DAVID BLANK-EDELMAN

practical Perl tools: tie me up, tie me down (part 2)

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the book *Perl for System Administration* (O'Reilly, 2000). He has spent the past 20 years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and is one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

WHEN WE LEFT OFF IN THE PREVIOUS column, I was standing over an anonymous hash holding a whip. OK, maybe not, but it did get you to check your back issues of *;login:*, no? Actually, we left off with something much more titillating: the ability to modify the fundamental nature of how Perl variables work using modules based on Perl's tie() functionality.

At the end of the last column we had just begun to contemplate the following list of things we wished a Perl hash could do:

- have elements that would automatically expire after a certain amount of time had elapsed
- keep a history of all changes made to it over time
- restrict the updates that are possible
- always keep track of the top *N* values or the rank of the values stored in it
- always return the keys in a sorted order based on the values in that hash
- transparently encrypt and decrypt itself
- easily store and retrieve multiple values per key

Let's take some time to make those wishes (and more that you didn't even know you had) come true, and then we'll end by discussing how to create our own tie()-based code.

Expiring Hashes

Hashes with entries that disappear after a certain time period are easy to construct, thanks to Tie::Hash::Expire:

```
use Tie::Hash::Expire;
tie my %hash, 'Tie::Hash::Expire', {'expire_ seconds' => 5};
$hash{'magician'} = 'Erik Weis';
$hash{'musicians'} = ['Jalil', 'Ecstasy', 'Grandmaster Dee'];
$hash{'software'} = 'Side Effects Software';

print scalar keys %hash; # prints '3'
# do something for 6 seconds...
print scalar keys %hash; # prints '0'
```

An interesting twist on this module is Tie::Hash::Cannabinol, which describes itself as a "Perl extension for creating hashes that forget things." Specifically, the doc says:

> Once a hash has been tied to Tie::Hash::Cannabinol, there is a 25% chance that it will forget anything that you tell it immediately and a further 25% chance

that it won't be able to retrieve any information you ask it for. Any information that it does return will be pulled at random from its keys.

Oh, and the return value from exists isn't to be trusted, either.

To get back on a slightly more even keel, it should be mentioned that Tie::Scalar::Timeout or Tie::Scalar::Decay can do similar expiration magic for scalar variables.

Hashes with a Sense of History

There are two modules that give a hash variable the ability to remember all changes made over time. Usually when you set a value for a key in a hash, that value replaces any previous value with no record of there ever having been a previous value. With Tie::History or Tie::HashHistory, magic takes place in the background, making it possible to access previous values. For example, let's assume we were tracking the price of corned beef. We could write code that looked like this:

```
use Tie::History;

my $hobj = tie my %historyhash,'Tie::History';

$historyhash{'cornbeef'} = 1.28;
$hobj->commit;

$historyhash{'cornbeef'} = 1.35;
$hobj->commit;

$historyhash{'cornbeef'} = 1.25;
$hobj->commit;
```

The initial tie() line creates a special hash called %historyhash and then returns an object through which the history methods for that hash are controlled. Values for the key cornbeef are set using the standard notation. Each time we want to remember the state of the hash, we use the control object ($hobj) to commit it. At this point in the execution of our program, if we call the getall() method for the hash control object, we'd see:

```
DB<1> x $hobj->getall
0  ARRAY(0x50bcac)
0  HASH(0x5248cc)
      'cornbeef' => 1.28
1  HASH(0x53e624)
      'cornbeef' => 1.35
2  HASH(0x53e660)
      'cornbeef' => 1.25
```

There are other methods such as previous(), current(), and get() that allow you to retrieve the state of %historyhash at a specific point in the running history. Revert() will actually revert the hash to a specified position in the history. Tie::History is pretty spiffy, because it also works for scalars and arrays.

To keep things moving along, I won't show you an example for Tie::HashHistory, but I do want to mention one way that it differs from Tie::History. Tie::HashHistory is meant to augment other tie() modules and give them history superpowers. For example, if you were using the Tie::DBI module talked about in the last column, it might be handy for debugging purposes to keep a running history of all changes made to a Tie::DBI'd hash. Tie::HashHistory makes that easy.

