Expresiones regulares en PHP

1. Introducción

Las expresiones regulares permiten definir patrones de coincidencia y aplicarlas a cadenas de texto para saber si la cadena (o parte de ella) cumple el patrón e incluso realizar transformaciones de la cadena.

En PHP existen dos sistemas de expresiones regulares:

- **Expresiones regulares POSIX extendido:**
  - Introducidas en PHP 2.0b7 en 1996, y consideras **obsoletas** a partir de PHP 5.3.0 (junio de 2009).
  - Siguen la recomendación POSIX 1003.2. POSIX (Portable Operating System Interface) es un conjunto de normas redactadas por el IEEE (The Institute of Electrical and Electronics Engineers) que definen la API de Unix.
  - Las funciones correspondientes empiezan por "ereg".
- **Expresiones regulares compatibles con Perl (en inglés, PCRE):**
  - Introducidas en PHP 4.2.0 en 2002.
  - Siguen la sintaxis y semánticas del lenguaje de programación Perl 5. PHP 4.2.0 y posteriores incluyen la biblioteca de código libre escrita en C PCRE (Perl Compatible Regular Expressions). En el siguiente enlace podemos ver la sintaxis en detalle para la creación de expresiones regulares PCRE en php.
  - Las funciones correpondientes empiezan por "preg".

Dado que las funciones ereg (POSIX extendido) se consideran obsoletas a partir de PHP 5.3.0 (publicado en junio de 2009), se recomienda utilizar **únicamente** las **funciones preg** (PCRE).

2. Introducción

Las **expresiones regulares** van encerradas en **delimitadores**, que son cualquier par de caracteres no alfanuméricos (excepto la barra invertida \ y el espacio en blanco). Los delimitadores más utilizados son las **barras oblícuas** (*/*), aunque también se pueden utilizar **almohadillas #**, **virgulillas ~**... Además de **()**, **[]**, **{}**, o **<>**.

```
/([A-Z])\w+/
+expresión+
```

```
#[a-zA-Z]#
```

Para **escapar valores** se utiliza la **barra invertida **.

Si el **delimitador** aparece frecuentemente dentro del patrón, se suele utilizar otro delimitador para que sea más legible:

```
/http:\/\//
#http://#
```

Podemos ver un ejemplo con una de las funciones más básicas, **preg_match()**, que devuelve 1 si encuentra el patrón, y 0 si no lo encuentra:

```
$abecedario = "abcdefghijklmnñopqrstuvwz";
echo preg_match("/abc/", $abecedario); // Devuelve 1
```

*Para el ejemplo anterior sería más recomendable utilizar *strpos()* y *strstr()*, que son mucho más rápidas.

3. Encontrar extremos de un string

Encontrar un patrón al principio de un string

Si queremos **comprobar si un string empieza con unos caracteres concretos**, podemos hacerlo de la siguiente forma:

```
$direccion = "Calle Cuéllar";
if(preg_match("/^Calle/", $direccion))
{
 echo "Es una calle";
} else {
    echo "No es una calle";
}
```

Hemos comprobado que la dirección es una calle. La **regex (expresión regular)** está delimitada por dos **barras oblícuas (*/*)** y comienzan con un **signo de intercalación** o **caret ^**, un **metacarácter** que sirve para indicar el comienzo que queremos que tenga el string buscado.

Coinciden porque ambas tienen C mayúscula, no lo harían si la **regex** o el *string $direccion* fueran en minúscula. Por defecto, las búsquedas son **sensibles a mayúsculas y minúsculas**, para cambiar esto se puede añadir una **i** de **insensitive** al final. Esto es lo que se llama un **modificador de patrón**:

```
...// Alteramos esta línea en el código anterior:
if(preg_match("/^calle/i", $direccion))
... // Daría igual que fuera CALle, cAllE, callE...
```

Encontrar un patrón al final de un string

Se hace de forma similar a encontrar un patrón al principio de un string. En lugar de usar **^** se usa el **símbolo de dólar $** al final:

```
$cadena = "Esto es un saludo: hola";
echo preg_match("/hola$/", $cadena); // Devuelve: 1
```

4. Metacaracteres

El símbolo caret ^ y el de dólar $ son **metacaracteres**. El poder de las expresiones regulares viene dado por la capacidad de incluir alternativas y repeticiones en el patrón. Éstas están codificadas en el patrón por el uso de **metacaracteres**, que no tienen una representación en el *string*, sino que modifican la interpretación del patrón.

