Module Version: 2.18

NAME

XML::Simple - Easy API to maintain XML (esp config files)

SYNOPSIS

```
    use XML::Simple;

    my $ref = XMLin([<xml file or string>] [, <options>]);

    my $xml = XMLout($hashref [, <options>]);
```

Or the object oriented way:

```
    require XML::Simple;

    my $xs = XML::Simple->new(options);

    my $ref = $xs->XMLin([<xml file or string>] [, <options>]);

    my $xml = $xs->XMLout($hashref [, <options>]);
```

(or see "SAX SUPPORT" for 'the SAX way').

To catch common errors:

```
    use XML::Simple qw(:strict);
```

(see "STRICT MODE" for more details).

QUICK START

Say you have a script called **foo** and a file of configuration options called **foo.xml** containing this:

```
  <config logdir="/var/log/foo/" debugfile="/tmp/foo.debug">
```

```
    <server name="sahara" osname="solaris" osversion="2.6">
      <address>10.0.0.101</address>
      <address>10.0.1.101</address>
    </server>
    <server name="gobi" osname="irix" osversion="6.5">
      <address>10.0.0.102</address>
    </server>
    <server name="kalahari" osname="linux" osversion="2.0.34">
      <address>10.0.0.103</address>
      <address>10.0.1.103</address>
    </server>
  </config>
```

The following lines of code in **foo**:

```
  use XML::Simple;

  my $config = XMLin();
```

will 'slurp' the configuration options into the hashref $config (because no arguments are passed to `XMLin()` the name and location of the XML file will be inferred from name and location of the script). You can dump out the contents of the hashref using Data::Dumper:

```
  use Data::Dumper;

  print Dumper($config);
```

which will produce something like this (formatting has been adjusted for brevity):

```
  {
      'logdir'        => '/var/log/foo/',
      'debugfile'     => '/tmp/foo.debug',
      'server'        => {
          'sahara'        => {
              'osversion'     => '2.6',
              'osname'        => 'solaris',
              'address'       => [ '10.0.0.101', '10.0.1.101' ]
          },
          'gobi'          => {
              'osversion'     => '6.5',
              'osname'        => 'irix',
              'address'       => '10.0.0.102'
          },
          'kalahari'      => {
              'osversion'     => '2.0.34',
              'osname'        => 'linux',
              'address'       => [ '10.0.0.103', '10.0.1.103' ]
          }
      }
  }
```

Your script could then access the name of the log directory like this:

```
  print $config->{logdir};
```

similarly, the second address on the server 'kalahari' could be referenced as:

```
  print $config->{server}->{kalahari}->{address}->[1];
```

What could be simpler? (Rhetorical).

For simple requirements, that's really all there is to it. If you want to store your XML in a different directory or file, or pass it in as a string or even pass it in via some derivative of an IO::Handle, you'll need to check out "OPTIONS". If you want to turn off or tweak the array folding feature (that neat little transformation that produced $config->{server}) you'll find options for that as well.

If you want to generate XML (for example to write a modified version of $config back out as XML), check out `XMLout()`.

If your needs are not so simple, this may not be the module for you. In that case, you might want to read "WHERE TO FROM HERE?".

DESCRIPTION

The XML::Simple module provides a simple API layer on top of an underlying XML parsing module (either XML::Parser or one of the SAX2 parser modules). Two functions are exported: `XMLin()` and `XMLout()`. Note: you can explicity request the lower case versions of the function names: `xml_in()` and `xml_out()`.

The simplest approach is to call these two functions directly, but an optional object oriented interface (see "OPTIONAL OO INTERFACE" below) allows them to be called as methods of an **XML::Simple** object. The object interface can also be used at either end of a SAX pipeline.

XMLin()

Parses XML formatted data and returns a reference to a data structure which contains the same information in a more readily accessible form. (Skip down to "EXAMPLES" below, for more sample code).

`XMLin()` accepts an optional XML specifier followed by zero or more 'name => value' option pairs. The XML specifier can be one of the following:

A filename

If the filename contains no directory components `XMLin()` will look for the file in each directory in the SearchPath (see "OPTIONS" below) or in the current directory if the SearchPath option is not defined. eg:

```
  $ref = XMLin('/etc/params.xml');
```

Note, the filename '-' can be used to parse from STDIN.

undef

If there is no XML specifier, `XMLin()` will check the script directory and each of the SearchPath directories for a file with the same name as the script but with the extension '.xml'. Note: if you wish to specify options, you must specify the value 'undef'. eg:

```
  $ref = XMLin(undef, ForceArray => 1);
```

A string of XML

A string containing XML (recognised by the presence of '<' and '>' characters) will be parsed directly. eg:

```
  $ref = XMLin('<opt username="bob" password="flurp" />');
```

An IO::Handle object

An IO::Handle object will be read to EOF and its contents parsed. eg:

```
  $fh = IO::File->new('/etc/params.xml');
  $ref = XMLin($fh);
```

XMLout()

Takes a data structure (generally a hashref) and returns an XML encoding of that structure. If the resulting XML is parsed using `XMLin()`, it should return a data structure equivalent to the original (see caveats below).

The `XMLout()` function can also be used to output the XML as SAX events see the `Handler` option and "SAX SUPPORT" for more details).

