

精通正则表达式第三版

前言.....I

第1章：正则表达式入门.... 1

解决实际问题... 2

作为编程语言的正则表达式... 4

以文件名做类比... 4

以语言做类比... 5

正则表达式的知识框架... 6

对于有部分经验的读者... 6

检索文本文件：Egrep. 6

Egrep 元字符... 8

行的起始和结束... 8

字符组... 9

用点号匹配任意字符... 11

多选结构... 13

忽略大小写... 14

单词分界符... 15

小结... 16

可选项元素... 17

其他量词：重复出现... 18

括号及反向引用... 20

神奇的转义... 22

基础知识拓展... 23

语言的差异... 23

正则表达式的目标...	23
更多的例子...	23
正则表达式术语汇总...	27
改进现状...	30
总结...	32
一家之言...	33
第2章：入门示例拓展....	35
关于这些例子...	36
Perl 简短入门...	37
使用正则表达式匹配文本...	38
向更实用的程序前进...	40
成功匹配的副作用...	40
错综复杂的正则表达式...	43
暂停片刻...	49
使用正则表达式修改文本...	50
例子：公函生成程序...	50
举例：修整股票价格...	51
自动的编辑操作...	53
处理邮件的小工具...	53
用环视功能为数值添加逗号...	59
Text-to-HTML 转换...	67
回到单词重复问题...	77
第3章：正则表达式的特性和流派概览....	83
在正则的世界中漫步...	85

正则表达式的起源...	85
最初印象...	91
正则表达式的注意事项和处理方式...	93
集成式处理...	94
程序式处理和面向对象式处理...	95
查找和替换...	98
其他语言中的查找和替换...	100
注意事项和处理方式：小结...	101
字符串，字符编码和匹配模式...	101
作为正则表达式的字符串...	101
字符编码...	105
正则模式和匹配模式...	110
常用的元字符和特性...	113
字符表示法...	115
字符组及相关结构...	118
锚点及其他“零长度断言”	129
注释和模式量词...	135
分组，捕获，条件判断和控制...	137
高级话题引导...	142
第4章：表达式的匹配原理....	143
发动引擎...	143
两类引擎...	144
新的标准...	144
正则引擎的分类...	145
几句题外话...	146

测试引擎的类型... 146

匹配的基础... 147

关于范例... 147

规则1：优先选择最左端的匹配结果... 148

引擎的构造... 149

规则2：标准量词是匹配优先的... 151

表达式主导与文本主导... 153

NFA 引擎：表达式主导... 153

DFA 引擎：文本主导... 155

第一想法：比较 NFA 与 DFA... 156

回溯... 157

真实世界中的例子：面包屑... 158

回溯的两个要点... 159

备用状态... 159

回溯与匹配优先... 162

关于匹配优先和回溯的更多内容... 163

匹配优先的问题... 164

多字符“引文” 165

使用忽略优先量词... 166

匹配优先和忽略优先都期望获得匹配... 167

匹配优先、忽略优先和回溯的要旨... 168

占有优先量词和固化分组... 169

占有优先量词，`?+`、`*+`、`++`和`{m,n}+` 172

环视的回溯... 173

多选结构也是匹配优先的吗... 174

发掘有序多选结构的价值... 175

NFA、DFA 和 POSIX.. 177

最左最长规则... 177

POSIX 和最左最长规则... 178

速度和效率... 179

小结：NFA 与 DFA 的比较... 180

总结... 183

第5章：正则表达式实用技巧.... 185

正则表达式的平衡法则... 186

若干简单的例子... 186

匹配连续行（续前）... 186

匹配 IP 地址... 187

处理文件名... 190

匹配对称的括号... 193

防备不期望的匹配... 194

匹配分隔符之内的文本... 196

了解数据，做出假设... 198

去除文本首尾的空白字符... 199

HTML 相关范例... 200

匹配 HTML Tag. 200

匹配 HTML Link. 201

检查 HTTP URL. 203

验证主机名... 203

在真实世界中提取 URL. 206

扩展的例子... 208

保持数据的协调性... 209

解析 CSV 文件... 213

第6章：打造高效正则表达式.... 221

典型示例... 222

稍加修改——先迈最好使的腿... 223

效率 vs 准确性... 223

继续前进——限制匹配优先的作用范围... 225

实测... 226

全面考查回溯... 228

POSIX NFA 需要更多处理... 229

无法匹配时必须进行的工作... 230

看清楚一点... 231

多选结构的代价可能很高... 231

性能测试... 232

理解测量对象... 234

PHP 测试... 234

Java 测试... 235

VB.NET 测试... 237

Ruby 测试... 238

Python 测试... 238

Tcl 测试... 239

常见优化措施... 240

有得必有失... 240

优化各有不同... 241

正则表达式的应用原理... 241

应用之前的优化措施...	242
通过传动装置进行优化...	246
优化正则表达式本身...	247
提高表达式速度的诀窍...	252
常识性优化...	254
将文字文本独立出来...	255
将锚点独立出来...	256
忽略优先还是匹配优先？具体情况具体分析...	256
拆分正则表达式...	257
模拟开头字符识别...	258
使用固化分组和占有优先量词...	259
主导引擎的匹配...	260
消除循环...	261
方法1：依据经验构建正则表达式...	262
真正的“消除循环”解法...	264
方法2：自顶向下的视角...	266
方法3：匹配主机名...	267
观察...	268
使用固化分组和占有优先量词...	268
简单的消除循环的例子...	270
消除 C 语言注释匹配的循环...	272
流畅运转的表达式...	277
引导匹配的工具...	277
引导良好的正则表达式速度很快...	279
完工...	281

总结：开动你的大脑... 281

第7章：Perl 283

作为语言组件的正则表达式... 285

Perl 的长处... 286

Perl 的短处... 286

Perl 的正则流派... 286

正则运算符和正则文字... 288

正则文字的解析方式... 292

正则修饰符... 292

正则表达式相关的 Perl 教义... 293

表达式应用场合... 294

动态作用域及正则匹配效应... 295

匹配修改的特殊变量... 299

qr/.../运算符与 regex 对象... 303

构建和使用 regex 对象... 303

探究 regex 对象... 305

用 regex 对象提高效率... 306

Match 运算符... 306

Match 的正则运算元... 307

指定目标运算元... 308

Match 运算符的不同用途... 309

迭代匹配：Scalar Context，不使用/g. 312

Match 运算符与环境的关系... 316

Substitution 运算符... 318

运算元 replacement 319

/e 修饰符... 319

应用场合与返回值... 321

Split 运算符... 321

Split 基础知识... 322

返回空元素... 324

Split 中的特殊 **Regex** 运算元... 325

Split 中带捕获型括号的 **match** 运算元... 326

巧用 **Perl** 的专有特性... 326

用动态正则表达式结构匹配嵌套结构... 328

使用内嵌代码结构... 331

在内嵌代码结构中使用 **local** 函数... 335

关于内嵌代码和 **my** 变量的忠告... 338

使用内嵌代码匹配嵌套结构... 340

正则文字重载... 341

正则文字重载的问题... 344

模拟命名捕获... 344

效率... 347

办法不只一种... 348

表达式编译、/o 修饰符、qr/.../和效率... 348

理解“原文”副本... 355

Study 函数... 359

性能测试... 360

正则表达式调试信息... 361

结语... 363

第8章：Java. 365

Java 的正则流派... 366

Java 对\p{...}和\P{...}的支持... 369

Unicode 行终结符... 370

使用 java.util.regex. 371

The Pattern.compile() Factory. 372

Pattern 的 matcher 方法... 373

Matcher 对象... 373

应用正则表达式... 375

查询匹配结果... 376

简单查找-替换... 378

高级查找-替换... 380

原地查找-替换... 382

Matcher 的检索范围... 384

方法链... 389

构建扫描程序... 389

Matcher 的其他方法... 392

Pattern 的其他方法... 394

Pattern 的 split 方法，单个参数... 395

Pattern 的 split 方法，两个参数... 396

拓展示例... 397

为 Image Tag 添加宽度和高度属性... 397

对于每个 Matcher，使用多个 Pattern 校验 HTML. 399

解析 CSV 文档... 401

Java 版本差异... 401

1.4.2和1.5.0之间的差异... 402

1.5.0和1.6之间的差异... 403

第9章：.NET. 405

.NET 的正则流派... 406

对于流派的补充... 409

使用.NET 正则表达式... 413

正则表达式快速入门... 413

包概览... 415

核心对象概览... 416

核心对象详解... 418

创建 Regex 对象... 419

使用 Regex 对象... 421

使用 Match 对象... 427

使用 Group 对象... 430

静态“便捷”函数... 431

正则表达式缓存... 432

支持函数... 432

.NET 高级话题... 434

正则表达式装配件... 434

匹配嵌套结构... 436

Capture 对象... 437

第10章：PHP.. 439

PHP 的正则流派... 441

Preg 函数接口... 443

“Pattern”参数... 444

Preg 函数罗列... 449

“缺失”的 preg 函数... 471

对未知的 Pattern 参数进行语法检查... 474

对未知正则表达式进行语法检查... 475

递归的正则表达式... 475

匹配嵌套括号内的文本... 475

不能回溯到递归调用之内... 477

匹配一组嵌套的括号... 478

PHP 效率... 478

模式修饰符 S: “研究”. 478

扩展示例... 480

用 PHP 解析 CSV.. 480

检查 tagged data 的嵌套正确性... 481

索引..... 485

前言

本书关注的是一种强大的工具——“正则表达式”。它将教会读者如何使用正则表达式解决各种问题，以及如何充分使用支持正则表达式的工具和语言。许多关于正则表达式的文档都没有介绍这种工具的能力，而本书的目的正是让读者“精通”正则表达式。

许多种工具都支持正则表达式（文本编辑器、文字处理软件、系统工具、数据库引擎，等等），不过，要想充分挖掘正则表达式的能力，还是应当将它作为编程语言的一部分。例如 Java、JScript、Visual Basic、VBScript、JavaScript、ECMAScript、C、C++、C#、elisp、Perl、Python、Tcl、Ruby、PHP、*sed* 和 *awk*。事实上，在一些用上述语言编写的程序中，正则表达式扮演了极其重要的角色。

正则表达式能够得到众多语言 and 工具的支持是有原因的：它们极其有用。从较低的层面上来说，正则表达式描述的是一串文本（a chunk of text）的特征。读者可以用它来验证用户输入的数据，或者也可以用它来检索大量的文本。从较高的层面上来说，正则表达式容许用户掌控他们自己的数据——控制这些数据，让它们为自己服务。掌握正则表达式，就是掌握自己的数据。

本书的价值

The Need for This Book

本书的第1 版写于1996年，以满足当时存在的需求。那时还没有关于正则表达式的详尽文档，所以它的大部分能力还没有被发掘出来。正则表达式文档倒是存在，但它们都立足于“低层次视角”。我认为，那种情况就好像是教一些人英文字母，然后就指望他们会说话。

第2 版与第1 版间隔了五年半的时间，这期间，互联网迅速流行起来，正则表达式的形式也有了极大的扩张，这或许并不是巧合。几乎所有工具软件和程序语言支持的正则表达式也变得更加强大和易于使用。**Perl**、**Python**、**Tcl**、**Java** 和 **Visual Basic** 都提供了新的正则支持。新出现的支持内建正则表达式的语言，例如 **PHP**、**Ruby**、**C#**，也已经发展壮大，流行开来。在这段时间里，本书的核心——如何真正理解正则表达式，以及如何使用正则表达式——仍然保持着它的重要性和参考价值。

不过，第1版已经逐渐脱离了时代，必须加以修订，才能适应新的语言和特性，也才能对应正则表达式在互联网世界中越来越重要的地位。第2 版出版于2002年，这一年的里程碑是 **java.util.regex**、**Microsoft .NET Framework** 和 **Perl 5.8**的诞生。第2 版全面覆盖了这些内容。关于第2 版，我唯一的遗憾就是，它没有提及 **PHP**。自第2 版诞生以来的4 年里，**PHP** 的重要性一直在增加，所以，弥补这一缺憾是非常迫切的。

第3 版在前面的章节中增加了 **PHP** 的相关内容，并专门为理解和应用 **PHP** 的正则表达式增加了一章全新的内容。另外，该版对 **Java** 的章节也进行了修订，做了可观的扩充，反映了 **Java1.5**和 **Java1.6**的新特性。

目标读者

Intended Audience

任何有机会使用正则表达式的人，都会对本书感兴趣。如果您还不了解正则表达式能提供的强大功能，这本书展示的全新世界将会让您受益匪浅。即使您认为自己已经是掌握正则表达式的高手了，这本书也能够深化您的认识。第1 版面世后，我时常会收到读者的电子邮件反映说“读这本书之前，我以为自己了解正则表达式，但现在我才真正了解”。

以与文本打交道为工作（如 **Web** 开发）的程序员将会发现，这本书绝对称得上是座金矿，因为其中蕴藏了各种细节、暗示、讲解，以及能够立刻投入到实用中的知识。在其他任何地方都难以找到这样丰富的细节。

正则表达式是一种思想——各种工具以各种方式(数目远远超过本书的列举)来实现它。如果读者理解了正则表达式的基本思想，掌握某种特殊的实现就是易如反掌的事情。本书关注的就是这种思想，所以其中的许多知识并不受例子中所用的工具软件和语言的束缚。

如何阅读

How to Read This Book

这本书既是教程，又是参考手册，还可以当故事看，这取决于读者的阅读方式。熟悉正则表达式的读者可能会觉得，这本书马上就能当作一本详细的参考手册，读者可以直接跳到自己需要的章节。不过，我并不鼓励这样做。

要想充分利用这本书，可以把前6章作为故事来读。我发现，某些思维习惯和思维方式的确有助于完整的理解，不过最好还是从这几章的讲解中学习它们，而不是仅仅记住其中的几张列表。

故事是这样的，前6章是后面4章——包括 Perl、Java、.NET 和 PHP——的基础。为了帮助读者理解每一部分，我交叉使用各章的知识，为了提供尽可能方便的索引，我投入了大量的精力（全书中有超过1 200处交叉引用，它们以符号加页码的形式标注）。

在读完整个故事以前，最好不要把本书作为参考手册。在开始阅读之前，读者可以参考其中的表格，例如第92页的图表，想象它代表了需要掌握的相关信息。但是，还有大量背景信息没有包含在图表中，而是隐藏在故事里。读者阅读完整个故事之后，会对这些问题有个清晰的概念，哪些能够记起来，哪些需要温习。

组织结构

Organization

全书共10章，可以从逻辑上粗略地分为三类，下面是略图：

导引

- 第1章：介绍正则表达式的基本概念。
- 第2章：考察利用正则表达式进行文本处理的过程。
- 第3章：提供对于特性和工具软件的概述以及简史。

细节

- 第4章：揭示了正则表达式的工作原理的细节。
- 第5章：利用第4章的知识，继续学习各种例子。
- 第6章：详细讨论效率问题。

特定工具的知识

- 第7章：详细讲解 Perl 的正则表达式。
- 第8章：讲解 Sun 提供的 `java.util.regex` 包。
- 第9章：讲解 .NET 的语言中立的正则表达式包。
- 第10章：讲解 PHP 中提供正则功能的 `preg` 套件。

导引部分会把完全的门外汉变成“对问题有感觉”的新手。对正则表达式有一定经验的读者完全可以快速翻阅这些章节，不过，即使是对于相当有经验的读者来说，我仍然要特别推荐第3章。

1 第1章，正则表达式入门，是为完全的门外汉准备的。我以应用相当广泛的程序 *egrep* 为例来介绍正则表达式，我也提供了我的视角：如何“理解”正则表达式，来为后面章节所包括的高级概念打下坚实的基础。即使是有经验的读者，浏览本章也会有所收获。

1 第2章，入门示例拓展，考查了支持正则表达式的程序设计语言的真实文本处理过程。附加的示例提供了后面章节详细讨论的基础，也展示了高级正则表达式调校背后的重要思考过程。为了让读者学会“正则表达式的套路”，这章出现了一个复杂问题，并讲解了两种全然不相关的工具如何分别通过正则表达式来解决它。

1 第3章，正则表达式的特性和流派概览，提供了当前经常使用的工具的多种正则表达式的概览。因为历史的混乱，当前常用的正则表达式的类型可能差异巨大。此章同时介绍了正则表达式以及使用正则表达式的工具的历史和演化历程。本章末尾也提供了“高级话题引导”。此引导是读者学习此后高级内容的路线图。

细节

The Details

了解了基础知识之后，读者需要弄明白“如何使用”及“这么做的原因”。就像“授人以渔”的典故一样，真正懂得正则表达式的读者，能够在任何时间、任何地点应用关于它的知识。

1 第4章，表达式的匹配原理，循序渐进地导入本书的核心。它从实践的角度出发，考察了正则引擎真实工作的重要的内在机制。懂得正则表达式如何处理工作细节，对读者掌握它们大有裨益。

1 第5章，正则表达式实用技巧，教育读者在高层次和实际的运用中应用知识。这一章会详细讲解常见（但复杂）的问题，目的在于拓展和深化读者对于正则表达式的认识。

1 第6章，打造高效正则表达式，考察真实生活中大多数程序设计语言提供的正则表达式的高效结果。本章运用第4章和第5章详细讲解的知识，来开发引擎的能力，并避免其中的缺陷。

特定工具的知识

Tool-Specific Information

学习完第4、5、6章的读者，不太需要知道特定的实现。不过，我还是用了4个整章来讲解4种流行的语言。

1 第7章，Perl，详细讲解了Perl的正则表达式，Perl大概是目前最流行的主要的正则表达式编程语言。在Perl中，与正则表达式相关的操作符只有四个，但它们组合出的选项和特殊情形带来了大量的程序选项——同时还有陷阱。对没有经验的开发人员来说，这种极其丰富的选项能够让他们迅速从概念转向程序，当然也可能是雷场。本章的详细介绍有助于给读者指出一条光明大道。

1 第8章，Java，详细介绍了 `java.util.regex` 包，从Java 1.4以后，它已经成为了Java语言的标准部分。本章主要关注的是Java 1.5，但也提及了它与Java 1.4.2和Java 1.6的差别。

1 第9章，.NET，是微软尚未提供的.NET正则表达式库的文档。无论使用VB.NET、C#、C++、JScript、VBScript、ECMAScript还是使用.NET组件的其他语言，本章都提供了详细内容，让读者能够充分利用.NET的正则表达式。

1 第10章，PHP，简要介绍了PHP内嵌的多个正则引擎，并详细介绍了 `preg` 正则表达式套件（`regex engine`）的类型和API，这些是由PCRE正则表达式库提供的。

体例说明

Typographical Conventions




在进行（或者谈论）详细的和复杂的文本处理时，保持精确性是很重要的。差一个空格字符，可能导致截然不同的结果，所以我会在这本书中使用下面的惯例：

1 正则表达式以「this」的形式出现。两端的符号表示“里面有一个正则表达式”，而正则表达式文字（例如用来检索的表达式）以‘this’的形式出现。有时候，省略两端的符号和单引号也不会造成歧义，此时我会省略它们。同样，屏幕截图通常以原来的样子出现，而不会用到上面两种符号。

1 在文字文本和表达式内部的省略号会被特别标出。例如，[...]表示一对尖括号，之间的内容无关紧要，而[...]表示一对尖括号，其中包含三个句点。

1 如果没有明确数字，可能很难判断“a b”之间有多少空格，所以出现在正则表达式和文字文本中的空格以“-”表示。这样“a——b”就清楚多了。

1 我使用可见的制表符，换行符和回车字符：

—	空格字符
	制表符
	换行符
	回车字符

1 有时候，我会使用下画线或有色背景高亮标注文字文本或正则表达式的一部分。下面这句话中，下画线的部分表示表达式真正匹配的部分：

Because 「cat」 matches ‘It-indicates-your-cat-is ...’ instead of the word ‘cat’, we realize...

这个例子中，下画线的部分高亮标记了表达式中添加的字符：

To make this useful, we can wrap 「Subject|Date」 with parentheses, and append a colon and a space. This yields 「(Subject|Date):-」

1 本书包含了大量的细节和例子，所以我设置了超过1 200处的交叉引用，帮助读者理解。它们通常表示为“F123”，意思是“请参阅第123页”。举个例子：“...的说明在表8-2中（F367）”。

练习

Exercises

有时候我会问个问题，帮助读者理解正在讲解的概念，尤其是在前几章这种问题更多。它们并不是摆设，我希望读者在继续阅读之前认真想想。请记住我的话，不要忽略它们的重

要意义，本书中这样的问题并不多。它们可以当作自我测试题：如果不是几句话就能说明白的问题，最好是在复习相关章节之后再继续阅读。

为了避免读者直接看到问题的答案，我使用了一点技巧：问题的答案都必须翻页才能看到。通常你必须翻过一页才能看到标着 v 的答案。这样答案在你思考问题的时候没法直接看到，但又很容易获得。

链接、代码、勘误及联系方式

Links, Code, Errata, and Contacts

写第1 版时，我发现修改书本上的 URL，保持与实际一致是件很费工夫的事情，所以，我没有在书后罗列一个 URL 附录，而是提供统一的地址：

<http://regex.info>

在这里你可以找到与正则表达式相关的链接，书中的所有代码，可检索的索引以及其他资源。本书也可能存在错误 J，所以我提供了勘误。

如果你找到书中的错误，或者仅仅是希望给我写几句话，请写邮件到：jfriedl@regex.info。

我们已尽力核验本书所提供的信息，尽管如此，仍不能保证本书完全没有瑕疵，而网络世界的变化之快，也使得本书永不过时的保证成为不可能。如果读者发现本书内容上的错误，不管是赘字、错字、语意不清，甚至是技术错误，我们都竭诚虚心接受读者指教。如果您有任何问题，请按照以下的联系方式与我们联系。

奥莱理软件（北京）有限公司

北京市 海淀区知春路49号 希格玛公寓 B 座809室

邮政编码：100080

网页：<http://www.oreilly.com.cn>

E-mail：info@mail.oreilly.com.cn

与本书有关的在线信息（包括勘误、范例程序、相关链接）如下所示。

<http://www.oreilly.com/catalog/regex3/>（原书）

<http://www.oreilly.com.cn/book.php?bn=0-596-52812-4>（中文版）

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市洪山区 吴家湾邮科院路特1号 湖北信息产业科技大厦1402室

邮政编码：430074

电话：（027）87690813 传真：（027）87690813转817

网页：<http://bv.csdn.net>

读者服务信箱：

sheguang@broadview.com.cn（读者信箱）

broadvieweditor@gmail.com（投稿信箱）

《精通正则表达式（第3版）》（即 *Mastering Regular Expression*, 3rd Edition）是一本好书。

我还记得，自己刚开始工作时，就遇到了关于正则表达式的问题（从此被逼上梁山）：若从文本中抽取 E-mail 地址，还可以用字符串来查找（先定位到@，然后向两端查找），若要抽取 URL，简单的文本查找就无能为力了。正当我一筹莫展之时，项目经理说：“可以用正则表达式，去网上找找资料吧。”抱着这根救命稻草，我搜索了之前只是听说过名字的正则表达式的资料，并打印了 `java.util.regex`（开发用的 Java）的文档来看。摸索了半天，我的感觉就是，这玩意儿，真神奇，真复杂，真好用。

此后，用到正则表达式的地方越来越多，我也越来越感觉到它的重要，然而使用起来却总感觉捉襟见肘。当时是夏天，北京非常热，我决定下班之后不再着急赶车回家，而是在公司安心看看技术文档，于是邂逅了这本 *Mastering Regular Expression*。该书原文是相当通畅易懂的，看完全书大概花了我一周的业余时间，之后便如拨云见日，感觉别有洞天——原来正则表达式可以这样用，真是奇妙，真是令人拍案叫绝。

此后我运用正则表达式便不用再看什么资料了，充其量就是查查语言的具体文档，表达式的基本模型和思路，完全是在阅读本书时确立的。也正是因为细心阅读过本书，所以有时我能以正则表达式解决某些复杂的问题。我的朋友郝培强（Tinyfool，昵称 Tiny）曾问过我一个正则表达式的问题：在 Apache 服务器的 Rewrite 规则中，要求以一个正则表达式匹配“除两个特定子域名之外的所有其他子域名”，其他人的办法都无法满足要求：要么只能匹配这两个特定的子域名，要么必须依赖程序分支才能进行判断。其实这个问题，是可以用一个正则表达式匹配的。事后，Tiny 说，看来，会用正则的人很多，但真正懂得正则的人很少。现实情况也确实如此，就我所见，不少同仁对正则表达式的运用，不外乎从网上找一些现成的表达式，套用在自己的程序中，但对到底该用几个反斜线转义，转义是在字符串级别还是表达式级别进行的，捕获型括号是否必须，表达式的效率如何，等等问题，往往都是一知半解，甚至毫无概念，在 Tiny 的问题面前，更是束手无策，一筹莫展。

就我个人来说，我所掌握的正则表达式的知识，绝大多数来自本书。正是依靠这些知识，我几乎能以正则表达式进行自己期望的任何文本处理，所以我相信，能够耐心读完这本书的读者，一定能深入正则表达式的世界，若再加以练习和思考，就能熟练地依靠它解决各种复杂的问题（其中就包括类似 Tiny 的问题）了。

去年，通过霍炬（Virushuo）的介绍，我参加了博文视点的试译活动，很幸运地获得了翻译本书的机会。有机会与大家分享这样一本好书，我深感荣幸。500多页的书，拖拖拉拉，也花了半年多的时间。虽然之前读过原著，积累了一些运用正则表达式的经验，也翻译过数十万字资料，但要尽可能准确、贴切地传达原文的阅读感觉，我仍感颇费心力。部分译文在确认理解原文的基础上，要以符合中文习惯的方式加以表述仍然颇费周折（例如，直译的“正则表达式确实容许出现这种错误”，原文的意思是“这样的错误超出了正则表达式的能力”，最后修改为“出现这样的错误，不能怪正则表达式”或“这样的问题，错不在正则表达式”）。另有部分词语，虽可译为中文，但为保证阅读的流畅，没有翻译（例如，“它包含特殊和一般两个部分，特殊部分之所以是特殊的，原因在于……”，此处 *special* 和 *normal* 是专指，故翻译为“它包含 *special* 和 *normal* 两个部分，*special* 部分之所以得名，原因在于……”），这样的处理，相信不会影响读者的理解。

在本书翻译结束之际，我首先要感谢霍炬，他的引荐让我获得了翻译这本书的机会；还要感谢博文视点的周筠老师，她谨慎严格的工作态度，时刻提醒我不能马虎对待这本经典之作；还有本书的责编晓菲，她为本书的编辑和校对做了大量细致而深入的工作。

另外我还要感谢东北师范大学文学学院的王确老师，在我求学期间，王老师给予我诸多指点，离校时间愈长，愈是怀念和庆幸那段经历，可以说，没有与他的相识，便没有我的今天。

最后我还要感谢我的女友侯芳，她的陪伴，让我能够保持好的心情，全身心地投入到翻译中去。

本书是讲授正则表达式的经典之作，翻译过程中，我虽力求把握原文，语言通畅，但翻译中的错误或许是在所难免的，对此本人愿负全部责任。希望广大读者发现错误能及时与我和出版社联系以便再版时修正，或是以勘误的形式公布出来以惠及其他读者。如果读者有任何想法或建议，欢迎给我写信，我的邮件地址是：yusheng.regex@gmail.com。

如今正则表达式已经成为几乎所有主流编程语言中的必备元素：Java、Perl、Python、PHP、Ruby……莫不如此，甚至功能稍强大一些的文本编辑工具，都支持正则表达式。尤其是在 Web 兴起之后，开发任务中的一大部分甚至全部，都是对字符串的处理。相比简单的字符串比较、查找、替换，正则表达式提供了强大得多的处理能力（最重要的是，它能够处理“符合某种抽象模式”的字符串，而不是固化的、具体的字符串）。熟练运用它们，能够节省大量的开发时间，甚至解决一些之前看来是 **mission impossible** 的问题。

本书是讲解正则表达式的经典之作。其他介绍正则表达式的资料，往往局限于具体的语法和函数的讲解，于语法细节处着墨太多，忽略了正则表达式本身。这样，读者虽然对关于正则表达式的具体规定有所了解，但终究是只见树木不见森林，遇上复杂的情况，往往束手无策，举步维艰。而本书自第1版开始便着力于教会读者“以正则表达式来思考（**think regular expression**）”，向读者讲授正则表达式的精髓（正则表达式的各种流派、匹配原理、优化原则，等等），而不拘泥于具体的规定和形式。了解这些精髓，再辅以具体操作的文档，读者便可做到“胸中有丘壑，下笔如有神”；即便问题无法以正则表达式来解决，读者也能很快作出判断，而不必盲目尝试，徒费工夫。

不了解正则表达式的读者，可循序渐进，依次阅读各章，即便之前完全未接触过正则表达式，读过前两章，也能在心中描绘出概略的图谱。第3、4、5、6章是本书的重点，也是核心价值所在，它们分别介绍了正则表达式的特性和流派、匹配原理、实用诀窍以及调校措施。这样的知识与具体语言无关，适用于几乎所有的语言和工具（当然，如果使用 DFA 引擎，第6章的价值要打个折扣），所谓“大象无形”，便是如此。读者如能仔细研读，悉心揣摩，之后解决各种问题时，必定获益匪浅。第7、8、9、10章分别讲解了 Perl、Java、.NET、PHP 中正则表达式的用法，看来类似参考手册，其实是对前面4章知识的包装，将抽象的知识辅以具体的语言规定，以具体的形式表现出来。所以，心急的读者，在阅读这些章节之前，最好先通读第3、4、5、6章，以便更好地理解其中的逻辑和思路。

相信仔细阅读完本书的读者，定会有登堂入室的感觉。不但能见识到正则表达式各种令人眼花缭乱的特性，更能够深入了解表达式、匹配、引擎背后的原理，从而写出复杂、神奇而又高效的正则表达式，快速地解决工作中的各种问题。

2007年6月于北京

想象一下这幅图景：你需要检索某台 Web 服务器上的页面中的重复单词（例如“this this”），进行大规模文本编辑时，这是一项常见的任务。你的任务是，给出满足下面条件的办法：

- 1 检查多个文件，报告包含重复单词的行，高亮标记每个重复单词（使用标准 ANSI 的转义字符序列（`escape sequence`）），同时必须显示这行文字来自哪个文件。

- 1 跨行查找，即使两个单词一个在某行末尾而另一个在下一行的开头，也算重复单词。

- 1 进行不区分大小写的查找，例如‘The the...’，重复单词之间可以出现任意数量的空白字符（空格符、制表符、换行符之类）（译注1）。

- 1 查找用 HTML tag 分隔的重复单词。HTML tag 用于标记互联网页上的文本，例如，粗体单词是这样表示的：‘...it is very very important...’。

这些问题并不容易解决，但又不能不解决。我在写作本书的手稿时，曾用一个工具来检查已经写好的部分，我惊奇地发现，其中竟有那么多的重复单词。能够解决这种问题的编程语言有许多，但是用支持正则表达式的语言来处理会相当简单。

正则表达式（**Regular Expression**）是强大、便捷、高效的文本处理工具。正则表达式本身，加上如同一门袖珍编程语言的通用模式表示法（**general pattern notation**），赋予使用者描述和分析文本的能力。配合上特定工具提供的额外支持，正则表达式能够添加、删除、分离、叠加、插入和修整各种类型的文本和数据。

正则表达式的使用难度只相当于文本编辑器的搜索命令，但功能却与完整的文本处理语言一样强大。本书将向读者展示正则表达式提高生产率的诸多办法。它会教导读者如何学会用正则表达式来思考（**think regular expressions**），以便于掌握它们，充分利用它们的强大功能。

如果使用当今流行的程序设计语言，解决重复单词问题的完整程序可能仅仅只需要几行代码。使用一个正则表达式的搜索和替换命令，读者就可以查找文档中的重复单词，并把它们标记为高亮。加上另一个，你可以删除所有不包含重复单词的行（只留下需要在结果中出现的行）。最后，利用第三个正则表达式，你可以确保结果中的所有行都以它所在文件的名字开头。在下一章里，我们会看到用 Perl 和 Java 编制的程序。

宿主语言（例如 Perl、Java 以及 VB.NET）提供了外围的处理支持，但是真正的能力来自正则表达式。为了驾驭这种语言，满足自己的需求，读者必须知道如何构建正则表达式，才能识别符合要求的文本，同时忽略不需要的文本。然后，就可以把表达式和语言支持的构建方式结合起来，真正处理这些文本（加入合适的高亮标记代码，删除文本，修改文本，等等）。

解决问题

Solving Real Problems

掌握正则表达式，可能带来超乎你之前想象的文本处理能力。每一天，我都依靠正则表达式解决各种大大小小的问题（通常的情况是，问题本身并不复杂，但没有正则表达式就成了大问题）。

