6th September 2019

Methods of RegExp and String

In this article we'll cover various methods that work with regexps in-depth.

str.match(regexp)

The method `str.match(regexp)` finds matches for `regexp` in the string `str`.

It has 3 modes:

1. If the `regexp` doesn't have flag `g`, then it returns the first match as an array with capturing groups and properties `index` (position of the match), `input` (input string, equals `str`):

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

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

alert( result[0] );     // JavaScript (full match)
alert( result[1] );     // Script (first capturing group)
alert( result.length ); // 2

// Additional information:
alert( result.index );  // 0 (match position)
alert( result.input );  // I love JavaScript (source string)
```

2. If the `regexp` has flag `g`, then it returns an array of all matches as strings, without capturing groups and other details.

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

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

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

3. If there are no matches, no matter if there's flag `g` or not, `null` is returned.

   That's an important nuance. If there are no matches, we don't get an empty array, but `null`. It's easy to make a mistake forgetting about it, e.g.:

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

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

```

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

If we want the result to be an array, we can write like this:

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

str.matchAll(regexp)

A recent addition

This is a recent addition to the language. Old browsers may need polyfills.

The method `str.matchAll(regexp)` is a “newer, improved” variant of `str.match`.

It’s used mainly to search for all matches with all groups.

There are 3 differences from `match`:

1. It returns an iterable object with matches instead of an array. We can make a regular array from it using `Array.from`.
2. Every match is returned as an array with capturing groups (the same format as `str.match` without flag `g`).
3. If there are no results, it returns not `null`, but an empty iterable object.

Usage example:

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

let matchAll = str.matchAll(regexp);

alert(matchAll); // [object RegExp String Iterator], not array, but an iterab

matchAll = Array.from(matchAll); // array now

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

If we use `for..of` to loop over `matchAll` matches, then we don’t need `Array.from`, разумеется, не нужен.

str.split(regexp|substr, limit)

Splits the string using the regexp (or a substring) as a delimiter.

We can use `split` with strings, like this:

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

But we can split by a regular expression, the same way:

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

str.search(regexp)

The method `str.search(regexp)` returns the position of the first match or `-1` if none found:

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

alert( str.search( /ink/i ) ); // 10 (first match position)
```

The important limitation: `search` only finds the first match.

If we need positions of further matches, we should use other means, such as finding them all with `str.matchAll(regexp)`.

str.replace(str|regexp, str|func)

This is a generic method for searching and replacing, one of most useful ones. The swiss army knife for searching and replacing.

We can use it without regexps, to search and replace a substring:

```
// replace a dash by a colon
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

There's a pitfall though.

When the first argument of `replace` is a string, it only replaces the first match.

You can see that in the example above: only the first `"-"` is replaced by `":"`.

To find all hyphens, we need to use not the string `"-"`, but a regexp `/-/g`, with the obligatory `g` flag:

```
// replace all dashes by a colon
alert( '12-34-56'.replace( /-/g, ":" ) )  // 12:34:56
```

The second argument is a replacement string. We can use special character in it:

| Symbols | Action in the replacement string |

Symbols	Action in the replacement string
`$&`	inserts the whole match
`` $` ``	inserts a part of the string before the match
`$'`	inserts a part of the string after the match
`$n`	if `n` is a 1-2 digit number, inserts the contents of n-th capturing group, for details see Capturing groups
`$<name>`	inserts the contents of the parentheses with the given `name`, for details see Capturing groups
`$$`	inserts character `$`

For instance:

```
let str = "John Smith";

// swap first and last name
alert(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John
```

For situations that require "smart" replacements, the second argument can be a function.

It will be called for each match, and the returned value will be inserted as a replacement.

The function is called with arguments `func(match, p1, p2, ..., pn, offset, input, groups)`:

1. `match` – the match,
2. `p1, p2, ..., pn` – contents of capturing groups (if there are any),
3. `offset` – position of the match,
4. `input` – the source string,
5. `groups` – an object with named groups.

If there are no parentheses in the regexp, then there are only 3 arguments: `func(str, offset, input)`.

