sur
  une ligne
- "Guillemets doubles\t"
- 'Guillemets simples\t'
```

Qui est parsé de la manière suivante :

```
[u'Cha\xeene', '3', u'Cha\xeene sur une ligne',
  'Guillemets doubles\t', 'Guillemets simples\\t']
```

Le résultat de ce parsing nous amène aux commentaires suivants :

- Les caractères accentués sont gérés, en fait, l'Unicode est géré de manière plus générale.
- Les retours à la ligne ne sont pas pris en compte dans les chaînes, ils sont gérés comme en HTML ou XML, à savoir qu'ils sont remplacés par des espaces.
- Les guillemets doubles gèrent les caractères d'échappement, comme \t pour la tabulation par exemple.
- Les guillemets simples ne gèrent pas les caractères d'échappement qui sont transcrits de manière littérale.
- La liste des caractères d'échappement gérés par YAML comporte les valeurs classiques, mais aussi nombre d'autres que l'on pourra trouver dans la spécification YAML.

D'autre part, il est possible d'écrire des caractères Unicode à l'aide des notations suivantes :

- `\xNN` : pour écrire des caractères Unicode sur 8 bits, où `NN` est un nombre hexadécimal.
- `\uNNNN` : pour des caractères Unicode sur 16 bits.
- `\UNNNNNNNN` : pour des caractères Unicode sur 32 bits.

Entiers

Voici quelques exemples :

```
canonique:   12345
decimal:     +12_345
sexagesimal: 3:25:45
octal:       030071
hexadecimal: 0x3039
```

Qui est parsé de la manière suivante :

```
{'octal': 12345, 'hexadecimal': 12345, 'canonique': 12345,
 'decimal': 12345, 'sexagesimal': 12345}
```

Nous constatons que les notations les plus courantes des langages de programmation (comme l'octal ou l'hexadécimal) sont gérées. A noter que toutes ces notations seront reconnues comme identiques par un parseur YAML et par conséquent seront équivalentes comme clef d'un tableau associatif par exemple.

Nombres à virgule flottante

Voyons les différentes notations pour ces nombres :

```
canonique:      1.23015e+3
exponentielle:  12.3015e+02
sexagesimal:    20:30.15
fixe:           1_230.15
infini negatif: -.inf
```

```
pas un nombre:  .NaN
```

Ce qui est parsé en :

```
{'pas un nombre': nan, 'sexagesimal': 1230.1500000000001,
 'exponentielle': 1230.1500000000001, 'fixe': 1230.1500000000001,
 'infini negatif': -inf, 'canonique': 1230.1500000000001}
```

Les notations classiques sont gérées ainsi que les infinis et les valeurs qui ne sont pas des nombres.

Dates

YAML reconnaît aussi des dates :

```
canonique: 2001-12-15T02:59:43.1Z
iso8601:   2001-12-14t21:59:43.10-05:00
espace:    2001-12-14 21:59:43.10 -5
date:      2002-12-14
```

Qui sont parsées de la manière suivante :

```
{'date': datetime.date(2002, 12, 14),
 'iso8601': datetime.datetime(2001, 12, 15, 2, 59, 43, 100000),
 'canonique': datetime.datetime(2001, 12, 15, 2, 59, 43, 100000),
 'espace': datetime.datetime(2001, 12, 15, 2, 59, 43, 100000)}
```

Les types résultant du parsing dépendent du langage et du parseur, mais correspondent à des types naturels pour les temps considérés.

Divers

Il existe d'autres scalaires reconnus par les parseurs YAML :

```
nul: null
nul bis: ~
vrai: true
vrai bis: yes
vrai ter: on
faux: false
faux bis: no
faux ter: off
```

Qui sera parsé en :

```
{'faux bis': False, 'vrai ter': True, 'vrai bis': True,
 'faux ter': False, 'nul': None, 'faux': False,
```

