XP Inc.

Regex: Um guia prático para expressões regulares

Alexandre Servian Jan 31, 2020 · 8 min read

Aprenda regex de forma simples e fácil!

Photo by Fran . on Unsplash

Em toda nossa vida como programadores estaremos lidando com **regex** em algum momento. Seja para coisas comuns como **validar** emails ou data e até mesmo para fazer **parse** em textos em busca de um determinado padrão.

Nesse post, irei mostrar de forma simples e fácil como você pode criar suas próprias expressões regulares. Todos os exemplos serão feitos usando **javascript**, porém, muito das regex criadas aqui também funcionarão em outras linguagens como **elixir**, **python**, **php** ou **ruby**.

O que é uma expressão regular?

De forma simples:

> Uma expressão regular é um método formal de se especificar um padrão de texto.

Com ela podemos lidar com as seguintes situações:

- procura;
- substituição;
- validação de formatos;
- filtragem de informações

Criando nossas primeiras regex

Usando javascript podemos optar por duas formas de se criar uma regex. Podemos criar usando um **constructor**:

```
const regex = new RegExp('dog','gi');
```

Ou criando de forma **literal**:

```
const regex = /dog/gi
```

Podemos buscar por somente uma letra:

```
pattern: /a/
string:  A casa está limpa.
matches:    ^
```

Ou buscar por uma palavra:

```
pattern: /celular/
string:  O celular está tocando.
matches:   ^^^^^^^
```

Podemos ver que, na primeira regex, ela ignorou o primeiro **A**. Isso acontece porque as regex são **case sensitive**. Notamos também na primeira regex que só houve **match** na primeira ocorrência. Para lidar com isso, usaremos **flags**.

Flags

Elas adicionam comportamentos adicionais a nossas regras, como:

- `g` - indicar achar todas as ocorrências da regex
- `i` - ignora case sensitive
- `m` - multilinha, lida com caracteres de inicio e fim `(^ e $)` ao operar em múltiplas linhas.

Podemos então obter todas as **ocorrências** com a flag `g` e em conjunto com a flag `i` podemos ignorar o **case sensitive** de uma busca:

```
pattern: /ontem/gi
string: Ontem faltou água, Anteontem faltou luz
matches:^^^^^                  ^^^^^
```

Operador pipe " | “

Algumas vezes precisamos dar match em mais de **um** termo, para isso usamos o operador pipe `|` . Ele funciona basicamente como nosso operador lógico **OR** `||` . Assim podemos escrever:

```
pattern: /ver|distrai/gi
string: Ver a linha do horizonte me distrai
matches:^^^                          ^^^^^^^

pattern: /20|nove/gi
string: Perdi 20 em 20 e nove amizades
matches:      ^^    ^^   ^^^^

pattern: /e|m|²/gi
string:  E = mc²
matches: ^   ^ ^
```

Conjuntos “ [ ] ”

Com os conjuntos dizemos a regex que uma determinada casa pode ter diversos valores para dar match. Vejamos o seu uso:

```
pattern: /[em²]/gi
string:  E = mc²
matches: ^   ^ ^
```

Uma funcionalidade interessante é de adicionar **range** (invervalos) dentro dos nossos conjuntos. Podemos determinar um conjunto de match em letras que vão de **A** à **Z** ou pegue qualquer digito (**0** à **9**).

```
pattern: /[a-z]/
string:  João de Santo Cristo
matches:  ^ ^ ^^  ^^^^  ^^^^^

pattern: /[A-Za-z]/
string:  Açucar e Café
matches: ^ ^^^^ ^  ^^

pattern: /[0-9.,]/
string:  Um ps4 custa R$ 1.600,00
matches:      ^          ^^^^^^^^
```

Um detalhe a ser observado é que o **range** obedece a mesma ordem da tabela **Unicode**, sendo assim regex como `[a-Z]` ou `[4-1]` produziram **erro**, pois ambas não estão na ordem correta da tabela Unicode.

Podemos também dar match em letras com **acentos** (é-à) ou (ç) usando:

```
pattern: /[À-ü]/
string:  Açaí é melhor
matches: ^^^^ ^ ^^^^^^
```

Temos também os **conjuntos negados**, que como o nome sugere, dar match em tudo que não faça parte do conjunto. Para definí-lo iniciamos a regra do conjunto com `^` , por exemplo `[^a-z]` que aceita tudo que não seja entre **a** à **z**. Vejamos um exemplo:

```
pattern: /[^aeiouí]/gi
string:  Paralelepípedo
matches: ^ ^ ^ ^ ^ ^ ^
```

Metacaracteres

Nas regex existem duas formas de caracteres, os **literais**, que representam o valor literal do caractere como `abc123` e os **metacaracteres** que possuem funções distintas dentro de um contexto na regex. Dois exemplos que acabamos de ver são o uso do `^` iniciando um conjunto negado e o uso do `-` em uma regra de conjunto com range `[1-9]` . Um metacaractere bastante recorrente é o **ponto** `.` , ele funciona como um **coringa**, sendo capaz de dar match em qualquer caractere, vejamos um exemplo:

```
pattern: /cas./gi
string:  Casa, caso, case
matches: ^^^^  ^^^^  ^^^^
```