要说明正则表达式的价值，可以举一个用正则表达式解决大而重要的问题的例子，但是它不一定能代表正则表达式在平时解决的那些“不值一提”（*uninteresting*）的问题。这里的“不值一提”是指这类问题并不能成为谈资，可是不解决它们，你就没法继续干活。

举个简单的例子，我需要检查许多文件（事实上，本书的手稿存放在70个文件中），确保每一行中‘*SetSize*’出现的次数与‘*ResetSize*’的一样多。为了应付复杂的情况，我还需要考虑大小写的情况（举例来说，‘*setSize*’也算做‘*SetSize*’）。人工检查32 000行文字显然不现实。

即便使用文本编辑器的“单词查找”功能，也不够方便，尤其是对所有文件进行同样的操作，何况还需要考虑所有可能的大小写情况。

正则表达式就是解决这个问题的灵丹妙药。只需要一个简单的命令，我就能够检查所有的文件，获得我需要知道的结果。时间是：写命令大概15秒，检索所有的数据实际只花了2秒。这真是棒极了（如果您有兴趣知道我是如何做的，不妨现在就翻到第36页）！

再举一个例子，我曾帮助一个朋友处理远端机器上的某些 E-mail，他希望我把他邮箱文件中的消息作为列表发送给他。我可以把整个文件导入文本编辑器，手工删除所有信息，只留下邮件头中的几行，作为内容的列表。尽管文件不是很大，连接速度也不算慢，这样的任务还是很耗费时间而且很乏味。而且，窥见他的邮件正文，也令我尴尬。

正则表达式再一次提供了帮助！我用一个简单的命令（使用本章稍后提到的一个常用工具 *egrep*）显示每封邮件的 From:和 Subject:字段。为了告诉 *egrep* 我需要提取哪些行，我使用了正则表达式「`^(From|Subject):`」。

朋友得到这个列表之后，让我找一封特殊的（5 000行！）邮件。使用文本编辑器或者邮件系统来提取一封邮件无疑非常耗时。相反，我借助另一个工具（叫做 *sed*），同样使用正则表达式来描述文件中我需要的内容。这样，我能迅速而方便地提取和发送需要的邮件。

使用正则表达式节省下来的时间或许并不能让人“激动”，但总比把时间消耗在文本编辑器中要好。如果我不知道有正则表达式这种玩意儿，根本就不会想到还有别的解决办法。所以，这个故事告诉我们，正则表达式和相关的工具能够让我们以可能未曾想过的方式来解决

问题。

一旦掌握了正则表达式，你就会知道到它简直是工具中的无价之宝，你也难以想象之前那些没有正则表达式的日子是怎么度过的（注1）。

全面掌握正则表达式是很有用的。本书提供了掌握这种技能所需要的信息，我同时也希望，这本书也提供了促使你学习的动机

作为编程语言的正则表达式

Regular Expressions as a Language

如果没有正则表达式相关经验，读者可能无法理解上个例子中正则表达式「`^(From|Subject):`」的意义，但是这个表达式并没有什么神奇之处。其实魔术本身也不神奇，只是缺乏训练的普通观众不明白魔术师掌握的那些技巧而已。如果你也懂得如何在手中藏一张牌，那么，熟练之后，你也可以“变魔术”。外语也是这样——一旦掌握了一门外语，你就不会觉得它像天书了。

以文件名做类比

The Filename Analogy

选择这本书的读者，大概对“正则表达式”多少有点认识。即便没有，也应该熟悉其中的基本概念。

我们都知道，*report.txt* 是一个文件名，但是，如果你用过 Unix 或者 DOS/Windows 的话，就会知道“*.txt”能够用来选择多个文件。在此类文件名（称为“文件群组”file globs 或者“通配符”wildcards）中，有些字符具有特殊的意义。星号表示“任意文本”，问号表示“任意单个字符”。所以，文件群组“*.txt”以能够匹配字符的「*」符号开头，以普通文字「.txt」结尾，所以，它的意思是：选择以任意文本开头，以.txt 结尾的所有文件。

大多数系统都提供了少量的附加特殊字符（additional special characters），但是，总的来说，这些文件名模式（filename patterns）的表达能力还很有限。不过，因为这类问题的领域很狭窄——只涉及文件名，所以这算不上缺陷。

不过，处理普通的文本就没有这么简单了。散文、诗、程序代码、报表、HTML、表格、单词表……到你想得出的任何文本。如果某种特殊的需求足够专业，例如“选择文件”，我们可以发明一些特殊的办法和工具来解决问题。不过，近年来，一种“通用的模式语言”

（generalized pattern language）已经发展起来，它功能强大，描述能力也很强，可以用来解决各种问题。不同的程序以不同的方式来实现和使用这种语言，但是综合来说，这种功能强大的模式语言和模式本身被称为“正则表达式”（regular expression）。

以语言做类比

The Language Analogy

完整的正则表达式由两种字符构成。特殊字符（special characters，例如文件名例子中的*）称为“元字符”（metacharacters），其他为“文字”（literal），或者是普通文本字符（normal text characters）。正则表达式与文件名模式（filename pattern）的区别就在于，正则表达式的元字符提供了更强大的描述能力。文件名模式只为有限的需求提供了有限的元字符，但是正则表达式“语言”为高级应用提供了丰富而且描述力极强的元字符。

为了便于理解，我们可以把正则表达式想象为普通的语言，普通字符对应普通语言中的单词，而元字符对应语法。根据语言的规则，按照语法把单词组合起来，就会得到能传达思想的文本。在 E-mail 的例子中，我用正则表达式「`^(From|Subject):`」来寻找以‘From:’或者‘Subject:’开头的行。下画线标注的就是特殊字符，稍后我们将解释它们的含义。

就像学习任何一门外语一样，第一眼看上去，正则表达式很不好理解。这也是那些对它只有粗浅了解或者根本不了解的人觉得正则表达式很神奇的原因。但是，就像学日语的人很快就能理解正規表現は簡単だよ！（注2）一样，读者很快也能够彻底明白下面这个正则表达式的含义：

```
s!<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>!<inet>$1</inet>!
```

这个例子取自一个 Perl 脚本，我的编辑器用它来修改手稿。手稿的作者错误地使用了<emphasis>这个 tag 来标注 IP 地址（类似209.204.146.22这样由数字和点号构成的字符串）。其中的奥妙就在于使用 Perl 的文本替换命令，使用：

```
「<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>」
```

把 IP 地址两端的 tag 替换为<inet>，而不改动其他的<emphasis>标签。在后面的章节中，读者会了解这个表达式的构造细节，然后就能按照自己的需求，在自己的应用程序或者开发语言中应用这些技巧。

本书的目的

你或许不需要重复把<emphasis>替换为<inet>的工作，不过很可能需要解决“把这些文字替换为那些文字”的问题。本书的目的不是提供具体问题的解决办法，而是教会读者利用正则表达式的知识框架

The Regular-Expression Frame of Mind

我们将会看到，完整的正则表达式由小的构建模块单元（building block unit）组成。每个单独的构建模块都很简单，不过因为它们能够以无穷多种方式组合，将它们结合起来实现特殊目标必须依靠经验。所以，本章提供了有关正则表达式的若干概念的总体描述。这一章并没有艰深的内容，而是为本书其余章节的知识打下基础，在深入探索正则表达式之前，把相关事宜阐释清楚。

某些例子看起来可能有点无聊（因为它们确实无聊），但它们代表了一类需要完成的任务，只是读者目前可能还没有意识到。即使觉得每个例子的意义都不大也不必担心，慢慢理解其中的道理就好。这就是本章的目的。

对于有部分经验的读者

If You Have Some Regular-Expression Experience

如果读者已经熟悉正则表达式，这些综述便没有太大价值，但务必不要忽略它们。你或许明白某些元字符的基本意义，但某些思维和看待正则表达式的方式可能是你不了解的。

就像真正懂演奏和仅仅会弹奏之间差别迥异一样，了解正则表达式和真正理解正则表达式并不是一回事。某些内容可能会重复读者已经了解的知识，但方式可能与之前的不同，而且这些方式正是真正理解正则表达式的第一步。

检索文本文件：Egrep

Searching Text Files: Egrep

文本检索是正则表达式最简单的应用之一——许多文本编辑器和文字处理软件都提供了正则表达式检索的功能。最简单的就是 *egrep*。在指定了正则表达式和需要检索的文件之后，*egrep* 会尝试用正则表达式来匹配每个文件的每一行，并显示能够匹配的行。

许多系统——例如 DOS、MacOS、Windows、Unix 等等——都对应有免费提供的 *egrep*。在本书的网页 <http://regex.info> 上可以找到获得对应读者操作系统的 *egrep* 拷贝的链接。

回到第3页的 E-mail 的例子，真正用来从 E-mail 文件中提取结果的命令如图1-1所示。*egrep* 把第一个命令行参数视为一个正则表达式，剩下的参数作为待搜检索的文件名。注意，图1-1中的单引号并不是正则表达式的一部分，而是根据 **command shell** 需要添加的（注3）。使用 *egrep* 时，我通常用单引号来包围正则表达式。如果要在支持对正则表达式提供了完整支持的程序设计语言中使用正则表达式——这是下一章开头的内容，重要的问题是知道特殊字符有哪些，具体文本是什么，针对什么对象（什么表达式，什么工具软件），以及按何种顺序解释这些字符。

我们马上就能明白，这个正则表达式的各个部分都是什么意思，但已经知道某些字符具有特殊含义的读者或许能够猜出大概了。在这里，「^」和「|」都是正则表达式的元字符，它们与其他字符结合起来，实现我们期望的功能。

如果一个正则表达式不包括任何 *egrep* 支持的元字符，它就成了一个简单的“纯文本”检索。例如，在一个文件中检索「cat」，会显示任何包含 c-a-t 这3个连续字母的行。例如，它包括所有出现了 **vacation** 的行。

即便这行文本中不包含单词 **cat**，**vacation** 中包含的 c-a-t 序列仍然符合匹配条件。如果某行中包含 **vacation**，*egrep* 就会把它显示出来。关键就在于，此处进行的正则表达式搜索不是基于“单词”的——*egrep* 能够理解文件中的字节和行，但它完全不理解英语（或者其他任何语言）的单词、句子、段落，或者是其他复杂概念。

Egrep 元字符

Egrep Metacharacters

现在我们来查看 *egrep* 中支持正则表达式功能的元字符。我会用几个例子来简要介绍它们，把详细的例子和描述留到后面的章节。

印刷体例 在开始之前，请务必回顾前言第 *V* 页上解释的体例说明。本书使用了一些新的文字形式，所以某些体例读者初次接触可能并不熟悉。

行的起始和结束

Start and End of the Line

或许最容易理解的元字符就是脱字符号「^」和美元符号「\$」了，在检查一行文本时，「^」

代表一行的开始，「\$」代表结束。我们曾经看到，正则表达式「cat」寻找的是一行文本中任意位置的 c-a-t，但是「^cat」只寻找行首的 c-a-t——「^」用来把匹配文本（这个表达式的其他部分匹配的字符）“锚定”（anchor）在这一行的开头。同样，「cat\$」只寻找位于行末的 c-a-t，例如以 scat 结尾的行。

读者最好能养成按照字符来理解正则表达式的习惯。例如，不要这样：

「^cat」匹配以 cat 开头的行

而应该这样理解：

「^cat」匹配的是以 c 作为一行的第一个字符，紧接一个 a，紧接一个 t 的文本。

这两种理解的结果并无差异，但按照字符来解读更易于明白新遇到的正则表达式的内部逻辑。*egrep* 会如何解释「^cat\$」、「^\$」和单个的「^」呢？v 请翻到下一页查看答案。

脱字符号和美元符号的特别之处就在于，它们匹配的是一个位置，而不是具体的文本。当然，有很多方式可以匹配具体文本。在正则表达式中，除了使用「cat」之类的普通字符，还可以使用下面几节介绍的元字符。

字符组

Character Classes

匹配若干字符之一

如果我们需要搜索的是单词“grey”，同时又不确定它是否写作“gray”，就可以使用正则表达式结构体（construct）「[...]」。它容许使用者列出在某处期望匹配的字符，通常被称作字符组（character class（译注2））。「e」匹配字符 e，「a」匹配字符 a，而正则表达式「[ea]」能匹配 a 或者 e。所以，「gr[ea]y」的意思是：先找到 g，跟着是一个 r，然后是一个 a 或者 e，最后是一个 y。我很不擅长拼写，所以总是用正则表达式从一大堆英文单词中找到正确的拼写。我经常使用的一个正则表达式是「sep[ea]r[ea]te」，因为我从来都记不住这个单词到底是写作“seperate”，“separate”，“separete”，还是别的什么样子。匹配的结果的就是正确的拼法，而正则表达式就是我的领路人。

请注意，在字符组以外，普通字符（例如「gr[ae]y」中的「g」和「r」）都有“接下来是（and then）”的意思——“首先匹配「g」，接下来是「r」……”。这与字符组内部的情况是完全相反的。字符组的内容是在同一个位置能够匹配的若干字符，所以它的意思是“或”。

来看另一个例子，我们还必须考虑单词的第一个字母为大写的情况，例如「[Ss]mith」。请记住，这个表达式仍然能够匹配内嵌在其他单词里头的 smith（或者是 Smith），例如 blacksmith。在综述阶段，我不打算为这种情况费太多笔墨，但是这确实是某些新手遇到的问题根源。等了解了更多的元字符以后，我会介绍一些办法来解决单词嵌套的问题。

在一个字符组中可以列举任意多个字符。例如「[123456]」匹配1到6中的任意一个数字。这个字符组可以作为「<H[123456]>」的一部分，用来匹配<H1>、<H2>、<H3>等等。在搜索 HTML 代码的头文件时这非常有用。

在字符组内部，字符组元字符（character-class metacharacter）「-」（连字符）表示一个范围：「<H[1-6]>」与「<H[123456]>」是完全一样的。「[0-9]」和「[a-z]」是常用的匹配数字和小写字母的简便方式。多重范围也是容许的，例如「[0123456789abcdefABCDEF]」可以写作「[0-9a-fA-F]」（或者也可以写作「[A-Fa-f0-9]」，顺序无所谓）。这3个正则表达式非常适用于处

理十六进制数字。我们还可以随心所欲地把字符范围与普通文本结合起来：「[0-9A-Z_!?.]」能够匹配一个数字、大写字母、下画线、惊叹号、点号，或者是问号。

请注意，只有在字符组内部，连字符才是元字符——否则它就只能匹配普通的连字符号。其实，即使在字符组内部，它也不一定就是元字符。如果连字符出现在字符组的开头，它表示的就只是一个普通字符，而不是一个范围。同样的道理，问号和点号通常被当作元字符处理，但在字符组中则不是如此（说明白一点就是，「[0-9A-Z_!?.]」里面，真正的特殊字符就只有那两个连字符）。

分析「^cat\$」、「^\$」和「^」

v 第8页问题的答案

「^cat\$」 文字意义：匹配的条件是，行开头（显然，每一行都有开头），然后是字母 c-a-t，然后是行末尾。

应用意义：只包含 cat 的行——没有多余的单词、空白字符.....只有‘cat’。

「^\$」 文字意义：匹配的条件是，行开头，然后就是行末尾。

应用意义：空行（没有任何字符，包括空白字符）。

「^」 文字意义：匹配条件是行的开头。

应用意义：无意义！因为每一行都有开头，所以每一行都能匹配——空行也不例外。

不妨把字符组看作独立的微型语言。在字符组内部和外部，关于元字符的规定（哪些是元字符，以及它们的意义）是不同的。

我们很快就会看到更多的例子。

排除型字符组

用「[^...]」取代「[...]」，这个字符组就会匹配任何未列出的字符。例如，「[^1-6]」匹配除了1到6以外的任何字符。这个字符组中开头的「^」表示“排除（negate）”，所以这里列出的不是希望匹配的字符，而是不希望匹配的字符。

读者可能注意到了，这里的^和第8页的表示行首的脱字符是一样的。字符确实相同，但意义截然不同。英语里的“wind”，根据情境的不同，可能表示一阵强烈的气流（风），也可能表示给钟表上发条；元字符也是如此。我们已经看过用来表示范围的连字符的例子。只有在字符组内部（而且不是第一个字符的情况下），连字符才能表示范围。在字符组外部，^表示一个行锚点（line anchor），但是在字符组内部（而且必须是紧接在字符组的第一个方括号之后），它就是一个元字符。请不要担心——这就是最复杂的情况，接下来的内容比这简单。

来看另一个例子，我们需要在一堆英文单词中搜索出一些特殊的单词：在这些单词中，字母 q 后面的字母不是 u。用正则表达式来表示，就是「q[^u]」。用这个正则表达式来搜索我手头的数据库，确实得到了一些结果，但显然不多，其中还有些是我没见过的英文单词。

下面是结果（我输入的命令用粗体表示）：

```
% egrep 'q[^u]' word.list
```

Iraqi
Iraqian
miqra
qasida
qintar
qoph
zaqqum%

其中有两个单词值得注意：伊拉克“Iraq”和澳大利亚航空公司的名字“Qantas”。尽管它们都在 *word.list* 文件中，但都不包含在 *egrep* 结果中。为什么呢？v 请动动脑筋，然后翻到下一页来检查你的答案。

请记住，排除型字符组表示“匹配一个未列出的字符（match a character that's not listed）”，而不是“不要匹配列出的字符（don't match what is listed）”。这两种说法看起来一样，但是 *Iraq* 的例子说明了其中的细微差异。有一种简单的理解排除型字符组的办法，就是把它们看作普通的字符组，里面包含的是除了“排除型字符组中所有字符”以外的字符。

用点号匹配任意字符

Matching Any Character with Dot

元字符「`.`」（通常称为点号 *dot* 或者小点 *point*）是用来匹配任意字符的字符组的简便写法。如果我们需要在表达式中使用一个“匹配任何字符”的占位符（placeholder），用点号就很方便。例如，如果我们需要搜索03/19/76、03-19-76或者03.19.76，不怕麻烦的话用一个明确容许「`/`」、「`-`」、「`.`」的字符组来构建正则表达式，例如「`03[-./]19[-./]76`」。也可以简单地尝试「`03.19.76`」。

读者第一次接触这个表达式时，可能还不清楚某些情况。在「`03[-./]19[-./]76`」中，点号并不是元字符，因为它们在字符组内部（记住，在字符组里面和外面，元字符的定义和意义是不一样的）。这里的连字符同样也不是元字符，因为它们都紧接在「`[`」或者「`^`」之后。如果连字符不在字符组的开头，例如「`[-./]`」，就是用来表示范围的，在本例中就是错误的用法。

测验答案

v 第11页问题的答案

为什么「`q[^u]`」无法匹配‘Qantas’或者‘Iraq’？

Qantas 无法匹配的原因是，正则表达式要求小写 *q*，而 Qantas 中的 *Q* 是大写的。如果我们改用 `Q[^u]`，就能匹配 Qantas，但是其他单词又不在结果中了，因为它们不包括大写 *Q*。`[Qq][^u]`则能找到上面所有的单词。

Iraq 的例子有点迷人。正则表达式要求 *q* 之后紧跟一个 *u* 以外的字符，这就排除

了 `q` 处在行尾的情况。通常来说，文本行的结尾都有一个换行字符，但是我首先没有提到（非常抱歉）`egrep` 会在检查正则表达式之前把这些换行符去掉，所以在行尾的 `q` 之后，没有能够匹配 `u` 以外的字符。

请不要因此灰心丧气（注4）。我向你保证，如果 `egrep` 保留换行符（许多其他软件会保留这些符号），或者 `Iraq` 后紧接着空格或者其他单词，这一行就能匹配。理解工具软件的细节固然很重要，但现在我只希望读者能通过这个例子认识到：一个字符组，即使是排除型字符组，也需要匹配一个字符。

在「03.19.76」中，点号是元字符——它能够匹配任意字符（包括我们期望的连字符、句号和斜线）。不过，我们也需要明白，点号可以匹配任何字符，所以这个正则表达式也能够匹配下面的字符串：‘lottery numbers: 19 203319 7639’

所以，「03[-./]19[-./]76」更加精确，但是更难读，也更难写。「03.19.76」更容易理解，但是不够细致。我们应该选择哪一个呢？这取决于你对需要检索的文本的了解，以及你需要达到的准确程度。一个重要但常见的问题是，书写正则表达式时，我们需要在对欲检索文本的了解程度与检索精确性之间求得平衡。例如，如果我们知道，针对某个检索文本，「03.19.76」这个正则表达式基本不可能匹配不期望的结果，使用它就是合理的。要想正确使用正则表达式，清楚地了解目标文本是非常重要的。

多选结构

Alternation

匹配任意子表达式

「`|`」是一个非常简捷的元字符，它的意思是“或”（or）。依靠它，我们能够把不同的子表达式组合成一个总的表达式，而这个总的表达式又能够匹配任意的子表达式。假如「`Bob`」和「`Robert`」是两个表达式，但「`Bob|Robert`」就是能够同时匹配其中任意一个的正则表达式。在这样的组合中，子表达式称为“多选分支（alternative）”。

回头来看「`gr[ea]y`」的例子，有意思的是，它还可以写作「`grey|gray`」，或者是「`gr(a|e)y`」。后者用括号来划定多选结构的范围（正常情况下，括号也是元字符）。请注意，「`gr[a|e]y`」不符合我们的要求——在这里，「`|`」只是一个和「`a`」与「`e`」一样的普通字符。

对表达式「`gr(a|e)y`」来说，括号是必须的，因为如果没有括号，「`graley`」的意思就成了“「`gra`」或者「`ey`」”，而这不符合我们的要求。多选结构可以包括很多字符，但不能超越括号的界限。另一个例子是「`(First|1st)-[Ss]treet`」（注5）。事实上，因为「`First`」和「`1st`」都以「`st`」结尾，我们可以把这个结合体缩略表示为「`(Fir|1)st-[Ss]treet`」。这样可能不容易看得清楚，但我们知道「`(First|1st)`」与「`(fir|1)st`」表示的是同一个意思。

下面是一些用多选结构来拼写我名字的例子。这3个表达式是一样的，请仔细比较：

「`Jeffrey|Jeffery`」

「`Jeff(rey|ery)`」

「Jeff(re|er)y」

英国拼写法如下：

「(Geoff|Jeff)(rey|ery)」

「(Geo|Je)ff(rey|ery)」

「(Geo|Je)ff(re|er)y」

最后要注意的是，这3个表达式其实与下面这个更长（但是更简单）的表达式是等价的：「Jeffrey|Geoffery|Jeffery|Geoffrey」。它们只是“殊途同归”而已。

「gr[ea]y」与「gr(a|e)y」的例子可能会让人觉得多选结构与字符组没太大的区别，但是请留神不要混淆这两个概念。一个字符组只能匹配目标文本中的单个字符，而每个多选结构自身都可能是完整的正则表达式，都可以匹配任意长度的文本。

字符组基本可以算是一门独立的微型语言（例如，对于元字符，它们有自己的规定），而多选结构是正则表达式语言“主体（main regular expression language）”的一部分。你将会发现，这两者都非常有用。

同样，在一个包含多选结构的表达式中使用脱字符和美元符的时候也要小心。比较「^From|Subject|Date:-」和「^(From|Subject|Date):-」就会发现，虽然它们看起来与之前的 E-mail 的例子很相似，匹配结果（即它们的用处）却大不相同。第一个表达式由3个多选分支构成，所以它能匹配「^From」或者「Subject」或者「Date:-」，实用性不大。我们希望在每一个多选分支之前都有脱字符，之后都有「:-」。所以应该使用括号来“限制”（constrain）这些多选分支：

「^(From|Subject|Date):-」

现在3个多选分支都受括号的限制，所以，这个正则表达式的意思是：匹配一行的起始位置，然后匹配「^From」、「Subject」或「Date」中的任意一个，然后匹配「:-」，所以，它能够匹配的文本是：

1) 行起始，然后是 F-r-o-m，然后是「:-」，

或者 2) 行起始，然后是 S-u-b-j-e-c-t，然后是「:-」，

或者 3) 行起始，然后是 D-a-t-e，然后是「:-」。

简单点说，就是匹配以「From:-」、「Subject:-」或者「Date:-」开头的文本行，在提取 E-mail 文件中的信息时这很有用。

下面是一个例子：

```
% egrep '^(From|Subject|Date):' mailbox
```

```
From: elvis@tabloid.org (The King)
```

```
Subject: be seein' ya around
```

```
Date: Mon, 23 Oct 2006 11:04:13
```

From: The Prez <president@whitehouse.gov>

Date: Wed, 25 Oct 2006 8:36:24

Subject: now, about your vote...

忽略大小写

Ignoring Differences in Capitalization

E-mail header 的例子很适合用来说明不区分大小写（case-insensitive）的匹配的概念。E-mail header 中的字段类型（field type）通常是以大写字母开头的，例如“Subject”和“From”，但是 E-mail 标准并没有对大小写进行严格的规定，所以“DATE”或者“from”也是合法的字段类型。但是，之前使用的正则表达式无法处理这种情况。

一种办法是用「[Ff][Rr][Oo][Mm]」取代「From」，这样就能匹配任何形式的“from”，但缺点之一就是很不方便。幸好，我们有一种办法告诉 *egrep* 在比较时忽略大小写，也就是进行不区分大小写的匹配，这样就能忽略大小写字母的差异。

该功能并不是正则表达式语言的一部分，却是许多工具软件提供的有用的相关特性。*egrep* 的命令行参数“-i”表示进行忽略大小写的匹配。把-i 写在正则表达式之前：

```
% egrep -i '^(From|Subject|Date): ' mailbox
```

结果除了包括之前的内容外，还包含这一行：

SUBJECT: MAKE MONEY FAST

我使用-i 参数的频率很高（也许与第12页的注解有关），所以我推荐读者记住它。在下面的章节中我们还会见到其他的简捷特性。

单词分界符

Word Boundaries

使用正则表达式时经常会遇到的一个问题，期望匹配的“单词”包含在另一个单词之中。在 cat、gray 和 Smith 的例子中，我曾提到过这个问题。不过，某些版本的 *egrep* 对单词识别提供了有限的支持：也就是单词分界符（单词开头和结束的位置）的匹配。

如果你的 *egrep* 支持“元字符序列（metasequences）”「\<」和「\>」，就可以使用它们来匹配单词分界的位置。可以把它们想象为单词版本的「^」和「\$」，分别用来匹配单词的开头和结束位置。就像作为行锚点的脱字符和美元符一样，他们锚定了正则表达式的其他部分，但在匹配过程中并不对应到任何字符。表达式「\<cat\>」的意思是“匹配单词的开头位置，然后是 c-a-t 这3个字母，然后是单词的结束位置”。更直接点说就是“匹配 cat 这个单词”。如果读者愿意，也可以用「\<cat」和「cat\>」来匹配以 cat 开头和结束的单词。

请注意，「<」和「>」本身并不是元字符——只有当它们与斜线结合起来的时候，整个序列才具有特殊意义。这就是我称其为“元字符序列”的原因。重要的是它们的特殊意义，而不是字符的个数，所以我说的“元字符”和“元（字符）序列”大多数时候是等价的。

请记住，并不是所有版本的 *egrep* 都支持单词分界符，即使是支持的版本也不见得聪明到能“认得出”英语单词。“单词的起始位置”只不过是一系列字母和数字符号（**alphanumeric characters**）开始的位置，而“结束位置”就是它们结尾的地方。下一页的图1-2说明了一行简单文本中的单词分界符。

（*egrep* 认定的）单词开头位置用向上的箭头标识，单词结束位置用向下的箭头标识。我们看到，“单词的开始和结束”准确地说是“字母数字符号的开始和结束”，不过这样说太麻烦了。

图1-2：“单词”的起始和结束位置

小结

In a Nutshell

表1-1总结了我们已经介绍过的元字符。

表1-1：至今为止所见的元字符小结

元字符	名 称	匹配对象
.	点号	单个任意字符
[...]	字符组	列出的任意字符
[^...]	排除型字符组	未列出的任意字符
^	脱字符	行的起始位置
\$	美元符	行的结束位置
\<	反斜线-小于	单词的起始位置（某些版本的 <i>egrep</i> 可能不支持）
\>	反斜线-大于	单词的结束位置（某些版本的 <i>egrep</i> 可能不支持）
	竖线	匹配分隔两边的任意一个表达式
(...)	括号	限制竖线的作用范围，其他功能下文讨论

另外还有几点需要注意：

- 1 在字符组内部，元字符的定义规则（及它们的意义）是不一样的。例如，在字符组外部，点号是元字符，但是在内部则不是如此。相反，连字符只有在字符组内部（这是普遍情况）才是元字符，否则就不是。脱字符在字符组外部表示一个意思，在字符组内部紧接着[时表示另一个意思，其他情况下又表示别的意思。
- 1 不要混淆多选项和字符组。字符组「[abc]」和多选项「(a|b|c)」固然表示同一个意

思，但是这个例子中的相似性并不能推广开来。无论列出的字符有多少，字符组只能匹配一个字符。相反，多选项可以匹配任意长度的文本，每个多选项可能匹配的文本都是独立的，例如「\<(1,000,000|million|thousand-thou)\>」。不过，多选项没有像字符组那样的排除功能。

1 排除型字符组是表示所有未列出字符的字符组的简便方法。因此，「`[^x]`」的意思并不是“只有当这个位置不是 `x` 时才能匹配”，而是说“匹配一个不等于 `x` 的字符”。其中的差别很细微，但很重要。例如，前面的概念可以匹配一个空行，而「`[^x]`」则不行。

1 `-i` 参数规定在匹配时不区分大小写（F15）（注6）。

1 目前介绍过的知识都很有用，但“可选项（optional）”和“计数（counting）”元素更重要，下文将马上介绍。

可选项元素

Optional Items

现在来看 `color` 和 `colour` 的匹配。它们的区别在于，后面的单词比前面的多一个 `u`，我们可以用「`colou?r`」来解决这个问题。元字符「`?`」（也就是问号）代表可选项。把它加在一个字符的后面，就表示此处容许出现这个字符，不过它的出现并非匹配成功的必要条件。

「`u?`」这个元字符与我们之前看到的元字符都不相同，它只作用于之前紧邻的元素。因此，「`colou?r`」的意思是：「`c`」，然后是「`o`」，然后是「`l`」，然后是「`o`」，然后是「`u?`」，最后是「`r`」。

「`u?`」是必然能够匹配成功的，有时它会匹配一个 `u`，其他时候则不匹配任何字符。关键在于，无论 `u` 是否出现，匹配都是成功的。但这并不等于，任何包含「`?`」的正则表达式都永远能匹配成功。例如，「`colo`」和「`u?`」都能在「`semicolon`」中匹配成功（前者匹配单词中的 `colo`，后者什么字符都没有匹配）。可是最后的「`r`」无法匹配，因此，最终「`colou?r`」无法匹配 `semicolon`。

来看另一个例子，我们需要匹配表示7月4日（`July fourth`）的文本，其中月份可能写作 `July` 或是 `Jul`，而日期可能写作 `fourth`、`4th` 或者是 `4`。显然，我们可以使用「`(July|Jul)-(fourth|4th|4)`」，但也可以找些其他的办法来解决这个问题。

首先，我们把「`(July|Jul)`」缩短为「`(July?)`」。你明白这种等价变换吗？删除「`|`」之后，就没必要保留括号了。当然保留也可以，但不保留括号显得更整洁一些。于是我们得到「`July?-(fourth|4th|4)`」。

现在来看第二部分，我们可以把「`4th|4`」简化为「`4(th)?`」。我们看到，现在「`?`」作用的元素是整个括号了。括号内的表达式可以任意复杂，但是“从括号外来看”它们是个整体。界定「`?`」的作用对象（还可以划定我即将介绍的其他类似元字符的作用对象）是括号的主要用途之一。

我们的表达式现在成了「`July?-(fourth|4(th)?)`」。尽管它包含了许多元字符，而且有嵌套的括号，但理解起来并不困难。我们花了相当的工夫来讲解这两个简单的例子，但同时也接触到了一些相关的知识，它们相当有助于——或许你现在还意识不到——我们理解正则表达式。同样，通过这些讲解，我们也积累了依靠不同思路解决问题的经验。在阅读本书（同时也是在加深理解）寻找复杂问题的最优解决方案的过程中，你可能会发现灵感可能在不断涌

现。正则表达式不是死板的教条，它更像是门艺术。

其他量词：重复出现

Other Quantifiers: Repetition

「+」（加号）和「*」（星号）的作用与问号类似。元字符「+」表示“之前紧邻的元素出现一次或多次”，而「*」表示“之前紧邻的元素出现任意多次，或者不出现”。换种说法就是，「...*」表示“匹配尽可能多的次数，如果实在无法匹配，也不要紧”。「...+」的意思与之类似，也是匹配尽可能多的次数，但如果连一次匹配都无法完成，就报告失败。问号、加号和星号这3个元字符，统称为量词（quantifiers），因为它们限定了所作用元素的匹配次数。

