NAME

perlfilter - Source Filters

DESCRIPTION

This article is about a little-known feature of Perl called *source filters*. Source filters alter the program text of a module before Perl sees it, much as a C preprocessor alters the source text of a C program before the compiler sees it. This article tells you more about what source filters are, how they work, and how to write your own.

The original purpose of source filters was to let you encrypt your program source to prevent casual piracy. This isn't all they can do, as you'll soon learn. But first, the basics.

CONCEPTS

Before the Perl interpreter can execute a Perl script, it must first read it from a file into memory for parsing and compilation. If that script itself includes other scripts with a "use" or "require" statement, then each of those scripts will have to be read from their respective files as well.

Now think of each logical connection between the Perl parser and an individual file as a *source stream*. A source stream is created when the Perl parser opens a file, it continues to exist as the source code is read into memory, and it is destroyed when Perl is finished parsing the file. If the parser encounters a "require" or "use" statement in a source stream, a new and distinct stream is created just for that file.

The diagram below represents a single source stream, with the flow of source from a Perl script file on the left into the Perl parser on the right. This is how Perl normally operates.

```
file -------> parser
```

There are two important points to remember:

1. Although there can be any number of source streams in existence at any given time, only one will be active.

2. Every source stream is associated with only one file.

A source filter is a special kind of Perl module that intercepts and modifies a source stream before it reaches the parser. A source filter changes our diagram like this:

```
file ----> filter ----> parser
```

If that doesn't make much sense, consider the analogy of a command pipeline. Say you have a shell

script stored in the compressed file *trial.gz*. The simple pipeline command below runs the script without needing to create a temporary file to hold the uncompressed file.

```
gunzip -c trial.gz | sh
```

In this case, the data flow from the pipeline can be represented as follows:

```
trial.gz ----> gunzip ----> sh
```

With source filters, you can store the text of your script compressed and use a source filter to uncompress it for Perl's parser:

```
 compressed          gunzip
Perl program ---> source filter ---> parser
```

USING FILTERS

So how do you use a source filter in a Perl script? Above, I said that a source filter is just a special kind of module. Like all Perl modules, a source filter is invoked with a use statement.

Say you want to pass your Perl source through the C preprocessor before execution. As it happens, the source filters distribution comes with a C preprocessor filter module called Filter::cpp.

Below is an example program, "cpp_test", which makes use of this filter. Line numbers have been added to allow specific lines to be referenced easily.

```
1: use Filter::cpp;
2: #define TRUE 1
3: $a = TRUE;
4: print "a = $a\n";
```

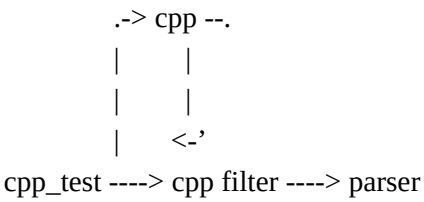
When you execute this script, Perl creates a source stream for the file. Before the parser processes any of the lines from the file, the source stream looks like this:

```
cpp_test ---------> parser
```

Line 1, "use Filter::cpp", includes and installs the "cpp" filter module. All source filters work this way. The use statement is compiled and executed at compile time, before any more of the file is read, and it attaches the cpp filter to the source stream behind the scenes. Now the data flow looks like this:

```
cpp_test ----> cpp filter ----> parser
```

As the parser reads the second and subsequent lines from the source stream, it feeds those lines through the "cpp" source filter before processing them. The "cpp" filter simply passes each line through the real C preprocessor. The output from the C preprocessor is then inserted back into the source stream by the filter.



The parser then sees the following code:

```
use Filter::cpp;
$a = 1;
print "a = $a\n";
```

Let's consider what happens when the filtered code includes another module with use:

```
1: use Filter::cpp;
2: #define TRUE 1
3: use Fred;
4: $a = TRUE;
5: print "a = $a\n";
```

The "cpp" filter does not apply to the text of the Fred module, only to the text of the file that used it ("cpp_test"). Although the use statement on line 3 will pass through the cpp filter, the module that gets included ("Fred") will not. The source streams look like this after line 3 has been parsed and before line 4 is parsed:

```
cpp_test ---> cpp filter ---> parser (INACTIVE)

Fred.pm ----> parser
```

As you can see, a new stream has been created for reading the source from "Fred.pm". This stream will remain active until all of "Fred.pm" has been parsed. The source stream for "cpp_test" will still exist, but is inactive. Once the parser has finished reading Fred.pm, the source stream associated with it will be destroyed. The source stream for "cpp_test" then becomes active again and the parser reads line 4 and subsequent lines from "cpp_test".

You can use more than one source filter on a single file. Similarly, you can reuse the same filter in as many files as you like.

