正则表达式教程

半年前我对正则表达式产生了兴趣,在网上查找过不少资料,看过不少的教程,最后在使用一个正则表达式工具 RegexBuddy 时发现他的教程写的非 常好,可以说是我目前见过最好的正则表达式教程。于是一直想把他翻译过来。这个愿望直到这个五一长假才得以实现,结果就有了这篇文章。关于本文的名字,使 用“深入浅出”似乎已经太俗。但是通读原文以后,觉得只有用“深入浅出”才能准确的表达出该教程给我的感受,所以也就不能免俗了。

1.什么是正则表达式

基本说来,正则表达式是一种用来描述一定数量文本的模式。Regex 代表 Regular Express。本文将用<<regex>>来表示一段具体的正则表达式。

一段文本就是最基本的模式,简单的匹配相同的文本。

2.不同的正则表达式引擎

正则表达式引擎是一种可以处理正则表达式的软件。通常,引擎是更大的应用程序的一部分。在软件世界,不同的正则表达式并不互相兼容。本教程会集中讨 论 Perl 5 类型的引擎,因为这种引擎是应用最广泛的引擎。同时我们也会提到一些和其他引擎的区别。许多近代的引擎都很类似,但不完全一样。例如.NET 正则 库,JDK 正则包。

3.文字符号

最基本的正则表达式由单个文字符号组成。如<<a>>,它将匹配字符串中第一次出现的字符“a”。如对字符串“Jack is a boy”。“J”后的“a”将被匹配。而第二个“a”将不会被匹配。正则表达式也可以匹配第二个“a”,这必须是你告诉正则表达式引擎从第一次匹配的地方开始搜索。在文本编辑器中,你可以使用“查找下一个”。在编程语言中,会有一个函数可以使你从前一次匹配的位置开始继续向后搜索。

类似的,<<cat>>会匹配“About cats and dogs”中的“cat”。这等于是告诉正则表达式引擎,找到一个<<c>>,紧跟一个<<a>>,再 跟一个<<t>>。

要注意,正则表达式引擎缺省是大小写敏感的。除非你告诉引擎忽略大小写,否则<<cat>>不会匹配“Cat”。

· 特殊字符

对于文字字符,有 11 个字符被保留作特殊用途。他们是:

[]\^$.|?*+()

这些特殊字符也被称作元字符。

如果你想在正则表达式中将这些字符用作文本字符,你需要用反斜杠“\”对其进行换码(escape)。例如你想匹配“1+1=2”,正确的表达式为<<1\+1=2>>.

需要注意的是,<<1+1=2>>也是有效的正则表达式。但它不会匹配“1+1=2”,而会匹配“123+111=234”中的“111=2”。因为“+”在这里表示特殊含义(重复 1 次到多次)。

在编程语言中,要注意,一些特殊的字符会先被编译器处理,然后再传递给正则引擎。因此正则表达式<<1\+2=2>>在 C++中要写成“1\\+1=2”。为了匹配“C:\temp”,你要用正则表达式<<C:\\temp>>。而在 C++中,正 则表达式则变成了“C:\\\\temp”。

·不可显示字符

可以使用特殊字符序列来代表某些不可显示字符:

<<\t>>代表 Tab(0x09)

<<\r>>代表回车符(0x0D)

<<\n>>代表换行符(0x0A)

要注意的是 Windows 中文本文件使用“\r\n”来结束一行而 Unix 使用“\n”。

4.正则表达式引擎的内部工作机制

知道正则表达式引擎是如何工作的有助于你很快理解为何某个正则表达式不像你期望的那样工作。

有两种类型的引擎:文本导向(text-directed)的引擎和正则导向(regex-directed)的引擎。

Jeffrey Friedl 把他们称作 DFA 和 NFA 引擎。本文谈到的是正则导向的引擎。这是因为一些非常有用的特性,如“惰性”量词(lazy quantifiers)和反向引用(backreferences),只能在正则导向的引擎中实现。所以毫不意外这种引擎是目前最流行的引擎。

