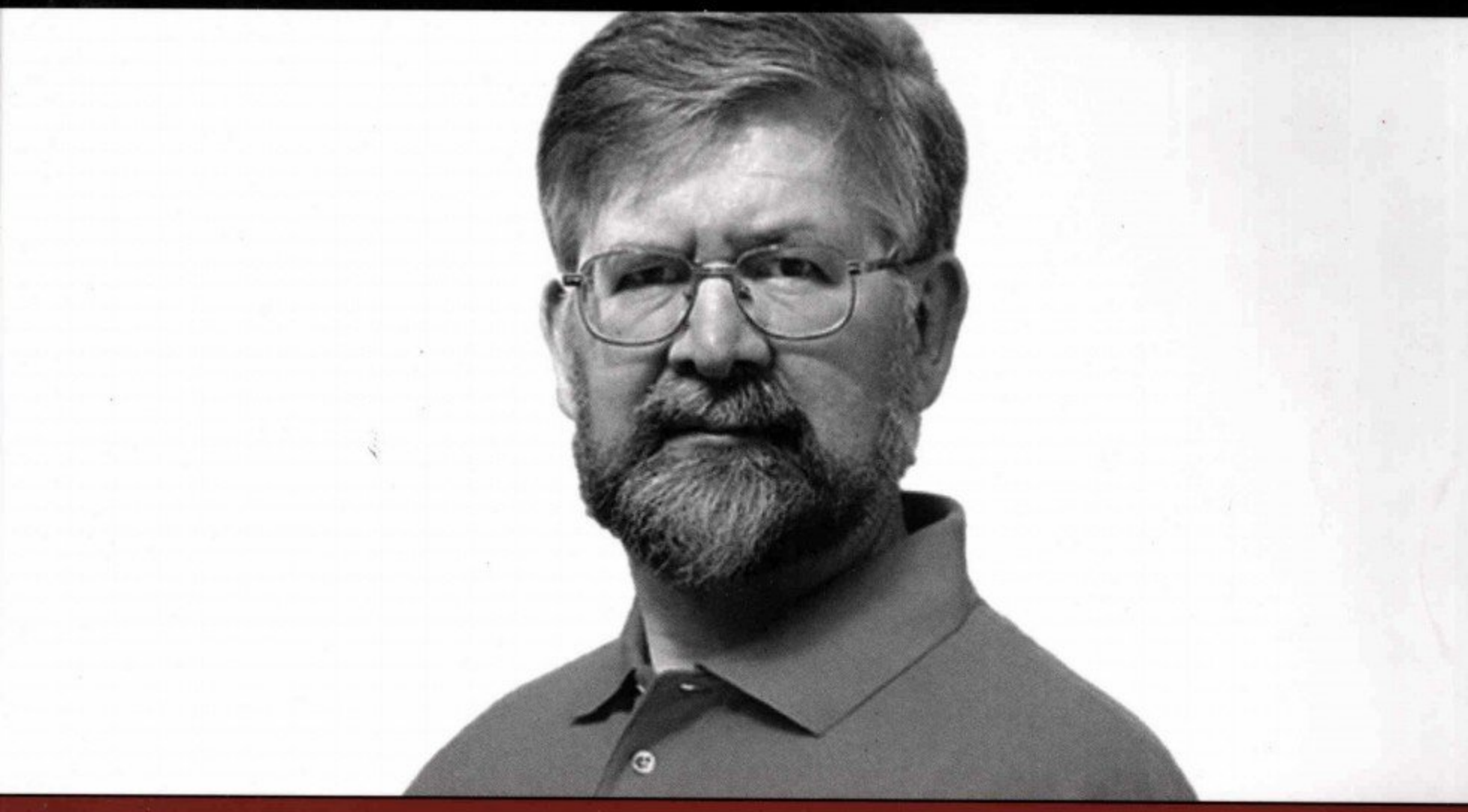
PROGRAMMER TO PROGRAMMER™



Beginning Regular Expressions

正则表达式入门经典

(美) Andrew Watt 著
李松峰 李丽 译

WILEY
wrox™
清华大学出版社

正则表达式入门经典

Andrew Watt 著

李松峰 李丽 译

清华大学出版社

北 京

Andrew Watt
Beginning Regular Expressions
EISBN：0-7645-7489-2

北京市版权局著作权合同登记号 图字：01-2007-4046

图书在版编目(CIP)数据

正则表达式入门经典/(美)瓦特(Watt, A.) 著；李松峰，李丽 译. —北京：清华大学出版社，2008.10
书名原文：Beginning Regular Expressions
ISBN 978-7-302-18382-2

Ⅰ. 正… Ⅱ. ①瓦… ②李… ③李… Ⅲ. 正则表达式 Ⅳ. TP301.2

中国版本图书馆 CIP 数据核字(2008)第 123106 号

责任编辑：王　军　郑雪梅
装帧设计：孔祥丰
责任校对：胡雁翎
责任印制：何　芊
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：北京密云胶印厂
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185×260　**印　张**：41　**字　数**：998 千字
版　　次：2008 年 10 月第 1 版　　**印　　次**：2008 年 10 月第1 次印刷
印　　数：1～4000
定　　价：79.99 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177 转 3103　　产品编号：025893—01

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

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

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

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

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

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

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

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

前　言

大量的商业和其他数据都是以文本形式保存的。因此，对文本进行搜索和操作也就成为了所有开发人员要从事的最重要的活动。正则表达式，无论对用户还是对开发人员而言都是一种最强大的工具。通过使用正则表达式，可以使得查找和操作文本更加有效，而且效率也更高。

事实上，许多开发人员对正则表达式都有一种谈虎色变的感觉，这种感觉从某种程度上来说也很正常。由于正则表达式非常简洁，所以常常会令人产生神秘感。而改变一个字符就可以从根本上改变正则表达式的含义。这些困难会使开发人员常常感到无法完全掌控自己的正则表达式代码。更糟糕的是，当他们要修改别人编写的正则表达式代码时总会感到无所适从。导致这种糟糕局面的一个原因是许多开发人员都没有对自己编写的正则表达式代码给予充分的说明。然而，如果把正则表达式分解成更小的组件，并认真思考你希望它们帮你做什么，那么正则表达式就会成为极其有用的工具(事实上应该是一种必备工具)。

本书的目的就是要帮助你克服学习正则表达式的障碍，并让你在理解正则表达式的能力与不足的基础上有效地发挥它的优势。

谁适合阅读本书

《正则表达式入门经典》适合那些需要操作文本但对正则表达式并不了解的开发人员，也适合那些曾经尝试使用过正则表达式，但却因为某些专家不了解新手需求而制定了不切实际的学习曲线，从而导致无法继续前进的开发人员。

阅读本书的开发人员应该使用 Windows 操作系统。因此，不需要理解如何在 Unix 中使用正则表达式。本书中出现的所有语言和工具都可以在 Windows 中运行，而其中的某些版本也能在其他平台中运行。

本书会以你可能已经知道的知识作为起点，比如当执行命令行文件搜索时使用 * 和 ? 字符。随着相关知识的丰富，你会看到一些贴近实际的例子，而对这些例子稍加修改就能将其用于解决你所遇到的实际问题。

无论你是一名临时的程序员，还就是一个从未使用过正则表达式的人，你都会学习到正则表达式的构成部件、这些部件的含义、用法以及在使用它们时应该知道的缺陷。而书中有效的实例构成了学习创建、理解和使用正则表达式的核心内容。书中大多数章节都包含一些“试一试”部分，用于展示如何让正则表达式工作。而每个“试一试”部分都会带有对应的“工作原理”部分或者其他说明，用以解释正则表达式的工作原理。

本书包含的内容

本书主要介绍了构成正则表达式模式的各个部件，解释了这些部件的含义，并通过实例演示了正则表达式的作用，同时解释了相应的工作原理。通过学习这些实例，你将充分理解如何让正则表达式完成你想做的事，而避免编写不能实现你的意图的正则表达式。

第 1 章对正则表达式进行了概述，并介绍了如何把一个文本操作问题分解为几个构成部分，以便在构建需要匹配目标内容的正则表达式时做出明智的选择，从而避免匹配不想要的文本。

为了解决更复杂的问题，最好从“问题定义”出发，然后逐渐按照人的思维对问题的表达进行不断提炼，最终使其在某种程度上达到构造所需正则表达式的要求。

本书的第二部分用一章的篇幅集中介绍了在 Windows 平台中可用的一些技术。在这一章中，你会看到如何通过各种工具或语言来使用正则表达式(例如，在 Perl 语言中如何实现向前查找，或在 C#中如何创建命名变量等)。

正则表达式对于像 Microsoft Word、OpenOffice.org Writer、Microsoft Excel 和 Microsoft Access 这样的应用程序而言也是很有用的。上述每个应用程序都有专门的一章来介绍正则表达式在其中的应用。

另外，对于像 Windows 系统下鲜为人知的 findstr 实用程序和商业性的 PowerGrep 这样的工具，也都分别有一章来介绍如何通过它们解决跨文件的文本操作问题。

而正则表达式在 MySQL 和 Microsoft SQL Server 数据库中的用法同样也是本书中内容的一部分。

至于几种常用的编程语言，本书也都通过独立的章节对这些语言中有效元字符的用法给予了说明。同时也对如何在正则表达式中使用相应语言的对象或类给出了丰富的例子。这些语言包括：VBScript、JScript、Visual Basic .NET、C#、PHP、Java 和 Perl。

XML 越来越多地被用于存储文本化的数据。W3C 的 XML Schema 定义语言能够使用正则表达式自动验证 XML 文档中的数据。W3C 的 XML Schema 也占据了本书一章的篇幅，其中重点介绍了如何通过 xs:pattern 元素使用正则表达式的内容。

本书是如何组织的

本书的 1~10 章分别介绍了构成正则表达式模式的各个部件、这些部件的作用以及它们在多种文本操作工具和语言中的用法。建议按顺序依次阅读这几章的内容，以便对正则表达式建立起全面系统的理解。

然后，本书的每一章分别专门介绍了一种文本操作工具或者编程语言。这些章中的内容都假设你已经掌握了 1~10 章中的知识。但是你可以根据自己的喜好，针对具体的工具或语言分别加以研究，而不必拘泥于各章的前后顺序。

基于本书的这种组织方式，你可以将 1~10 章中的内容作为掌握正则表达式的必备知识。而对于剩下的那些章节，则可以根据自己使用过的或者必须要在某个特殊项目中使用的技术，来具体选择学以致用的内容。

许多开发人员都曾有过被要求使用一门他们并未完全掌握的语言去编程的经历。而本书针对每种编程语言的章节中都提供了丰富的实例代码，这样你就可以在适当的时候把它们派上用场。

使用本书的要求

本书适用于多种工具和编程语言。1~10 章中的例子也涉及到从 Microsoft Word 和 OpenOffice.org Writer 到 PowerGrep、Java 和 Perl 等多种工具和语言。

本书的读者对象主要是 Windows 用户和开发人员。不过，本书中的很多内容也适用于其他平台的开发人员。

本书中涉及的这些工具和技术，很可能不会全都用到。例如，在正常情况下几乎不可能有人同时使用 JScript、Perl、C#、Java 和 PHP 编程。根据你感兴趣的语言，我们会假定你已经安装了相应的工具。不过，对于可以免费试用或免费下载的软件，我们都会为你提供取得它们以及在 Windows 系统中安装它们的相关信息。

源代码

在练习本书中的示例时，读者可以手动输入所有的代码，也可以使用随书附带的源代码。本书中的所有源代码均可在 http://www.wrox.com 或 http://www.tupwk.com.cn/downpage 站点下载。登录 Web 站点 http://www.wrox.com 后，只需找到本书的标题(可以使用 Search 功能或标题列表)并单击显示本书详细内容的页面上的 Download Code 链接即可获取源代码。

由于很多书籍的名称类似，因此通过 ISBN 查找可能会更容易一些。本书的 ISBN 号是 0-7645-7489-2。

在下载源代码之后，只需使用最喜欢的解压缩工具对其进行解压缩即可。另一个途径是到 Wrox 代码下载主页面 http://www.wrox.com/dynamic/books/download.aspx，这里有本书及其他所有 Wrox 书籍的源代码。

勘误表

我们尽最大努力确保本书在叙述和代码中没有错误。然而，没有人是完美的，错误时有发生。如果读者在本书中发现什么错误，例如拼写错误或编码错误，我们将会非常感谢您能反馈给我们。通过将错误添加到勘误表中，您也许能为其他读者节约数小时的时间，也可以帮助我们提供更高质量的书籍。

请给 wkservice@tup.tsinghua.edu.cn 发电子邮件，如果您的意见是正确的，我们将在本书的后续版本中采用。

要找到本书的勘误表，可以登录 Web 站点 http://www.wrox.com，然后搜索本书的书名或者使用标题列表。然后，在本书的详细内容页面上，单击 Book Errata 链接。在这个页面

上读者可以查看到所有已提交的、由 Wrox 的编辑发布的错误信息。也可以在 http://www.wrox.com/misc-pages/booklist.shtml 页面找到一个完整的标题列表，这个列表包含了每本书的勘误表链接。

http://p2p.wrox.com

如果想参与讨论，可以加入 P2P 论坛，网址是 http://p2p.wrox.com。这些论坛是基于 Web 站点的系统，其作用是让读者发布与 Wrox 的书籍和相关技术有关的消息，并与其他读者和技术用户联络。这些论坛提供订阅功能，当读者感兴趣的主题发布时，论坛会通过电子邮件把这些消息发送给读者。Wrox 的作家、编辑、其他行业专家及和您一样的读者都会出现在这些论坛上。

在 http://p2p.wrox.com 中，读者将找到很多不同的论坛，这些论坛不仅能帮助读者阅读本书，还可以帮助读者开发自己的应用程序。要加入这些论坛，可按如下步骤操作：

(1) 登录 http://p2p.wrox.com 并单击 Register 链接。

(2) 阅读用途条款并单击 Agree。

(3) 填写加入论坛所必需的信息和可选信息并单击 Submit。

(4) 读者将收到一封电子邮件，该邮件告诉读者怎样验证账户并成功加入。

没有加入 P2P 论坛也可以阅读该论坛上的信息，但是如果希望发布自己的消息，则必须加入该论坛。

加入之后，就可以发布新消息或者回复其他用户发布的消息了。可以在任何时间阅读 Web 站点上的消息。如果希望某个论坛能将最新的消息通过电子邮件发送给自己，则可以单击论坛列表中该论坛名称旁边的 Subscribe 图标。

要获得如何使用 P2P 论坛的更多信息，可以阅读 P2P FAQ 列表中的问题及其答复，这些问题与论坛软件的工作原理及很多与 P2P 和 Wrox 相关的常见问题有关。要阅读 FAQ，可以单击任意 P2P 页面上的 FAQ 链接。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

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

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

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

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

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

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

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目　录

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

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

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

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

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

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

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

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

第 1 章

正则表达式概述

文本是很多人在使用计算机时要面对的重要内容。无论是编写文档还是编辑代码，文本几乎无处不在。网页通常就是由大量文本组成的，其中一部分是由超文本标记语言(Hypertext Markup Language，HTML)或可扩展超文本标记语言(Extensible Hypertext Markup Language，XHTML)标记的，其他则是常规性文本。而所有这些文本组合到一起就构成了可以通过正则表达式进行匹配的字符序列。Web 页面中的表单可以接受文本作为输入，这些文本需要根据输入条件进行匹配。商业文档是由文本组成的，对其中特殊文本序列的搜索同样可以通过正则表达式实现。电子邮件内容和开发人员编写的代码是由文本组成的。而正则表达式对很多要处理文本的情况都是适用的。

文本不仅随处可见、数量庞大，而且还必须保证对文本内容及时更新或整合。在创建了大量的文本内容后，随着存取频率的增加，需要一种有效而且高效的手段去查找特别关注的文本，或者修改特定的文本块。

在长度只有一两页的单个文档中查找和替换个别文本块一般都很简单。但是如果要处理的文档很多，而每个文档都长达数百页或者都有数以千计的相关短文档，那么同样的查找和替换就会变成一件可怕的任务，而且极有可能导致人为错误。对于此类任务，就可以用正则表达式来处理，因为正则表达式可以实现多种有用的文本处理形式的自动化。

比如说，对一个 Web 表单中的信用卡号码进行验证以保证结构正确，或保证邮政编码的格式正确。在一份冗长的文档中，想找到一个重要信息源的链接，但却记不清 URL 地址了。你可能希望转换 HTML 代码，以便使其符合可扩展标记语言(eXtensible Markup Language, XML)的语法规则并遵守公司使用 XHTML 代码的制度。你还可能希望对用户在 Windows 应用程序中的输入进行检查，以保证输入满足必需的条件进而得以正确地处理。

在本章中将学习以下内容：

- 什么是正则表达式
- 可以使用正则表达式做什么
- 为什么正则表达式会令人望而却步

对于一种能够操作文本的工具，其可能的用途几乎无法穷举，因为文本的存在实在是太普遍了。而可悲的是，很多计算机用户和开发人员对如何使用正则表达式来操作文本却一知半解，甚至毫无概念。本书就致力于改变这种局面。

1.1 什么是正则表达式

正则表达式是一种匹配文本中的字符序列的字符模式。为了让开发人员创建正则表达式模式，某些字符和字符组合被赋予了特殊的含义和功能，本书将着重介绍这些内容。不过，我们需要首先理解一些更基本的概念。

正则表达式，从最基本的层面来说，可以让计算机用户和开发人员找到想要的文本块，而且通常会以更适合的内容来替换这些文本块。在其他情况下，正则表达式用于测试一个字符序列(如信用卡号码或者社会保险号(Social Security Number，SSN))中是否包含被允许的字符模式。无论是查找现有的字符序列，还是为了存储的有效性而对字符序列进行符合性测试，其中正则表达式的作用都可以归结为一句话——判断一个字符序列是否与一个模式相匹配。

从宽泛的意义上来看，说正则表达式是语言是没有问题的。但是，严格来讲，不存在正则表达式这门语言。与 JavaScript 和 VBScript 这样的脚本语言类似，它们都只能在另一种应用程序或者语言的环境下使用。而正则表达式同样也只能在一种“合适的”编程语言的环境下使用。与脚本语言类似，正则表达式要么是作为诸如 Microsoft Word 或 OpenOffice.org Writer 之类的应用程序的一部分，要么是以命令行实用程序的面目出现，例如 findstr 实用程序。虽然我们可以抽象地讨论正则表达式，但它们只能在其他语言或应用程序中使用。

本章主要介绍一些非常简单的正则表达式的例子。第 2 章会比较深入地讨论一些正则表达式工具。可以从 www.openoffice.org 上面下载最新版的 OpenOffice.org，来试练习本章中的简单例子，也可以简单地通读全章内容，并认真地观察书中的插图。本书建议下载 OpenOffice.org，因为这个程序非常简单易用，通过它能够对正则表达式语法的很多方面进行验证。

试一试：匹配直接量字符

最简单的正则表达式类型是一个字符序列。例如，如果想查找三个字符的序列 car，就可以使用 car 作为正则表达式模式来查找这些字符。

首先，用自然语言来表达这个问题：

匹配一个字符序列；首先匹配字母 c，接着匹配字母 a，再匹配字母 r。

假设在文档 Car.txt 中包含以下文本内容：

```
Carl spilt his carton of orange juice on the carpet of his new car.
If he had taken more care when opening the carton he wouldn't have had this
annoying and disappointing accident.
Some car shampoo would, Carl hoped, make the carpet look as good as new.
```

如图 1-1 中使用简单的正则表达式在 OpenOffice.org Writer 中搜索得到的结果所示，搜索字符序列 car 时产生了很多匹配项。如果想试验一下，可以遵循以下步骤操作：

(1) 在 OpenOffice.org Writer(其 1.1 及更高版本支持正则表达式)中打开 Car.txt。

(2) 使用 Ctrl+F 打开 Find and Replace 对话框。

(3) 选中 Regular expressions 复选框。

(4) 在 Search for 文本框中输入 car。

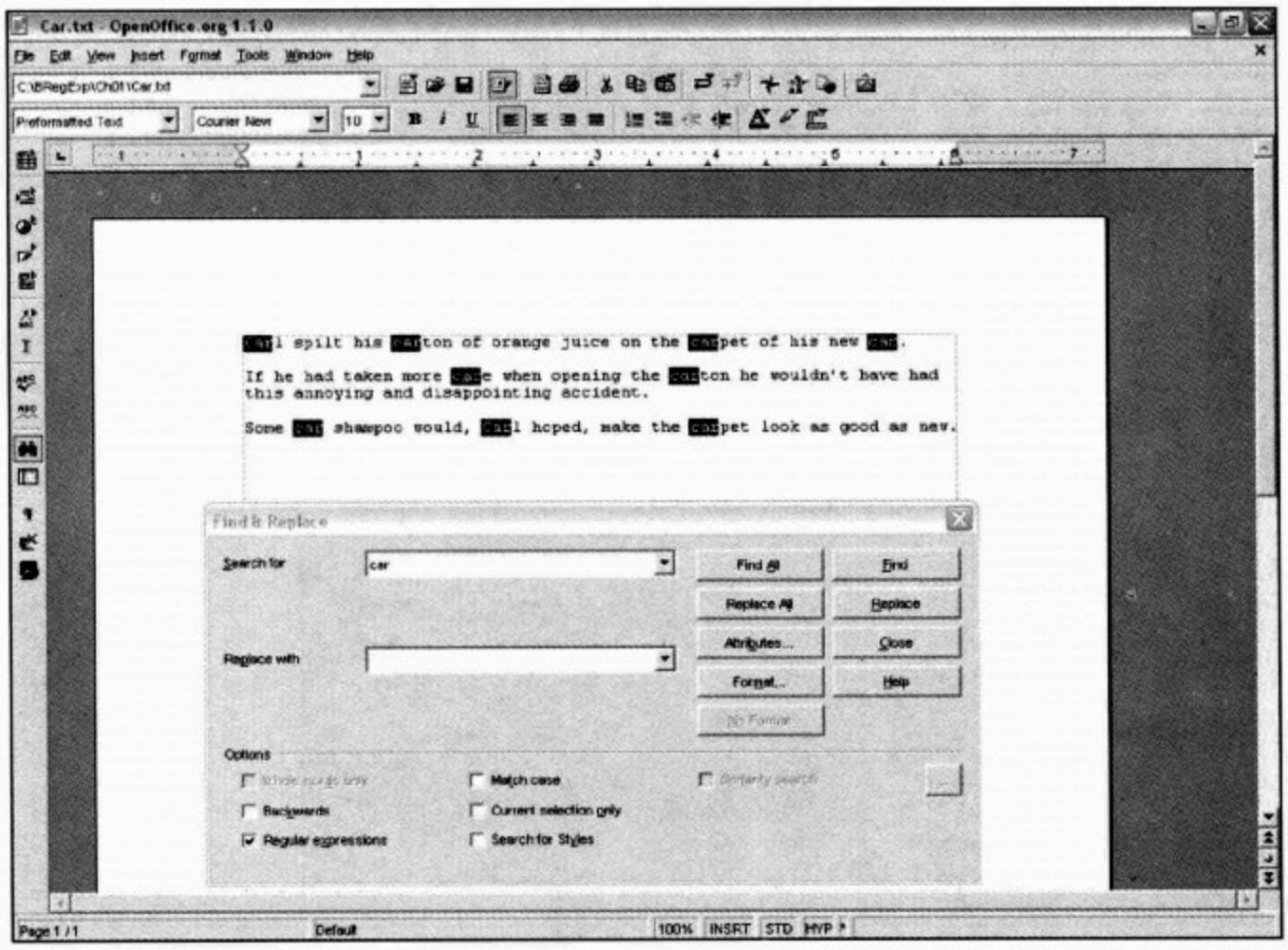


图　1-1

如图 1-1 所示，无论字符序列 car 是不是一个词，也无论这个字符序列的首字母是大写还是小写，结果是所有的 car 字符序列都被选中。

表 1-1 中对这个简单的正则表达式模式 car 进行了更正式的分析说明。

当使用像 car 这么简单的直接量模式时，对正则表达式的含义进行正式设计看起来有点小题大作。然而，当开始建立更加复杂的正则表达式模式时(详见本书后续的章节)，对正则表达式中每一部分进行规划则有助于把握模式中各个部分的含义。

表 1-1　正则表达式模式 car

字　母	说　明
c	匹配字母 c
a	匹配字母 a
r	匹配字母 r

在短文档(如本例中的文档)中，如果在 OpenOffice.org Writer 中搜索时使用 Find All 选项，那么无论匹配的是整个单词还是简单的字符序列，都可以快速找到所有的匹配项，因为这些匹配项都是突出显示的。

但是，这里的正则表达式还可以更严密一些。例如，只想查找与字符序列 car 匹配的

单词 car，而不想查找作为其他单词一部分的 car。而且，可能要执行区分大小写的搜索。

对于上述两种需求，在 OpenOffice.org Writer 中通过使用正则表达式 \<car\> 并选择 Match case 复选框来实现，如图 1-2 所示。其中 \< 和 \> 元字符简单地匹配单词的边界。

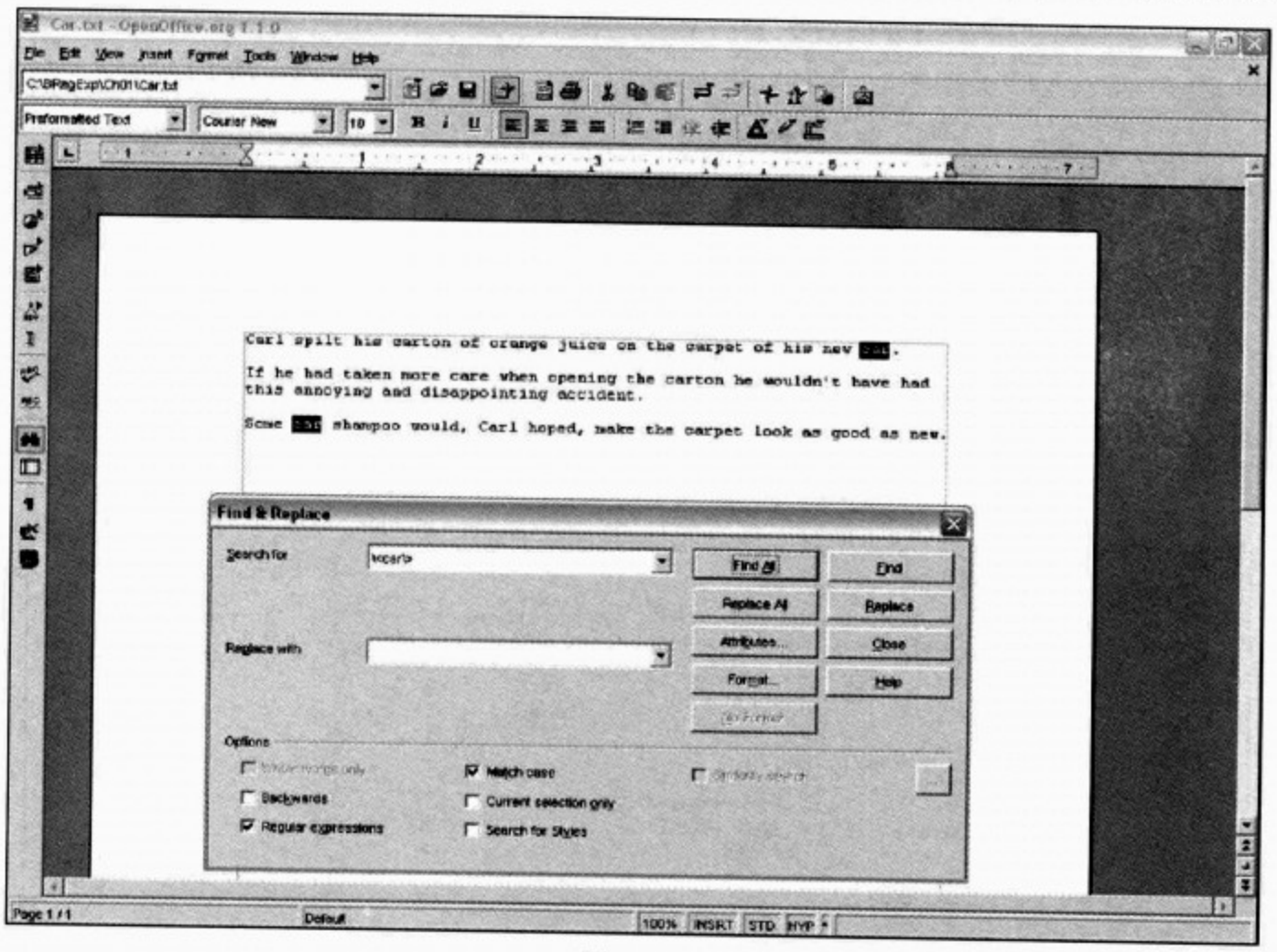


图 1-2

现在不必因为匹配单词的开始与结束位置的语法而担心了。第 6 章中会介绍实现这一功能的更多语法形式。

1.2 可以使用正则表达式做什么

正则表达式的潜在用途不计其数。本节只是简要地介绍几个使用正则表达式的例子。

1.2.1 查找重复的单词

使用正则表达式可以查找文本中是否包含重复的单词。在有些文本——例如下面的句子中，有意使用重复的单词：

```
It is for tasks such as that that regular expressions are used.
```

其中，重复的单词 that 是用来表达思想的一种方式。而在其他情况下，一个单词出现两次则可能是不合适的，也不是有意的，例如下面的句子：

```
Paris in the the Spring
```

查找重复单词的技术将在第 7 章中详细介绍。

在某些设置中，也可能不需要正则表达式来识别重复的单词。例如，上面句子中的第二个 the 通常会在 Microsoft Word 中被加上红色的波纹形下划线(取决于具体设置)。

1.2.2　检查 Web 表单的输入

正则表达式的另一种常见的用法是检查 Web 表单中输入的数据是否符合服务器端处理所允许的结构，以决定能否提交该表单数据。例如，假设有人试图输入下列字符串冒充美国社会保险号(SSN)：

```
Fred-123-4567
```

这不是一个有效的 SSN，因此必须保证它不会进入后台数据库中。如果要在客户端检查这个假的 SSN，可以使用 JavaScript 和它的 RegExp 对象来检查用户输入的字符串是否符合有效 SSN 的模式。

要对来自 Web 表单的输入进行检查的一个原因是，这些收集到的数据将会被发送到数据库(比如关系数据库)。如果有人在表单字段中输入了错误的数据类型，那么就有可能因为把人名写入日期列中而导致数据库出错。此时，还需要确保通过检查发现无效的日期，如 2005-02-31，因为二月份永远不会有 31 号。由于从表单中收集到的数据只是一个简单的字符序列，所以正则表达式就是确保在客户端发现不适当数据的理想选择，这样就可以及时提醒用户输入适当的数据以代替不正确的数据。

1.2.3　转换日期格式

假设我们要将一份美式英语的商务文档“翻译”为英式英语。文档的文本中很可能存在日期，因此必须找到日期并修改它们——因为美国和英国常用的日期表示法不同。例如，在美国 2001 年圣诞节这一天会写成 12/25/2001，而在英国，则会写成 25/12/2001。如果你还要向日本的客户提供这份文档，那么同一日期则需要表示为 2001-12-25。

假设文档中使用了美式英语的习惯表示日期，便可以建立一个正则表达式把文档中所有表示日期的字符序列找出来。然后，根据想要的输出格式决定是把输入端的美式英语日期替换成输出端的英式英语还是日本的日期格式。

1.2.4　发现错误的拼写

因为计算机领域中有许多专业术语和缩写词，所以在计算机相关的文档中很容易隐藏一些错误的拼写。这可能是由于字处理程序按照自己的(错误)想法将它认为不正确的双大写字母(首字母)进行了自动更正而引起的。这里的示例文档 XPath.txt 就说明了技术性文档中可能隐藏的此类问题。

```
Xpath is an abbreviation for the XML Path Language.
XPath is used to navigate around a tree model of an XML document.
There are significant differences in the data model of XpatH 1.0 and Xpath 2.0.
XSLT is one of the technologies with which xpath is often used.
```

XPath 的正确书写方式应该是前两个字母大写。这个例子的文档中存在一些不正确的拼写形式，都是因为一个或多个字符导致的错误。

对于此类问题，明智的解决办法部分取决于判断有争议的词在英语中是否具有实际的意义。在这个 XPath 的例子中只有一个正确的形式，而在英语中它是没有意义的。这样就可以简单地找到与字符串 xpath(无论大小写)匹配的所有内容，并使用正确的字符序列(XPath)把它们都替换掉。

关于这种类型的问题还有一些处理方法，在本书后面的章节中会举出很多解决这一问题的例子。

1.2.5 为 URL 添加链接

假设把一个包含 URL 的文档转换成在 Web 中显示的页面格式。如果 URL 保存在关系型数据库的一个单独的字段中，只要把该字段中的 URL 直接作为 HTML/XHTML 中 a 元素的 href 属性的值就可以了。然而，如果 URL 像下面这样包含在一段文本当中，那么识别 URL 的问题就会变得比较棘手了。

```
The World Wide Web Consortium, the W3C, has developed many specifications for
XML and associated languages. The W3C's home page is located at
http://www.w3.org and its technical reports are located at
http://www.w3.org/tr/.
```

要找到 URL 取决于在非常长的文档中的任何位置都能识别出 URL 来。另外，URL 所指向的网页的实际标题是什么并不知道，因此无法将相应页面的标题作为链接文本，而必须将 URL 同时用做 XHTML 中 a 元素的 href 属性的值和 a 元素开标签与闭标签之间包含的文本内容。我们所要做的就是找到每个 URL，然后使用一段新的结构化文本来替换它(斜体文本 theURL 表示在文本中实际找到的 URL)，如下所示：

```
<a href="theURL">theURL</a>
```

1.3 使用过的正则表达式

如果你使用计算机已经有一段时间了，那么很可能会熟悉一些正则表达式的用法，但是也许你并不知道所使用的文本模式就是正则表达式，比如，在文字处理软件中或者在命令行中查看目录列表时。

1.3.1 在文字处理软件中查找和替换

目前，大多数文字处理软件都能够在某种程度上支持正则表达式，但有的文字处理软件中却看不到正则表达式的字眼。例如，Microsoft Word 支持有限的正则表达式用法，它使用“通配符”来描述对正则表达式模式的支持。

最简单的正则表达式模式就是文本直接量。即，如果想查找 Star 的文本模式，可以直接将这 4 个字符输入 Microsoft Word 的 Find and Replace 对话框中的 Find What 文本框中。本章稍后会介绍，这种方法在搜索大量文本的时候存在一些问题。

1.3.2　目录列表

如果你曾经使用过命令行，那么很可能在查看目录列表时使用过简单的正则表达式。在命令行中，会涉及到两个元字符：* (星号)和 ?(问号)。

例如，在 Windows 平台中，如果想在当前目录中查找可执行的文件，可以在命令行中输入如下命令：

```
dir *.exe
```

其中，dir 命令会列出当前目录中所有的文件，它与下面的命令是等价的：

```
dir *.*
```

而模式 *.exe 则匹配文件名中包含零个或多个字符并且后跟一个句点和一个直接量字符序列 exe 的文件。类似地，模式 *.* 表示零个或多个字符后跟一个句点以及零个或多个字符。

事实上，我们在路径查询中所使用的通配符语法与正则表达式中的通配符语法差别很大，但本节的目的并不是分析路径通配符，而只是想让读者注意到在查找匹配的文本块时已经使用过模式。

在某些情况下，当搜索一个目录时，已经知道想要匹配的实际字符数。假设目录中包含很多 Excel 工作簿文件，每个工作簿中都保存着月份销售数据。如果知道它们的文件名由单词 Sales 后跟表示年份的两位数和表示月份的三个字母组成，那么就可以用如下命令来搜索 2004 年的所有销售工作簿：

```
Dir Sales04???.xls
```

这样就会列出文件名符合刚刚所描述的结构的 Excel 工作簿了。但是，如果目录中还有名为 Sales04123.xls、Sales04234.xls 的工作簿，那这个命令也会列出它们，尽管我们并不需要这两个工作簿。

由上可知，由于路径通配符中只有两个元字符，能够通过 dir 命令使用的正则表达式是非常有限的。

1.3.3　在线搜索

在另外一种应用环境——在线搜索中，也广泛使用了正则表达式(虽然属于简单应用)。例如，在 eBay.com 的搜索框中，也可以使用星号通配符。这样，photo* 可以匹配的词有 photo、photos、photograph 和 photographs 等。

1.4　为什么正则表达式看起来令人生畏

正则表达式之所以令许多开发人员望而却步，主要是因为它高度简洁且神秘的语法。正则表达式的另一个缺陷是它没有标准的规范，因此具有特定含义的正则表达式模式在支

持正则表达式的不同语言和工具之间也不相同。

1.4.1 简洁而神秘的语法

正则表达式的语法非常简洁，而且对于不熟悉正则表达式的人，看起来简直很神秘。有时，正则表达式中看起来好像到处都充斥着反斜杠、圆括号和方括号。而只要理解正则表达式中每个字符和元字符的作用，就能自己编写正则表达式或者分析其他开发人员编写的正则表达式。

元字符是指在正则表达式模式中具有特殊含义的字符或字符组合。

1.4.2 空格会导致含义改变

如果无意间在正则表达式中插入了空格，就会彻底改变正则表达式的含义，从而使得应该匹配的内容不再匹配，而不想匹配的内容却匹配了。所以，在建立正则表达式模式时，必须要小心处理空格符。

例如，假设要在保存人名信息的文档中搜索相关内容。以文档 People.txt 为例，其内容如下所示：

```
Cardoza, Fred
Catto, Philipa
Duncan, Jean
Edwards, Neil
England, Elizabeth
Main, Robert
Martin, Jane
Meens, Carol
Patrick, Harry
Paul, Jeanine
Roberts, Clementine
Schmidt, Paul
Sells, Simon
Smith, Peter
Stephens, Sheila
Wales, Gareth
Zinni, Hamish
```

假设每个名字都预先进行了格式化，那么就可以使用下面的正则表达式来查找所有姓氏以 S 开头的名字：

```
^S.*
```

图 1-3 显示了使用这个正则表达式在 OpenOffice.org Writer 中搜索 People.txt 文档的结果。其中，所有姓氏以 S 开头的名字都被选中了。

然而，如果在这个正则表达式模式的 ^ 和 S 之间插入一个空格字符(如下所示)，那么根本找不到匹配项，如图 1-4 所示。

```
^ S.*
```

这是因为 ^ 元字符是一个表示行开始位置的记号(在 OpenOffice.org Writer 中表示一个段落的开始位置)。因此，在 ^ 字符后面插入一个空格符的意思是，这个正则表达式要搜索以空格符开始的行。由于 People.txt 中的所有文本行都以字母字符开头，没有以空格符开头的行，所以不会有内容与这个正则表达式模式匹配。

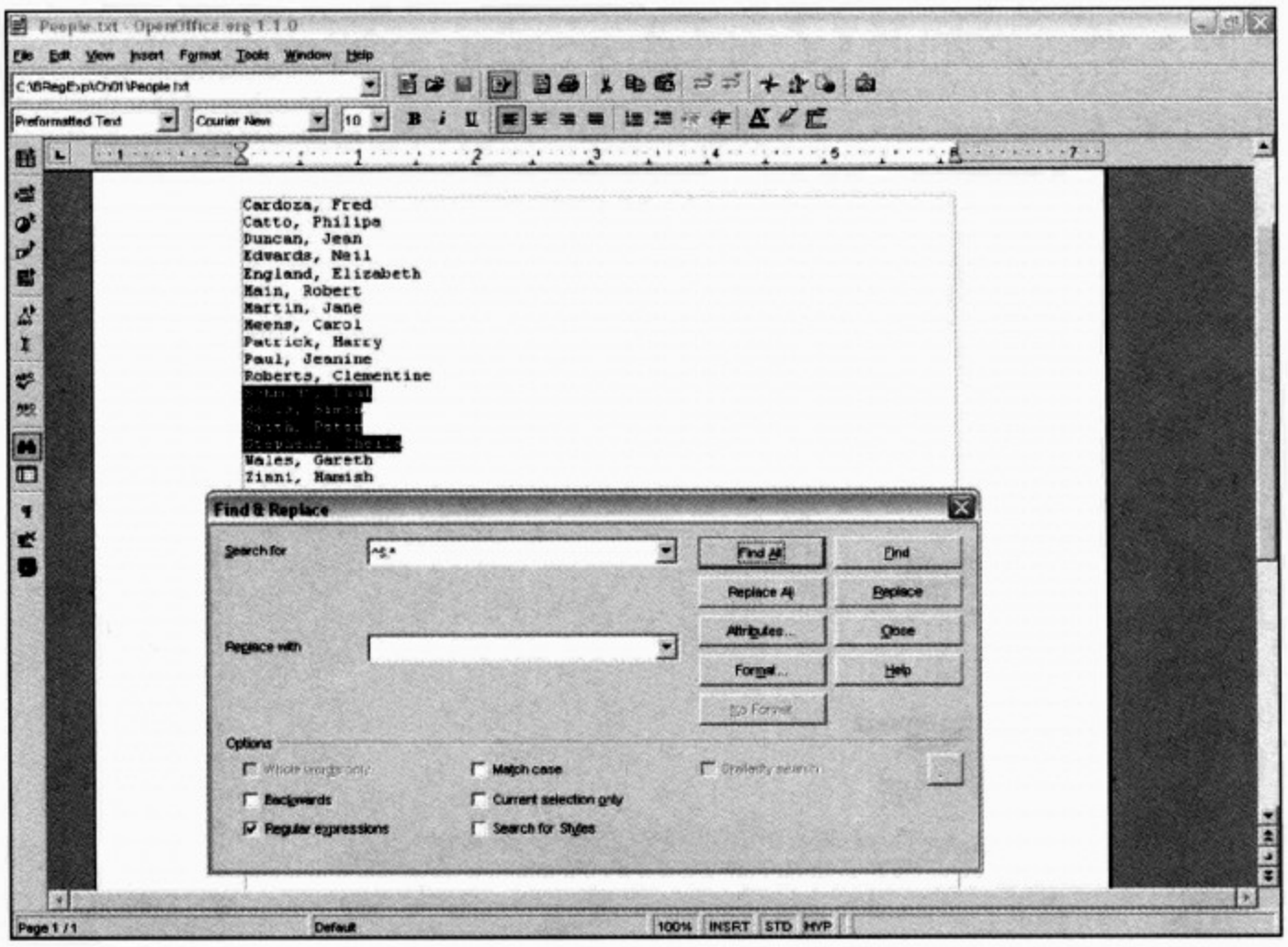


图　1-3

而且，要想检测到这种无意识插入的空格符是很困难的。建议始终要仔细地查看要操作的数据，以便掌握其中的文本特征，进而知道正则表达式模式至少会有一些匹配项。如果遵循这里的建议(有关这个话题第 2 章会详细讨论)，那么便会知道其中一些姓氏是以 S 开头的，因此当正则表达式返回零个匹配项时，就会发现这个正则表达式中可能存在错误。

但是，当正则表达式能返回几个匹配项时，便可能忽略由于意外输入空格符而导致的不想要的匹配项。假设想查找姓氏以 C、D、R 或 S 开头的名字，那么可以使用如下的正则表达式：

```
^(C|D|R|S).*
```

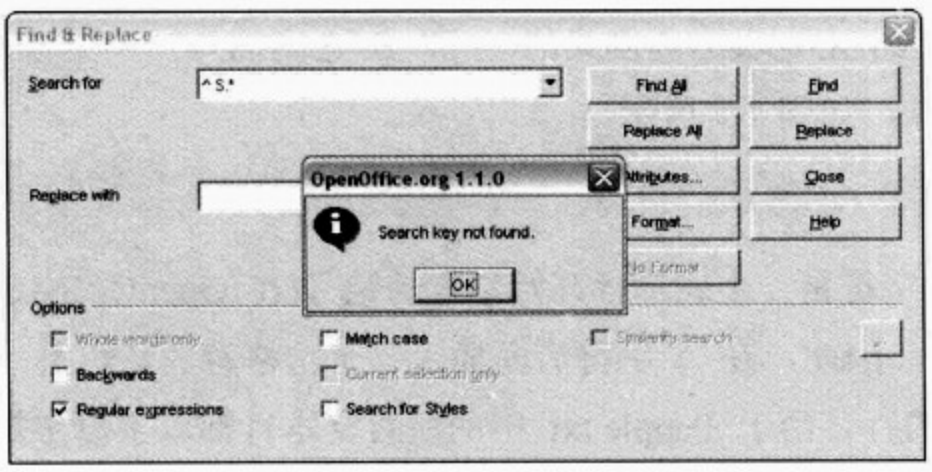


图 1-4

但如果不经意间在 D 后面插入了一个空格(如下所示)，会导致什么结果呢？

```
^(C|D |R|S).*
```

对于这个正则表达式，你可能不会注意到其中有一个空格，或者没有意识到这个空格会改变整个模式的含义。而且，由于这个模式返回了一些匹配项，你很可能会被这种“成功匹配”的假象所迷惑，尽管其中不会包含姓氏以 D 开头的名字，如图 1-5 所示。

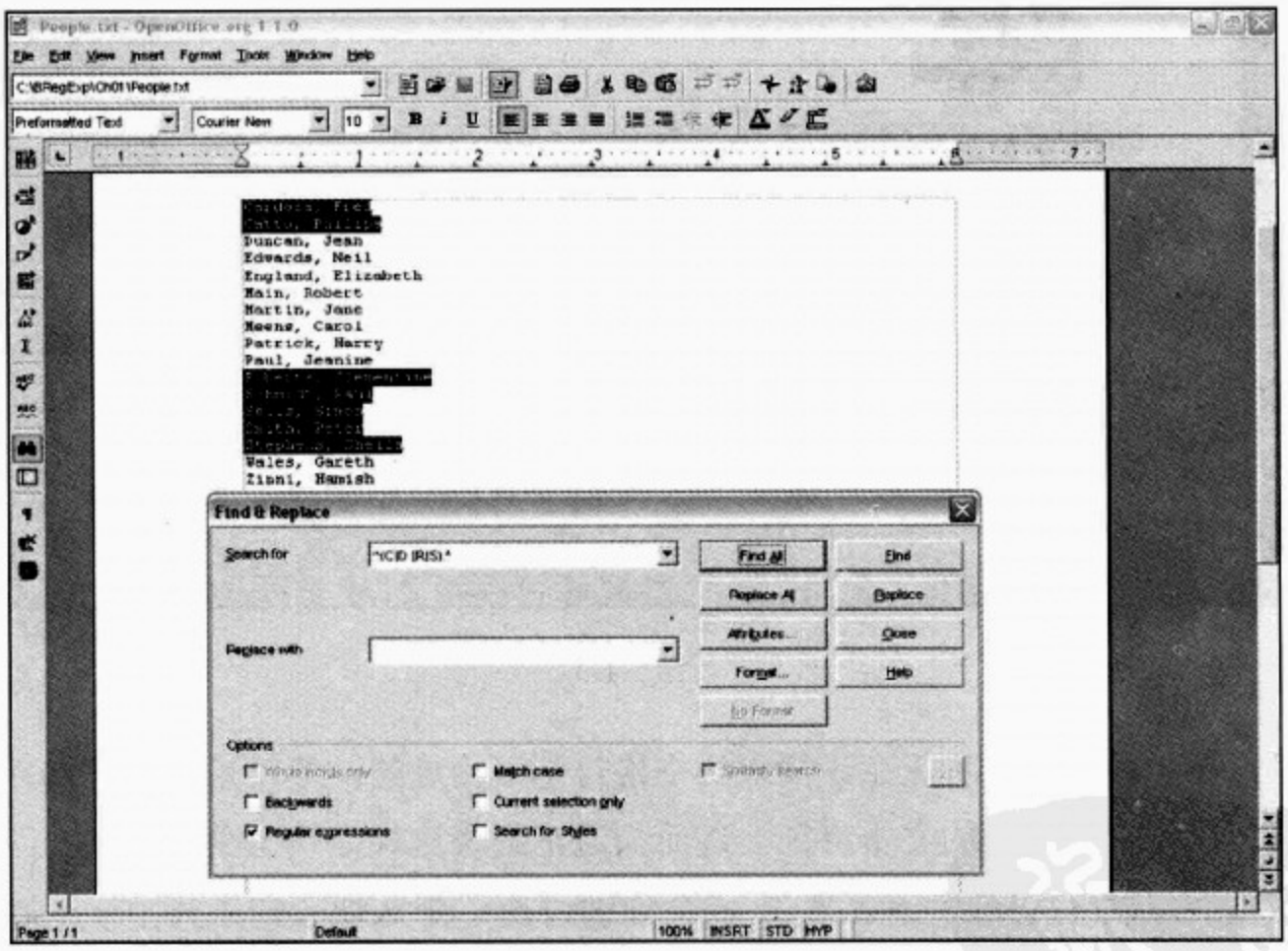


图 1-5

对于 People.txt 文档中少量的有序数据，可以很容易就注意到匹配结果中不包含姓氏以 D 开头的名字。如果是在一个包含大量文本且数据未经排序的文档中，要想发现这个问题就并非易事了。

第 4 章将讨论如何在正则表达式内部使用空格符。

1.4.3　没有统一的语法标准

正则表达式多样化是因为没有统一的标准定义正则表达式的语法。正则表达式最早是因为被 Perl 语言采用才引起人们注意的，由于遵循不同程度的精确性，导致了其他语言和应用程序中的正则表达式语法不相同。

由于正则表达式没有统一的标准，不同实现之间的差异也非常多。

1.4.4　各种实现之间的差别

虽然很多实现都支持正则表达式和通配符，但很多实现是不规范的。

如图 1-2 中所示，使用正则表达式模式 \<car\> 可以找到作为一个单词出现的字符串 car。但是，如果在 Microsoft Word 中执行同样的搜索——查找作为一个单词出现的 car，如图 1-6 所示，就必须使用正则表达式模式 <car>。

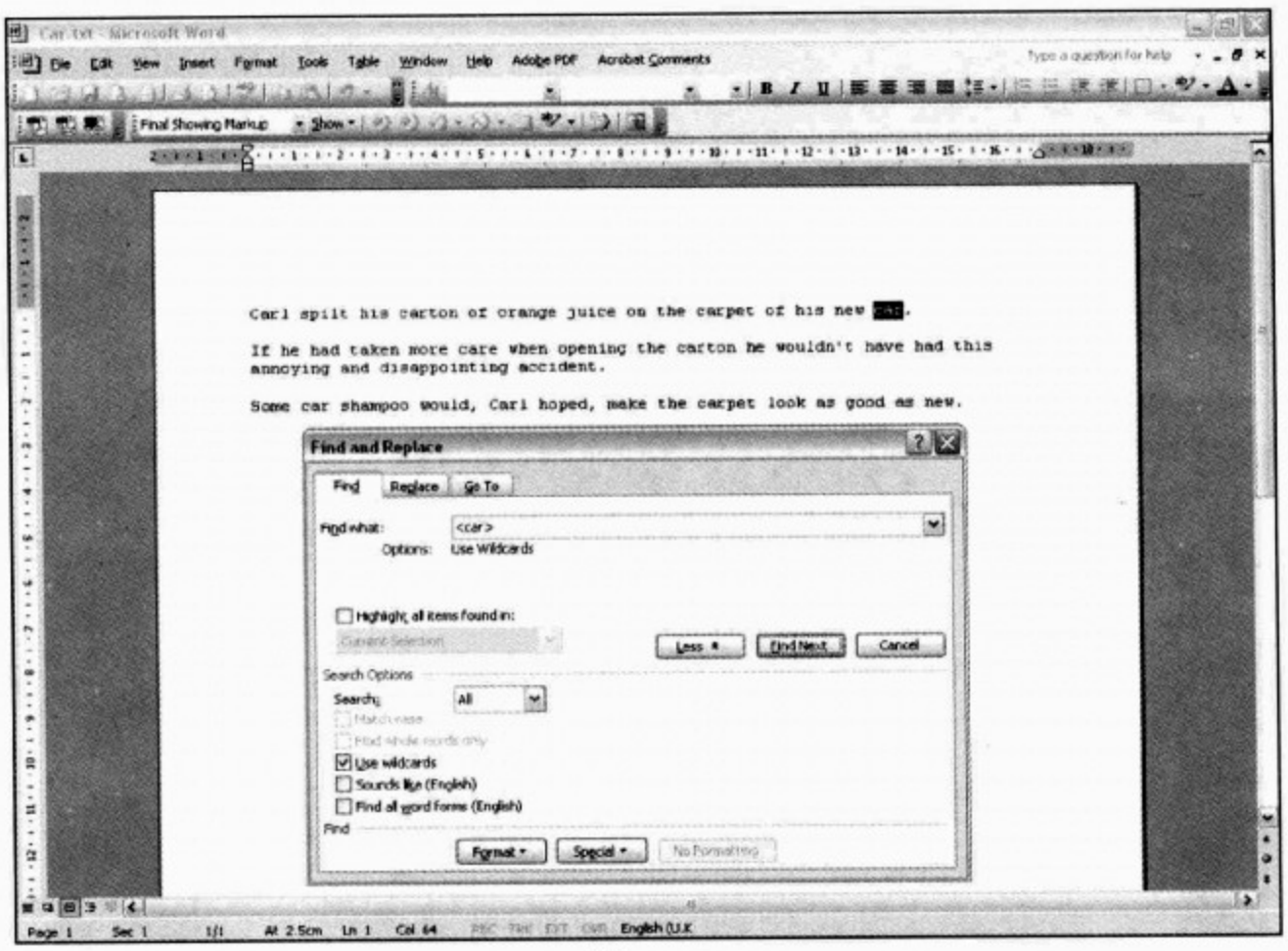


图　1-6

如果我们想查找单个字母，需要在诸如 OpenOffice.org Writer 和 Perl 这样的脚本语言中使用. (句点)元字符。但在 Word 中实现同样的功能则要使用 ?(问号)。在这里很显然 Word 是不规范的，而且像文件路径中那样的用法也并不是真正的正则表达式用法。

关于不同的正则表达式实现之间的区别，我们会在后面的几章里通过基本技术的介绍来讨论。另外，本书在后面的每一章都会专门讨论一种具体的语言或应用程序，以及其正则表达式实现的详细特点。

1.4.5　不同环境下的字符含义不同

正则表达式容易使人感到疑惑的另一个原因是，单个字符或者元字符，其使用环境不

同，含义也会不同。

例如，^ 元字符在某些语言的正则表达式中用于表示一行的开始位置。但就在同一种语言环境下，^ 元字符在字符类中使用时，就变成了否定的含义(或非、取反、补集——即不包含的意思。译者注)。由上可知，正则表达式模式 ^and 匹配字符序列 and 位于一行开头的情况，而正则表达式 [^and] 则表示一个不包含 a、n、d 的字符类。

我们会在第 5 章中介绍有关字符类的内容。

下面是一个用于测试的文档 And.txt，其内容如下：

```
and
but
and
Andrew
sand
button
but
band
hand
```

如果在 OpenOffice.org Writer 中使用正则表达式模式 ^and，那么表明选择位于一行开始位置的字符序列 a、n、d(忽略这些字符的大小写)的情况。图 1-7 中显示的就是在 OpenOffice.org Writer 中使用这个正则表达式的情况。

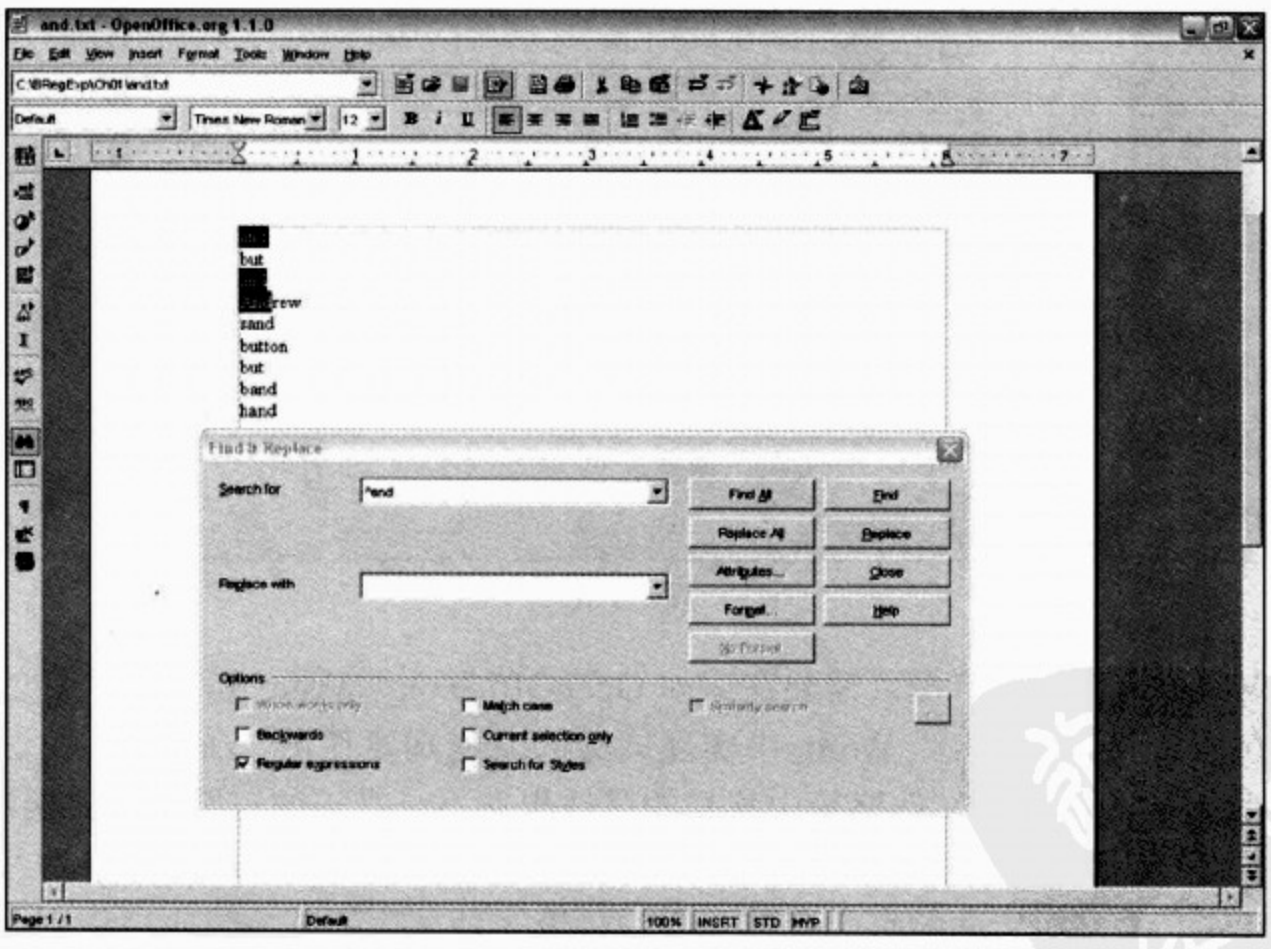


图 1-7

从图中可以看到，这个正则表达式选中了位于一行开始位置的单词 and(两次)和 Andrew 的前三个字符。之所以会选中 Andrew 的前三个字符，是因为这里的搜索没有区分大小写。

但是，如果在正则表达式模式 .[^and].* 中把 ^ 元字符用在了字符类的内部，那么正如从图 1-8 中所看到的，除 a、n、d 之外的字符序列会被选中。

这个正则表达式模式 .[^and].* 的含义是，匹配一个字符后不跟 a、n、d 字符，并且后可跟零个或多个任意字符的情况。

当把 ^ 元字符用在字符类中时，如果不把它放在第一个字符的位置上，那么它的含义与用在字符类外部时是一样的。换句话说，只在脱字字符(^)作为字符类左方括号后面的第一个字符时，它的作用才是否定(取反)。此时不必为理解这个问题而担心，第 5 章会对相关的话题进行详细介绍。

另一个在字符类内、外部具有不同含义的字符是连字符(-)。如果在字符类外部，短划线(连字符的另一种叫法。译者注)表示它自身，例如下面日期值中包含的短划线：

```
2004-12-25
```

但是，如果是在字符类内部，当短划线不是第一个字符时，它表示一个范围。例如，要指定一个包含所有小写和大写字母的字符类，可以使用下面的模式：

```
[a-zA-Z]
```

这个字符类是下面模式的简写形式：

```
[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]
```

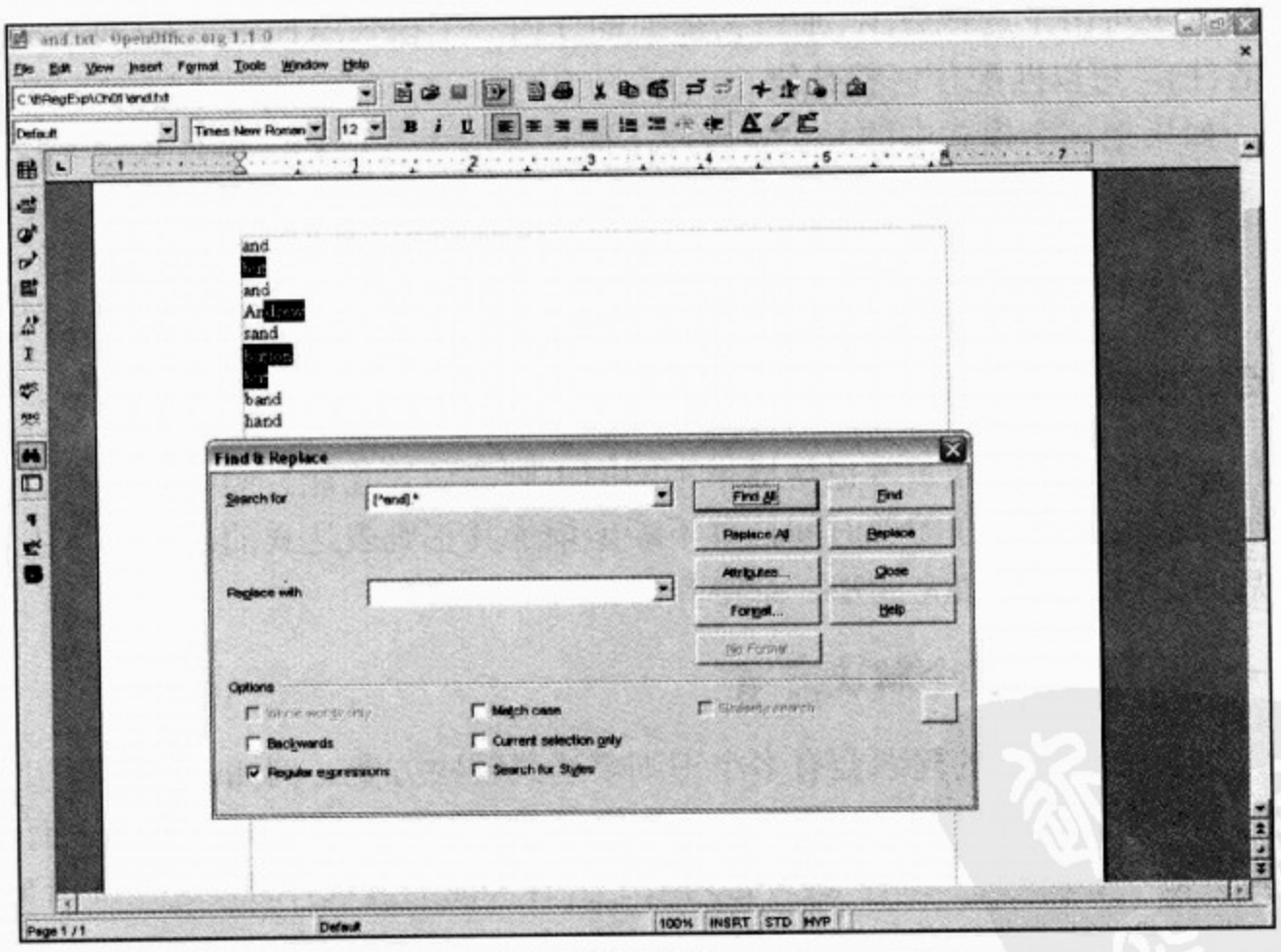


图　1-8

1.4.6　正则表达式可以区分大小写

使用正则表达式时会遇到的一个更复杂的问题，是在某些环境下正则表达式会区分大小写。区分大小写的意思有两方面：一方面指按照是否区分大小写的原则进行匹配；另一

方面指，元字符区分大小写。

1. 是否区分大小写的匹配

个别正则表达式的实现会以不同方式处理大小写。例如，OpenOffice.org Writer 在其 Find and Replace 对话框中提供了一个 Match Case 复选框，在本章前面的例子中提到过它默认进行不区分大小写的匹配。

相对而言，在编程语言中默认情况下通常是区分大小写的。许多编程语言也为在使用正则表达式时是否区分大小写提供了开关选项。这些问题会在第 4 章中介绍。另外，在介绍一些编程语言的章节中也列举了很多是否区分大小写搜索的例子。

2. 大小写和元字符

大小写对于元字符该如何解释往往至关重要。例如，如果想匹配数字，可以使用以下模式：

```
\d
```

这个模式可以匹配数字 0~9，因为 \d 与下面这个模式是等价的。

```
(0|1|2|3|4|5|6|7|8|9)
```

这个模式也可用于选择数字。通过为前面的任何一个模式添加限定符(即表示数量的元字符。译者注)，可以匹配任何整数值。

然而，如果把这个模式中的元字符改成下面这样的大写形式，那么其含义就会改变：

```
\D
```

模式 \D 用于匹配任何不是 0~9 的字符。

1.4.7 支持性技术的不断发展

同一门编程语言的不同版本也会提供不同的正则表达式功能。例如，Perl 这个首先支持正则表达式的语言，就在随着时间推移不断地增加其正则表达式的功能。其中一些差别会在本书后面专门讨论该语言的那一章中介绍。

1.4.8 一个问题对应多个解决方案

通常，对于某个特定的问题会有多个正则表达式解决方案。例如，如果你想列出库存零件目录中以一个大写字母开头，后跟两位数字的零件编号，那么可以使用下面的任何一种模式。其中第一个模式非常长，因为它列出了所有可能的选项，而且这些选项是互斥的：

```
(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)\d\d
```

在该模式的数字部分中，可以为 \d 元字符使用一个限定符：

```
(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)\d{2}
```

或者，也可以通过使用字符类使这个模式更加简洁：

```
[A-Z]\d{2}
```

对于经验丰富的用户来说，正则表达式语法的这种灵活性很有用。但是，对初学者而言，这只会让他们感到困惑不已。

1.4.9　使用正则表达式做什么

如何正确使用正则表达式取决于你使用它的目的和要操作的源数据内容。同样也取决于你对源数据的了解程度和所定义模式的精确程度。

例如，如果一个文档中包含一些 URL，只用字符 http 来通过纯文本搜索就能找到所有 URL。具体而言，如果万维网联盟(World Wide Web Consortium)的主页以 http://www.w3.org/ 的形式表示，以上方法就能奏效。但如果是以 www.w3.org/ 的形式表示，就会完全无效。因此，数据决定了模式的匹配行为。

当你认为 W3C 的主页 URL 可能是 www.w3.org 或者 www.w3c.org，但文档中使用的是哪一个却记不清了，那么可以通过以下模式来执行搜索：

```
\.w3c?\.org
```

这个模式会通过元字符 \. 匹配 URL 中的直接量句点字符，以及后跟直接量字符 w3 和一个可选的直接量字符 c，然后又是与 \. 元字符匹配的直接量句点字符，最后匹配直接量字符 org。

有关元字符的内容会在第 4 章中讨论，其中有许多在正则表达式中如何使用元字符的例子。

1.5　支持正则表达式的语言

支持正则表达式的工具和语言非常多，包括文本编辑器、文字处理程序、脚本语言和功能完整的编程语言等。其中一些程序和语言将在本书后面的几章中详细介绍。

在本章最后，我们再来看一个例子。这个例子中的问题是你很可能会遇到的，而且如果能正确地使用正则表达式，你会发现在解决这个问题时正则表达式是非常有用的。

1.6　替换大量文本

正则表达式的一种用途是在数千份文档中替换大量的文本。如下面的这个例子所示，如果你对自己要处理的问题缺乏透彻的理解，那么很容易会导致更大的问题。

设想一下，为了完成暑假实习的任务你刚刚加入一家虚构的公司—— Star Training Company。就在你加入该公司之际，突然有人决定要把 Star Training Company 这个名称改为 Moon Training Company。至于改名的原因可能是公司被接管或者更换了管理层人员。但是公司改名的结果会直接导致公司的网站需要修改，而且大量的内部和公共文档也必须更新。因为你在这家公司处于最底层，当然就把这个任务交给了你。

在你实习的第一天，你被要求更新 Moon Training Company 的文档和网站，以便在数百份现有的公司文档中显示出新的公司名称。

首先打开第一份文档——StarOriginal.doc，其内容大致如下：

```
Star Training Company

Starting from May 1st Star Training Company is offering a startling special offer
to our regular customers - a 20% discount when 4 or more staff attend a single Star
Training Company course.

In addition, each quarter our star customer will receive a voucher for a free
holiday away from the pressures of the office. Staring at a computer screen all day
might be replaced by starfish and swimming in the Seychelles.

Once this offer has started and you hear about other Star Training customers
enjoying their free holiday you might feel left out. Don't be left on the outside
staring in. Start right now building your points to allow you to start out on your
very own Star Training holiday.

Reach for the star. Training is valuable in its own right but the possibility of a
free holiday adds a startling new dimension to the benefits of Star Training
training.

Don't stare at that computer screen any longer. Start now with Star. Training is
crucial to your company's well-being. Think Star.
```

你是第一天上班，而之前从未做过类似的事。因此，你在自己喜欢的文字处理程序中打开这个文档，调出 Search and Replace 对话框，选择使用 Moon 来替换 Star，最后单击了 Replace All 按钮。

替换后的结果保存在 StarSimpleReplace.doc 中(内容如下)。请仔细地看一看这个简单的查找替换的结果。

```
Moon Training Company

Moonting from May 1st Moon Training Company is offering a Moontling special offer
to our regular customers - a 20% discount when 4 or more staff attend a single Moon
Training Company course.

In addition, each quarter our Moon customer will receive a voucher for a free
holiday away from the pressures of the office. Mooning at a computer screen all day
might be replaced by Moonfish and swimming in the Seychelles.

Once this offer has Moonted and you hear about other Moon Training customers
enjoying their free holiday you might feel left out. Don't be left on the outside
Mooning in. Moont right now building your points to allow you to Moont out on your
very own Moon Training holiday.

Reach for the Moon. Training is valuable in its own right but the possibility of a
free holiday adds a Moontling new dimension to the benefits of Moon Training
```

```
training.

Don't Moone at that computer screen any longer. Moont now with Moon. Training is
crucial to your company's well-being. Think Moon.
```

如你在这个文本中所看到的那样，很多地方都是错误的。虽然你的确替换了原始文档中所有的单词 Star，但同时也替换了很多不该替换的地方。例如，第一个句子的开始就不应该是 Moonting from May 1st。而且，文档中还存在其他问题，包括在一些句子中创造了英语中根本不存在的词和不适当的首字母大小写形式。

随着对以后几章的学习，在对文档 Star.txt 进行测试时会看到能够取得更好结果的方法。

第 2 章

正则表达式工具和使用方法

本章简单介绍一下可以使用正则表达式的一些工具。基本掌握这些工具之后，有助于理解后面几章中大量的例子，为示范可能用到正则表达式的种种情形夯实基础。

基于 Windows 平台的许多工具至少都会支持一些正则表达式——或者 Word 和其他程序中使用的术语：通配符。本书后面各章还会对下面几节中介绍的这些工具进行更加深入的讨论。

本章的后半部分内容会引导你按照严格的步骤来学习正则表达式的使用方法。一种系统性地使用正则表达式的方法(不是最简单的方法)，可以为创建能够实现你意图的正则表达式提供保障。当你为所创建的正则表达式编写详细的注释时，这种系统性的方法也能够使将来维护这些正则表达式变得更加简单且富有效率。

在本章中将学习以下内容：

- 如何使用一些正则表达式工具
- 如何在一些流行的编程语言和数据库管理系统中使用正则表达式

2.1 正则表达式工具

本节会介绍一些使用正则表达式的基于 Windows 平台的实用程序、工具和语言。其中的一些工具(如 MySQL)也可以在其他平台中使用，而不仅限于 Windows 平台。本章中简要的介绍性描述适用于这些工具在 Windows 平台中的特性和行为。注意，某些工具为特定平台所设计的不同版本间会存在一些小的差异。

在接下来的例子中，假设你已经下载了本书的源代码，并把它解压到了 C:\BRegExp\Ch02 文件夹中。如果把源代码解压到了其他地方，那么就需要对下面的指示进行适当的调整。

本书中会多次讨论到的一个问题是：由于这些工具实现正则表达式的方式不同而导致的差异。本章中的介绍性描述也会提到不同实现间的差异或者不规范的用法，不过有关这些差异更全面的描述则放在了本书后面的章节中。

2.1.1　findstr

在某些版本的 Windows 系统中，可以找到命令行实用程序 findstr。要在 Windows XP 中运行 findstr 实用程序，只需打开命令提示符窗口，并在命令行提示中输入以下命令：

```
findstr /?
```

回车后会看到如图 2-1 所示的画面。

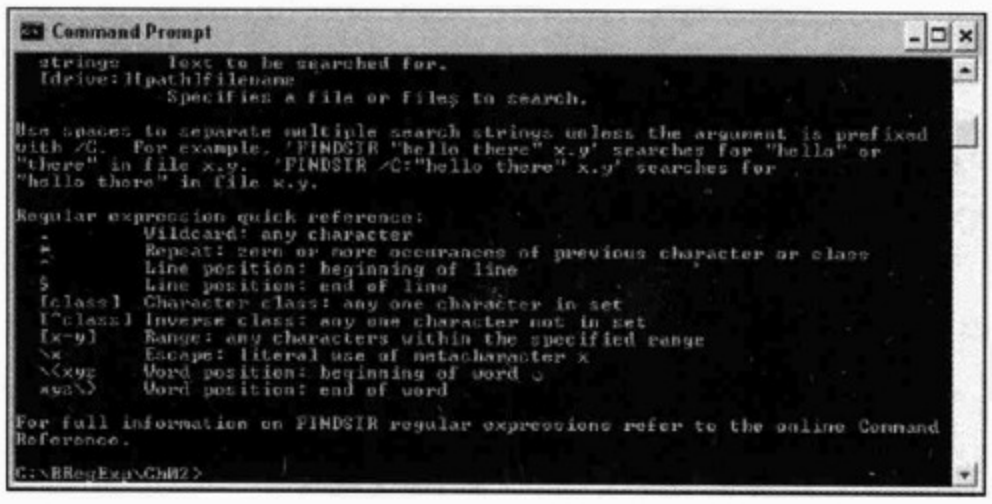


图　2-1

如果只在命令行中简单地输入 findstr 并按回车键，那么很可能会收到“错误的命令行”的错误信息，这是因为没有在命令行中提供必需的参数。

在 C:\BRegExp\Ch02 目录中，有一些可以使用的测试文件。第一个文件是 Test.txt，其内容如下所示：

```
test
text
tent
teat
```

在同一个目录中可以使用 findstr 实用程序，来查找包含文本 tent 的任何文本行，使用命令如下：

```
findstr /N tent *.*
```

严格来说，此时命令行中的文本 tent 就是一个正则表达式模式。参数 /N 表示在匹配的每行前显示行号，以及文件名和匹配的文本。因此，上面命令的结果如图 2-2 所示。

图　2-2

如图 2-2 所示，首先显示的是文件名，其次是包含与直接量正则表达式模式 tent 匹配的文本行的行号，然后是包含与直接量正则表达式模式 tent 匹配的文本行的文本内容。

在通过 findstr 实用程序使用正则表达式时也存在一些限制。比如说，findstr 实用程

序缺少能够确定单个可选字符数量的元字符。为了实现某些目标，使用可选字符出现不止一次的元字符的确可以适当地满足需要。但在其他情况下，它可能会导致一些不想要的匹配。

第 13 章将会对通过 findstr 实用程序来使用正则表达式进行更详细的讨论。

2.1.2 Microsoft Word

Microsoft Word 中所提供的通配符功能，是对某些相当简单的正则表达式功能的不完整及非规范的实现。

通配符不同于很多正则表达式实现中所支持的元字符，但它同样需要匹配文本中的字符的模式。

要在 Microsoft Word 中使用通配符功能，使用键盘快捷键 Ctrl+F 打开 Find and Replace 对话框。默认情况下，Word 中的搜索功能使用简单的直接量文本搜索。通过选中相应的复选框就可以开启通配符功能。而要访问该复选框则必须在 Find and Replace 对话框中单击 More 按钮(如图 2-3 所示，该图是 Word 2003 的屏幕截图)。

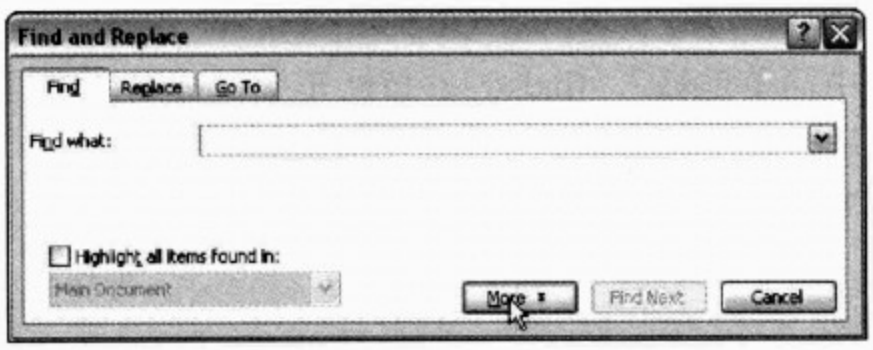


图 2-3

此时，Find and Replace 对话框中会显示出更多的选项。要使用通配符功能，单击 Use wildcards 复选框即可，如图 2-4 所示。

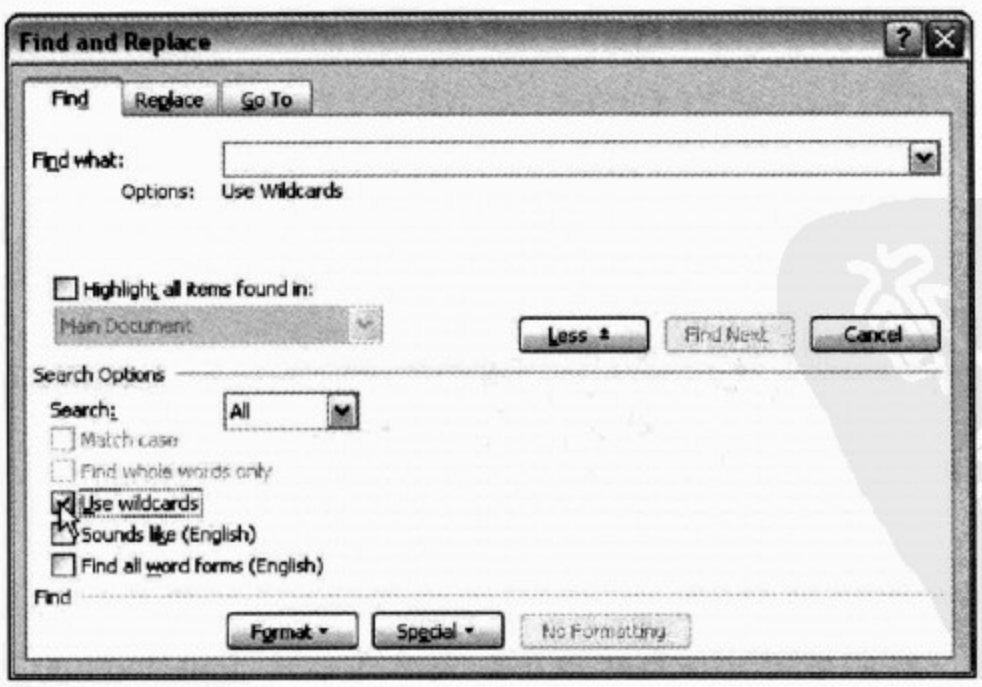


图 2-4

用通配符功能可以在一些单词中搜索只有一个字符不同的字符序列。假设想搜索下列单词，它们均在同一个文件 ight.txt 中：

```
right
sight
might
light
```

在 Microsoft Word 文档中，通过以下正则表达式模式可以找到以上所有单词：

```
?ight
```

其中问号(?)是表示一个单独字母字符的非标准的 Word 通配符。

Word 一次只能突出显示一个匹配项。图 2-5 显示的是使用正则表达式模式 ?ight 在文档中搜索了一次的结果。图 2-6 中显示的是匹配项 sight，这个匹配项在单击两次 Find Next 按钮后才突出显示。

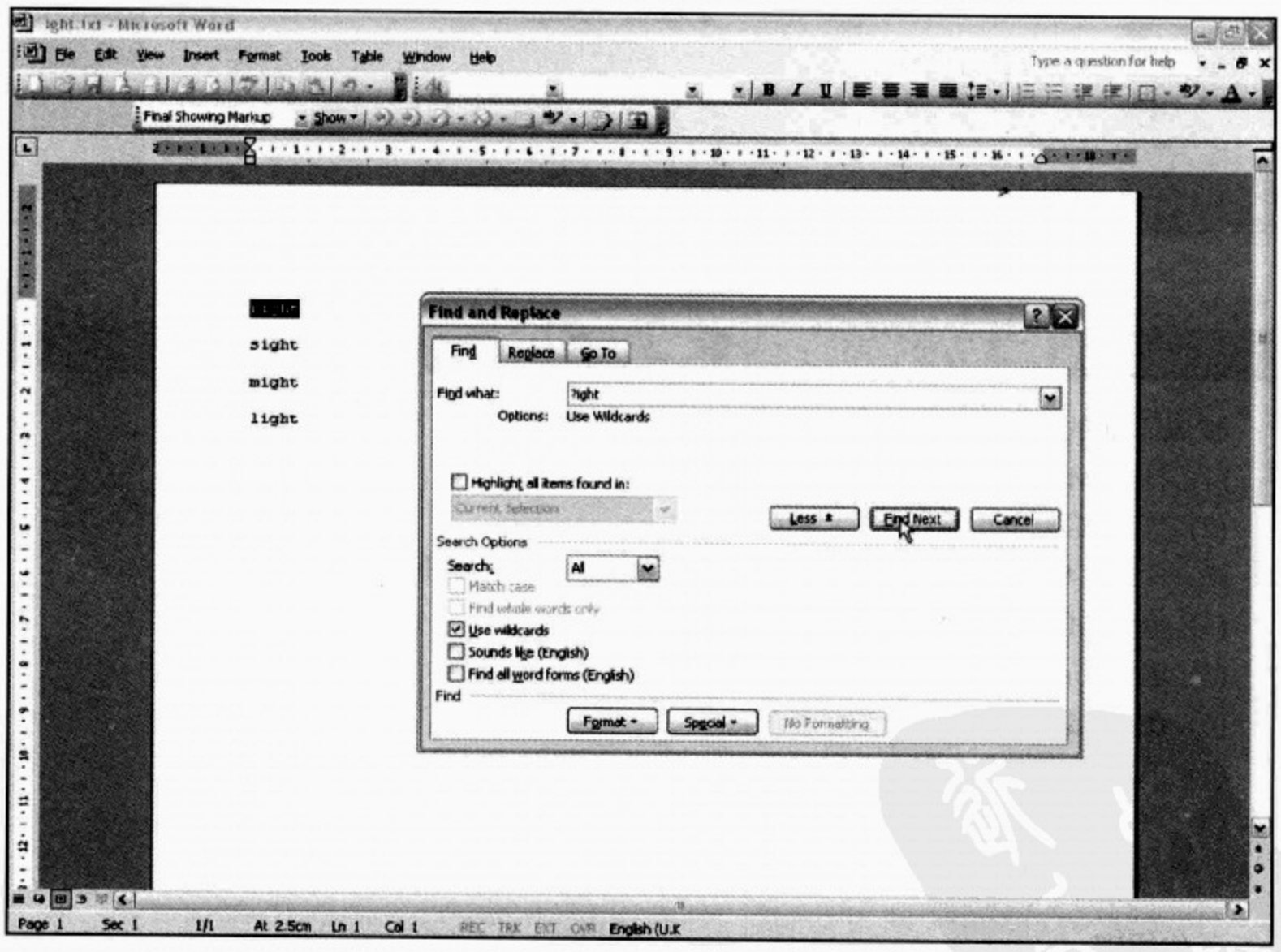


图　2-5

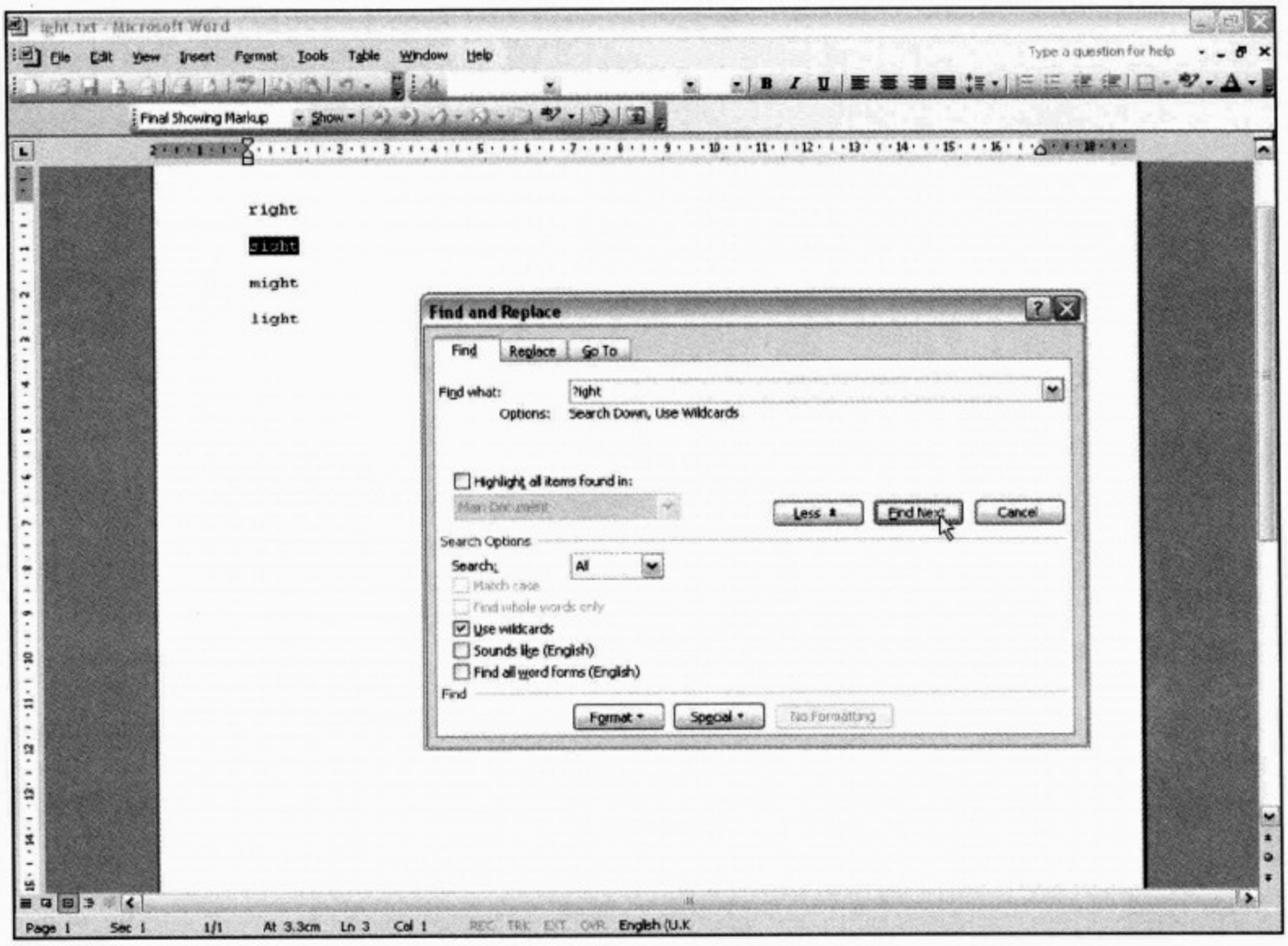


图 2-6

试一试：使用 Word 查找匹配项

按照下面的指示来体验这个简单的例子：

(1) 打开 Microsoft Word。

(2) 打开文件 ight.txt。

(3) 使用 Ctrl+F 键盘快捷键打开 Find and Replace 对话框。

(4) 单击 More 按钮。

(5) 选中 Use wildcards 复选框。

(6) 在 Find What 文本框中输入?ight。

(7) 单击 Find Next 按钮突出显示第一个匹配项。

(8) 再次单击 Find Next 按钮查找其他匹配项。

文件 ight.txt 中的这四个单词会依次成为匹配项。

工作原理

Word 中的正则表达式引擎尝试查找下列模式的匹配项：

```
?ight
```

在 Word 中，? 元字符匹配任何单个字母字符，但不匹配其他 ASCII 字符。比如说，

它不匹配同样包含在 ight.txt 文件中的换行符。当 ？元字符找到匹配的字符时，会根据模式 i(小写的 i)尝试匹配下一个字符。如果同样也找到一个匹配的字符，则会根据模式 g (小写的 g)匹配下一个字符。然后再根据模式 h(小写的 h)继续匹配下一个字符。如果全部四次匹配都成功，则继续根据模式 t(小写的 t)查找匹配的字符。如果所有匹配都成功，Word 就会把匹配结果突出显示(如图 2-5 和 2-6 所示)。在这个例子中，第一个匹配项是单词 right，它会在第一次单击 Find Next 按钮后突出显示。

当再次单击 Find Next 按钮后，Word 会尝试查找其他匹配项。而当在单词 sight 中匹配了词首的 s 以及 s 后面的四个字符后，就找到了另一个匹配项。

在 Microsoft Word 中还有很多使用通配符的方式。例如，模式 h?nd 会匹配字符序列 hand 和 hind，但不会匹配 hound。因为 hound 的 h 和 nd 之间有两个字符。而在 Word 中 ？元字符只能严格地匹配一个字母字符。

在多数正则表达式的实现中，？元字符是一个限定符，用于限制字符或组出现零次或一次(也就是可选)。

Word 中的另一个通配符——星号(*)，匹配零个或多个字母字符。因此，模式 h*nd 会匹配 hand、hind 和 hound。

关于在 Word 中使用正则表达式(通配符)的内容将在第 11 章中作详细介绍。

2.1.3　StarOffice Writer/OpenOffice.org Writer

OpenOffice.org Writer 1.1 和 Sun StarOffice Writer 6 及更高版本，都支持正则表达式。OpenOffice.org 是由文字处理程序、电子表格和演示工具包组成的一款开源程序的官方名称。

本书前几部分中提供的很多例子都是通过 OpenOffice.org Writer 打开例子文档来演示的。OpenOffice.org Writer 中的正则表达式实现在许多方面都很规范，而且通过它无须具备某种特定编程语言的知识就可以方便地示范正则表达式模式的效果。由于 OpenOffice.org Writer 的正则表达式实现是最规范的，所以它也是特别有用的教学工具。而且，在 OpenOffice.org Writer 中可以实时地显示某个正则表达式模式的所有匹配项。

关于在 OpenOffice.org 中使用正则表达式的内容将会在第 12 章中作详细介绍。

2.1.4　Komodo Rx Package

Komodo 开发人员编辑器软件是 Active State(其网站是 www.ActiveState.com)公司的产品。这款编辑器软件中有一款 Komodo Regular Expressions Toolkit 工具，通过该工具我们可以根据字符序列来方便地测试正则表达式模式。

