
正则表达式实践篇
/ ^ [A-Z] \d{5} $ /gi

正则表达式实践篇

by 暖暖 on 2016-12-07

简单的练习：

题目

1. 与搜索字符串开始处的 3 个数字匹配。
2. 与除 a、b 和 c 以外的任何字符匹配。
3. `'1234567'.match(/\d{1,3}/g)` 的结果。
4. 不以“th”开头的单词匹配。
5. 对密码应用以下限制：其长度必须介于 4 到 8 个字符之间，并且必须至少包含一个数字。
6. 匹配一个中文字符。

答案

1. 与搜索字符串开始处的 3 个数字匹配：`/^\d{3}/`。
2. 与除 a、b 和 c 以外的任何字符匹配：`/[^abc]/`。
3. ‘1234567’ .match(/\d{1,3}/g)，根据贪婪原则，结果是 `["123", "456", "7"]`。
4. 不以“th”开头的单词匹配：`/\b(?!th)\w+\b/` 。
5. 对密码应用以下限制：其长度必须介于 4 到 8 个字符之间，并且必须至少包含一个数字：`/^(?=.*\d).{4,8}$/` 。首先.{4,8} 表示与包含 4-8 个字符的字符串匹配；然后.*表示单个字符（除换行符 \n 外）零次或多次，且后面跟着一个数字，注意(?=)只匹配一个位置。

6. 匹配一个中文字符：`/[\u4e00-\u9fa5]/`。

当然，可能答案不唯一，不必较真啦~ 主要目的是回忆熟悉一下语法~
如果还不了解正则，可以前往正则表达式理论篇 了解哇~

真正的实践来了

要想在复杂性和完整性之间取得平衡，一个重要因素是要了解将要搜索的文本。
好的正则表达式：

- 只匹配期望的文本，排除不期望的文本；
- 易于控制和理解；
- 保证效率。

有时候处理各种极端情况会降低成本/收益的比例。所以某些情况下，不完全依赖正则表达式完成全部工作，比如某些字段用子表达式()括起来，让内存记忆下来，然后再用其他程序来验证。

不过本文还是从学习正则的角度出发，全部依赖正则表达式来写的哇~~

匹配美元

正则表达式：`/^\$[0-9]+(\.[0-9][0-9])?$/`。

分为四部分：

- `^\$` 以美元符号开头。

- `[0-9]+` 至少包含一个数字。
- `(\.[0-9][0-9])?` 由一个点和两位数组成，匹配0次或1次，因为可能是整数或者是小数。
- `$` 最后的$表示以数字结尾的。

缺点：不能匹配$1,000

匹配24小时制的时间，比如 09:59

- 小时部分

方法一：分类逻辑为第一个数字(0、1、2)，可以分为三部分：上午 00点到09点（0可选）；白天10到19点；晚上20到23点。

因此有三个多选分支，得到的结果为：

```
1  0?[0-9]|1[0-9]|2[0-3]
```

还可以优化一下，合并前面的两个多选分支，得到：

```
1  [01]?[0-9]|2[0-3]
```

方法二：分类逻辑为第二个数字，可以分为两部分：[0-3]和[4-9]。为什么这么分？看看下面这个图就知道了，[0-3]多了一行（以2为第一个数字）：

0	1	2	3		4	5	6	7	8	9
00	01	02	03		04	05	06	07	08	09
10	11	12	13		14	15	16	17	18	19
20	21	22	23							

因此有两个多选分支，结果为：

```
[012]?[0-3]|[01]?[4-9]
```

- 分钟部分

分钟数比较简单，第一个数范围在0-5之间，第二个数在0-9之间，因此得到分钟数为:

```
[0-5][0-9]
```

- 最后的结果：

小时部分用(?:)包起来，起到一个分组的作用，且不保存匹配项；
冒号、分钟数拼起来；
最后加上一个分界`\b`表示单词的开始或结束，得到最终的结果：

```
/\b(?:[01]?[0-9]|2[0-3]):[0-5][0-9]\b/
// 或者
/\b(?:[012]?[0-3]|[01]?[4-9]):[0-5][0-9]\b/
```

- 验证：

```
var reg = /\b(?:[01]?[0-9]|2[0-3]):[0-5][0-9]\b/;
'现在是09:49点'.match(reg);      // ["09:49"]
'现在是009:490点'.match(reg);     // null
```

