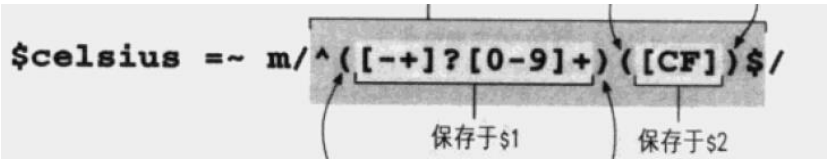
精通正则表达式读书笔记

1、 基本符号^ $ [] . \ () 等

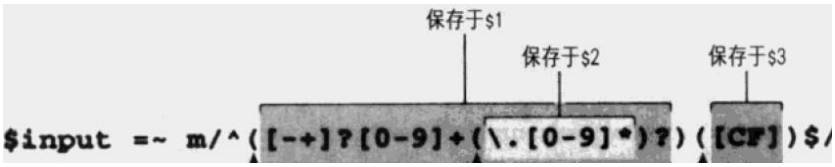
2、 ^[0-9]+$只含数字，[0-9]+含数字

1. [+-]?可以处理负数
2. [-+]?[0-9]+(\.[0-9]+)? 处理符号小数
3. 捕获性括号




```
if ($input =~ m/^([-+]?[0-9]+)([CF])$/)
{
    # 如果程序运行到此，则已经匹配，$1 保存数字，$2 保存"C"或者"F
    $InputNum = $1; #把数据保存到已命名变量中…
    $type     = $2; #…保证程序清晰易懂

    if ($type eq "C") {  # 'eq测试两个字符串是否相等
        # 输入为摄氏温度，则计算华氏温度
        $celsius = $InputNum;
        $fahrenheit = ($celsius * 9 / 5) + 32;
    } else {
        #如果不是"C"，则必然是"F"，计算摄氏温度
        $fahrenheit = $InputNum;
        $celsius = ($fahrenheit - 32) * 5 / 9;
    }
    #现在得到了两个温度值，显示结果：
    printf "%.2f C is %.2f F\n", $celsius, $fahrenheit;
```



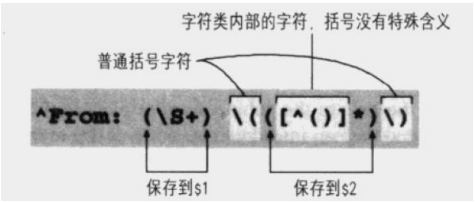
4. 加入 *来处理空格，用^\s*$来区分只有空格的行
5. [\t]*和(*|\t*)的区别：后者无法匹配 tab 和空格的混合体
6. 非捕获性括号：(?:)以？：开始的括号不被捕获
7. \s 匹配所有空白字符：空格，制表，回车，换行
8. /i 在正则尾时，正则表达式不区分大小写
9. Perl 用$variable ~= m/regex/来匹配正则表达式

\t 制表符
\n 换行符
\r 回车符
\s 任何“空白”字符（例如空格符、制表符、进纸符等）
\S 除「\s」之外的任何字符
\w 「[a-zA-Z0-9]」（在「\w+」中很有用，可以用来匹配一个单词）
\W 除「\w」之外的任何字符，也就是「[^a-zA-Z0-9]」
\d 「[0-9]」，即数字
\D 除「\d」外的任何字符，即「[^0-9]」

10. 正则表达式替换文本: $variable ~= s/regex/replacement/
11. 单词用/\bword\b/来匹配
12. s/\bJeff\b/Jeff/i：把所有 jeff（无视大小写）全部转化成 Jeff
13. 正则表达式可以实现 template 全替换
14. 所有超过 3 位的小数被砍到 3 位，最后一位如果是 0，同样砍掉

```
$price =~ s/(\.\d\d[1-9]?)\d*/$1/
```

15. 如何保存 FROM: wcyz666@126.com (wang cheng)中的邮箱和名字



16. 匹配标题

```
$line =~ m/^Subject: (.*)
```

17. 为数值添加逗号：环视，匹配位置，不匹配字符。肯定性环视(?=\d)表示当前所在位置右边一位是数字，(?<=)左边 `(?=Jeffrey)Jeff`，意思是先找到右边是 Jeffrey 的位置，再匹配 Jeff，如果找不到 Jeffrey，即使可以匹配 Jeff，也不会匹配
    1. 环视的括号分组不匹配
    2. 把所有的 Jeffs 替换成 Jeff's 的正则表达式
        1. m/Jeffs/Jeff's/
        2. m/\bJeffs\b/Jeff's/（单词分界）
        3. m/\b(Jeff)(s)\b/$1'$2/（分组匹配）
        4. m/\bJeff(?=s\b)/Jeff'/（分组，匹配位置）
        5. m/(?<=Jeff)(?=s)/'/（只匹配位置）
    3. 为数字添加逗号:左边有一个数字，右边有 3 的倍数个数字的【结尾】，如果去掉结尾，就会匹配每个左边有一个数字，右边有三个数字的位置（可以用\b 代替）

```
$pop =~ s/(?<=\d)(?=(\d\d\d)+$)/,/g;
```

    4. 不是环视的匹配会被作为最终匹配

$text =~ s/(\d)((\d\d\d)+\b)/$1,$2/g; 能够在数字中添加逗号吗?

结果并非我们的期望。得到的是类似“281,421906”的字符串。这是因为`(\d\d\d)+`匹配的数字属于最终匹配文本，所以不能作为“未匹配的”部分，供/g 的下一次匹配迭代使用。

个例子中，重新开始的起点是整个数值的末尾。使用顺序环视的意义在于，检查某个位置，但检查时匹配的字符并不算在（最终）“匹配的字符串”内。

实际上，这个表达式仍然可以用来解决这个问题，但正则表达式必须由宿主语言反复调用，例如通过一个 while 循环，每次检查的都是上次修改后的字符串。每次替换操作都会添加一个逗号（对目标字符串中的每个数值都是如此，因为/g 的存在），下面是一个例子：

```
while ( $text =~ s/(\d)((\d\d\d)+\b)/$1,$2/g ) {
      # 循环内不用进行任何操作——我们希望的是重复这个循环，直到匹配失败
}
```

5. 不用逆序环视的匹配

```
$text =~ s/(\d)(?=(\d\d\d)+(?!\d)/$1,/g;
```

6. 把短行转化成 html 语言

编码不正确可能会导致显示错误。我称这种简单的转换为“为 HTML 而加工(cooking the text for HTML)”，它的确非常简单：

```
$text =~ s/&/&amp;/g;   # 保证基本的 HTML…
$text =~ s/</&lt;/g;    # … 字符 &、<、and > …
$text =~ s/>/&gt;/g;    # … 转换后不出错
```

7. 增强的行锚点：匹配逻辑行

幸好，大多数支持正则表达式的语言提供了一个简单的办法，即“增强的行锚点”(enhanced line anchor)匹配模式，在这种模式下，「^」和「$」会从字符串模式切换到本例中需要的逻辑行模式。在 Perl 中，使用/m 修饰符来选择此模式：

```
$text =~ s/^$/<p>/mg;
```

