NAME

Filter::Simple - Simplified source filtering

SYNOPSIS

```
 # in MyFilter.pm:

     package MyFilter;

     use Filter::Simple;

     FILTER { ... };

     # or just:
     #
     # use Filter::Simple sub { ... };

 # in user's code:

     use MyFilter;

     # this code is filtered

     no MyFilter;

     # this code is not
```

DESCRIPTION

The Problem

Source filtering is an immensely powerful feature of recent versions of Perl. It allows one to extend the language itself (e.g. the Switch module), to simplify the language (e.g. Language::Pythonesque), or to completely recast the language (e.g. Lingua::Romana::Perligata). Effectively, it allows one to use the full power of Perl as its own, recursively applied, macro language.

The excellent Filter::Util::Call module (by Paul Marquess) provides a usable Perl interface to source filtering, but it is often too powerful and not nearly as simple as it could be.

To use the module it is necessary to do the following:

1. Download, build, and install the Filter::Util::Call module. (If you have Perl 5.7.1 or later, this is already done for you.)
2. Set up a module that does a `use Filter::Util::Call`.
3. Within that module, create an `import` subroutine.
4. Within the `import` subroutine do a call to `filter_add`, passing it either a subroutine reference.
5. Within the subroutine reference, call `filter_read` or `filter_read_exact` to "prime" $_ with source code data from the source file that will `use` your module. Check the status value returned to see if any source code was actually read in.
6. Process the contents of $_ to change the source code in the desired manner.
7. Return the status value.

8. If the act of unimporting your module (via a `no`) should cause source code filtering to cease, create an `unimport` subroutine, and have it call `filter_del`. Make sure that the call to `filter_read` or `filter_read_exact` in step 5 will not accidentally read past the `no`. Effectively this limits source code filters to line-by-line operation, unless the `import` subroutine does some fancy pre-pre-parsing of the source code it's filtering.

For example, here is a minimal source code filter in a module named BANG.pm. It simply converts every occurrence of the sequence `BANG\s+BANG` to the sequence `die 'BANG' if $BANG` in any piece of code following a `use BANG;` statement (until the next `no BANG;` statement, if any):

```
package BANG;

use Filter::Util::Call ;

sub import {
    filter_add( sub {
    my $caller = caller;
    my ($status, $no_seen, $data);
    while ($status = filter_read()) {
        if (/^\s*no\s+$caller\s*;\s*?$/) {
            $no_seen=1;
            last;
        }
        $data .= $_;
        $_ = "";
    }
    $_ = $data;
    s/BANG\s+BANG/die 'BANG' if \$BANG/g
        unless $status < 0;
    $_ .= "no $class;\n" if $no_seen;
    return 1;
    })
}

sub unimport {
    filter_del();
}

1 ;
```

This level of sophistication puts filtering out of the reach of many programmers.

A Solution

The Filter::Simple module provides a simplified interface to Filter::Util::Call; one that is sufficient for most common cases.

Instead of the above process, with Filter::Simple the task of setting up a source code filter is reduced to:

1. Download and install the Filter::Simple module. (If you have Perl 5.7.1 or later, this is already done for you.)

2. Set up a module that does a `use Filter::Simple` and then calls `FILTER { ... }`.

3. Within the anonymous subroutine or block that is passed to `FILTER`, process the contents of

$_ to change the source code in the desired manner.

In other words, the previous example, would become:

```
package BANG;
use Filter::Simple;

FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
};

1 ;
```

Note that the source code is passed as a single string, so any regex that uses ^ or $ to detect line boundaries will need the /m flag.

Disabling or changing <no> behaviour

By default, the installed filter only filters up to a line consisting of one of the three standard source "terminators":

```
no ModuleName;  # optional comment
```

or:

```
__END__
```

or:

```
__DATA__
```

but this can be altered by passing a second argument to `use Filter::Simple` or FILTER (just remember: there's *no* comma after the initial block when you use FILTER).

That second argument may be either a qr'd regular expression (which is then used to match the terminator line), or a defined false value (which indicates that no terminator line should be looked for), or a reference to a hash (in which case the terminator is the value associated with the key 'terminator'.

