编写一个正则表达式的模式

一个正则表达式模式是由简单的字符所构成的，比如 `/abc/`；或者是简单和特殊字符的组合，比如 `/ab*c/` 或 `/Chapter (\d+)\.\d*/`。最后的例子中用到了括号，它在正则表达式中常用作记忆设备。即这部分所匹配的字符将会被记住以备后续使用，例如使用括号的子字符串匹配。

使用简单模式

简单模式是由你想直接找到的字符构成。比如，`/abc/` 这个模式就能且仅能匹配 "abc" 字符按照顺序同时出现的情况。例如在 "Hi, do you know your abc's?" 和 "The latest airplane designs evolved from slabcraft." 中会匹配成功。在上述两个例子中，匹配的子字符串是 "abc"。但是在 "Grab crab" 中会匹配失败，因为它虽然包含子字符串 "ab c"，但并不是准确的 "abc"。

使用特殊字符

当你需要匹配一个不确定的字符串时，比如寻找一个或多个 "b"，或者寻找空格，可以在模式中使用特殊字符。比如，你可以使用 `/ab*c/` 去匹配一个单独的 "a" 后面跟了零个或者多个 "b"，同时后面跟着 "c" 的字符串：`*`的意思是前一项出现零次或者多次。在字符串 "cbbabbbbcdebc" 中，这个模式匹配了子字符串 "abbbbc"。

下面的页面与表格列出了一个正则表达式中可以利用的特殊字符的完整列表和描述。

断言（Assertions）

表示一个匹配在某些条件下发生。断言包含先行断言、后行断言和条件表达式。

边界（Boundaries）

表示行和单词的开始和结尾。

字符类别（Character Classes）

区分不同类型的字符，例如区分字母和数字。

组和范围（Groups and Ranges）

表示表达式字符的分组和范围。

量词（Quantifiers）

表示匹配的字符或表达式的数量。

Unicode 属性转义（Unicode Property Escapes）

基于 unicode 字符属性区分字符。例如大写和小写字母、数学符号和标点。

正则表达式中的特殊字符

字符	含义
`\`	依照下列规则匹配： 在非特殊字符之前的反斜杠表示下一个字符是特殊字符，不能按照字面理解。例如，前面没有 "\" 的 "b" 通常匹配小写字母 "b"，即字符会被作为字面理解，无论它出现在哪里。但如果前面加了 "\"，它将不再匹配任何字符，而是表示一个字符边界。 在特殊字符之前的反斜杠表示下一个字符不是特殊字符，应该按照字面理解。详情请参阅下文中的 "转义（Escaping）" 部分。 如果你想将字符串传递给 RegExp 构造函数，不要忘记在字符串字面量中反斜杠是转义字符。所以为了在模式中添加一个反斜杠，你需要在字符串字面量中转义它。`/[a-z]\s/i` 和 `new RegExp("[a-z]\\s", "i")` 创建了相同的正则表达式：一个用于搜索后面紧跟着空白字符（`\s` 可看后文）并且在 a-z 范围内的任意字符的表达式。为了通过字符串字面量给 RegExp 构造函数创建包含反斜杠的表达式，你需要在字符串级别和正则表达式级别都对它进行转义。例如 `/[a-z]:\\/i` 和 `new RegExp("[a-z]:\\\\","i")` 会创建相同的表达式，即匹配类似 "C:\" 字符串。
`^`	匹配输入的开始。如果多行标志被设置为 true，那么也匹配换行符后紧跟的位置。 例如，`/^A/` 并不会匹配 "an A" 中的 'A'，但是会匹配 "An E" 中的 'A'。 当 '`^`' 作为第一个字符出现在一个字符集合模式时，它将会有不同的含义。反向字符集合 一节有详细介绍和示例。
`$`	匹配输入的结束。如果多行标志被设置为 true，那么也匹配换行符前的位置。 例如，`/t$/` 并不会匹配 "eater" 中的 't'，但是会匹配 "eat" 中的 't'。
`*`	匹配前一个表达式 0 次或多次。等价于 `{0,}`。 例如，`/bo*/` 会匹配 "A ghost booooooed" 中的 'booooo' 和 "A bird warbled" 中的 'b'，但是在 "A goat grunted" 中不会匹配任何内容。

