初探正则表达式

perfectpan

2019 年 7 月 30 日

前言

- 讲正则表达式的念头

前言

- 讲正则表达式的念头
- 演讲人演讲水平较烂，如果有没听明白的地方请及时打断

正则散讲

正则表达式

~~正则表达式，又称正则表示式、正则表示法、规则表达式、常规表示法，是计算机科学的一个概念。正则表达式使用单个字符串来描述、匹配一系列匹配某个句法规则的字符串。在很多文本编辑器里，正则表达式通常被用来检索、替换那些匹配某个模式的文本。许多程序设计语言都支持利用正则表达式进行字符串操作。~~

正则散讲

- 数据结构

正则散讲

- 数据结构
- js 支持它的子集

正则散讲

- 数据结构
- js 支持它的子集
- js 引擎实现的方式是 NFA

正则表达式语法

- 正则表达式要么匹配字符，要么匹配位置

匹配字符

- 字符组

匹配字符

- 字符组
- $[abc]$ 表示"a" 或"b" 或"c"

匹配字符

- 字符组
- $[abc]$ 表示”a” 或”b” 或”c”
- 范围表示 [a-z],[1-7],[A-Z]

匹配字符

- 字符组
- [*abc*] 表示”a” 或”b” 或”c”
- 范围表示 [a-z],[1-7],[A-Z]
- 系统自带 \d,\D,\w,\W,\s,\S,.

匹配字符

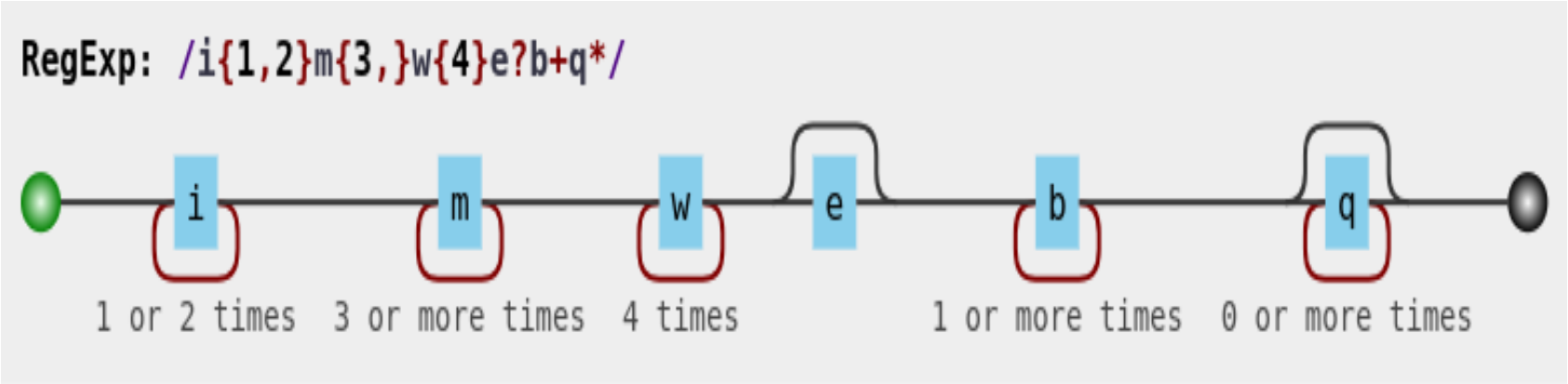
- 量词,{m,n}

匹配字符

- 量词,{m,n}
- /i{1,2}m{3,}w{4}e?b+q*/

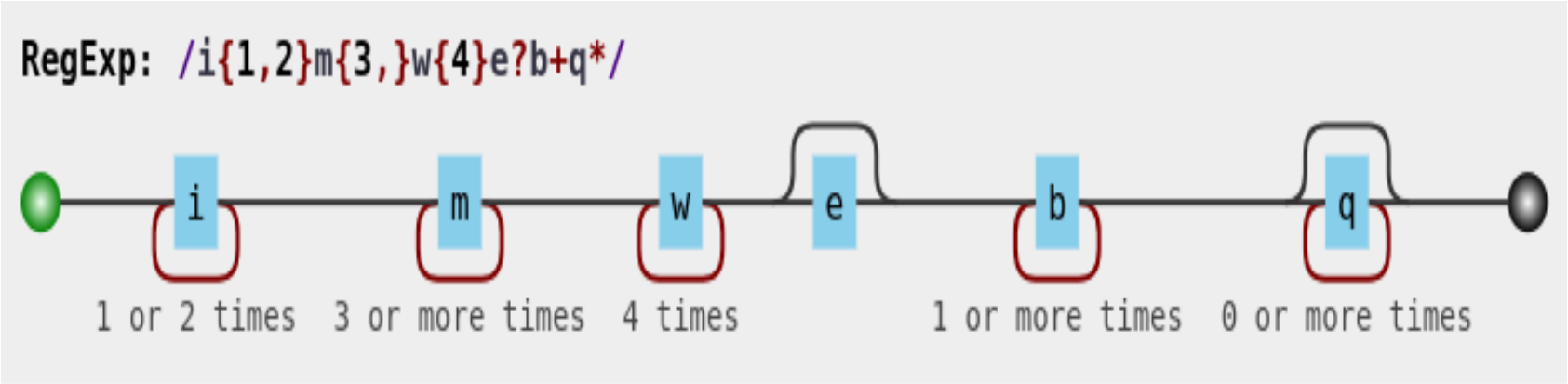
匹配字符

- 量词,{m,n}
- /i{1,2}m{3,}w{4}e?b+q*/



匹配字符

- 量词,{m,n}
- /i{1,2}m{3,}w{4}e?b+q*/




```
> /i{1,2}m{3,}w{4}e?b+q*/.test('immmwwwweb')
< true
> /i{1,2}m{3,}w{4}e?b+q*/.test('immmwwwwb')
< true
> /i{1,2}m{3,}w{4}e?b+q*/.test('immwwwwbq')
< false
```

匹配字符

- 量词默认是贪婪匹配 (贪婪量词)

