Expresiones regulares

Autores del manual

Este manual ha sido realizado por los siguientes colaboradores de DesarrolloWeb.com:

irv.
http://www.ignside.net/
(6 capítulos)

Expresiones regulares

Las expresiones regulares son una serie de carácteres que forman un patrón, normalmente representativo de otro grupo de carácteres mayor, de tal forma que podemos comparar el patrón con otro conjunto de carácteres para ver las coincidencias.

Las expresiones regulares estan disponibles en casi cualquier lenguaje de programación, pero aunque su sintaxis es relativamente uniforme, cada lenguaje usa su propio dialecto.

Si es la primera vez que te acercas al concepto de expresiones regulares (regex para abreviar) te animará saber que seguro que ya las has usado, aún sin saberlo, al menos en su vertiente mas básica. Por ejemplo, cuando en una ventana DOS ejecutamos dir *.* para obtener un listado de todos los archivos de un directorio, estamos utilizando el concepto de expresiones regulares, donde el patrón * coincide con cualquier cadena de caracteres.

Unos ejemplos simplificados:

```
<? am // este es nuestro patrón. Si lo comparamos con:
am // coincide
panorama // coincide
ambicion // coincide
campamento // coincide
mano // no coincide
?>
```

Se trata sencillamente de ir cotejando un patrón (pattern) -que en este ejemplo es la secuencia de letras 'am'- con una cadena (subject) y ver si dentro de ella existe la misma secuencia. Si existe, decimos que hemos encontrado una coincidencia (match, en inglés).

Otro ejemplo:

patrón: el
el ala aleve del leve abanico
Hasta ahora los ejemplos han sido sencillos, ya que los patrones usados eran literales, es decir que solo encontramos coincidencias cuando hay una ocurrencia exacta.

Si sabemos de antemano la cadena exacta a buscar, no es necesario quebrarse con un patrón complicado, podemos usar como patrón la exacta cadena que buscamos, y esa y no otra será la que de coincidencia. Asi, si en una lista de nombres buscamos los datos del usuario pepe podemos usar pepe como patrón. Pero si ademas de pepe nos interesa encontrar ocurrencias de pepa y pepito los literales no son suficientes.

El poder de las expresiones regulares radica precisamente en la flexibilidad de los patrones, que pueden ser confrontados con cualquier palabra o cadena de texto que tenga una estructura conocida.

De hecho normalmente no es necesario usar funciones de expresiones regulares si vamos a usar patrones literales. Existen otras funciones (las funciones de cadena) que trabajan mas eficaz y rapidamente con literales.

Caracteres y meta caracteres

Nuestro patrón puede estar formado por un conjunto de carácteres (un grupo de letras,

numeros o signos) o por meta caracteres que representan otros carácteres, o permiten una búsqueda contextual.

Los meta-caracteres reciben este nombre porque no se representan a ellos mismos, sino que son interpretados de una manera especial.

He aqui la lista de meta caracteres mas usados:

. * ? + [ ] ( ) { } ^ $ | \

Iremos viendo su utilización, agrupandolos segun su finalidad.

Meta caracteres de posicionamiento, o anclas

Los signos ^ y $ sirven para indicar donde debe estar situado nuestro patrón dentro de la cadena para considerar que existe una coincidencia.

Cuando usamos el signo ^ queremos decir que el patrón debe aparecer al principio de la cadena de carácteres comparada. Cuando usamos el signo $ estamos indicando que el patrón debe aparecer al final del conjunto de carácteres. O mas exactamente, antes de un caracter de nueva linea Asi:

```
<?

^am // nuestro patrón
am // coincide
cama // no coincide
ambidiestro // coincide
Pam // no coincide
caramba // no coincide

am$
am // coincide
salam // coincide
ambar // no coincide
Pam // coincide

^am$
am // coincide
salam // no coincide
ambar // no coincide

?>
```

o como en el ejemplo anterior:

patrón: ^el
el ala aleve del leve abanico
Las expresiones regulares que usan anclas solo devolveran una ocurrencia, ya que por ejemplo, solo puede existir una secuencia el al comienzo de la cadena.