与「...?」一样，正则表达式中的「...*」也是永远不会匹配失败的，区别只在于它们的匹配结果。而「...+」在无法进行任何一次匹配时，会报告匹配失败。

举例来说，「-?」能够匹配一个可能出现的空格，但是「-*」能够匹配任意多个空格。我们可以用这些量词来简化第9页<H[1-6]>的例子。按照 HTML 规范（注7），在 tag 结尾的>字符之前，可以出现任意长度的空格，例如<H3->或者<H4——>。把「-*」加入正则表达式中的可能出现（但不是必须）空格的位置，就得到「H[1-6]-*」。它仍然能够匹配<H1>，因为空格并不是必须出现的，但其他形式的 tag 也能匹配。

接下来看类似<HR-SIZE=14>这样的 HTML tag，它表示一条高度为14像素的穿越屏幕的水平线。与<H3>的例子一样，在最后的尖括号之前可以出现任意多个空格。此外，在等号两边也容许出现任意多个空格。最后，在 HR 和 SIZE 之间必须有至少一个空格。为了处理更多的空格，我们可以在「-」后添加「-*」，不过最好还是改写为「-+」。加号确保至少有一个空格出现，所以它与「-*」是完全等价的，只不过更简洁。所以我们得到「<HR-+SIZE-*=-*14-*>」。

尽管这个表达式不受空格数目的限制，但它仍然受 tag 中直线尺寸大小的约束。我们要找的不仅仅是高度为14的 tag，而是所有这些 tag。所以，我们必须用能匹配普通数值（general number）的表达式来替换「14」。在这里，“数值”（number）是由一位或多位数字（digits）构成的。「[0-9]」可以匹配一个数字，因为“至少出现一次”，所以我们使用加号量词，结果就是用「[0-9]+」替换「14」。（一个字符组是一个“元素”（unit），所以它可以直接加加号、星号等，而不需要用括号。）

这样我们就得到了「<HR-+SIZE-*=-*[0-9]+-*>」，尽管我用了粗体标识元字符，用空格来分隔各个元素，而且使用了“看得见的空格符”“-”，这个表达式仍然不容易看懂（幸好，*egrep* 提供了-i 的参数 F15，这样我就不需要用「[Hh][Rr]」来表示「HR」了）。否则，「<HR+SIZE-*=-*[0-9]+-*>」更令人迷惑。这个表达式之所以看起来有些诡异，是因为星号和加号作用的对象大都是空格，而人眼习惯于把空格和普通字符区分开来。在阅读正则表达式时，我们必须改变这种习惯，因为空格符也是普通字符之一，它与 j 或者4这样的字符没有任何差别（在后面的章节中，我们会看到，某些工具软件支持忽略空格的特殊模式）。

我们继续这个例子，如果尺寸这个属性也是可选的，也就是说<HR>就代表默认高度的直线（同样，在 > 之前也可能出现空格）。你能修改我们的正则表达式，让它匹配这两种

类型的 **tag** 吗？解决问题的关键在于明白表示尺寸的文本是可选出现的（这是个暗示）。v
请翻到下一页查看答案。

请仔细观察最后（答案中）的表达式，体会问号、星号和加号之间的差异，以及它们在实际应用中的真正作用。下一页的表1-2总结了它们的意义。

请注意，每个量词都规定了匹配成功至少需要的次数下限，以及尝试匹配的次数上限。对某些量词来说，下限是0，对某些量词来说，上限是无穷大。

把一个子表达式变为可选项
v 19页问题的答案

在本例中，“可选出现”（optional）的意思是可以但并不必须匹配一次。这需要使
用「?」。因为可选项多于一个字符，所以我们必须使用括号：「(...)?’。把它插入表达式
中，就得到「<HR(-+SIZE-*=[0-9]+)?-*>」

请注意，结尾的「-*」在「(...)?’以外。这样就能应付「<HR->」之类的情况。如果我
们把它包含在括号内，则只有在出现“SIZE=...”这段文本的情况下才容许在>之前出现
空格。

同样请注意 SIZE 之前的「-+」包含在括号内。如果把它拿到括号之外，HR 之后就
必须紧跟至少一个空格，即使 SIZE 没有出现也是如此。这样‘<HR>’就无法匹配了。

表1-2：“表示重复的元字符”含义小结

	次 数 下限	次 数 上 限	含 义
?	无	1	可以出现，也可以只出现一次（单次可 选）
*	无	无	可以出现无数次，也可以不出现（任意 次数均可）
+	1	无	可以出现无数次，但至少要是出现一次（至 少一次）

规定重现次数的范围：区间

某些版本的 *egrep* 能够使用元字符序列来自定义重现次数的区间：「...{min, max}」。这称
为“区间量词（interval quantifier）”。例如，「...{3,12}」能够容许的重现次数在3到12之间。有
人可能会用「[a-zA-Z]{1,5}」来匹配美国的股票代码（1到5个字母）。问号对应的区间量词是
{0,1}。

支持区间表示法的 *egrep* 的版本并不多，但有许多另外的工具支持它。在第3章我们会
仔细考察目前经常使用的元字符，那时候会涉及区间的支持问题。

括号及反向引用

Parentheses and Backreferences

到目前为止，我们已经见过括号的两种用途：限制多选项的范围；将若干字符组合为一个单元，受问号或星号之类量词的作用。现在我要介绍括号的另一种用途，虽然它在 *egrep* 中并不常见（不过流行的 GNU 版本确实支持这一功能），但在其他工具软件中很常见。

在许多流派（*flavor*）的正则表达式中，括号能够“记住”它们包含的子表达式匹配的文本。在解决本章开始提到的单词重复问题时就会用到这个功能。如果我们确切知道重复单词的第一个单词（比方说这个单词就是“the”），就能够明确无误地找到它，例如「the-the」。这样或许还是会匹配到 the-theory 的情况，但如果我们的 *egrep* 支持在第15页提到的单词分界符「\<the-the\>」，这个问题就很容易解决。我们可以添加「+」把这个表达式变得更灵活。

然而，穷举所有可能出现的重复单词显然是不可能完成的任务。如果我们先匹配任意一个单词，接下来检查“后面的单词是否与它一样”，就好办多了。如果你的 *egrep* 支持“反向引用（*backreference*）”，就可以这么做。反向引用是正则表达式的特性之一，它容许我们匹配与表达式先前部分匹配的同样的文本。

我们先把「\<the-+the\>」中的第一个「the」替换为能够匹配任意单词的正则表达式「[A-Za-z]+」；然后在两端加上括号（原因见下段）；最后把后一个「the」替换为特殊的元字符序列「\1」，就得到了「\<([A-Za-z]+)-+\1\>」。

在支持反向引用的工具软件中，括号能够“记忆”其中的子表达式匹配的文本，不论这些文本是什么，元字符序列「\1」都能记住它们。

当然，在一个表达式中我们可以使用多个括号。再用「\1」、「\2」、「\3」等来表示第一、第二、第三组括号匹配的文本。括号是按照开括号‘(’从左至右的出现顺序进行的，所以「([a-z])([0-9])\1\2」中的「\1」代表「[a-z]」匹配的内容，而「\2」代表「[0-9]」匹配的内容。

在‘the-the’的例子中，「[A-Za-z]+」匹配第一个‘the’。因为这个子表达式在括号中，所以「\1」代表的文本就是‘the’。如果「+」能够匹配，后面的「\1」要匹配的文本就是‘the’。如果「\1」也能成功匹配，最后的「\>」对应单词的结尾（如果文本是‘the-theft’，这一条就不满足）。如果整个表达式能匹配成功，我们就得到一个重复单词。有的重复单词并不是错误，例如‘that that’（译注3），这并不是正则表达式的错误，真正的判断还得靠人。

我决定使用上面这个例子的时候，已经用这个表达式检查过本书之前的内容了（我使用的是支持「\<...\>」和反向引用的 *egrep*）。我还使用了第15页提到的忽略大小写的参数 *-i* 来拓宽它的适用范围（注8），所以‘The-the’这样的单词重复也能提取出来。

我使用的命令如下：

```
% egrep -i '\<([a-z]+)-+\1\>' files...
```

结果令我惊奇，居然找到了14组重复单词。我把它们全都改正了，而且把这个表达式添加到我用来检查本书拼写错误的工具中，保证从此以后全书中不会出现这样的错误。

尽管这个表达式很有用，我们仍然需要重视它的局限。因为 *egrep* 把每行文字都当作一个独立部分来看待，所以如果单词重复的第一个单词在某行末尾，第二个单词在下一行的开头，这个表达式就无法找到。所以，我们需要更加灵活的工具，下一章我们会看到这方面的

例子。

神奇的转义

The Great Escape

有个重要的问题我尚未提及，即：如果需要匹配的某个字符本身就是元字符，正则表达式会如何处理呢？例如，如果我想要检索互联网的主机名 `ega.att.com`，使用 `「ega.att.com」` 可能得到 `megawatt-computing` 的结果。还记得吗？`「.`」本身就是元字符，它可以匹配任何字符，包括空格。

真正匹配文本中点号的元序列应该是反斜线(`backslash`)加上点号的组合：`「ega\\.att\\.com」`。`「\\`」称为“转义的点号”或者“转义的句号”，这样的办法适用于所有的元字符，不过在字符组内部无效（注9）。

这样使用的反斜线称为“转义符（`escape`）”——被它作用的元字符失去特殊含义，成了普通字符。如果你愿意，也可以把转义符和它之后的元字符看作特殊的元字符序列，这个元字符序列匹配的是元字符对应的普通字符。这两种看法是等价的。

我们还可以用 `「\\([a-zA-Z]+)」` 来匹配一个括号内的单词，例如 `「(very)」`。在开闭括号之前的反斜线消除了开闭括号的特殊意义，于是他们能够匹配文本中的开闭括号。

如果反斜线后紧跟的不是元字符，反斜线的意义就依程序的版本而定。例如，我们已经知道，某些版本的程序把 `「<」`、`「>」`、`「|」` 当作元字符序列对待。在后面的章节中我们会看到更多的例子。

基础知识拓展

Expanding the Foundation

我希望，前面的例子和解释已经帮助读者牢固地打下了正则表达式的基础，也请读者明白，这些例子都很浅显，我们需要掌握的还有很多。

语言的差异

Linguistic Diversification

我已经介绍过大多数版本的 *egrep* 支持的正则表达式的特性，这样的特性还有很多，其中一些并不是所有的版本都支持，这个问题留到后面的章节讲解。

任何语言中都存在不同的方言和口音，很不幸，正则表达式也一样。情况似乎是，每一种支持正则表达式的语言都提供了自己的“改进”。正则表达式不断发展，但多年的变化也造就了数目众多的正则表达式“流派”（`flavor`）。我们会在下面的章节中见到各种例子。

正则表达式的目标

The Goal of a Regular Expression

从最宏观的角度看，一个正则表达式要么能够匹配给定文本（对 *egrep* 来说，就是一行文本）中的某些字符，要么不能匹配。在编写正则表达式的时候，我们必须进行权衡：匹配符合要求的文本，同时忽略不符合要求的文本。

尽管 *egrep* 不关心匹配文本在行中的位置，但对正则表达式的其他应用来说，这个问题却很重要。如果文本是这样：

```
...zip is 44272. If you write, send $4.95 to cover postage and...
```

我们只希望找出包含「[0-9]+」的那些行，就不需要关心真正匹配的数字。相反，如果我们需要操作这些数字（例如保存到文件、添加、替换之类——我们会在下一章看到这样的处理），就需要关心确切匹配的那些数字。

更多的例子

A Few More Examples

在任何语言中，经验都是非常重要的，所以我会给出更多用正则表达式匹配常用文本结构的例子。

编写正则表达式时，按照预期获得成功的匹配要花去一半的工夫，另一半的工夫用来考虑如何忽略那些不符合要求的文本。在实践中，这两方面都非常重要，但是目前我们只关注“获得成功匹配”的方面。即使我没有对这些例子进行最全面彻底的解释，它们仍然能够提供有用的启示。

变量名

许多程序设计语言都有标识符（*identifier*，例如变量名）的概念，标识符只包含字母、数字以及下画线，但不能以数字开头。我们可以用「[a-zA-Z_][a-zA-Z_0-9]*」来匹配标识符。第一个字符组匹配可能出现的第一个字符，第二个（包括对应的「*」）匹配余下的字符。如果标识符的长度有限制，例如最长只能是32个字符，又能使用第20页介绍的区间量词「{*min*, *max*}」，我们可以用「{0,31}」来替代最后的「*」。

引号内的字符串

匹配引号内的字符串最简单的办法是使用这个表达式：「"[^"]*"」。

两端的引号用来匹配字符串开头和结尾的引号。在这两个引号之间的文本可以包括双引号之外的任何字符。所以我们用「[^"]」来匹配除双引号之外的任何字符，用「*」来表示两个引号之间可以存在任意数目的非双引号字符。

关于引号字符串，更有用（也更复杂）的定义是，两端的双引号之间可以出现由反斜线转义的双引号，例如"nail-the-2\"x4"-plank"。在后面的章节讲解匹配实际进行的细节时，我们会多次遇到这个例子。

美元金额（可能包含小数）

「\\$[0-9]+(\.[0-9][0-9])?» 是一种匹配美元金额的办法。

从整体上看，这个表达式很简单，分为三部分：「\\$」、「...+」和「(...)?»，可以大致理解为一个美元符号，然后是一组字符，最后可能还有另一组字符。这里的“字符”指的是数字（一

组数字构成一个数值)，‘另一组字符’是由一个小数点和两位数字构成的。

从几个方面来看，这个表达式还很简陋。比如，它只能接受\$1000，而无法接受\$1,000。它确实能接受可能出现的小数部分，但对于 *egrep* 来说意义不大。因为 *egrep* 从不关心匹配文字的内容，而只关心是否存在匹配。处理可能出现的小数部分对整个表达式能否匹配并没有影响。

但是，如果我们需要找到只包含价格而不含其他字符的行，倒是可以在这个表达式两端加上「^...\$」。这样一来，可选的小数部分就变得很重要了，因为在金额数值和换行符之间是否存在小数部分，决定了整个表达式的匹配结果是否存在差异。

另外，这个正则表达式还无法匹配‘\$.49’。你可能认为把加号换成星号能够解决问题，不过这条路走不通。在这我先卖个关子，答案留待第5章（F194）揭晓。

HTTP/HTML URL

Web URL 的形式可能有很多种，所以构造一个能够匹配所有形式的 URL 的正则表达式颇有难度。不过，稍微降低一点要求的话，我们能够用一个相当简单的正则表达式来匹配大多数常见的 URL。进行这种检索的原因之一是，我只能大概记得在收到的某封邮件中有一个 URL 地址，不过一见到它我就能认出来。

常见的 HTTP/HTML URL 是下面这样的：

http://hostname/path.html

当然，.htm 的结尾也很常见。

hostname（主机名，例如 *www.yahoo.com*）的规则比较复杂，但是我们知道，跟在‘http://’之后的就有可能是主机名，所以这个正则表达式就很简单，「[-a-z0-9_]+」。 *path* 部分的变化更多，所以我们需要使用「[-a-z0-9_:@&?+=,./~*%\$]*」。请注意，连字符必须放在字符组的开头，保证它是一个普通字符，而不是用来表示范围（F9）。

综合起来，我们第一次尝试的正则表达式就是：

```
% egrep -i '\<http://[-a-z0-9_]+/[-a-z0-9_:@&?+=,./~*%$]*\.html?>' files
```

因为我们降低了对匹配的要求，所以‘<http://.../foo.html>’也能匹配，虽然它显然不是一个合法的 URL。我们需要关心这一点吗？这取决于具体的情况。如果我只是需要扫描自己的 E-mail，得到一些错误结果并不算是问题。而且，我没准会用更简单的表达式：

```
% egrep -i '\<http://[^\ ]*\.html?>' files...
```

在深入了解如何调校正则表达式之后，读者会明白，要想在复杂性和完整性之间求得平衡，一个重要的因素是了解待搜索的文本。下一章，我们会更详细地考察这个例子。

HTML tag

对 *egrep* 这样的工具来说，简单地匹配包含 HTML tag 的行并不常见，也没什么用。但是，探索如何准确匹配一个 HTML tag 却是相当有启发的，在下一章我们深入考察更高级的工具时，这一点尤其明显。

简单的例子包括‘<TITLE>’和‘<HR>’，我们可能会想到‘<.*>’。这个简单的表达式往往是最直接的想法，但它显然是不对的。‘<.*>’的意思是，“先匹配一个‘<’，然后是任意多个任意字符，然后是‘>’”。所以，它无疑能够匹配不止一个 tag 的内容，例如‘this <I>short</I> example’中标记的内容。

也许结果有点出乎你的意料，但是我们目前还只在第1章，对正则表达式的理解也不够深入。我之所以举这个例子，是想说明正则表达式并不复杂，但是如果你不真正弄懂它们，可能会被搞得晕头转向。在下面的几章中，我们会学习理解和解决这个问题需要的所有细节。

表示时刻的文字，例如“9:17 am”或者“12:30 pm”

匹配表示时刻的文字可能有不同的严格程度。

「[0-9]?[0-9]:[0-9][0-9]-(am|pm)」

能够匹配9:17-am 或者12:30-pm，但也能匹配无意义的时刻，如99:99-pm。

首先看小时数，我们知道，如果小时数是一个两位数，第一位只能是1。但是「1?[0-9]」仍然能够匹配19（也能够匹配0），所以更好的办法应该是把小时部分分为两种情况来处理，「1[012]」匹配两位数，「[1-9]」匹配一位数，结果就是「(1[012]|[1-9])」。

分钟数就简单些。第一位数字应该是「[0-5]」，此时第二位数字应该是「[0-9]」。综合起来就是「(1[012]|[1-9]):[0-5][0-9]-(am|pm)」。

举一反三，你能够处理24小时制的时间吗？多动动脑筋，想想该如何处理以0开头的情况，比如09:59呢？v 答案请见下页。

正则表达式术语汇总

Regular Expression Nomenclature

正则（regex）

你或许已经猜到了，“正则表达式”（regular expression）这个全名念起来有点麻烦，写出来就更麻烦。所以，我一般会采用“正则”（regex）的说法。这个单词念起来很流畅（有点像联邦快递的FedEx，与regular一样，g发重音，而不同于Regina），而且说“如果你写一个正则”，“巧妙的正则”（budding regexers），甚至是“正则化”（regexification）（注10）（译注4）。

匹配（matching）

一个正则表达式“匹配”一个字符串，其实是指这个正则表达式能在字符串中找到匹配文本。严格地说，正则表达式「a」不能匹配 cat，但是能匹配 cat 中的 a。几乎没人会混淆这两个概念，但提一提还是有必要的。

元字符（metacharacter）

一个字符是否元字符（或者是“元字符序列”（metasequence），这两个概念是相等的），取决于应用的具体情况。例如，只有在字符组外部并且是在未转义的情况下，「*」才是一个元字符。“转义”（escaped）的意思是，通常情况下在这个字符之前有一个反斜线。「*」是对「*」的转义，而「*」则不是（第一个反斜线用来转义第二个反斜线），虽然在两个例子中，星号

之前都有一个反斜线。

正则表达式的流派（**flavor**）不同，关于字符转义的规定也不相同。第3章对此进行了详细讨论。

流派（**flavor**）

我已经说过，不同的工具使用不同的正则表达式完成不同的任务，每样工具支持的元字符和其他特性各有不同。我们再举单词分界符的例子。某些版本的 *egrep* 支持我们曾见过的 `\<...\>` 表示法。而另一些版本不支持单独的起始和结束边界，只提供了统一的 `\b` 元字符（这个元字符我们还没见过，下一章才会用到）。还有些工具同时支持这两种表示法，另有许多工具哪种也不支持。

我用“流派（**flavor**）”这个词来描述所有这些细微的实现规定。这就好像不同的人说不同的方言一样。从表面上看，“流派”指的是关于元字符的规定，但它的内容远远不止这些。

即使两个程序都支持 `\<...\>`，它们可能对这两个元字符的意义有不同的理解，对单词的理解也不相同。在使用具体的工具软件时，这个问题尤其重要。

改进匹配时间的表达式，处理24小时制时间
v 26页问题的答案

办法有许多种，不过思路和之前差不多。现在我们把问题分为3部分：其一是上午（小时数从00到09，开头的0可选），其二是白天（小时数从10到19），其三是夜晚（小时数从20到23）。这样答案就很明显了：`0?[0-9]1[0-9]2[0-3]`。

实际上，我们可以合并头两个多选分支，得到 `[01]?[0-9]2[0-3]`。你可能需要动点脑筋才能明白这个表达式与上面是完全等价的。下面的图可能有所帮助，它还提供了另一种思路。阴影部分表示单个多选分支能够匹配的数字。

左图

右图

请不要混淆“流派（**flavor**）”和“工具（**tool**）”这两个概念。两个人可以说同样的方言，两个完全不同的程序也可能属于同样的流派。同样，两个名字相同的程序（解决的任务也相同）所属的流派可能有细微（有时可能并非细微）的差别。有许多程序都叫 *egrep*，它们所属的流派也五花八门。

由 Perl 语言的正则表达式开创的流派，在20世纪90年代中期因为其强大的表达能力广为人们所知，其他语言紧随其后，提供了汲取其中灵感的正则表达式（其中许多为了标明自己的思想来源，直接给自己贴上“兼容 Perl (Perl-Compatible)”的标签）。它们包括 PHP、Python、Java 的大量正则包，微软的 .NET Framework、Tcl，以及 C 的各种类库。不过，所有这些语言在重要的方面各有不同。而且 Perl 的正则表达式也在不断演化和发展（现在，有时候是受了其他语言的正则表达式的刺激）。像往常一样，总的局面变得越来越复杂，让人困惑。

子表达式（subexpression）

“子表达式”指的是整个正则表达式中的一部分，通常是括号内的表达式，或者是由「|」分隔的多选分支。例如，在「^(Subject|Date):-」中，「Subject|Date」通常被视为一个子表达式。其中的「Subject」和「Date」也算得上子表达式。而且，严格说起来，「S」、「u」、「b」、「j」这些字符，都算子表达式。

1-6这样的字符序列并不能算「H[1-6]*」的子表达式，因为「1-6」所属的字符组是不可分割的“单元（unit）”。但是，「H」、「[1-6]」、「*」都是「H[1-6]*」的子表达式。

与多选分支不同的是，量词（星号、加号和问号）作用的对象是它们之前紧邻的子表达式。所以「mis+pell」中的 + 作用的是「s」，而不是「mis」或者「is」。当然，如果量词之前紧邻的是一个括号包围的子表达式，整个子表达式（无论多复杂）都被视为一个单元。

字符（character）

“字符”在计算机领域是一个有特殊意义的单词。一个字节所代表的单词取决于计算机如何解释。单个字节的值不会变化，但这个值所代表的字符却是由解释所用的编码来决定的。例如，值为64和53的字节，在 ASCII 编码中分别代表了字符“@”和“5”，但在 EBCDIC 编码中，则是完全不同的字符（一个是空格，一个是控制字符）。

另一方面，在流行的日文字符编码中，这两个字节代表一个字符 正。如果换一种日文字符编码，这个字就需要两个完全不同的字节。那两个字节，在通行的 Latin-1 编码中，表示“ ”，而在 Unicode 编码中又表示韩文的“ ”（注11）。问题在于，字节如何解释只是视角（称为“编码”encoding）的问题，我们要做的只是确保自己的视角和正在使用的工具的视角相同。

一直以来，文本处理软件一般都把数据视为一些 ASCII 编码的字节，而不考虑使用者期望采用的字符编码。不过，近来已经有越来越多的系统在内部使用某些格式的 Unicode 编码来处理数据（第3章介绍了 Unicode，F105）。如果这些系统中的正则表达式子系统的实现方式正确，使用者通常就不需要在编码的问题上费太多工夫。这个“如果”相当复杂，所以第3章深入讲解了这个问题。

改进现状

Improving on the Status Quo

总的来说，正则表达式并不难。但是，如果你与使用过支持正则表达式的程序或语言的人交流过就会发现，某些人确实“会用”正则表达式，但如果需要解决复杂的问题，或是换用他们不熟悉的工具，就会出问题。

传统的正则表达式文档大都只包含一两个元字符的简略介绍，然后就给出关于其他元字符的表格。给出的例子通常也是无意义的「a*((ab)*|b*)」，文本则是「a-xxx-ce-xxxxxx-ci-xxx-d」。这些文档大都忽略了细微但重要的知识点，总是声称自己与其他出名的工具属于同一流派，而忘记提及必然存在的差异。它们缺乏实用价值。

当然，我的意思并不是，本章就能够填补这道鸿沟，让读者掌握所有正则表达式，或是掌握 *egrep* 的正则表达式。相反，这一章只是为本书的其他内容铺垫基础。我希望本书能够为读者填补这道鸿沟，虽然这期望有点自负。很多读者很满意本书的第一版，我本人也为拓展这一版的深度和广度付出了艰苦的努力。

或许是因为正则表达式的文档一直都非常欠缺，我感到自己必须做出额外的努力，才能把知识梳理清楚。因为我希望保证读者能够充分运用正则表达式的潜力，我希望你们能够真正精通正则表达式。

这既是件好事也是件坏事。

好处在于，你将学会如何以正则表达式的方式来思考问题。你将学习到，在面对属于不同流派的新工具时，需要注意哪些差异和特性。你还将会学习到，如果某个流派的功能弱小、特性简陋，该如何表达自己的意图。你将会明白，一个正则表达式的效率优于其他表达式的原因所在，而且你将能够在复杂性、效率和匹配准确性间进行取舍权衡。

面对特别复杂的任务，你将会知道如何通过程序容许的方式来构建和使用正则表达式。总的来说，你能够得心应手地使用正则表达式的所有潜能。

问题在于，这种方法的学习曲线非常陡峭，而且还有几大难点：

1 正则表达式的使用 许多程序使用的正则表达式比 *egrep* 要复杂。在我们探讨如何构造真正有用的正则表达式的细节之前，需要知道正则表达式的使用方法。下一章关注这一问题。

1 正则表达式的特性（**feature**） 在问题面前，选择合适的工具是成功的一半，所以我会在全书中使用多种工具。不同的程序，甚至是同一个程序的不同版本，支持的特性和元字符都不一样。在了解使用细节之前，我们必须搞清楚这个问题。这是第3章的主题。

1 正则表达式的工作原理 在我们接触有用（但通常也很复杂）的例子之前，我们必须“揭开盖子”来了解正则表达式的工作原理。我们将会看到，对某些元字符进行尝试匹配的次序是一个重要的问题。实际上，正则表达式引擎（**regular expression engine**）不同，工作原理也不同，所以对于同样的正则表达式，不同的程序会得到不同的结果。我们将在第4、5、6章中探讨这个复杂的问题。

正则表达式的工作原理是最重要同时也是最难以掌握的知识。研究这个问题有时确实很枯燥，更糟糕的是，读者在接触真正有趣的内容——解决实际问题——之前，不得不耐着性子看完它们。然而，弄懂正则表达式的工作原理，才是真正理解的关键。

你或许会想，如果只希望学会开车，是不需要了解汽车运行原理的。但是，学习开车与学习正则表达式之间并没有多少相似性。我的目的是教会读者如何使用正则表达式——也就是编写正则表达式——来解决问题。更合适的比喻是，学习正则表达式就如同学习如何造车，而不是如何开车。在制造汽车以前，我们必须了解汽车的工作原理。

第2章提供了更多的关于开车的经验。第3章简要回顾了开车的历史，详细考察了正则表

达式流派的主要内容。第4章介绍了正则表达式流派的重要的引擎。第5章展示了一些更复杂的例子，第6章告诉你如何调校某种具体的引擎，之后的各章则是检查具体的产品和模型。在第4、5、6章中，我们花了大量的篇幅来探讨幕后的原理，所以请务必做好准备。

总结

Summary

表1-3总结了我们在本章中见过的 *egrep* 的元字符。

表1-3: *egrep* 的元字符总结

匹配单个字符的元字符		
	元字符	匹配对象
.	点号	匹配单个任意字符
[...]	字符组	匹配单个列出的字符
[^...]	排除型字符组	匹配单个未列出的字符
<i>char</i>	转义字符	若 <i>char</i> 是元字符, 或转义序列无特殊含义时, 匹配 <i>char</i> 对应的普通字符
提供计数功能的元字符		
?	问号	容许匹配一次, 但非必须
*	星号	可以匹配任意多次, 也可能不匹配
+	加号	至少需要匹配一次, 至多可能任意多次
{ <i>min</i> , <i>max</i> }	区间量词 ^①	至少需要 <i>min</i> 次, 至多容许 <i>max</i> 次
匹配位置的元字符		
^	脱字符	匹配一行的开头位置
\$	美元符	匹配一行的结束位置
\<	单词分界符 ^①	匹配单词的开始位置
\>	单词分界符 ^①	匹配单词的结束位置
其他元字符		
	alternation	匹配任意分隔的表达式
(...)	括号	限定多选结构的范围, 标注量词作用的元素, 为反向引用“捕获”文本
1, 2, ...	反向引用 ^①	匹配之前的第一、第二组括号内的子表达式匹配的文本

①并非所有版本的 *egrep* 都支持

此外，请务必理解以下几点：

- 1 各个 *egrep* 程序是有差别的。它们支持的元字符，以及这些元字符的确切含义，通常都有差别——请参考相应的文档（F23）。
- 1 使用括号的3个理由是：限制多选结构（F13）、分组（F14）和捕获文本（F21）。

1 字符组的特殊性在于，关于元字符的规定是完全独立于正则表达式语言“主体”的。

1 多选结构和字符组是截然不同的，它们的功能完全不同，只是在有限的情况下，它们的表现相同（F13）。

1 排除型字符组同样是一种“肯定断言”（**positive assertion**）——即使它的名字里包含了“排除”两个字，它仍然需要匹配一个字符。只是因为列出的字符都会被排除，所以最终匹配的字符肯定不在列出的字符之内（F12）。

1 -i 的参数很有用，它能进行忽略大小写的匹配（F15）。

1 转义有3种情况：

1. `\` 加上元字符，表示匹配元字符所使用的普通字符（例如 `*` 匹配普通的星号）。

2. `\` 加上非元字符，组成一个由具体实现规定其意义的元字符序列（例如，`\<` 表示“单词的起始边界”）。

3. `\` 加上任意其他字符，默认情况就是匹配此字符（也就是说，反斜线被忽略了）。

请记住，对大多数版本的 *egrep* 来说，字符组内部的反斜线没有任何特殊意义，所以此时它并不是一个转义字符。

1 由星号和问号限定的对象在“匹配成功”时可能并没有匹配任何字符。即使什么字符都不能匹配到，它们仍然会报告“匹配成功”。

一家之言

Personal Glimpses

本章开始的单词重复的例子可能有些让人迷惑，不过正则表达式的功能的确很强大，我们只需要 *egrep* 这样简单的工具，用第1章的知识，就能够基本解决这个问题。我倒是希望在本章多讲些复杂点的例子，但是因为我希望用第1章为后面的章节打下坚实的基础，我担心，如果这一章满是提醒、注意、规则之类，那些从未接触过正则表达式的人在读完之后，或许会感到困惑——“需要这么麻烦吗？”

我哥哥曾教朋友玩一种叫 *schaffkopf* 的纸牌游戏，这个游戏在我们家流传了好几代了。其中真正的乐趣并不会在第一次接触时体会到，而且学习的曲线也很陡峭。我的嫂子丽兹平时是最有耐心的人，但玩了半个小时就对这些复杂的规则感到灰心丧气了，她说“我们不能玩兰米（译注5）吗？”不过最后的结果是，包括丽兹在内的所有人都玩到很晚。只要

度过了开始的难关，接下来的刺激就会引诱他们继续向前。我哥哥知道肯定会这样，但丽兹和其他玩伴需要花费时间和工夫来继续深入才能体会到这个游戏的乐趣。

习惯正则表达式可能需要花一些时间，所以在你没有真切体会到用正则表达式解决问题的成就感时，可能会觉得这样有点迂腐。如果是，我希望你能够抵抗住“玩兰米游戏”的诱惑。一旦你掌握了正则表达式的强大功能，就会感到花在学习上的那些时间真是物超所值。

前一章在开头类比了正则表达式与汽车，余下的部分介绍了正则表达式的功能、特点以

及其他相关信息。本章仍会使用这个类比来说明重要的正则引擎及其工作原理。

为什么需要了解这些原理呢？读者将会了解到，正则引擎分为很多种，最常用的引擎类型——Perl、Tcl、Python、.Net、Ruby、PHP，我见过的所有的 Java 正则包，以及其他语言使用的工作原理，基于此原理，构建正则表达式的方式决定了某个正则表达式能否匹配一个特定字符串，在何处匹配，以及匹配成功或报告失败的速度。如果你认为这些问题很重要，请阅读本章。

发动引擎

Start Your Engines!

