

The Perl Journal

***I*value Accessor Methods With Value Validation**

Juerd Waalboer • 3

Monitoring Network Traffic With *Net::Pcap*

Robert Casey • 6

Exception Handling in Perl With *Exception::Class*

Dave Rolsky • 9

Designing for Pluggability

Simon Cozens • 14

Cleaning Up a Symlink Mess

Randal L. Schwartz • 18

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Tim Kientzle:

***High Performance MySQL* • 20**

Source Code Appendix • 21

LETTER FROM THE EDITOR

Futurism is a Tricky Business

Alan Kay, one of the truly great pioneers of the computer age, recently gave an interview to Fortune.com in which he lamented our lack of technological progress in recent decades, given the enormous potential of the computer. He lays part of the blame for this at the feet of business:

But business, instead [of education], has been the primary user of personal computers since their invention. And business, he says, "is basically not interested in creative uses for computers."

Hang on. I think this is overstating the culpability of business. Business is completely interested in creative uses for computers, as long as those uses generate revenue. Show me a creative use of a computer that makes money, and I'll show you a business that has embraced it whole-heartedly.

If he wants to argue that the ways of making money with creative computing are out there, and businesses just can't figure out what they are, fine. But a failure of imagination is not the same thing as a lack of interest.

Sadly, where we clearly have had a failure of both interest and imagination is in our educational systems. Computers in our schools simply provide digital typewriters and windows onto the unrefined, often inaccurate information on the Web. Kids coming out of our schools tend to have a very fixed idea of what a computer is for, and no idea of its true potential.

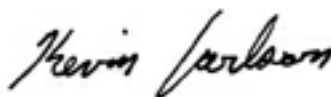
This, of course, is not news to Alan Kay. His efforts with the kid-friendly Squeak Smalltalk implementation (<http://www.squeakland.org/>) prove that he's well aware of these problems, and is committed to fixing them.

I agree that there has been much squandered potential in the last couple of decades. And Kay is right to point to some of the most prevalent computer technologies today—word processing, spreadsheets, and e-mail—as simply digital analogues of paper-and-ink processes, which are hardly revolutionary. In fact, it could be argued that the only really revolutionary idea in computer technology in the past 20 years is the Web. But I don't think it's quite fair to hold the business world responsible for somehow having a stranglehold on our ability to use computers creatively.

Truly creative uses of technology usually represent pretty revolutionary ideas. But those revolutions don't come along very often. In order to be revolutionary, a technology has to be used by the masses—but the majority of users will not embrace a technology until its benefits become obvious. The idea behind it has to be compelling and, let's face it, most of us don't come up with revolutionary ideas. They tend to come from a small group of smart people.

How many ideas in computing in the last 20 years have been compelling enough to qualify as revolutionary? There have been many good ideas, and many have succeeded, but most have been evolutionary. I suspect that the reason we're not advancing any faster than we are has less to do with either a lack of imagination or a lack of interest, and more to do with the fact that the truly revolutionary ideas are, by their very nature, few and far between.

(To read the whole *Fortune* interview with Kay, go to <http://www.fortune.com/fortune/fastforward/0,15704,661671,00.html>.)



Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.

THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2004 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Song

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, Brian D'Joy, Moshe Bar, Randal Schwartz, Andy Lester

CONTRIBUTING EDITORS

Simon Cozens, Dan Sugalski, Eugene Kim, Chris Dent, Matthew Lanier, Kevin O'Malley, Chris Nandor

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Timothy Trickett

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403
650-513-4300. <http://www.tpj.com/>

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND

MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, TECHNOLOGY SOLUTIONS Robert Faletta

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER ENTERPRISE

ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING

SOLUTIONS Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES

Marie Myers

Perl News

Perl Quiz of the Week

Mark-Jason Dominus has launched a mailing list delivering a weekly Perl programming puzzle to subscribers. The quizzes come in two degrees of difficulty—"normal" or "expert." An example of an expert-level quiz:

Write a subroutine, `'subst'`, which gets a string argument, `$s`. It should search `$s` and replace any occurrences of `"$1"` with the current value of `$1`, any occurrences of `"$2"` with the current value of `$2`, and so on. For example, if `$1`, `$2`, and `$3` happen to be `"dogs," "fish,"` and `"carrots,"` then `subst("$2, $1 and $3")` should return `"fish, dogs, and carrots"`.

To subscribe to the quiz mailing list, send e-mail to perl-qotw-subscribe@plover.com; past quizzes are temporarily archived at <http://www.urth.org/~metaperl/domains/semantic-elements.com/perl/>.

New Parrot Pumpking

Jeff Goff has stepped down as release manager and Keeper of the Keys and Source for Parrot, passing the title on to Steve Fink. In announcing the transfer, Dan Sugalski said, "Steve's been active with Parrot since near the beginning, and I have every confidence in his being able to keep the project moving forward."

Safe Module Patched

A security hole in the *Safe* module was discovered and reported to the [perl.perl5.porters](mailto:perl.perl5.porters@perl.org) newsgroup by Andreas Jurenda. The problem lies with *Safe::reval()*, which executes given code in a safe compartment. "But," explained Jurenda, "this routine has a one-time safeness. If you call *reval()* a second (or more) time with the same compartment, you are potentially unsafe." Version 2.09 of the *Safe* module, available at <http://search.cpan.org/author/ABERGMAN/Safe/>, fixes the problem.

The Perl Foundation Grant Wrap-Up

The 2002 recipients of Perl Foundation funding—Damian Conway, Dan Sugalski, and Larry Wall—have posted accounts of their accomplishments during the grant period (which ended in July) at <http://www.perlfoundation.org/gc/grants/2002.html>. Damian Conway, who supplemented his grant period with a final speaking tour, posted a final blog entry at the beginning of October, calling his grant tenure "the toughest year-and-a-half I've ever experienced, but also the most rewarding." During that time, Conway accepted 56 speaking engagements across four continents. He also wrote 21 new modules; 88 "node" entries on the Perl Monks site; four Exegeses; 192 newsgroup messages; and over 5000 e-mails in response to Perl questions.

The Perl Foundation solicits contributions to the grant fund at <http://www.perlfoundation.org/contribute.html>.

"Spoofathon" Fundraiser Opens

Another way to raise money for the Perl Foundation is by entering Nicholas Clark's Perl Advocacy Spoofathon, which offers six donations of 100 guineas each (a total of about \$160 in American dollars) to be paid to the Perl Foundation in honor of the best spoof essays submitted. The three fixed essay categories are: "INTERCAL Is Better than Perl," "Brainf**k Is Better than Perl," and "Befunge Is Better than Perl." The three remaining donations will be awarded in honor of the best `"$foo Is Better than Perl"` essays submitted.

"There have been several badly researched `$foo` is better than perl articles in the recent past, that have irritated myself and many other people in the perl community," explains Clark on the Spoofathon web page (<http://www.perl.org/advocacy/spoofathon/why.html>). "I can't stop third parties writing these articles. But I do hope I can start to make people treat them with the seriousness that they deserve."

There is no fixed essay deadline; the judges, Michael G. Schwern and Greg McCarroll, will announce the awards at their discretion. Entries can be submitted to spoofathon@perl.org.

Komodo 3.0 Released

ActiveState has released version 3.0 of their Komodo IDE, optimized for Perl and other open-source languages. New features include powerful code and object browsers, extended debugger capabilities (using the open-source DBGP protocol, coauthored by ActiveState), and a multilanguage (Perl, Python, and Tcl) interactive shell. See <http://www.activestate.com/Products/Komodo/> for more details.

New Perl Books

Several new Perl books have been released in the past month. Tim Jenness and Simon Cozens coauthored *Extending and Embedding PERL* (Manning Publications Company), dealing with the use of Perl in C programs. *Programming Perl in the .NET Environment*, by Yevgeny Menaker, Michael Saltzman, and Robert Oberg, is available from Prentice Hall and is designed for use as a textbook, beginning with a tutorial on the Perl language and ending with enterprise application integration. *Embedding Perl in HTML with Mason*, published by O'Reilly, covers the *HTML::Mason* framework. Authors Dave Rolsky and Ken Williams describe Mason as "a powerful text templating tool for embedding Perl in text." The book's web site is at <http://www.masonbook.com/>.

lvalue Accessor Methods With Value Validation

You like Perl. You appreciate its flexibility, expressiveness, power, and culture. Or maybe that's just me. I like that Perl enables me to write what I mean in a way that I feel comfortable with. More important is that it lets me change what I don't like. Even when something has always been done in a particular way, I can decide to make things easier for myself. Even better: I can share my code via CPAN and make things easier for everyone who likes it!

For a long time, I used accessor methods in the same style that most Perl coders know them. But it never felt natural. Let me try to explain this using some contrived example code:

```
{
    my $foo
    sub foo {
        @_ ? $foo = shift : $foo
    }
}

foo(42);
print foo;
```

Do you think that looks strange? I do. This is not something you're likely to encounter: a sub that exists only to provide access to a variable. Usually, the variable is used directly. I'm very comfortable with just *\$foo* and I think you are, too.

But for some reason, the object-oriented world is different. For that world, someone once thought that access to variables should only happen through accessor methods, like the one illustrated above. The OO variant would shift a *\$self* and then set an element in the structure that it represents, but the idea is the same.

When I learned OO in Perl, I found this very strange and I didn't like it. I don't find it strange anymore, but I still don't like it. In my perfect world, everything is simple and I don't think much about who can access my data and who should not be allowed to. But more importantly, things should be easy to use.

It may be easy to use such accessor methods and, with the help of some modules, it can even be done without thinking about the internals, but some very fundamental things that Perl programmers are used to are not possible simply because there is no *lvalue* to work with (see the sidebar "What is an *lvalue*?" for more on *lvalues*).

Juerd is the owner of Convolution (<http://convolution.nl/>), and is the author of numerous Perl modules, including `Attribute::Property`.

For example, when I made my first LWP program, I wanted to add the name of my program to the user agent string. The original user agent could stay because I was rather proud that I used Perl with LWP. For this, I had to call *\$ua->agent* twice, once for reading the value and once for storing the new one. This really disappointed me because I wanted to use *s/\$/ \$0/*, or at least just *. = " \$0"*. (This was before letting the user agent string end with a space meant anything.)

Of course, it's possible to just access the internal hash, but that's not part of the public API and I had already learned that laziness is a virtue, so I didn't like having to type curlies all the time.

I don't want OO programming to be that much different from "normal" programming. A method should behave like a sub and a property should behave like a variable. That parentheses are required for method calls with arguments is bad enough already. Properties should behave like variables—they already do in Ruby, Python, Javascript, Perl 6, Visual Basic, PHP, and the like. But not in current Perl. In Perl, a property is usually a method that

What is an *lvalue*?

