Expecting Perl

by Mark Schoonover
mark.schoonover@gmail.com

Imagine having to change the password on hundreds of servers. I could log in to each one individually and hope to get done before I retire, or I could use Perl with `Expect.pm` to accomplish this very quickly, with less error, by automating it. Anything I can manually do on the command line I can automate with the `Expect` module.

■ Introduction

Expect was originally a little language written for Tcl to pretend to be someone. From the Tcl website:

> *Expect is a tool primarily for automating interactive applications such as telnet, ftp, passwd, fsck, rlogin, tip, etc.*

For this article, I won't be using Tcl, but Perl with `Expect.pm`. Perl and Expect.pm make a powerful combination to automate the interaction, data collection and modification of servers, switches, routers, on just about any network device that has a command-line interface and remote access.

■ Getting Started

As an example, imagine me as a new system administrator tasked with inventorying every one of these servers. Of course, my predecessor didn't keep any inventory, and I only have host names and passwords. I have to figure out which distribution and kernel version of each server. Perl with `Expect.pm` to the rescue!

Expect simulates the human interaction, so it's very important I know the command line interface of my system. As an example, when I use `ssh` to login to a server, I'm expecting a password prompt (unless I'm using `ssh` keys). `Expect.pm` sends commands to the system and expects data to come back. It's best to manually try all the commands I want to automate before developing my `Expect.pm`-based program. When I do this first, I mostly eliminate command errors when developing my Expect script.

■ Connecting and Logging In

I can connect to my target system in several ways, including various Perl modules such as `Net::SSH`, `Net::SSH::Expect`, or `Net::Telnet`, *etc.* SSH is installed on my systems, so I use that directly. I construct a command, just like I would if I were doing this myself. I execute that command with the `spawn` method:

```
use Expect;

my $expect = Expect->new;

my $login =
    "/usr/local/bin/ssh $user\@$host";
my $expect = Expect->spawn($login)
  or die
    "Can't login to $host as $user: $!\n"
```

Where `$host` and `$user` are valid host and user names. If this method call dies, I won't be able to send any commands to the host. To help in debugging my Expect scripts, I enable logging by setting `$Expect::Exp_Interval` to a true value:

```
#Enables internal debugging
$expect->exp_Interval( 1 );
```

Additionally, I can redirect the logging output to a file:

```
#Record all communication to logfile.
    $expect->log_file("$host.log");
```

The log file will contain all communication between my script and the device. I watch this file to see what's going on with my program. I can use the tail command to watch the file as it's updated:

```
shell> tail example.com.log
```

If this is the first time I'll be logging into a system using SSH, I'll have to deal with the authenticity banner that tells me I'm connecting to a new host:

```
The authenticity of host ' 192.168.0.1
(192.168.0.1)' can't be established
RSA key fingerprint is 84:36:d7:18:09:ec:84
:04:a7:ca:78:49:0d:68:39:b0.
Are you sure you want to continue
connecting (yes/no)?
```

I use the `expect` method with a regular expression to attempt to match this text:

```
$expect->expect(5, '-re', '(RSA)');
```

Install Expect: `$prompt> cpan Bundle::Expect Test::Expect`

The `expect` method can take several parameters. In the example, `expect` times out after 5 seconds, uses a regular expression since I used `-re`, and looks for `RSA` in the text it gets. It will store `RSA` in an array that I can access with the `matchlist` method. The parentheses in my regular expression work like those in a Perl regular expression and capture what they enclose. I'll find what the regular expression captured in the list that `matchlist()` returns:

```
my @matchlist = $expect->matchlist;
```

I can use the standard regex character classes such as `\d` and `\w`: I can use any valid Perl regular expression.

If I saw that authenticity banner, I need to respond to it to go onto the next step. I need to send `yes` back to the server to continue connecting, but only if I matched something:

```
$expect->send("yes\n")
  if( ($expect->matchlist)[0] );
```

The send method will send yes back to the server. This normally won't show up in the log file, but if my server echoes commands, it might appear in the log. I have to remember not to forget the newline, `\n`. Sending `yes` back to the server isn't enough; it would sit there waiting for the newline. It's actually waiting for a carriage return, but Expect handles the translation.

The next piece of data I expect is the password prompt, which I treat like I handled the authenticity banner earlier:

```
$expect->expect(5, "password: ");
```

This is slightly different than above since I'm searching for an exact string instead of using a regex. I don't need to capture anything; I just need to respond to it. I send the password using the `send` method, and remember to include the newline:

```
$expect->send("JaPh2007!\n");
```

Note that putting passwords in scripts is not good security practice and I only did it for this demonstration. I can also configure `ssh` to use shared keys.

■ Additional Matching ----------

Expect provides additional before and after matching using the `before()` and `after()` methods. These are just like `$`` and `$'` variables that give the parts of the strings before and after the matched portion. This can come in handy if I need all the text up to a certain point. In the authenticity banner example earlier, I could have looked for "key" and saved all the text leading up to that:

```
$expect->send( "some command" );
$expect->expect(5, "-re", "key");

my $before = $expect->before();
```

If the regex fails to match, `before` returns all the text. The `$expect->after()` method matches all the text after "key" until the end of the output. If this match fails, it'll return `undef`. I can find all of the matched text in the first element of `matchlist`:

```
my $matched =($expect->matchlist)[0]
```

■ Putting Expect to Work----------

Now that I can login to the server, it's a matter of using the same techniques to record the Linux distribution and kernel version that's running. I look in the `/etc/lsb-release` file, which looks similar to this:

```
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE= 7.10
DISTRIB_DESCRIPTION="Ubuntu 7.10"
```

I'll get the file contents by sending the `cat` command:

```
$expect->send("cat /etc/lsb-release\n");
```

I want to capture the "Ubuntu 7.10". Since it's the only line with double-quotes, it'll be easy enough to capture with a simple regular expression:

```
$expect->expect(5, "\"(.*)\"");
```

I get the description by accessing what I matched:

```
my $distro = ( $expect->matchlist )[0];
```

To get the kernel version I send the `uname` command:

```
$expect->send("uname -r\n");
$expect->expect(5, "(.*)");
my $kernel = ( $expect->matchlist )[0];
```

Now that I've extracted the description, I can store that in a database or do anything else with it I like. I take what I've done for one machine and run a loop to do it for all of the host names. With a little bit of Perl I've finished my task before lunch!

■ References----------

The original `expect` by Don Libes, *http://expect.nist.gov/*

Expect Perl Module, *http://search.cpan.org/dist/Expect/*

■ About the Author----------

Mark Schoonover lives near San Diego, California with his wife, three boys, a neurotic cat, and a retired Greyhound. He's experienced as a DBA, system administrator, network engineer and web developer. He enjoys Amateur Radio, running marathons and cycling. He can also be found coaching youth soccer, and getting yelled at as a referee on the weekends.