By icarus

Table of Contents

Climbing A Tree

In the first part of this article, I said that there were two approaches to parsing an XML document. The event−based SAX approach is one; the other uses the DOM, or Document Object Model, to build a tree representation of the XML data structures in the document, and then offers built−in methods to navigate through this tree. Once a particular node has been reached, built−in methods can be used to obtain the value of the node, and use it within the script.

Over the next few pages, I'm going to take a look at a Perl package that supports this tree−based approach. I'll also attempt to duplicate the previous examples, marking up books and recipes with HTML tags, in order to demonstrate how either technique can be used to achieve the same result. Finally, I'll briefly discuss the pros and cons of the SAX and DOM approaches, and point you to some links for your further education.

Let's get started, shall we?

Meet Joe Cool

Perl comes with a DOM parser based on the expat library created by James Clark; it's implemented as a Perl package named XML::DOM, and currently maintained by T. J. Mather. If you don't already have it, you should download and install it before proceeding further; you can get a copy from CPAN (http://www.cpan.org/).

This DOM parser works by reading an XML document and creating objects to represent the different parts of that document. Each of these objects comes with specific methods and properties, which can be used to manipulate and access information about it. Thus, the entire XML document is represented as a "tree" of these objects, with the DOM parser providing a simple API to move between the different branches of the tree.

The parser itself supports all the different structures typically found in an XML document − elements, attributes, namespaces, entities, notations et al − but our focus here will be primarily on elements and the data contained within them. If you're interested in the more arcane aspects of XML − as you will have to be to do anything complicated with the language − the XML::DOM package comes with some truly excellent documentation, which gets installed when you install the package. Make it your friend, and you'll find things considerably easier.

Let's start things off with a simple example:

```
#!/usr/bin/perl

# create an XML-compliant string
$xml = "<?xml version=\"1.0\"?><me><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";

# include package
use XML::DOM;

# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parse($xml);

# print tree as string
print $doc->toString();

# end
```

In this case, a new instance of the parser is created and assigned to the variable $xp. This object instance can now be used to parse the XML data via its parse() function:

```
# instantiate parser
```

```
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parse($xml);
```

You'll remember the parse() function from the first part of this article − it was used by the SAX parser to parse a string. When you think about it, this isn't really all that remarkable − the XML::DOM package is built on top of the XML::Parser package, and therefore inherits many of the latter's methods.

With that in mind, it follows that the DOM parser should also be able to read an XML file directly, simply by using the parsefile() method, instead of the parse() method:

```
#!/usr/bin/perl

# XML file
$file = "me.xml";

# include package
use XML::DOM;

# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parsefile($file);

# print tree as string
print $doc->toString();

# end
```

The results of successfully parsing an XML document − whether string or file − is an object representation of the XML document (actually, an instance of the Document class). In the example above, this object is called $doc.

```
# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parsefile($file);
```

This Document object comes with a bunch of interesting methods − and one of the more useful ones is the toString() method, which returns the current document tree as a string. In the examples above, I've used this method to print the entire document to the console.

```
# print tree as string
print $doc->toString();
```

It should be noted that this isn't all that great an example of how to use the toString() method. Most often, this method is used during dynamic XML tree generation, when an XML tree is constructed in memory from a database or elsewhere. In such situations, the toString() method comes in handy to write the final XML tree to a file or send it to a parser for further processing.

Parents And Their Children

The Document object comes with another useful method, one which enables you to gain access to information about the document's XML version and character encoding. It's called the getXMLDecl() method, and it returns yet another object, this one representing the standard XML declaration that appears at the top of every XML document. Take a look:

```
#!/usr/bin/perl

# create an XML-compliant string
$xml = "<?xml version=\"1.0\"
encoding=\"utf-8\"?><me><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";

# include package
use XML::DOM;

# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parse($xml);

# get XML PI
$decl = $doc->getXMLDecl();

# get XML version
print $decl->getVersion();

# get encoding
print $decl->getEncoding();

# get whether standalone
print $decl->getStandalone();

# end
```

As you can see, the newly-created XMLDecl object comes with a bunch of object methods of its own. These methods provide a simple way to access the document's XML version, character encoding and status.

