介绍

正则表达式（Regular Expression，简称 RegExp），是一种文本模式匹配工具，可以匹配普通字符特殊字符等。它提供了一种灵活且强大的方式来查找、替换、验证和提取文本数据。它可以应用于各种编程语言，如 JavaScript、Python、Java、PHP 等。

正则表达式在线测试工具：https://c.runoob.com/front-end/854/

正则表达式基本格式：

```
/pattern/flags
```

pattern 表示模式，flags 表示修饰符。

模式放在斜杠 / 之间，斜杠后面添加修饰符，修饰符包含 i，g，m 等，修饰符放在斜杠的第二个斜杠后面。

1. 详细介绍

1.1. 模式（Pattern）

模式是正则表达式的主体部分，它由各种字符和元字符组成，定义了要匹配的文本模式。例如，[0-9]+ 是一个模式，表示匹配一个或多个数字。

1.2. 斜杠（/）

斜杠用于包围正则表达式的模式，将其限定在两个斜杠之间。例如，/pattern/。

1.3. 修饰符（Flags）

> - i：表示不区分大小写匹配。
> - g：表示全局匹配，即匹配所有而非仅匹配第一个。
> - m：表示多行匹配，即 ^ 和 $ 可以匹配字符串中每一行的开头和结尾。

例如想要匹配一个字符串中的数字，可以使用以下正则表达式：

```
/\d+/g
```

- \d+：匹配一个或多个数字。
- /：正则表达式的开始。
- g：全局匹配标志，表示匹配所有匹配项。

这个正则表达式将匹配字符串中的所有连续数字。如：123、javascript123、123ES6 等。

元字符

元字符在正则表达式中扮演着重要的角色，通过组合使用它们，能构建复杂的模式来匹配和处理文本。

序号	字符	含义	实例
1	`.`	匹配除换行符以外的任意单个字符	`a.b` 能匹配 `"aab"`、`"axb"`，但不匹配 `"a\nb"`
2	`^`	匹配字符串的开头	`^abc` 能匹配 "abc"，但不匹配 "aabc"
3	`$`	匹配字符串的结尾	`abc$` 能匹配 `"abc"`，但不匹配 `"abcc"`
4	`\b`	匹配单词的边界	`er\b` 能匹配 `"never"` 中的 `"er"`，但不能匹配 `"verb"` 中的 `"er"`
5	`\B`	匹配非单词的边界	`er\B` 能匹配 `"verb"` 中的 `"er"`，但不能匹配 `"never"` 中的 `"er"`
6	`\f`	匹配一个换页符	等价于 `\x0c` 和 `\cL`
7	`\n`	匹配一个换行符	等价于 `\x0a` 和 `\cJ`
8	`\r`	匹配一个回车符	等价于 `\x0d` 和 `\cM`
9	`\d`	匹配任意一个阿拉伯数字（0 到 9）	等价于 `[0-9]`
10	`\D`	匹配一个非数字字符	等价于 `[^0-9]`
11	`\w`	匹配字母、数字、下划线	等价于 `[A-Za-z0-9_]`
12	`\W`	匹配非字母、数字、下划线	`[^A-Za-z0-9_]`
13	`\s`	匹配所有空白符，包括换行、制表符、换页符	等价于 `[\f\n\r\t\v]`
14	`\S`	匹配非空白符，不包括换行	`[^\f\n\r\t\v]`
15	`[\s\S]`	匹配所有字符	
16	`*`	匹配前一个字符零次或多次	`ab*c` 能匹配 `"ac"`、`"abc"`、`"abbc"`
17	`+`	匹配前一个字符一次或多次	匹配前一个字符一次或多次
18	`?`	匹配前一个字符零次或一次	`ab?c` 能匹配 `"ac"`、`"abc"`，但不匹配 `"abbc"`
19	`{n}`	匹配前一个字符恰好 `n` 次	`o{2}` 不能匹配 `"Bob"` 中的 `o`，但是能匹配 `"food"` 中的两个 `o`。
20	`{n,}`	匹配前一个字符至少 `n` 次	`o{2,}` 不能匹配 `"Bob"` 中的 `o`，但能匹配 `"fooooood"` 中的所有 `o`。`o{1,}` 等价于 `o+`。`o{0,}` 则等价于 `o*`

