NAME

XML::SAX::Base − Base class SAX Drivers and Filters

SYNOPSIS

```
package MyFilter;
use XML::SAX::Base;
@ISA = ('XML::SAX::Base');
```

DESCRIPTION

This module has a very simple task − to be a base class for PerlSAX drivers and filters. It's default behaviour is to pass the input directly to the output unchanged. It can be useful to use this module as a base class so you don't have to, for example, implement the *characters()* callback.

The main advantages that it provides are easy dispatching of events the right way (ie it takes care for you of checking that the handler has implemented that method, or has defined an AUTOLOAD), and the guarantee that filters will pass along events that they aren't implementing to handlers downstream that might nevertheless be interested in them.

WRITING SAX DRIVERS AND FILTERS

The Perl Sax API Reference is at <http://perl−xml.sourceforge.net/perl−sax/>.

Writing SAX Filters is tremendously easy: all you need to do is inherit from this module, and define the events you want to handle. A more detailed explanation can be found at http://www.xml.com/pub/a/2001/10/10/sax−filters.html.

Writing Drivers is equally simple. The one thing you need to pay attention to is **NOT** to call events yourself (this applies to Filters as well). For instance:

```
package MyFilter;
use base qw(XML::SAX::Base);

sub start_element {
  my $self = shift;
  my $data = shift;
  # do something
  $self->{Handler}->start_element($data); # BAD
}
```

The above example works well as precisely that: an example. But it has several faults: 1) it doesn't test to see whether the handler defines start_element. Perhaps it doesn't want to see that event, in which case you shouldn't throw it (otherwise it'll die). 2) it doesn't check ContentHandler and then Handler (ie it doesn't look to see that the user hasn't requested events on a specific handler, and if not on the default one), 3) if it did check all that, not only would the code be cumbersome (see this module's source to get an idea) but it would also probably have to check for a DocumentHandler (in case this were SAX1) and for AUTOLOADs potentially defined in all these packages. As you can tell, that would be fairly painful. Instead of going through that, simply remember to use code similar to the following instead:

```
package MyFilter;
use base qw(XML::SAX::Base);

sub start_element {
  my $self = shift;
  my $data = shift;
  # do something to filter
  $self->SUPER::start_element($data); # GOOD (and easy) !
}
```

This way, once you've done your job you hand the ball back to XML::SAX::Base and it takes care of all those problems for you!

Note that the above example doesn't apply to filters only, drivers will benefit from the exact same feature.

METHODS

A number of methods are defined within this class for the purpose of inheritance. Some probably don't need to be overridden (eg parse_file) but some clearly should be (eg parse). Options for these methods are described in the PerlSAX2 specification available from http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/~checkout~/perl-xml/libxml-perl/doc/sax-2.0.html?rev=HEAD&content-t

- parse

  The parse method is the main entry point to parsing documents. Internally the parse method will detect what type of ''thing'' you are parsing, and call the appropriate method in your implementation class. Here is the mapping table of what is in the Source options (see the Perl SAX 2.0 specification for the meaning of these values):

```
Source Contains           parse() calls
==============            =============
CharacterStream (*)       _parse_characterstream($stream, $options)
ByteStream                _parse_bytestream($stream, $options)
String                    _parse_string($string, $options)
SystemId                  _parse_systemid($string, $options)
```

  However note that these methods may not be sensible if your driver class is not for parsing XML. An example might be a DBI driver that generates XML/SAX from a database table. If that is the case, you likely want to write your own *parse()* method.

  Also note that the Source may contain both a PublicId entry, and an Encoding entry. To get at these, examine `$options->{Source}` as passed to your method.

  (*) A CharacterStream is a filehandle that does not need any encoding translation done on it. This is implemented as a regular filehandle and only works under Perl 5.7.2 or higher using PerlIO. To get a single character, or number of characters from it, use the perl core *read()* function. To get a single byte from it (or number of bytes), you can use *sysread()*. The encoding of the stream should be in the Encoding entry for the Source.

- parse_file, parse_uri, parse_string

  These are all convenience variations on *parse()*, and in fact simply set up the options before calling it. You probably don't need to override these.

- get_options

  This is a convenience method to get options in SAX2 style, or more generically either as hashes or as hashrefs (it returns a hashref). You will probably want to use this method in your own implementations of *parse()* and of *new()*.

- get_feature, set_feature

  These simply get and set features, and throw the appropriate exceptions defined in the specification if need be.

  If your subclass defines features not defined in this one, then you should override these methods in such a way that they check for your features first, and then call the base class's methods for features not defined by your class. An example would be:

```
sub get_feature {
    my $self = shift;
    my $feat = shift;
    if (exists $MY_FEATURES{$feat}) {
        # handle the feature in various ways
    }
    else {
        return $self->SUPER::get_feature($feat);
    }
}
```

Currently this part is unimplemented.

- set_handler

  This method takes a handler type (Handler, ContentHandler, etc.) and a handler object as arguments, and changes the current handler for that handler type, while taking care of resetting the internal state that needs to be reset. This allows one to change a handler during parse without running into problems (changing it on the parser object directly will most likely cause trouble).

- set_document_handler, set_content_handler, set_dtd_handler, set_lexical_handler, set_decl_handler, set_error_handler, set_entity_resolver

  These are just simple wrappers around the former method, and take a handler object as their argument. Internally they simply call set_handler with the correct arguments.

- get_handler

  The inverse of set_handler, this method takes a an optional string containing a handler type (DTDHandler, ContentHandler, etc. 'Handler' is used if no type is passed). It returns a reference to the object that implements that class, or undef if that handler type is not set for the current driver/filter.

- get_document_handler, get_content_handler, get_dtd_handler, get_lexical_handler, get_decl_handler, get_error_handler, get_entity_resolver

  These are just simple wrappers around the *get_handler()* method, and take no arguments. Internally they simply call get_handler with the correct handler type name.

It would be rather useless to describe all the methods that this module implements here. They are all the methods supported in SAX1 and SAX2. In case your memory is a little short, here is a list. The apparent duplicates are there so that both versions of SAX can be supported.

- start_document
- end_document
- start_element
- start_document
- end_document
- start_element
- end_element
- characters
- processing_instruction
- ignorable_whitespace
- set_document_locator
- start_prefix_mapping
- end_prefix_mapping

- skipped_entity
- start_cdata
- end_cdata
- comment
- entity_reference
- notation_decl
- unparsed_entity_decl
- element_decl
- attlist_decl
- doctype_decl
- xml_decl
- entity_decl
- attribute_decl
- internal_entity_decl
- external_entity_decl
- resolve_entity
- start_dtd
- end_dtd
- start_entity
- end_entity
- warning
- error
- fatal_error

TODO

```
- more tests
- conform to the "SAX Filters" and "Java and DOM compatibility"
  sections of the SAX2 document.
```

AUTHOR

Kip Hampton (khampton@totalcinema.com) did most of the work, after porting it from XML::Filter::Base.

Robin Berjon (robin@knowscape.com) pitched in with patches to make it usable as a base for drivers as well as filters, along with other patches.

Matt Sergeant (matt@sergeant.org) wrote the original XML::Filter::Base, and patched a few things here and there, and imported it into the XML::SAX distribution.

SEE ALSO

XML::SAX