You may not have heard about *lvalues* and *rvalues* before. "L" and "R" stand for left and right. In short, an *lvalue* is a value that can occur on the left side of an assignment, as illustrated:

```
lvalue = rvalue;
```

A subroutine call usually results in an *rvalue*, so you can't use assignment and mutation methods on it. The *lvalue* attribute changes that. Some built-in operators can be used as an *lvalue*, like *substr*:

```
my $foo = "fooooooooo!";
substr($foo, 3) =~ s/o+/bar/g;
print $foo;
```

In this example, *s///* operates on the *lvalue* that *substr* returned. This *lvalue* represents only a part of *\$foo*.

—J.W.

accesses a hash element, and accessing that hash element directly is frowned upon.

Using accessor methods feels like using `FETCH` and `STORE` instead of a tied hash: It works, but I'd rather have something a little more convenient.

Attributes

Perl now has attributes. To clarify things, attributes don't really have anything to do with object orientation: at least in the Perl 5 world, an attribute is not the same as a property. (If this is hard to understand, prepare yourself for Perl 6, where attributes become traits, properties become attributes, and properties are runtime traits.)

Attributes can be attached to variables and subroutines when they are declared. For this, the colon is used. Several built-in attributes exist: *locked*, *method*, *lvalue*, and *unique*. All of these in some way change the behavior of your variable or sub.

The one I'm after is *lvalue*. This lets me do exactly what I want—have a subroutine (or more specifically, a method) behave like a variable.

Simply put, you have to return an *lvalue* and that *lvalue* is then used. This “returning” cannot be done with the *return* keyword but instead has to be the natural expression the sub evaluates to. In other words, the last expression in the sub.

So my previous example, using this new feature, would be:

```
{
    my $foo;
    sub foo : lvalue {
        $foo
    }
}

foo = 15;
print foo;
```

Or, in the simple OO world where every object is just a reference to a blessed hash:

```
sub foo : lvalue {
    my $self = shift;
    $self->{foo}
}

$object->foo = 15;
print $object->foo;
```

Now the method can be used as if it were a variable. All those nice `+=`-like operators work with it and even `s///` now works. It's still an accessor method, but at least it feels much better now—and I don't have to use curlies.

Tradition

History tells us that accessor methods don't just exist to hide internals but also to do something with the new value before it gets assigned: To trim whitespace from it, to automatically make it into an object, or more commonly, to test if the value is what we consider valid. Truth be told, it wasn't history that told me, it was the monks of Perl Monks (<http://www.perlmonks.org/>). Having learned about the *lvalue* methods and having used them for a while, I went there to ask why this technique was not used more often. Clearly, it was superior, I thought.

As it turns out, it's superior only for the class user—not for the class maintainer who likes to make sure no strange values are passed because that eventually means debugging a lot and tracing back to where the strange value came from. With normal accessor methods, you can see the new value before you decide that it is valid.

With *lvalue* methods, you usually don't know what happens later. Someone can take a reference to the variable you returned and assign it a new value 14 times. Traditionally, we have used accessor methods that do not behave like variables because if something behaves like a variable, we do not have total control.

*Traditionally, we have used
accessor methods that do not
behave like variables because if
something behaves like a variable,
we do not have total control*

Total Control

But that doesn't have to be true. It is probably true in other languages, but in Perl, we do in fact have the power to decide what should happen when our variables are used. We can tie them to a class and keep track of everything that happens, adjusting things if what happens is not what we want to happen.

tie is used either to link methods to a variable or to provide a variable-like interface to methods, depending on the point of view. The *FETCH* and *STORE* methods are called to fetch and store the value from and into the variable. Another word for variable is *lvalue*.

With *tie*, it is possible to keep track of what happens to an *lvalue*, and that is exactly what was missing: control over what happens to the variable after the sub has finished. Even though the recipe sounds simple, since all we have to do is *tie* our value before returning it, we get complex code when doing so. We also need a class to *tie* to.

Suppose we want to make sure that whatever value we put into *foo* is less than 50. That can be written as:

```
package Tie::Scalar::LessThanFifty;
use Carp::Croak;
use Tie::Scalar;
our @ISA = ('Tie::StdScalar');

sub STORE {
    my ($self, $new_value) = @_;
    croak "Invalid value" unless $new_value < 50;
    $$self = $new_value;
}

package My::Class;

sub foo : lvalue {
    my $self = shift;
    tie $self->{foo}, 'Tie::Scalar::LessThanFifty';
    $self->{foo}
}

$object->foo = 10;
$object->foo = 55; # "Invalid value"
```

Improvement

While this works and solves the immediate problem, it isn't really easy to type and maintain. When written like this, a separate class is needed for each value test, and `$self->{foo}` stays tied long after the calling code has no need for it anymore.

So, ideally, the class we *tie* to should allow a *coderef* to be passed for the validity test to avoid needing dozens of *Tie* classes. And only a temporary variable that acts as an interface for the hash element should be tied so that the actual hash element itself stays untied and things aren't tied any longer than necessary.

It would be even better if there was a very simple syntax to create such new style properties.

Enter `Attribute::Property`

`Attribute::Property` does all this and a little more. It lets you write the big block of code above simply as:

```
sub foo : Property { $_ < 50 }

$object->foo = 10;
$object->foo = 55; # "Invalid value for foo property"
```

Its interface is an attribute, like the built-in *lvalue* attribute. Attributes that are not built into Perl usually begin with a capital letter, like *Property* in this example. A *Class::MethodMaker*-like interface could just as easily have been used, but Perl has a new feature and I like trying out new features, especially if that feature lets people use very nice syntax without using a source filter.

Whenever you use the *Property* attribute, internally, your method is replaced with a complex and microoptimized method that has the *lvalue* attribute and returns a tied temporary variable. The original code block (`$_ < 50`, in this case) is hijacked and passed on to *tie*, so that it can be used later on when something decides to *STORE* a new value.

When a value is stored, the original code block is executed and its return value is evaluated in Boolean context. When True, the new value is assigned, but if False, something croaks. If the default error message might not be appropriate or you have a message that is more descriptive, it's just a matter of croaking before the *STORE* handler gets the chance to do so:

```
sub foo : Property {
    $_ < 50
    or croak "Value for foo property exceeds limit"
}
```

As you have probably already seen in the examples, the new value is available as `$_`. But just for convenience, it is also passed in `@_`. In `@_`, you'll find the object (or class) and the new value. `$_` and `$_[1]` are both aliases for the new value. That means that you can choose to change it and that the value exists only once in memory. For example, to trim trailing whitespace from the new value:

```
sub bar : Property { s/\s+//; 1 }
```

The extra 1 at the end is to make sure truth is returned. Otherwise, a value without trailing whitespace would have been considered an invalid value.

To ease migrating from the old accessor methods to the new *lvalue* accessor methods, `Attribute::Property` also lets the user use the old syntax. That means that `$object->foo = 5` and `$foo->object(5)` do the same thing.

In case no value validation is needed, the validation code block can simply be left out. For efficiency, the value is then left untied. Don't forget that without a code block, subroutine declarations need a semicolon:

```
sub baz : Property;
```

If you have several properties, vertically align the colons to get very neat and readable code:

```
package Article;

sub title      : Property;
sub subtitle   : Property;
sub author     : Property;
sub abstract   : Property;
sub body       : Property;
```

All that's missing now is a constructor. You can either craft one yourself (as long as the object is a reference to a blessed hash) or use the simple one that `Attribute::Property` provides:

```
sub new        : New;
```

This is all that is needed to get a functional *Article* class.

Perfection

Now, my properties finally behave like variables, without sacrificing validation and being able to alter the new value before it's stored. And all that comes with syntax that lets me write most properties on a single line.

A lot of time was spent on writing `Attribute::Property`, even though the module has only 80 lines of code, but now I write clean OO code in much less time, so it was for a good cause: laziness.

TPJ

Reading `Attribute::Property`

Using `Attribute::Property` is very easy, but reading its source code is somewhat harder. That is because when we made *A::P*, we wanted a sub to fit in 80 by 24 characters and we didn't want more subs than necessary. The module combines a lot of better- and less-known idioms, micro-optimized to extremes. While it is clean and logical code, it is not a good example of readability. One needs to be fluent in Perl to read it and, to make things even worse for the casual passer-by, most of the variables have one-letter names.

To help you understand what goes on inside of `Attribute::Property`, here is a list of variables used, with explanations:

<code>\$s</code>	Symbol
<code>\$r</code>	Reference (CODE)
<code>\$n</code>	Name
<code>\$foo</code>	Variable to be tied
<code>%p</code>	Existing properties
<code>\$c</code>	Class
<code>\$o</code>	Object
<code>\$l</code>	Local ref to shortmess
<code>\$m</code>	Method

—J.W.

Monitoring Network Traffic with *Net::Pcap*

Throughout its history, Perl has always found a home in the suite of tools employed by systems administrators in the maintenance, monitoring, and administration of computer systems. As the needs of systems administrators have grown, Perl has kept pace with an ever-expanding base of diverse application and tool components available from the comprehensive Perl archive network (CPAN, <http://www.cpan.org/>).

One tool available from CPAN of particular utility to systems administrators is the *Net::Pcap* library. This module provides an interface for Perl to the Lawrence Berkeley National Laboratory Network Research Group's *pcap* library, which is a system-independent interface for user-level packet capture. This library provides a portable framework for low-level network monitoring and can be used for a variety of network monitoring functions, including network statistics collection, security monitoring, and network debugging.

Setting the Device

The first step in building an application or tool using *Net::Pcap* and the underlying *libpcap* library for network monitoring is to determine an available network interface that can be used for monitoring this network traffic. This device can be specified by the user for specific network monitoring, particularly on multihomed machines, or can be determined via the *lookupdev* method of *Net::Pcap*.

The syntax of the *lookupdev* method is as follows:

```
$dev = Net::Pcap::lookupdev(\$err)
```

This method returns the name of a network device, which can be used for monitoring network traffic. For example:

```
use Net::Pcap;
use strict;

my $err;
my $dev = Net::Pcap::lookupdev(\$err);
if (defined $err) {
    die "Unable to determine network device for monitoring - ", $err;
}
```

Rob is technical manager for Bluebottle Solutions, a verification technology company based in Melbourne, Australia. He can be reached at rob.casey@bluebottle.com.

The string reference *\$err* is passed as an argument to this method and is returned with an error description in the event of method failure. Upon method failure, the returned device name is also undefined.

With the latest version of the *Net::Pcap* module (0.05), a list of all network devices that can be used for network monitoring can be retrieved using the *findalldevs* function. The syntax of this function is similar to that of the *lookupdev* function:

```
@devs = Net::Pcap::findalldevs(\$err)
```

Another useful method available from the *Net::Pcap* library is *lookupnet*, which can be used to determine the network address and netmask for a device. This method is useful for the validation of a device name supplied for network monitoring by a user.