Visto isso, você deve estar se perguntando: Como pegar a forma **literal** do ponto? Eis que é muito simples bastando usar um **escape** `\` mais o metacaractere desejado. Vejamos um exemplo:

```
pattern: /[a\-o]/gi
string:  cachorro-quente.
matches:  ^  ^  ^^
```

Para simplificar a escrita e leitura das regex, possuímos algumas **shorthands** que são extremamente úteis para deixar ainda mais claro nosso código. Veja como podemos escrever esse conjunto `[0-9]` para `\d` , `[a-zA-Z0-9_]` para `\w` ou para tratar espaços em branco `[\r\n\t\f\v ]` para `\s` simplificando ainda mais nossas regras. Vejamos alguns exemplos:

```
pattern: /\(\d\d\)\s\d\d\d\d\d-\d\d\d\d/
string:  (86) 95262-7297
matches: ^^^^^^^^^^^^^^^

pattern: /\w\w\w\w\w@\w\w\.com/
string:  ax_a5@5x.com
matches: ^^^^^^^^^^^
```

Para um guia de consulta, criei um **gist** contendo muitos **metacaracteres** e **shorthands** e seus respectivos significado.

Quantificadores

Uma maneira de deixar suas regras ainda mais simples é com o uso dos quantificadores. Com eles podemos dizer quantas vezes uma **mesma** regra pode aparecer em **sequência**. Vejamos elas:

- `?` - zero ou um ocorrência;
- `*` - zero ou mais ocorrências;
- `+` - uma ou mais ocorrências;
- `{n, m}` - de n até m.

> *Seu uso é simples, basta adicionar o quantificador após um caractere, metacaractere, conjunto ou mesmo um grupo (ainda veremos abaixo). Exemplo* `[0-9]? \w* a+` *e* `(\d){1,3}` .

Digamos que queremos pegar um documento como o **cpf**, que contêm muitos números e pontuações(`.` e `-`) onde a validação pode aceitar o cpf com e sem pontuação, ficando deste modo:

```
pattern: /\d{3}\.?\d{3}\.?\d{3}-?\d{2}/
string:  825.531.760-07
matches: ^^^^^^^^^^^^^^
string:  18646661024
matches: ^^^^^^^^^^^
```

Como podemos pegar uma repetição de caractere sem estipular algum limite, vejamos:

```
pattern: /go+l+/gi
string:  Goolllll da Alemanha!!!
matches: ^^^^^^^^
```

Âncoras

Muitas vezes vamos precisar **delimitar** a ação da nossa regex. Desse modo podemos usar três metas para nos auxiliar nessa função.

Quando queremos tratar uma **palavra** que em suas extremidades não possua outra letra ou palavra, usamos a shorthands `\b`.

```
pattern: /\bpar\b/gi
string:  Parcela par Parcial paraíso
matches:         ^^^

pattern: /\bpar[a-z]+/gi
string:  Parei parque topar
matches: ^^^^^ ^^^^^^

pattern: /[a-z]+par\b/gi
string:  parodiado escapar equipar parasitar
matches:           ^^^^^^^ ^^^^^^^

pattern: /\b[a-z]+par[a-z]+\b/gi
string:  limpar aparto aparta
matches:        ^^^^^^ ^^^^^^
```

> *Vale notar que caracteres com acentos ou `-` são considerados **bordas**.*

Podemos lidar com o **início** e **fim** de uma linha. Usamos a meta `^` para indicar o **início** de uma linha e `$` indicando o **fim** de uma linha. Algo importante a se notar é que para as âncoras funcionarem a cada quebra de linha `\n` a flag `m` tem que estar **habilitada**. Segue uma estrofe usando a meta `^` :

```
pattern: /^[a-z]*\b/gmi
Quantas chances desperdicei
^^^^^^^
Quando o que eu mais queria
^^^^^^
Era provar pra todo o mundo
^^^
Que eu não precisava provar nada pra ninguém
^^^
```

Confira também o uso do meta *`$` em uma estrofe:*

```
pattern: /[a-z]+nto$/gmi
O tempo cobre o chão de verde manto
                               ^^^^^
Que já coberto foi de neve fria,
E em mim converte em choro o doce canto
                                   ^^^^^
```

Conseguimos tratar início e final de um texto ao mesmo tempo. Confira um exemplo:

```
pattern: /^https:\/\/w{3}\.[a-z]+\.com$/gmi
https://google.com.br
https://www.facebook.com
^^^^^^^^^^^^^^^^^^^^^^^^
https://www.voxel.com.br
```

Grupos “ () ”