其实这个结果不能说完全正确，首先你要明白这个正则用在什么地方，比如是数据验证或者
复杂的字符串搜寻替换。

情景一：填写表单中的字符串必须为24小时制的时间，那么可能第一个`\b`需要改成^，第二个`\b`改成`$`。

情景二：用于复杂的字符串搜寻替换时，可能也会匹配这样子的字符串如’ 跑步用时19:50’ ,明显的，’ 19:50’ 表示19分50秒，而不是表示24小时制的时间19点50分。

匹配IP地址

IP地址的规则：点号分开的四个字段，每个字段在0-255之间。

- 第一步：

如果一个字段是一个数或两个数，肯定是在0-255的范围内的；
如果三位数，那么以0或者1开头的三位数也是合法的，即000-199。

从上面的陈述中我们就可以得到三个多选分支：

```
\d|\d\d|[01]\d\d
```

我们稍微合并一下这三个多选分支，得到：

```
[01]?\d\d?
```

- 第二步：

我们再来看以2开头的三位数：

第二位数小于5的时候，第三位数范围[0-9]都可以；第二位数等于5的时候，第三位数范围[0-5] ，因此得到两个多选分支：

```
2[0-4]\d|25[0-5]
```

- 第三步：

前两步合并起来，得到一个字段0-255的表示方法：

```
[01]?\d\d?|2[0-4]\d|25[0-5]
```

- 第四步：

四个字段合并起来，IP地址正则如下：

```
/^(?:[01]?\d\d?|2[0-4]\d|25[0-5])\.(?:[01]?\d\d?|2[0-4]\d|25[0-5])\.(?:[01]?\d\d?|2[0-4]\d|25[0-5])\.(?:[01]?\d\d?|2[0-4]\d|25[0-5])$/
```

点号要转义一下，^和$需要加上，否则可能匹配52123.3.22.993，因为其中的123.3.22.99是符合的。(?:)起到分组的作用，且不保存匹配项。

一些测试结果：

```
var reg = /^(?:[01]?\d\d?|2[0-4]\d|25[0-5])\.(?:[01]?\d\d?|2[0-4]\d|25[0-5])\.(?:[01]?\d\d?|2[0-4]\d|25[0-5])\.(?:[01]?\d\d?|2[0-4]\d|25[0-5])$/;

'123.11.22.33.44'.match(reg); // null
```

```
'52123.3.22.993'.match(reg);  // null
'123.11.22.33'.match(reg);  // ["123.11.22.33"]
'0.0.0.0'.match(reg);   // ["0.0.0.0"]
```

虽然0.0.0.0是合法的，但它是非法的IP地址，使用正则的否定顺序环视功能(零宽负向先行断言)，可加上(?!0+.0+.0+.0+$) ：

```
var reg = /^(?!0+.0+.0+.0+$)(?:[01]?\d\d?|2[0-4]\d|25[0-5])\.(?:[01]?\d\d?|2[0-4]\d|25[0-5])\.(?:[01]?\d\d?|2[0-4]\d|25[0-5])\.(?:[01]?\d\d?|2[0-4]\

'123.11.22.33'.match(reg); // ["123.11.22.33"]
'0.0.0.0'.match(reg);  // null
```

匹配分隔符之内的文本

常见的匹配要求

- 匹配 `/*` 和 `*/` 之间的css注释。
- 匹配引文字符串”“ ，且容许其中包含转义的引号\”。
- 匹配一个HTML tag，也就是尖括号之内的文本，例如。

匹配思路的步骤

1. 匹配起始分隔符。
2. 匹配正文（即结束分隔符之前的所有文本）。
3. 匹配结束分隔符。

容许引文字符串中出现转义引号

大概思路

描述：起始分隔符和结束分隔符都是 `"` ,且正文中容许出现转义之后的引号 `\"` 。

简单情况分析：

举例：匹配类似 `I "start \"x3\" end" U` 文本的 `"start \"x3\" end"` 引文字符串，注意 `\"` 属于转义引号。

- 起始分隔符和结束分隔符都是 `"` 。
- 字符不是引号，肯定是正文。即 `[^"]` 表示不是引号的其他任意字符。
- 引号 `"` 前面有反斜线\，且被反斜线\转义，则也属于正文。例如 `start\"` 引号的前面有一个反斜线，那么这个引号也属于正文。即(?<=\)”表示匹配一个引号，它的前面有一个 `\` ，注意正则的反斜线也要用 `\` 来转义一下,因为\是特殊字符。

用非捕获分组 `(?:)` 将 `[^"]|(?<=\\)"` 括起来，给个量词 `*` ，表示匹配正文0次或多次。
因此可以写出正则表达式： `/"(?:[^"]|(?<=\\)")*"/`
注意：ES7才支持逆序环视 `(?<=)`