```
        'nul bis': None, 'vrai': True}
```

Bien sûr, le type des valeurs parsées dépend du langage et d'autres valeurs spéciales peuvent être reconnues suivant les langages et les parseurs. Par exemple, le parseur Ruby reconnaît les symboles (notés `:symbole` par exemple) et les parse en symboles Ruby.

Collections

Il existe deux types de collections reconnues par YAML : les listes et les tableaux associatifs.

Listes

Ce sont des listes ordonnées, et qui peuvent contenir plusieurs éléments identiques (par opposition aux ensembles). Les éléments d'une liste sont identifiés par un tiret, comme suit :

```
- croissants au
  beurre
- chocolatines
- jambon
- oeufs
```

Les éléments de la liste sont distingués grâce à l'indentation : le premier élément est indenté de manière à ce que sa deuxième ligne soit reconnue comme faisant partie du premier élément de la liste. Cette syntaxe peut être comparée à celle de Python, si ce n'est qu'en YAML, **les caractères de tabulation sont strictement interdits pour l'indentation**. Cette dernière règle est importante et source de nombreuses erreurs de parsing. Il est important de paramétrer son éditeur de manière à interdire les tabulations pour l'indentation des fichiers YAML.

Il existe une notation alternative pour les listes, semblable à celle des langages Python ou Ruby :

```
[croissants, chocolatines, jambon, oeufs]
```

Cette notation, dite *en flux*, est plus compacte et permet parfois de gagner en lisibilité ou compacité.

Tableaux associatifs

Appelés Map ou Dictionnaires dans certains langages, ils associent une valeur à une clef :

```
croissants: 2
chocolatines: 1
jambon: 0
oeufs: 3
```

La notation *en flux* est la suivante :

```
{ croissants: 2, chocolatines: 1, jambon: 0, oeufs: 3}
```

Qui est parsé de la même manière. Cette notation est identique à celle de Python ou Javascript et se rapproche de celle utilisée par Ruby. A noter qu'il est question que Ruby 2 utilise aussi cette notation.

Commentaires

Il est possible d'inclure des commentaires dans un document de la même manière que dans la plupart des langages de script :

```
# commentaire
- Du texte
# autre commentaire
- Autre texte
```

A noter que ces commentaires ne doivent (et ne peuvent) contenir d'information utile au parsing dans la mesure où ils ne sont pas accessibles, généralement, au code client du parser.

Documents multiples

Dans un même fichier ou flux, on peut insérer plusieurs documents YAML å la suite, en les faisant commencer par une ligne composée de trois tirets (`---`) et en les termiant d'une ligne de trois points (`...`) comme dans l'exemple ci-dessous :

```
---
premier document
...
---
deuxième document
...
```

A noter que par défaut, les parsers YAML attendent un document par fichier et peuvent émettre une erreur s'ils rencontrent plus d'un document. Il faut alors utiliser une fonction particulière capable de parser des documents multiples (comme `yaml.load_all()` pour PyYaml par exemple).

On peut alors extraire ces documents de manière séquentielle du flux.

Syntaxe avancée

Avec la section précédente, nous avons vu le minimum vital pour se débrouiller avec YAML. Nous allons maintenant aborder des notions plus avancées dont on peut souvent se passer dans un premier temps.

Références

Les références YAML sont semblables aux pointeurs des langages de programmation. Par exemple :

```
lundi:    &p 'des patates'
mardi:    *p
mercredi: *p
jeudi:    *p
vendredi: *p
samedi:   *p
dimanche: *p
```

Donne, après parsing :

```
{'mardi': 'des patates', 'samedi': 'des patates',
 'jeudi': 'des patates', 'lundi': 'des patates',
 'vendredi': 'des patates', 'dimanche': 'des patates',
 'mercredi': 'des patates'}
```

A noter qu'un alias, indiqué par une astérisque `*`, doit pointer vers une ancre valide, indiquée par une esperluette `&`, sans quoi il en résulte une erreur de parsing. Ainsi le fichier suivant doit provoquer une erreur lors du parsing :

```
*foo
```

Tags

Les tags sont les indicateurs du type de données. Par défaut, il n'est pas nécessaire d'indiquer le type des données qui est déduit de leur forme. Cependant, dans certains cas, il peut être nécessaire de forcer le type d'une donnée et YAML définit les types par défaut suivants :