Using the Document object, it's also possible to obtain references to other nodes in the XML tree, and manipulate them using standard methods. Since the entire document is represented as a tree, the first step is always to obtain a reference to the tree root, or the outermost document element, and use this a stepping stone to other, deeper branches. Consider the following example, which demonstrates how to do this:

```
#!/usr/bin/perl

# create an XML-compliant string
$xml = "<?xml version=\"1.0\"?><me><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";

# include package
use XML::DOM;

# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parse($xml);

# get root node "me"
$root = $doc->getDocumentElement();

# end
```

An option here would be to use the getChildNodes() method, which is a common method available to every single node in the document tree. The following code snippet is identical to the one above:

```
#!/usr/bin/perl

# create an XML-compliant string
$xml = "<?xml version=\"1.0\"?><me><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";

# include package
use XML::DOM;

# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parse($xml);

# get root node "me"
@children = $doc->getChildNodes();
$root = $children[0];

# end
```

Note that the getChildNodes() method returns an array of nodes under the current node; each of these nodes is again an object instance of the Node class, and comes with methods to access the node name, type and

content. Let's look at that next.

What's In A Name?

Once you've obtained a reference to a node, a number of other methods become available to help you obtain the name and value of that node, as well as references to parent and child nodes. Take a look:

```
#!/usr/bin/perl

# create an XML-compliant string
$xml = "<?xml version=\"1.0\"?><me><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";

# include package
use XML::DOM;

# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parse($xml);

# get root node
$root = $doc->getDocumentElement();

# get name of root node
# returns "me"
print $root->getNodeName();

# get children as array
@children = $root->getChildNodes();

# this is the "name" element under "me"
# I could also have used $root->getFirstChild() to get here
$firstChild = $children[0];

# returns "name"
print $firstChild->getNodeName();

# returns "1"
print $firstChild->getNodeType();

# now to access the value of the text node under "name"
$text = $firstChild->getFirstChild();

# returns "Joe Cool"
print $text->getData();

# returns "#text"
```

```
print $text->getNodeName();

# returns "3"
print $text->getNodeType();

# go back up the tree
# start from the "name" element and get its parent
$parent = $firstChild->getParentNode();

# check the name - it should be "me"
# yes it is!
print $parent->getNodeName();

# end
```

As you can see, the getNodeName() and getNodeType() methods provide access to basic information about the node currently under examination. The children of this node can be obtained with the getChildNodes() method previously discussed, and node parents can be obtained with the getParentNode() method. It's fairly simple, and - once you play with it a little - you'll get the hang of how it works.

A quick note on the getNodeType() method above: every node is of a specific type, and this property returns a numeric code corresponding to the type. A complete list of defined types is available in the Perl documentation for the XML::DOM package.

Note also that the text within an element's opening and closing tags is treated as a child node of the corresponding element node, and is returned as an object. This object comes with a getData() method, which returns the actual content nested within the element's opening and closing tags. You'll see this again in a few pages.

Welcome To The Human Race

Just as it's possible to access elements and their content, it's also possible to access element attributes and their values. The getAttributes() method of the Node object provides access to a list of all available attributes, and the getNamedItem() and getValue() methods make it possible to access specific attributes and their values. Take a look at a demonstration of how it all works:

```
#!/usr/bin/perl

# create an XML-compliant string
$xml = "<?xml version=\"1.0\"?><me species=\"human\"><name>Joe
Cool</name><age>24</age><sex>male</sex></me>";

# include package
use XML::DOM;

# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parse($xml);

# get root node (Node object)
$root = $doc->getDocumentElement();

# get attributes (NamedNodeMap object)
$attribs = $root->getAttributes();

# get specific attribute (Attr object)
$species = $attribs->getNamedItem("species");

# get value of attribute
# returns "human"
print $species->getValue();

# end
```

Getting to an attribute value is a little more complicated than getting to an element. But hey − no gain without pain, right?

Building A Library

Using this information, it's pretty easy to re−create our first example using the DOM parser. Here's the XML data,

```
<?xml version="1.0"?>

<library>
<book>
<title>Dreamcatcher</title>
<author>Stephen King</author>
<genre>Horror</genre>
<pages>899</pages>
<price>23.99</price>
<rating>5</rating>
</book>

<book>
<title>Mystic River</title>
<author>Dennis Lehane</author>
<genre>Thriller</genre>
<pages>390</pages>
<price>17.49</price>
<rating>4</rating>
</book>

<book>
<title>The Lord Of The Rings</title>
<author>J. R. R. Tolkien</author>
<genre>Fantasy</genre>
<pages>3489</pages>
<price>10.99</price>
<rating>5</rating>
</book>

</library>
```

