



下载APP



## 11 | 如何理解正则的匹配原理以及优化原则？

2020-07-08 涂伟忠

正则表达式入门课

[进入课程 >](#)**讲述：涂伟忠**

时长 18:18 大小 16.78M



你好，我是伟忠，这一节课我们一起来学习正则匹配原理相关的内容，以及在书写正则时的一些优化方法。

这节课我主要给你讲解一下正则匹配过程，回顾一下之前讲的回溯，以及 DFA 和 NFA 引擎的工作方式，方便你明白正则是如何进行匹配的。这些原理性的知识，能够帮助我们快速理解为什么有些正则表达式不符合预期，也可以避免一些常见的错误。只有了解正则引擎的工作原理，我们才可以更轻松地写出正确的，性能更好的正则表达式。

### 有穷状态自动机



正则之所以能够处理复杂文本，就是因为采用了**有穷状态自动机 (finite automaton)**。那什么是有穷自动机呢？有穷状态是指一个系统具有有穷个状态，不同的状态代表不同的

意义。自动机是指系统可以根据相应的条件，在不同的状态下进行转移。从一个初始状态，根据对应的操作（比如录入的字符集）执行状态转移，最终达到终止状态（可能有一到多个终止状态）。

有穷自动机的具体实现称为正则引擎，主要有 DFA 和 NFA 两种，其中 NFA 又分为传统的 NFA 和 POSIX NFA。

[复制代码](#)

- 1 DFA：确定性有穷自动机 (Deterministic finite automaton)
- 2 NFA：非确定性有穷自动机 (Non-deterministic finite automaton)

接下来我们来通过一些示例，来详细看下正则表达式的匹配过程。

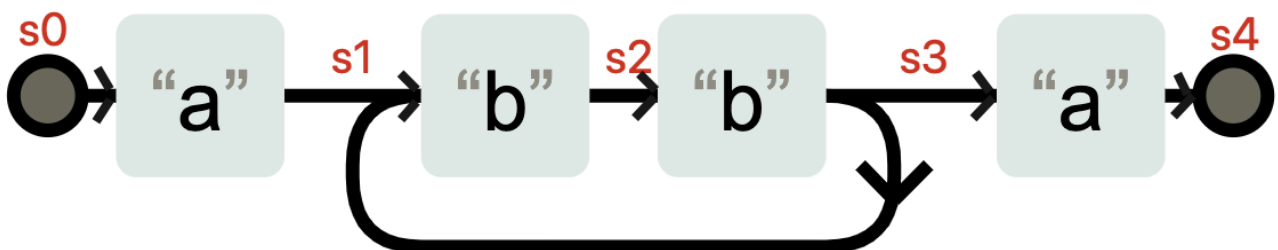
## 正则的匹配过程

在使用到编程语言时，我们经常会“编译”一下正则表达式，来提升效率，比如在 Python3 中它是下面这样的：

[复制代码](#)

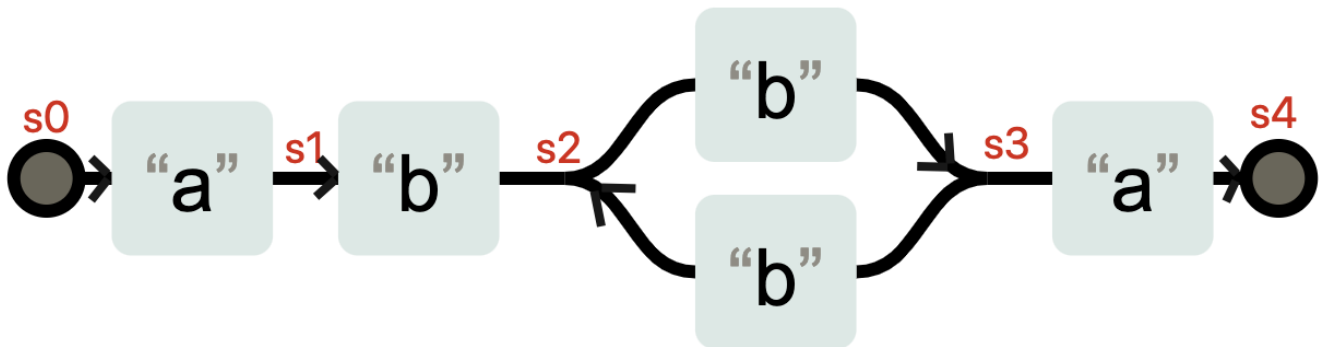
```
1 >>> import re
2 >>> reg = re.compile(r'a(?:bb)+a')
3 >>> reg.findall('abbbba')
4 ['abbbba']
```

