NAME

HTML::TreeBuilder - Parser that builds a HTML syntax tree

VERSION

This document describes version 5.03 of HTML::TreeBuilder, released September 22, 2012 as part of HTML-Tree.

SYNOPSIS

```
use HTML::TreeBuilder 5 -weak; # Ensure weak references in use

foreach my $file_name (@ARGV) {
my $tree = HTML::TreeBuilder->new; # empty tree
$tree->parse_file($file_name);
print "Hey, here's a dump of the parse tree of $file_name:\n";
$tree->dump; # a method we inherit from HTML::Element
print "And here it is, bizarrely rerendered as HTML:\n",
$tree->as_HTML, "\n";

# Now that we're done with it, we must destroy it.
# $tree = $tree->delete; # Not required with weak references
}
```

DESCRIPTION

(This class is part of the HTML::Tree dist.)

This class is for HTML syntax trees that get built out of HTML source. The way to use it is to:

1. start a new (empty) HTML::TreeBuilder object,

2. then use one of the methods from HTML::Parser (presumably with `$tree->parse_file($filename)` for files, or with `$tree->parse($document_content)` and `$tree->eof` if you've got the content in a string) to parse the HTML document into the tree `$tree`.

(You can combine steps 1 and 2 with the "new_from_file" or "new_from_content" methods.)

2b. call `$root->elementify()` if you want.

3. do whatever you need to do with the syntax tree, presumably involving traversing it looking for some bit of information in it,

4. previous versions of HTML::TreeBuilder required you to call `$tree->delete()` to erase the contents of the tree from memory when you're done with the tree. This is not normally required anymore. See "Weak References" in HTML::Element for details.

ATTRIBUTES

Most of the following attributes native to HTML::TreeBuilder control how parsing takes place; they should be set *before* you try parsing into the given object. You can set the attributes by passing a TRUE or FALSE value as argument. E.g., `$root->implicit_tags` returns the current setting for the `implicit_tags` option, `$root->implicit_tags(1)` turns that option on, and `$root->implicit_tags(0)` turns it off.

implicit_tags

Setting this attribute to true will instruct the parser to try to deduce implicit elements and implicit end tags. If it is false you get a parse tree that just reflects the text as it stands, which is unlikely to be useful for anything but quick and dirty parsing. (In fact, I'd be curious to hear from anyone who finds it useful to have `implicit_tags` set to false.) Default is true.

Implicit elements have the "implicit" in HTML::Element attribute set.

implicit_body_p_tag

This controls an aspect of implicit element behavior, if `implicit_tags` is on: If a text element (PCDATA) or a phrasal element (such as `<em>`) is to be inserted under `<body>`, two things can happen: if `implicit_body_p_tag` is true, it's placed under a new, implicit `<p>` tag. (Past DTDs suggested this was the only correct behavior, and this is how past versions of this module behaved.) But if `implicit_body_p_tag` is false, nothing is implicated — the PCDATA or phrasal element is simply placed under `<body>`. Default is false.

no_expand_entities
This attribute controls whether entities are decoded during the initial parse of the source. Enable this if you don't want entities decoded to their character value. e.g. '&amp;' is decoded to '&' by default, but will be unchanged if this is enabled. Default is false (entities will be decoded.)

ignore_unknown
This attribute controls whether unknown tags should be represented as elements in the parse tree, or whether they should be ignored. Default is true (to ignore unknown tags.)

ignore_text
Do not represent the text content of elements. This saves space if all you want is to examine the structure of the document. Default is false.

ignore_ignorable_whitespace
If set to true, TreeBuilder will try to avoid creating ignorable whitespace text nodes in the tree. Default is true. (In fact, I'd be interested in hearing if there's ever a case where you need this off, or where leaving it on leads to incorrect behavior.)

no_space_compacting
This determines whether TreeBuilder compacts all whitespace strings in the document (well, outside of PRE or TEXTAREA elements), or leaves them alone. Normally (default, value of 0), each string of contiguous whitespace in the document is turned into a single space. But that's not done if `no_space_compacting` is set to 1.

Setting `no_space_compacting` to 1 might be useful if you want to read in a tree just to make some minor changes to it before writing it back out.

This method is experimental. If you use it, be sure to report any problems you might have with it.

p_strict
If set to true (and it defaults to false), TreeBuilder will take a narrower than normal view of what can be under a `<p>` element; if it sees a non-phrasal element about to be inserted under a `<p>`, it will close that `<p>`. Otherwise it will close `<p>` elements only for other `<p>`'s, headings, and `<form>` (although the latter may be removed in future versions).