你可以轻易分辨出所使用的引擎是文本导向还是正则导向。如果反向引用或“惰性”量词被实现,则可以肯定你使用的引擎是正则导向的。你可以作如下测 试:将正则表达式<<regex|regex not>>应用到字符串“regex not”。如果匹配的结果是 regex,则引擎是正则导向的。如果结果是 regex not,则是文本导向的。因为正则导向的引擎是“猴急”的,它会很急切的进行表功,报告它找到的第一个匹配 。

·正则导向的引擎总是返回最左边的匹配

这是需要你理解的很重要的一点:即使以后有可能发现一个“更好”的匹配,正则导向的引擎也总是返回最左边的匹配。

当把<<cat>>应用到“He captured a catfish for his cat”,引擎先比较<<c>>和“H”,结果失败了。于是引擎再比较<<c>>和“e”,也失败了。 直到第四个字符,<<c>>匹配了“c”。<<a>>匹配了第五个字符。到第六个字 符<<t>>没能匹配“p”,也失败了。引擎再继续从第五个字符重新检查匹配性。直到第十五个字符开 始,<<cat>>匹配上了“catfish”中的“cat”,正则表达式引擎急切的返回第一个匹配的结果,而不会再继续查找是 否有其他更好的匹配。

5.字符集

字符集是由一对方括号“[]”括起来的字符集合。使用字符集,你可以告诉正则表达式引擎仅仅匹配多个字符中的一个。如果你想匹配一个“a”或一个 “e”,使用<<[ae]>>。你可以使用<<gr[ae]y>>匹配 gray 或 grey。这在你不确 定你要搜索的字符是采用美国英语还是英国英语时特别有用。相反,<<gr[ae]y>>将不会匹配 graay 或 graey。字符 集中的字符顺序并没有什么关系,结果都是相同的。

你可以使用连字符“-”定义一个字符范围作为字符集。<<[0-9]>>匹配 0 到 9 之间的单个数字。你可以使用不止一个范 围。<<[0-9a-fA-F] >>匹配单个的十六进制数字,并且大小写不敏感。

你也可以结合范围定义与单个字符定义。<<[0-9a-fxA- FX]>>匹配一个十六进制数字或字母 X。再次强调一下,字符和范围定义的先后顺序对结果没有影响。

·字符集的一些应用

查找一个可能有拼写错误的单词,比如<<sep[ae]r[ae]te>> 或 <<li[cs]en[cs]e>>。

查找程序语言的标识符,<<A-Za-z_][A-Za-z_0-9]*>>。(*表示重复 0 或多次)

查找 C 风格的十六进制数<<0[xX][A-Fa-f0-9]+>>。(+表示重复一次或多次)

·取反字符集

在左方括号“[”后面紧跟一个尖括号“^”,将会对字符集取反。结果是字符集将匹配任何不在方括号中的字符。不像“.”,取反字符集是可以匹配回车换行符的。

需要记住的很重要的一点是,取反字符集必须要匹配一个字符。<<q[^u]>>并不意味着:匹配一个 q,后面没有 u 跟着。 它意味着:匹配一个 q,后面跟着一个不是 u 的字符。所以它不会匹配“Iraq”中的 q,而会匹配“Iraq is a country”中的 q 和一个空格符。事实上,空格符是匹配中的一部分,因为它是一个“不是 u 的字符”。

如果你只想匹配一个 q,条件是 q 后面有一个不是 u 的字符,我们可以用后面将讲到的向前查看来解决。

·字符集中的元字符

需要注意的是,在字符集中只有 4 个 字符具有特殊含义。它们是:“] \ ^ -”。 “]”代表字符集定义的结束;“\”代表转义;“^”代表取反;“-”代表范围定义。其他常见的元字符在字符集定义内部都是正常字符,不需要转义。 例如,要搜索星号*或加号+,你可以用<<[+*]>>。当然,如果你对那些通常的元字符进行转义,你的正则表达式一样会工作得 很好,但是这会降低可读性。

