NAME

XML::LibXML − Perl Binding for libxml2

SYNOPSIS

```
use XML::LibXML;
my $dom = XML::LibXML->load_xml(string => <<'EOT');
<some-xml/>
EOT

$Version_String = XML::LibXML::LIBXML_DOTTED_VERSION;
$Version_ID = XML::LibXML::LIBXML_VERSION;
$DLL_Version = XML::LibXML::LIBXML_RUNTIME_VERSION;
$libxmlnode = XML::LibXML->import_GDOME( $node, $deep );
$gdomenode = XML::LibXML->export_GDOME( $node, $deep );
```

DESCRIPTION

This module is an interface to libxml2, providing XML and HTML parsers with DOM, SAX and XMLReader interfaces, a large subset of DOM Layer 3 interface and a XML::XPath−like interface to XPath API of libxml2. The module is split into several packages which are not described in this section; unless stated otherwise, you only need to `use XML::LibXML;` in your programs.

For further information, please check the following documentation:

XML::LibXML::Parser
: Parsing XML files with XML::LibXML

XML::LibXML::DOM
: XML::LibXML Document Object Model (DOM) Implementation

XML::LibXML::SAX
: XML::LibXML direct SAX parser

XML::LibXML::Reader
: Reading XML with a pull-parser

XML::LibXML::Dtd
: XML::LibXML frontend for DTD validation

XML::LibXML::RelaxNG
: XML::LibXML frontend for RelaxNG schema validation

XML::LibXML::Schema
: XML::LibXML frontend for W3C Schema schema validation

XML::LibXML::XPathContext
: API for evaluating XPath expressions with enhanced support for the evaluation context

XML::LibXML::InputCallback
: Implementing custom URI Resolver and input callbacks

XML::LibXML::Common
: Common functions for XML::LibXML related Classes

The nodes in the Document Object Model (DOM) are represented by the following classes (most of which ‘‘inherit’’ from XML::LibXML::Node):

XML::LibXML::Document
: XML::LibXML class for DOM document nodes

XML::LibXML::Node
: Abstract base class for XML::LibXML DOM nodes

XML::LibXML::Element
: XML::LibXML class for DOM element nodes

XML::LibXML::Text
: XML::LibXML class for DOM text nodes

XML::LibXML::Comment
: XML::LibXML class for comment DOM nodes

XML::LibXML::CDATASection
: XML::LibXML class for DOM CDATA sections

XML::LibXML::Attr
: XML::LibXML DOM attribute class

XML::LibXML::DocumentFragment
: XML::LibXML's DOM L2 Document Fragment implementation

XML::LibXML::Namespace
: XML::LibXML DOM namespace nodes

XML::LibXML::PI
: XML::LibXML DOM processing instruction nodes

ENCODINGS SUPPORT IN XML::LIBXML

Recall that since version 5.6.1, Perl distinguishes between character strings (internally encoded in UTF-8) and so called binary data and, accordingly, applies either character or byte semantics to them. A scalar representing a character string is distinguished from a byte string by special flag (UTF8). Please refer to *perlunicode* for details.

XML::LibXML's API is designed to deal with many encodings of XML documents completely transparently, so that the application using XML::LibXML can be completely ignorant about the encoding of the XML documents it works with. On the other hand, functions like `XML::LibXML::Document->setEncoding` give the user control over the document encoding.

To ensure the aforementioned transparency and uniformity, most functions of XML::LibXML that work with in-memory trees accept and return data as character strings (i.e. UTF-8 encoded with the UTF8 flag on) regardless of the original document encoding; however, the functions related to I/O operations (i.e. parsing and saving) operate with binary data (in the original document encoding) obeying the encoding declaration of the XML documents.

Below we summarize basic rules and principles regarding encoding:

1. Do NOT apply any encoding-related PerlIO layers (`:utf8` or `:encoding(...)`) to file handles that are an input for the parses or an output for a serializer of (full) XML documents. This is because the conversion of the data to/from the internal character representation is provided by libxml2 itself which must be able to enforce the encoding specified by the `<?xml version="1.0" encoding="..."?>` declaration. Here is an example to follow:

```
use XML::LibXML;
# load
open my $fh, '<', 'file.xml';
binmode $fh; # drop all PerlIO layers possibly created by a use open pragma
$doc = XML::LibXML->load_xml(IO => $fh);

# save
open my $out, '>', 'out.xml';
binmode $out; # as above
$doc->toFH($out);
# or
print {$out} $doc->toString();
```

2. All functions working with DOM accept and return character strings (UTF-8 encoded with UTF8 flag on). E.g.

```
my $doc = XML::LibXML::Document->new('1.0',$some_encoding);
my $element = $doc->createElement($name);
$element->appendText($text);
$xml_fragment = $element->toString(); # returns a character string
$xml_document = $doc->toString(); # returns a byte string
```

where `$some_encoding` is the document encoding that will be used when saving the document, and `$name` and `$text` contain character strings (UTF-8 encoded with UTF8 flag on). Note that the method `toString` returns XML as a character string if applied to other node than the Document node and a byte string containing the appropriate

```
<?xml version="1.0" encoding="..."?>
```

declaration if applied to a XML::LibXML::Document.

