Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

11. XML and HTML

XML and HTML are the most popular markup languages (textual ways of describing structured data). HTML is used to describe textual documents, like you see on the Web. XML is used for just about everything else: data storage, messaging, configuration files, you name it. Just about every software buzzword forged over the past few years involves XML.

Java and C++ programmers tend to regard XML as a lightweight, agile technology, and are happy to use it all over the place. XML *is* a lightweight technology, but only compared to Java or C++. Ruby programmers see XML from the other end of the spectrum, and from there it looks pretty heavy. Simpler formats like YAML and JSON usually work just as well (see Recipe 13.1 or Recipe 13.2), and are easier to manipulate. But to shun XML altogether would be to cut Ruby off from the rest of the world, and nobody wants that. This chapter covers the most useful ways of parsing, manipulating, slicing, and dicing XML and HTML documents.

There are two standard APIs for manipulating XML: DOM and SAX. Both are overkill for most everyday uses, and neither is a good fit for Ruby's code-block–heavy style. Ruby's solution is to offer a pair of APIs that capture the style of DOM and SAX while staying true to the Ruby programming philosophy.[1] Both APIs are in the standard library's REXML package, written by Sean Russell.

[1] REXML also provides the `SAX2Parser` and `SAX2Listener` classes, which implement the basic SAX2 API.

Like DOM, the `Document` class parses an XML document into a nested tree of objects. You can navigate the tree with Ruby accessors (Recipe 11.2)or with XPath queries (Recipe 11.4). You can modify the tree by creating your own `Element` and `Text` objects (Recipe 11.9). If even `Document` is too heavyweight for you, you can use the `XmlSimple` library to transform an XML file into a nested Ruby hash (Recipe 11.6).

With a DOM-style API like `Document`, you have to parse the entire XML file before you can do anything. The XML document becomes a large number of Ruby objects nested under a `Document` object, all sitting around taking up memory. With a SAXstyle parser like the `StreamParser` class, you can process a document as it's parsed, creating only the objects you want. The `StreamParser` API is covered in Recipe 11.3.

The main problem with the REXML APIs is that they're very picky. They'll only parse a document that's valid XML, or close enough to be have an unambiguous representation. This makes them nearly useless for parsing HTML documents off the World Wide Web,

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

since the average web page is not valid XML. Recipe 11.5 shows how to use the third-party tools Rubyful Soup and SGMLParser; they give a DOMor SAX-style interface that handles even invalid XML.

- http://www.germane-software.com/software/rexml/
- http://www.germane-software.com/software/rexml/docs/tutorial.html

Recipe 11.1. Checking XML Well-Formedness

Credit: Rod Gaither

Problem

You want to check that an XML document is well-formed before processing it.

Solution

The best way to see whether a document is well-formed is to try to parse it. The REXML library raises an exception when it can't parse an XML document, so just try parsing it and `rescue` any exception.

The `valid_xml?` method below returns `nil` unless it's given a valid XML document. If the document is valid, it returns a parsed `Document` object, so you don't have to parse it again:

```
require 'rexml/document'
def valid_xml?(xml)
 begin
   REXML::Document.new(xml)
 rescue REXML::ParseException
   # Return nil if an exception is thrown
 end
end
```

Discussion

To be useful, an XML document must be structured correctly or "well-formed." For instance, an opening tag must either be self-closing or be paired with an appropriate closing tag.

As a file and messaging format, XML is often used in situations where you don't have control over the input, so you can't assume that it will always be well-formed. Rather than just letting REXML throw an exception, you'll need to handle ill-formed XML gracefully, providing options to retry or continue on a different path.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

This bit of XML is not well-formed: it's missing ending tags for both the `pending` and `done` elements:

```
bad_xml = %{
<tasks>
 <pending>
   <entry>Grocery Shopping</entry>
 <done>
   <entry>Dry Cleaning</entry>
</tasks>}

valid_xml?(bad_xml)                          # => nil
```

This bit of XML is well-formed, so `valid_xml?` returns the parsed `Document` object.

```
good_xml = %{
<groceries>
 <bread>Wheat</bread>
 <bread>Quadrotriticale</bread>
</groceries>}

doc = valid_xml?(good_xml)
doc.root.elements[1]                         # => <bread> … </>
```

When your program is responsible for writing XML documents, you'll want to write unit tests that make sure you generate valid XML. You can use a feature of the `Test:: Unit` library to simplify the checking. Since invalid XML makes REXML throw an exception, your unit test can use the `assert_nothing_thrown` method to make sure your XML is valid:

```
doc = nil
assert_nothing_thrown {doc = REXML::Document.new(source_xml)}
```

This is a simple, clean test to verify XML when using a unit test.

Note that `valid_xml?` doesn't work perfectly: some invalid XML is unambiguous, which means REXML can parse it. Consider this truncated version of the valid XML example. It's missing its closing tags, but there's no ambiguity about which closing tag should come first, so REXML can parse the file and provide the closing tags:

```
invalid_xml = %{
<groceries>
 <bread>Wheat
}

(valid_xml? invalid_xml) == nil          # => false # That is, it is "valid"
REXML::Document.new(invalid_xml).write
# <groceries>
#   <bread>Wheat
# </bread></groceries>
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Official information on XML can be found at http://www.w3.org/XML/
- The Wikipedia has a good description of the difference between Well-Formed and Valid XML documents at http://en.wikipedia.org/wiki/Xml#Correctness_in_an_XML_document
- Recipe 11.5, "Parsing Invalid Markup"
- Recipe 17.3, "Handling an Exception"

Recipe 11.2. Extracting Data from a Document's Tree Structure

Credit: Rod Gaither

Problem

You want to parse an XML file into a Ruby data structure, to traverse it or extract data from it.

Solution

Pass an XML document into the `REXML::Document` constructor to load and parse the XML. A `Document` object contains a tree of subobjects (of class `Element` and `Text`) representing the tree structure of the underlying document. The methods of `Document` and `Element` give you access to the XML tree data. The most useful of these methods is `#each_element`.

Here's some sample XML and the load process. The document describes a set of orders, each of which contains a set of items. This particular document contains a single order for two items.

```
orders_xml = %{
<orders>
  <order>
    <number>105</number>
    <date>02/10/2006</date>
    <customer>Corner Store</customer>
    <items>
      <item upc="404100" desc="Red Roses" qty="240" />
      <item upc="412002" desc="Candy Hearts" qty="160" />
    </items>
  </order>
</orders>}

require 'rexml/document'
orders = REXML::Document.new(orders_xml)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

To process each order in this document, we can use `Document#root` to get the document's root element (`<orders>`)and then call `Element#each_element` to iterate over the children of the root element (the `<order>` elements). This code repeatedly calls `each` to move down the document tree and print the details of each order in the document:

```
orders.root.each_element do |order|    # each <order> in <orders>
  order.each_element do |node|         # <customer>, <items>, etc. in <order>
    if node.has_elements?
      node.each_element do |child|     # each <item> in <items>
        puts "#{child.name}: #{child.attributes['desc']}"
      end
    else
      # the contents of <number>, <date>, etc.
      puts "#{node.name}: #{node.text}"
    end
  end
end
# number: 105
# date: 02/10/2006
# customer: Corner Store
# item: Red Roses
# item: Candy Hearts
```

Discussion

Parsing an XML file into a `Document` gives you a tree-like data structure that you can treat kind of like an array of arrays. Starting at the document root, you can move down the tree until you find the data that interests you. In the example above, note how the structure of the Ruby code mirrors the structure of the original document. Every call to `each_element` moves the focus of the code down a level: from `<orders>` to `<order>` to `<items>` to `<item>`.

There are many other methods of `Element` you can use to navigate the tree structure of an XML document. Not only can you iterate over the child elements, you can reference a specific child by indexing the parent as though it were an array. You can navigate through siblings with `Element.next_element` and `Element.previous_element`. You can move up the document tree with `Element.parent`:

```
my_order = orders.root.elements[1]
first_node = my_order.elements[1]
first_node.name                        # => "number"
first_node.next_element.name           # => "date"
first_node.parent.name                 # => "order"
```

This only scratches the surface; there are many other ways to interact with the data loaded from an XML source. For example, explore the convenience methods `Element.each_element_with_attribute` and `Element.each_element_with_text`, which let you select elements based on features of the elements themselves.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- The RDoc documentation for the `REXML::Document` and `REXML::Element` classes
- The section "Tree Parsing XML and Accessing Elements" in the REXML Tutorial (http://www.germane-software.com/software/rexml/docs/tutorial.html#id2247335)
- If you want to start navigating the document at some point other than the root, an XPath statement is probably the simplest way to get where you want; see Recipe 11.4, "Navigating a Document with XPath"

Recipe 11.3. Extracting Data While Parsing a Document

Credit: Rod Gaither

Problem

You want to process a large XML file without loading it all into memory.