Por fim, temos os grupos que facilita ainda mais nossas regras. Eles nos possibilita a criação de regras isoladas, possibilita a criação de referencias (retrovisores) para o

reuso da mesma regra em outro local dentro de uma mesmo regex e ainda cria a possibilidade de **validações** dentro da regex. Seu uso é muito **diverso**, dando muito poder ao programador na hora de escrever suas regras. Veja um exemplo:

```
pattern: /(\d{2})\/?(\d{2})?\/(\d{4})/
string:  Hoje é dia 20/01/2020
matches:           ^^^^^^^^^^
```

Uma função muito interessante dos grupos é que quando criamos algum grupo, este grupo é criando uma **referência**, que podemos acessa-lo em funções como o método replace (que vamos ver a frente) ou usar como **retrovisores** (mirror words) para fazer reuso de algum grupo que deu match anteriormente. Vejamos um exemplo baseado no exemplo anterior:

```
pattern: /\d{2}(\/?)\d{2}?\1\d{4}/g
string:  20/01/2020 25091991 25-09/2000
matches: ^^^^^^^^^^ ^^^^^^^^
```

No exemplo acima, veja que criamos o grupo `(\/?)` e para não repetí-lo em outro momento que necessitamos da mesma regra, usamos o retrovisor `\1` sendo `1` é ligado a **ordem** em que esse grupo foi criado. Podemos criar diversas referências para o reuso de regras.

> *Uma dica é se por exemplo usamos um grupo `(\w)` o seu retrovisor será o caractere que deu match com `\w`. Ex: \w = R seu \1 sera R.*

Podemos definir grupos que podem ser **ignorados** (non-capturing groups) na hora do match usando a sintaxe `(?:)`. Vejamos um exemplo:

```
pattern: /([a-z]*)\s(?:ronaldo)/gi
string:  Cristiano Ronaldo
matches: ^^^^^^^^^^^^^^^^
```

No exemplo acima, só foi **nomeado** um grupo, no caso `([a-z]*)` *pois o grupo* `(?:ronaldo)` *foi definido usando* `(?:)` e com isso não conseguimos manipulá-lo.

Com os grupos podemos criar grupos **aninhados** (grupos dentro de grupos). Vejamos um exemplo:

```
pattern: /((d[io])|(co))([a-z]{2})(do)/gi
string:  ditado colado dosado
matches: ^^^^^^ ^^^^^^ ^^^^^^
```

Os grupos possuem **grupos especiais**. Como o **positive lookahead** `(?=)` e o seu oposto, **negative lookahead** `(?!)`. Com o positive lookahead podemos **verificar** se **existe** um grupo a frente de uma expressão ou grupo. Vejamos um exemplo:

```
pattern: /([a-z]+)(?=,)/gi
string:  Penso, logo existo
matches: ^^^^^
```

Falamos acima que a regex só dá match em palavras que à sua **frente** possuam virgula. Já o negative lookahead é exatamente o contrário do positive lookahed, ele pegará todos que não fazem parte do grupo especial. Vejamos um exemplo:

```
pattern: /([a-z]+)(?!,)\b/gi
string:  Penso, logo existo
matches:        ^^^^ ^^^^^^
```

Dentro dos grupos especiais ainda temos os **positive lookbehind** e **negative lookbehind**, porém como eles não possuem um bom suporte nos browsers decidi deixá-lo de fora deste post, porém pretendo abordá-los em post futuros.

Métodos de regex no js

O objeto **regex** possui dois métodos: `exec` e `test`. Já com string possui 4 métodos: `match`, `replace`, `search` e `split`. Porém neste post vou me ater somente a 3 métodos: test, match e replace.

test

Usado para verificar se uma regex da match com uma string. Ela retorna sempre valor **boolean**. Este método é ideal para fazer **validações** como por exemplo validar se um email, telefone ou data estão corretos. Vejamos um exemplo validando números de telefone:

match

Ele retorna um array, com as string que deram match com a regex. Se não houver valor, ele retorna **null**. Vejamos um exemplo procurando cep validos em um texto:

Vejamos acima que o último número não foi pego no match, pois ele não é um cep válido.

replace

Usado para **substituir** strings que deram match por uma nova string. Segue um exemplo:

Podemos ainda **manipular** grupos. Vejamos um exemplo:

Um recurso legal do replace é que podemos passar uma **função** em vez da string de substituição. Isso ainda nos dá mais possibilidades de alteração em algum texto que queremos substituir algo. Vejamos um exemplo:

Conclusão

Chegamos ao fim, o estudo de regex é muito interessante, sendo que possível fazer muitas coisas em diferentes linguagens. Fiz uma extensa lista de diversos problemas resolvidos com regex, confira abaixo:

- **18 aplicações comuns de regex no dia a dia**

Você ainda pode testar suas regex de forma mais visual pelos sites:

- **Regex101**
- **Regulex**

Este é meu primeiro post, espero que tenha ajudado. Vlws.

Referências:

- A guide to JavaScript Regular Expressions
- 20 Small Steps to Become a Regex Master

- Regex tutorial — A quick cheatsheet by examples

JavaScript Regex Regexp Tutorial Regular Expressions