NAME

Object::Realize::Later − Delayed creation of objects

SYNOPSIS

```
package MyLazyObject;

use Object::Realize::Later
   becomes => 'MyRealObject',
   realize => 'load';
```

DESCRIPTION

The `Object::Realize::Later` class helps with implementing transparent on demand realization of object data. This is related to the tricks on autoloading of data, the lesser known cousin of autoloading of functionality.

On demand realization is all about performance gain. Why should you spent costly time on realizing an object, when the data on the object is never (or not yet) used? In interactive programs, postponed realization may boost start-up: the realization of objects is triggered by the use, so spread over time.

METHODS

Construction

use(Object::Realize::Later `%options`)

When you invoke (`use`) the `Object::Realize::Later` package, it will add a set of methods to your package (see section ‘‘Added to YOUR class’’).

```
−Option            −−Default
 becomes             <required>
 believe_caller      <false>
 realize             <required>
 source_module       <becomes>
 warn_realization    <false>
 warn_realize_again  <false>
```

becomes => CLASS

Which type will this object become after realization.

believe_caller => BOOLEAN

When a method is called on the un-realized object, the AUTOLOAD checks whether this resolves the need. If not, the realization is not done. However, when realization may result in an object that extends the functionality of the class specified with `becomes`, this check must be disabled. In that case, specify true for this option.

realize => METHOD|CODE

How will transform. If you specify a CODE reference, then this will be called with the lazy-object as first argument, and the requested method as second.

After realization, you may still have your hands on the lazy object on various places. Be sure that your realization method is coping with that, for instance by using Memoize. See examples below.

source_module => CLASS

if the class (a package) is included in a file (module) with a different name, then use this argument to specify the file name. The name is expected to be the same as in the `require` call which would load it.

warn_realization => BOOLEAN

Print a warning message when the realization starts. This is for debugging purposes.

warn_realize_again => BOOLEAN

When an object is realized, the original object −which functioned as a stub− is reconstructed to work as proxy to the realized object. This option will issue a warning when that proxy is used, which means that somewhere in your program there is a variable still holding a reference to the stub.

This latter is not problematic at all, although it slows-down each method call.

Added to YOUR class

`$obj`−>**AUTOLOAD**()

When a method is called which is not available for the lazy object, the AUTOLOAD is called.

`$obj`−>**can**($method)

Object::Realize::Later−>**can**($method)

Is the specified `$method` available for the lazy or the realized version of this object? It will return the reference to the code.

example:

```
MyLazyObject->can('lazyWork')      # true
MyLazyObject->can('realWork')      # true

my $lazy = MyLazyObject->new;
$lazy->can('lazyWork');            # true
$lazy->can('realWork');            # true
```

`$obj`−>**forceRealize**()

You can force the load by calling this method on your object. It returns the realized object.

Object::Realize::Later−>**isa**($class)

Is this object a (sub−)class of the specified `$class` or can it become a (sub−)class of `$class`.

example:

```
MyLazyObject->isa('MyRealObject')      # true
MyLazyObject->isa('SuperClassOfLazy'); # true
MyLazyObject->isa('SuperClassOfReal'); # true

my $lazy = MyLazyObject->new;
$lazy->isa('MyRealObject');            # true
$lazy->isa('SuperClassOfLazy');        # true
$lazy->isa('SuperClassOfReal');        # true
```

`$obj`−>**willRealize**()

Returns which class will be the realized to follow-up this class.

Object::Realize::Later internals

The next methods are not exported to the class where the ‘use’ took place. These methods implement the actual realization.

Object::Realize::Later−>**import**(%options)

The `%options` used for `import` are the values after the class name with `use`. So this routine implements the actual option parsing. It generates code dynamically, which is then evaluated in the callers name-space.

Object::Realize::Later−>**realizationOf**(`$object`, [$realized])

Returns the `$realized` version of `$object`, optionally after setting it first. When the method returns `undef`, the realization has not yet taken place or the realized object has already been removed again.

Object::Realize::Later−>**realize**(%options)

This method is called when a `$object`−*forceRealize()*> takes place. It checks whether the realization has been done already (is which case the realized object is returned)

DETAILS

About lazy loading

There are two ways to implement lazy behaviour: you may choose to check whether you have realized the data in each method which accesses the data, or use the autoloading of data trick.

An implementation of the first solution is:

```
sub realize {
    my $self = shift;
    return $self unless $self->{_is_realized};

    # read the data from file, or whatever
    $self->{data} = ....;

    $self->{_is_realized} = 1;
    $self;
}

sub getData() {
    my $self = shift;
    return $self->realize->{data};
}
```

The above implementation is error-prone, where you can easily forget to call *realize()*. The tests cannot cover all ordenings of method-calls to detect the mistakes.

The *second approach* uses autoloading, and is supported by this package. First we create a stub-object, which will be transformable into a realized object later. This transformation is triggered by AUTOLOAD.

This stub-object may contain some methods from the realized object, to reduce the need for realization. The stub will also contain some information which is required for the creation of the real object.

`Object::Realize::Later` solves the inheritance problems (especially the *isa()* and *can()* methods) and supplies the AUTOLOAD method. Class methods which are not defined in the stub object are forwarded as class methods without realization.

Traps

Be aware of dangerous traps in the current implementation. These problems appear by having multiple references to the same delayed object. Depending on how the realization is implemented, terrible things can happen.

