

Expresiones regulares en PHP

Las expresiones regulares son patrones que permiten hacer búsquedas y comparaciones de strings muy precisas

Contenido modificable

Si ves errores o quieres modificar/añadir contenidos, puedes [crear un pull request](#). Gracias



Una **expresión regular** o **regex** es un **patrón** que se compara con una **cadena objetivo** de izquierda a derecha, carácter a carácter. La biblioteca **PCRE (Perl Compatible Regular Expressions)** es una extensión incorporada en PHP que permite utilizar **expresiones regulares** en funciones para **buscar, comparar y sustituir strings en PHP**.

Índice de contenido

1. Introducción	6. Modificadores
2. Encontrar extremos de un string	7. Declaraciones
3. Metacaracteres : [] \ . * + ? {} ()	8. Greedy y lazy
4. Subpatrones	9. Funciones PCRE
5. Secuencias especiales	

1. Introducción

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

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

2. 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** 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.

Coincidir 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**

```
...// 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
```

Usar ^ y \$ o \A y \Z para indicar el inicio y final de un string

Si se trabaja con un **string con múltiples líneas** (primera línea\nsegunda línea) y se quiere saber si el patrón coincide en el principio y el final de éstas, se pueden utilizar los símbolos ^ y \$ para indicar, respectivamente, el **principio y el final de cada línea de un string**. Para ello es necesario utilizar también el **modificador de patrón m**:

```
$direccion = "Calle Cuéllar\nCalle Augusta";  
// preg_match_all devuelve el número de matches  
// Hemos incluido dos modificadores de patrón, i y m:  
echo preg_match_all("/^calle/im", $direccion); // Devuelve: 2
```

Si no importan las líneas dentro de un string y se quiere saber exactamente el principio y el final de un string sin importar saltos de línea se usan **\A** y **\Z**:

```
$texto = "esto\nes\n\ttexto\nmultilinea\nlinea";  
// Da igual que se incluya el modificador multilinea con \A y \Z:  
echo preg_match_all('/\linea\Z/im', $texto); // Devuelve: 1  
echo preg_match_all('/\linea$/im', $texto); // Devuelve: 2  
echo preg_match_all('/\Aes/im', $texto); // Devuelve: 1  
echo preg_match_all('/^es/im', $texto); // Devuelve: 2
```

3. 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:

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

1. 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

```
$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';
// Si se añade un tercer elemento a preg_match_all devuelve un array con lo encontrado
preg_match_all("/[^2468]/", $string, $matches);
var_dump($matches); // Devuelve array con 1, 3, 5, 7 y 9
```

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_all("/[^aeiou]/", $string, $matches); // 10
// Filtrar vocales y espacios:
echo preg_match_all("/[^ aeiou]/", $string, $matches); // 8
// Filtrar consonantes:
$string = "NO coger MAYUSCULAS solo minusculas"; // 23
echo preg_match_all("/[^A-Z]/", $string, $matches);
```

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).

El punto coincide una vez con cualquier carácter excepto los saltos de linea como \r y \n:

```
$string = "6\nf^R\nL";
echo preg_match_all("./.", $string); // Devuelve 5: 6, f, ^, R y L
echo preg_match_all("//", $string); // Devuelve 8 (todos)
// Con la secuencia especial \s se muestra el número de líneas/espacios:
echo preg_match_all("/\s/", $string); // Devuelve 2 (sólo las líneas)
```

Metacarácter *

Encuentra cero o más ocurrencias del carácter que le precede:

```
$string1 = "hla";
$string2 = "hola";
$string3 = "ooooooooola";
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 carácter que le precede:

```
$string1 = "hla";
$string2 = "hola";
$string3 = "ooooooooola";
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 = "ooooooooola";
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 {}

```
$string1 = "hoooola";
$string2 = "hooooola";
$string2 = "hoola";
$string4 = "houielala";
$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_all("/este|esto|esta/i", $string); // Devuelve 3
```

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_all("/(and|vol)ando/i", $string1); // Devuelve 1
echo preg_match_all("/(and|vol)ando/i", $string2); // Devuelve 1
```

