正则表达式入门

涛叔 2021-01-06 ⏳ 12.6分钟(5.1千字) 创作✍️不易🤦请勿🚫广告

正则表达式说白了就是一堆约定俗成的匹配规则。如果从微观入手，你会发现有背不完的规则；如果从宏观入手，你会发现万变不离其宗。所以我们将从宏观到微观依次说起，依次为大家总结三大宏观规则、两大微观规则，再附带一部分高级内容。基本可以通过一篇文章让大家理解正则表达式的主要用法。

先说三大宏观规则。

宏观规则

> *我们用大写字母表示抽象的正则规则，主要讨论正则之间的关系，忽略具体的内容*

交集规则

如果有两个正则表达式 `E` 和 `F`，那么 `EF` 也是一个正则，表示同时匹配 `E` 和 `F` 的内容。这跟编程中的逻辑与是一个意思，跟集合中的交集也是一个意思。你也可以连

接任意多个正则，比如 EFGH 。

并集规则

如果有两个正则表达式 E 和 F ，那么 E|F 也是一个正则，表示匹配 E 或者匹配 F 。这跟编程中的**逻辑或**是一个意思，跟集合中的**并集**也是一个意思。 你可以使用 | 连接任意多个正则表达式，比如 E|F|G|H 。

如果你要连接非常多的正则，那就得写非常多的竖线，看起来非常乱。所以人们还约定了一种简化记法 [EFGH] 。也就是你写成 [EFGH] 跟写成 E|F|G|H 效果是一样，但前者更简短，更清晰。

补集规则

编程有**与或非**，集合有**交并补**。正则表达式有没有类似逻辑非或者集合中的补集呢？有，但记法比较复杂。如果你要排除匹配 E 和 F 的内容，需要写成 [^EF] 。

我们已经知道 [EF] 可以匹配那些匹配 E **或者**匹配 F 的内空，在前加上一个 ^ 表示取反。因为取了反，原来的逻辑或关系变成了逻辑与（具体请参见德摩根定律）。所以 [^EF] 匹配那些不能匹配 E **而且**不能匹配 F 的内容。

以上交并补规则单个看都不复杂，但它们可以任意组合，用起来就有点复杂。

比如 A|BC|[^D]E 表示匹配 A 或者同时匹配 B 和 C 或者不能匹配 D 但要匹配E的三种内容。这么长的规则只用 A|BC|[^D]E 就能准确表达，没有任何歧义，这就是正则的魅力。

这里有一个问题。 A|BC|[^D]E 表示 A 、 BC 和 [^D]E 的并集，还是表示 A|BC 跟 [^D]E 并集，还是表示 A 跟 BC|[^D]E 的并集呢？这就涉及到**结合优先级**的问题了。

正确的答案是第一种，被 | 分割的部分是平级的。如果你想表示 A|BC 跟 [^D]E 的并集，那你需要写成 (A|BC)|[^D]E 。对了，**遇事不决加括号**！

在前面的讨论中， ABCDE 都是抽像的正则，我们并**不关心具体的规则内容**。而对应的交、并补规则对所有正则都管用。现在我们开始讨论**两大微观规则**。

微观规则

单字符规则

所谓单字符就是一次匹配一个字符，但字符的取值可能是五花八门。

一个字母 a，一个数字 1 都是正则，分别匹配包含 a 和包含 1 的内容。

如果我们想匹配数字 1234，那么根据交集规则，我们写成 1234 就可以了。

如果我们想匹配所有可能出现的数字，则可以根据并集规则写成 0|1|2|3|4|5|6|7|8|9。是不有点长。我们可以简化成 [0123456789]。一下子少了很多坚线。

慢着，如果想匹配所有可能出现的小写字母呢？难不成要写成 [abc此处省略20个字母xyz]？太长了😂

正则表达式为此提供了一种更加简化方法——连字符，可以使用减号-表示**连续出现**的字符，只需写出头尾。所以我们可以把 [0123456789] 进一步简化成 [0-9]，把 [abc...xyz] 简化成 [a-z]。

