Perl como herramienta de administración

Martín Ferrari

16 de Julio de 2003

Resumen

Perl nació como herramienta de uso general para un ambiente UNIX, nada más apropiado para un administrador de sistemas. Por supuesto, Perl creció tanto como UNIX, y así las posibilidades ridículas de simplificar nuestro trabajo.

1. Introducción

Perl, en sus inicios fue el fruto de las frustraciones provocadas por las limitaciones de sed, awk y el shell. No es de extrañar que quienes usamos estas herramientas por años, cuando descubrimos Perl lo amamos instantáneamente.

La primera gran emoción sea quizás olvidarse para siempre del encomillado de variables, que nos ha hecho perder valiosos momentos que pudimos usar jugando nethack. En Perl, hay cosas propias de un lenguaje de *scripts* que nos permiten hacer el trabajo rápido y sencillo; a la vez que tenemos la confiabilidad de un lenguaje que puede manejar sin pestanear variables de contenido y tamaño irrestrictos, nos brinda velocidades increíbles y para colmo tenemos bibliotecas para casi toda utilidad imaginable.

2. Rápido y sucio

Muchas veces nos encontraremos escribiendo programas de una línea sin siquiera grabarlos en un archivo. Para eso usamos el modificador -e del comando perl que toma el siguiente parámetro cómo código fuente a ejecutar, también podemos importar módulos con el modificador -M.

Supongamos que necesitamos bajar el contenido de una página de manera automática, por ejemplo para verificar funcionamiento de un servidor web. Usamos del módulo `LWP::Simple` la función `getprint` que simplemente obtiene una URL y la imprime:

```
# perl -MLWP::Simple -e 'getprint "http://www.cpan.org/"'
```

No tenemos por qué limitarnos a páginas web, ni siquiera al protocolo HTTP. Supongamos que necesitamos mantener una copia local actualizada de definiciones de virus, por supuesto no para nosotros sino para los pobres usuarios que tienen que convivir con Windows; podríamos programar para que se ejecute diariamente el siguiente comando, que además muestra que perl puede manejar datos binarios de manera transparente:

```
# perl -MLWP::Simple -e 'open(OUT, ">", "Definiciones.dat");
    $contenido = get "ftp://ftp.antivirus.com/ultimo.dat";
    print OUT $contenido; close(OUT)'
```

3. Un poco de poder

Veamos ahora algunos ejemplos un poco más complejos

3.1. Envío de mail con archivos adosados

Enviar un mail con archivos de manera automatizada siempre es un problema, ya que el manejo de MIME es complejo y no es recomendable intentar hacerlo en cada script que escribamos.

Este programa toma como argumentos el destinatario, el motivo del mensaje y el/los archivos a enviar y se encarga diligentemente de la tarea. Para ello usa el módulo MIME::Entity, perteneciente a MIME-tools.

```
#!/usr/bin/perl
# sendattach.pl: Envío de mail con archivos adosados

# Nos forzamos a ser prolijos
use strict;
use warnings;

# Módulo que hace la magia
use MIME::Entity;

# Recibo los argumentos de línea de comando
die "Uso: $0 <email> <subject> <archivo1> <archivo2> ...\n"
```

```
    unless (@ARGV >= 3);
my($email, $subject, @files) = @ARGV;

# Compongo el cuerpo vacío
my($mail) = MIME::Entity->build(
    To => $email,
    Type => "text/plain",
    Subject => $subject,
    Data => "");

# Agrego cada archivo
foreach(@files) {
    $mail->attach(
        Disposition => "attachment",
        Type => "application/octet-stream",
        Path => $_)
        or die "Problemas con $_";
}

# Envío el mail listo
$mail->smtpsend();
```

3.2. Backups por la red

A veces sucede que necesitamos hacer un backup de una máquina en el disco rígido de otra. El problema es cómo mandar los datos, FTP es incómodo, SCP es lento, RCP es peligroso.

Este ejemplo implementa un servidor TCP que no necesita correr con privilegios, que recibe por la red el nombre de archivo a grabar y su contenido, toma el recaudo mínimo de limpiar al nombre de archivos de caracteres comprometedores (caracteres de control, barras y espacios) y graba los datos en el archivo solicitado dentro de un directorio predeterminado.

Código del servidor:

```
#!/usr/bin/perl

use warnings;
use strict;

use IO::Socket;
```

```
# Puerto en el que voy a escuchar
my($server_port) = 8000;
# Directorio donde voy a guardar los archivos
my($outdir) = "./backups";

my($server, $cliente, $arch);

# Creo el servidor
$server = IO::Socket::INET->new(
        LocalPort => $server_port,
        Type      => SOCK_STREAM,
        Listen    => 5,
        Reuse     => 1)
    or die "No puedo tomar el puerto $server_port: $@\n";

# Espero una conexión
while ($cliente = $server->accept()) {
    print STDERR "Conexión de ", $cliente->peerhost(), "\n";
    # Leo la primera línea, reinicio si no lei nada.
    $arch = <$cliente>;
    unless($arch) {
        warn "Conexión cerrada prematuramente\n";
        next;
    }
    # Limpio de caracteres de control
    $arch =~ tr#\000-\037##d;
    # Reemplazo barras y espacios
    $arch =~ tr#\\/ #___#;
    # Abro el archivo de salida
    print STDERR "Grabando en $outdir/$arch\n";
    unless(open(FILE, ">", "$outdir/$arch")) {
        warn "No puedo crear $outdir/$arch: $!\n";
        next;
    }
    # Bajo el contenido
    while(<$cliente>) {
        print FILE $_;
    }
    close(FILE);
}
```

```
close($server);
```

El cliente es muy sencillo, toma por línea de comando los datos del servidor y el nombre de archivo a grabar y luego copia la entrada estándar al servidor remoto.

Código del cliente:

```
#!/usr/bin/perl

use warnings;
use strict;

use IO::Socket;

die "Uso: $0 <host> <puerto> <archivo a grabar>\n"
    unless(@ARGV == 3);
my($host, $port, $file) = @ARGV;

my($remote) = IO::Socket::INET->new(
    Proto => "tcp",
    PeerAddr => $host,
    PeerPort => $port
    )
    or die "error conectando a $host/$port :$!";

print $remote "$file\n";
while ( <STDIN> ) { print $remote $_ }
```