在字符集定义中为了将反斜杠“\”作为一个文字字符而非特殊含义的字符,你需要用另一个反斜杠对它进行转义。<< [\\x]>>将会匹配一个反斜杠和一个 X。 “]^-”都可以用反斜杠进行转义,或者将他们放在一个不可能使用到他们特殊含义的位置。我们推 荐后者,因为这样可以增加可读性。比如对于字符“^”,将它放在除了左括号“[”后面的位置,使用的都是文字字符含义而非取反含义。如<< [x^]>>会匹配一个 x 或^。<<[]x]>>会匹配一个“]”或“x”。<< [-x]>>或<<[x-]>>都会匹配一个“-”或“x”。

·字符集的简写

因为一些字符集非常常用,所以有一些简写方式。

<<\d>>代表<<[0-9]>>;

<<\w>>代表单词字符。这个是随正则表达式实现的不同而有些差异。绝大多数的正则表达式实现的单词字符集都包含了<<A-Za-z0-9_]>>。

<<\s>>代表“白字符”。这个也是和不同的实现有关的。在绝大多数的实现中,都包含了空格符和 Tab 符,以及回车换行符<<\r\n>>。

字符集的缩写形式可以用在方括号之内或之外。<<\s\d>>匹配一个白字符后面紧跟一个数字。<<[\s\d]>>匹配单个白字符或数字。<<[\da-fA-F]>>将匹配一个十六进制数字。

取反字符集的简写

<<[\S]>> = <<[^\s]>>

<<[\W]>> = <<[^\w]>>

<<[\D]>> = <<[^\d]>>

·字符集的重复

如果你用“?*+”操作符来重复一个字符集,你将会重复整个字符集。而不仅是它匹配的那个字符。正则表达式<<[0-9]+>>会匹配 837 以及 222。

如果你仅仅想重复被匹配的那个字符,可以用向后引用达到目的。我们以后将讲到向后引用。

6.使用?*或+ 进行重复

?:告诉引擎匹配前导字符 0 次或一次。事实上是表示前导字符是可选的。

+:告诉引擎匹配前导字符 1 次或多次

*:告诉引擎匹配前导字符 0 次或多次

<[A-Za-z][A-Za-z0-9]*>匹配没有属性的 HTML 标签,“<”以及 “>”是文字符号。第一个字符集匹配一个字母,第二个字符集匹配一个字母或数字。

我们似乎也可以用<[A-Za-z0-9]+>。但是它会匹配<1>。但是这个正则表达式在你知道你要搜索的字符串不包含类似的无效标签时还是足够有效的。

·限制性重复

许多现代的正则表达式实现,都允许你定义对一个字符重复多少次。词法是:{min,max}。

min 和 max 都是非负整数。如果逗号有而 max 被忽略了,则 max 没有限制。如果逗号和 max 都被忽略了,则重复 min 次。

因此{0,}和*一样,{1,}和+ 的作用一样。

你可以用<<\b[1-9][0-9]{3}\b>>匹配 1000~9999 之间的数字(“\b”表示单词边界)。<<\b[1-9][0-9]{2,4}\b>>匹配一个在 100~99999 之间的数字。

·注意贪婪性

假设你想用一个正则表达式匹配一个 HTML 标签。你知道输入将会是一个有效的 HTML 文件,因此正则表达式不需要排除那些无效的标签。所以如果是在两个尖括号之间的内容,就应该是一个 HTML 标签。

许多正则表达式的新手会首先想到用正则表达式<< <.+> >>,他们会很惊讶的发现,对于测试字符串,“This is a <EM>first</EM> test”,你可能期望会返回<EM>,然后继续进行匹配的时候,返回</EM>。

但事实是不会。正则表达式将会匹配 “<EM>first</EM>”。很显然这不是我们想要的结果。原因在于 “+”是贪婪 的。也就是说,“+”会导致正则表达式引擎试图尽可能的重复前导字符。只有当这种重复会引起整个正则表达式匹配失败的情况下,引擎会进行回溯。也就是说, 它会放弃最后一次的 “重复” ,然后处理正则表达式余下的部分。