如果要匹配所有字母，不区分大小写，可以写成 [a-zA-Z]，如果还想顺手匹配所有数字，可以写成 [a-zA-Z0-9]。

因为 [0-9] 很常用，大家又进一步简化成了 \d （对应单词 digit）。并非所有的正则引擎都支持这个 \d，有的默认不支持，需要开启 perl 兼容的正则引擎才行。

因为 [a-zA-Z] 也很常用，大家就把它简化成 \a （对应单词 alpha）。

同样的，如果想匹配大小写字母、数字和下划线（也就是所有单词字符），可以写成 [a-zA-Z0-9_]。也是因为太常用，大家将其简化为 \w （对应单词 word）。

如果想匹配一些空白字符，可以写成 [\t\r\n\v\f]，这个正则会匹配空格、水平制表符、回车、换行、 垂直制表符和 Page break 记。这里用到了跟 c 语言 printf 函数一样的转义字符。同样因为使用广泛，被简化为 \s （对应单词 space）。

我们说过，正则支持取反操作。[0-9] 表示匹配所有数字，那 [^0-9] 就表示匹配所有非数字字符。因为使用广泛，人们将其简写成 \D。大家注意，[0-9] 简写成 \d

（小写字母），对应的 `[^0-9]` 简写成 `\D` （大写字母）。以此类推， `\a` 取反是 `\A` 、 `\w` 取反是 `\W` 、 `\s` 取反是 `\S` 。一下子都记住了吧。

有了连字符和并集规则，**理论上我们可以匹配所有字符**。但是 Unicode 有上百万字符，难道我们都要写到方括号里吗？不能够。

我们可以利用取反操作。只要**排除少量不常用字符**，就可以匹配剩下的大多数字符了。但排除哪个呢？最终人们决定排除 `\n` 。为什么呢？因为一般而言，正则都是**逐行匹配**的，一次匹配一行内容，不会遇到换行符。最终可以用 `[^\n]` 表示匹配所有字符。同样因为太常用，这一写法被简化成句点 `.` 。 也就是说在正则表达式中，一个 `.` 可以匹配 `\n` 以外的所有字符。

最后需要额外说一下字符反斜杠 `\` 。我们前面提到的 `\d` 、 `\w` 都使用反斜杠进行转义。如果要匹配 `\` 就得写成 `\\` 。因为好多语言的字符串也是使用反斜杠进行转义（比如用 `\n` 表示换行），所以你会在代码中看到像 `"\\\\"` 这样的写法。如果你是初学者，一定会这样的鬼画符吓到。其实很简单。 第一个反斜杠用于转义第二个反斜杠，表示一个反斜杠字符；第三个转义第四个。第一个和第三个反斜杠是给**语言编译器**用的。如果你用printf之类的函数将这段打印到标准输出，你会看到 `\\` ，又是两个反斜杠。 这次转义是给**正则引擎**用的，用于表示匹配 `\` 这个字符。

那么多斜杠确实容易出错。好多语言都提供所谓 raw 字符串，这种字符串**不支持转义功能**，也就不需要写额外的反斜杠。比如在 go 语言中 `"\\\\"` 可以写成 `` \\ `` （注意两侧的 ``）是不是清爽多了🤪。

以上基本上就是单字符规则的所有内容。下面我们继续讨论**第二条，多字符规则**。

多字符规则

如果想匹配两位数字，可以利用交集规则，写成 `\d\d` ，此正则会先匹配一个数字再匹配一个数字，最终匹配的是两位数字。如果想匹配三位数字，需要写成 `\d\d\d` ，四位数字写成 `\d\d\d\d` 。

那如果想匹配一位**或者**两位数字**或者**三位数字**或者**四位数字（也就是四位以内的数字），需要写成

```
\d|\d\d|\d\d\d|\d\d\d\d
```

有点长，但是 it works！如果想匹配所有的八位以内的数字呢？那就得写很长很长了。为此，人们又想了个简化的办法。这次引入了大括号 `{}` 。

刚才的正则是可以简化成 `\d{1,4}` ，展开就是 `\d|\d\d|\d\d\d|\d\d\d\d` 。大括号中第一个数字表示最短匹配的次数，第二个数字表示最长匹配的次数。