匹配字符

- 量词默认是贪婪匹配 (贪婪量词)
- 惰性匹配在相应量词后面加一个? 即可

匹配字符

- 量词默认是贪婪匹配 (贪婪量词)
- 惰性匹配在相应量词后面加一个? 即可

```
> /a{1,3}/.exec('aaa')
◂ ▶ ["aaa", index: 0, input: "aaa", groups: undefined]
> /a{1,3}?/.exec('aaa')
◂ ▶ ["a", index: 0, input: "aaa", groups: undefined]
```

匹配字符

- 分支能力

匹配字符

- 分支能力
- 整体能力

匹配字符

- 分支能力
- 整体能力
- 匹配 24 小时制

匹配字符

- 分支能力
- 整体能力
- 匹配 24 小时制
- /([01][0-9]|[2][0-3]):[0-5][0-9]/

```
> '1231204:58999'.match(/([01][0-9]|[2][0-3]):[0-5][0-9]/)
< ▶ (2) ["04:58", "04", index: 5, input: "1231204:58999", groups: undefined]
```

匹配位置

- 位置是什么

匹配位置

- 位置是什么
- 每个字符之间的空白

匹配位置

- 位置是什么
- 每个字符之间的空白
- 可以把 imweb 拆成””+i+””+m+””+w+””+e+””+b+””, 甚至可以把 i 和 m 之间看成多个位置

匹配位置

匹配位置

- 开头 ^

匹配位置

- 开头 ^
- 结尾 $

匹配位置

- 开头 ^
- 结尾 $
- 单词边界 \b \w 和 \W 之间的位置

匹配位置

- 开头 ^
- 结尾 $
- 单词边界 \b \w 和 \W 之间的位置
- 非单词边界 \B

```
> 'imweb nb.jpg'.replace(/\b/g,'-')
< "-imweb- -nb-.-jpg-"
> 'imweb nb.jpg'.replace(/\B/g,'-')
< "i-m-w-e-b n-b.j-p-g"
```

匹配位置

- p 前面的位置 (?=p)

匹配位置

- p 前面的位置 (?=p)
- 不是 p 前面的位置 (?!p)

```
> 'imweb'.replace(/(?=web)/g,'#')
<- "im#web"
> 'imweb'.replace(/(?!web)/g,'#')
<- "#i#mw#e#b#"
```

匹配位置

- 回到刚开始的问题

匹配位置

- 回到刚开始的问题

```
> /(?=im)^im(?=\w)web$\b$/.test('imweb')
< true
```

小练习

- 数字千位分隔符（123456789->123,456,789）

小练习

- 数字千位分隔符（123456789->123,456,789）
- 表单验证密码：密码长度 6-12 位，由数字、小写字符和大写字母组成，但必须至少包括 2 种字符

引用

```
> '1231204:58999'.match(/([01][0-9]|[2][0-3]):[0-5][0-9]/)
<- ▶(2) ["04:58", "04", index: 5, input: "1231204:58999", groups: undefined]
```

- 括号括起来即代表一个分组

引用

```
> '1231204:58999'.match(/([01][0-9]|[2][0-3]):[0-5][0-9]/)
< ▶ (2) ["04:58", "04", index: 5, input: "1231204:58999", groups: undefined]
```

- 括号括起来即代表一个分组
- javascript 里引用分组

引用

```
> '1231204:58999'.match(/([01][0-9]|[2][0-3]):[0-5][0-9]/)
< ▶ (2) ["04:58", "04", index: 5, input: "1231204:58999", groups: undefined]
```

- 括号括起来即代表一个分组
- javascript 里引用分组
- 正则表达式里引用

分组引用

- 每一个括号括起来的分组我们都可以直接引用

分组引用

- 每一个括号括起来的分组我们都可以直接引用
- 数据提取，数据替换

分组引用

- 每一个括号括起来的分组我们都可以直接引用
- 数据提取，数据替换
- 2019-07-25->07/25/2019

```
> var regex = /(\d{4})-(\d{2})-(\d{2})/;
  var string = "2019-07-25";
  var result = string.replace(regex, function () {
      return RegExp.$2 + "/" + RegExp.$3 + "/" + RegExp.$1;
  });
  console.log(result);

  07/25/2019
```

反向引用

- 在正则里引用之前的分组

反向引用

- 在正则里引用之前的分组
- 必要性（需要前后一致的时候）

反向引用

- 在正则里引用之前的分组
- 必要性（需要前后一致的时候）
- 匹配 2019-07-25 或 2019/07/25 或 2019.07.25

反向引用

- 在正则里引用之前的分组
- 必要性（需要前后一致的时候）
- 匹配 2019-07-25 或 2019/07/25 或 2019.07.25
- 简单的想法 /\d{4}(-|\/|\.)\d{2}(-|\/|\.)\d{2}/

反向引用

- 在正则里引用之前的分组
- 必要性（需要前后一致的时候）
- 匹配 2019-07-25 或 2019/07/25 或 2019.07.25
- 简单的想法 /\d{4}(-|\/|\.)\d{2}(-|\/|\.)\d{2}/

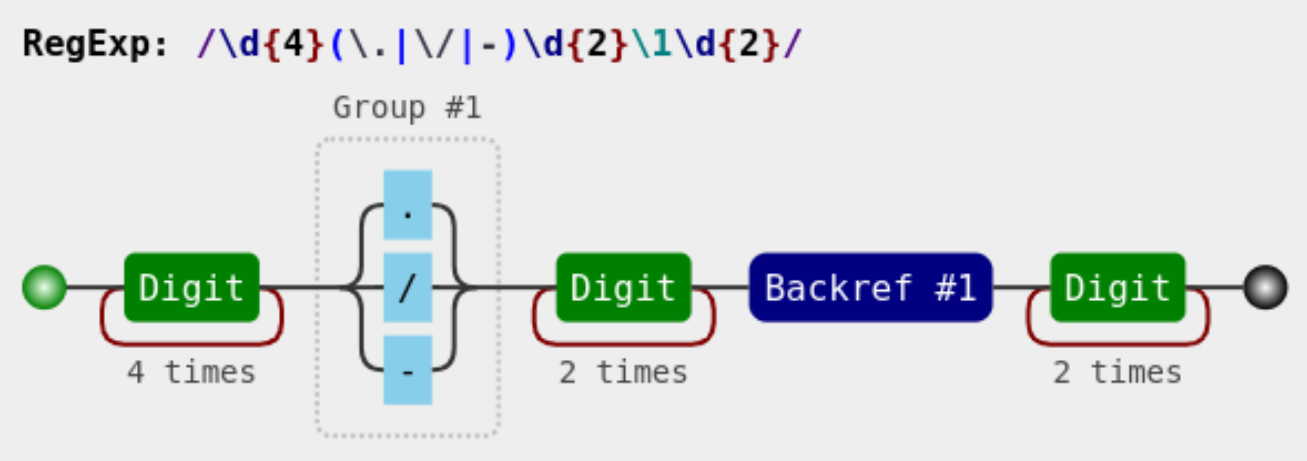
```
> /\d{4}(-|\/|\.)\d{2}(-|\/|\.)\d{2}/.test('2019-07/25')
< true
```

