NAME

XML::LibXML::Parser – Parsing XML Data with XML::LibXML

SYNOPSIS

```
use XML::LibXML '1.70';

# Parser constructor

$parser = XML::LibXML->new();
$parser = XML::LibXML->new(option=>value, ...);
$parser = XML::LibXML->new({option=>value, ...});

# Parsing XML

$dom = XML::LibXML->load_xml(
    location => $file_or_url
    # parser options ...
  );
$dom = XML::LibXML->load_xml(
    string => $xml_string
    # parser options ...
  );
$dom = XML::LibXML->load_xml(
    string => (\$xml_string)
    # parser options ...
  );
$dom = XML::LibXML->load_xml({
    IO => $perl_file_handle
    # parser options ...
  );
$dom = $parser->load_xml(...);

# Parsing HTML

$dom = XML::LibXML->load_html(...);
$dom = $parser->load_html(...);

# Parsing well-balanced XML chunks

$fragment = $parser->parse_balanced_chunk( $wbxmlstring, $encoding );

# Processing XInclude

$parser->process_xincludes( $doc );
$parser->processXIncludes( $doc );

# Old-style parser interfaces

$doc = $parser->parse_file( $xmlfilename );
$doc = $parser->parse_fh( $io_fh );
$doc = $parser->parse_string( $xmlstring);
$doc = $parser->parse_html_file( $htmlfile, \%opts );
$doc = $parser->parse_html_fh( $io_fh, \%opts );
$doc = $parser->parse_html_string( $htmlstring, \%opts );
```

```
  # Push parser

  $parser->parse_chunk($string, $terminate);
  $parser->init_push();
  $parser->push(@data);
  $doc = $parser->finish_push( $recover );

  # Set/query parser options

  $parser->option_exists($name);
  $parser->get_option($name);
  $parser->set_option($name,$value);
  $parser->set_options({$name=>$value,...});

  # XML catalogs

  $parser->load_catalog( $catalog_file );
```

PARSING

An XML document is read into a data structure such as a DOM tree by a piece of software, called a parser. XML::LibXML currently provides four different parser interfaces:

- A DOM Pull-Parser
- A DOM Push-Parser
- A SAX Parser
- A DOM based SAX Parser.

Creating a Parser Instance

XML::LibXML provides an OO interface to the libxml2 parser functions. Thus you have to create a parser instance before you can parse any XML data.

new

```
  $parser = XML::LibXML->new();
  $parser = XML::LibXML->new(option=>value, ...);
  $parser = XML::LibXML->new({option=>value, ...});
```

Create a new XML and HTML parser instance. Each parser instance holds default values for various parser options. Optionally, one can pass a hash reference or a list of option => value pairs to set a different default set of options. Unless specified otherwise, the options `load_ext_dtd`, and `expand_entities` are set to 1. See "Parser Options" for a list of libxml2 parser's options.

DOM Parser

One of the common parser interfaces of XML::LibXML is the DOM parser. This parser reads XML data into a DOM like data structure, so each tag can get accessed and transformed.

XML::LibXML's DOM parser is not only capable to parse XML data, but also (strict) HTML files. There are three ways to parse documents − as a string, as a Perl filehandle, or as a filename/URL. The return value from each is a XML::LibXML::Document object, which is a DOM object.

All of the functions listed below will throw an exception if the document is invalid. To prevent this causing your program exiting, wrap the call in an eval{} block

load_xml

```
  $dom = XML::LibXML->load_xml(
      location => $file_or_url
      # parser options ...
    );
  $dom = XML::LibXML->load_xml(
      string => $xml_string
      # parser options ...
    );
  $dom = XML::LibXML->load_xml(
      string => (\$xml_string)
      # parser options ...
    );
  $dom = XML::LibXML->load_xml({
      IO => $perl_file_handle
      # parser options ...
    );
  $dom = $parser->load_xml(...);
```

This function is available since XML::LibXML 1.70. It provides easy to use interface to the XML parser that parses given file (or URL), string, or input stream to a DOM tree. The arguments can be passed in a HASH reference or as name => value pairs. The function can be called as a class method or an object method. In both cases it internally creates a new parser instance passing the specified parser options; if called as an object method, it clones the original parser (preserving its settings) and additionally applies the specified options to the new parser. See the constructor `new` and ''Parser Options'' for more information.

load_html

```
  $dom = XML::LibXML->load_html(...);
  $dom = $parser->load_html(...);
```

This function is available since XML::LibXML 1.70. It has the same usage as `load_xml`, providing interface to the HTML parser. See `load_xml` for more information.