3. DOM methods also accept binary strings in the original encoding of the document to which the node belongs (UTF-8 is assumed if the node is not attached to any document). Exploiting this feature is NOT RECOMMENDED since it is considered bad practice.

```
my $doc = XML::LibXML::Document->new('1.0','iso-8859-2');
my $text = $doc->createTextNode($some_latin2_encoded_byte_string);
# WORKS, BUT NOT RECOMMENDED!
```

NOTE: libxml2 support for many encodings is based on the iconv library. The actual list of supported encodings may vary from platform to platform. To test if your platform works correctly with your language encoding, build a simple document in the particular encoding and try to parse it with XML::LibXML to see if the parser produces any errors. Occasional crashes were reported on rare platforms that ship with a broken version of iconv.

THREAD SUPPORT

XML::LibXML since 1.67 partially supports Perl threads in Perl >= 5.8.8. XML::LibXML can be used with threads in two ways:

By default, all XML::LibXML classes use CLONE_SKIP class method to prevent Perl from copying XML::LibXML::* objects when a new thread is spawn. In this mode, all XML::LibXML::* objects are thread specific. This is the safest way to work with XML::LibXML in threads.

Alternatively, one may use

```
use threads;
use XML::LibXML qw(:threads_shared);
```

to indicate, that all XML::LibXML node and parser objects should be shared between the main thread and any thread spawn from there. For example, in

```
my $doc = XML::LibXML->load_xml(location => $filename);
my $thr = threads->new(sub{
  # code working with $doc
  1;
});
$thr->join;
```

the variable `$doc` refers to the exact same XML::LibXML::Document in the spawned thread as in the main thread.

Without using mutex locks, parallel threads may read the same document (i.e. any node that belongs to the document), parse files, and modify different documents.

However, if there is a chance that some of the threads will attempt to modify a document (or even create new nodes based on that document, e.g. with `$doc->createElement`) that other threads may be reading at the same time, the user is responsible for creating a mutex lock and using it in *both* in the thread that modifies and the thread that reads:

```
my $doc = XML::LibXML->load_xml(location => $filename);
my $mutex : shared;
my $thr = threads->new(sub{
   lock $mutex;
   my $el = $doc->createElement('foo');
   # ...
  1;
});
{
  lock $mutex;
  my $root = $doc->documentElement;
  say $root->name;
}
$thr->join;
```

Note that libxml2 uses dictionaries to store short strings and these dictionaries are kept on a document node. Without mutex locks, it could happen in the previous example that the thread modifies the dictionary while other threads attempt to read from it, which could easily lead to a crash.

VERSION INFORMATION

Sometimes it is useful to figure out, for which version XML::LibXML was compiled for. In most cases this is for debugging or to check if a given installation meets all functionality for the package. The functions XML::LibXML::LIBXML_DOTTED_VERSION and XML::LibXML::LIBXML_VERSION provide this version information. Both functions simply pass through the values of the similar named macros of libxml2. Similarly, XML::LibXML::LIBXML_RUNTIME_VERSION returns the version of the (usually dynamically) linked libxml2.

XML::LibXML::LIBXML_DOTTED_VERSION

```
$Version_String = XML::LibXML::LIBXML_DOTTED_VERSION;
```

Returns the version string of the libxml2 version XML::LibXML was compiled for. This will be "2.6.2" for "libxml2 2.6.2".

XML::LibXML::LIBXML_VERSION

```
$Version_ID = XML::LibXML::LIBXML_VERSION;
```

Returns the version id of the libxml2 version XML::LibXML was compiled for. This will be "20602" for "libxml2 2.6.2". Don't mix this version id with `$XML::LibXML::VERSION`. The latter contains the version of XML::LibXML itself while the first contains the version of libxml2 XML::LibXML was compiled for.

XML::LibXML::LIBXML_RUNTIME_VERSION

```
$DLL_Version = XML::LibXML::LIBXML_RUNTIME_VERSION;
```

Returns a version string of the libxml2 which is (usually dynamically) linked by XML::LibXML. This will be "20602" for libxml2 released as "2.6.2" and something like "20602-CVS2032" for a CVS build of libxml2.