```
null:     !!null
integer:  !!int    3
float:    !!float  1.2
string:   !!str    string
boolean:  !!bool   true
binary:   !!binary dGVzdA==
map:      !!map    { key: value }
seq:      !!seq    [ element1, element2 ]
ensemble: !!set    { element1, element2 }
omap:     !!omap   [ key: value ]
```

Les tags correspondants commencent par deux points d'exclamation. Lors du parsing, on obtient les type suivants en Python :

```
{'binary': 'test',
 'string': 'string',
```

```
 'seq': ['element1', 'element2'],
 'map': {'key': 'value'},
 'float': 1.2,
 'boolean': True,
 'omap': [('key', 'value')],
 None: None,
 'integer': 3,
 'ensemble': set(['element1', 'element2'])}
```

A noter les deux types supplémentaires :

- L'ensemble : il n'est pas ordonné et ne peut comporter de doublon.
- Le tableau associatif ordonné : c'est un tableau associatif dont les entrées sont ordonnées.

L'utilité des tags pour ces types par défaut est limitée. La vrai puissance des tags réside dans la possibilité de définir ses propres tags pour ses propres types de données.

Par exemple, on pourrait définir son propre type pour les personnes, comportant deux champs : le nom et le prénom. On doit tout d'abord déclarer le tag au début du document, puis on peut l'utiliser dans la suite, comme dans cet exemple :

```
%TAG !personne! tag:foo.org,2004:bar
---
- !personne
    nom:    Simpson
    prenom: Omer
- !personne
    nom:    Simpson
    prenom: Bart
```

Nous verrons plus loin comment utiliser les tags avec les APIs Java et Python pour désérialiser des structures YAML en types de donnés personnalisés.

Il est aussi possible de ne pas déclarer le tag et de l'expliciter dans le document, de la manière suivante :

```
- !<tag:foo.org,2004:bar>
    nom:    Simpson
    prenom: Omer
- !<tag:foo.org,2004:bar>
    nom:    Simpson
    prenom: Bart
```

Directives

Les directives donnent des instructions au parser. Il en existe deux :

TAG

Comme vu précédemment, déclare un tag dans le document.

YAML

Indique la version de YAML du document. Doit être en en-tête du document, comme dans l'exemple ci-dessous :

```
%YAML 1.1
---
test
```

Un parser doit refuser de traiter un document d'une version majeure supérieure. Par exemple, un parser `1.1` devrait refuser de parser un document en version YAML `2.0`. Il devrait émettre un warning si on lui demande de parser un document de version mineure supérieure, comme `1.2` par exemple. Il doit parser sans protester toutes les versions égales ou inférieures, telles que `1.1` et `1.0`.

Jeu de caractères et encodage

Un parser YAML doit accepter tout caractère Unicode, à l'exception de certains caractères spéciaux. Ces caractères peuvent être encodés en *UTF-8* (encodage par défaut), *UTF-16* ou *UTF-32*. Les parsers YAML sont capables de déterminer l'encodage du texte en examinant le premier caractère. Il est donc **impossible** d'utiliser tout autre encodage dans un fichier YAML et en particulier `ISO-8859-1`.

APIs YAML

Nous allons maintenant jouer avec les principales APIs YAML.

JYaml

JYaml est une bibliothèque OpenSource pour manipuler les documents YAML en Java. Le projet est hébergé par SourceForge et on trouvera sa page à l'adresse http://jyaml.sourceforge.net/. On trouvera sur ce site un tutoriel ainsi que les références de l'API.

Utilisation de base

JYaml effectue un mapping par défaut des structures YAML en objets Java standards : il associe une liste à une instance de `java.util.ArrayList`, un tableau associatif à une instance de `java.util.HashMap` et les types primitifs de YAML à leur contrepartie Java.

Ainsi, pour charger un fichier YAML dans un objet Java, on écrira le code suivant :

```
Object object = Yaml.load(new File("object.yml"));
```

Par exemple, le fichier suivant :

```
- Un
- 2
- { trois: 3.0, quatre: true }
```

Pourra être chargé en mémoire et affiché dans le terminal avec le source suivant :

