複雑な正規表現

最近、HTML 解析について研究している際に、次のような正規表現に出会いました：

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

この正規表現は、特定のパターンにマッチする文字列を検索するために使用されます。以下に、各部分の説明を示します。

1. `([\w-:\*>]*)`:
    - `\w`: 単語文字（アルファベット、数字、アンダースコア）にマッチ。
    - `-`: ハイフンにマッチ。
    - `:`: コロンにマッチ。
    - `\*`: アスタリスクにマッチ。
    - `>`: 大なり記号にマッチ。
    - `*`: 直前の文字が 0 回以上繰り返されることを示す。
    - 全体として、これらの文字の任意の組み合わせにマッチ。
2. `(?:\#([\w-]+)|\.([\w-]+))?`:
    - `\#([\w-]+)`: ハッシュ記号に続く単語文字またはハイフンの 1 回以上の繰り返しにマッチ。
    - `\.([\w-]+)`: ドットに続く単語文字またはハイフンの 1 回以上の繰り返しにマッチ。
    - `|`: またはを示す。
    - `(...)?`: 全体が 0 回または 1 回出現することを示す。
3. `(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?`:
    - `\[`: 開き角括弧にマッチ。
    - `@?`: @ 記号が 0 回または 1 回出現することを示す。
    - `(!?[\w-:]+)`: 感嘆符が 0 回または 1 回出現し、その後に単語文字、ハイフン、コロンの 1 回以上の繰り返しにマッチ。
    - `(?:([!*^$]?=)["']?(.*?)["']?)?`:
        - `([!*^$]?=)`: 感嘆符、アスタリスク、キャレット、ドル記号のいずれかが 0 回または 1 回出現し、その後に等号が続く。
        - `["']?(.*?)["']?`: 引用符が 0 回または 1 回出現し、その間に任意の文字列が 0 回以上繰り返される。
    - `\]`: 閉じ角括弧にマッチ。
    - `(...)?`: 全体が 0 回または 1 回出現することを示す。

4. `([\/, ]+)`:
    - `\/`: スラッシュにマッチ。
    - `,`: カンマにマッチ。
    - ` `: スペースにマッチ。
    - `+`: 直前の文字が 1 回以上繰り返されることを示す。
5. `/is`:
    - `i`: 大文字小文字を区別しない。
    - `s`: ドットが改行文字にもマッチする。

この正規表現は、HTML や CSS のセレクタのようなパターンにマッチする文字列を検索するために使用されることがあります。

これは、`div > ul` のような CSS セレクタをマッチングするために使用されます。

以前、このような複雑な式を見たことが何度もあり、本能的に避けてきました。今日こそ、これを徹底的に理解しましょう！男として、自分に厳しくあるべきです！

`div > ul` のマッチング

`div > ul` というセレクタは、`div` 要素の直接の子要素である `ul` 要素にマッチします。つまり、`div` の直下にある `ul` だけが選択され、それより深い階層にある `ul` は選択されません。

例えば、以下の HTML があるとします：

```
<div>
  <ul> <!-- この ul はマッチする -->
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
  <div>
    <ul> <!-- この ul はマッチしない -->
      <li>Item 3</li>
      <li>Item 4</li>
    </ul>
  </div>
</div>
```

この場合、`div > ul` セレクタは最初の `ul` にのみマッチし、2 番目の `ul` にはマッチしません。

私は https://regex101.com/ というウェブサイトを見つけました。これはオンラインで正規表現のマッチングができ、さらにその解説も提供してくれます。

右側の説明を読んで、少しは理解できましたが、具体的にどのようにマッチングされるのかはまだわかりません。そこで、いくつかの例を挙げて、一つ一つ分析してみましょう。

具体的にこの正規表現が現れるコードは次のとおりです：

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

このコードは、PHP で正規表現を使用して文字列を検索し、マッチした結果を配列に格納するものです。具体的には、`preg_match_all` 関数を使って、指定されたパターン `$this->pattern` に基づいて、`trim($selector)` でトリミングされた文字列にスペースを追加したものを検索します。マッチした結果は `$matches` 配列に格納され、`PREG_SET_ORDER` フラグによって、各マッチが個別の配列として格納されます。

`preg_match_all` の意味は、パターンに一致するすべての文字列を取得することです。もし以下のような場合：

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

このコードは、PHP の `preg_match_all` 関数を使用して、文字列 `"abcdabc"` の中からパターン `"abc"` にマッチするすべての部分を検索し、その結果を `$matches` 配列に格納します。`preg_match_all` は、正規表現を使用して文字列内のすべてのマッチを検索するための関数です。この場合、パターン `"abc"` は文字列 `"abcdabc"` に 2 回出現するため、`$matches` には 2 つのマッチが含まれます。

最初の引数はパターン、2 番目の引数はマッチさせる文字列、3 番目の引数は結果の参照です。実行後、`$matches` 配列には 2 つの `abc` が含まれます。

