



Docs_

Expresiones regulares (avanzado)



1. Patrones y banderas (flags)
2. Clases de caracteres
3. Unicode: bandera "u" y clase `\p{...}`
4. Anclas: inicio `^` y final `$` de cadena
5. Modo multilínea de anclas `^` `$`, bandera "m"
6. Límite de palabra: `\b`
7. Escapando, caracteres especiales
8. Conjuntos y rangos `[...]`
9. Cuantificadores `+`, `*`, `?` y `{n}`
10. Cuantificadores codiciosos y perezosos
11. Grupos de captura
12. Referencias inversas en patrones: `\N` y `\k<nombre>`
13. Alternancia `(O) |`
14. Lookahead y lookbehind (revisar delante/detrás)
15. Backtracking catastrófico
16. Indicador adhesivo "y", buscando en una posición.
17. Métodos de `RegExp` y `String`



1. Patrones y banderas (flags)

Las expresiones regulares son patrones que proporcionan una forma poderosa de buscar y reemplazar texto.

En JavaScript, están disponibles a través del objeto [RegExp](#), además de integrarse en métodos de cadenas.

Expresiones Regulares

Una expresión regular (también "regexp", o simplemente "reg") consiste en un *patrón* y *banderas* opcionales.

Hay dos sintaxis que se pueden usar para crear un objeto de expresión regular.

La sintaxis "larga":

```
regexp = new RegExp("patrón", "banderas");
```

Y el "corto", usando barras "/":

```
regexp = /pattern/; // sin banderas  
regexp = /pattern/gmi; // con banderas g,m e i (para ser cubierto pronto)
```

Las barras `/.../` le dicen a JavaScript que estamos creando una expresión regular. Juegan el mismo papel que las comillas para las cadenas.

En ambos casos, `regexp` se convierte en una instancia de la clase incorporada `RegExp`.

La principal diferencia entre estas dos sintaxis es que el patrón que utiliza barras `/.../` no permite que se inserten expresiones (como los literales de plantilla de cadena con `${...}`). Son completamente estáticos.

Las barras se utilizan cuando conocemos la expresión regular en el momento de escribir el código, y esa es la situación más común. Mientras que `new RegExp`, se usa con mayor frecuencia cuando necesitamos crear una expresión regular "sobre la marcha" a partir de una cadena generada dinámicamente. Por ejemplo:

```
let tag = prompt("¿Qué etiqueta quieres encontrar?", "h2");  
  
igual que /<h2>/ si respondió "h2" en el mensaje anterior
```



Banderas

Las expresiones regulares pueden usar banderas que afectan la búsqueda.

Solo hay 6 de ellas en JavaScript:

i

Con esta bandera, la búsqueda no distingue entre mayúsculas y minúsculas: no hay diferencia entre A y a (consulte el ejemplo a continuación).

g

Con esta bandera, la búsqueda encuentra todas las coincidencias, sin ella, solo se devuelve la primera coincidencia.

m

Modo multilínea (cubierto en el capítulo [Modo multilínea de anclas ^ \\$, bandera "m"](#)).

s

Habilita el modo "dotall", que permite que un punto . coincida con el carácter de línea nueva \n (cubierto en el capítulo [Clases de caracteres](#)).

u

Permite el soporte completo de Unicode. La bandera permite el procesamiento correcto de pares sustitutos. Más del tema en el capítulo [Unicode: bandera "u" y clase \p{...}](#).

y

Modo "adhesivo": búsqueda en la posición exacta del texto (cubierto en el capítulo [Indicador adhesivo "y", buscando en una posición.](#))



Buscando: str.match

Como se mencionó anteriormente, las expresiones regulares se integran con los métodos de cadena.

El método `str.match(regex)` busca todas las coincidencias de `regex` en la cadena `str`.

Tiene 3 modos de trabajo:

1. Si la expresión regular tiene la bandera `g`, devuelve un arreglo de todas las coincidencias:

```
let str = "We will, we will rock you";

alert( str.match(/we/gi) ); // We,we (un arreglo de 2 subcadenas que coinciden)
```

Tenga en cuenta que tanto `we` como `we` se encuentran, porque la bandera `i` hace que la expresión regular no distinga entre mayúsculas y minúsculas.

2. Si no existe dicha bandera, solo devuelve la primera coincidencia en forma de arreglo, con la coincidencia completa en el índice `0` y algunos detalles adicionales en las propiedades:

```
let str = "We will, we will rock you";

let result = str.match(/we/i); // sin la bandera g

alert( result[0] ); // We (1ra coincidencia)
alert( result.length ); // 1

// Detalles:
alert( result.index ); // 0 (posición de la coincidencia)
alert( result.input ); // We will, we will rock you (cadena fuente)
```

El arreglo puede tener otros índices, además de `0` si una parte de la expresión regular está encerrada entre paréntesis.



Expresiones Regulares

3. Y, finalmente, si no hay coincidencias, se devuelve `null` (no importa si hay una bandera `g` o no).

Este es un matiz muy importante. Si no hay coincidencias, no recibimos un arreglo vacío, sino que recibimos `null`. Olvidar eso puede conducir a errores, por ejemplo:

```
let matches = "JavaScript".match(/HTML/); // = null

if (!matches.length) { // Error: No se puede leer la propiedad 'length' de
  null
  alert("Error en la línea anterior");
}
```

Si queremos que el resultado sea siempre un arreglo, podemos escribirlo de esta manera:

```
let matches = "JavaScript".match(/HTML/) || [];

if (!matches.length) {
  alert("Sin coincidencias"); // ahora si trabaja
}
```

Reemplazando: `str.replace`

El método `str.replace(regex, replacement)` reemplaza las coincidencias encontradas usando `regex` en la cadena `str` con `replacement` (todas las coincidencias si está la bandera `g`, de lo contrario, solo la primera).

Por ejemplo:

```
// sin la bandera g
alert( "We will, we will".replace(/we/i, "I") ); // I will, we will

// con la bandera g
alert( "We will, we will".replace(/we/ig, "I") ); // I will, I will
```



Expresiones Regulares

El segundo argumento es la cadena de replacement. Podemos usar combinaciones de caracteres especiales para insertar fragmentos de la coincidencia:

Símbolos	Acción en la cadena de reemplazo
<code>\$&</code>	inserta toda la coincidencia
<code>\$`</code>	inserta una parte de la cadena antes de la coincidencia
<code>\$'</code>	inserta una parte de la cadena después de la coincidencia
<code>\$n</code>	si n es un número de 1-2 dígitos, entonces inserta el contenido de los paréntesis n-ésimo.
<code>\$<name></code>	inserta el contenido de los paréntesis con el nombre dado, más del tema en el capítulo Grupos de captura
<code>\$\$</code>	inserta el carácter \$

Un ejemplo con `$&`:

```
alert( "Me gusta HTML".replace(/HTML/, "$& y JavaScript") ); // Me gusta HTML y JavaScript
```

Pruebas: `regexp.test`

El método `regexp.test(str)` busca al menos una coincidencia, si se encuentra, devuelve `true`, de lo contrario `false`.

```
let str = "Me gusta JavaScript";
let regexp = /GUSTA/i;

alert( regexp.test(str) ); // true
```

Más adelante en este capítulo estudiaremos más expresiones regulares, exploraremos más ejemplos y también conoceremos otros métodos.

La información completa sobre métodos se proporciona en el artículo [No se encontró el artículo "regexp-method"](#).



Expresiones Regulares

Resumen

- Una expresión regular consiste en un patrón y banderas opcionales: `g`, `i`, `m`, `u`, `s`, `y`.
- Sin banderas y símbolos especiales (que estudiaremos más adelante), la búsqueda por expresión regular es lo mismo que una búsqueda de subcadena.
- El método `str.match(regex)` busca coincidencias: devuelve todas si hay una bandera `g`, de lo contrario, solo la primera.
- El método `str.replace(regex, replacement)` reemplaza las coincidencias encontradas usando `regex` con `replacement`: devuelve todas si hay una bandera `g`, de lo contrario solo la primera.
- El método `regex.test(str)` devuelve `true` si hay al menos una coincidencia, de lo contrario, devuelve `false`.



2. Clases de caracteres

Considera una tarea práctica: tenemos un número de teléfono como "+7(903)-123-45-67", y debemos convertirlo en número puro: 79031234567.

Para hacerlo, podemos encontrar y eliminar cualquier cosa que no sea un número. La clase de caracteres pueden ayudar con eso.

Una *clase de caracteres* es una notación especial que coincide con cualquier símbolo de un determinado conjunto.

Para empezar, exploremos la clase "dígito". Está escrito como `\d` y corresponde a "cualquier dígito".

Por ejemplo, busquemos el primer dígito en el número de teléfono:

```
let str = "+7(903)-123-45-67";

let regexp = /\d/;

alert( str.match(regexp) ); // 7
```

Sin la bandera (flag) `g`, la expresión regular solo busca la primera coincidencia, es decir, el primer dígito `\d`.

Agreguemos la bandera `g` para encontrar todos los dígitos:

```
let str = "+7(903)-123-45-67";

let regexp = /\d/g;

alert( str.match(regexp) ); // array de coincidencias: 7,9,0,3,1,2,3,4,5,6,7

// hagamos el número de teléfono de solo dígitos:
alert( str.match(regexp).join('') ); // 79031234567
```

Esa fue una clase de caracteres para los dígitos. También hay otras.



Expresiones Regulares

Las más usadas son:

`\d` ("d" es de dígito)

Un dígito: es un caracter de 0 a 9.

`\s` ("s" es un espacio)

Un símbolo de espacio: incluye espacios, tabulaciones `\t`, líneas nuevas `\n` y algunos otros caracteres raros, como `\v`, `\f` y `\r`.

`\w` ("w" es carácter de palabra)

Un carácter de palabra es: una letra del alfabeto latino o un dígito o un guión bajo `_`. Las letras no latinas (como el cirílico o el hindi) no pertenecen al `\w`.

Por ejemplo, `\d\s\w` significa un "dígito" seguido de un "carácter de espacio" seguido de un "carácter de palabra", como 1 a.

Una expresión regular puede contener símbolos regulares y clases de caracteres.

Por ejemplo, `css\d` coincide con una cadena `css` con un dígito después:

```
let str = "¿Hay CSS4?";
let regexp = /CSS\d/

alert( str.match(regexp) ); // CSS4
```

También podemos usar varias clases de caracteres:

```
alert( "Me gusta HTML5!".match(/\s\w\w\w\w\d/) ); // ' HTML5'
```

La coincidencia (cada clase de carácter de la expresión regular tiene el carácter resultante correspondiente):

`\s \w \w \w \w \d`
I love HTML5



Expresiones Regulares

Clases inversas

Para cada clase de caracteres existe una "clase inversa", denotada con la misma letra, pero en mayúscula.

El "inverso" significa que coincide con todos los demás caracteres, por ejemplo:

`\D`

Sin dígitos: cualquier carácter excepto `\d`, por ejemplo, una letra.

`\S`

Sin espacio: cualquier carácter excepto `\s`, por ejemplo, una letra.

`\W`

Sin carácter de palabra: cualquier cosa menos `\w`, por ejemplo, una letra no latina o un espacio.

Al comienzo del capítulo vimos cómo hacer un número de teléfono solo de números a partir de una cadena como `+7(903)-123-45-67`: encontrar todos los dígitos y unirlos.

```
let str = "+7(903)-123-45-67";  
  
alert( str.match(/\d/g).join('') ); // 79031234567
```

Una forma alternativa y más corta es usar el patrón sin dígito `\D` para encontrarlos y eliminarlos de la cadena:

```
let str = "+7(903)-123-45-67";  
  
alert( str.replace(/\D/g, "") ); // 79031234567
```



Expresiones Regulares

Un punto es “cualquier carácter”

El patrón punto (.) es una clase de caracteres especial que coincide con “cualquier carácter excepto una nueva línea”.

Por ejemplo:

```
alert( "Z".match(/./) ); // Z
```

O en medio de una expresión regular:

```
let regexp = /CS.4/;

alert( "CSS4".match(regexp) ); // CSS4
alert( "CS-4".match(regexp) ); // CS-4
alert( "CS 4".match(regexp) ); // CS 4 (el espacio también es un carácter)
```

Tenga en cuenta que un punto significa “cualquier carácter”, pero no la “ausencia de un carácter”. Debe haber un carácter para que coincida:

```
alert( "CS4".match(/CS.4/) ); // null, no coincide porque no hay caracteres entre S y 4
```

Punto es igual a la bandera “s” que literalmente retorna cualquier carácter

Por defecto, *punto* no coincide con el carácter de línea nueva \n.

Por ejemplo, la expresión regular A.B coincide con A, y luego B con cualquier carácter entre ellos, excepto una línea nueva \n:

```
alert( "A\nB".match(/A.B/) ); // null (sin coincidencia)
```

Hay muchas situaciones en las que nos gustaría que *punto* signifique literalmente “cualquier carácter”, incluida la línea nueva.

Eso es lo que hace la bandera s. Si una expresión regular la tiene, entonces . coincide literalmente con cualquier carácter:

```
alert( "A\nB".match(/A.B/s) ); // A\nB (coincide!)
```



Expresiones Regulares

No soportado en IE

La bandera `s` no está soportada en IE.

Afortunadamente, hay una alternativa, que funciona en todas partes. Podemos usar una expresión regular como `[\s\S]` para que coincida con "cualquier carácter". (Este patrón será cubierto en el artículo [Conjuntos y rangos \[...\]](#)).

```
alert( "A\nB".match(/A[\s\S]B/) ); // A\nB (coincide!)
```

El patrón `[\s\S]` literalmente dice: "con carácter de espacio O sin carácter de espacio". En otras palabras, "cualquier cosa". Podríamos usar otro par de clases complementarias, como `[\d\D]`, eso no importa. O incluso `[^]`, que significa que coincide con cualquier carácter excepto nada.

También podemos usar este truco si queremos ambos tipos de "puntos" en el mismo patrón: el patrón actual . comportándose de la manera regular ("sin incluir una línea nueva"), y la forma de hacer coincidir "cualquier carácter" con el patrón `[\s\S]` o similar.

Presta atención a los espacios

Por lo general, prestamos poca atención a los espacios. Para nosotros, las cadenas `1-5` y `1 - 5` son casi idénticas.

Pero si una expresión regular no tiene en cuenta los espacios, puede que no funcione.

Intentemos encontrar dígitos separados por un guión:

```
alert( "1 - 5".match(/\d-\d/) ); // null, sin coincidencia!
```

Vamos a arreglarlo agregando espacios en la expresión regular `\d - \d`:

```
alert( "1 - 5".match(/\d - \d/) ); // 1 - 5, funciona ahora
```

// o podemos usar la clase `\s`:

```
alert( "1 - 5".match(/\d\s-\s\d/) ); // 1 - 5, tambien funciona
```

Un espacio es un carácter. Igual de importante que cualquier otro carácter.

No podemos agregar o eliminar espacios de una expresión regular y esperar que funcione igual.

En otras palabras, en una expresión regular todos los caracteres importan, los espacios también.



Expresiones Regulares

Resumen

Existen las siguientes clases de caracteres:

- `\d` – dígitos.
- `\D` – sin dígitos.
- `\s` – símbolos de espacio, tabulaciones, líneas nuevas.
- `\S` – todo menos `\s`.
- `\w` – letras latinas, dígitos, guión bajo `'_'`.
- `\W` – todo menos `\w`.
- `.` – cualquier carácter, si la expresión regular usa la bandera `'s'`, de otra forma cualquiera excepto **línea nueva** `\n`.