反向引用

- /\d{4}(-|\/|\.)\d{2}\1\d{2}/

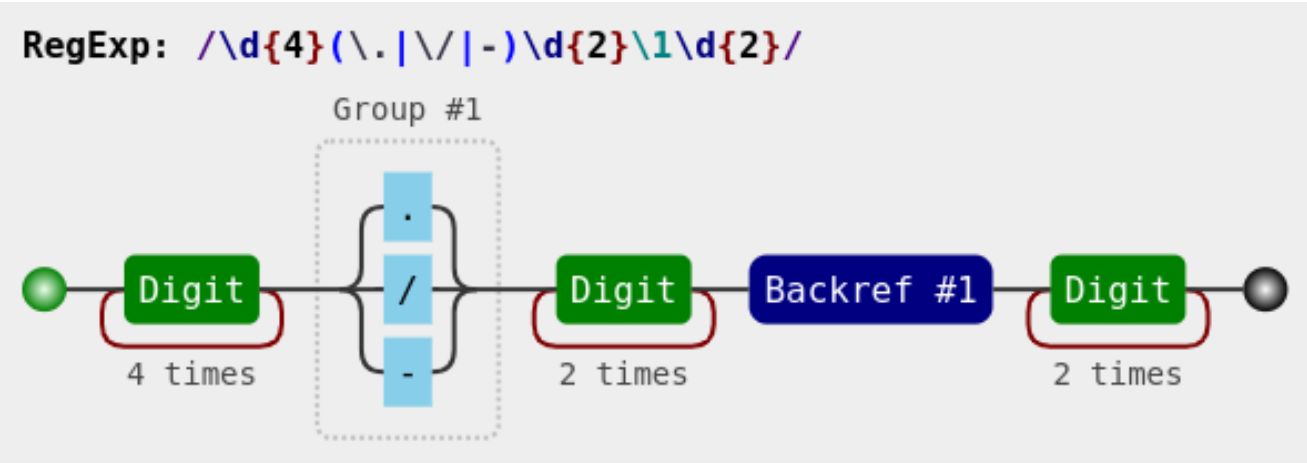
反向引用

- /\d{4}(-|\/|\.)\d{2}\1\d{2}/



反向引用

- /\d{4}(-|\/|\.)\d{2}\1\d{2}/




```
> /\d{4}(-|\/|\.)\d{2}\1\d{2}/.test('2019-07/25')
< false
```

反向引用

- 可以括号括起来但不引用吗

反向引用

- 可以括号括起来但不引用吗
- 非捕获括号 (?:p)

