The LuaXML library

Paul Chakravarti Michal Hoftich

Version v0.1n
2020-12-20

Contents

1 Introduction 2

2 The `DOM_Object` library 2
- 2.1 Void elements 3
- 2.2 Node selection methods 3
  - 2.2.1 The `DOM_Object:get_path` method 3
  - 2.2.2 The `DOM_Object:query_selector` method 4
  - 2.2.3 Supported CSS selectors 4
- 2.3 Element traversing 5
  - 2.3.1 The `DOM_Object:traverse_elements` method 5
- 2.4 DOM modifications 6

3 The `CssQuery` library 6
- 3.1 Example usage 7

4 The `luaxml-transform` library 8

5 The API documentation 8
- 5.1 luaxml-domobject 8
  - 5.1.1 Class: Class DOM_Object 8
  - 5.1.2 Class: function 12
- 5.2 luaxml-cssquery 12
  - 5.2.1 Class: Class CssQuery 13
  - 5.2.2 Class: function 14

6 Low-level functions usage 14
- 6.1 The simpleTreeHandler 14
- 6.2 The domHandler 16

I Original `LuaXML` documentation by Paul Chakravarti 18

7 Overview 18

8 Features **18**

9 Limitations **18**

10 API **19**

11 Options **20**

12 Usage **20**

13 Handlers **20**
- 13.1 Overview 20
- 13.2 Features 21
- 13.3 API 21
  - 13.3.1 printHandler 21
  - 13.3.2 domHandler 21
  - 13.3.3 simpleTreeHandler 21
- 13.4 Options 22
- 13.5 Usage 22

14 History **22**

15 License **22**

1 Introduction

`LuaXML` is pure lua library for processing and serializing of the `xml` files. The base code code has been written by Paul Chakravarti, with minor changes which brings Lua 5.3 or HTML 5 support. On top of that, new modules for accessing the `xml` files using `DOM` like methods or `CSS` selectors[1] have been added.

The documentation is divided to three parts – first part deals with the `DOM` library, second part describes the low-level libraries and the third part is original documentation by Paul Chakravarti.

2 The `DOM_Object` library

This library can process a `xml` sources using `DOM` like functions. To load it, you need to require `luaxml-domobject.lua` file. The `parse` function provided by the library creates `DOM_Object` object, which provides several methods for processing the `xml` tree.

```
local dom = require "luaxml-domobject"
local document = [[
<html>
<head><title>sample</title></head>
```

---

[1] Thanks to Leaf Corcoran for `CSS selector` parsing code.

```
<body>
<h1>test</h1>
<p>hello</p>
</body>
</html>
  ]]

-- dom.parse returns the DOM_Object
local obj = dom.parse(document)
-- it is possible to call methods on the object
local root_node = obj:root_node()
for _, x in ipairs(root_node:get_children()) do
  print(x:get_element_name())
end
```

The details about available methods can be found in the API docs, section 5.1. The above code will load a `xml` document, it will get the ROOT element and print all it's children element names. The `DOM_Object:get_children` function returns Lua table, so it is possible to loop over it using standard table functions.

```
html
```

2.1 Void elements

The `DOM_Object.parse` function tries to support the HTML void elements, such as `<img>` or `<hr>`. They cannot have closing tags, a parse error occurs when the closing tags are used.

It is possible to define a different set of void elements using the second parameter for `DOM_Object.parse`:

```
obj = dom.parse(document, {custom_void = true})
```

An empty table will disable all void elements. This setting is recommended for common `xml` documents.

2.2 Node selection methods

There are some other methods for element retrieving.

2.2.1 The `DOM_Object:get_path` method

If you want to print text content of all child elements of the body element, you can use `DOM_Object:get_path`:

```
local path = obj:get_path("html body")
for _, el in ipairs(path[1]:get_children()) do
  print(el:get_text())
end
```

The `DOM_Object:get_path` function always return array with all elements which match the requested path, even it there is only one such element. In this case, it is possible to use standard Lua table indexing to get the first and only one matched element and get it's children using `DOM_Object:get_children` method. It the children node is an element, it's text content is printed using `DOM_Object:get_text`.

```
test
hello
```

2.2.2 The `DOM_Object:query_selector` method

This method uses `CSS selector` syntax to select elements, similarly to JavaScript *jQuery* library.

```
for _, el in ipairs(obj:query_selector("h1,p")) do
  print(el:get_text())
end
```

```
test
hello
```

It supports also `XML` namespaces, using `namespace|element` syntax.