```
package jyaml;

import java.io.File;
import org.ho.yaml.Yaml;

public class Load {

    public static void main(String[] args)
        throws Exception {
        String filename = "test/object.yml";
        if (args.length > 0) filename = args[0];
        System.out.println(Yaml.load(new File(filename)));
    }

}
```

Cela affichera dans le terminal :

```
[Un, 2, {quatre=true, trois=3.0}]
```

Inversement, on peut sérialiser un objet Java dans un fichier YAML de la manière suivante :

```
Yaml.dump(object, new File("dump.yml"));
```

Ainsi, on pourra par exemple sérialiser une structure d'objets Java avec le code suivant :

```
package jyaml;

import java.io.File;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.ho.yaml.Yaml;

public class Dump {

    public static void main(String[] args)
```

```
            throws Exception {
            List<Object> object = new ArrayList<Object>();
            object.add("Un");
            object.add(2);
            Map<String, Object> map = new HashMap<String, Object>();
            map.put("trois", 3.0);
            map.put("quatre", true);
            object.add(map);
            Yaml.dump(object, new File("test/dump.yml"));
        }

    }
```

Ce code produira le fichier suivant :

```
---
- Un
- 2
- !java.util.HashMap
  quatre: true
  trois: !java.lang.Double 3.0
```

A noter que ce dump est un peu décevant dans la mesure où certains types standards de YAML (comme les tableaux associatifs et les nombres à virgule flottante) sont sérialisés en types Java (comme `java.util.HashMap` et `java.lang.Double`). Un tel fichier ne sera pas chargé correctement en utilisant un autre langage de programmation (voire même une autre implémentation en Java).

Usage avancé

Nous pouvons aussi travailler avec des types qui ne sont pas génériques et ainsi charger des instances de classes Java à partir de fichiers YAML.

La première solution consiste à indiquer le type des objets avec des tags YAML. Ainsi, le fichier YAML suivant :

```
--- !jyaml.Commande
id: test123
articles:
 - !jyaml.Article
    id:       test456
    prix:     3.5
    quantite: 1
 - !jyaml.Article
    id:       test567
    prix:     2.0
    quantite: 2
```

Sera-t-il chargé en utilisant les classes suivantes :

```
package jyaml;

public class Commande {

    private String id;
    private Article[] articles;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Article[] getArticles() {
        return articles;
    }

    public void setArticles(Article[] articles) {
        this.articles = articles;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer("[Commande id='")
            .append(id)
            .append("', articles='");
        for (int i=0; i<articles.length; i++) {
            Article article = articles[i];
            buffer.append(article.toString());
            if (i<articles.length-1) buffer.append(", ");
        }
        buffer.append("]");
        return buffer.toString();
    }

}
```

Et :

```
package jyaml;

public class Article {

    private String id;
    private double prix;
    private int quantite;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
```

```
    public double getPrix() {
        return prix;
    }

    public void setPrix(double prix) {
        this.prix = prix;
    }

    public int getQuantite() {
        return quantite;
    }

    public void setQuantite(int quantite) {
        this.quantite = quantite;
    }

    public String toString() {
        return "[Article id='"+id+"', prix='"+prix+"', quantite='"+quantite+"']";
    }

}
```

Ces classes respectent la convention JavaBean, à savoir qu'elles disposent d'accesseurs pour ses champs ainsi que d'un constructeur vide (sans arguments, implicite en Java). En le chargeant avec le source précédent, nous obtenons sur le terminal :

```
[Commande id='test123', articles='[
  [Article id='test456', prix='3.5', quantite='1'],
  [Article id='test567', prix='2.0', quantite='2']
  ]'
]
```

Il existe une autre solution pour charger ces objets sans avoir besoin de spécifier explicitement les types. Pour ce faire, il faut utiliser la méthode `loadType()` et lui passer le fichier YAML à charger ainsi que le type de l'objet racine du fichier. Ainsi, dans notre cas, nous pourrions écrire le fichier de commande de la manière suivante :