Komodo Regular Expressions Toolkit 允许开发人员根据测试字符串来测试正则表达式。

图 2-7 是使用 Komodo Regular Expressions Toolkit 来查找模式 .ight 的匹配项时的屏幕截图。在多数正则表达式实现中，句点字符(也称为点字符)用于匹配单个字符(除了换行符和其他类似的字符——但也取决于在某些情况下的具体设置)。

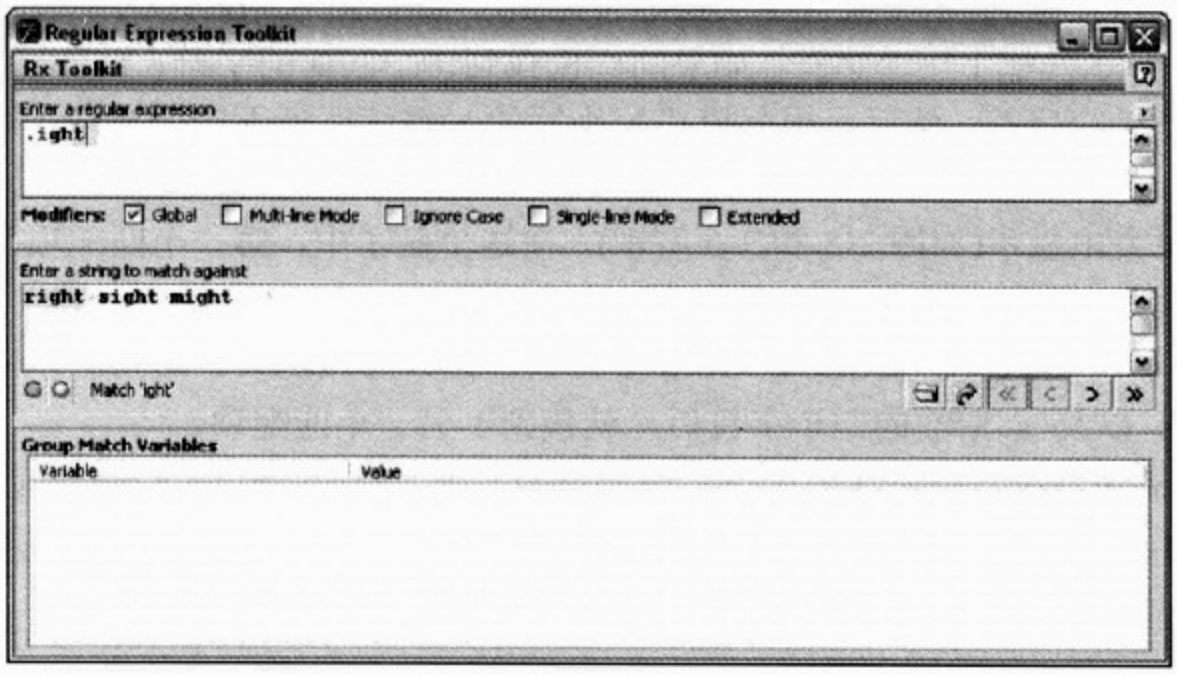


图 2-7

本章和后面几章的例子中使用的是 Komodo Regular Expressions Toolkit V 2.5。第 26 章使用的是 Komodo V 3.0。

2.1.5 PowerGrep

PowerGrep 是一款功能强大、灵活且具有示范意义的工具软件，因为它实现了正则表达式的很多功能。也可以使用它来完成现实的“搜索和替换”操作，并通过 Folder 和 File mask 文本框来指定搜索目标，如图 2-8 所示。

关于在 PowerGrep 中使用正则表达式的内容将在第 14 章中介绍。

2.1.6 Microsoft Excel

Microsoft Excel 支持通配符的有限功能。第 15 章会介绍在 Excel 中使用通配符的内容。

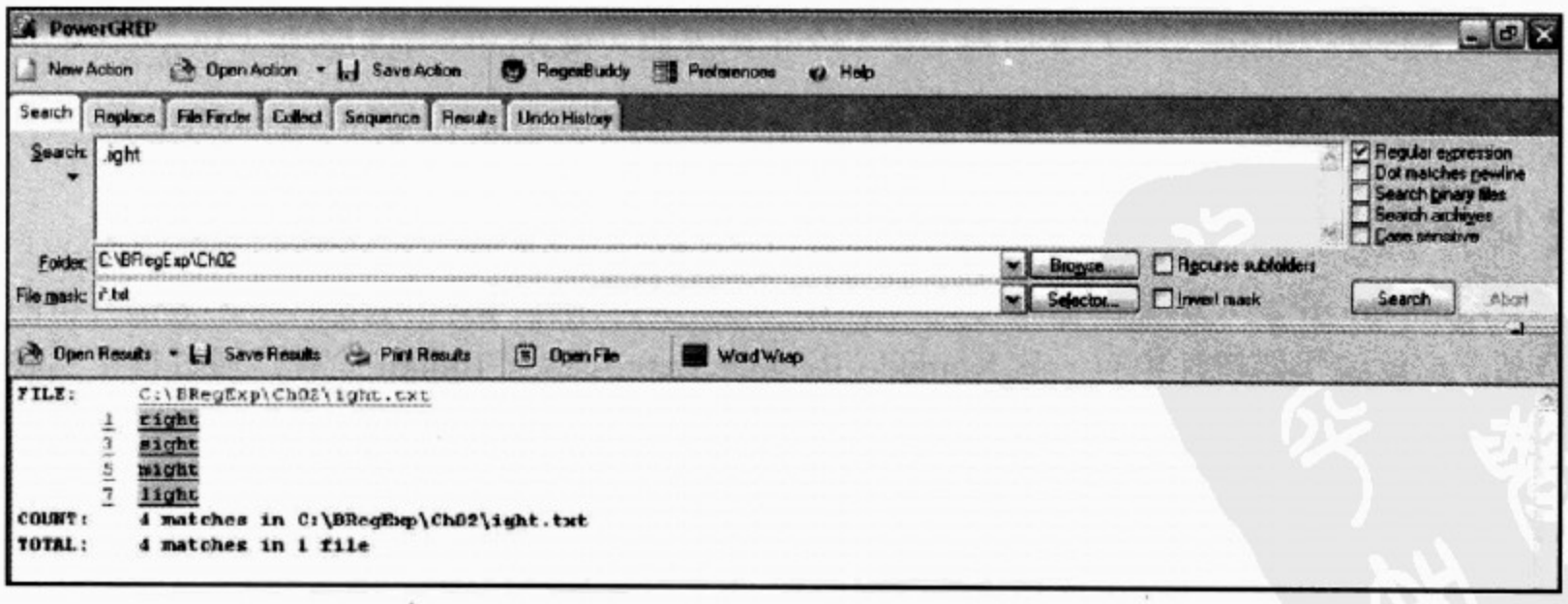


图 2-8

2.2　基于语言和平台的工具

以下几节内容会概要地介绍一下 Windows 平台中的一些编程语言和脚本语言对正则表达式的支持情况。

2.2.1　JavaScript 和 JScript

JavaScript 是 Netscape 公司推出的最早的客户端脚本语言。从官方角度来说，它正式的名称应该是 ECMAScript，因为 JavaScript 是经由欧洲计算机制造协会(European Computer Manufacturers Association，ECMA)认可的一个官方标准。在对正则表达式的支持方面，JavaScript 1.5 比 JavaScript 1.2 更加完善。

JavaScript 中对正则表达式的支持是通过 RegExp 和 String 对象来实现的。

关于在 JavaScript 中使用正则表达式的内容将在第 19 章中作详细介绍。

2.2.2　VBScript

VBScript(Visual Basic Scripting Edition)，与 JScript 中对正则表达式的支持类似。但在 VBScript 中底层的对象模型是不一样的。但仍可以在很多情况下使用类似于 JScript 中的一些技术。

关于在 VBScript 中使用正则表达式的内容将在第 20 章中作详细介绍。

2.2.3　Visual Basic.NET

Visual Basic.NET 对正则表达式的支持利用了.NET 框架中 System.Text.Regular Expressions 名称空间的类。

关于在 Visual Basic.NET 中使用正则表达式的内容将在第 21 章作中详细介绍。

2.2.4　C#

C#是一门新语言，与 Java 类似，C#是 Microsoft 作为其.NET 平台的组件发布的。与 VB.NET 类似，C#大量使用了.NET 基类库中 System.Text.RegularExpressions 名称空间中的类来实现其多数功能。这个类库实现了对正则表达式的广泛支持。

关于在 C# 中使用正则表达式的内容将在第 22 章中作详细介绍。

2.2.5　PHP

PHP (PHP Hypertext Processor)被广泛应用于 Web 服务器端，以处理网页表单提交的数据。PHP 对正则表达式也具有广泛的支持，且很大程度上也是规范的。

关于在 PHP 中使用正则表达式的内容将在第 23 章中作详细介绍。

2.2.6 Java

Java 是由 Sun Microsystem 公司开发的一门功能强大、面向对象的编程语言。与 Microsoft 公司的 C#语言类似，Java 也具有 C 语言的血统。

Java 语言中对正则表达式的支持主要体现在 java.util.regex 包中，它是在 Java 1.4 版本中引入的。

关于在 Java 中使用正则表达式的内容将在第 25 章中作详细介绍。

2.2.7 Perl

Perl 作为一种文本操作语言，第一次真正地实现了部分正则表达式。任何需要处理文本的编程语言几乎能要使用正则表达式功能。

Perl 对正则表达式支持的范围非常广，要实现同一目的可有多种方法。由于 Perl 代码和正则表达式的语法都非常简洁并具有隐秘性，所以对于新手来说，在 Perl 中初次使用正则表达式时可能会产生畏惧心理。然而，其强大的功能值得花力气好好掌握。

关于在 Perl 中使用正则表达式的内容将在第 26 章中作详细介绍。

2.2.8 MySQL

MySQL 是可以在 Windows、Unix 和 Linux 平台中使用的一种关系型数据库管理系统。

MySQL 支持 SQL 的 LIKE 关键字。另外，它还支持 REGEXP 关键字，这个关键字为应用正则表达式提供了更多的选择。

关于在 MySQL 中使用正则表达式的内容将在第 17 章中介绍。

2.2.9 SQL Server 2000

在本书写作时(2004 至 2005 年上半年期间。译者注)，Microsoft SQL Server 2000 是 Microsoft 公司的旗舰数据库管理系统。而 SQL Server 2005 也预定在本书出版后的几个月内发布(Microsoft SQL Server 2005 于 2005 年 11 月 7 日发布。译者注)。

有关在 SQL Server 中使用正则表达式的内容将在第 16 章中介绍。

2.2.10 W3C XML Schema

W3C XML Schema 是一种语言，它详细地说明了可以在 XML 文档中使用的类的结构。通过 W3C XML Schema 的 xs:pattern 元素，可以使用正则表达式来限制 XML 文档中可用的元素或属性的值。

关于在 W3C XML Schema 中使用正则表达式的内容将在第 24 章中介绍。

2.3　使用正则表达式的分析方法

本节将详细解释一种对思考、设计以及维护正则表达式非常有帮助的分析方法。

这种方法不能机械地去应用。如果只想创建一个简单的正则表达式，那么使用这里介绍的完整方法明显没有必要。在其他情况下，由于某种语言或工具可能不支持我们想要的文档化手段，也不能应用建议的所有部分。除了一些相当简单的正则表达式之外，如果能够利用下面介绍的分析方法，对创建能忠实地实现你的意图的正则表达式是非常有好处的。而且，也会使将来维护那些正则表达式变得非常容易。

我们会把这个分析方法分成几个部分来解释。首先，来看一下这种分析方法的构成部分：

- 用自然语言来表达和说明你的意图。
- 考虑数据源及其可能的内容。
- 考虑可以使用的正则表达式选项。
- 考虑灵敏度和特殊性。
- 创建适当的正则表达式。
- 对除了简单的正则表达式之外的正则表达式给予说明。
- 使用空白区域保持正则表达式说明的清晰。
- 测试正则表达式的结果。

如果只想在 Microsoft Word 中使用一个简单的正则表达式来完成搜索和替换的操作，那么没有必要使用这么详细的方法。但是，随着要完成操作的复杂性不断增加，这种系统性方法将会变得越来越有价值。

下面，我们就分别了解一下这个建议方法的各个组成部分。

2.3.1　用自然语言来表达和说明你的意图

在人类设计的任何规划工作中，对需要解决的问题进行超出一般认知的、相当正式的表达都是极其重要的。这同样适用于正则表达式。

同规划和开发一个应用程序需要反复几次一样，在提炼应用程序的设计细节时，同样也需要几次反复的过程才能得到符合要求的恰当描述。

在第 1 章中的 Star Training Company 示例中，一名新员工在初次定义自己的需求时可能会使用这样的描述：

> 用 Moon 替换所有的 Star。

在第 1 章中已经介绍过，在这种简单描述的指引下，模式 Star 会以不区分大小写的形式对文档进行“地毯式”替换操作，结果文档会更加混乱。这种简单思路会导致许多不想要的匹配。

出现不适当的替换操作的原因可以归结为对要匹配什么没有给出足够精确的定义。结果是使用了一个低特殊性的正则表达式——这个问题将在第 9 章再详细讨论。

所以，关键是要对问题给出更精确的说明。因此，第二次对需求的描述可能会如下所示：

用 Moon 替换首字母为大写 S 的 Star。

这种定义会从文件 StarOriginal.doc 中去掉一些不需要的匹配项，如 starfish 和 stare。定义这个问题的另一种可能的描述是：

用 Moon 替换本身是一个单词的 Star。

这是对初始问题定义的一种改进，它排除了像 startling 或 stare 这样不需要的匹配项。组合前面两种定义可获得对这个问题更加精确的描述：

用 Moon 替换本身是一个单词且首字母为大写 S 的 Star。

甚至还可以把问题定义得更精确一些，比如：

用 Moon 替换本身是一个单词且首字母为大写 S 的，并且后跟一个空格字符和单词 Training(首字母为大写 T)的 Star。

但如此精确的定义可能会导致错过希望得到的匹配项。例如，在文件 StarOriginal.doc 中查找下面句子中的匹配项。

```
Start now with Star.
```

以及

```
Think Star.
```

上面两句中的 Star 都应该匹配但没匹配。

现在定义得太精确了，会导致因为符合这个定义的正则表达式而丢失一些想要的匹配项。此时，这种正则表达式的灵敏度并非想要的。

有关正则表达式模式灵敏度的问题将在第 9 章中详细讨论。

对这种灵敏度失当的解决方案是稍微放宽对问题的定义，比如：

用 Moon 替换本身是一个单词且首字母为大写 S 的，并且后跟一个句点字符的或者后跟一个空格字符和单词 Training(首字母为大写 T)的 Star。

以上对问题的定义，当正确地把它转换成正则表达式模式后，将会相当精确地匹配 Star，而且也会排除前面提到的许多不想要的匹配项。第 9 章将会就如何平衡特殊性和灵敏度的问题进行专门讨论。另外，通过第 3~8 章学习到的正则表达式的各方面知识，能把多种版本的问题定义转换成可以根据例子文本进行测试的正则表达式模式。

注意，前面对问题定义不断修正的依据正是对所要使用数据源及其可能内容的理解。

2.3.2 数据源及其可能的内容

在第 1 章的 Star Training Company 示例中，我们已经看到因为缺乏对数据源的考虑，使用过份简单的搜索和替换，导致文档中出现许多问题。比如说，在搜索和替换操作完成后，把句子中不应该替换的 startling 也替换成了一个不存在的单词 Moontling。导致这种

结果的一个重要原因就是缺乏对要应用正则表达式的数据源的认真考量。

例如，对于以逗号分隔的结构化数据文件，你会发现它的结构差异很小。另一方面，如果是把按单词处理的文档作为数据源，就必须认真考虑诸如词与词之间的构成差别这样的问题了。

根据模式构造方式的不同，一个匹配字符序列 ball 的正则表达式可能会匹配其复数形式 balls，或所有格形式 ball's。同样，这个正则表达式也可能会匹配相关的单词，像 balloon、balloons 和 ballooned。

此外，你可能会希望匹配一些类似单词中的一部分，而不匹配其他部分。在现实中，还必须要考虑另外一个问题，即是否有必要花时间去匹配那些拼写错误的单词。

在验证用户通过网页表单输入的数据时，还需要考虑用户输入数据时的方式。比如说，用户在输入 16 位的信用卡号码时会不会带空格？在这种情况及其他情况下，考虑到用户输入的所有合理的变种，并通过代码将输入的数据转换为想要的类型是非常有意义的。

2.3.3　可用的正则表达式选项

在第 1 章中介绍过，用户或者开发人员并非能够在每一种语言或应用程序中都使用同一组正则表达式工具。因此，当遇到一个适合使用正则表达式来解决的问题时，就需要仔细考虑一下当前可用的正则表达式工具都有哪些。

一种使用正则表达式时会受到限制的情况是，在 Microsoft Word 中使用正则表达式时会受到限制。而 OpenOffice.org Writer 在对正则表达式的支持方面则较少限制。此外，程序员的编辑器——ActiveState Komodo、Microsoft Visual Studio 和 Microsoft Visual Basic 应用程序编辑软件都对正则表达式有不同程度的支持。

要仔细区分程序员的编辑器对正则表达式的支持和程序员的编辑器可能支持的语言对正则表达式的支持。

你可能经常会碰到被确定用于某个项目开发的语言，选择该语言可能与该语言支持的正则表达式功能无关。在这种情况下，需要在项目中充分利用该语言的正则表达式功能。在其他一些情况下，你的选择也可能会更灵活一些。例如，使用不同于项目开发所用的主编程语言的另一种语言来处理输入文件是可能的，也是合理的。

如果你的任务是处理静态文本文件，那么很可能不会被同一项目的其他人使用的语言所约束。但要注意换行符(有时候称为行终结符)之间的差别，因为在不同的操作系统中换行符是不同的。如果你的正则表达式模式依赖于某种特定类型的换行符，而你需要使用来自不同操作系统的文本文件，那么很可能会遇上想不到也不希望看到的结果。

2.3.4　灵敏度和特殊性

有关灵敏度和特殊性的概念将在第 9 章中详细解释。在这里先简单介绍一下。

假设使用一个简单的正则表达式来查找一个单词，比如说 ball，那么下面的模式一定会匹配：

```
.*
```

因为这个正则表达式的意思是“匹配零个或多个字母及数字式字符”，它会匹配任何字符序列。从它能够匹配任何情况下出现的 ball 的角度来说，它具有 100% 的灵敏度。但是，其特殊性却为 0%——因为它同样还会匹配一大堆不想要的单词。

从另外一个角度上说，假设希望匹配单词 ball 的所有形式，包括复数形式的 balls 和所有格形式的 ball's。此时，如果它处于只匹配完整单词的环境下，使用下面的正则表达式则只会匹配单数形式的 ball。

```
ball
```

或者，也有可能会匹配 balls 和 ball's 的前四个字母，但这可能不是你想要的结果。

为了更深入地讨论这个话题，需要理解更多的正则表达式语法，以便能够对多种可能的方案进行尝试，并且展示你在设计自己的正则表达式模式时所做选择的效果。

2.3.5 创建适当的正则表达式

当你对自己的需求进行了认真的分析，并且在对数据源进行研究的基础上充分理解了数据内容的特点后，就能创建符合你的需求的正则表达式模式了。不存在能够适合任何情形的魔术公式。只有开发人员自己能够决定自己想匹配什么，不想匹配什么。为了得到想要的结果，可能需要把文本操作分成两步来完成。但是，通过正则表达式结构的组合，一般都可以在一步之内完成匹配或替换操作。

2.3.6 对除简单正则表达式之外的正则表达式给予说明

如果创建的正则表达式不是简单的模式，那么建议认真地考虑是否需要为这个正则表达式加上说明。为什么要加说明呢？想一想如果在 6 个月或 12 个月后，由于用户反映你的代码出现了问题，当你查看自己创建的正则表达式时却无法解释它的精确意图，那将是一种什么情形。正是因为这种情况，所谓正则表达式难以破译的陈词滥调越发变得真实起来。不过，此时如果有清晰且完整的说明显然能够避免浪费时间，并有效减轻自己的挫败感。

本书后面针对具体语言的几章中会进一步介绍如何对正则表达式代码进行说明。有些语言(例如 Perl)可以允许通过指定一种模式来包含有关正则表达式的内部注释。以这种方式来说明正则表达式模式的每一个组件(正则表达式模式是由不同的组成部分构成的，本书把构成正则表达式的不同组成部分简称为组件。译者注)，能够使理解原始开发人员的意图变得更容易，从而在其手段和分析中发现不足，或者对其进行适当修改以适应变化的商业需求。

当以交互的方式使用正则表达式时(比如在 Microsoft Word 或 OpenOffice.org Writer 中)，说明正则表达式的意义并不大。一方面，在文字处理程序中使用正则表达式(通配符)时，用的只是一些简单的正则表达式；另一方面，文字处理程序并不会为说明正则表达式提供任何标准的途径。

在创建更复杂的正则表达式时，建议考虑说明正则表达式的三方面内容：

- 希望用正则表达式做什么？
- 想选择什么匹配？
- 不想选择什么匹配？

所要查找的问题定义越复杂，所创建的正则表达式也就越复杂，同时也就更需要花时间对以上这几个方面给出说明。

下面就分别讨论一下这三个方面的内容。

1. 说明希望用正则表达式做什么

当创建一个具有中、高度复杂性的正则表达式时，说明这个正则表达式的目的极其有用。当然，随着编写和解释正则表达式的经验和能力的提升，你所理解的高级或者复杂性的标准也会发生变化。许多正则表达式用户在几周或者几个月后，对某个正则表达式及其设计目的的直观感受能力会直线下降。为了把这种理解力的下降限制在最小程度内，我们宁肯说明得过了头，也不能敷衍了事。

如何来说明一个正则表达式呢？我们再回到 Star Training Company 示例中。这取决于在进行问题定义时反复改进的次数，通常也需要对包含在代码中的文档注释进行推敲和改进。

假设对于这个项目而言，你使用的是 Visual Basic .NET。那么首次说明可能会这样写：

```
'用 Moon 替换 Star
```

乍一看，这个说明很直观。然而，在第 1 章中我们已经看到了，如果没有对这个初始的想法进行改进，那么它描述的方法会把文档搞得更糟。

另一种定义应该做什么的说明可能会像下面这样：

```
'如果 Star 本身是一个词，用 Moon 替换它
'如果 Star 是某个词的一部分，那么保持不变
```

如果对问题的定义反复修改几次，并且在反复过程的早期就编写了文档注释，那么一定要确保在对正则表达式模式进行修改的同时对这些文档注释进行更新。假如忘记更新这些文档注释，就会导致文档注释无法反映你的思想变化。这种情况下有文档注释还不如没有呢。

在更正式或者更复杂的情况下，也可以考虑建立详细描述这些正则表达式的纸制文档，并将该文档作为项目文档的一部分。

2. 说明你想匹配什么

要尽可能恰如其分地表达出你想要匹配什么字符模式。养成规范地表达这种想法的习惯，全面地理解你真正想要的是什么。

正则表达式的作用是匹配某些字符序列，因此最好能明确地列举出想要匹配哪个(些)字符序列。

继续前面的 Star Training Company 示例，此时可能会把下面这些注释添加到代码中：

```
'当 Star 作为 Star Training Company 的一部分时匹配 Star
'当 Star 独立存在但表示 Star Training Company 时匹配 Star
'比如像在 “Star is the best” 这个短语中这样
'匹配任何所有格的 Star's
```

在明确地说明了你的目标是什么之后，就可以更好地创建与你的意图严格相符的正则表达式模式了。

3. 说明你不想选择什么

这一部分可能是说明内容中最不符合常规的部分，因为按照定义，你不想匹配的文本不应该是你感兴趣的地方。但当错误地匹配并修改了不想选择的文本时，可能会导致“moontling”式的结果。在 Star Training Company 示例中，由于搜索和替换不够精确，导致 starling 不当地被字符序列 Moontling 所替换。

如果 Star Training Company 的新员工肯花时间列举他不想修改的单词(比如下面那些)，那么搜索和替换的结果就不会如此之坏：

```
'不匹配像 start、startling 等这样的单词
```

对数据源和要考虑创建的正则表达式模式理解得越充分，在注释中添加的说明也会越具体明确。

3. 使用空白区域保持正则表达式说明的清晰

在某些语言中(如 Perl)，可以把正则表达式分开写在几行中。这样就可以巧妙利用空白区域来区分正则表达式中每个组件的注释内容，使注释内容更加清晰。由于正则表达式的每个组件都有自己独立的注释，因而可以有效地消除或避免多义性。

在其他语言(如 JavaScript)中，则不能使用空白区域来分隔注释。比如在 JavaScript 中，一条语句必须要写在一行中。在像 JavaScript 这样的语言中编写复杂的正则表达式时，建议直接在正则表达式模式的下方放置以分行形式编写的该正则表达式模式的一个副本作为注释。

就眼前而言，这会增加你的工作量，但实际上它会节省你的时间——因为这些注释能够保证你始终明白自己的意图。它还有助于你或者其他修改这些代码的开发人员更全面地理解最初的目标。

通过添加这种风格的注释，当模式被分开放到几行中时，你将会从其详细的说明中受益良多。而不同语言之间的关键区别在于：在像 Perl 和 Java 这样的语言中，可以在正则表达式中把注释作为其模式的内在部分穿插其中。而在 JavaScript 中，要把注释添加到正则表达式代码副本的各个组件中，才能让 JavaScript 解释器把它们当作注释。

当然，这样做会存在使工作中的正则表达式副本与构成文档说明的副本之间出现差异的风险。

2.3.7 测试正则表达式的结果

当要操作一个单独的、内容简单的文档时，除了人机交互之外并不需要测试正则表达式。出于对页面空间的现实考虑，本书中的多数例子都使用了短小、简化的文档。由于使用正则表达式的目的不同，在使用本书中的这些例子时，可能经常会发现能够完成正则表达式的交互式匹配(即当测试本书中的例子时，通常都能在屏幕中实时地看到将正则表达式应用到全部数据时的效果如何。译者注)。这种交互式匹配只是一种简单的测试方式。要想

知道你的模式是否只选择想要的匹配，必须通过实际的应用才能看到结果。

但是，交互式的方法也存在一定的局限。如果要对几十个、上百个，甚至数万个文档或者包含多达数兆字节信息的文档应用正则表达式，但又不希望像第 1 章中的 Star Training Company 例子那样制造混乱，则此时仍然使用交互式的测试，范围就会比较好。

因此，在一些适当的测试数据基础上对一个复杂的正则表达式进行测试具有重要意义。这可以确保发现所有想要的字符序列，以及确保没有无意地修改了不想修改的字符序列。

本书中提供的这些简短的例子文档，可能会对你创建测试文档有一些启发。但如果只是简单地从本书的例子文档中复制，则很可能没什么帮助。你必须认真考虑自己想要选择的数据和要确保不会选择的类似数据。只有在对要处理的实际数据和要实现怎样的修改进行认真思考的基础上，才有可能创建出真正有用的测试文档。

如果能按照这里介绍的如何处理非一般的正则表达式的步骤去做，就能够创建出相关且有效的测试文档。而且，由于对希望完成的文本操作任务有了充分理解，也就能够对问题做出适当的定义并将其转换成能够满足需求的、正确的正则表达式模式。通过在测试数据的基础上仔细测试正则表达式，会使你对成功完成大规模的文本化数据操作充满信心。

如果要处理的数据量很大，记住要在操作该数据之前进行备份。如果对文本操作过程规划得非常周密，那么一切都应该顺利完成，也就不会用到备份数据。但是，如果由于执行了不恰当的正则表达式操作，导致了数兆字节的数据中出现了不希望发生的问题，此时这些备份就会成为你的“后悔药”。

建立备份是一回事。而所建立的备份能够用于数据恢复则是另外一回事。要完全确保备份有效的唯一方法就是要对它们的可读性以及能够用于恢复原先的配置进行测试。这种测试应该作为对重要数据进行例行质量保证的一种程序。当真正需要这些备份数据来从(无论是因为错误的正则表达式还是其他原因导致的)某些灾难中拯救时才发现它们不能用，那就太迟了。

第 3 章

简单的正则表达式

本章讨论构造简单正则表达式的一些基本内容。讨论这些简单正则表达式是为了巩固我们在第 2 章中介绍的方法，以及让读者实际体验一下如何在简单的正则表达式中应用这种系统的方法。

本章中使用的例子虽然简单，但通过使用正则表达式来匹配简单的文本模式，你将会逐渐熟悉基本的正则表达式用法，从而为掌握更加复杂的正则表达式打下基础。本章之后的几章，将对正则表达式的其他重要构成部分进行讨论，并逐步过渡到解决更复杂的问题。

本章要详细讨论的另一个问题是匹配字符发生的位置，而不是简单地判定是否存在匹配项。

在本章中将学习以下内容：

- 如何匹配单个字符
- 如何匹配可选字符
- 如何匹配大量存在的字符，而不管该字符是否为可选的
- 如何匹配出现特定次数的字符

首先，我们讨论最简单的情况——如何匹配单个字符。

3.1 匹配单个字符

最简单的正则表达式包括匹配单个字符。如果要匹配一个特定的字母字符或者数字，那么只需使用由相应的字符或者数字组成的模式即可。举例来说，如果要匹配大写字母 L，应该使用下面的模式：

```
L
```

这个模式匹配任何大写的 L。我们没有给这个模式添加任何限制条件，目的就是想让它匹配任何大写的 L。当然，如果匹配是按照不区分大小写的方式(详见第 4 章中)进行的，那么大小写的 L 都将被匹配。

试一试：匹配单个字符

这个模式可以应用到示例文档 UpperL.txt 中，其内容如下：

```
Excel had XLM macros. They were replaced by Visual Basic for Applications in later
versions of the spreadsheet software.

CMLIII

Leoni could swim like a fish.

Legal difficulties plagued the Clinton administration. Lewinski was the source of
some of the former president's difficulties.
```

(1) 打开 OpenOffice.org Writer，并打开 UpperL.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框，并在 Options 部分选中 Regular expressions 复选框和 Match case 复选框。

(3) 在 Find & Replace 对话框顶部的 Search For 文本框中输入正则表达式模式 L，并单击 Find All 按钮。

如果一切顺利，文本中的每个大写 L 都会被突出显示出来。

图 3-1 显示的是在 OpenOffice.org Writer 中基于例子文档 UpperL.txt 对模式 L 匹配的结果。注意，有五个匹配项包含在字符序列 XLM、CMLIII、Leoni、Legal 和 Lewinski 中。

工作原理

OpenOffice.org Writer 默认执行不区分大小写的匹配。如图 3-1 所示，Match case 复选框被选中，这样，只有符合正则表达式中指定的大小写形式的字符才会被匹配。

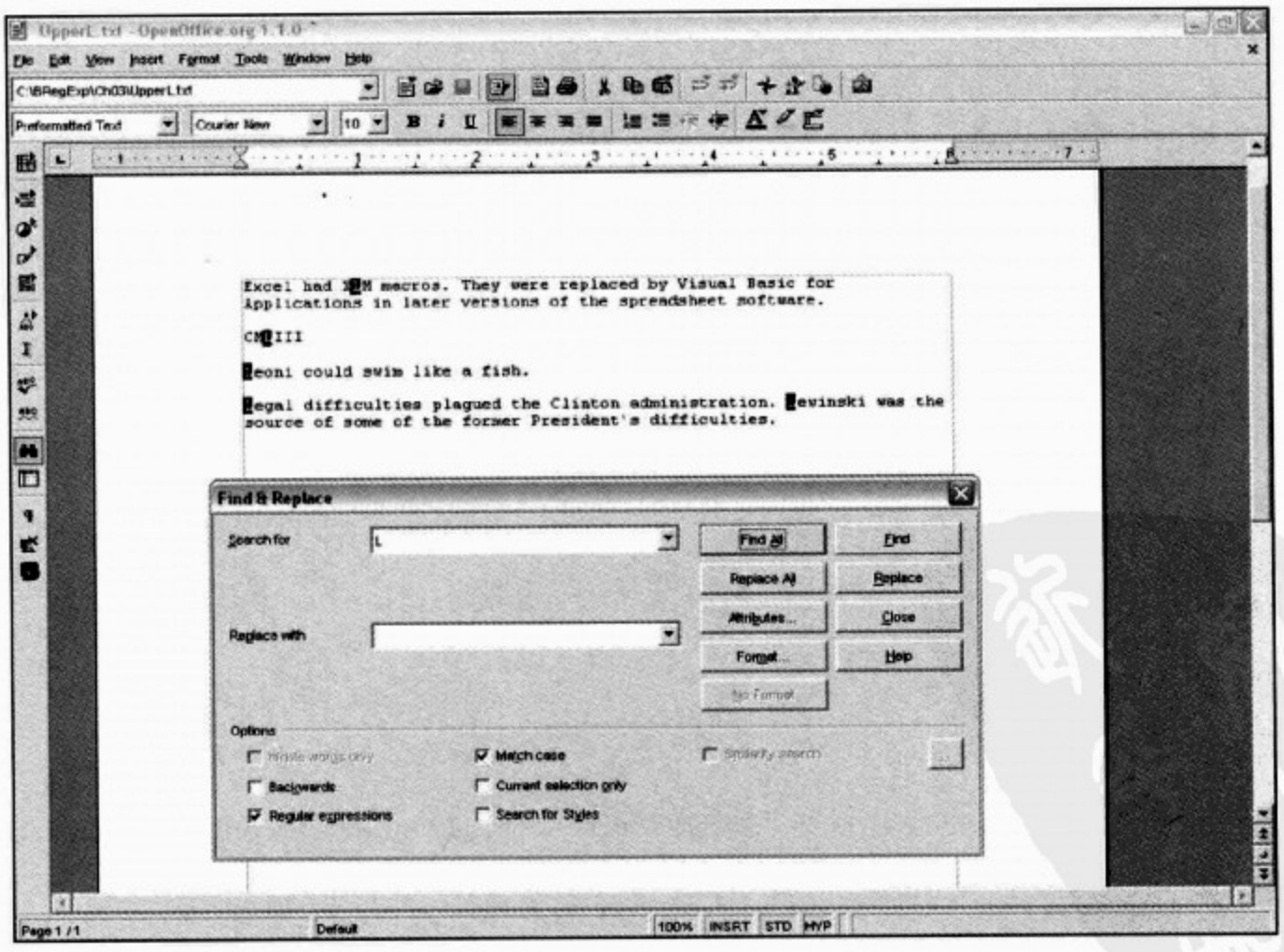


图　3-1

对于这个文档中的每一个字符，OpenOffice.org Writer 都会检查它是不是大写的 L。如果是，表示它与正则表达式模式匹配。在 OpenOffice.org Writer 中，如果单击 Find All 按钮，那么每个匹配项中的字符(这里只有一个字符)都会被突出显示出来。

那么，如何使用 JavaScript 来匹配单个字符呢？

试一试：在 JavaScript 中匹配单个字符

查找所有大写的 L，可以把这个正则表达式要完成的任务表示为：

匹配任意大写的 L。

在这里，以使用 JavaScript 为例，可以看到多数正则表达式引擎是如何在 XHTML 文件 UpperL.html 中使用模式 L 来完成匹配的，UpperL.html 的内容如下：

```
<html>
<head>
<title>Check for Upper Case L</title>
<script language="javascript" type="text/javascript">
var myRegExp = /L/;

function Validate(entry){
return myRegExp.test(entry);
} // end function Validate()

function ShowPrompt(){
var entry = prompt("This script tests for matches for the regular expression
pattern: " + myRegExp + ".\nType in a string and click on the OK button.", "Type
your text here.");
if (Validate(entry)){
alert("There is a match!\nThe regular expression pattern is: " + myRegExp + ".\n
The string that you entered was: '" + entry + "'.");
} // end if
else{
 alert("There is no match in the string you entered.\n" + "The regular expression
pattern is " + myRegExp + "\n" + "You entered the string: '" + entry + "'." );
} // end else

} // end function ShowPrompt()

</script>
</head>
<body>
<form name="myForm">
<br />
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
</form>
</body>
</html>
```

(1) 在 Windows 资源管理器中找到包含 UpperL.html 的目录，并双击该文件。它会在默认的浏览器中打开。

(2) 单击标签为 Click here to enter text 的按钮，会出现一个提示窗口，如图 3-2 所示。

(3) 在其文本框(包含默认文本“Type your text here”)中输入一个字符或字符串，JavaScript 会寻找是否存在与正则表达式模式 L 匹配的字符串。单击 OK 按钮。

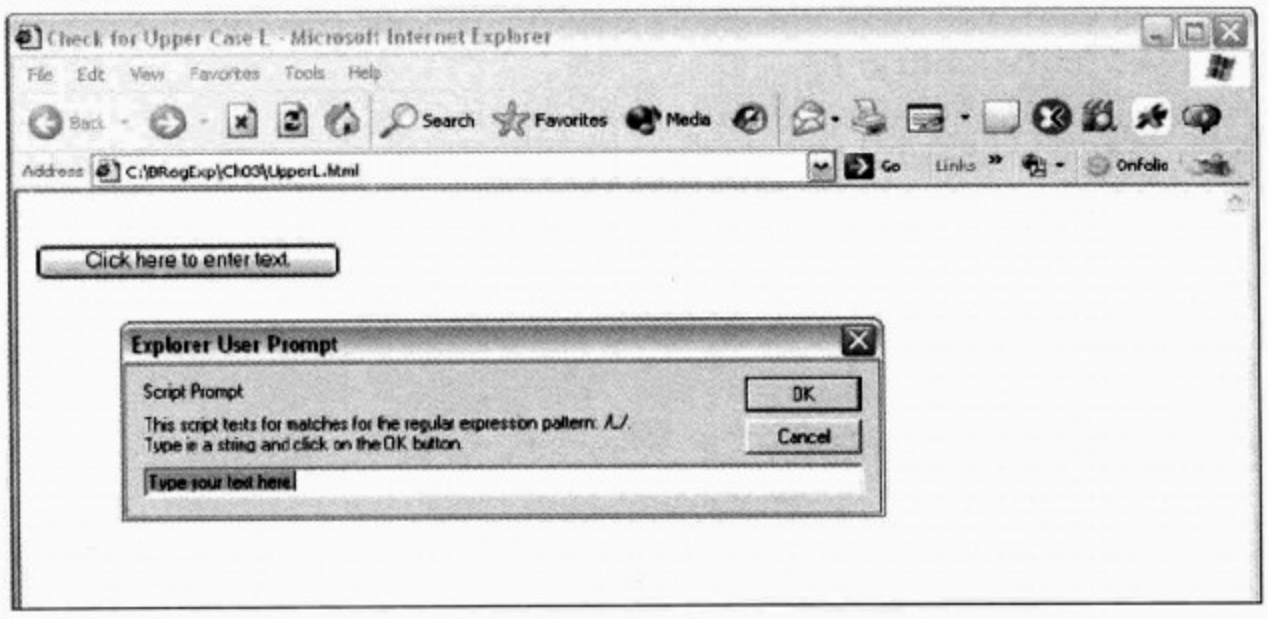


图　3-2

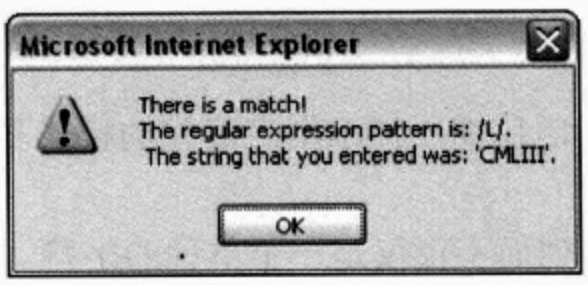


图　3-3

图　3-4

工作原理

这个简单的网页文件中包含着 JavaScript 代码。

其中，JavaScript 变量 myRegExp 的值是正则表达式模式的直接量 L，下面是赋值语句：

```
var myRegExp = /L/;
```

在 JavaScript 中，正斜杠用于界定正则表达式模式——就像使用一对引号来界定字符串一样。还有另一种关于界定的语法，在这里就不讨论了。

单击 Click here to enter text 按钮时，会调用 ShowPrompt 函数。

变量 entry 用于收集你在提示框中输入的字符串：

```
var entry = prompt("This script tests for matches for the regular expression
pattern: " + myRegExp + ".\nType in a string and click on the OK button.", "Type
your text here.");
```

最终的输出结果取决于输入的文本是否包含与正则表达式模式匹配的内容。当输入完文本并单击 OK 按钮后，会执行一个 if 语句，该语句检查输入的文本(保存在变量 entry 中)是否存在与包含在变量 myRegExp 中的正则表达式模式匹配的内容：

```
if (Validate(entry)){
```

在 if 语句中会调用 Validate 函数：

```
function Validate(entry){
return myRegExp.test(entry);
} // end function Validate()
```

变量 myRegExp 在 test 函数下用于判断是否存在匹配项。

如果 if 语句：

```
if (Validate(entry))
```

返回布尔值 true，则执行以下代码：

```
alert("There is a match!\nThe regular expression pattern is: " + myRegExp + ".\n
The string that you entered was: '" + entry + "'.");
```

以上代码使用变量 myRegExp 和 entry 来显示正则表达式模式输入的字符串，以及相应的说明性文本。

如果不存在匹配项，会执行以下代码——它们包含在 if 语句的 else 子句中：

```
alert("There is no match in the string you entered.\n" + "The regular expression
pattern is " + myRegExp + "\n" + "You entered the string: '" + entry + "'." );
```

同样，变量 myRegExp 和 entry 用于对用户给出反馈，提示用户匹配的模式和输入的字符串。

当然，在实践中我们通常都会匹配一个字符序列而不仅仅是一个单独的字符。

3.1.1 匹配连续的字符序列

在正则表达式模式 L 存在匹配项时，我们利用的是正则表达式引擎默认的模式，也就是说不会指出字符(或字符序列)出现的次数。在这种情况下，正则表达式引擎假设模式中的字符(或字符序列)恰好出现一次——除非在正则表达式模式中设定一个限定符来指定次数。这种方法也适用于相同字符序列的匹配。

在不使用插入字符的情况下匹配两个出现两次的相同字符(包括空白符)，可以在模式中把要匹配的字符重复两次。

试一试：匹配双字符

在这个例子中，我们来看一看如何匹配字符正好(连续)出现两次的情况——例如，两个 r 可能会出现在单词 arrow 和 narrative 中。

对这个问题的定义可以表达如下：

匹配任意小写字符 r 后面直接跟另一个小写 r 的情况。

示例文件 DoubledR.txt 的内容如下：

```
The arrow flew through the air at great speed.

This is a narrative of great interest to many readers.

Apples and oranges are both types of fruit.

Asses and donkeys are both four-legged mammals.

Several million barrels of oil are produced daily.
```

下面的模式会匹配示例文件中所有 rr：

```
rr
```

(1) 打开 OpenOffice.org Writer，并打开示例文件 DoubledR.txt。

(2) 使用快捷键 Ctrl+F 打开 Find and Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中输入模式 rr，然后单击 Find All 按钮。

图 3-5 显示的是按照前面的步骤在 OpenOffice.org Writer 中打开的 DoubledR.txt 文件的结果。注意，其中所有的 rr 都被匹配了，但是单个 r 没有被匹配。

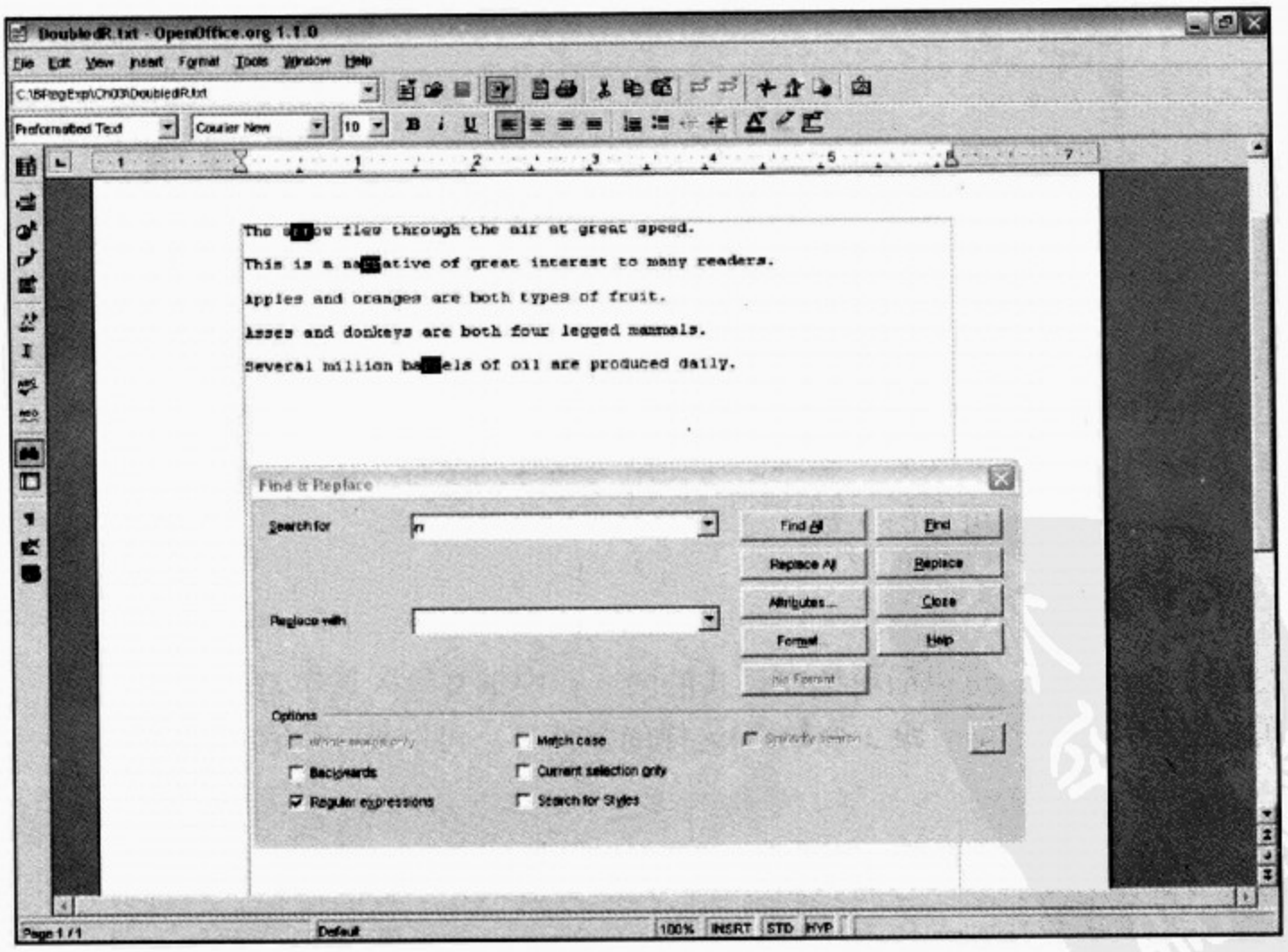


图　3-5

工作原理

模式 rr 告诉正则表达式引擎应匹配小写的字符 r，如果第 1 次匹配成功，那么再尝试匹配下一个字符。如果第 2 个字符也是小写的 r，那么表示全部匹配成功。

如果在匹配第 1 个字符时失败了，会测试下一个字符是不是小写的 r。如果不是小写的 r，匹配失败。然后，重新根据正则表达式模式中的第 1 个 r 来开始下一轮匹配。

如图 3-6 所示，可以在 Komodo Regular Expression Toolkit 中完成这一操作。在这个例子中，要匹配的是连续的小写字母 mm。可以从 http://activestate.com/Products/Komodo 上下载 Regular Expression Toolkit 的 Komodo IDE 的最新测试版。本章中使用的是 Komodo IDE v 2.5。清除 Komodo Toolkit 中的正则表达式和测试文本。在内容区中输入 mammals 作为要匹配的字符串，在正则表达式区域中输入 m。此时，mammals 的第 1 个字母会被匹配。然后继续在正则表达式区域中输入第 2 个 m，突出显示的匹配项表明 mammals 中间的 mm 被匹配了，如图 3-6 所示。

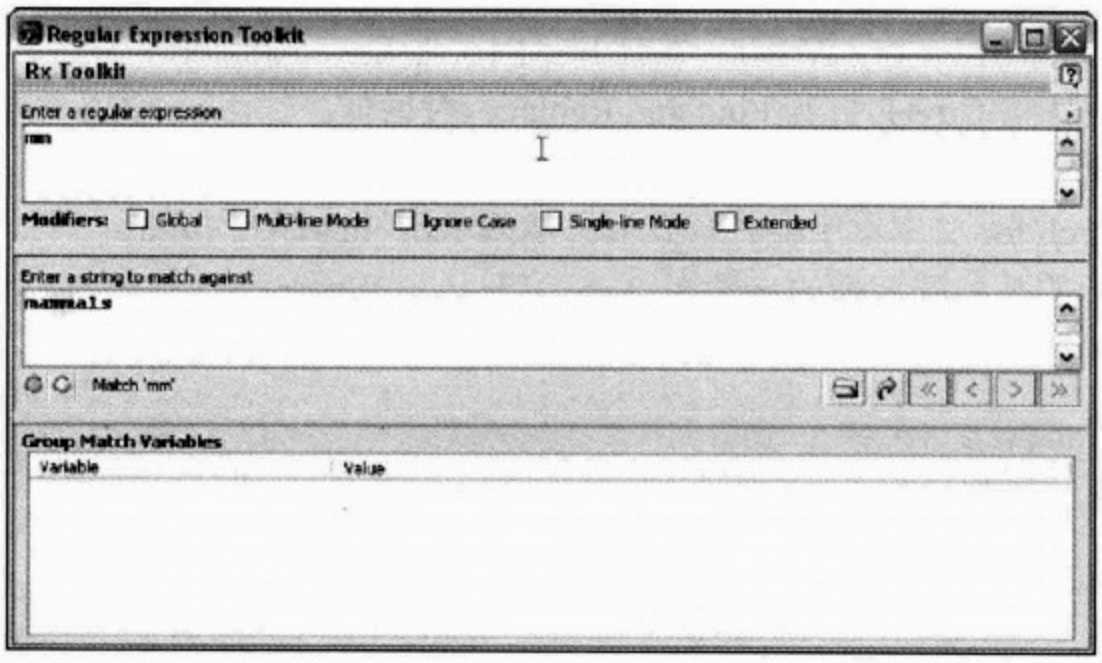


图 3-6

通过这两个例子，我们说明了使用一种语法规则来实现匹配双字符的过程。本章的后面，还会介绍严格匹配连续出现的同一字符的次数的另外一种语法，它可以匹配同一字符连续出现两次甚至更多的次数。那种语法使用大括号元字符，除了能够限定严格的匹配次数外，还可以限定匹配可变的次数。

3.1.2 元字符简介

要匹配三个字符，可以简单地在模式的同一行中重复输入该字符 3 次。例如，要匹配像 ABC123(3 个字母字符后跟 3 个数字)这样的零件号，可以使用下面的模式：

```
AAA
```

要匹配这种形式的零件号的其他部分，则需要引入元字符的概念。到目前为止，你所看到模式中包含的都是不变的直接量字符，它们的含义就是匹配字符本身。而元字符可能是一个单独的字符，也可能是一对字符(前一个字符通常是反斜杠)，并且具有不同于其直

接量字符的含义。

可以通过几种不同的方式来匹配 ABC123 这种类型的零件号中的 123 部分。一种方式是：

```
\d\d\d
```

其中每一个 \d 元字符都表示数字 0~9(包含 0 和 9)。\d 是一个元字符，它不表示反斜杠后跟一小写字母 d。

\d 元字符的含义与我们到目前为止在模式中用到的直接量字符的含义明显不同。字符 L 在模式中只能匹配大写的 L，而元字符 \d 却可以匹配任何数字：0、1、2、3、4、5、6、7、8 或 9。

一个元字符通常会匹配一类字符。这里，元字符 \d 匹配的就是数字的字符类。

当你看到模式 \d\d\d 时，就知道它匹配 3 个连续的数字，即它会匹配 012、234、345、999 以及其他任意三位数字。

试一试：匹配三个数字

若要匹配 3 个数字的序列。用纯粹的自然语言来表达：匹配一个三位数。而用稍微正式一点的说法是：匹配一个数字，如果是数字，那么尝试匹配下一个字符；如果前两个字符都是数字，则尝试匹配第 3 个连续的数字。元字符\d 只匹配一个数字，因此，如前所述可以使用以下模式：

```
\d\d\d
```

来匹配三个连续的数字。如果三次匹配都成功，则说明找到了一个与正则表达式模式匹配的匹配项。

这里要使用的测试文件是 ABC123.txt，其内容如下：

```
ABC123

A234BC

A23BCD4

Part Number DRC22

Part Number XFA221

Part Number RRG417
```

我们使用前面的模式 \d\d\d 来只匹配数字。

在这个例子中，我们使用 JavaScript 来示范，原因稍后解释。

(1) 找到包含文件 ABC123.txt 和 ThreeDigits.html 的目录。在 Web 浏览器中打开 ThreeDigits.html。

(2) 单击 Click here to enter text 按钮。

(3) 在打开的提示框中，输入一个要测试的字符串。这里将从 ABC123.txt 中复制的字

符串粘贴过来。

(4) 单击 OK 按钮并检查警告框，检查输入的字符串中是否包含与模式 \d\d\d 匹配的内容。

图 3-7 显示的是输入字符串 Part Number RRG417 之后的结果。

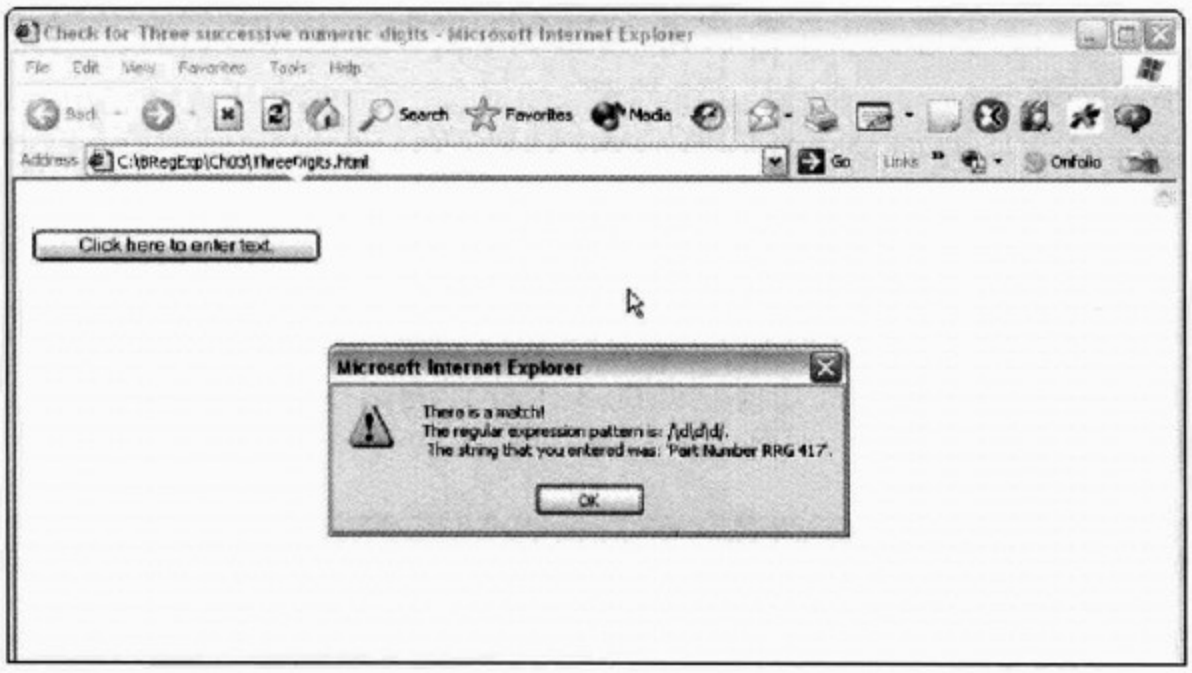


图 3-7

可以通过 ABC123.txt 中的字符串来做试验，也可以使用自己的字符串来测试。模式 \d\d\d 始终都会匹配三位连续的数字序列，但一位或两位数字不会被匹配。

工作原理

正则表达式引擎会根据模式查找一个数字。如果它测试的第 1 个字符不是一个数字，就会在被测试字符串中向前移动一个字符并测试该字符是否是一个数字。如果不是，则会继续上面的步骤。

如果找到了与模式中第 1 个\d 匹配的项，正则表达式引擎会测试下一个字符是否也是一个数字。如果第 2 个\d 也匹配，那么会继续测试下一个字符。如果 3 个连续的字符都是数字，那么正则表达式模式 \d\d\d 就取得了一个匹配项。

可以通过 Komodo Regular Expression Toolkit 来实际地体会一下匹配的过程。打开 Komodo Regular Expression Toolkit，清除当前存在的任何正则表达式和测试字符串，输入字符串 A234BC。然后，在正则表达式模式区域中，输入模式 \d。第 1 个数字 2 会作为匹配项突出显示出来。在正则表达式区域中再添加第 2 个\d，又会看到 23 作为匹配项被突出显示出来。最后，添加第 3 个 \d 完成整个模式 \d\d\d，此时会看到 234 作为匹配项被突出显示，如图 3-8 所示。

读者可以使用 ABC123.txt 中的其他文本来测试这个模式。建议读者使用自己的包含数字的测试文本。在 Komodo Regular Expression Toolkit 中，需要在输入的字符串末尾加一个空格才能保证匹配正常进行。

现在来回答刚刚提到的问题，为什么我们要用 JavaScript 示范前面的例子呢？因为 OpenOffice.org Writer 不能用来测试 \d 元字符。

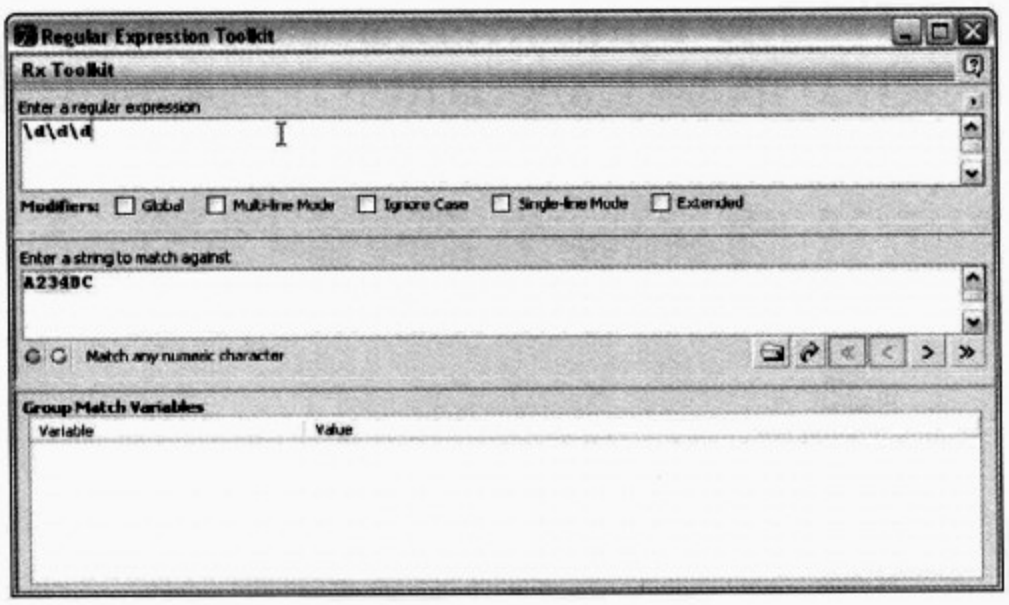


图　3-8

在 OpenOffice.org Writer 中匹配数字可能导致一些问题。图 3-9 显示了当使用 OpenOffice.org Writer 打开 ABC123.txt 并搜索模式 \d\d\d 后的结果。

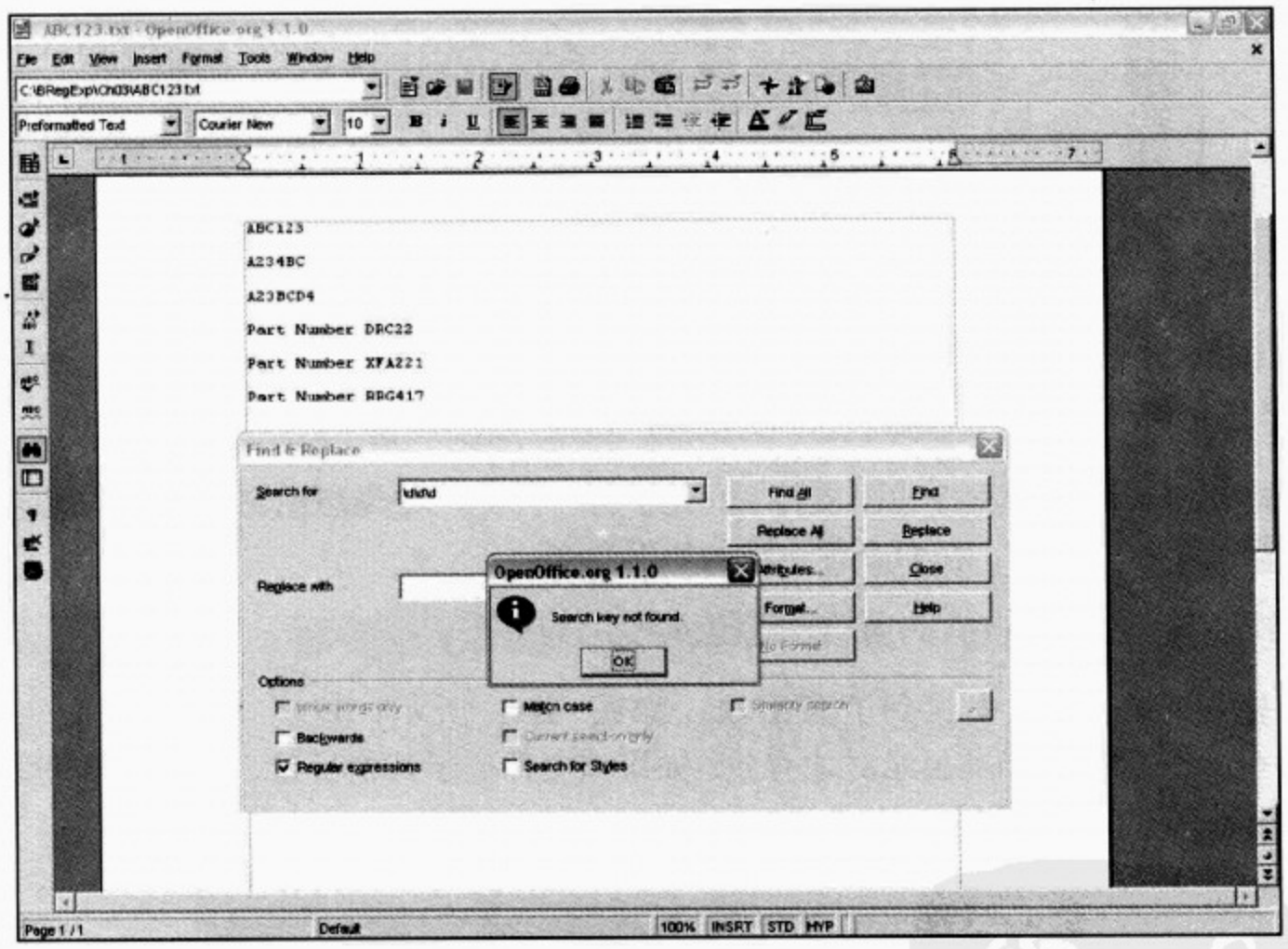


图　3-9

如图 3-9 所示，OpenOffice.org Writer 没有找到匹配项。由于 OpenOffice.org Writer 缺少对\d 元字符的支持，所以在 OpenOffice.org Writer 中要搜索数字需要使用非标准的语法。

可以使用字符类解决 OpenOffice.org Writer 中的这一问题，有关字符类我们将在第 5 章中介绍。现在，只要知道能够使用以下模式就可以了：

```
[0-9][0-9][0-9]
```

匹配结果与使用模式 \d\d\d 是一样的，因为[0-9][0-9][0-9] 与 \d\d\d 的含义完全相同。使用上面的字符类构成的模式匹配 ABC123.txt 中 3 个连续数字的结果如图 3-10 所示。

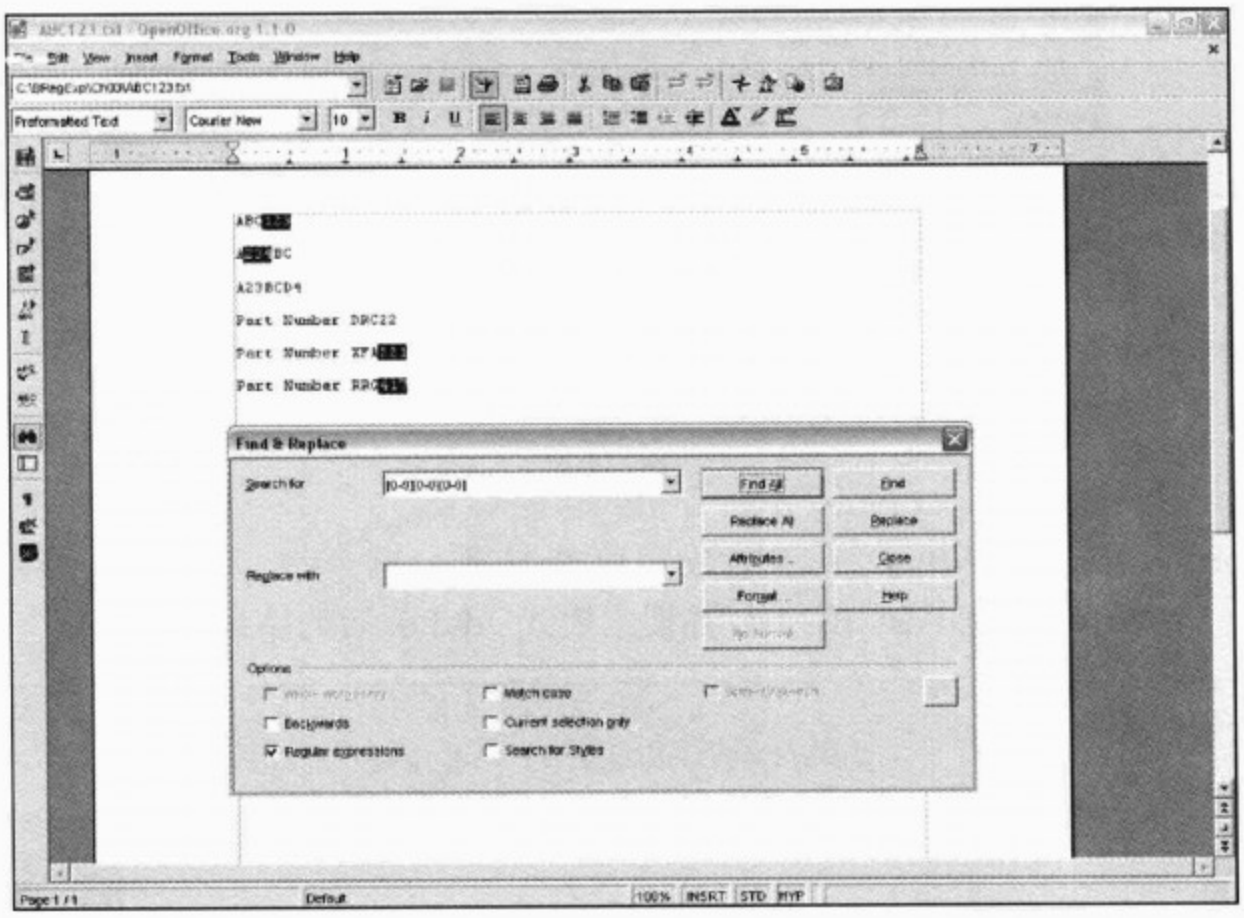


图 3-10

在 OpenOffice.org Writer 中，还可以使用另一种语法，即 POSIX 元字符，相关内容在第 12 章中介绍。

同样，findstr 实用程序也不支持 \d 元字符。因此，如果想用它来查找匹配项，必须使用前面的字符类方式，在命令行中输入如下命令：

```
findstr /N [0-9][0-9][0-9] ABC123.txt
```

结果显示在 4 行中找到了匹配项，如图 3-11 所示。注意，前面的命令行只有 ABC123.txt 文件位于当前目录时才有效。如果该文件位于其他目录，则必须要在命令行中写明该文件所在的路径。

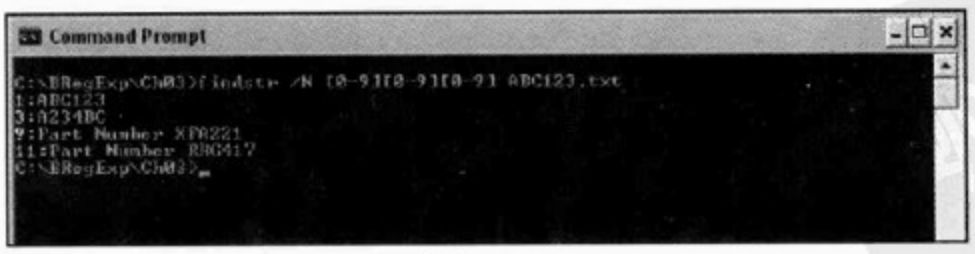


图 3-11

下一节将综合运用到目前为止所学到的技术，去查找直接量字符与字符序列的组合。

3.1.3　匹配不同的字符序列

常见的使用简单正则表达式实现的任务是查找直接指定的单个字符及一个字符序列。

在这种情况下，可以使用的测试文本有无数种。我们现在只来看一个简单的零件号列表，并查找编码 DOR 后跟 3 个连续数字的情况。这时，正则表达式要完成的操作如下：

查找大写 D 的匹配项。如果找到该匹配项，检查下一个字符是否与大写的 O 匹配。如果匹配，接着检查是否后面的字符匹配大写的 R。如果这 3 个匹配项都存在，检查后面的 3 个字符是否是连续的 3 个数字。

试一试：查找直接量字符和字符序列

这里使用测试文件 PartNumbers.txt，其内容如下：

```
BEF123

RRG417

DOR234

DOR123

CCG991
```

首先，在 OpenOffice.org Writer 中试验，记住要用正则表达式模式 [0-9] 代替 \d。

(1) 在 OpenOffice.org Writer 中打开 PartNumbers.txt，按 Ctrl+F 组合键打开 Find and Replace 对话框。

(2) 选中 Regular expressions 和 Match case 复选框。

(3) 在 Search for 文本框中输入模式 DOR[0-9][0-9][0-9]，然后单击 Find All 按钮。

文本 DOR234 和 DOR123 随即被突出显示，表示它们与正则表达式匹配。

工作原理

正则表达式引擎首先查找直接量字符、大写字符 D。文本中的每个字符都会依次被检测，以确定有没有匹配项。

如果找到了匹配项，正则表达式引擎就会查看下一个字符，判断它是否是一个大写的 O。如果这个字符也匹配，它会检查第 3 个字符是不是大写的 R。如果这 3 个字符都匹配，该引擎会接着检查第 4 个字符是否是一个数字。如果是，会继续检查第 5 个字符是不是数字。如果还是，则会检查第 6 个字符是不是数字。如果这些也都匹配，则说明整个正则表达式模式获得了匹配项。在 OpenOffice.org Writer 中，每个匹配项都以突出显示的字符序列表示。

也可以在命令行中使用 findstr 实用程序，通过下面的模式检查 PartNumbers.txt 文件中包含相应匹配项的行：

```
DOR[0-9][0-9][0-9]
```

在命令行中输入以下命令：

```
findstr /N DOR[0-9][0-9][0-9] PartNumbers.txt
```

如图 3-12 所示，包含字符序列 DOR234 和 DOR123 的文本行被匹配了。与上面的例子相同，如果包含 PartNumbers.txt 的目录不是命令窗口中的当前目录，则需要相应地调整文本的路径。

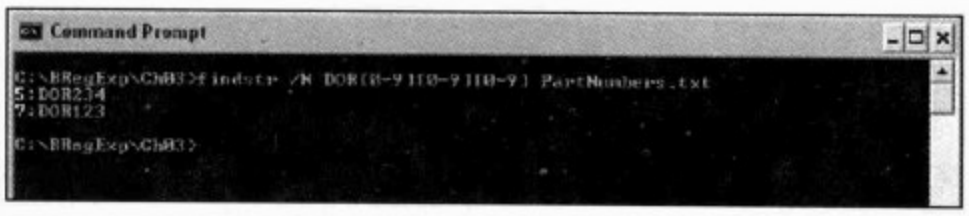


图 3-12

也可以使用 Komodo Regular Expression Toolkit 来测试模式 DOR\d\d\d，如图 3-13 所示。

在学习了如何匹配只出现一次的字符序列后，我们接着来看一看匹配一个字符序列可能会出现可变次数的情况。

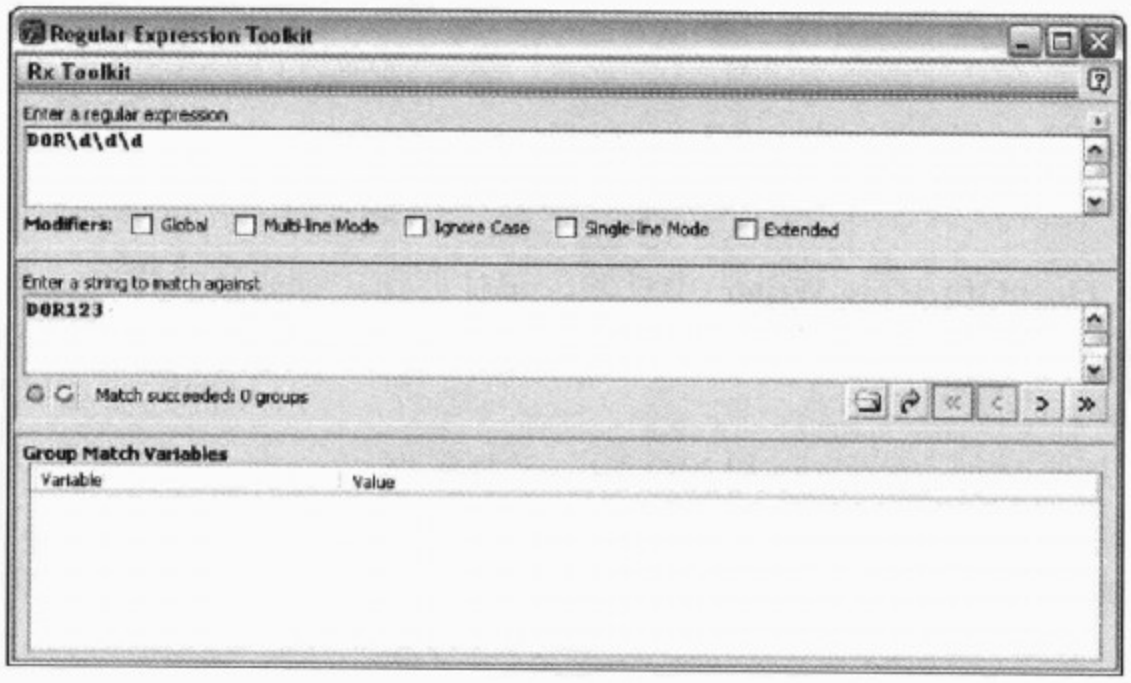


图 3-13

3.2 匹配可选字符

匹配直接量字符是非常直观的，特别是当你只想查找页与正则表达式模式中包含的直接量字符匹配一次的字符时。那么，在此基础上我们可以想象一下如果要匹配单个直接量字符出现零次或一次该怎样做。换句话说便是怎样匹配一个可选的字符。对此，多数正则表达式实现都使用问号(?)字符来表示前面的字符块是可选的。在这里使用通用的术语“块”来表示位于问号之前的模式。“块”可能是单个或多个(不同的)字符，也可能是更复杂的正则表达式结构。目前，我们要处理的只是单个、可选的字符。而其他更复杂的正则表达式结构，例如组，会在第 7 章中介绍。

假设要处理既包含美式英语又包含英式英语的一组文档。

你可能会发现单词 color(美式英语)在某些文档中被写做 colour(英式英语)。那么，可以通过下面的模式来同时表示这两个单词：

```
colou?r
```

接下来要把文档中所有单词都修改为美式英语的拼法。

试一试：匹配可选的字符

我们使用 Komodo Regular Expression Toolkit 来完成这个试验。

(1) 打开 Komodo Regular Expression Toolkit，清除原来残留的任何正则表达式或文本。

(2) 在内容区中插入 colour 作为要匹配的文本。

(3) 在正则表达式模式区域中输入 colou?r。

结果如图 3-14 所示。

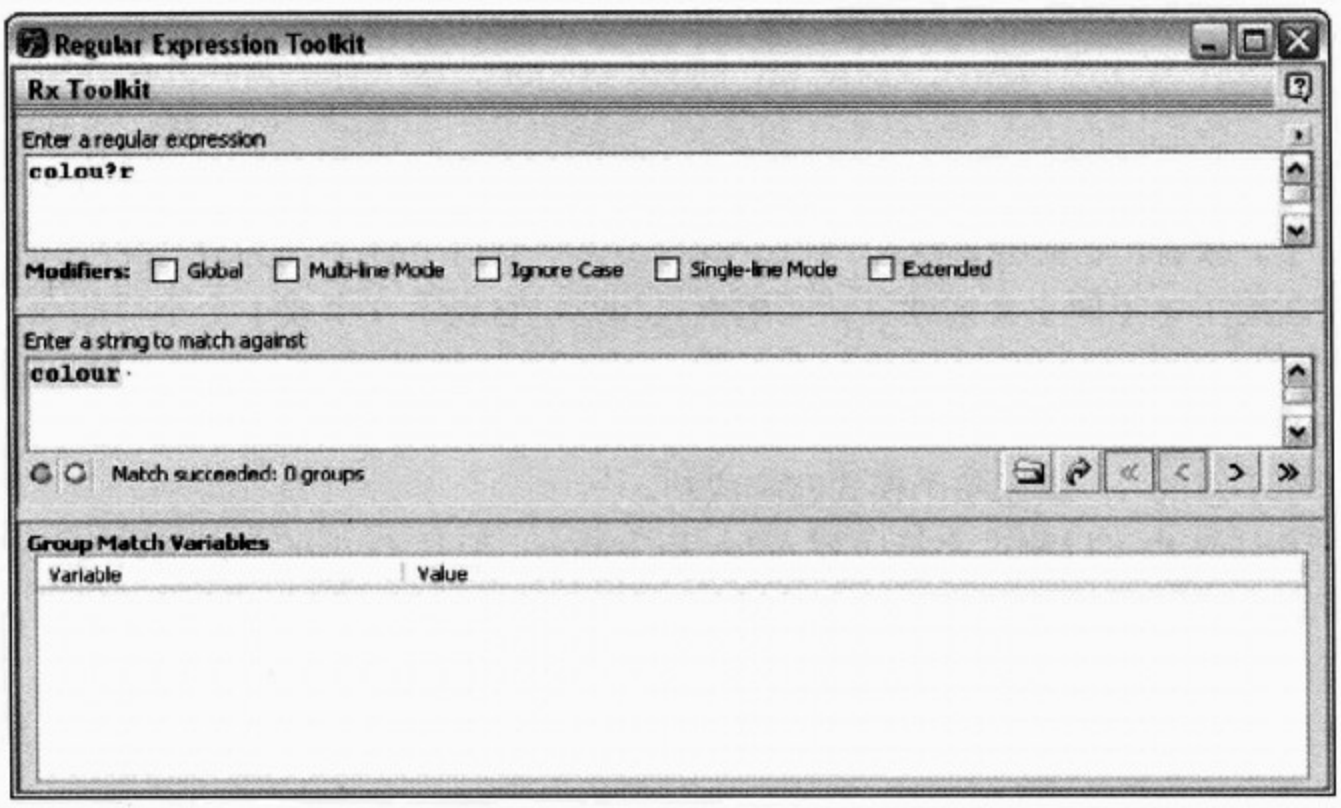


图　3-14

再试验一下对测试文件 Colors.txt(内容如下)应用这个正则表达式模式的结果：

```
Red is a color.

His collar is too tight or too colouuuurful.

These are bright colours.

These are bright colors.

Calorific is a scientific term.

"Your life is very colorful," she said.
```

工作原理

在 Red is a color. 这一行中的单词 color 会与模式 colou?r 匹配。

当正则表达式引擎恰好搜索到 color 中的 c 之前的位置时，它会尝试匹配小写的 c。这个匹配成功后，它会尝试匹配小写的 o，这次也成功了。然后，它又尝试依次匹配小写的 l 和 o，同样也成功了。此时，正则表达式引擎会尝试匹配模式 u?，它的含义是匹配 0 个或 1 个小写的 u。因为在小写的 o 后面实际上没有小写的 u(即零个小写的 u)，所以匹配成功。最后，引擎尝试匹配模式中最后一个字符——小写的 r。由于 color 字符串的下一个字符就是一个小写的 r，因此 color 最终与整个模式匹配。

但是，在 His collar is too tight or too colouuuurful. 一行中没有找到匹配项。该行中唯一可能出现匹配项的字符序列是 colouuuurful。而失败发生在正则表达式引擎在尝试匹配模式 u? 之后。因为模式 u? 的意思是“匹配 0 个或 1 个小写的 u”，虽然 colouuuurful 中的第一个 u 匹配。但在这次成功匹配后，正则表达式引擎会尝试将模式 colou?r 中的最后一个字符(小写的 r)与 colouuuurful 中的第二个小写的 u 进行匹配，结果自然就失败了。因此，模式 colou?r 与字符序列 colouuuurful 的匹配便失败了。

当正则表达式引擎试图在 These are bright colours. 这一行中查找匹配项时的情况如何呢？

当正则表达式引擎恰好搜索到 colours 中 c 之前的位置时，它尝试匹配一个小写的 c。匹配成功了。接着，它会尝试匹配一个小写的 o、一个小写的 l 和另外一个小写的 o。这些也成功匹配了。下面它会继续尝试匹配模式 u?，它表示 0 个或 1 个小写的 u。由于在 colours 中的确有一个小写的 u 位于小写的 o 之后，于是匹配又成功了。最后，正则表达式引擎会尝试匹配模式中最后一个字符——小写的 r。由于字符串 colours 中的下一个字符确实匹配小写的 r，所以整个模式匹配成功。

通过 findstr 也可以测试字符序列 color 和 colour，但是 findstr 中的正则表达式引擎存在表示可选字符的元字符的限制，所以使用匹配零个、一个或多个前面字符的 * 元字符才能顺利完成这个任务。

要使用 findstr 实用程序来查找包含 colour 和 color 的行，在命令行中输入以下命令：

```
findstr /N colou*r Colors.txt
```

前面的命令假设文件 Colors.txt 位于当前目录中。

图 3-15 显示了在 Colors.txt 中使用 findstr 实用程序搜索的结果。

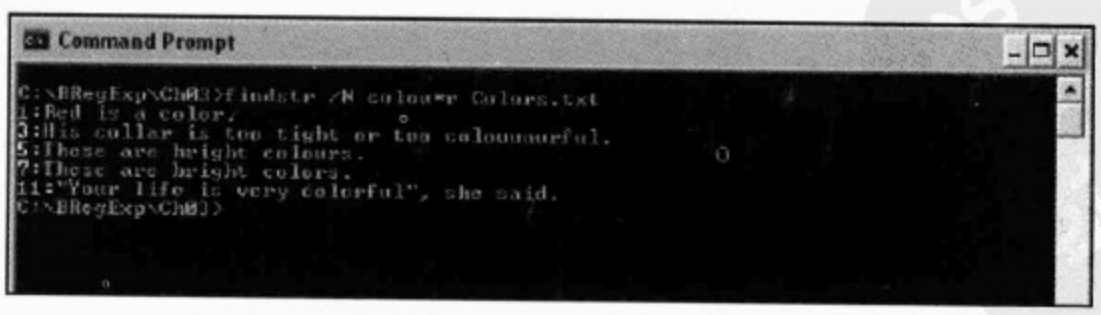


图 3-15

我们注意到，无论 color 和 colour 字符序列是作为一个单词还是作为其他更长单词的

一部分，它们所在的行都被成功匹配了。不过，也要注意那个有些奇怪的“单词” colouuuurful 也被匹配了，这是因为 * 元字符允许出现多个小写的字母 u。在多数实际的情形中，这个怪异的“单词”对你而言并不是一个问题，而 * 限定符在 findstr 实用程序中用来代替 ？限定符也比较合适。而在其他情况下，当必须要精确地匹配 0 个或 1 个特定的字符时，findstr 实用程序就无能为力了。因为它肯定还会匹配诸如 colonifier 这样的字符序列。

学习了如何在正则表达式模式中匹配单个可选字符之后，我们再来学习如何在单个正则表达式模式中使用多个可选字符。

匹配多个可选的字符

许多英语单词都有多种形式。许多情况下，必须匹配一个单词的所有形式。而要匹配所有可能的形式就必须在一个正则表达式模式中使用多个可选的字符。

我们来考虑一下单词 color(美式英语)和 colour(英式英语)的多种形式。主要包含以下这些：

```
color (U.S. English, singular noun)

colour (British English, singular noun)

colors (U.S. English, plural noun)

colours (British English, plural noun)

color's (U.S. English, possessive singular)

colour's (British English, possessive singular)

colors' (U.S. English, possessive plural)

colours' (British English, possessive plural)
```

下面包含三个可选的字符的正则表达式模式，可以用于匹配以上所有形式：

```
colou?r'?s?'?
```

如果想通过一种半正规的方式来表达这个模式，那么可以像下面这样来描述相应的问题定义：

匹配美式英语和英式英语中 color(colour)的所有形式，包括单数名词、复数名词、单数所有格和复数所有格。

我们来试一试，然后再解释为什么这个模式有效以及这种模式的内在局限性是什么。

试一试：匹配多个可选的字符

使用的测试文件是 Colors2.txt，其内容如下：

```
These colors are bright.
```

```
Some colors feel warm. Other colours feel cold.

A color's temperature can be important in creating reaction to an image.

These colours' temperatures are important in this discussion.

Red is a vivid colour.
```

按照以下步骤来测试这个正则表达式：

(1) 打开 OpenOffice.org Writer，再打开 Colors2.txt。

(2) 使用快捷键 Ctrl+F 打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中，输入正则表达式模式 colou?r'?s? '?，然后单击 Find All 按钮。如果一切顺利，应该看到如图 3-16 所示的匹配结果。

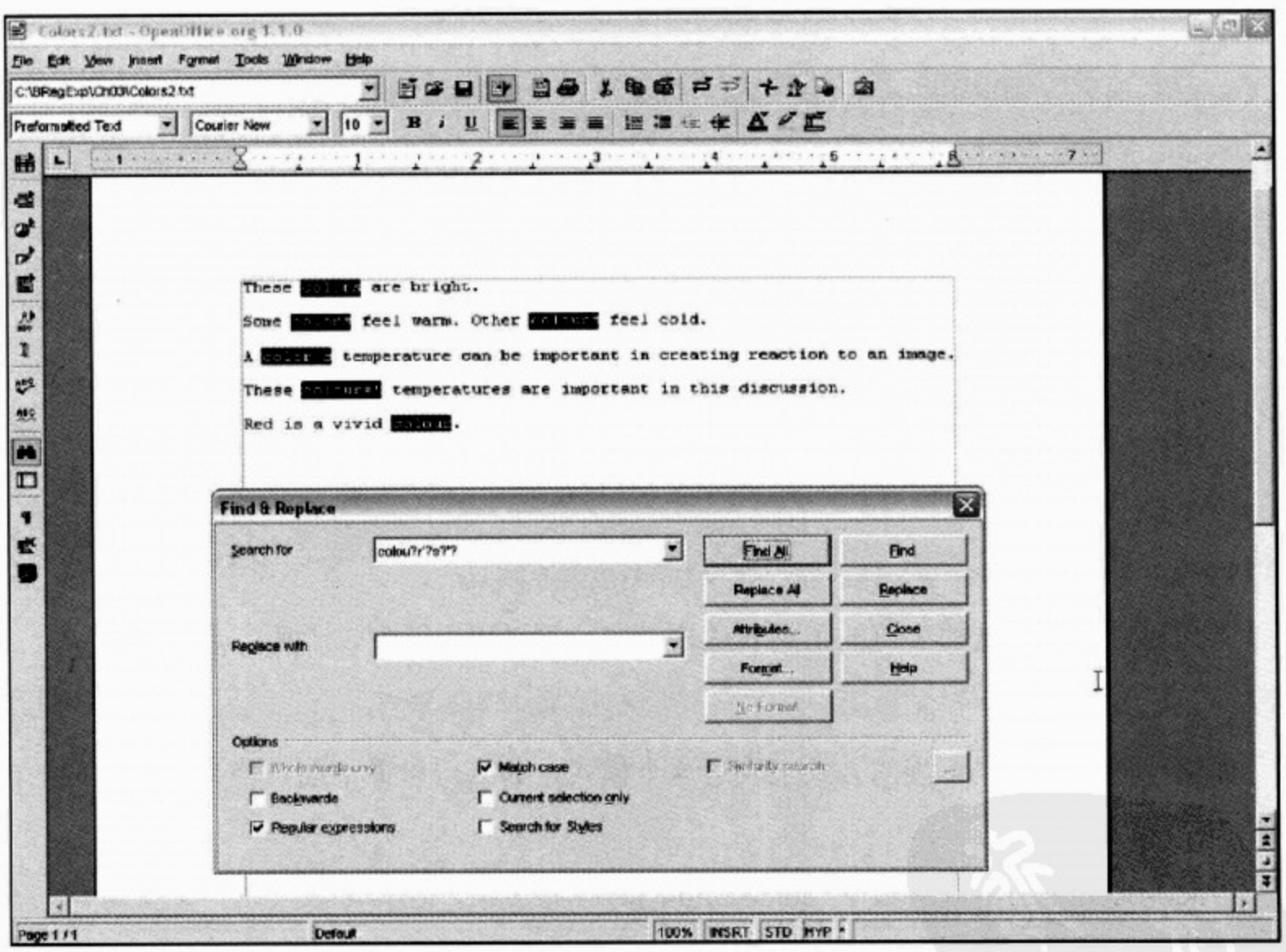


图 3-16

从图中可以看到，我们设定的所有形式都被匹配了。

工作原理

这里主要介绍单词 colour/color 形式的匹配。

模式 colou?r'?s?'?是怎么匹配 color 的呢？假设正则表达式引擎当前的位置直接位于

color 的第一个字符之前。它首先会尝试匹配小写的 c，因为小写的 c 必须要匹配。结果匹配了。进而又尝试匹配后来的小写字母 o、l 和 o。这些字符也都匹配了。然后会尝试匹配一个可选的小写字母 u(即，需要匹配 0 个或 1 个小写的 u)。由于是可选的，所以匹配也成功了。接着，会尝试匹配一个小写的 r。而 color 中的小写字母 r 也与之匹配。然后，要尝试匹配的是一个可选的撇号。这里也是可选的，所以匹配成功。接着，正则表达式引擎要匹配一个可选的小写字母 s ——即要匹配 0 个或 1 个小写的 s。如上所述，这次也匹配了。最后，尝试匹配一个可选的撇号。同理匹配成功。由于正则表达式模式的所有部分都找到了相应的匹配项，因此正则表达式模式 colour?r'?s?'? 也就获得了匹配项。