小总结

- 正则表达式要么匹配字符要么匹配位置

小总结

- 正则表达式要么匹配字符要么匹配位置
- 括号的捕获能力扩展了正则的可能性

Javascript 正则匹配原理

回溯法

简而言之就是暴力搜索，搜到一个状态发现无法再走通或者说不满足条件时就退一步往其他分支的状态进行搜索，复杂度感人

Javascript 正则匹配原理

回溯法

简而言之就是暴力搜索，搜到一个状态发现无法再走通或者说不满足条件时就退一步往其他分支的状态进行搜索，复杂度感人

看下面这个正则表达式

Javascript 正则匹配原理

回溯法

简而言之就是暴力搜索，搜到一个状态发现无法再走通或者说不满足条件时就退一步往其他分支的状态进行搜索，复杂度感人

看下面这个正则表达式

- /^\d{1,3}?\d{1,3}$/

Javascript 正则匹配原理

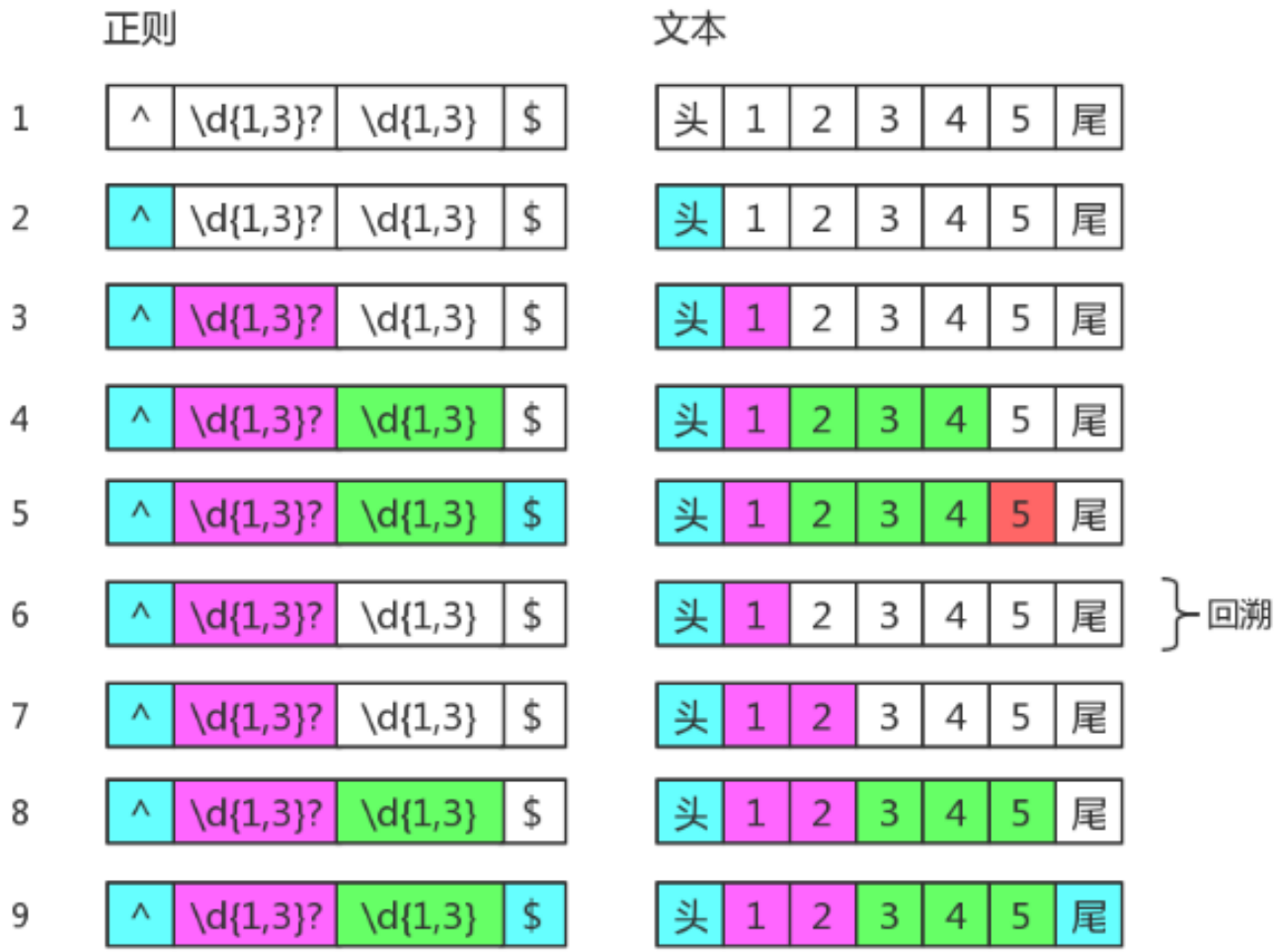
回溯法