Solution

The method `REXML::Document.parse_stream` gives you a fast and flexible way to scan a large XML file and process the parts that interest you.

Consider this XML document, the output of a hypothetical program that runs auto mated tasks. We want to parse the document and find the tasks that failed (that is, returned an error code other than zero).

```
event_xml = %{
<events>
  <clean system="dev" start="01:35" end="01:55" area="build" error="1" />
  <backup system="prod" start="02:00" end="02:35" size="2300134" error="0" />
  <backup system="dev" start="02:00" end="02:01" size="0" error="2" />
  <backup system="test" start="02:00" end="02:47" size="327450" error="0" />
</events>}
```

We can process the document as it's being parsed by writing a `REXML::StreamListener` subclass that responds to parsing events such as `tag_start` and `tag_end`. Here's a subclass that listens for tags with a nonzero value for their `error` attribute. It prints a message for every failed event it finds.

```
require 'rexml/document'
require 'rexml/streamlistener'

class ErrorListener
  include REXML::StreamListener
  def tag_start(name, attrs)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    if attrs["error"] != nil and attrs["error"] != "0"
      puts %{Event "#{name}" failed for system "#{attrs["system"]}" } +
    %{with code #{attrs["error"]}}
    end
  end
end
```

To actually parse the XML data, pass it along with the `StreamListener` into the method `REXML::Document.parse_stream`:

```
REXML::Document.parse_stream(event_xml, ErrorListener.new)
# Event "clean" failed for system "dev" with code 1
# Event "backup" failed for system "dev" with code 2
```

Discussion

We could find the failed events in less code by loading the XML into a `Document` and running an XPath query. That approach would work fine for this example, since the document only contains four events. It wouldn't work as well if the document were a file on disk containing a billion events. Building a `Document` means building an elaborate in-memory data structure representing the entire XML document. If you only care about part of a document (in this case, the failed events), it's faster and less memory-intensive to process the document as it's being parsed. Once the parser reaches the end of the document, you're done.

The stream-oriented approach to parsing XML can be as simple as shown in this recipe, but it can also handle much more complex scenarios. Your `StreamListener` subclass can keep arbitrary state in instance variables, letting you track complex combinations of elements and attributes.

See Also

- The RDoc documentation for the `REXML::StreamParser` class
- The "Stream Parsing" section of the REXML Tutorial (http://www.germane-software.com/software/rexml/docs/tutorial.html#id2248457)
- Recipe 11.2, "Extracting Data from a Document's Tree Structure"

Recipe 11.4. Navigating a Document with XPath

Problem

You want to find or address sections of an XML document in a standard, programming-language–independent way.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

The XPath language defines a way of referring to almost any element or set of elements in an XML document, and the REXML library comes with a complete XPath implementation. `REXML::XPath` provides three class methods for locating `Element` objects within parsed documents: `first`, `each`, and `match`.

Take as an example the following XML description of an aquarium. The aquarium contains some fish and a gaudy castle decoration full of algae. Due to an aquarium stocking mishap, some of the smaller fish have been eaten by larger fish, just like in those cartoon food chain diagrams. (Figure 11-1 shows the aquarium.)

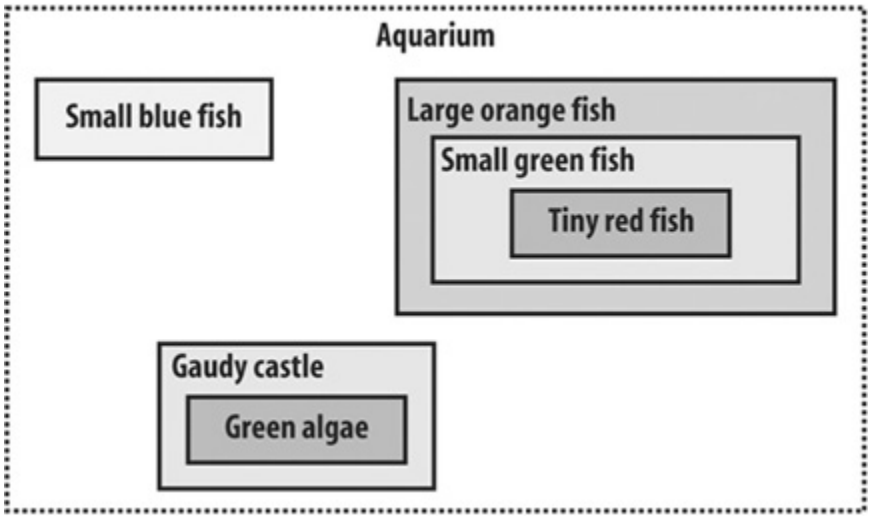
```
xml = %{
<aquarium>
 <fish color="blue" size="small" />

 <fish color="orange" size="large">
   <fish color="green" size="small">
    <fish color="red" size="tiny" />
   </fish>
 </fish>

 <decoration type="castle" style="gaudy">
   <algae color="green" />
  </decoration>
 </aquarium>}

 require 'rexml/document'
 doc = REXML::Document.new xml
```

Figure 11-1. The aquarium



We can use `REXML::Xpath.first` to get the `Element` object corresponding to the first `<fish>` tag in the document:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
REXML::XPath.first(doc, '//fish')
# => <fish size='small' color='blue'/>
```

We can use `match` to get an array containing all the elements that are green:

```
REXML::XPath.match(doc, '//[@color="green"]')
# => [<fish size='small' color='green'> … </>, <algae color='green'/>]
```

We can use `each` with a code block to iterate over all the fish that are inside other fish:

```
def describe(fish)
  "#{fish.attribute('size')} #{fish.attribute('color')} fish"
end
REXML::XPath.each(doc, '//fish/fish') do |fish|
  puts "The #{describe(fish.parent)} has eaten the #{describe(fish)}."
end
# The large orange fish has eaten the small green fish.
# The small green fish has eaten the tiny red fish.
```

Discussion

Every element in a `Document` has an `xpath` method that returns the canonical XPath path to that element. This path can be considered the element's "address" within the document. In this example, a complex bit of Ruby code is replaced by a simple XPath expression:

```
red_fish = doc.children[0].children[3].children[1].children[1]
# => <fish size='tiny' color='red'/>

red_fish.xpath
# => "/aquarium/fish[2]/fish/fish"

REXML::XPath.first(doc, red_fish.xpath)
# => <fish size='tiny' color='red'/>
```

Even a brief overview of XPath is beyond the scope of this recipe, but here are some more examples to give you ideas:

```
# Find the second green element.
REXML::XPath.match(doc, '//[@color="green"]')[1]
# => <algae color='green'/>

# Find the color attributes of all small fish.
REXML::XPath.match(doc, '//fish[@size="small"]/@color')
# => [color='blue', color='green']

# Count how many fish are inside the first large fish.
REXML::XPath.first(doc, "count(//fish[@size='large'][1]//*fish)")
# => 2
```

The `Elements` class acts kind of like an array that supports XPath addressing. You can make your code more concise by passing an XPath expression to `Elements#each`, or using it as an array index.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
doc.elements.each('//fish') { |f| puts f.attribute('color') }
# blue
# orange
# green
# red

doc.elements['//fish']
# => <fish size='small' color='blue'/>
```

Within an XPath expression, the first element in a list has an index of 1, not 0. The XPath expression `//fish[size='large'][1]` matches the first large fish, not the second large fish, the way `large_fish[1]` would in Ruby code. Pass a number as an array index to an `Elements` object, and you get the same behavior as XPath:

```
doc.elements[1]
# => <aquarium> … </>
doc.children[0]
# => <aquarium> … </>
```

See Also

- The XPath standard, at http://www.w3.org/TR/xpath, has more XPath examples
- *XPath and XPointer* by John E. Simpson (O'Reilly)

Recipe 11.5. Parsing Invalid Markup

Problem

You need to extract data from a document that's supposed to be HTML or XML, but that contains some invalid markup.

