



# 正则表达式 经典实例

使用8种编程语言的详细解决方案

包括一个正则表达式简明教程

[美] *Jan Goyvaerts* 著  
[美] *Steven Levithan* 著  
郭耀 译

O'REILLY®

人民邮电出版社  
POSTS & TELECOM PRESS

O'REILLY®

# 正则表达式经典实例

[美] Jan Goyvaerts Steven Levithan

郭耀 译

人民邮电出版社

北京



## 图书在版编目 (C I P) 数据

正则表达式经典实例 / (美) 高瓦特斯  
(Goyvaerts, J.) , (美) 利维森 (Levithan, S.) 著 ; 郭  
耀译. -- 北京 : 人民邮电出版社, 2010.6  
ISBN 978-7-115-22832-1

I. ①正… II. ①高… ②利… ③郭… III. ①正则表  
达式 IV. ①TP301. 2

中国版本图书馆CIP数据核字 (2010) 第076768号

## 版权声明

Copyright©2009 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2010. Authorized translation of the English edition, 2009 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版由 O'Reilly Media, Inc. 授权人民邮电出版社出版。未经出版者书面许可，对  
本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

## 正则表达式经典实例

- 
- ◆ 著 [美] Jan Goyvaerts Steven Levithan
  - 译 郭 耀
  - 责任编辑 刘映欣
  - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
  - 邮编 100061 电子函件 315@ptpress.com.cn
  - 网址 <http://www.ptpress.com.cn>
  - 北京鑫正大印刷有限公司印刷
  - ◆ 开本：787×1000 1/16
  - 印张：29.5
  - 字数：617 千字 2010 年 6 月第 1 版
  - 印数：1—4 000 册 2010 年 6 月北京第 1 次印刷

著作权合同登记号 图字：01-2009-6947 号

ISBN 978-7-115-22832-1

定价：69.00 元

读者服务热线：(010) 67132705 印装质量热线：(010) 67129223  
反盗版热线：(010) 67171154

# 前言

正则表达式在过去十多年来越来越普及。如今所有常用的编程语言都会包含一个强大的正则表达式函数库，或者甚至是在语言本身就内嵌了对于正则表达式的支持。许多开发人员都会利用这些正则表达式功能，在应用程序中为用户提供使用正则表达式对其数据进行查找或者过滤的能力。正则表达式真正是无处不在。

随着正则表达式的广泛采用，出现了许多相关的著作。大多数这类书籍都很好地讲解了正则表达式的语法，并且还会提供一些例子以及参考文献。然而，我们还没有看到有任何一本书能够面向现实世界中使用的计算机，以及在各种 Internet 应用中遇到的实际问题，为读者提供基于正则表达式的解决方案。因此，本书作者 Steve 和 Jan 决定写一本书来填补这种空白。

我们特别期望能够展现给读者的是：如何使用正则表达式来解决那些对正则表达式经验较少的人们认为无法解决的问题，以及软件纯粹主义者认为不能用正则表达式来解决的问题。因为如今正则表达式无处不在，所以它们通常可以作为工具，直接被最终用户使用，而不需要程序员团队的参与。即使是对于程序员来说，常常也可以在信息检索和更新的任务中采用一些正则表达式来节省大量时间，因为这些功能如果使用过程式代码来实现，可能就会需要几个小时甚至几天的时间，也可能会由于需要采用第三方的函数库，而不得不经过事先审查和经理层的审批。

## 不同版本带来的问题

与 IT 业界流行的东西一样，正则表达式也拥有许多种不同的实现，以及不同程度的兼容性。这就出现了许多不同的正则表达式流派（flavor），它们在处理一个特定正则表达式的时候并不总是拥有完全一样的表现，有时候甚至会无法正常使用。

在许多书中的确也提到了目前存在的不同流派，并且指出了其中的一些区别。但是，如果某种流派缺少特定功能的时候，它们通常会选择在这里或那里略掉一些流派，而不是为之提供可替代的解决方案或者是应急方案。而当你不得不在不同的应用程序或者不同程序语言中使用不同的正则表达式流派的时候，就会感到很受挫折。

在文献中常常会看到一些不严格的表达，例如“所有人现在都在使用 Perl 风格的正则表达式”，不幸的是这种说法会把很大范围的不兼容性边缘化。即使是“Perl 风格”的函数库也可能会有显著的区别，而且与此同时 Perl 也在持续不断地演化。一旦拥有了这种过度简单化的印象，就可能会导致有些程序员浪费长达半个小时的时间来运行调

试器却得不到任何结果，而不是去认真检查他们的正则表达式的实现细节。即使当他们发现所依赖的一些功能不存在的时候，也不一定总是知道该如何找到解决方案。

本书是市场上能看到的第一本讨论功能强大的各种常见正则表达式流派的书，并且本书从头到尾都会坚持这样的原则。

## 目标读者

如果你经常在计算机上处理文本，不管是搜索一大堆的文档，在文本编辑器中处理文本，抑或是开发需要搜索或处理文本的软件，都应该认真读一读这本书。正则表达式对于上述这些工作来说是一个非常优秀的工具。本书会教给你需要了解的关于正则表达式的所有东西，你不需要任何先前的经验。因为我们会从关于正则表达式的最基本的概念开始讲起。

如果你已经拥有关于正则表达式的经验，那么你会看到在其他教材和网上文章中经常一带而过的大量细节。如果你曾经遭遇过正则表达式在一个应用程序中可用，而在另外一个程序中不可用的情形，那么就会因为本书中对世界上最流行的 7 种正则表达式流派给出的翔实均衡的讲解，而感到受益颇多。我们把本书组织成一本经典实例（cookbook，原意为菜谱），从而可以直接跳到你想要细细阅读的话题。如果从头到尾阅读了整本书，你就会成为一个正则表达式的世界级“大厨”。

无论你是否是程序员，本书除了会教给你关于正则表达式所需知道的所有内容之外，还会讲解更多其他内容。如果你想要在文本编辑器、查找工具，或是任意含有带“正则表达式”标签的应用程序中使用正则表达式，那么你根本不需要任何编程经验就可以阅读本书。本书中的大多数例子都拥有完全基于一个或多个正则表达式的解决方案。

如果你是程序员，那么第 3 章会讲解在源代码中实现正则表达式所需的所有信息。本章假设读者对所选择的编程语言的基本语言特性是熟悉的，但是并不假设你在源代码中曾经使用过任何的正则表达式。

## 涉及的技术

.NET、Java、JavaScript、PCRE、Perl、Python 以及 Ruby，这些不只是一些用在封面上的热门词汇。它们是本书要讲到的 7 种正则表达式流派。我们会把这 7 种流派等同对待，还会特别小心地指出这些正则表达式流派中所能找到的所有不一致的地方。

关于编程的一章（第 3 章）中包含使用如下语言的代码示例：C#、Java、JavaScript、PHP、Perl、Python、Ruby 以及 VB.NET。虽然这样做会让这一章看起来有些重复，但是这样读者就可以很容易跳过那些不感兴趣的語言的讨论，而不会错过对于你所选择语言应当知道的任何内容。

# 本书的组织结构

本书的前 3 章讲解一些有用的工具和基本信息，这些会给读者提供使用正则表达式的基础。随后的每一章则介绍各种不同的正则表达式，并对文本处理的一个领域进行深入讲解。

第 1 章“正则表达式简介”讲解正则表达式的作用，并介绍了一系列工具，它们会使你学习、创建和调试正则表达式更加容易。

第 2 章“基本正则表达式技巧”介绍正则表达式的每个元素和特性，以及有效使用正则表达式的一些重要指南。

第 3 章“使用正则表达式编程”详细介绍了编码相关的技术，并且包含了在本书中涉及的每种编程语言中使用正则表达式的代码示例。

第 4 章“合法性验证和格式化”中包含如何处理常见用户输入的实例，例如日期、电话号码以及不同国家的邮政编码。

第 5 章“单词、文本行和特殊字符”探讨常用的文本处理任务，例如检查文本行中是否包含或者不包含某个特定的单词。

第 6 章“数字”会讲解如何检测整数、浮点数以及这种输入的几种其他格式。

第 7 章“URL、路径和 Internet 地址”展示如何能够把在 Internet 上和 Windows 系统中常用的这些字符串拆分开来，并且利用它们来查找数据。

第 8 章“标记语言和数据交换”讲解如何处理 HTML、XML、逗号分隔的取值 (CSV)，以及 INI 风格的配置文件。

## 阅读须知

本书中在排版上采用如下约定。

`<Regular•expression>` (正则表达式)

用来表示一个正则表达式，它可以单独出现，也可以出现在向某个应用程序的查找框中输入正则表达式的时候。如果不使用“宽松排列 (free-spacing)”模式，那么正则表达式中的空格会使用一个实心圆点来表示。

`«Replacement•text»` (替代文本)

用来表示在“查找和替换”的操作中，正则表达式所匹配的文本会被替换成的文本。在替代文本 (replacement text) 中的空格也会用一个实心圆点来表示。

Matched text (匹配文本)

用来表示与一个正则表达式相匹配的目标文本 (subject text) 中的一部分。

...

在正则表达式中的省略号会被用来说明在使用该正则表达式之前必须“把这里的空白填好”。相应的文字解释中会告诉你在其中应该填入什么样的内容。

#### CR、LF 和 CRLF

CR、LF 和 CRLF 放在黑框中用来表示在字符串中实际出现的换行字符，而不是正则表达式中的字符转义序列（character escapes），例如 \r、\n 和 \r\n。要创建这些字符，可以通过在应用程序的多行编辑面板中按回车键（Enter），或者也可以通过在源代码中使用多行字符常量，比如在 C# 中的逐字字符串（verbatim strings），或是 Python 语言中的三引号字符串（triple-quoted strings）。

这个符号表示回车箭头，它与键盘上的回车（Return 或 Enter）键上的符号一样，用来说明必须打断一行才能使之符合印刷页面的宽度。当你在源代码中键入这些代码的时候，不需要按回车键，而是应该把所有内容都键入同一行之中。



#### 提示

这个图标用来强调一个提示、建议或一般说明。



#### 警告

这个图标用来说明一个警告或注意事项。

## 代码示例的使用

本书的目的就是要帮助读者完成手头的工作。一般来说，读者可以随意在程序和文档中使用本书中出现的代码。除非你打算再利用本书中大量的代码，否则并不需要联系我们以获得许可。销售或者发布 O'Reilly 图书中包含示例的 CD-ROM 则必须要获得许可。引用本书或者引用其中的示例代码来回答问题并不需要获得许可。在你的产品文档中利用本书中的大量代码示例则需要获得许可。

如果读者在引用本书时提供出处，我们会很感激，虽然我们并不要求你一定这样做。提供出处的时候通常需要包括书名、作者、出版社和书号（ISBN）。例如：“Regular Expressions Cookbook, by Jan Goyvaerts and Steven Levithan. O'Reilly 2009, 978-0-596-2068-7.”。

如果你觉得对代码示例的使用可能会超出上面所给出的许可范围，或是属于合理使用的范围之外，那么请随时通过 [permissions@oreilly.com](mailto:permissions@oreilly.com) 联系我们。

## 联系我们

关于本书的意见和问题请按照如下地址与我们联系。

**美国：**

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

**中国：**

100035 北京市西城区西直门成铭大厦 C 座 807 室  
奥莱利技术咨询（北京）有限公司

我们还为本书建立了一个网页，其中包含了勘误表、示例和其他额外的信息。可以通过如下网址访问该网页：

<http://www.regexcookbook.com>

或者：

<http://oreilly.com/catalog/9780596520687>

关于本书的技术性问题或建议，请发邮件到：

[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

[info@mail.oreilly.com.cn](mailto:info@mail.oreilly.com.cn)

关于我们的书籍、会议、资源中心和 O'Reilly Network 的更多信息，请访问我们的网站：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

## 致谢

我们要感谢 O'Reilly Media, Inc.的编辑 Andy Oram，他从头到尾给我们提供了莫大的帮助。我们还要感谢 Jeffrey Friedl、Zak Greant、Nikolaj Lindberg 和 Ian Morse，他们提供的详细技术审阅意见使本书得以做到全面而准确。

## O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc.授权人民邮电出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc.是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 The Whole Internet User's Guide & Catalog (被纽约公共图书馆评为 20 世纪最重要的 50 本书之一) 到 GNN (最早的 Internet 门户和商业网站)，再到 WebSite (第一个桌面 PC 的 Web 服务器软件)，O'Reilly Media, Inc.一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc.具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以 O'Reilly Media, Inc.知道市场上真正需要什么图书。

## 内 容 提 要

本书讲解了基于 8 种常用的编程语言使用正则表达式的经典实例。书中提供了上百种可以在实战中使用的实例，以帮助读者使用正则表达式来处理数据和文本。对于如何使用正则表达式来解决性能不佳、误报、漏报等常见的错误以及完成一些常见的任务，本书给出了涉及基于 C#、Java、JavaScript、Perl、PHP、Python、Ruby 和 VB.NET 等编程语言的解决方案。

本书的读者对象是对正则表达式感兴趣的软件开发人员和系统管理员。本书旨在教会读者很多新的技巧以及如何避免语言特定的陷阱，读者可以通过本书提供的实例解决方案库来解决实践中的复杂问题。



# 计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

**Java** 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

**.Net** 技术精品资料下载汇总: **ASP.NET** 篇

**.Net** 技术精品资料下载汇总: **C#语言** 篇

**.Net** 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++** 编程语言学习资料尽收眼底 电子书+视频教程

**Visual C++(VC/MFC)** 学习电子书及开发工具下载

**Perl/CGI** 脚本语言编程学习资源下载地址大全

**Python** 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

**UML** 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

**Linux** 系统管理员必备参考资料下载汇总

**Linux shell**、内核及系统编程精品资料下载汇总

**UNIX** 操作系统精品学习资料<电子书+视频>分类总汇

**FreeBSD/OpenBSD/NetBSD** 精品学习资源索引 含书籍+视频

**Solaris/OpenSolaris** 电子书、视频等精华资料下载索引

# 目录

<b>第 1 章 正则表达式简介</b>	1
1.1 正则表达式的定义	1
1.2 使用正则表达式的工具	7
<b>第 2 章 正则表达式的基本技巧</b>	24
2.1 匹配字面文本	25
2.2 匹配不可打印字符	27
2.3 匹配多个字符之一	29
2.4 匹配任意字符	33
2.5 匹配文本行起始和/或文本行结尾	35
2.6 匹配整个单词	39
2.7 Unicode 代码点、属性、区块和脚本	42
2.8 匹配多个选择分支之一	52
2.9 分组和捕获匹配中的子串	54
2.10 再次匹配先前匹配的文本	57
2.11 捕获和命名匹配子串	59
2.12 把正则表达式的一部分重复多次	61
2.13 选择最小和最大重复次数	64
2.14 消除不必要的回溯	67
2.15 避免重复逃逸	69
2.16 检查一个匹配，但不添加到整体匹配中	71
2.17 根据条件匹配两者之一	77
2.18 向正则表达式中添加注释	79
2.19 在替代文本中添加字面文本	81
2.20 在替代文本中添加正则匹配	83
2.21 把部分的正则匹配添加到替代文本中	85
2.22 把匹配上下文插入到替代文本中	88
<b>第 3 章 使用正则表达式编程</b>	89
3.1 在源代码中使用字面正则表达式	94
3.2 导入正则表达式函数库	100

3.3	创建正则表达式对象	101
3.4	设置正则表达式选项	108
3.5	检查是否可以在目标字符串中找到匹配	114
3.6	检查正则表达式能否整个匹配目标字符串	121
3.7	获取匹配文本	126
3.8	决定匹配的位置和长度	132
3.9	获取匹配文本的一部分	137
3.10	获取所有匹配的列表	143
3.11	遍历所有匹配	148
3.12	在过程代码中对匹配结果进行验证	154
3.13	在另一个匹配中查找匹配	157
3.14	替换所有匹配	161
3.15	使用匹配的子串来替换匹配	168
3.16	使用代码中生成的替代文本来替换匹配	173
3.17	替换另一个正则式匹配中的所有匹配	179
3.18	替换另一个正则式匹配之间的所有匹配	181
3.19	拆分字符串	186
3.20	拆分字符串，保留正则匹配	194
3.21	逐行查找	199
<b>第 4 章 合法性验证和格式化</b>		203
4.1	E-mail 地址的合法性验证	203
4.2	北美电话号码的合法性验证和格式化	209
4.3	国际电话号码的合法性验证	213
4.4	传统日期格式的合法性验证	215
4.5	对传统日期格式进行精确的合法性验证	219
4.6	传统时间格式的合法性验证	224
4.7	检查 ISO 8601 格式的日期和时间	226
4.8	限制输入只能为字母数字字符	230
4.9	限制文本长度	232
4.10	限制文本中的行数	237
4.11	肯定响应的检查	241
4.12	社会安全号码的合法性验证	242
4.13	ISBN 的合法性验证	245
4.14	ZIP 代码的合法性验证	252
4.15	加拿大邮政编码的合法性验证	253
4.16	英国邮政编码的合法性验证	253

4.17	查找使用邮局信箱的地址 .....	254
4.18	转换姓名格式 .....	255
4.19	信用卡号码的合法性验证 .....	259
4.20	欧盟增值税代码 .....	265
<b>第 5 章</b>	<b>单词、文本行和特殊字符 .....</b>	<b>273</b>
5.1	查找一个特定单词 .....	273
5.2	查找多个单词之一 .....	275
5.3	查找相似单词 .....	277
5.4	查找除某个单词之外的任意单词 .....	281
5.5	查找后面不跟着某个特定单词的任意单词 .....	283
5.6	查找不跟在某个特定单词之后的任意单词 .....	284
5.7	查找临近单词 .....	287
5.8	查找重复单词 .....	293
5.9	删除重复的文本行 .....	294
5.10	匹配包含某个单词的整行内容 .....	298
5.11	匹配不包含某个单词的整行 .....	300
5.12	删除前导和拖尾的空格 .....	300
5.13	把重复的空白替换为单个空格 .....	303
5.14	对正则表达式元字符进行转义 .....	304
<b>第 6 章</b>	<b>数字 .....</b>	<b>309</b>
6.1	整数 .....	309
6.2	十六进制数字 .....	312
6.3	二进制数 .....	315
6.4	删除前导 0 .....	316
6.5	位于某个特定范围之内的整数 .....	317
6.6	在某个特定范围之内的十六进制数 .....	323
6.7	浮点数 .....	325
6.8	含有千位分隔符的数 .....	328
6.9	罗马数字 .....	329
<b>第 7 章</b>	<b>URL、路径和 Internet 地址 .....</b>	<b>332</b>
7.1	URL 合法性验证 .....	332
7.2	在全文中查找 URL .....	335
7.3	在全文中查找加引号的 URL .....	337
7.4	在全文中寻找加括号的 URL .....	338

7.5 把 URL 转变为链接.....	340
7.6 URN 合法性验证 .....	341
7.7 通用 URL 的合法性验证.....	343
7.8 从 URL 中提取通信协议方案.....	348
7.9 从 URL 中抽取用户名.....	350
7.10 从 URL 中抽取主机名.....	352
7.11 从 URL 中抽取端口号.....	354
7.12 从 URL 中抽取路径.....	355
7.13 从 URL 中抽取查询.....	358
7.14 从 URL 中抽取片段.....	359
7.15 域名合法性验证.....	360
7.16 匹配 IPv4 地址 .....	363
7.17 匹配 IPv6 地址 .....	365
7.18 Windows 路径的合法性验证 .....	378
7.19 分解 Windows 路径.....	381
7.20 从 Windows 路径中抽取盘符 .....	386
7.21 从 UNC 路径中抽取服务器和共享名 .....	387
7.22 从 Windows 路径中抽取文件夹 .....	388
7.23 从 Windows 路径中抽取文件名 .....	390
7.24 从 Windows 路径中抽取文件扩展名 .....	391
7.25 去除文件名中的非法字符.....	391
<b>第 8 章 标记语言和数据交换.....</b>	<b>393</b>
8.1 查找 XML 风格的标签.....	399
8.2 把标签<b>替换为<strong>.....	415
8.3 删掉除<em>和<strong>之外的所有 XML 风格标签.....	419
8.4 匹配 XML 名称.....	422
8.5 添加<p>和 标签将纯文本转换为 HTML .....	428
8.6 在 XML 风格的标签中查找某个特定属性.....	431
8.7 向不包含 cellspacing 属性的 <table>标签中添加该属性 .....	435
8.8 删除 XML 风格的注释.....	438
8.9 在 XML 风格的注释中查找单词.....	442
8.10 替换在 CSV 文件中使用的分隔符 .....	446
8.11 抽取某个特定列中的 CSV 域 .....	450
8.12 匹配INI段头 .....	453
8.13 匹配INI段块 .....	454
8.14 匹配INI名称-值对 .....	456

# 计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

**Java** 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

**.Net** 技术精品资料下载汇总: **ASP.NET** 篇

**.Net** 技术精品资料下载汇总: **C#语言** 篇

**.Net** 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++** 编程语言学习资料尽收眼底 电子书+视频教程

**Visual C++(VC/MFC)** 学习电子书及开发工具下载

**Perl/CGI** 脚本语言编程学习资源下载地址大全

**Python** 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

**UML** 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

**Linux** 系统管理员必备参考资料下载汇总

**Linux shell**、内核及系统编程精品资料下载汇总

**UNIX** 操作系统精品学习资料<电子书+视频>分类总汇

**FreeBSD/OpenBSD/NetBSD** 精品学习资源索引 含书籍+视频

**Solaris/OpenSolaris** 电子书、视频等精华资料下载索引

## 正则表达式简介

在你打开这本书的时候，很可能已经热切地期望，要在你的代码中插入本书章节中找到的那些包含诸多括号和问号的笨拙字符串了。如果你已经准备好要“即插即用”，我们非常欢迎，实际的正则表达式会在第 4~8 章中给出并加以讲解。

但是从长远来看，阅读本书的最初几章会节省你大量的时间。例如，本章会向读者介绍许多工具——其中一些工具是本书作者之一的 Jan 所创建的，这些工具会帮助你测试和调试一个正则表达式，而不会等到把它们埋藏到代码中之后，那时候错误就非常难以查找了。而且这最初几章还会向读者展示如何使用正则表达式的不同特性和选项，帮助你轻松应对遇到的问题，并帮助你理解正则表达式，从而可以提高它们的性能，以及学习不同语言——甚至是您最喜欢的编程语言的不同版本之间——在处理正则表达式的时候出现的细微差别。

因此，我们在这些背景内容上花费了大量的精力，相信读者在开始读本书之前会阅读这些内容，或是在使用正则表达式时受到挫折而想要巩固你的理解的时候，会回头来阅读它们。

### 1.1 正则表达式的定义

在本书的上下文中，正则表达式（regular expression）是一种可以在许多现代应用程序和编程语言中使用的特殊形式的代码模式。可以使用它们来验证输入是否符合给定的文本模式；在一大段文本中查找匹配该模式的文本；用其他文本来替换匹配该模式的文本或者重新组织匹配文本的一部分；把一块文本划分成一系列更小的文本；或者是搬起石头砸自己的脚。本书会帮助你确切理解正在做的事情，从而避免可能会造成的灾难性后果。

## 术语“正则表达式”的历史

术语“正则表达式”来源于数学与计算机科学理论，它用来反映被称为“正则性”的数学表达式特点。这样一个表达式可以通过一个确定性有限自动机（DFA）用软件来实现。一个 DFA 是一个不使用回溯的有限状态机。

最早版本的 grep 工具所使用的文本模式是数学意义上的正则表达式。尽管名字看起来是一样的，然而如今流行的 Perl 风格的正则表达式已经完全不是数学意义上的正则表达式了。它们是采用非确定性的有限自动机（NFA）来实现的。你稍后就会学到和回溯有关的所有内容。关于这条说明，一个实践中的程序员需要记住的所有内容就是：象牙塔里的一些计算机科学家对于他们拥有良好定义的术语，被用于现实世界中更为有用的技术而感到非常不满。

如果学会了使用正则表达式的技巧，它们就会简化许多编程和文本处理的任务，并且使得许多没有正则表达式根本无法实现的任务成为可能。从一个文档中提取所有的 E-mail 地址，至少需要几十行，甚至是几百行过程式代码——这些代码编写起来费事，而且也很难维护。但是如果采用了合适的正则表达式，例如在实例 4.1 中所给的一样，那么就只需要很少的几行代码，或者甚至只要一行代码就可以了。

但是如果你试图想要用一个正则表达式来做太多的事情，或者是在事实上根本不适合的情形中非要使用正则表达式，那么就会明白为什么会有存在如下的说法<sup>1</sup>：

有些人每当遇到一个问题的时候，就会想“我知道怎么做，用正则表达式就可以了。”那么现在他们就有两个问题需要解决了。

这些人所遇到的第二个问题指的就是他们并不会去阅读用户手册，也就是你现在手里拿到的这本书。请继续读下去。正则表达式是一个强大的工具。如果你的工作会涉及到在计算机上操作或是抽取文本，那么牢固地掌握正则表达式就会为你省去很多个不眠之夜。

## 多种正则表达式流派

说实话，上一小节的标题是在说谎。我们并没有定义正则表达式到底是什么。我们也不可能给出定义。对于哪些文本模式是正则表达式，而哪些不是正则表达式，并不存在正式的标准来给出恰如其分的定义。你可以想象得到，每种编程语言的设计人员，以及每个文本处理程序的开发人员，对于正则表达式应该是什么样子都会都有自己不同的想法。因此，我们就不得不面对这样一大堆不同的正则表达式流派。

幸运的是，绝大多数设计人员与开发人员都比较懒惰。如果你可以复制别人已经做好的工作，为什么非要自己创建一些全新的东西呢？正因为此，所有现代的正则表达

<sup>1</sup> Jeffrey Friedl 在他的博客 <http://regex.info/blog/2006-09-15/247> 中探讨了这句话的来源和历史。

式流派，其中当然包含了本书要讨论的这些流派，都可以把它们的历史追溯到 Perl 这种编程语言。我们把这些流派都称作 Perl 风格的正则表达式。它们的正则表达式语法是非常相似的，而且在大多数情况下是兼容的，但是并不是在所有情况下都完全兼容。即使是作者有时候也会偷懒。在本书中，我们通常会使用 `regex` 或者 `regexp` 来指代一个单个的正则表达式，而使用 `regexes` 来指代其复数形式<sup>1</sup>。

正则流派并不是和编程语言一一对应的。脚本语言倾向于拥有它们自己的、内置的正则表达式流派。其他语言则会依赖于函数库来提供正则表达式支持。有些函数库是对多种语言可用的，而某些特定的语言则会选用一些不同的函数库。

本章要讲解的内容只与正则表达式的流派有关，因此会彻底忽略任何与编程有关的考虑事项。从第 3 章开始，我们会给出一些代码列表，因此你可以先跳到第 3 章的“编程语言与正则表达式流派”一节，以了解你将会使用哪些流派来工作。但是现在请先忽略所有与编程相关的内容。下面一小节中列出的工具将会通过“动手学习”的方式，让你可以更方便地探索正则表达式的语法。

## 本书涉及的正则流派

在本书中，我们选择了如今在使用中的最为流行的正则流派。这些都是 Perl 风格的正则流派。有些流派会比其他流派多一些特性。但是如果两种流派拥有同一个特性的话，那么它们通常都会使用相同的语法。当然也会有例外，当我们遇到这些烦人的不一致情况时，在书中会加以提示。

所有这些正则流派都属于目前正在活跃开发中的编程语言和函数库的一部分。下面的流派列表会告诉你本书所用到的是哪些版本。在本书后面，如果所讲解的正则表达式在所有流派中效果都一样，那么我们在提到该流派时就会不区分其版本。而在几乎所有情况下都会是这种情况。除了会影响到一些边界情况的 bug 修复之外，正则表达式流派一般都不会改变，唯一例外是添加新的特性，从而原来被认为是错误的语法现在会被指定新的含义。

### Perl

Perl 对于正则表达式的内置支持是正则表达式今天得以流行的主要原因。本书会涉及 Perl 5.6、Perl 5.8 以及 Perl 5.10。

许多应用程序和正则库都号称它们使用的是 Perl 或者与 Perl 兼容的正则表达式，而事实上它们仅仅是使用了 Perl 风格的正则表达式。它们使用一种与 Perl 相似的正则语法，但是并不支持相同的正则特性集。最有可能的情形是，它们使用的是这一特性集中的正则表达式流派之一，而这些流派都是属于 Perl 风格的。

---

<sup>1</sup> 译者注：本书中将“Regular Expression”翻译为“正则表达式”，“regex”、“regexp”或者“regexes”则简称为“正则式”或“正则”。

## PCRE

PCRE 是由 Philip Hazel 开发的“与 Perl 兼容的正则表达式（Perl-Compatible Regular Expressions）”的 C 语言函数库。读者可以从 <http://www.pcre.org> 下载开源的库代码。本书会涉及的 PCRE 版本包括第 4 版到第 7 版。

虽然 PCRE 号称是 Perl 兼容的，而且与本书中的其他流派相比，它也是与 Perl 兼容性最好的，但是实际上它也只能称为是“Perl 风格”的正则流派。有些特性，比如 Unicode 支持，会与 Perl 稍微有些不同；并且不能像在 Perl 中所允许的那样把 Perl 代码混合到你的正则表达式之中。

因为它采用了开源许可，并且拥有稳定可靠的实现，所以 PCRE 被应用到了许多编程语言和程序中。它被内置到 PHP 中，并且被包装到了许多个 Delphi 的组件中。如果一个应用程序号称它支持“Perl 兼容”的正则表达式，而却没有具体列出它实际使用的正则流派，那么就很可能是 PCRE。

## .NET

微软公司的.NET 框架，通过 System.Text.RegularExpressions 包，提供了一个功能完整的 Perl 风格的正则流派。本书会涉及.NET 的 1.0~3.5 版。而严格来讲，只存在两个版本的 System.Text.RegularExpressions，即 1.0 和 2.0 版。在 .NET 1.1、.NET 3.0 和.NET3.5 中，并没有对正则表达式的类进行任何修改。

任何 .NET 编程语言，包括 C#、VB.NET、Delphi for .NET，以及甚至是 COBOL.NET，都可以完全访问 .NET 正则流派。如果一个使用 .NET 开发的程序为你提供了正则表达式支持，那么即使它号称使用了“Perl 正则表达式”，你也可以完全确定它使用的是 .NET 正则流派。令人大跌眼镜的一个例外则是 Visual Studio (VS) 自身。VS 集成开发环境 (IDE) 中依然使用的是它一开始就在用的同一个老版本的正则流派，而这个实现根本就不是 Perl 风格的。

## Java

Java 4 是提供内置正则表达式支持的第一个 Java 版本，它通过 java.util.regex 包提供对正则表达式的支持。这个包很快就超越了为 Java 提供的各种第三方正则库。除了它是标准的和内置的之外，它还提供了功能完整的 Perl 风格的正则流派，并且它的卓越性能完全可以与即使用 C 编写的应用程序不相上下。本书会讲解在 Java 4、Java 5 和 Java 6 中的 java.util.regex 包。

如果你在过去几年中用 Java 开发软件的话，那么使用的任意正则表达式支持都很可能是这里所谓的 Java 流派。

## JavaScript

在本书中，我们使用 JavaScript 这个术语指代在 ECMA-262 标准的第 3 版中定义的

正则表达式流派。这个标准定义了 ECMAScript 编程语言，而这个语言更广为人知的是它在不同网页浏览器中的 JavaScript 与 JScript 实现。Internet Explorer 5.5 到 Internet Explorer 8.0、Firefox、Opera 与 Safari 都实现了 ECMA-262 的第 3 版。然而，所有的浏览器都拥有各种不同的边界情形 bug，使得它们与该标准有所背离。我们会在必要的地方指出这些问题。

如果一个网站允许使用正则表达式进行查找或者过滤，并且不用等待网站服务器的响应，那么它使用的就是 JavaScript 正则流派，这是唯一的可以跨浏览器的客户端正则流派。即使是微软的 VBScript 与 Adob 公司的 ActionScript 3 使用的也是它。

## Python

Python 通过它的 `re` 模块来支持正则表达式。本书会讲到 Python 2.4 和 Python 2.5。Python 的正则表达式支持已经多年没有发生变化了。

## Ruby

Ruby 的正则表达式支持是 Ruby 语言自身的一部分，这与 Perl 语言类似。本书会涉及 Ruby 1.8 和 Ruby 1.9。Ruby 1.8 的默认编译会使用由 Ruby 源代码直接提供的正则表达式流派。而 Ruby 1.9 的默认编译则会使用 Oniguruma 正则表达式库。Ruby 1.8 也可以被编译来使用 Oniguruma，而 Ruby 1.9 也可以被编译来使用较早版本的 Ruby 正则流派。在本书中，我们会把内置的 Ruby 流派称为 Ruby 1.8，而把 Oniguruma 流派称为 Ruby 1.9。

如果想测试一下你的站点使用的是哪个 Ruby 正则流派，你可以尝试用一下正则表达式 `\a{++}`。Ruby 1.8 会说这个正则表达式是非法的，因为它并不支持占有量词 (possessive quantifiers)，而它在 Ruby 1.9 中则会匹配一个或者多个字符 `a` 组成的字符串。

Oniguruma 库被设计为与 Ruby 1.8 向后兼容，它只是在其上添加了新的功能，而不会破坏已有的正则表达式。该实现甚至把人们认为应该修改的功能也原封不动，例如：它依然使用 `(?m)` 表示“点号匹配换行符”，虽然其他的正则表达式流派使用的都是 `(?s)`。

## 使用正则表达式进行查找和替换

查找和替换 (Search-and-replace) 对正则表达式来说是一个非常常见的任务。一个查找和替换的功能，会接受一个目标字符串、一个正则表达式和一个替代字符串作为输入。它的输出则是把目标字符串中所有与正则表达式相匹配的字符串都替换为“替代文本”。

虽然替代文本 (replacement text) 并不是一个正则表达式，读者也可以使用某些特殊的

语法构造动态的替代文本。所有的流派都允许把正则表达式匹配到的文本或者某个捕获分组，重新添加到替代字符串中。实例 2.20 和实例 2.21 会对此加以讲解。有些流派还会支持把匹配的上下文添加到替代文本中，这会在实例 2.22 中讲解。在第 3 章中，实例 3.16 将教你如何在代码中为每个匹配都生成一个不同的替代文本。

## 多种替代文本流派

由于不同正则表达式软件开发人员的不同想法，就造成了非常多的正则表达式流派，每种流派都拥有不同的语法和特性集合。而对于替代文本来说也是一样的。事实上，替代文本拥有比正则表达式更多的流派。构造一个正则表达式引擎是非常困难的。大多数程序员都倾向于复用一个已有的引擎，而在一个已有的正则表达式引擎上绑定一个查找和替换的功能，则是相当容易的。这样做造成的结果是，对于不拥有内置查找和替换功能的正则表达式库来说，它们之上就会存在许多替代文本的流派。

幸运的是，除了 PCRE 之外，本书中所有的正则表达式流派都拥有相对应的替代文本流派。而 PCRE 中的这个问题则使得使用基于它的流派的程序员感到无所适从。结果是，所有基于 PCRE 的应用程序和编程语言都需要提供它们自己的查找和替换功能。大多数程序员都会试图去复制已有的语法，但是却从来不会按照完全相同的方式去做。

本书会讲到如下的替代文本流派。关于同这些替代文本流派相对应的正则表达式流派，请参考前面的“多种正则表达式流派”一节。

### Perl

Perl 使用 `s/regex/replace/` 操作符提供内置的正则表达式替换支持。Perl 的替代文本流派是与 Perl 的正则表达式流派相对应的。本书会涉及 Perl 5.6~Perl 5.10。后期的版本在替代文本中添加了命名向后引用（named backreference）的支持，并在正则表达式语法中添加了命名捕获（named capture）。

### PHP

在本书中，PHP 替代文本流派指的是在 PHP 中的 `preg_replace` 功能。这个功能使用了 PCRE 的正则表达式流派与 PHP 的替代文本流派。

其他使用 PCRE 的编程语言并没有使用与 PHP 相同的替代文本流派。根据所使用的编程语言的设计者是从哪里得到的灵感，其中替代文本的语法可能会与 PHP 相同，或者也可能与本书中任何一种其他替代文本流派相似。

PHP 还拥有一个 `ereg_replace` 功能。这个功能使用一种不同的正则表达式流派（POSIX ERE），以及一种不同的替代文本流派。PHP 中的 `ereg` 功能不在本书讨论的范围之内。

## .NET

`System.Text.RegularExpressions` 包中提供了各种不同的查找和替换功能。.NET 替代文本流派对应.NET 的正则表达式流派。所有的.NET 版本都使用相同的替代文本流派。.NET 2.0 中新的正则表达式功能并不会影响替代文本的语法。

## Java

`java.util.regex` 包中包含了内置的查找和替换功能。本书会涉及 Java 4、Java5 和 Java6。所有版本都会使用同样的替代文本语法。

## JavaScript

在本书中，我们使用 JavaScript 这个术语，指代在 ECMA-262 标准的第 3 版中所定义的替代文本流派与正则表达式流派。

## Python

Python 中的 `re` 模块提供了一个 `sub` 功能用于查找和替换。Python 的替代文本流派对应于 Python 正则表达式流派。本书将涉及 Python 2.4 和 Python 2.5。Python 的正则表达式支持，多年来已经非常稳定了。

## Ruby

Ruby 的正则表达式支持是 Ruby 语言自身的一部分，这其中也包括了查找和替换功能。本书会涉及到 Ruby 1.8 和 Ruby 1.9。Ruby 1.8 的默认编译会使用由 Ruby 源代码直接提供的正则表达式流派。而 Ruby 1.9 的默认编译则会使用 Oniguruma 正则表达式库。Ruby 1.8 也可以被编译来使用 Oniguruma，而 Ruby 1.9 也可以被编译来使用较早版本的 Ruby 正则表达式流派。在本书中，我们会把内置的 Ruby 流派称为 Ruby 1.8，而把 Oniguruma 流派称为 Ruby 1.9。

Ruby 1.8 和 Ruby 1.9 中的替代文本语法是相同的，唯一的例外是 Ruby 1.9 在替代文本中添加了对于命名向后引用的支持。命名捕获也是在 Ruby 1.9 的正则表达式中的一个新的特性。

## 1.2 使用正则表达式的工具

除非你已经拥有了相当长的使用正则表达式编程的经验，否则，我们建议你首先在一个工具中试验一下正则表达式，而不是一下子就在源代码中使用它们。本章和第 2 章中提供的正则表达式示例都是简单（plain）的正则表达式，其中并不包含编程语言（即使是 UNIX shell）所必需的额外的转义符号。因此你可以直接把这些正则表达式输入到一个应用程序的查找框中。

第 3 章讲解如何把正则表达式混合到源代码中。把一个字面的（literal）正则表达式作

为一个字符串引用会把它变得更加难懂，因为字符串的转义规则会与正则表达式的转义规则混杂在一起。我们在实例 3.1 才会开始讲解这些内容。一旦理解了正则表达式的基本内容，你就能够从无数的反斜杠的背后看到“森林”。

本节要介绍的工具同样会提供调试和语法检查的功能，以及在绝大多数编程环境中并不会获得的其他反馈信息。因此，在应用程序中开发正则表达式的时候，你可能会发现在把一个复杂的正则表达式插入到程序中之前，首先使用这类工具试验一下可能会非常有用。

## RegexBuddy

在本书写作之时，**RegexBuddy**（如图 1-1 所示）是用来创建、测试和实现正则表达式功能最为丰富的工具。它拥有独特的能力，可以仿真本书中讲到的所有正则表达式的流派，甚至可以在不同的流派之间进行转换。

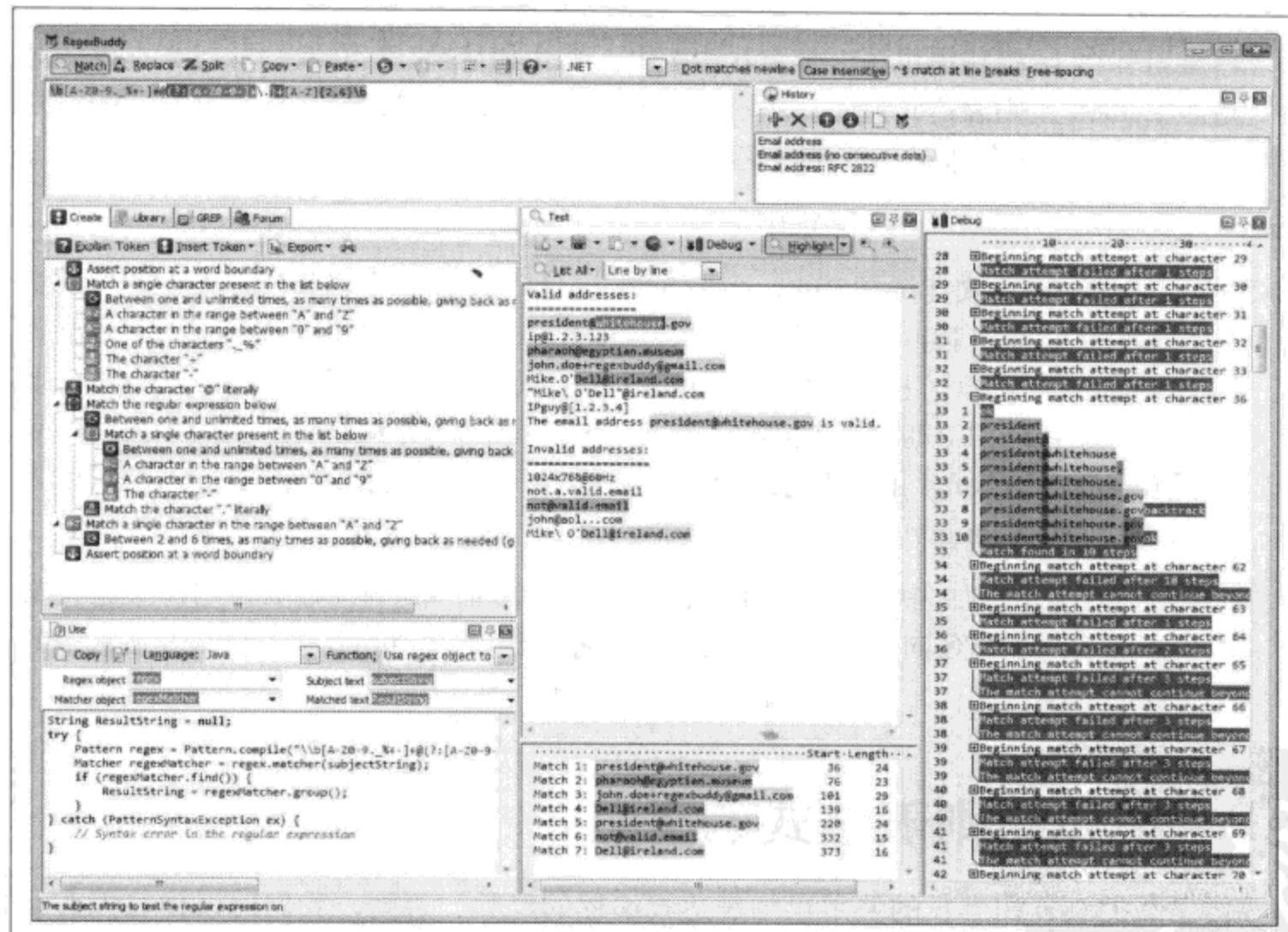


图 1-1 RegexBuddy

RegexBuddy 是由本书作者之一的 Jan Goyvaerts 设计和开发的。设计和开发 RegexBuddy 使 Jan 成为了正则表达式的专家，而由于使用了 RegexBuddy，使得本书的共同作者 Steven 迷恋上了正则表达式，以至于他向 O'Reilly 出版社建议了这本书。

如果读者觉得屏幕截图（如图 1-1 所示）看起来内容有些过多，这是因为我们特意在其中列出了大多数的面板，用来展示 RegexBuddy 的强大功能。实际上默认的视图会把所有的面板都很简洁地压缩成一行标签。另外，你还可以选择把一些面板拉到另外一个显示器上。

要想尝试本书中给出的一个正则表达式，只需要把它键入 RegexBuddy 窗口上端的编辑框中。RegexBuddy 会自动对你的正则表达式应用语法着色（syntax highlighting），从而使错误和括弧不匹配的情形更容易看清楚。

在键入正则表达式的同时，Create 面板会自动构造一个详细的用英语语言描述的分析。在该正则表达式的树中双击任意的描述，可以编辑正则表达式的特定部分。你还可以手动向正则表达式中添加新的部分，或者通过单击 Insert Token 按钮从菜单中选择想要的操作。例如，如果你不记得肯定型顺序环视（positive lookahead）的复杂语法，可以依靠 RegexBuddy 来帮助添加正确的字符。

在 Test 面板上可以键入或者粘贴一些示例文本。当 Highlight 按钮被激活的时候，RegexBuddy 会自动高亮显示与正则表达式匹配的文本。

最经常会用到的一些按钮如下。

#### List All

显示包含所有匹配的一个列表。

#### Replace

顶端的 Replace 按钮会显示一个新的窗口，使你可以输入替代文本。在 Test 框中的 Replace 按钮则能够查看在完成替换之后的目标文本。

#### Split（Test 面板上的按钮，而不是顶端的按钮）

把你给的正则表达式当做是一个分隔符号，使用这个正则表达式，根据在目标文本中找到的匹配把目标文本分隔成多个部分。

单击任意一个按钮，并选择 Update Automatically（自动更新），就可以让 RegexBuddy 在编辑正则表达式或者目标文本的时候，做到结果的动态同步显示。

要想看到正则表达式到底会（或者不会）如何执行，在 Test 面板上单击一个高亮显示的匹配或者是正则表达式没有能够产生匹配的地方，然后单击 Debug 按钮。RegexBuddy 会转到 Debug 面板上，逐步展示整个匹配的过程。单击调试器输出的任意地方就可以查看哪些正则表达式标记匹配了你所单击的文本。单击正则表达式则可以在调试器中高亮显示正则表达式的该部分。

在 Use 面板上，选择你最喜欢的编程语言。然后，选择一个功能就可以立即生成实现该正则表达式的源代码。RegexBuddy 的源代码模板可以使用内置的模板编辑

器进行完全编辑。你还可以添加新的特性，甚至是新的语言，或者是修改已经提供的模板。

如果想在一个更大的数据集上测试你的正则表达式，转到 GREP 面板上就可以在任意数量的文件和文件夹上进行查找（以及替换）。

当你在维护的源代码中发现一个正则表达式的时候，可以把它复制到剪贴板上，包括其中用于分隔的引号和斜杠。在 RegexBuddy 中，单击顶端的 Paste 按钮，并选择你的编程语言的字符串类型。该正则表达式就会以简单正则表达式的形式出现在 RegexBuddy 中，对于字符串字面量（string literals）来说，那些必需的额外的引号和转义符号都被去掉了。再使用顶部的 Copy 按钮就可以创建一个按照你所期望语法的字符串，从而可以把它重新粘贴回你的源代码中。

随着经验的增长，用户就可以在 Library 面板上构建起一个很方便使用的正则表达式库。当保存一个正则表达式的时候，别忘了添加一个详细的描述与一个测试对象。即使是对专家来说，正则表达式也可能会是难以捉摸的。

如果你确实无法搞明白一个正则表达式是怎么回事，可以单击 Forum 面板，然后单击 Login 按钮。如果你已经购买了 RegexBuddy，这里就会出现一个登录屏幕。单击 OK，就会被立即连接到 RegexBuddy 用户论坛中。Steven 和 Jan 经常会出现在这个论坛回答用户的问题。

RegexBuddy 可以运行在 Windows 98/ME/2000/XP/Vista 上。对于 Linux 和 Apple 爱好者，RegexBuddy 同样可以运行在 VMware、Parallels、CrossOver Office 之上，另外它在 WINE 上也可以运行，但是可能会存在一些问题。你可以从 <http://www.regexbuddy.com/RegexBuddyCookbook.exe> 下载到 RegexBuddy 的一个免费评估的版本。除了用户论坛之外，该试用版包含了完整的功能，可以实际使用 7 天。

## RegexPal

RegexPal（如图 1-2 所示）是由本书的作者之一 Steven Levithan 所创建的在线正则表达式测试工具。你所需要的仅仅是一个版本较新的网页浏览器。RegexPal 全部是由 JavaScript 编写的。因此，它只支持 JavaScript 正则流派，这与你用来访问它的网页浏览器中所实现的正则表达式一样。

如果想尝试一下本书中给出的一个正则表达式，只需浏览 <http://www.regexpal.com>。在标记了 Enter regex here 的输入框中键入正则表达式。RegexPal 会自动对你的正则表达式应用语法着色，从而可以即时显示在这个正则表达式中是否存在任何语法错误。RegexPal 很清楚在处理 JavaScript 正则表达式的时候，哪些问题可能会毁掉你一天的工作。如果某个特定的语法在有些浏览器中不能正常工作的话，那么 RegexPal 会把它突出显示成一个错误。

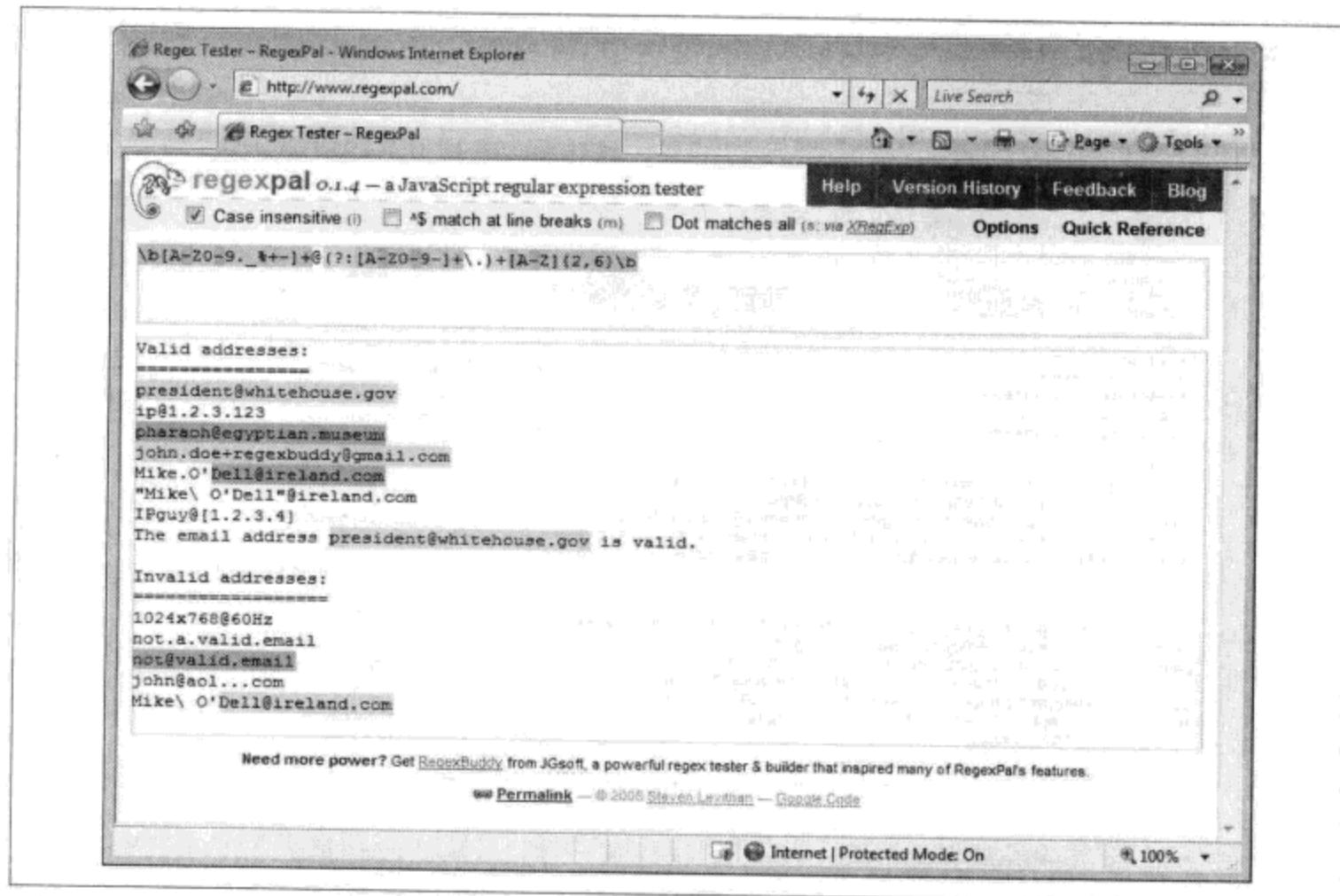


图 1-2 RegexPal

现在，你可以在标记为 Enter test data here. 的输入框中键入或者粘贴一些示例文本。RegexPal 会自动突出显示与你的正则表达式匹配的文本。

在这个工具中并不需要单击任何按钮，因此 RegexPal 是最为方便的在线正则表达式测试工具之一。

## 其他的正则表达式在线测试工具

创建一个简单的正则表达式在线测试工具并不困难。如果你拥有一些基本的 web 开发技巧，那么只需要第 3 章中的信息就足以构建自己的正则表达式工具了。成百上千的人已经这样做过：其中有些人还添加了一些额外的功能，我们会对这些工具简单加以介绍。

### regex.larsolavtorvik.com

Lars Olav Torvik 在 <http://regex.larsolavtorvik.com>（参见图 1-3）建立了一个非常优秀的小巧的在线正则表达式测试工具。

首先，用户可以单击在页面上端的流派名称来选择想要使用的正则表达式流派。Lars 提供了 PHP PCRE、PHP POSIX 和 JavaScript 的支持。PHP PCRE，也就是本书中介绍的 PCRE 正则流派，它是在 PHP 的 preg 函数中所使用的。POSIX 在 PHP 的 ereg 函数中所使用的是一个较老的、功能有限的正则表达式流派，在本书中并没有介绍这种类型。如果用户选择 JavaScript，那么使用的就是你的浏览器中所实现的 JavaScript 正则表达式。



图 1-3 regex.larsolavtorvik.com

在 Pattern 域中输入正则表达式，并在 Subject 域中输入目标文本。稍等片刻，在 Matches 域中就会显示含有着色之后的正则表达式匹配的目标文本。Code 域中会显示一行源代码来把你的正则表达式应用到目标文本之上。把这段代码复制并粘贴到你的代码编辑器中会节省大量时间，因为不必再费力气手动把正则表达式转换成一个字符串常量。该代码返回的任意字符串或者数组会显示在 Result 域中。因为 Lars 使用了 Ajax 技术来构建他的网站，所以只需片刻就会为所有的流派更新其结果。如果要使用该工具，你的电脑就必须保持在线状态，因为 PHP 是在服务器端进行处理的，而不是在浏览器中。

第二列给出了正则表达式命令和选项的一个列表。这些要依赖于你所选择的正则表达式流派。正则表达式命令通常包括匹配、替换和拆分操作。正则表达式选项包含常见的选项，比如不区分大小写，以及与实现有关的一些选项。这些命令和选项会在第 3 章中讲解。

## Nregex

<http://www.nregex.com>（如图 1-4 所示）是由 David Seruyange 开发的一个很简明的在线

正则表达式测试工具。虽然这个网站没有明确指出它实现的是哪个流派，在本书写作之时它实现的是.NET 1.x 版本。

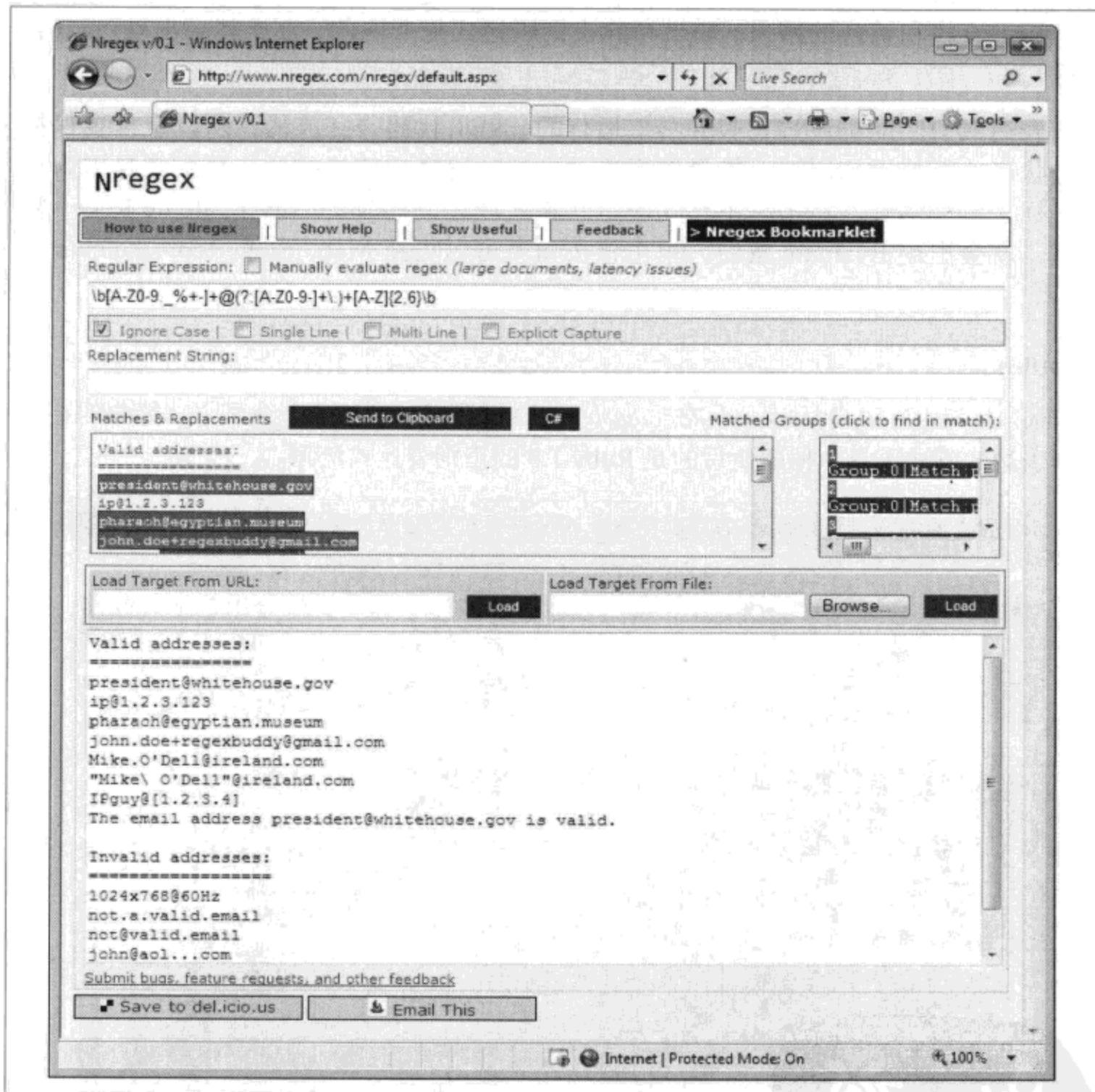


图 1-4 Nregex

这个网页的布局看起来让人费解。在 Regular Expression 标签下面的域中输入正则表达式，并且使用其下的复选框来设定正则表达式选项。然后在底部的大输入框中键入目标文本，替换掉默认的 If I just had \$5.00 then "she" wouldn't be so @#\$! mad.. 如果你的目标是一个网页的话，那么可以把它的 URL 键入 Load Target From URL 域中，然后单击该输入域之下的 Load 按钮。如果目标是在硬盘上的一个文件，那么单击 Browse 按

钮，找到想要使用的文件，然后单击在该输入域之下的 Load 按钮。

你的目标文本会重复出现在网页中心的 Matches & Replacements 域中，其中正则表达式的匹配会被突出显示。如果在 Replacement String 域中敲入一些内容，那么这里就会显示查找和替换的结果。如果你的正则表达式是非法的，那么就会出现 ... (省略号)。

正则表达式的匹配是通过在服务器上运行的 .NET 代码来完成的，所以也需要保持在线状态才能使用该网站。如果发现自动更新的速度比较慢的话，那可能是因为你的目标文本非常长，单击一下正则表达式输入域之上的 Manually Evaluate Regex 复选框，就会出现 Evaluate 按钮。单击该按钮就可以更新 Matches & Replacements 的显示。

## Rubular

Michael Lovitt 在 <http://www.rubular.com> (如图 1-5 所示) 建立了一个最小功能集的在线正则表达式测试工具，他所使用的是 Ruby 1.8 的正则表达式流派。

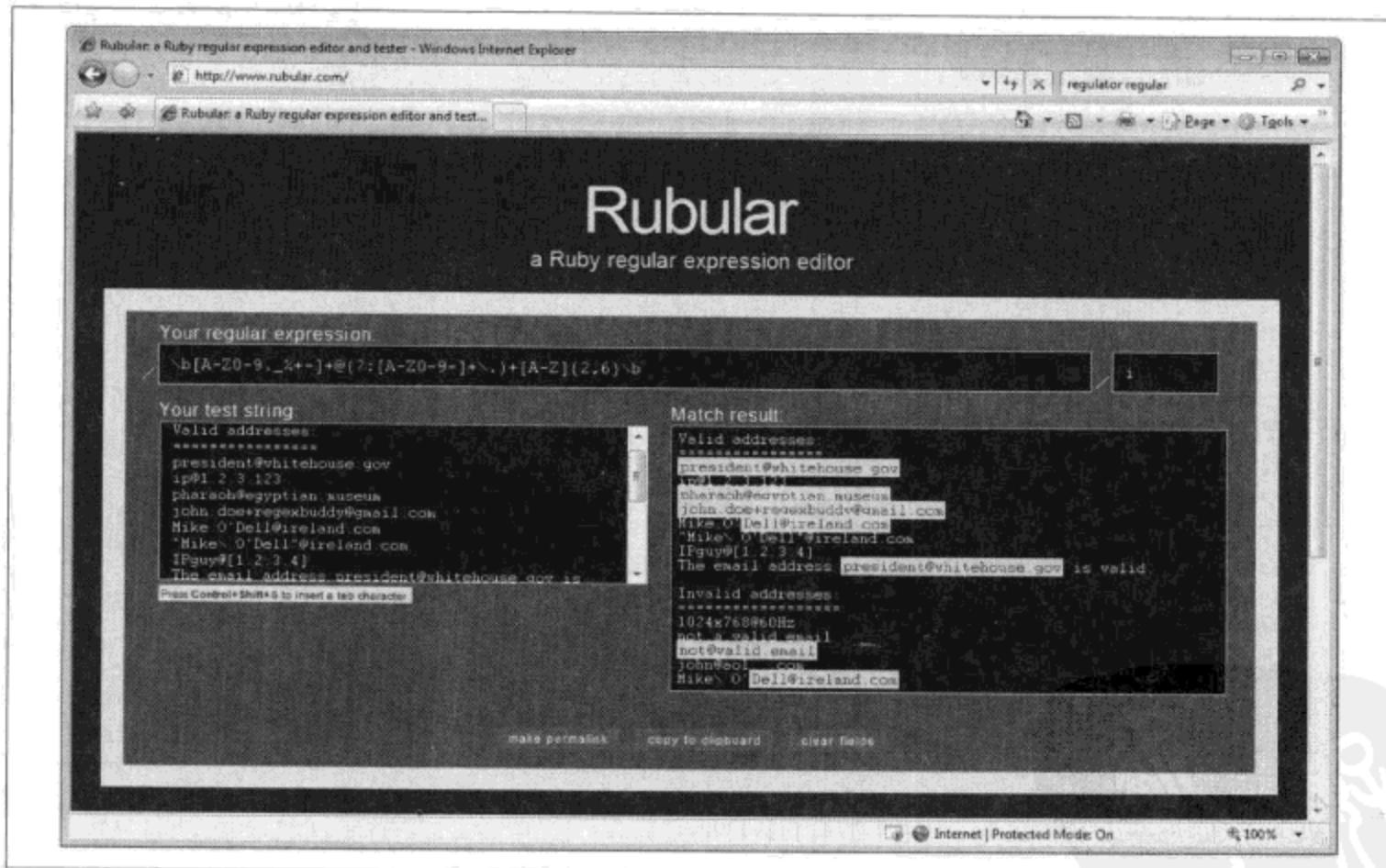


图 1-5 Rubular

在 Your regular expression 下面两个正斜杠之间的文本框中输入正则表达式。你可以通过在第二个斜杠之后的小输入框中键入一个 i 来开启大小写敏感选项。类似的，如果需要的话，还可以在同一个输入框中键入一个 m 来开启“点号匹配换行符”的选项。而

键入 im 则会同时开启这两个选项。如果你是刚刚接触 Ruby 的话，可能会觉得这些约定看起来有点儿不够用户友好，但是它们与用来在 Ruby 源代码中说明一个正则表达式时所使用的/regex/im 语法是一致的。

在 Your test string 文本框中输入或者粘贴你的目标文本。在右边会出现一个新的 Match results 文本框，用来显示所有正则表达式匹配被突出显示之后的目标文本。

### myregexp.com

Sergey Evdokimov 为 Java 开发人员创建了多个正则表达式测试工具。其中在 <http://www.myregexp.com> 网站主页（如图 1-6 所示）提供了一个在线的正则表达式测试工具。它是一个在浏览器中运行的 Java applet。在你的计算机上需要安装 Java 4（或者更新版本的）运行环境。这个 applet 使用 java.util.regex 包来运行你的正则表达式，这个包是在 Java 4 中新引入的。在本书中，“Java” 正则流派指的就是这个包。

在 Regular Expression 输入框中输入正则表达式。使用 Flags 菜单来设置你想要的正则选项。其中的三个选项也可以直接使用复选框。

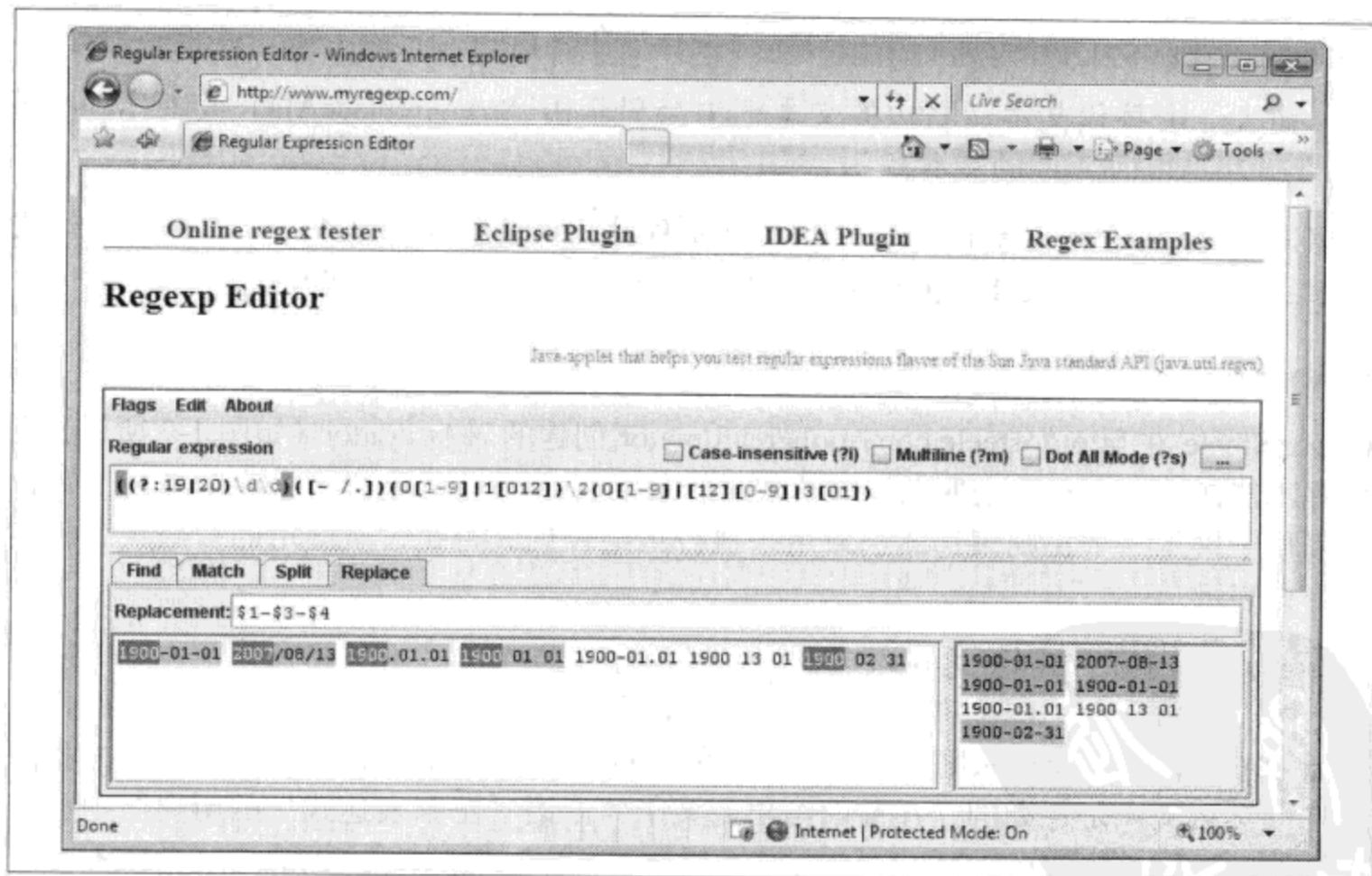


图 1-6 myregexp.com

如果想要测试一个已经在 Java 代码中使用的正则表达式的话，可以把整个字符串都复制到剪贴板上。在 myregexp.com 测试工具中，单击 Edit 菜单，然后选择 Paste Regex from

Java String。完成了正则表达式的编辑之后，在同一个菜单中选择 Copy Regex for Java Source。Edit 菜单对于 JavaScript 和 XML 也有类似的命令。

在正则表达式下面，有 4 个标签可以用来运行 4 种不同的测试。

#### Find

在示例文本中突出显示所有的正则表达式匹配。这些匹配是通过 Java 中的 Matcher.find()方法来找到的。

#### Match

检查这个正则表达式是否能够与示例文本完全匹配。如果是这样的话，那么整个文本会被全部突出显示。这是通过 String.matches()和 Matcher.matches()两个方法来实现的。

#### Split

当与你的正则表达式和示例文本一起使用时，右边的第二个文本框会显示由 String.split()或者 Pattern.split()所返回的字符串数组。

#### Replace

输入一个替代文本，右边的文本框中会显示由 String.replaceAll() 或者 Matcher.replaceAll()所返回的文本。

你可以通过在 <http://www.myregexp.com> 网页顶端的链接找到 Sergey 的其他正则表达式测试工具。其中一个是 Eclipse 的插件（plug-in），另外一个是 IntelliJ IDEA 的插件。

### reAnimator

Oliver Steele 在 <http://osteele.com/tools/reanimator> 创建的 reAnimator（如图 1-7 所示）当然并不能使一个已经死去的正则表达式重新复活<sup>1</sup>。事实上，它是一个有趣的小工具，可以用于展示一个正则表达式引擎用来执行正则表达式查找时所使用的有限状态机的图形化表示。

reAnimator 支持的正则表达式语法是非常有限的。它与本书中介绍的所有流派都是兼容的。在 reAnimator 中能够用来进行动画显示的任意正则表达式都能够在本书中的任意流派中使用，但是反过来则一定是不成立的。这是因为 reAnimator 的正则表达式指的是在数学意义上是正则的。在前面给出的关于“术语‘正则表达式’的历史”中已经对此给出了简单的解释。

首先从页面顶端的 Pattern 文本框开始，单击 Edit 按钮。把正则表达式输入 Pattern 域中，并单击 Set。然后在 Input 域中缓慢地键入目标文本。

<sup>1</sup>译者注：reanimator 这个词的本意为“使…复活”。

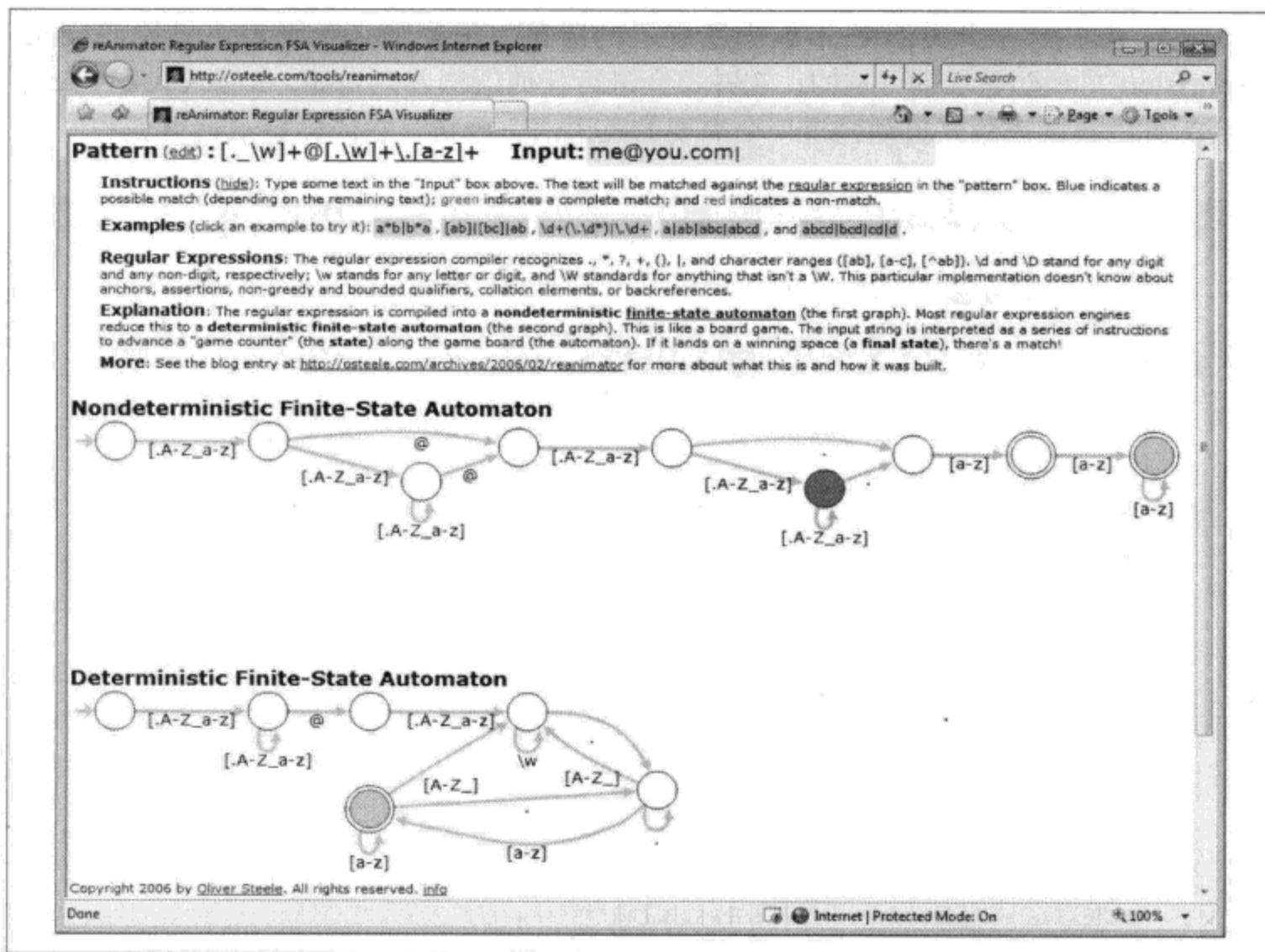


图 1-7 reAnimator

在键入每一个字符的同时，彩色小球会在状态机中移动，来说明你的输入在目前位置已经到达的最后节点。蓝色小球说明状态机接受该输入，但是还需要更多的输入才能到达完全匹配。绿色小球说明该输入匹配了整个模式。没有小球则意味着该状态机不能匹配这个输入。只有当正则表达式匹配整个输入字符串的时候，reAnimator 才会显示一个匹配，就好像是你把它放到了 `<^>` 和 `<$>` 定位符（anchor）之间一样。这实际上是表达式在数学意义上正则的另外一个属性。

## 更多的桌面正则表达式测试工具

### Expresso

Expresso（不要把它和富含咖啡因的浓咖啡 espresso 混为一谈）是用来创建和测试正则表达式的一个 .NET 应用程序。你可以从 <http://www.ultrapico.com/Expresso.htm> 下载它。在你的计算机上必须安装有 .NET framework 2.0 或者更新版本。

从网站下载的是一个 60 天的免费试用版。在试用期过后，必须注册该软件，否则 Expresso 就（大体上）无法使用了。注册码可以通过 E-mail 获得。

Expresso 会显示一个类似图 1-8 中所展示的屏幕。用来输入正则表达式的 Regular Expression 文本框是永远可见的。它并不会进行任何语法着色功能。Regex Analyzer 框则会为你的正则表达式自动构造一个简要的英语语言分析。它同样是永远可见的。

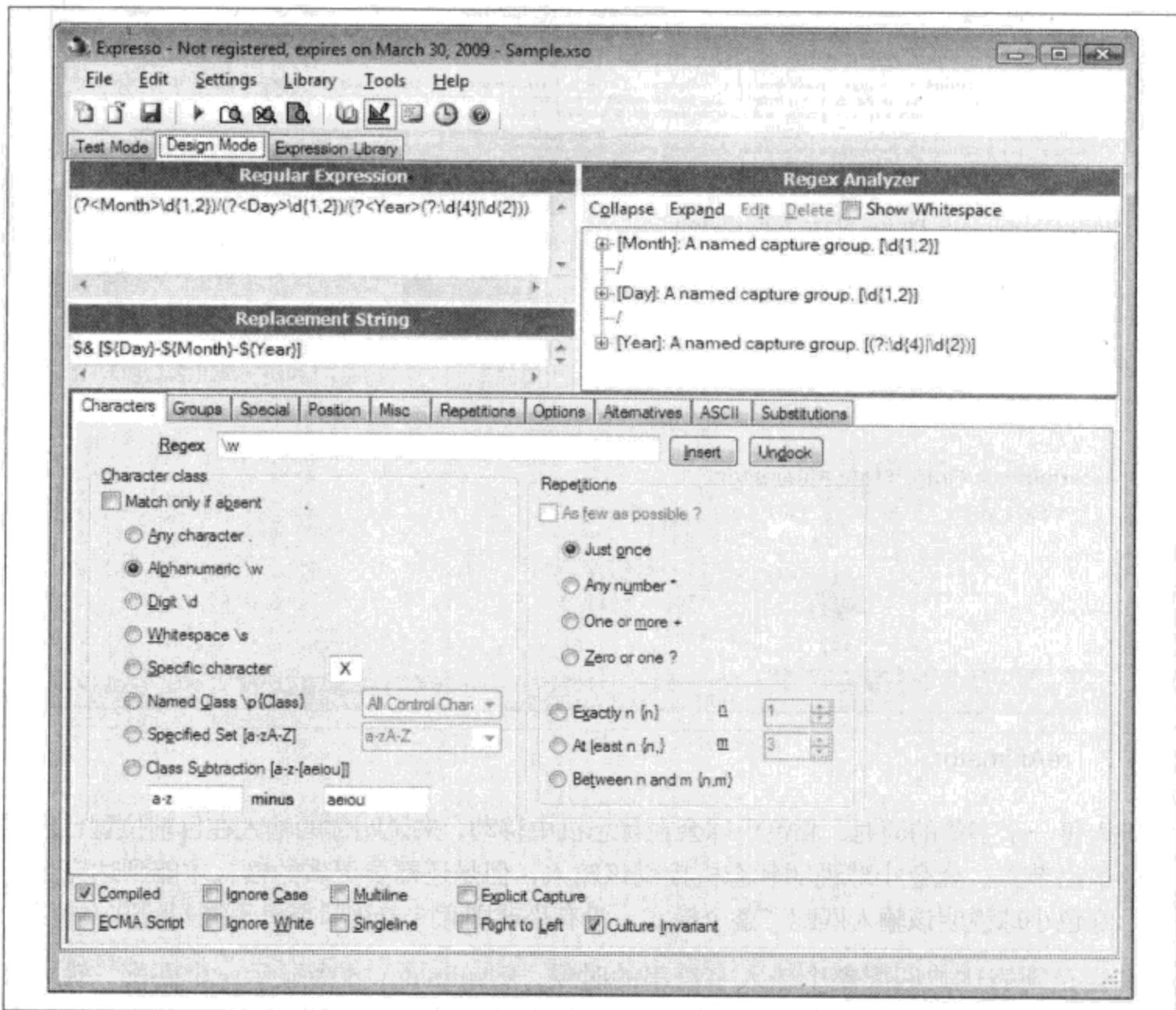


图 1-8 Expresso

在设计模式（Design Mode）下，可以在屏幕的底部设置诸如“Ignore Case（忽略大小写）”这样的匹配选项。屏幕空间的大部分被一行标签所占据，可以在此选择想要插入的正则表达式记号。如果你拥有两个显示器，或者一个大的显示器，可以单击 Undock 按钮来使这行标签都悬浮起来。接着仍然可以在其他模式（Test Mode）下构建正则表达式。

在测试模式（Test Mode）下，可以在屏幕左下角输入或者粘贴示例代码。然后，单击 Run Match 按钮，就可以在 Search Results 文本框中得到所有匹配的一个列表。这里并不会对示例文本应用任何的突出显示。在结果中单击一个匹配就可以选择在示例文本中对应的匹配。

Expression Library 会给出样例正则表达式的一个列表，以及最近使用正则表达式的列表。每次按下 Run Match 的时候，正则表达式都会被添加到这个列表中，可以通过在主菜单工具条上的 Library 菜单来编辑这个表达式库。

## The Regulator

可以从 <http://sourceforge.net/projects/regulator> 下载的 The Regulator 是另外一个用来创建和测试正则表达式的 .NET 应用。它的最新版本要求 .NET 2.0 或者更新版本。你还可以下载到用于 .NET 1.x 的较早版本。The Regulator 是开源软件，不需要付钱或者注册。

The Regulator 会在一个屏幕中（如图 1-9 所示）完成所有的工作。New Document 标签是用来输入正则表达式的地方。语法着色会被自动应用，但是在正则表达式中的语法错误却不会被突出显示。单击鼠标右键可以从一个菜单中选择想要添加的正则表达式记号，可以通过主工具条之上的按钮来设置正则表达式选项。这些图表看起来会有些费解，可以让鼠标稍微停留，等一下工具提示的出现，就能看到可以使用每个按钮来设置哪些选项了。

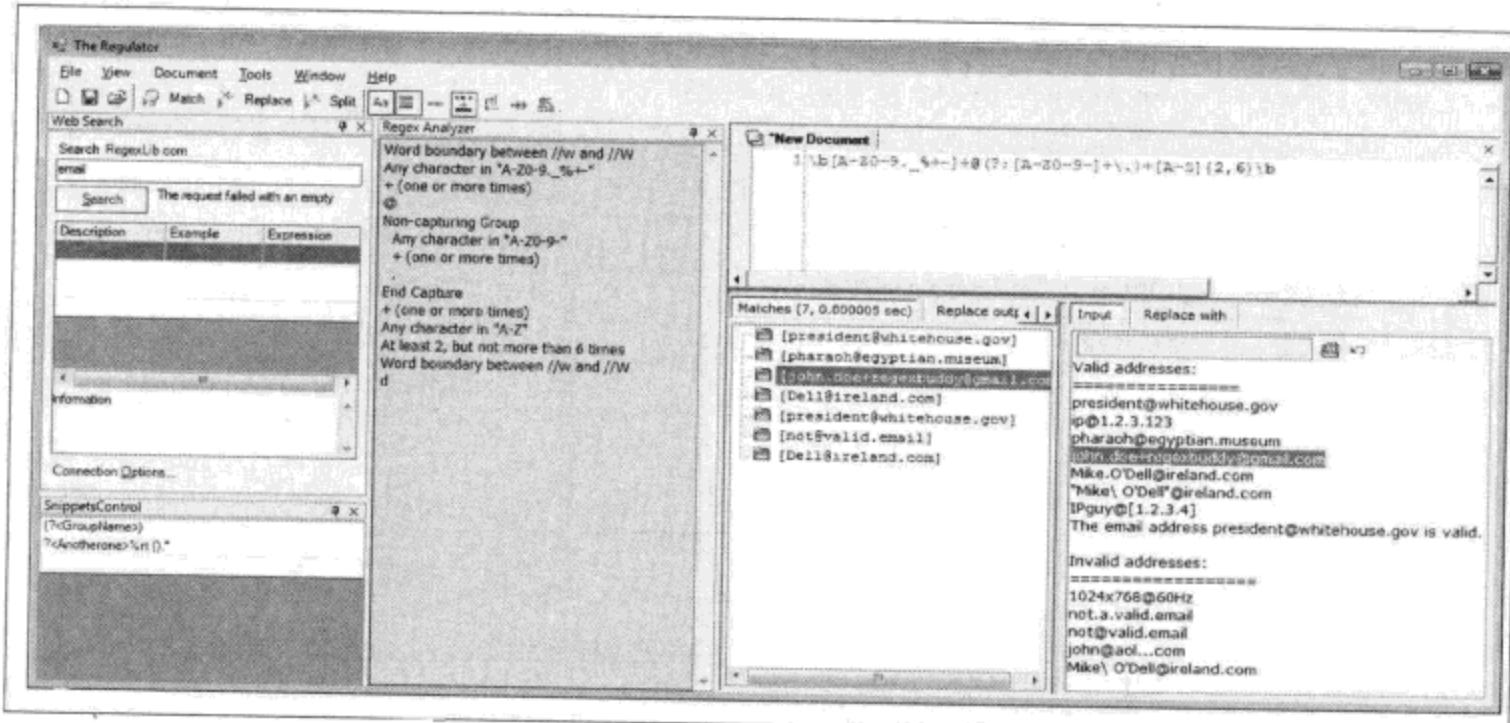


图 1-9 The Regulator

在正则表达式区域下面的右边，单击 Input 按钮就会显示可以用来粘贴示例文本的区域。如果想要进行查找和替换操作的话，你可以单击 Replace with 按钮来键入替代文本。在正则表达式下面偏左的地方，你会看到该正则表达式操作的结果。结果不会进行自动更新；你必须单击在工具条中的 Match、Replace 或 Split 按钮才能更新结果。另外也不会对输入应用任何突出显示。需要单击在结果中的一个匹配来在目标文本中选中它。

Regex Analyzer 面板展示的是对你的正则表达式进行的一个简单的英语语言分析，但是它既不是自动的，也不支持交互。要想更新这个分析结果，需要在 View 菜单中选择 Regex Analyzer，即使它已经是可见的也需要这样做。如果只是单击分析，那么只会移动文本指针。

## grep

*grep* 这个名字是从 *g/re/p* 这个命令推衍而来的，这个命令出现在最早支持正则表达式的一个应用，也就是 UNIX 下的文本编辑工具 *ed* 中，可以用来执行正则表达式的查找。该命令非常流行，以至于所有的 UNIX 系统中现在都包含一个专门的 *grep* 工具使用正则表达式来在文件中进行查找。如果你在使用 UNIX、Linux 或者 OS X，那么在一个终端窗口中键入 *man grep* 命令就可以对此有更多的了解。

下面的 3 个工具是用来完成 *grep* 功能的 Windows 应用程序，它们还添加了额外的功能。

## PowerGREP

由本书作者之一 Jan Goyvaerts 所开发的 PowerGREP，可能是在 Microsoft Windows 平台上可用的功能最为丰富的 *grep* 工具（如图 1-10 所示）。PowerGREP 使用一种定制的正则表达式流派，它组合了在本书中介绍的流派中最好的几种。这种流派在 RegexBuddy 中被标记为“JGsoft”。

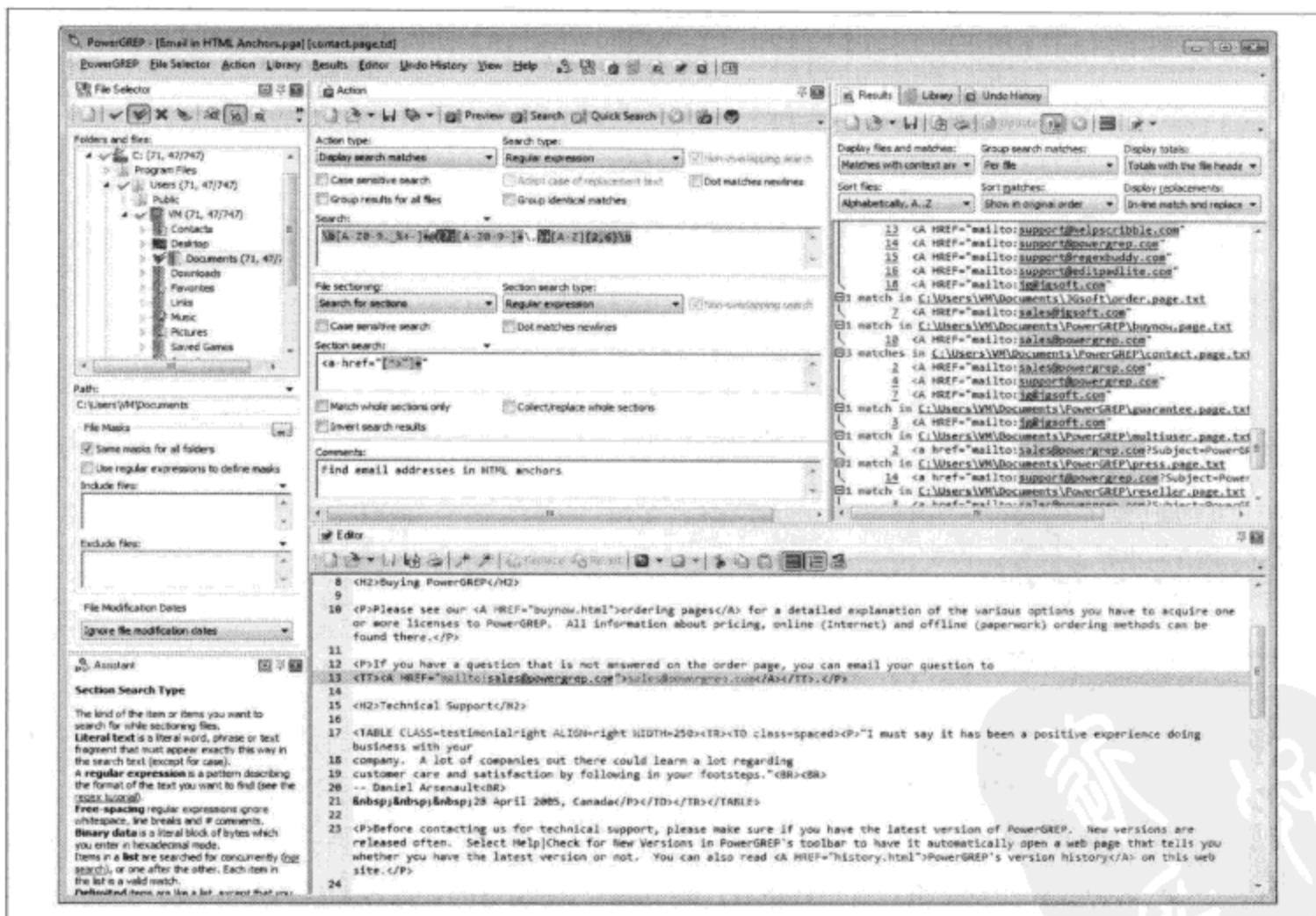


图 1-10 PowerGREP

要运行一个快速的正则表达式查找，可以简单地在 Action 菜单中选择 Clear，并在 Action 面板上的 Search 框中输入你的正则表达式。在 File Selector 面板上单击一个文件夹，然后在 File Selector 菜单中选择 Include File or Folder 或 Include Folder and Subfolders。接

着在 Action 菜单中选择 Execute 就可以进行查找。

要运行一个查找和替换的操作，在清除了上述动作之后，在 Action 面板左上角处的 action type 下拉列表中选择 search-and-replace。在那里输入你的替代文本。所有其余的步骤都与查找功能完全相同。

PowerGREP 具有独特的能力，可以在同一时刻使用 3 个正则表达式列表，在每个列表中可以包含任意数目的正则表达式。虽然前面两个段落给出了足够多的信息，使你可以像其他任何 grep 工具一样运行一些简单的搜索，但是要释放 PowerGREP 的全部潜力则需要多花点儿时间通读一下该工具的详细文档。

PowerGREP 可以在 Windows 98/ME/XP/Vista 上运行。你可以从 <http://www.powergrep.com/> PowerGREPCookbook.exe 下载一个免费评估版本。除了保存结果和库之外，试用版拥有可以实际使用 15 天的全部功能。尽管试用版不能保存在 Results 面板上显示的结果，但是它也能像完整版本一样，会实际执行查找和替换动作引入的对所有文件的修改。

## Windows Grep

Windows Grep (<http://www.wingrep.com>) 是在 Windows 平台上最古老的 grep 工具之一。它的年代可以从它的用户界面中看出一点儿端倪（如图 1-11 所示），但它的功能还是相

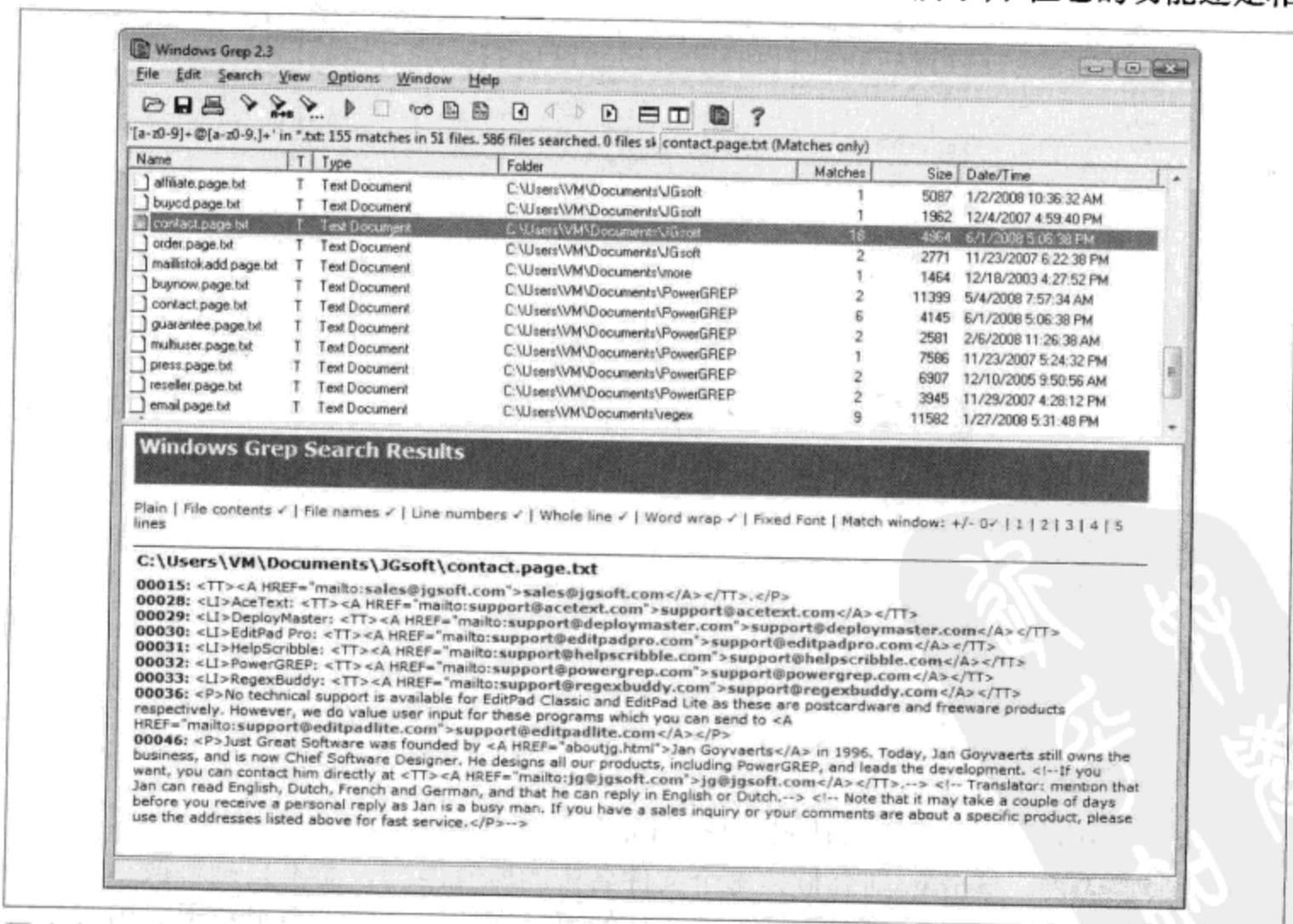


图 1-11 Windows Grep

对完整的。它支持一种功能有限的正则表达式流派，被称作 POSIX ERE。对于所支持的那部分功能，它所使用的语法与本书中介绍的流派是一样的。Windows Grep 是一个共享软件，这意味着可以免费下载它，但是如果期望长时间使用的话，你就需要付费。要想执行搜索，在 Search 菜单中选择 Search。接下来出现的屏幕将会根据你在 Options 菜单中所选择的是 Beginner Mode 或是 Expert Mode 而有所不同。初学者会看到一个逐步向导，而专家则会看到一个带标签的对话框。

设置好搜索之后，Windows Grep 会立即执行它，向你展示找到匹配的一个文件列表。单击一个文件就可以在底部面板中看到其中的匹配，而双击一个文件会打开该文件。在 View 菜单中选择 All Matches 会在底部面板中显示所有内容。

如果想要运行查找和替换，那么在 Search 菜单中选择 Replace 即可。

## RegexRenamer

RegexRenamer（如图 1-12 所示）实际上并不是一个 grep 工具。它不会在文件的内容中进行查找，而是查找和替换文件的名称。你可以从 <http://regexrenamer.sourceforge.net> 下载该工具。RegexRenamer 要求安装 Microsoft .NET Framework 的 2.0 或者更新版本。

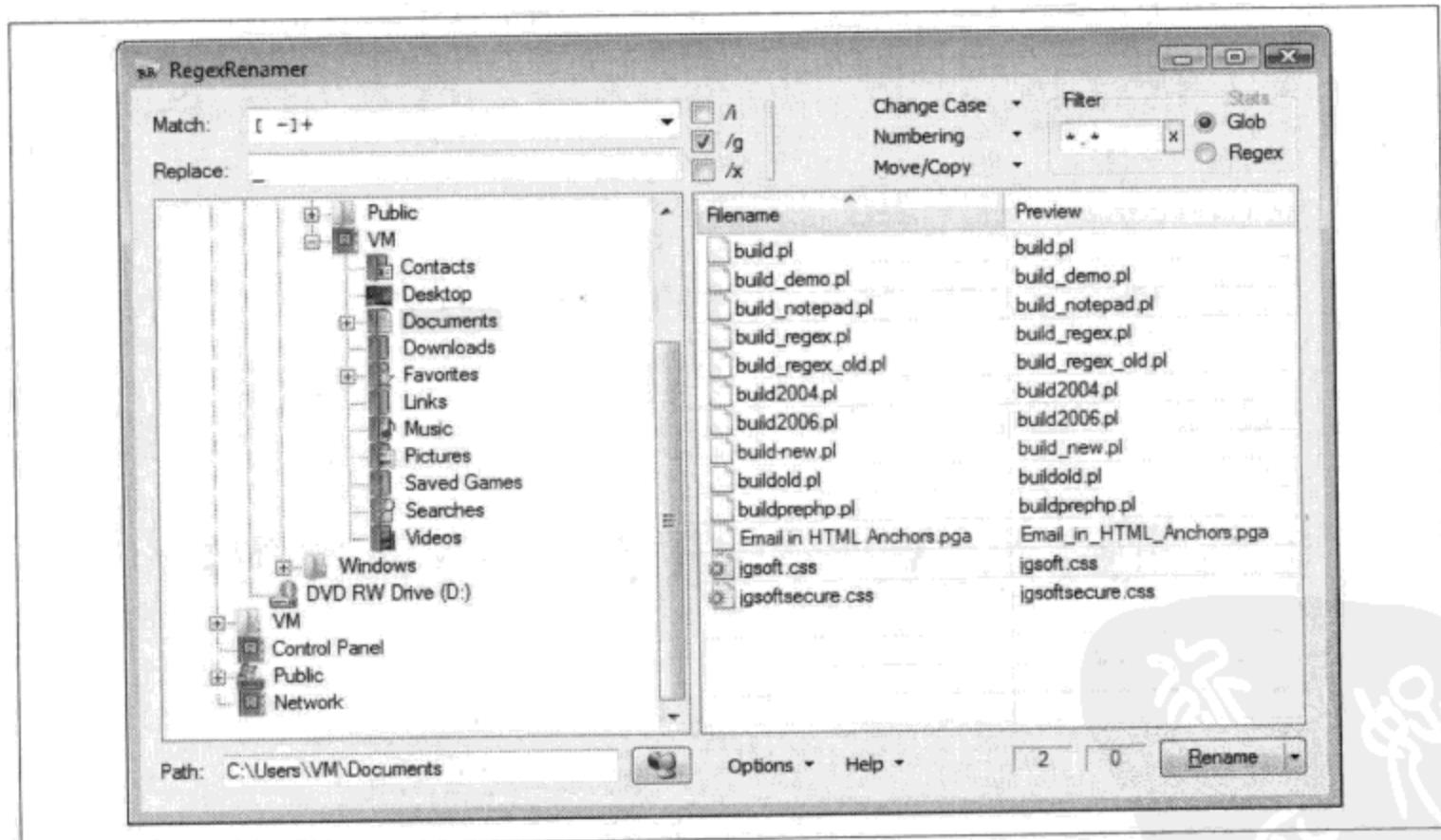


图 1-12 RegexRenamer

在 Match 框中键入你的正则表达式，并在 Replace 框中键入替代文本。单击/i 可以打开大小写敏感选项，单击/g 则会替换每个文件名中的所有匹配，而不只是替换第一个匹配。选择/x 会打开宽松排列（free-spacing）的语法选项，这并不会很有用，因为只有

一行空间可以输入你的正则表达式。

使用左边的树可以选择包含你想要重命名的文件的文件夹。在右上角，可以设置一个文件掩码或者正则表达式过滤器。这可以将查找和替换的正则表达式的应用限制到所选择的文件列表。与其试图只使用一个正则表达式来完成所有工作，使用一个正则表达式进行过滤，而使用另外一个正则表达式进行替换是更为方便的做法。

## 常见的文本编辑器

大多数现在的文本编辑器都至少拥有对正则表达式的基本支持。在查找或者查找和替换的面板上，你通常会发现一个复选框可以打开正则表达式模式。有些编辑器，比如 EditPad Pro，也会使用正则表达式来完成各种文本处理的功能，例如语法着色或是类和函数列表等。每个编辑器所带的文档会详细讲解所有这些功能。下面列出了一些常见的提供正则表达式支持的编辑器：

- Boxer Text Editor (PCRE)
- Dreamweaver (JavaScript)
- EditPad Pro (组合了本书中介绍的流派中最好部分的一种定制流派；在 RegexBuddy 中被称作是“JGsoft”)
- Multi-Edit (PCRE, 需要选择“Perl”选项)
- NoteTab (PCRE)
- UltraEdit (PCRE)
- TextMate (Ruby 1.9 [Oniguruma])

## 第 2 章

# 正则表达式的基本技巧

本章要讲解的问题并不是老板或客户会要求你解决的那一类现实世界中的问题。相反，它们是在你创建和编辑正则表达式来解决现实世界问题的过程中会遇到的技术性问题。例如，第一个实例会解释如何使用一个正则表达式来匹配字面文本 (literal text)。这个问题本身并不是很重要，因为当你只是要查找字面文本的时候，并不会需要使用正则表达式。但是，当创建正则表达式的时候，你可能会需要照字面来匹配某些文本，那么你就需要知道哪些字符需要进行转义。实例 2.1 会告诉你这该如何做到。

本章前一部分的实例会讲解一些非常基本的正则表达式技术。如果以前使用过正则表达式，那么你大概可以略读或者甚至是跳过它们。除非已经从头到尾认真读过 Jeffrey E. F. Friedl 所著的《Mastering Regular Expression (精通正则表达式)》一书，在本章后一部分所给出的实例一定会教给你一些新的东西。

本章实例的安排方式是每个实例会讲解正则表达式语法的一个方面。所有这些实例加在一起就形成了正则表达式的一个全面的指南。读者可以先从头到尾读完本章以做到深入领会正则表达式。或者读者也可以直接跳到第 4~第 8 章中要讲解的现实世界中的正则表达式，而当在那些章节中是你不很熟悉的语法时，再按照给出的引用跳回来阅读本章中的相应内容。

本章的指南只涉及正则表达式，因而会完全忽略与编程有关的任何考虑事项。下一章会讲解和代码有关的内容。可以先简单看一下第 3 章中的“编程语言和正则流派”一节，来了解一下你使用的编程语言所用的是哪种正则表达式流派。本章会讲到的所有流派本身都已经在上一章中的“本书涉及的正则流派”一节中介绍过。

## 2.1 匹配字面文本

### 问题描述

创建一个正则表达式来严格匹配下面这个复杂的句子：

```
The punctuation characters in the ASCII table  
are: !#$%&'()*+,-./:;<=>?@[\]^_`{|}~.
```

### 解决方案

```
The●punctuation●characters●in●the●ASCII●table●are:●←  
!"#$%&'()\*\+,-./:;<=>?\@[\[\]]\^_`\{\}\~  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### 讨论

不包含如下这些字符的任意正则表达式都可以简单地匹配其自身：`$()*+.?[^{}]`。如果要在正在编辑的文档中查找是否包含 `Mary had a little lamb`，那么你只需要简单地查找 `<Mary●had●a●little●lamb>` 即可。在这个例子中，你是否打开了文本编辑器中的“正则表达式”复选框并没有关系。

正则表达式之所以拥有巨大的魔力，就是因为有了这 12 个标点字符才产生的，它们被称作是元字符（metacharacters）。如果想要在正则表达式中照字面匹配它们，那么就需要在它们前面用一个反斜杠来进行转义。因此，下面的正则表达式

```
\$\\(\\)\*\+\\.\\?\\[\\\\\\^\\{\\}
```

会匹配如下的文本：

```
$()*+.?[\^{}]
```

特别应该注意的是在这个列表中并不包含右方括号]、连字号-和右花括号}。前两个符号只有在它们位于一个没有转义的[之后才成为元字符，而}只有在一个没有转义的{之后才是元字符。在任何时候都没有必要对}进行转义。在[和]之间的字符类中出现的元字符的转义规则会在实例 2.3 中加以解释。

对任意其他非字母数字的字符进行转义不会改变你的正则表达式的规则——至少在本书中讲到的所有流派中都不会这样。而对一个字母数字字符进行转义则会给它一个特殊含义，或者出现一个语法错误。

新接触正则表达式的人通常会对看到的每个标点字符进行转义。不要让任何人看出来你是个新手！要明智地选择何时需要转义。一大堆没必要的反斜杠会使正则表达式变得难以阅读，特别是当在源代码中把正则表达式作为字面文本串来引用的时候，所有这些反斜杠又必须加倍出现。

# 变体

## 块转义

```
The punctuation characters in the ASCII table are: •↓\Q!"#$%&'()*+,.-./:;<=>?@\[\]^_`{|}~\E
```

正则选项: 无

正则流派: Java 6、PCRE、Perl

Java 6、PCRE 和 Perl 支持使用正则记号 `\Q` 和 `\E`。`\Q` 会抑制所有元字符的含义，直到出现`\E`为止。如果漏掉了`\E`，那么在`\Q`之后直到正则表达式结束之前的所有字符都会被当作字面文本来对待。

使用`\Q...\\E`的唯一好处是它读起来会比`\.\.`要更容易一些。



### 警告

虽然 Java 4 和 Java 5 都支持这个特性，但是我们却不推荐读者使用它。在实现中会产生 bug，造成含有`\Q...\\E`的正则表达式匹配到与你的期望不同的内容，并且与 Java 6、PCRE 和 Perl 所匹配的内容不一样。这些 bug 在 Java 6 中得到了修正，使之与 PCRE 和 Perl 中的行为保持一致。

## 不区分大小写的匹配

```
ascii
```

正则选项: 不区分大小写

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```
(?i)ascii
```

正则选项: 无

正则流派: .NET、Java、PCRE、Perl、Python、Ruby

默认情况下，正则表达式是区分大小写的。`\regex` 会匹配 `regex`，但是不能匹配 `Regex`、`REGEX` 或是 `ReGeX`。如果也想匹配这些形式的话，那么就需要打开不区分大小写选项。

在大多数应用程序中，实现这样的功能往往只需要简单地选中或取消一个复选框。在下一章中要讲解的所有编程语言中都拥有一个标记或属性，可以用它来把你的正则表达式设置为不区分大小写的。下一章中的实例 3.4 会讲解如何在源代码中应用本书在每个正则表达式解答之下列出的正则选项。

如果无法在正则表达式之外打开不区分大小写选项的话，那么你还可以在正则表达式之内通过使用`\(?i)` 模式修饰符 (mode modifier) 来设置，例如`\(?i)\regex`。这个选项可以在.NET、Java、PCRE、Perl、Python 和 Ruby 流派中使用。

.NET、Java、PCRE、Perl 和 Ruby 支持局部的模式修饰符，这样它只会影响到一个正则表达式的一部分。例如，`\sensitive\(?i\)\caseless\(?-i\)\sensitive` 会匹配 `sensitive CASELESSsensitive`，但是不能匹配 `SENSITIVEcaselessSENSITIVE`。`\(?i)` 会在正则表达式的剩余部分打开

不区分大小写，而`(?-i)`会在正则表达式的剩余部分关掉这个选项。它们一起就可以像开关一样来使用。

实例 2.10 会讲解如何使用局部模式修饰符来实现分组。

## 参见

实例 2.3 和实例 5.14。

## 2.2 匹配不可打印字符

### 问题描述

匹配一个包含下列 ASCII 控制字符的字符串：响铃字符（bell）、退出符（escape）、换页符（form feed）、换行符（line feed）、回车符（carriage return）、水平制表符（horizontal tab）和垂直制表符（vertical tab）。这些字符的十六进制 ASCII 编码分别是：07、1B、0C、0A、0D、09、0B。

### 解决方案

`\a\c\f\n\r\t\v`

正则选项：无

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

`\x07\x1B\f\n\r\t\v`

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

这 7 个最常用的 ASCII 控制字符拥有专门的转义序列。它们都包含一个反斜杠与一个字母。这与在许多编程语言中用于字符串字面文本的语法是相同的。表 2-1 给出了常用的不可打印字符以及它们的表示方法。

表 2-1 不可打印字符

表 示	含 义	十六进制表示
<code>\a</code>	响铃	0x07
<code>\e</code>	退出	0x1B
<code>\f</code>	换页	0x0C
<code>\n</code>	换行（新的一行）	0x0A

续表

表示	含义	十六进制表示
\r	回车	0x0D
\t	水平制表符	0x09
\v	垂直制表符	0x0B

ECMA-262 标准不支持 \a 和 \e。因此在本书中的 JavaScript 示例中我们会使用不同的语法，虽然许多浏览器事实上也提供了对 \a 和 \e 的支持。Perl 不支持 \v，所以在 Perl 中就不得不使用另外一种语法来表示垂直制表符。

这些控制字符以及在下面的小节中会给出的替代语法，均可以在正则表达式中的字符类之内和之外同等地使用。

## 不可打印字符的表示变体

### 26 个控制字符

\cG\x1B\cL\cJ\cM\cI\cK

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Ruby 1.9

使用从 \cA 到 \cZ，可以匹配在 ASCII 表中占据了第 1~26 位的 26 个控制字符。其中的 c 必须是小写的。而在大多数流派中，c 之后的字母是不区分大小写的。我们推荐总是使用大写字母，因为在 Java 中是这样要求的。

如果习惯于在终端系统上通过按 Control 键加上一个字符来输入控制符号的话，那么这种语法对你来说会比较容易。例如，在终端上，Ctrl-H 会发送一个回车。而在正则表达式中，\cH 会匹配一个回车。

Python 和 Ruby 1.8 中的经典 Ruby 引擎并不支持这种语法。而 Ruby 1.9 中的 Oniguruma 引擎会支持这种语法。

在 ASCII 表中第 27 位的退出控制字符就超出了英语字母表的范围，因此我们会在正则表达式中使用 \x1B 来表示它。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	~	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

图 2-1 ASCII Table

## 7 比特字符集

\x07\x1B\x0C\x0A\x0D\x09\x0B

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

一个小写的 \x 后跟两个大写十六进制数字可以匹配 ASCII 字符集中的一一个字符。图 2-1 给出了在整个 ASCII 字符集中，从 \x00 到 \x7F 的每个组合所匹配的每个字符。在该表中，第一个十六进制数字从左边向下增长，而第二个数字则在第一行从左向右排列。

从 \x80 到 \xFF 会匹配哪些字符则要取决于你的正则引擎会如何解释它们，以及你的目标文本使用的编码是哪种代码页。我们推荐你不要使用从 \x80 到 \xFF。作为替代，应该使用在实例 2.7 中描述的 Unicode 代码点记号。

如果使用的是 Ruby 1.8，或者在编译 PCRE 时没有使用 UTF-8 支持，那么你就不能使用 Unicode 代码点。Ruby 1.8 和不带 UTF-8 的 PCRE 是 8 比特的正则引擎。它们对于文本编码和多字节字符是完全不支持的。在这些引擎中，\xAA 只会简单地匹配 0xAA，而不去管 0xAA 所表示的是哪个字符，或者 0xAA 是否属于某个多字节字符的一部分。

## 参见

实例 2.7。

## 2.3 匹配多个字符之一

### 问题描述

创建一个正则表达式来匹配 calendar 的所有常见的错误拼写形式，从而无论作者的拼写能力如何，你都能够在一个文档中找到这个单词。在每个元音位置都允许使用 a 或者 e。创建另外一个正则表达式来匹配一个单个的十六进制字符。再创建一个正则表达式来匹配不属于十六进制字符的单个字符。

### 解决方案

#### 包含错误拼写的 calendar

c[ae]l[ae]nd[ae]r

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

#### 十六进制字符

[a-fA-F0-9]

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 非十六进制字符

[^a-fA-F0-9]

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

使用方括号的表示法被称作是一个字符类（character class）。一个字符类匹配在一个可能的字符列表中的单个字符。在第一个正则表达式中的 3 个字符类可以匹配一个 a 或是一个 e。它们彼此之间是独立的。当你使用 calendar 来测试这个正则表达式的时候，第一个字符类匹配 a，第二个字符类匹配 e，而第三个字符类则会匹配 a。

在字符类之外，上面的 12 个标点字符是元字符。在一个字符类中，只有其中 4 个字符拥有特殊功能：`\`、`^`、`-`和`]`。如果你使用的是 Java 或者 .NET，那么左方括号`[`在字符类中也是一个元字符。所有其他字符都是字面量，只是把它们自身加入到了字符类中。正则表达式`<[$()*+.?{}]>`会匹配在方括号之间出现的 9 个字符中的任意一个。

反斜杠总是会对紧跟其后的字符进行转义，这与它在字符类之外的作用一样。被转义的字符可以是单个字符，也可以是一个范围的开始或结束。另外 4 个元字符只有当它们被放置到特定位置时才拥有特殊含义。如果不按照可能会包含特殊含义的方式来使用，那么也可以在一个字符类中不对其使用转义，而把它们作为字面字符来使用。例如，用`<[][^-]>`就可以，唯一例外的情形是你所使用的正好是严格遵循标准的 JavaScript 实现。但是，我们还是推荐你总是对这些元字符进行转义，因此前面的正则表达式应该总是使用`<\[\]^[-]>`的形式。在使用中总是对元字符进行转义会使你的正则表达式更容易让人理解。

字母数字字符则不能使用反斜杠来转义。如果这样做的话，要么会出现一个错误，要么会创建一个正则表达式记号（也就是在正则表达式中含有特殊含义的语法符号）。在前面的实例 2.2 中，我们讨论了一些其他正则表达式记号，其中提到了它们可以在字符类之内使用。所有这些记号都由反斜杠和一个字母组成，有时候后面还会跟一堆其他字符。因此，`<[\r\n]>`会匹配一个回车符（\r）或者换行符（\n）。

如果紧跟着左括号后面是一个脱字符（`^`）的话，那么就会对整个字符类取否。也就是说它会匹配不属于该字符类列表中的任意字符。一个否定字符类会匹配换行符号，除非把换行符也加入到否定字符类中。

连字符（`-`）被放到两个字符之间的时侯就会创建一个范围（range）。该范围所组成的字符类包含连字符之前的字符、连字符之后的字符，以及按照字母表顺序位于这两个字符之间的所有字符。要想知道一个范围内到底包含了哪些字符，需要查看 ASCII 或者 Unicode 字符表。`<[A-z]>`包含在 ASCII 表中在大写 A 到小写 z 之间的所有字符。注

意这个范围中会包含一些标点符号，因此可以使用 `<[A-Z\[\]]^_`a-z>` 来更加清晰地匹配相同的字符集合。我们推荐你所创建的范围只位于两个数字，或者两个同是大写或者小写的字母之间。反向的范围，例如`<z-a>`，是不允许的。

## 变体

### 简写

`[a-fA-F\d]`  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

由反斜杠和一个字母组成的 6 个正则表达式记号会构成简写（shorthand）字符类。你可以在字符类之内或者之外使用这些记号。`\d` 和 `\D` 都会匹配单个数字。每个小写的简写字符都拥有一个相关联的大写简写字符，其含义正好相反。因此 `\D` 会匹配不是数字的任意字符，所以同 `<[^d]>` 是等价的。

`\w` 匹配单个的单词字符（*word character*）。所谓单词字符指的是能够出现在一个单词中的字符。这包括了字母、数字和下划线。这里所选的字符集合看起来可能有些怪异，但是之所以这样选的原因是这些字符正好是在编程语言中的标识符中通常允许使用的字符。`\W` 则会匹配不属于上述字符集合的任意字符。

在 Java、JavaScript、PCRE 和 Ruby 中，`\w` 总是与`<[a-zA-Z0-9_]>` 的含义完全相同。而在 .NET 和 Perl 中，它包含来自所有其他字母表（西里尔语、泰语等）的字母和数字。在 Python 中，只有当你在创建正则表达式时传递了相关选项时，才会包含其他字母表的字符。在上述这些流派中，`\d` 遵循相同的规则。在.NET 和 Perl 中，其他字母表中的数字总是会被包含进来，而在 Python 中则只有当你传递了 UNICODE 或 U 选项才会包含它们。

`\s` 匹配任意的空白字符（*whitespace character*）。其中包括了空格、制表符和换行符。在.NET、Perl 和 JavaScript 中，`\s` 也会匹配根据 Unicode 标准被定义为白色符号的字符。需要注意的是 JavaScript 对于`\s` 使用 Unicode，而对于`\d` 和 `\w` 则使用 ASCII 标准。`\S` 会匹配`\s` 不能匹配的任意字符。

当我们还要考虑`\b` 的时候，就会遇到更多的不一致性。`\b` 不是一个简写字符类，而是一个字符边界。虽然你可以期望当`\w` 支持 Unicode 的时候`\b` 也应该支持 Unicode；而当只`\w` 支持 ASCII 的时候，`\b` 也应该是只使用 ASCII，然而事实上却不总是如此。在实例 2.6 中的“单词字符”小节中会介绍更多的细节。

### 区分大小写

`(?i) [A-F0-9]`  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

```
(?i) [^A-F0-9]  
正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

区分大小写也会影响字符类，它可以使用一个外部选项来设置（参见实例 3.4），也可以在正则表达式内采用模式修饰符来设置（参见实例 2.1）。上面给出的这两个正则表达式与最初的解答是等价的。

JavaScript 也采用类相同的规则，但是它并不支持 `<(?i)>`。要在 JavaScript 中把一个正则表达式设置为不区分大小写，就需要在创建的时候设置 `/i` 选项。

## 流派相关的特性

### .NET 字符类差

```
[a-zA-Z0-9-[g-zA-Z]]
```

这个正则表达式匹配一个单个的十六进制数字符，但是采用了一种迂回的方式。这里的基字符类可以匹配任意字母数字字符，但是后面的一个嵌套类则减去了从 g 到 z 的所有字母。这个嵌套类必须出现在基类的最后，紧跟在一个连字号之后：`<[class-[subtract]]>`。

当采用了 Unicode 的属性、区块和字母表时，字符类差（subtraction）会尤为有用。举例来说，`\p{IsThai}` 会匹配在 Thai 语区块中的任意字符。`\P{N}` 则匹配不拥有 Number 属性的任意字符。把二者使用字符类差组合起来，`\p{IsThai}-\P{N}]` 就可以匹配任意的 10 个泰语数字字符。

### Java 字符类并、差和交集

```
[a-f[A-F][0-9]]  
[a-f[A-F[0-9]]]
```

Java 支持把一个字符类嵌套在另外一个类中。如果嵌套类是直接包含的，则结果是两个字符类的并（union）。你可以嵌套任意多的类。上面给出的两个正则表达式与最初所给的不含多余方括号的正则表达式拥有完全相同的效果。

```
[\w&&[a-fA-F0-9\s]]
```

如果举办一个正则表达式猜谜大赛，那么上面这个式子很可能会得奖。这里的基本字符类可以匹配任意单词字符。嵌套类则会匹配任意十六进制数字与任意空白字符。最后所得到的类则是这两个类的交集（intersection），只会匹配十六进制的数字。因为基类并不匹配空白字符，而嵌套类不能匹配 `[g-zA-Z_]`，因此在最后的字符类中会去掉它们，只保留十六进制数字。

```
[a-zA-Z0-9&&[^g-zA-Z_]]
```

这个正则表达式匹配单个的十六进制数字符，它同样也采用了一种迂回的方式。这里

的基本字符类可以匹配任意的字母数字字符，而随后使用了一个嵌套类来减去从 g 到 z 的所有字母。这个嵌套类必须是一个否定字符类，并且要紧跟在两个&符号之后：`<[class&&[^subtract]]>`。

字符类交集和差在使用到 Unicode 的属性、区块和字母表时，是非常有用的。因此，`\p{InThai}` 会匹配在 Thai 语区块中的任意字符，而 `\P{N}` 则会匹配不拥有 Number 属性的任意字符。那么，`<[\p{InThai}&&[\P{N}]]>` 就可以匹配任意的 10 个泰语数字字符。如果你想了解在 `\p` 正则记号中可能会存在的细微差别，请参考实例 2.7 中的讲解。

## 参见

实例 2.1、2.2 和 2.7。

## 2.4 匹配任意字符

### 问题描述

匹配一个被单引号包住的字符。先给出一种解决方案允许在引号之间出现除了换行符之外的任意单个字符。再给出另外一种解决方案真正允许出现任意字符，包括换行符在内。

### 解决方案

#### 除了换行符之外的任意字符

```
'.'
```

正则选项：无（“点号匹配换行符”选项必须关闭）  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

#### 包含换行符在内的任意字符

```
'.'
```

正则选项：点号匹配换行符  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

```
'[\s\S]'
```

正则选项：无  
正则流派：JavaScript

### 讨论

#### 除了换行符之外的任意字符

点号是最古老也是最简单的正则表达式特性之一。它的含义永远是匹配任意单个字符。然而，对于任意字符到底意味着什么则存在一些混淆。用来处理正则表达式的最古老

的工具在处理文件时是逐行处理的，因此，从来不可能在目标文本中出现包含换行符的情形。在本书中讨论的编程语言则会把目标文本当作一个整体来处理，而不去管其中到底会包含多少个换行符。如果想要使用真正的逐行处理，那么你就必须编写一些代码来把目标文本分割成包含单个文本行的数组，然后把正则表达式应用到该数组中的每行之上。下一章中的实例 3.21 会讲解这样做的方法。

Perl 语言的开发者 Larry Wall 希望 Perl 能够保留基于文本行的工具的传统行为，也就是点号拥有不会匹配换行符 (`\n`)。本书中讨论的所有其他流派也都采取了相同的策略。因此，`\.` 会匹配除了换行字符之外的任意单个字符。

## 包含换行符在内的任意字符

如果确实想允许你的正则表达式可以跨越多个文本行，那么就需要打开“点号匹配换行符”的选项。这个选项可能会存于不同的名称之下。Perl 以及许多其他流派会把它称作“单行 (single line)”模式，这听起来有些让人摸不着头脑；而在 Java 中则把它称为“dot all”模式。下一章中的实例 3.4 会介绍所有的细节。不管在你所喜爱的编程语言中这个选项所用的名称是什么，都可以把它当作是“点号匹配换行符”模式。因为这就是该选项的确切含义。

而对于 JavaScript 来说，就需要使用另外一种解决方案，因为在其中并不包含“点号匹配换行符”的选项。按照在实例 2.3 中的讲解，`\s` 会匹配任意空白字符，`\S` 而则会匹配 `\s` 不能匹配的任意字符。把这两者组合起来构成 `\[\s\S\]`，这样就会得到一个包含所有字符的字符类，其中也包含了换行符。类似的，`\[d\D]` 和 `\[w\W]` 也会产生同样的效果。

## 点号的滥用

点号是最经常被滥用的正则表达式特性。例如，`\d\d.\d\d.\d\d` 并不是用来匹配日期的好方法。它的确会匹配到 05/16/08，但是它同时也会匹配 99/99/99。更为甚者，它还会匹配 12345678。

如何使用一个正则表达式来只匹配合法的日期会在后面的章节中讲解。然而显然把点号替换成一个更合适的字符类是非常容易的。`\d\d[/.-]\d\d[/.-]\d\d` 允许使用正向斜杠、点号或者连字符来作为日期分隔符。这个正则表达式还是会匹配 99/99/99，但是至少不会再匹配 12345678。



### 提示

虽然在前面的例子中，字符类之内包含了一个点号，但这只是一个巧合。事实上，在字符类之内，点号就是一个字面字符。在上面这个正则表达式中之所以会包含点号，是因为在一些国家（例如德国），点号会被用来作为日期分隔符。

最好只有当你确实想要允许出现任意字符时，才使用点号。而在任何其他场合，都应当使用一个字符类或者是否定字符类来实现。

## 变体

```
(?s)('.')
正则选项: 无
正则流派: .NET、Java、PCRE、Perl、Python

(?m)('.')
正则选项: 无
正则流派: Ruby
```

如果在正则表达式之外打开了“点号匹配换行符”模式，那么你可以在正则表达式的开始处放一个模式修饰符。在实例 2.1 中的“不区分大小写的匹配”小节中，我们已经介绍了模式修饰符的概念，并且也知道了 JavaScript 并不会对此提供支持。

在 .NET、Java、PCRE、Perl 和 Python 中，`<(?s)>` 是用于“点号匹配换行符”模式的模式修饰符。这里的 s 代表的是“single line (单行)”模式，也就是在 Perl 中给“点号匹配换行符”所起的令人很容易混淆的名字。

这个术语看起来实在很容易让人感到混淆，并因此造成了 Ruby 的正则引擎开发者在复制它的时候搞错了。Ruby 中使用 `<(?m)>` 来打开“点号匹配换行符”模式。除了使用的字母不同之外，功能是完全一样的。即使在 Ruby 1.9 的新引擎中，依然还在继续使用 `<(?m)>` 来表示“点号匹配换行符”。`<(?m)>` 在 Perl 中的含义会在实例 2.5 中进行讲解。

## 参见

实例 2.3、3.4 和 3.21。

## 2.5 匹配文本行起始和/或文本行结尾

### 问题描述

分别创建 4 个正则表达式。匹配单词 `alpha`，但是只有当它出现在目标文本最开始的时候。匹配单词 `omega`，但是只有当它出现在目标文本结尾处的时候。匹配单词 `begin`，但是只有当它出现在文本行开始处的时候。匹配单词 `end`，但是只有当它出现在文本行结尾的时候。

### 解决方案

#### 目标文本的开始

```
^alpha
正则选项: 无 ("^和$匹配换行处" 选项必须关掉)
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python
```

```
\Alpha
```

正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

## 目标文本的结尾

```
omega$
```

正则选项: 无 (“^和\$匹配换行处” 选项必须关掉)  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python

```
omega\Z
```

正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

## 一行开始

```
^begin
```

正则选项: ^和\$匹配换行处  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 一行结尾

```
end$
```

正则选项: ^和\$匹配换行处  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

### 定位符和文本行

正则表达式中的记号`(^)`、`(\$)`、`(\A)`、`(\Z)` 和 `(\z)` 被称作是定位符 (anchor)。它们并不匹配任意字符。事实上，它们匹配的是特定的位置，也就是说把正则表达式瞄准这些位置来进行匹配。

文本行指的是位于目标文本的开始和第一个换行符，在两个换行符之间，或者在最后一个换行符和目标文本结尾之间的那部分目标文本。如果在目标文本中不包含换行符的话，你的整个目标就会被认为是一行。因此，下面的文本包含有 4 行，分别是 `one`、`two`、一个空串和 `four`。

