Expressões Regulares em JavaScript: O Guia Definitivo para Iniciantes

Neste artigo eu vou resumir da forma mais simples as principais coisas que você precisa saber sobre RegExp, isso vai te servir tanto para JavaScript como qualquer outra linguagem de programação.

Primeiramente, o que são expressões regulares? **Uma expressão regular, RegExp, ou Regex, é uma forma especial de trabalhar com strings na programação. Expressões regulares podem ser usadas para encontrar ou substituir informações de uma string. Quase todas as linguagens de programação implementam expressões regulares.**

Se você, assim como eu, já olhou pra uma expressão regular e não entendeu absolutamente nada, eu recomendo fortemente que você invista tempo e acompanhe até o final, porque vale a pena!

Por que aprender Regex?

Depois de anos trabalhando, por muito tempo eu fui ignorante sobre esse assunto e acabei por usar só o que havia disponível na internet. De certa forma você fica refém do que é capaz de encontrar nos fóruns.

Mas quando você aprende, você finalmente para de perder tempo pesquisando expressões prontas, e mais tempo ainda tentando resolver problemas de forma menos eficiente, sem regex.

Na verdade, saber expressões regulares te coloca em outro nível como programador. Eu diria que uma minoria dos programadores têm pelo menos um conhecimento intermediário de expressões regulares, pois não é uma coisa tão fácil de aprender. Mas quando você aprende de fato, mesmo que o básico, você tem uma ferramenta extremamente útil nas mãos.

Introdução à Expressões Regulares

Como já definimos, expressões regulares são uma forma de trabalhar com strings para encontrar, substituir ou extrair certas informações.

Você provavelmente já viu em algum lugar o uso de caracteres curinga (ex *.jpg, *.png, http://site.com/*), para definir arquivos, extensões, ou URLs. A forma mais fácil de entender expressões regulares é pensando nesses casos, porém você pode fazer operações muito mais complexas, como por exemplo extrair um e-mail de um texto ou validar um número de telefone.

Antes de começarmos com exemplos, **saiba que uma expressão regular não é algo fácil de ler entender**, também não são fáceis de escrever, por isso não se intimide.

Na verdade também não é algo fácil de memorizar uma vez que você estuda, (você vai precisar fazer exercícios) vou tentar ajudar o máximo possível, mas no final **usar uma expressão regular muitas vezes é a única maneira sensata de resolver um problema de programação**.

Inicializando uma expressão regular

Em JavaScript, uma expressão regular é um objeto da classe RegExp. Podemos definir uma expressão regular de duas formas:

Usando a forma literal (mais comum):

```
const reg = /teste/;
```

JavaScript

Copiar

Ou inicializando o objeto RegExp:

```
const reg = new RegExp('teste');
```

JavaScript

Copiar

A forma literal é mais comum provavelmente porque ela é mais simples. Assim como temos duas formas de inicializar arrays e objetos `[]`, `new Array()`, `{}`, `new Object()`, a forma literal é sempre a mais usada. Mas vai ter casos em que vamos precisar inicializar o objeto RegExp, veremos mais adiante.

No exemplo acima, **teste** é chamado de pattern (padrão), e é o que vai definir nossa expressão regular para trabalharmos com outra string. Vamos agora entender como funciona esse padrão.

Métodos que utilizam expressões regulares

Precisamos falar rapidamente sobre alguns métodos que utilizam expressões regulares. Os métodos são todos derivados do objeto **String** ou do próprio objeto **RegExp**.

RegExp	exec	Faz uma pesquisa pela string e retorna null ou um array contendo a posição do padrão na string. Ex: /teste/.exec(string)
	test	Faz uma pesquisa pela string e retorna true ou false caso não tenha encontrado o padrão. Ex: /teste/.test(string)
String	match	Faz uma pesquisa na string pela correspondência do padrão. Retorna null ou array. Ex: 'teste'.match(/teste/)
	replace	Faz uma pesquisa na string pela correspondência do padrão e substitui por outra string. Ex: '-teste-'.replace(/teste/, '')
	search	Faz uma pesquisa na string pela correspondência do padrão e retorna a posição ou -1 caso nada seja encontrado. Ex: 'teste'.search(/teste/)
	split	Faz uma pesquisa na string pela correspondência do padrão e retorna um array com a divisão da string em substrings. Ex: '-teste-'.split(/teste/)

Neste artigo, nós vamos explicar sobre como criar padrões de RegExp, sem ir muito a fundo em cada método.

