Rocky Enterprise Linux 9.2 Manual Pages on command 'XML::SAX::Intro.3pm'

C:\>man XML::SAX::Intro.3pm

XML::SAX::Intro(3pm) User Contributed Perl Documentation XML::SAX::Intro(3pm)

NAME

XML::SAX::Intro - An Introduction to SAX Parsing with Perl

Introduction

XML::SAX is a new way to work with XML Parsers in Perl. In this article we'll discuss why you should be using SAX, why you should be using XML::SAX, and we'll see some of the finer implementation details. The text below assumes some familiarity with callback, or push based parsing, but if you are unfamiliar with these techniques then a good place to start is Kip Hampton's excellent series of articles on XML.com.

Replacing XML::Parser

The de-facto way of parsing XML under perl is to use Larry Wall and Clark Cooper's XML::Parser. This module is a Perl and XS wrapper around the expat XML parser library by James Clark. It has been a hugely successful project, but suffers from a couple of rather major flaws. Firstly it is a proprietary API, designed before the SAX API was conceived, which means that it is not easily replaceable by other streaming parsers. Secondly it's callbacks are subrefs. This doesn't sound like much of an issue, but unfortunately leads to code like:

```
sub handle_start {
  my ($e, $el, %attrs) = @_;
  if ($el eq 'foo') {
    $e->{inside_foo}++; # BAD! $e is an XML::Parser::Expat object.
```

```
    }
  }
```

As you can see, we're using the $e object to hold our state information, which is a bad idea because we don't own that object - we didn't create it. It's an internal object of XML::Parser, that happens to be a hashref. We could all too easily overwrite XML::Parser internal state variables by using this, or Clark could change it to an array ref (not that he would, because it would break so much code, but he could).

The only way currently with XML::Parser to safely maintain state is to use a closure:

```
my $state = MyState->new();
$parser->setHandlers(Start => sub { handle_start($state, @_) });
```

This closure traps the $state variable, which now gets passed as the first parameter to your callback. Unfortunately very few people use this technique, as it is not documented in the XML::Parser POD files.

Another reason you might not want to use XML::Parser is because you need some feature that it doesn't provide (such as validation), or you might need to use a library that doesn't use expat, due to it not being installed on your system, or due to having a restrictive ISP. Using SAX allows you to work around these restrictions.

Introducing SAX

SAX stands for the Simple API for XML. And simple it really is. Constructing a SAX parser and passing events to handlers is done as simply as:

```
use XML::SAX;
use MySAXHandler;
my $parser = XML::SAX::ParserFactory->parser(
      Handler => MySAXHandler->new
);
$parser->parse_uri("foo.xml");
```

The important concept to grasp here is that SAX uses a factory class called XML::SAX::ParserFactory to create a new parser instance. The reason for this is so that you can support other underlying parser implementations for different feature sets. This is one thing that XML::Parser has always sorely lacked.

In the code above we see the parse_uri method used, but we could have equally well called parse_file, parse_string, or parse(). Please see XML::SAX::Base for what these methods take as parameters, but don't be fooled into believing parse_file takes a filename. No, it takes a file handle, a glob, or a subclass of IO::Handle. Beware.

SAX works very similarly to XML::Parser's default callback method, except it has one major difference: rather than setting individual callbacks, you create a new class in which to receive the callbacks. Each callback is called as a method call on an instance of that handler class. An example will best demonstrate this:

```
 package MySAXHandler;
 use base qw(XML::SAX::Base);
 sub start_document {
   my ($self, $doc) = @_;
   # process document start event
 }
 sub start_element {
   my ($self, $el) = @_;
   # process element start event
 }
```

Now, when we instantiate this as above, and parse some XML with this as the handler, the methods start_document and start_element will be called as method calls, so this would be the equivalent of directly calling:

```
 $object->start_element($el);
```

Notice how this is different to XML::Parser's calling style, which calls:

```
 start_element($e, $name, %attribs);
```

It's the difference between function calling and method calling which allows you to subclass SAX handlers which contributes to SAX being a powerful solution.