简而言之就是暴力搜索，搜到一个状态发现无法再走通或者说不满足条件时就退一步往其他分支的状态进行搜索，复杂度感人

看下面这个正则表达式

- /^\d{1,3}?\d{1,3}$/
- 匹配"12345" 的过程是怎么样的

Javascript 正则匹配原理



Javascript 正则匹配原理

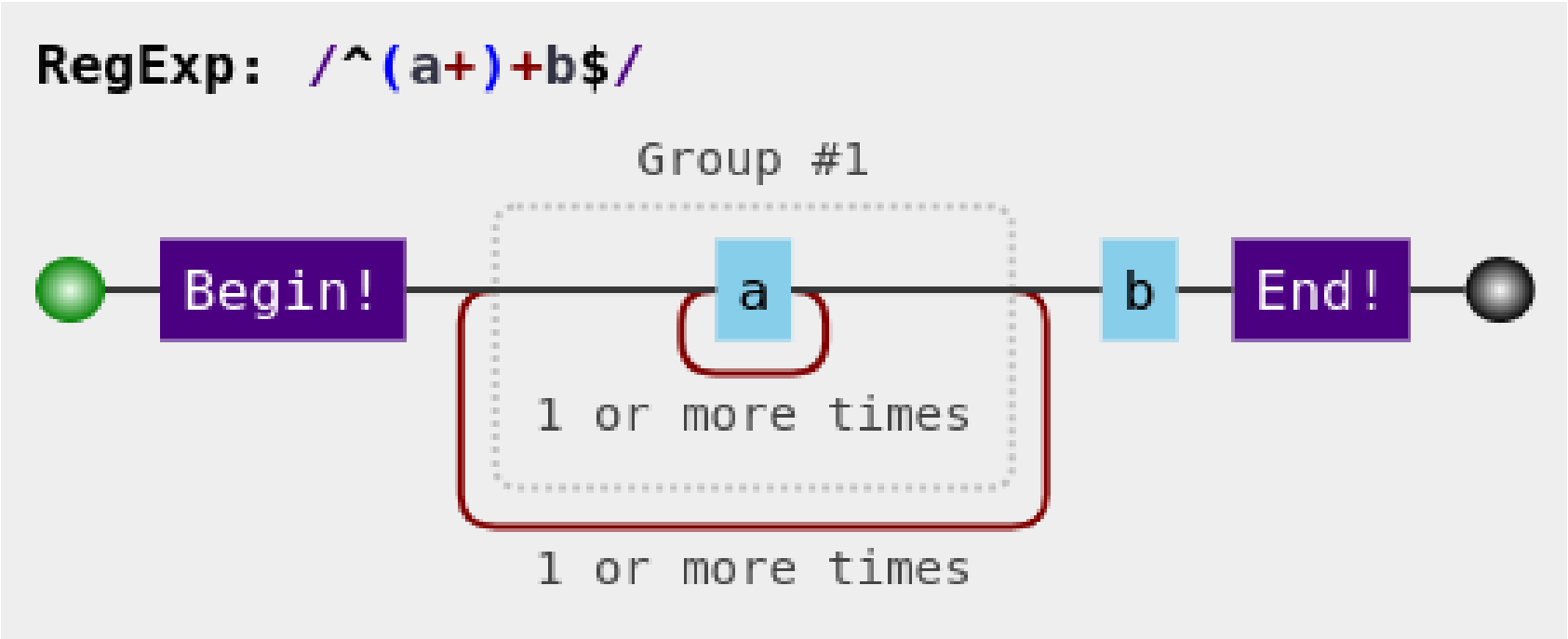
- 分析完以后你可能没什么感觉

Javascript 正则匹配原理

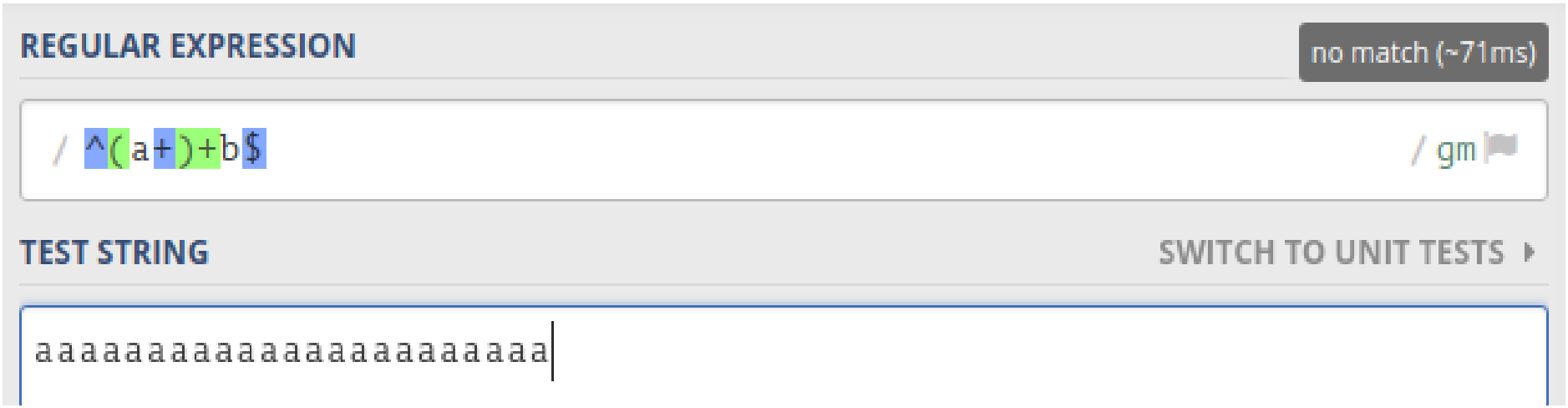
- 分析完以后你可能没什么感觉
- 但回溯的实现方式是会使一些写的有问题的正则表达式在匹配的过程中被卡到指数级的时间，具体我们来看一个例子

Javascript 正则匹配原理

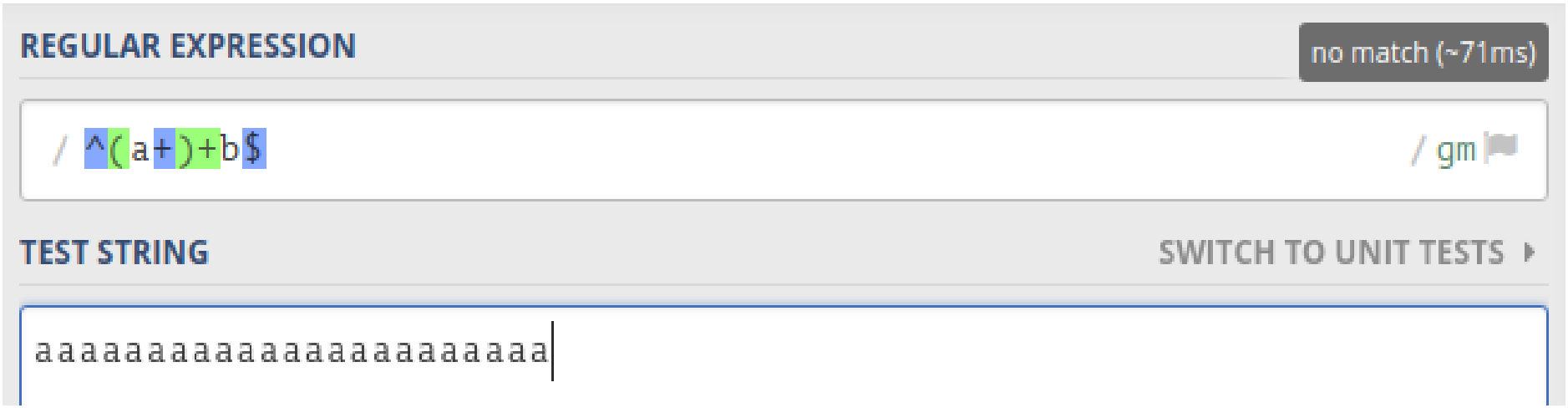
- 分析完以后你可能没什么感觉
- 但回溯的实现方式是会使一些写的有问题的正则表达式在匹配的过程中被卡到指数级的时间，具体我们来看一个例子

