By Tommie Jones

Table of Contents

Web Crawlers Are Not Just For Search Engines Anymore

Any organization that spends money for marketing research or generating sales leads can benefit from building a web crawler. Instead of spending tens of thousands of dollars for a boxed market research survey, a web crawler can be used to ferret information from the web.

For example: 1. Retail oriented companies can build web crawlers to find trends mentioned in web logs. 2. Software consulting companies could crawl industry specific news groups and mailing lists for potential customers asking for advice. 3. Job placement services could search company sites for an increase in Job postings.

All of these tasks can be accomplished with creative use of Perl and it's abundance of CPAN (Comprehensive Perl Archive − the repository of Perl module/libraries) modules. In this article the main topic will include some of the CPAN modules available and how they can be used to accomplish tasks similar to the ones above.

Why Perl? Why not? Perl is an excellent tool for a web mining project. Perl's basic but powerful built−in data structures, easily accessible regular expressions and large selection of CPAN modules show that Perl easily meets the application's requirements.

The rest of this article will discuss some CPAN Modules that will be useful when building a Perl−based web crawler.

Accessing The Net (LWP)

LWP, which stands for the libwww−Perl library, is a common module that may have comes with most installations of Perl. LWP (as quoted from the LWP perldoc) is a collection of Perl modules that provide a consistent and simple application−programming interface to the World Wide Web. LWP provides support for redirection, cookies, basic authentication and robot.txt parsing. For the majority of web−crawling requirements a developer can use LWP::Simple. LWP−Simple allows the developer to store the head or body of a web page (given its URL) in a scalar variable or file. Here is an example.

```
#!/usr/bin/perl
use LWP::Simple;

my $content = get("http://www.yahoo.com"); #Store the output
of the web page (html and all) in content
if (defined $content)
{
#$content will contain the html associated with the url
mentioned above.
print $content;
}
else
{
#If an error occurs then $content will not be defined.
print "Error: Get failed";
}
```

After loading the LWP::Simple module with the use command the get subroutine is called to download the html on the http://www.yahoo.com web site. The html is stored in the $content variable. If there is not an error the $content value is printed to standard output.

Other modules exist in the LWP::Bundle that handle cookies, automatic redirection and other things. For more information please read the perldoc on LWP::RobotUA and LWP::UserAgent.

Cut Along The Table Lines (HTML::TableExtract)

HTML Tables not only help visually segregate data on a web page but they also provide helpful landmarks when parsing web pages. Tables are used to align information on web pages. Tables can force information to be in one location or to take up a certain width of a screen.

Tables become even more important on dynamic data driven web sites. This is because on most websites content such as articles are stored separately from the page's visual aspects. When generating the HTML pages the content is separated from other features of the web page with a table. In other words the main page might change but the layout defined by tables rarely changes. This is important because when processing a web page the developer will often want to ignore a lot of the static or template data but want to access the dynamic data. The developer of a web crawler will want to identify what tables/rows/cells the data you are interested in is located and pull his information from there.

Fortunately there exists a Perl Module designed to parse HTML tables. The following example script shows how a particular table can be parsed out of an HTML page.

```
#!/usr/bin/Perl

use lib qw( ..);
use HTML::TableExtract;
use LWP::Simple;
use Data::Dumper;

my $te = new HTML::TableExtract( depth=>3, count=>0,
gridmap=>0);

my $content = get("http://www.computerjobs.com");

$te−>parse($content);
foreach $ts ($te−>table_states)
{
foreach $row ($ts−>rows)
{
print Dumper $row;
# print Dumper $row if (scalar(@$row) == 2);
}
}
```

Now to explain the highlights of the code.

```
my $te = new HTML::TableExtract( depth=>3, count=>0,
gridmap=>0);
```

This is where we create/initialize the TableExtract object. We pass three parameters to the page. depth => 3 − this is the depth of the table we want to work with. This suggest that this table is inside a table (depth=2) which is inside another table (depth = 1) which is at last in another table (depth=>0) count => 0 − More than one table can exists at the level three. count=>0 suggest that it is the first table that is at depth 3. gridmap => 0 − represents tables as a tree instead of a map.