8. 把 Email 转化成链接（注意@和/都需要转义，而且最好使用单词分隔符）

部分之前，我们先看看这个正则表达式的具体应用环境：

```
$text =~ s/\b(username regex\@hostname regex)\b/<a href="mailto:$1">$1<\/a>/g;
```

9. 注意：括号里出现了连字符的话，一定要把它放在字符组最前面

们不应该使用「\w+」，而应该用「\w[-.\w]*」。这就保证用户名以「\w」开头，后面的部分可以包括点号和连字符。(请注意，我们在字符组中把连字符排在第一位，这样就确保它们被作为连字符，而不是用来表示范围。对许多流派来说，.-\w 表示的范围肯定是错误的，它会产生一个随机的字母、数字和标点符号的集合，具体取决于程序和计算机所用的字符编码。Perl 能够正确处理.-\w，但是使用连字符时多加小心是个好习惯。)

10. 注意：\w 可能会匹配非 ASCII 码字符，两个点号中间必须要有字符。

```
「[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)」。
```

11. 注意/x 在 perl 里头表示以下意义（#表示忽略自身及自身后一直到第一个换行符的所有字符）

式做了两件简单但有意义的事情。首先，大多数空白字符会被忽略，用户能够以“宽松排列（free-format)”编排这个表达式，增强可读性。其次，它容许出现以#开头标记的注释。

12. 用正则表达式匹配 HTTP URL：@和$要转义
    注意：最后的逆序环视用于去除结尾标点。
    量词匹配会一直持续到失败为止

```
# 将 HTTP URL 转换为链接形式 ...
$text =~ s{
   \b
   # 将 URL 保存至$1 ...
   (
       http:// [-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info) \b # hostname
           (
               / [-a-z0-9_:\@&?=+,.!/~*'%\$]*    # path 不一定会出现
                 (?<![.,?!])                     # 不能以[.,?!]结尾
           )?
   )
}{<a href="$1">$1</a>}gix;
```

13. Qr：生成一个正则表达式对象，可以被复用

```
$HostnameRegex = qr/[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)/i;
```

14. 查找重复单词

```
while (< >) ❷
{
   next unless s{❸ #  (下面是正则表达式)
         ### 匹配一个单词:
         \b              # 单词的开始位置 ...
         ( [a-z]+ )     # 把读取的单词存储至$1 (和 \1)

         ### 下面是任意多的空白字符和/或 tag
         (               # 把空白保存到 $2
              (?:        #  (使用非捕获型括号)
                 \s          # 空白字符  (包括换行符，这样非常方便)
                 |           # 或者是
                 <[^>]+>     # <TAG>形式的 tag
              )+         # 至少需要出现一次，多次不受限制
         )

         ### 现在再次匹配第一个单词:
         (\1\b)          # \b 保证用来避免嵌套单词的情况，保存到$3
   # (正则表达式结束)
   }
   # 上面是正则表达式. 下面是 replacement 字符串，然后是修饰符、/i、/g 和/x
   {\e[7m$1\e[m$2\e[7m$3\e[m}igx; ❹
   s/^(?:[^\e]*\n)+//mg;   ❺  # 去掉所有未标记的行
   s/^/$ARGV: /mg;         ❻  # 在每行开头加上文件名
   print;
}
```

15. 块模式

```
$/  = ".\n"; ❶ # 设定特殊的“块模式”("chunk-mode")；一块文本的终结为点号和换
                 行符的结合体
```

因为单词重复问题必须应付单词重复位于不同行的情况，我们不能延续在 E-mail 的例子中使用的普通的按行处理的方式。在程序中使用特殊变量$/（没错，这确实是一个变量）能使用一种神奇的方式，让<>不再返回单行文字，而返回或多或少的一段文字。返回的数据仍然是一个字符串，只是这个字符串可能包含多个逻辑行。

16. JAVA：正则表达式必须以字符串的形式传入---》转义

17. Java 中一次读一段文本的方法

```
static String getPara(BufferedReader in) throws java.io.IOException
{
  StringBuffer buf = new StringBuffer();
  String line;

  while ((line = in.readLine()) != null &&
         (buf.length() == 0 || line.length() != 0))
  {
      buf.append(line + "\n");
  }
  return buf.length() == 0 ? null : buf.toString();
 }
}
```

第三章 正则表达式的流派

1. 三个主要问题
   1. 支持的元字符，以及元字符的意义【正则表达式的流派】
   2. 正则表达式与语言或工具的交互方式 interface.：起装饰性作用，描述对应编程语言中正则表达式的应用规则
   3. 正则表达式引擎如何将表达式应用到文本
2. 正则表达式的起源：1940 年神经学家对神经元的分析：正则集合
3. POSIX: 两种正则表达式的分类：基本和拓展

正则表达式特性	BREs	EREs
点号、^、$、[…]、[^…]	✓	✓
“任意数目”量词	*	*
+和?量词		+ ?
区间量词	\{*min, max*\}	{*min, max*}
分组	\(…\)	(…)
量词可否作用于括号	✓	✓
反向引用	\1 到\9	
多选结构		✓

   1. Locale 不同语言中的文字：\w
   2. Unicode
4. 流派整理

特性	Modern *grep*	Modern *egrep*	GNU Emacs	Tcl	Perl	.NET	Sun's Java package
*、^、$、[…]	✓	✓	✓	✓	✓	✓	✓
? + |	\? \+ |	? + |	? + |	? + |	? + |	? + |	? + |
分组	\(…\)	(…)	\(…\)	(…)	(…)	(…)	(…)
(?:…)					✓	✓	✓
单词分界符		\<\>	\<\>\b、\B	\n、\M、\y	\b、\B	\b、\B	\b、\B
\w、\W		✓	✓	✓	✓	✓	✓
反向引用	✓	✓	✓	✓	✓	✓	✓

✓表示支持

5. 正则表达式的三种处理方法
    1. 集成式（perl）：把正则表达式作为配置文件的一部分
    2. 面向过程式：
    3. 面向对象式：
6. 各种语言的处理技巧
    1. JAVA 的处理

```
import java.util.regex.*; // 这样使用 regex 包中的类更加容易
  ……
❶ Pattern r = Pattern.compile("^Sujbcet: (.*)", Pattern.CASE_INSENSITIVE);
❷ Matcher m = r.matcher(line);
❸ if (m.find()) {
❹     subject = m.group(1);
  }
```

❶ 检查正则表达式，将它编译为能进行不区分大小匹配的内部形式（internal form），得到一个“Pattern”对象。

❷ 将它与欲匹配的文本联系起来，得到一个“Matcher”对象。

❸ 应用这个正则表达式，检查之前与之建立联系的文本，是否存在匹配，返回结果。

❹ 如果存在匹配，提取第一个捕获括号内的子表达式匹配的文本。

Sun 有时也会把正则表达式整合到 Java 的其他部分，例如上面的例子可以使用 string 类的 matches 功能来完成：

```
if (! line.matches("\\s*", ))
{
   // ... 如果 line 不是空行 ...
}
```