Importante para entender

Regex é uma coisa que se aprende melhor com prática, então para entender como funcionam os padrões que vou descrever aqui, recomendo que você abra o console do seu navegador e vá testando os comandos descritos abaixo. Use Ctrl + Shift + J (Windows) ou Cmd + Option + J (Mac).

Você pode ir alterando os padrões para ver se é retornado true ou false. Assim você vai testando comigo aos poucos e pode pular pra próxima parte se não restarem dúvidas.

Verificando se uma string contém uma substring

Como você viu a respeito dos métodos, é possível dividir, substituir ou localizar padrões dentro de uma string usando expressões regulares. Neste artigo vamos usar mais o método `test` para explicar como criar esses padrões.

Digamos que você quer verificar se uma string contém a palavra "teste".

Ao definirmos `/palavra/`, criamos um objeto RegExp com o padrão "palavra" usando a forma literal. Então com o método `test` podemos verificar se o padrão existe em uma string, retornando true ou false:

```
/palavra/.test('palavras'); // ✅
/palavra/.test('Esta frase contém uma palavra'); // ✅
/palavra/.test('Esta frase não contém'); // ❌
```

JavaScript

Copiar

Substring no inicio ou final da string

Digamos que você precisa verificar se uma palavra existe no início ou no final de uma string.

Operador ^ (inicio)

Para validar se uma string começa com um determinado padrão, utilize o acento circunflexo `^`:

```
/^oi/.test('oi, tudo certo?'); // ✅
/^oi/.test('...oi, tudo certo?'); // ❌
```

JavaScript

Copiar

Operador $ (fim)

Para validar se uma string termina com um determinado padrão, utilize o operador `$`:

```
/fim$/.test('aqui termina com fim'); // ✅
/fim$/.test('aqui não termina com fim...'); // ❌
```

JavaScript

Copiar

Combinando inicio e fim de uma string

Para verificar se uma string começa e termina com o mesmo padrão, você pode usar `^` e `$` na mesma expressão:

```
/^oi$/.test('oi'); // ✅
/^oi$/.test('oi, tudo certo?'); // ❌
/^oi$/.test('oi, tudo certo? oi'); // ❌
```

JavaScript

Copiar

Note que no último exemplo, apesar de começar e terminar com "oi", nosso padrão reconhece somente o que começa e somente o que termina, portanto vai validar apenas a string "oi".

A utilidade destes operadores se dá principalmente quando nós temos alguma coisa no meio e sabemos como a string precisa iniciar ou terminar.

Flags

Podemos determinar como uma expressão regular deve ser interpretada através de flags. Você pode usar mais de uma flag na mesma expressão.

Por enquanto não vamos nos preocupar muito com isso, mas veja abaixo pra que serve cada uma:

- `g` Determina que a busca deve retornar todos os padrões encontrados. Caso não seja usado, será retornado apenas o primeiro padrão encontrado.
- `i` Determina que a busca não deve diferenciar letras maiúsculas de letras minúsculas. Caso não não seja usado, a busca vai retornar verdadeiro apenas se a string for exata.
- `m` Afeta apenas o comportamento dos operadores `^` e `$`. Caso não seja usado, vai retornar verdadeiro caso a string inteira corresponda ao padrão, mas se for usado, vai retornar verdadeiro caso uma linha corresponda ao padrão.
- `s` Permite com que o caractere `.` corresponda também à quebra da linha.
- `u` Permite o suporte à caracteres unicode. É possível fazer buscas por emojis, por exemplo. ?

A utilização de flags se dá na inicialização do objeto da expressão regular, das mesmas formas ensinadas no início do artigo. Da forma literal:

```javascript
/teste/i.test('Teste'); // ✅
```

JavaScript

Copiar

ou inicializando o objeto RegExp com a flag no segundo parâmetro:

```javascript
new RegExp('teste', 'i').test('Teste'); // ✅
```

JavaScript

Copiar

Conjuntos

Esse é um dos tópicos mais importantes e é usado na maioria das expressões regulares. Você pode usar colchetes para criar conjuntos de padrões. Esses conjuntos podem ser uma escala de "0" a "9", ou de "a" a "z", etc.