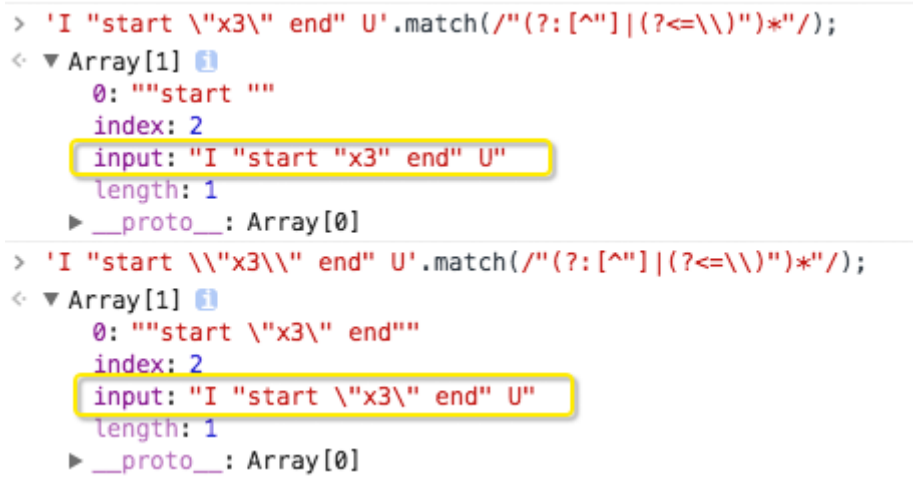
验证1：

验证正则： `/"(?:[^"]|(?<=\\)")*"/`

```
'I "start \"x3\" end" U'.match(/"(?:[^"]|(?<=\\)")*"/);
// 结果:  [""start ""]

'I "start \\"x3\\" end" U'.match(/"(?:[^"]|(?<=\\)")*"/);
// 结果: [""start \"x3\" end""]
```

为什么第2个才是对的呢？我们看一下返回的input属性就了解了：



验证2：

验证正则：`/"(?:[^"]|(?<=\\)")*"/`

```
'I "start \\"x3\\\\" end" U'.match(/"(?:[^"]|(?<=\\)")*"/);
// 结果与期望不符合：[""start \"x3\\" end""]
// 期望：[""start\"x3\\"]
// 注意返回的input属性为："I "start \"x3\\" end" U"
```

引号”前面有反斜线\，但是这个反斜线不是转义引号的，那么引号就不应该属于正文，而是属于结束分隔符。
什么情况反斜线\不转义引号呢？
这个反斜线\本身就是被转义的情况。

上面的结果按照预期结果应该返回 `[""start\"x3\\"]` ，但是现在多了`end"`。
因此验证这个正则表达式不正确。

也就是说，正文中可出现转义的字符，因此得出正则 `\\.` ，注意第一个 `\` 表示转义第二个 `\` ,点表示匹配除换行符 \n 之外的任何单个字符），例如可以匹配 `\+` 或者 `\\` 。而且转义的字符已经包含了 `\"` 的情况，因此正则 `(?<=\\)"` 可以不用写了，且替换成 `\\.` 。

因此改正后的正则： `/"(?:\\.|[^"])*"/`

你可能注意到了，我把[^”]和\.的位置调换一下，后面的验证3会讲到为什么要这么做。

验证3：

验证正则： `/"(?:\\.|[^"])*"/` 和 `/"(?:[^"]|\\.)*"/`

```
'I "start \\"x3\\\\" end" U'.match(/"(?:\\.|[^"])*"/);
// 结果与期望符合：[""start \"x3\\""]
// input: "I "start \"x3\\" end" U"

// [^"]和\\.的位置调换
'I "start \\"x3\\\\" end" U'.match(/"(?:[^"]|\\.)*"/);
// 结果与期望不符合：[""start \""]
// 期望：[""start\"x3\\"]
// input: "I "start \"x3\\" end" U"
```

`[^"]` 和 `\\.` 的位置调换后，结果与期望不符合。那是因为 `[^"]` 匹配 `start \` 后，遇到紧接着的 `"` 不匹配，交给后面的多选分支 `\\.` ，也不匹配，又刚好结束分隔符是 `"` ,导致匹配成功，结束匹配。

因此两个正则之间 正确的正则是 `/"(?:\\.|[^"])*"/`

验证4：

验证： `/"(?:\\.|[^"])*"/`

```
'I "start \\"x3\\" end U'.match(/"(?:\\.|[^"])*"/);
// 结果与期望不符合：[""start \"x3\""]
// 注意end后面少了",期望结果是null，不匹配
// input: "I "start \"x3\" end U"
```