现在我们来看看，引擎的类比能为我们提供多大帮助。引擎的价值在于，有了它，你不需要花多少气力就能从一个地方移动到另一个地方。驾驶员只需要放松或者听听音乐，发动机会完成余下的事情。它的主要任务就是驱动车轮，而驾驶员没必要关心它是如何工作的。是这样吗？

两类引擎

Two Kinds of Engines

设想一下驾驶电动汽车的情形？电动汽车已经诞生很久了，但他们不像汽油发动机驱动的汽车那样普遍，因为电动汽车还不够成熟。如果你有辆电动汽车，请记住别给它加油。如果你的汽车采用汽油发动机，请务必远离烟火。电动机几乎总是“能够运行”，汽油机则需要多加保养。更换火花塞、空气过滤器，或者换用不同品牌的汽油，有可能大大提升发动机的效率。当然，也可能降低汽油机的性能，或者导致发动机罢工。不同引擎的工作原理也有不同，但目的都是驱动车轮。不过，如果你想开车去某个地方，还得把好方向盘，当然，这是题外话。

新的标准

New Standards

让我们看看添加一条新规范：加利福尼亚州的尾气排放标准（注1）。一些发动机达到了加州的严格排放标准，一些则没有。这两类发动机并没有本质的不同，只是按标准划分为两类。这些标准规定了发动机尾气排放的成分，而并没有规定发动机应该怎样做才能达标。所以，现在我们可以把之前的两分法变为四分法：符合标准的电动机、不符合标准的电动机、符合标准的汽油机和不符合标准的汽油机。

回到原来的话题，我敢打赌，电动机不需要做多少改动就可以达标——标准只是“规定”尾气的成分，而电动机几乎没有尾气。相反，汽油机要达标可能就需要大的修改和更新。使用汽油发动机的驾驶员尤其需要注意汽油的型号——如果加错了油，他们就惹上大麻烦了。

标准的作用

更严格的排放标准是个好玩意儿，但驾驶员也需要考虑更多，同时更加小心（至少对汽

油车来说如此)。不过坦白说, 新标准对大多数人没有什么影响, 因为其他州不会施行加州的标准。

所以你知道, 四种类型的发动机其实可以分为三类: 两类是汽油机, 一类是电动机。虽然它们都是驱动车轮的, 但你明白了其中的差异。你不知道的是, 这堆复杂的玩意与正则表达式有什么关系! 其实这里面的关系远比你能想象的要复杂。

正则引擎的分类

Regex Engine Types

正则引擎主要可以分为基本不同的两大类: 一种是 DFA (相当于之前说的电动机), 另一种是 NFA (相当于前面的汽油机)。我们很快就会知道 DFA 和 NFA 的具体含义, 但是现在读者只需要知道这两个名字, 就像“Bill”和“Ted”, “汽油机”和“电动机”一样。

DFA 和 NFA 都有很长的历史, 不过, 正如汽油机一样, NFA 的历史更长一些。使用 NFA 的工具包括 .NET、PHP、Ruby、Perl、Python、GNU Emacs、*ea*、*sec*、*vi*、*grep* 的多数版本, 甚至还有某些版本的 *egrep* 和 *awk*。而采用 DFA 的工具主要有 *egrep*、*awk*、*lex* 和 *flex*。也有些系统采用了混合引擎, 它们会根据任务的不同选择合适的引擎 (甚至对同一表达式中的不同部分采用不同的引擎, 以求得功能与速度之间的最佳平衡)。表4-1列出了少量常用的工具及其大多数版本使用的引擎。如果你最喜欢的工具没有名列其中, 可以参考下一页的“测试引擎的类型”来找到答案。

表4-1: 部分程序及其所使用的正则引擎

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

第3章已经讲过, NFA 和 DFA 都发展了二十多年, 产生了许多不必要的变体, 导致现在的情况比较复杂。POSIX 标准的出台, 就是为了规范这种现象, POSIX 标准不但清楚地规定了前一章中提到的引擎应该支持的元字符和特性, 还明确规定了使用者期望由表达式获得的准确结果。除开表面细节不谈, DFA (也就是电动机) 显然已经符合新的标准, 但是 NFA 风格的结果却与此不一, 所以 NFA 需要修改才能符合标准。这样一来, 正则引擎可以粗略地分为3类:

- 1 DFA (符合或不符合 POSIX 标准的都属此类)。
- 1 传统型 NFA。

这里提到的 POSIX 是匹配意义上的，也就是说，POSIX 标准规定的某个正则表达式的应有行为（本章稍后部分将讨论）；而不是指 POSIX 标准引入的匹配特性。许多程序支持这些特性，但结果与 POSIX 规范不完全一致。

老式（功能极少的）程序，比如 *egrep*、*awk*、*lex* 之类，一般都是使用 DFA 引擎（电动机），所以，新的标准只是肯定了既有的情况，而没有大的改变。但是也存在一些汽油机版本的此类程序，如果它们需要达到 POSIX 标准，就需要做些修改。通过了加州排放标准测试（POSIX NFA）的汽油机能够产生符合标准的结果，但是这些必要的修改会增加保养的难度。以前，错位的火花塞也能应付着使用，但现在根本就点不着火。以前还能“凑合”的汽油，现在会弄得发动机砰砰乱响。不过，一旦掌握其中的门道，发动机就能平稳安静地运转了。

几句题外话

From the Department of Redundancy Department

现在，我请读者回过头去，重新思考关于引擎的故事。其中的每句话都涉及某些与正则表达式相关的事实。读第二遍会引起许多思考。尤其是，为什么说电动机（DFA 引擎）只是“能够运行”。什么影响了汽油机（NFA）？使用 NFA 引擎时，应该如何调整才能获得期望的结果？通过（排放）测试的 POSIX DFA 8 磅有什么特别之处？现实中“熄火的引擎”又是什么？

测试引擎的类型

Testing the Engine Type

工具所采用的引擎的类型，决定了引擎能够支持的特性以及这些特性的用途。所以，通常情况下，我们只需要几个测试用的表达式，就能判断出程序所使用的引擎类型（毕竟，如果你不能分辨引擎的类型，这种分类就没有意义）。现在，我并不期望读者理解下面的这些测试原理，我只是提供一些测试表达式，即使读者最喜欢使用的软件没有出现在表4-1之内，也可以判断出引擎的类型，继续阅读本章的其他内容。

是否传统型 NFA

传统型 NFA 是使用最广泛的引擎，而且它很容易识别。首先，看看忽略优先量词 (F141) 是否得到支持？如果是，基本就能确定这是传统型 NFA。我们将要看到，忽略优先量词是 DFA 不支持的，在 POSIX NFA 中也没有意义。为了确认这一点，只需要简单地用正则表达式 `[nfa|nfa-not]` 来匹配字符串 `'nfa-not'`，如果只有 `'nfa'` 匹配了，这就是传统型 NFA。如果整个 `'nfa not'` 都能匹配，则此引擎要么是 POSIX NFA，要么是 DFA。

DFA 还是 POSIX NFA

某些情况下，DFA 与 POSIX NFA 的区别是很明显的。DFA 不支持捕获型括号 (capturing parentheses) 和回溯 (backreferences)，这一点有助于判断，不过，也存在同时使用两种引擎的混合系统，在这种系统中，如果没有使用捕获型括号，就会使用 DFA。

下面这个简单的测试能说明很多问题，用「X(.+)+X」来匹配形如「=XX=====」的字符串，例如使用 *egrep* 命令：

```
echo =XX===== | egrep 'X(.+)+X'
```

如果执行需要花很长时间，就是 NFA（如果上一项测试显示这不是传统型 NFA，那么它肯定是 POSIX NFA）。如果时间很短，就是 DFA，或者是支持某些高级优化的 NFA。如果显示堆栈超溢（*stack overflow*），或者超时退出，那么它是 NFA 引擎。

匹配的基础

Match Basics

在了解不同引擎的差异之前，我们先看看它们的相似之处。汽车的各种动力系统在某些方面是一样的（或者说，从实用的角度考虑，它们是一样的），所以，下面的范例也能够适用于所有的引擎。

关于范例

About the Examples

本章关注的是一般的提供所有功能的正则引擎，所以，某些程序并不能完全支持它们。在本书所说的汽车里，机油油尺（*dipstick*）可能挨在机油滤清器（*oil filter*）的左边，而在读者那里，它却装在分电盘盖（*distributor cap*）的后面。不过，读者要做的只是理解这些概念，能够使用和维护自己最喜欢（以及他们最感兴趣）的正则表达式包。

在大部分例子中，我仍然使用 Perl 表示法，虽然我偶尔会用一些其他的表示法来提醒读者，表示法并不重要，我们讨论的问题与程序和表示法不属于一个层次。为节省篇幅，如果读者遇到不熟悉的构建方式，请查阅第3章（F114）。

本章详细阐释了匹配执行的实际流程。理想的情况是，所有的知识都能归纳为几条容易记忆的简单规则，使用者不需要了解这些规则包含的原理。很不幸，事实并非如此。整个第4章只能列出两条普适的原则：

1. 优先选择最左端（最靠开头）的匹配结果。
2. 标准的匹配量词（「*」、「+」、「?」和「{m,n}」）是匹配优先的。

在本章中，我们将考察这些规则，它们的结果，以及其他许多内容。首先我们详细讨论第一条规则。

规则1：优先选择最左端的匹配结果

Rule 1: The Match That Begins Earliest Wins

根据这条规则，起始位置最靠左的匹配结果总是优先于其他可能的匹配结果。这条规则并没有规定优先的匹配结果的长度（稍后将会讨论），而只是规定，在所有可能的匹配结果中，优先选择开始位置最左端的。实际上，因为可能有多个匹配结果的起始位置都在最左端，也许我们应该把这条规则中的“某个匹配结果（*a match*）”改为“该匹配结果（*the match*）”，不过这听起来有些别扭。

这条规则的由来是：匹配先从需要查找的字符串的起始位置尝试匹配。在这里，“尝试匹配（*attempt*）”的意思是，在当前位置测试整个正则表达式（可能很复杂）能匹配的每样

文本。如果在当前位置测试了所有的可能之后不能找到匹配结果，就需要从字符串的第二个字符之前的位置开始重新尝试。在找到匹配结果以前必须在所有的位置重复此过程。只有在尝试过所有的起始位置（直到字符串的最后一个字符）都不能找到匹配结果的情况下，才会报告“匹配失败”。

所以，如果要用「ORA」来匹配 FLORAL，从字符串左边开始第一轮尝试会失败（因为「ORA」不能匹配 FLO），第二轮尝试也会失败（「ORA」同样不能匹配 LOR），从第三个字符开始的尝试能够成功，所以引擎会停下来，报告匹配结果 FLORAL。

如果不了解这条规则，有时候就不能理解匹配的结果。例如，用「cat」来匹配：

The dragging belly indicates that your cat is too fat.

结果是 indicates，而不是后来出现的 cat。单词 cat 是能够被匹配的，但 indicates 中的 cat 出现的更早，所以得到匹配的是它。对于 *egrep* 之类的程序来说，这种差别是无关紧要的，因为它只关心“是否”能够匹配，而不是“在哪里”匹配。但如果是进行其他的应用，例如查找和替换，这种差别就很重要了。

这里有一个小测验（应该不困难）：如果用「fat|cat|belly|your」来匹配字符串‘The dragging belly indicates that your cat is too fat.’，结果是什么呢？v 请看下一页。

“传动装置（transmission）”和驱动过程（bump-along）

或许汽车变速箱（译注1）的例子有助于理解这条规则，驾驶员在换挡时，变速箱负责连接引擎和动力系统。引擎是真正产生动力的地方（它驱动曲轴），而变速箱把动力传送到车轮。

传动装置的主要功能：驱动

如果引擎不能在字符串开始的位置找到匹配的结果，传动装置就会推动引擎，从字符串的下一个位置开始尝试，然后是下一个，再下一个，如此继续。不过，如果某个正则表达式是以“字符串起始位置锚点（start-of-string anchor）”开头的，传动装置就会知道，不需要更多的尝试，因为如果能够匹配，结果肯定是从字符串的头部开始的。在第6章中，我们会讲解这一点，以及更多的内部优化措施。

引擎的构造

Engine Pieces and Parts

所有的引擎都是由不同的零部件组合而成的。如果对这些零件缺乏了解，也就不可能真正理解引擎的工作原理。正则引擎中的这些零件分为几类——文字字符（literal characters）、量词（qualifiers）、字符组（character classes）、括号，等等，我们在第3章介绍过（F114）。这些零件的组合方式（以及正则引擎对它们的处理方式）决定了引擎的特性，所以，这些零件的组合方式，以及它们之间的配合，是我们主要关注的东西。首先，让我们来看看这些零件：

文字文本（Literal Text）例如 a、*、!、枝...

对于非元字符的文字字符，尝试匹配时需要考虑就是“这个字符与当前尝试的字符相同吗？”。如果一个正则表达式只包含纯文本字符，例如「usa」，那么正则引擎会将其视为：一个「u」，接着一个「s」，接着一个「a」。进行不区分大小写的匹配时的情况要复杂

一点，因为「b」能够匹配 B，而「B」也能匹配 b，不过这仍然不难理解（Unicode 的情况稍微复杂一些 F110）。

字符组、点号、Unicode 属性及其他

通常情况下，字符组、点号、Unicode 属性及其他的匹配是比较简单的：无论字符组的长度是多少，它都只能匹配一个字符（注2）。

点号可以很方便地表示复杂的字符组，它几乎能匹配所有字符，所以它的作用也很简单，其他的简便方式还包括「\w」、「\W」和「\d」。

捕获型括号

用于捕获文本的括号（而不是用于分组的括号）不会影响匹配的过程。

测验答案

v 148页测验的答案

请记住，正则表达式的每一次尝试都要进行到底，所以「fat|cat|belly|your」用来匹配‘The dragging belly indicates your cat is too fat’的结果不是 fat，尽管「fat」在所有可能选项中列在最前头。

当然，正则表达式应该也能够匹配 fat 和其他可能，但它们都不是最先出现的匹配结果（除现在最左边的结果），所以不会被选择。在进行下一轮尝试之前，正则表达式的所有可能都会尝试，也就是说，在移动之前，「fat」、「cat」、「belly」和「your」都必须尝试。

锚点（e.g., 「^」 「\z」 「(?:=<\d)」...）

锚点可以分为两大类：简单锚点（^、\$、\G、\b、...F129）和复杂锚点（例如顺序环视和逆序环视 F133）。简单锚点之所以得名，就在于它们只是检查目标字符串中的特定位置的情况（^、\Z...），或者是比较两个相邻的字符（\<、\b、...）。相反，复杂锚点（环视）能包含任意复杂的子表达式，所以它们也可以任意复杂。

非“电动”的括号、反向引用和忽略优先量词

虽然本章希望讲解的是引擎之间的相似之处，但为了方便读者理解本章余下的内容，这里必须指出一些有意义的差异。捕获括号（以及相应的反向引用和\$1表示法）就像汽油添加剂一样——它们只对汽油机（NFA）起作用，对电动机（DFA）不起作用。忽略优先量词也是如此。这种情况是由 DFA 的工作原理决定的（注3）。这解释了，为什么 DFA 引擎不支持这些特性。读者会看到，awk、lex 和 egrep 都不支持反向引用和\$1功能（表示法）。

也许读者会注意到，GNU 版本的 egrep 确实支持反向引用。这是因为它包含了两台不同的引擎。它首先使用 DFA 查找可能的匹配结果，再用 NFA（支持包括反向引用在内的所有特性）来确认这些结果。接下来，我们将看到 DFA 不能支持反向引用及捕获括号的原因，以及这种引擎能够存在的理由（DFA 有很多显著的优势，例如匹配速度非常快）。

规则2：标准量词是匹配优先的

Rule 2: The Standard Quantifiers Are Greedy

至今为止，我们看到的特性都非常易懂。但仅仅靠它们还远远不够——要完成复杂点的任务，就需要使用星号、加号、多选结构之类功能更强大的元字符。要彻底理解这些功能，需要学习更多的知识。

读者首先需要记住的是，标准匹配量词（`?`、`*`、`+`，以及`{min, max}`）都是“匹配优先（greedy）”的。如果用这些量词来约束某个表达式，例如`「(expr)*」`中的`「(expr)」`、`「a？」`中的`「a」`和`「[0-9]+」`中的`「[0-9]」`，在匹配成功之前，进行尝试的次数是存在上限和下限的。在前面的章节中我们已经提到过这一点——而规则2表明，这些尝试总是希望获得最长的匹配（一些工具提供了其他的匹配量词，但是本节只讨论标准的匹配优先量词）。

简而言之，标准匹配量词的结果“可能”并非所有可能中最长的，但它们总是尝试匹配尽可能多的字符，直到匹配上限为止。如果最终结果并非该表达式的所有可能中最长的，原因肯定是匹配字符过多导致匹配失败。举个简单的例子：用`「b\w+s\b」`来匹配包含`「s」`的字符串，比如说`「regexes」`，`「\w+」`完全能够匹配整个单词，但如果用`「\w+」`来匹配整个单词，`「s」`就无法匹配了。为了完成匹配，`「\w+」`必须匹配`「regexes」`，把最后的`「s\b」`留出来。

如果表达式的其他部分能够成功匹配的唯一条件是，匹配优先的结构不匹配任何字符，在容许零匹配（译注2）的情况下（例如使用星号、问号，或者`{0, max}`），这是没有问题的。不过，这种情况只有在表达式的后续部分强迫下才能发生。匹配优先量词之所以得名，是因为它们总是（或者，至少是尝试）匹配多于匹配成功下限的字符。

匹配优先的性质可以非常有用（有时候也非常讨厌）。它可以用来解释`「[0-9]+」`为什么能匹配`March-1998`中的所有数字。1匹配之后，实际上已经满足了成功的下限，但此正则表达式是匹配优先的，所以它不会停在此处，而会继续下去，继续匹配`「998」`，直到这个字符串的末尾（因为`「[0-9]」`不能匹配字符串最后的空档，所以会停下来）。

邮件主题

显然，这种匹配方式并非只能用于匹配数字。举例来说，如果我们需要判断E-mail的header中的某行字符是否标题行（subject line）。前面（F55）我们已经说过，可以用`「^Subject:」`来实现。不过，如果使用`「^Subject:-(.*)」`，我们就在之后的程序中使用捕获型括号来访问主题的内容（例如Perl中的`$1`）（译注3）。

在探讨`「.*」`匹配邮件主题之前，请读者记住，一旦`「^Subject: -」`能够部分匹配，整个正则表达式就一定能够全部匹配。因为`「^Subject: -」`之后没有字符会导致表达式匹配失败：`「.*」`永远不会失败，因为“不匹配任何字符”也是`「.*」`的可能结果之一。

那么，为什么要添加`「.*」`呢？这是因为我们知道，星号是匹配优先的，它会用点号匹配尽可能多的字符，所以我们用它来“填充”`$1`。事实上，括号并没有影响正则表达式的匹配过程，在本例中，我们只是用它们来包括`「.*」`匹配的字符。

`「.*」`到达字符串的末尾之后点号不能继续匹配，所以星号最终停下来，尝试匹配表达式中的下一个元素（尽管`「.*」`无法继续匹配了，但下面的子表达式或许能够继续匹配）。不过，因为本例中不存在后面的元素，到达表达式的末尾之后，我们就获得了成功的匹配结果。

过度的匹配优先

现在让我们回过头去看“尽可能匹配”的匹配优先量词。如果我们在上面的例子中增加一个`「.*」`，把正则表达式写作`「^Subject:-(.*).*」`，结果会是如何呢？答案是，没有变化。开头的

「.*」（括号中的）会霸占整个标题的文本，而不给第二个「.*」留下任何字符。而第二个「.*」的匹配失败并不要紧，因为「.*」不匹配任何字符也能成功。如果我们给第二个「.*」也加上括号，\$2将会是空白。

这是否说明，在正则表达式中，「.*」的部分没有机会匹配任何字符呢？答案显然是否定的。就像我们在「\w+s」这个例子中看到的，如果进行全部匹配必须这样做，表达式中的某些部分可能“强迫”之前匹配优先的部分“释放”（或者说“交还（unmatch）”）某些字符。

「^.*([0-9][0-9])」或许是个有用的正则表达式，它能够匹配一行字符的最后两位数字，如果有的话，然后将它们存储在\$1中。下面是匹配的过程：「.*」首先匹配整行，而「[0-9][0-9]」是必须匹配的，在尝试匹配行末的时候会失败，这样它会通知「.*」：“嗨，你占的太多了，交出一些字符来吧，这样我没准能匹配。”匹配优先组件首先会匹配尽可能多的字符，但为了整个表达式的匹配，它们通常需要“释放”一些字符（抑制自己的天性）。当然，它们并不“愿意”这样做，只是不得已而为之。当然，“交还”绝不能破坏匹配成立必须的条件，比如加号的第一次匹配。

明白了这一点，我们来看「^.*([0-9][0-9])」匹配‘about-24-characters-long’的过程。「.*」匹配整个字符串以后，第一个「[0-9]」的匹配要求「.*」释放一个字符‘g’（最后的字符）。但是这并不能让「[0-9]」匹配，所以「.*」必须继续“交还”字符，接下来交还的字符是‘n’。如此循环15次，直到「.*」最终释放‘4’为止。

不幸的是，即使第一个「[0-9]」能够匹配‘4’，第二个「[0-9]」仍然不能匹配。为了匹配整个正则表达式，「.*」必须再次释放一个字符，这次是‘2’，由第一个「[0-9]」匹配。现在，‘4’能够由第二个「[0-9]」匹配，所以整个表达式匹配的是‘about-24-char...’，\$1的值是‘24’。

先来先服务

如果用「^.*[0-9]+」来匹配一行的最后两个数字，期望匹配的不止是最后两位数字，而是最后的整个数，结果会是多长呢？如果用它来匹配‘Copyright 2003.’，结果是什么？v 答案在下一页。

深入细节

在这里必须澄清一些东西。因为“「.*」必须交还...”或者“「[0-9]」迫使...”之类的说法或许容易引起混淆。我使用这些说法是因为它们易于理解，而且跟实际的结果一致。不过，事情的真相是由基本的引擎类型决定——是 DFA，还是 NFA。现在我们就来看这些。

表达式主导与文本主导

Regex-Directed Versus Text-Directed

DFA 和 NFA 反映了将正则表达式在应用算法上的根本差异。我把对应汽油机的 NFA 称为“表达式主导（regex-directed）”引擎，而对应电动机的 DFA 称为“文本主导（text-directed）”引擎。

NFA 引擎：表达式主导

NFA Engine: Regex-Directed

我们来看用「to(nite|knight|night)」匹配文本「...tonight...」的一种办法。正则表达式从「t」开始，每次检查一部分（由引擎查看表达式的一部分），同时检查“当前文本（current text）”是否匹配表达式的当前部分。如果是，则继续表达式的下一部分，如此继续，直到表达式的所有部分都能匹配，即整个表达式能够匹配成功。

测验答案

v 153页测验的答案

用「^.*([0-9]+)」匹配「copyright 2003.」，括号会捕获到什么？

这个表达式的本意是捕获整个数字2003，但结果并非如此。之前已经说过，为了满足「[0-9]+」的匹配，「.*」必须交还一些字符。在这个例子中，释放的字符是最后的‘3’和点号，之后‘3’能够由「[0-9]」匹配。「[0-9]」由「+」量词修饰，所以现在还只做到了最小的匹配可能，现在它遇到了‘.’，找不到其他可以匹配的字符。

与之前不同，此时没有“必须”匹配的元素，所以「.*」不会被迫交出0。否则，「[0-9]+」应当心存感激，接受匹配优先元素的馈赠，但请记住“先来先服务”原则。匹配优先的结构只会在被迫的情况下交还字符。所以，最终\$1的值是‘3’。

如果读者觉得难以理解，不妨这样想，「[0-9]+」和「[0-9]*」差不多，而本例中「[0-9]*」和「.*」是一样的。用它来替换原来的表达式「^.*([0-9]+)」，我们得到「^.*(.*)」，这与152页的「^Subject:-(.*)」很相似，第二个「.*」不会匹配任何字符。

在「to(nite|knight|night)」的例子中，第一个元素是「t」，它将会重复尝试，直到在目标字符串中找到‘t’为止。之后，就检查紧随其后的字符是否能由「o」匹配，如果能，就检查下面的元素。在本例中，“下面的元素”指「(nite|knight|night)」它的真正含义是“「nite」或者「knight」或者「night」”。引擎会依次尝试这3种可能。我们（具有高级神经网络的人类）能够发现，如果待匹配的字符串是 tonight，第三个选择能够匹配。不论神经学起源（F85）如何，表达式主导的引擎必须完全测试，才能得出结论。

尝试「nite」的过程与之前一样：“尝试匹配「n」，然后是「i」，然后是「t」，最后是「e」。”如果这种尝试失败——就像本例，引擎会尝试另一种可能，如此继续下去，直到匹配成功或是报告失败。表达式中的控制权在不同的元素之间转换，所以我称它为“表达式主导”。

NFA 引擎在操作上的优点

实质上，在表达式主导的匹配过程中，每一个子表达式都是独立的。这不同于反向引用，子表达式之间不存在内在联系，而只是整个正则表达式的各个部分。在子表达式与正则表达式的控制结构（多选分支、括号以及匹配量词）的层级关系（layout）控制了整个匹配过程。

因为 NFA 引擎是正则表达式主导的，驾驶员（也就是编写表达式的人）有充足的机会来实现他 / 她期望的结果（第5章和第6章将会告诉读者，如何正确高效地实现目标）。现在看起来，这点还有些模糊，但过一段就会变清晰。

DFA 引擎：文本主导

DFA Engine: Text-Directed

与表达式主导的 NFA 不同，DFA 引擎在扫描字符串时，会记录“当前有效（currently in the works）”的所有匹配可能。具体到 `tonight` 的例子，引擎移动到 `t` 时，它会在当前处理的匹配可能中添加一个潜在的可能：

字符串中的位置	正则表达式中的位置
after... <code>t</code> _æ onight	可能的匹配位置： <code>t</code> _æ <code>o(nite knight night)</code>
...	

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

字符串中的位置	正则表达式中的位置
after... <code>toni</code> _æ ght	可能的匹配位置： <code>to(ni</code> _æ <code>te knight ni</code> _æ <code>ght)</code>
...	

有效的可能匹配变为两个（`knight` 被淘汰出局）。扫描到 `g` 时，就只剩下一个可能匹配了。当 `h` 和 `t` 匹配完成后，引擎发现匹配已经完成，报告成功。

我称这种方式为“文本主导”，是因为它扫描的字符串中的每个字符都对引擎进行了控制。在本例中，某个未完成的匹配也许是任意多个（只要可行）匹配的开始。不合适的匹配可能在扫描后继文字时会被去除。在某些情况下，“处理中的未终结匹配（partial match in progress）”可能就是一个完整的匹配。例如正则表达式 `「to(...)？」`，括号内的部分并不是必须出现的，但考虑到匹配优先的性质，引擎仍然会尝试匹配括号内的部分。匹配过程中，在尝试括号内的部分时，完整匹配（`‘to’`）已经保留下来，以应付括号中的内容无法匹配的情况。

如果引擎发现，文本中出现的某个字符会令所有处理中的匹配可能失效，就会返回某个之前保留的完整匹配。如果不存在这样的完整匹配，则要报告在当前位置无法匹配。

第一想法：比较 NFA 与 DFA

First Thoughts: NFA and DFA in Comparison

如果读者根据上面介绍的知识比较 NFA 和 DFA，可能会得出结论：一般情况下，文本主导的 DFA 引擎要快一些。正则表达式主导的 NFA 引擎，因为需要对同样的文本尝试不同的子表达式匹配，可能会浪费时间（就好像上面例子中的3个分支）。

这个结论是对的。在 NFA 的匹配过程中，目标文本中的某个字符可能会被正则表达式中的不同部分重复检测（甚至有可能被同一部分反复检测）。即使某个子表达式能够匹配，为了检查表达式中剩下的部分，找到匹配，它也可能需要再一次应用（甚至可能反复多次）。单独的子表达式可能匹配成功，也可能失败，但是，直到抵达正则表达式的末尾之前，我们都无法确知全局匹配成功与否（也就是说“不到最后关头不能分胜负（It’s not over until the fat lady sings）”，但这句话又不符合本段的语境）。相反，DFA 引擎则是确定型的（deterministic）——目标文本中的每个字符只会检查（最多）一遍。对于一个已经匹配的字符，你无法知道

它是否属于最终匹配（它可能属于最终会失败的匹配），但因为引擎同时记录了所有可能的匹配，这个字符只需要检测一次，如此而已。

正则表达式引擎所使用的两种基本技术，都对应有正式的名字：非确定型有穷自动机（NFA）和确定型有穷自动机（DFA）。这两个名字实在是太饶舌，所以我坚持只用 DFA 和 NFA。下文中不会出现它们的全称了（注4）。

用户需要面对的结果

因为 NFA 具有表达式主导的特性，引擎的匹配原理就非常重要。我已经说过，通过改变表达式的编写方式，用户可以对表达式进行多方面的控制。拿 `tonight` 的例子来说，如果改变表达式的编写方式，可能会节省很多工夫，比如下面这3种方式：

```
1  「to(ni(ght|te)|knight)」
1  「tonite|toknight|tonight」
1  「to(k?night|nite)」
```

给出任意文本，这3个表达式都可以捕获相同的结果，但是它们以不同的方式控制引擎。现在，我们还无法分辨这3者的优劣，不过接下来会看到。

DFA 的情况相反——引擎会同时记录所有的匹配选择，因为这3个表达式最终能够捕获的文本相同，在写法上的差异并无意义。取得一个结果可能有上百种途径，但因为 DFA 能够同时记录它们（有点神奇，待稍后详述），选择哪一个表达式并无区别。对纯粹的 DFA 来说，即使「`abc`」和「`[aa-a](b|b{1}|b)c`」看来相差巨大，但其实是一样的。

如果要描述 DFA，我能想到的特征有：

- 1 DFA 匹配很迅速。
- 1 DFA 匹配很一致。
- 1 谈论 DFA 匹配很恼人。

最终我会展开这3点。

因为 NFA 是表达式主导的，谈论它是件很有意思的事情。NFA 为创造性思维提供了丰富的施展空间。调校好一个表达式能带来许多收益，调校不好则会带来严重后果。这就好比发动机的熄火和点不着火，他们并不只是汽油发动机的专利。为了彻底弄明白这个问题，我们来看回溯

Backtracking

NFA 引擎最重要的性质是，它会依次处理各个子表达式或组成元素，遇到需要在两个可能成功的可能中进行选择的时候，它会选择其一，同时记住另一个，以备稍后可能的需要。

需要做出选择的情形包括量词（决定是否尝试另一次匹配）和多选结构（决定选择哪个

多选分支，留下哪个稍后尝试）。

不论选择那一种途径，如果它能匹配成功，而且正则表达式的余下部分也成功了，匹配即告完成。如果正则表达式中余下的部分最终匹配失败，引擎会知道需要回溯到之前做出选择的地方，选择其他的备用分支继续尝试。这样，引擎最终会尝试表达式的所有可能途径（或者是匹配完成之前需要的所有途径）。

真实世界中的例子：面包屑

A Really Crummy Analogy

回溯就像是在道路的每个分岔口留下一小堆面包屑。如果走了死路，就可以照原路返回，直到遇见面包屑标示的尚未尝试过的道路。如果那条路也走不通，你可以继续返回，找到下一堆面包屑，如此重复，直到找到出路，或者走完所有没有尝试过的路。

在许多情况下，正则引擎必须在两个（或更多）选项中做出选择——我们之前看到的分支的情况就是一例。另一个例子是，在遇到「...x?...」时，引擎必须决定是否尝试匹配「x」。对于「...x+...」的情况，毫无疑问，「x」至少尝试匹配一次——因为加号要求必须匹配至少一次。第一个「x」匹配之后，此要求已经满足，需要决定是否尝试下一个「x」。如果决定进行，还要决定是否匹配第三个「x」，第四个「x」，如此继续。每次选择，其实就是洒下一堆“面包屑”，用于提示此处还有另一个可能的选择（目前还不能确定它能否匹配），保留起来以备后。

一个简单的例子

现在来看个完整的例子，用先前的「to(nite|knight|night)」匹配字符串‘hot-tonic- tonight!’（看起来有点无聊，但是个好例子）。第一个元素「t」从字符串的最左端开始尝试，因为无法匹配‘h’，所以在这个位置匹配失败。传动装置于是驱动引擎向后移动，从第二个位置开始匹配（同样也会失败），然后是第三个。这时候「t」能够匹配，接下来的「o」无法匹配，因为字符串中对应位置是一个空格。至此，本轮尝试宣告失败。