同样，这种办法不如合理使用面向对象的程序有效率，所以不适宜在对时间要求很高的循环中使用，但是“随手（casual）”用起来非常方便。

    2. PHP 和 python

PHP 中的正则处理

下面是使用 PHP 的 preg 套件中的正则表达式函数处理「Subject」的例子，这是纯粹的函数式方法（第 10 章详细介绍 PHP）。

```
if (preg_match('/^Subject: (.*)/i', $line, $matches))
    $Subject = $matches[1];
```

Python 中的正则处理

最后我们来看 Python 中「Subject」的例子，Python 采用的也是面向对象式的办法。

```
import re;
  ……
R = re.compile("^Subject: (.*)", re.IGNORECASE);
M = R.search(line)
if M:
    subject = M.group(1)
```

这个例子与我们之前看过的非常类似。

7. 高级正则表达式处理：查找替换
    1. JAVA：所有的反斜线都必须转义

```
import java.util.regex.*; // 一次性导入所有需要用到的类
    ……
Pattern r = Pattern.compile(
    "\\b                                                    \n"+
    "# 把捕获的地址保存到$1 ...                             \n"+
    "(                                                      \n"+
    "  \\w[-.\\w]*                                 # username  \n"+
    "  @                                                    \n"+
    "  [-\\w]+(\\.[-\\w]+)*\\.(com|edu|info)       # hostname  \n"+
    ")                                                      \n"+
    "\\b                                                    \n",
    Pattern.CASE_INSENSITIVE|Pattern.COMMENTS);
Matcher m = r.matcher(text);
text = m.replaceAll("<a href=\"mailto:$1\">$1</a>");
```

    2. emac：正则表达式搜索前进，过于依赖反斜线
8. 字符串，字符编码和匹配模式
    1. 字符串：注意编程语言定义的元字符
        1. 字符串中，必须使用双斜线才能表达正则表达式里的反斜线
        例子：为了表示\n， 必须使用\\n

字符串文字	"[\t\x2A]"	"[\\t\\x2A]"	"\t\x2A"	"\\t\\x2A"
字符串的值	'[*]'	'[\t\x2A]'	'*'	'\t\x2A'
作为正则表达式	「[*]」	「[\t\x2A]」	「*」	「\t\x2A」
能够匹配	星号或制表符	星号或制表符	任意数目的制表符	制表符和之后的星号
在/x 模式下	星号或制表符	星号或制表符	错误	制表符和之后的星号

        2. JAVA/C#：反斜线转义字符如果不存在会报错，例如\w【必须使用\\w】
        3. php：1. 无法识别的反斜线序列会被原封不动传入正则表达式
            2. 单引号字符串只有\\和\'，其他所有字符都不被识别为特殊字符
        4. python：三重引用等同于 php 的单引号字符
        5. perl：特殊特性
            1. 变量插值
            2. 通过\Q..\E 支持文字文本
            3. 能够支持\N{NAME}结构-》 用正式的 Unicode 名来指定字符
    2. 字符编码
        1. Unicode：字符所对应的数字，称为代码点。韩语 xx 代码点是 U+C0B5
        2. 编码方式多种多样：UTF8，UTF16 等
        3. 支持 Unicode 的正则表达式通常支持**\unum**【U+FFFF 以后的代码需要\u{num}】来支持
        4. 字符还是组合字符序列：某些带符号的音有两个代码点组成
            1. 点号应该匹配这种字符吗？
            2. 量词是针对整个字符还是代码点
            3. Java：CANON_EQ，能够匹配规则上等价的字符
        5. Unicode 的行终止符
    3. 正则模式和匹配模式

1. 不区分大小写的匹配
    1. 有些没大写/大小写不是一对一关系，只有 Perl 能处理
2. 宽松模式与注释模式：会忽略字符组以外的所有·空白字符
3. 点号通配模式：点号不受限制，可以匹配任意字符，包括换行符
4、增强的行锚点：改变了^和$的匹配方式

支持此模式的程序通常还提供了「\A」和「\Z」，它们的作用与普通的「^」和「$」一样，只是在此模式下它们的意义不会发生变化。也就是说「\A」和「\Z」永远不会匹配字符串内部的换行符。有些实现方式中，「$」和「\Z」能够匹配字符串内部的换行符，不过它们通常会提供「\z」，唯一匹配整个字符串的结尾位置。详见 129 页。

5. 文本模式：类似非正则表达式的直接搜索。可与正则表达式混用

9. 特殊字符
    1. 编程语言中的保留字符

\a **警报**（例如，在“打印”时扬声器发声）。通常对应 ASCII 中的<BEL>字符，八进制编码 007。

\b **退格** 通常对应 ASCII 中的<BS>字符，八进制编码 010。(在许多流派中，「\b」只有在字符组内部才表示这样的意义，否则代表单词分界符☛133)。

\e Escape **字符** 通常对应 ASCII 中的<ESC>字符，八进制编码 033。

\f **进纸符** 通常对应 ASCII 中的<FF>字符，八进制编码 014。

\n **换行符** 出现在几乎所有平台（包括 Unix 和 DOS/Windows）上，通常对应 ASCII 的<LF>字符，八进制编码 012。在 MacOS 中通常对应 ASCII 的<CR>字符，十进制编码 015。在 Java 或任意一种.NET 语言中，不论采用什么平台，都对应 ASCII<LF>字符。

\r **回车** 通常对应 ASCII 的<CR>字符。在 MacOS 中，对应到 ASCII 的<LF>字符。在 Java 或任意一种.NET 语言中，不论采用什么平台，都对应到 ASCII 的<CR>字符。

\t **水平制表符** 对应 ASCII 的<HT>字符，八进制编码 011。

\v **垂直制表符** 对应 ASCII 的<VT>字符，八进制编码 013。

2. 注意：**匹配 HTTP 中的字符时，应使用\015\012 来匹配换行和回车，匹配 Unix 和 Windows 的换行时，应使用\015?\012**
3. 八进制转义：不要超过\377
4. 十六进制与 Unicode 转义

\x*num*、\x{*num*}、\u*num*、\U*num*

5. 控制字符：\c**char** 匹配编码值小于 32 的控制字符，**推荐只使用大写**
    例子：\cH 表示 ctrl+H，退格键
6. 字符组
    1. 在字符组内部，*从来不是元字符，而-基本都是
    2. 使用范围表示效率较高
    3. 没有量词的字符组是肯定断言，必须匹配一个字符
    4. 用[a-Z]不如使用[a-zA-Z]
    5. 几乎能匹配任何字符的点号

