Linux Ubuntu 22.4.5 Manual Pages on command 'XML::LibXML::InputCallback.3pm'

$ man XML::LibXML::InputCallback.3pm

XML::LibXML::InputCallback(3User Contributed Perl DocumentaXML::LibXML::InputCallback(3pm)

NAME

XML::LibXML::InputCallback - XML::LibXML Class for Input Callbacks

SYNOPSIS

use XML::LibXML;

DESCRIPTION

You may get unexpected results if you are trying to load external documents during libxml2 parsing if the location of the resource is not a HTTP, FTP or relative location but a absolute path for example. To get around this limitation, you may add your own input handler to open, read and close particular types of locations or URI classes. Using this input callback handlers, you can handle your own custom URI schemes for example.

The input callbacks are used whenever XML::LibXML has to get something other than externally parsed entities from somewhere. They are implemented using a callback stack on the Perl layer in analogy to libxml2's native callback stack.

The XML::LibXML::InputCallback class transparently registers the input callbacks for the libxml2's parser processes.

How does XML::LibXML::InputCallback work?

The libxml2 library offers a callback implementation as global functions only. To work-around the troubles resulting in having only global callbacks - for example, if the same global callback stack is manipulated by different applications running together in a single Apache Web-server environment -, XML::LibXML::InputCallback

comes with a object-oriented and a function-oriented part.

Using the function-oriented part the global callback stack of libxml2 can be manipulated. Those functions can be used as interface to the callbacks on the C- and XS Layer. At the object-oriented part, operations for working with the "pseudo-localized" callback stack are implemented. Currently, you can register and de-register callbacks on the Perl layer and initialize them on a per parser basis.

Callback Groups

The libxml2 input callbacks come in groups. One group contains a URI matcher (match), a data stream constructor (open), a data stream reader (read), and a data stream destructor (close). The callbacks can be manipulated on a per group basis only.

The Parser Process

The parser process works on an XML data stream, along which, links to other resources can be embedded. This can be links to external DTDs or XIncludes for example. Those resources are identified by URIs. The callback implementation of libxml2 assumes that one callback group can handle a certain amount of URIs and a certain URI scheme. Per default, callback handlers for file://*, file::///*.gz, http://* and ftp://* are registered.

Callback groups in the callback stack are processed from top to bottom, meaning that callback groups registered later will be processed before the earlier registered ones.

While parsing the data stream, the libxml2 parser checks if a registered callback group will handle a URI - if they will not, the URI will be interpreted as file://URI. To handle a URI, the match callback will have to return '1'. If that happens, the handling of the URI will be passed to that callback group. Next, the URI will be passed to the open callback, which should return a reference to the data stream if it successfully opened the file, '0' otherwise. If opening the stream was successful, the read callback will be called repeatedly until it returns an empty string. After the read callback, the close callback will be called to close the stream.

Organisation of callback groups in XML::LibXML::InputCallback

Callback groups are implemented as a stack (Array), each entry holds a reference to an array of the callbacks. For the libxml2 library, the XML::LibXML::InputCallback

callback implementation appears as one single callback group. The Perl implementation however allows one to manage different callback stacks on a per libxml2-parser basis.

Using XML::LibXML::InputCallback

After object instantiation using the parameter-less constructor, you can register callback groups.

```
my $input_callbacks = XML::LibXML::InputCallback->new();
$input_callbacks->register_callbacks([ $match_cb1, $open_cb1,
                                       $read_cb1, $close_cb1 ] );
$input_callbacks->register_callbacks([ $match_cb2, $open_cb2,
                                       $read_cb2, $close_cb2 ] );
$input_callbacks->register_callbacks( [ $match_cb3, $open_cb3,
                                        $read_cb3, $close_cb3 ] );
$parser->input_callbacks( $input_callbacks );
$parser->parse_file( $some_xml_file );
```

What about the old callback system prior to XML::LibXML::InputCallback?