and here's the script which does all the work.

```
#!/usr/bin/perl

# XML file
$file = "library.xml";

# array of ratings
```

```
@ratings = ("Words fail me!", "Terrible", "Bad",
"Indifferent", "Good",
"Excellent");

# include package
use XML::DOM;

# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parsefile($file);

# set up HTML page
print "Content-Type: text/html\n\n";
print "<html><head></head><body>";
print "<h2>The Library</h2>";
print "<table border=1 cellspacing=1 cellpadding=5> <tr> <td
align=center>Title</td> <td align=center>Author</td> <td
align=center>Price</td> <td align=center>User Rating</td>
</tr>";

# get root node
$root = $doc->getDocumentElement();

# get children
@books = $root->getChildNodes();

# iterate through book list
foreach $node (@books)
{
print "<tr>";
# if element node
if ($node->getNodeType() == 1)
{
# get children
# this is the "title", "author"... level
@children = $node->getChildNodes();

# iterate through child nodes
foreach $item (@children)
{
# check element name
if (lc($item->getNodeName) eq "title")
{
# print text node contents under this element
print "<td><i>" . $item->getFirstChild()->getData .
"</i></td>";
}
```

```
elsif (lc($item->getNodeName) eq "author")
{
print "<td>" . $item->getFirstChild()->getData . "</td>";
}
elsif (lc($item->getNodeName) eq "price")
{
print "<td>\$" . $item->getFirstChild()->getData . "</td>";
}
elsif (lc($item->getNodeName) eq "rating")
{
$num = $item->getFirstChild()->getData;
print "<td>" . $ratings[$num] . "</td>";
}
}
}
print "</tr>";
}

print "</table></body></html>";

# end
```

This may appear complex, but it isn't really all that hard to understand. I've first obtained a reference to the root of the document tree, $root, and then to the children of that root node; these children are returned as a regular Perl array. I've then used a "foreach" loop to iterate through the array, navigate to the next level, and print the content found in the nodes, with appropriate formatting. The numerous "if" statements you see are needed to check the name of each node and then add appropriate HTML formatting to it.

As explained earlier, the data itself is treated as a child text node of the corresponding element node. Therefore, whenever I find an element node, I've used the node's getFirstChild() method to access the text node under it, and the getData() method to extract the data from that text node.

Here's what it looks like:

The Library

Title	Author	Price	User Rating
Dreamcatcher	Stephen King	$23.99	Excellent
Mystic River	Dennis Lehane	$17.49	Good
The Lord Of The Rings	J. R. R. Tolkien	$10.99	Excellent

Anyone For Chicken?

I can do the same thing with the second example as well. However, since there are quite a few levels to the document tree, I've decided to use a recursive function to iterate through the tree, rather than a series of "if" statements.

Here's the XML file,

```
<?xml version="1.0"?>

<recipe>

<name>Chicken Tikka</name>
<author>Anonymous</author>
<date>1 June 1999</date>

<ingredients>

<item>
<desc>Boneless chicken breasts</desc>
<quantity>2</quantity>
</item>

<item>
<desc>Chopped onions</desc>
<quantity>2</quantity>
</item>

<item>
<desc>Ginger</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Garlic</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Red chili powder</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Coriander seeds</desc>
<quantity>1 tsp</quantity>
</item>
```

```
<item>
<desc>Lime juice</desc>
<quantity>2 tbsp</quantity>
</item>

<item>
<desc>Butter</desc>
<quantity>1 tbsp</quantity>
</item>
</ingredients>

<servings>
3
</servings>

<process>
<step>Cut chicken into cubes, wash and apply lime juice and
salt</step>
<step>Add ginger, garlic, chili, coriander and lime juice in a
separate
bowl</step>
<step>Mix well, and add chicken to marinate for 3-4
hours</step>
<step>Place chicken pieces on skewers and barbeque</step>
<step>Remove, apply butter, and barbeque again until meat is
tender</step>
<step>Garnish with lemon and chopped onions</step>
</process>

</recipe>
```

and here's the script which parses it.

```
#!/usr/bin/perl

# XML file
$file = "recipe.xml";

# hash of tag names mapped to HTML markup
# "recipe" => start a new block
# "name" => in bold
# "ingredients" => unordered list
# "desc" => list items
# "process" => ordered list
# "step" => list items
%startTags = (
```

```
"name" => "<font size=+2>",
"date" => "<i>(",
"author" => "<b>",
"servings" => "<i>Serves ",
"ingredients" => "<h3>Ingredients:</h3><ul>",
"desc" => "<li>",
"quantity" => "(",
"process" => "<h3>Preparation:</h3><ol>",
"step" => "<li>"
);

# close tags opened above
%endTags = (
"name" => "</font><br>",
"date" => ")</i>",
"author" => "</b>",
"ingredients" => "</ul>",
"quantity" => ")",
"servings" => "</i>",
"process" => "</ol>"
);

# this function accepts an array of nodes as argument,
# iterates through it and prints HTML markup for each tag it
finds.
# for each node in the array, it then gets an array of the
node's children,
and
# calls itself again with the array as argument (recursion)
sub printData()
{
my (@nodeCollection) = @_;
foreach $node (@nodeCollection)
{
print $startTags{$node->getNodeName()};
print $node->getFirstChild()->getData();
my @children = &getChildren($node);
printData(@children);
print $endTags{$node->getNodeName()};
}
}

# this function accepts a node
# and returns all the element nodes under it (its children)
# as an array
sub getChildren()
{
my ($node) = @_;
# get children of this node
```

```
my @temp = $node->getChildNodes();
my $count = 0;
my @collection;

# iterate through children
foreach $item (@temp)
{
# if this is an element
# (need this to strip out text nodes containing whitespace)
if ($item->getNodeType() == 1)
{
# add it to the @collection array
$collection[$count] = $item;
$count++;
}
}

# return node collection
return @collection;
}

use XML::DOM;

# instantiate parser
$xp = new XML::DOM::Parser();

# parse and create tree
$doc = $xp->parsefile($file);

# send standard header to browser
print "Content-Type: text/html\n\n";

# print HTML header
print "<html><head></head><body><hr>";

# get root node
$root = $doc->getDocumentElement();

# get children
@children = &getChildren($root);

# run a recursive function starting here
&printData(@children);

print "</table></body></html>";

# end
```