Se pueden dividir en dos: los que se interpretan fuera de los corchetes y los que se interpretan dentro:

Metacaracteres fuera de los corchetes

Metacarácter	Descripción
\	escape
^	inicio de string o línea
$	final de string o línea
.	coincide con cualquier carácter excepto nueva línea
[	inicio de la definición de clase carácter
]	fin de la definición de clase carácter
|	inicio de la rama alternativa
(	inicio de subpatrón
)	fin de subpatrón
?	amplía el significado del subpatrón, cuantificador 0 ó 1, y hace lazy los cuantificadores greedy
*	cuantificador 0 o más
+	cuantificador 1 o más
{	inicio de cuantificador mín/máx

}	**fin de cuantificador mín/máx**

Metacaracteres dentro de los corchetes:

Metacarácter	Descripción
\	carácter de escape general
^	niega la clase, pero sólo si es el primer carácter
-	indica el rango de caracteres

Si queremos escapar cualquiera de los metacaracteres anteriores tan sólo hay que utilizar el **backslash **:

```
// Ejemplo de una multiplicación con el metacarácter *
$string = '3*4';
echo preg_match('/^3\*4/', $string); // Devuelve 1
```

Debido a las interpretaciones especiales del entrecomillado en PHP, si se quiere comparar \ mediante la expresión regular \\, se ha de usar "\\\\" ó '\\\\'.

Metacarácter []

Ya hemos visto el funcionamiento de ^ y $. Vamos a ver ahora el de los **corchetes []**, que representan una **clase carácter**, esto es un conjunto de caracteres que queremos hacer coincidir, y puede ser un conjunto de caracteres, por ejemplo *[hola]*, o en forma de agrupación como *[a-z]*

Hay que tener en cuenta que los rangos como [a-z], no tienen en cuenta la letra ñ ni las vocales con tilde, para que las tenga en cuenta hay que añadirlas a la clase:

```
$string = "hola";
echo preg_match("/h[aeiou]la/", $string); // Devuelve 1
// También coincidiría con hala, hela, hila y hula
```

Si en la **clase carácter** se presenta un rango, como a-z, significa que puede coincidir con cualquier carácter del abecedario en minúsculas. En mayúsculas el rango es A-Z.

Si se incluye un metacarácter dentro de una clase carácter se interpreta literalmente (salvo las excepciones ya nombradas \, ^ y -). Ejemplo: [abc$] podría coincidir con cualquiera de los 4 caracteres: a, b, c o $.

Un **caret ^** al principio de una clase carácter se interpreta como negación, y encontrará cualquier carácter salvo los incluídos dentro de la clase carácter:

```
$string = '123456789';
preg_match("/[^2468]/", $string);  //Devuelve 0 porque la cadena empieza por 1
```

El caret sólo es interpretado de esa forma si se coloca al principio, después se interpreta de forma literal. Podemos **filtrar** por ejemplo **las vocales, espacios de un string** o **mayúsculas**:

```
// Filtrar vocales:
$string = 'No coger vocales';
echo preg_match("/[^aeiou]/", $string); // 0

echo preg_match("/[^ aeiou]/", $string); // 1
// Filtrar consonantes añadimos la Ñ para que la tenga en cuenta:
```

Metacarácter \

Se utiliza como **escape de metacaracteres**:

```
$string = '[]';
// Interpreta los corchetes como corchetes
echo preg_match("/\[\]/", $string, $matches);
// Interpreta los corchetes como corchetes dentro de una clase carácter
echo preg_match("/[\[\]]/", $string, $matches);
```

O para el uso de secuencias especiales (explicadas después).