XML::LibXML issues a warning if the version of libxml2 dynamically linked to it is less than the version of libxml2 which it was compiled against.

EXPORTS

By default the module exports all constants and functions listed in the :all tag, described below.

EXPORT TAGS

`:all`

Includes the tags `:libxml`, `:encoding`, and `:ns` described below.

`:libxml`

Exports integer constants for DOM node types.

```
XML_ELEMENT_NODE            => 1
XML_ATTRIBUTE_NODE          => 2
XML_TEXT_NODE               => 3
XML_CDATA_SECTION_NODE      => 4
XML_ENTITY_REF_NODE         => 5
XML_ENTITY_NODE             => 6
XML_PI_NODE                 => 7
XML_COMMENT_NODE            => 8
XML_DOCUMENT_NODE           => 9
XML_DOCUMENT_TYPE_NODE      => 10
XML_DOCUMENT_FRAG_NODE      => 11
XML_NOTATION_NODE           => 12
XML_HTML_DOCUMENT_NODE      => 13
XML_DTD_NODE                => 14
XML_ELEMENT_DECL            => 15
XML_ATTRIBUTE_DECL          => 16
XML_ENTITY_DECL             => 17
XML_NAMESPACE_DECL          => 18
XML_XINCLUDE_START          => 19
XML_XINCLUDE_END            => 20
```

`:encoding`

Exports two encoding conversion functions from XML::LibXML::Common.

```
encodeToUTF8()
decodeFromUTF8()
```

`:ns`

Exports two convenience constants: the implicit namespace of the reserved `xml:` prefix, and the implicit namespace for the reserved `xmlns:` prefix.

```
XML_XML_NS    => 'http://www.w3.org/XML/1998/namespace'
XML_XMLNS_NS  => 'http://www.w3.org/2000/xmlns/'
```

RELATED MODULES

The modules described in this section are not part of the XML::LibXML package itself. As they support some additional features, they are mentioned here.

XML::LibXSLT

XSLT 1.0 Processor using libxslt and XML::LibXML

XML::LibXML::Iterator

XML::LibXML Implementation of the DOM Traversal Specification

XML::CompactTree::XS

Uses XML::LibXML::Reader to very efficiently to parse XML document or element into native Perl data structures, which are less flexible but significantly faster to process then DOM.

XML::LIBXML AND XML::GDOME

Note: *THE FUNCTIONS DESCRIBED HERE ARE STILL EXPERIMENTAL*

Although both modules make use of libxml2's XML capabilities, the DOM implementation of both modules are not compatible. But still it is possible to exchange nodes from one DOM to the other. The concept of this exchange is pretty similar to the function **cloneNode()**: The particular node is copied on the low-level to the opposite DOM implementation.

Since the DOM implementations cannot coexist within one document, one is forced to copy each node that should be used. Because you are always keeping two nodes this may cause quite an impact on a machines memory usage.

XML::LibXML provides two functions to export or import GDOME nodes: **import_GDOME()** and

export_GDOME(). Both function have two parameters: the node and a flag for recursive import. The flag works as in **cloneNode()**.

The two functions allow one to export and import XML::GDOME nodes explicitly, however, XML::LibXML also allows the transparent import of XML::GDOME nodes in functions such as **appendChild()**, **insertAfter()** and so on. While native nodes are automatically adopted in most functions XML::GDOME nodes are always cloned in advance. Thus if the original node is modified after the operation, the node in the XML::LibXML document will not have this information.

import_GDOME

```
$libxmlnode = XML::LibXML->import_GDOME( $node, $deep );
```

This clones an XML::GDOME node to an XML::LibXML node explicitly.

export_GDOME

```
$gdomenode = XML::LibXML->export_GDOME( $node, $deep );
```

Allows one to clone an XML::LibXML node into an XML::GDOME node.

CONTACTS

For bug reports, please use the CPAN request tracker on http://rt.cpan.org/NoAuth/Bugs.html?Dist=XML-LibXML

For suggestions etc., and other issues related to XML::LibXML you may use the perl XML mailing list (`perl-xml@listserv.ActiveState.com`), where most XML-related Perl modules are discussed. In case of problems you should check the archives of that list first. Many problems are already discussed there. You can find the list's archives and subscription options at <http://aspn.activestate.com/ASPN/Mail/Browse/Threaded/perl-xml>.

AUTHORS

Matt Sergeant, Christian Glahn, Petr Pajas

VERSION

2.0134

COPYRIGHT

LICENSE