_pregmatch() captura tanto los **patrones** como los **subpatrones** en un array:

```
$string = "Llegaré pronto que voy volando";
preg_match_all("/(and|vol)ando/i", $string, $matches);
var_dump($matches);
/*
array (size=2)
  0 =>
    array (size=1)
      0 => string 'volando' (length=7)
  1 =>
    array (size=1)
      0 => string 'vol' (length=3)
*/
```

4. Subpatrones

porción en paréntesis del string que coincide con el subpatrón:

```
$string = "marrón";
preg_match("/ma(r{2})ó)n/", $string, $matches);
var_dump($matches);
/*
array (size=2)
  0 => string 'marrón' (length=7)
  1 => string 'rró' (length=4)
*/
```

Las capturas se pueden referenciar después mediante un **backslash** y un **número**, dependiendo de la posición de la referencia: \1, \2, \3... hasta \9. Estas referencias se pueden hacer dentro de la misma **expresión regular**, aunque nunca dentro de las **clases carácter**:

```
$string = "hola y hola";
preg_match('/(hola) y \1/', $string, $matches);
var_dump($matches);
/*
array (size=2)
  0 => string 'hola y hola' (length=11)
  1 => string 'hola' (length=4)
*/
```

El metacarácter **?** modifica el comportamiento del subpatrón, las tres posibilidades más básicas son las siguientes:

- **(?:)**. Hace que el subpatrón no sea capturable.
- **(?=)**. Encuentra el subpatrón después de la expresión principal sin incluirlo en el resultado.
- **(?!)**. Opuesto del anterior, especifica el subpatrón que no puede ir después de la expresión principal. El subpatrón tampoco aparece en el resultado.

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]

Algunos ejemplos:

```
$string = 'abc123^+*<>?';
// Encontrar sólo caracteres alfanuméricos:
echo preg_match_all("/[\w]/", $string, $matches); // Devuelve 6
// Cualquier carácter:
echo preg_match_all("//", $string, $matches); // Devuelve 13
// Encuentra todo menos caracteres alfanuméricos:
echo preg_match_all("/[\W]/", $string, $matches); // Devuelve 6
// Encuentra todo string que comienza con número:
$string = '4 pares de zapatos';
echo preg_match_all("/^\d/", $string); // Devuelve 1
// Encuentra los espacios
$string = "Esto es un\n ejemplo con espacios\n y salto de linea";
echo preg_match_all("/\s/", $string); // Devuelve 10
```

6. Modificadores

Se utilizar para modificar la forma en que se evalúan las **expresiones regulares**. Se incluyen justo después del **delimitador de la expresión regular** y se pueden utilizar varios a la vez. Algunos de ellos son:

Modificador	Significado
i	Insensible a mayúsculas y minúsculas
m	Múltiples líneas
x	Se pueden añadir comentarios
u	Strings de patrones y objetivos son tratados como UTF-8

Algunos ejemplos:

```
// Ejemplo de insensibilidad a mayúsculas y minúsculas:
$string = "AaBb";
echo preg_match_all("/[ab]/i", $string); // Devuelve 4
// Ejemplo múltiples líneas, detecta un número al principio de cada una:
$string = "1Esto\n2son\n3varias\n4lineas";
echo preg_match_all("/^[0-9]/m", $string); // Devuelve 4
// Ejemplo de comentarios en expresión regular:
$string = "123 son\n3 números";
$regex = "
^ # buscar al principio de línea
[0-9] # cualquier número del 1 al 9
/mx";
echo preg_match_all($regex, $string); // Devuelve 2
```

7. Declaraciones

la expresión regular.

Declaración	Significado
\b	Límite de palabra
\B	NO límite de palabra
\A	Comienzo de string
\Z	Final de string o de línea
\z	Final de string
\G	Primer match en el string

Algunos ejemplos:

```
// Ejemplo para encontrar palabra exacta
$string1 = "Es ahora";
$string2 = "Es la hora";
echo preg_match("/\bhora\b/", $string1); // Devuelve 0
echo preg_match("/\bhora\b/", $string2); // Devuelve 1
// *Si se utiliza \b dentro de una clase carácter no hace de límite sino de backspace

// Ejemplo de límite y no límite:
$string = "Es ahora";
// La palabra debe acabar en hora, pero debe tener al menos algún carácter antes:
echo preg_match("/\Bhora\b/", $string); // Devuelve 1
$string = "Es la hora";
// Ahora vemos como no coincide, pues no tiene un carácter antes:
echo preg_match("/\Bhora\b/", $string); // Devuelve 0
```

8. Greedy y lazy

Por defecto PCRE es **greedy**, esto significa que va a intentar **coincidir con el mayor número de caracteres** que pueda a no ser que indiquemos lo contrario, que sea **lazy**, y que intente **coincidir con el menor número posible de caracteres**.