The two versions of realization:

- by reblessing

  This is the safe version. The realized object is the same object as the delayed one, but reblessed in a different package. When multiple references to the delayed object exists, they will all be updated at the same, because the bless information is stored within the refered variable.

- by new instance

  This is the nicest way of realization, but also quite more dangerous. Consider this:

```
package Delayed;
use Object::Realize::Later
     becomes => 'Realized',
     realize => 'load';

sub new($)      {my($class,$v)=@_; bless {label=>$v}, $class}
sub setLabel($) {my $self = shift; $self->{label} = shift}
sub load()      {$_[0] = Realized->new($_[0]->{label}) }

package Realized;  # file Realized.pm or use use(source_module)
sub new($)      {my($class,$v)=@_; bless {label=>$v}, $class}
sub setLabel($) {my $self = shift; $self->{label} = shift}
sub getLabel()  {my $self = shift; $self->{label}}
```

```
    package main;
    my $original = Delayed->new('original');
    my $copy     = $original;
    print $original->getLabel;     # prints 'original'
    print ref $original;           # prints 'Realized'
    print ref $copy;               # prints 'Delayed'
    $original->setLabel('changed');
    print $original->getLabel;     # prints 'changed'
    print $copy->getLabel;         # prints 'original'
```

Examples

Example 1

In the first example, we delay-load a message. On the moment the message is defined, we only take the location. When the data of the message is taken (header or body), the data is autoloaded.

```
package Mail::Message::Delayed;

use Object::Realize::Later
  ( becomes => 'Mail::Message::Real'
  , realize => 'loadMessage'
  );

sub new($) {
    my ($class, $file) = @_;
    bless { filename => $file }, $class;
}

sub loadMessage() {
    my $self = shift;
    Mail::Message::Real->new($self->{filename});
}
```

In the main program:

```
package main;
use Mail::Message::Delayed;

my $msg    = Mail::Message::Delayed->new('/home/user/mh/1');
$msg->body->print;     # this will trigger autoload.
```

Example 2

Your realization may also be done by reblessing. In that case to change the type of your object into a different type which stores the same information. Is that right? Are you sure? For simple cases, this may be possible:

```
package Alive;
use Object::Realize::Later
     becomes => 'Dead',
     realize => 'kill';

sub new()         {my $class = shift; bless {@_}, $class}
sub jump()        {print "Jump!\n"}
sub showAntlers() {print "Fight!\n"}
sub kill()        {bless(shift, 'Dead')}

package Dead;
```

```
sub takeAntlers() {...}
```

In the main program:

```
my $deer   = Alive->new(Animal => 'deer');
my $trophy = $deer->takeAntlers();
```

In this situation, the object (reference) is not changed but is *reblessed*. There is no danger that the unrealized version of the object is kept somewhere: all variable which know about this partical *deer* see the change.

Example 3

This module is especially useful for larger projects, which there is a need for speed or memory reduction. In this case, you may have an extra overview on which objects have been realized (transformed), and which not. This example is taken from the MailBox modules:

The Mail::Box module tries to boost the access-time to a folder. If you only need the messages of the last day, why shall all be read? So, MailBox only creates an invertory of messages at first. It takes the headers of all messages, but leaves the body (content) of the message in the file.

In MailBox' case, the Mail::Message-object has the choice between a number of Mail::Message::Body's, one of which has only be prepared to read the body when needed. A code snippet:

```
package Mail::Message;
sub new($$)
{   my ($class, $head, $body) = @_;
    my $self = bless {head => $head, body => $body}, $class;
    $body->message($self);          # tell body about the message
}
sub head()     { shift->{head} }
sub body()     { shift->{body} }

sub loadBody()
{   my $self = shift;
    my $body = $self->body;

    # Catch re-invocations of the loading.  If anywhere was still
    # a reference to the old (unrealized) body of this message, we
    # return the new-one directly.
    return $body unless $body->can('forceRealize');

    # Load the body (change it to anything which really is of
    # the promised type, or a sub-class of it.
    my ($lines, $size) = .......;    # get the data
    $self->{body} = Mail::Message::Body::Lines
                        ->new($lines, $size, $self);

    # Return the realized object.
    return $self->{body};
}

package Mail::Message::Body::Lines;
use base 'Mail::Message::Body';

sub new($$$)
{   my ($class, $lines, $size, $message) = @_;
    bless { lines => $lines, size => $size
          , message => $message }, $class;
```

```
}
sub size()    { shift->{size} }
sub lines()   { shift->{lines} }
sub message() { shift->{message);

package Mail::Message::Body::Delayed;
use Object::Realize::Later
    becomes => 'Mail::Message::Body',
    realize => sub {shift->message->loadBody};

sub new($)
{   my ($class, $size) = @_;
    bless {size => $size}, $class;
}
sub size() { shift->{size} }
sub message(;$)
{   my $self = shift;
    @_ ? ($self->{message} = shift) : $self->{messages};
}

package main;
use Mail::Message;
use Mail::Message::Body::Delayed;

my $body    = Mail::Message::Body::Delayed->new(42);
my $message = Mail::Message->new($head, $body);

print $message->size;         # will not trigger realization!
print $message->can('lines'); # true, but no realization yet.
print $message->lines;        # realizes automatically.
```

SEE ALSO

This module is part of Object-Realize-Later distribution version 0.21, built on January 24, 2018. Website: *http://perl.overmeer.net/CPAN/*

LICENSE