...¡Pero eso no es todo!

La codificación Unicode, utilizada por JavaScript para las cadenas, proporciona muchas propiedades para los caracteres, como: a qué idioma pertenece la letra (si es una letra), es un signo de puntuación, etc.

Se pueden hacer búsquedas usando esas propiedades. Y se requiere la bandera `u`, analizada en el siguiente artículo.



3. Unicode: bandera "u" y clase \p{...}

JavaScript utiliza [codificación Unicode](#) para las cadenas. La mayoría de los caracteres están codificados con 2 bytes, esto permite representar un máximo de 65536 caracteres.

Ese rango no es lo suficientemente grande como para codificar todos los caracteres posibles, es por eso que algunos caracteres raros se codifican con 4 bytes, por ejemplo como x (X matemática) o ☺ (una sonrisa), algunos sinogramas, etc.

Aquí los valores unicode de algunos caracteres:

Carácter	Unicode	conteo de Bytes en unicode
a	0x0061	2
≈	0x2248	2
x	0x1d4b3	4
y	0x1d4b4	4
☺	0x1f604	4

Entonces los caracteres como a e ≈ ocupan 2 bytes, mientras que los códigos para x , y y ☺ son más largos, tienen 4 bytes.

Hace mucho tiempo, cuando se creó el lenguaje JavaScript, la codificación Unicode era más simple: no había caracteres de 4 bytes. Por lo tanto, algunas características del lenguaje aún los manejan incorrectamente.

Por ejemplo, aquí `length` interpreta que hay dos caracteres:

```
alert('☺'.length); // 2
alert('X'.length); // 2
```

...Pero podemos ver que solo hay uno, ¿verdad? El punto es que `length` maneja 4 bytes como dos caracteres de 2 bytes. Eso es incorrecto, porque debe considerarse como uno solo (el llamado "par sustituto", puede leer sobre ellos en el artículo [Strings](#)).

Por defecto, las expresiones regulares manejan los "caracteres largos" de 4 bytes como un par de caracteres de 2 bytes cada uno. Y, como sucede con las cadenas, eso puede conducir a resultados extraños. Lo veremos un poco más tarde, en el artículo [No se encontró el artículo "regexp-character-sets-and-range"](#).



Expresiones Regulares

A diferencia de las cadenas, las expresiones regulares tienen la bandera `u` que soluciona tales problemas. Con dicha bandera, una expresión regular maneja correctamente los caracteres de 4 bytes. Y podemos usar la búsqueda de propiedades Unicode, que veremos a continuación.

Propiedades Unicode `\p{...}`

Cada carácter en Unicode tiene varias propiedades. Describen a qué “categoría” pertenece el carácter, contienen información diversa al respecto.

Por ejemplo, si un carácter tiene la propiedad `Letter`, significa que pertenece a un alfabeto (de cualquier idioma). Y la propiedad `Number` significa que es un dígito: tal vez árabe o chino, y así sucesivamente.

Podemos buscar caracteres por su propiedad, usando `\p{...}`. Para usar `\p{...}`, una expresión regular debe usar también `u`.

Por ejemplo, `\p{Letter}` denota una letra en cualquiera de los idiomas. También podemos usar `\p{L}`, ya que `L` es un alias de `Letter`. Casi todas las propiedades tienen alias cortos.

En el ejemplo a continuación se encontrarán tres tipos de letras: inglés, georgiano y coreano.

```
let str = "A ზ Ⴑ";

alert( str.match(/\p{L}/gu) ); // A,ზ,Ⴑ
alert( str.match(/\p{L}/g) ); // null (sin coincidencia, como no hay bandera "u")
```



Expresiones Regulares

Estas son las principales categorías y subcategorías de caracteres:

- Letter (Letra) L:
 - lowercase (minúscula) L_l
 - modifier (modificador) L_m,
 - titlecase (capitales) L_t,
 - uppercase (mayúscula) L_u,
 - other (otro) L_o.
- Number (número) N:
 - decimal digit (dígito decimal) N_d,
 - letter number (número de letras) N_l,
 - other (otro) N_o.
- Punctuation (puntuación) P:
 - connector (conector) P_c,
 - dash (guión) P_d,
 - initial quote (comilla inicial) P_i,
 - final quote (comilla final) P_f,
 - open (abre) P_s,
 - close (cierra) P_e,
 - other (otro) P_o.
- Mark (marca) M (acentos etc):
 - spacing combining (combinación de espacios) M_c,
 - enclosing (encerrado) M_e,
 - non-spacing (sin espaciado) M_n.
- Symbol (símbolo) S:
 - currency (moneda) S_c,
 - modifier (modificador) S_k,
 - math (matemática) S_m,
 - other (otro) S_o.
- Separator (separador) Z:
 - line (línea) Z_l,
 - paragraph (párrafo) Z_p,
 - space (espacio) Z_s.
- Other (otros) C:
 - control C_c,
 - format (formato) C_f,
 - not assigned (sin asignación) C_n,
 - private use (uso privado) C_o,
 - surrogate (sustituto) C_s.

Entonces, por ejemplo si necesitamos letras en minúsculas, podemos escribir `\p{Ll}`, signos de puntuación: `\p{P}` y así sucesivamente.



Expresiones Regulares

También hay otras categorías derivadas, como:

- `Alphabetic` (alfabético) (`Alfa`), incluye letras `L`, más números de letras `N1` (por ejemplo, `XII` – un carácter para el número romano 12), y otros símbolos `Other_Alphabetic` (`OA1pha`).
- `Hex_Digit` incluye dígitos hexadecimales: `0-9`, `a-f`.
- ...Y así.

Unicode admite muchas propiedades diferentes, la lista completa es muy grande, estas son las referencias:

- Lista de todas las propiedades por carácter: <https://unicode.org/cldr/utility/character.jsp> (enlace no disponible).
- Lista de caracteres por propiedad: <https://unicode.org/cldr/utility/list-unicodeset.jsp>. (enlace no disponible)
- Alias cortos para propiedades: <https://www.unicode.org/Public/UCD/latest/ucd/PropertyValueAliases.txt>.
- Aquí una base completa de caracteres Unicode en formato de texto, con todas las propiedades: <https://www.unicode.org/Public/UCD/latest/ucd/>.

Ejemplo: números hexadecimales

Por ejemplo, busquemos números hexadecimales, escritos como `xFF` donde `F` es un dígito hexadecimal (`0...9` o `A...F`).

Un dígito hexadecimal se denota como `\p{Hex_Digit}`:

```
let regexp = /x\p{Hex_Digit}\p{Hex_Digit}/u;

alert("número: xAF".match(regexp)); // xAF
```



Expresiones Regulares

Ejemplo: sinogramas chinos

Busquemos sinogramas chinos.

Hay una propiedad Unicode `Script` (un sistema de escritura), que puede tener un valor: `Cyrillic`, `Greek`, `Arabic`, `Han` (chino), etc. [lista completa](#).

Para buscar caracteres de un sistema de escritura dado, debemos usar `Script=<value>`, por ejemplo para letras cirílicas: `\p{sc=Cyrillic}`, para sinogramas chinos: `\p{sc=Han}`, y así sucesivamente:

```
let regexp = /\p{sc=Han}/gu; // devuelve sinogramas chinos

let str = `Hello Привет 你好 123_456`;

alert( str.match(regexp) ); // 你,好
```

Ejemplo: moneda

Los caracteres que denotan una moneda, como \$, €, ¥, tienen la propiedad unicode `\p{Currency_Symbol}`, el alias corto: `\p{Sc}`.

Usémoslo para buscar precios en el formato “moneda, seguido de un dígito”:

```
let regexp = /\p{Sc}\d/gu;
let str = `Precios: $2, €1, ¥9`;
alert( str.match(regexp) ); // $2,€1,¥9
```

Más adelante, en el artículo [Cuantificadores +, *, ? y {n}](#) veremos cómo buscar números que contengan muchos dígitos.

Resumen

La bandera `u` habilita el soporte de Unicode en expresiones regulares.

Eso significa dos cosas:

1. Los caracteres de 4 bytes se manejan correctamente: como un solo carácter, no dos caracteres de 2 bytes.
2. Las propiedades Unicode se pueden usar en las búsquedas: `\p{...}`.

Con las propiedades Unicode podemos buscar palabras en determinados idiomas, caracteres especiales (comillas, monedas), etc.



4. Anclas: inicio ^ y final \$ de cadena

Los patrones caret (del latín carece) ^ y dólar \$ tienen un significado especial en una expresión regular. Se llaman "anclas".

El patrón caret ^ coincide con el principio del texto y dólar \$ con el final.

Por ejemplo, probemos si el texto comienza con Mary:

```
let str1 = "Mary tenía un corderito";  
alert( /^Mary/.test(str1) ); // true
```

El patrón ^Mary significa: "inicio de cadena y luego Mary".

Similar a esto, podemos probar si la cadena termina con nieve usando nieve\$:

```
let str1 = "su vellón era blanco como la nieve";  
alert( /nieve$/.test(str1) ); // true
```

En estos casos particulares, en su lugar podríamos usar métodos de cadena `beginWith/endsWith`. Las expresiones regulares deben usarse para pruebas más complejas.

Prueba para una coincidencia completa

Ambos anclajes ^...\$ se usan juntos a menudo para probar si una cadena coincide completamente con el patrón. Por ejemplo, para verificar si la entrada del usuario está en el formato correcto.

Verifiquemos si una cadena esta o no en formato de hora 12:34. Es decir: dos dígitos, luego dos puntos y luego otros dos dígitos.

En el idioma de las expresiones regulares eso es `\d\d:\d\d`:

```
let goodInput = "12:34";  
let badInput = "12:345";  
  
let regexp = /^d\d:d\d$/;  
alert( regexp.test(goodInput) ); // true  
alert( regexp.test(badInput) ); // false
```

La coincidencia para `\d\d:\d\d` debe comenzar exactamente después del inicio de texto ^, y seguido inmediatamente, el final \$.

Toda la cadena debe estar exactamente en este formato. Si hay alguna desviación o un carácter adicional, el resultado es `false`.



Expresiones Regulares

Las anclas se comportan de manera diferente si la bandera `m` está presente. Lo veremos en el próximo artículo.

Las anclas tienen "ancho cero"

Las anclas `^` y `$` son pruebas. Ellas tienen ancho cero.

En otras palabras, no coinciden con un carácter, sino que obligan al motor regexp a verificar la condición (inicio/fin de texto).



5. Modo multilínea de anclas ^ \$, bandera "m"

El modo multilínea está habilitado por el indicador `m`.

Solo afecta el comportamiento de `^` y `$`.

En el modo multilínea, coinciden no solo al principio y al final de la cadena, sino también al inicio/final de la línea.

Buscando al inicio de línea ^

En el siguiente ejemplo, el texto tiene varias líneas. El patrón `/^\d/gm` toma un dígito desde el principio de cada línea:

```
let str = `1er lugar: Winnie
2do lugar: Piglet
3er lugar: Eeyore`;

console.log( str.match(/^\d/gm) ); // 1, 2, 3
```

Sin la bandera `m` solo coincide el primer dígito:

```
let str = `1er lugar: Winnie
2do lugar: Piglet
3er lugar: Eeyore`;

console.log( str.match(/^\d/g) ); // 1
```

Esto se debe a que, de forma predeterminada, un caret `^` solo coincide al inicio del texto y en el modo multilínea, al inicio de cualquier línea.

Por favor tome nota:

"Inicio de una línea" significa formalmente "inmediatamente después de un salto de línea": la prueba `^` en modo multilínea coincide en todas las posiciones precedidas por un carácter de línea nueva `\n`.

Y al comienzo del texto.



Expresiones Regulares

Buscando al final de la línea \$

El signo de dólar \$ se comporta de manera similar.

La expresión regular `\d$` encuentra el último dígito en cada línea

```
let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;

console.log( str.match(/\d$/gm) ); // 1,2,3
```

Sin la bandera `m`, dólar \$ solo coincidiría con el final del texto completo, por lo que solo se encontraría el último dígito.

Por favor tome nota:

“Fin de una línea” significa formalmente “inmediatamente antes de un salto de línea”: la prueba \$ en el modo multilínea coincide en todas las posiciones seguidas por un carácter de línea nueva `\n`.

Y al final del texto.

Buscando `\n` en lugar de `^` \$

Para encontrar una línea nueva, podemos usar no solo las anclas `^` y `$`, sino también el carácter de línea nueva `\n`.

¿Cual es la diferencia? Veamos un ejemplo.

Buscamos `\d\n` en lugar de `\d$`:

```
let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;

console.log( str.match(/\d\n/g) ); // 1\n,2\n
```

Como podemos ver, hay 2 coincidencias en lugar de 3.

Esto se debe a que no hay una línea nueva después de 3 (sin embargo, hay un final de texto, por lo que coincide con \$).

Otra diferencia: ahora cada coincidencia incluye un carácter de línea nueva `\n`. A diferencia de las anclas `^` y `$`, que solo prueban la condición (inicio/final de una línea), `\n` es un carácter, por lo que se hace parte del resultado.

Entonces, un `\n` en el patrón se usa cuando necesitamos encontrar caracteres de línea nueva, mientras que las anclas se usan para encontrar algo “al principio/al final” de una línea.



6. Límite de palabra: \b

Un límite de palabra \b es una prueba, al igual que ^ y \$.

Cuando el motor regex (módulo de programa que implementa la búsqueda de expresiones regulares) se encuentra con \b, comprueba que la posición en la cadena es un límite de palabra.

Hay tres posiciones diferentes que califican como límites de palabras:

- Al comienzo de la cadena, si el primer carácter de cadena es un carácter de palabra \w.
- Entre dos caracteres en la cadena, donde uno es un carácter de palabra \w y el otro no.
- Al final de la cadena, si el último carácter de la cadena es un carácter de palabra \w.

Por ejemplo, la expresión regular \bJava\b se encontrará en Hello, Java!, donde Java es una palabra independiente, pero no en Hello, JavaScript!.

```
alert( "Hello, Java!".match(/\bJava\b/) ); // Java
alert( "Hello, JavaScript!".match(/\bJava\b/) ); // null
```

En la cadena Hello, Java! las flechas que se muestran corresponden a \b, ver imagen:

↓ ↓ ↓ ↓
Hello, Java!

Entonces, coincide con el patrón \bHello\b, porque:

1. Al comienzo de la cadena coincide con la primera prueba: \b.
2. Luego coincide con la palabra Hello.
3. Luego, la prueba \b vuelve a coincidir, ya que estamos entre o y una coma.

El patrón \bHello\b también coincidiría. Pero no \bHel\b (porque no hay límite de palabras después de l) y tampoco Java!\b (porque el signo de exclamación no es un carácter común \w, entonces no hay límite de palabras después de eso).

```
alert( "Hello, Java!".match(/\bHello\b/) ); // Hello
alert( "Hello, Java!".match(/\bJava\b/) ); // Java
alert( "Hello, Java!".match(/\bHel\b/) ); // null (sin coincidencia)
alert( "Hello, Java!".match(/\bJava!\b/) ); // null (sin coincidencia)
```



Expresiones Regulares

Podemos usar `\b` no solo con palabras, sino también con dígitos.

Por ejemplo, el patrón `\b\d\d\b` busca números independientes de 2 dígitos. En otras palabras, busca números de 2 dígitos que están rodeados por caracteres diferentes de `\w`, como espacios o signos de puntuación (o texto de inicio/fin).

```
alert( "1 23 456 78".match(/\b\d\d\b/g) ); // 23,78  
alert( "12,34,56".match(/\b\d\d\b/g) ); // 12,34,56
```

El límite de palabra `\b` no funciona para alfabetos no latinos

La prueba de límite de palabra `\b` verifica que debe haber un `\w` en un lado de la posición y "no `\w`"- en el otro lado.