上面的字符串 `"start\"x3\"` 其实是没有结束分隔符的，但是还是匹配了。那是因为正则 `[^"]` 和 `\\.` 一起作用,导致匹配到了文本U末尾，后续想找结束分隔符的时候，结果却找不到，所以只能回溯文本去找结束分隔符，最后找到了 `x3\` 后面的引号，匹配成功，结束匹配。

回溯会导致不期望的结果，由于是卡在多选分支上出错的，因此猜测多选分支 | 匹配内容出现重叠。

你想想，如果符合正文的反斜线，不是以 `[^"]` 方式匹配,而是以 `\\.` 的方式匹配，那就不会把好好的 `\"` 拆开来匹配了。

综上所述，一定要让反斜线是以 `\\.` 的方式匹配，字符串里的反斜杆不能以 `[^"]` 方式匹配。
因此将 `[^"]` 改成 `[^\\"]`。这样子就可以确保正确识别正文特殊的 `\"` 和结束分隔符 `"` 了。

注意：很多字符在[]都会失去本来的意义，但是反斜杠字符 `\` 仍为转义字符。若要匹配反斜杠字符，请使用两个反斜杠 `\\`。

改正的正则：`/"(?:\\.|[^\\"])*"/`

验证5

验证：`/"(?:\\.|[^\\"])*"/`

```
'I "start \\"x3\\" end U'.match(/"(?:\\.|[^\\"])*"/);
// 结果与期望符合：null
// input: "I "start \"x3\" end" U"


'I "start \\"x3\\" end" U'.match(/"(?:\\.|[^\\"])*"/);
// 结果与期望符合：[""start \"x3\" end""]
```

```
8  // input: "I "start \"x3\" end" U"
```

为了优化，我们可以把 `[^\\"]` 放在前面，因为普通字符的匹配可能性更大。
注意：优化正则提高效率最需要考虑的问题：改动是否会影响匹配。只有在排序与匹配成功无关时才不会影响准确性，才能重新安排多选分支的顺序。

优化后的正则：`/"(?:[^\\"]|\\.)*"/`

HTML Tag

经历了容许引文字符串中出现转义引号的例子分析，瞬间觉得这个容易了许多。

描述与要求：匹配类似 `<input name=123 value=">" >` 的HTML标签，起始分隔符是 `<` ，结束分隔符是 `>` ，且HTML 标签属性值中可以出现 `>` 。

起始分隔符和结束分隔符都是明确的，我们来分类一下正文。

- 双引号引用文本
- 单引号引用文本
- 除了>和引号之外的任意字符

可能你会当心单双引号引用文本，会像“容许引文字符串中出现转义引号”那么复杂。幸好是HTML Tag的属性值中不允许出现转义引号，因为平常的转义符号 `\` 变成了普通字符。

根据三种情况，分别写出三个正则：

- `"[^"]*"`

- `'[^']*'`
- `[^'">]`

好了，用多选分支连起来`"[^"]*"|'[^']*'|[^'">]`，再用非捕获分组(?:)将多选分支括起来，如`(?:"[^"]*"|'[^']*'|[^'">])`，用`*`表示匹配任意次，最后前后加上开始结束分隔符，搞定：

`/<(?:"[^"]*"|'[^']*'|[^'">])*>/`

验证：

```
'<input name=123 value=">" >'.match(/<(?:"[^"]*"|'[^']*'|[^'">])*>/)
// 结果:  ["<input name=123 value=">" >"]
```

体会：

看到没有，几乎每个正则都包含多选分支，只要你懂得将数据分类，离成功就不远了。哈哈哈哈哈。

参考

《精通正则表达式》
http://imweb.io/topic/56e804ef1a5f05dc50643106

上次更新：2016-12-30 10:19:37

源码赏析 - 1K的Firewatch游戏

vim给代码行尾加分号的几种方法

开发者头条

感谢分享！已推荐到《开发者头条》：https://toutiao.io/posts/ai1hg7 欢迎点赞支持！
欢迎订阅《小弧光黑板报》：https://toutiao.io/subjects/458

6小时前 回复 顶 转发

服务/产品

拇指期刊

前端代码规范

邮件签名工具

Excel Filter

Athena

HaloJS

MAC全栈环境

友情链接

JDC京东设计中心

淘宝FED

ISUX

携程UED

百度FEX

TGIdeas

CDC