这个编译的过程，其实就是生成自动机的过程，正则引擎会拿着这个自动机去和字符串进行匹配。生成的自动机可能是这样的（下图是使用 [Regexper 工具](#) 生成，再次加工得到的）。



在状态  $s_3$  时，不需要输入任何字符，状态也有可能转换成  $s_1$ 。你可以理解成  $a(bb)^+a$  在匹配了字符  $abb$  之后，到底在  $s_3$  状态，还是在  $s_1$  状态，这是不确定的。这种状态机就是非确定性有穷状态自动机（Non-deterministic finite automaton 简称 NFA）。

**NFA 和 DFA 是可以相互转化的**，当我们把上面的状态表示成下面这样，就是一台 DFA 状态机了，因为在  $s_0$ - $s_4$  这几个状态，每个状态都需要特定的输入，才能发生状态变化。



那这两种状态机的工作方式到底有什么不同呢？我们接着往下看。

## DFA& NFA 工作机制

下面我通过一个示例，来简单说明 **NFA 与 DFA 引擎工作方式的区别**：

复制代码


```
1 字符串: we study on jikeshijian app
2 正则: jike(zhushou|shijian|shixi)
```

NFA 引擎的工作方式是，先看正则，再看文本，而且以正则为主导。正则中的第一个字符是  $j$ ，NFA 引擎在字符串中查找  $j$ ，接着匹配其后是否为  $i$ ，如果是  $i$  则继续，这样一直找到  $jike$ 。

复制代码

```
1 regex: jike(zhushou|shijian|shixi)
2      ^
3 text: we study on jikeshijian app
4      ^
```


我们再根据正则看文本后面是不是 z，发现不是，此时 zhushou 分支淘汰。

 复制代码

```
1 regex: jike(zhushou|shijian|shixi)
2           ^
3           淘汰此分支(zhushou)
4 text: we study on jikeshijian app
5           ^
```

我们接着看其它的分支，看文本部分是不是 s，直到 shijian 整个匹配上。shijian 在匹配过程中如果不失败，就不会看后面的 shixi 分支。当匹配上了 shijian 后，整个文本匹配完毕，也不会再看 shixi 分支。


假设这里文本改一下，把 jikeshijian 变成 jikeshixi，正则 shijian 的 j 匹配不上时 shixi 的 x，会接着使用正则 shixi 来进行匹配，重新从 s 开始（NFA 引擎会记住这里）。

 复制代码

```
1 第二个分支匹配失败
2 regex: jike(zhushou|shijian|shixi)
3           ^
4           淘汰此分支(正则j匹配不上文本x)
5 text: we study on jikeshixi app
6           ^
7
8 再次尝试第三个分支
9 regex: jike(zhushou|shijian|shixi)
10          ^
11 text: we study on jikeshixi app
12          ^
```

也就是说，NFA 是以正则为主导，反复测试字符串，这样字符串中同一部分，有可能被反复测试很多次。

而 DFA 不是这样的，DFA 会先看文本，再看正则表达式，是以文本为主导的。在具体匹配过程中，DFA 会从 we 中的 w 开始依次查找 j，定位到 j，这个字符后面是 i。所以我们接着看正则部分是否有 i，如果正则后面是个 i，那就以同样的方式，匹配到后面的 ke。

 复制代码

```
1 text: we study on jikeshijian app
```

```

2                ^
3 regex: jike(zhushou|shijian|shixi)
4                ^