Parsing HTML may cause problems, especially if the ampersand ('&') is used. This is a common problem if HTML code is parsed that contains links to CGI-scripts. Such links cause the parser to throw errors. In such cases libxml2 still parses the entire document as there was no error, but the error causes XML::LibXML to stop the parsing process. However, the document is not lost. Such HTML documents should be parsed using the *recover* flag. By default recovering is deactivated.

The functions described above are implemented to parse well formed documents. In some cases a program gets well balanced XML instead of well formed documents (e.g. an XML fragment from a database). With XML::LibXML it is not required to wrap such fragments in the code, because XML::LibXML is capable even to parse well balanced XML fragments.

parse_balanced_chunk

```
  $fragment = $parser->parse_balanced_chunk( $wbxmlstring, $encoding );
```

This function parses a well balanced XML string into a XML::LibXML::DocumentFragment. The first arguments contains the input string, the optional second argument can be used to specify character encoding of the input (UTF–8 is assumed by default).

parse_xml_chunk

This is the old name of **parse_balanced_chunk()**. Because it may causes confusion with the push parser interface, this function should not be used anymore.

By default XML::LibXML does not process XInclude tags within an XML Document (see options section below). XML::LibXML allows one to post-process a document to expand XInclude tags.

process_xincludes
```
  $parser->process_xincludes( $doc );
```
After a document is parsed into a DOM structure, you may want to expand the documents XInclude tags. This function processes the given document structure and expands all XInclude tags (or throws an error) by using the flags and callbacks of the given parser instance.

Note that the resulting Tree contains some extra nodes (of type XML_XINCLUDE_START and XML_XINCLUDE_END) after successfully processing the document. These nodes indicate where data was included into the original tree. if the document is serialized, these extra nodes will not show up.

Remember: A Document with processed XIncludes differs from the original document after serialization, because the original XInclude tags will not get restored!

If the parser flag "expand_xincludes" is set to 1, you need not to post process the parsed document.

processXIncludes
```
  $parser->processXIncludes( $doc );
```
This is an alias to process_xincludes, but through a JAVA like function name.

parse_file
```
  $doc = $parser->parse_file( $xmlfilename );
```
This function parses an XML document from a file or network; `$xmlfilename` can be either a filename or an URL. Note that for parsing files, this function is the fastest choice, about 6–8 times faster then **parse_fh()**.

parse_fh
```
  $doc = $parser->parse_fh( $io_fh );
```
parse_fh() parses a IOREF or a subclass of IO::Handle.

Because the data comes from an open handle, libxml2's parser does not know about the base URI of the document. To set the base URI one should use **parse_fh()** as follows:

```
  my $doc = $parser->parse_fh( $io_fh, $baseuri );
```
parse_string
```
  $doc = $parser->parse_string( $xmlstring);
```
This function is similar to **parse_fh()**, but it parses an XML document that is available as a single string in memory, or alternatively as a reference to a scalar containing a string. Again, you can pass an optional base URI to the function.

```
  my $doc = $parser->parse_string( $xmlstring, $baseuri );
  my $doc = $parser->parse_string(\$xmlstring, $baseuri);
```
parse_html_file
```
  $doc = $parser->parse_html_file( $htmlfile, \%opts );
```
Similar to **parse_file()** but parses HTML (strict) documents; `$htmlfile` can be filename or URL.

