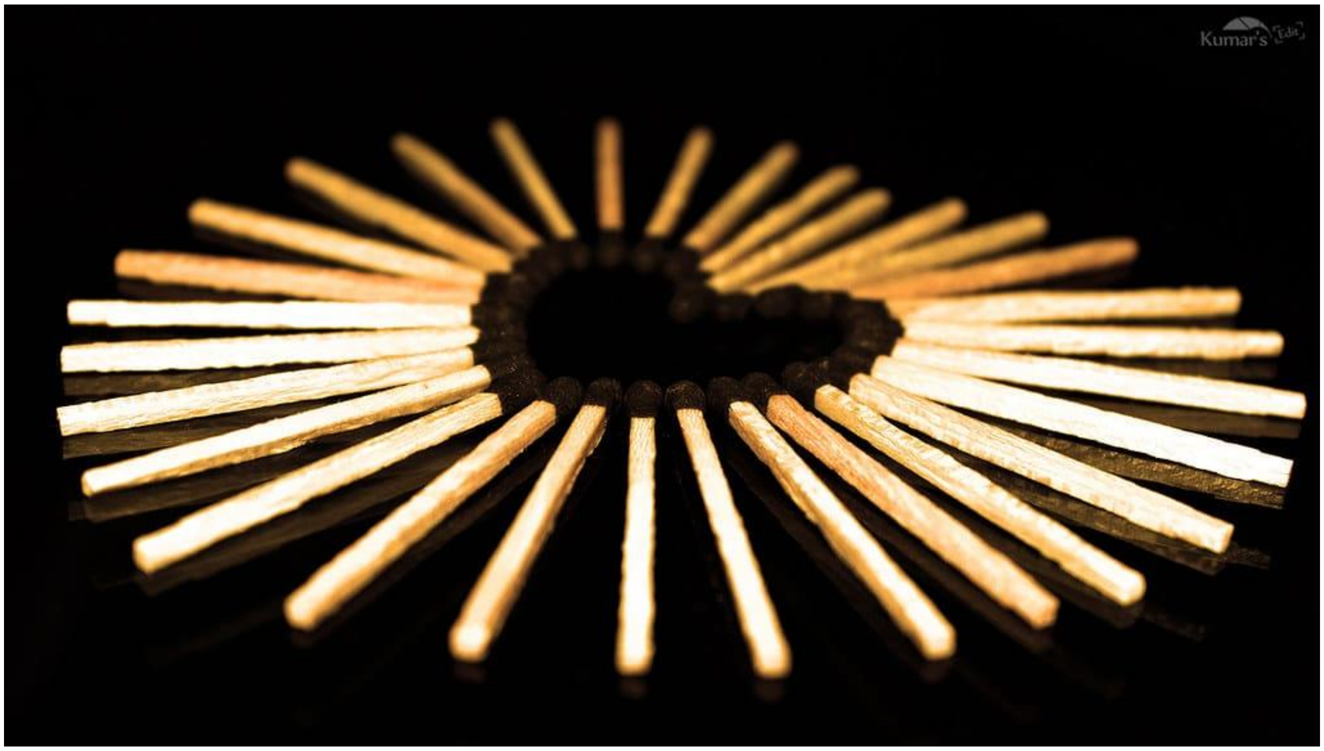
Javascript Regular Expressions — Search By Pattern

udacity.com/blog/2021/09/javascript-regular-expressions-search-by-pattern.html

September 17, 2021



Regular expressions provide the ability to "find" and "find and replace" data through text strings which specify search patterns. Regex (or regexp) are flexible, powerful, and make writing understandable and maintainable programs much easier. This blog entry will cover these search patterns and various ways to use regex within Javascript.

Javascript Regular Expression Examples

Regular expressions have many uses in any code which works with text strings (or streaming textual data). Regex find use across a wide variety of complex use cases, some examples:

- `colou?r` finds the similarly-spelled words **color** (American) and **colour** (Commonwealth)
- `[^ ]*cat[^ ,!]*` finds the words containing the root "cat" in "A **cat**, not a dog, and the toys the **cat's scattered** about — **categorically** a **catastrophe**!"

