NAME

Monkey::Patch::Action − Wrap/add/replace/delete subs from other package (with restore)

VERSION

version 0.04

SYNOPSIS

```
use Monkey::Patch::Action qw(patch_package);

package Foo;
sub sub1  { say "Foo's sub1" }
sub sub2  { say "Foo's sub2, args=", join(",", @_) }
sub meth1 { my $self = shift; say "Foo's meth1" }

package Bar;
our @ISA = qw(Foo);

package main;
my $h; # handle object
my $foo = Foo->new;
my $bar = Bar->new;

# replacing a subroutine
$h = patch_package('Foo', 'sub1', 'replace', sub { "qux" });
Foo::sub1(); # says "qux"
undef $h;
Foo::sub1(); # says "Foo's sub1"

# adding a subroutine
$h = patch_package('Foo', 'sub3', 'add', sub { "qux" });
Foo::sub3(); # says "qux"
undef $h;
Foo::sub3(); # dies

# deleting a subroutine
$h = patch_package('Foo', 'sub2', 'delete');
Foo::sub2(); # dies
undef $h;
Foo::sub2(); # says "Foo's sub2, args="

# wrapping a subroutine
$h = patch_package('Foo', 'sub2', 'wrap',
    sub {
        my $ctx = shift;
        say "wrapping $ctx->{package}::$ctx->{subname}";
        $ctx->{orig}->(@_);
    }
);
Foo::sub2(1,2,3); # says "wrapping Foo::sub2" then "Foo's sub2, args=1,2,3"
undef $h;
Foo::sub2(1,2,3); # says "Foo's sub2, args=1,2,3"

# stacking patches (note: can actually be unapplied in random order)
my ($h2, $h3);
$h  = patch_package('Foo', 'sub1', 'replace', sub { "qux" });
Foo::sub1(); # says "qux"
$h2 = patch_package('Foo', 'sub1', 'delete');
Foo::sub1(); # dies
$h3 = patch_package('Foo', 'sub1', 'replace', sub { "quux" });
Foo::sub1(); # says "quux"
```

```
undef $h3;
Foo::sub1(); # dies
undef $h2;
Foo::sub1(); # says "qux"
undef $h;
Foo::sub1(); # says "Foo's sub1"
```

DESCRIPTION

Monkey-patching is the act of modifying a package at runtime: adding a subroutine/method, replacing/deleting/wrapping another, etc. Perl makes it easy to do that, for example:

```
# add a subroutine
*{"Target::sub1"} = sub { ... };

# another way, can be done from any file
package Target;
sub sub2 { ... }

# delete a subroutine
undef *{"Target::sub3"};
```

This module makes things even easier by helping you apply a stack of patches and unapply them later in flexible order.

FUNCTIONS

patch_package($package, $subname, $action, $code, @extra) => HANDLE

Patch `$package`'s subroutine named `$subname`. `$action` is either:

- `wrap`

  `$subname` must already exist. `code` is required.

  Your code receives a context hash as its first argument, followed by any arguments the subroutine would have normally gotten. Context hash contains: `orig` (the original subroutine that is being wrapped), `subname`, `package`, `extra`.

- `add`

  `subname` must not already exist. `code` is required.

- `replace`

  `subname` must already exist. `code` is required.

- `add_or_replace`

  `code` is required.

- `delete`

  `code` is not needed.

Die on error.

Function returns a handle object. As soon as you lose the value of the handle (by calling in void context, assigning over the variable, undeffing the variable, letting it go out of scope, etc), the patch is unapplied.

Patches can be unapplied in random order, but unapplying a patch where the next patch is a wrapper can lead to an error. Example: first patch (P1) adds a subroutine and second patch (P2) wraps it. If P1 is unapplied before P2, the subroutine is now no longer there, and P2 no longer works. Unapplying P1 after P2 works, of course.

FAQ

Differences with Monkey::Patch?

This module is based on the wonderful Monkey::Patch by Paul Driver. The differences are:

- This module adds the ability to add/replace/delete subroutines instead of just wrapping them.
- Interface to *patch_package()* is slightly different (see previous item for the cause).

- Using this module, the wrapper receives a context hash instead of just the original subroutine.
- Monkey::Patch adds convenience for patching classes and objects. To keep things simple, no such convenience is currently provided by this module. `patch_package()` *can* patch classes and objects as well (see the next FAQ entry).

How to patch classes and objects?

Patching a class is basically the same as patching any other package, since Perl implements a class with a package. One thing to note is that to call a parent's method inside your wrapper code, instead of:

```
$self->SUPER::methname(...)
```

you need to do something like:

```
use SUPER;
SUPER::find_parent(ref($self), 'methname')->methname(...)
```

Patching an object is also basically patching a class/package, because Perl does not have per-object method like Ruby. But if you just want to provide a modified behavior for a certain object only, you can do something like:

```
patch_package($package, $methname, 'wrap',
sub {
    my $ctx = shift;
    my $self = shift;

    my $obj = $ctx->{extra}[0];
    no warnings 'numeric';
    if ($obj == $self) {
        # do stuff
    }
    $ctx->{orig}->(@_);
}, $obj);
```

SEE ALSO

Monkey::Patch

AUTHOR

Steven Haryanto <stevenharyanto AT gmail DOT com>

COPYRIGHT AND LICENSE