# 正则表达式 - Jrenc

> 知识学习 正则表达式是一种强大的文本处理工具，可以帮助你快速地搜索、替换、匹配和分析文本。

## 知识学习 #

正则表达式是一种强大的文本处理工具，可以帮助你快速地搜索、替换、匹配和分析文本。以下是一些基本的正则表达式语法和例子：

### 基本字符匹配 #

- `a`：匹配字符 "a"。
  例子：正则表达式 `a` 可以匹配字符串 "apple" 中的第一个'a'。

### 点号 (.) #

- `.`：匹配任何单个字符（除了换行符）。
  例子：正则表达式 `h.t` 可以匹配 "hat", "hot", "hit" 等。

### 星号 (*) #

- `*`：匹配前面的字符零次或多次。
  例子：正则表达式 `ho*` 可以匹配 "h", "ho", "hoo", "hooo" 等。

### 加号 (+) #

- `+`：匹配前面的字符一次或多次。
  例子：正则表达式 `ho+` 可以匹配 "ho", "hoo", "hooo" 等，但不能匹配 "h"。

### 问号 (?) #

- `?`：匹配前面的字符零次或一次。
  例子：正则表达式 `ho?` 可以匹配 "h" 和 "ho"。

## 字符集 ([]) #

- `[abc]` ：匹配方括号内的任何一个字符（a、b 或 c）。
  例子：正则表达式 `[ch]at` 可以匹配 "cat" 和 "hat"。

## 排除字符集 ([^]) #

- `[^abc]` ：匹配任何不在方括号内的字符。
  例子：正则表达式 `[^a]n` 可以匹配 "bn", "cn" 等，但不匹配 "an"。

## 范围符号 (-) #

- `[a-z]` ：匹配任何小写字母。
- `[A-Z]` ：匹配任何大写字母。
- `[0-9]` ：匹配任何数字。
  例子：正则表达式 `[A-Z]at` 可以匹配 "Cat", "Hat" 等，但不匹配 "cat"。

## 数字 (\d) 和非数字 (\D) #

- `\d` ：匹配任何数字，等价于 `[0-9]` 。
- `\D` ：匹配任何非数字字符。
  例子：正则表达式 `\d\d` 可以匹配 "12", "45" 等。

## 单词字符 (\w) 和非单词字符 (\W) #

- `\w` ：匹配任何单词字符（包括字母、数字和下划线）。
- `\W` ：匹配任何非单词字符。
  例子：正则表达式 `\w\w` 可以匹配 "ab", "12", "a1" 等。

## 边界匹配符 (\b 和 \B) #

- `\b` ：匹配单词边界。
- `\B` ：匹配非单词边界。
  例子：正则表达式 `\bcat\b` 可以匹配 "The cat sat." 中的 "cat"，但不匹配 "caterpillar"。

## 转义字符 (\) #

- `\\` ：用于匹配原本具有特殊意义的字符，如 `.` 、 `*` 、 `?` 、 `+` 等。
  例子：要匹配 "."，使用正则表达式 `\\.` 。

## 锚点 (^ 和 $) #

- `^` ：匹配字符串的开始。
- `$` ：匹配字符串的结束。
  例子：正则表达式 `^cat` 可以匹配 "cat" 和 "caterpillar"，但不匹配 "scatter"。

## 分组 ( ) #

- `( )` ：用于将多个字符组合成一个单元，可以使用 `|` 运算符进行分组。
  例子：正则表达式 `(ab|cd)` 可以匹配 "ab" 或 "cd"。
- `(?: )` ：用于创建一个非捕获分组，不会捕获匹配的内容。
  例子：正则表达式 `(?:ab|cd)` 可以匹配 "ab" 或 "cd"，但不会捕获匹配的内容。
- `|` ：用于在分组中创建一个逻辑或操作，匹配两个或多个模式中的一个。
  例子：正则表达式 `a|b` 可以匹配 "a" 或 "b"。
- 断言，见下面。