现在，我们来分析模式 colou?r'?s?'? 是如何匹配单词 colour 的。假设正则表达式引擎当前的位置直接位于 colour 的第一个字符之前。它首先尝试匹配小写的 c，因为一个小写的 c 必须要匹配。结果匹配了。进而又尝试匹配后来的小写字母 o、l 和 o。这些字符也都匹配了。然后会尝试匹配一个可选的小写字母 u(即需要匹配 0 个或 1 个小写的 u)。因为下一个字符就是小写的 u，所以这次也匹配了。接着，要尝试匹配一个小写的 r。而 colour 中的小写字母 r 也与之匹配。然后，引擎会尝试匹配一个可选的撇号。因为没有撇号，所以匹配成功。然后，正则表达式引擎要匹配一个可选的小写字母 s ——即要匹配 0 个或 1 个小写的 s。因为没有小写的 s，所以这次又匹配了。最后，是尝试匹配一个可选的撇号。因为没有撇号，所以还是匹配了。由于正则表达式模式的所有部分都找到了相应的匹配项，那么整个正则表达式模式 colour?r'?s? '? 也完全匹配。

依次分析前面列出的其他 6 种单词形式，你会发现每种单词形式事实上也都与这个正则表达式模式匹配。

虽然模式 colou?r'?s? '? 与前面列出的所有 8 种单词形式都匹配，但它会匹配下面的字符序列吗？

```
colour's'
```

你能分析出来其会匹配吗？你能说出它为什么会与这个模式匹配吗？如果这个正则表达式的三个可选字符中的任何一个都存在，那么就会与前面的字符序列匹配。这个相当古怪的字符序列可能没有出现在例子文档中，因此存在匹配失败的可能性(降低特殊性)也没有关系。

如果要避免像 colour's' 这么古怪的字符序列导致的问题该怎么办呢？你想要表达的内容可能大致如下：

匹配一个小写的 c。如果存在匹配项，尝试匹配一个小写的 o。如果这个匹配项也存在，尝试匹配一个小写的 l。如果仍有匹配项，尝试匹配一个小写的 o。如果还存在匹配项，尝试匹配一个可选的小写字母 u。如果匹配项存在，尝试匹配一个小写的 r。如果还有匹配项，尝试匹配一个可选的撇号。匹配项仍然存在的话，则尝试匹配一个可选的小写字母 s。如果前面可选的撇号不存在(匹配零次)，则再尝试匹配一个可选的撇号。

运用到目前为止所学的知识，我们还不能表达诸如“只有当前面没有一些特定的值时，才能匹配另一些项”这样的命题。而解决这个命题的方法可能会在增加复杂性的同时实现更高的特殊性。相关的问题将在第 9 章中介绍。

3.3 其他限量操作符

我们在前面“颜色”(color 和 colour)的例子中已经看到，仅匹配可选字符的测试是非常有用的。但是，对开发人员而言如果只能使用一个限定符的局限性是很大的。事实上，多数正则表达式的实现还提供了另外两个限量操作符(也称为限定符)：* 操作符和 + 操作符，下面几节将分别介绍这两个操作符。

3.3.1 * 限定符

* 操作符表示相关的模式出现零次或多次。换句话说，一个或一组字符是可选的，但也可能会出现多次。而且，* 限定符应该匹配块出现零次或一次的情况，进而，两次、三次，甚至十次都应该匹配。所以从原则上说，不论出现多少次也都会匹配。

我们使用 OpenOffice.org Writer 来举一个例子。

试一试：匹配零个或多个匹配项

测试文件 Parts.txt 中包含一组零件编号，每个编号都是由三个字母字符后跟零个或多个数字组成的。在这个简单的测试文件中，最大的数字个数是 4，由于 * 限定符会匹配发生 4 次的情况，所以我们可以用它来匹配例子中的零件编号。如果有充足的理由可以说明为什么存在最大的 4 位数字很重要，我们也可以通过另外一种语法来表达这一想法，这种语法我们将在本章后面介绍。本例中的每一个零件编号都由大写的字符序列 ABC 后跟零个或多个数字组成：

```
ABC

ABC123

ABC12

ABC889

ABC8899

ABC34
```

我们可以像下面这样表达我们的意图：

匹配一个大写的 A。如果匹配成功，尝试匹配一个大写的 B。如果匹配成功，尝试匹配一个大写的 C。如果全部三个大写字符都匹配，尝试匹配零个或多个数字。

由于所有零件编号都以直接字符 ABC 开头，所以可以用模式：

```
ABC[0-9]*
```

来匹配问题定义中所描述的零件编号。

(1) 打开 OpenOffice.org Writer，再打开测试文件 Parts.txt。

(2) 按快捷键 Ctrl+F 打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中输入正则表达式模式 ABC[0-9]*。

(5) 单击 Find All 按钮，观察突出显示的匹配结果。

图 3-17 显示了 OpenOffice.org Writer 中的匹配情况。从图中可以看出，所有零件编号都与这个模式匹配。

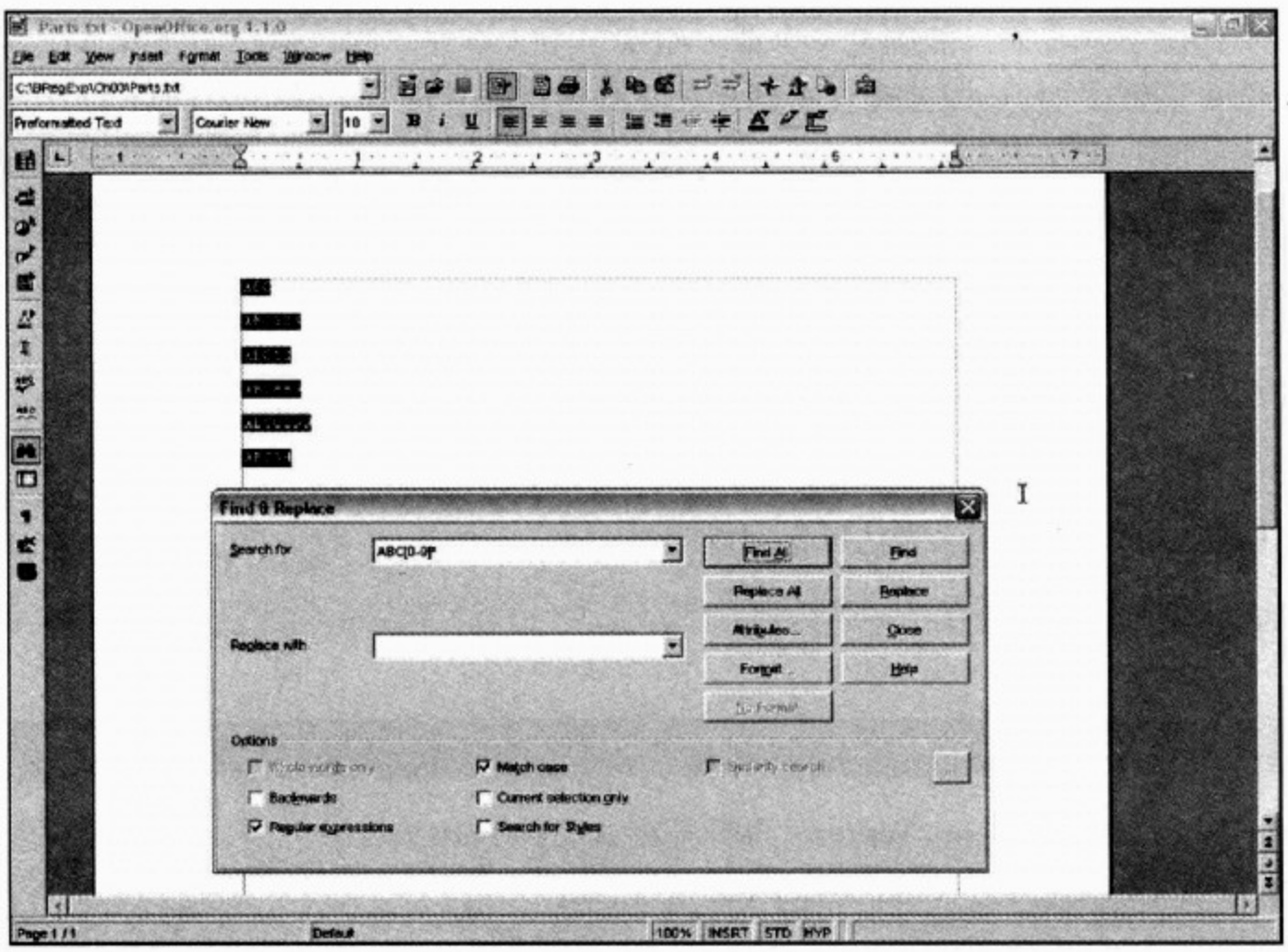


图　3-17

工作原理

在分析匹配项之前，我们先来简单地看一看正则表达式中的一部分 [0-9]*。其中当星号应用于字符类 [0-9]时，这个字符类便可以称为块。

为什么零件编号前面的 ABC 会被匹配呢？当正则表达式引擎直接位于 ABC 中的 A 之前时，它会尝试将零件编号中的第一个字符与大写的 A 匹配。因为 ABC 的第一个字符是一个大写的 A，所以匹配成功。接着，它尝试匹配一个大写的 B。这也会匹配，同样也会成功地匹配大写的 C。此时，正则表达式模式中的前三个字符都已经匹配了。最后，引擎会尝试匹配模式 [0-9]*，这个模式表示“匹配零个或多个数字”。因为字符 C 后面是一个换行符，没有数字。也就是说在 ABC 中的 C 后面有零个数字，因此匹配成功。因为该模式的所有组件都匹配，所以整个模式也匹配。

那么为什么零件编号 ABC8899 也会匹配呢？当正则表达式引擎直接位于 ABC8899 中的 A 之前时，它会尝试将零件编号中的第一个字符与大写的 A 匹配。因为 ABC8899

的第一个字符是一个大写的 A，所以匹配成功。接着，它尝试匹配一个大写的 B 和一个大写的 C。这两个字符也会匹配。这样，正则表达式模式中的前三个字符都已经匹配了。最后，引擎会尝试匹配模式 [0-9]*，这个模式表示“匹配零个或多个数字”。字符 C 后面是四位数字。因为在 ABC 中 C 的后面有四位数字，所以也会匹配成功(四个数字符合“零个或多个数字”的标准)。由于模式的所有元素都匹配，所以整个模式也匹配。

经过逐个分析其他的零件编号，我们会发现每一个都能匹配模式 ABC[0-9]*。

3.3.2 + 限定符

很多情况下，我们只能确定一个或一组字符至少存在一次，但也不排除该(组)字符出现多次的可能性。而 + 限量操作符就是为这种情况设计的。+ 操作符的含义是“匹配一次或多次前面的块”。

同样是 Parts.txt 的例子，这次我们来查找至少包含一个数字的零件编号。要求查找以大写字符 ABC 开头，然后带有一个或多个数字的零件编号。

你可以像下面这样来表达这个问题的定义：

匹配一个大写的 A。如果匹配成功，尝试匹配一个大写的 B。如果匹配成功，尝试匹配一个大写的 C。如果全部三个大写字符都匹配，尝试匹配一个或多个数字。

可以使用下面的模式来表达以上问题定义：

```
ABC[0-9]+
```

试一试：匹配一个或多个数字

(1) 打开 OpenOffice.org Writer，再打开测试文件 Parts.txt。

(2) 使用 Ctrl+F 打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中输入模式 ABC[0-9]+，单击 Find All 按钮，并观察匹配后突出显示的零件编号。如图 3-18 所示。

从图中可以看出，与使用模式 ABC[0-9]* 的结果唯一不同的是模式 ABC[0-9]+ 没有匹配零件编号 ABC。

工作原理

当正则表达式引擎直接处于零件编号 ABC 中的 A 之前时，它会尝试匹配一个大写的 A。匹配成功了。接着匹配大写的 B 和 C 的，也成功了。此时，正则表达式模式中的前三个字符都成功匹配。最后，则是尝试匹配模式 [0-9]+，该模式的含义是“匹配一个或多个数字”。但是，大写的 C 后面没有数字。由于大写的 ABC 中 C 的后面有零个数字，因此没有匹配(零个数字不符合由 + 限定符限定的“一个或多个数字”的标准)。由于该模式的最后一个组件匹配失败，所以整个模式的匹配失败。

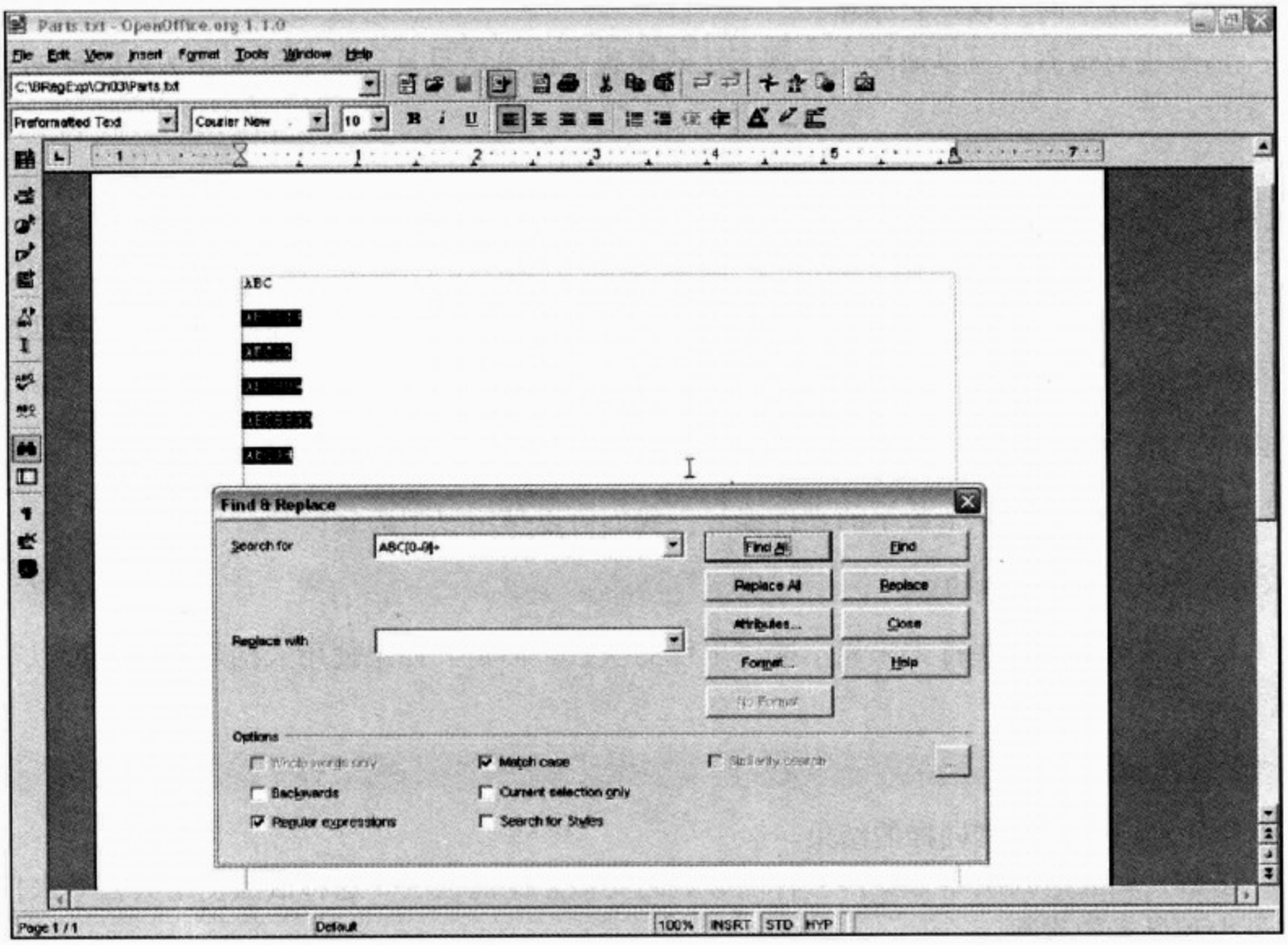


图　3-18

那么为什么零件编号 ABC8899 会匹配呢？当正则表达式引擎直接位于 ABC8899 中 A 的前面时，它会尝试将零件编号中的第一个字符与一个大写的 A 匹配。因为零件编号 ABC8899 的第一个字符是一个大写的 A，所以匹配成功。接着，它会尝试匹配一个大写的 B 和一个大写的 C，也都匹配成功。此时，正则表达式模式的前三个字符都已经匹配成功。最后，引擎尝试匹配模式 [0-9]+，该模式的含义是“匹配一个或多个数字”。而在大写的 ABC 中 C 的后面有 4 个数字，所以存在匹配(4 个数字符合“一个或多个数字”的标准)。由于模式中的所有组件都成功匹配，所以整个模式也是匹配的。

在讨论大括号限定符的语法之前，我们简单地总结一下前面介绍的几个限定符，如表 3-1 所示。

表 3-1　一些限定字符及其含义

限　定　符	含　义
?	匹配 0 或 1 次
*	匹配 0 或多次
+	匹配 1 或多次

虽然这些限定符都很有用，但如果想表达“匹配大于等于两次”或者“匹配至少 3 次

但不超过 6 次”时，该怎么办呢？

前面也有提到，可以通过在正则表达式模式中简单地重复一个字符来表达要匹配的重复字符。

3.4 大括号语法

如果想要指定的匹配次数非常多，我们可以使用一种大括号语法来精确地指定要匹配的次数。

3.4.1 {n} 语法

假设要匹配带有 3 个数字的零件编号，我们可以使用以下模式：

```
ABC[0-9][0-9][0-9]
```

即通过多次重复字符类来表示数字出现的次数。但是，如果使用大括号语法则可以写成下面这样：

```
ABC[0-9]{3}
```

以上模式也会得到同样的结果。

多数正则表达式引擎都支持一种能够表达类似意图的语法。这种语法使用大括号来指定最少和最多的次数。

3.4.2 {n,m} 语法

本章前面介绍过的 * 操作符，其实际含义是“匹配最少零次，最多无限次”。类似地，+ 限定符的含义是“匹配最少一次，最多无限次”。

使用大括号以及其中的数字可以让开发人员使用?、*和+限定符无法实现的限定次数。

下面的几个小节会分别介绍大括号语法的三种变体。首先，我们看一下通过这种语法指定“匹配零次到‘指定次数’”的用法。

3.4.3 {0,m}

{0,m} 语法可以指定最少匹配零次(由开始大括号后面的第一个数字指定)而最多匹配 m 次(由第二个数字指定，与指定最少次数的数字之间以逗号分隔，且位于结束的大括号前面)的情况。

因此，如果要指定最少匹配零次而最多匹配一次的情况，应该使用下面的模式：

```
{0,1}
```

这个模式与 ？限定符含义相同。

要指定最少匹配零次而最多匹配 3 次的情况，则可以使用这个模式：

```
{0,3}
```

而这个模式的含义是无法通过 ?、* 或 + 限定符来表达的。

假设要指定与字符序列 ABC 后跟最少零个数字或者最多两个数字匹配的情况。

可以半正式地将这一意图以下面的问题定义来表述：

匹配一个大写的 A。如果匹配成功，尝试匹配一个大写的 B。如果匹配成功，尝试匹配一个大写的 C。如果所有三个大写字符都匹配，则尝试匹配最少零个或者最多两个数字。

下面的模式可以达到这个目的：

```
ABC[0-9]{0,2}
```

其中，ABC 简单地匹配相应的直接量字符。[0-9] 表示匹配一个数字，而 {0,2} 是一个表示前面的块最少有零个(即 [0-9]，表示一个数字)而最多有两个的限定符。

试一试：匹配零个或两个匹配项

(1) 打开 OpenOffice.org Writer，再打开测试文件 Parts.txt。

(2) 使用 Ctrl+F 打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中输入正则表达式模式 ABC[0-9]{0,2}，单击 Find All 按钮，并观察突出显示的匹配结果，如图 3-19 所示。

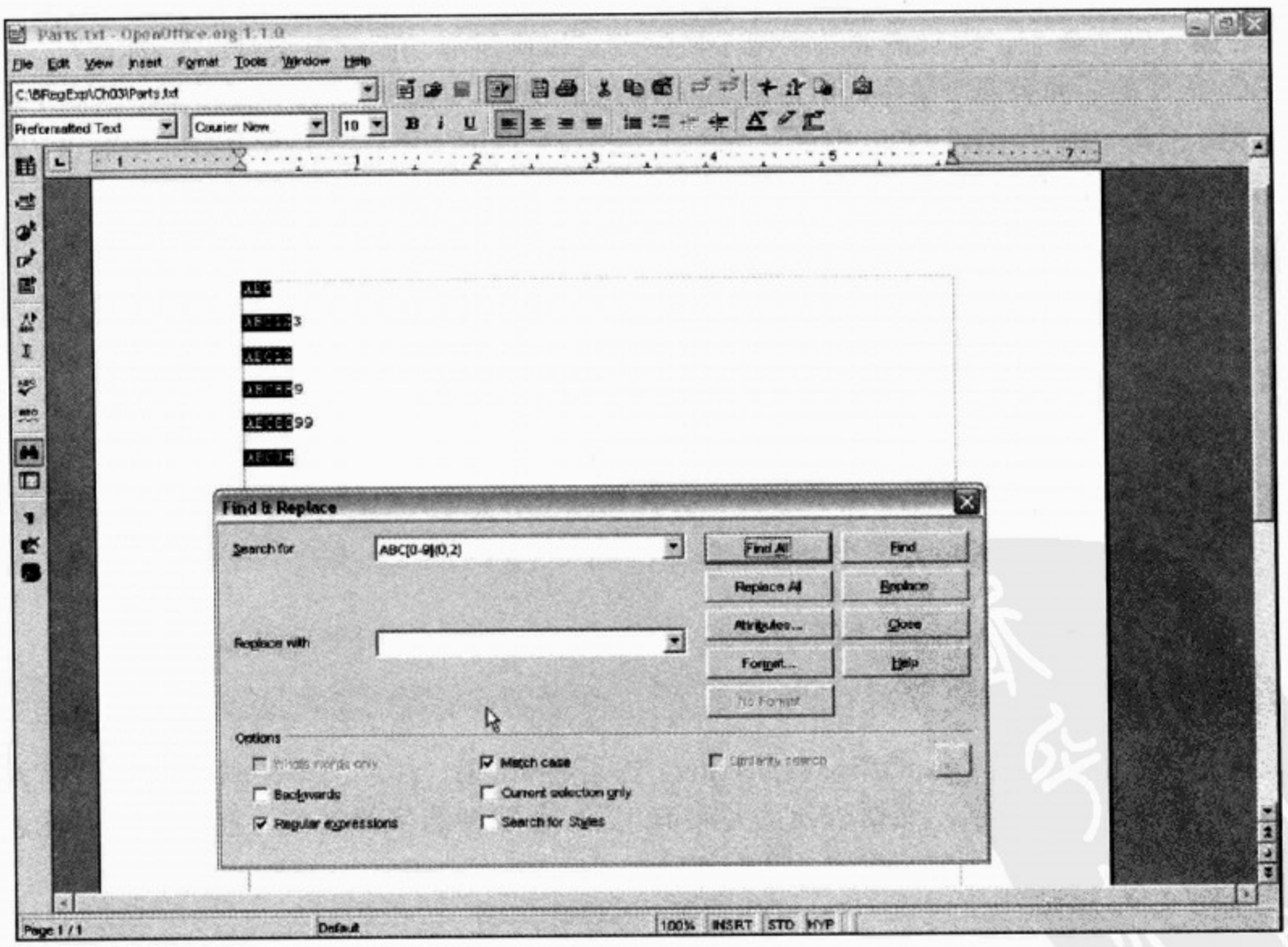


图　3-19

注意，其中有些行只有部分零件编号被匹配。如果读者不明白这是为什么，请参考问题定义中的描述。你是在匹配一个指定的字符序列，而并没有指定想要匹配一个(完整的)零件编号，仅仅只匹配一个字符序列。

工作原理

为什么这个模式会匹配零件编号 ABC 呢？当正则表达式引擎直接位于零件编号 ABC 中的 A 前面时，它会尝试匹配一个大写的 A。这个匹配成功。接着，它尝试匹配一个大写的 B。这个匹配也成功。然后，会尝试匹配一个大写的 C。这次也是成功的。此时，这个正则表达式模式的前三个字符都匹配成功。最后，引擎会尝试匹配模式 [0-9]{0,2}，这个模式的含义是“匹配最少零个而最多两个数字”。在大写的 ABC 中 C 的后面有零个数字。由于大写的 ABC 中 C 的后面恰好有零个数字，所以成功匹配(零个数字符合由限定符{0,2}的最小化说明符指定的“最少零个数字”的标准)。因为该模式的最后一个组件也成功匹配，所以整个模式成功匹配。

而当正则表达式引擎尝试匹配包含零件编号 ABC8899 的那一行时会出现什么情况呢？为什么零件编号 ABC8899 只有前五个字符匹配呢？当正则表达式引擎直接位于 ABC8899 中 A 的前面时，它会尝试将零件编号的第一个字符与大写的 A 匹配，并找到了匹配项。接着，它尝试去匹配大写的 B，而且也找到了匹配项。然后，它又尝试匹配一个大写的 C，这次同样也匹配。此时，正则表达式中的前三个字符都成功匹配了。最后，引擎尝试匹配模式 [0-9]{0,2}，该模式的含义是“匹配最少零个而最多两个数字”。在大写的 C 后面有四个数字，而只需要其中两个数字就可以满足成功匹配的条件了。因为大写的 ABC 中 C 的后面有四个数字，所以匹配(其中两个数字，而这也符合“最多两个数字”的标准)。但是，由于 ABC8899 中最后两个数字不需要匹配，所以它们没有被突出显示。由于模式的所有组件都匹配，所以整个模式也是匹配的。

3.4.4 {n,m}

大括号语法中的最小出现次数说明符不一定是 0。它可以是我们想要的任何数字，但不能超过最大出现次数说明符。

我们来看一个 1~3 个数字的例子。可以在问题定义中这样来描述：

匹配一个大写的 A。如果成功匹配，尝试匹配一个大写的 B。如果成功匹配，尝试匹配一个大写的 C。如果所有三个大写字符都匹配，则尝试匹配最少一个而最多三个数字。

因此，如果要在 Parts.txt 中匹配一到三个数字，就应该使用下面的模式：

```
ABC[0-9]{1,3}
```

图 3-20 显示了在 OpenOffice.org Writer 中匹配的结果。注意，零件编号 ABC 没有匹配，因为它有零个数字，而要查找的是带有 1~3 个数字的字符序列。同时，也要注意 ABC8899 中只有前三个数字组成了匹配部分。

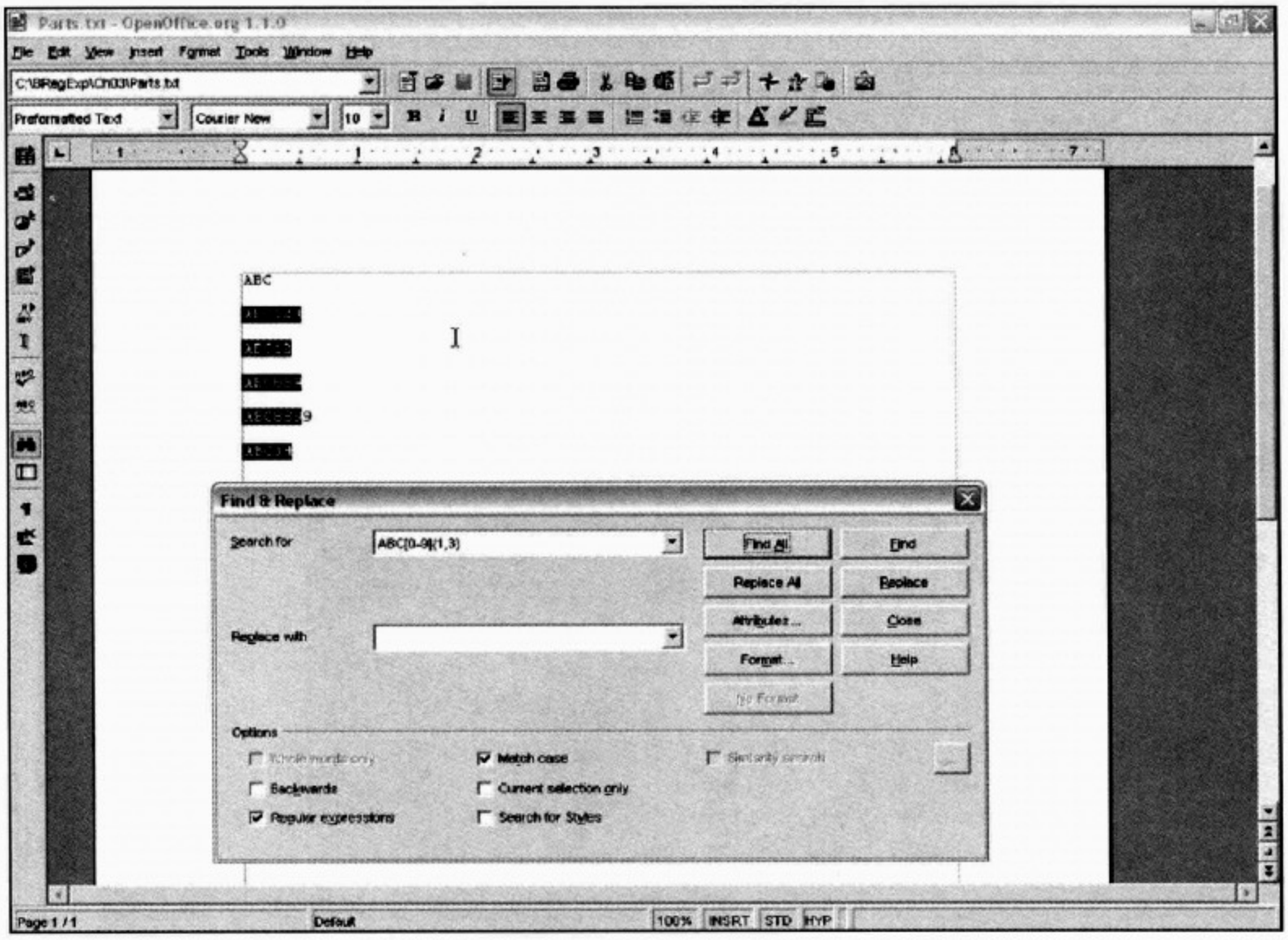


图　3-20

前一节介绍{0,m}语法部分的“工作原理”应该足以帮你理解本例中的匹配过程了。

3.4.5　{n,}

有时候，可能需要匹配无限次。这时，可以通过忽略大括号中的最大出现次数说明符来达到指定无限次的目的。

要指定至少两次而最多无限次，可以使用下面的问题定义：

匹配一个大写的 A。如果成功匹配，尝试匹配一个大写的 B。如果成功匹配，尝试匹配一个大写的 C。如果全部三个大写字符都匹配，则尝试匹配最少两个而最多无数个数字。

可以使用下面的模式来表达该问题定义：

```
ABC[0-9]{2,}
```

图 3-21 显示了在 OpenOffice.org Writer 中进行匹配的结果。注意，现在零件编号 ABC8899 中的所有四个数字都匹配了，因为现在的模式能够匹配的零件编号中的数字是无限个。

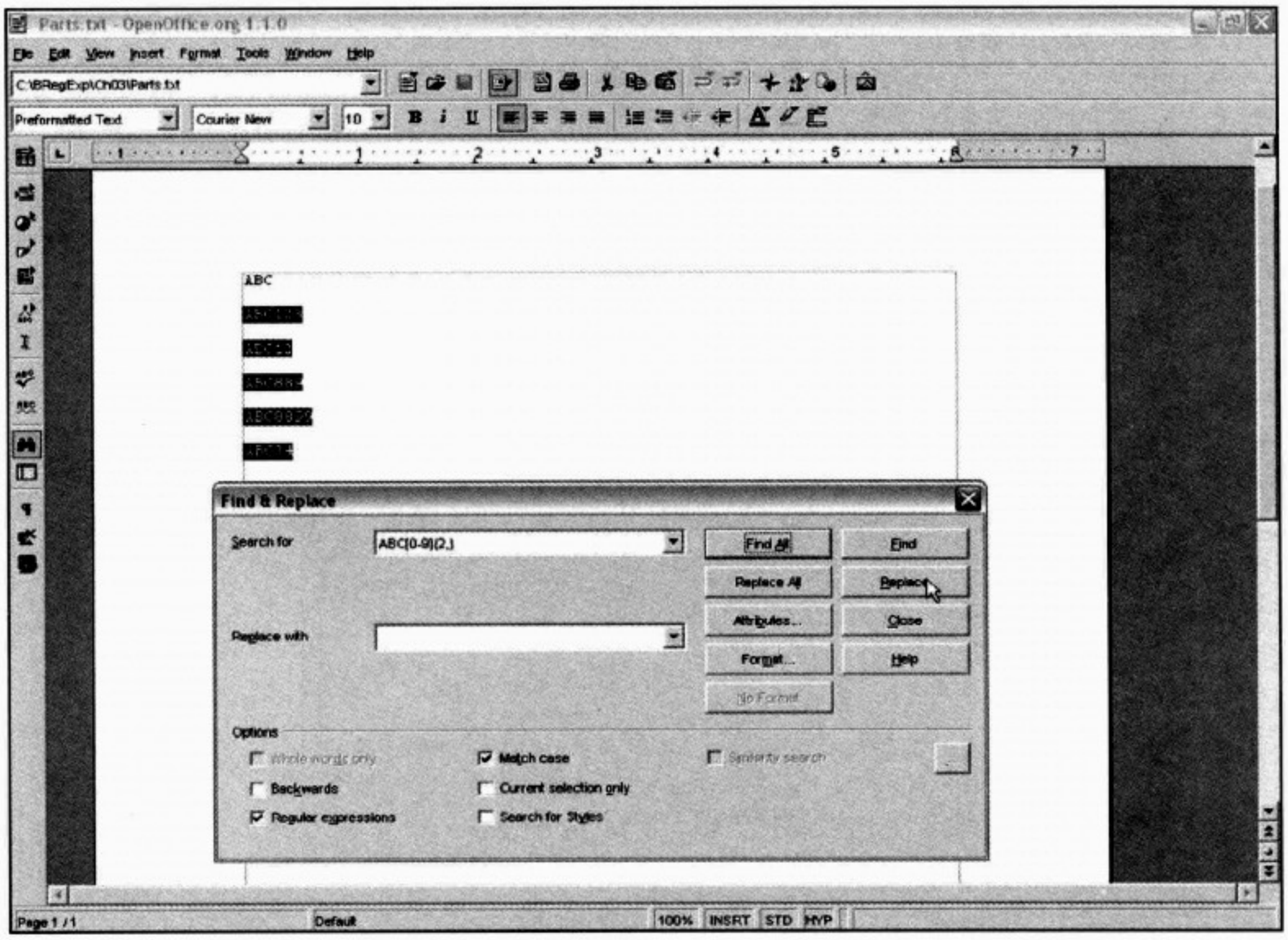


图 3-21

3.5 练习

通过以下练习，可以对自己在本章中学习的正则表达式语法进行自测。

1. 使用 DoubledR.txt 作为测试文件，试验几种匹配文件中双字母的正则表达式模式。例如，文件中有一些小写的双字母 s、m 和 l。请使用不同的语法来严格匹配其中一个字符出现两次的情况。

2. 创建一个正则表达式模式，用它来测试由两个字母字符——一个大写的 A 后跟一个大写的 B，及后跟两个数字组成的零件编号。

3. 修改 UpperL.html，使其中的正则表达式模式与 the 匹配。在浏览器中打开这个文件，并根据指定的正则表达式模式来测试各种不同的文本块。

第 4 章

元字符和修饰符

本章讨论一些正则表达式的元字符和修饰符。元字符可以与直接量字符和限定符(在第 3 章已讨论过)组合使用，来创建更加复杂的正则表达式模式。使用元字符能让正则表达式更有用，更灵活。

所谓元字符，就是用于传达非自身含义的字符。例如，句点字符就是能够表示任何字母字符的元字符——即它能表示任何大小写的英文字母及任何其他语言中的文字字符和任何数字 1 到 9。而其他元字符可以分别指定 ASCII 字母字符和数字。此外，还有匹配空白字符的元字符，所谓空白字符是指空格符和其他不可见的字符，如换行符等。

本章没有介绍全部元字符。某些元字符——如表示行的开始和结尾的元字符(^ 和 $)、表示单词开始和结尾的元字符(\< 和 \>)以及表示单词边界的元字符(\b)——它们会在第 6 章中被介绍和演示。第 6 章介绍的这些元字符表示位置。而本章介绍的元字符表示的是字符类。

修饰符，顾名思义，用来修饰如何应用正则表达式。根据所使用的语言和工具不同，有的修饰符用于指定正则表达式模式是应该按区分大小写的方式还是按不区分大小写的方式来解释，有的修饰符则用于指定如何处理跨行和跨段匹配。

在本章中将学习以下元字符：

- 元字符
- \w 和 \W 元字符
- \d 和 \D 元字符
- 匹配空白字符(如空格符)的元字符

4.1 正则表达式的元字符

在第 3 章中已经介绍过，组合使用直接量字符和限定符能创建出有用而简单的正则表达式模式。然而，直接量字符只能匹配本身，所以它具有很多局限性。有时需要使用更灵活的匹配方案。有些元字符可以匹配一个字符类而不仅仅是一个简单的直接量字符。这些元字符较广的作用域对创建正则表达式非常有用。

本章涉及并示范的许多元字符都是由两个字符组成的。有时，也把这种成对的、组合起来的、传达一个元字符含义的字符称为元序列。本书交替使用元字符和元序列这两个概念。

例如，考虑位于测试文件 Inventory.txt 中的一个零件目录，其内容如下：

```
D99C44

A9DC55

CODD29

RT2C23

MNZC55

UVCC83
```

注意在这些零件编号中，前三个字符的结构是如何变化的。例如，第一个零件编号是一个字母字符后跟两个数字。而第二个零件编号则是一个字母字符后跟一个数字和一个字母字符。用前面使用的技术无法为这种情况指定合适的正则表达式模式。因为零件编号的构成非常多变，我们无法在一个正则表达式模式中仅凭使用直接量字符就能解决问题。而我们要执行的任务是完成与下面的问题定义对应的匹配：

匹配零件编号，其中第 4 个字符是一个大写的 C，第 5 和第 6 个字符是数字。

如果数据结构简单，通过使用第 7 章中介绍的交替选择技术，用相对较少的几个字符，也会找到一个解决方案。而本章的内容则教你如何用其他技术解决数据结构复杂的问题。

4.1.1 考虑字符和位置

需要掌握的一个重要的基本概念是字符和位置之间的区别。

为清楚地理解字符和位置之间的区别，我们看下面的示例文本：

```
This is a simple sentence.
```

这段示例文本中的第一个字符是 This 中的 T。而在大写的 T 前还有一个位置。这个位置不可见，也不与第 3 章中讨论的任何直接量字符匹配。但有些元字符可以匹配位置，比如 ^ 元字符，它匹配上面示例文本中大写 T 前的位置。有关匹配位置而不匹配字符的元字符将在第 6 章中介绍。

上面示例文本中的第二个字符是 This 中小写的 h。在前面大写的字母 T 和这个小写的 h 之间，也有一个位置。通常，这种位于字符序列中字符之间的位置(也就是单词内部的位置)对开发人员而言意义不大。他们一般会对字符串开始处、结尾处以及一个字符序列的开始和结尾处更感兴趣。因此会有专门的元字符来表示这些位置。我们常说的词边界元字符(严格来讲，它匹配的是一个字母序列或者数字字母序列的边界)匹配一个字母字符和一个非字母字符之间的位置。在很多情况下，这种边界同时也是一个词的边界。这类元字

符将在第 6 章中讨论。

能够匹配字符类的元字符是很有用的，而它们正是本章所要简述的内容。

4.1.2　句点(.)元字符

句点元字符是适用范围最广的一个元字符。它可以匹配任何字母字符(无论大写还是小写)，也可以匹配数字。这种广泛适用性是一种优势，因为 . 元字符几乎可以匹配任何字符。所以在并不关心实际要匹配什么内容，或者最终有多少个匹配项的情况下，都可以选用它。不过，. 元字符也有不足，理由一样——它几乎可以匹配任何字符。比如，在一次搜索和替换操作中，要替换与 . 元字符匹配的字符序列是一件非常危险的事情。其结果很可能与第 1 章的 Star Training Company 示例中的 startling 被 Moontling 所替换一样，而潜在的风险却有过之而无不及。

试一试：句点(.)元字符

在 Komodo Regular Expression Toolkit 中测试句点元字符，输入字母字符和数字字符作为测试文本。不过，Komodo Regular Expression Toolkit 只匹配任何字符出现的第一次。

(1) 打开 Komodo 开发环境。

(2) 单击启动 Komodo Regular Expression Toolkit 的按钮，清除该工具包中残留的正则表达式及测试字符串。

(3) 在 Enter a string to match against 区域中输入一个字符串。本例中，输入 Andrew。

(4) 在 Enter a regular expression 区域中输入一个句点。然后直接在 Enter a string to match against 区域的下方观察其匹配结果。

结果是 Match succeeded: 0 groups。有关分组(groups)的概念将在第 7 章中讨论。

元字符匹配英文中的所有字母字符、所有数字、空白字符(如空格符)以及非英语语言中的很多文字字符。图 4-1 显示的是. 元字符在 Komodo Regular Expression Toolkit 中匹配一个大写 A 时的情形。

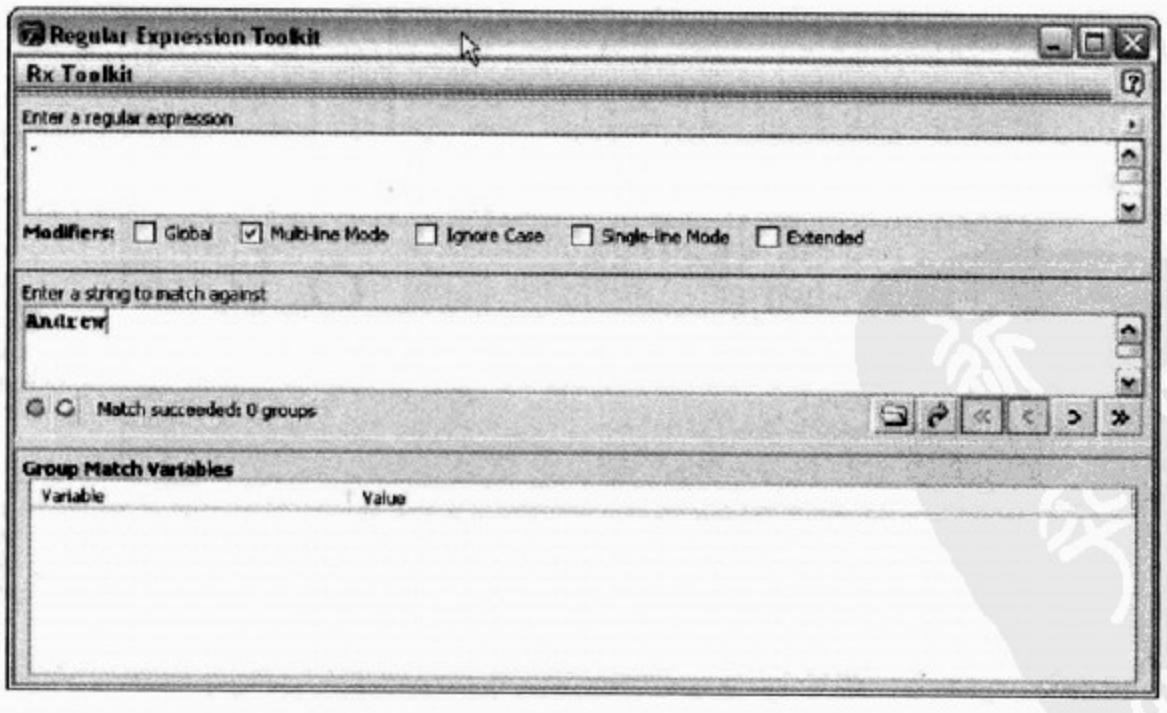


图　4-1

工作原理

当正则表达式模式中出现 . 元字符时，正则表达式引擎会以它来尝试匹配任何大写或小写的英文字母，或者任何数字，另外，还会匹配很多非英语语言中的字符。

正则表达式引擎从直接位于 Andrew 中初始字母 A 之前的位置开始查找匹配项。因为测试文本的第一个字符是 A，于是 A 就被作为 . 元字符的一个匹配项。因为有匹配项，所以初始的 A 以灰绿色背景显示了出来，这表示它是第一个匹配项。

. 元字符同样也匹配非英语语言中的文字字符。

试一试：. 元字符匹配非英语字符

如果已把 Komodo Regular Expression Toolkit 关闭了，按照下面的全部步骤做就可以了。如果没有关闭，则从第二步开始。

(1) 打开 Komodo 开发环境，单击启动 Komodo Regular Expression Toolkit 的按钮。

(2) 清除该工作包中残留的所有正则表达式和/或测试字符串。

(3) 打开 Windows Character Map(Windows 字符映射)。在 Windows XP 中，可以选择 Start | All Programs | Accessories | System Tools，最后选择 Character Map。

(4) 在 Character Map 窗口右侧的滚动条上面单击一下。再单击大写的 Ω(欧米伽)字符，会看到与图 4-2 所示类似的画面。

(5) 在选中大写 Ω 的情况下，单击 Select 按钮。此时，Ω 字符就会出现在 Character Map 窗口的 Characters to copy 文本框中。

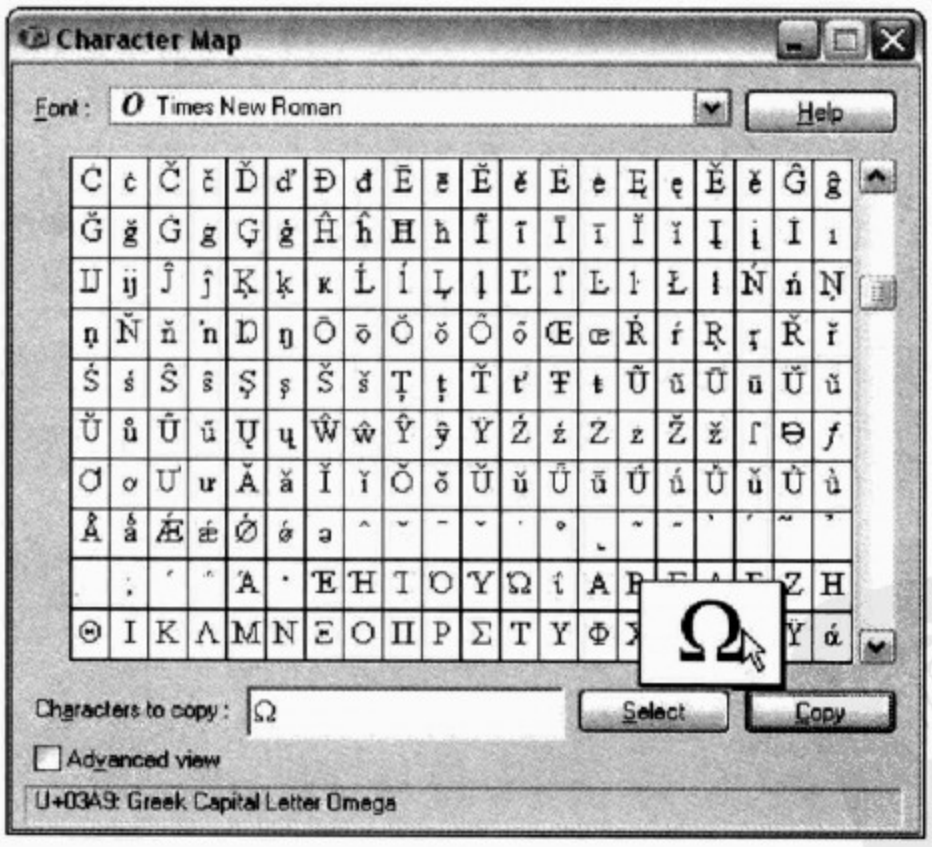


图 4-2

(6) 在 Character Map 窗口中，单击 Copy 按钮。

(7) 在 Komodo Regular Expression Toolkit 中，单击一下 Enter a string to match against 区域，然后按 Ctrl + V 快捷键把测试文本 Ω 粘贴到其中。

(8) 在工具包的 Enter a regular expression 区域中输入一个句点，然后直接观察显示在 Enter a string to match against 区域正下方的结果。注意，此时大写的欧米伽字符也以灰绿色突出显示了出来。这表示它也是 . 元字符的一个匹配项。

工作原理

正则表达式引擎会尝试将 . 元字符与换行符之外的任何字符匹配。首先，它会尝试从大写的欧米伽字符之前的位置开始匹配。第一个字符——大写的欧米伽，与 . 元字符匹配。由于大写的欧米伽字符不是换行符，所以匹配成功。因为整个正则表达式匹配成功(此时模式中只一个元字符)，所以匹配完全成功。

再参照图 4-2，我们知道 . 元字符也匹配希腊大写字母欧米伽。

也可以尝试用 . 元字符匹配其他数字或数字序列的情况(比如，234)，你会发现 . 元字符可以匹配任何从 0 到 9 的数字。

使用 . 元字符匹配英文文本时非常直观。在多数情况下，它都会匹配除了换行符之外的任何字符。然而，也可以通过修改. 元字符的这种匹配特性使其匹配换行符。在 Komodo Regular Expression Toolkit 中可以通过单行模式来实现这一点。

试一试：用 . 元字符匹配一个换行符

(1) 打开 Komodo 开发环境，单击启动 Komodo Regular Expressions Toolkit 的按钮。

(2) 清除工具包中任何正则表达式和测试字符串。

(3) 选中 Global 复选框和 Single-Line Mode 复选框。

(4) 在 Enter a string to match against 区域中单击。按一次回车键，这会在测试区域中添加一个换行符(作为第一个字符)。

(5) 在 Enter a regular expression 区域中输入 . 元字符，并观察结果。测试区域中第一个字符(换行符)所在的位置以灰绿色突出显示。在 Enter a string to match against 区域下方的灰色区域中显示的是： Match succeeded: 0 groups。

工作原理

在选中 Global 和 Single-Line Mode 复选框后，正则表达式引擎会匹配换行符，也会匹配其他正常匹配的字符(本章后面会详细讨论与此有关的修饰符)。因此，当正则表达式引擎在初始的换行符之前的位置开始尝试查找匹配项时，它会首先尝试匹配换行符，而且匹配成功。

由于句点元字符的适用范围很广，所以也存在匹配非目标字符的风险，特别是当后面跟着 * 或 + 限定符时。因为 * 和 + 都允许匹配无限个前面匹配的字符。在很多情况下，正则表达式引擎会进行“贪婪的”匹配，这意味着它会匹配尽可能多的字符。像 .* 和 .+ 这样的模式就可能匹配许多段落或者许多页的文本内容，而这可能并非是操作者想要的结果。

在了解了 . 元字符的作用后，我们回过头来再看一下本章开始时简单提到过的零件目录的例子。

1. 匹配可变的零件编号

相应的问题定义如下：

匹配第 4 个字符是一个大写的 C 而第 5 和第 6 个字符是数字的零件编号。

到底 . 元字符是不是正则表达式模式的理想组件这个问题，部分地取决于数据的结构。如果数据是像测试文件 Inventory.txt 中显示的零件目录那样，那么就可以使用下面的模式来满足前面的问题定义：

```
...C[0-9][0-9]
```

(三个句点后跟一个大写的 C，再跟两个数字)，这个模式与下面的模式等价：

```
.{3}C[0-9][0-9]
```

在这里之所以使用字符类[0-9]来表示数字，是因为这是在 OpenOffice.org Writer 中进行测试的，而它不支持用 \d 元字符匹配数字。

试一试：使用 . 元字符匹配零件目录

(1) 打开 OpenOffice.org Writer，然后打开测试文件 Inventory.txt。

(2) 使用 Ctrl+F 打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框，并在 Search for 文本框中输入模式 ...C[0-9][0-9]。

(4) 单击 Find All 按钮以突出显示所有匹配的文本，观察结果。其结果如图 4-3 所示。注意其中第三个零件编号没有匹配。

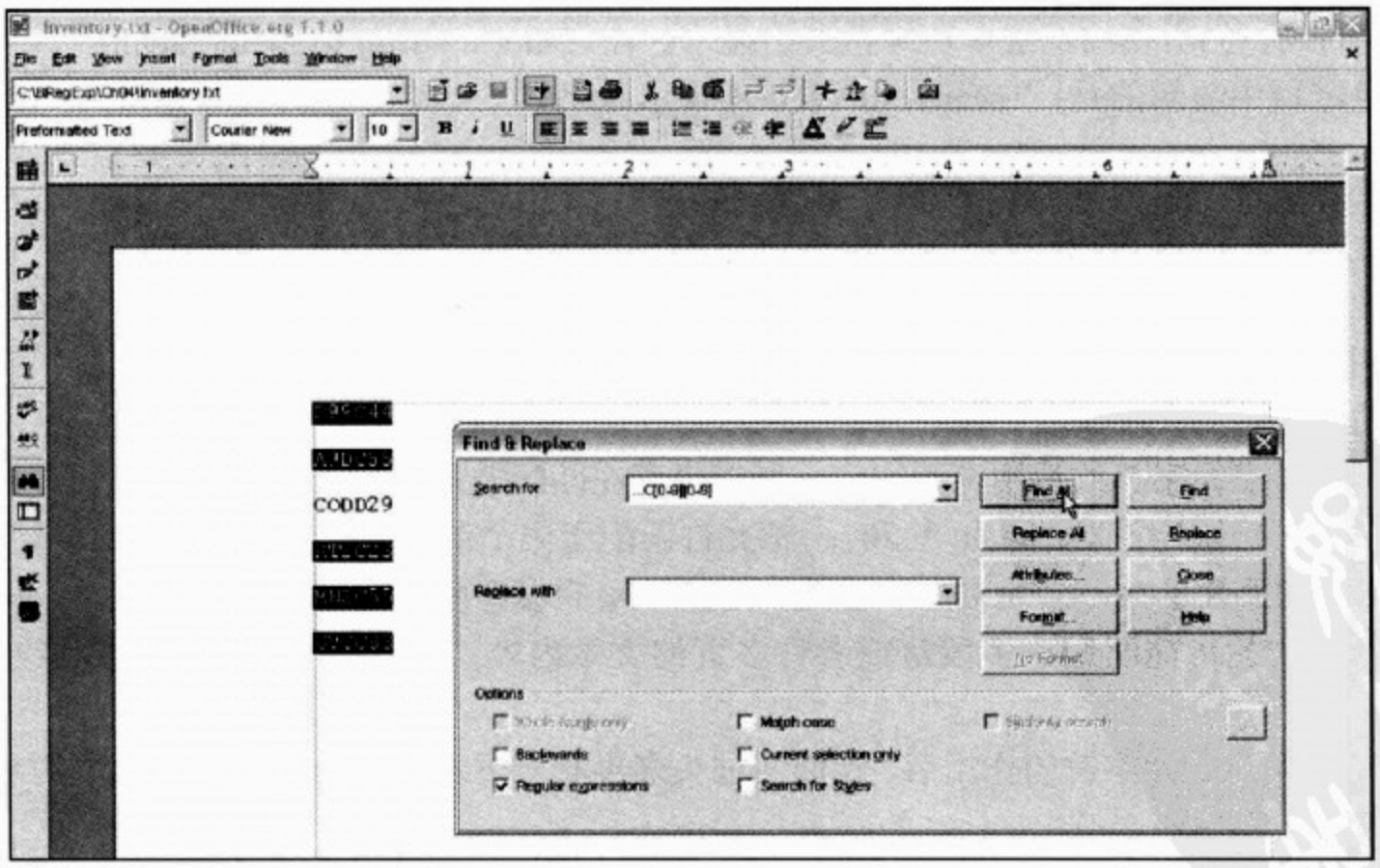


图 4-3

工作原理

我们来看为什么模式 ...C[0-9][0-9] 匹配零件编号 D99C44 ，却不匹配 CODD29。在下面的分析中，称之为零件编号，但严格来讲，正则表达式引擎只会匹配字符序列，它不知道什么是零件编号。

假设正则表达式引擎位于 D99C44 中 D 的前面，它首先尝试用 D 来匹配 . 元字符。结果匹配成功。接着，它尝试匹配第二个 . 元字符。因为该零件编号的第二个字符是 9，所以也与 . 元字符匹配。类似地，第三个 . 元字符匹配第二个 9。正则表达式模式中的第四个字符是大写的 C。这种零件编号中的第四个字符也是 C，所以匹配成功。之后，正则表达式引擎尝试匹配一个数字。因为字符序列 D99C44 中的第一个 4 与模式 [0-9] 匹配，所以零件编号的第五个字符也匹配成功。最后，引擎尝试匹配第二个 [0-9]，因为零件编号的第六个字符是一个数字 4，所以也匹配成功。由于正则表达式模式的所有组件都匹配，所以整个模式成功匹配。因此，该行文本在 OpenOffice.org Writer 中突出显示。

如果正则表达式引擎位于 CODD29 中 C 之前，它首先会尝试将模式中的第一个 . 元字符与 CODD29 的初始字母 C 进行匹配。匹配成功。接着，它会尝试将第二个 . 元字符与 CODD29 中的 O 进行匹配。这次也成功匹配成功。然后，它又尝试用大写的 C 与 CODD29 中的 D 进行匹配。但这次没有匹配成功。因为模式中的一个组件未成功匹配，所以整个模式匹配失败。如果单击的是 Find All 按钮，那么正则表达式引擎会在测试文档中继续向后尝试查找匹配项。

2. 匹配句点直接量

如果 . 元字符存在，就不能把句点作为一个直接量字符在模式中来匹配目标文档中的一个句点。如果要匹配目标文档中的一个句点，就必须在模式中使用一个反斜杠来转义句点元字符：

```
\.
```

试一试：匹配一个直接量句点字符

(1) 打开 Komodo 开发环境，并单击按钮启动 Komodo Regular Expression Toolkit。

(2) 清除所有残留的测试文本和正则表达式。

(3) 在 Enter a string to match against 区域中输入下列文本：This sentence has a period at the end. We will try to match it.

(4) 在 Enter a regular expression 区域中输入模式 \.，并观察其结果，如图 4-4 所示。

工作原理

正则表达式引擎从 This 中大写的 T 前面的位置开始，根据模式 \. 依次比较每一个字符来查找匹配项。匹配的第一个字符位于单词 end 的后面。

前面我们已经看到，. 元字符能够匹配的字符范围极其宽泛。而下面几节中，要介绍几个更具体一点的元字符，主要包括只匹配 ASCII 字母字符(A~Z)的大小写字母的元字符和只匹配数字的元字符。

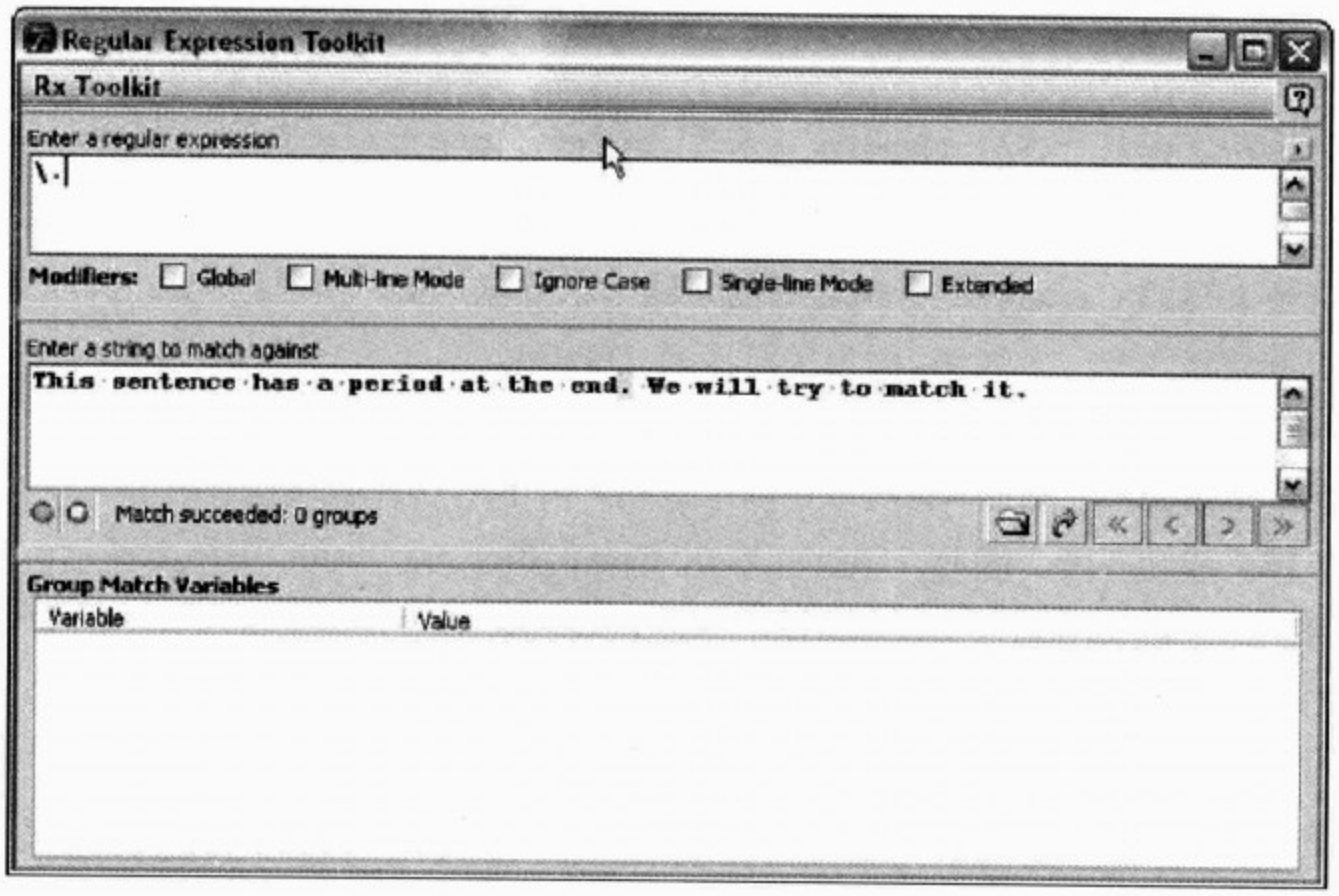


图 4-4

4.1.3 \w 元字符

\w 元字符只匹配英文字母字符、数字和下划线。它与 . 元字符不同，因为它不匹配象形符号、标点符号，在某些实现中也不匹配非英语语言中的字母字符。

通过某些设置，\w 元字符也可以被应用于 Unicode 字符集而不单单是 ASCII 字符集的环境下。在这种情况下，它所匹配的范围比上述的范围更宽一些。

试一试：使用 \w 元字符进行匹配

(1) 打开 Komodo Regular Expression Toolkit，并清除所有残留的正则表达式和测试文本。

(2) 在 Enter a string to match against 区域中，输入 This sentence has a period at the end。

(3) 在 Enter a regular expression 区域中，输入正则表达式 \w{3}。

(4) 在 Enter a string to match against 区域和其下面的灰色区域中观察匹配的结果。Thi 这三个字符应该是突出显示的(通过屏幕观看，它们是灰绿色)。

图 4-5 显示的是这一步的结果。

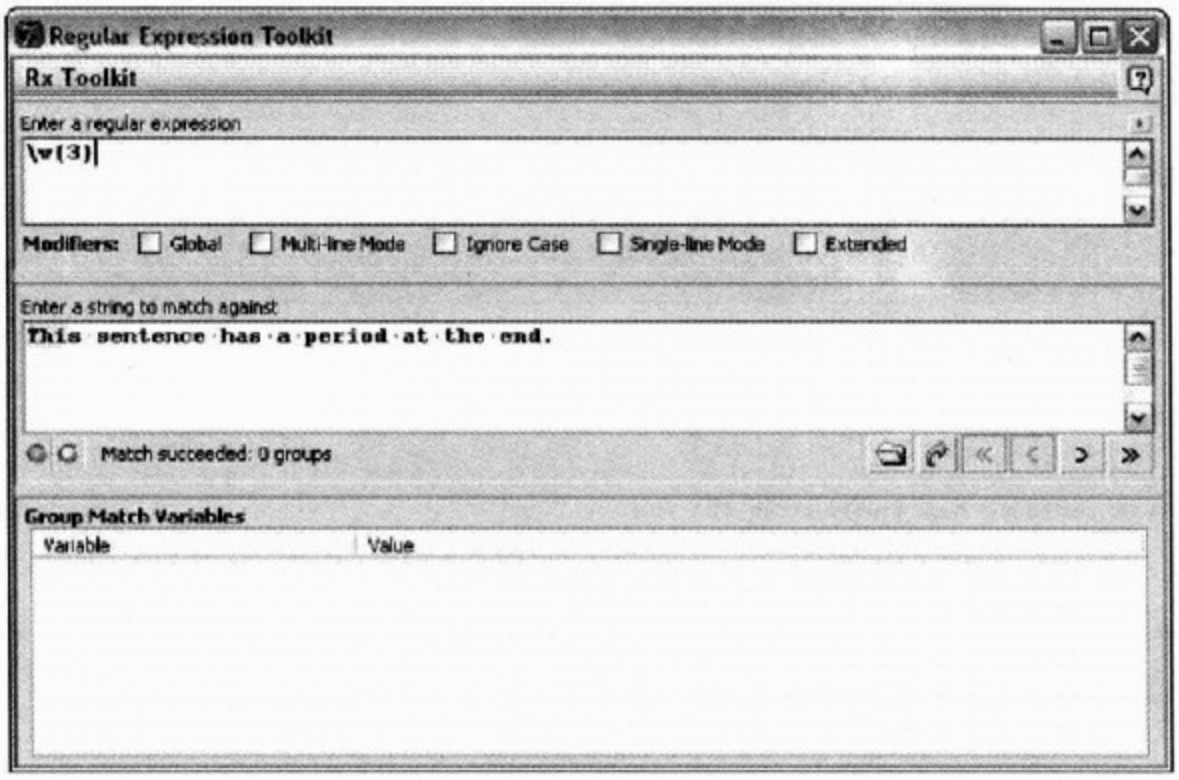


图　4-5

工作原理

模式 \w 表示一个 ASCII 字母字符(A~Z 或 a~z)、一个数字或一个下划线字符。而限定符 {3} 表示匹配三个连续的“单词”字符。

正则表达式引擎从 This 中 T 的前面开始查找匹配项。它首先尝试匹配一个单词字符，因为大写的 T 是一个字母字符，所以匹配成功。然后，它尝试查找另一个单词字符，因为 This 中的 h 也是一个字母字符，所以也匹配。最后，它尝试匹配第三个单词字符，因为 This 中的 i 也是一个字母字符，所以也找到了第三个匹配项。由于模式的所有组件都匹配，所以整个模式就匹配。因此最终三个字母 Thi 被以灰绿色突出显示。

上面用“单词字母(作者之所以提到‘单词字母’是因为元字符 \w 中的 w 取自英文单词 word。译者注)”来代指 \w 元字符匹配的字符容易令人误解，因为多数人并不认为数字和下划线是组成单词的字符。

4.1.4　\W 元字符

\W 元字符用于匹配 \w 元字符不匹配的字符。换句话说，\W 元字符匹配任何非 ASCII 字母字符、非数字以及非下划线字符。

试一试：使用 \W 元字符进行匹配

(1) 打开 Komodo Regular Expression Toolkit，并删除任何残留的正则表达式模式和测试文本。

(2) 在 Enter a string to match against 区域中，输入文本 This sentence has a period at the end。

(3) 在 Enter a regular expression 区域中，输入模式 \W。

(4) 在 Enter a string to match against 区域和其下方的灰色区域中观察结果。正确的结果

应该是 This 后面、sentence 之前的空格符被匹配——通过屏幕观看时它们以灰绿色突出显示(见图 4-6)。

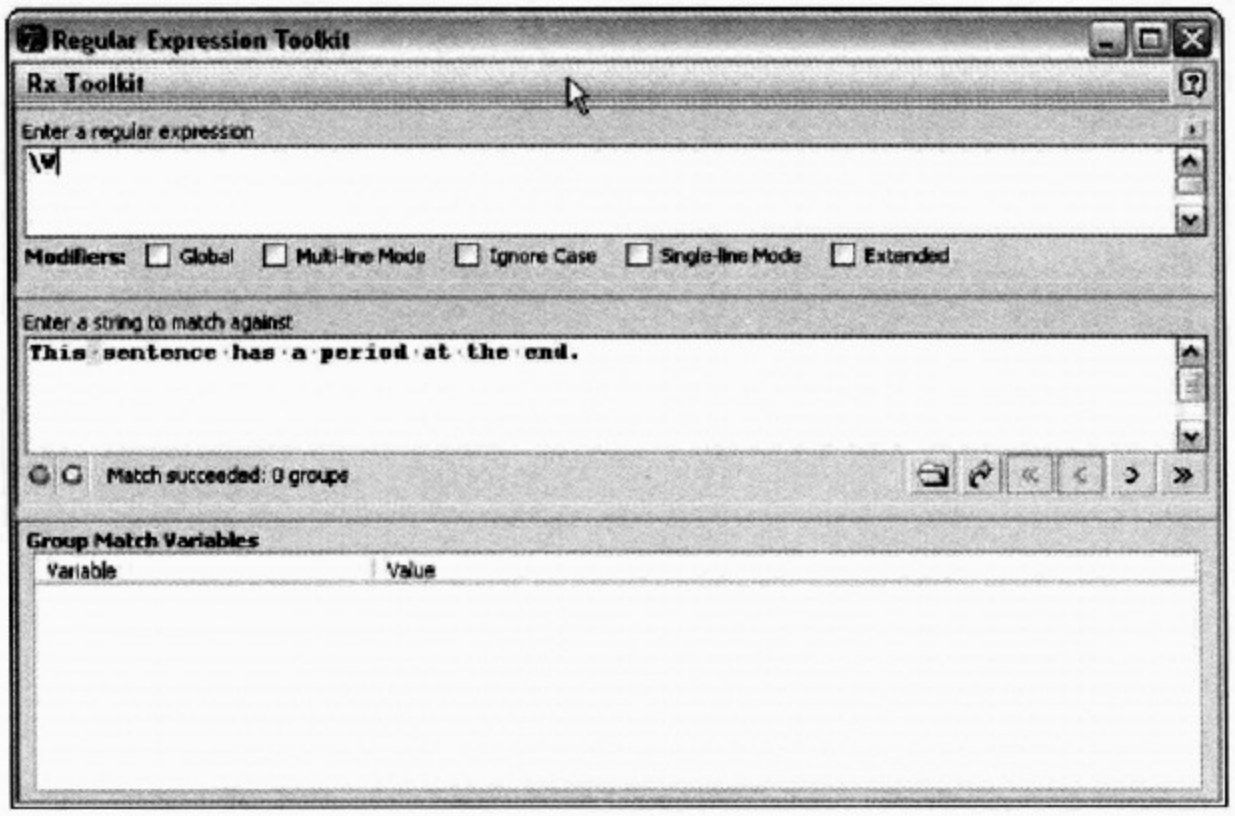


图 4-6

工作原理

正则表达式引擎从 This 中 T 之前的位置开始查找匹配项。它首先根据模式中的 \W 来匹配 This 中大写的 T。没有匹配。然后，它又依次尝试匹配 This 中其他的字母，但都没有成功匹配。因为这些字母都是“单词字符”。当正则表达式引擎到达 This 中 s 之后的位置时，它找到了匹配项。因为 s 后面的空格符不是一个单词字符，所以与 \W 元字符匹配。因而，这个匹配的字符(前面句子中 This 后面的空格符)就被突出显示出来了。

4.1.5 数字和非数字

许多正则表达式的实现用字符来表示数字或者非数字。

其中，元字符 \d 被广泛地用于表示数字。而元字符 \D 则在那些支持 \d 元字符的实现中被用于表示非数字。

OpenOffice.org Writer 不支持 \d 和 \D 元字符，所以无法用它来演示这两个元字符的作用。

1. \d 元字符

\d 元字符匹配一个 0~9 的数字。

打开 Digits.txt 示例，其内容如下：

```
D1
AB8
DE9
```

```
7ED
6py
0EC
E3
D2
F4
GHI5
ABC89
```

试一试：使用 \d 元字符进行匹配

(1) 打开 Komodo Regular Expression Toolkit，并清除所有残留的正则表达式和测试文本。

(2) 在 Enter a string to match against 区域中，输入 Digits.txt 中的前两行内容。

(3) 在 Enter a regular expression 区域中，输入模式 \d。

(4) 在 Enter a string to match against 区域和其下方的灰色区域中观察其结果。图 4-7 显示的是执行完这一步后出现的结果。

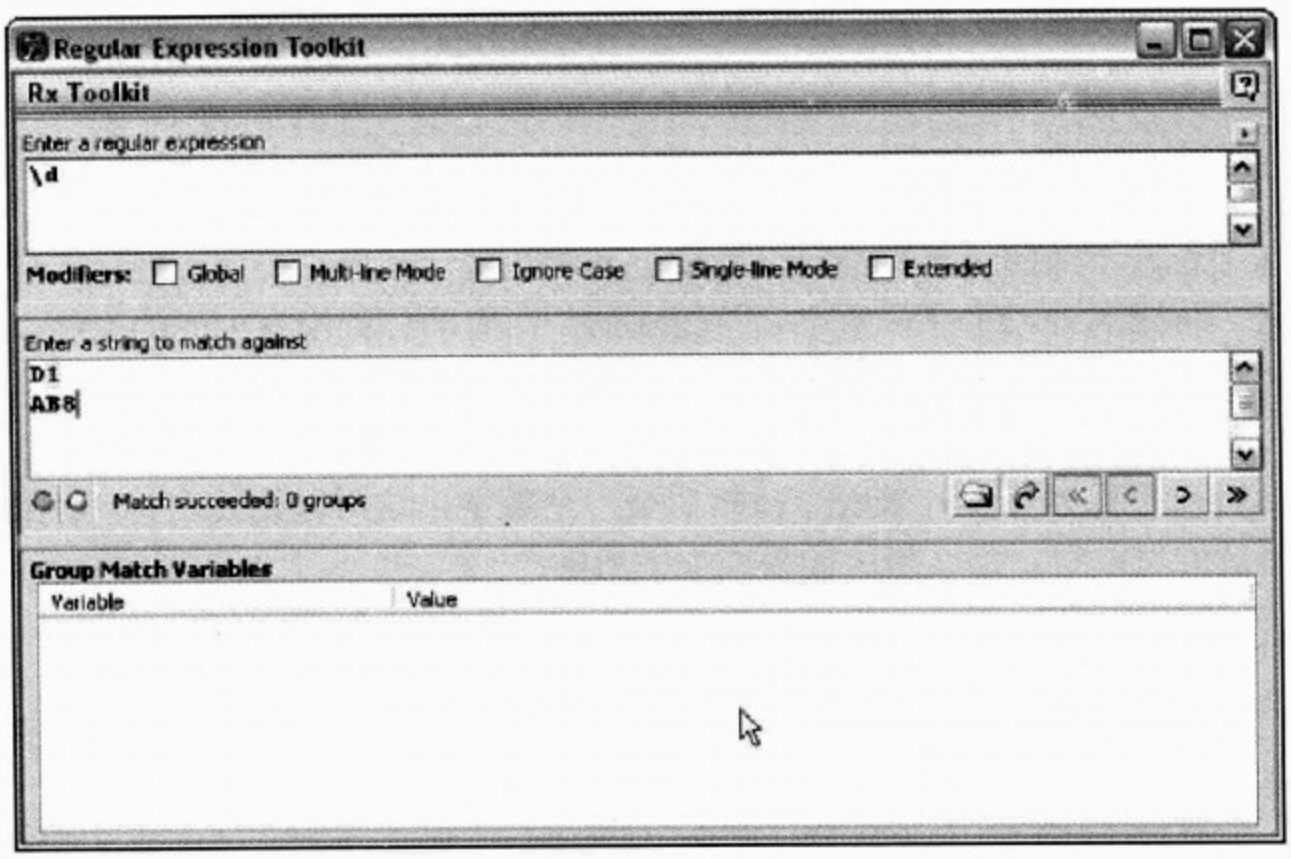


图　4-7

工作原理

\d 元字符只匹配一个数字。正则表达式引擎从 D1 中 D 之前的位置开始匹配。第一个字符——D，因其不是一个数字，所以不匹配。然后，正则表达式引擎移到 D 后面的位置开始匹配该位置之后的字符。因为那个字符是数字 1，所以匹配成功。

2. 加拿大邮政编码示例

加拿大邮政编码的形式是 A1A 1A1，即一个字母字符，一个数字，一个字母字符，一个空格(通常是一个)，然后一个数字，一个字母字符，再一个数字。

要匹配加拿大邮政编码，可以使用下面的问题定义：

> **匹配一个 ASCII 字母字符，后跟一个数字，再后跟一个 ASCII 字母字符，然后跟一个可选的空格符，之后跟一个数字，再后跟一个 ASCII 字母字符，最后跟一个数字。**

在样本文件 CanPostcodes.txt 中，包含一些示例字符序列。其中一些遵照了上面描述的格式，即按照加拿大邮政编码的结构构成(当然，这个文件中的例子都是简单的字符序列)。当然，当前实行的加拿大邮政编码的第一个字符并非全都是一个字母字符。要了解加拿大邮政编码的更多信息，可访问 www.canadapost.com。

```
T3Z 3N7
D8R 8C4
RR4 88D
P9C 3Q4
V2X 3RU
V5R8S4
M8N 7LK
J1M6U4
S1B 2R9
88B U2L
D7R 7L2
F9Z6G4
```

通过仔细查看上面的示例数据可以发现，其中某些行是由三个字母数字字符序列后跟一个空格符，再跟另外三个字母数字字符组成的。而有的行则没有中间的空格。因此，如果想查找所有有效的字符序列，必须要令模式中的空格字符可选。

首先，我们来设计一个与省略了空格符的字符序列匹配的模式。假设，先匹配一个字母字符，这可以用元字符 \w 来表示；然后是一个数字，这可以由元字符 \d 来匹配。如果不考虑可选的空格符，那么最终的模式如下所示：

```
\w\d\w\d\w\d
```

它匹配的是构成加拿大邮政编码的字母、数字、字母、数字、字母、数字序列。

如果还想再匹配一个可选的空格符，可以简单地在模式中添加一个空格符和一个 ? 限定符，这样最多只能匹配一个空格符；若将 ? 换成 *，就可以匹配任意多个空格符。假设要匹配没有空格或只有一个空格，可以使用下面的模式：

```
\w\d\w ?\d\w\d
```

试一试：匹配加拿大邮政编码

(1) 打开 Komodo Regular Expression Toolkit，并清除残留的正则表达式和测试文本。

(2) 在 Enter a string to match against 区域中，输入 CanPostcodes.txt 中的前两行作为测试文本。

(3) 在 Enter a regular expression 区域中，输入模式 \w\d\w ?\d\w\d。

(4) 在 Enter a string to match against 区域及其下方的灰色区域中观察其结果，如图 4-8 所示。

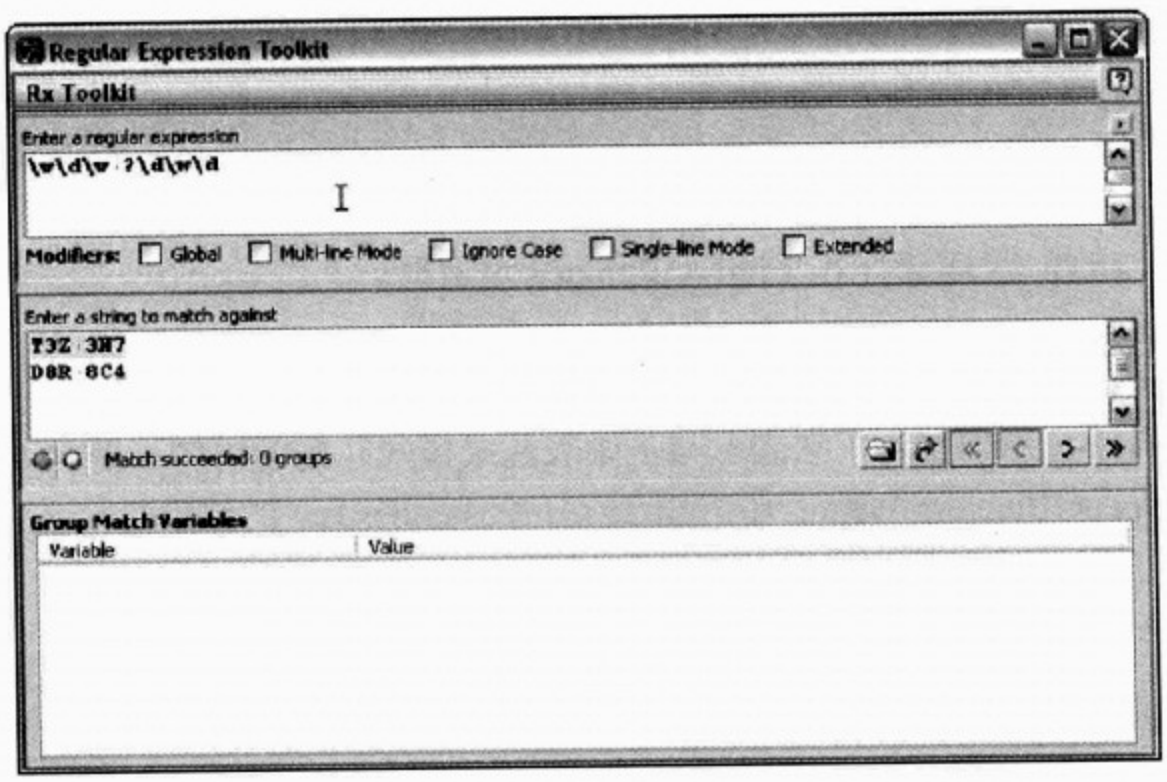


图　4-8

工作原理

结果是其中的文本 T3Z 3N7 匹配。正则表达式引擎从大写的 T 之前的位置开始尝试用模式中的第一个元字符 \w 匹配该位置之后的字符。因为 T 是一个 ASCII 字母字符，所以匹配成功。接着它尝试将数字 3 与 \d 元字符进行匹配。同样也匹配。然后，继续尝试根据大写的 Z 来匹配模式 \w。这次也匹配。之后，又尝试将随后的空格符(在 Komodo Regular Expression Toolkit 中显示为中点)与模式 ?(一个空格符后跟一个 ? 限定符)匹配。同样也匹配。接着，它尝试匹配模式 \d 与第二个数字 3。这也匹配。然后，它尝试匹配模式 \w 和大写的 N。由于 N 是一个字母字符，所以匹配成功。最后，它又尝试匹配元字符 \d 和数字 7，当然也匹配。因为正则表达式模式的所有组件都匹配，所以整个模式匹配。在 Komodo Regular Expression Toolkit 中，匹配的文本以灰绿色被突出显示出来。

如果希望匹配的是至少有一个空格符的字符序列，可以使用 + 限定符，因为它匹配一个或多个前面字符或组的实例(这里借用实例一词表示多个相同或者说重复的字符或字符组——犹如面向对象编程语言中同一个类可以有多个实例一样。译者注)。

有些正则表达式的实现(例如，在 OpenOffice.org Writer 中)不支持 \w 和 \d 元字符，此时必须使用字符类。有关字符类的更多内容将在第 5 章中介绍。

下面的字符类与元字符 \w 是对应的：

```
[A-Za-z0-9_]
```

而下面的字符类与元字符 \d 是对应的：

```
[0-9]
```

假设加拿大邮政编码中只使用大写的字母字符，那么通过使用字符类，下面的模式将

会得到与前面的模式同样的结果——只有大写的字母字符才会匹配：

```
[A-Z][0-9][A-Z] ?[0-9][A-Z][0-9]
```

试一试：匹配加拿大邮政编码的另一种方法

(1) 打开 OpenOffice.org Writer，再打开测试文件 CanPostcodes.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中输入模式 [A-Z][0-9][A-Z] ?[0-9][A-Z][0-9]。

(5) 观察突出显示的文本，它表示与正则表达式模式匹配的内容。图 4-9 显示的是将这个模式在 OpenOffice.org Writer 中应用于 CanPostcodes.txt 的结果。

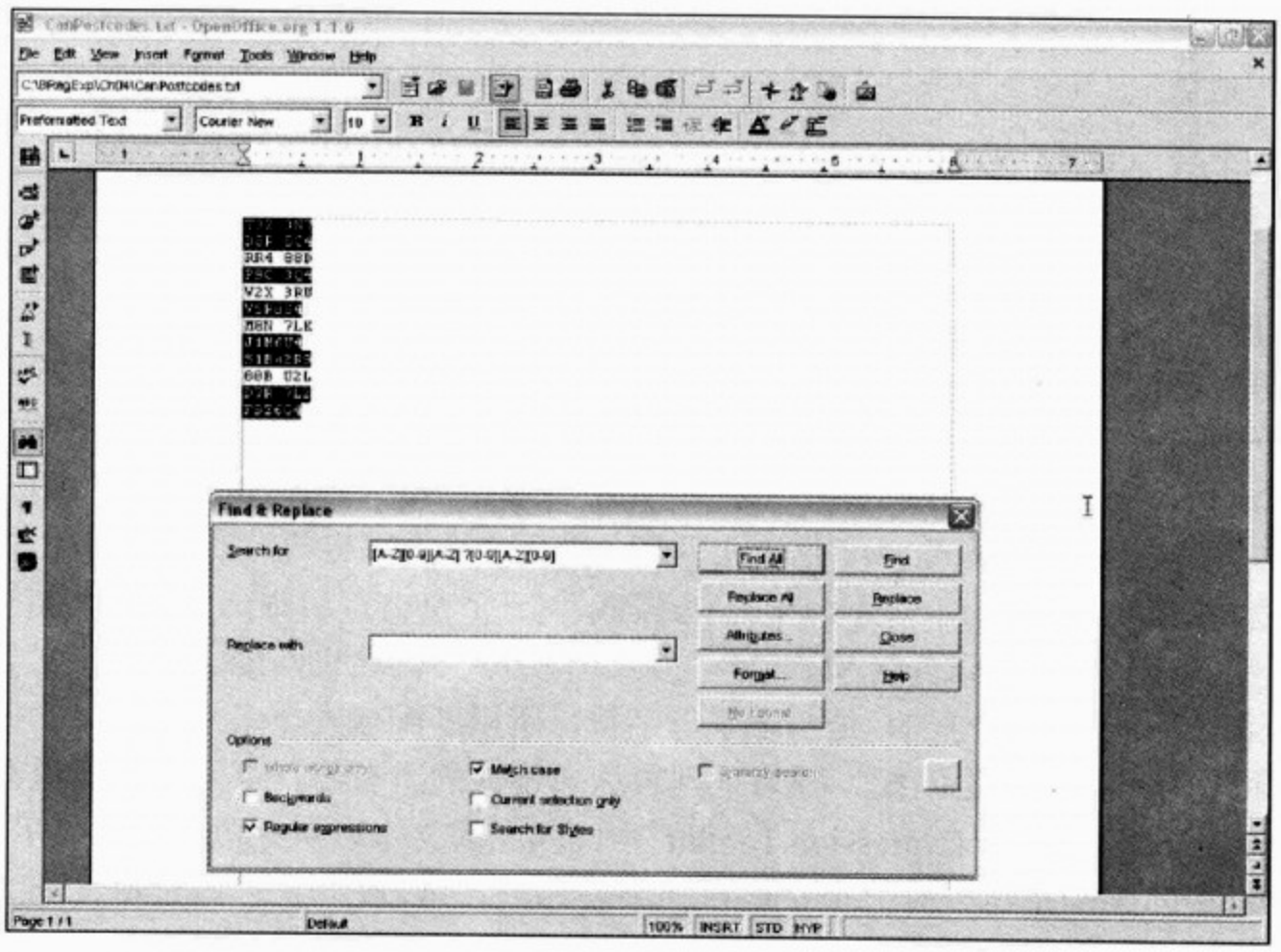


图 4-9

工作原理

结果是文本 T3Z 3N7 匹配。正则表达式引擎从大写的 T 之前的位置开始尝试将第一个字符类 [A-Z] 与该位置之后的字符进行匹配。匹配成功。然后尝试匹配字符类 [0-9] 与数字 3。这次也匹配。接着尝试匹配模式 [A-Z] 和字符 Z。也匹配。然后，尝试匹配模式 ?(一个空格符后跟一个 ? 限定符)和一个空格符。这也匹配。接着，再尝试匹配模式 [0-9] 和第二个数字 3。还是匹配。然后，尝试匹配模式 [A-Z] 和大写的 N。因为 N 是一个字母字符，所以匹配也成功。最后，它尝试匹配字符类 [0-9] 和数字 7，当然也匹配。由于这个正则表达式模式的所有组件都成功匹配，所以整个模式匹配。而匹配的文本会在

OpenOffice.org Writer 中突出显示。

如果假设也可以使用小写的字母字符，那么下面的模式就可以满足既有大写也有小写的字母字符的情况：

```
[A-Za-z][0-9][A-Za-z] ?[0-9][A-Za-z][0-9]
```

3. \D 元字符

\d 元字符匹配任何一个 0 到 9 的数字，而 \D 元字符则匹配 \d 元字符不匹配的其他字符。所以 \D 元字符匹配的字符包括字母字符(英文和非英文的字母/字符)和空白字符(如空格符)。

试一试：使用 \D 元字符

(1) 打开 Komodo Regular Expression Toolkit，并清除所有残留的正则表达式模式和测试文本。

(2) 在 Enter a string to match against 区域中输入测试文本 321ABC。

(3) 在 Enter a regular expression 区域中输入正则表达式模式\D。

(4) 在 Enter a string to match against 区域及其下方的区域中观察结果。在该区域下方，会看到预料中的信息 Match succeeded: 0 groups。而在该区域中，321ABC 中的 A 被突出显示。图 4-10 显示的是 Komodo Regular Expression Toolkit 中的匹配结果。

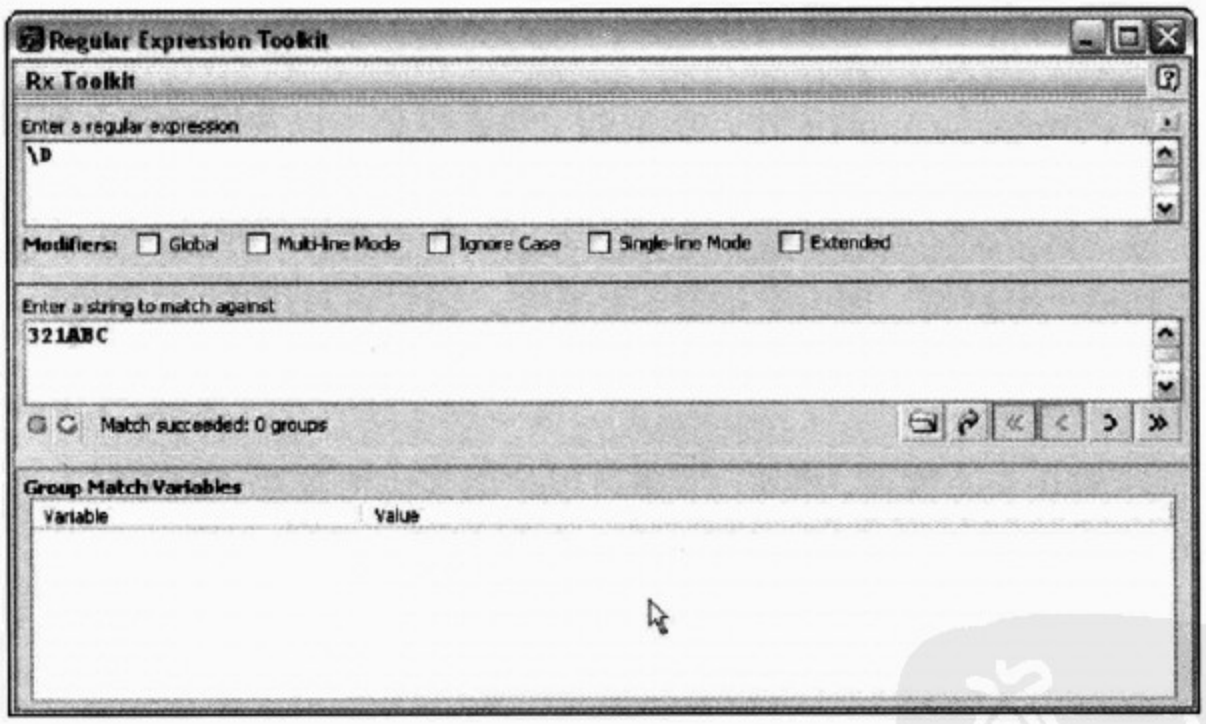


图　4-10

(5) 在 Enter a string to match against 区域中 321ABC 的 321 和 ABC 之间单击鼠标健并按一次空格键。

(6) 再观察 Enter a string to match against 区域及其下方灰色区域中的结果。在上面的文本区域中，位于 321ABC 中的 1 和 A 之间的空格符以灰绿色(在屏幕中)突出显示。而在下方的灰色区域中，会看到信息 Match succeeded: 0 groups。图 4-11 显示的就是操作后的结果。