For example, let's uppercase all matches:

```
let str = "html and css";

let result = str.replace(/html|css/gi, str => str.toUpperCase());

alert(result); // HTML and CSS
```

Replace each match by its position in the string:

```
alert("Ho-Ho-ho".replace(/ho/gi, (match, offset) => offset)); // 0-3-6
```

In the example below there are two parentheses, so the replacement function is called with 5 arguments: the first is the full match, then 2 parentheses, and after it (not used in the example) the match position and the source string:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (match, name, surname) => `${surname}

alert(result); // Smith, John
```

If there are many groups, it’s convenient to use rest parameters to access them:

Если в регулярном выражении много скобочных групп, то бывает удобно использовать остаточные аргументы для обращения к ним:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (...match) => `${match[2]}, ${match[1

alert(result); // Smith, John
```

Or, if we’re using named groups, then `groups` object with them is always the last, so we can obtain it like this:

```
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
```

Using a function gives us the ultimate replacement power, because it gets all the information about the match, has access to outer variables and can do everything.

regexp.exec(str)

The method `regexp.exec(str)` method returns a match for `regexp` in the string `str`. Unlike previous methods, it’s called on a regexp, not on a string.

It behaves differently depending on whether the regexp has flag `g`.

If there’s no `g`, then `regexp.exec(str)` returns the first match exactly as `str.match(regexp)`. This behavior doesn’t bring anything new.

But if there’s flag `g`, then:

- A call to `regexp.exec(str)` returns the first match and saves the position immediately after it in the property `regexp.lastIndex`.
- The next such call starts the search from position `regexp.lastIndex`, returns the next match and saves the position after it in `regexp.lastIndex`.
- …And so on.

- If there are no matches, `regexp.exec` returns `null` and resets `regexp.lastIndex` to `0`.

So, repeated calls return all matches one after another, using property `regexp.lastIndex` to keep track of the current search position.

In the past, before the method `str.matchAll` was added to JavaScript, calls of `regexp.exec` were used in the loop to get all matches with groups:

```
let str = 'More about JavaScript at https://javascript.info';
let regexp = /javascript/ig;

let result;

while (result = regexp.exec(str)) {
  alert( `Found ${result[0]} at position ${result.index}` );
  // Found JavaScript at position 11, then
  // Found javascript at position 33
}
```

This works now as well, although for newer browsers `str.matchAll` is usually more convenient.

We can use `regexp.exec` to search from a given position by manually setting `lastIndex`.

For instance:

```
let str = 'Hello, world!';

let regexp = /\w+/g; // without flag "g", lastIndex property is ignored
regexp.lastIndex = 5; // search from 5th position (from the comma)

alert( regexp.exec(str) ); // world
```

If the regexp has flag `y`, then the search will be performed exactly at the position `regexp.lastIndex`, not any further.

Let's replace flag `g` with `y` in the example above. There will be no matches, as there's no word at position `5`:

```
let str = 'Hello, world!';

let regexp = /\w+/y;
regexp.lastIndex = 5; // search exactly at position 5

alert( regexp.exec(str) ); // null
```

That's convenient for situations when we need to "read" something from the string by a regexp at the exact position, not somewhere further.

regexp.test(str)

The method `regexp.test(str)` looks for a match and returns `true/false` whether it exists.

For instance:

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

// these two tests do the same
alert( /love/i.test(str) ); // true
alert( str.search(/love/i) != -1 ); // true
```

An example with the negative answer:

```
let str = "Bla-bla-bla";

alert( /love/i.test(str) ); // false
alert( str.search(/love/i) != -1 ); // false
```

If the regexp has flag `g`, then `regexp.test` looks from `regexp.lastIndex` property and updates this property, just like `regexp.exec`.

So we can use it to search from a given position:

```
let regexp = /love/gi;

let str = "I love JavaScript";

// start the search from position 10:
regexp.lastIndex = 10;
alert( regexp.test(str) ); // false (no match)
```

⚠ Same global regexp tested repeatedly on different sources may fail

If we apply the same global regexp to different inputs, it may lead to wrong result, because `regexp.test` call advances `regexp.lastIndex` property, so the search in another string may start from non-zero position.

For instance, here we call `regexp.test` twice on the same text, and the second time fails:

```
let regexp = /javascript/g;  // (regexp just created: regexp.lastIndex=0)

alert( regexp.test("javascript") ); // true (regexp.lastIndex=10 now)
alert( regexp.test("javascript") ); // false
```

That's exactly because `regexp.lastIndex` is non-zero in the second test.

To work around that, we can set `regexp.lastIndex = 0` before each search. Or instead of calling methods on regexp, use string methods `str.match/search/...`, they don't use `lastIndex`.

‹ Previous lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)