### 捕获组与捕获组之间的关系 #

在正则表达式中，当你连续使用多个捕获组时，它们之间的关系是“顺序且关系”（Sequential AND）。这意味着，为了整个表达式匹配成功，每个捕获组必须按照它们出现的顺序在目标字符串中找到匹配，且每个捕获组的匹配结果紧跟在前一个捕获组的匹配结果之后。

举例来说，考虑正则表达式 `(A)(B)` ：

- 这个表达式包含两个捕获组： `(A)` 和 `(B)` 。
- 为了匹配成功，目标字符串中必须首先出现 `A` 的匹配，紧接着是 `B` 的匹配。
- 在这个例子中，字符串 `AB` 会成功匹配，因为它首先包含了 `A` ，紧跟着是 `B` 。

但值得注意的是，当捕获组被用在断言中时，比如前瞻断言 `(?!...)` 或者 `(?=...)` ，捕获组内的匹配不会消耗字符（即，它们不会改变正则表达式引擎当前的匹配位置）。这意味着，在断言中使用的捕获组允

许你在同一个位置检查多个条件，而不是要求紧跟在一起。例如，正则表达式 `^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])` 中的三个前瞻断言会在字符串的同一个起始位置检查三个条件：至少有一个数字、至少有一个小写字母和至少有一个大写字母。

因此，虽然捕获组通常遵循“顺序且关系”，但它们如何互动也取决于它们被如何使用，特别是当涉及到断言时。

### 捕获组顺序关系是怎么实现的？ #

正则表达式中捕获组的顺序关系是通过正则表达式引擎按照从左到右的顺序逐个评估每个表达式部分来实现的。这个过程通常包括以下步骤：

1. **开始匹配**：正则表达式引擎从字符串的开始处或上一个匹配结束的地方开始搜索。
2. **逐个评估**：引擎读取正则表达式的第一个元素（可能是字符、字符集、捕获组或断言等），并尝试在当前位置找到匹配。如果成功，引擎将移动到匹配的部分之后，准备评估下一个元素。
3. **继续向前**：引擎继续按顺序处理正则表达式的下一个元素，每次匹配成功后都会向前移动。
4. **捕获组处理**：当引擎遇到捕获组时，它会尝试匹配捕获组内的表达式。如果捕获组成功匹配，匹配的内容会被存储起来，以便以后引用。引擎随后继续从捕获组之后的位置向前移动，处理正则表达式的剩余部分。
5. **顺序匹配**：由于正则表达式引擎是按顺序处理正则表达式中的每个元素，所以捕获组（和其他表达式元素）之间的顺序关系自然形成。每个捕获组必须在前一个捕获组匹配的内容之后找到匹配，从而实现“顺序且关系”。
6. **整体匹配成功或失败**：如果正则表达式的所有部分都按顺序成功匹配，则整个表达式匹配成功；如果任何一个部分失败，则整个匹配尝试失败，引擎可能会在字符串中的下一个位置重新开始整个过程（取决于正则表达式的具体结构和匹配模式）。

通过这种方式，正则表达式引擎确保了表达式中的捕获组（和其他元素）必须按照它们出现的顺序在目标字符串中找到匹配，从而实现了它们之间的顺序关系。

## 量词 ({}) #

- `{n}` ： 匹配前面的字符恰好 n 次。
- `{n,}` ： 匹配前面的字符至少 n 次。
- `{n,m}` ： 匹配前面的字符至少 n 次，但不超过 m 次。
  例子：正则表达式 `a{2,4}` 可以匹配 "aa", "aaa" 或 "aaaa"。

## /s、/S、/t、/n #

- `\s`：匹配任何空白字符，包括空格、制表符、换行符等。
- `\S`：匹配任何非空白字符。
- `\t`：匹配制表符。
- `\n`：匹配换行符。
  例子：正则表达式 `\s+` 可以匹配一个或多个空白字符。

## 断言 #

在正则表达式中，断言用于在匹配过程中定义某些条件，而不消耗字符（即不前进到字符串中的下一个位置）。断言可以被看作是在特定位置进行的检查，它们决定了一个匹配是否成功，但它们不会影响实际匹配到的文本内容。常见的断言类型包括：

