Expressions régulières complexes

Récemment, en étudiant l'analyse HTML, je suis tombé sur une expression régulière :

```
/([\w-:\*>]*)(?:\#([\w-]+)|\.([\w-]+))?(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?([\/, ]+)/is
```

Ce code est une expression régulière (regex) utilisée pour analyser des chaînes de caractères. Voici une explication en français de ce qu'elle fait :

1. `([\w-:\*>]*)` : Capture une séquence de caractères alphanumériques (`\w`), tirets (`-`), deux-points (`:`), astérisques (`*`), ou chevrons (`>`). Cette séquence peut être vide.
2. `(?:\#([\w-]+)|\.([\w-]+))?` : Capture optionnellement soit un identifiant (commençant par `#`) soit une classe (commençant par `.`). Les caractères capturés peuvent être alphanumériques ou des tirets.
3. `(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?` : Capture optionnellement un attribut entre crochets. L'attribut peut commencer par un `@` ou un `!`, suivi de caractères alphanumériques, tirets ou deux-points. Il peut également contenir un opérateur de comparaison (`=`, `!=`, `*=`, `^=`, `$=`) et une valeur entre guillemets simples ou doubles.
4. `([\/, ]+)` : Capture une séquence de caractères composée de barres obliques (`/`), virgules (`,`), ou espaces (). Cette séquence doit contenir au moins un caractère.
5. Le modificateur `is` à la fin signifie que l'expression régulière est insensible à la casse (`i`) et que le point (`.`) correspond également aux sauts de ligne (`s`).

Cette expression régulière est souvent utilisée pour analyser des sélecteurs CSS ou des fragments de code HTML.

Il est utilisé pour correspondre à des sélecteurs CSS, comme par exemple `div > ul`.

J'ai souvent rencontré des expressions aussi complexes dans le passé, et j'ai toujours instinctivement reculé. Aujourd'hui, je vais enfin les comprendre à fond ! Un homme doit savoir se montrer dur envers lui-même !

Correspondance de `div > ul`

La sélection `div > ul` en CSS ou dans un sélecteur jQuery cible tous les éléments `<ul>` qui sont des enfants directs d'un élément `<div>`. Cela signifie que seuls les éléments `<ul>` qui sont immédiatement imbriqués dans un `<div>` seront sélectionnés, et non ceux qui sont imbriqués plus profondément dans d'autres éléments.

Exemple en HTML :

```
<div>
  <ul> <!-- Ce ul sera sélectionné -->
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
  <div>
    <ul> <!-- Ce ul ne sera PAS sélectionné -->
      <li>Item 3</li>
      <li>Item 4</li>
    </ul>
  </div>
</div>
```

Exemple en CSS :

```
div > ul {
  background-color: yellow;
}
```

Exemple en jQuery :

```
$('div > ul').css('background-color', 'yellow');
```

Dans cet exemple, seul le premier `<ul>` sera mis en surbrillance en jaune, car il est un enfant direct du `<div>`. Le deuxième `<ul>` ne sera pas affecté car il est imbriqué dans un autre `<div>`.

J'ai trouvé un site web, https://regex101.com/, qui permet de faire des correspondances en ligne et fournit également des explications.

Bien que les explications à droite aient clarifié certaines choses, il reste encore des incertitudes sur la manière exacte dont les correspondances fonctionnent. Alors, prenons quelques exemples et analysons-les un par un.

Le code spécifique où cette expression régulière apparaît est :

```
$matches = [];
preg_match_all($this->pattern, trim($selector).' ', $matches, PREG_SET_ORDER);
```

`preg_match_all` signifie récupérer toutes les chaînes qui correspondent au motif. Par exemple, si vous avez :

```
preg_match_all("abc", "abcdabc", $matches)
```

Note : Le code reste en anglais car il s'agit d'une syntaxe PHP spécifique qui ne doit pas être traduite.

Le premier paramètre est le motif, le deuxième paramètre est la chaîne à comparer, et le troisième paramètre est la référence du résultat. Après l'exécution, le tableau `$matches` contiendra deux occurrences de `abc`.

Avec cette compréhension, dans l'image ci-dessus, `div > ul` ne correspond qu'aux quatre premiers caractères `div >`. `regex101` ne supporte pas `preg_match_all` ? Heureusement, il suffit d'ajouter un modificateur appelé `g` :

En ajoutant `g`, cela correspondra à tous les éléments, plutôt que de retourner uniquement le premier trouvé.

Après l'avoir ajouté, nous avons trouvé une correspondance pour `div > ul` :

À droite, on voit que dans le premier match, c'est-à-dire `div`, nous avons utilisé les règles du premier groupe pour matcher `div`, puis les règles du septième groupe pour matcher l'espace .