Pero `\w` significa una letra latina a-z (o un dígito o un guión bajo), por lo que la prueba no funciona para otros caracteres, p.ej.: letras cirílicas o jeroglíficos.



7. Escapando, caracteres especiales

Como hemos visto, una barra invertida \ se usa para denotar clases de caracteres, p.ej. \d. Por lo tanto, es un carácter especial en expresiones regulares (al igual que en las cadenas regulares).

También hay otros caracteres especiales que tienen un significado especial en una expresión regular, tales como [] { } () \ ^ \$. | ? * +. Se utilizan para hacer búsquedas más potentes.

No intentes recordar la lista: pronto nos ocuparemos de cada uno de ellos por separado y los recordarás fácilmente.

Escapando

Digamos que queremos encontrar literalmente un punto. No "cualquier carácter", sino solo un punto.

Para usar un carácter especial como uno normal, agrégalo con una barra invertida: \.

A esto se le llama "escape de carácter".

Por ejemplo:

```
alert( "Capítulo 5.1".match(/\\d\\.\\d/) ); // 5.1 (¡Coincide!)
alert( "Capítulo 511".match(/\\d\\.\\d/) ); // null (buscando un punto real \\.)
```

Los paréntesis también son caracteres especiales, por lo que si los buscamos, deberíamos usar \(. El siguiente ejemplo busca una cadena "g()":

```
alert( "función g()".match(/g\\(\\)/) ); // "g()"
```

Si estamos buscando una barra invertida \, como es un carácter especial tanto en cadenas regulares como en expresiones regulares, debemos duplicarlo.

```
alert( "1\\2".match(/\\\\/) ); // '\\'
```



Expresiones Regulares

Una barra

Un símbolo de barra `'/'` no es un carácter especial, pero en JavaScript se usa para abrir y cerrar expresiones regulares: `/...pattern.../`, por lo que también debemos escaparlos.

Así es como se ve la búsqueda de una barra `'/'`:

```
alert( "/" .match(/\\/)) ; // '/'
```

Por otro lado, si no estamos usando `/.../`, pero creamos una expresión regular usando `new RegExp`, entonces no necesitamos escaparlos:

```
alert( "/" .match(new RegExp("/")) ) ; // encuentra /
```

new RegExp

Si estamos creando una expresión regular con `new RegExp`, entonces no tenemos que escapar la barra `/`, pero sí otros caracteres especiales.

Por ejemplo, considere esto:

```
let regexp = new RegExp("\\d\\.\\d");  
  
alert( "Capítulo 5.1".match(regexp) ); // null
```

En uno de los ejemplos anteriores funcionó la búsqueda con `/\\d\\.\\d/`, pero `new RegExp("\\d\\.\\d")` no funciona, ¿por qué?

La razón es que las barras invertidas son "consumidas" por una cadena. Como podemos recordar, las cadenas regulares tienen sus propios caracteres especiales, como `\\n`, y se usa una barra invertida para escapar esos caracteres especiales de cadena.

Así es como se percibe `"\\d\\.\\d"`:

```
alert("\\d\\.\\d"); // d.d
```

Las comillas de cadenas "consumen" barras invertidas y las interpretan como propias, por ejemplo:

- `\\n` – se convierte en un carácter de línea nueva,
- `\\u1234` – se convierte en el carácter Unicode con dicho código,
- ...Y cuando no hay un significado especial: como `\\d` o `\\z`, entonces la barra invertida simplemente se elimina.



Expresiones Regulares

Así que `new RegExp` toma una cadena sin barras invertidas. ¡Por eso la búsqueda no funciona!

Para solucionarlo, debemos duplicar las barras invertidas, porque las comillas de cadena convierten `\\` en `\`:

```
let regStr = "\\d\\.\\d";  
alert(regStr); // \d\.\d (ahora está correcto)  
  
let regexp = new RegExp(regStr);  
  
alert( "Capítulo 5.1".match(regexp) ); // 5.1
```

Resumen

- Para buscar literalmente caracteres especiales [`\` `^` `$` `.` `|` `?` `*` `+` `(` `)`, se les antepone una barra invertida `\` ("escaparlos").
- Se debe escapar `/` si estamos dentro de `/.../` (pero no dentro de `new RegExp`).
- Al pasar una cadena a `new RegExp`, se deben duplicar las barras invertidas `\\`, porque las comillas de cadena consumen una.



8. Conjuntos y rangos [...]

Varios caracteres o clases de caracteres entre corchetes [...] significa "buscar cualquier carácter entre los dados".

Conjuntos

Por ejemplo, [eao] significa cualquiera de los 3 caracteres: 'a', 'e', o 'o'.

A esto se le llama *conjunto*. Los conjuntos se pueden usar en una expresión regular junto con los caracteres normales:

```
// encontrar [t ó m], y luego "op"  
alert( "Mop top".match(/[tm]op/gi) ); // "Mop", "top"
```

Tenga en cuenta que aunque hay varios caracteres en el conjunto, corresponden exactamente a un carácter en la coincidencia.

Entonces, en el siguiente ejemplo no hay coincidencias:

```
// encuentra "V", luego [o ó i], luego "la"  
alert( "Voila".match(/V[oi]la/) ); // null, sin coincidencias
```

El patrón busca:

- V,
- después *una* de las letras [oi],
- después la.

Entonces habría una coincidencia para vo1a o v1la.

Rangos

Los corchetes también pueden contener *rangos de caracteres*.

Por ejemplo, [a-z] es un carácter en el rango de a a z, y [0-5] es un dígito de 0 a 5.

En el ejemplo a continuación, estamos buscando "x" seguido de dos dígitos o letras de A a F:

```
alert( "Excepción 0xAF".match(/x[0-9A-F][0-9A-F]/g) ); // xAF
```

Aquí [0-9A-F] tiene dos rangos: busca un carácter que sea un dígito de 0 a 9 o una letra de A a F.

Si también queremos buscar letras minúsculas, podemos agregar el rango a-f: [0-9A-Fa-f]. O se puede agregar la bandera i.



Expresiones Regulares

También podemos usar clases de caracteres dentro de los [...].

Por ejemplo, si quisiéramos buscar un carácter de palabra `\w` o un guion `-`, entonces el conjunto es `[\w-]`.

También es posible combinar varias clases, p.ej.: `[\s\d]` significa "un carácter de espacio o un dígito".

Las clases de caracteres son abreviaturas (o atajos) para ciertos conjuntos de caracteres.

Por ejemplo:

- `\d` – es lo mismo que `[0-9]`,
- `\w` – es lo mismo que `[a-zA-Z0-9_]`,
- `\s` – es lo mismo que `[\t\n\v\f\r]`, además de otros caracteres de espacio raros de unicode.

Ejemplo: multi-idioma `\w`

Como la clase de caracteres `\w` es una abreviatura de `[a-zA-Z0-9_]`, no puede coincidir con sinogramas chinos, letras cirílicas, etc.

Podemos escribir un patrón más universal, que busque caracteres de palabra en cualquier idioma. Eso es fácil con las propiedades unicode: `[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]`.

Decifrémoslo. Similar a `\w`, estamos creando un conjunto propio que incluye caracteres con las siguientes propiedades unicode:

- Alfabético (Alpha) – para letras,
- Marca (M) – para acentos,
- Numero_Decimal (Nd) – para dígitos,
- Conector_Puntuación (Pc) – para guion bajo `'_'` y caracteres similares,
- Control_Unión (Join_C) – dos códigos especiales `200c` and `200d`, utilizado en ligaduras, p.ej. en árabe.

Un ejemplo de uso:

```
let regexp = /[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]/gu;

let str = `Hola 你好 12`;

// encuentra todas las letras y dígitos:
alert( str.match(regexp) ); // H,o,l,a,你,好,1,2
```



Expresiones Regulares

Por supuesto, podemos editar este patrón: agregar propiedades unicode o eliminarlas. Las propiedades Unicode se cubren con más detalle en el artículo [Unicode: bandera "u" y clase \p{...}](#).

Las propiedades Unicode no son soportadas por IE

Las propiedades Unicode `p{...}` no se implementaron en IE. Si realmente las necesitamos, podemos usar la biblioteca [XRegExp](#).

O simplemente usa rangos de caracteres en el idioma de tu interés, p.ej. `[а-я]` para letras cirílicas.

Excluyendo rangos

Además de los rangos normales, hay rangos “excluyentes” que se parecen a `[^...]`.

Están denotados por un carácter caret `^` al inicio y coinciden con cualquier carácter *excepto los dados*.

Por ejemplo:

- `[^aeyo]` – cualquier carácter excepto 'a', 'e', 'y' u 'o'.
- `[^0-9]` – cualquier carácter excepto un dígito, igual que `\D`.
- `[^\s]` – cualquier carácter sin espacio, igual que `\S`.

El siguiente ejemplo busca cualquier carácter, excepto letras, dígitos y espacios:

```
alert( "alice15@gmail.com".match(/[^\d\sA-Z]/gi) ); // @ y .
```

Escapando dentro de corchetes [...]

Por lo general, cuando queremos encontrar exactamente un carácter especial, necesitamos escaparlos con `\`. Y si necesitamos una barra invertida, entonces usamos `\\`, y así sucesivamente.

Entre corchetes podemos usar la gran mayoría de caracteres especiales sin escaparlos:

- Los símbolos `.` `+` `(` `)` nunca necesitan escape.
- Un guion `-` no se escapa al principio ni al final (donde no define un rango).
- Un carácter caret `^` solo se escapa al principio (donde significa exclusión).
- El corchete de cierre `]` siempre se escapa (si se necesita buscarlo).

En otras palabras, todos los caracteres especiales están permitidos sin escapar, excepto cuando significan algo entre corchetes.



Expresiones Regulares

Un punto `.` dentro de corchetes significa solo un punto. El patrón `[.,]` Buscaría uno de los caracteres: un punto o una coma.

En el siguiente ejemplo, la expresión regular `[-().^+]` busca uno de los caracteres `-().^+:`

```
// no es necesario escaparlos
let regexp = /[-().^+]/g;

alert( "1 + 2 - 3".match(regexp) ); // Coincide +, -
...Pero si decides escaparlos "por si acaso", no habría daño:

// Todo escapado
let regexp = /[\-\(\)\.\^\+]/g;

alert( "1 + 2 - 3".match(regexp) ); // funciona también: +, -
```

Rangos y la bandera (flag) "u"

Si hay pares sustitutos en el conjunto, se requiere la flag `u` para que funcionen correctamente.

Por ejemplo, busquemos `[xy]` en la cadena `x`:

```
alert( 'x'.match(/[xy]/) ); // muestra un carácter extraño, como [?]
// (la búsqueda se realizó incorrectamente, se devolvió medio carácter)
El resultado es incorrecto porque, por defecto, las expresiones regulares "no saben" sobre
pares sustitutos.
```

El motor de expresión regular piensa que la cadena `[xy]` no son dos, sino cuatro caracteres:

1. mitad izquierda de `x` (1),
2. mitad derecha de `x` (2),
3. mitad izquierda de `y` (3),
4. mitad derecha de `y` (4).

Sus códigos se pueden mostrar ejecutando:

```
for(let i = 0; i < 'xy'.length; i++) {
  alert('xy'.charAt(i)); // 55349, 56499, 55349, 56500
};
```

Entonces, el ejemplo anterior encuentra y muestra la mitad izquierda de `x`.

Si agregamos la flag `u`, entonces el comportamiento será correcto:

```
alert( 'x'.match(/[xy]/u) ); // x
```



Expresiones Regulares

Ocurre una situación similar cuando se busca un rango, como `[x-y]`.

Si olvidamos agregar la flag `u`, habrá un error:

```
'X'.match(/[X-Y]/); // Error: Expresión regular inválida
```

La razón es que sin la bandera `u` los pares sustitutos se perciben como dos caracteres, por lo que `[x-y]` se interpreta como `[<55349><56499>-<55349><56500>]` (cada par sustituto se reemplaza con sus códigos). Ahora es fácil ver que el rango 56499-55349 es inválido: su código de inicio 56499 es mayor que el último 55349. Esa es la razón formal del error.

Con la bandera `u` el patrón funciona correctamente:

```
// buscar caracteres desde X a Z  
alert( 'y'.match(/[X-Z]/u) ); // y
```




9. Cuantificadores +, *, ? y {n}

Digamos que tenemos una cadena como +7 (903) -123-45-67 y queremos encontrar todos los números en ella. Pero contrastando el ejemplo anterior, no estamos interesados en un solo dígito, sino en números completos: 7, 903, 123, 45, 67.

Un número es una secuencia de 1 o más dígitos `\d`. Para marcar cuántos necesitamos, podemos agregar un *cuantificador*.

Cantidad {n}

El cuantificador más simple es un número entre llaves: `{n}`.

Se agrega un cuantificador a un carácter (o a una clase de caracteres, o a un conjunto `[...]`, etc) y especifica cuántos necesitamos.

Tiene algunas formas avanzadas, veamos los ejemplos:

El recuento exacto: `{5}`

`\d{5}` Denota exactamente 5 dígitos, igual que `\d\d\d\d\d`.

El siguiente ejemplo busca un número de 5 dígitos:

```
alert( "Tengo 12345 años de edad".match(/\d{5}/) ); // "12345"
```

Podemos agregar `\b` para excluir números largos: `\b\d{5}\b`.

El rango: `{3,5}`, coincide 3-5 veces

Para encontrar números de 3 a 5 dígitos, podemos poner los límites en llaves: `\d{3,5}`

```
alert( "No tengo 12, sino 1234 años de edad".match(/\d{3,5}/) ); //  
"1234"
```

Podemos omitir el límite superior

Luego, una regexp `\d{3,}` busca secuencias de dígitos de longitud 3 o más:

```
alert( "No tengo 12, sino, 345678 años de edad".match(/\d{3,}/) ); //  
"345678"
```

Volvamos a la cadena +7(903)-123-45-67.



Expresiones Regulares

Un número es una secuencia de uno o más dígitos continuos. Entonces la expresión regular es `\d{1,}`:

```
let str = "+7(903)-123-45-67";

let numbers = str.match(/\d{1,}/g);

alert(numbers); // 7,903,123,45,67
```

Abreviaciones

Hay abreviaciones para los cuantificadores más usados:

+

Significa "uno o más", igual que `{1,}`.

Por ejemplo, `\d+` busca números:

```
let str = "+7(903)-123-45-67";

alert( str.match(/\d+/g) ); // 7,903,123,45,67
```

?

Significa "cero o uno", igual que `{0,1}`. En otras palabras, hace que el símbolo sea opcional.

Por ejemplo, el patrón `ou?r` busca o seguido de cero o uno u, y luego r.

Entonces, `colou?r` encuentra ambos color y colour:

```
let str = "¿Debo escribir color o colour?";

alert( str.match(/colou?r/g) ); // color, colour
```

*

Significa "cero o más", igual que `{0,}`. Es decir, el carácter puede repetirse muchas veces o estar ausente.

Por ejemplo, `\d0*` busca un dígito seguido de cualquier número de ceros (puede ser muchos o ninguno):

```
alert( "100 10 1".match(/\d0*/g) ); // 100, 10, 1
```



Expresiones Regulares

Compáralo con + (uno o más):

```
alert( "100 10 1".match(/\d+/g) ); // 100, 10
// 1 no coincide, ya que 0+ requiere al menos un cero
```

Más ejemplos

Los cuantificadores se usan con mucha frecuencia. Sirven como el “bloque de construcción” principal de expresiones regulares complejas, así que veamos más ejemplos.