When translating hashes to XML, hash keys which have a leading '-' will be silently skipped. This is the approved method for marking elements of a data structure which should be ignored by `XMLout`. (Note: If these items were not skipped the key names would be emitted as element or attribute names with a leading '-' which would not be valid XML).

Caveats

Some care is required in creating data structures which will be passed to `XMLout()`. Hash keys from the data structure will be encoded as either XML element names or attribute names. Therefore, you should use hash key names which conform to the relatively strict XML naming rules:

Names in XML must begin with a letter. The remaining characters may be letters, digits, hyphens (-), underscores (_) or full stops (.). It is also allowable to include one colon (:) in an element name but this should only be used when working with namespaces (**XML::Simple** can only usefully work with namespaces when teamed with a SAX Parser).

You can use other punctuation characters in hash values (just not in hash keys) however **XML::Simple** does not support dumping binary data.

If you break these rules, the current implementation of `XMLout()` will simply emit non-compliant XML which will be rejected if you try to read it back in. (A later version of **XML::Simple** might take a more proactive approach).

Note also that although you can nest hashes and arrays to arbitrary levels, circular data structures are not supported and will cause `XMLout()` to die.

If you wish to 'round-trip' arbitrary data structures from Perl to XML and back to Perl, then you should probably disable array folding (using the KeyAttr option) both with `XMLout()` and with `XMLin()`. If you still don't get the expected results, you may prefer to use XML::Dumper which is designed for exactly that purpose.

Refer to "WHERE TO FROM HERE?" if `XMLout()` is too simple for your needs.

OPTIONS

XML::Simple supports a number of options (in fact as each release of **XML::Simple** adds more options, the module's claim to the name 'Simple' becomes increasingly tenuous). If you find yourself repeatedly having to specify the same options, you might like to investigate "OPTIONAL OO INTERFACE" below.

If you can't be bothered reading the documentation, refer to "STRICT MODE" to automatically catch common mistakes.

Because there are so many options, it's hard for new users to know which ones are important, so here are the two you really need to know about:

- check out `ForceArray` because you'll almost certainly want to turn it on
- make sure you know what the `KeyAttr` option does and what its default value is because it may surprise you otherwise (note in particular that 'KeyAttr' affects both `XMLin` and `XMLout`)

The option name headings below have a trailing 'comment' - a hash followed by two pieces of metadata:

- Options are marked with '*in*' if they are recognised by `XMLin()` and '*out*' if they are recognised by `XMLout()`.
- Each option is also flagged to indicate whether it is:

```
'important'   - don't use the module until you understand this one
'handy'       - you can skip this on the first time through
'advanced'    - you can skip this on the second time through
'SAX only'    - don't worry about this unless you're using SAX (or
                alternatively if you need this, you also need SAX)
'seldom used' - you'll probably never use this unless you were the
                person that requested the feature
```

The options are listed alphabetically:

Note: option names are no longer case sensitive so you can use the mixed case versions shown here; all lower case as required by versions 2.03 and earlier; or you can add underscores between the words (eg: key_attr).

AttrIndent => 1 # *out - handy*

When you are using `XMLout()`, enable this option to have attributes printed one-per-line with sensible indentation rather than all on one line.

Cache => [cache schemes] # *in - advanced*

Because loading the **XML::Parser** module and parsing an XML file can consume a significant number of CPU cycles, it is often desirable to cache the output of `XMLin()` for later reuse.

When parsing from a named file, **XML::Simple** supports a number of caching schemes. The 'Cache' option may be used to specify one or more schemes (using an anonymous array). Each scheme will be tried in turn in the hope of finding a cached pre-parsed representation of the XML file. If no cached copy is found, the file will be parsed and the first cache scheme in the list will be used to save a copy of the results. The following cache schemes have been implemented:

storable

> Utilises **Storable.pm** to read/write a cache file with the same name as the XML file but with the extension .stor

memshare

> When a file is first parsed, a copy of the resulting data structure is retained in memory in the **XML::Simple** module's namespace. Subsequent calls to parse the same file will return a reference to this structure. This cached version will persist only for the life of the Perl interpreter (which in the case of mod_perl for example, may be some significant time).

Because each caller receives a reference to the same data structure, a change made by one caller will be visible to all. For this reason, the reference returned should be treated as read-only.

memcopy

This scheme works identically to 'memshare' (above) except that each caller receives a reference to a new data structure which is a copy of the cached version. Copying the data structure will add a little processing overhead, therefore this scheme should only be used where the caller intends to modify the data structure (or wishes to protect itself from others who might). This scheme uses **Storable.pm** to perform the copy.

Warning! The memory-based caching schemes compare the timestamp on the file to the time when it was last parsed. If the file is stored on an NFS filesystem (or other network share) and the clock on the file server is not exactly synchronised with the clock where your script is run, updates to the source XML file may appear to be ignored.

ContentKey => 'keyname' # *in+out - seldom used*

When text content is parsed to a hash value, this option let's you specify a name for the hash key to override the default 'content'. So for example:

```
XMLin('<opt one="1">Text</opt>', ContentKey => 'text')
```

will parse to:

```
{ 'one' => 1, 'text' => 'Text' }
```

instead of:

```
{ 'one' => 1, 'content' => 'Text' }
```

`XMLout()` will also honour the value of this option when converting a hashref to XML.