Solution

For a quick solution, use Rubyful Soup, written by Leonard Richardson and found in the `rubyful_soup` gem. It can build a document model even out of invalid XML or HTML, and it offers an idiomatic Ruby interface for searching the document model. It's good for quick screen-scraping tasks or HTML cleanup.

```
require 'rubygems'
require 'rubyful_soup'

invalid_html = 'A lot of <b class=1>tags are <i class=2>never closed.'
soup = BeautifulSoup.new(invalid_html)
puts soup.prettify
# A lot of
#  <b class="1">tags are
#   <i class="2">never closed.
#   </i>
#  </b>
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
soup.b.i                                    # => <i class="2">never closed.</i>
soup.i                                      # => <i class="2">never closed.</i>
soup.find(nil, :attrs=>{'class' => '2'}) # => <i class="2">never closed.</i>
soup.find_all('i')                          # => [<i class="2">never closed.</i>]

soup.b['class']                             # => "1"

soup.find_text(/closed/)                    # => "never closed."
```

If you need better performance, do what Rubyful Soup does and write a custom parser on top of the event-based parser `SGMLParser` (found in the `htmltools` gem). It works a lot like REXML's `StreamListener` interface.

Discussion

Sometimes it seems like the authors of markup parsers do their coding atop an ivory tower. Most parsers simply refuse to parse bad markup, but this cuts off an enormous source of interesting data. Most of the pages on the World Wide Web are invalid HTML, so if your application uses other peoples' web pages as input, you need a forgiving parser. Invalid XML is less common but by no means rare.

The `SGMLParser` class in the `htmltools` gem uses regular expressions to parse an XMLlike data stream. When it finds an opening or closing tag, some data, or some other part of an XML-like document, it calls a hook method that you're supposed to define in a subclass. `SGMLParser` doesn't build a document model or keep track of the document state: it just generates events. If closing tags don't match up or if the markup has other problems, it won't even notice.

Rubyful Soup's parser classes define `SGMLParser` hook methods that build a document model out of an ambiguous document. Its `BeautifulSoup` class is intended for HTML documents: it uses heuristics like a web browser's to figure out what an ambiguous document "really" means. These heuristics are specific to HTML; to parse XML documents, you should use the `BeautifulStoneSoup` class. You can also subclass `BeautifulStoneSoup` and implement your own heuristics.

Rubyful Soup builds a densely linked model of the entire document, which uses a lot of memory. If you only need to process certain parts of the document, you can implement the `SGMLParser` hooks yourself and get a faster parser that uses less memory.

Here's a `SGMLParser` subclass that extracts URLs from a web page. It checks every A tag for an `href` attribute, and keeps the results in a set. Note the similarity to the `LinkGrabber` class defined in Recipe 11.13.

```
require 'rubygems'
require 'html/sgml-parser'
require 'set'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
html = %{<a name="anchor"><a href="http://www.oreilly.com">O'Reilly</a>
         <b>irrelevant</b><a href="http://www.ruby-lang.org/">Ruby</a>}

class LinkGrabber < HTML::SGMLParser
  attr_reader :urls

  def initialize
    @urls = Set.new
    super
  end

  def do_a(attrs)
    url = attrs.find { |attr| attr[0] == 'href' }
    @urls << url[1] if url
  end
end

extractor = LinkGrabber.new
extractor.feed(html)
extractor.urls
# => #<Set: {"http://www.ruby-lang.org/", "http://www.oreilly.com"}>
```

The equivalent Rubyful Soup program is quicker to write and easier to understand, but it runs more slowly and uses more memory:

```
require 'rubyful_soup'

urls = Set.new
BeautifulStoneSoup.new(html).find_all('a').each do |tag|
  urls << tag['href'] if tag['href']
end
```

You can improve performance by telling Rubyful Soup's parser to ignore everything except `A` tags and their contents:

```
puts BeautifulStoneSoup.new(html, :parse_only_these => 'a')
# <a name="anchor"></a>
# <a href="http://www.oreilly.com">O'Reilly</a>
# <a href="http://www.ruby-lang.org/">Ruby</a>
```

But the fastest implementation will always be a custom `SGMLParser` subclass. If your parser is part of a full application (rather than a one-off script), you'll need to find the best tradeoff between performance and code legibility.

See Also

- Recipe 11.13, "Extracting All the URLs from an HTML Document"
- The Rubyful Soup documentation (http://www.crummy.com/software/RubyfulSoup/documentation.html)
- The `htree` library defines a forgiving HTML/XML parser that can convert a parsed document into a REXML `Document` object (http://cvs.m17n.org/~akr/htree/)
- The HTML TIDY library can fix up most invalid HTML so that it can be parsed by a standard parser; it's a C library with Ruby bindings; see http://tidy.sourceforge.net/ for the library, and http://rubyforge.org/projects/tidy for the bindings

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 11.6. Converting an XML Document into a Hash

Problem

When you parse an XML document with `Document.new`, you get a representation of the document as a complex data structure. You'd like to represent an XML document using simple, built-in Ruby data structures.

Solution

Use the `XmlSimple` library, found in the `xml-simple` gem. It parses an XML document into a hash.

Consider an XML document like this one:

```
xml = %{
<freezer temp="-12" scale="celcius">
 <food>Phyllo dough</food>
 <food>Ice cream</food>
 <icecubetray>
  <cube1 />
  <cube2 />
 </icecubetray>
</freezer>}
```

Here's how you parse it with XMLSimple:

```
require 'rubygems'
require 'xmlsimple'

doc = XmlSimple.xml_in xml
```

And here's what it looks like:

```
require 'pp'
pp doc
# {"icecubetray"=>[{"cube2"=>[{}], "cube1"=>[{}]}],
#  "food"=>["Phyllo dough", "Ice cream"],
#  "scale"=>"celcius",
#  "temp"=>"-12"}
```

Discussion

`XmlSimple` is a lightweight alternative to the `Document` class. Instead of exposing a tree of `Element` objects, it exposes a nested structure of Ruby hashes and arrays. There's no performance savings (`XmlSimple` actually builds a `Document` class behind the scenes and iterates over it, so it's about half as fast as `Document`), but the resulting object is easy

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

to use. `XmlSimple` also provides several tricks that can make a document more concise and navigable.

The most useful trick is the `KeyAttr` one. Suppose you had a better-organized freezer than the one above, a freezer in which everything had its own `name` attribute:[2]

[2] Okay, it's not really better organized. In fact, it's exactly the same. But it sure looks cooler!

```
xml = %{
<freezer temp="-12" scale="celcius">
 <item name="Phyllo dough" type="food" />
 <item name="Ice cream" type="food" />
 <item name="Ice cube tray" type="container">
  <item name="Ice cube" type="food" />
  <item name="Ice cube" type="food" />
 </item>
</freezer>}
```

You could parse this data with just a call to `XmlSimple.xml_in`, but you get a more concise representation by specifing the `name` attribute as a `KeyAttr` argument. Compare:

```
parsed1 = XmlSimple.xml_in xml
pp parsed1
# {"scale"=>"celcius",
#  "item"=>
#   [{"name"=>"Phyllo dough", "type"=>"food"},
#    {"name"=>"Ice cream", "type"=>"food"},
# {"name"=>"Ice cube tray",
#    "type"=>"container",
#    "item"=>
#    [{"name"=>"Ice cube", "type"=>"food"},
#     {"name"=>"Ice cube", "type"=>"food"}]}],
# "temp"=>"-12"}

parsed2 = XmlSimple.xml_in(xml, 'KeyAttr' => 'name')
pp parsed2
# {"scale"=>"celcius",
#  "item"=>
#   {"Phyllo dough"=>{"type"=>"food"},
#    "Ice cube tray"=>
#     {"type"=>"container",
#      "item"=>{"Ice cube"=>{"type"=>"food"}}},
#    "Ice cream"=>{"type"=>"food"}},
# "temp"=>"-12"}
```

The second parsing is also easier to navigate:

```
parsed1["item"].detect { |i| i['name'] == 'Phyllo dough' }['type']
# => "food"
parsed2["item"]["Phyllo dough"]["type"]
# => "food"
```

But notice that the second parsing represents the ice cube tray as containing only one ice cube. This is because both ice cubes have the same `name`. When two tags at the same level have the same `KeyAttr`, one overwrites the other in the hash.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

You can modify the data structure with normal Ruby hash and array methods, then write it back out to XML with `XMLSimple.xml_out`:

```
parsed1["item"] << {"name"=>"Curry leaves", "type"=>"spice"}
parsed1["item"].delete_if { |i| i["name"] == "Ice cube tray" }

puts XmlSimple.xml_out(parsed1, "RootName"=>"freezer")
# <freezer scale="celcius" temp="-12">
#   <item name="Phyllo dough" type="food" />
#   <item name="Ice cream" type="food" />
#   <item name="Curry leaves" type="spice" />
# </freezer>
```

Be sure to specify a `RootName` argument when you call `xml_out`. When it parses a file, `XmlSimple` removes one level of indirection by throwing away the name of your document's root element. You can prevent this by using the `KeepRoot` argument in your original call to `xml_in`. You'll need an extra hash lookup to navigate the resulting data structure, but you'll retain the name of your root element.

```
parsed3 = XmlSimple.xml_in(xml, 'KeepRoot'=>true)
# Now there's no need to add an extra root element when writing back to XML.
XmlSimple.xml_out(parsed3, 'RootName'=>nil)
```

One disadvantage of `XmlSimple` is that, since it puts elements into a hash, it replaces the order of the original document with the random-looking order of a Ruby hash. This is fine for a document listing the contents of a freezer—where order doesn't matter—but it would give interesting results if you tried to use it on a web page.