For example, when going thru this snippet of code,

```
<p>stuff
<ul>
```

TreeBuilder will normally (with `p_strict` false) put the `<ul>` element under the `<p>` element. However, with `p_strict` set to true, it will close the `<p>` first.

In theory, there should be strictness options like this for other/all elements besides just `<p>`; but I treat this as a special case simply because of the fact that `<p>` occurs so frequently and its end-tag is omitted so often; and also because application of strictness rules at parse-time across all elements often makes tiny errors in HTML coding produce drastically bad parse-trees, in my experience.

If you find that you wish you had an option like this to enforce content-models on all elements, then I suggest that what you want is content-model checking as a stage after TreeBuilder has finished parsing.

store_comments
This determines whether TreeBuilder will normally store comments found while parsing content into `$root`. Currently, this is off by default.

store_declarations

This determines whether TreeBuilder will normally store markup declarations found while parsing content into `$root`. This is on by default.

store_pis

This determines whether TreeBuilder will normally store processing instructions found while parsing content into `$root` — assuming a recent version of HTML::Parser (old versions won't parse PIs correctly). Currently, this is off (false) by default.

It is somewhat of a known bug (to be fixed one of these days, if anyone needs it?) that PIs in the preamble (before the `<html>` start-tag) end up actually *under* the `<html>` element.

warn

This determines whether syntax errors during parsing should generate warnings, emitted via Perl's `warn` function.

This is off (false) by default.

METHODS

Objects of this class inherit the methods of both HTML::Parser and HTML::Element. The methods inherited from HTML::Parser are used for building the HTML tree, and the methods inherited from HTML::Element are what you use to scrutinize the tree. Besides this (HTML::TreeBuilder) documentation, you must also carefully read the HTML::Element documentation, and also skim the HTML::Parser documentation — probably only its parse and parse_file methods are of interest.

new_from_file

```
 $root = HTML::TreeBuilder->new_from_file($filename_or_filehandle);
```

This "shortcut" constructor merely combines constructing a new object (with the "new" method, below), and calling `$new->parse_file(...)` on it. Returns the new object. Note that this provides no way of setting any parse options like `store_comments` (for that, call `new`, and then set options, before calling `parse_file`). See the notes (below) on parameters to "parse_file".

If HTML::TreeBuilder is unable to read the file, then `new_from_file` dies. The error can also be found in `$!`. (This behavior is new in HTML-Tree 5. Previous versions returned a tree with only implicit elements.)

new_from_content

```
 $root = HTML::TreeBuilder->new_from_content(...);
```

This "shortcut" constructor merely combines constructing a new object (with the "new" method, below), and calling `for(...){$new->parse($_)}` and `$new->eof` on it. Returns the new object. Note that this provides no way of setting any parse options like `store_comments` (for that, call `new`, and then set options, before calling `parse`). Example usages: `HTML::TreeBuilder->new_from_content(@lines)`, or `HTML::TreeBuilder->new_from_content($content)`.

new_from_url

```
 $root = HTML::TreeBuilder->new_from_url($url)
```

This "shortcut" constructor combines constructing a new object (with the "new" method, below), loading LWP::UserAgent, fetching the specified URL, and calling `$new->parse( $response->decoded_content)` and `$new->eof` on it. Returns the new object. Note that this provides no way of setting any parse options like `store_comments`.

If LWP is unable to fetch the URL, or the response is not HTML (as determined by "content_is_html" in HTTP::Headers), then `new_from_url` dies, and the HTTP::Response object is found in `$HTML::TreeBuilder::lwp_response`.

You must have installed LWP::UserAgent for this method to work. LWP is not installed automatically, because it's a large set of modules and you might not need it.

new

```
$root = HTML::TreeBuilder->new();
```

This creates a new HTML::TreeBuilder object. This method takes no attributes.

parse_file

```
$root->parse_file(...)
```

[An important method inherited from HTML::Parser, which see. Current versions of HTML::Parser can take a filespec, or a filehandle object, like *FOO, or some object from class IO::Handle, IO::File, IO::Socket) or the like. I think you should check that a given file exists *before* calling `$root->parse_file($filespec)`.]

When you pass a filename to `parse_file`, HTML::Parser opens it in binary mode, which means it's interpreted as Latin-1 (ISO-8859-1). If the file is in another encoding, like UTF-8 or UTF-16, this will not do the right thing.

One solution is to open the file yourself using the proper `:encoding` layer, and pass the filehandle to `parse_file`. You can automate this process by using "html_file" in IO::HTML which will use the HTML5 encoding sniffing algorithm to automatically determine the proper `:encoding` layer and apply it.