字符	含义
`+`	匹配前面一个表达式 1 次或者多次。等价于 `{1,}`。 例如，`/a+/` 会匹配 "candy" 中的 'a' 和 "caaaaaaandy" 中所有的 'a'，但是在 "cndy" 中不会匹配任何内容。
`?`	匹配前面一个表达式 0 次或者 1 次。等价于 `{0,1}`。 例如，`/e?le?/` 匹配 "angel" 中的 'el'、"angle" 中的 'le' 以及 "oslo" 中的 'l'。 如果**紧跟在任何量词 *、 +、? 或 {} 的后面**，将会使量词变为**非贪婪**（匹配尽量少的字符），和缺省使用的**贪婪模式**（匹配尽可能多的字符）正好相反。例如，对 "123abc" 使用 `/\d+/` 将会匹配 "123"，而使用 `/\d+?/` 则只会匹配到 "1"。 还用于先行断言中，如本表的 `x(?=y)` 和 `x(?!y)` 条目所述。
`.`	（小数点）默认匹配除换行符之外的任何单个字符。 例如，`/.n/` 将会匹配 "nay, an apple is on the tree" 中的 'an' 和 'on'，但是不会匹配 'nay'。 如果 `s` ("dotAll") 标志位被设为 true，它也会匹配换行符。
`(x)`	像下面的例子展示的那样，它会匹配 'x' 并且记住匹配项。其中括号被称为*捕获括号*。 模式 `/(foo) (bar) \1 \2/` 中的 '`(foo)`' 和 '`(bar)`' 匹配并记住字符串 "foo bar foo bar" 中前两个单词。模式中的 `\1` 和 `\2` 表示第一个和第二个被捕获括号匹配的子字符串，即 `foo` 和 `bar`，匹配了原字符串中的后两个单词。注意 `\1`、`\2`、...、`\n` 是用在正则表达式的匹配环节，详情可以参阅后文的 \n 条目。而在正则表达式的替换环节，则要使用像 `$1`、`$2`、...、`$n` 这样的语法，例如，`'bar foo'.replace(/(...) (...)/, '$2 $1')`。`$&` 表示整个用于匹配的原字符串。
`(?:x)`	匹配 'x' 但是不记住匹配项。这种括号叫作*非捕获括号*，使得你能够定义与正则表达式运算符一起使用的子表达式。看看这个例子 `/(?:foo){1,2}/`。如果表达式是 `/foo{1,2}/`，`{1,2}` 将只应用于 'foo' 的最后一个字符 'o'。如果使用非捕获括号，则 `{1,2}` 会应用于整个 'foo' 单词。更多信息，可以参阅下文的 Using parentheses 条目.

字符	含义
`x(?=y)`	匹配'x'仅仅当'x'后面跟着'y'.这种叫做先行断言。 例如，/Jack(?=Sprat)/会匹配到'Jack'仅当它后面跟着'Sprat'。/Jack(?=Sprat|Frost)/匹配‘Jack’仅当它后面跟着'Sprat'或者是‘Frost’。但是‘Sprat’和‘Frost’都不是匹配结果的一部分。
`(?<=y)x`	匹配'x'仅当'x'前面是'y'.这种叫做后行断言。 例如，/(?<=Jack)Sprat/会匹配到' Sprat '仅仅当它前面是' Jack '。/(?<=Jack|Tom)Sprat/匹配‘ Sprat ’仅仅当它前面是'Jack'或者是‘Tom’。但是‘Jack’和‘Tom’都不是匹配结果的一部分。
`x(?!y)`	仅仅当'x'后面不跟着'y'时匹配'x'，这被称为正向否定查找。 例如，仅仅当这个数字后面没有跟小数点的时候，/\d+(?!\.)/ 匹配一个数字。正则表达式/\d+(?!\.)/.exec("3.141")匹配‘141’而不是‘3.141’
`(?<!y)x`	仅仅当'x'前面不是'y'时匹配'x'，这被称为反向否定查找。 例如, 仅仅当这个数字前面没有负号的时候，`/(?<!-)\d+/` 匹配一个数字。 `/(?<!-)\d+/.exec('3')` 匹配到 "3". `/(?<!-)\d+/.exec('-3')` 因为这个数字前有负号，所以没有匹配到。
`x|y`	匹配‘x’或者‘y’。 例如，/green|red/匹配“green apple”中的‘green’和“red apple”中的‘red’
`{n}`	n 是一个正整数，匹配了前面一个字符刚好出现了 n 次。 比如， /a{2}/ 不会匹配“candy”中的'a',但是会匹配“caandy”中所有的 a，以及“caaandy”中的前两个'a'。
`{n,}`	n是一个正整数，匹配前一个字符至少出现了n次。 例如, /a{2,}/ 匹配 "aa", "aaaa" 和 "aaaaa" 但是不匹配 "a"。

