Object::Trampoline

When having not the object you want
is what you need.

Steven Lembark
<lembark@wrkhors.com>

Er... what's “trampoline” object?

- Trampolines not the object you want.
- They are a proxy for the constructor of another object – the one you want.
- Their behavior is replacing themselves when you call a method on them.
- Aside from calling a separate constructor, the user shouldn't know the trampoline ever existed.

Why bother?

- When you don't *want* an object until you *need* it:
  - Connections to servers that not always used/available (e.g., during development or unit testing).
  - Avoid construcing expensive, seldom-used objects.
  - Delay connections to back-end servers until necessary.
- Think of starting up a heavily-forked apache server and not bringing your database to its knees.
- Or not parsing really large XML until you use it.

WARNING:

The code you are about to see contains graphic AUTOLOAD, literal blessing, and re-assignment of stack variables.

Parenthetical discresion is advised.

How do you bounce an object?

- Easily, in Perl (pity the poor slobs using Java!).
  - Perl's AUTOLOAD mechanism allows you to intercept method calls cleanly.
  - Passing arguments by reference allows replacing them on the stack: assigning to $_[0] gives your caller a new object on the fly.
  - “goto &sub” replaces one call with another.
- Result: a re-dispatched call with a new object.

Co-Operating Classes

- The Object::Trampoline (“O::T”) module uses two classes: a constructor and dispatcher.
- O::T itself is nothing but an AUTOLOAD.
- It returns a closure blessed into Object::Trampoline::Bounce (“O::T::B”).
- O::T::B is nothing but (surprise!) an AUTOLOAD.
- O::T::B replaces the object, re-dispatches the call.

Replacing an Object

- O::T::B::AUTOLOAD begins by replacing the stack argument with the result of running itself:

```
$_[0] = $_[0]->();
```

  - This replaces the caller's copy of the object with a delayed call to the constructor.
  - This new object is then used to locate the requested subroutine via “can”.

Using Object::Trampoline

- The difference you'll see in using a trampoline object is in the constructor.
- The 'real' class becomes the first argument, and "Object::Trampoline" becomes the new class:

```
my $dbh = DBI->connect( $dsn,@argz);
```

becomes:

```
my $dbh = Object::Trampoline->connect
( 'DBI', $dsn, @argz );
```

Under the hood

- O::T's AUTOLOAD handles the construction by blessing a closure that does the real work:

```
my ( undef, $class, @argz ) = @_;

my $meth = ( split $AUTOLOAD, '::' )[-1];

my $sub  = sub { $class->$meth( @argz ) };

bless $sub, 'Object::Trampoline::Bounce'
```

Using the object

- At this point the caller gets back what looks like an ordinary object:

```
# $dbh starts out as a trampoline

my $dbh =
Object::Trampoline->connect( 'DBI', ... );

# the method call converts it to a DBI object.

my $sth = $dbh->prepare( ... );

# from this point on there's no way to tell
# that $dbh wasn't a DBI object all along.
```

Converting the Object

- The assignment to $_[0] is made in O::T::B::AUTOLOAD.
- If $_[0]->can($method) then it uses goto, otherwise it has to try $_[0]->$method(@argz) and hope for the best (e.g., another AUTOLOAD).
- It also has contains a stub DESTROY to avoid constructing objects when the go out of scope.

Object::Trampoline::Bounce

```
our $AUTOLOAD = '';
AUTOLOAD
{
    $_[0] = $_[0]->();
    my $class  = ref $_[0];
    my $method = ( split /::/, $AUTOLOAD )[ -1 ];
    if( my $sub = $_[0]->can( $method ) )
    {
        goto &$sub
    }
    else
    {
        my $obj = shift;
        $obj->$method( @_ )
    }
}

DESTROY {}
```

But wait, there's more!

- What if requiring the module is the expensive part?
- You want to delay the “use” until necessary, not just the construction?
- Object::Trampoline::Use does exactly that:

```
my sub
= sub
{
    eval “package $caller; use $class;

    $class->$method( @argz )
};
```

Why use a closure?

- I could have stored the arguments in a hash, with $object->{ class } and $object->{ arguments }.
- But then there would be a difference in handling different objects that came from O::T or O::T::U.
- The closure allows each handler class to handle the construction its own way without to specialize O::T::B for each of them.

Example: Server Handles

- Centralizing the data for your server handles can be helpful.
- All of the mess for describing DBI, Syslog, HTTP, SOAP... connections can be pushed into one place.
- Catch: Not all of the servers are always available, or necessary.
- Fix: Export trampolines.

Server::Handles

```
package Server::Handle;
use Server::Configure
qw
(
    dbi_user ...
    syslog_server ...
);
my %handlerz =
(
    dbh     =>
    Object::Trampoline->connect( 'DBI', ... ),

    syslogh =>
    Object::Trampoline->openlog( 'Syslog::Wrapper', ... ),
);
sub import
{
    # push the handlers out as-is via *$glob = \$value.
    # the values are shared and the first place they are
    # used bounces them for the entire process
}
```

Trampoline as a Factory

- This cannot be avoided, therefore it is a feature.
- Unwrapping the stack into a lexical before calling a method on the trampoline updates the lexical, not the caller's copy.

```
$foo->my_wrapper;

sub my_wrapper
{
    my $obj    = shift; # my_wrapper copy of $foo

    $obj->some_method;  # updates $obj, not $foo
```

Caveat Utilitor

- Trampoline objects can only dispatch methods.
- If your object is tied then it'll blow up if you try to access its tied interface:
  - $dbh->{ AutoCommit } = 0; # dies here for trampoline
- None of the ways around this are transparent to the user, but even with DBI the simple fix is to use methods to configure the object.

“ref” is not a method

- Until a method is called, “ref $object” will give you “Object::Trampoline::Bounce” and “reftype” will give you back “CODE”.
- This mainly affects the use of inside-out data, since $object_data{ refaddr $object } will change after the first method call.

Prototypes are Evil.

- Notice the closure:

```
$class->$constructor( @argz )
```

- Defining $constructor with a prototype of ($$) will break even if you have two values in @argz!

- <*soapbox*>
  Add code or use Class::Contract (whatever) to actually validate the arguments. Breaking Perl's calling convention only causes pain.
  </*soapbox*>.