You can also prefix your selected key name with a '-' character to have `XMLin()` try a little harder to eliminate unnecessary 'content' keys after array folding. For example:

```
XMLin(
  '<opt><item name="one">First</item><item name="two">Second</item></opt>',
  KeyAttr => {item => 'name'},
  ForceArray => [ 'item' ],
  ContentKey => '-content'
)
```

will parse to:

```
  {
    'item' => {
      'one' =>  'First'
      'two' =>  'Second'
    }
  }
```

rather than this (without the '-'):

```
  {
    'item' => {
      'one' => { 'content' => 'First' }
      'two' => { 'content' => 'Second' }
    }
  }
```

DataHandler => code_ref *# in - SAX only*

When you use an **XML::Simple** object as a SAX handler, it will return a 'simple tree' data structure in the same format as `XMLin()` would return. If this option is set (to a subroutine reference), then when the tree is built the subroutine will be called and passed two arguments: a reference to the **XML::Simple** object and a reference to the data tree. The return value from the subroutine will be returned to the SAX driver. (See "SAX SUPPORT" for more details).

ForceArray => 1 *# in - important*

This option should be set to '1' to force nested elements to be represented as arrays even when there is only one. Eg, with ForceArray enabled, this XML:

```
    <opt>
      <name>value</name>
    </opt>
```

would parse to this:

```
    {
      'name' => [
                  'value'
                ]
    }
```

instead of this (the default):

```
    {
      'name' => 'value'
    }
```

This option is especially useful if the data structure is likely to be written back out as XML and the default behaviour of rolling single nested elements up into attributes is not desirable.

If you are using the array folding feature, you should almost certainly enable this option. If you do not, single nested elements will not be parsed to arrays and therefore will not be candidates for folding to a hash. (Given that the default value of 'KeyAttr' enables array folding, the default value of this option should probably also have been enabled too - sorry).

ForceArray => [names] *# in - important*

This alternative (and preferred) form of the 'ForceArray' option allows you to specify a list of element names which should always be forced into an array representation, rather than the 'all or nothing' approach above.

It is also possible (since version 2.05) to include compiled regular expressions in the list - any element names which match the pattern will be forced to arrays. If the list contains only a single regex, then it is not necessary to enclose it in an arrayref. Eg:

```
  ForceArray => qr/_list$/
```

ForceContent => 1 *# in - seldom used*

When `XMLin()` parses elements which have text content as well as attributes, the text content must be represented as a hash value rather than a simple scalar. This option allows you to force text content to always parse to a hash value even when there are no attributes. So for example:

```
  XMLin('<opt><x>text1</x><y a="2">text2</y></opt>', ForceContent => 1)
```

will parse to:

```
  {
    'x' => {           'content' => 'text1' },
    'y' => { 'a' => 2, 'content' => 'text2' }
  }
```

instead of:

```
  {
    'x' => 'text1',
    'y' => { 'a' => 2, 'content' => 'text2' }
  }
```

GroupTags => { grouping tag => grouped tag } # *in+out - handy*

You can use this option to eliminate extra levels of indirection in your Perl data structure. For example this XML:

```
   <opt>
    <searchpath>
      <dir>/usr/bin</dir>
      <dir>/usr/local/bin</dir>
      <dir>/usr/X11/bin</dir>
    </searchpath>
  </opt>
```

Would normally be read into a structure like this:

```
   {
     searchpath => {
                     dir => [ '/usr/bin', '/usr/local/bin', '/usr/X11/bin' ]
                   }
   }
```

But when read in with the appropriate value for 'GroupTags':

```
   my $opt = XMLin($xml, GroupTags => { searchpath => 'dir' });
```

It will return this simpler structure:

```
   {
     searchpath => [ '/usr/bin', '/usr/local/bin', '/usr/X11/bin' ]
   }
```

The grouping element (`<searchpath>` in the example) must not contain any attributes or elements other than the grouped element.

You can specify multiple 'grouping element' to 'grouped element' mappings in the same hashref. If this option is combined with `KeyAttr`, the array folding will occur first and then the grouped element names will be eliminated.

`XMLout` will also use the grouptag mappings to re-introduce the tags around the grouped elements. Beware though that this will occur in all places that the 'grouping tag' name occurs - you probably don't want to use the same name for elements as well as attributes.

Handler => object_ref # *out - SAX only*

Use the 'Handler' option to have `XMLout()` generate SAX events rather than returning a string of XML. For more details see "SAX SUPPORT" below.

Note: the current implementation of this option generates a string of XML and uses a SAX parser to translate it into SAX events. The normal encoding rules apply here - your data must be UTF8 encoded unless you specify an alternative encoding via the 'XMLDecl' option; and by the time the data reaches the handler object, it will be in UTF8 form regardless of the encoding you supply. A future implementation of this option may generate the events directly.

KeepRoot => 1 # *in+out - handy*

In its attempt to return a data structure free of superfluous detail and unnecessary levels of indirection, `XMLin()` normally discards the root element name. Setting the 'KeepRoot' option to '1' will cause the root element name to be retained. So after executing this code:

```
  $config = XMLin('<config tempdir="/tmp" />', KeepRoot => 1)
```

You'll be able to reference the tempdir as `$config->{config}->{tempdir}` instead of the default `$config->{tempdir}`.

Similarly, setting the 'KeepRoot' option to '1' will tell `XMLout()` that the data structure already contains a root element name and it is not necessary to add another.

