

Regular Expression
yet another introduction

正则指引

余晟 著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

正则指引

余晟 著

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

本书针对作者在开发中遇到的实际问题，以及其他开发人员咨询的问题，总结出一套使用正则表达式解题的办法，并通过具体的例子指导读者拆解、分析问题。全书分为三大部分：第一部分主要讲解正则表达式的基础知识，涵盖了常见正则表达式中的各种功能和结构；第二部分主要讲解关于正则表达式的更深入的知识，详细探讨了编码问题、匹配原理、解题思路；第三部分将之前介绍的各种知识落实到 6 种常用语言 .NET、Java、JavaScript、PHP、Python、Ruby 中，不但详细介绍了语言中正则表达式的用法，更点明了版本之间的细微差异，既可以作为专门学习的教材，也可以作为有用的参考手册。

本书适合经常需要进行文本处理（比如日志分析或网络运维）的技术人员、熟悉常用开发语言的程序员，以及已经对正则表达式有一定了解的读者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目 (CIP) 数据

正则指引 / 余晟著. — 北京：电子工业出版社，2012.5
ISBN 978-7-121-16551-1

I. ①正… II. ①余… III. ①正则表达式 IV. ①TP301.2

中国版本图书馆 CIP 数据核字 (2012) 第 046409 号

策划编辑：张月萍

责任编辑：葛 娜

印 刷：

装 订：中国电影出版社印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：21 字数：450 千字

印 次：2012 年 5 月第 1 次印刷

印 次：4000 册 定价：58.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线：(010) 88258888。

引子：关于正则表达式.....

正则表达式这个名字看起来总有点古怪，概念似乎也不简单，甚至需要用一整本书来讲解；可是，它到底是什么呢？

身为技术人员，我相信你总会与字符串打交道，相应地，各种语言也都提供了与字符串有关的函数。不妨先看看下面几个问题，字符串函数是如何解决的（下面的代码使用 Python 语言，它很直观，正文里有基础的介绍。现在，你只需要知道 def 是定义函数的关键词即可）。

1. 判断字符 ch 是否数字字符

```
def isDigit(ch) :  
    return ch == "0" or ch == "1" ..... or ch == "9"
```

2. 判断字符串 str 是否电话号码(为简单起见,现在只考虑固定电话号码,也就是长度在 7~8

位之间的数字字符串,且第一位不为 0)

```
def isPhoneNum (str) :  
    if len(str) >= 7 and len(str) <= 8 and str[0] != "0" :  
        for ch in str :  
            if not isDigit(ch) :  
                return false  
        return true  
    return false
```

任务本身并没有增加太多，但是程序复杂了很多倍；如果你不这样看，那么，来个更复杂的。

3. 找出一段文本中所有的电话号码

最直观的办法是，在字符串中的每个位置截取 7~8 个字符，调用之前的 isPhoneNum()。这么做看起来没问题，只是效率太低。

IV 引子：关于正则表达式……

当然，很容易就可以做点改进，只在“当前字符为数字字符”的情况下调用`isPhoneNum()`。这样效率倒是改进了，但是还有问题没有解决：要求找到的是长度大于等于 7 个字符，小于等于 8 个字符的“数字字符串”，而不是“子字符串”——也就是说，假如数字字符串是 `64240000`，需要将它找出来；如果数字字符串是 `13800138000`，则需要忽略它，以及其中的任何子串（比如 `13800138`，`00138000`）。

所以，用 `isPhoneNum()` 找出字符串之后，还需要保证它之前的字符不是数字字符，之后的字符也不是数字字符。看起来很简单，但为了避免越界错误，又需要判断：如果当前字符是整段文本的第一个字符，则不需要判断之前的字符，因为它不存在；同样，如果找出的字符串在整段文本的末尾，则不需要判断之后的字符，因为它同样不存在……

到现在为止，即便只是找到最简单的固定电话号码，程序也非常复杂，难以维护。如果要查找的是形式更多变的文本，比如带区号的电话号码（`021-64240000` 或者 `03718888888`）、手机号码（`13800138000` 或者 `+8613800138000` 或者 `013800138000`），程序更是不可想象，更不用说文件路径名、URL 地址、电子邮件地址了！然而，日常开发中我们又确实经常需要面对这类任务，有什么更好的办法呢？

正则表达式就是解决这类问题的万灵药。虽然许多人有点看不起它，觉得不入流，科班教材里也不会花太多篇幅来介绍它，但它确实是解决问题的利器——之前提到的三个例子，用正则表达式都可以轻松解决。

1. 判断字符 `ch` 是否数字字符

```
def isDigit(ch) :  
    return re.search(ch, "[0-9]") != None
```

看起来很复杂，其实并不复杂：这里真正要关心的就是正则表达式 `[0-9]`，它表示“从 0 到 9 之间的任意字符”，很形象吧？`re.search()` 是正则表达式运算函数，它判断 `ch` 能否由正则表达式 `[0-9]` 匹配，可以则返回一个结果，否则返回 `None`（这些细节正文中会讲到）。

2. 判断字符串 `str` 是否电话号码

```
def isPhoneNum(str) :  
    return re.search(str, "[1-9][0-9]{6,7}") != None
```

这个正则表达式最开始是 `[1-9]`，表示第一个字符必须是 1~9 之间的数字字符；之后是 `[0-9]{6,7}`，表示长度在 6 和 7 之间，由 0~9 之间的数字字符组成的字符串（两部分加起来，整个字符串的长度在 7 和 8 之间）。要解决的问题复杂了，正则表达式仍然直观形象。

3. 找出一段文本中所有的固定电话号码

```
def findNumStr(str) :
    return re.findall(str, '(?<![0-9])[1-9][0-9]{6,7}(?![0-9])')
```

这个正则表达式之前多出了 `(?<![0-9])`，表示“之前不能是[0-9]”；之后多出了 `(?![0-9])`，表示“之后不能是[0-9]”。虽然稍微复杂点，但意思明确，而且不难理解。`re.findall()`的意思也很明显：找到所有这样的字符串。

可以想象，循着这种思路，更复杂的电话号码、手机号码等任务都不难解决。更重要的是，之前需要许多行语句才能完成的任务，现在基本上只需要一个正则表达式，一条语句就可以完成。正因为如此，不少人虽然认为正则表达式不够花哨、漂亮，却不得不承认它是一种“匕首应用”——匕首，没有十八般兵刃那么大方，关键时刻却不可或缺，所以值得花时间练练。同样，正则表达式虽然不能用来显摆，但总有派得上用场的地方，花时间练练绝不是坏事。即便你的工作不是纯粹的文本处理（比如日志分析），也总会有用到正则表达式的地方（比如查找和修改源代码），所以我希望，这本书能陪伴你练出一身正则表达式的好功夫，在关键场合能亮出趁手的工具。

最后，为了尊重传统教科书的习惯，附上正则表达式的“科班史”：

正则表达式发源于与计算机密切相关的两个领域：计算理论和形式语言。20世纪40年代，两位神经生理学家 Warren McCulloch 和 Walter Pitts 研究出一种数学方式来描述神经网络的办法，它们把神经系统中的神经元描述成小而简单的自动控制单元。1956年，数学家 Stephen Cole Kleene 在他们研究的基础上，发表了一篇名为“神经网络事件的表示法”的论文，在其中，他采用了一些称之为“正则集合（regular set）”的数学符号来描述神经网络模型。

之后，UNIX 的主要发明人 Ken Thompson 将这个符号系统引入了文本编辑器 QED（意思是“在文本中搜索某种模式”），正则表达式由此也进入了计算机世界。随后 Ken Thompson 又将正则表达式引入了 UNIX 下的文本编辑器 ed，ed 最终演化为大家熟悉的 grep（grep 得名自 ed 编辑器中的正则表达式搜索命令 `g/re/p`，其中的 re 表示“正则表达式”）。

返璞归真——评《正则指引》

第一次接触正则表达式，是 2000 年我在西安一家公司使用 Perl 做网站开发时。之前我在工作中只使用过标准的 C 语言，Perl 这门编程语言的强大表达能力，令我印象极为深刻。Perl 的力量，除了语言本身的设计之外，很大程度上来自于它对正则表达式的完美支持。当时我们开发了一个网上商城的应用，允许很多商家在这里开店，可以选择一些不同的样式模板。我很快发现，使用 Perl+正则表达式是开发这类应用的利器。我们只花了大约一个月的时间，就完成了网站核心功能的开发。那时候我意识到，使用正则表达式是聪明人写程序的方法（没说我是聪明人，但是我非常希望与那些聪明人为伍），可以极大地提高代码的重用度和执行效率。如果完全不使用正则表达式，代码量会增加数倍甚至十倍。

后来因为一些原因，我告别了 Perl。在之后的工作中，我使用过 Java、JavaScript、Ruby 等编程语言。我发现这些语言对于正则表达式的支持，没有一个能够超越 Perl。Java 这种所谓的“工业主流编程语言”，一直到 2002 年 JDK 1.4 推出时，才正式把对正则表达式的支持加入到核心类库。因为长期缺乏对正则表达式的原生支持，以及语言本身表达能力欠缺，使用 Java 来做大量的文本处理，感觉非常笨拙，完全没有使用 Perl 那种指哪打哪的快感。直到 2007 年我发现了另一个更好的 Perl 语言——Ruby，才重新找回了 2000 年 Perl 带给我的编程快感。

因为我的工作主要是做 Web 开发，大量的时间花在与 HTML/CSS/JavaScript 以及关系数据库打交道上。在这里并没有很高深的算法，只有大量繁重的文本处理。难以想象，如果没有正则表达式，我们的开发将会是何等原始。

除了 Web 开发领域，需要实现大量自动化功能的一些领域，例如运维领域和自动化测试领域，

也是正则表达式大显身手的地方。无论使用稍显简陋的 sed/awk 还是更高级的 Perl/Python/Ruby, 实现自动化功能, 都必须依赖大量的正则表达式。

自从面向对象时髦起来之后, 甚至一度出现了面向对象万能论, 有人试图用 MDA 和可执行的 UML 来解决一切编程问题。但是我一直认为面向对象只解决了软件开发的一小部分问题, 而且是宏观方面的问题。正则表达式解决的问题, 是面向对象无能为力的一些微观方面的问题。在这里不需要坐而论道的方法论争论, 需要的是刺刀见红的肉搏战。这些问题即使使用完全面向对象的方式能够解决, 也会是很笨拙的。如果用物理学来比喻, 面向对象是“广义相对论”, 而正则表达式则是“量子力学”。

正则表达式已经成为了现代编程语言的基础模块, 现在很难找到一种不支持正则表达式的编程语言。除了编程语言外, 在很多工具软件, 例如文本编辑器 (Vi、Emacs、UltraEdit)、Web 服务器 (Apache、Nginx) 之中都能找到正则表达式的身影。

余晟老师是我的朋友, 我对他印象最为深刻的是他对于技术工作的严谨态度。“格物致知”是中国传统儒家学派所追求的一种道德修养, 也是一种境界。余老师是我的朋友中最接近“格物致知”这种境界的一位。我虽然从未精通过任何一门技术, 但是很喜欢结交余老师这样的朋友。

余老师潜心编著的这本《正则指引》深入浅出, 将正则表达式的由来和分支娓娓道来。阅读这本书, 我仿佛回到了 11 年前做 Perl 程序员时的快乐时光。国内很多程序员的一个通病是好高骛远, 像《正则指引》这样一本详细讲解基础知识的书未必会有很好的销路。但是等你做过很多年开发之后, 你会发现, 对你最有价值的, 正是这些基础知识和工具。软件开发的“道”, 正是隐藏在这些看起来不起眼的基础知识和工具之中的。

李锟

2011 年 11 月 25 日

前言

提到正则表达式，许多人很有点不屑一顾：这东西，不登大雅之堂，再说也不是总要用到，何必专门花时间学习？

没错，正则表达式并不“总要用到”，但到了需要的场合用不上，往往产生“一分钱难倒英雄汉”的尴尬。经常需要处理文本的程序员自然会知道正则表达式的价值，其他的程序员如果不会正则表达式，即便开发的领域与文本处理没什么关系，也难免“躺着中枪”的命运——前几天我遇到一个问题，将一行长长的地址拆分成多行，负责这部分的程序员的日常工作只是制作 PDF 而已，拆分地址是很“边缘”的功能，但不会正则表达式就无法准确折行（一般需要在标点符号出现的地方折行，而不能只在空白字符处折行，但是不同语言中的标点符号各有不同），结果一筹莫展；相反，如果了解正则表达式，就可以很容易地处理各种语言中的标点字符。

以我的开发经验来看，专门花点时间掌握正则表达式，确实是非常有必要的。目前可以见到的关于正则表达式的书籍和资料有不少，但又各有不足。

在互联网上，流传着一些编程语言的正则文档和《30 分钟教会你正则表达式》之类的帖子。这类资料的好处是简单直接，查到了，如果有现成的例子，而且适用于自己的语言，则可以直接拿来用；然而，其坏处也是简单直接，因为缺乏背后原理的讲解，如果找不到现成的例子，或者找不到能在自己所使用语言中行得通的例子（需知道，同样的正则表达式并不能直接套用到不同的语言中），则束手无策。

在正式的出版领域，已经有《精通正则表达式》、《正则表达式必知必会》之类的书籍出版，

尤其是前者，堪称关于正则表达式的经典著作，如果想认真学习正则表达式，这类书籍是必须阅读的。但这类书籍也有一个弱点，即都是由英文版本翻译而来的，更多地侧重英文文本的处理，身为中文世界的开发人员，我们经常需要处理中文文本——对于处理英文之外的字符，正则表达式已经提供了足够丰富的功能，但如何用对、用好这些功能，资料却很匮乏。

我经常需要给人讲解正则表达式的相关知识，时常惋惜的是，开发人员为这些问题所困扰；正因为如此，本书的写作动机便是着力弥补现有资料的缺陷。

相对于正则文档和速成教学帖子，本书深入讲解了匹配背后的原理，往往会举一反三，告诉读者，这里为何这样写，如果改成其他形式，会造成什么结构；并且，集中讲解和比较了多种语言中正则表达式用法的异同，方便读者把现成的正则表达式“移植”到自己的工作环境中。