Podemos usar conjuntos para verificar se uma string contém números de 0-9. Digamos que eu queira extrair um número (idade) de uma string:

```javascript
/[0-9]+/.exec('Idade: 22')[0]; // 22
```

JavaScript

Copiar

Veja mais exemplos de conjuntos:

```javascript
/[1-5]/ // 1, 2, 3, 4, 5
/[0-9]/ // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
/ab/ // a, b
/[a-d]/ // a, b, c, d
/[a-z]/ // a, b, c, d ... z
/A-Z/ // A, B, C ... Z
```

JavaScript

Copiar

Podemos também combinar conjuntos:

```
/[0-9a-zA-Z]/.test('1'); // ✅
/[0-9a-zA-Z]/.test('A'); // ✅
/[0-9a-zA-Z#]/.test('#'); // ✅
/[0-9a-zA-Z]/.test('#'); // ❌
/[0-9a-zA-Z]/.test('Á'); // ❌
```

JavaScript

Copiar

Negando conjuntos

O operador `^`, como explicado anteriormente, define o início de um padrão. Porém, quando utilizado dentro de conjuntos, tem a função de negar aquele conjunto. Exemplo:

```
// Testa se a string contém algo que não seja um número
/[^0-9]/.test('Teste'); // ✅
/[^0-9]/.test('012345teste'); // ✅
/[^0-9]/.test('012345'); // ❌

// Testa se a string contém algo que não seja um número e também não seja de a-z
/[^0-9a-z]/.test('012345teste!'); // ✅
/[^0-9a-z]/.test('012345teste'); // ❌
```

JavaScript

Copiar

Meta caracteres

Meta caracteres são outro tópico bastante usado em expressões regulares. Eles servem para abreviar certos conjuntos e também para especificar algumas correspondências especiais. O uso de barra invertida na maioria deles serve para "escapar" o caractere, para que o caractere não seja considerado na correspondência do padrão. Vejamos:

- `.` Corresponde à qualquer caractere, exceto quebra de linha. Para corresponder especificamente à um ponto (.), é necessário escapar este caractere com `\.`.
- `\d` Corresponde à um caractere numérico. É o mesmo que `[0-9]`.
- `\D` Corresponde à um caractere não numérico. É o mesmo que `[^0-9]`.
- `\w` Corresponde à qualquer caractere alfanumérico mais _. Equivalente à `[0-9a-zA-Z_]`.
- `\W` Corresponde à qualquer caractere não alfanumérico mais _. Equivalente à `[^0-9a-zA-Z_]`.
- `\s` Corresponde à um caractere de espaço, tab e quebra de linha.
- `\S` Corresponde à qualquer caractere que não seja espaço, tab ou quebra de linha.
- `\n` Corresponde à quebra de linha.
- `\t` Corresponde à tab.
- `\0` Corresponde à null.
- `\p{x}` Corresponde à um caractere unicode cuja propriedade passada em "x" seja verdadeira. Requer que a flag "u" seja utilizada.
- `\P{x}` Corresponde ao oposto de `\p{x}`.
- `[^]` Corresponde à qualquer caractere, incluindo quebra de linha (diferente do .).

Alguns mnemônicos para facilitar na memorização:

- Como você pôde perceber, quando o meta caractere está em maiúsculo ele sempre corresponde ao contrário do caractere em minúsculo.

- "." pode ser lembrado como reticências (...) para corresponder à qualquer coisa.
- "d" pode ser lembrado como abreviação de digit - somente números.
- "w" pode ser lembrado como abreviação de word - é usado com frequência juntamente com quantificadores para corresponder à palavras. Lembre-se que também pode conter números.
- "s" pode ser lembrado como abreviação de space.
- O "\p", talvez como abreviação de property, é uma inclusão recente do ES2018, vamos ver mais abaixo como usar isso.

Quantificadores

Se você quer testar se uma string correspnde à um padrão 0 ou N vezes, vai precisar usar quantificadores. Veja quais são todos os quantificadores:

`A?` **Corresponde a zero ou um "A":** Não faz muito sentido usar com o método test, mas você vai ver um exemplo abaixo onde ele pode fazer mais sentido. O `?` pode ser considerado como um operador para especificar algo opcional.

```
/A?/.test(''); // ✅
/A?/.test('A'); // ✅
/A?/.test('AAA'); // ✅
/A?/.test('B'); // ✅
```

JavaScript

Copiar

`A*` **Corresponde a zero ou mais "A":**

```
/A*/.test(''); // ✅
/A*/.test('A'); // ✅
/A*/.test('AAA'); // ✅
```

```
/A*/.test('B'); // ✓
```

JavaScript

Copiar

`A+` Corresponde a pelo menos um "A":

```
/A+/.test(''); // ✗
/A+/.test('A'); // ✓
/A+/.test('AAA'); // ✓
/A+/.test('B'); // ✗
```

