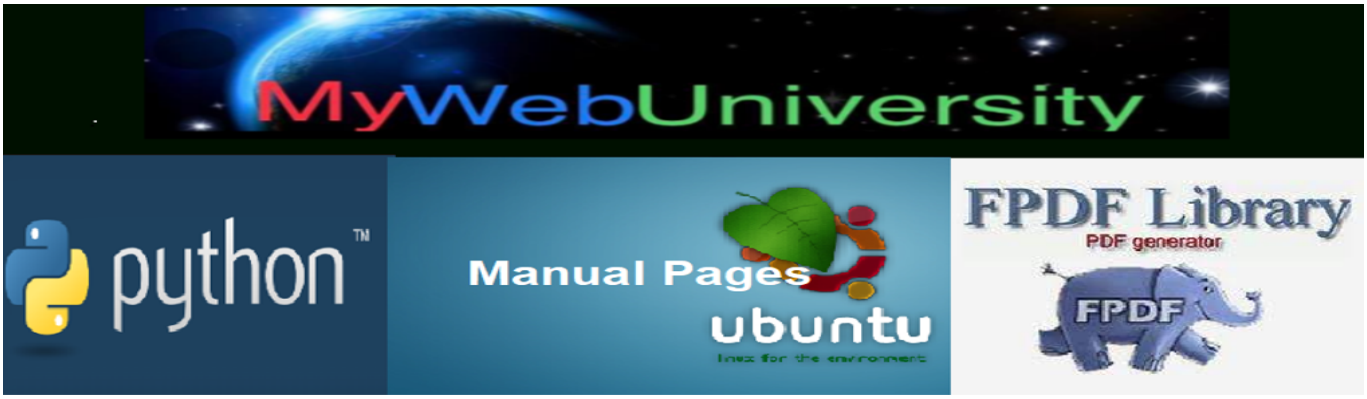

Rocky Enterprise Linux 9.2 Manual Pages on command 'HTML::Element::traverse.3pm'

$ man HTML::Element::traverse.3pm

HTML::Element::traverse(3pm) User Contributed Perl Documentation HTML::Element::traverse(3pm)

NAME

HTML::Element::traverse - discussion of HTML::Element's traverse method

VERSION

This document describes version 5.07 of HTML::Element::traverse, released August 31, 2017

as part of HTML-Tree.

SYNOPSIS

```
 # $element->traverse is unnecessary and obscure.
 #   Don't use it in new code.
```

DESCRIPTION

"HTML::Element" provides a method "traverse" that traverses the tree and calls user-specified callbacks for each node, in pre- or post-order. However, use of the method is quite superfluous: if you want to recursively visit every node in the tree, it's almost always simpler to write a subroutine does just that, than it is to bundle up the pre- and/or post-order code in callbacks for the "traverse" method.

EXAMPLES

Suppose you want to traverse at/under a node $tree and give elements an 'id' attribute unless they already have one.

You can use the "traverse" method:

```
 {
   my $counter = 'x0000';
   $start_node->traverse(
     [ # Callbacks;
```

```
        # pre-order callback:
        sub {
          my $x = $_[0];
          $x->attr('id', $counter++) unless defined $x->attr('id');
          return HTML::Element::OK; # keep traversing
        },
        # post-order callback:
        undef
      ],
      1, # don't call the callbacks for text nodes
    );
  }
```

or you can just be simple and clear (and not have to understand the calling format for "traverse") by writing a sub that traverses the tree by just calling itself:

```
  {
    my $counter = 'x0000';
    sub give_id {
      my $x = $_[0];
      $x->attr('id', $counter++) unless defined $x->attr('id');
      foreach my $c ($x->content_list) {
        give_id($c) if ref $c; # ignore text nodes
      }
    };
    give_id($start_node);
  }
```

See, isn't that nice and clear?

But, if you really need to know:

THE TRAVERSE METHOD

The "traverse()" method is a general object-method for traversing a tree or subtree and calling user-specified callbacks. It accepts the following syntaxes:

```
$h->traverse(\&callback)
or $h->traverse(\&callback, $ignore_text)
or $h->traverse( [\&pre_callback,\&post_callback] , $ignore_text)
```

These all mean to traverse the element and all of its children. That is, this method starts at node $h, "pre-order visits" $h, traverses its children, and then will "post-order visit" $h. "Visiting" means that the callback routine is called, with these arguments:

$_[0] : the node (element or text segment),

$_[1] : a startflag, and

$_[2] : the depth

If the $ignore_text parameter is given and true, then the pre-order call will not be happen for text content.

The startflag is 1 when we enter a node (i.e., in pre-order calls) and 0 when we leave the node (in post-order calls).

Note, however, that post-order calls don't happen for nodes that are text segments or are elements that are prototypically empty (like "br", "hr", etc.).

If we visit text nodes (i.e., unless $ignore_text is given and true), then when text nodes are visited, we will also pass two extra arguments to the callback:

$_[3] : the element that's the parent
of this text node

$_[4] : the index of this text node
in its parent's content list

Note that you can specify that the pre-order routine can be a different routine from the post-order one:

```
$h->traverse( [\&pre_callback,\&post_callback], ...);
```

You can also specify that no post-order calls are to be made, by providing a false value as the post-order routine:

```
$h->traverse([ \&pre_callback,0 ], ...);
```

And similarly for suppressing pre-order callbacks:

```
$h->traverse([ 0,\&post_callback ], ...);
```

Note that these two syntaxes specify the same operation:

```
$h->traverse([\&foo,\&foo], ...);
$h->traverse( \&foo       , ...);
```

The return values from calls to your pre- or post-order routines are significant, and are used to control recursion into the tree.