如果想匹配八位以内的数字，就可以写成 `\d{1,8}` ，是不是很简洁呢？

如果只想匹配一个八位数，则可以写成 `\d{8,8}` 。重复写两个8好像有点多余，还是简化成 `\d{8}` 吧。

那能不能匹配任意长度的数字呢？ 理论上应该写成 `\d{1,∞}` ，只是这个 ∞ 不好写，干脆省略，写成 `\d{1,}` 算了。所以 `\d{1,}` 表示可以匹配一位、两位、一直到任意长度的数字。也就是说 `{1,}` 表示前面的匹配内容至少出现一次。因为这个至少出现一次也是特别常用，人们又进一步将其简化成 `+` ，最终我们的正则变成了 `\d+` ，优雅的不行。

那能不能实现匹配出现零次这种语义呢？ 可以，只要将大括号内第一个数字写成0就行。所以 `a{0,}` 可以匹配 `a` 、 `aa……aaaa……` 等各种情况。也就是说 `{0,}` 表示前面的匹配的内容出现多次或者不出现。 同样十分常用，被人们简化成了 `*` 。所以原来的正则可以简化成 `a*` 。

最后就是 `{0,1}` 这种情况了，显然表示出现零次或者一次。 不用说，懒人们将其简化成了 `?` 。所以 `ab?` 只能匹配 `a` 和 `ab` 两种情况。

看到这里，你基本已经理解了正则的常用功能。如果还不确定，就返回去再读几遍。最后我们引申出一些高级内容。

高级内容

先说一下贪心。

贪心

给定一段 html `<h1>this is a title</h1><p>this is content</p>` 。

如果我们想匹配h1标签的内容，我们可以写成 `<h1>.*</h1>` 。这里写了两遍 `h1` 。 显然，不想重复。我看到 `/h1` 后面有一个 `>` ，我能不能将正则改成 `<h1>.*>` 呢?

大家可以自己试一下。正则引擎会一直匹配到 `</p>` 里面的 `>` 。这是为什么呢？因为正则表达式默认使用贪心模式，一次性匹配尽可能长的内容。所以在找到 `/h1` 后面的 `>` 之后还会继续向前找。

那有办法修正这种行为吗？有，使用 `?` 。你可以将正则改为 `<h1>.*?>` ，这样引擎就会在第一次遇到 `>` 的地方停下来。这里的 `?` 跟之前说的「出现零次或一次」可不是一个意思哈。跟在 `*` 后面表示非贪心模式。其实也是为了避免引入太多的特殊符号，所以复用了 `?` ，这一定程度上会给初学者带来困扰。没办法，大家只能克服了。

再说一下引用。

引用

给定一段 html `<h1>this is a title</h1><h1>this is content</h2>` 。这里第二个 `<h1>` 没有闭合，结束标签写成了 `h2` 。如果只想匹配正常结束的 `h1` ，那可以写成 `<(h1)>.*</\1>` 。

这里有两点。第一，前面的 `h1` 两边加了括号。第二，在结尾的地方使用了 `\1` 来引用前面加括号的内容。 正则引擎会为每个括号分配一个编号（从1开始记数），并记录括号的内容，大家可以使用 `\+数字` 的方式来引用。这样的正则等价于 `<h1>.*</h1>` 。

咋一看也没什么大不了的。但如果要匹配很多成对标签的话，引用的优势就体现出来了。例如 `<(h1|p|artice|div)>.*?</\1>` 可以匹配 `h1` 、 `p` 、 `article` 和 `div` 四种闭合标签。Do not repeat yourself.