patrón: el$
el ala aleve del leve abanico
Y aqui no encontramos ninguna, ya que en la cadena a comparar (la linea en este caso) el patrón "el" no está situado al final.
Para mostrar una coincidencia en este ejemplo, tendriamos que buscar "co":

patrón: co$
con el ala aleve del leve abanico
Hemos comenzado por unos metacaracteres especiales, ya que ^ $ no representan otros carácteres, sino posiciones en una cadena. Por eso, se conocen tambien como anchors o anclas.

Escapando caracteres

Puede suceder que necesitemos incluir en nuestro patrón algun metacaracter como signo literal, es decir, por si mismo y no por lo que representa. Para indicar esta finalidad usaremos un carácter de escape, la barra invertida \.
Así, un patrón definido como 12\$ no coincide con una cadena terminada en 12, y sí con 12$:

patrón: 100$
el ala aleve del leve abanico cuesta 100$
patrón: 100\$
el ala aleve del leve abanico cuesta 100$
Fijate en los ejemplos anteriores. En el primero, no hay coincidencia, porque se interpreta "busca una secuencia consistente en el número 100 al final de la cadena", y la cadena no termina en 100, sino en 100$.
Para especificar que buscamos la cadena 100$, debemos escapar el signo $

Como regla general, la barra invertida \ convierte en normales caracteres especiales, y hace especiales caracteres normales.

El punto . como metacaracter

Si un metacaracter es un caracter que puede representar a otros, entonces el punto es el metacaracter por excelencia. Un punto en el patrón representa cualquier caracter excepto nueva línea.

Y como acabamos de ver, si lo que queremos buscar en la cadena es precisamente un punto, deberemos escaparlo: \.
patrón: '.l'
el ala aleve del leve abanico
Observa en el ejemplo anterior como el patrón es cualquier caracter (incluido el de espacio en blanco) seguido de una l.

Metacaracteres cuantificadores

Los metacaracteres que hemos visto ahora nos informan si nuestro patron coincide con la cadena a comparar. Pero ¿y si queremos comparar con nuestra cadena un patrón que puede estar una o mas veces, o puede no estar ? Para esto usamos un tipo especial de meta carácteres: los multiplicadores.

Estos metacaracteres que se aplican al caracter o grupo de caracteres que les preceden indican en que número deben encontrarse presentes en la cadena para que haya una ocurrencia.

Por ello se llaman cuantificadores o multiplicadores. Los mas usados son * ? +

```
<?
* // coincide si el caracter (o grupo de caracteres) que le
// precede esta presente 0 o mas veces
```

```
// ab* coincide con "a", "ab", "abbb", etc.
//ejemplo:
cant*a // coincide con canta, cana, canttta

? // coincide si el carácter (o grupo de caracteres) que precede
// esta presente 0 o 1 vez
// ab? coincide con "a", "ab", no coincide con "abb"
// ejemplo:
cant?a // coincide con canta y cana
d?el // coincide con del y el
(ala)?cena // coincide con cena y alacena

+ // coincide si el carácter (o grupo) que le precede está
// presente al menos 1 o mas veces.
// ab+ coincide con "ab", "abbb", etc. No coincide con "a"
//ejemplo:
cant+a // coincide con canta, cantttta, NO coincide con
// cana ?>
```

patrón: 'a*le'
el ala aleve del leve abanico

patrón: ' ?le'
el ala aleve del leve abanico

patrón: ' +le'
el ala aleve del leve abanico
Ademas de estos cuantificadores sencillos tambien podemos especificar el numero de veces máximo y mínimo que debe darse para que haya una ocurrencia:

patrón: (.a){2}
el ala aleve del leve abanico

```
<?
{x,y} // coincide si la letra (o grupo) que le precede esta presente
// un minimo "x" veces y como máximo "y" veces
// "ab{2}" coincide con "abb": exactamente dos ocurrencias de "b"
// "ab{2,}" coincide con "abb", "abbbb" ... Como mínimo dos
// ocurrencias de b, máximo indefinido
// "ab{3,5}" coincide con "abbb", "abbbb", o "abbbbb": Como minimo
// dos ocurrencias, como máximo 5

a{2,3} // coincide con casaa, casaaa

a{2, } // coincide con cualquier palabra que tenga al
// menos dos "a" o mas: casaa o casaaaaaa, no con casa

a{0,3} // coincide con cualquier palabra que tenga 3 o
// menos letras "a".
// NOTA: puedes dejar sin especificar el valor máximo. NO
// puedes dejar el valor inicial vacío

a{5} // exactamente 5 letras "a"
?>
```

Por tanto, los cuantificadores * + ? pueden también ser expresados así:

* equivale a {0,} (0 o mas veces)
+ equivale a {1,} (1 o mas veces)
? equivale a {0,1} (0 o 1 vez)

Este artículo continúa en el manual de expresiones regulares.

*Artículo por **irv.***

Metacaracteres de rango, alternancia y agrupadores

Metacaracteres de rango

Los corchetes [] incluidos en un patrón permiten especificar el rango de caracteres válidos a comparar. Basta que exista cualquiera de ellos para que se de la condición:

```
<?

[abc] // El patrón coincide con la cadena si en esta hay
// cualquiera de estos tres carácteres: a, b, c

[a-c] // coincide si existe una letra en el rango ("a", "b" o "c")
c[ao]sa // coincide con casa y con cosa

[^abc] // El patrón coincide con la cadena si en esta NO hay
// ninguno de estos tres carácteres: a, b, c
// Nota que el signo ^ aqui tiene un valor excluyente

c[^ao]sa // Coincide con cesa, cusa, cisa (etc); no coincide
// con casa ni cosa

[0-9] // Coincide con una cadena que contenga cualquier
// número entre el 0 y el 9

[^0-9] // Coincide con una cadena que NO contenga ningun
// número

[A-Z] // Coincide con cualquier carácter alfabetico,
// en mayúsculas. No incluye numeros.

[a-z] // Como el anterior, en minúsculas

[a-Z] // Cualquier carácter alfabético, mayusculas o minusculas

?>
```

Una cuestión a recordar es que las reglas de sintaxis de las expresiones regulares no se aplican igual dentro de los corchetes. Por ejemplo, el metacarácter ^ no sirve aqui de ancla, sino de caracter negador. Tampoco es necesario escapar todos los metacaracteres con la barra invertida. Solo será necesario escapar los siguientes metacaracteres:] \ ^ -

El resto de metacaracteres pueden incluirse ya que son considerados -dentro de los corchetes- caracteres normales.

patrón: [aeiou]
el ala aleve del leve abanico

patrón: [^aeiou]
el ala aleve del leve abanico

patrón: [a-d]
el ala aleve del leve abanico

Como estos patrones se usan una y otra vez, hay atajos:

```
<?
// atajo equivale a significado

\d [0-9] // numeros de 0 a 9
\D [^0-9] // el contrario de \d

\w [0-9A-Za-z] // cualquier numero o letra
\W [^0-9A-Za-z] // contrario de \w, un carácter que no

// sea letra ni numero

\s [ \t\n\r] // espacio en blanco: incluye espacio,
// tabulador, nueva linea o retorno

\S [^ \t\n\r] // contrario de \s, cualquier carácter
// que no sea espacio en blanco

// solo regex POSIX
[[:alpha:]] // cualquier carácter alfabético aA - zZ.
[[:digit:]] // Cualquier número (entero) 0 - 9
[[:alnum:]] // Cualquier carácter alfanumérico aA zZ 0 9
[[:space:]] // espacio
?>
```

Metacaracteres de alternancia y agrupadores

```
<?
(xyz) // coincide con la secuencia exacta xyz
x|y // coincide si esta presente x ó y

(Don|Doña) // coincide si precede "Don" o "Doña"
?>
```

patrón: (al)
el ala aleve del leve abanico

patrón: a(l|b)
el ala aleve del leve abanico
Los paréntesis sirven no solamente para agrupar secuencias de caracteres, sino también para capturar subpatrones que luego pueden ser devueltos al script (backreferences). Hablaremos mas de ello al tratar de las funciones POSIX y PCRE, en las paginas siguientes.