相对于《精通正则表达式》等正式的书籍，本书辟出专门的内容讲解语言和编码，告诉读者如何设定编码，如何正确处理中文等字符。另外，本书还涵盖了.NET、Java、JavaScript、PHP、Python、Ruby 六种常用语言，对每种语言给出专门章节，不但详细介绍了语言中正则表达式的用法，更点明了版本之间的细微差异，不但可以作为专门学习的教材，还可以成为有用的参考手册。

本书结构

本书可以分为三大部分。

第一部分主要讲解正则表达式的基础知识，覆盖常见正则表达式中的各种功能和结构。看完前面 3 章，就可以基本弄明白现在流行的各种正则表达式；尤其是如果你之前有一些经验，会觉得阅读起来并不困难。但是我也希望读者不要忽略其他的内容，断言和匹配模式现在已经是正则表达式的“标准配备”了，而且确实可以派上大用场，所以第 4 章和第 5 章的内容，即便不是很熟悉，阅读起来可能有一些麻烦，也不应该忽略。最后的第 6 章，则厘清了正则表达式在使用中的若干疑惑，了解它们，你就可以相对自由地在正则表达式的世界里行走了。

第二部分主要讲解关于正则表达式的更深入的知识，这一部分用 3 章的内容，详细探讨了编码问题、匹配原理、解题思路。这部分内容更抽象，需要多花一点时间来阅读和理解，但是它们确实可以帮你在正则表达式的世界里登堂入室，脱离“术”的层面，掌握万变不离其宗的“道”。

第三部分的作用是接地气，将之前介绍的各种知识落实到六种常用语言.NET、Java、JavaScript、PHP、Python、Ruby 中来。每一章的开头有正则功能列表，其中的功能都对应着前面部分的讲解，这些功能的具体应用实例，以及不同版本之间的差异，则在章节中详细讲解，每一章的最后还给出了常见任务的示例代码，方便日后查询。在最后，第 16 章简要介绍了正则表达式在 Linux 下常用工具 vi、grep、awk、sed 中的使用，并通过一个实际的例子将这几种工具串起来，对比说明了它们适合解决的问题。

在本书的最后提供了用作参考的 3 个附录。

附录 A 是正则表达式的常用功能在不同语言中的比对，希望能给需要在多种语言中使用正则表达式或者移植正则表达式的读者提供一份有用的参考；附录 B 给出了若干常见的正则表达式，比如匹配邮政编码、身份证号、手机号、QQ 号、电子邮件地址等，希望能成为常见问题的“速查手册”；附录 C 列出了常用正则表达式的工具和资源，方便大家调试自己的正则表达式，以及继续深入学习。

本书读者

本书适合以下几类读者。

经常需要进行文本处理（比如日志分析或网络运维）的技术人员。这些读者或许已经熟悉了正则表达式的基本用法，但面对日益复杂化和海量化的数据，阅读本书可以帮助你更准确、更高效地处理文本，提升自己工作的价值。

熟悉常用开发语言的程序员。虽然这些读者不需要专职进行文本处理，但源代码和许多数据其实也是文本，如果不会正则表达式，在偶然遇到处理源代码或文本数据的任务时，往往会产生躺着中枪的无力感。本书第三部分可以帮你迅速找到有关的例子，并落实在自己的编程语言中。当然前两部分也非常有必要，因为它们可以帮你夯实基础。

对正则表达式已经有一定了解的读者。这些读者虽然能用正则表达式解决常见的任务，但未必了解正则表达式的编码问题、匹配原理、解题思路，仔细阅读本书的第二部分，可以深化完善对正则表达式的理解；而第三部分详细比较了使用正则表达式时各种语言，以及同一种语言中各种版本的差异。所有这一切，应该可以让你对正则表达式的掌握更上一层楼。

致谢

一本书的完成，必然离不开众多人的帮忙。

首先需要感谢的是周筠老师和徐定翔、卢鹤翔两位编辑，他们在我写作的最初阶段做了大量细心、耐心的工作，完全可以说，没有他们的这些工作，我就不会有写作这本书的念头，或者坚持写完的动力。

然后要感谢的是电子工业出版社的杨福平副总编和张月萍编辑，没有他们的关照和辛劳工作，这本书的出版定然会遇到更多的困难。

感谢我的朋友霍炬和韩磊，虽然我之前阅读过《精通正则表达式》，但与翻译和写作结缘，他们给了我莫大的帮助，有了这个契机，才有了现在的《正则指引》。尤其值得一提的是霍炬的夫人西乔，精心手绘了这本书的封面，在这里表示诚挚的谢意。

感谢我曾工作过的盛大创新院以及创新院的各位同事（李骏、郝培强、庄表伟、丁宇、许式伟、莫华枫、李道兵、赵劼、樊一鹏、张一宁等），创新院给大家宽松自由的工作环境，与各位同事的讨论加深了我对正则表达式的理解，也为我提供了许多例子。

感谢张东亮、陆亦斌、孙勇、叶劲峰等各位朋友，愿意拨冗阅读本书的草稿，并提出了大量专业的意见。

感谢何源、陈钢、贺钧、陈驰等读者，试读本书之后提出了大量的宝贵意见，在最后关头打消了我心中的许多忐忑。

在更早之前，我的父母从小就鼓励我研究和了解各种科学原理（“玩也要动脑筋”），我之所以有兴趣探究正则表达式背后的世界，而不满足于“够用/凑合”，归源都是受益于这种思维行为习惯。此外，在中小学阶段，我的语文老师罗碧玉、郭志鸿、易玺铭培养了我对于文字的兴趣，在大学阶段，东北师范大学文学学院的王确老师给了我这个理科生非常多的帮助和指引。对各位师长，在此一并表示感谢，能遇到你们是我的幸运。

最后还需要感谢许多为这本书做出过贡献的人，你们的名字我可能暂时无法记起，或者无法一一罗列，但我会心中存留对你们的谢意。

目 录

第 一 部 分

| | |
|-------------------------|----|
| 第 1 章 字符组 | 2 |
| 1.1 普通字符组 | 2 |
| 1.2 关于Python的基础知识 | 4 |
| 1.3 普通字符组（续） | 6 |
| 1.4 元字符与转义 | 8 |
| 1.5 排除型字符组 | 10 |
| 1.6 字符组简记法 | 12 |
| 1.7 字符组运算 | 14 |
| 1.8 POSIX字符组 | 15 |
| 第 2 章 量词 | 17 |
| 2.1 一般形式 | 17 |
| 2.2 常用量词 | 18 |
| 2.3 数据提取 | 21 |
| 2.4 点号 | 23 |
| 2.5 滥用点号的问题 | 23 |
| 2.6 忽略优先量词 | 26 |

| | |
|--------------------------|-----------|
| 2.7 转义 | 31 |
| 第 3 章 括号 | 33 |
| 3.1 分组 | 33 |
| 3.2 多选结构 | 39 |
| 3.3 引用分组 | 44 |
| 3.3.1 反向引用 | 48 |
| 3.3.2 各种引用的记法 | 50 |
| 3.3.3 命名分组 | 53 |
| 3.4 非捕获分组 | 54 |
| 3.5 补充 | 55 |
| 3.5.1 转义 | 55 |
| 3.5.2 URL Rewrite | 56 |
| 3.5.3 一个例子 | 58 |
| 第 4 章 断言 | 59 |
| 4.1 单词边界 | 59 |
| 4.2 行起始/结束位置 | 61 |
| 4.3 环视 | 68 |
| 4.4 补充 | 74 |
| 4.4.1 环视的价值 | 74 |
| 4.4.2 环视与分组编号 | 74 |
| 4.4.3 环视的支持程度 | 75 |
| 4.4.4 环视的组合 | 77 |
| 4.4.5 断言和反向引用之间的关系 | 79 |
| 第 5 章 匹配模式 | 81 |
| 5.1 不区分大小写模式 | 81 |
| 5.1.1 模式的指定方式 | 82 |
| 5.2 单行模式 | 84 |
| 5.3 多行模式 | 85 |
| 5.4 注释模式 | 87 |
| 5.5 补充 | 88 |
| 5.5.1 更多的模式 | 88 |
| 5.5.2 修饰符的作用范围 | 89 |

XIV 目 录

| | | |
|--------------|--------------------|-----------|
| 5.5.3 | 失效修饰符 | 90 |
| 5.5.4 | 模式与反向引用 | 90 |
| 5.5.5 | 冲突策略 | 91 |
| 5.5.6 | 哪种方式更好 | 92 |
| 第 6 章 | 其他 | 93 |
| 6.1 | 转义 | 93 |
| 6.1.1 | 字符串转义与正则转义 | 93 |
| 6.1.2 | 元字符的转义 | 97 |
| 6.1.3 | 彻底消除元字符的特殊含义 | 99 |
| 6.1.4 | 字符组中的转义 | 101 |
| 6.2 | 正则表达式的处理形式 | 101 |
| 6.2.1 | 函数式处理 | 102 |
| 6.2.2 | 面向对象式处理 | 102 |
| 6.2.3 | 比较 | 103 |
| 6.2.4 | 线程安全性 | 104 |
| 6.3 | 表达式中的优先级 | 106 |

第 二 部 分

| | | |
|--------------|------------------------|------------|
| 第 7 章 | Unicode | 110 |
| 7.1 | 关于编码 | 110 |
| 7.2 | 推荐使用Unicode编码 | 111 |
| 7.3 | Unicode匹配规则 | 115 |
| 7.4 | 单词边界 | 117 |
| 7.5 | 码值 | 119 |
| 7.6 | Unicode属性 | 121 |
| 7.6.1 | Unicode Property | 121 |
| 7.6.2 | Unicode Block..... | 122 |
| 7.6.3 | Unicode Script..... | 123 |
| 7.7 | Unicode属性列表 | 123 |
| 7.7.1 | Unicode Property | 123 |
| 7.7.2 | Unicode Block..... | 125 |
| 7.7.3 | Unicode Script..... | 128 |
| 7.8 | POSIX字符组 | 129 |

| | |
|---------------------------|-----|
| 第 8 章 匹配原理 | 130 |
| 8.1 有穷自动机 | 130 |
| 8.2 正则表达式的匹配过程 | 131 |
| 8.3 回溯 | 134 |
| 8.4 NFA和DFA | 136 |
| 第 9 章 常见问题的解决思路 | 138 |
| 9.1 关于元素的三种逻辑 | 138 |
| 9.1.1 必须出现 | 139 |
| 9.1.2 可能出现 | 139 |
| 9.1.3 不能出现 | 140 |
| 9.2 正则表达式的常见操作 | 142 |
| 9.2.1 提取 | 142 |
| 9.2.2 验证 | 148 |
| 9.2.3 替换 | 152 |
| 9.2.4 切分 | 157 |
| 9.3 正则表达式的优化建议 | 159 |
| 9.3.1 使用缓存 | 159 |
| 9.3.2 尽量准确地表达意图 | 160 |
| 9.3.3 避免重复匹配 | 160 |
| 9.3.4 独立出文本和锚点 | 161 |
| 9.4 别过分依赖正则表达式 | 162 |
| 9.4.1 彻底放弃字符串操作 | 162 |
| 9.4.2 思维定势 | 163 |
| 9.4.3 正则表达式可以匹配各种文本 | 164 |

第 三 部 分

| | |
|------------------------|-----|
| 第 10 章 .NET | 168 |
| 10.1 预备知识 | 168 |
| 10.2 正则功能详解 | 169 |
| 10.2.1 列表 | 169 |
| 10.2.2 字符组 | 170 |
| 10.2.3 Unicode属性 | 170 |

| | | |
|---------------|-------------------|------------|
| 10.2.4 | 字符组简记法 | 171 |
| 10.2.5 | 单词边界 | 171 |
| 10.2.6 | 行起始/结束位置 | 172 |
| 10.2.7 | 环视 | 173 |
| 10.2.8 | 匹配模式 | 173 |
| 10.2.9 | 捕获分组的引用 | 174 |
| 10.3 | 正则API简介 | 175 |
| 10.3.1 | Regex | 175 |
| 10.3.2 | Match | 179 |
| 10.4 | 常用操作示例 | 180 |
| 10.4.1 | 验证 | 180 |
| 10.4.2 | 提取 | 180 |
| 10.4.3 | 替换 | 181 |
| 10.4.4 | 切分 | 182 |
| 第 11 章 | Java | 183 |
| 11.1 | 预备知识 | 183 |
| 11.2 | 正则功能详解 | 184 |
| 11.2.1 | 列表 | 184 |
| 11.2.2 | 字符组 | 184 |
| 11.2.3 | Unicode属性 | 186 |
| 11.2.4 | 字符组简记法 | 186 |
| 11.2.5 | 单词边界 | 186 |
| 11.2.6 | 行起始/结束位置 | 187 |
| 11.2.7 | 环视 | 188 |
| 11.2.8 | 匹配模式 | 188 |
| 11.2.9 | 纯文本模式 | 189 |
| 11.2.10 | 捕获分组的引用 | 189 |
| 11.3 | 正则API简介 | 189 |
| 11.3.1 | Pattern | 190 |
| 11.3.2 | Matcher | 192 |
| 11.3.3 | String | 194 |
| 11.4 | 常用操作示例 | 195 |
| 11.4.1 | 验证 | 195 |
| 11.4.2 | 提取 | 196 |

| | | |
|---------------|-------------------------|------------|
| 11.4.3 | 替换 | 196 |
| 11.4.4 | 切分 | 197 |
| 第 12 章 | JavaScript | 198 |
| 12.1 | 预备知识 | 198 |
| 12.2 | 正则功能详解 | 199 |
| 12.2.1 | 列表 | 199 |
| 12.2.2 | 字符组 | 199 |
| 12.2.3 | 字符组简记法 | 200 |
| 12.2.4 | 单词边界 | 200 |
| 12.2.5 | 行起始/结束位置 | 201 |
| 12.2.6 | 环视 | 201 |
| 12.2.7 | 匹配模式 | 202 |
| 12.2.8 | 捕获分组的引用 | 203 |
| 12.3 | 正则API简介 | 203 |
| 12.3.1 | RegExp..... | 203 |
| 12.3.2 | String..... | 207 |
| 12.4 | 常用操作示例 | 210 |
| 12.4.1 | 验证 | 210 |
| 12.4.2 | 提取 | 210 |
| 12.4.3 | 替换 | 211 |
| 12.4.4 | 切分 | 211 |
| 12.5 | 关于ActionScript | 211 |
| 12.5.1 | RegExp..... | 211 |
| 12.5.2 | 匹配规则 | 212 |
| 12.5.3 | 匹配模式 | 212 |
| 12.5.4 | 正则API | 212 |
| 第 13 章 | PHP..... | 213 |
| 13.1 | 预备知识 | 213 |
| 13.2 | 正则功能详解 | 215 |
| 13.2.1 | 列表 | 215 |
| 13.2.2 | 字符组 | 216 |
| 13.2.3 | Unicode属性 | 217 |