Passons maintenant à l'explication du premier ensemble de règles :

Dans cette longue expression, la première partie entre parenthèses est appelée le premier groupe de règles. Il s'agit d'un groupe de capture. Les parenthèses elles-mêmes ne correspondent à rien, mais servent à regrouper des éléments. `[]` représente un ensemble de caractères, et les règles à l'intérieur définissent la nature de cet ensemble de caractères. Cet ensemble de caractères contient :

- `\w` représente les lettres majuscules et minuscules, les chiffres de 0 à 9 ainsi que le tiret bas.
- `-:` représente directement ces deux caractères dans l'ensemble.
- `\*` : comme `*` est un caractère réservé dans les expressions régulières avec une signification spéciale, il faut utiliser `\` pour l'échapper, indiquant qu'il s'agit d'un caractère `*` ordinaire.
- `>` représente simplement le caractère >.

`[\w-:\*>]*` Le dernier `*` signifie que le caractère précédent peut apparaître 0 fois ou un nombre illimité de fois, mais il essaiera de correspondre autant de fois que possible. La raison pour

laquelle il correspond à `div` est que `\w` correspond à `d`, `i`, `v`. La raison pour laquelle il ne continue pas à correspondre à l'espace qui suit est que l'espace n'est pas présent dans `[]`. Un groupe de capture signifie que cette correspondance apparaîtra dans le tableau des résultats. En revanche, il existe également des groupes non capturants, dont la syntaxe est `(?:)`. Si vous n'avez pas besoin du résultat du groupe dans `([\w-:\*>]*)`, vous pouvez l'écrire comme `(?:[\w-:\*>]*)`.

Alors, si cela n'apparaît pas dans le résultat, ne pas utiliser de parenthèses ne suffirait-il pas ? Les parenthèses servent à regrouper, et le regroupement a tout son sens. Vous pouvez vous référer à □What is a non capturing group? (?:) - StackOverflow□.

Après avoir expliqué pourquoi `div` satisfait la première série de règles, parlons maintenant de pourquoi l'espace satisfait les règles du septième groupe.

`[\/, ]` signifie qu'il correspond à l'un de ces quatre caractères, et `+` indique que la correspondance précédente apparaît une ou plusieurs fois, autant que possible. Par conséquent, puisque ces quatre caractères incluent un espace, il correspond à notre espace. De plus, comme le caractère suivant après `div` est `>`, il ne satisfait plus la règle du septième groupe et ne continue pas à correspondre.

J'ai compris la correspondance du `div`. Mais pourquoi les règles des groupes deux à six n'ont pas capturé les espaces ici, les laissant plutôt au septième groupe ?

Explication de la deuxième partie :

D'abord, `(?:)` indique qu'il s'agit d'un groupe non capturant. Le `?` à la fin signifie que la correspondance précédente peut apparaître 0 ou 1 fois. Donc, dans l'expression `(?:\#([\w-]+)|\.([\w-]+))?`, cette partie peut être présente ou non. Si on enlève les modificateurs externes, il reste `\#([\w-]+)|\.([\w-]+)`, où le `|` au milieu signifie "ou", c'est-à-dire que l'une ou l'autre des deux parties peut être satisfaite. Dans `\#([\w-]+)`, le `\#` correspond au caractère `#`, et `[\w-]+` correspond à d'autres caractères. Ensuite, dans la seconde partie, `\.([\w-]+)`, le `\.` correspond au caractère `.`.

Ainsi, les groupes 2 à 6 peuvent ne pas être satisfaits car l'espace n'est pas un caractère de début requis pour ces groupes. De plus, comme ces groupes ont un modificateur `?`, il est acceptable qu'ils ne soient pas satisfaits, ce qui permet de passer directement au septième groupe.

Ensuite, le `>` qui suit `div > ul` reste le même :

Le premier ensemble de règles `([\w-:\*>]*)` correspond à `>`, et le septième ensemble de règles `([\/, ]+)` correspond à un espace. Ensuite, `ul` fonctionne comme `div`.

Correspondance de `#answer-4185009 > table > tbody > td.answercell > div > pre`

Ce sélecteur CSS est utilisé pour cibler un élément spécifique dans une structure HTML. Voici une explication détaillée de chaque partie du sélecteur :

- `#answer-4185009` : Cible un élément avec l'ID `answer-4185009`.
- `> table` : Sélectionne un élément `table` qui est un enfant direct de l'élément précédent.
- `> tbody` : Sélectionne un élément `tbody` qui est un enfant direct de l'élément `table`.
- `> td.answercell` : Sélectionne un élément `td` avec la classe `answercell` qui est un enfant direct de l'élément `tbody`.
- `> div` : Sélectionne un élément `div` qui est un enfant direct de l'élément `td`.
- `> pre` : Sélectionne un élément `pre` qui est un enfant direct de l'élément `div`.

