Comparing Perl XML parsers

SPOILED FOR CHOICE

XML is one of today's most popular data exchange formats. Perl has a huge collection of methods for handling XML. This month's Perl column discusses the pros and cons of the most common XML modules to help you choose the best tool for your job. **BY MICHAEL SCHILLI**

When it comes to handling XML documents, Perl certainly sticks to its motto: "There is more than one way to do it." The Perl community really has developed many useful modules for handling XML. In this article, I will examine the different approaches these various Perl modules take to handling XML-based data. I'll start by considering the case of the example data shown in Figure 1. The file shown in Figure 1 contains two records of type <*cd*> within a <*result*> tag. Each of these XML records within the file consists of tags for the <*artists*> and <*title*> of a CD, where <*artists*> can include one or multiple <*artist*> tags.

Figure 1: The XML sample data is a subset of a CD archive.

Keep It Simple

The easiest way to parse XML in Perl is to use the XML::Simple Perl module from CPAN. The XML::Simple module exports the *XMLin* function, which reads a file or string with XML data and stores the data as a Perl data structure, as follows:

```
use XML::Simple;
my $ref = XMLin("data.xml");
```

Figure 2 shows a dump of the resulting data structure in *$ref*. You may have noticed two things: depending on the number of artists in <*artists*>, the resulting data structure is either a scalar or an array. This would make things difficult later. However, you can specify the *ForceArray* option to ensure that the field will be represented by an array. Calling *XMLin("data.xml", ForceArray => ['artist']);* ensures that *$ref->{cd}->[0]->{artists}->{artist}* will always return a reference to an array, even if there is only one artist in the source.

Listing 1: xptitles

```
01 #!/usr/bin/perl -w
02 use strict;
03 use XML::LibXML;
04
05 my $x = XML::LibXML->new()
06   or die "new failed";
07
08 my $d =
09   $x->parse_file("data.xml")
10   or die "parse failed";
11
12 my $titles =
13   "/result/cd/title/text()";
14
15 for my $title (
16   $d->findnodes($titles) ) {
17   print $title->toString(),
18     "\n";
19 }
```

Figure 2: This is the data structure that XML::Simple uses to store the sample data.

Additionally, -> {*artists*}-> {*artist*} is a bit clumsy, as -> {*artists*} does not have any subelements, apart from -> {*artist*}. XML::Simple has a *GroupTags* option that allows developers to collapse hierarchies.

The following code:

```
XMLin("data.xml",
  ForceArray => ['artist'],
  GroupTags  =>
    {'artists' => 'artist'});
```

creates the data structure shown in Figure 3, which is quite easy to handle. For example, you could use a simple For loop to find serial numbers:

```
for my $cd (@{$ref->{cd}}) {
    print $cd->
    {serial}, "\n";
}
```

XML::Simple parses the complete XML file into main memory, which is useful for small files. If you have a large XML file, however, this approach may be inefficient, or it could even cause the program to run out of memory.

Twisted Paths

If you are a fan of terse notation, you will love using XPath to navigate the XML jungle. The XML::LibXML module from CPAN relies on the Gnome project's libxml2 library and offers developers the *findnodes* method for accessing XML elements using XPath notation.

For example, the Xpath notation for retrieving the text content of all <*title*> elements is */result/cd/title/text()*: this starts at the document root, /, climbs down into the <*results*>, <*cd*>, and <*title*> elements, before using *text()* to retrieve the text content. Alternatively, you could simply specify *//title/text()* to tell XPath to locate all <*title*> elements no matter the level at which they are located in the XML hierarchy. *xptitles* in Listing 1 shows that the *findnodes()* method returns a series of text objects, whose *toString()* method finally gives us the title text.

XPath is quite capable of handling more complex tasks: Listing 2 retrieves the serial numbers of all CDs that have an <*artist*> tag enclosing the text "*Foo Fighters*". To retrieve these serial numbers, */result/cd/artists/artist[. = "Foo Fighters"]/../../@serial* first climbs down to the <*artist*> tags, and then checks each of the the <*artist*> tags for the *[. = "Foo Fighters"]* predicate. It uses ., the current node in the path, and checks if its value is identical to the search string, *Foo Fighters*. If so, XPath then climbs back up two levels using *../..* This higher level is where the <*cd*> tag lives; its *serial* parameter is then read, using *@serial*, and returned.