在 Sun 的 Java regex package 之类的支持 Unicode 的系统中，点号不能匹配 Unicode 的行终结符（☛109）。

匹配模式（☛111）会改变点号的匹配规则。

POSIX 规定，点号不能匹配 NUL（值为 0 的字符），尽管大多数脚本语言容许文本中出现 NULL（而且可以用点号来匹配）。

表 3-7：部分工具软件及它们的正则表达式支持的八进制和十六进制转义

	反向引用	八进制转义	十六进制转义
Python	✓	\0、\07、\377	\xFF
Tcl	✓	\0、\77、\377	\x…\uFFFF；\UFFFFFFFF
Perl	✓	\0、\77、\377	\xF；\xFF；\x{…}
Java	✓	\07、\77、\0377	\xFF；\uFFFF
GNU awk		\7、\77、\377	\x…
GNU sed	✓		
GNU Emacs	✓		
.NET	✓	\0、\77、\377	\xFF；\uFFFF
PHP（preg 套件）	✓	\0、\77、\377	\xF，\xFF，\x{…}
MySQL			
GNU egrep	✓		
GEU grep	✓		
flex		\7、\77、\377	\xF；\xFF
Ruby	✓	\7、\77、\377	\xF；\xFF

\0——「\0」匹配字节 NUL，而其他一位数字的八进制转义是不支持的。

\7，\77——一位和两位八进制转义都支持
\07——支持开头为 0 的两位八进制转义
\077 ——支持开头为 0 的 3 位八进制转义
\377 ——支持不超过\377 的 3 位八进制转义
\0377 ——支持不超过\0377 的 4 位八进制转义
\777 ——支持不超过\777 的 3 位八进制转义
\x… ——容许出现任意多位数字
\x{…} ——\x{…}容许出现任意多位数字
\xF、\xFF ——以\x 开头，容许出现一到两位十六进制转义
\uFFFF ——以\u 开头的 4 位十六进制转义
\UFFFF ——以\U 开头的 4 位十六进制转义
\UFFFFFFFF ——以\U 开头的 8 位十六进制数字（参考第 91 页的版本信息）

6. 单个字节：\c 只匹配一个字符，需要小心使用
7. 匹配字符：\X 匹配一个 Unicode 字，不论由多少个代码点组成
   与点号的区别：1. \X 可以匹配所有 Unicode 换行符，点号只能在特定模式下匹配　　2. \X 不能匹配以组合字符开头的字符
8. 字符组简记法

\d **数字** 等价于「[0-9]」，如果工具软件支持 Unicode，能匹配所有的 Unicode 数字。

\D **非数字字符** 等价于「[^\d]」。

\w **单词中的字符** 一般等价于「[a-zA-Z0-9_]」。某些工具软件中「\w」不能匹配下画线，而另一些工具软件的「\w」则能支持当前 locale（☛87）中的所有数字和字符。如果支持 Unicode，「\w」通常能表示所有数字和字符，而在 java.util.regex 和 PCRE（也包括 PHP）中，「\w」严格等价于「[a-zA-Z0-9_]」。

\W **非单词字符** 等价于「[^\w]」。

\s **空白字符** 在支持 ASCII 的系统中，它通常等价于「[·\f\n\r\t\v]」。在支持 Unicode 的系统中，有时包含 Unicode 的“换行”控制字符 U+0085，有时包含“空白（whitespace）”属性\p{Z}（参见下一节的介绍）。

\S **非空白字符** 等价于「[^\s]」。

9. Unicode 属性、字母表和区块

1. Unicode 不仅是一套字符映射规律，还定义了每个字符的性质，使用**\p{Prop}**和**\P{Prop}**来选择。普通性质由单字符选择，也可以使用多个字符来表示
2. 有些系统要求使用前缀 In 或者 Is 来匹配属性
3. 有些系统支持特殊的复合模式，例如**\p{L&}**表示分大小写的字符，即[\p{Lu}\p{Lt}\p{Ll}]

分 类	等价表示法及描述
\p{L}	\p{Letter}——字母
\p{M}	\p{Mark}——不能单独出现，而必须与其他基本字符一起出现（重音符号、包围框，等等）的字符
\p{Z}	\p{Separator}——用于表示分隔，但本身不可见的字符（各种空白字符）
\p{S}	\p{Symbol}——各种图形符号（Dingdats）和字母符号
\p{N}	\p{Number}——任何数字字符
\p{P}	\p{Punctuation}——标点字符
\p{C}	\p{Other}——匹配其他任何字符（很少用于正常字符）

4. 字母表。有些系统能够按照字母表匹配。如**\p{Hebrew}**表示匹配希伯来文。字母表不会包含该书写系统的所有字符（如标点），只会匹配独立隶属于此书写系统的字符。空格和标点用伪字符表匹配，如**\p{IsCommon}**
5. 区块。表示 Unicode 字符映射表中一定范围的代码点，例如**\p{InTibetan}**表示 U+0F00 到 u+0FFF 这 256 个代码点
6. 区块与字母表的区别
    1. 区块可能包含未赋值的代码点（西藏文区块 25%未赋值）
    2. 并不是所有看起来应该在区块的字符都在区块里
    3. 区块会包含不相关的字符
    4. 属于同一个字母表的字符可能在多个区块里
7. 前缀 Is/没有前缀：在 perl 和 java 里表示字母表

前缀 In：在 perl 和 java 里表示区块，在.NET 里表示字母表

表 3-9：基本的 Unicode 子属性

属　性	等价表示法及说明
\p{Ll}	\p{Lowercase_Letter}——小写字母。
\p{Lu}	\p{Uppercase_Letter}——大写字母。
\p{Lt}	\p{Titlecase_Letter}——出现在单词开头的字母（例如，字符 Dž是小写字母 dž和大写字母 Dž的首字母形式）。
\p{L&}	\p{Ll}、\p{Lu}、\p{Lt}并集的简记法。
\p{Lm}	\p{Modifier_Letter}——少数形似字母的，有特殊用途的字符。
\p{Lo}	\p{Other_Letter}——没有大小写形式，也不属于修饰符的字母，包括希伯来语、阿拉伯语、孟加拉语、泰国语、日语中的字母。
\p{Mn}	\p{Non_Spacing_Mark}——用于修饰其他字符的“字符（Characters）”，例如重音符号、变音符号、某些“元音记号”和语调标记。
\p{Mc}	\p{Spacing_Combining_Mark}——会占据一定宽度的修饰字符（各种语言中的大多数“元音记号”，这些语言包括孟加拉语、印度古哈拉地语、泰米尔语、泰卢固语、埃纳德语、马来语、僧伽罗语、缅甸语和高棉语）。
\p{Me}	\p{Encolsing_Mark}——可以围住其他字符的标记，例如圆圈、方框、钻石型等。
\p{Zs}	\p{Space_Separator}——各种空白字符，例如空格符、不间断空格（non-break space），以及各种固定宽度的空白字符。
\p{Zl}	\p{Line_Separator}——LINE SEPARATOR 字符（U+2028）。
\p{Zp}	\p{Paragraph_Separator}——PARAGRAPH SEPARATOR 字符（U+2029）。
\p{Sm}	\p{Math_Symbol}——数学符号、+、÷、表示分数的横线。
\p{Sc}	\p{Currency_Symbol}——货币符号、$、¢、¥、…。
\p{Sk}	\p{Modifier_Symbol}——大多数版本中它表示组合字符，但是作为功能完整的字符，它们有自己的意义。
\p{So}	\p{Other_Symbol}——各种印刷符号、框图符号、盲文符号，以及非字母形式的中文字符，等等。
\p{Nd}	\p{Decimal_Digit_Number}——各种字母表中从 0 到 9 的数字（不包括中文、日文和韩文）。
\p{Nl}	\p{Letter_Number}——几乎所有的罗马数字。
\p{No}	\p{Other_Number}——作为加密符号（superscripts）和记号的数字，非阿拉伯数字的数字表示字符（不包括中文、日文、韩文中的字符）。
\p{Pd}	\p{Dash_Punctuation}——各种格式的连字符（hyphen）和短划线（dash）。
\p{Ps}	\p{Open_Punctuation}——(、﴾和《等字符。
\p{Pe}	\p{Close_Punctuation}——)、﴿和》等字符。
\p{Pi}	\p{Initial_Punctuation}——«、“、<等字符。
\p{Pf}	\p{Final_Punctuation}——»、’、>等字符。
\p{Pc}	\p{Connector_Punctuation}——少数有特殊语法含义的标点，如下画线。
\p{Po}	\p{Other_Punctuation}——用于表示其他所有标点字符：!、&、.、:、⋮等。
\p{Cc}	\p{Control}——ASCII 和 Latin-1 编码中的控制字符（TAB、LF、CR 等）。
\p{Cf}	\p{Format}——用于表示格式的不可见字符。
\p{Co}	\p{Private_Use}——分配与私人用途的代码点（例如公司的 logo）。
\p{Cn}	\p{Unassigned}——目前尚未分配字符的代码点。