JavaScript

Copiar

`A{x}` Corresponde a exatamente x vezes "A" - Note que a sequência também deve corresponder:

```
/A{3}/.test('AA'); // ✗
/A{3}/.test('AAA'); // ✓
/A{3}/.test('AAAAA'); // ✓
/A{3}/.test('B'); // ✗
/A{3}/.test('ABAA'); // ✗
```

JavaScript

Copiar

`A{x,y}` Corresponde a exatamente de x a y vezes "A":

```
/A{2,3}/.test('A'); // ✗
/A{2,3}/.test('AA'); // ✓
/A{2,3}/.test('AAA'); // ✓
/A{2,3}/.test('AAAAA'); // ✓
/A{2,3}/.test('ABA'); // ✗
```

JavaScript

Copiar

`A{x,}` Corresponde a x até o infinito vezes "A":

```
/A{3,}/.test('A'); // ✕
/A{3,}/.test('AA'); // ✕
/A{3,}/.test('AAA'); // ✓
/A{3,}/.test('AAAAA'); // ✓
```

JavaScript

Copiar

Exemplos com quantificadores

Agora que você tem uma noção geral de grupos e quantificadores, vamos tentar combiná-los. Digamos que você quer validar um endereço de IP (v4):

```
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/.test('127.0.0.1');
// ✓
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/.test('127.0.0');
// ✕
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/.test('127.0.0.1000
'); // ✕
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/.test('localhost');
// ✕
```

JavaScript

Copiar

Note que usamos barra invertida no ponto, porque `.` é um caractere especial, por isso precisamos "escapá-lo".

Para verificar se um número de telefone contém o 9 na frente ou não, ignorando outros caracteres para simplificar:

```
/^9+[0-9]{8}$/.test(999998888); // ✓
/^9+[0-9]{8}$/.test(899998888); // ✕
/^9+[0-9]{8}$/.test(99998888); // ✕
```

JavaScript

Copiar

Para encontrar uma tag HTML span com seu conteúdo:

```javascript
/<span>.*<\/span>/.test('<span>Teste</span>');
```

JavaScript

Copiar

Note que a barra deve ser "escapada" com o uso da barra invertida para que a expressão não termine. Veja a seguir como funcionam esses escapes.

"Escapando" caracteres especiais

Estes são os caracteres que precisam ser "escapados", ou ignorados:

- `\` A barra serve para escapar outros caracteres. Use `\\` se quiser corresponder à "\"
- `/` Começa e termina uma expressão regular.
- `[ ]` Define um set ou conjunto.
- `{ }` Define uma propriedade.
- `( )` Define um grupo.
- `?+*` Quantificadores.
- `|` Operador "OR".
- `.` Curinga.
- `^` Define início de um padrão e também serve para negar um grupo.
- `$` Define o final de um padrão.

Não é necessário "escapar" caracteres especiais em conjuntos

Normalmente quando queremos escapar um caractere especial, usamos `\`. Porém isso não é necessário dentro de conjuntos. Por exemplo, para verificar se um conjunto tem números de 0-9, ponto ou vírgula, você pode usar `[0-`

`9.,]` sem problemas. Para corresponder ponto e vírgula fora do conjunto você teria que fazer `[0-9]\.,`.

Grupos

Usando parênteses, você pode criar grupos de padrões. Digamos que você queira validar uma string que possa ter dois formatos "Nº 1234" ou "1234":

```
/^(Nº\s)?[0-9]{4}$/.test('Nº 1234'); // ✅
/^(Nº\s)?[0-9]{4}$/.test('1234'); // ✅
/^(Nº\s)?[0-9]{4}$/.test('No 1234'); // ❌
```

JavaScript

Copiar

Agrupando a string "Nº " e adicionando o quantificador "?" (opcional), o grupo todo se torna opcional.

Quando você quiser adicionar um caractere opcional apenas, não é necessário criar grupos. Por exemplo, para verificar "pessoa" ou "pessoas".

```
/pessoas?/.test('pessoa'); // ✅
/pessoas?/.test('pessoas'); // ✅
```

JavaScript

Copiar

Outra vantagem de usar grupos é poder simplificar padrões repetidos. Por exemplo:

```
/(ha){2,}/.test('hahaha'); // ✅
```

JavaScript

Copiar

A expressão regular acima captura uma risada com mais de dois "ha". Se eu quisesse fazer essa expressão sem grupos, seria impossível testar "ha" infinitas vezes.