An optional second argument can be used to pass some options to the HTML parser as a HASH reference. See options labeled with HTML in "Parser Options".

parse_html_fh
```
  $doc = $parser->parse_html_fh( $io_fh, \%opts );
```
Similar to **parse_fh()** but parses HTML (strict) streams.

An optional second argument can be used to pass some options to the HTML parser as a HASH reference. See options labeled with HTML in "Parser Options".

Note: encoding option may not work correctly with this function in libxml2 < 2.6.27 if the HTML file declares charset using a META tag.

parse_html_string

```
$doc = $parser->parse_html_string( $htmlstring, \%opts );
```

Similar to **parse_string()** but parses HTML (strict) strings.

An optional second argument can be used to pass some options to the HTML parser as a HASH reference. See options labeled with HTML in "Parser Options".

Push Parser

XML::LibXML provides a push parser interface. Rather than pulling the data from a given source the push parser waits for the data to be pushed into it.

This allows one to parse large documents without waiting for the parser to finish. The interface is especially useful if a program needs to pre-process the incoming pieces of XML (e.g. to detect document boundaries).

While XML::LibXML parse_*() functions force the data to be a well-formed XML, the push parser will take any arbitrary string that contains some XML data. The only requirement is that all the pushed strings are together a well formed document. With the push parser interface a program can interrupt the parsing process as required, where the parse_*() functions give not enough flexibility.

Different to the pull parser implemented in **parse_fh()** or **parse_file()**, the push parser is not able to find out about the documents end itself. Thus the calling program needs to indicate explicitly when the parsing is done.

In XML::LibXML this is done by a single function:

parse_chunk

```
$parser->parse_chunk($string, $terminate);
```

parse_chunk() tries to parse a given chunk of data, which isn't necessarily well balanced data. The function takes two parameters: The chunk of data as a string and optional a termination flag. If the termination flag is set to a true value (e.g. 1), the parsing will be stopped and the resulting document will be returned as the following example describes:

```
my $parser = XML::LibXML->new;
for my $string ( "<", "foo", ' bar="hello world"', "/>") {
     $parser->parse_chunk( $string );
}
my $doc = $parser->parse_chunk("", 1); # terminate the parsing
```

Internally XML::LibXML provides three functions that control the push parser process:

init_push

```
$parser->init_push();
```

Initializes the push parser.

push

```
$parser->push(@data);
```

This function pushes the data stored inside the array to libxml2's parser. Each entry in `@data` must be a normal scalar! This method can be called repeatedly.

finish_push

```
$doc = $parser->finish_push( $recover );
```

This function returns the result of the parsing process. If this function is called without a parameter it will complain about non well-formed documents. If `$restore` is 1, the push parser can be used to restore broken or non well formed (XML) documents as the following example shows:

```
  eval {
      $parser->push( "<foo>", "bar" );
      $doc = $parser->finish_push();    # will report broken XML
  };
  if ( $@ ) {
     # ...
  }
```

This can be annoying if the closing tag is missed by accident. The following code will restore the document:

```
  eval {
      $parser->push( "<foo>", "bar" );
      $doc = $parser->finish_push(1);   # will return the data parsed
                                        # unless an error happened
  };

  print $doc->toString(); # returns "<foo>bar</foo>"
```

Of course **finish_push()** will return nothing if there was no data pushed to the parser before.

Pull Parser (Reader)

XML::LibXML also provides a pull-parser interface similar to the XmlReader interface in .NET. This interface is almost streaming, and is usually faster and simpler to use than SAX. See XML::LibXML::Reader.

Direct SAX Parser

XML::LibXML provides a direct SAX parser in the XML::LibXML::SAX module.

DOM based SAX Parser

XML::LibXML also provides a DOM based SAX parser. The SAX parser is defined in the module XML::LibXML::SAX::Parser. As it is not a stream based parser, it parses documents into a DOM and traverses the DOM tree instead.

The API of this parser is exactly the same as any other Perl SAX2 parser. See XML::SAX::Intro for details.