Another disadvantage is that, since an element's attributes and children are put into the same hash, you have no reliable way of telling one from the other. Indeed, attributes and subelements may even end up in a list together, as in this example:

```
pp XmlSimple.xml_in(%{
<freezer temp="-12" scale="celcius">
 <temp>Body of temporary worker who knew too much</temp>
</freezer>})
# {"scale"=>"celcius",
#  "temp"=>["-12", "Body of temp worker who knew too much"]}
```

See Also

- The `XmlSimple` home page at http://www.maik-schmidt.de/xml-simple.html has much more information about the options you can pass to `XmlSimple.xml_in`

Recipe 11.7. Validating an XML Document

Credit: Mauro Cicio

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to check whether an XML document conforms to a certain schema or DTD.

Solution

Unfortunately, as of this writing there are no stable, pure Ruby libraries that do XML validation. You'll need to install a Ruby binding to a C library. The easiest one to use is the Ruby binding to the GNOME `libxml2` toolkit. (There are actually two Ruby bindings to `libxml2`, so don't get confused: we're referring to the one you get when you install the `libxml-ruby` gem.)

To validate a document against a DTD, create a a `DTD` object and pass it into `Document#validate`. To validate against an XML Schema, pass in a `Schema` object instead.

Consider the following DTD, for a cookbook like this one:

```
require 'rubygems'
require 'libxml'

dtd = XML::Dtd.new(%{<!ELEMENT rubycookbook (recipe+)>
<!ELEMENT recipe (title?, problem, solution, discussion, seealso?)+>
<!ELEMENT title (#PCDATA)>
<!ELEMENT problem (#PCDATA)>
<!ELEMENT solution (#PCDATA)>
<!ELEMENT discussion (#PCDATA)>
<!ELEMENT seealso (#PCDATA)>})
```

Here's an XML document that looks like it conforms to the DTD:

```
open('cookbook.xml', 'w') do |f|
  f.write %{<?xml version="1.0"?>
<rubycookbook>
 <recipe>
   <title>A recipe</title>
   <problem>A difficult/common problem</problem>
   <solution>A smart solution</solution>
   <discussion>A deep solution</discussion>
   <seealso>Pointers</seealso>
 </recipe>
</rubycookbook>
}
end
```

But does it really? We can tell for sure with `Document#validate`:

```
document = XML::Document.file('cookbook.xml')
document.validate(dtd)                          # => true
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's a Schema definition for the same document. We can validate the document against the schema by making it into a `Schema` object and passing *that* into `Document#validate`:

```
schema = XML::Schema.from_string %{<?xml version="1.0"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="recipe" type="recipeType"/>

  <xsd:element name="rubycookbook" type="rubycookbookType"/>

  <xsd:element name="title"      type="xsd:string"/>
  <xsd:element name="problem"    type="xsd:string"/>
  <xsd:element name="solution"   type="xsd:string"/>
  <xsd:element name="discussion" type="xsd:string"/>
  <xsd:element name="seealso"    type="xsd:string"/>

  <xsd:complexType name="rubycookbookType">
   <xsd:sequence>
      <xsd:element ref="recipe"/>
    </xsd:sequence>
 </xsd:complexType>
  <xsd:complexType name="recipeType">
    <xsd:sequence>
      <xsd:element ref="title"/>
      <xsd:element ref="problem"/>
      <xsd:element ref="solution"/>
      <xsd:element ref="discussion"/>
      <xsd:element ref="seealso"/>
  </xsd:sequence>
</xsd:complexType>

</xsd:schema>
}

document.validate(schema)                               # => true
```

Discussion

Programs that use XML validation are more robust and less complicated than nonvalidating versions. Before starting work on a document, you can check whether or not it's in the format you expect. Most services that accept XML as input don't have forgiving parsers, so you *must* validate your document before submitting it or it might fail without you even noticing.

One of the most popular and complete XML libraries around is the GNOME Libxml2 library. Despite its name, it works fine outside the GNOME platform, and has been ported to many different OSes. The Ruby project `libxml` (http://libxml.rubyforge.org) is a Ruby wrapper around the GNOME Libxml2 library. The project is not yet in a mature state, but it's very active and the validation features are definitively usable. Not only does `libxml` support validation and a complete range of XML manipolation techniques, it can also improve your program's speed by an order of magnitude, since it's written in C instead of REXML's pure Ruby.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Don't confuse the `libxml` *project* with the `libxml` *library*. The latter is part of the `XML::Tools` project. It binds against the GNOME Libxml2 library, but it doesn't expose that library's validation features. If you try the example code above but can't find the `XML::Dtd` or the `XML::Schema` classes, then you've got the wrong binding. If you installed the `libxml-ruby` package on Debian GNU/Linux, you've got the wrong one. You need the one you get by installing the `libxml-ruby` gem. Of course, you'll need to have the actual GNOME `libxml` library installed as well.

See Also

- The Ruby `libxml` project page (http://www.rubyforge.org/projects/libxml)
- The *other* Ruby `libxml` binding (the one that doesn't do validation)is part of the XML::Tools project (http://rubyforge.org/projects/xml-tools/); don't confuse the two!
- The GNOME `libxml` project homepage (http://xmlsoft.org/)
- Refer to http://www.w3.org/XML for the difference between a DTD and a Schema

Recipe 11.8. Substituting XML Entities

Problem

You've parsed a document that contains internal XML entities. You want to substitute the entities in the document for their values.

Solution

To perform entity substitution on a specific text element, call its `value` method. If it's the first text element of its parent, you can call `text` on the parent instead.

Here's a simple document that defines and uses two entities in a single text node. We can substitute those entities for their values without changing the document itself:

```
require 'rexml/document'

str = %{<?xml version="1.0"?>
<!DOCTYPE doc [
  <!ENTITY product 'Stargaze'>
  <!ENTITY version '2.3'>
]>
<doc>
 &product; v&version; is the most advanced astronomy product on the market.
</doc>}
doc = REXML::Document.new str

doc.root.children[0].value
# => "\n Stargaze v2.3 is the most advanced astronomy product on the market.\n"
doc.root.text
# => "\n Stargaze v2.3 is the most advanced astronomy product on the market.\n"
```

```
doc.root.children[0].to_s
# => "\n &product; v&version; is the most advanced astronomy product on the market.\n"
doc.root.write
# <doc>
# &product; v&version; is the most advanced astronomy program on the market.
# </doc>
```

Discussion

Internal XML entities are often used to factor out data that changes a lot, like dates or version numbers. But REXML only provides a convenient way to perform substitution on a single text node. What if you want to perform substitutions throughout the entire document?

When you call `Document#write` to send a document to some `IO` object, it ends up calling `Text#to_s` on each text node. As seen in the Solution, this method presents a "normalized" view of the data, one where entities are displayed instead of having their values substituted in.

We could write our own version of `Document#write` that presents an "unnormalized" view of the document, one with entity values substituted in, but that would be a lot of work. We could hack `Text#to_s` to work more like `Text#value`, or hack `Text#write` to call the `value` method instead of `to_s`. But it's less intrusive to do the entity replacement outside of the `write` method altogether. Here's a class that wraps any `IO` object and performs entity replacement on all the text that comes through it:

```
require 'delegate'
require 'rexml/text'
class EntitySubstituter < DelegateClass(IO)
  def initialize(io, document, filter=nil)
    @document = document
    @filter = filter
    super(io)

  end

  def <<(s)
    super(REXML::Text::unnormalize(s, @document.doctype, @filter))
  end
end

output = EntitySubstituter.new($stdout, doc)
doc.write(output)
# <?xml version='1.0'?><!DOCTYPE doc [
# <!ENTITY product "Stargaze">
# <!ENTITY version "2.3">
# ]>
# <doc>
#  Stargaze v2.3 is the most advanced astronomy product on the market.
# </doc>
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Because it processes the entire output of `Document#write`, this code will replace *all* entity references in the document. This includes any references found in attribute values, which may or may not be what you want.