2.2.3 Supported CSS selectors

The `query_selector` method supports following CSS selectors:

Universal selector – `*` – select any element.

Type selector – `elementname` – Selects all elements that have the given node name.

Class selector – `.classname` – Selects all elements that have the given class attribute.

ID selector – `#idname` – Selects an element based on the value of its id attribute.

Attribute selector – `[attrname='value']` – Selects all elements that have the given attribute. It can have the following variants: `[attrname]` – elements that contain given attribute, `[attr|=value]` – attribute text is exactly the value, with optional hyphen at the end, `[attr~=value]` – attribute name of attr whose value is a whitespace-separated list of words, one of which is exactly value, `[attr^=value]` – attribute text starts with value, `[attr$=value]` – attribute text ends with value.

Grouping selector – `,` – This is a grouping method, it selects all the matching nodes.

It is also possible to combine selectors using *combinators* to make more specific searches. Supported combinators:

Descendant combinator – `A B` – match all B elements that are inside A elements.

Child combinator – `A > B` – match B elements that are nested directly inside a A element.

General sibling combinator – `A ~ B` – the second element follows the first (though not necessarily immediately), and both share the same parent.

Adjacent sibling combinator – `A + B` – the second element directly follows the first, and both share the same parent.

LuaXML also supports some CSS pseudo-classes. A pseudo-class is a keyword added to a selector that specifies a special state of the selected element. The following are supported:

:first-child – matches an element that is the first of its siblings.

:first-of-type – matches an element that is the first of its siblings, and also matches a certain type selector.

:last-child – matches an element that is the last of its siblings.

:last-of-type – matches an element that is the last of its siblings, and also matches a certain type selector.

:nth-child – matches elements based on their position in a group of siblings. It can be used like this: `li:nth-child(2)`.

2.3 Element traversing

2.3.1 The `DOM_Object:traverse_elements` method

It may be useful to traverse over all elements and apply a function on all of them.

```
obj:traverse_elements(function(node)
  print(node:get_text())
end)
```

```
 sample  test hello
 sample  test hello
sample
sample
 test hello
test
hello
```

The `get_text` method gets text from all children elements, so the first line shows all text contained in the `<html>` element, the second one in `<head>` element and so on.

2.4 DOM modifications

It is possible to add new elements, text nodes, or to remove them.

```
local headers = obj:query_selector("h1")
for _, header in ipairs(headers) do
  header:remove_node()
end
-- query selector returns array, we must retrieve the first element
-- to get the actual body element
local body = obj:query_selector("body")[1]
local paragraph = body:create_element("p", {})
body:add_child_node(paragraph)
paragraph:add_child_node(paragraph:create_text_node("This is a second paragraph"))

for _, el in ipairs(body:get_children()) do
  if el:is_element() then
    print(el:get_element_name().. ": ".. el:get_text())
  end
end
```

In this example, `<h1>` element is being removed from the sample document, and new paragraph is added. Two paragraphs should be shown in the output:

```
p: hello
p: This is a second paragraph
```

3 The `CssQuery` library

This library serves mainly as a support for the `DOM_Object:query_selector` function. It also supports adding information to the DOM tree.

3.1 Example usage

```
local cssobj = require "luaxml-cssquery"
local domobj = require "luaxml-domobject"

local xmltext = [[
<html>
<body>
<h1>Header</h1>
<p>Some text, <i>italics</i></p>
</body>
</html>
]]

local dom = domobj.parse(xmltext)
local css = cssobj()

css:add_selector("h1", function(obj)
  print("header found: "  .. obj:get_text())
end)

css:add_selector("p", function(obj)
  print("paragraph found: " .. obj:get_text())
end)

css:add_selector("i", function(obj)
  print("found italics: " .. obj:get_text())
end)

dom:traverse_elements(function(el)
  -- find selectors that match the current element
  local querylist = css:match_querylist(el)
  -- add templates to the element
  css:apply_querylist(el,querylist)
end)
```

```
header found: Header
paragraph found: Some text, italics
found italics: italics
```

More complete example may be found in the `examples` directory in the `LuaXML` source code repository[2].