```
id: test123
articles:
 - id:       test456
   prix:     3.5
   quantite: 1
 - id:       test567
   prix:     2.0
   quantite: 2
```

Et le charger avec le source suivant :

```
package jyaml;

import java.io.File;
```

```
import org.ho.yaml.Yaml;

public class Load2 {

    public static void main(String[] args)
        throws Exception {
        System.out.println(Yaml.loadType(new File("test/commande2.yml"),
                                         Commande.class));
    }

}
```

Cette manière de charger des types spécifiques est bien plus pratique car elle ne surcharge pas le fichier YAML avec les types Java, ce qui rend la portabilité entre langages nulle. Cependant, on pourra regretter l'obligation de déclarer le champ `articles` en tant que tableau d'`Article`. Si on le déclare du type `List<Article>`, JYaml charge les objets de la liste comme des instances de `Map`. Cependant, ceci est du au fait que l'information du type de la liste est perdu au runtime et donc JYaml ne peut connaître le type des éléments de la liste et les charge donc avec le type par défaut.

Alias et ancres

JYaml gère les alias et les ancres des fichiers YAML. Prenons l'exemple suivant :

```
- &rob
  nom: Robert
  age: 55
- &elo
  nom: Elodie
  age: 52
- nom: Mickael
  age: 31
  parents:
    - *rob
    - *elo
```

On peut le charger avec le source suivant :

```
package jyaml;

import java.io.File;
import org.ho.yaml.Yaml;

public class Alias {

    public static void main(String[] args)
        throws Exception {
        Personne[] personnes = Yaml.loadType(new File("test/alias.yml"),
                                             Personne[].class);
        for(int i=0; i<personnes.length; i++) {
            Personne personne = personnes[i];
            System.out.println(personne);
        }
```

```
            // on teste que les références sont bien identiques
            System.out.println("Ancre OK: "+(personnes[2].getParents()[0]==personnes[0]
        }

    }
```

Et l'on constate que les alias et ancres ont été gérés correctement.

Gestion des flux

Il est possible de sérialiser en YAML des objets Java dans un flux de la manière suivante :

```
    YamlEncoder enc = new YamlEncoder(outputStream);
    enc.writeObject(object1);
    enc.writeObject(object2);
    enc.close();
```

Il existe un raccourci pour sérialiser une collection d'objets dans un flux de la manière suivante :

```
    Yaml.dumpStream(collection.iterator(), file);
```

D'autre part, on peut désérialiser des objets Java à partir d'un flux YAML de la manière suivante :

```
    YamlDecoder dec = new YamlDecoder(inputStream);
    try{
      while(true) {
        Object object = dec.readObject();
        /* do something useful */
      }
    } catch(EOFException e) {
      System.out.println("Finished reading stream.");
    } finally {
      dec.close();
    }
```

Il existe un raccourci pour itérer sur les objets désérialisés d'un flux :

```
    for(Object object: Yaml.loadStream(input)) {
      /* do something useful */
    }
```

Fichier de configuration

Il est possible de configurer JYaml dans un fichier, qui doit être nommé `jyaml.yml` et se trouver dans le répertoire courant où tourne l'application ou bien à la racine de son *CLASSPATH*. Voici un exemple d'un tel fichier :

```
minimalOutput: true
indentAmount: "    "
suppressWarnings: true
encoding: "ISO-8859-1"
transfers:
  company: com.blah.Company
  employee: com.blah.Employee
handlers:
  com.mycompany.MyFunkyObject: com.mycompany.jyaml.MyFunkyObjectWrapper
```

Cette configuration permet en particulier de configurer des mappings entre tags YAML et classes Java. Dans l'exemple ci-dessus, le mapping suivant :