继续下去，从...**tonic**...开始的尝试则很有意思。to 匹配成功之后，剩下的3个多选分支都成为可能。引擎选取其中之一进行尝试，留下其他的备用（也就是洒下一些面包屑）。在讨论中，我们假定引擎首先选择的是「nite」。这个表达式被分解为“「n」+「i」+「t」+「e」”，在...**tonic**...遭遇失败。但此时的情况与之前不同，这种失败并不意味着整个表达式匹配失败——因为仍然存在没有尝试过的多选分支（就好像是，我们仍然可以找到先前留下的面包屑）。假设引擎然后选择「knight」，那么马上就会遭遇失败，因为「k」不能匹配‘n’。现在只剩下最后的选项「night」，但它不能失败。因为「night」是最后尝试的选项，它的失败也就意味着整个表达式在...**tonic**...的位置匹配失败，所以传动机构会驱动引擎继续前进。

直到引擎开始从...**tonight!**处开始匹配，情况又变得有趣了。这一次，多选分支「night」终于可以匹配字符串的结尾部分了（于是整体匹配成功，现在引擎可以报告匹配成功了）。

回溯的两个要点

Two Important Points on Backtracking

回溯机制的基本原理并不难理解，还是有些细节对实际应用很重要。它们是，面对众多选择时，哪个分支应当首先选择？回溯进行时，应该选取哪个保存的状态？第一个问题的答案是下面这条重要原则：

如果需要在“进行尝试”和“跳过尝试”之间选择，对于匹配优先量词，引擎会优先选择“进行尝试”，而对于忽略优先量词，会选择“跳过尝试”。

此原则影响深远。对于新手来说，它有助于解释为什么匹配优先的量词是“匹配优先”的，但还不完整。要想彻底弄清楚这一点，我们需要了解回溯时使用的是哪个（或者是哪些个）之前保存的分支，答案是：

距离当前最近储存的选项就是当本地失败强制回溯时返回的。使用的原则是 LIFO（last in first out，后进先出）。

用面包屑比喻就很好理解——如果前面是死路，你只需要沿原路返回，直到找到一堆面包屑为止。你会遇到的第一堆面包屑就是最近洒下的。传统的 LIFO 比喻也是这样：就像堆叠盘子一样，最后叠上去的盘子肯定是最先拿下来的。

备用状态

Saved States

用 NFA 正则表达式的术语来说，那些面包屑相当于“备用状态（saved state）”。它们用来标记：在需要的时候，匹配可以从这里重新开始尝试。它们保存了两个位置：正则表达式中的位置，和未尝试的分支在字符串中的位置。因为它是 NFA 匹配的基础，我们需要再看一遍某些已经出现过的简单但详细的例子，说明这些状态的意义。如果你觉得现有的内容都不难懂，请继续阅读。

未进行回溯的匹配

来看个简单的例子，用「ab?c」匹配 abc。「a」匹配之后，匹配的当前状态如下：

‘a _a bc’	「a _a b?c」
---------------------	----------------------

现在轮到「b?」了，正则引擎需要决定：是需要尝试「b」呢，还是跳过？因为?是匹配优先的，它会尝试匹配。但是，为了确保在这个尝试最终失败之后能够恢复，引擎会把：

‘a _a bc’	「ab? _a c」
---------------------	----------------------

添加到备用状态序列中。也就是说，稍后引擎可以从下面的位置继续匹配：从正则表达式中的「b?」之后，字符串的 b 之前（也就是当前的位置）匹配。这实际上就是跳过「b」的匹配，而问号容许这样做。

引擎放下面包屑之后，就会继续向前，检查「b」。在示例文本中，它能够匹配，所以新的当前状态变为：

‘ab _a c’	「ab? _a c」
---------------------	----------------------

最终的「c」也能成功匹配，所以整个匹配完成。备用状态不再需要了，所以不再保存它们。

进行了回溯的匹配

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

‘a _a c’	「ab? _a c」
--------------------	----------------------

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

不成功的匹配

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

‘a _a bX’	「ab? _a c」
---------------------	----------------------

「b」能够匹配，但这条路往下却走不通了，因为「c」无法匹配 X。于是引擎会回溯到之前的状态，“交还”b 给「c」来匹配。显然，这次测试也失败了。如果还有其他保存的状态，回溯会继续进行，但是此时不存在其他状态，在字符串中当前位置开始的整个匹配也就宣告失败。

事情到此结束了吗？没有。传动装置会继续“在字符串中前行，再次尝试正则表达式”，这可能被想象为一个伪回溯（pseudo-backtrack）。匹配重新开始于：

‘a _a bX’	「 _a ab?c」
---------------------	----------------------

从这里重新开始整个匹配，如同之前一样，所有的道路都走不通。接下来的两次（从 ab_aX 到 abX_a）都告失败，所以最终会报告匹配失败。

忽略优先的匹配

现在来看最开始的例子，使用忽略优先匹配量词，用「ab??c」来匹配‘abc’。「a」匹配之后的状态如下：

‘a _a bc’	「a _a b??c」
---------------------	-----------------------

接下来轮到「b??」，引擎需要进行选择：尝试匹配「b」，还是忽略？因为??是忽略优先的，

它会首先尝试忽略，但是，为了能够从失败的分支中恢复，引擎会保存下面的状态：

‘a _æ bc’	「a _æ bc」
---------------------	---------------------

到备用状态列表中。于是，引擎稍后能够用正则表达式中的「b」来尝试匹配文本中的 b（我们知道这能够匹配，但是正则引擎不知道，它甚至都不知道是否会要用到这个备用状态）。状态保存之后，它会继续向前，沿着忽略匹配的路走下去：

‘a _æ bc’	「ab?? _æ c」
---------------------	-----------------------

「c」无法匹配‘b’，所以引擎必须回溯到之前保存的状态：

‘a _æ bc’	「a _æ bc」
---------------------	---------------------

显然，此时匹配可以成功，接下来的「c」匹配‘c’。于是我们得到了与使用匹配优先的「ab?c」同样的结果，虽然两者所走的路不相同。

回溯与匹配优先

Backtracking and Greediness

如果工具软件使用的是 NFA 正则表达式主导的回溯引擎，理解正则表达式的回溯原理就成了高效完成任务的关键。我们已经看到「？」的匹配优先和「??」的忽略优先是如何工作的，现在来看星号和加号。

星号、加号及其回溯

如果认为「x*」基本等同于「x?x?x?x?x?...」（或者更确切地说是「(x(x(x(x...?)?)?)?)」）（注5），那么情况与之前没有大的差别。每次测试星号作用的元素之前，引擎都会保存一个状态，这样，如果测试失败（或者测试进行下去遭遇失败），还能够从保存的状态开始匹配。这个过程会不断重复，直到包含星号的尝试完全失败为止。

所以，如果用「[0-9]+」来匹配‘a-1234-num’，「[0-9]」遇到4之后的空格无法匹配，而此时加号能够回溯的位置对应了四个保存的状态：

a 1_æ234 num

a 12_æ34 num

a 123_æ4 num

a 1234_æ num

也就是说，在每个位置，「[0-9]」的尝试都代表一种可能。在「[0-9]」遇到空格匹配失败时，引擎回溯到最近保存的状态（也就是最下面的位置），选择正则表达式中的「[0-9]+_æ」和文本中的‘a-1234_æ-num’。当然，到此整个正则表达式已经结束，所以我们知道，整个匹配宣告完成。

请注意，‘a-1234-num’并不在列表中，因为加号限定的元素至少要匹配一次，这是必要条件。那么，如果正则表达式是‘[0-9]*’，这个状态会保存吗？（提示：这个问题得动点脑筋）。要知道答案，v 请翻到下一页。

重新审视更完整的例子

有了更详细的了解之后，我们再来看看第152页的‘^.*([0-9][0-9])’的例子。这一次，我们不是只用“匹配优先”来解释为什么会得到那样的匹配结果，我们能够根据 NFA 的匹配机制做出精确解释。

以‘CA-95472-USA’为例。在‘.*’成功匹配到字符串的末尾时，星号约束的点号匹配了13

个字符，同时保存了许多备用状态。这些状态表明稍后的匹配开始的位置：在正则表达式中是‘^.*_([0-9][0-9])’，在字符串中则是点号每次匹配时保存的备用状态。

现在我们已经到了字符串的末尾，并把控制权交给第一个‘[0-9]’，显然这里的匹配不能成功。没问题，我们可以选择一个保存的状态来进行尝试（实际上保存了许多的状态）。现在回溯开始，把当前状态设置为最近保存的状态，也就是‘.*’匹配最后的 A 之前的状态。忽略（或者，如果你愿意，可以使用“交还”）这个匹配，于是有机会用‘[0-9]’匹配这个 A，但这同样会失败。

这种“回溯-尝试”的过程会不断循环，直到引擎交还2为止，在这里，第一个‘[0-9]’可以匹配。但是第二个‘[0-9]’仍然无法匹配，所以必须继续回溯。现在，之前尝试中第一个‘[0-9]’是否匹配与本次尝试并无关系了，回溯机制会把当前的状态中正则表达式内的对应位置设置到第一个‘[0-9]’以前。我们看到，当前的回溯同样会把字符串中的位置设置到7以前，所以第一个‘[0-9]’可以匹配，而第二个‘[0-9]’也可以（匹配2）。所以，我们得到一个匹配结果‘CA-95472,-USA’，\$1得到72。

需要注意的是：第一，回溯机制不但需要重新计算正则表达式和文本的对应位置，也需要维护括号内的子表达式所匹配文本的状态。在匹配过程中，每次回溯都把当前状态中正则表达式的对应位置指向括号之前，也就是‘^.*_([0-9][0-9])’。在这个简单的例子中，所以它等价于‘^.*_ [0-9][0-9]’，因此我说“使用第一个[0-9]之前的位置”。然而，回溯对括号的这种处理，不但需要同时维护\$1的状态，也会影响匹配的效率。

最后需要注意的一点也许读者早就了解：由星号（或其他任何匹配优先量词）限定的部分不受后面元素影响，而只是匹配尽可能多的内容。在我们的例子中，‘.*’在点号匹配失败之前，完全不知道，到底应该在哪个数字或者是其他什么地方停下来。在‘^.*([0-9]+)’的例子中我们看到，‘[0-9]+’只能匹配一位数字（F153）。

关于匹配优先和回溯的更多内容

More About Greediness and Backtracking

NFA 和 DFA 都具备许多匹配优先相关的特性（也从中获益）。（DFA 不支持忽略优先，所以我们现在只关注匹配优先）。我将考察两种引擎共同支持的匹配优先特性，但只会用 NFA 来讲解。这些内容对 DFA 也适用，但原因与 NFA 不同。DFA 是匹配优先的，使用起来很

测试答案

v 162页测试的答案

如果用「`[0-9]*`」匹配「`a-1234-num`」，备用状态是否包括「`a-1234-num`」？

答案是否定的。之所以提这个问题，是因为这种错误很常见。记得吗，由星号限定的部分总是能够匹配。如果整个表达式都由星号控制，它能够匹配任何内容。在字符串的开始位置，传动机构对引擎进行第一次尝试时的状态，当然算匹配成功。在这种情况下，正则表达式匹配「`a-1234-num`」，而且在此处结尾——它根本没有触及到那些数字。

如果没答对也不要紧，因为这种情况还是有可能发生的。如果在表达式中，「`[0-9]*`」之后还出现了某些元素，因为它们的存在，引擎在达到下面状态之前无法获得全局匹配：

「 <code>a-1234...</code> 」	「 <code>a[0-9]*...</code> 」
那么，尝试「1」会生成下面的状态：	
「 <code>a-1234...</code> 」	「 <code>[0-9]*1234...</code> 」

方便，这一点我们只需要记住就够了，而且介绍起来也很乏味。相反，NFA 的匹配优先就很值得一谈，因为 NFA 是表达式主导的，所以匹配优先的特性能产生许多神奇的结果。除了忽略优先，NFA 还提供了其他许多特性，比如环视、条件判断（`conditions`）、反向引用，以及固化分组。最重要的是 NFA 能让使用者直接操控匹配的过程，如果运用得当，这会带来很大的方便，但也可能带来某些性能问题（具体见第6章）。

不考虑这些差异的话，匹配的结果通常是相通的。我介绍的内容适用于两种引擎，除了使用表达式主导的 NFA 引擎。读完本章，读者会明白，在什么情况下两种引擎的结果会不一样，以及为什么会不一致。

匹配优先的问题

Problems of Greediness

在上例中已经看到，「`.*`」经常会匹配到一行文本的末尾（注6）。这是因为「`.*`」匹配时只从自身出发，匹配尽可能多的内容，只有在全局匹配需要的情况下才会“被迫”交还一些字符。

有些时候问题很严重。来看用一个匹配双引号文本的例子。读者首先想到的可能是「`".*"`」，那么，请运用我们刚刚学到的关于「`.*`」的知识，猜一猜用它匹配下面文本的结果：

The name "McDonald's" is said "makudonarudo" in Japanese.

既然知道了匹配原理，其实我们并不需要猜测，因为我们“知道”结果。在最开始的双引号匹配之后，「`.*`」能够匹配任何字符，所以它会一直匹配到字符串的末尾。为了让最后的双引号能够匹配，「`.*`」会不断交还字符（或者，更确切地说，是正则引擎强迫它回退），直到满足为止。最后，这个正则表达式匹配的结果就是：

The name "McDonald's" is said "makudonarudo" in Japanese.

这显然不是我们期望的结果。我之所以提醒读者不要过分依赖「.*」，这就是原因之一，如果不注意匹配优先的特性，结果往往出乎意料。

那么，我们如何能够只取得"McDonald's"呢？关键的问题在于要认识到，我们希望匹配的不是双引号之间的“任何文本”，而是“除双引号以外的任何文本”。如果用「[^\"]*」取代「.*」，就不会出现上面的问题。

使用「[^\"]*」时，正则引擎的基本匹配过程跟上面是一样的。第一个双引号匹配之后，「[^\"]*」会匹配尽可能多的字符。在本例中，就是 McDonald's，因为「[^\"]」无法匹配之后的双引号。此时，控制权转移到正则表达式末尾的「」」。而它刚好能够匹配，所以获得全局匹配：

```
The name "McDonald's" is said "makudonarudo" in Japanese.
```

事实上，可能还存在一种出乎读者预料的情况，因为在大多数流派中，「[^\n]」能够匹配换行符，而点号则不能。如果不想让表达式匹配换行符，可以使用「[^\n]」。

多字符“引文”

Multi-Character“Quotes”

第1章出现了对 HTML tag 的匹配，例如，浏览器会把very中的“very”渲染为粗体。对...的匹配尝试看起来与对双引号匹配的情形很相似，只是“双引号”在这里成了多字符构成的和。与双引号字符串的例子一样，使用「.*」匹配多字符“引文”也会出错：

```
...<B>Billions</B> and <B>Zillions</B> of suns...
```

「.*」中匹配优先的「.*」会一直匹配该行结尾的字符，回溯只会进行到「」能够匹配为止，也就是最后一个，而不是与匹配开头的「」对应的「」。

不幸的是，因为结束 tag 不是单个字符，所以不能用之前的办法，也就是“排除型字符组”来解决，即我们不能期望用「[^]*」解决问题。字符组只能代表单个字符，而我们需要的「」是一组字符。请不要被「[]」迷惑，它只是表示一个不等于<、>、\、B 的字符，等价于「[^\<>\B]」，而这显然不是我们想要的“除结构之外的任何内容”（当然，如果使用环视功能，我们可以规定，在某个位置不应该匹配，下一节会出现这种应用）。

使用忽略优先量词

Using Lazy Quantifiers

上面的问题之所以会出现，原因在于标准量词是匹配优先的。某些 NFA 支持忽略优先的量词（F141），*? 就是与 * 对应的忽略优先量词。我们用「.*?」来匹配：

```
...<B>Billions</B> and <B>Zillions</B> of suns...
```

开始的「」匹配之后，「.*?」首先决定不需要匹配任何字符，因为它是忽略优先的。于是，控制权交给后面的「<」符号：

‘... _g Billions...’	「.*? _g 」
-----------------------------------	----------------------------

此时「<」无法匹配，所以控制权交还给「.*?」，因为还有未尝试过的匹配可能（事实上能够进行多次匹配尝试）。它的匹配尝试是步步为营的（*begrudgingly*），先用点号来匹配...Billions...中带下画线的 B。此时，.*?又必须选择，是继续尝试匹配，还是忽略？因为它是忽略优先的，会首先选择忽略。接下来的「<」仍然无法匹配，所以「.*?」必须继续尝试未匹配的分支。在这个过程重复8次之后，「.*?」最终匹配了 Billions，此时，解下来的「<」（以及整个「」）都能匹配：

...Billions and Zillions of suns...

所以我们知道了，星号之类的匹配优先量词有时候的确很方便，但有时候也会带来大麻烦。这时候，忽略优先的量词就能派上用场了，因为它们做到其他办法很难做到（甚至无法做到）的事情。当然，缺乏经验的程序员也可能错误地使用忽略优先量词。事实上，我们刚刚做的或许就不正确。如果用「.*?」来匹配：

...Billions and Zillions of suns...

结果如上图，虽然我假设匹配的结果取决于具体的需求，但我仍然认为，这种情况下的结果不是用户期望的。不过，「.*?」必然会匹配 Zillions 左边的，一直到。

这个例子很好地说明了，为什么通常情况下，忽略优先量词并不是排除类的完美替身。在「".*"'」的例子中，使用「[^\"]」替换点号能避免跨越引号的匹配——这正是我们希望实现的功能。

如果支持排除环视（F133），我们就能得到与排除型字符组相当的结果。比如，「(?:)」这个测试，只有当不在字符串中的当前位置时才能成功。这也是「.*?」中的点号期望匹配的内容，所以把点号改为「(?:.)」得到的正则表达式，就能准确匹配我们期望的内容。把这些综合起来，结果有点儿难看懂，所以我选用带注释的宽松格式模式（F111）来讲解：

```
<B>                # 匹配开头的<B>
(
    # 然后只匹配尽可能少的内容
    (?! <B> )      # 如果不是<B>...
    .              # ... 任何字符都可以
)*?               #
</B>              # ... 直到遇到结束标记
```

使用了环视功能之后，我们可以重新使用普通的匹配优先量词，这样看起来更清楚：

```
<B>                # 匹配开头的<B>
```



```
(
    # 然后匹配尽可能多的内容

(?! </? B>)    # 如果不是<B>，也不是</B> ...

.              # ... 任何字符都可以

)*            # （现在是匹配优先的量词）

</B>          # <ANNO> ... 直到结束分隔符匹配
```

现在，环视功能会禁止正则表达式的主体（**main body**）匹配和之外的内容，这也是我们之前试图用忽略优先量词解决的问题，所以现在可以不用忽略优先功能了。这个表达式还有能够改进的地方，我们将在第6章关于效率的讨论中看到它（F270）。

匹配优先和忽略优先都期望获得匹配

Greediness and Laziness Always Favor a Match

回忆一下第2章中显示价格的例子（F51）。我们会在本章的多个地方仔细检查这个例子，所以我先重申一下基本的问题：因为浮点数的显示问题，“1.625”或者“3.00”有时候会变成“1.62500000002828”和“3.00000000028822”。为解决这个问题，我使用：

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

来保存\$price 小数点后头两到三位十进制数字。「\\.\\d\\d」匹配最开始两位数字，而「[1-9]?」用来匹配可能出现的不等于零的第三个数字。

然后我写道：

到现在，我们能够匹配的任何内容都是希望保留的，所以用括号包围起来，作为\$1捕获。然后就能在 **replacement** 字符串中使用\$1。如果这个正则表达式能够匹配的文本就等于\$1，那么我们就是用用一个文本取代它本身——这没有多少意义。然而，我们继续匹配\$1括号之外的部分。在替代字符串中它们并不出现，所以结果就是它们被删掉了。在本例中，“要删掉”的文本就是任何多余的数字，也就是正则表达式中末尾的「\\d*」匹配的部分。

到现在看起来一切正常，但是，如果\$price 的数据本身规范格式，会出现什么问题呢？如果\$price 是27.625，那么「\\.\\d\\d[1-9]?」能够匹配整个小数部分。因为「\\d*」无法匹配任何字符，整个替换就是用‘.625’替换‘.625’——相当于白费工夫。

结果当然是我们需要的，但是否存在更有效率的办法，只进行真正必要的替换（也就是，只有当「\\d*」确实能够匹配到某些字符的时候才替换）呢？当然，我们知道怎样表示“至少一位数字”！只要把「\\d*」替换为「\\d+」就好了：

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

对于“1.62500000002828”这样复杂的数字，正则表达式仍然有效；但是对于“9.43”这种数字，末尾的「\\d+」不能匹配，所以不会替换。所以，这是个了不起的改动，对吗？不！如果要处理的数字是27.625，情况如何呢？我们希望这个数字能够被忽略，但是这不会发生。

请仔细想想27.625的匹配过程，尤其是表达式如何处理‘5’这个数字。

知道答案之后，这个问题就变得非常简单了。在「(\d\d[1-9]?)\d+」匹配27.625之后，「\d+」无法匹配。但这并不会影响整个表达式的匹配，因为就正则表达式而言，由「[1-9]」匹配‘5’只是可选的分支之一，还有一个备用状态尚未尝试。它容许「[1-9]?’匹配一个空字符，而把5留给至少必须匹配一个字符的「\d+」。于是，我们得到的是错误的结果：.625被替换成了.62。

如果「[1-9]?’是忽略优先的又如何呢？我们会得到同样的匹配结果，但不会有“先匹配5再回溯”的过程，因为忽略优先的「[1-9]?’首先会忽略尝试匹配的选项。所以，忽略优先量词并不能解决这个问题。

匹配优先、忽略优先和回溯的要旨

The Essence of Greediness, Laziness, and Backtracking

之前的章节告诉我们，正则表达式中的某个元素，无论是匹配优先，还是忽略优先，都是为全局匹配服务的，在这一点上（对前面的例子来说）它们没有区别。如果全局匹配需要，无论是匹配优先（或是忽略优先），在遇到“本地匹配失败”时，引擎都会回归到备用状态

（按照足迹返回找到面包屑），然后尝试尚未尝试的路径。无论匹配优先还是忽略优先，只要引擎报告匹配失败，它就必然尝试了所有的可能。

测试路径的先后顺序，在匹配优先和忽略优先的情况下是不同的（这就是两者存在的理由），但是，只有在测试完所有可能的路径之后，才会最终报告匹配失败。

相反，如果只有一条可能的路径，那么使用匹配优先量词和忽略优先量词的正则表达式都能找到这个结果，只是他们尝试路径的次序不同。在这种情况下，选择匹配优先还是忽略优先，并不会影响匹配的结果，受影响的只是引擎在达到最终匹配之前需要尝试的次数（这是关于效率的问题，第6章将会谈到）。

最后，如果存在不止一个可能的匹配结果，那么理解匹配优先、忽略优先和回溯的读者，就明白应该如何选择。「".*"'有3个可能的匹配结果：

The name "McDonald's" s said "makudonarudo" in Japanese.

我们知道，使用匹配优先星号的「".*"'匹配最长的结果，而使用忽略优先星号的「".*?'"匹配最短的结果。

占有优先量词和固化分组

Possessive Quantifiers and Atomic Grouping

上一页中‘.625’的例子展示了关于 NFA 匹配的重要知识，也让我们认识到，针对这个具体的例子，考虑不仔细就会带来问题。针对某些流派提供了工具来帮助我们，但是在接触它们以前，必须彻底弄明白“匹配优先、忽略优先和回溯的要旨”这一节。如果读者还有不明白的地方，请务必仔细阅读上一节。

那么，仍然来考虑‘.625’的例子，想想我们真正的目的。我们知道，如果匹配能够进行到「(\.d[1-9]?)\d+」中标记的位置，我们就不希望进行回溯。也就是说，我们希望的是，如果可能，「[1-9]」能够一个字符，果真如此的话，我们不希望交还这个字符。或者说的更直白一些就是，如果需要的话，我们希望在「[1-9]」匹配的字符交还之前，整个表达式就匹配失败。（这个正则表达式匹配‘.625’时的问题在于，它不会匹配失败，而是进行回溯，尝试其他备用状态）。

那么，如果我们能够避免这些备用状态呢（也就是在[1-9]进行尝试之前，放弃「?」保存的状态，）？如果没有退路，「[1-9]」的匹配就不会交还。而这就是我们需要的！但是，如果没有了这个备用状态会发生什么？如果我们用这个表达式来匹配‘.5000’呢？此时「[1-9]」不能匹配，就确实需要回溯，放弃「[1-9]」，让之后的「\d+」能够匹配需要删除的数字。

听起来，我们有两种相反的要求，但是仔细考虑就会发现，我们真正需要的是，只有在某个可选元素已经匹配成功的情况下才抛弃此元素的“忽略”状态。也就是说，如果「[1-9]」能够成功匹配，就必须抛弃对应的备用状态，这样就永远也不会回退。如果正则表达式的流派支持固化分组「(>...）」（F139），或者占有优先量词「[1-9]?+」（F142），就可以这么做。首先来看固化分组。

用(>...)实现固化分组

具体来说，使用「(>...）」的匹配与正常的匹配并无差别，但是如果匹配进行到此结构之后（也就是，进行到闭括号之后），那么此结构中的所有备用状态都会被放弃。也就是说，在固化分组匹配结束时，它已经匹配的文本已经固化为一个单元，只能作为整体而保留或放弃。括号内的子表达式中未尝试过的备用状态都不复存在了，所以回溯永远也不能选择其中的状态（至少是，当此结构匹配完成时，“锁定（locked in）”在其中的状态）。

所以，让我们来看「(\.d(>[1-9]?))\d+」。在固化分组内，量词能够正常工作，所以如果「[1-9]」不能匹配，正则表达式会返回「?」留下的备用状态。然后匹配脱离固化分组，继续前进到「\d+」。在这种情况下，当控制权离开固化分组时，没有备用状态需要放弃（因为在固化分组中没有创建任何备用状态）。

如果「[1-9]」能够匹配，匹配脱离固化分组之后，「?」保存的备用状态仍然存在。但是，因为它属于已经结束的固化分组，所以会被抛弃。匹配‘.625’或者‘.625000’时就会发生这种情况。在后一种情况下，放弃那些状态不会带来任何麻烦，因为「\d+」匹配的是‘.625000’，到这里正则表达式已经完成匹配。但是对于‘.625’来说，因为「\d+」无法匹配，正则引擎需要回溯，但回溯又无法进行，因为备用状态已经不存在了。既然没有能够回溯的备用状态，整体匹配也就失败，‘.625’不需要处理，而这正是我们期望的。

固化分组的要旨

从第168页开始的“匹配优先、忽略优先和回溯的要旨”这一节，说明了一个重要的事实，即匹配优先和忽略优先都不会影响需要检测路径的本身，而只会影响检测的顺序。如果不能匹配，无论是按照匹配优先还是忽略优先的顺序，最终每条路径都会被测试。

然而，固化分组与它们截然不同，因为固化分组会放弃某些可能的路径。根据具体情况的不同，放弃备用状态可能会导致不同的结果：

1 毫无影响 如果在使用备用状态之前能够完成匹配，固化分组就不会影响匹配。我们刚刚看过‘.625000’的匹配。全局匹配在备用状态尚未起作用之前就已经完成。

1 导致匹配失败 放弃备用状态可能意味着，本来有可能成功的匹配现在不可能成功了。‘6.25’的例子就是如此。

1 改变匹配结果 在某些情况下，放弃备用状态可能得到不同的匹配结果。

1 加快报告匹配失败的速度 如果不能找到匹配对象，放弃备用状态可能能让正则引擎更快地报告匹配失败。先做个小测验，然后我们来谈这点。

v 小测验：「(>.*?)」是什么意思？它能匹配什么文本？请翻到下一页查看答案。

某些状态可能保留 在匹配过程中，引擎退出固化分组时，放弃的只是固化分组中创建的状态。而之前创建的状态依然保留，所以，如果后来的回溯要求退回到之前的备用状态，固化分组部分匹配的文本会全部交还。

使用固化分组加快匹配失败的速度 我们一眼就能看出，「^\w+:」无法匹配‘Subject’，因为‘Subject’中不含冒号，但正则引擎必须经过尝试才能得到这个结论。

第一次检查「:」时，「\w+」已经匹配到了字符串的结尾，保存了若干状态——「\w」匹配的每个字符，都对应有“忽略”的备用状态（第一个除外，因为加号要求至少匹配一次）。「:」无法匹配字符串的末尾，所以正则引擎会回溯到最近的备用状态：

‘Subject’	「^\w+:」
-----------	---------

此处的字符是‘t’，「:」仍然无法匹配。于是回溯-测试-失败的循环会不断发生，最终退回开始的状态：

‘Subject’	「^\w+:」
-----------	---------

此处仍然无法匹配成功，所以报告整个表达式无法匹配。

测验答案

v 171页测试的答案

「(>.*?)」会匹配什么？

它永远无法匹配任何字符。充其量它只能算是个相当复杂的正则表达式，但不匹配任何字符。「.*?’是「.*」的忽略优先表示，它限定的是一个点号，所以首选的分支就是忽略点号，把匹配点号的状态保留下来备用。但是，这个保存的状态马上又会被放弃，因为匹配退出了固化分组，所以真正尝试的只有忽略点号的分支。总是被忽略的东西，实际上相当于不存在。

我们只消看一眼就能知道，所有的回溯都是白费工夫。如果冒号无法匹配最后的字符，那么它当然无法匹配「+」交还的任何字符。

既然我们知道，「\w+」匹配结束之后，从任何备用状态开始测试都不能得到全局匹配，就可以命令正则引擎不必检查它们：「^(?>\w+)」。我们已经全面了解了正则表达式的匹配过程，可以使用固化分组来控制「\w+」的匹配，放弃备用的状态（因为我们知道它们没有用），提高效率。如果存在可以匹配的文本，那么固化分组不会有任何影响，但是如果不存在能够匹配的文本，放弃这些无用的状态会让正则引擎更快地得出无法匹配的结论（先进的实现也许能够自动进行这样的优化，F251）。

我们将在第6章看到（第269页），固化分组非常有价值，我怀疑它可能会成为最常用的技巧。

占有优先量词，?+、*+、++和{m,n}+

Possessive Quantifiers, ?+, ++, and {m,n}+

占有优先量词与匹配优先量词很相似，只是它们从来不交还已经匹配的字符。我们在「^\w+」的例子中看到，加号在匹配结束之前创建了很少的备用状态，而占有优先的加号会直接放弃这些状态（或者，更形象地说，并不会创造这些状态）。

你也许会想，占有优先量词和固化分组关系非常紧密。像「\w++」这样的占有优先量词与「(?:>\w+)」的匹配结果完全相同，只是写起来更加方便而已（注7）。使用占有优先量词，「^(?>\w+);」可以写作「^\w++;」，「(\d+(?>[1-9]?))d+」写做「(\d+[1-9]?+)\d+」。

请务必区分「(?:>M)+」和「(?:M+)」。前者放弃「M」创建的备用状态，因为「M」不会制造任何状态，所以这样做没什么价值。而后者放弃「M+」创造的未使用状态，这样做显然有意义。

比较「(?:>M)+」和「(?:M+)」，显然后者就对应于「M++」，但如果表达式很复杂，例如「(\\["^])*+」，从占有优先量词转换为固化分组时大家往往会想到在括号中添加「?>」得到「(?:>\\["^"])*」。这个表达式或许有机会实现你的目的，但它显然不等于那个使用占有优先量词的表达式；它就好像是把「M++」写作「(?:>M)+」一样。正确的办法是，去掉表示占有优先的加号，用固化分组把余下的部分包括起来：「(?:>(\\["^"])*)」。

环视的回溯

The Backtracking of Lookaround

初看时并不容易认识到，环视（见第2章，F59）与固化分组和占有优先量词有紧密的联系。环视分为4种：肯定型（positive）和否定型（negative），顺序环视与逆序环视。它们只是简单地测试，其中表达式能否在当前位置匹配后面的文本（顺序环视），或者前面的文本（逆序环视）。

深入点看，在NFA的世界中包含了备用状态和回溯，环视是怎么实现的？环视结构中的子表达式有自己的世界。它也会保存备用状态，进行必要的回溯。如果整个子表达式能够成功匹配，结果如何呢？肯定型环视会认为自己匹配成功；而否定型环视会认为匹配失败。在任何一种情况下，因为关注的只是匹配存在与否（在刚才的例子中，的确存在匹配），此匹配尝试所在的“世界”，包括在尝试中创造的所有备用状态，都会被放弃。

如果环视中的子表达式无法匹配，结果如何呢？因为它只应用到这个“世界”中，所以回溯时只能选择当前环视结构中的备用状态。也就是说，如果正则表达式发现，需要进一步回溯到当前的环视结构的起点以前，它就认为当前的子表达式无法匹配。对肯定型环视来说，这就意味着失败，而对于否定型环视来说，这意味着成功。在任何一种情况下，都没有保留备用状态（如果有，那么子表达式的匹配尝试就没有结束），自然也谈不上放弃。

所以我们知道，只要环视结构的匹配尝试结束，它就不会留下任何备用状态。任何备用状态和例子中肯定环视成功时的情况一样，都会被放弃。我们在其他什么地方看到过放弃状态？当然是固化分组和占有优先量词。

用肯定环视模拟固化分组

如果流派本身支持固化分组，这么做可能有点多此一举，但如果流派不支持固化分组，这么做就很有用：如果所使用的工具支持肯定环视，同时可以在肯定环视中使用捕获括号（大多数风格都支持，但也有些不支持，Tcl 就是如此），就能模拟实现固化分组和占有优先量词。`「(>regex)」` 可以用 `「(?(=regex))\1」` 来模拟。举例来说，比较 `「(>\w+);」` 和 `「^(?=(\w+))\1;」`。

环视中的 `「\w+」` 是匹配优先的，它会匹配尽可能多的字符，也就是整个单词。因为它在环视结构中，当环视结束之后，备用状态都会放弃（和固化分组一样）。但与固化分组不一样的是，虽然此时确实捕获了这个单词，但它不是全局匹配的一部分（这就是环视的意义）。这里的关键就是，后面的 `「\1」` 捕获的就是环视结构捕获的单词，而这当然会匹配成功。在这里使用 `「\1」` 并非多此一举，而是为了把匹配从这个单词结束的位置进行下去。

这个技巧比真正的固化分组要慢一些，因为需要额外的时间来重新匹配 `「\1」` 的文本。不过，因为环视结构可以放弃备用状态，如果冒号无法匹配，它的失败会来得更快一些。

多选结构也是匹配优先的吗

Is Alternation Greedy?