For example, to cause the previous filter to filter only up to a line of the form:

```
GNAB esu;
```

you would write:

```
package BANG;
use Filter::Simple;

FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
}
qr/^\s*GNAB\s+esu\s*;\s*?$/;
```

or:

```
FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
```

```
    }
    { terminator => qr/^\s*GNAB\s+esu\s*;\s*?$/ };
```

and to prevent the filter's being turned off in any way:

```
    package BANG;
    use Filter::Simple;

    FILTER {
        s/BANG\s+BANG/die 'BANG' if \$BANG/g;
    }
    "";    # or: 0
```

or:

```
    FILTER {
        s/BANG\s+BANG/die 'BANG' if \$BANG/g;
    }
    { terminator => "" };
```

Note that, no matter what you set the terminator pattern to, the actual terminator itself *must* be contained on a single source line.

All-in-one interface

Separating the loading of Filter::Simple:

```
    use Filter::Simple;
```

from the setting up of the filtering:

```
    FILTER { ... };
```

is useful because it allows other code (typically parser support code or caching variables) to be defined before the filter is invoked. However, there is often no need for such a separation.

In those cases, it is easier to just append the filtering subroutine and any terminator specification directly to the `use` statement that loads Filter::Simple, like so:

```
    use Filter::Simple sub {
        s/BANG\s+BANG/die 'BANG' if \$BANG/g;
    };
```

This is exactly the same as:

```
    use Filter::Simple;
    BEGIN {
        Filter::Simple::FILTER {
            s/BANG\s+BANG/die 'BANG' if \$BANG/g;
        };
    }
```

except that the `FILTER` subroutine is not exported by Filter::Simple.

Filtering only specific components of source code

One of the problems with a filter like:

```
    use Filter::Simple;

    FILTER { s/BANG\s+BANG/die 'BANG' if \$BANG/g };
```

is that it indiscriminately applies the specified transformation to the entire text of your source program. So something like:

```
    warn 'BANG BANG, YOU'RE DEAD';
    BANG BANG;
```

will become:

```
    warn 'die 'BANG' if $BANG, YOU'RE DEAD';
    die 'BANG' if $BANG;
```

It is very common when filtering source to only want to apply the filter to the non-character-string parts of the code, or alternatively to *only* the character strings.

Filter::Simple supports this type of filtering by automatically exporting the `FILTER_ONLY` subroutine.

`FILTER_ONLY` takes a sequence of specifiers that install separate (and possibly multiple) filters that act on only parts of the source code. For example:

```
    use Filter::Simple;

    FILTER_ONLY
        code      => sub { s/BANG\s+BANG/die 'BANG' if \$BANG/g },
        quotelike => sub { s/BANG\s+BANG/CHITTY CHITTY/g };
```

The `"code"` subroutine will only be used to filter parts of the source code that are not quotelikes, POD, or `__DATA__`. The `quotelike` subroutine only filters Perl quotelikes (including here documents).

The full list of alternatives is:

`"code"`

Filters only those sections of the source code that are not quotelikes, POD, or `__DATA__`.

`"code_no_comments"`

Filters only those sections of the source code that are not quotelikes, POD, comments, or `__DATA__`.

`"executable"`

Filters only those sections of the source code that are not POD or `__DATA__`.

`"executable_no_comments"`

Filters only those sections of the source code that are not POD, comments, or `__DATA__`.

`"quotelike"`

Filters only Perl quotelikes (as interpreted by `&Text::Balanced::extract_quotelike`).

`"string"`

Filters only the string literal parts of a Perl quotelike (i.e. the contents of a string literal, either half of a `tr///`, the second half of an `s///`).

`"regex"`

Filters only the pattern literal parts of a Perl quotelike (i.e. the contents of a `qr//` or an `m//`, the first half of an `s///`).

`"all"`

Filters everything. Identical in effect to `FILTER`.

Except for `FILTER_ONLY code => sub {...}`, each of the component filters is called repeatedly, once for each component found in the source code.

Note that you can also apply two or more of the same type of filter in a single `FILTER_ONLY`. For example, here's a simple macro-preprocessor that is only applied within regexes, with a final debugging pass that prints the resulting source code:

```
use Regexp::Common;
FILTER_ONLY
    regex => sub { s/!\[/[^/g },
    regex => sub { s/%d/$RE{num}{int}/g },
    regex => sub { s/%f/$RE{num}{real}/g },
    all   => sub { print if $::DEBUG };
```

Filtering only the code parts of source code

Most source code ceases to be grammatically correct when it is broken up into the pieces between string literals and regexes. So the `'code'` and `'code_no_comments'` component filter behave slightly differently from the other partial filters described in the previous section.

Rather than calling the specified processor on each individual piece of code (i.e. on the bits between quotelikes), the `'code...'` partial filters operate on the entire source code, but with the quotelike bits (and, in the case of `'code_no_comments'`, the comments) "blanked out".