Regexp para fracciones decimales (un número con coma flotante): `\d+\.\d+`

En acción:

```
alert( "0 1 12.345 7890".match(/\d+\.\d+/g) ); // 12.345
```

Regexp para una “etiqueta HTML de apertura sin atributos”, tales como `` o `<p>`.

1. La más simple: `/<[a-z]+>/i`

```
alert( "<body> ... </body>".match(/<[a-z]+>/gi) ); // <body>
```

La regexp busca el carácter '`<`' seguido de una o más letras latinas, y el carácter '`>`'.

2. Mejorada: `/<[a-z][a-z0-9]*>/i`

De acuerdo al estándar, el nombre de una etiqueta HTML puede tener un dígito en cualquier posición excepto al inicio, tal como `<h1>`.

```
alert( "<h1>Hola!</h1>".match(/<[a-z][a-z0-9]*>/gi) ); // <h1>
```

Regexp para “etiquetas HTML de apertura o cierre sin atributos”: `/<\/?[a-z][a-z0-9]*>/i`

Agregamos una barra opcional `/?` cerca del comienzo del patrón. Se tiene que escapar con una barra diagonal inversa, de lo contrario, JavaScript pensaría que es el final del patrón.

```
alert( "<h1>Hola!</h1>".match(/<\/?[a-z][a-z0-9]*>/gi) ); // <h1>, </h1>
```



Expresiones Regulares

Para hacer más precisa una regexp, a menudo necesitamos hacerla más compleja

Podemos ver una regla común en estos ejemplos: cuanto más precisa es la expresión regular, es más larga y compleja.

Por ejemplo, para las etiquetas HTML debemos usar una regexp más simple: `<\w+>`. Pero como HTML tiene normas estrictas para los nombres de etiqueta, `<[a-z][a-z0-9]*>` es más confiable.

¿Podemos usar `<\w+>` o necesitamos `<[a-z][a-z0-9]*>`?

En la vida real, ambas variantes son aceptables. Depende de cuán tolerantes podamos ser a las coincidencias "adicionales" y si es difícil o no eliminarlas del resultado por otros medios.



10. Cuantificadores codiciosos y perezosos

Los cuantificadores son muy simples a primera vista, pero de hecho pueden ser complicados.

Debemos entender muy bien cómo funciona la búsqueda si planeamos buscar algo más complejo que `/\d+/.`

Tomemos la siguiente tarea como ejemplo.

Tenemos un texto y necesitamos reemplazar todas las comillas "... " con comillas latinas: «...». En muchos países los tipógrafos las prefieren.

Por ejemplo: "Hola, mundo" debe convertirse en «Hola, mundo». Existen otras comillas, como „Witaj, świecie!” (Polaco) o 「你好, 世界」 (Chino), pero para nuestra tarea elegimos «...».

Lo primero que debe hacer es ubicar las cadenas entre comillas, y luego podemos reemplazarlas.

Una expresión regular como `/".+"/g` (una comilla, después algo, luego otra comilla) Puede parecer una buena opción, ¡pero no lo es!

Vamos a intentarlo:

```
let regex = /".+"/g;

let str = 'una "bruja" y su "escoba" son una';

alert( str.match(regex) ); // "bruja" y su "escoba"
```

...¡Podemos ver que no funciona según lo previsto!

En lugar de encontrar dos coincidencias "bruja" y "escoba", encuentra una: "bruja" y su "escoba".

Esto se puede describir como "la codicia es la causa de todo mal".

Búsqueda codiciosa

Para encontrar una coincidencia, el motor de expresión regular utiliza el siguiente algoritmo:

- Para cada posición en la cadena
 - Prueba si el patrón coincide en esta posición.
 - Si no hay coincidencia, ir a la siguiente posición.

Expresiones Regulares



Estas palabras comunes no son tan obvias para determinar por qué la regexp falla, así que elaboremos el funcionamiento de la búsqueda del patrón ".+".

1. El primer carácter del patrón es una comilla doble ".

El motor de expresión regular intenta encontrarla en la posición cero de la cadena fuente una "bruja" y su "escoba" son una, pero hay una u allí, por lo que inmediatamente no hay coincidencia.

Entonces avanza: va a la siguiente posición en la cadena fuente y prueba encontrar el primer carácter del patrón allí, falla de nuevo, y finalmente encuentra la comilla doble en la 3ra posición:

a "witch" and her "broom" is one

2. La comilla doble es detectada, y después el motor prueba encontrar una coincidencia para el resto del patrón. Prueba ver si el resto de la cadena objetivo satisface a .+.

En nuestro caso el próximo carácter de patrón es . (un punto). Que denota "cualquiere carácter excepto línea nueva", entonces la próxima letra de la cadena encaja 'w':

a "witch" and her "broom" is one

3. Entonces el punto (.) se repite por el cuantificador .+. El motor de expresión regular agrega a la coincidencia un carácter uno después de otro.

...¿Hasta cuando? Todos los caracteres coinciden con el punto, entonces se detiene hasta que alcanza el final de la cadena:

a "witch" and her "broom" is one

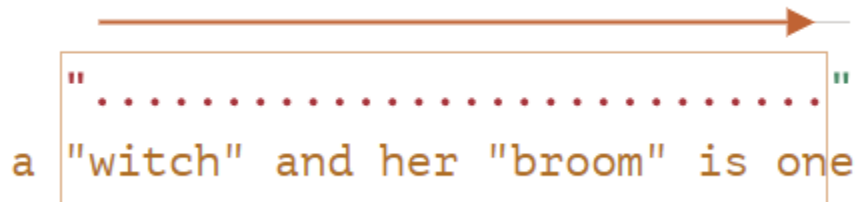


Expresiones Regulares

4. Ahora el motor finalizó el ciclo de `.` y prueba encontrar el próximo carácter del patrón. El cual es la comilla doble `"`. Pero hay un problema: la cadena ha finalizado, ¡no hay más caracteres!

El motor de expresión regular comprende que procesó demasiados `.` y *reinicia* la cadena.

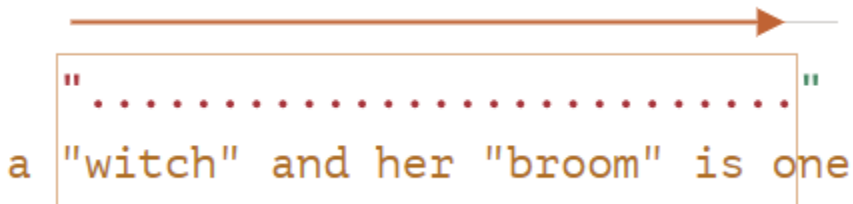
En otras palabras, acorta la coincidencia para el cuantificador en un carácter:



Ahora se supone que `.` finaliza un carácter antes del final de la cadena e intenta hacer coincidir el resto del patrón desde esa posición.

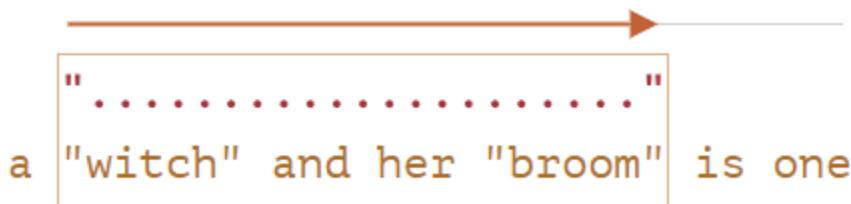
Si hubiera comillas doble allí, entonces la búsqueda terminaría, pero el último carácter es `'a'`, por lo que no hay coincidencia.

5. ...Entonces el motor disminuye el número de repeticiones de `.` en uno:



Las comillas dobles `'\"'` no coinciden con `'n'`.

6. El motor continua reiniciando la lectura de la cadena: decrementa el contador de repeticiones para `'.'` hasta que el resto del patrón (en nuestro caso `'\"'`) coincida:



7. La coincidencia está completa.
8. Entonces la primera coincidencia es `"bruja"` y su `"escoba"`. Si la expresión regular tiene la bandera `g`, entonces la búsqueda continuará desde donde termina la primera coincidencia. No hay más comillas dobles en el resto de la cadena son una, entonces no hay más resultados.



Expresiones Regulares

Probablemente no es lo que esperabamos, pero así es como funciona.

En el modo codicioso (por defecto) un carácter cuantificado se repite tantas veces como sea posible.

El motor de regexp agrega a la coincidencia tantos caracteres como pueda abarcar el patrón `.*`, y luego los abrevia uno por uno si el resto del patrón no coincide.

En nuestro caso queremos otra cosa. Es entonces donde el modo perezoso puede ayudar.

Modo perezoso

El modo perezoso de los cuantificadores es lo opuesto del modo codicioso. Eso significa: "repite el mínimo número de veces".

Podemos habilitarlo poniendo un signo de interrogación `?` después del cuantificador, entonces tendríamos `*?` o `+?` o incluso `??` para `'?'`.

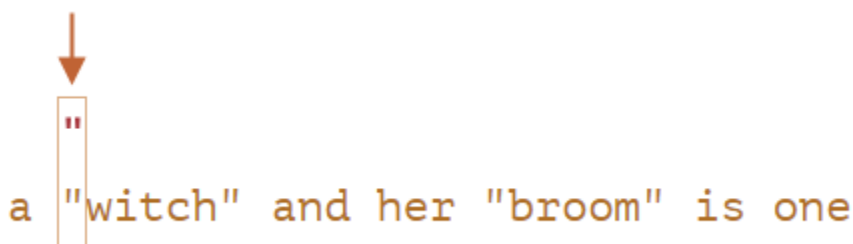
Aclarando las cosas: generalmente un signo de interrogación `?` es un cuantificador por si mismo (cero o uno), pero si se agrega *despues de otro cuantificador (o incluso el mismo)* toma otro significado, alterna el modo de coincidencia de codicioso a perezoso.

La regexp `/" .+?"/g` funciona como se esperaba: encuentra "bruja" y "escoba":

```
let regexp = /" .+?"/g;  
  
let str = 'una "bruja" y su "escoba" son una';  
  
alert( str.match(regexp) ); // "bruja", "escoba"
```

Para comprender claramente el cambio, rastreemos la búsqueda paso a paso.

1. El primer paso es el mismo: encuentra el inicio del patrón `'"` en la 5ta posición:



a "witch" and her "broom" is one

Expresiones Regulares



2. El siguiente paso también es similar: el motor encuentra una coincidencia para el punto '.':

a "witch" and her "broom" is one

3. Y ahora la búsqueda es diferente. Porque tenemos el modo perezoso activado en +?, el motor no prueba coincidir un punto una vez más, se detiene y prueba coincidir el resto del patrón ('') ahora mismo :

a "witch" and her "broom" is one

Si hubiera comillas dobles allí, entonces la búsqueda terminaría, pero hay una 'r', entonces no hay coincidencia.

4. Después el motor de expresión regular incrementa el número de repeticiones para el punto y prueba una vez más:

a "witch" and her "broom" is one

Falla de nuevo. Después el número de repeticiones es incrementado una y otra vez...

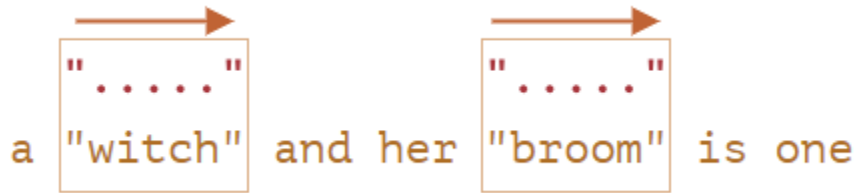
5. ...Hasta que se encuentre una coincidencia para el resto del patrón:

a "witch" and her "broom" is one



Expresiones Regulares

6. La próxima búsqueda inicia desde el final de la coincidencia actual y produce un resultado más:



En este ejemplo vimos cómo funciona el modo perezoso para `+`. Los cuantificadores `*` y `?` funcionan de manera similar, el motor regexp incrementa el número de repeticiones solo si el resto del patrón no coincide en la posición dada.

La pereza solo está habilitada para el cuantificador con `?`.

Otros cuantificadores siguen siendo codiciosos.

Por ejemplo:

```
alert( "123 456".match(/\d+ \d+?/) ); // 123 4
```

1. El patrón `\d+` intenta hacer coincidir tantos dígitos como sea posible (modo codicioso), por lo que encuentra 123 y se detiene, porque el siguiente carácter es un espacio `' '`.
2. Luego hay un espacio en el patrón, coincide.
3. Después hay un `\d+?`. El cuantificador está en modo perezoso, entonces busca un dígito 4 y trata de verificar si el resto del patrón coincide desde allí.

...Pero no hay nada en el patrón después de `\d+?`.

El modo perezoso no repite nada sin necesidad. El patrón terminó, así que terminamos. Tenemos una coincidencia 123 4.

Optimizaciones

Los motores modernos de expresiones regulares pueden optimizar algoritmos internos para trabajar más rápido. Estos trabajan un poco diferente del algoritmo descrito.

Pero para comprender cómo funcionan las expresiones regulares y construirlas, no necesitamos saber nada al respecto. Solo se usan internamente para optimizar cosas.

Las expresiones regulares complejas son difíciles de optimizar, por lo que la búsqueda también puede funcionar exactamente como se describe.



Expresiones Regulares

Enfoque alternativo

Con las regexps, por lo general hay muchas formas de hacer la misma cosa.

En nuestro caso podemos encontrar cadenas entre comillas sin el modo perezoso usando la regexp "[^"]+":

```
let regexp = /"[^"]+"/g;

let str = 'una "bruja" y su "escoba" son una';

alert( str.match(regexp) ); // "bruja", "escoba"
```

La regexp "[^"]+" devuelve el resultado correcto, porque busca una comilla doble '"' seguida por uno o más caracteres no comilla doble '[^"]', y luego la comilla doble de cierre.

Cuando la máquina de regexp busca el carácter no comilla '[^"]'+ se detiene la repetición cuando encuentra la comilla doble de cierre, y terminamos.

Nótese, ¡esta lógica no reemplaza al cuantificador perezoso!

Es solo diferente. Hay momentos en que necesitamos uno u otro.

Veamos un ejemplo donde los cuantificadores perezosos fallan y la variante funciona correctamente.

Por ejemplo, queremos encontrar enlaces en la forma , con cualquier href.

¿Cual expresión regular usamos?

La primera idea podría ser: //g.

Veámoslo:

```
let str = '...<a href="link" class="doc">...';
let regexp = /<a href=".*" class="doc">/g;

// ¡Funciona!
alert( str.match(regexp) ); // <a href="link" class="doc">
```



Expresiones Regulares

Funcionó. Pero veamos ¿que pasa si hay varios enlaces en el texto?

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href=".*" class="doc">/g;

// ¡Vaya! ¡Dos enlaces en una coincidencia!
alert( str.match(regexp) ); // <a href="link1" class="doc">... <a href="link2" class="doc">
```

Ahora el resultado es incorrecto por la misma razón del ejemplo de la bruja. El cuantificador `.*` toma demasiados caracteres.

La coincidencia se ve así:

```
<a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

Modifiquemos el patrón haciendo el cuantificador perezoso: `.*?`:

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href=".*?" class="doc">/g;

// ¡Funciona!
alert( str.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class="doc">
```

Ahora parece funcionar, hay dos coincidencias:

```
<a href="....." class="doc">    <a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

...Pero probemos ahora con una entrada de texto adicional:

```
let str = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let regexp = /<a href=".*?" class="doc">/g;

// ¡Coincidencia incorrecta!
alert( str.match(regexp) ); // <a href="link1" class="wrong">... <p style="" class="doc">
```



Expresiones Regulares

Ahora falla. La coincidencia no solo incluye el enlace, sino también mucho texto después, incluyendo `<p...>`.