```
company: com.blah.Company
```

Permet de raccourcir le tag YAML permettant d'indiquer la classe Java utilisée pour la désérialisation. Ainsi, on pourra indiquer le mapping pour la classe `com.blah.Company` par un simple tag `!company`.

D'autre part, ce fichier permet de lister des classes wrappers qui permettent de désérialiser des types particuliers dont les classes ne respectent pas la convention JavaBeans. On peut en trouver des exemples dans les sources du projet.

Autres fonctionnalités

A noter que JYaml a plus à offrir, en particulier :

- Un support de fichiers de configuration pour Spring en YAML. Cela permet d'écrire des fichiers plus lisibles et plus compacts que leurs équivalents XML. Pour plus d'information, voir cette page.
- Un format YAML pour les fichiers de configuration DBUnit. Voir sur cette page.

Cependant, JYaml semble soufrir de limitations, en particulier pour le parsing des dates qui ne sont pas toujours reconnues comme telles.

PyYAML

PyYaml est une bibliothèque en Python permettant de gérer les fichiers YAML. On peut le télécharger sur le site http://pyyaml.org/ et l'on trouvera la documentation sur cette page.

Installation

PyYaml peut utiliser la LibYaml écrite en C et très rapide ou bien une implémentation en pûr Python qui ne nécessite pas cette bibliothèque. Pour installer la bibliothèque, télécharger l'archive, la décompresser, se rendre dans le répertoire ainsi créé et taper `python setup.py install`, ou bien `python setup.py --wihout-libyaml install` pour ne pas utiliser la LibYaml.

Utilisation de base

Pour charger un fichier YAML, dont le nom est passé sur la ligne de commande, on pourra procéder somme suit :

```
#!/usr/bin/env python
# encoding: UTF-8

import sys
import yaml

print yaml.load(open(sys.argv[1]))
```

La fonction `load()` prend en paramètre une chaîne d'octets, unicode, un fichier binaire ou texte. Les chaînes d'octets et les fichiers doivent être encodés en *UTF-8* ou *UTF-16*. L'encoding est déterminé par le parser en examinant le BOM (Byte Order Mark), premier octet du fichier. Si aucun BOM n'est trouvé, l'*UTF-8* est choisi.

Si la chaîne ou le fichier contient plusieurs documents, on peut tous les charger à l'aide de la fonction `yaml.load_all()` qui renvoie un itérateur. On pourra donc écrire :

```
for document in yaml.load_all(documents):
    print document
```

Pour sérialiser un objet Python en YAML, on pourra utiliser la fonction `yaml.dump()` :

```
#!/usr/bin/env python
# encoding: UTF-8

import yaml

recette = {
    'nom': 'sushi',
    'ingredients': ['riz', 'vinaigre', 'sucre', 'sel', 'thon', 'saumon'],
    'temps de cuisson': 10,
    'difficulte': 'difficile'
}

print yaml.dump(recette)
```

La fonction `yaml.dump()` peut prendre un deuxième paramètre optionel qui doit être un fichier binaire ou texte ouvert. Elle écrit alors le résultat de la sérialisation dans le fichier.

Pour sérialiser plusieurs objets Python dans un flux, vous pouvez utiliser ma fonction `yaml.dump_all()`. Cette fonction prend en paramètre une liste ou un itérateur.

Les fonctions de dump prennent des paramètres supplémentaires pour indiquer des détails de formatage du YAML généré. On peut ainsi spécifier le nombre de caractères d'indentation à utiliser, le nombre de caractères par ligne, etc. En particulier, `default_flow_style` indique si on

utilise le style par flux pour les listes et les tableaux associatifs. Ainsi, le code suivant :

```
print yaml.dump(range(5), default_flow_style=True)
```

Produit une représentation utilisant la notation en flux :

```
[0, 1, 2, 3, 4]
```

Alors que le code :

```
print yaml.dump(range(5), default_flow_style=False)
```

Produit une notation en liste :

```
- 0
- 1
- 2
- 3
- 4
```

Sérialisation et désérialisation de classes Python

Il est possible de déclarer explicitement le type Python à utiliser pour désérialiser une structure YAML donnée à l'aide d'un tag YAML. Par exemple :

```
#!/usr/bin/env python
# encoding: UTF-8

import yaml

class Personne(object):

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def __repr__(self):
        return "%s(nom=%r, age=%r)" % \
               (self.__class__.__name__, self.nom, self.age)

print yaml.load("""
!!python/object:__main__.Personne
nom: Robert
age: 25
""")
```

Produit la sortie suivante dans le terminal :