| | | |
|---------------|----------------------------|------------|
| 13.2.4 | 字符组简记法 | 217 |
| 13.2.5 | 单词边界 | 217 |
| 13.2.6 | 行起始/结束位置 | 218 |
| 13.2.7 | 环视 | 219 |
| 13.2.8 | 匹配模式 | 219 |
| 13.2.9 | 纯文本模式 | 220 |
| 13.2.10 | 捕获分组的引用 | 220 |
| 13.3 | 正则API简介 | 221 |
| 13.3.1 | PREG 常量说明 | 221 |
| 13.3.2 | preg_quote..... | 222 |
| 13.3.3 | preg_grep | 223 |
| 13.3.4 | preg_match..... | 223 |
| 13.3.5 | preg_match_all..... | 225 |
| 13.3.6 | preg_last_error | 227 |
| 13.3.7 | preg_replace | 227 |
| 13.3.8 | preg_replace_callback..... | 227 |
| 13.3.9 | preg_filter..... | 228 |
| 13.3.10 | preg_split..... | 229 |
| 13.4 | 常见的正则操作举例 | 230 |
| 13.4.1 | 验证 | 230 |
| 13.4.2 | 提取 | 230 |
| 13.4.3 | 替换 | 231 |
| 13.4.4 | 切分 | 232 |
| 第 14 章 | Python | 233 |
| 14.1 | 预备知识 | 233 |
| 14.2 | 正则功能详解 | 234 |
| 14.2.1 | 列表 | 234 |
| 14.2.2 | 字符组 | 235 |
| 14.2.3 | Unicode属性 | 236 |
| 14.2.4 | 字符组简记法 | 236 |
| 14.2.5 | 单词边界 | 238 |
| 14.2.6 | 行起始/结束位置 | 239 |
| 14.2.7 | 环视 | 239 |
| 14.2.8 | 匹配模式 | 240 |

| | | |
|---------------|---|------------|
| 14.2.9 | 捕获分组的引用 | 240 |
| 14.3 | 正则API简介 | 241 |
| 14.3.1 | RegexObject..... | 241 |
| 14.3.2 | re.compile(regex[, flags])..... | 243 |
| 14.3.3 | re.search(pattern, string[, flags]) | 243 |
| 14.3.4 | MatchObject..... | 243 |
| 14.3.5 | re.match(pattern, string[, flags])..... | 244 |
| 14.3.6 | re.findall(pattern, sting[, flags]) | 245 |
| 14.3.7 | re.finditer(pattern, string[, flags])..... | 245 |
| 14.3.8 | re.split(pattern, string[, maxsplit=0, flags=0])..... | 246 |
| 14.3.9 | re.sub(pattern, repl, string[, count, flags]) | 247 |
| 14.4 | 常用操作示例 | 248 |
| 14.4.1 | 验证 | 248 |
| 14.4.2 | 提取 | 248 |
| 14.4.3 | 替换 | 249 |
| 14.4.4 | 切分 | 250 |
| 第 15 章 | Ruby..... | 251 |
| 15.1 | 预备知识 | 251 |
| 15.2 | 正则功能详解 | 252 |
| 15.2.1 | 列表 | 252 |
| 15.2.2 | 字符组 | 252 |
| 15.2.3 | Unicode属性 | 253 |
| 15.2.4 | 字符组简记法 | 254 |
| 15.2.5 | 单词边界 | 254 |
| 15.2.6 | 行起始/结束位置 | 255 |
| 15.2.7 | 环视 | 256 |
| 15.2.8 | 匹配模式 | 256 |
| 15.2.9 | 捕获分组的引用 | 257 |
| 15.3 | 正则API简介 | 257 |
| 15.3.1 | Regexp | 257 |
| 15.3.2 | Regexp.match(text) | 259 |
| 15.3.3 | Regexp.quote(text)和Regexp.escape(text)..... | 260 |
| 15.3.4 | String.index(Regexp) | 261 |
| 15.3.5 | String.scan(Regexp) | 261 |

| | | |
|---------------|-----------------------------|------------|
| 15.3.6 | String.slice(Regexp) | 262 |
| 15.3.7 | String.split(Regexp) | 262 |
| 15.3.8 | String.sub(Regexp, Str) | 263 |
| 15.3.9 | String.gsub(Regexp, String) | 264 |
| 15.4 | 常用操作示例 | 264 |
| 15.4.1 | 验证 | 264 |
| 15.4.2 | 提取 | 265 |
| 15.4.3 | 替换 | 265 |
| 15.4.4 | 切分 | 265 |
| 15.5 | Ruby 1.9 的新变化 | 266 |
| 第 16 章 | Linux/UNIX | 268 |
| 16.1 | POSIX | 268 |
| 16.1.1 | POSIX规范 | 268 |
| 16.1.2 | POSIX字符组 | 269 |
| 16.2 | vi | 271 |
| 16.2.1 | 字符组及简记法 | 271 |
| 16.2.2 | 量词 | 272 |
| 16.2.3 | 多选结构和捕获分组 | 272 |
| 16.2.4 | 环视 | 273 |
| 16.2.5 | 锚点和单词边界 | 273 |
| 16.2.6 | 替换操作的特殊字符 | 274 |
| 16.2.7 | replacement中的特殊变量 | 276 |
| 16.2.8 | 补充 | 276 |
| 16.3 | grep | 277 |
| 16.3.1 | 基本用法 | 277 |
| 16.3.2 | 字符组 | 277 |
| 16.3.3 | 锚点和单词边界 | 278 |
| 16.3.4 | 量词 | 278 |
| 16.3.5 | 多选结构和捕获分组 | 279 |
| 16.3.6 | options | 279 |
| 16.3.7 | egrep和fgrep | 280 |
| 16.3.8 | 补充 | 280 |
| 16.4 | awk | 281 |
| 16.4.1 | 基本用法 | 281 |

| | | |
|--------|---------------------|-----|
| 16.4.2 | 字符组及简记法 | 282 |
| 16.4.3 | 锚点和单词边界 | 283 |
| 16.4.4 | 量词 | 283 |
| 16.4.5 | 多选结构 | 284 |
| 16.4.6 | 补充 | 284 |
| 16.5 | sed | 284 |
| 16.5.1 | 基本用法 | 284 |
| 16.5.2 | 字符组及简记法 | 285 |
| 16.5.3 | 锚点和单词边界 | 285 |
| 16.5.4 | 量词 | 286 |
| 16.5.5 | 多选结构和捕获分组 | 286 |
| 16.5.6 | options | 286 |
| 16.5.7 | 补充 | 287 |
| 16.6 | 总结 | 288 |
| 附录A | 常用语言中正则特性一览 | 291 |
| 附录B | 常用的正则表达式 | 293 |
| 附录C | 常用的正则表达式工具及资源 | 309 |

第 1 章 字符组

1.1 普通字符组

字符组 (Character Class)¹是正则表达式最基本的结构之一，要理解正则表达式的“灵活”，认识它是第一步。

顾名思义，字符组就是一组字符，在正则表达式中，它表示“在同一个位置可能出现的各种字符”，其写法是在一对方括号 [和] 之间列出所有可能出现的字符，简单的字符组比如 `[ab]`、`[314]`、`[#.?]` 在解决一些常见问题时，使用字符组可以大大简化操作，下面举“匹配数字字符”的例子来说明。

字符可以分为很多类，比如数字、字母、标点等。有时候要求“只出现一个数字字符”，换句话说，这个位置上的字符只能是 `0`、`1`、`2`、`...`、`8`、`9` 这 10 个字符之一。要进行这种判断，通常的思路是：用 10 个条件分别判断字符是否等于这 10 个字符，对 10 个结果取“或”，只要其中一个条件成立，就返回 True，表示这是一个数字字符，其伪代码如例 1-1 所示。

例 1-1 判断数字字符的伪代码

```
charStr == "0" || charStr == "1" ... || charStr == "9"
```

注：因为正则表达式处理的都是“字符串” (String) 而不是“字符”，所以这里假设变量 `charStr` (虽然它只包含一个字符) 也是字符串类型，使用了双引号，在有些语言中字符串也用单引号表示。

这种解法的问题在于太烦琐——如果要判断是否是一个小写英文字母，就要用 `||` 连接 26 个判断；如果还要兼容大写字母，则要连接 52 个判断，代码长到几乎无法阅读。相反，用字符组解决起来却异常简单，具体思路是：列出可能出现的所有字符 (在这个例子里就是 10 个数字字符)，只要出现了其中任何一个，就返回 True。例 1-2 给出了使用字符组判断的例子，程序语言使用 Python。

例 1-2 用正则表达式判断数字字符

```
re.search("[0123456789]", charStr) != None
```

`re.search()` 是 Python 提供的正则表达式操作函数，表示“进行正则表达式匹配”；`charStr` 仍然是需要判断的字符串，而 `[0123456789]` 则是以字符串形式给出的正则表达式，它是一个字符组，表示“这里可以是 `0`、`1`、`2`、`...`、`8`、`9` 中的任意一个字符。只要 `charStr` 与其中任何一个字符相同 (或者说“`charStr` 可以由 `[0123456789]` 匹配”)，就会得到一个 `MatchObject` 对象 (这个对象暂时不必关心，在第 17 页会详细讲解)；否则，返回 `None`。所以判断结果是否为 `None`，就可以判断 `charStr` 是否是数字字符。

¹ 在有的资料中，写作 Character Set，所以也有人翻译为“字符类”或者“字符集”。不过在计算机术语中，“类”是和“对象”相关的，“字符集”常常表示 Character Set (比如 GBK、UTF-8 之类)，所以本书中没有采用这两个名字。

当今流行的编程语言大多支持正则表达式，上面的例子在各种语言中的写法大抵相同，唯一的区别在于如何调用正则表达式的功能，所以用法其实大同小异。例 1-3 列出了常见语言中的表示，如果你现在就希望知道语言的细节，可以参考本书第三部分的具体章节。

例 1-3 用正则表达式判断数字字符在各种语言中的应用²

.NET (C#)

```
//能匹配则返回 true, 否则返回 false  
Regex.IsMatch(charStr, "[0123456789]");
```

Java

```
//能匹配则返回 true, 否则返回 false  
charStr.matches("[0123456789]");
```

JavaScript

```
//能匹配则返回 true, 否则返回 false  
/[0123456789]/.test(charStr);
```

PHP

```
//能匹配则返回 1, 否则返回 0  
preg_match("/[0123456789]/", charStr);
```

Python

```
#能匹配则返回 RegexObject, 否则返回 None  
re.search("[0123456789]", charStr)
```

Ruby

```
#能匹配则返回 0, 否则返回 nil  
charStr =~ /[0123456789]/
```

可以看到，不同语言使用正则表达式的方法也不相同。如果仔细观察会发现Java、.NET、Python、PHP中的正则表达式，都要以字符串形式给出，两端都有双引号；而Ruby和JavaScript中的正则表达式则不必如此，只在首尾有两个斜线字符 /，这也是不同语言中使用正则表达式的不同之处。不过，这个问题现在不需要太关心，因为本书中大部分例子以Python程序来讲解，下面讲解关于Python的基础知识，其他语言的细节留到后文会详细介绍。

1.2 关于 Python 的基础知识

本书选择使用 Python 语言来演示实际的匹配结果，因为它能在多种操作系统中运行，安装也很方便；另一方面，Python 是解释型语言，输入代码就能看到结果，方便动手实践。考虑到不是所有人都熟悉 Python，这里专门用一节来介绍。

如果你的机器上没有安装Python，可以从<http://python.org/download/>下载，目前Python有 2 和 3 两个版本，本书的例子以 2 版本为准³。请选择自己平台对应的程序下载并安装（目前MacOS、Linux的各种发行版一般带有Python，具体可以在命令行下输入python，看是否启动对应的程序）。

然后可以启动 Python，在 MacOS 和 Linux 下是输入 python，会显示出 Python 提示符，进入交互

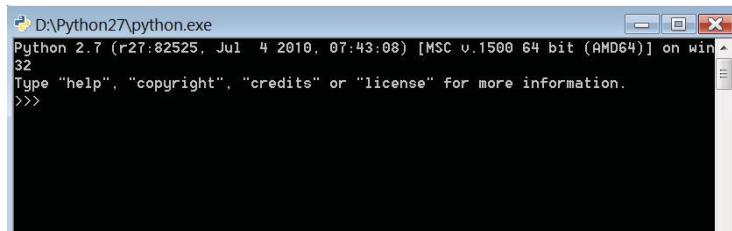
² 客观地说，Perl 是正则表达式处理最方便的编程语言，考虑到今天使用 Perl 的人数，以及 Perl 程序员一般都熟练掌握正则表达式的现实，本书没有给出 Perl 语言的例子。

³ 本书写作时，2.x 最新的版本为 2.7，示例以此为准。Python 3 虽然已经正式发行，但相对 2.x 变化较大，而 2.x 较为流行，所以采用了 2.x 版本。关于 2.x 和 3.x 的差别，在 Python 一章有详细介绍。

模式，如图 1-1（Linux 下的提示符与 MacOS 下的差不多，所以此处不列出）；而在 Windows 下，需要在“开始”菜单的“程序”中，选择 Python 目录下的 Python(command line)，如图 1-2 所示。

```
Python 2.7.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

图 1-1 MacOS下的Python提示符



```
D:\Python27\python.exe
Python 2.7 (r27:82525, Jul 4 2010, 07:43:08) [MSC v.1500 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

图 1-2 Windows下的Python提示符

Python中常用的关于正则表达式的函数是 `re.search()`，使用它必须首先导入正则表达式对应的包（package），也就是输入下面的代码。

```
#导入正则表达式对应的包
import re
```