Regex are not intuitive and seemingly always require some tweaking to get right, so an online regex tester is essential. As Jamie Zawinski said on *alt.religion.emacs* in 1997:

> Some people, when confronted with a problem, think “I know, I’ll use regular expressions.” Now they have two problems.

Javascript Regular Expression Basics

Syntax

A regex has the syntax `/pattern/modifiers`. Using the string `/cat/i` as an example, regex terminology is:

- `/cat/i` is a complete regular expression
- `cat` is the pattern
- `i` is the modifier
- the forward-slashes / are the pattern delimiters

The Javascript `search()` function takes a regular expression to search for a match, and returns the position of the match.

The Javascript `replace()` function takes a regular expression and returns a modified string where the pattern is replaced.

search()

The Javascript `search()` function takes a regex and returns the position within the string where the match is found (or a -1 if not found). For example, here we search through `string` for "Cat" with the `i`modifier (ignore case during the search).

```
let regex = /Cat/i ; // case-insensitive search for "Cat"
let string = "my cat, Cat" ; // search in this string

console.log( string.search( regex ) ) ; // --> 3
```

Equivalent to the above, `search()` may be invoked directly on any string variable:

```
console.log( "my cat, Cat".search(/Cat/) ) ; // --> 10
console.log( "my cat, Cat".search(/Cat/i) ) ; // --> 3
```

NOTE: Javascript `/Cat/` is a very different beast than is `"Cat"` in this context: the former is a regex, the latter a string. It is a frequent source of debugging rage to think one is searching with the power of regex matching only to discover a quoted string. Read on for mention of the `RegExp()` function, which makes this kind of typo less common.

replace()

The Javascript `replace()` function takes a regex and returns a string, possibly modified by the pattern if the search is successful. Compare the results of the following three calls:

Search by a quoted literal for the first occurrence:

```
console.log( "my cat, Cat".replace("Cat", "Dog") ) ; // --> my cat, Dog
```

Search by a regex, insensitively, for the first occurrence:

```
console.log( "my cat, Cat".replace(/Cat/i, "Dog") ) ; // --> my Dog, Cat
```

Search by a regex, insensitively, globally (all occurrences):

```
console.log( "my cat, Cat".replace(/Cat/ig, "Dog") ) ; // --> my Dog, Dog
```

Changing Regular Expression Evaluation

The behavior of the regular expression matching engine is changed and extended through the use of modifiers, character ranges, class type meta-characters, and quantifiers.

Modifiers

Modifiers change the default matching behavior (return the first match, use case-sensitive matching, and match only the first line of a multi-line variable):

Modifier	Description
g	global matching (all, rather first only)
i	case-insensitive matching
m	multi-line matching

Ranges

Ranges, delimited by square brackets `[]`, match a range of characters:

Expression	Description
`[abc]`	Find any character between the brackets
`[^abc]`	Find any character NOT between the brackets
`[0-9]`	Find any character between the brackets (any digit)
`[^0-9]`	Find any character NOT between the brackets (any non-digit)
`(x	y)`

Meta-characters

Meta-characters match specific kinds of characters:

Meta-character	Description
`.`	a single character, except newline or line terminator
`\w`	a word character
`\W`	a non-word character
`\d`	a digit
`\D`	a non-digit character
`\s`	a whitespace character
`\S`	a non-whitespace character
`\b`	a match at a word boundary, at the beginning with `\bWORD` or at the end with `WORD\b`
`\B`	a match, but not at the beginning or end of a word
`\0`	NULL character
`\n`	new line character
`\f`	form feed character
`\r`	carriage return character
`\t`	tab character
`\v`	vertical tab character
`\xxx`	the character specified by an octal number xxx
`\xdd`	the character specified by a hexadecimal number `dd`
`\udddd`	the Unicode character specified by a hexadecimal number `dddd`