Un ejemplo típico sería una expresion regular cuyo patrón capturase direcciones url validas y con ellas generase links al vuelo:

```
<? $text = "una de las mejores páginas es http://www.blasten.com";
$text = ereg_replace("http:\/\/(.*\.(com|net|org))",
"\1", $text);
print $text;
?>
```

El anterior ejemplo produciría un enlace usable, donde la url se tomaría de la retro-referencia \0 y la parte visible de la retro-referencia \1 una de las mejores páginas es www.ignside.net

Fijate que en el ejemplo anterior usamos dos grupos de parentesis (anidados), por lo que se producirian dos capturas: La retro-referencia \0 coincide con la coincidencia buscada. Para capturarla no es preciso usar parentesis.

La retro-referencia \1 coincide en este caso con "www.blasten.com" y es capturada por el parentesis (.*\.(com|net|org))

La retro-referencia \2 coincide con "net" y se corresponde con el parentesis anidado (com|net|org)

Ten en cuenta que esta caracteristica de capturar ocurrencias y tenerlas disponibles para retroreferencias consume recursos del sistema. Si quieres usar parentesis en tus expresiones regulares, pero sabes de antemano que no vas a reusar las ocurrencias, y puedes prescindir de la captura, coloca despues del primer parentesis ?:

```
<?
text = ereg_replace("http:\/\/(.*\.(?:com|net|org))",
     "<a href=\"\0\">\1</a>", $text);
?>
```

Al escribir (?:com|net|org) el subpatron entre parentesis sigue agrupado, pero la coincidencia ya no es capturada.

Como nota final sobre el tema, PHP puede capturar hasta 99 subpatrones a efectos de retro-referencia, o hasta 200 (en total) si buscamos subpatrones sin capturarlos.

*Artículo por **irv.***

Ejemplo práctico expresiones regulares

Un ejemplo práctico

Hemos dicho que las expresiones regulares son uno de los instrumentos mas útiles en cualquier lenguaje de programación. ¿Para que podemos usarlas?. Uno de sus usos mas típicos es el de validar entradas de datos que los visitantes de una página puedan mandarnos a través de formularios html.
El ejemplo mas corriente es el de una dirección email. Imaginemos que queremos filtrar las direcciones introducidas por los visitantes, para evitar introducir en la base de datos la típica dirección basura ghghghghghghg. Todos sabemos la estructura de una dirección email, formada por la cadena nombreusuario, el signo @ y la cadena nombredominio. Tambien sabemos que nombredominio esta formado por dos subcadenas, 'nombredomino', un '.' y un sufijo 'com', 'net', 'es' o similar.
Por tanto la solución a nuestro problema es idear una expresión regular que identifique una dirección email valida típica, y confrontarla con la cadena (dirección email) pasada por el visitante
Por ejemplo:

```
<?
^[^@ ]+@[^@ ]+.[^@ .]+$
?>
```

Vamos a diseccionar nuestra expresión regular:

```
<?
^ // queremos decir que el primer carácter que buscamos
// debe estar al principio de la cadena a comparar.

[^@ ] // ese primer signo no debe ser ni el signo @
// ni un espacio

+ // y se repite una o mas veces
@ // luego buscamos el signo @

[^@ ]+ // Seguido de otro signo que no es un @ ni un
// espacio y se repite una o mas veces

. // Seguido de un .

[^@ .] // Seguido de un carácter que no sea ni @,
// ni espacio ni punto

+$ // Que se repite una o mas veces y el último esta
// al final de la cadena
?>
```