```
one
two

four
```

这个文本可以在程序中采用如下的表示：`oneLFtwoLFLFfour`。

### 目标文本的起始

定位符`\A`总是会匹配目标文本的最开始处，也就是在第一个字符的前面。这也是它会

匹配的唯一位置。可以把 `\A` 放到你的正则表达式的开始处，这样就可以检查目标文本是否是由你想要匹配的文本来开头的。注意这里的“`A`”必须是大写的。

JavaScript 不支持 `\A`。

定位符 `\^` 与 `\A` 是等价的，前提是不能打开“`^` 和 `$` 匹配换行处”这个选项。对于除了 Ruby 之外的所有其他正则流派来说，该选项都是默认关闭的。但是要记住在 Ruby 中并没有提供可以关掉这个选项的手段。

除非你使用的是 JavaScript，那么我们推荐你总是使用 `\A`，而不是 `\^`。`\A` 的含义总是保持不变的，因此可以避免由于正则选项设置而造成的混淆或错误。

## 目标文本的结尾

定位符 `\Z` 和 `\z` 总是会匹配目标文本的结尾处，也就是说在最后一个字符之后。把 `\Z` 或 `\z` 放到正则表达式的结尾处，就可以测试是否目标文本会以你想要匹配的文本来作为结束。

.NET、Java、PCRE、Perl 和 Ruby 同时支持 `\Z` 和 `\z`。Python 只支持 `\Z`。JavaScript 则根本不提供对 `\Z` 或 `\z` 的支持。

`\Z` 和 `\z` 的唯一区别是当目标文本的最后一个字符是换行符的时候。在这种情形下，`\Z` 可以匹配目标文本的最结尾处，也就是在最后的换行符之后的位置，同时也可以匹配紧跟在这个换行符之前的位置。这样做的好处是，你可以放心地搜索 `\omega\Z`，而不必担心在你的目标文本的最后是不是会存在一个多余的换行符。当逐行读入一个文件的时候，有些工具会包含行末的换行符，而有些工具则不然；`\omega\Z` 会把这种区别隐藏起来。`\z` 则只会匹配目标文本的最末尾处，因此如果存在一个多余换行符的话，那么它就无法匹配。

定位符 `\$` 与 `\Z` 是等价的，前提是不要打开“`^` 和 `$` 匹配换行处”这个选项。对于除了 Ruby 之外的所有其他正则流派来说，该选项都是默认关闭的。而且在 Ruby 中并没有提供可以关掉这个选项的手段。正像 `\Z` 一样，`\$` 也会匹配目标文本的最末尾处，以及（如果存在的话）在最后一个换行符之前的位置。

为了帮助你更好地理解这里的细微差别，我们来看一个 Perl 中的例子。假设 `$/`（当前记录分隔符）被设置为缺省的 `\n`，那么下面的 Perl 语句会从终端（标准输入）读入一行：

```
$line = <>;
```

Perl 会在变量 `$line` 的内容中保留换行符。因此，采用像 `\end•of•input.\z` 这样的表达式就无法匹配该变量。而如果使用 `\end•of•input.\n` 和 `\end•of•input.\$` 都可以产生匹配，因为它们会忽略后面多余的换行。

为了方便起见，Perl 程序员通常会使用下面的命令来去掉换行符：

```
chomp $line;
```

在执行了上述操作之后，上面所给的 3 种定位符都会产生匹配。（技术上来说，`chomp`会去掉当前记录分隔符组成的字符串。）

除非你使用的是 JavaScript，那么我们推荐你总是使用 `\Z`，而不是 `\$`。`\Z` 的含义总是保持不变的，因此可以避免由于正则选项设置而造成的混淆或错误。

## 文本行开始

默认来说，`\^` 只会匹配目标文本的开始处，就像 `\A` 一样。只有在 Ruby 中，`\^` 才会总是匹配一行的开始位置。所有其他流派都要求打开一个选项来使得`^`和`$`这两个符号可以匹配换行处。这个选项通常被称作是“多行（multiline）”模式。

千万别把这个模式同“单行”模式搞混，因为单行模式最好应该还是被称作是“点号匹配换行符”模式。“多行”模式只会影响到`^`和`$`这两个符号；而“单行”模式则只会影响到点号（如实例 2.4 所示）。你完全可以同时打开“单行”和“多行”模式。在默认情况下，这两个选项都是关闭的。

在设置了正确的选项之后，`\^` 就可以匹配目标文本中每行的开始。严格来讲，它匹配文件中第一个字符之前的位置，以及在目标文本中每个换行符之后的位置。使用 `\n\^` 是冗余的，因为 `\^` 总是会匹配 `\n` 之后的位置。

## 文本行结束

默认来说，`\$` 只会匹配目标文本的结尾处或者是最后一个换行之前的位置，就像 `\Z` 一样。只有在 Ruby 中，`\$` 才会总是匹配一行的结尾处。所有其他流派都要求你打开“多行”模式来使得`^`和`$`这两个符号可以匹配换行处。

在设置了正确的选项之后，`\$` 会匹配目标文本中每行的结束。（当然，它同样会匹配目标文本中的最后一个字符之后的位置，因为这个位置也总是一行的结束。）使用 `\$\n` 是冗余的，因为 `\$` 总是会匹配 `\n` 之前的位置。

## 长度为 0 的匹配

对于一个正则表达式来说，它完全可以只包含一个或者多个定位符。这样一个正则表达式会在定位符能够匹配的每个位置查找一个长度为 0 的匹配。如果把多个定位符放在一起的话，只有当所有的定位符都在同一个位置匹配的时候，该正则式才会匹配成功。

可以在查找和替换的功能中使用这样的正则表达式。可以通过替换 `\A` 或 `\Z` 来在整个目标文本之前或者之后添加一些内容。也可以通过在“`^`和`$`匹配换行处”的模式下，替换 `\^` 或 `\$`，在目标文本的每行之前或者之后添加一些内容。

把两个定位符组合起来则可以检查空行或者缺失的输入。`\A\Z` 会匹配空串，以及包含单个换行符的字符串。`\A\z` 则只能匹配空串。在“`^`和`$`匹配换行处”的模式下，`\^$`

会匹配目标文本中的每个空行。

## 变体

```
(?m)^begin  
正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Python  
  
(?m)end$  
正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Python
```

如果不能在正则表达式之外打开“^和\$匹配换行处”的模式，那么你也可以在正则表达式之前使用一个模式修饰符来达到相同的效果。在实例 2.1 中的“不区分大小写的匹配”一节中，我们已经讲解了模式修饰符，并且了解了 JavaScript 并不对此提供支持。

在.NET、Java、PCRE、Perl 和 Python 中，`(?m)` 是“^和\$匹配换行处”模式的模式修饰符。其中的 m 指的是“多行 (multiline)”模式，这个是在 Perl 中引入的用来描述“^和\$匹配换行处”模式的名称，实在很容易让人感到混淆。

如前所述，这个术语实在是太容易让人弄混了，以致 Ruby 正则引擎的开发者都无法正确地遵循这种记法。在 Ruby 中使用 `(?m)` 来打开“点号匹配换行符”模式。因此 Ruby 中的 `(?m)` 与脱字符和美元符号都不存在任何关系。在 Ruby 中，`^` 和 `\$` 总是会匹配每一行的开始和结束。

除了令人遗憾的字母使用混淆之外，Ruby 选择使用 `^` 和 `\$` 只匹配每行是非常正确的。除非你使用的是 JavaScript，我们推荐你在自己的正则表达式中都采用这种方式。

在 Jan 设计 EditPad Pro 和 PowerGREP 的时候，他遵循了相同的想法。在这些工具中找不到关于“^和\$匹配换行处”的复选框，虽然其中有个复选框的说明是“点号匹配新行 (dot matches newlines.)”。除非在你的正则表达式之前添加了`(?-m)`，就不得不使用`<A>` 和 `<Z>` 来把正则表达式定位到文件的开始和结束。

## 参见

实例 3.4 和 3.21。

## 2.6 匹配整个单词

### 问题描述

创建一个正则表达式来匹配 My cat is brown 中的 cat，但是不会匹配 category 或是 bobcat。再创建一个正则表达式来匹配 staccato 中的 cat，但是却不会匹配到上面的 3 个目标字串。

# 解决方案

## 单词边界

\bcat\b  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 非单词边界

\Bcat\B  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

### 单词边界

正则表达式记号 \b 被称作是一个单词边界（word boundary）。它会匹配一个单词的开始或结束。就它自身而言，所产生的是一个长度为 0 的匹配。**\b** 也是一个定位符，这与在前一个小节中介绍过的记号是一样的。

严格来讲，\b 会匹配如下的 3 种位置：

- 在目标文本的第一个字符之前，如果第一个字符是单词字符；
- 在目标文本的最后一个字符之后，如果最后一个字符是单词字符；
- 在目标文本的两个字符之间，其中一个是单词字符，而另外一个不是单词字符。

本书中介绍的所有流派都不提供可以只匹配单词之前或者只匹配单词之后的单独记号。除非想要创建一个正则式只包含单词边界，而不包含任何其他内容，那么这些单独记号是不必要的。在正则表达式中，位于 \b 之前和之后的记号会决定 \b 可以匹配的位置。在 \bx 和 \!b 中的 \b 只可以匹配一个单词的开始。而在 x\b 和 \b! 中的 \b 只可以匹配一个单词的结束。x\bx 和 \!b! 则永远不会匹配任何位置。

如果要使用正则表达式来支持“只匹配完整单词”的查找，那么只需要把该单词放在两个单词边界之间，就像我们前面给出的 \bcat\b。第一个 \b 要求 c 出现在字符串的最开始处，或者是在一个非单词字符后面。换行符被认为是非单词字符。如果换行符紧跟在一个单词字符后面，那么 \b 会匹配换行符后面的位置。它同样会匹配紧跟着一个单词字符之前的换行符。这样的话，一个占据了一整行的单词也可以通过一个“只匹配完整单词”的查找来发现。 \b 不会受到“多行”模式或是 (?m) 的影响，而这也是为什么本书把“多行”模式称作是“^ 和 \$ 匹配换行处”的原因之一。

## 非单词边界

`\B` 会匹配在目标文本中 `\b` 不匹配的每一个位置。换句话说，`\B` 会匹配不属于单词开始或结束的每一个位置。

严格来讲，`\B` 匹配如下的 5 个位置：

- 在目标文本的第一个字符之前，如果第一个字符不是单词字符；
- 在目标文本的最后一个字符之后，如果最后一个字符不是单词字符；
- 在两个单词字符之间；
- 在两个非单词字符之间；
- 空串。

`\Bcat\B` 会匹配 `staccato` 中的 `cat`。但是不会匹配 `My cat is brown`、`category` 或者 `bobcat`。

如果想要进行与“只匹配完整单词”相反的查找（也就是说，不包括 `My cat is brown`，但是包括 `staccato`、`category` 和 `bobcat`），那么就需要采用多选结构来把 `\Bcat` 和 `\cat\B` 组合成为 `\Bcat|cat\B`。其中，`\Bcat` 会匹配 `staccato` 和 `bobcat` 中的 `cat`。`\cat\B` 会匹配 `category` 中的 `cat`（如果 `\Bcat` 还没有匹配到，它也可以匹配 `staccato`）。实例 2.8 会详细讲解多选结构。

## 单词字符

我们前面一直在讲单词边界，但是却没有涉及什么是单词字符（word character）。单词字符就是可以在单词中出现的字符。在实例 2.3 中的“简写”小节中，我们讨论了哪些字符是包含在 `\w` 中的，`\w` 匹配单个的单词字符。不幸的是，对于 `\b` 来说情形则有些不同。

虽然本书中的所有流派都支持 `\b` 和 `\B`，但是它们对于到底哪些字符属于单词字符则有所不同。

在.NET、JavaScript、PCRE、Perl、Python 和 Ruby 中，`\b` 都会匹配两个字符之间的位置，其中一个字符可以被 `\w` 匹配，而另外一个字符则可以被 `\W` 匹配。`\B` 则总是匹配两个同时被 `\w` 或 `\W` 匹配的字符之间的位置。

JavaScript、PCRE 和 Ruby 只把 ASCII 字符看做是单词字符。`\w` 因此与 `[a-zA-Z0-9_]` 是完全等同的。在这些流派中，你可以对像英语这样的只由不含读音符号的 A 到 Z 的字母组成的单词进行“只匹配整个单词”的查找。但是这些流派对于其他语言，例如西班牙语或俄语，就无法进行“只匹配整个单词”的查找。

.NET 和 Perl 把所有语言字母表中的字母和数字都当作单词字符。在这些流派中，你可以对任意语言中的单词进行“只匹配整个单词”的查找，这其中也包括不使用拉丁字母表的语言。

Python 则为你提供了一个选项。只有在创建正则式时传递了 UNICODE 或是 U 选项，非 ASCII 的字符才会被包括进来。这个选项会对 `\b` 和 `\w` 产生相同的影响。

Java 则表现得不是很一致。`\w` 只匹配 ASCII 字符。但是 `\b` 则是支持 Unicode 的，因此可以支持任何字母表。在 Java 中，`\b\w\b` 会匹配一个单个的英语字母、数字或是在任何语言中都不会作为单词一部分出现的下划线。`\bкошка\b` 会正确匹配 cat 在俄语中对应的单词，因为 `\b` 是支持 Unicode 的。但是 `\w+` 不会匹配任何泰国语单词，因为 `\w` 只会匹配 ASCII 字符。

## 参见

实例 2.3。

## 2.7 Unicode 代码点、属性、区块和脚本

### 问题描述

使用一个正则表达式来查找商标符号 (™)，要求通过指定其 Unicode 代码点，而不是复制并粘贴一个实际上的商标符号。如果你选择复制并粘贴，那么商标符号可以被看作另外一个字面字符，虽然并不能从键盘上直接输入它。字面字符已经在实例 2.1 中进行了讨论。

创建一个正则表达式来匹配拥有“货币符号 (Currency Symbol)” Unicode 属性的任意字符。Unicode 属性也被称作是 Unicode 类别。

创建一个正则表达式来匹配在“希腊扩展” Unicode 区块中的任意字符。

创建一个正则表达式来匹配根据 Unicode 标准属于希腊字母表一部分的任意字符。

创建一个正则表达式来匹配一个字形 (grapheme)，它平常也被当作一个字符：一个基本字符加上它所有的组合标记。

### 解决方案

#### Unicode 代码点

`\u2122`  
正则选项：无  
正则流派：.NET、Java、JavaScript、Python

当这个正则表达式被作为一个 Unicode 字符串 (`u"\u2122"`) 引用的时候，它只在 python 中可用。

`\x{2122}`  
正则选项：无  
正则流派：PCRE、Perl、Ruby 1.9

PCRE 必须使用 UTF-8 支持进行编译；在 PHP 中，需要使用/u 模式修饰符来打开 UTF-8 支持。Ruby 1.8 不支持 Unicode 正则表达式。

## Unicode 属性或类别

\p{Sc}

正则选项：无

正则流派：.NET、Java、PCRE、Perl、Ruby 1.9

PCRE 必须使用 UTF-8 支持进行编译；在 PHP 中，需要使用/u 模式修饰符来打开 UTF-8 支持。JavaScript 和 Python 不支持 Unicode 属性。Ruby 1.8 不支持 Unicode 正则表达式。

## Unicode 区块

\p{IsGreekExtended}

正则选项：无

正则流派：.NET、Perl

\p{InGreekExtended}

正则选项：无

正则流派：Java、Perl

JavaScript、PCRE、Python 和 Ruby 不支持 Unicode 区块。

## Unicode 字母表

\p{Greek}

正则选项：无

正则流派：PCRE、Perl、Ruby 1.9

Unicode 字母表（script）支持要求 PCRE 6.5 或者更新版本，而且 PCRE 必须使用 UTF-8 支持进行编译。在 PHP 中，需要使用/u 模式修饰符来打开 UTF-8 支持。.NET、JavaScript 和 Python 不支持 Unicode 属性。Ruby 1.8 不支持 Unicode 正则表达式。

## Unicode 字形

\X

正则选项：无

正则流派：PCRE、Perl

PCRE 和 Perl 都包含一个专门的记号来匹配字形，但是同时也支持如下的使用 Unicode 属性的方式来完成相同功能。

\P{M}\p{M}\*

正则选项：无

正则流派：.NET、Java、PCRE、Perl、Ruby 1.9

PCRE 必须使用 UTF-8 支持进行编译；在 PHP 中，需要使用/u 模式修饰符来打开 UTF-8 支持。JavaScript 和 Python 不支持 Unicode 属性。Ruby 1.8 不支持 Unicode 正则表达式。

# 讨论

## Unicode 代码点

代码点（code point）是在 Unicode 字符数据库中的一项。代码点与字符是不一样的，当然这还要基于你给“字符”什么样的含义。在 Unicode 中，在屏幕上作为字符出现的符号被称作是一个字形（grapheme）。

Unicode 代码点 U+2122 表示的是“商标符号”字符。根据所使用的正则流派的不同，你可以使用 `\u2122` 或 `\x{2122}` 来匹配这个字符。

`\w` 的语法要求使用正好四位十六进制数字。这意味着你只能用它来表示 Unicode 代码点 U+0000 到 U+FFFF。`\x` 语法则允许出现任意位数的十六进制数字，可以支持所有的代码点，也就是从 U+000000 直到 U+10FFFF。你可以采用 `\x{E0}` 或 `\u00E0` 来匹配 U+00E0。U+100000 之后代码点是很少使用的，同时在字体和操作系统中都没有对它们提供很好的支持。

代码点可以在字符类之内和之外进行使用。

## Unicode 属性或类别

每个 Unicode 代码点都正好拥有一个 Unicode 属性（property），或者也可以属于单个的 Unicode 类别（category）。这两个术语的实际含义是一样的。一共存在 20 个 Unicode 类别，被组织为 7 个超类。

<code>&lt;\p{L}&gt;</code>	任意语言的任意字母
<code>&lt;\p{Ll}&gt;</code>	一个拥有大写变体的小写字母
<code>&lt;\p{Lu}&gt;</code>	一个拥有小写变体的大写字母
<code>&lt;\p{Lt}&gt;</code>	当只有单词的第一个字母被大写时，出现在单词开头的字母
<code>&lt;\p{L&amp;}&gt;</code>	以小写和大写变体出现的字母（囊括了 Ll、Lu 和 Lt 的情形）
<code>&lt;\p{Lm}&gt;</code>	可以像字母一样使用的特殊字符
<code>&lt;\p{Lo}&gt;</code>	不拥有小写和大写变体的字母或者象形文字
<code>&lt;\p{M}&gt;</code>	专门与另外一个字符组合使用的字符（重音符号、变音符号、包围框等）
<code>&lt;\p{Mn}&gt;</code>	只能与另外一个字符组合使用，同时不占用额外空间的字符（重音符号、变音符号等）
<code>&lt;\p{Mc}&gt;</code>	只能与另外一个字符组合使用，但是会占用额外空间的字符（比如在许多东方语言中的元音符号）

<\p{Me}> 包含另外一个字符的字符（圆圈、方框、键帽等）

<\p{Z}> 任何种类的空白或不可见的分隔符

<\p{Zs}> 一个不可见的空白字符，但是却会占用空间

<\p{Zl}> 行分隔符 U+2028

<\p{Zp}> 段分隔符 U+2029

<\p{S}> 数学符号、货币符号、杂锦字型 (dingbats)、制表符 (box-drawing)，等等

<\p{Sm}> 任意数学符号

<\p{Sc}> 任意货币符号

<\p{Sk}> 组合符号（标记）自己作为一个完整字符

<\p{So}> 不属于属性符号、货币符号或者组合符号的各种符号

<\p{N}> 任意字母表中的任意种类的数字字符

<\p{Nd}> 除了表意字母表之外的任意字母表中的数字 0~9

<\p{Nl}> 看起来像是字母的数字，例如罗马数字

<\p{No}> 上标或者下标数字，或是非 0~9 的数字（不包括象形字母表中的数字）

<\p{P}> 任意种类的标点字符

<\p{Pd}> 任意种类的连字号或短横线

<\p{Ps}> 任意种类的左括号

<\p{Pe}> 任意种类的右括号

<\p{Pi}> 任意种类的左引号

<\p{Pf}> 任意种类的右引号

<\p{Pc}> 像下划线一样用来连接单词的标点符号

<\p{Po}> 不属于短横线、括号、引号或连接符的任意一种标点字符

<\p{C}> 不可见的控制字符和未使用的代码点

<\p{Cc}> ASCII 中的 0x00~0x1F 或 Latin-1 中的 0x80~0x9F 控制字符

<\p{Cf}>

不可见的格式标志  
<\p{Co}>  
保留使用的任意代码点  
<\p{Cs}>  
在 UTF-16 编码中一个替代对的一半  
<\p{Cn}>  
没有赋予任意字符的代码点

<\p{Ll}> 匹配拥有 Ll 属性，也就是“小写字母”属性的单个代码点。<\p{L}> 可以被用作 <[\p{Ll}\p{Lu}\p{Lt}\p{Lm}\p{Lo}]> 的简写形式，用来匹配在任意“字母”类别中的单个代码点。

<\P> 是 <\p> 的否定版本。<\P{Ll}> 匹配不拥有 Ll 属性的单个代码点。<\P{L}> 匹配不拥有任何“字母”属性的单个代码点。这与 <[\P{Ll}\P{Lu}\P{Lt}\P{Lm}\P{Lo}]> 是不一样的，后者会匹配所有的代码点。<\P{Ll}> 匹配拥有 Lu 属性（以及除了 Ll 之外的所有其他属性）的代码点，而 <\P{Lu}> 会包含 Ll 代码点。把这两个组合到一个代码点类中就可以把所有可能的代码点都包括进来。

## Unicode 区块

Unicode 字符数据库把所有代码点划分为不同的区块。每个区块由一个连续范围内的代码点组成。代码点 U+0000~U+FFFF 被划分为 105 个区块：

U+0000...U+007F \p{InBasic\_Latin}  
U+0080...U+00FF \p{InLatin-1\_Supplement}  
U+0100...U+017F \p{InLatin\_Extended-A}  
U+0180...U+024F \p{InLatin\_Extended-B}  
U+0250...U+02AF \p{InIPA\_Extensions}  
U+02B0...U+02FF \p{InSpacing\_Modifier\_Letters}  
U+0300...U+036F \p{InCombining\_Diacritical\_Marks}  
U+0370...U+03FF \p{InGreek\_and\_Coptic}  
U+0400...U+04FF \p{InCyrillic}  
U+0500...U+052F \p{InCyrillic\_Supplementary}  
U+0530...U+058F \p{InArmenian}  
U+0590...U+05FF \p{InHebrew}  
U+0600...U+06FF \p{InArabic}  
U+0700...U+074F \p{InSyriac}  
U+0780...U+07BF \p{InThaana}  
U+0900...U+097F \p{InDevanagari}  
U+0980...U+09FF \p{InBengali}  
U+0A00...U+0A7F \p{InGurmukhi}  
U+0A80...U+0AFF \p{InGujarati}  
U+0B00...U+0B7F \p{InOriya}  
U+0B80...U+0BFF \p{InTamil}  
U+0C00...U+0C7F \p{InTelugu}  
U+0C80...U+0CFF \p{InKannada}  
U+0D00...U+0D7F \p{InMalayalam}  
U+0D80...U+0DFF \p{InSinhala}

U+0E00...U+0E7F \p{InThai}  
U+0E80...U+0EFF \p{InLao}  
U+0F00...U+0FFF \p{InTibetan}  
U+1000...U+109F \p{InMyanmar}  
U+10A0...U+10FF \p{InGeorgian}  
U+1100...U+11FF \p{InHangul\_Jamo}  
U+1200...U+137F \p{InEthiopic}  
U+13A0...U+13FF \p{InCherokee}  
U+1400...U+167F \p{InUnified\_Canadian\_Aboriginal\_Syllabics}  
U+1680...U+169F \p{InOgham}  
U+16A0...U+16FF \p{InRunic}  
U+1700...U+171F \p{InTagalog}  
U+1720...U+173F \p{InHanunoo}  
U+1740...U+175F \p{InBuhid}  
U+1760...U+177F \p{InTagbanwa}  
U+1780...U+17FF \p{InKhmer}  
U+1800...U+18AF \p{InMongolian}  
U+1900...U+194F \p{InLimbu}  
U+1950...U+197F \p{InTai\_Le}  
U+19E0...U+19FF \p{InKhmer\_Symbols}  
U+1D00...U+1D7F \p{InPhonetic\_Extensions}  
U+1E00...U+1EFF \p{InLatin\_Extended\_Additional}  
U+1F00...U+1FFF \p{InGreek\_Extended}  
U+2000...U+206F \p{InGeneral\_Punctuation}  
U+2070...U+209F \p{InSuperscripts\_and\_Subscripts}  
U+20A0...U+20CF \p{InCurrency\_Symbols}  
U+20D0...U+20FF \p{InCombining\_Diacritical\_Marks\_for\_Symbols}  
U+2100...U+214F \p{InLetterlike\_Symbols}  
U+2150...U+218F \p{InNumber\_Forms}  
U+2190...U+21FF \p{InArrows}  
U+2200...U+22FF \p{InMathematical\_Operators}  
U+2300...U+23FF \p{InMiscellaneous\_Technical}  
U+2400...U+243F \p{InControl\_Pictures}  
U+2440...U+245F \p{InOptical\_Character\_Recognition}  
U+2460...U+24FF \p{InEnclosed\_Alphanumerics}  
U+2500...U+257F \p{InBox\_Drawing}  
U+2580...U+259F \p{InBlock\_Elements}  
U+25A0...U+25FF \p{InGeometric\_Shapes}  
U+2600...U+26FF \p{InMiscellaneous\_Symbols}  
U+2700...U+27BF \p{InDingbats}  
U+27C0...U+27EF \p{InMiscellaneous\_Mathematical\_Symbols-A}  
U+27F0...U+27FF \p{InSupplemental\_Arrows-A}  
U+2800...U+28FF \p{InBraille\_Patterns}  
U+2900...U+297F \p{InSupplemental\_Arrows-B}  
U+2980...U+29FF \p{InMiscellaneous\_Mathematical\_Symbols-B}  
U+2A00...U+2AFF \p{InSupplemental\_Mathematical\_Operators}  
U+2B00...U+2BFF \p{InMiscellaneous\_Symbols\_and\_Arrows}  
U+2E80...U+2EFF \p{InCJK\_Radicals\_Supplement}  
U+2F00...U+2FDF \p{InKangxi\_Radicals}

```
U+2FF0...U+2FFF \p{InIdeographic_Description_Characters}
U+3000...U+303F \p{InCJK_Symbols_and_Punctuation}
U+3040...U+309F \p{InHiragana}
U+30A0...U+30FF \p{InKatakana}
U+3100...U+312F \p{InBopomofo}
U+3130...U+318F \p{InHangul_Compatibility_Jamo}
U+3190...U+319F \p{InKanbun}
U+31A0...U+31BF \p{InBopomofo_Extended}
U+31F0...U+31FF \p{InKatakana_Phonetic_Extensions}
U+3200...U+32FF \p{InEnclosed_CJK_Letters_and_Months}
U+3300...U+33FF \p{InCJK_Compatibility}
U+3400...U+4DBF \p{InCJK_Unified_Ideographs_Extension_A}
U+4DC0...U+4DFF \p{InYijing_Hexagram_Symbols}
U+4E00...U+9FFF \p{InCJK_Unified_Ideographs}
U+A000...U+A48F \p{InYi_Syllables}
U+A490...U+A4CF \p{InYi_Radicals}
U+AC00...U+D7AF \p{InHangul_Syllables}
U+D800...U+DB7F \p{InHigh_Surrogates}
U+DB80...U+DBFF \p{InHigh_Private_Use_Surrogates}
U+DC00...U+DFFF \p{InLow_Surrogates}
U+E000...U+F8FF \p{InPrivate_Use_Area}
U+F900...U+FAFF \p{InCJK_Compatibility_Ideographs}
U+FB00...U+FB4F \p{InAlphabetic_Presentation_Forms}
U+FB50...U+FDFF \p{InArabic_Presentation_Forms-A}
U+FE00...U+FE0F \p{InVariation_Selectors}
U+FE20...U+FE2F \p{InCombining_Half_Marks}
U+FE30...U+FE4F \p{InCJK_Compatibility_Forms}
U+FE50...U+FE6F \p{InSmall_Form_Variants}
U+FE70...U+FEFF \p{InArabic_Presentation_Forms-B}
U+FF00...U+FFEF \p{InHalfwidth_and_Fullwidth_Forms}
U+FFF0...U+FFFF \p{InSpecials}
```

Unicode 区块是一个连续范围之内的代码点。虽然许多区块都拥有 Unicode 字母表和 Unicode 类别的名称，但是它们并不是百分之百相对应的。一个区块的名称只是用来说明它的主要用途。

Currency 区块中并不包含美元和日元符号。由于历史的原因，这些符号在 Basic\_Latin 和 Latin-1\_Supplement 区块中才能找到。但是二者都拥有 Currency Symbol 属性。如果要匹配任意的货币符号，那么应该使用 `\p{Sc}`，而不是 `\p{InCurrency}`。

大多数区块中都包含没有分配的代码点，这些都被包括在了属性 `\p{Cn}` 中。其他的 Unicode 属性，以及所有的 Unicode 字母表中，都不会包含未分配的代码点。

`\p{InBlockName}` 的语法可以在.NET 和 Perl 中使用。而 Java 使用的则是 `\p{IsBlockName}` 的语法。

Perl 同样支持 Is 变体形式，但是我们推荐你坚持使用 In 的语法，这是为了不与 Unicode 字母表发生混淆。对于字母表来说，Perl 支持 `\p{Script}` 和 `\p{IsScript}`，但是不支

持`\p{InScript}`。

## Unicode 字母表

除了未分配的代码点之外，每个 Unicode 代码点都是刚好属于一种 Unicode 字母表的一部分。未分配的代码点不属于任意字母表。到 U+FFFF 之前的已经分配的所有代码点被分配到了如下字母表中：

<code>\p{Common}</code>	<code>\p{Katakana}</code>
<code>\p{Arabic}</code>	<code>\p{Khmer}</code>
<code>\p{Armenian}</code>	<code>\p{Lao}</code>
<code>\p{Bengali}</code>	<code>\p{Latin}</code>
<code>\p{Bopomofo}</code>	<code>\p{Limbu}</code>
<code>\p{Braille}</code>	<code>\p{Malayalam}</code>
<code>\p{Buhid}</code>	<code>\p{Mongolian}</code>
<code>\p{CanadianAboriginal}</code>	<code>\p{Myanmar}</code>
<code>\p{Cherokee}</code>	<code>\p{Ogham}</code>
<code>\p{Cyrillic}</code>	<code>\p{Oriya}</code>
<code>\p{Devanagari}</code>	<code>\p{Runic}</code>
<code>\p{Ethiopic}</code>	<code>\p{Sinhala}</code>
<code>\p{Georgian}</code>	<code>\p{Syriac}</code>
<code>\p{Greek}</code>	<code>\p{Tagalog}</code>
<code>\p{Gujarati}</code>	<code>\p{Tagbanwa}</code>
<code>\p{Gurmukhi}</code>	<code>\p{TaiLe}</code>
<code>\p{Han}</code>	<code>\p{Tamil}</code>
<code>\p{Hangul}</code>	<code>\p{Telugu}</code>
<code>\p{Hanunoo}</code>	<code>\p{Thaana}</code>
<code>\p{Hebrew}</code>	<code>\p{Thai}</code>
<code>\p{Hiragana}</code>	<code>\p{Tibetan}</code>
<code>\p{Inherited}</code>	<code>\p{Yi}</code>
<code>\p{Kannada}</code>	

字母表是由某种人类特定语言书写系统使用的一组代码点。一些字母表，比如 Thai，对应于单个的人类语言。其他字母表，例如 Latin，则会涉及多种语言。有些语言是由多种字母表来组成的。例如，其中并不存在一种日语 Unicode 字母表；事实上，Unicode 提供了日语文档中通常会使用到的 Hiragana、Katakana、Han 和 Latin 字母表。

你可能已经注意到了，在上面的列表中列在第一个的 Common 字母表没有按照字母顺序排列。这种字母表包含了对于许多字母表相同的各种字符，例如标点、空白符号以及其他各色符号。

## Unicode 字形

当使用到组合标志（combining marks）的时候，代码点与字符的区别就展现出来了。Unicode 代码点 U+0061 是“拉丁小写字母 a”，而 U+00E0 是“加了重音符号的拉丁小写字母 a”。通常来说大多数人把二者都称作字符。

U+0300 是“重音组合符号”的组合标志。它只有在一个字母之后使用才有意义。一个包含 Unicode 代码点 U+0061 U+0300 的字符串会被显示为 à，这同 U+00E0 是完全一样

的。组合标志 U+0300 会被显示到字符 U+0061 的顶上。

之所以会出现两种不同方式来表示一个加重音符号的字符，是因为在许多历史上的字符集中，把“带有重音符号的 a”编码为了单个字符。Unicode 的设计者认为有必要与这些常用的遗留字符集保持一对一的映射，另外 Unicode 新增了把标志和基本字母分开的表示方式，这样可以使遗留字符集无法支持的任意组合成为可能。

对于一个正则表达式用户来说，重要的是本书中介绍的所有正则流派操作的都是代码点而不是图形化的字符。当我们说正则表达式`\.`匹配单个字符的时候，实际上的含义是它匹配单个的代码点。如果你的目标文本中包含了两个代码点 U+0061 U+0300，在像 Java 这样的编程语言中它可以使用字符串常量"`\u0061\u0300`"来表示，那么一个点号只能匹配代码点 U+0061（也就是 a），而不会匹配重音符号 U+0300。使用正则表达式`\.`才可以同时匹配二者。

Perl 和 PCRE 提供了一个特殊的正则记号`\X`，用来匹配任意单个的 Unicode 字形。本质上说，它是 Unicode 版本的特殊点号。它会匹配不带有组合标志的任意 Unicode 代码点，以及紧跟其后的所有组合标志（如果有的话）。我们也可以使用 Unicode 属性来做同样的事情：`\P{M}\p{M}*`。`\X`会在文本`\u00E0`中找到两个匹配，而不管它是如何进行编码的。如果它被编码为"`\u00E0\u0061\u0300`"，那么第一个匹配是"`\u00E0`"，第二个匹配则是"`\u0061\u0300`"。

## 变体

### 否定变体

大写形式的`\P`是小写形式的`\p`否定变体。例如，`\P{Sc}`会匹配不拥有“Currency Symbol” Unicode 属性的任意字符。所有支持`\p`的流派都会支持`\P`，并且会支持所有相应的属性、区块和字母表。

### 字符类

所有流派都允许把它们所支持的所有的`\u`、`\x`、`\p`和`\P`记号用在字符类之内。这样会把代码点所表示的字符，或者是在该类别、区块或者字母表中的字符添加到字符类中。例如，你可以用如下的正则式来匹配一个左引号（初始标点属性）、一个右引号（终止标点属性）或商标符号（U+2122）：

```
[ \p{Pi} \p{Pf} \x{2122} ].  
正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Ruby 1.9
```

### 列出所有字符

如果你的正则表达式流派不支持 Unicode 类别、区块或字母表的话，那么你可以把属于该类别、区块或字母表的字符罗列到一个字符类中。对于区块来说，这会比较容易：因为每个区块其实就是一个范围。例如，希腊语扩展（Greek

Extended) 区块包括 U+1F00~U+1FFF 的字符：

```
[\u1F00-\u1FFF]  
正则选项：无  
正则流派：.NET、Java、JavaScript、Python  
  
[\x{1F00}-\x{1FFF}]  
正则选项：无  
正则流派：PCRE、Perl、Ruby 1.9
```

然而对于大多数类别和许多字母表来说，与之等价的字符类是单个代码点和较短范围的一个冗长列表。构成每个类别和许多字母表的字符是散布在 Unicode 表中的。下面表示的是希腊语字母表：

```
[\u0370-\u0373\u0375\u0376-\u0377\u037A\u037B-\u037D\u0384\u0386-  
\u0388-\u038A\u038C\u038E-\u03A1\u03A3-\u03E1\u03F0-\u03F5\u03F6-  
\u03F7-\u03FF\u1D26-\u1D2A\u1D5D-\u1D61\u1D66-\u1D6A\u1DBF\u1F00-\u1F15-  
\u1F18-\u1F1D\u1F20-\u1F45\u1F48-\u1F4D\u1F50-\u1F57\u1F59\u1F5B\u1F5D-  
\u1F5F-\u1F7D\u1F80-\u1FB4\u1FB6-\u1FBC\u1FBD\u1FBE\u1FBF-\u1FC1-  
\u1FC2-\u1FC4\u1FC6-\u1FCC\u1FCD-\u1FCF\u1FD0-\u1FD3\u1FD6-\u1FDB-  
\u1FDD-\u1FDF\u1FE0-\u1FEC\u1FED-\u1FEF\u1FF2-\u1FF4\u1FF6-\u1FFC-  
\u1FFD-\u1FFE\u2126]
```

正则选项：无  
正则流派：.NET、Java、JavaScript、Python

在构造这个正则表达式的过程中，我们从 <http://www.unicode.org/Public/UNIDATA/Scripts.txt> 复制了希腊语的字母表，然后使用如下 3 个正则表达式进行了查找和替换。

1. 查找正则表达式 `<.*>`，并把它的匹配替换为空，从而可以删除所有注释。如果它不幸删除了所有内容，那么你需要取消操作，并记住要关闭“点号匹配换行符”的选项。
2. 打开“^和\$匹配换行处”选项，查找 `<^>`，并把它的匹配替换为 `<\u>`，这样可以给所有代码点都加上u前缀。把 `<.\>` 替换为 `<-u>` 可以得到正确的范围。
3. 最后，把 `\s+` 替换为空，以删除换行符。在字符类的前后添加括号就完成了该正则表达式。可能还需要在字符类的开始处添加u，或者删除最后多余的u，这取决于当你从 Scripts.txt 文件中复制该列表的时候是否包括了前导空行或者后续空行。

这看起来好像要做很多工作，但是实际上 Jan 只用了不到一分钟。把上面的描述写下来则要花费多出好几倍的时间。如果要用使用 `\x{}` 语法来表示的话，也同样很简单。

1. 查找正则表达式 `<.*>`，并把它的匹配替换为空，从而可以删除所有注释。如果它不幸删除了所有内容，那么你需要取消操作，并且关闭“点号匹配换行符”的选项。
2. 打开“^和\$匹配换行处”选项，查找 `<^>`，并把它的匹配替换为 `<\x{>}>`，这样可以给

所有代码点都加上\w{前缀。把<\.\> 替换为 «}-\w{» 可以得到正确的范围。

3. 最后，把<\s+> 替换为 «}-\w{»，在添加右括号的同时删除换行符。在字符类的前后添加括号就完成了该正则表达式。可能还需要在字符类的开始处添加\w{，或者删除最后多余的\u，这取决于当你从 Scripts.txt 文件中复制该列表的时候是否包括了前导空行或者后续空行。

这样所得到的结果如下：

```
[\x{0370}-\x{0373}\x{0375}\x{0376}-\x{0377}\x{037A}\x{037B}-\x{037D}+  
\x{0384}\x{0386}\x{0388}-\x{038A}\x{038C}\x{038E}-\x{03A1}+  
\x{03A3}-\x{03E1}\x{03F0}-\x{03F5}\x{03F6}\x{03F7}-\x{03FF}+  
\x{1D26}-\x{1D2A}\x{1D5D}-\x{1D61}\x{1D66}-\x{1D6A}\x{1DBF}+  
\x{1F00}-\x{1F15}\x{1F18}-\x{1F1D}\x{1F20}-\x{1F45}\x{1F48}-\x{1F4D}+  
\x{1F50}-\x{1F57}\x{1F59}\x{1F5B}\x{1F5D}\x{1F5F}-\x{1F7D}+  
\x{1F80}-\x{1FB4}\x{1FB6}-\x{1FBC}\x{1FBD}\x{1FBE}\x{1FBF}-\x{1FC1}+  
\x{1FC2}-\x{1FC4}\x{1FC6}-\x{1FCC}\x{1FCD}-\x{1FCF}\x{1FD0}-\x{1FD3}+  
\x{1FD6}-\x{1FDB}\x{1FDD}-\x{1FDF}\x{1FE0}-\x{1FEC}\x{1FED}-\x{1FEF}+  
\x{1FF2}-\x{1FF4}\x{1FF6}-\x{1FFC}\x{1FFD}-\x{1FFE}\x{2126}+  
\x{10140}-\x{10174}\x{10175}-\x{10178}\x{10179}-\x{10189}+  
\x{1018A}\x{1D200}-\x{1D241}\x{1D242}-\x{1D244}\x{1D245}]
```

正则选项：无

正则流派：PCRE、Perl、Ruby 1.9

## 参见

<http://www.unicode.org> 是 Unicode 协会的官方网站，可以下载官方的 Unicode 文档、字符表等内容。

Unicode 是一个庞大的话题，全部内容需要由整本书来进行讲解。我们推荐由 Jukka K. Korpela 所著的 *Unicode Explained* 一书。

我们不可能在一个小节中讲解你所需要知道的关于 Unicode 代码点、属性、区块和字母表的全部内容。我们甚至都还没有尝试去解释为什么你应当关心这些内容——事实上这些都是很重要的。虽然扩展的 ASCII 表示用起来会比较简单，但是在如今全球化的世界上，使用它的人早已寥寥无几了。

## 2.8 匹配多个选择分支之一

### 问题描述

创建一个正则表达式，当把它重复应用到目标文本 Mary, Jane, and Sue went to Mary's house 之上时，会匹配到 Mary、Jane、Sue，以及又一次匹配到 Mary。之后再进行的匹配尝试都会失败。

## 解决方案

Mary|Jane|Sue

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

竖线（vertical bar），或称作管道符号（pipe symbol），会把正则表达式拆分成多个选择分支。`<Mary|Jane|Sue>`会在每次匹配尝试时匹配 Mary，或者 Jane，或者 Sue。每次只会匹配一个名字，但是每次却可以匹配不同的名字。

本书中的所有正则表达式流派都会使用一个正则制导的引擎。这个引擎也就是那个使得正则表达式可以工作的软件。这里的正则制导<sup>1</sup>的含义是，在目标文本中的每个字符位置会首先匹配该正则表达式的所有可能排列，然后才会到下一个字符位置进行匹配尝试。

当你把 `<Mary|Jane|Sue>` 应用到 Mary, Jane, and Sue went to Mary's house 的时候，在字符串起始处立即就会找到匹配 Mary。

当你把同一个正则表达式应用到余下的字符串的时候，比如说，你可以在文本编辑器中单击“查找下一个”，正则引擎就会尝试在该字符串中的第一个逗号处匹配 `<Mary>`。匹配会失败。然后，它会在同一个位置尝试去匹配 `<Jane>`，这也会失败。接着在逗号处匹配 `<Mary>`，当然也会失败。只有在所有匹配都失败之后，正则引擎才会前进到字符串中的下一个字符。从第一个空格开始匹配，3个选择分支都会得到同样的失败结果。

从字母 J 开始匹配的时候，第一个选择分支，`<Mary>`，会出现匹配失败。接着第二个选择分支，也就是 `<Jane>`，会在字母 J 处匹配成功。它匹配的是 Jane。正则引擎宣布匹配成功。

需要注意的是：虽然在目标文本中还存在另外一个 Mary，而且在正则表达式中 `<Mary>` 出现在 `<Jane>` 之前，但是这里匹配到的依然是 Jane。至少在这个例子中，在正则表达式中的顺序并不重要。正则表达式会查找最左匹配。它会从左向右扫描目标文本，在扫描的每一步中都会尝试正则表达式中的所有选择分支，而当其中任意一个选择分支产生一个合法匹配的时候，匹配过程就会停在这个位置。

如果我们再次对字符串余下部分进行查找，那么会找到 Sue。而第四次查找则会再一次找到 Mary。如果你告诉正则引擎进行第五次查找，那么就会失败，因为这三个选择分支都无法匹配余下的字串。

<sup>1</sup> 另外一种引擎是文本制导（text-directed）的引擎。二者之间的关键区别是文本制导的引擎只会访问目标文本中的每个字符一次，而正则制导的引擎则会访问每个字符多次。文本制导的引擎速度更快，但是只对我们在第 1 章中开始处介绍的数学意义上的正则表达式提供支持。而使得本书趣味盎然的 Perl 风格的正则表达式只能使用正则制导的引擎来实现。

只有当在字符串中的同一个位置存在两个选择分支同时匹配的时候，正则式中的选择分支的顺序才有意义。例如，正则式 `<Jane|Janet>` 在匹配目标文本 `Her name is Janet` 的时候，就会有两个选择分支在同一位置出现匹配。在正则表达式中并不存在单词边界的概念。事实上，`<Jane>` 完全也可以只匹配到 `Her name is Janet` 中的单词 `Janet` 的一部分。

`<Jane|Janet>` 之所以会匹配到 `Her name is Janet` 中的单词 `Janet`，是因为一个正则制导的正则表达式引擎是积极的（eager）。除了会从左向右扫描目标文本，查找最左匹配之外，它还会从左向右扫描正则式中的选择分支。而一旦它找到一个匹配的选择分支，正则引擎就会立即停止。

当 `<Jane|Janet>` 到达了 `Her name is Janet` 中的 J 的时候，第一个选择分支，`<Jane>`，发生了匹配。第二个选择分支则根本没有进行尝试。如果我们告诉引擎接着查找下一个匹配的话，这时候在目标文本中剩下的只有 t。此时两个选择分支都不会产生匹配。

要想不让 Jane 抢夺 Janet 的光环的话，可以有两种方式。第一种方式是把较长的选择分支放在前面：`<Janet|Jane>`。另外一种更为可靠的方式是清晰地表达我们所期望的结果：我们在查找名字，而名字应该是完整的单词。正则表达式并不会处理单词，但是它们可以处理单词边界。

因此，`\bJane\b|\bJanet\b` 和 `\bJanet\b|\bJane\b` 都会匹配到 `Her name is Janet` 中的 `Janet`。由于单词边界的原因，只有一个选择分支会产生匹配。在这里我们看到选择分支的顺序依然无关紧要。

实例 2.12 会讲解本题目的最佳答案：`\bJanet?\b`。

## 参见

实例 2.9。

## 2.9 分组和捕获匹配中的子串

### 问题描述

改进匹配 Mary、Jane 或 Sue 的正则表达式，使之只能匹配整个单词。使用分组来实现这个功能，整个正则表达式只需要一对单词分界符，而不是给每个选择分支都使用一对分界符。

创建一个正则表达式，使之匹配任意 yyyy-mm-dd 格式的日期，并且分别捕获年、月和日。目标是在处理匹配的代码中可以更容易处理这些分别捕获的值。你可以假设目标文本中的所有日期都是合法的。正则表达式不必要考虑去掉像 `9999-99-99` 这样的非法数据，因为它们根本不可能出现在目标文本中。

## 解决方案

\b(Mary|Jane|Sue)\b

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

\b(\d\d\d\d)-(\d\d)-(\d\d)\b

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

在上一节中介绍过的多选结构操作符（alternation operator）在所有正则操作符中拥有最低的优先级。如果你写的是 `\bMary|Jane|Sue\b`，那么三个选择分支分别是 `\bMary`、`Jane` 和 `Sue\b`。这个正则式会匹配在 `Her name is Janet` 中的 `Jane`。

如果想要正则表达式中的一些内容不受替代操作影响的话，那么你就需要把这些选择分支进行分组。分组是通过圆括号来实现的。它们拥有在所有正则操作符中的最高优先级，这与绝大多数编程语言都是一致的。`\b(Mary|Jane|Sue)\b` 拥有三个选择分支 `(Mary)`、`(Jane)` 和 `(Sue)`——它们都位于两个单词分界符之间。这个正则式在 `Her name is Janet` 中不会产生任何匹配。

当正则引擎到达目标文本中的 `Janet` 的 `J` 位置时，会匹配第一个单词分界符，接着该引擎就会进入这个分组。分组中的第一个选择分支，`(Mary)`，匹配失败。第二个选择分支，`(Jane)`，则匹配成功。然后引擎就会退出该分组。接下来需要匹配的就只剩下了 `\b`。然而单词分界符无法匹配这里的 `e` 与目标结尾处的 `t` 之间的位置。在 `J` 位置开始的匹配，其最后的结果是失败。

一组圆括号不仅仅是一个分组；它还是一个捕获分组（capturing group）。对于前面的 `Mary-Jane-Sue` 的正则表达式来说，捕获并不是很有用，因为它只是简单地匹配这个正则表达式。当需要覆盖正则表达式的子串的时候，捕获才会有用，就像在正则式 `\b(\d\d\d\d)-(\d\d)-(\d\d)\b` 中。

这个正则表达式匹配一个 `yyyy-mm-dd` 格式的日期。正则表达式 `\b\d\d\d\d\d\d-\d\d-\d\d\b` 也会匹配完全一样的内容。因为这个正则式没有使用任何替代或者重复，因此圆括号的分组功能就没必要存在。但是捕获功能用起来是很方便的。

正则表达式 `\b(\d\d\d\d)-(\d\d)-(\d\d)\b` 拥有三个捕获分组。分组是按照左括号的顺序从左向右进行编号的。`(\d\d\d\d)` 是 1 号分组。`(\d\d)` 是 2 号。第二个 `(\d\d)` 是 3 号分组。

在匹配过程中，当正则表达式引擎到达右括号而退出分组的时候，它会把该捕获分组所匹配的文本的子串存储起来。当我们的正则式匹配 `2008-05-24` 的时候，`2008` 被存储到第一个捕获中，`05` 在第 2 个捕获，而 `24` 则在第 3 个捕获中。

使用捕获的文本可以有 3 种方式。本章中的实例 2.10 会讲解如何在同一个正则匹配中

在此匹配所捕获的文本。实例 2.21 会展示在执行查找和替换的时候，如何把捕获到的文本添加到替代文本中。下一章中的实例 3.9 会介绍在你的应用程序中如何使用正则匹配的子串。

## 变体

### 非捕获分组

在正则式 `\b(Mary|Jane|Sue)\b` 中，我们使用圆括号只是为了分组的目的。与其使用一个捕获分组，我们也可以使用非捕获分组：

```
\b(?:Mary|Jane|Sue)\b  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

由三个字符 `<?>` 作为起始的是一个非捕获分组。右括号 `>` 则作为该分组的结束。非捕获分组提供相同的分组功能，但是并不会捕获任何内容。

当计算捕获分组的左括号个数来确定它们的序号的时候，不要计算非捕获分组的括号。这也是非捕获分组的主要好处：你可以把它们添加到一个已有的正则表达式中，而不会破坏对于已经编号的捕获分组的引用。

非捕获分组的另外一个好处是它的性能较高。如果你不会使用到某个特定分组的反向引用（实例 2.10），需要把它重新插入到替代文本中（实例 2.21），或是在源代码中提取其匹配（实例 3.9）的话，那么使用捕获分组就会添加额外的性能开销，这可以通过使用非捕获分组来消除。在实践中，你可能很少会注意到性能上的差异，除非在一个紧缩循环（tight loop）中使用正则表达式，而且/或者使用它来处理大量数据。

### 带模式修饰符的分组

在实例 2.1 的变体中，我们解释了.NET、Java、PCRE、Perl 和 Ruby 支持局部的模式修饰符，其中使用了模式切换：`<sensitive(?i)caseless(?-i) sensitive>`。虽然这种语法也会涉及到圆括号，但像 `<(?i)>` 这样的切换并不会涉及任何分组。

如果不用切换，你也可以在一个非捕获分组中来指定模式修饰符：

```
\b(?:Mary|Jane|Sue)\b  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Ruby  
  
sensitive(?i:caseless)sensitive  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Ruby
```

在一个非捕获分组中添加模式修饰符会对在该分组中的正则表达式的子串设置该模式。而在右括号之后则会恢复之前的设置。因为默认来说正则表达式是区分大小写的，

因此只有在(?i:...)之内的正则表达式部分才是不区分大小写的。

还可以把多个修饰符组合起来，例如 «(?ism:group)»。使用一个连字号来关闭修饰符：«(?-ism:group)» 会关掉这 3 个选项。«(?i-sm)» 会打开不区分大小写 (i)，并且同时关闭“点号匹配换行符”(s) 和“^和\$匹配换行处”(m)。这些选项在实例 2.4 和实例 2.5 中已经进行了讲解。

## 参见

实例 2.10、2.11、2.21 和 3.9。

## 2.10 再次匹配先前匹配的文本

### 问题描述

创建一个正则表达式来匹配按照 yyyy-mm-dd 格式的“神奇”日期。一个神奇日期指的是如果年减去世纪、月份和该月的天数都是相同的数字。例如，2008-08-08 就是一个神奇日期。你可以假设目标文本中的所有日期都是有效的。这个正则表达式并不需要考虑去掉像 9999-99-99 这样的日期，因为它们不会出现在目标文本中。你只需要找到神奇的日期即可。

### 解决方案

\b\d\d(\d\d)-\1-\1\b

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

为了在一个正则表达式中匹配先前匹配到的文本，我们首先必须捕获上次匹配的文本。这可以使用一个捕获分组来实现，就像在实例 2.9 中所讲解的。在此之后，我们可以使用反向引用 (backreference) 来在该正则表达式中的任何地方匹配相同的文本。你可以使用一个反斜杠之后跟一个单个数字 (1~9) 来引用前 9 个捕获分组。而第 10~99 组，则要使用 «\10»~«\99»。



#### 警告

不要使用 «\01»。它或者是一个八进制的转义，或者会产生一个错误。在本书中我们不会用到八进制转义，因为 «\xFF» 这样的十六进制转义更加容易理解。

当正则表达式 «\b\d\d(\d\d)-\1-\1\b» 遇到 2008-08-08 的时候，第一个 «\d\d» 会匹配 20。接着根据目标文本中到达的位置，正则引擎会进入捕获分组。

在捕获分组中的 «\d\d» 会匹配 08，然后引擎会到达分组的右括号。在这个点上，部分

匹配 08 会被保存到 1 号捕获分组中。下一个记号是连字符，它会按照字面进行匹配。接着就遇到了反向引用。正则引擎会检查第一个捕获分组的内容：08。然后引擎会试图按照字面来匹配这个文本。如果该正则表达式是不区分大小写的，那么捕获分组也会按照这种方式进行匹配。在这里，反向引用会匹配成功。下一个连字符和反向引用也会匹配成功。最终，单词分界符会匹配目标文本的结尾，这样就找到了一个整体匹配：2008-08-08。现在捕获分组中依然保存的是 08。

如果一个捕获分组被重复的话，这可以通过一个量词（实例 2.12）或者是回溯（实例 2.13）来实现，保存的匹配会在每次捕获分组匹配成功的时候被覆盖。一个对该分组的反向引用只会匹配该分组最后一次捕获到的文本。

如果同一个正则表达式遇到 2008-05-24 2007-07-07 的时候，当 `\b\d\d(\d\d)` 匹配到 2008 的时候，该分组第一次捕获到的内容 08，会被保存到第一个（也是唯一一个）捕获分组中。接下来，连字符会匹配它自身。反向引用在试图用 `\1` 来匹配 08 的时候，匹配会产生失败。

由于在该正则表达式中不存在其他的选择分支，引擎会放弃匹配尝试。这包括清除所有的捕获分组。当引擎再次尝试的时候，从目标文本中的第一个 0 开始，`\1` 不再存有任何文本内容。

接下来继续处理 2008-05-24 2007-07-07，该分组下一次会捕获到内容是当 `\b\d\d(\d\d)` 匹配到 2007 的时候，它会把 07 保存起来。接下来，连字符匹配自身。现在反向引用会试图匹配 07。这次匹配是成功的，接着下一个连字符、反向引用以及单词边界都匹配成功。结果是找到了 2007-07-07。

因为正则引擎是从前向后处理的，因此应当把捕获括号放到反向引用的前面。正则表达式 `\b\d\d\1-(\d\d)\1` 和 `\b\d\d\1-\1-(\d\d)\b` 永远不可能匹配到任何东西。因为这里的反向引用是在捕获分组之前出现的，因此它还没有捕获到任何东西。除非你使用的是 JavaScript，那么如果反向引用指向一个还没有进行匹配尝试的分组，它总是会匹配失败。

一个还没有参与匹配的分组，并不等同于一个捕获到长度为 0 匹配的分组。对一个长度为 0 的捕获分组的反向引用总是会匹配成功。当 `\1` 匹配字符串的开始的时候，第一个捕获分组会捕获到 ^ 号的长度为 0 的匹配，从而 `\1` 会匹配成功。在实践中，这会发生在当所有捕获分组的内容都不是必需的情况下。



### 提示

JavaScript 是我们所知道的唯一与正则表达式中几十年反向引用的传统相违背的流派。在 JavaScript 中，或者至少在遵循 JavaScript 标准的实现中，对一个还没有参与匹配的分组的反向引用总是会匹配成功，这同捕获了长度为 0 的匹配的分组的反向引用是一样的。因此，在 JavaScript 中，`\b\d\d\1-\1-(\d\d)\b` 会成功匹配 12--34。

## 参见

实例 2.9、2.11、2.21 和 3.9。

## 2.11 捕获和命名匹配子串

### 问题描述

创建一个正则表达式，匹配以 yyyy-mm-dd 格式的任意日期，并且分别捕获年、月和日。这样做的目的是为了在处理匹配的代码中可以更容易处理这些分别捕获的值。为了更好地实现这个目标，向每个捕获的文本添加描述性的名称：“year”、“month” 和 “day”。

再创建一个正则表达式来匹配按照 yyyy-mm-dd 格式的“神奇”日期。一个神奇日期指的是如果年减去世纪、月份和该月的天数都是相同的数字。例如，2008-8-8 就是一个神奇日期。把神奇日期捕获下来（在这个例子中是 08），并把给它打上标签“magic”。

你可以假设目标文本中的所有日期都是合法的。正则表达式不必考虑去掉像 9999-99-99 这样的非法数据，因为它们根本不可能出现在目标文本中。

### 解决方案

#### 命名捕获

```
\b(?:<year>\d\d\d\d)-(?:<month>\d\d)-(?:<day>\d\d)\b
```

正则选项：无

正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

```
\b(?:'year'\d\d\d\d)-(?:'month'\d\d)-(?:'day'\d\d)\b
```

正则选项：无

正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

```
\b(?:<year>\d\d\d\d)-(?:<month>\d\d)-(?:<day>\d\d)\b
```

正则选项：无

正则流派：PCRE 4 或更新版本、Perl 5.10、Python

#### 命名反向引用

```
\b\d\d(?<magic>\d\d)-\k<magic>-\k<magic>\b
```

正则选项：无

正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

```
\b\d\d(?'magic'\d\d)-\k'magic'-\k'magic'\b
```

正则选项：无

正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

```
\b\d\d(?P<magic>\d\d)-(?P=magic)-(?P=magic)\b
```

正则选项：无

正则流派：PCRE 4 或更新版本、Perl 5.10、Python



## 讨论

### 命名捕获

实例 2.9 和 2.10 讲解了捕获分组和反向引用。更加准确地来讲：这两个实例中使用的是编号的捕获分组和编号的反向引用。每个分组会自动获得一个编号，你可以使用这些编号来进行反向引用。

除了编号分组之外，现代的正则表达式流派还支持命名捕获分组。在命名和编号分组之间的唯一区别是你可以给分组指派一个描述性的名称，而不是被限制为只能使用自动分配的编号。命名分组可以使正则表达式更加容易阅读，更加容易维护。把一个捕获分组添加到一个已有的正则表达式中可能会修改之前指派给所有捕获分组的编号。而你指派的名称则依然是保持不变的。

Python 是第一个支持命名捕获的正则表达式流派。它使用的语法是 `<(P<name>regex)>`。名称中包含的必须是可以被 `\w` 匹配的单词字符。`<(P<name>)` 是该分组的起始括号，而 `>` 则是该分组的结束括号。

.NET Regex 类的设计人员为命名捕获采用了他们自己的语法，他们使用了两种可以互换的变体。`<(<name>regex)>` 模仿了 Python 的语法，但是却去掉了其中的 P。这里的名称必须包含可以被 `\w` 匹配的单词字符。`<(<name>)` 是该分组的起始括号，而 `>` 则是分组的结束括号。

当你使用 XML 编码，或是像我们这样在 DocBook XML 中撰写本书的时候，在命名捕获语法中的尖括号会让人感到厌烦。这也正是.NET 中提供了另一种可替代的命名捕获语法的原因：`<('name'regex)>`。这里的尖括号被替换为了单引号。你可以出于自己的方便来选择使用哪种记法。它们的功能是完全等价的。

可能是基于.NET 的流行超过了 Python 的原因，.NET 语法似乎成为了其他正则库开发人员更乐于沿用的语法。Perl 5.10 使用的是它，而在 Ruby 1.9 的 Oniguruma 引擎中也是如此。

PCRE 在很早的时候就沿用了的 Python 的语法，在那个时候 Perl 还根本没有提供对于命名捕获的支持。PCRE 7 的版本中添加了 Perl 5.10 中添加的新功能，它同时提供对.NET 语法和 Python 语法的支持。可能是基于 PCRE 成功的考虑，Perl 5.10 采取了反向兼容的举动，也提供了对 Python 语法的支持。在 PCRE 和 Perl 5.10 中，用于命名捕获的.NET 语法和 Python 语法的功能是完全一样的。

读者应当选择对你来说最为有用的语言。如果在 PHP 中编码的话，你会希望代码能够用于老版本的 PHP，由于它采用的是 PCRE 的老版本，你应该使用 Python 语法。如果不希望与老版本的兼容性，而且也采用的是.NET 和 Ruby，那么在这些语言之间进行复制和粘贴的话，采用.NET 语法更为容易。如果你不是很确定，那么可以使用 PHP/PCRE

中的 Python 语法。如果有人把你的代码在老版本的 PCRE 下进行了重新编译，而代码中的正则表达式突然就不工作了，那么他们肯定会很不高兴。当你把一个正则表达式从.NET 复制到 Ruby 中时，删掉一些 P，应该不是太困难。

PCRE 7 和 Perl 5.10 的文档中几乎没有提到 Python 语法，但是它肯定不是要被淘汰的。正相反，我们实际上推荐在 PCRE 和 PHP 使用 Python 语法。

## 命名反向引用

有了命名捕获之后，紧接着就有了命名反向引用。正如命名捕获分组与编号捕获分组的功能完全相同一样，命名反向引用与编号反向引用的功能也是完全相同的。它们只是更加易于阅读和维护。Python 使用语法 `\g<name>` 来创建对一个分组 name 的反向引用。虽然该语法使用的是圆括号，但反向引用并不是一个分组。你不能在名称和结束括号之间放任何东西。一个反向引用 `\g<name>` 是单个的正则标记，就像是 `\1` 一样。

.NET 使用的语法是 `\k<name>` 和 `\k'<name>`。这两个变体在功能上是完全相同的，因此也可以随意混着使用。使用尖括号语法创建的命名分组可以采用引号语法来进行引用，反之亦然。

我们强烈推荐你不要在同一个正则表达式中混合使用命名和编号分组。不同流派对于出现在命名分组之间的非命名分组的编号方法会遵循不同的规则。Perl 5.10 和 Ruby 1.9 沿用了.NET 的语法，但是它们并没有遵循.NET 对于命名捕获分组，或者是混合使用编号捕获分组与命名分组进行编号的方式。与其在这里解释其中的差别，我们选择推荐大家不要把命名和编号分组混合使用。应当避开可能的混淆，给所有未命名的分组一个名称，或者把它们变成非捕获的。

## 参见

实例 2.9、2.10、2.21 和 3.9。

## 2.12 把正则表达式的一部分重复多次

### 问题描述

创建正则表达式来匹配下列种类的数字。

- 一个 googol (一个 100 位的十进制整数)。
- 一个 32 位的十六进制整数。
- 一个 32 位的十六进制整数，带有一个可选的 h 后缀。
- 一个浮点数，包含可选的整数部分、必需的小数部分和可选的指数部分。每个部分都允许任意多个数字。

# 解决方案

## Googol

```
\b\d{100}\b  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 十六进制整数

```
\b[a-f0-9]{1,8}\b  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 带可选后缀的十六进制整数

```
\b[a-f0-9]{1,8}h?\b  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 浮点数

```
\b\d*\.\d+(e\d+)?  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 讨论

### 固定次数重复

量词 (quantifier) `\{n\}`，其中 n 是一个正整数，用来重复之前的正则记号 n 次。在 `\b\d{100}\b` 中的 `\d{100}` 会匹配一个包含 100 个数字的字符串。你也可以通过把 `\d` 敲 100 遍来达到同样的效果。

`\{1\}` 重复之前的记号一次，这样和没有任何量词是等价的。`ab\{1\}c` 与 `abc` 是同样的正则表达式。

`\{0\}` 重复之前的记号 0 次，实际上也就是把它从正则表达式中删除。`ab\{0\}c` 与 `ac` 是同样的正则表达式。

### 可变次数重复

对于可变次数重复，我们使用量词 `\{n,m\}`，其中 n 是一个正整数，并且 m 大于 n。`\b[a-f0-9]{1,8}\b` 匹配一个包含 1~9 个数字的十六进制整数。对于可变次数重复的情形，其中所有选择分支重复的顺序就会产生影响。这会在实例 2.13 中详细加以讲解。

如果 n 和 m 是相等的，那么我们就会得到固定次数的重复。`\b\d{100,100}\b` 与 `\b\d{100}\b` 是同样的正则表达式。

## 无限次数重复

量词 `\{n\}`，其中  $n$  是一个正整数，支持无限次数重复。实际上，无限次数重复是一个没有上限的可变次数重复。

`\d\{1\}` 匹配一个或多个数字，`\d+` 也一样。在一个不是量词的正则记号之后添加一个加号，意味着“一次或多次”。实例 2.13 会讲解在量词之后跟加号的含义。

`\d\{0\}` 匹配 0 个或多个数字，`\d*` 也一样。星号永远都意味着“0 次或多次”。除了支持无限次数重复之外，`\{0\}` 和星号还把之前的记号变成了可选的。

## 把一些内容变成可选

如果我们使用可变次数重复，并把  $n$  设置为 0，那么我们事实上就是把在该量词之前的记号变成了可选的。`\h\{0,1\}` 匹配一次或者根本不存在。如果不存在 `\h` 的话，那么 `\h\{0,1\}` 会得到一个长度为 0 的匹配。如果你使用 `\h\{0,1\}` 自己来作为一个正则表达式，那么它会在目标文本中每个不是 `\h` 的字符之前找到一个长度为 0 的匹配。每个 `\h` 则会得到包含一个字符（也就是 `\h`）的匹配。

`\h?` 与 `\h\{0,1\}` 的效果是一样的。在一个合法和完整的非量词正则记号之后的问号意味着“0 或 1 次”。下一个实例会解释在量词之后的问号的含义。



### 提示

在一个起始括号之后使用问号或者任意其他量词，都是一个语法错误。

Perl 和沿用它的流派使用它向正则语法中添加“Perl 扩展”。前面的实例讲解了非捕获分组和命名捕获分组，它们都使用了在起始括号之后的问号来作为其语法的一部分。这些问号根本不是量词；它们只是属于非捕获分组和命名捕获分组的语法中的一部分。随后的实例中会讲解使用 `\(?>` 语法的更多分组风格。

## 重复分组

如果在分组的结束括号之后放一个量词的话，那么整个分组就会被重复。`\(?:abc\){3}` 与 `\abc\abc\abc` 是相同的。

量词还可以进行嵌套。`\(e\d+)\?>` 匹配一个 `e` 之后跟着一个或多个数字，或者是一个长度为 0 的匹配。在我们的浮点数的正则表达式中，这是可选的指数部分。

捕获分组也可以重复。在实例 2.9 中解释过，分组的匹配在每次引擎退出该分组的时候被捕获，并会覆盖该分组在之前匹配的任何文本。`\(\d\d\){1,3}` 会匹配一个包含 2 个、4 个和 6 个数字的字符串。引擎会退出该分组 3 次。当这个正则表达式匹配到 123456 的时候，捕获分组中保存的是 56，因为该分组的最后一次循环存储的是 56。另外两次分组匹配，也就是 12 和 34，是无法被获取的。

`\d{3}` 会与 `\d\d\d(\d\d)` 捕获相同的文本。如果想要用捕获分组来捕获所有两个、四个或六个字符，而不只是最后两个，那么就必须把捕获分组和量词一起使用，而不是只是重复该捕获分组：`((?:\d\d){1,3})`。这里我们用了一个非捕获分组取代了捕获分组的分组功能。我们也可以使用两个捕获分组：`((\d\d){1,3})`。当这最后一个正则表达式匹配 123456 的时候，`\1` 中保存的是 123456，而在 `\2` 中则保存了 56。

只有.NET 的正则表达式引擎才支持获取一个重复捕获分组的所有循环。如果直接查找该分组的 `Value` 属性，它会返回一个字符串，而你会得到 56，就像所有其他正则表达式引擎一样。在正则表达式中的反向引用和替代文本也会替代 56，但是如果使用分组的 `CaptureCollection`，那么你就会得到一个栈，其中包含 56、34 和 12。

## 参见

实例 2.9、2.13、2.14。

## 2.13 选择最小和最大重复次数

### 问题描述

匹配一对 XHTML 标记`<p>`和`</p>`，以及二者之间的所有文本。在标记之间的文本也可以包含其他 XHTML 标记。

### 解决方案

```
<p>.*?</p>
    正则选项：点号匹配换行符
    正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### 讨论

在实例 2.12 中讨论的所有量词都是贪心的 (greedy)，意味着它们会试图重复尽量多次，只有当剩余的正则表达式不能匹配的时候才会返回结果。

这对于把 XHTML（它是 XML 的一个版本，因此要求每个起始标记都存在匹配的结束标记）中的标记进行配对来说，可能会出现困难。考虑如下一个简单的 XHTML 片段：

```
<p>
The very <em>first</em> task is to find the beginning of a paragraph.
</p>
<p>
Then you have to find the end of the paragraph
</p>
```

在该断码片段中，存在两个起始`<p>`标记和两个结束`</p>`标记。你想要把第一个`<p>`与第一个`</p>`进行匹配，因为它们标记了一个段落。注意这个段落还包括了一个`<em>`标

记，因此该正则表达式不能是遇到<符号的时候就简单地停止。

我们来看一下下面这个不正确的解答：

```
<p>.*</p>
```

正则选项：点号匹配换行符

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

这个错误解答中唯一的区别是它缺少了在星号之后的额外问号。这个不正确的答案中使用了在实例 2.12 中讲解的同样的贪心星号。

在匹配了目标文本中的第一个<p>之后，引擎会到达<.\*>。其中的点号可以匹配任意字符，其中也包括换行符。星号则把它重复 0 次或更多次。这里的星号是贪心的，因此<.\*>会匹配直到目标文本结束的所有内容。需要再重申一遍：<.\*>会吃掉你的整个 XHTML 文件，从第一段开始。

当<.\*>把肚子吃饱之后，引擎才会试图去匹配在目标文本末尾的<>。这显然会失败。但是这还没完：正则引擎会进行回溯（backtrack）。

星号更喜欢夺取尽可能多的文本，但是它对于不匹配任何东西（0 次重复）同样也会感到完全满意。在超过量词最小次数的每次量词重复之后，正则表达式都会保存一个回溯位置。如果在该量词之后的正则表达式部分匹配失败，那么正则引擎可以回到这些位置。

当<>失败之后，引擎会进行回溯，让<.\*>放弃它的匹配中的一个字符。接着<>会被再次尝试匹配，这次在文件中最后一个字符的位置。如果它依然失败的话，那么引擎会再一次进行回溯，在文件的倒数第二个字符处尝试匹配<>。这个过程会一直继续，直到<>匹配成功为止。如果<>一直没有匹配成功，那么最终<.\*>会用完所有的回溯位置，然后整个匹配宣布失败。

如果在整个回溯的过程中，<>的确在某个点上匹配到了。那么就会接着尝试匹配</>。如果</>匹配失败的话，引擎会接着进行回溯。这个过程会一直重复，直到</p>>可以被整个匹配为止。

那么问题在哪里呢？因为星号是贪心的，所以上面给出的不正确的正则表达式会匹配在 XHTML 文件中的第一个<p>到最后一个</p>之间的所有内容。但是要想正确地匹配一个 XHTML 段落，我们需要的是匹配第一个<p>与跟在其后的第一个</p>。

这个时候，我们就需要使用懒惰（lazy）量词了。你可以在其后放一个问号来使任何量词变成懒惰的：<\*?>、<+?>、<??> 和 <{7,42}?> 都是懒惰量词。

懒惰量词也会进行回溯，但却是从不同的方向进行的。一个懒惰量词会重复尽可能少的次数，然后保存一个回溯位置，并且允许正则表达式继续。如果剩余的正则表达式匹配失败了，那么引擎会进行回溯，此时懒惰量词会再重复一次。如果正则表达式持续回溯，那么量词会扩展直到它允许的最大重复次数，或者直到它所重复的正则记号

匹配失败。

`<p>.*?</p>` 使用了一个懒惰量词来正确地匹配一个 XHTML 段落。当 `<p>` 匹配成功的时候，`.*?` 作为懒惰量词，最初什么也不做，只是稍作停顿。如果 `</p>` 在 `<p>` 之后立即出现，那么就会匹配一个空段落。如果不是这样，那么引擎会回溯到 `.*?`，这次会匹配一个字符。如果 `</p>` 还是匹配失败，`.*?` 会接着匹配下一个字符。这个过程会继续进行下去，直到 `</p>` 匹配成功，或者 `.*?` 扩展失败。因为点号会匹配任意字符，所以直到匹配完了所有内容，到达 XHTML 文件结束的时候，`.*?` 才会宣布匹配失败。

量词 `*` 和 `*?` 允许所有相同的正则表达式匹配。唯一的区别是这些可能匹配被尝试的顺序不同。贪心量词会找到最长可能的匹配。懒惰量词则会找到最短可能的匹配。

如果可能的话，最佳解决方案是确保只存在一个可能的匹配。在实例 2.12 中，如果你把所有量词都变成懒惰的，用来匹配数字的正则表达式还会匹配相同的数字。原因是这些正则表达式中拥有量词的部分和紧跟其后的部分是互斥的。`\d` 匹配一个数字，而只有当下一个字符不是数字（或字符）的时候 `\b` 才会匹配 `\d` 之后的位置。

为了有助于更好地理解贪心和懒惰量词重复的操作过程，我们可以比较一下 `\d+\b` 和 `\d+?\b` 在几个不同目标文本之上的表现。贪心和懒惰版本会产生相同的结果，但是却会按照不同的顺序来检查目标文本。

如果我们使用 `\d+\b` 来匹配 1234，`\d+` 那么会匹配所有的数字。接着 `\b` 匹配，然后就会找到一个完整匹配。如果我们使用 `\d+?\b`，那么 `\d+?` 首先只会匹配 1。`\b` 在 1 和 2 之间匹配失败。`\d+?` 会扩展到 12，但是 `\b` 还是会失败。这将一直继续，直到 `\d+?` 匹配 1234，`\b` 才会匹配成功。

如果我们的目标文本是 1234X，第一个正则式，`\d+\b`，依然会让 `\d+` 先匹配 1234。但是接着 `\b` 会匹配失败。`\d+` 回溯到 123。`\b` 还是会匹配失败。这会继续下去，直到 `\d+` 回溯到最小可能的 1，`\b` 还是会匹配失败。这样整个匹配尝试就会宣告失败。

如果我们使用 `\d+?\b` 来匹配 1234X，那么 `\d+?` 首先只会匹配 1。`\b` 在 1 和 2 之间匹配失败。`\d+?` 会扩展到 12，但是 `\b` 还是会失败。这将一直继续，直到 `\d+?` 匹配 1234，`\b` 还是匹配失败。正则引擎会试图再一次扩展 `\d+?`，但是 `\d` 无法匹配 X。整个匹配尝试就会宣告失败。

如果我们把 `\d+` 放到单词边界之中，那么它必须匹配在目标文本中的所有数字，否则它就会匹配失败。把量词变成懒惰的，并不会影响最后的正则匹配最终是成功还是失败。事实上，`\b\d+\b` 不用任何回溯会更好。下一个实例会解释如何可以使用一个占有量词 `\b\d++\b` 来达到这个目标，至少在某些流派中是可行的。

## 参见

实例 2.8、2.9、2.12、2.14 和 2.15

## 2.14 消除不必要的回溯

### 问题描述

上一个实例解释了贪心和懒惰量词之间的区别，以及它们是如何进行回溯的。在有些情形下，这种回溯是不必要的。

`\b\d+\b` 使用了一个贪心量词，而 `\b\d+?\b` 使用的是懒惰量词。它们都会匹配相同的内容：一个整数。如果给它们同样的目标文本，它们都会找到完全一样的匹配。在这里所做的任何回溯都是不必要的。试着改写这个正则表达式，明确地消除所有回溯，使正则表达式更加高效。

### 解决方案

`\b\d++\b`

正则选项：无

正则流派：Java、PCRE、Perl 5.10、Ruby 1.9

最容易的解决方案是使用一个占有量词。但是它只在几种最近的正则流派中才提供支持。

`\b(?:\d+)\b`

正则选项：无

正则流派：.NET、Java、PCRE、Perl、Ruby

一个原子分组可以提供完全一样的功能，但是需要使用稍微不是那么易读的语法。对原子分组的支持相比占有量词来说更为广泛。

JavaScript 和 Python 都不支持占有量词或原子分组。因此在这两种正则流派中无法消除不必要的回溯。

### 讨论

占有量词（possessive quantifier）与贪心量词是类似的：它也会试图去重复尽可能多的次数。它们之间的区别是占有量词永远不会回退，即使在回退是能够匹配正则表达式剩余部分的唯一手段的时候也不会这样做。占有量词也不会记录任何可能的回溯位置。

可以通过在一个量词之后添加一个加号来把它变成占有量词。例如，`*+`、`++`、`?+` 和 `{7,42}+` 都是占有量词。

在 Java 4 或者更新版本中，也就是在 Java 发布中包含了 `java.util.regex` 包之后，就提供了对占有量词的支持。本书中介绍的 PCRE 的所有版本（版本 4~7）都支持占有量词。Perl 从 5.10 版本也开始支持它们。经典的 Ruby 正则表达式不支持占有量词，但是 Oniguruma 引擎，也就是 Ruby 1.9 中使用的默认引擎是支持占有量词的。

在原子分组（atomic group）外面包一个贪心量词同使用一个占有量词会产生完全相同

的效果。当正则引擎退出原子分组的时候，量词会记住所有的回溯位置，并且分组中的可选项都会被丢弃。所使用的语法是 `\b(?>regex)\b`，其中`regex`可以是任意正则表达式。一个原子分组本质上是一个非捕获分组，加上拒绝回溯的功能。这里的问号不是量词；原子分组的起始括号中包括了三个字符 `\b(?>)`。

当把正则表达式 `\b\d++\b`（占有版本）应用到 `123abc 456` 的时候，`\b` 会匹配目标文本的开始，`\d++` 则会匹配 `123`。到目前为止，这和 `\b\d+\b`（贪心版本）所做的并没有区别。但是接着第二个`\b`会在 3 和 a 之前匹配失败。

占有量词不会保存任何回溯位置。既然在这个正则表达式中不存在其他量词或者选择，因此在第二个单词边界匹配失败的时候，就没有其他选择了。正则引擎会立即宣布从 1 开始的匹配失败。

正则引擎还会在字符串中的下一个字符位置尝试匹配该正则表达式，使用占有量词并不会改变这一点。如果这一正则表达式必须匹配整个目标文本，那么就需要使用定位符，这在实例 2.5 中已经讲解过。最终，正则引擎会从 4 开始的位置尝试匹配该正则表达式，并且找到了匹配 `456`。

使用贪心量词的区别是当在第一次匹配第二个`\b`的尝试失败之后，贪心量词会开始回溯。正则引擎接着会（没必要地）在 2 和 3 之间，以及 1 和 2 之间检查`\b`。

使用原子分组的匹配过程本质上是一样的。当你把正则表达式 `\b(?>\d+)\b`（占有版本）应用到 `123abc 456` 之上时，在目标文本的开始处会匹配单词边界。接着正则引擎会进入原子分组，`\d+` 会匹配 `123`。现在引擎退出原子分组。在这一点上，由`\d+` 所记住的回溯位置都被丢弃了。当第二个`\b`失败的时候，正则引擎没有其他选择，只能离开，因此会导致这次匹配尝试立即失败。与占有量词的版本一样，最终 `456` 会被找到。

在我们的解释中，占有量词不会去记住回溯位置，而原子分组则会把回溯位置丢弃。这样会更容易理解匹配过程，但是读者也不要太在意这里的区别，因为很可能在你所使用的正则流派中根本不存在这样的区别。在许多流派中，`x++` 仅仅是 `\b(?>x+)\b` 语法上的简写，而二者的实现则完全是一模一样的。至于引擎是从来没有记住回溯位置，还是说它稍后会把这些位置丢弃，对于匹配尝试的最后结果来说是根本无关紧要的。

占有量词和原子分组的不同之处是占有量词只应用于单个正则表达式记号，而原子分组则可以把整个正则表达式包起来。

`\w++\d++` 和 `\b(?>\w+\d+)\b` 是完全不一样的。`\w++\d++` 与 `\b(?>\w+)(?>\d+)\b` 则是一样的，二者都无法匹配 `abc123`。`\w++` 可以整个匹配 `abc123`。然后，正则引擎会在目标文本的结尾处试图匹配`\d++`。因为现在并不存在任何可以匹配的多余字符，所以`\d++`会匹配失败。如果不存在任何记住的回溯位置的话，匹配尝试就会失败。

`\b(?>\w+\d+)\b` 在同一个原子分组中拥有两个贪心量词。在这个原子分组中，回溯会正常发生。回溯的位置只有当引擎退出整个分组的时候才会被丢弃。当目标文本是 `abc123`

的时候，`\w+` 会匹配 `abc123`。贪心量词则会记住回溯的位置。当 `\d+` 匹配失败的时候，`\w+` 会主动放弃一个字符。这样 `\d+` 接着就可以匹配 3。现在，引擎会退出这个原子分组，并且丢弃掉为 `\w+` 和 `\d+` 记住的所有回溯位置。因为此时正则表达式已经到达了结尾，所以这并不会造成任何问题。结果是找到了整体匹配。

如果结尾还没有到达，像是在 `<(?>\w+\d+)\d+>` 中一样，我们就会遇到与 `\w++\d++` 一样的情形。在目标文本的结尾处，不存在任何内容可以匹配第二个 `\d+`。因为这时回溯位置已经被丢弃了，所以正则引擎只能宣布匹配失败。

占有量词和原子分组所做的不仅仅是对正则表达式进行优化。它们也可能会利用消除那些通过回溯可能会到达的匹配，而改变一个正则表达式最终找到的匹配。

本实例展示了如何使用占有量词和原子分组来进行一些较小的优化，而这些优化在实际例子中甚至可能不会表现出任何区别。下一个实例会讲解原子分组如何能够产生更加显著的影响。

## 参见

实例 2.12 和 2.15。

## 2.15 避免重复逃逸

### 问题描述

使用一个正则表达式来匹配一个完整的 HTML 文件，并检查其中的 `html`、`head`、`title` 和 `body` 标记是否进行了正确嵌套。当 HTML 文件中不拥有正确标记的时候，该正则表达式必须能够高效地宣布匹配失败。

### 解决方案

```
<html>(?>.*?<head>) (?>.*?<title>) (?>.*?</title>) ↪  
    (?>.*?</head>) (?>.*?<body[^>]*>) (?>.*?</body>).*?</html>  
正则选项：不区分大小写、点号匹配换行符  
正则流派：.NET、Java、PCRE、Perl、Ruby
```

JavaScript 和 Python 不支持原子分组。因此在这两种正则流派中无法消除不必要的回溯。当使用 JavaScript 或者 Python 编程时，可以通过对这些标记一一进行字面文本查找来解决这个问题，在找到一个标记之后，再在剩余的目标文本中查找下一个标记。

### 讨论

如果从下面这个最原始的解答入手，那么对这个问题的正确解答会更加容易理解：

```
<html>.*?<head>.*?<title>.*?</title>↪  
    .*?</head>.*?<body[^>]*>.*?</body>.*?</html>
```

正则选项：不区分大小写、点号匹配换行符

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

当你在一个正确的 HTML 文件上测试这个正则表达式的时候，它会完全正常地运行。`<.*?>` 会略过所有的内容，因为我们打开了“点号匹配换行符”的选项。惰性的星号量词会确保这个正则表达式一次只会前进一个字符，每次都会检查是否匹配到了下一个标记。实例 2.4 和 2.13 中已经讲解过这一切。

但是，当这个正则表达式需要处理并不包含所有 HTML 标记的目标文本的时候，你就会遇到困难。最坏的情形是当`</html>`缺失的时候。

想像一下正则引擎已经匹配了所有前面的标记，现在正在忙着扩展最后的`<.*?>`。因为`</html>`永远会产生匹配，所以`<.*?>`会一直扩展到文件的结尾。当它无法再进行扩展的时候，就会宣布匹配失败。

但是这还不是故事的结束。另外 6 个`<.*?>`都记住了一个回溯位置，因此会允许它们进一步扩展。当最后一个`<.*?>`匹配失败的时候，它前面的那个会进行扩展，逐步匹配`</body>`。同样的文本在前面会被正则表达式中的字面量`</body>`匹配。这个`<.*?>`也会一直扩展到文件的结尾，在它之前所有的`<.*?>`惰性点号量词也同样会这样做。只有当第一个到达了文件末尾的时候，正则引擎才会宣布匹配失败。

这个正则表达式拥有最坏情形的复杂度  $O(n^7)$ ，也就是目标文本长度的 7 次方。其中包括 7 个惰性点号量词，可能会一直扩展到文件的结尾。如果文件的大小增加一倍，正则表达式就可能需要 128 倍的步骤才能计算出它无法匹配。

我们把这种情形称作灾难性回溯（catastrophic backtracking）。由于出现了太多的回溯，所以正则表达式或者会无休止的匹配下去，或者会让你的应用程序死掉。一些正则表达式实现会比较聪明，可能会及早退出这种逃逸的匹配尝试，但是即使在这种情况下，正则表达式也还是会毁掉程序的性能。



### 提示

灾难回溯是一种被称作组合爆炸（combinatorial explosion）的现象的一种实例，其中几种正交条件交织起来，不得不尝试所有的组合。你也可以说正则表达式是不同重复操作符的笛卡尔乘积。

解决方案是使用原子分组来避免不必要的回溯。在`</body>`匹配成功之后，第 6 个`<.*?>`就没有必要进行扩展。如果`</html>`匹配失败的话，那么扩展第 6 个惰性点号也不可能神奇地变出一个终止的 html 标记。

当紧随其后的分界符匹配之后，要想使一个量词化的正则表达式记号停止扩展，就需要把正则表达式的量词部分与分界符一起放到一个原子分组中：`<(?:.*?</body>)>`。这样正则引擎就会在`</body>`被找到之后丢弃`<.*?</body>`所有的匹配位置。如果`</html>`随后匹配失败的话，那么正则引擎已经忘记了`<.*?</body>`的回溯位置，所

以不会进行任何扩展。

如果我们将正则表达式中所有其他的 `<.*?>` 都做同样的操作，那么它们就都不会再继续扩展。虽然在正则表达式中还是存在 7 个惰性点号，但是它们永远也不会产生交叉。这样就会把正则表达式的复杂度降低到  $O(n)$ ，这与目标文本的长度相比是线性的。而正则表达式的效率不可能比此更高了。

## 变体

如果你确实想知道灾难性回溯如何执行，那么可以在 `xxxxxxxxxx` 之上测试一下 `((x+x+)+y)`。如果它很快就会失败，那么在目标文本中再添加一个 `x`。重复这个过程，直到正则表达式开始要花费很长的时间来产生匹配，或者你的应用程序崩溃。除非你使用的是 Perl，否则并不需要添加太多的 `x` 字符就会出问题。

在本书中讨论的所有正则流派中，只有 Perl 有能力检测正则表达式太复杂的情形，并且会终止匹配尝试，而不会造成程序崩溃。

这个正则表达式的复杂度是  $O(2^n)$ 。当 `cy` 匹配失败的时候，正则引擎会尝试所有可能的排列组合，重复每个 `cx+` 以及包含它们的分组。例如，在尝试的过程深处，会出现一个这样的排列：`cx+` 匹配 `xxx`，第二个 `cx+` 匹配 `x`，然后接着这个分组会被重复 3 次，其中每个 `cx+` 匹配 `x`。如果存在 10 个 `x` 字符的话，那么就会存在 1024 种组合。如果把这个数量加到 32，那么我们就会要处理 4000000000 (40 亿) 种这样的组合，这肯定会让所有的正则引擎出现内存不足，除非它包含一个安全开关，能够自己放弃，并且宣布你的正则表达式过于复杂。

该例中，这个没有多大意义的正则表达式可以很容易被重写为 `(xx+y)`，这样它就可以在线性时间内找到完全一样的匹配。在实践中，对于更加复杂的正则表达式可能就不会很容易找到这样的解决方案了。

本质上来说，必须要小心当某个正则表达式的两个或者更多个部分会匹配相同文本的情形。在这些情形中，可能会需要原子分组来确保正则引擎不会去尝试所有的方式以把目标文本根据正则表达式的这两个部分进行分割。在测试你的正则表达式时，应该总要使用包含部分可以匹配，但是又不能整体被正则表达式匹配的（较长）测试目标。

## 参见

实例 2.13 和 2.14

## 2.16 检查一个匹配，但不添加到整体匹配中

### 问题描述

找出在一对 HTML 粗体标记之间的任何单词，但是不要把标记包含到正则表达式匹配

中。例如，如果目标文本是 My <b>cat</b> is fury，那么唯一的匹配应当是 cat。

## 解决方案

(?=<**>>)\w+(?=**</b>)  
正则选项：不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby 1.9

JavaScript 和 Ruby 1.8 支持向前看的 <(?=</b>>>，但是不支持向后看的 <(?=<**>>)>>**

## 讨论

### 环视

现代的正则流派都支持四种类型的环视（lookaround），它们拥有特殊的性质，可以放在环视内部的正则表达式部分所匹配的文本。实质上，环视会检查某些文本是否可以被匹配，但是并不会实际去匹配它。

向后看的环视被称作是逆序环视（lookbehind）。这是唯一可以从右向左遍历目标文本的正则表达式结构。肯定型逆序环视（positive lookbehind）的语法是 <(?=<text>>>。在这里由 <(?=<>4 个字符构成了起始括号。可以在环视内部放入什么内容（也就是这里的<text>所表示的）在不同正则表达式流派中是不一样的。但是总是可以放入字面文本，例如 <(?=<**>>)>>**

逆序环视会检查在逆序环视中的文本是否会出现紧挨着正则表达式引擎所到达位置的左边。如果用 <(?=<**>>)>> 来匹配 My <b>cat</b> is fury，逆序环视只有到正则表达式在目标文本中的字母 c 处开始进行匹配尝试时，才会匹配成功。正则引擎接着会进入逆序环视分组，告诉它向左边看。<**>> 在 c 的左边产生了匹配。引擎会在这个时候退出逆序环视，并且丢弃从匹配尝试开始由逆序环视所匹配到的任何文本。换句话说，正在进行的匹配会回到引擎刚刚进入逆序环视的地方。在这个例子中，正在进行匹配的是目标字符串中 c 之前的一个长度为 0 的匹配。逆序环视只会检查或者测试 <**>> 是否可以被匹配；但是它并不会实际上去匹配它。环视结构因此也被称作长度为 0 的断言。******

在逆序环视被匹配之后，字符类 \w+ 会尝试去匹配一个或者多个单词字符。它会匹配 cat。\w+ 并不属于任何类型的环视或者分组，因此它会正常地匹配文本 cat。我们说 \w+ 匹配并且消耗（consume）了 cat，而环视则只能匹配内容，却从来不会消耗任何东西。

向前看的环视，也就是照正则表达式通常遍历的方向看的情形，被称作顺序环视（lookahead）。顺序环视在本书中的所有正则流派中都拥有相同的支持。肯定型顺序环视（positive lookahead）的语法是 <(?=regex<>>>。这 3 个字符 <(?=> 构成了该分组的起始括号。在一个正则表达式中可以使用的任何东西都可以在顺序环视内部使用，在这里用 <regex> 来表示。

当在 `<(?=<b>)\w+(?=</b>)>` 中的 `\w+` 匹配了 `My <b>cat</b> is fury` 中的 `cat` 的时候，正则引擎就进入了顺序环视。在这个时候顺序环视唯一特殊的行为是正则引擎会记住它已经匹配了的文本部分，并把它同顺序环视关联起来。接着 `</b>` 会被正常匹配。现在正则引擎会退出顺序环视。在环视中正则表达式产生了匹配，因此环视自身也就匹配成功了。正则引擎接着会丢弃由环视匹配的文本，恢复在进入环视的时候它记住的正在匹配的内容。这样我们整体上的匹配进程就回到了 `cat`。因为这是我们正则表达式的结束，所以 `cat` 就成为了最终的匹配结果。

## 否定型环视

把环视中的相等符号换成感叹号的话，`<(?![regex])>` 就成为了一个否定型顺序环视（negative lookahead）。否定型顺序环视与肯定型顺序环视用起来是一样的，唯一的区别是肯定型顺序环视会在顺序环视中的正则式匹配时产生匹配，而否定型顺序环视则正好相反，它在当顺序环视内的正则式匹配时，会匹配失败。

匹配的过程则是完全相同的。引擎会在进入否定型顺序环视的时候保存当前匹配，然后试图正常地匹配顺序环视中的正则表达式。如果这个子表达式匹配的话，那么顺序环视会失败，而正则引擎会进行回溯。如果这个子表达式不能匹配的话，那么引擎会恢复当前匹配，然后继续处理正则表达式的剩余部分。

类似的，`<(?<!text)>` 是一个否定型逆序环视（negative lookbehind）。当逆序环视内的所有多选结构都无法通过从正则表达式在目标文本到达位置向后看来匹配的时候，否定型逆序环视才会匹配成功。

## 不同层次的逆序环视

顺序环视用起来比较容易。本书中讨论的所有正则流派都支持在顺序环视中放入一个完整的正则表达式。在正则表达式中可以使用的所有东西都可以用于顺序环视之内。你的大脑可能需要绕个弯，但是正则引擎会把这一切都处理得很好。

逆序环视则不同。正则表达式软件总是被设计来按照从左向右的方式查找目标文本。向后查找的实现通常需要一些特殊的处理：正则引擎会决定你可以在逆序环视中放多少个字符，先回跳那么多数量的字符，然后再在目标文本中从左向右匹配位于逆序环视中的文本。

基于这个原因，最早的实现中只允许在逆序环视中包含固定长度的字母文本。Perl、Python 和 Ruby 1.9 则更进一步，允许使用多选结构和字符类来把多个不同的固定长度的字面字符串放到逆序环视中。因此它们可以处理如下的例子：`<(?=<one|two|three|forty-two|gr[ae]y)>`。

在内部，Perl、Python 和 Ruby 1.9 会把这个表达式扩展为 6 个逆序环视的测试。首先，它们会回跳 3 个字符来检查 `<one|two>`，接着回跳 4 个字符来检查 `<gray|grey>`，然后回跳

5个字符来检查〈three〉，最后回跳9个字符检查〈forty-two〉。

PCRE 和 Java 对于逆序环视的实现则更进了一步。它们允许在逆序环视中使用任意的有限长度的正则表达式。这意味着可以使用除了无限长度量词（`(*)`、`(+)` 和 `({42,})`）之外的所有东西。在内部实现中，PCRE 和 Java 会计算在逆序环视中的正则表达式部分可能会匹配的文本的最小和最大长度。如果它匹配失败的话，那么它们会回跳一个字符再试，直到逆序环视产生匹配或者是尝试到了最大可能的字符数目。

如果这些听起来都不是很高效，事实上也正是如此。逆序环视用起来是非常方便的结构，但是它的速度就很一般了。稍后我们会讲解在根本不支持逆序环视的 JavaScript 和 Ruby 1.8 中的一个解决方案。这个解决方案实际上会比使用逆序环视的效率要高很多。

.NET 框架中的正则表达式引擎是唯一可以实际上从右向左应用一个完整正则表达式的引擎<sup>1</sup>。.NET 允许在逆序环视中使用任何东西，而且它会实际上从右向左来应用正则表达式。在逆序环视中的正则表达式和目标文本都是按照从右向左来进行扫描的。

## 两次匹配相同的文本

如果在正则表达式的开始处使用逆序环视，或者在正则表达式的结尾处使用顺序环视，那么这样做的效果就是要求在正则匹配之前或者之后出现一些东西，但不要把它们包含到匹配中。如果在正则表达式的中间使用环视的话，那么就可以对同一段文本应用多次检查。

在实例 2.3 的“流派相关的特性”小节中，我们讲解了如何使用字符类差来匹配一个泰国语的数字。只有在.NET 和 Java 中才会支持字符类的差。

如果一个字符既是泰国语字符（任何类型），又是数字（任意字符集），那么它就是一个泰国语数字。如果使用顺序环视，你可以在同一个字符上检查这两个要求：

```
(?=\p{Thai})\p{N}
```

正则选项：无  
正则流派：PCRE、Perl、Ruby 1.9

这个正则表达式只能用于支持 Unicode 字符集的 3 种流派，如实例 2.7 所示。但是使用顺序环视来匹配同一个字符多次的思想则可以用于本书中讨论的所有流派。

当正则引擎查找 `(?=\p{Thai})\p{N}` 的时候，它先是会在开始进行匹配的字符串中的每一个位置进入顺序环视。如果在该位置的字符不在泰国语字符集（也就是说 `\p{Thai}` 匹配失败）中，那么这次顺序环视就会失败。这也会导致整个匹配尝试失败，并迫使正则引擎到下一个字符处重新进行尝试。

当正则表达式遇到一个泰国语字符时，`\p{Thai}` 产生匹配。因此，环视 `(?=\p{Thai})`

<sup>1</sup> RegexBuddy 的正则引擎也允许在逆序环视中使用一个完整的正则表达式，但是它（还）没提供类似于.NET 中的 `RegexOptions.RightToLeft` 的功能来把整个正则表达式都倒过来。

也会匹配成功。当引擎退出环视的时候，它会恢复当前匹配进程。在这个例子中，也就是在刚找到泰国语字符之前长度为 0 的匹配。接下来要匹配的是 `\p{N}`。因为顺序环视已经丢弃了它的匹配，因此 `\p{N}` 会同 `\p{Thai}` 已经匹配了的那个字符进行比较。如果该字符拥有 Unicode 属性 Number 的话，那么 `\p{N}` 会匹配成功。因为 `\p{N}` 并不位于一个环视之内，所以它会消耗掉这个字符，同时我们也就找到了想要的泰国语数字。

## 环视是原子分组

当正则表达式引擎退出一个环视分组的时候，它会丢弃掉环视匹配的文本。因为该文本被丢弃了，所以由位于环视之内的多选结构或者量词所记住的任意回溯位置也都会被丢弃。这样实际上就会把顺序环视和逆序环视都变成了原子分组。实例 2.15 中详细讲解了原子分组的概念。

在大多数情形下，环视的原子特性是无关紧要的。环视只是一个用来检查位于环视中的正则表达式是匹配成功还是失败的一个断言。它可以通过多少种方式产生匹配并不重要，因为它不会消耗目标文本中的任何字符。

当你在顺序环视（以及逆序环视，如果你的正则流派支持）之内使用捕获分组的时候，它的原子特性才会产生意义。虽然顺序环视不会消耗任何文本，但是正则引擎会记住文本中哪些部分被位于顺序环视中的任何捕获分组匹配了。如果顺序环视位于正则表达式的结尾处，那么你实际上所得到的捕获分组所匹配的文本是正则表达式自身所不能匹配的。如果这个顺序环视位于正则表达式中间，那么你所得到的多个捕获分组可能会匹配到目标文本中的重叠部分。

环视的原子特性也可能会改变整个正则表达式的匹配，出现这种情况的唯一可能是当你在环视之外使用一个反向引用来指向在环视之内所创建的捕获分组。下面来看一下这个正则表达式：

```
(?= (\d+)) \w+ \1
正则选项: 无
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

一眼看过去，你可能会认为这个正则表达式能够匹配 123x12。`\d+` 会把 12 捕获到第一个捕获分组中，接着 `\w+` 会匹配 3x，最后 `\1` 会再次匹配 12。

但这不可能发生。正则表达式会进入环视并捕获分组。贪心的 `\d+` 会匹配 123。这个匹配被存储到第一个捕获分组中。引擎接着退出顺序环视，把当前匹配重新设置为字符串的开始，并且丢弃由贪心的加号所记住的回溯位置，但是会在第一个捕获分组中保留所存储的 123。

现在，贪心的 `\w+` 会在字符串开始处进行尝试。它会把 123x12 都吃掉。这时指向 123 的 `\1` 在字符串结尾处匹配失败。`\w+` 会回溯一个字符。`\1` 还会失败。`\w+` 会继续回溯，直到它放弃了除了目标文本中第一个 1 之外的所有字符。`\1` 在第一个 1 之后

还是会匹配失败。

如果正则引擎能够返回到顺序环视中，放弃 123 而选择 12，那么最后的 12 会匹配 `\1`。但是正则引擎并不会这样做。

正则引擎此时并不存在可以选择的任何回溯位置。`\w+` 已经回退到头了，而环视迫使 `\d+` 把它的回溯位置都丢掉了。因此匹配尝试会宣告失败。

## 不使用逆序环视的解决方案

虽然我们在前面讲了这么多内容，但是如果你用的是 Python 或者 JavaScript，那么这些对你都毫无用处，因为你根本就不能使用逆序环视。使用这两种正则流派无法恰好解决前面所给出的问题，但是你可以通过使用捕获分组来模仿逆序环视。下面给出的这个可选方案也可以在所有其他正则流派中使用：

```
(<b>) (\w+) (?=</b>)
正则选项：不区分大小写
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

作为逆序环视的替代，我们使用了一个捕获分组来匹配起始 tag：`<b>`。我们还把感兴趣的匹配部分，也就是`\w+`，放到了捕获分组中。

当把这个正则表达式应用到 My `<b>cat</b>` is fury 之上的时候，这个正则表达式的匹配会是cat。第一个捕获分组会保存<b>，而第二个会保存 cat。

如果题目的要求是只匹配 cat（在两个 `<b>` tag 之间的单词），因为你只想提取文本中的内容，那么可以通过只保存第二个捕获分组所匹配的文本，而不是整个正则表达式匹配的文本来达到这一目标。

如果要求是想要进行查找和替换，而只替换在两个 tag 之间的单词的话，那么可以使用一个反向引用来指向第一个捕获分组，把起始 tag 重新添加到替代文本中。在这个例子中，实际上并不需要捕获分组，因为起始 tag 总是相同的。但是当它可变的时候，捕获分组会重新插入与前面匹配到的一模一样的内容。实例 2.21 对此有更详细的讲解。

最后，如果你真的想要模拟逆序环视的话，那么可以使用两个正则表达式来完成。首先，不使用逆序环视来查找你的正则表达式。当它匹配成功时，把在匹配之前的目标文本子串复制到一个新的字符串变量中。然后用第二个正则表达式，加上字符串结束定位符（`\z` 或 `\$`），检查你在逆序环视中所做的事情。这个定位符会确保第二个正则式的匹配一定位于该字符串的结尾。因为剪切字符串的地方是第一个正则表达式匹配的地方，所以这样就会把第二个匹配刚好放到第一个匹配的左边。

在 JavaScript 中，可以使用如下的代码来完成这项任务：

```
var mainregexp = /\w+ (?=</b>)/;
var lookbehind = /<b>\$/;
if (match = mainregexp.exec("My <b>cat</b> is fury")) {
    var result = lookbehind.exec(match[0]);
    if (result) {
        // 替换逻辑
    }
}
```

```
// Found a word before a closing tag </b>
var potentialmatch = match[0];
var leftContext = match.input.substring(0, match.index);
if (lookbehind.exec(leftContext)) {
    // Lookbehind matched:
    // potentialmatch occurs between a pair of <b> tags
} else {
    // Lookbehind failed: potentialmatch is no good
}
} else {
    // Unable to find a word before a closing tag </b>
}
```

## 参见

实例 5.5 和 5.6。

## 2.17 根据条件匹配两者之一

### 问题描述

创建一个正则表达式，匹配一个由逗号分隔的单词列表 one、two 和 three。每个单词可以在该列表中出现任意多次，但是每个单词必须至少出现一次。

### 解决方案

```
\b(?:(?:one)|(two)|(three))(?:,|\b){3,}(?(1)|(?!))(?(2)|(?!))(?(3)|(?!))
```

正则选项：无

正则流派：.NET、JavaScript、PCRE、Perl、Python

Java 和 Ruby 并不支持条件判断。在 Java 或 Ruby（或者其他任何语言）中进行编程的时候，你可以使用不带有条件判断的正则表达式，然后再编写一些额外的代码来检查其中的三个捕获分组是否都匹配了相应的内容。

```
\b(?:(?:one)|(two)|(three))(?:,|\b){3,}
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

.NET、JavaScript、PCRE、Perl 和 Python 支持使用编号捕获分组的条件判断（conditional）。`<(?(1)then|else)>` 是用来检查第一个捕获分组是否产生匹配的一个条件判断。如果它产生了匹配，正则引擎会尝试去匹配 `then`。如果捕获分组到目前为止还没有参与匹配尝试，那么就会去尝试匹配 `else`。

这里的括号、问号和竖线都是属于条件判断语法的一部分。它们在这里并不拥有平时

的含义。你可以在 `<then>` 和 `<else>` 部分中使用任意种类的正则表达式。唯一的限制是如果想要在其中一个部分之内使用多选结构，那么你必须使用一个分组来把它包到一起。在条件判断中只允许直接出现一个竖线。

如果愿意，也可以省略掉 `<then>` 或 `<else>` 的部分。空的正则表达式总是会找到一个长度为 0 的匹配。这个实例所给的解决方案中使用了 3 个条件判断，它们都包含空的 `<then>` 部分。如果捕获分组参与了匹配，那么这个条件判断只会简单地产生匹配。

一个空的否定型顺序环视 `<(?!)>` 被用在了 `<else>` 部分。因为空的正则表达式总是会产生匹配，所以包含空正则表达式的否定型顺序环视则总是会匹配失败。因此，当第一个捕获分组没有匹配到任何东西的时候，条件判断 `<(?(1)|(?!))>` 总是会失败。

通过把这三个必需的多选结构放到自己的捕获分组中，我们可以在正则表达式的结尾使用 3 个条件判断来检查是否所有的捕获分组都捕获到了内容。

.NET 还支持命名的条件判断。`<(?(name)then|else)>` 会检查命名的捕获分组 name 是否参与了匹配尝试。

为了更好地理解条件判断是如何工作的，我们来看一个正则表达式 `<(a)?b(?(1)c|d)>`。它本质上是与 `<abc|bd>` 等价的一种更为复杂的形式。

如果目标字符串是由一个 `a` 开头的，那么它就会被捕获到第一个捕获分组中。如果不是，那么第一个捕获分组就不会参与到匹配尝试中。在该捕获分组之后的问号是很重要的，因为这使得整个分组成为可选的。如果不存在 `a` 的话，该分组会重复 0 次，因此不会有办法捕获任何内容。它不能捕获一个长度为 0 的字符串。

如果你使用的是 `<(a?)>`，该分组总是会参与到匹配尝试中。而在该分组之后并不存在量词，所以它会正好重复一次。该分组或者捕获 `a`，或者不捕获任何东西。

不管 `<a>` 是否会产生匹配，下一个记号是 `<b>`。然后是条件判断。如果捕获分组参与了匹配尝试，即使它捕获的是长度为 0 的字符串（在这里是不可能出现的），都会尝试匹配 `<c>`。如果没有的话，那么会尝试匹配 `<d>`。

用一句话来描述，`<(a)?b(?(1)c|d)>` 或者匹配 ab 之后跟着 c，或者匹配 b 之后跟着 d。

在.NET、PCRE 和 Perl 中（但是不包括 Python），条件判断中还可以使用环视。`<(?(?=if)then1else)>` 首先会把 `<(?=if)>` 当作一个正常的顺序环视来进行检查。实例 2.16 中讲解了它是如何执行的。如果环视匹配成功的话，那么会接着尝试匹配 `<then>` 部分。如果没有成功的话，那么会尝试匹配 `<else>` 中的正则表达式。因为环视的长度为 0，所以 `<then>` 和 `<else>` 中的正则表达式都会在目标文本中 `<if>` 匹配成功或者失败的同一位置处接着进行尝试。

在条件判断中，除了使用顺序环视，也可以使用逆序环视。你还可以使用否定型的环视，虽然我们并不推荐这样做，因为它会把“then”和“else”的含义反转，从而造成

不必要的混淆。



### 提示

使用环视的条件判断也可以写成不使用条件判断的形式：`<(?=if)then|(?!=if)else>`。如果肯定型顺序环视匹配成功，那么就会尝试匹配`<then>`部分。如果肯定型顺序环视匹配失败，那么会尝试匹配第二个选择。接下来的否定型顺序环视会做同样的检查。否定型顺序环视当`<if>`匹配失败的时候会匹配成功，因为`<(?=if)>`已经匹配失败了，所以这是可以确定的。然后，就会尝试匹配`<else>`。把顺序环视放到条件判断中会节省时间，因为条件判断只会尝试匹配`<if>`一次。

## 参见

实例 2.9 和 2.16。

## 2.18 向正则表达式中添加注释

### 问题描述

`\d{4}-\d{2}-\d{2}` 可以匹配一个 `yyyy-mm-dd` 形式的日期，但是不会对数字做任何检查。当你知道数据中不会包含非法日期时，这样一个简单的正则表达式就够用了。试着在该正则表达式中添加注释来说明该正则表达式的每个部分的含义。

### 解决方案

```
\d{4}      # Year
-
# Separator
\d{2}      # Month
-
# Separator
\d{2}      # Day
正则选项: 宽松排列 (Free-spacing)
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

### 讨论

#### 宽松排列模式

用户很容易就会把一个正则表达式变得复杂无比，并且难以理解。就像应该在源代码中添加注释一样，你也应该向除了最简单形式之外的所有正则表达式中添加注释。

本书中的所有正则表达式流派，除了 JavaScript 之外，都提供了某种形式的正则表达式语法来帮助你很容易地为正则表达式添加清晰的注释。你可以通过打开宽松排列 (free-spacing) 模式来应用这种语法。在不同的编程语言中它可能会有不同的名称。

在.NET 中，需要设置 RegexOptions.IgnorePatternWhitespace 选项。而在 Java 中，则要传递 Pattern.COMMENTS 选项。Python 用的是 re.VERBOSE。PHP、Perl 和 Ruby 则使用/x 选项。

把宽松排列模式打开会产生两个效果。首先，它会把位于字符类之外的井号 (#) 转变成一个元字符。井号会作为一个注释的开始，该注释的结尾是一行的结束或者该正则表达式的结束（二者中取先到的那个）。井号以及其后的所有内容都会被正则表达式引擎忽略。如果需要匹配一个字面上的井号字符，那么就需要把它放到一个字符类<#>中，或者把它转义为\#。

宽松排列模式的另外一个效果是：位于字符类之外的所有空白字符，包括空格、制表符和换行符，都会被忽略。为了匹配一个字面上的空格，就需要把它放到一个字符类<\p>中，或者把它转义为\p。如果你更关心可读性的话，那么也可以选择使用十六进制数的转义形式\x20，或者是 Unicode 转义形式\u0020或\x{0020}。如果要匹配一个制表符，则应该使用\t。至于换行符，则可以使用\r\n（Windows）或者\n（Unix/Linux/OS X）。

宽松排列模式并不会改变位于字符类之内的任何内容。一个字符类可以看作是一个单个记号。位于字符类之内的任意空白字符或者井号都是被添加到字符类中的字面字符。打断字符类来对其中的某个部分进行注释是不允许的。

## Java 中的宽松排列字符类

对于正则表达式来说，往往至少会有一种流派会与其他流派不兼容。在这个实例中，不兼容的那个流派就是 Java。

在 Java 中，字符类并不作为单个记号来进行分析。如果你打开了宽松排列模式，那么 Java 就会忽略字符类中的空格，而且在字符类中的井号也意味着注释的开始。这就意味着，你不能使用<\p> 和<\#> 匹配这些字面字符。作为替代选择的是使用\u0020 和\#。

## 变体

(?#Year)\d{4} (?#Separator)-(?#Month)\d{2}-(?#Day)\d{2}

正则选项：无

正则流派：.NET、PCRE、Perl、Python、Ruby

不管是出于什么原因，如果你不能使用或者是不想使用宽松排列语法，那么还可以通过<(?#comment)> 的方式添加注释。在<(?#>和<>之间的所有字符会被忽略。

不幸的是，本书中讲到的流派中唯一不支持宽松排列的流派，也就是 JavaScript，也不支持这种注释语法。另外在 Java 中也不支持这种语法。

```
(?x)\d{4}      # Year  
-             # Separator  
\d{2}         # Month
```

```
-          # Separator
\{2}      # Day
正则选项: 无
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

如果不能在正则表达式之外打开宽松排列模式，那么你可以把模式修饰符 `((?x))` 放到正则表达式的最开始处。一定要注意在 `((?x))` 之前不能存在任何空格。因为只有在这个模式修饰符之后才会开始宽松排列模式，因此在它之前的任意空格都是不能忽略的。

我们在实例 2.1 中的“区分大小写的匹配”中已经详细讲解了模式修饰符有关的内容。

## 2.19 在替代文本中添加字面文本

### 问题描述

查找并把任何正则表达式匹配从字面上替换为这 8 个字符：`$%*$1\\1`。

### 解决方案

```
$%\\*\\$1\\1
替代文本流派: .NET、JavaScript
\\$%\\*\\$1\\1
替代文本流派: Java
$%\\*\\$1\\1
替代文本流派: PHP
\\$%\\*\\$1\\1
替代文本流派: Perl
$%\\*\\$1\\1
替代文本流派: Python、Ruby
```

### 讨论

#### 在替代文本中转义字符的时机和方式

这个实例会讲解各种替代文本流派中使用的不同转义规则。在替代文本中，你可能会需要转义的两个字符是美元符号 (\$) 和反斜杠 (\)。而它们所使用的转义字符也分别是美元符号和反斜杠。

在这个例子中的百分号 (%) 和星号 (\*) 总是字面字符，然而一个前导的反斜杠也可能被当作一个转义字符，而不是一个实际上的反斜杠。`«$1»` 和/或 `«\\1»` 是指向一个捕获分组的反向引用。实例 2.21 中会讲解哪些流派对于反向引用会使用哪种语法。

这个题目对于我们讲到的 7 种替代文本流派就存在了 5 种解决方案，这个事实也说明了对于替代文本语法来说，根本不存在什么标准。

## .NET 和 JavaScript

.NET 和 JavaScript 总是把反斜杠当作一个字面字符。不需要再用另外一个反斜杠来对它进行转义，否则就会在替换中得到两个反斜杠。

单个出现的美元符号也是一个字面字符。只有当它们之后是一个数字、&、反引号、垂直引号、下划线、加号或者另外一个美元符号的时候，才需要被转义。要转义一个美元符号，只需要在它前面再加一个美元符号。

如果你觉得这样会读起来更加清晰，也可以选择把所有的美元符号都用两个来表示。下面这个解答也是同样正确的：

```
$$%\*$${$1}\1  
替代文本流派: .NET、JavaScript
```

.NET 还要求对后面跟着一个起始花括号的美元符号进行转义。在.NET 中，«\${group}» 是一个命名反向引用。JavaScript 则不支持命名反向引用。

## Java

在 Java 中，反斜杠被用来在替代文本中转义反斜杠和美元符号。所有字面的反斜杠和所有字面的美元符号都必须被转义。如果你不转义它们，Java 就会产生一个例外(exception)。

## PHP

PHP 要求后面跟数字的反斜杠、后面跟数字或者起始花括号的美元符号，都需要使用反斜杠来进行转义。

反斜杠也可以对另外一个反斜杠进行转义。因此，你需要使用«\\»来在替换文本中表示两个实际上的反斜杠。所有其他的反斜杠都被认为是字面上的反斜杠。

## Perl

Perl 与其他替代文本流派都有些不同：它其实并不拥有一种替代文本流派。虽然其他编程语言都在查找和替换过程中进行替换时拥有一些特殊逻辑，比如 «\$1»，然而 Perl 中这只是正常的变量插值(interpolation)。在替代文本中，需要对所有实际上的美元符号用反斜杠进行转义，就像在任意双引号字符串中一样。

一个例外是 Perl 并不支持反向引用的 «\\$1» 语法。因此，并不需要对后面跟着数字的反斜杠进行转义。为了避免反斜杠对美元符号进行转义，因此需要对后面跟着美元符号的反斜杠进行转义。

反斜杠也可以转义另外一个反斜杠。因此，你需要使用 «\\» 来在替换文本中表示两个实际上的反斜杠。所有其他的反斜杠都被认为是字面上的反斜杠。

## Python 和 Ruby

美元符号在 Python 和 Ruby 的替代文本中并不拥有特殊含义。如果反斜杠后面跟着一个会给反斜杠赋予特殊含义的字符，那么反斜杠需要使用另外一个反斜杠进行转义。

在 Python 中，`{{\1}}~{{\9}}`，以及`{{\g}}`会创建反向引用。因此这些反斜杠就需要进行转义。

在 Ruby 中，你需要对后面跟着一个数字、`&`、反引号、垂直引号或者加号的反斜杠进行转义。

在这两种语言中，反斜杠都可以转义另外一个反斜杠。因此，你需要使用`{{\\ \\ \\ \\}}`在替换文本中表示两个实际上的反斜杠。所有其他的反斜杠都被认为是字面上的反斜杠。

## 更多关于字符串字面量的转义规则

记住在本章中，我们关心的只是正则表达式和替代文本自身。下一章会讲解编程语言和字符串字面量的内容。

当你在要传递给`replace()`函数的实际字符串变量中包含这样的文本时，前面所给出的替代文本是没有问题的。换句话说，如果你在应用程序中给用户提供一个文本框，让用户输入替代文本，那么这些解决方案所给出的正是为了让查找和替换正常工作，用户需要输入的内容。如果你使用 RegexBuddy 或者另外一个正则测试工具来测试查找和替换命令，那么在这个实例中给出的替代文本也同样会得到所期望的结果。

但是如果你把它们直接粘贴到源代码中，在两边放上引号字符，那么这些同样的替代文本并不能正常使用。编程语言中的字符串文本拥有它们自己的转义规则，因此你需要在替代文本的转义规则之上再遵守这些转义规则才行。最终你所得到的很可能会是一堆乱七八糟的反斜杠。

## 参见

实例 3.14。

## 2.20 在替代文本中添加正则匹配

### 问题描述

执行查找和替换，把所有 URL 都转换成指向该 URL 的 HTML 链接，并使用该 URL 作为链接的文本。在这个练习中，把 URL 定义为“`http:`”以及其后所有的非空字符。例如，`Please visit http://www.regexcookbook.com` 应该被转换为`Please visit <a href="http://www.regexcookbook.com">http://www.regexcookbook.com</a>`。

# 解决方案

## 正则表达式

```
http:\S+
正则选项: 无
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 替代文本

```
<a●href="$&">$&</a>
    替代文本流派: .NET、JavaScript、Perl
<a●href="$0">$0</a>
    替代文本流派: .NET、Java、PHP
<a●href="\0">\0</a>
    替代文本流派: PHP、Ruby
<a●href="\&">\&</a>
    替代文本流派: Ruby
<a●href="\g<0>">\g<0></a>
    替代文本流派: Python
```

## 讨论

把整个正则匹配重新插回到替代文本中是向匹配文本之前、之后或者两边，甚至是在匹配文本的多个副本之间插入新文本的一种比较容易的方式。除非使用的是 Python，否则不必在你的正则表达式中添加任何捕获分组就能够重复使用整个的匹配结果。

在 Perl 中，«\$&» 实际上是一个变量。在每次正则表达式匹配成功之后，Perl 都会把整个正则匹配保存到这个变量中。

.NET 和 JavaScript 沿用了 «\$&» 的语法来把正则匹配插入到替代文本中。Ruby 使用反斜杠而不是美元符号作为替代文本中的记号，因此会使用 «\&» 来指代整个匹配。

Java、PHP 和 Python 并不存在一个特殊记号来重新插入整个正则匹配，但是它们也允许把捕获分组匹配到的文本插入到替代文本中，这会在下一小节中进行讲解。整个匹配则是一个编号为 0 的隐式捕获分组。对于 Python 来说，我们需要使用命名捕获的语法以引用 0 号分组。在 Python 中并不支持 «\0»。

.NET 和 Ruby 也支持第 0 个捕获分组的语法，但是使用什么语法并不重要。因为不管什么语法结果都是一样的。

## 参见

第 1 章中的“使用正则表达式查找和替换”以及实例 3.15。

## 2.21 把部分的正则匹配添加到替代文本中

### 问题描述

匹配任意连续的 10 个数字序列，例如 1234567890。并把这个序列转换成格式正确的电话号码形式，例如(123) 456-7890，

### 解决方案

#### 正则表达式

```
\b(\d{3})(\d{3})(\d{4})\b
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

#### 替代文本

```
($1)•$2-$3  
替代文本流派：.NET、Java、JavaScript、PHP、Perl  
(${1})•${2}-${3}  
替代文本流派：.NET、PHP、Perl  
(\1)•\2-\3  
替代文本流派：PHP、Python、Ruby
```

### 讨论

#### 使用捕获分组的替代文本

实例 2.10 讲解了在正则表达式中如何使用捕获分组来多次匹配同一个文本。在正则表达式中，每个捕获分组匹配到的文本在每次成功匹配之后都是可用的。你可以把一些或者所有捕获分组中的文本——按照任意顺序，或者甚至可以多次——插入到替代文本中。

一些流派，例如 Python 和 Ruby，在正则表达式和替代文本中对于反向引用使用相同的语法 «\1»。其他流派使用的则是 Perl 中的语法 «\$1»，也就是说使用的是美元符号而不是反斜杠。PHP 对于两种语法都支持。

在 Perl 中，«\$1» 以及其他编号的分组实际上都是变量，它们的值会在每次正则匹配成功之后进行设置。你可以在代码中的任意地方使用它们，直到下一次正则匹配开始。.NET、Java、JavaScript 和 PHP 只在替代文本语法中支持 «\$1»。这些编程语言还提供了在代码中访问捕获分组的其他方式。这会在第 3 章中详细解释。

#### \$10 及更大分组

本书中的所有正则流派都支持在一个正则表达式中使用最多 99 个捕获分组。在替代文

本中，对于 «\$10» 或 «\10» 以及更大的分组则会产生二义性。这些可以被解释为或者是第 10 个捕获分组，或者是第一个捕获分组后跟着一个字面上的 0。

.NET、PHP 和 Perl 允许在数字周围使用花括号来使你的意图更加清晰。«\${10}» 总是代表第 10 个捕获分组，而 «\${1}0» 则总是意味着第一个分组后跟着一个实际的 0。

Java 和 JavaScript 对于 «\$10» 使用了更加聪明的处理办法。如果在你的正则表达式中存在一个两位数的捕获分组的话，那么两个数字都会被用于捕获分组。如果并不存在这么多捕获分组的话，那么只有第一个数字被用来引用分组，这样第二个数字就被当作了字面字符。因此 «\$23» 只有在它存在的时候才被认为是第 23 个捕获分组。否则，它被当作是第 2 个捕获分组后面跟着一个字面上的 «3»。

.NET、PHP、Perl、Python 和 Ruby 总是把 «\$10» 和 «\10» 当作第 10 个捕获分组，而不管它们是否真的存在。如果它不存在的话，那么就会出现引用不存在的分组的行为。

## 对不存在分组的引用

在这个实例的解决方案中的正则表达式拥有 3 个捕获分组。如果你在替代文本中输入了 «\$4» 或 «\4»，那么就添加了一个到不存在的捕获分组的引用。这可能会触发如下的 3 种不同的行为。

Java 和 Python 会报错，并且产生一个例外或者返回一个错误消息。因此不要在这些流派中使用不合法的反向引用。（事实上，你不应当在任何流派中使用不合法的反向引用。）如果想要添加的是字面上的 «\$4» 或 «\4»，那么就需要对美元符号或反斜杠进行转义。实例 2.19 中对此有详细的解释。

PHP、Perl 和 Ruby 会替换在替代文本中所有的反向引用，也包括了那些指向不存在的分组的引用。不存在的分组显然不会捕获任何文本，因此对它们的引用也就简单地被替换为空。

最后，.NET 和 JavaScript 则把对不存在分组的引用当作是替代文本中的字面文本。

所有流派都会替换在正则表达式中存在但是并没有捕获到任何东西的分组。它们都会被替换为空。

## 使用命名捕获的解决方案

### 正则表达式

```
\b(?:area)\d{3})(?:exchange)\d{3})(?:number)\d{4})\b  
正则选项: 无  
正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9  
  
\b('area'\d{3})(?'exchange'\d{3})(?'number'\d{4})\b  
正则选项: 无  
正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9
```

```
\b(?P<area>\d{3})(?P<exchange>\d{3})(?P<number>\d{4})\b
```

正则选项: 无

正则流派: PCRE 4 或更新版本、Perl 5.10、Python

## 替代文本

```
(${area})•${exchange}-${number}
```

替代文本流派: .NET

```
(\g<area>)•\g<exchange>-\g<number>
```

替代文本流派: Python

```
(\k<area>)•\k<exchange>-\k<number>
```

替代文本流派: Ruby 1.9

```
(\k'area')•\k'exchange'-\k'number'
```

替代文本流派: Ruby 1.9

```
($1)•$2-$3
```

替代文本流派: .NET、PHP、Perl 5.10

```
(${1})•${2}-${3}
```

替代文本流派: .NET、PHP、Perl 5.10

```
(\1)•\2-\3
```

替代文本流派: PHP、Python、Ruby 1.9

## 支持命名捕获的流派

如果你在正则表达式中使用了命名捕获分组，那么在.NET、Python 和 Ruby 1.9 中允许在替代文本中使用命名反向引用。

在.NET 和 Python 中，命名反向引用的语法与命名和编号的捕获分组是一样的。只需要在花括号或者尖括号之间指明分组的名称或编号即可。

Ruby 在替代文本中使用与其在正则表达式中一样的反向引用语法。在 Ruby 1.9 中的命名捕获分组中，它的语法是 «\k<group>» 或 «\k'group'»。之所以提供在尖括号和单引号之间的两种选择只是出于书写方便的考虑。

Perl 5.10 和 PHP（使用 PCRE）在正则表达式中支持命名捕获分组，但是在替代文本中不提供支持。你可以在替代文本中使用编号的捕获分组来引用正则表达式中的命名捕获分组。Perl 5.10 和 PCRE 会对所有命名和不命名的分组进行编号，顺序是从左向右。

.NET、Python 和 Ruby 1.9 同样允许使用编号来引用命名分组。然而，.NET 对于命名分组则采用了不同的编号策略，这在实例 2.11 中已经进行了讲解。在.NET、Python 或 Ruby 中混合使用分组名称和编号是不推荐的。或者给所有分组都命名，或者所有分组都不要命名。对于命名分组应该总是使用命名的反向引用。

## 参见

第 1 章中的“使用正则表达式查找和替换”以及实例 2.9、2.10、2.11 和 3.15。

## 2.22 把匹配上下文插入到替代文本中

### 问题描述

创建一个替代文本，用来把正则表达式匹配替换为正则匹配之前的文本，然后是整个的目标文本，然后是正则匹配之后的文本。例如，如果在 `BeforeMatchAfter` 中找到的是 `Match`，那么把这个匹配换成 `Before$&After`，因此最终得到的新文本是 `Before$&After`。

### 解决方案

```
$`$$_$'  
替代文本流派: .NET、Perl  
\```\&\``'\  
替代文本流派: Ruby  
$`$`$&$'$'  
替代文本流派: JavaScript
```

### 讨论

术语上下文（context）指的是正则表达式所应用于的目标文本。总共存在 3 种上下文：正则匹配之前的目标文本、正则匹配之后的目标文本以及整个目标文本。在匹配之前的文本有时候被称作是左上下文（left context），而匹配之后的文本则相应地被称作是右上下文（right context）。整个目标文本则是左上下文、匹配加上右上下文。

.NET 和 Perl 支持使用 «\$`»、«\$'» 和 «\$\$\_» 来把所有 3 种形式的上下文插入到替代文本中。事实上，这些在 Perl 中是在每次成功的正则匹配之后都会赋值的变量，并且可以在任意代码中使用，直到遇到下一次匹配尝试。“美元符号+反引号”是左上下文。在一个 U.S. 标准键盘上，你可以按键盘左上角 1 键左边的键来输入反引号。“美元符号+垂直引号”是右上下文。垂直引号也就是通常的单引号。在一个 U.S. 键盘上，它位于分号和回车键（Enter）之间。“美元符号+下划线”是整个目标文本。

与.NET 和 Perl 类似，JavaScript 也使用 «\$`» 和 «\$'» 来代表左右上下文。然而，JavaScript 并不包含一个记号来插入整个目标文本。你可以自己重构目标文本，把整个正则匹配使用 «\$&» 添加到左右上下文之间即可。

Ruby 通过 «\`» 和 «\`'» 支持左右上下文，并且使用 «\&» 来插入整个正则匹配。与 JavaScript 一样，Ruby 中也不存在插入整个目标文本的记号。

### 参见

第 1 章中的“使用正则表达式查找和替换”以及实例 3.15。

# 使用正则表达式编程

## 编程语言和正则流派

本章讲解如何在你所选择的编程语言中使用正则表达式。本章中所给的实例假设你已经有了一个可用的正则表达式；上一章可以帮助你完成这个任务。现在，你要做的事情是把这个正则表达式插入到源代码中，从而可以让它完成一些实际的任务。

在本章中，我们会尽最大努力来讲解每段代码是如何工作的，以及为什么会这样。因为本章所包含的内容中有很多细节，所以从头到尾通读本章会让人觉得有点儿冗长乏味。如果你是第一次阅读本书，那么我们建议你先略读本章，以便能够让自己了解本章能够帮你做些什么。接下来，当想要在代码中使用随后章节中介绍的某个正则表达式的时候，再返回到本章来，学习到底如何把正则表达式集成到你所选择的编程语言中。

第 4~8 章会讲解如何使用正则表达式来解决现实世界中的问题。这些章节会关注于正则表达式自身，而且在这些章节中的许多实例并不会给出任何源代码。为了让你在这些章节中看到的正则表达式能够工作，可以简单地把它们插入到本章中所给的某个代码片段中。

因为其他章节所关注的都是正则表达式，所以它们给出的解决方案针对的是具体的正则表达式流派，而不是针对具体的编程语言。正则流派并不能同编程语言一一对应。脚本语言往往会拥有它们自己的内嵌正则表达式流派，而其他编程语言则需要依赖于函数库来提供正则表达式的支持。有些库可以用于多种语言，而有些语言则可以使用多种正则库。

第 1 章中的“多种正则表达式流派”一节中介绍了本书所涉及的所有正则表达式流派。而第 1 章中稍后的“多种替代文本流派”一节中则列出了替代文本的流派，它们都会用于使用正则表达式进行查找和替换的时候。本章中讲解的所有编程语言则会使用上述的流派之一。

## 本章涉及的编程语言

本章会讲到 8 种编程语言。每个实例都会为所有这 8 种编程语言给出单独的解答，而且在许多实例中还会对所有 8 种语言进行分别的讨论。如果一个技术可以应用于多种语言，那么我们会在每种语言的讨论中重复讨论该技术。之所以这样做，是因为便于读者放心地略去不感兴趣的编程语言的相关的讨论部分。

### C#

C# 使用了 Microsoft .NET 框架。System.Text.RegularExpressions 中的类使用的是“.NET”正则表达式和替代文本流派。本书涉及 C# 1.0~C# 3.5，或者说是 Visual Studio 2002 直到 2008。

### VB.NET

本书使用 VB.NET 和 Visual Basic.NET 来指代 Visual Basic 2002 和之后的版本，从而可以把这些版本同 Visual Basic 6 和之前的版本区分开来。Visual Basic 现在也使用 Microsoft .NET 框架。System.Text.RegularExpressions 中的类使用的是“.NET”正则表达式和替代文本流派。本书会涉及 Visual Basic 2002 直到 2008。

### Java

Java 4 是通过 java.util.regex 包来提供内置的正则表达式支持的第一个 Java 版本。java.util.regex 包使用的是“Java”正则表达式和替代文本流派。本书会涉及 Java 版本 4、5 和 6。

### JavaScript

这是通常被称作 JavaScript 的编程语言中所使用的正则流派。所有现代的网页浏览器中都实现了这种流派，包括 Internet Explorer（从 5.5 版开始）、Firefox、Opera、Safari 和 Chrome。许多其他应用程序也使用 JavaScript 来作为一种脚本语言。

严格来讲，在本书中我们使用 JavaScript 这个术语所指代的是在 ECMA-262 标准的第 3 版中所定义的编程语言。这个标准定义了 ECMAScript 编程语言，而该语言更为人知的版本则是在不同网页浏览器中所实现的 JavaScript 和 JScript 语言。

ECMA-262v3 同样还定义了 JavaScript 所使用的正则表达式和替代文本的流派。这些流派在本书中都被称作“JavaScript”流派。

### PHP

PHP 拥有 3 种不同的正则表达式函数集合。我们强烈推荐读者使用 preg 类的函数。因此，本书只会涉及 preg 类的函数，这些函数从版本 4.2.0 开始就被内置在 PHP 中。本书会涉及 PHP 版本 4 和 5。preg 类的函数是在 PCRE 函数库之上生成的 PHP 包装函数。PCRE 正则流派在本书中被称作“PCRE”。因为 PCRE 并没有包含查找

和替换功能，所以 PHP 开发者为 preg\_replace 设计了它们自己的替代文本语法。

mb\_ereg 类的函数属于 PHP 的“多字节”函数，后者被设计来能够更好地用于传统上使用多字节编码的字符集（比如汉语和日语）之上。在 PHP 5 中，mb\_ereg 类的函数使用的是最初为 Ruby 所开发的 Oniguruma 正则函数库。在本书中，Oniguruma 正则流派被称作“Ruby 1.9”。只有当你被明确要求需要处理多字节代码页，而且对于 PHP 的 mb\_ereg 类的函数已经比较熟悉的前提下，我们才推荐你使用 mb\_ereg 类的函数。

ereg 类的函数是 PHP 中使用的最古老的一组正则表达式函数，它们从 PHP 5.3.0 开始被正式标记为过时。它们并不依赖于外部的函数库，而且它们实现的是 POSIX ERE 流派。该流派提供了一个有限的特性集合，因而在本书中并没有进行讨论。POSIX ERE 是 Ruby 1.9 和 PCRE 流派的一个严格子集。你可以把任意的 ereg 函数中调用的正则表达式拿来，用在 mb\_ereg 或 ereg 类的函数中。对于 ereg 类的函数，你还需要添加 Perl 风格的分隔符（参考实例 3.1）。

### Perl

在 Perl 语言中内置的正则表达式支持是正则表达式得以流行的主要原因。Perl 语言中的 m// 和 s/// 操作符所使用的正则表达式和替代文本流派在本书中被称作“Perl”流派。本书涉及的 Perl 版本包括 5.6、5.8 和 5.10。

### Python

Python 通过它的 re 模块来支持正则表达式。该模块所使用的正则表达式和替代文本流派在本书中被称作“Python”。本书涉及的 Python 版本是 2.4 和 2.5。

### Ruby

Ruby 拥有对于正则表达式的内置支持。本书会讲到 Ruby 1.8 和 1.9 版。Ruby 的这两个版本拥有不同的默认正则表达式引擎。Ruby 1.9 使用的是 Oniguruma 引擎，它同在 Ruby 1.8 中使用的经典引擎相比拥有更多的正则表达式特性。第 1 章中的“本书涉及的正则流派”小节中对此有更加详细的解释。

在本章中，我们不会过多谈到在 Ruby 1.8 和 1.9 之间的区别。本章中讲到的正则表达式都是非常基本的，因此它们并不会使用在 Ruby 1.9 中引入的新特性。因为正则表达式支持是直接编译到 Ruby 语言中的，所以不管是用经典的正则引擎还是 Oniguruma 引擎来编译 Ruby，你用来实现正则表达式的 Ruby 代码都是一样的。如果需要 Oniguruma 引擎中的特性的话，那么你也可以使用它来重新编译 Ruby 1.8。

## 其他编程语言

虽然本书并不会讲到在下面列出的这些编程语言，但是它们也会使用本书中的某一种正则表达式流派。如果你使用如下的一种语言，那么可以略去本章不读，但是其他所

有章节还会是有用的。

### ActionScript

ActionScript 是 Adobe 所实现的 ECMA-262 标准。从 3.0 版开始，ActionScript 完全支持 ECMA-262v3 所描述的正则表达式。这个正则流派在本书中被称作“JavaScript”。ActionScript 语言也与 JavaScript 非常接近。你应当能够把本章中给出的 JavaScript 示例修改之后用于 ActionScript。

## C

C 可以使用多种不同的正则表达式函数库。在本书中讲解的所有流派中，开源的 PCRE 函数库可能会是最好的选择。可以从 <http://www.pcre.org> 下载该函数库全部的 C 源代码。它的代码支持多种不同平台之上的各种编译器。

## C++

C++也可以使用多种不同的正则表达式函数库。在本书中讲解的所有流派中，开源的 PCRE 函数库可能会是最好的选择。你可以选择使用 C 的 API，或者使用在 PCRE 下载中包含的 C++类包装（参见 <http://www.pcre.org>）。

在 Windows 上，也可以导入 VBScript 5.5 RegExp COM 对象，这会在后面的 Visual Basic 6 中加以解释。这样做可能有助于保持在 C++后端和 JavaScript 前端之间的一致性。

### Delphi for Win32

在本书写作的时候，Delphi 的 Win32 版本还不包含对于正则表达式的内置支持。但是有许多 VCL 构件可以用来提供正则表达式支持。我们推荐你选择一种基于 PCRE 的构件。Delphi 拥有把 C 的 object 文件链接到应用程序中的能力，而且许多 PCRE 的 VCL 包装都使用的是这样的 object 文件。这就允许你可以使应用依然保持为单个的 .exe 文件。

可以从 <http://www-regexp.info/delphi.html> 下载作者的 TPerlRegEx 构件。这个 VCL 构件会把自己安装到构件面板上，从而可以很容易地把它拖到一个窗体中。另外一个比较流行的 Delphi 的 PCRE 包装库是 JCL 函数库中的 TJclRegEx 类 (<http://www.delphi-jedi.org>)。TJclRegEx 是从 TObject 实现来的，所以不能把它拖放到一个窗体中。

上述两个函数库都是开源的（采用的是 Mozilla Public License）。

### Delphi Prism

在 Delphi Prism 中，你可以使用由 .NET 框架所提供的正则表达式支持。只需要在你想要使用正则表达式的任意 Delphi Prism 单元中的 uses 从句中添加 System.

