¿Qué son las expresiones regulares?

En pocas palabras, las expresiones regulares son notaciones simbólicas que se utilizan para identificar patrones en el texto. En cierto modo, se parecen al método comodín del shell para hacer coincidir los nombres de archivo y ruta, pero en una escala mucho mayor. Las expresiones regulares son compatibles con muchas herramientas de línea de comandos y con la mayoría de los lenguajes de programación para facilitar la solución de problemas de manipulación de texto. Sin embargo, para confundir aún más las cosas, no todas las expresiones regulares son iguales; varían ligeramente de una herramienta a otra y de un lenguaje de programación a otro. Para nuestra discusión, nos limitaremos a las expresiones regulares como se describe en el estándar POSIX (que cubrirá la mayoría de las herramientas de línea de comandos), a diferencia de muchos lenguajes de programación (más notablemente Perl), que usan conjuntos de notaciones un poco más grandes y ricos. .

grep

El programa principal que usaremos para trabajar con expresiones regulares es nuestro viejo amigo grep. El nombre grep en realidad se deriva de la frase "impresión de expresión regular global", por lo que podemos ver que grep tiene algo que ver con expresiones regulares. En esencia, grep busca en archivos de texto texto que coincida con una expresión regular especificada y genera cualquier línea que contenga una coincidencia con la salida estándar.

Hasta ahora, hemos usado grep con cadenas fijas, así:

```
[~]$ ls /usr/bin | grep zip
```

Esto lista todos los archivos en el directorio /usr/bin cuyos nombres contengan la subcadena zip.

el progarma grep acepta opciones y argumentos de esta manera, donde regex es una expresion regular

```
[-]$ grep [options] regex [files ..]
```

En la tabla siguiente describe las opciones gred más comunes

Opción	Opción larga	Descripción
-i	--ignore-case	Ignorar caso. No distinga entre caracteres en mayúscula y minúscula.
-v	--invert-match	Invertir coincidencia. Normalmente, grep imprime líneas que contienen una coincidencia. Esta opción hace que grep imprima cada línea que no contenga una coincidencia.
-c	--count	Imprima el número de coincidencias (o no coincidencias si también se especifica la opción -v) en lugar de las líneas en sí.

Opción	Opción larga	Descripción
-l	--files-with-matches	Imprima el nombre de cada archivo que contiene una coincidencia en lugar de las líneas mismas.
-L	--files-without-matches	Como la opción -l, pero imprime solo los nombres de los archivos que no contienen coincidencias.
-n	--line-number	Prefije cada línea coincidente con el número de la línea dentro del archivo.
-h	--no-filename	Para búsquedas de varios archivos, suprima la salida de nombres de archivo.

Para explorar grep más a fondo, creemos algunos archivos de texto para buscar.

```
[~]$ ls /bin > dirlist-bin.txt

[~]$ ls /usr/bin > dirlist-usr-bin.txt

[~]$ ls /sbin > dirlist-sbin.txt

[~]$ ls /usr/sbin > dirlist-usr-sbin.txt

[~]$ ls dirlist*.txt
```

```
dirlist-bin.txt
dirlist-sbin.txt
dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

Podemos realizar una búsqueda simple de nuestra lista de archivos como esta:

```
[~]$ grep bzip dirlist*.txt
```

```
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

En este ejemplo, grep busca en todos los archivos listados la cadena bzip y encuentra dos coincidencias, ambas en el archivo dirlist-bin.txt. Si solo estuviéramos interesados en la lista de archivos que contienen coincidencias en lugar de las coincidencias en sí, podríamos especificar la opción -l.

```
[~]$ grep -l bzip dirlist*.txt
```

```
dirlist-bin.txt
```

