XML2RB

Table of Contents

XML2RB

Executing XML

Fri May 25 15:06:34 UTC 2007

Erik Veenstra <erikveen@dds.nl>

1. Introduction

Like all of you, I make my money in the Java world. There's no money in the Ruby world. Life isn't perfect.

Like all of you, I see a lot of XML in the Java world. XML for configurations. XML for templates. XML for flows. XML for build scripts. XML left, XML right. Whatever your problem is, XML is the solution!

Like all of you, I hate XML. Well, uh, to be more specific... XML is great for storing data, or exchanging data. If you store data in one language and you want to read it with another, well, yes, XML really is great. Do you need to transfer data over a network to whatever kind of machine? Go for XML (if you do have enough bandwidth...). But I hate reading XML and I definitely hate writing XML. First and for all, XML is for *machines*, not for *human beings*.

Like all of you, I like coding in Ruby. I really do. Ruby is for human beings. Ruby is slow? I don't care: Ruby is not for machines, Ruby is for real people. To let them do their job quickly and with fun. I really, really like coding in Ruby. I would do it for free! Which is good, since there's no money in the Ruby world...

So, XML is *not* appropriate for configurations. XML is *not* appropriate for templates. XML is *not* appropriate for flows. XML is *not* appropriate for build scripts.

In the good old days of LISP, code was data which could be executed. Sometimes XML is data, and sometimes it's code. Sometimes it's even mixed. Have a look at ANT scripts: `<description>...</description>` is data, `<javac>...</javac>` is code, `<target>...</target>` is a definition of code, `<fileset>...</fileset>` is code which results in data.

Talking about ANT: ANT is bad. I don't mean the properties, which can only be set once. (What do you expect when you see this: `x=7;print(x)`? 7? Wrong!) I don't mean the functions (called "targets"), which simply do not receive parameters. (Well, not the way you expect...). I don't mean the keywords, which should be written twice (`<javac>` and `</javac>`). The real problem with ANT is this: It's written on top of XML.

This week, I had to track down a bug in a set of ANT scripts. It took me hours. I simply couldn't build a mental picture of what was going on. I was angry. "I don't want to read XML!", I shouted. "XML is for machines! I'm not a machine! I want to write Ruby!" Java Joe answer: "Go write Ruby, I don't care, but please shut up!". Just to make him happy, I immediately started writing Ruby...

I wrote a little script to analyze these ANT scripts. After all, they're just XML and can be treated as data. The parts which should be treated as code are fed to a little interpreter, which walked through the data, step-by-step. Something started to smell... Although I was *writing* Ruby, I was still handling XML and, above all, I was still *thinking* XML.

Can't we just translate XML to Ruby code? In theory we could: assume that we translate every tag in XML to a method call in Ruby, than the attributes of the tag are the named parameters of the call and the body of the tag (child tags and/or text) is handled by the block.

1. Introduction

Let's take this XML:

```
<project name="Freenet" default="dist" basedir=".">
  <description>
    This file builds freenet...
  </description>
</project>
```

It can be translated to Ruby, almost line-by-line:

```
project(:default=>"dist", :basedir=>".", :name=>"Freenet") {
  description {
    text! "\n    This file builds freenet...\n  "
  }
}
```

To avoid conflicts with Ruby keywords, like `class` and `module`, we can write this:

```
project(:default=>"dist", :basedir=>".", :name=>"Freenet") {|project1|
  project1.description {|description1|
    description1.text! "\n    This file builds freenet...\n  "
  }
}
```

(It's not really what we want, but, hey, it resembles XML, because it was generated from XML! Shit in, shit out...)

This Ruby code could run if we write a DSL library: `ruby -r ant build.rb`

Instead of writing an interpreter and coding against REXML, we only have to build the DSL. Or can it be generated?...

Let's turn theory into practice. After all, that's the best proof of a concept. "In theory, there's no difference between theory and practice; in practice, there is."

This is just an experiment. Maybe, after a couple of days, I say: "Stupid idea!" Maybe not. Let's find out...

2. Day One

Here are the first lines of the build script I'll use as demo. It's stolen from Freenet.

```
$ cat build.xml | head
<?xml version="1.0"?>
<project name="Freenet" default="dist" basedir=".">
        <description>
                This file builds freenet...
                Possible targets: compile, dist (default), clean
        </description>

        <property name="src" location="src"/>
        <property name="build" location="build"/>
        <property name="lib"    location="lib"/>
```

Can we turn this XML into Ruby code? Yes, we can. We can do it almost lexically, line-by-line (although I do use REXML):

```
$ cat build.xml | xml2rb | head
project(:default=>"dist", :basedir=>".", :name=>"Freenet") {|project1|
  project1.description {|description1|
    description1.text! "\n    This file builds freenet...\n    Possi.....
  }
  project1.property(:name=>"src", :location=>"src")
  project1.property(:name=>"build", :location=>"build")
  project1.property(:name=>"lib", :location=>"lib")
  project1.property(:name=>"javadoc", :location=>"javadoc")
  project1.property(:value=>"@custom@", :name=>"svn.revision")
```