These are the values you can return, listed in descending order of my estimation of their

usefulness:

HTML::Element::OK, 1, or any other true value

...to keep on traversing.

Note that "HTML::Element::OK" et al are constants. So if you're running under "use strict" (as I hope you are), and you say: "return HTML::Element::PRUEN" the compiler will flag this as an error (an unallowable bareword, specifically), whereas if you spell PRUNE correctly, the compiler will not complain.

undef, 0, '0', '', or HTML::Element::PRUNE

...to block traversing under the current element's content. (This is ignored if received from a post-order callback, since by then the recursion has already happened.) If this is returned by a pre-order callback, no post-order callback for the current node will happen. (Recall that if your callback exits with just "return;", it is returning undef -- at least in scalar context, and "traverse" always calls your callbacks in scalar context.)

HTML::Element::ABORT

...to abort the whole traversal immediately. This is often useful when you're looking for just the first node in the tree that meets some criterion of yours.

HTML::Element::PRUNE_UP

...to abort continued traversal into this node and its parent node. No post-order callback for the current or parent node will happen.

HTML::Element::PRUNE_SOFTLY

Like PRUNE, except that the post-order call for the current node is not blocked.

Almost every task to do with extracting information from a tree can be expressed in terms of traverse operations (usually in only one pass, and usually paying attention to only pre-order, or to only post-order), or operations based on traversing. (In fact, many of the other methods in this class are basically calls to traverse() with particular arguments.)

The source code for HTML::Element and HTML::TreeBuilder contain several examples of the use of the "traverse" method to gather information about the content of trees and subtrees.

(Note: you should not change the structure of a tree while you are traversing it.)

[End of documentation for the "traverse()" method]

Now, if you've been reading Structure and Interpretation of Computer Programs too much, maybe you even want a recursive lambda. Go ahead:

```
 {
   my $counter = 'x0000';
   my $give_id;
   $give_id = sub {
     my $x = $_[0];
     $x->attr('id', $counter++) unless defined $x->attr('id');
     foreach my $c ($x->content_list) {
       $give_id->($c) if ref $c; # ignore text nodes
     }
   };
   $give_id->($start_node);
   undef $give_id;
 }
```

It's a bit nutty, and it's still more concise than a call to the "traverse" method!

It is left as an exercise to the reader to figure out how to do the same thing without using a $give_id symbol at all.

It is also left as an exercise to the reader to figure out why I undefine $give_id, above; and why I could achieved the same effect with any of:

```
   $give_id = 'I like pie!';
  # or...
   $give_id = [];
  # or even;
   $give_id = sub { print "Mmmm pie!\n" };
```

But not:

```
   $give_id = sub { print "I'm $give_id and I like pie!\n" };
  # nor...
   $give_id = \$give_id;
  # nor...
   $give_id = { 'pie' => \$give_id, 'mode' => 'a la' };
```

Doing Recursive Things Iteratively

Note that you may at times see an iterative implementation of pre-order traversal, like

so:

```
{
  my @to_do = ($tree); # start-node
  while(@to_do) {
    my $this = shift @to_do;
    # "Visit" the node:
    $this->attr('id', $counter++)
     unless defined $this->attr('id');
    unshift @to_do, grep ref $_, $this->content_list;
     # Put children on the stack -- they'll be visited next
  }
}
```

This can under certain circumstances be more efficient than just a normal recursive routine, but at the cost of being rather obscure. It gains efficiency by avoiding the overhead of function-calling, but since there are several method dispatches however you do it (to "attr" and "content_list"), the overhead for a simple function call is insignificant.

Pruning and Whatnot

The "traverse" method does have the fairly neat features of the "ABORT", "PRUNE_UP" and "PRUNE_SOFTLY" signals. None of these can be implemented totally straightforwardly with recursive routines, but it is quite possible. "ABORT"-like behavior can be implemented either with using non-local returning with "eval"/"die":

```
my $died_on; # if you need to know where...
sub thing {
  ... visits $_[0]...
  ... maybe set $died_on to $_[0] and die "ABORT_TRAV" ...
  ... else call thing($child) for each child...
  ...any post-order visiting $_[0]...
}
eval { thing($node) };
if($@) {
  if($@ =~ m<^ABORT_TRAV>) {
    ...it died (aborted) on $died_on...
```

```
    } else {
      die $@; # some REAL error happened
    }
  }
```

or you can just do it with flags:

```
  my($abort_flag, $died_on);
  sub thing {
    ... visits $_[0]...
    ... maybe set $abort_flag = 1; $died_on = $_[0]; return;
    foreach my $c ($_[0]->content_list) {
      thing($c);
      return if $abort_flag;
    }
    ...any post-order visiting $_[0]...
    return;
  }
  $abort_flag = $died_on = undef;
  thing($node);
  ...if defined $abort_flag, it died on $died_on
```

SEE ALSO

HTML::Element

AUTHOR

Current maintainers:

? Christopher J. Madsen "<perl AT cjmweb.net>"

? Jeff Fearn "<jfearn AT cpan.org>"

Original HTML-Tree author:

? Gisle Aas

Former maintainers:

? Sean M. Burke

? Andy Lester

? Pete Krawczyk "<petek AT cpan.org>"

You can follow or contribute to HTML-Tree's development at <https://github.com/kentfredric/HTML-Tree>.

COPYRIGHT