---

[2] `https://github.com/michal-h21/LuaXML/blob/master/examples/xmltotex.lua`

4 The `luaxml-transform` library

This library is still a bit experimental. It enables XML transformation based on CSS selector templates. It is better to use XSLT in general, but it may succeed for simpler tasks[3].

5 The API documentation

5.1 luaxml-domobject

DOM module for LuaXML

5.1.1 Class: Class DOM_Object

DOM_Object:root_node()
Returns root element of the DOM_Object
Parameters:
Return:
DOM_Object

DOM_Object:get_node_type(el)
Get current node type
Parameters:
`el`: [optional] node to get the type of

DOM_Object:is_element(el)
Test if the current node is an element.
Parameters:
`el`: [optional] element to test
Return:
boolean

DOM_Object:is_text(el)
Test if current node is text
Parameters:
`el`: [optional] element to test
Return:
boolean

DOM_Object:get_element_name(el)
Return name of the current element
Parameters:
`el`: [optional] element to test

---

[3] See this example: `https://tex.stackexchange.com/a/574004/2891`

Return:
string

DOM_Object:get_attribute(name)
Get value of an attribute
Parameters:
`name`: Attribute name
Return:
string

DOM_Object:set_attribute(name, value)
Set value of an attribute
Parameters:
`name`:
`value`: Value to be set
Return:
boolean

DOM_Object:serialize(current)
Serialize the current node back to XML
Parameters:
`current`: [optional] element to be serialized
Return:
string

DOM_Object:get_text(current)
Get text content from the node and all of it's children
Parameters:
`current`: [optional] element which should be converted to text
Return:
string

DOM_Object:get_path(path, current)
Retrieve elements from the given path.
Parameters:
`path`:
`current`: [optional] element which should be traversed. Default element is the root element of the DOM_Object
Return:
table of elements which match the path

DOM_Object:query_selector(selector)
Select elements chidlren using CSS selector syntax
Parameters:
`selector`: String using the CSS selector syntax
Return:

table with elements matching the selector.

DOM_Object:get_children(el)
Get table with children of the current element
Parameters:
`el`: [optional] element to be selected
Return:
table with children of the selected element

DOM_Object:get_parent(el)
Get the parent element
Parameters:
`el`: [optional] element to be selected
Return:
DOM_Object parent element

DOM_Object:traverse_elements(fn, current)
Execute function on the current element and all it's children elements.
Parameters:
`fn`: function which will be executed on the current element and all it's children
`current`: [optional] element to be selected
Return:
nothing

DOM_Object:traverse_node_list(nodelist, fn)
Execute function on list of elements returned by DOM_Object:get_path()
Parameters:
`nodelist`:
`fn`: function to be executed

DOM_Object:replace_node(new)
Replace the current node with new one
Parameters:
`new`: element which should replace the current element
Return:
boolean, message

DOM_Object:add_child_node(child, position)
Add child node to the current node
Parameters:
`child`: element to be inserted as a current node child
`position`: [optional] position at which should the node be inserted

DOM_Object:copy_node(element)
Create copy of the current node
Parameters:

`element`: [optional] element to be copied
Return:
DOM_Object element

DOM_Object:create_element(name, attributes, parent)
Create a new element
Parameters:
`name`: New tag name
`attributes`: Table with attributes
`parent`: [optional] element which should be saved as the element's parent
Return:
DOM_Object element

DOM_Object:create_text_node(text, parent)
Create new text node
Parameters:
`text`: string
`parent`: [optional] element which should be saved as the element's parent
Return:
DOM_Object text object

DOM_Object:remove_node(element)
Delete current node
Parameters:
`element`: [optional] element to be removed

DOM_Object:find_element_pos(el)
Find the element position in the current node list
Parameters:
`el`: [optional] element which should be looked up
Return:
integer position of the current element in the element table

