正则式

目录

常规介绍

正则表达式（Regular Expression，简称 regex）是一套用于处理**字符串匹配和搜索**的强大工具。它的核心在于用一系列特殊的字符来定义一个**搜索模式**，然后可以用这个模式去**查找、替换、截取**文本中符合该模式的片段。

下面我将从**核心规则**和**常用符号**两个方面为你介绍。

一、核心规则与思想

学习正则表达式，首先要理解几个核心规则：

1. **字面匹配**：绝大多数字符，包括所有的字母和数字，都会简单地匹配它们自身。
    - 例如：正则 **abc** 会匹配字符串中任何位置的 **"abc"**。
2. **元字符**：一些特殊字符拥有超越字面意义的强大功能，它们是正则表达式的“魔法符号”。这也是学习的重点。
    - 例如：**.**、*****、**+**、**?**、****、**|**、**^**、**$**、**[**、**]**、**{**、**}**、**(**、**)**。
3. **转义**：如果你想匹配元字符本身，而不是使用它的特殊功能，就需要在它前面加上反斜杠 **** 进行转义。
    - 例如：要匹配字符串 **"1+1=2"** 中的加号，正则应该写成 **1\+1=2**，否

则 **+** 会被当作一个量词。

4. **贪婪与懒惰**：默认情况下，量词（如 *****, **+**, **{n,}**）是“贪婪”的，它会匹配**尽可能长**的字符串。你可以在量词后面加一个 **?** 来开启“懒惰”模式，让它匹配**尽可能短**的字符串。
    - **贪婪示例**：在字符串 **" hello world "** 中使用 ** .* ** 会匹配整个字符串，从第一个 ** ** 到最后一个 ** **。
    - **懒惰示例**：使用 ** .*? ** 则会匹配两次：第一次是 **" hello "**，第二次是 **" world "**。

二、常用符号详解

为了方便你查阅，我将常用符号分为几类，并用表格和示例说明。

1. 基本元字符

符号	含义与功能	示例与说明
.	**匹配除换行符外的任意一个字符**	**a.c** 可以匹配 **"abc"**, **"a@c"**, **"a c"**
****	**转义符**或**引入特殊序列**	**\.** 匹配真正的点号；**\d** 匹配数字
|	**或**，匹配左边或右边的表达式	**cat|dog** 匹配 **"cat"** 或 **"dog"**
()	**分组**，将多个元素组合成一个整体，并可对其使用量词或捕获内容	**(ab)+** 匹配 **"ab"**, **"abab"** 等；同时也用于**捕获**匹配的内容
[]	**字符簇**，匹配方括号内的任意**一个**字符	**[aeiou]** 匹配任意一个（小写）元音字母；**[a-z]** 匹配任意一个小写字母

2. 位置锚点

符号	含义与功能	示例与说明
^	**匹配字符串的开始位置**	**^Hello** 只会匹配以 **"Hello"** 开头的字符串
$	**匹配字符串的结束位置**	**world$** 只会匹配以 **"world"** 结尾的字符串
\b	**匹配一个单词的边界**（空格、标点或开头结尾）	**\bcat\b** 匹配 **"a cat"** 中的 **cat**，但不匹配 **"category"**

3. 预定义字符集（常用转义序列）

符号	等价于	含义与功能
\d	[0-9]	**匹配任意一个数字**
\D	[^0-9]	**匹配任意一个非数字字符**
\w	[a-zA-Z0-9_]	**匹配字母、数字、下划线**（单词字符）
\W	[^\w]	**匹配非单词字符**
\s	[\t\r\n\f]	**匹配任意空白符**（空格、制表符、换行等）
\S	[^\s]	**匹配任意非空白符**

4. 量词（指定数量）

符号	含义与功能	示例与说明
*	**匹配前面的元素零次或多次**	**ab*c** 匹配 **"ac"**, **"abc"**, **"abbc"** ...

符号	含义与功能	示例与说明
+	匹配前面的元素一次或多次	ab+c 匹配 "abc", "abbc" … 但不匹配 "ac"
?	匹配前面的元素零次或一次	colou?r 匹配 "color" 和 "colour"；也表示懒惰模式
{n}	匹配前面的元素恰好 n 次	\d{4} 匹配 4 位数字，如年份 "2023"
{n,}	匹配前面的元素至少 n 次	\d{2,} 匹配至少 2 位数字，如 "10", "100"
{n,m}	匹配前面的元素至少 n 次，至多 m 次	\d{3,5} 匹配 3 到 5 位数字，如 "123", "12345"

三、实用示例

1. **匹配邮箱地址**（简易版）：

regex

```
\b[\w.-]+@[\w.-]+\.\w+\b
```

   - **\b**：单词边界，确保邮箱是独立的
   - **[\w.-]+**：匹配用户名部分（字母、数字、下划线、点、横线，出现一次以上）
   - **@**：匹配字面 @ 符号
   - **[\w.-]+**：匹配域名部分（如 **gmail**, **yahoo**）
   - **\.**：匹配真正的点号
   - **\w+**：匹配顶级域名（如 **com**, **org**）
   - **\b**：单词边界