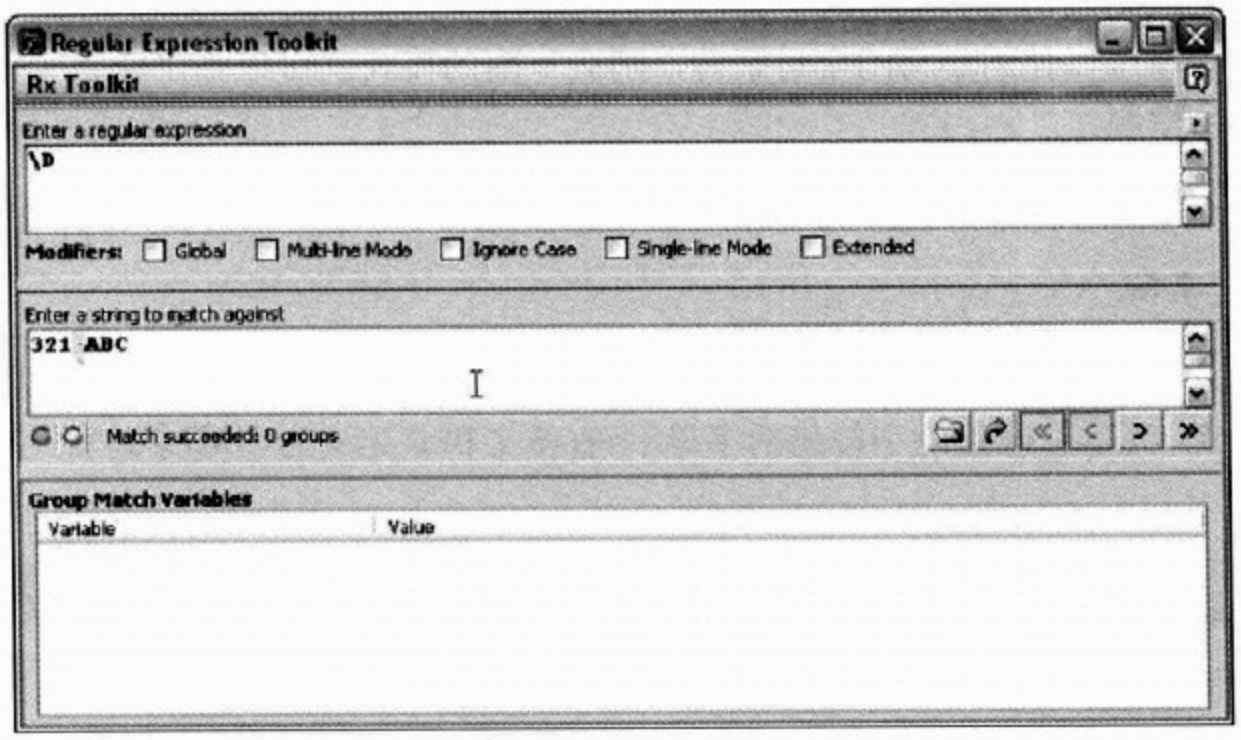


图 4-11

工作原理

首先，考虑测试文本为 321ABC 时的情况。正则表达式引擎在 321ABC 中 3 之前的位置开始尝试查找匹配项。因为该位置后面的第一个字符是数字 3，与 \D(它匹配非数字字符)不匹配。所以正则表达式引擎移动到 321ABC 中 3 后面的位置，并再次尝试查找匹配项。这次还是失败了。当正则表达式引擎移动到 321ABC 中 A 之前的位置时，它再尝试匹配，匹配成功。因为大写的 A 不是一个数字，所以它与 \D 匹配。如前所述，在 Komodo Regular Expression Toolkit 中，匹配的字符在屏幕中以灰绿色突出显示。

在第 5 步之后，测试文本 321ABC 中的 1 和 A 之间增加了一个空格符。正则表达式引擎在位于该空格之前的所有位置时的匹配都以失败告终。当正则表达式继续查找下一个匹配项时遇到这个空格符，因为空格符不是数字，所以它与模式 \D 匹配。

4. 交替选择\d 或\D

\d 元字符匹配 0~9 的数字。事实上，也可以使用其他稍微复杂的正则表达式模式来实现相同的定义。这种技术涉及到交替选择(详细内容在第 7 章中介绍)，或者使用字符类(在第 5 章介绍)。在本章开始的时候，我们看到过一个使用字符类的例子。现在，再来看一个使用交替选择的简单例子，以便掌握在不支持 \d 和 \D 元字符的实现中如何匹配数字或非数字的方法。

\d 元字符是“0 或 1 或 2 或 3 或 4 或 5 或 6 或 7 或 8 或 9”这一思想的简洁表达方式。而这一思想还可以通过下面对应的模式来表达，其中的竖线表示逻辑或：

```
(0|1|2|3|4|5|6|7|8|9)
```

图 4-12 中显示的是 OpenOffice.org Writer 中使用上述模式的结果，它不支持 \d 元字符。

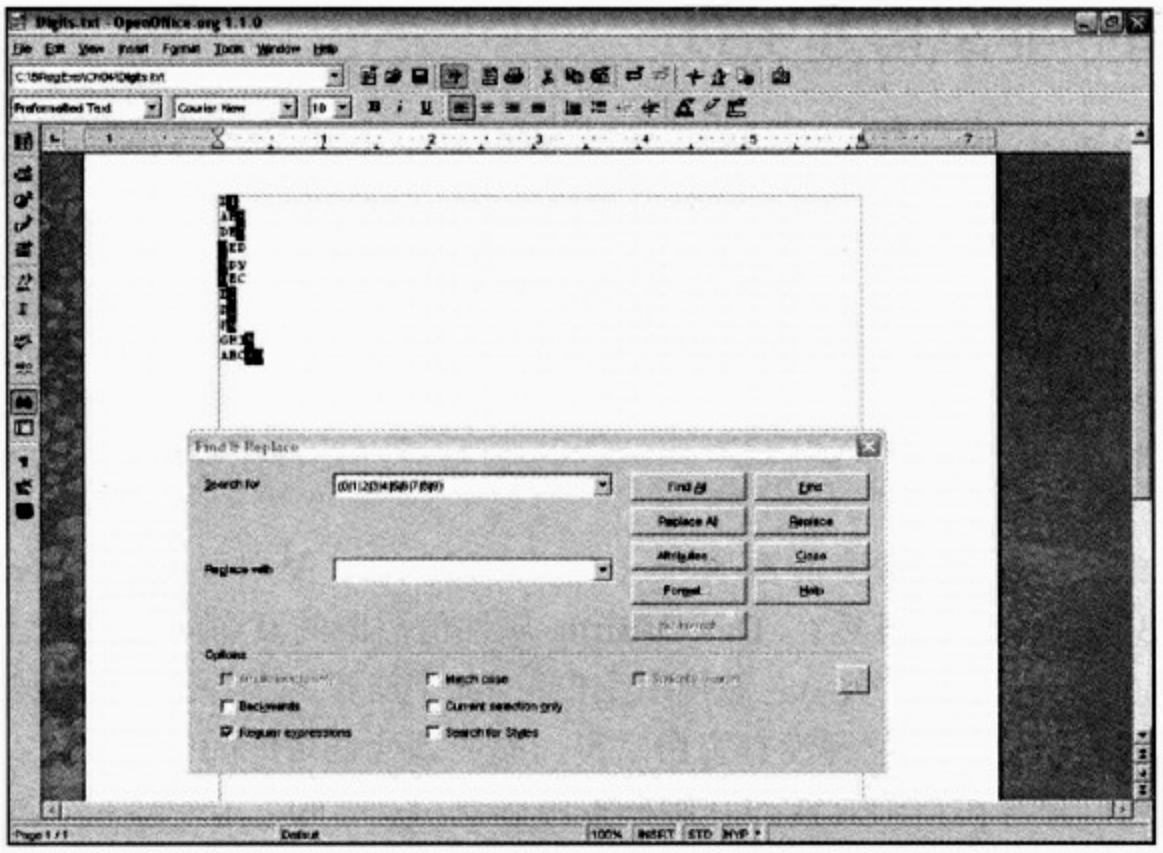


图　4-12

另外一个表达同一思想的、更简洁些的方法是使用字符类，字符类的开始和结束分别以 [和] 元字符表示。要表示包含数字的字符类，可以写成下面这样的模式：

```
[0123456789]
```

或者，更简洁些，在字符类中使用范围，如下所示：

```
[0-9]
```

图 4-13 显示的是用字符类[0-9]来匹配测试文件 Digits.txt 中所有数字的结果。

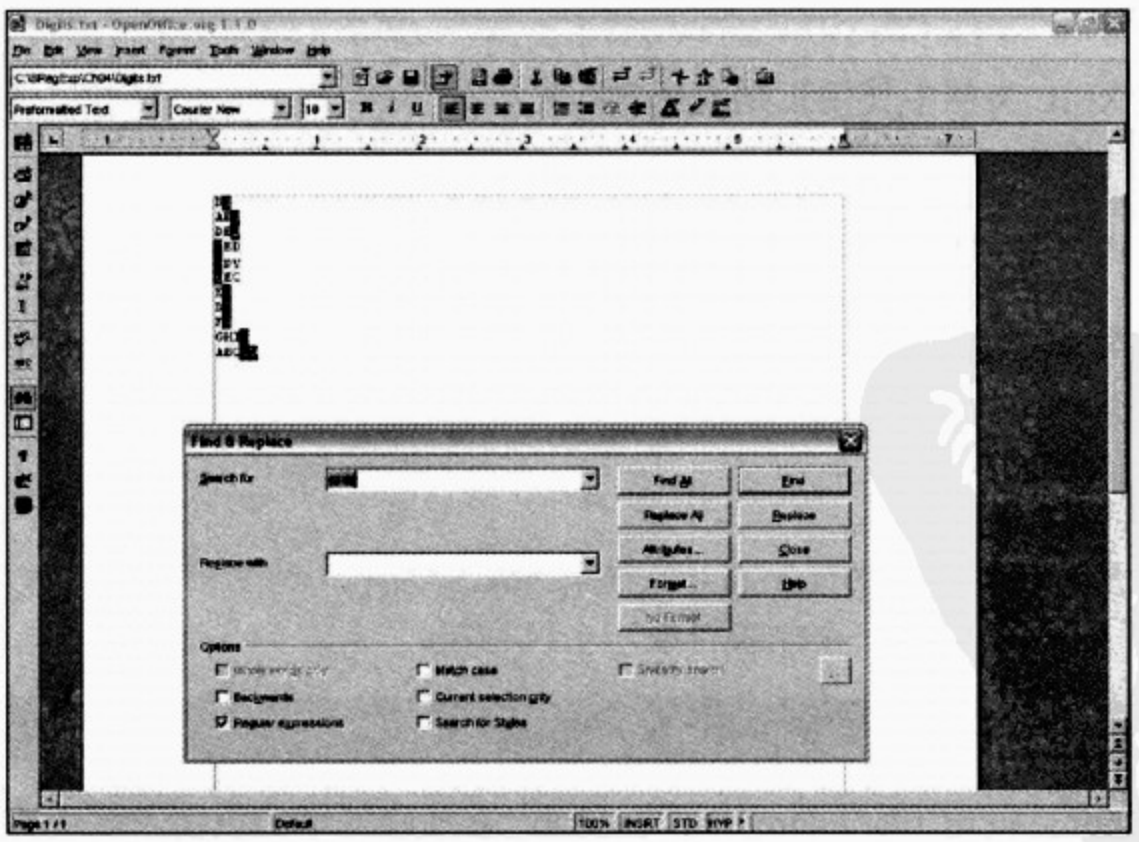


图　4-13

4.2 空白和非空白元字符

空白字符经常会出现在数据中的重要位置上。例如，在 XML 元素的开始标签中经常会有空白字符。假设有一个简单的 XML 文档 Person1.xml，其中包含一个人的有关数据：

```
<?xml version='1.0'?>
<Person DateOfBirth="1970/01/12">
<FirstName>John</FirstName>
<LastName>Scoliosis</LastName>
</Person>
```

其中的 DateOfBirth 属性前面有一个空白字符——否则，元素名会变成错误的 PersonDateOfBirth。此外，虽然在 DateOfBirth 属性值后面的双引号与 > 之间没有空白，但 XML 规则允许在右尖括号(>)之前使用空白符，如空格符、制表符或这两种字符的混合。所以，不能简单地假定那个位置没有空白字符，而应该允许可能的空白存在。

XML 规则也允许在一个元素的开始标签内部使用换行符。在 Person2.xml 文件中，Person 元素的开始标签中放置了两个换行符，使其更容易识别。而对 XML 解析器而言，这仍然是与 Person1.xml 相同的一个文档——因为它们的逻辑结构相同。但是，从正则表达式的角度来看，这两个文档则不一样——因为 Person 元素的开始标签中包含着不同的字符序列。

```
<?xml version='1.0'?>
<Person
DateOfBirth="1970/01/12"
>
<FirstName>John</FirstName>
<LastName>Scoliosis</LastName>
</Person>
```

如果想在 XML 文档中使用正则表达式匹配字符序列，那么必须要考虑到在 XML 文档中元素的开始标签内，或者其他地方可能存在的空白字符。

在对 HTML 和 XHTML 文档应用正则表达式时，同样也要考虑到空白字符的存在。

许多正则表达式的实现都提供了一个或几个字符，用来匹配其中一些或者全部可能出现的空白字符。首先，我们看一下 \s 元字符。

4.2.1 \s 元字符

\s 元字符是针对性最差的元字符，它能够匹配任何单个的空白字符。具体来说，\s 元字符可以匹配一个空格符、一个制表符或一个换行符。

试一试：使用 \s 元字符

(1) 打开 Komodo regular expression Toolkit，并清除残留的正则表达式和测试文本。

(2) 在 Enter a string to match against 区域中，输入 ABC 并按回车键，然后再输入

DEF。

(3) 在 Enter a regular expression 区域中，输入模式 \s。

(4) 在 Enter a string to match against 区域中和其下方的区域中观察其结果。注意在屏幕中，位于 ABC 中 C 之后的不可见字符(一个换行符)以灰绿色突出显示。

图 4-14 显示的是预料中的结果。此时，\s 元字符匹配了换行符。

(5) 删除正则表达式和测试文本(之所以要这样做，是因为在 Komodo V 2.5 版中，编辑之后突出显示的内容会被放错位置)。

(6) 在 Enter a string to match against 区域中，输入字符串 ABC DEF (即，ABC，然后是一个空格符，然后是 DEF)。

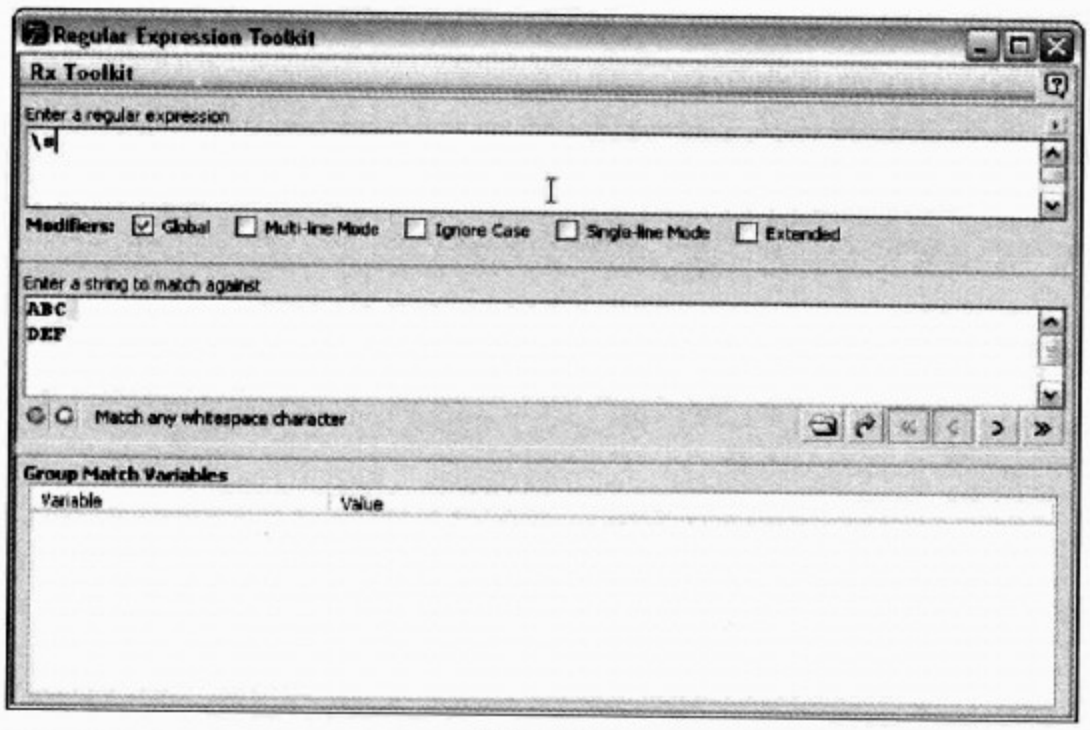


图　4-14

(7) 在 Enter a regular expression 区域中，输入模式 \s，并观察结果。

图 4-15 显示的是预料中的结果。此时，\s 元字符匹配了其中的空格符(在 Komodo Regular Expression Toolkit 中显示的是一个中点)。

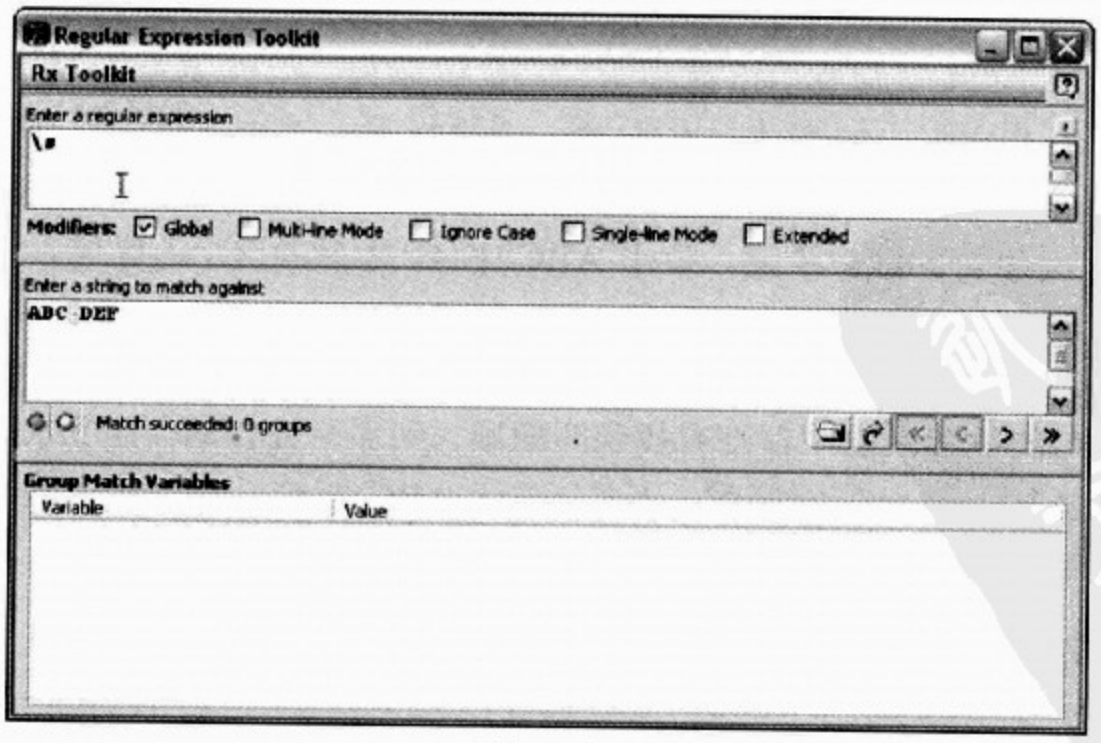


图　4-15

在 Enter a string to match against 区域中不可能输入一个制表符，但是可以粘贴一个制表符。

(8) 删除正则表达式及测试文本。

(9) 打开“记事本”程序，输入 ABC，然后按 Tab 键(加入一个制表符)，再输入 DEF。

(10) 按 Ctrl+A 快捷键选择记事本中的文本，然后按 Ctrl+C 快捷键复制选中的文本。

(11) 在 Komodo Regular Expression Toolkit 的 Enter a string to match against 区域中，单击鼠标，然后按 Ctrl+V 快捷键把文本粘贴进去(包括一个制表符)。在 Komodo Regular Expression Toolkit 中，制表符显示为一个右箭头(如图 4-16 所示)。

(12) 在 Enter a regular expression 区域中，输入正则表达式模式 \s，并观察结果。

图 4-16 显示的是预料中的结果。注意，位于 ABC 和 DEF 之间的右箭头(它表示一个制表符)作为一个匹配项被突出显示。

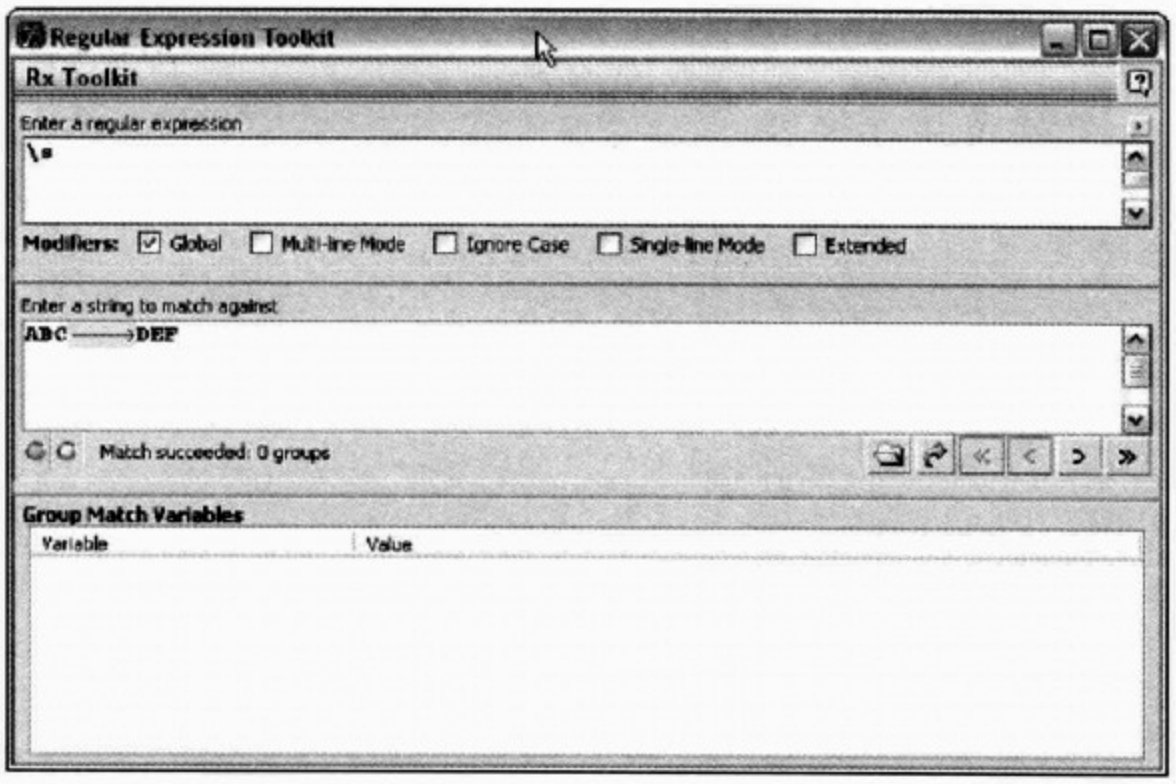


图 4-16

工作原理

\s 元字符可以匹配任何空白字符。

对于在第 2 步中指定的测试文本(包含一个换行符)，正则表达式引擎在到达 ABC 中 C 之后的位置以前一直没有找到匹配项。到那个位置后，接下来的字符是一个换行符。因为换行符与 \s 元字符匹配。于是，位于 ABC 中 C 之后的这个匹配的换行符立即在屏幕中以灰绿色突出显示出来。

对于在第 6 步中指定的测试文本(其中包含一个空格符)，正则表达式引擎在到达 ABC 中 C 之后的位置以前一直没有找到匹配项。到那个位置后，接下来的字符是一个空格符。因为空格符与 \s 元字符匹配。于是，位于 ABC 中 C 之后的这个匹配的空格符在屏幕中以灰绿色突出显示出来。

4.2.2 处理可选的空白符

在处理 HTML、XHTML 和 XML 文档时，经常需要考虑匹配可选的空白符。

为了演示下面的例子，假设在测试文本中只使用成对的双引号来界定 DateOfBirth 属性的值(XML 语法也允许使用成对的单引号)。

试一试：匹配可选的空白符

(1) 在 Windows 资源管理器中找到文件 CheckWhitespace.html，双击以在默认浏览器中打开这个文件。

(2) 单击 Click here to enter text 按钮，然后在打开的提示窗口中输入测试文本 <Person DateOfBirth="AnythingGoesHere" >。要保证在 = 的两侧没有任何空白字符。

(3) 单击 OK 按钮，然后在弹出的警告窗口观察显示的结果。图 4-17 显示的是在 Firefox 浏览器中执行第 3 步之后得到的结果。

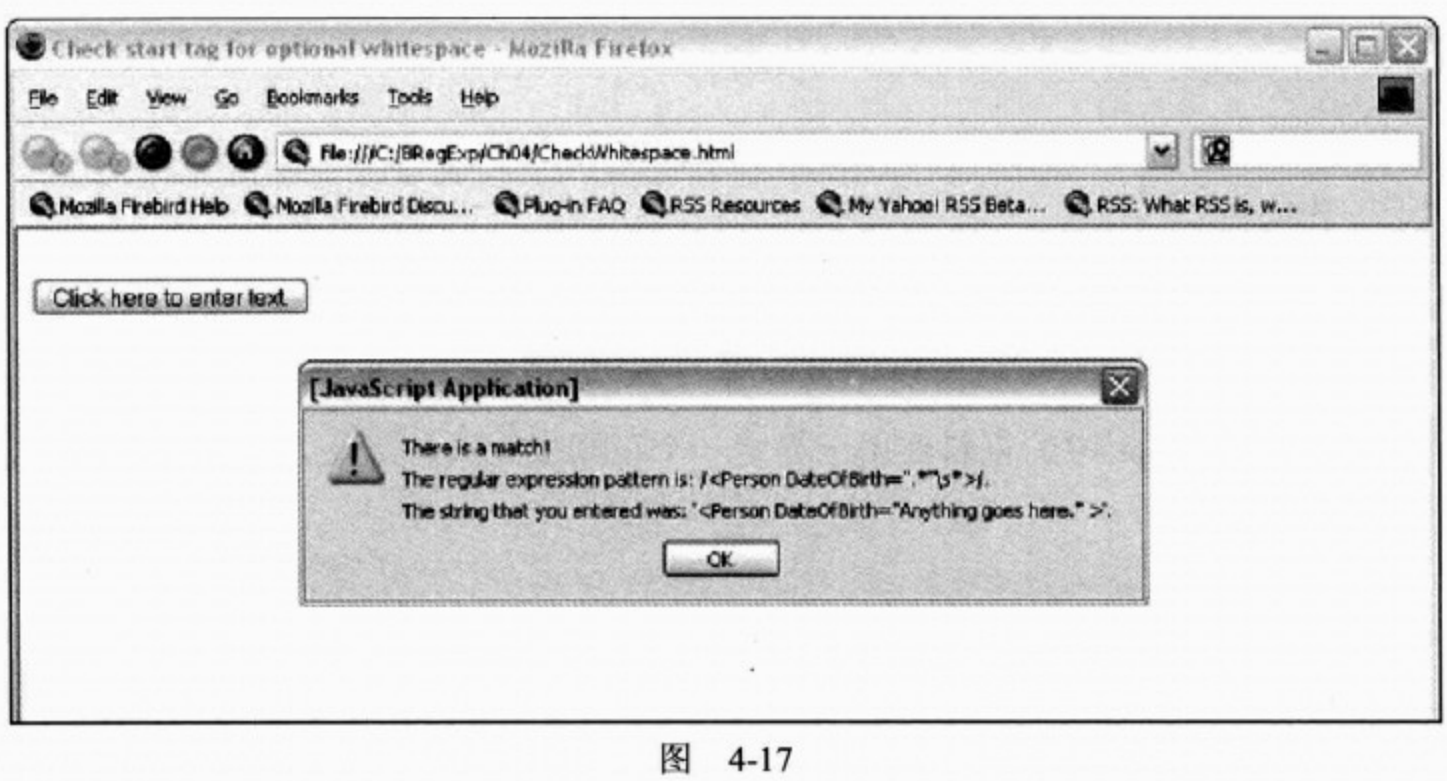


图　4-17

工作原理

测试文件 CheckWhitespace.html 的内容如下：

```
<html>
<head>
<title>Check start tag for optional whitespace</title>
<script language="javascript" type="text/javascript">
var myRegExp = /<Person DateOfBirth=".*"\s*>/;

function Validate(entry){
return myRegExp.test(entry);
} // end function Validate()

function ShowPrompt(){
var entry = prompt("This script tests for matches for the regular expression
pattern:\n " + myRegExp + ".\nType in a string and click on the OK button.", "Type
your text here.");
if (Validate(entry)){
alert("There is a match!\nThe regular expression pattern is: " + myRegExp + ".\n
```

```
The string that you entered was: '" + entry + "'.");
} // end if
else{
alert("There is no match in the string you entered.\n" + "The regular expression
pattern is " + myRegExp + "\n" + "You entered the string: '" + entry + "'." );
} // end else

} // end function ShowPrompt()

</script>
</head>
<body>
<form name="myForm">
<br />
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
</form>
</body>
</html>
```

注意声明变量 myRegExp 的那一行代码：

```
var myRegExp = /<Person DateOfBirth=".*"\s*>/;
```

记住在 JavaScript 中的正斜杠用于界定一个正则表达式。因此，要匹配的正则表达式模式如下所示：

```
<Person DateOfBirth=".*"\s*>
```

模式中的多数字符都是直接量字符。注意 DateOfBirth 属性的值用模式 .* 来匹配。换句话说，这个模式匹配零个或多个字符(匹配除了换行符之外的几乎任何字符)。在第二个双引号之后，模式 \s*(一个空白符后跟星号限定符)的含义是匹配零个或多个空白字符。

可以通过多次输入测试文本来测试这个正则表达式，即每次都在 > 字符前面加入不同数量的空格符。但是，这里无法直接测试 \s 元字符是否匹配制表符或者换行符。因为在输入制表符(按下 Tab 键)时光标焦点会从当前输入文本的区域中移开，而在输入换行符(按回车键)时就等于单击 OK 按钮。但是，可以把含有制表符或换行符的文本粘贴到这个对话框中。

之所以在这里可以使用句点元字符，是因为要测试的是用户输入的文本。在前面提到过，使用句点元字符存在贪婪匹配的问题，这样会导致匹配许多行甚至许多页的文本。

4.2.3 \S 元字符

\S 元字符用于匹配任何非空白字符。其他非英语语言中的字符也与\S 元字符匹配。但 \S 元字符不像句点元字符(根据设置，它会匹配除换行符之外的任何可能的字符)那样会匹配尽可能多的字符。

4.2.4 \t 元字符

\t 元字符匹配一个制表符。

如果不通过粘贴方式在文本中添加制表符，不可能在 Komodo Regular Expression Toolkit 中使用制表符。因为在 Enter a string to match against 区域中输入制表符会导致光标从文本区中移开，而不会向文本区中添加一个制表符。

测试文件 Tabs.txt 中包含由制表符分隔的一些单词：

```
Words  separated  by  tabs.
Some  more  tab-separated   words.
```

试一试：使用 \t 元字符

(1) 打开 OpenOffice.org Writer，然后再打开测试文件 Tabs.txt。

(2) 按 Ctrl+F 快捷键打开 Find & Replace 对话框，再选中 Regular expressions 和 Match case 复选框。

(3) 在 Search for 文本框中输入模式 \t，然后单击 Find All 按钮。观察与模式 \t 匹配的突出显示的文本，如图 4-18 所示。

工作原理

正则表达式引擎在 Words 中 W 之前的位置开始匹配。它依次检测每一个字符以判断该字符是否与 \t 模式(制表符)匹配。最后，文档中的所有制表符都被突出显示出来。

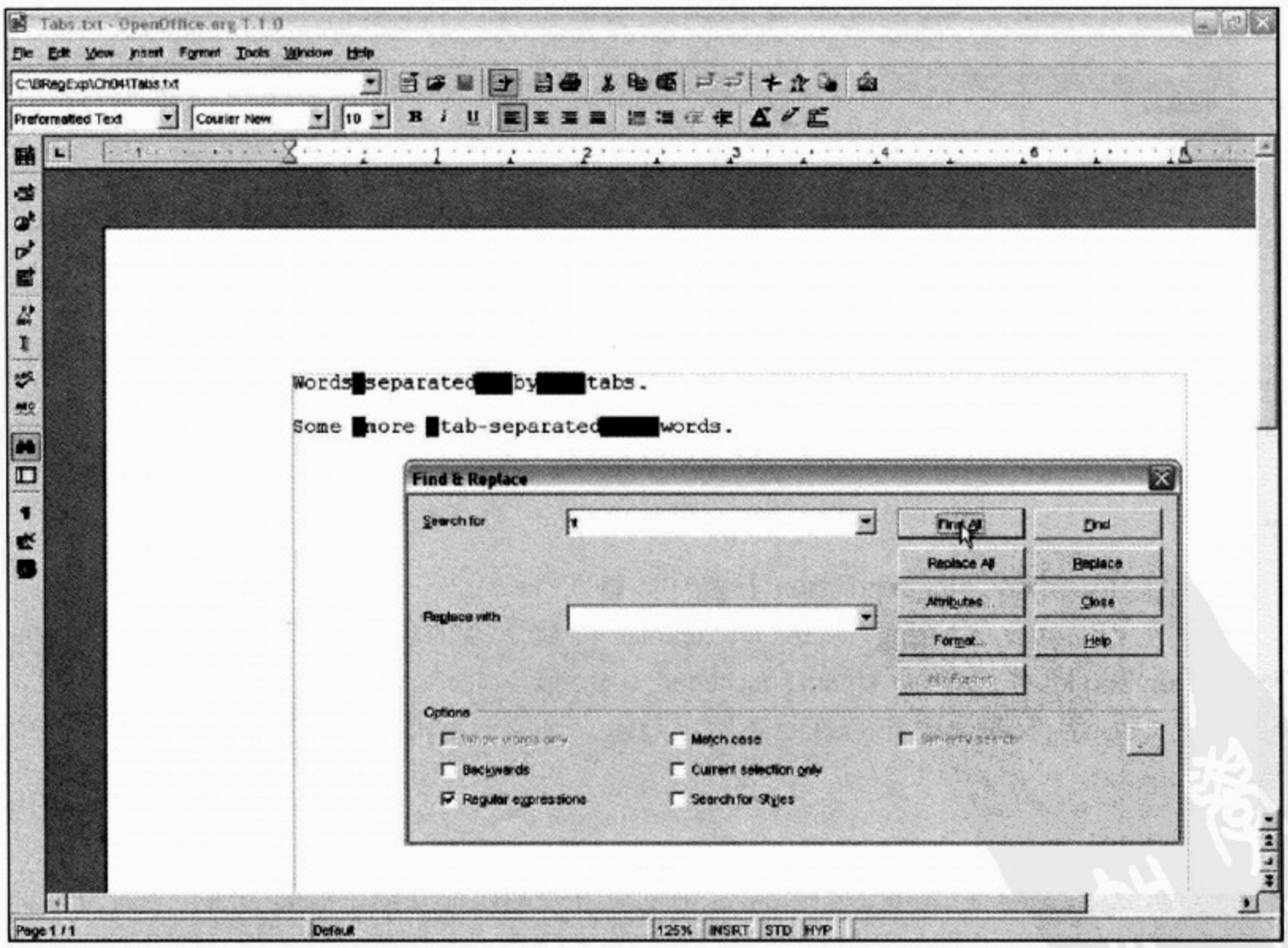


图　4-18

4.2.5 \n 元字符

\n 元字符匹配一个换行符——换句话说，它匹配当你按回车键时向文本文件中添加的那个字符。

试一试：使用 \n 元字符

(1) 打开 Komodo Regular Expression Toolkit，并清除任何现有的正则表达式模式和测试文本。

(2) 在 Enter a string to match against 区域中，输入下列测试文本：

```
Andrew
Watt
```

换句话说，输入 Andrew，按回车键，再输入 Watt。Enter a string to match against 区域中的文本如图 4-19 所示。

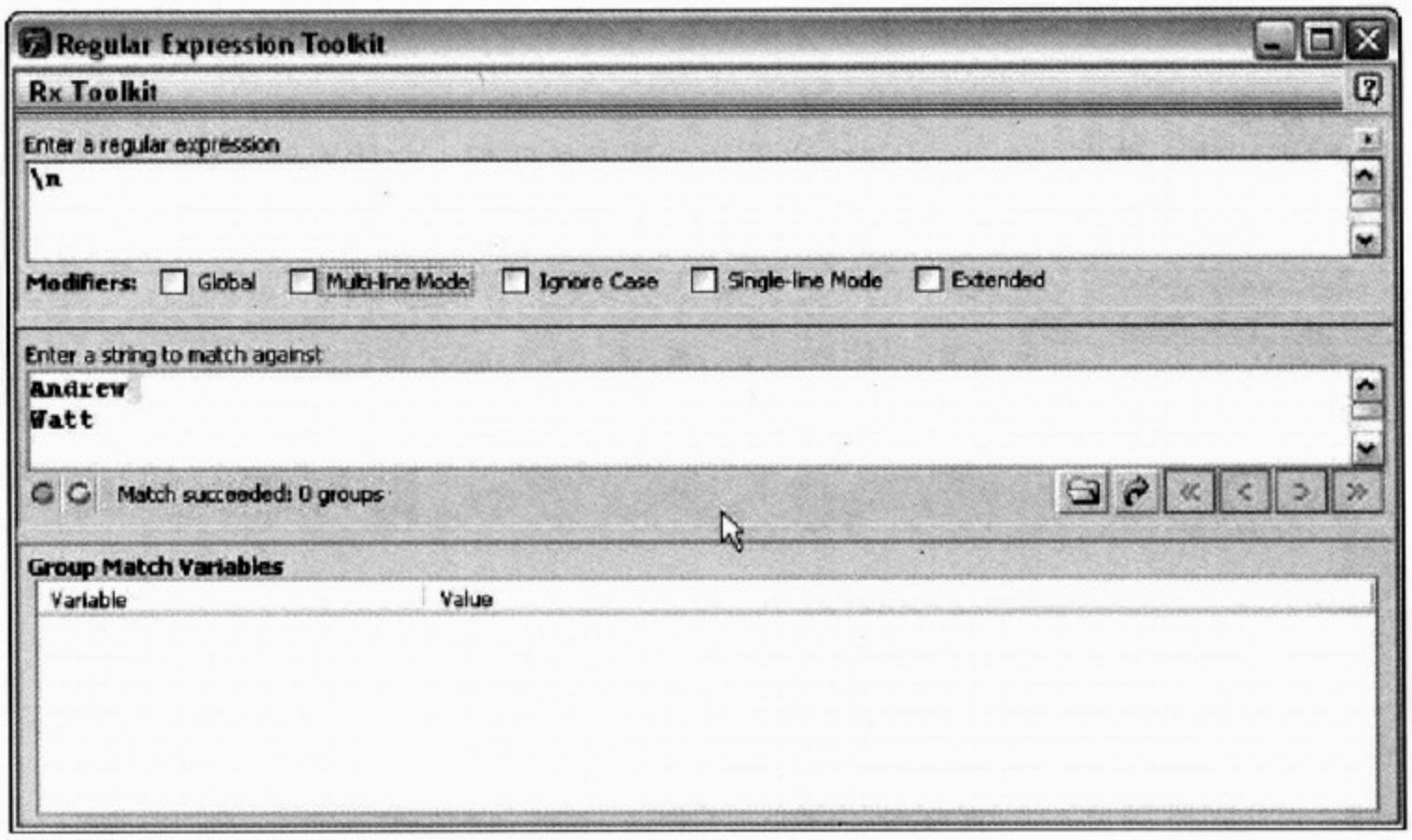


图 4-19

(3) 在 Enter a regular expression 区域中，输入 \n。

(4) 观察 Enter a string to match against 区域下方的区域。The Komodo Regular Expression Toolkit 中会显示 Match succeeded: 0 groups。此外，在第一行测试文本的末尾，会有一个空白字符(即换行符)以灰绿色突出显示。这表示正则表达式模式已经匹配了这个换行符。

工作原理

正则表达式引擎在 Andrew 中 A 之前的位置开始匹配并测试后面的字符是否是一个换行符，如果不是，会跳过刚测试过的字符并测试位于下一位置上的字符。如果下一个字

符是一个换行符，匹配成功。如前所述，在 Komodo Regular Expression Toolkit 中，匹配项是以灰绿色突出显示的。因为换行符是一个不可见的字符(不同于位于其后面新一行中的字符)，所以表面看去就像突出显示的是一个空字符。

试一试：Visual Studio 中的 \n 元字符

Microsoft Visual Studio 2003 编辑器也支持 \n 元字符。如果电脑中安装有 Visual Studio 2003，可以按照下列步骤进行试验：

(1) 启动 Microsoft Visual Studio 2003，并打开 Person2.xml 测试文件。

(2) 使用 Ctrl+F 快捷键打开 Find 窗口，选中 Use 复选框。

(3) 在 Find 窗口左下角的下拉菜单中选择 Regular expressions 选项。

(4) 在 Find what 文本框中输入模式 \n，并单击 Find Next 按钮。

(5) 观察突出显示的第一个匹配项。如果光标初始时位于文本的开始处，那么第一个与模式 \n 匹配的换行符应该位于第一行的末尾。

图 4-20 显示的是经过前面几个步骤后，Visual Studio 2003 编辑器中的匹配结果。

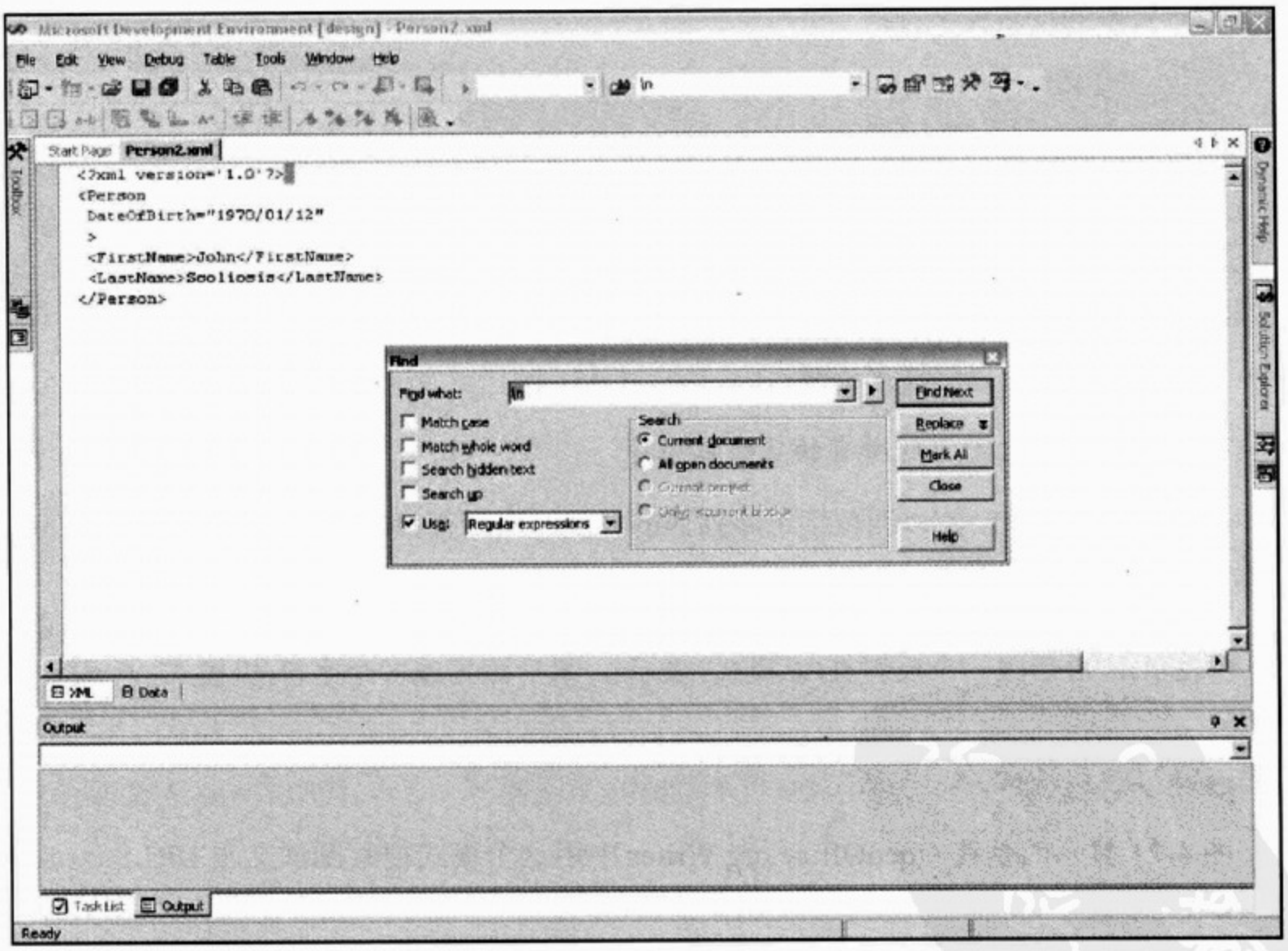


图　4-20

工作原理

在光标位于 Person2.xml 文件开始处的情况下，Visual Studio 2003 中的正则表达式引

擎会依次检查每一个字符看是否与模式 \n 匹配。结果第一个匹配项被突出显示在测试文本第一行的末尾。

4.2.6 转义字符

对于有些字符，比如.字符，如果要让它匹配对应的直接量字符，则必须要对它进行转义。在本章前面也使用过 \. 元序列来匹配一个句点字符。

另外一个需要特别对待的字符是反斜杠。

4.2.7 查找反斜杠

我们经常会在一个正则表达式模式中看到反斜杠(\)与另一个字符搭配使用，从而对该字符进行转义，以免它被当做元字符来解释。同样，也经常会看到反斜杠与另外一个字符共同组成一个字符对，而这个字符对在正则表达式模式中被解释为元字符。因此，现在的问题是如何选择一个反斜杠的直接量呢？

假设一个文档中包含着许多 URL。URL 只包含正斜杠字符(/)，而不会包含反斜杠。例子文档 URLS.txt 中就包含着一些 URL，其中一些正确地使用了正斜杠字符，而有的 URL 中还带有残留的反斜杠字符：

```
http://www.w3.org/
http://www.amazon.com/
http://www.w3.org\tr/
http://www.XMML.com\default.svg
http://www.wiley.com/
http:\\www.wrox.com/
http://www.example.org/
```

这个问题定义可以这样来描述：

URLS.txt 查找文本中的所有的反斜杠字符。

假设所有的 URL 都以四位字符序列 http 开头，而要查找的是这个初始的字符序列之后跟着任何反斜杠的情况。

下面的正则表达式模式会找到所有初始的四个字符序列后跟零个或多个字符(以模式 .* 表示)，再后跟一个反斜杠(以模式\\表示)，最后跟零或多个字符(以模式 .* 表示)的情况。

```
http.*\\.*
```

图 4-21 显示的是在 OpenOffice.org Writer 中用这个模式搜索测试文本 URLS.txt 后的结果。

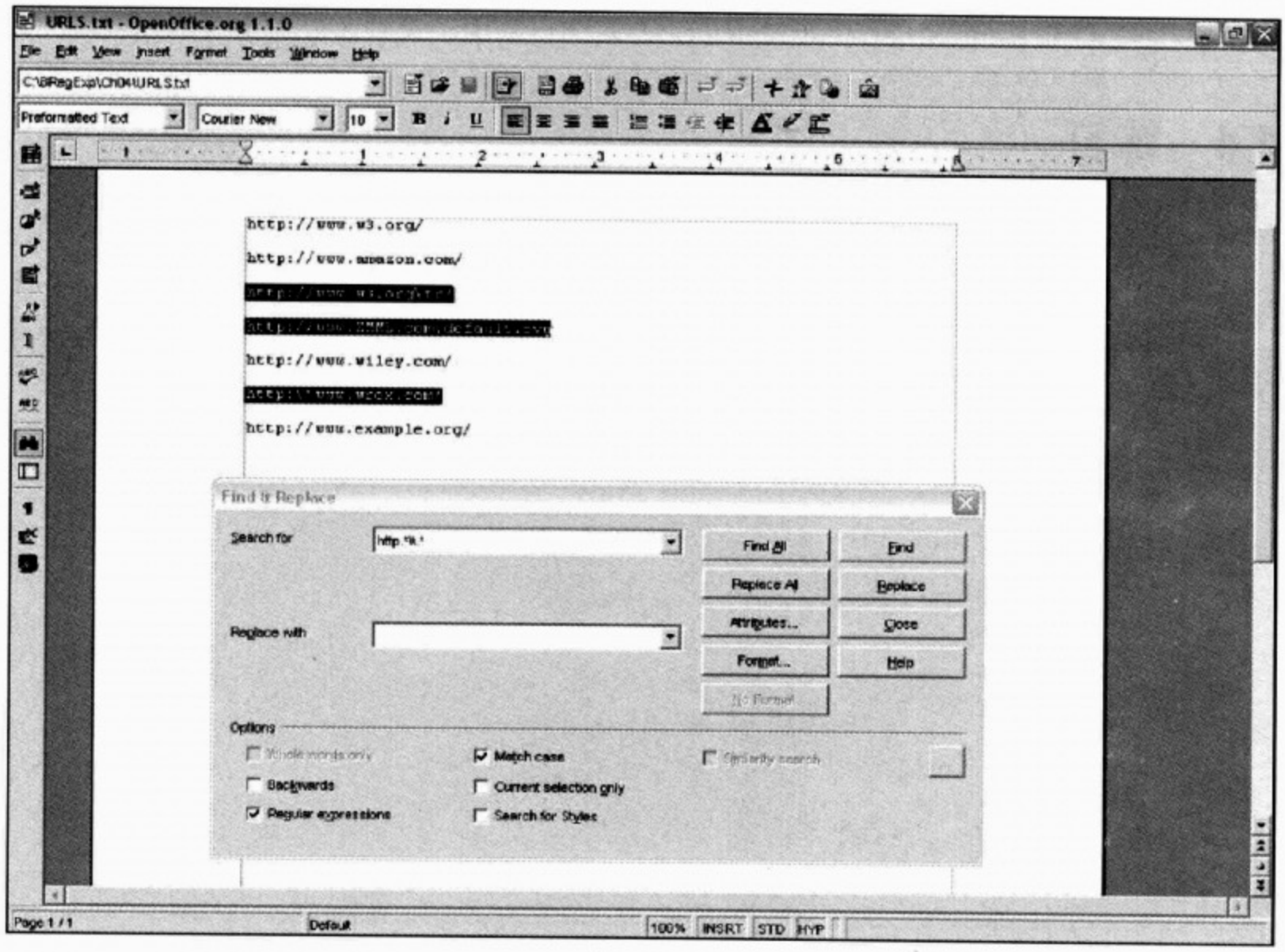


图　4-21

4.3　修饰符

修饰符——顾名思义，是用于修饰如何完成搜索的符号。这里，以演示正则表达式例子工具的方式来描述修饰符的概念。类似的修饰符在本书后面介绍的编程语言中同样有效。

4.3.1　全局搜索

全局搜索修饰符用于决定是查找一个匹配项，还是查找所有匹配项。当在搜索替换功能中使用全局修饰符时，它会决定是替换目标文本中的一个匹配项(在全局修饰符关闭时)，还是替换所有的匹配项(在全局修饰符打开时)。每种编程语言都有自己表达全局搜索的语法形式。

4.3.2　不区分大小写的搜索

在第 1 章的 Star Training Company 示例中示范了由于不区分大小写的搜索导致的意想不到的结果。然而，在有些情况下可能会需要用不区分大小写的搜索。例如，对于必须使用大写字母字符的零件编号目录，通过区分大小写的搜索可以识别出不正确地使用了小写字母的零件编号。而如果使用不区分大小写的搜索，则不会发现这种不规范的零件编号。

在 OpenOffice.org Writer 中，Match case 复选框就用于设置区分大小写的搜索。

4.4 练习

下面的问题和练习旨在帮助测试本章所学的内容。

1. .(句点)元字符和 \w 元字符之间有什么区别？

2. 请修改 CheckWhitespace.html 中的正则表达式模式，使其匹配在分隔 DateOfBirth 属性与值的 = 字符两侧均存在空白字符的字符串。

第 5 章

字 符 类

字符类用于匹配一个字符集合中的任何一个字符。例如，当需要在一个零件目录中匹配某些零件或者在一个员工名单中匹配某些名字的时候，都可能会用到字符类。

有一些字符类对应使用广泛的字符集合。像字符类 [A-Z] 对应大写的 ASCII 字符，而字符类 [0-9] 则匹配一个数字。

在本章中将学习以下内容：

- 字符类的工作原理
- 如何对字符类应用限定符
- 如何在字符类中使用范围
- 如何对字符类取反

5.1 字符类概述

字符类是一些字符的无序组合，正则表达式模式可以从这个组合中挑选出一个字符来完成匹配。如果对于当前要匹配的字符而言，字符类中指定的任何字符都不能与之匹配，则该字符类匹配失败。

下面包含字符类 [yi] 的模式会匹配 Smith 或 Smyth 姓氏，因为这两个姓的第三个字符都包含在指定的字符类中。

```
Sm[yi]th
```

当字符类不带有关联的限定符时，字符类只会指定其中一个字符用于匹配。因此，下面的模式可能会匹配 pear、peer 或 peir，但不会匹配 per。因为 per 中不包含字符类中所包含的任何一个字符。

```
pe[aei]r
```

有时可能会遇到用术语字符集来表示字符类的情况。但术语字符类使用得更广泛，本书将使用字符类这个术语。

体会一下下面的问题定义：

匹配一个大写的 A，后跟一个大写的 B，后跟一个数字 1 或者一个数字 2，后跟另外一个数字的情况。

要选择定义中给出的零件编号 AB10~AB29，可以使用下面的模式：

```
AB[12][0123456789]
```

其中第一个字符类 [12]，表示匹配的字符序列中的第三个字符可以是数字 1，也可以是数字 2。而字符类 [0123456789] 则表示该字符序列中的第四个字符必须是数字 0~9。

示例文件 ABPartNumbers.txt 中包含的数据内容如下：

```
AB31

AB2D
AB10

AB18

AB44

AB29

AB24
```

试一试：使用字符类

(1) 打开 OpenOffice.org Writer，并打开文件 ABPartNumbers.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中输入模式 AB[12][0123456789]，并单击 Find All 按钮。

(5) 观察示例文本，如图 5-1 所示，看其中哪个字符序列被突出显示了。注意，前两个字符序列都没有匹配成功。

工作原理

正则表达式引擎在 AB31 中 A 之前的位置开始匹配。它尝试将模式中大写的 A 和示例文本中大写的 A 进行匹配，匹配成功。接着，又尝试匹配模式中的第二个字符——大写的 B 与示例文本中的第二个字符匹配，匹配成功。然后，它尝试将模式中的第三个组件(是一个字符类[12],而不是一个单独的直接量字符)与字符序列中的第三个字符——数字 3——进行匹配。匹配没有成功。因为模式中的一个组件不能匹配，所以整个模式匹配失败。

同样，字符序列 AB2D 也匹配失败。这个序列中的前两个字符与模式中的前两个字符 AB 匹配。第三个字符是数字 2，也和模式中的字符类 [12] 匹配。但是，它的第四个字符是 D，不能与字符类 [0123456789] 匹配。由于模式中的一个组件匹配失败，所以整个模式也匹配失败。

不过，字符序列 AB10 是匹配的。因为这个序列中的第一个字符 A 与模式中的第一个字符 A 匹配。第二个字符 B 与模式中的第二个字符 B 匹配。第三个字符是数字 1，也和模式的第三个组件——字符类[12]匹配(因为数字 1 包含在这个字符类中)。最后，序列中的第四个字符——数字 0 又与字符类 [0123456789] 中的 0 匹配。

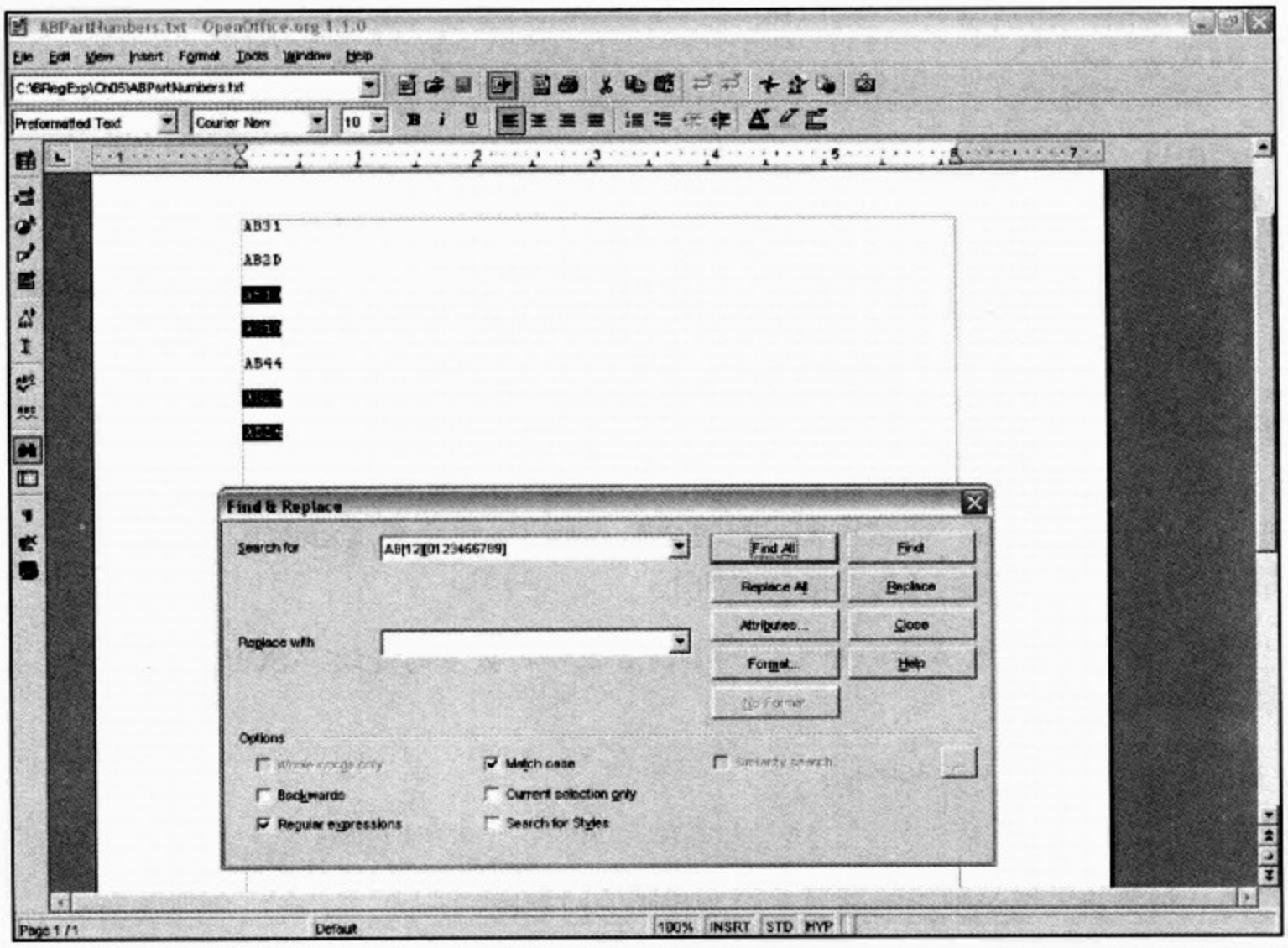


图 5-1

5.1.1 在两个字符中选择

可以使用字符类进行简单的二选一。不过，在这种情况下仅仅使用圆括号来包括两个选项也可以达到同样的效果。

有关圆括号以及如何将其用于交替性选择的内容将在第 7 章中介绍。

例如，从测试文件 People.txt 包含的下列名单中选择人名：

```
Cardoza, Fred
Catto, Philipa
Duncan, Jean
Edwards, Neil
England, Elizabeth
Main, Robert
Martin, Jane
Meens, Carol
Patrick, Harry
```

```
Paul, Jeanine
Roberts, Clementine
Schmidt, Paul
Sells, Simon
Smith, Peter
Stephens, Sheila
Wales, Gareth
Zinni, Hamish
```

按照以上数据显示，每个名字单独位于一行，而且姓在前面，然后是一个逗号和一个空格，最后是名字。如果要选择其中以 C 或 D 开头的姓，可以使用下面的问题定义：

匹配一个大写的 C 或者一个大写的 D，其后跟任何数量的连续 ASCII 小写字母字符。

下面的模式可以作为表达以上问题定义的一个解决方案：

```
[CD][a-z]+
```

但是，这个模式仍然不够精确。如果用它来测试 Roberts, Clementine，会发现它匹配其中的名字 Clementine，而这里要匹配的是姓。所以，这个模式还必须再具体一些。在这种情况下，我们把问题定义简单地修改如下：

匹配一个大写的 C 或者一个大写的 D，后跟任何数量的连续 ASCII 小写字母字符，再后跟一个逗号。

用更加具体的模式来表达是：

```
[CD][a-z]+,
```

另一种解决方案是使用圆括号来表达相同的问题定义：

```
(C|D)[a-z]+,
```

现在来试一试。

试一试：选择特定的姓

(1) 打开 OpenOffice.org Writer，并打开文件 People.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中输入模式[CD][a-z]+,，并单击 Find All 按钮。

(5) 观察结果。图 5-2 中显示选择了 People.txt 中所有以 C 或 D 开头的三个姓。注意，因为模式中包含逗号，所以测试文本中的 Meens，Carol 和 Roberts, Clementine 没有匹配。

(6) 把正则表达式模式中末尾的逗号删除。

(7) 单击 Find All 按钮，并观察匹配结果。注意，在删除结尾逗号的情况下，字符序列 Meens, Carol 和 Roberts, Clementine 是匹配的。

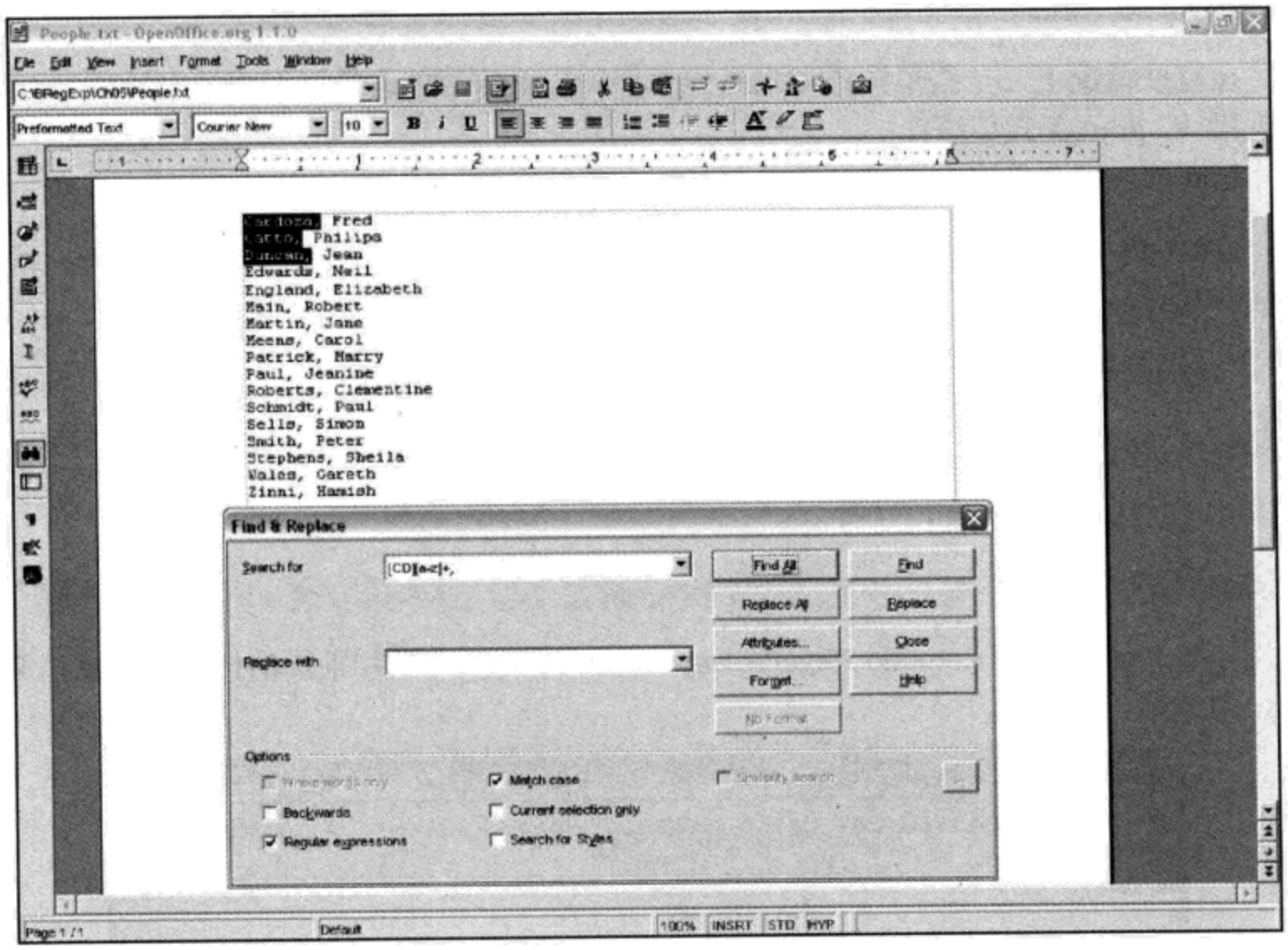


图 5-2

工作原理

当正则表达式引擎开始匹配时，它会从 Cardoza, Fred 中开头字母 C 之前的位置开始。首先，它尝试将模式的第一个组件——字符类 [CD]与测试文本中的第一个字符进行匹配，是一个大写的 C，结果匹配。接着，它尝试将模式的第二个组件——模式 [a-z]+(一个或多个小写的 ASCII 字符)与字符序列中第二个以及后面的字符进行匹配。a、r、d、o、z 和 a 中的每一个字符都是匹配的。虽然最后的逗号不与模式 [a-z]+ 匹配，但它却和正则表达式模式的最后一个组件——直接量逗号——匹配。因此，正则表达式模式的每个组件都是匹配的。大写的 C 匹配 [CD]，小写字符序列 ardoza 匹配 [a-z]+ 而最后的逗号与正则表达式模式中的逗号匹配。

当正则表达式引擎到达测试文本 Meens, Carol 的 Carol 中的 C 之前的位置时，它会像前面一样尝试把大写的 C 与字符类 [CD] 匹配。结果匹配成功。而后面的文本 arol，同样也和模式 [a-z]+ 匹配。但是，却没有字符能够匹配正则表达式模式中结尾的逗号。所以，整个模式也就没有匹配。

字符类具有很大的灵活性，而且能够自如地修改和扩充。比如，可以通过把模式修改为如下所示的样式，使其匹配姓氏以 C、D 或 S 开头的人：

```
[CDS]\w+,
```

当然，也可以使用圆括号，像下面这样：

```
(C|D|S) \w+,
```

在有些情况下，一些单词通常只有一个字符与正确的单词拼写不同。比如说 grey(英式英语)和 gray(美式英语)。

这时的问题定义可以描述如下：

匹配一个小写的 g，后跟一个小写的 r，后跟一个小写的 e 或小写的 a(二选一)，最后跟小写的 y。

一种用于表达这个问题定义的模式是：

```
gr[ae]y
```

也可以写成下面这样：

```
gr[ea]y
```

图 5-3 显示了在 Komodo Regular Expression Toolkit 中使用前面的模式匹配测试文本 grey 的结果。

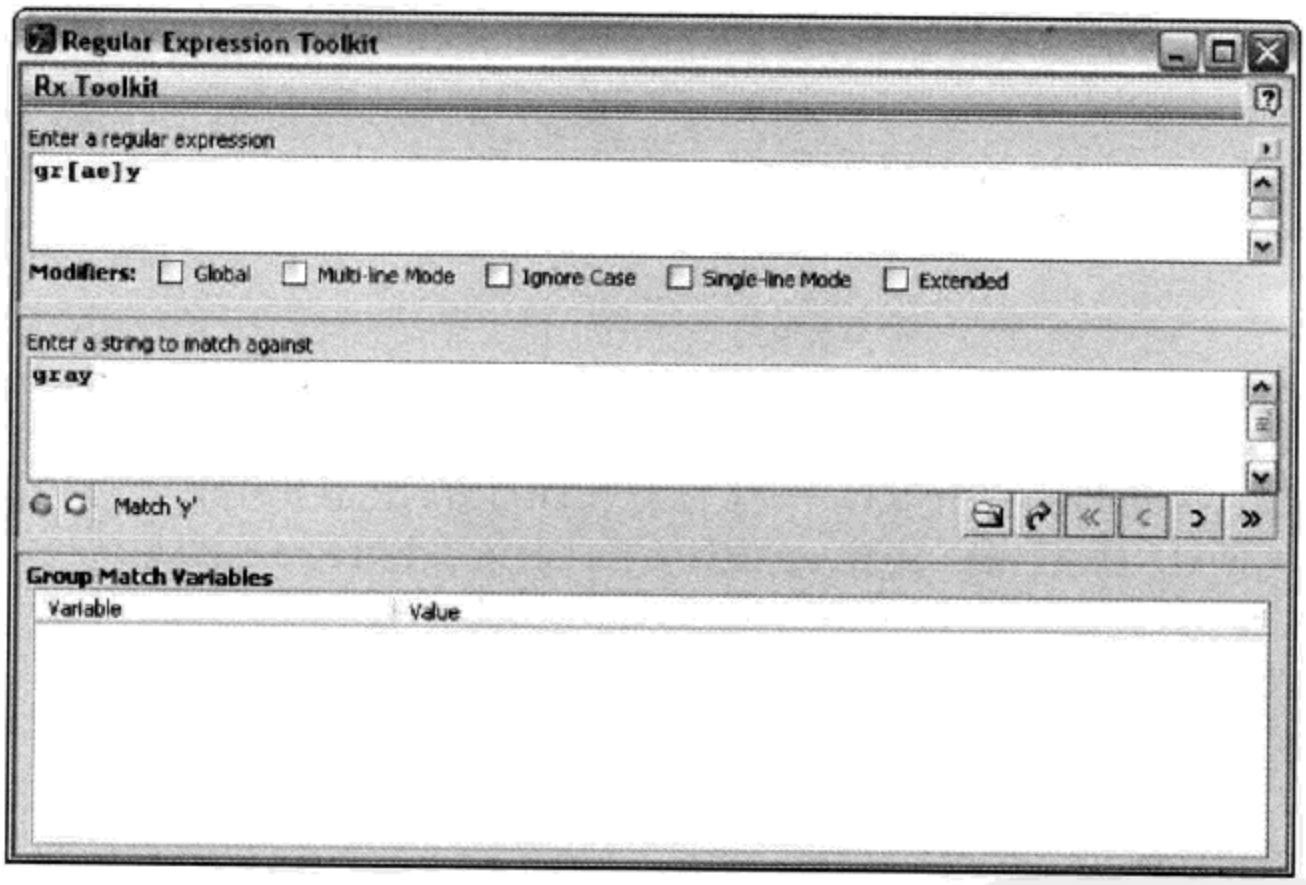


图 5-3

模式 gr[ae]y 的前两个字符匹配直接量字符 g 和 r。而字符类 [ae] 则匹配测试文本的第三个字符——小写的 a，模式中最后的 y 与 gray 中最后的 y 匹配。

5.1.2 对字符类应用限定符

到现在为止，我们使用的都是没有添加限定符的简单字符类。这种情况下的字符类就和一个单个字符一样，因为没有限定符意味着字符类中的字符只能有一个参与匹配。

事实上，?、* 和 + 以及大括号({})限定符语法也可以用于限定一个字符类，就如同它们限定单个字符一样。

例如，可以在下面的模式中使用{2} 限定符来匹配测试文本：

```
[AB]{2}[12][0-9]
```

图 5-4 显示了在 OpenOffice.org Writer 中单击 Find All 按钮后的结果。

从图中可以看出，示例文本中 AB10 和 AB29 之间的所有零件编号都匹配。

不过，对字符类 [AB] 使用限定符 {2} 有可能会导致错误的结果。例如，测试文本中包含零件编号 AA23 或 BB19，那么它们也都会匹配，但根据本章前面的问题定义，它们不是我们想要的匹配项。

先来看一下它的工作原理。如果正则表达式引擎处于 BB19 中 B 之前的位置，它会尝试将字符类 [AB] 与大写的 B 进行匹配。结果匹配。然后，由于有一个限定符 {2}，所以它会再次将字符类 [AB] 与第二个 B 进行匹配。这次也匹配。然后，它会用字符类 [12] 来匹配数字 1。匹配成功。最后，字符类 [0-9] 又和数字 9 匹配。因为模式的所有组件都匹配，所以这个测试文本与模式完全匹配。

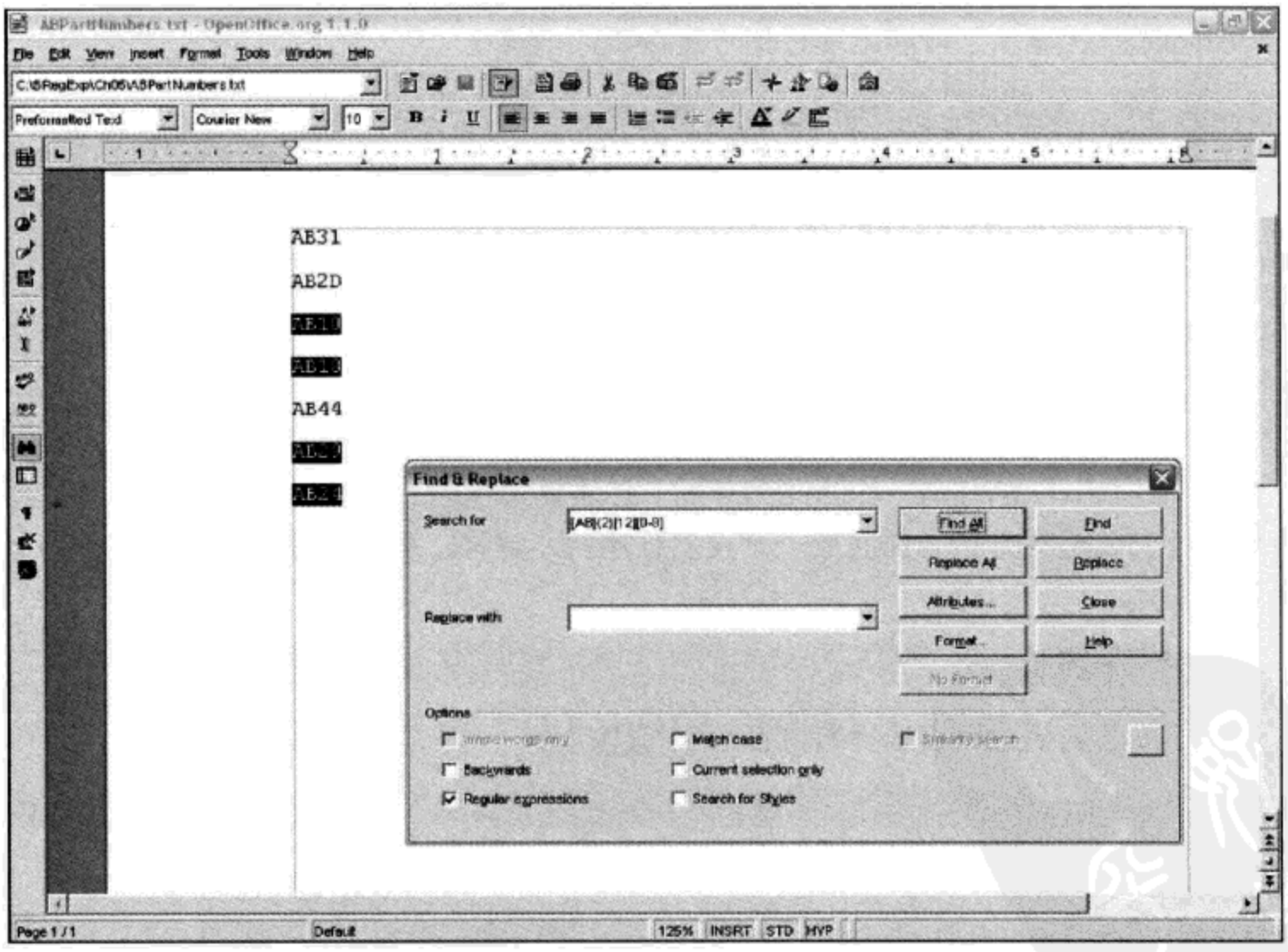


图 5-4

1. 在字符类中使用 \b 元字符

在第 6 章中，我们会介绍一个元字符，它在字符类内部和外部具有不同的含义。这个元字符就是 \b。

在字符类的内部，\b 元字符表示一个回格符。而在字符类的外部，\b 元字符表示一个词边界——至少在有些正则表达式实现中是这个含义。

有关在字符类外部使用 \b 元字符的内容将在第 6 章中介绍。

\b 元字符并不是唯一一个在字符类内外部具有不同含义的元字符。像连字符(-)和脱字符(^)，它们在字符类内部都有特殊的含义，这一点我们在本章后面会讨论到。此外，在字符类内部 $ 字符只简单地匹配它自身。但在字符类的外部，$ 元字符会匹配一个位置而非一个字符，这一点将在第 6 章讨论。

2. 选择方括号直接量

可能读者已经注意到，在定义字符类时我们使用的是 [和] 元字符。为此，我们不能同时使用它们来匹配其自身直接量(即匹配 [和] 字符自身)。下面的文本文件 SquareBrackets.txt 显示了可能出现方括号直接量的情况：

```
These are alphabetic characters [A to Z and a to z].

myVariable = myArray[3];

Character[7]

The first five characters in the ASCII character set after uppercase Z are [, \, ],

^, and _.
```

如果想选择其中的任何一个方括号字符，必须要对相应的方括号字符进行转义。转义，其简单的含义就是在方括号前面放一个反斜杠字符。这样，要选择左方括号字符([)应该使用下面的模式：

```
\[
```

而使用下面的模式可以选择右方括号字符(])：

```
\]
```

图 5-5 显示了在 OpenOffice.org Writer 中使用 \[模式选择左方括号字符的用法。

反斜杠(\)字符可用于对大多数元字符进行转义。要使用反斜杠直接量，同样也必须进行转义。所以，模式 \\ 会选择一个反斜杠字符。

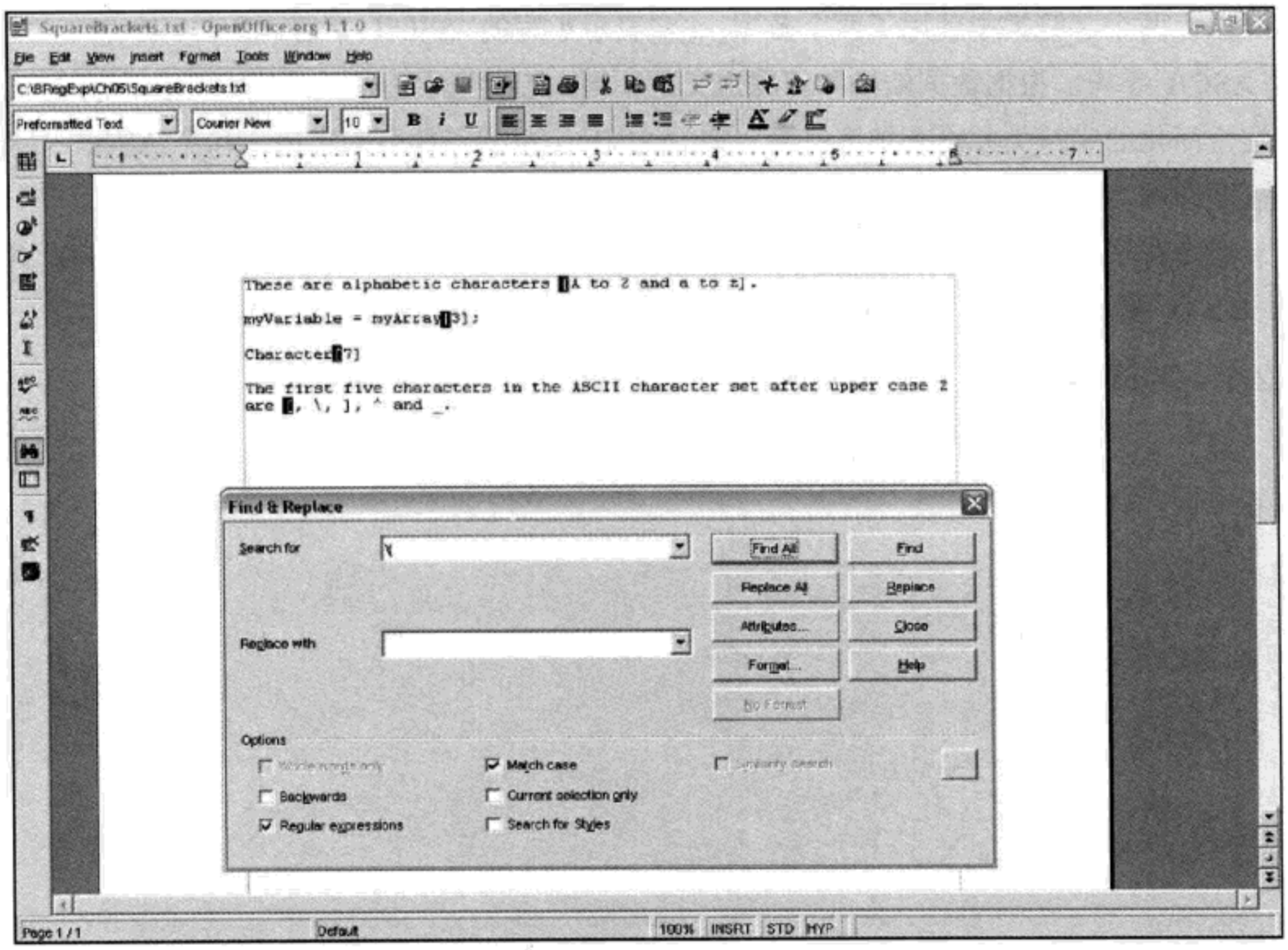


图 5-5

5.2 在字符类中使用范围

在字符类中使用范围不仅会使得模式更加简洁，也能避免因罗列一大堆字符而出错的可能性。例如，如果要选择英语中使用的所有字母字符(包括大写和小写)，可以使用下面的字符类：

```
[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]
```

但是，通过使用范围也可以表达同样的字符类：

```
[a-zA-Z]
```

或

```
[A-Za-z]
```

以上两个字符类中使用了两个范围：a-z 表示小写的 ASCII 字符，而 A-Z 表示大写的 ASCII 字符。根据数据结构和问题定义的不同，想要把它们简化成 [A-z]是不行的。因为这个简化的字符类并没有把字符限定在 ASCII 字母字符的范围之内。由于 ASCII 字符的排序问题，有些非字母字符(如方括号)的位置处于大写的 Z 后面和小写的 a 前面。

通过在字符类中使用范围，可以自定义我们自己的字符集。比如说，如果要选择以 A、B、C、D 或 E 开头的姓，那么可以用字符集 [A-E] 来匹配姓氏的第一个字母。

5.2.1 字母字符范围

来看一个在字符类中使用范围的简单例子，其中使用的测试文件是 Light.txt，其内容如下：

```
fight
light
sight
right
night
delight
plight
tight
fights
height
lightning
might
quite
rights
weight
bite
quite
```

如果要选择其中的字符序列 right、sight 和 tight，可以使用下面的正则表达式：

```
[rst]ight
```

这个模式在字符类中简单地枚举出了三个直接量字符。

不过，我们发现要选择的三个字符序列中的首字母在字母表中是连续的。所以，可以在这个字符类中使用范围：

```
[r-t]ight
```

模式 [r-t] 是一个字符类：其中 r 是字符类中的一个直接量字符，而后面的连字符是一个表示范围的元字符，最后的 t 也是一个直接量字符。图 5-6 显示这个模式选择了字符序列 right、sight 和 tight。

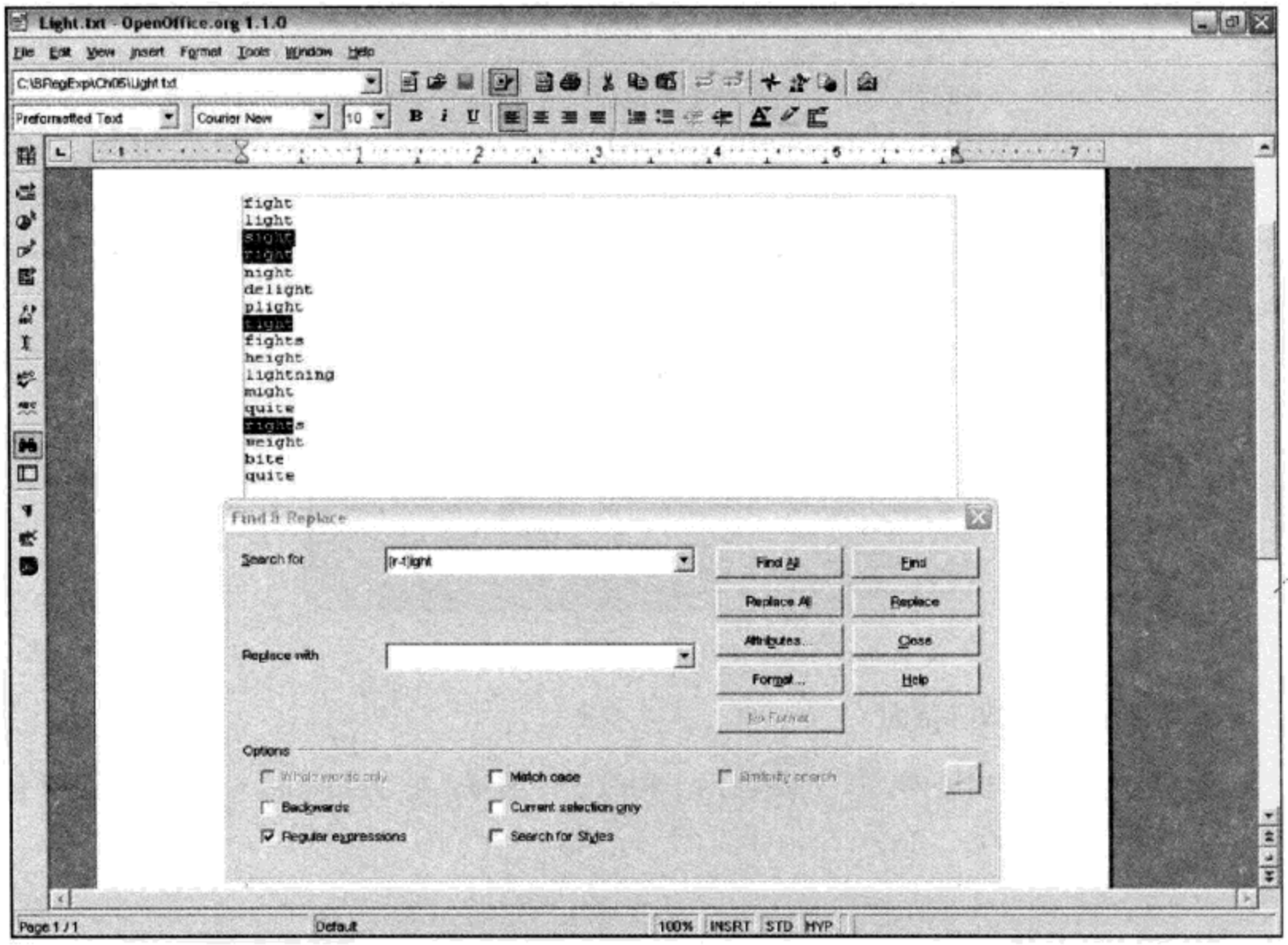


图 5-6

1. 慎用[A-z]

如果要选择英语语言中的字母字符，有时可能会尝试使用下面的正则表达式：

```
[A-z]
```

而且，将它等价于下面的字符类：

```
[A-Za-z]
```

但事实并非如此。因为在 ASCII 和 Unicode 字符集中，大写和小写的字母字符并不是连续存在的。如图 5-7 所示，其中显示的是 Windows 字符映射实用程序的界面。

我们看到，在 Z 的后面和小写 a 的前面有 6 个字符，并且全都不是字母字符。它们分别是[(左方括号)、\(反斜杠)、](右方括号)、^(脱字符)、_(下划线)和 `(重音符)。在图 5-7 中光标所在位置的字符是]。