序号	字符	含义	实例
21	`{n,m}`	匹配前一个字符至少 `n` 次，但不超过 `m` 次，**逗号前后不能有空格**	`o{1,3}` 将匹配 `"fooooood"` 中的前三个 `o`。`o{0,1}` 等价于 `o?`
22	`[]`	匹配所包含的任意一个字符	`[abc]` 能匹配 `"plain"` 中的 `a`
23	`[^]`	匹配所包含的任意一个字符以外的字符	`[^abc]` 能匹配 `"plain"` 中的 `p`、`l`、`i`、`n`
24	`[-]`	匹配指定范围内的任意字符	`[a-z]` 能匹配 `a` 到 `z` 范围内的任意小写字母字符
25	`|`	匹配竖线两侧的任意一个	`"z | food"`能匹配`"z"`或`"food"`。`"(z | f)ood"`则匹配`"zood"`或`"food"`
26	`()`	用于创建分组，并允许对分组应用量词	`(abc)+` 能匹配 `"abc"`、`"abcabc"`，不匹配 `"ab"`
27	`\`	用于转义下一个字符，取消其特殊含义	`n` 匹配字符 `"n"`。`\n` 匹配一个换行符

修饰符

在正则表达式中，修饰符是用来修改搜索模式的标志，添加在正则表达式的末尾，以控制匹配的方式。格式为：`/pattern/flags`。`pattern` 为正则表达式，`flags` 为修饰符。

正则表达式的修饰符可以单独使用，也可以组合使用，它们提供了更灵活的匹配选项，适应不同的需求。

1. g - 全局搜索（Global）

- 示例：`/abc/g`
- 匹配：`"abc"`, `"abcabc"`, `"abcxyzabc"`
- 不匹配： `"ac"`。

2. i - 不区分大小写（Case Insensitive）

- 示例：`/abc/i`
- 匹配：`"abc"`, `"AbC"`, `"ABC"`
- 不匹配：`"ac"`

3. m - 多行匹配（Multiline）

- 示例：`/^abc/m`
- 匹配：`"abc"`（字符串的开头），`"xyz\nabc"`（字符串的第二行）
- 不匹配：`"xyz\nabc"`（字符串的开头）。

4. s - 单行匹配（Single line）

- 示例：`/abc/s`
- 匹配：`"abc"`（字符串中的任何位置，包括换行符）
- 不匹配：`"ab\nc"`, `"a\nb\nc"`

5. u - Unicode 匹配模式

- 示例：/[\u4e00-\u9fa5]+/u
- 匹配：匹配中文字符
- 不匹配：字母、数字

6. y - 粘附匹配

`y` 修饰符是 ECMAScript 6 中引入的，它使得正则表达式的匹配从字符串的当前位置开始，而不是从上次匹配的位置开始，这种方式被称为粘附匹配（sticky matching）。

- 示例：`/abc/y`
- 匹配：`"abc"`（字符串的开头）
- 不匹配：`"xyz\nabc"`（字符串的开头之后）

7. x - 忽略空白字符（Whitespace）

x 修饰符用于忽略正则表达式中的空白字符（除了在字符类中的空白字符），这样可以使正则表达式更易读，可以添加注释和格式化。

- 示例：`/a b c/x`
- 匹配：`"abc"`
- 不匹配：`"a b c"`

`x` 和 `y` 这两个修饰符都提供了更多的灵活性和可读性，但需要注意的是，它们可能在某些环境中不被完全支持，使用时，最好检查目标环境的正则表达式引擎的兼容性。

断言

正则表达式的断言是一种特殊的模式匹配技术，用于在匹配时对字符串进行条件性的预测。断言不会消耗输入字符串，仅仅是在匹配的位置上进行条件判断。断言分为正向断言和负向断言，分别用于描述匹配位置前面或后面的条件。

正则表达式的断言有 4 种形式：

- `(?=pattern)` -- 零宽正向先行断言：匹配位置之前有指定的条件。
- `(?!pattern)` -- 零宽负向先行断言：匹配位置之前没有指定的条件。
- `(?<=pattern)` -- 零宽正向后行断言：匹配位置之后有指定的条件。
- `(?<!pattern)` -- 零宽负向后行断言：匹配位置之后没有指定的条件。

几个名字概念说明:

> - 零宽：只匹配位置，零宽意味着断言在匹配时不会"消耗"字符串，它只是对位置进行条件判断，不包括匹配位置之前或之后的字符在匹配结果中。
> - 先行：表示断言发生在匹配位置之前。
> - 后行：表示断言发生在匹配位置之后。
> - 正向：匹配括号中的表达式，即断言所作的条件判断是肯定的，即只有当条件成立时，匹配才成功。
> - 负向：不匹配括号中的表达式，即断言所作的条件判断是否定的，即只有当条件不成立时，匹配才成功。