Y para comprobarlo en la práctica, usamos una de las funciones de php relacionadas con las expresiones regulares:ereg().
Acudiendo al manual php, podemos averiguar que esta función tiene la siguiente sintaxis:
ereg (string pattern, string string)
Busca en string las coincidencias con la expresión regular pattern. La búsqueda diferencia entre mayúsculas y minúsculas.
Devuelve un valor verdadero si se encontró alguna coincidencia, o falso in no se encontraron coincidencias u ocurrió algún error. Podríamos usar esta funcion para un validador email con algo asi como:

```
<?

// establecemos una secuencia condicional: si la variable $op no existe y
// es igual a "ds", se muestra un formulario

if ($op != "ds") { ?>
<form>
<input type=hidden name="op" value="ds">
<strong>Tu email:</strong><br />
<input type=text name="email" value="" size="25" />
<input type=submit name="submit" value="Enviar" />
</form>
<?
}

// Si $op existe y es igual a "ds"m, se ejecuta la función ereg buscando
// nuestra cadena dentro del patrón $email que es la direccion enviada por
// el usuario desde el formulario anterior

else if ($op == "ds")
{
   if (ereg("^[^@ ]+@[^@ ]+.[^@ .]+$", $email ) )
   {
     print "<BR>Esta dirección es correcta: $email"; }
   else {echo "$email no es una dirección valida";}
  }

?>
```

No hace falta advertir que se trata de un ejemplo muy elemental, que dará por válida cualquier dirección email que tenga una mínima apariencia de normalidad (por ejemplo, daria por valida 'midireccionnn@noteimporta.commm')

*Artículo por **irv.***

Referencia de expresiones regulares en PHP

Esta referencia de expresiones regulares es muy interesante para tener a mano. Está recogida de un comentario del manual de php.net.

```
<?

^ // Comienzo de la zadena
$ // Final de la cadena

n* // Cero o mas "n" (donde n es el carácter precedente)
n+ // Uno o mas "n"
n? // Un posible "n"

n{2} // Exactamente dos "n"
n{2,} // Al menos dos o mas "n"
n{2,4} // De dos a cuatro "n"

() // Parentesis para agrupar expresiones
(n|a) // o "n" o "a"

. // Cualquier carácter

[1-6] // un número entre 1 y 6
[c-h] // una letra en minuscula entre c y h
[D-M] // una letra en mayúscula entre D y M
[^a-z] // no hay letras en minuscula de a hasta z
[_a-zA-Z] // un guion bajo o cualquier letra del alfabeto

^.{2}[a-z]{1,2}_?[0-9]*([1-6]|[a-f])[^1-9]{2}a+$

/* Una cadena que comienza por dos carácteres cualquiera
  Seguidos por una o dos letras (en minuscula)
  Seguidos por un guion _ bajo opcional
   Seguidos por cero o mas números
  Seguidos por un numero del 1 al 6 o una letra de la -a- a la -f-
  Seguidos por dos carácteres que no son números del 1 al 9
  Seguidos de uno o mas carácteres al final de la cadena
  Tomado de una anotacion al manual de php.net, de mholdgate -
  wakefield dot co dot uk */

?>
```

*Artículo por **irv.***

Funciones PHP para expresiones regulares

PHP tiene dos conjuntos distintos de funciones relacionadas con expresiones regulares, llamadas POSIX y PCRE.

Las funciones "PCRE" son "PERL Compatible", es decir, similares a las funciones nativas Perl, aunque con ligeras diferencias. Son bastante mas poderosas que las funciones POSIX, y correlativamente mas complejas.