Retirando o grupo, o quantificador funciona para o caractere anterior:

```
/ha{2,}/.test('hahaha'); // ✕
/ha{2,}/.test('haaaaa'); // ✓
```

JavaScript

Copiar

Capturando grupos

Uma das vantagens de se usar grupos é o que podemos capturar o seu conteúdo.

Até agora usamos o método `test` para validar grupos e outras expressões, mas se você usar o método `exec` ou `String.match`, é possível capturar o conteúdo de um grupo.

Digamos que você deseja ler uma string contendo cidade e estado, e deseja separar a cidade e estado em duas correspondências:

```
/(.+) - ([\w]{2})/.exec('São Paulo - SP'); // [ "São Paulo - SP", "São Paulo", "SP" ]
```

JavaScript

Copiar

Primeiro delimitamos qualquer caractere (.) pelo menos uma vez (+), depois especificamos os caracteres " - " como delimitador, e em seguida usamos \w vezes 2 para corresponder as 2 letras da unidade federativa.

Note que é retornado um array contendo o primeiro índice como a string completa, o segundo sendo o primeiro grupo e assim por diante.

Referência de Grupos + substituições

Grupos também são muito úteis em substituições, quando precisamos que um grupo seja referenciado. Podemos chamar os grupos definidos com `$1`, `$2`, `$3`...

Vamos supor que você precisa capturar um valor numérico em uma string e apresentá-lo em valor monetário, você pode fazer isso de forma muito simples com regex:

```
'O valor é de 1000'.replace(/(\d+)/, 'R$ $1'); // "O valor é de R$ 1000"
```

JavaScript

Copiar

Ignorando grupos

Ao definirmos um grupo, ele é automaticamente considerado e pode ser referenciado com `$1`, `$2`, etc. Mas se você quiser ignorar um grupo, é possível usando a sintaxe `?:` dentro do inicio do grupo `(?: ... )`, por exemplo:

```
// Sem ignorar
/(https?):\/\/(.*)/.exec('https://metring.com.br'); // [ "https://metring.com.br", "https", "metring.com.br" ]
// Ignorando o protocolo
/(?:https?):\/\/(.*)/.exec('https://metring.com.br'); // [ "https://metring.com.br", "metring.com.br" ]
```

JavaScript

Copiar

Grupos com nomes

Grupos também podem ter nomes para facilitar a manipulação da string. Essa é uma função recente, foi implementada na versão ES2018 do JavaScript.

Digamos que você quer saber o dia, mês e ano de uma data, utilize a seguinte sintaxe para nomear os dados em grupos diferentes:

```
const data =
/(?<dia>\d{2})\/(?<mes>\d{2})\/(?<ano>\d{4})/.exec('03/05
/2019');
data.groups.dia; // "03"
data.groups.mes; // "05"
data.groups.ano; // "2019"
```

JavaScript

Copiar

Referência de grupos dentro do padrão

Como você viu na referência de grupos em substituições, é possível referenciar o primeiro grupo usando `$1`, e assim por diante.

Também é possível fazer referência do primeiro grupo anteriormente especificado dentro da própria expressão regular usando `\1`. Isso pode ser um pouco mais complicado de entender, por isso é necessário um exemplo.

Pense no seguinte problema: Imagine que você precisa testar uma string que contém aspas simples ou aspas duplas, retornando o seu conteúdo:

```
/['"](.*)['"]/.exec('<img src="test.jpg">'); // [
""test.jpg"", "test.jpg" ]
```

JavaScript

Copiar

Tudo bem até aqui, o primeiro grupo retornou apenas a string dentro de aspas duplas. Mas e se dentro de aspas duplas eu também tenho uma aspas simples? Ex: **<img title="Ricardo's site">**. O regex acima não vai funcionar.

Ao invés disso, precisamos encapsular as aspas dentro de um grupo e usar como referência no fechamento das aspas, assim a expressão não vai fechar com a aspas simples no meio da string. Ex:

```
/(['"])(.*?)\1/.exec(`<img title="Ricardo's site">`); //
[ ""Ricardo's site"", """, "Ricardo's site" ]
```

JavaScript

Copiar

Note que usamos `\1` para fazer referência ao primeiro grupo `['"]`, assim não importa se foi usado aspas simples ou dupla para o atributo title, a expressão só vai fechar quando bater com o caractere do mesmo grupo que abriu, no caso `"`.