Por el contrario, si quisiéramos ver solo una lista de los archivos que no contienen una coincidencia, podríamos hacer esto:

```
[~]$ grep -L bzip dirlist*.txt
```

```
dirlist-sbin.txt
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

Metacaracteres y literales

Si bien puede que no parezca evidente, nuestras búsquedas grep han estado usando expresiones regulares todo el tiempo, aunque muy simples. La expresión regular bzip se considera que significa que se producirá una coincidencia solo si la línea del archivo contiene al menos cuatro caracteres y que en algún lugar de la línea los caracteres b, z, i y p se encuentran en ese orden, sin otros personajes en el medio.

Los caracteres de la cadena bzip son todos caracteres literales, ya que coinciden con ellos mismos. Además de los literales, las expresiones regulares también pueden incluir metacaracteres que se utilizan para especificar coincidencias más complejas. Los metacaracteres de expresión regular constan de lo siguiente:

- [ ] { } - ? * + ( ) | \

Todos los demás caracteres se consideran literales, aunque el carácter de barra invertida se utiliza en algunos casos para crear metasecuencias, además de permitir que los metacaracteres se escapen y se traten como literales en lugar de interpretarlos como metacaracteres.

Como podemos ver, muchos de los metacaracteres de expresión regular también son caracteres que tienen significado para el shell cuando se realiza la expansión. Cuando pasamos expresiones regulares que contienen metacaracteres en la línea de comando, es vital que estén entre comillas para evitar que el shell intente expandirlos.

El Carácter cualquiera.

El primer metacarácter que veremos es el carácter de punto (.) o caracter periodico, que se usa para hacer coincidir cualquier carácter. Si lo incluimos en una expresión regular, coincidirá con cualquier carácter en esa posición de carácter. Aquí tienes un ejemplo:

```
[~]$ grep -h '.zip' dirlist*.txt
```

Buscamos cualquier línea en nuestros archivos que coincida con la expresión regular .zip. Hay un par de cosas interesantes que destacar sobre los resultados. Observe que no se encontró el programa zip. Esto se debe a que la inclusión del metacarácter de punto en nuestra expresión regular aumentó la longitud de la coincidencia requerida a cuatro caracteres, y debido a que el nombre zip contiene solo tres, no coincide. Además, si alguno de los archivos de nuestras listas hubiera tenido la extensión de archivo .zip, también

habría coincidido, porque el carácter de punto en la extensión del archivo coincidiría con el "cualquier carácter" también.

Anclas

El signo de intercalación (^) y el signo de dólar ($) se tratan como anclas en las expresiones regulares. Esto significa que hacen que la coincidencia se produzca solo si la expresión regular se encuentra al principio de la línea (^) o al final de la línea ($).

```
[~]$ grep -h '^zip' dirlist*.txt
```

```
...
```

```
[~]$ grep -h 'zip$' dirlist*.txt
```

```
...
```

```
[~]$ grep -h '^zip$' dirlist*.txt
```

```
...
```

Aquí buscamos en la lista de archivos el zip de cadena ubicado al principio de la línea, al final de la línea y en una línea donde está tanto al principio como al final de la línea (es decir, por sí mismo en el línea). Tenga en cuenta que la expresión regular ^ $ (un principio y un final sin nada en el medio) coincidirá con las líneas en blanco.

Expresiones entre corchetes y clases de caracteres

Además de hacer coincidir cualquier carácter en una posición determinada en nuestra expresión regular, también podemos hacer coincidir un solo carácter de un conjunto específico de caracteres mediante el uso de expresiones de corchetes. Con las expresiones de corchetes, podemos especificar un conjunto de caracteres (incluidos los caracteres que de otro modo se interpretarían como metacaracteres) para que coincidan. En este ejemplo, usando un conjunto de dos caracteres, hacemos coincidir cualquier línea que contenga la cadena bzip o gzip:

```
[]$ grep -h '[bg]zip' dirlist*.txt
```

```
bzip2
bzip2recover
gzip
```

Un conjunto puede contener cualquier número de caracteres y los metacaracteres pierden su significado especial cuando se colocan entre corchetes. Sin embargo, hay dos casos en los que los metacaracteres se utilizan dentro de expresiones entre corchetes y tienen significados diferentes. El primero es el signo de intercalación (^), que se utiliza para indicar negación; el segundo es el guión (-), que se utiliza para indicar un rango de caracteres.

Negación [^]

Si el primer carácter de una expresión entre corchetes es un signo de intercalación (^), los caracteres restantes se toman como un conjunto de caracteres que no deben estar presentes en la posición de carácter dada. Hacemos esto modificando nuestro ejemplo anterior, de la siguiente manera:

```
[~]$ grep -h '[^bg]zip' dirlist*.txt
```

```
bunzip2
gunzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

Con la negación activada, obtenemos una lista de archivos que contienen la cadena zip precedida por cualquier carácter excepto bo g. Observe que no se encontró el archivo zip.