表 3-10：属性/字母表/区块的支持情况

特　性	Perl	Java	.NET	PHP/PCRE
✓基本属性，例如\p{L}	✓	✓	✓	✓
✓基本属性缩略表示法，例如\pL	✓	✓		✓
基本属性缩略表示法，例如\p{IsL}	✓	✓		
✓基本属性的全名，例如\p{Letter}	✓			
✓复合属性，例如\p{L&}	✓			✓
✓字母表，例如\p{Greek}	✓			✓
字母表全名，例如\p{**Is**Greek}	✓			
✓区块，例如\p{Cyrillic}	如果没有字母	✓		
✓区块全名，例如\p{**In**Cyrillic}	✓	✓		
区块全名，例如\p{**Is**Cyrillic}			✓	
✓排除功能，例如 P{…}	✓	✓	✓	✓
排除功能，例如\p{^…}	✓			✓
✓\p{Any}	✓	等于\p{all}		✓
✓\p{Assigned}	✓	等于\P{Cn}	等于\P{Cn}	等于\P{Cn}
✓\p{Unassigned}	✓	等于\p{Cn}	等于\p{Cn}	等于\p{Cn}
以✓开头的行是新实现方式中推荐的用法（请参考第 91 页的版本信息）				

10. 字符组

1. 字符组减法:**[[a-z] – [aeiou]]（辅音）**

.NET 允许使用减法来表示特定的字符组

2. 完整字符组集合运算：**[[a-z] && [^aeiou]]（与减法等同）**

例子：**[\p{InThai} && \P{Cn}]**

注意：第二个是大写 P，匹配 Thai 区块中已经赋值的代码点

环视功能模拟: **(?!\p{Cn})\p{InThai}**

3. POSIX 字符组方括号表示法

1. **[[:lower:]]**比传统[a-z]更好用，因为能包含当前定义的 locale 小写

[:alnum:]	字母字符和数字字符。
[:alpha:]	字母。
[:blank:]	空格和制表符。
[:cntrl:]	控制字符。
[:digit:]	数字。
[:graph:]	非空字符（即空白字符，控制字符之外的字符）。
[:lower:]	小写字母。
[:print:]	类似[:graph:]，但是包含空白字符。
[:punct:]	标点符号。
[:space:]	所有的空白字符（[:blank:]、换行符、回车符及其他）。
[:upper:]	大写字母。
[:xdigit:]	十六进制中容许出现的数字（例如 0-9a-fA-F）。

4. POSIX “collating”序列方括号表示法：[[.span-ll.]]
    1. collating 序列会把多个实体字符映射到单个逻辑字符，比如西班牙语 ll，会被[^abc]匹配。Collating 序列因此可以匹配多个实体字符
5. POSIX “字符等价类”方括号表示法：[[=n=]]
    1. 表示当前 locale 下的语言系统里所有与 n 等价的字母
    2. 如果没有特殊的等价类，则[[=a=][=n=]]就是[an]
6. Emacs 使用特殊的语法类，\sw 匹配单词字符，\s-匹配空白字符，字符组的字符可以根据所编辑文本的变化而变化

11. 锚点和其他零长度断言
    1. 行/字符串的起始位置：^和\A
        1. \A 总是能匹配待搜索文本的起始位置
        2. 使用了增强的行锚点匹配模式后，^能匹配每个换行符后的位置
    2. 行/字符串的结束位置：$, \Z 和\z
        1. $的意义
            1. 匹配目标字符串的末尾
            2. 匹配整个字符串末尾的换行符之前的位置：**s$**匹配以 s 结尾的行
            3. 匹配目标文本的结束位置
            4. 匹配任何一个换行符之前的位置
        2. \Z 表示未指定任何模式下$所匹配的字符，通常是字符串的末尾位置，或者字符串末尾的换行符之前的位置
        3. \z 匹配字符串的末尾，不考虑换行符
    3. 匹配的起始位置：\G
        1. 对迭代操作非常有用，第一次迭代时匹配开头，和\A 一样
        2. 匹配不成功，则重新指向字符串开头，不影响之后的操作
        3. 优良性质
            1. \G 指向的是每个目标字符串的属性，可以被多个正则应用
            2. perl 的/c 修饰符：匹配失败时，不重新设置\G，保持不变
            用法：在某个位置开始尝试使用多个正则表达式匹配
            3. \G 对应的属性可以使用与正则表达式无关的结构修改
        4. 在 perl 里，必须出现在正则表达式的开头
        5. 之前匹配的结束位置不等于当前匹配的开始位置
    4. 单词分解符：\b \B \< \>： 不做语义分析，例如：M.I.T.
        理解：位置一边是单词字符，一边是非单词字符（注意 Unicode）
    5. 环视
        1. 逆序环视的长度限制
            1. 只能匹配固定长度：Python 和 Perl
            2. 允许出现长度不同的分支：PHP
            3. 支持任意长度的文本，不能无限：java
            4. 支持匹配无限长度文本：.NET，会带来效率问题
    6. 注释和模式修饰符
        1. (?i)开启不分大小写的匹配，用(?-i)关闭:**(?i)very(?-i)**
        2. 不支持(?-i)：使用括号: **((?i)very)**

```
my $need_close_anchor = 0; # 如果遇见了<A>而没有对应的</A>，则返回True

while (not $html =~ m/\G\z/gc) # 在整个字符串没有处理完之前
{
  if ($html =~ m/\G(\w+)/gc) {
    ...如果$1中包含数字或单词——可以检查语言的规范性...
  } elsif ($html =~ m/\G[^<>&\w]+/gc) {
    #  其他非HTML代码——无关紧要
  } elsif ($html =~ m/\G<img\s+([^>]+)>/gci) {
    ...包含image tag——可以检查它是否符合规范...

  } elsif (not $need_close_anchor and $html =~ m/\G<A\s+([^>]+)>/gci) {
    ...包含超链接，这里可以进行验证...

    $need_close_anchor = 1; # 我们现在需要的是</A>
  } elsif ($need_close_anchor and $html =~ m{\G</A>}gci){
    $need_close_anchor = 0; # 需求已经满足，不再容许出现
  } elsif ($html =~ m/\G&(#\d+<\w+);/gc){
    # 容许出现&gt;和&#123;之类的entity
  } else {# 此处完全无法匹配，必然有错误。记下当前位置，从HTML中提取若干字符，报告错误
    my $location = pos($html); # 记下这段HTML的起始位置
    my ($badstuff) = $html =~ m/\G(.{1,12})/s;
    die "Unexpected HTML at position $location: $badstuff\n";
  }
}

# 确保没有孤立的 <A>
if ($need_close_anchor) {
```