For example, if you have a uuencoded and compressed source file, it is possible to stack a uudecode filter and an uncompression filter like this:

```
use Filter::uudecode; use Filter::uncompress;
M'XL(".H<US4''V9I;F%L')Q;>7/;1I;_>_I3=&E=%:F*I"T?22Q/
M6]9*<IQCO*XFT"0[PL%%'Y+IG?WN^ZYN-$'J.[.JE$,20/?K=_[>
...
```

Once the first line has been processed, the flow will look like this:

```
file ---> uudecode ---> uncompress ---> parser
          filter        filter
```

Data flows through filters in the same order they appear in the source file. The uudecode filter appeared before the uncompress filter, so the source file will be uudecoded before it's uncompressed.

WRITING A SOURCE FILTER

There are three ways to write your own source filter. You can write it in C, use an external program as a filter, or write the filter in Perl. I won't cover the first two in any great detail, so I'll get them out of the way first. Writing the filter in Perl is most convenient, so I'll devote the most space to it.

WRITING A SOURCE FILTER IN C

The first of the three available techniques is to write the filter completely in C. The external module you create interfaces directly with the source filter hooks provided by Perl.

The advantage of this technique is that you have complete control over the implementation of your filter. The big disadvantage is the increased complexity required to write the filter - not only do you need to understand the source filter hooks, but you also need a reasonable knowledge of Perl guts. One of the few times it is worth going to this trouble is when writing a source scrambler. The "decrypt" filter (which unscrambles the source before Perl parses it) included with the source filter distribution is an example of a C source filter (see Decryption Filters, below).

Decryption Filters

All decryption filters work on the principle of "security through obscurity." Regardless of how well you write a decryption filter and how strong your encryption algorithm is, anyone determined enough can retrieve the original source code. The reason is quite simple - once the decryption filter has decrypted the source back to its original form, fragments of it will be stored

in the computer's memory as Perl parses it. The source might only be in memory for a short period of time, but anyone possessing a debugger, skill, and lots of patience can eventually reconstruct your program.

That said, there are a number of steps that can be taken to make life difficult for the potential cracker. The most important: Write your decryption filter in C and statically link the decryption module into the Perl binary. For further tips to make life difficult for the potential cracker, see the file *decrypt.pm* in the source filters distribution.

CREATING A SOURCE FILTER AS A SEPARATE EXECUTABLE

An alternative to writing the filter in C is to create a separate executable in the language of your choice. The separate executable reads from standard input, does whatever processing is necessary, and writes the filtered data to standard output. "Filter::cpp" is an example of a source filter implemented as a separate executable - the executable is the C preprocessor bundled with your C compiler.

The source filter distribution includes two modules that simplify this task: "Filter::exec" and "Filter::sh". Both allow you to run any external executable. Both use a coprocess to control the flow of data into and out of the external executable. (For details on coprocesses, see Stephens, W.R., "Advanced Programming in the UNIX Environment." Addison-Wesley, ISBN 0-210-56317-7, pages 441-445.) The difference between them is that "Filter::exec" spawns the external command directly, while "Filter::sh" spawns a shell to execute the external command. (Unix uses the Bourne shell; NT uses the cmd shell.) Spawning a shell allows you to make use of the shell metacharacters and redirection facilities.

Here is an example script that uses "Filter::sh":

```
use Filter::sh 'tr XYZ PQR';
$a = 1;
print "XYZ a = $a\n";
```

The output you'll get when the script is executed:

```
PQR a = 1
```

Writing a source filter as a separate executable works fine, but a small performance penalty is incurred. For example, if you execute the small example above, a separate subprocess will be created to run the Unix "tr" command. Each use of the filter requires its own subprocess. If creating subprocesses is expensive on your system, you might want to consider one of the other options for creating source filters.

WRITING A SOURCE FILTER IN PERL

The easiest and most portable option available for creating your own source filter is to write it completely in Perl. To distinguish this from the previous two techniques, I'll call it a Perl source filter.

To help understand how to write a Perl source filter we need an example to study. Here is a complete source filter that performs rot13 decoding. (Rot13 is a very simple encryption scheme used in Usenet postings to hide the contents of offensive posts. It moves every letter forward thirteen places, so that A becomes N, B becomes O, and Z becomes M.)

```
package Rot13;

use Filter::Util::Call;

sub import {
  my ($type) = @_;
  my ($ref) = [];
  filter_add(bless $ref);
}

sub filter {
  my ($self) = @_;
  my ($status);

  tr/n-za-mN-ZA-M/a-zA-Z/
    if ($status = filter_read()) > 0;
  $status;
}

1;
```

All Perl source filters are implemented as Perl classes and have the same basic structure as the example above.

First, we include the "Filter::Util::Call" module, which exports a number of functions into your filter's namespace. The filter shown above uses two of these functions, "filter_add()" and "filter_read()".