If you create a `Text` object manually, or set the value of an existing object, REXML assumes that you're giving it unnormalized text, and normalizes it. This can be problematic if your text contains strings that happen to be the values of entities:

```
text_node = doc.root.children[0]
text_node.value = "&product; v&version; has a catalogue of 2.3 " +
                  "million celestial objects."

doc.write
# <?xml version='1.0'?><!DOCTYPE doc [
# <!ENTITY product "Stargaze">
# <!ENTITY version "2.3">
# ]>
# <doc>&product; v&version; has a catalogue of &version; million celestial objects.
  </doc>
```

To avoid this, you can create a "raw" text node:

```
text_node.raw = true
doc.write
# <?xml version='1.0'?><!DOCTYPE doc [
# <!ENTITY product "Stargaze">
# <!ENTITY version "2.3">
# ]>
# <doc>&product; v&version; has a catalogue of 2.3 million celestial objects.</doc>

text_node.value
# => "Stargaze v2.3 has a catalogue of 2.3 million celestial objects."
text_node.to_s
# => "&product; v&version; has a catalogue of 2.3 million celestial objects."
```

In addition to entities you define, REXML automatically processes five named character entities: the ones for left and right angle brackets, single and double quotes, and the ampersand. Each is replaced with the corresponding ASCII character.

```
str = %{
  <!DOCTYPE doc [ <!ENTITY year '2006'> ]>
  <doc>&#169; &year; Komodo Dragon &amp; Bob Productions</doc>
}

doc = REXML::Document.new str
text_node = doc.root.children[0]

text_node.value
# => "&copy; 2006 Komodo Dragon & Bob Productions"
text_node.to_s
# => "&copy; &year; Komodo Dragon &amp; Bob Productions"
```

"©" is an HTML character entity representing the copyright symbol, but REXML doesn't know that. It only knows about the five XML character entities. Also, REXML only

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

knows about internal entities: ones whose values are defined within the same document that uses them. It won't resolve external entities.

See Also

- The section "Text Nodes" of the REXML tutorial (http://www.germane-software.com/software/rexml/docs/tutorial.html#id2248004)

Recipe 11.9. Creating and Modifying XML Documents

Problem

You want to modify an XML document, or create a new one from scratch.

Solution

To create an XML document from scratch, just start with an empty `Document` object.

```
require 'rexml/document'
require
doc = REXML::Document.new
```

To add a new element to an existing document, pass its name and any attributes into its parent's `add_element` method. You don't have to create the `Element` objects yourself.

```
meeting = doc.add_element 'meeting'
meeting_start = Time.local(2006, 10, 31, 13)
meeting.add_element('time', { 'from' => meeting_start,
                              'to' => meeting_start + 3600 })

doc.children[0]                                 # => <meeting> … </>
doc.children[0].children[0]
# => "<time from='Tue Oct 31 13:00:00 EST 2006'
#      to='Tue Oct 31 14:00:00 EST 2006'/>"

doc.write($stdout, 1)
# <meeting>
#  <time from='Tue Oct 31 13:00:00 EST 2006'
#        to='Tue Oct 31 14:00:00 EST 2006'/>
# </meeting>
doc.children[0]                                 # => <?xml … ?>
doc.children[1]                                 # => <meeting> … </>
```

To append a text node to the contents of an element, use the `add_text` method. This code adds an `<agenda>` element to the `<meeting>` element, and gives it two different text nodes:

```
agenda = meeting.add_element 'agenda'
doc.children[1].children[1]               # => <agenda/>

agenda.add_text "Nothing of importance will be decided."
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
agenda.add_text " The same tired ideas will be rehashed yet again."

doc.children[1].children[1]               # => <agenda> … </>

doc.write($stdout, 1)
# <meeting>
#  <time from='Tue Oct 31 13:00:00 EST 2006'
#        to='Tue Oct 31 14:00:00 EST 2006'/>
#  <agenda>
#   Nothing of importance will be decided. The same tired ideas will be
#   rehashed yet again.
#  </agenda>
# </meeting>
```

`Element#text=` is a nice shortcut for giving an element a single text node. You can also use to overwrite a document's initial text nodes:

```
item1 = agenda.add_element 'item'
doc.children[1].children[1].children[1]       # => <item/>
item1.text = 'Weekly status meetings: improving attendance'
doc.children[1].children[1].children[1]       # => <item> … </>
doc.write($stdout, 1)
# <meeting>
#  <time from='Tue Oct 31 13:00:00 EST 2006'
#       to='Tue Oct 31 14:00:00 EST 2006'/>
#  <agenda>
#   Nothing of importance will be decided. The same tired ideas will be
#   rehashed yet again.
#   <item>Weekly status meetings: improving attendance</item>
#  </agenda>
# </meeting>
```

Discussion

If you can access an element or text node (numerically or with XPath), you can modify or delete it. You can modify an element's name with `name=`, and modify one of its attributes by assigning to an index of `attributes`. This code uses these methods to make major changes to a document:

```
doc = REXML::Document.new %{<?xml version='1.0'?>
<girl size="little">
 <foods>
  <sugar />
  <spice />
 </foods>
 <set of="nice things" cardinality="all" />
</girl>
}

root = doc[1]                # => <girl size='little'> … </>
root.name = 'boy'

root.elements['//sugar'].name = 'snails'
root.delete_element('//spice')

set = root.elements['//set']
set.attributes["of"] = "snips"
set.attributes["cardinality"] = 'some'

root.add_element('set', {'of' => 'puppy dog tails', 'cardinality' => 'some' })
doc.write
# <?xml version='1.0'?>
# <boy size='little'>
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
#  <foods>
#   <snails/>
#
#  </foods>
#  <set of='snips' cardinality='some'/>
# <set of='puppy dog tails' cardinality='some'/></boy>
```

You can delete an attribute with `Element#delete_attribute`, or by assigning `nil` to `it`:

```
root.attributes['size'] = nil
doc.write($stdout, 0)
# <?xml version='1.0'?>
# <boy>
#   <foods>
#   …
# </boy>
```

You can use methods like `replace_with` to swap out one node for another:

```
doc.elements["//snails"].replace_with(REXML::Element.new("escargot"))
```

All these methods are convenient, but `add_element` in particular is not very idiomatic. The `cgi` library lets you structure method calls and code blocks so that your Ruby code has the same nesting structure as the HTML it generates. Why shouldn't you be able to do the same for XML? Here's a new method for `Element` that makes it possible:

```
class REXML::Element
  def with_element(*args)
    e = add_element(*args)
    yield e if block_given?
  end
end
```

Now you can structure your Ruby code the same way you structure your XML:

```
doc = REXML::Document.new
doc.with_element('girl', {'size' => 'little'}) do |girl|
  girl.with_element('foods') do |foods|
    foods.add_element('sugar')
    foods.add_element('spice')
  end
  girl.add_element('set', {'of' => 'nice things', 'cardinality' => 'all'})
end

doc.write($stdout, 0)
# <girl size='little'>
#   <foods>
#     <sugar/>
#     <spice/>
#   </foods>
#   <set of='nice things' cardinality='all'/>
# </girl>
```

The `builder` gem also lets you build XML this way.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 7.10, "Hiding Setup and Cleanup in a Block Method," has an example of using the `XmlMarkup` class in the `builder` gem.

Recipe 11.10. Compressing Whitespace in an XML Document

Problem

When REXML parses a document, it respects the original whitespace of the document's text nodes. You want to make the document smaller by compressing extra whitespace.

Solution

Parse the document by creating a `REXML::Document` out of it. Within the `Document` constructor, tell the parser to compress all runs of whitespace characters:

```
require 'rexml/document'

text = %{<doc><a>Some whitespace</a> <b>Some more</b></doc>}

REXML::Document.new(text, { :compress_whitespace => :all }).to_s
# => "<doc><a>Some whitespace</a> <b>Some more</b></doc>"
```

Discussion

Sometimes whitespace within a document is significant, but usually (as with HTML) it can be compressed without changing the meaning of the document. The resulting document takes up less space on the disk and requires less bandwidth to transmit.