The syntax of the *lookupnet* method is as follows:

```
Net::Pcap::lookupnet($dev, \$net, \$mask, \$err)
```

This method returns the network address and netmask for the device specified—*\$dev*. This method also follows the conventions of the underlying library of returning 0 for success and -1 for failure and, as such, error checking for this and other *Net::Pcap* functions may use the pseudoreverse mentality of the *die if...* idiom. For example:

```
my ($address, $netmask, $err);
if (Net::Pcap::lookupnet($dev, \$address, \$netmask, \$err)) {
    die "Unable to look up device information for ", $dev, " - ", $err;
}
print STDOUT "$dev: addr/mask -> $address/$mask\n";
```

Capturing Packets

Once an appropriate network device has been determined, the process of packet capturing can be initiated. The *Net::Pcap* function *open_live* returns a packet capture descriptor, which can be used for capturing and examining network packets.

The syntax of the *open_live* method is as follows:

```
$object = Net::Pcap::open_live($dev, $snaplen, $promisc, $to_ms, \$err)
```

The *\$dev* parameter specifies the network interface from which to capture network packets while the *\$snaplen* and *\$promisc* parameters specify the maximum number of bytes to capture from each packet and whether to put the interface into promiscuous mode, respectively. The latter of these parameters, the promiscuous

mode, places the network card into a “snooping” mode where network packets not necessarily directed to the packet-capturing machine are captured. In a nonswitched network environment, this can capture all network traffic. The *\$to_ms* parameter specifies a read time-out for packet capturing in milliseconds—a *\$to_ms* value of 0 captures packets until an error occurs, while a value of -1 captures packets indefinitely.

Individual packets can be retrieved from the network interface using the *next* method of the *Net::Pcap* module in the following manner:

```
$packet = Net::Pcap::next($object, \%header)
```

This method will return the next packet available on the network interface registered with the *Net::Pcap* packet descriptor object. The *%header* hash reference is populated with details relating to packet header information, namely:

- *len*, the total length of the packet.
- *caplen*, the captured length of the packet; this corresponds to the *\$snaplen* argument passed to the *Net::Pcap::open_live* method.
- *tv_sec*, the seconds value of the packet timestamp.
- *tv_usec*, the microseconds value of the packet timestamp.

If no packets are available on the network interface for capture, the return value of the *Net::Pcap::next* method is undefined.

Alternatively, continuous packet capture can be performed by establishing and registering a callback function to which *Net::Pcap* can pass captured packets to for analysis and reporting. For this, the *loop* method of *Net::Pcap* is called:

```
Net::Pcap::loop($object, $count, \%callback_function, $user_data)
```

This method takes four mandatory arguments. *\$object* is the *Net::Pcap* object returned from the *Net::Pcap::open_live* method. *\$count* is a number indicating the number of packets to capture. If the number passed to this function is negative, *Net::Pcap::loop* will capture packets indefinitely (or until an error occurs if the *\$to_ms* argument of the *open_live* method is set to 0). The third parameter is a subroutine reference to the callback function. The fourth argument represents arbitrary data that is passed with the callback function (along with captured packets) and can be used as a method to tag captured packets or distinguish between several open-packet capture sessions.

The callback function specified by the *Net::Pcap::loop* method receives the following arguments when called:

- The *\$user_data* string passed to the *Net::Pcap::loop* method.
- A reference to a hash containing packet header information (as described in association to the *Net::Pcap::next* method above).
- A copy of the entire packet.

An example of the callback function associated with packet capture may look like the following:

```
sub callback_function {  
    my ($user_data, $header, $packet) = @_;  
    ...  
}
```

One important limitation of this module in its current version, however, is that only a single callback function and user data scalar can be registered at any given time, as both elements are stored within global variables within the *Net::Pcap* namespace.

Filtering Packets

While the previously methods described provide the means by which to capture all network traffic, the real power offered by the *libpcap* library is to selectively filter network packets to monitor specific traffic. The filtering of network packets can be set through the use of a filter language specific to the *libpcap* library. A description of this filter language can be found in the *libpcap* source code or on the *tcpdump*(8) man page. The use of this filter language for the selective capture of network packets does require some knowledge of TCP/IP networking and the underlying packet structure, and a detailed description of this filter language would be beyond the scope of this article.

*The real power offered by the
libpcap library is to selectively
filter network packets to monitor
specific traffic*

The *Net::Pcap* module provides methods for the compilation and setting of filters for network packet capture by means of the *Net::Pcap::compile* and *Net::Pcap::setfilter* methods.

The arguments of the *Net::Pcap::compile* method are as follows:

```
Net::Pcap::compile($object, \%filter_compiled, $filter_string, $optimize, $netmask)
```

This method will compile and check the filter specified in *\$filter_string* for the *Net::Pcap* object *\$object* and return the compiled filter in the scalar *\$filter_compiled*. The filter is optimized where possible if the *\$optimize* variable is True. This function, like other *Net::Pcap* functions, returns 0 if successful or -1 if an error occurs.

The compiled filter string, *\$filter_compiled*, can then be applied against the *Net::Pcap* object using the *Net::Pcap::setfilter* method. For example:

```
Net::Pcap::setfilter($object, $filter_compiled)
```

Decoding Captured Packets

Once packets have been captured using the *Net::Pcap* interface to *libpcap*, the next step is to decode these packets and make sense of the network packet data collected. This can be performed by constructing unpack templates for captured data or (more easily) through the *NetPacket::* collection of modules. These modules each contain methods for extracting information from and about network packets, the most useful of which is (arguably) the *decode* method. This method returns a hash of metadata about the passed packet specific to the packet type.

For example, the *NetPacket::Ethernet::decode* method will return the following information on captured ethernet packets:

- *src_mac*, the source MAC address for the Ethernet packet as a hex string.
- *dest_mac*, the destination MAC address for the Ethernet packet as a hex string.

- *type*, the protocol type of the Ethernet packet; for example, IP, ARP, PPP, and SNMP.
- *data*, the data payload for the Ethernet packet.

Further information on each of the *NetPacket::* modules and the information returned by the decode function can be found on their respective man pages.

In addition to this, each of the *NetPacket::* modules also contain a *strip* method, which simply returns the data payload of the network packet. This is useful when the network encapsulation is of little or no concern to your application.

Error Handling

While scalar references can be passed as arguments to a number of the *Net::Pcap* methods (namely *lookupdev*, *findalldevs*, *lookupnet*, and *open_live*), in order to return error messages, there are a number of dedicated methods supplied by this module for more generic error handling:

```
Net::Pcap::geterr($object)
Net::Pcap::perror($object, $prefix)
Net::Pcap::strerror($errno);
```

The *Net::Pcap::geterr* function returns an error message for the last error associated with the packet capture object *\$object*. The *Net::Pcap::perror* function prints the test of the last error associated with the packet capture object *\$object* on standard error prefixed by the string in *\$prefix*. *Net::Pcap::strerror* returns a string-describing error number *\$errno*.

Cleaning Up

Once finished capturing packets, the *Net::Pcap::close* method should be called to close the packet capture device. For example:

```
Net::Pcap::close($object)
```

Putting It All Together

Listing 1 shows a way of putting all of the aforementioned techniques to use for network administration. In this example, details of all TCP packets with the SYN header flag set captured by a machine will be reported. These network packets are used by a client in initiating a connection with a server and can be used to initiate denial of service attacks against a network host. For further information on TCP packet structure and the SYN header flag, see RFC793.

From Here

This article has touched upon the basic functionality of the *Net::Pcap* module and how it can be used in network administration. Other features of this excellent module that have not been covered in this article include saving captured network packets to files and interface statistics handling. These features are well described in the documentation for this module, and readers should have no trouble employing these extended features of the *Net::Pcap* module.

References

Blank-Edelman, David N. *Perl for System Administration*, O'Reilly, 2000, ISBN 1-56592-609-9.

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

Listing 1

```
use Net::Pcap;
use NetPacket::Ethernet;
use NetPacket::IP;
use NetPacket::TCP;
use strict;

my $err;

# Use network device passed in program arguments or if no
# argument is passed, determine an appropriate network
# device for packet sniffing using the
# Net::Pcap::lookupdev method

my $dev = $ARGV[0];
unless (defined $dev) {
    $dev = Net::Pcap::lookupdev(\$err);
    if (defined $err) {
        die "Unable to determine network device for monitoring - ", $err;
    }
}

# Look up network address information about network
# device using Net::Pcap::lookupnet - This also acts as a
# check on bogus network device arguments that may be
# passed to the program as an argument

my ($address, $netmask);
if (Net::Pcap::lookupnet($dev, \$address, \$netmask, \$err)) {
    die "Unable to look up device information for ", $dev, " - ", $err;
}

# Create packet capture object on device

my $object;
$object = Net::Pcap::open_live($dev, 1500, 0, 0, \$err);
unless (defined $object) {
    die "Unable to create packet capture on device ", $dev, " - ", $err;
}

# Compile and set packet filter for packet capture
# object - For the capture of TCP packets with the SYN
# header flag set directed at the external interface of
# the local host, the packet filter of '(dst IP) && (tcp
# [13] & 2 != 0)' is used where IP is the IP address of
```

```
# the external interface of the machine. For
# illustrative purposes, the IP address of 127.0.0.1 is
# used in this example.

my $filter;
Net::Pcap::compile(
    $object,
    \$filter,
    '(dst 127.0.0.1) && (tcp[13] & 2 != 0)',
    0,
    $netmask
) && die "Unable to compile packet capture filter";
Net::Pcap::setfilter($object, $filter) &&
    die "Unable to set packet capture filter";

# Set callback function and initiate packet capture loop

Net::Pcap::loop($object, -1, \$syn_packets, '') ||
    die "Unable to perform packet capture";

Net::Pcap::close($object);

sub syn_packets {
    my ($user_data, $header, $packet) = @_;

    # Strip ethernet encapsulation of captured packet

    my $ether_data = NetPacket::Ethernet::strip($packet);

    # Decode contents of TCP/IP packet contained within
    # captured ethernet packet

    my $ip = NetPacket::IP->decode($ether_data);
    my $tcp = NetPacket::TCP->decode($ip->{'data'});

    # Print all out where its coming from and where its
    # going to!

    print
        $ip->{'src_ip'}, ":", $tcp->{'src_port'}, " -> ",
        $ip->{'dest_ip'}, ":", $tcp->{'dest_port'}, "\n";
}
```

TPJ

Exception Handling in Perl with *Exception::Class*

Perl 5's built-in exception-handling facilities are minimal, to say the least. Basically, they consist of *eval* blocks (*eval { ... }*) and the `$@` variable. If you want something a little more full featured, you'll have to turn to CPAN. Surprisingly enough, this is one area of CPAN where the number of choices is not exactly overwhelming.

The three exception-related modules on CPAN that I think are most useful and well-documented are *Error.pm*, *Exception.pm*, and my own module, *Exception::Class*. The first two aim at providing a more sophisticated *try/catch* syntax for Perl. My module, *Exception::Class*, is solely aimed at making it easier to declare hierarchies of exception classes. *Error.pm* and *Exception.pm* do provide base classes for exception objects, but they do not do much to help you create your own exception classes.