3. 其他修饰符

字 母	模 式
i	不区分大小写的匹配模式 (☞110)
x	宽松排列和注释模式 (☞111)
s	点号通配模式 (☞111)
m	增强的行锚点模式 (☞112)

4. 注释：使用(?#)来添加注释
5. 文本文字范围：\Q…\E：消除除了\E 以外所有的元字符含义
6. 用 Regex.Escape(.NET)等函数消除用户输入的元字符

7. 分组，捕获，条件判断和控制
   1. 非捕获性括号的作用
      1. 把复杂的表达式变清晰
      2. 提高效率
      3. 利用多个成分构建正则表达式

程　序	完整的匹配	第一组括号匹配的文本
GNU *egrep*	N/A	N/A
GNU Emacs	(match_string 0) (replacement 字符串中为\&)	(match-string 1) (replacement 字符串中为\1)
GNU awk	Substr($*text*, RSTART, RLENGTH) (replacement 字符串中为\&)	\1 (在 gensub 替换中)
MySQL	N/A	N/A
Perl ☞41	$&	$1
PHP ☞450	$*matches*[0]	$*matches*[1]
Python ☞97	*MatchObj*.group(0)	*MatchObj*.group(1)
Ruby	$&	$1
GNU sed	& (只能在 replacement 字符串中使用)	\1(只能出现在 regex 和 replacement 中)
Java ☞95	*MatcherObj*.group()	*MatcherObj*.group(1)
Tcl	通过 regexp 命令设置为用户选择的变量	
VB.NET ☞96	*MatchObj*.Groups(0)	*MatchObj*.Groups(1)
C#	*MatchObj*.Gropus[0]	*MatchObj*.Groups[1]
vi	&	\1

(请参考第 91 页的版本信息)

2. 捕获命名(?<Name>)
    1. perl: <?P<name>…>

```
「\b(?P<Area>\d\d\d\)-(?P<Exch>\d\d\d)-(?P<Num>\d\d\d\d)\b」
```

们可以通过名称来访问各个括号捕获的内容，例如在 VB.NET 和大多数.NET 语言中，可以使用 *RegexObj*.Groups("Area")，在 C#中使用 *RegexObj*.Groups["Area"]，在 Python 中使用 *RegexObj*.group("Area")，在 PHP 中使用$*matches*["Area"]。这样程序看起来更清

3. 固化分组(?>)：被匹配的内容不会返还

「¡.* !」能够匹配 '¡Hola!'，但是如果「.*」在固化分组「¡(?>.*)!」中就无法匹配。在这两种情况下，「.*」都会首先匹配尽可能多的内容（'¡Hola!'），但是之后的「!」无法匹配，会强迫「.*」释放之前匹配的某些内容（最后的'!'）。如果使用了固化分组，就无法实现，因为「.*」在固化分组中，它永远也不会"交还"已经匹配的任何内容。

4. 多选结构：竖杠的优先级很低
    1. 大多数流派允许空匹配（POSIX 标准不允许）
5. 条件分支
    1. (<)?\w+(?(1)>)这个正则表达式中，第一个问号必须在括号外，因为只要参与了匹配就返回 true，无论是否匹配成功。
6. 区间：{min, max}和\{min, max\}
    1. 计数量词，通过区间来指定匹配的次数
    2. X{0,0}不代表 X 不出现，而是代表没有这个匹配。不出现应使用否定环视
7. 忽略优先量词：*?, ??, {min, max}?, +?
    匹配尽可能少的字符，和正常量词相反（greedy）
8. 占有优先量词：*+, ++, ?+等
    与固定分组基本相同

第四章 正则表达式的匹配原理

1. 正则引擎的分类：DFA 和 NFA

引擎类型	程 序
DFA	awk（大多数版本）、*egrep*（大多数版本）、*flex*、*lex*、MySQL、Procmail
传统型 NFA	GNU Emacs、Java、*grep*（大多数版本）、*less*、*more*、.NET 语言、PCRE library、Perl、PHP（所有三套正则库）、Python、Ruby、sed（大多数版本）、*vi*
POSIX NFA	*mawk*、Mortice Kern Systems' utilities、GNU Emacs （明确指定时使用）
DFA/NFA 混合	GNU awk、GNU *grep/egrep*、Tcl

2. 粗略分类：
    1. DFA，无论是否符合 POSIX 标准
        不支持忽略优先量词，捕获性括号，反向引用和回溯
    2. 传统型 NFA：支持忽略优先量词，
    3. POSIX NFA
3. 匹配的基础
    1. 优先选择最左边的匹配结果
    2. 标准的匹配量词+, ?, *, {min, max}是匹配优先的
4. 规则 1：优先选择最左边的匹配结果
    1. 正则表达式只关心是否匹配，而不是在哪里匹配
    2. 传动装置和驱动装置
        1. 传动装置：如果引擎不能在字符串开始的位置找到匹配结果，传动装置就会推动引擎前进到下一个位置尝试
        2. 驱动装置【引擎】的构造
            1. 文字文本：非元字符的文本只考虑是否相同
            2. 点号，字符组，Unicode 属性：只匹配一个字符
            3. 捕获性括号：不影响
            4. 简单锚点：只检查特定位置
            5. 复杂锚点：环视
            6. 分组括号，反向引用和忽略优先量词：DFA 不支持，但是 DFA 匹配快
5. 规则 2：标准量词是匹配优先的
    1. 匹配结果并非是最长的，但标准量词总是尝试匹配最长的字符，直到匹配上线
    2. 后续部分无法匹配时，引擎会强迫量词交还 unmatch 它所匹配的字符
6. 表达式主导：NFA 引擎
    1. 表达式的控制权在各个自表达式间来回转换
    2. 正则表达式的结构控制了整个匹配过程。
    3. 用户可以针对正则表达式的结构做优化
7. 文本主导：DFA 引擎（有限状态机引擎）
    1. 会记录当前有效的所有匹配可能