1. 前瞻断言（Lookahead Assertion）：前瞻断言用来查看某个位置之后的文本，以判断是否满足特定条件。它分为正向前瞻断言和负向前瞻断言。
    1. 正向前瞻断言（`(?=pattern)`）：只有当 `pattern` 能在当前位置之后匹配时，断言才会成功。例如，`q(?=u)` 会匹配到“quiet”中的“q”，但不会匹配到“Iraq”中的“q”。
    2. 负向前瞻断言（`(?!pattern)`）：只有当 `pattern` 不能在当前位置之后匹配时，断言才会成功。例如，`q(?!u)` 会匹配到“Iraq”中的“q”，但不会匹配到“quiet”中的“q”。
2. 后顾断言（Lookbehind Assertion）：后顾断言用来查看某个位置之前的文本，以判断是否满足特定条件。它也分为正向后顾断言和负向后顾断言。
    1. 正向后顾断言（`(?<=pattern)`）：只有当 `pattern` 能在当前位置之前匹配时，断言才会成功。例如，`(?<=\$)\d+` 会匹配到“$100”中的“100”，但不会匹配到“100 dollars”中的“100”。
    2. 负向后顾断言（`(?<!pattern)`）：只有当 `pattern` 不能在当前位置之前匹配时，断言才会成功。例如，`(?<!\$)\d+` 会匹配到“100 dollars”中的“100”，但不会匹配到“$100”中的“100”。
3. 单词边界断言（`\b` 和 `\B`）：用于确定字符是否位于单词的边界。
    1. `\b`：匹配一个单词边界，即字母和非字母字符之间的位置。例如，`\bword\b` 可以匹配到“word is”中的“word”，但不会匹配到“swordfish”中的“word”。
    2. `\B`：与 `\b` 相反，它匹配非单词边界。
4. 字符串开始和结束断言（`^` 和 `$`）：这些断言分别用于匹配字符串的开始和结束。
    1. `^`：匹配字符串的开始。例如，`^Hello` 会匹配到以“Hello”开始的字符串。

2. `$`：匹配字符串的结束。例如，`world$` 会匹配到以“world”结束的字符串。

断言是正则表达式中非常强大的工具，它们使得正则表达式能够进行复杂的条件匹配，而无需实际包含匹配的文本。这在处理复杂文本模式时非常有用，比如在密码验证、数据验证和文本分析中。

## 断言如果失败会怎样？ #

当断言失败时，意味着在当前检查的位置，文本没有满足断言所指定的条件。这将导致整个模式的匹配在当前位置失败，正则表达式引擎可能会继续在文本中寻找新的位置尝试再次匹配。以下是每种断言失败的情况及其示例：

1. 正向前瞻断言失败（`(?=pattern)`）：当后面的文本不匹配 `pattern` 时，断言失败。
    1. 示例：正则表达式 `X(?=Y)` 旨在匹配后面紧跟着 `Y` 的 `X`。在字符串 `"XY"` 中，它会匹配 `X`，因为 `X` 后面紧跟着 `Y`。但在字符串 `"XA"` 中，由于 `X` 后面是 `A` 而不是 `Y`，断言失败，因此不匹配。

2. 负向前瞻断言失败（`(?!pattern)`）：当后面的文本匹配 `pattern` 时，断言失败。
    1. 示例：正则表达式 `X(?!Y)` 旨在匹配后面没有紧跟着 `Y` 的 `X`。在字符串 `"XA"` 中，它会匹配 `X`，因为 `X` 后面没有紧跟着 `Y`。但在字符串 `"XY"` 中，由于 `X` 后面紧跟着 `Y`，断言失败，因此不匹配。

3. 正向后顾断言失败（`(?<=pattern)`）：当前面的文本不匹配 `pattern` 时，断言失败。
    1. 示例：正则表达式 `(?<=Y)X` 旨在匹配前面有 `Y` 的 `X`。在字符串 `"YX"` 中，它会匹配 `X`，因为 `X` 前面是 `Y`。但在字符串 `"AX"` 中，由于 `X` 前面是 `A` 而不是 `Y`，断言失败，因此不匹配。