DOM_Object:get_siblings(el)
Get node list which current node is part of
Parameters:
`el`: [optional] element for which the sibling element list should be retrieved
Return:
table with elements

DOM_Object:get_sibling_node(change)
Get sibling node of the current node
Parameters:
`change`: Distance from the current node
Return:
DOM_Object node

DOM_Object:get_next_node(el)
Get next node
Parameters:
`el`: [optional] node to be used
Return:
DOM_Object node

DOM_Object:get_prev_node(el)
Get previous node
Parameters:
`el`: [optional] node to be used
Return:
DOM_Object node

5.1.2 Class: function

serialize_dom(parser, current, level, output)
It serializes the DOM object back to the XML.
Parameters:
`parser`: DOM object
`current`: Element which should be serialized
`level`:
`output`:
Return:
table Table with XML strings. It can be concenated using table.concat() function to get XML string corresponding to the DOM_Object.

parse(xmltext, voidElements)
XML parsing function Parse the XML text and create the DOM object.
Parameters:
`xmltext`:
`voidElements`: hash table with void elements
Return:
DOM_Object

5.2 luaxml-cssquery

CSS query module for LuaXML

5.2.1 Class: Class CssQuery

CssQuery:calculate_specificity(query)
Calculate CSS specificity of the query
Parameters:
`query`: table created by CssQuery:prepare_selector() function
Return:
integer speficity value

CssQuery:match_querylist(domobj, querylist)
Test prepared querylist
Parameters:
`domobj`: DOM element to test
`querylist`: [optional] List of queries to test
Return:
table with CSS queries, which match the selected DOM element

CssQuery:get_selector_path(domobj, selectorlist)
Get elements that match the selector
Parameters:
`domobj`: DOM_Object
`selectorlist`: prepare_selector
Return:
table with DOM_Object elements

CssQuery:prepare_selector(selector)
Parse CSS selector to a query table.
Parameters:
`selector`: string CSS selector query
Return:
table querylist

CssQuery:add_selector(selector, func, params)
Add selector to CSS object list of selectors, func is called when the selector matches a DOM object params is table which will be passed to the func
Parameters:
`selector`: CSS selector string
`func`: function which will be executed on matched elements
`params`: table with parameters for the function
Return:
integer number of elements in the prepared selector

CssQuery:sort_querylist(querylist)
Sort selectors according to their specificity It is called automatically when the selector is added
Parameters:

`querylist`: [optional] querylist table
Return:
querylist table

CssQuery:apply_querylist(domobj, querylist)
It tests list of queries agaings a DOM element and executes the coresponding function that is saved for the matched query.
Parameters:
`domobj`: DOM element
`querylist`: querylist table
Return:
nothing

5.2.2 Class: function

cssquery()
CssQuery constructor
Parameters:
Return:
CssQuery object

6 Low-level functions usage

The original `LuaXML` library provides some low-level functions for `XML` handling. First of all, we need to load the libraries:

```
xml = require('luaxml-mod-xml')
handler = require('luaxml-mod-handler')
```

The `luaxml-mod-xml` file contains the xml parser and also the serializer. In `luaxml-mod-handler`, various handlers for dealing with xml data are defined. Handlers transforms the `xml` file to data structures which can be handled from the Lua code. More information about handlers can be found in the original documentation, section 13.

6.1 The simpleTreeHandler

```
sample = [[
<a>
  <d>hello</d>
  <b>world.</b>
  <b at="Hi">another</b>
</a>]]
treehandler = handler.simpleTreeHandler()
x = xml.xmlParser(treehandler)
```

```
x:parse(sample)
```

You have to create handler object, using `handler.simpleTreeHandler()` and xml parser object using `xml.xmlParser(handler object)`. `simpleTreehandler` creates simple table hierarchy, with top root node in `treehandler.root`

```
-- pretty printing function
function printable(tb, level)
  level = level or 1
  local spaces = string.rep(' ', level*2)
  for k,v in pairs(tb) do
    if type(v) ~= "table" then
      print(spaces .. k..'='..v)
    else
      print(spaces .. k)
      level = level + 1
      printable(v, level)
    end
  end
end

-- print table
printable(treehandler.root)
-- print xml serialization of table
print(xml.serialize(treehandler.root))
-- direct access to the element
print(treehandler.root["a"]["b"][1])
```

This code produces the following output:

```
output:
  a
    d=hello
    b
      1=world.
      2
        1=another
        _attr
          at=Hi
<?xml version="1.0" encoding="UTF-8"?>
<a>
  <d>hello</d>
    <b>world.</b>
    <b at="Hi">
      another
    </b>
</a>

world.
```

First part is pretty-printed dump of Lua table structure contained in the handler, the second part is `xml` serialized from that table and the last part demonstrates direct access to particular elements.