`Text.RegularExpressions` 即可。

完成上述动作之后，你就可以使用本章中给出的在 C# 和 VB.NET 代码片段中所使用的完全一样的技术。

#### Groovy

可以通过 `java.util.regex` 包来在 Groovy 中使用正则表达式，用法与在 Java 中完全一样。事实上，在本章中所给的所有 Java 解答都能够在 Groovy 中使用。Groovy 自己的正则表达式语法只是简单地提供了一些简写记号。可以自由地混合使用 Groovy 语法和标准的 Java 语法——它们的类和对象都是完全一样的。

#### PowerShell

PowerShell 是 Microsoft 提供的 shell 脚本语言，它基于的是 .NET 框架。PowerShell 内置的 `-match` 和 `-replace` 操作符使用的是本书中讲解的 .NET 正则流派和替代文本流派。

#### R

R 语言通过在 `base` 包中的 `grep`、`sub` 和 `regexpr` 函数提供对正则表达式的支持。所有这些函数都会接受一个标记为 `perl` 的参数，它的默认值是 `FALSE`。如果要使用本书中介绍的 PCRE 正则流派，那么就需要把它设置为 `TRUE`。本书中为 PCRE 7 给出的正则表达式可以用于 R 2.5.0 以及之后的版本。至于更早版本的 R，需要使用在书中标记为“PCRE 4 和更早版本”的正则表达式。R 所支持的“basic”和“extended”流派是本书中没有讨论的较早的、使用范围较小的正则流派。

#### REALbasic

REALbasic 拥有一个内置的 `RegEx` 类。这个类的内部使用的是 UTF-8 版本的 PCRE 函数库。这就意味着你可以使用 PCRE 的 Unicode 支持，但是在把它传送给 `RegEx` 类之前，还必须使用 REALbasic 的 `TextConverter` 类来把非 ASCII 文本转换为 UTF-8。

本书给出的 PCRE 6 的所有正则表达式都能够用于 REALbasic。需要注意的一点是，在 REALbasic 中，“不区分大小写”和“脱字符和美元符号匹配换行处”（也就是“多行”）选项是默认打开的。如果你想要使用本书中的一个正则表达式，而它并没有告诉你需要打开这些匹配模式的话，你就必须记住要在 REALbasic 中把相应选项关闭。

#### Scala

Scala 通过 `scala.util.matching` 包来提供内置的正则表达式支持。这个支持包是基于 `java.util.regex` 包的正则表达式引擎所开发的。在本书中，Java 和 Scala 所使用的正则表达式和替代文本流派都被称作“Java”。

## Visual Basic 6

Visual Basic 6 是不需要使用 .NET 框架的最后一个 Visual Basic 版本。这也就意味着 Visual Basic 6 不能使用在 .NET 框架中所提供的优秀的正则表达式支持。在本书中给出的 VB.NET 代码示例完全不可能在 Visual Basic 6 中使用。

Visual Basic 6 让用户可以很容易使用 ActiveX 和 Com 函数库所提供的功能。其中一个这样的库是 Microsoft 的 VBScript 脚本库，其中从 5.5 版之后就包含了较好的正则表达式能力。这个脚本库实现了与 JavaScript 中相同的正则表达式流派，也就是在 ECMA-262v3 所规定的标准。这个库包含在 Internet Explorer 5.5 以及之后的版本中。它可以在所有运行 Windows XP 或者 Vista 的机器上找到，对于较早版本的 Windows，如果用户升级到了 IE 5.5 或者更新版本也可以找到。这几乎就包括了所有目前连接到 Internet 上的 Windows 个人计算机。

如果想要在你的 Visual Basic 程序中使用这个库，在 VB IDE 的菜单中选择 Project|References。向下翻页找到 Microsoft VBScript RegularExpressions 5.5，该选项位于 Microsoft VBScript Regular Expressions 1.0 之下。这里提供 1.0 版本只是为了向后兼容的目的，因为它的功能已经很弱了。

在做了如上选择之后，你就可以看到该脚本库所提供的类和类成员。在菜单中选择 View|Object Browser。在对象浏览器（Object Browser）中，就可以从左上角的下拉菜单中选择 VBScript\_RegExp\_55 库。

## 3.1 在源代码中使用字面正则表达式

### 问题描述

给定一个正则表达式 `<[$"\n\d\\]>`，它是某个题目的解答。该正则表达式包含单个的字符类，它会匹配一个美元符号、一个双引号、一个单引号、一个换行符、0~9 的任意数字、一个正斜杠或者一个反斜杠。尝试把该正则表达式硬编码到你的源代码中，使之成为一个字符串常量，或者是正则表达式的操作符。

### 解决方案

#### C#

作为一个正常字符串：

```
"[$\"'\n\\d/\\\\]"
```

作为一个逐字字符串：

```
@"[$\"'\n\\d/\\\\]"
```

## VB.NET

```
"[$"'\n\d/\\\"]"
```

## Java

```
"[$\"'\n\\d/\\\\\\"]"
```

## JavaScript

```
/[$"'\n\d/\\\\"]/
```

## PHP

```
'%[$"'\n\d/\\\\\\"]%'
```

## Perl

模式匹配操作符：

```
/[$"'\n\d/\\\"]/  
m![$"'\n\d/\\\"]!
```

替代操作符：

```
s![$"'\n\d/\\\"]!!
```

## Python

三引号的 Raw 字符串：

```
r"""[$"'\n\d/\\\"]"""
```

正常字符串：

```
"[$"'\n\\d/\\\\\\"]"
```

## Ruby

使用正斜杠分隔的字面正则表达式：

```
/[$"'\n\d/\\\"]/
```

使用你选择的标点符号来分隔的字面正则表达式：

```
%r![$"'\n\d/\\\"]!
```

## 讨论

当本书向读者展示一个正则表达式自身的时候（而不是作为更大的代码片段中的一部分），它总是按照没有进行任何附加修饰的方式来进行展示的。这个实例是唯一的例外。

如果你在使用一个像 RegexBuddy 或 RegexPal 这样的正则表达式测试工具，那么应当不用任何修饰来输入一个正则表达式。如果你的程序接受一个正则表达式作为用户输入，那么用户也应当这样来输入。

但是，如果想要把一个正则表达式硬编码到你的源代码中，那么就需要做额外的工作。如果从正则表达式测试工具中把正则表达式随意复制并粘贴到源代码中——或者反过来——常常都会使你抓耳挠腮，不知道为什么正则表达式在工具中执行正常，而在代码中却不然，抑或是为什么你从别人的代码中复制来的正则表达式在测试工具中却无法正常使用。本书中讨论的所有编程语言都要求把字面上的正则表达式以某种特定方式进行分隔，有些语言要求使用字符串，有些语言则会要求使用一个特殊的正则表达式常量。如果你的正则表达式包括了该语言的分隔符，或者在该语言中拥有特殊含义的其他字符，那么就必须对它们进行转义。

反斜杠是最为常用的转义字符。这也解释了为什么原始的正则表达式只包含 4 个反斜杠，而这个问题的大多数解答中都要包含更多反斜杠。

## C#

在 C# 中，你可以把一个字面正则表达式传递给 `Regex()` 构造函数，以及在 `Regex` 类中的其他成员函数。接受正则表达式的参数总是被声明为一个字符串 (`string`)。

C# 支持两种字符串字面量。最常见的一种是双引号字符串，这种形式在 C++ 和 Java 语言中最为常见。在双引号字符串之内，双引号和反斜杠必须使用一个反斜杠来进行转义。在字符串中同时也支持对于不可打印字符的转义，例如 `\n`。在使用 `RegexOptions.IgnorePatternWhitespace` 来打开宽松排列模式的时候（实例 3.4），"`\n`" 和 "`\n`" 是有区别的。宽松排列模式在实例 2.18 中讲解过。`\n` 是包含一个字面换行符的字符串，会被当作空白忽略。`\n` 则是包含正则表达式记号的一个字符串，它会匹配一个换行。

逐字字符串（`verbatim string`）以 @ 和双引号作为开始，并以一个双引号作为结束。如果要在一个逐字字符串中包含一个双引号，那么就需要用两个双引号来表示。反斜杠不需要进行转义，因此会得到一个显然更加易读的正则表达式。`@"\n"` 总是代表正则表达式记号 `\n`，它会匹配一个换行，即使在宽松排列模式中也是如此。虽然逐字字符串并不在字符串的层次上支持 `\n`，但是它可以跨多行。这样就使得逐字字符串用于宽松排列模式下的正则表达式最为理想。

很显然，你应当使用逐字字符串把正则表达式插入到 C# 源代码中。

## VB.NET

在 VB.NET 中，可以把字面正则表达式传递给 `Regex()` 构造函数，以及在 `Regex` 类中的其他成员函数。接受正则表达式的参数总是被声明为一个字符串。

Visual Basic 使用的是双引号字符串。在字符串之内，双引号必须使用两个双引号来表示。所有其他字符都不需要转义。

## Java

在 Java 中，可以把字面正则表达式传递给 `Pattern.compile()` 类厂（class factory），以及在 `String` 类中的各种其他函数。接受正则表达式的参数总是被声明为一个字符串。

Java 使用双引号字符串。在双引号字符串之内，双引号和反斜杠必须使用一个反斜杠来进行转义。在字符串中同时也支持对于不可打印字符（例如`\n`）的转义，以及 Unicode 转义序列（例如`\uFFFF`）。

在使用 `Pattern.COMMENTS` 来打开宽松排列模式的时候（实例 3.4），“`\n`”和“`\n`”是有区别的。宽松排列模式已经在实例 2.18 中进行了讲解。“`\n`”是包含一个字面换行符的字符串，会被当作空白忽略。而“`\n`”则是包含正则表达式记号的一个字符串，它会匹配一个换行。

## JavaScript

在 JavaScript 中，创建正则表达式最好的方式是使用声明字面正则表达式的特殊语法。你可以简单地把正则表达式放到两个正斜杠之间。如果在正则表达式之内包含任何正斜杠，那么就需要使用反斜杠来对它进行转义。

虽然使用字符串也可以创建一个 `RegExp` 对象，但是在你的代码中使用字面正则表达式的字符串记法并没有任何意义。因为必须要对引号和反斜杠进行转义，这样通常会导致一大堆的反斜杠出现。

## PHP

在 PHP 的 `preg` 函数类中使用的字面正则表达式则需要小心翼翼地处理。与 JavaScript 和 Perl 不同，PHP 自己并不拥有一种内置的正则表达式类型。正则表达式必须总是像字符串一样用引号引起来。对于 `ereg` 和 `mb_ereg` 函数类也是如此。但是，在尽量模仿 Perl 的过程中，PHP 的包装函数开发者添加了一个额外的要求。

在字符串中，正则表达式必须按照 Perl 风格的字面正则表达式引起来。这意味着如果在 Perl 中你写的是`/regex/`，那么在 PHP 的 `preg` 函数类中就变成了`'/regex/'`。同在 Perl 中一样，可以使用任意的标点符号对作为分隔符。如果正则表达式分隔符出现在正则表达式中的话，那么它就必须使用反斜杠来转义。为了避免这种情形，可以选择一个不会在正则表达式中出现的分隔符。对于本实例来说，我们使用了百分号，因为在正则表达式中会出现正斜杠，但是不会出现百分号。如果正斜杠不在该正则表达式中出现，那么最好使用正斜杠，因为它是在 Perl 中最为常用的分隔符，同时也是在 JavaScript 和 Ruby 中必须使用的分隔符。

PHP 同时支持单引号和双引号字符串。二者都要求把在正则式之内的引号（单引号和双引号）与反斜杠都用反斜杠进行转义。在双引号字符串中，美元符号也需要转义。对于正则表达式来说，你应当使用单引号字符串，除非真的需要在正则表达式中插入变量的值。

## Perl

在 Perl 中，字面正则表达式可以使用在模式匹配操作符和替换操作符中。模式匹配操作符包含两个正斜杠，正则表达式就出现在它们之间。在正则表达式之内的正斜杠必须使用反斜杠进行转义。并不需要再对其他任何字符进行转义（\$和@可能会例外，这会在本小节最后进行解释）。

对于模式匹配操作符来说，另外一种可选的方式是把正则表达式放到任意一对标点符号之间，前面需要一个前导的字母 m。如果使用任意的起始和结束标点（圆括号、花括号或方括号）的话，那么它们需要成对出现：例如 m{regex}。如果使用的是其他标点，那么可以使用同一个字符两次。在这个实例的解答中使用的是感叹号。这样就可以不必对正则表达式中的字面正斜杠进行转义。如果起始和结束分隔符不一样的话，那么只有结束分隔符作为字面字符出现在正则表达式中的时候，才需要对它使用反斜杠进行转义。

替换操作符与模式匹配操作符类似。它使用 s 开头，而不是 m，另外其中还需要用到替代文本。当使用括号或者类似标点作为分隔符的时候，你需要 2 对标点：s[regex] [replace]。对于其他所有的标点，则需要出现 3 次：s/regex/replacement/。

Perl 会把模式匹配和替换操作符当作双引号字符串来进行分析。如果你写的是 m/I am \$name/，而\$name 中保存的是"Jan"，那么最终会得到的正则表达式是<I•am•Jan>。在 Perl 中，\$”也是一个变量，所以在本实例的正则表达式中，必须对字符类中的字面美元符号进行转义。

如果想要把美元符号当作定位符（参见实例 2.5）使用，那么永远也不要对它进行转义。一个转义的美元符号总是代表一个字面量。Perl 采用了比较聪明的处理方式，可以区分用作定位符的美元符号和用作变量插值的美元符号，这是由于定位符一般来说只会用于一个分组，或者整个正则式的末尾，或者一个换行符之前。在<m/^regex\$/>中，如果想要检查是否 “regex” 会匹配整个目标字符串，那么就不应当对其中的美元符号进行转义。

在正则表达式中，@符号并不拥有特殊含义，但是它在 Perl 中会被用于变量插值。在 Perl 代码中，你需要对字面正则表达式之中的@符号进行转义，就像在双引号字符串中一样。

## Python

Python 的 re 模块中的函数期望字面正则表达式被当作字符串来进行传递。你可以使用

Python 提供的各种不同方式来把字符串引起来。根据在正则表达式中出现的字符的不同，采用不同的引号方式，可能会减少需要使用反斜杠来进行转义的字符数量。

通常来说，raw 字符串是最好的选择。Python 中的 raw 字符串并不要求对任何字符进行转义。如果使用的是 raw 字符串，那么就不要对你的正则表达式中出现的反斜杠都使用加倍表示。r"\d+"读起来肯定比"\\"d+"更为容易，特别是当正则式变得很长的时候尤为如此。

Raw 字符串唯一不是很理想的情形是当你的正则表达式中同时包括单引号和双引号字符时。这时，无法使用以一对单引号或者双引号来分隔的 raw 字符串，因为你无法对正则表达式之内的引号进行转义。在这种情形下，可以对 raw 字符串使用三引号，就像我们在前面给出的 Python 解答中一样。为了比较起见，我们在解答中同时给出了正常字符串的形式。

如果想要在正则表达式中使用在实例 2.7 中讲解的 Unicode 特性，那么就需要使用 Unicode 字符串。你可以在一个字符串之前添加一个 u 来把它变成 Unicode 字符串。

Raw 字符串并不支持对像\n这样的不可打印字符进行转义。Raw 字符串会把转义序列当作字面文本。这对于 re 模块来说并不是问题。它支持把这些转义作为正则表达式语法的一部分，以及替代文本语法的一部分。在一个 raw 字符串中的字面的 \n 在你的正则表达式和替代文本中还是会被解释成一个换行符。

在使用 re.VERBOSE 打开宽松排列模式的时候（实例 3.4），字符串"\n"，与字符串"\\\n" 和 raw 字符串 r"\n"是有区别的。宽松排列模式已经在实例 2.18 中进行了讲解。"\n"是包含一个字面换行符的字符串，会被当作空白忽略。而"\\n"和 r"\n"则是包含正则表达式记号的一个字符串，它会匹配一个换行。

当使用宽松排列模式的时候，三引号 raw 字符串，例如 r"""\n"""\n是最好的解决方案，因为它们可以跨多行。同样，\n不会在字符串的层次被解释，因此它可以在正则表达式的层次被解释来匹配一个换行符。

## Ruby

在 Ruby 中，创建正则表达式的最好方式是使用声明字面正则表达式的特殊语法。你可以简单地把正则表达式放到两个正斜杠之间。如果在正则表达式之内包含任何正斜杠，那么就需要使用反斜杠来对它进行转义。

如果不想在正则表达式中对正斜杠进行转义，那么可以在正则表达式前面加上前缀%r，然后就可以使用你所选择的任意标点符号作为分隔符。

虽然使用字符串也可以创建一个 Regexp 对象，但是在代码中使用字面正则表达式的字符串记法并没有任何意义。因为你必须要对引号和反斜杠进行转义，这样通常会导致一大堆反斜杠的出现。



在这一点上，Ruby 和 JavaScript 非常相似，唯一的区别是在 Ruby 中使用的类名是作为一个单词的 Regexp，而在 JavaScript 中使用的类名则是两个词首字母大写的形式 RegExp。

## 参见

实例 2.3 讲解了字符类是如何工作的，以及为什么在正则表达式中需要两个反斜杠来包含字符类中的一个反斜杠。

实例 3.4 讲解了如何设置正则表达式选项，在有些语言中这是通过字面正则表达式的一部分来实现的。

## 3.2 导入正则表达式函数库

### 问题描述

要想在应用程序中能够使用正则表达式，就需要把正则表达式函数库或命名空间导入到源代码中。



在本节之后的代码片段中，都会假设如果有需要的话，你已经导入了相应的函数库。

### 解决方案

#### C#

```
using System.Text.RegularExpressions;
```

#### VB.NET

```
Imports System.Text.RegularExpressions
```

#### Java

```
import java.util.regex.*;
```

#### Python

```
import re
```

### 讨论

有些编程语言包含了内置的正则表达式。对于这些语言来说，你不再需要做任何事情

来使之支持正则表达式。其他语言会通过一个函数库来提供正则表达式的功能，这个库就需要在源代码中使用一个 `import` 语句来导入。有些语言根本不提供正则表达式支持。对于这些语言，你不得不自己设法把正则表达式支持编译和链接进来。

## C#

如果把 `using` 语句添加到 C# 源文件的开始处，那么你就可以直接引用提供正则表达式功能的类，而不必再去使用完全限定的名称。例如，你可以使用 `Regex()`，而不必非要使用 `System.Text.RegularExpressions.Regex()`。

## VB.NET

如果把 `Imports` 语句添加到 VB.NET 源文件的开始处，那么你就可以直接引用提供正则表达式功能的类，而不必再去使用完全限定的名称。例如，可以使用 `Regex()`，而不必非要使用 `System.Text.RegularExpressions.Regex()`。

## Java

要想能够使用 Java 内置的正则表达式函数库，就必须把 `java.util.regex` 包导入到你的应用程序中。

## JavaScript

JavaScript 的正则表达式是内置的，因此总是可用的。

## PHP

在 PHP 4.2.0 版本之后，`preg` 函数是内置的，因此总是可用的。

## Perl

Perl 的正则表达式是内置的，因此总是可用的。

## Python

要想使用 Python 的正则表达式功能，就必须把 `re` 模块导入你的脚本中。

## Ruby

Ruby 的正则表达式是内置的，因此总是可用的。

## 3.3 创建正则表达式对象

### 问题描述

你想要实例化一个正则表达式对象，或者是编译一个正则表达式，从而可以在应用程

序中更为有效地使用它。

## 解决方案

### C#

如果你知道该正则表达式是正确的：

```
Regex regexObj = new Regex("regex pattern");
```

如果该正则表达式是由最终用户提供的（`UserInput` 是一个字符串变量）：

```
try {
    Regex regexObj = new Regex(UserInput);
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

### VB.NET

如果你知道该正则表达式一定是正确的：

```
Dim RegexObj As New Regex("regex pattern")
```

如果该正则表达式是由最终用户提供的（`UserInput` 是一个字符串变量）：

```
Try
    Dim RegexObj As New Regex(UserInput)
Catch ex As ArgumentException
    'Syntax error in the regular expression
End Try
```

### Java

如果你知道该正则表达式一定是正确的：

```
Pattern regex = Pattern.compile("regex pattern");
```

如果该正则表达式是由最终用户提供的（`UserInput` 是一个字符串变量）：

```
try {
    Pattern regex = Pattern.compile(userInput);
} catch (PatternSyntaxException ex) {
    // Syntax error in the regular expression
}
```

要想在一个字符串上能够使用该正则表达式，那么就需要创建一个 `Matcher`：

```
Matcher regexMatcher = regex.matcher(subjectString);
```

要在另外一个字符串上也可以使用该正则式，那么你可以按照上面的方法创建一个新的 `Matcher`，或者也可以复用已有的 `Matcher`：

```
regexMatcher.reset(anotherSubjectString);
```

## JavaScript

在你的代码中的字面正则表达式：

```
var myregexp = /regex pattern/;
```

从用户输入获取的正则表达式，作为一个字符串存储在变量 userinput 中：

```
var myregexp = new RegExp(userinput);
```

## Perl

```
$myregex = qr/regex pattern/
```

从用户输入获取的正则表达式，作为一个字符串存储在变量\$userinput 中：

```
$myregex = qr/$userinput/
```

## Python

```
reobj = re.compile("regex pattern")
```

从用户输入获取的正则表达式，作为一个字符串存储在变量 userinput 中：

```
reobj = re.compile(userinput)
```

## Ruby

在你的代码中的字面正则表达式：

```
myregexp = /regex pattern/;
```

从用户输入获取的正则表达式，作为一个字符串存储在变量 userinput 中：

```
myregexp = Regexp.new(userinput);
```

## 讨论

在正则表达式引擎能把正则表达式匹配到一个字符串之前，正则表达式首先需要被编译。这个编译的动作是在你的应用程序运行过程中完成的。正则表达式构造函数或者编译功能会分析包含你的正则表达式的字符串，并把它转换成一个树形结构或者是状态机。执行实际的模式匹配的函数则会在扫描该字符串的同时遍历该树或者自动机。支持字面正则表达式的编程语言会在执行过程到达这个正则表达式操作符的时候进行编译。

## .NET

在 C# 和 VB.NET 中，.NET 的 System.Text.RegularExpressions.Regex 类中会包含一个编

译好的正则表达式。最简单的构造函数只接受一个参数，也就是包含你的正则表达式的一个字符串。

如果在正则表达式中存在一个语法错误的话，那么 `Regex()` 构造函数会产生一个 `ArgumentException` 例外。这个例外消息会说明到底是遇到了哪个错误。如果该正则表达式是由程序的用户提供的话，那么就很有必要捕获这个例外。把例外消息显示给用户，然后要求用户对正则表达式进行修改。如果你的正则表达式是一个硬编码的字符串常量，那么可以忽略捕获例外的过程，但是你需要使用代码覆盖工具来保证该行代码会正常执行，而不会产生例外。你不能通过修改状态或者模式来使同一个字面正则表达式在一种情形下可以编译，而在另外一种情形下会编译失败。注意，如果你的字面正则表达式中存在语法错误，那么它会在你的程序运行时产生例外，而不是在程序编译的过程中。

如果你会在一个循环内部，或者在整个程序中会重复使用某个正则表达式，那么应当构造一个 `Regex` 对象。构造一个正则表达式对象并不会带来额外的开销。因为不管你 how 使用，把该正则式作为字符串参数的 `Regex` 类的静态成员总是会在内部构造一个 `Regex` 对象，所以完全可以在你自己的代码中进行构造，这样还可以保留一个到该对象的引用。

如果只计划使用该正则式一次或者少数几次，那么你可以使用 `Regex` 类的静态成员，这样可以少写几行代码。静态的 `Regex` 成员并不会立即丢弃在内部构造的正则表达式对象；事实上，它们会在缓存中保留最近使用过的 15 个正则表达式。你可以通过设置 `Regex.CacheSize` 属性来修改缓存的大小。但是不要过于依赖这个缓存。如果需要频繁使用许多正则对象的话，那么还是需要保存一份自己的缓存，这样检索效率才会比字符串搜索更为高效。

## Java

在 Java 语言中，`Pattern` 类中会包含一个编译好的正则表达式。你可以使用 `Pattern.compile()` 类厂来创建该类的对象，它只需要一个参数，即包含你的正则表达式的一个字符串。

如果在正则表达式中存在一个语法错误，那么 `Pattern.compile()` 类厂会产生一个 `PatternSyntaxException` 例外。这个例外消息会说明到底是遇到了哪个错误。如果该正则表达式是由程序的用户提供的话，那么就很有必要捕获这个例外。把例外消息显示给用户，然后要求用户对正则表达式进行修改。如果你的正则表达式是一个硬编码的字符串常量，那么可以忽略捕获例外的过程，但是你需要使用代码覆盖工具来保证该行代码会正常执行，而不会产生例外。你不能通过修改状态或者模式来使同一个字面正则表达式在一种情形下可以编译，而在另外一种情形下会编译失败。注意，如果你的字面正则表达式中存在语法错误，那么它会在你的程序运行时产生例外，而不是在程

序编译的过程中。

除非你计划只使用这个正则表达式一次，否则都应当创建一个 `Pattern` 对象，而不是使用 `String` 类的静态成员。虽然可能会需要多写几行代码，但是它运行起来会更加高效。静态的调用会在每一次都重新编译你的正则表达式。事实上，Java 只对很少几个非常基本的正则任务才提供静态调用。

一个 `Pattern` 对象只会保存一个编译好的正则表达式，它并不会做任何实际的工作。实际上的正则匹配是由 `Matcher` 类来完成的。如果想要创建一个 `Matcher` 对象，就需要使用编译好的正则表达式来调用 `matcher()` 函数。把目标字符串作为 `matcher()` 的唯一参数。

你可以调用 `matcher()` 任意多次，从而可以使用同一个正则表达式来匹配多个字符串。你也可以同时使用同一个正则式的多个匹配器（`matcher`），只要把所有的工作都限制在单个线程中即可。`Pattern` 和 `Matcher` 这两个类都不是线程安全的。如果想要在多个线程中使用同一个正则表达式，那么就要在每个线程中分别调用 `Pattern.compile()`。

如果在把一个正则式应用到一个字符串之后，还想要把同一个正则式应用到另外一个字符串，那么你可以通过调用 `reset()` 函数来重复使用 `Matcher` 对象。把下一个目标字符串作为唯一参数传递给该函数。这比创建一个新的 `Matcher` 对象的效率更高。`reset()` 会返回你所调用的同一个 `Matcher`，并且允许你很容易地只使用一行代码就可以完成重置并且接着使用该匹配器，例如：`regexMatcher.reset(nextString).find()`。

## JavaScript

在实例 3.2 中给出的字面正则表达式的记法已经创建了一个新的正则表达式对象。要想重复使用同一个对象，那么就可以简单地把它赋给一个变量。

如果有一个正则表达式存储在一个字符串变量中（比如，因为你要求用户输入一个正则表达式），那么可以使用 `RegExp()` 构造函数来编译这个正则表达式。注意在字符串之内的正则表达式并没有使用正斜杠来进行分隔。这些斜杠是属于 JavaScript 为字面的 `RegExp` 对象使用的记法，而不属于正则表达式自身的一部分。



由于把字面正则表达式赋给一个变量是很简单的，因此在本章中绝大多数的 JavaScript 解答都略掉了这行代码，直接使用字面的正则表达式。在你自己的代码中，当多次使用同一个正则表达式的时候，你应当把该正则表达式赋给一个变量，然后再使用该变量，而不是每次都把同一个字面正则表达式复制到你的代码中。这样不仅可以提高性能，还能使你的代码更加容易维护。

## PHP

PHP 并没有提供把编译好的正则表达式保存到一个变量中的方法。每当你想要使用一

个正则表达式的时候，就必须把它作为一个字符串来传递给其中的一个 preg 函数。

preg 函数类会保存包含最多 4096 个编译好的正则表达式的一个缓存。虽然基于哈希的缓存查找并不像访问一个变量那么快，但是与每次都重新编译同一个正则表达式相比，它对性能的影响并不是很大。当缓存已满时，最早被编译的正则表达式会被移除。

## Perl

你可以使用 “*quote regex*” 操作来编译一个正则表达式，并把它赋给一个变量。它使用的语法与在实例 3.1 中讲解的匹配操作符是一样的，唯一的区别是它以两个字母 qr 开始，而不是字母 m 开头。

在重复使用较早编译好的正则表达式的时候，Perl 通常是比较高效的。因此，在本章的代码示例中，我们并没有使用 qr//。只有实例 3.5 中演示了它的使用方法。

当你要在正则表达式中插入变量，或者是当你把整个正则表达式当作字符串来获取的时候（比如是从用户输入获得），qr//会很有用。使用 qr/\$regexstring/，可以控制正则表达式什么时候被重新编译，以反映\$regexstring 的新内容。m/\$regexstring/会每次都重新编译该正则表达式，而 m/\$regexstring/o 永远不会进行重新编译。实例 3.4 中会解释/o 的含义。

## Python

在 Python 的 re 模块中的 compile()函数会接受一个包含你的正则表达式的字符串，然后返回包含编译好的正则表达式的一个对象。

如果你计划反复使用同一个正则表达式，那么应当显式地调用 compile()函数。在 re 模块中的所有函数都会首先调用 compile()，然后再在编译好的正则表达式对象之上调用你想要用的函数。

compile()函数会保存它编译的最后 100 个正则表达式的引用列表。这会把对于最后 100 个使用过的正则表达式的重新编译过程都降级为一个词典查找动作。当缓存已满时，缓存会被全部清除。

如果性能不是问题的话，那么缓存会工作得足够好，从而你可以直接使用 re 模块中的函数。但是如果性能有问题，那么就应当选择调用 compile()。

## Ruby

在实例 3.2 中给出的字面正则表达式的记法已经创建了一个新的正则表达式对象。要想重复使用同一个对象，那么就可以简单地把它赋给一个变量。

如果有一个正则表达式存储在一个字符串变量中（比如，因为你要求用户输入一个正

则表达式), 那么可以使用 `Regexp.new()` 类厂或者它的同义函数 `Regexp.compile()` 来编译这个正则表达式。注意在字符串之内的正则表达式并没有使用正斜杠来进行分隔。这些斜杠只属于 Ruby 为字面的 `Regexp` 对象使用的记法, 而不属于正则表达式自身的一部分。



由于把字面正则表达式赋给一个变量是很简单的, 因此在本章中绝大多数的 Ruby 解答都略掉了这行代码, 直接使用字面的正则表达式。在你自己的代码中, 当多次使用同一个正则表达式的时候, 你应当把该正则表达式赋给一个变量, 然后使用该变量, 而不是每次都把同一个字面正则表达式复制到你的代码中。这样不仅可以提高性能, 还能使你的代码更加容易维护。

## 把正则表达式编译为 CIL

### C#

```
Regex regexObj = new Regex("regex pattern", RegexOptions.Compiled);
```

### VB.NET

```
Dim RegexObj As New Regex("regex pattern", RegexOptions.Compiled)
```

## 讨论

当你在 .NET 中不传递任何选项来构造一个 `Regex` 对象的时候, 正则表达式会按照我们在本实例前面的“讨论”小节中描述的方式来进行编译。如果你把 `RegexOptions.Compiled` 作为第二个参数传递给 `Regex()` 构造函数, `Regex` 类会采用一种不同的方式: 它会把你的正则表达式编译为 CIL, 或者也被称作 MSIL。CIL 指的是通用中间语言 (Common Intermediate Language), 它是一种低级语言, 与 C# 和 Visual Basic 相比, 它离汇编语言更近一些。所有 .NET 编译器都会产生 CIL。你的应用程序第一次运行的时候, .NET 框架会把 CIL 继续编译成适合用户计算机的机器代码。

使用 `RegexOptions.Compiled` 来编译正则表达式的好处是, 它的运行速度比不使用这个选项来编译正则表达式快 10 倍以上。缺点是编译过程本身会比把正则式字符串分析成一棵树要慢两个数量级。CIL 代码也会成为程序中持久的一部分, 直到程序运行结束。CIL 代码不会被垃圾收集。

只有在一个正则表达式要么非常复杂, 要么需要处理非常多的文本, 从而用户在使用该正则表达式操作的过程中会感受到明显停顿的时候, 我们才推荐使用 `RegexOptions.Compiled`。如果正则表达式在一刹那间就能够完成任务, 那么就不值得花费这么多时间来进行编译和汇编。

## 参见

实例 3.1、3.2 和 3.4。

## 3.4 设置正则表达式选项

### 问题描述

你想要使用所有可用的匹配模式来编译一个正则表达式：宽松排列、不区分大小写、点号匹配换行符以及脱字符和美元符号匹配换行处。

### 解决方案

#### C#

```
Regex regexObj = new Regex("regex pattern",
    RegexOptions.IgnorePatternWhitespace | RegexOptions.IgnoreCase |
    RegexOptions.Singleline | RegexOptions.Multiline);
```

#### VB.NET

```
Dim RegexObj As New Regex("regex pattern",
    RegexOptions.IgnorePatternWhitespace Or RegexOptions.IgnoreCase Or
    RegexOptions.Singleline Or RegexOptions.Multiline)
```

#### Java

```
Pattern regex = Pattern.compile("regex pattern",
    Pattern.COMMENTS | Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE |
    Pattern.DOTALL | Pattern.MULTILINE);
```

#### JavaScript

代码中的字面正则表达式：

```
var myregexp = /regex pattern/im;
```

从用户输入获取的正则表达式（作为一个字符串 userinput）：

```
var myregexp = new RegExp(userinput, "im");
```

#### PHP

```
regexstring = '/regex pattern/simx';
```

#### Perl

```
m/regex pattern/simx;
```

## Python

```
reobj = re.compile("regex pattern",
    re.VERBOSE | re.IGNORECASE |
    re.DOTALL | re.MULTILINE)
```

## Ruby

代码中的字面正则表达式：

```
myregexp = /regex pattern/mix;
```

从用户输入获取的正则表达式（作为一个字符串 userinput）：

```
myregexp = Regexp.new(userinput,
    Regexp::EXTENDED or Regexp::IGNORECASE or
    Regexp::MULTILINE);
```

## 讨论

本书中的许多正则表达式，以及你在其他地方看到的正则表达式，都需要使用特定的正则匹配模式。几乎所有现代正则流派都会支持这 4 种基本模式。不幸的是，对于实现这些模式的选项，有一些流派使用了不一致的和让人容易混淆的名称。使用错误的模式通常会破坏一个正则表达式。

在这个实例中的所有解答会使用编程语言或者正则表达式类所提供的标志或选项来设置不同的模式。设置模式的另外一种方式是在正则表达式之内使用模式修饰符。在正则表达式之内的模式修饰符总是会覆盖在正则表达式之外设置的选项或标志。

## .NET

Regex()构造函数会接受一个可选的第二个参数，用来指定正则表达式选项。你可以在 RegexOptions 枚举体中找到可用的选项。

宽松排列： RegexOptions.IgnorePatternWhitespace  
不区分大小写： RegexOptions.IgnoreCase  
点号匹配换行符： RegexOptions.Singleline  
脱字符和美元符号匹配换行处： RegexOptions.Multiline

## Java

Pattern.compile()方法会接受一个可选的第二个参数，用来指定正则表达式选项。Pattern 类中定义了几个用来设置不同选项的常量。把它们使用按位或 (bitwise inclusive or) 操作符组合起来，就可以设置多种不同选项。

宽松排列： Pattern.COMMENTS  
不区分大小写： Pattern.CASE\_INSENSITIVE | Pattern.UNICODE\_CASE

点号匹配换行符: Pattern.DOTALL

脱字符和美元符号匹配换行处: Pattern.MULTILINE

对于不区分大小写来说，实际上存在两个选项，而且必须全部设置才能做到完全不区分大小写。如果只设置了 Pattern.CASE\_INSENSITIVE，那么只有英语字母 A~Z 才会按照不区分大小写进行匹配。如果你两个都设置了，那么所有字母表中的所有字符都会按照不区分大小写来匹配。不使用 Pattern.UNICODE\_CASE 的唯一理由可能就是性能，前提是事先知道你只会处理 ASCII 文本。当在你的正则表达式内使用模式修饰符的时候，使用`(?i)`来表示只对 ASCII 不区分大小写，而用`(?iu)`来表示完全不区分大小写。

## JavaScript

在 JavaScript 中，你可以通过在 RegExp 字面量之后，也就是在正则表达式结束的正斜杠之后，添加一个或者多个单字母的标志，以指定正则选项。虽然标志本身只含有一个字母，但是当在文档中探讨这些标志的时候，它们通常会被写作是/i 和/m。在指定正则表达式模式标志时，则不需要额外的斜杠。

在使用 `RegExp()` 构造函数来把字符串编译成正则表达式的时候，你可以传递一个包含标志的可选的第二个参数给构造函数。这第二个参数应当是一个字符串，其中包含了你想要设置的选项的字母。不要在该字符串中添加任何斜杠。

宽松排列: JavaScript 不支持

不区分大小写: /i

点号匹配换行符: JavaScript 不支持

脱字符和美元符号匹配换行处: /m

## PHP

实例 3.1 中讲解了 PHP 的 `preg` 函数类要求字面正则表达式使用两个标点字符来进行分隔，通常是使用正斜杠，而所有内容都加起来会被格式化为一个字符串字面量。可以通过在该字符串末尾添加一个或者多个单字符的修饰符指定正则表达式选项。也就是说，修饰符字母会在终止的正则分隔符之后，但是依然位于字符串的单引号或双引号之内。虽然标志本身只含有一个字母，而且在正则式和修饰符之间的分隔符也不一定非要是一个正斜杠，但是当在文档中探讨这些标志的时候，它们通常还是会被写作/x。

宽松排列: /x

不区分大小写: /i

点号匹配换行符: /s

脱字符和美元符号匹配换行处: /m

## Perl

你可以通过在模式匹配或替换操作符末尾添加一个或者多个单字符的修饰符来指定正

则表达式选项。虽然标志本身只含有一个字母，而且在正则式和修饰符之间的分隔符也不一定非要是一个正斜杠，但是当在文档中探讨这些标志的时候，它们通常还是会写作/x。

宽松排列: /x  
不区分大小写: /i  
点号匹配换行符: /s  
脱字符和美元符号匹配换行处: /m

## Python

compile()函数（在上一个实例中讲解过）会接受一个可选的第二个参数，其中包含正则表达式选项。可以通过使用 | 操作符把在 re 模块中定义的常量组合起来，构造这个参数的值。在 re 模块中的许多其他接受字面正则表达式作为参数的函数同样也会接受正则表达式选项作为最后一个可选的参数。

正则表达式选项的常量是成对出现的。每个选项都可以使用一个完整名称的常量或者只是用单个字符来表示。它们的功能则是完全等价的。唯一的区别是，完整名称会使你的代码对于那些不是非常熟悉正则表达式选项字母的开发者来说更加容易阅读。在这个小节中列出的基本单字母选项同在 Perl 中是完全一样的。

宽松排列: re.VERBOSE 或 re.X  
不区分大小写: re.IGNORECASE 或 re.I  
点号匹配换行符: re.DOTALL 或 re.S  
脱字符和美元符号匹配换行处: re.MULTILINE 或 re.M

## Ruby

在 Ruby 中，你可以通过在 Regexp 字面量之后，也就是在正则表达式结束的正斜杠之后，添加一个或者多个单字母的标志，以指定正则选项。虽然标志本身只含有一个字母，但是当在文档中探讨这些标志的时候，它们通常会被写作是/i 和/m。在指定正则表达式模式标志时，则不需要额外的斜杠。

在使用 Regexp.new()类厂把字符串编译成正则表达式的时候，你可以传递一个包含标志的可选的第二个参数给构造函数。这第二个参数应当是：使用 nil 来关闭所有选项，或者是由 or 操作符结合起来的 Regexp 类中的常量组合。

宽松排列: /r 或 Regexp::EXTENDED  
不区分大小写: /i 或 Regexp::IGNORECASE  
点号匹配换行符: /m 或 Regexp::MULTILINE。Ruby 实际上在这里使用“m”和“multiline”，而所有其他流派都是用“s”或“single line”来表示“点号匹配换行符”。  
脱字符和美元符号匹配换行处: 在 Ruby 中，脱字符和美元符号总是会在换行处产生匹配。你无法关闭这个选项。使用<\A>和<\Z>来匹配目标字符串的开始或结尾。

## 语言相关的其他选项

### .NET

`RegexOptions.ExplicitCapture` 会把除了命名分组之外的所有分组都变成非捕获分组。使用这个选项之后，`\(group)` 就与 `\(?:group)` 完全一样了。如果你总是对捕获分组命名的话，那么打开这个选项会使你的正则表达式更加高效，而不需要使用 `\(?:group)` 这样的语法。除了使用 `RegexOptions.ExplicitCapture` 之外，还可以通过把 `\(?n)` 放在你的正则表达式的起始处来打开这个选项。实例 2.9 中讲解了分组的知识。实例 2.11 讲解了命名分组。

如果想要在 .NET 代码和 JavaScript 代码中使用相同的正则表达式，而且想要确保它们会按照相同方式工作，那么就需要指定 `RegexOptions.ECMAScript`。当你使用 JavaScript 来开发 web 应用的客户端，而使用 ASP.NET 来开发服务器端的时候，这个选项尤其有用。它所产生的最重要的效果就是，`\w` 和 `\d` 被限制为只能应用于 ASCII 字符，这与在 JavaScript 中是一样的。

### Java

对 Java 来说比较特别的一个选项是 `Pattern.CANON_EQ`，它会打开“规范等价 (canonical equivalence)”。我们在实例 2.7 中的“Unicode 字形”小节中已经讨论过，Unicode 提供了不同的方式来表示带有读音符号的字符。当你打开这个选项的时候，即使一个字符在目标字符串中的编码方式不同，正则表达式也可以匹配到该字符。举例来说，正则表达式 `\u00E0` 会匹配 `\u00E0` 和 `\u0061\u0300`，因为它们是规范等价的。它们在屏幕上都会以“à”的形式出现，用户并不会看到任何区别。如果没有规范等价，那么正则式 `\u00E0` 就无法匹配字符串 `\u0061\u0300`。在本书中讨论的所有其他正则流派都是这样来处理的。

最后，`Pattern.UNIX_LINES` 会告诉 Java 只把 `\n` 当作点号、脱字符和美元符号的换行符号。默认来说，所有 Unicode 换行都会被当作换行符号。

### JavaScript

如果想要把一个正则表达式反复应用到同一个字符串之上，例如，遍历所有匹配，或者查找和替换所有的匹配，而不只是处理第一个匹配，那么你应该使用 `/g` 或称“全局 (global)”标志。

### PHP

`/u` 选项会告知 PCRE 把正则表达式和目标字符串都解释为 UTF-8 字符串。这个修饰符还会支持 Unicode 正则标记，例如 `\p{FFFF}` 和 `\p{L}`。这些在实例 2.7 中进行了讲解。如果没有使用这个修饰符，PCRE 会把每个字节当作单独的字符，因此 Unicode 正则标

记就会产生错误。

/U 会切换向一个量词添加问号所产生的“贪心”和“懒惰”行为表现。通常来说，`<.*>` 是贪心的，而`<.*?>`是懒惰的。但如果使用了/U 之后，`<.*>` 是懒惰的，而`<.*?>` 是贪心的。作者强烈建议永远也不要使用这个标志，因为它会让阅读你的代码的程序员感到混淆，因为/U 这个修饰符只有 PHP 才有，所以很容易被人忽视。另外，如果你在别人的代码中看到它的话，不要对/U 和/u 产生混淆。记住正则修饰符是要区分大小写的。

## Perl

如果想要把一个正则表达式反复应用到同一个字符串之上（例如，遍历所有匹配，或者查找和替换所有的匹配，而不只是处理第一个匹配），那么你应该使用/g (“global”) 标志。

如果要在正则表达式中插入一个变量（例如，`m/I am $name/`），那么 Perl 会在每次使用该正则表达式的时候都重新进行编译，因为 \$name 的内容可能会发生改变。你可以使用/o 修饰符来避免这种情形。使用 `m/I am $name/o`，Perl 就会只在第一次需要它的时侯编译，而在随后会重复使用前面的编译结果。如果 \$name 的内容发生改变，那么这个正则表达式就无法反映它的改变。如果想要知道如何控制正则表达式重新编译的时机，请参考实例 3.3。

## Python

Python 拥有两个其他的选项，会改变单词边界的含义（参见实例 2.6），并改变简写字类 `\w`、`\d` 与 `\s`，以及它们的否定形式（参见实例 2.3）的含义。默认来说，这些记号只能处理 ASCII 字母、数字和空白。

选项 `re.LOCAL` 或 `re.L` 会使这些记号依赖于当前的本地设置。然后本地设置会决定哪些字符会被这些正则记号当作字母、数字和空白。当目标字符串不是一个 Unicode 字符串，而且想要让含有读音符号的字母按照本地设置来处理的时候，你就应当使用这个选项。

选项 `re.UNICODE` 或 `re.U` 会使这些记号依赖于 Unicode 标准。上面的这些正则记号会按照 Unicode 标准所定义的字母、数字和空白来进行处理。当你要应用正则表达式的目目标字符串是一个 Unicode 字符串的时候，应当使用这个选项。

## Ruby

`Regexp.new()`类厂会接受一个可选的第 3 个参数，让你选择正则表达式所支持的字符串编码。如果不为正则表达式指定一个编码的话，那么它就会使用同你的源代码一样的编码。在绝大多数情况下，都应当使用源代码的编码。

要想明确地选择一种编码，就可以向这个参数传递一个字符。该参数是不区分大小写

的。可选的取值如下。

n

它表示“None(无)”。在字符串中的每个字节都会被当作一个字符。你应当为 ASCII 文本使用这个选项。

e

为远东语言采用“EUC”编码。

s

支持日语的“Shift-JIS”编码。

u

支持 UTF-8，它支持每个字符使用 1~4 个字节，并且支持 Unicode 标准中的所有语言（其中包括了所有稍微重要的“活”语言）。

当使用一个字面正则表达式的时候，你可以使用修饰符/n、/e、/s 和/u 来设置编码格式。对于单个正则表达式来说，只能使用其中的一个修饰符。它们可以同任意或者所有的/x、/i 和/m 修饰符一起来使用。



别把 Ruby 中的/s 和 Perl、Java 或 .NET 中的/s 搞混。在 Ruby 中，/s 会强制加载 Shift-JIS 编码。在 Perl 和大多数其他正则流派中，这个选项会打开“点号匹配换行符”模式。在 Ruby 中，后者是用/m 来实现的。

## 参见

在第 2 章中，我们已经详细讲解了匹配模式的效果。下面这些小节还会讲解在正则表达式之内如何使用模式修饰符。

宽松排列：实例 2.18

不区分大小写：实例 2.1 的“不区分大小写匹配”小节

点号匹配换行符：实例 2.4

脱字符和美元符号匹配换行处：实例 2.5

实例 3.1 和实例 3.3 讲解了如何在源代码中使用字面正则表达式，以及如何创建正则表达式对象。当你创建一个正则表达式时候，就需要设置正则表达式选项。

## 3.5 检查是否可以在目标字符串中找到匹配

### 问题描述

你想要检查是否在某个特定字符串中可以找到一个特定正则表达式的匹配。这里只需要一个部分匹配就足够了。例如，正则表达式 `<regex>•pattern` 会部分匹配 The regex

pattern can be found。不必关心匹配的太多细节。只需要知道该正则表达式是否能匹配这个字符串。

## 解决方案

### C#

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
bool foundMatch = Regex.IsMatch(subjectString, "regex pattern");
```

如果该正则式是由最终用户提供的，那么你就需要在使用静态调用时进行完整的例外处理：

```
bool foundMatch = false;
try {
    foundMatch = Regex.IsMatch(subjectString, UserInput);
} catch (ArgumentNullException ex) {
    // Cannot pass null as the regular expression or subject string
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

要想重复使用同一个正则式，那么就需要构造一个 Regex 对象：

```
Regex regexObj = new Regex("regex pattern");
bool foundMatch = regexObj.IsMatch(subjectString);
```

如果该正则式是由最终用户提供的，那么你应当使用带有完整例外处理的 Regex 对象：

```
bool foundMatch = false;
try {
    Regex regexObj = new Regex(UserInput);
    try {
        foundMatch = regexObj.IsMatch(subjectString);
    } catch (ArgumentNullException ex) {
        // Cannot pass null as the regular expression or subject string
    }
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

### VB.NET

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "regex pattern")
```

如果该正则式是由最终用户提供的，那么你就需要在使用静态调用时进行完整的例外处理：

```
Dim FoundMatch As Boolean
Try
    FoundMatch = Regex.IsMatch(SubjectString, UserInput)
Catch ex As ArgumentNullException
    'Cannot pass Nothing as the regular expression or subject string
Catch ex As ArgumentException
    'Syntax error in the regular expression
End Try
```

要想重复使用同一个正则式，那么就需要构造一个 `Regex` 对象：

```
Dim RegexObj As New Regex("regex pattern")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

其中的 `IsMatch()` 调用应当使用 `SubjectString` 作为唯一的参数，并且该调用应当由 `RegexObj` 发起，而不是由 `Regex` 类来发起：

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

如果该正则式是由最终用户提供的，那么你应当使用带有完整例外处理的 `Regex` 对象：

```
Dim FoundMatch As Boolean
Try
    Dim RegexObj As New Regex(UserInput)
    Try
        FoundMatch = Regex.IsMatch(SubjectString)
    Catch ex As ArgumentNullException
        'Cannot pass Nothing as the regular expression or subject string
    End Try
    Catch ex As ArgumentException
        'Syntax error in the regular expression
    End Try
End Try
```

## Java

能够检查部分匹配的唯一途径是创建一个 `Matcher`：

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.find();
```

如果该正则式是由最终用户提供的，那么你还应当使用例外处理：

```
boolean foundMatch = false;
try {
    Pattern regex = Pattern.compile(UserInput);
    Matcher regexMatcher = regex.matcher(subjectString);
    foundMatch = regexMatcher.find();
} catch (PatternSyntaxException ex) {
    // Syntax error in the regular expression
}
```

## JavaScript

```
if (/regex pattern/.test(subject)) {  
    // Successful match  
} else {  
    // Match attempt failed  
}
```

## PHP

```
if (preg_match('/regex pattern/', $subject)) {  
    // Successful match  
} else {  
    // Match attempt failed  
}
```

## Perl

如果目标字符串保存在特殊变量\$\_中：

```
if (m/regex pattern/) {  
    # Successful match  
} else {  
    # Match attempt failed  
}
```

如果目标字符串保存在特殊变量\$subject中：

```
if ($subject =~ m/regex pattern/) {  
    # Successful match  
} else {  
    # Match attempt failed  
}
```

使用一个预编译好的正则表达式：

```
$regex = qr/regex pattern/;  
if ($subject =~ $regex) {  
    # Successful match  
} else {  
    # Match attempt failed  
}
```

## Python

如果只是做一次性的快速检查，你可以使用如下的全局函数：

```
if re.search("regex pattern", subject):  
    # Successful match  
else:  
    # Match attempt failed
```

要想重复使用同一个正则表达式，就需要使用一个编译好的对象：

```
reobj = re.compile("regex pattern")
if reobj.search(subject):
    # Successful match
else:
    # Match attempt failed
```

## Ruby

```
if subject =~ /regex pattern/
    # Successful match
else
    # Match attempt failed
end
```

下面的代码会完成同样的功能：

```
if /regex pattern/ =~ subject
    # Successful match
else
    # Match attempt failed
end
```

## 讨论

正则表达式最基本的任务就是检查一个字符串是否会匹配该正则表达式。在大多数编程语言中，只需要一个部分匹配就可以使匹配函数返回 `true`。匹配函数会扫描整个目标字符串，检查是否该正则表达式会匹配其中的任何子串。一旦找到一个匹配，该函数会立即返回 `true`。只有当它到达了字符串的结尾，并未找到任何匹配的情形下，才会返回 `false`。

在这个实例中的代码示例可以用于检查一个字符串中是否包含特定的数据。如果想要检查一个字符串是否能够整体匹配某个特定的模式（比如，用于用户输入的验证），那么你需要使用下一个实例。

## C# and VB.NET

`Regex` 类提供了 `.IsMatch()`方法的 4 个重载版本，其中的两个版本是静态方法。这样我们就可以使用不同的参数来调用 `.IsMatch()`。目标字符串总是第一个参数。它就是正则表达式要试图在其中寻找匹配的那个字符串。第一个参数一定不能是 `null`。否则，`.IsMatch()`会产生一个 `ArgumentNullException` 例外。

你可以只使用一行代码，不必构造 `Regex` 对象，只通过调用 `Regex.IsMatch()`就可以完成这个检查。只需要把正则表达式作为第二个参数，而把正则选项作为可选的第三个参数传递给这个函数。如果你的正则表达式出现了语法错误，那么 `.IsMatch()`会产生一

个 `ArgumentException` 例外。如果你的正则式是合法的，那么如果找到部分匹配，这个调用就会返回 `true`，或者如果没有找到匹配，就会返回 `false`。

如果想要把同一个正则表达式用于多个字符串之上，那么可以通过首先构造一个 `Regex` 对象，然后再用该对象来调用 `IsMatch()`，这样可以使你的代码更有效率。这样的话，只有第一个保存目标字符串的参数才是必需的。你还可以指定一个可选的第二个参数来说明正则表达式应当开始检查的字符位置。实质上，传递给第二个参数的数值，也就是你希望正则表达式忽略的目标字符串开始字符的数量。当你已经处理了这个字符串的一部分，而想要检查是否剩余部分会有匹配的时候，会用到这样的方式。如果你指定了一个数量，它必须大于或等于 0，而且要小于或者等于目标字符串的长度。否则的话，`IsMatch()`会产生一个 `ArgumentOutOfRangeException` 例外。

静态重载并不支持使用参数来指定正则表达式应该在字符串中开始尝试的位置。也不存在一个重载函数支持你告诉 `IsMatch()` 在字符串结束之前的某个位置停止。如果需要这样做，那么你可以调用 `Regex.Match("subject", start, stop)`，然后检查返回的 `Match` 对象的 `Success` 属性。更多细节，请参考实例 3.8。

## Java

要想检查一个正则表达式是否可以部分或者整体匹配一个字符串，需要按照在实例 3.3 中讲解的方式来实例化一个 `Matcher` 对象。然后使用新创建的或者刚重置过的匹配器来调用 `find()` 函数。

不要调用 `String.matches()`、`Pattern.matches()`、或者 `Matcher.matches()`。这几个函数都要求正则表达式匹配整个字符串。

## JavaScript

要检查一个正则表达式能否匹配某个字符串的一部分，用你的正则表达式调用 `test()` 函数。把目标字符串作为唯一的参数。

如果该正则表达式会匹配目标字符串的一部分或者全部，那么 `regexp.test()` 就会返回 `true`，否则返回 `false`。

## PHP

函数 `preg_match()` 可以用于不同的目的。调用它的最基本的方式是只需用两个必须的参数：包含正则表达式的字符串，以及包含你想要正则表达式搜索的目标文本的字符串。如果 `preg_match()` 能够找到匹配，就会返回 1；如果正则式没有产生匹配，那么函数会返回 0。

本章稍后的一些实例中会讲解你可以传递给 `preg_match()` 的可选参数。

## Perl

在 Perl 中，`m//` 实际上是一个正则表达式操作符，而不仅仅是一个正则表达式容器。如果你只是使用 `m//` 自身，它会使用 `$_` 变量来作为目标字符串。

如果你想要把匹配操作符用于另外一个变量的内容，就需要使用绑定操作符 `=~`，把正则操作符和你的变量关联起来。把正则式绑定到一个字符串会立即执行该正则式。如果正则式匹配目标字符串的一部分的话，那么模式匹配操作符会返回 `true`，否则返回 `false`。

如果你想要检查正则表达式是否不能匹配一个字符串，那么你可以使用 `!~`，其含义正好与 `=~` 相反。

## Python

在 `re` 模块中的 `search()` 函数会搜索一个字符串来查找正则表达式是否可以匹配其中的一部分。把正则表达式作为第一个参数、目标字符串作为第二个参数传递给它。你还可以把正则表达式选项作为可选的第三个参数。

`re.search()` 函数会调用 `re.compile()`，然后编译好的正则表达式对象会调用 `search()` 方法。这个方法只接受一个参数：目标字符串。

如果正则表达式找到一个匹配的话，`search()` 会返回一个 `MatchObject` 实例。如果正则表达式无法找到匹配，那么 `search()` 会返回 `None`。当你在一个 `if` 语句中来使用该返回值时，`MatchObject` 等价于 `True`，而 `None` 则等价于 `False`。本章后面的一些实例中会接着介绍如何使用在 `MatchObject` 中保存的信息。



别把 `search()` 和 `match()` 搞混。你不能在字符串的中间用 `match()` 查找匹配。  
下一个实例中会用到 `match()`。

## Ruby

操作符 `=~` 是模式匹配操作符。把它放到正则表达式和一个字符串中间，就可以查找第一个正则表达式匹配。这个操作符会返回一个整数，其值是正则匹配在字符串中的开始位置。如果不能找到匹配，它就会返回 `nil`。

这个操作符在 `Regexp` 和 `String` 两个类中都有实现。在 Ruby 1.8 中，把哪个类放到左边或哪个类放到右边都无关紧要。在 Ruby 1.9 中，这样做则会引起和命名捕获分组有关的特殊副作用。这会在实例 3.9 中进行解释。



在本书中所有其他的 Ruby 代码片段中，我们都把目标字符串放到 `=~` 操作符的左边，而把正则表达式放到它的右边。这样可以保持与 Perl 的一致性，而 Ruby 的 `=~` 语法也正是从 Perl 借鉴而来的，另外如此还可以避免可能会遇到的在 Ruby 1.9 中与命名捕获分组有关的麻烦。

## 参见

实例 3.6 和 3.7。

# 3.6 检查正则表达式能否整个匹配目标字符串

## 问题描述

你想要检查一个字符串是否整体符合某个特定的模式。也就是说，想要检查包含该模式的正则表达式是否可以从头到尾匹配该字符串。举例来说，如果你的正则表达式是 `<regex•pattern>`，那么它会匹配包含 `regex pattern` 的输入文本，但是不能匹配更长的字符串 `The regex pattern can be found.`

## 解决方案

### C#

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
bool foundMatch = Regex.IsMatch(subjectString, @"\Aregex pattern\Z");
```

要想重复使用同一个正则式，那么就需要构造一个 `Regex` 对象：

```
Regex regexObj = new Regex(@"\Aregex pattern\Z");
bool foundMatch = regexObj.IsMatch(subjectString);
```

### VB.NET

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "\Aregex pattern\Z")
```

要想重复使用同一个正则式，那么就需要构造一个 `Regex` 对象：

```
Dim RegexObj As New Regex("\Aregex pattern\Z")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

其中的 `IsMatch()` 调用应当使用 `SubjectString` 作为唯一的参数，而且该调用应当由 `RegexObj` 发起，而不是由 `Regex` 类来发起：

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

### Java

如果只是要检查一个字符串，你可以使用如下的静态调用：

```
boolean foundMatch = subjectString.matches("regex pattern");
```

如果想要把同一个正则表达式用于多个字符串之上，那么就需要编译该正则表达式，并创建一个匹配器：

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.matches(subjectString);
```

## JavaScript

```
if (/^regex pattern$/ .test(subject)) {
    // Successful match
} else {
    // Match attempt failed
}
```

## PHP

```
if (preg_match('/\A regex pattern \Z /', $subject)) {
    # Successful match
} else {
    # Match attempt failed
}
```

## Perl

```
if ($subject =~ m/\A regex pattern \Z /) {
    # Successful match
} else {
    # Match attempt failed
}
```

## Python

如果只是做一次性的快速检查，你可以使用如下的全局函数：

```
if re.match(r"regex pattern \Z ", subject):
    # Successful match
else:
    # Match attempt failed
```

要想重复使用同一个正则表达式，就需要使用一个编译过的对象：

```
reobj = re.compile(r"regex pattern \Z ")
if reobj.match(subject):
    # Successful match
else:
    # Match attempt failed
```

## Ruby

```
if subject =~ /\A regex pattern \Z /  
    # Successful match  
else  
    # Match attempt failed  
end
```

## 讨论

通常来说，一个成功的正则表达式匹配会告诉用户你想要的模式位于目标文本的某个地方。然而在许多情形下，你还想确切知道它是完全匹配的，也就是说在目标文本中不再包含任何其他内容。需要完全匹配的最常见的场景很可能是验证用户输入。如果用户输入了一个电话号码或者 IP 地址，但是其中包含了错误的字符，那么你希望能够拒绝该输入。

当你逐行处理一个文件（实例 3.21），而且你所使用的方法在获取每行时都把换行符留在每行结尾，那么也可以使用采用定位符`\$`和`\Z`的解决方案。在实例 2.5 中讲解过，这些定位符同样会匹配最后一个换行之前的位置，实际上也就是允许忽略最后一个换行。

在下面的几个小节中，我们会分别详细解释每种语言的解决方案。

## C# and VB.NET

在 .NET 框架的 `Regex` 类中并不包含一个函数可以用来检查正则表达式是否能够整体匹配一个字符串。我们的解决方案是把字符串起始定位符`\A`添加到正则表达式的开头，并且把字符串结尾定位符`\Z`加到正则表达式的最后。这样，正则表达式就只能整体匹配一个字符串，或者根本不能匹配。如果你的正则表达式使用了多选结构，比如`<one|two|three>`，那么就要注意需要在添加定位符之前把多选结构括起来作为一个分组：`\A(?:one|two|three)\Z`。

在正则表达式被修改来匹配整个字符串之后，你就可以继续使用在上一个实例中介绍的 `IsMatch()` 方法。

## Java

Java 有 3 个名为 `matches()` 的方法。它们都会检查一个正则表达式是否能整体匹配一个字符串。这些方法可以用来迅速完成输入验证，而不必把你的正则表达式包在字符串起始和结束定位符之间。

`String` 类中包含一个 `matches()` 方法，它接受一个正则表达式作为唯一参数。它会返回 `true` 或 `false`，说明正则式是否匹配了整个字符串。`Pattern` 类中包含一个静态的 `matches()` 方

法，它接受两个字符串作为参数：第一个是正则表达式，第二个是目标字符串。事实上，你可以把任意的 CharSequence 作为目标字符串传递给 Pattern.matches()。这也是使用 Pattern.matches()而不是 String.matches()的唯一理由。

String.matches()和 Pattern.matches()每次都会重新编译正则表达式，这会通过调用 Pattern.compile("regex").matcher(subjectString).matches()来完成。因为每次正则式都会重新编译，所以只有当你只打算使用这个正则表达式一次，或者说性能不是问题的情形下，才应当使用这两个方法。这些方法没有提供可以从正则表达式外部来说明匹配选项的方式。如果你的正则表达式包含语法错误，那么就会产生一个 PatternSyntaxException 例外。

如果想要高效地使用同一个正则表达式来检查多个字符串，那么你就应当对正则式进行编译，并创建和复用一个 Matcher 对象，这在实例 3.3 中已经进行了讲解。然后，使用你的 Matcher 实例来调用 matches()。这个函数并不能接受任何参数，因为你在创建或重置该匹配器的时候，已经指定好目标字符串了。

## JavaScript

JavaScript 并不包含一个函数可以用来检查一个正则表达式是否能够整体匹配一个字符串。我们的解决方案是把`\^`添加到正则表达式的开头，并且把`\$`加到正则表达式的最后。一定要注意不要为正则表达式设置/m 标志。只有在不使用/m 的情况下，脱字符和美元符号才会只匹配目标文本的开始和结束。如果你设置了/m 的话，那么它们还会匹配字符串中间的换行符。

在把定位符添加入你的正则表达式之后，就可以继续使用在上一个实例中介绍的同样的 regexp.test()方法。

## PHP

在 PHP 中并不包含一个函数可以用来检查正则表达式是否能够整体匹配一个字符串。我们的解决方案是把字符串起始定位符`\A`添加到正则表达式的开头，并且把字符串结尾定位符`\Z`加到正则表达式的最后。这样，正则表达式就只能整体匹配一个字符串，或者根本不能匹配。如果你的正则表达式使用了多选结构，比如`\one|two|three`，那么就要注意需要在添加定位符之前把多选结构括起来作为一个分组：`\A(?:one|two|three)\Z`。

在正则表达式被修改来匹配整个字符串之后，你就可以继续使用在上一个实例中介绍的同样的 preg\_match()方法。

## Perl

Perl 只包含一个模式匹配操作符，它只要能找到部分匹配就满意了。如果想要检查正则

表达式是否能够匹配整个目标字符串，那么需要把字符串起始定位符`\A`添加到正则表达式的开头，并且把字符串结尾定位符`\Z`加到正则表达式的最后。这样，正则表达式就只能整体匹配一个字符串，或者根本不能匹配。如果你的正则表达式使用了多选结构，比如`<one|two|three>`，那么就要注意需要在添加定位符之前把多选结构括起来作为一个分组：`\A(?:one|two|three)\Z`。

在正则表达式被修改来匹配整个字符串之后，你就可以像上一个实例中一样来使用它。

## Python

`match()`函数与我们在上一个实例中讲解的 `search()` 函数非常相似。它们之间的主要区别是 `match()` 只会在目标字符串的最开始处检查正则表达式，如果正则式无法在字符串的开始产生匹配，那么 `match()` 就会立即返回 `None`。然而，`search()` 函数会继续尝试字符串中的每一个后续位置，直至找到一个匹配或者到达了目标字符串的结尾。

`match()` 函数并不要求正则表达式必须匹配整个字符串。部分匹配也是可以接受的，只要它是从字符串起始处开始的匹配。如果想要检查正则式是否可以匹配整个字符串，那么就需要把字符串结束定位符`\Z`添加到你的正则表达式的最后。

## Ruby

Ruby 中的 `Regexp` 类并不包含一个函数可以用来检查正则表达式是否能够整体匹配一个字符串。解决方案是把字符串起始定位符`\A`添加到正则表达式的开头，并且把字符串结尾定位符`\Z`加到正则表达式的最后。这样，正则表达式就只能整体匹配一个字符串，或者根本不能匹配。如果你的正则表达式中使用了多选结构，比如`<one|two|three>`，那么就要注意需要在添加定位符之前把多选结构括起来作为一个分组：`\A(?:one|two|three)\Z`。

在正则表达式被修改以匹配整个字符串之后，你就可以继续使用在上一个实例中介绍的同样的`=~`操作符。

## 参见

实例 2.5 详细讲解了定位符的工作原理。

实例 2.8 和实例 2.9 讲解了多选结构和分组。如果你的正则表达式在任何分组之外使用了多选结构，那么就需要在添加定位符之前把你的正则表达式括起来作为一个分组。如果你的正则式没有使用多选结构，或者如果它只在分组中使用多选结构，那么在使用定位符的时候就不再需要添加额外的分组。

如果只需要部分匹配就可以接受的话，请参考实例 3.5。

## 3.7 获取匹配文本

### 问题描述

你已经有了一个正则表达式可以匹配目标文本的一部分，现在你想要把匹配到的文本提取出来。如果正则表达式可以多次匹配该字符串，你只想要得到第一个匹配结果。例如，当你把正则式 `\d+` 应用到字符串 `Do you like 13 or 42?`之上时，它应当返回 `13`。

### 解决方案

#### C#

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
string resultString = Regex.Match(subjectString, @"\d+").Value;
```

如果该正则式是由最终用户提供的，那么你就需要在使用静态调用时进行完整的例外处理：

```
string resultString = null;
try {
    resultString = Regex.Match(subjectString, @"\d+").Value;
} catch (ArgumentNullException ex) {
    // Cannot pass null as the regular expression or subject string
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

要想重复使用同一个正则式，那么就需要构造一个 `Regex` 对象：

```
Regex regexObj = new Regex(@"\d+");
string resultString = regexObj.Match(subjectString).Value;
```

如果该正则式是由最终用户提供的，那么你应当使用带有完整例外处理的 `Regex` 对象：

```
string resultString = null;
try {
    Regex regexObj = new Regex(@"\d+");
    try {
        resultString = regexObj.Match(subjectString).Value;
    } catch (ArgumentNullException ex) {
        // Cannot pass null as the subject string
    }
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

## VB.NET

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
Dim ResultString = Regex.Match(SubjectString, "\d+").Value
```

如果该正则式是由最终用户提供的，那么你就需要在使用静态调用时进行完整的例外处理：

```
Dim ResultString As String = Nothing
Try
    ResultString = Regex.Match(SubjectString, "\d+").Value
Catch ex As ArgumentNullException
    'Cannot pass Nothing as the regular expression or subject string
Catch ex As ArgumentException
    'Syntax error in the regular expression
End Try
```

要想重复使用同一个正则式，那么就需要构造一个 `Regex` 对象：

```
Dim RegexObj As New Regex("\d+")
Dim ResultString = RegexObj.Match(SubjectString).Value
```

如果该正则式是由最终用户提供的，那么就应当使用带有完整例外处理的 `Regex` 对象：

```
Dim ResultString As String = Nothing
Try
    Dim RegexObj As New Regex("\d+")
    Try
        ResultString = RegexObj.Match(SubjectString).Value
    Catch ex As ArgumentNullException
        'Cannot pass Nothing as the subject string
    End Try
    Catch ex As ArgumentException
        'Syntax error in the regular expression
    End Try
End Try
```

## Java

创建一个 `Matcher` 来执行查找并保存结果：

```
String resultString = null;
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
    resultString = regexMatcher.group();
}
```

如果该正则式是由最终用户提供的，那么你还应当使用完整的例外处理：

```
String resultString = null;
try {
```

```
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
    resultString = regexMatcher.group();
}
} catch (PatternSyntaxException ex) {
    // Syntax error in the regular expression
}
```

## JavaScript

```
var result = subject.match(/\d+);
if (result) {
    result = result[0];
} else {
    result = '';
}
```

## PHP

```
if (preg_match('/\d+/', $subject, $groups)) {
    $result = $groups[0];
} else {
    $result = '';
}
```

## Perl

```
if ($subject =~ m/\d+/) {
    $result = $&;
} else {
    $result = '';
}
```

## Python

如果只是做一次性的快速检查，你可以使用如下的全局函数：

```
matchobj = re.search("regex pattern", subject)
if matchobj:
    result = matchobj.group()
else:
    result = ""
```

要想重复使用同一个正则表达式，就需要使用一个编译过的对象：

```
reobj = re.compile("regex pattern")
matchobj = reobj.search(subject)
```

```
if match:  
    result = matchobj.group()  
else:  
    result = ""
```

## Ruby

你可以使用`=~`操作符和它的变量`$&`:

```
if subject =~ /regex pattern/  
    result = $&  
else  
    result = ""  
end
```

另外，你也可以使用一个 `Regexp` 对象来调用 `match` 方法:

```
matchobj = /regex pattern/.match(subject)  
if matchobj  
    result = matchobj[0]  
else  
    result = ""  
end
```

## 讨论

正则表达式的另外一个主要工作是从一个较长的字符串中提取符合模式的子串。本书中讨论的所有编程语言都会提供一种很容易的方式来获得字符串中的第一个正则表达式匹配。这样一个函数会从字符串的开始来尝试匹配该正则表达式，并继续向后搜索该串，直至找到正则表达式匹配为止。

## .NET

.NET 中的 `Regex` 类并不包含一个成员可以返回正则表达式匹配的字符串。但是它却包含一个 `Match()`方法，可以返回 `Match` 类的一个实例。这个 `Match` 对象有一个属性名为 `Value`，其中保存的是由正则表达式匹配的文本。如果正则表达式匹配失败，那么它依然会返回一个 `Match` 对象，但其 `Value` 属性中包含的是一个空串。

当调用 `Match()`方法的时候，总共有 5 种不同的重载方法可以使用。第一个参数永远都是包含目标文本的一个字符串，正则表达式会在其中查找匹配。第一个参数不能是 `null`。否则，`Match()`会产生一个 `ArgumentNullException` 例外。

如果你只想使用这个正则表达式几次，那么可以使用一个静态调用。此时，第二个参数是你想要使用的正则表达式。你还可以把正则选项作为可选的第三个参数。如果你的正则表达式中包含语法错误，就会产生一个 `ArgumentException` 例外。

如果你需要在许多字符串上使用同一个正则表达式，那么可以先通过构造一个 `Regex`

对象，然后用这个对象来调用 Match() 函数，这样做更为高效。此时，第一个参数，也就是目标字符串，是唯一必需的参数。你还可以使用一个可选的第二个参数来说明正则表达式应当开始进行查找的目标字符串中的位置（字符索引）。实质上，你传递给第二个参数的数值，也就是你想要正则表达式忽略的目标字符串开头字符的数量。当你已经处理了这个字符串的一部分，而想要检查剩余部分是否会有匹配的时候，会用到这样的方式。如果指定了一个数量，那么它必须大于或等于 0，而且要小于或者等于目标字符串的长度。否则，`IsMatch()` 会产生一个 `ArgumentOutOfRangeException` 例外。

如果你使用了说明起始位置的第二个参数，那么还可以使用第三个参数来说明正则表达式被允许搜索的子串的长度。这个数应该大于或等于 0，但是一定不能超过目标字符串的长度（第一个参数）减去起始偏移量（第二个参数）。例如，`regexObj.Match("123456", 3, 2)` 会尝试在 "45" 中找到匹配。如果第三个参数比目标字符串的长度要大，那么 `Match()` 会产生一个 `ArgumentOutOfRangeException` 例外。如果第三个参数虽然不比目标字符串的长度大，但是第二个和第三个参数加起来要大于字符串的长度，那么还是会生成一个 `ArgumentOutOfRangeException` 例外。如果你允许用户来指定起始和结束位置，那么就需要在调用 `Match()` 之前进行检查，或者保证一定会捕获这两个越界例外。

静态的重载函数不支持在参数中说明正则表达式可以搜索目标字符串中的哪个部分。

## Java

要得到正则表达式匹配的字符串子串，你就需要创建一个 `Matcher`，可以参考在实例 3.3 中的讲解。然后使用你的匹配器来调用 `find()` 方法，不用提供任何参数。如果 `find()` 的返回值是 `true`，那么可以不使用任何参数来调用 `group()`，就能够获取你的正则表达式匹配到的文本。如果 `find()` 的返回值是 `false`，你就不能再调用 `group()`，否则会得到一个 `IllegalStateException` 例外。

`Matcher.find()` 会接受一个可选参数，用来说明在目标字符串中的起始位置。你可以使用它来在字符串中的某个特定位置开始查找。如果想要从字符串开头查找，把该参数置为 0。如果把起始位置设为一个负数，或者是大于目标字符串长度的一个数，那么就会产生一个 `IndexOutOfBoundsException` 例外。

如果不使用这个参数，那么 `find()` 函数就会从上次找到匹配的位置之后的字符开始查找。如果是在 `Pattern.matcher()` 或 `Matcher.reset()` 之后第一次调用 `find()`，那么 `find()` 就会从字符串的开头进行查找。

## JavaScript

`string.match()` 函数接受一个正则表达式作为它的唯一参数。在这个参数中，你可以使用

字面正则表达式、正则表达式对象或者一个字符串。如果你传递的参数是一个字符串，那么 `string.match()` 会创建一个临时的 `regexp` 对象。

当匹配尝试失败的时候，`string.match()` 会返回 `null`。这样你就可以区分到底是一个正则表达式没有找到匹配，还是找到了一个长度为 0 的匹配。它也意味着你不能显示结果，因为可能会显示“`null`”，或者是一个空对象错误消息。

当匹配尝试成功时，`string.match()` 会返回一个包含匹配细节的数组。在数组中的第 0 个元素是包含该正则表达式匹配文本的一个字符串。

一定要保证你在正则表达式中没有添加/g 标志。因为如果添加了它，`string.match()` 就会有不同的表现，这会在实例 3.10 中讲解。

## PHP

在前面两个实例中讨论过的 `preg_match()` 函数会接受一个可选的第三个参数，在其中会保存该正则表达式匹配到的文本，以及它的捕获分组。当 `preg_match()` 的返回值是 1 时，该变量会包含一个字符串数组。在数组中的第 0 个元素中会包含整个的正则表达式匹配。其他的元素会在实例 3.9 中进行讲解。

## Perl

当模式匹配操作符 `m//` 找到一个匹配的时候，它会设置几个特殊变量。其中一个变量是 `$&`，它会保存该正则表达式所匹配到的子串。其他的特殊变量会在稍后的实例中加以讲解。

## Python

在实例 3.5 中已经讲解了 `search()` 函数。在这里，我们会把 `search()` 返回的 `MatchObject` 实例保存到一个变量中。要获得正则表达式匹配到的字符串子串，我们可以使用这个匹配对象（match object）来调用 `group()` 方法，不需要提供任何参数。

## Ruby

在实例 3.8 中会解释变量 `$~` 与 `MatchData` 对象。在字符串的上下文中，这个对象的取值是正则表达式匹配到的文本。在数组的上下文中，这个对象的取值是一个数组，它的第 0 个元素包含整个的正则表达式匹配。

`$&` 是一个特殊的只读变量。它是 `$~[0]` 的一个别名，其中包含的是正则表达式匹配到的文本的一个字符串。

## 参见

实例 3.5、3.8、3.9、3.10 和 3.11。

## 3.8 决定匹配的位置和长度

### 问题描述

在上一个实例中，我们所做的提取正则表达式匹配到的子串。这个实例要求你决定该匹配的开始位置和长度。有了这些信息之后，就可以在你自己的代码中提取匹配的内容，或者按照你的想法对正则表达式匹配的原始字符串做任何事情。

### 解决方案

#### C#

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
int matchstart, matchlength = -1;
Match matchResult = Regex.Match(subjectString, @"\d+");
if (matchResult.Success) {
    matchstart = matchResult.Index;
    matchlength = matchResult.Length;
}
```

要想重复使用同一个正则式，那么就需要构造一个 `Regex` 对象：

```
int matchstart, matchlength = -1;
Regex regexObj = new Regex(@"\d+");
Match matchResult = regexObj.Match(subjectString).Value;
if (matchResult.Success) {
    matchstart = matchResult.Index;
    matchlength = matchResult.Length;
}
```

#### VB.NET

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
Dim MatchStart = -1
Dim MatchLength = -1
Dim MatchResult = Regex.Match(SubjectString, "\d+")
If MatchResult.Success Then
    MatchStart = MatchResult.Index
    MatchLength = MatchResult.Length
End If
```

要想重复使用同一个正则式，那么就需要构造一个 `Regex` 对象：

```
Dim MatchStart = -1
Dim MatchLength = -1
Dim RegexObj As New Regex("\d+")
```

```
Dim MatchResult = Regex.Match(SubjectString, "\d+")
If MatchResult.Success Then
    MatchStart = MatchResult.Index
    MatchLength = MatchResult.Length
End If
```

## Java

```
int matchStart, matchLength = -1;
Pattern regex = Pattern.compile("\d+");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
    matchStart = regexMatcher.start();
    matchLength = regexMatcher.end() - matchStart;
}
```

## JavaScript

```
var matchstart = -1;
var matchlength = -1;
var match = /\d/.exec(subject);
if (match) {
    matchstart = match.index;
    matchlength = match[0].length;
}
```

## PHP

```
if (preg_match('/\d+/', $subject, $groups, PREG_OFFSET_CAPTURE)) {
    $matchstart = $groups[0][1];
    $matchlength = strlen($groups[0][0]);
}
```

## Perl

```
if ($subject =~ m/\d+/g) {
    $matchlength = length($&);
    $matchstart = length($`);
}
```

## Python

如果只是做一次性的快速检查，你可以使用如下的全局函数：

```
matchobj = re.search(r"\d+", subject)
if matchobj:
    matchstart = matchobj.start()
    matchlength = matchobj.end() - matchstart
```

要想重复使用同一个正则表达式，就需要使用一个编译过的对象：

```
reobj = re.compile(r"\d+")
matchobj = reobj.search(subject)
if matchobj:
    matchstart = matchobj.start()
    matchlength = matchobj.end() - matchstart
```

## Ruby

你可以使用`=~`操作符和它的变量 `$&`:

```
if subject =~ /regex pattern/
    matchstart = $~.begin()
    matchlength = $~.end() - matchstart
end
```

另外，也可以使用一个 `Regexp` 对象来调用 `match` 方法：

```
matchobj = /regex pattern/.match(subject)
if matchobj
    matchstart = matchobj.begin()
    matchlength = matchobj.end() - matchstart
end
```

## 讨论

### .NET

要想得到匹配的位置和长度，我们可以使用在上一个实例中讲过的同一个 `Regex.Match()` 方法。这次，我们使用的是 `Regex.Match()` 返回的 `Match` 对象的 `Index` 和 `Length` 两个属性。

`Index` 是在目标字符串中正则匹配开始的位置。如果正则匹配是从字符串中的第一个字符开始的，那么 `Index` 就是 0。如果匹配是从字符串中的第二个字符开始的，那么 `Index` 就是 1。`Index` 的最大值是目标字符串的长度。只有在正则式在字符串的结尾找到一个长度为 0 的匹配时，才会出现这种情况。例如，只包含字符串结尾定位符的正则式`\Z`总是会匹配到字符串的末尾。

`Length` 指的是匹配到的字符数量。一个合法的匹配也有可能长度为 0。例如，只包含单词边界`\b`的正则表达式总是在字符串中第一个单词的开始处找到一个长度为 0 的匹配。

如果匹配失败，那么 `Regex.Match()` 还是会返回一个 `Match` 对象。它的 `Index` 和 `Length` 属性的值都是 0。这些值在一个成功的匹配中也可能出现。包含字符串起始定位符`\A`的正则表达式就会在字符串的开始处找到一个长度为 0 的匹配。因此，你不能依赖

`Match.Index` 或 `Match.Length` 来决定匹配尝试是否成功。判断匹配是否成功应当使用 `Match.Success`。

## Java

要想得到匹配的位置和长度，可以调用在前一个实例中讲过的 `Matcher.find()`。当 `find()` 返回 `true` 的时候，不用任何参数调用 `Matcher.start()` 就可以获得正则匹配的第一个字符的位置索引。不用任何参数调用 `end()` 则可以得到该匹配之后的第一个字符的位置索引。把结束位置减去起始位置就可以得到匹配的长度，它的值也可能是 0。如果你在调用 `find()` 之前调用 `start()` 或者 `end()`，那么就会得到一个 `IllegalStateException` 例外。

## JavaScript

使用一个 `regexp` 对象调用 `exec()` 方法，可以得到一个关于匹配详细信息的数组。这个数组中会包含几个属性。`index` 属性会保存正则匹配在目标字符串中的开始位置。如果匹配是从字符串的开头开始，那么 `index` 的值是 0。数组中的 0 号元素中保存的是整个正则匹配的一个字符串。使用这个字符串的 `length` 属性，就可以知道匹配的长度。

如果正则表达式不能匹配这个字符串，那么 `regexp.exec()` 会返回 `null`。

不要使用 `exec()` 返回的数组中的 `lastIndex` 属性来确定匹配的终止位置。在一个严格的 JavaScript 实现中，`lastIndex` 根本不会在返回的数组中出现，而只会出现在 `regexp` 对象中。你同样也不应当使用 `regexp.lastIndex`。由于不同浏览器之间的区别，它也是不可靠的（实例 3.11 中会介绍更多的细节）。正确的做法是，你应当把 `match.index` 和 `match[0].length` 加起来确定正则匹配的结束位置。

## PHP

上一个实例中讲解了如何能够通过给 `preg_match()` 传递第三个参数来得到正则表达式匹配到的文本。你可以向这个函数传递一个常量 `PREG_OFFSET_CAPTURE` 作为第四个参数以获得匹配的位置。如果 `preg_match()` 的返回值是 1，这个参数就会改变在第三个参数中保存的内容。

当你没有使用第四个参数，或者是把它设置为 0 的时候，传递给第三个参数的变量会获得一个字符串数组。当你把 `PREG_OFFSET_CAPTURE` 作为第四个参数的时候，该变量会获得一个数组的数组。在外层数组的第一个元素中依然是整个匹配（参见上一个实例中的讲解），而后面数组元素中依然是不同的捕获分组（参见下一个实例）。但是，在每个元素中保存的不仅仅是正则式或捕获分组匹配到的文本的一个字符串，数组每个元素会实际上是包含两个值的一个数组：这两个值分别是匹配到的文本，以及匹配在字符串中的位置。

要得到关于整个匹配的详细信息，第 0 个元素的第 0 个子元素会告诉我们正则式匹配到的文本。我们把它传递给 `strlen()` 函数就可以计算它的长度。第 0 个元素的第 1 个子元素会保存一个整数，代表匹配在目标字符串中的起始位置。

## Perl

要得到匹配的长度，我们只需要计算保存整个正则表达式匹配的变量 `$&` 的长度。要得到匹配的开始位置，我们可以计算保存在正则匹配之前的字符串中文本的变量 `$`` 的长度。

## Python

`MatchObject` 的 `start()` 方法会返回在字符串中正则表达式匹配开始的位置。`end()` 方法则会返回在匹配之后的第一个字符的位置。当找到一个长度为 0 的正则表达式匹配的时候，这两个方法会返回相同的值。

可以通过给 `start()` 和 `end()` 提供一个参数来获取正则表达式中的某个捕获分组匹配到的文本范围。调用 `start(1)` 可以得到第一个捕获分组的开始位置，而调用 `end(2)` 会得到第二个分组的结束位置，以此类推。Python 最多支持 99 个捕获分组。第 0 个分组是整个正则表达式匹配。如果使用一个比正则表达式中的捕获分组个数还大的数（最大不能超过 99），那么就会造成 `start()` 和 `end()` 产生一个 `IndexError` 例外。如果分组编号是合法的，但是分组并不参与正则匹配，那么 `start()` 和 `end()` 都会为该分组返回 -1。

如果想要在一个二元组中保存起始和终止位置，那么应该使用匹配对象来调用 `span()` 方法。

## Ruby

实例 3.5 使用 `$~` 操作符来查找字符串中的第一个正则匹配。这个操作符的一个副作用是它会用 `MatchData` 类的一个实例来填充特殊变量 `$~`。这个变量对于线程和函数来说都是一个局部变量。这意味着你可以使用该变量的内容，直到当前方法退出，或者直到你下次在方法中使用 `=~` 操作符，在此过程中也不必担心另外一个线程或你的线程中的另外一个方法会修改这个变量。

如果想要得到多个正则匹配的详细信息，那么可以使用一个 `Regexp` 对象调用 `match()` 方法。这个方法会接受目标字符串作为它的唯一参数。当找到匹配时，它会返回一个 `MatchData` 实例，否则返回 `nil`。它还会把变量 `=~` 赋值为同一个 `MatchObject` 实例，但并不会修改保存在其他变量中的 `MatchObject` 实例。

`MatchData` 对象保存关于一个正则表达式匹配的所有详细信息。实例 3.7 和 3.9 会讲解如何获得正则表达式和捕获分组匹配到的文本。

`begin()` 方法会返回正则匹配在目标字符串中的开始位置。`end()` 会返回在正则匹配之后

的第一个字符的位置。`offset()`会返回包含起始和结束位置的一个数组。这三个方法都可以接受一个参数。传 0 给它们的话就可以得到整个正则匹配的位置，而传一个正整数给它们，就可以得到相应的捕获分组的位置。例如，`begin(1)`会返回第一个捕获分组的起始位置。

不要试图使用 `length()` 或 `size()` 获得匹配的长度。这两个方法都会返回从 `MatchData` 得到的数组中的元素个数，这会在实例 3.9 中加以讲解。

## 参见

实例 3.5 和 3.9。

## 3.9 获取匹配文本的一部分

### 问题描述

在实例 3.7 中，用一个正则表达式来匹配目标文本中的一个子串，但是在这个问题中，我们想要匹配的只是该子串的一部分。如果想把需要的部分分离出来，就应该在你的正则表达式中添加一个捕获分组（参见实例 2.9）。

例如，正则表达式`<http://([a-zA-Z0-9.-]+)>`会在字符串 Please visit `http://www.regexcookbook.com` for more information 中匹配到 `http://www.regexcookbook.com`。在第一个捕获分组中的正则部分匹配到的是 `www.regexcookbook.com`，而你想要把第一个捕获分组捕获到的域名保存到一个字符串变量中。

我们用这个简单的正则表达式来解释捕获分组的概念。在第 7 章中，还可以找到用来匹配 URL 的更加精确的正则表达式。

### 解决方案

#### C#

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
string resultString = Regex.Match(subjectString,
    "http://([a-zA-Z0-9.-]+)").Groups[1].Value;
```

要想重复使用同一个正则式，那么就需要构造一个 `Regex` 对象：

```
Regex regexObj = new Regex("http://([a-zA-Z0-9.-]+)");
string resultString = regexObj.Match(subjectString).Groups[1].Value;
```

#### VB.NET

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
Dim ResultString = Regex.Match(SubjectString,  
    "http://([a-z0-9.-]+)").Groups(1).Value
```

要想重复使用同一个正则式，那么就需要构造一个 **Regex** 对象：

```
Dim RegexObj As New Regex("http://([a-z0-9.-]+)")  
Dim ResultString = RegexObj.Match(SubjectString).Groups(1).Value
```

## Java

```
String resultString = null;  
Pattern regex = Pattern.compile("http://([a-z0-9.-]+)");  
Matcher regexMatcher = regex.matcher(subjectString);  
if (regexMatcher.find()) {  
    resultString = regexMatcher.group(1);  
}
```

## JavaScript

```
var result = "";  
var match = /http:\/\/([a-z0-9.-]+)/.exec(subject);  
if (match) {  
    result = match[1];  
} else {  
    result = '';  
}
```

## PHP

```
if (preg_match('%http://([a-z0-9.-]+)%', $subject, $groups)) {  
    $result = $groups[1];  
} else {  
    $result = '';  
}
```

## Perl

```
if ($subject =~ m!http://([a-z0-9.-]+)!) {  
    $result = $1;  
} else {  
    $result = '';  
}
```

## Python

如果只是做一次性的快速检查，你可以使用如下的全局函数：

```
matchobj = re.search("http://([a-z0-9.-]+)", subject)  
if matchobj:  
    result = matchobj.group(1)
```

```
else:  
    result = ""
```

要想重复使用同一个正则表达式，就需要使用一个编译过的对象：

```
reobj = re.compile("http://([a-zA-Z0-9.-]+)")  
matchobj = reobj.search(subject)  
if match:  
    result = matchobj.group(1)  
else:  
    result = ""
```

## Ruby

你可以使用 `=~` 操作符和它提供的编号变量，比如 `$1`：

```
if subject =~ %r!http://([a-zA-Z0-9.-]+)!  
    result = $1  
else  
    result = ""  
end
```

另外，你也可以使用一个 `Regexp` 对象来调用 `match` 方法：

```
matchobj = %r!http://([a-zA-Z0-9.-]+)!.match(subject)  
if matchobj  
    result = matchobj[1]  
else  
    result = ""  
end
```

## 讨论

实例 2.10 和实例 2.21 讲解了如何在正则表达式中使用编号的向后引用和替代文本，以再次匹配同一段文本，或者是把正则匹配的一部分插入到替代文本中。你可以使用同样的引用编号来在代码中获取由一个或多个捕获分组匹配到的文本。

在正则表达式中，捕获分组的编号是从 1 开始的。编程语言则通常都是从 0 开始对数组和列表进行编号的。本书中所有会把捕获分组保存到一个数组或者列表中的编程语言都会使用同正则表达式一样的编号，也就是从 1 开始编号。在数组或列表中的第 0 个元素因此就会被用于保存整个的正则表达式匹配。这意味着如果你的正则表达式有三个捕获分组，那么保存它们的匹配的数组会包含 4 个元素。元素 0 保存的是总的匹配，而元素 1、2 和 3 分别保存由三个捕获分组匹配的文本。

## .NET

要获取关于捕获分组的更多详细信息，我们还需要求助于最初在实例 3.7 中介绍过的 `Regex.Match()` 成员函数。它返回的 `Match` 对象拥有一个名为 `Groups` 的属性。它是类型

为 GroupCollection 的一个集合（collection）属性。这个集合中会保存你的正则表达式中所有捕获分组的详细信息。Groups[1]保存关于第一个捕获分组的详细信息，Groups[2]保存第二个分组，以此类推。

在 Groups 集合中会为每个捕获分组都保存一个 Group 对象。除了 Groups 属性之外，Group 类与 Match 类拥有完全相同的属性集合。Match.Groups[1].Value 会返回第一个捕获分组所匹配的文本，这与 Match.Value 会返回整个正则表达式匹配是一样的。Match.Groups[1].Index 和 Match.Groups[1].Length 则会分别返回由该分组匹配的文本的起始位置和长度。实例 3.8 中会介绍关于 Index 和 Length 的更多详细信息。

Groups[0]中保存的是整个正则匹配的详细信息，它同时也会被直接保存在匹配对象中。Match.Value 和 Match.Groups[0].Value 是等价的。

如果你传递了一个非法的分组编号，Groups 集合并不会产生一个例外。例如，Groups[-1]依然会返回一个 Group 对象，但是这个 Group 对象中会说明以-1 为编号的捕获分组并没有产生匹配。检查它最好的方法是使用 Success 属性。Groups[-1].Success 总是会返回 false。

要想确定到底存在多少个捕获分组，可以检查 Match.Groups.Count。这里的 Count 属性与在 .NET 中所有其他 collection 对象的 Count 属性是一样的：它会返回在集合中的元素个数，也就是最大允许的索引再加 1。在我们的例子中，Groups 集合中只包含 Groups[0]和 Groups[1]。所以 Groups.Count 会返回 2。

## Java

用来获得捕获分组所匹配的文本，或者是捕获分组的匹配细节的代码，实际上与我们在前面两个实例中所给出的用来处理整个正则匹配的代码是一样的。Matcher 类中的 group()、start()和 end()这三个方法都会接受一个可选参数。如果没有这个参数，或者把这个参数设置为 0，那么所得到的就是这个正则匹配的文本或者位置信息。

如果传一个正整数给它们，那么你就会得到相应的捕获分组的详细信息。分组的编号是从 1 开始的，这与在正则表达式中的向后引用的编号是一致的。如果你给了一个大于正则表达式中捕获分组个数的参数，那么这三个函数都会产生一个 IndexOutOfBoundsException 例外。如果捕获分组存在，但是并没有参与匹配，那么 group(n)会返回 null，而 start(n)和 end(n)都会返回-1。

## JavaScript

在前一个实例中已经介绍过，一个正则表达式对象的 exec()方法会返回关于匹配结构详细信息的一个数组。数组的元素 0 中会包含整个的正则匹配。元素 1 包含第一个捕获

分组匹配的文本，元素 2 保存第二个分组的匹配，以此类推。

如果正则表达式不能匹配目标字符串，那么 `regexp.exec()` 会返回 `null`。

## PHP

实例 3.7 讲解了可以通过向 `preg_match()` 传递第三个参数来获得正则表达式所匹配的文本。当 `preg_match()` 的返回值是 1 的时候，这个参数会被赋值为一个数组。其中元素 0 中会包含整个的正则匹配。元素 1 包含第一个捕获分组匹配的文本，元素 2 保存第二个分组的匹配，以此类推。数组的长度就是捕获分组的个数加 1。数组索引则正好对应于在正则表达式中的向后引用编号。

在上一个实例中讲到，如果你使用常量 `PREG_OFFSET_CAPTURE` 作为第四个参数，那么数组的长度依然是捕获分组的个数加 1。但是它在每个位置保存的就不再单单是一个字符串，而是一个包含两个元素的子数组。子元素 1 是一个整数，用来说明匹配文本在目标字符串中的起始位置。

## Perl

当模式匹配操作符 `m//` 找到一个匹配的时候，它会设置几个特殊变量。这些变量中包含了编号变量 `$1, $2, $3, 等等`，它们会分别保存在正则表达式中的捕获分组所匹配的文本。

## Python

这个问题的解答几乎与实例 3.7 中的解答是一模一样的。上个实例中我们在调用 `group()` 的时候没有使用任何参数，在这里我们给出了感兴趣的捕获分组的编号。调用 `group(1)` 就可以获得第一个捕获分组匹配的文本，而使用 `group(2)` 可以得到第二个分组，以此类推。Python 支持最多 99 个捕获分组。分组编号 0 是整个正则表达式匹配。如果你传递的参数大于正则表达式中的捕获分组的个数，那么 `group()` 就会产生一个 `IndexError` 例外。如果分组编号是合法的，但是分组并没有参与正则表达式的匹配，那么 `group()` 会返回 `None`。

可以一次向 `group()` 传递多个分组编号，从而通过一次调用获得几个捕获分组匹配的文本。所得到的结果是一个字符串的列表。

如果想要获取包含所有捕获分组所匹配文本的一个元组，那么你就需要调用 `MatchObject` 的 `groups()` 方法。对于没有参与到匹配中的分组，在返回的元组中对应的值是 `None`。

如果想要得到的不是一个元组，而是所有捕获分组匹配文本的一个字典（`dictionary`），那么就应该调用 `groupdict()`，而不是 `groups()`。你还可以向 `groupdict()` 传递一个参数，作为不参与匹配的分组在字典中对应的返回值（这样就不必非要返回 `None`）。

## Ruby

实例 3.8 中介绍了变量\$~以及 MatchData 对象。在数组的上下文中，这个对象的值是一个包含你的正则表达式中所有捕获分组匹配到的文本的一个数组。捕获分组的编号是从 1 开始的，这与在正则表达式中向后引用的编号是一致的。数组中的元素 0 中保存的是总的正则表达式匹配。

\$1、\$2 等是特殊的只读变量。\$1 是的\$~[1]简写，其中保存的是第一个捕获分组匹配到的文本。\$2 能得到第二个分组的信息，以此类推。

## 命名捕获

如果你的正则表达式中使用了命名的捕获分组，那么就可以在代码中使用分组的名称来获取它的匹配。

### C#

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
string resultString = Regex.Match(subjectString,
    "http://(?<domain>[a-zA-Z0-9.-]+)").Groups["domain"].Value;
```

要想重复使用同一个正则式，那么就需要构造一个 Regex 对象：

```
Regex regexObj=new Regex("http://(?<domain>[a-zA-Z0-9.-]+)");
string resultString=regexObj.Match(subjectString).Groups["domain"].Value;
```

在 C# 中，对于命名分组和编号分组来说，获取 Group 对象的代码并没有实质上的区别。这里我们不再使用一个整数来对 Groups 集合进行索引，替代它的是一个字符串。另外在这个例子中，如果一个分组不存在，那么 .NET 并不会产生一个例外。Match.Groups["nosuchgroup"].Success 只会简单地返回 false。

### VB.NET

如果只是做一次性的快速检查，你可以使用如下的静态调用：

```
Dim ResultString=Regex.Match(SubjectString,
    "http://(?<domain>[a-zA-Z0-9.-]+)").Groups("domain").Value
```

要想重复使用同一个正则式，那么就需要构造一个 Regex 对象：

```
Dim RegexObj As New Regex("http://(?<domain>[a-zA-Z0-9.-]+)")
Dim ResultString = RegexObj.Match(SubjectString).Groups("domain").Value
```

在 VB.NET 中，对于命名分组和编号分组来说，获取 Group 对象的代码并没有实质上的区别。这里我们不再使用一个整数来对 Groups 集合进行索引，替代它的是一个字符串。另外在这个例子中，如果一个分组不存在，那么 .NET 并不会产生一个例外。Match.Groups["nosuchgroup"].Success 只会简单地返回 false。

## PHP

```
if (preg_match('%http://(P<domain>[a-z0-9.-]+)%', $subject, $groups)) {  
    $result = $groups['domain'];  
} else {  
    $result = '';  
}
```

如果你的正则表达式中包含命名的捕获分组，那么赋给\$groups 的数组就是一个关联数组（associative array）。每个命名捕获分组所匹配到的文本会在该数组中添加两次。你可以使用该分组的编号或者分组的名称来获得它所匹配到的文本。在上面的代码示例中，\$groups[0]保存的是总的正则匹配，而\$groups[1]和\$groups['domain']保存的是正则表达式中的唯一捕获分组所匹配到的文本。

## Perl

```
if ($subject =~ '!http://(P<domain>[a-z0-9.-]+)!') {  
    $result = $+{'domain'};  
} else {  
    $result = '';  
}
```

Perl 从 5.10 版开始支持命名捕获分组。“\$+ hash”会保存所有命名捕获分组匹配到的文本。Perl 中会把命名分组与编号分组一起进行编号。在这个例子中，\$1 和 \$+{name} 保存的都是这个正则表达式中的唯一捕获分组所匹配到的文本。

## Python

```
matchobj = re.search("http://(P<domain>[a-z0-9.-]+)", subject)  
if matchobj:  
    result = matchobj.group("domain")  
else:  
    result = ""
```

如果你的正则表达式中包含命名捕获分组，那么你在调用 group()方法的时候，可以以不使用分组编号，而使用分组的名称。

## 参见

实例 2.9 讲解编号捕获分组。

实例 2.11 讲解命名捕获分组。

## 3.10 获取所有匹配的列表

### 问题描述

本章中前面所有的实例处理的都是一个正则表达式在目标字符串中找到的第一个匹

配。但是在许多情形下，正则表达式在部分匹配了一个字符串之后，还能在字符串的剩余部分找到其他匹配。而且在第二个匹配之后还可能会存在第三个匹配，以此类推。例如，正则表达式 `\d+` 会在目标字符串 `The lucky numbers are 7, 13, 16, 42, 65, and 99` 中找到 6 个匹配，分别是：7、13、16、42、65 和 99。

## 解决方案

### C#

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
MatchCollection matchlist = Regex.Matches(subjectString, @"\d+");
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 `Regex` 对象：

```
Regex regexObj = new Regex(@"\d+");
MatchCollection matchlist = regexObj.Matches(subjectString);
```

### VB.NET

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Dim matchlist = Regex.Matches(SubjectString, "\d+")
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 `Regex` 对象：

```
Dim RegexObj As New Regex("\d+")
Dim MatchList = RegexObj.Matches(SubjectString)
```

### Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("\\\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    resultList.add(regexMatcher.group());
}
```

### JavaScript

```
var list = subject.match(/\d+/g);
```

### PHP

```
preg_match_all('/\\d+/', $subject, $result, PREG_PATTERN_ORDER);
$result = $result[0];
```

### Perl

```
@result = $subject =~ m/\\d+/g;
```

这个解答只能用于不包含捕获分组的正则表达式，因此我们使用的是非捕获分组。更多细节请参考实例 2.9。

## Python

如果你打算只使用同一个正则表达式处理少量字符串，可以使用如下的全局函数：

```
result = re.findall(r"\d+", subject)
```

要想重复使用同一个正则表达式，那么就需要使用一个编译好的对象：

```
reobj = re.compile(r"\d+")
result = reobj.findall(subject)
```

## Ruby

```
result = subject.scan(/\d+/)
```

## 讨论

### .NET

Regex 类中的 Matches()方法会把正则表达式反复应用到字符串之上，直至找到所有的匹配为止。它会返回一个 MatchCollection 对象，其中会包含所有的匹配。目标字符串总是它的第一个参数。这个字符串，也就是正则表达式，要尝试去寻找匹配的字符串。第一个参数不能是 null。否则，Matches()会产生一个 ArgumentNullException 例外。

如果你只想获得在少量字符串之上的正则表达式匹配，那么可以使用 Matches()的静态重载。把目标字符串作为第一个参数，而你的正则表达式作为第二个参数。你还可以把正则选项作为可选的第三个参数。

如果要处理大量字符串，那么你可以先通过构造一个 Regex 对象，然后用这个对象来调用 Matches()函数。此时，目标字符串就成了唯一必需的参数。你还可以使用一个可选的第二个参数来说明正则表达式应当开始进行查找的目标字符串中的位置（字符索引）。实质上，你传递给第二个参数的数值，也就是你想要正则表达式忽略的目标字符串开始字符的数量。当你已经处理了这个字符串的一部分，而想要检查剩余部分是否会有匹配的时候，会用到这样的方式。如果你指定了一个数量，它必须大于或等于 0，而且要小于或者等于目标字符串的长度。否则，IsMatch()会产生一个 ArgumentOutOfRangeException 例外。

静态重载并不支持使用参数来指定正则表达式应该在字符串中开始尝试的位置。也不存在一个重载函数支持你告知 IsMatch()在字符串结束之前停止。如果你需要这样做，那么可以在一个循环中调用 Regex.Match("subject", start, stop)（参考下一个实例中的讲解），然后把所有找到的匹配都添加到一个自己的列表中。

## Java

Java 没有提供一个函数来获取所有匹配的列表。但是你可以轻松地在自己的代码中通过修改实例 3.7 的代码完成这样的功能。要改动的地方是不再使用 if 语句，而是用一个 while 循环来调用 find()。

这个例子中用到了 List 和 ArrayList 两个类，因此需要在代码的开始添加一个 import 语句：import java.util.\*;。

## JavaScript

这里的代码也会调用 string.match()，就像在实例 3.7 中给出的 JavaScript 解答一样。但是这里有一个很小但是却很重要的区别：/g 标志。实例 3.4 中讲解了正则表达式标志。

/g 标志会告知 match() 函数在字符串中反复查找所有的匹配，并把它们都放到一个数组中。在上面的代码示例中，list[0] 会保存第一个正则匹配，list[1] 包含第二个，以此类推。检查 list.length 就可以得知匹配的个数。如果没有找到任何匹配，那么 string.match 会照常返回 null。

在返回数组中的所有元素都是字符串。当你使用一个包含/g 标志的正则式时，string.match() 不会提供关于正则表达式匹配的更多细节。如果想要得到关于所有正则匹配的匹配细节，那么就需要按照在实例 3.11 中的解答来循环访问所有匹配。

## PHP

前面实例中给出的所有 PHP 实例使用的函数都是 preg\_match()，这个函数会在字符串中查找第一个正则匹配。函数 preg\_match\_all() 也非常类似。主要的区别是它会找到一个字符串中的所有匹配。它会返回一个整数，说明正则表达式能找到的所有匹配的个数。

preg\_match\_all() 的前 3 个参数同 preg\_match() 的前 3 个参数是完全一样的：一个正则表达式的字符串，你要查找的目标字符串，以及用来接受查找结果数组的变量。唯一的区别是第三个参数现在是必需的，而且这个数组总是多维的。

至于第 4 个参数，可以使用常量 PREG\_PATTERN\_ORDER 或者 PREG\_SET\_ORDER。如果你没有给出第 4 个参数，那么默认值是 PREG\_PATTERN\_ORDER。

如果使用了 PREG\_PATTERN\_ORDER，你会得到一个数组，其中第 0 个元素保存的是总的匹配的详细信息，而从第 1 个元素开始则会分别保存捕获分组的详细信息。数组的长度是捕获分组的个数加 1。这样的顺序与 preg\_match() 所使用的顺序是一样的。区别是这里每个元素保存的不再是 preg\_match() 所找到的正则匹配的字符串，而是 preg\_matches() 找到的所有匹配构成的一个子数组。每个子数组的长度与 preg\_matches() 的返回值是一样的。

要想得到字符串中所有正则匹配的一个列表，并把捕获分组匹配的文本丢掉，那么可

以使用 `PREG_PATTERN_ORDER` 作为第 4 个参数，然后提取数组中的第 0 个元素。如果你只对某个特定捕获分组匹配到的文本感兴趣，那么可以使用 `PREG_PATTERN_ORDER` 和捕获分组的编号。例如，在调用 `preg_match('%http:// ([a-zA-Z0-9.-]+)%', $subject, $result)` 之后使用 `$result[1]` 就可以得到在你的目标字符串中所有 URL 的域名列表。

`PREG_SET_ORDER` 会在数组中填充相同的字符串，但是方式却有所不同。数组的长度是 `preg_matches()` 的返回值。数组中的每个元素是一个子数组，其中子元素 0 中是总的正则匹配，而子元素 1 之后则保存的是捕获分组。如果你使用了 `PREG_SET_ORDER`，那么中 `$result[0]` 保存的数组与你调用 `preg_match()` 时得到的数组是一样的。

你可以把 `PREG_OFFSET_ORDER` 同 `PREG_PATTERN_ORDER` 或 `PREG_SET_ORDER` 组合起来使用。这样做的效果就好像把 `PREG_OFFSET_ORDER` 作为第 4 个参数传递给了 `preg_match()`。这样数组中的每个元素不再是一个字符串，而是包含 2 个元素的数组，一个是字符串，另外一个是在原始的目标字符串中匹配开始的位置索引。

## Perl

实例 3.4 中讲到，需要添加 `/g` 修饰符，才能用正则表达式在目标字符串中找到多于一个匹配。如果在列表的上下文中使用一个全局正则表达式，那么它会找到使用的匹配并把它们返回来。在这个实例中，赋值操作符左边的列表变量提供了一个列表上下文。

如果正则表达式中不包含任何捕获分组，该列表中会只包含总的正则匹配。如果正则表达式中包含捕获分组，那么列表中会包含对每个正则匹配来说的所有捕获分组匹配的文本。总的正则匹配却不在其中，除非把整个正则表达式声明成一个捕获分组。如果你只是想获得总的正则匹配的一个列表，那么可以把所有捕获分组都替换成非捕获分组。实例 2.9 中介绍了这两种分组。

## Python

在 `re` 模块中的 `findall()` 函数会重复搜索一个字符串，来查找正则表达式的所有匹配。把正则表达式当作第一个参数，而目标字符串作为第二个参数。你还可以把正则表达式选项作为可选的第三个参数。

`re.findall()` 函数会调用 `re.compile()`，然后编译好的正则表达式对象会调用 `findall()` 方法。这个方法只有一个必需的参数，也就是目标字符串。

`findall()` 方法还接受 2 个在全局的 `re.findall()` 函数中不支持的可选参数。在目标字符串之后，你可以传递一个 `findall()` 应当在字符串中开始进行搜索的字符位置。如果没有使用这个参数，那么 `findall()` 会处理整个字符串。如果指定了一个起始位置，那么你还可以指定一个结束位置。如果没有指定结束位置，那么搜索会一直执行到字符串的结束。

不管如何调用 `findall()`，得到的结果总是包含找到的所有匹配的列表。如果正则式中不包含捕获分组，那么你就会得到一个字符串列表。如果它包含捕获分组，那么你会得到一个元组列表，每个元组中包含每个正则匹配对应的所有捕获分组匹配到的文本。

## Ruby

`String` 类中的 `scan()`方法接受一个正则表达式作为唯一参数。它会在字符串中重复找到所有的正则表达式匹配。当不使用块（block）来调用它的时候，`scan()`会返回包含所有正则匹配的一个数组。

如果正则表达式中不包含任何捕获分组，`scan()`会返回一个字符串数组。数组中的每个元素是一个正则匹配，其中保存的是匹配到的文本。

当存在捕获分组的时候，`scan()`会返回一个数组的数组。数组中每个元素对应一个正则匹配。每个元素都是所有捕获分组匹配到的文本的一个数组。子元素 0 保存第一个捕获分组匹配到的文本，子元素 1 保存第二个捕获分组，以此类推。整体的正则匹配并不包含在该数组中。如果想要把整体匹配包含进来，那么可以添加一个捕获分组来把整个正则表达包起来。

Ruby 中没有提供选项可以让 `scan()`在正则表达式中包含捕获分组的情况下返回一个字符串数组。唯一解决方案只能是把所有命名的和编号的捕获分组都替换成非捕获分组。

## 参见

实例 3.7、3.11 和 3.12。

## 3.11 遍历所有匹配

### 问题描述

上一个实例讲解了如何把一个正则表达式反复应用到字符串之上以得到匹配的一个列表。现在，我们要求你在代码中遍历所有的匹配。

### 解决方案

#### C#

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Match matchResult = Regex.Match(subjectString, @"\d+");
while (matchResult.Success) {
    // Here you can process the match stored in matchResult
```

```
        matchResult = matchResult.NextMatch();
    }
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 Regex 对象：

```
Regex regexObj = new Regex(@"\d+");
matchResult = regexObj.Match(subjectString);
while (matchResult.Success) {
    // Here you can process the match stored in matchResult
    matchResult = matchResult.NextMatch();
}
```

## VB.NET

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Dim MatchResult = Regex.Match(SubjectString, @"\d+")
While MatchResult.Success
    'Here you can process the match stored in MatchResult
    MatchResult = MatchResult.NextMatch
End While
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 Regex 对象：

```
Dim RegexObj As New Regex("\d+")
Dim MatchResult = RegexObj.Match(SubjectString)
While MatchResult.Success
    'Here you can process the match stored in MatchResult
    MatchResult = MatchResult.NextMatch
End While
```

## Java

```
Pattern regex = Pattern.compile("\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    // Here you can process the match stored in regexMatcher
}
```

## JavaScript

如果你的正则表达式可能会产生一个长度为 0 的匹配，或者你无法确切知道这一点，那么就要注意，在遇到长度为 0 匹配和 exec() 时可能会遇到跨浏览器的问题：

```
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
    // Don't let browsers such as Firefox get stuck in an infinite loop
    if (match.index == regex.lastIndex) regex.lastIndex++;
    // Here you can process the match stored in the match variable
}
```

如果确切地知道正则表达式不可能返回一个长度为 0 的匹配，那么你可以直接在正则表达式上进行循环：

```
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
    // Here you can process the match stored in the match variable
}
```

## PHP

```
preg_match_all('/\d+/', $subject, $result, PREG_PATTERN_ORDER);
for ($i = 0; $i < count($result[0]); $i++) {
    # Matched text = $result[0][$i];
}
```

## Perl

```
while ($subject =~ m/\d+/g) {
    # matched text = $&
}
```

## Python

如果你只打算使用同一个正则表达式处理少量字符串，可以使用如下的全局函数：

```
for matchobj in re.finditer(r"\d+", subject):
    # Here you can process the match stored in the matchobj variable
```

要想重复使用同一个正则表达式，就需要使用一个编译好的对象：

```
reobj = re.compile(r"\d+")
for matchobj in reobj.finditer(subject):
    # Here you can process the match stored in the matchobj variable
```

## Ruby

```
subject.scan(/\d+/) { |match|
    # Here you can process the match stored in the match variable
}
```

## 讨论

### .NET

实例 3.7 中讲解了如何使用 Regex 类的成员函数 Match() 来获取字符串中的第一个正则表达式匹配。为了能够遍历该字符串中的所有匹配，我们还是需要调用 Match() 函数来获取第一个匹配的详细信息。Match() 函数会返回 Match 类的一个实例，我们会把它保

存到变量 `matchResult` 中。如果 `matchResult` 对象的 `Success` 属性的内容是 `true`，那么就会进入循环。

在循环的开始，你可以使用 `Match` 类的属性来处理第一个匹配的详细信息。在实例 3.7 中已经介绍了 `Value` 属性，实例 3.8 中介绍了 `Index` 和 `Length` 两个属性，而在实例 3.9 中介绍了 `Groups` 集合。

处理完第一个匹配之后，用 `matchResult` 变量调用 `NextMatch()` 成员函数。`Match.NextMatch()` 会返回 `Match` 类的一个实例，这和 `Regex.Match()` 是一样的。新返回的实例中保存的是第二个匹配的详细信息。

我们把 `matchResult.NextMatch()` 的结果赋值给同一个 `matchResult` 变量，这样做可以比较容易实现对所有匹配的遍历。然后需要再次检查 `matchResult.Success` 的值来看 `NextMatch()` 是否又找到了一个匹配。当 `NextMatch()` 失败的时候，它依然会返回一个 `Match` 对象，但是它的 `Success` 属性会被设为 `false`。通过只使用一个 `matchResult` 变量，我们可以把最初的检查与在调用 `NextMatch()` 之后的检查都放到一个 `while` 循环语句中。

调用 `NextMatch()` 并不会改变用来调用它的 `Match` 对象。如果你愿意的话，可以把每个正则表达式匹配对应的 `Match` 对象都保存起来。

`NextMatch()` 方法不接受任何参数。它会使用与你传递给 `Regex.Match()` 方法一样的正则表达式和目标字符串。`Match` 对象中会保存到正则表达式和目标字符串的引用。

即使你的目标字符串中包含大量的正则匹配，也可以使用静态的 `Regex.Match()` 调用。`Regex.Match()` 会只编译你的正则表达式一次，而返回的 `Match` 对象中会保存到编译好的正则表达式的引用。即使你使用静态的 `Regex.Match()` 调用，`Match.MatchAgain()` 也会使用 `Match` 对象指向的前面编译好的正则表达式。只有当你想要重复调用 `Regex.Match()` 的情形下，例如要在多个字符串之上使用同一个正则表达式，才需要对 `Regex` 类进行实例化。

## Java

在 Java 中很容易就可以遍历字符串中的所有匹配。只需要在一个 `while` 循环中调用实例 3.7 中介绍过的 `find()` 方法。每次调用 `find()` 都会得到一个新的 `Matcher` 对象，并且更新关于匹配的详细信息和下一次匹配尝试的开始位置。

## JavaScript

在你开始之前，必须保证要放到循环中去的正则表达式中使用了 `/g` 标志。实例 3.4 中讲解了这个标志。当 `regexp = /\d+/g` 时，`while (regexp.exec())` 会找到目标字符串中的所有数字。如果 `regexp = /\d+/,` 那么 `while (regexp.exec())` 会反复找到字符串中的第一个数字，直到你的脚本崩溃，或是被浏览器强行终止。

需要注意的是 `while (/d+/g.exec())`（在一个含有/g 的字面正则表达式之上进行循环）也可能会陷入同样的无限循环之中，至少在某些特定的 JavaScript 实现中会是如此，这是因为在 while 循环每次重复时正则表达式都会被重新进行编译。当正则表达式被重新编译时，匹配尝试的开始位置也会被重置为字符串的起始位置。因此应当把正则表达式在循环外赋给一个变量，然后要确保它只会被编译一次。

实例 3.8 和 3.9 中解释了 `regexp.exec()` 返回的对象。不管是否在循环中调用 `exec()`，这个对象总是一样的。接着你就可以随意操作这个对象。

/g 产生的唯一效果是它会在你每次调用 `exec()` 的时候，更新 `regexp` 对象的 `lastIndex` 属性值。在本实例的第二个 JavaScript 解答中可以看到，即使当你使用一个字面正则表达式时也会如此。当下一次再调用 `exec()` 的时候，匹配尝试会从 `lastIndex` 开始。如果你赋给 `lastIndex` 一个新的值，那么匹配尝试就会从指定的位置开始执行。

然而，这也是 `lastIndex` 带来的一个很大问题。如果你认真阅读过 ECMA-262v3 标准，其中会要求 `exec()` 把 `lastIndex` 设置为本次匹配之后的第一个字符。这意味着即使匹配的长度为 0，下一次匹配也会从刚刚找到的匹配位置继续开始，这样就会造成一个无限循环。

在这种情形下，如果上一次匹配的长度为 0，那么本书中讨论的所有正则表达式引擎（除了 JavaScript）都会自动调整下一个匹配尝试到字符串中的下一个字符开始。在 Internet Explorer 的实现中，如果匹配长度为 0，那么 `lastIndex` 就会被加 1。这也就是为什么在实例 3.7 中提到不能使用 `lastIndex` 来决定匹配的截止位置，因为在 Internet Explorer 中会得到错误的值。

然而，Firefox 的开发人员则严格按照要求实现了 ECMA-262v3 标准，虽然这也就意味着 `regexp.exec()` 可能会陷入一个无限循环中。这种结果并不是不可能出现的。例如，你可以使用 `re = /^.*$/gm; while (re.exec())` 来遍历一个多行字符串中的所有行，而如果该字符串中包含一个空行，Firefox 就会死在那里。

一种解决方案是如果 `exec()` 函数没有自动增加 `lastIndex` 的值，你可以在自己的代码中添加这样的功能。本实例中的第一个 JavaScript 解答就是这样做的。如果你不确定，那么可以简单地把这一行代码粘贴进去就可以了。

当使用 `string.replace()`（实例 3.14），或者使用 `string.match()` 来查找所有匹配（实例 3.10）的时候，并不会遇到同样的问题。这是因为对于这两个在内部使用 `lastIndex` 的方法，在 ECMA-262v3 标准中明确说明了 `lastIndex` 遇到长度为 0 的匹配时必须加 1。

## PHP

`preg_match()` 函数会接受一个可选的第 5 个参数，指定在字符串中匹配尝试应当开始的位置。在第二次调用 `preg_match()` 的时候，你可以修改实例 3.8 把 `$matchstart +`

`$matchlength` 作为第 5 个参数传递过去，这样就可以找到字符串中的第二个匹配；接着可以在第三次及以后的匹配中反复这样做，直到 `preg_match()` 返回 0 为止。实例 3.18 中使用了这个方法。

除了需要为每次匹配尝试添加额外代码来计算起始位置之外，反复调用 `preg_match()` 还存在效率不高的问题，因为我们无法把一个编译过的正则表达式保存到一个变量中。在每次调用时，`preg_match()` 必须都去到缓存中查找编译好的正则表达式。

另外一种更容易也更高效的解答是，按照上一个实例中讲解的方法调用 `preg_match_all()`，然后对包含匹配结果的数组进行遍历。

## Perl

实例 3.4 中讲到，要想让你的正则表达式在目标字符串中查找多余一个匹配，你就需要添加/g 修饰符。如果在一个标量上下文中使用全局正则表达式，那么它会从上一次匹配结束的地方开始寻找下一个匹配。在这个实例中，while 语句提供了一个标量上下文。在 while 循环中可以使用所有的特殊变量，例如\$&（参考实例 3.7 中的讲解）。

## Python

`re` 模块中的 `finditer()` 函数会返回一个迭代器，你可以用它来找到正则表达式的所有匹配。把正则表达式作为第一个参数，目标字符串作为第二个参数。你还可以把正则表达式选项作为可选的第三个参数。

`re.finditer()` 函数会调用 `re.compile()`，然后用编号的正则表达式对象调用 `finditer()` 方法。这个方法只需要一个必需参数，也就是目标字符串。

`finditer()` 方法还接受两个 `re.finditer()` 不支持的可选参数。在目标字符串之后，可以传递一个字符串中的字符位置，作为 `finditer()` 应当开始搜索的位置。如果你不提供这个参数，那么迭代器就会处理整个目标字符串。如果你指定了一个起始位置，那么还可以指定一个结束位置。如果没有指定一个结束位置，那么搜索就会一直执行到字符串结束。

## Ruby

`String` 类中的 `scan()` 方法接受一个正则表达式作为唯一的参数，并会遍历目标字符串中的所有正则表达式匹配。当使用一个 `block` 来调用它时，你可以处理每一个找到的匹配。如果你的正则表达式中不包括任何捕获分组，那么可以在 `block` 中说明一个迭代器变量。这个变量会得到包含正则表达式匹配到的文本的一个字符串。

如果你的正则表达式中包含了一个或多个捕获分组，那么就需要为每个分组建立一个变量。第一个变量会得到第一个捕获分组匹配到的文本构成的字符串，第二个变量会得到第二个捕获分组，以此类推。然而整体正则匹配却不会被放到任何变量中。如果

你也想得到整体匹配，那么就需要把整个正则表达式也放到一个捕获分组中去。