和 “+”类似,“?*”的重复也是贪婪的。

·深入正则表达式引擎内部

让我们来看看正则引擎如何匹配前面的例子。第一个记号是 “<”,这是一个文字符号。第二个符号是 “.”,匹配了字符 “E”,然后 “+”一直可 以匹配其余的字符,直到一行的结束。然后到了换行符,匹配失败(“.”不匹配换行符)。于是引擎开始对下一个正则表达式符号进行匹配。也即试图匹配 “>”。到目前为止,“<.+”已经匹配了 “<EM>first</EM> test”。引擎会试图将 “>”与换行符进行匹配,结果失败了。于是引擎进行回溯。结果是现在 “<.+”匹配 “<EM>first</EM> tes”。于是引擎将 “>”与 “t”进行匹配。显然还是会失败。这个过程继续,直到 “<.+”匹配 “<EM>first</EM”,“>”与 “>”匹配。于是引擎找到了一个匹配 “<EM>first</EM>”。记住,正则导向的引擎是 “急切的” ,所以它会急着报告它找到的第一个匹配。而不是继续回溯,即使可能会有更好的匹配,例如 “<EM>”。所以我们可以看到,由于 “+”的贪婪性,使得正则表达式引擎返回了一个最左边的最长的匹配。

·用懒惰性取代贪婪性

一个用于修正以上问题的可能方案是用“+”的惰性代替贪婪性。你可以在“+”后面紧跟一个问号“?”来达到这一点。“*”,“{}”和“?”表示的 重复也可以用这个方案。因此在上面的例子中我们可以使用“<.+?>”。让我们再来看看正则表达式引擎的处理过程。

再一次,正则表达式记号“<”会匹配字符串的第一个“<”。下一个正则记号是“.”。这次是一个懒惰的“+”来重复上一个字符。这告诉 正则引擎,尽可能少的重复上一个字符。因此引擎匹配“.”和字符“E”,然后用“>”匹配“M”,结果失败了。引擎会进行回溯,和上一个例子不同,因为是惰性重复,所以引擎是扩展惰性重复而不是减少,于是“<.+”现在被扩展为“<EM”。引擎继续匹配下一个记号“>”。这次得到 了一个成功匹配。引擎于是报告“<EM>”是一个成功的匹配。整个过程大致如此。

·惰性扩展的一个替代方案

我们还有一个更好的替代方案。可以用一个贪婪重复与一个取反字符集:“<[^>]+>”。之所以说这是一个更好的方案在于使用惰性重复时,引擎会在找到一个成功匹配前对每一个字符进行回溯。而使用取反字符集则不需要进行回溯。

最后要记住的是,本教程仅仅谈到的是正则导向的引擎。文本导向的引擎是不回溯的。但是同时他们也不支持惰性重复操作。

7.使用“.”匹配几乎任意字符

在正则表达式中,“.”是最常用的符号之一。不幸的是,它也是最容易被误用的符号之一。

“.”匹配一个单个的字符而不用关心被匹配的字符是什么。唯一的例外是新行符。在本教程中谈到的引擎,缺省情况下都是不匹配新行符的。因此在缺省情况下,“.”等于是字符集[^\n\r](Window)或[^\n](Unix)的简写。

这个例外是因为历史的原因。因为早期使用正则表达式的工具是基于行的。它们都是一行一行的读入一个文件,将正则表达式分别应用到每一行上去。在这些工具中,字符串是不包含新行符的。因此“.”也就从不匹配新行符。

现代的工具和语言能够将正则表达式应用到很大的字符串甚至整个文件上去。本教程讨论的所有正则表达式实现都提供一个选项,可以使“.”匹配所有的字 符,包括新行符。在RegexBuddy, EditPad Pro 或 PowerGREP 等工具中,你可以简单的选中“点号匹配新行符”。在 Perl 中,“.”可以匹配新行符的模式被称作“单行模式”。很不幸,这是 一个很容易混淆的名词。因为还有所谓“多行模式”。多行模式只影响行首行尾的锚定(anchor),而单行模式只影响“.”。

其他语言和正则表达式库也采用了 Perl 的术语定义。当在.NET Framework 中使用正则表达式类时,你可以用类似下面的语句来激活单行模式:

Regex.Match(“string”,”regex”,RegexOptions.SingleLine)

·保守的使用点号“.”