In this case, I've utilized a slightly different method to mark up the XML. I've first initialized a couple of hashes to map XML tags to corresponding HTML markup, in much the same manner as I did last time. Next, I've used DOM functions to obtain a reference to the first set of child nodes in the DOM tree.

This initial array of child nodes is used to "seed" my printData() function, a recursive function which takes an array of child nodes, matches their tag names to values in the associative arrays, and outputs the corresponding HTML markup to the browser. It also obtains a reference to the next set of child nodes, via the getChildren() function, and calls itself with the new node collection as argument.

By using this recursive function, I've managed to substantially reduce the number of "if" conditional statements in my script; the code is now easier to read, and also structured more logically.

Here's what it looks like:

Chicken Tikka

Anonymous *(1 June 1999)*

Ingredients:

- Boneless chicken breasts (2)
- Chopped onions (2)
- Ginger (1 tsp)
- Garlic (1 tsp)
- Red chili powder (1 tsp)
- Coriander seeds (1 tsp)
- Lime juice (2 tbsp)
- Butter (1 tbsp)

Serves 3

Preparation:

1. Cut chicken into cubes, wash and apply lime juice and salt
2. Add ginger, garlic, chili, coriander and lime juice in a separate bowl
3. Mix well, and add chicken to marinate for 3-4 hours
4. Place chicken pieces on skewers and barbeque
5. Remove, apply butter, and barbeque again until meat is tender
6. Garnish with lemon and chopped onions

Conclusions...

As you can see, you can parse a document using either DOM or SAX, and achieve the same result. The difference is that the DOM parser is a little slower, since it has to build a complete tree of the XML data, whereas the SAX parser is faster, since it's calling a function each time it encounters a specific tag type. You should experiment with both methods to see which one works better for you.

There's another important difference between the two techniques. The SAX approach is event−centric − as the parser travels through the document, it executes specific functions depending on what it finds. Additionally, the SAX approach is sequential − tags are parsed one after the other, in the sequence in which they appear. Both these features add to the speed of the parser; however, they also limit its flexibility in quickly accessing any node of the DOM tree.

As opposed to this, the DOM approach builds a complete tree of the document in memory, making it possible to easily move from one node to another (in a non−sequential manner). Since the parser has the additional overhead of maintaining the tree structure in memory, speed is an issue here; however, navigation between the various "branches" of the tree is easier. Since the approach is not dependent on events, developers need to use the exposed methods and attributes of the various DOM objects to process the XML data.

...And Links

That just about concludes this little tour of parsing XML data with Perl. I've tried to keep it as simple as possible, and there are numerous aspects of XML I haven't covered here. If you're interested in learning more about XML and XSL, you should visit the following links:

The XML specification, at http://www.w3.org/TR/2000/REC−xml−20001006

The XSLT specification, at http://www.w3.org/TR/xslt.html

The SAX project, at http://www.saxproject.org/

The W3C's DOM specification, at http://www.w3.org/DOM/

A number of developers have built and released Perl packages to handle XML data − if you're ever on a tight deadline, using these packages might save you some development time. Take a look at the following links for more information:

The Perl XML module list, at http://www.perlxml.com/modules/perl−xml−modules.html

CPAN, at http://www.cpan.org/

The Perl XML FAQ, at http://www.perlxml.com/faq/perl−xml−faq.html

And that's just about all for the moment. I hope you found this interesting, and that it helped to make the road ahead a little clearer. Till next time, stay healthy!

Note: All examples in this article have been tested on Linux/i586 with Perl 5.005. Examples are illustrative only, and are not meant for a production environment. YMMV!