En résumé, ce sélecteur cible un élément `pre` qui se trouve à l'intérieur d'une structure spécifique de table dans un élément avec l'ID `answer-4185009`.

Ensuite, voici un sélecteur un peu plus complexe : `#answer-4185009 > table > tbody > td.answercell > div > pre` (vous pouvez également ouvrir https://regex101.com/ et coller cela là-bas pour tester) :

Voici ce qui a été copié-collé depuis Chrome :

Première correspondance :

Parce que dans la règle du premier groupe `([\w-:\*>]*)`, aucun des caractères dans `[]` ne peut correspondre à `#`, et ensuite, parce que le `*` à la fin permet de correspondre 0 fois ou un nombre illimité de fois, ici c'est 0 fois. Ensuite, la description de la règle du deuxième groupe est :

Comme nous l'avons déjà analysé, examinons directement la partie `\#([\w-]+)` avant le `|`. Ici, `\#` correspond au caractère `#`, et `[\w-]+` correspond à `answer-4185009`. Ensuite, pour la partie `\.([\w-]+)`, si nous avons `.answer-4185009`, cette correspondance sera appliquée.

Ensuite, examinons la correspondance `td.answercell`,

La première partie de la règle `([\w-:\*>]*)` correspond à `td`, et la deuxième partie de la règle `(?:\#([\w-]+)|\.([\w-]+))?`, plus précisément `\.([\w-]+)`, correspond à `.answercell`.

L'analyse de ce sélecteur s'achève ici.

Correspondre à `a[href="http://google.com/"]`

Ensuite, nous allons faire correspondre le sélecteur `a[href="http://google.com/"]` :

Regardons le troisième bloc :

La troisième grande partie de l'expression est `(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?`. Tout d'abord, la partie la plus externe `(?:)` indique qu'il s'agit d'un groupe non capturant, et le `?` à la fin signifie que cette grande partie peut correspondre 0 ou 1 fois. Si on l'enlève, on obtient `\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\]`. `\[` correspond au caractère `[`. `@?` signifie que le caractère `@` est optionnel. Ensuite, le groupe `(!?[\w-:]+)` indique que `!` est optionnel, et `[\w-:]+` correspond à `href`. Le groupe suivant `(?:([!*^$]?=)["']?(.*?)["']?)` est un groupe non capturant. Si on enlève la couche la plus externe, on obtient `([!*^$]?=)["']?(.*?)["']?`. Ici, `([!*^$]?=)` signifie que `[!*^$]?` correspond à 0 ou 1 caractère parmi ceux entre `[]`. Ensuite, `=` correspond directement. Puis, `["']?(.*?)["']?` correspond à `"http://google.com/"`. `["']?` signifie qu'il correspond à `"` ou `'` ou aucun des deux. Si on enlève cette couche externe, on obtient `(.*?)` qui correspond à `http://google.com/`. Ici, `*?` signifie qu'il correspond à aussi peu de caractères que possible, c'est-à-dire que s'il y a `"` ou `'`, il doit être laissé pour l'expression suivante `["']?` à correspondre. Ainsi, il ne correspondra pas à `http://google.com/"`, mais seulement à `http://google.com/`. Donc, le sélecteur entier `a[href="http://google.com/"]` marque la fin de la correspondance.

Résumé

Enfin, j'ai compris ! Reprenons une fois de plus pour clarifier. L'expression complexe `([\w-:\*>]*)(?:\#([\w-]+)|\.([\w-]+))?(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?([\/, ]+)` est composée de quatre grandes parties :

- `([\w-:\*>]*)`
- `(?:\#([\w-]+)|\.([\w-]+))?`
- `(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?`
- `([\/, ]+)`

La troisième partie, la plus complexe, est elle-même composée des éléments suivants :

- `\[`
- `(!?[\w-:]+)`
- `(?:([!*^$]?=)["']?(.*?)["']?)?`
- `\]`

Ainsi, ces parties suffisamment petites peuvent être résolues une par une. Ensuite, cherchez plus d'exemples pour voir comment chaque exemple correspond, tout en utilisant les expli-

cations de https://regex101.com/ pour analyser. De cette manière, vous comprendrez cette expression régulière qui semble complexe, et vous réaliserez qu'elle n'est en fait qu'un tigre de papier !