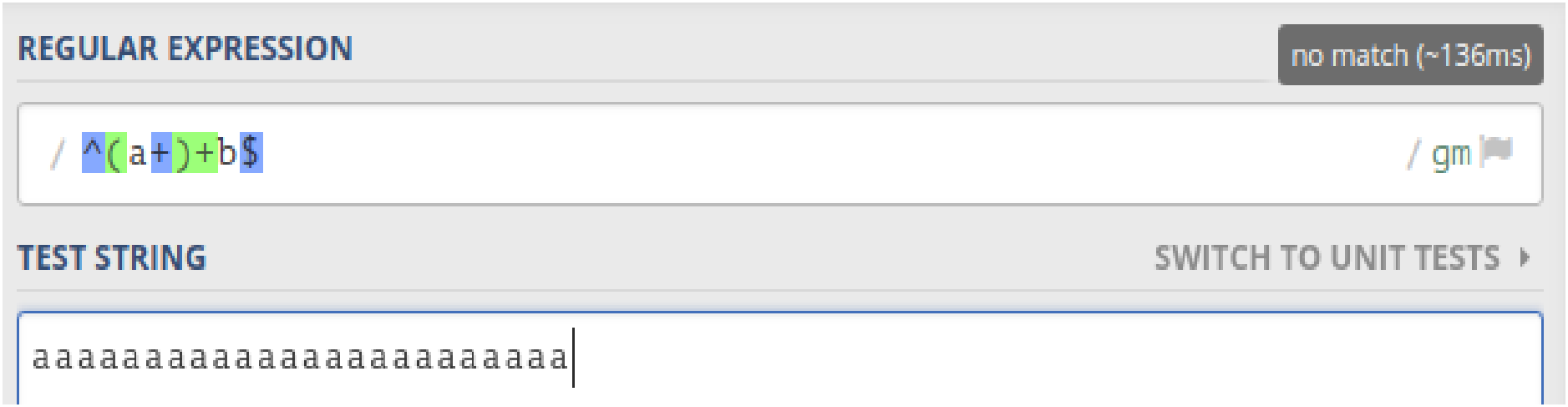


Javascript 正则匹配原理

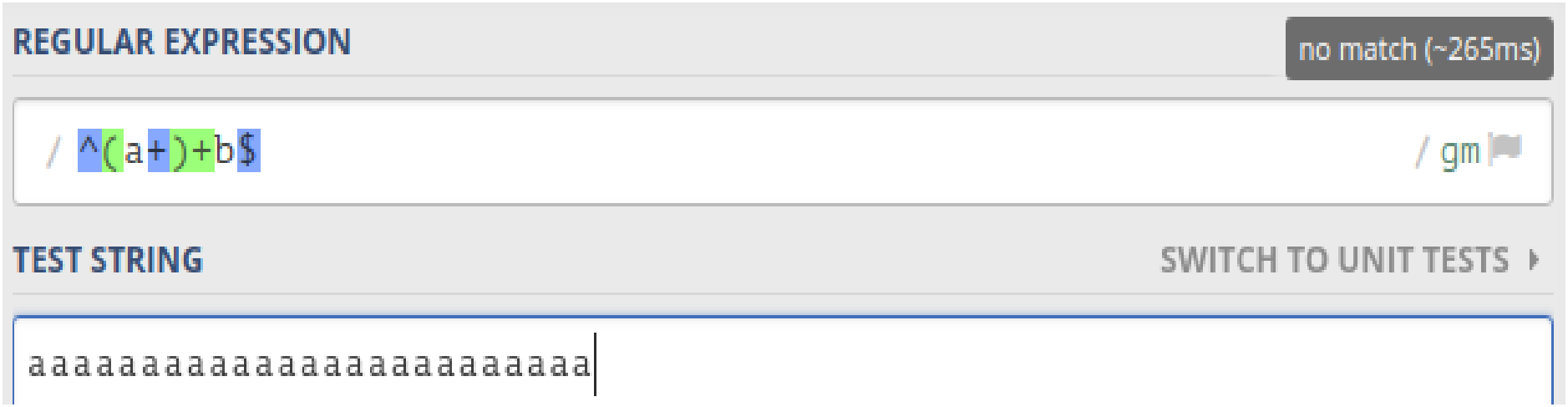


Javascript 正则匹配原理





Javascript 正则匹配原理



Javascript 正则匹配原理

Javascript 正则匹配原理

- 正则表达式：/^(a+)+b$/

Javascript 正则匹配原理

- **正则表达式**：/∧(a+)+b$/
- $(a+)+$ **可以看成无限多个** $(a+)$ **相连**

Javascript 正则匹配原理

- 正则表达式：/^(a+)+b$/
- $(a+)+$ 可以看成无限多个 $(a+)$ 相连
- 在一个长度为 n 的全是 a 的字符串中匹配的时候等价于把一个长度为 n 的线段分成若干段长度大于等于 1 的小线段

Javascript 正则匹配原理

- 正则表达式：/^(a+)+b$/
- $(a+)+$ 可以看成无限多个 $(a+)$ 相连
- 在一个长度为 n 的全是 a 的字符串中匹配的时候等价于把一个长度为 n 的线段分成若干段长度大于等于 1 的小线段
- 其他长度小于 n 的字符串会在上面这种情况回溯的时候匹配到，所以只考虑上面的种类数

Javascript 正则匹配原理

- 正则表达式：/^(a+)+b$/
- $(a+)+$ 可以看成无限多个 $(a+)$ 相连
- 在一个长度为 n 的全是 a 的字符串中匹配的时候等价于把一个长度为 n 的线段分成若干段长度大于等于 1 的小线段
- 其他长度小于 n 的字符串会在上面这种情况回溯的时候匹配到，所以只考虑上面的种类数
- 枚举分成多少段，从 1 到 n

Javascript 正则匹配原理

- 正则表达式：/^(a+)+b$/
- $(a+)+$ 可以看成无限多个 $(a+)$ 相连
- 在一个长度为 n 的全是 a 的字符串中匹配的时候等价于把一个长度为 n 的线段分成若干段长度大于等于 1 的小线段
- 其他长度小于 n 的字符串会在上面这种情况回溯的时候匹配到，所以只考虑上面的种类数
- 枚举分成多少段，从 1 到 n
- $\sum_{i=0}^{n-1} \binom{n-1}{i} = 2^{n-1}$

Javascript 正则匹配原理

- 正则表达式：/^(a+)+b$/
- $(a+)+$ 可以看成无限多个 $(a+)$ 相连
- 在一个长度为 n 的全是 a 的字符串中匹配的时候等价于把一个长度为 n 的线段分成若干段长度大于等于 1 的小线段
- 其他长度小于 n 的字符串会在上面这种情况回溯的时候匹配到，所以只考虑上面的种类数
- 枚举分成多少段，从 1 到 n
- $\sum_{i=0}^{n-1} \binom{n-1}{i} = 2^{n-1}$
- 时间复杂度 $O(2^n)$

Javascript 正则匹配原理

- 现在已经知道了回溯算法导致糟糕的时间复杂度，所以我们要尽可能规避这类问题，那么有哪些场景会出现回溯

Javascript 正则匹配原理

- 现在已经知道了回溯算法导致糟糕的时间复杂度，所以我们要尽可能规避这类问题，那么有哪些场景会出现回溯
- 贪婪量词：先贪心的匹配最多的然后匹配失败了再退一步继续匹配

Javascript 正则匹配原理

- 现在已经知道了回溯算法导致糟糕的时间复杂度，所以我们要尽可能规避这类问题，那么有哪些场景会出现回溯
- 贪婪量词：先贪心的匹配最多的然后匹配失败了再退一步继续匹配
- 惰性量词（上面的例子）

Javascript 正则匹配原理

- 现在已经知道了回溯算法导致糟糕的时间复杂度，所以我们要尽可能规避这类问题，那么有哪些场景会出现回溯
- 贪婪量词：先贪心的匹配最多的然后匹配失败了再退一步继续匹配
- 惰性量词（上面的例子）
- 分支结构：第一个分支匹配失败了就回溯匹配第二个分支