¿Por qué?

Eso es lo que está pasando:

1. Primero la regexp encuentra un enlace inicial `<a href=`.
2. Después busca para el patrón `. *?`: toma un carácter (¡perezosamente!), verifica si hay una coincidencia para `" class="doc">` (ninguna).
3. Después toma otro carácter dentro de `. *?`, y así... hasta que finalmente alcanza a `" class="doc">`.

Pero el problema es que: eso ya está más allá del enlace `<a...>`, en otra etiqueta `<p>`. No es lo que queremos.

Esta es la muestra de la coincidencia alineada con el texto:

```
<a href="....." class="doc">
<a href="link1" class="wrong">... <p style="" class="doc">
```

Entonces, necesitamos un patrón que busque ``, pero ambas variantes, codiciosa y perezosa, tienen problemas.

La variante correcta puede ser: `href="[^"]*" class="doc"`. Esta tomará todos los caracteres dentro del atributo href hasta la comilla doble más cercana, justo lo que necesitamos.

Un ejemplo funcional:

```
let str1 = '...<a href="link1" class="wrong">... <p style=""
class="doc">...';
let str2 = '...<a href="link1" class="doc">... <a href="link2"
class="doc">...';
let regexp = /<a href="[^"]*" class="doc">/g;

// ¡Funciona!
alert( str1.match(regexp) ); // null, sin coincidencia, eso es correcto
alert( str2.match(regexp) ); // <a href="link1" class="doc">, <a
href="link2" class="doc">
```



Expresiones Regulares

Resumen

Los cuantificadores tienen dos modos de funcionamiento:

Codiciosa

Por defecto el motor de expresión regular prueba repetir el carácter cuantificado tantas veces como sea posible. Por ejemplo, `\d+` consume todos los posibles dígitos. Cuando es imposible consumir más (no hay más dígitos o es el fin de la cadena), entonces continúa hasta coincidir con el resto del patrón. Si no hay coincidencia entonces se decrementa el número de repeticiones (reinicios) y prueba de nuevo.

Perezoso

Habilitado por el signo de interrogación `?` después de un cuantificador. El motor de regex prueba la coincidencia para el resto del patrón antes de cada repetición del carácter cuantificado.

Como vimos, el modo perezoso no es una “panacea” de la búsqueda codiciosa. Una alternativa es una búsqueda codiciosa refinada, con exclusiones, como en el patrón `"[^"]+"`.



11. Grupos de captura

Una parte de un patrón se puede incluir entre paréntesis (...). Esto se llama "grupo de captura".

Esto tiene dos resultados:

1. Permite obtener una parte de la coincidencia como un elemento separado en la matriz de resultados.
2. Si colocamos un cuantificador después del paréntesis, se aplica a los paréntesis en su conjunto.

Ejemplos

Veamos cómo funcionan los paréntesis en los ejemplos.

Ejemplo: gogogo

Sin paréntesis, el patrón `go+` significa el carácter `g`, seguido por `o` repetido una o más veces. Por ejemplo, `goooo` o `goooooooo`.

Los paréntesis agrupan los caracteres juntos, por lo tanto `(go)+` significa `go`, `gogo`, `gogogo` etcétera.

```
alert( 'Gogogo now!'.match(/(go)+/ig) ); // "Gogogo"
```

Ejemplo: dominio

Hagamos algo más complejo: una expresión regular para buscar un dominio de sitio web.

Por ejemplo:

```
mail.com  
users.mail.com  
smith.users.mail.com
```

Como podemos ver, un dominio consta de palabras repetidas, un punto después de cada una excepto la última.

En expresiones regulares eso es `(\w+\.)+\w+`:

```
let regexp = /(\w+\.)+\w+/g;  
  
alert( "site.com my.site.com".match(regexp) ); // site.com,my.site.com
```



Expresiones Regulares

La búsqueda funciona, pero el patrón no puede coincidir con un dominio con un guión, por ejemplo, `my-site.com`, porque el guión no pertenece a la clase `\w`.

Podemos arreglarlo al reemplazar `\w` con `[\w-]` en cada palabra excepto el último: `([\w-]+\.\.)+\w+`.

Ejemplo: email

El ejemplo anterior puede ser extendido. Podemos crear una expresión regular para emails en base a esto.

El formato de email es: `name@domain`. Cualquier palabra puede ser el nombre, guiones y puntos están permitidos. En expresiones regulares esto es `[-.\w]+`.

El patrón:

```
let regexp = /[-.\w]+@([\w-]+\.\.)+[\w-]+/g;
```

```
alert("my@mail.com @ his@site.com.uk".match(regexp)); // my@mail.com,  
his@site.com.uk
```

Esa expresión regular no es perfecta, pero sobre todo funciona y ayuda a corregir errores de escritura accidentales. La única verificación verdaderamente confiable para un correo electrónico solo se puede realizar enviando una carta.

Contenido del paréntesis en la coincidencia (match)

Los paréntesis están numerados de izquierda a derecha. El buscador memoriza el contenido que coincide con cada uno de ellos y permite obtenerlo en el resultado.

El método `str.match(regexp)`, si `regexp` no tiene indicador (flag) `g`, busca la primera coincidencia y lo devuelve como un array:

1. En el índice 0: la coincidencia completa.
2. En el índice 1: el contenido del primer paréntesis.
3. En el índice 2: el contenido del segundo paréntesis.
4. ...etcétera...

Por ejemplo, nos gustaría encontrar etiquetas HTML `<.*?>`, y procesarlas. Sería conveniente tener el contenido de la etiqueta (lo que está dentro de los ángulos), en una variable por separado.

Envolvamos el contenido interior en paréntesis, de esta forma: `<(.*?)>`.



Expresiones Regulares

Ahora obtendremos ambos, la etiqueta entera <h1> y su contenido h1 en el array resultante:

```
let str = '<h1>Hello, world!</h1>';

let tag = str.match(/<(.*?)>/);

alert( tag[0] ); // <h1>
alert( tag[1] ); // h1
```

Grupos anidados

Los paréntesis pueden ser anidados. En este caso la numeración también va de izquierda a derecha.

Por ejemplo, al buscar una etiqueta en tal vez nos pueda interesar:

1. El contenido de la etiqueta como un todo: span class="my".
2. El nombre de la etiqueta: span.
3. Los atributos de la etiqueta: class="my".

Agreguemos paréntesis: <(([a-z]+)\s*([>]*))>.

Así es cómo se enumeran (izquierda a derecha, por el paréntesis de apertura):

```
      span class="my"
1 |-----|
<([([a-z]+\s*([>]*))>)>
  2 |-----|  3 |-----|
    span      class="my"
```



Expresiones Regulares

En acción:

```
let str = '<span class="my">';  
  
let regexp = /<(([a-z]+\s*([^\s]*)>))/;  
  
let result = str.match(regexp);  
alert(result[0]); // <span class="my">  
alert(result[1]); // span class="my"  
alert(result[2]); // span  
alert(result[3]); // class="my"
```

El índice cero de `result` siempre contiene la coincidencia completa.

Luego los grupos, numerados de izquierda a derecha por un paréntesis de apertura. El primer grupo se devuelve como `result[1]`. Aquí se encierra todo el contenido de la etiqueta.

Luego en `result[2]` va el grupo desde el segundo paréntesis de apertura `([a-z]+)` – nombre de etiqueta, luego en `result[3]` la etiqueta: `([^\s]*)`.

El contenido de cada grupo en el string:

1 `span class="my"`
`<((([a-z]+\s*([^\s]*)>))>`
2 `span` 3 `class="my"`



Expresiones Regulares

Grupos opcionales

Incluso si un grupo es opcional y no existe en la coincidencia (p.ej. tiene el cuantificador (...)?), el elemento array `result` correspondiente está presente y es igual a `undefined`.

Por ejemplo, consideremos la expresión regular `a(z)?(c)?`. Busca "a" seguida por opcionalmente "z", seguido por "c" opcionalmente.

Si lo ejecutamos en el string con una sola letra a, entonces el resultado es:

```
let match = 'a'.match(/a(z)?(c)?/);

alert( match.length ); // 3
alert( match[0] ); // a (coincidencia completa)
alert( match[1] ); // undefined
alert( match[2] ); // undefined
El array tiene longitud de 3, pero todos los grupos están vacíos.
```

Y aquí hay una coincidencia más compleja para el string `ac`:

```
let match = 'ac'.match(/a(z)?(c)?/)

alert( match.length ); // 3
alert( match[0] ); // ac (coincidencia completa)
alert( match[1] ); // undefined, ¿porque no hay nada para (z)?
alert( match[2] ); // c
```

La longitud del array es permanente: 3. Pero no hay nada para el grupo `(z)?`, por lo tanto el resultado es `["ac", undefined, "c"]`.

Buscar todas las coincidencias con grupos: `matchAll`

`matchAll` es un nuevo método, polyfill puede ser necesario

El método `matchAll` no es compatible con antiguos navegadores.

Cuando buscamos todas las coincidencias (flag `g`), el método `match` no devuelve contenido para los grupos.



Expresiones Regulares

Por ejemplo, encontremos todas las etiquetas en un string:

```
let str = '<h1> <h2>';

let tags = str.match(/<(.*)>/g);

alert( tags ); // <h1>,<h2>
```

El resultado es un array de coincidencias, pero sin detalles sobre cada uno de ellos. Pero en la práctica normalmente necesitamos contenidos de los grupos de captura en el resultado.

Para obtenerlos tenemos que buscar utilizando el método `str.matchAll(regex)`.

Fue incluido a JavaScript mucho después de `match`, como su versión "nueva y mejorada".

Al igual que `match`, busca coincidencias, pero hay 3 diferencias:

1. No devuelve un array sino un objeto iterable.
2. Cuando está presente el indicador `g`, devuelve todas las coincidencias como un array con grupos.
3. Si no hay coincidencias, no devuelve `null` sino un objeto iterable vacío.

Por ejemplo:

```
let results = '<h1> <h2>'.matchAll(/<(.*)>/gi);

// results - no es un array, sino un objeto iterable
alert(results); // [object RegExp String Iterator]

alert(results[0]); // undefined (*)

results = Array.from(results); // lo convirtamos en array

alert(results[0]); // <h1>,h1 (1er etiqueta)
alert(results[1]); // <h2>,h2 (2da etiqueta)
```

Como podemos ver, la primera diferencia es muy importante, como se demuestra en la línea (*). No podemos obtener la coincidencia como `results[0]`, porque ese objeto no es pseudo array. Lo podemos convertir en un Array real utilizando `Array.from`. Hay más detalles sobre pseudo arrays e iterables en el artículo. [Iterables](#).



Expresiones Regulares

No se necesita `Array.from` si estamos iterando sobre los resultados:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

for(let result of results) {
  alert(result);
  // primer alert: <h1>,h1
  // segundo: <h2>,h2
}
```

...O utilizando desestructurización:

```
let [tag1, tag2] = '<h1> <h2>'.matchAll(/<(.*?)>/gi);
```

Cada coincidencia devuelta por `matchAll` tiene el mismo formato que el devuelto por `match` sin el flag `g`: es un array con propiedades adicionales `index` (coincide índice en el string) e `input` (fuente string):

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);
```

```
let [tag1, tag2] = results;

alert( tag1[0] ); // <h1>
alert( tag1[1] ); // h1
alert( tag1.index ); // 0
alert( tag1.input ); // <h1> <h2>
```

¿Por qué el resultado de `matchAll` es un objeto iterable y no un array?

¿Por qué el método está diseñado de esa manera? La razón es simple – por la optimización.

El llamado a `matchAll` no realiza la búsqueda. En cambio devuelve un objeto iterable, en un principio sin los resultados. La búsqueda es realizada cada vez que iteramos sobre ella, es decir, en el bucle.

Por lo tanto, se encontrará tantos resultados como sea necesario, no más.

Por ejemplo, posiblemente hay 100 coincidencias en el texto, pero en un bucle `for...of` encontramos 5 de ellas: entonces decidimos que es suficiente y realizamos un `break`. Así el buscador no gastará tiempo buscando otras 95 coincidencias.



Grupos con nombre

Es difícil recordar a los grupos por su número. Para patrones simples, es factible, pero para los más complejos, contar los paréntesis es inconveniente. Tenemos una opción mucho mejor: poner nombres entre paréntesis.

Eso se hace poniendo `<name>` inmediatamente después del paréntesis de apertura.

Por ejemplo, busquemos una fecha en el formato "año-mes-día":

```
let dateRegex = /(?!<year>[0-9]{4})-(?!<month>[0-9]{2})-(?!<day>[0-9]{2})/;
let str = "2019-04-30";

let groups = str.match(dateRegex).groups;

alert(groups.year); // 2019
alert(groups.month); // 04
alert(groups.day); // 30
```

Como puedes ver, los grupos residen en la propiedad `.groups` de la coincidencia.

Para buscar todas las fechas, podemos agregar el flag `g`.

También vamos a necesitar `matchAll` para obtener coincidencias completas, junto con los grupos:

```
let dateRegex = /(?!<year>[0-9]{4})-(?!<month>[0-9]{2})-(?!<day>[0-9]{2})/g;

let str = "2019-10-30 2020-01-01";

let results = str.matchAll(dateRegex);

for(let result of results) {
  let {year, month, day} = result.groups;

  alert(`${day}.${month}.${year}`);
  // primer alert: 30.10.2019
  // segundo: 01.01.2020
}
```



Expresiones Regulares

Grupos de captura en reemplazo

El método `str.replace(regex, replacement)` que reemplaza todas las coincidencias con `regex` en `str` nos permite utilizar el contenido de los paréntesis en el string `replacement`. Esto se hace utilizando `$n`, donde `n` es el número de grupo.

Por ejemplo,

```
let str = "John Bull";
let regex = /(\w+) (\w+)/;

alert( str.replace(regex, '$2, $1') ); // Bull, John
```

Para los paréntesis con nombre la referencia será `$<name>`.

Por ejemplo, volvamos a darle formato a las fechas desde "year-month-day" a "day.month.year":

```
let regex = /(?(?<year>[0-9]{4})-(?(?<month>[0-9]{2})-(?(?<day>[0-9]{2}))/g;

let str = "2019-10-30, 2020-01-01";

alert( str.replace(regex, '$<day>.$<month>.$<year>') );
// 30.10.2019, 01.01.2020
```

Grupos que no capturan con ?:

A veces necesitamos paréntesis para aplicar correctamente un cuantificador, pero no queremos su contenido en los resultados.

Se puede excluir un grupo agregando `?:` al inicio.

Por ejemplo, si queremos encontrar `(go)+`, pero no queremos el contenido del paréntesis `(go)` como un ítem separado del array, podemos escribir: `(?:go)+`.



Expresiones Regulares

En el ejemplo de arriba solamente obtenemos el nombre John como un miembro separado de la coincidencia:

```
let str = "Gogogo John!";

// ?: excluye 'go' de la captura
let regexp = /(?:go)+ (\w+)/i;

let result = str.match(regexp);

alert( result[0] ); // Gogogo John (coincidencia completa)
alert( result[1] ); // John
alert( result.length ); // 2 (no hay más ítems en el array)
```

Resumen

Los paréntesis agrupan una parte de la expresión regular, de modo que el cuantificador se aplique a ella como un todo.