The combination of these two parameters uniquely identify any table in an html page. Note that the table identified by (depth=>3, count=>1) is not necessarily the neighbor to the (depth=>3, count=>0) table. For instance

```
<table> <tr><td> /*Table depth=>0 count=>0 */
<table><tr><td> /* Table depth=>1 count=>0 */
<table><tr><td>
/* Table depth=>2 count=>0 */
</td></tr></table>
</td></tr></table>
<table><tr><td> /*Table depth=>1 count =>1 */
<table><tr><td>
/* Table depth=>2 count=>1 */
</td></tr></table>
<table><tr><td>
/* Table depth=>2 count=>2 */
</td></tr></table>
</td></tr></table>
</table><tr><td>
```

In the example shown above there are three tables at depth 2 . For the tables (depth=>2 count=>0) and (depth=>2 count=>1) notice that they do not share the same parent table. The count does not reset to zero when the html backs out of the depth. The table identified as (depth=>2 count=>1) is literally the second table(count = 1) at the third depth (both numbers start at zero.).

The gridmap option tells whether to logically represent data as a grid or a tree. Consider the following example.

```
<table>
<tr>
<td> location [1,1] </td>
<td> location [1,2] </td>
</tr>
<tr colspan=2>
<td> location [2,1] </td>
</tr>
<table>
```

If gridmap=1 (as is by default) then the cell [2,2] will be defined but empty. This is because gridmap=1 forces the table to look like a grid. If gridmap=0 the map table would look like a tree where each row could have a different number of cells. Trying to access position[2,2] will not be defined.

After the table is identified, the object representing the table can be accessed. These verbs include table_states and table_state. Table_state takes a depth and a count as an identifier to return a table state object. Table_states returns an array of table_states to represent our code.

A TableExtract object can represent multiple tables. This can be accomplished by only specifying depth or count (not both). This will return an object representing multiple tables.

In the first for loop we are going through the list of tables. This is done with the table_states object. The inner loop loops through the rows inside each table (represented by the tr tag.) The results of the rows tag is an array of arrays that represent the two−dimensional table.

Learning From Links (HTML::LinkExtor)

A lot can be learned from what web sites link to. Contributors to Web based Bulletin boards will supply links to the subject of their discussions. Articles that mention products will provide links to where that product can be ordered. Google's web page ranking and grouping algorithms calculate how important a web page is by how many sites link to it. Another possible use would be an organization that wants to write a web crawler for marketing their product may search the web for any mention of their or their competitor's product.

A useful module to extract links out of a web page is HTML::LinkExtor. Here is an example:

```
#!/usr/bin/Perl

use LWP::Simple;
use HTML::LinkExtor;
use Data::Dumper;

my $content = get("http://www.yahoo.com"); #Get web page in
content
die "get failed" if (!defined $content);
my $parser = HTML::LinkExtor->new(); #create LinkExtor object
with no callbacks
$parser->parse($content); #parse content

my @links = $parser->links; #get list of links

print Dumper \@links; #print list of links out.
```

This script will parse a website specified in the get command. After the web site's content is parsed the resulting sub array will be stored in the links array (@links). Each sub array in the links array will represent a link. The sub array contains the element tag name as the first element in the sub array. The remaining elements are name/value pairs that were in each tag. The tags that are processed are not only 'A' link tags but also 'img' tags and 'form' tags. Any tag that can have a 'href' as an attribute.

Another approach to use HTML::LinkExtor is to provide a callback function in the new method. When parsing the content the callback function will be called for every link tag found. Please review the perldoc for HTML::LinkExtor for more information.

Generic HTML Parsing (HTML::Parser)

The two previously mentioned modules inherit from the HTML::Parser. HTML::Parser works in a similar manner to the SAX interface for XML. HTML::Parser is an event driven parser designed to work with HTML. Recognized events include start tags, end tags, text and comments. For each event handler argspec, which is a list of information that will be passed to the handler, can be defined.