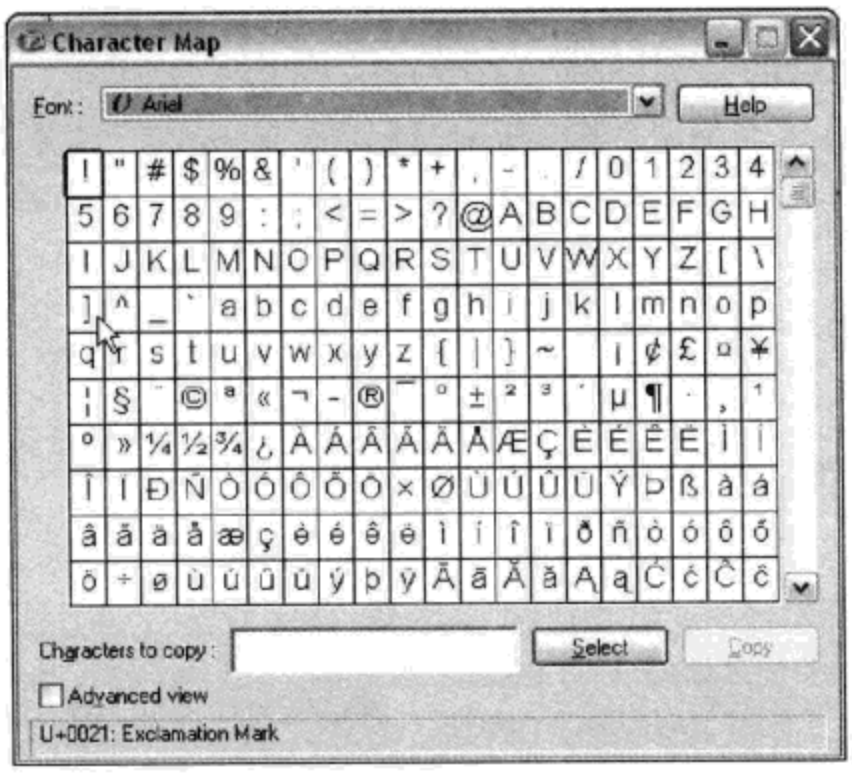


图 5-7

2. 字符类中的数字范围

数字是被广泛使用的字符，比如说用于构成零件编号。在 PartNums.txt 测试文件中包含以下一些零件编号：

```
DB992
AX891
FG339
GE919
GE442
FG935
DB882
AX717
AX803
FG919
GE604
```

注意，零件编号由两个字母字符后跟三个数字组成。其中，前面的两位字母字符分别是 AX、DB、FG 和 GE。而用于数字部分的字符则可以是从 0 到 9 的所有数字。

在支持 \d 元字符的实现中，可以使用 \d 来表示数字的类。而另一个方法是使用字符类 [0-9]。

试一试：用字符类匹配数字

如果假设前面两个字母大写，那么可以使用字符类 [ADFG] 来表示第一个字符，而用字符类 [XBGE] 来表示第二个字符。同样，每一个数字都可以使用字符类 [0-9] 来匹配。

所以，最终的模式如下：

```
[ADFG][XBGE][0-9]{3}
```

(1) 打开 OpenOffice.org Writer，然后打开测试文件 PartNums.txt。
(2) 按快捷键 Ctrl+F 打开 Find & Replace 对话框。
(3) 选中 Regular expressions 和 Match case 复选框。
(4) 在 Search for 文本框中输入模式[ADFG][XBGE][0-9]{3}，然后单击 Find All 按钮。
(5) 观察结果，如图 5-8 所示(所有零件编号都突出显示)。

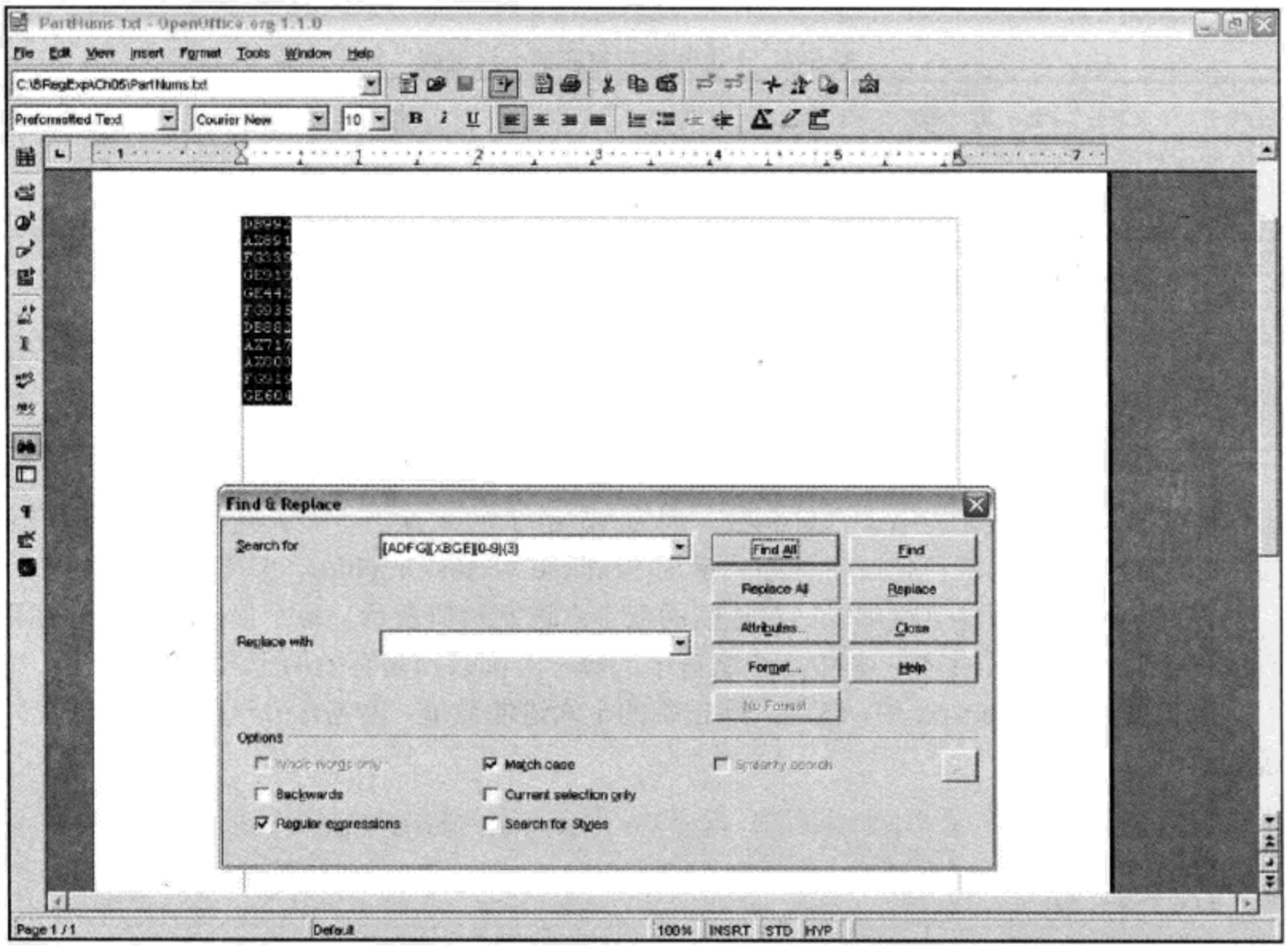


图 5-8

工作原理

分析第一个零件编号 DB992 匹配的过程。正则表达式引擎会从 D 之前的位置开始匹配。首先，它尝试匹配模式的第一个组件、字符类 [ADFG]，DB992 的第一个字符是 D，所以匹配。接着，它尝试匹配模式的第二个组件、字符类 [XBGE]，它与测试文本的第二个字符 B 匹配。然后，又将字符类的第三个组件、带有限定符 {3} 的字符类 [0-9] 与三个连续的数字进行匹配。由于测试文本中的第三、四、五个字符分别是数字 9、9、 2，所以匹配成功。因为正则表达式模式的所有组件都匹配，所以整个模式匹配。

但是，如果不想匹配假设的字符序列 AG123，那么使用前两个字符类将会导致错误的结果。因为模式 [ADFG][XBGE] 也会匹配 AG123 的前两个字符。

3. 十六进制数字

字符类对于识别十六进制数值而言非常有用。十六进制数是基于 16 而不是通常的 10 来表示数字的值。所以，为了表示数值 0~15，会使用字母字符 A~F 或 a~f (大小写都可以)来表示 10~15。表 5-1 中显示的是 10~15 的十进制数和十六进制数，如果对十六进制数不熟悉可以参考一下。其中，0~9 这 10 个数值在十六进制中与十进制中是一样的。

表 5-1　十进制数与对应的十六进制

十　进　制	十 六 进 制
10	A 或 a
11	B 或 b
12	C 或 c
13	D 或 d
14	E 或 e
15	F 或 f

许多计算机内部都使用十六进制数，因为 16 是 2 的 4 次方。十六进制数也用于定义 HTML/XHTML 中的颜色值或可伸缩矢量图(Scalable Vector Graphics，SVG)中的属性值。

在 SVG 中，颜色值通常以三对连续的双十六进制数值表示。每个值的字符序列都会写成一个#，后跟六个字符，其中六个字符中的每一对都必须是有效的十六进制数值。

测试文件 Numbers.txt 中，包含一些正确的十六进制数值，也包含另一些不正确的十六进制数值。

```
#DE88D9
#DE88D9
#DG3399
#0099FF
#99FG00
#CCCCCC
#669933
#66330
#8i8824
#902332
#8F8F8F
#2099CC
#88CCFF
#CFE
#994488
#CFEE
```

这些字符序列中的一些值包含着超出 0~9 和 A(a)~F(f)范围的字符，而另一些则没有包含六个字符或数字。

可以像下面这样来表达问题定义：

匹配一个直接量字符#，后跟六个连续的字符，其中每个字符都以 0~15(十进制)的十六进制数——即 0~F (十六进制)来表示。

下面的正则表达式模式将会匹配有效的字符序列：

```
#[0-9a-fA-F]{6}
```

图 5-9 显示的是将这个模式应用到 Numbers.txt 的结果。

除了匹配有效的十六进制数之外，你可能也希望选择由于种种原因导致的无效数值。在本章后面，当学习了对字符范围取反的概念后，就会知道如何完成这样的匹配了。

4. IP 地址

另一个可以通过正则表达式来匹配的例子是 IP 地址。IP 地址用于在万维网中查找服务器。当在浏览器中输入 www.WhereIWantToGo.com 时，这些字符序列就会被转换成 IP 地址，格式像 123.2.234.23 这样。其中，在句点分隔的四个组中，每个组都可以是一位、两位或者三位数字。

严格来讲，像 002 这样的值也可以用于 IP 地址。但是，前导零并不常用。为了演示下面的例子，我们假设前导零不会出现在数据中。

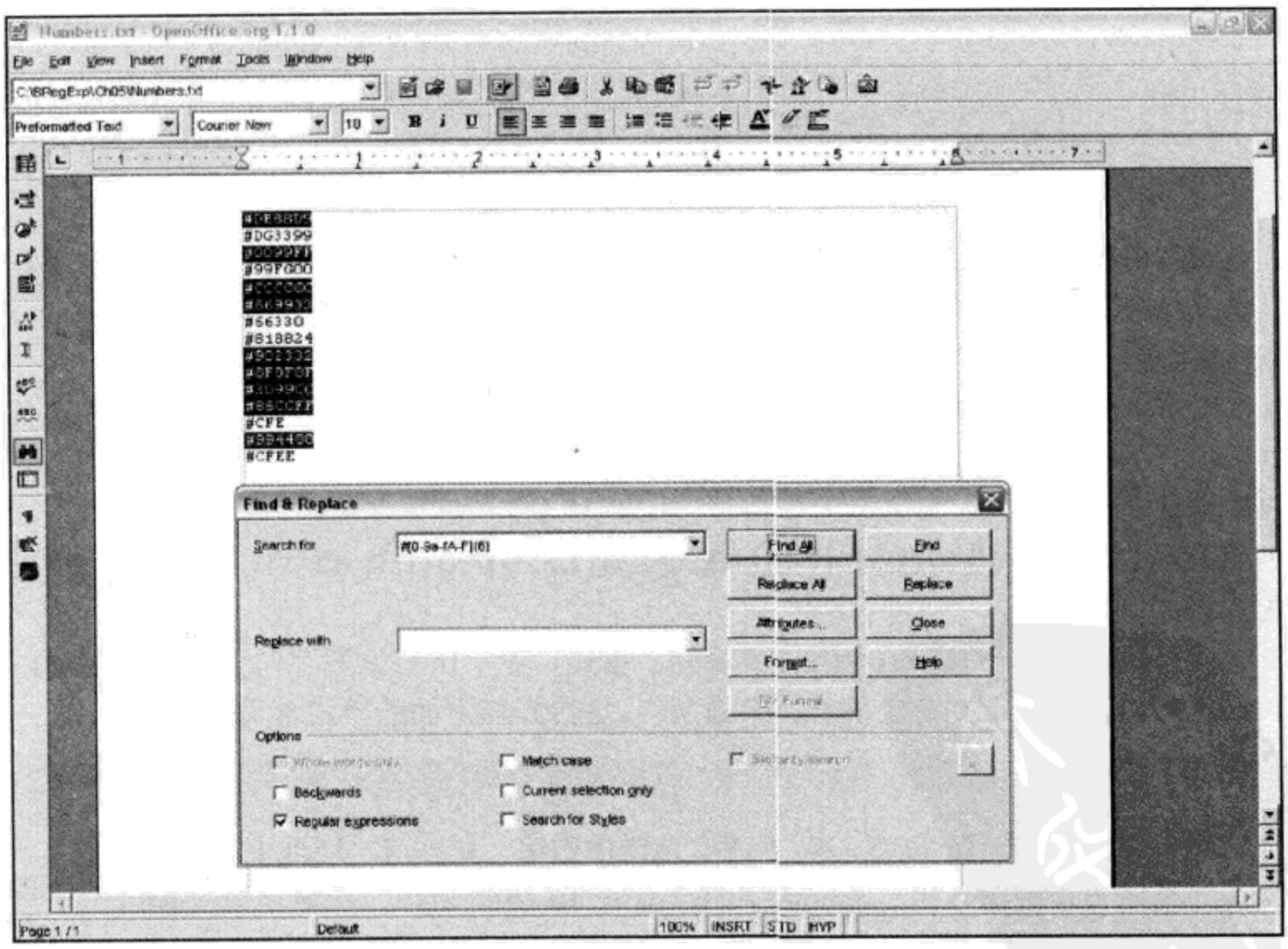


图 5-9

如果要用语言来描述 IP 地址的结构，则可以尝试用以下问题定义表达：

匹配 1~3 个数字后跟一个句点字符，后跟 1~3 个数字，后跟一个句点字符，后跟 1~3 个数字，后跟一个句点字符，最后跟 1~3 个数字。

基于以上描述，尝试创建一个用于识别 IP 地址的正则表达式模式，如下所示：

```
[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
```

记住，必须要用转义序列 \. 来匹配句点。

在示例文件 IPLike.txt 中，包含一些有效的 IP 地址和一些无效的 IP 地址：

```
12.12.12.12

255.255.256.255

12.255.12.255

256.123.256.123

8.234.88.55

196.83.83.191

8.234.88,55

88.173.71.66

241.92.88.103
```

图 5-10 显示的是在 OpenOffice.org Writer 中将上述模式应用到 IPLike.txt 后的结果。

在 IPLike.txt 中只有一个字符串——8.234.88,55 没有匹配模式，因为它包含一个逗号，而前面的描述中要求包含的是句点。

但是到目前为止，我们一直没有考虑 IP 地址中包含的数值最大不能超过 255 这条规则。所以，尽管前面的模式匹配 IPLike.txt 中的字符串 256.123.256.123，但它是一个无效的 IP 地址。

也就是说，模式 [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3} 既匹配 IP 地址，也可能匹配无效的 IP 地址。

再来仔细分析一下 IP 地址的构成规则。首先，看一下一位数字的情况。我们可以把这种情况描述为“位于 0~9 之间的一个数字”，这恰好与下面的模式含义吻合：

```
[0-9]
```

这个字符类会匹配像 1、3、4、6 或 9 这样的数字，这在 IP 地址中都是有效的。

第二种情况是两位数字。两位数字的范围是 10~99。所以，仍然可以分别用一个字符类来表示两位数中的位数：

```
[1-9][0-9]
```

之所以用字符类[1-9]，而不是 [0-9]，来匹配两个字符中的第一个字符，是因为小于 10

的数字属于一位数，可以用前面定义的字符类 [0-9] 来表示。然而，两位数的第二位数可以是 0，如 10 或 50 中，或者是 9，如在 29 或 79 中等。所以，与两位数中的第二位数对应的字符类是[0-9]。

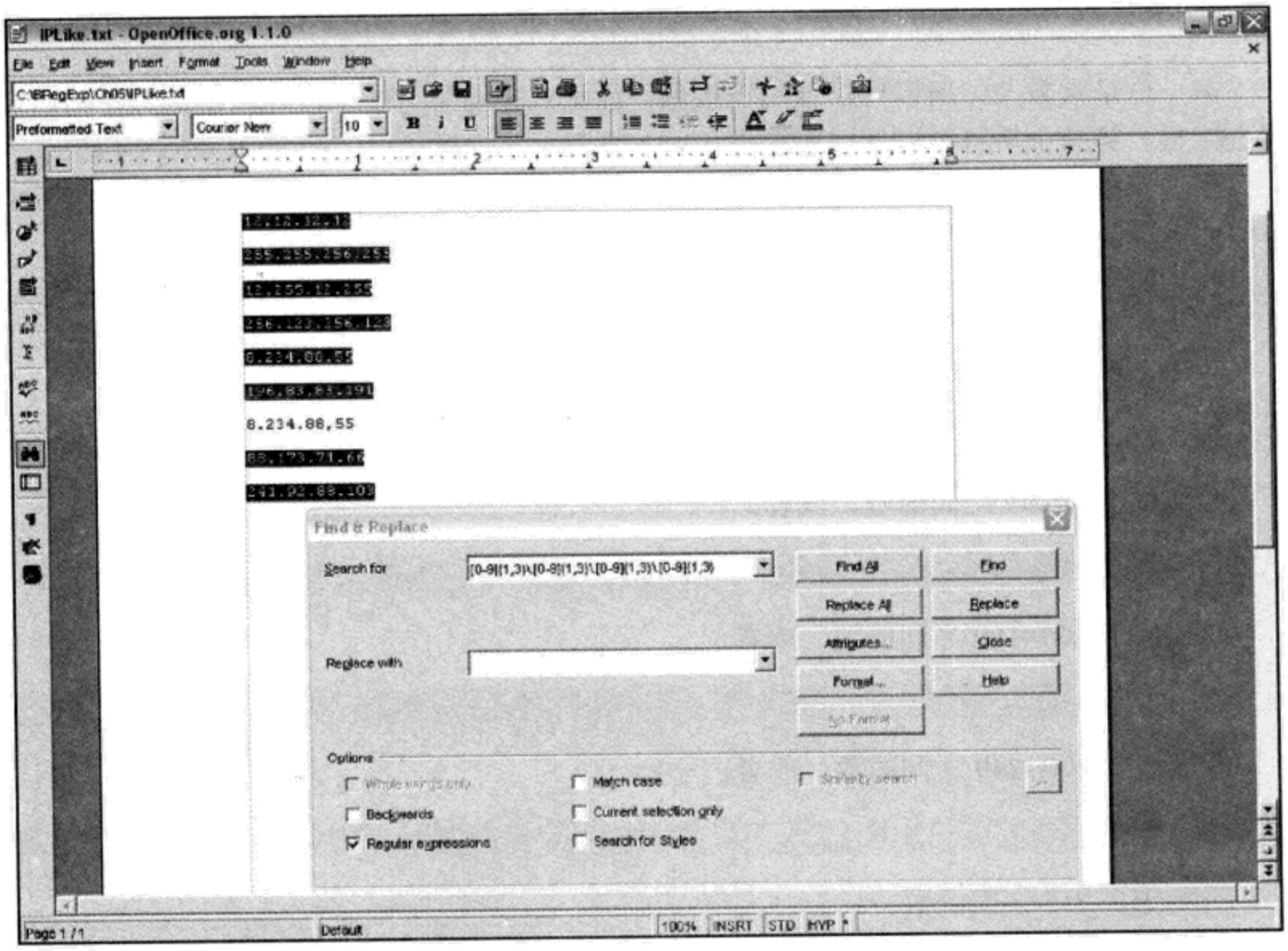


图 5-10

我们已经分析了与 IP 地址中包含的一位或两位数字匹配的、由字符类组成的模式。现在再来看三位数的情况。为了让读者更好理解，分两步来说明。首先，创建一个匹配 100~199 之间的数字的模式；然后，再创建匹配 200~255 之间的数字的模式。

首先，匹配 100~199 之间的数字的模式如下所示：

```
1[0-9][0-9]
```

此时，我们知道所有数字都以 1 开头，所以可以将其作为一个直接量字符包含在模式中。第二位数字可以是从 0(如 103 或 106)到 9(如 191 或 197)之间的任何数字。所以字符类 [0-9] 可以对应第二位数字。同样，第三位数字也可以是 0~9 之间的任何数字，所以字符类 [0-9] 也是合适的。

然后，看一看范围在 200~249 之间的三位数的情况。可以使用下面的模式：

```
2[0-4][0-9]
```

模式中的第一个字符 2 对于该范围内的所有值来说都是一样的，如 202、226 和 241

等。模式中的第二个字符可以通过字符类 [0-4] 来表示，而第三个字符可以用字符类 [0-9] 表示，比如 203、228 或 249 中的第三位数字。

接下来，还需要一个与范围 250~255 匹配的模式。可以使用下面的模式：

```
25[0-5]
```

第一位是常数 2，所以可以使用直接量。而模式中的第二个字符则是常数 5，所以也用直接量。第三个字符可以使用字符类 [0-5] 来表示，这样可以匹配 250、253 或 255。

现在，我们把以上分析构建的模式组合到一起。我们要匹配数值 0~255，而这个范围内的数值可能会与下面的任何一种模式匹配：

```
[0-9]
```

匹配一位数。或者：

```
[1-9][0-9]
```

匹配位于 10~99 之间的数。或者：

```
1[0-9][0-9]
```

匹配位于 100~199 之间的数。或者：

```
2[0-4][0-9]
```

匹配位于 200~249 之间的数。或者：

```
25[0-5]
```

匹配 250~255 之间的数。

如果要通过问题定义来描述这种情况，那么这个定义应该是迄今为止最长的一个定义：

匹配下面任何一种情况：

(1) 如果是一位数，匹配任何 0~9 的数字。

(2) 如果是两位数，匹配第一个字符为 1~9 且第二个字符为 0~9 的数字。

(3) 如果是三位数，那么与下面任一项匹配即可：

a. 匹配数字 1，后跟一个数字 0~9，后跟一个数字 0~9；

b. 匹配数字 2，后跟一个数字 0~4，后跟一个数字 0~9；

c. 匹配数字 2，后跟一个数字 5，后跟一个数字 0~5。

要想把前面五种可能的模式都组织到一起，需要用到圆括号。而且，因为这些模式都是互斥的，所以需要使用竖线 | 来分割这些选项。因此，要匹配一个位于 0~255 间的数值并且只匹配该范围内的一个值，就要使用下面的模式：

```
([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])
```

想象一下，当你六个月后再看到这个模式，或者在维护其他人编写的、没有注释的类似代码时会是什么样的情景。现在，你应该能够体会到为类似这样的问题添加完整的注释

的价值所在了。

这比此前遇到的问题要复杂得多，所以为了深入体会，我们在 OpenOffice.org Writer 中练习一下。

试一试：尝试匹配一个最大为 255 的数值

(1) 打开 OpenOffice.org Writer，然后打开测试文件 IPLike.txt。

(2) 使用快捷键 Ctrl+F 打开 Find & Replace 对话框，然后选中 Regular expressions 和 Match case 复选框。

(3) 在 Search for 文本框中输入模式 ([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])。注意，不要在圆括号中加入任何空白符。

(4) 单击 Find All 按钮，并观察结果，结果如图 5-11 所示。你可能会惊奇地发现 256 仍然被匹配了。

工作原理

仔细观察图 5-11，会发现第二行和第四行中的 256 都是匹配的。可是，我们刚刚还花了很多时间来精心设计这个正则表达式，让它最多只匹配到 255。问题出在哪里呢？

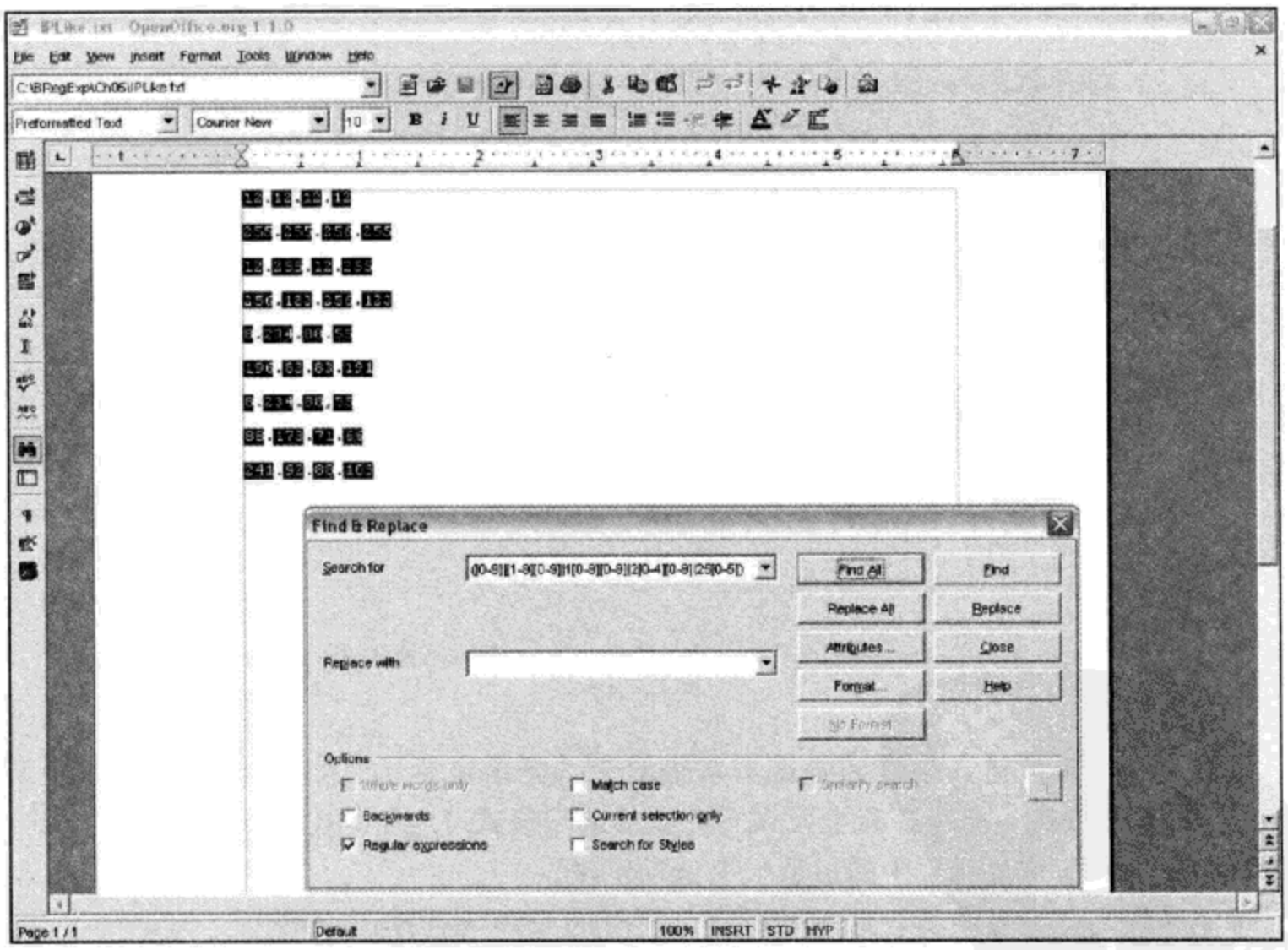


图 5-11

为了找出问题的答案，切换到 Komodo Regular Expression Toolkit 中来使用下面的模式测试 256 这个值：

```
([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])
```

图 5-12 显示了测试的结果。

结果显示 256 中的第一个数字 2 被匹配了。换句话说，256 中的 2 与圆括号中的第一个选项匹配。

这就为我们分析 OpenOffice.org Writer 中所出现的问题给出了一个提示。即模式中的第一个选项以及字符类 [0-9]，与 IPLike.txt 中 256 的第一位数 2 匹配。而同样是圆括号中的这个模式选项，也会匹配 256 中的 5 和 6。所以，在 OpenOffice.org Writer 中，我们看到的是 256 整体匹配，而实际上那是三次独立的匹配：一次匹配 2，一次匹配 5，一次匹配 6。而由于这三个连续的字符都突出显示，才使它看起来好像是正则表达式有问题一样。

那么，如果我们调整一下圆括号中几个选项的顺序会怎样呢？首先，把 [0-9] 移到最后：

```
([1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]|[0-9])
```

如果在 OpenOffice.org Writer 1.1 的 Find & Replace 对话框中修改这个模式，那么有可能触发该编辑器的某些 bug，结果会导致重写现有的字符或者无法确认新字符。当碰到这个问题时，建议在其他地方修改模式，比如在“记事本”中，修改后再把模式粘贴到 Search for 文本框中。

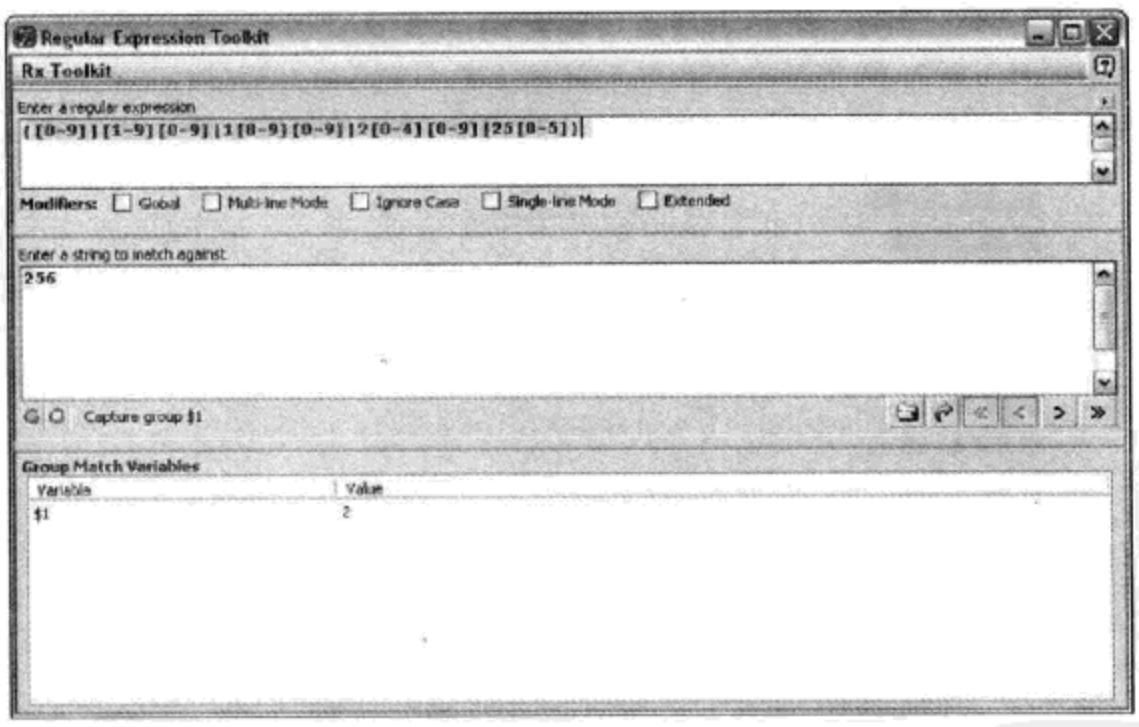


图 5-12

单击 Find All 按钮，会看到测试文本中的 256 突出显示。而如果单击 Find(不是 Find All)按钮来逐个查找匹配项，就会发现首先是 25 (与模式 [1-9][0-9])匹配，然后是 6 (与字符类[0-9]——现在是圆括号中的最后一个选项)匹配。

即使是颠倒圆括号中所有选项的顺序，所有的 256 也仍然会匹配：

```
(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])
```

现在，完成这个例子所需要的正则表达式模式，应该是一个能够将第一个句点之前的所有字符当做一个单独的块来匹配的模式；而位于两个句点之间的数字也要作为一个单独

的块；位于最后一个句点和字符串结尾之间的数字仍然要以一个单独块的形式来匹配。在第 6 章中学习了 ^ 和 $ 元字符的含义及用法后，我们还会再回来讨论这个问题。

如果现在就需要拿出一个方案来解决上面提出的问题，下面的模式可以满足要求——也就是说，它不会匹配任何包含 256 或更大数值的类似 IP 地址的字符序列。

```
^((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\.){3}(25[0-5]|2[0-
4][0-9]|1[0-
9][0-9]|[1-9][0-9]|[0-9])$
```

上面模式的工作原理在第 6 章中解释。

5.2.2 反转字符类的范围

到现在为止，我们接触过的范围遵循的都是字母表或者数字顺序。然而，通过使用某些工具可以编写出符合反转的字母或数字顺序的范围。对此，我们称做反转范围。

当在字符类中使用字符范围时，如果要反转范围则必须谨慎从事。因为不同的产品或语言在处理这种语法时存在不一致的情况。

例如下面这个正则表达式：

```
[t-r]ight
```

它在 OpenOffice.org Writer 就不会按照你的意图选择相应字符序列中的 r、s 或 t，而是把 t 和 r 当做字符类中的直接量字符，并忽略连字符。可以通过图 5-13 看到这一点。注意字符序列 sight 也包含在文件 Light2.txt 中，但没有被该正则表达式模式匹配。

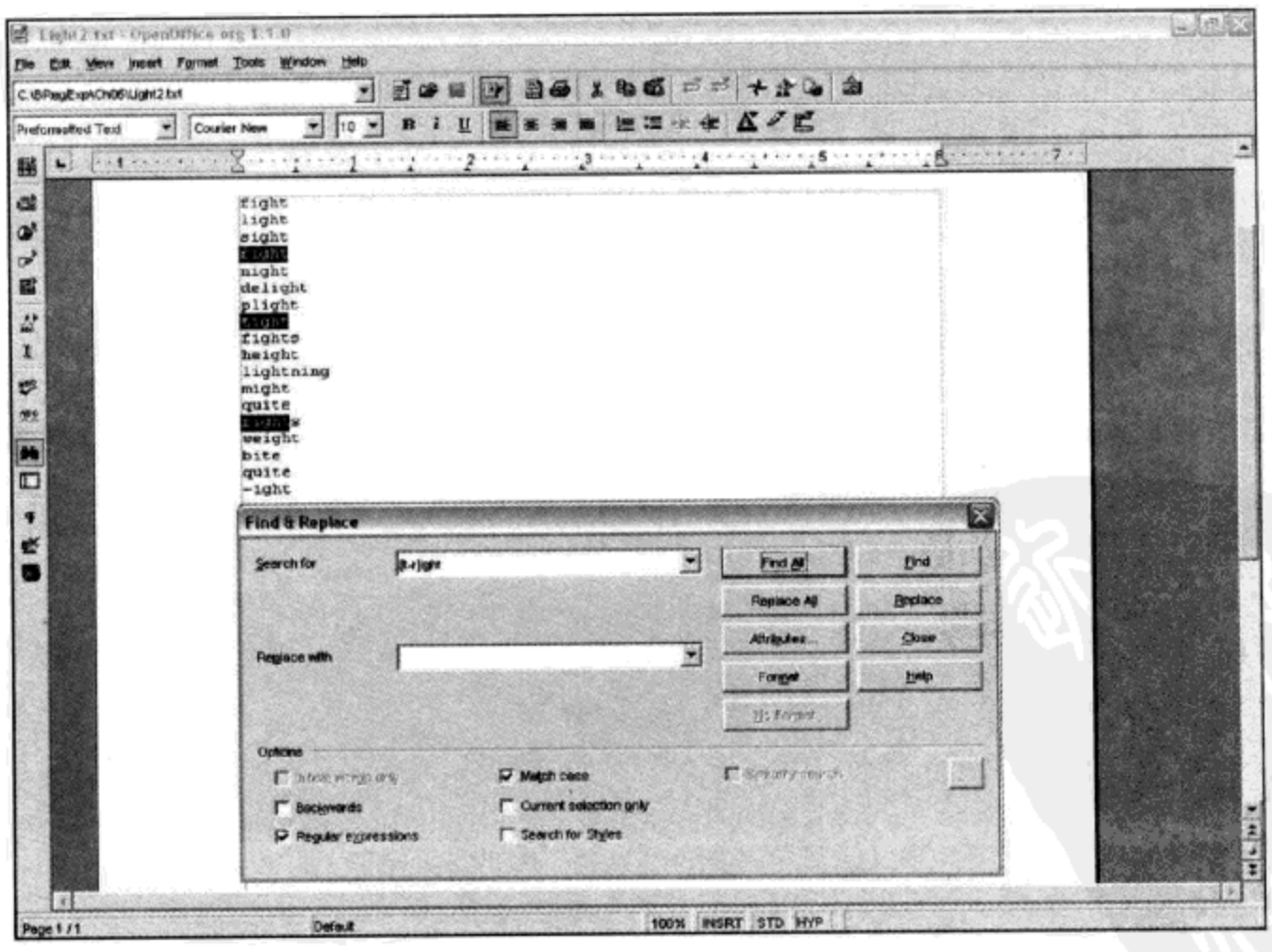


图 5-13

然而，在 PowerGrep 中，正则表达式模式 [t-r]ight 不会被接受并生成如图 5-14 所示的错误信息。

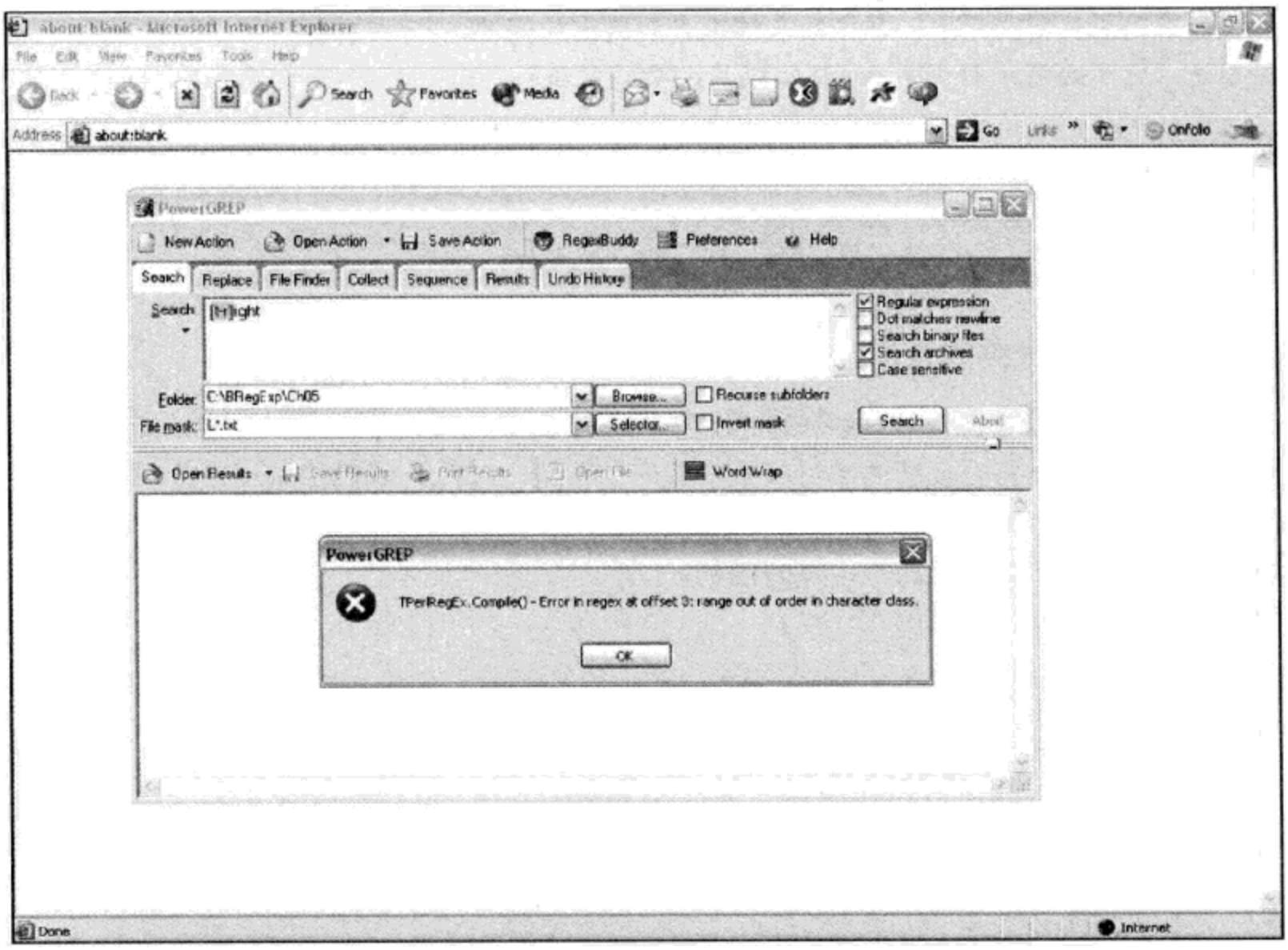


图 5-14

这些事实说明，在字符类中使用反转范围没有任何益处，所以建议读者最好不要使用它。

5.2.3 潜在的范围陷阱

假设你可以在一份或一组文档的日期中使用不同的分隔符。那么这些文档发布后可能会导致一些问题。它们便是表达字符范围时存在的陷阱。

这里要测试的文档是 Dates.txt，其内容如下：

```
2004-12-31
2001/09/11
2003.11.19
2002/04/29
2000/10/19
2005/08/28
2006/09/18
```

从上面可以看到，文件中日期的格式是 YYYY/MM/DD。但个别的会使用连字符，或者句点作为分隔符。你的任务就是选择所有表示日期的字符序列(在这个例子中，我们假设日期全部由数字加分隔符构成，不包含年、月)。

这样，如果想选择所有日期，则无论分隔符还是连字符，正斜杠还是句点，都可能会用下面的正则表达式模式来达到目的：

```
(20|19)[0-9]{2}[.-/][01][0-9][.-/][0123][0-9]
```

其中，字符类 [.-/] 的作用是匹配分隔符。但是实际上，这个字符序列(句点后跟一个连字符和一个正斜杠)会被解释为一个从句点到正斜杠的范围。然而，在图 5-15 中可以看到，连字符是 U+002D，而句点(U+002E)是直接位于正斜杠(U+002F)前面的字符。所以，模式 .-/ 最终出人意料地指定了一个只包含句点和正斜杠字符的范围。

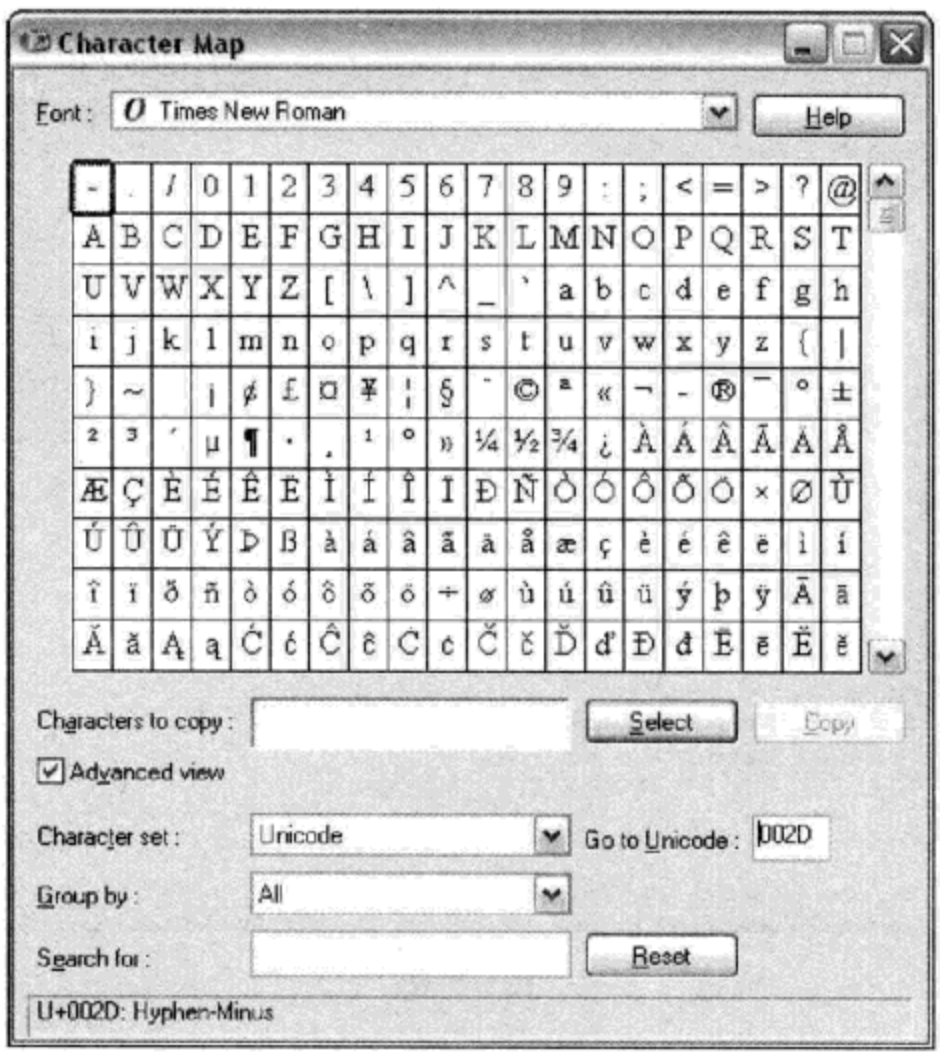


图 5-15

字符可以使用 Unicode 数值引用(numeric references)来表示。句点是 U+002E，大写的 A 是 U+0041。在 Windows 的字符映射实用程序中，通过把光标悬停在感兴趣的字符上可以查看到相应字符的数值引用。

如果不想让连字符表示范围，那么连字符应该作为字符类中的第一个字符：

```
[-./]
```

下面是与 Dates.txt 中的每个日期都匹配的模式：

```
(20|19)[0-9]{2}[-./][01][0-9][-./][0123][0-9]
```

试一试：匹配日期

(1) 打开 PowerGrep，在 Search 文本框中输入正则表达式模式(20|19)[0-9]{2}[-./][01][0-9][-./][0123][0-9]。

(2) 在 Folder 文本框中输入 C:\BRegExp\Ch05，假设下载的第 5 章文件被放在这个目录中。

(3) 在 File mask 文本框中输入 Dates.txt 。

(4) 单击 Search 按钮，并观察如图 5-16 所示的结果。特别注意一下包含连字符的第一个匹配项 2004-12-31，这表示正则表达式模式按照我们的期望成功匹配了连字符。

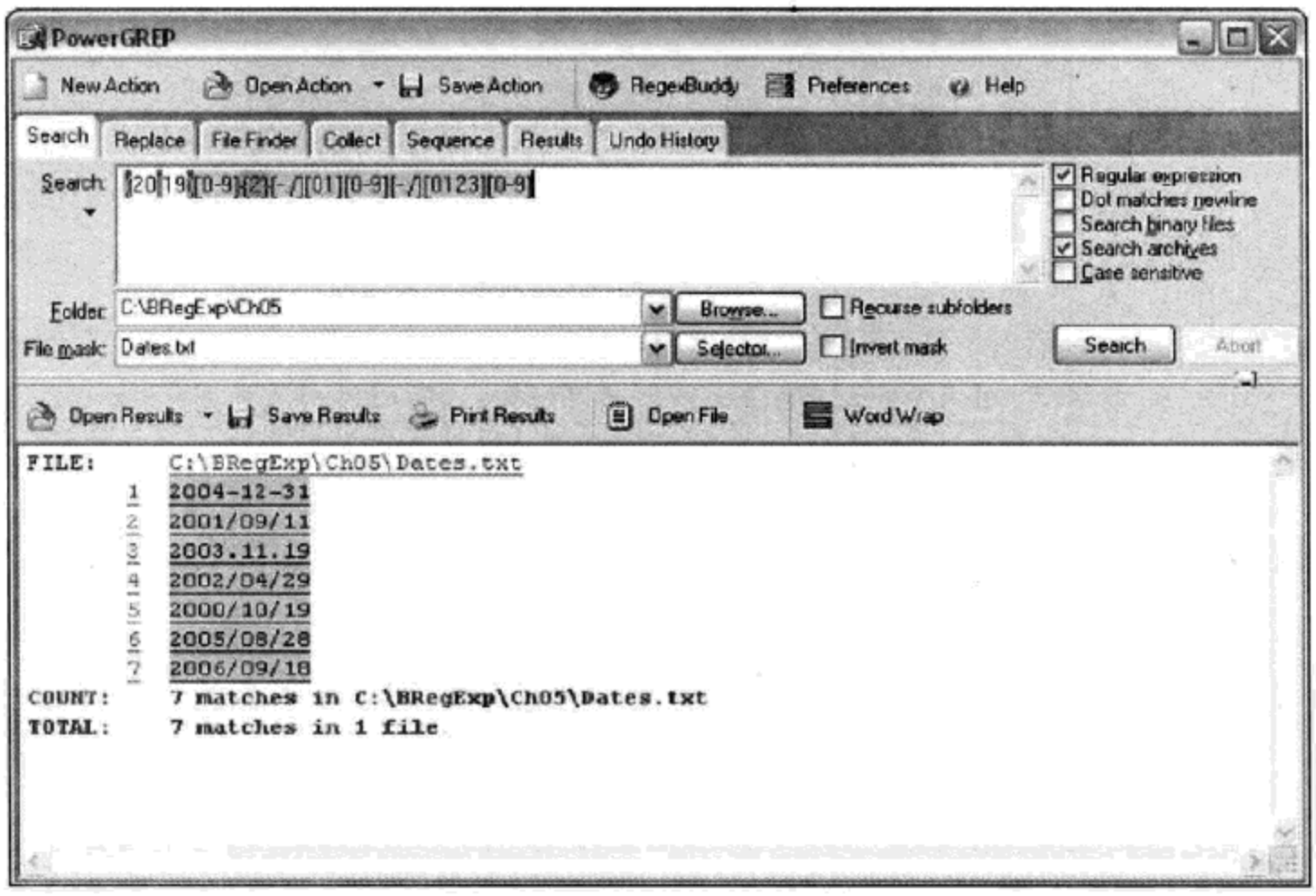


图 5-16

工作原理

模式的第一部分 (20|19)，允许被测试的字符序列的前两个字符是 20 或 19。接着，模式 [0-9]{2} 匹配从 0 到 9 的两个连续的数字。然后，字符类模式 [-./] 匹配一个单独的字符，可以是一个连字符、句点或者一个正斜杠。模式的下一个组件是[01]，它匹配数字 0 或 1，因为在日期格式中月份的第一位数字始终是 0 或 1。同理，下一个组件、字符类 [0-9]，匹配包含在 0~9 之间的数字。这样，月份甚至可以是 14 或 18——显然，这不是我们想要的。本章最后的一道练习题就是要读者为只允许从 01 到 12 (包含 01 和 12)匹配的情况给出一个更具体的模式。

下一个字符类模式 [-./] 匹配一个字符——连字符、句点或者正斜杠。

最后，模式 [0123][0-9] 匹配每月的以 0、1、2 或 3 开头的天数。而这个模式也会允

许像 00、34 或 38 这样的值。本章后面的一道练习题中会让读者创建一个更具体的模式把天数值限制在 01~31。

5.2.4 查找 HTML 中的标题元素

字符类的一个潜在的用法是查找 HTML/XHTML 中的标题元素。HTML 和 XHTML 1.0 规定了六个标题元素：h1、h2、h3、h4、h5 和 h6。在 XHTML 中 h 必须小写。而在 HTML 中则可以使用 h 或 H。

首先，假设所有元素都使用小写字母，这样就可以匹配所有六个标题元素的开始标签。再假设标题中没有属性，于是可以使用下面带有圆括号的穷举式的正则表达式：

```
<(h1|h2|h3|h4|h5|h6)>
```

在这个模式中，< 是一个直接量尖括号字符，它是开始标签的第一个字符。然后是包含在圆括号中的六个双字符序列选项，用于表示每个 HTML/XHTML 中的标题元素。最后，> 表示开始标签的结束符。

但是，由于模式中包含数字序列 1~6，所以也可以使用一个字符类来匹配相同的开始标签——即可以列举每个数字直接量：

```
<h[123456]>
```

也可以在字符类中使用范围：

```
<h[1-6]>
```

示例文件 HTMLHeaders.txt 的内容如下：

```
<h1>Some sample header text.</h1>

<h3>Some text.</h3>

<h6>Some header text.</h6>

<h4></h4>

<h5>Some text.</h5>

<h2>Some fairly meaningless text.</h2>
```

其中包含全部六种标题元素。

试一试：匹配 HTML 标题元素

(1) 打开 PowerGrep，并在 Search 文本框中输入正则表达式模式<h[1-6]> 。

(2) 在 Folder 文本框中输入 C:\BRegExp\Ch05，假设下载的第 5 章文件被放在这个目录中。

(3) 在 File mask 文本框中输入 HTMLHeaders.txt。

(4) 单击 Search 按钮并观察结果，结果如图 5-17 所示。

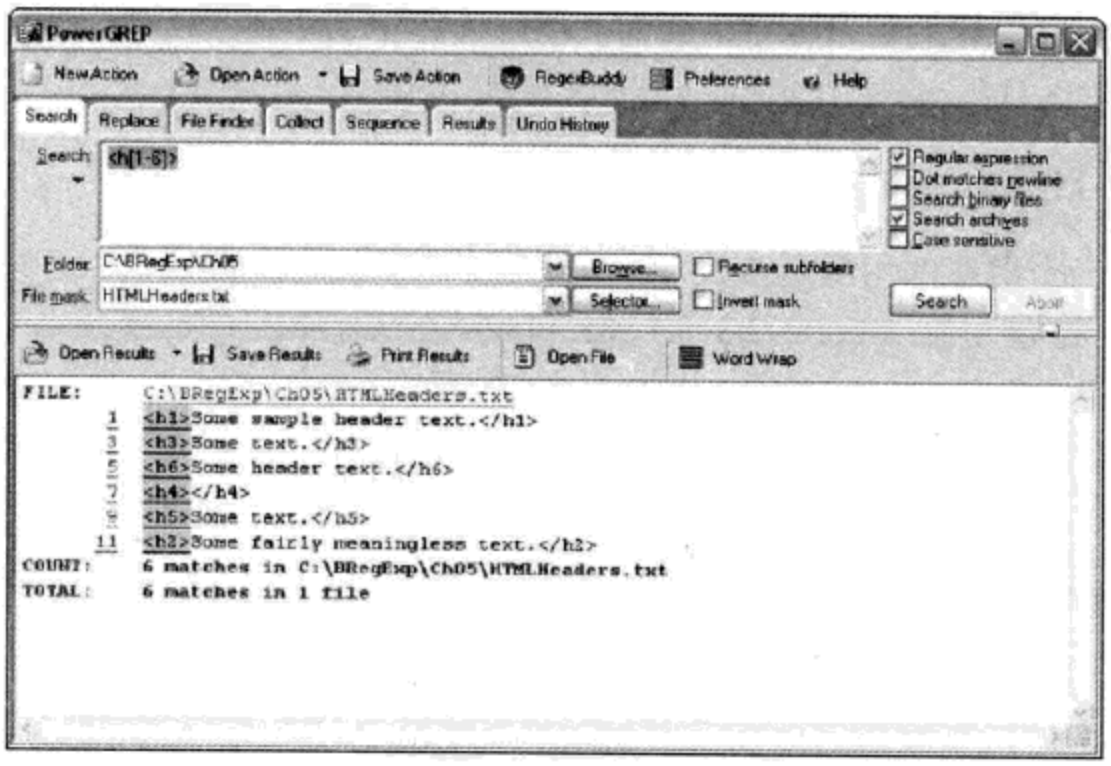


图 5-17

5.3 字符类中元字符的含义

多数情况下(但不是所有情况下)，单个字符在字符类内部的含义与它在字符类外部的含义是相同的。

5.3.1 ^ 元字符

^ 元字符(也称为脱字符)，当它是字符类中左方括号后面的第一个字符时，表示的是方括号中指定的任何字符都不能匹配的情况。关于 ^ 元字符的用法，将在后面对字符类取反一节中讨论。

如果 ^ 元字符不作为方括号中的第一个字符出现，而是出现在其他任何位置上，它的含义都是其直接量本身——即匹配 ^ 字符。

下面要用到的一个测试文件是 Carets.txt，其内容如下：

```
14^2 expresses the idea of 14 to the power 2.

The ^ character is called a caret.

The _ character is called an underscore or underline character.

3^2 = 9

Eating ^s helps you see in the dark. At least that's what I think he said.
```

问题定义可以这样来表达：

匹配下列任何字符：下划线、脱字符或者数字 3。

满足这一问题定义的字符类如下：

```
[_^3]
```

试一试：在字符类中使用 ^ 元字符

这个例子用于匹配前面问题定义中提到的三个字符：

(1) 打开 OpenOffice.org Writer，然后打开测试文件 Carets.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框，并在 Search for 文本框中输入模式 [_^3] 。

(4) 单击 Find All 按钮，并观察结果，结果如图 5-18 所示。

(5) 将正则表达式模式修改为 [^_3]。

(6) 单击 Find All 按钮，将如图 5-19 所示的结果与前一次的匹配结果进行比较。

工作原理

当使用模式[_^3]时，其含义仅仅是一个匹配三个字符——下划线、脱字符和数字 3 的普通字符类。

而当 ^ 直接位于字符类的左方括号([ )后面时，等于是创建了一个取反的字符类，在这种情况下其含义变成了“匹配除了下划线或者数字 3 之外的任何字符”。

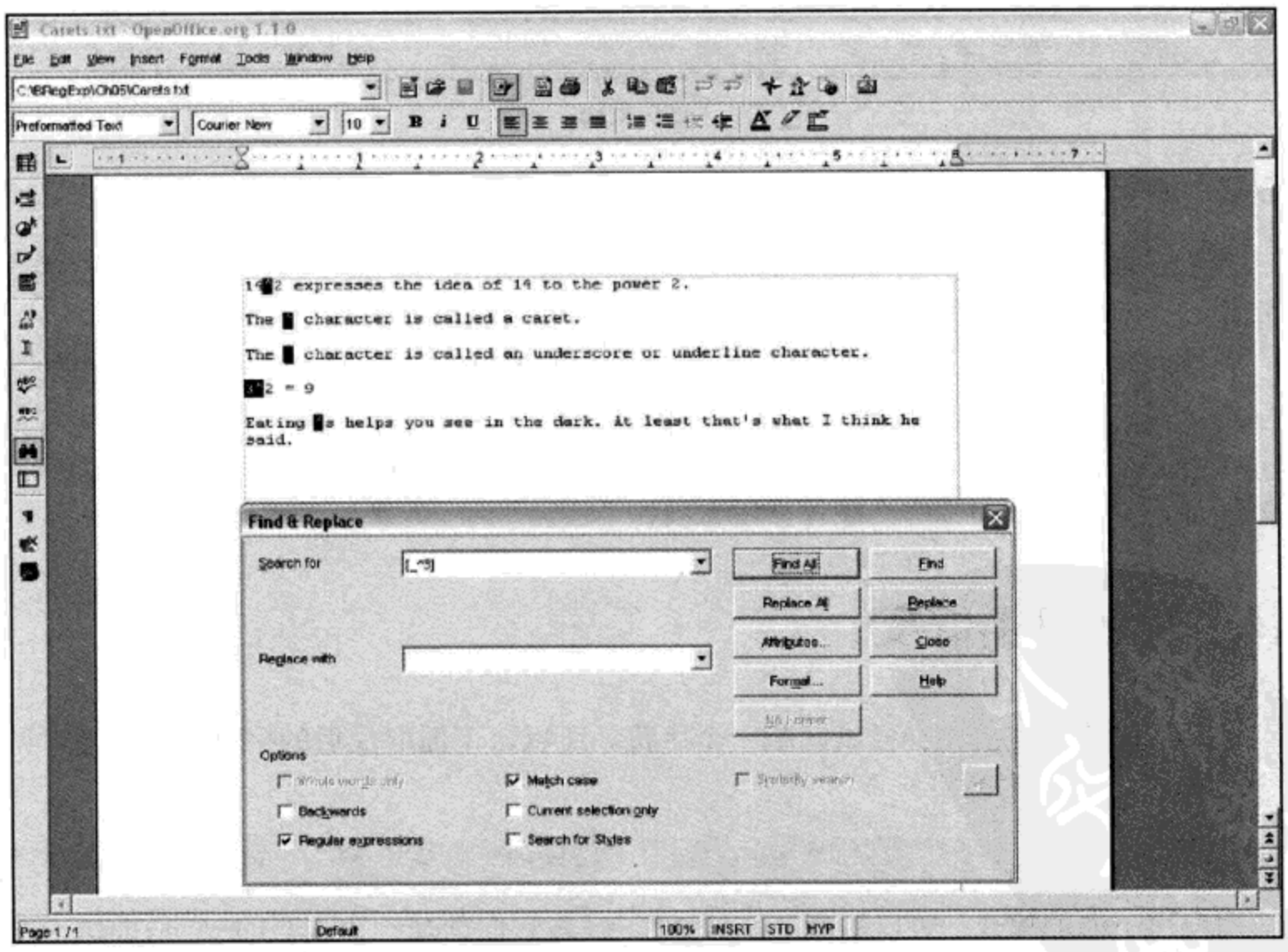


图 5-18

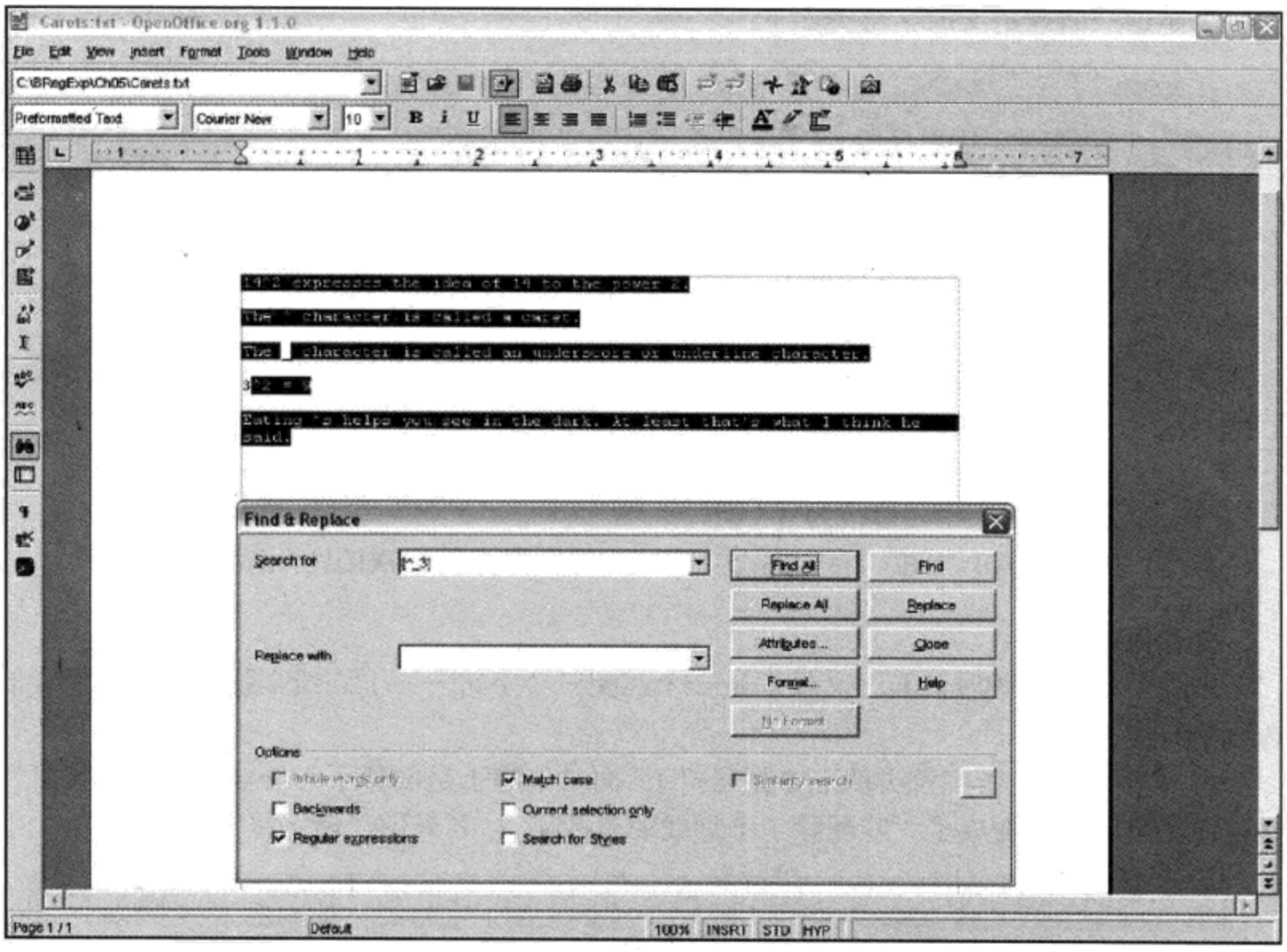


图 5-19

5.3.2 如何使用 - 元字符

我们已经学习了如何在字符类中用连字符表示范围。因此，也就引出了在字符类中如何指定一个直接量连字符的问题。

在字符类中使用直接量连字符的最常见的方式是将连字符作为左方括号后面的第一个字符。在有些工具中，比如 Komodo Regular Expression Toolkit 中，也可以把连字符放在右方括号的前面来匹配连字符本身。但在 OpenOffice.org Writer 中，将连字符作为字符类内部的最后一个字符则不会匹配连字符本身。

5.4 对字符类取反

取反后的字符类仍然会尝试匹配一个字符。所以，下面取反后的字符类意味着“匹配一个不在 A~F 范围内的字符”。

```
[^A-F]
```

而使用下面的模式，可以匹配 AG 和 AZ。因为它们都是一个大写的 A 后面跟一个不属于 A~F 范围之内的字符的情况：

```
A[^A-F]
```

这个模式不会匹配两个连续的 A，因为虽然第一个 A 会匹配，但取反的字符类 [^A-F] 却不会匹配第二个 A。

组合正负(取反的)字符类

在某些语言(如 Java)中，可以组合使用正负字符类。

下面的例子示范了如何组合使用字符类。相关的问题定义如下：

匹配字符 A 和 E~Z。

另一种表达相同意图的方式是：

匹配字符 A~Z，但不包括 B~D。

在 Java 中，可以使用下面组合的字符类来表达这一定义：

```
[A-Z&&[^B-D]]
```

注意其中的一对“与符号(&)”，其含义是逻辑“与”。因此，这个模式的含义是“匹配处于范围 A~Z 之间并且不在范围 B~D 之间的字符“。

在此，我们提供了一个简单的 Java 命令行程序—— CombinedClass2.java：

```
import java.util.regex.*;

public class CombinedClass2{
  public static void main(String args[])
  throws Exception{

  String TestString = args[0];

  String regex = "[A-Z&&[^B-D]]";
  Pattern p = Pattern.compile(regex);

  Matcher m = p.matcher(TestString);
  String match = null;

  System.out.println("INPUT: " + TestString);
  System.out.println("REGEX: " + regex);
  while (m.find())
        {
         match = m.group();
         System.out.println("MATCH: " + match);
        } // end while

         if (match == null){
           System.out.println("There were no matches.");
         } // end if

  } // end main()
}
```

试一试：使用组合的字符类

以上代码假设已经正确安装并配置好了 Java 1.4。下面的例子演示了如何在 Java 中使用组合的字符类：

(1) 打开命令提示窗口，在命令行中输入 javac CombinedClass2.java 来编译源代码。

(2) 输入 java CombinedClass2.java "A C E G" 并运行该程序，对"A C E G"进行测试。

(3) 观察结果，如图 5-20 所示。注意，A、E 和 G 匹配，但是 C 没有匹配。

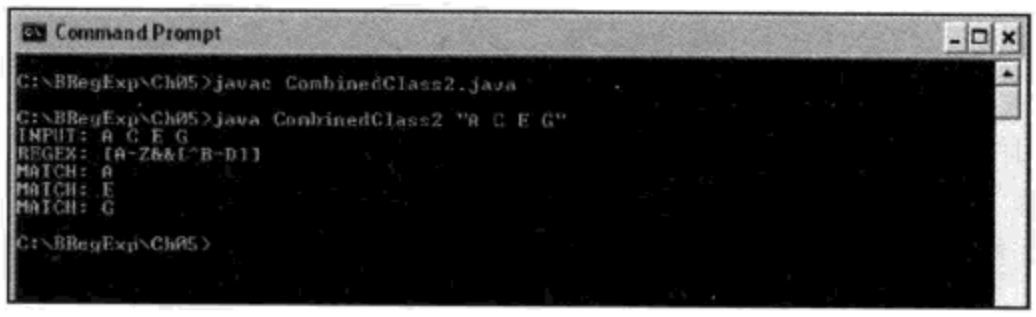


图 5-20

工作原理

在命令行中提交测试字符串。该测试字符串被赋值给变量 TestString：

```
String TestString = args[0];
```

而程序将一个正则表达式指定给变量 regex：

```
String regex = "[A-Z&&[^B-D]]";
```

这个正则表达式是由前面介绍的字符类组合而成的。

在调用 Pattern 对象的 compile() 方法时将 regex 变量作为参数：

```
Pattern p = Pattern.compile(regex);
```

然后，在执行 Pattern 对象 p 的 matcher() 方法时以变量 TestString 作为参数：

```
Matcher m = p.matcher(TestString);
```

下面的代码将 null 值赋给一个新变量 match：

```
String match = null;
```

在简单的输出中显示了命令行中提供的测试字符串、所用的正则表达式模式，以及是否存在一个或多个匹配项，如果有则显示匹配项；否则，显示没有找到匹配项的信息：

```
System.out.println("INPUT: " + TestString);
System.out.println("REGEX: " + regex);
while (m.find())
{
      match = m.group();
      System.out.println("MATCH: " + match);
      } // end while
```

```
if (match == null){
System.out.println("There were no matches.");
} // end if
```

可以在命令行中使用包含其他大写字母的字符串作为输入进一步进行测试。

5.5 POSIX 字符类

有些正则表达式实现还支持一些非常特殊的字符类表示法——POSIX 字符类表示法。POSIX 方法使用一种命令来表示很多有用的字符类，而不是以我们在本章前面所看到的方式来指定字符类。例如，POSIX 不会像字符类 [A-Za-z0-9] 那样，列出相关的字符，而是使用 [:alnum:] 表示同一含义。其中，alnum 是 alphanumeric(字母数字字符)的简写形式。从个人的角度上来说，笔者更喜欢本章前面使用的定义字符类的语法。但是，因为读者有可能会在代码中看到 POSIX 字符类，所以本节会简单地介绍一下 POSIX 字符类的内容。

就以上面提到的 [:alnum:] 字符类为例。

POSIX 的语法有地域性。本节所介绍的语法是与英语语言地区相关的语法。

[:alnum:]字符类

在支持 POSIX 字符类的不同工具中，[:alnum:] 字符类也不相同。一般来说，[:alnum:] 字符类等价于下面的字符类：

```
[A-Za-z0-9]
```

然而，[:alnum:] 字符类还有其他不同的解释。

试一试：在 OpenOffice.org Writer 中使用 [:alnum:] 字符类

在 OpenOffice.org Writer 中，要顺利地使用 [:alnum:] 字符类，必须要为其添加一个 ? 限定符(或其他限定符)：

(1) 打开 OpenOffice.org Writer，然后打开测试文件 AlnumTest.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框，并在 Search for 文本框中输入模式 [:alnum:]? 。

(4) 单击 Find All 按钮，并观察突出显示的文本，如图 5-21 所示。确定由模式 [:alnum:]? 匹配的内容。

注意，在示例文件中的最后一行有两个下划线字符没有被模式 [:alnum:]? 匹配。

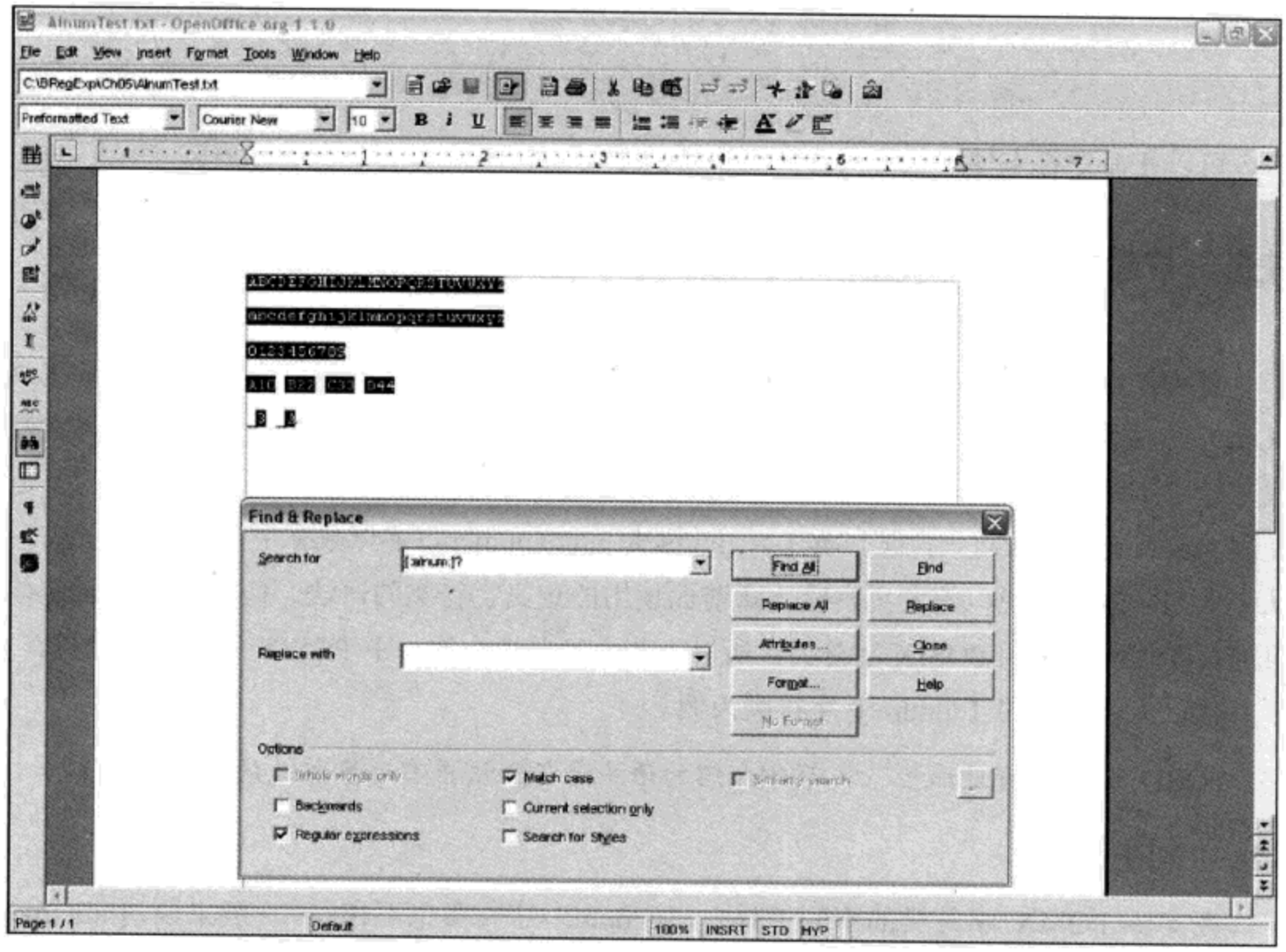


图 5-21

如果在第 4 步中单击的是 Find 按钮，假设那时的光标处于示例文件的开始位置，那么第一行中的首字母 A 会匹配，因为它是第一个与模式匹配的字符。

工作原理

如果正则表达式引擎在测试文件中第一行的 A 之前的位置开始匹配，那么首先会根据模式[:alnum:]? 来测试 A。因为大写的 A 是一个字母字符，所以匹配成功。匹配的文本将突出显示。

如果单击了 Find All 按钮，在第一次成功匹配后，正则表达式引擎会移动到 A 和 B 之间的位置，并尝试匹配下一个字符 B。这次同样也匹配，因而 B 也将突出显示。然后，正则表达式引擎又向前移动一个字符的位置并匹配 C，依次类推。当它遇到换行符时，因为换行符不会与模式 [:alnum:]? 匹配，所以正则表达式引擎会移动到这个换行符之后的下一个位置上并尝试与下一个字符进行匹配。

当正则表达式引擎到达第三行的下划线字符之前的位置并尝试匹配下划线时，匹配失败，因为下划线字符既不是一个字母字符也不是一个数字。

5.6 练习

1. 假设你有一个包含美式英语和英式英语的文档。首先，请声明一个查找 license(美式英语)和 licence(英式英语)的问题定义。然后，使用字符类建立一个正则表达式来匹配这两个字符序列。

2. 本章中曾经使用模式(20|19)[0-9]{2}[-./][01][0-9][-./][0123][0-9] 来匹配日期。但是，这个模式也会匹配 00、13 或 19 这样的月份，而且还会匹配 00、32 和 39 这样的天数值。请修改这个模式中的相关组件，使它只匹配 01~12 的月份和 01~31 这样的天数。

第 6 章

字符串、行和词边界

本章不介绍匹配字符，而是介绍位于字符之前、之间或之后位置的元字符。这些匹配位置的元字符是对第 4 章中介绍的那些元字符的补充。

举例来说，本章会学习如何匹配位于一行开始位置处的字符或字符序列。比如在英语中，可能需要匹配的只是那些位于一行开始或者整个测试文件开始处的特定的字符序列。也就是说，不想匹配位于其他任何位置的这个字符序列。此时，使用位置元字符可以明确地指定要匹配的字符序列的位置。

通常，你可能希望查找的是一个单词而不是字符序列，也不是一个单词开始或结尾处的字符序列。许多正则表达式的实现中都提供了这样的位置元字符。

本章的内容主要是介绍如何基于字符序列的位置实现匹配。

> 在某些文档(如.NET 正则表达式功能的文档)中，这些位置元字符也会被称做“原子零宽度断言(atomic zero-width assertions)”。

本章将解释如何使用：

- ^ 元字符，它匹配一个字符串或一行的开始位置
- $ 元字符，它匹配一个字符串或一行的结束位置
- \< 和 \> 元字符，它们分别匹配一个单词的开始和结束位置
- \b 元字符，它匹配一个单词的边界(出现在一个单词的开始或结束位置)

6.1 字符串、行和词边界

能用于创建与特定位置的字符序列匹配的正则表达式的元字符非常有用。

比如，想查找所有以单词 The 开始的行。虽然运用在前几章学习的知识很容易通过创建一个直接量模式来匹配字符序列 The，但是用这些技术却无法指定这个字符序列出现的位置，不论是一个单词的位置还是组成一个较长单词的字符序列的位置。单纯的模式 The，除匹配单词 The 外，还会匹配 There、Then 中以及其他句子开始处的 The，比如人名或娱乐设施名 Theodore 和 Theatre。

同理，如果在不区分大小写的情况下使用模式 The，那么还有可能会匹配位于单词中的字符序列，如 lathe 中的 the(这是不希望看到的一个负面效果)。

首先我们来讨论和演示 ^ 元字符和 $ 元字符，它们的作用是指定一行或一个字符串的开始和结束的位置。

6.1.1　^ 元字符

^ 元字符会直接匹配位于一行或一个字符串开始位置之后的目标字符。

因此把模式：

```
The
```

应用到下面的测试文本时：

```
The Thespian Theatre opens at 19:00.
```

会匹配单词 The、Thespian 和 Theatre 中的字符序列 The。

但是，如果在这个模式的前面加上 ^ 元字符：

```
^The
```

再把它应用到相同的测试文本则会导致只有单词 The 中的字符序列 The 匹配——因为此处的 The 直接位于字符串的开始位置之后。

当把 ^ 元字符用于字符类之外时，它就失去了在字符类内部作为第一个字符时所具有的取反的含义。

试一试：Theatre 的例子

使用测试文件 Theatre.txt 中非常简单的测试文本：

```
The Thespian Theatre opens at 19:00.
```

(1) 打开 PowerGrep，并选中 Regular expression 复选框。

(2) 在 Search 文本框中输入模式 The。

(3) 在 Folder 文本框中输入 C:\bregExp\Ch06。

(4) 在 File mask 文本框中输入 Theatre.txt。

(5) 单击 Search 按钮，并观察 Results 区域中的结果，如图 6-1 所示。注意，Results 区域中的信息表明有三个模式 The 的匹配项。

(6) 把正则表达式模式修改为 ^The。

(7) 单击 Search 按钮，然后再观察 Results 区域中的结果，如图 6-2 所示。注意，现在只剩下一个匹配项了，这与修改正则表达式之前查找到三个匹配项形成了对照。

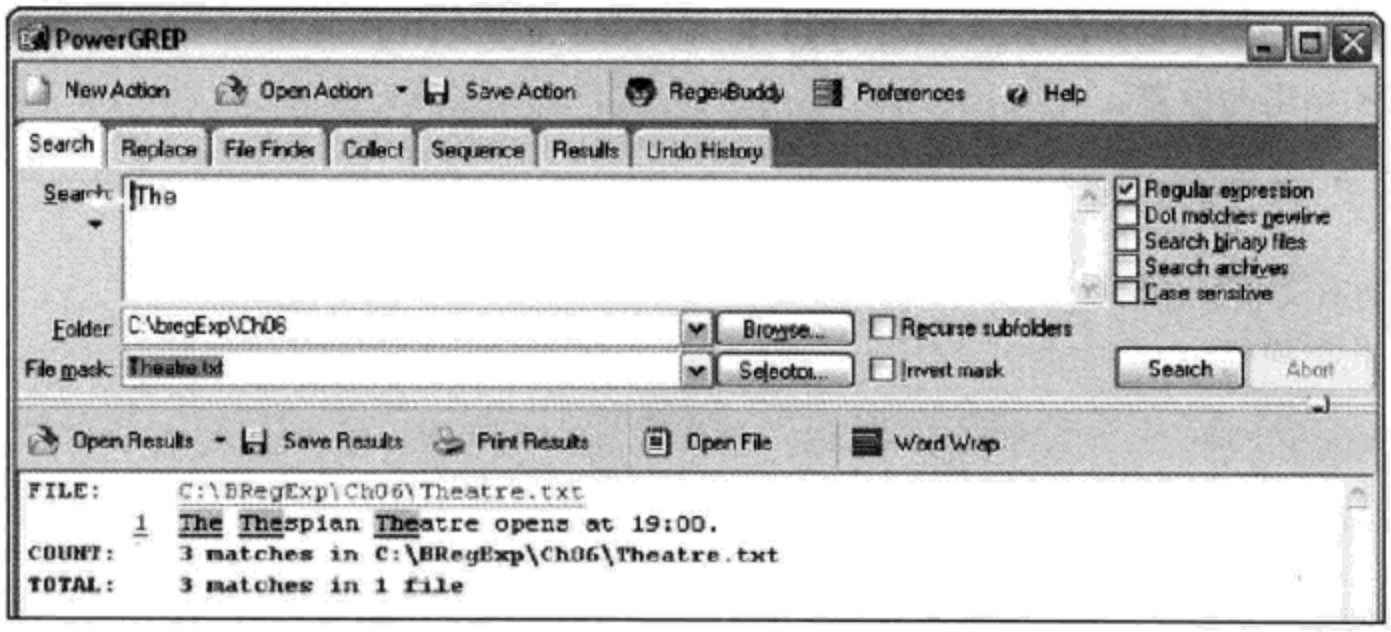


图 6-1

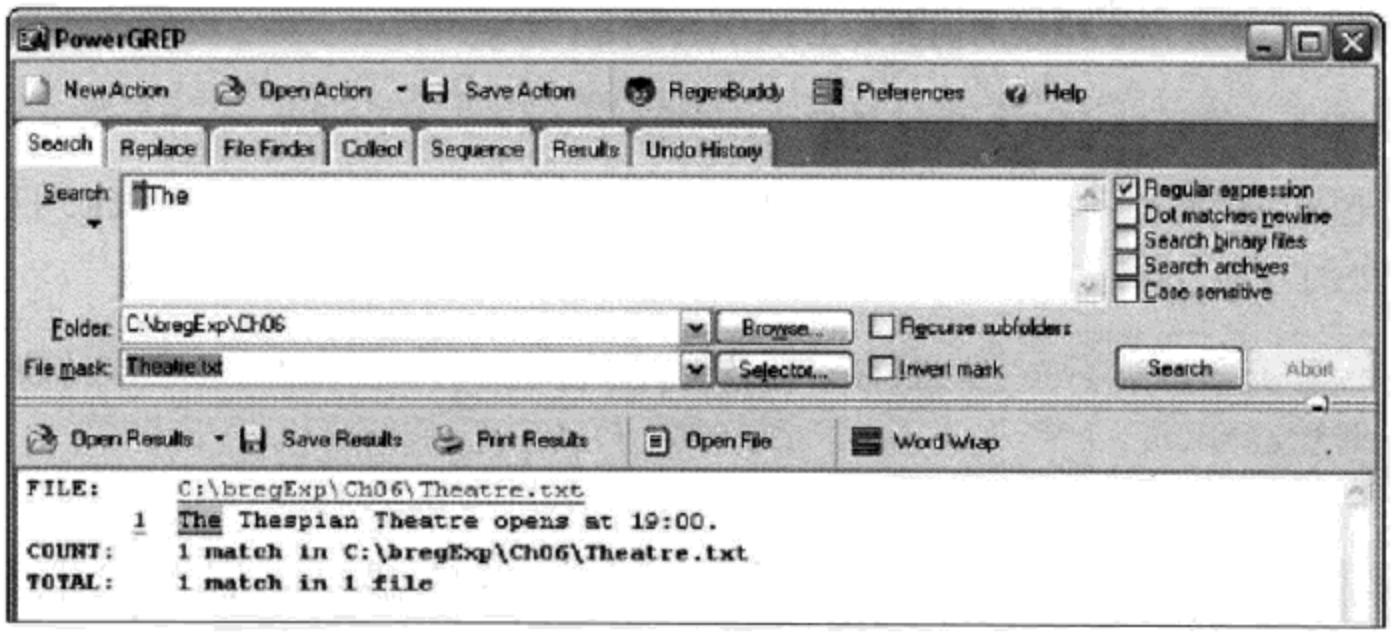


图 6-2

工作原理

正则表达式引擎从测试文件中第一个字符之前的位置开始查找匹配项。模式中的第一个字符是 ^ 元字符，它会尝试与正则表达式引擎的当前位置进行匹配。因为此时正则表达式引擎位于文件的开始位置，仅满足了 ^ 元字符规定的条件(匹配成功)。接着，正则表达式引擎会继续尝试匹配模式中的另一个字符。模式中的另一个字符、直接量 T 会尝试与测试文件中的第一个字符进行匹配，而那个字符也是 T，匹配成功。然后，正则表达式引擎又尝试将模式中的下一个字符——小写的 h，与测试文本的第二个字符匹配，而那个字符恰好也是小写的 h，所以模式中的直接量 h 与文本中的直接量 h 也匹配成功。之后，正则表达式引擎尝试将模式中的直接量 e 与测试文本中的第三个字符——小写的 e 进行匹配，当然匹配也是成功的。因为正则表达式中的所有组件都匹配成功，所以整个正则表达式匹配成功。

当正则表达式引擎再次尝试匹配模式中的第一个字符——^ 元字符时，发现当前位置不在测试文本中的第一个字符之前，那么匹配失败。因此，整个模式也不会匹配。也就是说，只要当前位置不是测试文本的开始处，那么匹配就失败。

6.1.2　^ 元字符和多行模式

在前面的例子中，测试文本只有一行，因此不用考虑 ^ 元字符会与测试文本的开始处匹配，还是会与每一行的开始处匹配这个问题——因为在只有一行文本的情况下，这两个概念指的是同一位置。但是，在某些工具和语言中，可以修改 ^ 元字符的行为让它匹配每一个行的开始位置，或者只匹配测试文本中第一行的开始位置。

例如，使用 Komodo Regular Expression Toolkit 对下面的测试文本：

```
This

Then
```

应用下面的模式，那么将找不到匹配项：

```
^The
```

图 6-3 显示的是匹配失败的结果。

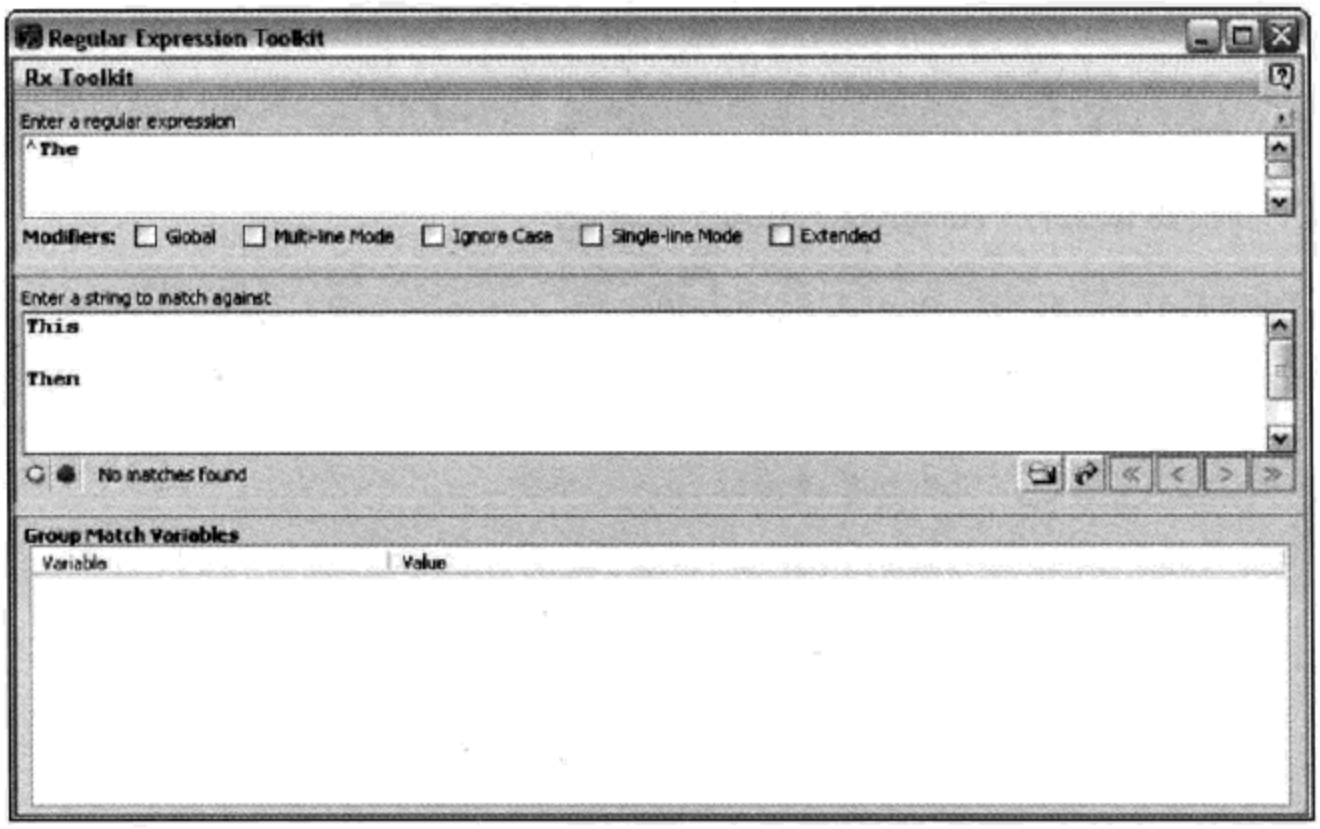


图　6-3

然而，如果选中 Multi-Line Mode 复选框，则位于第二行的字符序列 The 会突出显示出来，在下面的灰色区域中会显示信息 Match succeeded: 0 groups，如图 6-4 所示。