As you can see, unlike XML::Parser, we have to define a new package in which to do our processing (there are hacks you can do to make this uneccessary, but I'll leave figuring those out to the experts). The biggest benefit of this is that you maintain your own state variable ($self in the above example) thus freeing you of the concerns listed above. It is also an improvement in maintainability - you can place the code in a separate file if you wish to, and your callback methods are

always called the same thing, rather than having to choose a suitable name for them as you had to with XML::Parser. This is an obvious win.

SAX parsers are also very flexible in how you pass a handler to them. You can use a constructor parameter as we saw above, or we can pass the handler directly in the call to one of the parse methods:

```
$parser->parse(Handler => $handler,
               Source => { SystemId => "foo.xml" });
# or...
$parser->parse_file($fh, Handler => $handler);
```

This flexibility allows for one parser to be used in many different scenarios throughout your script (though one shouldn't feel pressure to use this method, as parser construction is generally not a time consuming process).

Callback Parameters

The only other thing you need to know to understand basic SAX is the structure of the parameters passed to each of the callbacks. In XML::Parser, all parameters are passed as multiple options to the callbacks, so for example the Start callback would be called as my_start($e, $name, %attributes), and the PI callback would be called as my_processing_instruction($e, $target, $data). In SAX, every callback is passed a hash reference, containing entries that define our "node". The key callbacks and the structures they receive are:

start_element

The start_element handler is called whenever a parser sees an opening tag. It is passed an element structure consisting of:

LocalName

: The name of the element minus any namespace prefix it may have come with in the document.

NamespaceURI

: The URI of the namespace associated with this element, or the empty string for none.

Attributes

: A set of attributes as described below.

Name

: The name of the element as it was seen in the document (i.e. including any

prefix associated with it)

Prefix

The prefix used to qualify this element's namespace, or the empty string if none.

The Attributes are a hash reference, keyed by what we have called "James Clark" notation. This means that the attribute name has been expanded to include any associated namespace URI, and put together as {ns}name, where "ns" is the expanded namespace URI of the attribute if and only if the attribute had a prefix, and "name" is the LocalName of the attribute.

The value of each entry in the attributes hash is another hash structure consisting of:

LocalName

The name of the attribute minus any namespace prefix it may have come with in the document.

NamespaceURI

The URI of the namespace associated with this attribute. If the attribute had no prefix, then this consists of just the empty string.

Name

The attribute's name as it appeared in the document, including any namespace prefix.

Prefix

The prefix used to qualify this attribute's namepace, or the empty string if none.

Value

The value of the attribute.

So a full example, as output by Data::Dumper might be:

....

end_element

The end_element handler is called either when a parser sees a closing tag, or after start_element has been called for an empty element (do note however that a parser may if it is so inclined call characters with an empty string when it sees an empty element. There is no simple way in SAX to determine if the parser in fact saw an empty element, a start and end element with no content..

The end_element handler receives exactly the same structure as start_element, minus the Attributes entry. One must note though that it should not be a reference to the same data as start_element receives, so you may change the values in start_element but this will not affect the values later seen by end_element.

characters

The characters callback may be called in several circumstances. The most obvious one is when seeing ordinary character data in the markup. But it is also called for text in a CDATA section, and is also called in other situations. A SAX parser has to make no guarantees whatsoever about how many times it may call characters for a stretch of text in an XML document - it may call once, or it may call once for every character in the text. In order to work around this it is often important for the SAX developer to use a bundling technique, where text is gathered up and processed in one of the other callbacks. This is not always necessary, but it is a worthwhile technique to learn, which we will cover in XML::SAX::Advanced (when I get around to writing it).

The characters handler is called with a very simple structure - a hash reference consisting of just one entry:

Data

The text data that was received.

comment

The comment callback is called for comment text. Unlike with "characters()", the comment callback *must* be invoked just once for an entire comment string. It receives a single simple structure - a hash reference containing just one entry:

Data

The text of the comment.

processing_instruction

The processing instruction handler is called for all processing instructions in the document. Note that these processing instructions may appear before the document root element, or after it, or anywhere where text and elements would normally appear within the document, according to the XML specification.