```
Personne(nom='Robert', age=25)
```

Inversement, une classe Python peut être sérialisée en un flux YAML de la manière suivante :

```
#!/usr/bin/env python
# encoding: UTF-8

import yaml

class Personne(object):

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def __repr__(self):
        return "%s(nom=%r, age=%r)" % \
               (self.__class__.__name__, self.nom, self.age)

print yaml.dump(Personne('Robert', 25), default_flow_style=False)
```

Ce code produit la sortie suivante :

```
!!python/object:__main__.Personne
age: 25
nom: Robert
```

Nous voyons que PyYaml a sérialisé la classe Python en utilisant la même notation à base de tag YAML. A noter que la construction d'objets Python arbitraires à partir de sources non fiables (typiquement venant d'internet) peut être dangereuse. C'est pourquoi PyYaml propose la fonction `yaml.safe_load()` qui limite la construction d'objets aux types de base de YAML.

La notation vue ci-dessus permet de désérialiser des structures YAML en instances de classes Python arbitraires. Cependant, il est possible d'utiliser une notation plus simple en faisant hériter notre classe `Personne` du parent `yaml.YAMLObject` comme suit :

```
#!/usr/bin/env python
# encoding: UTF-8

import yaml

class Personne(yaml.YAMLObject):

    yaml_tag = '!personne'

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def __repr__(self):
```

```
            return "%s(nom=%r, age=%r)" % \
                   (self.__class__.__name__, self.nom, self.age)

print yaml.dump(Personne('Robert', 25), default_flow_style=False)
```

Cela produit sur la console :

```
!personne
age: 25
nom: Robert
```

Bien sûr, on peut désérialiser cette structure YAML en utilisant la fonction `yaml.load()`. Cependant, cette notation, qui comporte le tag YAML `!personne` est bien plus élégante que la précédente, qui indique la nom qualifié de la classe et non un nom symbolique plus court et plus parlant.

Il y a un peu de magie derrière tout cela : la classe `Personne` s'enregistre en tant que type Python associé au tag YAML `!personne` ce qui permet cette notation élégante.

Il existe un moyen d'associer une expression rationnelle à une classe Python de manière à ce que, par exemple, la notation `3d6` soit associée à l'appel au constructeur `Dice(3, 6)`. Je vous laisse le soin de creuser la question dans la section adéquate de la documentation de PyYaml.

Goyaml

Goyaml est une bibliothèque Go permettant de parser des fichiers YAML pour en injecter le contenu dans des structures définies par l'utilisateur. Cette méthode de parsing est généralisée en Go : elle est identique à celle du XML ou du JSON par exemple.

Installation

Pour installer la bibliothèque dans son *GOPATH*, on tapera la ligne de commande suivante :

```
$ go get gopkg.in/yaml.v2
```

Définition de la structure

Supposons que nous souhaitions parser le fichier YAML suivant, qui représente un utilisateur :

```
name: Robert
age:  25
```

Je pourrais définir la structure suivante pour représenter cet utilisateur :

```
type User struct {
  Name string
  Age  int
}
```

Parsing du fichier

Pour parser le fichier, il faut :

- Créer une structure vide.
- Lire le contenu du fichier YAML.
- Parser le contenu du fichier en passant l'adresse de la structure vide.

Ainsi pour parser notre fichier d'utilisateur :

```
var user User
source, err := ioutil.ReadFile("user.yml")
if err != nil {
  panic(err)
}
err = yaml.Unmarshal(source, &user)
if err != nil {
  panic(err)
}
fmt.Printf("user: %v\n", user)
```

Cette approche est très simple et répond à la plupart des cas d'usage du parsing YAML.

Tags de structure

Il est possible de taguer la structure afin de préciser le comportement du parseur. Par exemple, pour indiquer un nom alternatif pour le champ, on ajoutera la tag suivant :