KeyAttr => [list] # *in+out - important*

This option controls the 'array folding' feature which translates nested elements from an array to a hash. It also controls the 'unfolding' of hashes to arrays.

For example, this XML:

```
    <opt>
      <user login="grep" fullname="Gary R Epstein" />
      <user login="stty" fullname="Simon T Tyson" />
    </opt>
```

would, by default, parse to this:

```
    {
      'user' => [
                  {
                    'login' => 'grep',
                    'fullname' => 'Gary R Epstein'
                  },
                  {
                    'login' => 'stty',
                    'fullname' => 'Simon T Tyson'
                  }
                ]
      }
```

If the option 'KeyAttr => "login"' were used to specify that the 'login' attribute is a key, the same XML would parse to:

```
      {
        'user' => {
                    'stty' => {
                                'fullname' => 'Simon T Tyson'
                              },
                    'grep' => {
                                'fullname' => 'Gary R Epstein'
                              }
                  }
      }
```

The key attribute names should be supplied in an arrayref if there is more than one. `XMLin()` will attempt to match attribute names in the order supplied. `XMLout()` will use the first attribute name supplied when 'unfolding' a hash into an array.

Note 1: The default value for 'KeyAttr' is ['name', 'key', 'id']. If you do not want folding on input or unfolding on output you must setting this option to an empty list to disable the feature.

Note 2: If you wish to use this option, you should also enable the `ForceArray` option. Without 'ForceArray', a single nested element will be rolled up into a scalar rather than an array and therefore will not be folded (since only arrays get folded).

KeyAttr => { list } # *in+out - important*

This alternative (and preferred) method of specifiying the key attributes allows more fine grained control over which elements are folded and on which attributes. For example the option 'KeyAttr => { package => 'id' } will cause any package elements to be folded on the 'id' attribute. No other elements which have an 'id' attribute will be folded at all.

Note: `XMLin()` will generate a warning (or a fatal error in "STRICT MODE") if this syntax is used and an element which does not have the specified key attribute is encountered (eg: a 'package' element without an 'id' attribute, to use the example above). Warnings will only be generated if **-w** is in force.

Two further variations are made possible by prefixing a '+' or a '-' character to the attribute name:

The option 'KeyAttr => { user => "+login" }' will cause this XML:

```
<opt>
  <user login="grep" fullname="Gary R Epstein" />
  <user login="stty" fullname="Simon T Tyson" />
</opt>
```

to parse to this data structure:

```
{
  'user' => {
              'stty' => {
                          'fullname' => 'Simon T Tyson',
                          'login'    => 'stty'
                        },
              'grep' => {
                          'fullname' => 'Gary R Epstein',
                          'login'    => 'grep'
                        }
            }
}
```

The '+' indicates that the value of the key attribute should be copied rather than moved to the folded hash key.

A '-' prefix would produce this result:

```
{
  'user' => {
              'stty' => {
                          'fullname' => 'Simon T Tyson',
                          '-login'    => 'stty'
                        },
              'grep' => {
                          'fullname' => 'Gary R Epstein',
                          '-login'    => 'grep'
                        }
            }
}
```

As described earlier, `XMLout` will ignore hash keys starting with a '-'.

NoAttr => 1 *# in+out - handy*

When used with `XMLout()`, the generated XML will contain no attributes. All hash key/values will be represented as nested elements instead.

When used with `XMLin()`, any attributes in the XML will be ignored.

NoEscape => 1 # *out - seldom used*

By default, `XMLout()` will translate the characters '<', '>', '&' and '"' to '&lt;', '&gt;', '&amp;' and '&quot' respectively. Use this option to suppress escaping (presumably because you've already escaped the data in some more sophisticated manner).

NoIndent => 1 # *out - seldom used*

Set this option to 1 to disable `XMLout()`'s default 'pretty printing' mode. With this option enabled, the XML output will all be on one line (unless there are newlines in the data) - this may be easier for downstream processing.

NoSort => 1 # *out - seldom used*

Newer versions of XML::Simple sort elements and attributes alphabetically (*), by default. Enable this option to suppress the sorting - possibly for backwards compatibility.

* Actually, sorting is alphabetical but 'key' attribute or element names (as in 'KeyAttr') sort first. Also, when a hash of hashes is 'unfolded', the elements are sorted alphabetically by the value of the key field.

NormaliseSpace => 0 | 1 | 2 # *in - handy*

This option controls how whitespace in text content is handled. Recognised values for the option are:

- 0 = (default) whitespace is passed through unaltered (except of course for the normalisation of whitespace in attribute values which is mandated by the XML recommendation)
- 1 = whitespace is normalised in any value used as a hash key (normalising means removing leading and trailing whitespace and collapsing sequences of whitespace characters to a single space)
- 2 = whitespace is normalised in all text content

Note: you can spell this option with a 'z' if that is more natural for you.

NSExpand => 1 # *in+out handy - SAX only*

This option controls namespace expansion - the translation of element and attribute names of the form 'prefix:name' to '{uri}name'. For example the element name 'xsl:template' might be expanded to: '{http://www.w3.org/1999/XSL/Transform}template'.

By default, `XMLin()` will return element names and attribute names exactly as they appear in the XML. Setting this option to 1 will cause all element and attribute names to be expanded to include their namespace prefix.

