DAVID BLANK-EDELMAN

practical Perl tools: these inodes were made for walkin'

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

dnb@ccs.neu.edu

SINCE THIS IS THE SYSTEM ADMINISTRATION issue of *;login;*, I thought I'd take a break from covertly writing about sysadmin topics and overtly write about one instead. Today we're going to talk a bit about using Perl to walk filesystems. As filesystems continue to grow in size and complexity it helps to know what tools are available to make navigating these filesystems easier and more efficient.

(Ohhhh, I remember the good old days back when our filesystem had just two files: dot and dot-dot, and we were happy! . . .)

Best not to dilly or dally; we've got a lot of walking to do, so let's get right to it.

Putting One Foot in Front of the Other

The simplest way to begin to walk filesystems is to use the directory and file functions that are built into Perl:

```
opendir()
readdir()
closedir()
chdir()
```

Open a directory, read the contents of that directory, close the directory again, and then change directories to one of the subdirectories you just found. Repeat as necessary, most likely via recursion. Can't get much simpler than that.

This sort of approach appeals to the do-it-yourself crowd and those people who don't get enough recursion in their life (..their life..life..their life...in their life). In general I don't find using bare metal code like this particularly productive. It works fine for directory walks, where you might do something like this to find all of the files in a directory:

```
my $path = "/Users/dnb";
opendir my $DH, $path
      or die "Can't open $path: $!";
my (@files) = grep { -f "$path/$_" } readdir $DH;
closedir $DH
```

Here's a quick related aside that may keep you from banging your head against a wall someday: When first writing walking code of any sort, many people find themselves becoming very frustrated because their code only partially works. Close examination under a debugger shows that the

opendir(), closedir(), and even the readdir() all appear to be doing the right thing but for some reason their -f test doesn't return any results.

In my experience the most common reason for this is that the code they have written looks like this:

```
opendir my $DH, $path;
my (@files) = grep { -f } readdir $DH; # probably broken
```

This certainly looks as though it should work, since readdir() is correctly returning a list of the contents of $path. Pity we're not testing items in that directory, though!

We're performing the -f test on some (probably nonexistent) name in the script's *current directory* by mistake. If you want to perform a test such as -f on a file in some other directory, you need to specify that directory, as in our original code sample ("$path/$_").

If using the Perl built-in functions isn't the most productive way to spend your time, what is? Avid readers of this column know just where I'm going. Yup, it's module time.

File::Find

For a fairly long time, the File::Find module was the only game in town. And it is still a pretty good one. File::Find is shipped with Perl ("in the core"). It has steadily improved over the years, remaining a decent option for filesystems walking tasks.

Here's how it works: File::Find provides one subroutine, find(), which you call to start a walk from a specific path. Each time File::Find encounters something (e.g., a file or a directory) on its journey it calls a user-specified subroutine. This subroutine is responsible for doing all of the selection and disposition. It is the code that decides which items should be worked on and what to do with them. Here's a very simple example to make this a little clearer:

```
use File::Find;

find( \&wanted,'.' );

sub wanted { print "$File::Find::name\n" if ( -f $_ ); }
```

The first line says to begin the walk from the current directory and to call a subroutine called "wanted" each time it finds something. The wanted() subroutine checks to see whether the item in question is a file and, if it is, the full path for that name is printed. File::Find makes several useful variables such as $File::Find::name available for your wanted() subroutine while it is running, including $File::Find::dir (the current directory at that point in the walk) and $_ (the name of the item that was just found).

Your wanted() subroutine can be arbitrarily complex. Here's a contrived example that attempts a DNS lookup based on the name of every file found during the walk:

```
use File::Find;
use Net::DNS;

our $res = Net::DNS::Resolver->new;

find(\&wanted, "/data/suspect_hostnames");
```

```
sub wanted {
        next unless -f $File::Find::name;
        my $query = $res->search($_);
        print "ok $_" if defined $query;
}
```

This subroutine prints out the names of the files that successfully return an answer to the query. Notice that I said that wanted() *can* be arbitrarily complex, but here's the rub: It shouldn't be. This subroutine gets called for every single item in your filesystem, so it is incumbent upon you to write code to exit the subroutine as fast as possible. The code in the last example is a bad example of this, because DNS queries can take a relatively long time. If you can write a quick test that might lead to an early exit from the subroutine, by all means do it.

Before we move on to a better way to use File::Find, I do want to mention that the File::Find way of doing things was so compelling that Guido Flohr wrote a module that emulates it for use with complex data structures. In Data::Walk scalars are treated like files and lists and hashes are treated like directories. With Data::Walk you start a walk() of the data structure and a user-specified wanted() subroutine is called for each item in that structure.

File::Find::Rule

File::Find by itself makes the walking pretty easy through its simple interface. But this interface can be a bit too simple. It can make you work too hard to perform common tasks. For example, to collect a list of the files that are bigger than 2 MB you need to work out your own way to accumulate these results between calls to the wanted() subroutine. This means using global variables (yuck!), a caching/shared memory/persistence mechanism (pant, pant, hard work), or creating a closure ("Too much thinking give Oog a headache!"). To help make standard tasks like this easier there are a number of File::Find wrapper modules available. My favorite by far is the File::Find::Rule family of modules. This is the module I reach for most often for this sort of work (followed as a close second by the technique we'll see in the next section).