字符	含义
`{n,m}`	n 和 m 都是整数。匹配前面的字符至少n次，最多m次。如果 n 或者 m 的值是0， 这个值被忽略。 例如，/a{1, 3}/ 并不匹配“cndy”中的任意字符，匹配“candy”中的a，匹配“caandy”中的前两个a，也匹配“caaaaaaandy”中的前三个a。注意，当匹配”caaaaaaandy“时，匹配的值是“aaa”，即使原始的字符串中有更多的a。
`[xyz]`	一个字符集合。匹配方括号中的任意字符，包括转义序列。你可以使用破折号（-）来指定一个字符范围。对于点（.）和星号（*）这样的特殊符号在一个字符集中没有特殊的意义。他们不必进行转义，不过转义也是起作用的。 例如，[abcd] 和[a-d]是一样的。他们都匹配"brisket"中的‘b’,也都匹配“city”中的‘c’。/[a-z.]+/ 和/[\w.]+/与字符串“test.i.ng”匹配。
`[^xyz]`	一个反向字符集。也就是说， 它匹配任何没有包含在方括号中的字符。你可以使用破折号（-）来指定一个字符范围。任何普通字符在这里都是起作用的。 例如，[^abc] 和 [^a-c] 是一样的。他们匹配"brisket"中的‘r’，也匹配“chop”中的‘h’。
`[\b]`	匹配一个退格(U+0008)。（不要和\b混淆了。）
`\b`	匹配一个词的边界。一个词的边界就是一个词不被另外一个“字”字符跟随的位置或者前面跟其他“字”字符的位置，例如在字母和空格之间。注意，匹配中不包括匹配的字边界。换句话说，一个匹配的词的边界的内容的长度是0。（不要和[\b]混淆了） 使用"moon"举例: /\bm/匹配“moon”中的‘m’; /oo\b/并不匹配"moon"中的'oo'，因为'oo'被一个“字”字符'n'紧跟着。 /oon\b/匹配"moon"中的'oon'，因为'oon'是这个字符串的结束部分。这样他没有被一个“字”字符紧跟着。 /\w\b\w/将不能匹配任何字符串，因为在一个单词中间的字符永远也不可能同时满足没有“字”字符跟随和有“字”字符跟随两种情况。 **注意:** JavaScript的正则表达式引擎将特定的字符集定义为“字”字符。不在该集合中的任何字符都被认为是一个断词。这组字符相当有限：它只包括大写和小写的罗马字母，十进制数字和下划线字符。不幸的是，重要的字符，例如“é”或“ü”，被视为断词。

字符	含义
`\B`	匹配一个非单词边界。匹配如下几种情况： • 字符串第一个字符为非“字”字符 • 字符串最后一个字符为非“字”字符 • 两个单词字符之间 • 两个非单词字符之间 • 空字符串 例如，/\B../匹配"noonday"中的'oo', 而/y\B../匹配"possibly yesterday"中的‘yes‘
`\c`*`X`*	当X是处于A到Z之间的字符的时候，匹配字符串中的一个控制符。 例如，`/\cM/` 匹配字符串中的 control-M (U+000D)。
`\d`	匹配一个数字。 等价于`[0-9]`。 例如， `/\d/` 或者 `/[0-9]/` 匹配"B2 is the suite number."中的'2'。
`\D`	匹配一个非数字字符。 等价于`[^0-9]`。 例如， `/\D/` 或者 `/[^0-9]/` 匹配"B2 is the suite number."中的'B' 。
`\f`	匹配一个换页符 (U+000C)。
`\n`	匹配一个换行符 (U+000A)。
`\r`	匹配一个回车符 (U+000D)。
`\s`	匹配一个空白字符，包括空格、制表符、换页符和换行符。等价于[\f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]。 例如，`/\s\w*/` 匹配"foo bar."中的' bar'。