字符串中的位置	正则表达式中的位置
after…t▲onight…	可能的匹配位置：「t▲o(nite|knight|night)」

接下来扫描的每个字符，都会更新当前的可能匹配序列。继续扫描两个字符以后的情况是：

字符串中的位置	正则表达式中的位置
after…toni▲ght…	可能的匹配位置：「to(ni▲te|knight|ni▲ght)」

2. 扫描的每个字符串的每个字符都对引擎进行了控制（有限状态机接收字符换状态）

3. 如果引擎发现文本中出现的某个字符会令匹配失效，就会返回某个之前的完整匹配。如果不存在这样的匹配，则报告无法匹配

8. 比较 NFA 和 DFA

1. DFA 更快，因为 NFA 需要对相同的文本尝试不同的表达式

2. NFA 在抵达正则表达式末尾前都不知道全局匹配是否成功。DFA 是确定型的，每个字符只被检查一遍

3. DFA 的特征：匹配迅速，一致，不宜谈论

9. 回溯

1. 定义：正则表达式回一次处理各个子表达式或组成元素，遇到需要在两个可能成功的分支做出选择时，会选择其一，并记住另一个。失败时，引擎会回溯到最近的备选分支继续尝试

2. 要点

1. 对于匹配优先量词，引擎会优先选择“进行尝试”；对于忽略优先量词，则会选择跳过尝试

2. 距离当前最近储存的选项就是失败时回溯的选项

3. 例子

进行了回溯的匹配

如果需要匹配的文本是 'ac'，在尝试「b」之前，一切都与之前的过程相同。显然，这次「b」无法匹配。也就是说，对「…?」进行尝试的路走不通。因为有一个备用状态，这个“局部匹配失败”并不会导致整体匹配失败。引擎会进行回溯，也就是说，把“当前状态”切换为最近保存的状态。在本例中，情况就是：

'a▲c'	「ab?▲c」

在「b」尝试之前保存的尚未尝试的选项。这时候，「c」可以匹配 c，所以整个匹配宣告完成。

不成功的匹配

现在，我们用同样的表达式匹配 'abX'。在尝试「b」以前，因为存在问号，保存了这个备用状态：

'a▲bX'	「ab?▲c」

3. 回溯与匹配优先

1. 星号，加号及其回溯：每次测试星号作用的元素前，引擎都会保存一个状态。如果测试失败，还能够从保存的状态开始匹配回溯，直到成功或者所有都失败

2. 例子：a 1234 num 用[0-9]*匹配，会保留 'a•▲1234•num' 状态吗

答案：不会，在 a 之前就匹配成功了

3. 注意：

1. 回溯机制不但要重新计算正则表达式和文本的相对位置，也需要维护括号里的子表达式所匹配的文本状态，每次回溯都会把当前状态中正则表达式的对应位置指向括号【或者最近的备用状态位置】前.

2. 有星号或者其他匹配量词限定的部分不受后面元素影响，只是匹配尽可能多的内容。在随后的匹配中，非固定分组匹配到的字符可能会被强制交还

4. 例子：双引号匹配

   The name “wcyz666” is the username of “wang cheng”

   使用**”.*”**匹配结果**“wcyz666” is the username of “wang cheng”（匹配尽可能多）**

   如果想匹配严格单引号中的，应使用**”[^”]*”（防止匹配换行符”[“\n]*”）**

   对于所有配对的单字符，都应该使用这种方法

4. 环视与忽略量词优先

   1. 某些支持忽略优先量词的 NFA 引擎可以选择忽略量词优先

      “.*?”只会匹配最短的两个引号之间的字符，相同于**”[^”]*“**

   2. 排除环视：能得到与排除字符组相同的效果

5. 例子：多字符匹配（例如 HTML 标签匹配）

   <a>hello<a>world

   使用<a>.*会匹配所有字符

   解决方法 1：忽略优先量词**<a>.*?**，可能会匹配<a>hello<a>world

   解决方法 2：否定环视

```
<B>                     #  匹配开头的<B>
(                       #  然后匹配尽可能多的内容
   (?! </? B>)          #  如果不是<B>，也不是</B> ...
   .                    #  ... 任何字符都可以
)*                      #   (现在是匹配优先的量词)
</B>                    #  <ANNO> ... 直到结束分隔符匹配
```

6. 匹配优先与忽略优先

   1. 只是尝试路径的顺序不一样
   2. 如果没有匹配，则两种方式都会使引擎报告匹配失败
   3. 如果只有一种匹配可能，两种方式都会找到这种可能，只是失败次数不同
   4. 如果有多种匹配，则结果可能不同

7. 占有优先量词和固定分组

   1. 希望某个可选元素已经匹配成功之后，放弃此元素的所有备用状态

   2. (?>)固定分组：锁定已经匹配的分组，放弃括号内所有的备用状态，但是回溯到括号之前时，会交还已经匹配的字符

   3. 匹配优先和占有优先都不会影响路径本身，而只会影响监测顺序，但是固化分组放弃的某些路径，因此会影响最终结果

      1. 毫无影响
      2. 导致匹配失败
      3. 改变匹配结果
      4. 加快报告匹配失败的速度(?>\w+):

   4. 例子：(?>.*?)不会匹配任何字符，相当于没有

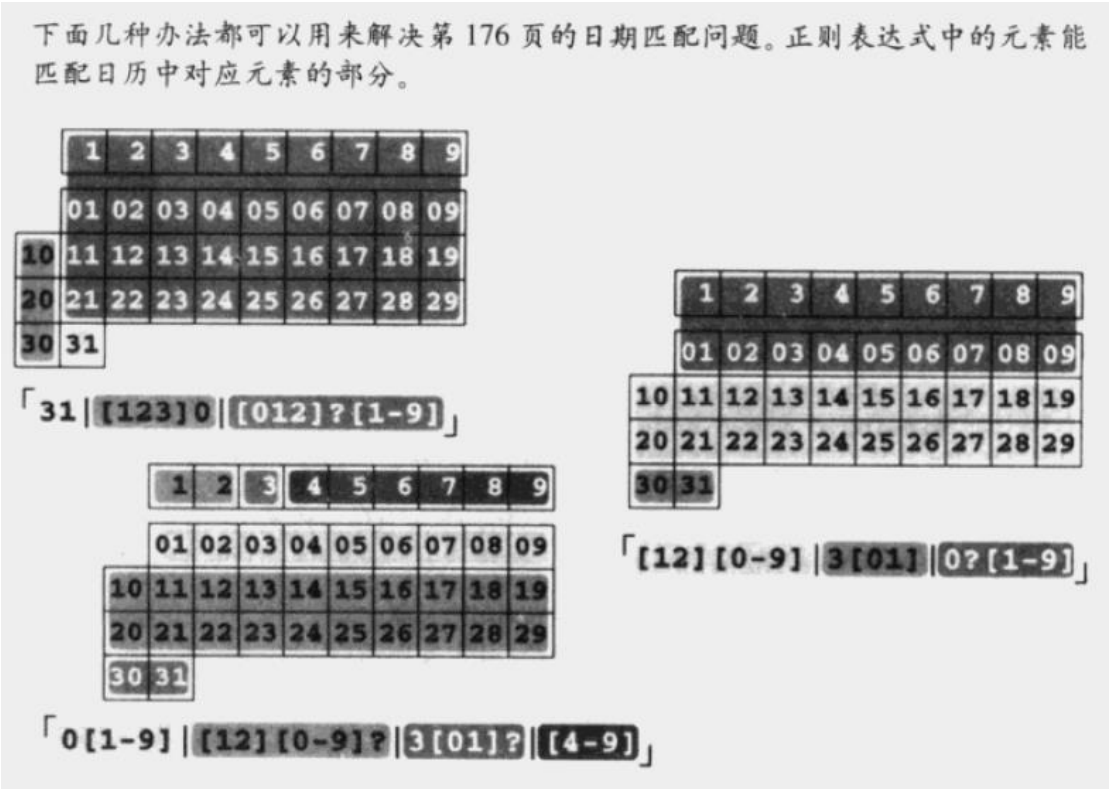
      例子：(?>M+)和(?>M)+的区别。前者放弃的状态有意义，后者 M 不会产生任何备用状态，没有意义