1. 零宽正向先行断言

零宽正向先行断言也称正向向前查找，模式：`(?=pattern)`。

这个断言用于在匹配位置之前添加一个条件，只有当这个条件匹配成功时，整个模式才会成功匹配，但匹配位置之前的内容并不包括在匹配结果中。

实例：匹配包含 `"mozilla"` 后面跟着 `"org"` 的字符串。

```
/mozilla(?=org)/
```

在上述正则表达式中，`(?=org)` 表示在 `"mozilla"` 之后必须紧跟着 `"org"` 才算匹配成功。

2. 零宽负向先行断言

零宽负向先行断言也称负向向前查找，模式：`(?!pattern)`。

这个断言用于在匹配位置之前添加一个条件，只有当这个条件不匹配时，整个模式才会成功匹配。

实例：匹配包含 `"mozilla"` 后面不跟着 `"org"` 的字符串。

```
/mozilla(?!org)/
```

在上述正则表达式中，`(?!org)` 表示在 `"mozilla"` 之后不能跟着 `"org"` 才算匹配成功。

3. 零宽正向后行断言

零宽正向后行断言，又称正向向后查找，模式：`(?<=pattern)`。

这个断言用于在匹配位置之后添加一个条件，只有当这个条件匹配成功时，整个模式才会成功匹配。同样，匹配位置之后的内容并不包括在匹配结果中。

实例：匹配前面跟着 `"mozilla"` 的单词。

```
/(?<=mozilla)\w+/
```

在上述正则表达式中，`(?<=mozilla)` 表示在匹配位置之前必须有 `"mozilla"` 才算匹配成功。

4. 零宽负向后行断言

零宽负向后行断言，又称负向向后查找，模式：`(?<!pattern)`。

这个断言用于在匹配位置之后添加一个条件，只有当这个条件不匹配时，整个模式才会成功匹配。

实例：匹配前面不跟着 "mozilla" 的单词。

```
/(?<![0-9]+)mozilla/
```

在上述正则表达式中，`(?<![0-9]+)` 表示在匹配的字符串 `"mozilla"` 之前不能有数字才算匹配成功。

运算符优先级

正则表达式从左到右进行计算，并遵循优先级顺序，这与算术表达式非常类似。

相同优先级的从左到右进行运算，不同优先级的运算先高后低。各种正则表达式运算符的优先级从最高到最低顺序：

序号	字符	含义
1	`\`	转义符
2	`()`, `(?:)`, `(?=)`, `[]`	圆括号和方括号
3	`*`, `+`, `?`, `{n}`, `{n,}`, `{n,m}`	限定符（量词）
4	`^`, `$`, `\任何元字符、任何字符`	定位点和序列（即：位置和顺序）
5	`竖线`	替换，"或"操作

字符具有高于替换运算符的优先级，使得 `"m|food"` 匹配 `"m"` 或 `"food"`。若要匹配 `"mood"` 或 `"food"`，请使用括号创建子表达式，从而产生 `"(m|f)ood"`。

