

Modern Perl

Monday July 1, 2024

Last year, I wrote an article detailing [my experience re-creating a Python script in Perl](#). This was a pretty cumbersome exercise, but I didn't use many of Perl's newer features. I decided to re-visit this, leveraging some of these newer features to see how well it improves the process.

Enabling Features

To take advantage of specific features, use the aptly named feature pragma.

You can enable an individual feature by name, and multiple features at once with `qw()`. For example, this:

```
use feature 'fc';  
use feature 'say';
```

Is the same as this:

```
use feature qw(fc say);
```

Full example:

```
use feature qw(fc say);  
  
$x = 'Test';  
  
say "The case-folded version of $x is: " . fc $x;
```

Output:

```
The case-folded version of Test is: test
```

Without `use feature`, you'd get this:

```
String found where operator expected (Do you need to predeclare "say"?) at  
./test.pl line 7, near "say "The case-folded version of $x is: ""  
syntax error at ./test.pl line 7, near "say "The case-folded version of $x  
is: ""  
Execution of ./test.pl aborted due to compilation errors.
```

You can also take advantage of **feature bundles**, enabling all of the features available as of a given version:

```
# implicitly loads 5.36 feature bundle
```

```
use v5.36;
```

But, you must also have (at least) that version of Perl installed, of course:

```
use v5.40;
```

```
Perl v5.40.0 required--this is only v5.38.2, stopped at ./test.pl line 3.
BEGIN failed--compilation aborted at ./test.pl line 3.
```

Useful Features

Lots of new features have been added in recent years, but I'll take advantage of only a small subset of them for this exercise. I'll be using features available in 5.38.2 and earlier.

Feature	Version	Notes
New <code>class</code> feature	5.38.0	Adds support for object-oriented code.
New <code>say</code> built-in	5.10.0	Works like <code>print</code> , adds a newline.
Subroutine signatures no longer considered experimental	5.36.0	Supports named parameters in subroutines.

Refactoring As Structured Code

We'll start with the first subroutine that's called.

Here's the original:

```
sub exec_backup_set {
    my @source_paths = @ { $_[0] };
    my $target_path = $_[1];

    foreach my $source_path (@source_paths) {
        exec_backup( $source_path, $target_path );
    }
}
```

And here's the new version, using the `signatures` feature to implement subroutine arguments as lexical variables:

```
sub exec_backup_set ( $target_path, @source_paths ) {
    foreach my $source_path (@source_paths) {
        exec_backup( $source_path, $target_path );
    }
}
```

You'll notice that using named arguments in the method signature is a lot cleaner. But, we have to swap the **source_paths** and **target_path** arguments. Why? Since **source_paths** is an array, it's a 'slurpy' argument, meaning it will reference all remaining arguments in the signature. So, if it were the first argument, it would also absorb the contents of **target_path**.

A "slurpy" parameter is a list or hash parameter that "slurps up" all remaining arguments. Since any following parameters can't receive values, there can be only one slurpy parameter.

Slurpy parameters must come at the end of the signature and they must be positional.

Slurpy parameters are optional by default.

<https://metacpan.org/pod/Method::Signatures#Slurpy-parameters>

Next, we'll refactor the `exec_backup` subroutine. Here's the original:

```
sub exec_backup {
    my ( $source, $target ) = @_;

    my $proc_name = "rsync -lrv --delete \"${source}\" \"${target}\"";

    unless ( -d $target ) {
        mkdir($target);
    }

    if ( -d $target ) {
        print("Syncing ${source}...\n");
        system($proc_name);
        print("Synced ${source}\n");
    }

    print("-----\n");
}
```

For our updated version, we'll use the `signatures` feature again, along with the `say` feature:

```
sub exec_backup_modern ( $source, $target ) {
    my $proc_name = "rsync -lrv --delete \"${source}\" \"${target}\"";

    unless ( -d $target ) {
        mkdir($target);
    }

    if ( -d $target ) {
        say("Syncing ${source}...");

        system($proc_name);

        say("Synced ${source}");
    }
}
```

```
say("-----");  
}
```