最后说一下环视。

环视

环视说起来有点抽像 。例子不太好举，这里我引用 [stack over flow] 的一篇回答内容。

比如我们有字符串 foobarbarfoo 。下面我将用大写字母表示想要匹配的内容。为了跟英语原文对应，我们规定当前字符**右边为前**（未处理），**左边为后**（已处理）。

如果只想匹配第一个出现的 bar ，也就是 fooBARbarfoo 。肯定不能只写成 bar ，因为第二个 bar 也会匹配到。我们希望正则引擎在每碰到一个 bar 的时候**继续向前（右）看看**还有没有 bar ，如果**还有**则说明不是第一个。所以，需要写成 bar(?=bar) 。括号里以 ? 开头， = 表示检查是否**出现**，因为是继续**向前（右）看**，所以叫做**前向肯定环视**（Look ahead positive）。

如果想匹配第二个 bar 也就是 foobarBARfoo ，则需要写成 bar(?!bar) 。也就是说查到 bar 只后还要继续**向前（右）看**，没有 bar 才算匹配到。因为是没有，所以叫**前向否定环视**（Look ahead negative）。

我还可以通过**向后（左）看**的办法来解决类似的问题。

如果想匹配第一个 bar ,也就是 fooBARbarfoo ，我们可以写成 (?<=foo)bar ，这是告诉正则引擎在找到 bar 之后还要回顾一下有没有遇到 foo ，只有碰到才算匹配成功。因为需向后（左）确定匹配成功，所以叫**后向肯定环视**（Look behind positive）。

如果想匹配第二个 bar ，也就是 foobarBARfoo ，我们可以写成 (?<!foo)bar ，这是让引擎在找到 bar 之后回顾一下有没有遇到过 foo ，没有碰到才算匹配成功。因为需要向后（左）确定匹配不成功，所以叫**后向否定环视**（Look behind negative）。

我们稍微回顾一下。所有的环视都需要用括号括起来，以 ? 开始。匹配之后继续向**前（右）检查**叫前**叫前向环视**，如果需要确保另一模式也匹配，叫**肯定环视**，用 =，否则是**否定环视**，用 ! ；匹配之后继续向**后（左）检查**叫**后向环视**，为了跟前向有所区别，所以在 ? 之后加了一个 < ，为大家指明方向，肯定和否标记则跟前向一样。

理解了环视，我们还可以做一些更有意思的事情。

第一个，可以匹配单词的边界。一个单词两边都有空格。单词的左边界是一个虚拟的位置，它右边的字符肯定是匹配 \w ，它左边的内容肯定匹配 \W 。我们可以要求正则引擎**同时向前看和向后看**。所以可以写成 (?<=\W)(?=\w) 。这里两个括号之间是空的，表示只匹配位置，不消耗内容。同样的，右边的边界也是个虚拟位置，它左

边的字符肯定是匹配 `\w`，它右边的字符肯定匹配 `\W` 的位置，所以可以写成 `(?<=\w)(?=\W)`。把这两个正则使用并集规则合到一起就是 `(?<=\W)(?=\w)|(?<=\w)(?=\W)`，就可以匹配单词的左右边界。同样因为常用，人们把它简化成了 `\b`。

也就是说 `\b` 匹配单词边界。正则 `\bbar` 只会匹配字符串 `foo bar` 中的 `bar`，而不会匹配 `foobar` 中的 `bar`。

第二个，可以匹配一行的开始和结束。一行的开始，顾名思义，就是第一个字符之前的位置，在它之前没有字符，在它之后是任意字符，所以我们可以写成 `(?<!.)(?=.)`。对于一行的结束，我们可以如法炮制，写成 `(?<=.)(?!.)`。同样因为常用，此二者被分别简化成了 `^` 和 `$`。

> *因为一行的开始跟结束非常特殊，正则引擎可以直接标记，根本用不到环视这样的大招。 我将它们放到一起讲只是为了逻辑上的统一。*

小结

差不多该搁笔了。所谓言有尽而意无穷。我们讲重构、讲抽象，说到底就是要站在更高的视角看问题，要有全局意识，要有大局观。学习正则就是一个很好的例子，如果不从整体上去认识它，就会陷入死记硬背各种模式境地，费时费力容易出错不说，最关键的是会消磨你的学习热情。所以，我们在埋头学习的时候一定要时常浮出水面透透气，多思考、多总结，这样才能事半功倍。与君共勉💯