Los grupos de paréntesis se numeran de izquierda a derecha y, opcionalmente, se pueden nombrar con (`?<name>...`).

El contenido, emparejado por un grupo, se puede obtener en los resultados:

- El método `str.match` devuelve grupos de captura únicamente sin el indicador (flag) `g`.
- El método `str.matchAll` siempre devuelve grupos de captura.

Si el paréntesis no tiene nombre, entonces su contenido está disponible en el array de coincidencias por su número. Los paréntesis con nombre también están disponible en la propiedad `groups`.

También podemos utilizar el contenido del paréntesis en el string de reemplazo de `str.replace`: por el número `$n` o el nombre `$<name>`.

Un grupo puede ser excluido de la enumeración al agregar `?:` en el inicio. Eso se usa cuando necesitamos aplicar un cuantificador a todo el grupo, pero no lo queremos como un elemento separado en el array de resultados. Tampoco podemos hacer referencia a tales paréntesis en el string de reemplazo.



12. Referencias inversas en patrones: \N y \k<nombre>

Podemos utilizar el contenido de los grupos de captura (...) no solo en el resultado o en la cadena de reemplazo, sino también en el patrón en sí.

Referencia inversa por número: \N

Se puede hacer referencia a un grupo en el patrón usando \N, donde N es el número de grupo.

Para aclarar por qué es útil, consideremos una tarea.

Necesitamos encontrar una cadena entre comillas: con cualquiera de los dos tipos, comillas simples '...' o comillas dobles "...". Ambas variantes deben coincidir.

¿Cómo encontrarlas?

Ambos tipos de comillas se pueden poner entre corchetes: ['"](.*)['"], pero encontrará cadenas con comillas mixtas, como "... ' y '...". Eso conduciría a coincidencias incorrectas cuando una cita aparece dentro de otra, como en la cadena "She's the one!" (en este ejemplo los strings no se traducen por el uso de la comilla simple):

```
let str = `He said: "She's the one!".`;

let regexp = /['](.*)[']/g;

// El resultado no es el que nos gustaría tener
alert( str.match(regexp) ); // "She'
```

Como podemos ver, el patrón encontró una cita abierta ", luego se consume el texto hasta encontrar la siguiente comilla ', esta cierra la coincidencia.

Para asegurar que el patrón busque la comilla de cierre exactamente igual que la de apertura, se pone dentro de un grupo de captura y se hace referencia inversa al 1ero: (['])(.*)\1.

Aquí está el código correcto:

```
let str = `He said: "She's the one!".`;

let regexp = /(['])(.*)\1/g;

alert( str.match(regexp) ); // "She's the one!"
```



Expresiones Regulares

¡Ahora funciona! El motor de expresiones regulares encuentra la primera comilla (['"]) y memoriza su contenido. Este es el primer grupo de captura.

Continuando en el patrón, \1 significa "encuentra el mismo texto que en el primer grupo", en nuestro caso exactamente la misma comilla.

Similar a esto, \2 debería significar: el contenido del segundo grupo, \3 – del tercer grupo, y así sucesivamente.

Por favor tome nota:

Si usamos ?: en el grupo, entonces no lo podremos referenciar. Los grupos que se excluyen de las capturas (?:...) no son memorizados por el motor.

No confundas: el patrón \1, con el reemplazo: \$1

En el reemplazo de cadenas usamos el signo dólar: \$1, mientras que en el patrón – una barra invertida \1.

Referencia inversa por nombre: \k<nombre>

Si una regexp tiene muchos paréntesis, es conveniente asignarle nombres.

Para referenciar un grupo con nombre usamos \k<nombre>.

En el siguiente ejemplo, el grupo con comillas se llama ?<quote>, entonces la referencia inversa es \k<quote>:

```
let str = `He said: "She's the one!".`;

let regexp = /( ?<quote>['"])(.*?)\k<quote>/g;

alert( str.match(regexp) ); // "She's the one!"
```



13. Alternancia (O) |

Alternancia es un término en expresión regular que simplemente significa "O".

En una expresión regular se denota con un carácter de línea vertical |.

Por ejemplo, necesitamos encontrar lenguajes de programación: HTML, PHP, Java o JavaScript.

La expresión regular correspondiente es: `html|php|java(script)?`.

Un ejemplo de uso:

```
let regexp = /html|php|css|java(script)?/gi;

let str = "Primera aparición de HTML, luego CSS, luego JavaScript";

alert( str.match(regexp) ); // 'HTML', 'CSS', 'JavaScript'
```

Ya vimos algo similar: corchetes. Permiten elegir entre varios caracteres, por ejemplo `gr[ae]y` coincide con `gray` o `grey`.

Los corchetes solo permiten caracteres o conjuntos de caracteres. La alternancia permite cualquier expresión. Una expresión regular `A|B|C` significa una de las expresiones A, B o C.

Por ejemplo:

- `gr(a|e)y` significa exactamente lo mismo que `gr[ae]y`.
- `gra|ey` significa `gra` o `ey`.

Para aplicar la alternancia a una parte elegida del patrón, podemos encerrarla entre paréntesis:

- `I love HTML|CSS` coincide con `I love HTML` o `CSS`.
- `I love (HTML|CSS)` coincide con `I love HTML` o `I love CSS`.

Ejemplo: Expresión regular para el tiempo

En artículos anteriores había una tarea para construir una expresión regular para buscar un horario en la forma `hh:mm`, por ejemplo `12:00`. Pero esta simple expresión `\d\d:\d\d` es muy vaga. Acepta `25:99` como tiempo (ya que 99 segundos coinciden con el patrón, pero ese tiempo no es válido).



Expresiones Regulares

¿Cómo podemos hacer un mejor patrón?

Podemos utilizar una combinación más cuidadosa. Primero, las horas:

- Si el primer dígito es 0 o 1, entonces el siguiente dígito puede ser cualquiera: `[01]\d`.
- De otra manera, si el primer dígito es 2, entonces el siguiente debe ser `[0-3]`.
- (no se permite ningún otro dígito)

Podemos escribir ambas variantes en una expresión regular usando alternancia: `[01]\d|2[0-3]`.

A continuación, los minutos deben estar comprendidos entre 00 y 59. En el lenguaje de expresiones regulares se puede escribir como `[0-5]\d`: el primer dígito 0-5, y luego cualquier otro.

Si pegamos minutos y segundos juntos, obtenemos el patrón: `[01]\d|2[0-3]:[0-5]\d`.

Ya casi terminamos, pero hay un problema. La alternancia `|` ahora pasa a estar entre `[01]\d` y `2[0-3]:[0-5]\d`.

Es decir: se agregan minutos a la segunda variante de alternancia, aquí hay una imagen clara:

```
[01]\d | 2[0-3]:[0-5]\d
```

Este patrón busca `[01]\d` o `2[0-3]:[0-5]\d`.

Pero eso es incorrecto, la alternancia solo debe usarse en la parte "horas" de la expresión regular, para permitir `[01]\d` O `2[0-3]`. Corregiremos eso encerrando las "horas" entre paréntesis: `([01]\d|2[0-3]):[0-5]\d`.

La solución final sería:

```
let regexp = /([01]\d|2[0-3]):[0-5]\d/g;

alert("00:00 10:10 23:59 25:99 1:2".match(regexp)); // 00:00,10:10,23:59
```



14. Lookahead y lookbehind (revisar delante/detrás)

A veces necesitamos buscar únicamente aquellas coincidencias donde un patrón es precedido o seguido por otro patrón.

Existe una sintaxis especial para eso llamadas "lookahead" y "lookbehind" ("ver delante" y "ver detrás"), juntas son conocidas como "lookaround" ("ver alrededor").

Para empezar, busquemos el precio de la cadena siguiente `1 pavo cuesta 30€`. Eso es: un número, seguido por el signo €.

Lookahead

La sintaxis es: `x(?=y)`. Esto significa "buscar x, pero considerarlo una coincidencia solo si es seguido por y". Puede haber cualquier patrón en x y y.

Para un número entero seguido de €, la expresión regular será `\d+(?=.*€)`:

```
let str = "1 pavo cuesta 30€";  
  
alert( str.match(/\d+(?=.*€)/) ); // 30, el número 1 es ignorado porque no  
está seguido de €
```

Tenga en cuenta que "lookahead" es solamente una prueba, lo contenido en los paréntesis `(?=.*...)` no es incluido en el resultado `30`.

Cuando buscamos `x(?=y)`, el motor de expresión regular encuentra x y luego verifica si existe y inmediatamente después de él. Si no existe, entonces la coincidencia potencial es omitida y la búsqueda continúa.

Es posible realizar pruebas más complejas, por ejemplo `x(?=y)(?=z)` significa:

1. Encuentra x.
2. Verifica si y está inmediatamente después de x (omite si no es así).
3. Verifica si z está también inmediatamente después de x (omite si no es así).
4. Si ambas verificaciones se cumplen, el x es una coincidencia. De lo contrario continúa buscando.

En otras palabras, dicho patrón significa que estamos buscando por x seguido de y y z al mismo tiempo.

Eso es posible solamente si los patrones y y z no se excluyen mutuamente.



Expresiones Regulares

Por ejemplo, `\d+(?=\s)(?=. *30)` busca un `\d+` que sea seguido por un espacio (`?=\s`) y que también tenga un `30` en algún lugar después de él (`?=. *30`):

```
let str = "1 pavo cuesta 30€";  
  
alert( str.match(/\d+(?=\s)(?=. *30)/) ); // 1
```

En nuestra cadena eso coincide exactamente con el número 1.

Lookahead negativo

Digamos que queremos una cantidad, no un precio de la misma cadena. Eso es el número `\d+` NO seguido por `€`.

Para eso se puede aplicar un "lookahead negativo".

La sintaxis es: `x(?!Y)`, que significa "busca x, pero solo si no es seguido por Y".

```
let str = "2 pavos cuestan 60€";  
  
alert( str.match(/\d+\b(?!€)/g) ); // 2 (el precio es omitido)
```

Lookbehind

Compatibilidad de navegadores en lookbehind

Ten en cuenta: Lookbehind no está soportado en navegadores que no utilizan V8, como Safari, Internet Explorer.

"lookahead" permite agregar una condición para "lo que sigue".

"Lookbehind" es similar. Permite coincidir un patrón solo si hay algo anterior a él.

La sintaxis es:

- Lookbehind positivo: `(?<=Y)x`, coincide x, pero solo si hay Y antes de él.
- Lookbehind negativo: `(?<!Y)x`, coincide x, pero solo si no hay Y antes de él.



Expresiones Regulares

Por ejemplo, cambiemos el precio a dólares estadounidenses. El signo de dólar usualmente va antes del número, entonces para buscar \$30 usaremos `(?<=\$)\d+`: una cantidad precedida por \$:

```
let str = "1 pavo cuesta $30";

// el signo de dólar se ha escapado \$
alert( str.match(/(?<=\$)\d+/) ); // 30 (omite los números aislados)
```

Y si necesitamos la cantidad (un número no precedida por \$), podemos usar "lookbehind negativo" `(?<!\$)\d+`:

```
let str = "2 pavos cuestan $60";

alert( str.match(/(?<!\$)\b\d+/g) ); // 2 (el precio es omitido)
```

Atrapando grupos

Generalmente, los contenidos dentro de los paréntesis de "lookaround" (ver alrededor) no se convierten en parte del resultado.

Ejemplo en el patrón `\d+(?=€)`, el signo € no es capturado como parte de la coincidencia. Eso es esperado: buscamos un número `\d+`, mientras `(?=€)` es solo una prueba que indica que debe ser seguida por €.

Pero en algunas situaciones nosotros podríamos querer capturar también la expresión en "lookaround", o parte de ella. Eso es posible: solo hay que rodear esa parte con paréntesis adicionales.

En los ejemplos de abajo el signo de divisa (€|kr) es capturado junto con la cantidad:

```
let str = "1 pavo cuesta 30€";
let regexp = /\d+(?=(€|kr))/; // paréntesis extra alrededor de €|kr

alert( str.match(regexp) ); // 30, €
```

Lo mismo para "lookbehind":

```
let str = "1 pavo cuesta $30";
let regexp = /(?<=(\$|£))\d+/;

alert( str.match(regexp) ); // 30, $
```



Resumen

Lookahead y lookbehind (en conjunto conocidos como "lookaround") son útiles cuando queremos hacer coincidir algo dependiendo del contexto antes/después.

Para expresiones regulares simples podemos hacer lo mismo manualmente. Esto es: coincidir todo, en cualquier contexto, y luego filtrar por contexto en el bucle.

Recuerda, `str.match` (sin el indicador `g`) y `str.matchAll` (siempre) devuelven las coincidencias como un array con la propiedad `index`, así que sabemos exactamente dónde están dentro del texto y podemos comprobar su contexto.

Pero generalmente "lookaround" es más conveniente.

Tipos de "lookaround":

Patrón	Tipo	Coincidencias
<code>X(?:=Y)</code>	lookahead positivo	X si está seguido por Y
<code>X(?:!Y)</code>	lookahead negativo	X si no está seguido por Y
<code>(?<=Y)X</code>	lookbehind positivo	X si está después de Y
<code>(?<!Y)X</code>	lookbehind negativo	X si no está después de Y



15. Backtracking catastrófico

Algunas expresiones regulares parecen simples, pero pueden ejecutarse durante demasiado tiempo e incluso "colgar" el motor de JavaScript.

Tarde o temprano la mayoría de los desarrolladores se enfrentan ocasionalmente a este comportamiento. El síntoma típico: una expresión regular funciona bien a veces, pero para ciertas cadenas se "cuelga" consumiendo el 100% de la CPU.

En este caso el navegador sugiere matar el script y recargar la página. No es algo bueno, sin duda.

Para el lado del servidor de JavaScript tal regexp puede colgar el proceso del servidor, que es aún peor. Así que definitivamente deberíamos echarle un vistazo.

Ejemplo

Supongamos que tenemos una cadena y queremos comprobar si está formada por palabras `\w+` con un espacio opcional `\s?` después de cada una.

Una forma obvia de construir una regexp sería tomar una palabra seguida de un espacio opcional `\w+\s?` y luego repetirla con `*`.

Esto nos lleva a la regexp `^\w+\s?)*$` que especifica cero o más palabras de este tipo, que comienzan al principio `^` y terminan al final `$` de la línea.

En la práctica:

```
let regexp = /^(\\w+\\s?)*$/;

alert( regexp.test("A good string") ); // true
alert( regexp.test("Bad characters: $@#") ); // false
```

La regexp parece funcionar. El resultado es correcto. Aunque en ciertas cadenas tarda mucho tiempo. Tanto tiempo que el motor de JavaScript se "cuelga" con un consumo del 100% de la CPU.



Expresiones Regulares

Si ejecuta el ejemplo de abajo probablemente no se verá nada ya que JavaScript simplemente se "colgará". El navegador dejará de reaccionar a los eventos, la interfaz de usuario dejará de funcionar (la mayoría de los navegadores sólo permiten el desplazamiento). Después de algún tiempo se sugerirá recargar la página. Así que ten cuidado con esto:

```
let regexp = /^(\w+\s?)*$/;  
let str = "An input string that takes a long time or even makes this regexp  
hang!";  
  
// tardará mucho tiempo  
alert( regexp.test(str) );
```

Para ser justos observemos que algunos motores de expresión regular pueden manejar este tipo de búsqueda con eficacia, por ejemplo, la versión del motor V8 a partir de la 8.8 puede hacerlo (por lo que Google Chrome 88 no se cuelga aquí) mientras que el navegador Firefox sí se cuelga.