Metacarácter *

Encuentra **cero o más ocurrencias** del carácter que le precede:

```
$string1 = "hla";
$string2 = "hola";
$string3 = "hooooooola";
echo preg_match("/ho*la/", $string1); // Devuelve 1
echo preg_match("/ho*la/", $string2); // Devuelve 1
echo preg_match("/ho*la/", $string3); // Devuelve 1
```

Metacarácter +

Encuentra **una o más ocurrencias** del caracter que le precede:

```
$string1 = "hla";
$string2 = "hola";
$string3 = "hooooooola";
echo preg_match("/ho+la/", $string1); // Devuelve 0
echo preg_match("/ho+la/", $string2); // Devuelve 1
echo preg_match("/ho+la/", $string3); // Devuelve 1
```

Metacarácter ?

Encuentra **0 o 1 ocurrencias** del carácter o expresión regular que le precede. Se utiliza para hacer algún carácter opcional:

```
$string1 = "hla";
$string2 = "hola";
$string3 = "hooooooola";
echo preg_match("/ho?la/", $string1); // Devuelve 1
echo preg_match("/ho?la/", $string2); // Devuelve 1
```

```
echo preg_match("/ho?la/", $string3); // Devuelve 0
```

Metacarácter {}

Las llaves indican el número exacto de matches del carácter o grupo de caracteres que le preceden:

```
$string1 = "hooola";
$string2 = "hooooola";
$string2 = "hoola";
$string4 = "houiela";
$string5 = "hla";
echo preg_match("/ho{3}la/", $string1); // Devuelve 1
// Si se indica {n,} quiere decir que al menos n elementos
echo preg_match("/ho{3,}la/", $string2); // Devuelve 1
// Si se indica {n, m} quiere decir un número entre n y m
echo preg_match("/ho{2,3}la/", $string3); // Devuelve 1
echo preg_match("/h[aeiou]{4}la/", $string4); // Devuelve 1
echo preg_match("/ho{0}la/", $string5); // Devuelve 1
```

Metacarácter |

El operador de **barra vertical |** permite alternar entre posibles coincidencias:

```
$string = "Este es un Estado de Estados Unidos";
echo preg_match("/este|esto|esta/i", $string); // Devuelve 1
```

Metacarácter ()

Los paréntesis en las expresiones regulares permiten **crear subpatrones**, como pequeños patrones dentro del patrón principal:

```
$string1 = "Llegaré pronto que voy volando";
$string2 = "Llegaré tarde que voy andando";
echo preg_match("/(and|vol)ando/i", $string1); // Devuelve 1
echo preg_match("/(and|vol)ando/i", $string2); // Devuelve 1
```

5. Secuencias especiales

Las secuencias especiales utilizan el **backslash ** y son **conjuntos predefinidos de caracteres** que permiten reducir el tamaño de las **expresiones regulares**:

Secuencia	Coincidencia	Equivalencia
\d	Cualquier carácter numérico	[0-9]
\D	Cualquier carácter no numérico	[^0-9]

\s	**Cualquier espacio**	**[\t\n\r\f\v]**
\S	**Cualquiera que no sea espacio**	**[^ \t\n\r\f\v]**
\w	**Cualquier carácter alfanumérico. No tiene en cuenta tildes ni ñ**	**[a-zA-Z0-9_]**
\W	**Cualquier carácter no alfanumérico**	**[^a-zA-Z0-9_]**

Algunos ejemplos:

```
$string = 'abc123^+*<>?';
// Encontrar sólo caracteres alfanuméricos:
echo preg_match("/[\w]/", $string); // Devuelve 1
// Cualquier carácter:
echo preg_match("//", $string); // Devuelve 1
// Encuentra todo menos caracteres alfanuméricos:
echo preg_match_all("/[\W]/", $string, $matches);  Devuelve 1
// Encuentra todo string que comienza con número:
$string = '4 pares de zapatos';
echo preg_match("/^\d/", $string); // Devuelve 1
// Encuentra los espacios
$string = "Esto es un\nejemplo con espacios\n y salto de linea";
echo preg_match_all("/\s/", $string); // Devuelve 1
```