通常的用法是提供两个参数：`re.search(pattern, string)`，其中 `pattern`是字符串形式提供的正则表达式，`string`是需要匹配的字符串；如果能匹配，则返回一个 `MatchObject`（详细介绍请参考第**错误！未定义书签。**页，暂时可以不必关心），这时提示符会显示类似 `<_sre.SRE_Match object at 0x000000001D8E578>`之类的结果；如果不能匹配，结果是`None`（这是Python中的一个特殊值，类似其他某些语言中的`Null`），不会有任何显示。图 1-3 演示了运行Python语句的结果。

```
>>> import re
>>> re.search("[0123456789]", "4")
<_sre.SRE_Match object at 0x000000001D8E578>
>>> re.search("[0123456789]", "a")
>>>
```

图 1-3 观察re.search()匹配的返回值

注：`>>>`是等待输入的提示符，以`>>>`开头的行，之后文本是用户输入的语句；其他行是系统生成的，比如打印出语句的结果（在交互模式下，匹配结果会自动输出，便于观察；真正程序运行时不会如此）。

为讲解清楚、形象、方便，本书中的程序部分需要做两点修改。

第一，因为暂时还不需要关心匹配结果的细节，只关心有没有结果，所以在 `re.search()`之后添加判断返回值是否为`None`，如果为`True`，则表示匹配成功，否则返回`False`表示匹配失败。为节省版面，尽可能用注释表示这类匹配结果，如 `# => True`或者 `# => False`，附在语句之后。

第二，目前我们关心的是整个字符串是否能由正则表达式匹配。但是，在默认情况下 `re.search(pattern, string)`只判断 `string`的某个子串能否由 `pattern`匹配，即便 `pattern`只能匹配 `string`的一部分，也不会返回`None`。为了测试整个 `string`能否由 `pattern`匹配，在 `pattern`两端加上 `^`和 `$`。`^`和 `$`是正则表达式中的特殊字符，它们并不匹配任何字符，只是表示“定位到字符串的起始位置”和“定位到字符串的结束位置”（原理如图 1-4 所示，如果你现在就希望详细了解这两个特殊

字符，可以参考第**错误！未定义书签。**页），这样就保证：只有在整个 *string* 都可以由 *pattern* 匹配时，才算匹配成功，不返回 None，如例 1-4 所示。

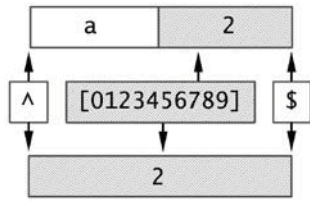


图 1-4 `^[0123456789]$` 的匹配

例 1-4 使用 `^` 和 `$` 测试 *string* 由 *pattern* 完整匹配

```
# 只要字符串中包含数字字符，就可以匹配
re.search("[0123456789]", "2") != None           # => True
re.search("^[0123456789]$", "12") != None       # => False
re.search("[0123456789]", "a2") != None         # => True

# 整个字符串就是一个数字字符，才可以匹配
re.search("[0123456789]", "2") != None           # => True
re.search("^[0123456789]$", "12") != None       # => False
re.search("^[0123456789]$", "a2") != None       # => False
```

1.3 普通字符组 (续)

介绍完关于 Python 的基础知识，继续讲解字符组。字符组中的字符排列顺序并不影响字符组的功能，出现重复字符也不会影响，所以 `[0123456789]` 完全等价于 `[9876543210]`、`[1029384756]`、`[9988876543210]`。

不过，代码总是要容易编写，方便阅读，正则表达式也是一样，所以一般并不推荐在字符组中出现重复字符。而且，还应该让字符组中的字符排列更符合认知习惯，比如 `[0123456789]` 就好过 `[0192837465]`。为此，正则表达式提供了 **-范围表示法** (range)，它更直观，能进一步简化字符组。

所谓“-范围表示法”，就是用 `[x-y]` 的形式表示 `x` 到 `y` 整个范围内的字符，省去一一列出的麻烦，这样 `[0123456789]` 就可以表示为 `[0-9]`。如果你觉得这不算什么，那么确实比 `[abcdefghijklmnopqrstuvwxyz]` 简单太多了。

你可能会问，“-范围表示法”的范围是如何确定的？为什么要写作 `[0-9]`，而不写作 `[9-0]`？

要回答这个问题，必须了解范围表示法的实质。在字符组中，`-` 表示的范围，一般是根据字符对应的 **码值** (Code Point，也就是字符在对应编码表中的编码的数值) 来确定的，码值小的字符在前，码值大的字符在后。在 ASCII 编码中 (包括各种兼容 ASCII 的编码中)，字符 `0` 的码值是 48 (十进制)，字符 `9` 的码值是 57 (十进制)，所以 `[0-9]` 等价于 `[0123456789]`；而 `[9-0]` 则是错误的范围，因为 `9` 的码值大于 `0`，所以会报错。程序代码见例 1-5。

例 1-5 `[0-9]` 是合法的，`[9-0]` 会报错

```
re.search("^[0-9]$", "2") != None           # => True
```

```
re.search("^[9-0]$", "2") != None
Traceback (most recent call last):
error: bad character range
```

如果知道 0~9 的码值是 48~57, a~z 的码值是 97~122, A~Z 的码值是 65~90, 能不能用 [0-z] 统一表示数字字符、小写字母、大写字母呢?

答案是: 勉强可以, 但不推荐这么做。根据惯例, 字符组的范围表示法都表示一类字符 (数字字符是一类, 字母字符也是一类), 所以虽然 [0-9]、[a-z] 都是很好理解的, 但 [0-z] 却很难理解, 不熟悉ASCII编码表的人甚至不知道这个字符组还能匹配大写字母, 更何况, 在码值 48 到 122 之间, 除去数字字符 (码值 48~57)、小写字母 (码值 97~122)、大写字母 (码值 65~90), 还有不少标点符号 (参见表 1-1), 从字符组 [0-z] 中却很难看出来, 使用时就容易引起误会, 例 1-6 所示的程序就可能让人莫名其妙。

表 1-1 ASCII编码表 (片段)

| 码值 | 字符 | 码值 | 字符 | 码值 | 字符 | 码值 | 字符 | 码值 | 字符 |
|----|----|----|----|----|----|-----|----|-----|----|
| 48 | 0 | 63 | ? | 78 | N | 93 |] | 108 | l |
| 49 | 1 | 64 | @ | 79 | O | 94 | ^ | 109 | m |
| 50 | 2 | 65 | A | 80 | P | 95 | _ | 110 | n |
| 51 | 3 | 66 | B | 81 | Q | 96 | ` | 111 | o |
| 52 | 4 | 67 | C | 82 | R | 97 | a | 112 | p |
| 53 | 5 | 68 | D | 83 | S | 98 | b | 113 | q |
| 54 | 6 | 69 | E | 84 | T | 99 | c | 114 | r |
| 55 | 7 | 70 | F | 85 | U | 100 | d | 115 | s |
| 56 | 8 | 71 | G | 86 | V | 101 | e | 116 | t |
| 57 | 9 | 72 | H | 87 | W | 102 | f | 117 | u |
| 58 | : | 73 | I | 88 | X | 103 | g | 118 | v |
| 59 | ; | 74 | J | 89 | Y | 104 | h | 119 | w |
| 60 | < | 75 | K | 90 | Z | 105 | i | 120 | x |
| 61 | = | 76 | L | 91 | [| 106 | j | 121 | y |
| 62 | > | 77 | M | 92 | \ | 107 | k | 122 | z |

例 1-6 [0-z]的奇怪匹配

```
re.search("^[0-z]$", "A") != None      # => True
re.search("^[0-z]$", ":") != None      # => True
```

在字符组中可以同时并列多个“-范围表示法”, 字符组 [0-9a-zA-Z] 可以匹配数字、大写字母或小写字母; 字符组 [0-9a-fA-F] 可以匹配数字, 大、小写形式的 a~f, 它可以用来验证十六进制字符, 代码见例 1-7。

例 1-7 [0-9a-fA-F]准确判断十六进制字符

```
re.search("^[0-9a-fA-F]$", "0") != None # => True
re.search("^[0-9a-fA-F]$", "c") != None # => True
re.search("^[0-9a-fA-F]$", "i") != None # => False
```

```
re.search("[^[\0-9a-fA-F]$", "C") != None # => True
re.search("[^[\0-9a-fA-F]$", "G") != None # => False
```

在不少语言中，还可以用转义序列 `\xhex` 来表示一个字符，其中 `\x` 是固定前缀，表示转义序列的开头，`num` 是字符对应的码值（Code Point，详见第 127 页，下文用 `☞127` 表示），是一个两位的十六进制数值。比如字符 `A` 的码值是 41（十进制则为 65），所以也可以用 `\x41` 表示。

字符组中有时会出现这种表示法，它可以表现一些难以输入或者难以显示的字符，比如 `\x7F`；也可以用来方便地表示某个范围，比如所有 ASCII 字符对应的字符组就是 `[\x00-\x7F]`，代码见例 1-8。这种表示法很重要，在第 **错误！未定义书签**。页还会讲到它，依靠这种表示法可以很方便地匹配所有的中文字符。

例 1-8 `[\x00-\x7F]` 准确判断 ASCII 字符

```
re.search("^[\\x00-\\x7F]$", "c") != None # => True
re.search("^[\\x00-\\x7F]$", "I") != None # => True
re.search("^[\\x00-\\x7F]$", "0") != None # => True
re.search("^[\\x00-\\x7F]$", "<") != None # => True
```

1.4 元字符与转义

在上面的例子里，字符组中的横线 `-` 并不能匹配横线字符，而是用来表示范围，这类字符叫做 **元字符**（meta-character）。字符组的开方括号 `[`、闭方括号 `]` 和之前出现的 `^`、`$` 都算元字符。在匹配中，它们有着特殊的意义。但是，有时候并不需要表示这些特殊意义，只需要表示普通字符（比如“我就想表示横线字符 `-`”），此时就必须做特殊处理。

先来看字符组中的 `-`，如果它紧邻着字符组中的开方括号 `[`，那么它就是普通字符，其他情况下都是元字符；而对于其他元字符，取消特殊含义的做法都是转义，也就是在正则表达式中的元字符之前加上反斜线字符 `\`。

如果要在字符组内部使用横线 `-`，最好的办法是将其排列在字符组的最开头。`[-09]` 就是包含三个字符 `-`、`0`、`9` 的字符组；`[0-9]` 是包含 `0~9` 这 10 个字符的字符组，`[-0-9]` 则是由“-范围表示法”`0-9` 和横线 `-` 共同组成的字符组，它可以匹配 11 个字符，例 1-9 说明了使用横线 `-` 的各种情况。

例 1-9 `-` 出现在不同位置，含义不同

```
#作为普通字符
re.search("^[0-9]$", "3") != None # => False
re.search("^[0-9]$", "-") != None # => True
#作为元字符
re.search("^[0-9]$", "3") != None # => True
re.search("^[0-9]$", "-") != None # => False
#转义之后作为普通字符
re.search("^[0\\-9]$", "3") != None # => False
re.search("^[0\\-9]$", "-") != None # => True
```

仔细观察会发现，在正文里说“在正则表达式中的元字符之前加上反斜线字符 `\`”，而在代码里写的却不是 `[0-9]`，而是 `[0\\-9]`。这并不是输入错误。

因为在这段程序里，正则表达式是以字符串（String）的方式⁴提供的，而字符串本身也有关于转义的规定（你或许记得，在字符串中有 `\n`、`\t` 之类的转义序列）。上面说的“正则表达式”，其实是经过“字符串转义处理”之后的字符串的值，正则表达式 `[0\ -9]` 包含 6 个字符：`[`、`0`、`\`、`-`、`9`、`]`，在字符串中表达这 6 个字符；但是在源代码里，必须使用 7 个字符：`\` 需要转义成 `\\`，因为处理字符串时，反斜线和之后的字符会被认为是转义序列（Escape Sequence），比如 `\n`、`\t` 都是合法的转义序列，然而 `\-` 不是。

这个问题确实有点麻烦。正则表达式是用来处理字符串的，但它又不完全等于字符串，正则表达式中的每个反斜线字符 `\`，在字符串中（也就是正则表达式之外）还必须转义为 `\\`。所以之前所说的是“正则表达式 `[0\ -9]`”，程序里写的却是 `[0\\ -9]`，这确实有点麻烦。

不过，Python 提供了 **原生字符串**（Raw String），它非常适合于正则表达式：正则表达式是怎样，原生字符串就是怎样，完全不需要考虑正则表达式之外的转义（只有双引号字符是例外，原生字符串内的双引号字符必须转义写成 `\"`）。原生字符串的形式是 `r"string"`，也就是在普通字符串之前添加 `r`，示例代码如例 1-10。

例 1-10 原生字符串的使用

```
#原生字符串和字符串的等价
r"^[0\ -9]$" == "[0\\ -9]$"          # => True
#原生字符串的转义要简单许多
re.search(r"^[0\ -9]$", "3") != None # => False
re.search(r"^[0\ -9]$", "-") != None # => True
```

原生字符串清晰易懂，省去了烦琐的转义，所以从现在开始，本书中的 Python 示范代码都会使用原生字符串来表示正则表达式。另外，.NET 和 Ruby 中也有原生字符串，也有一些语言并没有提供原生字符串（比如 Java），所以在第 6 章（**错误！未定义书签。**）会专门讲解转义问题。不过，现在只需要知道 Python 示范代码中使用了原生字符串即可。

继续看转义，如果希望在字符组中列出闭方括号 `]`，比如 `[012]345]`，就必须在它之前使用反斜线转义，写成 `[012\]345]`；否则，结果就如例 1-11 所示，正则表达式将 `]` 与最近的 `[` 匹配，这个表达式就成了“字符组 `[012]` 加上 4 个字符 `345]`”，它能匹配的是字符串 `0345]` 或 `1345]` 或 `2345]`，却不能匹配 `]`。

例 1-11 `]` 出现在不同位置，含义不同

```
#未转义的]
re.search(r"^[012]345]$", "2345") != None # => True
re.search(r"^[012]345]$", "5") != None    # => False
re.search(r"^[012]345]$", "]") != None    # => False
#转义的]
re.search(r"^[012\]345]$", "2345") != None # => False
re.search(r"^[012\]345]$", "5") != None    # => True
re.search(r"^[012\]345]$", "]") != None    # => True
```