2. **匹配日期（YYYY-MM-DD 格式）**：

regex

```
\d{4}-\d{2}-\d{2}
```

- 非常简单直接：4 位数字-2 位数字-2 位数字

3. **提取 HTML 标签中的内容**:

regex

<title>(.*?)</title>

- **<title>**：匹配开始标签
- **(.*?)**：**懒惰模式**捕获任意内容（**?** 让 **.*** 变得不贪婪）
- **</title>**：匹配结束标签
- 最终，分组 **(.*?)** 捕获到的内容就是标题文本。

注意：区分 match()和 search()。Match 需要从头匹配

在字符类 [] 中的转义规则

在正则表达式的字符类（方括号 **[]** 内）中，转义规则确实与外面有所不同。简单来说：

核心规则

在大多数情况下，在字符类 [] 中不需要对特殊字符进行转义，因为它们会失去特殊含义，只表示字面字符本身。但有几个重要的例外。

不需要转义的情况

在字符类中，以下元字符会失去特殊含义，不需要转义：

- **.** → 就表示字面点号，而不是"任意字符"
- **+** → 就表示加号字符
- ***** → 就表示星号字符

- **?** → 就表示问号字符
- **$** → 就表示美元符号
- **^** → 但只有当它不是第一个字符时!
- **|** → 就表示竖线字符
- **()** → 就表示括号字符

例如:

- **[a-z.+*?$|()]** 匹配任何小写字母或 **.**、**+**、*****、**?**、**$**、**|**、**(**、**)** 字符

需要转义的情况

有少数字符在字符类中仍然具有特殊含义，需要转义:

1. **]** - 右方括号：表示字符类的结束
    - 需要转义：**[\[\]]** 匹配 **[** 或 **]**
2. **** - 反斜杠：仍然用作转义字符
    - 需要转义：**[\\]** 匹配反斜杠
3. **-** - 连字符：当不在开头或结尾时表示范围
    - 需要转义或放在开头/结尾：**[a\-z]** 或 **[-az]** 匹配 **a**、**-** 或 **z**
4. **^** - 脱字符：当是第一个字符时表示取反
    - 需要转义或不要放在开头：**[\^a]** 或 **[a^]** 匹配 **a** 或 **^**

正则表达式中的懒惰模式

懒惰模式（Lazy Mode），也称为非贪婪模式（Non-greedy Mode）或最小匹配（Minimal Matching），是正则表达式中一个非常重要的概念，它与默认的贪婪模式（Greedy Mode）相对。

贪婪模式 vs 懒惰模式

贪婪模式（默认行为）

- **匹配尽可能多的字符**
- 量词(*, +, ?, {})会尝试匹配**尽可能长**的字符串

- 这是正则表达式默认的行为

懒惰模式

- **匹配尽可能少的字符**
- 在量词后面加上**?**来启用(***?**, **+?**, **??**, **{}?**)
- 它会匹配**尽可能短**的字符串，同时仍满足整个表达式的匹配条件

示例说明

假设我们有这样一个字符串：
"Hello 'world' and 'universe'"

贪婪模式匹配

regex

'.*'

- 匹配结果：**'world' and 'universe'**
- 解释：**.**匹配任意字符，*****匹配 0 次或多次，贪婪模式会匹配从第一个**'**到最后一个**'**之间的所有内容

懒惰模式匹配

regex

'.*?'

- 匹配结果：**'world'** 和 **'universe'**（两次匹配）
- 解释：**.*?**会匹配尽可能少的内容，所以它会在遇到第一个闭合引号**'**时就停止匹配

实际应用场景

1. 提取 HTML 标签内容

html

Hello World

- 贪婪模式：** .* ** → 匹配整个字符串
- 懒惰模式：** .*? ** → 匹配** Hello **和** World **

2. 提取引号内的内容

text

She said "hello" and he said "goodbye"

- 贪婪模式：**".*"** → 匹配**"hello" and he said "goodbye"**
- 懒惰模式：**".*?"** → 匹配**"hello"**和**"goodbye"**

3. 匹配最短的可能序列

text

aaa bbb aaa bbb aaa

- 贪婪模式：**aaa.*aaa** → 匹配从第一个 **aaa** 到最后一个 **aaa** 的所有内容
- 懒惰模式：**aaa.*?aaa** → 匹配从第一个 **aaa** 到第二个 **aaa** 的内容

性能考虑

懒惰模式通常比贪婪模式**效率稍低**，因为正则表达式引擎需要做更多的回溯来找到最短的可能匹配。但在大多数情况下，这种差异可以忽略不计，而且懒惰模式提供了更精确的匹配控制。