Aside from the regular parsing methods, you can access the DOM tree traverser directly, using the **generate()** method:

```
  my $doc = build_yourself_a_document();
  my $saxparser = $XML::LibXML::SAX::Parser->new( ... );
  $parser->generate( $doc );
```

This is useful for serializing DOM trees, for example that you might have done prior processing on, or that you have as a result of XSLT processing.

WARNING

This is NOT a streaming SAX parser. As I said above, this parser reads the entire document into a DOM and serialises it. Some people couldn't read that in the paragraph above so I've added this warning. If you want a streaming SAX parser look at the XML::LibXML::SAX man page

SERIALIZATION

XML::LibXML provides some functions to serialize nodes and documents. The serialization functions are described on the XML::LibXML::Node manpage or the XML::LibXML::Document manpage. XML::LibXML checks three global flags that alter the serialization process:

- skipXMLDeclaration
- skipDTD
- setTagCompression

of that three functions only setTagCompression is available for all serialization functions.

Because XML::LibXML does these flags not itself, one has to define them locally as the following example shows:

```
local $XML::LibXML::skipXMLDeclaration = 1;
local $XML::LibXML::skipDTD = 1;
local $XML::LibXML::setTagCompression = 1;
```

If skipXMLDeclaration is defined and not '0', the XML declaration is omitted during serialization.

If skipDTD is defined and not '0', an existing DTD would not be serialized with the document.

If setTagCompression is defined and not '0' empty tags are displayed as open and closing tags rather than the shortcut. For example the empty tag *foo* will be rendered as *<foo></foo>* rather than *<foo/>*.

PARSER OPTIONS

Handling of libxml2 parser options has been unified and improved in XML::LibXML 1.70. You can now set default options for a particular parser instance by passing them to the constructor as `XML::LibXML->new({name=>value, ...})` or `XML::LibXML->new(name=>value,...)`. The options can be queried and changed using the following methods (pre−1.70 interfaces such as `$parser->load_ext_dtd(0)` also exist, see below):

option_exists
```
$parser->option_exists($name);
```

Returns 1 if the current XML::LibXML version supports the option `$name`, otherwise returns 0 (note that this does not necessarily mean that the option is supported by the underlying libxml2 library).

get_option
```
$parser->get_option($name);
```

Returns the current value of the parser option `$name`.

set_option
```
$parser->set_option($name,$value);
```

Sets option `$name` to value `$value`.

set_options
```
$parser->set_options({$name=>$value,...});
```

Sets multiple parsing options at once.

IMPORTANT NOTE: This documentation reflects the parser flags available in libxml2 2.7.3. Some options have no effect if an older version of libxml2 is used.

Each of the flags listed below is labeled

/parser/
if it can be used with a `XML::LibXML` parser object (i.e. passed to `XML::LibXML->new`, `XML::LibXML->set_option`, etc.)

/html/
if it can be used passed to the `parse_html_*` methods

/reader/
if it can be used with the `XML::LibXML::Reader`.

Unless specified otherwise, the default for boolean valued options is 0 (false).

The available options are:

URI
/parser, html, reader/

In case of parsing strings or file handles, XML::LibXML doesn't know about the base uri of the document. To make relative references such as XIncludes work, one has to set a base URI, that is then used for the parsed document.

line_numbers
/parser, html, reader/

If this option is activated, libxml2 will store the line number of each element node in the parsed document. The line number can be obtained using the `line_number()` method of the `XML::LibXML::Node` class (for non-element nodes this may report the line number of the containing element). The line numbers are also used for reporting positions of validation errors.

IMPORTANT: Due to limitations in the libxml2 library line numbers greater than 65535 will be returned as 65535. Unfortunately, this is a long and sad story, please see <http://bugzilla.gnome.org/show_bug.cgi?id=325533> for more details.

encoding
/html/

character encoding of the input

recover
/parser, html, reader/

recover from errors; possible values are 0, 1, and 2

A true value turns on recovery mode which allows one to parse broken XML or HTML data. The recovery mode allows the parser to return the successfully parsed portion of the input document. This is useful for almost well-formed documents, where for example a closing tag is missing somewhere. Still, XML::LibXML will only parse until the first fatal (non-recoverable) error occurs, reporting recoverable parsing errors as warnings. To suppress even these warnings, use recover=>2.