多选分支的工作原理非常重要，因为在不同的正则引擎中它们是迥然不同的。如果遇到多个多选分支都能够匹配，究竟会选择哪一个呢？或者说，如果不只一个多选分支能够匹配，最后究竟应该选择哪一个呢？如果选择的是匹配文本最长的多选分支，有人也许会说多选结构也是匹配优先的；如果选择的是匹配文本最短的多选分支，有人也许会说它是忽略优先的？那么（如果只能是一个的话）究竟是哪个？

让我们看看 Perl、PHP、Java、.NET 以及其他语言使用的传统型 NFA 引擎。遇到多选结构时，这种引擎会按照从左到右的顺序检查表达式中的多选分支。拿正则表达式 `「^(Subject|Date):-」` 来说，遇到 `「Subject|Date」` 时，首先尝试的是 `「Subject」`。如果能够匹配，就转而处理接下来的部分（也就是后面的 `「:-」`）。如果无法匹配，而此时又有其他多选分支（就是例子中的 `「Date」`），正则引擎会回溯来尝试它。这个例子同样说明，正则引擎会回溯到存在尚未尝试的多选分支的地方。这个过程会不断重复，直到完成全局匹配，或者所有的分支（也就是本例中的所有多选分支）都尝试穷尽为止。

所以，对于常见的传统型 NFA 引擎，用 `「tour|to|tournament」` 来匹配 `「three-tournaments-`

won’时，会得到什么结果呢？在尝试到‘three-_atournaments-won’时，在每个位置进行的匹配尝试都会失败，而且每次尝试时，都会检查所有的多选分支（并且失败）。而在这个位置，第一个多选分支‘tour’能够匹配。因为这个多选结构是正则表达式中的最后部分，‘tour’匹配结束也就意味着整个表达式匹配完成。其他的多选分支就不会尝试了。

因此我们知道，多选结构既不是匹配优先的，也不是忽略优先的，而是按顺序排列的，至少对传统型 NFA 来说是如此。这比匹配优先的多选结构更有用，因为这样我们能够对匹配的过程进行更多的控制——正则表达式的使用者可以用它下令：“先试这个，再试那个，最后试另一个，直到试出结果为止”。

不过，也不是所有的流派都支持按序排列的多选结构。DFA 和 POSIX NFA 确实有匹配优先的多选结构，它们总是匹配所有多选分支中能匹配最多文本的那个（也就是本例中的‘tournament’）。但是，如果你使用的是 Perl、PHP、.NET、java.util.regex，或者其他使用传统型 NFA 的工具，多选结构就是按序排列的。

发掘有序多选结构的价值

Taking Advantage of Ordered Alternation

回过头来看第167页‘(\.d\d[1-9]?)\d*’的例子。如果我们明白，‘\.d\d[1-9]?’其实等于‘\.d\d’或者‘\.d\d[1-9]’，我们就可以把整个表达式重新写作‘(\.d\d|\.d\d[1-9])\d*’。（这并非必须的改动，只是举例说明）。这个表达式与之前的完全一样吗？如果多选结构是匹配优先的，那么答案就是肯定的，但如果多选结构是有序的，两者就完全不一样。

我们来看多选结构有序的情形。首先选择和测试的是第一个多选分支，如果能够匹配，控制权就转移到紧接的‘\d*’那里。如果还有其他的数字，‘\d*’能够匹配它们，也就是任何不为零的数字，它们是原来问题的根源（如果读者还记得，当时的问题就在于，这位数字我们只希望在括号里匹配，而不通过括号外面的‘\d*’）。所以，如果第一个多选分支无法匹配，第二个多选分支同样无法匹配，因为二者的开头是一样的。即使第一个多选结构无法匹配，正则引擎仍然会对第二个多选分支进行徒劳的尝试。

不过，如果交换多选分支的顺序，变成‘(\.d\d[1-9]\.d\d)\d*’，它就等价于匹配优先的‘(\.d\d[1-9]?)\d*’。如果第一个多选分支结尾的‘[1-9]’匹配失败，第二个多选分支仍然有机会成功。我们使用的仍然是有序排列的多选结构，但是通过变换顺序，实现了匹配优先的功能。

第一次拆分‘[1-9]?’成两个多选分支时，我们把较短的分支放在了前面，得到了一个不具备匹配优先功能的‘?’。在这个具体的例子中，这么做没有意义，因为如果第一个多选分支不能匹配，第二个肯定也无法匹配。我经常看到这样没有意义的多选结构，对传统型 NFA 来说，这肯定不对。我曾看到有一本书以‘a((ab)*|b*)’为例讲解传统型 NFA 正则表达式的括号。这个例子显然没有意义，因为第一个多选分支‘(ab)*’永远也不会匹配失败，所以后面的其他多选分支毫无意义。你可以继续添加：

```
「a*((ab)*|b*|. *|partridge-in-a-pear-tree|[a-z])」
```

这个正则表达式的意义都不会有丝毫的改变。要记住的是，如果多选分支是有序的，而能够匹配同样文本的多选分支又不只一个，就要小心安排多选分支的先后顺序。

有序多选结构的陷阱

有序多选分支容许使用者控制期望的匹配，因此极为便利，但也会给不明就里的人造成麻烦。如果需要匹配‘Jan 31’之类的日期，我们需要的就不是简单的「Jan-[0123][0-9]」，因为这样的表达式能够匹配‘Jan-00’和‘Jan-39’这样的日期，却无法匹配‘Jan-7’。

一种办法是把日期部分拆开。用「0?[1-9]」匹配可能以0开头的前九天的日期。用「[12][0-9]」处理十号到二十九号，用「3[01]」处理最后两天。把上面这些连起来，就是「Jan-(0?[1-9]|[12][0-9]|3[01])」。

如果用这个表达式来匹配‘Jan 31 is Dad's birthday’，结果如何呢？我们希望获得的当然是‘Jan 31’，但是有序多选分支只会捕获‘Jan 3’。奇怪吗？在匹配第一个多选分支「0?[1-9]」时，前面的「0?」无法匹配，但是这个多选分支仍然能够匹配成功，因为「[1-9]」能够匹配‘3’。因为此多选分支位于正则表达式的末尾，所以匹配到此完成。

如果我们重新安排多选结构的顺序环视，把能够匹配的数字最短的放到最后，这个问题就解决了：「Jan-([12][0-9]|3[01]|0?[1-9])」。

另一种办法是使用「Jan-(31|[123]0|[012]?[1-9])」。但这也要求我们仔细地安排多选分支的顺序避免问题。还有一种办法是「Jan-(0[1-9]|[12][0-9]?|3[01]?[4-9])」，这样不论顺序环视如何都能获得正确结果。比较和分析这3个不同的表达式，会有很多发现（我会给读者一些时间来想这个问题，尽管下一页的补充内容会有所帮助）。

拆分日期的几种办法

下面几种办法都可以用来解决第176页的日期匹配问题。正则表达式中的元素能匹配日历中对应元素的部分。

NFA、DFA 和 POSIX

NFA, DFA, and POSIX

最左最长规则

“The Longest-Leftmost”

之前我们说过：如果传动装置在文本的某个特定位置启动 DFA 引擎，而在此位置又有一个或多个匹配的可能，DFA 就会选择这些可能中最长的。因为在所有同样从最左边开始的可能的匹配文本中它是最长的，所以叫它“最左最长的（longest-leftmost）”匹配。

绝对最长

这里说的“最长”不限于多选结构。看看 NFA 如何用「one(self)?(selfsufficient)?」来匹配

字符串 `oneselfsufficient`。NFA 首先匹配 `one`，然后是匹配优先的 `(self)?`，留下 `(selfsufficient)?` 来匹配 `sufficient`。它显然无法匹配，但整个表达式并不会因此匹配失败，因为这个元素不是必须匹配的。所以，传统型 NFA 返回 `oneselfsufficient`，放弃没有尝试的状态（POSIX NFA 的情况与此不同，我们稍后将会看到）。

与此相反，DFA 会返回更长的结果：`oneselfsufficient`。如果最开始的 `(self)?` 因为某些原因无法匹配，NFA 也会返回跟 DFA 一样的结果，因为 `(self)?` 无法匹配，`(selfsufficient)?` 就能成功匹配。传统型 NFA 不会这样，但是 DFA 则会这样，因为会选择最长的可能匹配。DFA 同时记录多个匹配，在任何时候都清楚所有的匹配可能，所以它能做到这一点。

我选这个简单的例子是因为它很容易理解，但是我希望读者能够明白，这个问题在现实中很重要。举例来说，如果希望匹配连续多行文本，常见的情况是，一个逻辑行（logical line）可以分为许多现实的行，每一行以反斜线结尾，例如：

```
SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c \
missing.c msg.c node.c re.c version.c
```

读者可能希望用 `^\w+=.*` 来匹配这种“`var=value`”的数据，但是正则表达式无法识别连续的行（在这里我们假设点号无法匹配换行符）。为了匹配多行，读者可能需要在表达式最后添加 `(\\n.*)*`，得到 `^\w+=.*(\\n.*)*`。显然，这意味着任何后继的逻辑行都能匹配，只要他们以反斜线结尾。这看起来没错，但在传统型 NFA 中行不通。`.*` 到达行尾的时候，已经匹配了反斜线，而表达式中后面的部分不会强迫进行回溯（F152）。但是，DFA 能够匹配更长的多行文本，因为它确实是最长的。

如果能够使用忽略优先的量词，也许可以考虑用它们来解决问题，例如 `^\w+=.*?(\\n.*?)*`。这样点号每次实际匹配任何字符之前，都需要测试转义的换行符部分，这样 `\\` 就能够匹配换行符之前的反斜线。不过这也行不通。如果忽略优先量词匹配某些可选的部分，必然是在全局匹配必须的情况下发生。但是在本例中，`=` 后面的所有部分都不是必须匹配的，所以没有东西会强迫忽略优先量词匹配任何字符。忽略优先的正则表达式只能匹配“`SRC=`”，这显然不是我们期望的结果。

这个问题还有其他的解决办法，我们会在下一章继续这个问题（F186）。

POSIX 和最左最长规则

POSIX and the Longest-Leftmost Rule

POSIX 标准规定，如果在字符串的某个位置存在多个可能的匹配，应当返回的是最长的匹配。

POSIX 标准文档中使用了“最左边开始的最长匹配（longest of the leftmost）”。它并没有规

定必须使用 DFA，那么，如果希望使用 NFA 来实践 POSIX，程序员应该如何做？如果你希望执行 POSIX NFA，那么必须找到完整的 `oneselfsufficient` 和所有的连续行，虽然这个结果是违反 NFA“天性”的。

传统型 NFA 引擎会在第一次找到匹配时停下来，但是如果让它继续尝试其他分支（状态）会怎样呢？每次匹配到表达式的末尾时，它都会获得另一个可能的匹配结果。如果所有的分支都穷尽了，就能从中选择最长的匹配结果。这样，我们就得到了一台 POSIX NFA。

在上面的例子中，NFA 匹配「(self)?」时保存了一个备用状态：「one(self)?」(selfsufficient)?」在「one」selfsufficient」。传统型 NFA 在 oneseelfsufficient 之后停止匹配，而 POSIX NFA 仍然会继续检查余下的所有状态，最终得到那个更长的结果（其实是最长的）oneseelfsufficient。

第7章有一个例子，可以让 Perl 模拟 POSIX 的做法，返回最长的匹配字符（F225）。

速度和效率

Speed and Efficiency

如果传统型 NFA 的效率是我们应当关注的问题（对提供回溯的传统型 NFA 来说，这确实是一个问题），那么 POSIX NFA 的效率就更值得关注，因为它需要进行更多的回溯。POSIX NFA 需要尝试正则表达式的所有变体（译注4）。第6章告诉我们，正则表达式写得糟糕的话，匹配的效率就会很低。

DFA 的效率

文本主导的 DFA 巧妙地避免了回溯造成的效率问题。DFA 同时记录了所有可能的匹配，这样来提高速度。它是如何做到这一切的呢？

DFA 引擎需要更多的时间和内存，它第一次遇见正则表达式时，在做出任何尝试之前会用比 NFA 详细得多的（也是截然不同的）办法来分析这个正则表达式。开始尝试匹配的时候，它已经内建了一张路线图（map），描述“遇到这个和这个字符，就该选择这个和那个可能的匹配”。字符串中的每个字符都会按照这张路线图来匹配。

有时候，构造这张路线图可能需要相当的时间和内存，不过只要建立了针对特定正则表达式的路线图，结果就可以应用到任意长度的文本。这就好像为你的电动车充电一样。首先，你得把车停到车库里面，插上电源等待一段时间，但只要发动了汽车，清洁的能源就会源源而来。

NFA 理论与现实

NFA（译注5）真正的数学和计算学意义是不同于通常说的“NFA 引擎”的。在理论上，NFA 和 DFA 引擎应该匹配完全一样的文本，提供完全一样的功能。但是在实际中，因为人们需要更强的功能，更具表达能力的正则表达式，它的语意发生了变化。反向引用就是一例。

DFA 引擎的设计方案就排除了反向引用，但是对于真正（数学意义上的）的 NFA 引擎来说，提供反向引用的支持只需要很小的改动。这样我们就得到了一个功能更强大的工具，但它绝对是非正则（nonregular）的（数学意义上的）。这是什么意思呢？或许，你不应该继续叫它“NFA”，而是“非正则表达式（nonregular expressions）”，因为这个名词才能描述（在数学意义上）新的情形。但是实际没人这么做，所以这个名字就这样流传下来，虽然实现方式都不再是（数学意义上的）NFA。

这对用户有什么意义？显然没有什么意义。作为用户，你不需要关心它是正则还是非正则，而只需要知道它能为你做什么（也就是本章的内容）。

对那些希望了解更多的正则表达式的理论的人，经典的计算机科学教学文本是，Aho、Sethi、Ullman 的 *Compilers—Principles, Techniques, and Tools* (Addison-Wesley, 1986) 的第3章，通常说的“恐龙书”，因为它的封面。更确切地说，这是一条“红龙”，“绿龙”指的是它的前任，Aho 和 Ullman 的 *Principles of Compiler Design*。

小结：NFA 与 DFA 的比较

Summary: NFA and DFA in Comparison

NFA 与 DFA 各有利弊。

DFA 与 NFA：在预编译阶段（pre-use compile）的区别

在使用正则表达式搜索之前，两种引擎都会编译表达式，得到一套内化形式，适应各自的匹配算法。NFA 的编译过程通常要快一些，需要的内存也更少一些。传统型 NFA 和 POSIX NFA 之间并没有实质的差别。

DFA 与 NFA：匹配速度的差别

对于“正常”情况下的简单文本匹配测试，两种引擎的速度差不多。一般来说，DFA 的速度与正则表达式无关，而 NFA 中两者直接相关。

传统的 NFA 在报告无法匹配以前，必须尝试正则表达式的所有变体。这就是为什么我要用整章（第6章）来论述提高 NFA 表达式匹配速度的技巧。我们将会看到，有时候一个 NFA 永远无法结束匹配。传统型 NFA 如果能找到一个匹配，肯定会停止匹配。

相反，POSIX NFA 必须尝试正则表达式的所有变体，确保获得最长的匹配文本，所以如果匹配失败，它所花的时间与传统型 NFA 一样（有可能很长）。因此，对 POSIX NFA 来说，表达式的效率问题更为重要。

在某种意义上，我说得绝对了一点，因为优化措施通常能够减少获得匹配结果的时间。我们已经看到，优化引擎不会在字符串开头之外的任何地方尝试带「^」锚点的表达式，我们会在第6章看到更多的优化措施。

DFA 不需要做太多的优化，因为它的匹配速度很快，不过最重要的是，DFA 在预编译阶段所作的工作提供的优化效果，要好于大多数 NFA 引擎复杂的优化措施。

现代 DFA 引擎经常会尝试在匹配需要时再进行预编译，减少所需的时间和内存。因为应用的文本各异，通常情况下大部分的预编译都是白费工夫。因此，如果在匹配过程确实需要的情况下再进行编译，有时候能节省相当的时间和内存（技术术语就是“延迟求值（lazy evaluation）”）。这样，正则表达式、待匹配的文本和匹配速度之间就建立了某种联系。

DFA 与 NFA：匹配结果的差别

DFA（或者 POSIX NFA）返回最左边的最长的匹配文本。传统型 NFA 可能返回同样的

结果，当然也可能是别的文本。针对某一型具体的引擎，同样的正则表达式，同样的文本，总是得到同样的结果，在这个意义上来说，它不是“随机”的，但是其他 NFA 引擎可能返回不一样的结果。事实上，我见过的所有传统型 NFA 返回的结果都是一样的，但并没有任何标准来硬性规定。

DFA 与 NFA：能力的差异

NFA 引擎能提供一些 DFA 不支持的功能，例如：

1 捕获由括号内的子表达式匹配的文本。相关的功能是反向引用和后匹配信息（*after-match information*），它们描述匹配的文本中每个括号内的子表达式所匹配文本的位置。

1 环视，以及其他复杂的零长度确认（注8）（F133）。

1 非匹配优先的量词，以及有序的多选结构。DFA 很容易就能支持选择最短的匹配文本（尽管因为某些原因，这个选项似乎从未向用户提供过），但是它无法实现我们讨论过的局部的忽略优先性和有序的多选结构。

1 占有优先量词（F142）和固化分组（F139）。

兼具 DFA 的速度和 NFA 的功能：正则表达式的终极境界

我已经多次说过，DFA 不能支持捕获括号和反向引用。这无疑是对的，但这并不是说，我们不能组合不同的技术，以达到正则表达式的终极境界。180页的补充内容描述了 NFA 为了追求更强大的功能，如何脱离了纯理论的道路和限制，DFA 的情况也是如此。受自身结构的限制，DFA 进行这种突破更加困难，但并非不可能。

GNU *grep* 采取了一种简单但有效的策略。它尽可能多地使用 DFA，在需要反向引用的时候，才切换到 NFA。GNU *awk* 的办法也差不多——在进行“是否匹配”的检查时，它采用 GNU *grep* 的 DFA 引擎，如果需要知道具体的匹配文本的内容，就采用不同的引擎。这里的“不同的引擎”就是 NFA，利用自己的 *gensub* 函数，GNU *awk* 能够很方便地提供捕获括号。

Tcl 的正则引擎由 Henry Spencer（你或许记得，这个人在正则表达式的早期发展和流行中扮演了重要的角色）开发，它也是混合型的。Tcl 引擎有时候像 NFA——它支持环视、捕获括号、反向引用和忽略优先量词。但是，它也确实能提供 POSIX 的最左最长匹配（F177），但没有我们将在第6章看到的 NFA 的问题。这点确实很棒。

DFA 与 NFA：实现难度的差异

尽管存在限制，但简单的 DFA 和 NFA 引擎都很容易理解和实现。对效率（包括时间和空间效率）和增强性能的追求，令实现越来越复杂。

用代码长度来衡量的话，支持 NFA 正则表达式的 *ea* Version 7（1979年1月发布）只有不到350行的 C 代码（所以，整个 *grep* 只有区区478行代码）。Henry Spencer 1986年免费提供的 Version 8 正则程序差不多有1 900行 C 代码，1992年 Tom Lord 的 POSIX NFA package *rx*（被 GNU *sed* 和其他工具采用）长达9 700行。

为了糅合 DFA 和 NFA 的优点，GNU *egrep* Version 2.4.2使用了两个功能完整的引擎（差不多8 900行代码），Tcl 的 DFA/NFA 混合引擎（请看上一页的补充内容）更是长达9 500行。

某些实现很简单，但这并不是说它们支持的功能有限。我曾经想要用 **Pascal** 的正则表达式来处理某些文本。从毕业以后我就没用过 **Pascal** 了，但是写个简单的 NFA 引擎并不需要太多工夫。它并不追求花哨，也不追求速度，但是提供了相对全面的功能，非常实用。

总结

Summary

如果你希望一遍就能读懂本章的所有内容，大概得做点准备。至少，这些东西不那么容易理解。我花了些时间才理解它，花了更长的时间才真正弄懂。我希望这章简要的讲解能够降低读者理解的难度。我尝试过简单地解释，同时不要调入太简单的陷阱（不幸的是，太过直白的解释总是妨碍了真正的理解）。本章有许多这样的陷阱，所以我在其中安排了许多对其他页的引用，在下面的摘要中，读者可以很快地找到其他的内容。

实现正则表达式匹配引擎有两种常见的技术，一种是“表达式主导的 NFA”（F153），另一种是“文本主导的 DFA”（F155）。它们的全称见第156页。

这两种技术，结合 **POSIX** 标准，可以按照实用标准划分3种正则引擎：

- 1 传统型 NFA（汽油驱动，功能强大）。
- 1 **POSIX** NFA（汽油驱动，符合标准）。
- 1 **DFA**（不一定符合 **POSIX**）（电力驱动，运转稳定）。

为了对手头的工具有个大致的了解，你需要知道它采用的是什么引擎，以对正则表达式做相应的调校。最常见的引擎就是传统型 NFA，其次是 **DFA**。表4-1（F145）列出了若干常用工具及它们使用的引擎类型，“测试引擎的类型”（F146）给出了测试引擎类型的方法。

对于任何引擎来说，都有一条通用的规则：开始位置靠先的匹配文本优先于开始位置靠后的匹配文本。因为“传动机构”会从前往后在文本的各个位置展开尝试（F148）。

对于从某个位置开始的匹配：

文本主导的 **DFA** 引擎：

寻找可能的最长的匹配文本。不再介绍（F177）。稳定、速度快（F179），讲解起来很麻烦。

表达式主导的 **NFA** 引擎：

匹配过程中可能需要“反复尝试（work through）”。**NFA** 匹配的灵魂是回溯（F157，162）。控制匹配过程的元字符：标准量词（星号等等）是匹配优先的（F151），其他量词是忽略优

先或者占有优先的（F169）。在传统型 NFA 中，多选结构是有序排列的（F174），在 POSIX NFA 中是匹配优先的。

POSIX NFA 必须找到最长的匹配文本。但是，匹配并不难理解，只须考虑效率（第6章的问题）。

传统型 NFA 控制能力最强的正则引擎，因此使用者可以使用该引擎的表达式主导性质来精确控制匹配过程。

理解本章的概念和练习是书写正确而高效的正则表达式的基础，这也是接下来两章的主题。

正则表达式实用技巧

Practical Regex Techniques

现在我们已经掌握了编写正则表达式所需的基本知识，我希望在更复杂的环境中应用这些知识来处理更复杂的问题。每个正则表达式都必须在下面两个方面求得平衡：准确匹配期望匹配的内容，忽略不期望匹配的字符。我们已经看过许多例子都说明，如果应用得当，匹配优先非常有用，但如果不够小心，也可能带来麻烦，在本章我们还将看到许多例子。

NFA 引擎还需要平衡另外一个因素：效率，这也是下一章的主题。设计糟糕的正则表达式——即使可以认为没犯错误——也足以让引擎瘫痪。

本章出现的主要是各种实例，我会带领读者循着我的思路去解决各种问题。某些例子或许对读者并没有现实价值，但我仍然推荐读者阅读这些实例。

例如，即使你的工作不涉及 HTML，我仍推荐你从处理 HTML 的实例中吸取知识。原因在于，编写巧妙的正则表达式不仅仅是一种手艺（skill）——而且还是一种艺术（art）。它的教授和学习，不是依靠罗列规则，而是依靠经验，所以，我用这些例子告诉读者，自己在过去的若干年从经验中获得的深刻启示。

当然，读者仍然需要自己掌握这些知识，但是研究本章的例子是个好的起点。

正则表达式的平衡法则

Regex Balancing Act

好的正则表达式必须在这些方面求得平衡：

- 1 只匹配期望的文本，排除不期望的文本。

- 1 必须易于控制和理解。

- 1 如果使用 NFA 引擎，必须保证效率（如果能够匹配，必须很快地返回匹配结果，如果不能匹配，应该在尽可能短的时间内报告匹配失败）。

这些方面常常是与具体文本相关的。如果我只使用命令行，只需要快速地 *grep* 某些东西，可能不会过分关心匹配的准确性，通常也不会花太多精力来调校。我不在乎多花点时间来手工排查，因为我能够迅速地在输出中找到自己需要的内容。但是，如果处理重要的程序，就需要花费时间精力来保证正确性：如果需要，正则表达式也可能很复杂。这些因素都需要权衡。

即使使用同样的程序，效率也是与具体文本相关的。如果是 NFA，用「`^(display|geometry|cemap|...|quick24|random|raw)$`」之类长长的正则表达式来检验命令行参数的效率就很低，因为多选分支过多，但如果它只用于检验命令行参数（可能只是在程序开始的时候运行若干次），即使所需时间比正常的长100倍也不要紧，因为这时候效率并不是问题。但是，如果要逐行检查很大的文件，低效率的程序运行起来会让你痛苦不堪。

若干简单的例子

A Few Short Examples

匹配连续行（续前）

Continuing with Continuation Lines

继续前一章中匹配连续行的例子（F178），我们发现(在传统型 NFA 中使用「`^w+.*(\\n.*)*`」并不能匹配下面的两行文本：

```
SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c\  
  
missing.c msg.c node.c re.c version.c
```

问题在于，第一个「`.*`」一直匹配到反斜线之后，这样「`(\\n.*)*`」就不能按照预期匹配反斜线了。所以，本章出现的第一条经验就是：如果不需要点号匹配反斜线，就应该在正则表达式中做这样的规定。我们可以把每个点号替换成「`^[^n\\]`」（请注意，`n` 包含在排除性字符组中。你应该记得，原来的正则表达式的假设之一就是，点号不会匹配换行符，我们也不希望它的替代品能够匹配换行符 F119页）。

于是，我们得到：

```
「^w+=[^[^n\\]*(\\n[^[^n\\]]*)*」
```

它确实能够匹配连续行，但因此也产生了一个新的问题：这样反斜线就不能出现在一行的非结尾位置。如果需要匹配的文本中包含其他的反斜线，这个正则表达式就会出问题。现在在我们假设它会包含，所以需要继续改进正则表达式。

迄今为止，我们的思路都是，“匹配一行，如果还有连续行，就继续匹配”。现在换另一种思路，这种思路我觉得通常都会奏效：集中关注在特定时刻真正容许匹配的字符。在匹配一行文本时，我们期望匹配的要么是普通（除反斜线和换行符之外）字符，要么是反斜线与其他任何字符的结合体。在点号通配模式中，「`\\`」能匹配反斜线加换行符的结合体。

所以，正则表达式就变成了「`^w+=[^[^n\\]|\\.*]`」，在点号通配模式下。因为开头是「`^`」，如果需要，可能得使用增强的文本行锚点匹配模式（F112）。

但是，这个答案仍然不够完美——我们会在下一章讲解效率问题时再次看到它（F270）。

匹配 IP 地址

Matching an IP Address

来看个复杂点的例子，匹配一个 IP（Internet Protocol，因特网协议）地址：用点号分开的四个数，例如1.2.3.4。通常情况下，每个数都有三位，例如001.002.003.004。你可能会想到用「`[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*`」从文本中提取一个 IP 地址，但是这个表达式显然不够精致，它甚至会匹配‘and then?’。仔细看看就会发现，这个表达式甚至不需要匹配任何数字——它只需要三个点号（当然也可能包括其间的数字）。

为解决这个问题，我们首先把星号改成加号，因为我们知道，每一段必须有至少一位数字。为确保整个字符串的内容就是一个 IP 地址，我们可以在首尾加上「`^...$`」，于是我们得到：

```
「^([0-9]+\.[0-9]+\.[0-9]+\.[0-9])+$」
```

如果用「`\d`」替换「`[0-9]`」，就得到「`^\d+\.\d+\.\d+\.\d+$`」，这样可能更好看一些（注1），但是，这个表达式仍然会捕获一些并非 IP 地址的数据，例如‘1234.5678.9101112.131415’

（IP 地址的每个字段都在0-255以内）。那么，你可以强行规定每个字段必须包含三位数字，就是「`^\d\d\d\d\d\d\d\d\d\d\d\d\d\d\d$`」，但这样未免太不灵活（too specific）了。即使某个字段只有一位或者两位数字（例如1.234.5.67），也应该匹配。如果流派支持区间量词 {*min,max*}，就可以这么写「`^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$`」。如果不支持，则可以用「`\d\d?\d?`」或者「`\d(\d\d)?`」。这两种方式略有不同，但都能匹配一到三个数字。

现在，正则表达式中的匹配精度可能已经满足需求了。如果要更精确，就必须考虑到，「`\d{1,3}`」能够匹配999，而它超过了255，所以它不是一个合法的 IP 地址。

我们有好几种办法来匹配0和255之间的数字。最笨的办法就是「`0|1|2|3|...253|254|255`」。不过这又不能处理以0开头的数字，所以必须写成「`0|00|000|1|01|001|...`」，这样一来，正则表达式就长得过分了。对于 DFA 引擎来说，问题还只是它太长太繁杂——但匹配的速度与其他等价正则表达式是一样的。但对于 NFA 引擎，太多的多选分支简直就是效率杀手。

实际的解决办法是，关注字段中什么位置可以出现哪些数字。如果一个字段只包含一个或者两个数字，就无需担心这个字段的值是否合法，所以「`\d\d\d`」就能应付。也不比担心那些以0或者1开头的三位数，因为000-199都是合法的 IP 地址。所以我们加上「`[01]\d\d`」，得到「`\d\d\d|[01]\d\d`」。你可能觉得这有点像第1章里匹配时间的例子（F28），和前一章中匹配日期的例子（F177）。

继续看这个正则表达式，以2开头的三位数字，如果小于255就是合法的，所以第二位数字小于5就代表整个数也是合法的。如果第二位数字是5，第三位数字就必须小于6。这可以表示为「`2[0-4]\d|25[0-5]`」。

现在这个正则表达式有点看不懂了，但分析之后还是能够理解其中包含的思路。结果就是「`\d\d\d|[01]\d\d|2[0-4]\d|25[0-5]`」。其实我们可以合并前面三个多选分支，得到

「[01]?d\d?|2[0-4]\d|25[0-5]」。在 NFA 中，这样做的效率更高，因为任何多选分支匹配失败都会导致回溯。请注意，第一个多选分支中用的是「\d\d?」，而不是「\d?d」，这样，如果根本不存在数字，NFA 会更快地报告匹配失败。我把这个问题的分析留给读者——通过一个简单的验证就能发现二者的区别。我们还可以做些修改进一步提高这个表达式的效率，不过这要留待下一章讨论了。

现在这个表达式能够匹配0到255之间的数，我们用括号把它包起来，用来取代之前表达式中的「d{1,3}」，就得到：

```
「^([01]?d\d?|2[0-4]\d|25[0-5])\.([01]?d\d?|2[0-4]\d|25[0-5])\.([01]?d\d?|2[0-4]\d|25[0-5])\.([01]?d\d?|2[0-4]\d|25[0-5])$」
```