```
subject.scan(/(a)(b)(c)/) {|a, b, c|
    # a, b, and c hold the text matched by the three capturing groups
}
```

如果你声明的变量个数少于正则表达式中包含的捕获分组的数量，那么就只能访问到你提供了变量的捕获分组。如果所给的变量个数多出了捕获分组的数量，那么多余的变量会被设为 nil。

如果只列出了一个迭代器变量，而你的正则表达式中包含一个或多个捕获分组，那么这个变量会得到的是一个字符串的数组。这个数组中会包含每个捕获分组得到的字符串。如果只存在一个捕获分组，那么这个数组就只会包含一个元素：

```
subject.scan(/(a)(b)(c)/) {|abc|
    # abc[0], abc[1], and abc[2] hold the text
    # matched by the three capturing groups
}
```

## 参见

实例 3.7、3.8、3.10 和 3.12。

## 3.12 在过程代码中对匹配结果进行验证

### 问题描述

实例 3.10 中讲解了如何通过把正则表达式反复应用到一次匹配成功剩余的字符串之上，从而可以获取所有的正则匹配列表。在这个实例中，你想要获得的匹配列表需要满足一些在正则表达式无法（很容易地）表示的标准。例如，在获取一个幸运数字的列表时，你只想保留那些是 13 倍数的整数。

### 解决方案

#### C#

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
StringCollection resultList = new StringCollection();
Match matchResult = Regex.Match(subjectString, @"\d+");
while (matchResult.Success) {
    if (int.Parse(matchResult.Value) % 13 == 0) {
        resultList.Add(matchResult.Value);
    }
    matchResult = matchResult.NextMatch();
}
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 Regex 对象：

```
StringCollection resultList = new StringCollection();
Regex regexObj = new Regex(@"\d+");
MatchResult = regexObj.Match(subjectString);
while (matchResult.Success) {
    if (int.Parse(matchResult.Value) % 13 == 0) {
        resultList.Add(matchResult.Value);
    }
    matchResult = matchResult.NextMatch();
}
```

## VB.NET

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Dim ResultList = New StringCollection
Dim MatchResult = Regex.Match(SubjectString, "\d+")
While MatchResult.Success
    If Integer.Parse(MatchResult.Value) Mod 13 = 0 Then
        ResultList.Add(MatchResult.Value)
    End If
    MatchResult = MatchResult.NextMatch
End While
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 Regex 对象：

```
Dim ResultList = New StringCollection
Dim RegexObj As New Regex("\d+")
Dim MatchResult = RegexObj.Match(SubjectString)
While MatchResult.Success
    If Integer.Parse(MatchResult.Value) Mod 13 = 0 Then
        ResultList.Add(MatchResult.Value)
    End If
    MatchResult = MatchResult.NextMatch
End While
```

## Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    if (Integer.parseInt(regexMatcher.group()) % 13 == 0) {
        resultList.add(regexMatcher.group());
    }
}
```

## JavaScript

```
var list = [];
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
    // Don't let browsers such as Firefox get stuck in an infinite loop
    if (match.index == regex.lastIndex) regex.lastIndex++;
    // Here you can process the match stored in the match variable
    if (match[0] % 13 == 0) {
        list.push(match[0]);
    }
}
```

## PHP

```
preg_match_all('/\d+/', $subject, $matchdata, PREG_PATTERN_ORDER);
for ($i = 0; $i < count($matchdata[0]); $i++) {
    if ($matchdata[0][$i] % 13 == 0) {
        $list[] = $matchdata[0][$i];
    }
}
```

## Perl

```
while ($subject =~ m/\d+/g) {
    if ($& % 13 == 0) {
        push(@list, $&);
    }
}
```

## Python

如果你打算只使用同一个正则表达式处理少量字符串，可以使用如下的全局函数：

```
list = []
for matchobj in re.finditer(r"\d+", subject):
    if int(matchobj.group()) % 13 == 0:
        list.append(matchobj.group())
```

要想重复使用同一个正则表达式，那么就需要使用一个编译好的对象：

```
list = []
reobj = re.compile(r"\d+")
for matchobj in reobj.finditer(subject):
    if int(matchobj.group()) % 13 == 0:
        list.append(matchobj.group())
```

## Ruby

```
list = []
subject.scan(/\d+/) { |match|
```

```
list << match if (Integer(match) % 13 == 0)
}
```

## 讨论

正则表达式处理的是文本。虽然正则表达式 `\d+` 会匹配我们所谓的数字，但是对于正则表达式引擎来说，它也只是包含一个或多个数字的字符串。

如果想要找到某些特殊的数字，例如能够被 13 整除的整数，那么更容易的做法是：使用一个通用的正则表达式匹配所有的数字，然后使用一小段过程代码去掉那些你不感兴趣的正则匹配。

这个实例的解答的基础都是前面一个实例的解答。在前一个实例中讲解了如何遍历所有的匹配。在循环中，我们把正则表达式的匹配转换成了一个数字。

有些语言会自动这样做；而有些语言则需要显式的函数调用来把字符串转换成一个整数。然后我们会检查该整数是否能被 13 整除。如果是的话，那么正则匹配就会被添加到一个列表中。否则，正则匹配会被丢弃。

## 参见

实例 3.7、3.10 和 3.11。

## 3.13 在另一个匹配中查找匹配

### 问题描述

你想要找一个特定正则表达式的所有匹配，但是只想局限在目标字符串的某些特定的片段。另外一个正则表达式可以用来匹配位于字符串中的这些片段。

假设你有一个 HTML 文件，其中有一些段落使用 `<b>` tag 标记为了粗体。现在你想要找到所有标记为粗体的数字。如果有的粗体文本中包含了多个数字，那么你想要分别匹配它们。例如，当处理字符串 `<b>2</b> 3 4 <b>5 6 7</b>` 的时候，你想要找到 4 个匹配：2、5、6 和 7。

### 解决方案

#### C#

```
StringCollection resultList = new StringCollection();
Regex outerRegex = new Regex("<b>(.*)</b>", RegexOptions.Singleline);
Regex innerRegex = new Regex(@"\d+");
// Find the first section
Match outerMatch = outerRegex.Match(subjectString);
while (outerMatch.Success) {
```

```

    // Get the matches within the section
    Match innerMatch = innerRegex.Match(outerMatch.Groups[1].Value);
    while (innerMatch.Success) {
        resultList.Add(innerMatch.Value);
        innerMatch = innerMatch.NextMatch();
    }
    // Find the next section
    outerMatch = outerMatch.NextMatch();
}

```

## VB.NET

```

Dim ResultList = New StringCollection
Dim OuterRegex As New Regex("<b>(.*)</b>", RegexOptions.Singleline)
Dim InnerRegex As New Regex("\d+")
'Find the first section
Dim OuterMatch = OuterRegex.Match(SubjectString)
While OuterMatch.Success
    'Get the matches within the section
    Dim InnerMatch = InnerRegex.Match(OuterMatch.Groups(1).Value)
    While InnerMatch.Success
        ResultList.Add(InnerMatch.Value)
        InnerMatch = InnerMatch.NextMatch
    End While
    OuterMatch = OuterMatch.NextMatch
End While

```

## Java

使用 2 个匹配器来遍历会比较容易，这可以用于 Java 4 及更高版本：

```

List<String> resultList = new ArrayList<String>();
Pattern outerRegex = Pattern.compile("<b>(.*)</b>", Pattern.DOTALL);
Pattern innerRegex = Pattern.compile("\d+");
Matcher outerMatcher = outerRegex.matcher(subjectString);
while (outerMatcher.find()) {
    Matcher innerMatcher = innerRegex.matcher(outerMatcher.group());
    while (innerMatcher.find()) {
        resultList.add(innerMatcher.group());
    }
}

```

下面的代码效率更高（因为 innerMatcher 只创建了一次），但是要求 Java 5 或者更高版本：

```

List<String> resultList = new ArrayList<String>();
Pattern outerRegex = Pattern.compile("<b>(.*)</b>", Pattern.DOTALL);
Pattern innerRegex = Pattern.compile("\d+");
Matcher outerMatcher = outerRegex.matcher(subjectString);
Matcher innerMatcher = innerRegex.matcher(subjectString);
while (outerMatcher.find()) {

```

```
    innerMatcher.region(outerMatcher.start(), outerMatcher.end());
    while (innerMatcher.find()) {
        resultList.add(innerMatcher.group());
    }
}
```

## JavaScript

```
var result = [];
var outerRegex = /<b>([\s\S]*?)</b>/g;
var innerRegex = /\d+/g;
var outerMatch = null;
while (outerMatch = outerRegex.exec(subject)) {
    if (outerMatch.index == outerRegex.lastIndex)
        outerRegex.lastIndex++;
    var innerSubject = subject.substr(outerMatch.index,
                                      outerMatch[0].length);
    var innerMatch = null;
    while (innerMatch = innerRegex.exec(innerSubject)) {
        if (innerMatch.index == innerRegex.lastIndex)
            innerRegex.lastIndex++;
        result.push(innerMatch[0]);
    }
}
```

## PHP

```
$list = array();
preg_match_all('%<b>(.*)</b>%s', $subject, $outermatches,
              PREG_PATTERN_ORDER);
for ($i = 0; $i < count($outermatches[0]); $i++) {
    if (preg_match_all('/\d+/', $outermatches[0][$i], $innermatches,
                      PREG_PATTERN_ORDER)) {
        $list = array_merge($list, $innermatches[0]);
    }
}
```

## Perl

```
while ($subject =~ m!<b>(.*)</b>!gs) {
    push(@list, ($& =~ m/\d+/g));
}
```

这只能用于在内层的正则表达式（在这个例子中的`\d+`）中不包含任何捕获分组的情况，因此你应当只使用非捕获分组。更多细节请参考实例 2.9。

## Python

```
list = []
innerre = re.compile(r"\d+")
```

```
for outermatch in re.finditer("(?s)<b>(.*)</b>", subject):
    list.extend(innerre.findall(outermatch.group(1)))
```

## Ruby

```
list = []
subject.scan(/<b>(.*)</b>/m) { |outergroups|
    list += outergroups[0].scan(/\d+/)
}
```

## 讨论

正则表达式非常适合于对输入进行 token 解析 (tokenizing)，但是它们却很不适合对输入进行分析。token 解析指的是在字符串中识别不同的组成部分，例如数字，单词，符号，标记，注释，等等。它涉及从左向右扫描文本，在匹配的过程中尝试不同的选择，以及尝试不同的字符数量。正则表达式可以很好地完成这一任务。

分析意味着处理这些 token 之间的关系。例如，在编程语言中，这些 token 的组合就形成了语句、函数、类、命名空间等。要想把这些 token 在更大的上下文中的含义记录下来，最好还是留给过程代码来做。具体来讲，正则表达式不能记录非线性化的上下文，例如嵌套结构<sup>1</sup>。

试图在一种 token 中寻找另一种 token 则是人们通常会用正则表达式来解决的任务。一堆 HTML 粗体 tag 是很容易用正则表达式来进行匹配的：`<b>(.*)</b>`<sup>2</sup>。用正则表达式匹配一个数字就更加容易了：`\d+`。但是如果你试图把这两个正则表达式组合到一个正则表达式中，那么就会得到一个看起来很不相同的正则表达式：

```
\d+ (?=(?:(?![<b>]).)*</b>)
正则选项: None
正则流派: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby
```

虽然刚刚给出的这个正则表达式也能够解决这个实例所提出的问题，但它却很不直观。即使一个正则表达式专家也需要花很长时间研究才能知道它是做什么用的，或者可能还需要求助于工具来检查它的匹配。而这仅仅是两个简单正则式的组合而已。

更好的一种解决方案是不要试图改变这两个正则表达式，而是使用过程代码来组合它们。这样所得到的代码虽然有点儿长，但是却更加容易理解和维护，而我们最初使用正则表达式的主要原因就是为了创造简单的代码。像 `<b>(.*)</b>` 这样的正则表达式是任何稍微懂一些正则表达式的人都很容易理解的，而且能够很快解决可能需要许多行代码才能解决的问题。

<sup>1</sup> 有些现代的正则流派也尝试着引入了平衡或者递归匹配。然而，这些特性却导致了非常复杂的正则表达式，这反而证明了我们所提到的“过程代码更适合做分析”的论点。

<sup>2</sup> 要想支持跨多行的边界，需要打开“点号匹配换行符”模式。在 JavaScript 中，使用`<b>([\s\S]*?)</b>`。

虽然这个实例中的解答是本章中看起来最为复杂的，但它们却都很直截了当。其中使用了两个正则表达式。“外层”正则表达式会匹配 HTML 的粗体 tag 以及两个 tag 之间的文本，然后两个 tag 之间的文本会被捕获到第一个捕获分组中。这个正则表达式的实现与实例 3.11 中的代码是一样的。唯一的区别是之前占位置的注释（说明在哪里使用匹配结果）被替换为了实际做事情的“内层”正则表达式。

第二个正则表达式会匹配一个数字。这个正则表达式的实现与实例 3.10 中的代码是一样的。唯一的区别是我们处理的不是整个目标字符串，而是外层正则表达式的一个捕获分组。

可以有两种方式来把内存正则表达式限制为只能处理外层正则表达式（或者是它的一个捕获分组）匹配到的文本。有些语言会提供一个函数来把正则表达式应用到字符串的子串之上。如果匹配函数不会自动把捕获分组匹配的文本填到一个结构中，这样会省去一次额外的字符串复制。我们当然也可以先获得捕获分组匹配到的子串，然后再把内层正则表达式应用于其上。

不管采用哪种方式，在一个循环中使用两个正则表达式总是会比使用一个包含嵌套的顺序环视分组的正则表达式要快很多。后者要求正则引擎进行大量的回溯。如果用在大文件之上，那么使用一个正则式的解决方案会明显慢很多，因为它需要为目标字符串中的每个数字确定片段边界（也就是 HTML 粗体 tag），这也包括了不在两个<b> tag 之间的数字。使用两个正则表达式的解决方案在找到片段边界之前，甚至不会开始查找数字，而它只需要线性时间就可以找到所有的片段。

## 参见

实例 3.8、3.10 和 3.11。

## 3.14 替换所有匹配

### 问题描述

你想要把正则表达式 <before> 的所有匹配都替换为替代文本 <after>。

### 解决方案

#### C#

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
string resultString = Regex.Replace(subjectString, "before", "after");
```

如果该正则式是由最终用户提供的，那么你就需要在使用静态调用时进行完整的例外处理：

```
string resultString = null;
try {
    resultString = Regex.Replace(subjectString, "before", "after");
} catch (ArgumentNullException ex) {
    // Cannot pass null as the regular expression, subject string,
    // or replacement text
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

如果想要把同一个正则表达式用于大量的字符串之上，那么就需要构造一个 `Regex` 对象：

```
Regex regexObj = new Regex("before");
string resultString = regexObj.Replace(subjectString, "after");
```

如果该正则式是由最终用户提供的，那么你应当使用带有完整例外处理的 `Regex` 对象：

```
string resultString = null;
try {
    Regex regexObj = new Regex("before");
    try {
        resultString = regexObj.Replace(subjectString, "after");
    } catch (ArgumentNullException ex) {
        // Cannot pass null as the subject string or replacement text
    }
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

## VB.NET

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Dim ResultString = Regex.Replace(SubjectString, "before", "after")
```

如果该正则式是由最终用户提供的，那么你就需要在使用静态调用时进行完整的例外处理：

```
Dim ResultString As String = Nothing
Try
    ResultString = Regex.Replace(SubjectString, "before", "after")
Catch ex As ArgumentNullException
    'Cannot pass null as the regular expression, subject string,
    'or replacement text
Catch ex As ArgumentException
    'Syntax error in the regular expression
End Try
```

如果想要把同一个正则表达式用于大量的字符串之上，那么就需要构造一个 `Regex` 对象：

```
Dim RegexObj As New Regex("before")
Dim ResultString = RegexObj.Replace(SubjectString, "after")
```

如果该正则式是由最终用户提供的，那么你应当使用带有完整例外处理的 Regex 对象：

```
Dim ResultString As String = Nothing
Try
    Dim RegexObj As New Regex("before")
    Try
        ResultString = RegexObj.Replace(SubjectString, "after")
    Catch ex As ArgumentNullException
        'Cannot pass null as the subject string or replacement text
    End Try
    Catch ex As ArgumentException
        'Syntax error in the regular expression
    End Try
```

## Java

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
String resultString = subjectString.replaceAll("before", "after");
```

如果该正则式是由最终用户提供的，那么你就需要在使用静态调用时进行完整的例外处理：

```
try {
    String resultString = subjectString.replaceAll("before", "after");
} catch (PatternSyntaxException ex) {
    // Syntax error in the regular expression
} catch (IllegalArgumentException ex) {
    // Syntax error in the replacement text (unesCAPED $ signs?)
} catch (IndexOutOfBoundsException ex) {
    // Non-existent backreference used the replacement text
}
```

如果想要把同一个正则表达式用于大量的字符串之上，那么就需要构造一个 Matcher 对象：

```
Pattern regex = Pattern.compile("before");
Matcher regexMatcher = regex.matcher(subjectString);
String resultString = regexMatcher.replaceAll("after");
```

如果该正则式是由最终用户提供的，那么你应当使用带有完整例外处理的 Matcher 对象：

```
String resultString = null;
try {
    Pattern regex = Pattern.compile("before");
```

```
Matcher regexMatcher = regex.matcher(subjectString);
try {
    resultString = regexMatcher.replaceAll("after");
} catch (IllegalArgumentException ex) {
    // Syntax error in the replacement text (unesCAPed $ signs?)
} catch (IndexOutOfBoundsException ex) {
    // Non-existent backreference used the replacement text
}
} catch (PatternSyntaxException ex) {
    // Syntax error in the regular expression
}
```

## JavaScript

```
result = subject.replace(/before/g, "after");
```

## PHP

```
$result = preg_replace('/before/', 'after', $subject);
```

## Perl

假设目标字符串保存在特殊变量 `$_` 中，而结果也会保存到 `$_` 中：

```
s/before/after/g;
```

假设目标字符串保存在特殊变量 `$subject` 中，而结果也会保存到 `$subject` 中：

```
$subject =~ s/before/after/g;
```

假设目标字符串保存在特殊变量 `$subject` 中，而结果会保存到 `$result` 中：

```
($result = $subject) =~ s/before/after/g;
```

## Python

如果你只需要处理少量字符串，可以使用全局函数：

```
result = re.sub("before", "after", subject)
```

如果要反复使用同一个正则表达式，那么就需要使用一个编译好的对象：

```
reobj = re.compile("before")
result = reobj.sub("after", subject)
```

## Ruby

```
result = subject.gsub(/before/, 'after')
```

## 讨论

### .NET

在.NET 中，你总是可以使用 `Regex.Replace()`方法来使用正则表达式进行查找和替换。这个 `Replace()`方法有 10 个重载形式。其中一半会接受字符串作为替代文本；我们会在下面讨论这些形式。另外一半则接受一个 `MatchEvaluator` 委派（delegate）来作为替代，这些形式会在实例 3.16 中讲解。

`Replace()`期望的第一个参数总是一个字符串，其中包含你要在其上执行查找和替换的原始目标文本。这个参数不能是 `null`。否则，`Replace()`会产生一个 `ArgumentNullException` 例外。`Replace()`的返回值总是应用了替换之后的字符串。

如果你只需要使用一个正则表达式少数几次，那么你可以使用静态调用。这样的话，第二个参数是你想要使用的正则表达式。把替代文本作为第三个参数。你还可以使用可选的第四个参数来传递正则选项。如果你的正则表达式中有语法错误的话，那么就会产生一个 `ArgumentException` 例外。

如果你需要在许多字符串上使用同一个正则表达式的话，那么你可以先通过构造一个 `Regex` 对象，然后用这个对象来调用 `Replace()`函数，这样做会更为高效。此时，第一个参数是目标字符串，第二个参数是替代文本。这两个是唯一必需的参数。

当你在 `Regex` 类的实例之上调用 `Replace()`的时候，还可以使用一些额外的参数对查找和替换进行限制。如果省略掉这些参数，那么在目标字符串中的所有正则表达式匹配都会被替换掉。`Replace()`函数的静态重载形式不会支持这些额外的参数；它们总是会替换掉所有的匹配。

在目标字符串和替代文本之后的第三个可选参数中，你可以传递想要执行的替换次数。如果你传了一个大于等于 1 的数字，那么它就是替换执行的最大次数。例如，`Replace(subject, replacement, 3)` 只会替换前三个正则表达式匹配，并且会忽略随后的所有匹配。如果你在第三个参数中传了 0，那么就根本不会执行任何替换，目标字符串会照原样返回来。如果你传递的是 -1，那么所有的正则匹配都会被替换。如果给的参数小于 -1，就会导致 `Replace()`产生一个 `ArgumentOutOfRangeException` 例外。

如果你使用了第三个参数来指定替换执行的次数，那么你还可以指定一个可选的第四个参数来指定正则表达式应该开始搜索的字符位置（索引）。实质上，你传递给第四个参数的数值，也就是你想要正则表达式忽略的目标字符串开始字符的数量。当你已经处理了这个字符串的一部分，而又想要对字符串中的剩余部分进行查找和替换时，就会用到这样的方式。如果你指定了一个数量，它必须大于或等于 0，而且要小于或者等于目标字符串的长度。否则，`Replace()`会产生一个 `ArgumentOutOfRangeException` 例外。与 `Match()`不同的是，`Replace()`并不支持再使用一个参数来指定正则表达式允

许搜索的子串长度。

## Java

如果只想使用同一个正则表达式来查找和替换一个字符串，那么可以用你的字符串来直接调用方法 `replaceFirst()` 或 `replaceAll()`。这两个方法都会接受 2 个参数：一个是含有正则表达式的字符串，一个是包含替代文本的字符串。这些函数是一种简便形式，它们实际上会调用 `Pattern.compile("before").matcher(subjectString).replaceFirst("after")` 和 `Pattern.compile("before").matcher(subjectString).replaceAll("after")`。

要得到一个正则表达式匹配的字符串中的子串，你就需要创建一个 `Matcher`，可以参考在实例 3.3 中的讲解。然后，使用你的匹配器来调用 `replaceFirst()` 或 `replaceAll()`，这时只需要把替代文本作为唯一参数。

如果正则式和替代文本都是由最终用户提供的，那么你就需要考虑 3 种不同的例外类。如果正则表达式包含语法错误，那么 `Pattern.compile()`、`String.replaceFirst()` 和 `String.replaceAll()` 会产生 `PatternSyntaxException` 例外。如果在替代文本中包含语法错误，那么 `replaceFirst()` 和 `replaceAll()` 会产生 `IllegalArgumentException` 例外。而如果替代文本语法上是正确的，但是却引用了一个不存在的捕获分组，那么就会产生一个 `IndexOutOfBoundsException` 例外。

## JavaScript

要使用正则表达式来在字符串中执行查找和替换，可以在该字符串之上调用 `replace()` 函数。把正则表达式作为第一个参数，而包含替代文本的字符串作为第二个参数。`replace()` 函数会返回一个应用了替换之后的新字符串。

如果你想要替换字符串中的所有正则匹配，那么在创建正则表达式对象时要设置/g 标志。这可以参考实例 3.4 中的讲解。如果你没有使用/g 标志，那么就只会替换第一个匹配。

## PHP

使用 `preg_replace()` 函数，可以很容易地在一个字符串中执行查找和替换。把正则表达式作为第一个参数，把替代文本作为第二个参数，而目标字符串作为第三个参数。函数返回值是应用了替换之后的一个字符串。

可选的第四个参数允许你对替换执行次数进行限制。如果你没有使用这个参数，或者是指定了 -1，那么所有的正则匹配都会被替换掉。如果使用的是 0，那么就不会进行任何替换。如果使用的是一个正整数，那么 `preg_replace()` 在字符串中执行的替换次数最多不会超过你指定的参数。如果匹配次数小于该参数，那么所有匹配都会被替换掉，并不会产生任何错误。

如果想要知道执行了多少次替换，那么你可以在调用中添加第五个参数。这个参数会得到一个实际上执行了的替换次数的一个整数。

`preg_replace()`的一个特殊功能是你可以在前 3 个参数中传递数组来取代字符串。如果你在第三个参数中传递的不是单个字符串，而是一个字符串数组，那么 `preg_replace()`会返回在所有字符串之上完成了查找和替换操作之后的一个字符串数组。

如果你在第一个参数中传递了一个正则表达式字符串数组，那么 `preg_replace()`会依次使用每个正则表达式来对目标字符串进行查找和替换。如果你传递的是目标字符串的一个列表，那么所有的正则表达式都会被应用到所有的目标字符串之上。当使用一个正则表达式数组来搜索的时候，你可以指定单个字符串作为替代（所有的正则表达式都用它），也可以指定一个替代数组。如果两个参数都是数组，那么 `preg_replace()`会遍历正则式和替代文本 2 个数组，对每个正则式都使用一次不同的替代文本。当 `preg_replace()`遍历数组的时候，会按照它们在内存中的存放顺序，而这不一定是按照数组中的索引的数值顺序。如果你的数组不是照数值顺序来构造的，那么在把它们传递给 `preg_replace()`之前需要对正则表达式和替代文本的数组调用 `ksort()` 函数。

下面的例子中构造的 `$replace` 数组就是逆序的：

```
$regex[0] = '/a/';
$regex[1] = '/b/';
$regex[2] = '/c/';
$replace[2] = '3';
$replace[1] = '2';
$replace[0] = '1';

echo preg_replace($regex, $replace, "abc");
ksort($replace);
echo preg_replace($regex, $replace, "abc");
```

第一个对 `preg_replace()` 的调用会显示 321，而这并不是你所期望的结果。在使用了 `ksort()` 之后，替换结果才变成预期的 123。`ksort()` 会修改你传递给它的变量，所以不要把它的返回值（`true` 或 `false`）传递给 `preg_replace()`。

## Perl

在 Perl 中，`s///` 实际上是一个替代操作符。如果只使用 `s///`，那么它会对变量 `$_` 执行查找和替换，并把结果保存到变量 `$_` 中。

如果想要在另外一个变量之上使用替换操作符，那么就需要使用 `=~` 绑定操作符来把替换操作符关联到你的变量之上。把替换操作符绑定到一个字符串会立即执行查找和替换。结果会被保存回包含目标字符串的变量中。

`s///` 操作符总是会修改你绑定到其上的变量。如果想要把查找和替换的结果保存到一个新的变量中，而不想去修改原来的变量，那么就需要首先把原来的字符串赋给

结果变量，然后再把替换操作符绑定到该变量之上。这个实例中 Perl 的解决方案在一行代码内完成了这两个步骤。

在实例 3.4 中讲过，使用/g 修饰符会替换所有的正则匹配。如果不使用它，那么 Perl 只会替换第一个匹配。

## Python

re 模块中的 sub() 函数会使用正则表达式执行查找和替换操作。把你的正则表达式作为第一个参数，替代文本作为第二个参数，而目标字符串作为第三个参数。全局的 sub() 函数并不支持接受正则表达式选项作为参数。

re.sub() 函数会先调用 re.compile()，然后使用编译好的正则表达式对象来调用 sub() 方法。这个方法有 2 个必需的参数：替代文本和目标字符串。

sub() 的两种形式都会返回一个把所有正则表达式替换之后的字符串。它们都会接受一个可选参数，允许你使用它来限制替换执行的次数。如果略去了这个参数，或者设为 0，那么所有正则匹配都会被替换。如果你传递了一个正整数，那么它就是最大可能会替换的匹配个数。如果找到的匹配个数小于你指定的参数，那么所有的匹配都会被替换，而不会产生任何错误。

## Ruby

String 类中的 gsub() 函数会使用正则表达式执行查找和替换操作。把你的正则表达式作为第一个参数，替代文本作为第二个参数。返回值是应用了替换之后的一个新字符串。如果没有找到任何匹配，那么 gsub() 会返回原来的字符串。

gsub() 函数并不会修改调用它的字符串。如果你想要修改最初的字符串，那么就需要使用 gsub!()。如果没有找到任何正则匹配，那么 gsub!() 会返回 nil。否则，它会返回应用了替换之后的调用它的字符串。

## 参见

第 1 章中的“使用正则表达式查找和替换”小节，以及实例 3.15 和 3.16。

## 3.15 使用匹配的子串来替换匹配

### 问题描述

你想要在执行查找和替换的时候，在替代文本中重新插入正则匹配的一部分。在你的正则表达式中，已经把想要重新插入的子串分离到了捕获分组中（参考实例 2.9）。

例如，你想要匹配由等号分隔的单词对，然后把等号两边的单词进行交换。

# 解决方案

## C#

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
string resultString = Regex.Replace(subjectString, @"(\w+)=(\w+)",
                                     "$2=$1");
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 Regex 对象：

```
Regex regexObj = new Regex(@"(\w+)=(\w+)");
string resultString = regexObj.Replace(subjectString, "$2=$1");
```

## VB.NET

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Dim ResultString = Regex.Replace(SubjectString, "(\w+)=(\w+)", "$2=$1")
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 Regex 对象：

```
Dim RegexObj As New Regex("(\w+)=(\w+)")
Dim ResultString = RegexObj.Replace(SubjectString, "$2=$1")
```

## Java

如果你打算只使用同一个正则表达式处理一个字符串，那么可以调用 String.replaceAll()：

```
String resultString = subjectString.replaceAll("(\\w+)=(\\w+)", "$2=$1");
```

如果要把同一个正则表达式用于大量字符串，那么就需要构造一个 Matcher 对象：

```
Pattern regex = Pattern.compile("(\w+)=(\w+)");
Matcher regexMatcher = regex.matcher(subjectString);
String resultString = regexMatcher.replaceAll("$2=$1");
```

## JavaScript

```
result = subject.replace(/(\w+)=(\w+)/g, "$2=$1");
```

## PHP

```
$result = preg_replace('/(\w+)=(\w+)/', '$2=$1', $subject);
```

## Perl

```
$subject =~ s/(\w+)=(\w+)/$2=$1/g;
```

## Python

如果你打算只使用同一个正则表达式处理少量字符串，可以使用如下的全局函数：

```
result = re.sub(r"(\w+)=(\w+)", r"\2=\1", subject)
```

要想重复使用同一个正则表达式，那么就需要使用一个编译好的对象：

```
reobj = re.compile(r"(\w+)=(\w+)")
result = reobj.sub(r"\2=\1", subject)
```

## Ruby

```
result = subject.gsub(/(\w+)=(\w+)/, '\2=\1')
```

## 讨论

正则表达式 `\w+=\w+` 会匹配一对单词，并把每个词放到自己的捕获分组中。在等号之前的单词会被捕获到第一个分组中，而等号之后的单词被放到第二个分组中。

在替换的时候，需要说明你要使用第二个分组匹配到的文本，然后是等号，再然后才是第一个捕获分组匹配到的文本。你可以在替代文本中使用特殊占位符来完成这样的功能。替代文本的语法在不同编程语言中是有很大区别的。在第 1 章中的“使用正则表达式查找和替换”一节中介绍了替代文本流派，在实例 2.21 中介绍了如何在替代文本中引用捕获分组。

## .NET

在 .NET 中，你还可以使用上一个实例中介绍的 `Regex.Replace()` 方法，并使用一个字符串作为替代文本。用来向替代文本中添加向后引用的语法遵循的是实例 2.21 中的 .NET 替代文本流派。

## Java

在 .NET 中，你还可以使用上一个实例中介绍的 `replaceFirst()` 和 `replaceAll()` 方法。用来向替代文本中添加向后引用的语法遵循的是本书中介绍的 Java 替代文本流派。

## JavaScript

在 JavaScript 中，你还可以使用上一个实例中介绍的 `string.replace()` 方法。用来向替代文本中添加向后引用的语法遵循的是本书中介绍的 JavaScript 替代文本流派。

## PHP

在 JavaScript 中，你还可以使用上一个实例中介绍的 `preg_replace()` 函数。用来向替代文本中添加向后引用的语法遵循的是本书中介绍的 PHP 替代文本流派。

## Perl

在 Perl 中，在 `s/regex/replace/` 中的 `replace` 部分会被简单地解释为一个双引号字符串。

你可以在替代文本中使用在实例 3.7 和实例 3.9 中介绍的特殊变量，包括\$&、\$1、\$2 等。在找到正则匹配之后，在替换执行之前会设置这些变量的值。你还可以在所有其他的 Perl 代码中使用这些变量。它们的值会一直保留到你下一次要求 Perl 查找另外一个正则匹配。

本书中所有其他的编程语言都会提供一个把替代文本作为字符串参数的函数调用。这个函数调用会对字符串进行分析，找到诸如 \$1 或 \1 这样的向后引用。但是在替代文本字符串之外，\$1 在这些语言中并不拥有任何含义。

## Python

在 Python 中，你还可以使用上一个实例所介绍的同样的 sub() 函数。用来向替代文本中添加向后引用的语法遵循的是本书中介绍的 Python 替代文本流派。

## Ruby

在 Ruby 中，你还可以使用上一个实例所介绍的同样的 String.gsub() 函数。用来向替代文本中添加向后引用的语法遵循的是本书中介绍的 Ruby 替代文本流派。

你不能在替代文本中插入像 \$1 这样的变量。这是因为 Ruby 不支持在 gsub() 调用执行之前进行变量插值。在调用之前，gsub() 还没有找到任何匹配，所以向后引用也无法进行替代。如果你试图插入 \$1 的值，那么得到的就会是在调用 gsub() 之前的上一个正则匹配中的第一个捕获分组匹配到的文本。

你应当使用的像 «\1» 这样的替代文本记号。gsub() 函数会为每个正则匹配替换在替代文本中的这些记号。这里推荐为替代文本使用单引号字符串。在双引号字符串中，反斜杠会被用作转义字符，而转义的数字是八进制转义。「\1」 和 "\\\1" 会使用第一个捕获分组匹配到的文本作为替代，而则 "\1" 会被替换为一个字面字符 0x01。

## 命名捕获

如果你在正则表达式中使用了命名捕获分组，那么就可以在替代字符串中使用名称来引用它们。

## C#

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
string resultString = Regex.Replace(subjectString,
    @"\(?<left>\w+\)=\(?<right>\w+\)", "${right}=${left}");
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 Regex 对象：

```
Regex regexObj = new Regex(@"\(?<left>\w+\)=\(?<right>\w+\)");
string resultString = regexObj.Replace(subjectString, "${right}=${left}");
```

## VB.NET

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Dim ResultString = Regex.Replace(SubjectString,  
    "(?<left>\w+)= (?<right>\w+)", "${right}=${left}")
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 Regex 对象：

```
Dim RegexObj As New Regex("(?<left>\w+)= (?<right>\w+)")  
Dim ResultString = RegexObj.Replace(SubjectString, "${right}=${left}")
```

## PHP

```
$result = preg_replace('/(?P<left>\w+)= (?P<right>\w+)/',  
    '$2=$1', $subject);
```

PHP 中的 preg 函数使用的是 PCRE 函数库，而后者会支持命名捕获。preg\_match() 和 preg\_match\_all() 两个函数会把命名捕获分组添加到匹配结果所在的数组中。不幸的是，preg\_replace() 并没有提供一种方式来在替代文本中使用向后引用。如果你的正则表达式中包含命名捕获分组，那么就需要从左向右统计所有命名和非命名的捕获分组来确定每个分组的向后引用编号。在替代文本中使用这些编号。

## Perl

```
$subject =~ s/(?<left>\w+)= (?<right>\w+)/$+{right}=$+{left}/g;
```

Perl 从 5.10 版开始支持命名捕获分组。“\$+ hash” 会保存上一次用到的正则表达式中所有命名捕获分组匹配到的文本。同在其他任何地方一样，你也可以在替代文本字符串中使用这样的哈希（hash）。

## Python

如果你打算只使用同一个正则表达式处理少量字符串，可以使用如下的全局函数：

```
result = re.sub(r"(?P<left>\w+)= (?P<right>\w+)", r"\g<right>=\g<left>",  
    subject)
```

要想重复使用同一个正则表达式，那么就需要使用一个编译好的对象：

```
reobj = re.compile(r"(?P<left>\w+)= (?P<right>\w+)")  
result = reobj.sub(r"\g<right>=\g<left>", subject)
```

## Ruby

```
result = subject.gsub(/(?<left>\w+)= (?<right>\w+)/, '\k<left>=\k<right>')
```

## 参见

第 1 章中“使用正则表达式查找和替换”小节中介绍了替代文本的流派。

实例 2.21 中讲解了如何在替代文本中引用捕获分组。

## 3.16 使用代码中生成的替代文本来替换匹配

### 问题描述

你想要把正则表达式的所有匹配都替换为你在过程代码中构造的一个新的字符串。你想要能够把每个匹配替换为按照实际匹配到的内容来决定的不同字符串。

例如，假设你想要把一个字符串中的所有数字都替换为该数字的 2 倍。

### 解决方案

#### C#

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
string resultString = Regex.Replace(subjectString, @"\d+",
                                     new MatchEvaluator(ComputeReplacement));
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 `Regex` 对象：

```
Regex regexObj = new Regex(@"\d+");
string resultString = regexObj.Replace(subjectString,
                                         new MatchEvaluator(ComputeReplacement));
```

这两个代码片段中都调用了函数 `ComputeReplacement`。你应当把这个方法添加到用来实现这个解答的类中：

```
public String ComputeReplacement(Match matchResult) {
    int twiceasmuch = int.Parse(matchResult.Value) * 2;
    return twiceasmuch.ToString();
}
```

#### VB.NET

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf ComputeReplacement)
Dim ResultString = Regex.Replace(SubjectString, "\d+", MyMatchEvaluator)
```

如果想要把同一个正则表达式应用于大量的字符串，那么就需要构造一个 `Regex` 对象：

```
Dim RegexObj As New Regex("\d+")
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf ComputeReplacement)
Dim ResultString = RegexObj.Replace(SubjectString, MyMatchEvaluator)
```

这两个代码片段中都调用了函数 `ComputeReplacement`。你应当把这个方法添加到用来实现这个解答的类中：

```
Public Function ComputeReplacement(ByVal MatchResult As Match) As String
    Dim TwiceAsMuch = Int.Parse(MatchResult.Value) * 2;
    Return TwiceAsMuch.ToString();
End Function
```

## Java

```
StringBuffer resultString = new StringBuffer();
Pattern regex = Pattern.compile("\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
    Integer twiceasmuch = Integer.parseInt(regexMatcher.group()) * 2;
    regexMatcher.appendReplacement(resultString, twiceasmuch.toString());
}
regexMatcher.appendTail(resultString);
```

## JavaScript

```
var result = subject.replace(/\d+/g,
                            function(match) { return match * 2; });

```

## PHP

使用一个声明的回调函数：

```
$result = preg_replace_callback('/\d+/', compute_replacement, $subject);
function compute_replacement($groups) {
    return $groups[0] * 2;
}
```

使用一个匿名的回调函数：

```
$result = preg_replace_callback(
    '/\d+/',
    create_function(
        '$groups',
        'return $groups[0] * 2;'
    ),
    $subject
);
```

## Perl

```
$subject =~ s/\d+/$& * 2/ge;
```

## Python

如果你打算只使用同一个正则表达式处理少量字符串，可以使用如下的全局函数：

```
result = re.sub(r"\d+", computereplacement, subject)
```

要想重复使用同一个正则表达式，那么就需要使用一个编译好的对象：

```
reobj = re.compile(r"\d+")
result = reobj.sub(computereplacement, subject)
```

这两个代码片段中都调用了函数 `computereplacement`。这个函数需要在你把它传递给 `sub()` 之前进行声明：

```
def computereplacement(matchobj):
    return str(int(matchobj.group()) * 2)
```

## Ruby

```
result = subject.gsub(/\d/) {|match|
    Integer(match) * 2
}
```

## 讨论

当使用字符串作为替代文本的时候，你只能做最基本的文本替换。要想把每个匹配根据要替换的匹配内容替换成完全不同的字符串，那么就需要在自己的代码中构造替代文本。

## C#

实例 3.14 中讨论了可以调用 `Regex.Replace()` 方法的不同方式，都是把替代文本作为一个字符串参数。当使用静态调用的时候，替代文本是第三个参数，位于目标字符串和正则表达式之后。如果把正则表达式传递给 `Regex()` 构造函数，那么你就可以在这个对象之上只用替代文本作为第二个参数来调用 `Replace()`。

这时候，我们不再传递一个字符串作为第二或第三个参数，我们传递的是一个 `MatchEvaluator` 委派（delegate）。这个委派是一个引用，指向的是你添加到正在做查找和替换的类中的一个成员函数。要创建这个委派，需要使用 `new` 关键字来调用 `MatchEvaluator()` 构造函数。把你的成员函数作为传递给 `MatchEvaluator()` 的唯一参数。

你想要用于委派的函数应当返回一个字符串，并且接受一个类型为 `System.Text.RegularExpressions.Match` 类的参数。这与本章中前面所给的实例中的 `Regex.Match()` 成员所返回的 `Match` 类是同一个。

当你用一个 `MatchEvaluator` 作为替代来调用 `Replace()` 时，需要被替换的每个正则表达式匹配都会调用你的函数。你的函数需要返回一个替代文本。可以使用 `Match` 对象的任意属性来构造你的替代文本。在前面给出的例子中使用了 `matchResult.Value` 来获取整个正则匹配的字符串。通常，你也可以使用 `matchResult.Groups[]` 从正则表达式的捕

获分组中构造你的替代文本。

如果你不想替换某些特定的正则匹配，你的函数应当返回 `matchResult.Value`。如果你返回的是 `null` 或者是一个空串，那么正则匹配就会被替换为空（也就是被删除）。

## VB.NET

实例 3.14 中讨论了你可以调用 `Regex.Replace()` 方法的不同方式，都是把替代文本作为一个字符串参数。当使用静态调用的时候，替代文本是第三个参数，位于目标字符串和正则表达式之后。如果你使用 `Dim` 关键字为正则表达式创建了一个对象，那么你就可以在这个对象之上只用替代文本作为第二个参数来调用 `Replace()`。

这时候，我们不再传递一个字符串作为第二或第三个参数，我们传递的是一个 `MatchEvaluator` 对象。这个对象中包含一个引用，指向的是你添加到正在做查找和替换的类中的一个函数。你需要使用 `Dim` 关键字来创建类型为 `MatchEvaluator()` 的一个变量。`AddressOf` 操作符会返回到你的函数的一个引用，而不会在这里实际调用该函数。

你想要用于 `MatchEvaluator` 的函数应当返回一个字符串，并且接受一个类型为 `System.Text.RegularExpressions.Match` 类的参数。这与本章中前面所给的实例中的 `Regex.Match()` 成员所返回的 `Match` 类是同一个。这里的参数是通过传值来传递的，所以你必须把它用 `ByVal` 来声明。

当你用一个 `MatchEvaluator` 作为替代来调用 `Replace()` 的时候，需要被替换的每个正则表达式匹配都会调用你的函数。函数需要返回一个替代文本。而你可以使用 `Match` 对象的任意属性来构造你的替代文本。在前面给出的例子中使用了 `MatchResult.Value` 来获取整个正则匹配的字符串。通常，你也可以使用 `MatchResult.Groups[]` 从正则表达式的捕获分组中构造你的替代文本。

如果不替换某些特定的正则匹配，你的函数应当返回 `matchResult.Value`。如果你返回的是 `Nothing` 或者是一个空串，那么正则匹配就会被替换为空（也就是被删除）。

## Java

Java 的解决方案看起来很直接。我们会按照实例 3.11 中所讲的方式遍历所有的正则匹配。在循环中，我们使用 `Matcher` 对象来调用 `appendReplacement()`。当 `find()` 不能再找到更多匹配的时候，我们会调用 `appendTail()`。`appendReplacement()` 和 `appendTail()` 这两个方法使得我们可以很容易为每个正则匹配使用一个不同的替代文本。

`appendReplacement()` 接受两个参数。第一个类型是 `StringBuffer`，用来（临时）存储正在进行的查找和替换的结果，第二个是用于上一次由 `find()` 找到匹配的替代文本。这个替代文本中可以包含到捕获分组的引用，例如 "\$1"。如果在你的替代文本中有语法错误的话，那么会产生一个 `IllegalArgumentException` 例外。如果替代文本引用

了一个不存在的捕获分组，那么会产生一个 `IndexOutOfBoundsException` 例外。如果你在没有进行成功的 `find()` 调用之前就调用了 `appendReplacement()`，那么会产生一个 `IllegalStateException` 例外。

如果你正确地调用了 `appendReplacement()`，那么它会做两件事情。首先，它会把位于上一次和当前的正则匹配之间的文本复制到字符串缓冲区中，而不会对该文本做任何修改。如果当前匹配是第一个匹配，那么它会复制在该匹配之前的所有字符。在此之后，它会把你的替代文本附加上来，并把其中存在的任何向后引用替换为相应捕获分组匹配到的文本。

如果你想要删除某个特定的匹配，那么可以把它替换为一个空串。如果你想要让字符串中的某个匹配保持不变，那么可以对该匹配省去到 `appendReplacement()` 的调用。当提到“上一次正则匹配”的时候，指的是你调用 `appendReplacement()` 的上一个匹配。如果你没有为某个匹配调用 `appendReplacement()`，那么这些匹配就会成为位于你替换了的两个匹配之间的文本，因此也就会原封不动复制到目标字符串缓冲区中。

完成了匹配替换之后，需要调用 `appendTail()`。这个函数会把字符串中位于最后一个调用了 `appendReplacement()` 的正则匹配之后的剩余文本复制到缓冲区中。

## JavaScript

在 JavaScript 中，一个函数其实就是一个可以被赋值到变量的对象。在这里，传递给 `string.replace()` 函数的不再是一个字面字符串，或者是一个包含字符串的变量，我们这次会传递一个返回字符串的函数。这个函数在每次需要进行替换的时候都会被调用。

可以让你的替代函数接受一个或多个参数。如果这样做，第一个参数会被设为正则表达式匹配到的文本。如果你的正则表达式包含捕获分组，那么第二个参数中会保存第一个捕获分组匹配到的文本，第三个参数会保存第二个捕获分组的文本，以此类推。你可以在这些参数中使用正则表达式匹配的片段来构成替代文本。

在这个实例中给出的 JavaScript 解决方案中的替换函数只是接受了正则表达式匹配的文本，并返回它的 2 倍。JavaScript 会自动处理字符串到整数和整数到字符串的转换。

## PHP

`preg_replace_callback()` 函数与在实例 3.14 中讲解的 `preg_replace()` 函数的工作原理是相同的。它会接受一个正则表达式、替代、目标字符串、可选的替换限制以及可选的替换计数。正则表达式和目标字符串可以使用单个字符串或者数组。

区别是 `preg_replace_callback()` 并不会接受一个字符串或字符串数组作为替代；它接受的是一个函数。你可以在代码中声明这个函数，或者使用 `create_function()` 创建一个匿名

函数。这个函数应当接受一个参数，并返回一个字符串（或者可以被强制转换为字符串的类型）。

每次当 `preg_replace_callback()` 找到一个正则匹配的时候，它都会调用你提供的回调函数。它的参数会被填充为一个字符串数组。元素 0 保存的是整个的正则匹配，而元素 1 之后保存的则是由各个捕获分组匹配到的文本。你可以使用这个数组来构造替代文本，既可以使用正则表达式的匹配，也可以使用一个或者多个捕获分组的匹配。

## Perl

`s///` 操作符支持一个会被 `m//` 忽略的额外修饰符：`/e`。这个`/e` 或“execute（执行）”修饰符会告诉替换操作符把替代部分当作 Perl 代码来执行，而不是把它解释为双引号字符串的内容。使用了这个修饰符，我们就可以很容易获取变量 `$&` 中的匹配文本，并把它乘以 2。这段代码的执行结果会被当作是替代字符串。

## Python

Python 的 `sub()` 函数会允许你不传递字符串，而是传递一个函数来作为替代文本。在每次需要替换正则匹配时都会调用这个函数。

你需要在引用这个函数之前声明它。它应当接受一个 `MatchObject` 实例的参数，这个对象同 `search()` 函数返回的对象是同一个。你可以使用它来获取正则匹配（或者其中的一部分），从而可以构造自己的替代文本。更多细节可以参考实例 3.7 和实例 3.9。

你的函数应当返回包含替代文本的一个字符串。

## Ruby

前面两个实例中调用 `String` 类的 `gsub()` 函数时使用了 2 个参数：正则表达式和替代文本。这个方法还存在一种 `block` 形式。

在 `block` 形式中，`gsub()` 会接受正则表达式作为唯一参数。它会使用正则表达式匹配到的文本构成的字符串来填充一个迭代器变量。如果你提供额外的迭代器变量，那么它们会被设置为 `nil`，即使你的正则表达式中包含有捕获分组也一样。

在 `block` 中放一个表达式，用来对你想要用作替代文本的字符串进行求值。在 `block` 中，你可以使用特殊的正则匹配变量，例如 `$~`、`$&` 和 `$1`。在 `block` 每次被执行来进行替换时，这些变量的值都会改变。更多细节请参考实例 3.7、3.8 和 3.9。

你不能使用诸如 `«\1»` 这样的替代文本记号。这些记号依然会被当作字面文本来处理。

## 参见

实例 3.9 和 3.15。

## 3.17 替换另一个正则式匹配中的所有匹配

### 问题描述

你想要替换某个特定正则表达式的所有匹配，但是只有当它位于目标字符串的特定子串中时才会这样做。另外一个正则表达式会匹配字符串中的这些子串。

假设你有一个 HTML 文件，其中有不同的段落使用**<b>** tag 被标记为粗体。在每对粗体 tag 之间，你想要把正则表达式 `<before>` 的所有匹配都替换为 `<after>`。例如，当处理字符串 `before <b>first before</b> before <b>before before</b>` 的时候，你最终会得到的是 `before <b>first after</b> before <b>after after</b>`。

### 解决方案

#### C#

```
Regex outerRegex = new Regex("<b>.*?</b>", RegexOptions.Singleline);
Regex innerRegex = new Regex("before");

string resultString = outerRegex.Replace(subjectString,
                                         new MatchEvaluator(ComputeReplacement));
public String ComputeReplacement(Match matchResult) {
    // Run the inner search-and-replace on each match of the outer regex
    return innerRegex.Replace(matchResult.Value, "after");
}
```

#### VB.NET

```
Dim OuterRegex As New Regex("<b>.*?</b>", RegexOptions.Singleline)
Dim InnerRegex As New Regex("before")
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf ComputeReplacement)
Dim ResultString = OuterRegex.Replace(SubjectString, MyMatchEvaluator)
Public Function ComputeReplacement(ByVal MatchResult As Match) As String
    'Run the inner search-and-replace on each match of the outer regex
    Return InnerRegex.Replace(MatchResult.Value, "after");
End Function
```

#### Java

```
StringBuffer resultString = new StringBuffer();
Pattern outerRegex = Pattern.compile("<b>.*?</b>");
Pattern innerRegex = Pattern.compile("before");
Matcher outerMatcher = outerRegex.matcher(subjectString);
while (outerMatcher.find()) {
    outerMatcher.appendReplacement(resultString,
```

```
        innerRegex.matcher(outerMatcher.group()).replaceAll("after"));
    }
    outerMatcher.appendTail(resultString);
}
```

## JavaScript

```
var result = subject.replace(/<b>.*?</b>/g,
                            function(match) {
                                return match.replace(/before/g, "after");
                            });
}
```

## PHP

```
$result = preg_replace_callback('%<b>.*?</b>%',
                                replace_within_tag, $subject);
function replace_within_tag($groups) {
    return preg_replace('/before/', 'after', $groups[0]);
}
```

## Perl

```
$subject =~ s%<b>.*?</b>%($match = $&) =~ s/before/after/g;
$match;%eg;
```

## Python

```
innerre = re.compile("before")
def replacewithin(matchobj):
    return innerre.sub("after", matchobj.group())
result = re.sub("<b>.*?</b>", replacewithin, subject)
```

## Ruby

```
innerre = /before/
result = subject.gsub(/<b>.*?</b>/) {|match|
    match.gsub(innerre, 'after')
}
```

## 讨论

这个解答同样是前面两个解答的组合，其中使用了两个正则表达式。“外层”正则表达式，`<b>.*?</b>`，会匹配 HTML 粗体 tag 以及两个 tag 之间的文本。“内层”正则表达式会匹配“before”，然后把它替换为“after”。

实例 3.16 中讲解了如何运行查找和替换，并且在自己的代码中如何为每个正则表达式构造替代文本。在这里，我们使用外层正则表达式来完成这个功能。每次它找到一对

起始和结束的 <b> tag 时，我们都会使用内层正则表达式来执行查找和替换，这与我们在实例 3.14 中所做的是一样的。内层正则表达式查找和替换的目标字符串就是外层正则表达式匹配到的文本。

## 参见

实例 3.11、3.13 和 3.16。

## 3.18 替换另一个正则式匹配之间的所有匹配

### 问题描述

你想要替换某个特定正则表达式的所有匹配，但是只在目标字符串的某些片段中进行。另外一个正则表达式会匹配位于这些片段之间的文本。换句话说，你想要查找和替换另外一个正则表达式不能匹配的目标字符串中的那些片段。

假设你有一个 HTML 文件，想要把其中垂直的双引号替换为智能（弯曲的）双引号，但是你只想替换那些位于 HTML tag 之外的引号。因为位于 HTML tag 之内的引号必须要使用 ASCII 的垂直引号，不然你的 web 浏览器可能就会无法分析 HTML 文件了。例如，你想要把 "text" <span class="middle">"text"</span> "text" 转换成 “text” <span class="middle">“text”</span> “text”。

### 解决方案

#### C#

```
string resultString = null;
Regex outerRegex = new Regex("<[^<>]*>");
Regex innerRegex = new Regex("\"([^\"]*)\"");
// Find the first section
int lastIndex = 0;
Match outerMatch = outerRegex.Match(subjectString);
while (outerMatch.Success) {
    // Search-and-replace through the text between this match,
    // and the previous one
    string textBetween =
        subjectString.Substring(lastIndex, outerMatch.Index - lastIndex);
    resultString = resultString +
        innerRegex.Replace(textBetween, "\u201C$1\u201D");
    lastIndex = outerMatch.Index + outerMatch.Length;
    // Copy the text in the section unchanged
    resultString = resultString + outerMatch.Value;
    // Find the next section
    outerMatch = outerMatch.NextMatch();
}
```

```

// Search-and-replace through the remainder after the last regex match
string textAfter = subjectString.Substring(lastIndex,
                                              subjectString.Length - lastIndex);
resultString = resultString + innerRegex.Replace(textAfter,
                                                 "\u201C$1\u201D");

```

## VB.NET

```

Dim ResultString As String = Nothing
Dim OuterRegex As New Regex("<[^<>]*>")
Dim InnerRegex As New Regex("""([^\"]*)""")
'Find the first section
Dim LastIndex = 0
Dim OuterMatch = OuterRegex.Match(SubjectString)
While OuterMatch.Success
    'Search-and-replace through the text between this match,
    'and the previous one
    Dim TextBetween = SubjectString.Substring(LastIndex,
                                                OuterMatch.Index - LastIndex);
    ResultString = ResultString + InnerRegex.Replace(TextBetween,
                                                    ChrW(&H201C) + "$1" + ChrW(&H201D))
    LastIndex = OuterMatch.Index + OuterMatch.Length
    'Copy the text in the section unchanged
    ResultString = ResultString + OuterMatch.Value
    'Find the next section
    OuterMatch = OuterMatch.NextMatch
End While
'Search-and-replace through the remainder after the last regex match
Dim TextAfter = SubjectString.Substring(LastIndex,
                                         SubjectString.Length - LastIndex);
ResultString = ResultString +
    InnerRegex.Replace(TextAfter, ChrW(&H201C) + "$1" + ChrW(&H201D))

```

## Java

```

StringBuffer resultString = new StringBuffer();
Pattern outerRegex = Pattern.compile("<[^<>]*>");
Pattern innerRegex = Pattern.compile("\"([^\"]*)\"");
Matcher outerMatcher = outerRegex.matcher(subjectString);
int lastIndex = 0;
while (outerMatcher.find()) {
    // Search-and-replace through the text between this match,
    // and the previous one
    String textBetween = subjectString.substring(lastIndex,
                                                outerMatcher.start());
    Matcher innerMatcher = innerRegex.matcher(textBetween);
    resultString.append(innerMatcher.replaceAll("\u201C$1\u201D"));
    lastIndex = outerMatcher.end();
    // Append the regex match itself unchanged
}

```

```

        resultString.append(outerMatcher.group());
    }
    // Search-and-replace through the remainder after the last regex match
    String textAfter = subjectString.substring(lastIndex);
    Matcher innerMatcher = innerRegex.matcher(textAfter);
    resultString.append(innerMatcher.replaceAll("\u201C$1\u201D"));
}

```

## JavaScript

```

var result = "";
var outerRegex = /<[^>]*>/g;
var innerRegex = /"(^[^"]*)" /g;
var outerMatch = null;
var lastIndex = 0;
while (outerMatch = outerRegex.exec(subject)) {
    if (outerMatch.index == outerRegex.lastIndex) outerRegex.lastIndex++;
    // Search-and-replace through the text between this match,
    // and the previous one
    var textBetween = subject.substring(lastIndex, outerMatch.index);
    result = result + textBetween.replace(innerRegex, "\u201C$1\u201D");
    lastIndex = outerMatch.index + outerMatch[0].length;
    // Append the regex match itself unchanged
    result = result + outerMatch[0];
}
// Search-and-replace through the remainder after the last regex match
var textAfter = subject.substr(lastIndex);
result = result + textAfter.replace(innerRegex, "\u201C$1\u201D");

```

## PHP

```

$result = '';
$lastindex = 0;
while (preg_match('/<[^>]*>/', $subject, $groups, PREG_OFFSET_CAPTURE,
                  $lastindex)) {
    $matchstart = $groups[0][1];
    $matchlength = strlen($groups[0][0]);
    // Search-and-replace through the text between this match,
    // and the previous one
    $textbetween = substr($subject, $lastindex, $matchstart-$lastindex);
    $result .= preg_replace('/"(^[^"]*)" /', '"$1"', $textbetween);
    // Append the regex match itself unchanged
    $result .= $groups[0][0];
    // Move the starting position for the next match
    $lastindex = $matchstart + $matchlength;
    if ($matchlength == 0) {
        // Don't get stuck in an infinite loop
        // if the regex allows zero-length matches
        $lastindex++;
    }
}

```

```
// Search-and-replace through the remainder after the last regex match
$textafter = substr($subject, $lastindex);
$result .= preg_replace('/"(^\")*/"', '"$1"', $textafter);
```

## Perl

```
use encoding "utf-8";
$result = '';
while ($subject =~ m/<[^<>]*>/g) {
    $match = $&;
    $textafter = $';
    ($textbetween = $') =~ s/"(^\")"/\x{201C}$1\x{201D}/g;
    $result .= $textbetween . $match;
}
$textafter =~ s/"(^\")"/\x{201C}$1\x{201D}/g;
$result .= $textafter;
```

## Python

```
innerre = re.compile('"(^\")*"')
result = "";
lastindex = 0;
for outermatch in re.finditer("<[^<>]*>", subject):
    # Search-and-replace through the text between this match,
    # and the previous one
    textbetween = subject[lastindex:outermatch.start()]
    result += innerre.sub(u"\u201C\\1\u201D", textbetween)
    lastindex = outermatch.end()
    # Append the regex match itself unchanged
    result += outermatch.group()
# Search-and-replace through the remainder after the last regex match
textafter = subject[lastindex:]
result += innerre.sub(u"\u201C\\1\u201D", textafter)
```

## Ruby

```
result = '';
textafter = ''
subject.scan(/<[^<>]*>/) {|match|
    textafter = $'
    textbetween = $.gsub(/"(^\")"/, '"\1"')
    result += textbetween + match
}
result += textafter.gsub(/"(^\")"/, '"\1"')
```

## 讨论

实例 3.13 讲解了如何使用两个正则表达式找到位于文件中特定部分（第一个正则式的

匹配) 中的匹配 (第二个正则式)。这个实例的解答也会使用同样的技巧来只对目标字符串中的特定片段进行查找和替换。

需要注意的一点是你用来查找片段的正则表达式总是在对原始的目标字符串进行操作。如果修改了原始的目标字符串，那么你就必须在内层正则式添加或删除字符的同时，修改外层正则式的起始位置。更重要的是，这样的修改可能会带来意想不到的副作用。例如，如果你的外层正则式使用了定位符 `<^>` 来匹配一行起始处的内容，而内层正则式则在外层正则式找到的每个片段的结尾添加一个换行符，那么由于新添加了换行符，那么外层正则式中的 `<^>` 就会匹配到上一个片段之后的位置。

虽然这个实例的解决方案看起来都比较长，但是实际上它们并不难理解。我们使用了两个正则表达式。“外层” 正则表达式，`<<[^>]*>>`，会匹配一对尖括号以及二者之间除了尖括号之外的所有内容。这是一种用来匹配任意 HTML tag 的比较粗糙的方法。只要在 HTML 文件中不包含任何没有被编码为实体的字面尖括号的情况下，这个解答都是可以正确工作的。我们实现这个正则表达式的代码与在实例 3.11 中给出的代码是相同的。唯一的区别是在前面代码中的占位符注释被替换为进行实际查找和替换的代码。

在循环中进行查找和替换的代码则参照了在实例 3.14 中给出的代码。查找和替换使用的目标字符串是在外层正则式的上一次匹配和这次匹配之间的文本。我们把内层正则式查找和替换的结果附加到总的结果字符串之上。我们同样还把外层正则表达式的匹配原封不动地添加到结果字符串中。

当外层正则式无法找到更多匹配的时候，我们会再一次执行内存查找和替换操作，这次针对的是外层正则式最后一次匹配之后的文本。

在循环中用于查找和替换的正则式，`<"[^"]*>"`，会匹配一对双引号字符以及二者之间除了双引号之外的所有内容。这些位于引号之间的文本会被捕获到第一个捕获分组中。

对于替代文本来说，我们使用了到第一个捕获分组的引用，它被放到了两个智能引号之间。智能引号在 Unicode 代码点中的位置是 U+201C 和 U+201D。通常来说，你可以简单地把智能引号直接粘贴到你的源代码中。然而，Visual Studio 2008 则坚持认为自己更聪明，它会把所有代码中的字面智能引号都自动替换为垂直引号。

在正则表达式中，你可以使用 `\u201C` 或 `\x{201C}` 匹配一个 Unicode 代码点，但是本书中讨论的所有编程语言都不会在替代文本中支持这种记号。如果最终用户想要在使用编辑控制框输入的替代文本中插入智能引号，那么就必须从字符映射表中粘贴过来才行。在源代码中，如果你的编程语言支持把 Unicode 转义作为字面字符串的一部分，那么你可以在替代文本中使用 Unicode 转义序列。例如，C# 和 Java 支持在字符串中使用 \u201C，但 VB.NET 则不支持在字符串中对 Unicode 字符进行转义。在 VB.NET 中，你可以使用 ChrW 函数来把 Unicode 代码点转换为一个字符。

## Perl 和 Ruby

Perl 和 Ruby 的解答中使用了我们还没有介绍过的两个特殊变量。`$'`(美元符号+反引号) 中包含目标匹配左边的文本，而`$'`(美元符号+单引号) 则会包含目标匹配右边的文本。我们没有在原来的目标字符串之上遍历匹配，而是在前一个匹配之后的字符串片段上开始一次新的搜索。这样，我们用变量`$'`就可以很容易获取本次匹配和上一次匹配之间的文本。

## Python

这段代码的结果是一个 Unicode 字符串，因为替代文本被说明为一个 Unicode 字符串。你可能会需要调用 `encode()` 才能显示这个字符串，例如：

```
print result.encode('1252')
```

## 参见

实例 3.11、3.13 和 3.16。

## 3.19 拆分字符串

### 问题描述

你想要使用正则表达式来拆分一个字符串。在拆分之后，你会得到由位于正则表达式匹配之间的字符串组成的一个数组或者列表。

例如，你想要使用 HTML tag 来拆分一个包含 HTML tag 的字符串。把 `I●like●<b>bold</b>●and●<i>italic</i>●fonts` 拆分之后应当得到一个包含如下 5 个字符串的数组：`I●like●`、`bold`、`●and●`、`italic` 和`●fonts`。

### 解决方案

#### C#

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
string[] splitArray = Regex.Split(subjectString, "<[^<>]*>");
```

如果该正则式是由最终用户提供的，那么你就需要在使用静态调用时进行完整的例外处理：

```
string[] splitArray = null;
try {
    splitArray = Regex.Split(subjectString, "<[^<>]*>");
} catch (ArgumentNullException ex) {
```

```
// Cannot pass null as the regular expression or subject string
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

如果你想要把同一个正则表达式用于大量的字符串之上，那么就需要构造一个 Regex 对象：

```
Regex regexObj = new Regex("<[^<>]*>");
string[] splitArray = regexObj.Split(subjectString);
```

如果该正则式是由最终用户提供的，那么你应当使用带有完整例外处理的 Regex 对象：

```
string[] splitArray = null;
try {
    Regex regexObj = new Regex("<[^<>]*>");
    try {
        splitArray = regexObj.Split(subjectString);
    } catch (ArgumentNullException ex) {
        // Cannot pass null as the subject string
    }
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

## VB.NET

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Dim SplitArray = Regex.Split(SubjectString, "<[^<>]*>")
```

如果该正则式是由最终用户提供的，那么你就需要在使用静态调用时进行完整的例外处理：

```
Dim SplitArray As String()
Try
    SplitArray = Regex.Split(SubjectString, "<[^<>]*>")
Catch ex As ArgumentNullException
    'Cannot pass null as the regular expression or subject string
Catch ex As ArgumentException
    'Syntax error in the regular expression
End Try
```

如果你想要把同一个正则表达式用于大量的字符串之上，那么就需要构造一个 Regex 对象：

```
Dim RegexObj As New Regex("<[^<>]*>")
Dim SplitArray = RegexObj.Split(SubjectString)
```

如果该正则式是由最终用户提供的，那么你应当使用带有完整例外处理的 Regex 对象：

```
Dim SplitArray As String()
Try
    Dim RegexObj As New Regex("<[^<>]*>")
    Try
        SplitArray = RegexObj.Split(SubjectString)
    Catch ex As ArgumentNullException
        'Cannot pass null as the subject string
    End Try
    Catch ex As ArgumentException
        'Syntax error in the regular expression
    End Try
```

## Java

当你打算只使用同一个正则表达式处理一个字符串的时候，可以直接调用 `String.Split()`：

```
String[] splitArray = subjectString.split("<[^<>]*>");
```

如果该正则式是由最终用户提供的，那么你还需要进行完整的例外处理：

```
try {
    String[] splitArray = subjectString.split("<[^<>]*>");
} catch (PatternSyntaxException ex) {
    // Syntax error in the regular expression
}
```

如果想要把同一个正则表达式用于大量的字符串之上，那么就需要构造一个 `Pattern` 对象：

```
Pattern regex = Pattern.compile("<[^<>]*>");
String[] splitArray = regex.split(subjectString);
```

如果该正则式是由最终用户提供的，那么你应当使用带有完整例外处理的 `Pattern` 对象：

```
String[] splitArray = null;
try {
    Pattern regex = Pattern.compile("<[^<>]*>");
    splitArray = regex.split(subjectString);
} catch (ArgumentException ex) {
    // Syntax error in the regular expression
}
```

## JavaScript

`string.split()` 函数可以使用正则表达式来拆分字符串：

```
result = subject.split(/<[^<>]*>/);
```

但是，当用 `string.split()` 来处理正则表达式的时候，会存在许多跨浏览器的问题。如果用自己的代码来构造一个列表会更加可靠：

```
var list = [];
var regex = /<[^<>]*>/g;
var match = null;
var lastIndex = 0;
while (match = regex.exec(subject)) {
    // Don't let browsers such as Firefox get stuck in an infinite loop
    if (match.index == regex.lastIndex) regex.lastIndex++;
    // Add the text before the match
    list.push(subject.substring(lastIndex, match.index));
    lastIndex = match.index + match[0].length;
}
// Add the remainder after the last match
list.push(subject.substr(lastIndex));
```

## PHP

```
$result = preg_split('/<[^<>]*>/', $subject);
```

## Perl

```
@result = split(m/<[^<>]*>/, $subject);
```

## Python

如果只需要处理少量字符串，你可以使用全局函数：

```
result = re.split("<[^<>]*>", subject)
```

如果要反复使用同一个正则表达式，那么就需要使用一个编译好的对象：

```
reobj = re.compile("<[^<>]*>")
result = reobj.split(subject)
```

## Ruby

```
result = subject.split(/<[^<>]*>/)
```

## 讨论

使用正则表达式拆分字符串实质上就是要产生与实例 3.10 相反的结果。我们想要获取的不是所有正则匹配的一个列表，而是位于匹配之间的文本的一个列表，其中也包含了在第一个匹配之前和最后一个匹配之后的文本。正则匹配自身则不会出现在拆分函数的输出中。

## C# and VB.NET

在 .NET 中，你总是可以使用 `Regex.Split()` 方法来利用正则表达式拆分一个字符串。`Split()` 期望的第一个参数是包含你想要拆分的目标文本的字符串。这个参数不能是 `null`。

如果它是空的话，那么 Split()就会产生一个 ArgumentNullException 例外。Split()的返回值永远是一个字符串数组。

如果只需要使用一个正则表达式少数几次，那么你可以使用静态调用。第二个参数因此就是你想要使用的正则表达式。你还可以在可选的第三个参数中传递正则选项。如果你的正则表达式中包含语法错误的话，那么就会产生一个 ArgumentException 例外。

如果想要在多个字符串之上使用同一个正则表达式，那么你可以首先构造一个 Regex 对象，然后用它来调用 Split()，这样的做法效率更高。这样的话唯一必需的参数就是目标字符串。

当使用 Regex 类的一个实例来调用 Split()时，你还可以传递一些额外的参数来对拆分操作进行限制。如果你不用这些参数，那么字符串就会拆分目标字符串中的所有正则表达式匹配。Split()的静态重载形式不支持使用这些额外的参数。它们总是会按照所有的匹配来拆分目标字符串。

在目标字符串之后的可选的第二个参数中，可以指定你想要最终得到的拆分字符串的最大数目。例如，如果你调用 `regexObj.Split(subject, 3)`，那么就会得到一个最多包含 3 个字符串的数组。Split()函数会尝试找到两个正则匹配，然后返回一个数组，其中包括第一个匹配之前的文本、两个匹配之间的文本和第二个匹配之后的文本。在该目标字符串之后所有可能的正则匹配都会被忽略掉，因此会全部被放到数组中的最后一个字符串里。

如果没有足够的正则匹配来满足你的上限，那么 Split()会按照所有可用的正则匹配来进行拆分，它返回的数组中包含的字符串数量因此会比你指定的参数要少。`regexObj.Split(subject, 1)`不会对字符串进行拆分，而是直接把整个目标字符串作为数组的唯一元素返回。`regexObj.Split(subject, 0)`会按照所有正则匹配进行拆分，它的效果和忽略第二个参数时的 Split()是一样的。如果使用一个负数作为参数，那么就会导致 Split()产生一个 ArgumentOutOfRangeException 例外。

如果你使用第二个参数指定了返回数组中的最大字符串数量，那么还可以指定一个可选的第三个参数来说明正则表达式应当开始查找匹配的字符索引位置。实质上，你传递给第三个参数的整数，也就是你想要正则表达式忽略的位于字符串开头的字符数量。在你已经开始对字符串进行了一部分处理，而只想对剩余的字符串进行拆分时，就会需要用到这样的处理。

正则表达式忽略的字符还是会被添加到返回的数组中。数组中的第一个字符串是在你指定的起始位置之后的第一个正则匹配之前的整个子串，其中也包含了在指定起始位置之前的所有字符。如果你指定了第三个参数，那么它必须要大于 0，并且小于目标字符串的长度。否则，Split()会产生一个 ArgumentOutOfRangeException 例外。与 Match()不一样的是，Split()不会允许你指定一个参数来设定想要正则表达式搜索

的子串的长度。

如果在目标字符串的开始产生了一个匹配，那么在返回数组中的第一个字符串就是一个空串。当在目标字符串中找到两个紧接着的正则匹配，而二者之间不存在任何文本的话，那么也会在返回数组中添加一个空串。如果在目标字符串的结尾产生了一个匹配，那么数组中的最后一个元素也是一个空串。

## Java

如果你只需要拆分一个字符串，那么可以直接用目标字符串调用 `Split()` 方法。把正则表达式作为唯一的参数。这个方法会调用 `Pattern.compile("regex").split(subjectString)`。

如果你想要拆分多个字符串，那么就应当使用 `Pattern.compile()` 来创建一个 `Pattern` 对象。这样，你的正则表达式就只需要编译一次，然后就可以使用你的 `Pattern` 实例来调用 `Split()` 方法，把目标字符串作为它的参数。这里没必要再创建一个 `Matcher` 对象。在 `Matcher` 类中根本不支持 `Split()` 方法。

`Pattern.split()` 会接受一个可选的第二个参数，但是 `String.split()` 却不能。你可以使用这第二个参数来传递你最终想要得到的拆分字符串的最大个数。例如，如果你调用 `Pattern.split(subject, 3)`，那么就会得到一个最多包含 3 个字符串的数组。`Split()` 函数会尝试找到 2 个正则匹配，然后返回一个数组，其中包括第一个匹配之前的文本、两个匹配之间的文本和第二个匹配之后的文本。在该目标字符串之后所有可能的正则匹配都会被忽略掉，因此会被全部放到数组中的最后一个字符串里。如果不存在足够的正则匹配来满足你的上限，那么 `Split()` 会按照所有可用的正则匹配来进行拆分，从而它返回的数组中包含的字符串数量因此会比你指定的参数要少。`regexObj.Split(subject, 1)` 不会对字符串进行拆分，而是直接把整个目标字符串作为数组的唯一元素返回。

如果在目标字符串的开始产生了一个匹配，那么在返回数组中的第一个字符串就是一个空串。当在目标字符串中找到两个紧接着的正则匹配，而二者之间不存在任何文本，那么也会在返回数组中添加一个空串。如果在目标字符串的结尾产生了一个匹配，那么数组中的最后一个元素也是一个空串。

然而，Java 会清除位于数组最后的空字符串。如果你想要包含空字符串，那么就需要向 `Pattern.split()` 传递一个负数作为第二个参数。这会告诉 Java 尽可能多地拆分字符串，并且保留位于数组最后的空串。但是，你不能告知 Java 对一个字符串拆分指定的次数，同时还要求在数据结尾保留空串。

## JavaScript

在 JavaScript 中，使用你想要拆分的字符串来调用 `split()` 方法。把正则表达式作为唯一的参数就可以得到一个字符串数组，其中会对目标字符串尽可能多次进行拆分。你还可以传递一个可选的第二个参数来指定你想要在返回数组中包含的最大字符串数量。

这个参数应当是一个正整数。如果你传递的是 0，那么会得到一个空数组。如果没有使用第二个参数，或者传递了一个负数，那么字符串就会被拆分尽量多次。在正则表达式中使用/g 标志（实例 3.4）并不会产生任何作用。

不幸的是，所有流行的 Web 浏览器都没有按照 JavaScript 标准中的描述来实现 split() 方法的每个特性。具体来说，有些浏览器在数组中包含了捕获分组匹配的文本，有些则不然。那些包含了捕获分组的浏览器在处理非参与分组时，也没有进行一致的处理。为了避免这些问题，我们建议你在传递给 split() 的正则表达式中只使用非捕获分组（实例 2.9）。

有些 JavaScript 实现会忽略在返回数组中的长度为 0 的字符串。当两个正则匹配完全紧邻，或者正则匹配到字符串的开始或结尾的时候，那么数组中就必须要包含长度为 0 的字符串。因为无法通过对你的正则表达式进行简单修改来解决这个问题，所以通常来说最好还是选择使用更复杂的 JavaScript 解决方案。这个解答中包含了所有长度为 0 的字符串，但是你可以很容易地对它进行编辑，把长度为 0 的字符串去掉。

这个较长的解决方案是在实例 3.12 之上的修改。它把正则匹配之间的文本以及正则匹配自身都添加到了数组中。为了得到位于匹配之间的文本，我们使用了在实例 3.8 中讲解的匹配详细信息。

如果你想要一个遵循标准的 String.prototype.split 实现，而且希望在所有浏览器中都可以工作，那么可以参考 Steven Levithan 给的解决方案：<http://blog.stevenlevithan.com/archives/cross-browser-split>。

## PHP

调用 preg\_split() 就可以按照正则匹配把字符串拆分成一个字符串数组。使用正则表达式作为第一个参数，目标字符串作为第二个参数。如果你省略了第二个参数，那么\$\_ 就会被用作目标字符串。

你可以传递一个可选的第三个参数来指定最终想要得到的拆分字符串的最大数量。例如，如果你调用 preg\_split(\$regex, \$subject, 3)，那么就会得到一个最多包含 3 个字符串的数组。preg\_split() 函数会尝试找到 2 个正则匹配，然后返回一个数组，其中包括第一个匹配之前的文本、两个匹配之间的文本和第二个匹配之后的文本。在该目标字符串之后的所有可能的正则匹配都会被忽略掉，因此会被全部放到数组中的最后一个字符串里。如果不存在足够的正则匹配来满足你的上限，那么 preg\_split() 会按照所有可用的正则匹配来进行拆分，从而它返回的数组中包含的字符串数量会比你指定的参数要少。如果没有使用第三个参数，或者把它设为 -1，那么字符串就会被拆分尽量多次。

如果在目标字符串的开始产生了一个匹配，那么在返回数组中的第一个字符串就是一个空串。当在目标字符串中找到两个紧接着的正则匹配，而二者之间不存在任何文本，那么也会在返回数组中添加一个空串。如果在目标字符串的结尾产生了一个匹配，那

么数组中的最后一个元素也是一个空串。默认来说，`preg_split()`会在它返回的数组中包含所有的空串。如果你不想要数组中的空串，那么你可以使用常量 `PREG_SPLIT_NO_EMPTY` 来作为第 4 个参数。

## Perl

调用 `split()` 就可以按照正则匹配把字符串拆分成一个字符串数组。使用正则表达式作为第一个参数，目标字符串作为第二个参数。

你还可以传递一个可选的第三个参数来指定最终想要得到的拆分字符串的最大数量。例如，如果调用 `split(/regex/, subject, 3)`，那么你就会得到一个最多包含 3 个字符串的数组。`split()` 函数会尝试找到 2 个正则匹配，然后返回一个数组，其中包括第一个匹配之前的文本、两个匹配之间的文本和第二个匹配之后的文本。在该目标字符串之后的所有可能的正则匹配都会被忽略掉，因此会被全部放到数组中的最后一个字符串里。如果不存在足够的正则匹配来满足你的上限，那么 `split()` 会按照所有可用的正则匹配来进行拆分，从而它返回的数组中包含的字符串数量会因此比你指定的参数要少。

如果你没有使用第三个参数，那么 Perl 会决定最适合的长度限制。如果像这个实例中一样，把结果赋给了一个数组变量，那么字符串就会被拆分尽量多次。如果把结果赋给了一个标量变量的列表，那么 Perl 会把上限设为变量的个数加 1。换句话说，Perl 会尝试填满所有的变量，但是会丢掉没有拆分的剩余部分。例如，`($one, $two, $three) = split(/,/)` 会使用上限 4 来对 `$_` 进行拆分。

如果在目标字符串的开始产生了一个匹配，那么在返回数组中的第一个字符串就是一个空串。当在目标字符串中找到两个紧接着的正则匹配，而二者之间不存在任何文本，那么也会在返回数组中添加一个空串。如果在目标字符串的结尾产生了一个匹配，那么数组中的最后一个元素也是一个空串。

## Python

`re` 模块中的 `split()` 函数会使用正则表达式来拆分字符串。把你的正则表达式作为第一个参数，而目标字符串作为第二个参数。全局的 `split()` 函数不支持包含正则表达式选项的参数。

`re.split()` 会调用 `re.compile()`，然后使用编译好的正则表达式对象来调用 `split()` 方法。这个方法只有一个必需的参数：目标字符串。

`split()` 函数的两种形式都会返回一个位于所有正则匹配之间的文本列表。它们都接受一个可选的参数，用来指定你想要把字符串进行拆分的最大次数。如果你省略了这个参数，或者把它设置为 0，那么字符串就会被拆分尽量多次。如果你传递了一个正整数，那么它就是用来拆分字符串的正则匹配的最大个数。因此返回的结果列表中的字符串个数是你指定的个数加 1。最后一个字符串是在最后一个正则匹配之后没有被拆分的目

标字符串中的所有剩余部分。如果能找到的匹配个数比你指定的个数要少，那么字符串就会按照所有正则匹配进行拆分，而不会产生任何错误。

## Ruby

使用目标字符串来调用 `split()`，并且把正则表达式作为第一个参数，就可以按照正则匹配把字符串拆分成一个字符串数组。

`split()`函数还接受一个可选的第二个参数，可以用它来指定最终想要得到的拆分字符串的最大数目。例如，如果调用 `subject.split(re, 3)`，那么你就会得到一个最多包含 3 个字符串的数组。`split()`函数会尝试找到 2 个正则匹配，然后返回一个数组，其中包括第一个匹配之前的文本、两个匹配之间的文本和第二个匹配之后的文本。在该目标字符串之后所有可能的正则匹配都会被忽略掉，因此会被全部放到数组中的最后一个字符串里。如果不存在足够的正则匹配来满足你的上限，那么 `split()`会按照所有可用的正则匹配来进行拆分，从而它返回的数组中包含的字符串数量会比你指定的个数要少。`split(re, 1)`则不会对字符串进行拆分，只是把原始字符串作为数组的唯一元素返回。

如果在目标字符串的开始产生了一个匹配，那么在返回数组中的第一个字符串就是一个空串。当在目标字符串中找到两个紧接着的正则匹配，而二者之间不存在任何文本，那么也会在返回数组中添加一个空串。如果在目标字符串的结尾产生了一个匹配，那么数组中的最后一个元素也是一个空串。

然而，Ruby 会清除位于数组最后的空字符串。如果你想要包含空字符串，那么就需要向 `split()`传递一个负数作为第二个参数。这会告知 Ruby 尽可能多地拆分字符串，并且保留位于数组最后的空串。但是，你不能告知 Ruby 对一个字符串拆分指定的次数，同时还要求在数据结尾保留空串。

## 参见

实例 3.20。

## 3.20 拆分字符串，保留正则匹配

### 问题描述

你想要使用正则表达式来拆分一个字符串。在拆分之后，你会得到一个字符串的数组或列表，其中会包含位于正则表达式匹配之间的文本、以及正则匹配自身。

假设你想要按照 HTML tag 来拆分一个包含 HTML tag 的字符串，但是还想要保留其中的 HTML tag。例如，对 `I•like•<b>bold</b>•and•<i>italic</i>•fonts` 进行拆分后会得到一个包含 9 个字符串的数组：`I•like•`、`<b>`、`bold`、`</b>`、`•and•`、`<i>`、`italic`、`</i>` 和 `•fonts`。

# 解决方案

## C#

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
string[] splitArray = Regex.Split(subjectString, "<[^<>]*>");
```

如果你想要把同一个正则表达式用于大量的字符串之上，那么就需要构造一个 Regex 对象：

```
Regex regexObj = new Regex("<[^<>]*>");  
string[] splitArray = regexObj.Split(subjectString);
```

## VB.NET

当你打算只使用同一个正则表达式处理少量字符串的时候，可以使用如下的静态调用：

```
Dim SplitArray = Regex.Split(SubjectString, "<[^<>]*>")
```

如果想要把同一个正则表达式用于大量的字符串之上，那么就需要构造一个 Regex 对象：

```
Dim RegexObj As New Regex("<[^<>]*>")  
Dim SplitArray = RegexObj.Split(SubjectString)
```

## Java

```
List<String> resultList = new ArrayList<String>();  
Pattern regex = Pattern.compile("<[^<>]*>");  
Matcher regexMatcher = regex.matcher(subjectString);  
int lastIndex = 0;  
while (regexMatcher.find()) {  
    resultList.add(subjectString.substring(lastIndex,  
                                           regexMatcher.start()));  
    resultList.add(regexMatcher.group());  
    lastIndex = regexMatcher.end();  
}  
resultList.add(subjectString.substring(lastIndex));
```

## JavaScript

```
var list = [];  
var regex = /<[^<>]*>/g;  
var match = null;  
var lastIndex = 0;  
while (match = regex.exec(subject)) {  
    // Don't let browsers such as Firefox get stuck in an infinite loop  
    if (match.index == regex.lastIndex) regex.lastIndex++;  
    list.push(subject.substring(lastIndex, match.index));  
    list.push(match[0]);  
    lastIndex = match.index + match[0].length;
```

```
// Add the text before the match, as well as the match itself
list.push(subject.substring(lastIndex, match.index), match[0]);
lastIndex = match.index + match[0].length;
}
// Add the remainder after the last match
list.push(subject.substr(lastIndex));
```

## PHP

```
$result = preg_split('/(<[^<>]*>)/', $subject, -1,
PREG_SPLIT_DELIM_CAPTURE);
```

## Perl

```
@result = split(m/(<[^<>]*>)/, $subject);
```

## Python

如果只需要处理少量字符串，你可以使用全局函数：

```
result = re.split("<[^<>]*>", subject)
```

如果要反复使用同一个正则表达式，那么就需要使用一个编译好的对象：

```
reobj = re.compile("<[^<>]*>")
result = reobj.split(subject)
```

## Ruby

```
list = []
lastindex = 0;
subject.scan(/<[^<>]*>/) { |match|
  list << subject[lastindex..$~.begin(0)-1];
  list << $&
  lastindex = $~.end(0)
}
list << subject[lastindex..subject.length()]
```

## 讨论

### .NET

在 .NET 中，`Regex.Split()`方法会把捕获分组所匹配到的文本包含到返回数组中。.NET 1.0 和 1.1 只会包含第一个捕获分组。.NET 2.0 和更高版本会把所有捕获分组都当作单独字符串包含在数组中。如果想要把总的正则匹配也加到数组中，那么就需要把整个正则表达式放到一个捕获分组中。对于 .NET 2.0 和更高版本来说，所有其他分组都必须是非捕获的，否则它们也会被添加到数组中。

捕获分组不会被算到你传递给 Split() 函数的字符串计数中。如果使用本实例中的示例字符串和正则式来调用 regexObj.Split(subject, 4)，那么你会得到一个包含 7 个字符串的数组。它们分别是在三个正则匹配之前、之间和之后的 4 个字符串，再加上由正则表达式中的唯一捕获分组捕获到的位于它们之间的三个正则匹配。简单来说，你会得到一个包含如下字符串的数组：I●like●、<b>、bold、</b>、●and●、<italic> 和 italic</italic>●fonts。如果你的正则表达式中包含 10 个捕获分组，而且你使用的是 .NET 2.0 及更高版本，那么 regexObj.Split(subject, 4) 会返回一个包含 34 个字符串的数组。

.NET 没有提供一个选项来把捕获分组从数组中去掉。你唯一的解决方案只能是把所有命名和编号的捕获分组都替换为非捕获分组。在 .NET 中可以使用 RegexOptions.ExplicitCapture 很容易地把正则表达式中的所有命名分组都替换成正常分组（也就是一对圆括号）。

## Java

Java 中的 Pattern.split() 方法没有提供选项可以把正则匹配添加到结果数组中。但是，我们可以通过修改实例 3.12 来把正则匹配与位于正则匹配之间的文本一起添加到一个列表中。为了得到位于正则匹配之间的文本，我们使用了实例 3.8 中讲解的匹配详细信息。

## JavaScript

JavaScript 中的 string.split() 函数没有提供一个选项来控制是否应该把正则匹配添加到数组中。根据 JavaScript 标准，所有的捕获分组都应该把它们的匹配添加到数组中。不幸的是，当前流行的 web 浏览器却都没有这样做，或者做得不够一致。

为了得到一个在所有浏览器上都可以工作的解决方案，可以修改实例 3.12 来把正则匹配与位于正则匹配之间的文本一起添加到一个列表中。为了得到位于正则匹配之间的文本，我们使用了实例 3.8 中讲解的匹配详细信息。

## PHP

把 PREG\_SPLIT\_DELIM\_CAPTURE 作为第 4 个参数传递给 preg\_split()，就会把捕获分组匹配到的文本添加到返回数组中。你还可以使用|操作符把 PREG\_SPLIT\_DELIM\_CAPTURE 和 PREG\_SPLIT\_NO\_EMPTY 组合起来使用。

捕获分组不会被算到你传递给 preg\_split() 函数的字符串计数中。如果你使用本实例中的示例字符串和正则式，而把字符串个数上限设为 4，那么你会得到一个包含 7 个字符串的数组。它们分别是在 3 个正则匹配之前、之间和之后的 4 个字符串，再加上由正则表达式中的唯一捕获分组捕获到的位于它们之间的 3 个正则匹配。简单来说，你会得到一个包含如下字符串的数组：I●like●、<b>、bold、</b>、●and●、<italic> 和 italic</italic>●fonts。

## Perl

Perl 中的 `split()` 函数会把所有捕获分组匹配到的文本都添加到数组中。如果你想要把整个正则匹配也加到数组中，那么需要把整个正则表达式放到一个捕获分组中。

捕获分组不会被算到你传递给 `split()` 函数的字符串计数中。如果使用本实例中的示例字符串和正则式调用 `split(/(<[^>]*>)/, $subject, 4)`，那么你会得到一个包含 7 个字符串的数组。它们分别是在 3 个正则匹配之前、之间和之后的 4 个字符串，再加上由正则表达式中的唯一捕获分组捕获到的位于它们之间的 3 个正则匹配。简单来说，你会得到一个包含如下字符串的数组：`I•like•`、`<b>`、`bold`、`</b>`、`•and•`、`<italic>` 和 `italic</italic>•fonts`。如果你的正则表达式中包含 10 个捕获分组，那么 `split($regex, $subject, 4)` 会返回一个包含 34 个字符串的数组。

Perl 没有提供一个选项来把捕获分组从数组中去掉。唯一的解决方案只能是把所有命名和编号的捕获分组都替换为非捕获分组。

## Python

Python 中的 `split()` 函数会把所有捕获分组匹配到的文本都添加到数组中。如果想要把整个正则匹配也加到数组中，那么就需要把整个正则表达式放到一个捕获分组中。

捕获分组不会被算到你传递给 `split()` 函数的字符串计数中。如果使用本实例中的示例字符串和正则式来调用 `split(/(<[^>]*>)/, $subject, 3)`，那么你会得到一个包含 7 个字符串的数组。它们分别是在三个正则匹配之前、之间和之后的 4 个字符串，再加上由正则表达式中的唯一捕获分组捕获到的位于它们之间的三个正则匹配。简单来说，你会得到一个包含如下字符串的数组：`I•like•`、`<b>`、`bold`、`</b>`、`•and•`、`<italic>` 和 `italic</italic>•fonts`。如果你的正则表达式中包含 10 个捕获分组，那么 `split($regex, $subject, 3)` 会返回一个包含 34 个字符串的数组。

Python 没有提供一个选项来把捕获分组从数组中去掉。唯一的解决方案只能是把所有命名和编号的捕获分组都替换为非捕获分组。

## Ruby

Ruby 中的 `String.split()` 方法没有提供选项可以把正则匹配添加到结果数组中。但是，我们可以通过修改实例 3.12 来把正则匹配与位于正则匹配之间的文本一起添加到一个列表中。为了得到位于正则匹配之间的文本，我们使用了实例 3.8 中讲解的匹配详细信息。

## 参见

实例 2.9 中讲解了捕获和非捕获分组。

实例 2.11 中讲解了命名分组。

## 3.21 逐行查找

### 问题描述

传统的 grep 工具在使用正则表达式时每次只会应用到目标文本的一行之上，并且会显示正则表达式匹配到（或者没有匹配到）的行。这个问题要求你采用这种方式来处理一个字符串数组或一个字符串。

### 解决方案

#### C#

如果你有一个多行字符串，那么首先把它拆分成一个字符串数组，该数组中的每个字符串都包含一行文本：

```
string[] lines = Regex.Split(subjectString, "\r?\n");
```

然后，对这个 lines 数组进行遍历：

```
Regex regexObj = new Regex("regex pattern");
for (int i = 0; i < lines.Length; i++) {
    if (regexObj.IsMatch(lines[i])) {
        // The regex matches lines[i]
    } else {
        // The regex does not match lines[i]
    }
}
```

#### VB.NET

如果你有一个多行字符串，那么首先把它拆分成一个字符串数组，该数组中的每个字符串都包含一行文本：

```
Dim Lines = Regex.Split(SubjectString, "\r?\n")
```

然后，对这个 lines 数组进行遍历：

```
Dim RegexObj As New Regex("regex pattern")
For i As Integer = 0 To Lines.Length - 1
    If RegexObj.IsMatch(Lines(i)) Then
        'The regex matches Lines(i)
    Else
        'The regex does not match Lines(i)
    End If
Next
```

## Java

如果你有一个多行字符串，那么首先把它拆分成一个字符串数组，该数组中的每个字符串都包含一行文本：

```
String[] lines = subjectString.split("\r?\n");
```

然后，对这个 lines 数组进行遍历：

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher("");
for (int i = 0; i < lines.length; i++) {
    regexMatcher.reset(lines[i]);
    if (regexMatcher.find()) {
        // The regex matches lines[i]
    } else {
        // The regex does not match lines[i]
    }
}
```

## JavaScript

如果你有一个多行字符串，那么首先把它拆分成一个字符串数组，该数组中的每个字符串都包含一行文本。在实例 3.19 中提到过，有些浏览器会把数组中的空行去掉。

```
var lines = subject.split(/\r?\n/);
```

然后，对这个 lines 数组进行遍历：

```
var regexp = /regex pattern/;
for (var i = 0; i < lines.length; i++) {
    if (lines[i].match(regexp)) {
        // The regex matches lines[i]
    } else {
        // The regex does not match lines[i]
    }
}
```

## PHP

如果你有一个多行字符串，那么首先把它拆分成一个字符串数组，该数组中的每个字符串都包含一行文本：

```
$lines = preg_split('/\r?\n/', $subject)
```

然后，对这个 \$lines 数组进行遍历：

```
foreach ($lines as $line) {
    if (preg_match('/regex pattern/', $line)) {
        // The regex matches $line
```

```
    } else {
        // The regex does not match $line
    }
}
```

## Perl

如果你有一个多行字符串，那么首先把它拆分成一个字符串数组，该数组中的每个字符串都包含一行文本：

```
$lines = split(m/\r?\n/, $subject)
```

然后，对这个\$lines 数组进行遍历：

```
foreach $line ($lines) {
    if ($line =~ m/regex pattern/) {
        # The regex matches $line
    } else {
        # The regex does not match $line
    }
}
```

## Python

如果你有一个多行字符串，那么首先把它拆分成一个字符串数组，该数组中的每个字符串都包含一行文本：

```
, lines = re.split("\r?\n", subject);
```

然后，对这个 lines 数组进行遍历：

```
reobj = re.compile("regex pattern")
for line in lines[:]:
    if re.search(line):
        # The regex matches line
    else:
        # The regex does not match line
```

## Ruby

如果你有一个多行字符串，那么首先把它拆分成一个字符串数组，该数组中的每个字符串都包含一行文本：

```
lines = subject.split(/\r?\n/)
```

然后，对这个 lines 数组进行遍历：

```
re = /regex pattern/
lines.each { |line|
    if line =~ re
        # The regex matches line
```

```
    else
        # The regex does not match line
}
```

## 讨论

在处理基于行的数据时，如果你不是试图去处理包含内嵌换行符的一个长字符串，而是选择先把数据拆分成包含每一行字符串的数组，那么就可以省去很多麻烦。然后，你可以把实际的正则表达式应用到数组中的每个字符串之上，而不必担心它是否会匹配多于一行的内容。这种方法同样使得跟踪行之间的关系较为容易。例如，你可以很容易地使用一个正则式遍历数组来查找题头行，然后使用另外一个正则表达式来查找脚注行。在找到了分界的两行之后，你就可以使用第三个正则表达式来查找你感兴趣的数据行。虽然这样看起来要做很多工作，但其实却并不复杂，而且会得到执行效率很高的代码。如果你试图要构造一个正则式来一次找到题头、数据和脚注，那么就会复杂很多，而且会得到一个速度很慢的正则表达式。

逐行处理字符串还使得我们可以比较容易对正则表达式进行取反。正则表达式并没有提供简单的方式来解决“匹配不包含这个或那个单词的一行内容”。只有字符类才能很容易地被取反。但是如果你已经把字符串拆分成了行，那么找到不包含某个单词的行就变得非常容易，你只需在所有行上进行字面文本查找，然后去掉那些没有找到该单词的行即可。

实例 3.19 中讲解了如何把一个字符串拆分为数组。正则表达式 `\r\n` 会匹配一对 **CR** 和 **LF** 字符，它们是在 Microsoft Windows 平台上使用的换行符号。`\n` 会匹配一个 **LF** 字符，它是在 Unix、Linux 和 OS X 等平台上使用的换行符。因为这两个正则表达式其实就是简单的文本，所以在这里甚至都不需要使用正则表达式。如果你的编程语言支持使用字面文本来拆分字符串，那么肯定应该选择这种方式。

如果不确定你的数据中使用的换行风格，那么你可以使用正则表达式 `\r?\n` 来对它进行拆分。通过把 **CR** 变成可选的，这个正则式就可以同时匹配一个 **CRLF** Windows 换行，或者一个 **LF** Unix 换行。

一旦把字符串放到了数组中，你就可以很容易地对它进行循环访问。在循环中，按照实例 3.5 中给出的解决方案来检查哪些行会匹配，而哪些行不会匹配。

## 参见

实例 3.11 和 3.19。

# 合法性验证和格式化

本章的实例会讲解如何对常见类型的用户输入进行合法性验证和格式化。其中有些解决方案会向读者展示如何支持合法输入的不同变种，例如，美国邮政编码可以包含 5 位或者 9 位数字。其他一些实例中的正则表达式则会对诸如电话号码、日期和信用卡号码等数据对应的常见格式进行矫正或整理。

除了可以帮助你完成排除不合法输入的任务之外，这些实例还会帮助你改进应用程序的用户体验。如果经常在电话或信用卡号码输入框旁边显示诸如“缺少空格或连字符”的消息，就会让用户感到很沮丧，有时还会被用户直接忽略。幸运的是，在许多情况下，你只需要做很少量的工作，正则表达式就可以允许用户按照他们认为熟悉和方便的格式输入数据。

某些编程语言会在它们的内置类或函数库中提供与本章中某些实例类似的功能。根据你的具体需要，也可能使用这些内置的方式会更好一些，因此我们也会在适当的时候指出你可能拥有的选择。

## 4.1 E-mail 地址的合法性验证

### 问题描述

在你的网站或者应用程序的对话框中，有一个输入框要求用户输入一个电子邮件（E-mail）地址。在使用这个地址发送 E-mail 之前，你要使用一个正则表达式来对它进行合法性验证，这会有助于减少你可能会收到的由于无法投递而返回的邮件数量。

### 解决方案

#### 简单形式

这第一个解决方案只进行非常简单的检查。它只会验证 E-mail 地址中包含了单个的@

符号，并且不包含任何空白：

```
^\$+@\$+$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python  
  
\A\$+@\$+\Z  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

## 对字符加限制的简单形式

在@符号之后的域名部分被限制为只能使用域名中允许的字符。在@符号之前的用户名部分被限制为只能使用在 E-mail 用户名中常见的字符，这比大多数邮件客户端和服务器所接受的标准要更为严格：

```
^[A-Z0-9+_-.]+@[A-Z0-9._-]+$  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python  
  
\A[A-Z0-9+_-.]+@[A-Z0-9._-]+\Z  
正则选项：不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

## 包括所有字符的简单形式

这个正则表达式对上一个进行了扩展，它允许在用户名中使用更大范围的较少使用的字符。不是所有的 E-mail 软件都能够处理所有这些字符，但是我们在这里包括了规定 E-mail 消息格式的 RFC 2822 标准中允许的所有字符。在这些允许的字符中，如果直接把它们从用户输入传递给一个 SQL 语句，其中有些字符可能会带来潜在的安全风险，比如单引号 (') 和管道符号 (|)。当你把 E-mail 地址插入到传递给另外一个程序的字符串中之前，一定要记住对这些敏感字符进行转义，这样就可以避免诸如 SQL 注入攻击这样的安全漏洞。

```
^[\w!#$%&'*+/=?'{|}~^._-]+@[A-Z0-9._-]+$  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python  
  
\A[\w!#$%&'*+/=?'{|}~^._-]+@[A-Z0-9._-]+\Z  
正则选项：不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

## 不允许前导、拖尾或连续的点号

在用户名和域名中都可以包含一个或多个点号，但是不允许出现两个连续的点号。另外，在用户名和域名中的第一个和最后一个字符都不允许是点号：

```
^[\w!#$%&'*+/=?'{|}~^.-]+(?:\.[!#$%&'*+/=?'{|}~^.-]+)*@[  
[A-Z0-9-]+(?:\.[A-Z0-9-]+)*$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

```
\A[\w!#$%&'*+/=?'{|}~^-[)+(?:\.[!#$%&'*+/=?'{|}~^-[+])*@\n[A-Z0-9-]+(?:\.[A-Z0-9-]+)*\Z
```

正则选项：不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

## 顶级域名必须包含 2~6 个字母

在前一个版本的正则表达式上添加了新的要求：域名必须至少包含一个点号，而且在最右边的点号之后的域名只能包含字母。也就是说，域名必须至少包含两级，例如 secondlevel.com 或 thirdlevel.secondlevel.com。而顶级域名，比如.com，则必须由 2~6 个字符组成。所有国家代码的顶级域名都包含 2 个字母。而普通顶级域名则会包含 3 (.com) ~6 个字母 (.museum)：

```
^[\w!#$%&'*+/=?'{|}~^-[)+(?:\.[!#$%&'*+/=?'{|}~^-[+])*@\n(?:[A-Z0-9-]+\.)+[A-Z]{2,6}$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

```
\A[\w!#$%&'*+/=?'{|}~^-[)+(?:\.[!#$%&'*+/=?'{|}~^-[+])*@\n(?:[A-Z0-9-]+\.)+[A-Z]{2,6}\Z
```

正则选项：不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

## 讨论

### 关于 E-mail 地址

如果你认为像验证 E-mail 地址这样概念上很简单的任务，可以使用一个简单普适的正则表达式解决方案，那就大错特错了。这个实例告诉我们，在你开始写正则表达式之前，你必须先严格地确定到底想要匹配什么。至于说哪些 E-mail 地址合法而哪些不合法，并没有普遍认可的规则。这要取决于你如何来定义合法性。

根据定义 E-mail 地址语法的 RFC 2822 标准，`asdf@asdf.asdf` 是合法的。但是如果在你的定义中要求一个合法 E-mail 地址必须是可以接收邮件的地址，那么上述地址就是不合法的。因为并不存在 `asdf` 这样的顶级域名。

对合法性问题的简单回答是：如果你实际上不去尝试给 `john.doe@somewhere.com` 发送邮件，就不可能确切知道它是不是一个可以接收邮件的地址。事实上，就算你发了邮件，如果没有收到回复，你也不知道是 `somewhere.com` 域名把发送给不存在的邮箱的邮件悄悄丢掉了，还是 John Doe 自己删除了你的邮件，抑或是他的垃圾邮件过滤器先他一步把你的邮件删掉了。

因为你最终反正都需要实际发一封邮件才能检查该地址是否真的存在，因此可以选择使用一个稍微简单一些、不是那么严格的正则表达式来检查它。允许出现一些不合法的地址，可能会比不接受合法的地址让人觉得更容易接受。基于这个原因，你可能会倾向于选择前面“包含所有字符的简单形式”的正则表达式。虽然它显然会接受很多不是 E-mail 地址的输入，比如说 `#$%@_-`，但是这个正则式速度快而简单，并且它拥有也不可能拒绝任何合法的 E-mail 地址。

如果你想要避免发送太多无法投递的邮件，而依然不想拒绝任何真实 E-mail 地址，那么你应当选择在“顶级域名必须包含 2~6 个字母”中的正则表达式。

你还必须要考虑允许正则表达式复杂到什么程度。如果在验证用户输入，那么你可能想要一个更复杂的正则式，因为用户可能会输入任何东西。但是如果是在扫描一个数据库文件，而你知道其中只会包含合法邮件地址的话，那么就可以使用一个非常简单的正则式，只需把 E-mail 地址同其他数据区分开来即可。即使在前面的“简单形式”小节中的解决方案也足以应付这种情况。

最后，你还必须要考虑是否期待你的正则表达式满足未来可能发生的变化。在过去，一种合理的选择是把顶级域名限制为只能使用 2 个字母的国家代码组合，并且列出所有可能的普通顶级域名，也就是使用 `<com|net|org|mil|edu>`。由于总是会有新的顶级域名不断加入，所以这样的正则表达式很快就会过时。

## 正则表达式的语法

这个实例所给出的正则表达式中，使用了正则表达式语法的所有基本组成部分。如果阅读了第 2 章中的相关部分，那么你已经可以完成正则表达式最适合解决的问题的 90%。

本实例中的所有正则表达式都要求打开不区分大小写的选项。否则，在邮件地址中就会只允许出现大写字符。在把这个选项打开之后，你就可以输入 `<[A-Z]>`，而不必再用 `<[A-Za-z]>`，这样可以节省一些按键时间。如果你使用最后两个正则表达式之一，那么使用不区分大小写的选项就会很方便。否则，你就必须把每个字母 `<X>` 都替换为 `<[Xx]>`。

`\S` 和 `\w` 是简写的字符类，它们在实例 2.3 中进行了讲解。`\S` 会匹配任意非空白字符，而`\w`会匹配一个单词字符。

`<@>` 和 `<.>` 会分别匹配一个字面的@符号和一个点号。由于点号在字符类之外使用时是一个元字符，所以需要对它使用反斜杠来转义。在本书中介绍的所有正则表达式流派中，@符号都不会拥有任何特殊含义。实例 2.1 中给出了所有需要转义的元字符的一个列表。

`<[A-Z0-9.-]>` 和位于方括号之间的其他序列是字符类。这个字符类允许出现在 A 和 Z 之间的所有字母、0 和 9 之间的所有数字，以及一个字面的点号和连字符。尽管连字符通

常会在字符类中创建一个范围，但是当它出现在字符类中的最后一个字符时，还是会当做字面量。

实例 2.3 中讲解了关于字符类的内容，包括如何使用简写把它们组合起来，例如 `\w!#$%&*+/?{|}~^.-]`。这个类会匹配一个单词字符，或者是其中列出的 19 个标点字符之一。

当在字符类之外使用的时候，`(+)` 和 `(*)` 是量词。加号会重复它前面的正则记号一次或多次，而星号则会重复 0 次或多次。在这些正则表达式中，被量化的记号通常是一个字符类，有时候也会是一个分组。因此，`[A-Z0-9.-]+` 会匹配一个或多个字符、数字、点号和/或连字符。

再举一个使用分组的例子，`(?:[A-Z0-9-]+\.)+` 会匹配一个或多个字符、数字和/或连字符，后面紧跟着一个字面点号。加号会重复这个分组一次或多次。因此该分组会被至少匹配一次，但是可以匹配任意多次。实例 2.12 中详细解释了类似这样结构的工作原理。

`(?:group)` 是一个非捕获分组。可以使用它来创建一个包含正则表达式中一部分的分组，这样你就可以在整个分组之上应用一个量词。捕获分组 `(group)` 的功能完全一样，而且语法上更为简洁，因此你可以把我们到目前为止用到的所有正则表达式中的 `(?:)` 都替换为 `( )`，并不会改变总的匹配结果。

但是，因为我们对于单独捕获 E-mail 地址的组成部分并不感兴趣，所以使用非捕获分组的效率会稍微高一些，虽然它会使正则表达式读起来有些困难。实例 2.9 中讲解了捕获和非捕获分组的知识。

定位符 `(^)` 和 `(\$)` 会分别要求正则表达式在目标字符串的开头和结尾寻找匹配。把整个字符串放到这两个字符之间，实际上就会要求正则表达式匹配整个目标字符串。

使用定位符在验证用户输入的时候是很重要的。你并不想接受 `drop database; --joe@server.com haha!` 这样的字符串为合法的 E-mail 地址。如果没有使用定位符的话，那么所有前面所给的正则表达式都会产生匹配，因为它们都会在给定文本的中间找到 `joe@server.com`。更多细节请参考实例 2.5，该实例中还解释了为什么必须关掉“脱字符和美元符号匹配换行处”的选项。

在 Ruby 中，脱字符和美元符号总是会匹配换行处。使用了脱字符和美元符号的正则表达式只有在你要验证的字符串中不包含换行的时候，才能在 Ruby 中正常工作。如果字符串中可能包含换行，那么所有使用 `(^)` 和 `(\$)` 的正则式都会匹配 `drop database; --LFjoe@server.comLFhaha!`，其中 `LF` 代表的是换行符。

为了避免这种情况的出现，就应当转而使用 `\A` 和 `\Z`。除了 JavaScript 之外，在本书讨论的所有流派中，这两个定位符不管在任何选项之下，都只会匹配字符串的起始和结束。JavaScript 根本不支持 `\A` 和 `\Z`。实例 2.5 会讲解这些定位符。



### 提示

`^` 和 `$` 与 `\A` 和 `\Z` 的问题在所有用来验证输入的正则表达式之上都存在。在本书中这样的情况会出现多次。虽然我们会偶尔提供一些提醒，但并不会总是重复这个建议，或者在每个实例中都为 JavaScript 和 Ruby 给一份单独的解决方案。在许多情况下，我们只会给出一个使用脱字符和美元符号的解决方案，而把 Ruby 列为一种兼容流派。如果你使用的是 Ruby，而且想要避免在一个多行字符串中匹配到一行，那么就要记住使用 `\A` 和 `\Z`。

## 构造正则表达式的步骤

下面讲解如何逐步构造一个正则表达式。这里讲的技巧尤为适合手边有一个正则表达式测试工具（比如 RegexBuddy）的读者。

首先，在工具中加载一大堆合法和不合法的样例数据。对这个实例来说，也就是需要一个合法 E-mail 地址的列表，和一个不合法的 E-mail 地址的列表。

然后，写一个正则表达式来匹配所有合法的 E-mail 地址。现在先暂时不考虑不合法的 E-mail 地址。`^\S+@\S+$` 就可以定义一个 E-mail 地址的基本结构：一个用户名、一个@符号和一个域名。

在定义了文本模式的基本结构之后，你就可以开始精化每个组成部分，直到正则表达式不再匹配任何不合法的数据为止。如果你的正则表达式只需要能够用于已有数据，那么这样就可以了。如果你的正则表达式还要能够用于任意用户输入，就需要继续对该正则表达式进行编辑，直到把它变得足够精确，这样做会比只是让它匹配合法数据要难很多。

## 变体

如果你想要在很多数据中查找 E-mail 地址，而不是检查一个输入是否是 E-mail 地址，那么就不能再使用定位符 `^` 和 `$`。只是把这两个定位符从正则表达式中移走，得到的也不是正确答案。如果你对最后一个把顶级域名限制为字母的正则式这样做，那么它会匹配到字符串 `asdf@asdf.as99` 中的 `asdf@asdf.as`。因此，我们选择的做法不是把正则匹配定位到目标字符串的开始和结尾，而是必须指定用户名的开头和顶级域名的结尾不能属于某个更长单词的一部分。

这可以很容易地使用一对单词分界符来完成。把 `^` 和 `$` 都替换为 `\b`。例如，`^[A-Z0-9+_-.]+@[?:[A-Z0-9-]+\.]+[A-Z]{2,6}$` 会被改变为 `\b[A-Z0-9+_-.]+@[?:[A-Z0-9-]+\.]+[A-Z]{2,6}\b`。

这个正则表达式实际上是组合了“对字符加以限制的简单形式”中的用户名部分，以及“顶级域名必须包含 2~6 个字母”中的域名部分。在实际应用中，我们发现这个正

则表达式是比较理想的。

## 参见

RFC 2822 中定义了 E-mail 消息的结构和语法，其中也就包括了在邮件消息中使用的 E-mail 地址。你可以从 <http://www.ietf.org/rfc/rfc2822.txt> 下载 RFC 2822。

## 4.2 北美电话号码的合法性验证和格式化

### 问题描述

你想要确定用户是否输入了一个常见格式的北美电话号码，其中包括区号。这些格式可以包括 1234567890、123-456-7890、123.456.7890、123 456 7890、(123) 456 7890 和所有类似的组合形式。如果电话号码是合法的，那么你还需要把它转换为一种标准格式，(123) 456-7890，这样可以使它们在电话号码记录中保持一致。

### 解决方案

一个正则表达式可以很容易地检查用户输入的数据看起来是否像一个合法的电话号码。通过使用捕获分组来记住每组数字，就可以使用同一个正则表达式，把目标文本替换成你想要的任何格式。

### 正则表达式

```
^\((?([0-9]{3})\))?[ -●]?([0-9]{3})[ -●]?([0-9]{4})$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### 替代文本

```
($1)●$2-$3  
替代文本流派: .NET、Java、JavaScript、Perl、PHP  
(\1)●\2-\3  
替代文本流派: Python、Ruby
```

### C#

```
Regex regexObj =  
    new Regex(@"^(\(?([0-9]{3})\)?[ - ]?([0-9]{3})[ - ]?([0-9]{4}))$");  
  
if (regexObj.IsMatch(subjectString)) {  
    string formattedPhoneNumber =  
        regexObj.Replace(subjectString, "($1) $2-$3");  
} else {  
    // Invalid phone number  
}
```

## JavaScript

```
var regexObj = /^(\d{3})\)?[-. ]?(\d{3})[-. ]?(\d{4})$/;
if (regexObj.test(subjectString)) {
    var formattedPhoneNumber =
        subjectString.replace(regexObj, "($1) $2-$3");
} else {
    // Invalid phone number
}
```

## 其他编程语言

如果读者希望了解如何在其他编程语言中实现这个正则表达式，请参考实例 3.5 和 3.15 中的讲解。

## 讨论

这个正则表达式会匹配 3 组数字。第一组可以选择使用圆括号括起来，而前两组之后则可以选择使用 3 种分隔符（连字符、点号或空格）之一。下面对该正则表达式进行了分解，其中略去了冗余的数字分组：

```
^          # 判断字符串的起始位置
\(\        # 匹配一个字面的 "("...
?          # 0 次或者 1 次
(          # 把括号中的匹配捕获到向后引用#1 中...
  \d{3}    # 匹配一个数字...
  {3}      # 正好三次
)          # 捕获分组#1 结束
\)\        # 匹配一个字母的 ")"...
?          # 0 次或者 1 次
[-. ]     # 匹配 "-." 中的一个字符...
?          # 0 次或者 1 次
...        # [匹配其余的数字和分隔符]
$          # 判断字符串到结束位置
```

我们仔细看一下上面的每一个组成部分。

位于正则表达式开头和结尾的 `\^` 和 `\$` 是一种特殊的元字符，被称为定位符 (anchor) 或断言 (assertion)。断言并不会匹配任何文本，它们匹配的是文本中的位置。具体来说，`\^` 会匹配文本的开始，而 `\$` 匹配文本的结束。这样就可以保证正则表达式匹配到的电话号码不会属于某个更长字符串的一部分，比如说 123-456-78901。

我们已经反复看到过，圆括号在正则表达式中是一个特殊字符，但是在这个例子中，我们想要让用户可以输入圆括号，并要求我们的正则表达式可以识别它们。这是一个需要使用反斜杠来对特殊字符进行转义的范例，从而正则表达式可以把它当做一个字面输入。因此，包含第一组数字的序列 `\(\` 和 `\)\` 会匹配字面的圆括号字符。二者之后

都跟着一个问号，这样就把它们变为可有可无的。我们会在解释这个正则表达式中的其他记号类型时讲解关于问号的更多内容。

用不带反斜杠的圆括号括起来的内容是一个捕获分组，可以用来记住在括号之间匹配到的值，从而可以在以后再用到这些匹配文本。在这个例子中，到匹配文本的向后引用被用在了替代文本中，从而我们可以很容易地对电话号码按需重新格式化。

在这个正则表达式中的两种其他记号类型是字符类和量词。字符类允许你匹配一个字符集合中的任意一个。`<[0-9]>`是一个匹配任意数字的字符类。本书中涉及的正则表达式流派都支持简写的字符类`\d`，它也可以用来匹配一个数字，但是在有些流派中，`\d`会匹配来自任何语言的字符集或字母表的数字，而这并不是我们在这里想要的。关于`\d`的更多信息，请参考实例 2.3。

`<[-•]>`也是一个字符类，它可以匹配 3 个分隔符中的任意一个。在这里让连字符在字符类中出现在第一个是很重要的，因为如果出现在其他字符之间的话，它就会创建一个范围，就像在`<[0-9]>`中一样。确保在字符类中的连字符会匹配一个字面连字符的方式是使用反斜杠来对它转义。因此，`<\.-•>`也会产生相同的效果。

最后，量词会允许你重复一个记号或分组。`<{3}>`是一个量词，会让位于它之前的元素恰好重复 3 次。因此，正则表达式`<[0-9]{3}>`与`<[0-9][0-9][0-9]>`是等价的，但是却更加简洁易读。问号（前面已经提到过）是一个特殊的量词，会让位于它之前的元素重复 0 次或 1 次。它也可以写成`<{0,1}>`。允许只重复 0 次的任何量词也就会在实际上把这个元素变成可选的。因为在每个分隔符之后都使用了问号，所以我们也允许电话号码数字都连续出现在一起。

注意，虽然这个实例号称能够处理北美的电话号码，但它实际上只可用于北美电话编号方案（NANP）的号码。NANP 是使用国家代码“1”的国家共同支持的电话编号方案。这些国家中包括美国及其托管领土、加拿大、百慕大和 16 个加勒比国家，其中并不包括墨西哥和中美洲国家。

## 变体

### 排除不合法的电话号码

目前，这个正则表达式可以匹配任何 10 位数字。如果你想要把匹配限制为符合北美电话编号方案的合法电话号码，那么需要遵守如下的基本规则：

- 区号（area code）以 2~9 开头，第二位是 0~8，然后第三位可以是任意数字。
- 第二组中的 3 位数字，也被称为是中心局（central office）或交换机号（exchange code），以 2~9 开头，后面两位可以是任意数字。
- 最后 4 位数字，也被称为是基站号（station code），可以不加限制地使用任何数字。

这些规则可以很容易地使用几个字符类来实现。

```
^\((?([2-9][0-8][0-9])\)?[-.●]?([2-9][0-9]{2})[-.●]?([0-9]{4}))$
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

除了上面列出的基本规则之外，还存在各种被保留的、没有分配的和受限制的电话号码。除非有具体的需要，要求你过滤掉尽可能多的电话号码，一般没有必要费更多的力气去排除未使用的号码。符合上述规则的新区号也会被定期添加进来，而且即使一个电话号码是合法的，也并不一定就意味着它被分配了或者是依然在使用中。

## 寻找文档中的电话号码

如果要在更长的文本中匹配电话号码，就需要在前面的正则表达式上做两个简单的改动：

```
\(?(\b([0-9]{3})\)?[-.●]?([0-9]{3})[-.●]?([0-9]{4}))\b
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

这里去掉了用来把正则表达式绑定到文本开头和结尾的 `<^>` 和 `<$>` 两个断言。在它们的位置，添加了单词边界标记 (`\b`)，用来保证匹配到的文本是独立存在的，而不会属于更长的数字或单词的一部分。

与 `<^>` 和 `<$>` 类似，`\b` 也是一个断言，用来匹配一个位置，而不是匹配任何实际的文本。`\b` 会匹配在一个单词字符和一个非单词字符或者是文本的开始和结束之间的位置。字母、数字和下划线都被认为是单词字符（参见实例 2.6）。

注意第一个单词边界记号会出现在可选的左圆括号之后。这一点很重要，因为在两个非单词字符之间并没有单词边界可以匹配，比如在左圆括号和前导的空白字符之间。第一个单词边界只有在匹配一个不含括号的数字的时候才有意义，因为单词边界总是会匹配左圆括号和电话号码的第一个数字之间的位置。

## 允许前导“1”

你可以允许一个可选的、前导“1”作为国家代码（覆盖了北美电话编号方案地区），修改之后的正则表达式如下所示：

```
^(?:\+?1[-.●]?)?\(?([0-9]{3})\)?[-.●]?([0-9]{3})[-.●]?([0-9]{4}))$
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

除了前面给出的电话号码格式之外，这个正则表达式还会匹配像 +1 (123) 456-7890 和 1-123-456-7890 这样的字符串。其中使用了一个非捕获分组，被记为 `(?:...)`。当在问号后面跟一个没有转义的左圆括号时，问号就不再是一个量词，而是用来识别分组类型。标准的捕获分组要求正则表达式引擎记住向后引用，所以如果分组匹配的文本并不需要在后面引用，那么使用非捕获分组就会效率更高。在这里使用非捕获分组的

另外一个原因是，允许你继续使用在上一个例子中的同一个替代字符串。如果添加的是一个捕获分组，我们就必须在本实例中前面所给的替代文本中把 \$1 修改为 \$2（以此类推）。

在这个版本中添加的全部内容是：`<(?:(\+?1[-.●]?))>`。在这个模式中的“1”之前还有一个可选的前导加号，后面还可以选择使用 3 个分隔符（连字符、点号或空格）之一。添加的整个非捕获分组也是可选的，但是因为在该分组中“1”是必须的，所以如果不存在一个前导“1”，那么之前的加号和后面的分隔符都不允许出现。

## 支持七位电话号码

为了可以匹配略去本地区号的电话号码，需要把第一个数字分组和包含它的圆括号，以及之后的分隔符都添加到一个可选的非捕获分组中：

```
^(?:(?(\([0-9]{3}\))\)?[-.●]?)([0-9]{3})[-.●]?([0-9]{4}))$
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

因为在匹配中不再需要区号，所以如果还是用`«(\$1)●\$2-\$3»`来替换匹配，那么就可能会得到像`() 123-4567`这样的结果，其中会包含一对空的圆括号。要想解决这个问题，可以在正则表达式之外添加代码来检查第一个分组是否匹配到任何文本，从而可以相应地调整替代文本的形式。

## 参见

实例 4.3 中会讲解如何对国际电话号码进行合法性验证。

北美电话编号方案（NANP）是用于美国及其托管领土、加拿大、百慕大和 16 个加勒比国家的电话编号方案。更多信息可以参考 <http://www.nanpa.com>。

## 4.3 国际电话号码的合法性验证

### 问题描述

你想要对国际电话号码进行合法性验证。号码应当以加号开头，然后是国家代码和国内号码。

### 解决方案

#### 正则表达式

```
^(\+\d{6,14}\d)$
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## JavaScript

```
function validate (phone) {
    var regex = /^+\(?:[0-9]\ ?\){6,14}[0-9]$/;

    if (regex.test(phone)) {
        // Valid international phone number
    } else {
        // Invalid international phone number
    }
}
```

## 其他编程语言

如果读者希望了解如何在其他编程语言中实现这个正则表达式，请参考实例 3.5 中的讲解。

## 讨论

国际电话号码使用的规则和约定在全球不同地区有显著的不同，因此，除非你采用一种严格的格式，否则就很难对国际电话号码提供有意义的合法性验证。幸运的是，ITU-T E.123 规定了一种简单的、符合产业标准的格式。这种格式要求国际电话号码包含一个前导的加号（被称为国际前缀符号），然后只允许使用空格来分隔数字分组。虽然根据标准，否定字符（tilde, ~）可以出现在电话号码中，用以说明存在一个额外的拨号音（dial tone），但是在上述正则表达式中我们并没有考虑它，因为它只是一个过程性的元素（换句话说，它并不会出现在实际拨号中），而且并不常用。根据国际电话编码方案（ITU-T E.164），电话号码中不能包含超过 15 个数字。目前使用中的最短的国际电话号码中只包含 7 个数字。

当理解了所有这些之后，我们再把这个正则表达式拆开来看看。因为这个版本使用的是宽松排列模式，所以字面的空格被替换为了`\x20`：

```
^          # 判断字符串的开始位置
\+          # 匹配一个字母的 "+" 字符
(?:        # 分组但是不捕获...
    [0-9]    # 匹配一个数字
    \x20     # 匹配一个空格字符...
    ?        # 0 次或 1 次
)          # 非捕获分组的结束
{6,14}     # 重复之前的分组 6~14 次
[0-9]      # 匹配一个数字
$          # 判断字符串的结尾位置
正则选项: 宽松排列
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

位于正则表达式两边的定位符`<^>`和`<$>`会保证它能匹配整个目标字符串。非捕获分组——

也就是 `<(?::>)` 之间的内容——会匹配单个数字，然后是一个可选的空白字符。使用区间量词 `<{6,14}>` 来重复这个分组会执行关于最小和最大数字个数的规则，同时允许在数字中的任何地方出现空格分隔符。字符类 `<[0-9]>` 的第二个实例完成了关于数字个数的规则（把字符总数从 6~14 增加到 7~15），并保证电话号码不会以空格结束。

## 变体

### 检查 EPP 格式的国际电话号码

```
^\+[0-9]{1,3}\.[0-9]{4,14}(?:x.+)?$
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

这个正则表达式遵守的是由 Extensible Provisioning Protocol (EPP) 规定的国际电话号码格式。EPP 是一个相对较新的协议 (2004 年才正式确定)，它的设计目的是为了域名注册中心和注册人员之间进行通信。它已经被用于越来越多的域名注册中心，其中包括 `.com`、`.info`、`.net`、`.org` 和 `.us`。它之所以重要，是因为 EPP 风格的国际电话号码被越来越多的人使用和认可，因此它也就提供了用来存储 (和验证) 国际电话号码的一种较好的替代格式。

EPP 风格的电话号码使用的格式是 `+CCC.NNNNNNNNNNxEEEE`，其中 `C` 是 1~3 位的国家代码，`N` 最多可以用 14 位，而 `E` 是 (可选的) 分机号。前导加号和国家代码之后的点号是必需的。字面的“`x`”字符只有当需要提供分机号的时候才是需要的。

## 参见

实例 4.2 中提供了对北美电话号码进行合法性验证的更多选择。

ITU-T 推荐标准 E.123 (“Notation for national and international telephone numbers, e-mail addresses and Web addresses”) 可以从 <http://www.itu.int/rec/T-REC-E.123> 下载。

ITU-T 推荐标准 E.164 (“The international public telecommunication numbering plan”) 可以从 <http://www.itu.int/rec/T-REC-E.164> 下载。

全国编码方案可以从 <http://www.itu.int/ITU-T/inr/nnp> 下载。

RFC 4933 标准定义了 EPP 联系标识符 (其中包括了国际电话号码) 的语法和语义。可以从 <http://tools.ietf.org/html/rfc4933> 下载 RFC 4933。

## 4.4 传统日期格式的合法性验证

### 问题描述

你想要检查传统格式的日期：`mm/dd/yy`、`mm/dd/yyyy`、`dd/mm/yy` 和 `dd/mm/yyyy`。你

想要使用一个简单的正则表达式检查输入看起来是否像一个日期，而不用设法排除像 February 31st 这样的日期。

## 解决方案

匹配任意的上述日期格式，并允许省略前导 0：

```
^[0-3]?[0-9]/[0-3]?[0-9]/(?:[0-9]{2})?[0-9]{2}$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

匹配任意的上述日期格式，要求必须使用前导 0：

```
^[0-3][0-9]/[0-3][0-9]/(?:[0-9][0-9])?[0-9][0-9]$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

匹配 m/d/yy 和 mm/dd/yyyy，允许为日和月使用一个或两个数字，年使用两个或 4 个数字的任意组合：

```
^(1[0-2]|0?[1-9])/([3|01]|12)[0-9]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

匹配 mm/dd/yyyy，要求必须使用前导 0：

```
^(1[0-2]|0[1-9])/([3|01]|12)[0-9]|0[1-9])/[0-9]{4}$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

匹配 d/m/yy 和 dd/mm/yyyy，允许为日和月使用一个或两个数字、年使用两个或 4 个数字的任意组合：

```
^(3[01]|12)[0-9]|0?[1-9])/([1|0-2]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

匹配 dd/mm/yyyy，要求必须使用前导 0：

```
^(3[01]|12)[0-9]|0[1-9])/([1|0-2]|0[1-9])/[0-9]{4}$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

更严格地匹配任意的上述日期格式，允许省略前导 0：

```
^(?:([1|0-2]|0?[1-9])/([3|01]|12)[0-9]|0?[1-9])|  
(3[01]|12)[0-9]|0?[1-9])/([1|0-2]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