在使用多行模式的情况下，位于 Unicode 换行符之后的位置(即下一行的开始位置，译者注)会与测试文件的开始位置等同。Unicode 换行符匹配可以用于表达换行意图的任何字符或者字符组合。

并不是所有的编程语言都支持多行模式。在本书后面介绍具体编程言的章节中将讨论它们各自对多行模式的支持问题，并进行相应的示范。

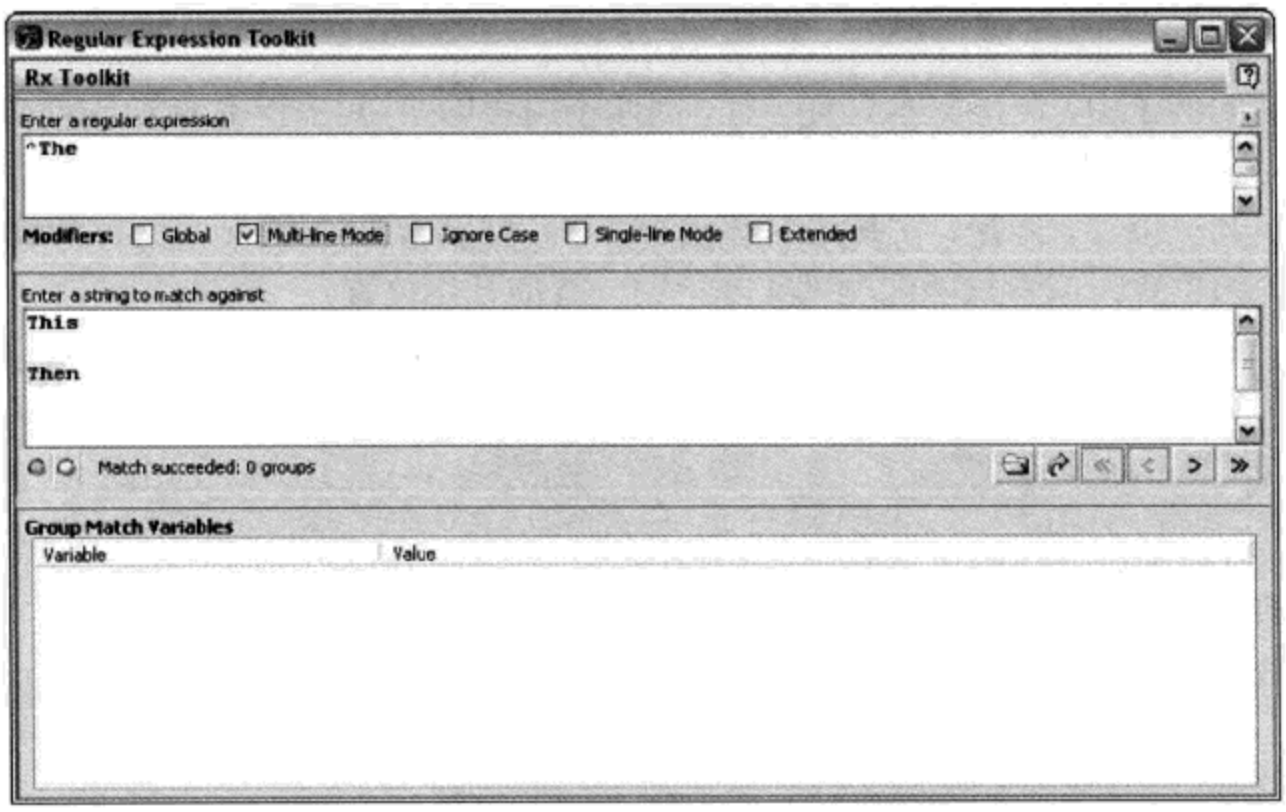


图 6-4

试一试：^ 元字符与多行模式

使用下面的测试文件 TheatreMultiline.txt 试验：

```
The Thespian Theatre opens at 19:00.
Then theatrical people enter the building.

They greatly enjoy the performance.

The interval is the time for liquid refreshment.
```

注意，测试文件中的每一行文本都以字符序列 The 开头。

有些工具在默认情况下会启用多行模式，比如这里使用的 PowerGrep。

(1) 打开 PowerGrep，并选中 Regular expression 复选框。

(2) 在 Search 文本框中输入正则表达式模式 ^The。

(3) 在 Folder 文本框中输入 C:\BRegExp\Ch06。如果把下载的文件放在其他地方，可以对这一步中的路径进行相应调整。

(4) 在 File mask 文本框中输入 TheatreMultiline.txt。

(5) 单击 Search 按钮，并在观察 Results 区域中的结果，结果如图 6-5 所示。

注意每一行开始处的字符序列 The 都被突出显示出来了，这表明 PowerGrep 中的默认行为就是多行模式。

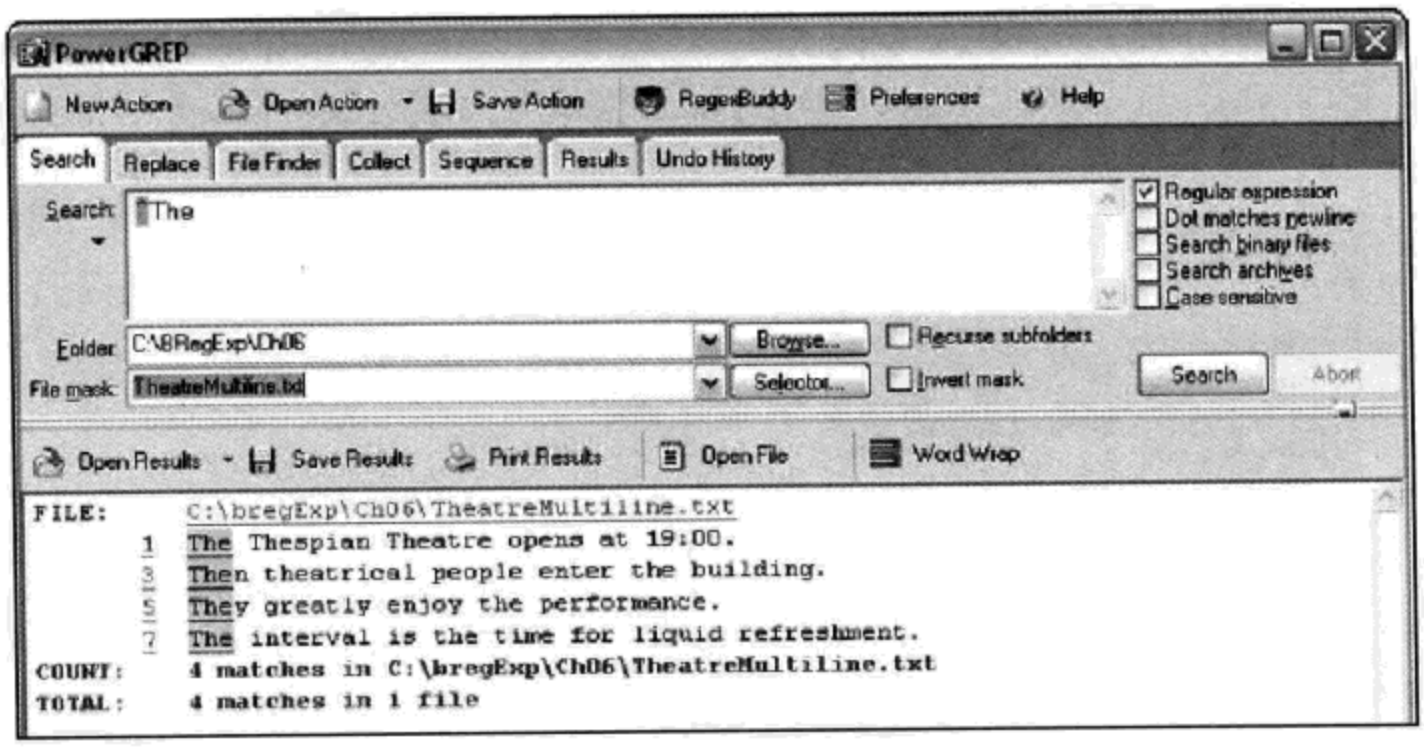


图　6-5

6.1.3　$ 元字符

通过 ^ 元字符可以指定出现在文件开始处或文件中一行开始位置的字符序列。而 $ 元字符对此提供了补充功能，通过 $ 元字符可以指定出现在一个文件结束位置或文件中一行结束位置之前的字符序列。

首先，来看一个简单的例子，例子中使用的测试文件 Lathe.txt 中只包含一行文本：

```
The tool to create round wooden or metal objects is the lathe
```

如上所见，在这个测试文本中字符序列 the 出现了不止一次。为演示$元字符的作用，我们故意忽略了作为一个句子结束的句点字符。下面的模式将只与位于测试字符串结尾之前的字符序列匹配：

```
the$
```

试一试：使用 $ 元字符

下面的例子演示的是模式 the$的用法：

(1) 打开 PowerGrep，并选中 Regular expression 复选框。

(2) 在 Search 文本框中输入模式 the$。

(3) 在 Folder 文本框中输入 C:\BRegExp\Ch06。

(4) 在 File mask 文本框中输入 Lathe.txt。

(5) 单击 Search 按钮，并观察显示于 Results 区域的结果，如图 6-6 所示。

注意，结果中只有一个匹配项，而位于测试文本开头的字符序列 The 和单词 lathe 前的 the 都没有匹配。

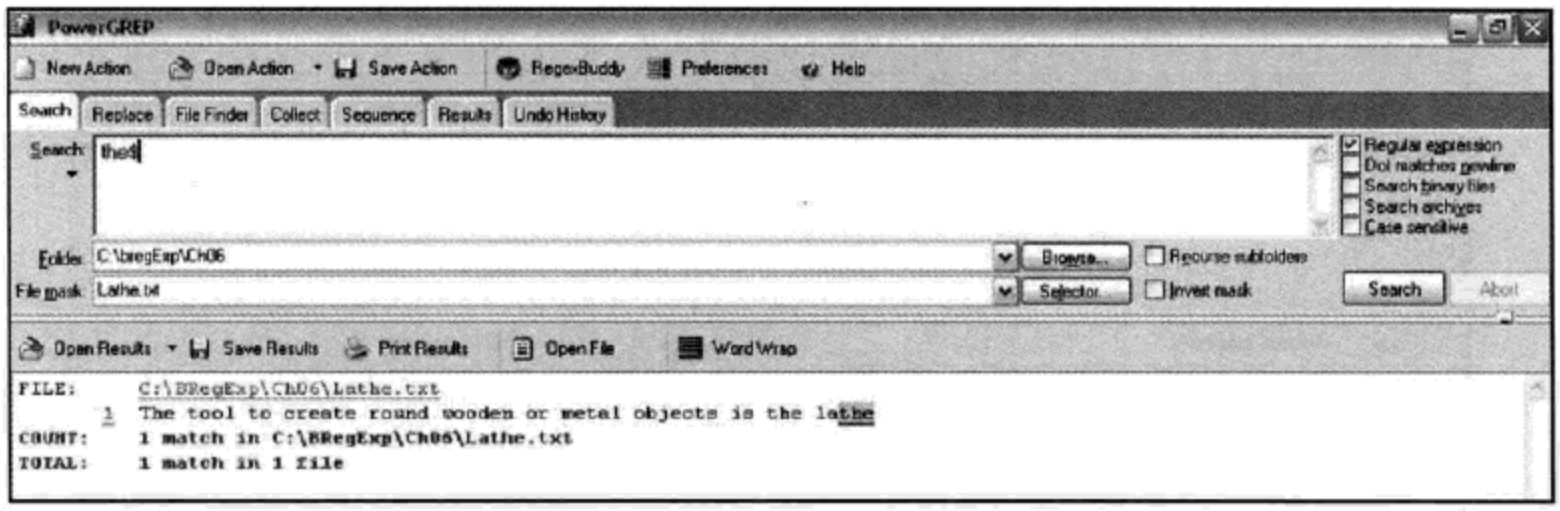


图 6-6

(6) 在 Search 文本框中删除 $ 元字符。

(7) 单击 Search 按钮，并观察 Results 区域中的结果。

注意，从模式中删除 $ 元字符后，结果中出现了三个匹配项(这里没有提供示意图)。第一个匹配项是位于测试文本开头的 The。出现这个匹配项是因为 PowerGrep 中的默认搜索行为是不区分大小写的。第二个匹配项是单词 lathe 前面的 the。而第三个则是包含在 lathe 中的 the。

工作原理

PowerGrep 的默认行为是进行不区分大小写的匹配。当正则表达式引擎在第 6 步之后开始匹配时，它会从第一个的 The 开始匹配。正则表达式引擎尝试匹配 The 并成功了。最后，正则表达式引擎尝试将 $ 元字符与测试文本中小写的 e 后面的位置进行匹配。而该位置不是这个测试字符串的结尾，所以匹配失败。由于模式中的一个组件匹配失败，所以整个模式也匹配失败。

尝试匹配的进程继续前行。当正则表达式引擎到单词 the 之前的位置时，模式中的前三个字符成功匹配。但是，如上面所分析的，接下来的位置并没有与 $ 元字符匹配。所以，整个模式匹配失败。

但是，当正则表达式引擎到 lathe 中 a 后面的位置并尝试匹配时，找到一个匹配项。模式中的第一个字符、小写的 t 与 lathe 中的下一个字符、小写的 t 匹配。模式中的第二个字符、小写的 h，与 lathe 中的 h 匹配。模式中的第三个字符、小写的 e，与 lathe 中的 e 匹配。而模式中最后的 $ 元字符也找到了匹配项，因为 lathe 中的 e 是这个测试字符串中的最后一个字符。由于模式的所有组件都匹配，所以整个模式匹配成功，并且 lathe 中的字符序列 the 也被突出显示出来，如图 6-6 所示。

1. 多行模式下的$元字符

与 ^ 元字符类似，$ 元字符的匹配行为在多行模式下也会发生变化。但并非所有工具或语言都能够支持在多行模式下使用 $ 元字符。

支持在多行模式下使用 $ 元字符的工具或语言使用 $ 元字符匹配直接位于 Unicode 换行符之前的位置。有一些也匹配测试字符串结尾之前的位置，但不是全都如此，这一点我们在后面将会看到。

下面使用的测试文件是 ArtMultiple.txt，其内容如下：

```
A part for his car

Wisdom which he wants to impart

Leonardo da Vinci was a star of medieval art

At the start of the race there was a false start
```

注意，为测试 $ 元字符的作用，我们把每个句子结尾处的句点都省略了。

试一试：多行模式下的$元字符

本例演示在多行模式下使用 $ 元字符的过程：

(1) 打开 PowerGrep，并选中 Regular expression 复选框。

(2) 在 Search 文本框中输入模式 art。

(3) 在 Folder 文本框中输入 C:\BRegExp\Ch06。

(4) 在 File mask 文本框中输入文本 ArtMultiple.txt。

(5) 单击 Search 按钮，并观察 Results 区域中的结果，如图 6-7 所示。

注意，字符序列 art 不论是在一行的结尾还是在其他位置都被匹配了——在本例中，即第 1 行中的 part 和第 7 行中的第一个 start。

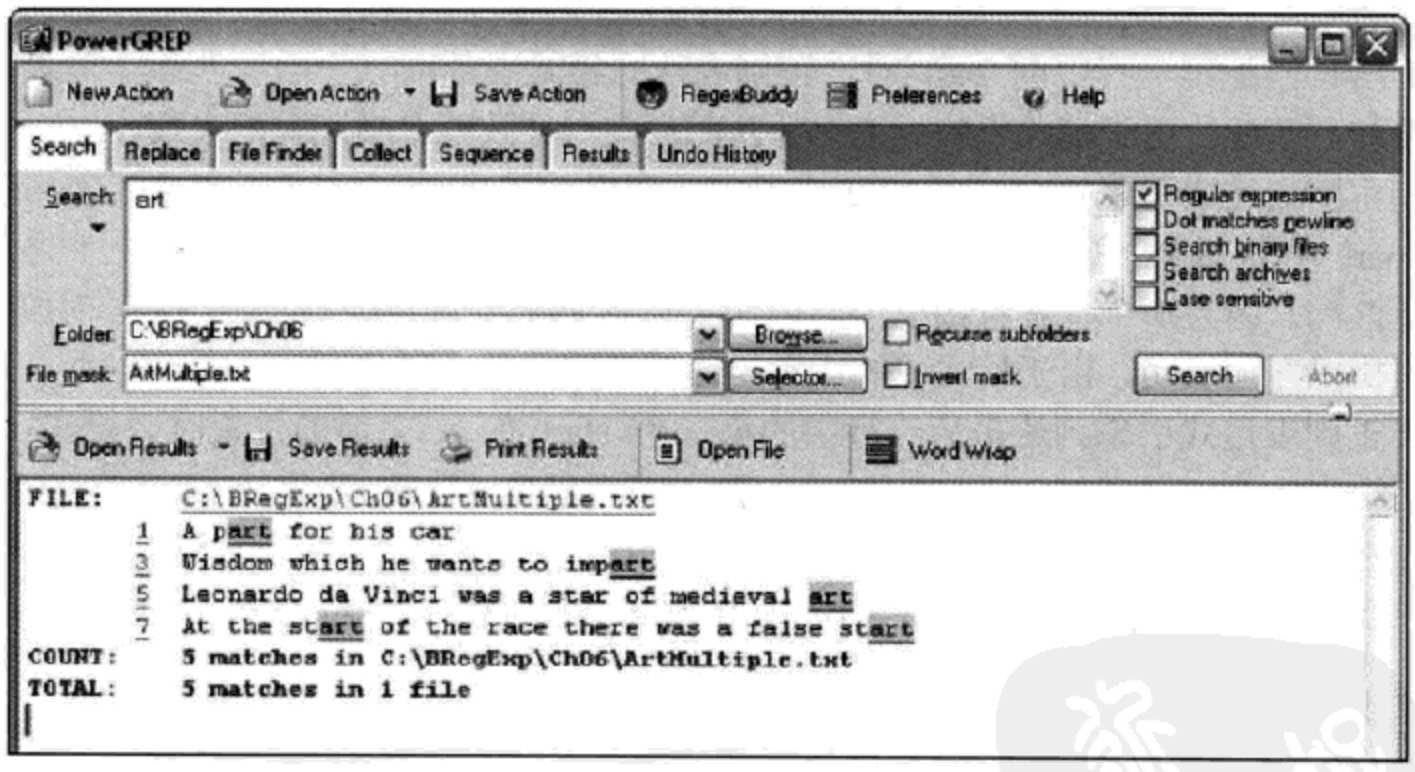


图　6-7

(6) 修改正则表达式模式，在其末尾加上 $ 元字符，得到 art$。

(7) 单击 Search 按钮，并观察 Results 区域中的结果，如图 6-8 所示。

注意，原先第 1 行中的单词 part 和第 7 行中第一个 start 中与模式 art 匹配的文本现在都不匹配了。这是因为它们并没有出现在一行的结尾处，而 $ 元字符匹配的必须是一行的结尾位置。

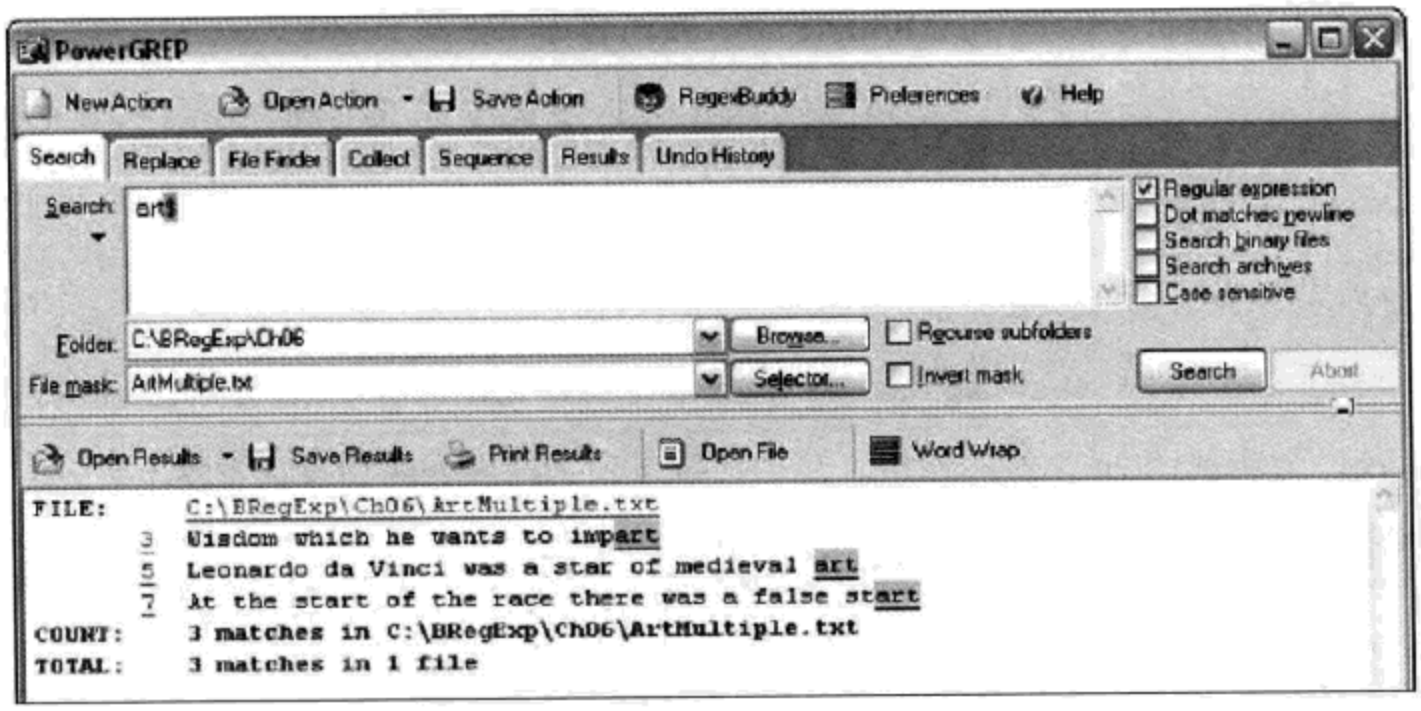


图 6-8

工作原理

当正则表达式模式是简单的三个直接量字符 art 时，这三个直接量字符不管处于什么位置都会匹配。

然而，当把 $ 元字符添加到模式中后，正则表达式引擎在匹配三个直接量字符序列 art 的同时，还必须匹配一个位于 Unicode 换行符之前，或者位于测试字符串结尾之前的位置。

当尝试匹配第一行中的 part 时，正则表达式模式中的前三个字符匹配，但最后的 $ 元字符却不匹配。由于模式中的一个组件匹配失败，导致整个模式匹配失败。

当正则表达式引擎到 impart 中 a 之前的位置时，模式 art$ 中的前三个字符可以分别与 impart 中的 a、r 和 t 匹配。最后，要将模式中的 $ 元字符与 impart 中 t 之后的位置进行匹配。因为那个位置直接位于一个 Unicode 换行符(也就是那一行的最末尾的位置)之前，所以匹配成功。由于模式中的所有组件都匹配，整个模式匹配。

当正则表达式引擎到达最后一行中的第二个 start 的 a 之前的位置时，模式 art$ 中的前三个字符可以分别与 start 中的 a、r 和 t 成功匹配。最后，要将 $ 元字符与 start 中 t 之后的位置进行匹配。因为那个位置直接位于测试字符串的结尾(即测试文件的末尾位置)之前，所以匹配成功。由于模式中的所有组件都匹配，整个模式匹配。

2. 同时使用 ^ 和 $ 元字符

同时使用 ^ 和 $ 元字符可以用于查找由要找的字符组成的行。这种用法在验证用户输入时是非常有用的。

看一个测试文件 ABCPartNumbers.txt，其内容如下：

```
ABC123

There is a part number ABC123.

ABC234
```

```
A purchase order for 400 of ABC345 was received yesterday.

ABC789
```

注意，其中有些行只包含零件编号，而有的行则除了包含零件编号外还包含其他文本内容。

我们的目的是找出只由零件编号构成的行。问题定义如下：

匹配一行的开始位置，后跟直接量字符串 A、B 和 C，后跟三个数字，最后跟一个行结尾的位置或者一个字符串结尾的位置。

试一试：匹配零件编号

本例演示的是在同一个模式中同时使用 ^ 和 $ 元字符的用法。

(1) 打开 OpenOffice.org Writer，然后打开测试文件 ABCPartNumbers.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框，然后选中 Regular expressions 和 Match case 复选框。

(3) 在 Search for 文本框中输入模式 ^ABC[0-9]{3}$。

(4) 单击 Find All 按钮，并观察突出显示的文本，如图 6-9 所示。注意有三个表示零件编号的字符序列作为匹配项突出显示，而另外两个零件编号没有突出显示是因为它们与模式不匹配。

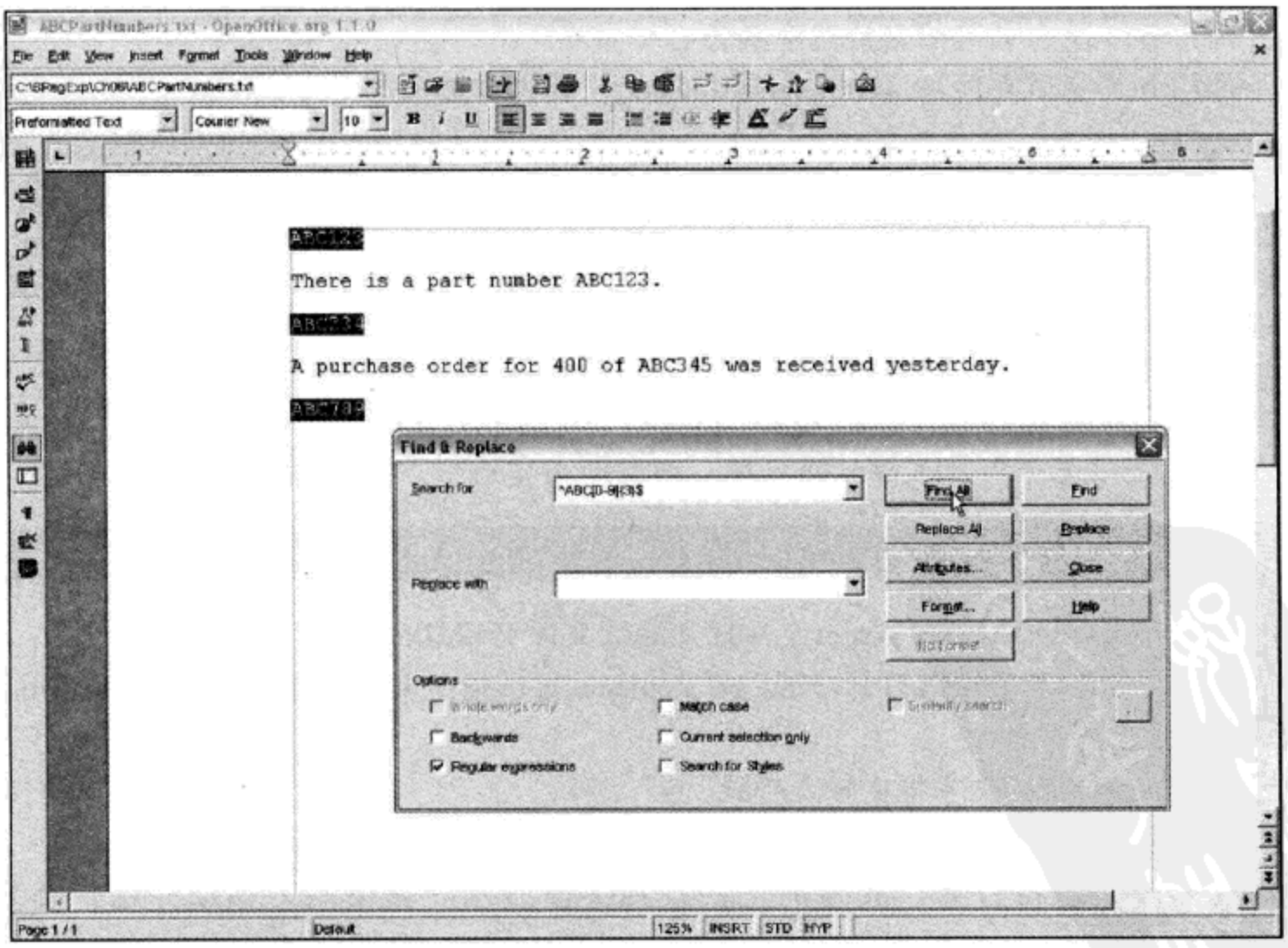


图　6-9

工作原理

正则表达式引擎从测试文本的开始处开始匹配。首先，它会将 ^ 元字符与当前所在位置匹配。结果匹配成功。然后，又尝试匹配模式中的直接量字符 A 和第一行中的第一个字符——大写的 A。匹配成功。匹配进程再次成功匹配模式中的直接量字符 B 和 C。然后，正则表达式引擎尝试匹配模式[0-9]{3}。它先将字符类[0-9]与测试文本中的字符 1 进行匹配——匹配成功。然后再次用字符类 [0-9] 与字符 2 匹配——匹配也成功。然后根据限定符 {3} 的指示第三次使用字符类 [0-9] 与字符 3 进行匹配——这次匹配也成功。最后，它将$元字符与字符 3 之后的位置进行匹配。因为这个位置直接位于一个 Unicode 换行符之前，所以它与 $ 元字符也匹配。模式中的每个组件都匹配，整个模式就匹配。

在第二行的开始位置处，正则表达式成功地匹配了 ^ 元字符。然后它继续尝试将模式中的直接量字符 A 与该行的第一个字符、大写的 T 匹配。这次匹配失败。而且，接下来在该行中匹配 ^ 元字符的任何尝试都以失败告终，因为后面的任何位置都不是一行的开始位置。

3. 匹配空白行

组合使用 ^ 和 $ 元字符的一个潜在的应用是匹配空白行。因为 ^ 元字符匹配行的开始位置，而 $ 元字符匹配位于一个 Unicode 换行符或者测试字符串结尾之前的位置，所以下面的模式能匹配一个空白行。

```
^$
```

然而，并非所有工具都支持这个模式。

下面用到的测试文件是 WithBlankLines.txt，其内容如下：

```
Line 1

Line 3 which follows a blank line

Line 5 which follows a second blank line

Line 7 which follows a third blank line
```

在第 7 行之后和测试文件结尾之间，还有两个空白行。

试一试：替换空白行

(1) 打开 OpenOffice.org Writer，并打开测试文件 WithBlankLines.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框，并选中 Regular expressions 和 Match case 复选框。

(3) 在 Search for 文本框中输入模式 ^$。

(4) 单击 Find All 按钮，并观察结果。其结果如图 6-10 所示。

除最后两个空白行外，其他的空白行都突出显示。如果向下移动光标，会发现 OpenOffice.org Writer 忽略了在记事本上打开 WithBlanklines.txt 时会显现的存在的某一个空

白行。而如果在 OpenOffice.org Writer 中通过手工再次输入被丢掉的空白行，则又会多出一个匹配的空白行。OpenOffice.org Writer 似乎不会匹配位于文件末尾的空白行。

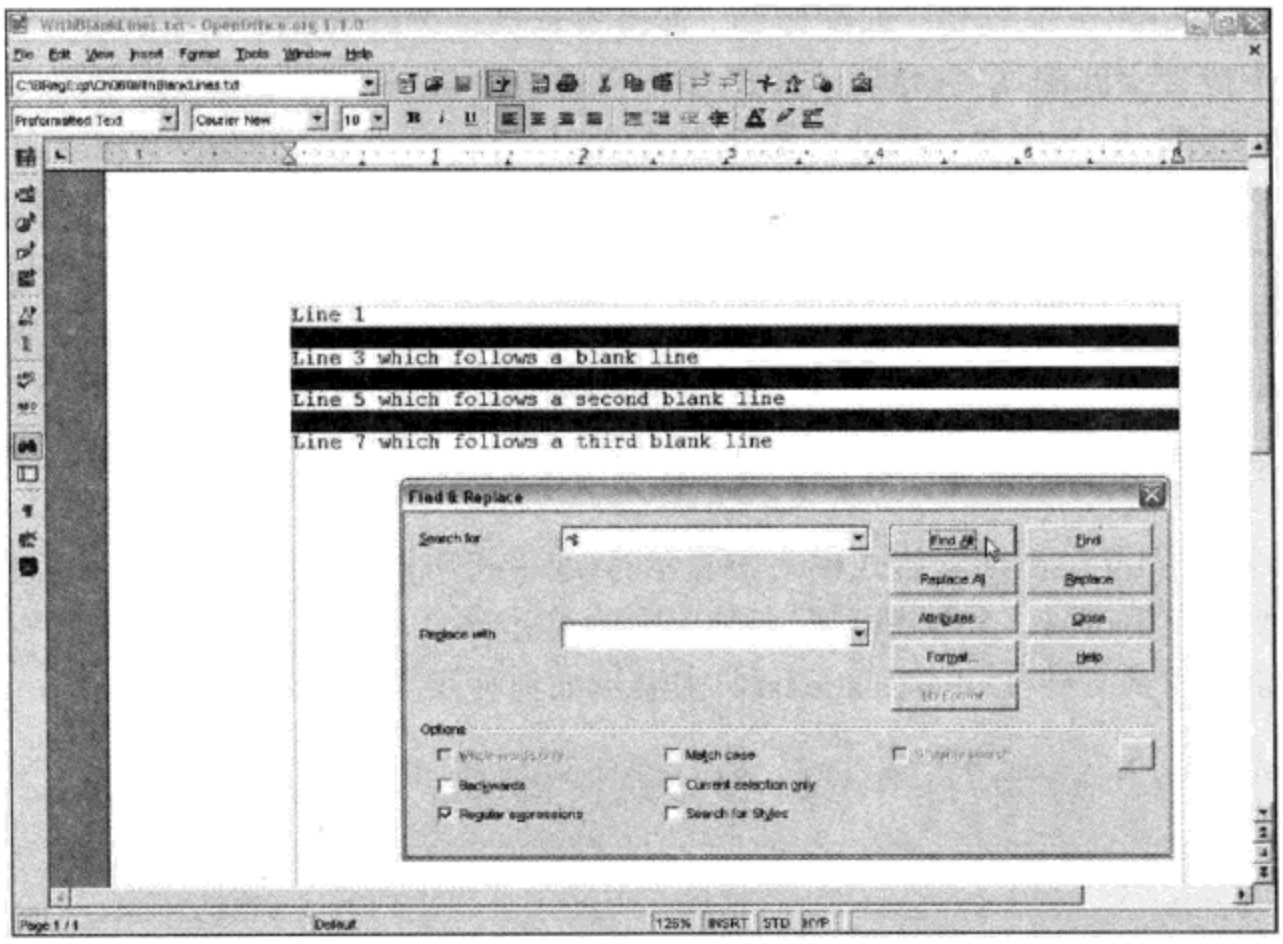


图　6-10

(5) 单击 Replace All 按钮，并观察结果，如图 6-11 所示。注意，前三个突出显示的空白行已被删除了。

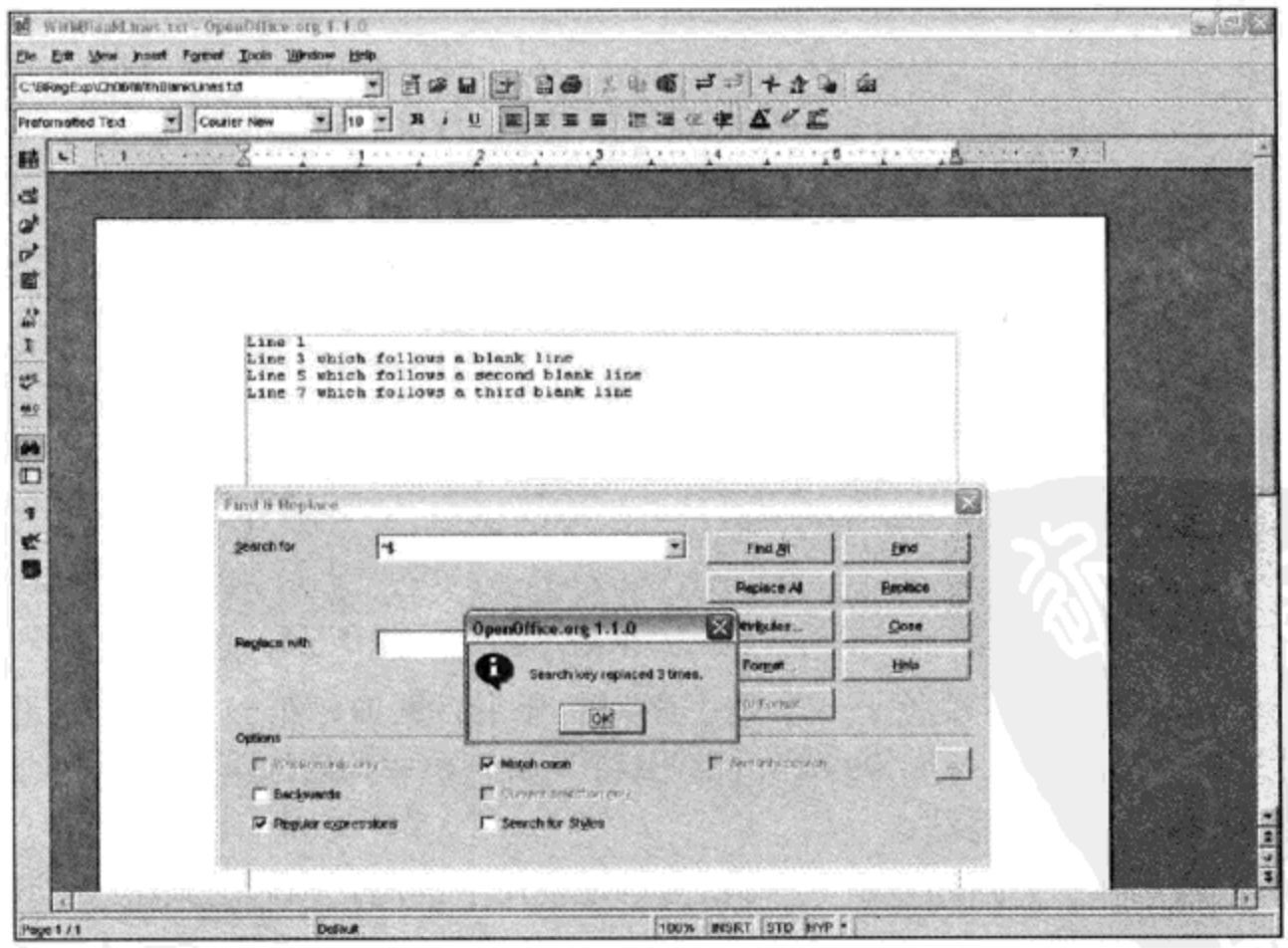


图　6-11

工作原理

原始测试文件中的第二行是一个空白行。当正则表达式引擎位于该空白行的开始位置时，它会尝试匹配 ^ 元字符。匹配成功。在没有移动位置的情况下，正则表达式引擎又尝试用同一个位置来匹配 $ 元字符。由于那个位置直接位于一个 Unicode 换行符之前，所以该位置也匹配 $ 元字符。因此，整个模式匹配。在 OpenOffice.org Writer 中，匹配并突出显示的空白行与整个文本区同宽。

当正则表达式引擎位于原始测试文件的第三行的开始位置时，它首先尝试匹配 ^ 元字符，匹配成功。然后，又尝试用同一个位置来匹配 $ 元字符。因为该位置的后面是一个大写的 L，所以这个位置并不符合位于一个 Unicode 换行符之前这个条件。所以，匹配失败。

4. 处理美元表示的金额

由于 $ 元字符在正则表达式模式中表示行(或字符串)的结束位置，所以不能用它匹配文档中的美元货币符号。如果想匹配字符串中的美元符号，必须使用 \$ 转义序列。

下面通过测试文件 DollarUsage.txt，来演示如何使用 \$ 转义序列：

```
The pound, £, and US dollar, $, are major global currencies.

$99.00

99,00$

$1,000,000

$1000

$1,000

$1,000.00

$0.50

$2 # A Perl variable

$ 0.99

$myVariable

$2.25
```

如你所见，$符号不单单位于一个数字序列的开头。例如，第一行表明美元符号如何出现在一段文本中。而第三行，99.00$，说明$符号如何在非英语地区表示美元金额，或者说是一个不讲英语的人的写法。

要匹配直接量 $ 符号很直观——可以简单地使用下面的正则表达式模式，它匹配文本中出现的所有美元符号：

```
\$
```

图 6-12 显示的是在 PowerGrep 中使用这个简单的模式的结果。

假设只想查找那些后跟数字的美元符号。即使这个目标看起来很简单，也不能一概而论。比如说，倒数第三行的美元符号后面还带有一个空格符，如果想匹配所有相关的 $ 符号，必须将这种情况考虑在内：

```
$ 0.99
```

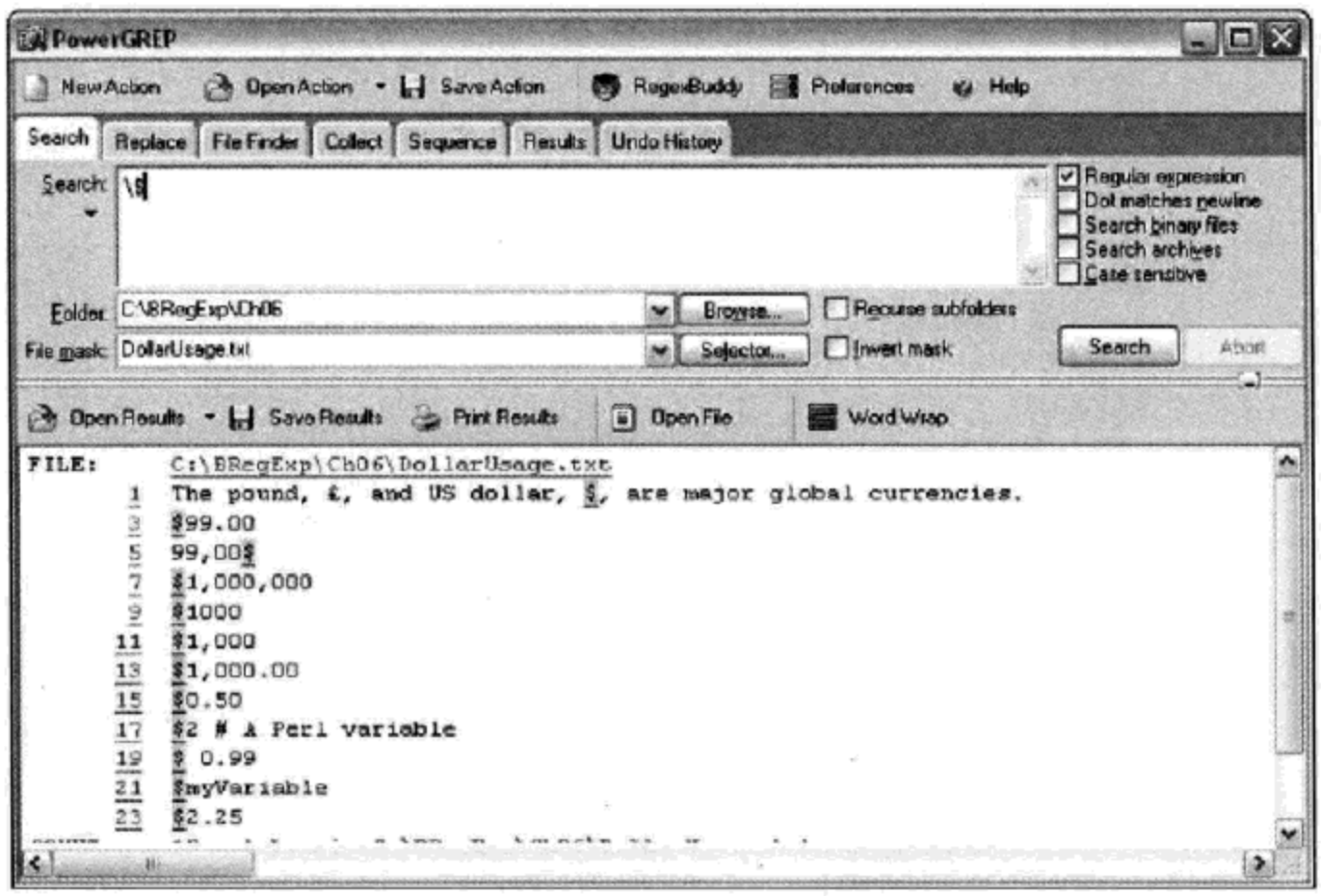


图　6-12

注意在倒数第四行中，美元符号后面会以一种能够接受的表示美元金额的方式跟着一个数字，但实际上这一行的含义并非如此：

```
$2 # A Perl variable
```

要确定类似这样的数据是否属于要处理的数据范畴，就需要了解这些数据以及数据的含义是什么。为简单起见，假设想匹配所有$符号后跟数字，并带有或不带有空白符的数据。

根据正则表达式实现不同，表达数字的方式有以下几种：\d、[0-9] 和 [:digit:]。

首先，尝试匹配美元符号后跟一个或多个数字加一个句点，再跟零个或多个数字的情况。用正则表达式模式表达如下：

```
\$[0-9]+\.[0-9]*
```

其中，\$ 匹配直接量美元符号。而字符类 [0-9] 匹配一个数字，其后的 + 限定符表示至少要有一个数字。接下来的直接量句点字符由转义序列 \. 表示。最后，模式 [0-9]* 表示可以跟在句点后面的零个或多个数字。

图 6-13 显示的是在 OpenOffice.org Writer 中对 DollarUsage.txt 应用这个模式的结果。注意只有三个匹配项突出显示。你知道$1 000 000 和$1000 为什么没有被匹配吗？

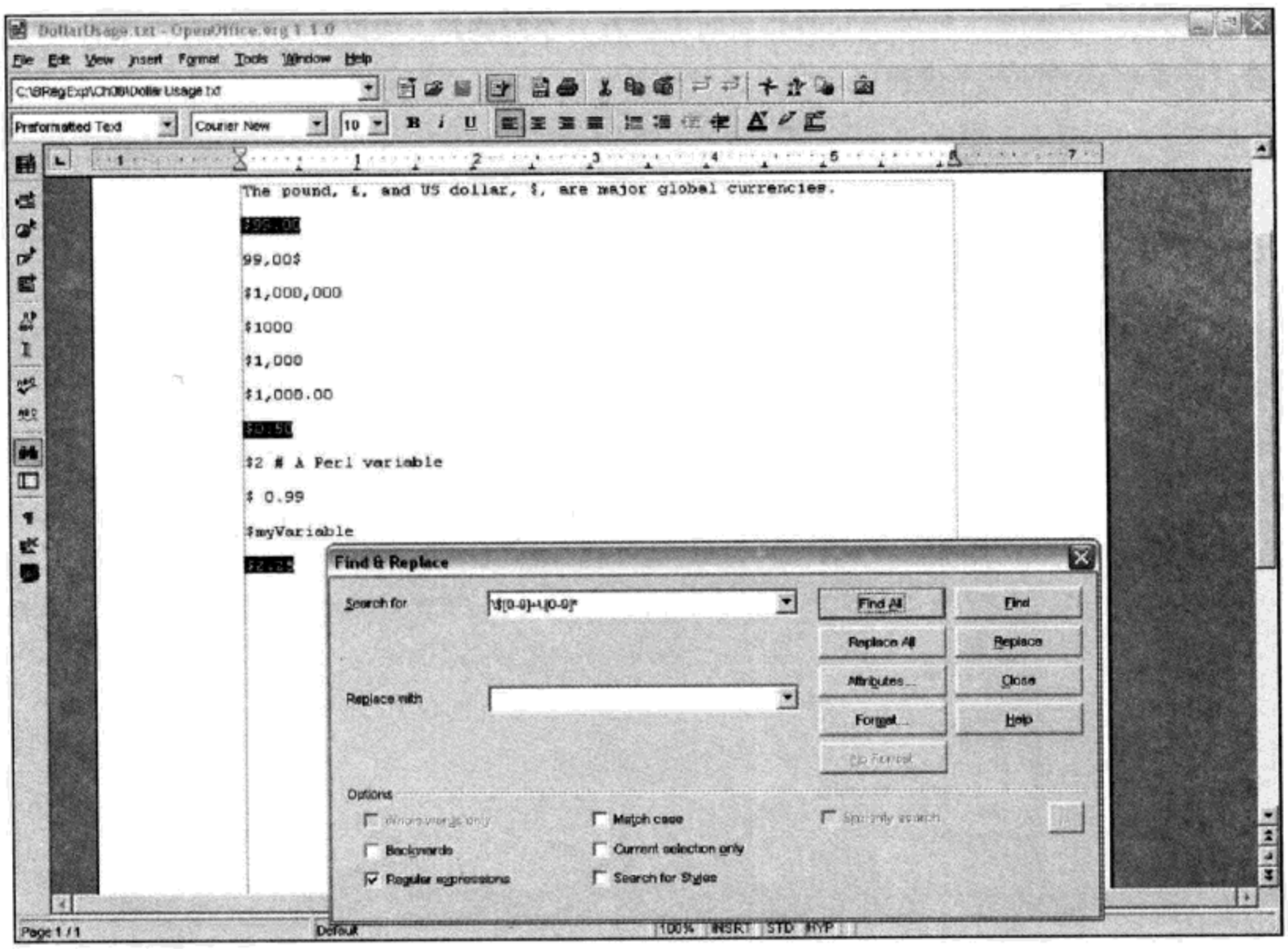


图 6-13

我们首先来讨论$1000，它不匹配是因为模式 \.(匹配一个句点)在其中没有匹配项。所以，如果要允许匹配不包含小数点的美元值，就必须添加一个 ？限定符来表示小数点是可选的。修正后的模式 \.? 匹配零个或一个小数点。

当对 DollarUsage.txt 应用如下这个修正后的模式时，会看到如图 6-14 所示的结果。

```
\$[0-9]+\.?[0-9]*
```

对于包含以逗号作为千分位或百万分位分隔符的美元值，是其中的逗号导致了所有相关的美元值没有被匹配。若在上面的模式基础上再加入一个允许一个或多个空格符存在的模式就完成了本章的这个最终模式：

```
\$ *[0-9]+\.?[0-9]*
```

通过在模式前面的空格之后使用 * 限定符，可以允许在美元符号后面出现不止一个空格字符。

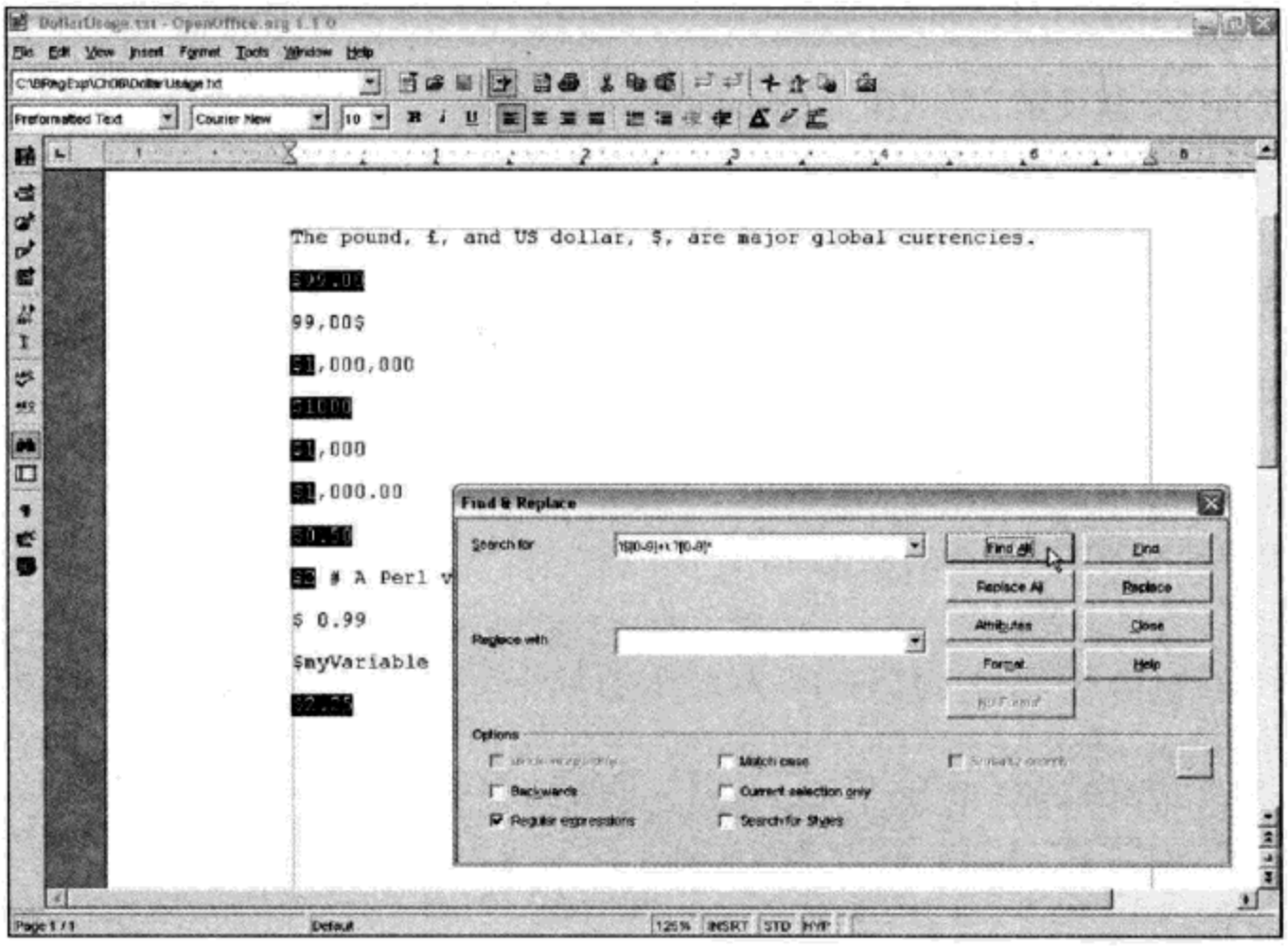


图　6-14

5. 重温 IP 地址的例子

在第 5 章中，花费相当时间研究了如何使用字符类匹配 IP 地址，当时使用的是测试文件 PLike.txt，其内容如下：

```
12.12.12.12

255.255.256.255

12.255.12.255

256.123.256.123

8.234.88.55

196.83.83.191

8.234.88,55

88.173.71.66

241.92.88.103
```

在了解 ^ 和 $ 元字符的含义和作用之后，就能为那个例子找到一个圆满的解决方案。

试一试：匹配 IP 地址

下面的指令假设你已经关闭了 OpenOffice.org Writer。

(1) 打开 OpenOffice.org Writer，然后打开测试文件 IPLike.txt。

(2) 用 Ctrl+F 快捷键打开 Find & Replace 对话框，并选中 Regular expressions 和 Match case 复选框。

(3) 在 Search for 文本框中输入正则表达式模式 ^((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\.){3}(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])$ 。

(4) 单击 Find All 按钮，并观察结果，其结果如图 6-15 所示。注意，包含 256 的那一行没有匹配，这正是我们想要的结果。

这里所使用的正则表达式模式如下：

```
^((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\.){3}(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])$
```

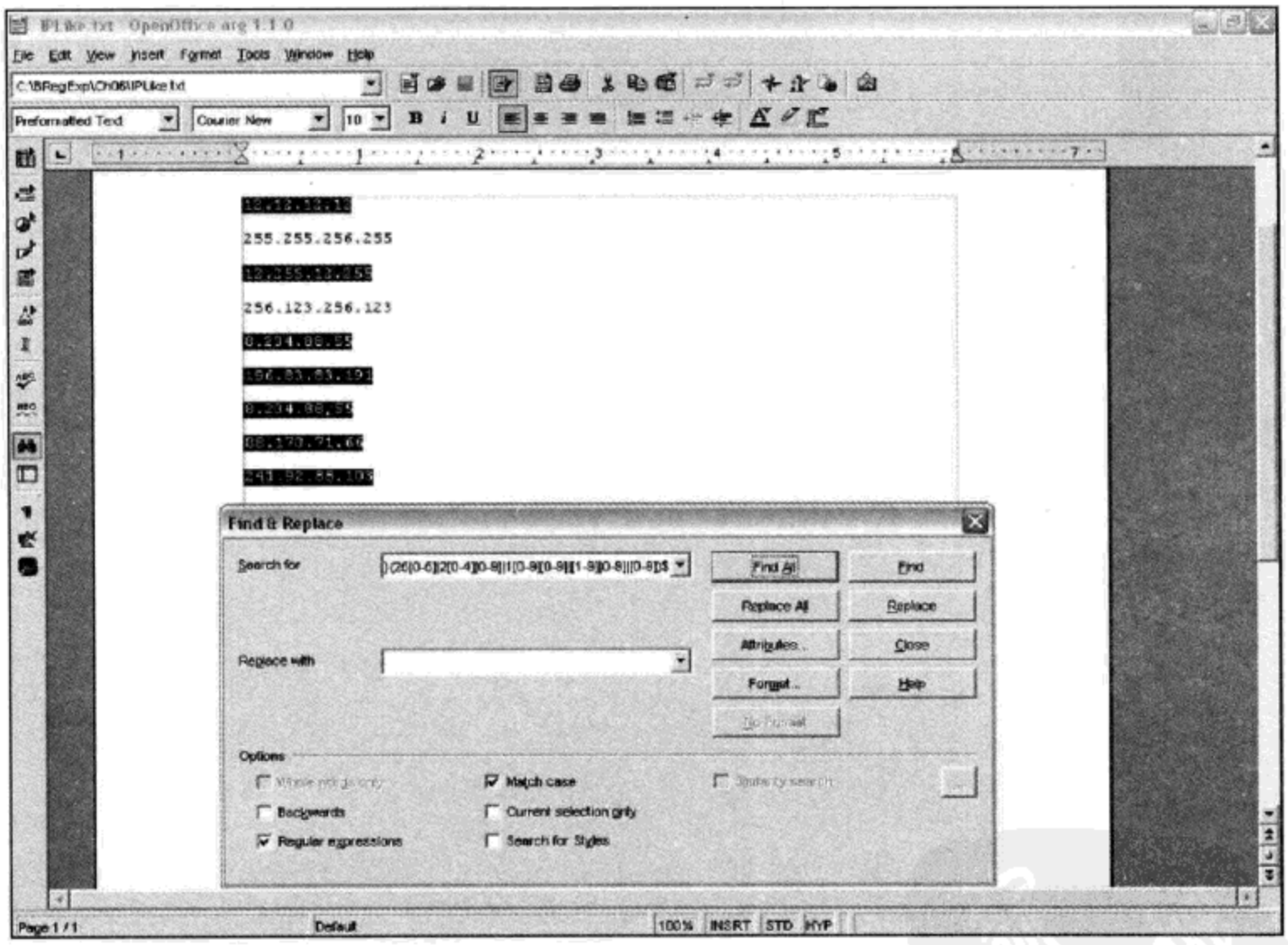


图 6-15

工作原理

首先，把这个正则表达式按照它的组件进行分解。

开始的 ^ 元字符表示只有从一行的开始位置匹配时才有效。

下面的组件表示的是一些数值的选项，每个选项都后跟一个直接量句点：

```
((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\.){3}
```

记住匹配直接量句点的转义序列是 \.。如果在这个模式中使用的是一个句点，那么测试文本会匹配，但是可能匹配的是任何字母数字字符。这样会导致出现如下所示的匹配项，很明显它不是一个 IP 地址：

```
12G255F12H255
```

使用 . 元字符会丢失使用 \. 元字符时才会有的很多特殊性。

下面的模式开始匹配时，它先匹配直接位于一行开始位置之后的数值：

```
((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\.){3}
```

这个模式的含义是，如果在嵌套的圆括号内部有任何选项在行的开始位置与一个直接量句点字符之间找到匹配项，匹配就成功一次。对于该结构的选项，只有数字从 0 到 255 可以匹配。

下面的模式第二次匹配时，知道在要匹配的数值前面是一个直接量句点字符(因为第二次匹配紧随第一次匹配之后，而第一次匹配结束于一个直接量句点)：

```
((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\.){3}
```

换句话说，要查找的是一个处于两个直接量句点之间的、0~255 的数字。

类似地，第三次匹配结果的前面和后面也都是一个直接量句点(前面的句点在第二次匹配中与 \. 元字符匹配)。

如果模式中所有前面的组件都匹配，那么正则表达式引擎就会尝试匹配下面的模式：

```
(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])$
```

因为这个模式以 $ 元字符结尾，所以它只匹配一个 0~255 的数字，而且这个值必须位于一个直接量句点(在使用前面的模式组件进行第三次匹配时所匹配的最后一个字符)后面并后跟一个 Unicode 换行符。

6.2　什么是词

有关词是由什么组成的这个命题，初看起来似乎很明显。但是，如果你被人问起下面这些行的字符序列中哪一个是词，你会怎么回答呢？而你又会用什么标准来支持自己的观点呢？

```
cat
ja
jar
Nein
parr
smolt
pomme
Claire
spil
```

你觉得这里面的有几个词？可能 cat 和 jar 在你的选择范围内。但是，ja 和 Nein 呢？

如果你懂德语，你肯定也会把它们两个也列入词的范围中。同样地，如果你会法语，你会把 pomme 作为一个词；但是，如果你不懂法语，恐怕要持相反的观点了。而如果一个讲英语的人熟悉大西洋鲑鱼(Atlantic salmon)的成长历程，也会毫不费力地认出 parr(幼鲑)和 smolt(两岁大的小鲑鱼)分别是它们成长过程中的两个时期。但恐怕有不少讲英语的人都不熟悉这两个词。

很明显，要想让一个文字处理程序知道这些词哪个属于英语、哪个属于法语，而哪个属于德语，或者属于其他语种是不现实的。类似地，我们也不能指望文字处理程序熟悉所有技术性领域的专业词汇。因而，就需要另外一种技术——更加机械的技术——识别词的边界。

6.3 识别词边界

可以用两个位置来定义词边界：一个是构成一个单词的字符序列的开始位置，另一个是构成一个单词的字符序列的结束位置。

根据所使用的工具或语言不同，有一些元字符可以用于匹配位于词的开始位置的词边界，或者匹配词的结束位置的词边界，或者匹配前面两种词边界。

6.3.1 \< 语法

\< 元字符用于识别位于一个词开始位置的词边界。它的前面是一个非字母字符(比如，一个空格符)或者是一行的开始位置。

下面试验要用到测试文件 BoundaryTest.txt，其内容如下：

```
ABC DEF GHI
GHI ABC DEF
ABC DEF GHI
CAB CBA AAA
```

相应的问题定义如下：

匹配一个大写的 A，这个 A 必须直接位于一个词边界之后。

换句话说，匹配一个前面是非文字字符、一个字符串开始或者行开始位置的大写字母 A。

试一试：匹配词开始处的词边界

(1) 打开 OpenOffice.org Writer，然后打开测试文件 BoundaryTest.txt。

(2) 用 Ctrl+F 快捷键打开 Find & Replace 对话框，并选中 Regular expressions 和 Match case 复选框。

(3) 在 Search for 文本框中输入模式\<A。

(4) 单击 Find All 按钮，并观察结果，如图 6-16 所示。

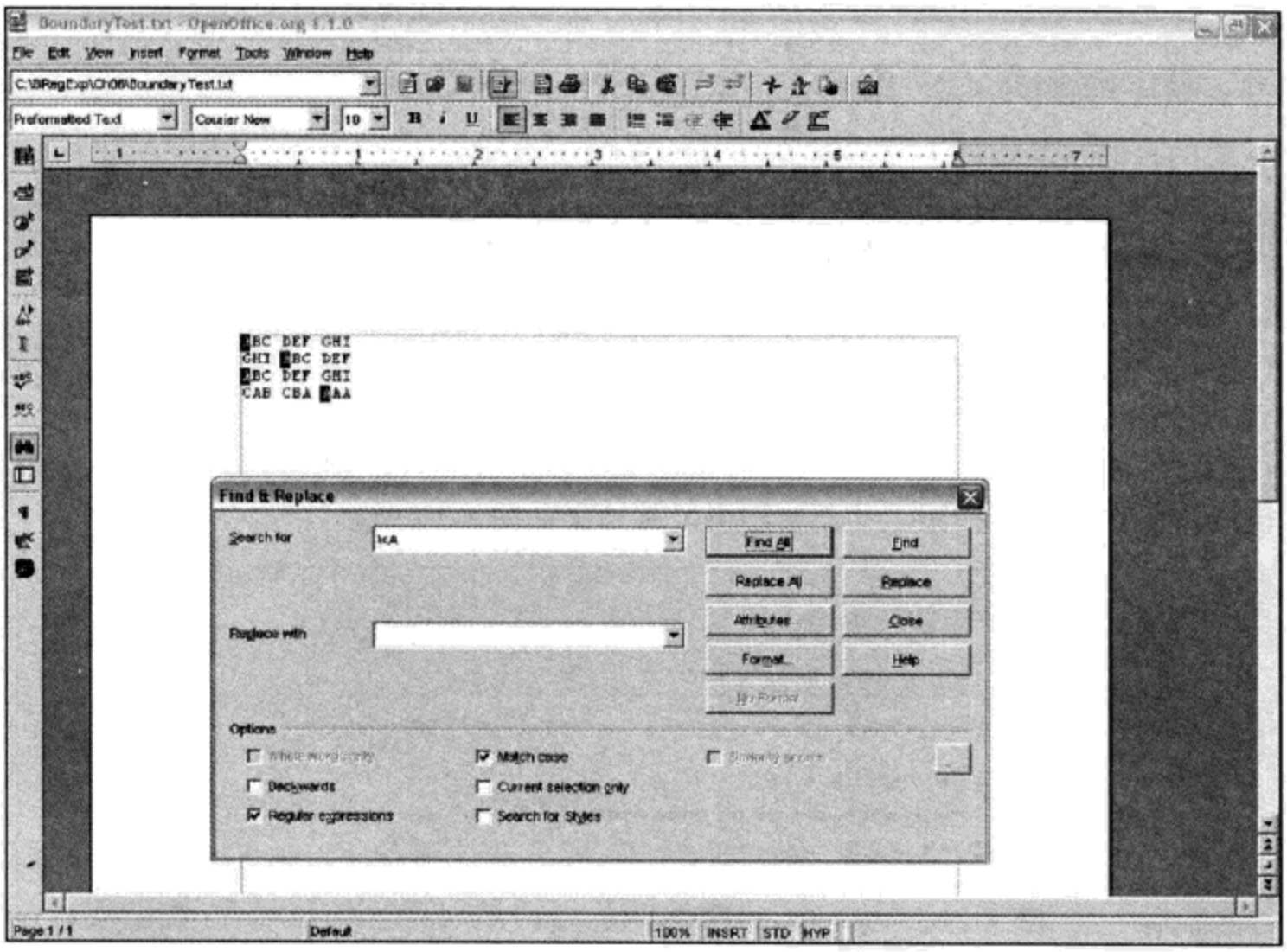


图　6-16

工作原理

第一行中，ABC 中的 A 位于文本的开始位置之后，所以匹配成功。

第二行中，ABC 中的 A 位于空格符(一个非字母字符)之后，所以也匹配成功。

第三行中，ABC 中的 A 位于该行开始位置之后，所以它也与模式 \<A 匹配。

最后一行，CAB 中的 A 前面有一个字母字符，所以它与模式 \<A 不匹配。而 CBA 中的 A 后面是一个非字母字符，但前面却是一个字母字符，所以也与模式 \<A 不匹配。

AAA 中的第一个 A 前面是一个非字母字符，所以与模式 \<A 匹配。然后，第二和第三个 A 前面都是一个字母字符，所以没有匹配。

6.3.2　\> 语法

\> 元字符表示位于一个字母字符序列结尾处的词边界。换句话说，它匹配词结尾处的词边界。

这里要用的测试文件是 EndBoundary.txt，其内容如下：

```
Theodore said "This is a lathe

I shaved today and my new shaving cream made a good lather.

A lathe is a tool for turning wood or metal.
```

```
The Thespian Theatre is something I am loathe to attend.

The quick brown fox jumped over the lazy dog.
```

试验的目的是匹配位于词结尾处的词边界之前的字符序列 the 。

试一试：使用 \> 元字符

(1) 打开 OpenOffice.org Writer，然后打开测试文件 EndBoundary.txt。

(2) 按 Ctrl+F 快捷键打开 Find & Replace 对话框，并选中 Regular expressions 复选框，但不选中 Match case 复选框——因为这里要进行一次不区分大小写的搜索。

(3) 在 Search for 文本框中输入模式 the\>。

(4) 单击 Find All 按钮，并观察结果，其结果如图 6-17 所示。

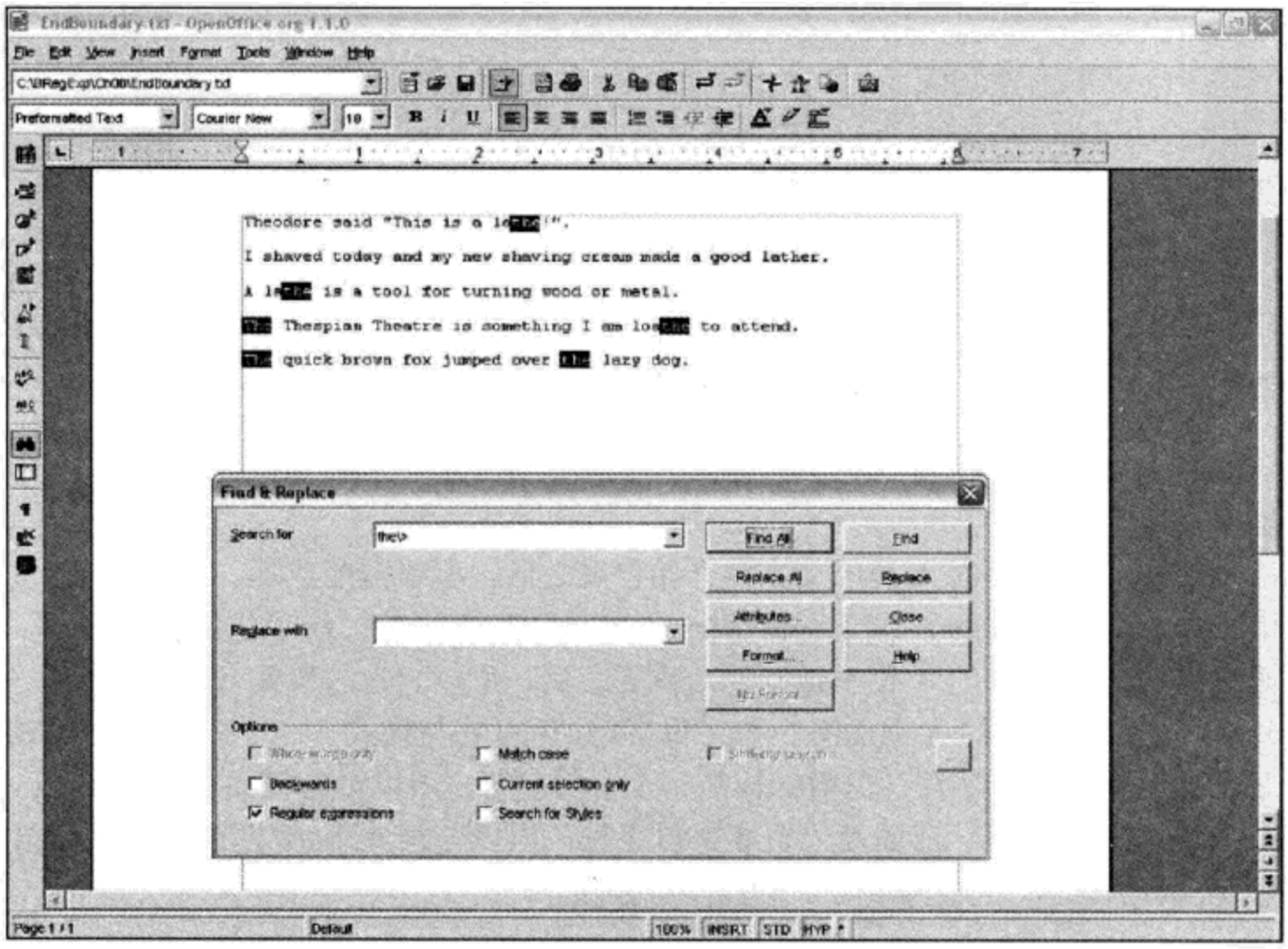


图 6-17

工作原理

第一行中匹配的字符序列 the 直接位于一个感叹号之前，因为感叹号不是一个字母字符，所以这个 the 字符序列位于一个词边界之前，它与模式 the \> 匹配。

在第二行中，lather 中的字符序列 the 的后面是一个字母字符，一个小写的 r。因为在这个 the 后面没有词边界，因此它不匹配。

测试文件中后续的匹配项都位于一个空格符(一个非字母字符)的前面，所以它们都位于一个词结尾处的词边界之前。因此，它们都与模式 the \> 匹配。

6.3.3　\b 语法

\b 元字符既可以用于匹配位于词开始处的词边界也可以用于匹配词结尾处的词边界，通过下面的试验可以验证这个元字符的作用。

试一试：使用 \b 元字符

(1) 打开 PowerGrep，并在 Search 文本框中输入模式\bA。
(2) 在 Folder 文本框中输入文本 C:\BRegExp\Ch06。
(3) 在 File mask 文本框中输入文件名 BoundaryTest.txt。
(4) 单击 Search 按钮，并在 Results 区域中观察结果，如图 6-18 所示。

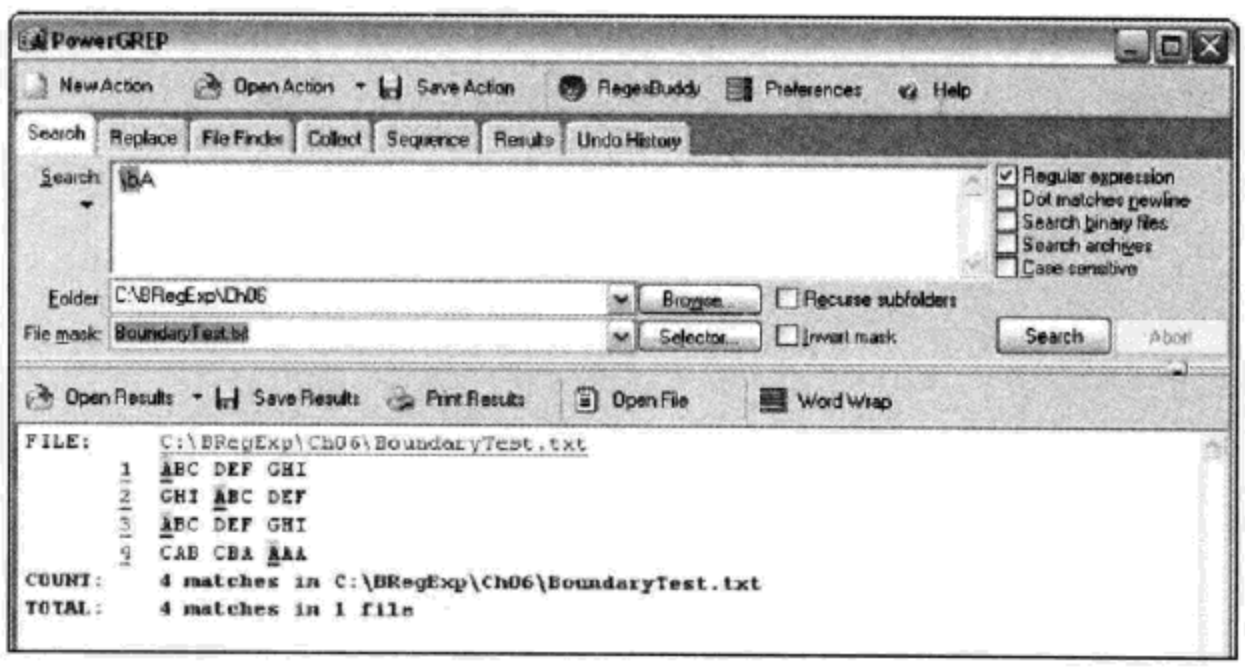


图　6-18

工作原理

模式 \bA 与 \<A 是等效的。它的含义是一个词边界后跟一个大写的 A。从理论上讲，\bA 与 \<A 的含义是不一样的，因为 \<A 的意思如下：

匹配词开始处的词边界，并且后跟一个大写的 A。

而 \bA 的意思则是：

匹配一个词开始处或结尾处的词边界，并且后跟一个大写的 A。

在实践中，后跟一个字母字符的词边界一定是一个位于词开始处的词边界。在本章后面的练习题中，有一道就是要求使用\b 元字符来匹配位于词结尾处的词边界。

\B 元字符

\B 元字符的含义与 \b 元字符相反。\B 元字符匹配一个非词边界的位置。

6.3.4 不常见的词边界元字符

在某些语言或工具中，还有一些其他的元字符用于标记位于词开始处或词结尾处的词边界。

6.4 练习

下面的练习题可以让你测试对本章内容的理解掌握情况。

1. \b 元字符可以匹配一个位于词开始处或结尾处的词边界。那么请使用 \b 元字符编写一个能与模式 the\> 匹配相同文本的模式。

2. 请编写一个模式，让这个模式匹配 lather 中的字符序列 the，但不匹配 then 或 lathe 中的字符序列 the。

第 7 章

正则表达式中的圆括号

在使用正则表达式时，圆括号是一种很有用的工具。可以用它们根据不同的目的对字符进行分组，本章会介绍圆括号在正则表达式中的所有用途。比如说，圆括号可以用来表达简单的二中择一或多种选项。

圆括号可以创建一或多个字符组。而匹配的字符组可以用在后来的文本操作中——例如，对指定的匹配文本块进行替换操作等。

在本章中将讨论如何实现以下内容：

- 使用圆括号来分组
- 对字符和元字符组使用限定符
- 匹配直接量开闭圆括号字符
- 在正则表达式中提供二中择一或多种选项
- 使用捕获(capturing)或非捕获(non-capturing)圆括号
- 使用反向引用

7.1 使用圆括号分组

正则表达式模式中圆括号的作用是对字符进行分组，并保存匹配的文本。通常，这些通过圆括号分组的字符都是为了文本操作的目的而建立的。本节介绍什么是组和如何进行分组。

要创建一个字符组，只需简单地在需要分组的字符前面加一个圆开括号字符，而后面加一个圆闭括号字符即可。例如，下面的模式：

```
United States
```

只包含直接量字符，这个模式可以用于匹配文本 United States。现在，再看下面的模式：

```
(United)( )(States)
```

它也会匹配相同的文本，不过，它同时还建立了三个组：第一个组中包含字符序列 United，第二个组中包含一个空格符，而第三个组中则包含字符序列 States。例如，假设

想用大写的 U 来替换组 United，用零长度字符替换包含空格符的组，用大写字母 S 来替换组 States，那么就可以使用稍微麻烦一点的方法来把字符串 United States 替换成简写的 US。

圆括号可以用于分组字符序列，而如何分组则取决于要完成的操作任务。

当在模式中使用圆括号时，要特别注意圆括号里不能包含任何空白符。因为如果包含了空白符，正则表达式引擎就会将这个空白符作为字符序列的一部分来处理。而如果它没有找到匹配的空白符，便会导致匹配失败。

试一试：对字符进行分组

本例示范简单的分组过程：

(1) 打开 Komodo Regular Expression Toolkit，并删除任何残留的正则表达式模式或测试文本。

(2) 在 Enter a string to match against 区域中输入测试文本 The hot water。

(3) 在 Enter a regular expression 区域中输入模式 hot。

(4) 观察匹配的文本(即字符串 hot)，并注意 Enter a string to match against 区域下方的灰色区域中的信息：Match succeeded: 0 groups。

(5) 把正则表达式模式修改为(hot)，然后再观察匹配的文本(仍然还是字符串 hot)，但此时 Enter a string to match against 区域下方的灰色区域中的信息已经变成了 Match succeeded: 1 group。

图 7-1 显示了第 5 步之后的结果。通过向简单的正则表达式模式中添加圆括号，我们创建了一个组——如图 7-1 所示。注意，在 Group Match Variables 区域中，变量 $1 出现在了 Variable 列中，而值 hot 出现在了 Value 列中。这表示组的匹配项可以通过变量 $1 引用，而它的值是字符序列 hot。有关引用组的主题将在本章后面再讨论。

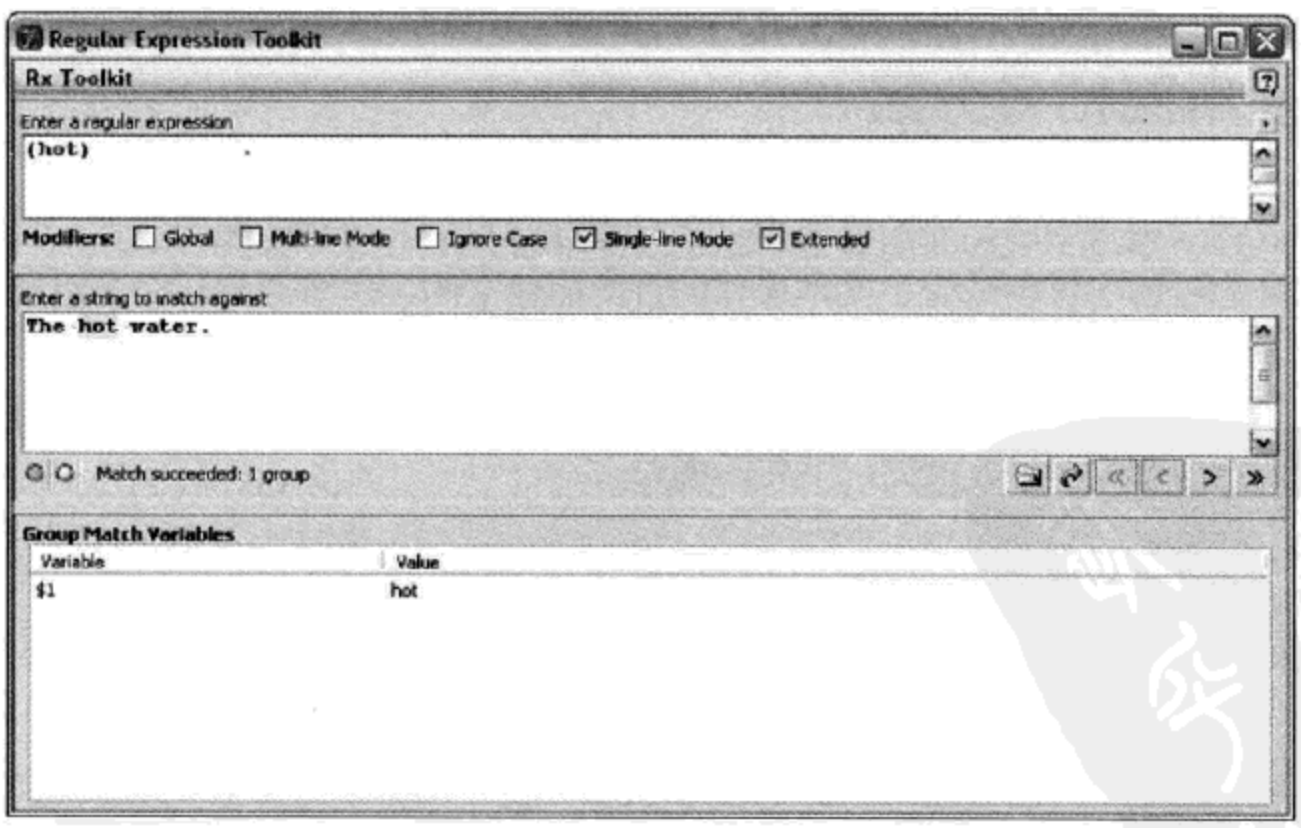


图 7-1

工作原理

第一个模式 hot 只包含三个直接量字符，因而会匹配简单的字符序列：hot。

当在模式中使用圆括号包围 hot 时，这三个直接量字符的匹配项便构成了一个组。

7.1.1　圆括号和限定符

圆括号的一个基本用法是对字符或元字符进行分组，这样在括号内可以对字符组合使用限定符。

例如，若要一组字符出现多次，可以把相关的字符或元字符包含在一对圆括号内，然后在这个组(圆闭括号)的后面再应用一个适当的限定符。

举例来说，假设要匹配一个由大写的 A 后跟一个数字，再后跟另外一个大写的 A 和一个数字组成的字符序列。那么，就可以使用下面的模式：

```
(A\d){2}
```

其中，模式的组件(A\d)匹配一个大写的 A 后跟一个数字。圆括号什么也不匹配——既不会匹配测试文本中的任何字符也不会匹配位置。限定符 {2} 表示只有被圆括号括起来的那些字符被匹配两次才能算匹配成功。

试一试：圆括号和限定符

这里所用的测试文件是 QuantifierTest.txt，其内容如下：

```
A3A4CDE

B9B6XYZ

A2A9RTE

B4B4UIO

G2H1WEQ
```

(1) 打开 PowerGrep，然后在 Search 文本框中输入正则表达式模式(A\d){2}。

(2) 在 Folder 文本框中输入文本 C:\BRegExp\Ch07。如果把下载的代码放在别的地方，需要修正这里的路径。

(3) 在 File mask 文本框中输入文件名 QuantifierTest.txt。

(4) 单击 Search 按钮并观察 Results 区域显示的结果，如图 7-2 所示。

工作原理

模式(A\d){2}对于能匹配的字符序列与模式 A\dA\d 具有相同的含义。换句话说，它匹配一个大写的 A，后跟一个数字，再后跟另一个大写的 A 和一个数字。

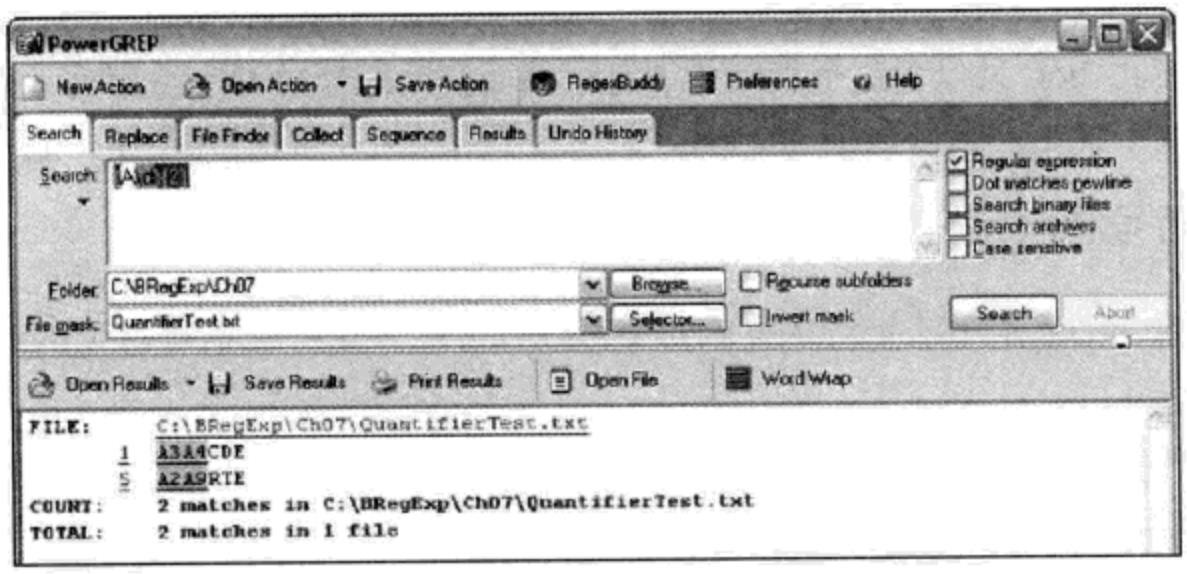


图 7-2

PowerGrep 显示在第一行和第五行存在匹配项。

当正则表达式模式引擎位于第一行开头的 A 之前的位置时，它首先尝试将该行中第一个字符 A 与圆括号中的第一个字符——大写的 A 进行匹配。匹配成功后，又尝试匹配圆括号中的第二个组件——元字符 \d 和数字 3。这也匹配。目前圆括号中的所有组件都已经匹配了一次。因为有一个{2}限定符，所以正则表达式模式引擎会尝试再次将圆括号中的第一个组件——字符 A 与第一行中的第三个字符 A 进行匹配。匹配成功。然后，它又尝试将圆括号中的第二个组件——元字符 \d 与第一行中的第四个字符——数字 4 进行匹配。匹配也成功。现在，圆括号中的各个组件都已经匹配了两次，所以匹配完成。而由于模式中的所有组件都匹配成功，那么整个模式匹配成功。结果是在 PowerGrep 中字符序列 A3A4 突出显示出来。

7.1.2 匹配圆括号直接量

由于圆开括号(和结束的圆闭括号)字符在正则表达式模式中具有特殊的功能，所以它们不能用于匹配自身直接量。而要想匹配圆开括号直接量，必须使用下面的模式：

```
\(
```

要匹配圆闭括号直接量，则需要使用下面模式：

```
\)
```

假设要匹配下面文本中的 (Home)：

```
Tel. 123 456 7890 (Home)
```

就应该使用下面的模式：

```
\(Home\)
```

7.1.3 美国电话号码的例子

在实践中，匹配圆括号直接量的用法经常出现在匹配构成美国电话号码的字符序列中。美国电话号码有几种格式。本例为了演示需要，假设数据源中应该包含下面的一种

格式：

```
(123) 123-4567
```

对于要匹配这种数据结果的问题定义，可以描述如下：

匹配一个圆开括号，后跟三个数字，后跟一个圆闭括号，后跟一个空格符，再后跟三个数字，后跟一个连字符，最后跟四个数字。

如果使用字符类来匹配数字，可以使用以下模式：

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

试一试：匹配美国电话号码的例子

这个例子使用前面的模式匹配特定格式的美国电话号码。

本例用到的测试文件是 PhoneNumbers.txt，其内容如下：

```
(987) 133-4477

(123) 876-3456

123-456-7890

(898 123-1234

879) 345-8765
```