Também é possível passar nomes de grupos como referência usando `\k<nome>`. Exemplo:

```
/(?<aspas>['"])(.*?)\k<aspas>/.exec(`Empresa: "Ricardo's Pizzaria"`); // [ ""Ricardo's Pizzaria"", """, "Ricardo's Pizzaria" ]
```

JavaScript

Copiar

Aninhamento de grupos

Grupos podem ser aninhados assim como funções, usando parênteses dentro de parênteses. A ordem dos grupos é definida da esquerda para a direita conforme a abertura do parênteses. Digamos que você queira ler uma string contendo uma tag `<a>` HTML, e retornar o atributo href, o link contido em href e o conteúdo da tag:

```
'<a href="https://metring.com.br">Metring</a>'.match(/<a (href="(.*)")>(.*)<\/a>/);
// [ "<a href="https://metring.com.br">Metring</a>", "href="https://metring.com.br"", "https://metring.com.br", "Metring" ]
```

JavaScript

Copiar

Operador | (OR)

Se você quiser escolher entre uma correspondência ou outra, utilize o operador `|`. Ex:

```
/whiskey|vodka/.test('whiskey'); // ✅
/whiskey|vodka/.test('vodka'); // ✅
/whiskey|vodka/.test('pinga'); // ❌
```

JavaScript

Copiar

O operador OR também funciona dentro de grupos e entre grupos:

```
/(whiskey|vodka)|(refrigerante|suco)/.test('whiskey'); // ✅
/(whiskey|vodka)|(refrigerante|suco)/.test('refrigerante'); // ✅
/(whiskey|vodka)|(refrigerante|suco)/.test('pinga'); // ❌
```

JavaScript

Copiar

Lookahead

Literalmente significa "olhe para frente" e serve para testar se uma string é seguida de um padrão. É definida por `?=`. Exemplo:

Se após a letra A vier a letra B:

```
/A(?=B)/.test('AB'); // ✅
/A(?=B)/.test('AZ'); // ❌
/A(?=B)/.test('BA'); // ❌
```

JavaScript

Copiar

É possível também negar a expressão acima utilizando `?!`. Exemplo:

```
/A(?!B)/.test('AB'); // ✕
/A(?!B)/.test('BA'); // ✓
/A(?!B)/.test('ABA'); // ✓ Contém e ao mesmo tempo não contém B na frente de A
```

JavaScript

Copiar

Lookbehind

Lookbehind significa "olhe para trás". Funciona da mesma forma que o lookahead, só que ao inverso. Utilize `?<=`:

```
/(?<=A)B/.test('ABC'); // ✓
/(?<=A)B/.test('BA'); // ✕
```

JavaScript

Copiar

Para negar o lookbehind, utilize `?<!`:

```
/(?<!A)B/.test('ABC'); // ✕
/(?<!A)B/.test('BA'); // ✓
```

JavaScript

Copiar

Propriedades interessantes unicode

Algumas propriedades adicionadas recentemente na versão ES2018 do JavaScript permitem que você faça testes sobre propriedades dos caracteres unicode da sua string.

Você pode fazer isso por meio do meta caractere \p ou \P (negado).

É importante lembrar que como se trata de um meta caractere de propriedade unicode, a flag "u" precisa ser usada na inicialização da nossa expressão regular.

Para testar se uma string unicode contém um ou mais caracteres em maiúsculo:

```
/\p{Uppercase}+/u.test('MAIÚSCULO'); // ✅
/\p{Uppercase}+/u.test('Maiúsculo'); // ✅
/\p{Uppercase}+/u.test('minúsculo'); // ❌
```

JavaScript

Copiar

E a forma negada das expressões acima usando `\P`. Note que não vai retornar false para "Maiúsculo", pois a string ao mesmo tempo que contém caracteres em maiúsculo, contém caracteres em minúsculo.

```
/\P{Uppercase}+/u.test('MAIÚSCULO'); // ❌
/\P{Uppercase}+/u.test('Maiúsculo); // ✅
/\P{Uppercase}+/u.test('minúsculo'); // ✅
```

JavaScript

Copiar

Outros exemplos usando `\p`:

```
/^\p{Lowercase}$/u.test('a'); // ✅
/^\p{Lowercase}$/u.test('A'); // ❌
/^\p{Emoji}$/u.test('?') // ✅
/^\p{Emoji}$/u.test('A') // ❌
/^\p{Script=Arabic}+$/u.test('صفـيحة'); // ✅
/^\p{Script=Hebrew}+$/u.test('ואיה'); // ✅
/^\p{Script=Latin}+$/u.test('FestaNoAP'); // ✅
```