We can simplify this code. If we're going to `instance_eval` each block in the context of the object itself, instead of calling the block with the object as argument:

```
$ cat build.xml | xml2rb --simple | head
project(:default=>"dist", :basedir=>".", :name=>"Freenet") {
  description {
    text! "\n    This file builds freenet...\n    Possi.....
  }
  property(:name=>"src", :location=>"src")
  property(:name=>"build", :location=>"build")
  property(:name=>"lib", :location=>"lib")
  property(:name=>"javadoc", :location=>"javadoc")
  property(:value=>"@custom@", :name=>"svn.revision")
```

(Although this code is better readable and looks more like the original XML, we run into trouble if Ruby keywords (like `class` and `module`) are used in the xml. So, we continue with the less-readable code.)

Can we execute this code?

```
$ cat build.xml | xml2rb | ruby
-:1: undefined method 'project' for main:Object (NoMethodError)
```

Uh, no, we can't. We have not yet implemented the DSL library with which the code should run. We could implement this DSL by hand, but, since good programmers are lazy, we will use another tool to generate at least the base definitions of this DSL. This tool uses `const_missing` and

`method_missing` to find the necessary classes and methods empirically, after which it generates the DSL base library.

(In theory, we should read the specs and build this base library by hand. In practice, we simply use all ANT scripts we have on our machine (527 on mine!) to generate it automatically.)

```
$ cat build.xml | xml2rb | xml2rb2dsl > ant_base.rb
Analyzing stdin...
```

Can we run it now?

```
$ cat build.xml | xml2rb | ruby -r ant_base.rb
```

Success! But there's no output! That's because the generated DSL only defines default (close to empty) methods, without the real stuff. It's time to write some code. (Not too much, please...)

```
$ cat > ant_description.rb
class Description
  def text!(s)
    puts s
  end
end
```

This overwrites the default method `Description#text!`, so it prints the text *instead of* adding the text to the internally built tree.

(We can avoid this "instead of". Instead of overwriting the method, we should add some code to the already existing method. Sounds scary? Well, maybe it is, but it can be done. How? I'll tell you on day two...)

```
$ cat build.xml | xml2rb | ruby -r ant_base.rb -r ant_description.rb

               This file builds freenet...
               Possible targets: compile, dist (default), clean
```

Voilà! Result! How much did we code?

3. Day Two

On Day One, we defined this `ant_description.rb`. There was also a note about *adding code to an already existing method*. We can add code to an existing method by using `Module.pre_condition` and `Module.post_condition`, as defined and explained here.

```
$ cat > ant_description.rb
class Description
  pre_condition(:text!) do |obj, method_name, args, block|
    puts args[0]
  end
end
```

Does this work?

```
$ cat build.xml | xml2rb | ruby -r ant_base.rb -r ant_description.rb

               This file builds freenet...
               Possible targets: compile, dist (default), clean
```

Yes, it does.

The point of *adding code to a method* instead of *overwriting a method* is obvious: `xml2rb2dsl` generates the base declarations of the DSL. If we overwrite these methods, we have to reimplement the generated code (`super` won't work). If we simply add code to an already existing method, either before (`Module#pre_condition`) or after (`Module#post_condition`), we can reuse the generated code. If adding code to either end of the method isn't going to work, we could use the more flexible `Module#wrap_method` (defined and explained here).

We've seen before, that `<description>...</description>` is translated to something like `project1.description{|description1| description1.text! "..."}`. As you see, we have a class `Description`, as well as a method `Project#description` (the latter instantiated the former). In the `previous` version of `ant_description.rb`, we added code to `Description#text!`. Now we are going to add code to *the end of* `Project#description`:

```
$ cat > ant_description.rb
class Project
  property :description

  post_condition(:description) do |obj, method_name, args, block|
    obj.instance_eval do
      puts @description.text!
    end
  end
end
```

By default, all nodes in the tree have *attributes*, *properties* and *text*. Since an ANT project only has one description, we can replace the default `properties :description` with `property :description`. This means that `Project@description` is a pointer to aDescription, instead of to an array of aDescription's.

Since the block given to `post_condition` is executed in the original context (in casu in the context `Project`), but we have to read `@description` in the context of aProject, we have to use this `obj.instance_eval{}`.

Does this work?

```
$ cat build.xml | xml2rb | ruby -r ant_base.rb -r ant_description.rb

               This file builds freenet...
               Possible targets: compile, dist (default), clean
```

Yes, it still does.

We've seen three different ways to find and print the description of a project in an ANT script. On day three, we're going to export the internal tree to... XML!

4. Day Three

(To be continued...)