Un conjunto de caracteres negado todavía requiere un personaje en la posición dada, pero el personaje no debe ser miembro del conjunto negado.

El carácter de intercalación invoca la negación sólo si es el primer carácter dentro de una expresión de corchetes; de lo contrario, pierde su significado especial y se convierte en un personaje ordinario del conjunto.

Rangos de caracteres tradicionales

Si quisiéramos construir una expresión regular que encontrara todos los archivos de nuestras listas que comienzan con una letra mayúscula, podríamos hacer esto:

```
[~]$ grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXZY]' dirlist*.txt
```

Es solo cuestión de poner las 26 letras mayúsculas en una expresión entre corchetes. Pero la idea de todo ese tipo de escritura es profundamente preocupante, así que aquí hay otra forma.

```
[~]$ grep -h '^[A-Z]' dirlist*.txt
```

Utilizando un rango de tres caracteres, podemos abreviar las 26 letras. Cualquier rango de caracteres se puede expresar de esta manera, incluidos varios rangos, como esta expresión que coincide con todos los nombres de archivo que comienzan con letras y números:

```
[~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

En los rangos de caracteres, vemos que el carácter de guión se trata de forma especial, entonces, ¿cómo incluimos realmente un carácter de guión en una expresión de corchetes? Haciéndolo el primer carácter de la expresión. Considere estos dos ejemplos:

```
[~]$ grep -h '[A-Z]' dirlist*.txt
```

Esto coincidirá con cada nombre de archivo que contenga una letra mayúscula. Lo siguiente coincidirá con cada nombre de archivo que contenga un guión o una A mayúscula o una Z mayúscula:

```
[~]$ grep -h '[-AZ]' dirlist*.txt
```

Clases de caracteres POSIX

Los rangos de caracteres tradicionales son una forma fácil de entender y eficaz de manejar el problema de especificar rápidamente conjuntos de caracteres. Desafortunadamente, no siempre funcionan. Si bien no hemos encontrado ningún problema con el uso de grep hasta ahora, es posible que tengamos problemas al usar otros programas.

En el Capítulo 4, analizamos cómo se utilizan los comodines para realizar la expansión del nombre de ruta. En esa discusión, dijimos que los rangos de caracteres podrían usarse de una manera casi idéntica a la forma en que se usan en las expresiones regulares, pero aquí está el problema:

```
[~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*
```

```
/usr/sbin/ModemManager
/usr/sbin/NetworkManager
```

(Dependiendo de la distribución de Linux, obtendremos una lista diferente de archivos, posiblemente una lista vacía. Este ejemplo es de Ubuntu). Este comando produce el resultado esperado: una lista de solo los archivos cuyos nombres comienzan con una letra mayúscula, pero con el siguiente comando obtenemos un resultado completamente diferente (solo se muestra una lista parcial de los resultados):

```
[~]$ ls /usr/sbin/[A-Z]*
```

```
/usr/sbin/biosdecode
/usr/sbin/chat
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/chroot
/usr/sbin/cleanup-info
/usr/sbin/complain
/usr/sbin/console-kit-daemon
```

¿Porqué es eso? Es una historia larga, pero esta es la versión corta:

Cuando se desarrolló Unix por primera vez, solo conocía los caracteres ASCII, y esta característica refleja ese hecho. En ASCII, los primeros 32 caracteres (números del 0 al 31) son códigos de control (cosas como tabulaciones, retrocesos y retornos de carro). Los siguientes 32 (32–63) contienen caracteres imprimibles, incluida la mayoría de los caracteres de puntuación y los números 0–9. Los siguientes 32 (números 64–95) contienen las letras mayúsculas y algunos símbolos más de puntuación. Los 31 finales (números 96-127) contienen letras minúsculas y aún más símbolos de puntuación. Según esta disposición, los sistemas que utilizan ASCII utilizaron un orden de clasificación que se ve así:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

Esto difiere del orden correcto del diccionario, que es así:

```
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
```

A medida que la popularidad de Unix se extendió más allá de los Estados Unidos, creció la necesidad de admitir caracteres que no se encuentran en el inglés estadounidense. La tabla ASCII se expandió para usar ocho bits completos, agregando caracteres 128-255, que acomodaron muchos más idiomas. Para admitir esta capacidad, los estándares POSIX introdujeron un concepto llamado locale, que podría ajustarse para seleccionar el juego de caracteres necesario para una ubicación particular. Podemos ver la configuración de idioma de nuestro sistema usando el siguiente comando.