更严格地匹配任意的上述日期格式，要求必须使用前导 0：

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

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

如果使用宽松排列选项, 可以使最后两个正则表达式变得更加易读:

```
^(?:
  # m/d or mm/dd
  (1[0-2]|0?[1-9])/ (3[01]| [12] [0-9]|0?[1-9])
  |
  # d/m or dd/mm
  (3[01]| [12] [0-9]|0?[1-9])/ (1[0-2]|0?[1-9])
)
# /yy or /yyyy
/(?:[0-9]{2})? [0-9]{2}$
正则选项: 宽松排列
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^(?:
  # mm/dd
  (1[0-2]|0[1-9])/ (3[01]| [12] [0-9]|0[1-9])
  |
  # dd/mm
  (3[01]| [12] [0-9]|0[1-9])/ (1[0-2]|0[1-9])
)
# /yyyy
/[0-9]{4}$
正则选项: 宽松排列
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

## 讨论

你可能会认为像日期这样概念上很简单的格式对正则表达式来说应当很简单。但是事实并非如此, 这有两个原因。因为日期是大家每天都要用到的, 所以人们对于日期都比较随意。4/1 对你来说可能是愚人节。但是对其他人来说, 如果元旦是星期五的话, 那么它可能就是一年的第一个工作日。这里给的解答可以匹配一些最常见的日期格式。

另外一个问题时正则表达式并不会直接处理数字。举例来说, 你不能告知一个正则表达式“匹配 1~31 之间的一个数字”。正则表达式会挨个字符进行处理。我们使用 `<3[01]>|[12][0-9]|0?[1-9]>` 来匹配 3 后面跟着 0 或 1, 或者匹配 1 或 2 之后跟着任何数字, 或者匹配一个可选的 0 跟着 1~9。在字符类中, 我们可以使用单个字符的范围, 例如 `<[1-9]>`。这是因为数字 0~9 的字符在 ASCII 和 Unicode 字符表中占据的位置是连续的。第 6 章中会介绍关于使用正则表达式匹配各种数字的更多细节。

正因为此, 你必须在正则表达式的简单程度和精确程度之间做出选择。如果已经知道你的目标文本中不包含任何非法数据, 就可以使用一个简单的正则式: `\d{2}\d{2}\d{4}`。

虽然这个正则表达式会匹配像 99/99/9999 这样的数据，但是因为它们并不会出现在目标文本中，所以并没有关系。你可以很快就写出这样一个简单的正则式，而且它的执行速度也很快。

这个实例的前两个解答也很简单迅速，虽然它们也会匹配不合法的日期，比如 0/0/00 和 31/31/2008。它们只使用了字面字符来匹配日期分隔符，并使用字符类（参见实例 2.3）来把某些数字变成可选的。`\((?:[0-9]{2})?[0-9]{2}\)` 允许日期包含两位或四位数字。`[0-9]{2}` 会刚好匹配两位数字。`\((?:[0-9]{2})?\)` 匹配 0~2 个数字。其中的非捕获分组（参见实例 2.9）是必需的，因为问号需要被应用到字符类和量词 `{2}` 的组合之上。和 `[0-9]{2}` 一样，`\([0-9]{2}\)?` 也会正好匹配两个数字。没有这个分组的话，问号就会把量词变成懒惰的，因为 `{2}` 不能重复多余两次或者少于两次，那么问号起始不会起作用。

第 3~6 个解答中把月份限制为了 1~12 的数字，日期限制为了 1~31 的数字。我们在一个分组中使用了多选结构（参考实例 2.8）匹配不同的数字对，从而可以组成两位数字的一个范围。我们在这里使用了捕获分组，因为你一般总是会有需要把日和月的取值抓取下来。

最后两个解答则更为复杂一些，所以我们分别使用了紧凑和宽松排列模式来解释它们。这两种形式的唯一区别是可读性。JavaScript 并不支持宽松排列。最后的两个解决方案支持所有的日期格式，这和最前面的两个例子是一样的。区别是最后两个会使用额外的多选结构来把日期限制为 12/31 和 31/12，其中去掉了不合法的月份，例如 31/31。

## 变体

如果想要在更大的文本范围内查找日期，而不是检查一个输入是否是一个日期，那么你就不能再使用定位符 `(^)` 和 `(\$)`。只是简单地把定位符从正则表达式中删掉也不是正确的解决方案。举例来说，这样做会允许前面列出的正则表达式可以匹配在 9912/12/200199 中的 12/12/2001。因此，我们的做法不是把正则匹配定位到目标字符串的开始和结尾，而是必须规定日期不能作为更长数字序列的一部分。

这可以很容易地使用一对单词分界符来完成。在正则表达式中，数字都是被当作字符来处理的，因此它们可以作为单词的一部分。对这个问题的解答需要把 `(^)` 和 `(\$)` 都替换为 `\b`。例如：

```
\b(1[0-2]|0[1-9])/(3[01]|1[12])[0-9]|0[1-9])/[0-9]{4}\b  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 参见

实例 4.5、4.6 和 4.7。

## 4.5 对传统日期格式进行精确的合法性验证

### 问题描述

你想要验证传统格式的日期：mm/dd/yy、mm/dd/yyyy、dd/mm/yy 和 dd/mm/yyyy。你还想去掉像 February 31st 这样不合法的日期。

### 解决方案

#### C#

月在日前面：

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
    "^(?<month>[0-3]?[0-9])/(<day>[0-3]?[0-9])/" +
    "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
    int year = int.Parse(matchResult.Groups["year"].Value);
    if (year < 50) year += 2000;
    else if (year < 100) year += 1900;
    try {
        foundDate = new DateTime(year,
            int.Parse(matchResult.Groups["month"].Value),
            int.Parse(matchResult.Groups["day"].Value));
    } catch {
        // Invalid date
    }
}
```

日在月前面：

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
    "^(?<day>[0-3]?[0-9])/(<month>[0-3]?[0-9])/" +
    "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
    int year = int.Parse(matchResult.Groups["year"].Value);
    if (year < 50) year += 2000;
    else if (year < 100) year += 1900;
    try {
        foundDate = new DateTime(year,
            int.Parse(matchResult.Groups["month"].Value),
            int.Parse(matchResult.Groups["day"].Value));
    } catch {
        // Invalid date
    }
}
```

## Perl

月在日前面：

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);  
$validdate = 0;  
if ($subject =~ m!^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$!) {  
    $month = $1;  
    $day = $2;  
    $year = $3;  
    $year += 2000 if $year < 50;  
    $year += 1900 if $year < 100;  
    if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 ||  
                                              $year % 400 == 0)) {  
        $validdate = 1 if $day >= 1 && $day <= 29;  
    } elsif ($month >= 1 && $month <= 12) {  
        $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];  
    }  
}
```

日在月前面：

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);  
$validdate = 0;  
if ($subject =~ m!^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$!) {  
    $day = $1;  
    $month = $2;  
    $year = $3;  
    $year += 2000 if $year < 50;  
    $year += 1900 if $year < 100;  
    if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 ||  
                                              $year % 400 == 0)) {  
        $validdate = 1 if $day >= 1 && $day <= 29;  
    } elsif ($month >= 1 && $month <= 12) {  
        $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];  
    }  
}
```

## 完全使用正则表达式

月在日前面：

```
^(?:  
    # February (29 days every year)  
    (?<month>0?2) / (?<day>[12][0-9]|0?[1-9])  
    |  
    # 30-day months  
    (?<month>0?[469]|11) / (?<day>30|[12][0-9]|0?[1-9])  
    |  
    # 31-day months
```

```

(?<month>0?[13578]|1[02])/(?<day>3[01]|[12][0-9]|0?[1-9])
)
# Year
/ (?<year>(?:[0-9]{2})?[0-9]{2})$
正则选项: 宽松排列
正则流派: .NET

^(?:
  # February (29 days every year)
  (0?2)/([12][0-9]|0?[1-9])
|
  # 30-day months
  (0?[469]|11)/(30|[12][0-9]|0?[1-9])
|
  # 31-day months
  (0?[13578]|1[02])/([3][01]|[12][0-9]|0?[1-9]))
)
# Year
/ ((?:[0-9]{2})?[0-9]{2})$
正则选项: 宽松排列
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^(?:(0?2)/([12][0-9]|0?[1-9])|(0?[469]|11)/(30|[12][0-9]|0?[1-9]))|←
(0?[13578]|1[02])/([3][01]|[12][0-9]|0?[1-9]))/((?:[0-9]{2})?[0-9]{2})$
正则选项: 无
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```

日在月前面：

```

^(?:
  # February (29 days every year)
  (?<day>[12][0-9]|0?[1-9])/(?<month>0?2)
|
  # 30-day months
  (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[469]|11)
|
  # 31-day months
  (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[13578]|1[02]))
)
# Year
/ (?<year>(?:[0-9]{2})?[0-9]{2})$
正则选项: 宽松排列
正则流派: .NET

^(?:
  # February (29 days every year)
  ([12][0-9]|0?[1-9])/([0?2]
|
  # 30-day months
  (30|[12][0-9]|0?[1-9])/([469]|11)
|
  # 31-day months

```

```
(3[01] | [12][0-9] | 0?[1-9]) / (0?[13578] | 1[02])
)
# Year
/((?:[0-9]{2})? [0-9]{2})$  
正则选项: 宽松排列  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby  
  
^((?:([12][0-9] | 0?[1-9]) / (0?2) | (30 | [12][0-9] | 0?[1-9])) / ([469] | 11) | ←  
(3[01] | [12][0-9] | 0?[1-9]) / (0?[13578] | 1[02])) / ((?:[0-9]{2})? [0-9]{2})$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 讨论

本质上来说，使用正则表达式来精确验证日期可以有两种方式。一种是使用一个简单的的正则式，只是捕获看起来像是一个“年/月/日”组合的一组数字，然后使用过程代码来检查该日期是否正确。我们使用了前一个实例中的第一个正则式，允许在月和日中出现 0~39 的任意数字。这样就会很容易把日期格式从 mm/dd/yy 转换为 dd/mm/yy，因为我们只需要改变一下哪个捕获分组被当作是月就可以了。

这种方法的主要好处是可以很容易添加额外的限制，例如把日期限制为某个特定时期。许多编程语言都提供了处理日期的特定支持。C#解决方案中使用了 .NET 中的 `DateTime` 结构来检查日期是否合法，并且把日期以一种有用的格式返回，这些都可以在一步之内完成。

另外一种方法是使用正则表达式来完成所有任务。如果我们可以把每年都当作闰年，这种解决方案也是可行的。我们可以使用同样的技巧来罗列多选结构，就像我们在前一个实例中的最后几个方案中所给的解答一样。

使用单个正则表达式的问题在于它无法干净地把月和日都捕获到一个捕获分组中。现在，我们使用了 3 个捕获分组来处理月，用来处理日的也需要有 3 个分组。当正则表达式匹配到一个日期时，在正则式中的 7 个捕获分组中只有 3 个会真的捕获到内容。如果月是二月，分组 1 和 2 会捕获月和日。如果该月有 30 天，那么分组 3 和 4 会返回月和日。如果该月有 31 天，那么分组 5 和 6 会起作用。分组 7 总是会捕获到年份。

在这种情况下，只有.NET 正则流派对我们有帮助。.NET 支持给多个命名分组（参考实例 2.11）使用同一个名称，并且会为拥有相同名称的分组使用同一个存储空间。如果你使用前面给出的只适合.NET 的命名捕获解决方案，那么你可以从分组“month”和“day”匹配的文本中得到结果，而不用去担心一个月到底有多少天。但本书中讨论的所有其他流派或者不支持命名捕获，或者不支持两个分组拥有相同名称，或者是只会为任何给定名称返回最后一个分组所捕获的文本。对于这些流派来说，编号分组是唯一的解决方法。

只有当遇到必须只能使用一个正则表达式的情形，例如你使用的程序只提供一个输入

框来嵌入正则表达式的时候，才有必要采用完全正则表达式的解决方案。在编程的时候，多用一点儿代码会更加容易。而且如果你想要随后在日期之上再添加一些额外检查，那么使用代码也会非常有帮助。下面是一个用来匹配 2 May 2007~29 August 2008 的任意日期的一个纯正则表达式解决方案：

```
# 2 May 2007 till 29 August 2008
^(?:
  # 2 May 2007 till 31 December 2007
  (?:
    # 2 May till 31 May
    (?<day>3[01]|12[0-9]|0?[2-9])/(?<month>0?5)/(?<year>2007)
  |
    # 1 June till 31 December
    (?:
      # 30-day months
      (?<day>30|12[0-9]|0?[1-9])/(?<month>0?69|11)
    |
      # 31-day months
      (?<day>3[01]|12[0-9]|0?[1-9])/(?<month>0?78|1[02])
    )
  /(?<year>2007)
  )
|
# 1 January 2008 till 29 August 2008
(?:
  # 1 August till 29 August
  (?<day>[12][0-9]|0?[1-9])/(?<month>0?8)/(?<year>2008)
|
  # 1 January till 30 June
  (?:
    # February
    (?<day>[12][0-9]|0?[1-9])/(?<month>0?2)
  |
    # 30-day months
    (?<day>30|12[0-9]|0?[1-9])/(?<month>0?46)
  |
    # 31-day months
    (?<day>3[01]|12[0-9]|0?[1-9])/(?<month>0?1357)
  )
/(?<year>2008)
)
$
```

正则选项：宽松排列

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

## 参见

实例 4.4、4.6 和 4.7。

## 4.6 传统时间格式的合法性验证

### 问题描述

你想要验证各种不同传统时间格式的时间数据，例如 hh:mm 和 hh:mm:ss，并且要包括 12 小时制和 24 小时制两种时间格式。

### 解决方案

小时和分钟，12 小时制：

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9])$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

小时和分钟，24 小时制：

```
^(2[0-3]|01)?([0-9]):([0-5]?[0-9])$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

小时、分钟和秒，12 小时制：

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9]):([0-5]?[0-9])$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

小时、分钟和秒，24 小时制：

```
^(2[0-3]|01)?([0-9]):([0-5]?[0-9]):([0-5]?[0-9])$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

在所有这些正则表达式中的问号都会使前导 0 成为可选的。把问号都去掉就要求必须使用前导 0。

### 讨论

相对日期的合法性验证来说，检查时间会比较容易。每小时包含 60 分钟，而每分钟包含 60 秒。这意味着我们并不需要在正则表达式中使用太复杂的多选结构。对于分钟和秒来说，我们根本不需要使用多选结构。`[0-5]?[0-9]` 会匹配 0~5 之间的一个数字，紧跟着 0~9 的一个数字。这样就可以正确匹配到 0~59 的任意数字。在第一个字符类之后的问号会把它变成可选的。这样，单个 0~9 之间的数字同样也会被认为是一个合法的分钟或秒。如果你期望把小于 10 的分钟和秒写作是 00~09，那么就应该把问号去掉。关于字符类以及问号等量词的更多细节，请参考实例 2.3 和实例 2.12。

对于小时来说，我们还是要使用多选结构（参考实例 2.8）。根据第一个数字，第二个

数字会允许不同的取值范围。对于 12 小时制的时钟，如果第一个数字是 0，那么第二个数字允许所有的 10 个数字，但是如果第一个数字是 1，那么第二个数字必须是 0、1 或 2。在正则表达式中，我们可以把它表示为 `<1[0-2]|0?[1-9]>`。如果使用 24 小时制的时钟，那么如果第一个数字是 0 或 1，第二个数字允许所有 10 个数字，但如果第一个数字是 2，那么第二个数字必须位于 0~3 之间。使用正则表达式，它可以表示为 `2[0-3]|01|[0-9]`。在这里使用的问号同样支持小于 10 的小时可以被写作单个数字。把问号去掉就会要求必须使用两位数字。

我们在匹配小时、分钟和秒的正则组成部分两边都加了括号。这样就可以很容易把冒号略去，分别获得小时、分钟和秒的取值。实例 2.9 中讲解了如何使用圆括号创建捕获分组。实例 3.9 中讲解了你如何在过程代码中获取捕获分组匹配到的文本。

小时部分两边的括号会把两种不同的小时格式保持在一起。如果你去掉这些括号，那么正则表达式就无法正常工作。而把分钟和秒两边的括号去掉则不会产生问题，只是会造成我们无法单独获取表示它们的数字。

## 变体

如果想要在更大的文本范围内查找时间，而不是检查一个输入是否是一个日期，那么你就不能再使用定位符 `<^>` 和 `<$>`。只是简单地把定位符从正则表达式中删掉也不是正确的解决方案。举例来说，这样做会允许表示小时和分钟的正则表达式可以匹配到在 9912:1299 中的 12:12。因此，我们的做法不是把正则匹配定位到目标字符串的开始和结尾，而是必须规定时间不能作为更长数字序列的一部分。

这可以很容易地使用一对单词分界符来完成。在正则表达式中，数字被当作字符来处理，它可以作为单词的一部分出现。因此，我们可以把 `<^>` 和 `<$>` 都替换为 `\b`。例如：

```
\b(2[0-3]|01)?[0-9]:([0-5]?[0-9])\b  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

单词边界并不会禁止所有字符；它们只会禁止字母、数字和下划线。上面的正则表达式会匹配 24 小时制时钟的小时和分钟，它们依然会匹配目标文本 The time is 16:08:42 sharp 中的 16:08。空格并不是单词字符，而 `:` 是单词字符，所以单词边界 `\b` 会匹配二者之间的位置。`8` 是一个单词字符，而冒号不是，所以 `\b` 也会匹配它们两个之间的位置。

如果你想要同时禁止单词字符和冒号的出现，那么就需要使用环视（参考实例 2.16）。下面的正则表达式不会匹配 The time is 16:08:42 sharp 中的任何部分。它只适用于支持环视的正则流派：

```
(?<![:\w])(2[0-3]|01)?[0-9]:([0-5]?[0-9])(?![:\w])  
正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby 1.9
```

## 参见

实例 4.4、实例 4.5 和实例 4.7。

## 4.7 检查 ISO 8601 格式的日期和时间

### 问题描述

你想要匹配在官方的 ISO 8601 格式中的日期和/或时间，这个格式是许多标准化时间和日期格式的基础。例如，在 XML Schema 中，内置的 date、time 和 dateTime 这些类型都是基于 ISO 8601 的。

### 解决方案

下面的正则表达式会匹配一个公历月份，例如 2008-08。连字符是必需的：

```
^([0-9]{4})-(1[0-2]|0[1-9])$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby  
  
^(?<year>[0-9]{4})-(?<month>1[0-2]|0[1-9])$  
正则选项: 无  
正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9  
  
^(?P<year>[0-9]{4})-(?P<month>1[0-2]|0[1-9])$  
正则选项: 无  
正则流派: PCRE、Python
```

下面的正则表达式会匹配一个公历日期，例如 2008-08-30。连字符是可选的。这个正则式允许出现 YYYY-MMDD 和 YYYYMM-DD，这两种形式并不遵守 ISO 8601：

```
^([0-9]{4})-?(1[0-2]|0[1-9])-?(3[0-1]|0[1-9]|1[2][0-9])$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby  
  
^(?<year>[0-9]{4})-?(?<month>1[0-2]|0[1-9])-?  
(?<day>3[0-1]|0[1-9]|1[2][0-9])$  
正则选项: 无  
正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9
```

下面的正则表达式会匹配一个公历日期，例如 2008-08-30。连字符是可选的。这个正则式使用了一个条件来排除 YYYY-MMDD 和 YYYYMM-DD。对于第一个连字符添加了一个额外的捕获分组：

```
^([0-9]{4})(-)?(1[0-2]|0[1-9])(?(2)-)(3[0-1]|0[1-9]|1[2][0-9])$  
正则选项: 无  
正则流派: .NET、PCRE、Perl、Python
```

下面的正则表达式会匹配一个公历日期，例如 2008-08-30。连字符是可选的。这个正则

式使用多选结构来排除 YYYY-MMDD 和 YYYYMM-DD。对于月份来说存在两个捕获分组：

```
^([0-9]{4})(?:([1][0-2]|0[1-9])|-[([1][0-2]|0[1-9]))-?)$  
(3[0-1]|0[1-9]|1[1-2][0-9])$
```

正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

下面的正则表达式会匹配一年中的某一周，例如 2008-W35。连字符是可选的：

```
^([0-9]{4})-?W(5[0-3]|1-4)[0-9]|0[1-9])$
```

正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|1-4)[0-9]|0[1-9])$
```

正则选项：无  
正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

下面的正则表达式会匹配一年中用周次来表示的日期，例如 2008-W35-6。连字符是可选的：

```
^([0-9]{4})-?W(5[0-3]|1-4)[0-9]|0[1-9])-?([1-7])$
```

正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|1-4)[0-9]|0[1-9])-?(<day>[1-7])$
```

正则选项：无  
正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

下面的正则表达式会匹配用序数表示的日期，例如 2008-243。连字符是可选的：

```
^([0-9]{4})-?(36[0-6]|3[0-5][0-9]|12)[0-9]{2}|0[1-9][0-9]|00[1-9])$
```

正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```
^(?<year>[0-9]{4})-?  
(?<day>36[0-6]|3[0-5][0-9]|12)[0-9]{2}|0[1-9][0-9]|00[1-9])$
```

正则选项：无  
正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

下面的正则表达式会匹配小时和分钟，例如 17:21。冒号是可选的：

```
^(2[0-3]|01)?[0-9]):?([0-5]?[0-9])$
```

正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```
^(?<hour>2[0-3]|01)?[0-9]):?(<minute>[0-5]?[0-9])$
```

正则选项：无  
正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

下面的正则表达式会匹配小时、分钟和秒，例如 17:21:59。冒号是可选的：

```
^(2[0-3]|01)?[0-9]):?([0-5]?[0-9]):?([0-5]?[0-9])$
```

正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```
^(?<hour>2[0-3]| [01]?[0-9]):?(?<minute>[0-5]?[0-9]):?←  
(?<second>[0-5]?[0-9])$  
正则选项: 无  
正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9
```

下面的正则表达式会匹配时区指示，例如 Z、+07 或 +07:00。冒号和分钟是可选的：

```
^(Z| [+-] (?::2[0-3]| [01]?[0-9]) (?::? (?::[0-5]?[0-9]))?)$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

包含时区指示的小时、分钟和秒，例如 17:21:59+07:00。所有的冒号都是可选的。在时区指示中的分钟也是可选的：

```
^(2[0-3]| [01]?[0-9]):?([0-5]?[0-9]):?([0-5]?[0-9])←  
(Z| [+-] (?::2[0-3]| [01]?[0-9]) (?::? (?::[0-5]?[0-9]))?)$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby  
^(?<hour>2[0-3]| [01]?[0-9]):?(?<minute>[0-5]?[0-9]):?(?<sec>[0-5]?[0-9])←  
(?<timezone>Z| [+-] (?::2[0-3]| [01]?[0-9]) (?::? (?::[0-5]?[0-9]))?)$  
正则选项: 无  
正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9
```

包含可选时区指示的日期，例如 2008-08-30 或 2008-08-30+07:00。连字符是必需的。这个是 XML Schema 中使用的 date 类型：

```
^(-?(:[1-9][0-9]*)?[0-9]{4})-(1[0-2]| 0[1-9])- (3[0-1]| 0[1-9]| [1-2][0-9])←  
(Z| [+-] (?::2[0-3]| [0-1][0-9]):[0-5][0-9]))??$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby  
^(?<year>-?(:[1-9][0-9]*)?[0-9]{4})- (?<month>1[0-2]| 0[1-9])-←  
(?<day>3[0-1]| 0[1-9]| [1-2][0-9])←  
(?<timezone>Z| [+-] (?::2[0-3]| [0-1][0-9]):[0-5][0-9]))??$  
正则选项: 无  
正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9
```

包含可选小数部分和时区的时间格式，例如 01:45:36 或 01:45:36.123+07:00。这是 XML Schema 中使用的 time 类型：

```
^(2[0-3]| [0-1][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?←  
(Z| [+-] (?::2[0-3]| [0-1][0-9]):[0-5][0-9]))??$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby  
^(?<hour>2[0-3]| [0-1][0-9]):(?<minute>[0-5][0-9]):(?<second>[0-5][0-9])←  
(?<ms>\.[0-9]+)?(?<timezone>Z| [+-] (?::2[0-3]| [0-1][0-9]):[0-5][0-9]))??$  
正则选项: 无  
正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9
```

包含可选小数部分和时区的日期和时间格式，例如 2008-08-30T01:45:36 或 2008-08-

30T01:45:36.123Z。这是 XML Schema 中使用的 `dateTime` 类型：

```
^(?:(?:[1-9][0-9]*)(?:[0-9]{4})-(1[0-2]|0[1-9])-(3[0-1]|0[1-9]|1[2-9][0-9])  
T(2[0-3]|0[1-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?  
(Z|[-](:2[0-3]|0[1-9]):[0-5][0-9])?  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby  
  
^(?<year>-?(:[1-9][0-9]*)(?:[0-9]{4})-(?<month>1[0-2]|0[1-9])-  
(?<day>3[0-1]|0[1-9]|1[2-9][0-9])T(?<hour>2[0-3]|0[1-9]):  
(?<minute>[0-5][0-9]):(?<second>[0-5][0-9])(?<ms>\.[0-9]+)?  
(?<timezone>Z|[-](:2[0-3]|0[1-9]):[0-5][0-9])?  
正则选项：无  
正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9
```

## 讨论

ISO 8601 定义了各种形式的日期和时间格式。这里所给的正则表达式只覆盖了其中最常见的形式，但是大多数使用 ISO 8601 的系统也只会使用其中的一个子集。例如，在 XML Schema 的日期和时间中，连字符和冒号是必需的。要把连字符和冒号变为必需的，只需要简单地去掉它们之后紧跟的问号。要禁止连字符和冒号的话，把它们连同随后的问号一起去掉。但一定要注意使用 `<(:group)>` 语法的非捕获分组。如果问号和冒号紧跟在一个左圆括号之后，那么这三个字符是作为一个非捕获分组的开始。

这里给的正则表达式可以把单独的连字符和冒号变成可选，而这并没有严格遵守 ISO 8601 的规定。例如，1733:26 就不是一个合法的 ISO 8601 时间，但是会被我们给的时间正则式所接受。要想实现所有的连字符和冒号要么同时出现，要么同时不出现，会得到一个相当复杂的正则表达式。在日期正则式中，我们采用了这样的规定，但是在时间中，比如是在 XML Schema 中，通常分隔符不是随意可选的，而通常是只能选择必需，或者禁止使用。

我们在正则式的所有数字部分两边都放了括号。这样可以很容易就获得关于年、月、日，小时、分钟、秒和时区的对应数字。实例 2.9 中讲解如何使用圆括号来创建捕获分组。实例 3.9 中讲解了如何在过程代码中获得捕获分组匹配到的文本。

对于大多数正则式，我们都给出了另外一个使用命名捕获的解答。对读者或者你们的同事来说，其中一些日期和时间格式可能会不是那么熟悉。命名捕获会使正则表达式更加容易理解。.NET、PCRE 7、Perl 5.10 和 Ruby 1.9 都支持命名捕获的 `<(?<name>group)>` 语法。本书中讲解的所有 PCRE 和 Python 版本都会支持另外一种形式的 `<(P<name>group)>` 语法。更多细节，请参考实例 2.11 和实例 3.9。

在所有正则式中的数字范围都是严格的。例如，公历日期被限制在 01~31。你永远你不会得到 32 日或者 13 月。本实例中的所有正则式都没有试图去排除不合法的日月组合，例如 2 月 31 日；实例 4.5 中讲解了如何进行这样的处理。

虽然其中有些正则式看起来比较长，但是它们都比较容易理解，而且使用的是在实例 4.4 和实例 4.6 中讲解过相同的技巧。

## 参见

实例 4.4、实例 4.5 和实例 4.6。

# 4.8 限制输入只能为字母数字字符

## 问题描述

你的程序要求用户的响应中只能使用英语字母表中的一个或多个字母数字字符。

## 解决方案

既然你可以使用正则表达式，这个问题就变得无比简单了。一个字符类就可以说明所允许的字符范围。再加上一个量词来重复这个字符类一次或多次，使用定位符来把匹配定位到字符串的开始和结束。你就大功告成了！

## 正则表达式

`^[A-Z0-9]+$`  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## Ruby

```
if subject =~ /^[A-Z0-9]+$/i
    puts "Subject is alphanumeric"
else
    puts "Subject is not alphanumeric"
end
```

## 其他编程语言

如果读者想了解如何在其他编程语言中实现这个正则表达式，请参考实例 3.4 和实例 3.5。

## 讨论

我们来分别看一下这个正则表达式中的 4 个组成部分：

`^` # 判断字符串的起始位置  
`[A-Z0-9]` # 匹配从"A"到"Z"或从"0"到"9"的一个字符...  
`+` # 重复 1 次或多次  
`$` # 判断字符串的结尾位置  
正则选项：不区分大小写，宽松排列  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

位于正则表达式开头和结尾的两个断言 `\^` 和 `\$` 会确保我们测试的是整个输入字符串。如果没有它们，那么正则式会匹配一个更长字符串中的任意部分，这样就会导致出现非法字符。加号量词 `\+\>` 会重复之前的元素一次或多次。如果想要允许正则式匹配一个全空的字符串，那么你可以把 `\+\>` 替换为 `\*\>`。星号量词 `\*\>` 会允许 0 次或多次重复，实际上也就可以把之前的元素变成可选的。

## 变体

### 限制输入为 ASCII 字符

下面的正则表达式会把输入限制为 7 比特 ASCII 字符表中的 128 个字符。其中包括了 33 个不可见的控制字符：

```
^[\x00-\x7F]+\$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### 限制输入为 ASCII 非控制字符和换行符

使用下面的正则表达式可以把输入限制为在 ASCII 字母表中的可见字符和空白，而把控制字符去掉。换行符和回车符（分别位于位置 0x0A 和 0x0D）是最常用的控制字符，所以我们使用 `\n\r`（换行）和 `\r\n`（回车）把这两个字符明确地包含了进来：

```
^[\n\r\x20-\x7E]+\$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### 限制输入为 ISO-8859-1 和 Windows-1252 共享的字符

ISO-8859-1 和 Windows-1252（通常被称作是 ANSI）是两种最常用的 8 比特字符编码格式，它们都基于 Latin-1 标准（更正式的名称是 ISO/IEC 8859-1）。然而，它们在位置 0x80 和 0x99 之间映射的字符是不兼容的。ISO-8859-1 把这些位置用于控制代码，而 Windows-1252 则把这些位置用于扩展的字符和标点。

这些区别有时候会造成字符显示的困难，特别是对于没有明确声明它们使用的编码格式的文档，或者接收者用的不是 Windows 系统的时候。下面的正则表达式可以用来把输入限制为只能是 ISO-8859-1 和 Windows-1252 中共享的字符（包括共享的控制字符）：

```
^[\x00-\x7F\xA0-\xFF]+\$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

十六进制的记号可能会让这个正则表达式看起来比较难读，但是它的工作原理同 `[A-Z0-9]` 这样的字符类是一样的。它会匹配下列两个范围之内的字符：`\x00-\x7F` 和 `\xA0-\xFF`。

## 限制输入为任何语言的字母数字字符

这个正则表达式把输入限制为可以是任何语言或字母表中的字母和数字。它使用的字符类中包含了在 Unicode 字母和数字类别中的所有代码点的属性：

```
^[\p{L}\p{N}]+$  
正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Ruby 1.9
```

不幸的是，并不是本书中的所有正则表达式流派都支持 Unicode 属性。具体来讲，这个正则式就不能用于 JavaScript、Python 或 Ruby 1.8。另外，在 PCRE 中使用这个正则式时要求 PCRE 必须使用 UTF-8 来进行编译。如果在正则式中添加了/u 选项，那么 Unicode 属性可以用于 PHP 中的 preg 函数类（它依赖的是 PCRE）。

在 Python 中可以使用如下的正则表达式：

```
^[^\w_]+$  
正则选项: Unicode  
正则流派: Python
```

这里，我们设法绕开了在 Python 中不支持 Unicode 属性的问题，在创建正则表达式的时候需要使用 UNICODE 或 U 标志。这样就可以通过迫使一些正则记号使用 Unicode 字符表来改变它们的含义。`\w` 对我们来说差不多就足够了，但是它会匹配字母数字字符以及下划线。通过在一个否定字符类中使用 `\w` 的取反 (`\W`)，我就可以从这个集合中把下划线去掉。像这样的双重否定在正则表达式中偶尔会很有用，虽然这可能有时候会让你有点儿摸不着头脑<sup>1</sup>。

## 参见

实例 4.9 中会演示如何限制文本的长度，而不是限制其中的字符集。

## 4.9 限制文本长度

### 问题描述

你想要检查一个字符串是否是由 1~10 个 A~Z 的字母组成的。

### 解决方案

本书中的所有编程语言都会提供简单高效的方式来检查文本的长度。例如，JavaScript 字符串有一个 `length` 属性，其中会保存一个说明字符串长度的整数。然而在有些情形

<sup>1</sup> 这种情形还可以更加有趣（如果你可以把它称为有趣）：读者可以尝试把否定型环视（参见实例 2.16）和字符类查（参见实例 2.3 中的“流派相关的特性”一节）也加进来，创建一些三重、四重甚至更多重的否定。

下，使用正则表达式来检查文本长度会比较有用，特别是当长度只是用来决定目标文本是否符合指定模式的多条规则之一的时候。下面的正则表达式会确保目标文本中只包含 1~10 个字符，并且还进一步把文本限制为只能包含大写字母 A~Z。你也可以修改正则表达式，来支持任意的最小和最大文本长度，或者允许出现 A~Z 之外的字符。

## 正则表达式

`^[A-Z]{1,10}$`

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## Perl

```
if ($ARGV[0] =~ /^[A-Z]{1,10}$/) {  
    print "Input is valid\n";  
} else {  
    print "Input is invalid\n";  
}
```

## 其他编程语言

关于如何使用其他编程语言来实现这个正则表达式，请参考实例 3.5 中的讲解。

## 讨论

下面是对上面这个非常容易理解的正则式的分解：

`^` # 判断字符串的起始位置  
`[A-Z]` # 匹配从"A"到"Z"的一个字母...  
`{1,10}` # 1 到 10 次  
`$` # 判断字符串的结束位置

正则选项：宽松排列

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

两个定位符(`^`)和(`$`)会确保正则表达式匹配整个目标字符串；否则，它就可能会匹配位于更长文本中的 10 个字符。字符类(`[A-Z]`)会匹配 A~Z 的任意单个大写字母，而区间量词(`{1,10}`)会重复该字符类 1~10 次。通过把区间量词与字符串开始和结束定位符组合起来之后，如果目标文本的长度不在指定范围之内，这个正则表达式就不会产生匹配。

注意字符类(`[A-Z]`)明确说明了只允许大写字母。如果你想要把 a~z 的小写字母也加进来，那么可以选择把字符类替换成`[A-Za-z]`，或者是使用不区分大小写的选项。更多细节，请参考实例 3.4 中的讲解。

正则表达式的新用户常常会犯的一个错误是，想要使用字符类范围`[A-z]`来节省几个字符。一眼看去这样好像可以把所有大小写字母都包含进来。然而，在 ASCII 字符表中，`A~Z` 和 `a~z` 的范围之间还包含了几个标点字符。因此，实际上和`[A-z]`等价的是

<[A-Z[\]^\_a-z]>。

## 变体

### 限制任意模式的长度

因为像<{1,10}>这样的量词只能应用到紧跟在它前面的元素，因此如果要对包含超过一个记号的模式所能匹配的字符个数进行限制，那么就需要采用一种不同的方式。

在实例 2.16 中讲过，顺序环视（以及和它相对的逆序环视）是与<^>和<\$>类似的一种特殊的断言，用来匹配在目标字符串的一个位置，而不会消费任何字符。顺序环视可以是肯定型或者否定型的，这意味着它们可以在匹配中检查一个模式符合或者不符合当前的位置。肯定型顺序环视（采用<(?=...)>的格式）可以被用在模式的开头来保证字符串的长度在指定范围之内。正则表达式的剩余部分则可以去放心检查想要的模式，而不用再担心文本的长度。下面是一个简单的例子：

```
^(?=.{1,10}$).*
正则选项：点号匹配换行符
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

^(?=[\S\s]{1,10}$)[\S\s]*
正则选项：无
正则流派：JavaScript
```

注意一定要把定位符<^>放到顺序环视之内，因为只有确定在到达上限之后不再存在更多字符时，我们才能认为它符合最大长度检查。因为在正则式开头的顺序环视会保证长度的范围，所以剩余的模式可以应用额外的验证规则。在这个例子中，模式<\*>（或者是添加了 JavaScript 支持的<[\S\s]\*>版本）被用来简单地匹配整个目标文本，而没有添加任何限制。

第一个正则式使用了“点号匹配换行符”选项，从而在你的目标字符串中包含换行符的时候也能正常工作。关于如何在编程语言中应用这个修饰符，请参考实例 3.4。JavaScript 中不支持“点号匹配换行符”的选项，所以第二个正则式中使用了一个字符类来匹配任意字符。更多相关信息，请参考实例 2.4 中的“包含换行符在内的任意字符”小节。

### 限制非空白字符的个数

下面的正则表达式会匹配包含 10~100 个非空白字符的任意字符串：

```
^\s*(?:(\S\s*){10,100})
正则选项：无
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

在 Java、PCRE、Python 和 Ruby 中，<\s>会只匹配 ASCII 空白字符，而<\S>则会匹配所有其他的字符。在 Python 中，你可以在创建正则式的时候，使用 UNICODE 或 U 标

志来让 `\s` 可以匹配所有的 Unicode 空白符号。使用 Java、PCRE 和 Ruby 1.9 的开发人员，如果想要避免把任何 Unicode 空白符号算进它们的字符个数中，就需要采用如下的版本来利用 Unicode 属性（参考实例 2.7）。

```
^[\p{Z}\s]*(?:[^p{Z}\s][\p{Z}\s]*){10,100}$  
正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Ruby 1.9
```

PCRE 必须使用 UTF-8 来编译，上述正则式才能工作。在 PHP 中，需要使用/u 模式修饰符来打开 UTF-8 支持。

上面这个正则式组合了 Unicode 中的 `\p{Z}` Separator (分隔符) 属性，以及空白符号的 `\s` 简写。这是因为由 `\p{Z}` 和 `\s` 匹配的字符并不会完全重叠。`\s` 中会包含位置从 0x09 到 0x0D (制表符、换行、垂直制表符、换页和回车) 的字符，而这些字符并没有在 Unicode 标准中赋予 Separator 属性。通过在一个字符类中把 `\p{Z}` 和 `\s` 组合起来，你就可以确保能够匹配所有的空白字符。

在两个正则式中，区间量词 `{10,100}` 都被应用到了它之前的非捕获分组上，而不是应用到单个记号。该分组会匹配任意单个的非空白字符紧跟 0 个或多个空白字符。因为每次循环会恰好匹配一个非空白字符，所以区间量词就会可靠地记录到底匹配了多少个非空白字符。

## 限制单词的个数

下面的正则表达式与上一个例子中非常相似，区别只是每次重复会匹配整个单词，而不是单个的非空白字符。它会匹配 10~100 个单词，并忽略任意非单词字符，其中包括标点符号和空白：

```
^\w*(?:\w+\b\w*){10,100}$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

在 Java、JavaScript、PCRE 和 Ruby 中，`\w` 单词字符记号只会匹配 ASCII 字符中的 A~Z、a~z、0~9 和 \_，因此这并不能正确地统计包含非 ASCII 字母和数字的单词个数。在 .NET 和 Perl 中，`\w` 会基于 Unicode 字符表（这也包括了和它相反的 `\W`，以及单词边界 `\b`），因此会匹配到所有 Unicode 字母表中的字符和数字。在 Python 中，你可以通过在创建正则式的时候传递一个 UNICODE 或 U 标志来决定这些记号是否是基于 Unicode 的。

如果你想要统计包含非 ASCII 字母和数字的单词，那么下面的正则表达式会为其他正则流派提供这种能力：

```
^[\p{L}\p{N}_]*(?:[\p{L}\p{N}_]+\b[\p{L}\p{N}_]*){10,100}$  
正则选项: 无  
正则流派: .NET、Java、Perl
```

```
^[\^p{L}\p{N}_]*(?:[\p{L}\p{N}_]+(?:[\^p{L}\p{N}_]+\$)){10,100}$  
正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Ruby 1.9
```

在 PCRE 中必须使用 UTF-8 支持来进行编译才可以使用。在 PHP 中，需要使用/u 模式修饰符来打开 UTF-8 支持。

我们已经提到，之所以会有这么多不同（但是等价）的正则式，是因为对于单词字符和单词边界记号存在不同的处理方法，这在实例 2.6 中的“单词字符”小节中有详细的解释。

在最后两个正则式使用的字符类中为单词和字母使用了分别的 Unicode 属性 (<\p{L}> 和 <\p{N}>)，并且手动在每个类中添加了下划线字符，从而它们可以同前面依赖于<\w> 和 <\W> 的正则式保持等价。

在这 3 个正则式的前两个中，每次非捕获分组的重复都会匹配一个单词与其后跟着的 0 个或多个非单词字符。在分组中的记号<\W>（或<[\^p{L}\p{N}\_]>）会被允许重复 0 次，因为字符串可能会以一个单词字符结束。然而，因为这样就会使得非单词字符在整个匹配过程中都变成了可选的，所以就需要在<\w> 和 <\W>（或者 <[\p{L}\p{N}\_]> 和 <[\^p{L}\p{N}\_]>）之间添加断言，以确保分组的每次重复都确实会匹配整个单词。如果不使用这个单词分界，那么一次重复就可能会匹配到单词中的任意片段，而随后的重复会匹配其他的片段。

正则式的第三个版本（添加了对 PCRE 和 Ruby 1.9 的支持）的工作原理则有些不同。它使用了一个加号量词（1 次或多次），而不是星号量词（0 次或多次）。并且明确指出只有在匹配过程到达字符串结尾的时候，才允许匹配 0 个字符。这样就允许我们可以避免使用单词分解标记，这样还有助于保证准确性，因为在 PCRE 或 Ruby 中，<\b> 并不支持 Unicode。

不幸的是，这些选项都无法使 JavaScript 或者 Ruby 1.8 可以正确处理使用非 ASCII 字符的单词。一种可能的解决方案是让正则式统计空白个数，而不是单词字符序列的个数，如下所示：

```
^\s*(?:\S+(?:\s+|$)){10,100}$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、Perl、PCRE、Python、Ruby
```

在很多情况下，这个正则式会同前面的解决方案效果相同，但是它并不是完全等价的。例如，一个区别是由连字符连接的复合单词（例如“far-reaching”）现在会被当作一个单词，而不是两个。

## 参见

实例 4.8 和实例 4.10。

## 4.10 限制文本中的行数

### 问题描述

你需要检查一个字符串中是否包含 5 行或者更少内容，而不用管在字符串中总共出现了多少个字符。

### 解决方案

根据你的操作系统中的约定、应用程序或用户选项等不同，实际用作换行分隔符的字符或是字符序列也会发生改变。因此，要想构造一个完美的解决方案就需要回答关于应当支持使用什么样的约定来作为新行开始标志的问题。下面的解决方案中支持标准的 MS-DOS/Windows (`\r\n`)、老版本的 Mac OS (`\r`) 和 Unix/Linux/OS X (`\n`) 中的换行约定。

### 正则表达式

下列 3 个特定于流派的正则式包含两个区别。第一个正则式使用了原子分组，它的格式是`(?>...)`，而没有使用`(?:...)`格式的非捕获分组，因为如果正则流派支持前者，那么它可能会带来性能上潜在的小幅提高。Python 和 JavaScript 不支持原子分组，所以它们就不能使用第一个正则式。另外一个区别是用来判断字符串开始和结束位置的记号 (`\A` 或 `\^` 用于字符串的开始，`\z`、`\Z` 或 `\$` 用于结束)。这种变化的原因会在本实例随后内容中深入进行讨论。所有这 3 个特定于流派的正则式都会匹配完全一样的字符串：

```
\A(?>(?:\r\n?|\n)?[^\\r\\n]*){0,5}\z
正则选项: 无
正则流派: .NET、Java、PCRE、Perl、Ruby

\A(?:(?:\r\n?|\n)?[^\\r\\n]*){0,5}\Z
正则选项: 无
正则流派: Python

^(?:(?:\r\n?|\n)?[^\\r\\n]*){0,5}$
正则选项: 无
正则流派: JavaScript
```

### PHP (PCRE)

```
if (preg_match('/^(?>(?:\r\n?|\n)?[^\\r\\n]*){0,5}\z/', $_POST['subject'])) {
    print 'Subject contains five or fewer lines';
} else {
    print 'Subject contains more than five lines';
}
```

## 其他编程语言

如果读者想知道在其他编程语言中如何实现这些正则表达式，请参考实例 3.5 中的讲解。

## 讨论

本实例中前面给出的所有正则表达式都使用了一个分组，来匹配 MS-DOS/Windows、legacy Mac OS 或者 Unix/Linux/OS X 中的一个换行序列，紧跟着任意个数的非换行字符。分组会被重复 0~5 次，从而我们只能匹配最多 5 行。

在下面的例子中，我们把上面 JavaScript 版本的正则式拆分开来研究。我们之所以在这里使用 JavaScript 的版本，是因为它的组成部分对于更多读者来说可能会更为熟悉。随后我们会解释不同正则流派之间的区别：

```
^          # 判断字符串的开始位置
(?:        # 分组但是并不捕获...
 (?:      # 分组但是并不捕获...
    \r      # 匹配一个回车字符 (CR, ASCII 位置 0x0D)
    \n      # 匹配一个换行字符 (LF, ASCII 位置 0x0A)...
    ?       # 0 次或 1 次
  |       # 或者...
    \n      # 匹配一个换行字符
  )       # 非捕获分组的结束
  ?       # 重复之前的分组 0 次或 1 次
[^\\r\\n]   # 匹配除了 CR or LF 之外的任意单个字符...
  *       # 0 次或无限多次
)         # 非捕获分组的结束
{0,5}     # 重复之前的分组 0~5 次
$          # 判断字符串的结束位置
正则选项：宽松排列
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

最开始的 `<^>` 会匹配字符串的开始位置。这会有助于确保整个字符串中包含不多于 5 行，因为除非正则式一定从字符串的开头开始匹配，否则它就可能会匹配一个更长字符串中的任意 5 行内容。

接着，我们使用了一个非捕获分组来包括一个换行序列和任意个数的非换行字符的组合。紧跟其后的量词会允许这个分组重复 0~5 次（0 次重复会匹配一个全空字符串）。在外层分组中，一个可选的子分组会匹配一个换行序列。后面跟着的是一个字符类，它可以匹配任意个数的非换行字符。

再仔细看一下外层分组中元素的顺序（首先是一个换行符，然后是一个非换行字符序列）。如果我们把顺序反过来，那么这个分组就会被写作 `<(?:[^\\r\\n]*(?:\\r\\n?|\\n)?)>`，这样在第 5 次重复中会允许出现一个拖尾的换行。实际上，这样改动之后就会允许出现一

个空的第 6 行。

子分组中允许出现下列三种换行序列：

- 一个回车之后跟着一个换行（`\r\n`，在 MS-DOS/Windows 使用的换行序列）；
- 单独的回车（`\r`，在老的 Mac OS 中的换行字符）；
- 单独的换行（`\n`，在 Unix/Linux/OS X 中使用的换行字符）。

现在我们来解释一下不同流派之间的区别。

正则式的第一版（用于除了 Python 和 JavaScript 之外的所有流派）中使用了原子分组，而不是简单的非捕获分组。虽然在有些情况下，使用原子分组会带来更加复杂的影响，在这个例子中，它们只是简单告诉正则引擎在匹配尝试失败时避免可能会执行的一些不必要的回溯（关于原子分组的更多信息，请参考实例 2.15）。

其他跨流派的区别是用来判断字符串起始和结束位置的记号。在上面的分解形式中使用的是`^` 和 `$`。虽然这两个定位符在所有的正则流派中都支持，但是在本节的其他正则式中使用了`\A`、`\Z` 和 `\z`。对此的简单解释是在不同的正则表达式流派之间这些元字符的含义稍有区别。如果要详细解释的话，那么就需要深入了解一些正则表达式的历史。

当使用 Perl 来从文件中读入一行的时候，得到的结果字符串会以换行符作为结束。因此，Perl 对 `$` 的传统含义引入了一个“改进”，而大多数其他正则流派也沿用了这种改进。除了匹配一个字符串的绝对结束之外，Perl 中的 `$` 还会匹配结束字符串的换行位置。Perl 同时还引入了两个断言来匹配字符串的结束：`\Z` 和 `\z`。Perl 中的 `\Z` 定位符与 `$` 拥有同样诡异的含义，但是当使用额外选项允许 `^` 和 `$` 匹配换行处的时候，前者并不会产生同样的变化。`\z` 总是会只匹配字符串的绝对结束，从无例外。因为这个实例中需要处理换行符来统计一个字符串中的行数，所以在支持 `\z` 的流派中都使用了 `\z` 断言，以确保不会匹配到一个空的第 6 行。

大多数其他正则流派都沿用了 Perl 中的行结束和字符串结束定位符。.NET、Java、PCRE 和 Ruby 都会支持 `\Z` 和 `\z`，其含义与在 Perl 中是一样的。Python 中只包含了大写形式的 `\Z`，但是却令人费解地把它的含义改变成了只匹配字符串的绝对结束，这与 Perl 中小写形式的 `\z` 是一样的。JavaScript 没有包括任何“z”定位符，但是与所有其他的正则流派都不一样的是，它的 `$` 定位符只会匹配字符串的绝对结束位置（在允许 `^` 和 `$` 匹配换行处的选项没有打开的前提下）。

对于 `\A` 的情形会稍微简单一些。它总是会只匹配一个字符串的开始位置，它的含义在我们讨论的除了 JavaScript（不支持该定位符）之外的所有流派中都是一样的。

虽然存在这么多种令人容易混淆的跨流派的不一致性会带来很多不便，但是使用本书中的正则表达式的好处之一就是你通常不用去担心这些区别。除非喜欢追根究底，否

则一般并不需要了解类似我们刚刚讨论的这样复杂的细节。

## 变体

### 处理奇怪的换行符

前面给出的正则式的支持只限于传统的 MS-DOS/Windows、Unix/Linux/OS X 和老版本的 Mac OS 中的换行序列。然而，你可能偶尔还会遇到一些较不常见的垂直空白字符。下面的正则式会在把匹配限制为不超过 5 行文本的同时，考虑这些额外的字符。

```
\A(?:\r\n?|[\n-\f\x85\x{2028}\x{2029}])?+  
[^-\r\x85\x{2028}\x{2029}]*){0,5}\z  
正则选项: 无  
正则流派: PCRE, Perl
```

```
\A(?:\r\n?|[\n-\f\x85\u2028\u2029])?[^-\r\x85\u2028\u2029]*{0,5}\z  
正则选项: 无  
正则流派: .NET, Java, Ruby
```

```
\A(?:\r\n?|[\n-\f\x85\u2028\u2029])?[^-\r\x85\u2028\u2029]*{0,5}\z  
正则选项: 无  
正则流派: Python
```

```
^(?:\r\n?|[\n-\f\x85\u2028\u2029])?[^-\r\x85\u2028\u2029]*{0,5}$  
正则选项: 无  
正则流派: JavaScript
```

所有这些正则式可以处理在表 4-1 中列出的行分隔符，表中同时列出了它们的 Unicode 位置和名称。

表 4-1 行分隔符

Unicode 序列	等价正则式	名 称	使 用 场 合
U+000D U+000A	\r\n	回车和换行 (CRLF)	Windows 和 MS-DOS 文本文件
U+000A	\n	换行 (LF)	Unix、Linux 和 OS X 文本文件
U+000B	\v	垂直制表符 (VT)	(不常见)
U+000C	\f	换页 (FF)	(不常见)
U+000D	\r	回车 (CR)	Mac OS 文本文件
U+0085	\x85	下一行 (NEL)	IBM 主机文本文件
U+2028	\u2028或\x{2028}	行分隔符	(不常见)
U+2929	\u2029或\x{2029}	段分隔符	(不常见)

## 参见

实例 4.9。

## 4.11 肯定响应的检查

### 问题描述

你想要检查一个配置选项或者命令行响应是否返回了一个肯定取值。你希望在可接受的响应中提供一些灵活性，可以接受 true、t、yes、y、okay、ok 和 1，以及所有相应的大小写组合形式。

### 解决方案

使用一个正则表达式来把所有可接受的形式组合起来，只进行一个简单测试就可以完成这个检查。

### 正则表达式

```
^(?:1|t(?:rue)?|y(?:es)?|ok(?:ay)?)$  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### JavaScript

```
var yes = /^(?:1|t(?:rue)?|y(?:es)?|ok(?:ay)?)$/i;  
if (yes.test(subject)) {  
    alert("Yes");  
} else {  
    alert("No");  
}
```

### 其他编程语言

如果读者希望了解在其他编程语言中如何实现这个正则表达式，请参考实例 3.4 和实例 3.5 中的讲解。

### 讨论

下面的分解中给出了上述正则表达式的每个组成部分。容易读懂的记号组合被显示在了同一行中：

```
^          # 判断字符串的开始位置  
(?:       # 分组但是不捕获...
```

```
1          # 匹配一个字面的 "1"
|
# 或者...
t(?:rue)? # 匹配 "t", 后面可能会跟着 "rue"
|
# 或者...
y(?:es)? # 匹配 "y", 后面可能会跟着 "es"
|
# 或者...
ok(?:ay)? # 匹配 "ok", 后面可能会跟着 "ay"
)
# 非捕获分组的结束
$          # 判断字符串的结束位置
```

正则选项：不区分大小写、宽松排列  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

这个正则式实质上是对 7 个不区分大小写的字面量之一的简单检查。它可以采用很多不同的形式。例如，`^(?:[1ty]|true|yes|ok(?:ay))$` 也是一种完全正确的解答。简单地在所有 7 个取值中进行选择也没有问题，比如采用`^(?:1|t|true|y|yes|ok|okay)$`，然而就性能来说，减少使用管道操作符实现的多选结构的大小，而改用字符类或者可选后缀（使用量词`(?)`）的效率会比较高。在这个例子中，性能的差别可能不会超过几个微秒，但是在脑子里对正则式性能问题有个概念是比较好的。有时候，在不同实现方式之间的区别可能会让你大吃一惊。

所有这些例子都把可能的匹配取值放到了一个非捕获分组中，以便限制多选操作符的作用范围。如果我们略掉了分组结构，而使用了类似`^true|yes$` 的形式，那么正则引擎就会去搜索或者“字符串的开始加上‘true’”或者“‘yes’加上字符串的结束”。`^(?:true|yes)$` 会告诉正则引擎去查找字符串的开始，然后是“true”或者“yes”，再然后是字符串的结束。

## 参见

实例 5.2 和实例 5.3。

## 4.12 社会安全号码的合法性验证

### 问题描述

你想要检查用户输入的文本是不是一个合法的社会安全号码（Social Security number）。

### 解决方案

如果你只是需要保证一个字符串遵守了基本的社会安全号码格式，从而可以排除明显不合法的号码，那么下面的正则表达式提供了一个简单的解决方案。如果你需要更加严格的解答，可以通过社会安全管理局（Social Security Administration）来检查到底该号码是否真的属于一个人，那么请参考在本实例的“参见”小节中给出的链接。

## 正则表达式

```
^(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-  
(?!00)[0-9]{2}-(?!0000)[0-9]{4}$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## Python

```
if re.match(r"^(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-  
(?!00)[0-9]{2}-(?!0000)[0-9]{4}$", sys.argv[1]):  
    print "SSN is valid"  
else:  
    print "SSN is invalid"
```

## 其他编程语言

如果读者希望知道在其他编程语言中如何实现这个正则表达式，那么请参考实例 3.5 中的讲解。

## 讨论

美国的社会安全号码包含 9 个数字，它采用的形式是 *AAA-GG-SSSS*。

- 前三个数字是根据地理区域分配的，被称作是区号（area number）。区号不能是 000 或 666，在本书写作之时，合法的社会安全号码中包含的区号都不大于 772。
- 第 4 位和第 5 位数字被称作是组号（group number），范围可以是 01~99。
- 最后四位是序号（serial number），范围可以是 0001~9999。

这个实例遵守了上面列出的规则。下面我们再对这个正则表达式进行一下分解：

```
^          # 判断字符串的开始位置  
(?!000|666) # 检查这里不能匹配 "000" 或 "666"  
(?:       # 分组但是不捕获...  
  [0-6]     # 匹配 "0" 到 "6" 之间的一个数字  
  [0-9]{2}   # 匹配一个数字刚好两次  
  |         # 或者...  
  7         # 匹配一个字面的 "7"  
(?:       # 分组但是不捕获...  
  [0-6]     # 匹配 "0" 到 "6" 之间的一个数字  
  [0-9]     # 匹配一个数字  
  |         # 或者...  
  7         # 匹配一个字面的 "7"  
  [0-2]     # 匹配 "0" 到 "2" 之间的一个数字  
 )         # 非捕获分组的结束  
 )         # 非捕获分组的结束。  
 -        # 匹配一个字面的 "-"  
(?!00)    # 检查这里不能匹配 "00"
```

```
[0-9]{2}      # 匹配一个数字刚好两次  
-           # 匹配一个字面的 "-"  
(?!.0000)    # 检查这里不能匹配 "0000"  
[0-9]{4}      # 匹配一个数字刚好四次  
$           # 判断字符串的结束位置
```

正则选项：宽松排列  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

除了用来判断字符串开始和结束位置的记号 `<^>` 和 `<$>` 之外，这个正则式可以被分解为由连字符分隔的 3 组数字。第一组是最为复杂的。第二组和第三组会分别简单匹配 2 个或 4 个数字，但是使用一个前置的否定型顺序环视来排除匹配全 0 的可能性。

第一组中的数字要复杂很多，而且比其他两组更难理解，因为它匹配了一个数值范围。首先，它使用否定型顺序环视 `<(?!.000|666)>` 来排除特殊的取值“000”和“666”。接下来需要做的任务是去掉任何大于 772 的数字。

因为正则表达式处理的文本，而不是数字，所以我们就必须要逐个字符来分解数值范围。首先，我们知道可以匹配 0~6 开头的任意三个数字，因为前面的否定型顺序环视已经排除了不合法的数字 000 和 666。这第一个部分很容易使用两个字符类和一个量词来实现：`<[0-6][0-9]{2}>`。因为我们还要为以 7 开头的数字提供一个多选结构，所以我们刚刚构造的模式被放到了一个分组 `<(?:[0-6][0-9]{2})>` 中，从而可以对多选结构操作符的作用范围加以限制。

以 7 开头的数字只允许位于 700~722 之间，所以下面就需要根据第二个数字来进一步划分以 7 开头的数字。如果第二个数字在 0 和 6 之间，那么第三个可以是任意数字。如果第二个数字是 7，那么第三个数字必须在 0 和 2 之间。把这些规则都放到一起，我们就得到了 `<7(?:[0-6][0-9]|7[0-2])>`，这个正则式会匹配数字 7 后面跟着两种不同的第二个和第三个数字的选择。

最后，把它加到第一组数字的外层分组中，你就得到了最后的结果：`<(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))>`。这样你就成功创建了可以匹配 000~772 之间的一个三位数的正则表达式。

## 变体

### 在文档中查找社会安全号码

如果你是从一个更大的文档或输入字符串中查找社会安全号码，那么需要把定位符 `<^>` 和 `<$>` 替换为单词边界。正则表达式引擎会把所有的字母数字字符和下划线都当作单词字符。

```
\b(?:!.000|666) (?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-\b  
(?:!.00)[0-9]{2}-(?!.0000)[0-9]{4}\b
```

正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 参见

社会安全局网站 (<http://www.socialsecurity.gov>) 提供了关于最新分配的区号和组号的详细列表。

位于 <http://www.socialsecurity.gov/employer/ssnv.htm> 的社会安全号码合法性验证服务 (SSNVS) 提供了两种途径，可以通过 Internet 来检查姓名和社会安全号码是否能够匹配社会安全局的记录。

关于如何匹配数值范围的详细讨论，以及如何匹配数字个数可变的范围的例子，请参考实例 6.5。

## 4.13 ISBN 的合法性验证

### 问题描述

你需要检查国际标准书号 (International Standard Book Number, ISBN) 的合法性，它可以是较老的 ISBN-10，或者是当前使用的 ISBN-13 格式。你想要允许出现前导的 ISBN 标识符，而且 ISBN 中可以选择使用连字符或空格分隔。ISBN 978-0-596-52068-7、ISBN-13: 978-0-596-52068-7、978 0 596 52068 7、9780596520687、ISBN-10 0-596-52068-9 和 0-596-52068-9 都是合法输入的例子。

### 解决方案

你无法只使用一个正则式来检查 ISBN，因为最后一个数字是使用校验和 (checksum) 算法计算出来的。这个小节中的正则表达式会检查 ISBN 格式是否合法，而随后的代码实例中会包含对最后一个数字的合法性验证。

### 正则表达式

ISBN-10：

```
^(?:ISBN(?:-10)?:?•)?(?=[-0-9X•]{13}$|[-0-9X]{10}$)[0-9]{1,5}[-•]?  
(?:[0-9]+[-•]?)?{2}[0-9X]$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

ISBN-13：

```
^(?:ISBN(?:-13)?:?•)?(?=[-0-9•]{17}$|[0-9]{13}$97[89][-•]?[0-9]{1,5}  
[-•]?(?:[0-9]+[-•]?)?{2}[0-9]$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

ISBN-10 或者 ISBN-13：

`^(?:ISBN(?:-1[03])?:(?:[-●]?(?:[-●]?[0-9]{1,5}[-●]?|([0-9]{1,5}){2}))|(?=([-●]?[0-9]{1,5}){17}$|[-●]?[0-9X]{13}$|[0-9X]{10}$|97[89][-●]?[0-9]{1,5}[-●]?([0-9]{1,5}){2})[0-9X]$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## JavaScript

```
// 'regex' checks for ISBN-10 or ISBN-13 format
var regex = /^(?:ISBN(?:-1[03])?:(?:[-●]?(?:[-●]?[0-9]{1,5}[-●]?|([0-9]{1,5}){2}))|(?=([-●]?[0-9]{1,5}){17}$|[-●]?[0-9X]{13}$|[0-9X]{10}$|97[89][-●]?[0-9]{1,5}[-●]?([0-9]{1,5}){2})[0-9X]$/;

if (regex.test(subject)) {
    // Remove non ISBN digits, then split into an array
    var chars = subject.replace(/[^0-9X]/g, "").split("");
    // Remove the final ISBN digit from 'chars', and assign it to 'last'
    var last = chars.pop();
    var sum = 0;
    var digit = 10;
    var check;
    if (chars.length == 9) {
        // Compute the ISBN-10 check digit
        for (var i = 0; i < chars.length; i++) {
            sum += digit * parseInt(chars[i], 10);
            digit -= 1;
        }
        check = 11 - (sum % 11);
        if (check == 10) {
            check = "X";
        } else if (check == 11) {
            check = "0";
        }
    } else {
        // Compute the ISBN-13 check digit
        for (var i = 0; i < chars.length; i++) {
            sum += (i % 2 * 2 + 1) * parseInt(chars[i], 10);
        }
        check = 10 - (sum % 10);
        if (check == 10) {
            check = "0";
        }
    }

    if (check == last) {
        alert("Valid ISBN");
    } else {
        alert("Invalid ISBN check digit");
    }
} else {
    alert("Invalid ISBN");
}
```

## Python

```
import re
import sys

# 'regex' checks for ISBN-10 or ISBN-13 format
regex = re.compile("^(?:ISBN(?:-1[03])?|(?:(?=[-0-9 ]{17}$)|[-0-9X ]{13}$|(0-9X){10}$)(?:97[89][- ]?)?[0-9]{1,5}[- ]?)?(?:[0-9]+[- ]?)?{2}[0-9X]$")

subject = sys.argv[1]

if regex.search(subject):
    # Remove non ISBN digits, then split into an array
    chars = re.sub("[^0-9X]", "", subject).split("")
    # Remove the final ISBN digit from 'chars', and assign it to 'last'
    last = chars.pop()
    if len(chars) == 9:
        # Compute the ISBN-10 check digit
        val = sum((x + 2) * int(y) for x,y in enumerate(reversed(chars)))
        check = 11 - (val % 11)
        if check == 10:
            check = "X"
        elif check == 11:
            check = "0"
    else:
        # Compute the ISBN-13 check digit
        val = sum((x % 2 * 2 + 1) * int(y) for x,y in enumerate(chars))
        check = 10 - (val % 10)
        if check == 10:
            check = "0"

    if (str(check) == last):
        print "Valid ISBN"
    else:
        print "Invalid ISBN check digit"
else:
    print "Invalid ISBN"
```

## 其他编程语言

如果读者希望知道在其他编程语言中如何实现这些正则表达式，那么请参考实例 3.5 中的讲解。

## 讨论

ISBN 是用于销售的图书和图书类产品的唯一标识符。10 位的 ISBN 格式是在 1970 年发布的国际标准 ISO 2108。从 2007 年 1 月 1 日开始分配的 ISBN 则都是 13 位的。

ISBN-10 和 ISBN-13 的号码被分别划分为 4 个或 5 个部分。其中 3 个部分是长度可变的，剩余的一个或两个部分则是固定长度的。所有 5 个组成部分通常都是由连字符或空格进行分隔。下面是关于每个部分的简要描述。

- 13 位的 ISBN 以前缀 978 或 979 作为开头。
- 区位代码用来标识使用相同语言的国家分组。它的长度可以是 1~5 个数字。
- 出版社代码的长度是可变的，它由各国的 ISBN 管理机构分配。
- 书序码也是长度可变的，它是由出版社来指定的。
- 最后一个字符是校验码，它可以使用校验和算法得到。ISBN-10 的校验码可以是从 0~9 的一个数字，或者是字母 X(罗马数字 10)，而 ISBN-13 的校验码则只能是 0~9。之所以有这样的区别是因为两种 ISBN 类型采用了不同的校验和算法。

在下面给出了“ISBN-10 或 ISBN-13”的正则表达式的分解形式。因为这个正则式使用了宽松排列模式，在正则式中的字母空格就使用反斜杠进行了转义。Java 中要求即使在字符类之内的空白在宽松排列模式下也应该转义：

```
^          # 判断字符串的开始位置
(?:        # 分组但是不捕获...
  ISBN      # 匹配文本 "ISBN"
  (?:-1[03])? # 匹配 (可选的) 文本 "-10" 或 "-13"
  :?        # 匹配 (可选的) 字面 ":" 
  \         # 匹配一个空白字符 (转义过的)
)?        # 重复该分组 0 到 1 次
(?:=      # 判断这里可以匹配下面的...
  [-0-9\ ]{17}$ # 匹配 17 个连字符、数字和空白，然后是字符串结束
  |
  [-0-9X\ ]{13}$ # 匹配 13 个连字符、数字、X 和空白，然后是字符串结束
  |
  [0-9X]{10}$   # 匹配 10 个字符和 X，然后是字符串结束
)
(?:        # 肯定型顺序环视的结束
  # 分组但是不捕获...
  97[89]    # 匹配文本 "978" 或 "979"
  [-\ ]?    # 匹配 (可选的) 一个连字符或空白
)?
[0-9]{1,5} # 重复该分组 0 次或 1 次
[-\ ]?    # 匹配 (可选的) 一个连字符或空白
(?:        # 分组但是不捕获...
  [0-9]+    # 匹配一个数字 1 到无数次
  [-\ ]?    # 匹配 (可选的) 一个连字符或空白
){2}       # 重复这个分组恰好两次
[0-9X]     # 匹配一个数字或 "X"
$          # 判断字符串的结束位置
```

正则选项：宽松排列

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

前导的 `<(?ISBN(?:-1[03])??:?•)?>` 包含 3 个可选的部分，允许它匹配如下的 7 个字符串之一（除了空字符串之外，其他字符串都会在最后包含一个空格）：

- ISBN•
- ISBN-10•
- ISBN-13•
- ISBN:•
- ISBN-10:•
- ISBN-13:•
- 空字符串（无前缀）

接下来，肯定型顺序环视 `<(?=([-9•]{17}|[-9X•]{13}|[0-9X]{10})$)>` 会匹配剩余匹配的（由多选操作符 `|` 分隔的）三种选择之一，每种都有不同的长度和字符集。所有三种选择（如下所示）都以定位符 `$` 作为结束，这样可以确保不会包含任何不符合这些模式的拖尾文本：

`<[-9•]{17}$>`

允许包含 4 个分隔符的 ISBN-13 代码（总共 17 个字符）

`<[-9X•]{13}$>`

允许不含有分隔符的 ISBN-13 代码，或者包含 3 个分隔符的 ISBN-10 代码（总共 13 个字符）

`<[0-9X]{10}$>`

允许不含有分隔符的 ISBN-10 代码（总共 10 个字符）

在肯定型顺序环视确认了长度和字符集之后，我们就可以不用再担心它们的总长度，从而可以放心匹配 ISBN 的每个组成部分。`<(?97[89][-•]?)>` 会匹配 ISBN-13 中要求的“978”或“979”前缀。这个非捕获分组是可选的，因为在 ISBN-10 格式的目标字符串中就不会匹配到该分组。`<[0-9]{1,5}[-•]?>` 会匹配 1~5 个数字的区位代码和紧跟其后的一个可选的分隔符。`<(?:[0-9]+[-•]?){2}>` 会匹配可变长度的出版社代码和书序码，以及它们相应的可选分隔符。最后，`<[0-9X]$>` 匹配字符串末尾的检验码。

虽然正则表达式可以用来检查最后一位数字使用的是合法字符（数字或 X），但是它不能决定它是不是一个正确的 ISBN 校验和。根据你要处理的是 ISBN-10 或者 ISBN-13，它们会使用两种不同的校验和算法之一，用来确保 ISBN 数字没有被不小心改变次序，或者是输入错误。前面所给的 JavaScript 和 Python 示例代码中实现了这两种算法。下面的小节会讲解校验和的规则，以帮助你在其他语言中实现这些算法。

## ISBN-10 校验和

ISBN-10 代码的校验码范围是 0~10（使用罗马数字 X 来代替 10）。它的计算方法如下：

1. 把前 9 位数字中的每一位分别乘以 10~2（按顺序递减），然后取和；
2. 把上面得到的结果除以 11；

3. 计算 11 减去上面除法结果中余数（不是商）的差；
4. 如果结果是 11，那么使用数字 0；如果是 10，使用字母 X。

下面是一个用来推导 ISBN-10 校验码的例子，假设书号为 0-596-52068-?。

第 1 步：

$$\begin{aligned}\text{加权和} &= 10 \times 0 + 9 \times 5 + 8 \times 9 + 7 \times 6 + 6 \times 5 + 5 \times 2 + 4 \times 0 + 3 \times 6 + 2 \times 8 \\ &= 0 + 45 + 72 + 42 + 30 + 10 + 0 + 18 + 16 \\ &= 233\end{aligned}$$

第 2 步：

$$233 \div 11 = 21, \text{ 余数为 } 2$$

第 3 步：

$$11 - 2 = 9$$

第 4 步：

9 [不需要替换]

计算出来的校验码是 9，所以完整的代码序列是 ISBN 0-596-52068-9。

## ISBN-13 校验和

ISBN-13 代码的校验码范围是 0~9。它的计算方法会采用类似的步骤。

1. 把前 12 位数字中的每一位分别乘以 1 或 3（从左向右依次交替使用 1 和 3），然后取和；
2. 把上面得到的结果除以 10；
3. 计算 10 减去上面除法结果中余数（不是商）的差；
4. 如果是 10，那么使用数字 0。

下面是一个用来推导 ISBN-13 校验码的例子，假设书号为 978-0-596-52068-?。

第 1 步：

$$\begin{aligned}\text{加权和} &= 1 \times 9 + 3 \times 7 + 1 \times 8 + 3 \times 0 + 1 \times 5 + 3 \times 9 + 1 \times 6 + 3 \times 5 + 1 \times 2 + 3 \times 0 + 1 \times 6 + 3 \times 8 \\ &= 9 + 21 + 8 + 0 + 5 + 27 + 6 + 15 + 2 + 0 + 6 + 24 \\ &= 123\end{aligned}$$

第 2 步：

$$123 \div 10 = 12, \text{ 余数为 } 3$$

第 3 步：

$$10 - 3 = 7$$

第 4 步：

7 [不需要替换]

计算出来的校验码是 7，所以完整的序列是 ISBN 978-0-596-52068-7。

## 变体

### 在文档中查找 ISBN

这个版本的“ISBN-10 或者 ISBN-13”的正则式中会使用单词分界符，而不是定

位符来帮助你在更长的文本中查找 ISBN，同时保证它们还是独立存在的。在这个版本中，“ISBN”标识符也变成了一个必需的字符串。这样做有两个原因。首先，有了它之后有助于去除误报（如果没有前导“ISBN”，那么正则式就可能会匹配到任意 10 位或 13 位的数字）；其次，在实际打印 ISBN 的时候，这个标识符是必需的：

```
\bISBN(?:-1[03])?:?•(?:[-0-9•]{17}$|[-0-9X•]{13}$|[0-9X]{10}$)↵
(?:97[89][-•]?)?[0-9]{1,5}[-•]?(?:[0-9]+[-•]?)\b
正则选项: 无
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 排除不正确的 ISBN 标识符

上述正则式的一个缺陷是，它们可以匹配“ISBN-13”标识符后面跟一个 ISBN-10 号码，反过来也一样。下面的正则式会使用正则条件（参考实例 2.17）来保证一个“ISBN-10”或“ISBN-13”标识符之后跟着的是正确的 ISBN 类型。当类型没有明确指定时，它会同时支持 ISBN-10 和 ISBN-13 的号码。在很多情形下，这个正则式都是没有必要的，因为同样的结果完全可以使用我们在前面给出的 ISBN-10 和 ISBN-13 的正则式来分别进行匹配，那样反而会更加容易管理。这里，我们给出这个解答只是为了演示正则表达式的一个有趣实例：

```
^
(?:ISBN(-1(?:0|3))?:?\ )?
(?:1
(?:2
(?:[-0-9X]{13}$|[0-9X]{10}$)
[0-9]{1,5}[- ]?(?:[0-9]+[- ]?)\{2\}[0-9X]$|
(?:[-0-9]{17}$|[0-9]{13}$)
97[89][- ]?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?)\{2\}[0-9]$|
(?:[-0-9]{17}$|[-0-9X]{13}$|[0-9X]{10}$)
(?:97[89][- ]?)?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?)\{2\}[0-9X]$|
)
|
(?:[-0-9]{17}$|[-0-9X]{13}$|[0-9X]{10}$)
(?:97[89][- ]?)?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?)\{2\}[0-9X]$|
)
$
```

正则选项: 宽松排列

正则流派: .NET、PCRE、Perl、Python

## 参见

ISBN 用户手册的最新版本可以在国际 ISBN 管理局的网站 (<http://www.isbn-international.org>) 找到。

官方的区位代码列表 (<http://www.isbn-international.org/en/identifiers/allidentifiers.html>) 会有助于你根据一本书的 ISBN 的前 1~5 个字符识别它是来自哪个国家或者地区。

## 4.14 ZIP 代码的合法性验证

### 问题描述

你需要对 ZIP 代码（美国邮政编码）进行合法性验证，允许 5 位和 9 位（ZIP+4）的格式。你设计的正则式应当可以匹配 12345 和 12345-6789，但是不能匹配 1234、123456、123456789 或 1234-56789。

### 解决方案

#### 正则表达式

```
^[0-9]{5}(?:-[0-9]{4})?$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

#### VB.NET

```
If Regex.IsMatch(subjectString, "^[0-9]{5}(?:-[0-9]{4})?$") Then  
    Console.WriteLine("Valid ZIP code")  
Else  
    Console.WriteLine("Invalid ZIP code")  
End If
```

#### 其他编程语言

如果读者希望了解如何使用其他编程语言来实现这个正则表达式，请参考实例 3.5 中的讲解。

### 讨论

下面是对上述 ZIP 代码正则表达式的分解：

```
^          # 判断字符串的开始位置  
[0-9]{5}   # 匹配一个数字恰好 5 次  
(?:       # 分组但是不捕获...  
-          # 匹配一个字面的 "-"  
[0-9]{4}   # 匹配一个数字恰好 4 次  
)         # 非捕获分组的结束  
?          # 重复之前的分组 0 次或 1 次。  
$          # 判断字符串的结束位置  
正则选项: 宽松排列  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

这个正则式比较容易理解，所以也没有太多要解释的。进行一个简单的改动就可以在更长的输入字符串中来查找 ZIP 代码，只需把定位符 `<^>` 和 `<$>` 替换为单词边界即可，

这样你得到的正则式是 `\b[0-9]{5}(?:-[0-9]{4})?\b`。

## 参见

实例 4.15、实例 4.16 和实例 4.17。

# 4.15 加拿大邮政编码的合法性验证

## 问题描述

你想要检查一个字符串是不是一个加拿大邮政编码。

## 解决方案

```
^(?!.*[DFIOQU])[A-VXY][0-9][A-Z]*[0-9][A-Z][0-9]$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 讨论

在这个正则表达式开头的否定型顺序环视避免了在目标字符串中的任何地方出现 D、F、I、O、Q 或 U。字符类 `[A-VXY]` 进一步防止在第一个字符中会出现 W 或 Z。除了这两个例外之外，加拿大的邮政编码中会使用 6 个字母数字字符的交替序列，中间位置还要添加一个空格。例如，该正则式会匹配 K1A 0B1，这个是加拿大邮局渥太华总部的邮政编码。

## 参见

实例 4.14、实例 4.16 和实例 4.17。

# 4.16 英国邮政编码的合法性验证

## 问题描述

你需要一个正则表达式来匹配一个英国邮政编码。

## 解决方案

```
^[A-Z]{1,2}[0-9R][0-9A-Z]?[0-9][ABD-HJLNP-UW-Z]{2}$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 讨论

英国使用的邮政编码是由 5~7 个字母或数字字符组成，中间由一个空格来分隔。关于

哪些字符可以出现在特定位置的规则是相当复杂的，而且充满了各种例外。这个正则表达式遵守了最基本的规则。

## 参见

英国标准 BS7666 描述了英国邮政编码的规则，它可以在 <http://www.govtalk.gov.uk/gdsc/html/frames/PostCode.htm> 找到。

实例 4.14、实例 4.15 和实例 4.17。

## 4.17 查找使用邮局信箱的地址

### 问题描述

你想要找到所有包含邮局信箱（P.O. box）的地址，并且提醒用户他们的送货地址必须使用街道地址<sup>1</sup>。

### 解决方案

#### 正则表达式

```
^(?:Post(?:Office)?|P[.●]?O\..?●)?Box\b  
正则选项：不区分大小写、^ 和 $ 匹配换行处  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

#### C#

```
Regex regexObj = new Regex(  
    @"^(?:Post(?:Office)?|P[. ]?O\..?)?Box\b",  
    RegexOptions.IgnoreCase | RegexOptions.Multiline  
);  
if (regexObj.IsMatch(subjectString)) {  
    Console.WriteLine("The value does not appear to be a street address");  
} else {  
    Console.WriteLine("Good to go");  
}
```

#### 其他编程语言

如果读者希望了解如何在其他编程语言中实现这个正则表达式，请参考实例 3.5 中的讲解。

<sup>1</sup> 译注：在美国，邮局信箱地址可以用来接收普通信件，但是无法接收稍微大一些的包裹或需要签名的快递等，因此从网站在线购物时一般都要求提供街道地址（Street address，也就是有人可以接收包裹的住址）。

## 讨论

下面的解释使用了宽松排列模式，所以在正则表达式中的每个有意义的空白字符都使用反斜杠进行了转义：

```
^          # 判断字符串的开始位置
(?:        # 分组但是不捕获...
 Post\     # 匹配 "Post"
(?:Office\ )? # 匹配（可选的）"Office"
 |
 P[.\ ]?   # 匹配 "P" 和一个可选的点号或空白字符
 O\.?\\    # 匹配 "O"、一个可选的点号和一个空白字符
)?        # 重复上述分组 0 次或 1 次
Box       # 匹配 "Box"
\b        # 判断单词边界的位置
```

正则选项：不区分大小写、^ 和 \$ 匹配换行处

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

这个正则表达式会匹配所有下列的例子，前提是它们必须出现在一行的起始处：

- Post Office Box
- post box
- P.O. box
- P O Box
- Po. box
- PO Box
- Box

虽然我们在这里已经做了很小心的处理，但是你还是可能会遇到一些误报或漏报，因为许多人都习惯于送货方在解释地址时会比较灵活。为了消除这种风险，最好一开始就明确规定不允许使用邮局信箱作为地址。而如果你使用这个正则表达式发现了一个匹配，要考虑向用户提醒一下他使用的好像是一个邮箱，但是以防万一还可以让用户选择保留该输入。

## 参见

实例 4.14、实例 4.15 和实例 4.16。

## 4.18 转换姓名格式

### 问题描述

你想要把人的姓名顺序从“FirstName LastName”的格式转换为“LastName, FirstName”，这样会方便用于对 LastName 按字母排序的列表中。另外你还需要处理名字的其他部分，例如，你可以把“FirstName MiddleNames Particles LastName Suffix”转换成“LastName,

FirstName MiddleNames Particles Suffix”。

## 解决方案

不幸的是，使用正则表达式是不可能对姓名进行可靠分析的。正则表达式是严格的，而姓名则是非常灵活的，即使人也可能会把它们搞错。要确定一个姓名的结构，或者决定它应该如何按顺序排列，通常需要考虑传统、国家惯例甚至是个人喜好。然而，如果你愿意对数据做一些假设，并且可以接受一定程度的错误率，那么正则表达式可以作为一个快速的解决方案。

作者有意对下面的正则表达式采用了简化处理，而没有去考虑所有的边界情况。

## 正则表达式

```
^(.+?)•([^\s,]+)(,?•(?:[JS]r\.?.?|III?|IV))?$  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 替代文本

```
$2,$1$3  
替代文本流派：.NET、Java、JavaScript、Perl、PHP  
\2,\1\3  
替代文本流派：Python、Ruby
```

## JavaScript

```
function formatName (name) {  
    return name.replace(/^(.+?) ([^\s,]+)(,? (?:[JS]r\.?.?|III?|IV))?$/i,  
                        "$2, $1$3");  
}
```

## 其他编程语言

如果读者希望了解在其他编程语言中如何实现这个正则表达式，请参考实例 3.15 中的讲解。

## 讨论

首先，我们来逐个部分解释一下这个正则表达式。其中提供了注释，以便帮助解释姓名中的哪个部分是由正则式中的某个片段来进行匹配的。因为下面这个正则式采用了宽松排列模式，所以字面的空白字符都使用反斜杠进行了转义：

```
^          # 判断字符串的开始位置  
(          # 把括号内的匹配捕获到向后引用 #1 中...  
.+?       # 匹配一个或多个字符，匹配次数尽量少  
)          # 捕获分组的结束  
\          # 匹配一个字面的空格字符
```

```

(
    # 把括号内的匹配捕获到向后引用 #2 中...
    [^\s,]+      # 匹配不是空格或逗号的一个或多个字符
)
# 捕获分组的结束
(
    # 把括号内的匹配捕获到向后引用 #3 中...
    ,?\          # 匹配 "," 或 ""
    (?:
        [JS]r\..? # 匹配 "Jr", "Jr.", "Sr", 或 "Sr."
        |
        III?       # 匹配 "II" 或者 "III"
        |
        # 或者...
        IV         # 匹配 "IV"
    )
    # 非捕获分组的结束
)?           # 重复该分组 0 次或 1 次
$             # 判断字符串的结束位置

```

正则选项：不区分大小写、宽松排列

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

这个正则表达式会对目标数据做如下假设：

- 它包含至少一个 first name 和一个 last name（其他姓名部分是可选的）；
- first name 应该出现在 last name 之前。
- 如果姓名中包含一个后缀（suffix），那么它应该是逗号之后跟着的下列取值之一，即 “Jr”、“Jr.”、“Sr”、“Sr.”、“II”、“III” 或者 “IV”。

另外还有一些需要考虑的事项。

- 正则表达式无法识别不使用连字符的复姓。例如，Sacha Baron Cohen 会被替换为 Cohen, Sacha Baron，而无法替换为正确的结果 Baron Cohen, Sacha。
- 它无法把保留在姓（family name）之前的附加词（particle），虽然这有时候是根据习俗或个人喜欢来决定的。例如，根据 *Chicago Manual of Style*（《芝加哥论文格式》）第 15 版，“Charles de Gaulle” 在排序时应该被列为“de Gaulle, Charles”，然而这和《韦伯斯特传记词典》中的规定是相矛盾的。
- 因为定位符 `^` 和 `\$` 会把匹配定位到字符串的开始和结束，所以如果整个目标文本不能符合这个模式，那么就不会进行任何匹配。因此，如果没有找到合适的匹配（例如，如果目标文本只包含一个名字，即使用一个单词作为其姓名），那么该名字就会照原样返回。

下面来解释上述正则表达式是如何工作的，它使用了三个捕获分组来拆分姓名。然后会在替代字符串中使用向后引用来重新组合这些片段。捕获分组 1 使用了最为灵活的 `.+?` 模式来抓取 first name、任意个数的 middle name 以及姓的附加词，比如德语的“von”，或者法语、葡萄牙语和西班牙语中的“de”。这些姓名组成部分会被放在一起处理，因为它们会在输出中按原来的顺序列出。把 first name 和 middle name 放在一起也可以避免出错，因为正则表达式无法区分到底是复合的 first name，还是一个 first

name 加上 middle name，例如对于“Mary Lou”或“Norma Jeane”。即使人也无法只是通过肉眼观察就能准确做出区分。

捕获分组 2 会使用 `\[^s,]+` 来匹配 last name。同在捕获分组 1 中使用的点号一样，这个字符类的灵活性允许它匹配带读音符号的字符，以及任意其他非拉丁语字符。捕获分组 3 会从一个预定义的取值列表中匹配一个可选的后缀，例如“Jr.”或“III”。这个后缀会同 last name 分别进行处理，因为它应当出现在重新格式化之后的姓名的末尾。

我们再回来看一下捕获分组 1。为什么在分组 1 中的点号之后跟着的是懒惰的 `+?` 量词，而在分组 2 中的字符类之后则跟的是贪心的 `+` 量词呢？如果分组 1（它会处理可变个数的元素，因此需要尽量匹配姓名中的更多部分）使用了贪心量词，那么分组 3（它会尝试匹配一个后缀）就不可能参与到匹配中来。分组 1 中的点号会一直匹配到字符串的结束，然后因为捕获分组 3 是可选的，所以正则引擎就只会回溯到为分组 2 找到一个匹配，然后就会宣布匹配成功。捕获分组 2 可以使用贪心量词，因为它的字符类中包含更多限制，它只能匹配一个名字。

表 4-2 给出了一些例子，展示使用这个正则表达式和替代字符串对姓名进行格式化处理之后的结果。

表 4-2 姓名的格式化

输入	输出
Robert Downey, Jr.	Downey, Robert, Jr.
John F. Kennedy	Kennedy, John F.
Scarlett O'Hara	O'Hara, Scarlett
Pepé Le Pew	Pew, Pepé Le
J.R.R. Tolkien	Tolkien, J.R.R.
Catherine Zeta-Jones	Zeta-Jones, Catherine

## 变体

### 把姓附加词列在姓名的开头

在下面的正则表达式中添加了一个片段，允许你把一个预定义列表中给定的姓附加词输出到 last name 的前面。具体来说，这个正则表达式会考虑的附加词包括“De”、“Du”、“La”、“Le”、“St”、“St.”、“Ste”、“Ste.”、“Van”和“Von”。在姓名序列中会允许这些值出现任意多次（例如，“de la”）：

```
^(.+?)•((?:(:D[eu]|L[ae]|Ste?\..?|V[ao]n)•)*[^s,]+)+  
(,?•(?:JS)r\..?|III?|IV))?$  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

\$2,•\$1\$3

替代文本流派: .NET、Java、JavaScript、Perl、PHP

\2,•\1\3

替代文本流派: Python、Ruby

## 4.19 信用卡号码的合法性验证

### 问题描述

你有个任务，要给一个公司实现一个接受信用卡付账的订购表单。由于信用卡处理中心对每次交易尝试都要收费，其中也包括失败的尝试在内，所以你想要使用一个正则表达式来先排除掉明显不合法的信用卡号码。

这样做同样会改善客户的使用体验。在客户完成网页上的输入之后，正则式可以在瞬间检查出比较明显的错误。相比来说，如果要访问信用卡处理中心，那么一个来回可能需要花费 10~30 秒钟。

### 解决方案

#### 去掉空格和连字符

先获取用户输入的信用卡号码并把它保存到一个变量中。在执行合法号码检查之前，先执行查找和替换来去掉其中的空白和连字符。这可以通过把下面这个正则表达式全局替换为空白的替换文本来完成：

[•-]

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

实例 3.14 中讲解了如何来用这个正则式执行替换。

#### 检查号码

在从输入中去掉空格和连字符之后，下面这个正则表达式会检查该信用卡号码是否使用了 6 个主要信用卡公司使用的任意一种格式。它同时还使用了命名捕获来检查该客户使用的是哪个品牌的信用卡：

```
^(?:  
    (?<visa>4[0-9]{12}(?:[0-9]{3})?) |  
    (?<mastercard>5[1-5][0-9]{14}) |  
    (?<discover>6(?:011|5[0-9]{2}[0-9])[0-9]{12}) |  
    (?<amex>3[47][0-9]{13}) |  
    (?<diners>3(?:0[0-5]|68)[0-9]{11}) |  
    (?<jcb>(?:2131|1800|35\d{3})\d{11})  
)$
```

正则选项：宽松排列

正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

```
^(?:  
    (?P<visa>4[0-9]{12}(?:[0-9]{3})?) |  
    (?P<mastercard>5[1-5][0-9]{14}) |  
    (?P<discover>6(?:011|5[0-9][0-9])[0-9]{12}) |  
    (?P<amex>3[47][0-9]{13}) |  
    (?P<diners>3(?:0[0-5]|68)[0-9]{11}) |  
    (?P<jcb>(?:2131|1800|35\d{3})\d{11})  
)$
```

正则选项：宽松排列

正则流派：PCRE、Python

Java、Perl 5.6、Perl 5.8 和 Ruby 1.8 不支持命名捕获。因此你可以使用编号捕获作为替代。分组 1 中会捕获 Visa 信用卡号，分组 2 中捕获 MasterCard，以此类推，分组 6 会捕获 JCB 信用卡：

```
^(?:  
    (4[0-9]{12}(?:[0-9]{3})?) # Visa  
    (5[1-5][0-9]{14}) # MasterCard  
    (6(?:011|5[0-9][0-9])[0-9]{12}) # Discover  
    (3[47][0-9]{13}) # AMEX  
    (3(?:0[0-5]|68)[0-9]{11}) # Diners Club  
    ((?:2131|1800|35\d{3})\d{11}) # JCB  
)$
```

正则选项：宽松排列

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

JavaScript 不支持宽松排列。在把空白和注释去掉之后，我们会得到如下的正则表达式：

```
^(?: (4[0-9]{12}(?:[0-9]{3})?) | (5[1-5][0-9]{14}) |  
       (6(?:011|5[0-9][0-9])[0-9]{12}) | (3[47][0-9]{13}) |  
       (3(?:0[0-5]|68)[0-9]{11}) | ((?:2131|1800|35\d{3})\d{11}))$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

如果你不需要决定使用的是哪种信用卡，那么可以去掉不必要的捕获分组：

```
^(?:  
    4[0-9]{12}(?:[0-9]{3})? | # Visa  
    5[1-5][0-9]{14} | # MasterCard  
    6(?:011|5[0-9][0-9])[0-9]{12} | # Discover  
    3[47][0-9]{13} | # AMEX  
    3(?:0[0-5]|68)[0-9]{11} | # Diners Club  
    (?:2131|1800|35\d{3})\d{11} # JCB  
)$
```

正则选项：宽松排列

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

这是 JavaScript 的版本：

```
^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|6(?:011|5[0-9])[0-9]{12})|+  
3[47][0-9]{13}|3(?:0[0-5]|68)[0-9]{11}|(?:2131|1800|35\d{3})\d{11})$  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

按照实例 3.6 中所给的代码示例，可以把这个正则表达式添加到你的订购表单中来验证信用卡号码。如果你对于不同卡会使用不同的处理中心，或者只是想记录一些统计数据，那么可以使用实例 3.9 来检查匹配位于哪个命名或编号的捕获分组中。这会告诉你客户使用的是哪种品牌的信用卡。

## 使用 JavaScript 的网页示例

```
<html>  
<head>  
<title>Credit Card Test</title>  
</head>  
<body>  
<h1>Credit Card Test</h1>  
  
<form>  
<p>Please enter your credit card number:</p>  
  
<p><input type="text" size="20" name="cardnumber"  
onkeyup="validatecardnumber(this.value)"></p>  
  
<p id="notice">(no card number entered)</p>  
</form>  
  
<script>  
function validatecardnumber(cardnumber) {  
    // Strip spaces and dashes  
    cardnumber = cardnumber.replace(/[-]/g, '');  
    // See if the card is valid  
    // The regex will capture the number in one of the capturing groups  
    var match = /^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14})|+  
    (6(?:011|5[0-9])[0-9]{12})|(3[47][0-9]{13})|(3(?:0[0-5]|68)[0-9])|  
    [0-9]{11})|((?:2131|1800|35\d{3})\d{11}))$/.exec(cardnumber);  
    if (match) {  
        // List of card types, in the same order as the regex capturing groups  
        var types = ['Visa', 'MasterCard', 'Discover', 'American Express',  
                    'Diners Club', 'JCB'];  
        // Find the capturing group that matched  
        // Skip the zeroth element of the match array (the overall match)  
        for (var i = 1; i < match.length; i++) {  
            if (match[i]) {  
                // Display the card type for that group  
                document.getElementById('notice').innerHTML = types[i - 1];  
                break;  
            }  
        }  
    }  
}
```

```
        }
    } else {
    document.getElementById('notice').innerHTML = '(invalid card number)';
}
}
</script>
</body>
</html>
```

## 讨论

### 去掉空格和连字符

在一张实际的信用卡之上，突出的卡号通常会按照 4 个一组来进行排列。这样可以使人在读卡号的时候更加容易。很自然的，许多人就会试图在订购表单中按照相同的方式来输入卡号，并且把空格也包含进来。

构造一个正则表达式来检查支持空白、连字符和不知道什么奇怪符号的信用卡号码，比构造一个正则表达式来检查只包含数字的卡号要复杂很多倍。因此，除非你想要反复提醒用户重新输入不含空格和连字符的卡号，那么应该先用一个快速的查找和替换来去掉这些字符，然后再检查卡号并把它送到信用卡处理中心。

正则表达式 `<[•-]>` 会匹配一个空格或连字符。把这个正则表达式的所有匹配都替换为空，实际上就会删除所有的空白和连字符。

信用卡号码中只可以包含数字。除了使用 `<[•-]>` 去掉空格和连字符之外，你也可以使用简写字符类 `\D` 来去掉所有的非数字。

### 检查号码

每个信用卡公司都会使用不同的号码格式。我们可以利用这种区别来允许用户直接输入号码，而不必指定一个公司；公司可以通过号码来确定。下面列出了每个公司的号码格式：

Visa

13 位或 16 位数字，以 4 开头。

MasterCard

16 位数字，以 51~55 开头。

Discover

16 位数字，以 6011 或者 65 开头。

American Express

15 位数字，以 34 或 37 开头。



## Diners Club

14 位数字，以 300~305、36 或 38 开头。

## JCB

15 位数字，以 2131 或 1800 开头；或者 16 位数字，以 35 开头。

如果你只接受某些种类的信用卡，那么你可以从正则表达式中删除你不接受的信用卡种类。当删除 JCB 的时候，一定要同时删掉正则表达式最后剩下的 `\b`。如果在正则表达式中包含了 `\|\b` 或者 `\|\b`，那么你就会把空串作为合法卡号接受。

举例来说，如果只接受 Visa、MasterCard 和 AMEX，那么你可以使用：

```
^(?:  
    4[0-9]{12}(?:[0-9]{3})? |          # Visa  
    5[1-5][0-9]{14} |                  # MasterCard  
    3[47][0-9]{13}                   # AMEX  
)$
```

正则选项：宽松排列

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

或者：

```
^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|3[47][0-9]{13})$
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

如果你需要在一个更大的文本中查找信用卡号码，那么就需要把定位符替换为单词边界 `\b`。

## 在网页中使用本实例

前面“使用 JavaScript 的网页示例”小节中的代码展示了如何把这两个正则表达式添加到你的订购表单中。信用卡号码输入框拥有一个 `onkeyup` 事件处理器，它会调用 `validatecardnumber()` 函数。这个函数会从输入框中获取卡号，去掉空白和连字符，然后使用包含编号捕获分组的正则表达式来对它进行检查。检查的结果会通过替换网页上最后一个段落中的文本来显示给用户。

如果正则表达式没有产生匹配，`regexp.exec()` 会返回 `null`，并且显示 `(invalid card number)`。如果正则表达式匹配成功，`regexp.exec()` 会返回一个字符串数组。其中第 0 个元素中保存的是整个匹配。元素 1~6 会分别保存 6 个捕获分组匹配到的文本。

我们的正则表达式中包含 6 个捕获分组，它们之间由多选操作符分开。这意味着只有一个捕获分组会参与匹配，并且会包含匹配的卡号。其他分组都是空的（根据你的浏览器不同，或者是 `undefined`，或者是空字符串）。这个函数会逐个检查 6 个捕获分组。当它找到非空的分组的时候，就识别到了卡号，并且会将它显示给用户。

## 使用 Luhn 算法做进一步验证

在处理订单之前，你还可以对信用卡号码做进一步的合法性验证。信用卡号的最后一个数字是根据 Luhn 算法来计算的校验和。因为该算法需要基本的算术运算，所以你无法在正则表达式中实现它。

你可以把 Luhn 检查添加到这个实例中的网页示例中，在 validatecardnumber() 函数之前调用 luhn(cardnumber)。这样，就只有在正则表达式找到合法匹配，并且确定了信用卡类型之后，才会执行 Luhn 检查。然而，在做 Luhn 检查的时候并不一定非要知道信用卡的类型。所有信用卡都会使用相同的方法来计算。

在 JavaScript 中，你可以按照下面的方式来实现 Luhn 函数：

```
function luhn(cardnumber) {
    // Build an array with the digits in the card number
    var getdigits = /\d/g;
    var digits = [];
    while (match = getdigits.exec(cardnumber)) {
        digits.push(parseInt(match[0], 10));
    }
    // Run the Luhn algorithm on the array
    var sum = 0;
    var alt = false;
    for (var i = digits.length - 1; i >= 0; i--) {
        if (alt) {
            digits[i] *= 2;
            if (digits[i] > 9) {
                digits[i] -= 9;
            }
        }
        sum += digits[i];
        alt = !alt;
    }
    // Card number turns out to be invalid anyway
    if (sum % 10 == 0) {
        document.getElementById("notice").innerHTML += 'Luhn check passed';
    } else {
        document.getElementById("notice").innerHTML += 'Luhn check failed';
    }
}
```

这个函数会接受包含信用卡号码的一个字符串作为参数。卡号应当只包含数字。在我们的例子中，validatecardnumber()已经去掉了空格和连字符，并且确定卡号中包含了正确的数字个数。

首先，这个函数使用正则表达式 `\d` 来遍历字符串中的所有数字。注意这里使用了 `/g` 修饰符。在循环中，`match[0]` 会获取匹配的数字。由于正则表达式只能处理文本（字符串），

所以我们调用了 `parseInt()` 来确保变量中存储的是一个整数，而不是一个字符串。如果不这样做，那么 `sum` 变量就可能会得到所有数字的字符串连接，而不是数字的整数加法的结果。

实际的算法是在数组之上计算一个校验和。如果结果除以 10 余数为 0，那么卡号是合法的。否则，号码就是不合法的。

## 4.20 欧盟增值税代码

### 问题描述

你接到一个任务，需要为欧盟的一家企业实现一个在线订购表单。

欧盟税收法规定，当一个欧盟国家的 VAT 注册企业（你的客户）从另一个欧盟国家的卖方（你的公司）购买产品的时候，卖方不得征收 VAT（增值税）。如果买家不是 VAT 注册的，那么卖方就必须征收 VAT，并把 VAT 上交给当地税务机关。卖方必须提供买方的 VAT 注册代码给税务机关，以证明它并没有拖欠任何 VAT。这意味着对于卖方来说，在处理免税销售之前，就有必要对买方的 VAT 代码进行验证。

VAT 代码最常见的错误原因可能是由于客户不小心敲错了。要使订购过程更加迅速和友好，你就应当在客户填完在线订购表单之后，立即使用一个正则表达式来对 VAT 代码进行检查。你可以使用客户端的 JavaScript，或者在网站服务器之上的 CGI 脚本中来做这样的检查。如果号码并不能匹配正则表达式，那么客户就可以立即对它进行修改。

### 解决方案

#### 去掉空白和连字符

获取客户输入的 VAT 代码，并把它保存到一个变量中。在检查它是否是合法代码之前，先执行一个查找和替换操作，把下面的正则表达式全局替换为空的替代文本：

[ - . • ]

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

实例 3.14 中讲解了如何使用这个正则式执行替换。我们假设客户并不会输入除了连字符、点号和空格之外的其他标点。如果存在其他错误输入的字符，那么会在随后的检查中发现。

#### 检查代码

在从输入中去掉空格和连字符之后，下面这个正则表达式会检查这个 VAT 代码是否属于任意的 27 个欧盟国家之一所使用的合法代码：

```

^(
(AT)?U[0-9]{8} | # Austria
(BE)?0?[0-9]{9} | # Belgium
(BG)?[0-9]{9,10} | # Bulgaria
(CY)?[0-9]{8}L | # Cyprus
(CZ)?[0-9]{8,10} | # Czech Republic
(DE)?[0-9]{9} | # Germany
(DK)?[0-9]{8} | # Denmark
(EE)?[0-9]{9} | # Estonia
(EL|GR)?[0-9]{9} | # Greece
(ES)?[0-9A-Z][0-9]{7}[0-9A-Z] | # Spain
(FI)?[0-9]{8} | # Finland
(FR)?[0-9A-Z]{2}[0-9]{9} | # France
(GB)?([0-9]{9})([0-9]{3})?|[A-Z]{2}[0-9]{3}) | # United Kingdom
(HU)?[0-9]{8} | # Hungary
(IE)?[0-9]S[0-9]{5}L | # Ireland
(IT)?[0-9]{11} | # Italy
(LT)?([0-9]{9}|[0-9]{12}) | # Lithuania
(LU)?[0-9]{8} | # Luxembourg
(LV)?[0-9]{11} | # Latvia
(MT)?[0-9]{8} | # Malta
(NL)?[0-9]{9}B[0-9]{2} | # Netherlands
(PL)?[0-9]{10} | # Poland
(PT)?[0-9]{9} | # Portugal
(RO)?[0-9]{2,10} | # Romania
(SE)?[0-9]{12} | # Sweden
(SI)?[0-9]{8} | # Slovenia
(SK)?[0-9]{10} | # Slovakia
)${}

```

正则选项：宽松排列、不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

这个正则表达式使用了宽松排列模式，这样可以很容易在之后对正则表达式进行修改。新的国家还会不时地加入到欧盟中来，而且欧盟成员国也会修改 VAT 代码的规则。不幸的是，JavaScript 不支持宽松排列。在这种情况下，你就不得不把所有内容都放到一行中：

```

^((AT)?U[0-9]{8}|(BE)?0?[0-9]{9}|(BG)?[0-9]{9,10}|(CY)?[0-9]{8}L|←
(CZ)?[0-9]{8,10}|(DE)?[0-9]{9}|(DK)?[0-9]{8}|(EE)?[0-9]{9}|←
(EL|GR)?[0-9]{9}|(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]|(FI)?[0-9]{8}|←
(FR)?[0-9A-Z]{2}[0-9]{9}|(GB)?([0-9]{9})([0-9]{3})?|[A-Z]{2}[0-9]{3})|←
(HU)?[0-9]{8}|(IE)?[0-9]S[0-9]{5}L|(IT)?[0-9]{11}|←
(LT)?([0-9]{9}|[0-9]{12})|(LU)?[0-9]{8}|(LV)?[0-9]{11}|(MT)?[0-9]{8}|←
(NL)?[0-9]{9}B[0-9]{2}|(PL)?[0-9]{10}|(PT)?[0-9]{9}|(RO)?[0-9]{2,10})|←
(SE)?[0-9]{12}|(SI)?[0-9]{8}|(SK)?[0-9]{10})$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

按照实例 3.6 中所给的代码示例，可以把这个正则表达式添加到你的订购表单中。

## 讨论

### 去掉空格和连字符

为了使 VAT 代码更加易读，人们通常会在其中添加一些额外的标点符号把数字分成不同的组。例如，一个德国客户可能会把它的 VAT 号码 DE123456789 写成 DE 123.456.789。

使用单个正则表达式匹配来自 27 个国家的 VAT 代码的任何可能格式是一个不可能完成的任务。因为标点符号只是为了可读的目的，所以我们可以先去掉所有标点符号，然后再验证得到的 VAT 代码，这样会更加容易。

正则表达式 `<[•.-]>` 会匹配一个空格、点号或连字符。把这个正则表达式的所有匹配都替换为空，就会实际上删除 VAT 代码中所有常用的标点符号。

VAT 代码中只可以包含字母和数字。除了使用 `<[•.-]>` 去掉常用标点之外，你也可以使用简写字符类 `<[^A-Z0-9]>` 来去掉所有的非法字符。

### 检查代码

用来检查代码的两个正则表达式是完全相同的。唯一的区别是第一个使用了宽松排列的语法，这样可以使正则表达式更加易读，并且可以在其中添加国家说明。JavaScript 中不支持宽松排列模式，但是其他流派则会提供这样的选择。

前面给的正则式使用分组结构来匹配所有 27 个欧盟国家的 VAT 代码。它们的基本格式如下所示：

奥地利

U99999999

比利时

999999999 或 0999999999

保加利亚

999999999 或 9999999999

塞浦路斯

99999999L

捷克共和国

99999999、99999999 或 9999999999

德国

999999999

丹麦

99999999

爱沙尼亚

999999999

希腊

999999999

西班牙

X9999999X

芬兰

99999999

法国

XX999999999

英国

99999999、9999999999 或 XX999

匈牙利

99999999

爱尔兰

9S99999L

意大利

9999999999

立陶宛

99999999 或 9999999999

卢森堡

99999999

拉脱维亚

9999999999

马耳他

99999999

荷兰

999999999B99

波兰

999999999

葡萄牙

999999999

罗马尼亚

99、999、9999、99999、999999、9999999、99999999、999999999 或 9999999999

瑞典

99999999999

斯洛文尼亚

99999999

斯洛伐克

999999999

严格来讲，两个字母的国家代码也是属于 VAT 代码的一部分。然而，人们通常会把它略去，因为通过账单地址通常已经可以分辨出是哪个国家。这个正则表达式可以接受包含或者不包含国家代码的 VAT 代码。如果想要把国家代码变成必需的，那么你可以去掉正则表达式中的所有问号。如果你这样做，那么就要记住在提供给用户的错误消息中说明你要求必须使用国家代码。

如果你只接受来自某些特定国家的订单，那么可以去掉一些不会出现在订购表单的国家选择列表中的国家代码。当你删除一个选择分支的时候，一定记住同时删除分隔不同选择分支的管道操作符`|`。否则，你就可能会在正则表达式中出现`|`。`|`会添加一个匹配空字符串的选择分支，这样就意味着你的订购表单会接受没有输入的 VAT 代码作为合法的 VAT 代码。

这 27 个选择分支都被放到了一个分组中。这个分组被放到了一个脱字符和一个美元符号之间，这样就把正则表达式定位到了目标字符串的开始和结束位置。因此整个输入必须被确认为一个合法的 VAT 代码。

如果你要在更大的文本中搜索 VAT 代码，那么就需要把定位符替换成单词边界`\b`。

## 变体

使用一个正则表达式来检查所有 27 个国家代码的好处是，你只需要在订购表单中

添加一个正则表达式验证。你也可以使用 27 个分别的正则表达式来改进订购表单。首先，检查客户在账单地址中所列出的国家。然后，根据国家来查找合适的正则表达式：

奥地利

```
<^(AT)?[0-9]{8}$>
```

比利时

```
<^(BE)?[0-9]{9}$>
```

保加利亚

```
<^(BG)?[0-9]{9,10}$>
```

塞浦路斯

```
<^(CY)?[0-9]{8}L$>
```

捷克共和国

```
<^(CZ)?[0-9]{8,10}$>
```

德国

```
<^(DE)?[0-9]{9}$>
```

丹麦

```
<^(DK)?[0-9]{8}$>
```

爱沙尼亚

```
<^(EE)?[0-9]{9}$>
```

希腊

```
<^(EL|GR)?[0-9]{9}$>
```

西班牙

```
<^(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]$>
```

芬兰

```
<^(FI)?[0-9]{8}$>
```

法国

```
<^(FR)?[0-9A-Z]{2}[0-9]{9}$>
```

英国

```
<^(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3})$>
```

匈牙利

```
<^(HU)?[0-9]{8}$>
```

爱尔兰

```
<^(IE)?[0-9]S[0-9]{5}L$>
```

意大利

```
<^(IT)?[0-9]{11}$>
```

立陶宛

```
<^(LT)?([0-9]{9}|[0-9]{12})$>
```

卢森堡

```
<^(LU)?[0-9]{8}$>
```

拉脱维亚

```
<^(LV)?[0-9]{11}$>
```

马耳他

```
<^(MT)?[0-9]{8}$>
```

荷兰

```
<^(NL)?[0-9]{9}B[0-9]{2}$>
```

波兰

```
<^(PL)?[0-9]{10}$>
```

葡萄牙

```
<^(PT)?[0-9]{9}$>
```

罗马尼亚

```
<^(RO)?[0-9]{2,10}$>
```

瑞典

```
<^(SE)?[0-9]{12}$>
```

斯洛文尼亚

```
<^(SI)?[0-9]{8}$>
```

斯洛伐克

```
<^(SK)?[0-9]{10}$>
```

按照实例 3.6 在代码中可以实现用选定的正则表达式来验证 VAT 代码。这会告诉你该代码对于客户声称他所在的国家是否合法。

使用单独的正则表达式的主要好处是你可以要求 VAT 代码必须以正确的国家代码开头，而不用要求客户自己来输入国家代码。当正则表达式成功匹配所给的号码的时候，接下来需要检查第一个捕获分组的内容。在实例 3.9 中的代码示例讲解了该如何做。

如果第一个捕获分组是空的，那么客户并没有在 VAT 代码的开头输入国家代码。这样你可以在把验证过的代码添加到订单数据中之前，把国家代码添上。

希腊的 VAT 代码支持两个国家代码。EL 传统上被用于希腊的 VAT 代码，但是 GR 是 ISO 标准中的希腊国家代码。

## 参见

本节中的正则表达式只会检查代码看起来是不是一个合法的 VAT 代码。这对于清除无心的过错就足够了。显然我们无法使用正则表达式来检查 VAT 代码是不是分配给了下订单的那个企业。欧盟提供了一个网页 [http://ec.europa.eu/taxation\\_customs/vies/vieshome.do](http://ec.europa.eu/taxation_customs/vies/vieshome.do)，你可以使用它来检查某个特定的 VAT 代码属于哪家企业。

关于在本节的正则表达式中使用的技巧，请参考实例 2.3、实例 2.5 和实例 2.8。

# 单词、文本行和特殊字符

本章中的实例会讲解如何在各种不同上下文中查找和处理文本。其中一些实例会展示如何完成一些高级搜索引擎中的任务，例如查找几个单词中的任意一个，或者查找位置临近的单词。其他例子则会帮助你查找包含特定单词的整行内容，删除重复单词，或者对正则表达式中的元字符进行转义。

本章的中心议题是展示各种不同的正则表达式结构，以及在实际应用中会用到的技巧。阅读本章的过程就好像是使用大量正则表达式语法特性的一次热身练习，它将会有助于你应用正则表达式来解决今后遇到的实际问题。在本章的许多例子中，虽然要查找的内容都比较简单，但是我们在解答中提供的模板则可以通过定制来解决你遇到的具体问题。

## 5.1 查找一个特定单词

### 问题描述

你接到一个简单的任务，要查找单词“cat”出现的所有地方，并且不区分大小写。这里的关键点是它必须以整个单词的形式出现。在结果中不能包括它作为更长单词的一部分的情形，例如 hellcat、application 或 Catwoman。

### 解决方案

使用单词边界记号就可以很容易地解决这个问题：

```
\bcat\b
正则选项：不区分大小写
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

实例 3.7 中会讲解如何使用这个正则表达式来找到所有的匹配。实例 3.14 中会讲解如何使用其他文本来替换匹配。

## 讨论

这个正则表达式两端使用了单词边界，以保证只有当 cat 以整个单词的形式出现时，才会产生匹配。更精确地来说，单词边界要求在 cat 与其他文本之间必须由字符串的开始或结束、空格、标点符号或者其他非单词字符分隔开。

正则表达式引擎会把字母、数字和下划线都当作是单词字符。关于单词边界的详细讨论，请参考实例 2.6。

如果使用这个正则表达式在 JavaScript、PCRE 和 Ruby 中处理国际化文本时，就可能会遇到一些问题，这是因为这些正则表达式流派中，只有位于 ASCII 字符表中的字母才会创建单词边界。换句话说，单词边界只会匹配到 `\b[^A-Za-z0-9_]` 和 `[A-Za-z0-9_]` 之间，或者 `[A-Za-z0-9_]` 和 `[^A-Za-z0-9_]\$` 之间的位置。在 Python 中，在没有设置 UNICODE 或 U 选项的时候，也是如此。这样就会使 `\b` 无法用于在包含带读音符号的字母或使用非拉丁字母表的单词的文本中进行“全字匹配”的查找。举例来说，在 JavaScript、PCRE 和 Ruby 中，`\büber\b` 会在 `darüber` 中找到一个匹配，但是在 `dar über` 中却无法找到匹配。在大多数情况下，这刚好与读者的期望相反。问题的根源在于 ü 被当作一个非单词字符，因此在 `rü` 两个字符之间就找到了一个单词边界。而在空格字符和 ü 之间却无法找到单词边界，这是因为它们被当作是非单词字符的一个连续序列。

你可以通过不采用单词边界，而是使用顺序环视和逆序环视（两个加起来统称为环视）解决这个问题。与单词边界一样，环视也会匹配一个宽度为 0 的位置。在 PCRE（使用 UTF-8 支持进行编译）和 Ruby 1.9 中，你可以使用环视来模拟基于 Unicode 的单词边界，例如可以使用 `\b(?<=\P{L})\bcat(?=\P{L})\b`。这个正则表达式中使用了在实例 2.7 中讨论过的否定型的 Unicode Letter 属性记号 (`\P{L}`)。如果要了解更多关于环视的讲解，请参考实例 2.16。如果你想要让环视也把数字和下划线当作单词字符（就像 `\b` 一样），那么可以把 `\P{L}` 的两次出现都替换成 `\p{L}\p{N}`。

JavaScript 和 Ruby 1.8 既不支持逆序环视，也不支持 Unicode 属性。对于不支持逆序环视的问题，你可以通过下面的方式来解决：先匹配每个匹配之前的非单词字符，然后再把它从匹配中删除，或者在替换匹配的时候放回到替代字符串中去（关于在替代字符串中如何使用匹配到的子串，请参考实例 3.15 中的示例）。至于不支持 Unicode 属性的问题（再加上事实上这两种编程语言中的记号 `\w` 和 `\W` 都只能匹配 ASCII 字符），这就意味着你不得不采用一种更为严格的解决方案。在字母类别中的代码点散布于 Unicode 字符集中，因此如果要想使用 Unicode 转义序列和字符类范围来完全模拟 `\p{L}`，那么就需要上千个字符。作为一种较好的妥协方案，我们可以使用如下的正则表达式：`[A-Za-z\xAA\xB5\xBA\xC0-\xD6\xD8-\xF6\xF8-\xFF]`，它会匹配位于 8 比特地址空间（也就是前 256 个 Unicode 代码点，它们的位置是 `0x00~0xFF`）中所有的 Unicode 字母字符。这个字符类会允许你匹配位于 7 比特 ASCII 地址空间之

外的各种常用的带读音符号的字符，或者也可以使用它的否定形式来排除这些字符。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
i																
4	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
5	P	Q	R	S	T	U	V	W	X	Y	Z					
6	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
7	p	q	r	s	t	u	v	w	x	y	z					
i																
A																
B																
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ì	Í	Ï	Ý	
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ì	í	ï	ý	
F	ð	ñ	ò	ó	ô	õ	ö	ø	ù	ú	û	ü	ý	þ	ÿ	

图 5-1 位于 8 比特地址空间中的 Unicode 字母字符

下面是一个 JavaScript 的例子，用来把所有出现的单词“cat”都替换为“dog”。它能够正确处理常见的包含读音符号的字符，所以并不会对 écat 进行修改。要完成这样的任务，你就需要构造自己的字符类，而不能采用 JavaScript 提供的 `\b` 或者 `\w`：

```
// 8-bit-wide letter characters
var L = 'A-Za-z\xAA\xB5\xBA\xC0-\xD6\xD8-\xF6\xF8-\xFF';
var pattern = '([^\{L\}]|^)cat([^\{L\}]|\$)'.replace(/\{L\}/g, L);
var regex = new RegExp(pattern, 'gi');

// replace cat with dog, and put back any
// additional matched characters
subject = subject.replace(regex, '$1dog$2');
```

需要注意的是，JavaScript 字符串字面量中可以使用 `\xHH`（其中 `HH` 是两位的十六进制数）来插入特殊字符。因此，传递给正则表达式的变量 `L` 中实际上会包含该字符的字面版本。如果想要把元字符序列 `\xHH` 传递给正则表达式，那么你就需要在字符串字面量中对反斜杠进行转义（例如，“`\\\xHH`”）。然而，这样做在这个具体例子中并不会产生任何作用，也不会改变正则表达式的匹配结果。

## 参见

实例 5.2、实例 5.3 和实例 5.4。

## 5.2 查找多个单词之一

### 问题描述

你想要查找一个单词列表中的任意一个单词，但是不要对目标字符串进行多遍搜索。

# 解决方案

## 使用多选结构

这个问题的简单解答是在你想要匹配的不同单词之间使用多选操作：

```
\b(?:one|two|three)\b  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

关于匹配相似单词的更复杂例子，请参考实例 5.3。

## JavaScript 解答示例

```
var subject = 'One times two plus one equals three.';  
  
var regex = /\b(?:one|two|three)\b/gi;  
  
subject.match(regex);  
// returns an array with four matches: ['One', 'two', 'one', 'three']  
  
// This function does the same thing but accepts an array of words to  
// match. Any regex metacharacters within the accepted words are escaped  
// with a backslash before searching.  
  
function match_words (subject, words) {  
    var regex_metachars = /[\(\){}\[\]^+?.\\^$|,\\-]/g;  
  
    for (var i = 0; i < words.length; i++) {  
        words[i] = words[i].replace(regex_metachars, '\\$&');  
    }  
    var regex = new RegExp('\\b(?:' + words.join('|') + ')\\b', 'gi');  
    return subject.match(regex) || [];  
}  
  
match_words(subject, ['one', 'two', 'three']);  
// returns an array with four matches: ['One', 'two', 'one', 'three']
```

## 讨论

### 使用多选结构

这个正则表达式中包含三个部分：位于两端的单词边界、一个非捕获分组和单词列表（使用多选操作符 `|` 隔开）。单词边界会确保正则表达式不会匹配更长单词的一部分。非捕获分组会限制多选操作符的作用范围；否则，你可能就会需要使用 `\bone\b\btwo\b\b\btthree\b` 得到相同的效果。每个单词则会匹配其自身。

因为正则引擎会尝试从左向右匹配该列表中的每个单词，所以你可能会发现把目标文本中最可能被找到的单词放到列表的前面位置会在性能方面有较小的提高。因为单词的两端都是用单词边界来围住的，所以它们可以按照任何顺序出现。如果没

有单词边界，那么就有必要把长的单词放到前面；否则，你在搜索`<awe|awesome>`的时候就会永远也不可能找到“awesome”。这个正则表达式总是会在单词的开始处先匹配到“awe”。

注意这个正则表达式的意图是展示如何按照通用的方式匹配一个单词列表中的任意一个单词。因为这个例子中的`<two>`和`<three>`都以相同的字母开头，所以你还可以把这个正则式改写成`\b(?:one|two|hree)\b`。关于如何有效地匹配相似单词列表中的一个单词，请参考实例 5.3 中讲解的更多示例。

## JavaScript 解答示例

这个 JavaScript 示例使用两种不同方式来匹配相同的单词列表。第一种方式是简单地创建一个正则表达式，然后使用 JavaScript 字符串之上的 `match` 方法来搜索目标字符串。当 `match` 方法被传递了一个使用了`/g`（“global”）标志的正则表达式的时候，它会返回在该字符串中找到的所有匹配组成的一个数组，或者是在没有找到匹配的情况下返回 `null`。

第二种方法使用了一个函数（`match_words`）来接受要搜索的目标字符串，以及包含要查找的单词的一个数组。这个函数首先会对在所给单词中可能会出现的任何正则式元字符进行转义，然后把单词列表组合成一个新的正则表达式，用它来对字符串进行搜索。该函数会返回一个找到的所有匹配的一个数组，或者如果生成的正则式不能找到任何匹配，那么返回一个空字符串。由于使用了不区分大小写的`(i)` 标志，所以它可以按照大小写字母的任意组合来匹配所给的单词列表。

## 参见

实例 5.1、实例 5.3 和实例 5.4。

## 5.3 查找相似单词

### 问题描述

在这个实例中你要解决如下的几个问题：

- 你想要在一个字符串中找到所有的 `color` 和 `colour`；
- 你想要找到这 3 个以“at”结尾的单词中的任意一个：`bat`、`cat` 或 `rat`；
- 你想要找到以 `phobia` 结尾的任意单词；
- 你想要找到名字“`Steven`”的常见变形：`Steve`、`Steven` 和 `Stephen`；
- 你想要匹配术语“`regular expression`”的所有常见形式。

## 解决方案

下面会按顺序列出用来解决这些问题的正则表达式。所有这些解决方案中都使用了不区分大小写的选项。

### Color 或 colour

\bcolor?r\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### Bat、cat 或 rat

\b[bcr]at\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 以 “phobia” 结尾的单词

\b\w\*phobia\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### Steve、Steven 或 Stephen

\bSte(?:ven|phen)\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### “regular expression”的变体

\breg(?:ular|expressions|ex(?:ps|e[sn]))?\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

### 使用单词边界来匹配整个单词

所有这 5 个正则表达式都使用了单词边界（\b）以确保它们只会匹配整个单词。它们采用的模式中则使用了几种不同方式来在它们匹配的单词中允许出现不同变体。

下面我们来仔细介绍一下每个正则表达式。

### Color 或 colour

这个正则表达式会匹配 color 或者 colour，但是不会在 colorblind 中产生匹配。它使用

了量词 `<?>` 来使得位于该量词之前的“u”成为可选的。像 `<?>` 这样的量词与许多人更为熟悉的通配符的工作原理并不相同。相反的，它们会绑定到紧跟在它之前的元素之上，这个元素可以是单个记号（在这个例子中，就是字面字符“u”）或者用圆括号围起来的一组记号。量词 `<?>` 会重复之前的元素 0 次或 1 次。正则引擎首先会尝试匹配该量词绑定的元素，如果它无法产生匹配，那么引擎就会在不匹配它的情况下继续执行。允许 0 次重复的任意量词实际上也就会把它之前的元素变成可选的，而这正是我们在这里想要的。

## Bat、cat 或 rat

这个正则表达式使用了一个字符类来匹配“b”、“c”或“r”，然后是两个字面字符“at”。要完成同样的任务，你也可以使用`\b(?:b|c|r)at\b`、`\b(?:bat|cat|rat)\b`或者`\bbat\b|\bcat\b|\brat\b`。然而，如果在所允许的匹配之间的区别只是在一个字符列表中进行选择，那么你最好还是使用字符类。这不仅是因为字符类会提供更加简洁易读的语法（你可以去掉所有的垂直竖线，并且可以使用类似 A~Z 这样的范围），而且大多数正则引擎还会对字符类提供非常好的优化。使用垂直竖线的多选结构要求引擎使用在计算上代价很高的回溯算法，而字符类则只使用非常简单的搜索算法。

但是，还是有需要注意的地方。字符类属于最经常被错误使用的正则表达式特性之一。很可能这是因为它们并没有很好的文档，或者是因为有些读者忽略了其中的细节。不管原因如何，一定不要让你自己犯同样的新手错误。字符类只能从在指定的字符中一次匹配一个字符——从无例外。

下面是字符类最经常被用错的两种方式。

### 把单词放到字符类中

当然，如果你使用`<[cat]{3}>`也能匹配到 `cat`，但是它还会匹配 `act`、`ttt`，以及任意由列表中字符组成的三字符的组合形式。这在诸如`<[^cat]>`这样的否定字符类之上也会产生同样的效果，它会匹配不是 c、a 或 t 的任意单个字符。

### 试图在字符类中使用多选操作符

根据定义，字符类允许在所给出的字符列表之间进行选择。`<[a|b|c]>` 会匹配集合“abc|”中的一个字符，而这很可能并不是你所想要的。即使它是你想要的，那么这个字符类中也包含了一个冗余的垂直竖线。

关于如何正确和有效使用字符类的更多详细信息，请参考实例 2.3。

### 以“phobia”结尾的单词

与上一个正则表达式一样，这个正则式同样使用了一个量词来提供它所匹配字符串中的变化形式。例如，这个正则表达式会匹配 `arachnophobia` 和 `hexakosioihexekontahexaphobia`，

而且因为允许 0 次重复，所以正则式也会匹配 phobia。如果你想要在后缀“phobia”之前至少应该有一个字符，那么可以把 `(*)` 改成 `(+)`。

## Steve、Steven 或 Stephen

这个正则式组合了我们在前面例子中使用到的两个特性。一个非捕获分组（采用 `\((?:\w+?)\)` 的形式）会对多选操作符 `\|` 的作用范围进行限制。在该分组的第一个选择中使用量词 `(?)` 则会把它之前的字符 `\w+` 变成可选的。与等价的 `\bSte(?:ve|ven|phen)\b` 相比，这样做会提高效率（和简洁度）。同样的原则可以用来解释为什么把字面字符串 `\Ste` 放到在正则表达式的前面，而不是采用 `\b(?:Steve|Steven|Stephen)\b` 或者 `\bSteve\b|\bSteven\b|\bStephen\b` 的形式被重复 3 次。有些回溯的正则表达式引擎并没有聪明到能够推出来后面这两个正则式匹配到的任意文本都必须以 `Ste` 开头。相反地，当引擎逐步遍历目标字符串查找匹配的时候，它会先查找一个单词边界，然后检查随后的字符是不是一个 `S`。如果不是，那么引擎必须在尝试正则表达式中的所有可选的路径之后才会继续，并在字符串中的下一个位置重复这样的检查。虽然对于人来说很容易就可以看出这样做是在浪费精力（因为正则式中的所有可选路径都是以“`Ste`”开头的），但是正则引擎无法知道这个道理。而如果你把正则式写成 `\bSte(?:ven?|phen)\b`，那么正则引擎马上就会意识到它不可能匹配不以这三个字符开头的任何字符串。

关于回溯的正则表达式引擎的详细讨论，请参考实例 2.13。

## “regular expression”的变体

本实例中的这最后一个例子里用到了多选结构、字符类和量词，以匹配术语“regular expression”的任意常见变体。由于这个正则表达式一眼看去可能有些难懂，我们来把它进行分解，介绍其中的每个组成部分。

下面给出的正则式使用了宽松排列选项，这在 JavaScript 中是不支持的。因为在宽松排列模式中空白都会被忽略，所以字母的空格字符都使用反斜杠进行了转义：

```
\b          # 判断一个单词边界位置
reg         # 匹配 "reg"
(?:        # 分组但是不捕获...
  ular\    # 匹配 "ular"
  expressions? # 匹配 "expression" 或 "expressions"
  |        # 或者...
  ex       # 匹配 "ex"
  (?:      # 分组但是不捕获...
    ps?     # 匹配 "p" 或 "ps"
    |        # 或者...
    e       # 匹配 "e"
    [sn]    # 匹配集合"sn"中的一个字符
  )        # 非捕获分组的结束
  ?        # 重复之前的分组 0 次或 1 次
```

```
)          # 非捕获分组的结束  
\b        # 判断一个单词边界位置  
  
正则选项: 宽松排列, 不区分大小写  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

上述模式会匹配如下的 7 个字符串中的任意一个：

- regular expressions
- regular expression
- regexp
- regexes
- regexen
- regex

## 参见

实例 5.1、实例 5.2 和实例 5.4。

## 5.4 查找除某个单词之外的任意单词

### 问题描述

你想要使用一个正则表达式来匹配除了 cat 之外任意的完整单词。你应当可以匹配 Catwoman 和包含“cat”这几个字符的任意其他单词——只是不能匹配 cat。

### 解决方案

使用一个否定型顺序环视可以帮助你排除特定的单词，而这正是下面这个正则式的关键所在：

```
\b(?!cat\b)\w+  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### 讨论

虽然一个否定字符类（采用 `<[^…]>` 的形式）可以让你很容易匹配除了某个特定字符之外的任意字符，但是你并不能使用 `<[^cat]>` 来匹配除了单词 cat 之外的任意单词。虽然 `<[^cat]>` 是一个合法的正则式，但是它会匹配除了 c、a 或 t 之外的任意字符。因此，虽然 `\b[^cat]+\b` 会避免匹配单词 cat，但是它却同样无法匹配单词 cup，因为后者也包含了它不允许的字母 c。正则表达式 `\b[^c][^a][^t]\w*` 也还是不正确的，因为它会拒绝以 c 作为第一个字母，以 a 作为第二个字母，或者以 t 作为第三个字母的单词。而且，它也没有把前三个字母都限制为是单词字符，另外它只会匹配至少包含 3 个字符的单词，因为其中三个否定字符类都不是可忽略的。

在了解了这些情况之后，我们来仔细看一下这个实例开始处所给的正则表达式是如何来解决这个问题的：

