ORIGAMID

REGULAR EXPRESSION

Regexp Métodos

Regexp Constructor

Toda regexp é criada com o constructor `RegExp()` e herda as suas propriedades e métodos. Existem diferenças na sintaxe de uma Regexp criada diretamente em uma variável e de uma passada como argumento de `RegExp`.

```
const regexp = /\w+/gi;

// Se passarmos uma string, não precisamos dos //
// e devemos utilizar \\ para meta characters, pois é
necessário
// escapar a \ especial. As Flags são o segundo argumento
const regexpObj1 = new RegExp('\\w+', 'gi');
const regexpObj2 = new RegExp(/\w+/, 'gi');

'JavaScript Linguagem 101'.replace(regexpObj1, 'X');
// X X X

// Exemplo complexo:
const regexpTELEFONE1 = /(?:\+?55\s?)?(?:\(?\d{2}\)?[-\s]?)?
\d{4,5}[-\s]?\d{4}/g;
const regexpTELEFONE2 = new RegExp('(?:\\+?55\\s?)?(?:\\(?
```

Propriedades

Uma regexp possui propriedades com informações sobre as flags e o conteúdo da mesma.

```
const regexp = /\w+/gi;

regexp.flags; // 'gi'
regexp.global; // true
regexp.ignoreCase; // true
regexp.multiline; // false
regexp.source; // '\w+'
```

regexp.test()

O método `test()` verifica se existe ou não uma ocorrência da busca. Se existir ele retorna true. A próxima vez que chamarmos o mesmo, ele irá começar do index em que parou no último true.

```
const regexp = /Java/g;
const frase = 'JavaScript e Java';

regexp.lastIndex; // 0
regexp.test(frase); // true
regexp.lastIndex; // 4
regexp.test(frase); // true
regexp.lastIndex; // 17
regexp.test(frase); // false
regexp.lastIndex; // 0
regexp.test(frase); // true (Reinicia
regexp.lastIndex;  // 4
```

test() em loop

Podemos utilizar o while loop, para mostrar enquanto a condição for verdadeira. Assim retornamos a quantidade de match's.

```
const regexp = /Script/g;
const frase = 'JavaScript, TypeScript e CoffeeScript';

let i = 1;
while(regexp.test(frase)) {
  console.log(i++, regexp.lastIndex);
}
// 1 10
// 2 22
// 3 37
```

regexp.exec()

O `exec()` diferente do `test()`, irá retornar uma Array com mais informações do que apenas um valor booleano.

```
const regexp = /\w{2,}/g;
const frase = 'JavaScript, TypeScript e CoffeeScript';

regexp.exec(frase);
// ["JavaScript", index: 0, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
regexp.exec(frase);
// ["TypeScript", index: 12, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
regexp.exec(frase);
// ["CoffeeScript", index: 25, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
regexp.exec(frase);
// null
regexp.exec(frase); // Reinicia
// ["JavaScript", index: 0, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
```

Loop com exec()

Podemos fazer um loop com exec e parar o mesmo no momento que encontre o null.

```
const regexp = /\w{2,}/g;
const frase = 'JavaScript, TypeScript e CoffeeScript';
let regexpResult;

while((regexpResult = regexp.exec(frase)) !== null) {
  console.log(regexpResult[0]);
}
```

str.match()

O `match()` é um método de strings que pode receber como argumento uma Regexp. Existe uma diferença de resultado quando utilizamos a flag `g` ou não.

```
const regexp = /\w{2,}/g;
const regexpSemG = /\w{2,}/;
const frase = 'JavaScript, TypeScript e CoffeeScript';

frase.match(regexp);
// ['JavaScript', 'TypeScript', 'CoffeeScript']

frase.match(regexpSemG);
// ["JavaScript", index: 0, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
```

> *Se não tiver match retorna null*

str.split()

O `split` serve para distribuirmos uma string em uma array, quebrando a string no argumento que for passado. Este método irá remover o match da array final.

```
const frase = 'JavaScript, TypeScript, CoffeeScript';

frase.split(', ');
// ["JavaScript", "TypeScript", "CoffeeScript"]
frase.split(/Script/g);
// ["Java", ", Type", ", Coffee", ""]

const tags = `
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
`;

tags.split(/(?<=<\/?)\w+/g).join('div');
// <div>
//   <div>Item 1</div>
```

```
// <div>
```

str.replace()

O método `replace()` é o mais interessante por permitir a utilização de funções de callback para cada match que ele der com a Regexp.

```
const tags = `
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
`;

tags.replace(/(?<=<\/?)\w+/g, 'div');
// <div>
//   <div>Item 1</div>
//   <div>Item 2</div>
// <div>
```

Captura

É possível fazer uma referência ao grupo de captura dentro do argumento do replace. Então podemos utilizar `$&`, `$1` e mais.

```
const tags = `
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
`;

tags.replace(/<li/g, '$& class="ativo"');
// <ul>
//   <li class="ativo">Item 1</li>
//   <li class="ativo">Item 2</li>
// </ul>
```

Grupos de Captura

É possível definirmos quantos grupos de captura quisermos.

```
const emails = `
empresa@email.com
contato@email.com
suporte@email.com
`;

emails.replace(/(\w+@)[\w.]+/g, '$1gmail.com');
// empresa@gmail.com
// contato@gmail.com
// suporte@gmail.com
```

Callback

Para substituições mais complexas, podemos utilizar um callback como segundo argumento do replace. O valor do return será o que irá substituir cada match.

```
const regexp = /(\w+)(@[\w]+)/g;
const emails = `joao@homail.com.br
marta@ggmail.com.br
bruna@oulook.com.br`

emails.replace(regexp, function(...args) {
  console.log(args);
  if(args[2] === '@homail') {
    return `${args[1]}@hotmail`;
  } else if(args[2] === '@ggmail') {
    return `${args[1]}@gmail`;
  } else if(args[2] === '@oulook') {
    return `${args[1]}@outlook`;
  } else {
    return 'x';
  }
});
```

```
// marta@gmail.com.br
// bruna@outlook.com.br
```

> Código apenas para demonstração