字符	含义
`\S`	匹配一个非空白字符。等价于 `[^ \f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]`。 例如，`/\S\w*/` 匹配"foo bar."中的'foo'。
`\t`	匹配一个水平制表符 (U+0009)。
`\v`	匹配一个垂直制表符 (U+000B)。
`\w`	匹配一个单字字符（字母、数字或者下划线）。等价于 `[A-Za-z0-9_]`。 例如，`/\w/` 匹配 "apple," 中的 'a'，"$5.28,"中的 '5' 和 "3D." 中的 '3'。
`\W`	匹配一个非单字字符。等价于 `[^A-Za-z0-9_]`。 例如，`/\W/` 或者 `/[^A-Za-z0-9_]/` 匹配 "50%." 中的 '%'。
`\n`	在正则表达式中，它返回最后的第n个子捕获匹配的子字符串(捕获的数目以左括号计数)。 比如 `/apple(,)\sorange\1/` 匹配"apple, orange, cherry, peach."中的'apple, orange,' 。
`\0`	匹配 NULL（U+0000）字符， 不要在这后面跟其它小数，因为 `\0<digits>` 是一个八进制转义序列。
`\xhh`	与代码 hh 匹配字符（两个十六进制数字）
`\uhhhh`	与代码 hhhh 匹配字符（四个十六进制数字）。
`\u{hhhh}`	（仅当设置了u标志时）使用 Unicode 值 hhhh 匹配字符（十六进制数字）。

Escaping

如果你需要使用任何特殊字符的字面值（例如，搜索字符'*'），你必须通过在它前面放一个反斜杠来转义它。 例如，要搜索'a'后跟'*'后跟'b'，你应该使用`/a\*b/`- 反斜杠"转义"字符'*'，使其成为文字而非特殊符号。

类似地，如果您正在编写正则表达式文字并且需要匹配斜杠（'/'），那么需要转义它（否则，斜杠是正则终止符）。 例如，要搜索字符串"/ example /"后跟一个或多个字母字符，您需要使用`/\/example\/[a-z]+/i`——每个斜杠之前使用反斜杠使它们成为普通字符。

要匹配文本符号反斜杠，您需要转义反斜杠。 例如，要匹配字符串"C:\"，其中"C"可以是任何字母，您将使用`/[A-Z]:\\/` —— 第一个反斜杠转义后面的那个反斜杠，因此表达式搜索单个普通字符反斜杠。

如果将RegExp构造函数与字符串文字一起使用，请记住反斜杠是字符串文字中的转义，因此要在正则表达式中使用它，您需要在字符串文字级别转义它。 `/a\*b/` 和`new RegExp("a\\*b")`创建的表达式是相同的，搜索"a"后跟文字"*"后跟"b"。

将用户输入转义为正则表达式中的一个字面字符串，可以通过简单的替换来实现：

```
function escapeRegExp(string) {
  return string.replace(/[.*+?^${}()|[\]\\]/g, "\\$&");
  //$&表示整个被匹配的字符串
}
```

使用插入语

任何正则表达式的插入语都会使这部分匹配的副字符串被记忆。一旦被记忆，这个副字符串就可以被调用于其它用途，如同 使用括号的子字符串匹配之中所述。

比如， `/Chapter (\d+)\.\d*/` 解释了额外转义的和特殊的字符，并说明了这部分pattern应该被记忆。它精确地匹配后面跟着一个以上数字字符的字符 'Chapter ' (`\d` 意为任何数字字符，`+` 意为1次以上)，跟着一个小数点（在这个字符中本身也是一个特殊字符；小数点前的 \ 意味着这个pattern必须寻找字面字符 '.')，跟着任何数字字符0次以上。 (`\d` 意为数字字符，`*` 意为0次以上)。另外，插入语也用来记忆第一个匹配的数字字符。

此模式可以匹配字符串"Open Chapter 4.3, paragraph 6"，并且'4'将会被记住。此模式并不能匹配"Chapter 3 and 4"，因为在这个字符串中'3'的后面没有点号'.'。

括号中的"?:"，这种模式匹配的子字符串将不会被记住。比如，(?:\d+)匹配一次或多次数字字符，但是不能记住匹配的字符。

使用正则表达式