```
[~]$ echo $LANG
```

Con esta configuración, las aplicaciones compatibles con POSIX utilizarán un orden de clasificación de diccionario en lugar de un orden ASCII. Esto explica el comportamiento de los comandos anteriores. Un rango de caracteres de [A-Z] cuando se interpreta en el orden del diccionario incluye todos los caracteres alfabéticos excepto la a minúscula, de ahí nuestros resultados.

Para solucionar parcialmente este problema, el estándar POSIX incluye una serie de clases de caracteres que proporcionan rangos de caracteres útiles, como se describe en la Tabla siguiente.

Clases de caracteres POSIX

Character class	**Description**
[:alnum:]	The alphanumeric characters. In ASCII, equivalent to: [A-Za-z0-9]
[:word:]	The same as [:alnum:] , with the addition of the underscore (_) character.
[:alpha:]	The alphabetic characters. In ASCII, equivalent to: [A-Za-z]
[:blank:]	Includes the space and tab characters.
[:cntrl:]	The ASCII control codes. Includes the ASCII characters 0 through 31 and 127.
[:digit:]	The numerals 0 through 9.
[:graph:]	The visible characters. In ASCII, it includes characters 33 through 126.
[:lower:]	The lowercase letters.
[:punct:]	The punctuation characters. In ASCII, equivalent to: `[-!"#$%&'()*+,./:;<=>?@[\\\]_`{

Character class	Description
[:print:]	The printable characters. All the characters in [:graph:] plus the space character.
[:space:]	The whitespace characters including space, tab, carriage return, newline, vertical tab, and form feed. In ASCII, equivalent to: `[ \t\r\n\v\f]`
[:upper:]	The uppercase characters.
[:xdigit:]	Characters used to express hexadecimal numbers. In ASCII, equivalent to: [0-9A-Fa-f]

Incluso con las clases de caracteres, todavía no existe una forma conveniente de expresar rangos parciales, como [A – M]. Usando clases de caracteres, podemos repetir nuestra lista de directorios y ver un resultado mejorado.

```
[~]$ ls /usr/sbin/[[:upper:]]*
```

```
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

Sin embargo, recuerde que este no es un ejemplo de expresión regular; más bien, es el shell que realiza la expansión del nombre de ruta. Lo mostramos aquí porque las clases de caracteres POSIX se pueden usar para ambos.

para ver la configuracion local, usamos el comando locale

```
[~]$ locale
```

```
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

Para cambiar la configuración regional para usar los comportamientos tradicionales de Unix, establezca la variable LANG en POSIX.

[~]$ export LANG=POSIX

puedes hacer este cambio permanente añadiendo la linea siguiente al archivo ~/.bashrc:

export LANG=POSIX

POSIX Expresiones regulares básicas frente a extendidas

Justo cuando pensamos que esto no podría ser más confuso, descubrimos que POSIX también divide las implementaciones de expresiones regulares en dos tipos: expresiones regulares básicas (BRE) y expresiones regulares extendidas (ERE). Las características que hemos cubierto hasta ahora son compatibles con cualquier aplicación que sea compatible con POSIX e implemente BRE. Nuestro programa grep es uno de esos programas.

¿Cuál es la diferencia entre BRE y ERE? Es una cuestión de metacaracteres. Con BRE, se reconocen los siguientes metacaracteres:

- [ ] *

Todos los demás caracteres se consideran literales. Con ERE, se agregan los siguientes metacaracteres (y sus funciones asociadas):

```
( ) { } ? + |
```

Sin embargo (y esta es la parte divertida), los caracteres (,), {, y} se tratan como metacaracteres en BRE si se escapan con una barra invertida, mientras que con ERE, antes de cualquier metacarácter con una barra invertida, se trata como un literal. Cualquier rareza que surja se tratará en las discusiones que siguen.

Debido a que las características que vamos a discutir a continuación son parte de ERE, necesitaremos usar un grep diferente. Tradicionalmente, esto lo ha realizado el programa egrep, pero la versión GNU de grep también admite expresiones regulares extendidas cuando se usa la opción -E.

Alternancia

La primera de las características extendidas de las expresiones regulares que discutiremos se llama alternancia, que es la función que permite que se produzca una coincidencia entre un conjunto de expresiones. Así como una expresión entre corchetes permite que un solo carácter coincida con un conjunto de caracteres especificados, la alternancia permite coincidencias con un conjunto de cadenas u otras expresiones regulares.

Para demostrarlo, usaremos grep junto con echo. Primero, intentemos una simple combinación de cuerdas.