这可真叫复杂！需要这么麻烦吗？这得根据具体需求来决定。这个表达式只会匹配合法的 IP 地址，但是它也会匹配一些语意不正确的 IP 地址，例如0.0.0.0（所有字段都为零的 IP 地址是非法的）。使用环视功能（F133）可以在「^」后添加「(?:0+\.0+\.0+\.0+\$)」，但是某些时候，处理各种极端情形会降低成本/收益的比例。某些情况下，更合适的做法就是不依赖正则表达式完成全部工作。例如，你可以只使用「^d{1,3}\.d{1,3}\.d{1,3}\.d{1,3}\$」，用括号把每个字段括起来，把数字变成程序中的\$1、\$2、\$3、\$4，这样就可以用其他程序来验证了。

确定应用场合（context）

这个正则表达式必须借助锚点「^」和「\$」才能正常工作，认识到这一点很重要。否则，它就可能匹配 ip=72123.3.21.993，如果使用传统型 NFA，则可能匹配 ip=123.3.21.223。

在第二个例子中，这个表达式甚至连最后的223都无法完整匹配。但是，问题并不在于表达式本身，因为没有东西（例如分隔符，或者末尾的锚点）强迫它匹配223。最后那个分组的第一个多选分支「[01]?d\d?」，匹配了前面两位数字，如果末尾没有「\$」，匹配到此就结束了。在前一章日期匹配的例子中，我们可以安排多选分支的次序来达到期望的目的。现在我们也把能把匹配三位数字的多选分支放在最前面，这样在匹配两位数的多选分支获得尝试机会之前，任何三位数都能完全匹配（DFA 和 POSIX NFA 当然不需要这样安排，因为它们总是返回最长的匹配文本）。

无论是否重新排序，第一个错误仍然不可避免。“啊哈！”，你可能会想，“我可以用单词分界符锚点来解决这个问题。”不幸的是，这也不能奏效，因为这样的正则表达式仍然能够匹配1.2.3.4.5.6。为了避免匹配这样内嵌的文本，我们必须确保匹配文本两侧至少没有数字或者点号。如果使用环视，可以在原来表达式的首尾添加「(?:[^\w.]|...(?:[^\w.]|」来保证匹配文本之前（以及之后）不出现「[^\w.]」能匹配的字符。如果不支持环视，在首尾添加「(^|...(-|\$)」也能够应付某些情况。

处理文件名

Working with Filenames

处理文件名和路径，例如 Unix 下的/usr/local/bin/Perl 或者 Windows 下的\Program Files\Yahoo!\Messenger，很适合用来讲解正则表达式的应用。因为“动手（using）”比“观摩

（reading）”更有意思，我会同时用 Perl、PHP（preg 程序）、Java 和 VB.NET 来讲解。如果你对其中的某些语言不感兴趣，不妨完全跳过那些代码——其中蕴含的思想才是最重要的。

去掉文件名开头的路径

第一个例子是去掉文件名开始的路径，例如把/usr/local/bin/gcc 变成 gcc。从本质层面来考虑问题是成功的一半。在本例中，我们希望去掉在最后的斜线（含）之前（在 Windows 中是反斜线）的任何字符。如果没有斜线最好，因为什么也不用干。我曾说过，「.*」常常被滥用，但是此处我们需要匹配优先的特性。「^.*」中的「.*」可以匹配一整行，然后回退（也就是回溯）到最后的斜线，来完成匹配。

下面是四种语言的代码，去掉变量 f 中的文件名中开头的路径。对于 Unix 的文件名：

语 言	代 码
Perl	\$f =~ s{^.*//}{};
PHP	\$f = preg_replace('^.*//', '', \$f);
java.util.regex	f = f.replaceFirst("^.*//", "");
VB.NET	f = Regex.Replace(f, "^.*//", "");

正则表达式（或者说用来表示正则表达式的字符串）以下画线标注，正则表达式相关的组件则由粗体标注。

下面是处理 Windows 文件名的代码，Windows 中的分隔符是反斜线而不是斜线，所以要用正则表达式「^.*\\」。在正则表达式中，我们需要在反斜线前再加一个反斜线，才能表示转义的反斜线，不过，在中间两段程序中添加的这个反斜线本身也需要转义：

语 言	代 码
Perl	\$f =~ s/^.*\\//;
PHP	\$f = preg_replace('/^.*\\\\\\/', '', \$f);
java.util.regex	f = f.replaceFirst("^.*\\\\\\\\", "");
VB.NET	f = Regex.Replace(f, "^.*\\\", "");

从中很容易看出各种语言的差异，尤其是 Java 中那4个反斜线（F101）。

有一点请务必记住：别忘了时常想想匹配失败的情形。在本例中，匹配失败意味着字符串中没有斜线，所以不会替换，字符串也不会变化，而这正是我们需要的。

为了保证效率，我们需要记住 NFA 引擎的工作原理。设想下面这种情况：我们忘记在正则表达式的开头添加「^」符号（这个符号很容易忘记），用来匹配一个恰好没有斜线的字符串。同样，正则引擎会在字符串的起始位置开始搜索。「.*」抵达字符串的末尾，但必须不断回退，以找到斜线或者反斜线。直到最后它交还了匹配的所有字符，仍然无法匹配。所以，正则引擎知道，在字符串的起始位置不存在匹配，但这远远没有结束。

接下来传动装置开始工作，从在目标字符串的第2个字符开始，依次尝试匹配整个正则表达式。事实上，它需要在字符串的每个位置（从理论上说）进行扫描-回溯。文件名通常很短，因此这不是一个问题，但原理确实如此。如果字符串很长，就可能存在大量的回溯（当然，DFA 不存在这个问题）。

在实践中，经过合理优化的传动装置能够认识到，对几乎所有以「.*」开头的正则表达式来说，如果在某个字符串的起始位置不能匹配，也就不能在其他任何位置匹配，所以它只会在字符串的起始位置（F246）尝试一次。不过，在正则表达式中写明这一点更加明智，在例子中我们正是这样做的。

从路径中获取文件名

另一种办法是忽略路径，简单地匹配最后的文件名部分。最终的文件名就是从最后一个斜线开始的所有内容：「`[/]*$`」。这一次，锚点不仅仅是一种优化措施，我们确实需要在结尾设置一个锚点。现在我们可以这样做，以 Perl 来说明：

```
$WholePath =~ m{[/]*$}; # 利用正则表达式检测$wholePath
```

```
$Filename = $1; # 记录匹配内容
```

你也许注意到了，这里并没有检查这个正则表达式能否匹配，因为它总是能匹配。这个表达式的唯一要求就是，字符串有「\$」能够匹配的结束位置，而即使是空字符串也有一个结束位置。因此，我用「\$1」来引用括号内的表达式匹配的文本，因为它必定包括某些字符（如果文件名以斜线结尾，结果就是空字符）。

这里还需要考虑到效率：在 NFA 中，「`[/]*$`」的效率很低。仔细想想 NFA 引擎的匹配过程，你会明白它包括了太多的回溯。即使是短短的「`/usr/local/bin/perl`」，在获得匹配结果

之前，也要进行四十多次回溯。考虑从「...ocal...」开始的尝试。「`[/]*`」一直匹配到第二个 l，之后匹配失败，然后对 l、a、c、o、l 的存储状态依次尝试「\$」（都无法匹配）。如果这还不够，又会从「...l_ocal/...」开始重复这个过程，接着从「...lo_cal/...」开始，不断重复。

这个例子不应该消耗我们太多的精力，因为文件名一般都很短（40 次回溯几乎可以忽略不计——4 000 万次回溯才真正要紧）。再一次，重要的是理解问题本身，这样才能选择合适的通用规则来解决具体的问题。

需要指出的是，纵然本书是关于正则表达式的，但正则表达式也不总是最优解。例如，大多数程序设计语言都提供了处理文件名的非正则表达式函数。不过为了讲解正则表达式，我仍会继续下去。

所在路径和文件名

下一步是把完整的路径分为所在路径和文件名两部分。有许多办法做到这一点，这取决于我们的要求。开始，你可能想要用「`^(.*)/(.*)$`」的 \$1 和 \$2 来提取这两者。看起来这个正则表达式非常直观，但知道了匹配优先量词的工作原理之后，我们知道第一个「.*」会首先捕获所有的文本，而不给「/」和 \$2 留下任何字符。第一个「.*」能交还字符的唯一原因，就是在尝试匹配「`/(.*)$`」时进行的回溯。这会把“交还的”部分留给后面的「.*」。因此，\$1 就是文件所在的路径，\$2 就是文件的名称。

需要注意的是，我们依靠开头的「`(.*)/`」来确保第二个「`(.*)`」不会匹配任何斜线。理解匹配优先之后，我们知道这没问题。如果要做的更精确，可以使用「`[/]*`」来捕捉文件名。于是

我们得到「`^(.*)/([^\]*)$`」。这个表达式准确地表达了我们的意图，一眼就能看明白。

这个表达式有个问题，它要求字符串中必须出现一个斜线，如果我们用它来匹配 `file.txt`，因为无法匹配，所以没有结果。如果我们希望精益求精，可以这样：

```
if ($WholePath = ~m!^(.*) / ([^\]*)$!) {  
  
# 能够匹配 --$1和$2都合法  
  
$LeadingPath = $1;  
  
$FileName = $2;  
  
} else {  
  
# 无法匹配, 文件名中不包含 '/'  
  
$LeadingPath = "."; # 所以"file.txt"应该是"./file.txt" ( "."表示当前路径)  
  
$FileName = $WholePath;  
  
}
```

匹配对称的括号

Matching Balanced Sets of Parentheses

对称的圆括号、方括号之类的符号匹配起来非常麻烦。在处理配置文件和源代码时，经常需要匹配对称的括号。例如，解析 C 语言代码时可能需要处理某个函数的所有参数。函数的参数包含在函数名称之后的括号里，而这些参数本身又有可能包含嵌套的函数调用或是算式中的括号。我们先不考虑嵌套的括号，你或许会想到「`bfoo\([^\]*)\`」，但这行不通。

秉承 C 的光荣传统，我把示范函数命名为 `foo`。表达式中的标记部分是用来捕获参数的。对于 `foo(2,-4.0)`和 `foo(somevar,-3.7)`之类的参数，这个表达式完全没问题。但是，它也可以匹配 `foo(bar(somevar),-3.7)`，这可不是我们需要的。所以要用到比「`[^\]*`」更聪明的办法。

为了匹配括号部分，我们可以尝试下面的这些正则表达式：

1. `\(.*)` 括号及括号内部的任何字符。
2. `\([^\]*\)` 从一个开括号到最近的闭括号。
3. `\([^(]*\)` 从一个开括号到最近的闭括号，但是不容许其中包含开括号。

图5-1显示了对一行简单代码应用这些表达式的结果。

图5-1：三个表达式的匹配位置

我们看到，第一个正则表达式匹配的内容太多（注2），第二个正则表达式匹配的内容太少，第三个正则表达式无法匹配。孤立地看，第三个正则表达式能够匹配‘(this)’，但是因为表达式要求它必须紧接在 `foo` 之后，所以无法匹配。所以，这三个表达式都不合格。

真正的问题在于，大多数系统中，正则表达式无法匹配任意深度的嵌套结构。在很长的时间内，这是放之四海而皆准的规则，但是现在 Perl、.NET 和 PCRE/PHP 都提供了解决的办法（参见第328、436、475页）。但是，即使不用这些功能，我们也可以用正则表达式来匹配特定深度的嵌套括号，但不是任意深度的嵌套括号。处理单层嵌套的正则表达式是：

```
「\[^O]*\(\([^\^O]*)\)[^\^O]*)」
```

这样类推下去，更深层次的嵌套就复杂得可怕。但是，下面的 Perl 程序，在指定嵌套深度 `$depth` 之后，生成的正则表达式可以匹配最大深度为 `$depth` 的嵌套括号。它使用的是 Perl 的“`string x count`”运算符，这个运算符会把 `string` 重复 `count` 次：

```
$regex = '\C' . '?(?:\[^O]\C x $depth . '\[^O]*' . '\))' * x $depth . '\)';
```

这个表达式留给读者分析。

防备不期望的匹配

Watching Out for Unwanted Matches

有个问题很容易忘记，即，如果待分析的文本不符合使用者的预期，会发生什么。假设你需要编写一个过滤程序，把普通文本转换为 HTML，你希望把一行连字符号转换为 HTML 中代表一条水平线的 `<HR>`。如果使用搜索-替换命令 `s/-*/<HR>/`，它能替换期望替换的文本，但只限于它们在行开头的情况。很奇怪吗？事实上，`s/-*/<HR>/` 会把 `<HR>` 添加到每一行的开头，而无论这些行是否以连字符开头。

请记住，任何文本，如果不是匹配必须的，一旦匹配之后，通常都被认为是成功的。「`-*`」从字符串的起始位置开始尝试匹配，它会匹配可能的任何连字符。但是，如果没有连字符，它仍然能匹配成功，这完全符合星号的定义。

在某位我非常尊重的作者的作品中出现过类似的例子，他用这个例子来讲解正则表达式匹配一个数，或者是整数或者是浮点数。在它的正则表达式中，这个数可能以负数符号开头，

然后是任意多个数字，然后是可能的小数点，再是任何多的数字。他的正则表达式是「-?[0-9]*\.[0-9]*」。

确实，这个正则表达式可以匹配1、-272.37、129238843.、.191919，甚至是-.0这样的数。这样看来，它的确是个不错的正则表达式。

但是，你想过这个表达式如何匹配‘this-has-no-number’‘nothing-here’或是空字符串吗？仔细看看这个正则表达式——每一个部分都不是匹配必须的。如果存在一个数，如果正则表达式从在字符串的起始位置开始，的确能够匹配，但是因为匹配没有任何必须元素。

此正则表达式可以匹配每个例子中字符串开头的空字符。实际上它甚至可以匹配‘num=123’开头的空字符，因为这个空字符比数字出现得更早。

所以，把真正意图表达清楚是非常重要的。一个浮点数必须要有至少一位数字，否则就不是一个合法的值。我们首先假设，在小数点之前至少有一位数字（之后我们会去掉这个条件）。如果是，我们需要用加号来控制这些数字「-[0-9]+」。

如果要用正则表达式来匹配可能存在的小数点（及其后的数字），就必须认识到，小数部分必须紧接在小数点之后。如果我们简单地用「\.[0-9]*」，那么无论小数点是否存在，「[0-9]*」都可能匹配。

解决的办法还是厘清我们的意图：小数点（以及之后的数字）是可能出现的：「(\.[0-9]*)?」。这里，问号限定（也可以叫“统治 governs”或者“控制 controls”）的不再是小数点，而是小数点和后面的小数部分。在这个结合体内部，小数点是必须出现的，如果没有小数点，「[0-9]*」根本谈不上匹配。

把它们结合起来，就得到「-[0-9]+(\.[0-9]*)?」。这个表达式不能匹配‘.007’，因为它要求整数部分必须有一位数字。如果我们作些修改，容许整数部分为空，就必须同时修改小数部分，否则这个表达式就可以匹配空字符（这是我们一开始就准备解决的问题）。

解决的办法是为无法覆盖的情况添加多选分支：「-[0-9]+(\.[0-9]*)?|-\.[0-9]+」。这样就能匹配以小数点开头的小数（小数点是必须的）。仔细看看，仔细看看。你注意到了吗？第二个多选分支同样能够匹配负数符号开头的小数？这很容易忘记。当然，你也可以把「-?」提出来，放到所有多选结构的外面：「-?([0-9]+(\.[0-9]*)?|-\.[0-9]+)」。

虽然这个表达式比最开始的好得多，但它仍然会匹配‘2003.04.12’这样的数字。要想真正匹配期望的文本，同时忽略不期望的文本，求得平衡，就必须了解实际的待匹配文本。我们用来提取浮点数的正则表达式必须包含在一个大的正则表达式内部，例如用「^...\$」或者「num\s*=\s*...\$」。

匹配分隔符之内的文本

Matching Delimited Text

匹配用分隔符（以某些字符表示）之类的文本是常见的任务，之前的匹配双引号内的文本和 IP 地址只是这类问题中的两个典型例子。其他的例子还包括：

- 1 匹配‘/*’和‘*/’之间的 C 语言注释。
- 1 匹配一个 HTML tag，也就是尖括号之内的文本，例如<CODE>。
- 1 提取 HTML tag 标注的文本，例如在 HTML 代码‘a <I>super exciting</I> offer!’中的‘super exciting’。
- 1 匹配 *mailrc* 文件中的一行内容。这个文件的每一行都按下面的数据格式来组织：

alias 简称 电子邮件地址

例如 ‘alias jeff jfriedl@regex.info’（在这里，分隔符是每个部分之间的空白和换行符）。

- 1 匹配引文字符串（quoted string），但是容许其中包含转义的引号，例如‘a passport needs a "2"x3" likeness" of the holder’。
- 1 解析 CSV（逗号分隔值，comma-separated values）文件。

总的来说，处理这些任务的步骤是：

1. 匹配起始分隔符（opening delimiter）。
2. 匹配正文（main text，即结束分隔符之前的所有文本）。
3. 匹配结束分隔符。

我曾经说过，如果结束分隔符不只一个字符，或者结束分隔符能够出现在正文中，这种任务就很难完成。

容许引文字符串中出现转义引号

来看2"x3"的例子，这里的结束分隔符是一个引号，但正文也可能包含转义之后的引号。匹配开始和结束分隔符很容易，诀窍就在于，匹配正文的时候不要超越结束分隔符。

仔细想想正文里能够出现的字符，我们知道，如果一个字符不是引号，也就是说如果这个字符能由「`[^"]`」匹配，那么它肯定属于正文。不过，如果这个字符是一个引号，而它前面又有一个反斜线，那么这个引号也属于正文。把这个意思表达出来，使用环视（F133）功能来处理“如果之前有反斜线”的情况，就得到「`([""](?<=\\)")*`」，这个表达式完全能够匹配2"x3"。

不过，这个例子也能用来说明，看起来正确的正则表达式如何会匹配意料之外的文本，它虽然看起来正确，但不是任何情况下都正确。我们希望它匹配下面这个无聊的例子中的划线部分：

Darth Symbol: `/-|-\\` or `"[^-]"`

但它匹配的是：

Darth Symbol: `/-|-\\` or `"[^-]"`

这是因为，第一个闭引号之前的确存在一个反斜线。但这个反斜线本身是被转义的，它不是用来转义之后的双引号的（也就是说这个引号其实是表示引用文本的结束）。而逆序环视无法识别这个被转义的反斜线，如果在这个引号之前有任意多个‘\’，用逆序环视只会把事情弄得更糟。原来的表达式的真正问题在于，如果反斜线是用来转义引号的，在我们第一次处理它时，不会认为它是表示转义的反斜线。所以，我们得用别的办法来解决。

仔细想想我们想要匹配的位于开始分隔符和结束分隔符之间的文本，我们知道，其中可以包括转义的字符（‘\.’），也可以包括非引号的任何字符‘[^‘]’。于是我们得到‘“(\.|\. [^‘]) *”’。不错，现在这个问题解决了。不幸的是，这个表达式还有问题。不期望的匹配仍然会发生，比如对这个文本，它应该是无法匹配的，因为其中没有结束分隔符。

“You need a 2\x3\ Photo.

为什么能匹配呢？回忆一下“匹配优先和忽略优先都期望获得匹配”（F167）。即使这个表达式一开始匹配到了引号之后的文本，如果找不到结束的引号，它就会回溯，到达

‘...2x\3\“-...’	‘(\. \. [^‘])’
-----------------	----------------

从这里开始，‘[^‘]’匹配到反斜线，之后的那个引号被认为是一个结束的引号。

这个例子给我们的重要启示是：

如果回溯会导致不期望，与多选结构有关的匹配结果，问题很可能在于，任何成功的匹配都不过是多选分支的排列顺序造成的偶然结果。

实际上，如果我们把这个正则表达式的多选分支反过来排列，它就会错误地匹配任何包含转义双引号的字符串。真正的问题在于，各个多选分支能够匹配的内容发生了重叠。

那么，应该如何解决这个问题呢？就像第186页的那个连续行的例子一样，我们必须确保，这个反斜线不能以其他的方式匹配，也就是说把‘[^‘]’改为‘[^\“]’。这样就能识别双引

号和文本中的“特殊”反斜线，必须根据情况分别处理。结果就是‘“(\.|\. [^\“]) *”’，它工作得很好（尽管这个正则表达式能够正常工作，但对于 NFA 引擎来说，仍然有提升效率的改进，我们会在下一章更详细地看这个例子，F222）。

这个例子告诉我们一条重要的原理：

不应该忘记考虑这样的“特殊”情形：例如针对“糟糕（bad）”的数据，正则表达式不应该能够匹配。

我们的修改是正确的，但是有意思的是，如果有占有优先量词（F142）或者是固化分组（F139），这个正则表达式可以重新写作‘“(\.|\. [^\“]) *+”’和‘“(?>(\.|\. [^\“]) *)”’。这两个正则表达式禁止引擎回溯到可能出问题的地方，所以它们都可以满足要求。

理解占有优先量词和固化分组解决此问题的原理非常有价值，但是我仍然要继续之前的修正，因为对读者来说它更具描述性（更直观）。其实在这个问题上，我也愿意使用占有优

先量词和固化分组——不是为了解决之前的问题，而是为了效率，因为这样报告匹配失败的速度更快。

了解数据，做出假设

Knowing Your Data and Making Assumptions

现在是时候强调我曾经数次提到过的关于构建和使用正则表达式的一般规则了。知道正则表达式会在什么情况中应用，关于目标数据又有什么样的假设，这非常重要。即使简单如「a」这样的数据也假设目标数据使用的是作者预期的字符编码（F105）。这都是一些很基本的常识，所以我一直没有过分细致地介绍。

但是，许多对某个人来说明显的常识，可能对其他人来说并不明显。例如，前一节的解决办法假设转义的换行符不会被匹配，或者会被应用于点号通配模式（F111）。如果我们真的想要保证点号可以匹配换行符，同时流派也支持，我们应该使用「(?s:.)」。

前一节中我们还假设了正则表达式将应用的数据类型，它不能处理表示其他用途的双引号。如果用这个正则表达式来处理任何程序的源代码，就可能出错，因为注释中可能包括双引号。

对数据做出假设，对正则表达式的应用方式做出假设，都无可厚非。问题在于，假设如果

存在，通常会过分乐观，也会低估了作者的意图和正则表达式最终应用间的差异。记录下这些假设会有帮助。

去除文本首尾的空白字符

Stripping Leading and Trailing Whitespace

去除文本首尾的空白字符并不难做到，却是经常要完成的任务。总的来说最好的办法是使用下面两个替换：

```
s/^\s+//;
```

```
s/\s+$//;
```

为了增加效率，我们用「+」而不是「*」，因为如果事实上没有需要删除的空白字符，就不用做替换。

出于某些考虑，人们似乎更希望用一个正则表达式来解决整个问题，所以我会提供一些方法供比较。我不推荐这些办法，但对理解这些正则表达式的工作原理及其问题所在，非常有意义。

```
s/\s*(.*?)\s*$/\1/s
```

这个正则表达式曾被用作降解忽略优先量词的绝佳例子，但现在不是了，因为人们认识到它比普通的办法慢得多（在 Perl 中要慢5倍）。之所以效率这么低，是因为忽略优先约束的点号每次应用时都要检查「\s*\$」。这需要大量的回溯。

```
s/^\s*((?!\s)*\S?)\s*$/s
```

这个表达式看起来比上一个要复杂，不过它的匹配倒是很容易理解，而且所花的时间也只是普通方法的2倍。在「^\s*」匹配了文本开头的空格之后，「.*」马上匹配到文本的末尾。后面的「\S」强迫它回溯直到找到一个非空的字符，把剩下的空白字符留给最后的「\s*\$」，捕获括号之外的。

问号在这里是必须的，因为如果一行数据只包含空白字符的行，必须出现问号，表达式才能正常工作。如果没有问号，可能会无法匹配，错过这种只有空白字符的行。

```
s/^\s+|\s+$/g
```

这是最容易想到的正则表达式，但它不正确（其实这三个正则表达式都不正确），这种顶极的（top-leveled）多选分支排列严重影响本来可能使用的优化措施（参见下一章）。

/g 这个修饰符是必须的，它容许每个多选分支匹配，去掉开始和结束的空格。看起来，用/g 是多此一举，因为我们知道我们只希望去掉最多两部分空白字符，每部分对应单独的子表达式。这个正则表达式所用的时间是简单办法的4倍。

测试时我提到了相对速度，但是实际的相对速度取决于所用的软件和数据。例如，如果目标文本非常非常长，而且在首尾只有很少的空格，中间的那个表达式甚至会比简单的方法更快。不过，我自己在程序中仍然使用下面两种形式的正则表达式：

```
s/^\s+//;
s/\s+$/;
```

因为它几乎总是最快的，而且显然最容易理解。

HTML 相关范例

HTML-Related Examples

在第2章，我们曾讨论过把纯文本转换为 HTML 的例子（F67），其中要使用正则表达式从文本中提取 E-mail 地址和 http URL。本节来看一些与 HTML 相关的其他处理。

匹配 HTML Tag

Matching an HTML Tag

最常见的办法就是用「<[>]+>」来匹配 HTML 标签。它通常都能工作，例如下面这段用来去除标签的 Perl 语句：

```
$html =~ s/<[>]+>/g;
```

如果 tag 中含有「>」，它就不能正常匹配了，而这样的 tag 明明是合乎 HTML 规范的：<input name=dir value="">>。虽然这种情况很少见，也不为大家推荐，但 HTML 语言确实容许在引号内的 tag 属性中出现非转义的「<」和「>」。这样，简单的「<[>]+>」就无法匹配了，得

想个聪明点的办法。

‘<...>’中能够出现引用文本和非引用形式的“其他文本（other stuff）”，其中包括除了‘>’和引号之外的任意字符。HTML 的引文可以用单引号，也可以用双引号。但不容许转义嵌套的引号，所以我们可以直接用「`"[^"]*"`」和「`'[^']*'`」来匹配。

把这些和“其他文本”表达式「`^[^>]`」合起来，我们得到：

```
<("[^"]*"|'[^']*'|^[^>])*>
```

这个表达式可能有点难看懂，那么加上注释，按宽松格式来看：

```
<                                # 开始尖括号 "<"

(                                # 任意数量的 ...

    "[^"]*"                    # 双引号字符串
    |                          # 或者是...
    "'[^']*'"                  # 单引号字符串
    |                          # 或者是...
    "[^>]"                    # "其他文本"

)*                               #

>                                # 结束尖括号 ">"
```

这个表达式相当漂亮，它会把每个引用部分单作为一个单元，而且清楚地说明了在匹配的什么位置容许出现什么字符。这个表达式的各部分不会匹配重复的字符，因此不存在模糊性，也就不需要担心发生前面例子中出现的，“不小心冒出来（sneaking in）”非期望匹配。

不知你是否注意到了，最开始的两个多选分支的引号中使用了「*」，而不是「+」。引用字符串可能为空（例如‘`alt=""`’），所以要用「*」来处理这种情况。但不要在第三个多选分支中用「*」取代「+」，因为「`^[^>]`」只接受括号外的「*」的限定。给它添加一个加号得到「`(^[^>]+)*`」，可能导致非常奇怪的结果，我不期望读者现在就能理解，下一章会详细讲解它。

在使用 NFA 引擎时，我们还需要考虑关于效率的问题：既然没有用到括号匹配的文本，我们可以把它们改为非捕获型括号（F137）。因为多选分支之间不存在重复，如果最后的「>」无法匹配，那么回头来尝试其他的多选分支也是徒劳的。如果一个多选分支能够在某个位置匹配，那么其他多选分支肯定无法在这里匹配。所以，不保存状态也无所谓，这样做还可以更快地导致失败，如果找不到匹配结果的话。我们可以用固化分组「`(?>...)`」而不是非捕获型括号（或者用占有优先的星号限定）。

匹配 HTML Link

Matching an HTML Link

假设我们需要从一份文档中提取 URL 和链接文本，例如下面的文本中标记的内容：

```
...<a href="http://www.oreilly.com">O'Reilly Media</a>...
```

因为<A> tag 的内容可能相当复杂，我会分两步实现这个任务。第一个是提取<A> tag 内部的内容，也就是链接文本，然后从<A> tag 中提取 URL 地址。

实现第一步有个简单办法，就是在点号通配模式下应用不区分大小写的「<a\b([>]+)>(.*?)」，这里使用了忽略优先量词。它会把<A>的内容放入\$1，把链接文本放入\$2。当然，像之前一样，我不应该用「[>]+」，而应该使用前几节中的表达式。不过在本节，我会继续使用这个简单的形式，因为这样正则表达式更短，也更容易讲解。

<A>的内容存入字符串之后，就可以用独立的正则表达式来检查它们。其中，URL 是 href= value 属性的值。之前已经说过，HTML 容许等号的任意一侧出现空白字符，值可以

以引用形式出现，也可以以非引用形式出现。下面的 Perl 代码用来输出变量\$Html 中的链接。

请注意: while(...)中的正则表达式是简化的形式，请参见正文

```
while ($Html =~ m{a\b([>]+)>(.*?)</a>}ig)
{
    my $Guts = $1; # 把匹配结果存入 ...
    my $Link = $2; # ...对应变量

    if ($Guts =~ m{
        \b HREF          # "href" 属性
        \s* = \s*        # "=" 两端可能出现空白字符
        (?              # 其值为...
            "[^"]*)"      # 双引号字符串
            |              # 或者是...
            '([^']*)'      # 单引号字符串
            |              # 或者是...
```

```

([^">\s]+)      #      "其他文本"

)                  #

}xi)

{

my $Url = $+;      # 获得$1、$2等中实际参与匹配的编号最大的捕获型括号的内容

print "$Url with link text: $Link\n";

}

}

```

有几点需要注意：

1 我们为匹配值的每个多选结构都添加了括号，来捕获确切的文本。

1 因为我使用了某些括号来捕获文本，在不需要捕获的地方我使用非捕获型括号，这样做既清楚又高效。

1 “其他字符”部分排除了空白字符，也排除了引号和‘>’。

1 因为需要捕获整个 `href` 的值，这里使用了「+」来限制“其他文本”多选分支。这是否会和第200页对其他字符应用「+」一样导致“非常奇怪的结果”呢？不会，因为这外面没有直接作用于整个多选结构的量词。其中的细节同样会在下一章讨论。

根据具体文本的不同，最后，URL 可能保存在\$1、\$2或者\$3中。此时其他捕获型括号就为空或是未定义。Perl 提供了特殊变量\$+，代表\$1、\$2之类中编号最靠后的捕获文本。在本例中，这就是我们真正需要的 URL。

Perl 中的\$+很方便，其他语言也提供了其他办法来选择捕获的 URL。常用的程序语言结构就可以检查捕获型括号，找到需要的内容。如果能够支持，命名捕获（F138）最适用于干这个，就像204页的 VB.NET 的例子那样（幸亏.NET 提供了命名捕获，因为它的\$+有问题，F424）。

检查 HTTP URL

Examining an HTTP URL

现在我们得到了 URL 地址，来看看它是否是 HTTP URL，如果是，就把它分解为主机名（hostname）和路径（path）两部分。因为已经有了 URL，任务就比从随机文本中识别 URL 要简单许多，识别的程序要难许多，这将在后文介绍。

所以，如果拿到一个 URL，我们需要能够将它拆分为各个部分。主机名是「`http://`」之后和第一个反斜线（如果有的话）之前的内容，而路径就是除此之外的内容：「`^http://([^\s]+)(/.*)?$`」。

URL 中有可能包含端口号，它位于主机名和路径之间，以一个冒号开头：「`^http://([^\:]+)(:(\d+))?(/*.*)?$`」。

下面是一个分解 URL 的 Perl 代码：

```
if ($url =~ m{^http://([^\:]+)(:(\d+))?(/*.*)?$}i)
{
    my $host = $1;

    my $port = $3 || 80;    # 如果存在，就使用$3；否则默认为80

    my $path = $4 || "/"; # 如果存在，就使用$4；否则默认为"/"

    print "Host: $host\n";

    print "Port: $port\n";

    print "Path: $path\n";

} else {

    print "Not an HTTP URL\n";

}
```

验证主机名

Validating a Hostname

在上面的例子中，我们用「`^\:]+`」来匹配主机名。不过，在第2章中（F76）我们使用的是更复杂的「`[-a-z]+(\.[-a-z]+)*\.(com|edu|...|info)`」。做同样的事情，复杂程度为什么会有这么大的差别？

而且，虽然二者都用来“匹配主机名”，方法却大不相同。从已知文本（例如，从现成的 URL 中）中提取一些信息是一回事，从随机文本中准确提取同样信息是另一回事。

而且，在上例中我们假设，「`http://`」之后就是主机名，所以用「`^\:]+`」来匹配就是理所当然的。但是在第2章的例子中，我们使用正则表达式从随机文本中寻找主机名，所以它必须更加复杂。