Note that validation is switched off automatically in recovery mode.

expand_entities
/parser, reader/

substitute entities; possible values are 0 and 1; default is 1

Note that although this flag disables entity substitution, it does not prevent the parser from loading external entities; when substitution of an external entity is disabled, the entity will be represented in the document tree by an XML_ENTITY_REF_NODE node whose subtree will be the content obtained by parsing the external resource; Although this nesting is visible from the DOM it is transparent to XPath data model, so it is possible to match nodes in an unexpanded entity by the same XPath expression as if the entity were expanded. See also ext_ent_handler.

ext_ent_handler
/parser/

Provide a custom external entity handler to be used when expand_entities is set to 1. Possible value is a subroutine reference.

This feature does not work properly in libxml2 < 2.6.27!

The subroutine provided is called whenever the parser needs to retrieve the content of an external entity. It is called with two arguments: the system ID (URI) and the public ID. The value returned by the subroutine is parsed as the content of the entity.

This method can be used to completely disable entity loading, e.g. to prevent exploits of the type described at (<http://searchsecuritychannel.techtarget.com/generic/0,295582,sid97_gci1304703,00.html>), where a service is tricked to expose its private data by letting it parse a remote file (RSS feed) that contains an entity reference to a local file (e.g. `/etc/fstab`).

A more granular solution to this problem, however, is provided by custom URL resolvers, as in

```
my $c = XML::LibXML::InputCallback->new();
sub match {   # accept file:/ URIs except for XML catalogs in /etc/xml/
  my ($uri) = @_;
  return ($uri=~m{^file:/}
          and $uri !~ m{^file:///etc/xml/})
         ? 1 : 0;
}
$c->register_callbacks([ \&match, sub{}, sub{}, sub{} ]);
$parser->input_callbacks($c);
```

load_ext_dtd
/parser, reader/

load the external DTD subset while parsing; possible values are 0 and 1. Unless specified, XML::LibXML sets this option to 1.

This flag is also required for DTD Validation, to provide complete attribute, and to expand entities, regardless if the document has an internal subset. Thus switching off external DTD loading, will disable entity expansion, validation, and complete attributes on internal subsets as well.

complete_attributes
/parser, reader/

create default DTD attributes; possible values are 0 and 1

validation
/parser, reader/

validate with the DTD; possible values are 0 and 1

suppress_errors
/parser, html, reader/

suppress error reports; possible values are 0 and 1

suppress_warnings
/parser, html, reader/

suppress warning reports; possible values are 0 and 1

pedantic_parser
/parser, html, reader/

pedantic error reporting; possible values are 0 and 1

no_blanks
/parser, html, reader/

remove blank nodes; possible values are 0 and 1

no_defdtd
/html/

do not add a default DOCTYPE; possible values are 0 and 1

the default is (0) to add a DTD when the input html lacks one

expand_xinclude or xinclude
/parser, reader/

Implement XInclude substitution; possible values are 0 and 1

Expands XInclude tags immediately while parsing the document. Note that the parser will use the URI resolvers installed via `XML::LibXML::InputCallback` to parse the included document (if any).

no_xinclude_nodes
/parser, reader/

do not generate XINCLUDE START/END nodes; possible values are 0 and 1

no_network
/parser, html, reader/

Forbid network access; possible values are 0 and 1

If set to true, all attempts to fetch non-local resources (such as DTD or external entities) will fail (unless custom callbacks are defined).

It may be necessary to use the flag `recover` for processing documents requiring such resources while networking is off.

clean_namespaces
/parser, reader/

remove redundant namespaces declarations during parsing; possible values are 0 and 1.

no_cdata
/parser, html, reader/

merge CDATA as text nodes; possible values are 0 and 1

no_basefix
/parser, reader/

not fixup XINCLUDE xml#base URIS; possible values are 0 and 1

huge
/parser, html, reader/

relax any hardcoded limit from the parser; possible values are 0 and 1. Unless specified, XML::LibXML sets this option to 0.