POSIX
Incluye seis funciones diferentes. Como nota común, pasas primero el patrón (la expresión a buscar) y como segundo argumento la cadena a comparar

ereg confronta la cadena con el patrón de búsqueda y devuelve TRUE o FALSE segun la encuentre.
eregi como la anterior, SIN distinguir entre mayusculas-minusculas

```
<?
ereg ("^am", "america"); // TRUE

$es_com = ereg("(\.)(com$)",$url);
// buscamos dos subpatrones en $url. El primero es
// un punto (literal) y por eso va escapado con la barra.
// el segundo subpatron también literal busca la secuencia "com"
// al final de una palabra.

?>
```

ereg_replace: Busca cualquier ocurrencia del patrón en la cadena y la reemplaza por otra.
eregi_replace: como la anterior pero sin distinguir mayusculas minusculas:

patrón, reemplazo, cadena a confrontar

```
<?
$cadena = ereg_replace ("^am", "hispano-am", "america"); // $cadena = hispano-america
?>
```

split() divide una cadena en piezas (que pasan a un array) usando expresiones regulares:
spliti: como el anterior, sin diferenciar Mayusculas-minusculas

Es básicamente igual que explode, pero utilizando expresiones regulares para dividir la cadena, en lugar de expresiones literales

```
<?
$date = "24-09-2003";
list($month, $day, $year) = split ('[/.-]', $date);

?>
```

Almacenando los resultados con ereg

Podemos pasar un patrón con subpatrones agrupados en parentesis. Si en este caso usamos ereg, podemos añadir un tercer parámetro: el nombre de un array que almacenará las ocurrencias; $array[1] contendrá la subcadena que empieza en el primer paréntesis izquierdo; $array[2] la que comienza en el segundo, etc. $array[0] contendrá una copia de la cadena.

```
<?
$date = "24-09-2003"; // pasamos una fecha formato dd-mm-yyyy

if (ereg ("([0-9]{1,2})-([0-9]{1,2})-([0-9]{4})", $date, $mi_array)) {
   echo "$mi_array[3].$mi_array[2].$mi_array[1]"; // coincide. Lo mostramos en orden inverso porque somos asi : )
} else {
    echo "Invalid date format: $date"; // no coincide
}
?>
```

Almacenando los resultados con ereg_replace: backreferences

De forma muy similar a ereg, las funciones de búsqueda y sustitución ereg_replace y eregi_replace pueden almacenar y reutilizar subocurrencias (subpatrones encontrados en la cadena).

La principal diferencia es la forma de llamar las subocurrencias almacenadas, ya que necesitamos utilizar la barra invertida: \\0, \\1, \\2, y asi hasta un máximo de 9.

La primera referencia \\0 hace referencia a la coincidencia del patrón entero; el resto, a las sub-ocurrencias de los sub-patrones, de izquierda a derecha.

Por ejemplo vamos a usar esta capacidad para convertir una cadena que muestra una url en un enlace:

```
<?
$url = "la página blasten.com (http://www.blasten.com)";
$url = ereg_replace ("(http|ftp)://(www\.)?(.+)\.(com|net|org)","<a href=\"\\0 \">\\3</a>",$url);
echo $url; ?>
```

*Artículo por **irv.***

Funciones PCRE

Las funciones PCRE son "perl-compatibles". Perl es uno de los lenguajes de programación con mejor motor de expresiones regulares, y además es muy conocido. Esta librería es utilizada (con distintas variantes) no solo por Perl, sino por el propio PHP y tambien en otros entornos, como el server Apache, Phyton o KDE.

Las funciones de expresiones regulares PCRE de PHP son mas flexibles, potentes y rápidas que las POSIX.

Prácticamente no existe ninguna diferencia sintáctica entre un patrón PCRE o POSIX; muchas veces son intercambiables. Naturalmente existe alguna diferencia. La mas evidente, que nuestro patrón deberá estar marcado en su principio y final unos delimitadores, normalmente dos barras:

/patron/

Delimitadores

Se puede usar como delimitador cualquier carácter especial (no alfanumerico) salvo la barra invertida \.