一些常见正则表达式运算符按照优先级从高到低的顺序：

- 转义符号： `\` 是用于转义其他特殊字符的转义符号。它具有最高的优先级。

示例：`\d`、`\.` 等，其中 `\d` 匹配数字，`\.` 匹配点号。

- 括号： 圆括号 `()` 用于创建子表达式，具有高于其他运算符的优先级。

示例：`(abc)+` 匹配 `"abc"` 一次或多次。

- 量词： 量词指定前面的元素可以重复的次数。

示例：`a*` 匹配零个或多个 `"a"`。

- 字符类： 字符类使用方括号 [] 表示，用于匹配括号内的任意字符。

示例：`[aeiou]` 匹配任何一个元音字母。

- 断言： 断言是用于检查字符串中特定位置的条件的元素。

示例：`^` 表示行的开头，`$` 表示行的结尾。

- 连接： 连接在没有其他运算符的情况下表示字符之间的简单连接。

示例：`abc` 匹配 `"abc"`。

- 管道： 管道符号 `|` 表示"或"关系，用于在多个模式之间选择一个。

示例：`cat|dog` 匹配 `"cat"` 或 `"dog"`。

常用案例

1. 匹配邮箱地址

```
/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/
```

2. 匹配 URL

```
/^(https?|ftp):\/\/(-\.)?([^\s\/?\.#-]+\.?)+(\/[^\s]*)?$/
```

3. 匹配日期（YYYY-MM-DD）

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

4. 手机号码

```
/^[0-9]{10}$/
```

5. 身份证号码（18位）

```
/^\d{17}(\d|X|x)$/
```

6. 用户名（包含字母、数字、下划线，长度为 3 到 16 个字符）

```
/^[a-zA-Z0-9_]{3,16}$/
```

7. 匹配 IP 地址

```
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/
```

8. HTML 标签

```
/<([a-zA-Z][a-zA-Z0-9]*)\b[^>]*>(.*?)<\/\1>/
```

9. 匹配数字（整数或浮点数）

```
/^\d+(\.\d+)?$/
```

10. 匹配空白行

```
/^\s*$/
```

11. 匹配中文字符

`[\u4e00-\u9fa5]` 匹配 `Unicode` 范围内的中文字符。

```
/[\u4e00-\u9fa5]/
```

12. 匹配邮政编码

以非零数字开头，以 5 位任意数字结尾。

```
/^[1-9]\d{5}$/
```

13. 匹配十六进制颜色代码

```
/^#?([a-fA-F0-9]{6}|[a-fA-F0-9]{3})$/
```

14. 匹配时间（24小时制）

```
/^(0[0-9]|1[0-9]|2[0-3]):[0-5][0-9]$/
```

15. 匹配HTML注释

```
/<!--[\s\S]*?-->/
```

16. 匹配 Markdown 标题

```
/^#{1,6}\s.*$/
```

17. 匹配英文句子

```
/[A-Z][^.!?]*[.!?]/g
```

18. 匹配 JSON 键值对

`\s` 代表空白字符。

```
/"([^"]+)":\s*"([^"]+)"/
```

19. 匹配 HTML 图片标签

```
/<img\s+src="([^"]+)"\s*\/?>/
```

JavaScript 使用

JavaScript 的正则表达式是由 RegExp 对象表示的，同时也可以使用正则表达式字面量。

1. 使用 RegExp 对象

```
let pattern = new RegExp("pattern", "flags");
```

`pattern` 是字符串形式的正则表达式模式。`flags` 是字符串形式的修饰符，可以包含 `i`, `g`, `m` 等。

```
let pattern = new RegExp("abc", "i");           // 匹配 "abc"，不区分大小写
let globalPattern = new RegExp("abc", "g");     // 匹配所有的 "abc"
let multilinePattern = new RegExp("^abc", "m"); // 匹配每一行的开头是 "abc"
```

2. 使用字面量

```
let pattern = /pattern/flags;
```

`pattern` 是正则表达式的模式，可以包含字符、字符集、量词等。

`flags` 是修饰符，可以是以下之一或它们的组合：

- `i`：忽略大小写匹配。
- `g`：全局匹配，匹配所有符合条件的字符串。
- `m`：多行匹配，`^` 和 `$` 匹配每一行的开头和结尾。

```
let pattern = /mozilla/i;
```

3. 常用的正则表达式方法

3.1. test

`test` 方法用于检测字符串是否匹配正则表达式，返回布尔值：

```
let pattern = /\d+/;
let result = pattern.test("123abc"); // true
```

上例检测字符串 `"123abc"` 是否包含一个或多个数字。

3.2. exec

`exec` 方法返回第一个匹配的结果数组，或者在没有匹配时返回 `null`：

```
let pattern = /\d+/;
let result = pattern.exec("123abc");  // ["123"]
```

上例在字符串 `"123abc"` 中查找第一个匹配模式 `\d+`（即一个或多个数字）的子字符串，并返回包含匹配结果的数组 `["123"]`。

3.3. match

`match` 方法在字符串中查找一个或多个匹配，返回一个包含匹配结果的数组：

```
let pattern = /\d+/;
let result = "123abc".match(pattern);  // ["123"]
```

上例在字符串 `"123abc"` 中查找第一个匹配模式 `\d+`（即一个或多个数字）的子字符串，并返回包含匹配结果的数组 `["123"]`。

与 `exec` 方法相比，`match` 方法用于在字符串中查找第一个匹配，但返回结果的形式略有不同。

3.4. search

`search` 方法返回字符串中第一个匹配的索引，如果没有匹配则返回 -1：

```
let pattern = /\d+/;
let result = "abc123".search(pattern);  // 3
```

上例在字符串 `"abc123"` 中查找是否包含匹配模式 `\d+`（即一个或多个数字）的子串，并返回匹配的子串在原字符串中的索引，即返回 3。

3.5. replace

`replace` 方法用指定的字符串或函数替换匹配的子串：

```
let pattern = /\d+/;
let result = "abc123".replace(pattern, "X");  // "abcX"
```

上例将字符串 `"abc123"` 中匹配模式 `\d+`（即一个或多个数字）的子串替换为字符串 `"X"`，返回替换后的新字符串 `"abcX"`。

3.6. split

`split` 方法使用正则表达式或指定的子字符串拆分字符串，并返回一个数组：

```
let pattern = /\s+/;
let result = "This is a sentence".split(pattern);  // ["This", "is", "a",
"sentence"]
```

上例将字符串 `"This is a sentence"` 根据空白字符拆分为一个数组，每个数组元素都是原字符串中的一个单词，返回的结果是 `["This", "is", "a", "sentence"]`。