Restrictive Hashes

Hashes are fabulous data structures. This notion of a collection of information in key/value pairs is both very powerful and very easy to use. But hashes can be a bit of a pushover. They are happy to store anything you throw at them, even mistakes. For example, a hash has no way to know that the key in:

```
$authors{'Charles Dikkens'} = "Hard Times";
```

is a typo (after all, it could be the well-known Dutch author). Just as the use strict pragma offers a good way to avoid variable name typos, the module Tie::StrictHash will do the same for hash keys:

```
use Tie::StrictHash;
my $hobj = tie my %authors, 'Tie::StrictHash';
```

Now we have two things: a hash (called %authors), which will behave in an unusual way, and a hash control object ($hobj) that will be used to make changes to that object. %authors is now unusual because, as the documentation says:

- No new keys may be added to the hash except through the add method of the hash control object.
- No keys may be deleted except through the delete method of the hash control object.
- The hash cannot be reinitialized (cleared) except through the clear method of the hash control object.
- Attempting to retrieve the value for a key that doesn't exist is a fatal error.
- Attempting to store a value for a key that doesn't exist is a fatal error.

So if we wanted to add a new key to the hash, we would call:

```
$hobj->add('Charles Dickens' => 'Hard Times');
```

Once in place, existing keys are changed just as one would expect:

```
$authors{'Charles Dickens'} = 'Great Expectations';
```

In the interests of full disclosure (since this isn't tie()-related), Perl 5.8.x versions implement something known as "restricted hashes" that have very similar properties. Using the Hash::Util module that ships with Perl, it is possible to lock down a hash or even individual keys in a hash. You don't get all of the Tie::StrictHash functionality or its very clear semantics (i.e., method calls for making changes), but it doesn't require installing a separate module.

Automatic Ranking

It's fairly common to write code that reads in a set of values and then has to report back the rank of each value in the whole list. Tie::Hash::Rank makes it easy to determine where a particular value stands in the ranking by constructing magical hashes that return a rank for each key stored in them instead of the associated value. For example:

```
use Tie::Hash::Rank;

tie my %rhash, 'Tie::Hash::Rank';

# grams of sugar
%rhash = (
        'countchocula' => 12,
        'booberry'     => 15,
```

```
        'trix'        => 13,
        'cheerios'    => 1,
);

print $rhash{'countchocula'}, "\n";    # prints 3
print $rhash{'booberry'},     "\n";    # prints 1
print $rhash{'trix'},         "\n";    # prints 2
print $rhash{'cheerios'},     "\n";    # prints 4
```

There are other similar modules that make it easy to keep track of the top *N* values in the hash (e.g., Tie::CacheHash).

Automatic Sorting

There are a few modules that handle keeping a hash's elements in a sorted order; these include Tie::Hash::Sorted and Tie::IxHash. If you find yourself repeatedly sorting and resorting your hash keys before processing, this can be a big win (especially when dealing with large data sets).

In a related vein, Tie::Array::Sorted helps you maintain an array whose elements stay (or appear to stay) sorted even in the face of element additions and deletions. I say "appear" because there is also a Tie::Array::Sorted::Lazy module in the package that is smart enough to only re-sort the array when its contents are retrieved. This works well for cases where you have an array that will be modified quite a bit before being read.

Encypted Hashes and Multi-Valued Storage

We're getting close to the end of our wish list, so let's take a quick look at the last two items on the list so we can move on to creating our own tie()-based code. The first is the ability to transparently encrypt and decrypt data in a hash. Tie::EncryptedHash is an excellent module for this purpose if it fits your program's model. Tie::EncryptedHash is good for those cases where you want to create a collection of information that needs to be encrypted when not in active use.

A hash tied using this module can contain both normal key/value pairs and "encrypting fields." Encrypting fields are those key/value pairs that begin with an underscore (e.g., $hash{'_name'}, $hash{'_socsecur'}, etc.). The hash itself is kept either in transparent/unencrypted or opaque/encrypted mode. The mode designates whether the encrypting fields found in that hash are encrypted or not.