Declaring Exception Classes

To declare an exception class with *Exception::Class*, you simply use *Exception::Class*:

```
use Exception::Class ( 'My::Exception::Class' );
```

The above code snippet magically creates the new class *My::Exception::Class*. If you don't say otherwise, all exception classes created by *Exception::Class* are subclasses of *Exception::Class::Base*, which provides a basic constructor and accessors.

Of course, the above example isn't all that exciting, since you could just as easily have done it this way:

```
package My::Exception::Class;
use base 'Exception::Class::Base';
```

and gotten exactly the same result.

But what if you want to declare multiple classes at once, maybe for use within an application or suite of modules? You could type those *package* and *use base* bits over and over, or you could let *Exception::Class* do it for you. Here's an example adapted from the Mason code base:

```
use Exception::Class
( 'HTML::Mason::Exception' =>
  { description => 'generic base class for all Mason exceptions' },
  'HTML::Mason::Exception::Compiler' =>
```

```
{ isa => 'HTML::Mason::Exception',
  description => 'error thrown from the compiler' },

  'HTML::Mason::Exception::Compilation' =>
  { isa => 'HTML::Mason::Exception',
    description => "error thrown in eval of the code for a component" },

  'HTML::Mason::Exception::Compilation::IncompatibleCompiler' =>
  { isa => 'HTML::Mason::Exception::Compilation',
    description => "a component was compiled by a compiler/lexer
      . " with incompatible options. recompilation is needed" },
  );
```

This is an extremely abbreviated version of the declarations found in the *HTML::Mason::Exceptions* module, but it demonstrates one of the core features of *Exception::Class*: creating a hierarchy of exception classes from a simple declaration. Note that after each class name, we pass a hash reference containing various options describing the particular class we want created.

The *description* parameter is a generic description of that whole class of exceptions. If your code catches an exception of a class you didn't create, adding the description to a log message may be handy. The description is not required, however.

The *isa* parameter is used to declare the parent of a given class. In the last example, we create a hierarchy several levels deep. The base class of our hierarchy is *HTML::Mason::Exception*. Below that we have *HTML::Mason::Compiler* and *HTML::Mason::Compilation*. And then finally, we have *HTML::Mason::Exception::Compilation::IncompatibleCompiler*, which subclasses *HTML::Mason::Exception::Compilation*.

How *isa* is Handled

Exception::Class does its best to handle whatever you throw at it in the "*isa*" parameter. You are not required to list your classes in any specific order, which means that that this works fine:

```
use Exception::Class
( 'My::Exception::Class::Subclass::Deep' =>
  { isa => 'My::Exception::Class::Subclass' },

  'My::Exception::Class::Subclass' =>
  { isa => 'My::Exception::Class' },

  'My::Exception::Class',
);
```

Dave is the coauthor of *Embedding Perl in HTML with Mason* (O'Reilly and Associates) and can be reached at autarch@urth.org.

Also, if you want to use your own custom class as the base class, that also works:

```
package My::Exception::Class;

# no need to inherit from Exception::Class::Base if you don't want
# to, but you can.

sub new {
    ...
}

...

# in another file

use My::Exception::Class;
use Exception::Class

( 'My::Exception::Class::Subclass' =>
  { isa => 'My::Exception::Class' } );
```

You do need to make sure that your base class is loaded before you have *Exception::Class* create subclasses.

Aliases

In using exception classes, I've found that they can acquire some rather long and unwieldy names. The aforementioned *HTML::Mason::Exception::Compilation::IncompatibleCompiler* class name is a particularly grotesque example. Typing this more than once can overwhelm even the most iron-fingered typist, and besides that, it's incredibly easy to make a typo in the name. These typos won't be caught at compile time, so you only find out about them when that particular bit of code is executed.

Because of this, *Exception::Class* makes it easy to create an alias for the class, which is a function that raises the appropriate exception.

Instead of typing this:

```
HTML::Mason::Exception::Compilation::IncompatibleCompiler->throw
( error => ... );
```

we can type the much less finger-straining version:

```
wrong_compiler_error ...;
```

Of course, you need to tell *Exception::Class* to make an alias:

```
use Exception::Class
( ...,

  'HTML::Mason::Exception::Compilation::IncompatibleCompiler' =>
  { isa      => 'HTML::Mason::Exception::Compilation',
    alias    => 'wrong_compiler_error',
    description =>
    'a component was compiled by a compiler/lexer'
    . ' with incompatible options.  recompilation is needed' },
);
```

When an *alias* parameter is specified, *Exception::Class* creates a handy little subroutine in the caller:

```
sub wrong_compiler_error {
    HTML::Mason::Exception::Compilation::IncompatibleCompiler->throw(@_);
}
```

Because of the way Perl parses subroutines, if we write a call to that subroutine without parentheses, Perl will check that subroutine name at compile time. Isn't that nice?

Of course, if you define your exceptions in one package and want to use them from many others, you'll need to reexport those generated subroutines. More on this later when we look at the use of *Exception::Class* within an application.

Exception::Class is solely aimed at making it easier to declare hierarchies of exception classes

Fields

You'll often find that you want to add an extra field or two to an exception. For example, let's say that you are writing a parser and, if you encounter some bad syntax, you want to throw an exception. It would be very nice to include the name of the file and line number in the file where that error occurs. This could make debugging much easier because you could display the offending line when run in debugging mode.

Exception::Class allows you to declare that a particular exception class has one or more arbitrary fields. These fields can be set when throwing an exception and will be accessible later via an accessor with the same name as the field.

So let's define our exception class with some fields:

```
use Exception::Class
( 'My::Parser::Exception::Syntax' =>
  { alias => 'syntax_error',
    fields => [ 'filename', 'line_number' ] } );
```

Now when we throw the exception, we can set those fields:

```
My::Parser::Exception::Syntax->throw
( error      => 'Expected a florble after a glorp',
  filename   => $filename,
  line_number => $line_number );
```

And, of course, we can pass those fields to our handy *alias* subroutine:

```
syntax_error
error      => 'Expected a florble after a glorp',
filename   => $filename,
line_number => $line_number;
```

Later, we can get those values by calling *\$_->filename()* and *\$_->line_number()*.

Throwing and Catching Exceptions

As I mentioned earlier, if you want true *try/catch* syntax in Perl, you should take a look at either *Error.pm* or *Exception.pm*. In older

versions of Perl, using these modules could cause memory leaks because they made it very easy to create a nested closure. Some testing I did recently seems to indicate that this is no longer a problem with Perl 5.8.4, though nested closures aren't specifically mentioned in any of the changelogs for recent versions.

If you are using an older Perl, you are probably better off avoiding these modules because creating a memory leak with them is so easy.

Of course, you can do exception handling with just *eval* and *\$@*:

```
eval {
    open my $fh, ">$file"
    or My::Exception::System->throw
        ( error => "Cannot open $file for writing: $!" );
    ... # write something to the file
};

if ($?) {
    log_error( $@->error );
    exit;
}
```

But there are a couple of pitfalls in this example.

First of all, we can't assume that, just because *\$@* is true, that it contains an exception object. If the Perl interpreter itself threw an exception, then *\$@* would just be a plain string.

```
eval {
    open my $fh, ">$file"
    or My::Exception::System->throw
        ( error => "Cannot open $file for writing: $!" );
    print $fh 10 / $x;
    close $fh;
};

if ($?) {
    log_error( $@->error );
    exit;
}
```

If *\$x* happens to be 0, then *\$@* will contain something like "Illegal division by zero at foo.pl line 20." If you try to call the *error()* method on *\$@* now, your program will die with a really helpful message like this:

```
Can't locate object method "error" via package "Illegal division by zero at
foo.pl line 20."
```

Talk about confusing!

So you need to ensure that *\$@* is an object before calling methods on it. One way to do this is to set *\$_SIG{__DIE__}* to something like this:

```
sub make_exception_an_object {
    my $e = shift;
    ref $e && $e->can('rethrow')
        ? $e->rethrow
        : My::Exception::Generic->throw( error => $e );
}
```

Of course, this has several problems of its own. First, setting *\$_SIG{__DIE__}* is very, very global. Every single module you use in your code will be affected by this, which could cause some strange results. Even worse, there's no guarantee that some other code won't set it to a different value.

To be pedantic, I will point out that the test for *ref \$e* does not actually indicate that *\$e* is an exception object because it could

very well be a plain, unblessed reference. So, if you can't cleanly force *\$@* to always be an exception object, you're stuck testing for this when handling it:

```
use Scalar::Util qw(blessed);

eval {
    open my $fh, ">$file"
    or My::Exception::System->throw
        ( error => "Cannot open $file for writing: $!" );
    print $fh 10 / $x;
    close $fh;
};

if ($?) {
    if ( blessed $@ && $@->can('error') ) {
        log_error( $@->error );
    } else {
        log_error( $@ );
    }
    exit;
}
```

But wait, there are more problems. We need to be very careful about using *\$@* after calling *eval* because it can mysteriously disappear. Consider, if you will, the following:

```
if ($?) {
    log_error($@);
    warn $@ if DEBUG;
}
```

It's possible that you might turn on debugging, see the value of *\$@* logged, and then not see it printed via *warn*. In fact, *warn* might say something like "Warning: something's wrong at foo.pl line 23." Helpful, huh?

The reason this happens is that the *\$@* variable is a global across all packages, and it gets reset every time Perl enters an *eval* block. So, if our *log_error()* subroutine, or for that matter any code that it calls, uses an *eval* block, then *\$@* gets reset before the call to *warn* on the following line.

Because of that, I strongly recommend this idiom:

```
eval { ... };

my $e = $@;
if ($e) { ... }
```

If you copy *\$@* into another variable right away, you can guarantee it won't be lost.

Of course, you can always encapsulate this into a nice little subroutine:

```
eval { ... };

handle_error();

sub handle_error {
    return unless $@;
    my $e = $@;
    ...
}
```

You'll notice that I've consistently tested *\$@* for truth rather than defined-ness. This is because *\$@* is set to the empty string "" when it is empty, rather than *undef*. Of course, if I were really diligent, I would test for *length \$@*, in case someone does something like *die 0*.

Constructor Parameters

The method usually used to create a new exception is the `throw()` method, which constructs a new exception and then immediately calls `die` with that exception. You can also call `new()` to create a new exception if you want to do something else with it.

The constructor accepts several parameters. The most important parameter is the message you want to store in the exception. This can be passed in either the `error` or `message` parameter. As a special case, if one and only one parameter is passed to the constructor, then this is assumed to be the `message` parameter. This is done to make it possible to write code like this:

```
system_error "Cannot write to $file: $!";
```

If no message is given, then the `$!` variable is used, so you could even just write:

```
open my $fh, "$file" or system_error;
```

I'm not sure this is an idiom I'd use myself, but some may like it.