```

继续进行匹配，文本 e 后面是字符 s，DFA 接着看正则表达式部分，此时 zhushou 分支被淘汰，开头是 s 的分支 shijian 和 shixi 符合要求。

[复制代码](#)

```

1 text: we study on jikeshijian app
2                ^
3 regex: jike(zhushou|shijian|shixi)
4                ^      ^      ^
5                淘汰    符合    符合

```

然后 DFA 依次检查字符串，检测到 shijian 中的 j 时，只有 shijian 分支符合，淘汰 shixi，接着看分别文本后面的 ian，和正则比较，匹配成功。

[复制代码](#)

```

1 text: we study on jikeshijian app
2                ^
3 regex: jike(zhushou|shijian|shixi)
4                ^      ^
5                符合    淘汰

```

从这个示例你可以看到，DFA 和 NFA 两种引擎的工作方式完全不同。NFA 是以表达式为主导的，先看正则表达式，再看文本。而 DFA 则是以文本为主导，先看文本，再看正则表达式。

一般来说，DFA 引擎会更快一些，因为整个匹配过程中，字符串只看一遍，不会发生回溯，相同的字符不会被测试两次。也就是说 DFA 引擎执行的时间一般是线性的。DFA 引擎可以确保匹配到可能的最长字符串。但由于 DFA 引擎只包含有限的状态，所以它没有反向引用功能；并且因为它不构造显示扩展，它也不支持捕获子组。

NFA 以表达式为主导，它的引擎是使用贪心匹配回溯算法实现。NFA 通过构造特定扩展，支持子组和反向引用。但由于 NFA 引擎会发生回溯，即它会对字符串中的同一部分，进行很多次对比。因此，在最坏情况下，它的执行速度可能非常慢。

## POSIX NFA 与 传统 NFA 区别

因为传统的 NFA 引擎“急于”报告匹配结果，找到第一个匹配上的就返回了，所以可能会导致还有更长的匹配未被发现。比如使用正则 `pos|posix` 在文本 `posix` 中进行匹配，传统的 NFA 从文本中找到的是 `pos`，而不是 `posix`，而 POSIX NFA 找到的是 `posix`。



POSIX NFA 的应用很少，主要是 Unix/Linux 中的某些工具。POSIX NFA 引擎与传统的 NFA 引擎类似，但不同之处在于，POSIX NFA 在找到可能的最长匹配之前会继续回溯，也就是说它会尽可能找最长的，如果分支一样长，以最左边的为准（“The Longest-Leftmost”）。因此，POSIX NFA 引擎的速度要慢于传统的 NFA 引擎。

我们日常面对的，一般都是传统的 NFA，所以通常都是最左侧的分支优先，在书写正则的时候务必要注意这一点。

下面是 DFA、传统 NFA 以及 POSIX NFA 引擎的特点总结：



| 引擎类型       | 程序                                                                                             | 忽略优先量词<br>(懒惰) | 捕获型<br>括号 | 回溯     |
|------------|------------------------------------------------------------------------------------------------|----------------|-----------|--------|
| DFA        | Golang、MySQL、awk (大多数版本)、egrep (大多数版本)、flex、lex、Procmail                                       | 不支持            | 不支持       | 不支持    |
| 传统型 NFA    | PCRE library、Perl、PHP、Java、Python、Ruby、grep (大多数版本)、GNU Emacs、less、more、.NET 语言、sed (大多数版本)、vi | 支持             | 支持        | 支持     |
| POSIX NFA  | mawk、Mortice Kern Systems'utilities、GNU Emacs (明确指定时使用)                                        | 不支持            | 不支持       | 支持     |
| DFA/NFA 混合 | GNU awk、GNU grep/egrep、Tcl                                                                     | 支持             | 支持        | DFA 支持 |

## 回溯

回溯是 NFA 引擎才有的，并且只有在正则中出现**量词**或**多选分支结构**时，才可能会发生回溯。

比如我们使用正则 `a+ab` 来匹配文本 `aab` 的时候，过程是这样的，`a+` 是贪婪匹配，会占用掉文本中的两个 `a`，但正则接着又是 `a`，文本部分只剩下 `b`，只能通过回溯，让 `a+` 吐出一个 `a`，再次尝试。

如果正则使用 `. *ab` 去匹配一个比较长的字符串就更糟糕了，因为 `. *` 会吃掉整个字符串（不考虑换行，假设文本中没有换行），然后，你会发现正则中还有 `ab` 没匹配到内容，只能将 `. *` 匹配上的字符串吐出一个字符，再尝试，还不行，再吐出一个，不断尝试。

.**\*ab** The lab assistant was wearing a white overall.