To read or modify the encrypting fields in the hash, you unlock it with a special __password key; deleting this password will lock it again. In locked mode, you can safely drop the contents of the hash to disk or copy it over a network without fear of those fields being disclosed (the normal key/value pairs will continue to stay in plaintext). This is really simple in practice:

```
use Tie::EncryptedHash;
use Data::Dumper;

tie my %eh, 'Tie::EncryptedHash';

$eh{'normal'} = 'no magic here';

# let's unlock the hash

$eh{'__password'} = 'supersecretsquirrel';

# and store an encrypting field
$eh{'_encrypting'} = 'now you see it ...';
```

```
print "transparent: " . Dumper( \%eh ) . "\n";

# lock the hash
delete $eh{'__password'};

print "opaque: " . Dumper( \%eh );
```

The output of this program is:

```
transparent: $VAR1 = {
        'normal' => 'no magic here',
        '_encrypting' => 'now you see it ...'
};
opaque: $VAR1 = {
        'normal' => 'no magic here',
        '_encrypting' => 'Blowfish FHt07w3l/xyfd1/c4hskvQ
53616c7465645f5f4e8203d51070213d75fdf19b4c26b13435bc375600c49f
27b07e21be89f631df'
};
```

The last item on the wish list is one that makes a programmer's life easier. Tie::Hash::MultiValue is helpful in those cases where you want to store multiple values in a single hash key. The standard way to handle this situation is to store a reference to an anonymous array for that hash key (the Hash of Lists idea). Tie::Hash::MultiValue actually does this, but it makes the process a little easier. For example, instead of having to write something like this:

```
$mvh{'mike'} = [qw(greg peter bobby)];

# add a new element to the list, not the most pleasant syntax
push(@{$mvh{'mike'}},'tiger');
```

you can write:

```
use Tie::Hash::MultiValue;

tie my %mvh, 'Tie::Hash::MultiValue';

$mvh{'mike'} = "greg";
$mvh{'mike'} = "peter";
$mvh{'mike'} = "bobby";
$mvh{'mike'} = "tiger";
```

to get the same result. The module will also make sure that only unique values are stored, so that:

```
$mvh{'mike'} = "alice";
$mvh{'mike'} = "alice";
```

only stores "alice" once in the anonymous array associated with the key "mike." (A quick warning: The doc for Tie::Hash::MultiValue says that it is possible to assign multiple values at a time. This unfortunately does not work in the current version available on CPAN.)

That's the last of the items on our wish list, but there are still many impressive tie()-based modules available on CPAN that we could talk about. I could continue to blather on about modules such as Tie::File (which reads and writes lines in a file as if it were an array), Tie::HashVivify (in which you call your own subroutine every time you attempt to read a key in a hash that doesn't exist), or Tie::RemoteVar (which implements a client/server model that allows you to share the same variables between programs running on different machines).

Instead, let's move on to creating our own tie()-based code.

Don't Do It

As I mentioned in the first part of this series, there are a number of valid objections to writing tie()-based code. They include a couple of concerns:

- Performance: Tied variables can be quite slow compared to other approaches, because of all of the overhead.
- Maintainability: Without seeing the tie() call, other programmers can't know whether a tied variable will go "oogah-boogah" every time it is accessed, instead of exhibiting the usual variable behavior.

Both of these are perfectly reasonable concerns, so let me quickly state the alternative to tie()-based code: Write your code using standard OOP practices, create objects instead of tie()'d variables, and call the methods of those objects explicitly. Instead of:

```
$special{'key'} = 'value'; # prints "oogah-boogah"
```

use something like:

```
$special->oogahboogah('key','value')
```

instead. Although this is not the most glamorous alternative, it does help with both performance and maintainability.

