複雜正則表達式

最近在研究 HTML 解析，遇到一個正則表達式：

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

它用來匹配 CSS 選擇器，比如 `div > ul`。

過去見過很多這樣複雜的表達式，我都本能地退縮了。今天就來徹底搞明白它！男人，該對自己狠一點！

匹配 `div > ul`

我找了一個網站 https://regex101.com/ ，能線上匹配，還有解釋。

雖然有了右邊的說明，清楚了一些。但還是不清楚具體匹配起來是怎樣的。那就找幾個例子，逐個分析。

具體出現這個正則表達式的代碼是：

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

`preg_match_all` 的意思是獲取所有滿足模式的串。如果有：

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

第一個參數是模式，第二個參數是要匹配的字符串，第三個參數是結果引用。運行後，`$matches` 數組裡會包含兩個 `abc`。

有了這個理解之後，上圖的 `div > ul` 只匹配了前面四個字符 `div >`。`regex101` 不支持 `preg_match_all`？還好加個叫 `g` 的修飾符就行了：

加了 `g` 後，會匹配所有的，而不是匹配到第一個就返回。

加了之後，我們匹配到了 `div > ul`：

右邊顯示，第一個匹配中，即 `div`，我們用第一組的規則匹配到了 `div`，然後用第七組的規則匹配到了空格 。

我們接著來看第一組規則的解釋：

在這一大串表達式中，第一個括號括起來的叫第一組規則。這是一個捕獲組。括號自身不匹配，而是用來分組。`[]` 表示一個字符集合，裡面的規則說明這是一個怎樣的字符集合。這個字符集合裡有：

- `\w` 表示大小寫字母和 0 到 9 以及下劃線。
- `-`: 直接表示這兩個字符在集合裡。
- `\*` 因為 `*` 是正則表達式中的保留字符，有特殊含義，所以要用 `\` 來轉義，表示這是一個普通的 `*` 字符。
- `>` 簡單地表示 `>` 這個字符。

`[\w-:\*>]*` 最後的 `*` 表示前面的字符能出現 0 次或無數次，但要匹配盡可能多的次數。之所以能匹配 `div` 是因為，`\w` 匹配了 `d`、`i`、`v`。之所以不再繼續匹配後面的空格，是因為空格沒有出現在 `[]` 中。捕獲組的意思是，這組匹配會出現在結果數組中。相對應的還有非捕獲組，語法是 `(?:)`。上面的 `([\w-:\*>]*)` 如果不需要這組結果，可以記為 `(?:[\w-:\*>]*)`。

那麼不出現在結果中，直接不用括號不就行了？括號是為了分組，分組還是很有意義的。可以參考 《What is a non capturing group? (?:) - StackOverflow》。

接著講完了 `div` 滿足第一組規則後，講下空格 為什麼滿足第七組的規則。

`[\/, ]` 的意思是匹配這四個字符的任意一個，`+` 表示前面的匹配出現一次或無數次，次數要盡可能多。所以因為這四個字符包含空格，就匹配了我們的空格。又因為 `div` 之後下一個字符是 `>`，所以不再滿足第七組的規則，不繼續匹配了。

搞明白了 `div` 的匹配。那為什麼第二到第六組的規則沒有匹配這裡的空格，而是留給了第七組？

第二部分的解釋：

首先 `(?:)` 表示這是一個非捕獲組。最後面的 `?` 表示前面的匹配可以出現 0 次或 1 次。所以上面的 `(?:\#([\w-]+)|\.([\w-]+))?` 可以有或沒有。去掉外層的修飾符後，剩下的是 `\#([\w-]+)|\.([\w-]+)`，中間的 `|` 表示或，滿足其中一個即可。`\#([\w-]+)` 中的 `\#` 匹配 `#` 字符，`[\w-]+` 匹配其他字符。再看後一半，`\.([\w-]+)` 中的 `.` 匹配 `.` 字符。

所以 2 到 6 組都可能因為空格不是這些組要求的開頭字符而不滿足，又因為這些組有個 `?` 修飾符，不滿足也可以，因此跳到了第七組。

接著 `div > ul` 後面的 `>`，還是一樣：

第一組規則 `([\w-:\*>]*)` 匹配了 `>`，第七組規則 `([\/, ]+)` 匹配了空格。接著 `ul` 像 `div` 一樣。

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

接著來一個稍微複雜一點的選擇器 `#answer-4185009 > table > tbody > td.answercell > div > pre`（你也可以打開 https://regex101.com/ 把這個粘貼到那裡去來測試）：

這是從 Chrome 裡複製粘貼的：

第一個匹配：

因為第一組的規則 `([\w-:\*>]*)` 中 `[]` 裡的字符集沒有一個能匹配 `#`，接著因為最後面的 `*` 支持匹配 0 次或無數次，這裡是 0 次。接著第二組規則的描述是：

上面已經分析過。直接來看 `|` 前的 `\#([\w-]+)`，`\#` 把 `#` 匹配了，`[\w-]+` 匹配了 `answer-4185009`。後面的 `\.([\w-]+)`，如果是 `.answer-4185009` 就會應用這個匹配。

接著來看 `td.answercell` 這個匹配，

第一組的規則 `([\w-:\*>]*)` 匹配了 `td`，第二大部分的 `(?:\#([\w-]+)|\.([\w-]+))?` 的後面部分，即 `\.([\w-]+)`，匹配了 `.answercell`。

這個選擇器的分析也到此結束。

匹配 `a[href="http://google.com/"]`

接著我們來匹配選擇器 `a[href="http://google.com/"]`：

看看第三大塊：

第三大塊的表達式為 `(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?`，首先最外層 `(?:)` 表示這是一個非捕獲組，最後面的 `?` 表示這整個大的可以匹配 0 次或 1 次。去掉之後為 `\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\]`。`\[` 匹配 `[` 字符。`@?` 表示 `@` 字符可有可無。那麼接下來的一組 `(!?[\w-:]+)`，`!` 可有可無，`[\w-:]+` 匹配 `href`。接著的一組 `(?:([!*^$]?=)["']?(.*?)["']?)` 是非捕獲組，去掉最外層後是 `([!*^$]?=)["']?(.*?)["']?`。這裡的 `([!*^$]?=)`，`[!*^$]?` 表示匹配 0 個或 1 個 `[]` 裡的字符。接著 `=` 直接匹配。接著 `["']?(.*?)["']?`，匹配 `"http://google.com/"`，`["']?` 表示匹配 `"` 或 `'` 或者兩個都不匹配，去掉這個最外層後是 `(.*?)` 匹配 `http://google.com/`，這裡的 `*?` 表示盡可能少的匹配，也就是說，如果有 `"` 或 `'`，要留給後面的表達式 `["']?` 來匹配。所以不至於匹配到 `http://google.com/"`，而是只匹配 `http://google.com/`。所以整個選擇器 ‘a[href=”http://google.com/

”]‘就匹配結束了。

總結

終於搞懂了！再讓我們理清一次，首先整個複雜的表達式 `([\w-:\*>]*)(?:\#([\w-]+)|\.([\w-]+))?(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?` 由四大部分組成：

- `([\w-:\*>]*)`

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

最複雜的第三部分又由這幾個部分組成：

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

所以這些足夠小的部分都可以逐個擊破。然後找多一點例子，看看每個例子是怎麼匹配的，同時結合 https://regex101.com/ 的解釋來分析。就把這個看似複雜的正則表達式搞懂了，原來它是個紙老虎！