In the next major release of HTML-Tree, I plan to have it use IO::HTML automatically. If you really want your file opened in binary mode, you should open it yourself and pass the filehandle to `parse_file`.

The return value is `undef` if there's an error opening the file. In that case, the error will be in `$!`.

parse

```
$root->parse(...)
```

[A important method inherited from HTML::Parser, which see. See the note below for `$root->eof()`.]

eof

```
$root->eof();
```

This signals that you're finished parsing content into this tree; this runs various kinds of crucial cleanup on the tree. This is called *for you* when you call `$root->parse_file(...)`, but not when you call `$root->parse(...)`. So if you call `$root->parse(...)`, then you *must* call `$root->eof()` once you've finished feeding all the chunks to `parse(...)`, and before you actually start doing anything else with the tree in `$root`.

parse_content

```
$root->parse_content(...);
```

Basically a handy alias for `$root->parse(...); $root->eof`. Takes the exact same arguments as `$root->parse()`.

delete

```
$root->delete();
```

[A previously important method inherited from HTML::Element, which see.]

elementify

```
$root->elementify();
```

This changes the class of the object in `$root` from HTML::TreeBuilder to the class used for all the rest of the elements in that tree (generally HTML::Element). Returns `$root`.

For most purposes, this is unnecessary, but if you call this after (after!!) you've finished building a tree, then it keeps you from accidentally trying to call anything but HTML::Element methods on it. (I.e., if you accidentally call `$root->parse_file(...)` on the already-complete and elementified tree, then instead of charging ahead and *wreaking havoc*, it'll throw a fatal error — since `$root` is now an object just of class HTML::Element which has no `parse_file` method.

Note that `elementify` currently deletes all the private attributes of `$root` except for "_tag", "_parent", "_content", "_pos", and "_implicit". If anyone requests that I change this to leave in yet more private attributes, I might do so, in future versions.

guts

```
@nodes = $root->guts();
$parent_for_nodes = $root->guts();
```

In list context (as in the first case), this method returns the topmost non-implicit nodes in a tree. This is useful when you're parsing HTML code that you know doesn't expect an HTML document, but instead just a fragment of an HTML document. For example, if you wanted the parse tree for a file consisting of just this:

```
<li>I like pie!
```

Then you would get that with `@nodes = $root->guts();`. It so happens that in this case, `@nodes` will contain just one element object, representing the `<li>` node (with "I like pie!" being its text child node). However, consider if you were parsing this:

```
<hr>Hooboy!<hr>
```

In that case, `$root->guts()` would return three items: an element object for the first `<hr>`, a text string "Hooboy!", and another `<hr>` element object.

For cases where you want definitely one element (so you can treat it as a "document fragment", roughly speaking), call `guts()` in scalar context, as in `$parent_for_nodes = $root->guts()`. That works like `guts()` in list context; in fact, `guts()` in list context would have returned exactly one value, and if it would have been an object (as opposed to a text string), then that's what `guts` in scalar context will return. Otherwise, if `guts()` in list context would have returned no values at all, then `guts()` in scalar context returns undef. In all other cases, `guts()` in scalar context returns an implicit `<div>` element node, with children consisting of whatever nodes `guts()` in list context would have returned. Note that that may detach those nodes from `$root`'s tree.

disembowel

```
@nodes = $root->disembowel();
$parent_for_nodes = $root->disembowel();
```

The `disembowel()` method works just like the `guts()` method, except that disembowel definitively destroys the tree above the nodes that are returned. Usually when you want the guts from a tree, you're just going to toss out the rest of the tree anyway, so this saves you the bother. (Remember, "disembowel" means "remove the guts from".)

INTERNAL METHODS

You should not need to call any of the following methods directly.

element_class

```
$classname = $h->element_class;
```

This method returns the class which will be used for new elements. It defaults to HTML::Element, but can be overridden by subclassing or esoteric means best left to those will will read the source and then not complain when those esoteric means change. (Just subclass.)

comment

Accept a "here's a comment" signal from HTML::Parser.

declaration

Accept a "here's a markup declaration" signal from HTML::Parser.

done

TODO: document

end

Either: Acccept an end-tag signal from HTML::Parser Or: Method for closing currently open elements in some fairly complex way, as used by other methods in this class.

TODO: Why is this hidden?

process

Accept a "here's a PI" signal from HTML::Parser.

start

Accept a signal from HTML::Parser for start-tags.

TODO: Why is this hidden?

stunt

TODO: document

stunted

TODO: document

text

Accept a "here's a text token" signal from HTML::Parser.