```
\b      # 判断一个单词边界位置
(?!    # 判断下面的正则式不能在这里匹配...
  cat   # 匹配 "cat"
  \b     # 判断一个单词边界位置
)       # 否定型顺序环视的结束
\w+    # 匹配一个或多个单词字符
正则选项：宽松排列，不区分大小写
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

这个模式中的关键所在是一个否定型顺序环视，它采用的形式是 `<(?!...)>`。否定型顺序环视不允许出现在 `cat` 之后紧跟着一个单词边界的序列，但是并不会禁止这几个字符以其他顺序出现，或者是它们作为更长或者更短单词的一部分出现。在正则表达式的结尾处没有使用单词边界，因为它不会改变这个正则式的匹配。在 `\w+` 中的量词 `(+)` 会重复单词字符记号任意多次，这也就意味着它总是会一直匹配到下一个单词边界为止。

如果我们把这个正则式应用到目标字符串 `categorically match any word except cat` 之上，那么它会找到 5 个匹配：categorically、match、any、word 和 except。

## 变体

### 查找不包含另一个单词的单词

如果说我们现在要找的不是非 `cat` 的任意单词，而是要找到不包含 `cat` 的任意单词，那么就需要使用一个稍有不同的方式：

```
\b(?!cat)\w+\b
正则选项：不区分大小写
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

在这个实例的前面小节中，正则表达式开始处的单词边界为我们提供了一个方便的定位符，用来把否定型顺序环视放在单词的开始处。虽然这里使用的解决方案的效率并不是很好，但是不管怎样它也是一个较为常用的结构，用来让你可以匹配除了某个特定单词或模式之外的内容。它的做法是，对包含一个否定型顺序环视的分组和单个的单词字符进行重复。在匹配每个字符之前，正则引擎会确保单词 `cat` 不会从当前位置产生匹配。与上一个正则表达式不同之处是，这个正则式要求使用一个终止的单词边界。否则的话，它就会只匹配到一个单词的前面一部分，直到出现了 `cat` 的位置为止。

## 参见

实例 2.16 中对于环视（其中包括了肯定型和否定型的顺序环视和逆序环视）进行了更加深入的探讨。

实例 5.1、实例 5.5、实例 5.6 和实例 5.11。

## 5.5 查找后面不跟着某个特定单词的任意单词

### 问题描述

你想要匹配其后不会紧跟着单词 cat 的任意单词，可以忽略二者之间的任何空格、标点或其他非单词字符。

### 解决方案

否定型顺序环视是这个正则表达式的关键所在：

`\b\w+\b(?!W+cat\b)`

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

在实例 3.7 和实例 3.14 中可以找到如何在代码中实现这个正则表达式的示例。

### 讨论

与本章中的许多其他实例一样，单词边界（`\b`）和单词字符记号（`\w`）在一起使用就可以匹配一个完整单词。关于这些特性的深入讲解，读者可以参考实例 2.6。

把这个正则式的第二个部分包起来的 `<(...)>` 语法是一个否定型的顺序环视。顺序环视会告诉正则引擎在字符串中临时向前搜索，检查是否可以从当前位置向前匹配到位于顺序环视之内的模式。它并不会消费在顺序环视中匹配到的任何字符。它只是用来判断是否可能产生一个匹配。因为我们使用了一个否定型的顺序环视，所以判断的结果会被取反。换句话说，如果在顺序环视中的模式能够向前产生匹配，那么匹配尝试会宣告失败，而正则引擎会在目标字符串中从下一个字符开始从头来尝试所有的可能。关于顺序环视（以及与之相对的逆序环视）的更多详细信息，请参考实例 2.16。

在位于顺序环视之内的模式中，`\W+` 会匹配出现在 `cat` 之前的一个或多个非单词字符，而在结尾处的单词边界则会确保我们只会略掉不跟着作为整个单词的 `cat`，而不是略掉以 `cat` 作为开头的任意单词。

注意这个正则表达式甚至会匹配到单词 `cat`，只要其后跟着的单词不是 `cat` 即可。如果你还想要避免匹配到 `cat`，那么你就需要把这个正则式同在实例 5.4 中的正则表达式结合起来，这样会得到如下的正则表达式：`\b(?!cat\b)\w+\b(?!W+cat\b)`。

### 变体

如果你想要只匹配后面跟着 `cat` 的单词（但是在匹配文本中不包括 `cat` 和它之前的非单词字符），那么只需要把顺序环视从否定型改成肯定型就可以了：

`\b\w+\b(=?\W+cat\b)`

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 参见

实例 2.16 中包括关于环视的更详细讨论（包括肯定型、否定型的顺序环视和逆序环视）。  
实例 5.4 和实例 5.6。

## 5.6 查找不跟在某个特定单词之后的任意单词

### 问题描述

你想要匹配不会紧跟在单词 cat 之后的任意单词，可以忽略在二者之间的任意空格、标点或其他非单词字符。

### 解决方案

#### 逆序环视

逆序环视允许你检查在某个给定位置之前出现的文本。它的工作原理是告知正则引擎临时在字符串中进行回退，检查是否可以找到以逆序环视出现位置作为结束的匹配。如果读者想温习一下关于逆序环视的更多细节，请参考实例 2.16。

下面的三个正则表达式都使用了否定型逆序环视，它采用的语法是 `<(?!>)`。不幸的是，本书中讲解的正则流派对于在逆序环视中可以使用哪种模式有一些区别。因此，这些解答在每种情形下的执行情况也会有所不同。关于更多细节，请一定仔细阅读本实例中随后的“讨论”小节。

#### 不跟在“cat”之后的单词

```
(?<!\\bcat\\w+)\\b\\w+
正则选项：不区分大小写
正则流派：.NET

(?<!\\bcat\\w{1,9})\\b\\w+
正则选项：不区分大小写
正则流派：.NET、Java、PCRE

(?<!\\bcat) (?:\\w+|^) (\\w+)
正则选项：不区分大小写
正则流派：.NET、Java、PCRE、Perl、Python、Ruby 1.9
```

#### 模拟逆序环视

虽然 JavaScript 和 Ruby 1.9 都支持顺序环视，但是它们根本不支持逆序环视。然而，因为这个问题中的逆序环视正好出现在正则式的开头，所以我们完全可以通过把这个正则式划分成两个部分来对逆序环视进行模拟。下面的 JavaScript 代码示例给出了一

个示范：

```
var subject = 'My cat is furry.',
    main_regex = /\b\w+/g,
    lookbehind = /\bcat\W+$/i,
    lookbehind_type = false, // negative lookbehind
    matches = [],
    match,
    left_context;
while (match = main_regex.exec(subject)) {
    left_context = subject.substring(0, match.index);
    if (lookbehind_type == lookbehind.test(left_context)) {
        matches.push(match[0]);
    } else {
        main_regex.lastIndex = match.index + 1;
    }
}
// matches: ['My', 'cat', 'furry']
```

## 讨论

### 固定长度、有限长度和无限长度的逆序环视

第一个正则表达式使用的是否定型的逆序环视 `(?<!\bcat\W+)`。因为用在逆序环视之内的量词`(+)`并没有对于它可以匹配多少个字符进行限制，所以这个版本只能用在.NET 正则表达式流派中。本书中讲到的所有其他正则表达式流派都要求在逆序环视中的模式必须拥有固定长度或最大（有限）长度的限制。

第二个正则表达式把逆序环视中的量词`(+)`改成了`{1,9}`。因此，它可用于.NET、Java 和 PCRE 中，所有这些流派都支持可变长度的逆序环视，但是要求指定它们可以匹配的字符个数的上限。在这个例子中，作者随便指定了用来分隔单词的非单词字符串的最大长度是 9。这样就允许在单词之间出现一些标点和几个空行。除非你要处理的目标文本非常怪异，否则这个解答与上一个只支持.NET 的解答的执行结果应该是完全一样的。然而，即使在.NET 中，为逆序环视中的任意量词提供一个合理的重复次数上限，也很可能会提高正则表达式的执行效率，因为这样会减少可能在逆序环视中出现的不必要的回溯的数量。

第三个正则表达式的构造方式是允许逆序环视检测一个固定长度的字符串，因此就可以支持更多的正则表达式流派。为了能够达到这个目标，我们把非单词字符使用的简写字符类 (`\W`) 移到了逆序环视的外面。这意味着在你要查找的单词之前的非单词字符（比如标点和空格）就会被当作实际上匹配到的字符串的一部分，被包含在正则表达式的返回结果中。为了能够更加容易忽略掉匹配中的这个部分，我们在最后的单词字符串序列之上添加了一个捕获分组。采用一点额外的代码，你就可以不读整个匹配的

结果，而是只读出向后引用 #1 中的内容，从而可以得到与前两个正则表达式相同的结果。关于用来处理向后引用的代码，请参考实例 3.9。

## 模拟逆序环视

JavaScript 不支持逆序环视，但是前面所给的 JavaScript 代码示例中展示了如何使用两个正则表达式来模拟出现在正则表达式开头的逆序环视。该代码对于（被模拟的）逆序环视可以匹配的文本长度并没有施加任何限制。

首先，我们把最初解决方案所给的正则表达式 `<(?<!\bcat\W+)\b\w+>` 拆分成了两个部分：位于逆序环视中的模式 (`\bcat\W+`) 和紧随其后的内容 (`\b\w+`)。在逆序环视正则式的结尾添加一个 `$` 标记。如果你需要在 `lookbehind` 正则式中采用“脱字符和美元符号匹配换行处”的选项 (`/m`)，那么应当使用的是 `$(?!s)` 而不是 `$`，其目的是为了确保它能匹配到目标文本的结尾处。变量 `lookbehind_type` 用来指定我们模拟的是肯定型还是否定型的逆序环视，使用 `true` 来模拟肯定型逆序环视，而 `false` 则表示否定型。

在上述变量设置好之后，我们使用 `main_regex` 和 `exec` 两个方法来遍历目标字符串（关于这个过程的描述，请参考实例 3.11）。当找到匹配的时候，在该匹配之前的目标文本子串会被拷贝到一个新的字符串变量中 (`left_context`)，然后我们会检查 `lookbehind` 正则式是否匹配该字符串。因为在 `lookbehind` 正则式的结尾添加了定位符，所以这样就会把第二个匹配刚好定位到第一个匹配的左边。通过把逆序环视检查的结果同 `lookbehind_type` 做对比，我们就可以得知该匹配是不是符合总的成功匹配的标准。

最后，我们选择下列的步骤之一。如果得到了成功匹配，那么就把匹配到的文本添加到 `matches` 数组中。否则，把继续搜索下一个匹配的位置（使用的是 `main_regex.lastIndex`）修改为 `main_regex` 对象的上一个匹配的开始位置之后的一个字符，而不是让 `exec` 方法从当前匹配的结尾继续下一次循环。

大功告成！

这里使用了一个高级的技巧，利用了在采用/g (“global”) 标志的 JavaScript 正则表达式中会动态进行更新的 `lastIndex` 属性。通常来说，对 `lastIndex` 进行更新和重置都是会自动进行的。在这里，我们用它来控制正则表达式在目标字符串中的行进路线，按照需要向前向后移动。这个技巧只能允许你模拟出现在一个正则式开头的逆序环视。然而，它并不能作为逆序环视支持的完全替代。由于逆序环视和回溯的互相影响，这种方式无法帮助你准确地模拟出现在正则表达式中间的逆序环视的行为。

## 变体

如果你只想匹配紧跟在 `cat` 之后的单词（在匹配到的文本中不包括 `cat` 和跟在它之后的非单词字符），那么只需把否定型逆序环视改为肯定型逆序环视：

```
(?<=\bcat\W+)\b\w+
正则选项: 不区分大小写
正则流派: .NET

(?<=\bcat\W{1,9})\b\w+
正则选项: 不区分大小写
正则流派: .NET、Java、PCRE

(?<=\bcat)(?:\w+|^)(\w+)
正则选项: 不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby 1.9
```

## 参见

实例 2.16 中包括关于环视的更详细讨论（包括肯定型、否定型的顺序环视和逆序环视）。  
实例 5.4 和实例 5.5。

## 5.7 查找临近单词

### 问题描述

你想要使用一个正则表达式来模拟 NEAR 查找。可能有些读者对于 NEAR 这个术语并不熟悉，在有些查找工具中除了可以使用 NOT 和 OR 这样的布尔操作之外，还支持一种特殊的操作符 NEAR。查找“word1 NEAR word2”会找到按任意顺序出现的 word1 和 word2，只要它们位于某个特定距离之内即可。

### 解决方案

如果你要查找的只是两个不同单词，那么可以把两个正则表达式组合起来——一个先匹配 word1 再匹配 word2，另一个则按照相反顺序来匹配。下面的正则表达式允许在你要查找的两个单词之间出现最多 5 个单词：

```
\b(?:word1\W+(?:\w+\W+){0,5}?word2|word2\W+(?:\w+\W+){0,5}?word1)\b
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

\b(?:#
  word1          # 第一个目标单词
  \W+ (?:\w+\W+){0,5}?
  word2          # 第二个目标单词
  |
  word2          # 或者, 把同样的模式反过来...
  \W+ (?:\w+\W+){0,5}?
  word1          # 第一个目标单词
) \b
正则选项: 宽松排列, 不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

这里所给的第二个正则表达式使用了宽松排列选项，因此可以在其中添加空格和注释来提高可读性。除此之外，这两个正则表达式是完全相同的。JavaScript 不支持宽松排列模式，但是前面列出的其他正则流派都支持你选择这二种形式之一。实例 3.5 和实例 3.7 中讲解的例子会告诉你如何把这些正则表达式添加到搜索表单或其他代码中。

## 讨论

这个正则表达式把同一个模式的两个顺序相反的拷贝放到一起，并且为它们添加了单词边界。第一个子模式会匹配 word1，然后是 0~5 个单词，然后是 word2；第二个子模式则会匹配相同的内容，但是 word1 和 word2 的顺序被倒了过来。

在两个子模式中都使用了懒惰量词 `\{0,5\}?`。它会使正则表达式在你要查找的两个目标单词之间匹配尽量少的单词。如果你使用目标文本 `word1 word2 word2` 来执行这个正则表达式，那么它会匹配到的是 `word1 word2`，因为在这个子串中两个目标单词之间包含了最少的（0 个）单词。如果想要对目标单词之间允许的距离进行配置，那么可以把在两个量词中的 0 和 5 修改为你想要的取值。例如，如果你把它们修改为 `\{1,15\}?`，那么就会允许在你要查找的两个单词之间出现最多 15 个单词，同时要求它们之间必须有一个其他单词作为分隔。

这里用来匹配单词字符和非单词字符的简写字符类（分别是 `\w` 和 `\W`）所遵循的正则表达式定义要求在单词中只能包含字母、数字和下划线。

## 变体

### 使用条件判断

通常来说，同一个正则表达式会有不同的构造方法。在本书中，我们尽量尝试在可移植性、简洁性、效率和其他考虑事项之间进行权衡。然而，有时候不是很理想的解决方案也同样有学习的价值。下面这两个正则表达式会讲解用来查找临近单词的其他方式。我们并不推荐在实际中使用这些方式，因为虽然它们会匹配同样的文本，但是它们通常需要花费更多的时间。另外，它们适用的正则表达式流派也较少。

第一个正则表达式使用一个条件判断来决定在正则式的结尾是匹配 word1 还是 word2，而不是简单尝试把两个反转的模式放到一起。条件判断会检查捕获分组 1 是否参与了匹配，如果是，就意味着匹配是以 word2 开头的：

```
\b(?:word1|(word2))\W+(?:\w+\W+)\{0,5\}?(?(1)word1|word2)\b  
正则选项: 无  
正则流派: .NET、PCRE、Perl、Python
```

这下一个版本同样采用了一个条件判断来决定在最后应该匹配哪个单词，但是它在其中添加了两个正则表达式特性：

```
\b(?:(<w1>word1) | (<w2>word2))\w+(:\w+\w+)\{0,5\}?(?w2)(?&w1) | (?&w2))\b  
正则选项: 无  
正则流派: PCRE 7、Perl 5.10
```

在这里采用了命名捕获分组（语法是 `<(?<name>…)>`）来把 `<word1>` 和 `<word2>` 的第一个实例括了起来。这样就允许你使用子例程（*subroutine*）语法 `<(?&name)>` 来复用通过名称来指定的子模式。这与到一个命名分组的向后引用是不一样的。命名向后引用的格式是 `\k<name>`（.NET、PCRE 7 和 Perl 5.10）或 `<(P=name)>`（PCRE 4 及更高版本、Perl 5.10 和 Python），它允许你重新匹配由一个命名的捕获分组已经匹配到的文本。一个采用 `<(?&name)>` 形式的子例程则会允许你复用在相应分组中包含的实际模式。在这里不能使用向后引用，因为那样只会允许你重新匹配已经匹配过的单词。在正则匹配结尾处的条件判断中的子例程会匹配前面提供的两个选项中还没有匹配过的那个单词，而不必再重新说明具体要匹配的是哪个单词。这意味着如果需要重复使用这个正则式来匹配不同单词，那么你只需在正则式中修改一个地方就可以了。

## 匹配临近的 3 个或更多单词

指数级增长的排列组合。匹配临近的两个单词是一个相对比较简单的任务。毕竟只有两种不同的方式可以对它们进行排列。但是如果你想要匹配任意顺序的 3 个单词该怎么办？现在就存在六种可能的顺序（参见图 5-2）。对于一个给定的单词集合来说，可能的排列顺序个数是  $n!$ ，也就是从 1 到  $n$  的所有整数的乘积（“ $n$  的阶乘”）。如果有 4 个单词，那么就会存在 24 种不同的排列方式。等你要处理 10 个单词的时候，可能的排列方式就急剧增长到了数百万种。因此，再使用我们前面讨论的正则表达式来匹配临近的多个单词显然就不现实了。

难看的解决方案。解决这个问题的一种方式是重复可以匹配目标单词或其他任意单词的一个分组（在首次匹配了一个目标单词之后），然后使用条件判断来要求直到所有的目标单词都匹配后，才会宣布匹配成功。下面给出的例子可以匹配任意顺序的 3 个单词，彼此之间可以有不超过 5 个单词作为分隔：

```
\b(?:(>(word1) | (word2) | (word3) | (?1) | (?2) | (?3) | (?!)))\w+\b\W*?\{3  
,8}\b  
(?1) (?2) (?3) | (?!) | (?!) | (?!)  
正则选项: 不区分大小写  
正则流派: .NET、PCRE、Perl
```

下面这个正则式把上面的原子分组（参见实例 2.14）替换为了标准的非捕获分组，从而添加了对 Python 的支持：

```
\b(?:(>(word1) | (word2) | (word3) | (?1) | (?2) | (?3) | (?!)))\w+\b\W*?\{3,8}\b  
(?1) (?2) (?3) | (?!) | (?!) | (?!)  
正则选项: 不区分大小写  
正则流派: .NET、PCRE、Perl、Python
```

2 个值：

[ 12, 21 ]  
= 2 种可能排列

3 个值：

[ 123, 132,  
213, 231,  
312, 321 ]  
= 6 种可能排列

4 个值：

[ 1234, 1243, 1324, 1342, 1423, 1432,  
2134, 2143, 2314, 2341, 2413, 2432,  
3124, 3142, 3214, 3241, 3412, 3421,  
4123, 4132, 4213, 4231, 4312, 4321 ]  
= 24 种可能排列

阶乘：

2! = 2 × 1	=	2
3! = 3 × 2 × 1	=	6
4! = 4 × 3 × 2 × 1	=	24
5! = 5 × 4 × 3 × 2 × 1	=	120
...		
10! = 10 × 9 × 8 × 7 × 6 × 5 × 4 × 3 × 2 × 1	=	3628800

图 5-2 一个集合可能会存在多种排列方式

在这个正则表达式中的量词 `\{3,8\}` 要求必须出现 3 个目标单词，因此允许在它们之间出现 0~5 个单词。空的否定型顺序环视 (`\((?!)\)`) 永远都不可能产生匹配，因此被用来阻塞正则表达式中的某些特定路径，直到一个或多个目标单词被匹配到为止。控制这些路径的逻辑使用两个嵌套条件判断的集合来实现。第一个集合避免会在至少一个目标单词被匹配之前使用 `\w+`。而在结尾处的第二个条件判断集合则会强制要求正则引擎在所有要求的目标单词都被匹配到之前，只能进行回溯，或者宣告匹配失败。

上面给出的只是对其中的基本工作原理的简单介绍，与其进一步深入了解如何添加更多的目标单词，我们先来看下面这个改进的实现，其中添加了对更多正则流派的支持，并且涉及了一个小技巧。

利用空的向后引用。上面给的难看的解决方案虽然也是可以工作的，但是由于它非常难于理解和管理，所以拿出去或许可以赢得一场正则式猜谜大赛。如果你在其中添加更多的目标单词，那么只会把它变得更糟。

幸运的是，还有一个正则表达式技巧可以用来使之较为容易理解，同时还可以添加对 Java 和 Ruby (二者都不支持条件判断) 的支持。



## 警告

如果要把在这个小节中解释的行为真正应用于产品中，一定要非常小心。我们在这里使用的正则表达式行为在大多数正则函数库中都是找不到的。

```
\b(?:(?>word1()|word2()|word3()|(?>\1|\2|\3)\w+)\b\W*){3,8}\1\2\3
```

正则选项：不区分大小写

正则流派：.NET、Java、PCRE、Perl、Ruby

```
\b(?:(?:word1()|word2()|word3()|(?:\1|\2|\3)\w+)\b\W*){3,8}\1\2\3
```

正则选项：不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

使用上述结构，就可以很容易添加更多的目标单词。下面的例子允许 4 个目标单词以任意顺序出现，并且在它们之间可以包含最多 5 个其他单词：

```
\b(?:(?>word1()|word2()|word3()|word4())|  
(?>\1|\2|\3|\4)\w+)\b\W*?){4,9}\1\2\3\4
```

正则选项：不区分大小写

正则流派：.NET、Java、PCRE、Perl、Ruby

```
\b(?:(?:word1()|word2()|word3()|word4())|  
(?:\1|\2|\3|\4)\w+)\b\W*?){4,9}\1\2\3\4
```

正则选项：不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

这些正则表达式有意在每个目标单词之后都使用了一个空的捕获分组。因为一个捕获分组还没有参与匹配，那么试图匹配与之相对应的向后引用（比如 `\1`）就一定会失败，这样就可以使用对空分组的向后引用来控制正则引擎在一个模式中所选取的路径，它的功能与我们在前面所给的更加晦涩的条件判断是非常类似的。如果在正则引擎到达一个向后引用的时候，与之对应的分组已经参与了匹配尝试，那么它就会匹配一个空串，然后继续执行。

在这里的 `(?>\1|\2|\3)` 分组会避免在至少匹配一个目标单词之前使用 `\w+` 来匹配单词。在模式结尾处的向后引用会被重复执行，从而直到所有必需的目标单词都被找到之后，匹配才能成功完成。

Python 不支持原子分组，因此上面把 Python 列在正则流派中的示例里，又一次把原子分组替换为了标准的非捕获分组。虽然这样会使正则式的效率有所降低，但是它不会改变匹配的结果。最外层的分组在任意流派中都不能是原子的，因为要想能够达到目标，如果在模式最后的向后引用匹配失败，那么正则引擎必须能够回溯到外层分组中。

按照 JavaScript 规则的向后引用。虽然 JavaScript 支持在这个模式的 Python 版本中使用的所有语法，但是它包含的两条行为规则会使这个技巧无法像在其他流派中一样工作。第一个问题是指向还没有参与匹配的捕获分组的向后引用会匹配到什么内

容。JavaScript 规范中说明这样的向后引用会匹配到空字符串，或者换句话说，它们总是会匹配成功。而在几乎所有其他正则表达式流派中，得到的结果都是恰好相反的：它们永远不可能匹配成功，而结果是它们会迫使正则引擎回溯，直到整个匹配失败，或者它们引用的分组参与了匹配，从而就有可能让所有的向后引用都最终匹配成功。

JavaScript 流派的第二个区别是嵌套在一个重复的外层分组（例如 `\((a)|(b)\)+`）中的捕获分组所记住的取值。对于大多数正则流派来说，在一个重复分组中的捕获分组所记住的取值是该分组上一次参与到匹配中时所匹配到的内容。因此，在 `\((?:(a)|(b))\)+` 被用来匹配 `ab` 之后，向后引用 #1 中的值是 `a`。然而，根据 JavaScript 规范的说明，到嵌套分组中的向后引用的值在每次外层分组重复时都会被重置。因此，`\((?:(a)|(b))\)+` 还是会匹配 `ab`，只不过在匹配完成之后，向后引用 #1 指向的是一个未参与匹配的捕获分组，而在 JavaScript 中它会在正则式中匹配一个空字符串，并且在调用它的方法（例如 `RegExp.prototype.exec`）所返回的数组中返回 `undefined`。

根据我们上面的讲解，读者会发现在 JavaScript 正则流派中会遇到的这两种行为所造成的变化都会使我们无法使用空捕获分组来模拟条件判断。

## 多个单词、彼此之间可以是任意距离

如果你只是想要检查在一个目标字符串中是否能找到一个单词列表，而不管它们之间是否接近，那么可以在一个查找操作中使用肯定型顺序环视来完成这个任务。



### 提示

在许多情况下，对你要查找的每个目标单词进行分别查找，然后跟踪判断是否所有测试都匹配成功，这样的做法可能会更加简单有效。

`\A(?=.*?\bword1\b)(?=.*?\bword2\b).*\Z`

正则选项：不区分大小写、点号匹配换行符

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

`^(?=.*[\s\S]*?\bword1\b)(?=.*[\s\S]*?\bword2\b)[\s\S]*$`

正则选项：不区分大小写（“^和\$匹配换行处”必须关闭）

正则流派：JavaScript

如果你要的所有目标单词都能在目标字符串中找到，那么这些正则表达式会匹配整个字符串；否则，它们不会找到任何匹配。JavaScript 程序员不能使用第一种形式，因为 JavaScript 中不支持定位符 `\A` 和 `\Z`，或者是“点号匹配换行符”的选项。

你可以根据实例 3.6 中的代码来实现这些正则表达式。只需要把其中的占位符 `\word1` 和 `\word2` 替换为你要查找的单词。如果需要检查超过两个单词，那么你可以在正则式的前面根据需要添加更多的顺序环视。例如，`\A(.*?\bword1\b)(.*?\bword2\b)(.*?\bword3\b).*\Z` 就可以查找 3 个单词。

## 参见

实例 5.5 和实例 5.6。

## 5.8 查找重复单词

### 问题描述

你在编辑一个文档的时候，想要检查其中是否包含被错误重复的单词。在查找这些重复单词的时候不必在意大小写的区别，例如是 “The the”。你同样还要允许在单词之间出现任意数量的空白，这样，即使会造成两个单词出现在不同的行中也可以找到。

### 解决方案

向后引用会匹配一些前面匹配过的内容，因此也就为这个实例提供了关键的组成成分：

```
\b([A-Z]+)\s+\1\b  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

如果你想要使用这个正则表达式来保留第一个单词，但去掉随后的重复单词，就需要把所有匹配都替换为向后引用 #1。另外一种方式是对匹配到的内容进行突出显示，比如在它们两边使用一些其他字符（例如 HTML tag），从而可以在随后检查的时候更加容易识别它们。实例 3.15 讲解了如何在替代文本中使用向后引用，而在实现前面提到的这两种方式的时候都会用到。

如果你只是想要找到重复单词，从而可以手动检查它们是否需要被更正，那么可以使用实例 3.7 中的代码。一个文本编辑器或类似 grep 这样的工具，例如在第 1 章中“使用正则表达式的工具”一节中提到的那些工具，也会在查找重复单词的时候帮助你提供所需的上下文，从而可以决定有问题的单词是否真的使用正确。

### 讨论

捕获分组和向后引用都需要匹配先前匹配过的内容。把你想要匹配多次的内容放到一个捕获分组中，然后使用向后引用来再次匹配它。这同简单地使用量词来重复一个记号或分组的工作原理是不一样的。我们来考虑一下两个简单正则表达式 `\w\1` 和 `\w{2}` 之间的区别。第一个正则式使用一个捕获分组和向后引用来两次匹配相同的单词字符，而后者则使用一个量词来匹配任意两个单词字符。实例 2.10 中更加深入地讨论了向后引用的用法。

再回到我们这里的问题。这个实例只找到由字母 A~Z 和 a~z 组成的重复单词（因为应用了不区分大小写的选项）。如果要在单词中允许出现含读音符号的字母和其

他字母表中的字母，那么如果在你的正则流派中支持，可以使用 Unicode 中的 Letter 属性 (`\p{L}`)。更多信息请参考实例 2.7 中的“Unicode 属性或类别”小节中的讲解。

在捕获分组和向后引用之间，`\s+` 会匹配任意的空白字符，例如空格、制表符或者换行符。如果你想要把可以分隔重复单词的字符限制为水平空白（也就是说不包含换行），那么可以把 `\s` 替换为 `[^\S\r\n]`。这样会避免匹配跨多行出现的重复单词。PCRE 7 和 Perl 5.10 中都包含简写字符类 `\h`，你在这里也可以用它来实现，因为它就是被设计来匹配水平空白符号的。最后，在正则表达式开始和结尾的单词边界会确保它不会匹配其他单词中的一部分，例如是“this thistle”。

要注意使用重复单词并不总是错误的，因此如果不做进一步审查就简单删除重复单词就有可能会是危险的。例如，像“that that”和“had had”这样的结构在英语口语中通常是可以接受的。同形异义词、姓名、拟声词（例如“oink oink”或者“ha ha”），以及其他一些结构也会偶尔造成有意重复的单词。因此，在大多数情况下，你需要人工对每个匹配进行检查。

## 参见

实例 2.10 中深入讨论了向后引用。

实例 5.9 中讲解如何匹配重复的文本行。

## 5.9 删除重复的文本行

### 问题描述

你要处理一个日志文件、数据库查询输出或者是其他一些类型的文件或字符串，其中可能会包含重复的文本行。你需要使用一个文本编辑器或其他类似工具来删除所有重复的文本行，只保留其中一个。

### 解决方案

各种不同的软件（包括 Unix 命令行工具 `uniq` 和 Windows PowerShell 命令行 `Get-Unique`）都可以帮助你删除文件或字符串中的内容重复的文本行。下面的小节中包含 3 种基于正则式的解决方案，它们都比较适合用于在一个不支持脚本的文本编辑器中使用正则表达式的查找和替换支持来完成这个任务。

在编程的时候，应当避免使用方法 2 和方法 3，因为相对其他可选方案（例如使用一个 `hash` 对象来记录所有不重复的文本行）来说，它们的效率较低。然而，第一种方法（它要求首先对所有文本行进行排序，除非你只想删除相邻的重复文本行）会是一种可以接受的方案，因为它不仅速度快而且使用简便。

## 方法 1：先对文本行排序然后删除相邻的重复行

如果能够对要处理的文件或字符串中的所有文本行进行排序，从而可以让所有重复的文本行都出现在相邻位置，那么除非你一定要保留每行原来的顺序，否则，都应该这样做。这个方法会允许你使用更简单与更高效的查找与替换操作来删除其中的重复，如果不排序，就不可能这样做。

在对所有文本行排序之后，使用下面的正则式和替代字符串就可以删除重复行：

```
^(.*)(?:\r?\n|\r)\1)+$
```

正则选项：^ 和 \$ 匹配换行处（“点号匹配换行符”必须关闭）

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

替换为：

```
$1
```

替代文本流派：.NET、Java、JavaScript、Perl、PHP

```
\1
```

替代文本流派：Python、Ruby

这个正则表达式使用了一个捕获分组和一个向后引用（以及其他组成部分）来匹配两个或多个连续的重复行。在替代文本中使用一个向后引用，把第一行放回到目标文本中。读者可以通过修改实例 3.15 中所给的代码示例来实现这个正则表达式。

## 方法 2：在未排序的文件中保留每个重复行的最后一次出现

如果你使用的文本编辑器没有内置的功能来对文本行进行排序，或者是有必要保留原来的文本行顺序，那么下面的解决方案可以帮助你在没有进行排序的情况下，对不连续的重复行实施删除：

```
^(?![\r\n]*)(?:\r?\n|\r)(?=.*^\1$)
```

正则选项：点号匹配换行符、^ 和 \$ 匹配换行处

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

要想构造一个 JavaScript 兼容的正则表达式，可以保留上面的正则式，只要去掉“点号匹配换行符”的选项就可以了：

```
^(.*)(?:\r?\n|\r)(?=#[\s\S]*^\1$)
```

正则选项：^ 和 \$ 匹配换行处（“点号匹配换行符”必须关闭）

正则流派：JavaScript

替换为：

（空字符串，也就是替换为空）

替代文本流派：N/A

## 方法 3：在未排序的文件中保留每个重复行的第一次出现

如果想要保留每个重复行的第一次出现，那么你就需要使用稍微不同的方式。首先，

下面是我们要使用的正则表达式和替代字符串：

```
^([^\\r\\n]*$|.*?)(?:(?:\\r?\\n|\\r)\\1$)+  
正则选项：点号匹配换行符、^ 和 $ 匹配换行处  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

因为 JavaScript 不支持“点号匹配换行符”的选项，因此如果需要让上面的正则表达式与 JavaScript 流派兼容，那么就需要做一些改动。

```
^(.*)$((\\s\\S)*?) (?:(?:\\r?\\n|\\r)\\1$)+  
正则选项：^ 和 $ 匹配换行处（“点号匹配换行符”必须关闭）  
正则流派：JavaScript
```

替换为：

```
$1$2  
替代文本流派：.NET、Java、JavaScript、Perl、PHP  
\\1\\2  
替代文本流派：Python、Ruby
```

与方法 1 和方法 2 中的正则式不同，这个版本不能采用一次查找和替换操作就删除所有的重复行。你需要持续应用“全部替换”，直到正则式无法在字符串中找到匹配，也就意味着所有重复行都已经被删除了。更多细节请参考本实例随后的“讨论”小节。

## 讨论

### 方法 1：对文本行排序然后删除相邻的重复行

这个正则表达式删除了除第一个之外的所有相邻的重复行。它并不会删除被其他文本行分隔的重复行。我们来分步骤看一下这个过程。

首先，在正则表达式开始的脱字符（`\^`）会匹配一行的开头。通常它只会匹配目标字符串的开始，因此你就需要保证应用了让`^`和`$`可以匹配换行处的选项（关于如何设置正则选项，请参考实例 3.4 中的讲解）。接下来，在捕获圆括号之内的`(*)`会匹配一行的所有内容（包括空行），匹配到的结果被保存到向后引用 #1 中。要想让这里能正确工作，就一定不能设置“点号匹配换行符”的选项；否则，点号加星号的组合会一直匹配到字符串的结束。

在外层的非捕获分组中，我们使用了`<(?:\\r?\\n|\\r)>`来匹配在 Windows（`\r\n`）、Unix/Linux/OS X（`\n`）或老版本的 Mac OS（`\r`）的文本文件中所使用的换行分隔符。接着，向后引用`\1`会尝试匹配我们刚才匹配到的那行内容。如果在这个位置没有找到同样一行，那么匹配尝试会失败，然后正则引擎会继续执行。如果它匹配成功，那么我们会使用量词`+>`来重复这个分组（由换行序列和向后引用 #1 组成），以尝试匹配更多的重复行。

最后，在正则表达式结尾的美元符号会判断一行的结束位置。这会确保我们只匹配完

全相同的文本行，而不是以前面一行的内容作为开头的文本行。

因为我们在做的操作是查找和替换，所以每个整体匹配（包括原来的文本行与行分隔符）都会从字符串中删除。然后我们会用向后引用 #1 把原来的文本行再替换回来。

## 方法 2：在未排序的文件中保留每个重复行的最后一次出现

与方法 1 中只能找到相邻重复行的正则表达式相比，这个正则式做了几处改动。首先，在方法 2 正则式的非 JavaScript 版本中，捕获分组中的点号被替换为了 «[^\\r\\n]»（除换行之外的任意字符），而且还设置了“点号匹配换行符”的选项。这是因为点号会在这个正则式的后面用来匹配任意字符，其中也包括换行符。其次，这个正则式添加了一个顺序环视来继续在字符串中的任何位置搜索可能的重复行。因为顺序环视并不会消费任何字符，所以该正则式匹配到的文本总是出现在字符串稍后部位的一行内容（包括它之后的换行符）。把所有匹配替换为空串就会删除重复行，从而把每组重复行的最后一次出现保留在字符串中。

## 方法 3：在为排序的文件中保留每个重复行的第一次出现

因为逆序环视并没有得到像顺序环视一样广泛的支持（而且即使在支持它的时候，也不能够向后看太远），因此方法 3 中的正则表达式与方法 2 中有很大不同。这里匹配的不是已知在字符串前面部分已经有重复的文本行（这样就可以与方法 2 中的技术相一致），而是要匹配一行内容、在字符串后面与该行重复的另外一行以及在二者之间的所有文本行。最初的那行内容被保存为向后引用 #1，在二者之间（如果有的话）的内容会被保存为向后引用 #2。通过把每个匹配替换为向后引用 #1 和 #2，你就可以把你想要的内容放回去，而同时删除最后的重复行以及它前面的换行。

这种方法也存在一些问题。首先，因为每个重复行的匹配集合中可能会包含其他行，所以就有可能会在你匹配到的文本中包含其他的重复行，而这些重复行会在进行“全部替换”操作的过程中被忽略。其次，如果一行被重复超过两次的话，那么正则式会首先匹配重复行 1 和 2，但是在此之后，当继续遍历字符串剩余部分的时候，还需要一对重复行才能使这个正则式产生匹配。因此，只是一次“全部替换”动作最多只能删除任意特定行的每两次重复中的一次。为了解决上述两个问题，并确保所有重复行都可以被删除，你就需要继续在目标字符串之上应用查找和替换操作，直到正则表达式不会在其中找到匹配为止。我们来看一个例子，假设把上述正则表达式应用到下面的目标字符串之上：

```
value1  
value2  
value2  
value3  
value3  
value1  
value2
```

从这个字符串中删除所有重复行需要执行 3 遍替换。表 5-1 展示了每遍操作之后的结果。

表 5-1 每遍替换之后的结果

第 1 遍	第 2 遍	第 3 遍	第 4 遍
value1	value1	value1	value1
value2	value2	value2	value2
value2	value2	value3	value3
value3	value3	value2	
value3	value3		
value1	value2		
value2			
1 个匹配/替换	2 个匹配/替换	1 个匹配/替换	没有找到重复

## 参见

实例 2.10 详细讨论了关于向后引用的内容。

实例 5.8 讲解了如何匹配重复单词。

## 5.10 匹配包含某个单词的整行内容

### 问题描述

你想要匹配在其中任何地方包含单词 `ninja` 的所有文本行。

### 解决方案

`^.*\bninja\b.*$`

正则选项：不区分大小写、^和\$匹配换行处（“点号匹配换行符”必须关闭）

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

常常会有需要匹配整行内容，从而可以收集或者删除它们。如果要想匹配包含单词 `ninja` 的任意行，我们首先构造一个正则表达式 `\bninja\b`。在两端的单词边界会确保只能以整个单词的形式匹配 “`ninja`”。更多细节，请参考实例 2.6。

要把这个正则式进行扩展来匹配整行内容，就需要在两端都添加 `<*>`。点号加星号的序列会在当前行匹配 0 个或多个字符。星号量词是贪心的，所以它们会尽量匹配更多的文本。第一个“点号-星号”会一直匹配到在该行中最后一次出现 “`ninja`” 的位置，而第二个“点号-星号”则会匹配在其后的任意非换行符的字符。

最后，我们把脱字符和美元符号分别放到正则表达式的开始和结尾，以确保它只会匹配一整行的内容。严格来讲，结尾处的定位符美元符号是冗余的，因为点号和贪心的星号总是会匹配到一行的结尾。然而，添加它总不会带来什么坏处，而且会使该正则表达式看起来更加易读。在正则式中添加合适的行定位符或字符串定位符，有时候会帮助你避免意想不到的问题，因此形成这样的习惯是有益的。注意，与美元符号不一样，在正则表达式开始处的脱字符则不一定是冗余的，因为它确保正则式只会匹配到整行，因为也许由于某种原因，匹配查找有可能会从一行的中间开始。

需要记住用来把匹配限制到一行的 3 个关键元字符（定位符 `\^`、`\$` 以及点号）的含义并不固定。要想让它们都是面向行来处理的，你就必须打开让 `\^` 和 `\$` 匹配换行处的选项，同时确保没有打开让点号匹配换行符的选项。实例 3.4 中讲解了如何在代码中应用这些选项。如果你使用的是 JavaScript 或者 Ruby，那么就可以少担心一个选项，因为 JavaScript 中没有让点号匹配换行符的选项，而 Ruby 中的脱字符和美元符号则总是会匹配换行处。

## 变体

要搜索包含多个单词之一的文本行，可以使用多选结构：

```
^.*\b(one|two|three)\b.*$
```

正则选项：不区分大小写、`\^` 和 `\$` 匹配换行处（“点号匹配换行符”必须关闭）

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

刚刚这个正则表达式可以匹配至少包含 3 个单词 “one”、“two” 或 “three” 中至少一个的任意一行内容。单词两边的圆括号有两个目的。首先，它们会限制多选操作的作用范围；其次，它们会把实际出现在该行中的单词捕获到向后引用 #1 中。如果该行包含超过一个这些单词，那么向后引用会包含最靠右边的那个单词。这是因为出现在圆括号之前的星号量词是贪心的，从而会把点号扩展为匹配尽可能多的文本。如果你把星号变成懒惰的，比如使用 `^.*?\b(one|two|three)\b.*$`，那么向后引用 #1 中就会包含最左边出现的那个属于列表中的单词。

要找到必须包含多个单词的文本行，需要使用顺序环视：

```
^(?=.*?\bone\b)(?=.*?\btwo\b)(?=.*?\bthree\b).+$
```

正则选项：不区分大小写、`\^` 和 `\$` 匹配换行处（“点号匹配换行符”必须关闭）

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

这个正则表达式使用肯定型顺序环视来匹配在文本行中任意地方包含 3 个必需单词的文本行。在顺序环视决定了该文本行满足需求之后，位于结尾处的 `.+$` 会被用来实际上匹配该文本行。

## 参见

实例 5.11 讲解了如何匹配不包含某个特定单词的整行内容。

## 5.11 匹配不包含某个单词的整行

### 问题描述

你想要匹配不包含单词 `ninja` 的整行内容。

### 解决方案

```
^(?:(?!ninja\b).)*$
```

正则选项：不区分大小写、`^` 和 `$` 匹配换行处（“点号匹配换行符”必须关闭）

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

为了匹配不包含某些内容的一行，需要使用否定型的顺序环视（参考实例 2.16 中的讲解）。注意在这个正则表达式中，使用一个非捕获分组来重复了一个否定型顺序环视和一个点号。这样就可以确保正则式 `\bninja\b` 会在该行的每个位置都匹配失败。定位符 `<^>` 和 `<$>` 被放到正则表达式的两端，用来保证你所匹配到的内容是一整行。

应用这个正则表达式的选项决定了它是否会尝试匹配整个目标字符串，还是每次只匹配一行。如果使用了让 `<^>` 和 `<$>` 匹配换行处的选项，而关闭了让点号匹配换行符的选项，那么这个正则表达式就会按照前面的说明执行，并且会逐行进行匹配。如果你把这两个选项都反过来，那么正则表达式就会匹配不包含单词“`ninja`”的任意字符串。



#### 警告

在一行或者一个字符串中的每个位置检查否定型顺序环视的效率是相当低的。这个解决方案只打算被应用于只能使用一个正则表达式的情形，例如你在使用的应用程序不支持对它进行编程。如果可以使用编程实现，那么实例 3.21 中会讲解一种效率高很多的解决方案。

### 参见

实例 5.10 会讲解如何匹配包含某个特定单词的整行内容。

## 5.12 删除前导和拖尾的空格

### 问题描述

你想要删除一个字符串中的前导和拖尾空格。

## 解决方案

为了简单快速起见，最好用的万能解答是使用两次替换——一次删除前导空格，另外一次删除拖尾空格：

```
^\s+
正则选项：无（“^ 和 $ 匹配换行处”必须关闭）
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

\s+$
正则选项：无（“^ 和 $ 匹配换行处”必须关闭）
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

只需要把使用这两个正则表达式找到的所有匹配都替换为空串即可。实例 3.14 中会讲解如何这样做。有了这两个正则表达式，你只需要替换它们找到的第一个匹配，因为它们会一次就匹配到所有的前导或拖尾空格。

## 讨论

删除前导和拖尾空格是一个简单但是很常见的任务。前面刚刚给出的 2 个正则表达式都包含了三个部分：一个定位符来判断字符串的起始位置或结束位置（分别使用的是 `^` 和 `\$`），用来匹配任意空格字符的简写字符类 (`\s`)，以及把该字符类重复一次或多次的量词 (`+`)。

许多编程语言都会提供一个函数，通常被命名为 trim 或者 strip，可以用来去掉前导和拖尾空格。表 5-2 中给出了在各种编程语言中如何使用这种内置的函数或方法。

表 5-2 删除前导和拖尾空格的标准函数

编程语言	用法示例
C#、VB.NET	String.Trim([chars])
Java	string.trim()
PHP	trim(\$string)
Python、Ruby	string.strip()

JavaScript 和 Perl 在它们的标准函数库中不存在等价的函数，但是你可以很容易地创建自己的函数来完成同样的任务。

在 Perl 中：

```
sub trim {
    my $string = shift;
    $string =~ s/^\s+//;
    $string =~ s/\s+$//;
    return $string;
}
```

在 JavaScript 中：

```
function trim (string) {  
    return string.replace(/^\s+/, '').replace(/\s+$/, '');  
}  
// Alternatively, use this to make trim a method of all strings:  
String.prototype.trim = function () {  
    return this.replace(/^\s+/, '').replace(/\s+$/, '');  
};
```



### 提示

在 Perl 和 JavaScript 中，除了通常被当作是空白的空格、制表符、换行和回车字符之外，`\s`会匹配根据 Unicode 标准被定义为任意空白字符。

## 变体

构造一个正则表达式来帮助你修整(trim)一个字符串中的空格实际上可以有多种方式。然而，在处理较长的字符串时（也就是当性能最为重要的时候），它们的速度无一例外地都比不上采用两个简单的替换操作。下面会给出一些你可能会遇到的常见的解决方案。它们都是用 JavaScript 写的，因为 JavaScript 中没有“点号匹配换行符”的选项，所以这些正则表达式中使用`\s\S`来匹配任意单个字符，其中包括了换行符。在其他编程语言中，你可以使用一个点号来替代，并且要打开“点号匹配换行符”的选项。

```
string.replace(/^\s+|\s+$/g, '')
```

这个很可能是最为常见的解决方案。它通过多选结构（参见实例 2.8）组合了两个简单正则式，并且使用了`/g`（全局）选项来替换所有的匹配，而不是只替换第一个匹配（如果在字符串中同时包含前导和拖尾空格，它会匹配两次）。这并不是一个很差的解决方案，但是在处理长字符串的时候，它的速度会比使用两个简单替换要慢一些。

```
string.replace(/^\s*([\s\S]*?)\s*$/, '$1')
```

这个正则表达式的工作原理是：匹配整个字符串，并且捕获从第一个到最后一个非空白字符（如果存在），把它放到向后引用 #1 中。通过把整个字符串替换为向后引用 #1，你就得到了一个修整好的字符串。

这个方法在概念上是很简单的，但是在捕获分组中使用的懒惰量词会让正则引擎不得不做许多额外工作，因此也会造成这种方法在处理长的目标字符串时速度较慢。在匹配过程中，当正则引擎进入捕获分组之后，懒惰量词要求字符类`\s\S`被重复的次数尽量少。因此，正则引擎会每次匹配一个字符，在每个字符之后都要停顿来尝试匹配剩余的模式`(\s*$)`。如果因为在字符串的当前位置之后依然含有非空白字符而匹配失败，那么正则引擎会再多匹配一个字符，然后接着尝试匹配模式的剩余部分。

```
string.replace(/^\s*([\s\S]*\S)?\s*$/, '$1')
```

这个正则表达式与上一个比较类似，但是基于性能的考虑，它把懒惰量词替换成贪心量词。为了确保捕获分组依然只会匹配到最后一个非空白字符，我们使用了一个必需的拖尾 `\S`。然而，因为这个正则式必须能够匹配只包含空白的字符串，所以整个捕获分组被添加了一个拖尾的问号量词，从而变成了可选的。

我们再来退一步看一下它是如何工作的。这里，`\[\s\S]*` 中的贪心星号量词会重复模式“任意字符”直到字符串的结束。正则引擎随后会从字符串结尾开始一次一个字符回溯，直到它能匹配后面的 `\S`，或者直到它回溯到了在捕获分组中匹配到的第一个字符。除非是后面的空白字符个数比到它为止的所有文本中的字符还多，这个正则式通常会比上一个使用懒惰量词的解决方案速度要快。当然，它还是无法与使用两个简单替换的性能相比。

```
string.replace(/\^\s*(\S*(?:\s+\S+)*)\s*\$/,'$1')
```

这是一个相对比较常见的方法，因此我们也把它包含进来作为一个对比。其实并不存在好的理由需要使用这个正则式，因为它比前面给的解决方案的速度都要慢。它与前两个正则式的类似之处是它也会匹配整个字符串，然后把它替换为你想要保留的部分，但是由于内层的非捕获分组只会一次匹配一个单词，因此正则表达式引擎就需要执行许多独立的步骤。当你用它来修整较短字符串的时候，性能的影响可能并不明显，但是如果处理包含大量单词的非常长的字符串，那么这个正则式就会成为一个性能瓶颈。

有些正则表达式实现中会包含比较聪明的优化，从而改变这里所讲解的内部匹配过程，因此会使这里所给的方法的实际执行比我们所讲解的要更快或更慢一些。不过，使用两个替换的简单方式总是能够对不同字符串长度和各种字符串内容提供一贯可靠的性能，因此它总的来说是最好的全面解决方案。

## 参见

实例 5.13。

## 5.13 把重复的空白替换为单个空格

### 问题描述

作为对用户输入或者其他数据的一个清理程序，你想要把重复的空白字符都替换为单个空格。所有的制表符、换行或其他空白都应该被替换成一个空格。

### 解决方案

要实现下面这些正则表达式中的任意一个，只需要把所有匹配简单替换为单个空格字符。实例 3.14 中可以找到完成这样任务的代码示例。

## 清除任意空白字符

```
\s+  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 清除水平空白字符

```
[•\t]+  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 讨论

一种常见的文本清理程序是把所有重复的空白字符都替换为单个空格。例如，在HTML中，当要显示页面的时候，重复的空白会被简单地忽略掉（其中会有几处例外），因此把重复空白删除会有助于减少页面文件大小，而不会带来任何负面效果。

## 清除任意空白字符

在这个解答中，任意的空白字符（换行、制表符、空格等）序列都会被替换为单个空格。由于量词会重复空白字符类（`\s`）一次或多次，所以即使是单个的制表符字符也会被替换为一个空格。如果你把`\t`替换成`\{2,\}`，那么只有两个或者更多空白字符组成的序列才会被替换。这样可能会使替换需要得较少，因此能够提高性能，但是它也可能会造成制表符字符或者换行符被保留，而没有替换成空格字符。因此，哪种方式更好要取决于你想要达成什么样的目标。

## 清除水平空白字符

这个正则表达式同上一个几乎是完全一样的，唯一的例外是它没有处理换行符。只有制表符和空格会被替换掉。

## 参见

实例 5.12。

## 5.14 对正则表达式元字符进行转义

### 问题描述

你想要把用户提供的，或者是来自其他来源的字面字符串用作一个正则表达式中的一部分。然而，你想要在把该字符串嵌入到正则表达式之前，将该字符串中的所有正则表达式元字符都进行转义，这样可以避免造成不想要的后果。

## 解决方案

如果一个字符可能会在正则表达式中拥有特殊含义，那么在它之前添加一个反斜杠，就可以安全地使用得到的模式来匹配一个字面的字符序列。在本书讲到的所有编程语言中，除了 JavaScript 之外的所有语言都包含一个内置的函数或方法，可以用来执行这个任务（参见表 5-3）。然而，为了讲解的完整性，即使在已经拥有现成解决方案的语言中，我们也会讲解如何使用你自己的正则表达式来完成这个任务。

### 内置的解决方案

表 5-3 列出了用来解决这个问题的内置函数。

表 5-3 对正则表达式元字符进行转义的内置解决方案

语 言	函 数
C#、VB.NET	Regex.Escape(str)
Java	Pattern.quote(str)
Perl	quotemeta(str)
PHP	preg_quote(str, [delimiter])
Python	re.escape(str)
Ruby	Regexp.escape(str)

注意在这个列表中并没有包含 JavaScript，因为它并不包含用于这个目的的内置函数。

### 正则表达式

虽然如果可能的话，读者最好还是应该使用内置的解决方案，但是你也完全可以使用下面的正则表达式与合适的替代字符串（随后会介绍）来构造自己的解决方案。你只需要确保会替换所有的匹配，而不是只替换第一个匹配。实例 3.15 中给出了把匹配替换为一个包含向后引用的字符串的代码示例。这里需要一个向后引用来把匹配到的特殊字符与前面添加的反斜杠一起加回到字符串中：

[ [\]{}()^+?.\\|^\$\\-, \s]

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 替代



#### 提示

下面的替代字符串中包含一个字面的反斜杠字符。这里的字符串中没有显示可能会需要用来对反斜杠进行转义的额外的反斜杠，这在有些编程语言中使用字符串字面量时是必须的。关于替代文本流派的更多信息，请参考实例 2.19。

```
\$&
    替代文本流派: .NET、JavaScript

\\$&
    替代文本流派: Perl

\\$0
    替代文本流派: Java、PHP

\\\"0
    替代文本流派: PHP、Ruby

\\\"&
    替代文本流派: Ruby

\\\"g<0>
    替代文本流派: Python
```

## JavaScript 函数示例

下面的代码示例会展示如何把正则表达式和替代文本用在一起，在 JavaScript 中创建一个静态的方法 RegExp.escape：

```
RegExp.escape = function (str) {
    return str.replace(/[\[\]\{\}\*\+\?\.\\\\|^$\,-,\#\s]/g, "\\$&");
};

// Test it out...
var str = "Hello.World?";
var escaped_str = RegExp.escape(str);
alert(escaped_str == "Hello\\.World\\?"); // -> true
```

## 讨论

这个实例中的正则表达式把所有的正则元字符都放到一个字符类中。我们再来看一下所有这些字符，并且探讨一下如何对它们进行转义。其中有些字符会比其他字符更加容易处理。

[] {} ()

<[> 和 <]> 会用于创建字符类。<{> 和 <}> 会用于创建区间量词，并且还可以与其他一些特殊结构共同使用，比如 Unicode 属性。<(> 和 <)> 被用于分组、捕获和其他特殊结构。

\* + ?

这三个字符是用来分别把之前元素重复 0 次或多次、1 次或多次以及 0 次或 1 次的量词。问号也会被用在一个左圆括号之后来创建特殊的分组和其他结构（在 Perl 5.10 和 PCRE 7 中的星号也是如此）。

. \ |

点号匹配一行或一个字符串中的任意字符，反斜杠转义一个特殊字符，或者让某个字面字符变成特殊字符，竖线表示多个可替换的选项。

^ \$

脱字符和美元符号是用来匹配一行或字符串的开始或结束的定位符。脱字符还可以用来对字符类取反。

下面的这些正则表达式匹配的字符只有在特殊情况下才拥有特殊含义。为了谨慎起见，作者也把它们包含在这个列表中。

连字符会在字符类中创建一个范围。它在这里被转义是为了避免它被放在字符类中间的时候，会不小心创建一个范围。记住如果你在字符类中嵌入文本的话，得到的正则式不会匹配嵌入的字符串，而只会匹配在所嵌入的字符串中的任意一个字符。

逗号被用在诸如 `{1,5}` 这样的区间量词中。因为大多数正则表达式流派在花括号不能构成合法量词的时候，会把它们都当作字面字符，所以也可能会出现（虽然不大可能）在没有量词的地方，因为插入了字面文本，但由于没有对逗号进行转义，从而创建了不应该有的量词。

&

在这个列表中包括了 and 符号 (`&`)，因为在 Java 中，连续的两个 and 符号会被用于字符类交集。在其他语言中，完全可以把 and 符号从需要转义的字符列表中去掉，但是留下来当然也不会有问题。

# 和空白

井号和空白（由 `\s` 匹配）只有在宽松排列选项打开的时候才是元字符。然而，对它们转义不会带来任何问题。

在替代文本中，5 个记号（`«$&»`、`«\&»`、`«$0»`、`«\0»` 或 `«\g<0>»`）中的一个被用来恢复匹配的字符，并且在前面添加一个反斜杠。在 Perl 中，`$&` 实际上是一个变量，在任何正则表达式中使用它会对所有正则表达式带来全局的性能影响。如果在你的 Perl 程序的其他地方也用到 `$&`，那么可以随意使用这个变量，因为你已经在别的地方付出了代价。否则，最好还是把整个正则式包在一个捕获分组中，在替代中使用 `$1` 而不是 `$&`。

## 变体

在实例 2.1 中的“块转义”小节中讲到过，你可以使用 `\Q...\E` 在正则式中创建一个块转义序列。然而，只有 Java、PCRE 和 Perl 中才支持块转义，而且即使在这些语言中，

块转义的功能也不是十分安全。为了尽量的安全，你还是应当对想要嵌入到正则式中的字符串中任意的\& 进行转义。在大多数情形下，很可能还是会使用对所有正则式元字符都转义，并且可以应用于不同语言的方法，这样更为简单。

## 参见

实例 2.1 讨论了如何匹配字面字符；其中列出的需要被转义的字符比本实例中要少，这是因为在宽松排列模式下或是把自己放到一个更长的任意模式中的时候，它并不必考虑可能会需要进行转义的字符。

正则表达式是用来处理文本的，因此它无法理解人给数字字符组成的字符串所赋予的数字含义。对于正则表达式来说，56 指的不是一个数“五十六”，而是一个字符串中包含了两个数字字符 5 和 6。正则引擎会知道它们是数字，因为可以使用简写字符类 `\d` 匹配它们（参见实例 2.3）。但是仅此而已。正则引擎并不知道 56 还拥有更深层的含义，就好像对它来说，`:-)` 只是可以用 `\p{P}{3}` 匹配的三个标点字符而已。

但是数字却是你很可能要处理的一些最为重要的输入，而且有时候需要在一个正则表达式内部来处理它们，而不是当每次需要回答像“这个数字是否属于 1~100 的范围之内？”这样的问题时，总是把它们传递给某种传统的编程语言来解决。因此，我们会用一整章的内容来讲解如何使用正则表达式匹配各种数字。一开始会介绍一些貌似微不足道的例子，但是实际上它们讲解的是最重要的基本概念。随后的实例则会在假设你掌握了这些基本概念的前提下，讲解更为复杂的正则表达式。

## 6.1 整数

### 问题描述

你想要在一大段文本中查找各种十进制整数，或者是检查一个字符串变量中保存的是不是一个十进制整数。

### 解决方案

在一大段文本中查找任意的十进制正整数：

`\b[0-9]+\b`

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

检查一个文本字符串中是否只包含了一个十进制正整数：

```
\A[0-9]+\Z  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby  
  
^[0-9]+\$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python
```

在一大段文本中查找独立存在的任意十进制正整数：

```
(?<=^|\s)[0-9]+(?=$|\s)  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby 1.9
```

在一大段文本中查找独立存在的任意十进制正整数，允许在正则匹配中包含前导的空白：

```
(^|\s)([0-9]+)(?=$|\s)  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

查找任意的十进制整数，前面可以包含一个可选的前导正号或负号：

```
[+-]?\b[0-9]+\b  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

检查一个文本字符串中是否只包含了一个带可选符号的十进制整数：

```
\A[+-]?(0-9)+\Z  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby  
  
^[+-]?(0-9)+\$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python
```

查找带可选符号的任意十进制整数，允许在整数和符号之间包含空格，但是如果不含符号，则不允许包含前导空格：

```
([+-]•*)?\b[0-9]+\b  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 讨论

一个整数是连续出现的一个或多个 0~9 的数字序列。使用一个字符类（实例 2.3）和一个量词（实例 2.12），我们就可以很容易地表示一个整数：`<[0-9]+>`。



### 提示

我们更倾向于使用明确的范围 `<[0-9]>`，而不是简写字符类 `\d`。在 .NET 和 Perl 中，`\d` 会匹配任意文字中的任意数字，但是 `<[0-9]>` 只会匹配位于 ASCII 字符表中的 10 个数字。如果你知道在目标文本中不会包含任何非 ASCII 数字的话，那么也可以选择使用 `\d` 而不是 `<[0-9]>`，这样可以少敲几下键盘。

如果你不知道在目标文本中是否会包含 ASCII 字符表之外的数字，那么就需要认真考虑你想要拿正则匹配来做什么以及用户的期望是什么，以此来决定应当使用 `\d` 还是 `<[0-9]>`。如果你打算把正则表达式匹配到的文本转换为一个整数，那么就需要检查所使用的编程语言中的字符串到整数的转换函数是否可以解释非 ASCII 的数字。以本国文字来编写文档的用户可能会期望你的软件能够识别在其本国文字中的数字。

除了必须是一串数字字符之外，一个整数还必须是独立存在的。例如，A4 是一种纸的尺寸，而不是一个整数。要保证你的正则表达式只匹配到纯数字，可以采用以下几种不同的方式。

如果想要检查你的字符串中是否只包含了一个数字，那么只需要在正则表达式的两边添加字符串开始和结束的定位符。最好使用 `\A` 和 `\Z`，因为它们的含义总是不会变的。不幸的是，JavaScript 并不支持这两个定位符，因此你需要在 JavaScript 中采用 `\^` 和 `\$`，同时要保证你没有使用会让脱字符和美元符号匹配换行处的标志/m。在 Ruby 中，脱字符和美元符号总是会匹配换行处，从而无法使用它们来可靠地强制正则表达式匹配整个字符串。

当你在一大段文本中查找数字的时候，则可以采用单词边界（实例 2.6）作为比较容易的解决方案。当你把它们放到一个匹配数字的正则记号之前或之后的时候，单词边界会确保在匹配到的数字之前或之后没有其他单词字符。例如，`\b4\b` 会匹配到 A4 中的 4。`\b4\b` 同样会匹配到其中的 4，这是因为在 4 之后并没有任何单词字符。`\b4\b` 和 `\b4\b\b` 则都不会在 A4 中匹配到任何内容，因为在 A 和 4 两个单词字符之间 `\b` 会匹配失败。在正则表达式中，单词字符包括字母、数字和下划线。

如果在你的正则表达式中要把正负号或空格这些非单词字符也包括进来，那么就必须小心处理单词边界的位置。要想匹配到 +4，但是却不能匹配 +4B，那么可以使用 `\b+4\b`，但是不能用 `\b\b+4\b`。`\b\b+4\b` 会匹配到 3+4 中的 +4，这是因为 3 是一个单词字符，而 + 不是单词字符。

`\b+4\b` 只需要使用一个单词边界。在 `\b+4\b` 中的第一个 `\b` 是多余的。当这个正则式匹配成功的时候，第一个 `\b` 总是位于一个 + 和 4 之间，从而永远也不会排除掉任何情况。只有当正号是可选的时候，第一个 `\b` 才是重要的。`\b+?4\b` 不会匹配在 A4

中的 4，而 `\+?4\b` 则会匹配到。

单词边界并不总是正确的解答。下面我们以目标文本 \$123,456.78 为例。如果你使用正则表达式 `\b[0-9]+\b` 在这个字符串之上重复进行匹配，它会匹配到 123、456 和 78。美元符号、逗号和点号都不是单词字符，所以单词边界会匹配一个数字和任意这 3 个字符之间的位置。有时候这可能是你想要的结果，而有时候则不然。

如果你只是想找到两边是空白或者是字符串开始和结束的整数，那么就需要使用环视，而不能只使用单词边界。`(?=$\s)` 会匹配字符串的结尾或者在一个空白字符（空白字符包括换行符）之前的位置。`(?<^$\s)` 会匹配字符串的开始，或者是空白字符之后的位置。你可以把其中的 `\s` 替换为一个字符集，用来匹配你想要在数字之前或之后出现的任意字符集合。要了解环视的工作原理，请参考实例 2.16。

JavaScript 和 Ruby 1.8 中不支持逆序环视。你也可以不使用逆序环视，而使用一个普通分组来检查数字是否出现在字符串的开头，或者在它之前是否是空白。这样做的缺点是如果数字不是从字符串开头开始，那么空白字符会被包含在总的正则匹配中。对于这个问题的一个简单解决方案是把正则式中用来匹配数字的那个部分放到一个捕获分组中。在前面“解决方案”小节中的第 5 个正则表达式在第一个捕获分组中会捕获空白字符，而在第二个捕获分组中包含的是匹配到的整数。

## 参见

实例 2.3 和实例 2.12。

## 6.2 十六进制数字

### 问题描述

你想要在一大段文本中查找各种十六进制的整数，或者是检查一个字符串变量中保存的是不是一个十六进制整数。

### 解决方案

在一大段文本中查找任意十六进制数：

`\b[0-9A-F]+\b`

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

`\b[0-9A-Fa-f]+\b`

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

检查一个文本字符串中保存的是否只是一个十六进制数：

\A[0-9A-F]+\Z

正则选项：不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

^[0-9A-F]+\$

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

查找一个带 0x 前缀的十六进制数：

\b0x[0-9A-F]+\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个带&H 前缀的十六进制数：

&H[0-9A-F]+\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个带 H 后缀的十六进制数：

\b[0-9A-F]+H\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个十六进制字节值（byte value），或者一个 8bit 数：

\b[0-9A-F]{2}\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个十六进制字值（word value），或者一个 16bit 数：

\b[0-9A-F]{4}\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个十六进制双字值（double word value），或者一个 32bit 数：

\b[0-9A-F]{8}\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个十六进制四字值（quad word value），或者一个 64bit 数：

\b[0-9A-F]{16}\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个由十六进制字节组成的字符串（即偶数个十六进制数字组成的字符串）：

\b(?:[0-9A-F]{2})+\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

使用正则表达式来匹配十六进制整数的技巧与匹配十进制整数是一样的。唯一的区别是用来匹配单个数字的字符类中现在需要包含字母 A~F。你还必须要考虑这些字母应该采用大写还是小写形式，或者是大小写均可。上面所给的正则表达式都支持大小写均可的形式。

默认来说，正则表达式是区分大小写的。`\[0-9a-f]` 只能匹配小写形式的十六进制数字，而 `\[0-9A-F]` 只能匹配大写形式的十六进制数字。要想允许大小写均可，那么可以使用 `\[0-9a-fA-F]`，或者是打开正则表达式中不区分大小写的选项。实例 3.4 中讲解了如何在编程语言中打开该选项。上述解答中的第一个正则式出现了两次，分别采用了实现大小写无关的两种不同方式。后面的正则表达式则都只列出了第二种方式。

如果你只想要在十六进制数中允许出现大写字母，那么应该把不区分大小写的选项关掉，就可以使用同样的正则式。要想只允许出现小写字母，那么需要关掉不区分大小写选项，并且把 `\A-F` 都替换为 `\a-f`。

正则表达式 `\((?:\[0-9A-F]\{2}\))+` 会匹配偶数个数的十六进制数字。`\[0-9A-F]\{2}` 正好会匹配 2 个十六进制数字。`\((?:\[0-9A-F]\{2}\))+` 则会在此基础上重复 1 次或多次。其中的非捕获分组（参见实例 2.9）是必需的，因为加号量词需要重复的是字符类和量词 `\{2\}` 组合的整体。在 Java、PCRE 和 Perl 5.10 中，`\[0-9]\{2\}+` 也不会产生语法错误，但是所产生的效果则不是你想要的。额外的 `\(+\)` 会把量词 `\{2\}` 变成占有量词。这并不会产生任何效果，因为 `\{2\}` 本来也无法重复少于两次。

其中的几种解决方案讲解了如何要求十六进制数必须包含一个通常用来标识十六进制数的前缀或者后缀。这些前缀或后缀被用来区分十进制数与那些恰好只包含十进制数字的十六进制数。例如，10 可以是在 9 和 11 之间的十进制数，或者是在 F 和 11 之间的十六进制数。

大多数解决方案中都包含单词边界（参见实例 2.6）。按照所给的方式使用单词边界可以从一大段文字中找到其中的数字。注意使用 `\&H` 前缀的正则式在开始并没有使用单词边界。这是因为 `\&` 符号并不是一个单词字符。如果我们在这个正则表达式的开始放一个单词边界，那么它就只能找到紧跟在单词字符之后的十六进制数。

如果你想要检查你的字符串中保存的是否刚好是一个十六进制数，那么可以简单地在你的正则式两边放上字符串开始和字符串结束定位符。你最好的选择是使用 `\A` 和 `\Z`，因为它们的含义总会保持不变。在 JavaScript 中，需要使用 `\^` 和 `\$`，并且要确保没有使用会让脱字符和美元符号匹配换行处的标志/m。在 Ruby 中，脱字符和美元符号总是会匹配换行处，所以你无法可靠地使用它们来强制一个正则表达式匹配整个字符串。

## 参见

实例 2.3 和实例 2.12。

## 6.3 二进制数

### 问题描述

你想要在一大段文本中查找其中的二进制数，或者是检查一个字符串变量中保存的是否是一个二进制数。

### 解决方案

在一大段文本中查找二进制数：

\b[01]+\b

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

检查一个文本字符串中是否只保存了一个二进制数：

\A[01]+\Z

正则选项：无

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

^[01]+\\$

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

查找带后缀 B 的二进制数：

\b[01]+B\b

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个二进制字节值（byte value）或是一个 8bit 数：

\b[01]{8}\b

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个二进制字值（word value）或是一个 16bit 数：

\b[01]{16}\b

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

查找一个字节串（也就是多个 8bit 数组成的字符串）：

\b(?:[01]{8})+\b

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

所有这些正则表达式中使用的技巧都已经在前面两个实例中讲解过。关键的区别是这里的每个数字只能是 0 或者 1。使用一个只包含两个字符的字符类`[01]`，我们就可以很容易地匹配这个模式。

## 参见

实例 2.3 和实例 2.12。

## 6.4 删 除 前 导 0

### 问题描述

你想要匹配一个整数，然后或者返回不包含任何前导 0 的整数，或者是删除所有的前导 0。

### 解决方案

#### 正则表达式

```
\b0*([1-9][0-9]*|0)\b  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

#### 替代文本

```
$1  
替代文本流派: .NET、Java、JavaScript、PHP、Perl  
\1  
替代文本流派: PHP、Python、Ruby
```

#### 在 Perl 中获取数字

```
while ($subject =~ m/\b0*([1-9][0-9]*|0)\b/g) {  
    push(@list, $1);  
}
```

#### 在 PHP 中删除前导 0

```
$result = preg_replace('/\b0*([1-9][0-9]*|0)\b/', '$1', $subject);
```

## 讨 论

我们使用了一个捕获分组来把数字同它前面的 0 分开。在该分组之前，`\b0*` 会匹配所

有的前导 0（也可以不存在）。在分组中，`<[1-9][0-9]*>` 会匹配包含 1 个或多个数字的一个整数，它的第一个数字不能是 0。这个数只有在整个数的值为 0 的时候，才能以 0 开头。其中的单词边界用来保证我们不会匹配到部分数字，这在实例 6.1 中已经有过讲解。

要想得到目标文本中所有不含前导 0 的数字的一个列表，需要按照实例 3.11 中的讲解对所有正则匹配进行遍历。在循环中，按照实例 3.9 中的讲解，获取由第一个（也是唯一一个）捕获分组匹配到的文本。随后的解决方案中讲解了如何在 Perl 中完成这个任务。

把前导 0 去掉也可以很容易地使用查找和替换操作来完成。在我们的正则式中包含一个捕获分组，它可以把整数与其前导 0 分离开来。如果我们把整个正则匹配（包含前导 0 的整数）替换为第一个捕获分组匹配到的文本，那么实际上也就是删除了所有的前导 0。前面的解决方案中给出了在 PHP 中的代码示例。实例 3.15 中讲解了如何在其他编程语言中完成同样的任务。

## 参见

实例 3.15 和实例 6.1。

## 6.5 位于某个特定范围之内的整数

### 问题描述

你想要匹配位于某个特定取值范围内的一个整数。你希望正则表达式可以准确限制数值范围，而不只是对其中的数字个数进行限制。

### 解决方案

1~12（小时或者月份）：

```
^(1[0-2]|1[1-9])$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

1~24（小时）：

```
^(2[0-4]|1[0-9]|1[1-9])$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

1~31（每月中的天次）：

```
^(3[01]|12)[0-9]|1[1-9])$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

1~53 (每年中的周次):

`^(5[0-3]|1-4)[0-9]|1-9)$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~59 (分钟或秒钟):

`^[1-5]?[0-9]$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~100 (百分数):

`^(100|[1-9]?[0-9])$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

1~100:

`^(100|[1-9][0-9]?)$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

32~126 (可打印的 ASCII 字符):

`^(12[0-6]|1[01][0-9]|4-9)[0-9]|3[2-9])$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~127 (非负有符号字节数):

`^(12[0-7]|1[01][0-9]|1-9)?[0-9])$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

-128~127 (有符号字节数):

`^(12[0-7]|1[01][0-9]|1-9)?[0-9]|-(12[0-8]|1[01][0-9]|1-9)?[0-9]))$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~255 (无符号字节数):

`^(25[0-5]|2[0-4][0-9]|1[0-9]{2}|1-9)?[0-9])$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

1~366 (一年中的天次):

`^(36[0-6]|3[0-5][0-9]|12)[0-9]{2}|1-9)[0-9]?)$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

1900~2099 (年份):

`^(19|20)[0-9]{2}$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~32767 (非负有符号字数):

`^(3276[0-7]|327[0-5][0-9]|32[0-6][0-9]{2}|3[01][0-9]{3}|[12][0-9]`

`{4})|←`

`[1-9][0-9]{1,3}|[0-9])$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

-32768~32767 (有符号字数):

`^(3276[0-7]|327[0-5][0-9]|32[0-6][0-9]{2}|3[01][0-9]{3}|[12][0-9]{4})|←`

`[1-9][0-9]{1,3}|[0-9])|-(3276[0-8]|327[0-5][0-9]|32[0-6][0-9]{2})|←`

`3[01][0-9]{3}|[12][0-9]{4}|[1-9][0-9]{1,3}|[0-9]))$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~65535 (无符号字数):

`^(6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4})|←`

`[1-9][0-9]{1,3}|[0-9])$`

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

前面几个实例会匹配包含任意数目的数字的一个整数，或者是包含特定数目的数字的一个整数。它们允许在该整数的所有数字中都可以出现所有可能的数字范围。

匹配位于某个特定范围之内的整数，例如 0~255 之间的一个整数，就无法采用正则表达式来简单实现了。你不能使用 `<[0-255]>`。如果这样用，那么就根本无法匹配 0~255 之间的一个整数。这个字符类与 `<[0125]>` 是等价的，因此它只能匹配数字 0、1、2 和 5 之中的任意一个。



### 提示

因为这些正则表达式都比较长，所以所有解决方案中都使用了定位符来使得正则式可以适用于检查一个字符串（例如用户输入）中包含的是不是一个可接受的数。实例 6.1 中讲到，为了一些其他的目的，你可以不使用这些定位符，而是使用单词边界或环视。在这个讨论中所给的正则式都不包含任何定位符，这样使得我们可以更好地关注如何处理数值范围。如果你想要使用其中的任何一个正则表达式，那么就需要添加定位符或者单词边界，以确保你的正则表达式不会匹配到属于更长数字的一部分。

正则表达式会一个字符一个字符来进行处理。如果我们想要匹配包含多于一个数字的整数，那么就必须罗列出所有可能的数字组合。最重要的构造单元是字符类（实例 2.3）和多选结构（实例 2.8）。

在字符类中，我们可以使用单个数字的范围，例如 `<[0-5]>`。这是因为 0~9 的数字字符在 ASCII 字符表和 Unicode 字符表中占据的都是连续的位置。`<[0-5]>` 会匹配其中的 6 个字符中的一个，而 `<[j-o]>` 和 `<[\x09-\x0E]>` 也都会匹配不同范围之内的 6 个字符中的一个。

当用文本来表示一个数值范围的时候，它会包含各种不同的位置。每个位置会允许特定范围内的数字。有些范围拥有固定数量的位置，例如 12~24。其他范围则可能会拥有可变数量的位置，例如 1~12。每个位置所允许出现的数字范围则有可能与其他位置的数字有关或无关。在 40~59 的范围内，每个位置之间是无关的。而在范围 44~55 中，位置之间则是存在关系的。

最容易实现的范围是那些包含固定数量的无关位置的范围，例如 40~59。要把这样的范围编码为一个正则表达式，你所需要做的只是把几个字符类串起来。为每个位置采用一个字符类，然后指定每个位置所允许的数字范围。

`[45][0-9]`

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

范围 40~59 要求必须包含两个数字。因此我们只需要两个字符类。第一个数字必须是 4 或者 5，因此可以使用字符类 `<[45]>`。第二个数字可以是任意的 10 个数字之一，因此我们使用的字符类是 `<[0-9]>`。



### 提示

我们也可以不使用 `<[0-9]>`，而是使用简写字符类 `\d`。为了与其他字符类保持一致，我们使用了明确的范围 `<[0-9]>`，这样也有助于提高正则表达式的可读性。如果你使用的是类似 Java 这样的编程语言，其中要求在字面字符串中对反斜杠转义，那么减少在正则式中的反斜杠同样也会大有帮助。

在 44~55 的范围中的数字同样需要使用两个位置，但是它们之间不是无关的。第一个数字必须是 4 或者 5。如果第一个数字是 4 的话，那么第二个数字就必须在 4 和 9 之间。这样可以覆盖 44~49 的所有整数。如果第一个数字是 5 的话，那么第二个数字必须在 0~5 之间。这样就可以覆盖 50~55。要创建这个一个正则表达式，我们只需要使用多选结构来把这两个范围组合起来：

`4[4-9]|5[0-5]`

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

通过使用多选结构，我们告诉正则引擎它可以匹配 `<4[4-9]>` 或者 `<5[0-5]>`。多选操作符

在所有正则操作符中拥有最低的优先级，因此我们并不需要对数字添加分组，比如采用 `\((4[4-9])|(5[0-5])\)`。

你可以使用多选结构来组合任意多个范围。例如，范围 34~65 也拥有两个相关的位置。第一个数字必须是 3~6。如果第一个数字是 3，那么第二个必须是 4~9。如果第一个数字是 4 或 5，那么第二个可以是任意数字。如果第一个数字是 6，那么第二个必须是从 0~5：

`3[4-9]|4[5][0-9]|6[0-5]`

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

使用多选结构，我们不仅可以将相关联的位置划分成多个包含无关位置的范围，还可以使用多选结构来把包含可变数量位置的范围划分成多个包含固定数量位置的范围。例如范围 1~12 可以包含一个或两个位置。我们把它划分为包含一个位置的范围 1~9，以及包含两个位置的范围 10~12。在这两个范围中的位置则都是独立的，因此我们不需要对它进行更进一步的划分：

`1[0-2]|1[1-9]`

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

我们把包含两个数字的范围放到包含单个数字的范围之前。这样做是有原因的，因为正则表达式引擎是贪心的。它会从左向右对所有选择分支进行扫描，而一旦产生匹配则会立即停止。如果你的目标文本是 12，那么 `<1[0-2][1-9]>` 会匹配到 12，而 `<[1-9]1[0-2]>` 则只能匹配到 `<1>`。这是因为第一个选择分支 `<1-9>` 会被最先尝试。因为这个选择分支完全可以只匹配 1，因此正则引擎也就不会再尝试检查第二个选择分支 `<1[0-2]>` 是否会产生“更好”的匹配结果。

### 有些正则引擎不是贪心的

POSIX 兼容的正则引擎和 DFA 正则引擎并不会遵守这个规则。它们会尝试所有的选择分支，然后返回找到最长匹配的那个结果。然而，在本书中讨论的所有流派使用的都是 NFA 引擎，而它不需要按照 POSIX 的要求执行额外的查找。它们会告诉你 `<[1-9]1[0-2]>` 在 12 中能匹配到 1。

在实践中，你通常会在选择分支的列表中使用定位符或者单词边界。这样，选择分支的顺序就不再重要了。在本书中的所有正则流派，以及所有 POSIX “扩展” 正则表达式和 DFA 引擎中，`^(1[1-9]|1[0-2])$` 和 `^(1[0-2]|1[1-9])$` 都会在 12 中匹配到 1。其中的定位符会要求正则式必须匹配整个字符串，或者什么也不要匹配。DFA 和 NFA 的定义，请参考第 1 章中的“术语‘正则表达式’的历史”小节。

区间 85~117 中包含了两种不同长度的数字。85~99 是两位数，而 100~117 则是三位

数。在这两种区间之内的位置都是相互依赖的，所以我们还需要对它们进一步拆分。对于两位数的区间来说，如果第一位数字是 8，则第二位必须在 5 和 9 之间。如果第一个数字是 9，则第二个数字可以是任意数字。对于三位数的区间来说，第一位只允许出现数字 1。如果第二个位置上是数字 0，那么第三个位置可以是任意数字。但是如果第二个数字是 1，那么第三个数字必须位于 0 到 7 之间。这样我们就得到了四个区间：85~89、90~99、100~109 和 110~117。虽然这看起来似乎有些复杂，但是所采用的正则表达式则依然很直接的：

```
8[5-9]|9[0-9]|10[0-9]|11[0-7]
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

上面我们介绍的就是使用正则表达式来匹配整数区间的所有技巧：你只需要对区间进行简单的拆分，直到拆分之后的所有区间都只包含固定个数的彼此无关的数字为止。采用这种方式，总是能够得到一个易读、易维护的正确的正则表达式，虽然它看起来可能会显得有点儿啰嗦。

还可以使用一些额外的技巧来缩短正则表达式的长度。例如，使用上面介绍的方式，匹配 0~65536 的区间就需要采用如下的正则表达式：

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9][0-9]|6[0-4][0-9][0-9][0-9]|←  
[1-5][0-9][0-9][0-9][0-9]|[1-9][0-9][0-9][0-9][0-9]|[1-9][0-9][0-9][0-9]|←  
[1-9][0-9]|[0-9]
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

这个正则表达式是完全正确的，而且你也不可能再给出明显比它速度更快的正则式来完成同样的任务。这是因为在正则表达式引擎对该正则表达式进行编译的过程中，已经应用了所有可能的优化（例如，其中包含了多个以 6 开头的选择分支）。因此你也就没有必要再浪费时间为了提高它的速度而使之变得更为复杂。但是如果你觉得有必要，可以缩短这个正则表达式的长度，这样可以在依然保持其可读性的前提下，减少键盘输入。

其中的几个选择分支中包含了相邻的同一个字符类。你可以通过使用量词来消除这些重复内容。实例 2.12 中可以找到关于此的讲解。

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|←  
[1-9][0-9]{3}|[1-9][0-9]{2}|[1-9][0-9]|[0-9]
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

这个正则表达式中的 `<[1-9][0-9]{3}|[1-9][0-9]{2}|[1-9][0-9]>` 部分中包含了 3 个非常相似的选择分支，而且它们都拥有同一对字符类。唯一的区别是第二个字符类需要被重复的次数不同。因此我们可以很容易地把它们组合在一起：

<[1-9][0-9]{1,3}>.

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|←  
[1-9][0-9]{1,3}|[0-9]
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

如果再继续进行任何更多合并，就会影响到可读性。例如，你还可以从前 4 个选择分支中把开头的 6 都分离出来：

```
6(?:553[0-5]|55[0-2][0-9]|5[0-4][0-9]{2}|[0-4][0-9]{3})|[1-5][0-9]{4}|←  
[1-9][0-9]{1,3}|[0-9]
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

但是这个正则表达式实际长度比刚才反而增加了一个字符，这是因为我们不得不添加一个非捕获分组来把前导的 6 与其他的选择分支分隔开来。在本书讨论的所有正则流派中，你也不可能因此而得到任何性能上的提高。因为所有的正则流派都会在内部进行这样的优化。

## 参见

实例 2.8、实例 4.12 和实例 6.1。

## 6.6 在某个特定范围之内的十六进制数

### 问题描述

你想要匹配在某个特定数值范围之内的十六进制数。你希望正则表达式能够精确说明这个范围，而不只是对数字的个数进行限制。

### 解决方案

1~C (1~12：小时或月份)：

```
^[1-9a-c]$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

1~18 (1~24：小时)：

```
^(1[0-8]|[1-9a-f])$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

1~1F (1~31：一个月中的天次)：

```
^(1[0-9a-f]|[1-9a-f])$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

1~35 (1~53: 每年的周次):

`^(3[0-5]|12[0-9a-f]|1-9a-f)$`  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~3B (0~59: 分钟或秒钟):

`^(3[0-9a-b]|12)?[0-9a-f]$`  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~64 (0~100: 百分数):

`^(6[0-4]|1-5)?[0-9a-f]$`  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

1~64 (1~100):

`^(6[0-4]|1-5)[0-9a-f]|1-9a-f)$`  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

20~7E (32~126: 可打印的 ASCII 代码):

`^(7[0-9a-e]|2-6)[0-9a-f]$`  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~7F (0~127: 7 比特数):

`^[1-7]?[0-9a-f]$`  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~FF (0 到 255: 8 比特数):

`^[1-9a-f]?[0-9a-f]$`  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

1~16E (1~366: 一年中的天次):

`^(16[0-9a-e]|1[0-5][0-9a-f]|1-9a-f)[0-9a-f]?)$`  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

76C~833 (1900~2099: 年份):

`^(83[0-3]|8[0-2][0-9a-f]|7[7-9a-f][0-9a-f]|76[c-f])$`  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

0~7FFF: (0 到 32767: 15 比特数):

```
^([1-7][0-9a-f]{3}|[1-9a-f][0-9a-f]{1,2}|[0-9a-f])$  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

0~FFFF: (0~65535: 16 比特数):

```
^([1-9a-f][0-9a-f]{1,3}|[0-9a-f])$  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 讨论

采用正则表达式来匹配十进制数字范围和十六进制数字范围之间并不存在实质的区别。读者还可以按照上一个实例中所解释的方法，把一个区间拆分成多个区间，直到每个区间中都包含固定个数的位置，并且每个位置包含彼此无关的数字。然后，就只需要在每个位置使用一个字符类，最后把这些区间使用多选结构组合起来。

因为字母和数字在 ASCII 和 Unicode 字符表中占据的区域并不连续，因此你无法使用字符类 <[0-F]> 来匹配 16 个十六进制数字中的任意一个。虽然这个字符类实际上也能完成这个任务，但是它还会匹配到在 ASCII 字符表中位于数字和字母之间的一些标点符号。因此，我们需要在字符类中使用两个字符区间来表示：[0-9A-F]。

另外一个需要注意的问题是大小写敏感性。默认来说，正则表达式都是区分大小写的。因此，<[0-9A-F]> 只能匹配到大写形式的字符，<[0-9a-f]> 而只能匹配到小写字符。要想大小写都可以匹配，就需要采用 <[0-9A-Fa-f]>。

在每个字符类中都显式地明确指定大写和小写的字符区间很快就会让人感到乏味。因此，读者可以选择转而采用不区分大小写的选项。关于如何在你所选择的编程语言中打开正则选项，请参考实例 3.4。

## 参见

实例 2.8 和实例 6.2。

## 6.7 浮点数

### 问题描述

你想要匹配一个浮点数，并且可以分别指定该浮点数中的符号、整数部分、小数部分和指数部分是必需的、可选的或者是不允许的。你并不想用这个正则表达式来把其取值限制到某个具体范围之内，而是希望按照在实例 3.12 中的讲解，把这个任务留给过

程代码来完成。

## 解决方案

符号、整数部分、小数部分和指数部分都是必需的：

`^[-+]?[0-9]+\.\.[0-9]+[eE][-+]?[0-9]+$`  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

符号、整数部分和小数部分是必需的，但是不要指数部分：

`^[-+]?[0-9]+\.\.[0-9]+$`  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

符号可选，整数部分和小数部分是必需的，并且不要指数部分：

`^[-+]?[0-9]+\.\.[0-9]+$`  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

符号和整数部分是可选的，小数部分是必需的，并且不要指数部分：

`^[-+]?[0-9]*\.\.[0-9]+$`  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

符号、整数部分和小数部分是可选的。如果整数部分没有的话，那么小数部分是必需的。如果小数部分没有的话，那么小数点也必须不能出现。没有指数部分。

`^[-+]?(([0-9]+(\.\.[0-9]+)?|\.\.[0-9]+)$`  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

符号、整数部分和小数部分是可选的。如果没有整数部分，那么小数部分是必需的。如果没有小数部分，那么小数点是可选的。没有指数部分。

`^[-+]?(([0-9]+(\.\.[0-9]*)?|\.\.[0-9]+)$`  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

符号、整数部分和小数部分是可选的。如果没有整数部分，那么小数部分是必需的。如果没有小数部分，那么小数点也必须不能出现。指数部分可选：

`^[-+]?(([0-9]+(\.\.[0-9]+)?|\.\.[0-9]+)([eE][-+]?[0-9]+)?$`  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

符号、整数部分和小数部分是可选的。如果没有整数部分，那么小数部分是必需的。如果没有小数部分，那么小数点是可选的。指数部分可选：

`^[-+]?(([0-9]+(\.\.[0-9]*)?|\.\.[0-9]+)([eE][-+]?[0-9]+)?$`  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

对上一个正则表达式进行修改，使之可以匹配在一大段文本中的数字：

```
[+-]?(\\b[0-9]+(\\.?[0-9]*?)?|\\.?[0-9]+)([eE][+-]?[0-9]+\\b)?  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby\
```

## 讨论

这里的所有正则表达式两边都添加了定位符（实例 2.5），以确保我们会检查整个输入是一个浮点数，而不是在更大的字符串中的一个浮点数。如果你想要在一大段文本中查找浮点数，那么可以按照在实例 6.1 中的讲解使用单词边界或者环视。

不包含任何可选组成部分的解答是很直接的：它们只是简单地从左向右说明了要匹配的内容。我们使用字符类（实例 2.3）来匹配符号、数字和 e。其中的加号和问号量词（实例 2.12）允许出现任意多个数字和一个可选的指数符号。

如果只是把符号和整数部分变为可选的也比较容易。在表示符号的字符类之后添加一个问号就可以把它变成可选的。而把加号量词替换为星号就可以允许对整数数字重复 0 次或多次，而不是 1 次或多次。

当我们需要把符号、整数和小数部分都变成可选的时候，就会遇到比较复杂的情况。虽然单个来讲，它们都是可选的，但是它们不能同时都不出现，而空字符串显然不是一个合法的浮点数。如果采用最简单的解答形式，<[+-]?[0-9]\*\\.?[0-9]\*>，它也可以匹配所有合法的浮点数，但是也能匹配到空字符串。而且由于略掉了定位符，所以正则式会匹配到目标文本中在任意两个字符之间的长度为 0 的字符串。如果用这个正则式和替代文本«{\$&}»对字符串 123abc456 执行查找和替换，你会得到{123}{}a{}b{}c{456}{}。正则式可以正确匹配 123 和 456，但是它还会在每次其他匹配尝试中都找到一个长度为 0 的匹配。

如果你要创建一个有可能所有内容都可选的正则表达式，那么就需要考虑一下如果其中一个部分实际上被略去之后，其他所有内容是否还真的是可选的。例如，浮点数就必须拥有至少一个数字。

这个实例的解决方案中清楚地说明了当整数和小数部分都是可选的时候，二者之一依然是必需的。它们同样还明确说明了 123.到底是一个带小数点的浮点数，还是一个整数之后跟着一个不属于数字的点号。例如，在有些编程语言中，整数之后的点号可以是一个连接操作符，或者是由两个点号表示的区间操作符中的第一个点号。

要实现整数和小数部分不能同时被省略的要求，我们在一个分组（实例 2.9）中使用了多选结构（实例 2.8）来把两种情形都列出来。`<[0-9]+(\\.?[0-9]+)?>` 会匹配一个必需的整数部分和可选的小数部分。`<\\.?[0-9]+>` 会只匹配一个小数部分。

把二者组合起来，`<[0-9]+(\\.?[0-9]+)?|\\.?[0-9]+>` 就可以覆盖所有 3 种情形。第一个选择分支覆盖了包含整数和小数部分的浮点数，以及不包含小数部分的数字。第二个选择分

支则只匹配了小数部分。因为多选操作符的优先级是最低的，所以在我们把它们添加到一个更长的正则表达式之前，必须把这两个选择分支放到一个分组中。

`<[0-9]+(\.[0-9]+)?\.[0-9]+>` 要求在略去小数部分的时候，也同样要略去小数点。如果即使没有小数数字，也可以出现小数点，那么我们可以使用 `<[0-9]+(\.[0-9]*?)?\.[0-9]+>`。在这个正则式的第一选择分支中，小数部分还是被放到了一个问号量词的分组中，这样它就成为可选的。区别是现在小数部分数字本身也是可选的。我们把加号（1次或多次）替换为了星号（0次或多次）。造成的结果是这个正则式中的第一个选择分支会匹配包含可选小数部分的整数，而小数部分则可以是带数字的小数部分，或者只是一个小小数点。第二个选择分支没有改变。

前面所给的最后一个示例的意义在于我们要求修改的是一个东西，而在正则式中改动的却是另外一个地方。要求修改的是可以让小数点自己出现，而不用一定要和小数部分的数字一起出现。为了达到这个目的，我们修改的是用于小数部分数字的字符类之上的量词。之所以这样做，是因为小数点和字符类都已经被放到了一个分组中，可以让二者同时都成为可选的。

## 参见

实例 2.3、实例 2.8、实例 2.9 和实例 2.12。

## 6.8 含有千位分隔符的数

### 问题描述

你想要匹配使用逗号来作为千位分隔符，点号作为小数分隔符的数字。

### 解决方案

整数和小数部分都是必需的：

```
^[0-9]{1,3}(,[0-9]{3})*\.[0-9]+$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

整数和小数部分是必需的。如果略掉小数部分就必须略去小数点：

```
^[0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

整数和小数部分是可选的。如果略掉小数部分就必须略去小数点。

```
^(([0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?)|\.[0-9]+)$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

对上一个正则式进行修改，从而可以用于在一大段文本中查找数字：

```
\b[0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?\b|\.\.[0-9]+\b  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 讨论

因为所有这些正则表达式都会匹配浮点数，所以它们使用的技巧与上一个实例是相同的。唯一的区别是，在匹配整数部分的时候，我们用的不再是简单的`<[0-9]+>`，而是`<[0-9]{1,3}(,[0-9]{3})*>`。这个正则表达式可以匹配1~3个数字，然后是0个或多个包含逗号和3个数字的分组。

要把整数部分变成可选的，我们并不能用`<[0-9]{0,3}(,[0-9]{3})*>`，这是因为它会匹配到包含一个前导逗号的数字，例如，123。这个陷阱同我们在上一个实例中讲解的把所有内容都变成可选的情形是相同的。要把整数部分变成可选的，我们并不去修改正则式中用于整数的组成部分，而是把它整个都改成可选的。这个解答中的最后两个正则式使用了多选结构来完成这个任务。在多选结构中包含了整数部分必需和小数部分可选的正则式，与匹配不带整数部分的小数部分的正则式。这样就可以产生一个允许整数和小数部分可选，但是又不能同时被省略的正则表达式。

## 参见

实例 2.3、实例 2.9 和实例 2.12。

## 6.9 罗马数字

### 问题描述

你想要匹配罗马数字，例如 IV、XIII 和 MVIII。

### 解决方案

不进行合法性验证的罗马数字：

```
^ [MDCLXVI] +$  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

严格的现代罗马数字：

```
^ (=?[MDCLXVI]) M* (C[MD] | D?C{0,3}) (X[CL] | L?X{0,3}) (I[XV] | V?I{0,3}) $  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

灵活的现代罗马数字：

`^(?=[MDCLXVI])M*(C[MD]|D?C*)(X[CL]|L?X*)(I[XV]|V?I*)$`

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

简单的罗马数字：

`^(?=[MDCLXVI])M*D?C{0,4}L?X{0,4}V?I{0,4}$`

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

罗马数字中使用的是字母 M、D、C、L、X、V 和 I，它们代表的数值分别是 1000、500、100、50、10、5 和 1。第一个正则式匹配包含这些字母的任意字符串，而不去检查这些字母的顺序或者数量是否能够构成一个正确的罗马数字。

在现代（也就是说过去的几百年间），罗马数字通常都会遵守一个严格的规则集。这些规则会为每个数字正好对应一个罗马数字。例如，4 总是会被写作是 IV，而不能是 IIII。在解决方案中的第二个正则表达式只能匹配遵守这些现代规则的罗马数字。

十进制数中每个非 0 的数字都会在罗马数字中单独进行表示。例如，1999 会被写作 MCMXCIX，其中，M 代表 1000，CM 代表 900，XC 代表 90，而 IX 则代表 9。我们不能把它表示成 MIM 或者 IMM。

千位比较容易匹配：一个 M 代表一千，因此很容易用<M\*>来匹配。

对于百位来说，有 10 种不同的变体，我们使用了两个选择分支来进行匹配。<C[MD]>可以匹配 CM 和 CD，分别用来表示 900 和 400。<D?C{0,3}>会匹配 DCCC、DCC、DC、D、CCC、CC、C 以及空串，它们分别代表 800、700、600、500、300、200、100 和空。这样我们就可以匹配百位的所有 10 个数字。

我们用 <X[CL]|L?X{0,3}> 来匹配十位，用 <I[XV]|V?I{0,3}> 来匹配个位。它们的语法与百位的表示是相同的，但是使用了不同的数字。

这个正则式中的所有 4 个组成部分允许所有内容都是可选的，因为每个数字都可以为 0。罗马人没有用来表示 0 的符号或者单词。因此，在罗马数字中无法表示 0。虽然正则式中的每个部分实际上都是可选的，但是它们却不允许同时被省略掉。我们必须确保正则式不能匹配长度为 0 的匹配。要想实现这样的功能，我们在正则式之前添加了顺序环视 <(?= [MDCLXVI])>。在实例 2.16 中讲解过，这个顺序环视可以确保在正则匹配中至少包含一个字母。顺序环视并不会消费它匹配到的字母，因此正则式的剩余部分还可以再次匹配该字母。

第 3 个正则式则要稍微灵活一些。它在接受 IV 的同时，还会接受类似 IIII 这样的数字。

第 4 个正则式只允许使用不带减法的数字，因此所有的字母都必须按照降序出现。例如 4 必须被写作是 IIII，而不是 IV。罗马人自己通常是这样来写数字的。



### 提示

所有正则表达式都被包在定位符（实例 2.5）之间，从而可以保证我们检查整个输入是不是一个罗马数字，而不是在一个更大的字符串中的一个浮点数。如果你想要在一大段文本中查找罗马数字的话，那么你可以把<^>和<\$>替换为单词边界<\b>。

## 把罗马数字转换为十进制

下面的 Perl 函数使用了这个实例中的“严格”正则表达式，来检查一个输入是否是一个合法的罗马数字。如果是的话，它会使用正则式<[MDLV]C[MD]?|X[CL]?|I[XV]?>遍历该数字中的所有字母，并把它们的值加起来。

```
sub roman2decimal {
    my $roman = shift;
    if ($roman =~
        m/^([MDCLXVI])+
            (M*)          # 1000
            (C(MD)|D?C{0,3})  # 100
            (X(CL)|L?X{0,3})  # 10
            (I(XV)|V?I{0,3})  # 1
            $/ix)
    {
        # Roman numeral found
        my %r2d = ('I' => 1, 'IV' => 4, 'V' => 5, 'IX' => 9,
                    'X' => 10, 'XL' => 40, 'L' => 50, 'XC' => 90,
                    'C' => 100, 'CD' => 400, 'D' => 500, 'CM' => 900,
                    'M' => 1000);
        my $decimal = 0;
        while ($roman =~ m/[MDLV]C[MD]?|X[CL]?|I[XV]?/ig) {
            $decimal += $r2d{uc($&)};
        }
        return $decimal;
    } else {
        # Not a Roman numeral
        return 0;
    }
}
```

## 参见

实例 2.3、实例 2.8、实例 2.9、实例 2.12、实例 2.16、实例 3.9 和实例 3.11。

## 第7章

# URL、路径和 Internet 地址

前一章我们讨论了如何处理数字，而在程序中经常需要处理的另一重要对象是在各种各样的路径和定位符中寻找数据：

- URL、URN 和相关的字符串
- 域名
- IP 地址
- Microsoft Windows 文件和文件夹名



特别来讲，其中的 URL 格式被证实相当灵活有用，即使和 Internet 无关的资源也广泛地采纳了 URL 格式。本章要讲解的正则表达式分析工具将会在非常多的情况下展现其价值所在。

## 7.1 URL 合法性验证

### 问题描述

你要检查某段给定的文本是否是一个符合需要的合法 URL。

### 解决方案

允许使用几乎任意的 URL：

```
^(https?|ftp|file)://.+$  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python  
  
\A(https?|ftp|file)://.+\\Z  
正则选项：不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

必须包含一个域名，但不允许出现用户名或者口令：

```
\A                                         # 定位符
(https?|ftp)://                         # 通信协议方案 (scheme)
[a-z0-9-]+(\.[a-z0-9-]+)+                 # 域
([/?].*)?                                    # 路径和/或参数
\Z                                         # 定位符

正则选项: 宽松排列、不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^(https?|ftp)://[a-z0-9-]+(\.[a-z0-9-]+)+$ # 定位符
([/?].+)?$                                     # 路径和/或参数
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

必须包含一个域名，不允许使用用户名或口令。如果在子域名（www 或 ftp）中隐含了通信协议方案（也称作 scheme，即 http 或 ftp 等），那么允许省略通信协议方案：

```
\A                                         # 定位符
((https?|ftp)://|(www|ftp)\.)           # 通信协议方案或子域名
[a-z0-9-]+(\.[a-z0-9-]+)+                 # 域名
([/?].*)?                                    # 路径和/或参数
\Z                                         # 定位符

正则选项: 宽松排列、不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^((https?|ftp)://|(www|ftp)\.)[a-z0-9-]+(\.[a-z0-9-]+)+([/?].*)?$ # 定位符
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python
```

必须包含一个域名和指向一个图像文件的路径。不允许使用用户名、口令或参数：

```
\A                                         # 定位符
(https?|ftp)://                         # 通信协议方案
[a-z0-9-]+(\.[a-z0-9-]+)+                 # 域
(/[\w-]+)*                                    # 路径
/[\w-]+\.(gif|png|jpg)                      # 文件
\Z                                         # 定位符

正则选项: 宽松排列、不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^(https?|ftp)://[a-z0-9-]+(\.[a-z0-9-]+)+(/[\w-]+)*[/\w-]+\.(gif|pn
g|jpg)$
```

**正则选项:** 不区分大小写  
**正则流派:** .NET、Java、JavaScript、PCRE、Perl、Python

## 讨论

你无法创建一个能匹配每个合法的 URL，并且不会匹配任何不合法 URL 的正则表达式。原因是在某些尚未问世的的通信协议方案中，很可能几乎任何字符串都是一个合法的 URL。

因此，只有当我们知道要求 URL 合法的上下文环境时，URL 的合法性验证才是有意义的。这时，可以把能够接受的 URL 限制为当前正在使用的软件所支持的通信协议方案。本实例中所有正则表达式都是在 web 浏览器中使用的。这类 URL 使用的是如下格式：

```
scheme://user:password@domain.name:80/path/file.ext?param=value&param2= value2#fragment
```

其中所有的组成部分实际上都是可选的。一个 file: URL 可以只包含有一个路径。而另一个 http: URL 则可以只包含一个域名。

在解决方案中的第一个正则表达式会检查 URL 是否以 web 浏览器普遍采用的通信协议方案开头：http、https、ftp 和 file。正则表达式使用脱字符来定位到字符串的开始（实例 2.5）。多选结构（实例 2.8）被用来罗列出通信协议方案的列表。`<https?>` 是`<http|https>`的一种更为聪明的表示方式。

因为第一个正则表达式允许截然不同的通信协议方案，例如 http 和 file，所以它并不会试图对通信协议方案后的文本进行合法性验证。`<.+>` 会简单地抓取直到字符串结尾之前的一切字符，这个字符串中并不包含任何换行字符。

默认来说，点号（实例 2.4）匹配除了换行字符之外的所有字符，而美元符号（实例 2.5）并不会匹配任何内嵌的换行符。这里 Ruby 会是一个例外。在 Ruby 中，脱字符和美元符号总是会匹配任何内嵌的换行符，所以只能使用`\A` 和`\Z` 来替代它们（实例 2.5）。严格来讲，对于 Ruby 来说，你必须对此实例中所有其他正则表达式进行同样的修改。如果输入可以由多行组成，而且你想要避免匹配一个在几行文本中占用一行的 URL，那么你就应该这样做。

接下来的两个正则表达式分别是同一正则表达式的宽松排列版本（实例 2.18）和常规版本。宽松排列的正则表达式更容易阅读，然而常规版本的输入则会更快。JavaScript 并不支持宽松排列的正则表达式。

这两种正则表达式只接受 web 和 FTP URL，且要求 HTTP 或者 FTP 通信协议方案后面必须跟随一个看起来像是合法域名的字符串。域名必须以 ASCII 字符组成。国际化域名（IDN）是不被接受的。域名后面必须可以跟随一个路径或者用斜杠或问号分隔的参数列表。因为问号位于一个字符类中（实例 2.3），所以我们不需要对它进行转义。问号是字符类中的一个普通字符，而斜杠在正则表达式中任何地方都是一个普通字符。（如果你在源代码中看到它被转义了，那是因为在 Perl 和其他一些编程语言中使用斜杠来界定字面的正则表达式。）

在该正则式中，并没有企图对路径或者参数进行合法性验证。`<.*>` 只是简单地匹配了不包含换行符的任何东西。由于路径和参数都是可选的，`<[/?].*>` 被放在一个用问号标示为可选的分组中（实例 2.12）。

这些正则表达式，以及接下来的一些正则表达式，都不允许用户名和口令被定义为 URL 的一部分。出于安全考虑，把用户信息放在 URL 中被认为是一种坏习惯。

大多数 web 浏览器都会接受不明确指明通信协议方案的 URL，并且能够从域名中正确地推断出对应的通信协议方案。例如，[www.regexbuddy.com](http://www.regexbuddy.com) 是 <http://www.regexbuddy.com> 的简写。为了支持这类 URL，我们简单地扩充了正则表达式所允许的通信协议方案列表，使之包含子域名 www. 和 ftp.。

`<(https?|ftp)://(www|ftp)\.›` 很好地做到了这一点。这个列表中包含 2 个选择分支，每个都以 2 个选择分支作为开始。第一个选择分支允许 `<https?›` 和 `<ftp›`，后面必须紧随 `<://›`。第二个选择分支允许 `<www›` 和 `<ftp›`，后面必须跟随着一个点号。你可以很容易地编辑每个列表，改变正则表达式应该接受的通信协议方案和子域名。

最后 2 个正则表达式要求包含一个通信协议方案、一个 ASCII 的域名、路径和代表一个 GIF、PNG 或者 JPEG 图像文件的文件名。在任何脚本中，路径和文件名中都允许出现所有的字母、下划线和连字符。简写字符类 `\w` 包括了所有这些字符，除了连字符之外（实例 2.3）。

那么到底应该使用哪个正则表达式呢？这实际上要取决于你到底想要做什么。在许多情况下，答案可能是根本不要使用任何正则表达式。试着简单地去访问 URL。如果它返回了有效的内容，就接受它。如果获得了 404 或者其他错误，则拒绝它。最终来讲，这是检查一个 URL 是否有效的唯一有实际意义的测试。

## 参见

实例 2.3、实例 2.8、实例 2.9 和实例 2.12。

## 7.2 在全文中查找 URL

### 问题描述

你要在一大段文本中查找 URL。URL 可能被诸如圆括号这样的标点符号包围起来，也可能没有，用来包住它的标点符号并不属于 URL 的一部分。

### 解决方案

不包含空格的 URL：

```
\b(https?|ftp|file)://\S+
正则选项：不区分大小写
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

不包含空格和结尾标点符号的 URL：

```
\b(https?|ftp|file)://[-A-Z0-9+\&%#=~_|$!:,.;]*\d+
[A-Z0-9+\&%#=~_|$]
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

不包含空格或者结尾标点符号的 URL，以 www 或 ftp 子域名开始的 URL 可以省略掉通信协议方案：

```
\b((https?|ftp|file)://|(www|ftp)\.)[-A-Z0-9+&@#/%?=~_|\$!:,.;]*\b
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

假设有这样一段文字：

```
Visit http://www.somesite.com/page, where you will find more information.
```

你能找到其中的 URL 是什么吗？

在你说出 `http://www.somesite.com/page` 这样的答案之前，请认真想一想：标点符号和空格都是 URL 中的合法字符。逗号、点号甚至空格都不一定要被转义为`%20`。字面的空格是完全合法的。有些所见即所得（WYSIWYG）的网页写作工具甚至让用户能够方便地在文件和文件夹名中放入空格，并且把这些空格包含在指向这些文件的链接中。

这意味着如果使用一个能够允许所有合法 URL 的正则表达式，它将在上述文本中找到如下 URL：

```
http://www.somesite.com/page, where you will find more information.
```

很少有人在输入这句话时有意把空格作为 URL 的一部分，这是因为 URL 中含有未转义的空格毕竟是很罕见的。在解决方案中的第一个正则表达式使用简写字符类`\S`排除了这种情况，简写字符类`\S`会包含所有非空格的字符。尽管这个正则表达式采用了“不区分大小写”的选项，但是 S 必须是大写的，因为`\S`不同于`\s`。实际上，它们恰好完全相反。具体讲解可参见实例 2.3。

第一个正则表达式还是非常粗糙。它会把示例文本中的逗号也包括到 URL 中。尽管在 URL 中包含逗号和其他标点符号并不很罕见，但标点符号还是很少会在 URL 结尾处出现的。

下一个正则表达式使用了两个字符类来替代单个的简写`\S`。第一个字符类比第二个要包括更多的标点符号。第二个字符类排除了当 URL 放在英文句子中时，跟在 URL 后在英文中可能作为标点符号出现的字符。第一个字符类中有一个星号量词（实例 2.12），允许 URL 为任意长度。第二个字符类则没有使用量词，要求 URL 以此字符类中的一个字符结尾。字符类不包括小写字母，“不区分大小写”选项会被用来保证这一点。关于如何在你的编程语言中使用这类选项，请参考实例 3.4 中的讲解。

第二个正则表达式在处理使用了一些特殊标点符号的 URL 时会出现错误，它将会只能

匹配 URL 的一部分。但是这个正则表达式确实解决了非常普遍的 URL 后面跟随一个逗号或者句号的情况，同时依然允许 URL 中包含有逗号或者点号。

许多 Web 浏览器可以接受不指明通信协议方案的 URL，然后从域名中正确地推断出通信协议方案。例如，[www.regexbuddy.com](http://www.regexbuddy.com) 是 <http://www.regexbuddy.com> 的简写。为了允许使用这些 URL，最后一个正则表达式扩充了所支持的通信协议方案的列表，把子域名 `www.` 和 `ftp.` 包括在所允许的通信协议方案列表中。

`<(https?|ftp)://|(www|ftp)\.›` 很好地做到了这一点。这个列表中包含 2 个选择分支，每个又会以 2 个选择分支开始。第一个选择分支允许 `<https?›` 和 `<ftp›`，后面必须紧随 `<://›`。第二个选择分支则允许 `<www›` 和 `<ftp›`，后面必须跟随着一个点号。你可以很容易地编辑每个列表，从而改变正则表达式应该接受的通信协议方案和子域名。

## 参见

实例 2.3 和实例 2.6。

## 7.3 在全文中查找加引号的 URL

### 问题描述

你要在一大段文本中查找 URL。URL 可能被标点符号包括起来，也可能没有，外包的标点符号是属于外围文本的一部分，而不是 URL 的一部分。你要使用户可以选择是否把 URL 放到引号中，这样就能显式地指出标点符号或者甚至是空格，是否应当是 URL 的一部分。

### 解决方案

```
\b(?:(:https?|ftp|file)://| (www|ftp)\.) [-A-Z0-9+&@#/%?=~_|$!:,.;]* [-A-Z0-9+&@#/%?=~_|$]  
| "(?:(:https?|ftp|file)://| (www|ftp)\.)[^"\r\n]+"  
| '(:https?|ftp|file)://| (www|ftp)\.)[^'\r\n]+'  
正则选项：宽松排列、不区分大小写、点号匹配换行符、定位符匹配换行处  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### 讨论

前一实例解释了 URL 和英语文本混合在一起的问题，以及如何将英语标点符号和 URL 字符区分开来。尽管前一实例的解决方案非常实用，并且它在大多数时候都能正确工作，但是没有一个正则表达式能够在所有时候都是对的。

如果把你的正则表达式用在尚未写出的文本中，那么可以为用户提供一个为 URL 周围加引号的方法。本节我们所给的解决方案都允许把一对单引号或者双引号加在 URL 周

围。当一个 URL 被加引号时，它必须以其中的一个通信协议方案 `<https?|ftp|file>` 或者是两个子域名 `<www|ftp>` 之一作为开头。在一个通信协议方案或者子域名之后，正则表达式允许 URL 包括除了换行符和作为分界符的引号以外的任意字符。

整个正则表达式可以被拆分为 3 个选择分支。第一个选择分支是来自前一实例的正则表达式，它匹配未加引号的 URL，试图区分英文中的标点符号和 URL 字符。第二个选择分支会匹配一个加了双引号的 URL。第 3 个选择分支匹配一个加了单引号的 URL。这里我们用了 2 个选择分支，而不是用一个带捕获分组的选择分支来获取起始引号，然后使用向后引用来自匹配结束引号，因为我们不能在一个否定字符类中使用向后引用排除 URL 中的引号符。

之所以选择单引号和双引号，是因为这是在 HTML 和 XHTML 文件中 URL 通常会出现的方式。这种为 URL 加引号的方法对于在 Web 上工作的人们来说很自然，当然你也可以很容易地编辑这个正则表达式，使之支持使用不同的字符对来分隔 URL。

## 参见

实例 2.8 和实例 2.9。

## 7.4 在全文中寻找加括号的 URL

### 问题描述

你要在一大段文本中找到 URL。URL 可能被标点符号包括起来，也可能没有，外包的标点符号是属于外围文本的一部分，而不是 URL 的一部分。你要正确地匹配在内部包含了括号对的 URL，而不能匹配位于整个 URL 两边的一对括号。

### 解决方案

```
\b(?:(:https?|ftp|file)://|www\.|ftp\.)  
(?:\([-A-Z0-9+&@#/%=~_|$?!:,.\]*\)|[-A-Z0-9+&@#/%=~_|$?!:,.\]*)  
(?:\([-A-Z0-9+&@#/%=~_|$?!:,.\]*\)|[A-Z0-9+&@#/%=~_|$])
```

正则选项：宽松排列、不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

```
\b(?:(:https?|ftp|file)://|www\.|ftp\.) (?:\([-A-Z0-9+&@#/%=~_|$?!:,.\]*\)|  
[-A-Z0-9+&@#/%=~_|$?!:,.\]*) (?:\([-A-Z0-9+&@#/%=~_|$?!:,.\]*\)|  
[A-Z0-9+&@#/%=~_|$])
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

几乎任意字符都是 URL 中的合法字符，这其中也包含了括号。然而，括号在 URL 中极为罕见，这就是为什么我们不在上一个实例的任何一个正则表达式中包含括号的原

因。但是一些重要的网站已经开始使用括号了：

```
http://en.wikipedia.org/wiki/PC_Tools_(Central_Point_Software)  
http://msdn.microsoft.com/en-us/library/aa752574(VS.85).aspx
```

一种解决方案是要求你的用户为这样的 URL 加引号。另外一种解决方案是强化你自己的正则式来接受这类 URL。这里的困难之处在于确定一个结束括号是否是 URL 的一部分，还是它被用作包围 URL 的标点符号，如下例所示：

```
RegexBuddy's web site (at http://www.regexbuddy.com) is really cool.
```

因为有可能两个括号中的一个与 URL 较为紧邻，而另一个括号则距离 URL 较远，所以我们不能使用前一实例中加引号的正则式技巧。最直观的解决方案是只允许 URL 中的括号以不嵌套的左括号和右括号对出现。前面所给的例子中，维基百科和微软的 URL 符合这一需求。

前面解决方案中所给的两个正则表达式是一样的。第一个使用了宽松排列模式，使得它更容易阅读。

这些正则表达式本质上和实例 7.2 的解决方案是一样的。所有正则表达式都由 3 个部分组成：通信协议方案的列表，紧接着是一个 URL 主体，其中使用了星号量词允许 URL 为任意长度，以及 URL 的结尾，它没有使用量词（也就是说，它只能出现一次）。在实例 7.2 最初的正则表达式中，URL 主体和 URL 结尾都只包含了一个字符类。

本实例的解决方案用更精细的内容替换了 2 个字符类。

中间的字符类：

```
[-A-Z0-9+&@#/=%=~_|$?!:,.]
```

被替换为：

```
\([-A-Z0-9+&@#/=%=~_|$?!:,.]*)|[-A-Z0-9+&@#/=%=~_|$?!:,.]
```

最后一个字符类：

```
[A-Z0-9+&@#/=%=~_|$]
```

被替换为：

```
\([-A-Z0-9+&@#/=%=~_|$?!:,.]*)|[A-Z0-9+&@#/=%=~_|$]
```

这两个字符类都被替换成和多选结构有关的内容（实例 2.8）。因为多选结构在所有的正则式操作符中的优先级最低，因此我们使用了非捕获分组（实例 2.9）来保证这两个多选结构是并列的。

对于两个字符类来说，我们添加了选择分支`\([-A-Z0-9+&@#/=%=~_|$?!:,.]*)``，并将原来的字符类作为另外一个选择分支。新添加的选择分支会匹配一对括号，并允许任意长度的任意字符出现在其间的 URL 中。

最后一个字符类会使用同一个选择分支，允许 URL 以两个括号之间的文本结束，或者以某个在英文中不太可能成为标点符号的单个字符结束。

把这些组合起来，就可以得到一个匹配包含任意数量括号的 URL 的正则式，其中包括没有括号的 URL，以及甚至是只由括号组成的 URL，只要这些括号成对存在即可。

对于 URL 主体，我们把星号量词放在整个非捕获分组的两边。这样就允许任意数量的括号对出现在 URL 中。因为在非捕获分组的外面有星号，所以就不再需要对原来的字符类直接加星号。实际上，我们必须确保不把星号包括进去。

在本解决方案中的正则式的中间存在 `<(ab*c|d)*>` 的格式，这里 `<a>` 和 `<c>` 是字面的括号，而 `<b>` 和 `<d>` 是字符类。如果把它写成 `<(ab*c|d*)*>` 则会是一个错误。初看起来这似乎符合逻辑，因为我们允许 `<d>` 中的字符出现任意多次，然而外围的 `<*>` 已经能够很好地重复 `<d>`。如果在 `<d>` 上直接加上一个内部的星号，该正则表达式的复杂度就会出现指数级的增长。`<(d*)*>` 可以按照许多不同方式来匹配 `dddd`。例如，外围的星号可以重复 4 次，而内部的星号每次重复 1 次。外围星号也可以重复 3 次，而内部星号则可以采取重复 2-1-1、1-2-1 或者 1-1-2。如果外围星号重复 2 次，那么内部星号可以是 2-2、1-3 或者 3-1。你可以想像，当字符串长度增加时，可能的组合数量是快速爆炸式增长的。我们把这叫作灾难性的回溯，它是在实例 2.15 中引人的概念。当正则表达式不能找到一个合法的匹配时就会出现这个问题，例如，这可能是因为你已经在正则表达式尾部附加了一些内容，来查找包含特定要求字符或者以特定要求的字符作为结尾的 URL。

## 参见

实例 2.8 和实例 2.9。

## 7.5 把 URL 转变为链接

### 问题描述

你有一段可能包含一个或多个 URL 的文本。你需要把其中所包含的 URL 转变为链接，把 HTML 定位符标签添加到 URL 之上。URL 字符串本身则可以作为链接的目标和被加链接的文本。

### 解决方案

要找到文本中的 URL，可以使用实例 7.2 或者实例 7.4 中的正则表达式。至于替代文本，习使用：

```
<a●href="$&">$&</a>
    替代文本流派: .NET、JavaScript、Perl
<a●href="$0">$0</a>
    替代文本流派: .NET、Java、PHP
```

```
<a•href="\0">\0</a>
    替代文本流派: PHP、Ruby
<a•href="\&">\&</a>
    替代文本流派: Ruby
<a•href="\g<0>">\g<0></a>
    替代文本流派: Python
```

在编程时，你可以参考在实例 3.15 中讲解过的查找和替换操作来实现。

## 讨论

本问题的解决方案非常直观。我们使用了一个正则表达式来匹配 URL，然后用 «<URL>» 来替代它，这里的 URL 代表我们所匹配到的 URL。不同的编程语言会使用不同的语法来作为替代文本，因而这个问题的解决方案中包含了一长串的列表。但是它们所做的的确都是一样的事情。实例 2.20 中讲解了替代文本语法。

## 参见

实例 2.21、实例 3.15、实例 7.2 和实例 7.4。

## 7.6 URN 合法性验证

### 问题描述

你要检查是否一个字符串代表了一个合法的统一资源名称 (URN)，并且符合 RFC 2141 中的规定，或者是要在一大段文本中查找 URN。

### 解决方案

检查一个字符串是否整体上由一个合法 URN 组成：

```
\Aurn:
# 命名空间标识
[a-z0-9] [a-z0-9-]{0,31}:
# 特定于命名空间的字符串
[a-z0-9()+, \-.:=@;$_!*'%/?#]+
\Z
    正则选项: 宽松排列、不区分大小写
    正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

```
^urn: [a-z0-9] [a-z0-9-]{0,31}: [a-z0-9()+, \-.:=@;$_!*'%/?#]+$
```

正则选项: 不区分大小写

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python

在一大段文本中查找 URN：

```
\burn:  
# 命名空间标识  
[a-zA-Z0-9][a-zA-Z0-9-]{0,31}:  
# 特定于命名空间的字符串  
[a-zA-Z0-9()]+, \-.:=@;$_!*'%/?#+  
    正则选项: 宽松排列、不区分大小写  
    正则流派: .NET、Java、PCRE、Perl、Python、Ruby  
  
\burn:[a-zA-Z0-9][a-zA-Z0-9-]{0,31}:[a-zA-Z0-9()]+, \-.:=@;$_!*'%/?#+  
    正则选项: 不区分大小写  
    正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

在一大段文本中找到一个 URN，并假设 URN 结尾处的标点符号是引用该 URN 的（英文）文本的一部分，而不属于 URN 本身的一部分：

```
\burn:  
# 命名空间标识  
[a-zA-Z0-9][a-zA-Z0-9-]{0,31}:  
# 特定于命名空间的字符串  
[a-zA-Z0-9()]+, \-.:=@;$_!*'%/?#+*[a-zA-Z0-9+=@$/]  
    正则选项: 宽松排列、不区分大小写  
    正则流派: .NET、Java、PCRE、Perl、Python、Ruby  
  
\burn:[a-zA-Z0-9][a-zA-Z0-9-]{0,31}:[a-zA-Z0-9()]+, \-.:=@;$_!*'%/?#+*[a-zA-Z0-9+=@$/]  
    正则选项: 不区分大小写  
    正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 讨论

一个 URN 由 3 个部分组成。第一部分是 4 个字符 urn:，我们可以在正则表达式中以字面方式添加这 4 个字符。

第二部分是命名空间标识 (NID)。它的长度介于 1~32 个字符之间。第一个字符必须是字母或者数字。余下部分的字符可以是字母、数字或者连字符。我们用了 2 个字符类（参见实例 2.3）来进行匹配：第一个字符类匹配一个字母或数字，第二个字符类匹配 0~31 个字母、数字和连字符。NID 必须以冒号分隔，正则式中可以在字面上添加冒号。

URN 的第三个部分是特定于命名空间的字符串 (NSS)。它可以是任意长度的，可以包括字母、数字以及各种各样的标点符号字符。我们用另一字符类可以很容易地匹配到它。在字符类后的加号使之重复一次或者多次（实例 2.12）。

假如你要检查一个字符串是否代表了一个合法的 URN，剩下还需要做的事情就是在正则式的开头和结尾加上定位符以匹配字符串的开头和结尾。除了 Ruby 之外的所有正则流派都可以用 <^> 和 <\$> 匹配字符串开头和结尾，而除了 JavaScript 之外的所有正则流派都可以用 <\A> 和 <\Z>。关于这些定位符的详细信息，可以参考实例 2.5。

如果你要在一大段文本中来查找 URN，这就有一点麻烦了。实例 7.2 中讨论过的 URL

标点符号的问题对 URN 也存在。假设你有如下的文本：

```
The URN is urn:nid:nss, isn't it?
```

这里存在的问题在于逗号是否是 URN 的一部分。用逗号结尾的 URN 从语法上来说是合法的，但是任何人读这个英文句子时都会将这个逗号看成英文的标点，而不是 URN 的一部分。解决方案中的最后一个正则表达式采用了比 RFC 2141 更为严格的限制解决了这个问题。它把 URN 的最后一个字符限制为在 NSS 部分中合法的字符，这样它就不可能在用到 URN 的句子中以英文标点符号的身份出现。

要完成这个任务，还需要把加号量词（1 个或多个）替换为星号量词（0 个或多个），并且为结尾字符添加第二个字符类。如果我们不更改量词而添加了这个字符类，那么就会要求 NSS 至少有 2 个字符长，而这不是我们所想要的结果。

## 参见

实例 2.3 和实例 2.12。

## 7.7 通用 URL 的合法性验证

### 问题描述

你要检查一段给定的文本是否是一个符合 RFC 3986 标准的 URL。

### 解决方案

```
\A
  (# 通信协议方案
  [a-z] [a-z0-9+\-.]*:
  (# 授权和路径
  //
  ([a-z0-9\-\_\~%\!$&'\()^+,;=:]+@) ?          # 用户名
  ([a-z0-9\-\_\~%]+                                # 命名主机
  | \[[a-f0-9::]+\]                                # IPv6 主机
  | \[v[a-f0-9] [a-z0-9\-\_\~%\!$&'\()^+,;=:]+\]) # I PvFuture 主机
  (:[0-9]+)?                                         # 端口
  (/ [a-z0-9\-\_\~%\!$&'\()^+,;=:@]+) */?        # 路径
  |# 不包含授权的路径
  (/? [a-z0-9\-\_\~%\!$&'\()^+,;=:@]+(/ [a-z0-9\-\_\~%\!$&'\()^+,;=:@]+) */?) ?
  )
  |# 相对 URL (不包含通信协议方案或授权)
  (# 相对路径
  [a-z0-9\-\_\~%\!$&'\()^+,;=@]+(/ [a-z0-9\-\_\~%\!$&'\()^+,;=:@]+) */?
  |# 绝对路径
  (/ [a-z0-9\-\_\~%\!$&'\()^+,;=:@]+) */?
  )
```

```

)
# 查询
(\? [a-z0-9\-.~%!$&' ()*+,;=:@/?]* )?
# 片段
(\#[a-z0-9\-.~%!$&' ()*+,;=:@/?]* )?
\Z
    正则选项: 宽松排列、不区分大小写
    正则流派: .NET、Java、PCRE、Perl、Python、Ruby

\A
    (# 通信协议方案
        (?<scheme>[a-z] [a-z0-9+\-.]*):
        (# 授权和路径
            //
            (?<user>[a-z0-9\-.~%!$&' ()*+,;=:@]+) # 用户名
            (?<host>[a-z0-9\-.~%]+ # 命名主机
            | \[[a-f0-9:.\]+] # IPv6 主机
            | \[v[a-f0-9][a-z0-9\-.~%!$&' ()*+,;=:]+\]) # IPvFuture 主机
            (?<port>:[0-9]+)? # 端口
            (?<path>(/ [a-z0-9\-.~%!$&' ()*+,;=:@]+)*/?) # 路径
        | # 不包含授权的路径
            (?<path>/? [a-z0-9\-.~%!$&' ()*+,;=:@]+
                (/ [a-z0-9\-.~%!$&' ()*+,;=:@]+)*/?)?
        )
    | # 相对 URL (不包含通信协议方案或授权)
        (?<path>
            # 相对路径
            [a-z0-9\-.~%!$&' ()*+,;=:@]+(/ [a-z0-9\-.~%!$&' ()*+,;=:@]+)*/?
        | # 绝对路径
            (/ [a-z0-9\-.~%!$&' ()*+,;=:@]+)*/?
        )
    |
    # 查询
    (?<query>\? [a-z0-9\-.~%!$&' ()*+,;=:@/?]* )?
    # 片段
    (?<fragment>\#[a-z0-9\-.~%!$&' ()*+,;=:@/?]* )?
\Z
    正则选项: 宽松排列、不区分大小写
    正则流派: .NET

\A
    (# 通信协议方案
        (?<scheme>[a-z] [a-z0-9+\-.]*):
        (# 授权和路径
            //
            (?<user>[a-z0-9\-.~%!$&' ()*+,;=:@]+) # 用户名
            (?<host>[a-z0-9\-.~%]+ # 命名主机
            | \[[a-f0-9:.\]+] # IPv6 主机
            | \[v[a-f0-9][a-z0-9\-.~%!$&' ()*+,;=:]+\]) # IPvFuture 主机
            (?<port>:[0-9]+)? # 端口

```