この理解を得た後、上の図の `div > ul` は最初の 4 文字 `div >` にのみマッチします。`regex101` は `preg_match_all` をサポートしていないのでしょうか？幸い、`g` という修飾子を追加すれば問題ありません：

`g` を追加すると、最初に一致したものだけでなく、すべての一致するものを検索します。

追加した後、`div > ul` にマッチしました：

右側に表示されている通り、最初のマッチでは、`div` に対して第一グループのルールが適用され、その後、第七グループのルールがスペース に適用されています。

それでは、最初のルールセットの説明を見ていきましょう：

この長い式の中で、最初の括弧で囲まれた部分を第一のルールグループと呼びます。これはキャプチャグループです。括弧自体はマッチングに使用されず、グループ化のために使われます。`[]` は文字集合を表し、その中のルールはどのような文字集合であるかを示します。この文字集合には以下のものが含まれます：

- `\w` は大文字小文字のアルファベット、0 から 9 までの数字、およびアンダースコアを表します。
- `-:` は直接、これらの 2 つの文字が集合内にあることを表します。
- `\*` は、`*` が正規表現における予約文字で特殊な意味を持つため、`\` を使ってエスケープし、これは通常の `*` 文字であることを示します。
- `>` は単純に `>` という文字を表します。

`[\w-:\*>]*` の最後の `*` は、前の文字が 0 回以上、できるだけ多く出現することを意味します。`div` にマッチする理由は、`\w` が `d`、`i`、`v` にマッチするためです。後続のスペースにマッチしない理由は、スペースが `[]` の中に含まれていないためです。キャプチャグループとは、このグループのマッチが結果の配列に含まれることを意味します。これに対応する非キャプチャグループもあり、その構文は `(?:)` です。上記の `([\w-:\*>]*)` が結果として不要な場合は、`(?:[\w-:\*>]*)` と記述することができます。

では、結果に現れないのであれば、単に括弧を使わなければいいのでは？括弧はグループ化のためであり、グループ化は非常に意味があります。参考として 《What is a non capturing group? (?:) - StackOverflow》 をご覧ください。

次に、`div` が最初のルールセットを満たすことを説明した後、スペース がなぜ第七のルールセットを満たすのかについて説明します。

`[\/, ]` の意味は、これら 4 つの文字のいずれかにマッチすることを示し、`+` は前のマッチが 1 回以上、可能な限り多く出現することを表します。したがって、これら 4 つの文字にはスペースが含まれているため、スペースにマッチします。また、`div` の次の文字が `>` であるため、第 7 グループのルールを満たさなくなり、それ以上のマッチは行われません。

`div` のマッチングが理解できました。では、なぜ 2 番目から 6 番目のグループのルールがここのスペースにマッチせず、7 番目のグループに残されたのでしょうか？

第二部の解説：

まず、`(?:)` は非キャプチャグループを表します。最後の `?` は、前のマッチが 0 回または 1 回出現することを示します。したがって、上記の `(?:\#([\w-]+)|\.([\w-]+))?` は、存在しても存在しなくても構いません。外側の修飾子を取り除くと、残りは `\#([\w-]+)|\.([\w-]+)` となり、中間の `|` は「または」を意味し、どちらか一方が満たされれば良いです。`\#([\w-]+)` の `\#` は `#` 文字にマ

ッチし、[\w-]+ は他の文字にマッチします。後半を見ると、\.([\w-]+) の . は . 文字にマッチします。

したがって、2 番目から 6 番目のグループは、スペースがこれらのグループの要求する開始文字ではないため、条件を満たさない可能性があります。また、これらのグループには ? 修飾子が付いているため、条件を満たさなくても問題ありません。そのため、7 番目のグループに進みます。

div > ul の後の > も同様です：

最初のルールグループ ([\w-:*>]*) は > にマッチし、7 番目のルールグループ ([\/,]+) はスペースにマッチします。その後、ul は div と同様に処理されます。

マッチ #answer-4185009 > table > tbody > td.answercell > div > pre

次に、少し複雑なセレクター #answer-4185009 > table > tbody > td.answercell > div > pre を紹介します（テストするために、https://regex101.com/ を開いてそこに貼り付けることもできます）：

これは Chrome からコピー＆ペーストしたものです：

最初の一致：

最初のグループのルール ([\w-:*>]*) において、[] 内の文字集合には # にマッチするものが含まれていません。そのため、最後の * が 0 回以上のマッチをサポートしていることから、ここでは 0 回のマッチとなります。続いて、第二のグループのルールの説明は以下の通りです：

上記で既に分析しました。直接 | の前の \#([\w-]+) を見てみましょう。\# は # にマッチし、[\w-]+ は answer-4185009 にマッチします。後ろの \.([\w-]+) は、もし .answer-4185009 であれば、この部分にマッチします。