By default, when an exception object's `as_string()` method is called, it returns an error message without a stack trace. If you want this method's return value to include a stack trace, you can set the `show_trace` parameter to a true value when creating the object. If you always want objects of a certain class to include a trace, this can be controlled through a class method called `Trace()`, which will be covered later.

And as was shown earlier, any fields defined for the subclass being thrown can be set by passing them to the constructor:

```
My::Parser::Exception::Syntax->throw
( error      => 'Expected a florble after a glorp',
  filename   => $filename,
  line_number => $line_number );
```

Exception Object Methods

Once you've caught an exception object, you'll probably want to do something with it. Since `Exception::Class::Base` overloads stringification, you can always just treat an exception object as a string, which is handy for logging. It also means that an uncaught exception that causes a program to die will generate some sort of useful error message.

If you want to get at individual pieces of information, you can use the following methods:

- **`message()`, `error()`.** Both of these methods return the error/message associated with the exception. Note that this is not the same as calling `as_string()`, which may include a stack trace.
- **`pid()`.** Returns the PID at the time the exception was thrown.
- **`uid()`.** Returns the real user ID at the time the exception was thrown.
- **`gid()`.** Returns the real group ID at the time the exception was thrown.
- **`euid()`.** Returns the effective user ID at the time the exception was thrown.
- **`egid()`.** Returns the effective group ID at the time the exception was thrown.
- **`time()`.** Returns the time in seconds since the epoch at the time the exception was thrown.
- **`package()`.** Returns the package from which the exception was thrown.
- **`file()`.** Returns the file within which the exception was thrown.
- **`line()`.** Returns the line where the exception was thrown.
- **`trace()`.** Returns the `Devel::StackTrace` object associated with the object. This class also overloads stringification, so you probably don't need to worry too much about the methods it offers unless you want to walk through the stack trace frame by frame.

- **`description()`.** This method returns the description associated with the exception's class when the class was created. This can also be called as a class method.
- **`as_string()`.** Returns the error message in string form, something like what you'd expect from `die`. If the class or object is set to show traces, then the full trace is also included, and the result looks a lot like calling `Carp::confess()`.
- **`full_message()`.** This method is called by the `as_string()` method to get the message for the exception object. By default, this is the same as calling the `message()` method but may be overridden by a subclass. An example of why this is useful is shown later.

If you want true try/catch syntax in Perl, you should take a look at either Error.pm or Exception.pm

If you want to rethrow an exception object, you can call the `rethrow()` method. This is basically syntactic sugar for `die $exception`. This does not change any of the object's attribute values. You can also call the `show_trace()` method with a Boolean parameter to set whether a stack trace is included when the `as_string()` method is called.

Class Methods

Besides the `throw()` and `new()` constructors, `Exception::Class::Base` also offers several class methods that let you set certain behaviors on a per-class basis. These methods make use of the `Class::Data::Inheritable` module. That means that a subclass inherits the value set for a parent until the method is called for the subclass, at which point the subclass's value becomes independent.

The most important of these methods is `Trace()`, which allows you to control whether or not a stack trace is included in the value of the `as_string()` method for an object of a class. If `Trace()` is set to a true value, then the class and its children will default to including a trace.

The other two methods, `NoRefs()` and `RespectOverload()`, also take Booleans. They default to true and false, respectively. These methods control aspects of how `Devel::StackTrace` stores and serializes arguments when generating a string representation of a stack trace, and the defaults are probably acceptable for most uses. See the `Devel::StackTrace` and `Exception::Class` documentation for further details.

Overriding `as_string()` and `full_message()`

The default implementation of the `as_string()` method does not take into account any fields that might exist for a given exception class. If you want to include these when an object is stringified or when `as_string()` is called, the correct way to do this is to override `full_message()` instead of overriding `as_string()`.

By default, the `full_message()` method is equivalent to the `message()` method. If you recall the `My::Parser::Exception::Syntax` class shown earlier, we could override `full_message()` to include the file and line number where the syntax error occurred:

```
sub full_message {
    my $self = shift;
    return $self->message . ' at' . $self->filename
        . ' line ' . $self->line_number;
}
```

This is much easier than overriding `as_string()`, which you are encouraged not to do.

Using `Exception::Class` in an Application

If you are using `Exception::Class` in an application or a large suite of modules, you will probably want to declare all your exceptions in one module. Additionally, you may want to provide some utility functions related to your exceptions.

For example, you might want a function that provides an easy way of determining if you are looking at a specific subclass in your exception hierarchy. Here is an example drawn from my *Alzabo* module:

```
sub isa_alzabo_exception {
    my ($err, $name) = @_;
    return unless defined $err;

    if ($name) {
        my $class = "Alzabo::Exception::$name";
        return blessed $err && $err->isa($class);
    } else {
        return blessed $err && $err->isa('Alzabo::Exception');
    }
}
```

So, inside the *Alzabo* code, if I want to know if an exception is of a certain subclass, I can do this:

```
eval { .. };
my $e = $@;

if ( isa_alzabo_exception($err, 'System' ) ) {
    ...
} elsif ( isa_alzabo_exception($e, 'Params' ) ) {
    ...
} elsif ( isa_alzabo_exception($e) ) {
    ...
} else {
    # something else entirely
    ...
}
```

This function is exported by my *Alzabo::Exceptions* module whenever it is loaded. You may also want to reexport the functions created by `Exception::Class` when the `alias` parameter is used. This is easily done using the *Exporter* module:

```
package Alzabo::Exceptions;

use base 'Exporter';
use Exception::Class

( ...,

    'Alzabo::Exception::Logic' =>
    { isa => 'Alzabo::Exception',
```

```
alias => 'logic_exception',
},

    'Alzabo::Exception::NoSuchRow' =>
    { isa => 'Alzabo::Exception',
      alias => 'no_such_row_exception',
    },

    ...
);

@Alzabo::Exceptions::EXPORT_OK = ( 'logic_exception', 'no_such_row_exception' );
```

As long as you import these reexported aliases at compile time, you can call them without using parentheses, as was shown previously.

Integration with *Error.pm*

If you want to use exception classes created by `Exception::Class` with the `try/catch` syntax provided by *Error.pm*, you'll need to add the following hack to your code at some point:

```
push @Exception::Class::Base::ISA, 'Error'
unless Exception::Class::Base->isa('Error');
```

It's a nasty little hack, but it works.

Integration with *Exception.pm*

Because of the way *Exception.pm* implements exception throwing and handling, I am not sure if integrating it with `Exception::Class` is possible. My brief experiments resulted in failure. If someone figures out how to get them to play nice together, please let me know so I can include this in the `Exception::Class` documentation.

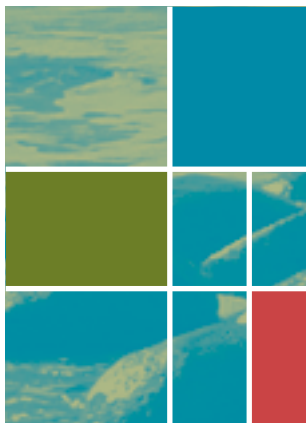
Practical Applications

David Wheeler created a handy module called `Exception::Class::DBI`, which integrates with DBI's error handling very nicely.

If you're interested in seeing how other people are using `Exception::Class`, there are a number of modules on CPAN to look at. My own *Alzabo* module uses `Exception::Class`, as does *HTML::Mason*. David Wheeler's *Bricolage*, a full-fledged CMS application, uses it as well. See the *Bric::Util::Fault* class to start with. If you want to test code that throws exceptions, take a look at Adrian Howard's *Test::Exception* module.

TPJ





Designing for Pluggability

Simon Cozens

Someone once said that the mark of a great piece of software is that it can be used to do things that its author never anticipated. The key to this is building sufficient flexibility into the application and, in this article, I'm going to show you some of the modules and techniques in my toolbox for doing just that, starting from relatively humble beginnings and ending with a new way of creating extensible database applications that I'm quite proud of.

UNIVERSAL::require

We begin with the `UNIVERSAL::require` module. This isn't so much related to extensibility itself, but it will be used as a building block for many of the other techniques we'll look at.

`UNIVERSAL::require` is a simple module and does a simple job. When you need to load some code at runtime (the essence of pluggable design), you find there are several ways to do it in Perl. You can use *do* or *string eval* if you know where the code is coming from, but what if you have a module name instead of a file name? You can't use *use* because that takes place at compile-time and you can't use *require \$module_name* because *require* with a variable, rather than a string constant, expects a filename, not a module name.

So, if we're trying to programmatically load an extension module at runtime—again, something we'll be doing a lot when developing pluggable software—we end up writing fudges like:

```
eval "require $module_name";
```

`UNIVERSAL::require` exists purely to tidy up this very case. It adds a *require* method to the `UNIVERSAL` namespace, meaning that we can call *require* on any class:

```
My::Module->require;
```

This method just does the *eval "require My::Module"* fudge with a little better error checking, so we can now say:

Simon is a freelance programmer and author, whose titles include Beginning Perl (Wrox Press, 2000) and Extending and Embedding Perl (Manning Publications, 2002). He's the creator of over 30 CPAN modules and a former Parrot pumpking. Simon can be reached at simon-cozens.org.

```
use UNIVERSAL::require;
$module_name->require;
```

This is the first step on the road to building our own extensible applications.

Do-It-Yourself Pluggability

The Perl module *Mail::Miner* analyzes a piece of e-mail for various features, which it stores in a database table. It does this by calling a set of “recognizers,” which are its plug-in modules. Here's how we load up the plug-in modules:

```
use File::Spec::Functions qw(:DEFAULT splitdir);

my @files = grep length, map { glob(catfile($_, "*.pm")) }
    grep { -d $_ }
    map { catdir($_, "Mail", "Miner", "Recogniser") }
    exists $INC["blib.pm"] ? grep { /blib/ } @INC : @INC;

my %seen;
@files = grep {
    my $key = $_;
    $key =~ s|.*Mail/Miner/Recogniser||;
    !$seen{$key}++
} @files;

require $_ for @files;
```

This is quite horrible but it's instructive to look at. We're trying to find all the files called “Mail/Miner/Recogniser/*something*.pm” in the include path, `@INC`, and the first `@files = statement` does this: It adds “Mail/Miner/Recogniser” to the end of each include path and checks to see if it is a directory. If it is, then we look for all the *.pm files in that directory.

The `blib.pm` bit is to be used for testing new recognizers. If we've said *use blib* somewhere, then we're in a test suite and we're only interested in the recognizer modules underneath the `blib` staging directory. This allows us to ensure that we're loading up the new modules instead of already installed ones. When we say *use blib*, or indeed any other module, Perl turns the module name into a short filename (`blib.pm`, say, or “Mail/Miner/Mail.pm”) and puts this in the `%INC` hash with the value being the full path of the

module file. Hence, looking in *%INC* is a good way of telling whether a particular module is loaded.