   5. 可以使用固化分组加快匹配速度，放弃备用状态【极其有价值】

   6. 占有优先量词++，?+ *+等

      与固化分组转化：占有优先量词去掉加号，剩下的部分用括号包裹

8. 匹配优先和忽略优先都期望获得匹配
    1. 例子：给数字加逗号分界符。(\.\d\d[1-9]?\d*)
    2. 更有效率的版本：只匹配超过 3 位和第 3 位是 0 的数字
        解决方案 1：(\.\d\d[1-9]?\d+)。不行，会匹配并截断三位且第三位不是 0 的数字，因为\d+会强制?交还匹配的字符
        解决方案 2：忽略优先，也不能解决问题。
        解决方案 3：**(\.\d\d(?>[1-9]?)\d+)**可解决问题
9. 环视中的回溯：表达式能否在当前位置匹配前后的文本
    1. 环视会保留自己的备用状态和进行独立的子回溯
    2. 回溯时只能选择自己的备用状态，如果引擎发现环视需要回退到外部世界的状态，则认为环视匹配失败。
    3. 环视匹配结束后，放弃一切自身的备用状态
10. 用肯定环视模拟固化分组：**(?>\w+):可以转化成(?=(\w+))\1:**
    原因：环视匹配了足够多的字符后，报告成功。同时丢弃所有备份状态。
11. 多选结构
    1. 传统 NFA，是按分支的顺序选择并报告第一个匹配的分支
    2. DFA 和 POSIX NFA 是按照最长匹配报告的
    3. 如果多选分支的第一个总是能匹配，后面的分支就没有价值
        例子：a((ab)*|b*)
    4. 注意谨慎排序：例子，匹配日期



4. NFA, DFA 和 POSIX
    1. 最左最长规则：DFA 会选择从最左边开始，最长的匹配
    2. 使用 ^\w+=.*(\\\n.*)* 能匹配多个逻辑行吗？答案：不行，因为.*会匹配斜杠，而表达式后面的部分不会主动回溯。（DFA 可以匹配）

忽略优先匹配 ^\w+=.*?(\\\n.*?)* 也不可以，因为整个表达式都没有强制要求匹配，所以会直接回溯

3. POSIX NFA：传统型 NFA 穷尽所有回溯并返回最长的匹配

5. 速度和效率

1. NFA：

1. 尝试匹配开始之前会先建立一个有限状态机，针对输入跳转的相应的状态
2. 在报告无法匹配前，需要穷尽所有变体
3. NFA 独有的性质

1. 捕获性括号和它的后续应用
2. 环视
3. 非匹配优先量词和有序的多选分组
4. 占有优先量词和固化分组

2. POSIX NFA 和 DFA：

1. 建立一个内化形式，NFA 需要的资源较少
2. POSIX NFA 必须穷尽所有变体，需要优化。
3. DFA 不需要太多的优化

第五章 正则表达式使用技巧

1. 好的正则表达式的特征
   1. 只匹配期望的文本，忽略不期望的文本
   2. 必须易于理解和控制
   3. 如果使用 NFA，必须保证效率（匹配就迅速返回结果，不匹配则迅速返回失败）

2. 若干的例子
   1. **经验 1： 如果不需要点号匹配反斜线，就应该在正则表达式里说明**
      匹配逻辑行：（把.*换成[^\\\n]*）^\w+=([^\\\n]*)(\\\n[^\\\n]*)*，要求反斜线不能出现在句中，指导思想是匹配一行，如果还有其他行，继续匹配
      另一种思路：匹配要么是正常的字符，要么跟着反斜线的其他符号
      正则表达式：^\w+=([^\\\n] | \\.)*
   2. **经验 2： 避免多选**
   3. **经验 3： 关注每一位上应该出现什么数字**
      匹配 IP 地址：^(\d\d? | [01]\d\d | 2[0-4]\d | 25[0-6])\. (\d\d? | [01]\d\d | 2[0-4]\d | 25[0-6])\. (\d\d? | [01]\d\d | 2[0-4]\d | 25[0-6])\. (\d\d? | [01]\d\d | 2[0-4]\d | 25[0-6]) （技巧：匹配两位数字应该使用**\d\d?** 这样能更快报告错误）
      确认 IP 地址两侧需要空格
   4. **经验 4： 确定应用场合**
   5. **经验 5： 时常想想匹配失败的情况。**
   6. 处理文件名：
      a) 匹配文件名开头的路径：^.*/ Unix “^.*\\” 或者 “^.*\\\\” Windows. **注意四个反斜线，因为反斜线也需要转义**
      b) 经过合理优化的引擎，会主动在.*前加入^
      c) 从路径中获取文件名: [^/]*$. **注意这个表达式总能匹配（虽然效率低但是文件名短，并不碍事）**

d) 把路径名和文件名分开: ^(.*)/([^/]*)$ **注意字符串如果没有斜线就不能匹配**

e) 匹配对称的括号

i. \(.*\) 匹配最外层的括号+括号内字符

ii. \([^)]*\) 匹配开括号到**最近的闭括号**

iii. \[^()]*(\([^()]*\)[^()]*)*\) **匹配单层嵌套括号**

7. **经验 6： 防备不期望的匹配**

a) **经验 7：如果某个元素的匹配没有规定任何必须出现的字符，那么他总能匹配成功**

b) 匹配一个小数或者浮点数：-?([0-9]+(\.[0-9]+)? | \.[0-9]+) **会匹配 2004.3.4**

c) 匹配分隔符里的内容：带有被转义分隔符的情况（注意多重转义符号）
“(\\. | [^\\”])*)” **注意各个可选分支不能重叠，如果回溯会导致不期望的匹配，可以使用固化分组或者固化量词**

8. 经验 8：了解数据，作出假设

a) 例子：去掉文本首尾的空白字符