.**\*ab** The lab assistant was wearing a white overall.

.**\*ab** The lab assistant was wearing a white overall.

中间过程省略，一直回溯到I（吐出I之后的所有匹配上的）

.**\*ab** The lab assistant was wearing a white overall.

.**\*ab** The lab assistant was wearing a white overall.

.**\*ab** The lab assistant was wearing a white overall.

所以在工作中，我们要尽量不用 `.*`，除非真的有必要，因为点能匹配的范围太广了，我们要尽可能精确。常见的解决方式有两种，比如要提取引号中的内容时，使用 `"[^"]+"`，或者使用非贪婪的方式 `".+?"`，来减少“匹配上的内容不断吐出，再次尝试”的过程。

我们再回头看一下之前讲解的店铺名匹配示例：



**REGULAR EXPRESSION** v1 ▾
no match, 9021 steps (~4ms)

---

⋮ / 
 

^([A-Za-z0-9.\_()&'\"- ]|  
[aAàÀảẢãÃáÁạẠăĂằẰẳẲẵẴắẮặẶâÂầẦẩỂấẤẫỖ  
ấẤậẬbBcCdDđĐeEèÈẻỄểỄếẾẹẸêÊềỀểỂếẾ  
ẾệỆfFgGhHiIìÌỉỈĩĨíÍịỊjJkKlLmMnNoO  
òÒỏỎõÕóÓọỌôÔồỒổỔốỐộỘơƠờỜởỞỡỠợỢ  
ỚợPqQrRsStTuUùÙủỦữỮứỨ  
ựỰvVwWxXyYỳỠỷỠỹỠýÝỵỠzZ])+\$

/ gm 🚩

---

**TEST STRING**
SWITCH TO UNIT TESTS ▶

this is a cat, cat

从 [🔗 示例](#) 我们可以看到，一个很短的字符串，NFA 引擎尝试步骤达到了 9021 次，由于是贪婪匹配，第一个分支能匹配上 `this is a cat` 部分，接着后面的逗号匹配失败，使用第二个分支匹配，再次失败，此时贪婪匹配部分结束。NFA 引擎接着用正则后面的 `$` 来进行匹配，但此处不是文本结尾，匹配不上，发生回溯，吐出第一个分支匹配上的 `t`，使用第二个分支匹配 `t` 再试，还是匹配不上。







**REGULAR EXPRESSION** v3 ▾ no match, 32 steps (~0ms)

/ ^([A-Za-z0-9.\_()&'\"'- ]|  
[aAàÀảẢãÃáÁạẠăĂằẰẳẲẵỖắẮặẶậẬbBcCdDđĐeEèÈẻỂẽỄéÉẹẸêÊềỀểỂếẾ  
ệỆfFgGhHiIìÌỉỈĩĨíÍịỊjJkKlLmMnNoO  
òÒỏỎỗỠốỐọỌôÔồỒổỔốỔộỘơƠờỜởỞỡỠợỢ  
ơPqQrRsStTuUùÙủỦữỮứỨưƯừỪửỬữỬửỬửỬửỬ)  
vVwWxXyYỳỠỷỠỹỠýỠỵỠzZ])++\$

**TEST STRING**

this is a cat, cat

**SWITCH TO UNIT TESTS ▶**

但要提醒你的是，独占模式“不吐出已匹配字符”的特性，会使得一些场景不能使用它。另外，只有少数编程语言支持独占模式。

解决这个问题还有其它的方式，比如我们可以尝试移除多选分支选择结构，直接用中括号表示多选一（[🔗 示例](#)）。

[illegible]