Next, we make sure we only have one copy of a given recognizer; this avoids problems when a module is installed in multiple places. Finally, we have a file name, so we can pass it to *require* and Perl will load the module.

So now we have loaded up all the *Mail::Miner::Recogniser::** modules that we can find on the system. That's solved one problem. The second problem is, now that we have them, what do we do with them? How do they relate to the rest of the system?

The way *Mail::Miner* opted to do this was to have each of the plug-in modules write into a hash when they load and supply meta-data about what they do:

```
package Mail::Miner::Recogniser::Phone;
$Mail::Miner::recognisers{"$_PACKAGE__"} =
{
    title => "Phone numbers",
    help  => "Match messages which contain a phone number",
    keyword => "phone"
};
```

Now *Mail::Miner* can look at the packages it has available in *%recognisers*, and call a particular interface on each one of them:

```
sub modules { sort keys %recognisers };

for my $module (Mail::Miner->modules()) {
    # ...
    $module->process(%hash);
}
```

This way, *Mail::Miner* can call out to additional installed modules without the author (me) knowing what plug-ins the user (you) has installed. Anyone can write a *Mail::Miner::Recogniser::Meeting* module; for instance, to attempt to identify meeting locations and times in an e-mail. Once it's installed in a Perl *include* path, it'll be automatically picked up and its *process* method will be called to examine an e-mail.

Module::Pluggable

As I said, that's how I used to do it—until *Module::Pluggable* appeared. We've seen the two problems involved in developing pluggable applications: first, finding the plug-ins; and second, working out what to do with them. *Module::Pluggable* helps with the first. It does away with all the nasty code we saw earlier. Now, to find all the recognizers installed in the *Mail::Miner::Recogniser* namespace, I can rewrite my code as follows:

```
package Mail::Miner;
use Module::Pluggable search_path => ['Mail::Miner::Recogniser'];
```

This gives me a *plugins* method that returns a list of class names just like the *modules* method did in our original code. If I wanted to make it completely compatible, I could also change the name of the method to *modules* with the *sub_name* configuration parameter:

```
use Module::Pluggable search_path => ['Mail::Miner::Recogniser'],
    sub_name => "modules";
```

This doesn't actually cause the modules to be loaded, so we could say:

```
$_->require for Mail::Miner->modules;
```

but we can also have the *modules* method itself load up the plug-ins by passing another configuration parameter:

```
use Module::Pluggable search_path => ['Mail::Miner::Recogniser'],
    sub_name => "modules",
    require => 1;
```

This is a drop-in—and much simpler—replacement for all the messing about with paths and *@INC* we saw earlier; it even handles the test case when *blib* is loaded. But I haven't replaced *Mail::Miner's* plug-in system with this and we'll see why later.

Class::Trigger allows you to add "trigger points" to your objects or classes to which third parties can attach code to be called

Making Callbacks with Class::Trigger

First, though, another CPAN module that handles the second problem—knowing what to do with your plug-ins when you have them. In *Mail::Miner*, we called a method that was assumed to be defined in all the plug-ins—whether or not they wanted it.

Sometimes this is the right way to do it, but often, an individual plug-in will want more control about what it responds to, especially if you're going to be calling your plug-ins on several different occasions for different things.

In these cases, you might find the CPAN module *Class::Trigger* a better fit. *Class::Trigger* allows you to add "trigger points" to your objects or classes to which third parties can attach code to be called.

For instance, if we have a method that displays some status information about an object, we could declare a trigger before printing the information out:

```
sub display_status {
    my $object = shift;
    my $message = $object->status;
    $object->call_trigger("display_status", \$message)
    print $message;
}
```

Now individual plug-in modules can register with the class subroutine references to be called when the trigger is called. For instance, a module might want to modify the message because it's going to be sent out as HTML:

```
use HTML::Entities;
MyClass->add_trigger( display_status => sub {
    my ($obj, $message) = @_;
    $message = encode_entities($message);
});
```

Notice how we pass in a reference to the message, so that we can modify the message. Another plug-in could provide links for all the URIs it finds in the message:


```

use URI::Find::Simple qw(change_uris);
MyClass->add_trigger( display_status => sub {
    my ($obj, $message) = @_;
    $$message = change_uris( $$message,
        sub { qq{<a href="$_[0]">$_[0]</a>} } );
});

```

And now we come to a problem—how do we know that the “mark up URIs” trigger is going to be loaded after the “escape HTML entities” trigger? If we can’t guarantee the ordering of the

We also want these plug-ins to specify some kind of relative order in which they’re called

two triggers, we could end up with our link tags denatured by the entity escaping.

This was a problem that I came up against, albeit from a slightly different angle.

Pluggable Callbacks in *Email::Store*

You see, the reason I haven’t rewritten *Mail::Miner* in the new plug-in style with *Module::Pluggable* is that I’ve been working on a much more extensible and advanced framework for storing and data-mining e-mail, which I’ve called *Email::Store*. In the way *Email::Store* works, pretty much everything is a plug-in.

When you store e-mail, *Email::Store* itself loads up the *Email::Store::Mail* plug-in, which sets up a placeholder database row for the mail. Then *Email::Store::Mail* calls all of the other plug-ins to examine the mail and file away the things they want to note about it—what mailing lists it came from, what attachments it has, and so on.

However, we also want these plug-ins to specify some kind of relative order in which they’re called. For example, it’s more efficient if the attachment handler strips the e-mail of its attachments before other plug-ins poke around in the e-mail body because, once you’ve gotten rid of the attachments, there’s less e-mail body to poke around in.

All great ideas have been had before, of course, and this made me think of the UNIX System V *init* process. When a UNIX machine starts up, it consults files in an “rc” directory to start up particular services. These files are named in a particular way so that, when the initialization process looks at the directory, it sees the services in the order that they should be started up. For instance, *S10syslogd* means “start the system logger at position 10,” and *S91apache* means “start the Apache web server at position 91,” the logger gets started first and Apache later. Now, this isn’t perfect because there can be several things in position 10, and they get run in alphabetical order; and besides, nobody’s policing the numbers anyway. If you think *S01foo* means “very early” and someone else comes along and installs *S00bar*, theirs gets run first. But it gives you a rough way of providing an order to the process.

What I wanted to do was give my plug-ins a similar rough ordering: Attachment handling had to happen at position 1; working out the mailing list an e-mail came from was a low priority task and could happen at position 90 towards the end; everything else could go somewhere in the middle.

I also didn’t really like the *Class::Trigger* approach of specifying a subroutine reference to be called. I prefer just writing methods. So, plug-ins that want to influence the way an e-mail gets indexed can provide two methods:

```

package Email::Store::Summary;

sub on_store_order { 80 }

sub on_store {
    my ($self, $mail) = @_;
    # ...
}

```

on_store_order is the position in which we’ll be called by the indexing process; *on_store* is what we do when we get called. This is implemented in the *::Mail* class like so:

```

use Module::Pluggable::Ordered search_path => ["Email::Store"];

sub store {
    my ($class, $rfc822) = @_;
    my $simple = Email::Simple->new($rfc822);
    my $msgid = $class->fix_msg_id($simple);
    my $self;
    $self = $class->create ({ message_id => $msgid,
        message => $rfc822,
        simple => $simple });
    $class->call_plugins("on_store", $self);
    $self;
}

```

Module::Pluggable::Ordered provides the same functionality as *Module::Pluggable* but also provides a *call_plugins* method: You give it a name of a trigger and some parameters and it looks through your plug-ins, finds those that provide that method, orders them by their positions, and then calls them. In our normal *Email::Store* case, that one line would be the equivalent of:

```

Email::Store::Attachment->on_store($self);
Email::Store::Entity->on_store($self);
Email::Store::Summary->on_store($self);
Email::Store::List->on_store($self);

```

As new modules are developed and dropped into place, they’re ordered by their *on_store_order* if they provide an *on_store* method and then placed into the list of *on_store* calls—all without *Email::Store::Mail* needing to know about them. The single *call_plugins* line combines both locating plug-ins and calling triggers to provide a facility for extending the indexing process.

Mixing Plug-Ins with Databases

Let’s now go on to write the rest of the *Email::Store::Summary* class that we looked at earlier. This is going to store summary information about an e-mail so that it can be displayed in a friendly way—we’ll store the subject of the mail and the first line of original content; that is, the first thing we see after removing an attribution and a quote. These will go in the summary database table, so we need to inherit from *Email::Store::DBI* the *Class::DBI* class that knows about the current database, and we need to tell it about the table’s columns:

```

package Email::Store::Summary;
use base 'Email::Store::DBI';

```

```
Email::Store::Summary->table("summary");
Email::Store::Summary->columns(All => qw/mail subject original/);
Email::Store::Summary->columns(Primary => qw/mail/);
```

We'll use *Text::Original*, a module extracted from the code of the Mariachi mail archiver, which hunts out the first piece of original text in a message body:

```
use Text::Original qw(first_sentence);

sub on_store_order { 80 }
sub on_store {
    my ($self, $mail) = @_;
    my $simple = $mail->simple;
    Email::Store::Summary->create((
        mail => $mail->id,
        subject => scalar($simple->header("Subject")),
        original => first_sentence($simple->body)
    ));
}
```

When e-mail is indexed, the *on_store* callback is called and it receives a copy of the *Email::Store::Mail* object that's being indexed. The *simple* method returns an *Email::Simple* object, which we use to extract the subject header and the body of the e-mail. Then we create a row in the summary table for this e-mail.

Next, for this to be useful, we need to tell *Email::Store::Mail* how this summary information relates to the mail:

```
Email::Store::Summary->has_a(mail => "Email::Store::Mail");
Email::Store::Mail->might_have(
    summary => "Email::Store::Summary" => qw(subject original)
);
```

Now an *Email::Store::Mail* object has two new methods—which, of course, we'll highlight in the documentation for our module. *subject* will return the first subject header and *original* will return the first sentence of original text. We use *might_have* to consider the summary table an extension of the mail table.

But now comes the clever bit. If this is truly to be a drop-in plug-in module, where is the summary table going to come from? It's one thing to be able to add concepts to a database-backed application, but these new concepts have to be supported by tables in the database. For the plug-in module to be completely self contained, it must also contain information about the table's schema. And this is precisely what *Email::Store* plug-ins do. In the DATA section of *Email::Store::Summary*, we'll put:

```
__DATA__
CREATE TABLE IF NOT EXISTS summary (
    mail varchar(255) NOT NULL PRIMARY KEY,
    subject varchar(255),
    original text
);
```

There's a mix-in module called *Class::DBI::DATA::Schema*, which is used by *Email::Store::DBI* (and hence anything that inherits from it), which provides the *run_data_sql* method. As its name implies, this method runs any SQL it finds in the DATA section of a class. So all we need to do is go through all of our plug-ins and run *run_data_sql* on them to create their tables:

```
sub setup {
    for (shift->plugins()) {
        $_->require or next;
```

```
    if ($_->can("run_data_sql")) {
        warn "Setting up database in $_\n";
        $_->run_data_sql ;
    }
}
```

With this in place, a plug-in module is truly self contained: It specifies what to do at trigger points like *on_store*, it specifies the relationships that tie it in to the rest of the *Email::Store* application, and it specifies how to create the database table that it relates to.