Next, we create the filter object and associate it with the source stream by defining the "import" function. If you know Perl well enough, you know that "import" is called automatically every time a module is included with a use statement. This makes "import" the ideal place to both create and install a filter object.

In the example filter, the object ($ref) is blessed just like any other Perl object. Our example uses an anonymous array, but this isn't a requirement. Because this example doesn't need to store any context information, we could have used a scalar or hash reference just as well. The next section demonstrates context data.

The association between the filter object and the source stream is made with the "filter_add()" function. This takes a filter object as a parameter ($ref in this case) and installs it in the source stream.

Finally, there is the code that actually does the filtering. For this type of Perl source filter, all the filtering is done in a method called "filter()". (It is also possible to write a Perl source filter using a closure. See the "Filter::Util::Call" manual page for more details.) It's called every time the Perl parser needs another line of source to process. The "filter()" method, in turn, reads lines from the source stream using the "filter_read()" function.

If a line was available from the source stream, "filter_read()" returns a status value greater than zero and appends the line to $_. A status value of zero indicates end-of-file, less than zero means an error. The filter function itself is expected to return its status in the same way, and put the filtered line it wants written to the source stream in $_. The use of $_ accounts for the brevity of most Perl source filters.

In order to make use of the rot13 filter we need some way of encoding the source file in rot13 format. The script below, "mkrot13", does just that.

```
die "usage mkrot13 filename\n" unless @ARGV;
my $in = $ARGV[0];
my $out = "$in.tmp";
open(IN, "<$in") or die "Cannot open file $in: $!\n";
open(OUT, ">$out") or die "Cannot open file $out: $!\n";

print OUT "use Rot13;\n";
while (<IN>) {
  tr/a-zA-Z/n-za-mN-ZA-M/;
  print OUT;
}

close IN;
close OUT;
unlink $in;
rename $out, $in;
```

If we encrypt this with "mkrot13":

```
print " hello fred \n";
```

the result will be this:

```
use Rot13;
cevag "uryyb serq\a";
```

Running it produces this output:

```
hello fred
```

USING CONTEXT: THE DEBUG FILTER

The rot13 example was a trivial example. Here’s another demonstration that shows off a few more features.

Say you wanted to include a lot of debugging code in your Perl script during development, but you didn’t want it available in the released product. Source filters offer a solution. In order to keep the example simple, let’s say you wanted the debugging output to be controlled by an environment variable, "DEBUG". Debugging code is enabled if the variable exists, otherwise it is disabled.

Two special marker lines will bracket debugging code, like this:

```
## DEBUG_BEGIN
if ($year > 1999) {
  warn "Debug: millennium bug in year $year\n";
}
## DEBUG_END
```

The filter ensures that Perl parses the code between the <DEBUG_BEGIN> and "DEBUG_END" markers only when the "DEBUG" environment variable exists. That means that when "DEBUG" does exist, the code above should be passed through the filter unchanged. The marker lines can also be passed through as-is, because the Perl parser will see them as comment lines. When "DEBUG" isn’t set, we need a way to disable the debug code. A simple way to achieve that is to convert the lines between the two markers into comments:

```
## DEBUG_BEGIN
#if ($year > 1999) {
#     warn "Debug: millennium bug in year $year\n";
```

```
    #}
    ## DEBUG_END
```

Here is the complete Debug filter:

```
    package Debug;

    use strict;
    use warnings;
    use Filter::Util::Call;

    use constant TRUE => 1;
    use constant FALSE => 0;

    sub import {
      my ($type) = @_;
      my (%context) = (
        Enabled => defined $ENV{DEBUG},
        InTraceBlock => FALSE,
        Filename => (caller)[1],
        LineNo => 0,
        LastBegin => 0,
      );
      filter_add(bless \%context);
    }

    sub Die {
      my ($self) = shift;
      my ($message) = shift;
      my ($line_no) = shift || $self->{LastBegin};
      die "$message at $self->{Filename} line $line_no.\n"
    }

    sub filter {
      my ($self) = @_;
      my ($status);
      $status = filter_read();
      ++ $self->{LineNo};

      # deal with EOF/error first
```

```
    if ($status <= 0) {
        $self->Die("DEBUG_BEGIN has no DEBUG_END")
            if $self->{InTraceBlock};
        return $status;
    }

    if ($self->{InTraceBlock}) {
      if (/^\s*##\s*DEBUG_BEGIN/ ) {
          $self->Die("Nested DEBUG_BEGIN", $self->{LineNo})
      } elsif (/^\s*##\s*DEBUG_END/) {
          $self->{InTraceBlock} = FALSE;
      }

      # comment out the debug lines when the filter is disabled
      s/^/#/ if ! $self->{Enabled};
    } elsif ( /^\s*##\s*DEBUG_BEGIN/ ) {
      $self->{InTraceBlock} = TRUE;
      $self->{LastBegin} = $self->{LineNo};
    } elsif ( /^\s*##\s*DEBUG_END/ ) {
      $self->Die("DEBUG_END has no DEBUG_BEGIN", $self->{LineNo});
    }
    return $status;
}

1;
```