Para hacer un **patrón lazy** se usa el interrogante ?. Esto le dice a **Perl** que encuentre el menor número posible de los caracteres anteriores antes de continuar con la siguiente parte del patrón. También se puede hacer con el **modificador /U**.

Cuando un **regex es lazy** se produce **backtracking**, esto es, comprueba con más repeticiones el string, lo que hace que le cueste más comprobar el patrón. A no ser que se manejen bien las **expresiones regulares**, no es recomendable cambiar el **comportamiento por defecto greedy**.

```
// Ejemplo usando laziness con "?"
$string = "hla hola hoola hooola";
```



```
// Ejemplo usando la función preg_match()
$string = "hola hoola hooola";
preg_match("/h(.*)la/U", $string, $matches);
var_dump($matches); // hola
```

9. 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]
```

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
```

Es igual que `prereplace()`, pero `pregfilter()` sólo devuelve los resultados donde hay una coincidencia.

```
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_match

```
int preg_match (string $pattern, string $subject [, array &$matches [,
```

Hace una comparación en `$subject` con una expresión regular `$pattern`.

Si se proporciona `$matches`, se crea este array con los resultados de la búsqueda. `$matches[0]` contiene el texto que coincidió con el patrón completo, `$matches[1]` el texto que coincidió con el primer subpatrón, y así sucesivamente.

Si se indica el flag `PREG_OFFSET_CAPTURE`, por cada coincidencia la posición del string también se devolverá.

Devuelve 1 si `$pattern` coincide con `$subject`, 0 si no y false si ocurrió un error.

```
$string = "1Esto 2es 3un 4string";
preg_match('/\d\w{2} /', $string, $matches, PREG_OFFSET_CAPTURE);
var_dump($matches);
/*
array (size=1)
```

```
0 -> string 2es (length=4)
  1 => int 6
*/

```

- `preg_match_all`

```
int preg_match_all (string $pattern, string $subject [, array &$matches [, int $flags [, int $offset [, int $limit ]]]])

```

Busca en `$subject` todas las coincidencias de la expresión regular `$pattern` y las introduce en `$matches` en el orden especificado por `$flags`.

Con PREG_PATTERN_ORDER los resultados se ordenan de forma que `$matches[0]` es un array de coincidencias del patrón completo, `$matches[1]` es un array de coincidencias del primer subpatrón, etc.

Con PREG_SET_ORDER `$matches[0]` es un array del primer conjunto de coincidencias, `$matches[1]` es un array del segundo conjunto, etc.

PREG_OFFSET_CAPTURE añade un elemento por cada coincidencia con la posición del string.

```
$string = "1Esto 2es 3un 4string";
preg_match_all('/\d\w{2} /', $string, $matches, PREG_SET_ORDER);
var_dump($matches);
/*
array (size=2)
  0 =>
    array (size=1)
      0 => string '2es ' (length=4)
  1 =>
    array (size=1)
      0 => string '3un ' (length=4)
*/

```

- `preg_replace_callback`

```
mixed preg_replace_callback (mixed $pattern, callable $callback, mixed

```

Funciona de la misma forma que `_preg_replace()`, pero en lugar de un string `$replacement` se especifica un `$callback`.

```
$miEdad = "Tengo 26 años\n";
// El callback:
function cumpleaños($edad)
{

```

```
miCumpleaños = preg_replace('(\d{2})', '$1', $miCumpleaños);
echo $miCumpleaños; // Tengo 27 años
```

- `preg_split`

```
array preg_split (string $pattern, string $subject [, int $limit = -1
```

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 `_strsplit()`.



[Diego Lázaro](#)[Angular](#)[Libro de PHP](#)