```
type User struct {
  Nom string `yaml:"name"`
  Age int
}
```

On indique ici que le nom du champ est *name* dans le source YAML alors que le nom dans structure laisse supposer que ce doit être *nom*.

La forme générale du tag est la suivante : `yaml:"[<key>][,<flag1>[,<flag2>]]"`, où *key* est le nom du champ et *flag* peut avoir les valeurs suivantes :

- **omitempty** indique que le champ doit être omis si le champ est vide.
- **flow** sérialise en utilisant le style en ligne (les listes seront alors représentées par `[1, 2, 3]` et les maps par `{foo: 1, bar: 2}` par exemple).

- **inline** injecte la structure à laquelle il est appliqué de sorte que ses champs sont traités comme s'ils appartenaient à la structure englobante.

Usage avancé

Il semble impossible de parser une fichier YAML quelconque avec la technique décrite ci-dessus, mais il n'en est rien. En effet, il est possible d'écrire le code suivant :

```
var thing interface{}
source, err := ioutil.ReadFile("user.yml")
if err != nil {
  panic(err)
}
err = yaml.Unmarshal(source, &thing)
if err != nil {
  panic(err)
}
fmt.Printf("Thing: %#v\n", thing)
```

Ce qui produit la sortie suivante :

```
$ go run generic.go user.yml
Thing: map[interface {}]interface {}{"name":"Robert", "age":25}
```

Comme `interface{}` désigne un type quelconque, on peut parser ainsi tout fichier YAML. Il faudra ensuite introspecter le résultat du parsing avec le package *reflect*, mais ceci est une autre histoire (bien plus compliquée...).

On pourra voir un exemple de mise en œuvre de cette technique dans mon projet NeON. A noter que l'usage du package *reflect* est à réserver à des utilisateurs expérimentés du langage Go, sous peine d'en perdre la raison.

Conclusion

Maintenant que nous avons une bonne idée de ce qu'est YAML, nous pouvons le comparer à des technologies proches telles que JSON et XML.

YAML et JSON

Ces deux formats de représentation textuelle de données sont très proches, à tel point qu'à partir de la version *1.2* de la spécification YAML, tout document JSON est un document YAML valide (et peut donc être parsé par un parser YAML conforme à la version *1.2* de la spécification).

Cependant, YAML bénéficie d'une plus grande lisibilité. D'autre part, il n'est pas lié à un langage de programmation particulier (comme l'est JSON avec JavaScript).

YAML et XML

Ces deux technologies sont assez différentes et YAML ne peut pas faire tout ce que peut faire XML. En particulier, YAML n'est pas adapté comme format de texte structuré, on ne pourrait donc pas remplacer XML par YAML pour écrire du DocBook par exemple.

Par contre, dans le domaine de sérialisation de données, YAML est bien plus spécialisé et puissant de par sa reconnaissance des types primitifs usuels et des structures de données telles que listes et tableaux associatifs.

D'autre part, YAML possède un énorme avantage en ce qui concerne la syntaxe et sa lisibilité, même par une personne qui n'a pas connaissance des spécification YAML. Ainsi, dans le domaine des fichiers de configuration qui doivent être manipulés par des gens qui n'ont pas de connaissance particulière, la syntaxe naturelle de YAML est-elle un énorme avantage.

Utilisation de YAML

Les deux utilisations principales de YAML sont :

- Les fichiers de configuration. YAML permet des fichiers de configuration typés et d'une syntaxe naturelle. Les fichiers de configuration de Ruby on Rails sont en YAML.
- La sérialisation de données. Il peut être commode d'échanger des données au format YAML dans la mesure où ce format est indépendant de la plateforme et du langage de programmation. La présence d'alias et d'ancres permet des sérialisations intelligentes.

J'espère que cette présentation de YAML vous aura donné l'envie d'utiliser ce format de données dans vos propres applications et de répandre la bonne parole autour de vous !

Ressources

Voici quelques URLs utiles relatives à YAML :

- Page d'accueil YAML.
- Spécification YAML 1.2.
- Carte de référence YAML.
- Page d'accueil JYaml.
- Tutoriel JYaml.
- Références de l'API JYaml.
- Page d'accueil PyYaml.
- Documentation PyYaml.
- Pade d'accueil GoYaml.
- Documentation GoYaml.

Enjoy!