Now we have enough information to refactor our entire script. Here's the "modern" version of our [old script](#):

```
use strict;  
use warnings;  
  
use feature 'signatures';  
use feature 'say';  
  
sub exec_backup ( $source, $target ) {  
    my $proc_name = "rsync -lrtv --delete \"${source}\" \"${target}\"";  
  
    unless ( -d $target ) {  
        mkdir($target);  
    }  
  
    if ( -d $target ) {  
        say("Syncing ${source}...");  
  
        system($proc_name);  
  
        say("Synced ${source}");  
    }  
  
    say("-----");  
}  
  
sub exec_backup_set ( $target_path, @source_paths ) {  
    foreach my $source_path (@source_paths) {  
        exec_backup( $source_path, $target_path );  
    }  
}  
  
my @target_paths =  
    ( "/target1/", "/target2/" );  
  
# regular file sets  
my @regular_files = (  
    "/home/jimc/source1", "/home/jimc/source2",  
    "/home/jimc/source3", "/home/jimc/source4"  
);  
exec_backup_set( $target_paths[0], @regular_files );  
  
# large file sets  
my @large_files = ( "/home/jimc/large1", "/home/jimc/large2" );  
exec_backup_set( $target_paths[1], @large_files );
```

```
sleep(2); # pause before exiting
```

With these changes, we have a script that's more concise and readable.

Refactoring As Object-Oriented Code

We've improved the code, but what if we want to implement it using an object-oriented paradigm instead of structured? We can accomplish this using the `class` feature.

Let's start with a very simple example:

```
use strict;
use warnings;

use feature 'class';
use feature 'say';

class MyUtil::Backup {
    method say_hello {
        say("Hello!");
    }
}

my $backup_obj = MyUtil::Backup->new();
$backup_obj->say_hello;
```

If you're familiar with "old" Perl, this will look *very* unusual. We have a couple of new keywords we're using, `class` and `method`. In this example, we've defined a new class `Backup` in a namespace `MyUtil`. Inside the class, we have a single callable method named `say_hello`. We create an instance of the class named `$backup_obj` and then call the `say_hello` method. We run the code and...

```
class is experimental at ./fullsync_oop.pl line 9.
method is experimental at ./fullsync_oop.pl line 11.
Hello!
```

The code runs, but the interpreter is warning us about the experimental nature of the `class` and `method` keywords. We can suppress this with a `no warnings` directive. With our code now looking like this:

```
use strict;
use warnings;

use feature 'class';
use feature 'say';

no warnings 'experimental::class';
```

```
class MyUtil::Backup {
    method say_hello {
        say("Hello!");
    }
}

my $backup_obj = MyUtil::Backup->new();
$backup_obj->say_hello;
```

We see this:

```
Hello!
```

Now let's add a field to hold our name, and use it in our greeting:

```
use strict;
use warnings;

use feature 'class';
use feature 'say';

no warnings 'experimental::class';

class MyUtil::Backup {

    field $name : param;

    method say_hello {
        say("Hello, $name!");
    }
}

my $backup_obj = MyUtil::Backup->new( name => 'Jim' );
$backup_obj->say_hello;
```

We've added a field named `$name`. `param` indicates that the field value will be initialized in the constructor, which we call as `MyUtil::Backup->new(name => 'Jim')`.

When we run, we see this:

```
Hello, Jim!
```

Now we've learned enough to refactor our structured script, and make it object-oriented:

```
use strict;
use warnings;

use feature 'class';
```

```
use feature 'say';

no warnings 'experimental::class';

class MyUtil::Backup {

    method exec_backup ( $source, $target ) {
        my $proc_name = "rsync -lrv --delete \"${source}\" \"${target}\"";

        unless ( -d $target ) {
            mkdir($target);
        }

        if ( -d $target ) {
            say("Syncing ${source}...");

            system($proc_name);

            say("Synced ${source}");
        }

        say("-----");
    }

    method exec_backup_set ( $target_path, @source_paths ) {
        foreach my $source_path (@source_paths) {
            $self->exec_backup( $source_path, $target_path );
        }
    }
}

my $backup_obj = MyUtil::Backup->new();

my @target_paths =
    ( "/target1/", "/target2/" );

# regular file sets
my @regular_files = (
    "/home/jimc/source1", "/home/jimc/source2",
    "/home/jimc/source3", "/home/jimc/source4"
);

$backup_obj->exec_backup_set( $target_paths[0], @regular_files );

# large file sets
my @large_files = ( "/home/jimc/large1", "/home/jimc/large2" );

$backup_obj->exec_backup_set( $target_paths[1], @large_files );

sleep(2); # pause before exiting
```

This one's a bit more verbose, but still pretty clean and readable.



Note the syntax of the call to the `exec_backup` method from the `exec_backup_set` method: I had to prefix it with `$self` to indicate that a method of the current instance of the class is being called.

The Verdict

I still don't consider Perl to be the friendliest language in the world, but taking advantage of modern features definitely improves the experience!

Links To References

<https://perldoc.perl.org/feature>

<https://perldoc.perl.org/5.38.0/perlclass>

From:

<https://blog.devtoprd.com/> - **Jim's Blog**

Permanent link:

https://blog.devtoprd.com/doku.php?id=posts:2024:2024_07_01_modern_perl_update

Last update: **2025/03/31 04:58**