4. 负向后顾断言失败（`(?<!pattern)`）：当前面的文本匹配 `pattern` 时，断言失败。
    1. 示例：正则表达式 `(?<!Y)X` 旨在匹配前面没有 `Y` 的 `X`。在字符串 `"AX"` 中，它会匹配 `X`，因为 `X` 前面没有 `Y`。但在字符串 `"YX"` 中，由于 `X` 前面是 `Y`，断言失败，因此不匹配。

5. 单词边界断言失败（`\b` 或 `\B`）：当 `\b` 用于期望单词边界但实际上位置不在单词边界时，或 `\B` 用于期望非单词边界但实际上位置在单词边界时，断言失败。
    1. 示例（`\b` 失败）：正则表达式 `\bword\b` 旨在匹配完整的单词 "word"。在字符串 "word is" 中，它会匹配 "word"，因为 "word" 前后都是单词边界。但在字符串 "swordfish" 中，由于 "word" 前后没有单词边界，`\b` 断言失败，因此不匹配。
    2. 示例（`\B` 失败）：正则表达式 `\Bis\B` 旨在匹配被其他字母包围的 "is"。在字符串 "visit" 中，它会匹配 "is"，因为 "is" 前后没有单词边界。但在字符串 "island" 中，"is" 前是单词边界，`\B` 断言失败，因此不匹配。

断言失败意味着正则表达式引擎在当前位置无法找到满足整个模式的匹配。根据正则表达式的其他部分和具体的引擎实现，引擎可能会继续在字符串中其他位置尝试匹配。
`^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(.{8,})$`
这个有三个断言，就是自动匹配在字符串开始的时候，任意一个值的后面有没有一个 \d，一个字母，一个大写字母。然后判断完后。表示任意的值，{8,} 表示 8 个以上。
`(?=.*\d)` 表示后面有没有一个数字，`(?=.*[a-z])` 表示后面有没有一个小写字母，`(?=.*[A-Z])` 表示后面有没有一个大写字母。

## 断言的边界在哪里？ #

正则表达式 `^(?=.*[A-Z])$` 看似要求字符串中至少包含一个大写字母，但实际上这个表达式并不会匹配到任何包含字符的字符串，包括你提供的 "aaSin12112"。这是因为这个表达式的构造方式导致了一个逻辑上的矛盾。下面是这个表达式的组成部分：

- `^` 和 `$`：这两个符号分别代表字符串的开始和结束。当它们紧挨在一起时，意味着它们之间不能有任何字符，这对应于一个空字符串。
- `(?=.*[A-Z])`：这是一个正向前瞻断言，它要求在当前位置之后的某个位置存在至少一个大写字母。这个断言本身不消耗任何字符（即不移动匹配位置），它只是检查条件是否满足。

当这个断言和 `^` 与 `$` 组合在一起时，`^(?=.*[A-Z])$` 实际上要求字符串必须是空的（因为 `^` 和 `$` 直接相连），但同时又要求在这个空的字符串中某处存在一个大写字母，这显然是不可能满足的。因此，这个正则表达式不会匹配包含任何字符的字符串，包括那些包含大写字母的字符串。

如果你想要匹配包含至少一个大写字母的字符串，你应该去掉 `^` 和 `$` 之间的直接联系，使得正则表达式允许其他字符的存在，例如：`^(?=.*[A-Z]).+$`。这里 `.+` 至少匹配一个任意字符，确保字符串不为空，同时 `(?=.*[A-Z])` 确保字符串中至少包含一个大写字母。
单独的 `(?=.*[A-Z])` 是一个正向前瞻断言，它用来检查字符串中是否至少包含一个大写字母。这个断言本身并不匹配任何字符，而是设置了一个条件，这个条件必须在进行任何实际匹配之前满足。

## 那 (?=.*[A-Z]) 呢？ #

在这个特定的断言中：

- `(?= ... )` 是断言的开始和结束，表示这是一个查找操作，不会消耗任何字符。
- `.*` 表示任意数量的任意字符（除换行符外），表示大写字母可以出现在字符串的任意位置。
- `[A-Z]` 匹配任何一个大写字母。