Javascript 正则匹配原理

- 你可能会有疑问，为什么要采用这种坑爹方式

Javascript 正则匹配原理

- 你可能会有疑问，为什么要采用这种坑爹方式
- NFA 与 DFA

Javascript 正则匹配原理

- ~~复习时间~~

Javascript 正则匹配原理

- ~~复习时间~~
- 一个有限状态自动机是一个五元组：状态集合 Q、字母表 $\sum$、转移函数 δ、开始状态 q_0、接收状态集合 F

Javascript 正则匹配原理

- ~~复习时间~~
- 一个有限状态自动机是一个五元组：状态集合 Q、字母表 $\sum$、转移函数 δ、开始状态 q_0、接收状态集合 F
- 确定性有限状态自动机 (DFA)：开始状态唯一，每个状态对于同个字符只会转移到一个唯一的后继节点

Javascript 正则匹配原理

- ~~复习时间~~
- 一个有限状态自动机是一个五元组：状态集合 Q、字母表 $\sum$、转移函数 δ、开始状态 q_0、接收状态集合 F
- 确定性有限状态自动机 (DFA)：开始状态唯一，每个状态对于同个字符只会转移到一个唯一的后继节点
- 非确定性有限状态自动机 (NFA)：开始状态是一个集合，每个状态对于同个字符可以转移到多个后继节点，另外还存在 ϵ 边（一个状态不读入字符就可以跳转到另一个状态）

Javascript 正则匹配原理

- ~~复习时间~~
- 一个有限状态自动机是一个五元组：状态集合 Q、字母表 $\sum$、转移函数 δ、开始状态 q_0、接收状态集合 F
- 确定性有限状态自动机 (DFA)：开始状态唯一，每个状态对于同个字符只会转移到一个唯一的后继节点
- 非确定性有限状态自动机 (NFA)：开始状态是一个集合，每个状态对于同个字符可以转移到多个后继节点，另外还存在 ϵ 边（一个状态不读入字符就可以跳转到另一个状态）
- NFA 可以转化成 DFA

Javascript 正则匹配原理

- NFA

Javascript 正则匹配原理

- NFA
- 空间复杂度（即状态数和转移数）是 $O(m)$

Javascript 正则匹配原理

- NFA
- 空间复杂度（即状态数和转移数）是 $O(m)$
- 匹配的时间复杂度最坏情况下是指数级的，即 $O(2^n)$

Javascript 正则匹配原理

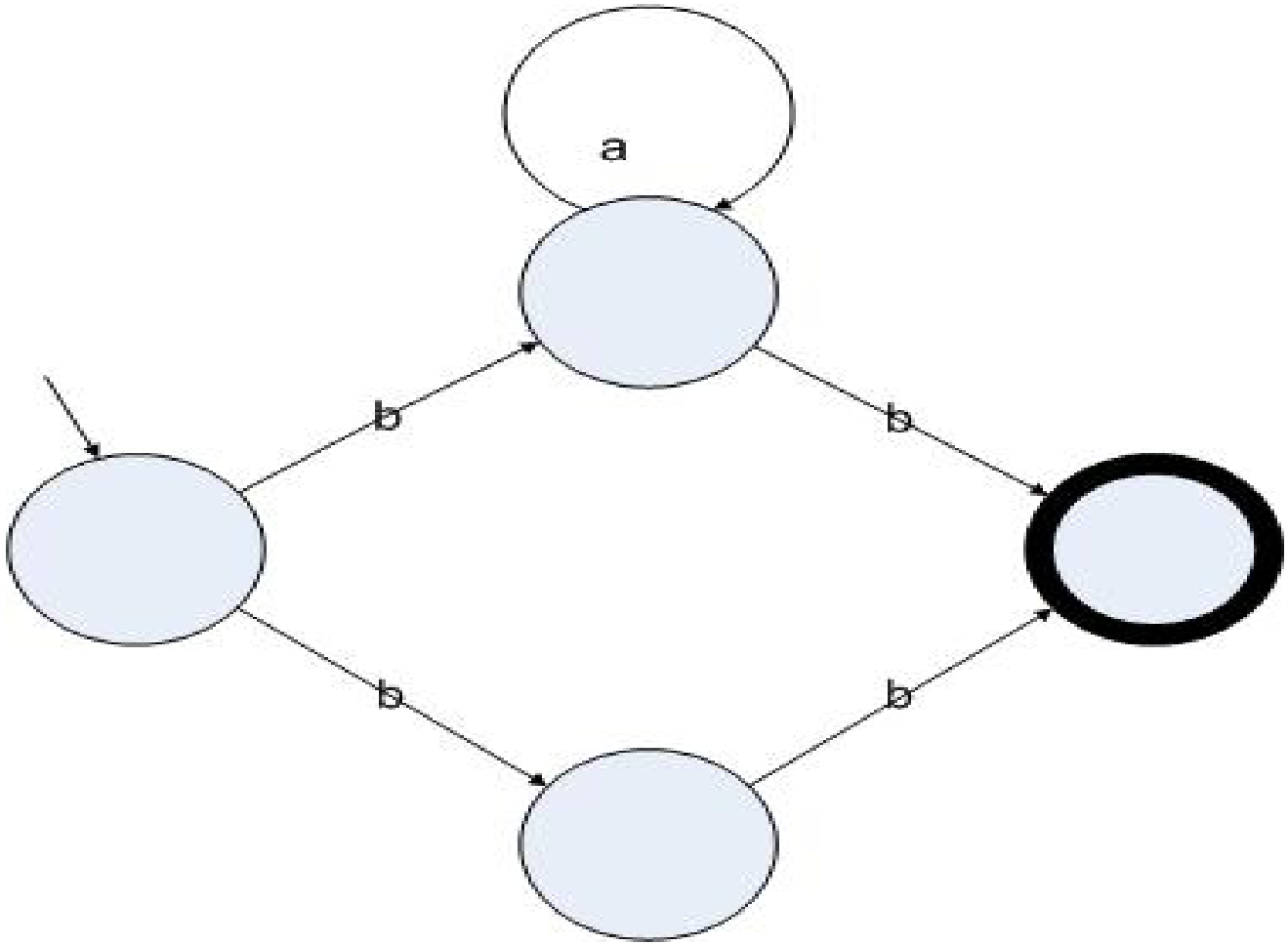


图: /ba*b/

Javascript 正则匹配原理

- DFA

Javascript 正则匹配原理

- DFA
- 匹配的时间复杂度是线性的，即 $O(n)$

Javascript 正则匹配原理

- DFA
- 匹配的时间复杂度是线性的，即 $O(n)$
- 最坏情况下，转化需要的空间复杂度和时间复杂度是 $O(2^m)$

Javascript 正则匹配原理

- DFA
- 匹配的时间复杂度是线性的，即 $O(n)$
- 最坏情况下，转化需要的空间复杂度和时间复杂度是 $O(2^m)$
- 空间换时间

Javascript 正则匹配原理

- DFA
- 匹配的时间复杂度是线性的，即 $O(n)$
- 最坏情况下，转化需要的空间复杂度和时间复杂度是 $O(2^m)$
- 空间换时间
- 功能受限，分组捕获