只有前两个电话号码符合问题定义中的描述。

(1) 打开 PowerGrep，并在 Search 区域中输入正则表达式模式 \(\d{3}\) \d{3}-\d{4}。

(2) 在 Folder 文本框中输入文件夹名 C:\BRegExp\Ch07。

(3) 在 File mask 文本框中输入文件名 PhoneNumbers.txt。

(4) 单击 Search 按钮，并观察 Results 区域中显示出来的结果。

图 7-3 显示的是操作完成后的结果。

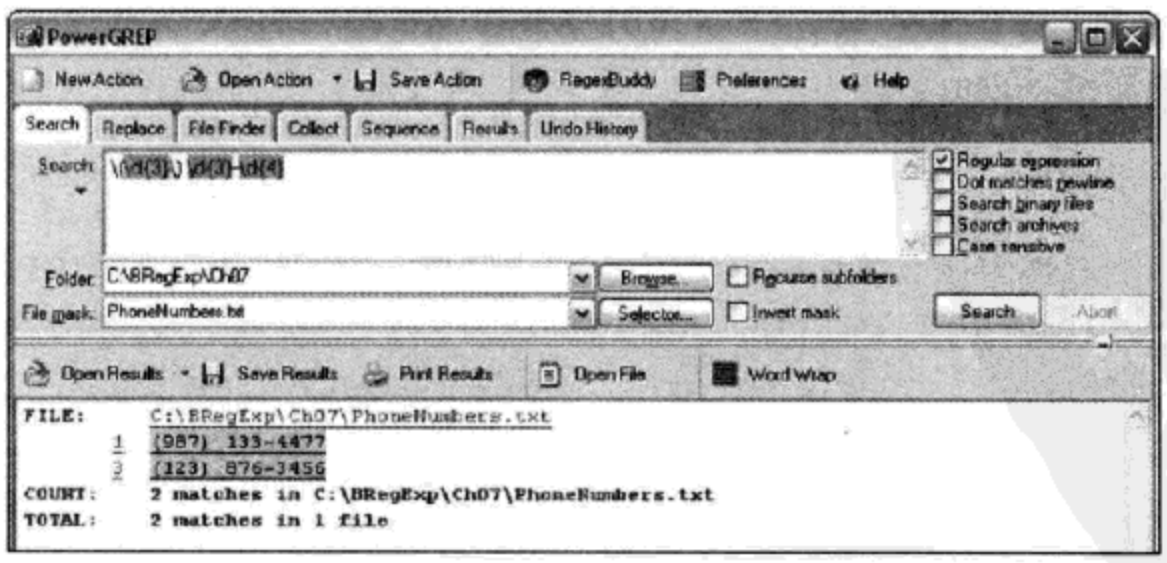


图　7-3

工作原理

首先，我们看一下第一行中的测试文本(987) 133-4477 是如何匹配的。假设正则表达式引擎位于圆开括号之前，它尝试将模式\ (与圆开括号字符——(进行匹配。匹配成功了。然后，模式 \d{3} 也匹配了，因为该字符串中的第二、第三和第四个字符都是数字。所以匹配成功。接着，直接量) 与模式 \) 匹配。然后，直接量空格又和测试字符串中的第六个字符——也是一个空格符尝试匹配。结果仍然匹配。接着，模式 \d{3} 与测试字符串中的第七、第八和第九个字符——133 尝试匹配。因为这三个字符都是数字，所以匹配成功。然后，是模式中的直接量连字符与测试字符串中的连字符进行匹配，当然也匹配成功。最后，模式 \d{4} 与测试字符串的最后四个字符——4477，尝试匹配。由于这四个字符都是数字，所以匹配成功。因为正则表达式模式的所有组件都匹配，所以整个正则表达式匹配。

测试字符串 123-456-7890 之所以没有匹配，是因为模式中的第一个组件 \(没有找到匹配项。

为什么测试字符串(898 123-1234 也没有匹配呢？我们假设正则表达式引擎从圆开括号之前的位置尝试匹配，首先模式 \(匹配成功。而模式 \d{3} 与字符序列 898 也匹配成功。然而，模式中的第三个组件\)却没有找到匹配项。因此，整个正则表达式的匹配失败。

7.2 交替选择

圆括号的另一种重要的应用是表示可选择性。要根据可以选择的情况建立支持二选一或多选一的应用，涉及到使用圆括号和 | 元字符(也有时候也称吧(bar)元字符)——后者用于表示逻辑或的意思。

严格来讲，选项不能超过两个。否则，如果是在三个或多个选项中选择一个，那些选项就不是交替的选项了。但是，术语“二选一”或者“交替”在正则表达式的领域中却可以完美地用于表示两个或多个选项。所以，在本节中，无论是涉及两个选项还是多个选项，我们都一律称为“交替选择”。

交替选择的最简单用法就是从两个直接量选项中选择一个。例如，当处理同时包含美式英语和英式英语的文档时，表示颜色的 gray(灰色)可能有两种拼写方法：在英式英语中是 gray，而在英式英语中是 grey。

相应的问题定义可以表达为：

匹配一个小写的 g，后跟一个 r，后跟一个 a 或者一个 e，最后跟一个 y。

读者可能会奇怪为什么要强调首字母 g 是小写的。原因是如果文档中包含姓氏 Grey 或 Gray，或者以这四个字母的任何一种组合开头的名字，我们都不希望替换其中的 e (比如 Mr. Grey)以保持一致的美式英语式拼写。因为我们不想修改任何人的姓。

我们可以用下面的模式来表达上面的问题定义：

```
(gray|grey)
```

或者

```
gr(a|e)y
```

这两个模式都具有同样的逻辑含义。实际上，在第二个模式中也可以使用等价的字符类。使用字符类来代替交替选择通常会提高匹配效率：

```
gr[ae]y
```

试一试：选择二选一的直接量

这个例子演示如何通过两个以直接量表达的交替选择完成匹配。假设想从零件编号列表中选择想要的零件编号，测试文档 PartNums.txt 中包含的零件编号如下：

```
ABC03
ABC08
ABC11
ABC13
ABC18
ABC25
ABC45
ABC12
ABC19
ABC88
ABC71
ABC04
ABC02
ABC55
```

从上面可以看到，零件编号并没有按顺序排列。如果要选择位于 ABC01~ABC19 之间的零件编号，其中一种方法就是使用下面包含圆括号的正则表达式模式：

```
ABC(0|1)[0-9]
```

(1) 打开 OpenOffice.org Writer，并打开测试文件 PartNums.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中输入模式 ABC(0|1)[0-9]。

(5) 单击 Find All 按钮，并观察结果。如图 7-4 所示，例子文档中的所有位于 ABC01~ABC19 之间的零件编号都突出显示了(数据列表呈现黑白交替的外观)。

零件编号中第一个数字不是 0 或 1 的没有匹配。

工作原理

对第一行 ABC03 的匹配，是这样完成的：假设正则表达式引擎从 ABC03 中的 A 之前的位置开始匹配，它首先尝试匹配模式中的第一个字符 A，而测试文本中的第一个字符是 A，所以匹配成功。然后，尝试匹配第二个字符 B，再然后是字符 C。接着，又查找与模式 (0|1) 匹配的字符。这时的测试对象是测试文本中的第四个字符——数字 0。全部匹配成功。最后，又将字符类 [0-9] 与数字 3 进行匹配。结果还是匹配成功。由于正则表达式中所有组件都成功匹配，所以整个正则表达式也就匹配成功了。

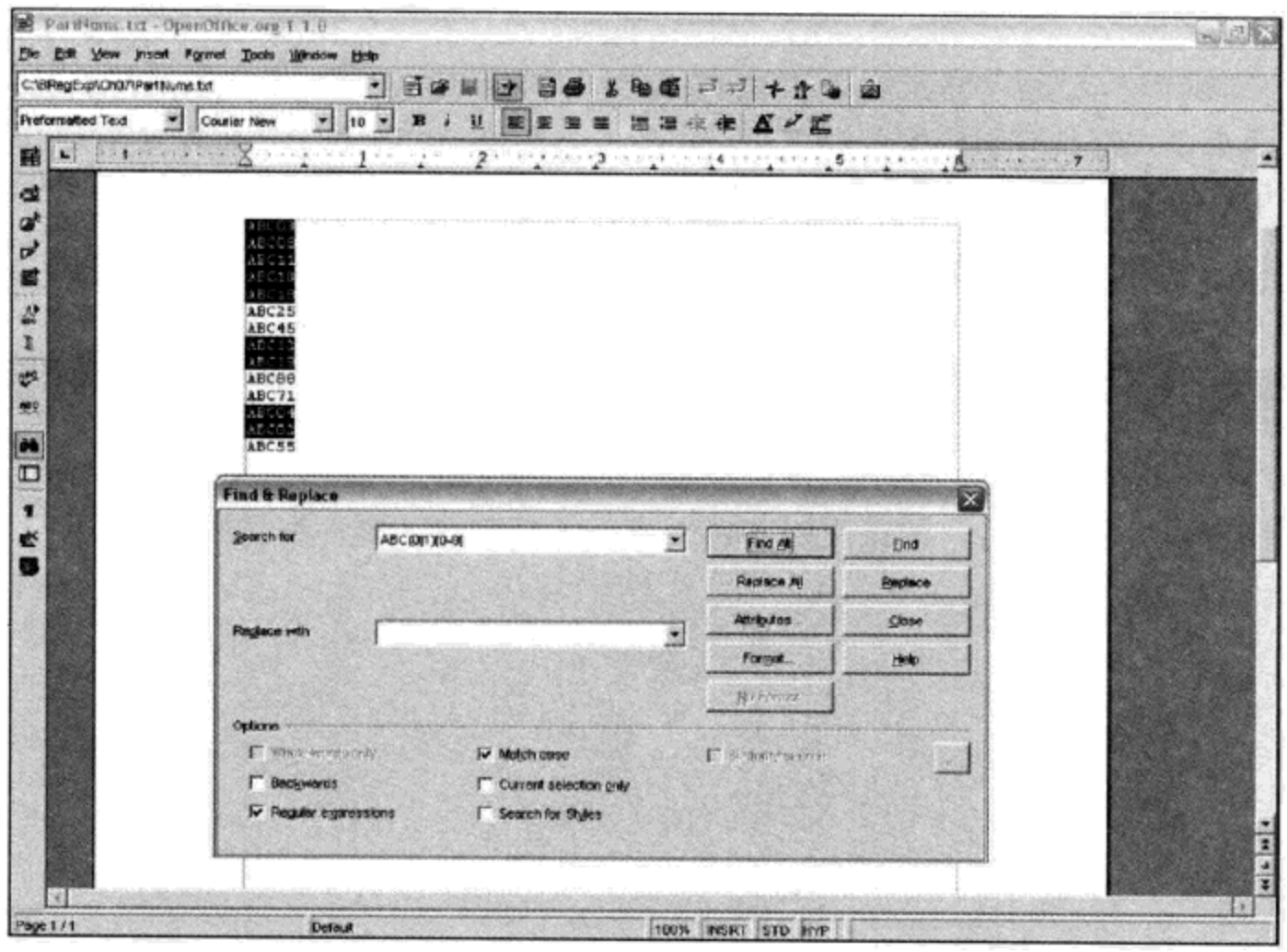


图 7-4

测试文本中的 ABC11 也是匹配的。其中，前三个字符会按照上一段中描述的过程成功匹配。当模式(0|1)与数字 1 进行匹配时，也匹配成功了。然后是字符类[0-9]与第五个字符——数字 1 进行匹配。结果也是匹配。

测试文本中的 ABC25 没有匹配成功，是因为模式(0|1)不能与数字 2 成功匹配。

7.2.1 在多个选项中做出选择

假设你有一些关于人的文本，其中包含在医院实习的人的相关信息。你想查找所有医生。

在这些数据中，可能会存在 Doctor 或 doctor，以及两种缩写方式——Dr.(带句点)和 Dr(不带句点)的写法。是否需要在搜索中包含 doctor 取决于你的需要。假设你只想查找首字母为大写 D 的医生。那么相应的问题定义可以描述如下：

匹配字符序列 D、o、c、t、o 和 r 或者匹配字符序列 D 和 r 再或者匹配字符序列 D、r 和 .(一个句点)。

下面的模式会满足问题定义的要求：

```
(Doctor|Dr|Dr\.)
```

记住，句点在正则表达式模式中是一个元字符，它可以匹配任何字母数字字符。所以要严格匹配测试文本中的直接量句点字符，必须在模式中使用转义序列 \.。另一个可选的

模式是：

```
(Doctor|Dr\.?)
```

试一试：匹配多个选项

这里使用的测试文件是 Doctors.txt，其内容如下：

```
Doctor

Drf

Dr

Dr.

Drs

Doctors
```

(1) 打开 OpenOffice.org Writer，并打开测试文件 Doctors.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case (保证不会匹配字符序列 doctor)复选框。

(4) 在 Search for 文本框中输入模式 (Doctor|Dr\.|Dr)。

(5) 单击 Find All 按钮，并观察结果，如图 7-5 所示。注意只有 Dr.中的 Dr 匹配。

(6) 在测试文件的结尾添加测试文本 Drive，并单击 Find All 按钮。

(7) 观察结果。注意 Drf 中的 Dr 被匹配。这说明我们前面的问题定义中存在问题，因为不想要的文本也被匹配了。经过修正后的问题定义如下：

匹配字符序列 D、o、c、t、o 和 r，或匹配字符序列 D 和 r，或匹配字符序列 D、r 和 .(句点)。在前面描述的每个选项之后，必须是一个词边界的位置。

不过，前面的问题定义中还是有一些不足，因为句点本身就是一个非字母字符。所以，更好的问题定义应该是像下面这样：

匹配字符序列 D、o、c、t、o 和 r 或匹配字符序列 D 和 r。在前面描述的选项后面必须是一个词边界的位置。

这个问题定义更加精确并且指定了一个结束处的词边界位置。因为它指定的位于词边界之前的选项都是字母字符(词)，所以这个位置只能是一个结束处的词边界。

(8) 把模式修改为 (Doctor|Dr)\>。

(9) 单击 Find All 按钮，并观察结果。注意 Drive 前面的 Dr 这次没有匹配。而且整个 Dr. 现在被匹配了。

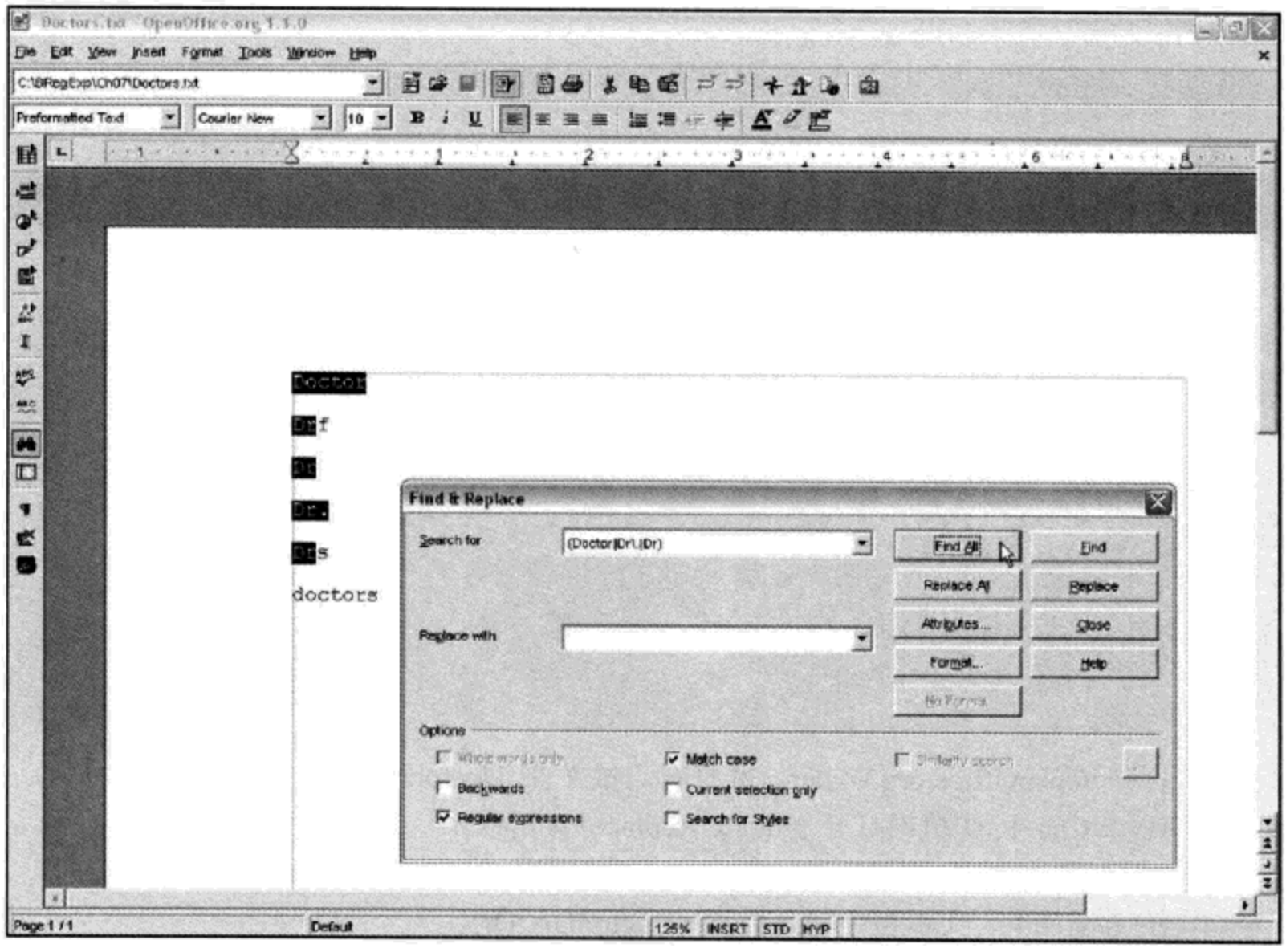


图 7-5

工作原理

首先，考虑在第 4 步之后发生的情况。

第一行中的文本 Doctor 匹配，这是因为模式圆括号中的第一个选项就是 Doctor。也就是说，模式中的每个直接量字符分别与测试文本中的每个直接量字符匹配。

第三行中的 Dr 匹配，是因为圆括号中的第三个选项 Dr 与测试文本中相应的字符匹配。

在第四行中，三个字符的序列——Dr.也同样匹配。

而 Drive 中的 Dr 匹配是因为它匹配了圆括号中的第三个选项。

现在，我们看一下在结束的圆括号之后添加了词边界元字符后的变化。

一开始，第二行中的 Drf 中的 Dr 是匹配的，现在却不匹配了。这是因为，虽然 Drf 中的 Dr 仍然与圆括号中的第三个选项匹配，而后跟的字符 f 前面的位置并不是一个结束处的词边界位置。

在第四行中，现在有两个字符序列匹配。圆括号中的第一个选项虽然没有匹配，但第二个选项却和第四行的第一和第二个字符匹配。此外，第四行中的直接量句点并不是一个字母字符，所以位于 Dr 中 r 后面的位置是一个结束处的词边界。此时的词边界存在，是因为 r 之后的句点，而不是因为句点之后的空格符。

7.2.2　错误匹配的交替行为

在使用交替选择时，有时候可能会出现并不想要的结果。这个问题在选项是不同长度的字符序列，而较短的字符序列位于左侧，同时较短的选项又被较长的选项所包含时尤为突出。这句话比较令人费解，所以我们还是先看一个例子。

假设匹配一个单独的小写字符 a 或者小写的字符序列 ab 中的任何一个。可以用下面的模式来表达这个想法：

```
(a|ab)
```

注意，其中较短的选项 a 位于左侧，而它同时也是较长选项 ab 的一部分。因而，同时满足了本节一开始提到的两个条件。

我们使用 ab.txt 作为测试文件，其内容如下所示：

```
a

ab

ac ab

ba

bab
```

在第三行中，存在字符序列 ab。

试一试：不平衡的交替选择

首先，用下面的模式来试一试：

```
(a|ab)
```

(1) 在 OpenOffice.org Writer 中打开文件 ab.txt。

(2) 按 Ctrl+F 快捷键打开 Find & Replace 对话框，并选中 Regular expressions 和 Match case 复选框。

(3) 在 Search for 文本框中输入正则表达式模式(a|ab)。

(4) 单击 Find All 按钮，并观察结果，结果如图 7-6 所示。
注意其中每个突出显示的匹配项都是一个字符，并且匹配小写的字符 a。

(5) 现在将模式中的两个选项调换位置：

```
(ab|a)
```

将 Search for 文本框中的模式修改为(ab|a)。

(6) 单击 Find All 按钮，并观察如图 7-7 所示的结果。现在有 6 个匹配项，其中与小写的字符序列 ab 匹配的三个字符序列被突出显示。

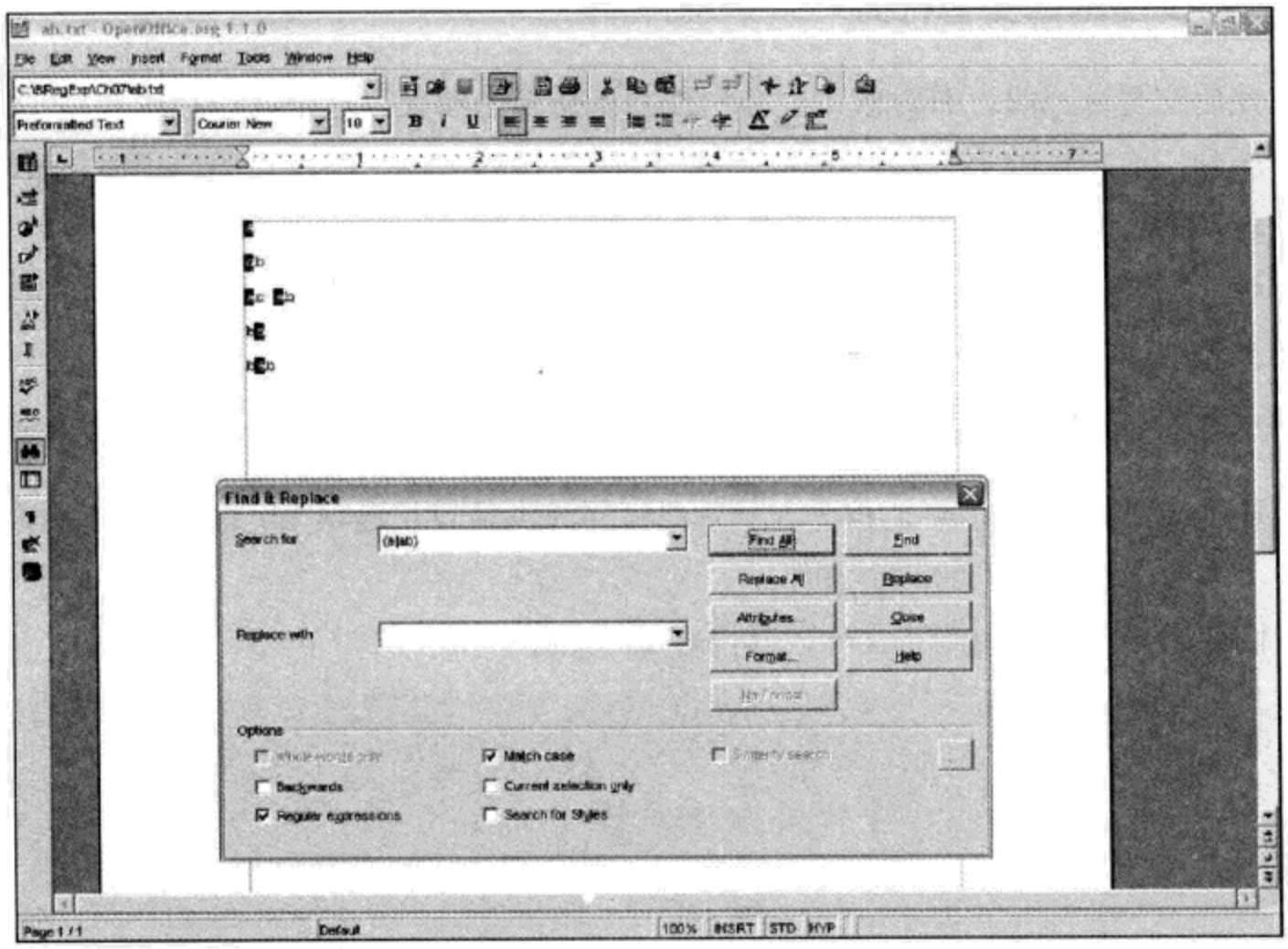


图 7-6

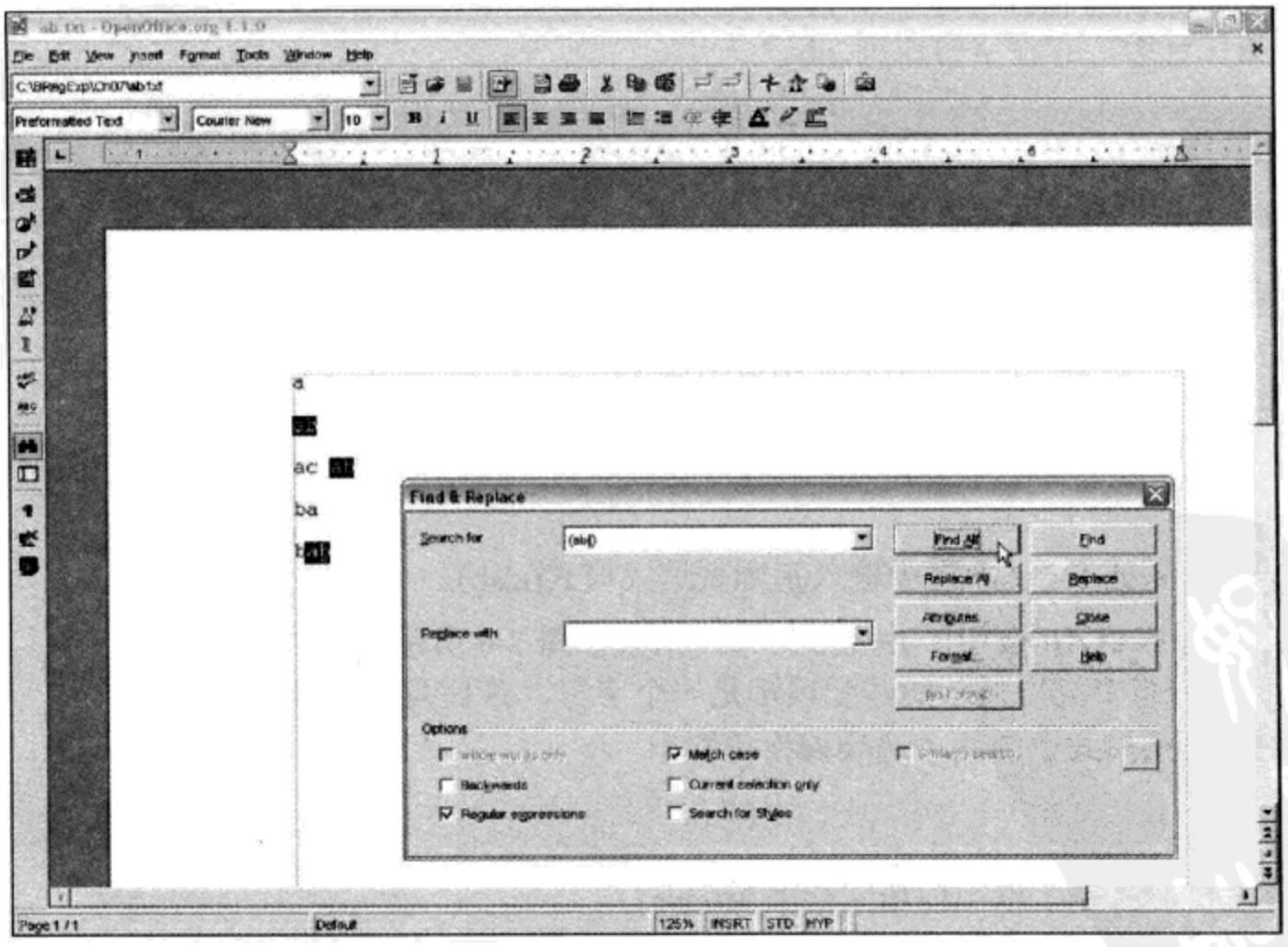


图 7-7

工作原理

在第二行，使用(a|ab) 时有一个匹配项，但只匹配了一个字符。假设正则表达式引擎在 a 之前的位置开始匹配，它首先尝试将第一个选项与该行中的第一个字符——也是一个 a——进行匹配。匹配成功。这样，正则表达式引擎就不会再尝试匹配第二个选项了。于是，它把当前位置移动到了刚找到的匹配项之后(位于 a 和 b 之间)的位置上。然后再重新尝试匹配，但此时不论是用 a 还是 ab 来匹配字符 b，结果都不会成功。

也就是说，使用模式 (a|ab) 不会匹配字符序列 ab。因为模式的前一个选项总是先匹配 a，而第二个选项永远不会有机会参与匹配。

但是，当把模式修改为 (ab|a) 时，在第二行中找到了两个字符的匹配。因为第一个选项 ab 会首先参与匹配。假设匹配从 a 之前的位置开始，那么第一个选项会匹配，因为模式 ab 与字符序列 ab 是匹配的。由于第一个选项匹配，所以第二个选项不再参与匹配。

7.3　捕获圆括号

在之前使用圆括号的过程中，与位于圆开括号和圆闭括号之间的模式匹配的内容都会被捕获。我们在图 7-1 中曾经看到过，圆括号定义了一个组，这个组模式匹配的内容 hot 被捕获并指定给了变量$1。

7.3.1　捕获组的编号

捕获组的编号由圆开括号在正则表达式模式中的位置决定。

例如，在使用下面的模式：

```
(United) (States)
```

匹配下面的文本时：

```
The United States
```

字符序列 United(它位于第一个圆开括号之后)就是变量$1 的值，而字符序列 States(它位于第二个圆开括号之后)是变量$2 的值。图 7-8 显示了在 Komodo regular expression toolkit 中完成这次匹配的结果。

假设修改了这个模式，在其中添加了一个包含空格符的圆括号：

```
(United)( )(States)
```

那么，模式中会有三个组，这时的结果如图 7-9 所示。

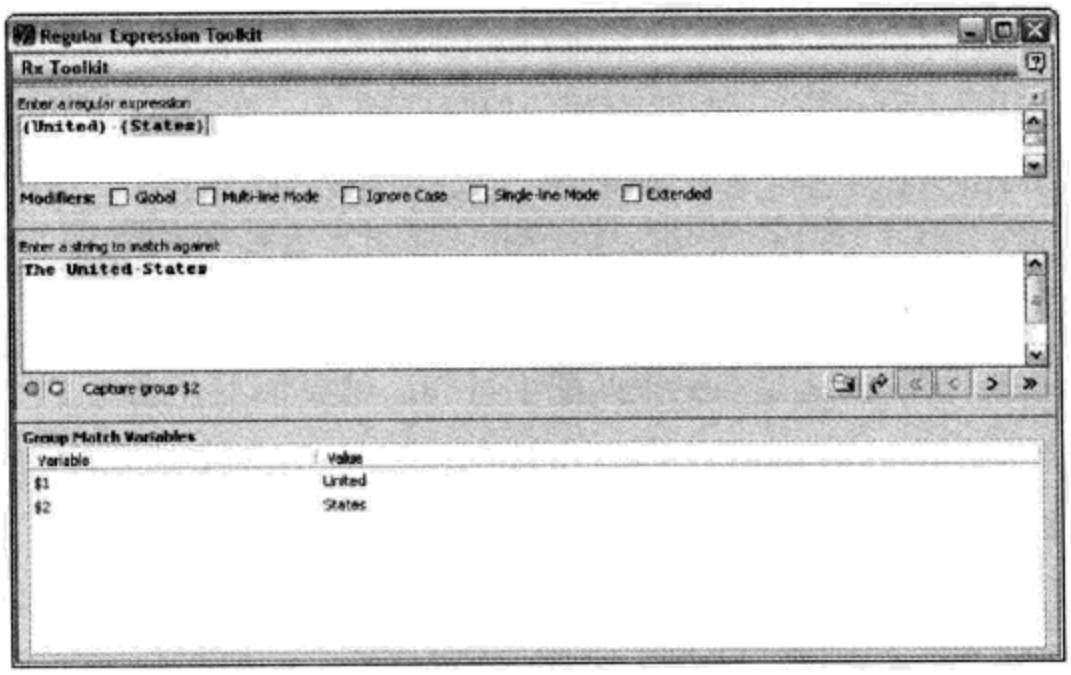


图 7-8

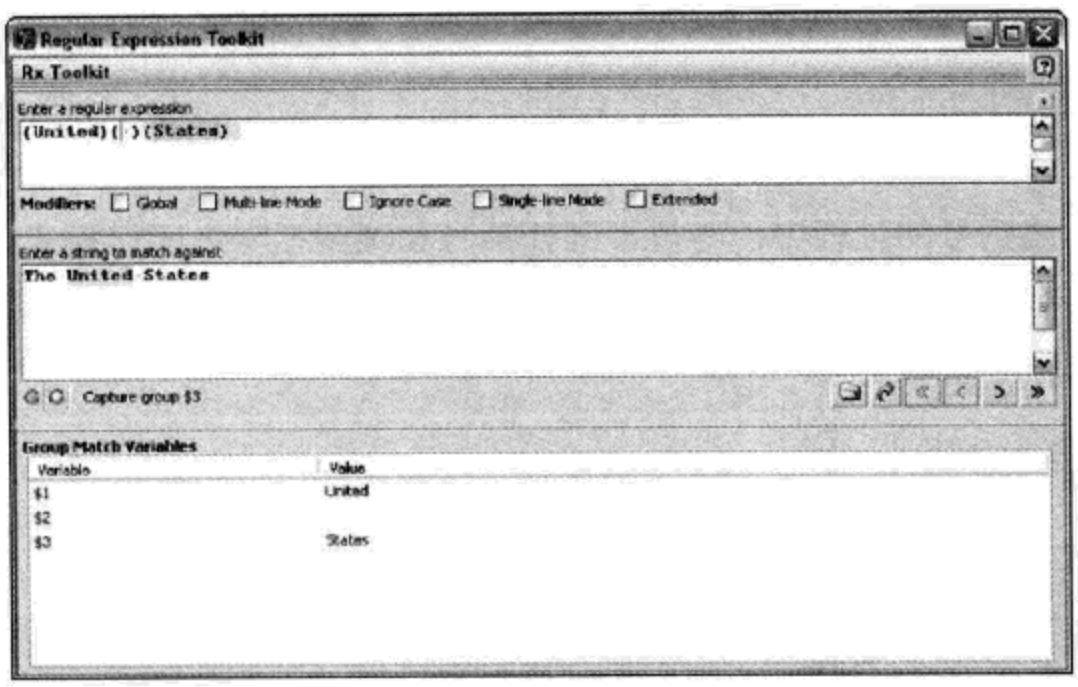


图 7-9

捕获一个空白符并不是不能实现的，只需把组合两个名字的字符序列：

```
([A-Za-z])( )([A-Za-z])
```

作为一个文件名：

```
FirstWord_SecondWord.html
```

并明确地用一个下划线字符替换其中的空格符就可以了。

7.3.2 使用嵌套的圆括号时的编号

在使用嵌套的圆括号时，同样的规则也适用。即变量的编号会按照模式中圆开括号出现的位置依次进行。

以下面文本为例：

```
A22 33
```

如果想用下面这个相当难懂的模式来匹配以上文本：

```
((\w(\d{2}))(( )(\d{2})))
```

那么你要问的第一个问题可能是为什么要这样来匹配。而且，你可能还会想到使用第 10 章中介绍并在本书后面针对具体语言的一些章节中进一步阐明的注释技术。

若想了解嵌套的圆括号为何有用，请参见图 7-10。

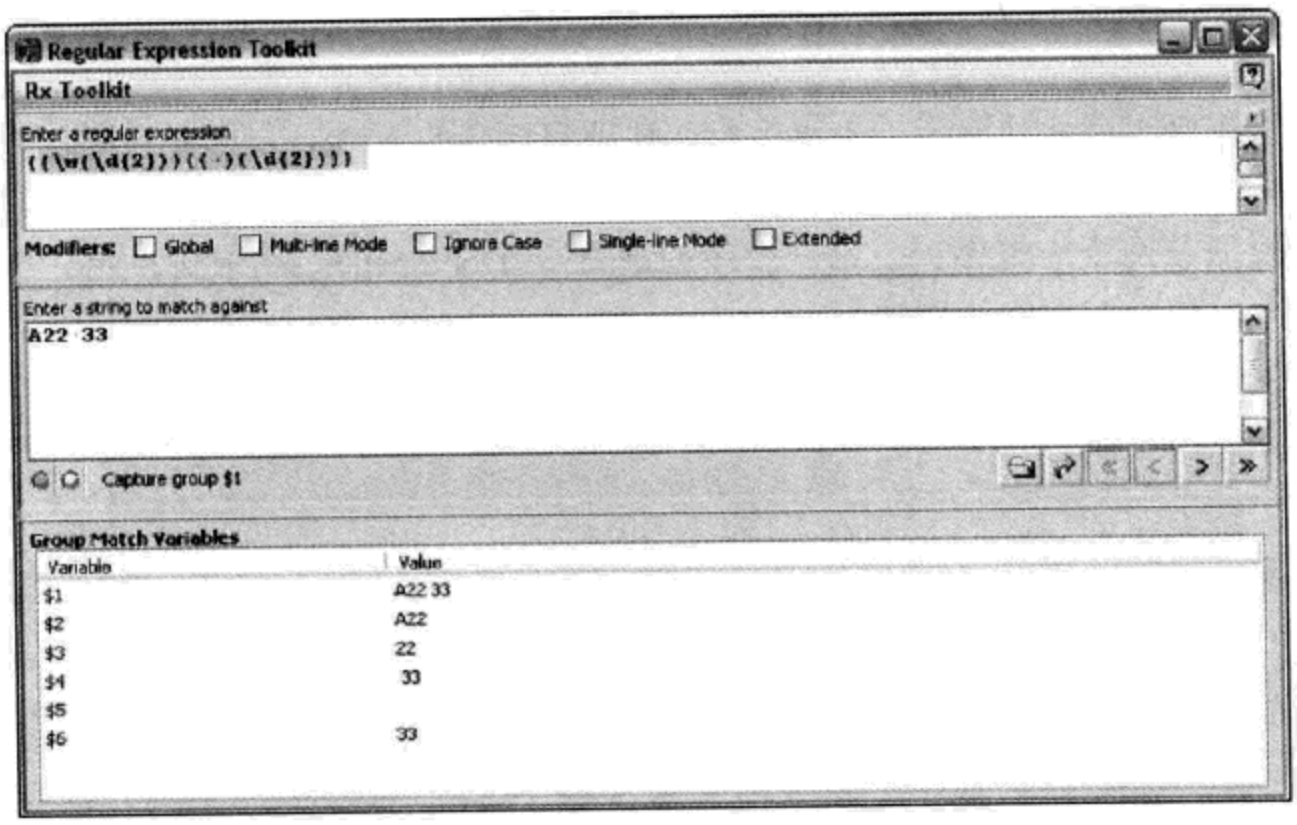


图　7-10

如果出于不同的编程目的而需要使用测试文本 A22 33 中的不同部分，现在就可以付诸行动了。变量 $1 中保存着值 A22 33。所以可以将这全部六个字符(包括空格符)用于试验。然而，如果要将其中的三个字符 A22 重用于其他方面，那么现在的变量 $2 可以马上派上用场。

所创建的其他变量中则保存着与测试文本各个部分对应的值。

7.3.3　命名的组

在 Python 和 .NET 语言中可以创建命名的组。.NET 中相应的语法是：

```
(?<组名>模式)
```

或

```
(?'组名'模式)
```

而 Python 中相应的语法是：

```
(?P<组名>模式)
```

因此，如果有以下形式的数据：

```
Temperature:22.2
```

并希望将捕获的温度保存在命名变量 MeanTemp 中，那么可以使用下面的模式——此处假设温度的范围为 10° ~99° ，并且保留一位小数。

```
Temperature:(?<MeanTemp>\d{2}\.\d)
```

7.4 非捕获的圆括号

在某些正则表达式实现中，也提供对非捕获圆括号的支持。

对非捕获圆括号的另一种称呼是仅分组的圆括号(或非捕获组。译者注)。

假设要捕获两个字符序列，第一个序列是表示 Doctor 的不同形式，第二个序列是这个医生的姓。假设数据的结构如下：

```
Doctor Firstname LastName
Dr FirstName LastName
Dr. FirstName LastName
```

可以通过下面的模式来捕获 Doctor 或者它的一种缩写形式，并将相应的拼写形式保存在$1 中，而把他的姓保存在 $2 中，如图 7-11 中所示：

```
(Doctor|Dr.|Dr)(\s\w{1,}\s)(\w{1,})
```

模式 (Doctor|Dr.|Dr) 创建了组 $1 并捕获到 Doctor 或它的一种缩写形式。而模式 (\s\w{1,}\s) 则创建了组 $2 并捕获到一个空格符、医生的名字和另一个空格符。模式 (\w{1,}) 则创建了组 $3 并捕获到了医生的姓。

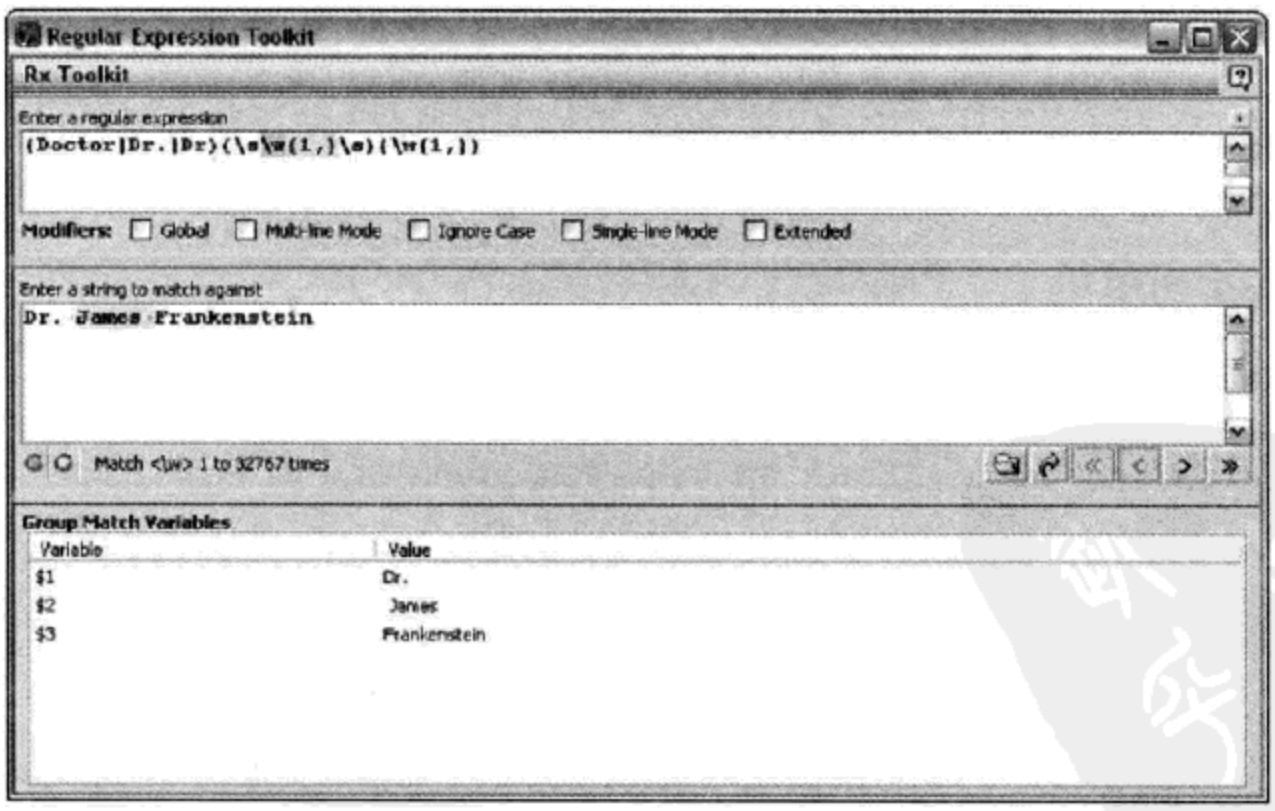


图 7-11

要创建非捕获圆括号的模式如下：

```
(?:the-non-captured-content)
```

换句话说，当在圆开括号后面放置一个问号和一个冒号时，相应的这对圆括号就不会再捕获内容了。

如果把前面例子中的模式修改为：

```
(Doctor|Dr.|Dr)(?:\s\w{1,}\s)(\w{1,})
```

那么则表明$1 中保存的是 doctor 的某种拼写形式，而 $2 中则保存的是姓，如图 7-12 所示。因为模式 (?:\s\w{1,}\s) 不会捕获包含名字的组内容。

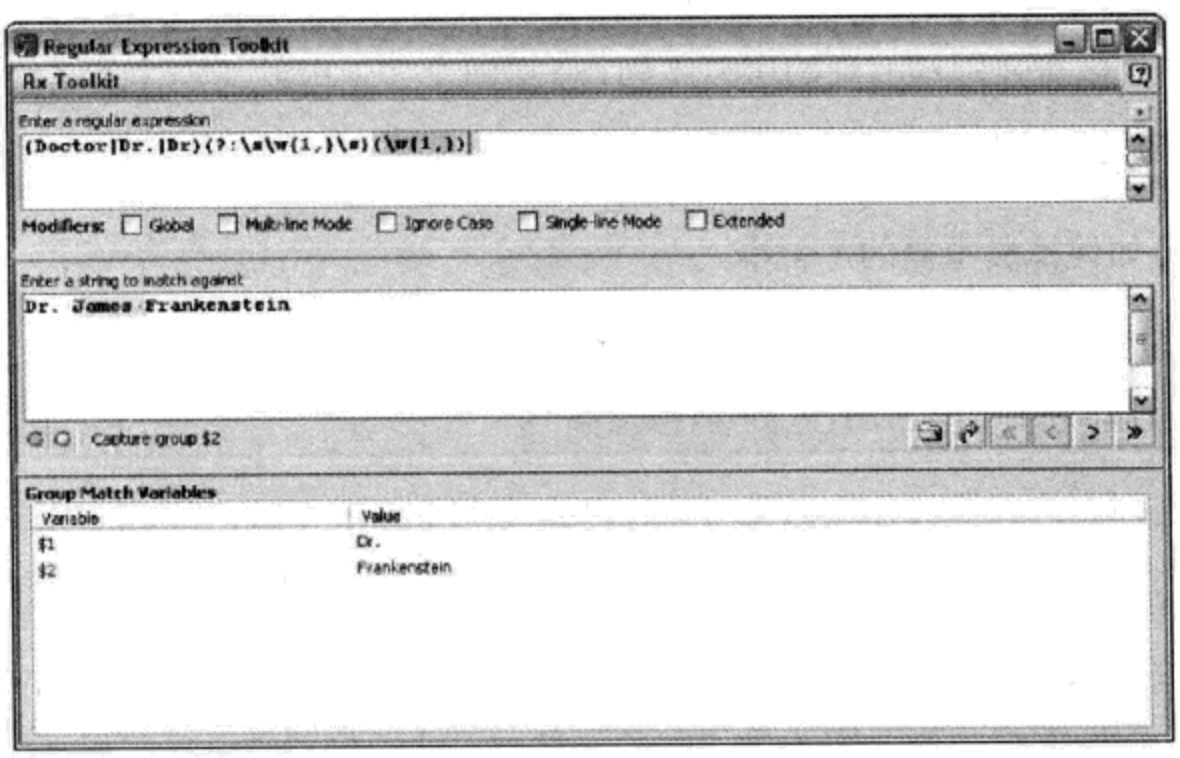


图　7-12

非捕获组确实让这个模式看起来有点复杂。但是，在复杂的同时，它却能够减少要处理的组数，因而使编程更容易一些。

另外，如果正则表达式引擎需要跟踪的组少一些，也能够获得效率的提升。

7.5　反向引用

有关捕获圆括号的常见用法是将其用于反向引用。在英文中，back references(反向引用)有时候也写成一个单词——backreferences。

能够体现使用反向引用方便性的一种情况是要找到某个句子当中无意间添加双定冠词——正如笔者在下面例句中故意添加的那样。而对于重要的文档，一个主要任务就是要挑出其中重复的单词。

我们这里使用的测试文档是 DoubledWords.txt，其内容为：

```
Paris in the the spring.
```

```
The theoretical viewpoint is of little value here.

I view the theoretical viewpoint as being of little value here.

I think that that is often overdone.

This sentence contains contains a doubled word or two two.

Fear fear is a fearful thing.

Writing successful programs requires that the the programmer fully understands the
problem to be solved.
```

在这个测试文档中，包含着一些采取相同大小写形式或者不同大小写形式的重复单词，以及一些虽然重复但却合理的字符序列。在第四个句子中的双单词是完全可以接受的。

试一试：检测重复的单词

本例示范如何检测重复的字符串。

在 OpenOffice.org Writer 中，表示匹配第一个组的变量被指定为 \1，而不像在 Komodo Regular Expression Toolkit 中那样是 $1。

下面的正则表达式模式使用圆括号来捕获由字符类定义的字母字符序列：

```
([A-Za-z]+) +\1
```

在与字符类匹配的一个或多个字符后面，是一个带有限定符 + 的空格符，用于表示圆括号中模式的匹配项与后面的字符之间必须有一个或多个空格。这个模式的最后一个组件是 \1，它是一个反向引用第一对捕获圆括号(这里唯一的一对圆括号)中内容的模式。

(1) 在 OpenOffice.org Writer 中打开 DoubledWords.txt。

(2) 打开 Find & Replace 对话框，并选中 Regular expressions 和 Match case 复选框。

(3) 在 Search for 文本框中输入模式 ([A-Za-z]+) +\1。

(4) 单击 Find All 按钮，并观察结果。图 7-13 显示了在 OpenOffice.org Writer 中将这个模式应用到 DoubledWords.txt 之后的结果。

从这幅插图中，我们可以看到在 DoubledWords.txt 中存在一些不想要的匹配项，这主要是因为使用的正则表达式模式选择的是重复的字符序列而非重复的单词。

因此，我们需要对模式进行修改以明确第一个单词的开始位置和可能重复的单词的结束位置。修改后的模式如下：

```
\<([A-Za-z]+) +\1\>
```

(5) 将 Search for 文本框中的模式修改为 \<([A-Za-z]+) +\1\>。

(6) 单击 Find All 按钮，并观察结果。结果如图 7-14 所示，现在的匹配项只集中于重复的单词，而忽略了非单词的重复序列。

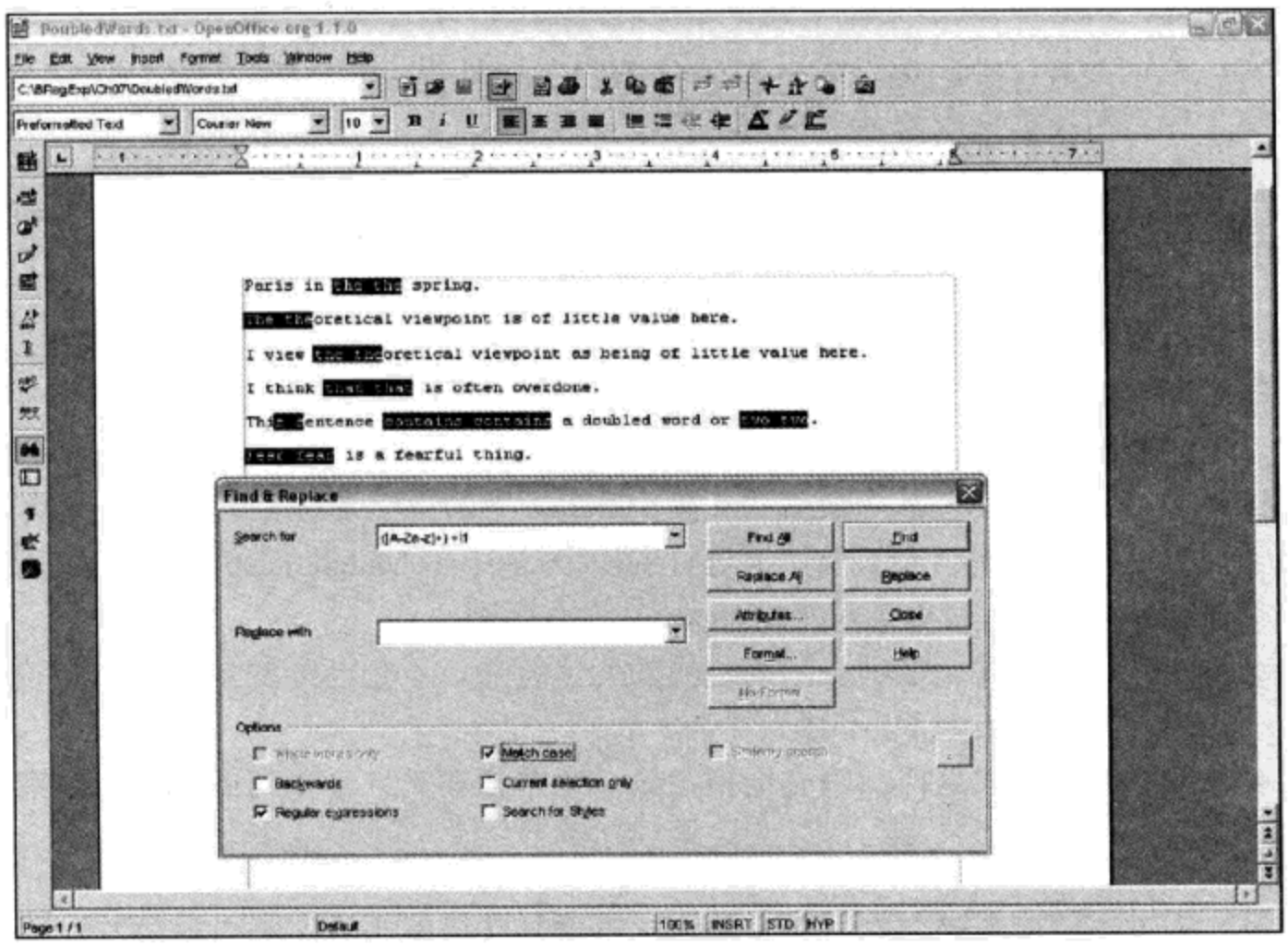


图　7-13

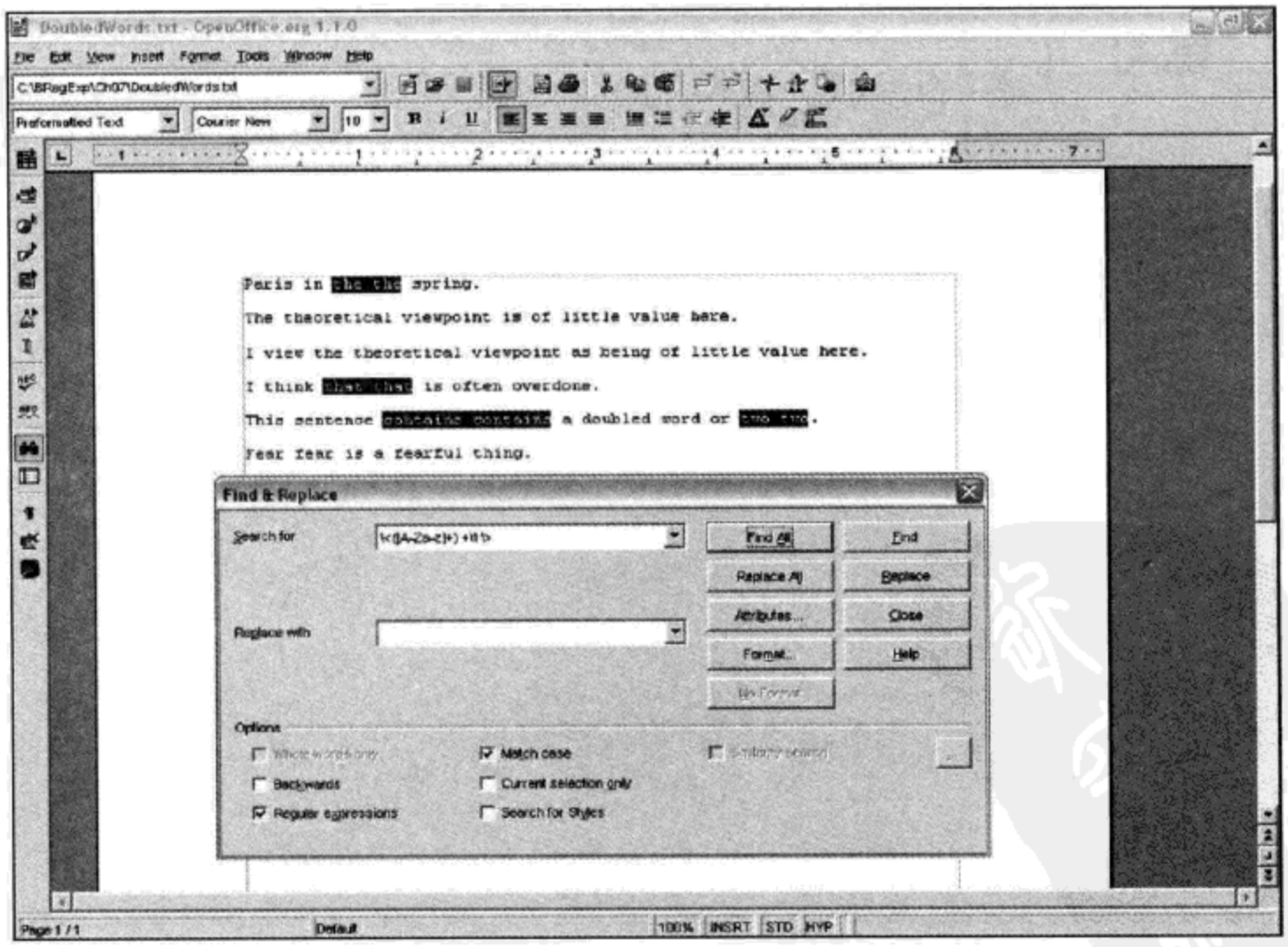


图　7-14

工作原理

首先，我们来看一下在使用模式 ([A-Za-z]+) +\1 时的匹配过程。

与这个模式对应的问题定义可以表达为：

匹配一个或多个 ASCII 字母字符，后跟一个或多个空格。然后，尝试匹配前面匹配的 ASCII 字母字符序列。

其中并没有指定所匹配的字符序列一定是一个单词。所以，我们会发现在第三行中匹配的字符序列 the，前面的是一个单词，而后面的则是单词 theoretical 的前三个字符。虽然匹配，但并非我们想要的。

同样，在第五行中，This 中最后的 s 在 sentence 中的首字母位置同样也是一次重复。

由于搜索是在区分大小写的条件下进行的(因为选中了 Match case 复选框)，所以第六行中的单词 Fear 和以小写形式重复的 fear 并没有匹配。

当对模式进行修改并添加了表示词边界的元字符后，不想匹配的单词中的字符序列，如 theoretical 中的 the，则不再匹配。

也可以像下面的模式这样，明确地指定第一个单词和第二个单词的开始与结束位置：

```
\<([A-Za-z]+)\> +\<\1\>
```

然而，不管怎样都要保证在前面匹配的字符序列之后至少有一个空格符存在。使用最后两个模式都会得到相同的匹配结果。

7.6 练习

通过下面的练习题可以测试对本章介绍内容的理解情况：

1. 请指出能够匹配字符序列 license 和 licence 的三种模式。

2. 请找出一种能够识别出重复的单词 fear 的解决方案。不用考虑 fear 是否位于一个句子的开头，并假设在重复的两个单词之间只有一个空格字符。

第 8 章

向前查找和向后查找

第 7 章中简单介绍了反向引用的概念。在该章查找重复单词的例子中，展示了对文本相关部分进行等同测试或检查的一种特殊形式。通过反向引用可以测试一个字符序列在测试文本中是否曾经出现过，并且可以将上次出现的字符序列用于某些特殊的目的。这对于限定使用的范围非常有帮助，比如查找单词。而查知前面或后面文本更常见的用途是可以让开发人员们表达诸如“如果前面是一个特殊的字符序列则匹配某个单词”，或者“如果某个字符序列后跟另一个字符序列，则匹配这个字符序列”这样的意图。

如果能够根据某个字符序列的后面或前面是什么来决定是否匹配这个字符序列，就可以排除许多不想要的匹配。这对于处理大量数据或者在搜索替换操作中降低错误匹配的风险具有重要意义。

通过模式来实现问题定义的匹配，使其匹配方式取决于要查找的单词或字符序列所处的上下文环境。

在某些情况下，会用术语双向查找(lookaround)来同时指代向前查找和向后查找。

向前查找和向后查找各自又分为肯定式和否定式两种类型。所以，双向查找包含肯定式向前查找、否定式向前查找、肯定式向后查找和否定式向后查找。

在本章将学习以下内容：

- 在什么情形下适合使用向前查找和向后查找
- 如何使用肯定式和否定式的向前查找
- 如何使用肯定式和否定式的向后查找

8.1 为什么需要向前查找和向后查找

在第 1 章中虚构的 Star Training Company 示例中，初次修改文档时使用了一种相当笨拙的方法。那次对文本替换的方法中存在的问题就是无法表达“只有当单词 Star 后跟单词 Traning 时才匹配 Star”这样的意图。这种思路就是向前查找。

为了使对 Star Training Company 文档替换的方法更加具体有效，可以对问题给予如下定义：

匹配字符序列 S、t、a 和 r，但该字符序列必须后跟一个空格符和另一个字符序列 T、r、a、i、n、i、n 和 g。

换句话说，只有当单词 Star 后跟一个带有空格符的单词 Training 时才可以匹配 Star。这种匹配方式比在前几章中使用的技术更加具体。

下面的模式实现以上问题定义：

```
Star(?= Training)
```

严格来讲，在使用上面的模式时，所匹配的是 Star 后跟一个空格符和一个字符序列 Training。如果想保证在向前查找时只匹配单词 Training ，可以根据所用工具所支持的元字符，在模式中加上表示词边界的 \b 或表示词结束位置的 \> 元字符。

```
Star(?= Training\b)
```

(?=之后和)之前的字符不会被捕获。

(?元字符

有一些元字符可以被看做是特殊的开始圆括号，而这些元字符都是以字符序列 (? 开头的。由于很多开发人员认为很难区分这些元字符，所以在这里简要地介绍一些这类元字符，以便帮助理解各种组合的作用。

在第 7 章中，介绍过(?: 形式的元字符，它表示非捕获组。本章还会介绍应用于向前查找和向后查找的其他特殊的开始圆括号元字符。同时还会给出如何使用这些元字符的一些例子。

(?= 组合形式用于肯定式向前查找中。为了方便比较，我们将把肯定式向前查找、否定式向前查找、肯定式向后查找和否定式向后查找的语法总结在表 8-1 中。

表 8-1 (?元字符及其含义

元 字 符	含 义
(?: ...)	非捕获组
(?= ...)	肯定式向前查找
(?! ...)	否定式向前查找
(?<= ...)	肯定式向后查找
(?<! ...)	否定式向后查找

8.2 向前查找

向前查找根据要匹配的字符序列后面存在一个特定的字符序列(肯定式向前查找)或者不存在一个特定的字符序列(否定式向前查找)来决定是否匹配(实际上，向前查找指的是一个子模式，这个子模式匹配特定的字符序列，但不返回匹配结果。所以从本质上来说，向前查找子模式匹配的是测试文本中的位置。而所谓向前，也正是基于这个位置而言的。向

前查找也就是向左、向文本中先出现的字符序列中查找要匹配并返回结果的字符序列。后面讲到的向后查找的意思类同。译者注)。

> 有些文档(比如 .NET 文档)把向前查找称为零宽度向前查找断言(所谓断言，即判别标准，可以理解为向前查找或向后查找中的子模式。译者注)。本章一般使用简称——向前查找，含义相同。而相关的术语肯定式和否定式将被用于术语向前查找的限定词。(向前查找和向后查找实际上是匹配但不返回结果的子模式，因而其结果总是零字符长，或者说是零宽度。所以在有的文档中将向前和向后查找也称为零宽度查找。译者注)

理解向前查找的关键在于，要认识到出现在指定匹配项之后的字符序列不会被正则表达式引擎返回这一事实。

这可能有些抽象，所以我们通过下面这个简单的例子来说明正则表达式引擎的行为。

假设有一个包含零件编号的文档，零件编号的格式是字母字符、字母字符、数字和数字，比如：

```
BC99
```

但是，我们只关心其中的字母字符。那么，相应的问题定义可能是：

匹配两个连续的字母字符，这两个字母字符位于一行的开始并且其后跟两个数字。

最终匹配的只有两个字母字符。所以实现这个问题定义的模式不匹配任何数字。下面是一个实现了该问题定义的模式：

```
^[A-Za-z]{2}(?=\d\d)
```

其中，^ 元字符表示一行开始的位置。而 [A-Za-z]{2} 表示两个 ASCII 字母字符。最后的(?=\d\d)是向前查找，这表示在这两个字母字符后面必须是两个数字。

这里使用测试文件 PartNumbers.txt，其内容如下：

```
AB21

AB1

CD8D3

RD/25
```

这四个零件编号中只有一个匹配。

试一试：零件编号的例子

(1) 打开 PowerGrep，并在 Search 文本区域中输入模式 ^[A-Za-z]{2}(?=\d\d)。

(2) 在 Folder 文本框中，输入文件夹名 C:\BRegExp\Ch08。如果下载时把示例代码存放到其他位置，则修改这里的路径。

(3) 在 File mask 文本框中输入 PartNumbers.txt，然后单击 Search 按钮。

(4) 在 Results 区域中观察结果，其结果如图 8-1 所示。注意，在这四个零件编号中只有一个与模式匹配。

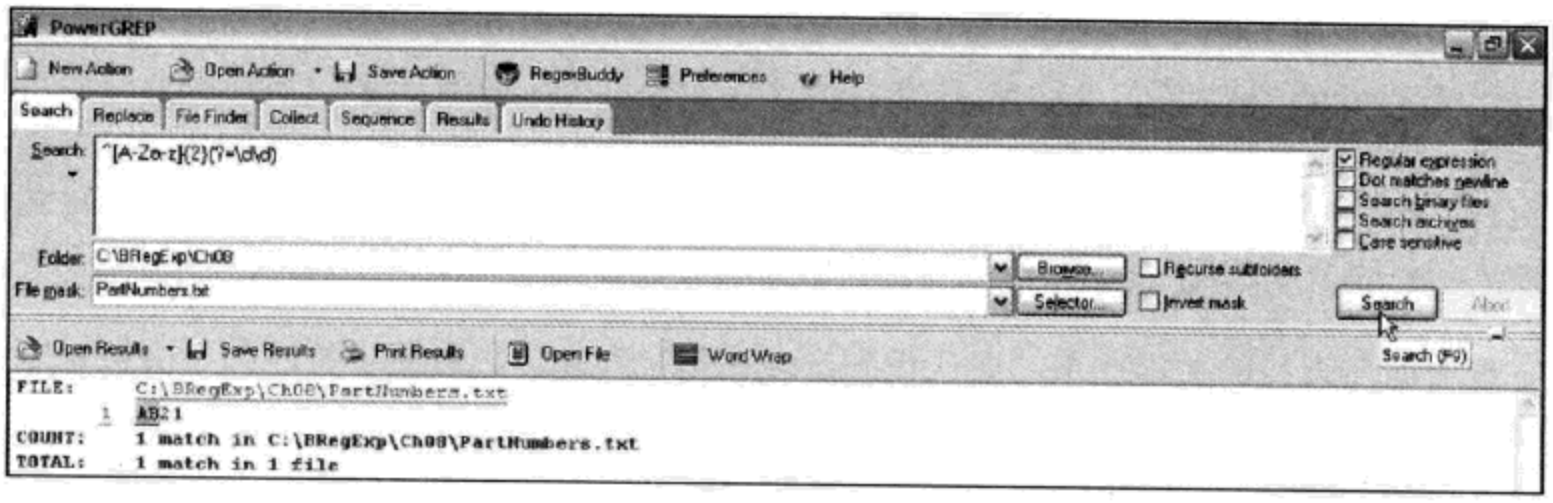


图 8-1

工作原理

假设正则表达式引擎从第一行中的 A 之前的位置开始匹配，它首先尝试将当前位置与 ^ 元字符进行匹配。匹配成功。接着，它尝试将模式 [A-Za-z] 与该行的第一个字符进行匹配。大写的 A 与这个字符类匹配成功。然后，它尝试再次将这个字符类与第二个字符——B 进行匹配。匹配也成功。在到此为止的匹配过程中，正则表达式引擎都是以我们所知道的方式完成匹配。然而，模式 (?=\d\d) 会告诉正则表达式引擎，必须检查随后的字符序列，只有当后面的字符序列中存在 \d\d 表示的两个数字时，才能匹配这两个字母字符。第一个 \d 元字符匹配 21 中的 2，第二个 \d 元字符匹配 21 中的 1。在两个字母字符后面存在两个数字的序列，因此向前查找模式 (?=\d\d) 强加的限制条件满足了。因此，整个模式匹配。

然后，正则表达式引擎到达第二行的开始位置。该行的测试文本是 AB1。正则表达式引擎首先尝试匹配 ^ 元字符。因为引擎当前位置就是行开始的位置，所以匹配成功。随后尝试匹配 AB1 中的 A 和 B 也都成功。这些组件匹配完成后，正则表达式引擎开始处理向前查找部分。由于 AB1 中只有一个数字，所以与模式 (?=\d\d) 匹配失败。虽然想要查找的所有字符都匹配了，但由于向前查找匹配失败，因此整个正则表达式失败。

8.2.1 肯定式向前查找

肯定式向前查找是指在要匹配的字符序列后面必须存在某些字符序列(通常与要匹配的字符序列不同)的条件约束下完成的匹配过程。

与正则表达式中的其他技术一样，对于同样的匹配结果经常会有多种使用向前查找的方法。例如，要匹配字符序列 States 中的 State，可以用：

```
(?=States)State
```

也可以用：

```
State(?=s)
```

第一种方法的含义是“查找一个后跟字符序列 States 的位置。如果该位置存在，则匹配字符序列 State”。第二种方法的含义是“匹配字符序列 State，但它后面必须跟一个小写的 s”。这两种模式都会匹配相同的字符序列。

1. 肯定式向前查找——Star Training 示例

在理解了如何使用肯定式向前查找后，再来看一下 Star Training Company 示例，并改进匹配的针对性。

假设这一次只用向前查找，那么解决问题的最直观方式就是使用下面的问题定义：

匹配字符序列 S、t、a 和 r，且它后跟一个空格符并后跟一个字符序列 T、r、a、i、n、i、n 和 g。

下面的模式实现这一问题定义：

```
Star(?= Training)
```

试一试：肯定式向前查找——Star Training 示例

(1) 打开 PowerGrep，并在 Search 文本区域中输入模式 Star(?= Training)。

(2) 在 Folder 文本框中输入文件夹名 C:\BRegExp\Ch08。如果下载时把示例文件存放在其他地方，则需要修改这里的路径。

(3) 在 File Mask 文本框中输入文件名 StarOriginal.txt，并单击 Search 按钮。

(4) 在 Results 区域中观察结果——如图 8-2 所示。注意，所有 6 个位于 Training(带有一个空格符)之前的字符序列 Star 都匹配了。由于 PowerGrep 显示文本的方式，可能需要拖动水平滚动条才能看到全部匹配项。

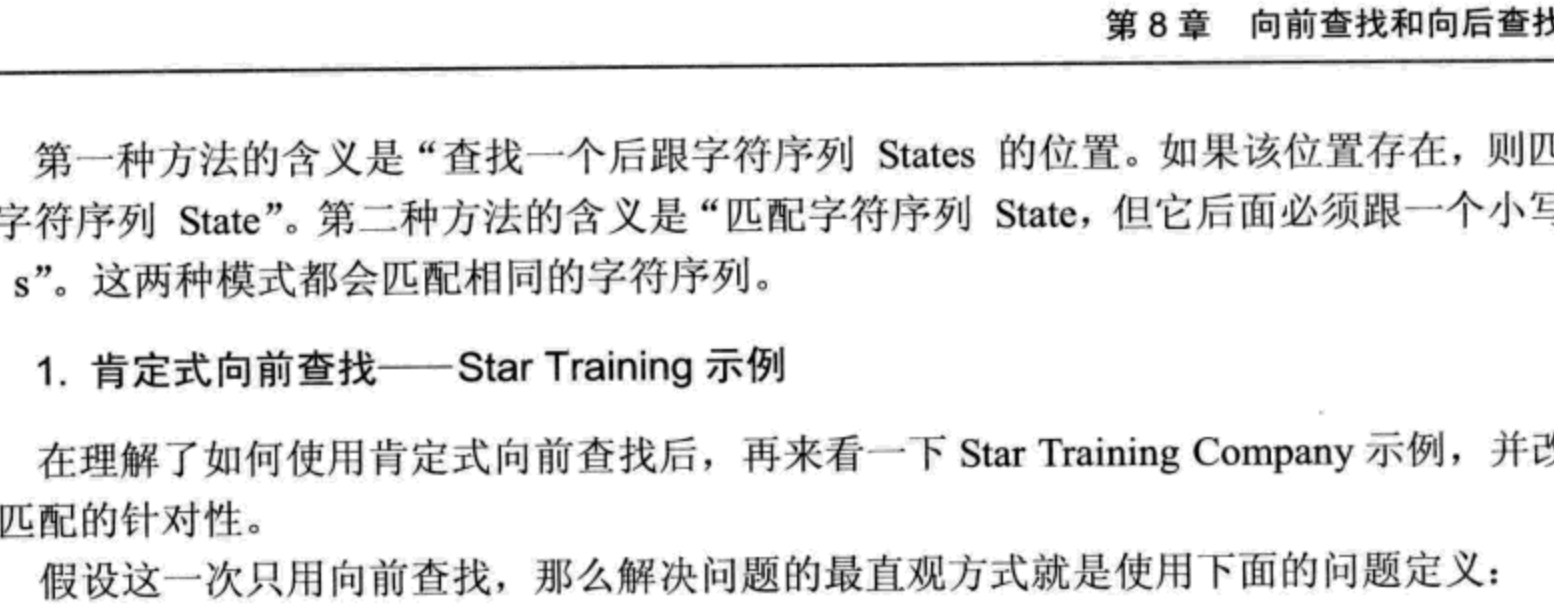


图　8-2

工作原理

正则表达式引擎从测试文档的开始位置进行匹配，它首先尝试匹配字符序列 Star。匹配 Star 的过程按照正常方式进行。然而，每次正常匹配字符序列 Star 之后，正则表达式引擎都会处理模式中向前查找的组件，查看在 Star 后面是否跟随一个空格符和字符序列

Training。

位于第一行的 Star 后跟指定的序列，因而匹配成功。

但是，第二行中的 Star 是 Starting 的一部分，虽然它与模式 Star 匹配，但在向前查找时却匹配失败。

2. 肯定式向前查找——根据同一个句子的后面匹配

肯定式向前查找能在同一个句子中查找两个单词。本节讨论如何在同一个句子中包含第二个单词的情况下找到第一个单词。假定数据中不包含带小数点的数(否则无法与句点字符区分)，而且也不包含由句点字符构成的省略号。

测试文件 Sentence.txt 的内容如下：

```
Here is a sentence where one can look ahead to interesting character sequences.

This sentence does not contain interesting characters.

Here is a sequence of characters.

Which sequence of characters is contained in this sentence?
```

问题定义如下：

匹配字符序列 sentence，并且在同一个句子中还跟有字符序列 sequence。

这些行中只有一行(第一行)中包含字符序列 sentence，而且在同一个句子的后面还跟有字符序列 sequence。

试一试：肯定式向前查找——根据同一个句子的后面匹配

(1) 打开 PowerGrep，并在 Search 文本区域中输入模式 sentence(?=.*sequence.*\.)。

(2) 在 Folder 文本框中输入文件夹名 C:\BRegExp\Ch08。如果下载时把本章测试文件放在其他文件夹中，则修改这里的路径。

(3) 在 File Mask 文本框中输入文件名 Sentence.txt，并单击 Search 按钮。

(4) 观察显示于 Results 区域中的结果，如图 8-3 所示。

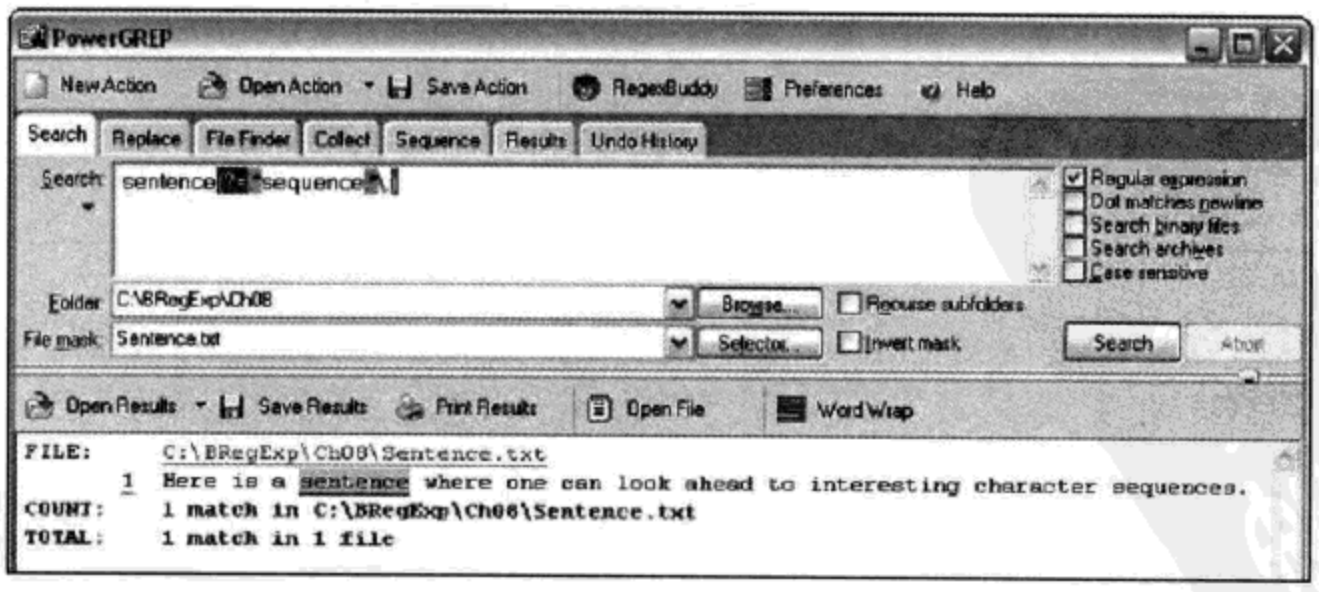


图 8-3

工作原理

正则表达式引擎将查找字符序列 sentence。如果它找到了这个字符序列，则再测试向前查找的条件是否满足。在测试向前查找模式(?=.*sequence.*\.) 时，引擎会从 sentence 中最后的 e 之后的位置开始，查找字符序列 sequence 及由模式 .* 表示的任何数量的居间字符。最后再测试由模式 \. 表示的后面的句点字符及由模式 .* 表示的任何数量的居间字符。

在第一行中，正则表达式模式的所有组件——包括向前查找，都被满足了，所以匹配成功。

在第二行中，尽管可以找到字符序列 sentence，但向前查找匹配失败。因为在同一行中没有发现字符序列 sequence，所以匹配失败。

在第三行中，根本没找到字符序列 sentence，所以匹配直接失败。

在第四行中，虽然字符序列 sentence 和 sequence 都存在，但顺序颠倒，不能满足向前查找的限制，所以匹配失败。

8.2.2　否定式向前查找

否定式向前查找是指在要匹配的字符序列后面不存在某些字符序列的附加条件约束下完成的匹配过程。

例如，在 Star Training Company 示例中，可能希望找到字符序列 s、t、a 和 r，但这个字符序列之后不能跟空格符和单词 Training。这样就可以找到那些不太可能作为 Star Training Company 名称一部分的 Star 的字符序列 s、t、a 和 r。

否定式向前查找由一个开始的圆括号后跟一个问号和一个感叹号，然后是要查找的字符串，最后是结束的圆括号组成。因此，如果想查找后面没有跟着单词 Training 的字符序列，可以把下面的模式作为正则表达式中的向前查找组件——假设使用的工具或语言支持 \b 元字符：

```
(?!\bTraining\b)
```

否定式向前查找也可以和其他限制条件组合使用。例如，通过指定匹配过程按区分大小写的方式进行并使用模式(小写的)star，可以避免 Star 与提供的模式成功匹配。

相应的问题定义可以声明如下：

匹配字符序列 S、t、a 和 r，但这个字符序列后面不能紧跟一个空格符和另一个字符序列 T、r、a、i、n、i、n 和 g。

试一试：否定式向前查找

(1) 打开 PowerGrep，并在 Search 文本区域中输入模式 Star(?! Training)。

(2) 在 Folder 文本框中，输入文件夹名 C:\BRegExp\Ch08。

(3) 在 File Mask 文本框中，输入文件名 StarOriginal.txt，并单击 Search 按钮。

(4) 在 Results 区域中观察结果，如图 8-4 所示(由于 PowerGrep 解释换行符的方式不同，可能会看到与该插图中类似的文本溢出屏幕的效果)。

注意，其中位于空格符及字符序列 Training 之前的 6 个 Star 都没有匹配。

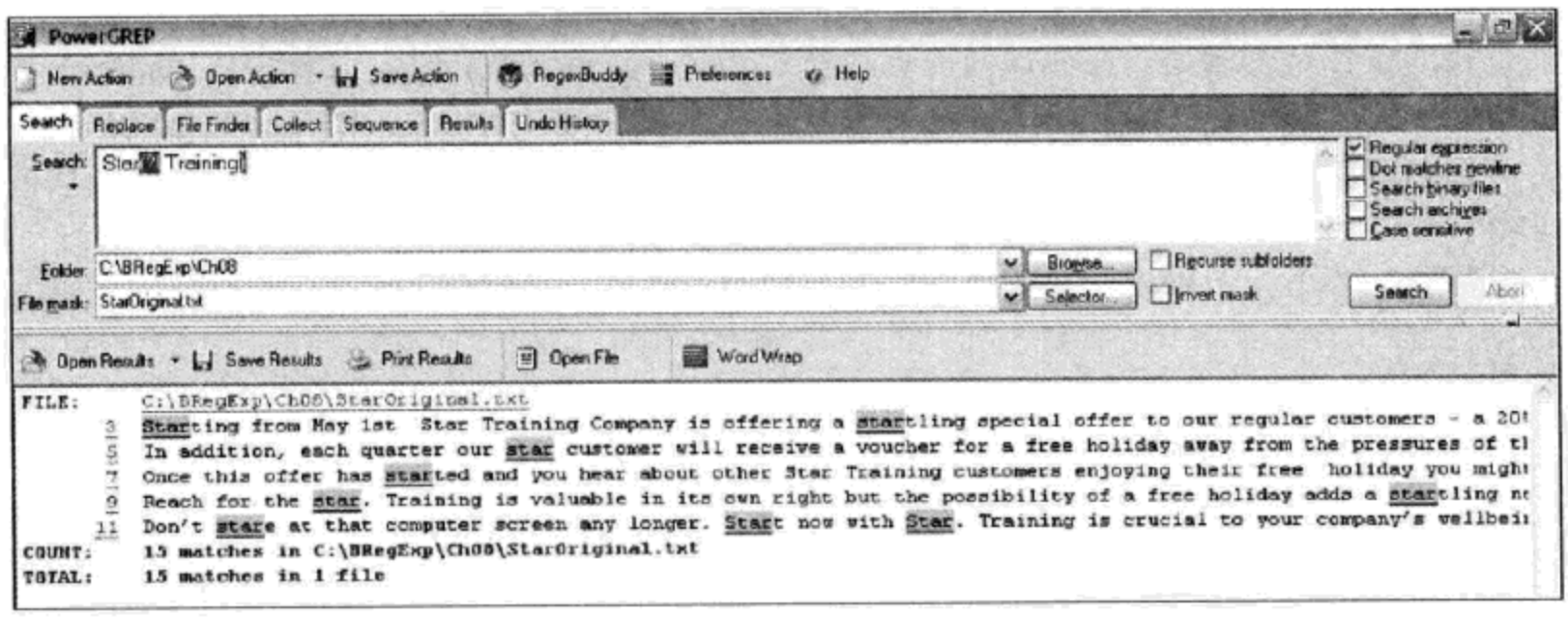


图 8-4

工作原理

首先，按正常方式匹配字符序列 Star。找到了 Star 后，正则表达式引擎就会测试向前查找组件。如果 Star 后面的字符序列是一个空格后跟 Training，那么向前查找组件匹配失败(因为这是否定式向前查找)。只有否定式向前查找匹配成功，Star 才能最终匹配。

在第一行，Star 后跟一个空格符，然后是 Training Company。但因为否定式向前查找指定的字符序列被找到了，所以向前查找失败。因此，第一行中的 Star 不匹配。

在第二行，Star 是 Starting 的一部分。正则表达式引擎匹配 Star 后，测试下一个字符是不是一个空格符。此时的那个字符是 Starting 中的第二个 t。所以，由向前查找指定的模式没有找到匹配项。因为它是一个否定式向前查找，所以该向前查找的条件满足。因此，Starting 中的 Star 匹配了。

8.3 肯定式向前查找的例子

下面一节介绍肯定式向前查找的几种用途。

有时可能需要测试一个文档并从中找出某些文本，同时在其后还存在其他一些文本。

8.3.1 在同一文档中使用肯定式向前查找

假设有一个文档 Databases.txt，我们希望测试其中是否提到 Microsoft SQL Server，而文档后面是否也提到 MySQL 数据库。Databases.txt 的内容如下：

```
The current version of Microsoft SQL Server is SQL Server 2000. However a new
version, SQL Server 2005 is scheduled for release for the calendar year 2005. The
MySQL database product lacks some of the features of big commercial database
products like SQL Server but the product team is working hard to provide an
improved set of features.
```

试一试：在同一个文档中使用肯定式向前查找

(1) 打开 RegexBuddy，并在上面的选项卡中选择 Match。

(2) 在 Match 选项卡中输入正则表达式模式 SQL Server(?:.*MySQL)，并选择面板中部的 Test 选项卡。

(3) 单击 Open File 按钮，找到 C:\BRegExp\Ch08，并打开 Databases.txt 文件。如果你下载时把测试文件存放在其他位置，则要在相应位置中查找。

(4) 单击 Find First 图标，并观察下方窗格中突出显示的文本。

图 8-5 显示了结果。注意，RegexBuddy 突出显示了 MySQL 出现之前的匹配 SQL Server 的全部文本。严格来讲，正则表达式只匹配了字符串 SQL Server。高亮区域突出显示了向前查找组件所找到的文本。按照惯例，PowerGrep 也会如此。

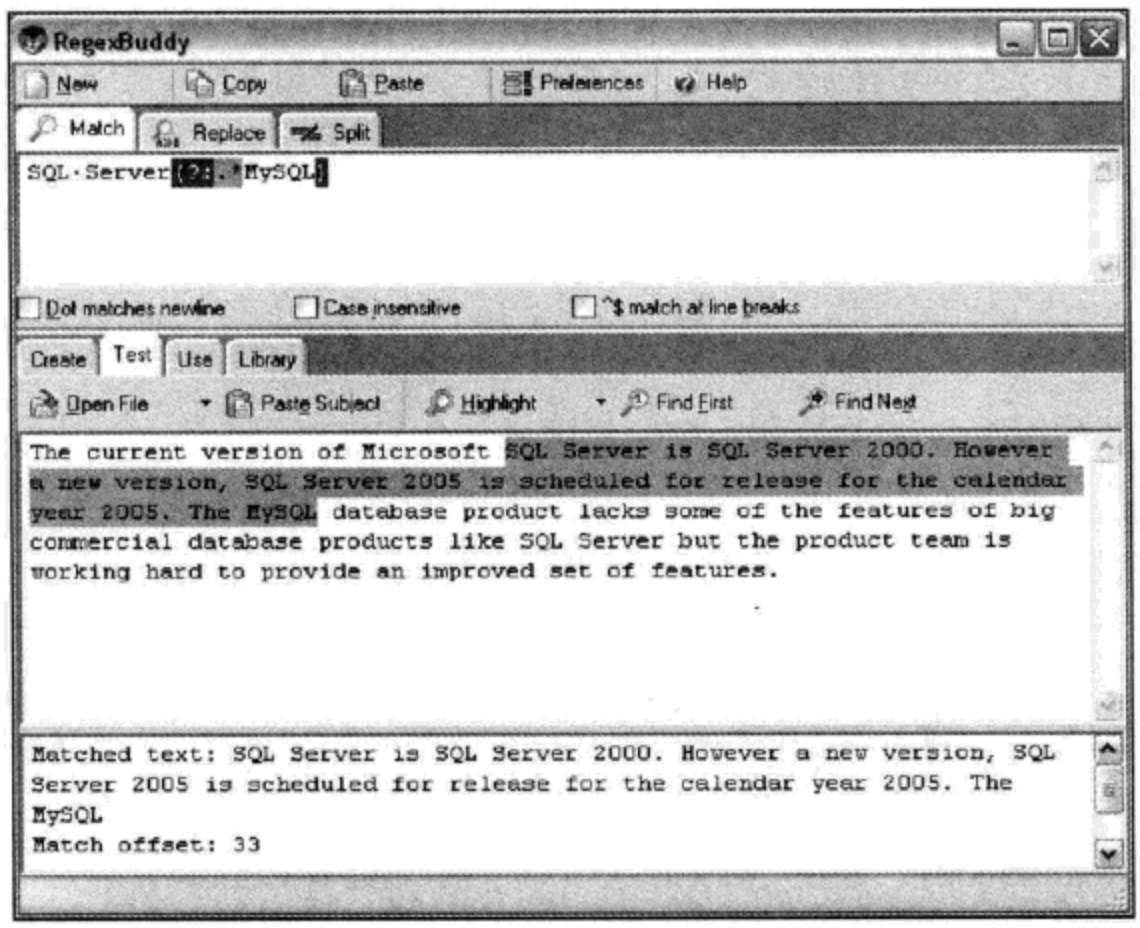


图　8-5

工作原理

匹配过程以正常方式找到直接量字符序列 SQL Server。正则表达式引擎从第一行中 The 之前的位置开始尝试匹配。然后，找到了与模式 SQL Server 匹配的直接量文本。接着，正则表达式引擎尝试满足向前查找限定的条件。它会继续查找随后出现的字符序列 MySQL。模式 .* 表示字符序列 MySQL 可以出现在文档中位于 SQL Server 之后的任何位置。

8.3.2　插入单引号

本例会在适当位置补充一个可能是无意间忽略的单引号(即表示名词所有格的单引号。译者注)。随着在手机中编辑文本的情况日渐增多，有时候一些拼写错误或不恰当的缩写词

会充斥于正式的文档中。比如下面的测试文本：

```
This is not Andrews first book.

This book is Andrews.
```

在测试文本的两行中，Andrews 的 w 与 s 之间都应该有一个单引号，用以表示所有格。假定字符序列 Andrews 不可能是更长的字符序列的一部分，且字符序列 Andrews 后面的字符可能是一个空格符或者句点字符。

还有一种可能，比如说 Andrews 后面可能跟一个问号，如下所示：

```
Is this book Andrews?
```

因此还必须考虑小写的 s 后面可能出现其他字符的情况。一种方案就是指定字符 s 后面必须跟着一个词边界。此时的 s 后面既可以是一个空白符，也可以是一个标点符号。

相应的问题定义可以表述如下：

匹配字符序列 A、n、d、r、e、w，后面跟一个小写的 s，且 s 后面是一个词边界。

这个问题定义可以通过下面的模式来表达：

```
Andrew(?=s\b)
```

如果想把匹配限定为 s 后跟跟一个空格或者一个句点字符的情况，那么可以使用下面的模式——通过交替选择指定二选一的向前查找限定条件：

```
Andrew((?=s )|(?=s\.))
```

试一试：插入一个单引号

本例会示范前面两个模式的效果。

(1) 打开 RegexBuddy，并在 Match 选项卡中输入模式 Andrew((?=s)|(?=s\.))。

(2) 在 Test 选项卡中，输入下列测试文本：

```
This is not Andrews first book.

This book is Andrews.

Is this book Andrews?
```

(3) 单击 Find First 图标，并观察匹配的字符序列。

(4) 再单击两次 Find Next 图标，观察每次单击后是否存在匹配项。

图 8-6 显示了结果，匹配了第一次出现的 Andrews 字符序列。但是，第三次单击 Find Next 按钮后，找不到匹配项。这是因为后跟一个问号的字符序列 Andrews 不匹配向前查找。