我们会发现性能也是有显著提升（这里只是测试，真正使用的时候，重复的元素都应该去掉，另外这里也不需要保存子组）。

## 优化建议

学习了原理之后，有助于我们写出更好的正则。我们必须先保证正则的功能是正确的，然后再进行优化性能，下面我给了你一些优化的方法供你参考。

## 1. 测试性能的方法

我们可以使用 `ipython` 来测试正则的性能，`ipython` 是一个 Python shell 增强交互工具，在 macOS/Windows/Linux 上都可以安装使用。在测试正则表达式时，它非常有用，比如下面通过一个示例，来测试在字符串中查找 `abc` 时的时间消耗。

 复制代码

```
1 In [1]: import re
2 In [2]: x = '-' * 10000000 + 'abc'
```



```
3 In [3]: timeit re.search('abc', x)
4
```

另外，你也可以通过前面 regex101.com 查看正则和文本匹配的次数，来得知正则的性能信息。

## 2. 提前编译好正则

编程语言中一般都有“编译”方法，我们可以使用这个方法提前将正则处理好，这样不用在每次使用的时候去反复构造自动机，从而可以提高正则匹配的性能。

```
1 >>> import re
2 >>> reg = re.compile(r'ab?c') # 先编译好，再使用
3 >>> reg.findall('abc')
4 ['abc']
5
6 >>> re.findall(r'ab?c', 'abc') # 正式使用不建议，但测试功能时较方便
7 ['abc']
8
```

[复制代码](#)

## 3. 尽量准确表示匹配范围

比如我们要匹配引号里面的内容，除了写成 “.+?” 之外，我们可以写成 “[^"]+”。使用 “[^”] 要比使用点号好很多，虽然使用的是贪婪模式，但它不会出现点号将引号匹配上，再吐出的问题。

## 4. 提取出公共部分

通过上面对 NFA 引擎的学习，相信你应该明白 (abcd | abxy) 这样的表达式，可以优化成 ab(cd | xy)，因为 NFA 以正则为主导，会导致字符串中的某些部分重复匹配多次，影响效率。

因此我们会知道 th(?:is|at) 要比 this|that 要快一些，但从可读性上看，后者要好一些，这个就需要用的时候去权衡，也可以添加代码注释让代码更容易理解。

类似地，如果是锚点，比如 (^this|^that) is 这样的，锚点部分也应该独立出来，可以写成比如 ^th(is|at) is 的形式，因为锚点部分也是需要尝试去匹配的，匹配次数要尽可

能少。

## 5. 出现可能性大的放左边

由于正则是从左到右看的，把出现概率大的放左边，域名中 .com 的使用是比 .net 多的，所以我们可以写成 `\.(?:com|net)\b`，而不是 `\.(?:net|com)\b`。

## 6. 只在必要时才使用子组

在正则中，括号可以用于归组，但如果某部分后续不会再用，就不需要保存成子组。通常的做法是，在写好正则后，把不需要保存子组的括号中加上 `?:` 来表示只用于归组。如果保存成子组，正则引擎必须做一些额外工作来保存匹配到的内容，因为后面可能会用到，这会降低正则的匹配性能。