TODO: Why is this hidden?

tighten_up

Legacy

Redirects to "delete_ignorable_whitespace" in HTML::Element.

warning

Wrapper for CORE::warn

TODO: why not just use carp?

SUBROUTINES

DEBUG

Are we in Debug mode? This is a constant subroutine, to allow compile-time optimizations. To control debug mode, set `$HTML::TreeBuilder::DEBUG` *before* loading HTML::TreeBuilder.

HTML AND ITS DISCONTENTS

HTML is rather harder to parse than people who write it generally suspect.

Here's the problem: HTML is a kind of SGML that permits "minimization" and "implication". In short, this means that you don't have to close every tag you open (because the opening of a subsequent tag may implicitly close it), and if you use a tag that can't occur in the context you seem to using it in, under certain conditions the parser will be able to realize you mean to leave the current context and enter the new one, that being the only one that your code could correctly be interpreted in.

Now, this would all work flawlessly and unproblematically if: 1) all the rules that both prescribe and describe HTML were (and had been) clearly set out, and 2) everyone was aware of these rules and wrote their code in compliance to them.

However, it didn't happen that way, and so most HTML pages are difficult if not impossible to correctly parse with nearly any set of straightforward SGML rules. That's why the internals of HTML::TreeBuilder consist of lots and lots of special cases — instead of being just a generic SGML parser with HTML DTD rules plugged in.

TRANSLATIONS?

The techniques that HTML::TreeBuilder uses to perform what I consider very robust parses on everyday code are not things that can work only in Perl. To date, the algorithms at the center of HTML::TreeBuilder have been implemented only in Perl, as far as I know; and I don't foresee getting around to implementing them in any other language any time soon.

If, however, anyone is looking for a semester project for an applied programming class (or if they merely enjoy *extra-curricular* masochism), they might do well to see about choosing as a topic the implementation/adaptation of these routines to any other interesting programming language that you feel currently suffers from a lack of robust HTML-parsing. I welcome correspondence on this subject, and point out that one can learn a great deal about languages by trying to translate between them, and then comparing the result.

The HTML::TreeBuilder source may seem long and complex, but it is rather well commented, and symbol names are generally self-explanatory. (You are encouraged to read the Mozilla HTML parser source for comparison.) Some of the complexity comes from little-used features, and some of it comes from having the HTML tokenizer (HTML::Parser) being a separate module, requiring somewhat of a different interface than you'd find in a combined tokenizer and tree-builder. But most of the length of the source comes from the fact that it's essentially a long list of special cases, with lots and lots of sanity-checking, and sanity-recovery — because, as Roseanne Rosannadanna once said, it's always *something*.

Users looking to compare several HTML parsers should look at the source for Raggett's Tidy (<`http://www.w3.org/People/Raggett/tidy/`>), Mozilla (<`http://www.mozilla.org/`>), and possibly root around the browsers section of Yahoo to find the various open-source ones (<`http://dir.yahoo.com/Computers_and_Internet/Software/Internet/World_Wide_Web/Browsers/`>).

BUGS

* Framesets seem to work correctly now. Email me if you get a strange parse from a document with framesets.

* Really bad HTML code will, often as not, make for a somewhat objectionable parse tree. Regrettable, but unavoidably true.

* If you're running with `implicit_tags` off (God help you!), consider that `$tree->content_list` probably contains the tree or grove from the parse, and not `$tree` itself (which will, oddly enough, be an implicit `<html>` element). This seems counter-intuitive and problematic; but seeing as how almost no HTML ever parses correctly with `implicit_tags` off, this interface oddity seems the least of your problems.

BUG REPORTS

When a document parses in a way different from how you think it should, I ask that you report this to me as a bug. The first thing you should do is copy the document, trim out as much of it as you can while still producing the bug in question, and *then* email me that mini-document *and* the code you're using to parse it, to the HTML::Tree bug queue at `<bug-html-tree at rt.cpan.org>`.

Include a note as to how it parses (presumably including its `$tree->dump` output), and then a *careful and clear* explanation of where you think the parser is going astray, and how you would prefer that it work instead.

SEE ALSO

For more information about the HTML-Tree distribution: HTML::Tree.

Modules used by HTML::TreeBuilder: HTML::Parser, HTML::Element, HTML::Tagset.

For converting between XML::DOM::Node, HTML::Element, and XML::Element trees: HTML::DOMbo.

For opening a HTML file with automatic charset detection: IO::HTML

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 <http://github.com/madsen/HTML-Tree>.

COPYRIGHT AND LICENSE