In XML::LibXML versions prior to 1.59 - i.e. without the XML::LibXML::InputCallback module - you could define your callbacks either using globally or locally. You still can do that using XML::LibXML::InputCallback, and in addition to that you can define the callbacks on a per parser basis!

If you use the old callback interface through global callbacks, XML::LibXML::InputCallback will treat them with a lower priority as the ones registered using the new interface. The global callbacks will not override the callback groups registered using the new interface. Local callbacks are attached to a specific parser instance, therefore they are treated with highest priority. If the match callback of the callback group registered as local variable is identical to one of the callback groups registered using the new interface, that callback group will be replaced.

Users of the old callback implementation whose open callback returned a plain string, will have to adapt their code to return a reference to that string after upgrading to version >= 1.59. The new callback system can only deal with the open callback returning a reference!

Global Variables

$_CUR_CB

Stores the current callback and can be used as shortcut to access the callback stack.

@_GLOBAL_CALLBACKS

Stores all callback groups for the current parser process.

@_CB_STACK

Stores the currently used callback group. Used to prevent parser errors when dealing with nested XML data.

Global Callbacks

_callback_match

Implements the interface for the match callback at C-level and for the selection of the callback group from the callbacks defined at the Perl-level.

_callback_open

Forwards the open callback from libxml2 to the corresponding callback function at the Perl-level.

_callback_read

Forwards the read request to the corresponding callback function at the Perl-level and returns the result to libxml2.

_callback_close

Forwards the close callback from libxml2 to the corresponding callback function at the Perl-level..

Class methods

new()

A simple constructor.

register_callbacks([$match_cb, $open_cb, $read_cb, $close_cb])

The four callbacks have to be given as array reference in the above order match, open, read, close!

unregister_callbacks([$match_cb, $open_cb, $read_cb, $close_cb])

With no arguments given, "unregister_callbacks()" will delete the last registered callback group from the stack. If four callbacks are passed as array reference, the callback group to unregister will be identified by the match callback and deleted from the callback stack. Note that if several identical

match callbacks are defined in different callback groups, ALL of them will be deleted from the stack.

init_callbacks($parser)

Initializes the callback system for the provided parser before starting a parsing process.

cleanup_callbacks()

Resets global variables and the libxml2 callback stack.

lib_init_callbacks()

Used internally for callback registration at C-level.

lib_cleanup_callbacks()

Used internally for callback resetting at the C-level.

EXAMPLE CALLBACKS

The following example is a purely fictitious example that uses a MyScheme::Handler object that responds to methods similar to an IO::Handle.

```
 # Define the four callback functions
 sub match_uri {
    my $uri = shift;
    return $uri =~ /^myscheme:/; # trigger our callback group at a 'myscheme' URIs
 }
 sub open_uri {
    my $uri = shift;
    my $handler = MyScheme::Handler->new($uri);
    return $handler;
 }
 # The returned $buffer will be parsed by the libxml2 parser
 sub read_uri {
    my $handler = shift;
    my $length = shift;
    my $buffer;
    read($handler, $buffer, $length);
    return $buffer; # $buffer will be an empty string '' if read() is done
 }
 # Close the handle associated with the resource.
```

```
    sub close_uri {
        my $handler = shift;
        close($handler);
    }
    # Register them with a instance of XML::LibXML::InputCallback
    my $input_callbacks = XML::LibXML::InputCallback->new();
    $input_callbacks->register_callbacks([ \&match_uri, \&open_uri,
                                           \&read_uri, \&close_uri ] );
    # Register the callback group at a parser instance
    $parser->input_callbacks( $input_callbacks );
    # $some_xml_file will be parsed using our callbacks
    $parser->parse_file( $some_xml_file );
```

AUTHORS

Matt Sergeant, Christian Glahn, Petr Pajas

VERSION

2.0134

COPYRIGHT

LICENSE