Note: You must be using a SAX parser for this option to work (ie: it does not work with XML::Parser).

This option also controls whether `XMLout()` performs the reverse translation from '{uri}name' back to 'prefix:name'. The default is no translation. If your data contains expanded names, you should set this option to 1 otherwise `XMLout` will emit XML which is not well formed.

Note: You must have the XML::NamespaceSupport module installed if you want `XMLout()` to translate URIs back to prefixes.

NumericEscape => 0 | 1 | 2 # *out - handy*

Use this option to have 'high' (non-ASCII) characters in your Perl data structure converted to numeric entities (eg: &#8364;) in the XML output. Three levels are possible:

0 - default: no numeric escaping (OK if you're writing out UTF8)

1 - only characters above 0xFF are escaped (ie: characters in the 0x80-FF range are not escaped), possibly useful with ISO8859-1 output

2 - all characters above 0x7F are escaped (good for plain ASCII output)

OutputFile => <file specifier> # *out - handy*

The default behaviour of `XMLout()` is to return the XML as a string. If you wish to write the XML to a file, simply supply the filename using the 'OutputFile' option.

This option also accepts an IO handle object - especially useful in Perl 5.8.0 and later for output using an encoding other than UTF-8, eg:

```
  open my $fh, '>:encoding(iso-8859-1)', $path or die "open($path): $!";
  XMLout($ref, OutputFile => $fh);
```

Note, XML::Simple does not require that the object you pass in to the OutputFile option inherits from IO::Handle - it simply assumes the object supports a `print` method.

ParserOpts => [XML::Parser Options] # *in - don't use this*

Note: This option is now officially deprecated. If you find it useful, email the author with an example of what you use it for. Do not use this option to set the ProtocolEncoding, that's just plain wrong - fix the XML.

This option allows you to pass parameters to the constructor of the underlying XML::Parser object (which of course assumes you're not using SAX).

RootName => 'string' # *out - handy*

By default, when `XMLout()` generates XML, the root element will be named 'opt'. This option allows you to specify an alternative name.

Specifying either undef or the empty string for the RootName option will produce XML with no root elements. In most cases the resulting XML fragment will not be 'well formed' and therefore could not be read back in by `XMLin()`. Nevertheless, the option has been found to be useful in certain circumstances.

SearchPath => [list] # *in - handy*

If you pass `XMLin()` a filename, but the filename include no directory component, you can use this option to specify which directories should be searched to locate the file. You might use this option to search first in the user's home directory, then in a global directory such as /etc.

If a filename is provided to `XMLin()` but SearchPath is not defined, the file is assumed to be in the current directory.

If the first parameter to `XMLin()` is undefined, the default SearchPath will contain only the directory in which the script itself is located. Otherwise the default SearchPath will be empty.

SuppressEmpty => 1 | '' | undef # *in+out - handy*

This option controls what `XMLin()` should do with empty elements (no attributes and no content). The default behaviour is to represent them as empty hashes. Setting this option to a true value (eg: 1) will cause empty elements to be skipped altogether. Setting the option to 'undef' or the empty string will cause empty elements to be represented as the undefined value or the empty string respectively. The latter two alternatives are a little easier to test for in your code than a hash with no keys.

The option also controls what `XMLout()` does with undefined values. Setting the option to undef causes undefined values to be output as empty elements (rather than empty attributes), it also suppresses the generation of warnings about undefined values. Setting the option to a true value (eg: 1) causes undefined values to be skipped altogether on output.

ValueAttr => [names] # *in - handy*

Use this option to deal elements which always have a single attribute and no content. Eg:

```
<opt>
  <colour value="red" />
  <size   value="XXL" />
</opt>
```

Setting `ValueAttr => [ 'value' ]` will cause the above XML to parse to:

```
{
  colour => 'red',
  size   => 'XXL'
}
```

instead of this (the default):

```
  {
    colour => { value => 'red' },
    size   => { value => 'XXL' }
  }
```

Note: This form of the ValueAttr option is not compatible with `XMLout()` - since the attribute name is discarded at parse time, the original XML cannot be reconstructed.

ValueAttr => { element => attribute, ... } # *in+out - handy*

This (preferred) form of the ValueAttr option requires you to specify both the element and the attribute names. This is not only safer, it also allows the original XML to be reconstructed by `XMLout()`.

Note: You probably don't want to use this option and the NoAttr option at the same time.

Variables => { name => value } # *in - handy*

This option allows variables in the XML to be expanded when the file is read. (there is no facility for putting the variable names back if you regenerate XML using `XMLout`).

A 'variable' is any text of the form `${name}` which occurs in an attribute value or in the text content of an element. If 'name' matches a key in the supplied hashref, `${name}` will be replaced with the corresponding value from the hashref. If no matching key is found, the variable will not be replaced. Names must match the regex: `[\w.]+` (ie: only 'word' characters and dots are allowed).