Note: the default value for this option was changed to protect against denial of service through entity expansion attacks. Before enabling the option ensure you have taken alternative measures to protect your application against this type of attack.

gdome
/parser/

THIS OPTION IS EXPERIMENTAL!

Although quite powerful, XML::LibXML's DOM implementation is incomplete with respect to the DOM level 2 or level 3 specifications. XML::GDOME is based on libxml2 as well, and provides a rather complete DOM implementation by wrapping libgdome. This flag allows you to make use of XML::LibXML's full parser options and XML::GDOME's DOM implementation at the same time.

To make use of this function, one has to install libgdome and configure XML::LibXML to use this library. For this you need to rebuild XML::LibXML!

Note: this feature was not seriously tested in recent XML::LibXML releases.

For compatibility with XML::LibXML versions prior to 1.70, the following methods are also supported for querying and setting the corresponding parser options (if called without arguments, the methods return the current value of the corresponding parser options; with an argument sets the option to a given value):

```
  $parser->validation();
  $parser->recover();
  $parser->pedantic_parser();
  $parser->line_numbers();
  $parser->load_ext_dtd();
  $parser->complete_attributes();
  $parser->expand_xinclude();
  $parser->gdome_dom();
  $parser->clean_namespaces();
  $parser->no_network();
```

The following obsolete methods trigger parser options in some special way:

recover_silently

```
    $parser->recover_silently(1);
```

If called without an argument, returns true if the current value of the `recover` parser option is 2 and returns false otherwise. With a true argument sets the `recover` parser option to 2; with a false argument sets the `recover` parser option to 0.

expand_entities

```
    $parser->expand_entities(0);
```

Get/set the `expand_entities` option. If called with a true argument, also turns the `load_ext_dtd` option to 1.

keep_blanks

```
    $parser->keep_blanks(0);
```

This is actually the opposite of the `no_blanks` parser option. If used without an argument retrieves negated value of `no_blanks`. If used with an argument sets `no_blanks` to the opposite value.

base_uri

```
    $parser->base_uri( $your_base_uri );
```

Get/set the `URI` option.

XML CATALOGS

`libxml2` supports XML catalogs. Catalogs are used to map remote resources to their local copies. Using catalogs can speed up parsing processes if many external resources from remote addresses are loaded into the parsed documents (such as DTDs or XIncludes).

Note that libxml2 has a global pool of loaded catalogs, so if you apply the method `load_catalog` to one parser instance, all parser instances will start using the catalog (in addition to other previously loaded catalogs).

Note also that catalogs are not used when a custom external entity handler is specified. At the current state it is not possible to make use of both types of resolving systems at the same time.

load_catalog

```
    $parser->load_catalog( $catalog_file );
```

Loads the XML catalog file `$catalog_file`.

```
    # Global external entity loader (similar to ext_ent_handler option
    # but this works really globally, also in XML::LibXSLT include etc..)

    XML::LibXML::externalEntityLoader(\&my_loader);
```

ERROR REPORTING

XML::LibXML throws exceptions during parsing, validation or XPath processing (and some other occasions). These errors can be caught by using *eval* blocks. The error is stored in *$@*. There are two implementations: the old one throws $@ which is just a message string, in the new one $@ is an object from the class XML::LibXML::Error; this class overrides the operator "" so that when printed, the object flattens to the usual error message.

XML::LibXML throws errors as they occur. This is a very common misunderstanding in the use of XML::LibXML. If the eval is omitted, XML::LibXML will always halt your script by ‘‘croaking’’ (see Carp man page for details).

Also note that an increasing number of functions throw errors if bad data is passed as arguments. If you cannot assure valid data passed to XML::LibXML you should eval these functions.

Note: since version 1.59, **get_last_error()** is no longer available in XML::LibXML for thread-safety reasons.

AUTHORS

Matt Sergeant, Christian Glahn, Petr Pajas

VERSION

2.0134

COPYRIGHT

LICENSE