File::Find::Rule uses an object-oriented calling structure with a procedural interface also available if you'd prefer. This means you get to type lots of little arrows (->). To start off slow, here's the code that returns a list of all of the items in /var:

```
use File::Find::Rule;
my @items = File::Find::Rule->in('/var');
```

We can retrieve just the files or directories by chaining the appropriate method:

```
use File::Find::Rule;
my @files = File::Find::Rule->file()->in('/var');
my @dirs = File::Find::Rule->directory()->in('/var');
```

And that task we mentioned above—all the files above 2 MB in size—becomes as easy as pie:

```
use File::Find::Rule;
my @bigfiles = File::Find::Rule->size('>2M')->file()->in('/var');
```

Much more complex filtering expressions are also possible; see the File::Find::Rule documentation for more details.

If File::Find::Rule only provided a better interface to File::Find-like operations that would be enough reason to use it, but there's even more yummy goodness inside. I mentioned the File::Find::Rule *family* of modules before because File::Find::Rule provides an extension mechanism. Other File::Find::Rule::* modules can provide new predicates. A sample plug-in is File::Find::Rule::Permissions, which allows you to write things such as:

```
use File::Find::Rule::Permissions;
@files = File::Find::Rule::Permissions->file()->
        permissions(isReadable =>1, user => 'dnb')->in('/');
```

to determine which files in the filesystem are readable by the user "dnb." There are other extension modules that let you easily exclude CVS/SVN administrative directories (.cvs/.svn) based on image size or MP3 bitrate and so on. If you want to get really crazy, there's a module that can detect every time you've used a certain Perl operator in all of the scripts found in your filesystem:

```
# example from the documentation for File::Find::Rule::PPI
use File::Find::Rule ();
use File::Find::Rule::PPI ();

# Find all perl modules that use here-docs (<<EOF)
my $Find = File::Find::Rule->file
        ->name('*.pm')
        ->ppi_find_any('Token::HereDoc');
my @heredoc = $Find->in( $dir );
```

The Iterator Gang

File::Find::Rule may be one of my favorite tools for filesystem walking, but its default mode isn't always the best choice. For situations where you are dealing with a huge number of items, iterator-based modules can often offer a far better approach. (Note that File::Find::Rule also offers iterators.)

Iterators are well championed in the Perl community by Mark-Jason Dominus. The best exposition on the topic I have seen is in his excellent book *Higher-Order Perl*. In this book he says:

> An *iterator* is an object interface to a list.
>
> The object's member data consists of the list and some state information marking a "current" position in the list. The iterator supports one method, which we will call NEXTVAL. The NEXTVAL method returns the list element at the current position and updates the current position so that, the next time NEXTVAL is called, the next list element will be returned.

If you've used a module where you've called a next() method to get the next item back from some operation, for example an LDAP search, you've already dealt with iterators. In his book Dominus details a number of reasons why iterators are cool (and even shows you how to turn recursive code into iterator-based code). For our purposes iterator's most compelling argument comes into play when dealing with a huge filesystem. If you need to operate on the list off all of the files in a multi-terabyte filesystem, you don't want to use a module that will try to fill up all of your machine's memory with a massive list of names. An iterator-based module will let you work on that list one element at a time without having to store the whole thing at once. Clearly, this is a big win.

There are a number of Perl modules for filesystem walking that fit into this category; these include Iterator::IO, Find::File::Iterator, File::Walker,

Path::Class::Iterator, and File::Next. In the interests of time and space we'll only look at the last one.

File::Next is interesting because it describes itself as "lightweight, taint-safe . . . and has no non-core prerequisites" and also because of a new tool that makes use of it. We'll mention that tool in a second, but here's how File::Next gets used:

```
# example from the Find::Next documentation
use File::Next;
my $iter = File::Next->files( '/tmp' );

while ( my $file = $iter->() ) {
        print $file, "\n";
}
```

Simple, no? To make the iterator do more complex filtering, more parameters can be passed to the files() method. There is a similar dirs() method (although no methods are available to return special file types such as fifos).

It is possible to do some pretty powerful stuff with this module. The module's author has built a utility called Ack based on it. Ack (or App::Ack if you want to install it from CPAN) is a souped-up grep-like program that knows how to search filesystems in a more intelligent fashion. It knows to ignore certain files by default (e.g., backup files and core dumps) and can be told to search only certain types of files. Ack help types shows:

```
--[no]asm       .s .S
--[no]binary    Binary files, as defined by Perl's -B op (default: off)
--[no]cc        .c .h .xs
--[no]cpp       .cpp .m .h .C .H
--[no]csharp    .cs
--[no]css       .css
--[no]elisp     .el
--[no]haskell   .hs .lhs
--[no]html      .htm .html .shtml
--[no]java      .java
--[no]js        .js
--[no]lisp      .lisp
--[no]mason     .mas
--[no]ocaml     .ml .mli
--[no]parrot    .pir .pasm .pmc .ops .pod .pg .tg
--[no]perl      .pl .pm .pod .tt .ttml .t
--[no]php       .php .phpt
--[no]python    .py
--[no]ruby      .rb .rhtml .rjs
--[no]scheme    .scm
--[no]shell     sh .bash .csh .ksh .zsh
--[no]sql       .sql .ctl
--[no]tcl       .tcl
--[no]tex       .tex .cls .sty
--[no]tt        .tt .tt2
--[no]vim       .vim
--[no]xml       .xml .dtd .xslt
--[no]yaml      .yaml .yml
```

Ack offers a number of handy features such as color highlighting and full use of Perl regular expressions in addition to this filetype recognition.

Now that you've seen three approaches to filesystem walking and an application built on one of those approaches, I think it is time for you to hit the filesystem and start walkin'. Take care, and I'll see you next time.