VarAttr => 'attr_name' # *in - handy*

In addition to the variables defined using `Variables`, this option allows variables to be defined in the XML. A variable definition consists of an element with an attribute called 'attr_name' (the value of the `VarAttr` option). The value of the attribute will be used as the variable name and the text content of the element will be used as the value. A variable defined in this way will override a variable defined using the `Variables` option. For example:

```
  XMLin( '<opt>
            <dir name="prefix">/usr/local/apache</dir>
            <dir name="exec_prefix">${prefix}</dir>
            <dir name="bindir">${exec_prefix}/bin</dir>
          </opt>',
         VarAttr => 'name', ContentKey => '-content'
        );
```

produces the following data structure:

```
{
  dir => {
           prefix      => '/usr/local/apache',
           exec_prefix => '/usr/local/apache',
           bindir      => '/usr/local/apache/bin',
         }
}
```

XMLDecl => 1 or XMLDecl => 'string' # *out - handy*

If you want the output from `XMLout()` to start with the optional XML declaration, simply set the option to '1'. The default XML declaration is:

```
      <?xml version='1.0' standalone='yes'?>
```

If you want some other string (for example to declare an encoding value), set the value of this option to the complete string you require.

OPTIONAL OO INTERFACE

The procedural interface is both simple and convenient however there are a couple of reasons why you might prefer to use the object oriented (OO) interface:

- to define a set of default values which should be used on all subsequent calls to `XMLin()` or `XMLout()`
- to override methods in **XML::Simple** to provide customised behaviour

The default values for the options described above are unlikely to suit everyone. The OO interface allows you to effectively override **XML::Simple**'s defaults with your preferred values. It works like this:

First create an XML::Simple parser object with your preferred defaults:

```
my $xs = XML::Simple->new(ForceArray => 1, KeepRoot => 1);
```

then call `XMLin()` or `XMLout()` as a method of that object:

```
my $ref = $xs->XMLin($xml);
my $xml = $xs->XMLout($ref);
```

You can also specify options when you make the method calls and these values will be merged with the values specified when the object was created. Values specified in a method call take precedence.

Note: when called as methods, the `XMLin()` and `XMLout()` routines may be called as `xml_in()` or `xml_out()`. The method names are aliased so the only difference is the aesthetics.

Parsing Methods

You can explicitly call one of the following methods rather than rely on the `xml_in()` method automatically determining whether the target to be parsed is a string, a file or a filehandle:

parse_string(text)

> Works exactly like the `xml_in()` method but assumes the first argument is a string of XML (or a reference to a scalar containing a string of XML).

parse_file(filename)

> Works exactly like the `xml_in()` method but assumes the first argument is the name of a file containing XML.

parse_fh(file_handle)

> Works exactly like the `xml_in()` method but assumes the first argument is a filehandle which can be read to get XML.

Hook Methods

You can make your own class which inherits from XML::Simple and overrides certain behaviours. The following methods may provide useful 'hooks' upon which to hang your modified behaviour. You may find other undocumented methods by examining the source, but those may be subject to change in future releases.

handle_options(direction, name => value ...)

> This method will be called when one of the parsing methods or the `XMLout()` method is called. The initial argument will be a string (either 'in' or 'out') and the remaining arguments will be name value pairs.

default_config_file()

> Calculates and returns the name of the file which should be parsed if no filename is passed to `XMLin()` (default: `$0.xml`).

build_simple_tree(filename, string)

> Called from `XMLin()` or any of the parsing methods. Takes either a file name as the first argument or `undef` followed by a 'string' as the second argument. Returns a simple tree data structure. You could override this method to apply your own transformations before the data structure is returned to the caller.

new_hashref()

When the 'simple tree' data structure is being built, this method will be called to create any required anonymous hashrefs.

sorted_keys(name, hashref)

Called when `XMLout()` is translating a hashref to XML. This routine returns a list of hash keys in the order that the corresponding attributes/elements should appear in the output.

escape_value(string)

Called from `XMLout()`, takes a string and returns a copy of the string with XML character escaping rules applied.

numeric_escape(string)

Called from `escape_value()`, to handle non-ASCII characters (depending on the value of the NumericEscape option).

copy_hash(hashref, extra_key => value, ...)

Called from `XMLout()`, when 'unfolding' a hash of hashes into an array of hashes. You might wish to override this method if you're using tied hashes and don't want them to get untied.

Cache Methods

XML::Simple implements three caching schemes ('storable', 'memshare' and 'memcopy'). You can implement a custom caching scheme by implementing two methods - one for reading from the cache and one for writing to it.

For example, you might implement a new 'dbm' scheme that stores cached data structures using the MLDBM module. First, you would add a `cache_read_dbm()` method which accepted a filename for use as a lookup key and returned a data structure on success, or undef on failure. Then, you would implement a `cache_read_dbm()` method which accepted a data structure and a filename.

You would use this caching scheme by specifying the option:

```
Cache => [ 'dbm' ]
```

STRICT MODE

If you import the **XML::Simple** routines like this:

```
use XML::Simple qw(:strict);
```

the following common mistakes will be detected and treated as fatal errors

- Failing to explicitly set the `KeyAttr` option - if you can't be bothered reading about this option, turn it off with: KeyAttr => [ ]
- Failing to explicitly set the `ForceArray` option - if you can't be bothered reading about this option, set it to the safest mode with: ForceArray => 1
- Setting ForceArray to an array, but failing to list all the elements from the KeyAttr hash.
- Data error - KeyAttr is set to say { part => 'partnum' } but the XML contains one or more <part> elements without a 'partnum' attribute (or nested element). Note: if strict mode is not set but -w is, this condition triggers a warning.
- Data error - as above, but non-unique values are present in the key attribute (eg: more than one <part> element with the same partnum). This will also trigger a warning if strict mode is not enabled.
- Data error - as above, but value of key attribute (eg: partnum) is not a scalar string (due to nested elements etc). This will also trigger a warning if strict mode is not enabled.