Note that `simpleTreeHandler` creates tables that can be easily accessed using standard lua functions, but if the xml document is of mixed-content type[4]:

```
<a>hello
  <b>world</b>
</a>
```

then it produces wrong results. It is useful mostly for data `xml` files, not for text formats like `xhtml`.

6.2 The domHandler

For complex xml documents, it is best to use the `domHandler`, which creates object which contains all information from the `xml` document.

```
-- file dom-sample.lua
-- next line enables scripts called with texlua to use luatex libraries
--kpse.set_program_name("luatex")
function traverseDom(current,level)
  local level = level or 0
  local spaces = string.rep(" ",level)
  local root= current or current.root
  local name = root._name or "unnamed"
  local xtype = root._type or "untyped"
  local attributes = root._attr  or {}
  if xtype == "TEXT" then
    print(spaces .."TEXT : " .. root._text)
  else
    print(spaces .. xtype .. " : " .. name)
  end
  for k, v in pairs(attributes) do
    print(spaces .. "  ".. k.."="..v)
  end
  local children = root._children or {}
  for _, child in ipairs(children) do
    traverseDom(child, level + 1)
  end
end

local xml = require('luaxml-mod-xml')
local handler = require('luaxml-mod-handler')
local x = '<p>hello <a href="http://world.com/">world</a>, how are you?</p>'
local domHandler = handler.domHandler()
local parser = xml.xmlParser(domHandler)
parser:parse(x)
traverseDom(domHandler.root)
```

---

[4] This means that element may contain both children elements and text.

The ROOT element is stored in `domHandler.root` table, it's child nodes are stored in `_children` tables. Node type is saved in `_type` field, if the node type is `ELEMENT`, then `_name` field contains element name, `_attr` table contains element attributes. `TEXT` node contains text content in `_text` field.

The previous code produces following output in the terminal:

```
ROOT : unnamed
 ELEMENT : p
  TEXT : hello
  ELEMENT : a
    href=http://world.com/
   TEXT : world
  TEXT : , how are you?
```

Part I

Original `LuaXML` documentation by Paul Chakravarti

This document was created automatically from the original source code comments using Pandoc[5]

7 Overview

This module provides a non-validating XML stream parser in Lua.

8 Features

- Tokenises well-formed XML (relatively robustly)
- Flexible handler based event api (see below)
- Parses all XML Infoset elements - ie.
    - Tags
    - Text
    - Comments
    - CDATA
    - XML Decl
    - Processing Instructions
    - DOCTYPE declarations
- Provides limited well-formedness checking (checks for basic syntax & balanced tags only)
- Flexible whitespace handling (selectable)
- Entity Handling (selectable)

9 Limitations

- Non-validating
- No charset handling
- No namespace support
- Shallow well-formedness checking only (fails to detect most semantic errors)

---

[5] `http://johnmacfarlane.net/pandoc/`

10 API

The parser provides a partially object-oriented API with functionality split into tokeniser and hanlder components.

The handler instance is passed to the tokeniser and receives callbacks for each XML element processed (if a suitable handler function is defined). The API is conceptually similar to the SAX API but implemented differently.

The following events are generated by the tokeniser

```
handler:starttag       - Start Tag
handler:endtag         - End Tag
handler:text        - Text
handler:decl        - XML Declaration
handler:pi          - Processing Instruction
handler:comment     - Comment
handler:dtd         - DOCTYPE definition
handler:cdata       - CDATA
```

The function prototype for all the callback functions is

```
callback(val,attrs,start,end)
```

where attrs is a table and val/attrs are overloaded for specific callbacks - ie.