除去字符组内部的 `]`，其他元字符的转义都必须在字符之前添加反斜线，`[` 的转义也是如此。如果只希望匹配字符串 `[012]`，直接使用正则表达式 `[012]` 是不行的，因为这会被识别为一个字符组，它只能匹配 `0`、`1`、`2` 这三个字符中的任意一个；而必须转义，把正则表达式写作 `\[012]`，请注意，只有开方括号 `[` 需要转义，闭方括号 `]` 不需要转义，如例 1-12 所示。

⁴ 具体来说，在 Java、PHP、Python、.NET 等语言中，正则表达式都是以字符串的形式给出的，在 Ruby 和 JavaScript 中则不是这样。详细的说明，请参考第 96 页。

例 1-12 取消其他元字符的特殊含义

```
re.search(r"^[012]345$", "3") != None      # => False
re.search(r"^[012\\]345$", "3") != None    # => True
re.search(r"^[012]$", "[012]") != None     # => False
re.search(r"^\[012]$", "[012]") != None    # => True
```

1.5 排除型字符组

在方括号 `[...]` 中列出希望匹配的所有字符，这种字符组叫做“普通字符组”，它的确非常方便。不过，也有些问题是普通字符组不能解决的。

给定一个由两个字符构成的字符串 `str`，要判断这两个字符是否都是数字字符，可以用 `[0-9][0-9]` 来匹配。但是，如果要求判断的是这样的字符串——第一个字符不是数字字符，第二个字符才是数字字符（比如 `A8`、`x6`）⁵——应当如何办？数字字符的匹配很好处理，用 `[0-9]` 即可；“不是数字”则很难办——不是数字的字符太多了，全部列出几乎不可能，这时就应当使用排除型字符组。

排除型字符组（Negated Character Class）非常类似普通字符组 `[...]`，只是在开方括号 `[` 之后紧跟一个脱字符 `^`，写作 `[^...]`，表示“在当前位置，匹配一个没有列出的字符”。所以 `[^0-9]` 就表示“`0~9`之外的字符”，也就是“非数字字符”。那么，`[^0-9][0-9]` 就可以解决问题了，如例 1-13 所示。

例 1-13 使用排除型字符组

```
re.search(r"^[^0-9][0-9]$", "A8") != None    # => True
re.search(r"^[^0-9][0-9]$", "x6") != None    # => True
```

排除型字符组看起来很简单，不过新手常常会犯一个错误，就是把“在当前位置，匹配一个没有列出的字符”理解成“在当前位置不要匹配列出的字符”两者其实是不同的，后者暗示“这里不出现任何字符也可以”。例 1-14 很清楚地说明：**排除型字符组必须匹配一个字符**，这点一定要记住。

例 1-14 排除型字符组必须匹配一个字符

```
re.search(r"^[^0-9][0-9]$", "8") != None     # => False
re.search(r"^[^0-9][0-9]$", "A8") != None    # => True
```

除了开方括号 `[` 之后的 `^`，排除型字符组的用法与普通字符组几乎完全相同，唯一需要改动的是：在排除型字符组中，如果需要表示横线字符 `-`（而不是用于“-范围表示法”），那么 `-` 应该紧跟在 `^` 之后；而在普通字符组中，作为普通字符的横线 `-` 应该紧跟在开方括号之后，如例 1-15 所示。

例 1-15 在排除型字符组中，紧跟在^之后的-不是元字符

```
#匹配一个-、0、9之外的字符
re.search(r"^[^-09]$", "-") != None          # => False
```

⁵ 一般来说，计算机中的偏移值都是从 0 开始的。此处考虑到叙述自然，使用了“第一个字符”和“第二个字符”的说法，其中“第一个字符”指最左端，也就是偏移值为 0 的字符；“第二个字符”指紧跟在它右侧，也就是偏移值为 1 的字符。

```

re.search(r"^[^0-9]$", "8") != None          # => True

#匹配一个 0~9 之外的字符
re.search(r"^[^0-9]$", "-") != None          # => True
re.search(r"^[^0-9]$", "8") != None          # => False

```

在排除型字符组中，`^`是一个元字符，但只有它紧跟在`[`之后时才是元字符，如果想表示“这个字符组中可以出现`^`字符”，不要让它紧挨着`[`即可，否则就要转义。例 1-16 给出了三个正则表达式，后两个表达式实质是一样的，但第三种写法很麻烦，理解起来也麻烦，不推荐使用。

例 1-16 排除型字符组的转义

```

#匹配一个 0、1、2 之外的字符
re.search(r"^[^012]$", "^") != None          # => True
#匹配 4 个字符之一：0、^、1、2
re.search(r"^[0^12]$", "^") != None          # => True
#^紧跟在[之后，但经过转义变为普通字符，等于上一个表达式，不推荐
re.search(r"^\[^012]$", "^") != None          # => True

```

1.6 字符组简记法

用`[0-9]`、`[a-z]`等字符组，可以很方便地表示数字字符和小写字母字符。对于这类常用的字符组，正则表达式提供了更简单的记法，这就是**字符组简记法**（shorthands）。

常见的字符组简记法有`\d`、`\w`、`\s`。从表面上看，它们与`[...]`完全没联系，其实是一致的。其中`\d`等价于`[0-9]`，其中的d代表“数字（digit）”；`\w`等价于`[0-9a-zA-Z_]`，其中的w代表“单词字符（word）”；`\s`等价于`[\t\r\n\v\f]`（第一个字符是空格），s表示“空白字符（space）”。例 1-17 说明了这几个字符组简记法的典型匹配。

例 1-17 字符组简记法\d、\w、\s

```

#如果没有原生字符串，\d 就必须写作\\d
re.search(r"^\d$", "8") != None              # => True
re.search(r"^\d$", "a") != None              # => False

re.search(r"^\w$", "8") != None              # => True
re.search(r"^\w$", "a") != None              # => True
re.search(r"^\w$", "_") != None              # => True

re.search(r"^\s$", " ") != None              # => True
re.search(r"^\s$", "\t") != None            # => True
re.search(r"^\s$", "\n") != None            # => True

```

一般印象中，单词字符似乎只包含大小写字母，但是字符组简记法中的“单词字符”不只有大小写单词，还包括数字字符和下划线`_`，其中的下划线`_`尤其值得注意：在进行数据验证时，有可能只容许输入“数字和字母”，有人会偷懒用`\w`验证，而忽略了`\w`能匹配下划线，所以这种匹配并不严格，`[0-9a-zA-Z_]`才是准确的选择。

“空白字符”并不难定义，它可以是空格字符、制表符`\t`，回车符`\r`，换行符`\n`等各种“空白”字符，只是不方便展现（因为显示和印刷出来都是空白）。不过这也提醒我们注意，匹配时看到的“空

白”可能不是空格字符，因此，`\s`才是准确的选择。

字符组简记法可以单独出现，也可以使用在字符组中，比如 `[0-9a-zA-Z]` 也可以写作 `[\da-zA-Z]`，所以匹配十六进制字符的字符组可以写成 `[\da-fA-F]`。字符组简记法也可以用在排除型字符组中，比如 `^[^0-9]` 就可以写成 `^[^d]`，`^[^0-9a-zA-Z]` 就可以写成 `^[^w]`，代码如例 1-18。

例 1-18 字符组简记法与普通字符组混用

```
#用在普通字符组内部
re.search(r"^[da-zA-Z]$", "8") != None # => True
re.search(r"^[da-zA-Z]$", "a") != None # => True
re.search(r"^[da-zA-Z]$", "C") != None # => True
#用在排除型字符组内部
re.search(r"^[^w]$", "8") != None # => False
re.search(r"^[^w]$", "_") != None # => False
re.search(r"^[^w]$", ",") != None # => True
```

相对于 `\d`、`\w` 和 `\s` 这三个普通字符组简记法，正则表达式也提供了对应排除型字符组的简记法：`\D`、`\W` 和 `\S`——字母完全一样，只是改为大写。这些简记法匹配的字符互补：`\s`能匹配的字符，`\S`一定不能匹配；`\w`能匹配的字符，`\W`一定不能匹配；`\d`能匹配的字符，`\D`一定不能匹配。例 1-19 示范了这几个字符组简记法的应用。

例 1-19 \D、\W、\S的使用

```
#\d和\D
re.search(r"^\d$", "8") != None # => True
re.search(r"^\d$", "a") != None # => False
re.search(r"^\D$", "8") != None # => False
re.search(r"^\D$", "a") != None # => True
#\w和\W
re.search(r"^\w$", "c") != None # => True
re.search(r"^\w$", "!") != None # => False
re.search(r"^\W$", "c") != None # => False
re.search(r"^\W$", "!") != None # => True
#\s和\S
re.search(r"^\s$", "\t") != None # => True
re.search(r"^\s$", "0") != None # => False
re.search(r"^\S$", "\t") != None # => False
re.search(r"^\S$", "0") != None # => True
```

妥善利用这种互补的属性，可以得到一些非常巧妙的效果，最简单的应用就是字符组 `[\s\S]`。初看起来，在同一个字符组中并列两个互补的简记法，这种做法有点奇怪，不过仔细想想就会明白，`\s` 和 `\S` 组合在一起，匹配的就是“所有的字符”（或者叫“任意字符”）。许多语言中的正则表达式并没有直接提供“任意字符”的表示法，所以 `[\s\S]`、`[\w\W]`、`[\d\D]` 虽然看起来有点古怪，但确实可以匹配任意字符⁶。

关于字符组简记法，最后需要补充两点：第一，如果字符组中出现了字符组简记法，最好不要出现单独的 `.`，否则可能引起错误，比如 `[\d-a]` 就很让人迷惑，在有些语言中，`.` 会被作为普通字符，而在有些语言中，这样写会报错；第二，以上说的 `\d`、`\w`、`\s` 的匹配规则，都是针对 ASCII 编码而

⁶ 许多关于正则表达式的文档说：点号 `.` 能匹配“任意字符”。但在默认情况下，点号其实不能匹配换行符，具体请参考第 84 页。

言的，也叫 **ASCII匹配规则**。但是，目前一些语言中的正则表达式已经支持了Unicode字符，那么数字字符、单词字符、空白字符的范围，已经不仅仅限于ASCII编码中的字符。关于这个问题，具体细节在后文有详细的介绍，如果你现在就想知道，可以翻到第**错误！未定义书签**。页。

1.7 字符组运算

以上介绍了字符组的基本功能，它们在常用的语言中都有提供；还有些语言中为字符组提供了更强大的功能，比如 Java 和 .NET 就提供了字符组运算的功能，可以在字符组内进行集合运算，在某些情况下这种功能非常实用。

如果要匹配所有的元音字母（为讲解简单考虑，暂时只考虑小写字母的情况），可以用 `[aeiou]`，但是要匹配所有的辅音字母却没有什么方便的办法，最直接的写法是 `[b-df-hj-np-tv-z]`，不但烦琐，而且难理解。其实，从 26 个字母中“减去”元音字母，剩下的就是辅音字母，如果有办法做这个“减法”，就方便多了。

Java语言中提供了这样的字符组：`[[a-z]&&[^aeiou]]`，虽然初看有点古怪，但仔细看看，也不难理解。`[a-z]`表示 26 个英文字母，`[^aeiou]`表示除元音字母之外的所有字符（还包括大写字母、数字和各种符号），两者取交集，就得到“26 个英文字母中，除去 5 个元音字母，剩下的 21 个辅音字母”。

.NET中也有这样的功能，只是写法不一样。同样是匹配辅音字母的字符组，.NET中写作 `[a-z-[aeiou]]`，其逻辑是：从 `[a-z]`能匹配的 26 个字符中，“减去”`[aeiou]`能匹配的元音字母。相对于Java，这种逻辑更符合直觉，但写法却有点古怪——不是 `[[a-z]-[aeiou]]`，而是 `[a-z-[aeiou]]`。例 1-20 集中演示了Java和.NET中的字符组运算。

例 1-20 字符组运算

```
Java
"a".matches("^[a-z]&&[^aeiou]$"); // => True
"b".matches("^[a-z]&&[^aeiou]$"); // => False

.NET
Regex.IsMatch("^[a-z-[aeiou]]$", "a"); // => True
Regex.IsMatch("^[a-z-[aeiou]]$", "b"); // => False
```

1.8 POSIX 字符组

前面介绍了常用的字符组，但是在某些文档中，你可能会发现类似 `[:digit:]`、`[:lower:]` 之类的字符组，看起来不难理解（digit就是“数字”，lower就是“小写”），但又很奇怪，它们就是 **POSIX 字符组**（POSIX Character Class）。因为某些语言的文档中出现了这些字符组，为避免困惑，这里有必要做个简要介绍。如果只使用常用的编程语言，可以忽略文档中的POSIX字符组，也可以忽略本节；如果了解POSIX字符组，或者需要在Linux/UNIX下的各种工具（sed、awk、grep等）中使用正则表达式，最好阅读本节。

之前介绍的字符组，都属于Perl衍生出来的正则表达式流派（Flavor），这个流派叫做PCRE（Per Compatible Regular Expression）。在此之外，正则表达式还有其他流派，比如POSIX（Portable Operating

System Interface for uniX)，它是一系列规范，定义了UNIX操作系统应当支持的功能，其中也包括关于正则表达式的规范，[:digit:]之类的字符组就是遵循POSIX规范的字符组。

常见的 [a-z] 形式的字符组，在POSIX规范中仍然获得支持，它的准确名称是 **POSIX方括号表达式** (POSIX bracket expression)，主要用在UNIX/Linux系统中。POSIX方括号表达式与之前所说的字符组最主要的差别在于：在POSIX字符组中，反斜线 \ 不是用来转义的。所以POSIX方括号表达式 [\d] 只能匹配 \ 和 d 两个字符，而不是 [0-9] 对应的数字字符。

为了解决字符组中特殊意义字符的转义问题，POSIX方括号表达式规定：如果要在字符组中表达字符] (而不是作为字符组的结束标记)，应当让它紧跟在字符组的开方括号之后，所以 []a 能匹配的字符就是] 或 a；如果要在字符组中标识字符 - (而不是“-范围表示法”)，就必须将它放在字符组的闭方括号] 之前，所以 [a-] 能匹配的字符就是 a 或 -。