Here is an example to work through.

```
#!/usr/bin/Perl

use LWP::Simple;
use HTML::Parser;
use Data::Dumper;
my $url = shift @ARGV;
die "No URL specified on command line." unless (defined $url);
my $content = get($url); #put site html in $content.
die "get failed" if (!defined $content);

# create parser object
my $parser = HTML::Parser->new(api_version=>3,
start_h=>[\&startTag, 'tag, attr'] ,
end_h=>[\&endTag, 'tag'] ,
text_h=>[\&textElem, 'text']
);
#parse object.
$parser->parse($content);

sub startTag
{
my ($tag, $attrHash) = @_;
print "TAG: $tag \n";
print "ATTR HASH: " , Dumper $attrHash , "\n";
print "-----\n";
}

sub endTag
{
my $tag = shift;
print "END TAG: $tag \n";
print "-----\n";
}

sub textElem
{
my $text = shift;
print "TEXT: $text \n";
print "-----\n";
}
```

Note that in the above code the events are defined in the HTML::Parser tag. For each event defined (The name component of the passed variables that end in _h) A reference to a subroutine and a string which is the argspec. Whenever the event occurs the referenced subroutine will be called passing as parameters the argspec. For example whenever a start tag occurs (A tag that looks like ' <...> ') the startTag subroutine is called passing the 'tag' scalar and the attr hash−ref. Possible argspecs include: self, tokens, tokenpos, token0, tagname, tag, attr, attrseq, @attr, text, dtext, is_cdata, offset, event and many others. Most of these are the same data in a slightly different format. The perldoc for HTML::Parser will provides more information.

HTML::Parser is an excellent module and is very flexible.

Checking For Sameness (String::CRC)

String::CRC is a simple and little known module that provides simple checksum support. Checksums are often used as sanity checks. Given a string of text they generate a number. Doing small modifications to the string drastically changes the value of the checksum. That is not to say that a checksum is unique for every string. What is important for a checksum is that a minor change in the string requires a drastically changed string.

How would a checksum be used? An example would be if you transfer a file from one machine to another and were not sure the file had been corrupted. A checksum can be run on the original file and the file at its new location. If the checksums are the same then the transfer can be considered successful. It would be very unlikely that a file could be corrupted (by accident) and still generate the same checksum.

Here is an example of String::CRC in action:

```
#!/usr/bin/perl

use String::CRC;
my $str = " some text string ";
my ($crc) = crc($str, 32);
print "Check sum $str −> $crc\n";
$str = $str . " ";
$crc = crc($str, 32);
print "Check sum $str −> $crc\n";
```

By running this script you will see just adding an additional white space can significantly change the result of crc.

Bringing It All Together

The following code is a Perl script that uses the discussed modules (except HTML::Parser). For this script you give it three URLs. All three URLs should refer to different pages on a single web site.

The first two URLs are divided into tables and their cells contents are compared to each other. When the contents of the cells of each table match they are assumed to make up the template. If the cells content is different between the two tables then these cells are identified as the dynamic content. Each table is stored in a four dimensional array. The first two dimensions identify a table (depth, count) as discussed in the HTML::TableExtract section. The last two dimensions refer too a particular cell in each table (row, column). The cells are compared to each other and any cell where the content differs between the tow pages is identified and stored in the @cells array. The theory is that cells that contain the site's menus and other template type content will be ignored. The cells that contain content that changes from one page to the next will be recorded.