La costumbre mas extendida es, como hemos visto, usar la barra /, sin embargo, si posteriormente vamos a necesitar incluir el mismo caracter delimitador en el patrón, tendremos que escaparlo, por lo que tiene sentido usar en esos casos unos delimitadores distintos: (), {}, [], o < >

Modificadores

Se colocan despues del patrón:

```
<?
m // multilínea. Si nuestra cadena contiene varias lineas físicas (\n)
  // respeta esos saltos de línea, lo que significa, por ejemplo,
  // que las anclas ^ $ no se aplican al principio y final de la
  // cadena, sino al principio y final de cada linea

s // El metacaracter . representa cualquier carácter menos el de
  // nueva linea. Con el modificador "s" tambien representa la nueva linea.

i // Se confronta el patrón con la cadena ignorando Mayusculas minusculas

x // ignora espacios (salvo que esten escapados o incluidos
  // específicamente dentro del rango de búsqueda. Ignora cualquier caracter
  // despues de almohadilla (#) hasta nueva línea. Sirve para incluir
  // comentarios y hacer mas legible el patrón.

e // solo en preg_replace. Evalua las ocurrencias como código php antes
  // de realizar la sustitución.

A // El patrón es forzado a ser "anclado", esto es, solo existirá una
  // ocurrencia si es al inicio de la cadena.

E // el carácter $ en el patrón casará con el fin de la cadena.
  // Sin este modificador, $ casa tambien con el carácter inmediatamente
// antes del de una nueva línea.

U // Este modificador invierte la "codicia" de los cuantificadores.
  // Si aplicamos U, * se convierte en perezoso (lazy) y *? vuelve a su
  // comportamiento normal.
  // Un cuantificador es "codicioso" (greedy) cuando intenta capturar todas
  // las ocurrencias posibles, y perezoso (lazy) cuando captura la
  // ocurrencia mas corta. ?>
```

Delimitadores de palabra

En las funciones PCRE puedes usar \b que indica el comienzo y fin de una palabra (o mejor, de una secuencia alfanumerica):
/\bpatron\b/

\B, por el contrario, se refiere a un patrón que no está al comienzo o fin de una palabra.

Codicioso o no

Las expresiones regulares que usan cuantificadores tienden a ser todo lo codiciosas que les sea permitido, siempre que respeten el patrón a seguir. Con el modificador U se invierte este comportamiento.

En modo codicioso el patrón casará todas las ocurrencias que pueda, mientras que en modo lazy o ungreedy, casará solo la mas corta posible (e).
Advierte que las dos soluciones son correctas.

patrón: /http:\/\/.*\.(com|net|org)/esto es un link: http://www.abc.com y este otro mas: http://www.blah.com

patrón: /http:\/\/.*\.(com|net|org)/Uesto es un link: http://www.abc.com y este otro mas: http://www.blah.com

Las funciones

Tenemos cinco funciones PCRE:

preg_match()
preg_match_all()
preg_replace()
preg_split()
preg_grep()

preg_match busca el patrón dentro de la cadena, devolviendo TRUE si es hallado:
patron, cadena [,array de resultados]

Si indicamos el tercer parámetro, tendremos los resultados en un array; $array[0] contendrá la ocurrencia del patrón; $array[1] tendrá la cadena que case con el primer subpatrón y así sucesivamente.

preg_match_all encuentra todas las ocurrencias del patrón en la cadena.
patron, cadena, array de patrones, orden

Si proporcionamos el cuarto parámetro, colocara las ocurrencias en un array siguiendo el orden indicado.

preg_replace busca y reemplaza el patrón en la cadena. Tanto el patrón como la sustitución pueden pasarse en array, y pueden contener expresiones regulares.
Puede tambien especificarse un límite máximo de ocurrencias.

preg_split opera como split aunque tambien puedes pasarle expresiones regulares

preg_grep busca el patrón dentro de un array, y devuelve otro array con las ocurrencias.
patron, array

El autor de este artículo ha creado una herramienta para probar expresiones regulares, que seguro nos será de utilidad para comprobar nuestros conocimientos.

*Artículo por **irv.***