另一方面，POSIX规范还定义了POSIX字符组 (POSIX character class)，它大致等于之前介绍的字符组简记法，都是使用类似 [:digit:]、[:lower:] 之类有明确意义的记号表示某类字符。

表 1-2 简要介绍了 POSIX 字符组，注意表格中与其对应的是 ASCII 字符组，也就是能匹配的 ASCII 字符 (ASCII 编码表中码值在 0~127 之间的字符)。因为 POSIX 规范中有一个重要概念：locale (通常翻译为“语言环境”)，它是一组与语言和文化相关的设定，包括日期格式、货币币值、字符编码等。POSIX 字符组的意义会根据 locale 的变化而变化，表 1-2 介绍的只是这些 POSIX 字符组在 ASCII 编码中的意义；如果换用其他的 locale (比如使用 Unicode 字符集)，它们的意义可能会发生变化，具体请参考第 **错误！未定义书签**。页。

表 1-2 POSIX字符组

| POSIX 字符组 | 说明 | ASCII 字符组 | 等价的 PCRE 简记法 |
|------------|-----------|-------------|--------------|
| [:alnum:]* | 字母字符和数字字符 | [a-zA-Z0-9] | |
| [:alpha:] | 字母 | [a-zA-Z] | |

(续表)

| POSIX 字符组 | 说明 | ASCII 字符组 | 等价的 PCRE 简记法 |
|------------|----------------------|------------------------------------|--------------|
| [:ASCII:] | ASCII 字符 | [\x00-\x7F] | |
| [:blank:] | 空格字符和制表符 | [\t] | |
| [:cntrl:] | 控制字符 | [\x00-\x1F\x7F] | |
| [:digit:] | 数字字符 | [0-9] | \d |
| [:graph:] | 空白字符之外的字符 | [\x21-\x7E] | |
| [:lower:] | 小写字母字符 | [a-z] | |
| [:print:] | 类似 [:graph:]，但包括空白字符 | [\x20-\x7E] | |
| [:punct:] | 标点符号 | [!\"#\$%&'()*+,-./:;<=>?@\^_`{ }~] | |
| [:space:] | 空白字符 | [\t\r\n\v\f] | \s |
| [:upper:] | 大写字母字符 | [A-Z] | |
| [:word:]* | 字母字符 | [A-Za-z0-9_] | \w |
| [:xdigit:] | 十六进制字符 | [A-Fa-f0-9] | |

注：标记*的字符组简记法并不是 POSIX 规范中的，但使用很多，一般语言中都提供，文档中也会出现。

POSIX 字符组的使用也与 PCRE 字符组简记法的使用有所不同，主要区别在于，PCRE 字符组简记法可以脱离方括号直接出现，而 POSIX 字符组必须出现在方括号内。所以同样是匹配数字字符，PCRE 中可以直接写 `\d`，而 POSIX 字符组必须写成 `[[:digit:]]`。

在本书介绍的 6 种语言中，Java、PHP、Ruby 支持使用 POSIX 字符组。

在 PHP 中可以直接使用 POSIX 字符组，但是 PHP 中的 POSIX 字符组只识别 ASCII 字符，也就是说，任何非 ASCII 字符（比如中文字符）都不能由任何一个 POSIX 字符组匹配。

Ruby 的情况稍微复杂一点。Ruby 1.8 中的 POSIX 字符组只能匹配 ASCII 字符，而且不支持 `[[:word:]]` 和 `[[:ASCII:]]`；Ruby 1.9 中的 POSIX 字符组可以匹配 Unicode 字符，而且支持 `[[:word:]]` 和 `[[:ASCII:]]`。

Java 中的情况更加复杂。POSIX 字符组 `[[:name:]]` 必须使用 `\p{name}` 的形式，其中 `name` 为 POSIX 字符组对应的名字，比如 `[[:space:]]` 就应当写作 `\p{Space}`，请注意第一个字母要大写，其他 POSIX 字符组都是这样，只有 `[[:xdigit:]]` 要写作 `\p{XDigit}`。并且 Java 中的 POSIX 字符组，只能匹配 ASCII 字符。

第 2 章 量词

2.1 一般形式

根据上一章的介绍，可以用字符组 `[0-9]` 或者 `\d` 匹配单个数字字符。现在用正则表达式来验证更复杂的字符串，比如大陆地区的邮政编码。

粗略来看，邮政编码并没有特殊的规定，只是 6 位数字构成的字符串，比如 `201203`、`100858`，所以用正则表达式来表示就是 `\d\d\d\d\d\d`，如例 2-1 所示，只有同时满足“长度是 6 个字符”和“每个字符都是数字”两个条件，匹配才成功（同样，这里不能忽略 `^` 和 `$`）。

例 2-1 匹配邮政编码

```
re.search(r"^\d\d\d\d\d\d$", "100859") != None      # => True
re.search(r"^\d\d\d\d\d\d$", "201203") != None      # => True

re.search(r"^\d\d\d\d\d\d$", "20A203") != None      # => False
re.search(r"^\d\d\d\d\d\d$", "20103") != None       # => False
re.search(r"^\d\d\d\d\d\d$", "2012036") != None     # => False
```

虽然这不难理解，但 `\d` 重复了 6 次，读写都不方便。为此，正则表达式提供了量词（quantifier），比如上面匹配邮政编码的表达式，就可以如例 2-2 那样，简写为 `\d{6}`，它使用阿拉伯数字，更简洁也更直观。

例 2-2 使用量词简化字符组

```
re.search(r"^\d{6}$", "100859") != None      # => True
re.search(r"^\d{6}$", "201203") != None      # => True

re.search(r"^\d{6}$", "20A203") != None      # => False
re.search(r"^\d{6}$", "20103") != None      # => False
re.search(r"^\d{6}$", "2012036") != None     # => False
```

量词还可以表示不确定的长度，其通用形式是 `{m,n}`，其中 `m` 和 `n` 是两个数字（有些人习惯在代码中的逗号之后添加空格，这样更好看，但是量词中的逗号之后绝不能有空格），它限定之前的元素⁷能够出现的次数，`m` 是下限，`n` 是上限（均为闭区间）。比如 `\d{4,6}`，就表示这个数字字符串的长度最短是 4 个字符（“单个数字字符”至少出现 4 次），最长是 6 个字符。

如果不确定长度的上限，也可以省略，只指定下限，写成 `\d{m,}`，比如 `\d{4,}` 表示“数字字符串的长度必须在 4 个字符以上”。

量词限定的出现次数一般都有明确下限，如果没有，则默认为 0。有一些语言（比如 Ruby）支持 `{,n}` 的记法，这时候并不是“不确定长度的下限”，而是省略了“下限为 0”的情况，比如 `\d{,6}` 表示“数字字符串最多可以有 6 个字符”。不过，这种用法并不是所有语言中都通用的，比如 Java 就不支持这种写法，所以必须写明 `{0,n}`。我推荐的做法是：最好使用 `{0,n}` 的记法，因为它是广泛支持的。表 2-1 集中说明了这几种形式的量词，例 2-3 展示了它们的使用。

表 2-1 量词的一般形式

| 量词 | 说明 |
|--------------------|---|
| <code>{n}</code> | 之前的元素必须出现 <code>n</code> 次 |
| <code>{m,n}</code> | 之前的元素最少出现 <code>m</code> 次，最多出现 <code>n</code> 次 |
| <code>{m,}</code> | 之前的元素最少出现 <code>m</code> 次，出现次数无上限 |
| <code>{0,n}</code> | 之前的元素可以不出现，也可以出现，最多出现 <code>n</code> 次（在某些语言中可以写为 <code>{,n}</code> ） |

例 2-3 表示不确定长度的量词

```
re.search(r"^\d{4,6}$", "123") != None      # => False
re.search(r"^\d{4,6}$", "1234") != None     # => True
re.search(r"^\d{4,6}$", "123456") != None   # => True
re.search(r"^\d{4,6}$", "1234567") != None  # => False

re.search(r"^\d{4,}$", "123") != None        # => False
re.search(r"^\d{4,}$", "1234") != None      # => True
re.search(r"^\d{4,}$", "123456") != None    # => True

re.search(r"^\d{0,6}$", "12345") != None     # => True
re.search(r"^\d{0,6}$", "123456") != None   # => True
re.search(r"^\d{0,6}$", "1234567") != None  # => False
```

⁷ 在上一章提到，字符组是正则表达式的基本“结构”之一，而此处提到之前的“元素”，在此做一点解释。在本书中，“结构”一般指的是正则表达式所提供功能的记法。比如字符组就是一种结构，下一章要提到的括号也是一种结构；而“元素”指的是具体的正则表达式中的某个部分，比如某个具体表达式中的字符组 `[a-z]`，可以算作一个元素，“元素”也叫“子表达式”（sub-expression）。

2.2 常用量词

`{m,n}`是通用形式的量词，正则表达式还有三个常用量词，分别是`+`、`?`、`*`。它们的形态虽然不同于`{m,n}`，功能却是相同的（也可以把它们理解为“量词简记法”），具体说明见表 2-2。

表 2-2 常用量词

| 常用量词 | {m,n}等价形式 | 说明 |
|------|-----------|----------------------|
| * | {0,} | 可能出现，也可能不出现，出现次数没有上限 |
| + | {1,} | 至少出现 1 次，出现次数没有上限 |
| ? | {0,1} | 至多出现 1 次，也可能不出现 |

在实际应用中，在很多情况只需要表示这三种意思，所以常用量词的使用频率要高于`{m,n}`，下面分别说明。

大家都知道，美国英语和英国英语有些词的写法是不一样的，比如 `traveler` 和 `traveller`，如果希望“通吃”`traveler`和`traveller`，就要求第 2 个 `l` 是“至多出现 1 次，也可能不出现”的，正好使用`?`量词：`travell?er`，如例 2-4 所示。

例 2-4 量词?的应用

```
re.search(r"^travell?er$", "traveler") != None # => True
re.search(r"^travell?er$", "traveller") != None # => True
```

其实这样的情况还有很多，比如 `favor` 和 `favour`、`color` 和 `colour`。此外还有很多其他应用场合，比如 `http` 和 `https`，虽然是两个概念，但都是协议名，可以用`https?`匹配；再比如表示价格的字符串，有可能是 100 也有可能是 ¥100，可以用`¥?100`匹配⁸。

量词也广泛应用于解析 HTML 代码。HTML 是一种“标签语言”，它包含各种各样的 tag（标签），比如 `<head>`、``、`<table>` 等，这些 tag 的名字各异，形式却相同：从 `<` 开始，到 `>` 结束，在 `<` 和 `>` 之间有若干字符，“若干”的意思是长度不确定，但不能为 0（`<>` 并不是合法的 tag），也不能是 `>` 字符⁹。如果要用一个正则表达式匹配所有的 tag，需要用 `<` 匹配开头的 `<`，用 `>` 匹配结尾的 `>`，用 `[^>]+` 匹配中间的“若干字符”，所以整个正则表达式就是 `<[^>]+>`，程序如例 2-5 所示。

例 2-5 量词+的应用

```
re.search(r"^<[^>]+>$", "<bold>") != None # => True
re.search(r"^<[^>]+>$", "</table>") != None # => True
re.search(r"^<[^>]+>$", "<>") != None # => False
```

类似的，也可以使用正则表达式匹配双引号字符串。不同的是，双引号字符串的两个双引号之间可以没有任何字符，`""` 也是一个完全合法的双引号字符串，应该使用量词 `*`，于是整个正则表达式就

⁸ 实际上，这个问题比较复杂，因为 `¥` 并不是一个 ASCII 字符，所以 `¥?` 可能会产生问题，具体情况请参考第 7 章。

⁹ 如果你对 HTML 代码比较了解，可能会有疑问，假如 tag 内部出现 `>` 符号，怎么办？这种情况确实存在，比如 `<input name=tst value=">">`。以目前已经讲解的知识还无法解决这个问题，不过下一章就会给出它的解法。

成了 "[^"]*"，程序见例 2-6。

例 2-6 量词*的应用

```
re.search(r"^\"[^\"]*$\"", "\\some\\") != None # => True
re.search(r"^\"[^\"]*$\"", "\\\"") != None # => True
```

注：字符串之中表示双引号需要转义写成 \，这并不是正则表达式中的规定，而是为字符串转义考虑。

量词的使用有很多学问，不妨多看几个tag匹配的例子：tag可以粗略分为open tag和close tag，比如 <head>就是open tag，而 </html>就是close tag；另外还有一类标签是self-closing tag，比如
。现在来看分别匹配这三类tag的正则表达式。

open tag的特点是以 <开头，然后是“若干字符”（但不能以 / 开头），最后是 >，所以对应的正则表达式是 <[^/][^>]*>；注意：因为 [^/] 必须匹配一个字符，所以“若干字符”中其他部分必须写成 [^>]*，否则它无法匹配名字为单个字符的标签，比如 。

close tag的特点是以 </开头，之后是 / 字符，然后是“若干字符（但不能以 / 开头）”，最后是 >，所以对应的正则表达式是 </[^>]+>；

self-closing tag的特点是以 </开头，中间是“若干字符”，最后是 />，所以对应的正则表达式是 <[^>]+/>。注意：这里不是 <[^>+]/>，排除型字符组只排除 >，而不排除 /，因为要确认的只是在结尾的 >之前出现 /，如果写成 <[^>+]/>，则要求tag内部不能出现 /，就无法匹配 这类的tag了。

表 2-3 列出了匹配几类 tag 的表达式。

表 2-3 各类tag的匹配

| 匹配所有 tag 的表达式 | tag 分类 | 匹配分类 tag 的表达式 |
|---------------|------------------|---------------|
| <[^>]+> | open tag | <[^/][^>]*> |
| | close tag | </[^>]+> |
| | self-closing tag | <[^>+]/> |

对比表格中“匹配所有 tag 的表达式”和“匹配分类 tag 的表达式”，可以发现它们的模式是相近的，只是细节上有差异。也就是说，通过变换字符组和量词，可以准确控制正则表达式能匹配的字符串的范围，达到不同的目的。这其实是使用正则表达式时的一条根本规律：使用合适的结构（包括字符组和量词），精确表达自己的意图，界定能匹配的文本。

再仔细观察，你或许会发现，匹配open tag的表达式，也可以匹配self-closing tag: <[^/][^>]*>能够匹配
，因为 [^>]* 并不排除对 / 的匹配。那么将表达式改为 <[^/][^>]*[^/]>，就保证匹配的open tag不会以 />结尾了。