SAX SUPPORT

From version 1.08_01, **XML::Simple** includes support for SAX (the Simple API for XML) - specifically SAX2.

In a typical SAX application, an XML parser (or SAX 'driver') module generates SAX events (start of element, character data, end of element, etc) as it parses an XML document and a 'handler' module processes the events to extract the required data. This simple model allows for some interesting and powerful possibilities:

- Applications written to the SAX API can extract data from huge XML documents without the memory overheads of a DOM or tree API.
- The SAX API allows for plug and play interchange of parser modules without having to change your code to fit a new module's API. A number of SAX parsers are available with capabilities ranging from extreme portability to blazing performance.
- A SAX 'filter' module can implement both a handler interface for receiving data and a generator interface for passing modified data on to a downstream handler. Filters can be chained together in 'pipelines'.
- One filter module might split a data stream to direct data to two or more downstream handlers.
- Generating SAX events is not the exclusive preserve of XML parsing modules. For example, a module might extract data from a relational database using DBI and pass it on to a SAX pipeline for filtering and formatting.

XML::Simple can operate at either end of a SAX pipeline. For example, you can take a data structure in the form of a hashref and pass it into a SAX pipeline using the 'Handler' option on `XMLout()`:

```
  use XML::Simple;
```

```
use Some::SAX::Filter;
use XML::SAX::Writer;

my $ref = {
             ....   # your data here
          };

my $writer = XML::SAX::Writer->new();
my $filter = Some::SAX::Filter->new(Handler => $writer);
my $simple = XML::Simple->new(Handler => $filter);
$simple->XMLout($ref);
```

You can also put **XML::Simple** at the opposite end of the pipeline to take advantage of the simple 'tree' data structure once the relevant data has been isolated through filtering:

```
use XML::SAX;
use Some::SAX::Filter;
use XML::Simple;

my $simple = XML::Simple->new(ForceArray => 1, KeyAttr => ['partnum']);
my $filter = Some::SAX::Filter->new(Handler => $simple);
my $parser = XML::SAX::ParserFactory->parser(Handler => $filter);

my $ref = $parser->parse_uri('some_huge_file.xml');

print $ref->{part}->{'555-1234'};
```

You can build a filter by using an XML::Simple object as a handler and setting its DataHandler option to point to a routine which takes the resulting tree, modifies it and sends it off as SAX events to a downstream handler:

```
my $writer = XML::SAX::Writer->new();
my $filter = XML::Simple->new(
               DataHandler => sub {
                                my $simple = shift;
                                my $data = shift;

                                # Modify $data here

                                $simple->XMLout($data, Handler => $writer);
                              }
             );
my $parser = XML::SAX::ParserFactory->parser(Handler => $filter);

$parser->parse_uri($filename);
```

Note: In this last example, the 'Handler' option was specified in the call to `XMLout()` but it could also have been specified in the constructor.

ENVIRONMENT

If you don't care which parser module **XML::Simple** uses then skip this section entirely (it looks more complicated than it really is).

XML::Simple will default to using a **SAX** parser if one is available or **XML::Parser** if SAX is not available.

You can dictate which parser module is used by setting either the environment variable 'XML_SIMPLE_PREFERRED_PARSER' or the package variable $XML::Simple::PREFERRED_PARSER to contain the module name. The following rules are used:

- The package variable takes precedence over the environment variable if both are defined. To force **XML::Simple** to ignore the environment settings and use its default rules, you can set the package variable to an empty string.
- If the 'preferred parser' is set to the string 'XML::Parser', then XML::Parser will be used (or `XMLin()` will die if XML::Parser is not installed).
- If the 'preferred parser' is set to some other value, then it is assumed to be the name of a SAX parser module and is passed to XML::SAX::ParserFactory. If XML::SAX is not installed, or the requested parser module is not installed, then `XMLin()` will die.
- If the 'preferred parser' is not defined at all (the normal default state), an attempt will be made to load XML::SAX. If XML::SAX is installed, then a parser module will be selected according to XML::SAX::ParserFactory's normal rules (which typically means the last SAX parser installed).
- if the 'preferred parser' is not defined and **XML::SAX** is not installed, then **XML::Parser** will be used. `XMLin()` will die if XML::Parser is not installed.

Note: The **XML::SAX** distribution includes an XML parser written entirely in Perl. It is very portable but it is not very fast. You should consider installing XML::LibXML or XML::SAX::Expat if they are available for your platform.

ERROR HANDLING

The XML standard is very clear on the issue of non-compliant documents. An error in parsing any single element (for example a missing end tag) must cause the whole document to be rejected. **XML::Simple** will die with an appropriate message if it encounters a parsing error.

If dying is not appropriate for your application, you should arrange to call `XMLin()` in an eval block and look for errors in $@. eg:

```
    my $config = eval { XMLin() };
    PopUpMessage($@) if($@);
```

Note, there is a common misconception that use of **eval** will significantly slow down a script. While that may be true when the code being eval'd is in a string, it is not true of code like the sample above.