总结

- **贪婪模式**：量词默认行为，匹配尽可能多的字符
- **懒惰模式**：在量词后加**?**，匹配尽可能少的字符
- **使用场景**：当你想要匹配最短的可能序列，而不是最长的可能序列时
- **示例转换**：
    - **.*** → **.*?** (贪婪 → 懒惰)
    - **.+** → **.+?**
    - **.?** → **.??**

懒惰模式是处理诸如提取 HTML 标签内容、引号内文本等场景的必备工具，它让你能够更精确地控制正则表达式的匹配行为。

前瞻和后顾断言

regex

a(?=...)　正向前瞻：找 a，是不是，在(?=b)前面

a(?!...)　负向前瞻：找 a，是不是，不在(?!b)前面

(?<=...)a 正向后顾：找 a，是不是，在(?<=b)后面

(?<!...)a 负向后顾：找 a，是不是，不在(?<!b)后面

正则表达式中的捕获组

捕获组（Capturing Group）是正则表达式中一个非常重要且强大的功能。它允许您从匹配的文本中提取特定的部分，或者在后序操作中引用这些部分。

基本概念

捕获组是通过圆括号 **()** 来定义的。当正则表达式匹配文本时，圆括号内的模式会被"捕获"并存储起来，供后续使用。

简单示例

考虑这个正则表达式：**(\d{3})-(\d{3})-(\d{4})**

它可以匹配美国电话号码格式：**123-456-7890**

这里有三个捕获组：

1. **(\d{3})** - 捕获区号 "123"
2. **(\d{3})** - 捕获前缀 "456"
3. **(\d{4})** - 捕获线路号码 "7890"

捕获组的用途

1. 提取信息

在编程中，您可以使用捕获组来提取匹配文本的特定部分：

python

```
import re


text = "我的电话号码是：123-456-7890"
pattern = r"(\d{3})-(\d{3})-(\d{4})"
```

```
match = re.search(pattern, text)
if match:
    print("完整匹配:", match.group(0))   # 123-456-7890
    print("区号:", match.group(1))       # 123
    print("前缀:", match.group(2))       # 456
    print("线路号:", match.group(3))     # 7890
```

2. 在替换中使用（反向引用）

您可以在替换操作中使用捕获组的内容：

```
python
# 重新格式化电话号码
text = "123-456-7890"
result = re.sub(r"(\d{3})-(\d{3})-(\d{4})", r"(\1) \2-\3", text)
print(result)  # 输出: (123) 456-7890
```

这里 **\1**, **\2**, **\3** 分别引用第一个、第二个和第三个捕获组的内容。

3. 在正则表达式内部引用（反向引用）

您可以在同一个正则表达式中引用前面的捕获组：

```
python
# 匹配重复的单词
text = "hello hello world"
pattern = r"(\b\w+\b) \1"  # \1 引用第一个捕获组匹配的内容

match = re.search(pattern, text)
if match:
    print("找到重复单词:", match.group(1))  # 输出: hello
```

捕获组 vs 非捕获组

有时候您需要使用圆括号进行分组，但不想捕获内容。这时可以使用非捕获组 **(?:...)**：

```
python
```

```
# 使用非捕获组
text = "123-456-7890"
pattern = r"(?:\d{3})-(\d{3})-(\d{4})"

match = re.search(pattern, text)
if match:
    print("完整匹配:", match.group(0))  # 123-456-7890
    print("第一个捕获组:", match.group(1))  # 456 (注意：现在这是第一个捕获组)
    print("第二个捕获组:", match.group(2))  # 7890
```

在这个例子中，**(?:\d{3})** 是一个非捕获组，它仍然参与匹配但不创建捕获组。

命名捕获组

您还可以给捕获组命名，使代码更清晰：

python

```
# 使用命名捕获组
text = "2023-05-15"
pattern = r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})"

match = re.search(pattern, text)
if match:
    print("年份:", match.group("year"))    # 2023
    print("月份:", match.group("month"))   # 05
    print("日期:", match.group("day"))     # 15
```

实际应用示例

提取 URL 的各个部分

python

```
url = "https://www.example.com:8080/path/to/page?query=string#fragment"
pattern =
```

```
r"(?P<protocol>https?)://(?P<host>[^:/]+)(?::(?P<port>\d+))?(?P<path>/[^?#]*)?(?:\?(?P<query>[^#]*))?(?:#(?P<fragment>.*))?"


match = re.match(pattern, url)

if match:

    print("协议:", match.group("protocol"))    # https

    print("主机:", match.group("host"))        # www.example.com

    print("端口:", match.group("port"))        # 8080

    print("路径:", match.group("path"))        # /path/to/page

    print("查询:", match.group("query"))       # query=string

    print("片段:", match.group("fragment"))    # fragment
```

总结

- **捕获组** 使用圆括号 **()** 定义，用于提取匹配文本的特定部分
- 可以通过数字（**\1**, **\2**）或名称（命名组）引用捕获组
- **非捕获组 (?:...)** 用于分组但不捕获内容
- 捕获组在文本提取、替换操作和模式匹配中非常有用
- 命名捕获组使代码更易读和维护