The handler is passed a structure containing just two entries:

Target

The target of the processing instruction

Data

The text data in the processing instruction. Can be an empty string for a processing instruction that has no data element. For example <?wiggle?> is a perfectly valid processing instruction.

Tip of the iceberg

What we have discussed above is really the tip of the SAX iceberg. And so far it looks like there's not much of interest to SAX beyond what we have seen with XML::Parser. But it does go much further than that, I promise.

People who hate Object Oriented code for the sake of it may be thinking here that creating a new package just to parse something is a waste when they've been parsing things just fine up to now using procedural code. But there's reason to all this madness. And that reason is SAX Filters.

As you saw right at the very start, to let the parser know about our class, we pass it an instance of our class as the Handler to the parser. But now imagine what would happen if our class could also take a Handler option, and simply do some processing and pass on our data further down the line? That in a nutshell is how SAX filters work. It's Unix pipes for the 21st century!

There are two downsides to this. Number 1 - writing SAX filters can be tricky. If you look into the future and read the advanced tutorial I'm writing, you'll see that Handler can come in several shapes and sizes. So making sure your filter does the right thing can be tricky. Secondly, constructing complex filter chains can be difficult, and simple thinking tells us that we only get one pass at our document, when often we'll need more than that.

Luckily though, those downsides have been fixed by the release of two very cool modules. What's even better is that I didn't write either of them!

The first module is XML::SAX::Base. This is a VITAL SAX module that acts as a base class for all SAX parsers and filters. It provides an abstraction away from calling the handler methods, that makes sure your filter or parser does the right thing, and it does it FAST. So, if you ever need to write a SAX filter, which if you're processing XML -> XML, or XML -> HTML, then you probably do, then you need to be writing it as a subclass of XML::SAX::Base. Really - this is advice not to ignore lightly. I will not go into the details of writing a SAX filter here. Kip Hampton, the author of XML::SAX::Base has covered this nicely in his article on XML.com here

<URI>.

To construct SAX pipelines, Barrie Slaymaker, a long time Perl hacker whose modules you will probably have heard of or used, wrote a very clever module called XML::SAX::Machines. This combines some really clever SAX filter-type modules, with a construction toolkit for filters that makes building pipelines easy. But before we see how it makes things easy, first lets see how tricky it looks to build complex SAX filter pipelines.

```
  use XML::SAX::ParserFactory;
  use XML::Filter::Filter1;
  use XML::Filter::Filter2;
  use XML::SAX::Writer;
  my $output_string;
  my $writer = XML::SAX::Writer->new(Output => \$output_string);
  my $filter2 = XML::SAX::Filter2->new(Handler => $writer);
  my $filter1 = XML::SAX::Filter1->new(Handler => $filter2);
  my $parser = XML::SAX::ParserFactory->parser(Handler => $filter1);
  $parser->parse_uri("foo.xml");
```

This is a lot easier with XML::SAX::Machines:

```
  use XML::SAX::Machines qw(Pipeline);
  my $output_string;
  my $parser = Pipeline(
        XML::SAX::Filter1 => XML::SAX::Filter2 => \$output_string
        );
  $parser->parse_uri("foo.xml");
```

One of the main benefits of XML::SAX::Machines is that the pipelines are constructed in natural order, rather than the reverse order we saw with manual pipeline construction. XML::SAX::Machines takes care of all the internals of pipe construction, providing you at the end with just a parser you can use (and you can re-use the same parser as many times as you need to).

Just a final tip. If you ever get stuck and are confused about what is being passed from one SAX filter or parser to the next, then Devel::TraceSAX will come to your rescue. This perl debugger plugin will allow you to dump the SAX stream of events as it goes by. Usage is really very simple just call your perl script that uses SAX

as follows:

```
  $ perl -d:TraceSAX <scriptname>
```

And preferably pipe the output to a pager of some sort, such as more or less. The output is extremely verbose, but should help clear some issues up.

AUTHOR

Matt Sergeant, matt@sergeant.org

Id