There's one more slight niggle—because the end user specifies what SQL database to use and because not all databases use the same variant of SQL, what if the schema in a DATA section isn't appropriate for what the end user is using? *Class::DBI::DATA::Schema* handles this, too, by using *SQL::Translator* to automatically translate the schema to a different variant of SQL. We can say

```
use Class::DBI::DATA::Schema (translate => [ "MySQL" => "SQLite" ] );
```

and write our DATA schemas in MySQL's SQL—except that we don't know at compile time that the end user is going to choose SQLite for his database; in fact, we don't know until the database is set up. So we end up doing something like this:

```
package Email::Store::DBI;
use base 'Class::DBI';
require Class::DBI::DATA::Schema;

sub import {
    my ($self, @params) = @_;
    if (@params) {
        $self->set_db(Main => @params);
        Class::DBI::DATA::Schema->import( translate =>
            [ "MySQL" => $self->__driver ]
        );
    }
}
```

When I say *use Email::Store 'dbi:SQLite:mailstore.db'*, *Email::Store::DBI* first sets up the database, and then it imports *CDBI::DATA::Schema*, telling it to translate between MySQL and SQLite, the *__driver* for our database. The reality is slightly more complex than this because we use *DBD::Pg* and *SQL::Translator* expects it to be called not "Pg" but "PostgreSQL." But the basics are there. See the source to *Email::Store::DBI* for the full story.

We've looked at various tools to increase the pluggability of our applications, from merely requiring classes at runtime to using modules to help us find plug-ins and provide trigger points or callbacks for extensions to influence the behavior of a process. We put all these together in *Module::Pluggable::Ordered*, which also allows us to specify a rough ordering for the extension modules, and we added the concept of extending a database-based application by using *Class::DBI::DATA::Schema* to allow us to write fully self-contained database-backed plug-ins.

Making your applications pluggable is an excellent way of reducing the complexity of a design—*Email::Store::Mail* hardly does anything itself but delegates to plug-ins for almost all of its functionality. *Module::Pluggable::Ordered* and the database techniques we've looked at provide a low-effort way of doing that and allow your applications to be stretched and expanded in ways you might not imagine!



Cleaning Up a Symlink Mess

Randal L. Schwartz

The box that hosts <http://www.stonehenge.com/> also takes care of <http://www.geekcruises.com/>, the company web site for my buddy, “Captain” Neil. As such, I have a dual role: I’m not only a frequent Geek Cruise attendee—I’m also the webmaster!

Recently, I noticed that Neil had moved a few pages around on his site to reorganize some of the information on past cruises. As quite a few links have been announced and bookmarked to the old location for a given page, he didn’t want to break those. So he naïvely placed a symbolic link from the old location to the new location. This means that a reference to the old location such as:

```
http://www.geekcruises.com/cruises/2003/perlwhirl3.html
```

would be the same as:

```
http://www.geekcruises.com/past_cruises/perlwhirl3.html
```

because he had moved the page as follows:

```
$ cd /data/web/geekcruises # the DocumentRoot for his server
$ cd cruises/2003
$ mv perlwhirl3.html ../past_cruises
$ ln -s ../past_cruises/perlwhirl3.html .
```

Now, at first glance, this appears to work. When either page is referenced, the same material is delivered by the server. However, there’s no way for anyone outside my server to know that these two pages are absolutely identical. This means that any cache (including browser caches, outward border caches at large organizations, or even our own reverse proxy cache) would now have two copies of the same material, having fetched the material needlessly twice.

Worse, some of the relative URLs are now somewhat broken. In the original location, getting back up to the index page requires `../index.html`, but in the new location, it was merely `../index.html`. It was for this reason that I actually noticed the symlinks in the first place, because a badly constructed web crawler was sucking down multiple copies of the web site, thinking that each `index.html` at the top level was different as well.

Randal is a coauthor of Programming Perl, Learning Perl, Learning Perl for Win32 Systems and Effective Perl Programming, as well as a founding board member of the Perl Mongers (perl.org). Randal can be reached at merlyn@stonehenge.com.

The correct way to move such a page that might have been bookmarked or indexed is to have Apache issue an *http redirect* when the old URL is referenced. For example, in the configuration file for the Geek Cruises website, we can add:

```
Redirect /cruises/2003/perlwhirl3.html
http://www.geekcruises.com/past_cruises/perlwhirl3.html
```

With this line in the configuration, a browser requesting the old URL will be asked to fetch the new URL instead. This redirect (also called an *external redirect*) is sufficient to ensure that caches will cache only one version (at the new URL) and indexers such as Google will invalidate the old URL over time.

Now, Neil doesn’t have direct access to the web-server configuration master file, but he can add `.htaccess` files in the various affected directories. That particular command can be placed directly into the `cruises/2003` subdirectory, and it would have the same result.

When I saw a few dozen of these symlinks all over the document tree for Neil’s server, I explained this to him and then said it’d actually be a small matter of programming to automatically replace all of those symlinks with updated `.htaccess` files. When all I heard was silence at the other end of the connection, I recognized that I’d need to write the program myself, since I’d now claimed it could be done. And that program is in Listing 1.

Lines 1 through 3 start nearly every program I write, enabling warnings for development, compiler restrictions (forbidding undeclared variables, symbolic references, and barewords), and turning off the pesky output buffering.

Lines 5 through 9 define my configuration parameters. The `$URL` is needed because an external redirect has to include the hostname and there’s no easy way to get at that from inside the `.htaccess` file. The `$USER` and `$GROUP` are the values for the newly created or updated `.htaccess` file; I’m running this as root so I have to set it correctly for Neil to be able to edit later. And `$MODE` gives the new permissions for a new `.htaccess` file.

Line 11 pulls in the `abs_path` routine from the core module `Cwd`. Lines 13 and 14 use my `File::Finder` module (found in CPAN) to easily get a list of directories below the document root.

Lines 17 to 60 iterate over each of those directories, which can be considered completely separately. Line 18 finds all the symbolic links within that directory using a simple `grep` over the result of a `glob`. Note that I’m presuming UNIX file syntax here, but that’s safe because I know my server box is not likely to ever

be anything but UNIX. Had I wanted this a bit more portable, I'd use *File::Spec* to construct the path.

Line 20 sets up the list of *@deletes*. These are the candidate symbolic links that are being replaced with *.htaccess* redirects, and can be deleted once the updated *.htaccess* file is in place. Line 21 computes the name of the *.htaccess* file for this particular directory.

Lines 23 to 45 process each symbolic link that was found in the directory separately. First, the target of the symbolic link is read in lines 24 and 25. If the *\$path* is not defined, it's either not a symbolic link or something went horribly wrong, and we ignore it.

Next, lines 26 and 27 ignore absolute symbolic links. I'm not sure why this code is in there, but it seemed to be the safest thing to do, since I only wanted to fix relative symbolic links. I've learned over the years that when you have root power, and you're mucking around with stuff and deleting and replacing a lot of files, it's safest to try to ignore everything that doesn't precisely fit your desired goal.

Line 28 uses *abs_path* to compute the resulting absolute path of the symbolic link target. Line 29 is left over from debugging, where I wanted to see if my calculations were correct for all of the existing links.

Lines 30 and 31 strip off the document root path from the source of the symbolic link. I need to do this to ensure that my *Redirect* command is framed in terms of URLs and not UNIX pathnames. The *\Q* quotes any metacharacters in the pathname. Again, this is a safe thing to do, even though I know there are no metacharacters in the particular paths I've configured at the top. Always be very conservative with root power.

Line 32 takes the matched tail part of the symbolic link source and builds the source URL for the *Redirect*. I'm presuming the *\$I* here has been properly set from the previous match and there's no possible way I'm using the value from a stale match.

Lines 33 through 35 repeat the stripping and building for the destination path, although I now have to create a full scheme-based URL for the path. Without the *http* prefix, Apache would have treated this operation as an *internal redirect*, with all the same problems as a simple symlink, because no indication would be sent to the client that something had moved.

Lines 36 to 42 create the *NEW* handle to which the new *.htaccess* file is written. This happens only once per directory because, after the first time, the *@deletes* array contains some previous entry. The existing *.htaccess* file (if any) is also copied to the beginning of the new *.htaccess* file. Note that we're using the *OLD* filehandle in a list context, so it gets slurped in as a list of lines, then immediately dumped to the new filehandle.

Line 43 writes the proper *Redirect* command to the new *.htaccess* file. Line 44 marks the symbolic link as one to be deleted once the *.htaccess* file is in place. Lines 46 to 59 are executed once per directory but only if some symbolic link was found that was eligible for removal.

First, line 47 closes the output file handle to ensure that the data is completely flushed. Then, lines 48 through 51 set the permissions and ownership on the new file to be defined by the configuration parameters at the top of the program. Lines 52 to 53 try to rename the existing *.htaccess* file out of the way to end in *.OLD*. Again, with Great Power comes Great Responsibility, including the mandate to have a Great Undo when one makes a Great Mistake. So, after each run of this program, I can verify that I have not completely mangled the *.htaccess* files, and then delete the *.OLD* files manually. And carefully.

Lines 54 and 55 move the newly created *.htaccess* file into place. Note that, because I've created the completed file in a separate location and then renamed it atomically into place, there's no chance that the live Apache process will read a partially written *.htaccess* file. This is a very important principle when dealing with live production activity. Finally, lines 56 to 58 delete the existing useless symbolic links one at a time. And that's all there is!

Oddly enough, as I was writing this article, I discovered yet another symlink in the tree. So I ran the code once again and, sure enough, it got replaced with the right redirect. Good thing I've kept this code around. Looks like it's time to send Neil another refresher message, or maybe just disable symbolic links in his tree. Until next time, enjoy!