Listing 2 (*xpserial*) shows the entire script, which ultimately calls the returned object's *value()* method to retrieve the text value of the serial number.

XPath offers a compact notation, but if things don't work the first time around, troubleshooting can be a pain. This said, the combination of Perl and XPath makes up for many a disadvantage, as it supports a useful mix of quick XPath hacks, solid program logic, and excellent debugging abilities. Compared to that, using a simple XSLT processor can be a pain.

Figure 3: GroupTags-based XML data-structure with XML::Simple.

XML::Parser

The XML::Parser module implements more of a classical parser. It nibbles its way through the XML document, tag by tag, and calls user-definable callbacks when specific conditions apply. To find the serial numbers of all CDs where the artist tag contains "Foo Fighters," the code needs to keep track of the parser state while the parser traverses down the XML hierarchy.

As *xmlparse* in Listing 3 shows, the *XML::Parser* constructor *new()* expects callbacks for events such as *Start* (when the parser finds an opening XML tag) or *Char* (when the parser finds text between markups.)

When the parser finds an opening tag, such as <*cd serial* = "*001*">, it calls the *start()* function, with a reference to the parser, the tag name, and a key/value attribute list. In our example, the *start()* function is passed the "*cd*" string as its

Listing 2: xpserial

```
01 #!/usr/bin/perl -w
02 use strict;
03 use XML::LibXML;
04
05 my $x = XML::LibXML->new()
06   or die "new failed";
07
08 my $d =
09   $x->parse_file("data.xml")
10   or die "parse failed";
11
12 my $serials = q{
13     /result/cd/artists/
14     artist[.="Foo Fighters"]/
15     ../../@serial
16 };
17
18 for my $serial (
19   $d->findnodes($serials) ) {
20   print $serial->value(),
21     "\n";
22 }
```

Figure 4: twigfilter outputting the modified XML.

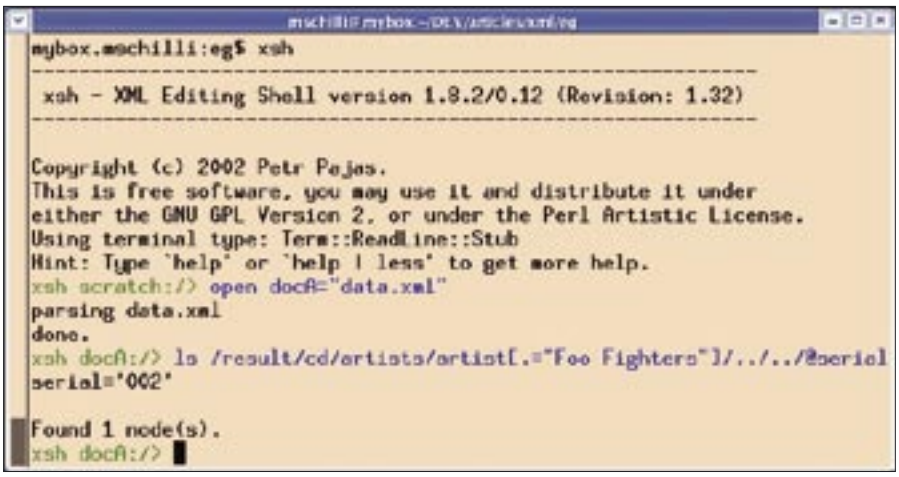

Figure 5: XPath queries in the interactive xsh shell.

second parameter; the third and fourth parameters are "*serial*" and "*001*".

The *text()* callback defined in line 29 ff. receives two parameters from XML::Parser when it finds a text element: a reference to the parser and the string containing the text.

For the parser to know whether a piece of text it has found contains an artist's name (and not some other string), it needs to check if it is currently inside an <*artist*> tag. The only way for the parser to find out is to check if the *$is_artist* global variable has been set to a true value by the *start* callback. The *$serial* global variable uses the same approach to pass the serial number, which *start* finds in the *serial* attribute of the <*cd*> tag. This allows the *print* function inside the *text()* callback to output the serial number of the CD currently being investigated. This approach assumes that every CD actually has a <*serial*> attribute, but you could easily validate this using a DTD, for example.