现在从另外一个角度来看主机名的匹配，我们可以用正则表达式来验证主机名。也就是说，我们需要知道，一串字符是否是形式规范、语意正确的主机名。按规定，主机名由点号分

VB.NET 中的 link 检查程序

下面的程序会列出 *Htm1* 变量中的链接：


```
Imports System.Text.RegularExpressions
' 设置循环中将会遇到的正则表达式
Dim A_RRegex as Regex = New Regex(
    "<a\b(?:<guts>[^\>]+)>(?:<Link>.*?)</a>", _
    RegexOptions.IgnoreCase)
Dim GutsRegex as Regex = New Regex( _
    "\b HREF                (?# 'href' 属性) " & _
    "\s* = \s*"              (?# '=' 可能存在空白字符) " & _
    "(?:                    (?# 其值为...) " & _
    "\"\"(?:<url>[^\"]*)\""    (?# 双引号字符串) " & _
    "|                      (?# 或者是...) " & _
    "'(?:<url>[^\']*)"         (?# 单引号字符串) " & _
    "|                      (?# 或者是...) " & _
    "(?:<url>[^\"]*>\s+)"      (?# '其他文本') " & _
    ")"                      (?# ) ", _
    RegexOptions.IgnoreCase Or RegexOptions.IgnorePatternWhitespace)
' 现在检查 'Html' 变量 ...
Dim CheckA as Match = A_RRegex.Match(Html)
' For each match within ...
While CheckA.Success
    ' 已匹配 <a> tag, 现在检查 URL
    Dim UrlCheck as Match = _
    GutsRegex.Match(CheckA.Groups("guts").Value)
    If UrlCheck.Success
        ' 已经匹配完毕, 得到 URL/link
        Console.WriteLine("Url " & UrlCheck.Groups("url").Value & _
            " WITH LINK " & CheckA.Groups("Link").Value)
    End If
    CheckA = CheckA.NextMatch
End While
```

需要注意的几点：

- 在 VB.NET 中使用正则表达式，需要首先执行对应的 Imports 语句，告诉编译器应当导入的库文件。
- 程序中使用了「(?:...)」风格的注释，因为 VB.NET 中加入换行符很不方便，所以普通的「#」注释会延伸到下一个换行符或者字符串的结尾（第一种情况即意味着正则表达式剩下的所有内容都作为注释）。为了使用正常的「#...」注释，请在每一行的结尾添加 &chr(10) (F420)。
- 表达式中的每个双引号都需要以「\"」表示 (F103)。
- 两个表达式都用到了命名捕获，Groups("url")比 Groups(1)和 Groups(2)之类更为清晰。

隔的部分组成，每个部分可以包括 ASCII 字符、数字和连字符，但是不能以连字符作为开头和结尾。所以，我们可以在不区分大小写的模式下使用这个正则表达式：「[a-z0-9][a-z0-9][a-z0-9]*[a-z0-9]」。结尾的后缀部分（‘com’、‘edu’、‘uk’等）只有有限多个可能，这

在第2章的例子中提到过。结合起来，下面的正则表达式就能够匹配一个语意正确的主机名：

```
^
(?:          # 进行不区分大小写的匹配

# 零个或多个据点分隔的部分

(?: [a-z0-9]\. | [a-z0-9][a-z0-9]*[a-z0-9]\. )+

# 然后是结尾的后缀部分...

(?: com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z] )

$
```

因为存在长度的限制，能够由这个正则表达式匹配的可能并不是合法的主机名：每个部分不能超过63个字符。也就是说，「`[a-z0-9]*`」应该改为「`[a-z0-9]{0,61}`」。

还需要做最后的改动。按规定，只包括后缀的主机名同样是语意正确的。但实践证明，这些“主机名”不存在，但是对于两个字母的后缀来说情况可不是如此。例如，安哥拉的域名‘ai’就有一个 Web 服务器 *http://ai*。我见过其他这样的链接：cc、co、dk、mm、ph、tj、tv 和 tw。

如果希望匹配这些特殊情况，应该把中间的「`(?:...)+`」改为「`(?:...)*`」：

```
^
(?:          # 进行不区分大小写的匹配

# 零个或多个据点分隔的部分

(?: [a-z0-9]\. | [a-z0-9][a-z0-9]{0,61}[a-z0-9]\. )*

# 然后是结尾的后缀部分...

(?: com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z] )

$
```

现在它可以用来验证包含主机名的字符串了。因为这是我们想出的与主机名相关的三个正则表达式中最复杂的，你也许会想，不要这些锚点，可能比之前那个从随机文本中提取主机名的表达式更好。但情况并非如此。这个正则表达式能匹配任意双字母单词，正因为如此，第2章中不那么精妙的正则表达式的实际效果更好。但是在下一节我们会看到，某些情况下它仍然不够完善。

在真实世界中提取 URL

Plucking Out a URL in the Real World

供职于 Yahoo! Finance 的工作时，我曾写过处理收录的财经新闻和数据的程序。新闻通常是以纯文本格式提供的，我的程序将其转化为 HTML 格式以便于显示（如果你在过去10年中曾经在 *<http://finance.yahoo.com>* 浏览过财经新闻，没准看过我处理过的新闻）。

因为接受的数据的“格式”（其实是无格式）很杂乱，从纯文本中识别（recognize）出 hostname 和 URL 又比验证（validate）它们困难得多，这任务就很不轻松。前面的内容并没有体现这一点，在本节，你会看到我在 Yahoo!用来解决这个问题的程序。

这个程序从文本中提取几种类型的 URL——mailto、http、https 和 ftp。如果我们在文本中找到‘http://’，就知道这肯定是一个 URL 的开头，所以我们可以直接用「`http://[-\w]+(\.\w[-\w]*)+`」来匹配主机名。我们知道，要处理的文本肯定是 ASCII 编码的英文字母，所以完全可以用「`\w`」来取代「`a-z0-9`」。「`\w`」同样可以匹配下画线，在某些系统中，它还可以匹配所有的 Unicode 字符，但是我们知道，这个程序在运行时不会遇到这些问题。

不过，URL 通常不是以 http://或者 mailto:开头的，例如：

...visit us at www.oreilly.com or mail to orders@oreilly.com...

在这种情况下，我们需要加倍小心。我在 Yahoo!使用的正则表达式与前面那节的非常相似，只是有一点点不同：

```
(?:[a-z0-9](?:[-a-z0-9]*[a-z0-9])?\.\s)+ # 子域名 s
```

```
# .com 之类的后缀. 要求小写
```

```
(?:-i: com\b
```

```
| edu\b
```

```
| biz\b
```

```
| org\b
```

```
| gov\b
```

```
| in(?:t|fo)\b # .int 或者.info
```

```
| mil\b
```

```
| net\b
```

```
| name\b
```

```
| museum\b

| coop\b

| aero\b

|[a-z][a-z]\b    # 双字母国家代码

)
```

在这个正则表达式中，我们用「(?i...)」和「(?-i...)」用来规定正则表达式的某个部分是否区分大小写（F135）。我们希望匹配‘www.OReilly.com’，但不是‘NT.TO’这样的股票

代码（NT.TO 是北电网络在多伦多证券交易市场的代号，因为要处理的是财经新闻和数据，这样的股票代码很多）。按规定，URL 的结尾部分（例如‘.com’）可能是大写的，但我不准备处理这些情况。因为我需要保持平衡——匹配期望的文本（尽可能多的 URL），忽略不期望的文本（股票代码）。我希望「(?-i...)」只包括国家代码，但是在现实中，我们没有遇到大写的 URL 地址，所以不必这么做。

下面是从纯文本中查找 URL 的框架，我们可以在其中添加匹配主机名的子表达式：

```
\b

# 匹配开头部分 （proto://hostname，或直接是 hostname）

(

# ftp://、http:// 或 https:// 开头部分

(ftp|https?)/[-\w]+(\.\w[-\w]*)+

|

# 或者用更准确的子表达式找到 hostname

full-hostname-regex

)

# 可能出现端口号

(:\d+)?

# 下面部分可能出现，以/开头

(
```

/ path-part

)?

我还没有谈论过正则表达式的 **path**（路径）部分，它接在主机名后面（例如 <http://www.oreilly.com/catalog/regex/> 中的划线部分）。**path** 是最难正确匹配的文本，因为它需要一些猜测才能做得很漂亮。我们在第2章说过，通常出现在 URL 之后的文本也能被作为 URL 的一部分。例如：

Read his comments at http://www.oreilly.com/ask_tim/index.html. He...

我们观察之后就会发现，在‘[index.html](#)’之后的句号是一个标点，不应该作为 URL 的一部分，但是‘[index.html](#)’中的点号却是 URL 的一部分。

肉眼很容易分辨这两种情况，但程序做起来却很难，所以必须想些聪明的办法来尽可能好地解决问题。第2章的例子使用逆序环视来确保 URL 不会以句末的句号结尾。我在 **Yahoo! Finance** 写程序时还没有逆序环视，所以我用的办法要复杂的多，不过效果是一样的。代码在下一页。

示例5-1：从财经新闻中提取 URL

```
\b

# 匹配开头部分（proto://hostname，或直接是 hostname）

(

# ftp://、http://或 https:// 开头部分

(ftp|https?):/[^\w]+(\.[^\w-]*)+

|

# 或者用更准确的子表达式找到 hostname

(?:[a-z0-9] (?:[a-z0-9]*[a-z0-9])? \. )+          # sub domains

# .com 之类的后缀. 要求小写

(?:-i: com\b

| edu\b

| biz\b

| gov\b
```

```

| in(?:t|fo)\b      # .int 或者.info

| mil\b

| net\b

| org\b

| [a-z][a-z]\b      # 双字母国家代码

)

)

# 可能出现端口号

(: \d+ )?

# 剩下的部分可能出现，以/开头 ...

(

/

# 虽然很复杂，但确实管用

[^\.,?;"'<>()\[\]{} \x7F-\xFF]*

(?:

[^\.,?]+ [^\.,?;"'<>()\[\]{} \x7F-\xFF]+

)*

)?

```

这里用到的办法与第2章第75页用到的办法有很多不同，比较起来也很有意思。下一页里使用此表达式的 **Java** 程序详细介绍了它的构造。

在实际生活中，我怀疑自己是否会写这样繁杂的正则表达式，但是作为取代，我会建立一个正则表达式“库（**library**）”，需要时取用。这方面一个简单的例子就是第76页的 **\$HostnameRegex**，以及下面的补充内容。

扩展的例子

Extended Examples

下面的几个例子讲解了一些关于正则表达式的重要诀窍。讨论会稍微多一些，关于解决办法和错误思路的着墨也会更多一些，最终会给出正确答案。

在 Java 中通过变量构建正则表达式

```
String SubDomain = "(?i:[a-z0-9][a-z0-9]([-a-z0-9]*[a-z0-9]))";
String TopDomains = "(?x-i:com\\b          \\n" +
"    |edu\\b          \\n" +
"    |biz\\b          \\n" +
"    |in(?:t|fo)\\b    \\n" +
"    |mil\\b          \\n" +
"    |net\\b          \\n" +
"    |org\\b          \\n" +
"    |[a-z][a-z]\\b    \\n" + // country codes
")
\\n";
String Hostname = "(?:" + SubDomain + "\\.)+" + TopDomains;
String NOT_IN = ";'\"<>()\\[\\]\\{\\}\\s\\x7F-\\xFF";
String NOT_END = "!.,?";
String ANYWHERE = "[" + NOT_IN + NOT_END + "]";
String EMBEDDED = "[" + NOT_END + "]";
String UriPath = "/" + ANYWHERE +
"*(\"" + EMBEDDED + "\" + ANYWHERE + \")*";
String Url =
"(?x:
"    \\b          \\n" +
"    ##  匹配 hostname          \\n" +
"    (
"        (?: ftp | http s? ): // [-\\w]+(\\.\\w[-\\w]*)+ \\n" +
"        |
"        " + Hostname + "          \\n" +
"    )          \\n" +
"    #  可能出现端口号          \\n" +
"    (?: :\\d+ )?          \\n" +
"    #  下面的部分可能出现，以\\开头          \\n" +
"    (?: " + UriPath + ")?          \\n" +
"    )";
// 现在把正则表达式编译为正则对象
Pattern UriRegex = Pattern.compile(Url);
// 现在准备在文本中应用，寻找 url...
.....
```

保持数据的协调性

Keeping in Sync with Your Data

我们来看一个长一点的例子，它有点极端，但很清楚地说明了保持协调的重要性（同时提供了一些保持协调的方法）。

假设，需要处理的数据是一系列连续的5位数美国邮政编码（ZIP Codes），而需要提取的是以44开头的那些编码。下面是一点抽样，我们需要提取的数值用粗体表示：

```
03824531449411615213441829505344272752010217443235
```

最容易想到的「`\d\d\d\d\d`」，它能匹配所有的邮政编码。在 Perl 中可以用 `@zips=m\d\d\d\d\d/g` 来生成以邮政编码为元素的 list（为了让这些例子看起来更整洁，我们假设需要处理的文本在 Perl 的默认目标变量 `$_` 中，见 F79）。如果使用其他语言，也只需要循环调用正则表达式的 `find` 方法。我们关注的是正则表达式本身，而不是语言的实现机制，所以下面继续使用 Perl。

回到「`\d\d\d\d\d`」，下面提到的这一点很快就会体现出其价值；在整个解析过程中，这个正则表达式任何时候都能够匹配——绝对没有传动装置的驱动和重试（我假设所有的数据都是规范的，此假设与具体情况密切相关）。

很明显，把「`\d\d\d\d\d`」改为「`44\d\d\d`」来查找以44开头的邮政编码不是个好办法——匹配失败之后，传动装置会驱动前进一个字符，对「`44...`」的匹配不再是从每个邮政编码的第一位开始。「`44\d\d\d`」会错误地匹配「`...5314494116...`」。

当然，我们可以在正则表达式的开头添加「`\A`」，但是这样只能对付一行文本中的第一个邮政编码。我们需要手动保持正则引擎的协调，才能忽略不需要的邮政编码。这里的关键是，要跳过完整的邮政编码，而不是使用传动装置的驱动过程（bump-along）来进行单个字符的移动。

根据期望保持匹配的协调性

下面列举了几种办法用来跳过不需要的邮政编码。把它们添加到正则表达式「`44\d\d\d`」之前，可以获得期望的结果。非捕获型括号用来匹配不期望的邮政编码，这样能够快速地略过它们，找到匹配的邮政编码，在第一个\$1的捕获括号中：

```
「(?:^4)\d\d\d\d\d(?:^4)\d\d\d)*...」
```

这种硬办法（brute-force method）主动略过非44开头的邮政编码（当然，用「`[1235-9]`」替代「`^4`」可能更合适，但我之前说过，假设处理的是规范的数据）。注意，我们不能使用「`?:^4[^4]\d\d\d)*`」，因为它不会匹配（也就无法略过）43210这样不期望的邮政编码。

```
「(?:?!44)\d\d\d\d\d)*...」
```

这个办法跳过非44开头的邮政编码。其中的想法与之前并无差别，但用正则表达式写出来就显得大不一样。比较这两段描述和相关的正则表达式就会发现，在这里，期望的邮政编码（以44开头）导致逆序环视（`?!44`）失败，于是略过停止。

```
「(?:\d\d\d\d\d)*?...」
```

这个办法使用忽略优先量词，只有在需要的时候才略过某些文本。我们把它放在真正需要匹配的正则表达式前面，所以如果那个表达式失败，它就会匹配一个邮政编码。忽略优先「(...) * ?」导致这一切的发生。因为存在忽略优先量词，「(?:\d\d\d\d)」甚至都不会尝试匹配，在后面的表达式失败之前。星号确保了，它会重复失败，直到最终找到匹配文本，这样就能只跳过我们希望跳过的文本。

把这个表达式和「(44\d\d\d)」合起来，就得到：

```
@zips=m/(?:\d\d\d\d)*?(44\d\d\d)/g;
```

它能够提取以44开头的邮编，而主动跳过其他的邮编（在“@array = m/.../g”的情况下，Perl 会用每次尝试中找到的匹配文本来填充这个数组，F311）。这个表达式能够重复应用于字符串，因为我们知道每次匹配的“起始匹配位置”都是某个邮政编码的开头位置，也就保证下一次匹配是从一个邮政编码的开始，这正是正则表达式期望的。

不匹配时也应当保证协调性

我们是否能保证，每次正则表达式都在邮政编码字符串的开头位置应用？显然不是！我们手动跳过了不符合要求的邮政编码，可一旦不需要继续匹配，本轮匹配失败之后自然就是驱动过程和重试，这样就会从邮政编码字符串之中的某个位置开始——我们的方法不能处理这种情况。

再来看数据样本：

03824531449411615213441829503544272752010217443235

匹配的代码以粗体标注（第三组不符合要求），主动跳过的代码以下画线标注，通过驱动过程-重试略过的字符也标记出来。在44272匹配之后，目标文本中再也找不到匹配，所以本轮尝试宣告失败。但总的尝试并没有宣告失败。传动机构会进行驱动，从字符串的下一个字符开始应用正则表达式，这样就破坏了协调性。在第四次驱动之后，正则表达式略过10217，错误地匹配44323。

如果在字符串的开头应用，这三个表达式都没有问题，但是传动装置的驱动过程会破坏协调性。如果我们能取消驱动过程，或者保证驱动过程不会添麻烦，问题就解决了。

办法之一是禁止驱动过程，即在前两种办法中的「(44\d\d\d)」之后添加「?」，将其改为匹配优先的可选项。这样，刻意安排的「(?:(!44)\d\d\d\d)*...」或「(?:[!4]\d\d\d\d)\d

[!4]\d\d\d\d)*...」就只会两种情况下停止：发生符合要求的匹配，或者邮政编码字符串结束（这也是此方法不适用于第三个表达式的原因）。这样，如果存在符合要求的邮政编码，「(44\d\d\d)?」就能匹配，而不会强迫回溯。

这个办法仍然不够完善。原因之一是，即便目标字符串中没有符合要求的邮政编码，也会匹配成功，接下来的处理程序会变复杂。不过，其优点在于速度很快，因为不需要回溯，也不需要传动装置进行任何驱动过程。

使用\G 保证协调

更通用的办法是在这三个表达式末尾添加「\G」（F130）。因为每个表达式的每次匹配都以符合要求的邮政编码结尾，下次匹配开始时就不会进行驱动。而如果有驱动过程，开头的「\G」会立刻导致匹配失败，因为在大多数流派中，只有在未发生驱动过程的情况下，它才能成功匹配（但在 Ruby 和其他规定「\G」表示“本次匹配起始位置”的流派中不成立 F131）

所以第二个表达式就变成了：

```
@zips = m/\G(?:?!44)\d\d\d\d)*(44\d\d\d)/g;
```

匹配之后不需要进行任何特殊检查。

本例的意义

我首先承认，这个例子有点极端，不过，它包含了许多保证正则表达式与数据协调性的知识。如果现实生活中需要处理这样的问题，我可能不会完全用正则表达式来解决。我会直接用「\d\d\d\d\d」来提出每个邮政编码，然后检查它是否以‘44’开头。在 Perl 中是这样：

```
@zips = ();      # 确保数组为空

while (m/(\d\d\d\d\d)/g) {

    $zip = $1;

    if (substr($zip, 0, 2) eq "44") {

        push @zips, $zip;

    }

}
```

对「\G」有兴趣的读者请参考132页的补充内容，尽管本书写作时只能举 Perl 的例子。

解析 CSV 文件

Parsing CSV Files

解析 CS（逗号分隔值）文件有点麻烦，因为每个程序都有自己的 CSV 文件格式。首先来看如何解析 Microsoft Excel 生成的 CSV 文件，然后再看其他格式（注3）。幸运的是，Microsoft 的格式是最简单的。以逗号分隔的值要么是“纯粹的”（仅仅包含在括号之前），要么是在双引号之间（这时数据中的双引号以一对双引号表示）。

下面是个例子：

```
Ten Thousand,10000, 2710 ,,"10,000","It's ""10 Grand"" , baby",10K
```

这一行包含七个字段（fields）：

Ten-Thousand

10000

-2710-

空字段

10,000

It's-"10-Grand",-baby

10K

为了从此行解析出各个字段，我们的正则表达式需要能够处理两种格式。非引号格式包含引号和逗号之外的任何字符，可以用「`[^,]+`」匹配。

双引号字段可以包含逗号、空格，以及双引号之外的任何字符。还可以包含连在一起的两个双引号。所以，双引号字段可以由「`"..."`」之间任意数量的「`[^"]|""`」匹配，也就是「`"(?:[^\"]|")*"`」（为效率考虑，我们可以使用固化分组「`(?>...)`」来替代「`(?:...)`」，不过这个话题留到下一章 F259）。

综合起来，「`[^,]+|"(?:[^\"]|")*"`」能够匹配一个字段。这可能有点难看懂，下面我们给出宽松排列（F111）格式：

```
# 引号和逗号之外的文本...
```

```
[^,]+
```

```
# ...或者是...
```

```
|
```

```
# ...双引号字段（其中容许出现连在一起的成对双引号）
```

```
"# 起始双引号
```

```
(?: [^"] | "" )*
```

```
"# 结束双引号
```

现在这个表达式可以实际应用到包含 CSV 文本行的字符串上了，但如果我们希望真正

利用匹配结果，就应该知道具体是哪个多选分支匹配了。如果是双引号字符串，就需要去掉首尾两端的双引号，把其中紧挨着的两个双引号替换为单个双引号。

我能想到的办法有两个。其一是检查匹配结果的第一个字符是否双引号，如果是，则去掉第一个和最后一个字符（双引号），然后把中间的“”替换为“”。这办法够简单，但如果使用捕获型括号会更简单。如果我们给捕获字段的每个子表达式添加捕获型括号，可以在匹配之后检查各个分组的值：

```
# 引号和逗号之外的文本...

([^\,]+)

# ... 或者是...

|

# ...双引号字段（其中容许出现连在一起的成对双引号）

" # 起始双引号

((?: [^"] | "" )*)

" # 结束双引号
```

如果是第一个分组捕获，则不需要进行任何处理，如果是第二个分组，则只需要把“”替换为“”即可。

下面给出 Perl 的程序，稍后（找出某些 bug 之后）给出 Java 和 VB.NET（在第10章给出 PHP 的程序 F480）。下面是 Perl 程序，假设数据位于\$line 中，而且已经去掉了结尾的换行符（换行符不属于最后的字段!）：

```
while ($line =~ m{

# 引号和逗号之外的文本...

([^\,]+)

# ...或者是...

|

# ...双引号字段（其中容许出现连在一起的成对双引号）

" # 起始双引号

( (?: [^"] | "" )*)
```

```
"# 结束双引号

}gx)

{

if (defined $1) {

$field = $1;

} else {

$field = $2;

$field =~ s/"/"/g;

}

print "[$field]";# 输出$field 供调试

现在可以处理$field 了...

}
```

将其应用于测试数据，结果为：

```
[Ten-Thousand][10000][-2710-][10,000][It's-"10-Grand",-baby][10K]
```

看来没问题，但不幸的是它不会输出为空的第四个字段。如果“处理\$field”是将字段的值存入数组，完成后访问数组的第五个元素得到第五个字段（“10,000”）。这显然不对，因为数组的元素与空字段不对应。

想到的第一个办法是把「^,]+」改为「^,]*」，这看来是显而易见的，但它正确吗？

测试一下，下面是结果：

```
[Ten-Thousand][][10000][][-2710-][][10,000][][It's-"10-Grand", ...
```

哇，现在出来了一堆空字段！仔细检查检查，就不会这么吃惊。「(...)»的匹配可以不占用任何字符。如果真的遇到空字段，确实能匹配，那么考虑第一个字段匹配之后的情况呢，此时正则表达式从‘Ten-Thousand_a,10000...’开始应用。如果表达式中没有元素可以匹配逗号（就本例来说），就会发生长度为0的成功匹配。实际上，这样的匹配可能有无穷多次，因为正则引擎可能在同一位置重复这样的匹配，现代的正则引擎会强迫进行驱动过程，所以同一位置不会发生两次长度为0的匹配（F131）。所以每个有效匹配之间还有一个空匹配，在每个引号字段之前会多出一个空匹配（而且数组末尾还会有一个空匹配，只是此处没有列出来）。

分解驱动过程

要解决问题，我们就不能依赖传动机构的驱动过程来越过逗号。所以，我们需要手工来控制。能想到的办法有两个：

1. 手工匹配逗号。如果采取此办法，需要把逗号作为普通字段匹配的一部分，在字符串中“迈步（pace ourselves）”。

2. 确保每次匹配都从字段能够开始的位置开始。字段可以从行首，或者是逗号开始。

可能更好的办法是把两者结合起来。从第一种办法（匹配逗号本身）出发，只需要保证逗号出现在第一个字段之外的所有字段开头。或者，保证逗号出现在最后一个字段之外的所有字段的末尾。可以在表达式前面添加「^|」，或者后面添加「\$|」，用括号控制范围。

在前面添加，就得到：

```
(?:^|,)
```

```
(?:
```

```
# 引号和逗号之外的文本....
```

```
( [^",]* )
```

```
# ...或者是..
```

```
|
```

```
# ... 双引号字段（其中容许出现连在一起的成对双引号）
```

```
" # 起始双引号
```

```
((?: [^"] | "" )*)
```

```
" # 结束双引号
```

```
)
```

看起来它应当没错，但实际的结果却是：

```
[Ten-Thousand][10000][-2710-][][000][[-baby][10K]
```

而我们期望的是：

```
[Ten-Thousand][10000][-2710-][][10,000][It's-"10•Grand",-baby][10K]
```


问题出在哪里呢？似乎是双引号字段没有正确处理，所以问题出在它身上，对吗？不对，问题在前面。或许176页的告诫有所帮助：如果多个多选分支能够在同一位置匹配，必须小心地排列顺序。第一个多选分支「`[""]*`」不需要匹配任何字符就能成功，除非之后的元素强迫，否则第二个多选分支不会获得尝试的机会。而这两个多选分支之后没有任何元素，所以第二个多选分支永远不会得到尝试的机会，这就是问题所在！

哇，现在我们已经找到了问题所在。OK，交换一下多选分支的顺序：

```
(?:^|,)  
  
(?: # 或者是匹配双引号字段（其中容许出现连在一起的成对双引号） ...  
  
" # （起始双引号）  
  
(?: [""] | "" ) * )  
  
" # （起始双引号）  
  
|  
  
# ... 或者是引号和逗号之外的文本 ...  
  
( [^",]* )  
  
)
```

对了！至少对测试数据来说是对了。如果数据变了，还是这样吗？本节的标题是“分解驱动过程”，而最保险的办法就是以完整测试作为基础的思考，故可以用「`\G`」来确保每次匹配从上一次匹配结束的位置开始。考虑到构建和应用正则表达式的过程，这样做应该绝对没问题。如果在表达式开始添加「`\G`」，就会禁止引擎的驱动过程。我们希望这样修改不会出问

题，但是结果并非如此。之前输出

```
[Ten-Thousand][10000][-2710-][][000][[-baby][10K]
```

的正则表达式添加`\G`之后，得到

```
[Ten-Thousand][10000][-2710-][[]]
```

如果起初没看明白，这样看会更明显。

CSV Processing in Java

这里有一个使用 Sun 的 `java.util.regex` 解析 CSV 的例子。这段程序着眼于简洁的、更有效的版本——第8章（F401）将会介绍。

```
import java.util.regex.*;
```

```
.
```

```

.
.
String regex = // 把双引号字段存入 group(1)、非引号字段存入 group(2)
"\G(?:\,|)                                \n"+
"(?:                                         \n"+
"    # 要么是双引号字段...                    \n"+
"    \"          # 字段起始双引号            \n"+
"    ( (?: [^"]++ | \"\")*+ )                \n"+
"    \"          # 字段结束双引号            \n"+
"|# ... 要么是 ...                          \n"+
"    # 非引号非逗号文本 ...                  \n"+
"    ([^\",,]*)                             \n"+
" )                                         \n";
// 创建使用上面正则表达式的 matcher，暂时不指定需要应用的文本
Matcher mMain = Pattern.compile( regex, Pattern.COMMENTS).matcher("");
// 为「"""」创建一个 matcher，暂时不指定需要应用的文本
Matcher mQuote = Pattern.compile("\"\"").matcher("");
.
.
.
// 上面都是准备工作，下面的代码逐行处理文本
mMain.reset( line); //下面处理 line 中的 CSV 文本
while ( mMain.find())
{
String field;
if ( mMain.start(2) >= 0)
field = mMain.group(2);      // 非引号字段，直接使用
else
// 引号字段，替换其中的成对双引号
field = mQuote.reset(mMain.group(1)).replaceAll("\"");
// 处理字段...
System.out.println("Field [" + field + "]");
}

```

另一个办法

本节的开头提到有两种办法正确匹配各个字段。之二是确保匹配只能在容许出现字段的地方开始。从表面上看，这类似于添加「^」，只是使用了逆序环视「(?<=^)」。

不幸的是，按照第3章（F133）的解释，即使支持逆序环视，也不见得会支持变长的逆序环视，所以此方法可能无法使用。如果问题在于长度可变，我们可以把「(?<=^)」替换为「(?:^|(?<=,))」，但是相比第一种办法，它太麻烦了。而且，它仍然依赖传动装置的驱动过程来越过逗号，如果别的地方出了什么差错，它会容许在「..."10, 000"」处的匹配。总的来说就是，不如第一种办法保险。

不过我们可以略施小计——要求匹配在逗号之前（或者是一行结束之前）结束。在表达

式结尾添加「(?:= \$|,)」可以确保它不会进行错误的匹配。实际生活中, 我会这样做吗? 直率地说我觉得第一种方法很合用, 所以遇到这种情况我可能不会采取第二种办法, 不过如果需要, 这技巧却是很有用的。

进一步提高效率

尽管在下一章之前都不会谈论效率, 但对于支持固化分组 (F139) 的系统, 我还是愿意在这里给出提高效率的修改: 把匹配双引号字段的子表达式从「(?:[^\"]|")*」改为「(?:> [^"]+|")*」。下一页用 VB.NET 的例子做了说明。

如果像 Sun 的 Java regex package 那样支持占有优先量词 (F142), 也可以使用占有优先量词。Java CSV 程序的补充内容说明了这一点。

这些修改背后的道理会在下一章讲解, 最终我们会在271页给出效率最高的办法。

其他 CSV 格式

Micorsoft 的 CSV 格式很流行, 因为它是 Microsoft 的 CSV 格式, 但其他程序可能有不同格式, 我见过的情况还有:

- 1 使用任意字符, 例如';'或者制表符作为分隔。(不过这样名字还能叫“逗号分隔值”吗?)
- 1 容许分隔符之后出现空格, 但不把它们作为值的一部分。
- 1 用反斜线转义引号 (例如用\"而不是\"类表示值内部的引号)。通常这意味着反斜线可以在任何字符前出现 (并忽略)。

这些变化都很容易处理。第一种情况只需要把逗号替换为对应的分隔符, 第二种只需要在第一个分隔符之后添加「\s*」, 例如以「(?:^\s*)」开头。

第三种情况, 我们可以用之前的办法 (F198), 把「[^\"]+|\"」替换为「[^\"]+|\\。」。当然, 我们必须把后面的 s/"/g 改为更通用的 s/\\(.)/\$1/g, 或者对应语言中的代码。

VB.NET 的 CSV 处理

```
Imports System.Text.RegularExpressions
.....
Dim FieldRegex as Regex = New Regex( _
    "(?:^|,)"                                     " & _
    "(?:"                                           " & _
        "      (?# 要么是双引号字段 ...)"         " & _
        "      \"\"      (?# 字段起始双引号)"      " & _
        "      (      (?> [^\"]+|\"\"\"\" )*)"      " & _
        "      \"\"      (?# 字段结束双引号)"      " & _
        " (?# ... or ...)"                         " & _
```

```

" |                                     " & _
"    (?#    ... 非引号非逗号文本  ...)    " & _
"    ([^"'],)*                            " & _
" )", RegexOptions.IgnorePatternWhitespace)
Dim QuotesRegex as Regex = New Regex(" "" "" ")    '# 双引号字符串
.
.
.
.

Dim FieldMatch as Match = FieldRegex.Match(Line)
While FieldMatch.Success
Dim Field as String
If FieldMatch.Groups(1).Success
Field = QuotesRegex.Replace(FieldMatch.Groups(1).Value, "")
Else
Field = FieldMatch.Groups(2).Value
End If
Console.WriteLine("[ " & Field & " ]")
' 现在可以处理'Field'
FieldMatch = FieldMatch.NextMatch

End While

```