Whitespace compression doesn't have to be all-or-nothing. REXML gives two ways to configure it. Instead of passing `:all` as a value for `:compress_whitespace`, you can pass in a list of tag names. Whitespace will only be compressed in those tags:

```
REXML::Document.new(text, { :compress_whitespace => %w{a} }).to_s
# => "<doc><a>Some whitespace</a> <b>Some more</b></doc>"
```

You can also switch it around: pass in `:respect_whitespace` and a list of tag names whose whitespace you *don't* want to be compressed. This is useful if you know that whitespace is significant within certain parts of your document.

```
REXML::Document.new(text, { :respect_whitespace => %w{a} }).to_s
# => "<doc><a>Some whitespace</a> <b>Some more</b></doc>"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

What about text nodes containing *only* whitespace? These are often inserted by XML pretty-printers, and they can usually be totally discarded without altering the meaning of a document. If you add `:ignore_whitespace_nodes => :all` to the parser configuration, REXML will simply decline to create text nodes that contain nothing but whitespace characters. Here's a comparison of `:compress_whitespace` alone, and in conjunction with `:ignore_whitespace_nodes`:

```
text = %{<doc><a>Some text</a>\n <b>Some more</b>\n\n}
REXML::Document.new(text, { :compress_whitespace => :all }).to_s
# => "<doc><a>Some text</a>\n <b>Some more</b>\n</doc>"
REXML::Document.new(text, { :compress_whitespace => :all,
                            :ignore_whitespace_nodes => :all }).to_s
# => "<doc><a>Some text</a><b>Some more</b></doc>"
```

By itself, `:compress_whitespace` shouldn't make a document less human-readable, but `:ignore_whitespace_nodes` almost certainly will.

See Also

- Recipe 1.11, "Managing Whitespace"

Recipe 11.11. Guessing a Document's Encoding

Credit: Mauro Cicio

Problem

You want to know the character encoding of a document that doesn't declare it explicitly.

Solution

Use the Ruby bindings to the `libcharguess` library. Once it's installed, using `libcharguess` is very simple.

Here's an XML document written in Italian, with no explicit encoding:

```
doc = %{<?xml version="1.0"?>
     <menu tipo="specialità" giorno="venerdì">
    <primo_piatto>spaghetti al ragù</primo_piatto>
        <bevanda>frappè</bevanda>
     </menu>}
```

Let's find its encoding:

```
require 'charguess'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
CharGuess::guess doc
# => "windows-1252"
```

This is a pretty good guess: the XML is written in the ISO-8859-1 encoding, and many web browsers treat ISO-8859-1 as Windows-1252.

Discussion

In XML, the character-encoding indication is optional, and may be provided as an attribute of the XML declaration in the first line of the document:

```
<xml version="1.0" encoding="utf-8"?>
```

If this is missing, you must guess the document encoding to process the document. You can assume the lowest common denominator for your community (usually this means assuming that everything is either UTF-8 or ISO-8859-1), or you can use a library that examines the document and uses heuristics to guess the encoding.

As of the time of writing, there are no pure Ruby libraries for guessing the encoding of a document. Fortunately, there is a small Ruby wrapper around the Charguess library. This library can guess with 95% accuracy the encoding of any text whose charset is one of the following: BIG5, HZ, JIS, SJIS, EUC-JP, EUC-KR, EUC-TW, GB2312, Bulgarian, Cyrillic, Greek, Hungarian, Thai, Latin1, and UTF8.

Note that `Charguess` is not XML-or HTML-specific. In fact, it can guess the encoding of an arbitrary string:

```
CharGuess::guess("\xA4\xCF")              # => "EUC-JP"
```

It's fairly easy to install `libcharguess`, since the library is written in portable C++. Unfortunately, it doesn't take care to put its header files in a standard location. This makes it a little tricky to compile the Ruby bindings, which depend on the `charguess.h` header. When you run `extconf.rb` to prepare the bindings, you must explicitly tell the script where to find `libcharguess`'s headers. Here's how you might compile the Ruby bindings to `libcharguess`:

```
$ ruby extconf.rb --with-charguess-include=/location/of/charguess.h
$ make
$ make install
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- To find your way through the jungle of character encodings, the Wikipedia entry on character encodings makes a good reference (http://en.wikipedia.org/wiki/Character_encoding)
- A good source for sample texts in various charsets is http://vancouver-webpages.com/multilingual/
- The XML specification has a section on character encoding autodetection (http://www.w3.org/TR/REC-xml/#sec-guessing)
- The Charguess library is at http://libcharguess.sourceforge.net; its Ruby bindings are available from http://raa.ruby-lang.org/project/charguess

Recipe 11.12. Converting from One Encoding to Another

Credit: Mauro Cicio

Problem

You want to convert a document to a given charset encoding (probably UTF-8).

Solution

If you don't know the document's current encoding, you can guess at it using the Charguess library described in the previous recipe. Once you know the current encoding, you can convert the document to another encoding using Ruby's standard `iconv` library.

Here's an XML document written in Italian, with no explicit encoding:

```
doc = %{<?xml version="1.0"?>
     <menu tipo="specialità" giorno="venerdì">
    <primo_piatto>spaghetti al ragù</primo_piatto>
       <bevanda>frappè</bevanda>
     </menu>}
```

Let's figure out its encoding and convert it to UTF-8:

```
require 'iconv'
require 'charguess' # not necessary if input encoding is known

input_encoding = CharGuess::guess doc                # => "windows-1252"
output_encoding = 'utf-8'

converted_doc = Iconv.new(output_encoding, input_encoding).iconv(doc)

CharGuess::guess(converted_doc)                      # => "UTF-8"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

The heart of the `iconv` library is the `Iconv` class, a wrapper for the Unix 95 `iconv( )` family of functions. These functions translate strings between various encoding systems. Since `iconv` is part of the Ruby standard library, it should be already available on your system.

Iconv works well in conjunction with Charguess: even if Charguess guesses the encoding a little bit wrong (such as guessing Windows-1252 for an ISO-8859-1 document), it always makes a good enough guess that `iconv` can convert the document to another encoding.

Like Charguess, the Iconv library is not XML-or HTML-specific. You can use `libcharguess` and `iconv` together to convert an arbitrary string to a given encoding.

See Also

- Recipe 11.11, "Guessing a Document's Encoding"
- The `iconv` library is documented at http://www.ruby-doc.org/stdlib/libdoc/iconv/rdoc/classes/Iconv.html; you can find pointers to *The Open Group* Unix library specifications

Recipe 11.13. Extracting All the URLs from an HTML Document

Problem

You want to find all the URLs on a web page.

Solution

Do you only want to find links (that is, URLs mentioned in the `HREF` attribute of an `A` tag)? Do you also want to find the URLs of embedded objects like images and applets? Or do you want to find *all* URLs, including ones mentioned in the text of the page?

The last case is the simplest. You can use `URI.extract` to get all the URLs found in a string, or to get only the URLs with certain schemes. Here we'll extract URLs from some HTML, whether or not they're inside A tags:

```
require 'uri'

text = %{"My homepage is at
<a href="http://www.example.com/">http://www.example.com/</a>, and be sure
to check out my weblog at http://www.example.com/blog/. Email me at <a
href="mailto:bob@example.com">bob@example.com</a>.}

URI.extract(text)
# => ["http://www.example.com/", "http://www.example.com/",
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
#        "http://www.example.com/blog/.", "mailto:bob@example.com"]

# Get HTTP(S) links only.
URI.extract(text, ['http', 'https'])
# => ["http://www.example.com/", "http://www.example.com/"
#        "http://www.example.com/blog/."]
```

If you only want URLs that show up inside certain tags, you need to parse the HTML. Assuming the document is valid, you can do this with any of the parsers in the `rexml` library. Here's an efficient implementation using REXML's stream parser. It retrieves URLs found in the `HREF` attributes of `A` tags and the `SRC` attributes of `IMG` tags, but you can customize this behavior by passing a different map to the constructor.