Quantifiers

Quantifiers change the specific number match sequences:

Quantifier	Description

Quantifier	Description
`n+`	Match at least one `n`
`n*`	Match zero or more occurrences of `n`
`n?`	Match zero or one occurrences of `n`
`n{X}`	Match a sequence of X `n`
`n{X,Y}`	Match a sequence of X to Y `n`
`n{X,}`	Match a sequence of at least X `n`
`n$`	Match `n` at the end
`^n`	Match`n` at the beginning
`?=n`	Match any string followed by a specific string `n`
`?!n`	Match any string not followed by a specific string `n`

Javascript RegExp object

The Javascript built-in `RegExp` object provides a more flexible mechanism for using variables for each of the matching elements. You've been watching it in action — the forward slashes automagically create regexps — so the following are equivalent:

```
let foo = new RegExp("is a") ;
let bar = /is a/ ;
```

test() and exec()

Let's begin exploring the `RegExp` object with running a literal string match on a literal string object:

```
console.log( /Cat/ig.exec( "My cat, Thor, is a Bombay." )) ;
```

To give the Javascript programmer as much computational flexibility as possible we place the string and pattern into variables and then use the `test()` and `exec()` methods to return a boolean and a results array, respectively:

```
let string = "My cat, Thor, is a Bombay cat." ;
let pattern = /Cat/ig ;

console.log( pattern.test( string )) ; // --> true
console.log( pattern.exec( string )) ; // --> ["cat"]
```

search()

The Javascript `search()` method returns the location of the first match:

```
console.log( string.search( pattern )) ; // --> 3
```

split()

The Javascript `split()` method treats the pattern as a delimiter and returns an array with all the string cut apart by the pattern.

```
let string = "My cat, Thor, is a Bombay cat." ;
let pattern = /Cat/ig ;

console.log( string.split( pattern )) ; // --> ["My ", ", Thor, is a Bombay ", "."]
```

match()

The Javascript `match()` method returns an array with all the string matches.

```
console.log( string.match( pattern )) ; // --> ["cat","cat"]
```

That's not very impressive, but consider the search results (from the example given far above) — now some of the power of `match()` becomes evident:

```
let string = "A cat, not a dog, and the toys the cat's scattered about —
categorically a catastrophe!" ;
let pattern = /[^ ]*cat[^ ,!]*/g ;

console.log( string.match( pattern )) ;
  // --> [ "cat", "cat's", "scattered", "categorically", "catastrophe" ]
```

replace()

The Javascript `replace()` method

```
let string = "A cat, not a dog, and the toys the cat's scattered about —
categorically a catastrophe!" ;
let pattern = /Cat/ig ;
let replacement = "Dog" ;

console.log( string.replace( pattern, replacement )) ;
```

The output shows that we might need more work to get satisfactory results:

> A Dog, not a dog, and the toys the Dog's sDogtered about — Dogegorically a Dogastrophe!

Regex Groups

As powerful as regex are, mention must be made of grouping, which provides a short-term memory of the matches found.

In the following code snippet, we're matching two words `\w+` that are separated by any whitespace character `\s` (which may be a space, tab, carriage return, newline, vertical tab, or form feed character).

The matches, remembered by the grouping parenthesis, are recalled in the replacement specification by the positional markers `$1` and `$2` and reversed by changing their order in the string `'$2, $1'`:

```
let regex = /(\w+)\s(\w+)/
let string = 'Firstname Lastname'

console.log( string.replace(regex, '$2, $1')) ; // --> Lastname, Firstname
```

Conclusion

This blog post covered the basic concepts of Javascript regular expressions, including:

- the syntax of modifiers, ranges, meta-characters, and quantifiers
- the Javascript RegExp object and its methods
- regular expression groups and positional variables

Start Learning

To learn more about Javascript, check out other Javascript blog posts and enroll in our Javascript Nanodegree programs, including Intermediate Javascript.