The XML::Parser module is not normally used directly, but as the base class of user-defined classes. In fact, XML::Simple, which we looked at earlier, may use XML::Parser, depending on your installation environment, and can easily be talked into using the module if you specify *$XML::Simple::PREFERRED_PARSER = "XML::Parser";*.

If you are working on a difficult platform, "XML::SAX::PurePerl", another CPAN parser, could be an option, although it won't be the quickest. It can be installed without a C compiler. Installing XML::Parser can take a while, as it requires you to have a working *expat* parser installation.

To avoid all that work, you could simply misuse another module from CPAN, HTML::Parser, for XML-related chores. Its syntax is only slightly different, and you can stipulate *xml_mode* to switch from loose HTML interpretation to the stricter world of XML.

Listing 3: xmlparse

```
01 #!/usr/bin/perl -w
02 use strict;
03 use XML::Parser;
04
05 my $p = XML::Parser->new();
06 $p->setHandlers(
07   Start => \&start,
08   Char  => \&text,
09 );
10 $p->parsefile("data.xml");
11
12 my $serial;
13 my $is_artist;
14
15 ################################
16 sub start {
17 ################################
18   my ($p, $tag, %attrs) = @_;
19
20   if ( $tag eq "cd" ) {
21     $serial = $attrs{serial};
22   }
23
24   $is_artist =
25     ( $tag eq "artist" );
26 }
27
28 ################################
29 sub text {
30 ################################
31   my ( $p, $text ) = @_;
32
33   if (  $is_artist and
34         $text eq
35           "Foo Fighters" ) {
36     print "$serial\n";
37   }
38 }
```

Wrong Tools, Right Results

If you look at the *htmlparse* in Listing 4, you will note that the *HTML::Parser* constructor expects a slightly different syntax than *XML::Parser*. After specifying which API version you are using, the *start_h* and *text_h* parameters set the callbacks for opening tags and text content outside of the XML markup. The constructor also specifies which parameters the parser should hand to the callbacks: *start()* will be passed the name of the opening tag and an attribute list (as a reference to an array in this case), but the function *text()* is simply handed whatever text has been found.

Do the Twig

XML::Twig by Michel Rodriguez provides amazingly effective mapping of XML to data structures in Perl code. It can handle enormous documents where XML::Simple would simply run and hide; to do so, it parses each document piece by piece rather than attempting to load the whole document into memory.

XML::Twig has so many different XML navigation methods that it can be hard to find the most suitable method for a job in hand. The *twig* script (see Listing

Listing 4: htmlparse

```
01 #!/usr/bin/perl -w
02 use strict;
03 use HTML::Parser;
04
05 my $p = HTML::Parser->new(
06   api_version => 3,
07   start_h     => [
08     \&start, "tagname, attr"
09   ],
10   text_h =>
11     [ \&text, "dtext" ],
12   xml_mode => 1,
13 );
14
15 $p->parse_file("data.xml")
16   or die "Cannot parse";
17
18 my $serial;
19 my $artist;
20
21 ################################
22 sub start {
23 ################################
24   my ( $tag, $attrs ) = @_;
25
26   if ( $tag eq "cd" ) {
27     $serial =
28       $attrs->{serial};
29   }
30
31   $artist =
32     ( $tag eq "artist" );
33 }
34
35 ################################
36 sub text {
37 ################################
38   my ($text) = @_;
39
40   if ($artist and
41       $text eq
42         "Foo Fighters" ) {
43     print "$serial\n";
44   }
45 }
```

5) calls the *XML::Twig::new* constructor with the *Twighandlers* parameter, which maps the XML path */result/cd/artists /artist* to the *artist* handler defined in line 15. Whenever XML::Twig stumbles across an <*artist*> tag while parsing an XML document, it calls the *artist* function with two parameters. The first one is an *XML::Twig* object and the second an XML::Twig::Elt object (apparently Elt is short for element). The latter represents the node in the XML tree to which the <*artist*> tag is attached.