次に、td.answercell というマッチングを見ていきましょう。

最初のグループのルール ([\w-:*>]*) は td にマッチし、第二の大部分 (?:\#([\w-]+)|\.([\w-]+))? の後半部分、すなわち \.([\w-]+) は .answercell にマッチします。

このセレクタの分析はこれで終了です。

a[href="http://google.com/"] にマッチ

次に、セレクター a[href="http://google.com/"] をマッチングさせます：

第三の大きなブロックを見てみましょう：

第三大块的表达式は (?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])? です。まず、最外層の (?:) は非キャプチャグループを表し、最後の ? はこの大きなブ

ロック全体が0回または1回マッチすることを意味します。これを取り除くと `\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\]` になります。`\[` は `[` 文字にマッチします。`@?` は `@` 文字が存在しても存在しなくても良いことを示します。次のグループ `(!?[\w-:]+)` では、`!` が存在しても存在しなくてもよく、`[\w-:]+` は `href` にマッチします。次のグループ `(?:([!*^$]?=)["']?(.*?)["']?)` は非キャプチャグループで、最外層を取り除くと `([!*^$]?=)["']?(.*?)["']?` になります。ここで、`([!*^$]?=)` の `[!*^$]?` は `[]` 内の文字を0個または1個マッチすることを示します。続く `=` は直接マッチします。そして `["']?(.*?)["']?` は `"http://google.com/"` にマッチします。`["']?` は `"` または `'` のいずれか、またはどちらもマッチしないことを示し、この最外層を取り除くと `(.*?)` は `http://google.com/` にマッチします。ここで、`*?` はできるだけ少ない文字列にマッチすることを意味します。つまり、`"` または `'` がある場合、それらは後続の式 `["']?` にマッチさせるため、`http://google.com/"` ではなく、`http://google.com/` だけにマッチします。したがって、セレクタ全体 `a[href="http://google.com/` はこのように解析されます。

`"]` でマッチングが終了します。

まとめ

やっと理解できました！もう一度整理してみましょう。まず、この複雑な正規表現 `([\w-:\*>]*)(?:\#([\w-]+)|\.([\w-]+))?(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?([\/, ]+)` は、大きく4つの部分で構成されています：

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

これらの正規表現パターンは、特定の文字列をマッチングさせるために使用されます。それぞれのパターンの意味は以下の通りです：

1. `([\w-:\*>]*)`:
    - アルファベット、数字、アンダースコア (`\w`)、ハイフン (`-`)、コロン (`:`)、アスタリスク (`*`)、大なり記号 (`>`) のいずれかの文字が0回以上繰り返される文字列にマッチします。
2. `(?:\#([\w-]+)|\.([\w-]+))?`:
    - `#` で始まる1つ以上のアルファベット、数字、アンダースコア、ハイフンからなる文字列、または `.` で始まる1つ以上のアルファベット、数字、アンダースコア、ハイフンか

らなる文字列にマッチします。このグループはオプションで、存在しなくてもマッチします。

3. `(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?`:
    - `[` で始まり `]` で終わる、`@` がオプションで付いた 1 つ以上のアルファベット、数字、アンダースコア、ハイフン、コロンからなる文字列にマッチします。さらに、`!`、`*`、`^`、`$` のいずれかの記号がオプションで付いた `=` 記号と、その後に続くオプションの引用符で囲まれた任意の文字列にマッチします。このグループもオプションで、存在しなくてもマッチします。
4. `([\/, ]+)`:
    - スラッシュ (/)、カンマ (,)、スペース () のいずれかの文字が 1 回以上繰り返される文字列にマッチします。

これらのパターンは、HTML や CSS のセレクタ、あるいは特定のフォーマットの文字列を解析する際に使用されることがあります。

最も複雑な第三部分は、以下のいくつかの部分で構成されています：

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

上記の正規表現パターンを日本語で説明すると以下のようになります：

- `\[`：開き角括弧 `[` にマッチします。
- `(!?[\w-:]+)`：感嘆符 `!` が 0 回または 1 回続き、その後、単語文字（アルファベット、数字、アンダースコア）、ハイフン、コロンが 1 回以上続く文字列にマッチします。
- `(?:([!*^$]?=)["']?(.*?)["']?)?`：オプションで、感嘆符 `!`、アスタリスク `*`、キャレット `^`、ドル記号 `$` のいずれかが 0 回または 1 回続き、等号 `=` が続くパターンにマッチします。その後、オプションでシングルクォート `'` またはダブルクォート `"` で囲まれた任意の文字列にマッチします。
- `\]`：閉じ角括弧 `]` にマッチします。

このパターンは、特定の形式の角括弧内のテキストを抽出するために使用されます。

したがって、これらの十分に小さな部分は一つ一つ解決できます。次に、もっと多くの例を見つけて、それぞれの例がどのようにマッチするかを確認し、同時に https://regex101.com/ の説明を組み合わせて分析します。このようにして、一見複雑に見える正規表現を理解し、実は紙の虎であることがわかります！