```
[~]$ echo "AAA" | grep AAA
```

```
AAA
```

```
[~]$ echo "BBB" | grep AAA
```

```

```

Este es un ejemplo bastante sencillo, en el que canalizamos la salida de echo a grep y vemos los resultados. Cuando ocurre una coincidencia, la vemos impresa; cuando no se produce ninguna coincidencia, no vemos resultados.

Ahora agregaremos alternancia, representada por el metacarácter de barra vertical.

```
[~]$ echo "AAA" | grep -E 'AAA|BBB'
```

```
AAA
```

```
[~]$ echo "BBB" | grep -E 'AAA|BBB'
```

```
BBB
```

```
[~]$ echo "CCC" | grep -E 'AAA|BBB'
```

Aquí vemos la expresión regular 'AAA | BBB', que significa "coincidir con la cadena AAA o la cadena BBB". Tenga en cuenta que, dado que esta es una característica extendida, agregamos la opción -E a grep (aunque podríamos haber usado el programa egrep en su lugar), y encerramos la expresión regular entre comillas para evitar que el shell interprete el metacarácter de barra vertical como un operador de tubería. La alternancia no se limita a dos opciones.

```
[~]$ echo "AAA" | grep -E 'AAA|BBB|CCC'
```

```
AAA
```

Para combinar la alternancia con otros elementos de expresión regular, podemos usar () para separar la alternancia.

```
[~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

Esta expresión coincidirá con los nombres de archivo de nuestras listas que comienzan con bz, gz o zip. Si hubiéramos dejado los paréntesis, el significado de esta expresión regular cambia para coincidir con cualquier nombre de archivo que comience con bz o contenga gz o contenga zip:

```
[~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

Cuantificadores

```
Las expresiones regulares extendidas admiten varias formas de especificar
el número de veces que se hace coincidir un elemento, como se describe en
las secciones siguientes.
```

? — Combina un elemento cero o una vez

```
Este cuantificador significa, en efecto, "hacer que el elemento anterior
sea opcional".
Supongamos que queremos comprobar la validez de un número de teléfono y
consideramos que un número de teléfono es válido si coincide con cualquiera
de estas dos formas, donde n es un número:
```

> - (nnn) nnn-nnnn
> - nnn nnn-nnnn

podriamos construir una expresion regular tal que así:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

En esta expresión, seguimos los caracteres entre paréntesis con signos de interrogación para indicar que deben coincidir cero o una vez. Nuevamente, debido a que los paréntesis son normalmente metacaracteres (en ERE), los precedemos con barras invertidas para hacer que se traten como literales.

Vamos a intentarlo.

```
[~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9] \)? [0-9][0-9][0-9]-
[0-9][0-9][0-9][0-9]$' (555) 123-4567 [~]$ echo "555 123-4567" | grep -E '^\(?
[0-9][0-9][0-9]\) ? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$' 555 123-4567 [~]$
echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\) ? [0-9][0-9][0-9]-[0-9][0-
9][0-9][0-9]$' [~]$
```

Aquí vemos que la expresión coincide con ambas formas del número de teléfono, pero no coincide con una que contenga caracteres no numéricos. Esta expresión no es perfecta ya que aún permite paréntesis no coincidentes alrededor del código de área, pero realizará la primera etapa de una verificación.

* — Combina un elemento cero o más veces

Como el metacarácter ?, el * se usa para denotar un elemento opcional; sin embargo, a diferencia del ? , el elemento puede aparecer tantas veces como desee, no solo una vez. Digamos que queríamos ver si una cadena era una oración; es decir, comienza con una letra mayúscula, luego contiene cualquier número de letras mayúsculas y minúsculas y espacios, y termina con un punto. Para hacer coincidir esta definición (cruda) de una oración, podríamos usar una expresión regular como esta:

```
[[:upper:]][[:upper:][:lower:] ]*\.
```

La expresión consta de tres elementos: una expresión entre corchetes que contiene la clase de caracteres [: superior:], una expresión entre corchetes que contiene las clases de caracteres [: superior:] y [: inferior:] y un espacio, y un punto de escape con una barra invertida . El segundo elemento va seguido con un metacarácter * de modo que después de la letra mayúscula inicial en nuestra oración, cualquier número de letras mayúsculas y minúsculas y espacios pueden seguirlo y aun así coincidir.

[~]$ echo "This works." | grep -E '[[:upper:]][[:upper:][:lower:]]*.'

This works.

[~]$ echo "This Works." | grep -E '[[:upper:]][[:upper:][:lower:]]*.'

This Works.

[~]$ echo "this does not" | grep -E '[[:upper:]][[:upper:][:lower:]]*.'

[~]$

La expresión coincide con las dos primeras pruebas, pero no con la tercera, ya que carece del carácter inicial en mayúsculas y el punto final requeridos.

+ —Combina un elemento una o más veces

El metacarácter + funciona de manera muy similar al *, excepto que requiere al menos una instancia del elemento anterior para generar una coincidencia. Aquí hay una expresión regular que coincidirá solo con las líneas que constan de grupos de uno o más caracteres alfabéticos separados por espacios simples:

```
^([[:alpha:]]+ ?)+$
```

Intentemoslo.

[~]$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'

This that

[~]$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'

a b c

[~]$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'

[~]$ echo "abc d" | grep -E '^([[:alpha:]]+ ?)+$'

[~]$

Vemos que esta expresión no coincide con la línea a b 9 porque contiene un carácter no alfabético; tampoco coincide con abc d porque más de un carácter de espacio separa los caracteres cy d.

{} -- Hacer coincidir un elemento una cantidad específica de veces

Los metacaracteres {y} se utilizan para expresar el número mínimo y máximo de coincidencias requeridas. Pueden especificarse de cuatro formas posibles, como se describe en la Tabla siguiente.

Tabla: Especificación del número de coincidencias

Especificador	Sentido
{n}	Coincide con el elemento anterior si ocurre exactamente n veces.
{n, m}	Coincide con el elemento anterior si ocurre al menos n veces pero no más de m veces.
{n,}	Haga coincidir el elemento anterior si aparece n o más veces.

Especificador	Sentido
{, m}	Coincide con el elemento anterior si no aparece más de m veces.

Volviendo a nuestro ejemplo anterior con los números de teléfono, podemos usar este método de especificar repeticiones para simplificar nuestra expresión regular original de lo siguiente:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

a la siguiente expresion:

```
^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$
```

brobemoslo:

[~]$ echo "(555) 123-4567" | grep -E '^(?[0-9]{3})? [0-9]{3}-[0-9]{4}$'

(555) 123-4567

[~]$ echo "555 123-4567" | grep -E '^(?[0-9]{3})? [0-9]{3}-[0-9]{4}$'

555 123-4567

[~]$ echo "5555 123-4567" | grep -E '^(?[0-9]{3})? [0-9]{3}-[0-9]{4}$'

[~]$

Como podemos ver, nuestra expresión revisada puede validar con éxito ambas expresiones numéricas con y sin paréntesis, mientras rechaza aquellas expresiones que no están formateados correctamente.

Poner en práctica las expresiones regulares

Veamos algunos de los comandos que ya conocemos y veamos cómo se pueden usar con expresiones regulares.

Validación de una lista de teléfonos con grep

En nuestro ejemplo anterior, observamos números de teléfono individuales y verificamos que tengan el formato adecuado. Un escenario más realista sería verificar una lista de números, así que hagamos una lista. Haremos esto recitando un encantamiento mágico en la línea de comandos. Será mágico porque no hemos cubierto la mayoría de los comandos involucrados, pero no se preocupe. Llegaremos ahí más adelante. Aquí está el encantamiento.

[~]$ for i in {1..10}; do echo "(${RANDOM:0:3}) ${RANDOM:0:3}-$ {RANDOM:0:4}" >> phonelist.txt; done

Este comando producirá un archivo llamado phonelist.txt que contiene 10 números de teléfono. Cada vez que se repite el comando, se agregan otros 10 números a la lista. También podemos cambiar el valor 10 cerca del comienzo del comando para producir más o menos números de teléfono. Sin embargo, si examinamos el contenido del archivo, vemos que tenemos un problema.

[~]$ cat phonelist.txt

(232) 298-2265 (624) 381-1078 (540) 126-1980 (874) 163-2885 (286) 254-2860 (292) 108-518 (129) 44-1379 (458) 273-1642 (686) 299-8268 (198) 307-2440

Algunos de los números están mal formados, lo cual es perfecto para nuestros propósitos porque usaremos grep para validarlos.

Un método útil de validación sería escanear el archivo en busca de números no válidos y mostrar la lista resultante.

[~]$ grep -Ev '^([0-9]{3}) [0-9]{3}-[0-9]{4}$' phonelist.txt

(292) 108-518

(129) 44-1379

[me@linuxbox ~]$

Aquí usamos la opción -v para producir una coincidencia inversa de modo que solo generemos las líneas en la lista que no coinciden con la expresión especificada. La expresión en sí incluye los metacaracteres de anclaje en cada extremo para garantizar que el número no tenga caracteres adicionales en ninguno de los extremos. Esta expresión también requiere que los paréntesis estén presentes en un número válido, a diferencia de nuestro ejemplo de número de teléfono anterior.

Encontrar nombres de archivo feos con find

El comando de búsqueda find admite una prueba basada en una expresión regular. Hay una consideración importante a tener en cuenta al usar expresiones regulares en find versus grep. Mientras que grep imprimirá una línea cuando la línea contenga una cadena que coincida con una expresión, find requiere que el nombre de la ruta coincida exactamente con la expresión regular. En el siguiente ejemplo, usaremos find con una expresión regular para encontrar cada nombre de ruta que contenga cualquier carácter que no sea miembro del siguiente conjunto:

```
[-_./0-9a-zA-Z]
```

Tal análisis revelaría nombres de ruta que contienen espacios incrustados y otros caracteres potencialmente ofensivos.

```
[me@linuxbox ~]$ find . -regex '.*[^-_./0-9a-zA-Z].*'
```

Debido al requisito de una coincidencia exacta de la ruta completa, usamos. * En ambos extremos de la expresión para que coincida con cero o más instancias de cualquier carácter. En el medio de la expresión, usamos una expresión de corchete negado que contiene nuestro conjunto de caracteres de nombre de ruta aceptables.

Buscando archivos con localizar

El programa de localización admite expresiones regulares básicas (la opción --regexp) y extendidas (la opción --regex). Con él, podemos realizar muchas de las mismas operaciones que realizamos anteriormente con nuestros archivos dirlist.

```
[~]$ locate --regex 'bin/(bz|gz|zip)'
```

Utilizando la alternancia, realizamos una búsqueda de nombres de ruta que contienen bin/bz, bin/gz o /bin/zip.

Buscando texto con less y vim

less y vim comparten el mismo método de búsqueda de texto. Si presiona la tecla / seguida de una expresión regular, se realizará una búsqueda. Si usamos less para ver nuestro archivo phonelist.txt, así:

```
[~]$ less phonelist.txt
```

```
/^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

less resaltará las cadenas que coincidan, dejando las inválidas fáciles de detectar.

vim, por otro lado, admite expresiones regulares básicas, por lo que nuestra expresión de búsqueda se vería así

```
/([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}
```

Podemos ver que la expresión es prácticamente la misma; sin embargo, muchos de los caracteres que se consideran metacaracteres en las expresiones extendidas se consideran literales en las expresiones básicas. Se tratan solo como metacaracteres cuando se escapan con una barra invertida. Dependiendo de la configuración particular de vim en nuestro sistema, se resaltará la coincidencia. Si no es así, pruebe este comando de modo de comando para activar el resaltado de búsqueda:

```
:hlsearch
```

Nota:

> Dependiendo de su distribución, **vim** puede o no admitir el resaltado de búsqueda de texto. Ubuntu, en particular, proporciona una versión reducida de vim de forma predeterminada. En tales sistemas, es posible que desee utilizar su administrador de paquetes para instalar una versión más completa de **vim**.