Introduction to YAML

Michel Casabianca
casa@sweetohm.net

This article is an introduction to YAML, a language to write structured data, as XML would for example, but in a more natural and less verbose way. We will see a description of the YAML syntax as well as examples in Java, Python and Go.

Une version en français est disponible ici.

A ZIP archive with source code for examples can be found at:
http://www.sweetohm.net/arc/introduction-yaml.zip.

What is YAML?

The name YAML means **YAML Ain't Markup Language**. If this immediately distances XML, it does not tell us what YAML is. YAML is, according to its specification, a data serialization language designed to be human-readable and working well with modern programming languages for everyday tasks.

Specifically, we could note the list of ingredients for a breakfast as follows:

```
- croissants
- chocolate breads
- ham
- eggs
```

This is a valid YAML file that represents a list of strings. To be convinced, we can write the following Python script that parses the file, whose name is passed on the command line, and displays the result:

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

import sys
import yaml

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

This script will produce the following result:

```
['croissants', 'chocolate breads', 'ham', 'eggs']
```

Which means that the result of this parsing is a Python list containing the appropriate strings! The parser is therefore able to render *natural* data structures of the language used for parsing.

Now let's write a countdown:

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

We obtain the following result:

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

This is always a list, but the parser has recognized each element as an integer and not a simple string as in the previous example. The parser is therefore able to distinguish data types such as strings and integers, floating-point numbers, and dates. To do this, we use a natural syntax: *3* is recognized as an integer while *ascending* is not because the integer cannot be converted to an integer, a floating point number, or any other type recognized by YAML. To force YAML to interpret *3* as a string, you can surround it with quotation marks.

YAML can also recognize associative arrays, so we could note a breakfast order as follows:

```
croissants: 30
chocolate breads: 30
ham: 0
eggs: 0
```

Which will be loaded in the following way:

```
'chocolate breads': 30, 'croissants': 30, 'ham': 0, 'eggs': 0}
```

By combining the basic data types in YAML-recognized collections, almost any data structure can be represented. On the other hand, the textual representation of these data is very legible and almost natural.

It is also possible to perform the reverse operation, namely to serialize data structures in memory as text. In the following example, we write on standard output a Python `Dictionary`:

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

import yaml

recipe = {
    'name': 'sushi',
    'ingredients': ['rice', 'vinegar', 'sugar', 'salt', 'tuna', 'salmon'],
```

```
    'cooking time': 10,
    'difficulty': 'difficult'
}

print yaml.dump(recipe)
```

Which will produce the following output:

```
difficulty: difficult
ingredients: [rice, vinegar, sugar, salt, tuna, salmon]
name: sushi
cooking time: 10
```

Basic syntax

After this brief introduction, here is a more exhaustive description of the YAML syntax.

Scalar

Scalars are the set of YAML types that are not collections (list or associative array). They can be represented by a list of Unicode characters. Here is a list of scalars recognized by YAML parsers:

String of characters

Here is an example :

```
- String
- "3"
- String on
  a single line
- "Double quotation marks\t"
- 'Single quotation marks\t'
```

Which is parsed in the following way:

```
[u'String', '3', u'String on a single line',
  'Double quotation marks\t', 'Simple quotation marks\\t']
```

The result of this parsing leads us to the following comments:

- Accented characters are handled, and more generally, Unicode is managed.
- Line breaks are not taken into account in strings, they are managed as in HTML or XML, namely they are replaced by spaces.
- Double quotation marks handle escape characters, such as \t for tabulation for instance.
- Single quotes do not handle escape characters that are transcribed literally.

- The list of escape characters managed by YAML includes the classical values, but also many others that can be found in the YAML specification.

On the other hand, it is possible to write Unicode characters using the following notations:

- `\xNN`: to write 8-bit Unicode characters, where `NN` is a hexadecimal number.
- `\uNNNN`: for 16-bit Unicode characters.
- `\UNNNNNNNN`: for 32-bit Unicode characters.

Integers

Here are some examples:

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

Which is parsed in the following way:

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

Thus most common notations of programming languages (such as octal or hexadecimal) are handled. Note that all these notations will be recognized as identical by a YAML parser and consequently will be equivalent like key of an associative array for example.

Floating point numbers

Let's see the different notations for these numbers:

```
canonical:         1.23015e+3
exponential:       12.3015e+02
sexagesimal:       20:30.15
fixed:             1_230.15
infinite negative: -.inf
not a number:      .NaN
```

Which is parsed in:

```
{'not a number': nan, 'sexagesimal': 1230.1500000000001,
 'exponential': 1230.1500000000001, 'fixed': 1230.1500000000001,
 'infinite negative': -inf, 'canonical': 1230.1500000000001}
```

Classical notations are handled as well as infinites and values that are not numbers.

Dates

YAML also recognizes dates:

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

Which are parsed in the following way:

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

The types resulting from the parsing depend on the language and the parser, but correspond to natural types for the considered times.

Miscellaneous

There are other scalars recognized by YAML parsers:

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

Who will be parsed in:

```
{'false bis': False, 'true ter': True, 'true bis': True,
 'false ter': False, 'null': None, 'false': False,
 'null bis': None, 'true': True}
```

Of course, the type of parsed values depends on the language and other special values can be recognized according to the languages and parsers. For example, the Ruby parser recognizes symbols (denoted `: symbol` for instance) and parses them into Ruby symbols.

Collections

There are two types of collections recognized by YAML: lists and associative arrays.

Lists

These are ordered lists, and may contain several identical elements (as opposed to sets). The elements of a list are identified by a hyphen, as follows:

```
- butter
  croissants
- chocolate breads
- ham
- eggs
```

The elements of the list are distinguished thanks to the indentation: the first element is identified so that its second line is recognized as part of the first element of the list. This syntax can be compared to that of Python, except that in YAML, **tab characters are strictly forbidden for indentation**. This last rule is important and causes many parsing errors. It is important to set its editor to prohibit tabs for indenting YAML files.

There is an alternative notation for lists, similar to that of Python or Ruby languages:

```
[croissants, chocolate breads, ham, eggs]
```

This notation, called *in flow*, is more compact and sometimes allows to gain readability or compactness.

Associative Tables

Called *Maps* or *Dictionaries* in some languages, they associate a value with a key:

```
croissants: 2
chocolate breads: 1
ham: 0
eggs: 3
```

Flow notation is as follows:

```
{croissants: 2, chocolate breads: 1, ham: 0, eggs: 3}
```

Which is parsed in the same way. This notation is identical to that of Python or Javascript and is similar to that used by Ruby. Note that Ruby 2 is also using this notation.

Comments

It is possible to include comments in a document in the same way as in most scripting languages:

```
# comment
- Some text
# other comment
- Other text
```

Note that these comments must not (and can not) contain useful parsing information since they are not generally accessible to the parser client code.

Multiple Documents

In the same file or stream, you can insert several YAML documents afterwards, starting them with a line consisting of three dashes (`---`) and terminating them with a line of three dots (`...`) as in the example below:

```
---
first document
...
---
second document
...
```

Note that by default, YAML parsers wait for one document per file and may issue an error if they encounter more than one document. It is then necessary to use a particular function able to parse multiple documents (like `yaml.load_all()` for PyYaml for instance).

These documents can then be extracted sequentially from the stream.

Advanced syntax

In previous section, we saw the bare minimum to get by with YAML. We will now deal with more advanced notions that are not necessary in the first place.

References

YAML references are similar to pointers in programming languages. For instance:

```
Monday:    &p 'potatoes'
Tuesday:   *p
Wednesday: *p
Thursday:  *p
Friday:    *p
Saturday:  *p
Sunday:    *p
```

Which gives while parsed:

```
{'Tuesday': 'potatoes', 'Saturday': 'potatoes',
 'Thursday': 'potatoes', 'Monday': 'potatoes',
 'Friday': 'potatoes', 'Sunday': 'potatoes',
 'Wednesday': 'potatoes'}
```

Note that an alias, indicated by an asterisk `*`, must point to a valid anchor, indicated by an ampersand `&`, otherwise it results in a parsing error. So the following file will cause an error during parsing:

```
*foo
```

tags

Tags are data types indicators. By default, it is not necessary to indicate the type of data that is deduced from their form. However, in some cases, it may be necessary to force the data type, and YAML defines following default types:

```
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 ]
set:     !!set    { element1, element2 }
omap:    !!omap   [ key: value ]
```

Matching tags begin with two exclamation points. When parsing, we get following types in Python:

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

Note two additional types:

- The set: not ordered and cannot contain duplicates.
- The ordered associative array: it is an associative array whose entries are ordered.

The usefulness of tags for these default types is limited. The real power of tags lies in the ability to define your own tags for your own data types.

For instance, one could define his own type for people, with two fields: first name and last name. We must first declare the tag at the beginning of the document, then we can use it in the document, as in this example:

```
%TAG !people! tag:foo.org,2004:bar
---
- !people
    first name: Omer
    last name:  Simpson
- !people
    first name: Bart
    last name:  Simpson
```

We will see later how to use tags with Java and Python APIs to deserialize YAML structures into custom data types.

It is also possible not to declare the tag and to explain it in the document, as follows:

```
- !<tag:foo.org,2004:bar>
    first name: Omer
    last name:  Simpson
- !<tag:foo.org,2004:bar>
    first name: Bart
    last name:  Simpson
```

Directives

The directives give instructions to parser. There are two of them:

TAG

As seen previously, declare a tag in the document.

YAML

Indicates the YAML version of the document. Must be in the header of the document, as in the example below:

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

A parser must refuse to process a document of a higher major version. For example, a parser in version `1.1` should refuse to parse a document in YAML version `2.0`. It should issue a warning if it is asked to parse a document of minor higher version, such as `1.2` for instance. It must parse without protesting all versions equal or inferior, such as `1.1` and `1.0`.

Character sets and encoding

A YAML parser must accept any Unicode character, except for some special characters. These characters can be encoded in *UTF-8* (default encoding), *UTF-16* or *UTF-32*. YAML parsers are able to determine the encoding of the text by examining the first character. It is therefore **impossible** to use any other encoding in a YAML file and in particular `ISO-8859-1`.

YAML APIs

We will now play with YAML APIs for Java, Python and Go.

JYaml

JYaml is an OpenSource library for manipulating YAML documents in Java. The project is hosted by SourceForge and can be found at http://jyaml.sourceforge.net/. You will find on this site a tutorial and the API References.

Basic Usage

JYaml performs a default mapping of YAML structures to standard Java objects: it associates a list with an instance of `java.util.ArrayList`, an associative array with an instance of `java.util.HashMap` and YAML primitive types to their Java counterpart.

Thus, to load a YAML file in a Java object, we will write the following code:

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

For instance, following file:

```
- One
- 2
- { three: 3.0, four: true }
```

Can be loaded into memory and displayed in the terminal with following code:

```
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)));
```

```
        }

    }
```

This will print in the terminal:

```
    [One, 2, {four=true, three=3.0}];
```

Conversely, you can serialize a Java object in a YAML file as follows:

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

Thus, we can serialize a Java object structure with following code:

```
    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("One");
            object.add(2);
            Map<String, Object> map = new HashMap<String, Object>();
            map.put("three", 3.0);
            map.put("four", true);
            object.add(map);
            Yaml.dump(object, new File("test/dump.yml"));
        }

    }
```

This code will produce following file:

```
    ---
    - One
    - 2
    - !java.util.HashMap
      four: true
      three: !java.lang.Double 3.0
```

Note that this dump is a little disappointing because some standard types of YAML (such as associative arrays and floating point numbers) are serialized to Java types (such as `java.util.HashMap` and `java.lang. Double`). Such a file will not be loaded correctly using another programming language (or even another implementation in Java).

Advanced use

We can also work with types that are not generic and thus load instances of Java classes from YAML files.

The first solution is to indicate the type of objects with YAML tags. Thus, the following YAML file:

```
--- !jyaml.Order
id: test123
Articles:
 - !jyaml.Article
    id:       test456
    price:    3.5
    quantity: 1
 - !jyaml.Article
    id:       test567
    price:    2.0
    quantity: 2
```

Will it be loaded using the following classes:

```
package jyaml;

public class Order {

    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("[Order 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();
        }

    }
```

And:

```
    package jyaml;

    public class Article {

        private String id;
        private double price;
        private int quantity;

        public String getId() {
            return id;
        }

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

        public double getPrice() {
            return price;
        }

        public void setPrice(double price) {
            this.price = price;
        }

        public int getQuantity() {
            return quantity;
        }

        public void setQuantity(int quantity) {
            this.quantity = quantity;
        }

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

    }
```

These classes respect the JavaBean convention, namely that they have accessors for its fields as well as an empty constructor (without arguments, implicit in Java). By loading it with the previous source, we get on the terminal:

```
[Order id='test123', articles='[
  [Article id='test456', price='3.5', quantity='1'],
  [Article id='test567', price='2.0', quantity='2']
  ] '
]
```

There is another way to load these objects without the need to explicitly specify types. To do this, use the `loadType()` method and pass it the YAML file to load as well as the type of the root object of the file. So, in our case, we could write the order file as follows:

```
id: test123
articles:
 - id:       test456
   price:    3.5
   quantity: 1
 - id:       test567
   price:    2.0
   quantity: 2
```

And load it with following source:

```
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/order2.yml"),
                                         Order.class));
    }

}
```

This way of loading specific types is much more convenient because it does not overload the YAML file with Java types, which makes the portability between languages impossible. However, we may regret the obligation to declare the `articles` field as an `Article` array. If declared as `List <Article>`, JYaml loads the objects in the list as instances of `Map`. However, this is because the list type information is lost in the runtime and therefore JYaml cannot know the type of items in the list and therefore loads them with the default type.

Aliases and anchors

JYaml handles aliases and anchors for YAML files. Let's consider following example:

```
- &rob
  name: Robert
```

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

It can be loaded with following code:

```
package jyaml;

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

public class Alias   {

    public static void main(String[] args)
        throws Exception {
        People[] people = Yaml.loadType(new File("test/alias.yml"),
                                        People[].class);
        for (int i=0; i<people.length; i++) {
            People individual = people[i];
            System.out.println(individual);
        }
        // we test that references are identical
        System.out.println("Anchor OK:"+(people[2].getParents()[0]==people[0]));
    }

}
```

And we see that the aliases and anchors have been handled correctly.

Flow management

It is possible to serialize Java objects into a YAML stream as follows:

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

There is a shortcut to serialize a collection of objects in a stream as follows:

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

On the other hand, you can deserialize Java objects from a YAML stream as follows:

```
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();
}
```

There is a shortcut to iterate on deserialized objects in a stream:

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

Configuration file

It is possible to configure JYaml in a file, which must be named `jyaml.yml` and be in current directory where the application runs or at the root of its *CLASSPATH*. Here is an example of such a file:

```
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
```

This configuration makes it possible in particular to configure mappings between YAML tags and Java classes. In the example above, the following mapping:

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

Allows you to shorten the YAML tag to indicate the Java class used for deserialization. Thus, we can indicate the mapping for the `com.blah.Company` class by a simple `!Company` tag.

On the other hand, this file is used to list wrappers classes that deserialize particular types whose classes do not respect the JavaBeans convention. Examples can be found in the project sources.

Other features

It should be noted that JYaml has more to offer, in particular:

- Support for YAML Spring configuration files. This allows you to write more readable and compact files than their XML counterparts. For more information, see this page.
- A YAML format for DBUnit configuration files. See this page.

However, JYaml seems to suffer from limitations, especially for parsing dates that are not always recognized as such.

PyYAML

PyYaml is a Python library to manage YAML files. It can be downloaded from http://pyyaml.org/ and you can find documentation on this page.

Installation

PyYaml can use LibYaml written in C and very fast or a Python implementation that does not require this library. To install the library, download the archive, unzip it, go to created directory and type `python setup.py install`, or `python setup.py --wihout-libyaml install` if you don't want to use LibYaml.

Basic Usage

To load a YAML file, whose name is passed on command line, we can proceed as follows:

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

import sys
import yaml

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

The `load()` function takes a string of bytes, unicode, binary or text as parameter. Byte strings and files must be encoded in *UTF-8* or *UTF-16*. Encoding is determined by the parser by examining the Byte Order Mark (BOM), the first byte of the file. If no BOM is found, *UTF-8* is chosen.

If the string or file contains multiple documents, we can load them all using the `yaml.load_all()` function that returns an iterator. We can write:

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

To serialize a Python object into YAML, we can use the `yaml.dump()` function:

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

import yaml

recipe = {
    'name': 'sushi',
    'ingredients': ['rice', 'vinegar', 'sugar', 'salt', 'tuna', 'salmon'],
    'cooking time': 10,
    'difficulty': 'difficult'
}

print yaml.dump(recipe)
```

The `yaml.dump()` function can take a second optional parameter which must be a binary or open text file. It then writes the result of serialization to the file.

To serialize multiple Python objects into a stream, you can use my `yaml.dump_all()` function. This function takes a list or an iterator as parameter.

The dump functions take additional parameters to indicate formatting details of generated YAML. We can thus specify the number of indentation characters to use, the number of characters per line, etc. In particular, `default_flow_style` indicates whether we use stream style for lists and associative arrays. So, following code:

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

Produces a representation using flow notation:

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

While following code:

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

Produces a list notation:

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

Serialization and deserialization of Python classes

It is possible to explicitly declare the Python type to be used to deserialize a given YAML structure using a YAML tag. For example :

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

import yaml

class Person(object):

    def __init __(self, name, age):
        self.name = name
        self.age = age

    def __repr __(self):
        return "%s(name=%r, age=%r)" % \
               (self.__class__.__name__, self.name, self.age)

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

Produces following output on the terminal:

```
Person(name='Robert', age=25)
```

Conversely, a Python class can be serialized into a YAML stream as follows:

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

import yaml

class Person(object):

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

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

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

This code produces following output:

```
!!python/object:__main__.Person
age: 25
name: Robert
```

We see that PyYaml serialized the Python class using the same YAML tag notation. Note that building arbitrary Python objects from unreliable sources (typically from the internet) can be dangerous. This is why PyYaml offers the `yaml.safe_load()` function which limits the building of objects to the basic types of YAML.

The notation seen above allows to deserialize YAML structures into instances of arbitrary Python classes. However, it is possible to use a simpler notation by inheriting our `Person` class from the parent `yaml.YAMLObject` as follows:

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

import yaml

class Person (yaml.YAMLObject):

    yaml_tag = '!person'

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

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

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

This produces on the console:

```
!person
age: 25
name: Robert
```

Of course, we can deserialize this YAML structure by using the `yaml.load()` function. However, this notation, which has the YAML `!person` tag, is much more elegant than previous one, which indicates the qualified name of the class and not a shorter, more meaningful symbolic name.

There is a bit of magic behind all this: the `Person` class is registered as a Python type associated with the YAML `person` tag which allows this elegant notation.