Javascript 正则匹配原理

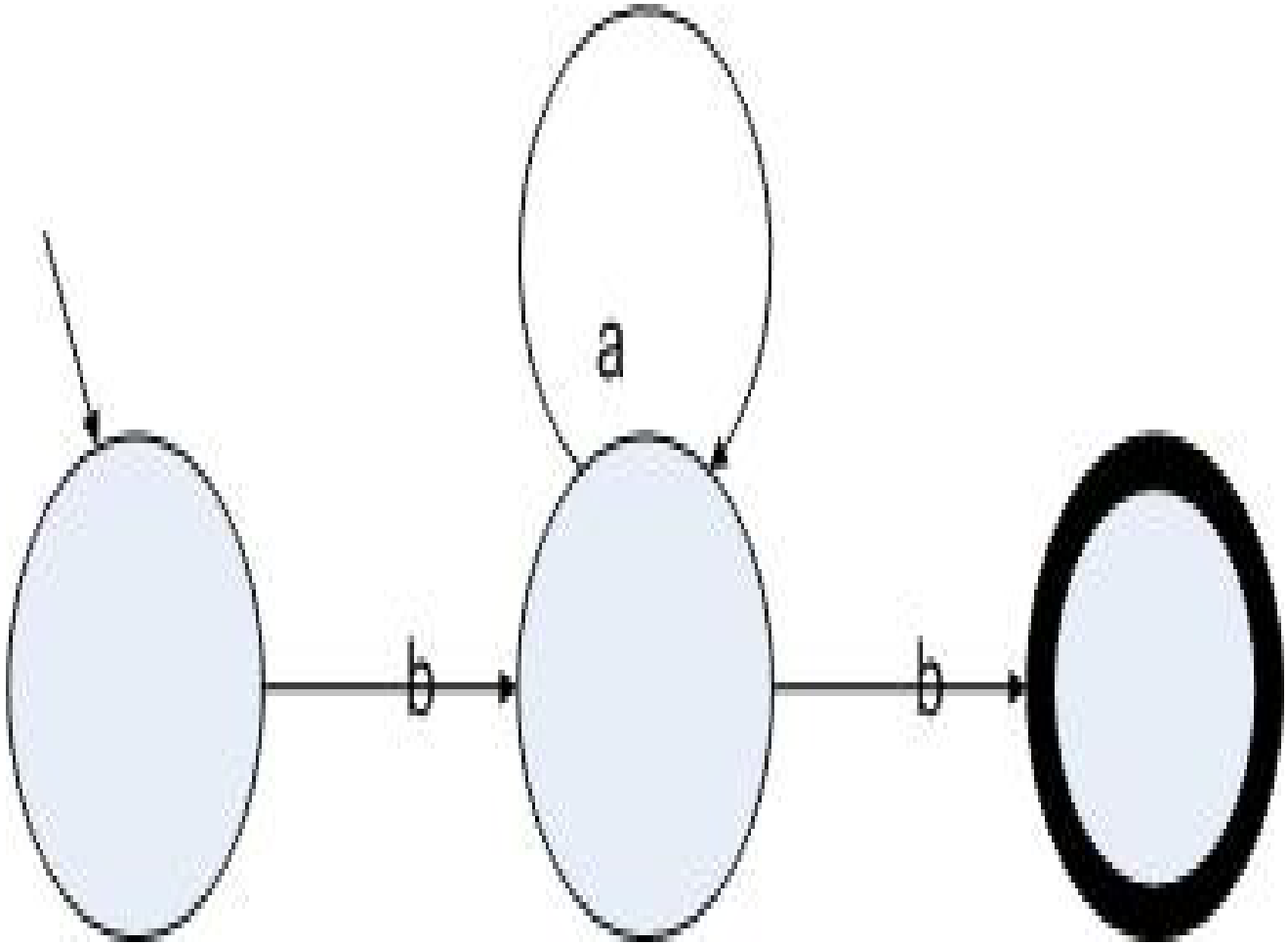


图: /ba*b/

Javascript 正则匹配原理

- 数据结构在特殊化的情况下，通常复杂度，代码难度，常数都会变优，但是可以实现的功能会变少

Javascript 正则匹配原理

- 数据结构在特殊化的情况下，通常复杂度，代码难度，常数都会变优，但是可以实现的功能会变少
- 链表与数组

Javascript 正则匹配原理

- 数据结构在特殊化的情况下，通常复杂度，代码难度，常数都会变优，但是可以实现的功能会变少
- 链表与数组
- 块状链表

实现一个简单的 NFA

- 正则表达式转为语法树

实现一个简单的 NFA

- 正则表达式转为语法树
- 遍历语法树生成 NFA

实现一个简单的 NFA

- 正则表达式转为语法树
- 遍历语法树生成 NFA
- 匹配的时候搜索回溯匹配即可

总结

- 正则表达式是一种匹配模式

总结

- 正则表达式是一种匹配模式
- 正则表达式很强大，但强大带来的就是性能的下降，回溯的匹配方式决定了它会在一些场景下产生爆炸的匹配复杂度

总结

- 正则表达式是一种匹配模式
- 正则表达式很强大，但强大带来的就是性能的下降，回溯的匹配方式决定了它会在一些场景下产生爆炸的匹配复杂度
- 一些可以继续探讨的东西：如何去看一个复杂的正则表达式、符号优先级顺序是怎样的、如何自己写出一个好的正则表达式、如何测试自己写的正则表达式是否优秀（最优化问题）

Q&A