```
require 'rexml/document'
require 'rexml/streamlistener'
require 'set'

class LinkGrabber
  include REXML::StreamListener
  attr_reader :links

def initialize(interesting_tags = {'a' => %w{href}, 'img' => %w{src}}.freeze)
   @tags = interesting_tags
   @links = Set.new
 end
  def tag_start(name, attrs)
    @tags[name].each do |uri_attr|
      @links << attrs[uri_attr] if attrs[uri_attr]
    end if @tags[name]
  end

  def parse(text)
    REXML::Document.parse_stream(text, self)
  end
end

grabber = LinkGrabber.new
grabber.parse(text)
grabber.links
# => #<Set: {"http://www.example.com/", "mailto:bob@example.com"}>
```

Discussion

The `URI.extract` solution uses regular expressions to find everything that looks like a URL. This is faster and easier to write than a REXML parser, but it will find *every* absolute URL in the document, including any mentioned in the text and any in the document's initial DOCTYPE. It will *not* find relative URLs hidden within `HREF` attributes, since those don't start with an access scheme like "http://".

`URI.extract` treats the period at the end of the first sentence ("check out my weblog at...")as though it were part of the URL. URLs contained within English text are often ambiguous in this way. "http://www.example.com/blog/." is a perfectly valid URL and might be correct, but that period is probably just punctuation. Accessing the URL is the only sure way to know for sure, but it's almost always safe to strip those characters:

```
END_CHARS = %{.,'?!:;}
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
URI.extract(text, ['http']).collect { |u| END_CHARS.index(u[-1]) ? u.chop : u }
# => ["http://www.example.com/", "http://www.example.com/",
#         "http://www.example.com/blog/"]
```

The parser solution defines a listener that hears about every tag present in its `interesting_tags` map. It checks each tag for attributes that tend to contain URLs: "href" for `<a>` tags and "src" for `<img>` tags, for instance. Every URL it finds goes into a set.

The use of a set here guarantees that the result contains no duplicate URLs. If you want to gather (possibly duplicate)URLs in the order they were found in the document, use a list, the way `URI.extract` does.

The `LinkGrabber` solution will not find URLs in the text portions of the document, but it will find relative URLs. Of course, you still need to know how to turn relative URLs into absolute URLs. If the document has a `<base>` tag, you can use that. Otherwise, the base depends on the original URL of the document.

Here's a subclass of `LinkGrabber` that changes relative links to absolute links if possible. Since it uses `URI.join`, which returns a `URI` object, your set will end up containing `URI` objects instead of strings:

```
class AbsoluteLinkGrabber < LinkGrabber
  include REXML::StreamListener
  attr_reader :links

  def initialize(original_url = nil,
            interesting_tags = {'a' => %w{href}, 'img' => %w{src}}.freeze)
    super(interesting_tags)
    @base = original_url
  end

  def tag_start(name, attrs)
    if name == 'base'
      @base = attrs['href']
    end
    super
  end

  def parse(text)
    super
    # If we know of a base URL by the end of the document, use it to
    # change all relative URLs to absolute URLs.
    @links.collect! { |l| URI.join(@base, l) } if @base
  end
end
```

If you want to use the parsing solution, but the web page has invalid HTML that chokes the REXML parsers (which is quite likely), try the techniques mentioned in Recipe 11.5.

Almost 20 HTML tags can have URLs in one or more of their attributes. If you want to collect *every* URL mentioned in an appropriate part of a web page, here's a big map you can pass in to the constructor of `LinkGrabber` or `AbsoluteLinkGrabber`:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
URL_LOCATIONS = { 'a' => %w{href},
  'area' => %w{href},
  'applet' => %w{classid},
  'base' => %w{href},
  'blockquote' => %w{cite},
  'body' => %w{background},
  'codebase' => %w{classid},
  'del' => %w{cite},
  'form' => %w{action},
  'frame' => %w{src longdesc},
  'iframe' => %w{src longdesc},
  'input' => %w{src usemap},
  'img' => %w{src longdesc usemap},
  'ins' => %w{cite},
  'link' => %w{href},
  'object' => %w{usemap archive codebase data},
  'profile' => %w{head},
  'q' => %w{cite},
  'script' => %w{src}}.freeze
```

See Also

- Recipe 11.4, "Navigating a Document with XPath"
- Recipe 11.5, "Parsing Invalid Markup"
- I compiled that big map of URI attributes from the W3C's Index of Attributes for HTML 4.0; look for the attributes of type `%URI;` (http://www.w3.org/TR/REC-html40/index/attributes.html)

Recipe 11.14. Transforming Plain Text to HTML

Problem

You want to add simple markup to plaintext and turn it into HTML.

Solution

Use RedCloth, written by "why the lucky stiff" and available as the `RedCloth` gem. It extends Ruby's string class to support Textile markup: its `to_html` method converts Textile markup to HTML.

Here's a simple document:

```
require 'rubygems'
require 'redcloth'

text = RedCloth.new %{Who would ever write "HTML":http://www.w3.org/MarkUp/
markup directly?

I mean, _who has the time_? Nobody, that's who:

|_. Person |_. Has the time?        |
|   Jake   |     No                 |
|  Alice   |     No                 |
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
|  Rodney  | Not since the accident |
}

puts text.to_html
# <p>Who would ever write
# <a href="http://www.w3.org/MarkUp/"><span class="caps">HTML</span></a>
# markup directly?</p>
#
# <p>I mean, <em>who has the time</em>? Nobody, that&#8217;s who:</p>
#
# <table>
#  <tr>
#   <th>Person </th>
#   <th>Has the time?        </th>
# </tr>
# …
```

The Textile version is more readable and easier to edit.

Discussion

The Textile markup language lets you produce HTML without having to write any HTML. You just add punctuation to plain text, to convey what markup you'd like. Paragraph breaks are represented by blank lines, italics by underscores, tables by ASCII-art drawings of tables.

A text-based markup that converts to HTML is very useful in weblog and wiki software, where the markup will be edited many times. It's also useful for hiding the complexity of HTML from new computer users. We wrote this entire book using a Textile-like markup, though it was converted to Docbook instead of HTML.

See Also

- The RedCloth homepage (http://www.whytheluckystiff.net/ruby/redcloth/)
- A comprehensive Textile reference (http://hobix.com/textile/)and a quick reference (http://hobix.com/textile/quick.html)
- You can experiment with Textile markup at the language's homepage (http://www.textism.com/tools/textile/)
- Markdown (http://daringfireball.net/projects/markdown/)is another popular simple markup language for plain text; you can turn Markdown text to XHTML with the `BlueCloth` gem (project page: http://www.deveiate.org/projects/BlueCloth); because BlueCloth and RedCloth both define `String#to_html`, it's not easy to use them both in the same program

Recipe 11.15. Converting HTML Documents from the Web into Text

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to get a text summary of a web site.

Solution

The `open-uri` library is the easiest way to grab the content of a web page; it lets you open a URL as though it were a file:

```
require 'open-uri'

example = open('http://www.example.com/')
# => #<StringIO:0xb7bb601c>

html = example.read
```

As with a file, the `read` method returns a string. You can do a series of `sub` and `gsub` methods to clean the code into a more readable format.

```
plain_text = html.sub(%r{<body.*?>(.*?)</body>}mi, '\1').gsub(/<.*?>/m, ' ').
    gsub(%r{(\n\s*){2}}, "\n\n")
```

Finally, you can use the standard `CGI` library to unescape HTML entities like `<` into their ASCII equivalents (<):

```
require 'cgi'
plain_text = CGI.unescapeHTML(plain_text)
```

The final product:

```
puts plain_text
# Example Web Page
#
# You have reached this web page by typing "example.com",
# "example.net",
# or "example.org" into your web browser.
# These domain names are reserved for use in documentation and are not available
# for registration. See RFC
# 2606 , Section 3.
```

Discussion

The `open-uri` library extends the `open` method so that you can access the contents of web pages and FTP sites with the same interface used for local files.