The XML::Twig::Elt object's *text()* method gives us the text between the opening and closing <*artist*> tags. If this happens to be "*Foo Fighters*", lines

22 and 23 navigate to the superordinate *<cd>* tag by calling the *parent()* method twice. The CD object found in this way can be queried for the value of the *serial* attribute using the *att()* method; the value can then be printed.

After an *<artist>* tag has been processed, line 31 calls the XML Twig object's *purge()* method to tell Twig that the tree up to the tag currently being processed is no longer needed, and that this part of the tree can be released. XML::Twig is intelligent enough not to remove the direct parents of the tag it is currently processing, but it will trash any siblings that it has already processed. This kind of memory management does not make much sense for a short piece of XML, but it may make sense if you need to handle an enormous document.

XML::Twig not only has elegant XML navigational features; a script can also rename tags, call methods to dynamically change the tree, or even drop parts to save memory. For example, to convert the cd tag's *serial='xxx'* attribute from *<cd serial="xxx">* ... *</cd>* to the more verbose *<cd><id>xxx</id>* ... *</cd>* notation, and at the same time remove the artist information, the *twigfilter* script (Listing 6) first uses *root()* to retrieve the root object (*<results>*). The *children()* method then returns all the child objects for the root object, that is the *cd* elements. The *att_to_field()* method then transforms the *cd* element's *serial* attributes to *id* field elements.

Then, *first_child()* retrieves the first (and only) *artist* element; and the element's own *delete()* method destructs the node and removes it from the tree.

Listing 5: twig

```
01 #!/usr/bin/perl -w
02 use strict;
03 use XML::Twig;
04
05 my $twig = XML::Twig->new(
06   TwigHandlers => {
07 "/result/cd/artists/artist"
08       => \&artist
09   }
10 );
11
12 $twig->parsefile("data.xml");
13
14 ##############################
15 sub artist {
16 ##############################
17   my ( $t, $artist ) = @_;
18
19   if ( $artist->text() eq
20     "Foo Fighters" ) {
21     my $cd =
22       $artist->parent()
23       ->parent();
24
25     print $cd->att('serial'),
26       "\n";
27   }
28
29 # Release memory of processed
   tree
30 # up to here
31   $t->purge();
32 }
```

Finally, the *set_gi()* method (*gi* stands for generic identifier) renames the cd object of the *<cd>* tag which has just been parsed to *Compact Disc*. Figure 4 shows you the results.

Since we set the *PrettyPrint* parameter for the constructor to "indented," the *print()* method called in line 23 gives us a neatly indented results tree as output.

XML::Twig gives developers the ability to write unbelievably compact programs; it just takes a bit of practice to find the right methods.

XML::XSH

If you prefer an interactive approach, you might like to try the *XML::XSH* module's *xsh*-shell. Calling *xsh* opens a command interpreter that allows you to read XML documents on disk, or retrieve XML documents off the web. You can then shoot arbitrarily complex XPath requests at the document. The results of these requests are immediately displayed in the command line window, allowing you to continuously improve your queries.

Figure 5 on p74 shows the shell user loading the XML document from disk by entering *open docA = "data.xml"*, before going on to type *ls* to issue an XPath query. The result of this XPath query is output as a single serial number: *serial='002'*.

The scripts discussed in this article are just a few hand-picked examples of the kinds of scripts you can build with the huge collection of XML modules available from CPAN. XML::XPath, XML::DOM, XML::Mini, XML::SAX, and XML::Grove are more examples of the infinite options Perl programmers have for digging into XML. ■

Listing 6: twigfilter

```
01 #!/usr/bin/perl -w
02 use strict;
03 use XML::Twig;
04
05 my $twig =
06   XML::Twig->new(
07   PrettyPrint => "indented");
08
09 $twig->parsefile("data.xml")
10   or die "Parse error";
11
12 my $root = $twig->root();
13
14 for my $cd (
15   $root->children('cd') ) {
16   $cd->att_to_field(
17     'serial', 'id' );
18   $cd->first_child('artists')
19     ->delete();
20   $cd->set_gi("CompactDisc");
21 }
22
23 $root->print();
```

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/58/Perl*

[2] XML::Twig tutorial: *http://www.xmltwig.com/xmltwig/tutorial/index.html*