当你只使用 `(?=.*[A-Z])` 时，你是在说：“在当前位置之后，我期望找到零个或多个任意字符后跟一个大写字母。”但是，由于这个表达式只是一个断言，而没有指定要匹配的具体字符，它本身不会导致任何实际

的匹配。

要使这个断言成为一个有意义的正则表达式，可以用来匹配实际的字符串，你需要添加一些能够匹配字符的表达式部分。例如，如果你想匹配包含至少一个大写字母的任意字符串，你可以使用如下表达式：

```
(?=.*[A-Z]).*
```

在这个表达式中，`.` 匹配任意单个字符，`*` 表示匹配零个或多个前面的元素。因此，`.*` 会匹配任意长度的任意字符序列，确保整个表达式可以匹配包含至少一个大写字母的任意字符串。

## 断言是如何执行的？ #

断言是匹配的相应字符串的一个前提条件！
在正则表达式 `(?=.*[A-Z]).*` 中，断言 `(?=.*[A-Z])` 的执行过程遵循正则表达式引擎的工作方式。这个过程大致可以分解为以下步骤：

1. 引擎定位到字符串的起始位置：正则表达式引擎从字符串的开头开始工作。

2. 执行断言检查：当引擎遇到 `(?=.*[A-Z])` 这个正向前瞻断言时，它会向前查看（但不移动匹配点），以检查从当前位置开始后面的字符序列是否满足断言内的模式 `.*[A-Z]` 。
    1. `.*` 表示任意数量的任意字符（除换行符外），这意味着大写字母 `[A-Z]` 可以位于字符串中的任意位置。
    2. `[A-Z]` 表示任何一个大写字母。

3. 断言成功与否：
    1. 如果从当前位置开始的字符序列中存在至少一个大写字母，即使这个大写字母不在当前位置，断言会成功。这是因为 `.*` 允许任意数量的任意字符出现在大写字母前面。
    2. 如果断言成功，引擎会继续向下执行正则表达式中的其他部分，但它的匹配位置仍然保持在原来的起始位置，因为前瞻断言不消耗任何字符。
    3. 如果从当前位置开始的字符序列中不存在大写字母，断言会失败，整个表达式的匹配尝试在这个起始位置失败。引擎可能会尝试从字符串中的下一个位置重新开始整个匹配过程（这取决于正则表达式的具体使用方式和上下文）。

4. 处理 `.*` ：如果断言成功，正则表达式引擎接下来会遇到 `.*` 部分。
    1. 这会匹配从当前位置（也就是字符串起始位置）开始的任意数量的任意字符，直到字符串的末尾。
    2. 这意味着，如果断言成功，整个表达式 `(?=.*[A-Z]).*` 将匹配整个字符串，因为 `.*` 确保了从当前位置到字符串末尾的所有字符都被包括在匹配中。

总之，断言 `(?=.*[A-Z])` 在正则表达式 `(?=.*[A-Z]).*` 中充当了一个前提条件的角色，确保整个字符串中至少包含一个大写字母，而 `.*` 则负责实际匹配符合这个条件的整个字符串。

# 测试题目 #

可以练习正则表达式的网站 – Regex101

以下是一些正则表达式的练习题，可以帮助你加深对正则表达式基本语法的理解：

1. 匹配简单的数字

   编写一个正则表达式来匹配任何一个简单的正整数（不含前导零）。

   `\d+`

2. 电子邮件地址验证

   `^[\w]+@[A-Za-z0-9]+.[A-Za-z]+$`

   创建一个正则表达式来验证一个简单的电子邮件地址，电子邮件地址应该包含 "@" 和 "."，并且 "@" 应该出现在 "." 之前。

   `^[\w.-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$`

   以下是这个正则表达式各部分的解释：

   1. `^`：表示字符串的开始。
   2. `[\w.-]+`：匹配一个或多个字母数字字符、下划线、点或连字符。这部分用于匹配电子邮件地址的用户部分。
   3. `@`：确保电子邮件地址中包含 "@"。
   4. `[A-Za-z0-9.-]+`：匹配一个或多个字母、数字、点或连字符。这部分用于匹配电子邮件地址的域名部分。
   5. `\.`：确保电子邮件地址中包含 "."。
   6. `[A-Za-z]{2,}`：匹配两个或更多的字母。这部分通常用于匹配顶级域名。
   7. `$`：表示字符串的结束。