The third URL is used to test the content extraction theory. The HTML::LinkExtor first parses the contents of each changed cell. This finds the html links stored in the content. The content is then stripped of html and printed to the screen. Last of all the links found in the content is printed to the screen.

```
#!/usr/bin/Perl

use lib qw(. ..);
use HTML::TableExtract;
use LWP::Simple;
use String::CRC;
use Data::Dumper;
use HTML::LinkExtor;

# Data Entry Portion
print "Enter first URL: ";
my $url = <>;
chomp $url;
my $t1 = pageParse($url, 1, 0); #Parse out tables in first URL
print "Enter next URL:";
my $url2 = <>;
my $t2 = pageParse($url2, 1, 0); # Parse out tables in second
URL
my ($depth, $count, $row, $col);

# Loop through elements of array and find the cells that do
not
# have equivalent content
for ($depth=0;$depth< max(scalar(@$t1), scalar(@$t2));
$depth++)
{
for ($count=0;$count<
max(scalar(@{$t1->[$depth]}),scalar(@{$t2->[$depth]}));
```

```
$count++)
{
for ($row=0;
$row < max(scalar(@{$t1->[$depth][$count]}),
scalar(@{$t2->[$depth][$count]}));
$row++)
{
for ($col=0;
$col<
max(scalar(@{$t1->[$depth][$count][$row]}),scalar(@{$t2->[$depth][$count][
$col++)
{
if (defined $t2->[$depth][$count][$row][$col])
{
if ($t1->[$depth][$count][$row][$col] ne
$t2->[$depth][$count][$row][$col])
{
print " Cell $depth $count $row $col differs\n";
push @cells, [$depth, $count, $row, $col];
} #if ($t1->[$depth][$count][$row][$col] ne
$t2->[$depth][$count][$row][$col])
} # if (defined $t2->[$depth][$count][$row][$col])
} #for $col
} #for $row
} #for $count
} #for $depth

print "Enter URL You want to rip links from:";
$url = <>;
chomp $url;
my $tab = pageParse($url, 0, 1);
foreach my $coords (@cells)
{
my ($depth, $count, $row, $col) = @$coords;
my $linkParser = HTML::LinkExtor->new();
my $content = $tab->[$depth][$count][$row][$col];
$linkParser->parse($content);
$content =~ s/<.*?>//g;
my @links = $linkParser->links; # get Links

print "-----Depth $depth ; Count $count ; Row $row ; Col $col
\n";
print $content;
print "-----Links:\n";
foreach my $link (@links)
{
my $tag = shift @$link;
if ($tag eq 'a')
{
```

```
my %linkHash = @$link;
print $linkHash{href}, "\n"
}
}
print "-----END CONTENT\n";
}

#Parses HTML page and store resulting tables into a four
dimensional array.
sub pageParse
{
my $url=shift;
my $func = shift ;
my $keep_html = shift || 0;
my $te = new HTML::TableExtract( depth=>0, gridmap=>0,
keep_html=> $keep_html, br_translate=>1);
chomp $url;
my $content = get($url);
$te->parse($content);
my $tables=[];
foreach my $ts ($te->table_states()) # Loop through All tables
on page
{
my $row_idx =0;
foreach my $row ($ts->rows) # Loop through rows for a table
{
my $col_idx =0;
foreach my $column ( @$row) # Loop through columns in row.
{
if ( $func)
{
$column =~ s/\s//g;
my $crc= crc($column, 32); # Build checksum
$column = $crc;
}
else
{
$column =~ s/\s+/ /g;
}
$tables->[$ts->depth()][$ts->count()][$row_idx][$col_idx] =
$column;
$col_idx++;
}
$row_idx++;
}
}
return $tables;
}
```

```
sub max # returns max of two values
{
my ($x1, $x2) = @_;
return $x1 if ($x1 gt $x2);
return $x2;
}
```

Obviously, the previous script is not very practical However it could be modified and be very useful.

With a few changes you can create an automated personal newsletter. Instead of asking for three URLs the script could be modified to watch one particular site. After the first execution the generated hash from the pageParse subroutine could be stored off. The next time the script is run the new pageParse result could be compared to the original. If content is different in any of table cells the content in that cell could be emailed thus creating an automated newsletter.

Conclusion

These are but a few of the modules that you will find useful in building a Web Crawler. Other functionality available include; Text Processing: Lingua::Wordnet, Lingua::LinkParser XML: Xml::Parser, XML::Xalan, XML::RSS EMail:: MailTools bundle News Groups: News::Scan, News::Article just to name a few.

The flexibility of Perl and it's rich set of available modules is a perfect tool for developing web crawlers.