## 7. 警惕嵌套的子组重复

如果一个组里面包含重复，接着这个组整体也可以重复，比如 `(.)*` 这个正则，匹配的次数会呈指数级增长，所以尽量不要写这样的正则。

## 8. 避免不同分支重复匹配

在多选分支选择中，要避免不同分支出现相同范围的情况，上面回溯的例子中，我们已经进行了比较详细的讲解。

## 总结

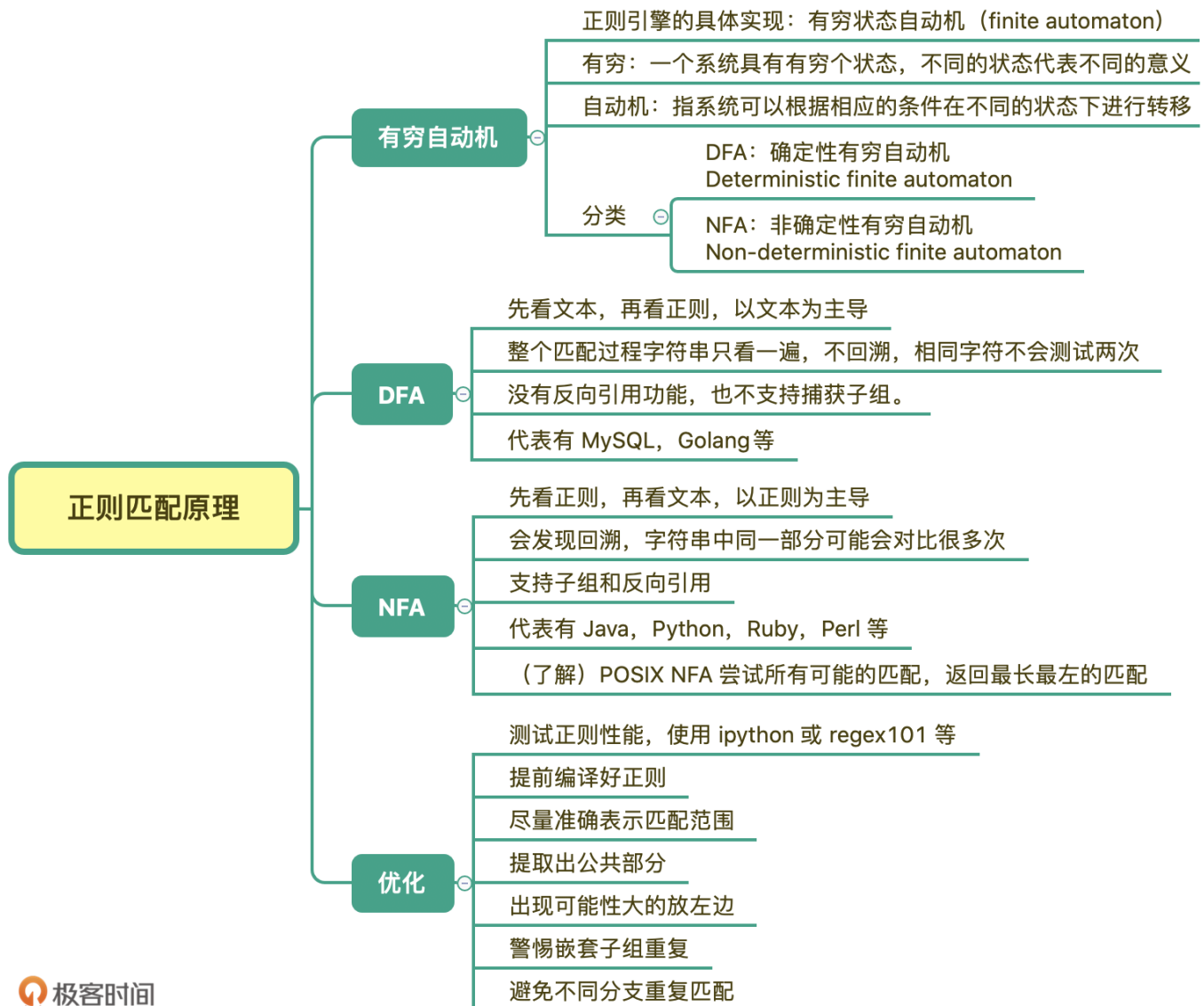
好了，今天的内容讲完了，我来带你总结回顾一下。

今天带你简单学习了有穷自动机的概念，自动机的具体实现称之为正则引擎。

我们学习了正则引擎的匹配原理，NFA 和 DFA 两种引擎的工作方式完全不同，NFA 是以表达式为主导的，先看正则表达式，再看文本。而 DFA 则是以文本为主导的，先看文本，再看正则表达式。POSIX NFA 是指符合 POSIX 标准的 NFA 引擎，它会不断回溯，以确保找到最左侧最长匹配。