6. Algunas funciones PCRE

preg_replace

La función **preg_replace** realiza una **búsqueda** y una **sustitución de una expresión regular**.

mixed **preg_replace (***mixed* **$pattern,** *mixed* **$replacement,** *mixed* **$subject [,** *int* **$limit = -1 [,** *int* **&$count]])**

Busca en *$subject* coincidencias con el patrón *$pattern* y las sustituye por *$replacement*. Se puede indicar el límite de sustituciones con *$limit* que por defecto son infinitas, y se puede pedir el número de veces que se ha reemplazado el patrón con *$count*.

```
$string = 'Vamos a reemplazar la palabra coche';
$cambio = preg_replace("/coche/", 'moto', $string);
echo $cambio; // Devuelve: Vamos a reemplazar la palabra moto
```

Con este ejemplo anterior sería más rápido emplear **str_replace()**, pero *preg_replace()* permite hacer cosas más complejas mediante expresiones regulares, además de aceptar arrays también:

```
$string = 'Vamos a cambiar el animal perro de color verde';
$patrones = array();
$patrones[0] = '/perro/';
$patrones[1] = '/verde/';
$sustituciones = array();
$sustituciones[0] = 'gato';
$sustituciones[1] = 'azul';
echo preg_replace($patrones, $sustituciones, $string);
```

```
// Devuelve: Vamos a cambiar el animal gato de color azul
```

preg_filter

mixed **preg_filter (***mixed* **$pattern,** *mixed* **$replacement,** *mixed* **$subject [,** *int* **$limit = -1 [,** *int* **&$count]])**

Es igual que *pre_replace()*, pero *preg_filter()* sólo devuelve los resultados donde hay una coincidencia.

```
$array = ["1 perro", "gato", "avestruz", "1 cerdo"];
$cambio = preg_filter('/\d/', "un", $array);
var_dump($cambio);
/*
array (size=2)
  0 => string 'un perro' (length=8)
  3 => string 'un cerdo' (length=8)
*/
```

preg_grep

array **preg_grep (***string* **$pattern,** *array* **$input [,** *int* **$flags = 0])**

Devuelve un nuevo array con los elementos de *$input* que coinciden con *$pattern*.

Se puede establecer el flag **PREG_GREP_INVERT**, para devolver un array con los elementos que **no** coinciden con *$pattern*.

```
$array = ["1 perro", "gato", "avestruz", "1 cerdo"];
$otro = preg_grep('/\d/', $array);
var_dump($otro);
/*
array (size=2)
  0 => string '1 perro' (length=7)
  3 => string '1 cerdo' (length=7)
*/
```

preg_split

array **preg_split (***string* **$pattern,** *string* **$subject [,** *int* **$limit = -1 [,** *int* **$flags = 0]])**

Divide un string mediante una expresión regular. Se pueden especificar los siguientes flags:

PREG_SPLIT_NO_EMPTY. Sólo se devolverán los elementos que no están vacíos.

PREG_SPLIT_DELIM_CAPTURE. Las expresiones entre paréntesis en el patrón delimitador serán capturadas y devueltas.

PREG_SPLIT_OFFSET_CAPTURE. Por cada coincidencia, la posición del string también será añadida.

```
$frase = "Esto es PHP";
$palabras = preg_split('/\s/', $frase, null, PREG_SPLIT_OFFSET_CAPTURE);
var_dump($palabras);
/*
array (size=3)
  0 =>
    array (size=2)
      0 => string 'Esto' (length=4)
      1 => int 0
  1 =>
    array (size=2)
      0 => string 'es' (length=2)
```

```
    1 => int 5
  2 =>
    array (size=2)
      0 => string 'PHP' (length=3)
      1 => int 8
*/
```

Si no se necesita la precisión de las expresiones regulares es mejor emplear las funciones *explode()* o *str_split()*.

Fuentes:

HYPERLINK "http://php.net/manual/es/reference.pcre.pattern.syntax.php" \n _blank php.n