点号可以说是最强大的元字符。它允许你偷懒:用一个点号,就能匹配几乎所有的字符。但是问题在于,它也常常会匹配不该匹配的字符。

我会以一个简单的例子来说明。让我们看看如何匹配一个具有“mm/dd/yy”格式的日期,但是我们想允许用户来选择分隔符。很快能想到的一个方案 是<<\d\d.\d\d.\d\d>>。看上去它能匹配日期“02/12/03”。问题在于 02512703 也会被认为是一个 有效的日期。<<\d\d[-/.]\d\d[-/.]\d\d>>看上去是一个好一点的解决方案。记住点号在一个字符集里不是

元字 符。这个方案远不够完善,它会匹配 “99/99/99”。而<<[0-1]\d[-/.][0-3]\d[-/.]\d\d>>
又更进一步。尽管他也会匹配 “19/39/99”。你想要你的正则表达式达到如何完美的程度取
决于你想达到什么样的目的。如果你想校验用户输入,则需要尽 可能的完美。如果你只是
想分析一个已知的源,并且我们知道没有错误的数据,用一个比较好的正则表达式来匹配你
想要搜寻的字符就已经足够。

8.字符串开始和结束的锚定

锚定和一般的正则表达式符号不同,它不匹配任何字符。相反,他们匹配的是字符之前或之
后的位置。 “^”匹配一行字符串第一个字符前的位 置。<<^a>>将会匹配字符串 “abc”中的 a。
<<^b>>将不会匹配 “abc”中的任何字符。

类似的,$匹配字符串中最后一个字符的后面的位置。所以<<c$>>匹配 “abc”中的 c。

·锚定的应用

在编程语言中校验用户输入时,使用锚定是非常重要的。如果你想校验用户的输入为整数,
用<<^\d+$>>。

用户输入中,常常会有多余的前导空格或结束空格。你可以用<<^\s*>>和<<\s*$>>来匹
配前导空格或结束空格。

·使用 “^”和 “$”作为行的开始和结束锚定

如果你有一个包含了多行的字符串。例如:“first line\n\rsecond line”(其中\n\r 表示一个新
行符)。常常需要对每行分别处理而不是整个字符串。因此,几乎所有的正则表达式引擎都
提供一个选项,可以扩展这两种锚 定的含义。 “^”可以匹配字串的开始位置(在 f 之前),以及
每一个新行符的后面位置(在\n\r 和 s 之间)。类似的,$会匹配字串的结束位置(最后一个 e
之后),以及每个新行符的前面(在 e 与\n\r 之间)。

在.NET 中,当你使用如下代码时,将会定义锚定匹配每一个新行符的前面和后面位置:

Regex.Match("string", "regex", RegexOptions.Multiline)

应用:string str = Regex.Replace(Original, "^", "> ", RegexOptions.Multiline)–将会在每行的
行首插入 “> ”。

· 绝对锚定

<<\A>>只匹配整个字符串的开始位置,<<\Z>>只匹配整个字符串的结束位置。即使你使用
了 “多行模式”,<<\A>>和<<\Z>>也从不匹配新行符。

即使\Z 和$只匹配字符串的结束位置,仍然有一个例外的情况。如果字符串以新行符结束,
则\Z 和$将会匹配新行符前面的位置,而不是整个字符串的最 后面。这个 “改进” 是由 Perl 引
进的,然后被许多的正则表达式实现所遵循,包括 Java,.NET 等。如果应用<<^[a- z]+$>>
到 “joe\n”,则匹配结果是 “joe”而不是 “joe\n”。