接着我们学习了测试正则表达式性能的方法，以及优化的一些方法，比如提前编译好正则，提取出公共部分，尽量准确地表示范围，必要时才使用子组等。

今天所讲的内容总结脑图如下，你可以回顾一下：



## 课后思考

最后，我们来做一个小练习吧。通过今天学习的内容，这里有一个示例，要求匹配“由字母或数字组成的字符串，但第一个字符要是小写英文字母”，你能说一下针对这个示例，NFA 引擎的匹配过程么？

复制代码

- 1 文本：a12
- 2 正则：`^(?=[a-z])[a-z0-9]+$`

好，今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我参与讨论，并把文章分享给你的朋友或者同事，一起交流一下。

提建议

更多课程推荐

# 设计模式之美

前 Google 工程师手把手教你写高质量代码

王争

前 Google 工程师

《数据结构与算法之美》专栏作者



涨价倒计时 🕒

限时秒杀 **¥149**，7月31日涨价至 **¥299**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | 应用3：如何在语言中用正则让文本处理能力上一个台阶？

下一篇 12 | 问题集锦：详解正则常见问题及解决方案

## 精选留言 (7)

写留言



一步

2020-07-08

看了一下这个匹配过程分为几步：

- 1: 拿到正则表达式的 开始符号 ^, 去匹配字符串的开始
- 2: 拿到正则的 (?=[a-z]), 发现是一个环视, 不进行看字符串
- 3: 解析环视中的 表达式为: [a-z], 和下一个字符串进行比较, 发现找到了a符合要求

4: 继续取下一部分的正则为：[a-z0-9]+，和接下来的字符串进行比较，贪婪模式，匹...  
展开 ∨

作者回复: 没什么问题，环视只匹配位置，是零宽度的，区别就在于这儿。

元字符 “^” 和 “\$” 匹配的只是位置，顺序环视 “(?:=[a-z])” 只进行匹配，并不占有字符，也不将匹配的内容保存到最终的匹配结果，所以都是零宽度的。

匹配过程：

首先由元字符 “^” 取得控制权，从位置0开始匹配，“^” 匹配的就是开始位置 “位置0”，匹配成功，控制权交给顺序环视 “(?:=[a-z])”；

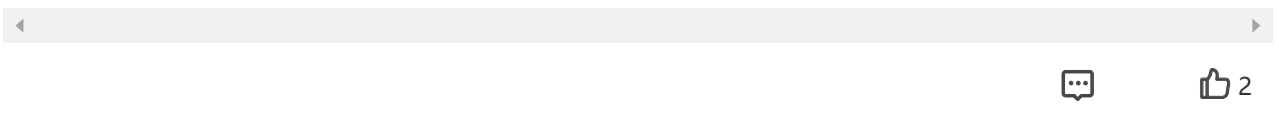
“(?:=[a-z])” 要求它所在位置右侧必须是字母才能匹配成功，零宽度的子表达式之间是不互斥的，即同一个位置可以同时由多个零宽度子表达式匹配，所以它也是从位置0尝试进行匹配，位置0的右侧是字符 “a”，符合要求，匹配成功，控制权交给 “[a-z0-9]+”；

因为 “(?:=[a-z])” 只进行匹配，并不将匹配到的内容保存到最后结果，并且 “(?:=[a-z])” 匹配成功的位置是位置0，所以 “[a-z0-9]+” 也是从位置0开始尝试匹配的，“[a-z0-9]+” 首先尝试匹配 “a”，匹配成功，继续尝试匹配，可以成功匹配接下来的 “1” 和 “2”，此时已经匹配到位置3，位置3的右侧已没有字符，这时会把控制权交给 “\$”；

元字符 “\$” 从位置3开始尝试匹配，它匹配的是结束位置，也就是 “位置3”，匹配成功。

此时正则表达式匹配完成，报告匹配成功。匹配结果为 “a12”，开始位置为0，结束位置为3。其中 “^” 匹配位置0，“(?:=[a-z])” 匹配位置0，“[a-z0-9]+” 匹配字符串 “a12”，“\$” 匹配位置3。