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

Listing 1

```
=1=    #!/usr/bin/perl -w
=2=    use strict;
=3=    $|++;
=4=
=5=    my $URL = "http://www.geekcruises.com";
=6=    my $DIR = "/data/web/geekcruises";
=7=    my $USER = 2100;
=8=    my $GROUP = 2100;
=9=    my $MODE = 0644;
=10=
=11=    use Cwd qw(abs_path);
=12=
=13=    use File::Finder;
=14=    my @dirs = File::Finder->type('d')->in($DIR);
=15=
=16=    # print "\n" for @dirs;
=17=    for my $dir (@dirs) {
=18=        my @symlinks = grep -l, glob "$dir/**";
=19=        # print "$dir: @symlinks\n";
=20=        my @deletes;
=21=        my $htaccess = "$dir/.htaccess";
=22=
=23=        for my $symlink (@symlinks) {
=24=            defined(my $path = readlink($symlink)) or
=25=                warn("Cannot read $symlink: $!"), next;
=26=            $path =~ m{^/} and
=27=                warn("skipping absolute $path for $symlink\n"), next;
=28=            my $abs_path = abs_path("$dir/$path");
=29=            # print "$symlink -> $path => $abs_path\n";
=30=            $symlink =~ m{^\Q$DIR\E/(.*)}s or
=31=                warn("$symlink doesn't begin with $DIR"), next;
```

```
=32=    my $original_url = "/"$1";
=33=    $abs_path =~ m{^\Q$DIR\E/(.*)}s or
=34=        warn("$abs_path doesn't begin with $DIR"), next;
=35=    my $redirect_url = "$URL/$1";
=36=    unless (@deletes) {
=37=        ## print "in $dir...\n";
=38=        open NEW, ">$htaccess.NEW" or die;
=39=        if (open OLD, $htaccess) {
=40=            print NEW <OLD>;
=41=        }
=42=    }
=43=    print NEW "Redirect $original_url $redirect_url\n";
=44=    push @deletes, $symlink;
=45= }
=46= if (@deletes) {
=47=     close NEW;
=48=     chown $USER, $GROUP, "$htaccess.NEW"
=49=         or die "Cannot chown $htaccess.NEW: $!";
=50=     chmod $MODE, "$htaccess.NEW"
=51=         or die "Cannot chmod $htaccess.NEW: $!";
=52=     ! -e $htaccess or rename $htaccess, "$htaccess.OLD"
=53=         or die "Cannot mv $htaccess $htaccess.OLD: $!";
=54=     rename "$htaccess.NEW", $htaccess
=55=         or die "Cannot mv $htaccess.NEW $htaccess: $!";
=56=     for (@deletes) {
=57=         unlink $_ or warn "Cannot unlink $_: $!";
=58=     }
=59= }
=60= }
```

TPJ



High Performance MySQL

Tim Kientzle

Like many people, I first started using MySQL because it was a cheap and easily administered tool for development and prototyping. However, I have in the past had some misgivings about using MySQL on serious production systems. Since reading *High Performance MySQL*, however, I will have no difficulty recommending MySQL for highly demanding applications.

Under the Hood

Through the course of their book, Jeremy Zawodny and Derek Balling look at every stage of MySQL's operation, from query parsing and optimization, back to the storage engine responsible for a particular table, and finally to the operating system and hardware. For the most part, they present enough information for you to understand the issues involved without wasting your time on details that may not even be correct for the current release.

The authors' experience shows through clearly. In addition to presenting numerous concrete examples drawn from Zawodny's work at Yahoo! and elsewhere, they also manage to derive simple lessons from that experience. For example, the authors point out that query and index optimizations can often yield thousand-fold improvements in performance, while hardware upgrades generally offer much more modest increases. They also discuss what types of applications are most likely to become CPU bound or I/O bound.

The one area that was not as well covered as I would like is the impact of application architecture on database performance. I've personally seen enormous performance gains from application-level changes, and would like to read a good general discussion of what kinds of application-level changes are generally most effective and how those benefits compare to or complement the gains from schema, database, or operating-system optimizations.

Making Choices

Choosing a storage engine is a subtle task that the authors explain well. Roughly speaking, the default MyISAM tables provide good performance for read-heavy applications, while the various alternatives provide improved concurrency and transactional support. These trade-offs are introduced in an early chapter, but there are additional issues that come out later in the book. For example, based on the initial discussion, there would seem to be little reason to ever choose BDB over InnoDB, but a later chapter points out that InnoDB tables require more involved backup and restore strategies.

Choosing Your Tools

I especially appreciated the authors' practical approach to diagnosing problems. The book is sprinkled with numerous examples

Tim can be reached at kientzle@acm.org.

High Performance MySQL
Jeremy D. Zawodny
and Derek J. Balling
O'Reilly and Associates, 2004
294 pp., \$39.95
ISBN 0-596-00306-4

of *EXPLAIN SELECT* and *SHOW STATUS*. The authors also explain how to look at log files and performance counters and compare several simple benchmarking and administrative tools.

Putting It All Together

Of course, no server stands alone. Integrating a MySQL server into a larger system requires that you think about issues such as replication, load-balancing, backup, and security. These are exactly the topics that more introductory books often overlook.

Replication is a particularly key feature. It can be used to greatly improve performance of enterprise applications by putting a copy of the database close to each group of users. It can also simplify your backup strategy: It's much less disruptive to shut down a slave for backup than to shut down your master server. Of course, replication can also be used to dramatically improve performance by spreading query load across more hardware.

Finally, the authors give a good overview of MySQL security issues, including a tour of MySQL's privilege tables and a quick overview of how to use SSH and SSL to protect MySQL sessions. No doubt many readers will want more, but the sysadmin who has already spent some time thinking about security will find this section a good source of useful ideas.

Conclusion

This book is not a substitute for a good MySQL reference. You'll need the MySQL documentation or a book such as *Managing and Using MySQL* (O'Reilly and Associates, 2002) in which to look up detailed command syntax and configuration details.

But the clear presentation and practical, no-nonsense approach make this an excellent book. If you're currently managing a MySQL installation, you'll find plenty of practical advice that you can put to work immediately. If, like me, you've ever had misgivings about using MySQL in a critical application, this book will help you to better understand exactly how capable MySQL really is.

TPJ

Source Code Appendix

Robert Casey “Monitoring Network Traffic with *Net::Pcap*”

Listing 1

```
use Net::Pcap;
use NetPacket::Ethernet;
use NetPacket::IP;
use NetPacket::TCP;
use strict;

my $err;

# Use network device passed in program arguments or if no
# argument is passed, determine an appropriate network
# device for packet sniffing using the
# Net::Pcap::lookupdev method

my $dev = $ARGV[0];
unless (defined $dev) {
    $dev = Net::Pcap::lookupdev(\$err);
    if (defined $err) {
        die "Unable to determine network device for monitoring - ", $err;
    }
}

# Look up network address information about network
# device using Net::Pcap::lookupnet - This also acts as a
# check on bogus network device arguments that may be
# passed to the program as an argument

my ($address, $netmask);
if (Net::Pcap::lookupnet($dev, \$address, \$netmask, \$err)) {
    die "Unable to look up device information for ' ", $dev, ' - ', $err;
}

# Create packet capture object on device

my $object;
$object = Net::Pcap::open_live($dev, 1500, 0, 0, \$err);
unless (defined $object) {
    die "Unable to create packet capture on device ' ", $dev, ' - ', $err;
}

# Compile and set packet filter for packet capture
# object - For the capture of TCP packets with the SYN
# header flag set directed at the external interface of
# the local host, the packet filter of '(dst IP) && (tcp
# [13] & 2 != 0)' is used where IP is the IP address of
# the external interface of the machine. For
# illustrative purposes, the IP address of 127.0.0.1 is
# used in this example.

my $filter;
Net::Pcap::compile(
    $object,
    \$filter,
    '(dst 127.0.0.1) && (tcp[13] & 2 != 0)',
    0,
    $netmask
) && die "Unable to compile packet capture filter";
Net::Pcap::setfilter($object, $filter) &&
    die "Unable to set packet capture filter";

# Set callback function and initiate packet capture loop

Net::Pcap::loop($object, -1, \$syn_packets, '') ||
    die "Unable to perform packet capture";

Net::Pcap::close($object);

sub syn_packets {
    my ($user_data, $header, $packet) = @_;

    # Strip ethernet encapsulation of captured packet

    my $ether_data = NetPacket::Ethernet::strip($packet);

    # Decode contents of TCP/IP packet contained within
    # captured ethernet packet

    my $ip = NetPacket::IP->decode($ether_data);
    my $tcp = NetPacket::TCP->decode($ip->{'data'});
```

```

# Print all out where its coming from and where its
# going to!

print
$ip->{'src_ip'}, ":", $tcp->{'src_port'}, " -> ",
$ip->{'dest_ip'}, ":", $tcp->{'dest_port'}, "\n";
}

```

Randal L. Schwartz “Cleaning Up a Symlink Mess”

Listing 1

```

=1=      #!/usr/bin/perl -w
=2=      use strict;
=3=      $|++;
=4=
=5=      my $URL = "http://www.geekcruises.com";
=6=      my $DIR = "/data/web/geekcruises";
=7=      my $USER = 2100;
=8=      my $GROUP = 2100;
=9=      my $MODE = 0644;
=10=
=11=     use Cwd qw(abs_path);
=12=
=13=     use File::Finder;
=14=     my @dirs = File::Finder->type('d')->in($DIR);
=15=
=16=     # print "$_\n" for @dirs;
=17=     for my $dir (@dirs) {
=18=         my @symlinks = grep -l, glob "$dir/**";
=19=         # print "$dir: @symlinks\n";
=20=         my @deletes;
=21=         my $htaccess = "$dir/.htaccess";
=22=
=23=         for my $symlink (@symlinks) {
=24=             defined(my $path = readlink($symlink)) or
=25=                 warn("Cannot read $symlink: $!"), next;
=26=             $path =~ m{^/} and
=27=                 warn("skipping absolute $path for $symlink\n"), next;
=28=             my $abs_path = abs_path("$dir/$path");
=29=             # print "$symlink -> $path => $abs_path\n";
=30=             $symlink =~ m{^Q$DIR\E/(.*)}s or
=31=                 warn("$symlink doesn't begin with $DIR"), next;
=32=             my $original_url = "/"$1;
=33=             $abs_path =~ m{^Q$DIR\E/(.*)}s or
=34=                 warn("$abs_path doesn't begin with $DIR"), next;
=35=             my $redirect_url = "$URL/$1";
=36=             unless (@deletes) {
=37=                 ## print "in $dir...\n";
=38=                 open NEW, ">$htaccess.NEW" or die;
=39=                 if (open OLD, $htaccess) {
=40=                     print NEW <OLD>;
=41=                 }
=42=             }
=43=             print NEW "Redirect $original_url $redirect_url\n";
=44=             push @deletes, $symlink;
=45=         }
=46=         if (@deletes) {
=47=             close NEW;
=48=             chown $USER, $GROUP, "$htaccess.NEW"
=49=                 or die "Cannot chown $htaccess.NEW: $!";
=50=             chmod $MODE, "$htaccess.NEW"
=51=                 or die "Cannot chmod $htaccess.NEW: $!";
=52=             ! -e $htaccess or rename $htaccess, "$htaccess.OLD"
=53=                 or die "Cannot mv $htaccess $htaccess.OLD: $!";
=54=             rename "$htaccess.NEW", $htaccess
=55=                 or die "Cannot mv $htaccess.NEW $htaccess: $!";
=56=             for (@deletes) {
=57=                 unlink $_ or warn "Cannot unlink $_: $!";
=58=             }
=59=         }
=60=     }

```

TPJ