No, Really. Tie() Me Up, Tie() Me Down

If you've determined you do want to write code for tie() there are a few ways to go about it. In the interests of space and time, I'm going to show you only one way, using a very simple example that makes SNMP (Simple Network Management Protocol) queries using hash semantics. If you are not familiar with how SNMP works or the Net::SNMP module (perhaps a future column topic), the short version is that it is a protocol for querying management information from a device (e.g., a router).

The standard way to construct a tie()-based module requires a bit of Perl OOP knowledge. If you don't have that knowledge or just want something quick and dirty you can use the Tie::Simple module to hide the details for you. These details are found in the _perltie_ manual page (perldoc perltie) and are the subject of Chapter 9 in Damian Conway's *Object Oriented Perl*.

Here's how the code you are about to see works. To create tie() code, you build an OOP-based package. This package creates objects with methods that implement all of the standard variable operations (fetch, store, exists, delete, etc.) required of that variable type. Code for tie()-ing scalars needs to contain 4 subroutines to cover all of the operations; for hashes the number is 9, and for arrays it goes to 13. To avoid having to write all that code for this example, we're going to inherit a set of default subroutines from Tie::StdHash module in the Tie::Hash package. These subroutines mimic the standard hash behavior, leaving us free to redefine just the operations that suit our purpose. In the example that follows, we redefine only the TIEHASH operation, called when the tie() function is executed, and FETCH, called when a key is looked up in a hash.

Here's the code, with explanation to follow:

```
package SNMPHash;
require Tie::Hash;
use Net::SNMP;

@ISA = (Tie::StdHash);

sub TIEHASH {
    my ( $class, $arghash ) = @_;
```

```
    # create the object
    my $self = {};
    bless $self, $class;

    # create an SNMP session and store it in the object
    my ( $session, $error ) = Net::SNMP->session( %{$arghash} );
    die "Could not establish SNMP session: $error" unless defined $session;
    $self{'session'} = $session;

    return $self; # return the object
}

sub FETCH {
    my $self = shift;
    my $key  = shift;

    # do the actual SNMP lookup
    my $result = $self{'session'}->get_request( -varbindlist => [$key] );

    return $result->{$key};
}
1; # to allow for loading as a module
```

Here's a very brief tour of the code: We start by declaring the name of the package (which will become the class of the object created). After the usual loading of modules we declare that we'll be inheriting from the Tie::StdHash module. This gives us the freedom to redefine two operations, TIEHASH and FETCH.

For TIEHASH, we create an empty object, initialize an SNMP session object based on the arguments in the tie() statement, and store a reference to this session in the object for later use. That use happens in the very next subroutine when we define what should happen upon key lookup. In this case, we take the key, turn it into a standard SNMP _get_ request, and then return the answer. When this happens it looks like the tie()'d hash has magical keys consisting of SNMP variables (in OID form), which can be queried to see the live data.

How does this get used?

```
# assumes we saved the previous example in a file called SNMPHash.pm
# someplace Perl can find it
use "SNMPHash";
tie my %snmphash, 'SNMPHash',
      { '-hostname' => 'router', '-community' => 'public' };
# this long string of numbers is just a way of referencing (in SNMP
# OID form) the SNMP variable that holds the description for a system
# (i.e. sysDescr.0). See Elizabeth Zwicky's article at
# http://www.usenix.org/publications/login/1998-12/snmp.html or
# another SNMP tutorial for more info
print $snmphash{'.1.3.6.1.2.1.1.1.0'},"\n";
```

This yields something like:

```
Cisco Internetwork Operating System Software
IOS (tm) s72033_rp Software (s72033_rp-PK9S-M), Version 12.2(18)SXD1,
RELEASE SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2004 by Cisco Systems, Inc.
Compiled Wed
```

There's a ton of things missing from this sample code; it is just meant to be a snippet to start your motor running. There's virtually no error checking. Most of the hash operations aren't implemented. (It isn't exactly clear just

how all of the SNMP operations should map onto hash operations. Some are obvious, but the others deserve some head scratching.) Still, you've now received a taste of what it takes to write your own tie()-based code. And with that, we need to wrap up this issue's column. Take care, and I'll see you next time.