Ejemplo simplificado

¿Qué ocurre? ¿Por qué se cuelga la expresión regular?

Para entenderlo simplifiquemos el ejemplo: elimine los espacios `\s?`. Entonces se convierte en `^(\w+)*$`.

Y, para hacer las cosas más obvias sustituyamos `\w` por `\d`. La expresión regular resultante sigue colgando, por ejemplo:

```
let regexp = /^(\d+)*$/;  
  
let str = "012345678901234567890123456789z";  
  
// tardará mucho tiempo (¡cuidado!)  
alert( regexp.test(str) );  
¿Qué ocurre con la regexp?
```

En primer lugar uno puede notar que la regexp `(\d+)*` es un poco extraña. El cuantificador `*` parece extraño. Si queremos un número podemos utilizar `\d+`.

Efectivamente la regexp es artificial; la hemos obtenido simplificando el ejemplo anterior. Pero la razón por la que es lenta es la misma. Así que vamos a entenderlo y entonces el ejemplo anterior se hará evidente.



Expresiones Regulares

¿Qué sucede durante la búsqueda de `^(\\d+)*$` en la línea `123456789z` (acortada un poco para mayor claridad, por favor tenga en cuenta un carácter no numérico `z` al final, es importante) que tarda tanto?

Esto es lo que hace el motor regexp:

1. En primer lugar el motor regexp intenta encontrar el contenido de los paréntesis: el número `d+`. El `+` es codicioso por defecto, por lo que consume todos los dígitos:

```
\\d+.....  
(123456789)z
```

Una vez consumidos todos los dígitos se considera que se ha encontrado el `d+` (como `123456789`).

Entonces se aplica el cuantificador de asterisco `(\\d+)*`. Pero no hay más dígitos en el texto, así que el asterisco no da nada.

El siguiente carácter del patrón es el final de la cadena `$`. Pero en el texto tenemos `z` en su lugar, por lo que no hay coincidencia:

```
                X  
\\d+.....$  
(123456789)z
```

2. Como no hay ninguna coincidencia, el cuantificador codicioso `+` disminuye el recuento de repeticiones, retrocede un carácter.

Ahora `\\d+` toma todos los dígitos excepto el último (`12345678`):

```
\\d+.....  
(12345678)9z
```

3. Entonces el motor intenta continuar la búsqueda desde la siguiente posición (justo después de `12345678`).

Se puede aplicar el asterisco patrón: `(\\d+)*` : da una coincidencia más de patrón: `\\d+`, el número `9`:

```
\\d+.....\\d+  
(12345678)(9)z
```

El motor intenta coincidir con `$` de nuevo, pero falla, porque encuentra `z` en su lugar:

```
                X  
\\d+.....\\d+  
(12345678)(9)z
```



Expresiones Regulares

4. No hay coincidencia así que el motor continuará con el retroceso disminuyendo el número de repeticiones. El retroceso generalmente funciona así: el último cuantificador codicioso disminuye el número de repeticiones hasta llegar al mínimo. Entonces el cuantificador codicioso anterior disminuye, y así sucesivamente.

Se intentan todas las combinaciones posibles. Estos son sus ejemplos.

El primer número `\d+` tiene 7 dígitos y luego un número de 2 dígitos:

```
          X
\d+.....\d+
(1234567)(89)z
```

El primer número tiene 7 dígitos y luego dos números de 1 dígito cada uno:

```
          X
\d+.....\d+\d+
(1234567)(8)(9)z
```

El primer número tiene 6 dígitos y luego un número de 3 dígitos:

```
          X
\d+.....\d+
(123456)(789)z
```

El primer número tiene 6 dígitos, y luego 2 números:

```
          X
\d+.....\d+ \d+
(123456)(78)(9)z
```

...Y así sucesivamente.

Hay muchas formas de dividir una secuencia de dígitos 123456789 en números. Para ser precisos, hay $2^n - 1$, donde n es la longitud de la secuencia.

- Para 123456789 tenemos $n=9$, lo que da 511 combinaciones.
- Para una secuencia más larga con " $n=20$ " hay alrededor de un millón (1048575) de combinaciones.
- Para $n=30$ – mil veces más (1073741823 combinaciones).

Probar cada una de ellas es precisamente la razón por la que la búsqueda lleva tanto tiempo.



Expresiones Regulares

Volver a las palabras y cadenas

Lo mismo ocurre en nuestro primer ejemplo, cuando buscamos palabras por el patrón `^(\w+\s?)*$` en la cadena `An input that hangs!`.

La razón es que una palabra puede representarse como un `\w+` o muchos:

```
(input)
(inpu)(t)
(inp)(u)(t)
(in)(p)(ut)
...
```

Para un humano es obvio que puede no haber coincidencia porque la cadena termina con un signo de exclamación `!` pero la expresión regular espera un carácter denominativo `\w` o un espacio `\s` al final. Pero el motor no lo sabe.

El motor prueba todas las combinaciones de cómo la regexp `(\w+\s?)*` puede “consumir” la cadena, incluyendo las variantes con espacios `(\w+\s)*` y sin ellos `(\w+)*` (porque los espacios `\s?` son opcionales). Como hay muchas combinaciones de este tipo (lo hemos visto con dígitos), la búsqueda lleva muchísimo tiempo.

¿Qué hacer?

¿Debemos activar el lazy mode?

Desgraciadamente eso no ayudará: si sustituimos `\w+` por `\w+?` la regexp seguirá colgada. El orden de las combinaciones cambiará, pero no su número total.

Algunos motores de expresiones regulares hacen análisis complicados y automatizaciones finitas que permiten evitar pasar por todas las combinaciones o hacerlo mucho más rápido, pero la mayoría de los motores no lo hacen. Además, eso no siempre ayuda.



Expresiones Regulares

¿Cómo solucionarlo?

Hay dos enfoques principales para solucionar el problema.

El primero es reducir el número de combinaciones posibles.

Hagamos que el espacio no sea opcional reescribiendo la expresión regular como `^(\w+\s)*\w*$` buscaremos cualquier número de palabras seguidas de un espacio `(\w+\s)*`, y luego (opcionalmente) una palabra final `\w*`.

Esta regexp es equivalente a la anterior (coincide con lo mismo) y funciona bien:

```
let regexp = /^(\w+\s)*\w*$/;  
let str = "An input string that takes a long time or even makes this regex  
hang!";
```

```
alert( regexp.test(str) ); // false
```

¿Por qué ha desaparecido el problema?

Porque ahora el espacio es obligatorio.

La regexp anterior, si omitimos el espacio, se convierte en `(\w+)*`, dando lugar a muchas combinaciones de `\w+` dentro de una misma palabra

Así, `input` podría coincidir con dos repeticiones de `\w+` así:

```
\w+  \w+  
(inp)(ut)
```

El nuevo patrón es diferente: `(\w+\s)*` especifica repeticiones de palabras seguidas de un espacio. La cadena `input` no puede coincidir con dos repeticiones de `\w+\s`, porque el espacio es obligatorio.

Ahora se ahorra el tiempo necesario para probar un montón de combinaciones (en realidad la mayoría).



Previnendo el backtracking

Sin embargo no siempre es conveniente reescribir una regexp. En el ejemplo anterior era fácil, pero no siempre es obvio cómo hacerlo.

Además una regexp reescrita suele ser más compleja y eso no es bueno. Las regexps son suficientemente complejas sin necesidad de esfuerzos adicionales.

Por suerte hay un enfoque alternativo. Podemos prohibir el retroceso para el cuantificador.

La raíz del problema es que el motor de regexp intenta muchas combinaciones que son obviamente erróneas para un humano.

Por ejemplo, en la regexp `(\d+)*$` es obvio para un humano que patrón: `+` no debería retroceder. Si sustituimos un patrón: `\d+` por dos `\d+\d+` separados nada cambia:

```
\d+.....  
(123456789)!
```

```
\d+...\d+....  
(1234)(56789)!
```

Y en el ejemplo original `^(\w+\s?)*$` podemos querer prohibir el backtracking en `\w+`. Es decir: `\w+` debe coincidir con una palabra entera, con la máxima longitud posible. No es necesario reducir el número de repeticiones en `\w+` o dividirlo en dos palabras `\w+\w+` y así sucesivamente.

Los motores de expresiones regulares modernos admiten cuantificadores posesivos para ello. Los cuantificadores regulares se convierten en posesivos si añadimos `+` después de ellos. Es decir, usamos `\d++` en lugar de `\d+` para evitar que `+` retroceda.

Los cuantificadores posesivos son de hecho más simples que los "regulares". Simplemente coinciden con todos los que pueden sin ningún tipo de retroceso. El proceso de búsqueda sin retroceso es más sencillo.

También existen los llamados "grupos de captura atómicos", una forma de desactivar el retroceso dentro de los paréntesis.

...Pero la mala noticia es que, por desgracia, en JavaScript no están soportados.

Sin embargo, podemos emularlos utilizando "lookahead transform".



Expresiones Regulares

Lookahead al rescate!

Así que hemos llegado a temas realmente avanzados. Nos gustaría que un cuantificador como `+` no retrocediera porque a veces retroceder no tiene sentido.

El patrón para tomar tantas repeticiones de `\w` como sea posible sin retroceder es: `(?=(\w+))\1`. Por supuesto, podríamos tomar otro patrón en lugar de `\w`.

Puede parecer extraño, pero en realidad es una transformación muy sencilla.

Vamos a descifrarla:

- Lookahead `?=` busca la palabra más larga `\w+` a partir de la posición actual.
- El contenido de los paréntesis con `?=...` no es memorizado por el motor así que envuelva `\w+` en paréntesis. Entonces el motor memorizará su contenido
- ...y nos permitirá hacer referencia a él en el patrón como `\1`.

Es decir: miramos hacia adelante y si hay una palabra `\w+`, entonces la emparejamos como `\1`.

¿Por qué? Porque el lookahead encuentra una palabra `\w+` como un todo y la capturamos en el patrón con `\1`. Así que esencialmente implementamos un cuantificador posesivo más `+`. Captura sólo la palabra entera patrón: `\w+`, no una parte de ella.

Por ejemplo, en la palabra `JavaScript` no sólo puede coincidir con `Java` sino que deja fuera `Script` para que coincida con el resto del patrón.

He aquí la comparación de dos patrones:

```
alert( "JavaScript".match(/\w+Script/)); // JavaScript
alert( "JavaScript".match(/(?=(\w+))\1Script/)); // null
```

1. En la primera variante, `\w+` captura primero la palabra completa `JavaScript`, pero luego `+` retrocede carácter por carácter, para intentar coincidir con el resto del patrón, hasta que finalmente tiene éxito (cuando `\w+` coincide con `Java`).
2. En la segunda variante `(?=(\w+))` mira hacia adelante y encuentra la palabra `JavaScript`, que está incluida en el patrón como un todo por `\1`, por lo que no hay manera de encontrar `Script` después de ella.

Podemos poner una expresión regular más compleja en `(?=(\w+))\1` en lugar de `\w`, cuando necesitemos prohibir el retroceso para `+` después de ella.

Por favor tome nota:

Hay más (en inglés) acerca de la relación entre los cuantificadores posesivos y lookahead en los artículos [Regex: Emulate Atomic Grouping \(and Possessive Quantifiers\) with LookAhead](#) y [Mimicking Atomic Groups](#).



Expresiones Regulares

Reescribamos el primer ejemplo utilizando lookahead para evitar el backtracking:

```
let regexp = /^(?=(\w+)\2\s?)*$/;

alert( regexp.test("A good string") ); // true

let str = "An input string that takes a long time or even makes this regex hang!";

alert( regexp.test(str) ); // false, funciona, ¡y rápido!
```

Aquí se utiliza `\2` en lugar de `\1` porque hay paréntesis exteriores adicionales. Para evitar enredarnos con los números, podríamos dar a los paréntesis un nombre, por ejemplo `(?<word>\w+)`.

```
// nombramos a los parentesis ?<word>, y los referenciamos como \k<word>
let regexp = /^(?=(?<word>\w+)\k<word>\s?)*$/;

let str = "An input string that takes a long time or even makes this regex hang!";

alert( regexp.test(str) ); // false

alert( regexp.test("A correct string") ); // true
```

El problema descrito en este artículo se llama “backtracking catastrófico”.

Cubrimos dos formas de resolverlo:

- Reescribir la regexp para reducir el número de combinaciones posibles.
- Evitar el retroceso.



16. Indicador adhesivo “y”, buscando en una posición.

EL indicador `y` permite realizar la búsqueda en una posición dada en el string de origen.

Para entender el caso de uso del indicador `y` exploremos un ejemplo práctico.

Una tarea común para regexps es el “Análisis léxico”: tomar un texto (como el de un lenguaje de programación), y analizar sus elementos estructurales. Por ejemplo, HTML tiene etiquetas y atributos, el código JavaScript tiene funciones, variables, etc.

Escribir analizadores léxicos es un área especial, con sus propias herramientas y algoritmos, así que no profundizaremos en ello; pero existe una tarea común: leer algo en una posición dada.

Por ej. tenemos una cadena de código `let varName = "value"`, y necesitamos leer el nombre de su variable, que comienza en la posición 4.

Buscaremos el nombre de la variable usando regexp `\w+`. En realidad, el nombre de la variable de JavaScript necesita un regexp un poco más complejo para un emparejamiento más preciso, pero aquí eso no importa.

Una llamada a `str.match(/\w+/)` solo encontrará la primera palabra de la línea (`let`). No es la que queremos. Podríamos añadir el indicador `g`, pero al llamar a `str.match(/\w+/g)` buscará todas las palabras del texto y solo necesitamos una y en la posición 4. De nuevo, no es lo que necesitamos.

Entonces, ¿cómo buscamos exactamente en un posición determinada?

Usemos el método `regexp.exec(str)`.

Para un regexp sin los indicadores `g` y `y`, este método busca la primera coincidencia y funciona exactamente igual a `str.match(regexp)`.

...Pero si existe el indicador `g`, realiza la búsqueda en `str` empezando desde la posición almacenada en su propiedad `regexp.lastIndex`. Y si encuentra una coincidencia, establece `regexp.lastIndex` en el index inmediatamente posterior a la coincidencia.

En otras palabras, `regexp.lastIndex` funciona como punto de partida para la búsqueda, cada llamada lo reestablece a un nuevo valor: el posterior a la última coincidencia.

Entonces, llamadas sucesivas a `regexp.exec(str)` devuelve coincidencias una después de la otra.



Expresiones Regulares

Un ejemplo (con el indicador g):

```
let str = 'let varName'; // encontremos todas las palabras del string
let regexp = /\w+/g;

alert(regexp.lastIndex); // 0 (inicialmente lastIndex=0)

let word1 = regexp.exec(str);
alert(word1[0]); // let (primera palabra)
alert(regexp.lastIndex); // 3 (Posición posterior a la coincidencia)

let word2 = regexp.exec(str);
alert(word2[0]); // varName (2da palabra)
alert(regexp.lastIndex); // 11 (Posición posterior a la coincidencia)

let word3 = regexp.exec(str);
alert(word3); // null (no más coincidencias)
alert(regexp.lastIndex); // 0 (se reinicia al final de la búsqueda)
Podemos conseguir todas las coincidencias en el loop:
```

```
let str = 'let varName';
let regexp = /\w+/g;

let result;

while (result = regexp.exec(str)) {
    alert( `Found ${result[0]} at position ${result.index}` );
    // Found let at position 0, then
    // Found varName at position 4
}
```

Tal uso de `regexp.exec` es una alternativa al método `str.match` `ball`, con más control sobre el proceso.