```

(?<hostpath>(/ [a-z0-9\-.~%!$&' () *+, ;=:@]+) */?) # 路径
| # 不包含授权的路径
  (?<schemepath>/? [a-z0-9\-.~%!$&' () *+, ;=:@] +
    (/ [a-z0-9\-.~%!$&' () *+, ;=:@]+) */?) ?
)
| # 相对 URL (不包含通信协议方案或授权)
  (?<relpath>
    # 相对 URL
    [a-z0-9\-.~%!$&' () *+, ;=:@]+ (/ [a-z0-9\-.~%!$&' () *+, ;=:@]+) */?
  | # 绝对 URL
    (/ [a-z0-9\-.~%!$&' () *+, ;=:@]+) */?
  )
|
# 查询
(?<query>\? [a-z0-9\-.~%!$&' () *+, ;=:@/?]*)?
# 片段
(?<fragment>\# [a-z0-9\-.~%!$&' () *+, ;=:@/?]*)?
\Z

```

正则选项：宽松排列、不区分大小写  
 正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

```

\A
(# 通信协议方案
(?P<scheme>[a-z][a-z0-9+\-.]*):
(# 授权和路径
//
(?P<user>[a-z0-9\-.~%!$&' () *+, ;=]+@)? # 用户名
(?P<host>[a-z0-9\-.~%]+ # 命名主机
| \[[a-f0-9:]+]\] # IPv6 主机
| \[v[a-f0-9][a-z0-9\-.~%!$&' () *+, ;=]+\]) # IPvFuture 主机
(?P<port>:[0-9]+)? # 端口
(?P<hostpath>(/ [a-z0-9\-.~%!$&' () *+, ;=:@]+) */?) # 路径
| # 不包含授权的路径
  (?P<schemepath>/? [a-z0-9\-.~%!$&' () *+, ;=:@] +
    (/ [a-z0-9\-.~%!$&' () *+, ;=:@]+) */?) ?
)
| # 相对 URL (不包含通信协议方案或授权)
  (?P<relpath>
    # 相对 URL
    [a-z0-9\-.~%!$&' () *+, ;=:@]+ (/ [a-z0-9\-.~%!$&' () *+, ;=:@]+) */?
  | # 绝对 URL
    (/ [a-z0-9\-.~%!$&' () *+, ;=:@]+) */?
  )
|
# 查询
(?P<query>\? [a-z0-9\-.~%!$&' () *+, ;=:@/?]*)?
# 片段
(?P<fragment>\# [a-z0-9\-.~%!$&' () *+, ;=:@/?]*)?
\Z

```

正则选项：宽松排列、不区分大小写

正则流派：PCRE 4 及更高版本、Perl 5.10、Python

```
^([a-z][a-z0-9+\-.]*:(\/\/|([a-z0-9\-\_\~%\$\&'()*\+,;=:]+@)?([a-z0-9\-\_\~%]+|\+|\-|\|[a-f0-9:\.]+\]|\\[v[a-f0-9][a-z0-9\-\_\~%\$\&'()*\+,;=:]+\])(:[0-9]+)?|\(\/[a-z0-9\-\_\~%\$\&'()*\+,;=:@]+\)*\/?|\(\/\?[a-z0-9\-\_\~%\$\&'()*\+,;=:@]+\)+|\(\/[a-z0-9\-\_\~%\$\&'()*\+,;=:@]+\)*\/?|\(\/[a-z0-9\-\_\~%\$\&'()*\+,;=:@]+\)+|\(\/[a-z0-9\-\_\~%\$\&'()*\+,;=:@]+\)*\/?|\(\/[a-z0-9\-\_\~%\$\&'()*\+,;=:@]+\)+|\(\/?[a-z0-9\-\_\~%\$\&'()*\+,;=:@]\/?)\*)?(\#[a-z0-9\-\_\~%\$\&'()*\+,;=:@]\/?)\*$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

## 讨论

本章中前面的许多实例处理的都是 URL，并且这些实例中的正则表达式所处理的是特定类型的 URL。有时候出于一些特定的目的，我们会选择采用某些特定的正则表达式，例如要决定标点符号是 URL 的一部分还是引用了该 URL 的文本的一部分。

本实例中的正则表达式处理的则是通用的 URL。这些正则式的目的不是在一大段文本中寻找 URL，而是为了验证其中所包含的 URL 字符串的合法性，并且将 URL 分割成不同的组成部分。前文的实例为各种类型的 URL 完成了这个任务，但是在实践中，你经常会需要让正则式更有针对性。这之后的实例会展示更有针对性的正则式。

RFC 3986 标准描述了一个合法 URL 该是什么样的。它覆盖了每个可能的 URL，包括了相对 URL 和支持尚未问世的通信协议方案的 URL。这样做的结果是，RFC 3986 是非常宽泛的，实现它的正则表达式会相当长。本实例中的正则表达式只实现了其中最基本的部分。它能够可靠地把 URL 分成不同的组成部分，但还没有对各个部分分别执行合法性验证。

验证各部分的合法性需要对每个 URL 通信协议方案的特定知识。RFC 3986 并没有覆盖你在现实生活中可能遇到的所有 URL。例如，许多浏览器和 Web 服务器接受含有字面空格的 URL，但是 RFC 3986 中则要求空格都必须被转义为%20。

一个绝对的 URL 必须以通信协议方案作为开始，例如 http: 或者 ftp:。通信协议方案的第一个字符必须是字母。余下的字符可以是字母、数字或者一些特定的标点符号字符。我们能很容易地用 2 个字符类来匹配它：`<[a-z][a-z0-9+\-.]*>`。

许多 URL 要求包含 RFC 3986 中所谓的“授权”(authority)。授权是一个域名或者服务器的 IP 地址，可以有一个用户名作为前缀，还可以有一个端口号作为其后缀。

用户名可以由字母、数字和一些标点符号组成。它必须用一个@符号和域名或 IP 地址分隔开。`<[a-z0-9\-\_\~%\$\&'()*\+,;=:]+@>` 会匹配用户名和分隔符。

RFC 3986 对于所允许的域名的规定是相当自由的。实例 7.15 解释了域名一般允许哪些

字符：字母、数字、连字符和点号。RFC 3986 也允许波浪号（~）以及使用百分号引用的其他字符。域名必须转换为 UTF-8，任何非字母、数字、连字符或波浪号的字节必须编码为%*FF*，这里 *FF* 是该字节的十六进制表示形式。

为了使我们的正则表达式更为简洁，我们不检查每个百分号是否恰好跟着两个十六进制数。最好在把 URL 分割成为不同部分之后，再进行类似这样的合法性验证。因此我们只需要用 <[a-z0-9\-.~%]+> 来匹配主机名，它同样会匹配 IPv4 地址（在 RFC 3986 的标准所支持的）。

除了域名或 IPv4 地址之外，主机也可以采用在两个方括号之间的一个 IPv6 地址，或者是未来版本的 IP 地址的方式进行说明。我们用 <\[[a-f0-9:]+\\]> 来匹配 IPv6 地址，用 <\v[a-f0-9][a-z0-9\-.~%!\$&'()\*+,;=:]+\\> 匹配未来可能的 IP 地址。尽管不能用还没定义的未来 IP 地址版本对 IP 地址进行合法性验证，但是我们可以对 IPv6 地址更严格些。但是这样的任务最好还是应该在把地址从 URL 中提取出来之后，留给另外一个正则式来做。实例 7.17 中会展示对 IPv6 地址执行合法性验证的确不是一个很简单的任务。

如果指定了端口号，那么端口号就仅仅是一个用冒号和主机名分隔开来的数字。用 <:[0-9]+> 来表示就足够了。

如果指定了授权，那么它后面必须跟随一个绝对路径，或者完全没有路径。一个绝对路径以正斜杠开始，之后跟着一个或多个用斜杠分隔的段（segment）。段由一个或多个字母、数字或标点符号组成。这里不允许出现连续的正斜杠。路径可以以一个正斜杠作为结束。<(/[a-z0-9\-.~%!\$&'()\*+,;=:@]+)\*?> 会匹配这样的路径。

如果 URL 没有指定授权，那么路径可以是绝对的、相对的或者被省略的。绝对路径由一个正斜杠作为开始，相对路径则不是这样。因为前导的正斜杠现在是可选的，所以我们需要一个稍微长一些的正则式来同时匹配绝对和相对路径：</?[a-z0-9\-.~%!\$&'()\*+,;=:@]+(/[a-z0-9\-.~%!\$&'()\*+,;=:@]+)\*?>。

相对 URL 中并不会指定通信协议方案，因此也不包含授权。路径则是必需的，它可以是绝对的，也可选相对的。因为 URL 没有指定通信协议方案，一个相对路径中的第一段就不能包含任何冒号。否则冒号会被视为通信协议方案的分隔符。这样就需要两个正则表达式来匹配一个相对 URL 的路径。我们用 [a-z0-9\-.~%!\$&'()\*+,;=:@]+(/[a-z0-9\-.~%!\$&'()\*+,;=:@]+)\*?> 匹配相对路径，这和含有通信协议方案但是没有授权的正则式很接近。唯一的不同是没有使用可选的前导正斜杠；另外一个区别是第一个字符类中也没有包含冒号。我们用 <(/[a-z0-9\-.~%!\$&'()\*+,;=:@]+)+?> 匹配绝对路径。这和匹配指定了通信协议方案和授权的 URL 中的路径的正则式是一样的，除了用来重复路径中的段的星号变成了加号。这是因为相对 URL 要求至少包含一个路径段。

URL 的查询部分是可选的。如果存在，那么它必须以一个问号作为开始，直到 URL 的

结尾或者到 URL 中的第一个井号之前。因为井号不在 URL 查询部分的合法标点符号集合中，所以我们可以很容易地用 `\?([a-zA-Z\-.~%!$&'()*+,;=:@/?]*)` 进行匹配。这个正则式中的两个问号是字面意义上的问号。第一个问号在字符类之外，因此必须被转义。第二个问号在字符类之内，所以总是被看成字面意义上的问号符。

URL 中的最后一部分是信息片段 (fragment)，它也是可选的。它以一个井号作为开始，直到 URL 结束。`\#[a-zA-Z\-.~%!$&'()*+,;=:@/?]*` 会匹配信息片段的部分。

为了更容易地处理 URL 中的不同部分，我们使用了命名捕获分组。实例 2.11 解释了在本书中所讨论的不同的正则流派中如何使用命名捕获。.NET 是其中唯一将多个分组用同一个名字进行处理的，就好像它们是一个分组一样。在现在的情况下这种做法会非常实用，因为根据是否指定了通信协议方案和/或授权，我们的正则式有多种途径可以来匹配 URL 路径。如果我们把 3 个分组命名为同一名字，就可以简单地查询“path”分组来获得路径，而不用去管 URL 是否含有通信协议方案或者授权。

其他的流派不支持.NET 中的这种命名捕获行为，尽管多数都支持同样的命名捕获语法。对其他的流派而言，路径的 3 个捕获分组都会拥有不同的名字。当一个匹配被找到时，只有其中之一才能真正作为 URL 的路径。另外两个分组则不会参与到匹配中。

## 参见

实例 2.3、实例 2.8、实例 2.9 和实例 2.12。

## 7.8 从 URL 中提取通信协议方案

### 问题描述

你要从一个作为 URL 的字符串中提取其中的 URL 通信协议方案。例如，你想要从 `http://www.regexcookbook.com` 中抽取到 `http`。

### 解决方案

从已知是合法的 URI 中抽取通信协议方案

```
^([a-zA-Z][a-zA-Z0-9+\-.]*):  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

在对 URL 做合法性验证的同时抽取通信协议方案

```
\A  
([a-zA-Z][a-zA-Z0-9+\-.]*):  
(# 授权和路径  
//
```

```

([a-zA-Z\-.~%!$&^()^+,;=]+@)? # 用户名
([a-zA-Z\-.~%]+ # 命名主机
|\[[a-f0-9:.\]]+\] # IPv6 主机
|\v[a-f0-9][a-zA-Z\-.~%!$&^()^+,;=:@]+\]) # IPvFuture 主机
(:[0-9]+)? # 端口
(/[a-zA-Z\-.~%!$&^()^+,;=:@]+/*?) # 路径
| # 不包含授权的路径 (/?[a-zA-Z\-.~%!$&^()^+,;=:@]+(/[a-zA-Z\-.~%!$&^()^+,;=:@]+/*?))?
)
# 查询
(\?/[a-zA-Z\-.~%!$&^()^+,;=:@/?]*?)?
# 片段
(\#[a-zA-Z\-.~%!$&^()^+,;=:@/?]*?)?
\Z

正则选项: 不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^([a-zA-Z][a-zA-Z0-9+\-.]*)://(([a-zA-Z\-.~%!$&^()^+,;=]+@)?([a-zA-Z\-.~%]+|\-
|\[[a-f0-9:.\]]+\]|\v[a-f0-9][a-zA-Z\-.~%!$&^()^+,;=:@]+\])(:[0-9]+)?\-
(/[a-zA-Z\-.~%!$&^()^+,;=:@]+/*?|(/?[a-zA-Z\-.~%!$&^()^+,;=:@]+|\-
(/[a-zA-Z\-.~%!$&^()^+,;=:@]+/*?))?) (\?/[a-zA-Z\-.~%!$&^()^+,;=:@/?]*?)?\-
(\#[a-zA-Z\-.~%!$&^()^+,;=:@/?]*?)?$

正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python

```

## 讨论

如果已知你的目标文本是一个合法的 URL，那么从 URL 中抽取通信协议方案并不难。假如 URL 中含有通信协议方案，那么它总是会出现在 URL 的开端。在这个正则式中的脱字符（实例 2.5）就可以满足这个要求。通信协议方案以字母开始，后面可以跟随更多的字母、数字、加号、连字符和点号。我们可以用 2 个字符类来匹配它：`<[a-zA-Z][a-zA-Z0-9+\-.]*>`（参见实例 2.3）。

通信协议方案和 URL 的其余部分之间以一个冒号作为分隔。我们把冒号加到正则式中，以确保在 URL 真正从一个通信协议方案开始时才会匹配通信协议方案。相对 URL 不是以通信协议方案开始的。RFC 3986 规范中的 URL 规定了相对 URL 中不能含有任何冒号，除非这些冒号前面出现的是通信协议方案中不允许出现的字符。这就是为何在实例 7.7 中我们要把冒号排除在匹配路径的字符类中的原因。如果你在一个合法但是相对的 URL 之上使用本实例中的正则式，就不会找到任何一个匹配。

因为正则式所匹配的不是通信协议方案本身（还包含了冒号），所以我们要在正则表达式上添加一个捕获分组。当正则式找到一个匹配时，你可以从第一个（也是唯一的一个）捕获分组中提取所匹配的文本来获得不含有冒号的通信协议方案。实例 2.9 告诉你关于捕获分组的所有相关内容。关于如何使用常用的编程语言来获取捕获分组所匹配

到的文本，请参阅实例 3.9。

如果还不知道你的目标文本是否是一个合法的 URL，那么可以使用实例 7.7 中的正则式的一个简化版本。由于我们需要抽取的是通信协议方案，因此就可以排除不指定通信协议方案的相对 URL。这样就可以得到一个稍微简单一些的正则表达式。

由于正则式匹配到的是整个 URL，所以我们在匹配通信协议方案的正则式部分之外需要再加上一个捕获分组。提取捕获分组 #1 所匹配到的文本就可以获得该 URL 中的通信协议方案。

## 参见

实例 2.9、实例 3.9 和实例 7.7。

## 7.9 从 URL 中抽取用户名

### 问题描述

你要从包含一个 URL 的字符串中抽取出其中的用户名。例如，你想要从 `ftp://jan@www.regexcookbook.com` 中抽取出其中的用户名 `jan`。

### 解决方案

#### 从已知是合法的 URL 中抽取用户名

```
^[a-zA-Z0-9+\-.]+\://(([a-zA-Z0-9\-\.\~%\!$&'\(\)\*\+,;=]+)@  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

#### 在对 URL 做合法性验证的同时抽取用户名

```
\A  
[a-zA-Z][a-zA-Z0-9+\-.]*://  
(([a-zA-Z0-9\-\.\~%\!$&'\(\)\*\+,;=]+)@  
(([a-zA-Z0-9\-\.\~%]+  
|\\[[a-f0-9\:\.]+\\])  
|\\v[a-f0-9][a-zA-Z0-9\-\.\~%\!$&'\(\)\*\+,;=:]+\\])  
(:[0-9]+)?  
(/([a-zA-Z0-9\-\.\~%\!$&'\(\)\*\+,;=:@]+)+/?  
(\\?([a-zA-Z0-9\-\.\~%\!$&'\(\)\*\+,;=:@/?]+)?  
(\\#[a-zA-Z0-9\-\.\~%\!$&'\(\)\*\+,;=:@/?]+)?  
\Z  
正则选项：不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-\.\~%\$\&'()*\+,;=:]+)@([a-z0-9\-\.\~%]+|←  
\[[a-f0-9:\.]+\]|\[v[a-f0-9][a-z0-9\-\.\~%\$\&'()*\+,;=:]+\])(:[0-9]+)?←  
(/([a-z0-9\-\.\~%\$\&'()*\+,;=:@]+)*?(\?([a-z0-9\-\.\~%\$\&'()*\+,;=:@/?]+)*?)?  
(\#[a-z0-9\-\.\~%\$\&'()*\+,;=:@/?]+)*$)
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

## 讨论

如果已知你的目标文本是一个合法的 URL，那么从 URL 中抽取用户名并不难。假如在 URL 中含有用户名，那么它总是会紧跟在通信协议方案以及以两个斜杠号开始的 URL 的“授权”部分的后面。用户名会以一个@符号和紧随其后的主机名分隔开。由于主机名中不允许出现@符号，因此假如在两个斜杠符和 URL 中下一个斜杠符号之间找到了一个@，就可以在 URL 中抽取出用户名的部分。因为在用户名中不允许出现斜杠，所以我们不需要对斜杠做特殊处理。

如果已知 URL 是合法的，以上这些规则意味着抽取用户名会很容易。我们只要用<[a-z0-9+\-.]+>跳过通信协议方案，再跳过://。然后，就可以找到紧随其后的用户名。如果我们匹配到了@号，就可以知道在它之前的部分就是用户名。字符类<[a-z0-9+\-.]+>列出了在用户名中的所有合法字符。

本正则式只能在 URL 确实指定了用户名的时候可以找到一个匹配。如果它能找到匹配的话，那么正则式会同时匹配通信协议方案和 URL 的用户名部分。因而，我们在正则表达式中需要加上一个捕获分组。当这个正则式找到一个匹配时，你可以提取第一个（也是唯一的）捕获分组中所匹配到的文本，从中得到没有任何分隔符和 URL 其他部分的用户名。关于捕获分组的知识都可以在实例 2.9 中找到。关于如何使用习惯的编程语言提取捕获分组匹配的文本，可以参考实例 3.9。

如果尚不明确你的目标文本是否是一个合法的 URL，可以使用实例 7.7 中正则表达式的一个简化版本。既然想要抽取的只是用户名，我们可以排除不包含授权的 URL。在前面解决方案中的正则式实际上只匹配了指定授权且在授权中包含一个用户名的 URL。如果要求在 URL 中必须包含授权部分，就会让这个正则表达式简单不少，甚至比实例 7.7 中的正则式还要简单。

因为这个正则式匹配了整个 URL，所以我们在匹配用户名的正则式部分周围还需要添加一个捕获分组。提取第一个捕获分组所匹配到的文本可得到 URL 中的用户名。

如果希望正则式匹配任意的合法 URL，其中也要包括那些没有指定用户名的 URL，那么你可以选取实例 7.7 中的一个正则式。假如 URL 中存在用户名，该实例中的第一个正则式的第 3 个捕获分组就会捕获到用户名。在该捕获分组中会包含@符号。如果你想捕获不包含@符号的用户名，那么可以向这个正则表达式中再添加一个额外的捕获分组。

## 参见

实例 2.9、实例 3.9 和实例 7.7。

## 7.10 从 URL 中抽取主机名

### 问题描述

你要从包含一个 URL 的字符串中抽取其中的主机名（域名或 IP 地址）。例如，你想要从 <http://www.regexcookbook.com/> 中抽取出其中的 www.regexcookbook.com。

### 解决方案

#### 从已知是合法的 URL 中抽取主机名

```
\A
[a-z][a-z0-9+\-.]*://          # 通信协议方案
([a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:]+@)?      # 用户名
([a-z0-9\-\_\~%]+            # 命名的或 IPv4 主机
|\[[[a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:]+\]])      # IPv6+ 主机

正则选项: 宽松排列、不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

^[a-z][a-z0-9+\-.]*://(([a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:]+@)?([a-z0-9\-\_\~%]+\|+
\[[[a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:]+\]))      # 通信协议方案
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

#### 在对 URL 做合法性验证的同时抽取主机名

```
\A
[a-z][a-z0-9+\-.]*://          # 通信协议方案
([a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:]+@)?      # 用户名
([a-z0-9\-\_\~%]+            # 命名主机
|\[[a-f0-9:\.]+\])          # IPv6 主机
|\[[v[a-f0-9][a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:]+\]])      # IPvFuture 主机
(:[0-9]+)?                  # 端口
(/[a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:@]+)*/?      # 路径
(\?([a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:@/?]*))?      # 查询
(\#[a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:@/?]*))?      # 片段
\Z

正则选项: 不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^[a-z][a-z0-9+\-.]*://(([a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:]+@)?([a-z0-9\-\_\~%]+\|+
\[[a-f0-9:\.]+\])\[[v[a-f0-9][a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:]+\]](:[0-9]+)?\|
(/[a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:@]+)*?\(\?([a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:@/?]*))?\|
(\#[a-z0-9\-\_\~%\$\&'\(\)\*\+,;=:@/?]*))?\$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

## 讨论

如果已知目标文本是一个合法的 URL，那么从 URL 中抽取主机名并不难。我们使用 `\A` 或 `\^` 作为定位符把匹配固定到字符串的开头。使用 `<[a-z][a-z0-9+\-.]*://>` 跳过通信协议方案，使用 `<([a-z0-9\-.~%!$&'()*+,;=:]+@[a-z0-9\-.~%!$&'()*+,;=:]+\.)?>` 跳过可选的用户名部分。主机名就是紧随其后的那部分内容。

RFC 3986 中允许主机有两种不同的标记方式。主机可以指定为域名和 IPv4 地址，在两边不需要使用方括号；而如果是 IPv6 地址和未来的 IP 地址，就需要用方括号包起来。因为带方括号的表示法比不带方括号的允许出现更多的标点符号，所以我们需要对它们进行分别处理。特别是，在方括号之间允许出现冒号，但是域名或 IPv4 地址不允许有冒号出现。冒号也可以用来把端口号和（带方括号的和不带方括号的）主机名分隔开。

`<[a-z0-9\-.~%]+>` 可以匹配域名和 IPv4 地址。`\[[a-z0-9\-.~%!$&'()*+,;=:]+\]` 处理 IPv6 和以后的 IP 地址版本。我们在一个分组中使用多选结构（实例 2.8）将它们组合在一起。捕获分组也让我们能抽取出主机名。

只有当 URL 确实指定了主机的时候，本正则式才能找到一个匹配。这个正则式会同时匹配到通信协议方案、用户和 URL 中的主机部分。当正则式找到一个匹配时，你可以提取在第二个捕获分组中匹配到的文本，获得不含分隔符和 URL 其他部分的主机名。如果匹配到的是一个 IPv6 地址，那么在捕获分组中就会包括方括号。关于捕获分组的知识都可以在实例 2.9 中找到。关于在你习惯使用的编程语言中如何提取捕获分组匹配到的文本，可以参看实例 3.9。

如果你不清楚目标文本中包含的是否是一个合法的 URL，那么可以使用实例 7.7 中正则式的一个简化版本。既然我们希望抽取的是主机名，就可以先排除没有指定授权的 URL。这样可以使这个正则表达式简单不少。它和我们在实例 7.9 中用到的正则式非常相像。唯一的区别是这次授权中的用户部分也是可选的，这与在实例 7.7 中是一样的。

正则式还为主机的不同标记方式使用了多选结构，在多选结构之外使用了一个捕获分组。提取第二个捕获分组所匹配到的文本，就可以获得 URL 中的主机名。

如果希望这个正则式可以匹配任意的合法 URL，其中也包括那些没有指定主机名的 URL，那么你可以选取实例 7.7 中的一个正则表达式。假如 URL 中存在主机名，那么该实例中的第一个正则式的第 4 个捕获分组会捕获到主机。

## 参见

实例 2.9、实例 3.9 和实例 7.7。

## 7.11 从 URL 中抽取端口号

### 问题描述

你要从一个包含 URL 的字符串中抽取出其中的端口号。例如，你想要从 `http://www.regexcookbook.com:80/` 中抽取出端口号 80。

### 解决方案

#### 从已知是合法的 URL 中抽取端口号

```
\A
[a-z][a-zA-Z0-9+\-.]*://          # 通信协议方案
([a-zA-Z0-9\-.~%!$&`()^+,;=:]+@)? # 用户名
([a-zA-Z0-9\-.~%]+               # 命名的或 IPv4 主机
|\[[a-zA-Z0-9\-.~%!$&`()^+,;=:]+\]) # IPv6+ 主机
:(?<port>[0-9]+)                # 端口号

正则选项: 宽松排列、不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

^[a-z][a-zA-Z0-9+\-.]*://([a-zA-Z0-9\-.~%!$&`()^+,;=:]+@)?  

([a-zA-Z0-9\-.~%]+|\[[a-zA-Z0-9\-.~%!$&`()^+,;=:]+\]):([0-9]+)

正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

#### 在对 URL 做合法性验证的同时抽取端口号

```
\A
[a-z][a-zA-Z0-9+\-.]*://          # 通信协议方案
([a-zA-Z0-9\-.~%!$&`()^+,;=:]+@)? # 用户名
([a-zA-Z0-9\-.~%]+               # 命名主机
|\[[a-f0-9\-.~%]+]\])          # IPv6 主机
|\[[a-f0-9][a-zA-Z0-9\-.~%!$&`()^+,;=:]+\]) # IPvFuture.主机
:(?<port>[0-9]+)                # 端口
(/[a-zA-Z0-9\-.~%!$&`()^+,;=:@]+)*? # 路径
(\?([a-zA-Z0-9\-.~%!$&`()^+,;=:@/?]+)*?)? # 查询
(\#[a-zA-Z0-9\-.~%!$&`()^+,;=:@/?]+)? # 片段
\Z

正则选项: 不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^[a-z][a-zA-Z0-9+\-.]*://([a-zA-Z0-9\-.~%!$&`()^+,;=:]+@)?  

([a-zA-Z0-9\-.~%]+|\[[a-f0-9\-.~%]+]\]|[\[[a-f0-9][a-zA-Z0-9\-.~%!$&`()^+,;=:]+\]]  

|[\[[a-f0-9][a-zA-Z0-9\-.~%!$&`()^+,;=:]+\]):([0-9]+)(/[a-zA-Z0-9\-.~%!$&`()^+,;=:@]+)*?  

(\?([a-zA-Z0-9\-.~%!$&`()^+,;=:@/?]+)*?)?(\#[a-zA-Z0-9\-.~%!$&`()^+,;=:@/?]+)?$
```

**正则选项:** 不区分大小写

**正则流派:** .NET、Java、JavaScript、PCRE、Perl、Python

## 讨论

如果已知目标文本是一个合法的 URL，那么从 URL 中抽取端口号并不难。我们使用 `\A` 或 `\^` 将匹配定位到字符串的开头。使用 `<[a-z][a-z0-9+\-\.]*://>` 跳过通信协议方案，使用 `<([a-z0-9\-\_~%\!$&'()*\+,;=]+@)?>` 跳过可选的用户名部分，并且使用 `<([a-z0-9\-\_~%]+|[a-z0-9\-\_~%\!$&'()*\+,;=:]+\])>` 跳过主机名。

端口号与主机名之间会用一个冒号分隔开来，我们在正则表达式中把冒号作为一个字面上的字符处理。端口号本身只是个单纯的数字字符串，因此可以简单地用 `[0-9]+` 进行匹配。

只有在 URL 中指定了一个端口号时，这个正则式才能找到匹配。这个正则式会同时匹配 URL 的通信协议方案、用户名、主机和端口部分。

当正则式找到了一个匹配的时候，你可以提取第 3 个捕获分组所匹配到的文本，从而可以获得不含任何分隔号或 URL 其他部分的端口号。

其他 2 个分组一个是为了让用户名可选，另一个是为了让主机名的 2 个选择分支组合到一起。关于捕获分组的讲解，请参考实例 2.9。如果想了解如何在习惯使用的编程语言中来提取捕获分组所匹配到的文本，请参考实例 3.9 中的讲解。

如果你尚不清楚目标文本是否是一个合法的 URL，那么可以使用实例 7.7 中正则式的一个简化版本。既然我们希望抽取的是端口号，那么就可以排除掉不包含授权的 URL。这样可以使正则表达式简单不少。它和我们在实例 7.10 中用到的正则式非常相像。

在这里的唯一区别是它的端口号不再是可有可无的，并且我们把端口号的捕获分组移动了位置，排除了把端口号和主机名分隔开来的冒号。捕获分组的序号是第 3 个。

如果想要用一个正则式来匹配任意合法 URL，其中也包括那些没有指定端口号的 URL，那么你可以使用实例 7.7 中的一个正则式。如果端口号存在，那么该实例中的第一个正则式的第 5 个捕获分组中会捕获端口号。

## 参见

实例 2.9、实例 3.9 和实例 7.7。

## 7.12 从 URL 中抽取路径

### 问题描述

你想要从一个包含 URL 的字符串中抽取其中的路径。例如，你想要从 `http://www.regexcookbook.com/index.html` 或者从 `/index.html#fragment` 中抽取其中的路径 `/index.html`。

## 解决方案

从一个已知是合法的 URL 中抽取路径。下面的正则式会匹配所有的 URL，其中也包括不含路径的 URL：

```
\A
# 略过通信协议方案和授权（如果有） .
([a-z][a-z0-9+\-.]*:(//[^/?#]+)+)??
# 路径
([a-z0-9\-\_\~%\$\&'()*\+,;=:@/]*)
正则选项：宽松排列、不区分大小写
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

^([a-z][a-z0-9+\-.]*:(//[^/?#]+)+)?([a-z0-9\-\_\~%\$\&'()*\+,;=:@/]*)
正则选项：不区分大小写
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

从一个已知是合法的 URL 中抽取路径。只匹配确实包含路径的 URL：

```
\A
# 略过通信协议方案和授权（如果有）
([a-z][a-z0-9+\-.]*:(//[^/?#]+)+)??
# 路径
(/?[a-z0-9\-\_\~%\$\&'()*\+,;=@]+(/[a-z0-9\-\_\~%\$\&'()*\+,;=:@]+)*/?|/)
# 查询、信息片段或 URL 结束
([#?]|\.Z)
正则选项：宽松排列、不区分大小写
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

^([a-z][a-z0-9+\-.]*:(//[^/?#]+)+)?(/?[a-z0-9\-\_\~%\$\&'()*\+,;=@]+<
(/?[a-z0-9\-\_\~%\$\&'()*\+,;=:@]+)*/?|/) ([#?]|$)
正则选项：不区分大小写
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

从一个已知是合法的 URL 中抽取路径。使用原子分组来只匹配实际上包含路径的 URL：

```
\A
# 略过通信协议方案和授权（如果有）
(?>([a-z][a-z0-9+\-.]*:(//[^/?#]+)+))?
# 路径
([a-z0-9\-\_\~%\$\&'()*\+,;=:@/]++)
正则选项：宽松排列、不区分大小写
正则流派：.NET、Java、PCRE、Perl、Ruby
```

## 讨论

如果已知目标文本是一个合法的 URL，那么你就可以使用一个非常简单的正则表达式抽取路径。根据 URL 是否会指定通信协议方案和/或授权，实例 7.7 中的通用正则式有 3 种不同的方式可以匹配路径，然而从已知是合法的 URL 中抽取路径的特定正则式只需要匹配路径一次。

我们使用 `\A` 或 `\^` 将匹配定位到字符串的开头。使用 `<[a-z][a-z0-9+\-.]>*://>` 跳过通信协议方案，并且使用 `</[^/?#]+>` 跳过授权。鉴于已经知道 URL 是合法的，并且我们对从授权中抽取用户名、主机和端口号不感兴趣，因此就可以为授权使用非常简单的正则式。

授权以两个正斜杠作为开始，一直匹配到路径的开始（正斜杠）、查询的开始（问号）或者是片段的开始（井号）。其中的否定字符类会匹配下一个正斜杠、问号或井号之前的一切内容（参考实例 2.3）。

因为授权是可选的，所以我们会把它放进一个分组中，在它的后面紧跟着一个问号：`<(//[^/?#]+)?>`。通信协议方案也是可选的。如果通信协议方案被省略，那么授权就必须被省略。为了匹配这一点，我们把正则式中匹配通信协议方案和可选的授权放到另外一个分组中，同时用一个问号把它也变成是可选的。

因为 URL 已知是合法的，所以我们可以很容易地用单个字符类来匹配包括正斜杠的路径：`<[a-z0-9\-\_\~%\!$&'()^+;=:@/]*>`。这里不需要检查两个连续的斜杠，因为在 URL 中不允许存在这样的连续斜杠。

匹配路径的字符类事实上使用星号而不是加号作为量词。对于一个目的只是抽取 URL 中路径的正则式而言，把路径设为可选的可能有些奇怪。然而实际上，把路径设为可选是至关重要的，因为我们采用了跳过通信协议方案和授权的简洁做法。

在实例 7.7 的通用正则式中，根据 URL 中是否存在通信协议方案和/或授权，我们会有 3 种方法可以匹配路径。这使得通信协议方案不会意外地被匹配为路径。

为了努力保持正则式的简单，我们只使用了一个字符类来匹配路径。假设我们有一个 URL 是 `http://www.regexcookbook.com`，其中包含了通信协议方案和授权，但没有包含路径。我们的正则式的第一部分无疑会匹配到通信协议方案和授权。正则式引擎会接着努力匹配路径的字符类，但是已经不剩下什么字符了。如果路径是可选的（使用星号量词），正则式引擎就毫无疑问地不为路径匹配任何字符。当遇到正则式结尾时，就宣布找到了一个整体的匹配。

但是如果路径的字符类不是可选的，那么正则式引擎就会回溯。（如果不熟悉回溯，可以参考实例 2.13）。由于我们的正则式中授权和通信协议方案是可选的，于是引擎会决定再尝试一次，而不会让 `<(//[^/?#]+)?>` 匹配到任何东西。这样 `<[a-z0-9\-\_\~%\!$&'()^+;=:@/]*>` 就会匹配到 `//www.regexcookbook.com` 为路径，这显然不是我们想要的。如果为路径使用一个更为准确的正则式，不允许出现成对的正斜杠，那么正则式引擎就会简单地再次回溯，这次会假设 URL 中不包含通信协议方案。对路径使用一个更加准确的正则式，它可能会把 `http` 匹配为路径。为了防止这一点，我们要再加上一个检查，确保路径后跟随着查询、片段或者什么也没有。如果做到了全部这些检查，那么我们得到的就是本实例的“解决方案”中标明为“只匹配实际含有路径的 URL”

的正则表达式。这个正则表达式比前面两个要更复杂一些，而这些增加的复杂性只是为了让正则式不要匹配不含路径的 URL。

如果你的正则流派支持原子分组，那么就可以使用更简单的办法。本书中讨论的所有流派除了 JavaScript 和 Python 外，都支持原子分组（参见实例 2.15）。本质上来说，一个原子分组的首要作用是告知正则式引擎不需要回溯。如果我们把正则式的通信协议方案和授权部分放到一个原子分组里，即使匹配路径的字符类没有余下的可匹配空间，正则引擎也会被强迫保留通信协议方案和授权部分所匹配到的内容。这种解决方案与把路径改成可选的一样高效。

不管你在本实例中选择了哪种正则表达式，路径都会保存在第 3 个捕获分组中。如果你使用了前两个允许路径是可选的正则式，那么第 3 个捕获分组就可能会返回空字符串，或者是在 JavaScript 中返回 null。

如果你并不知道目标文本是一个合法的 URL，那么可以使用实例 7.7 中的正则式。如果你使用的是 .NET，那么你可以使用 .NET 专用的正则式，其中包含了 3 个名为“path”的分组，它们会分别捕获可能会匹配到 URL 路径的 3 个正则式组成部分。如果你使用的是另外一种支持命名捕获的流派，那么路径会被捕获到如下 3 个分组之一：“hostpath”、“schemepath”或者“relpath”。因为 3 个路径中只有一个实际会捕获到真正的路径，因此可以用一个简单的小窍门，把这 3 个分组所返回的字符串连接起来就可以获得路径。因为它们中的两个一定会返回空字符串，所以并不会有实际的连接发生。

如果你喜欢使用的编程语言并不支持命名捕获，那么可以使用实例 7.7 中的第一个正则式。它会在分组 6、7 或 8 中捕获路径。你可以用同样的小窍门来连接 3 个分组获得的文本，因为它们中的 2 个只会返回空字符串。不过在 Javascript 中这样就行不通。JavaScript 会对没有实际参与匹配的分组返回 undefined。

关于在你喜欢使用的编程语言中，如何从命名和编号捕获分组所匹配的文本中提取文本，可以参考实例 3.9 中的讲解。

## 参见

实例 2.9、实例 3.9 和实例 7.7。

## 7.13 从 URL 中抽取查询

### 问题描述

你想要从一个包含 URL 的字符串中抽取其中的查询（query）。例如，你想要从 `http://www.regexcookbook.com? param=value` 或者 `/index.html? param=value` 中抽取其中的 param=value。

## 解决方案

`^[^?#]+\?\([^\#]+\)`  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

如果已知你的目标文本是个合法的 URL，那么从 URL 中抽取查询是件很轻松的事。查询和它之前的 URL 部分之间会用一个问号进行分隔。这是 URL 中第一个允许出现问号的地方。因此，我们可以用 `<[^?#]+\?\>` 简单地跳过第一个问号之前的部分。只有在字符类之外的问号才是个元字符，而在字符类内则不是。所以我们要为这个字符类外的字面上的问号进行转义。第一个 `\` 是一个定位符（实例 2.5），第二个 `\` 则用来对字符类取反（实例 2.3）。

在查询之后，问号可以作为（可选的）片段中的一部分出现在 URL 中，所以只用 `\?\>` 是不行的，我们需要使用 `<[^?#]+\?\>` 以确保所获得的是在 URL 中的第一个问号，从而可以确保这个问号并不会出现在不包含查询的 URL 的片段中。

查询的部分一直延伸到片段开始部分或者 URL 的结尾为止。片段会用一个井号和 URL 的其余部分隔开。因为井号只能在片段部分出现，所以我们只需要用 `<[^#]+\>` 来匹配查询。这个否定字符类匹配到第一个井号出现前的所有字符，当目标文本不含有任何井号的时候，它会匹配到目标字符串结尾为止。

只有当 URL 确实含有一个查询时，这个正则表达式才能找到一个匹配。当匹配到一个 URL 时，匹配中会包括从 URL 开端开始的任何字符，所以我们需要把正则式匹配到的查询的部分 `<[^#]+\>` 放到一个捕获分组中。当正则式找到匹配时，你可以提取第一个（也是唯一的）捕获分组来获取匹配到的文本，就可以得到不含任何分隔符或者 URL 其他部分的查询部分。关于捕获分组的讲解请参考实例 2.9。关于在习惯使用的编程语言中如何提取捕获分组匹配到的文本，请参看实例 3.9。

如果你尚不清楚目标文本是否是一个合法的 URL，那么可以使用实例 7.7 中正则式中的任意一个。当查询存在于 URL 中时，该实例中的第一个正则式会把查询捕获到第 12 个捕获分组中。

## 参见

实例 2.9、实例 3.9 和实例 7.7。

## 7.14 从 URL 中抽取片段

### 问题描述

你想要从一个包含 URL 的字符串中抽取其中的片段（fragment）。例如，你想要从

<http://www.regexcookbook.com#top> 或者 </index.html#top> 中抽取其中的 top。

## 解决方案

```
# (.+)
```

正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

如果已知你的目标文本是个合法的 URL，那么从 URL 中抽取片段是件很轻松的事。片段和它之前的 URL 部分之间会用一个井号进行分隔。片段是 URL 中第一个允许出现井号的地方，而且片段总是在 URL 中最后出现的部分。因此，我们可以用 `\#.+` 简单地跳过第一个井号之前的部分，然后把直到字符串结束之前的所有内容都抓下来。一定要记住关掉宽松排列模式；否则，你就需要对字面的井号用一个反斜杠进行转义。

只有当 URL 确实含有一个片段的时候，这个正则表达式才能找到一个匹配。这个匹配中只会包含片段中的内容，但是会包含把片段和 URL 其余部分进行分隔的井号。这个解决方案中使用了一个额外的捕获分组，只获取片段，把用来分隔的井号去掉。

如果你尚不清楚目标文本是否是一个合法的 URL，那么可以使用实例 7.7 中正则式中的任意一个。当片段存在于 URL 中时，该实例中的第一个正则式会把片段捕获到第 13 个捕获分组中。

## 参见

实例 2.9、实例 3.9 和实例 7.7。

## 7.15 域名合法性验证

### 问题描述

你要检查一个字符串是否看起来像一个有效的全限定（fully-qualified）域名，或者在一大段文本中查找一个这样的域名。

### 解决方案

检查一个字符串是否看起来像一个有效域名：

```
^([a-zA-Z]+(-[a-zA-Z]+)*\.)+[a-zA-Z]{2,}$
```

正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

```
\A([a-zA-Z]+(-[a-zA-Z]+)*\.)+[a-zA-Z]{2,}\Z
```

正则选项：不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

在更长的文本中查找合法的域名：

```
\b([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\b
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

检查域名的每个组成部分的长度是否不超过 63 个字符：

```
\b((?= [a-z0-9-]{1,63}\. ) [a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,63}\b
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

允许使用 Punycode 标准编码的国际化域名：

```
\b((xn--)?[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\b
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

检查域名的每个部分的长度是否不超过 63 个字符，并且允许使用 Punycode 标准编码的国际化域名：

```
\b((?= [a-z0-9-]{1,63}\. )(xn--)?[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,63}\b
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

域名的格式可以是 domain.tld、subdomain.domain.tld 或者再加上任何数量的子域名。顶级域名（tld）由 2 个或更多字母组成。这是正则式中最简单的部分：`\b([a-z]{2,})\b`。

域名，或者任何子域名，都是由字母、数字和连字符组成的。连字符不能成对出现，也不能作为域名中的第一个或者最后一个字符。我们可以用正则表达式 `\b([a-z0-9]+(-[a-z0-9]+)*)\b` 来匹配这样的模式。这个正则式会匹配任何长度的字母和数字组合，后面可以跟随可选的任意多个分组，每个分组中包含一个连字符以及紧跟其后的字母数字序列。记住，连字符如果出现在字符类中是一个元字符（实例 2.3），但是在字符类外它就是一个普通字符，所以我们不需要在正则式中对连字符进行转义。

域名和子域名由字面上的点号分隔开，在正则表达式中以 `\.` 表示。既然域名后可添加任意数量的子域名，我们把正则式的域名部分和字面上的点号放在一个重复的分组中：`\b([a-z0-9]+(-[a-z0-9]+)*\.)+\b`。由于子域名和域名遵从同一语法，所以只需使用一个分组就可以处理子域名和域名这两种情况。

如果你想检查一个字符串是否包含一个有效的域名，剩下要做的只是需要把定位符加在正则式的开端和结尾处，用来匹配字符串的开端和结尾。除了 Ruby 之外的所有正则

流派都可以使用 `\^` 和 `\$`，而除了 JavaScript 之外的所有流派都可以使用 `\A` 和 `\Z`。更多信息请参见实例 2.5。

如果要在一大段文本中查找域名，那么你可以在正则式两边添加单词边界（`\b`；请参考实例 2.6）。

第一组正则表达式没有检查域名中的每个组成部分是否包含多于 63 个字符。做到这一点并不太容易，因为正则式中表示每个域名组成部分的正则式 `\[a-zA-Z]+\(-[a-zA-Z]+\)*` 中包含 3 个量词。因此我们无法告知正则式引擎这些部分加起来不超过 63 个字符。

我们可以使用 `\[-a-zA-Z]\{1,63\}` 匹配一个长度为 1~63 个字符的域名组成部分，或者对整个域名使用 `\b\[-a-zA-Z]\{1,63\}\.+\[a-zA-Z]\{2,63\}`。但是这样就无法排除连字符位置不正确的字符串了。

要解决这个问题，我们可以使用顺序环视来两次匹配同样的文本。如果不熟悉顺序环视，可以参阅实例 2.16。我们使用同样的正则式 `\[a-zA-Z]+\(-[a-zA-Z]+\)*\.` 匹配带有合法连字符的域名，并把 `\[-a-zA-Z]\{1,63\}\.` 放在顺序环视中，用来检查它的长度是否同样少于或等于 63 个字符。这样我们就可得到如下的正则表达式：`\((?=\[-a-zA-Z]\{1,63\}\.)\[-a-zA-Z]+\(-[a-zA-Z]+\)*\.)`

顺序环视 `\((?=\[-a-zA-Z]\{1,63\}\.)\)` 首先会检查在下一个点号之前会包含 1~63 个字母、数字和连字符。在这里把点号包含在顺序环视中是非常重要的。如果你不这样做的话，那么多于 63 个字符的域名依然可以满足顺序环视对 63 个字符的要求。只有把字面上的点号放在顺序环视中，我们才能真正满足最多 63 个字符的需求。

顺序环视并不会消耗它所匹配到的字符。所以，如果顺序环视匹配成功，`\[-a-zA-Z]+\(-[a-zA-Z]+\)*\.` 就会被匹配到顺序环视已经匹配到的同一段文本上。我们已经确认了这段文本的长度不会超过 63 个字符，现在就可以检查它是否为正确的连字符和非连字符组合。

国际化域名理论上可以包含几乎所有字符。实际上可以包含的字符列表则是由管理顶级域名的注册机构决定的。例如，.es 允许在域名中包含西班牙语字符。

在实践中，国际化域名常使用所谓 *punycode* 编码标准。虽然 punycode 算法相当复杂，但是我们在这里关心的是，按照它的规定，域名可以是任何字母、数字和连字符的组合，这与我们在域名正则表达式中已经使用过的那些规则是一样的。唯一不同的是，由 punycode 生成的域名会包含一个前缀 `xn--`。为了在我们的正则式中支持这些域名，我们只需要在正则式中的匹配域名组成部分的分组内添加 `\((xn--)\?)`。

## 参见

实例 2.3、实例 2.12 和实例 2.16。

## 7.16 匹配 IPv4 地址

### 问题描述

你要检查某个特定字符串是否包含了一个以 255.255.255.255 标记法来表示的 IPv4 地址。另外，你还想把这个地址转化为一个 32 位整数。

### 解决方案

#### 正则表达式

检查一个 IP 地址的简单正则式：

```
^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

检查一个 IP 地址的精确正则式：

```
^(?:(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

从一大段文本中抽取 IP 地址的简单正则式：

```
\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

从一大段文本中抽取 IP 地址的精确正则式：

```
\b(?:(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\b  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

捕获 IP 地址中的 4 个部分的简单正则式：

```
^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

捕获 IP 地址中的 4 个部分的精确正则式：

```
^(25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)$  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## Perl

```
if ($subject =~ m/^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})/){  
    $ip = $1 << 24 | $2 << 16 | $3 << 8 | $4;  
}
```

## 讨论

一个 IPv4 地址通常会写成 255.255.255.255 的形式，其中的 4 个数字每个都必须在 0~255 之间。匹配这样的 IP 地址所需要使用的正则表达式是非常直观的。

在解决方案中我们展示了 4 种正则表达式，其中 2 个可以归为“简单”的，另外 2 个被标为“精确”的。

简单的正则式版本使用了 <[0-9]{1,3}> 来逐个匹配 IP 地址中的 4 段数字。它们实际上允许出现 0~999 之间的数字，而不只是 0~255。当你已知输入文本中只包含合法 IP 地址的时候，这些简单的正则式会更高效，你需要做的只是把 IP 地址和其他内容分开即可。

精确的正则式则使用 <25[0-5]>、<2[0-4]>、<[01]?> 和 <[0-9]> 分别逐个匹配 IP 地址中的 4 个数字。这个正则式会精确地匹配 0~255 范围内的数字，对于 10~99 之间的数字，前面可以包含一个可选的 0；而对于 0~9 之间的数字，前面可以包含两个可选的 0。<25[0-5]> 会匹配 250~255，<2[0-4]> 匹配 200~249，而 <[01]?> 会匹配 0~199，其中也包括了可选的前缀 0。实例 6.5 中详细讲解了如何使用一个正则表达式来匹配数字区间。

如果你要检查整个字符串是否是一个合法的 IP 地址，就需要使用一个以脱字符开始、以美元符结束的正则式。它们分别是字符串开始和字符串结束的定位符，相关的详细信息，请参考实例 2.5。如果你要在一大段文本中查找一个 IP 地址，那么就需要使用一个以单词边界 <\b>（实例 2.6）作为开始和结束的正则式。

前面 4 个正则表达式中使用了 <(?:number\.){3}number> 的形式。在 IP 地址中的前 3 个数字会用一个非捕获分组来匹配（实例 2.9），它可以重复 3 次（实例 2.12）。该分组会匹配字符和一个字面上的点号，而在一个 IP 地址中总是会包含 3 个点号。正则式的最后一部分匹配了 IP 地址中的最后一个数字。使用非捕获分组并把它重复 3 次可以使我们的正则表达式更简短、更高效。

要把 IP 地址中的文本表示转换为一个整数，我们需要分别捕获 4 个数字。解决方案中最后两个正则式完成的是这个任务。原来我们对一个分组重复 3 次，而现在则需要使用 4 个捕获分组，分别处理一个数字。只有用这种罗列的办法才能在 IP 地址中分别捕获所有 4 个数字。

一旦捕获到了这些数字，把它们组合为 32 位的整数就是很容易的。在 Perl 中，特殊变量 \$1、\$2、\$3 和 \$4 含有正则表达式中 4 个分组所匹配到的文本。实例 3.9 讲解了在其他编程语言中如何提取捕获分组的方法。在 Perl 中，对字符串应用逐位操作的左移位操作符 (<<)，就可以把捕获分组中的字符串变量自动强制转换为数字。在其他语言中，在用一个逐位操作 or 对数字进行移位和组合之前，你可能必须先调用 String.toInteger() 或者其他类似的手段。

## 参见

实例 2.3、实例 2.8、实例 2.9 和实例 2.12。

## 7.17 匹配 IPv6 地址

### 问题描述

你要检查一个字符串代表的是否是一个使用标准、紧凑和/或混合标记法来表示的 IPv6 地址。

### 解决方案

#### 标准标记法

匹配一个使用标准标记法的 IPv6 地址，标准的 IPv6 地址由 8 个使用十六进制表示的 16 位数组成，各数之间用冒号分隔（例如 1762:0:0:0:B03:1:AF18）。前导的 0 是可选的。

检查整个目标文本是否是一个使用标准标记法的 IPv6 地址：

```
^(?:(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4})$  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python  
  
\A(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}\Z  
正则选项：不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

在一大段文本集合中查找一个使用标准标记法的 IPv6 地址：

```
(?<![:.\w]) (?:(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4})(?![:.\w])  
正则选项：不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby 1.9
```

JavaScript 和 Ruby 1.8 并不支持逆序环视。我们必须去掉在正则式开头，用来保证不会找到位于一长串十六进制数和冒号序列中的 IPv6 地址的检查。现在我们使用单词边界来部分执行这个检查：

```
\b(?:[A-F0-9]{1,4}:\){7}[A-F0-9]{1,4}\b
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 混合标记

匹配一个使用混合标记法的 IPv6 地址。一个混合标记法的 IPv6 地址由 6 个十六进制的 16 位数和后面跟随的 4 个十进制字节数组成。各个字数之间用冒号分隔，而字节之间则用点号分隔。在字数和字节数之间用一个冒号作为分隔。在每个十六进制数和十进制字节中的前导 0 都是可选的。这种标记法会用于 IPv4 和 IPv6 地址混用的情况下。IPv6 地址是 IPv4 地址的扩展。762:0:0:0:B03:127.32.67.15 就是一个使用混合标记法的 IPv6 地址。

检查整个目标文本是否是一个使用混合标记法的 IPv6 地址：

```
^(?:[A-F0-9]{1,4}:\){6}(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.\.){3}^
(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)$
```

正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

在一大段文本中查找一个使用混合标记法的 IPv6 地址：

```
(?<![:.\w])(?:[A-F0-9]{1,4}:\){6}^
(?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.\.){3}^
(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?) (?![.:.\w])
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

JavaScript 和 Ruby 1.8 并不支持逆序环视。我们必须去掉在正则式开头，用来保证不会找到位于一长串十六进制数和冒号序列中的 IPv6 地址的检查。现在使用单词边界来部分执行这个检查：

```
\b(?:[A-F0-9]{1,4}:\){6}(?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.\.){3}^
(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\b
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 标准或混合标记法

匹配一个使用标准或混合标记法的 IPv6 地址。

检查整个目标文本是否是一个使用标准或混合标记法的 IPv6 地址：

```
\A
(?:[A-F0-9]{1,4}:\){6}# 字符串开始
(?:[A-F0-9]{1,4}:[A-F0-9]{1,4})# 6 个字数
| (?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.\.){3} # 或 4 个字节数
 (?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)# 2 个字数
)\Z# 字符串结束
```

正则选项：宽松排列、不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

```
^(?::[A-F0-9]{1,4}::){6}(?::[A-F0-9]{1,4}:[A-F0-9]{1,4}|←  
(?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.){3}|←  
(?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?) )$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

在一大段文本中找到一个使用标准或混合标记法的 IPv6 地址：

```
(?<![:.\w]) # 地址定位  
(?::[A-F0-9]{1,4}::){6} # 6 个字数  
(?::[A-F0-9]{1,4}:[A-F0-9]{1,4}) # 2 个字数  
| (?: (?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.){3} # 或 4 个字节数  
| (?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)  
) (?![.:.\w]) # 地址定位
```

正则选项：宽松排列、不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python 1.9

JavaScript 和 Ruby 1.8 并不支持逆序环视。我们必须去掉在正则式开头，用来保证不会找到位于一长串十六进制数和冒号序列中的 IPv6 地址的检查。现在使用单词边界来部分执行这个检查：

```
\b # 单词边界  
(?::[A-F0-9]{1,4}::){6} # 6 个字数  
(?::[A-F0-9]{1,4}:[A-F0-9]{1,4}) # 2 个字数  
| (?: (?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.){3} # 或 4 个字节数  
| (?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)  
) \b # 单词边界
```

正则选项：宽松排列、不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

```
\b(?:[A-F0-9]{1,4}::){6}(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}|←  
(?: (?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.){3}|←  
(?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?) )\b
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 压缩标记法

匹配一个使用压缩标记法的 IPv6 地址。压缩标记法和标准标记法大体相同，只是可以省略一个或者多个值为 0 的数字序列，而在略去 0 的数字序列的前后就会只剩下冒号。使用压缩标记法的地址中会出现两个连续的冒号，因此可以通过这个特征来进行识别。一个地址中只能省略一个值为 0 的数字序列，否则，我们就无法确定在每个序列中有多少个数字被省略掉了。如果省略的为 0 的数字出现在 IP 地址的开端或结尾，那么该地址就会以两个冒号作为开始或结束。如果所有的数都是 0，那么 IPv6 地址就会以两个冒号组成，其中不包含任何数字。

例如，1762::B03:1:AF18 是 1762:0:0:0:B03:1:AF18 的压缩形式。本节中的正则表达式会匹配使用压缩和标准标记法的 IPv6 地址。

检查整个目标文本是否是一个使用标准或压缩标记法的 IPv6 地址：

```
\A(?:  
    # 标准形式  
    (?:[A-F0-9]{1,4}::){7}[A-F0-9]{1,4}  
    # 压缩为最多包含 7 个冒号  
    |(?=(?:[A-F0-9]{0,4}::){0,7}[A-F0-9]{0,4}  
        \Z) # 和定位符  
    # 以及最多 1 对连续冒号  
    |([0-9A-F]{1,4}::){1,7}|::|(:[0-9A-F]{1,4}){1,7}|::  
)\Z  
正则选项：宽松排列、不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby  
  
^(?: (?:[A-F0-9]{1,4}::){7}[A-F0-9]{1,4}|  
    (?=(?:[A-F0-9]{0,4}::){0,7}[A-F0-9]{0,4}$)|([0-9A-F]{1,4}::){1,7}|::)  
    |(:[0-9A-F]{1,4}){1,7}|::)$  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python
```

在一大段文本中查找一个使用标准或压缩标记法的 IPv6 地址：

```
(?<![:.\w]) (?:  
    # 标准形式  
    (?:[A-F0-9]{1,4}::){7}[A-F0-9]{1,4}  
    # 压缩为最多包含 7 个冒号  
    |(?=(?:[A-F0-9]{0,4}::){0,7}[A-F0-9]{0,4}  
        (?![.:.\w])) # 和定位符  
    # 以及最多 1 对连续冒号  
    |([0-9A-F]{1,4}::){1,7}|::|(:[0-9A-F]{1,4}){1,7}|::  
) (?![.:.\w])  
正则选项：宽松排列、不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python 1.9
```

JavaScript 和 Ruby 1.8 并不支持逆序环视。我们必须去掉在正则式开头，用来保证不会找到位于一长串十六进制数和冒号序列中的 IPv6 地址的检查。但是我们也不能使用单词边界作为替代，因为地址可能会以冒号开始，而冒号不是一个单词字符：

```
(?:  
    # 标准形式  
    (?:[A-F0-9]{1,4}::){7}[A-F0-9]{1,4}  
    # 压缩为最多包含 7 个冒号  
    |(?=(?:[A-F0-9]{0,4}::){0,7}[A-F0-9]{0,4}  
        (?![.:.\w])) # 和定位符  
    # 以及最多 1 对连续冒号  
    |([0-9A-F]{1,4}::){1,7}|::|(:[0-9A-F]{1,4}){1,7}|::  
) (?![.:.\w])
```

正则选项：宽松排列、不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

```
(?:(?:[A-F0-9]{1,4}):(7)[A-F0-9]{1,4}|(?=(?:[A-F0-9]{0,4}):(0,7)←  
[A-F0-9]{0,4}(?![:.\w]))(([0-9A-F]{1,4}):(1,7)|:)((:[0-9A-F]{1,4})←  
(1,7)|:))(?![:.\w])
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 压缩的混合标记法

匹配一个使用压缩的混合标记法的 IPv6 地址。压缩的混合地址和混合地址大体相同，只是可以省略一个或者多个为 0 的字数，而在省略掉 0 的数字位置前后就会只剩下冒号。但是 4 个十进制字节数即使是 0 也必须被显式说明。识别使用压缩的混合标记法的地址的方法是：在地址第一部分中会出现两个连续的冒号，而在第二部分中会出现 3 个点号。一个地址中只能省略掉一个值为 0 的数字序列，否则的话，我们就无法确定每个序列中有多少个数字被省略了。如果省略的为 0 的数字出现在 IP 地址的开端或结尾，那么该地址就会以两个冒号作为开始或结束。

例如，IPv6 地址 1762::B03:127.32.67.15 就是 1762:0:0:0:B03:127.32.67.15 的压缩形式。本节中的正则表达式会匹配使用压缩和或非压缩混合标记法的混合 IPv6 地址。

检查整个目标文本是否是一个使用压缩或非压缩混合标记法的 IPv6 地址的正则表达式：

```
\A  
(?:  
    # 没有压缩的形式  
    (?:[A-F0-9]{1,4}):(6)  
    # 被压缩为至少 6 个冒号  
    | (?=(?:[A-F0-9]{0,4}):(0,6)  
        (?:[0-9]{1,3}\.){3}[0-9]{1,3}          # 和 4 个字节数  
        \Z)                                    # 定位符  
    # 以及最多一对连续冒号  
    (([0-9A-F]{1,4}):(0,5)|:)((:[0-9A-F]{1,4})(1,5):|:)  
)  
# 255.255.255.  
(?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.){3}  
# 255  
(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)  
\Z  
正则选项：宽松排列、不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby  
^(?:(?:[A-F0-9]{1,4}):(6)|(?=(?:[A-F0-9]{0,4}):(0,6)(?:[0-9]{1,3}\.){3}[0-9]{1,3}$)(([0-9A-F]{1,4}):(0,5)|:)((:[0-9A-F]{1,4})(1,5):|:))|  
(?:(?:25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4]  
[0-9]| [01]?[0-9][0-9]?)$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

在一大段文本中查找一个使用压缩或非压缩的混合标记法的 IPv6 地址：

```
(?<![:.\w])
(?:  
# 没有压缩的形式
(?:[A-F0-9]{1,4}::){6}
# 压缩为包含至少 6 个冒号
|(?=(?:[A-F0-9]{0,4}::){0,6}
 (?:[0-9]{1,3}\.){3}[0-9]{1,3} # 和 4 个字节数
  (?![.:.\w])) # 和定位符
# 以及最多 1 对连续冒号
((:[0-9A-F]{1,4}::){0,5}::)((:[0-9A-F]{1,4}::){1,5}::)
)
# 255.255.255.
(?:(:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.{3}
# 255
(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?
(?![.:.\w])
```

正则选项：宽松排列、不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby 1.9

JavaScript 和 Ruby 1.8 并不支持逆序环视。我们必须去掉在正则式开头，用来保证不会找到位于一长串十六进制数和冒号序列中的 IPv6 地址的检查。但是我们也不能使用单词边界，因为地址可能会以冒号开始，而冒号不是一个单词字符：

```
(?:  
# 没有压缩的形式
(?:[A-F0-9]{1,4}::){6}
# 被压缩为包含最多 6 个冒号
|(?=(?:[A-F0-9]{0,4}::){0,6}
 (?:[0-9]{1,3}\.){3}[0-9]{1,3} # 和 4 个字节数
  (?![.:.\w])) # 和定位符
# 以及最多 1 对连续冒号
((:[0-9A-F]{1,4}::){0,5}::)((:[0-9A-F]{1,4}::){1,5}::)
)
# 255.255.255.
(?:(:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.{3}
# 255
(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?
(?![.:.\w])
```

正则选项：宽松排列、不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

```
(?:(:[A-F0-9]{1,4}::){6}|(?=(?:[A-F0-9]{0,4}::){0,6}(?:[0-9]{1,3}\.){3}+  
[0-9]{1,3}(?![:.\w]))((:[0-9A-F]{1,4}::){0,5}::)((:[0-9A-F]{1,4}::){1,5}::)+  
(?:(:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.{3}+  
(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)(?![.:.\w])
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 标准、混合或压缩标记法

匹配一个使用任意一种前文解释过的标记法的 IPv6 地址，其中可以是标准、混合、压缩或压缩混合标记法。

检查整个目标文本是否是一个 IPv6 地址：

```
\A(?:  
    # 混合形式  
    (?:  
        # 没有压缩的形式  
        (?:[A-F0-9]{1,4}:){6}  
        # 被压缩为包含最多 6 个冒号  
        | (?:=(?:[A-F0-9]{0,4}:){0,6}  
            (?:[0-9]{1,3}\.){3}[0-9]{1,3}      # 和 4 个字节数  
            \Z)                                # 和定位符  
        # 以及最多 1 对连续冒号  
        (([0-9A-F]{1,4}:){0,5}|::)(:[0-9A-F]{1,4}){1,5}:|:  
    )  
    # 255.255.255.  
    (?: (?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.){3}  
    # 255  
    (?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?  
    | # 标准形式  
        (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}  
    | # 压缩为最多包含 7 个冒号  
        (?:=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}  
            \Z) # 和定位符  
    # 以及最多 1 对连续冒号  
    (([0-9A-F]{1,4}:){1,7}|::)(:[0-9A-F]{1,4}){1,7}:|:  
)\Z  
正则选项：宽松排列、不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

^ (?: (?: (?:[A-F0-9]{1,4}:){6} | (?:=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.){3} |  
[0-9]{1,3}\\$) (([0-9A-F]{1,4}:){0,5}|::)(:[0-9A-F]{1,4}){1,5}:|::) |  
(?: (?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?\.){3} |  
(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]? | (?:[A-F0-9]{1,4}:){7} |  
[A-F0-9]{1,4} | (?:=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}\\$) |  
(([0-9A-F]{1,4}:){1,7}|::)(:[0-9A-F]{1,4}){1,7}:|::)\\$

正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

在一大段文本中查找一个标准、混合、压缩或压缩混合标记法的 IPv6 地址：

```
(?<![:.\w]) (?:  
    # 混合形式
```

```

(?:          # 没有压缩的形式
 (?:[A-F0-9]{1,4}::){6}
  # 被压缩为包含最多 6 个冒号
| (?=(?:[A-F0-9]{0,4}::){0,6}
  (?:[0-9]{1,3}\.){3}[0-9]{1,3}      # 和 4 个字节数
  (?![:.\w]))                      # 和定位符
  # 以及最多 1 对连续冒号
  (([0-9A-F]{1,4}::){0,5}::)((:[0-9A-F]{1,4}){1,5}::)
)
# 255.255.255.
(?:(:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.{3}
# 255
(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?
| # 标准形式
  (?:[A-F0-9]{1,4}::){7}[A-F0-9]{1,4}
| # 压缩为最多包含 7 个冒号
  (?=(?:[A-F0-9]{0,4}::){0,7}[A-F0-9]{0,4}
  (?![:.\w])) # 和定位符
  # 以及最多 1 对连续冒号
  (([0-9A-F]{1,4}::){1,7}::)((:[0-9A-F]{1,4}){1,7}::)
) (?![:.\w])
  正则选项: 宽松排列、不区分大小写
  正则流派: .NET、Java、PCRE、Perl、Python、Ruby 1.9

```

JavaScript 和 Ruby 1.8 并不支持逆序环视。我们必须去掉在正则式开头，用来保证不会找到位于一长串十六进制数和冒号序列中的 IPv6 地址的检查。但是我们也不能使用单词边界，因为地址可能会以冒号开始，而冒号不是一个单词字符：

```

(?:          # 混合形式
(?:          # 没有压缩的形式
 (?:[A-F0-9]{1,4}::){6}
  # 被压缩为包含最多 6 个冒号
| (?=(?:[A-F0-9]{0,4}::){0,6}
  (?:[0-9]{1,3}\.){3}[0-9]{1,3}      # 和 4 个字节数
  (?![:.\w]))                      # 和定位符
  # 以及最多 1 对连续冒号
  (([0-9A-F]{1,4}::){0,5}::)((:[0-9A-F]{1,4}){1,5}::)
)
# 255.255.255.
(?:(:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.{3}
# 255
(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?
| # 标准形式
  (?:[A-F0-9]{1,4}::){7}[A-F0-9]{1,4}
| # 压缩为最多包含 7 个冒号
  (?=(?:[A-F0-9]{0,4}::){0,7}[A-F0-9]{0,4}

```

```

(?![:.\w])) # 和定位符
# 以及最多 1 对连续冒号
(([0-9A-F]{1,4}):\{1,7}\|:)((:[0-9A-F]{1,4})\{1,7}\|:)
) (?![:.\w])
正则选项: 宽松排列、不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

(?:(?:(:[A-F0-9]{1,4}:\{6}\|(?=(?:[A-F0-9]{0,4}:\{0,6}(?:[0-9]\{1,3}\.\{3}\{0-9}\{1,3}\(?![:.\w]))((:[0-9A-F]{1,4}:\{0,5}\|:)((:[0-9A-F]{1,4})\{1,5}\|:))\{3}(?:25[0-5]\|2[0-4]\{0-9]\|01]\?0-9]\{0-9]\?)\.\{3}(?:25[0-5]\|2[0-4]\{0-9]\|01]\?0-9]\{0-9]\?)\|(?=[A-F0-9]\{1,4}:\{7}\{A-F0-9]\{1,4}\|(?=(?:[A-F0-9]\{0,4}:\{0,7]\{A-F0-9]\{0,4}\(?![:.\w]))\{([0-9A-F]\{1,4}:\{1,7}\|:)((:[0-9A-F]\{1,4})\{1,7}\|:))\(?![.:.\w])
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```

## 讨论

由于 IPv6 地址拥有不同的标记法，所以匹配 IPv6 地址就不像匹配 IPv4 地址那么简单。你打算支持哪种标记法的 IPv6 地址会在很大程度上影响到正则表达式的复杂度。根本上来说，IPv6 总共包含两种标记法：标准和混合的。你可以决定是否只允许两种标记法的其中之一，还是二者都允许。这样我们就需要使用 3 组正则表达式。

标准标记法和混合标记法都包含一种省略 0 的压缩形式。如果允许压缩标记法，那么我们还需要再添加 3 组正则表达式。

根据你是想检查一个给定的字符串是一个有效的 IPv6 地址，还是要在一大段文本中查找一个 IP 地址，正则表达式也会稍微有点不同。为了对 IP 地址做合法性验证，我们需要使用在实例 2.5 中讲解的定位符。JavaScript 中需要使用 `\^` 和 `\$` 定位符，而 Ruby 中可以使用 `\A` 和 `\Z`。其他所有流派则可以支持这两种定位符。Ruby 也支持 `\^` 和 `\$`，但是会同时允许它们匹配在字符串中内嵌的换行符。因此只有当你已经知道在字符串汇总中不包含任何内嵌的换行符的时候，才能使用脱字符和美元符号。

如果要在一大段文本中查找 IPv6 地址，那么我们就需要使用否定型逆序环视 `\(?<![.:.\w])\`` 和否定型顺序环视 `\(?![.:.\w])\``，以保证地址之前和之后不会跟着一个单词字符（字母、数字或下划线），或者是跟着一个点号或冒号，这样就可以确保我们不会匹配到一大段数字和冒号序列中的一部分内容。实例 2.16 中讲解了如何使用逆序环视或者顺序环视。假如不能使用环视，那么可以使用单词边界来检查 IP 地址之前和之后不会紧跟着一个单词字符，但是只有当地址的第一个和最后一个字符确定是（十六进制）数字时，才可以使用单词边界。压缩标记法允许地址以冒号开始和结束。如果我们在冒号之前或之后放一个单词边界，就会要求必须存在一个紧邻的字母或数字，而这并不是我们想要的。实例 2.6 详细讲解了单词边界。

## 标准标记法

标准的 IPv6 标记法用正则表达式处理起来会非常容易。我们只需要匹配用 7 个冒号隔开的 8 个十六进制标记的数。`<[A-F0-9]{1,4}>` 匹配 1~4 个十六进制字符，它可以匹配一个带有可选的前缀 0 的 16bit 字数。在其中的字符类（实例 2.3）中只列出了大写字母。使用大小写无关匹配模式可以把小写字母也匹配进来。关于你的编程语言中如何设定匹配模式，请参考实例 3.4 中的讲解。

非捕获分组 `<(?:[A-F0-9]{1,4}):>{7}` 会匹配一个后面紧跟着一个字面上的冒号的十六进制数。后面的量词会把这个组重复 7 次。如实例 2.9 中的说明一样，正则式中的第一个冒号是正则式对非捕获分组语法的一部分，正则式中的第二个冒号是一个字面上的冒号。在一些非常特殊的情况下，冒号会被用作更大的正则记号中的一部分，这种时候冒号是元字符，而除了这些情况之外，冒号在正则表达式中一般都不是一个元字符。所以，我们并不需要使用反斜杠来转义正则表达式中字面上的冒号。虽然可以对它进行转义，但这只会使我们的正则表达式更加难以读懂。

## 混合标记法

用来匹配混合 IPv6 标记法的正则式由两个部分组成。`<(?:[A-F0-9]{1,4}):>{6}` 会匹配 6 个十六进制数字，每个数字之后都会跟一个字面上的点号，这与在标准 IPv6 标记法中使用的 7 个这样的数字序列的形式是相同的。

和标准标记法不同的是，最后一个十六进制数被替代为一个完整的 IPv4 地址。我们使用实例 7.16 中所示的“精确”正则式来匹配这个 IPv4 地址部分。

## 标准或混合标记法

允许使用标准和混合标记法需要一个稍微长一点的正则表达式。两种标记法的区别只在于 IPv6 地址的最后 32 位。标准标记法使用两个 16 位数，混合标记法使用 4 个十进制字节，这正好是一个 IPv4 地址。

正则式的第一部分匹配 6 个十六进制数，这同只支持混合标记法的正则式中是一样的。正则式的第二部分现在是一个非捕获分组，对最后 32 位则使用了一个二选一的多选结构。在实例 2.8 中我们讲过，多选操作符（竖线）在所有正则式操作符中的优先级最低。所以，我们需要使用一个非捕获分组，这样就不会把前面的 6 个十六进制数也包含到多选结构中。

位于竖线的左边的第一个选择分支会匹配 2 个中间有冒号的十六进制数。第二个选择分支会匹配一个 IPv4 地址。

## 压缩标记法

如果我们允许使用压缩标记法的话，那么情况就会变得更为复杂。这是因为压缩标记

法中允许省略掉不定数量的 0。1:0:0:0:6:0:0, 1::6:0:0 和 1:0:0:0:0:6:: 是同一个 IPv6 地址的 3 种不同写法。一个地址中可能最多包含 8 个数字，但也可以一个数都没有。如果在地址中的数少于 8 个，那么必然有一对连续的冒号序列存在，用来表示被省略掉的 0。

在正则表达式中很容易实现可变次数的重复。如果一个 IPv6 地址有一对连续的冒号，则在这对冒号的前后至多只能有 7 个数字。我们可以很容易地把它写成：

```
(  
  ([0-9A-F]{1,4}::){1,7}    # 左边包含 1~7 个数字  
  | :                      # 或者以一对连续冒号作为开始  
 )  
(  
  ::[0-9A-F]{1,4}){1,7}    # 右边包含 1~7 个数字  
  | :                      # 或者以一对连续冒号作为结束  
)
```

正则选项：宽松排列、不区分大小写  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby



### 提示

如果去掉其中的注释和多余的空格，那么这个正则表达式和本讨论之后所给的正则表达式也可以用于 JavaScript。JavaScript 支持这些正则式中用到的除了宽松排列之外的所有特性，这里使用宽松排列是为了让正则式更加容易理解。

这个正则表达式匹配所有的压缩 IPv6 地址，但它并不会匹配任何使用非压缩的标准标记法的地址。

这个正则式非常简单。它的第一个部分匹配了 1~7 个后面跟随着冒号的数字，或者如果地址中在一对冒号的左边没有任何数字的话，只匹配一个冒号。第二部分会匹配 1~7 个前面有冒号的数字，或者如果地址中在一对冒号右边不包含任何数字的话，只匹配一个冒号。把它们放在一起，它能匹配到的就是一对冒号本身、一对冒号以及只在其左侧包含 1~7 个数、一对冒号以及只在其右侧包含 1~7 个冒号、还有一对冒号以及在其左右两侧分别包含 1 到 7 个数。

上面这最后一部分会有问题。这个正则式允许一对冒号在左右两侧各包含 1~7 个数，但是它并没有指定左右两侧数字的总数加起来必须要少于 7 个。一个 IPv6 地址中可以包含 8 个数字。但是因为一对冒号意味着至少有一个数字被省略掉了，所以最多只剩下 7 个。

正则表达式不会做算术。它们能统计某个东西是否出现了 1~7 次。但是它们无法统计两样东西加起来是否一共存在了 7 次，而且可以把这 7 次按任意组合划分到两样东西中。

为了更好地理解这个问题，我们可以先来看一个简单的类比。假设我们要匹配以

`aaaaxbbb` 形式出现的字符串。字符串的长度必须在 1~8 个字符之间，其中可以包含 0~7 个 a，恰好一个 x，然后是 0~7 个 b。

使用正则表达式来解决这个问题可以有两种方法。一种方法是罗列出所有可能的选择。下面一个小节会讨论压缩混合标记法，其中就使用了这种方法。这样会得到一个冗长但是很容易理解的正则式。

```
\A(?:a{7}x
| a{6}xb?
| a{5}xb{0,2}
| a{4}xb{0,3}
| a{3}xb{0,4}
| a{2}xb{0,5}
| axb{0,6}
| xb{0,7}
)\Z
```

正则选项：宽松排列  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

在这个正则式中，每个可能的字母 a 的数量都对应一个选择分支。每个选择分支会罗列出有多少字母 b 可以跟随在已经匹配了的特定数量的字母 a 和字母 x 之后。

另外一种解决方案是使用顺序环视。在“解决方案”小节中，用来匹配使用压缩标记法的 IPv6 地址的正则式中使用的就是这种方法。如果你不熟悉顺序环视，那么可以参考实例 2.16。使用顺序环视，我们就可以匹配同样一段文本两次，这样就可以检查 2 个条件。

```
\A
(?:[abx]{1,8})\Z
a{0,7}xb{0,7}
\Z
```

正则选项：宽松排列  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

在这个正则式开端的 `\A` 将它定位到目标文本的开端。然后就会使用一个肯定型的顺序环视。它会检查是否可以匹配到 1~8 个由字母 `\a`、`\b` 和/或 `\x` 组成的序列，而且当匹配到这 1~8 个字母之后，就到达了字符串的结尾。在顺序环视内的 `\Z` 是至关重要的。为了把正则式匹配的字符串限制在 8 个或更少的字符，这个顺序环视必须检查在它已经匹配到的字符之后不会再跟着更多字符。

在另外一种情况下，你可能要使用另外一种分界符来代替 `\A` 和 `\Z`。如果要对 `aaaaxbbb` 和类似的字符串来执行“整字匹配”的查找，那么你就需要使用某种分隔符，并且必须把匹配字符串结尾的分隔符放在顺序环视内和正则表达式结尾处。如果不这样做，那么正则表达式会部分匹配一个含有太多字符的字符串。

当满足了顺序环视的要求后，顺序环视就会放弃已经匹配到的数据。这样，当正则式

引擎企图匹配 `a{0,7}` 时，就会回退到字符串的开始。顺序环视并不会消耗它所匹配到的文本，这正是顺序环视和非捕获分组之间的一个关键区别，而这就会允许我们对一个文本片段应用两个模式。

尽管 `a{0,7}xb{0,7}` 自身可以匹配至多 15 个字母，在这个例子中它只能匹配到 8 个字母，这是因为顺序环视已经保证了只会包含 8 个字母。`a{0,7}xb{0,7}` 需要做的所有事情只是检查一下它们是否会以正确的顺序出现。实际上，`a*xb*` 在这个正则表达式中会同 `a{0,7}xb{0,7}` 有完全一样的效果。

正则式结尾的第二个 `\Z` 同样也是至关重要的。正如顺序前视需要确保其中不会包含太多字母一样，顺序环视之后的第二个测试需要确保所有的字母出现的顺序是正确的。这样就可以确保我们不会匹配到诸如 `axba` 这样的字符串，尽管它的长度在 1~8 字符之间，可以满足顺序环视的要求。

## 压缩的混合标记法

混合标记法可以和标准标记法一样进行压缩。虽然结尾的 4 个字节数哪怕是零也必须被指定，在它们之前的十六进制数的个数还是可变的。如果所有的十六进制数字都是零，那么这个 IPv6 地址结果看起来会像是个前面带了两个冒号的 IPv4 地址。

为压缩的混合标记法创建一个正则式会涉及压缩的标准标记法要解决的同样问题。上一个小节详细解释了这些问题。

用于压缩的混合标记法的正则式和用于压缩的（标准的）标记法的正则式之间的主要区别在于，用于压缩的混合标记法的正则式中需要检查 6 个十六进制数后的 IPv4 地址。我们会在正则式的结尾处做这个检查，所使用的正则式与我们在实例 7.16 中用来精确匹配非压缩混合标记法的 IPv6 地址中的 IPv4 地址的正则式是一样的。

我们必须在正则式的结尾匹配一个 IPv4 部分，但也必须在顺序环视内检查 IPv4 地址部分，以确保不会在 IPv6 地址中匹配超过 6 个冒号或 6 个十六进制数字。因为我们已经在正则式结尾做了一个精确的检查，因此顺序环视中只用一个简单的 IPv4 检查就足够了。该顺序环视不需要对 IPv4 部分做合法性验证，因为正则式的主体部分已经保证了这一点。但是它还是必须匹配 IPv4 地址部分，从而在顺序环视结尾处的字符串结尾定位符才能正确产生匹配。

## 标准、混合或压缩标记法

最后一个正则表达式集合把前面讲到的所有内容都放到了一起。这样就可以匹配到使用所有标记法的 IPv6 地址：标准的或混合的，压缩的或不压缩的。

在这些正则表达式中，会把用于压缩混合标记法和压缩（标准）标记法的两个正则式

使用多选结构组合起来。这些正则式已经使用了多选结构来匹配它们所支持的压缩和非压缩的不同的 IPv6 标记法。

这样做所得到的正则表达式中会包含 3 个顶层的选择分支，其中第一个选择分支自身是由 2 个选择分支组成的。第一个选择分支会匹配使用混合标记的 IPv6 地址，其中包括压缩的和非压缩的形式。第二个选择分支匹配使用标准标记法的 IPv6 地址。第三个选择分支会匹配压缩的（标准）标记法。

在这里我们使用了 3 个顶层的选择分支，而不是使用各自含有 2 个子选择分支的 2 个选择分支，原因是没有什么特别理由把标准和压缩标记法的选择分支放在一起。对于混合标记法，我们确实要把压缩和非压缩的选择分支放在一起，因为它让我们可以避免罗列出 IPv4 部分 2 次。

根本上来讲，我们把下面这个正则式：

```
^(6words|compressed6words) ip4$
```

和这个正则式：

```
^(8words|compressed8words)$
```

组合成为：

```
^((6words|compressed6words) ip4|8words|compressed8words)$
```

而不是：

```
^((6words|compressed6words) ip4| (8words|compressed8words))$
```

## 参见

实例 2.16 和实例 7.16。

## 7.18 Windows 路径的合法性验证

### 问题描述

你想要检查一个字符串看起来是不是像 Microsoft Windows 操作系统上的文件夹或文件的一个合法路径。

### 解决方案

#### 盘符路径

```
\A  
[a-z]:\\  
(?:[^\\/:*?"<>|\r\n]+\\)*      # 盘符  
                                # 文件夹
```

```
[^\//:*?">|\r\n]* # 文件
\Z
正则选项: 宽松排列、不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^ [a-z]:\\(?:[^\\/:*?">|\r\n]+\\)* [^\//:*?">|\r\n]*$ # 盘符
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python
```

## 盘符和 UNC 路径

```
\A
(?:[a-z]:\\\\\\[a-z0-9_.]+\\[a-z0-9_.]+)\\ # 盘符
(?:[^\\/:*?">|\r\n]+\\)* # 文件夹
[^\//:*?">|\r\n]* # 文件
\Z
正则选项: 宽松排列、不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^(?:[a-z]:\\\\\\[a-z0-9_.]+\\)(?:[^\\/:*?">|\r\n]+\\)*# 盘符
[^\//:*?">|\r\n]*$ # 相对 URL
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python
```

## 盘符、UNC 和相对路径

```
\A
(?:(?:[a-z]:\\\\\\[a-z0-9_.]+\\[a-z0-9_.]+)\\| # 盘符
  \\\\?[^\\/:*?">|\r\n]+\\?) # 相对 URL
(?:[^\\/:*?">|\r\n]+\\)* # 文件夹
[^\//:*?">|\r\n]* # 文件
\Z
正则选项: 宽松排列、不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^(?:(?:[a-z]:\\\\\\[a-z0-9_.]+\\[a-z0-9_.]+)\\|\\\\\?[^\\/:*?">|\r\n]+\\?)# 盘符
(?:[^\\/:*?">|\r\n]+\\)*[^\\/:*?">|\r\n]*$ # 相对 URL
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python
```

## 讨论

### 盘符路径

匹配包含盘符的驱动器上的文件或文件夹的全路径是非常简单的。驱动器是用单个字母表示的，之后会跟着一个冒号和反斜杠。我们可以使用 `<[a-z]:\\>` 很容易地匹配这个部分。反斜杠在正则表达式中是一个元字符，所以我们需要使用另外一个反斜杠对它进行转义，才能对它进行字面上的匹配。

Windows 上的文件夹和文件名中可以包含除了 `\V:*?"<>|` 之外的所有字符。另外也不允许出现换行符。我们可以很容易地使用一个否定字符类 `\[^V:*?"<>|\r\n]+` 来匹配这样的一个字符序列。反斜杠在字符类中也是一个元字符，所以还需要对它进行转义。`\r` 和 `\n` 是 2 个换行字符。关于（否定）字符类的更多信息，请参考实例 2.3。后面的加号量词（实例 2.12）则指定了我们想要匹配一个或更多这些字符。

文件夹之间会使用反斜杠来进行分隔。我们可以用 `\((?:[^V:*?"<>|\r\n]+\\\\)*\)` 来匹配 0 个或多个文件夹，其中把匹配文件夹名称和一个字面的反斜杠的正则式放到了一个非捕获分组（实例 2.9）中，该分组会使用星号量词（实例 2.12）匹配 0 次或多次。

要匹配文件名的话，我们可以使用 `\[^V:*?"<>|\r\n]*`。星号意味着文件名是可以省略的，因此我们也可以匹配以反斜杠结尾的路径。如果你不想匹配以反斜杠结尾的路径，那么可以把最后一个正则式中的 `*` 改成 `+`。

## 盘符和 UNC 路径

网络驱动器的文件可以不被映射到盘符之上，访问它的路径可以使用统一命名规范（Universal Naming Convention，UNC）路径来访问。UNC 路径的形式是 `\server\share\folder\file`。

我们可以很容易地修改盘符路径的正则表达式来使之也支持 UNC 路径。我们所要做的事情只是把用来匹配盘符的 `[a-z]:` 部分替换为用来匹配一个盘符或服务器名称的正则式。

`\((?:[a-z]:|\\\\\\[a-z0-9_.]+\\\\[a-z0-9_.]+)\)` 就可以满足这个目的。其中的竖线是多选结构操作符（实例 2.8）。它允许我们由 `[a-z]:` 匹配的盘符或者由 `\\\\\\[a-z0-9_.]+\\\\[a-z0-9_.]+` 匹配的服务器和共享名称之间进行选择。

多选操作符在所有正则操作符中的优先级最低。要想把两个选择分支组合到一起，我们就需要使用非捕获分组。在实例 2.9 中讲过，字符序列 `\((?:)` 会被当作一个非捕获分组的稍微有些复杂的起始括号。位于圆括号之后的问号失去了它通常的含义。

正则表达式中余下的部分还可以保持不变。在 UNC 路径中的共享名称可以通过匹配文件夹名称的正则式部分来匹配。

## 盘符、UNC 和相对路径

相对路径是以文件夹名称作为开始的路径（也有可能是使用特殊文件夹来选择父文件夹），或者是只包含一个文件名的路径。为了支持相对路径，我们在正则式中的“驱动器”部分添加了第 3 个选择分支。这个选择分支匹配的不是盘符或者服务器名称，而是会匹配一个相对路径的开始。

`\?[^\\:*?">|\\n]+\\?` 会匹配相对路径的开始部分。路径可以用反斜杠作为开始，但是它并不一定非要是这样。`\?` 会匹配一个反斜杠，或者什么也不匹配。`[^\\:*?">|\\n]+` 会匹配一个文件夹或者文件名。如果相对路径中只包含一个文件名，那么最后的 `\?` 不会匹配到任何内容，而且这个正则式中的“文件夹”和“文件”部分也不会匹配到任何内容，因为这二者也都是可以省略的。如果相对路径中指定了文件夹，那么最后的 `\?` 会匹配用来分隔相对路径中第一个文件夹和路径中其余部分的反斜杠。“文件夹”部分则会匹配文件名。

用来匹配相对路径的正则表达式无法再使用正则式中的独立部分来恰好匹配目标文本中的独立部分。如果路径是相对，那么被标记为“相对路径”的正则式部分实际上匹配的是一个文件夹或文件名。而如果相对路径中指定了一个或多个路径，那么“相对路径”部分会匹配第一个路径，而“文件夹”和“文件”路径会匹配剩余的部分。如果相对路径只是一个文件名，那么它会被“相对路径”部分匹配，从而“文件夹”和“文件”部分就无法匹配到任何内容。因为我们只对检查路径的合法性感兴趣，所以这样做并不会产生任何问题。正则式中包含的注释只是为了更加容易理解而添加的标签。

如果我们想要把路径中的不同部分提取到捕获分组中，那么我们就不得不更加谨慎地分别匹配驱动器、文件夹和文件名。下一个实例会用来讲解这个问题。

## 参见

实例 2.3、实例 2.8、实例 2.9 和实例 2.12。

## 7.19 分解 Windows 路径

### 问题描述

你想要检查一个字符串看起来是否像是指向 Microsoft Windows 操作系统上的文件夹或文件的一个合法路径。如果该字符串中保存的是一个合法的 Windows 路径，那么你还想要分别抽取其中的驱动器、文件夹和文件名部分。

### 解决方案

#### 盘符路径

```
\A
(?<drive>[a-z]:)\\
(?<folder>(?:[^\\/:*?">|\\r\\n]+\\)*)
(?<file>[^\\/:*?">|\\r\\n]*)
\Z
```

正则选项：宽松排列、不区分大小写

正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9

```
\A
(?P<drive>[a-z]:)\\
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

正则选项: 宽松排列、不区分大小写

正则流派: PCRE 4 及更高版本、Perl 5.10、Python

```
\A
([a-z]:)\\
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

正则选项: 宽松排列、不区分大小写

正则流派: .NET、Java、PCRE、Perl、Python、Ruby

```
^([a-z]:)\\((?:[^\\/:*?"<>|\r\n]+\\)*)([^\\/:*?"<>|\r\n]*)$
```

正则选项: 不区分大小写

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python

## 盘符和 UNC 路径

```
\A
(?<drive>[a-z]:\\\\\[a-z0-9_.\$]+\\\[a-z0-9_.\$]+)\\\
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

正则选项: 宽松排列、不区分大小写

正则流派: .NET、PCRE 7、Perl 5.10、Ruby 1.9

```
\A
(?P<drive>[a-z]:\\\\\[a-z0-9_.\$]+\\\[a-z0-9_.\$]+)\\\
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

正则选项: 宽松排列、不区分大小写

正则流派: PCRE 4 及更高版本、Perl 5.10、Python

```
\A
([a-z]:\\\\\[a-z0-9_.\$]+\\\[a-z0-9_.\$]+)\\\
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

正则选项: 宽松排列、不区分大小写

正则流派: .NET、Java、PCRE、Perl、Python、Ruby

```
^([a-z]:\\\\\[a-z0-9_.\$]+\\\[a-z0-9_.\$]+)\\\\((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*$
```

正则选项: 不区分大小写

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python

## 盘符、UNC 和相对路径



### 警告

这些正则表达式可以匹配空字符串。更多细节请参考后面的“讨论”小节，其中还会介绍一种可替代的解决方案。

```
\A
(?<drive>[a-z]:\\|\\\\\[a-z0-9_.\$]+\\\[a-z0-9_.\$]+\\|\?\)
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

正则选项：宽松排列、不区分大小写

正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

```
\A
(?P<drive>[a-z]:\\|\\\\\[a-z0-9_.\$]+\\\[a-z0-9_.\$]+\\|\?\)
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

正则选项：宽松排列、不区分大小写

正则流派：PCRE 4 及更高版本、Perl 5.10、Python

```
\A
([a-z]:\\|\\\\\[a-z0-9_.\$]+\\\[a-z0-9_.\$]+\\|\?\)
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\/:*?"<>|\r\n]*)
\Z
```

正则选项：宽松排列、不区分大小写

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

```
^([a-z]:\\|\\\\\[a-z0-9_.\$]+\\\[a-z0-9_.\$]+\\|\?\)↵
((?:[^\\/:*?"<>|\r\n]+\\)*)([^\/:*?"<>|\r\n]*)$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python

## 讨论

在本实例中的正则表达式与上一个实例中的正则式非常类似。下面的讨论会假设你已经阅读并且理解了上一个实例中的讨论部分。

## 盘符路径

与前一个实例相比，在匹配盘符路径的正则表达式中只做了一处改动。我们添加了3个捕获分组，从而可以用它们来提取路径中的不同组成部分：`<drive>`、`<folder>`和`<file>`。如果你的正则流派支持命名捕获（实例 2.11），那么就可以使用这些名称。否则，你就只好对这些捕获分组使用它们的编号进行引用，也就是1、2和3。要想了解在你喜欢的编程语言中如何获得命名和/或编号的分组所匹配到的文本，请

参考实例 3.9。

## 盘符和 UNC 路径

在匹配 UNC 路径的正则表达式中，我们添加了相同的 3 个捕获分组。

## 盘符、UNC 和相对路径

如果我们还想支持相对路径，那么机会变得相对更加复杂。在上一个实例中，我们只需要在正则式的驱动器部分添加第 3 个选择分支，匹配相对路径的开始部分。但在这里我们却不能这样做。因为对于相对路径来说，用于驱动器的捕获分组应该是空的。

这里的解决方案中，在“盘符和 UNC 路径”小节内的正则式中驱动器使用的捕获分组之后的字面上的反斜杠被移动到了捕获分组中。我们把它添加到了盘符选择分支和网络共享的结尾。对于可能会以反斜杠开始的相对路径，我们还添加了第 3 个选择分支来匹配一个可选的反斜杠。因为第 3 个选择分支是可选的，所以用于驱动器的整个分组实质上也就是可选的。

这样得到的正则表达式可以正确地匹配所有的 Windows 路径。然而问题是因为把驱动器部分变为了可选，我们现在就得到一个任何内容都是可选的一个正则式。在只支持绝对路径的正则表达式中，文件夹和文件部分已经是可选了。换句话说：我们的正则表达式会匹配空字符串。

如果想要确保这个正则式不会匹配到空字符串，那么我们就不得不再添加额外的选择分支来处理指定了文件夹的相对路径（此时文件名是可选的），以及没有指定文件夹的相对路径（此时文件名是必需的）：

```
\A
(?:(
    (?<drive>[a-z]:|\\|[a-zA-Z0-9_.]+\\|[a-zA-Z0-9_.]+)\\\
    (?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
    (?<file>[^\\/:*?"<>|\r\n]*)
    | (?<relativefolder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
        (?<file2>[^\\/:*?"<>|\r\n]*)
    | (?<relativefile>[^\\/:*?"<>|\r\n]+)
)
\Z
```

正则选项：宽松排列、不区分大小写  
正则流派：.NET、PCRE 7、Perl 5.10、Ruby 1.9

```
\A
(?:(
    (?P<drive>[a-z]:|\\|[a-zA-Z0-9_.]+\\|[a-zA-Z0-9_.]+)\\\
    (?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
    (?P<file>[^\\/:*?"<>|\r\n]*)
```

```

| (?P<relativefolder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
| (?P<file2>[^\\/:*?"<>|\r\n]*)
| (?P<relativefile>[^\\/:*?"<>|\r\n]+)
|
\Z
    正则选项: 宽松排列、不区分大小写
    正则流派: PCRE 4 及更高版本、Perl 5.10、Python

\A
(?::
  ([a-z]:\\\\\\[a-z0-9_.\$]+\\[a-z0-9_.\$]+)\\\
  ((?:[^\\/:*?"<>|\r\n]+\\)*)
  ([^\\/:*?"<>|\r\n]*)
| (\\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  ([^\\/:*?"<>|\r\n]*)
| ([^\\/:*?"<>|\r\n]+)
)
\Z
    正则选项: 宽松排列、不区分大小写
    正则流派: .NET、Java、PCRE、Perl、Python、Ruby

^(?:([a-z]:\\\\\\[a-z0-9_.\$]+\\[a-z0-9_.\$]+)\\((?:[^\\/:*?"<>|\r\n]+\\)*
*)\$\n
  ([^\\/:*?"<>|\r\n]*))|(\\\?(?:[^\\/:*?"<>|\r\n]+\\)+)([^\\/:*?"<>|\r\n]*
)|\$\n
  ([^\\/:*?"<>|\r\n]+)\$\n
    正则选项: 不区分大小写
    正则流派: .NET、Java、JavaScript、PCRE、Perl、Python

```

为了排除长度为 0 的字符串，我们付出的代价是现在要使用 6 个捕获分组来匹配该路径的 3 种不同组成部分。现在有必要检查一下你使用这个正则表达式的具体情形，来判断是在使用该正则式之前额外检查空字符串容易，还是多花点儿时间在一个匹配找到之后处理多个捕获分组更容易。

如果使用的是.NET 正则流派，那么你可以赋给多个命名分组同样的名称。.NET 是唯一把多个相同命名的分组当作一个捕获分组来使用的正则流派。有了.NET 中的这个正则式，你就可以获得文件夹或文件分组的匹配，而不用担心是这 2 个文件夹分组或 3 个文件分组中的哪一个实际上参与了正则匹配：

```

\A
(?::
  (?<drive>[a-z]:\\\\\\[a-z0-9_.\$]+\\[a-z0-9_.\$]+)\\\
  (?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<folder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<file>[^\\/:*?"<>|\r\n]+)
)
\Z

```

正则选项：宽松排列、不区分大小写

正则流派：.NET

## 参见

实例 2.9、实例 2.11、实例 3.9 和实例 7.18。

## 7.20 从 Windows 路径中抽取盘符

### 问题描述

你有一个字符串，其中包含了一个指向 Windows PC 或网络上的文件或文件夹的（语法上）合法的路径。你想要从路径中抽取其中的盘符（如果存在）。例如，你想要从路径 c:\folder\file.ext 中抽取盘符 c。

### 解决方案

`^([a-z]):`

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

从已知包含合法路径的字符串中抽取盘符是很简单的，即使你不知道该路径实际上是否会以盘符作为开始。路径也可以是一个相对路径或者 UNC 路径。

在 Windows 路径中，冒号只能用来作为盘符的分隔符，否则就是不合法的字符。因此，如果在字符串的开始包含一个字母后面跟着冒号，那么我们就可以确定该字母是一个盘符。

定位符 `\^` 会匹配字符串的开始（实例 2.5）。在 Ruby 中，虽然脱字符同样可以匹配内嵌的换行，但是这也不会带来问题，因为在合法的 Windows 路径中不会包含换行。字符类 `[a-z]` 会被用来匹配单个字母（实例 2.3）。我们把字符类放到一对圆括号之间（这样会构成一个捕获分组），因此你可以只得到盘符，而不会把正则表达式也匹配到的字面冒号包含进来。我们把冒号添加到正则表达式是为了确保我们抽取的是盘符，而不是某个相对路径中的第一个字母。

## 参见

实例 2.9 会告诉你关于捕获分组的一切。

要了解如何在你喜欢的编程语言中获取捕获分组匹配到的文本，请参考实例 3.9。

如果你事先不能确定字符串中是否包含了合法的 Windows 路径，那么需要先按照实

例 7.19 进行判断。

## 7.21 从 UNC 路径中抽取服务器和共享名

### 问题描述

你有一个字符串，其中包含了一个指向 Windows PC 或网络上的文件或文件夹的（语法上）合法的路径。如果该路径是一个 UNC 路径，那么你想要从其中抽取网络服务器的名称和该路径在服务器上指向的共享名。例如，你想要从路径\\server\\share\\folder\\file.ext 中抽出 server 和 share。

### 解决方案

```
^\\\\(([a-zA-Z0-9_\.\$]+)\\\\([a-zA-Z0-9_\.\$]+)
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

从已知包含合法路径的字符串中抽取网络服务器和共享名称是很简单的，即使你不知道该路径实际上是一个 UNC 路径。路径也可以是一个相对路径或者使用盘符。

UNC 路径会以两个反斜杠作为开始。在 Windows 路径中，两个连续的反斜杠只能用来作为 UNC 路径的开始，否则一定是不允许的。因此，如果一个已知是合法的路径以两个反斜杠作为开始，那么我们就可以确定之后跟着的是服务器和共享名称。

定位符 `<^>` 会匹配字符串的开始（实例 2.5）。在 Ruby 中，虽然脱字符同样可以匹配内嵌的换行，但是这也不会带来问题，因为在合法的 Windows 路径中不会包含换行。`\\\\` 会匹配两个字面的反斜杠。因为在正则表达式中反斜杠是元字符，因此如果我们想要把它作为字面字符来匹配，就必须使用另外一个反斜杠对它进行转义。其中的第一个字符类 `[a-zA-Z0-9_\.\$]+` 会匹配网络服务器的名称。而在另外一个字面反斜杠之后的第二个字符类则会匹配共享名称。我们把两个字符类分别放到一对圆括号之间，这样就会构成一个捕获分组。使用这种方式，你可以从第一个捕获分组中获得服务器名称，而在第二个捕获分组中获得共享名称。总的正则匹配将会是`\\server\\share`。

### 参见

实例 2.9 会告诉你关于捕获分组的一切。

要了解如何在你喜欢的编程语言中获取捕获分组匹配到的文本，请参考实例 3.9。

如果你事先不能确定字符串中是否包含了合法的 Windows 路径，那么需要先按照实例 7.19 进行判断。

## 7.22 从 Windows 路径中抽取文件夹

### 问题描述

你有一个字符串，其中包含了一个指向 Windows PC 或网络上的文件或文件夹的（语法上）合法的路径，你想要从路径中抽取其中的文件夹。例如，你想要从路径 c:\folder\subfolder\file.ext 或者 \\server\share\folder\subfolder\file.ext 中抽取出其中的 \folder\subfolder\。

### 解决方案

```
^([a-z]:\\|[a-z0-9_.]+\\[a-z0-9_.]+)?((?:\\|^)↵
(?:[^\\/:*?"<>|\r\n]+\\)+)
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### 讨论

如果我们想要支持 UNC 路径，那么从一个 Windows 路径中抽取文件夹就会有些复杂，因为我们无法简单从反斜杠之间抓取路径中的组成部分。如果这样做，就会同时抓取到 UNC 中的服务器和共享名称。

正则式的第一部分 `^([a-z]:\\|[a-z0-9_.]+\\[a-z0-9_.]+)?` 会略过位于路径开始的盘符或网络服务器与网络共享名称。这部分正则式中包含一个带 2 个选择分支的捕获分组。第一个选择分支会匹配盘符（参考实例 7.20），而第二个选择分支会匹配 UNC 路径中的服务器和共享名（参考实例 7.21）。实例 2.8 中讲解了多选结构操作符。

在该分组之后的问号会使这个分组变为可选的。这样就允许我们支持相对路径，因为其中既不包含盘符，也不包含网络共享。

使用 `(?:[^\\/:*?"<>|\r\n]+\\)+` 可以很容易地匹配到文件夹。其中的字符类会匹配一个文件夹名称。而非捕获分组会匹配一个文件夹名后面跟着一个字面的反斜杠，反斜杠会用来分隔两个文件夹，或者文件夹与文件名。我们会重复这个分组一次或多次。这意味着我们的正则表达式会只匹配实际上指定了文件夹的那些路径。只指定一个文件名、盘符或者网络共享的路径则不会产生匹配。

如果路径以盘符或网络共享作为开始，那么它之后必须跟一个反斜杠。一个相对路径则不一定会以反斜杠作为开始。因此，我们需要向用来匹配路径中文件夹部分的分组的开始处添加一个可选的反斜杠。因为我们只会使用这个正则表达式来匹配已知是合法的路径，所以就不必严格要求必须在有盘符或网络共享的情形下才能出现反斜杠。我们只需要允许它出现即可。

因为我们要求这个正则式至少要匹配一个文件夹，所以必须要确保我们的正则式不会把在 \\server\\share\\ 中的 e\\ 作为文件夹来匹配。这也就是我们为什么使用 <(\|^)>，而不是 <\^?> 来在用于文件夹的捕获分组开始添加可选的反斜杠。

如果你想知道为什么 \\server\\shar 可能会被匹配为驱动器，而 e\\ 被匹配为文件夹，那么请复习一下实例 2.13。正则表达式引擎会回溯。假设我们有下面这样一个正则表达式：

```
^([a-z]:|\\\\\\[a-z0-9_.]+\\\\[a-z0-9_.]+)+?  
((?:\\?(:[^\\/:*?"<>|\r\n]+\\))+)
```

与在解决方案中所给的正则式一样，这个正则式也要求在路径中至少包含一个非反斜杠的字符和一个反斜杠。如果这个正则式在 \\server\\share\\ 中匹配驱动器时先匹配到了 \\server\\share，然后在试图匹配文件夹分组时产生失败，但是此时它并不会选择放弃；它会接着尝试这个正则式的不同组合情形。

在这个例子中，正则引擎会记住用来匹配网络共享名的字符类 <[a-z0-9\_.]+>，并不一定非要匹配所有可用的字符。只需要一个字符就可以满足 <+>。因此，引擎会进行回溯，强迫该字符类放弃一个字符，然后试图继续执行匹配。

当引擎继续的时候，在目标字符串中就有两个剩余字符可以匹配文件夹：e\\。这两个字符足以满足正则式 <(:[^\\/:\*?"<>|\r\n]+\\)+>，因此我们就得到了这个正则式的一个整体匹配。但是这并不是我们想要的结果。

如果使用 <(\|^)> 来代替 <\^?> 就可以讲解这个问题。它依然会允许一个可选的反斜杠，但是当反斜杠缺失的时候，它会要求文件夹必须从字符串的开头开始。这也就意味着如果一个驱动器被匹配之后，并且因此正则引擎的执行已经跨越了字符串的开头，那么就必须要求有反斜杠。正则引擎如果不能匹配到任何文件夹，那么它还是会选择回溯，但是它这样做也是徒劳，因为 <(\|^)> 是无法产生匹配的。正则引擎会一直回溯，直到它又回到了字符串的开始。用来捕获盘符和网络共享的分组是可选的，所以正则引擎也允许从字符串的开始来尝试匹配文件夹。虽然 <(\|^)> 会在这里产生匹配，正则式的其余部分却无法匹配，这是因为 <(:[^\\/:\*?"<>|\r\n]+\\)+> 不允许在盘符或者网络共享的双反斜杠之后紧跟着出现冒号。

如果你想知道为什么我们没有在实例 7.18 和实例 7.19 中使用这个技巧，那是因为这些正则表达式并不要求一定有文件夹。因为在那些正则式中，在匹配驱动器的部分之后的所有内容都是可选的，因此正则引擎从不需要回溯。当然，如果把内容都变成可选的也会带来不同的问题，这在实例 7.19 中已经讨论过。

当这个正则式找到一个匹配的时候，第一个捕获分组中会包含盘符或网络共享名，而第二个捕获分组中会包含文件夹。如果是相对路径，那么第一个捕获分组会是空的。第二个捕获分组则总是会至少包含一个文件夹。如果你使用这个正则式来匹配一个没有指定文件夹的路径，那么这个正则式根本就不会产生匹配。

## 参见

实例 2.9 会告诉你关于捕获分组的一切。

要了解如何在你喜欢的编程语言中获取捕获分组匹配到的文本，请参考实例 3.9。

如果你事先不能确定字符串中是否包含了合法的 Windows 路径，那么需要先按照实例 7.19 进行判断。

## 7.23 从 Windows 路径中抽取文件名

### 问题描述

你有一个字符串，其中包含了一个指向 Windows PC 或网络上的文件或文件夹的（语法上）合法的路径，你想要从路径中抽取其中的文件名（如果有）。例如，你想要从路径 c:\folder\file.ext 中抽出 file.ext。

### 解决方案

```
[^\\/:*?"<>|\\r\\n]+$
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

从已知包含合法路径的字符串中抽取网络服务器和共享名称是很简单的，即使你不知道该路径实际上是否会以一个文件名结束。

文件名总是会出现在字符串的结尾处。它不能包含任何冒号或反斜杠，因此它也就不会同文件夹、盘符或网络共享名产生混淆，因为它们都会用到反斜杠和/或冒号。

定位符 \$ 会匹配字符串的结束（实例 2.5）。在 Ruby 中，虽然脱字符同样可以匹配内嵌的换行，但是这也不会带来问题，因为在合法的 Windows 路径中不会包含换行。否定字符类型 <[^\\/:\*?"<>|\\r\\n]+> （实例 2.3）会匹配可以出现在文件名中的字符。虽然正则引擎会从左向右扫描字符串，在正则式末尾的定位符会确保只有位于字符串结尾的文件名字符集合才会被匹配，这样就可以得到我们想要的文件名。

如果字符串以反斜杠结束，例如没有指定文件名的路径，那么正则式就不会产生任何匹配。当它成功匹配的时候，它就只会匹配到文件名，因此我们就不再需要使用任何捕获分组来把文件名同路径中的其余部分分隔开来。

## 参见

要了解如何在你喜欢的编程语言中获取捕获分组匹配到的文本，请参考实例 3.9。

如果你事先不能确定字符串中是否包含了合法的 Windows 路径，那么需要先按照实

例 7.19 进行判断。

## 7.24 从 Windows 路径中抽取文件扩展名

### 问题描述

你有一个字符串，其中包含了一个指向 Windows PC 或网络上的文件或文件夹的（语法上）合法的路径，你想要从路径中抽取其中的文件扩展名（如果有）。例如，你想要从路径 c:\folder\file.ext 中抽出其中的.ext。

### 解决方案

\. [^.\\\/\*?"<>|\r\n]+\$

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 讨论

在这里我们可以使用同在实例 7.23 中抽取整个文件名一样的技巧来抽取其中的文件扩展名。

唯一的区别是如何来处理点号。在实例 7.23 中的正则表达式并没有包含任何点号。在该正则式中的否定字符类只是会简单匹配恰好在文件名中出现的任何点号。

文件扩展名必须以点号作为开始。因此，我们在正则式的开始插入了 \. 来匹配一个字面的点号。

类似 Version 2.0.txt 这样的文件名中可能会包含多个点号。最后一个点号是用来把扩展名同文件名分开的。扩展名自身不应当包含任何点号。在正则式中，我们通过在字符类中放一个点号来说明这一点。这个点号在字符类中只是一个字面字符，因此我们不需要对它进行转义。

出现在正则式结尾的定位符 <\\$> 会确保我们匹配的是.txt 而不是.0。如果字符串以反斜杠结束，或者是以不包含任何点号的文件名结束，那么这个正则式就不会匹配到任何内容。当它产生匹配的时候，它就会匹配到扩展名，其中包括用来分隔扩展名和文件名的点号。

### 参见

如果你事先不能确定字符串中是否包含了合法的 Windows 路径，那么需要先按照实例 7.19 中的讲解来进行判断。

## 7.25 去除文件名中的非法字符

### 问题描述

你要去除一个字符串中不符合 Windows 文件名要求的字符。例如，你有一个包含文档

标题的字符串，当用户第一次单击“保存”按钮时，你想要用它来作为默认文件名。

## 解决方案

### 正则表达式

[\\/:/\*?<>|]+

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

### 替换

替换文本为空。

替代文本流派：.NET、Java、JavaScript、PHP、Perl、Python、Ruby

## 讨论

下列字符在 Windows 文件名中是不合法的：`\/*?<>|`。这些字符会被用来分隔盘符和文件夹、引用路径或用来指定通配符和命令行的重定向符。

我们可以容易地用字符类 `\/*?<>|` 来匹配这些字符。反斜杠是字符类中的元字符，所以需要用另外一个反斜杠来转义。所有其他字符在字符类中都是字面字符。

出于效率的考虑，我们用 `++` 来对字符类进行重复。这样，如果字符串中包含一串连续的非法字符，那么整串非法字符都会被一次性去除，而不是逐个字符去除。当处理一小段字符串时，你可能不会注意到性能上的区别，但当你处理一大段数据时，就很可能会包含比较长的需要删除的字符串，因此这是一种值得牢记的好技巧。

因为我们只需要删除有问题的字符，所以在执行查找和替换操作的时候使用空字符串作为替代文本。

## 参见

实例 3.14 讲解了在你所使用的编程语言中如何使用固定的替代文本执行查找和替换操作。

# 标记语言和数据交换

在这最后一章中，我们会着重讲解在处理常见的标记语言时可能会遇到的常见任务，这些标记语言包括：HTML、XHTML、XML、CSV 和 INI。虽然我们会假设读者对于这些技术至少已经有了基本的了解，但是在本章开始时，我们还是会对每种语言进行简单的介绍，以确保在进一步讲解之前都站在同一起跑线上。这里的描述主要关注的是在搜索每种格式的数据结构时所需要知道的基本语法规则。当我们在后面遇到相关问题的时候，届时会再引入更多的细节。

要想准确处理和操作其中的一些格式虽然从表面上看并不总是很难，然而有时候这样的任务会极其复杂，至少使用正则表达式来处理时是如此。如果需要完成本章中讲解的许多任务，特别是当在准确性非常关键的时候（例如，如果你的处理过程可能会涉及系统安全），那么通常最好的选择应当是使用专门的分析器和 API，而不是采用正则表达式。然而，这些实例依然介绍了许多有用的技巧，并且可以用于许多需要快速处理的任务。

因此，我们先来看一下要处理的到底是什么东西。在本章中会遇到的许多困难都涉及如何处理与下列规则不一致的情形，这其中可能是可以预期的情形，也可能是无法预期的情形。

### 超文本标记语言 (HTML)

HTML 被用来描述数十亿的网页和其他文档所采用的结构、语义和表现形式。使用正则表达式来处理 HTML 是比较常见的任务，但你还是应当事先清楚该语言并不适合正则表达式的严格性与精确性。对于在许多网页中常见的各种杂乱的 HTML 尤其如此，原因是如今的网页浏览器对于结构糟糕的 HTML 依然表现出极端的容忍。在本章中，我们会关注处理格式良好的 HTML 所需的规则，其中包括：元素（以及它们所包含的属性）、字符引用、注释和文档类型声明。本书要讲解的版本是 HTML 4.01，这个版本是 1999 年最后定稿的，同时也是在本书写作之时最后一个定稿的 HTML 版本。

最基本的 HTML 构造模块被称作元素 (element)。元素是使用标签 (tag) 来表示的，标签会使用两个尖括号包围起来。元素可以被分为块级 (block-level) 元素（例如段落、标题、列表、表格和表单）和内联级 (inline) 元素（例如超链接、引用、斜体和表单输入控制）。元素通常会包含一个起始标签（例如<html>）和一个终止标签（例如</html>）。元素的起始标签中可以包含属性 (attribute)，这会在后面进行解释。在两个标签之间的是元素的内容，其中可能会包含文本和其他元素，或者是空的。元素还可以嵌套，但是不能交叠（例如，可以使用<div><div></div></div>，但是不能使用<div><span></div></span>）。对于有些元素（例如用来标记段落的<p>），终止标签是可选的。拥有可选的终止标签的元素会在新的块级元素的开始处自动终止。有些元素（包括用来终止一行内容的<br>）中不能包含任何内容，而且也永远不会使用终止标签。然而，一个空元素依然可以包含属性。HTML 元素名称都会以 A~Z 的一个字母开头。所有合法元素在它们的名称中只会使用字母和数字。元素名称不区分大小写。

我们需要特别注意两个元素<script> 和 <style>，因为它们会分别允许你在文档中嵌入脚本语言的代码和样式表 (stylesheet)。第一次出现 </style> 或者 </script> 的位置会作为这些元素的结束，并且结束标签也可以出现在注释中，或者是出现在样式或脚本语言内的字符串中。

属性会出现在一个元素的起始标签中，位于元素名称之后，多个属性之间会使用一个或多个空格分隔。大多数属性的形式都是一对“名称-值”。因此，下面的例子所给的是一个<a> (anchor) 元素，它包含了两个属性以及内容 “Click me!”：

```
<a href="http://www.regexcookbook.com"
    title = 'Regex Cookbook'>Click me!</a>
```

如上所示，一个属性的名称和值之间会用一个等号进行分隔，另外还可以包含可选的空格。值会用单引号或双引号括起来。如果要在值中使用对应的引号，那么就需要使用字符引用（稍后会进行讲解）。如果值中只包含字符 A~Z、a~z、0~9、下划线、句点、冒号和连字符（如果用正则表达式表示，也就是`^[-.0-9:A-Z_a-z]+$`），那么也可以不使用引号。有些属性（例如在有些表单元素中的属性 selected 和 checked）只要出现就会影响包含它们的元素，它们并不需要再带一个值。在这些情况下，用来分隔属性名称和值的等号也会被省略掉。另外一种表示方法是可以把这些属性的名称作为它们的值出现（例如，`selected="selected"`）。属性名称都会以字母 A~Z 开头。所有合法属性的名称中只能包含字母和连字符。属性可以按照任意顺序出现，而且它们的名称是不区分大小写的。

HTML 第 4 版定义了 252 个字符实体引用 (character entity references) 以及超过 100 万个数字字符引用 (numeric character references)，二者加起来被统称为字符引用 (character references)。数字字符引用会使用一个字符的 Unicode 代码点来指代一个字符，它采用的格式是`&#nnnn` 或`&xhhhh`，其中 `nnnn` 是一个或多个 0~9 的十进制数字，而 `hhhh` 是一个或多个 0~9 和 A~F (不区分大小写) 的十六进制数字。字符实体引用被写作

`&entityname;` (区分大小写, 这与大多数其他 HTML 特性不同), 当需要输入在某些情境中比较敏感的字面字符, 例如尖括号 (< 和 >)、双引号 ("") 和 and 符号 (&amp;) 时, 它们会格外有用。

同样很常见的还有实体引用 `&ampnbsp` (即不换行的空格, 位置 0xA0), 这个实体是很有用的, 因为它会使该字符的所有出现都进行显示, 如果它们连续出现的时候会显示多次。而空格、制表符和换行符通常都会被显示为单个的空格字符, 即使它们中的许多个连续出现也只显示一次。在字符引用之外不能使用 and 字符 (&)。

下面是 HTML 中注释的语法:

```
<!-- this is a comment -->
<!-- so is this, but this comment
spans more than one line -->
```

在注释中的内容不拥有特殊的含义, 并且对于大多数用户代理 (agent) 来说都是看不到的。在结束的 -- 和 > 之间允许出现空格。为了保持与早期 (1995 年之前) 的浏览器兼容, 有些人会把 `<script>` 和 `<style>` 元素的内容放到一个 HTML 注释中。现代浏览器则会忽略这些注释, 按照正常的方式来处理脚本或样式内容。

最后, HTML 文档通常会使用一个文档类型声明 (document type declaration) 来作为开始, 有时候也被非正式地称作一个“DOCTYPE”, 它会用来标识一种机器可读的规范, 说明该文档允许和禁止出现的内容。DOCTYPE 有点类似一个 HTML 元素, 下面所给的例子中说明该文档希望严格遵守 HTML 4.01 的严格定义:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
```

上面只给出了 HTML 文档的物理结构的一个简要介绍。需要注意的是现实世界中的 HTML 通常会偏离这些规则, 而大多数浏览器也都会很乐意适应这些变体。除了这些基本规则, 在一个合法的 HTML 文档中, 每个元素对于在其中可以出现的内容和属性也会有一些限制。这些内容规则超出了本书的范围, 但是如果读者希望了解更多的信息, 我们推荐读者参考 O'Reilly 出版社出版的, 由 Chuck Musciano 和 Bill Kennedy 所著的 *HTML & XHTML: The Definitive Guide* 一书。



### 提示

因为 HTML 的结构与 XHTML 和 XML (这两种语言随后会进行介绍) 非常类似, 因此本章中的许多正则表达式都会被设计成同时支持所有这 3 种标记语言。

## 可扩展的超文本标记语言 (XHTML)

XHTML 在设计时是作为 HTML 4.01 的后续版本, 它把 HTML 从它的 SGML 传统格式转向了以 XML 为基础的格式。然而, HTML 的发展现在又走向了另外一条路, 因此

XHTML 更准确地说可以被认为是 HTML 的一个替代选择。本书会涉及 XHTML 1.0 和 XHTML 1.1 版本。虽然这些版本的标准大多与 HTML 是向后兼容的，但是其中与我们刚刚讲过的 HTML 结构还是存在一些关键的区别。

- XHTML 文档可以用一个 XML 声明作为开头，例如`<?xml version="1.0" encoding="UTF-8"?>`。
- 非空元素必须包含一个结束标签。空元素则必须使用一个结束标签或者以`>`作为结束。
- 元素和属性名称是区分大小写的，并且要使用小写字母形式。
- 由于使用了 XML 命名空间前缀，元素和属性名称中除了 HTML 名称中允许的字符之外，还可以包含一个冒号。
- 不允许出现没有使用引号括起来的属性值。属性值必须只能出现在两个单引号或双引号之间。
- 属性必须拥有一个对应的值。

在 HTML 和 XHTML 之间还存在许多其他的区别，通常被用来处理边界情况和错误处理，因此它们通常不会影响到本章中的正则表达式。如果希望了解 HTML 和 XHTML 之间区别的更多信息，请参考 <http://www.w3.org/TR/xhtml1/#diffs> 和 [http://wiki.whatwg.org/wiki/HTML\\_vs.\\_XHTML](http://wiki.whatwg.org/wiki/HTML_vs._XHTML)。



#### 提示

因为 XHTML 的结构与 HTML 非常类似，而它又是从 XML 而来的，因此本章中的许多正则表达式都会被设计来支持所有这 3 种标记语言。如果一个实例中标明了“(X)HTML”，那么它可以同时处理 HTML 和 XHTML。一般很难遇到一个文档中只使用 HTML 或 XHTML，因为现在通常遇到的都是混合的情形，而浏览器也会对此提供支持。

## 可扩展标记语言 (XML)

XML 是一种通用语言，主要设计的目的是为了共享有结构的数据。以它为基础创建了很多种标记语言，其中就包括了我们刚刚介绍的 XHTML。本书会涉及 XML 的 1.0 版和 1.1 版。本书无法对 XML 特性和语法进行完整的描述，但是为了方便起见，我们会指出它与刚刚介绍过的 HTML 结构之间的关键区别。

- XML 文档可以以一个 XML 声明作为开始，例如`<?xml version="1.0" encoding="UTF-8"?>`，它也可以包含其他类似格式的处理指示。例如，`<?xmlstylesheet type="text/xsl" href="transform.xslt"?>`会声明需要把 XSL 转换文件 transform.xslt 应用到该文档之上。

- DOCTYPE 中可以在方括号之内包含内部的标记声明。例如：

```
<!DOCTYPE example [
  !ENTITY copy "&#169;">
  !ENTITY copyright-notice "Copyright &copy; 2008, O'Reilly Media">
]>
```

- CDATA 部分可以用来对文本段进行转义。它们以字符串 <![CDATA[ 作为开始，并且在遇到第一个 ]]> 之后结束。
- 非空元素必须包含一个结束标签。空元素则必须使用一个结束标签，或者以 /> 作为结束。
- XML 名称（它会管理元素、属性和实体引用的名称规则）是区分大小写的，而且可以使用很大一部分 Unicode 字符。其中允许出现的字符包括 A~Z、a~z、冒号 (:) 和下划线 (\_)，并且在第一个字符之后允许出现 0~9、连字符 (-) 和句点 (.)。更多细节请参考实例 8.4。
- 不允许出现没有使用引号括起来的属性值。属性值必须只能出现在单引号或双引号之间。
- 属性必须拥有一个对应的值。

除此之外，要编写一个格式良好的 XML 文档，或者是你想要编写自己的 XML 解释器，那么还需要遵守许多其他的规则。然而，对于简单的正则表达式查找来说，我们刚刚介绍过的规则（再加上已经介绍过的 HTML 文档结构规则）通常来说就足够了。



### 提示

因为 XML 的结构与 HTML 非常类似，而它又是 XHTML 的基础，因此本章中的许多正则表达式都会被设计来支持所有这 3 种标记语言。如果一个实例中标明了处理的是“XML 风格”的标记，那么它可以同时处理 XML、XHTML 和 HTML。

## 逗号分隔值 (CSV)

CSV 是一种虽然很古老但是依然非常常用的文件格式，它可以用来自表示类似电子表格的数据。现在大多数的电子表格和数据库管理系统都依然支持 CSV 格式。在应用程序之间进行数据交换时，CSV 是尤为常见的格式。虽然并不存在官方的 CSV 规范，但是在 2005 年 10 月发布的 RFC 4180 标准尝试为 CSV 提供通用的定义，并且在 IANA 注册了 MIME 类型“ext/csv”。在这个 RFC 发布之前，Microsoft Excel 中使用的 CSV 风格通常被或多或少地当作一个事实上的标准。因为 RFC 所规定的规则与 Excel 使用的规则非常类似，所以这并没有带来太大的问题。本章会涉及 RFC 4180 中规定的以及在 Microsoft Excel 2003 及更高版本中使用的 CSV 格式。

从它的名字中就可以看出来，CSV 文件中包含由逗号作为分隔的值或域 (field) 的列

表。每行，或者称作一个记录（record），会出现在自己的一行中。一个记录中的最后一个域之后不用跟着逗号。一个文件中的最后一个记录之后可以也可以不必跟一个换行符。在整个文件中，每个记录都必须拥有相同数量的域。

每个 CSV 域的取值可以是没有修饰的，也可以用双引号括起来。域同样也可以是全空的。如果一个域中包含了逗号、双引号或者换行符，那么它必须用双引号括起来。出现在一个域内部的双引号会在之前加一个双引号来进行转义。

在一个 CSV 文件中的第一个记录有时候会被用作包含每列名称的一个标题行。因为无法通过编程来完全根据一个 CSV 文件的内容来决定第一行是否是标题行，因此有些应用程序会提示用户决定第一行应该如何进行处理。

RFC 4180 中规定了在一个域中前导和拖尾的空格都会被当作该值的一部分。有些老版本的 Excel 会忽略这些空格，但是 Excel 2003 和更高版本都在这一点上遵守了 RFC 的规定。RFC 并没有规定对于没有转义的双引号或者几乎其他任何出错情况应该如何进行处理。Excel 的错误处理在一些边界情况下也会出现无法预测的情形，因此一定要确保双引号都进行了转义，而且包含双引号的域都使用双引号括起来，并且括起来的域不会在双引号之外包含前导或者拖尾的空格。

下面的 CSV 示例展示了我们刚刚介绍过的许多规则。它包含了 2 个记录，每个记录包含 3 个域：

```
aaa,b b,"""c"" cc"  
1,, "333, three,  
still more threes"
```

表 8-1 给出了上面的 CSV 内容在表格中表现出来的形式。

表 8-1 CSV 输出示例

aaa	b b	"c" cc
1	(空)	333, three, still more threes

我们刚刚介绍的是本章中使用的 CSV 规则，然而在不同程序对 CSV 文件的读写过程中，还是会存在相当多的变体。许多应用程序中甚至会允许使用“csv”扩展名的文件中使用除了逗号之外的任意分隔符。其他常见的变体包括如何在域中嵌入逗号（或其他域分隔符）、双引号和换行符，以及在没有使用引号的域中如何处理前导和拖尾的空格（可以忽略它们，或者当作字面文本来处理）。

## 初始化文件 (INI)

轻量级的 INI 文件格式通常被用作配置文件。它不拥有严格的定义，因为当该格式被不同程序和系统解释的时候，会出现相当多的变体。本章中的正则表达式会遵守我们下

面要介绍的最常见的INI文件约定。

INI文件中的参数（parameter）是名称-值对，使用等号和可选的空格或制表符进行分隔。值会被包在单引号或双引号之间，这样就允许它们包含前导或拖尾的空格与其他特殊字符。

参数会被组织成多个段（section），每个段会以只包含用方括号括起来的段名称的一行作为开始。段的内容会一直延伸到下一个段的声明，或者是文件结束。段之间不能嵌套。

分号用来标记一个注释的开始，注释会延伸到一行的结束。注释可以同一个参数或段声明出现在同一行中。在注释中包含的内容不拥有任何特殊含义。

下面是一个INI文件的例子，其中包含了一个介绍性注释（用来说明文件最后一个修改的日期）、两个段（“user”和“post”）和总共三个参数（“name”、“title”和“content”）：

```
; last modified 2008-12-25

[user]
name=J. Random Hacker

[post]
title = Regular Expressions Rock!
content = "Let me count the ways..."
```

## 8.1 查找 XML 风格的标签

### 问题描述

你想要匹配一个字符串之中的任意HTML、XHTML或XML标签，目的是对它们执行删除、修改、统计或者其他操作。

### 解决方案

什么是最适合的解决方案取决于几个因素，包括你认为可以接受的准确度、效率和对错误标记的容忍程度。一旦确定了适合你的需要的方式，接着就可以对得到的结果执行各种各样的操作。然而，不管你是想要删除标签、在标签之内进行查找、添加或删除属性，还是把它们替换为其他的标记，第一步都是要首先找到它们。

请注意这个实例会比较长，而且其中充满了微妙的细节、例外情况和各种变化形式。如果你要找的是一个简单快速的解答，而不想花费很多时间来确定适合你需求的最佳解决方案，那么可以选择直接跳到本实例中的“(X)HTML 标签（灵活方案）”小节，在那里你会找到一个适合最普通的大众化需求的解答。

## 速成方案

下面给出的第一种解决方案非常简单，而且你可能想不到它的使用也很频繁，然而我们把它放在这里主要是用于对比的目的，并且会逐一检查它的不足之处。如果你知道正在处理的文本是什么内容，而且也并不过分关心错误处理会带来的后果，那么它有时候也足够用了。这个正则表达式会首先匹配一个 <，然后简单地匹配到出现第一个 > 为止：

```
<[^>]*>
正则选项：无
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

### 在属性值中允许出现 >

下面这个正则式其实相当简单，而且也无法正确处理所有的情形。然而，如果你只用它来处理合法的(X)HTML 片段，那么它也可能足够满足你的需要。与上一个正则式相比，它的优点是能够正确忽略掉出现在属性值之中的 > 字符：

```
<(?:[^>"']+"[^"]*"|'[^']*')*>
正则选项：无
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面是这个正则式的宽松排列形式，为了提高可读性，在其中添加了空白和注释：

```
<
 (?: [^>"']          # 没有被引起来的字符，或者...
    | "[^"]*"        # 双引号引起来的属性值，或者...
    | '[^']*'         # 单引号引起来的属性值
)*
>
正则选项：宽松排列
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

刚刚所给的两个正则式用起来是完全一样的，因此可以选择使用你喜欢的形式。如果你用的是 JavaScript，那么只能使用第一种形式，因为 JavaScript 不支持宽松排列模式。