EXAMPLES

When `XMLin()` reads the following very simple piece of XML:

```
    <opt username="testuser" password="frodo"></opt>
```

it returns the following data structure:

```
    {
      'username' => 'testuser',
      'password' => 'frodo'
    }
```

The identical result could have been produced with this alternative XML:

```
    <opt username="testuser" password="frodo" />
```

Or this (although see 'ForceArray' option for variations):

```
    <opt>
      <username>testuser</username>
      <password>frodo</password>
    </opt>
```

Repeated nested elements are represented as anonymous arrays:

```
    <opt>
      <person firstname="Joe" lastname="Smith">
        <email>joe@smith.com</email>
        <email>jsmith@yahoo.com</email>
      </person>
      <person firstname="Bob" lastname="Smith">
        <email>bob@smith.com</email>
      </person>
    </opt>

    {
      'person' => [
                    {
                      'email' => [
                                   'joe@smith.com',
                                   'jsmith@yahoo.com'
                                 ],
                      'firstname' => 'Joe',
                      'lastname' => 'Smith'
                    },
                    {
                      'email' => 'bob@smith.com',
                      'firstname' => 'Bob',
                      'lastname' => 'Smith'
                    }
                  ]
    }
```

Nested elements with a recognised key attribute are transformed (folded) from an array into a hash keyed on the value of that attribute (see the `KeyAttr` option):

```
      <opt>
        <person key="jsmith" firstname="Joe" lastname="Smith" />
        <person key="tsmith" firstname="Tom" lastname="Smith" />
        <person key="jbloggs" firstname="Joe" lastname="Bloggs" />
      </opt>

      {
        'person' => {
                      'jbloggs' => {
                                     'firstname' => 'Joe',
                                     'lastname' => 'Bloggs'
                                   },
                      'tsmith' => {
                                    'firstname' => 'Tom',
                                    'lastname' => 'Smith'
                                  },
                      'jsmith' => {
                                    'firstname' => 'Joe',
                                    'lastname' => 'Smith'
                                  }
                    }
      }
```

The <anon> tag can be used to form anonymous arrays:

```
      <opt>
        <head><anon>Col 1</anon><anon>Col 2</anon><anon>Col 3</anon></head>
        <data><anon>R1C1</anon><anon>R1C2</anon><anon>R1C3</anon></data>
        <data><anon>R2C1</anon><anon>R2C2</anon><anon>R2C3</anon></data>
        <data><anon>R3C1</anon><anon>R3C2</anon><anon>R3C3</anon></data>
      </opt>

      {
        'head' => [
                    [ 'Col 1', 'Col 2', 'Col 3' ]
                  ],
        'data' => [
                    [ 'R1C1', 'R1C2', 'R1C3' ],
                    [ 'R2C1', 'R2C2', 'R2C3' ],
                    [ 'R3C1', 'R3C2', 'R3C3' ]
                  ]
      }
```

Anonymous arrays can be nested to arbirtrary levels and as a special case, if the surrounding tags for an XML document contain only an anonymous array the arrayref will be returned directly rather than the usual hashref:

```
    <opt>
      <anon><anon>Col 1</anon><anon>Col 2</anon></anon>
      <anon><anon>R1C1</anon><anon>R1C2</anon></anon>
      <anon><anon>R2C1</anon><anon>R2C2</anon></anon>
    </opt>

    [
      [ 'Col 1', 'Col 2' ],
      [ 'R1C1', 'R1C2' ],
      [ 'R2C1', 'R2C2' ]
    ]
```

Elements which only contain text content will simply be represented as a scalar. Where an element has both attributes and text content, the element will be represented as a hashref with the text content in the 'content' key (see the `ContentKey` option):

```
  <opt>
    <one>first</one>
    <two attr="value">second</two>
  </opt>

  {
    'one' => 'first',
    'two' => { 'attr' => 'value', 'content' => 'second' }
  }
```

Mixed content (elements which contain both text content and nested elements) will be not be represented in a useful way - element order and significant whitespace will be lost. If you need to work with mixed content, then XML::Simple is not the right tool for your job - check out the next section.

WHERE TO FROM HERE?

XML::Simple is able to present a simple API because it makes some assumptions on your behalf. These include:

- You're not interested in text content consisting only of whitespace

- You don't mind that when things get slurped into a hash the order is lost
- You don't want fine-grained control of the formatting of generated XML
- You would never use a hash key that was not a legal XML element name
- You don't need help converting between different encodings

In a serious XML project, you'll probably outgrow these assumptions fairly quickly. This section of the document used to offer some advice on chosing a more powerful option. That advice has now grown into the 'Perl-XML FAQ' document which you can find at: http://perl-xml.sourceforge.net/faq/

The advice in the FAQ boils down to a quick explanation of tree versus event based parsers and then recommends:

For event based parsing, use SAX (do not set out to write any new code for XML::Parser's handler API - it is obselete).

For tree-based parsing, you could choose between the 'Perlish' approach of XML::Twig and more standards based DOM implementations - preferably one with XPath support.

SEE ALSO

XML::Simple requires either XML::Parser or XML::SAX.

To generate documents with namespaces, XML::NamespaceSupport is required.

The optional caching functions require Storable.

Answers to Frequently Asked Questions about XML::Simple are bundled with this distribution as: XML::Simple::FAQ

COPYRIGHT

syntax highlighting: no syntax highlighting ▼