The big difference between this filter and the previous example is the use of context data in the filter object. The filter object is based on a hash reference, and is used to keep various pieces of context information between calls to the filter function. All but two of the hash fields are used for error reporting. The first of those two, Enabled, is used by the filter to determine whether the debugging code should be given to the Perl parser. The second, InTraceBlock, is true when the filter has encountered a "DEBUG_BEGIN" line, but has not yet encountered the following "DEBUG_END" line.

If you ignore all the error checking that most of the code does, the essence of the filter is as follows:

```
sub filter {
    my ($self) = @_;
    my ($status);
    $status = filter_read();
```

```
    # deal with EOF/error first
    return $status if $status <= 0;
    if ($self->{InTraceBlock}) {
      if (/^\s*##\s*DEBUG_END/) {
        $self->{InTraceBlock} = FALSE
      }

      # comment out debug lines when the filter is disabled
      s/^/#/ if ! $self->{Enabled};
    } elsif ( /^\s*##\s*DEBUG_BEGIN/ ) {
      $self->{InTraceBlock} = TRUE;
    }
    return $status;
  }
```

Be warned: just as the C-preprocessor doesn't know C, the Debug filter doesn't know Perl. It can be fooled quite easily:

```
  print <<EOM;
  ##DEBUG_BEGIN
  EOM
```

Such things aside, you can see that a lot can be achieved with a modest amount of code.

CONCLUSION

You now have better understanding of what a source filter is, and you might even have a possible use for them. If you feel like playing with source filters but need a bit of inspiration, here are some extra features you could add to the Debug filter.

First, an easy one. Rather than having debugging code that is all-or-nothing, it would be much more useful to be able to control which specific blocks of debugging code get included. Try extending the syntax for debug blocks to allow each to be identified. The contents of the "DEBUG" environment variable can then be used to control which blocks get included.

Once you can identify individual blocks, try allowing them to be nested. That isn't difficult either.

Here is an interesting idea that doesn't involve the Debug filter. Currently Perl subroutines have fairly limited support for formal parameter lists. You can specify the number of parameters and their type, but you still have to manually take them out of the @_ array yourself. Write a source filter that allows you to have a named parameter list. Such a filter would turn this:

```
sub MySub ($first, $second, @rest) { ... }
```

into this:

```
sub MySub($$@) {
  my ($first) = shift;
  my ($second) = shift;
  my (@rest) = @_;
  ...
}
```

Finally, if you feel like a real challenge, have a go at writing a full-blown Perl macro preprocessor as a source filter. Borrow the useful features from the C preprocessor and any other macro processors you know. The tricky bit will be choosing how much knowledge of Perl's syntax you want your filter to have.

LIMITATIONS

Source filters only work on the string level, thus are highly limited in its ability to change source code on the fly. It cannot detect comments, quoted strings, heredocs, it is no replacement for a real parser. The only stable usage for source filters are encryption, compression, or the byteloader, to translate binary code back to source code.

See for example the limitations in Switch, which uses source filters, and thus is does not work inside a string eval, the presence of regexes with embedded newlines that are specified with raw "/.../" delimiters and don't have a modifier "//x" are indistinguishable from code chunks beginning with the division operator "/". As a workaround you must use "m/.../" or "m?...?" for such patterns. Also, the presence of regexes specified with raw "?...?" delimiters may cause mysterious errors. The workaround is to use "m?...?" instead. See <https://metacpan.org/pod/Switch#LIMITATIONS>.

Currently the content of the "__DATA__" block is not filtered.

Currently internal buffer lengths are limited to 32-bit only.

THINGS TO LOOK OUT FOR

Some Filters Clobber the "DATA" Handle

Some source filters use the "DATA" handle to read the calling program. When using these source filters you cannot rely on this handle, nor expect any particular kind of behavior when operating on it. Filters based on Filter::Util::Call (and therefore Filter::Simple) do not alter the "DATA" filehandle, but on the other hand totally ignore the text after "__DATA__".

REQUIREMENTS

The Source Filters distribution is available on CPAN, in

    CPAN/modules/by-module/Filter

Starting from Perl 5.8 Filter::Util::Call (the core part of the Source Filters distribution) is part of the standard Perl distribution. Also included is a friendlier interface called Filter::Simple, by Damian Conway.

AUTHOR

Paul Marquess <Paul.Marquess@btinternet.com>

Reini Urban <rurban@cpan.org>

Copyrights