That is, a `'code...'` filter *replaces* each quoted string, quotelike, regex, POD, and __DATA__ section with a placeholder. The delimiters of this placeholder are the contents of the `$;` variable at the time the filter is applied (normally `"\034"`). The remaining four bytes are a unique identifier for the component being replaced.

This approach makes it comparatively easy to write code preprocessors without worrying about the form or contents of strings, regexes, etc.

For convenience, during a `'code...'` filtering operation, Filter::Simple provides a package variable (`$Filter::Simple::placeholder`) that contains a pre-compiled regex that matches any placeholder...and captures the identifier within the placeholder. Placeholders can be moved and re-ordered within the source code as needed.

In addition, a second package variable (`@Filter::Simple::components`) contains a list of the various pieces of `$_`, as they were originally split up to allow placeholders to be inserted.

Once the filtering has been applied, the original strings, regexes, POD, etc. are re-inserted into the code, by replacing each placeholder with the corresponding original component (from `@components`). Note that this means that the `@components` variable must be treated with extreme care within the filter. The `@components` array stores the "back- translations" of each placeholder inserted into `$_`, as well as the interstitial source code between placeholders. If the placeholder backtranslations are altered in `@components`, they will be similarly changed when the placeholders are removed from `$_` after the filter is complete.

For example, the following filter detects concatentated pairs of strings/quotelikes and reverses the order in which they are concatenated:

```
package DemoRevCat;
use Filter::Simple;
```

```
    FILTER_ONLY code => sub {
        my $ph = $Filter::Simple::placeholder;
        s{ ($ph) \s* [.] \s* ($ph) }{ $2.$1 }gx
    };
```

Thus, the following code:

```
    use DemoRevCat;

    my $str = "abc" . q(def);

    print "$str\n";
```

would become:

```
    my $str = q(def)."abc";

    print "$str\n";
```

and hence print:

```
    defabc
```

Using Filter::Simple with an explicit import subroutine

Filter::Simple generates a special `import` subroutine for your module (see *How it works*) which would normally replace any `import` subroutine you might have explicitly declared.

However, Filter::Simple is smart enough to notice your existing `import` and Do The Right Thing with it. That is, if you explicitly define an `import` subroutine in a package that's using Filter::Simple, that `import` subroutine will still be invoked immediately after any filter you install.

The only thing you have to remember is that the `import` subroutine *must* be declared *before* the filter is installed. If you use `FILTER` to install the filter:

```
    package Filter::TurnItUpTo11;

    use Filter::Simple;

    FILTER { s/(\w+)/\U$1/ };
```

that will almost never be a problem, but if you install a filtering subroutine by passing it directly to the `use Filter::Simple` statement:

```
    package Filter::TurnItUpTo11;

    use Filter::Simple sub{ s/(\w+)/\U$1/ };
```

then you must make sure that your `import` subroutine appears before that `use` statement.

Using Filter::Simple and Exporter together

Likewise, Filter::Simple is also smart enough to Do The Right Thing if you use Exporter:

```
    package Switch;
    use base Exporter;
```

```
use Filter::Simple;

@EXPORT    = qw(switch case);
@EXPORT_OK = qw(given  when);

FILTER { $_ = magic_Perl_filter($_) }
```

Immediately after the filter has been applied to the source, Filter::Simple will pass control to Exporter, so it can do its magic too.

Of course, here too, Filter::Simple has to know you're using Exporter before it applies the filter. That's almost never a problem, but if you're nervous about it, you can guarantee that things will work correctly by ensuring that your `use base Exporter` always precedes your `use Filter::Simple`.

How it works

The Filter::Simple module exports into the package that calls `FILTER` (or uses it directly) -- such as package "BANG" in the above example -- two automagically constructed subroutines -- `import` and `unimport` -- which take care of all the nasty details.

In addition, the generated `import` subroutine passes its own argument list to the filtering subroutine, so the BANG.pm filter could easily be made parametric:

```
package BANG;

use Filter::Simple;

FILTER {
    my ($die_msg, $var_name) = @_;
    s/BANG\s+BANG/die '$die_msg' if \${$var_name}/g;
};

# and in some user code:

use BANG "BOOM", "BAM";  # "BANG BANG" becomes: die 'BOOM' if $BAM
```

The specified filtering subroutine is called every time a `use BANG` is encountered, and passed all the source code following that call, up to either the next `no BANG;` (or whatever terminator you've set) or the end of the source file, whichever occurs first. By default, any `no BANG;` call must appear by itself on a separate line, or it is ignored.

AUTHOR

Damian Conway (damian@conway.org)

COPYRIGHT