不过这会产生新的问题：<[^/][^>]*[^/]>能匹配的tag，在 <和 >之间出现了两个 [^/]，上一章已经讲过，排除型字符组表示“在当前位置，匹配一个没有列出的字符”，所以tag里的字符串必须至少包含两个字符，这样就无法匹配 <u>了。

仔细想想，真正要表达的意思是，在tag内部的字符串不能以 / 开头，也不能以 / 结尾，如果这个字符串只包含一个字符，那么它既是开头，又是结尾，使用两个排除型字符组显然是不合适的，看起来没办法解决了。实际上，只是现有的知识还不足够解决这个问题而已，在第**错误！未定义书签**。页有这个问题的详细解法。

2.3 数据提取

正则表达式的功能很多，除去之前介绍的验证（字符串能否由正则表达式匹配），还可以从某个字符串中提取出某个字符串能匹配的所有文本。

上一章提到，`re.search()`如果匹配成功，返回一个`MatchObject`对象。这个对象包含了匹配的信息，比如表达式匹配的结果，可以像例 2-7 那样，通过调用`MatchObject.group(0)`来获得。这个方法以后详细介绍，现在只需要了解一点：调用它可以得到表达式匹配的文本。

例 2-7 通过MatchObject获得匹配的文本

```
#注意这里使用链式编程
print re.search(r"\d{6}", "ab123456cd").group(0)
123456
print re.search(r"^<[^>]+>$", "<bold>").group(0)
<bold>
```

这里再介绍一个方法：`re.findall(pattern, string)`。其中`pattern`是正则表达式，`string`是字符串。这个方法会返回一个数组，其中的元素是在`string`中依次寻找`pattern`能匹配的文本。

以邮政编码的匹配为例，假设某个字符串中包含两个邮政编码：`zipcode1:201203, zipcode2:100859`，仍然使用之前匹配邮政编码的正则表达式`\d{6}`，调用`re.findall()`可以将这两个邮政编码提取出来，如例 2-8。注意，这次要去掉表达式首尾的`^`和`$`，因为要使用正则表达式在字符串中寻找匹配，而不是验证整个字符串能否由正则表达式匹配。

例 2-8 使用re.findall()提取数据

```
print re.findall(r"\d{6}", "zipcode1:201203, zipcode2:100859")
['201203', '100859']

#也可以逐个输出
for zipcode in re.findall(r"\d{6}", "zipcode1:201203, zipcode2:100859"):
    print zipcode
201203
100859
```

借助之前的匹配各种tag的正则表达式，还可以通过`re.findall()`将某个HTML页面中所有的tag提取出来，下面以Yahoo首页为例。

首先要读入`http://www.yahoo.com/`的HTML源代码，在Python中先获得URL对应页面的源代码，保存到`htmlSource`变量中，然后针对匹配各类tag的正则表达式，分别调用`re.findall()`，获得各类tag的列表（因为这个页面中包含的tag太多，每类tag只显示前3个）。

因为这段程序的输出很多，在交互式界面下不方便操作和观察，建议将这些代码单独保存为一个.py文件，比如`findtags.py`，然后输入`python findtags.py`运行。如果输入`python`没有结果（一般在Windows下会出现这种情况），需要准确设定PATH变量，比如`d:\Python\python`。之后，就会看到例 2-9 显示的结果。

例 2-9 使用re.findall()提取tag

```

#导入需要的 package
import urllib
import re
#读入 HTML 源代码
sock = urllib.urlopen("http://yahoo.org/")
htmlSource = sock.read()
sock.close()
#匹配, 输出结果 ([0:3]表示取前 3 个)
print "open tags:"
print re.findall(r"<[^>]*[>]", htmlSource)[0:3]
print "close tags:"
print re.findall(r"</[^>]+>", htmlSource) [0:3]
print "self-closing tags:"
print re.findall(r"<[^>]+/>", htmlSource) [0:3]

open tags:
['<!DOCTYPE html>', '<html lang="en-US" class="y-fp-bg y-fp-pg-grad bkt701">',
'<!-- m2 template 0 -->']
close tags:
['</title>', '</script>', '</script>']
self-closing tags:
['<br/>', '<br/>', '<br/>']

```

2.4 点号

上一章讲到了各种字符组，与它相关的还有一个特殊的元字符：点号 `.`。一般文档都说，点号可以匹配“任意字符”，点号确实可以匹配“任意字符”，常见的数字、字母、各种符号都可以匹配，如例 2-10 所示。

例 2-10 点号的匹配

```

re.search(r"^. $", "a") != None      # => True
re.search(r"^. $", "0") != None      # => True
re.search(r"^. $", "*") != None      # => True

```

有一个字符不能由点号匹配，就是换行符 `\n`。这个字符平时看不见，却存在，而且在处理时并不能忽略（下一章会给出具体的例子）。

如果非要匹配“任意字符”，有两种办法：可以指定使用单行匹配模式，在这种模式下，点号可以匹配换行符（**错误！未定义书签。**）；或者使用上一章的介绍“自制”通配字符组 `[\s\S]`（也可以使用 `[\d\D]` 或 `[\w\W]`），正好涵盖了所有字符。例 2-11 清楚地说明，这两个办法都可以匹配换行符。

例 2-11 换行符的匹配

```

re.search(r"^. $", "\n") != None     # => False
#单行模式
re.search(r"(?s)^. $", "\n") != None # => True
#自制“通配字符组”
re.search(r"^[ \s\S] $", "\n") != None # => True

```

2.5 滥用品点号的问题

因为点号能匹配几乎所有的字符，所以实际应用中许多人图省事，随意使用 `.*` 或 `.+`，结果却事

与愿违，下面以双引号字符串为例来说明。

之前我们使用表达式 `"[^"]*"` 匹配双引号字符串，而“图省事”的做法是 `".*"`。通常这么用是没有问题的，但也可能有意外，例 2-12 就说明了一种如此。

例 2-12 “图省事”的意外结果

```
#字符串的值是"quoted string"
print re.search(r"\.*\"", "\"quoted string\"").group(0)
"quoted string"

#字符串的值是"quoted string" and another"
print re.search(r"\.*\"", "\"quoted string\" and another\"").group(0)
"quoted string" and another"
```

用 `".*"` 匹配双引号字符串，不但可以匹配正常的双引号字符串 `"quoted string"`，还可以匹配格式错误的字符串 `"quoted string" and another"`。这是为什么呢？

这个问题比较复杂，现在只简要介绍，以说明图省事导致错误的原因，更深入的原因涉及正则表达式的匹配原理，在第 8 章详细介绍。

在正则表达式 `".*"` 中，点号 `.` 可以匹配任何字符，`*` 表示可以匹配的字符串长度没有限制，所以 `.*` 在匹配过程结束以前，每遇到一个字符（除去无法匹配的 `\n`），`.*` 都可以匹配，但是到底是匹配这个字符，还是忽略它，将其交给之后的 `"` 来匹配呢？

答案是，具体选择取决于所使用的量词。在正则表达式中的量词分为几类，之前介绍的量词都可以归到一类，叫做 **匹配优先量词**（greedy quantifier，也有人翻译为**贪婪量词**¹⁰）。匹配优先量词，顾名思义，就是在拿不准是否要匹配的时候，优先尝试匹配，并且记下这个状态，以备将来“反悔”。

来看表达式 `".*"` 对字符串 `"quoted string"` 的匹配过程。

一开始，`"` 匹配 `"`，然后轮到字符 `q`，`.*` 可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以 `.*` 先匹配 `q`，并且记录下这个状态【`q`也可能是 `.*` 不应该匹配的】：

接下来是字符 `u`，`.*` 可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以 `.*` 先匹配 `u`，并且记录下这个状态【`u`也可能是 `.*` 不应该匹配的】：

.....

现在轮到字符 `g`，`.*` 可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以 `.*` 先匹配 `g`，并且记录下这个状态【`g`也可能是 `.*` 不应该匹配的】：

最后是末尾的 `"`，`.*` 可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以 `.*` 先匹配 `"`，并且记录下这个状态【`"`也可能是 `.*` 不应该匹配的】。

这时候，字符串之后已经没有字符了，但正则表达式中还有 `"` 没有匹配，所以只能查询之前保存备用的状态，看看能不能退回几步，照顾 `"` 的匹配。查询到最近保存的状态是：【`"`也可能是 `.*` 不应该匹配的】。于是让 `.*` “反悔”对 `"` 的匹配，把 `"` 交给 `"`，测试发现正好能匹配，所以整个匹配宣告成功。这个“反悔”的过程，专业术语叫做 **回溯**（backtracking），具体的过程如图 2-1 所示。

¹⁰ 许多文档都翻译为“贪婪量词”，单独来看这是没问题的，但考虑到正则表达式中还有其他类型的量词，其英文名字的形式较为统一，所以我在翻译《精通正则表达式》时采用了“匹配优先/忽略优先/占有优先”的名字，也未见读者反对，故此处延用此译法。

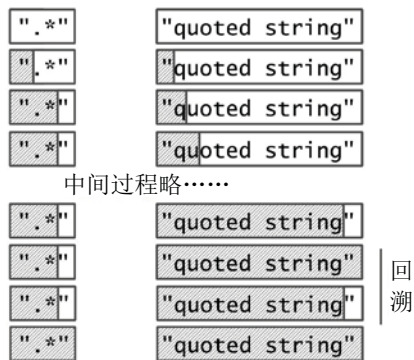


图 2-1 表达式 `.*` 对字符串 `"quoted string"` 的匹配过程



图 2-2 表达式 `.*` 的匹配过程

如果要准确匹配双引号字符串,就不能图省事使用 `.*`, 而需要使用 `"[^"]*"`, 过程如图 2-3 所示。

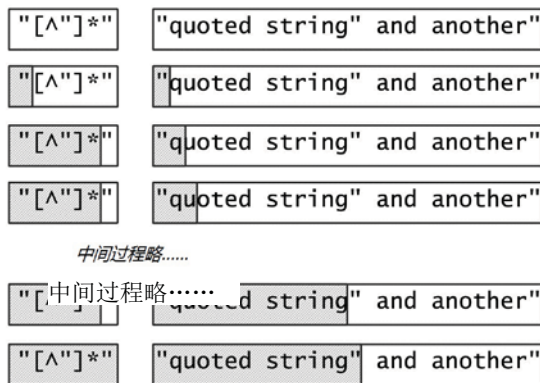


图 2-3 表达式 `"[^"]*"` 的匹配过程

2.6 忽略优先量词

也有些时候,确实需要用到 `.*` (或者 `[\s\S]*`), 比如匹配HTML代码中的JavaScript示例就是如此。

```
<script type="text/javascript">...</script>
```

匹配的模式仍然是: 匹配 `open tag` 和 `close tag`, 以及它们之间的内容。 `open tag` 是 `<script type="text/javascript">`, `close tag` 是 `</script>`, 这两段的内容是固定的, 非常容易写出对应的表达式, 但之间的内容怎么匹配呢? 在JavaScript代码中, 各种字符都可能出现, 所以不能用排除型字符

组，只能用 `.*`。比如，用一个正则表达式匹配下面这段HTML源代码：

```
<script type="text/javascript">
alert("some punctuation <>/");
</script>
```

开头和结尾的tag都容易匹配，中间的代码要比较麻烦，因为点号 `.` 不能匹配换行符，所以必须使用 `[\s\S]`（或者 `[\d\D]`、`[\w\W]`）。

```
<script type="text/javascript">[\s\S]*</script>
```

这个表达式确实可以匹配上面的 JavaScript 代码。但是如果遇到更复杂的情况就会出错，比如针对下面这段 HTML 代码，程序运行结果如例 2-13。

```
<script type="text/javascript">
alert("1");
</script>
<br />
<script type="text/javascript">
alert("2");
</script>
```

例 2-13 匹配JavaScript代码的错误

```
#假设上面的 JavaScript 代码保存在变量 htmlSource 中
jsRegex = r"<script type=\"text/javascript\">[\s\S]*</script>"
print re.search(jsRegex, htmlSource).group(0)
```

```
<script type="text/javascript">
alert("1");
</script>
<br />
<script type="text/javascript">
alert("2");
</script>
```

用 `<script type="text/javascript">[\s\S]*</script>` 来匹配，会一次性匹配两段 JavaScript 代码，甚至包含之间的非 JavaScript 代码。

按照匹配原理，`[\s\S]*` 先匹配所有的文本，回溯时交还最后的 `</script>`，整个表达式的匹配就成功了，逻辑就是如此，无可改进。而且，这个问题也不能模仿之前双引号字符串匹配，用 `[""]*` 匹配 `<script...>` 和 `</script>` 之间的代码，因为排除型字符组只能排除单个字符，`["</script>"]` 不能表示“不是 `</script>` 的字符串”。

换个角度来看，通过改变 `[\s\S]*` 的匹配策略解决问题：在不确定是否要匹配的场所，先尝试不匹配的选择，测试正则表达式中后面的元素，如果失败，再退回来尝试 `.*` 匹配，如此就没问题了。

循着这个思路，正则表达式中还提供了 **忽略优先量词**（lazy quantifier 或 reluctant quantifier，也有人翻译为 **懒惰量词**），如果不确定是否要匹配，忽略优先量词会选择“不匹配”的状态，再尝试表达式中之后的元素，如果尝试失败，再回溯，选择之前保存的“匹配”的状态。

对 `[\s\S]*` 来说，把 `*` 改为 `*?` 就是使用了忽略优先量词，`*?` 限定的元素出现次数范围与 `*` 完全一样，都表示“可能出现，也可能不出现，出现次数没有上限”。区别在于，在实际匹配过程中，遇到 `[\s\S]` 能匹配的字符，先尝试“忽略”，如果后面的元素（具体到这个表达式中，是 `</script>`）

不能匹配，再尝试“匹配”，这样就保证了结果的正确性，代码见例 2-14。

例 2-14 准确匹配JavaScript代码

```
#仍然假设 JavaScript 代码保存在变量 htmlSource 中
jsRegex = r"<script type=\"text/javascript\">[\s\S]*?</script>"
print re.search(jsRegex, htmlSource) .group(0)

<script type="text/javascript">
alert("1");
</script>

#甚至可以逐次提取出两段 JavaScript 代码
jsRegex = r"<script type=\"text/javascript\">[\s\S]*?</script>"
for jsCode in re.findall(jsRegex, htmlSource) :
    print jsCode + "\n"

<script type="text/javascript">
alert("1");
</script>

<script type="text/javascript">
alert("2");
</script>
```