可以参考 <https://blog.csdn.net/lxcnn/article/details/4304651>



Regina

2020-07-21

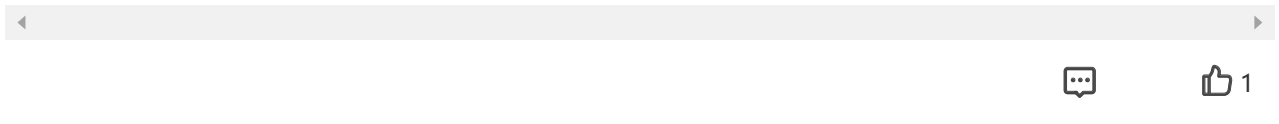
DFN引擎匹配那，为什么是shixi被淘汰而不是shijian  
text: we study on jikeshixi app

^

regex: jike(zhushou|shijian|shixi)...

展开 ∨

作者回复: 感谢指出，这里DFA部分，文本应该是  
we study on jikeshijian app



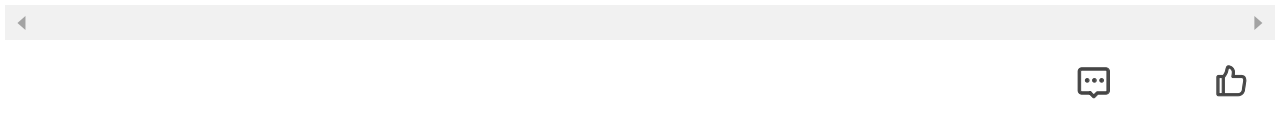
吴小智

2020-07-24

NFA 有  $\epsilon$  的状态转移，但是 DFA 没有。

展开 ∨

作者回复: NFA功能要多一些，复杂一些。  
DFA简单，速度快



简简单单

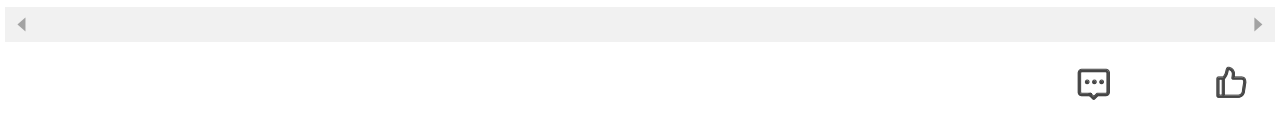
2020-07-22

[^"]：在中括号中表示 非双引号的所有字符吗？

^"：在非中括号中表示 必须是行头，且行头右侧第一个字符必须是个双引号吗？

展开 ∨

作者回复: 对的，理解正确，在正则开头 和 在中括号里面第一个位置是含义不一样



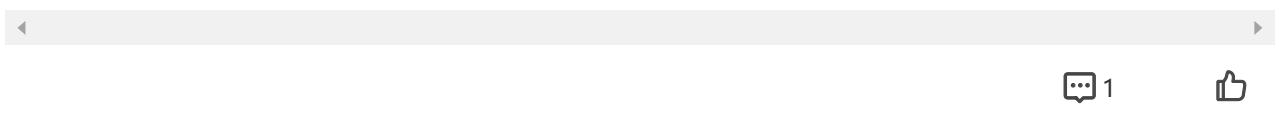
L

2020-07-19

感觉最后要是能一起实现一个小型的正则引擎就好了

展开 ∨

作者回复: 这个记下来了，后面有机会加餐看看



一步

2020-07-08

NFA 通过构造特定扩展，支持子组和反向引用



-----

这里的扩展是什么意思？指什么

展开

作者回复: 你可以理解成NFA可以把匹配到的内容记下来，“扩展”可以理解成做了一些额外的工作。

原理部分最主要的是理解DFA的“文本主导”，NFA“正则主导”以及回溯相关的内容。



**Robot**  
2020-07-08

文本：a12  
正则：^(?=[a-z])[a-z0-9]+\$

- 1、正则^先开始匹配到a12的开始位置
- 2、正则(?=[a-z])正向环视检查,开始位置之后的字符是否是a-z之一，匹配...

展开

作者回复: 对的