3. URL 匹配

   编写一个正则表达式来匹配标准的 HTTP 或 HTTPS URL。URL 应该以 http:// 或 https:// 开始，并且可以包含域名和路径。

   `^http[s]*:\/\/[\w]+.[\w]{2,}+(\/[\w]*|)$`
   `^https?:\/\/[\w.-]+\.[\w]{2,}(\/[\w\/.-]*)?$`

   1. `^`：表示字符串的开始。
   2. `https?`：`s` 后面的 `?` 表示 `s` 字符是可选的，这样可以匹配 `http` 和 `https`。
   3. `:\/\/`：匹配 "://"。斜杠在正则表达式中需要转义，所以使用了 `\\`。
   4. `[\w.-]+`：匹配一个或多个字母数字字符、下划线、点或连字符。用于匹配域名的一部分。
   5. `\.`：点字符用于分隔域名的各个部分，需要转义为 `\\.`。
   6. `[\w]{2,}`：匹配两个或更多的字母数字字符，用于顶级域名。
   7. `(\/[\w\/.-]*)?`：这是一个捕获组，用于匹配 URL 的路径部分。`\/` 匹配斜杠（路径的开始），`[\w\/.-]*` 匹配路径中的字母数字字符、斜杠、点或连字符。整个组是可选的，这由最后的 `?` 表示。

8. `$`：表示字符串的结束。

4. 日期格式验证

   创建一个正则表达式来匹配日期格式 "YYYY-MM-DD"，其中年份为四位数字，月份和日期为两位数字。

   `^\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$`

   1. `^`：表示字符串的开始。
   2. `\d{4}`：匹配四位数字，用于表示年份。
   3. `-`：字面量字符，表示日期各部分之间的分隔符。
   4. `(0[1-9]|1[0-2])`：这是一个捕获组，用于匹配 01 到 09 月和 10 到 12 月。`0[1-9]` 匹配 01 至 09，`1[0-2]` 匹配 10 至 12。
   5. `-`：再次匹配日期各部分之间的分隔符。
   6. `(0[1-9]|[12][0-9]|3[01])`：这是另一个捕获组，用于匹配月份中的日。`0[1-9]` 匹配 01 至 09 日，`[12][0-9]` 匹配 10 至 29 日，`3[01]` 匹配 30 和 31 日。
   7. `$`：表示字符串的结束。

5. 手机号码验证

   编写一个正则表达式来验证一个简单的手机号码，手机号码应该以 "1" 开始，总共有 11 位数字。

   `1[\d]{10}`

6. IP 地址匹配

   创建一个正则表达式来匹配标准的 IPv4 地址，每个八位字节应该是 0 到 255 之间的数字，四个八位字节由 "." 分隔。

   `(25[0-5]|[0-1][0-9][0-9]|2[0-4][0-9])(.(25[0-5]|[0-1][0-9][0-9]|2[0-4][0-9])){3}`

7. HTML 标签匹配

   编写一个正则表达式来匹配简单的 HTML 标签，例如 `<div>` 或 `<a href="...">`，标签名称可以是任意字母组合。

   `<([a-zA-Z]+)(\s+[a-zA-Z]+="[^"]*")*\s*>`

   考虑一个带有多个属性且属性值中包含特殊字符的 HTML 标签作为例子。假设我们有下面这样一个带有 `data-attribute` 的 `<a>` 标签，其中 `data-attribute` 的值包含一个 JSON 对象：