Callback	val	attrs (table)
starttag	name	`{ attributes (name=val).. }`
endtag	name	nil
text	`<text>`	nil
cdata	`<text>`	nil
decl	”xml”	`{ attributes (name=val).. }`
pi	pi name	

```
{ attributes (if present)..
  _text = <PI Text>
}
```

comment	`<text>`	nil
dtd	root element	

```
{ _root = <Root Element>,
  _type = SYSTEM|PUBLIC,
  _name = <name>,
  _uri = <uri>,
  _internal = <internal dtd>
}
```

(starttag & endtag provide the character positions of the start/end of the element)

XML data is passed to the parser instance through the ‘parse’ method (Note: must be passed as single string currently)

11 Options

Parser options are controlled through the ‘self.options’ table. Available options are -

- stripWS

  Strip non-significant whitespace (leading/trailing) and do not generate events for empty text elements

- expandEntities

  Expand entities (standard entities + single char numeric entities only currently - could be extended at runtime if suitable DTD parser added elements to table (see obj._ENTITIES). May also be possible to expand multibyre entities for UTF–8 only

- errorHandler

  Custom error handler function

NOTE: Boolean options must be set to ‘nil’ not ‘0’

12 Usage

Create a handler instance -

```
h = { starttag = function(t,a,s,e) .... end,
      endtag = function(t,a,s,e) .... end,
      text = function(t,a,s,e) .... end,
      cdata = text }
```

(or use predefined handler - see luaxml-mod-handler.lua)

Create parser instance -

```
p = xmlParser(h)
```

Set options -

```
p.options.xxxx = nil
```

Parse XML data -

```
xmlParser:parse("<?xml... ")
```

13 Handlers

13.1 Overview

Standard XML event handler(s) for XML parser module (luaxml-mod-xml.lua)

13.2 Features

```
printHandler        - Generate XML event trace
domHandler          - Generate DOM-like node tree
simpleTreeHandler   - Generate 'simple' node tree
simpleTeXhandler    - SAX like handler with support for CSS selectros
```

13.3 API

Must be called as handler function from xmlParser and implement XML event callbacks (see xmlParser.lua for callback API definition)

13.3.1 printHandler

printHandler prints event trace for debugging

13.3.2 domHandler

domHandler generates a DOM-like node tree structure with a single ROOT node parent - each node is a table comprising fields below.

```
node = { _name = <Element Name>,
        _type = ROOT|ELEMENT|TEXT|COMMENT|PI|DECL|DTD,
        _attr = { Node attributes - see callback API },
        _parent = <Parent Node>
        _children = { List of child nodes - ROOT/NODE only }
      }
```

13.3.3 simpleTreeHandler

simpleTreeHandler is a simplified handler which attempts to generate a more 'natural' table based structure which supports many common XML formats.

The XML tree structure is mapped directly into a recursive table structure with node names as keys and child elements as either a table of values or directly as a string value for text. Where there is only a single child element this is inserted as a named key - if there are multiple elements these are inserted as a vector (in some cases it may be preferable to always insert elements as a vector which can be specified on a per element basis in the options). Attributes are inserted as a child element with a key of '_attr'.

Only Tag/Text & CDATA elements are processed - all others are ignored.

This format has some limitations - primarily

- Mixed-Content behaves unpredictably - the relationship between text elements and embedded tags is lost and multiple levels of mixed content does not work
- If a leaf element has both a text element and attributes then the text must be accessed through a vector (to provide a container for the attribute)

In general however this format is relatively useful.

13.4 Options

```
simpleTreeHandler.options.noReduce = { <tag> = bool,.. }

    - Nodes not to reduce children vector even if only
      one child

domHandler.options.(comment|pi|dtd|decl)Node = bool

    - Include/exclude given node types
```

13.5 Usage

Pased as delegate in xmlParser constructor and called as callback by xmlParser:parse(xml) method.

14 History

This library is fork of LuaXML library originaly created by Paul Chakravarti. Some files not needed for use with luatex were droped from the distribution. Documentation was converted from original comments in the source code.

15 License

This code is freely distributable under the terms of the Lua license (`http://www.lua.org/copyright.html`)