Volvamos a nuestra tarea.

Podemos establecer manualmente `lastIndex` a 4, para comenzar la búsqueda desde la posición dada.



Expresiones Regulares

Como aquí:

```
let str = 'let varName = "value"';

let regexp = /\w+/g; // Sin el indicador "g", la propiedad lastIndex es ignorada.

regexp.lastIndex = 4;

let word = regexp.exec(str);
alert(word); // varName
¡Problema resuelto!
```

Realizamos una búsqueda de `\w+`, comenzando desde la posición `regexp.lastIndex = 4`.

El resultado es correcto.

...Pero espera, no tan rápido.

Nota que la búsqueda comienza en la posición `lastIndex` y luego sigue adelante. Si no hay ninguna palabra en la posición `lastIndex` pero la hay en algún lugar posterior, entonces será encontrada:

```
let str = 'let varName = "value"';

let regexp = /\w+/g;

// comenzando desde la posición 3
regexp.lastIndex = 3;

let word = regexp.exec(str);
// encuentra coincidencia en la posición 4
alert(word[0]); // varName
alert(word.index); // 4
```

Para algunas tareas, incluido el análisis léxico, esto está mal. Necesitamos la coincidencia en la posición exacta, y para ello es el flag `y`.

El indicador `y` hace que `regexp.exec` busque "exactamente en" la posición `lastIndex`, no "comenzando en" ella.



Expresiones Regulares

Aquí está la misma búsqueda con el indicador y:

```
let str = 'let varName = "value"';  
  
let regexp = /\w+/y;  
  
regexp.lastIndex = 3;  
alert( regexp.exec(str) ); // null (Hay un espacio en la posición 3, no una  
palabra)  
  
regexp.lastIndex = 4;  
alert( regexp.exec(str) ); // varName (Una palabra en la posición 4)  
Como podemos ver, el /\w+/y de regexp no coincide en la posición 3 (a diferencia del  
indicador g), pero coincide en la posición 4.
```

No solamente es lo que necesitamos, el uso del indicador y mejora el rendimiento.

Imagina que tenemos un texto largo, y no hay coincidencias en él. Entonces la búsqueda con el indicador g irá hasta el final del texto, y esto tomará significativamente más tiempo que la búsqueda con el indicador y.

En tareas tales como el análisis léxico, normalmente hay muchas búsquedas en una posición exacta. Usar el indicador y es la clave para un buen desempeño.



17. Métodos de RegExp y String

En este artículo vamos a abordar varios métodos que funcionan con expresiones regulares a fondo.

str.match(regex)

El método `str.match(regex)` encuentra coincidencias para las expresiones regulares (`regex`) en la cadena (`str`).

Tiene 3 modos:

1. Si la expresión regular (`regex`) no tiene la bandera `g`, retorna un array con los grupos capturados y las propiedades `index` (posición de la coincidencia), `input` (cadena de entrada, igual a `str`):

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/);

alert( result[0] );    // JavaScript (toda la coincidencia)
alert( result[1] );    // Script (primer grupo capturado)
alert( result.length ); // 2

// Additional information:
alert( result.index ); // 7 (match position)
alert( result.input ); // I love JavaScript (cadena de entrada)
```

2. Si la expresión regular (`regex`) tiene la bandera `g`, retorna un array de todas las coincidencias como cadenas, sin capturar grupos y otros detalles.

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/g);

alert( result[0] ); // JavaScript
alert( result.length ); // 1
```



Expresiones Regulares

3. Si no hay coincidencias, no importa si tiene la bandera `g` o no, retorna `null`.

Esto es algo muy importante. Si no hay coincidencias, no vamos a obtener un array vacío, pero sí un `null`. Es fácil cometer un error olvidándolo, ej.:

```
let str = "I love JavaScript";

let result = str.match(/HTML/);

alert(result); // null
alert(result.length); // Error: Cannot read property 'length' of null
```

Si queremos que el resultado sea un array, podemos escribirlo así:

```
let result = str.match(regex) || [];
```

`str.matchAll(regex)`

Una adición reciente

Esta es una adición reciente al lenguaje. Los navegadores antiguos pueden necesitar polyfills.

El método `str.matchAll(regex)` es una variante ("nueva y mejorada") de `str.match`.

Es usado principalmente para buscar por todas las coincidencias con todos los grupos.

Hay 3 diferencias con `match`:

1. Retorna un objeto iterable con las coincidencias en lugar de un array. Podemos convertirlo en un array usando el método `Array.from`.
2. Cada coincidencia es retornada como un array con los grupos capturados (el mismo formato de `str.match` sin la bandera `g`).
3. Si no hay resultados devuelve un objeto iterable vacío en lugar de `null`.



Expresiones Regulares

Ejemplo de uso:

```
let str = '<h1>Hello, world!</h1>';
let regexp = /<(.*?)>/g;

let matchAll = str.matchAll(regexp);

alert(matchAll); // [object RegExp String Iterator], no es un array, pero sí
un objeto iterable

matchAll = Array.from(matchAll); // ahora es un array

let firstMatch = matchAll[0];
alert( firstMatch[0] ); // <h1>
alert( firstMatch[1] ); // h1
alert( firstMatch.index ); // 0
alert( firstMatch.input ); // <h1>Hello, world!</h1>
```

Si usamos `for..of` para iterar todas las coincidencias de `matchAll`, no necesitamos `Array.from`.

str.split(regexp|substr, limit)

Divide la cadena usando la expresión regular (o una sub-cadena) como delimitador.

Podemos usar `split` con cadenas, así:

```
alert('12-34-56'.split('-')) // array de ['12', '34', '56']
```

O también dividir una cadena usando una expresión regular de la misma forma:

```
alert('12, 34, 56'.split(/,\s*/)) // array de ['12', '34', '56']
```

str.search(regexp)

El método `str.search(regexp)` retorna la posición de la primera coincidencia o `-1` si no encuentra nada:

```
let str = "A drop of ink may make a million think";

alert( str.search( /ink/i ) ); // 10 (posición de la primera coincidencia)
```




Expresiones Regulares

Limitación importante: `search` solamente encuentra la primera coincidencia.

Si necesitamos las posiciones de las demás coincidencias, deberíamos usar otros medios, como encontrar todos con `str.matchAll(regex)`.

str.replace(str|regexp, str|func)

Este es un método genérico para buscar y reemplazar, uno de los más útiles. La navaja suiza para buscar y reemplazar.

Podemos usarlo sin expresiones regulares, para buscar y reemplazar una sub-cadena:

```
// reemplazar guion por dos puntos
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

Sin embargo hay una trampa:

Cuando el primer argumento de `replace` es una cadena, solo reemplaza la primera coincidencia.

Puedes ver eso en el ejemplo anterior: solo el primer "-" es reemplazado por ":".

Para encontrar todos los guiones, no necesitamos usar una cadena "-" sino una expresión regular `/-/g` con la bandera `g` obligatoria:

```
// reemplazar todos los guiones por dos puntos
alert('12-34-56'.replace(/-/g, ":")) // 12:34:56
```

El segundo argumento es la cadena de reemplazo. Podemos usar caracteres especiales:

Símbolos	Acción en la cadena de reemplazo
<code>\$&</code>	inserta toda la coincidencia
<code>\$`</code>	inserta una parte de la cadena antes de la coincidencia
<code>\$'</code>	inserta una parte de la cadena después de la coincidencia
<code>\$n</code>	si <code>n</code> es un número, inserta el contenido del <code>n</code> ésimo grupo capturado, para más detalles ver Grupos de captura
<code>\$<nombre></code>	inserta el contenido de los paréntesis con el nombre dado, para más detalles ver Grupos de captura
<code>\$\$</code>	inserta el carácter <code>\$</code>



Expresiones Regulares

Por ejemplo:

```
let str = "John Smith";

// intercambiar el nombre con el apellido
alert(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John
```

Para situaciones que requieran reemplazos “inteligentes”, el segundo argumento puede ser una función.

Puede ser llamado por cada coincidencia y el valor retornado puede ser insertado como un reemplazo.

La función es llamada con los siguientes argumentos `func(match, p1, p2, ..., pn, offset, input, groups)`:

1. `match` – la coincidencia,
2. `p1, p2, ..., pn` – contenido de los grupos capturados (si hay alguno),
3. `offset` – posición de la coincidencia,
4. `input` – la cadena de entrada,
5. `groups` – un objeto con los grupos nombrados.

Si hay paréntesis en la expresión regular, entonces solo son 3 argumentos: `func(str, offset, input)`.

Por ejemplo, hacer mayúsculas todas las coincidencias:

```
let str = "html and css";

let result = str.replace(/html|css/gi, str => str.toUpperCase());

alert(result); // HTML and CSS
```

Reemplazar cada coincidencia por su posición en la cadena:

```
alert("Ho-Ho-ho".replace(/ho/gi, (match, offset) => offset)); // 0-3-6
```



Expresiones Regulares

En el ejemplo anterior hay dos paréntesis, entonces la función de reemplazo es llamada con 5 argumentos: el primero es toda la coincidencia, luego dos paréntesis, y después (no usado en el ejemplo) la posición de la coincidencia y la cadena de entrada:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (match, name, surname) =>
`${surname}, ${name}`);

alert(result); // Smith, John
```

Si hay muchos grupos, es conveniente usar parámetros rest para acceder a ellos:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (...match) => `${match[2]},
${match[1]}`);

alert(result); // Smith, John
```

O, si estamos usando grupos nombrados, entonces el objeto `groups` con ellos es siempre el último, por lo que podemos obtenerlos así:

```
let str = "John Smith";

let result = str.replace(/(?<name>\w+) (?<surname>\w+)/, (...match) => {
  let groups = match.pop();

  return `${groups.surname}, ${groups.name}`;
});

alert(result); // Smith, John
```

Usando una función nos da todo el poder del reemplazo, porque obtiene toda la información de la coincidencia, ya que tiene acceso a las variables externas y se puede hacer de todo.



str.replaceAll(str|regexp, str|func)

Este método es esencialmente el mismo que `str.replace`, con dos diferencias principales:

1. Si el primer argumento es un string, reemplaza *todas las ocurrencias* del string, mientras que `replace` solamente reemplaza la *primera ocurrencia*.
2. Si el primer argumento es una expresión regular sin la bandera `g`, habrá un error. Con la bandera `g`, funciona igual que `replace`.

El caso de uso principal para `replaceAll` es el reemplazo de todas las ocurrencias de un string.

Como esto:

```
// reemplaza todos los guiones por dos puntos
alert('12-34-56'.replaceAll("-", ":")) // 12:34:56
```

regexp.exec(str)

El método `regexp.exec(str)` retorna una coincidencia por expresión regular `regexp` en la cadena `str`. A diferencia de los métodos anteriores, se llama en una expresión regular en lugar de en una cadena.

Se comporta de manera diferente dependiendo de si la expresión regular tiene la bandera `g` o no.

Si no está la bandera `g`, entonces `regexp.exec(str)` retorna la primera coincidencia igual que `str.match(regexp)`. Este comportamiento no trae nada nuevo.

Pero si está la bandera `g`, entonces:

- Una llamada a `regexp.exec(str)` retorna la primera coincidencia y guarda la posición inmediatamente después en `regexp.lastIndex`.
- La siguiente llamada de la búsqueda comienza desde la posición de `regexp.lastIndex`, retorna la siguiente coincidencia y guarda la posición inmediatamente después en `regexp.lastIndex`.
- ...y así sucesivamente.
- Si no hay coincidencias, `regexp.exec` retorna `null` y resetea `regexp.lastIndex` a 0.

Entonces, repetidas llamadas retornan todas las coincidencias una tras otra, usando la propiedad `regexp.lastIndex` para realizar el rastreo de la posición actual de la búsqueda.



Expresiones Regulares

En el pasado, antes de que el método `str.matchAll` fuera agregado a JavaScript, se utilizaban llamadas de `regexp.exec` en el ciclo para obtener todas las coincidencias con sus grupos:

```
let str = 'More about JavaScript at https://javascript.info';
let regexp = /javascript/ig;

let result;

while (result = regexp.exec(str)) {
  alert( `Se encontró ${result[0]} en la posición ${result.index}` );
  // Se encontró JavaScript en la posición 11, luego
  // Se encontró javascript en la posición 33
}
```

Esto también funciona, aunque para navegadores modernos `str.matchAll` usualmente es lo más conveniente.

Podemos usar `regexp.exec` para buscar desde una posición dada configurando manualmente el `lastIndex`.

Por ejemplo:

```
let str = 'Hello, world!';

let regexp = /\w+/g; // sin la bandera "g", la propiedad `lastIndex` es ignorada
regexp.lastIndex = 5; // buscar desde la 5ta posición (desde la coma)

alert( regexp.exec(str) ); // world
```

Si la expresión regular tiene la bandera `y`, entonces la búsqueda se realizará exactamente en la posición del `regexp.lastIndex`, no más adelante.

Vamos a reemplazar la bandera `g` con `y` en el ejemplo anterior. No habrá coincidencias, ya que no hay palabra en la posición 5:

```
let str = 'Hello, world!';

let regexp = /\w+/y;
regexp.lastIndex = 5; // buscar exactamente en la posición 5

alert( regexp.exec(str) ); // null
```



Expresiones Regulares

Esto es conveniente cuando con una expresión regular necesitamos “leer” algo de la cadena en una posición exacta, no en otro lugar.

regexp.test(str)

El método `regexp.test(str)` busca por una coincidencia y retorna `true/false` si existe.

Por ejemplo:

```
let str = "I love JavaScript";

// estas dos pruebas hacen lo mismo
alert( /love/i.test(str) ); // true
alert( str.search(/love/i) !== -1 ); // true
```

Un ejemplo con respuesta negativa:

```
let str = "Bla-bla-bla";

alert( /love/i.test(str) ); // false
alert( str.search(/love/i) !== -1 ); // false
```

Si la expresión regular tiene la bandera `g`, el método `regexp.test` busca la propiedad `regexp.lastIndex` y la actualiza, igual que `regexp.exec`.

Entonces podemos usarlo para buscar desde un posición dada:

```
let regexp = /love/gi;

let str = "I love JavaScript";

// comienza la búsqueda desde la posición 10:
regexp.lastIndex = 10;
alert( regexp.test(str) ); // false (sin coincidencia)
```

La misma expresión regular probada (de manera global) repetidamente en diferentes lugares puede fallar

Si nosotros aplicamos la misma expresión regular (de manera global) a diferentes entradas, puede causar resultados incorrectos, porque `regexp.test` anticipa las llamadas usando la propiedad `regexp.lastIndex`, por lo que la búsqueda en otra cadena puede comenzar desde una posición distinta a cero.



Expresiones Regulares

Por ejemplo, aquí llamamos `regex.test` dos veces en el mismo texto y en la segunda vez falla:

```
let regex = /javascript/g; // (expresión regular creada:
regex.lastIndex=0)

alert( regex.test("javascript") ); // true (ahora regex.lastIndex es 10)
alert( regex.test("javascript") ); // false
```

Eso es porque `regex.lastIndex` no es cero en la segunda prueba.

Para solucionarlo, podemos establecer `regex.lastIndex = 0` antes de cada búsqueda. O en lugar de llamar a los métodos en la expresión regular usar los métodos de cadena `str.match/search/...`, ellos no usan el `lastIndex`.