The simple regular expression substitutions above do nothing but remove HTML tags and clean up excess whitespace. They work well for well-formatted HTML, but the web is full of mean and ugly HTML, so you may consider taking a more involved approach. Let's define a `HTMLSanitizer` class to do our dirty business.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

An `HTMLSanitizer` will start off with some HTML, and through a series of search-and-replace operations transform it into plain text. Different HTML tags will be handled differently. The contents of some HTML tags should simply be removed in a plaintext rendering. For example, you probably don't want to see the contents of `<head>` and `<script>` tags. Other tags affect what the rendition should look like, for instance, a `<p>` tag should be represented as a blank line:

```
require 'open-uri'
require 'cgi'

class HTMLSanitizer
  attr_accessor :html

  @@ignore_tags = ['head', 'script', 'frameset' ]
  @@inline_tags = ['span', 'strong', 'i', 'u' ]
  @@block_tags = ['p', 'div', 'ul', 'ol' ]
```

The next two methods define the skeleton of our HTML sanitizer:

```
  def initialize(source='')
    begin
      @html = open(source).read
    rescue Errno::ENOENT
      # If it's not a file, assume it's an HTML string
      @html = source
    end
  end

  def plain_text
    # remove pre-existing blank spaces between tags since we will
    # be adding spaces on our own
    @plain_text = @html.gsub(/\s*(<.*?>)/m, '\1')

    handle_ignore_tags
    handle_inline_tags
    handle_block_tags
    handle_all_other_tags

    return CGI.unescapeHTML(@plain_text)
  end
```

Now we need to fill in the `handle_` methods defined by `HTMLSanitizer#plain_text`. These methods perform search-and-replace operations on the `@plain_text` instance variable, gradually transforming it from HTML into plain text. Because we are modifying `@plain_text` in place, we will need to use `String#gsub!` instead of `String#gsub`.

```
  private

  def tag_regex(tag)
    %r{<#{tag}.*?>(.*?)</#{tag}>}mi
  end

  def handle_ignore_tags
   @@ignore_tags.each { |tag| @plain_text.gsub!(tag_regex(tag), '') }
  end
  def handle_inline_tags
    @@inline_tags.each { |tag| @plain_text.gsub!(tag_regex(tag), '\1 ') }
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  end
  def handle_block_tags
    @@block_tags.each { |tag| @plain_text.gsub!(tag_regex(tag), "\n\\1\n") }
  end

  def handle_all_other_tags
    @plain_text.gsub!(/&#x00A;/mi, "\n")
    @plain_text.gsub!(/<.*?>/m, ' ')
    @plain_text.gsub!(/(\n\s*){2}/, "\n\n")
  end
end
```

To use this class, simply initialize it with a URL and call the `plain_text` method:

```
puts HTMLSanitizer.new('http://slashdot.org/').plain_text
# Stories
# Slash Boxes
# Comments
#
# Slashdot
#
# News for nerds, stuff that matters
#
# Login
#
# Why Login? Why Subscribe?
# …
```

See Also

- Recipe 14.1, "Grabbing the Contents of a Web Page"
- For a more sophisticated text renderer, parse the HTML document with the techniques described in Recipe 11.2, "Extracting Data from a Document's Tree Structure," or Recipe 11.5, "Parsing Invalid Markup"

Recipe 11.16. A Simple Feed Aggregator

Credit: Rod Gaither

XML is the basis for many specialized langages. One of the most popular is RSS, an XML format often used to store lists of articles from web pages. With a tool called an aggregator, you can collect weblog entries and articles from several web sites' RSS feeds, and read all those web sites at once without having to skip from one to the other. Here, we'll create a simple aggregator in Ruby.

Before aggregating RSS feeds, let's start by reading a single one. Fortunately we have several options for parsing RSS feeds into Ruby data structures. The Ruby standard library has built-in support for the three major versions of the RSS format (0.9, 1.0, and 2.0). This example uses the standard `rss` library to parse an RSS 2.0 feed and print out the titles of the items in the feed:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'rss/2.0'
require 'open-uri'

url = 'http://www.oreillynet.com/pub/feed/1?format=rss2'
feed = RSS::Parser.parse(open(url).read, false)
puts "=== Channel: #{feed.channel.title} ==="
feed.items.each do |item|
  puts item.title
  puts " (#{item.link})"
  puts
  puts item.description
end
# === Channel: O'Reilly Network Articles ===
# How to Make Your Sound Sing with Vocoders
# (http://digitalmedia.oreilly.com/2006/03/29/vocoder-tutorial-and-tips.html)
# …
```

Unfortunately, the standard `rss` library is a little out of date. There's a newer syndication format called Atom, which serves the same purpose as RSS, and the `rss` library doesn't support it. Any serious aggregator must support all the major syndication formats.

So instead, our aggregator will use Lucas Carlson's Simple RSS library, available as the `simple-rss` gem. This library supports the three main versions of RSS, plus Atom, and it does so in a relaxed way so that ill-formed feeds have a better chance of being read.

Here's the example above, rewritten to use Simple RSS. As you can see, only the name of the class is different:

```
require 'rubygems'
require 'simple-rss'
url = 'http://www.oreillynet.com/pub/feed/1?format=rss2'
feed = RSS::Parser.parse(open(url), false)
puts "=== Channel: #{feed.channel.title} ==="
feed.items.each do |item|
  puts item.title
  puts " (#{item.link})"
  puts
  puts item.description
end
```

Now we have a general method of reading a single RSS or Atom feed. Time to work on some aggregation!

Although the aggregator will be a simple Ruby script, there's no reason not to use Ruby's object-oriented features. Our approach will be to create a class to encapsulate the aggregator's data and behavior, and then write a sample program to use the class.

The `RSSAggregator` class that follows is a bare-bones aggregator that reads from multiple syndication feeds when instantiated. It uses a few simple methods to expose the data it has read.

```
#!/usr/bin/ruby
# rss-aggregator.rb - Simple RSS and Atom Feed Aggregator
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'rubygems'
require 'simple-rss'
require 'open-uri'

class RSSAggregator
  def initialize(feed_urls)
    @feed_urls = feed_urls
    @feeds = []
    read_feeds
  end

  protected
  def read_feeds
    @feed_urls.each { |url| @feeds.push(SimpleRSS.new(open(url).read)) }
  end
  public
  def refresh
    @feeds.clear
    read_feeds
  end

  def channel_counts
    @feeds.each_with_index do |feed, index|
      channel = "Channel(#{index.to_s}): #{feed.channel.title}"
      articles = "Articles: #{feed.items.size.to_s}"
      puts channel + ', ' + articles
    end
  end

  def list_articles(id)
    puts "=== Channel(#{id.to_s}): #{@feeds[id].channel.title} ==="
    @feeds[id].items.each { |item| puts ' ' + item.title }
  end

  def list_all
    @feeds.each_with_index { |f, i| list_articles(i) }
  end
end
```

Now we just need a few more lines of code to instantiate and use an `RSSAggregator` object:

```
test = RSSAggregator.new(ARGV)
test.channel_counts
puts "\n"
test.list_all
```

Here's the output from a run of the test program against a few feed URLs:

```
$ ruby rss-aggregator.rb http://www.rubyriver.org/rss.xml \
  http://rss.slashdot.org/Slashdot/slashdot \
  http://www.oreillynet.com/pub/feed/1 \
  http://safari.oreilly.com/rss/
Channel(0): RubyRiver, Articles: 20
Channel(1): Slashdot, Articles: 10
Channel(2): O'Reilly Network Articles, Articles: 15
Channel(3): O'Reilly Network Safari Bookshelf, Articles: 10
=== Channel(0): RubyRiver ===
 Mantis style isn't eas…
 It's wonderful when tw…
 Red tailed hawk
 37signals
 …
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

While a long way from a fully functional RSS aggregator, this program illustrates the basic requirements of any real aggregator. From this starting point, you can expand and refine the features of `RSSAggregator`.

One very important feature missing from the aggregator is support for the If-Modified-Since HTTP request header. When you call `RSSAggregator#refresh`, your aggregator downloads the specified feeds, even if it just grabbed the same feeds and none of them have changed since then. This wastes bandwidth.

Polite aggregators keep track of when they last grabbed a certain feed, and when they request it again they do a conditional request by supplying an HTTP request header called If-Modified Since. The details are a little beyond our scope, but basically the web server serves the reuqested feed only if it has changed since the last time the `RSSAggregator` downloaded it.

Another important feature our `RSSAggregator` is missing is the ability to store the articles it fetches. A real aggregator would store articles on disk or in a database to keep track of which stories are new since the last fetch, and to keep articles available even after they become old news and drop out of the feed.

Our simple aggregator counts the articles and lists their titles for review, but it doesn't actually provide access to the article detail. As seen in the first example, the `SimpleRSS.item` has a `link` attribute containing the URL for the article, and a `description` attribute containing the (possibly HTML) body of the article. A real aggregator might generate a list of articles in HTML format for use in a browser, or convert the body of each article to text for output to a terminal.

See Also

- Recipe 14.1, "Grabbing the Contents of a Web Page"
- Recipe 14.3, "Customizing HTTP Request Headers"
- Recipe 11.15, "Converting HTML Documents from the Web into Text"
- A good comparison of the RSS and Atom formats (http://www.intertwingly.net/wiki/pie/Rss20AndAtom10Compared)
- Details on the Simple RSS project (http://simple-rss.rubyforge.org/)
- The FeedTools project has a more sophisticated aggregator library that supports caching and If-Modified-Since; see http://sporkmonger.com/projects/feedtools/ for details
- "HTTP Conditional Get for RSS Hackers" is a readable introduction to If-Modified-Since (http://fishbowl.pastiche.org/2002/10/21/http_conditional_get_for_rss_hackers)

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01