There is a way to associate a regular expression with a Python class so that, for example, the notation `3d6` is associated with a call to the constructor `Dice(3, 6)`. I leave you to dig deeper in the appropriate section of the PyYaml documentation.

Goyaml

Goyaml is a Go library for parsing YAML files to inject content into user-defined structures. This method of parsing is generalized in Go: it is identical to that of XML or JSON for example.

Installation

To install the library in your *GOPATH*, you should type following command line:

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

Definition of the structure

Suppose we want to parse the following YAML file, which represents a user:

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

I could define following structure to represent this user:

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

Parsing the file

To parse the file, you must:

- Create an empty structure.
- Read the contents of the YAML file.
- Parse the contents of the file by passing the address of the empty structure.

So to parse our user file:

```
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)
```

This approach is very simple and addresses most YAML parsing use cases.

Structure tags

It is possible to tag the structure in order to specify the behavior of the parser. For instance, to

specify an alternative name for the field, add following tag:

```
type User struct {
  Name string `yaml:"first_name"`
  Age  int
}
```

It indicates that the name of the field is *first_name* in YAML source file while the name in structure suggests that it should be *name*.

The general form of the tag is: `yaml:"[<key>][,<flag1>[,<flag2>]]`, where *key* is the name of the field and *flag* can have following values:

- **omitempty** indicates that the field must be omitted if field is empty.
- **flow** serializes using the inline style (lists will then be represented by `[1, 2, 3]` and the maps by `{foo: 1, bar: 2}` for example).
- **inline** injects the structure to which it is applied so that its fields are treated as if they belonged to the enclosing structure.

Advanced usage

It seems impossible to parse an abitrary YAML file with the technique described above, but it is false. Indeed, it is possible to write following code:

```
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)
```

Which produces following output:

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

Since `interface{}` designates any type, we can parse any YAML file. You will have to introspect the result of the parsing with *reflect* package, but this is another story (much more complicated ...).

An example of implementation of this technique can be seen in my NeON project. Note that the use of the *reflect* package is to be reserved for experienced users of the Go language, otherwise they might lose their sanity.

Conclusion

Now that we have a good idea of what YAML is, we can compare it to similar technologies such as JSON and XML.

YAML and JSON

These two formats of textual representation of data are very close, so much so that starting from the *1.2* version of the YAML specification, any JSON document is a valid YAML document (and can therefore be parsed by a YAML parser conforming to version *1.2* of the specification).

However, YAML has a greater readability. On the other hand, it is not tied to a particular programming language (as is JSON with JavaScript).

YAML and XML

These two technologies are quite different and YAML can not do everything XML can do. In particular, YAML is not suitable as a structured text format, so we could not replace XML by YAML to write DocBook for example.

On the other hand, in the field of serialization of data, YAML is much more specialized and powerful by its recognition of the usual primitive types and data structures such as lists and associative arrays.

Furthermore, YAML has a huge advantage in terms of syntax and readability, even by someone who is not familiar with the YAML specification. So, in the domain of configuration files that need to be manipulated by people who have no particular knowledge, is YAML's natural syntax a huge advantage.

YAML Usage

The two main uses of YAML are:

- Configuration files. YAML allows typed configuration files and natural syntax. The Ruby on Rails configuration files are in YAML format.
- Serialization of data. It may be convenient to exchange data in YAML format as this format is independent of platform and programming language. The presence of aliases and anchors allows intelligent serializations.

I hope this YAML presentation has given you the urge to use this data format in your own applications and spread the good word around you!

Resources

Here are some useful URLs for YAML:

- YAML Homepage.
- YAML 1.2 Specification.
- YAML Reference Card.
- JYaml Home Page.
- JYaml Tutorial.
- JYaml API References.
- PyYaml Homepage.
- PyYaml Documentation.
- GoYaml Homepage.
- GoYaml Documentation.

Enjoy!