### (X)HTML 标签（灵活方案）

除了会在属性值中支持嵌入的 > 字符，下一个正则式还会模拟浏览器中通常会实现的关于(X)HTML 标签名称的灵活模式。它会让正则式避免匹配到看起来不像是标签的内容，这其中包括了注释、DOCTYPE 和在文本中没有进行编码的 < 字符。它在处理属性和可能出现在标签中的其他字符的时候，使用的是与上一个正则式相同的方式，但是它还添加了对标签名称的特殊处理。具体来说，它要求标签名称必须以英语字母开头。标签名称会被捕获到第 1 个向后引用中，以便你可能需要在后面使用它：

```
</?([A-Za-z][^\s>/]*)(&?:[^>"']+"[^"]*"|'[^\']*')*>
正则选项：无
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面给出了这个正则式的宽松排列形式：

```
<
/?
([A-Za-z][^\s>/]*)
(?: [^>"]
| "[^"]*"#
| '[^']*'#
)*
>

正则选项：宽松排列
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

上面这两个正则式用起来也是完全一样的，然面后者不能用于 JavaScript，因为它不支持宽松排列的选项。

## (X)HTML 标签（严格方案）

相比本实例中前面所给的正则表达式，下面这个要复杂很多，因为它会实际遵守我们在本章最开始处所介绍的(X)HTML 标签的规则。这并不总是有必要的，因为浏览器也不会严格遵守这些规则。换句话说，这个正则表达式会避免匹配到不像是合法的(X)HTML 标签的内容，但是有可能要付出的代价是无法匹配到浏览器，事实上会解释为标签的一些内容（例如，如果你的标记中使用的属性名称里包括了这里没有提到的字符，或者是属性被添加到了结束标签中）。这里同时处理了 HTML 和 XHTML 的标签，因为它们的两种语法通常会被混起来使用。标签名称被捕获到了第 1 个或者第 2 个向后引用中（根据它是一个起始标签还是结束标签），从而可以在后面使用该标签的名称：

```
<(?:([A-Z][-:A-Z0-9]*)|(?:\s+[A-Z][-:A-Z0-9]*|(?:\s*=|\s*(?:"[^"]*"|  
'[^']*'|[-.:\\w]+))?)|\s*/?|/([A-Z][-:A-Z0-9]*)\s*)>

正则选项：不区分大小写
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

为了解释其中的奥妙，下面给出了上面正则表达式的宽松排列形式，其中添加了注释：

```
<#
(?:#
([A-Z][-:A-Z0-9]*))#
(?:#
\s+#
[A-Z][-:A-Z0-9]*#
)#
\s*=|\s*#
(?:#
"[^"]*"#
| '[^']*'#
| [-.:\\w]+#
)#
)?#
)*#
```

```

\s*          # 允许拖尾的空白
/?          # 允许自己结束的标签 (XHTML)
|
# 结束标签的分支...
/
([A-Z][-A-Z0-9]*) # 把结束标签捕获到第 2 个向后引用中
\s*          # 允许拖尾的空白
)
#
>
正则选项: 不区分大小写、宽松排列
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

```

## XML 标签 (严格方案)

XML 是一种拥有精确规范的语言，它要求用户代理严格遵守并且强制应用它的所有规则。这同 HTML 以及大家常用的用来处理它的长期受困的浏览器来说，是一个彻底的转变：

```

<(?:([_A-Z][-.\w]*)(?:\s+[_A-Z][-.\w]*\s*=\s*(?:"[^"]*"|'[^\']*'))*\s*+>
/?|/([_A-Z][-.\w]*\s*)>
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```

下面，我们还是给出了添加注释之后的按照宽松排列模式显示的同一个正则表达式：

```

<          #
(?:         # 起始标签的分支...
  ([_A-Z][-.\w]*)) # 把起始标签名称捕获到第 1 个向后引用
  (?:         # 支持 0 个或多个属性...
    \s+
    [_A-Z][-.\w]* # 属性名称
    \s*=\s*         # 属性名称-值之间的分隔符
    (?: "[^"]*" # 双引号引起的属性值
      | '[^\']*' # 单引号引起的属性值
    )
  )*
  \s*          # 允许拖尾的空白
  /?          # 允许自己结束的标签
|
# 结束标签的分支...
/
([_A-Z][-.\w]*) # 把结束标签捕获到第 2 个向后引用中
\s*          # 允许拖尾的空白
)
>
正则选项: 不区分大小写、宽松排列
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

```

同我们前面所给的(X)HTML 标签的两种解答一样，这些正则式也会把标签名称捕获到第 1 个或者第 2 个向后引用中，这会取决于匹配到的是一个起始标签还是结束标签。XML 标签的正则表达式与前面的(X)HTML 版本相比稍微要短一些，这是因为它不用

处理只有 HTML 才支持的语法（不带值的属性和没有引起来的值）。它同样允许在元素和属性名称中使用更大范围的字符。

## 讨论

### 几句箴言

尽管想要使用正则表达式来匹配 XML 风格的标签是很常见的任务，要安全地完成这个任务则需要做出多种权衡，并且需要认真考虑你要处理的数据。由于存在这些困难，所以有些人选择不使用正则表达式来处理任意类型的 XML 或(X)HTML 任务，而是选择采用特殊的分析器和 API。这也是我们强烈建议读者认真考虑的方式，因为这些工具通常是经过优化的，可以很快执行它们的指定任务，而且它们还包含了对于错误标记的健壮检查或错误处理功能。例如，在浏览器的领域中，通常最好利用基于树形结构的文档对象模式 (DOM) 来执行 HTML 的查找和处理需求。在其他场合，一般可以使用一个 SAX 分析器或 XPath。然而，偶尔你也许会遇到一些场合，它们更适合使用基于正则表达式的解决方案。

在了解了这些事实之后，我们再来仔细看一下本实例前面所给的正则表达式。对于大多数情形来说，最前面两个解答都有些过于简单，但是可以同时处理 XML 风格的标记语言。后面 3 种形式遵循了更加严格的规则，因此可以被裁剪来适应相应的标记语言。然而，即使在后面几个解答中，HTML 和 XHTML 标签的规则也还是一起来处理的，这是因为它们常常会在不经意间被混合起来使用。举例来说，一个作者可能会在一个 HTML 文档中使用了一个 XHTML 风格的、自结束的 `<br />` 标签，或者也可能会在一个拥有 XHTML DOCTYPE 的文档中使用了大写的元素名称。

### 速成方案

这种解决方案的优点是它的简单性，这不仅使其容易让人记住和输入，而且运行速度也很快。因而带来的缺点是它会不正确地处理某些合法和不合法的 XML 和(X)HTML 结构。如果你要处理的是自己写的标记文件，并且知道在目标文本中肯定不会出现这种情形，或者说即使它们出现了，你也不在乎它可能会带来的后果，那么这样的权衡折中可能是可以接受的。可以使用这个解决方案的另外一个例子是你使用的文本编辑器会支持对正则匹配进行预览。

这个正则式一上来先会查找一个字面的 `<>` 字符（它意味着一个标签的开始）。接着它使用了一个否定字符类和贪心的星号量词 `<[^>]*>` 来匹配非 `>` 的 0 个或多个字符。这样就可以匹配标签的名称、属性和一个前导或拖尾的 `/` 字符。我们也可以在这里使用一个惰性量词 (`<[^>]*?>`)，但是这样并不会改变任何结果，只是会导致这个正则式的速度稍微慢一点，因为它会造成更多的回溯（参见实例 2.13）。作为标签的结束，这个正则式最后匹配了一个字面的 `<>`。

如果你更喜欢在否定字符类 `<[^>]>` 的地方使用一个点号来替代，那么也可以这样做。只要同时使用了惰性量词 `(.*?)`，并且确保打开了“点号匹配换行符”的选项（在 JavaScript 中，你可以使用`<[\s\S]*?>`），那么它也能够正确执行。点号再加上贪心星号量词（也就是把整个模式改成`<.*>`）会改变正则表达式的含义，造成它错误地匹配到目标字符串中从第一个 `<` 直到最后一个 `>` 之间的内容，而在此过程中，正则式也可能会吞掉多个其他标签。

下面我们来介绍几个例子。这个正则式会整体匹配下面的每一行内容：

```
<div>
</div>
<div class="box">
<div id="pandoras-box" class="box" />
<!-- comment -->
<!DOCTYPE html>
<< < w00t! >
<>
```

需要注意的是这个模式匹配到的不只是标签。更糟糕的是，它无法正确匹配到在目标字符串 `<input type="button" value=">>">` 或者 `<input type="button" onclick="alert(2 > 1)">` 之中的整个标签。相反地，它只会匹配到属性值中出现的第一个 `>`。在处理注释、XML CDATA 片段、DOCTYPE、在`<script>`元素中的代码以及任何其他包含内嵌的 `>` 符号的内容的时候，都会遇到类似的问题。

如果你要处理的不只是最基本的标记，特别是如果目标文本来自混合的或者未知的来源，那么你可能就需要使用本实例中后面给出的一种更加健壮的解决方案。

## 在属性值中允许出现 `>`

与上面刚刚讲解过的速成方案的正则式一样，接下来的这一个正则表达式被引入的目的主要是为了同后面更加健壮的解决方案作比较。然而，它还是覆盖了用来匹配 XML 风格标签的基本需求，因此如果你只是用它来处理其中仅包含有元素和文本的合法标记的代码片段，那么它也可以正常工作。与上一个正则式的区别是它会忽略出现在属性值中的 `>` 字符。例如，它能够正确匹配在我们前面提到过的字符串 `<input type="button" value=">>">` 和 `<input type="button" onclick="alert(2 > 1)">` 之中的整个 `<input>` 标签。

与前面一样，这个正则式在两头使用了字面的尖括号字符来匹配一个标签的开始和结束。在二者之间，它会重复一个非捕获分组，其中包含了 3 个选择分支，彼此之间通过多选结构元字符 `\|` 进行分隔。

其中第一个选择分支是否定型字符类 `<[^>"]>`，它会匹配除了右尖括号（用来结束标签）、双引号或单引号（两个引号都可以用作属性值的开始）之外的任意单个字符。这第一个选择分支负责匹配标签和属性名称，以及在被引起的值以外的任意其他字符。

这些选择分支之间的顺序是有意安排的，在设计时考虑了执行的效率。正则表达式引擎会从左向右尝试一个正则表达式中的不同选择分支路径，而匹配这第一个选项的尝试很可能总是会比用来匹配引起来的值的选择分支的成功可能性要大很多（特别是考虑到它一次只会匹配一个字符）。

接下来的选择分支会匹配双引号和单引号括起来的属性值（`<"[^"]*'">` 和 `<'[^']*'">`）。它们使用了否定字符类，从而在匹配的时候可以跨过其中包含的 `>` 字符、换行符和非结束引号的其他任何内容。

需要注意的是，这个解决方案并没有做任何特殊的处理来排除或者正确匹配注释，以及在你的文档中的其他特殊节点。所以在真的使用这个正则表达式之前，一定要确保你很熟悉要处理的内容种类。

### 一种（安全的）效率优化方法

在阅读本节之后，你可能会认为如果在前面的否定字符类（`<[^>]">`）之后添加一个量词 `(*|+)`，就能够提高这个正则表达式的性能。如果正则式会在目标字符串内部的位置找到匹配，那么你这样想是正确的。通过每次匹配多于一个字符，你就能够使正则引擎在到达成功匹配的途中省略掉许多不必要的步骤。

然而在正则引擎只能找到部分匹配的时候，这样的改动还可能会带来一个可能看起来不是那么明显的消极后果。当正则式匹配到一个起始的 `<` 字符，但是在后面却无法找到相应的`>`，那么你就会遇到我们在实例 2.15 中讲解过的“灾难性回溯”的问题。这是因为新添加的内层量词与（在非捕获分组之后的）外层量词组合起来，就会产生无数种方式可以匹配在 `<` 之后跟着的文本，而正则引擎必须要在尝试所有这些方式之后才能宣布本次匹配尝试失败。

对于支持占有量词或者原子分组的正则流派（JavaScript 和 Python 二者都不支持），可以在依然能够获得每次匹配多个字符带来的性能好处的前提下，避免上述问题的出现。事实上，我们可以更进一步减少在正则式的其他地方可能会出现的回溯。如果你所使用的正则流派这两个特性都支持，那么使用下面给出的占有量词形式更好，因为它们会减短正则式的长度，使之更加易读。

采用原子分组的形式：

```
<(?:(>[^>"]++|"[^"]*"|'[^\']*')*)>  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Ruby
```

采用占有量词的形式：

```
<(?:[^>"]++|"[^"]*"|'[^\']*')*+>  
正则选项：无  
正则流派：Java、PCRE、Perl 5.10、Ruby 1.9
```

## (X)HTML 标签（灵活方案）

这个正则式中只添加了很简单功能，就可以做到非常接近当前浏览器在源代码中识别(X)HTML 标签时所采用的灵活规则。当你试图要模仿浏览器的行为，而不需要考虑你所匹配到的内容实际上是否遵守了合法标记的所有规则时，这个正则式会是一个较好的解决方案。要记住的是，在处理某些非常奇怪的非法 HTML 的时候，这个正则式可能无法做到同某些浏览器处理得一样好，因为浏览器通常会采用自己的独特方式来分析错误标记中的边界情形。

与前一个解决方案相比，这个正则式中最重要的区别是它要求在起始的左尖括号 (<) 之后紧跟着的必须是一个 A~Z 或 a~z 的字母，在该字母之前还可以有一个可选的 / (用来表示结束标签)。这个限制就可以排除掉在文本中的许多零散的、没有编码的 < 字符，以及注释、DOCTYPE、XML 声明和处理指令，CDATA 片段，等等。这并不能让它避免匹配位于注释、脚本语言代码和 <textarea> 元素的内容等之内看起来貌似标签的东西。后面的一个小节“跳过复杂的(X)HTML 和 XML 片段”中会讲解一种解决这个问题的办法。但是我们首先还是来看一下这个正则式的工作原理。

<> 会首先匹配一个字面的左尖括号。紧跟其后的 </> 会允许出现一个用于结束标签的可选正斜杠。接下来是一个捕获分组 <([A-Za-z][^\s>/]\*)>，它会匹配标签的名称，并把它保存到第 1 个向后引用中。如果你不需要在之后引用标签的名称（例如，你只是想删除所有标签），那么可以去掉用来捕获的两个括号（但是要保留两个括号之间的模式）。在分组中有两个字符类。第一个字符类 <[A-Za-z]> 设定的是标签名称的首字母规则。接下来的字符类 <[^>/]> 允许接着出现几乎任何字符来作为名称的一部分。唯一的例外是空白字符（即<\s>，用来把标签名称和随后的任意属性分隔开来）、>（用作标签的结束）和 /（出现在结束的 > 之前，用于说明 XHTML 风格的单体标签）。任何其他字符（甚至包括引号）都可以被当作标签名称的一部分。看起来似乎有些太过宽松，但是这正是大多数浏览器所采用的方式。对于那些似是而非的标签，它们可能根本不会影响到网页上显示的任何效果，但是依然可以通过 DOM 树来进行访问，而且即使包含在标签之内的内容会被显示，这些标签也不会被当作文本来显示。

在标签名称之后是属性的处理，这部分内容是从上一个正则式中直接拿来的：  
<(?:[^>""]|[\""]\*|[^\"]\*)\*>。最后添加了一个右尖括号来结束这个标签，我们的任务就完成了。

下面的正则式会展示如何对这个模式进行修改成只匹配起始、结束或单体（自结束的）标签：

### 起始标签

```
<([A-Za-z][^\s>/]*)>(?:(^>''/>|"[^"]*"|[^\"]*')*)>  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

这个版本中，在非捕获分组中的第一个否定字符类之后添加了正斜杠 (/)，从而可以避免在除了被引起的属性值外的任何其他地方出现正斜杠。

### 单体标签

```
<([A-Za-z][^>/]*)(&?:[^>"']|["]*"|[']*)*>  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

在这里，我们在结束的右尖括号之前添加了一个必需的正斜杠。

### 起始标签和单体标签

```
<([A-Za-z][^>/]*)(&?:[^>"']|["]*"|[']*)*>  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

这里没有添加任何内容。我们删除了在原来的正则式中的起始 <> 之后出现的 </?>。

### 结束标签

```
</([A-Za-z][^>/]*)(&?:[^>"']|["]*"|[']*)*>  
正则选项: 无  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

在起始左尖括号之后的正斜杠在这里成为了匹配中必需的一部分。注意，我们有意在结束标签内允许出现属性，因为这里所依据的是“灵活的”解决方案。虽然浏览器不会使用出现在结束标签的中属性，它们也并不介意你在其中使用这样的属性。

前面的“一种（安全的）效率优化方法”边栏中讲解了如何通过使用原子分组或者占有量词来提高匹配标签的性能。这次潜在的性能提高可能会更显著，因为字符类 <[^>/]> 能匹配到的字符会与正则式稍后部分中的字符出现重叠，因此在正则引擎放弃某个部分匹配尝试之前，需要尝试更多可能的组合情形。

如果在你所使用的正则流派中支持原子分组或占有量词，那么利用它们可能会得到相当的性能优化。下面的改动也可以被应用到刚刚所给的“起始/结束/单体”特定的正则式之上：

```
</?( [A-Za-z] (>[^>/]*) ) (>(&?:(>[^>"']+)|"["]*"|[']*)*)>  
正则选项: 无  
正则流派: .NET、Java、PCRE、Perl、Ruby  
</?( [A-Za-z] [^>/]*)+(&?:[^>"']+|["]*"|[']*)*+>  
正则选项: 无  
正则流派: Java、PCRE、Perl 5.10、Ruby 1.9
```

## (X)HTML tags (严格方案)

之所以说这个解决方案是严格的，是因为它会尝试按照在本章最开始部分所讲解的 HTML 和 XHTML 语法规则来进行匹配，而不是模仿在实际中浏览器分析某个文档源

代码时所采用的规则。与前面的正则式相比，这个严格方案中添加了如下的规则。

- 标签和属性名称都必须以 A~Z 或 a~z 之中的一个字母开头，而它们的名称则只能使用字符 A~Z、a~z、0~9、连字号和冒号（如果用正则式来表示，也就是`<[:-A-Za-z0-9]+$>`）。
- 在标签名称之后不允许出现不合适的、零星的字符。在标签名称之后只能使用空格、属性（无论带不带相应的值）以及一个可选的正斜杠。
- 没有引起来的属性值中只能使用字符 A~Z、a~z、0~9、下划线、连字符、句号和冒号（以正则式来表示，也就是`<[:-A-Za-z0-9_]+$>`）。
- 结束标签中不能包含属性。

因为这个模式被划分为了 2 个分支（第一个分支是起始和单体标签，第二个分支是结束标签），根据你所匹配到的标签类型的不同，标签名称会被捕获到第 1 个或第 2 个向后引用中。如果你在之后不会再用到标签名称，也可以去掉这两组捕获括号。

### 起始和单体标签

```
<([A-Z][-:A-Z0-9]*) (?:\s+[A-Z][-:A-Z0-9]*(?:\s*=\\s*+|  
(?:"[^"]*"|'[^\']*'|[-.:\\w]+))?)*\s*/?>  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

刚好出现在结束的 `</>` 之前的 `</?>` 会允许这个正则式匹配起始和单体标签。把它删掉的话就只能用于匹配起始标签。只删除问号量词（把`</>`变为必需的），它就只能匹配单体标签。

### 结束标签

```
</([A-Z][-:A-Z0-9]*)\\s*>  
正则选项: 不区分大小写  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

在前面两个小节中，我们讲解了如何通过添加原子分组或者占有量词来获得潜在的性能提高。使用这个正则式来严格定义的路径意味着我们无法用多于一种方式来匹配相同的字符串，因此也就不必担心可能出现的回溯。这个正则式并不依赖回溯，因此，如果愿意，你也可以把位于最后的每一个量词 `(*)`、`(+)` 或 `(?)` 变成占有量词（或者使用原子分组来达到相同的效果）。虽然这样做可能会有帮助，但是为了控制本实例的篇幅，我们对于该正则式这些可能的变形略去不谈。

想知道如何可以避免匹配到在注释、标签和其他内容之内的标签，请参考后面的“跳过复杂的(X)HTML 和 XML 片段”小节。

## XML 标签（严格方案）

XML 采用了严格的规范，并且要求支持它的解析器不要处理格式不正确的标记，因此

我们不需要再给 XML 也来一个“灵活方案”。虽然你也可能会使用前面所给的一个正则式处理 XML 文档，但是它们的简单性无法让你得到执行更可靠查找的好处，因为我们并不需要去模拟任何的 XML 用户代理行为。

这个正则式本质上是前面所给的“(X)HTML 标签（严格方案）”正则表达式的简化版本，因为我们从中删除了在 XML 中不支持的两个 HTML 特性：没有引起来的属性值和最小化的属性（也就是不带对应值的属性）。其他唯一的区别是在标签和属性名称中允许出现的字符。事实上，XML 名称规则（用来规定对标签和属性名称的要求）比我们这里所给的要更为宽松，它会允许成千上万个其他 Unicode 字符。如果需要在查找中支持这些字符，那么你可以把其中 3 处出现 <[\_:A-Z][-.:\\w]\*> 的地方替换为在实例 8.4 中讲解的一种模式。注意 XML 支持的字符与你所使用的 XML 版本是有关系的。

与(X)HTML 正则表达式一样，根据你匹配到的是起始/单体标签还是结束标签，标签名称会被捕获到第 1 个或第 2 个向后引用中。当然，如果你不需要在后面使用这些名称，那么也可以去掉用来捕获的括号。

在下面的例子中，上述模式的 2 个分支会被分别放到它们自己的正则式中。这样做的结果是，两个正则式都会把标签名称捕获到第 1 个向后引用中：

### 起始和单体标签

```
<(_:[_A-Z][-.:\\w]*)>(?:\\s+(_:[_A-Z][-.:\\w]*)\\s*=\\s*+>
(?!\"[^"]*|'[^']*'))*\\s*/?>
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

刚好出现在结束的 </> 之前的 </?> 会允许这个正则式匹配起始和单体标签。把它删掉的话就只能用于匹配起始标签。只删除问号量词（把 </> 变为必需的），它就只能匹配单体标签。

### 结束标签

```
</(_:[_A-Z][-.:\\w]*)\\s*>
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

想知道如何避免匹配到在注释、CDATA 片段和 DOCTYPE 之内的标签，请参考随后的“跳过复杂的(X)HTML 和 XML 片段”小节。

### 跳过复杂的(X)HTML 和 XML 片段

当你尝试要在一个源文件或字符串中匹配 XML 风格的标签时，很多精力会花费到如何避免匹配到貌似标签的内容上，这些看起来像是标签的内容由于其所在位置或者其他上下文，使之不能被解释为标签。在本实例中我们所给的(X)HTML 和 XML 风格的正则表达式中都会通过限制元素名称的首字母来避免一些有问题的内容。有时

候你可能会期望能够更进一步要求标签满足(X)HTML 或 XML 语法规则。这样一种健壮的解决方案还会要求我们避免匹配在注释、脚本语言代码（其中可能会在数学操作中用到大于和小于符号）、XML 的 CDATA 片段和各种其他结构中出现的内容。要解决这个问题，可以首先查找这些可能会有问题的片段，然后只在这些匹配之外的内容中查找标签。

实例 3.18 中讲解了如何编写代码来查找另外一个正则表达式匹配之间的内容。它需要 2 个模式：一个内层正则式和一个外层正则式。前面所给的解决方案可以用作我们的内层正则式。这个方案可以把有问题的片段隐藏到内层正则式的视线范围之外，因此得到一个相对简单的解答。

**(X)HTML 的外层正则式。** 下面的正则式会匹配注释，以及元素 `<script>`、`<style>`、`<textarea>` 和 `<xmp>`<sup>1</sup>（包括它们的内容）：

```
<!--.*?--\s*>|<(script|style|textarea|xmp)\b(?:[^>"]|"[""]*"|<\br>[^']*')*?(?:/>|>.*?</\1\s*>)
```

正则选项：不区分大小写、点号匹配换行符  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

这个正则表达式读起来很可能并不是那么容易，因此我们在下面给出了添加了注释之后的宽松排列形式：

```
# 注释
<!-- .*? --\s*>
|
# 特殊元素及其内容
<( script | style | textarea | xmp )\b
 (?: [^>"]      # 没有引起来的字符
    | "[^"]*"   # 双引号引起的值
    | '[^']*'    # 单引号引起的值
  )*?
 (?: # 单体标签
    />
    | # 否则，把元素的内容和对应的结束标签包含进来
      > .*? </\1\s*>
  )

```

正则选项：不区分大小写、点号匹配换行符、宽松排列  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

因为在 JavaScript 中并不支持“点号匹配换行符”和“宽松排列”选项，所以上面的 2 个正则式在 JavaScript 中都无法正确执行。下面的正则式又回到了前面不易读懂的形式，并且把其中的点号替换为 `\s`，使之可以用于 JavaScript：

---

<sup>1</sup> `<xmp>` 是一个较少看到但是却得到广泛支持的元素。它和`<pre>`类似，会保留所有的空格，并且使用一种默认的等宽字体，但是它会更进一步把所有的内容（包括 HTML 标签在内）都显示为纯文本。在 HTML 3.2 中，`<xmp>`被标记为不推荐使用，而在 HTML 4 中已经被彻底删除。

```
<!--[\s\S]*?--\s*>|<(script|style|textarea|xmp)\b(?:[^>"]|"[^"]*"|←
'[^']*'|)?(?:/>|>[\s\S]*?</\1\s*>)
正则选项: 不区分大小写
正则流派: JavaScript
```

这些正则式给我们出了一个难题：因为它们会匹配到`<script>`、`<style>`、`<textarea>` 和 `<xmp>` 这些标签，所以这些标签就不会被第二个（内层）正则式匹配到，而我们想要查找的是所有的标签。然而，其实我们只需要添加一点点额外代码就可以对这些标签进行特殊的处理。这些（外层）正则式已经把标签名称捕获到了第 1 个向后引用中，因此可以用它来检查所匹配到的是一个注释还是标签（如果是标签，那么你还可以知道是哪个标签）。

**XML 的外层正则式。**这个正则式会匹配注释、CDATA 片段和 DOCTYPE。其中每种情形都会使用一个单独的模式来匹配，而这 3 种模式会使用多选操作符 `↪` 组合成一个正则式：

```
<!--.*?--\s*>|<!\[CDATA\[.*?\]]>|<!DOCTYPE\s(?:[^>"]|"[^"]*"|←
'[^']*'|<! (?:[^>"]|"[^"]*"|'[^\']*'|)[^>])*>)↪
正则选项: 不区分大小写、点号匹配换行符
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

下面我们给出它的宽松排列模式：

```
# 注释
<!-- .*? --\s*>
|
# CDATA 片段
<!\[CDATA\[ .*? ]>
|
# 文档类型声明
<!DOCTYPE\s
  (?: [^>"]| # 非特殊字符
    | "[^"]*"| # 双引号引起的值
    | '[^\']*'| # 单引号引起的值
    | <! (?:[^>"]|"[^"]*"|'[^\']*'|)[^>])* # 标记声明
  )*
>
正则选项: 不区分大小写、点号匹配换行符、宽松排列
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

下面是一个可以用于 JavaScript 的版本（其中不支持“点号匹配换行符”和宽松排列选项）：

```
<!--[\s\S]*?--\s*>|<!\[CDATA\[ [\s\S]*?\]]>|<!DOCTYPE\s(?:[^>"]|"[^"]*"|←
'[^']*'|<! (?:[^>"]|"[^"]*"|'[^\']*'|)[^>])*>)↪
正则选项: 不区分大小写
正则流派: JavaScript
```

# 变体

## 匹配合法的 HTML 4 标签

你偶尔可能会想要把查找范围限制到合法的 HTML 元素之上，特别是在非 HTML 文档中查找标签的时候，这时就需要额外小心，不要匹配到假阳性的标签。下面的正则式会只匹配 91 个合法的 HTML 4 元素。这个列表中没有包含非标准的 HTML，例如一些专有的标签 `<blink>`、`<bgsound>`、`<embed>` 和 `<nobr>`。它也没有包含在 XHTML 1.1 中的元素（XHTML 1.0 没有添加新的标签），或者是在 HTML 5 中计划添加的新元素：

```
</?(a|abbr|acronym|address|applet|area|b|base|basefont|bdo|big|blockquote|←
body|br|button|caption|center|cite|code|col|colgroup|dd|del|dfn|dir|div|←
d1|dt|em|fieldset|font|form|frame|frameset|h1|h2|h3|h4|h5|h6|head|hr|html|←
i|iframe|img|input|ins|isindex|kbd|label|legend|li|link|map|menu|meta|←
noframes|noscript|object|ol|optgroup|option|p|param|pre|q|s|samp|script|←
select|small|span|strike|strong|style|sub|sup|table|tbody|td|textarea|←
tfoot|th|thead|title|tr|tt|u|ul|var)\b(?:[^>"]|"[^"]*"|'[^']*')*>
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

可以通过减少由元字符 `\b` 分隔的选择分支，提高这个正则式的执行速度。因此我们会尽量使用字符类和可选的后缀。这些改动会显著减少正则引擎需要执行的回溯的数量。我们来看一个例子，假设正则引擎在目标文本中遇到了 `<0`。这不可能作为一个标签的开始，因为标签名称是不能以数字 0 开头的，但是在正则引擎把这种情况排除在匹配外之前，它必须检查所有 91 个选择分支中是否有以字面的 0 字符开头的情形。通过把选择分支的数目降到最少（可以作为标签开始的每个字母对应一个选择分支），我们就可以把在左尖括号之后需要进行尝试的步骤从 91 降低到 19。下面是这个正则式：

```
</?(a(?:bbr|cronym|ddress|pplet|rea)?|b(?:ase(?:font)?|do|ig|lockquote|←
ody|r|utton)?|c(?:aption|enter|ite|o(?:de|l(?:group)?)|d(?:[dlt]|el|fn|←
i[rv])|em|f(?:ieldset|o(?:nt|rm)|rame(?:set)?|h(?:[1-6r]|ead|tml)|←
i(?:frame|mg|n(?:put|s)|sindex)?|kbd|l(?:abel|egend|i(?:nk)?|m(?:ap|←
e(?:nu|ta))|no(?:frames|script)|o(?:bject|l|p(?:tgroup|ition))|p(?:aram|←
re)?|q|s(?:amp|cript|elect|mall|pan|t(?:rike|rong|yle)|u[bp])?|t(?:able|←
body|[dhrt]|extarea|foot|head|itle)|ul?|var)\b(?:[^>"]|"[^"]*"|'[^']*')*>
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

这读起来肯定会更难懂一些，下面的宽松排列形式稍好一些：

```
<
/? # 支持结束标签
( # 把标签名称捕获到第 1 个向后引用中
a(?:bbr|cronym|ddress|pplet|rea)?|
```

```

b(?:ase(?:font)?|do|ig|lockquote|ody|r|utton)?|
c(?:aption|enter|ite|o(?:de|l(?:group)?)?)|
d(?:[dl]t|el|fn|i[rv])|
em|
f(?:ieldset|o(?:nt|rm)|rame(?:set)?)|
h(?:[l-6r]|ead|tml)|
i(?:frame|mg|n(?:put|s)|sindex)?|
kbd|
l(?:abel|egend|i(?:nk)?)|
m(?:ap|e(?:nu|ta))|
no(?:frames|script)|
o(?:bject|l|p(?:tgroup|tion))|
p(?:aram|re)?|
q|
s(?:amp|cript|elect|mall|pan|t(?:rike|rong|yle)|u[bp])?|
t(?:able|body|[dhrt]|extarea|foot|head|title)|
ul?|
var

\b          # 不允许匹配部分名称
(?: [^>"']  # 除了 >、" 或 ' 之外的任意字符
| "[^"]*"  # 双引号引起来的属性值
| '[^']*'  # 单引号引起来的属性值
)*
>
正则选项：不区分大小写
正则流派：.NET、Java、PCRE、Perl、Python、Ruby

```

如果不介意其中使用太多空白，那么下面是同一个正则表达式的另外一种宽松排列形式，它读起来会更加容易：

```

<
/? # 支持 结束标签
( # 把标签名称捕获到第 1 个向后引用中

  a(?: bbr          #
    | cronym        #
    | ddress        #
    | pplet          #
    | rea            #
  )?|                                # 可选分组 (支持<a>)
  b(?: ase(?:font)?          # <base>, <basefont>
    | do             #
    | ig             #
    | lockquote     #
    | ody            #
    | r              #
    | utton          #
  )?|                                # 可选分组 (支持<b>)
  c(?: aption      #
    | enter          #

```

```

    | ite                      #
    | o (?:del(?:group)?) # <code>, <col>, <colgroup>
)
|
d (?: [dlt]                  # <dd>, <dl>, <dt>
    | el                      #
    | fn                      #
    | i[rv]                   # <dir>, <div>
)
|
em |                         #
|
f (?: ieldset               #
    | o (?:nt|rm)           # <font>, <form>
    | rame (?:set)?         # <frame>, <frameset>
)
|
h (?: [1-6r]                 # <h1>, <h2>, <h3>, <h4>, <h5>, <h6>, <hr>
    | ead                     #
    | tml                     #
)
|
i (?: frame                #
    | mg                      #
    | n (?:put|s)            # <input>, <ins>
    | sindex                  #
) ?|                         # 可选分组（支持<i>）
|
kbd |
|
l (?: abel                #
    | egend                  #
    | i (?:nk)?              # <li>, <link>
)
|
m (?: ap                  #
    | e (?:nulta)            # <menu>, <meta>
)
|
no (?: frames              #
    | script                  #
)
|
o (?: bject                #
    | l                      #
    | p (?:tgroup|tion)      # <optgroup>, <option>
)
|
p (?: aram                #
    | re                      #
) ?|                         # 可选分组（支持<p>）
|
q |                         #
|
s (?: amp                  #
    | cript                  #
    | elect                  #
    | mail                   #
    | pan                    #
    | t (?:rike|rong|yle)   # <strike>, <strong>, <style>
)

```

```

    | u[bp]                      # <sub>, <sup>
) ? |                         # 可选分组 (支持<s>)
t (?: able
| body
| [dhrt]
| extarea
| foot
| head
| itle
) |
ul? |                         # <u>, <ul>
var                           #

) \b                  # 不允许匹配部分名称
(?: [^>"']      # 除了 >、" 或 ' 之外的任意字符
| "[^"]*"      # 双引号引起来的属性值
| '[^']*'       # 单引号引起来的属性值
)*
>

```

正则选项: 不区分大小写  
 正则流派: .NET、Java、PCRE、Perl、Python、Ruby

如果你使用的是 XHTML，那么要注意，虽然 XHTML 1.0 没有添加新的标签，但它删除了如下 14 个标签:<applet>、<basefont>、<center>、<dir>、<font>、<frame>、<frameset>、<iframe>、<isindex>、<menu>、<noframes>、<s>、<strike> 和 <u>。

XHTML 1.1 保留了 XHTML 1.0 中的所有元素，并且添加了 6 个新元素（会都与亚洲语言的 ruby 文本有关）：<rb>、<rbc>、<rp>、<rt>、<rtc> 和 <ruby>。我们把创建专门用于匹配 XHTML 1.0 和 1.1 中的合法元素的正则表达式作为练习留给读者。

## 参见

匹配任意和所有的标签是有用的，但是通常你也可能会想要匹配某个或者某几个特定的标签；实例 8.2 中会讲解如何来完成这两个任务。

实例 8.4 中会介绍在合法 XML 元素和属性名称中可以使用的字符。

## 8.2 把标签<b>替换为<strong>

### 问题描述

你想要把所有的起始和结束的<b>标签都替换为相应的<strong>标签，同时保留已经存在的任意属性。

### 解决方案

这个正则表达式会匹配起始和结束的<b>标签，包括带属性的和不带属性的：

```
<(/?)b\b((?:[^>"]|"[^"]*"|'['])*)>
```

正则选项：不区分大小写

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

上面正则式的宽松排列模式：

```
<          #
  (/?)      #
  b \b      # 把可选的前导斜杠捕获到第 1 个向后引用中
  (         # 完整的标签名称，以及单词边界
    (?: [^>"] # 把所有属性与其他信息都捕获到第 2 个向后引用中
      | "[^"]*" # 除了 >、“ 或 ’ 之外的任意字符
      | '[']*" # 双引号引起的属性值
      )*
  )
>          #
```

正则选项：不区分大小写、宽松排列

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

要想在改变标签名称的同时保留所有的属性，可以使用下面的替代文本：

```
<$1strong$2>
替代文本流派：.NET、Java、JavaScript、Perl、PHP
```

```
<\1strong\2>
替代文本流派：Python、Ruby
```

如果想要在该过程中删除所有的属性，那么可以在替代文本中省略第 2 个向后引用：

```
<$1strong>
替代文本流派：.NET、Java、JavaScript、Perl、PHP

<\1strong>
替代文本流派：Python、Ruby
```

实例 3.15 中讲解了实现上述功能的代码。

## 讨论

前一个实例中讨论了如何匹配任意的 XML 风格标签的多种方式。以此为基础，本实例会直接关注如何查找一种特定类型的标签。我们将会以**<b>**和其替代**<strong>**作为例子，但你可以把它们替换成任意两种其他的标签名称。

这个正则式首先会匹配一个字面的 **<>** ——也就是任何标签的第一个字符。它接着会使用 **</?>** 来匹配在结束标签中会出现的一个正向斜杠，并把它放到了捕获括号中。把这个模式的结果（它可以是一个空串或正斜杠）捕获起来，让你可以很容易地在替代字符串中恢复正斜杠，而不必再借助任何的条件判断逻辑。

接下来，我们会匹配标签自身，也就是 `<b>`。可以在这里使用你想要的任意其他标签名称。我们在此使用了不区分大小写的选项，以确保同样能匹配到大写的 `B`。

人们经常容易忽略在标签名称之后紧跟的单词边界 (`\b`)，但它是这个正则式中最重要的成分之一。这个单词边界让我们可以只匹配标签 `<b>`，而不会匹配到 `<br>`、`<body>`、`<blockquote>` 或者任何其他以字母 “`b`” 开头的标签。我们也可以选择在名称之后匹配一个空白记号 (`\s`) 来解决这个问题，但是这样并不能用于不包含属性、从而在标签名称之后不跟随任何空白符号的标签。解决这个问题还是使用单词边界更为简单而且优雅。



### 提示

在处理 XML 和 XHTML 的时候，应当小心在命名空间中使用的冒号，以及 XML 名称中允许出现的连字符和其他字符，因为它们会创建一个单词边界。例如，上面的正则式会匹配到类似 `<b-sharp>` 的内容。如果你担心这种情况，那么可以使用顺序环视 `<(?= [\s/>])>` 来代替单词边界的作用。它能够得到相同的结果，使得我们不会匹配到部分的标签名称，但是它会更加可靠。

在标签名称之后，模式 `<((?:[^>"]|"[^"]*"|"[^']*')*)>` 被用来匹配在标签内直到结束的右尖括号之前的所有内容。把这个模式放到一个捕获分组中，可以很容易地把任意的属性和其他字符（例如单体标签中的拖尾斜杠）加到我们的替代字符串中去。在捕获括号之内的模式中重复了包含 3 个选择分支的一个非捕获分组。第一个选择分支 `<[^>"]>`，会匹配除了 `>`、`"` 和 `'` 之外的任意单个字符。剩余的两个选择分支会分别匹配单引号或双引号引起起来的整个字符串，这样会允许你匹配其中包含右尖括号的属性值，而不必担心正则式会把它当作标签的结束。

## 变体

### 替换一个标签列表

如果你想要匹配一个标签名称列表中的任意标签，那么需要做一个简单的改动。把你想要的所有标签名称都放到一个分组中，然后在它们之间使用多选操作符。把所有名称放到一个分组中会限制多选操作元字符 (`|`) 的作用范围。

下面的正则表达式会匹配起始和结束的 `<b>`、`<i>`、`<em>` 和 `<big>` 标签。随后给出的替代文本则会把它们都替代成相应的 `<strong>` 或 `</strong>` 标签，并且保持所有的属性：

```
<(/?) ([bi] | em | big) \b ((?:[^>"]|"[^"]*"|"[^']*')*)>  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面是这个正则表达式的宽松排列模式：

```
<          #
(/?)
  ([bi]|\em|big) \b      # 把可选的前导斜杠捕获到第 1 个向后引用中
(
  (?:
    [^>"]+
   | "[^"]*"
   | '[^']*'
  )*
)
>          #
正则选项：不区分大小写、宽松排列
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

我们使用了字符类 `<[bi]>` 同时匹配标签 `<b>` 和 `<i>`，而不是像我们处理`<em>`和`<big>`一样采用多选操作元字符 (`|`)。字符类的执行速度比多选结构要快许多，这是因为它们不需要回溯就可以完成任务。如果两个选项之间的区别只是单个字符，那么应当使用字符类。

我们还为标签名称添加了一个捕获分组，这样就使得匹配属性和其他内容的分组把它的匹配保存在了第 3 个向后引用中。虽然如果你只是要把所有匹配都替换为`<strong>`标签，就没有必要对标签名称采用向后应用，但是把标签名称保存到自己的向后引用中会有助于你在有必要的时候检查所匹配到的标签种类。

如果要在替换标签名称的同时保留所有的属性，只需要使用如下的替代文本：

```
<$1strong$3>
替代文本流派：.NET、Java、JavaScript、Perl、PHP

<\1strong\3>
替代文本流派：Python、Ruby
```

在替代字符串中把第 3 个向后引用去掉，你就可以在相同的处理过程中把匹配到的标签中的属性丢掉：

```
<$1strong>
替代文本流派：.NET、Java、JavaScript、Perl、PHP

<\1strong>
替代文本流派：Python、Ruby
```

## 参见

实例 8.1 讲解了如何匹配所有 XML 风格的标签，并且讨论了如何进行利弊权衡，其中包括如何容忍不合法的标记。

实例 8.3 与本实例正好相反，它会讲解如何匹配除了一个指定标签列表之外的所有标签。

## 8.3 删掉除<em>和<strong>之外的所有 XML 风格标签

### 问题描述

你想删除一个字符串中除了<em>和<strong>之外的所有标签。

另外一个单独的任务是，你不仅要去掉除了<em>和<strong>之外的所有标签，你还想要去掉包含属性的所有<em>和<strong>标签。

### 解决方案

这里正好可以利用否定型的顺序环视（参考实例 2.16 中的讲解）。把否定型顺序环视应用到这个问题之上，它可以让你匹配到貌似标签的内容，但是会排除掉在起始的<或</之后紧跟着某些特定单词的内容。如果你接着想把所有匹配都替换为一个空字符串（参考实例 3.14 中的讲解），那么保留下来的就只剩下之前列表中的标签。

#### 解决方案 1：匹配除了<em>和<strong>之外的所有标签

```
</?(?!(:em|strong)\b)[a-z](?:[^>"']|[\""]*|[^\']*)*>  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面是宽松排列模式：

```
< /? # 支持结束标签  
(?! # 否定型顺序环视  
  (: em | strong ) # 需要避免的标签列表  
  \b # 用来避免部分单词匹配的单词边界  
) #  
[a-z] # 标签名称的首字母必须是 a~z  
(?: [^>"'] # 除了 >、" 或 ' 之外的任意字符  
  | "[^"]*" # 双引号引起来的属性值  
  | '[^']*' # 单引号引起来的属性值  
)* #  
> #  
正则选项：不区分大小写、宽松排列  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

#### 解决方案 2：匹配除了<em>和<strong>，以及包含属性的所有标签

只需要一个很小的修改（把<\b>替换为<\s\*>），就可以使上面的正则式也能匹配到包含属性的任意<em>和<strong>标签：

```
</?(?!(:em|strong)\s*)[a-z](?:[^>"']|[\""]*|[^\']*)*>  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面我们依然给出了同一正则式的宽松排列形式：

```
< /?                      # 支持结束标签
(?!                     # 否定型顺序环视
  (?:: em | strong )    # 要避免匹配的标签列表
  \s* >                  # 只避开包含属性的标签
)
[a-z]                   # 标签名称的首字母必须是 a~z
(?: [^>"']              # 除了 >、" 或 ' 之外的任意字符
 | "[^"]*"             # 双引号引起来的属性值
 | '[^']*'              # 单引号引起来的属性值
)*                      #
>                      #

正则选项：不区分大小写、宽松排列
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

## 讨论

本实例中的正则表达式与在本章前面所讲的用来匹配 XML 风格标签的正则表达式拥有很多相同之处。除了添加了否定型顺序环视以避免匹配到某些标签，这些正则表达式几乎是同实例 8.1 中所给的“(X)HTML 标签（灵活方案）”中的正则式是等同的。另外一个主要区别是我们没有把标签名称捕获到第 1 个向后引用中。

下面仔细审视一下这个实例中引入的新内容。解决方案 1 永远不会匹配到<em>或<strong>标签，无论它们是否包含属性，但是会匹配到所有其他的标签。解决方案 2 能匹配到解决方案 1 中匹配到的所有标签，并且还能匹配到包含一个或多个属性的<em>和<strong>标签。表 8-2 中给出了一些示例目标字符串，用来对此加以解释。

表 8-2 几个目标字符串实例

目标字符串	解决方案 1	解决方案 2
<i>	匹配	匹配
</i>	匹配	匹配
<i style="font-size:500%; color:red;">	匹配	匹配
<em>	不能匹配	不能匹配
</em>	不能匹配	不能匹配
<em style="font-size:500%; color:red;">	不能匹配	匹配

因为这些正则式的最终目的是为了把所有匹配都替换为空字符串（也就是说把这些标签删掉），所以解决方案 2 更加不会滥用允许的<em>和<strong>标签来造成不可预期的格式或其他怪异的效果。



### 提示

到目前为止，本实例在介绍最后只剩下几种标签的情形时，故意避免了使用“白名单（whitelist）”这个单词，因为这个单词拥有安全的意味。要使用这个模式来限制使用人工注入的恶意 HTML 字符，会有许多种不同的方式。如果你担心会遇到恶意 HTML 和跨站点脚本（XSS）攻击的话，那么最好的选择是把所有的 <、> 和 & 字符都转变成它们对应的字符实体引用（<、> 和 &amp;），然后再把已知是安全的标签加进来（前提是它们不包含属性，或者只包含一个批准过的列表中的属性）。style 是一个不安全属性的例子，因为有些浏览器允许把脚本语言代码嵌入到你的 CSS 中。举例来说，要想在把 <、> 和 & 替换为字符实体引用之后把标签再加回来，可以先使用正则式 <<(/?)em>> 查找，再把所有匹配都替换为 «<\$1em>»（如果在 Python 和 Ruby 中，替换为«<\1em>»）。

## 变体

### 白名单对应的特定属性

考虑这样的新需求：你需要匹配除了<a>、<em>和<strong>之外的所有标签，但是有两个例外。你应当能够匹配包含除了 href 或 title 之外的其他属性的任意 <a> 标签，以及不包含任何属性的<em>和<strong>标签。

换句话说，除了在白名单上的标签（<a>、<em>和<strong>），你想要删除所有其他的标签。在白名单上的属性则只包括 href 和 title，并且它们只允许出现在<a>标签之内。如果在任意标签中出现没有进入白名单的属性，那么整个标签都应当被删除。

下面是用来完成这个任务的一个正则表达式，分别给出了不使用和使用宽松排列模式的形式：

<(?! (?:(em|strong)|a(?:\s+(?:href|title)\s\*=\s\*(?:"[^"]\*"|'[^\']\*'))\*))\s\*>)+  
[a-z](?:[^>"']|"[^"]\*"|'[^\']\*')\*>  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

```
< /?          # 支持结束标签
(?!          # 否定型顺序环视
  (?:
    em        # 不要匹配 <em>...
   | strong   # 或者 <strong>...
   | a        # 或者 <a>...
  (?:
    \s+        # 避免匹配 <a> 标签，如果其中只包含...
    (?:
      \s+        # href 和/或 title 属性
      (?:
        href|title
        \s*=\s*
        (?:"[^"]*"|'[^\']*')      # 双引号或单引号引起来的属性值
      )*
    )*
  )*
) *
```

```

)
\s* >          # 只有当这些标签包含上面列出的属性时...
)
[a-z]          # 标签名称首字母必须是 a~z
(?: [^>"']+
 | "[^"]*"#
 | '[^']*'#
)*
>
正则选项: 不区分大小写、宽松排列
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

```

至于使用这样复杂的正则表达式来解决问题是不是合乎情理，这可能已经快接近极限了。如果你的规则比这还要复杂，那么我们推荐你最好使用实例 3.11 或实例 3.16 中所讲解的代码来检查每个正则式所匹配到的标签值，然后再决定如何对它们进行处理（这可以依据标签名称、所包含的属性或者所需的其他任何信息来进行）。

## 参见

实例 8.1 讲解了如何匹配所有 XML 风格的标签，并且讨论了如何进行利弊权衡，其中包括如何容忍不合法的标记。

实例 8.2 与本实例正好相反，讲解如何匹配一个指定标签列表中的所有标签。

## 8.4 匹配 XML 名称

### 问题描述

你想要检查一个字符串是否是一个合法的 XML 名称（一种常见的语法结构）。对于在一个名称中可以出现的字符，XML 提供了精确的规则，并且会对元素、属性和实体名称、处理指令目标以及许多其他内容都使用同样的规则。名称的组成规则如下：必须由字母、下划线或冒号作为第一个字母，后面可以是任意组合的字母、数字、下划线、冒号、连字符和点号。这实际上只是一个近似的介绍，但是它已经相当接近实际情况了。实际上允许出现的字符要取决于你使用的 XML 版本。

如果为了更加精确而不惜采用更复杂的形式，那么另外一种可选的方式是把匹配合法名称的模式插入到其他处理 XML 的正则表达式中。

下面是一些合法名称的例子：

- **thing**
- **\_thing\_2\_**
- **:Российские-Вещь**
- **fantastic4:the.thing**
- **日本の物**

需要注意的是，这里允许出现非拉丁语的字母表，而且在最后一个示例中还包括了表意文字。类似的，在第一个字符之后会允许出现任意的 Unicode 数字，而不只是阿拉伯数字 0~9。

作为对比，下面给出了一些这个正则式不应当匹配的非法名称的例子：

- thing!
- thing with spaces
- .thing.with.a.dot.in.front
- -thingamajig
- 2nd\_thing

## 解决方案

与许多编程语言中的标识符一样，在 XML 名称中可以出现的字符会组成一个集合，而其中的一个子集可以用于第一个字符。对于 XML 1.0 第 4 版（以及更早版本）和 XML 1.1 和 1.0 第 5 版来说，这些字符列表有很大的区别。本质上来说，XML 1.1 名称可以使用在 1.0 第 4 版中允许的所有字符，另外再加上几乎 100 万个其他字符。然而，这些额外的字符仅仅是在 Unicode 字符表中的一个位置而已。大多数位置还没有被赋予任何实际字符，但是它们被预留以便 Unicode 字符数据库扩展时能做到与未来兼容。

为了简洁起见，在本实例中提到 XML 1.0 的时候指的是 XML 1.0 的第 1~4 版。当我们提到 XML 1.1 名称时，也包括了 XML 1.0 第 5 版的规则。第 5 版直到 2008 年 12 月底才正式成为 W3C 的推荐标准，这比 XML 1.1 要晚了几乎五年。



### 提示

在这个实例中所给出的正则式都使用了字符串开始和结束定位符（使用 `\^$` 或 `\A\Z`），这样就会造成你的目标字符串要么整体匹配，要么根本不能匹配。如果想要把这些模式嵌入到一个用来匹配 XML 元素的更长的正则表达式之中，就一定要记住把位于这些模式首尾的定位符去掉。  
关于定位符的讲解，请参考实例 2.5。

## XML 1.0 名称（近似方案）

```
\A[:_\p{L1}\p{Lu}\p{Lt}\p{Lo}\p{N1}][:_\-\.\p{L}\p{M}\p{Nd}\p{Nl}]*\Z  
正则选项：无  
正则流派：.NET、Java、PCRE、Perl、Ruby 1.9
```

PCRE 必须使用 Unicode 属性 (`\p{…}`) 的 UTF-8 支持进行编译才能正确执行。在 PHP 中，需要使用模式修饰符/u 打开 UTF-8 支持。

在 JavaScript、Python 或 Ruby 1.8 中不支持 Unicode 属性。随后要给出的 XML 1.1 名称的正则式并不依赖于 Unicode 属性，因此如果你使用的是这些编程语言之中的一种，那

么可以选择使用下面一个正则式。即使你的正则流派支持 Unicode 属性，最好也还是不要使用基于 XML 1.1 的解决方案，要想了解其中的详细原因，请参见本实例随后的“讨论”小节。

## XML 1.1 名称（精确方案）

下面给出的是同一个正则表达式的适用于不同流派的 3 种版本。前两个正则式之间的唯一区别是在模式开始和结尾使用的定位符不同。第 3 个版本中使用了 `\x{…}` 而不是 `\u` 来指定大于十六进制数 FF（对应于十进制数中的 255）的 Unicode 代码点。

```
\A[:_A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\u02FF\u0370-\u037D\u037F-\u1FFF\u200C-
\u200D\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCE\uFDF0-\uFFFD]-
[:_-.A-Za-z0-9\xB7\xC0-\xD6\xD8-\xF6\xF8-\u036F\u0370-\u037D\u037F-\u1FFF-
\u200C\u200D\u203F\u2040\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-
\uFDCE\uFDF0-\uFFFD]*\Z
```

正则选项：无

正则流派：.NET、Java、Python、Ruby 1.9

```
^[:_A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\u02FF\u0370-\u037D\u037F-\u1FFF\u200C-
\u200D\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCE\uFDF0-\uFFFD]-
[:_-.A-Za-z0-9\xB7\xC0-\xD6\xD8-\xF6\xF8-\u036F\u0370-\u037D\u037F-\u1FFF-
\u200C\u200D\u203F\u2040\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-
\uFDCE\uFDF0-\uFFFD]*$
```

正则选项：无（“点号匹配换行符”必须关掉）

正则流派：.NET、Java、JavaScript、Python

```
\A[:_A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\x{2FF}\x{370}-\x{37D}\x{37F}-\x{1FFF}-
\x{200C}\x{200D}\x{2070}-\x{218F}\x{2C00}-\x{2FEF}\x{3001}-\x{D7FF}-
\x{F900}-\x{FDCE}\x{FDF0}-\x{FFFD}][:_-.A-Za-z0-9\xB7\xC0-\xD6\xD8-\xF6-
\xF8-\x{36F}\x{370}-\x{37D}\x{37F}-\x{1FFF}\x{200C}\x{200D}\x{203F}-
\x{2040}\x{2070}-\x{218F}\x{2C00}-\x{2FEF}\x{3001}-\x{D7FF}\x{F900}-
\x{FDCE}\x{FDF0}-\x{FFFD}]*\Z
```

正则选项：无

正则流派：PCRE、Perl

PCRE 必须使用支持元字符序列 `\x{…}` 的 UTF-8 支持进行编译才能正确使用大于十六进制数 FF 的值。在 PHP 中，需要使用模式修饰符 `/u` 来打开 UTF-8 支持。

Ruby 1.8 并不支持 Unicode 正则表达式，如果需要，可以参考本实例随后的“变体”小节中所给的一个精确度稍低的替代解决方案。

虽然我们声称这些正则表达式是完全遵守 XML 1.1 名称规则的，实际上这只对小于 16bit 宽度的字符（位置是 0x0~0xFFFF）是正确的。在名称首字母之后，XML 1.1 中另外还支持使用 0x10000~0xEFFFF 之间的 917503 个代码点。然而，只有 PCRE、Perl 和 Python 中才会支持访问超过 0xFFFF 的代码点，而且你也不大可能会遇到这个范围之内的字符（一个原因是，绝大多数这些位置都还没有分配任何字符）。如果需要添加对这些额外代码点的支持，在 PCRE 和 Perl 中，你可以在第 2 个字符类的结尾添加

`\x{10000}-\x{FFFF}`，在 Python 中可以添加`\u00010000-\u000EFFFF`（注意要使用大写的 U，其后必须跟着 8 位的十六进制数字）。但是即使不添加这样大的范围，XML 1.1 名称字符列表已经要比 XML 1.0 的列表大很多了。

## 讨论

因为在本章中的许多正则表达式要处理的都是匹配 XML 元素，本实例的目标是当你想要对标签和属性名称中能匹配具体哪些内容感兴趣的时候，为读者提供更为完整的讨论。在书中其他地方，我们一般只会使用更简单、从而不是这么精确的模式，主要是为了可读性和效率的考虑。

因此，我们下面再来更深入地挖掘这些模式背后的规则。

### XML 1.0 名称

XML 1.0 规范在它的名称规则中使用的是白名单的方法，明确列出了它允许使用的所有字符。名称首字符可以是一个冒号 (:)、下划线 (\_) 或者在如下 Unicode 类别中的几乎任何字符：

- 小写字母 (Ll)
- 大写字母 (Lu)
- 标题大写 (Titlecase) 字母 (Lt)
- 不区分大小写的字母 (Lo)
- 字母数字 (Nl)

在首字母之后，除了上面介绍的字符之外，还可以使用连字符 (-)、点号 (.) 和在下列类别中的任意字符：

- 标记 (M)，其中包括了 3 个子类：非间距标记 (Mn)、间距组合标记 (Mc) 和封闭标记 (Me)。
- 修饰符字母 (Lm)
- 十进制数字 (Nd)

根据这些规则，我们可以得到在本实例的“解决方案”小节中所给的正则表达式。下面我们给出了这个正则表达式的宽松排列形式：

```
\A                                         # 字符串开始
[:_]\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nl}]    # 名称首字母
[:_\-. \p{L}\p{M}\p{Nd}\p{Nl}]*          # 名称字母 (0 个或多个)
\Z                                         # 字符串结束

正则选项: 宽松排列
正则流派: .NET、Java、PCRE、Perl、Ruby 1.9
```

同样需要注意 PCRE 必须使用 UTF-8 支持进行编译。在 PHP 中，需要使用模式修饰符 `/u` 来打开 UTF-8 支持。

注意在第 2 个字符类中，所有这些字母子类（Ll、Lu、Lt、Lo 和 Lm）被合起来表示为其基类 `\p{L}`。

在早些时候，我们提到过这里描述的规则是近似的。这有两个原因。首先，XML 1.0 规范（记住我们在这里讲的不包括它的第 5 版）中列出了不遵守这些规则的一些例外。其次，XML 1.0 字符列表是从 Unicode 2.0 中导出的，而后者是早在 1996 年发布的。Unicode 标准的更新版本中添加了一些新的文字，其中的字符在 XML 1.0 规则中是不支持的。然而，如果把正则表达式同你的正则引擎使用的 Unicode 版本之间分开来，从而可以把匹配限制到 Unicode 2.0 的字符集之上，就会把这个模式变成一个包含几百个范围和代码点的畸形正则表达式，它会占据一整页的内容。如果你的确想要创建这样一个怪物，那么可以参考 XML 1.0 第 4 版(<http://www.w3.org/TR/2006/REC-xml-20060816>) 中第 2.3 节“常见的语法结构”以及附录 B “字符类”。

如果我们只考虑把这个正则式用于某些特定正则流派，那么我们还可以按照下面的方式来简化这个正则式。

Perl 和 PCRE 允许把小写字母（Ll）、大写字母（Lu）和标题大写字母（Lt）子类合并成为一个特殊的“带大小写的字母”（L&）类别。这些正则流派同样允许省略在转义序列 `\p{…}` 中的花括号，前提是其中只包含一个字符。在下面这个正则式中我们利用了这个特性，因此可以使用 `\p{L}\p{M}` 来替代 `\p{L}\p{M}`：

```
\A[:_ \p{L&} \p{Lo} \p{Nl}] [:_ \-. \p{L}\p{M}\p{Nd} \p{Nl}] * \Z  
正则选项：无  
正则流派：PCRE、Perl
```

.NET 支持字符类差，在第一个字符类中，我们用它从 L 中减去了 Lm 子类，这样就不需要一一列出所有其他的字母子类：

```
\A[:_ \p{L} \p{Nl} - [\p{Lm}]] [:_ \-. \p{L} \p{M} \p{Nd} \p{Nl}] * \Z  
正则选项：无  
正则流派：.NET
```

同 PCRE 和 Perl 一样，Java 也允许省略掉单字母 Unicode 类别两边的花括号。下面的正则式还利用了 Java 中更为复杂的字符类减法（通过否定字符类的交集来实现），把 Lm 子类从字母类 L 中减掉：

```
\A[:_ \p{L}\p{Nl} && [^\p{Lm}]] [:_ \-. \p{L}\p{M}\p{Nd} \p{Nl}] * \Z  
正则选项：无  
正则流派：Java
```

JavaScript、Python 和 Ruby 1.0 根本不支持 Unicode 类别。Ruby 1.9 中也不包含前面讲到的花哨功能，但是它会支持我们在本实例的“解决方案”中给出的正则式的更加轻便的版本。

## XML 1.1 名称

XML 1.0 犯了个错误，把自己和 Unicode 2.0 绑到了一起。Unicode 标准的随后版本中添加了对更多字符的支持，其中有些来自于之前根本没有考虑过的文字（例如切洛基语、埃塞俄比亚语和蒙古语）。因为 XML 希望把自己当作一种统一的格式，因此它试图在 XML 1.1 和 1.0 第 5 版中改正这个问题。在这些更新版本中，对于名称中可以使用的字符从白名单的做法改成了黑名单，这样就可以不仅能够支持在 Unicode 2.0 之后添加的字符，同样还可以支持未来添加的字符。

这种新的策略支持没有明确声明禁止的内容，这样就提高了未来兼容性，而且它还使得我们更加容易做到精确遵守这些规则。这也就是为什么 XML 1.1 名称的正则式被标记为精确方案，而 XML 1.0 的正则式却是近似方案。

## 变体

在本章的有些实例中（例如实例 8.1），用来处理 XML 名称的模式片段，要么几乎不做任何限制，要么就会禁止某些事实上合法的外国文字与其他字符。这样做的目的是为了把问题变得简单。然而，如果想要在允许外国文字的同时还能提供基本的限制（而且你也不需要在本实例内前面正则式中精确的名称验证），那么下面的几个正则式可能会比较适合你的需要。



### 提示

在下面的正则式中，我们省略了字符串开始和结束的定位符，因为这些正则式并不打算单独使用，而是会用作更长模式的一部分。

下面第一个正则式会简单地避免匹配在 XML 标签中会被用作分隔符和分界符的字符，并且还会进一步避免匹配到第一个字符是数字的情形：

```
[^\d\s''/<=>] [^\s''/<=>]*
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

下面是另外一个更短的形式，它能达到相同的目的。这里我们不再使用两个单独的字符类，而是采用了否定型顺序环视来禁止数字作为首字符。虽然在字符类之后的量词会允许正则式匹配无限多个字符，但是前面的限制只会应用到第一个匹配到的字符：

```
(?!\\d) [^\s''/<=>]+
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 参见

XML 1.1 规范的编辑者之一的 John Cowan 解释了在 XML 1.1 中哪些字符是被禁止的，

以及为什么会这样做。读者可以从他的博客中找到相关内容：<http://recycledknowledge.blogspot.com/2008/02/which-characters-are-excluded-in-xml.html>。

文档“XML 1.0 第 5 版改动的背景说明”([http://www.w3.org/XML/2008/02/xml10\\_5th\\_edition\\_background.html](http://www.w3.org/XML/2008/02/xml10_5th_edition_background.html)) 中讨论了把 XML 1.1 的名称规则向后移植到 XML 1.0 第 5 版中的原因。

## 8.5 添加< p >和< br >标签将纯文本转换为 HTML

### 问题描述

给定一个纯文本字符串，例如一个通过表单提交的多行值，你想要把它转换成一个 HTML 片段，从而可以在网页上显示。由两个连续换行符分隔的段落应该用< p >……</p>包起来。而其他的换行则应当被替换为标签< br >。

### 解决方案

这个问题可以通过 4 个简单的步骤来解决。在大多数编程语言中，只有中间的 2 个步骤才会得益于正则表达式。

#### 步骤 1：把 HTML 特殊字符替换为字符实体引用

当我们把纯文本转换为 HTML 的时候，第一步需要把 3 个特殊的 HTML 字符 &、< 和 > 替换成相应的字符实体引用（参见表 8-3）。否则，所得到的标记在网页浏览器中显示的时候就会得到意想不到的结果。

表 8-3 HTML 特殊字符替换

查 找	替 换 为
<&>	<&gt;>
<<	<&lt;>
>>	>&gt;>

必须首先替换 & 符号，因为在替换其他的字符实体引用的时候还会向目标字符串中添加额外的 & 符号。

#### 步骤 2：把所有换行符替换为 <br>

查找：

\r\n?|\n

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

\R

正则选项: 无

正则流派: PCRE 7、Perl 5.10

替换为:

<br>

替代文本流派: .NET、Java、JavaScript、Perl、PHP、Python、Ruby

### 步骤 3: 把连续两个<br>标签替换为</p><p>

查找:

<br>\s\*<br>

正则选项: 无

正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

替换为:

</p><p>

替代文本流派: .NET、Java、JavaScript、Perl、PHP、Python、Ruby

### 步骤 4: 把整个字符串用<p>…</p>包起来

这一步只需要简单的字符串连接操作，因此不需要用到正则表达式。

## JavaScript 示例

为了把所有这 4 个步骤连在一起，我们还创建了一个 JavaScript 函数 `html_from_plaintext`。这个函数会接受一个字符串作为参数，使用我们刚刚讲解的步骤进行处理，然后返回新的 HTML 字符串：

```
function html_from_plaintext (subject) {
    // step 1 (plain text searches)
    subject = subject.replace(/&/g, "&amp;");
                    replace(/</g, "&lt;");
                    replace(/>/g, "&gt;");

    // step 2
    subject = subject.replace(/\r\n?|\n/g, "<br>");

    // step 3
    subject = subject.replace(/<br>\s*<br>/g, "</p><p>");

    // step 4
    subject = "<p>" + subject + "</p>";
    return subject;
}
/*
html_from_plaintext("Test.")          -> "<p>Test.</p>"
html_from_plaintext("Test.\n")         -> "<p>Test.<br></p>"
html_from_plaintext("Test.\n\n")        -> "<p>Test.</p><p></p>"
```

```
html_from_plaintext("Test1.\nTest2.")    -> "<p>Test1.<br>Test2.</p>"  
html_from_plaintext("Test1.\n\nTest2.")  -> "<p>Test1.</p><p>Test2.</p>"  
html_from_plaintext("< AT&T >")       -> "<p>< AT&amp;T ></p>"  
*/
```

在上面的代码片段包含了几个例子，用来展示把这个函数应用到不同字符串之上时的输出。如果你对 JavaScript 不是很熟悉，那么要注意在每个正则式字面量之后添加的修饰符/g 会导致 replace 方法替换该模式的所有匹配，而不只是替换第一个匹配。在这个示例目标字符串中的元字符序列\n会在一个 JavaScript 字符串字面量中添加一个换行字符（ASCII 位置 0x0A）。

## 讨论

### 步骤 1：把 HTML 特殊字符替换为字符实体引用

完成本步骤的最简单方式是使用 3 个单独的查找和替换操作（参考前面的表 8-3 中所列出的替换字符串）。JavaScript 总是会使用正则表达式进行全局的查找和替换操作，但是在其他编程语言中，你通常可以通过简单的纯文本替换来获得更好的性能。

### 步骤 2：把所有换行符替换为<br>

在这一步中，我们使用正则表达式 `\r\n?|\n` 找到遵守 Windows/MS-DOS (CRLF)、Unix/Linux/OS X (LF) 和老版本的 Mac OS (CR) 所约定的换行序列。Perl 5.10 和 PCRE 7 用户可以使用专门的记号 `\R`（注意这里用的是大写的 R），而不用匹配上面所列的或者其他换行序列。

在下一步中添加段落标签之前，把所有换行符都替换为<br>可以降低整体上的复杂度，因为这样就可以在随后的替换中在</p><p>标签之间添加空白。这样做可以帮助你维护 HTML 代码的可读性，不会把它们都堆在一起。

如果你更喜欢使用 XHTML 风格的单体标签，那么可以使用«<br●/>»来代替«<br>»作为替代字符串。你同样还需要修改第 3 步中的正则表达式来反映这个改动。

### 步骤 3：把连续两个<br>标签替换为</p><p>

如果看到有两个连续的换行，那么就意味着遇到了一段的结束和下一段的开始，因此在这个步骤中，我们的替代文本是一个结束的标签</p>，其后紧跟一个起始标签<p>。如果在目标文本中只包含一个段落（也就是说，不会出现连续的两个换行），那么就不会进行任何替换。步骤 2 中已经把几种不同的换行约定进行了替换（现在只剩下了<br>标签），因此在这一步中，我们只需采用纯文本进行替换。然而，在这里使用一个正则式可以更进一步忽略出现在换行符之间的空白。而在一个 HTML 文档中也不会对任意额外的空格字符进行显示。

如果要生成的是 XHTML 代码，因此需要把换行替换为«<br•/>»而不是«<br>»，那么你就需要把这一步的正则式也调整为：<<br•/>\s\*<br•/>>。

#### 步骤 4：把整个字符串用<p>…</p>包起来

步骤 3 只是简单地在段落之间添加了标记。现在你还需要在整个目标字符串的开始添加一个<p>标签，并且在字符串的最后添加一个结束的</p>标签。这样我们的任务就完成了，不管在文本中包含的是 1 个还是 100 个段落都一样。

#### 参见

实例 4.10 讲解了关于 Perl 和 PCRE 中的\R 标记的更多信息，并且讲解了如何手动来匹配\R 所支持的其他奇形怪状的换行符号。

## 8.6 在 XML 风格的标签中查找某个特定属性

### 问题描述

你想要在一个(X)HTML 或 XML 文件中查找包含一个特定属性（例如 id）的标签。

本实例会讲解这个问题的几种变体。假设你想要使用不同的正则表达式来匹配下面一种类型的字符串：

- 包含 id 属性的标签；
- 包含 id 属性的<div>标签；
- 包含一个 id 属性且其值为 my-id 的标签；
- 在其 class 属性值中包含 my-class 的标签（类之间使用空格来分开）。

### 解决方案

#### 包含 id 属性的标签（速成方案）

如果你想要在编辑器中进行快速的查找，并且编辑器允许对结果进行预览检查，那么可能使用下面这个（过于简化的）正则式就足够了：

```
<[^>]+\\sid\\b[^>]*>  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面是上面正则式的宽松排列形式：

```
<          # 标签开始  
[>]+      # 标签名称、属性等  
\\s id \\b  # 要查找的属性名称，整字匹配
```

```
[^>]*          # 标签的剩余部分，包括 id 属性的值
>              # 标签结束
正则选项：不区分大小写、宽松排列
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

## 包含 id 属性的标签（更可靠的版本）

与上面所给的正则式不同，下面这个正则式虽然解决的是相同的问题，但是会支持引号引起的包含字面的 > 字符的属性值，并且不会匹配到只是在其中某个属性值中包含单词 id 的标签：

```
<(?:[^>"']|"[^"]*"|'[^']*')+?\sid\s*=\s*("[^"]*"|'[^']*')↵
(?:[^>"']|"[^"]*"|'[^']*')*>
正则选项：不区分大小写
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面是宽松排列的形式：

```
<          #
(?: [^>"']
 | "[^"]*"#
 | '[^']*'#
)+?
\s id          # 要查找的目标属性名称，整字查找
\s* = \s*      # 属性名称-值之间的分隔符
( "[^"]*" | '[^']*' )# 把属性值捕获到第 1 个向后引用中
(?: [^>"']
 | "[^"]*"#
 | '[^']*'#
)*          #
>          #
正则选项：不区分大小写、宽松排列
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

这个正则式会把 id 属性的值和包围它的引号一起捕获到第 1 个向后引用中。这样就可以允许你在正则式之外的代码中或者替代字符串中使用这个值。如果不需要重复使用该值，那么你可以使用一个非捕获分组，或者把整个的`\s*=\s*("[^"]*"|'[^']*')` 序列替换为`\b`。这个正则式的剩余部分会接下来匹配 id 属性的值。

## 包含 id 属性的<div>标签

要查找一个特定的标签类型，你需要把它的名称添加到正则式的开始，并且再对上面的正则式做一些较小的修改。在下面的例子中，我们在起始的<>之后添加了<div\s>。<\s>（空白）记号确保不会匹配到名称中以“div”3个字母开头的标签。我们知道在标签名称之后会有一个空格字符，因为要查找的标签中至少会包含一个属性（id）。另外，<+?\sid>序列被替换成了<\*?\bid>，从而当时 id 是标签内的第一个属性，而且在标签名称之后没有其他分隔字符（除了最开始的空格），那么也可以使

用下面这个正则式：

```
<div\s(?:[^>"']|"[^"]*"|'[^\']*')*?\bid\s*=\s*(\"[^\"]*|[^\']*')  
(?:[^>"']|"[^"]*"|'[^\']*')*>  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面是同一个正则式的宽松排列形式：

```
<div \s # 标签名称和随后的空白字符  
(?: [^>"'] # 标签和属性名称等  
| "[^"]*" # ...以及引号引起的属性值  
| '[^\']*'  
) *? #  
\b id # 要查找的目标属性名称，整字匹配  
\s* = \s* # 属性名称-值之间的分隔符  
( "[^"]*" | '[^\']*' ) # 把属性值捕获到第1个向后引用中  
(?: [^>"'] # 任意剩余的字符  
| "[^"]*" # ...以及被引起的属性值  
| '[^\']*'  
) * #  
> #  
正则选项：不区分大小写、宽松排列  
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

## 包含 id 属性且其值为“my-id”的标签

与前面题为“包含 id 属性的标签（更可靠的版本）”中的正则式相比，这次我们去掉了在 id 属性值两边的捕获分组，这是因为已经在之前就知道了它的值。具体来说，子模式<("[^"]\*"|[^\']\*")> 被替换为 <(?:"my-id"|"my-id")>：

```
<(?:[>"']|"[^"]*"|'[^\']*')+?\sid\s*=\s*(?:"my-id"|"my-id')  
(?:[>"']|"[^"]*"|'[^\']*')*>  
正则选项：不区分大小写  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面是宽松排列的版本：

```
< #  
(?: [^>"'] # 标签和属性名称等  
| "[^"]*" # ...以及引起的属性值  
| '[^\']*'  
) +? #  
\s id # 要查找的目标属性名称，整字匹配  
\s* = \s* # 属性名称-值之间的分隔符  
(?: "my-id" # 目标属性值  
| 'my-id' ) # ...用单引号或双引号引起  
(?: [^>"'] # 任意剩余的字符  
| "[^"]*" # ...以及引起的属性值  
| '[^\']*'  
) #
```

```
) *          #
>          #
    正则选项: 不区分大小写、宽松排列
    正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

再回来看一下其中的子模式 «(?:“my-id”|‘my-id’»，如果不想重复使用“my-id”，可以把它替换为 «([""]my-id\1»（这样会牺牲一些效率）。它使用了一个捕获分组和一个向后引用用来确保属性值是以相同类型的引号开始和结束的。

## 在其 class 属性值中包含“my-class”的标签

如果说上一个正则表达式还可以接受，那么下面这一个就已经明显超出了单个正则表达式的能力极限。把这个过程划分为多个正则式会更容易一些，因此我们会把这个查找分成 3 个部分。第一个正则式会匹配标签，下一个会在其中查找 class 属性（并把它的值保存到一个向后引用中），最后一个正则式会在它的值中查找 my-class。

查找标签的正则表达式：

```
<(?:[^>"']|[^\"]*|[^\']*')+>
    正则选项: 无
    正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```



### 提示

实例 8.1 中专门讲解了如何匹配 XML 风格的标签。它讲解了上面这个正则式的工作原理，并且给出了包含不同程度复杂性和准确性的多个可选解决方案。

接下来，可以使用在实例 3.13 中的代码，用下面的正则表达式在上面的每个匹配中查找 class 属性：

```
^(?:[^>"']|[^\"]*|[^\']*')+?\sclass\s*=|\s*(?:\"([^\"]*)\"|'([^\']*')')
    正则选项: 不区分大小写
    正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

这会把整个 class 属性值根据它两边的引号类型捕获到第 1 个或第 2 个向后引用中。在 class 属性之前的所有内容可以使用 «^(?:[^>"']|[^\"]\*|[^\']\*')+?» 匹配，它会一步匹配到引起来的值，而不会匹配到另外一个属性值之内的“class”单词。在该模式的右边，只要我们到达了 class 属性的值的结尾，匹配就会马上结束。在此之后的所有内容都和我们的查找无关，因此也就完全没有必要一路匹配到你正在查找的标签的结尾。

在正则式开始处的脱字符会把它定位到目标字符串的开始。这并不会改变匹配到的内容，但是一旦使用了它，那么如果正则引擎无法在字符串开始找到匹配，它就不会继续在随后的每个字符位置进行尝试（这种匹配尝试注定是要失败的）。

最后，如果前面两个正则式都匹配成功，那么你还需要使用下面的模式来在第二个正则式匹配到的第 1 个或第 2 个向后引用中进行查找：

```
(?:^|\s)my-class(?:\s|$)
```

正则选项：无

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

因为类之间是由空格分隔开的，所以 my-class 的两边必须是空格或者什么都没有。如果不是因为在类名称中可以包含连字符，你甚至可以使用单词边界记号来替代这里的两个非捕获分组。然而，连字符会创建单词边界，因此 `\bmy-class\b` 会在 not-my-class 中产生匹配。

## 讨论

本实例中的“解决方案”小节已经讲解了这些正则表达式的详细工作原理，因此我们在这里就不再重复了。要记住的是正则表达式通常并不是标记查找问题的理想解决方案，特别是用来解决这个实例中讲解的复杂情况的时候。在使用这些正则表达式之前，读者需要考虑是否采用其他的解决方案可能会更好，例如 XPath、SAZX 分析器或者 DOM。在这里我们之所以讲解这些正则式，是因为人们有时候也会试着采用这种形式，但是这并不意味着我们没有提醒你。我们希望这至少会有助于展示与标记查找有关的一些问题，并且会帮助读者避免使用甚至更为幼稚的解决方案。

## 参见

实例 8.7 从概念上正好和本实例相反，它会查找不包含某个特定属性的标签。

## 8.7 向不包含 cellspacing 属性的 <table>标签中添加该属性

### 问题描述

你想要查找一个(X)HTML 文件，并且向其中没有包含 cellspacing 属性的所有表格中都添加 cellspacing="0"。

本实例会作为向 XML 风格的标签中添加不存在的属性的一个示例。你可以把它替换为想要的任意标签和属性名称。

### 解决方案

#### 正则式 1：简单的解决方案

你可以使用否定型顺序环视来匹配不包含 cellspacing 单词的<table>标签，如下所示：

```
<table\b(?![^>]*?\s cellpadding\b) ([^>]*)>
正则选项: 不区分大小写。
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面是同一个正则式的宽松排列版本:

```
<table \b          # 匹配 "<table", 后面跟着一个单词边界
(?!              # 判断这里不能匹配下面的正则式
 [^>]          # 匹配除了 ">" 之外的任意字符...
 *?             # 0 次或多次, 匹配尽量少次 (懒惰量词)
 \s cellpadding \b # 整字匹配 "cellspacing"
)
(
 [^>]          # 把下面正则式捕获到第 1 个向后引用中
 *              # 匹配除了 ">" 之外的任意字符...
 ?              # 0 次或多次, 匹配尽量多次 (贪心量词)
)
>              # 匹配一个字面的 ">" 作为标签的结束
正则选项: 不区分大小写
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

## 正则式 2: 更可靠的解决方案

下面的正则表达式会把前面最简单解决方案中的否定型字符类 <[^>]> 的两个实例都替换为<(?:[^>"]|"[^"]\*"|'[^']\*')>。这会从两个方面来提高这个正则表达式的可靠性。首先，它添加了对引号引起的，其中包含字面的“>”字符的属性值的支持。其次，它确保了我们不会排除掉只是在一个属性值中包含了单词“cellspacing”的标签。

下面是包含了我们所说的这些改动之后的正则式:

```
<table\b(?!(?:[^>"]|"[^"]*"|'[^']*')*?\s cellpadding\b)-
((?:[^>"]|"[^"]*"|'[^']*')*)>
正则选项: 不区分大小写
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

这里是它的宽松排列版本:

```
<table \b          # 匹配 "<table", 之后跟着一个单词边界
(?!              # 判断在这里不能匹配下面的正则式
 (?: [^>"]
 | "[^"]*"      # 匹配除了 >、" 或 ' 之外的任意字符
 | '[^']*'
 )*?
 \s cellpadding \b # 整字匹配 "cellspacing"
)
(
 (?: [^>"]
 | "[^"]*"      # 把下面的正则式捕获到第 1 个向后引用中
 | '[^']*'
 # 匹配除了 >、" 或 ' 之外的任意字符
 # 或者, 一个双引号引起的值
 # 或者, 一个单引号引起的值
)*

```

```
| '[^']*'          # 或者，一个单引号引起的值
)*                 # 0 次或多次，匹配尽量多次（贪心量词）
)
>                 #
正则选项：不区分大小写
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

## 插入新的属性

本实例中所给的所有正则式都可以使用相同的替代字符串，因为这些正则式都会把在匹配到的<table>标签（如果存在）中的属性捕获到第 1 个向后引用中。这会允许你把这些属性作为替代字符串的一部分放回去，同时可以添加新的 cellspacing 属性。下面是所需的替代字符串：

```
<table●cellspacing="0"$1>
替代文本流派：.NET、Java、JavaScript、Perl、PHP

<table●cellspacing="0"\1>
替代文本流派：Python、Ruby
```

实例 3.15 中展示了如何在替代字符串中使用向后引用来执行替换。

## 讨论

为了详细解释这些正则式如何工作，我们先来对最简单的解答做一个分解。你可以看到其中包含了 4 个逻辑组成部分。

第一个部分 <<table\b> 会先匹配字面字符串 <table，然后是一个单词边界 (\b)。单词边界会防止匹配到仅仅以“table”开始的标签名称。虽然当我们处理(X)HTML 的时候，这样做显得没有必要（因为其中并不存在诸如“tablet”、“tableau”、或“tablespoon”这样的元素名称），然而这是我们推荐采用的形式，而且它还会帮助你在用它查找其他标签时避免可能出现的错误。

该正则式的第二个部分 <(?![^>]\*?\s+cellspacing\b)> 是一个否定型顺序环视。它并不会在匹配的时候消费任何文本，但是如果在起始标签中的任何地方出现了单词 cellspacing，那么它会让匹配尝试失败。因为要把 cellspacing 属性添加到所有匹配中，因此我们并不想匹配到已经包含它的标签。

由于顺序环视会在匹配尝试中从当前位置向前看，所以其中使用了前导的 <[^>]\*?>，使之可以尽量搜索更多的内容，直到被认为是标签结束的地方为止（也就是第一次出现 > 的地方）。顺序环视子模式中的剩余部分 (<\s+cellspacing\b>) 会简单以整字形式匹配字面的字符串 cellspacing。因为在属性名称与标签名称或者前面的属性之间总是使用空格作为分隔，所以我们会匹配一个前导的空格字符 (<\s>)。我们会匹配一个拖尾的单词边界，而不是匹配另外一个空格字符，这是因为单词边界能够帮助我们满足整字匹

配 `cellspacing` 的目的，而且即使该属性没有取值，或者该属性名称之后紧跟着一个等号也可以正常工作。

接下来，我们来看该正则式的第三个部分：`<([>]*>)`。这是一个否定型的字符类加上一个“0 次或多次”量词，它们都被包围在一个捕获分组中。把这部分匹配捕获起来允许你很容易地把每个匹配到的标签作为替代字符串的一部分放回去。而且与否定型的顺序环视不一样，这部分实际上会把标签内的属性添加到该正则式匹配到的字符串中。

最后，正则式会匹配字面字符 `>>` 来结束这个标签。

第二个正则式，也就是被称作是更可靠版本的那个，与我们刚刚讲解的正则式工作原理是一模一样的，唯一的区别是两个否定型字符类`[>]`的实例都被替换成了`<(?:[^>"]|"[^"]*"|[^"]*)>`。这个较长的模式会在一步之中就略去双引号和单引号引起来的属性值。

至于替代字符串，它们可以用于这两个正则式，把每个匹配到的标签都替换成一个包含 `cellspacing` 作为第一个属性的新标签，随后是在原来标签中出现的任意属性（第 1 个向后引用）。

## 参见

实例 8.6 从概念上正好和本实例相反，它会查找包含某个特定属性的标签。

## 8.8 删除 XML 风格的注释

### 问题描述

你想要删除一个(X)HTML 或 XML 文档中的注释。例如，在把一个网页传递给 web 浏览器之前，你想要删除其中的注释，从而可以减少该页面的文件大小，并因此降低那些使用低速 Internet 连接的用户的加载时间。

### 解决方案

因为有了懒惰量词可以使用，所以查找注释并不是一件很困难的工作。下面是完成这个任务的正则表达式：

```
<!--.*?-->
正则选项：点号匹配换行符
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

这个正则式很直接。然而，我们知道在 JavaScript 中并不包括一个“点号匹配换行符”的选项，这就意味着你需要把点号替换为一个包含所有字符的字符类，从而这个正则

表达式可以匹配跨多行的注释。下面是一个在 JavaScript 中可用的版本：

```
<!--[\s\S]*?-->
正则选项: 无
正则流派: JavaScript
```

要想删除注释，只需要把所有匹配都替换为空字符串（也就是替换为空）。实例 3.14 中可以找到用来替换一个正则式的所有匹配的代码。

## 讨论

### 工作原理

在这个正则表达式的开始和结束处分别是字面的字符序列<!--> 和 <-->。因为这些字符在正则式语法中都不是特殊字符（除了在字符类中连字符会创建范围），因此它们并不需要进行转义。这样我们就只需要认真来解释一下位于正则式的中间的 <\*?> 或 <[\s\S]\*?>。

因为有了“点号匹配换行符”的选项，所以该正则式中的点号会首先匹配任意单个的字符。在 JavaScript 版本中，代替点号出现的是字符类<[\s\S]>。然而，这两个正则式是完全等价的。<\s> 会匹配任意空白字符，而<\S>会匹配所有其他字符。因此二者组合起来就可以匹配任意字符。

懒惰的 <\*?> 量词会重复它之前的“任意字符”元素 0 次或多次，而且是重复尽量少的次数。因此，前面的记号会被一直重复到第一次出现-->为止。（关于在懒惰和贪心量词中如何进行回溯的内容，请参考实例 2.13。）因为 XML 风格的注释不允许彼此之间进行嵌套，所以这个简单的策略就足够好了。换句话说，它们总是会在第一次出现（也就是最左边的）-->的时候结束。

### 当注释不能被删除的时候

大多数 Web 开发人员都熟悉在<script>和<style>元素中使用 HTML 注释，从而可以同老版本的浏览器兼容。如今，这样做大体上只是一种没有任何意义的咒语，但是由于人们通常喜欢复制粘贴，从而这种形式也会长期存在下去。我们会假设在要从一个(X)HTML 文档中删除注释的时候，你并不想把其中内嵌的 JavaScript 和 CSS 都删除掉。你可能同样想要保留<textarea>元素的内容、CDATA 片段以及在标签中的属性值。

早些时候，我们说过删除注释并不是一件很难的工作。然而事实是，只有当你忽略(X)HTML 或 XML 中语法规则总会发生变化的区域之时，才会真的很简单。换句话说，如果你忽略了这个问题中比较难的部分，那么它就容易了。

当然，在有些情形中，你可能会评估你要处理的标记，并决定是否可以忽略这些问题情形，可能是因为你自己写的标记代码，从而知道下面会遇到什么。如果你是在一个

文本编辑器中进行查找和替换，并且能够在删除每个匹配之前可以对它进行手动审查，那么可能也是可以接受的。

但是，如果想要知道如何解决这些问题，在本章前面的“跳过复杂的(X)HTML 和 XML 片段”小节中，我们讨论了在匹配 XML 风格标签的时候如何解决相同的这些问题。在查找注释时，还可以使用类似的手段。使用在实例 3.18 中的代码，先用下面所给的正则表达式来找到这些复杂的片段，然后再把匹配之间找到的注释都替换为空字符串（也就是删除注释）：

```
<(script|style|textarea|xmp)\b(?:[^>"]|"[^"]*"|'[^\']*')*?<!--
```

```
(?:/>|>.*?</\1\s*>)|<[a-z](?:[^>"]|"[^"]*"|'[^\']*')*?>|<!\[CDATA\[.*?\]]>
```

正则选项：不区分大小写、点号匹配换行符

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

为了更容易理解这个正则式，我们采用宽松排列模式，在其中添加了空白和一些注释：

# 特殊元素：标签和内容

```
<( script | style | textarea | xmp )\b
```

(?: [^>"] # 匹配任意属性名称

| "[^"]\*" # ...和值

| '[^']\*' #

) \*?

(?: # 单体标签

/>

| # 否则，把元素的内容和匹配的结束标签包括进来

> .\*? </\1\s\*>

)

|

# 标准元素：只包含标签

```
<[a-z] # 标签名称首字符
```

(?: [^>"] # 匹配标签名称的剩余部分

| "[^"]\*" # ...以及属性

| '[^']\*' # ...名称和值

) \*

>

|

# CDATA 片段

```
<!\[CDATA\[ .*\? \]]>
```

正则选项：不区分大小写、点号匹配换行符、宽松排列

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

下面是一个用于 JavaScript 的等价版本，其中不支持“点号匹配换行符”和“宽松排列”的选项：

```
<(script|style|textarea|xmp)\b(?:[^>"]|"[^"]*"|'[^\']*')*?<!--
```

```
(?:/>|>[\s\S]*?</\1\s*>)|<[a-z](?:[^>"]|"[^"]*"|'[^\']*')*?>|<!\[CDATA\[
```

```
[ \s\S]*? ]>  
正则选项：不区分大小写  
正则流派：JavaScript
```

## 变体

### 查找合法的 XML 风格注释

实际上，对于(X)HTML 和 XML 注释来说，除了简单地以`<!--`开始和以`-->`结束之外，还存在一些其他的语法规则。具体来讲包括。

- 2个连字符不能在注释中连续出现。例如，`<!--com--ment -->`是不合法的，因为在中间出现了2个连字符。
- 结束的分隔符前面不能出现术语注释一部分的连字符。例如，`<!-- comment --->`是不合法的，但是内容完全为空的注释`<!---->`则是允许的。
- 在结束的`--`和`>`之间可以出现空格。例如，`<!-- comment-- >`是一个合法的完整注释。

用一个正则式来表示这些规则也并不是很困难：

```
<!-- [^-]* (?:-[^-]+)* --\s*>  
正则选项：无  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

注意在起始和结束注释分界符之间的所有内容都还是可选的，因此它会匹配到完全为空的注释`<!---->`。然而，如果在两个分界符之间存在一个连字符，那么它之后必须跟着至少一个非连字符的字符。而且因为这个正则式的内层部分不能匹配连续两个连字符，所以在本实例开始的时候所给的正则式中的懒惰量词被替换为了贪心量词。虽然懒惰量词也是正确的，但是却会导致不必要的回溯（参见实例 2.13）。

有些读者在看到这个新的正则式时，可能会想要知道为什么其中的否定字符类`[^-]`被使用了两次，而不是简单地把位于非捕获分组中的连字符设为可选的（例如，`<!--(?:-?[^-]+)*--\s*>`）。之所以这样做是有原因的，读者可以参考实例 2.15 中关于“灾难性回溯”的讨论。

所谓的嵌套量词（nested quantifiers）总是需要额外的小心，一定要确保不会带来灾难性回溯的可能。当一个量词出现在一个使用量词重复的分组中的时候，我们称之为嵌套量词。例如，模式`<(?:-?[^-]+)*>`中包含了两个嵌套量词：在连字符之后跟着的问号和在否定字符类之后的加号。

然而，从性能角度来看，嵌套量词并不是造成这种危险的原因所在。事实上，原因在于尝试匹配一个字符串的时候，外层的`(* )`量词与内层量词之间的组合可能会存在非常多的方式。如果正则引擎无法在一个部分匹配的结尾处找到`-->`（当你把这个模式片

段插入匹配注释的正则表达式中时，这是必需的），引擎就必须尝试所有可能的重复组合，然后才能宣布匹配尝试失败，并继续执行。这种可选择的数量会随着引擎必须尝试匹配的每个额外字符极端迅速地扩张。然而，如果这种情形能够避免的话，那么使用嵌套量词也不会带来任何危险。例如，模式 `<(?:-[^-]+)*>` 就不会带来任何风险，即使其中包含了一个嵌套的 `<+>` 量词，这是因为在这里分组的每次重复都必须刚好匹配一个连字符，这样所需的回溯点的个数就会按照目标字符串的长度线性增长。

避免潜在的回溯问题的另外一种方式是使用原子分组。下面是一个同本节中第一个正则表达式等价的正则表达式，但是它使用了较少的字符，而且在 JavaScript 或 Python 中不支持：

```
<!-- (?>-?[^-]+)*--\s*>
正则选项: 无
正则流派: .NET、Java、PCRE、Perl、Ruby
```

关于原子分组（以及和它相对的占有量词）的详细工作原理，请参考实例 2.14。

## 查找 C 语言风格的注释

前面给出的 XML 风格的注释所使用的模式，同样可以用于其他种类的非嵌套的多行注释。C 语言风格的注释可以以 `/*` 开始，以随后出现的第一个 `*/` 结束，或者以 `//` 作为开始而以一行的结尾作为结束。下面的正则表达式可以匹配这两种注释类型，其中使用一个竖线操作符来把两种类型的模式组合在了一起：

```
/\*[ \s\S]*?\*/|//.*
正则选项: 无（“点号匹配换行符”必须关闭）
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

## 参见

实例 8.9 会讲解如何在 XML 风格的注释中查找特定的单词。

## 8.9 在 XML 风格的注释中查找单词

### 问题描述

你想要找到在(X)HTML 或 XML 注释之内的出现的所有 TODO 单词。例如，你想要只匹配到下面字符串中加下划线的文本：

```
This "TODO" is not within a comment, but the next one is. <!-- TODO: -->
Come up with a cooler comment for this example. -->
```

### 解决方案

这个问题至少有 2 种解决方案，二者都各有优点。第一种方法是后面要介绍的“两步

解决方案”，它会使用一个外层正则式查找注释，然后在每个匹配中使用另外一个正则式或者甚至是使用纯文本搜索进行查找。如果你会撰写代码来完成这个任务，这样做是最好的，因为把任务划分成 2 个步骤会把事情变得简单而且快速。然而，如果你要使用一个文本编辑器或者 grep 工具来在多个文件中进行查找，那么把任务分解成两步就不可以了，除非你所选择的工具提供了特殊选项，允许你从另外一个正则式找到的匹配中进行查找<sup>1</sup>。

当你需要使用单个正则式在注释中查找单词，那么可以通过环视的帮助来完成这个任务。这第二种方法会在后面的“单步解决方案”小节中找到。

## 两步解决方案

如果情况允许，那么可以选择的较好解决方案是把这个任务分解为 2 个步骤：先查找注释，然后再在注释中查找 TODO。

下面是用来查找注释的方法：

```
<!--.*?-->
正则选项：点号匹配换行符
正则流派：.NET、Java、PCRE、Perl、Python、Ruby
```

JavaScript 中没有“点号匹配换行符”的选项，但是你可以使用一个包含所有字符的字符类来代替点号的位置，如下所示：

```
<!--[\s\S]*?-->
正则选项：无
正则流派：JavaScript
```

使用上面所给的一个正则式找到注释之后，你可以在匹配到的每一个注释中查找字面的字符串 <TODO>。如果愿意，还可以使用一个不区分大小写的正则式，并且在两端添加单词边界，使之只能匹配到完整的 TODO 单词，如下所示：

```
\bTODO\b
正则选项：不区分大小写
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

实例 3.13 中展示了如何从一个外层正则式的匹配中进行查找。

## 单步解决方案

顺序环视（参考实例 2.16 中的讲解）允许使用单个正则式来解决这个问题，虽然这样做的效率会有些低。在下面这个正则表达式中，使用了肯定型顺序环视来确保单词 TODO 之后会跟着结束的注释分隔符号 -->。如果只有它，并不能说明该单词是位于一个注释内部，还是在它之后跟着一个注释，因此我们使用了一个否定型顺序环视来确

---

<sup>1</sup> 本书第一章中“使用正则表达式的工具”小节中介绍过的 PowerGREP 工具就允许在匹配中进行查找。

保在 `-->` 之前不会出现起始注释分隔符号 `<!--:`

```
\bTODO\b (?=(?:(?!\<!--).)*?-->)
正则选项: 不区分大小写、点号匹配换行符
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

因为 JavaScript 中没有“点号匹配换行符”的选项，所以我们在点号的位置中使用了一个 `\s\S` 作为替代：

```
\bTODO\b (?=(?:(?!\<!--)[\s\S])*?-->)
正则选项: 不区分大小写
正则流派: JavaScript
```

## 讨论

### 两步解决方案

实例 3.13 中给出了在另外一个正则表达式的匹配中进行查找的代码。它需要一个内层和一个外层正则表达式。查找注释的正则式会用作外层正则式，而 `\bTODO\b` 则会被用作内层正则式。这里需要注意的一点是在注释正则式中的点号或者字符类之后的懒惰 `(*?)` 量词。我们在实例 2.13 中解释过，它会允许你匹配之后的第一个 `-->`（也就是当前注释的结束），而不是在目标字符串中的最后一个 `-->`。

### 单步解决方案

这个解决方案更为复杂，而且速度也较慢。它的优点是把前面解决方案中的 2 个步骤结合成了一个正则表达式。因此，它可以被用在文本编辑器、IDE 或者其他不允许在另外一个正则式的匹配中进行查找的工具中。

下面我们给出了这个正则式在宽松排列模式下的分解形式，仔细看一下它的各个组成部分：

```
\b TODO \b      # 匹配字符序列 "TODO"，整字匹配
(?=             # 判断下面的正则式可以在这里产生匹配
(?:             # 分组但是不捕获...
  (?!\<!-- )    # 判断 "<!--" 不能在这里产生匹配
  .
) *?           # 0 次或多次，匹配尽量少次（懒惰）
  -->          # 匹配字符序列 "-->"
```

正则选项: 点号匹配换行符、宽松排列  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby

这个包含注释的版本不能用于 JavaScript，因为 JavaScript 中不支持“宽松排列”和“点号匹配换行符”的模式。

注意这个正则式中包含了一个嵌套在一个外层的肯定型顺序环视之内的否定型顺序环

视。这样就允许你要求所匹配到的任何 TODO 之后都会跟着 `-->`，而且在二者之间不会出现 `<!--`。

如果你已经很清楚这个正则表达式如何工作，那么可以跳过本小节的剩余部分。但是如果你仍有些迷惑，那么就再往回退一步，看一下这个正则表达式中外层的肯定型顺序环视是如何逐步构造的。

现在暂时假设我们只是想要匹配在字符串后面某处会跟着字符序列 `-->` 的所有单词 TODO。这样我们就得到了一个正则表达式 `\bTODO\b(=?.*?-->)`（要求打开“点号匹配换行符”选项），它会匹配在字符串 `<!--TODO-->` 中带下划线的文本。我们需要在顺序环视的开始使用 `.*?`，否则，这个正则式只能匹配其后紧跟着 `-->` 的单词 TODO，二者之间不能包含任何字符。量词 `*?` 会重复点号 0 次或多次，但是会重复尽量少的次数，这是因为我们只想匹配到第一个出现的 `-->` 为止。

稍微离一下题，我们可以把到目前为止所得到的这个正则式重新写作 `\bTODO(=?.*?-->)\b`，把其中第 2 个 `\b` 移到循序环视之后，这样不会影响它将匹配到的文本内容。这是因为两个单词边界和顺序环视都是长度为 0 的断言（参考实例 2.16 中的“环视”小节）。然而，最好还是先使用单词边界，这样可以提高可读性和效率。在一个部分匹配的中间，正则引擎可以更快地检查单词边界，宣布匹配失败，然后继续在字符串中的下一个字符进行尝试，而不必在根本没有必要的情况下（因为这样匹配到的 TODO 不是一个完整的单词）还要去花费时间检查顺序环视。

现在我们继续刚才的讨论，正则式 `\bTODO\b(=?.*?-->)` 现在看来应该够用了，但是如果把它应用到目标字符串 TODO `<!-- separate comment -->` 之上会怎样呢？这个正则式还是会匹配到 TODO，因为它之后会跟着 `-->`，然而这里的 TODO 并不位于一个注释之间。因此，我们还需要把位于顺序环视之内的点号从可以匹配任意字符替换为可以匹配不属于字符串 `<!--` 一部分的任意字符，否则就意味着中间会包含一个新的注释的开始。我们不能在这里使用诸如 `[^<!--]` 这样的否定型字符类，因为我们会允许出现`<`、`!` 和 `-` 这些字符，只是不允许它们正好按照 `<!--` 的序列出现。

这个时候就需要引入嵌套的否定型顺序环视了。`((?!<!--).)*` 会匹配不属于起始注释分隔符一部分的任意单个字符。把这个模式放到一个非捕获分组 `((?:((?!<!--).)*))` 中，就允许我们可以用之前应用在点号之上的懒惰量词 `*?` 来重复整个序列。

把我们前面讲的这些结合起来，就得到解决这个问题的最终正则表达式：`\bTODO\b(=?((?:((?!<!--).)*?))-->)`。在 JavaScript 中，因为不能使用“点号匹配换行符”选项，它的等价形式是 `\bTODO\b(=?((?:((?!<!--)[\s\S])*?))-->)`。

## 变体

虽然“单步解决方案”中的正则表达式会确保所匹配到的任意 TODO 之后都会跟着

-->, 而且二者之间不会出现 <!--, 但是它并没有检查反过来的情形：也就是说目标单词之前应该包括 <!--, 而且二者之间不能有 -->。之所以没有做这样的检查，是基于如下的几个原因：

- 通常可以不必进行这种双重检查也不会出现问题，特别是因为单步正则表达式的本意是用在文本编辑器或类似工具中，从而你可以亲眼确认得到的结果；
- 进行更少的检查也就意味着在执行检查时会花费更少的时间（也就是说，略去额外的检查会提高速度）；
- 最重要的是，由于你并不知道该注释的起始位置在前面多远的地方，因此这样的向后看要求无限长度的逆序环视，而它只有在.NET 中才能支持。

如果你用得正好是.NET，而且想要添加这样的检查，那么就可以使用如下的正则表达式：

```
(?<=<!--(?: (?!-->).)*?)\bTODO\b(?(=(?: (?!<!--).)*?-->))
```

正则选项：不区分大小写、点号匹配换行符

正则流派：.NET

这个只能用于.NET 的更加严格的正则表达式在前面添加了一个肯定型逆序环视，它与结尾处的顺序环视的工作原理一样，只不过它是反过来进行检查的。因为逆序环视会从它找到 <!-- 的位置向前进行检查，所以逆序环视中包含了一个嵌套的否定型顺序环视，它允许匹配不属于序列 --> 的任意字符。

因为前导的顺序环视和结尾处的顺序环视都是长度为 0 的断言，所以最后匹配到的仍然只是单词 TODO。在环视中匹配到的字符串并不会被加入到最后匹配到的文本中。

## 参见

实例 8.8 中包含关于如何匹配 XML 风格注释的详细讨论。

## 8.10 替换在 CSV 文件中使用的分隔符

### 问题描述

你想要把在 CSV 文件中的所有的逗号域分隔符都替换为制表符。在双引号引起的值中的逗号应当保持不变。

### 解决方案

下面的正则表达式会匹配一个单独的 CSV 域，以及之前可能存在的分隔符。前面的分隔符通常是一个逗号，但是如果匹配到的是第一个记录中的第一个域，那么也可以是

一个空字符串（也就是什么都没有），如果匹配到的是之后任意记录的第一个域，那么也可以是一个换行符。每次找到一个匹配时，这个域本身，包括两边可能会用到的双引号，会被一起捕获到第 2 个向后引用中，而它之前的分隔符则会被匹配到第 1 个向后引用中。



### 提示

在本实例中的正则表达式只能用于合法的 CSV 文件，它们必须遵守在本章前面讨论过的“逗号分隔值（CSV）”小节中的格式规则。

```
(, | \r?\n|^) ([^", \r\n]+|"(?:[^"]|"")*")?
```

正则选项：无（“点号匹配换行符”必须关掉）

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

下面给出了同一个正则表达式的宽松排列模式：

```
( , | \r?\n | ^ )      # 捕获分组 #1 匹配域分隔符或者字符串开始  
(                           # 捕获分组 #2 匹配一个单独的域，它可以是：  
  [^", \r\n]+           # 一个没有被引起的域  
  |                   # 或者...  
  " (?:[^"]|"")* "    # 一个引起的域（可能会包含被转义的双引号）  
)?                      # 这个分组是可选的，因为域可以为空
```

正则选项：宽松排列（“^ 和 \$ 匹配换行处”选项必须关闭）

正则流派：.NET、Java、PCRE、Perl、Python、Ruby

使用这个正则表达式和在实例 3.11 中的代码，你就可以遍历一个 CSV 文件，并且在得到每个匹配之后检查第 1 个向后引用中的取值。每个匹配所需要的替代字符串则要取决于这个向后引用的值。如果它是一个逗号，那么把它替换成一个制表字符。如果向后引用中为空，或者包含一个换行符，那么就不要对这个值做任何改动（也就是说把它作为替代字符串的一部分放回去）。因为 CSV 域会被捕获到第 2 个向后引用中，所以还需要把它也作为每个替代字符串的一部分放回去。你实际上要替换的内容只是在第 1 个向后引用中所捕获到的逗号。

### JavaScript 示例

下面的代码是一个完整的网页，其中包含了 2 个多行的文本输入域，在二者之间有一个标签为 *Replace* 的按钮。单击这个按钮，就可以把你输入到第一个文本框（标签为 *Input*）中的任意字符串，使用刚才所给的正则表达式，把其中的任意逗号分隔符都替换成制表符，然后把新的字符串输出到第二个文本框（标签为 *Output*）中。如果你使用合法的 CSV 内容作为输入，那么在第二个文本框中出现的就会是把其中所有逗号分隔符都替换成制表符之后的内容。要测试它，只需把这段代码保存到一个后缀为 “.html”的文件中，然后在你的 Web 浏览器中打开这个文件：

```
<html>  
<head>  
<title>Change CSV delimiters from commas to tabs</title>
```

```
</head>

<body>
<p>Input:</p>
<textarea id="input" rows="5" cols="75"></textarea>

<p><input type="button" value="Replace" onclick="commas_to_tabs()"></p>

<p>Output:</p>
<textarea id="output" rows="5" cols="75"></textarea>

<script>
function commas_to_tabs () {
    var input = document.getElementById('input'),
        output = document.getElementById('output'),
        regex = /(,|\r?\n|^)([^,\r\n]+|"(?:[^"]|"")*")?/g,
        result = '',
        match;

    while (match = regex.exec(input.value)) {
        // Check the value of backreference 1
        if (match[1] == ',') {
            // Add a tab (in place of the matched comma)
            // and backreference 2 to the result.
            // If backreference 2 is undefined (because the
            // optional, second capturing group did not participate
            // in the match), use an empty string instead.
            result += '\t' + (match[2] || '');
        } else {
            // Add the entire match to the result
            result += match[0];
        }

        // Prevent some browsers from getting stuck in an infinite loop
        if (match.index == regex.lastIndex) regex.lastIndex++;
    }

    output.value = result;
}
</script>
</body>
</html>
```

## 讨论

本实例中所给的方法允许你可以一次跳过一个完整的 CSV 域（包括其中内嵌的任何换行符、转义的双引号和逗号）。这样，每个匹配都会正好从下一个域分隔符处开始。

在这个正则表达式中的第一个捕获分组 `<,(|\r?\n|^)>` 会匹配一个逗号、换行符或者在目

标字符串最开始处的位置。因为正则引擎会从左向右尝试所有的选择分支，这些选项所采用的顺序是它们在普通的 CSV 文件中最可能出现的频率。这个捕获分组是这个正则式唯一必须匹配的部分。因此，完整的正则式有可能会匹配到一个空字符串，因为 `\^` 定位符总是只会匹配一次。这第一个捕获分组匹配到的值必须要在位于正则式之外把逗号替换为新的替代分隔符（例如制表符）的代码中进行检查。

我们还没有介绍完整的正则表达式，但是目前为止讲过的方式已经有点儿让人摸不着头脑了。你可能会想要知道为什么不把这个正则式写成只用来匹配应当被替换为制表符的逗号。如果能够这样做，那么就只需要对所有匹配到的文本使用一个简单的替代，这样就不再需要正则式之外的代码来检查第 1 个捕获分组中匹配到的是一个逗号还是其他字符串。毕竟，使用顺序环视和逆序环视来确定一个逗号是位于被引起的一个 CSV 域之内还是之外应该是可行的，对吗？

不幸的是，要想采用这样的一种方式来准确决定哪些逗号位于双引号引起的域之外，你需要无限长度的逆序环视，而它只有在 .NET 正则流派中才会支持（关于各种不同的逆序环视的限制，请参考实例 2.16 中的“不同级别的逆序环视”小节）。即使是 .NET 开发人员也应当避免使用基于环视的方案，因为它会使之变得明显更加复杂，同样会使这个正则式速度更慢。

我们再回头来看这个正则式是如何工作的，大多数的模式都位于下面一对圆括号之内：也就是第 2 个捕获分组之内。这第二个分组会匹配单独的一个 CSV 域，其中包括两边的双引号。与前一个捕获分组不同，这个分组是可选的，这样它就可以允许匹配空的域。

注意在正则式中的第 2 个捕获分组中包含了由元字符 `\|` 分开的两个可选择的模式。第一个选择分支 `\[^",\r\n]+` 是一个否定型字符类，其后跟着一个重复 1 次或多次的量词，它们一起会匹配一整个没有引起的域。要想产生匹配，这个域中不能包括任何双引号、逗号或换行符。

在第 2 个分组中的第二个选择分支 `"(?:[^"]|""")*">` 会匹配一个由双引号包起来的域。更精确地来讲，它会匹配一个双引号字符，随后是 0 个或多个非双引号字符和重复的（被转义的）双引号，再之后是一个结束的双引号。

位于内层的非捕获分组结尾处的量词 `*` 会重复内层的两个选择尽量多次，直至遇到了一个没有被重复的双引号，因此也就匹配到了一个域的结尾。

假设你要处理的是一个合法的 CSV 文件，那么这个正则式的第一个匹配应当出现在目标字符串的开始，而随后的每个匹配都会紧跟在上一个匹配的结尾之后。

## 参见

实例 8.11 中会讲解如何复用本实例中的正则表达式来抽取某个特定的列中的 CSV 域。

## 8.11 抽取某个特定列中的 CSV 域

### 问题描述

你想要抽取一个 CSV 文件的第 3 列中的每一个域。

### 解决方案

我们在这里可以复用在实例 8.10 中给出的正则表达式，用它来在一个 CSV 目标字符串中对每个域进行遍历。如果再添加一些额外代码，就可以从左向右统计每行（或者每个记录）中的列数，然后抽取你所感兴趣的位置处的域。

下面的正则表达式（分别给出了使用和不使用宽松排列选项的形式）会在两个单独的捕获分组中匹配单个的 CSV 域和位于它之前的分隔符。因为换行符可以出现在双引号引起的域中，所以如果只是简单在你的 CSV 字符串中从一行的开始来查找是不够准确的。通过匹配每个域并逐个跳过它们，就能够很容易确定哪些换行符是位于双引号引起的域之外，因此它会开始一个新的记录。



#### 提示

在本实例中的正则表达式只能用于合法的 CSV 文件，它们必须遵守在本章前面讨论过的“逗号分隔值（CSV）”的格式规则。

```
(, | \r?\n|^) ([^",\r\n]+|"(?:[^"]|"")*")?  
正则选项: 无（“^ 和 $ 匹配换行处”选项必须关闭）  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby  
  
( , | \r?\n | ^ )      # 捕获分组 #1 匹配域分隔符或者字符串开始  
(                           # 捕获分组 #2 匹配一个单独的域，它可以是:  
[^",\r\n]+                # 一个没有被引起的域  
|                           # 或者...  
" (?:[^"]|"")* "          # 一个引起的域（可能会包含被转义的双引号）  
)?                         # 这个分组是可选的，因为域可以为空  
正则选项: 宽松排列（“^ 和 $ 匹配换行处”选项必须关闭）  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

这些正则表达式与实例 8.10 中所给的正则式是完全一样的，并且可以通过修改它们来完成许多其他的 CSV 处理任务。下面的示例代码会展示如何使用前面不带宽松排列选项的版本来帮助你抽取一个 CSV 列的内容。

### JavaScript 示例

下面的代码是一个完整的网页，其中包含了两个多行的文本输入域，在二者之间有一个标签为 Extract Column 3 的按钮。单击这个按钮，就可以把输入到第一个文本框（标

签为 Input) 中的任意字符串, 使用本实例中刚才所给的正则表达式来把其中每个记录中的第 3 个域中的值抽取出来, 然后把整个列 (每个值之间用换行符隔开) 输出到 Output 文本框中。要测试它, 只需要把这段代码保存到一个后缀为 “.html” 的文件中, 然后在你的 web 浏览器中打开这个文件:

```
<html>
<head>
<title>Extract the third column from a CSV string</title>
</head>

<body>
<p>Input:</p>
<textarea id="input" rows="5" cols="75"></textarea>

<p><input type="button" value="Extract Column 3"
    onclick="display_csv_column(2)"></p>

<p>Output:</p>
<textarea id="output" rows="5" cols="75"></textarea>

<script>
function display_csv_column (index) {
    var input = document.getElementById('input'),
        output = document.getElementById('output'),
        column_fields = get_csv_column(input.value, index);

    if (column_fields.length > 0) {
        // Show each record on its own line, separated by a line feed (\n)
        output.value = column_fields.join('\n');
    } else {
        output.value = '[No data found to extract]';
    }
}

// Return an array of CSV fields at the provided, zero-based index
function get_csv_column (csv, index) {
    var regex = /(,|\r?\n|^)([^,\r\n]+|"(?:[^"]|"")*")?/g,
        result = [],
        column_index = 0,
        match;

    while (match = regex.exec(csv)) {
        // Check the value of backreference 1. If it's a comma,
        // increment column_index. Otherwise, reset it to zero.
        if (match[1] == ',') {
            column_index++;
        } else {

```

```
        column_index = 0;
    }
    if (column_index == index) {
        // Add the field (backref 2) at the end of the result array
        result.push(match[2]);
    }
    // Prevent some browsers from getting stuck in an infinite loop
    if (match.index == regex.lastIndex) regex.lastIndex++;
}

return result;
}
</script>
</body>
</html>
```

## 讨论

因为这里的正则表达式是从实例 8.10 中拿来的，所以我们不会在这里重复解释它们的工作原理。然而，本实例中包括了新的 JavaScript 示例代码，它使用了这个正则式从一个 CSV 目标字符串中的每个记录中抽取某个特定索引的域。

在前面所给的代码中，`get_csv_column` 函数会每次遍历目标字符串中的一个匹配。在每个匹配之后，它会检查第 1 个向后引用，判断其中包含的是不是一个逗号。如果是，那么就匹配到了除了一行中第一个域之外的内容，因此变量 `column_index` 会被加 1，来跟踪你正在处理的是第几列。如果第 1 个向后引用中包含的是除了逗号之外的任何内容（例如是一个空字符串或者换行），那么就匹配到了一个新行中的第一个域，这样 `column_index` 会被重新设置为 0。

在代码中的下一步是检查计数器 `column_index` 是否已经到达了你想要抽取的列索引。每次到达该索引的时候，第 2 个向后引用中的值（也就是除了前导的分隔符之外的所有内容）会被推送到数组 `result` 中。在遍历了整个目标字符串之后，`get_csv_column` 函数会返回一个包含整个指定列（在这个例子中是第 3 列）中的值的数组。所得到的匹配列表接着会被输出到页面上的第二个文本框中，每个值之间使用一个换行符作为分隔（\n）。

一个简单的改进是可以让用户指定应当抽取的列的索引，这可以通过提示框或者添加一个文本域来实现。我们前面所讨论的 `get_csv_column` 函数已经考虑到了这个功能，它允许在第 2 个参数（`index`）中指定你想要抽取的列索引（从 0 开始的整数）。

## 变体

虽然使用代码来每次遍历字符串中的一个 CSV 域会支持额外的灵活性，但是如果你使用一个文本编辑器来执行这个任务，那么可能就只限于使用查找和替换功能。在这种

情形下，你可以通过匹配每个完整的记录，然后把它替换成要查找的列索引处的域的取值（使用向后引用）来得到类似的结果。下面列出的正则表达式会为特定的列索引讲解抽取的方法。

在所有这些正则表达式中，如果任何记录中所包括的域的个数小于要查找的列索引，那么这个记录就不会产生匹配，从而也不会发生改变。

匹配一个 CSV 记录，并把第 1 列中的值捕获到第 1 个向后引用中

```
^([^\n,\r]+|^((?:[^"]|"")*)?)?(?:,(?:[^",\r\n]+|^((?:[^"]|"")*))?)*
```

正则选项：^ 和 \$ 匹配换行处

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

匹配一个 CSV 记录，并把第 2 列中的值捕获到第 1 个向后引用中

```
^((?:[^",\r\n]+|^((?:[^"]|"")*)?)?,([^\n,\r]+|^((?:[^"]|"")*))?)?←  
(?:,(?:[^",\r\n]+|^((?:[^"]|"")*))?)*
```

正则选项：^ 和 \$ 匹配换行处

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

匹配一个 CSV 记录，并把第 3 列中的值捕获到第 1 个向后引用中

```
^((?:[^",\r\n]+|^((?:[^"]|"")*)?)?(?:,(?:[^",\r\n]+|^((?:[^"]|"")*))?)?)?{1},←  
([^\n,\r]+|^((?:[^"]|"")*))?)?(?:,(?:[^",\r\n]+|^((?:[^"]|"")*))?)*
```

正则选项：^ 和 \$ 匹配换行处

正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

接着再增加在量词 <{1}> 之内的数字就可以让最后一个正则式查找第 3 列之后的内容。例如，把它修改为 <{2}> 就可以捕获第 4 列中的域，改成 <{3}> 可以捕获第 5 列，以此类推。如果要处理的是第 3 列，那么你也可以把<{1}>去掉，因为它在这里不会产生任何作用。

## 替代字符串

对于所有这些正则式来说，都可以使用同一个替代字符串（也就是第 1 个向后引用）。把每个匹配都替换为第 1 个向后引用就可以只留下你在查找的域。

```
$1
```

替代文本流派：.NET、Java、JavaScript、Perl、PHP

```
\1
```

替代文本流派：Python、Ruby

## 8.12 匹配 INI 段头

### 问题描述

你想要匹配一个 INI 文件中的所有段头。

## 解决方案

这个问题很容易解决。INI 段头会出现在一行的开始处，它的形式是在两个方括号之间放一个名称（例如，[Section1]）。这样的规则很容易就可以转换成一个正则表达式：

```
^\[[^\]\r\n]+]
```

正则选项：^ 和 \$ 匹配换行处  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

## 讨论

这个正则表达式并不复杂，所以可以很容易地对它进行分解。

- 前导的 `\^` 会匹配一行的开始位置，因为我们打开了“^ 和 \$ 匹配换行处”的选项。
- `\[` 会匹配一个字面的 [ 字符。我们使用了一个反斜杠对它进行转义，以避免 [ 会作为一个字符类的开始。
- `\[^]\r\n]+` 是一个否定字符类，它会匹配除了 ]、回车 (\r) 或换行 (\n) 之外的任意字符。紧跟其后的 `+>` 量词会使这个类可以匹配一个或多个字符，然后……
- 最后面的 `\]` 会匹配一个字面的 ] 字符，作为段头的结束。这里没有必要对这个字符用反斜杠转义，因为它没有出现在一个字符类之中。

如果你只想找到某个特定的段头，那么解决方案会更加容易。下面的正则表达式可以匹配一个名为 Section1 的段的头部：

```
^\[Section1]
```

正则选项：^ 和 \$ 匹配换行处  
正则流派：.NET、Java、JavaScript、PCRE、Perl、Python、Ruby

在这个例子中，与纯文本查找 “[Section1]” 相比，唯一的区别是这个匹配必须出现在一行的开始。这样就可以避免匹配到被注释掉的段头（前面会使用一个分号），或者看起来像是段头，但实际上是一个参数值的一部分的内容（例如，Item1 = [Value1]）。

## 参见

实例 8.13 会讲解如何匹配 INI 段块。

实例 8.14 会讲解如何匹配 INI 名称-值对。

## 8.13 匹配 INI 段块

### 问题描述

你需要匹配每个完整的 INI 段块（Section Blocks，即一个段头和该段中所有的参数-值

对)，从而可以把一个INI文件分解开来，分别对每个块进行处理。

## 解决方案

实例 8.12 中讲解了如何匹配一个INI段头。要想匹配整个的段，我们会首先使用上一个实例中相同的模式，但是会继续匹配，直至到达字符串的结束，或者是在一行开始处出现了一个 [ 字符为止（因为这意味着一个新的段的开始）：

```
^\[[^\]\r\n]+] (?:\r?\n(?:[^[\r\n].*)?)*  
正则选项: ^ 和 $ 匹配换行处（“点号匹配换行符”必须关掉）  
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby
```

下面是同一个正则表达式的宽松排列形式：

```
^ \[ [^\]\r\n]+ ]      # 匹配一个段头  
(?:                  # 然后是该段中的其余内容  
  \r?\n            # 匹配一个换行字符序列  
(?:                # 在每行开始之后，匹配  
  [^\r\n]          # 除了 "[" 或换行字符之外的任意字符  
  .*               # 匹配该行的剩余内容  
)?                 # 该分组是可选的，从而可以匹配空行  
)*                # 继续匹配，直到该段结束  
正则选项: ^ 和 $ 匹配换行处、宽松排列（“点号匹配换行符”必须关掉）  
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

## 讨论

这个正则表达式会首先使用模式 `^\[[^\]\r\n]+]` 来匹配一个INI段头，并且会继续每次匹配一行，只要这些行都不以 [ 开头。以下面这个目标文本为例：

```
[Section1]  
Item1=Value1  
Item2=[Value2]  
  
; [SectionA]  
; The SectionA header has been commented out  
  
ItemA=ValueA ; ItemA is not commented out, and is part of Section1  
  
[Section2]  
Item3=Value3  
Item4 = Value4
```

使用上面给出的这个字符串，前面的正则式能找到两个匹配。第一个匹配从字符串的开头一直到 “[Section2]” 之前，并且包含它之前的空行。第二个匹配则从 Section2 头开始直到字符串的结束。

## 参见

实例 8.12 讲解如何匹配INI段头。

实例 8.14 讲解如何匹配INI名称-值对。

## 8.14 匹配INI名称-值对

### 问题描述

你想要匹配INI参数名称-值对（例如，Item1=Value1），使用捕获分组把每个匹配分隔到两个部分中。第1个向后引用中应当包含参数名称（Item1），而第2个向后引用中应当包含值（Value1）。

### 解决方案

下面是完成这个任务的正则表达式（第2个正则式采用了宽松排列模式）：

```
^( [^=;\r\n]+ ) = ([^;\r\n]*)
正则选项: ^ 和 $ 匹配换行处
正则流派: .NET、Java、JavaScript、PCRE、Perl、Python、Ruby

^          # 一行开始
( [^=;\r\n]+ )      # 把名称捕获到向后引用#1中
=
( [^;\r\n]* )       # 把直捕获到向后引用#2中
正则选项: ^ 和 $ 匹配换行处、宽松排列
正则流派: .NET、Java、PCRE、Perl、Python、Ruby
```

### 讨论

与本章中其他INI相关的实例一样，这里我们要处理的也是比较直接的内容。这个正则式以`<^>`开头，它会匹配一行的开始位置（要确保“`^`和`$`匹配换行处”的选项被打开）。这是很重要的，因为如果不能确保匹配是从一行开头开始，那么你就可能会匹配被注释掉的一行中的一部分。

接下来，这个正则式中使用了一个捕获分组，其中包含一个否定字符类（`<[^=;\r\n]>`），之后跟着一次或多次量词（`<+>`）来匹配参数的名称，并把它记在第1个向后引用中。这个否定字符类会匹配除了下面4个字符之外的任意字符：等号、分号、回车（`\r`）和换行（`\n`）。回车和换行字符都可以被用来作为一个INI参数的结束，而分号可以表示一个注释的开始，等号则用来分隔参数的名称和值。

在匹配了参数名称之后，这个正则式会匹配一个字面的等号（名称-值分隔符），然后匹配参数的值。这个值会用第2个捕获分组来匹配，它同用来匹配参数名称的模式很类似，但是要少两个限制。首先，这第2个子模式允许匹配等号作为值的一部分（也就

是说，在字符类中可以少一个否定字符）。其次，它使用了一个`<*>`量词来去掉了必须至少匹配一个字符的需求（也就是说和名称不一样，值可以为空）。

## 参见

实例 8.12 讲解如何匹配INI段头。

实例 8.13 详细讲解如何匹配INI段块。

## 作者简介

本书的作者是 Jan Goyvaerts 和 Steven Levithan，他们都是正则表达式的世界级专家。

Jan Goyvaerts 领导着 Just Great Software 公司，他在这个公司设计和开发了一些最流行的正则表达式软件。他的产品中包括 RegexBuddy，世界上唯一可以模拟 15 种正则表达式流派特性的正则表达式编辑器，以及 PowerGREP，Microsoft Windows 平台上功能最强大的 grep 工具。

Steven Levithan 是顶尖的 JavaScript 正则表达式专家，他拥有一个很流行的以正则表达式为主题的博客 <http://blog.stevenlevithan.com>。过去几年中，他一直致力于深入研究正则表达式流派和函数库支持。

## 封面介绍

本书封面图片是一只麝鼩（英文名 musk shrew，俗称铁鼠，鼩鼱科，麝鼩属）。麝鼩可以分为多种，包括白齿和红齿的鼩鼱、灰麝鼩和红麝鼩。鼩鼱原产于非洲南部和印度。

虽然不同种类鼩鼱的体态特征不同，但是所有鼩鼱都拥有很多共同点。例如，鼩鼱被认为是世界上最小的食虫动物，所有鼩鼱都是短腿，每只脚有 5 个脚趾，长鼻子，触觉毛。区别包括它们牙齿的颜色不同（主要是白齿鼩鼱和红齿鼩鼱），以及毛发颜色不同（包括红色、褐色和灰色）。

虽然鼩鼱通常以昆虫为食，但它还会吃老鼠或田野里的其他小型啮齿动物，从而帮助农民控制田里的危害。

许多麝鼩会释放出一种强烈的麝香气味（它们也因此而得名），并用这种气味来标记它们的领土。曾经有传闻说麝鼩的气味是如此强烈，甚至会渗透到麝鼩所经过的任何葡萄酒或啤酒瓶中，从而把酒也污染上麝香的气味，但后来这个传闻被证明是假的。

封面的图片来自于 Lydekker 的 *Royal Natural History*。



# 计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

**Java** 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

**.Net** 技术精品资料下载汇总: **ASP.NET** 篇

**.Net** 技术精品资料下载汇总: **C#语言** 篇

**.Net** 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++** 编程语言学习资料尽收眼底 电子书+视频教程

**Visual C++(VC/MFC)** 学习电子书及开发工具下载

**Perl/CGI** 脚本语言编程学习资源下载地址大全

**Python** 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

**UML** 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

**Linux** 系统管理员必备参考资料下载汇总

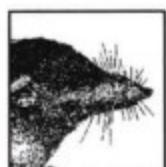
**Linux shell**、内核及系统编程精品资料下载汇总

**UNIX** 操作系统精品学习资料<电子书+视频>分类总汇

**FreeBSD/OpenBSD/NetBSD** 精品学习资源索引 含书籍+视频

**Solaris/OpenSolaris** 电子书、视频等精华资料下载索引

# 正则表达式经典实例



每个程序员都会遇到需要使用正则表达式的情况，但是要用好正则表达式却不容易。本书提供了100多个实例，以帮助读者使用正则表达式处理数据和文本。即使有经验的用户也经常会遇到性能不佳、误报、漏报等让人挠头的错误，本书对于如何使用正则表达式来解决一些常见的问题给出了按部就班的解决方案，其中包括C#、Java、JavaScript、Perl、PHP、Python、Ruby和VB.NET的实例。

本书主要包括以下内容：

- 通过一个精练的教程理解正则表达式的基本原理和技巧；
- 在不同的编程语言和脚本语言中有效使用正则表达式；
- 学习如何对输入进行合法性检查和格式化；
- 处理单词、文本行、特殊字符和数值；
- 学习如何在URL、路径、标记语言和数据交换中使用正则表达式；
- 学习更高深的正则表达式特性中的微妙之处；
- 理解在不同语言中正则表达式的API、语法和行为之间的区别；
- 创建更好的正则表达式来满足个性化的需求。

[www.oreilly.com](http://www.oreilly.com)

O'Reilly Media, Inc.授权人民邮电出版社出版

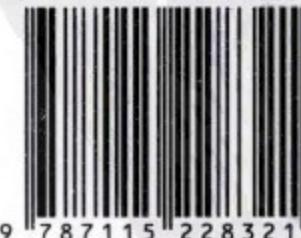
此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行  
This Authorized Edition for sale only in the territory of People's Republic of China  
(excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/程序设计

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)



ISBN 978-7-115-22832-1



ISBN 978-7-115-22832-1

定价：69.00 元

“这是一本严谨的著作，内容非常丰富。仅仅阅读前几章我就学到了大量的新技巧。”

——Nikolaj Lindberg

计算语言学家  
STTS语音技术服务公司

“本书为紧迫的问题提供了很好的解决方案。对于实例中所包含的细节我感到非常震惊。”

——Zak Greant

开放技术倡导者  
和策略专家

Jan Goyvaerts领导着Just Great Software公司，他在这个公司设计和开发了一些非常流行的正则表达式软件。

Steven Levithan是业界顶尖的JavaScript正则表达式专家，维护着一个以正则表达式为主题的人气博客。