从表 2-4 可以看到，匹配优先量词与忽略优先量词逐一对应，只是在对应的匹配优先量词之后添加 `?`，两者限定的元素能出现的次数也一样，遇到不能匹配的情况同样需要回溯；唯一的区别在于，忽略优先量词会优先选择“忽略”，而匹配优先量词会优先选择“匹配”。

表 2-4 匹配优先量词与忽略优先量词

| 匹配优先量词 | 忽略优先量词 | 限定次数 |
|--------------------|---------------------|-------------------------|
| <code>*</code> | <code>*?</code> | 可能不出现，也可能出现，出现次数没有上限 |
| <code>+</code> | <code>+?</code> | 至少出现 1 次，出现次数没有上限 |
| <code>?</code> | <code>??</code> | 至多出现 1 次，也可能不出现 |
| <code>{m,n}</code> | <code>{m,n}?</code> | 出现次数最少为 m 次，最多为 n 次 |
| <code>{m,}</code> | <code>{m,}?</code> | 出现次数最少为 m 次，没有上限 |
| <code>{,n}</code> | <code>{,n}?</code> | 可能不出现，也可能出现，最多出现 n 次 |

忽略优先量词还可以完成许多其他功能，典型的例子就是提取代码中的 C 语言注释。

C 语言的注释有两种：一种是在行末，以 `//` 开头；另一种可以跨多行，以 `/*` 开头，以 `*/` 结束。第一种注释很好匹配，使用 `//.*` 即可，因为点号 `.` 不能匹配换行符，所以 `//.*` 匹配的就是从 `//` 直到行末的文本，注意这里使用了量词 `*`，因为 `//` 可能就是该行最后两个字符；第二种注释稍微复杂一点，因为 `/*...*/` 的注释和 JavaScript 一样，可能分成许多段，所以必须用到忽略优先量词；同时因为注释可能横跨多行，所以必须使用 `[\s\S]`。因此，整个表达式就是 `/*[\s\S]*?*/`（别忘了 `*` 的转义）。

另一个典型的例子是提取出 HTML 代码中的超链接。常见的超链接形似 `text`。它以 `<a` 开头，以 `` 结束，`href` 属性是超链接的地址。我们无法预先判断 `<a>` 和 `` 之间到底会出现哪些字符，不会出现哪些字符，只知道其中的

内容一直到 `` 结束¹¹，程序代码见例 2-15。

例 2-15 提取网页中所有的超链接tag

```
#仍然获得 yahoo 网站的源代码，存放在 htmlSource 中
for hyperlink in re.findall(r"<a\s[\s\S]+?</a>", htmlSource):
    print hyperlink
#更多结果未列出
<a href="http://search.yahoo.com/">Web</a>
<a href="http://images.search.yahoo.com/images">Images</a>
<a href="http://video.search.yahoo.com/video">Video</a>
```

值得注意的是，在这个表达式中的 `<a` 之后并没有使用普通空格，而是使用字符组简记法 `\s`。HTML 语法并没有规定此处的空白只能使用空格字符，也没有规定必须使用一个空白字符，所以我们用 `\s` 保证“至少出现一个空白字符”（但是不能没有这个空白字符，否则就不能保证匹配 tag name 是 `a`）。

之前匹配 JavaScript 的表达式是 `<script language="text/javascript">[\s\S]*?</script>`，它能应对的情况实在太少了：在 `<script` 之后可能不是空格，而是空白字符；再之后可能是 `type="text/javascript"`，也可能是 `type="application/javascript"`，也可能用 `language` 取代 `type`（实际上 `language` 是以前的写法，现在大都用 `type`），甚至可能没有属性，直接是 `<script>`¹²。

所以必须改造这个表达式，将条件放宽：在 `script` 之后，可能出现空白字符，也可能直接是 `>`，这部分可以用一个字符组 `[\s>]` 来匹配，之后的内容统一用 `[\s\S]+?` 匹配，忽略优先量词保证了匹配进行到到最近的 `</script>` 为止。最终得到的表达式就是 `<script[\s>][\s\S]+?</script>`。

对这个表达式稍加改造，就可以写出匹配类似 tag 的表达式。在解析页面时，常见的需求是提取表格中各行、各单元（cell）的内容。表格的 tag 是 `<table>`，行的 tag 是 `<tr>`，单元的 tag 是 `<td>`，所以，它们可以分别用下面的表达式匹配，请注意其中的 `[\s>]`，它兼顾了可能存在的其他属性（比如 `<table border="1">`），同时排除了可能的错误（比如 `<table!>`）。

| | |
|----------|--|
| 匹配 table | <code><table[\s>][\s\S]+?</table></code> |
| 匹配 tr | <code><tr[\s>][\s\S]+?</tr></code> |
| 匹配 td | <code><td[\s>][\s\S]+?</td></code> |

在实际的 HTML 代码中，`table`、`tr`、`td` 这三个元素经常是嵌套的，它们之间存在着包含关系。但是，仅仅使用正则表达式匹配，并不能得到“某个 `table` 包含哪些 `tr`”、“某个 `td` 属于哪个 `tr`”这种信息。此时需要像例 2-16 的那样，用程序整理出来。

例 2-16 用正则表达式解析表格

```
# 这里用到了 Python 中的三重引号字符串，以便字符串跨越多行，细节可参考第 14 章
htmlSource = """<table>
<tr><td>1-1</td></tr>
<tr><td>2-1</td><td>2-2</td></tr>
```

¹¹ 根据 HTML 规范，`<a>` 这个 tag 可用来表示超链接，也可以用作书签，或兼作两种用途，考虑到书签的情况很少见，这里没有做特殊处理。

¹² 严格说起来，如果只出现 `<script>`，无法保证这里出现的就是 JavaScript 代码，也可能是 VBScript 代码，但考虑到真实世界中的情况，基本可以认为 `<script` 标识的“就是”JavaScript 代码，所以这里不作区分。

```
</table>""
```

```
for table in re.findall(r"<table[\s>][\s\S]+?</table>", htmlSource):
    for tr in re.findall(r"<tr[\s>][\s\S]+?</tr>", table):
        for td in re.findall(r"<td[\s>][\s\S]+?</td>", tr):
            print td,
            #输出一个换行符，以便显示不同的行
            print ""
<td>1-1</td>

<td>2-1</td> <td>2-2</td>
```

注：因为tag是不区分大小写的，所以如果还希望匹配大写的情况，则必须使用字符组，`table`写成 `[tT][aA][bB][lL][eE]`，`tr`写成 `[tT][rR]`，`td`写成 `[tT][dD]`。

这个例子说明，正则表达式只能进行纯粹的文本处理，单纯依靠它不能整理出层次结构；如果希望解析文本的同时构建层次结构信息，则必须将正则表达式配合程序代码一起使用。

回过头想想双引号字符串的匹配，之前使用的正则表达式是 `"[^"]*"`，其实也可以使用忽略优先量词解决 `".*?"`（如果双引号字符串中包含换行符，则使用 `"[\s\S]*?"`）。两种办法相比，哪个更好呢？

一般来说，`"[^"]*"`更好。首先，`[^"]`本身能够匹配换行符，涵盖了点号 `.`可能无法应付的情况，出于习惯，很多人更愿意使用点号 `.`而不是 `[\s\S]`；其次，匹配优先量词只需要考虑自己限定的元素能否匹配即可，而忽略优先量词必须兼顾它所限定的元素与之后的元素，效率自然大大降低，如果字符串很长，两者的速度可能有明显的差异。

而且，有些情况下确实必须用到匹配优先量词，比如文件名的解析就是如此。UNIX/Linux下的文件名类似这样 `/usr/local/bin/python`，它包含两个部分：路径是 `/usr/local/bin/`；真正的文件名是 `python`。为了在 `/usr/local/bin/python`中解析出两个部分，使用匹配优先量词是非常方便的。从字符串的起始位置开始，用 `.*`匹配路径，根据之前介绍的知识，它会回溯到最后（最右）的斜线字符 `/`，也就是文件名之前；在字符串的结尾部分，`[^/]*`能匹配的就是真正的文件名。前一章介绍过 `^`和 `$`，它们分别表示“定位到字符串的开头”和“定位到字符串的结尾”，所以应该把 `^`加在匹配路径的表达式之前，得到 `^.*`，而把 `$`加在匹配真正文件名的表达式之后，得到 `[^/]*$`，代码见例 2-17。

例 2-17 用正则表达式拆解Linux/UNIX的路径

```
print re.search(r"^.*/", "/usr/local/bin/python").group(0)
/usr/local/bin

print re.search(r"[^/]*$", "/usr/local/bin/python").group(0)
python
```

Windows下的路径分隔符是 `\`，比如 `C:\Program Files\Python 2.7.1\python.exe`，所以在正则表达式中，应该把斜线字符 `/`换成反斜线字符 `\`。因为在正则表达式中反斜线字符 `\`是用来转义其他字符的，为了表示反斜线字符本身，必须连写两个反斜线，所以两个表达式分别改为 `^.*\`和 `[^\\]*$`，代码见例 2-18。

例 2-18 用正则表达式拆解Windows的路径

```
#反斜线\必须转义写成\\
print re.search(r"^.*\\", "C:\\Program Files\\Python 2.7.1\\python.exe").group(0)
```

```
C:\Program Files\Python 2.7.1\
print re.search(r"^[^\\]*$", "C:\\Program Files\\Python 2.7.1\\python.exe").group(0)
python.exe
```

2.7 转义

前面讲解了匹配优先量词和忽略优先量词，现在介绍量词的转义¹³。

在正则表达式中，*、+、?等作为量词的字符具有特殊意义，但有些情况下只希望表示这些字符本身，此时就必须使用转义，也就是在它们之前添加反斜线 \。

对常用量词所使用的字符 +、*、?来说，如果希望表示这三个字符本身，直接添加反斜线，变为 \+、*、\?即可。但是在一般形式的量词 {m,n} 中，虽然具有特殊含义的字符不止一个，转义时却只需要给第一个 { 添加反斜线即可，也就是说，如果希望匹配字符串 {m,n}，正则表达式必须写成 \{m,n}。

另外值得一提的是忽略优先量词的转义，虽然忽略优先量词也包含不只一个字符，但是在转义时却不像一般形式的量词那样，只转义第一个字符即可，而需要将两个量词全部转义。举例来说，如果要匹配字符串 *?，正则表达式就必须写作 *\?，而不是 *?，因为后者的意思是“*这个字符可能出现，也可能不出现”。

表 2-5 列出了常用量词的转义形式。

表 2-5 各种量词的转义

| 量词 | 转义形式 |
|-------|--------|
| {n} | \{n} |
| {m,n} | \{m,n} |
| {m,} | \{m,} |

(续表)

| 量词 | 转义形式 |
|------|-------|
| {,n} | \{,n} |
| * | * |
| + | \+ |
| ? | \? |
| *? | *\? |
| +? | \+\? |
| ?? | \?\? |

之前还介绍了点号 .，所以还必须讲解点号的转义：点号 . 是一个元字符，它可以匹配除换行符之外的任何字符，所以如果只想匹配点号本身，必须将它转义为 \。

因为未转义的点号可以匹配任何字符，其中也可以包含点号，所以经常有人忽略了对点号的转义。如果真的这样做了，在确实需要严格匹配点号时就可能出错，比如匹配小数（如 3.14）、IP地址（如 192.168.1.1）、E-mail地址（如 someone@somehost.com）。所以，如果要匹配的文本包含点号，一定不要忘记转义正则表达式中的点号，否则就有可能出现例 2-19 那样的错误。

¹³ Java 等语言还支持“占有优先量词 (possessive quantifier)”，但这种量词较复杂，使用也不多，所以本书中不介绍占有优先量词。

例 2-19 忽略转义点号可能导致错误

```
#错误判断浮点数
print re.search(r"^\d+\d+$", "3.14") != None # => True
print re.search(r"^\d+\d+$", "3a14") != None # => True
#准确判断浮点数
print re.search(r"^\d+\.\d+$", "3.14") != None # => True
print re.search(r"^\d+\.\d+$", "3a14") != None # => False
```


正则指引

掌握正则表达式应该是IT工程师的一项标准技能，遗憾的是，过去，不少人多多少少忽视了这一点，所以在工作中总要应对正则表达式带来的「麻烦」。

我相信只有掌握并熟练运用它才有可能成为一个高效率的工程师。期待每个人手边都有一本正则表达式的参考书，当然，最好就是你现在看到的这本。

就在写这句话的几分钟前，我又从这本书中学到了一个有用的技巧。

冯大辉

配合恰当的案例，大量的反问，使读者自问、思考，扣人心弦，比较有代入感，加上配图，很容易让读者全面认识正则表达式。在原理讲解的章节，对比两种理论模型的区别，顺其自然地引入NFA引擎的关键要素——回溯，使读者从匹配原理上了解回溯，写出高效、严谨的正则表达式。

陈驰 <http://www.cnxct.com/>

本书由浅入深地讲述了正则表达式，在正则的应用和调优方面有非常详细的介绍，特别在正则表达式处理中文方面有独到的阐述，对于需要经常处理中文的国内技术人员来说，无疑是非常值得拥有的一本手册。

贺钧 <http://www.freecw.com>

正则表达式是程序员的必备知识。如果您还没有使用过这个强大的工具，或者学习正则表达式总不得要领，确实可以读读《正则指引》。

何源 <http://www.cppblog.com/lambdacpp>

这是一本通俗版的“精通正则表达式”。高手很难挑出毛病，一般程序员会受益匪浅，普通用户一步步读下去也能登堂入室。

张东亮 <http://iregex.org>

余晟在之前翻译业内名著的基础上，结合中文环境和自己的丰富经验，再接再厉推出自己的原创著作，实在是我等码农的一大幸事。

陈钢 <http://gossipcoder.com>



策划编辑：张月萍
责任编辑：葛娜

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

上架建议：计算机程序设计

ISBN 978-7-121-16551-1



9 787121 165511 >

定价：58.00元