   `<a href="/example" data-attribute="{\"key\": \"value\"}">`

   在这个例子中：

   1. 使用 `<[\w\/\s\+\=\"\']+>` 正则表达式可能无法正确匹配整个标签，因为属性值中的 `{`，`}`，和 `:` 等特殊字符不在这个表达式定义的字符集中。这可能导致匹配在第一个遇到的特殊字符处中断。
   2. 使用 `<([a-zA-Z]+)(\s+[a-zA-Z]+="[^"]*")*\s*>` 正则表达式可以更有效地匹配整个标签，因为它专门寻找由空格分隔的、格式为 `属性="值"` 的模式，其中值被双引号包围。这个表达式不关心双引号内的具体内容，允许属性值内包含 `{`，`}`，`:`，或其他任何字符，只要它们位于双引号内部。

因此，尽管第一个正则表达式更为宽泛和灵活，但它在处理复杂属性值时可能会遇到困难。第二个正则表达式通过更严格地定义属性结构，提供了更稳定和准确的匹配，尤其是在处理包含特殊字符的属性值时。

两者的区别在于，在属性内它使用 ^" 定义，这样就能匹配到属性内的特殊字符，但其实在 vue 里还有 @符号，所以这个正则表达式还是不够完善的。

8. 密码强度验证

创建一个正则表达式来验证密码强度。密码必须至少包含一个数字、一个大写字母、一个小写字母，且总长度至少为 8 个字符。

要创建一个正则表达式以验证密码强度，确保密码至少包含一个数字、一个大写字母、一个小写字母，并且总长度至少为 8 个字符，我们可以使用如下正则表达式：

```
^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$
```

这个表达式的各部分解释如下：

1. `^`：表示字符串的开始。
2. `(?=.*\d)`：这是一个前瞻断言，用于确保字符串中至少有一个数字。`.` 匹配任何字符（除了换行符），`*` 表示零个或多个字符，`\d` 表示一个数字字符。
3. `(?=.*[a-z])`：这是另一个前瞻断言，用于确保字符串中至少有一个小写字母。`[a-z]` 匹配任何一个小写字母。
4. `(?=.*[A-Z])`：这是第三个前瞻断言，用于确保字符串中至少有一个大写字母。`[A-Z]` 匹配任何一个大写字母。
5. `.{8,}`：`.` 表示任何字符（除了换行符），`{8,}` 表示匹配前面的字符至少 8 次。
6. `$`：表示字符串的结束。

这个正则表达式通过使用前瞻断言来独立检查每个必需的字符类型（数字、小写字母和大写字母），并且使用 `.{8,}` 来确保密码的总长度至少为 8 个字符。请注意，这个表达式不限制密码的最大长度，并假设密码中可以包含任何字符，只要它们满足至少有一个数字、一个大写字母和一个小写字母的条件。

9. 代码注释匹配

编写一个正则表达式来匹配 JavaScript 或 C 风格的单行注释，即以 "//" 开始的注释。

要匹配 JavaScript 或 C 风格的单行注释，即那些以 "//" 开头直到行尾的注释，你可以使用以下正则表达式：

```
^\/\/.*
```

这个表达式的组成部分解释如下：

1. `^`：这表示字符串的开始。如果你想在文本的任意位置匹配注释，可以去掉这个符号。
2. `\/\/`：由于 `/` 在正则表达式中是一个特殊字符，需要使用反斜线 `\` 进行转义。因此，`\/\/` 用来匹配文本中的 "//" 字符串。
3. `.*`：这表示匹配 "//" 后面的任意数量的任意字符（除换行符外），直到行尾。

使用这个正则表达式时，请注意几个方面：

1. 如果你使用这个表达式进行全局搜索，不希望它必须从每行的开头开始匹配，就去掉 `^` 。
2. 这个表达式假设注释不包含换行符。如果注释可以跨越多行（尽管这与常规的单行注释定义不符），那么这个表达式就不适用。
3. 在某些正则表达式处理器中，为了匹配任意字符包括换行符，你可能需要使用一个不同的符号或模式标志，具体取决于你使用的工具或语言。

10. 文件路径匹配

    创建一个正则表达式来匹配 Unix 风格的文件路径，路径可以包含字母、数字、斜杠 `/` 和点 `.` 。

    ```
    ^[\/\w.]+$
    ```

    你可以使用在线正则表达式测试工具，如 Regex101 ，来练习这些题目并测试你的解决方案。

全文完