复杂正则表达式原来是纸老虎

最近在研究 HTML 解析，遇到一个正则表达式：

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

它用来匹配 CSS 选择器，比如 `div > ul`。

过去见过很多这样复杂的表达式，我都本能地退缩了。今天就来彻底搞明白它！男人，该对自己狠一点！

匹配 `div > ul`

我找了一个网站 https://regex101.com/ ，能在线匹配，还有解释。

虽然有了右边的说明，清楚了一些。但还是不清楚具体匹配起来是怎样的。那就找几个例子，逐个分析。

具体出现这个正则表达式的代码是：

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

`preg_match_all` 的意思是获取所有满足模式的串。如果有：

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

第一个参数是模式，第二个参数是要匹配的字符串，第三个参数是结果引用。运行后，`$matches` 数组里会包含两个 `abc`。

有了这个理解之后，上图的 `div > ul` 只匹配了前面四个字符 `div >`。`regex101` 不支持 `preg_match_all`？还好加个叫 `g` 的修饰符就行了：

加了 `g` 后，会匹配所有的，而不是匹配到第一个就返回。

加了之后，我们匹配到了 `div > ul`：

右边显示，第一个匹配中，即 `div`，我们用第一组的规则匹配到了 `div`，然后用第七组的规则匹配到了空格 。

我们接着来看第一组规则的解释：

在这一大串表达式中，第一个括号括起来的叫第一组规则。这是一个捕获组。括号自身不匹配，而是用来分组。`[]` 表示一个字符集合，里面的规则说明这是一个怎样的字符集合。这个字符集合里有：

- `\w` 表示大小写字母和 0 到 9 以及下划线。

- `-`: 直接表示这两个字符在集合里。
- `\*` 因为 `*` 是正则表达式中的保留字符，有特殊含义，所以要用 `\` 来转义，表示这是一个普通的 `*` 字符。
- `>` 简单地表示 `>` 这个字符。

`[\w-:\*>]*` 最后的 `*` 表示前面的字符能出现 0 次或无数次，但要匹配尽可能多的次数。之所以能匹配 `div` 是因为，`\w` 匹配了 `d`、`i`、`v`。之所以不再继续匹配后面的空格，是因为空格没有出现在 `[]` 中。捕获组的意思是，这组匹配会出现在结果数组中。相对应的还有非捕获组，语法是 `(?:)`。上面的 `([\w-:\*>]*)` 如果不需要这组结果，可以记为 `(?:[\w-:\*>]*)`。

那么不出现在结果中，直接不用括号不就行了？括号是为了分组，分组还是很有意义的。可以参考《What is a non capturing group? (?:) - StackOverflow》。

接着讲完了 `div` 满足第一组规则后，讲下空格 为什么满足第七组的规则。

`[\/, ]` 的意思是匹配这四个字符的任意一个，`+` 表示前面的匹配出现一次或无数次，次数要尽可能多。所以因为这四个字符包含空格，就匹配了我们的空格。又因为 `div` 之后下一个字符是 `>`，所以不再满足第七组的规则，不继续匹配了。

搞明白了 `div` 的匹配。那为什么第二到第六组的规则没有匹配这里的空格，而是留给了第七组？

第二部分的解释：

首先 `(?:)` 表示这是一个非捕获组。最后面的 `?` 表示前面的匹配可以出现 0 次或 1 次。所以上面的 `(?:\#([\w-]+)|\.([\w-]+))?` 可以有或没有。去掉外层的修饰符后，剩下的是 `\#([\w-]+)|\.([\w-]+)`，中间的 `|` 表示或，满足其中一个即可。`\#([\w-]+)` 中的 `\#` 匹配 `#` 字符，`[\w-]+` 匹配其他字符。再看后一半，`\.([\w-]+)` 中的 `.` 匹配 `.` 字符。

所以 2 到 6 组都可能因为空格不是这些组要求的开头字符而不满足，又因为这些组有个 `?` 修饰符，不满足也可以，因此跳到了第七组。

接着 `div > ul` 后面的 `>`，还是一样：

第一组规则 `([\w-:\*>]*)` 匹配了 `>`，第七组规则 `([\/, ]+)` 匹配了空格。接着 `ul` 像 `div` 一样。

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

接着来一个稍微复杂一点的选择器 `#answer-4185009 > table > tbody > td.answercell > div > pre`（你也可以打开 https://regex101.com/ 把这个粘贴到那里去来测试）：

这是从 Chrome 里复制粘贴的：

第一个匹配：

因为第一组的规则 `([\w-:\*>]*)` 中 `[]` 里的字符集没有一个能匹配 `#`，接着因为最后面的 `*` 支持匹配 0 次或无数次，这里是 0 次。接着第二组规则的描述是：

上面已经分析过。直接来看 | 前的 \#([\w-]+)，\# 把 # 匹配了，[\w-]+ 匹配了 answer-4185009。后面的 \.([\w-]+)，如果是 .answer-4185009 就会应用这个匹配。

接着来看 td.answercell 这个匹配，

第一组的规则 ([\w-:*>]*) 匹配了 td，第二大部分的 (?:\#([\w-]+)|\.([\w-]+))? 的后面部分，即 \.([\w-]+)，匹配了 .answercell。

这个选择器的分析也到此结束。

匹配 a[href="http://google.com/"]

接着我们来匹配选择器 a[href="http://google.com/"]：

看看第三大块：

第三大块的表达式为 (?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?，首先最外层 (?:) 表示这是一个非捕获组，最后面的 ? 表示这整个大的可以匹配 0 次或 1 次。去掉之后为 \[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\]。\[匹配 [字符。@? 表示 @ 字符可有可无。那么接下来的一组 (!?[\w-:]+)，! 可有可无，[\w-:]+ 匹配 href。接着的一组 (?:([!*^$]?=)["']?(.*?)["']?) 是非捕获组，去掉最外层后是 ([!*^$]?=)["']?(.*?)["']?。这里的 ([!*^$]?=)，[!*^$]? 表示匹配 0 个或 1 个 [] 里的字符。接着 = 直接匹配。接着 ["']?(.*?)["']?，匹配 "http://google.com/"，["']? 表示匹配 " 或 ' 或者两个都不匹配，去掉这个最外层后是 (.*?) 匹配 http://google.com/，这里的 *? 表示尽可能少的匹配，也就是说，如果有 " 或 '，要留给后面的表达式 ["']? 来匹配。所以不至于匹配到 http://google.com/"，而是只匹配 http://google.com/。所以整个选择器 ‘a[href= “http://google.com/

“]‘ 就匹配结束了。

总结

终于搞懂了!再让我们理清一次,首先整个复杂的表达式 ([\w-:*>]*)(?:\#([\w-]+)|\.([\w-]+))?(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?([\/,]+) 由四大部分组成：

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

最复杂的第三部分又由这几个部分组成：

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

- \]

所以这些足够小的部分都可以逐个击破。然后找多一点例子，看看每个例子是怎么匹配的，同时结合 https://regex101.com/ 的解释来分析。就把这个看似复杂的正则表达式搞懂了，原来它是个纸老虎！