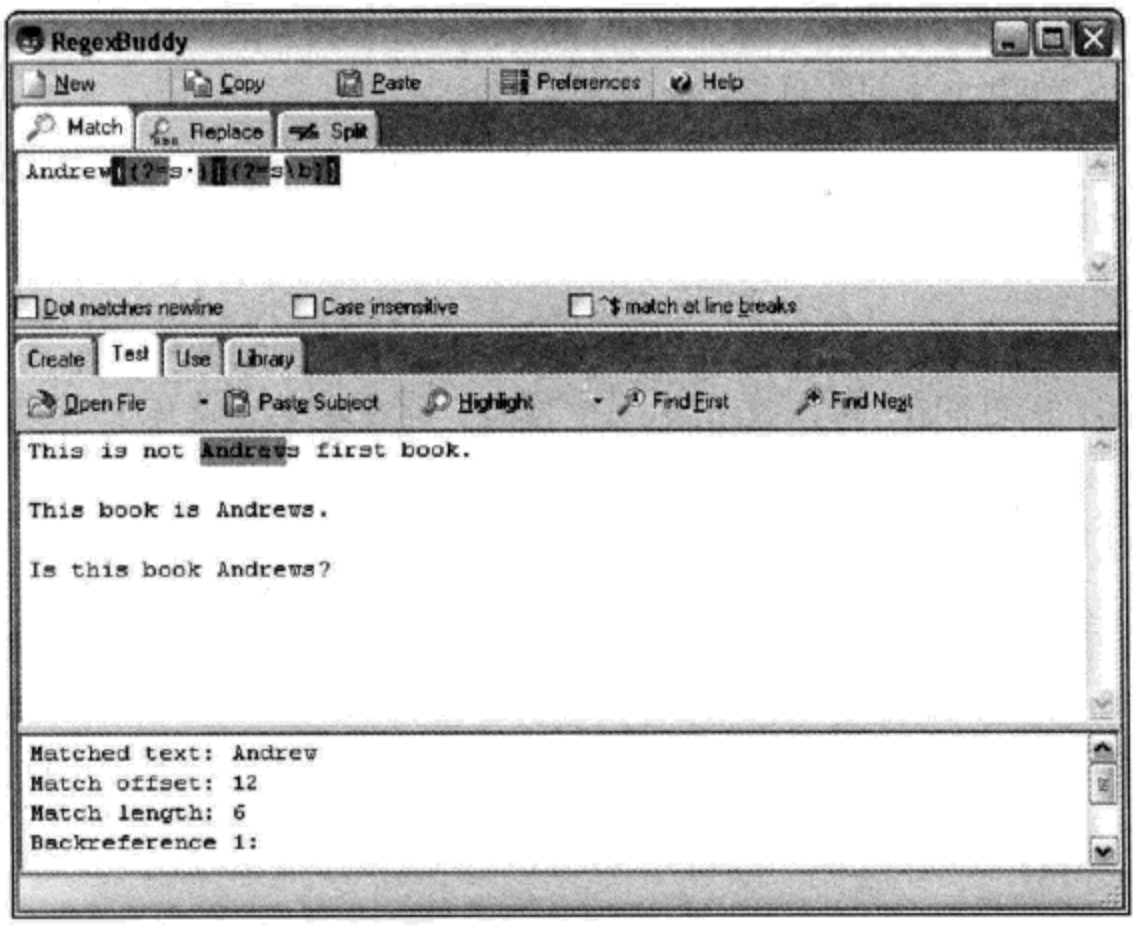


图　8-6

(5) 在 Match 选项卡中把正则表达式修改为 Andrew(?=s\b)。

(6) 单击 Find First 图标，然后再单击两次 Find Next 图标，观察每次单击后匹配的字符序列。

图 8-7 显示的是在单击两次 Find Next 图标后的界面外观。使用修改后的正则表达式后，所有三个字符序列 Andrews 现在都匹配了。

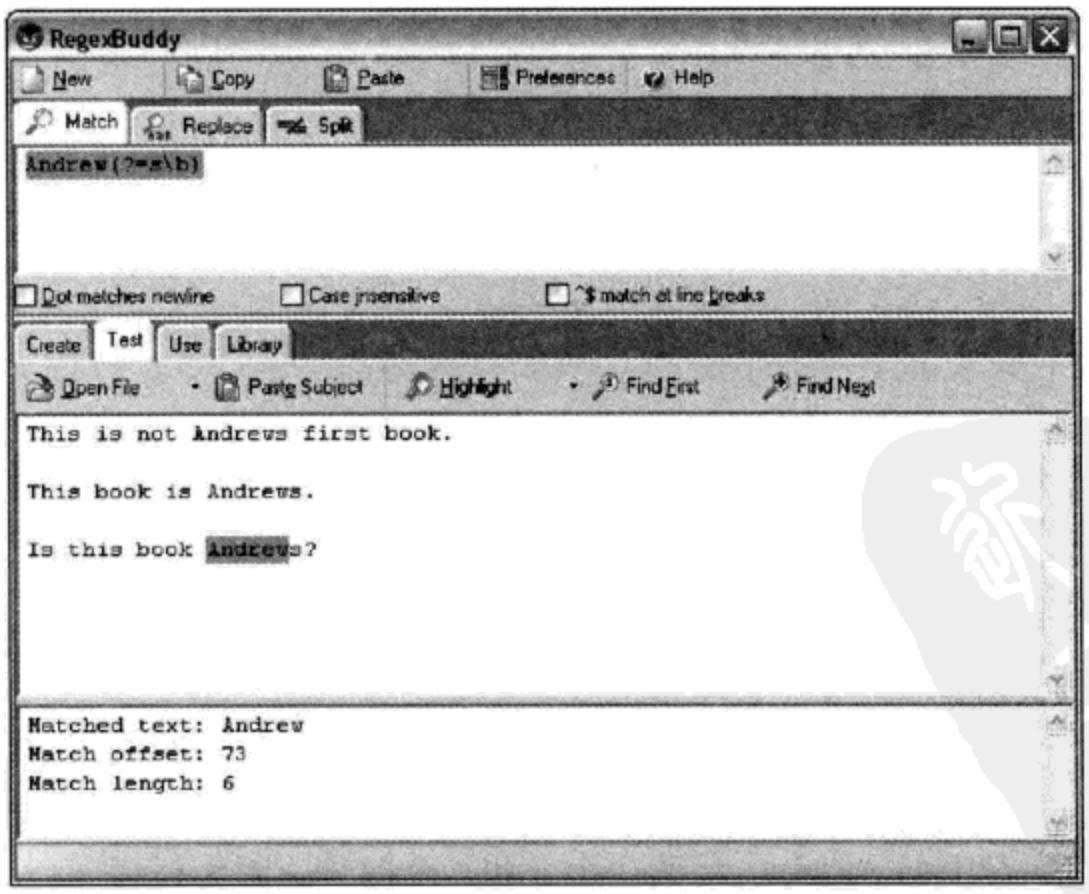


图　8-7

(7) 既然知道了每个匹配项，那么就可以修改正则表达式来创建两个组，以便插入必要的单引号构成 Andrew's 的所有格。

在 Match 选项卡中把正则表达式修改为 (Andrew)(s)(?=\b)。

(8) 使用 Find First 和 Find Next 图标，确定在对模式进行小修改之后的三个字符序列 Andrews 仍然匹配。

(9) 单击 Replace 选项卡。在 Replace 选项卡下方的文本区域中输入 $1'$2。

(10) 在 Test 选项卡中，单击 Replace All 图标，并观察 Test 选项卡下方显示的结果(可能需要调整窗口大小以便看到所有结果)。

图 8-8 显示的是这个步骤之后的结果。

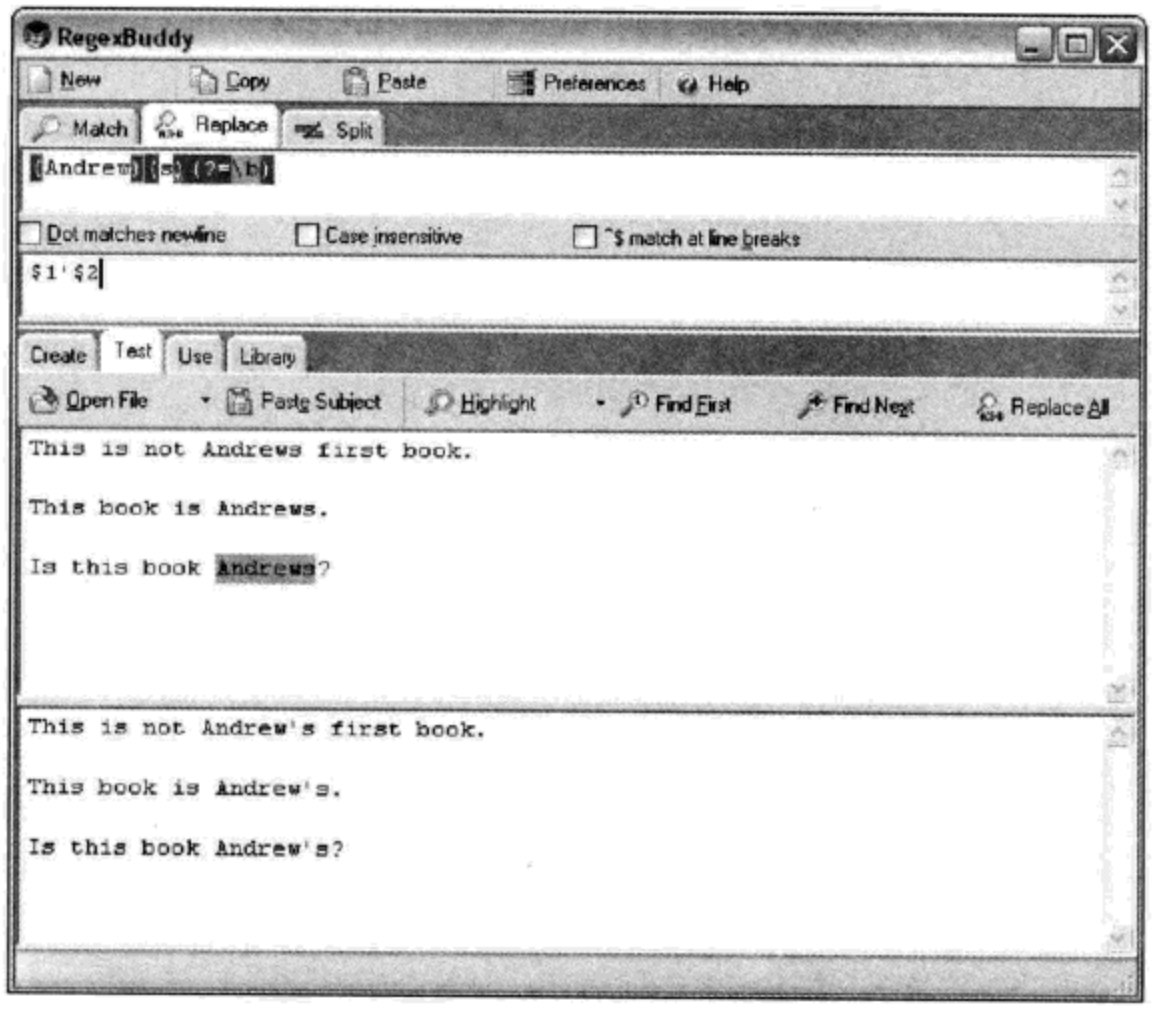


图 8-8

工作原理

模式 Andrew((?=s)|(?=s\.)) 匹配字符序列 Andrew 后跟 s 和一个空格符或者 s 和一个句点字符。

在第一行，字符序列 Andrew 后跟一个 s 和一个空格符，所以它满足了第一个向前查找的限定条件。因为匹配成功且满足向前查找限定条件，所以整个正则表达式匹配成功。

在第二行中，字符序列 Andrew 后跟 s 和一个句点字符。这满足第二个向前查找的限定条件。

在第三行，字符序列 Andrew 后跟一个 s 和一个问号。因为任何一个向前查找模式都不匹配问号，所以匹配失败。

当把正则表达式模式修改为 Andrew(?=s\b) 后，当匹配 Andrew 时，向前查找限定条件是 s 后跟一个词边界。所有三行中的 Andrews 后面都是一个词边界。在第一行中，空格符之前是一个词边界。在第二行中，句点字符前面是一个词边界。在第三行中，在问号前面是一个词边界。所以，每个 Andrew 都匹配。

当把正则表达式修改为 (Andrew)(s)(?=\b) 时，把捕获的字符序列 Andrew 保存在$1 中，把捕获的 s 保存在$2 中。但向前查找不会捕获字符。所以，要插入一个单引号，就是要让 $1(Andrew)后跟一个单引号，再后跟 $2(小写的 s)。

8.4　向后查找

向后查找测试的是一个要匹配的字符序列前面有(肯定式向后查找)或者没有(否定式向后查找)另外一个字符序列。

比如说，如果只想在前面有字符序列 Dr.(一个大写的 D，一个小写的 r，一个句点和一个空格)的情况下才匹配姓氏 Jekyll，那么就可以使用下面的模式：

```
(?<=Dr. )Jekyll
```

组件 (?<=Dr.) 表示用于向后查找的一个字符序列(即从该字符序列所在的位置向后查找。译者注)，而组件 Jekyll 则匹配直接量。

8.4.1　肯定式向后查找

肯定式向后查找是对匹配的一个限定条件。即，只有当向后查找中包含的模式位于要匹配的模式之前时，该模式才会匹配。

试一试：肯定式向后查找

(1) 打开 Komodo Regular Expression Toolkit，并删除残留的正则表达式和测试文本。

(2) 在 Enter a string to match against 区域中输入测试文本 Mr. Hyde and Dr. Jekyll are characters in a famous novel。

(3) 在 Enter a regular expression 区域中，输入模式 (?<=Dr.)Jekyll。

(4) 观察 Enter a string to match against 区域中突出显示的文本，而其下方灰色区域中有关结果的描述是：Match succeeded: 0 groups。

如图 8-9 所示，注意字符序列 Jekyll 突出显示。

(5) 把正则表达式模式修改为 (?<=Mr.)Jekyll。

(6) 此时灰色区域中显示的描述信息为 No matches found。

(7) 把正则表达式模式修改为 ((?<=Mr.)|(?<=Mister))Hyde。确保 Mister 的 r 后面有一个空格符。如果没有这个空格符，将找不到匹配项。

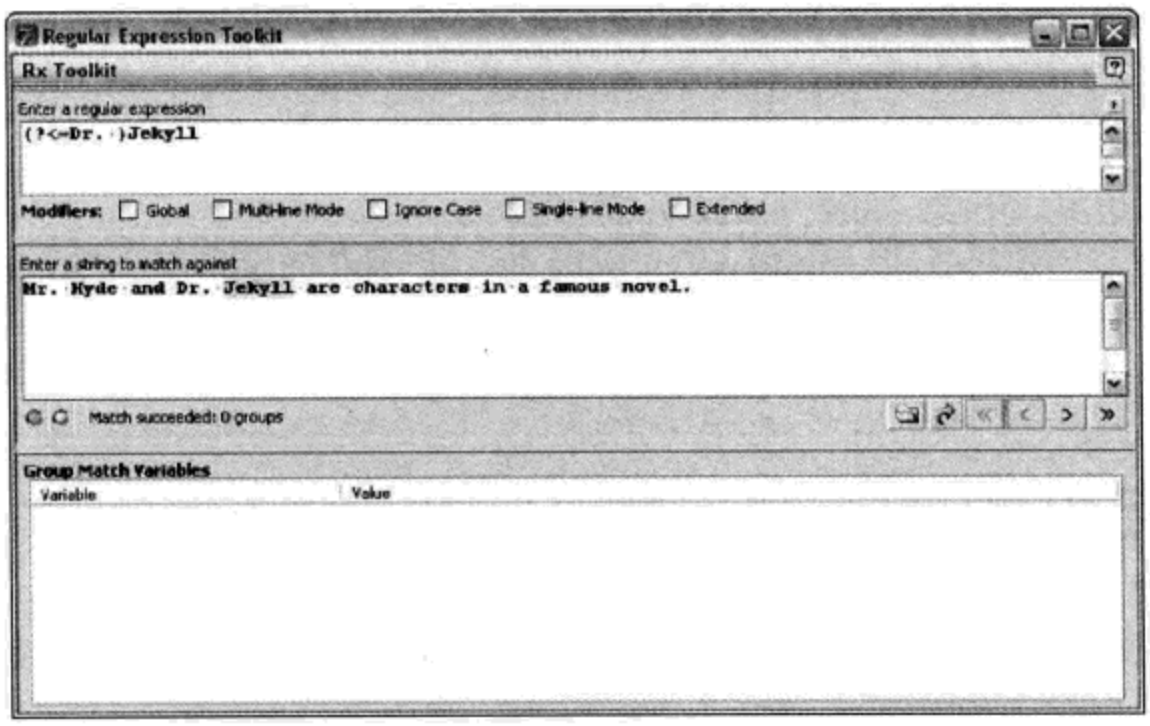


图 8-9

(8) 此时观察到灰色区域中的描述信息为：Match succeeded: 1 group。我们也会注意到，字符序列 Hyde 被突出显示。

(9) 将测试文本中的 Mr.修改为 Mister。

(10) 此时，灰色区域中显示的描述信息还是 Match succeeded: 1 group。图 8-10 显示了这一结果。

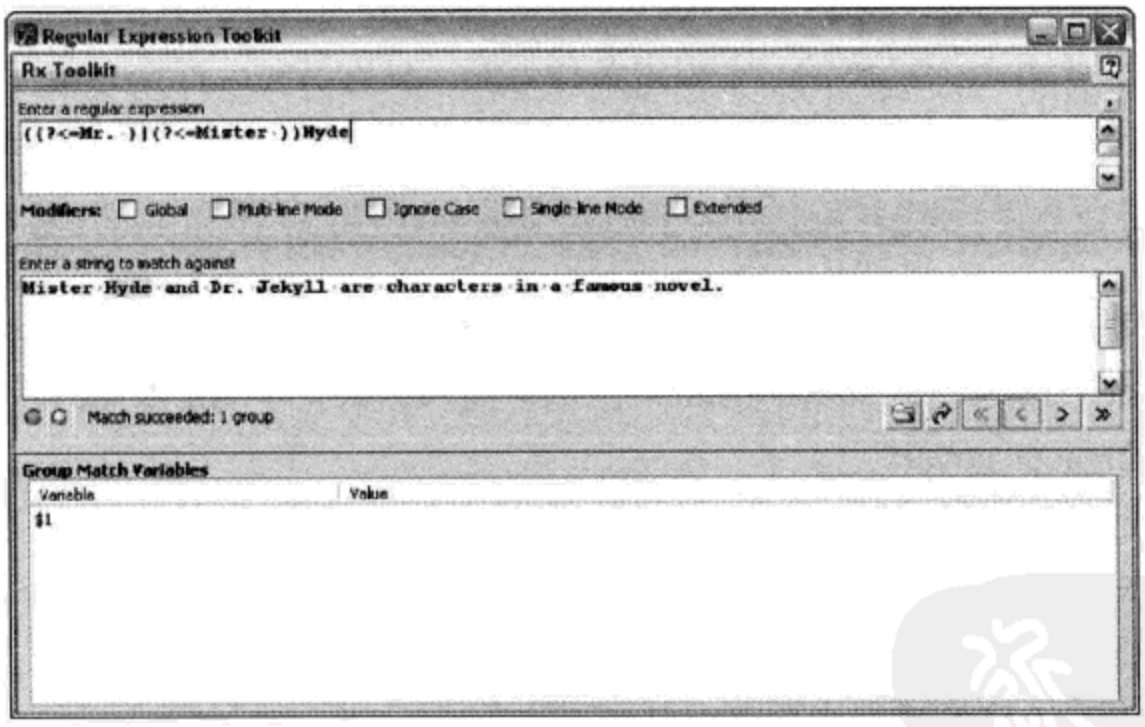


图 8-10

工作原理

下面有关正则表达式引擎如何操作的描述只是从概念上进行分析，可能无法反映每个正则表达式引擎实际的运作方式。本例中要匹配的文本是字符序列 Jekyll。

匹配过程起始于测试文本的开始处。正则表达式引擎首先会检查是否有一个大写的 J。如果有，则匹配成功。然后，继续尝试匹配组成 Jekyll 的其他字符。如果其中有一次匹配

失败，整个模式就失败，那么正则表达式引擎就会在文本中向前移动重新开始匹配字符序列 Jekyll。

如果找到了字符序列 Jekyll，那么正则表达式引擎会转到 Jekyll 中 J 之前的位置。它会检查前面的字符是不是一个空格。如果是，再测试空格之前的字符是不是一个句点字符。如果是，再测试句点之前是不是一个小写的 r。如果还是，那么最后再测试 r 之前是否是一个大写的 D。如果是，那么向后查找模式匹配成功。由于 Jekyll 匹配成功，而字符序列 Jekyll 前面是一个字符序列 Dr.(包括一个空格符)的限定条件也满足，整个正则表达式就成功匹配。

当把模式修改为 (?<=Mr.)Jekyll 后，字符序列 Jekyll 仍和以前一样成功匹配。但是，当正则表达式引擎检查它前面的字符序列时，尽管事实上(在反方向依次匹配时)空格符、句点字符和小写的 r 都是存在的，但再往前没有找到大写的 D，所以向后查找限定条件匹配失败。由于向后查找限定条件没有满足，所以整个正则表达式不匹配。

向后查找中也可以使用交替选择。相应的问题定义可以描述如下：

匹配字符序列 Hyde，但它前面必须是字符序列 Mr.(包括最后的空格符)，或者是字符序列 Mister(包括最后的空格符)。

当把模式修改为 ((?<=Mr.)|(?<=Mister))Hyde 之后，正则表达式引擎首先尝试匹配字符序列 Hyde。当它返回到 Hyde 中 H 之前的位置时，意味着它已经成功匹配了这个字符序列。然后，它还需要满足匹配 Hyde 前面字符序列的限定条件。

模式 ((?<=Mr.)|(?<=Mister))Hyde 使用圆括号将必须位于 Hyde 前面的两个可以相互替代的模式进行了分组。其中，由模式 (?<=Mr.) 指定的第一个选项，要求在 Hyde 之前必须有包含 M、r、一个句点和一个空格符的四个字符序列。在前面第 8 步中，这个四个字符的序列匹配。

当在测试文本中把 Mr. 改为 Mister 后，另一个选项开始起作用。模式 (?<=Mister) 要求在 Hyde 前面有一个包含 7 个字符的序列(Mister 加上一个空格符)。事实上，此时同样满足了向后查找的条件。

在下面的例子中会看到，向后查找非常重要。

试一试：肯定式向后查找的位置

(1) 打开 RegexBuddy，单击 Match 选项卡，并输入正则表达式 (?<=like)SQL Server。

(2) 单击 Test 选项卡，再单击 Open File 图标并打开 Databases.txt 文件。

(3) 单击 Find First 图标，并在 Test 选项卡的窗格中观察突出显示的文本，如图 8-11 所示。

(4) 在 Match 选项卡中把正则表达式修改为 SQL Server(?<=like)。

(5) 单击 Test 选项卡中的 Find First 图标。确认当前没有突出显示的文本。

(6) 在 Match 选项卡中把正则表达式修改为 SQL Server(?<=like SQL Server)。

(7) 单击 Test 选项卡中的 Find First 图标。确认此时文本中又出现了一个匹配项，如图 8-12 所示。

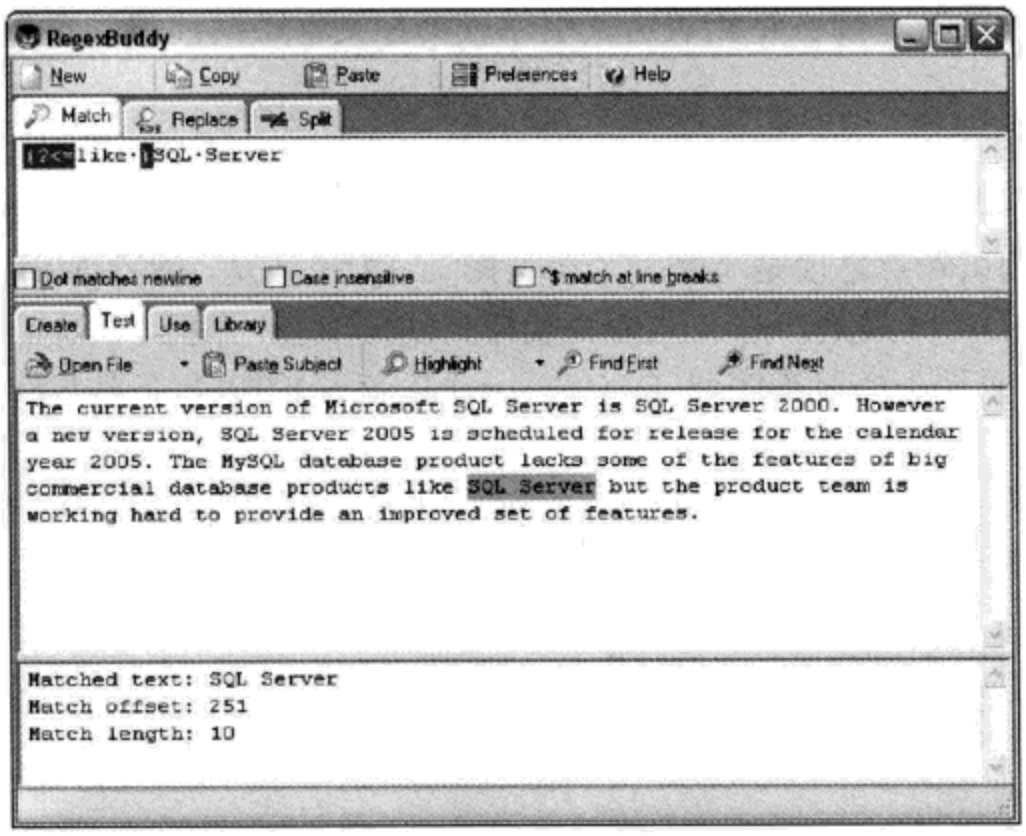


图 8-11

工作原理

当模式是(?<=like)SQL Server 时，向后查找会从 SQL 中 S 之前的位置开始反向查找。由于测试文本中存在字符序列 like SQL Server，所以匹配成功。而当模式修改为 SQL Server(?<=like)时，向后查找则从 Server 中 r 之后的位置开始反向查找。由于该位置的前面是 Server 不是 like，而向后查找尝试匹配的是字符序列 like，所以匹配失败。

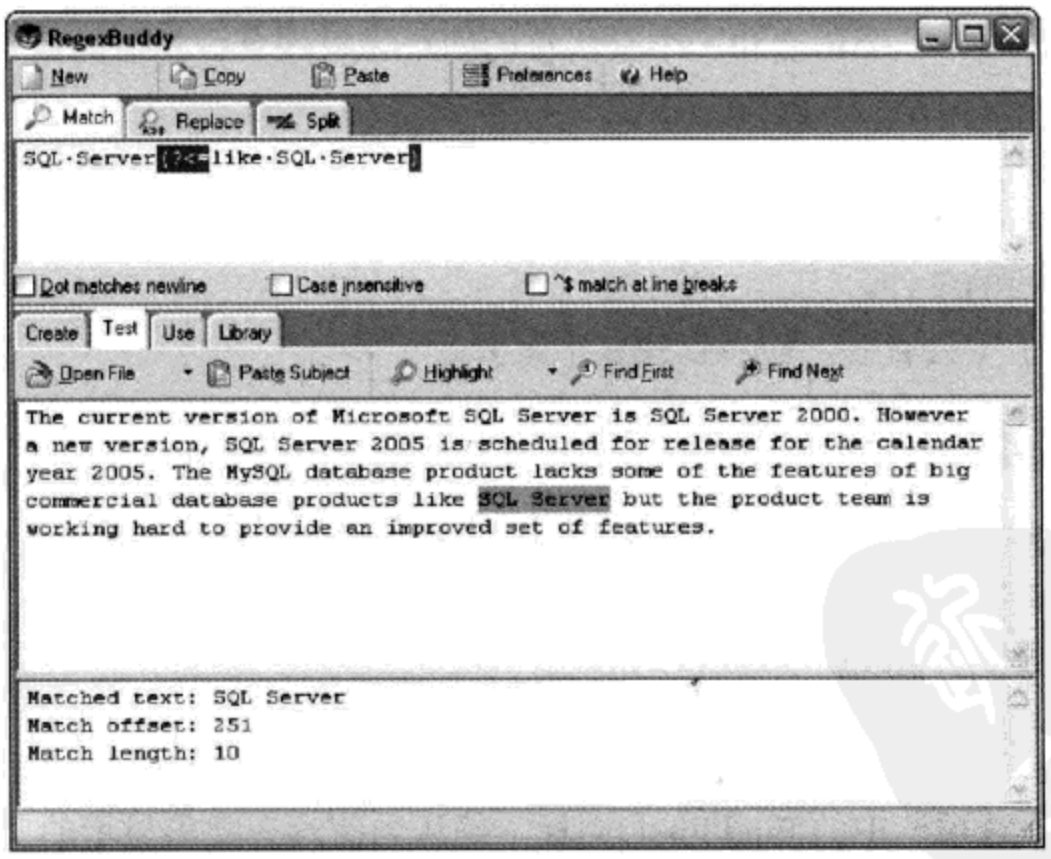


图 8-12

8.4.2　否定式向后查找

否定式向后查找是另外一种对匹配的限定条件。即，只有当向后查找模式匹配的字符序列没有出现在要匹配的模式之前时，该模式才会匹配。

试一试：否定式向后查找

查找前面没有字符序列 like 后跟一个空格符的字符序列 SQL Server。

(1) 打开 RegexBuddy，单击 Match 选项卡，并输入正则表达式(?<!like)SQL Server。

(2) 单击 Test 选项卡，再单击 Open File 图标，并打开 Databases.txt 文件。

(3) 单击 Find First 图标，并在 Test 选项卡的窗格中观察突出显示的文本，如图 8-13 所示。

(4) 通过单击几次 Find Next 图标查找其他匹配项。注意哪个 SQL Server 匹配，哪个不匹配。

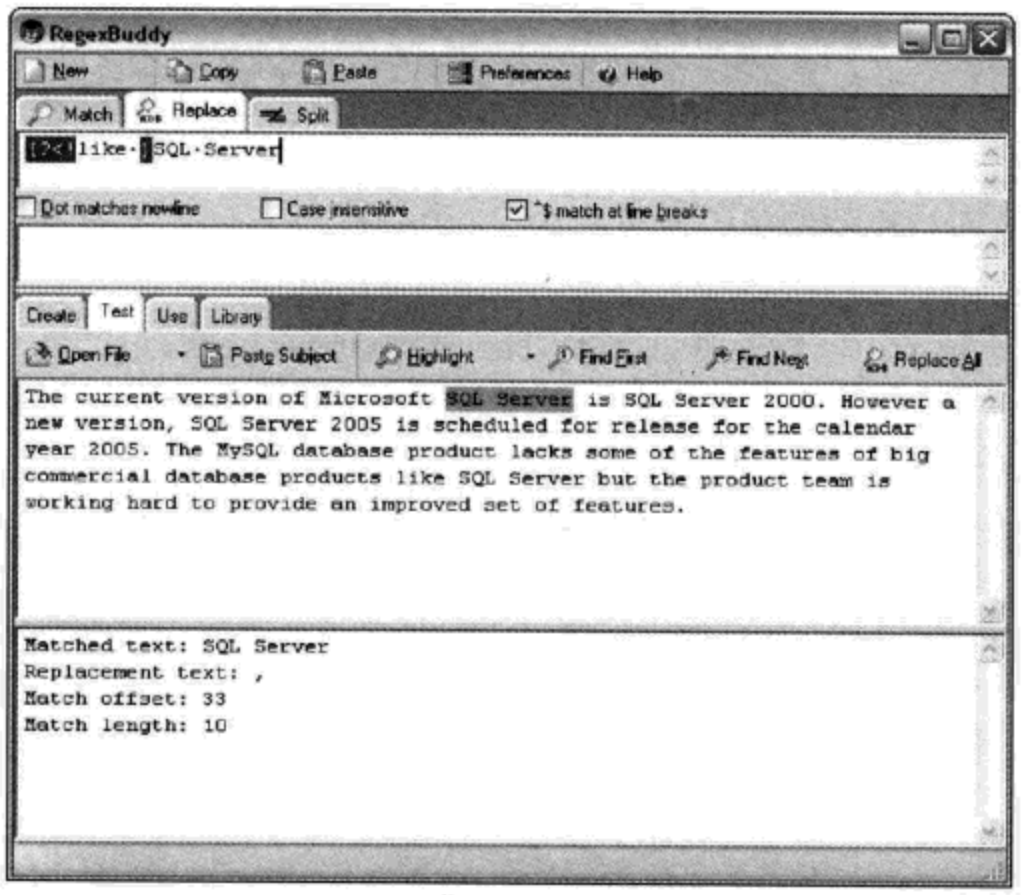


图　8-13

工作原理

当正则表达式引擎匹配字符序列 SQL Server 后，它会检查前一个字符序列是否与在向后查找中指定的模式匹配。

第一次匹配的 SQL Server 前面不是字符序列 like 后跟一个空格符。所以，满足否定式向后查找的限定条件。由于字符序列 SQL Server 匹配且否定式向后查找的限定条件也满足，所以整个正则表达式匹配。

测试文本中唯一没有成功匹配的字符序列 SQL Server 是位于单词 like 之后的那个实例。由于字符序列 like 后跟一个空格符的存在不满足向后查找限定条件，所以尽管字符序列 SQL Server 匹配了，但因其不满足向后查找的限定条件，整个正则表达式匹配失败。

8.5 如何匹配位置

通过组合使用向前查找和向后查找，可以匹配字符间的位置。例如，想匹配下面示例文本中 Andrew 之前的位置：

```
This is Andrews book.
```

可以把问题定义声明如下：

匹配一个前面是字符序列 is 后跟空格符，后面是字符序列 Andrew 的位置。

可以使用下面的模式来匹配这个位置：

```
(?<=is )(?=Andrew)
```

试一试：匹配一个位置

(1) 打开 RegexBuddy。在 Match 选项卡中输入正则表达式模式 (?<=is)(?=Andrew)。

如果在本章前面替换的例子中使用过 RegexBuddy，那么在 Replace 选项卡中删除替换文本。

(2) 在 Test 选项卡中，输入测试文本 This is Andrews book。

(3) 单击 Find First 图标，观察 Test 选项卡下方窗格中的结果，如图 8-14 所示。在屏幕中，可以看到光标在 Andrews 中 A 之前的位置上闪烁。

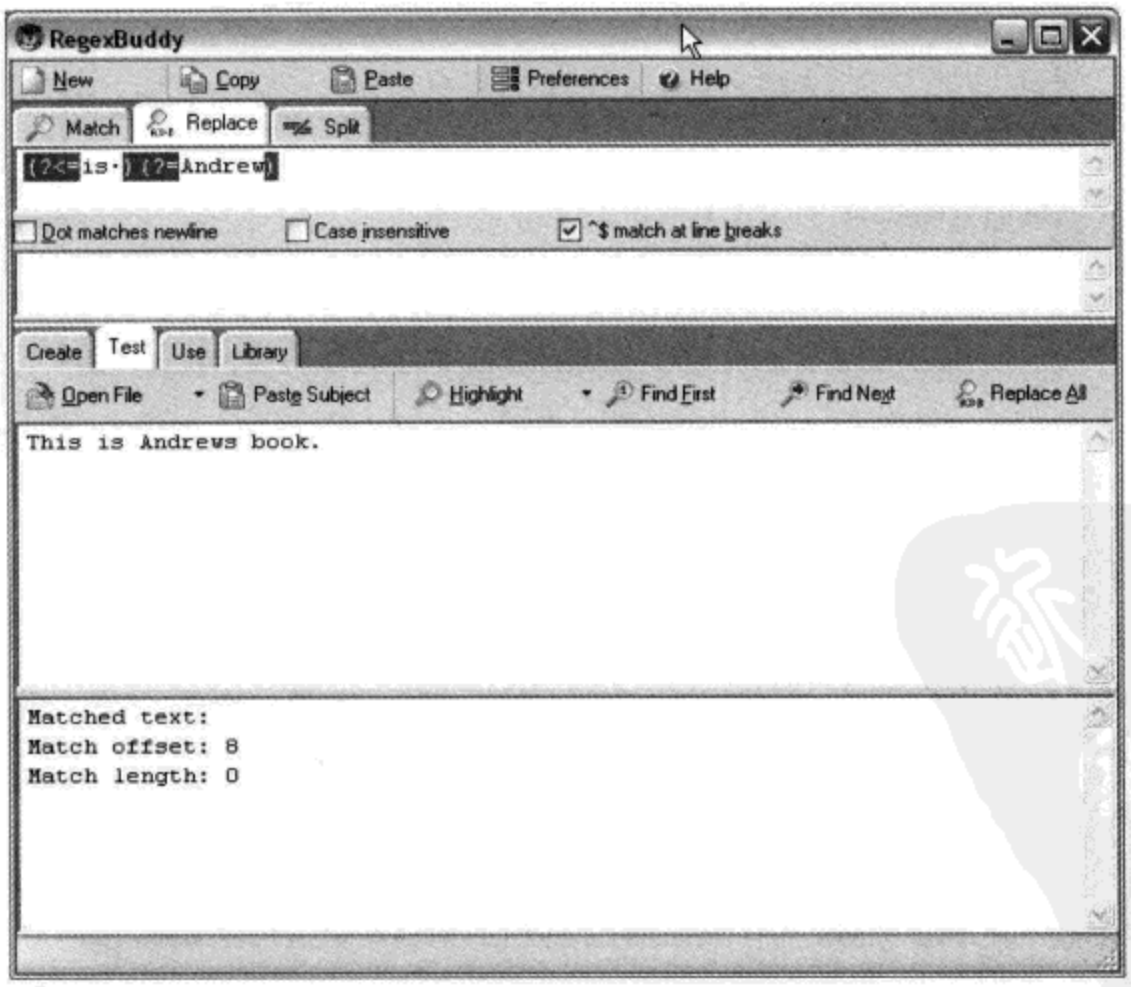


图 8-14

工作原理

正则表达式引擎会从文档的开始位置开始匹配，并测试每一个位置，看哪个位置能同时满足向后查找和向前查找的限定条件。在测试文本中，只有 Andrews 中 A 之前的位置能同时满足这两个条件。因此，这个位置就是唯一匹配的位置。

为大数添加逗号

另外一种组合使用向后查找和向前查找的方法是为大数添加逗号。

假设虚构的 Star Training Company 的销售额是$1,234,567。这个数值可能会不包含其中的逗号，以整数形式保存。然而，出于可读性的考虑，通常在财务数据或其他数值数据中又会添加逗号。

为大数添加逗号的过程，本质上是匹配数字间的位置并以逗号替换该位置的过程。

在某些欧洲语言中，使用句点字符作为千分位分隔符，正如在英语中使用逗号一样。可以对下面所用的技术稍做修改来实现向数字值中添加句点。

首先，假设有一个数字 1234，分析如何在适当的位置加上逗号。我们想在 1 和 2 之间插入一个逗号。在该位置插入逗号是因为在这个位置与该字符串结尾之间有三个数字。

试一试：为一个四位数添加逗号分隔符

(1) 打开 RegexBuddy。在 Replace 选项卡上方的窗格中输入模式(?<=\d)(?=\d\d\d)，并在下方的窗格中输入一个逗号。

(2) 在 Test 面板中，单击 Find First 图标。确认存在一个匹配项——在 Test 选项卡下方的窗格中可以看到。

(3) 单击 Replace All 图标，并观察在 Test 选项卡下方窗格中显示的替换文本(如图 8-15 所示)。替换后的文本是 1,234，这正是我们想要的结果。但这个正则表达式只能用于为 4 位数字添加逗号。

(4) 在 Test 选项卡上方的窗格中将测试文本修改为 1234567。

(5) 单击 Replace All 图标，并观察 Test 选项卡下方窗格中的替换文本。替换后的文本是 1,2,3,4,567，这不是我们想要的结果。如图 8-16 所示，在所有离右边至少有三个数字的位置上都被插入了一个逗号。

(6) 把模式修改为 (?<=\d)(?=(\d\d\d)+)。

(7) 单击 Replace All 图标，并观察 Test 选项卡下方窗格中的替换文本，发现仍然包含我们不想要的逗号。

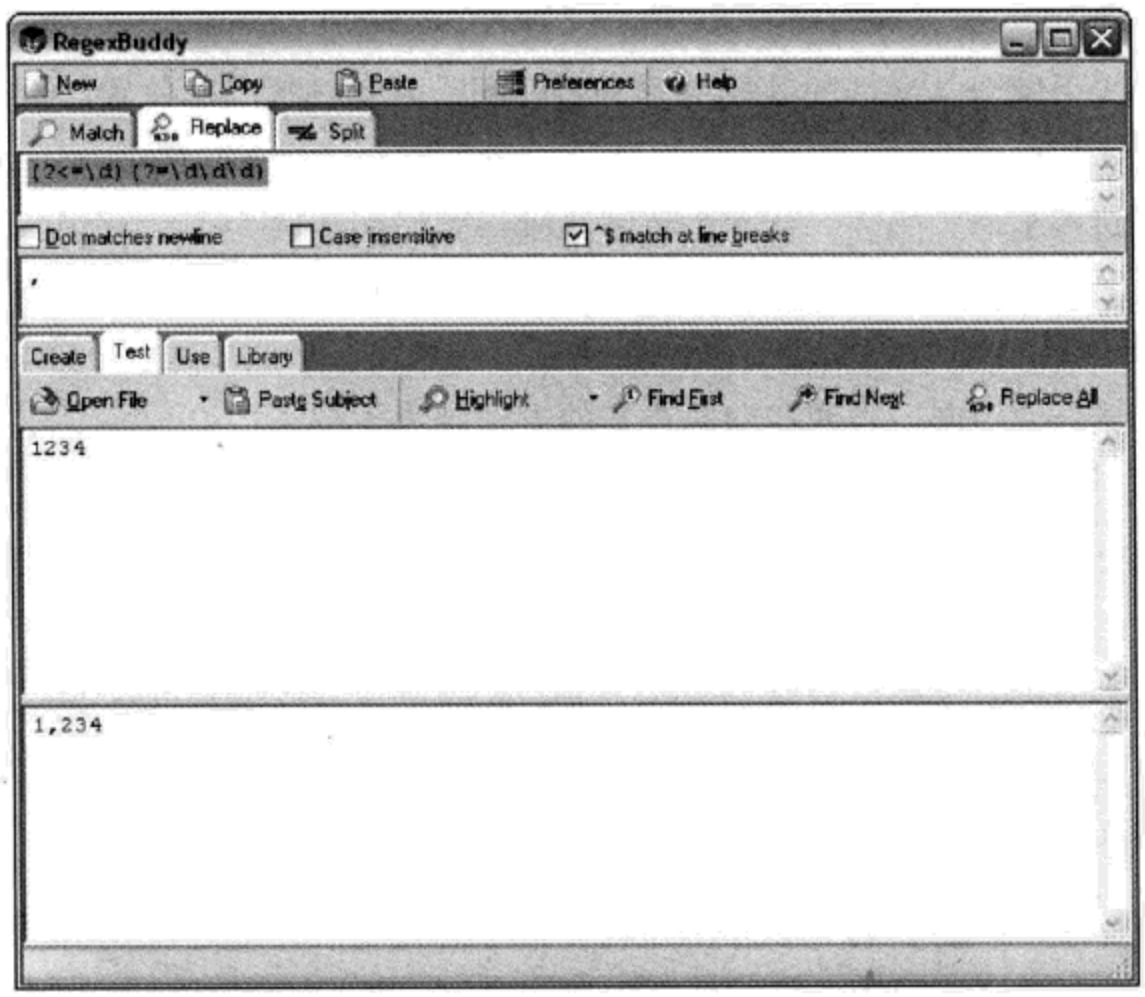


图 8-15

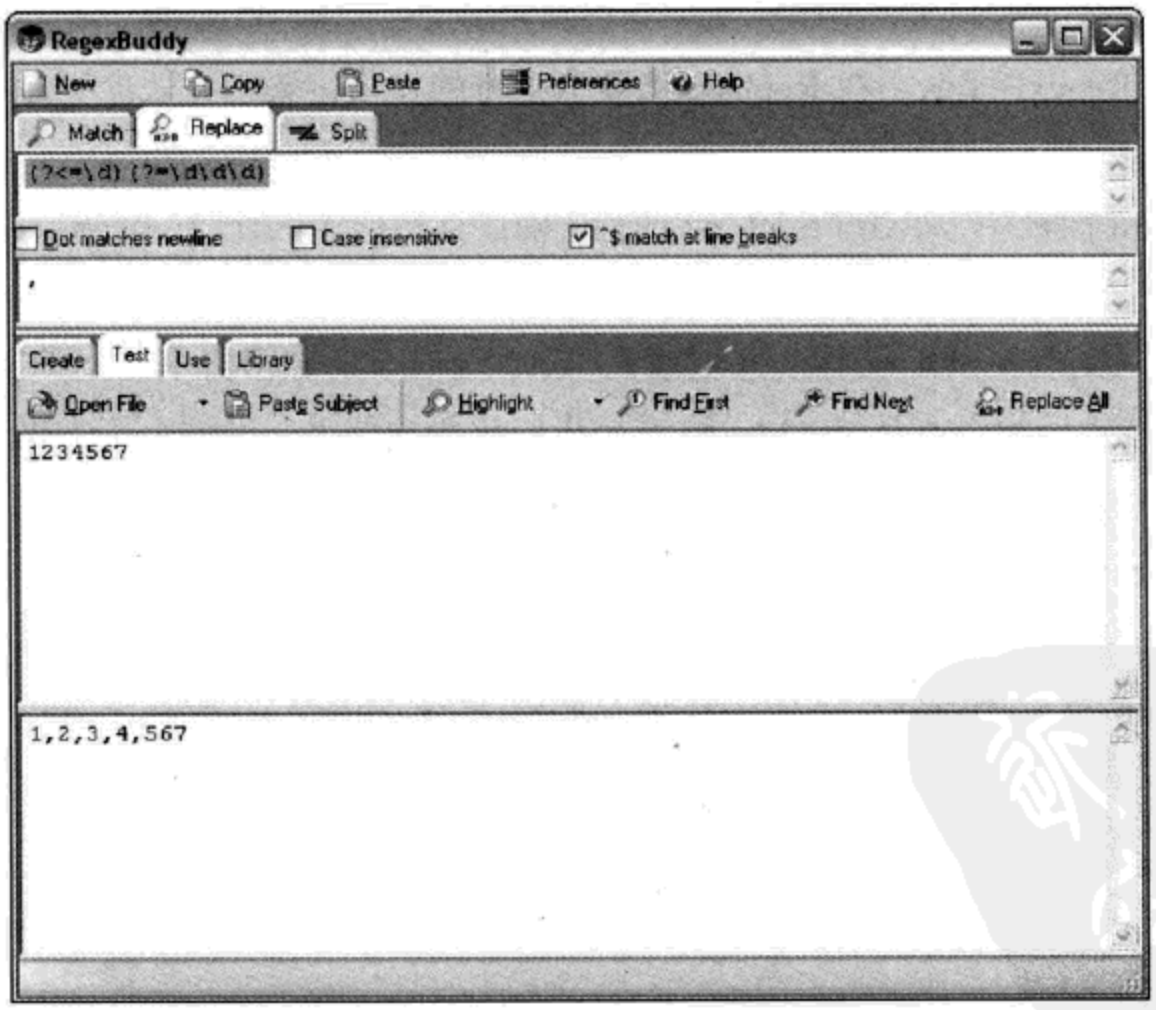


图 8-16

(8) 再把模式修改为 (?<=\d)(?=(\d\d\d)+$)。

(9) 单击 Replace All 图标，并观察 Test 选项卡下方窗格中的替换文本(如图 8-17 所

示)。这次是我们想要的 1,234,567 了。

(10) 对于特定的源数据，模式 (?<=\d)(?=(\d\d\d)+$) 也可能无效。想象一下想添加逗号的数值的最后一位数字后面是一个单独的字符(比如说一个句点字符)的情况。把测试文本修改为 Monthly sales figures are 1234567。

(11) 在 Replace 选项卡中把正则表达式修改为 (?<=\d)(?=(\d\d\d)+\W)。

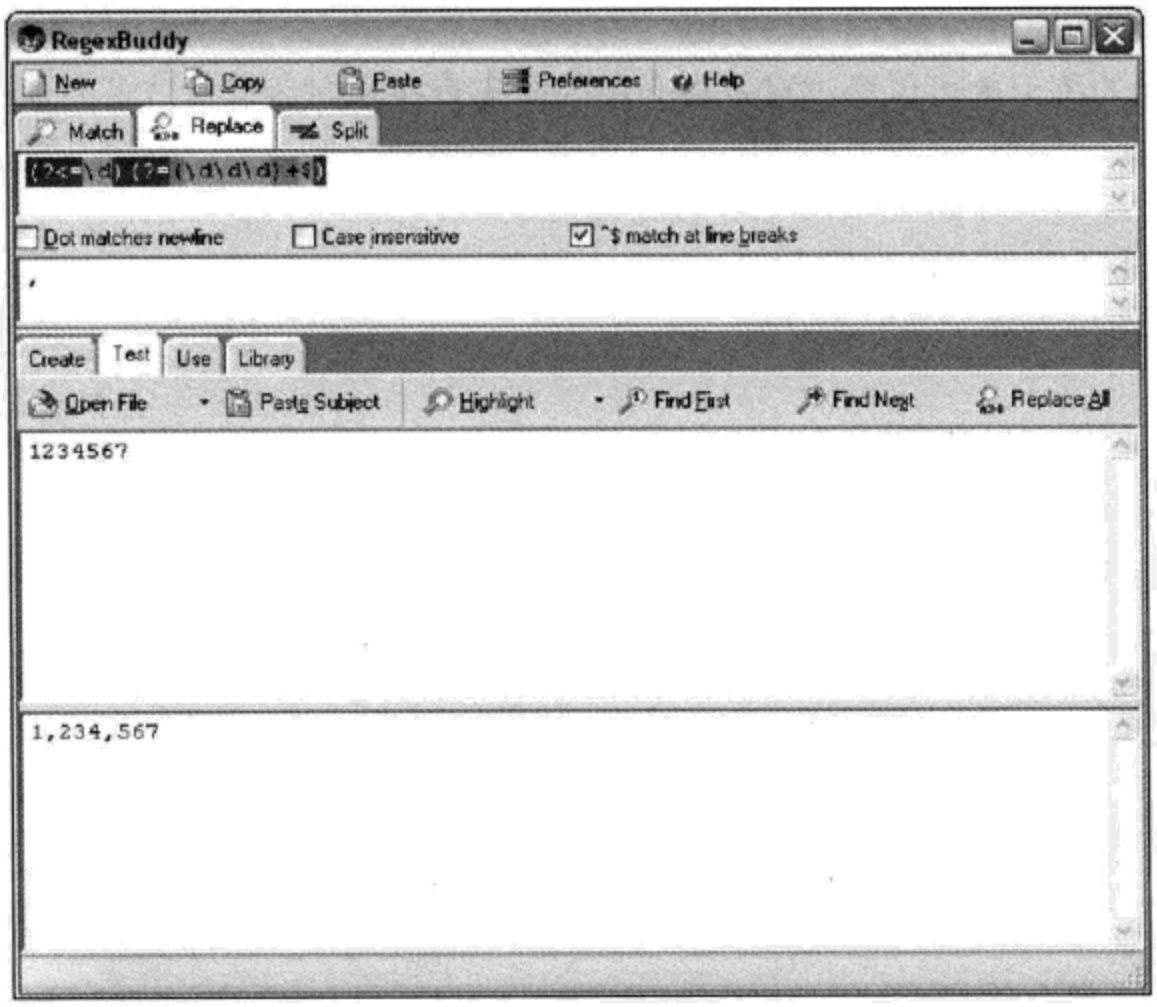


图　8-17

工作原理

模式 (?<=\d)(?=\d\d\d) 查找位于一个数字后面和三个数字前面的位置。在测试文本 1234 中，只有一个位置同时满足向后查找和向前查找的条件——数字 1 后面的位置。

当把测试文本修改为 1234567 后，模式 (?<=\d)(?=\d\d\d) 会匹配好几次。比如说，数字 2 后面的位置前面是一个数字而后面有三个数字。所以，该位置也同时满足向后查找和向前查找的条件。

我们需要通过对数字进行分组来排除不想插入逗号的位置。模式 (?<=\d)(?=(\d\d\d)+) 把向前查找中的数字分成了三个一组，以避免不想要的逗号，但如图 8-16 所示，结果仍然失败了。因为在数字 2 后面的位置之后，仍然有三个数字，所以该位置还是匹配，因而在该位置也插入了一个逗号(尽管不符合数字格式化规范)。

当把模式修改为 (?<=\d)(?=(\d\d\d)+$) 后，得到了想要的结果。这次，数字 2 后面的位置没有满足向前查找的限定条件。它后面有 5 个数字，与模式 (\d\d\d)+ 不匹配(在把模式修改为 (?<=\d)(?=(\d\d\d)+$) 之后，匹配的位置后面不仅要有一组或多组连续的三位数字，而且还必须匹配表示字符串结束位置的元字符 $。而修改模式之前 2 和 3 后面的位置匹配恰恰因为只需匹配一次连续的三位数字，而不用匹配字符串的结束位置。译者注)。

然而，数字 1 后面的位置仍然匹配。因为它的后面是 6 个数字，这与模式(\d\d\d)+ 匹配。类似地，数字 4 后面的位置也匹配，因为它的后面有三个数字，这与模式(\d\d\d)+也匹配。因此，在这两个匹配的位置上都插入了逗号。

8.6 练习

通过下面的练习题可以测试对本章介绍的向前查找和向后查找技术的理解情况：

1. 请编写一个模式，使其匹配一个或多个字母字符序列，且该序列后面必须有一个逗号字符。

2. 请使用向后查找和向前查找编写一个匹配单词 sheep 的模式。但不要在模式中使用表示词边界的元字符。

第 9 章

正则表达式的灵敏度和特殊性

本章讨论正则表达式的灵敏度和特殊性问题。灵敏度和特殊性分别涉及到在使用正则表达式过程中的两个基本目标，即尽可能地确保匹配所有想匹配的文本和尽可能地排除不想匹配的文本。

假设要操作某些数据，没有匹配到想要的数据显然意味着这部分任务尚未完成。如果对自己要操纵的数据和作用于该数据的正则表达式没有一个正确的评价，很可能一点都意识不到自己会忽略某些数据。至少，在管理人员或者客户责备和抱怨之前还意识不到。

相反，如果匹配并操作了不想修改的数据同样也会导致部分数据遭到破坏。这些问题是否严重取决于数据的用途、范围以及修改内容的重要程度。同样，意想不到的后果仍然会导致客户满意度的下降。所以，灵敏度和特殊性是必须认真对待的重大问题。

在本章中将学习以下内容：

- 什么是灵敏度和特殊性
- 如何评估在最大化灵敏度和特殊性时应该投入的时间和精力
- 如何使用正则表达式技术获得灵敏度和特殊性之间的最佳平衡
- 数据源的细节如何影响灵敏度和特殊性
- 对于 Star Training Company 的例子，如何在灵敏度和特殊性中取得更好的平衡

9.1 什么是灵敏度和特殊性

灵敏度是匹配模式的能力。特殊性是把模式选择的字符序列限定为所要选择的字符序列的能力。

灵敏度和特殊性这两个概念来源于像统计学和流行病学这样的学科中的量化标准。广义上讲，灵敏度可以用实际找到的匹配项中的正确匹配项数除以在匹配全部相关字符序列的情况下应该找到的匹配项数来度量。而特殊性则可以用实际找到的匹配项中的正确匹配项数除以找到的匹配项总数来表示。在使用正则表达式时，灵敏度越高，则表明找到的真正匹配项数量越接近要找的全部匹配项；而特殊性越高，则表明找到的匹配项中正确的匹

配项越多(区分灵敏度和特殊性的关键是要明确三个数据：“正确的个数”、“找到的个数”和“要找的个数”。根据上面的定义，灵敏度是指用“正确的”除以“要找的”，这个比例越高说明正则表达式越敏感——比如要找 100 个，实际找到了 150 个，而其中有 100 个是正确的，那么灵敏度就是 100%；而特殊性则是指用“正确的”除以“找到的”，这个比例越高说明正则表达式越特殊<即更具有针对性>——比如上面说找到的 150 个当中有 100 个是正确的，那么特殊性就是 66.67%。所以可以理解为灵敏度是从量的角度衡量匹配目标的完成情况，而特殊性则是从质的角度来衡量匹配目标的完成情况——译者注)。

这两种指标的定义听起来可能有些抽象，所以我们再通过下面的例子进一步理解什么是灵敏度和特殊性。

9.1.1 极端的灵敏度和糟糕的特殊性

假设要匹配字符序列 ABC。那么使用下面的模式便很容易达到 100% 的灵敏度：

```
.*
```

这个模式会选择零个或多个字母数字字符。

来看测试文件 ABitOfEverything.txt，其内容如下：

```
ABC123

DEF9FR

Mary had a little lamb.

var x = 234 / 1.56;

<html><body></body></html>

<book></book>

This is a random 58#Gooede garbled piece of 8983ju**nk but it is still selected.
```

这个文件内容很杂，而且并非都有用。但是，如果对这个文件使用正则表达式模式 .* ，则可以达到 100% 的灵敏度，因为唯一的那个字符序列 ABC 会被匹配。然而，测试文件中的所有其他文本也被选择了。图 9-1 显示了在 OpenOffice.org Writer 中完成这一匹配的结果。

通过这个有点极端的例子能够说明一个重要问题，即过于敏感的正则表达式模式可能会没有意义。当然，没人会只用 .* 这么一个模式，但是这个例子也说明当在处理一些微妙的问题时，必须重视正则表达式的使用效率。

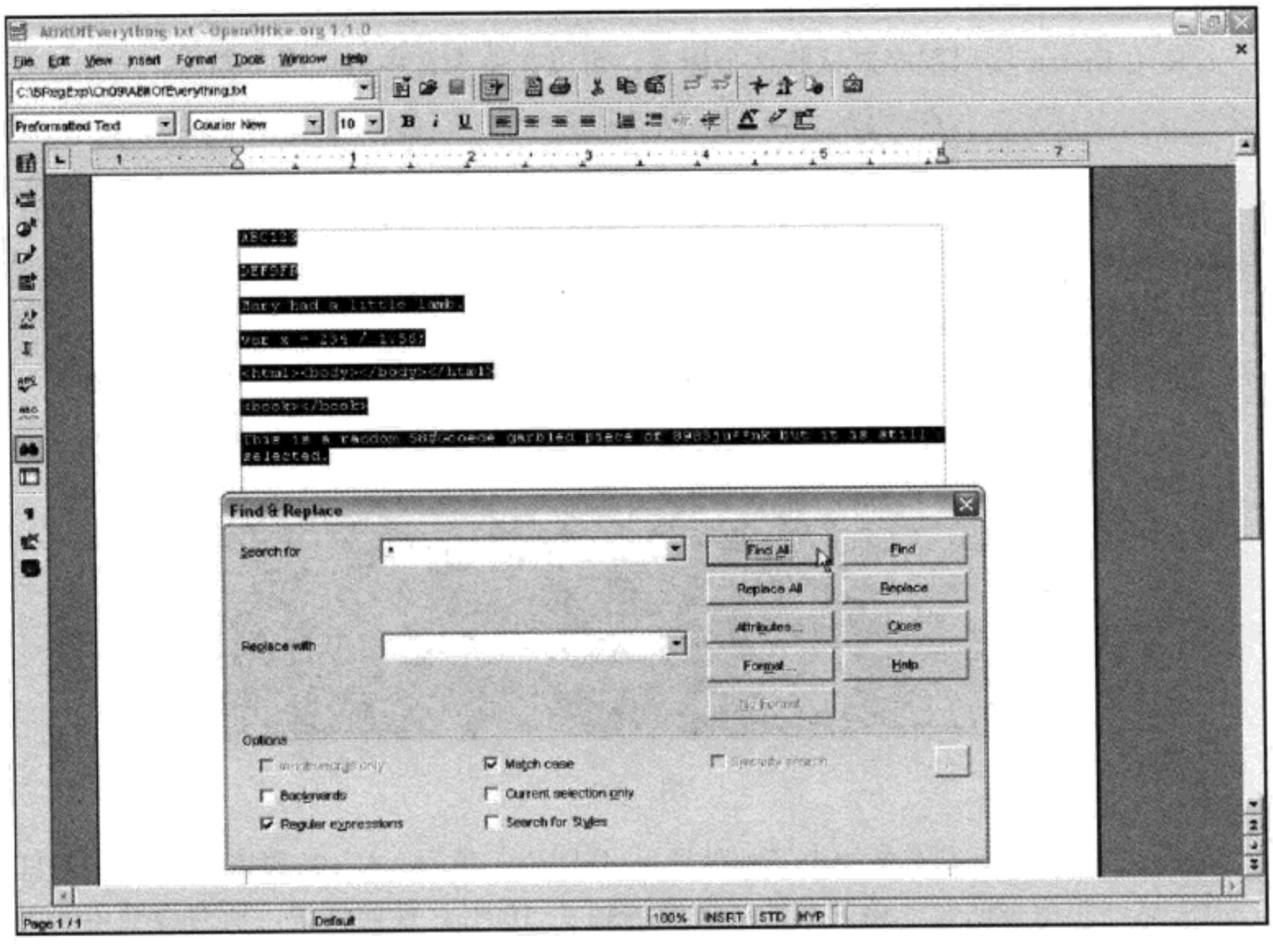


图　9-1

一个有用的正则表达式应该是在保持模式 .* 这样 100% (或者非常接近 100%)灵敏度的同时，具有更高的特殊性。

9.1.2　电子邮件地址的例子

假设需要从大量文档或电子邮件中搜索有效的电子邮件地址。我们以 EmailOrNot Email.txt 为例，其中包含的数据内容如下：

```
@Home
@ttitude
John@somewhere.invalid
Peter@example.org
Peter@example.info
John@Smith@example.com
20 @ $10 each
@@@ This is a comment @@@
Jane@example.net
Peter.Smith@example.net
```

EmailOrNotEmail.txt 中包含有效的和无效的电子邮件地址。

要查找里面所有的电子邮件地址可以使用下面的正则表达式：

```
.*@.*
```

若使用 findstr 实用程序来试验这个模式，可以在命令行中输入以下命令：

```
findstr /N /i .*@.* EmailOrNotEmail.txt
```

这表示要使用下面的正则表达式模式来搜索一个文件——EmailOrNotEmail.txt：

```
.*@.*
```

其中参数 /N 表示显示包含与模式匹配内容的行的行号。而 /i 表示以不区分大小写的方式进行搜索(在这个例子中它并不重要)。图 9-2 显示了运行指定命令后的结果。

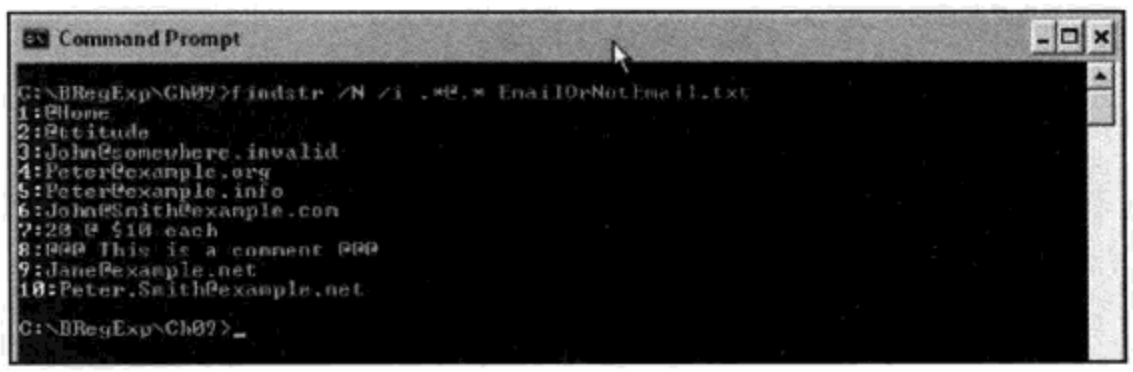


图 9-2

如图 9-2 所示，所有有效的电子邮件地址(分别位于第 4、5、9 和第 10 行)都被匹配了。对于这一组数据而言，该模式的灵敏度达到了 100%。换句话说，所有有效电子邮件地址都被匹配了。但同时也匹配了所有其他行中明显不是有效电子邮件地址的字符序列。所以必须要找一个更具有特殊性的模式来提高匹配的特殊性。

我们来分析一下有效电子邮件地址的构成。总的来说，一个电子邮件地址的结构应该像下面这样：

```
用户名@域名
```

要实现更好的匹配，必须要找到能够匹配用户名和域名的并且比前面的模式更具有特殊性的模式。

结构中的用户名可能是一个简单的字符序列，比如：

```
AWatt@XMML.com
```

或者，也可以包含一个句点字符，比如：

```
A.Watt@XMML.com
```

因此，模式需要包括匹配电子邮件地址的用户名部分包含句点的情况。而下面模式中使用的 \w+ 组件可以保证最少匹配一个单独的字母字符：

```
\w*\.?\w+
```

而组件 \w*\.? 则允许在强制的字母字符(序列)前面可以是零个或多个字母字符后跟一个可选的单个句点字符。

你可能不想匹配一个以句点字符开头的电子邮件地址，比如：

```
.Watt@XMML.com
```

所以，可以使用向后查找来限定只有当句点前面至少有一个字母字符时才能匹配。下面的模式只有在前面存在字母字符的情况下才会匹配句点字符：

```
\w*(?<=\w)\.?\w+
```

试一试：电子邮件地址

(1) 打开 PowerGrep，在 Search 文本区域中输入模式 \w*(?<=\w)\.?\w+@.* 。

(2) 在 Folder 文本框中输入文件夹名 C:\BRegExp\Ch09 。如果把下载的测试文件放在其他地方，请修改这里的路径。

(3) 在 File mask 文本框中输入文件名 EmailOrNotEmail.txt，并单击 Search 按钮。

(4) 在 Results 区域中观察结果。比较图 9-2 和图 9-3 中现在的匹配结果，很明显有几个字符序列不再匹配。

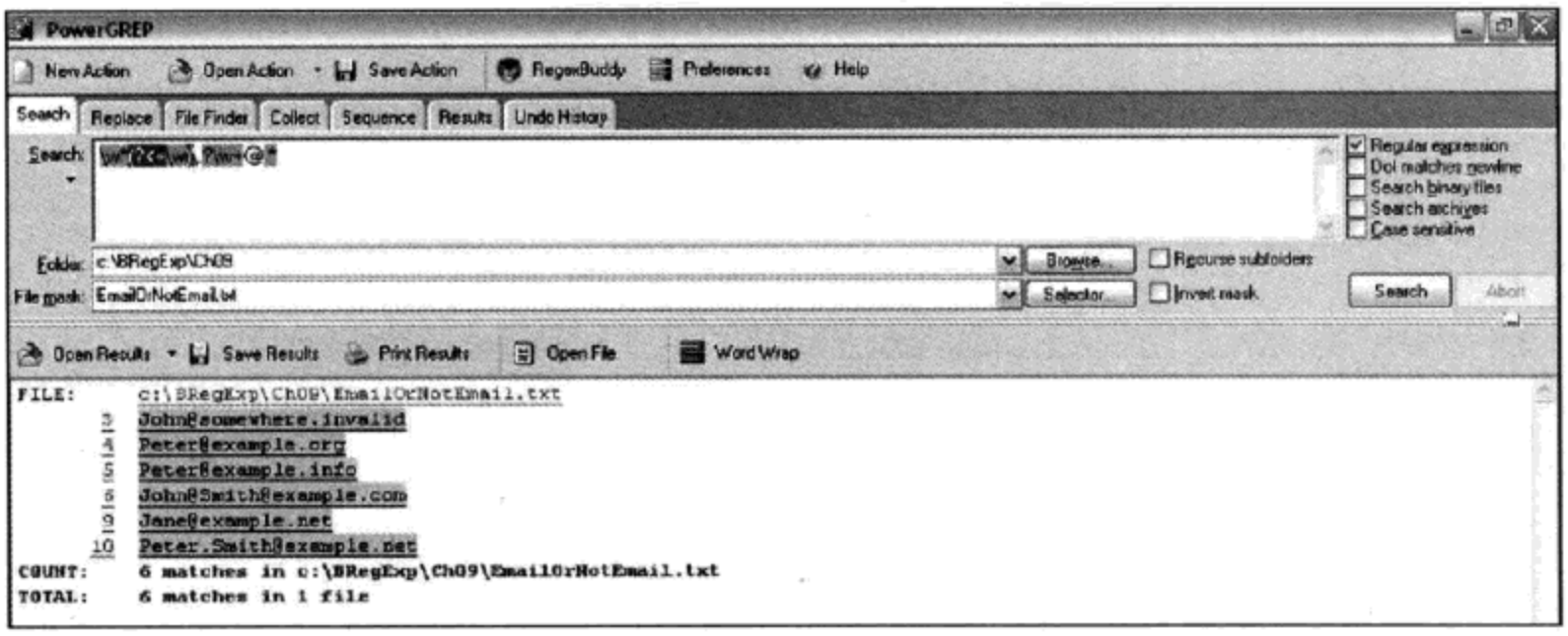


图　9-3

这是一次进步，而模式也更加具有特殊性。因为这次没有匹配第 1、2、7 和 8 行中不想要的字符序列。然而，第 3 行中的字符序列 John@somewhere.invalid，并不是一个有效的电子邮件地址。

可以通过将电子邮件地址的域名部分变得更加特殊化来排除这个不想要的匹配项。众所周知，所有的域名都是字母字符序列后跟一个句点字符，然后跟着三个(com、net、org 或 biz)或四个(info)字母字符。为了更方便让读者理解，我们先不考虑 example.co.uk 这样的域名。那么，下面的模式对应刚刚描述的结构：

```
\w+\.\w{3,4}
```

其中 \w+会匹配单个字符的域名(.com、.net 和 .org 域名允许使用单个字符)。而 \. 转义序列匹配单独一个句点字符，\w{3,4} 则匹配三个或四个字母字符。

将这个模式与前面使用的模式组合起来就是：

```
\w*(?<=\w)\.?\w+@\w+\.\w{3,4}
```

(5) 在 Search 文本区域中输入模式 **\w*(?<=\w)\.?\w+@\w+\.\w{3,4}**，并单击 Search 按钮。

(6) 观察结果。我们发现第 3 行中不想要的匹配项这次没有匹配。然而，在第 6 行中却出现了一个新的问题。第 6 行中的电子邮件地址中包含两个@字符，而这不是合法的邮件地址。

要解决这个问题，一种方法是在匹配第一个 @ 字符后使用向前查找，这样后面的@字符便不会出现了。如果我们继续假设在电子邮件地址中只允许使用字母字符，那么就可以通过向前查找来限定在第一次匹配的非字母字符或句点字符之前只有一个 @ 字符。

可以通过下面的模式实现这种限定：

```
\w*(?<=\w)\.?\w+@(?=[\w\.]+\W)\w+\.\w{3,4}
```

(7) 在 Search 文本区域中把模式修改为 \w*(?<=\w)\.?\w+@(?=[\w\.]+\W)\w+\.\w{3,4}，并单击 Search 按钮。

(8) 结果如图 9-4 所示。

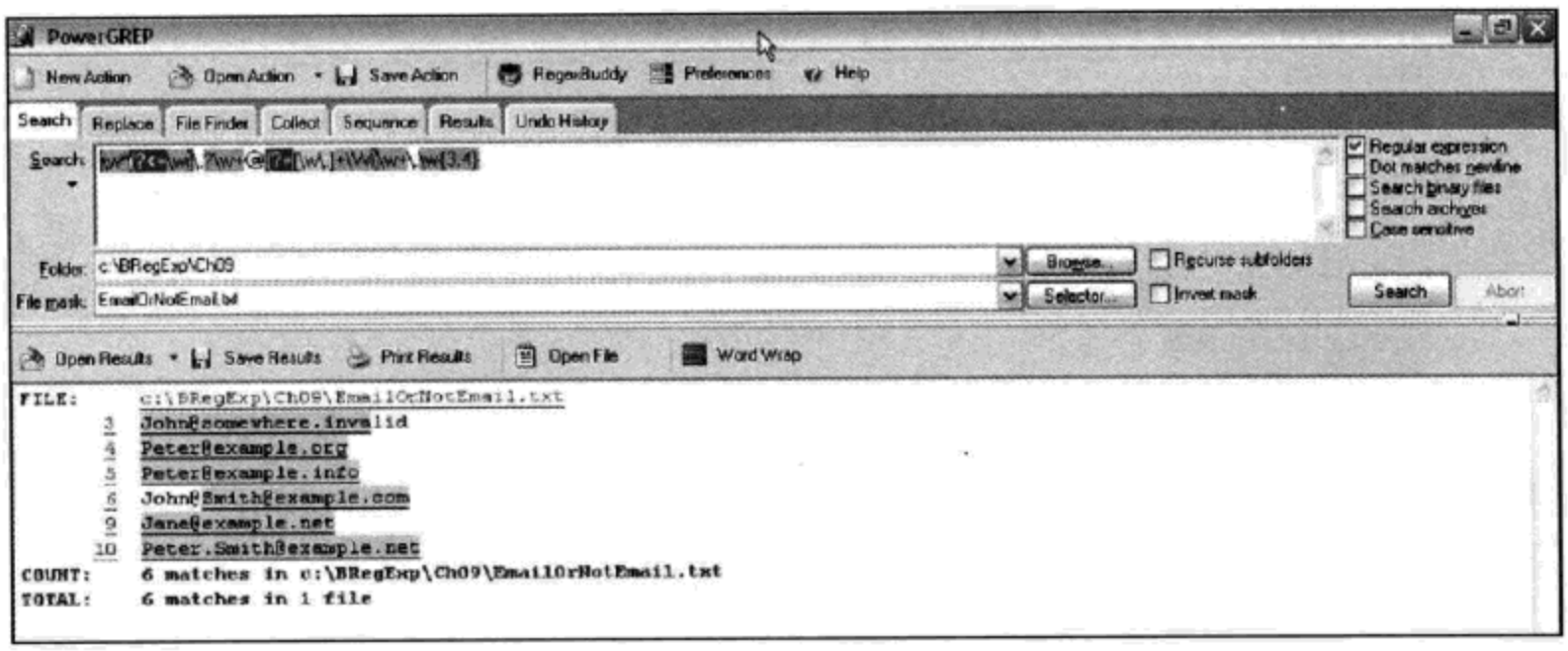


图 9-4

然而，向前查找并没有解决第 3 行和第 6 行中匹配错误的问题。我们还需要限定模式要匹配的是一行中的全部文本。换句话说，要添加 ^ 元字符指定一行的开始位置和 $ 元字符指定该行的结束位置。

(9) 在 Search 文本区域中把模式修改为^\w*(?<=\w)\.?\w+@(?=[\w\.]+\W) \w+\.\w{3,4}$，并单击 Search 按钮。

(10) 结果如图 9-5 所示。

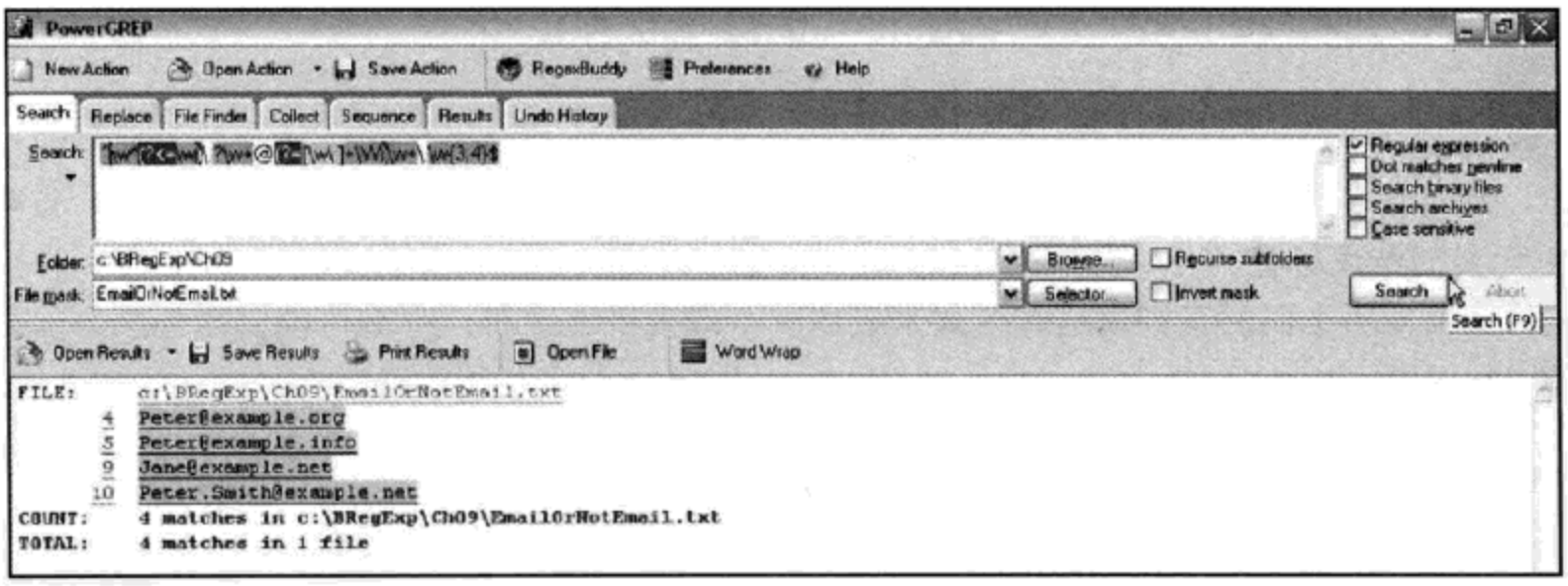


图　9-5

这次成功地排除了第 3 行和第 6 行中不想要的匹配项。至少在这个简单的测试数据文件中，我们获得了 100% 的灵敏度和 100% 特殊性。

灵敏度和特殊性这两个概念来源于量化研究的科学，比如统计学和流行病学。在这两种学科中，都是用数字来表示灵敏度和特殊性的，而且通常使用百分比来表示。因此，对于前面的例子来说，因为使用第一个正则表达式模式时找到了所有正确的电子邮件地址，所以灵敏度达到了 100%；而因为 10 个匹配结果中有 6 个(从不是有效的电子邮件地址的角度来说)是错误的匹配，所以特殊性仅有 40%。而到了该例子的最后，修改后正则表达式的特殊性最终上升到了 100%。

9.1.3　替换连字符的例子

这个例子反映出在没有仔细考虑正则表达式含义的情况下会导致的另一个问题。

假设需要把一组文本文档转换成 HTML/XHTML。本例主要针对要将一行连字符替换成 HTML/XHTML 中的<hr>以便满足创建水平线的需求。

HyphenTest.txt 是本例中要用到的文档：

```
something
not much
----
a little text
Fred
-------------
-Fred
```

第一次可能会将问题定义如下：

用字符序列 <hr> 替换任何连字符。

然而，这个定义太不严密了。比如说，根据这个定义第三行可能会被替换成下面这样：

```
<hr><hr><hr><hr>
```

所以，更精确的问题定义应该是：

用字符序列<hr>替换任意连续的连字符。

因为许多 Web 浏览器在解释空标签 <hr/> 时都会出现问题，所以我们假设你会省略 hr 元素的结束标签。

如果使用下面的正则表达式模式来表达一个或多个连字符，那么可能会因为两个原因而碰到问题：

```
-*
```

首先，并非所有的正则表达式引擎都能正确地解释这个模式。模式 -* 的含义是“匹配零个或多个连字符”，也就是说零个连字符也会匹配。因此，文本 Fred 应该匹配，而这并不是我们想要的。为什么 Fred 会匹配？因为它包含零个连字符。

OpenOffice.org Writer 根据你所表达的意思实现了 -* 模式，因为它只匹配至少一个连字符的情况，如图 9-6 所示(并没有因为每一行中都包含零个连字符而匹配每一行)。

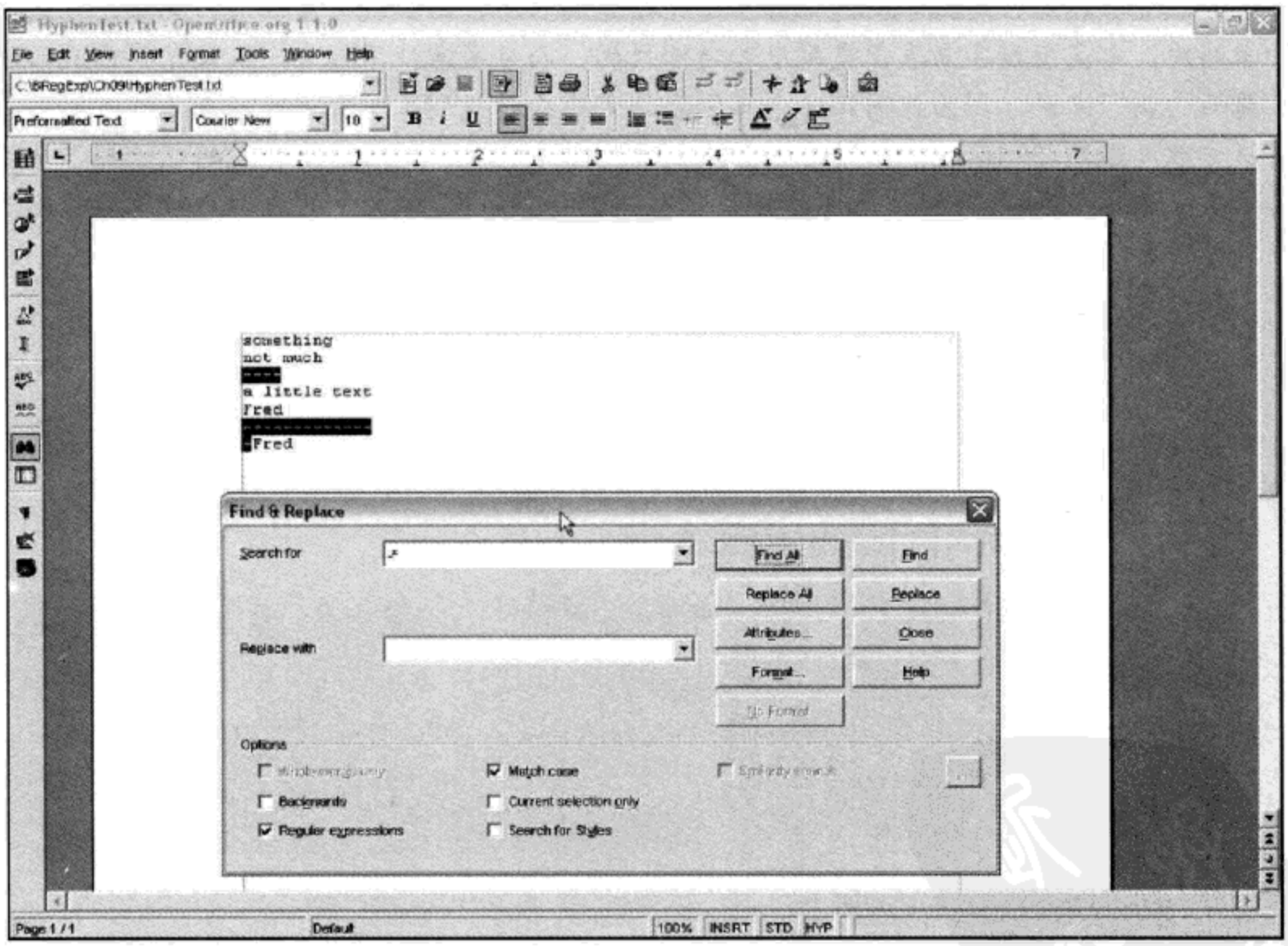


图 9-6

Komodo Regular Expression Toolkit 则可以正确地解释这个正则表达式，比如在查找文本 Fred 的匹配项时，如图 9-7 所示。

当然，模式 -+ 会更合适一些，因为你想至少匹配一个连字符。由于 * 限定符甚至会

匹配它限定的字符或元字符不存在的情况，所以有可能在某些情况下造成混乱。

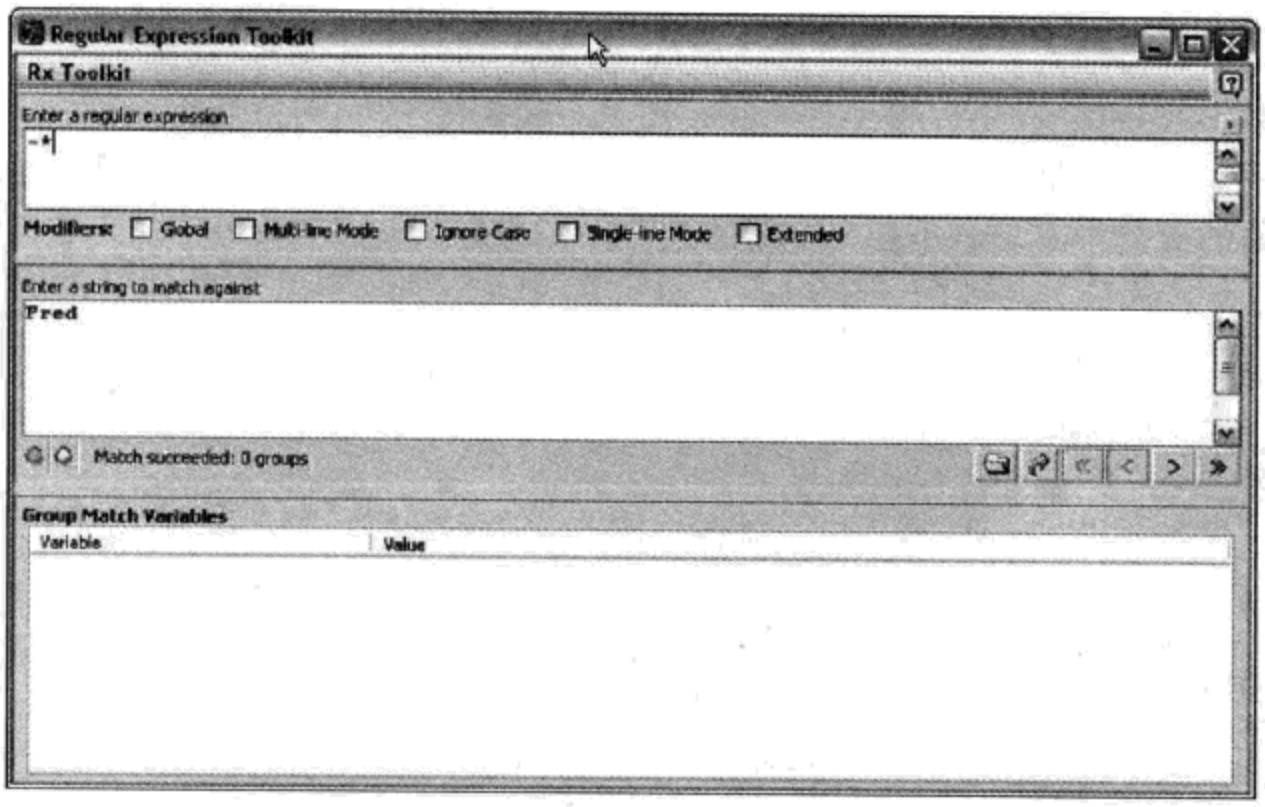


图　9-7

9.2　灵敏度和特殊性的平衡

灵敏度和特殊性就像是天平的两端一样。但是，在某些情况下要想同时获得 100% 的灵敏度和 100% 的特殊性恐怕是不现实的。有一些正则表达式只要具有不明确但却实用的特殊性就足够了。而此时的平衡，最终表现为如何判断为使正则表达式完成特定任务需要付出多少努力才是适当的。

灵敏度重要还是特殊性重要？答案是“视情况而定”。在许多情况下，当需要高灵敏度(理想的情况是 100%的灵敏度)时，同时也会需要高特殊性。而其他情况下，这两者中的一个可能不会显得很重要。本节讨论一下对灵敏度和特殊性的重要程度有影响的一些因素。

一般而言，重要程度取决于客户是谁。如果你使用正则表达式搜索只是出于自己的需要，那么即使丢下一到两个匹配项也不足为虑。但是，如果你是在为一个被别人接管后的公司替换文档中的每一个公司名称，那么如果灵敏度降到 100%之下就是一个严重的问题了。

9.3　元字符如何影响灵敏度和特殊性

一般来说，模式中使用的元字符越多，那么模式的特殊性也就越高。如果使用模式 cat，那么无论相应的字符序列是猫(即单词 cat。译者注)，还是构成 cathode 或 caterpillar 单词的一部分都会被匹配。

如果给这个模式加上元字符，比如 \b 词边界，就会使模式 cat 的意义更加明确。模式 \bcat\b 将只匹配单词 cat(单数)。

在使用这样的具体模式时，就要警惕可能会降低灵敏度。比如，模式 \bcat\b 只能匹配 cat，却不能匹配 cats。所以，如果你想找到文档中所有提到猫的字符序列(即不论单数、复数，还是所有格形式。译者注)，那么模式 \bcat\b 可能并不是最佳选择。因为你可能也想查找复数形式 cats，和所有格形式 cat's。模式 \bcat'?s?'?\b 则会匹配 cat、cats、cats' (复数所有格)和 cat's(单数所有格)，但是也有可能匹配 cat' ——而这很可能不是想要的匹配项。假设你的数据中本来就不包含字符序列 cat'，那么 \bcat'?s?'?\b 就足够用了。但是，如果包含 cat'，而你只想匹配 cat、cats、cat's 和 cats'，那么就需要其他更具体的模式了。可以简单地选择下面的模式：

```
(cat|cats|cat's|cats')
```

另一种方案是：

```
ca(t|ts|t's|ts)
```

而类似的问题也存在于其他要匹配的单词或字符序列中。

9.3.1 灵敏度、特殊性和位置字符

第 6 章中介绍的位置字符在许多情况下都会同时影响到灵敏度和特殊性。

在下面的例子中，刚开始对问题定义的描述可能是这样的：

以不区分大小写的方式匹配所有字符序列 t、h 和 e 。

相应的模式会在下面的文本中找到两个匹配项：

```
Paris in the the spring.
```

而在下面的文本中只能匹配一次：

```
The spring has sprung.
```

然而，假设把问题定义修改如下：

匹配一个字符串的开始位置，然后以不区分大小写的方式匹配所有字符序列 t、h 和 e。

现在，模式 ^the 将不再匹配第一个测试文本中的两个 the，但仍然会匹配第二个测试文本中开头的 The。添加一个或多个位置字符所带来的影响还取决于模式所要匹配的数据。

9.3.2 灵敏度、特殊性和模式

当为执行的正则表达式指定以不区分大小写或者区分大小写的模式进行匹配时，就会影响到将要返回的匹配项。继续以前面的例子为例，如果以区分大小写的模式应用正则表达式^the，那么两个测试文本中的任何一个都不会有匹配项。在第二个测试文本中，^元字符匹配字符串开始处的位置，但是正则表达式中小写的 t 却不会匹配测试文本中大写的 T。

同理，也可以指定句点元字符(它匹配很大范围内的字符)匹配或者不匹配换行符。

9.3.3　灵敏度、特殊性和向前、向后查找

当在现有的正则表达式中加入向前查找或向后查找时，其灵敏度与特殊性的变化是不确定的。

如果能够仔细地编写向后查找模式，应该不会降低灵敏度。然而，如果在向后查找的模式中犯了一个错误，则会导致无法匹配想要匹配的内容，最后降低灵敏度。假设要查找有关 Anne Smith 的信息。下面的模式会在 Anne 拼写正确并且后跟一个空格符的情况下匹配：

```
(?<=Anne )Smith
```

但是，如果文档中的某处把 Anne 拼写成了 Ann，那么就会丢掉该匹配项——因为它不再与模式 (?<=Anne)Smith 匹配。

同样地，如果人的名字在数据中的某处被写成了 A. Smith，也不会被匹配。所以要知道匹配项是否是想要的，就必须对数据进行更详细的分析。因为字符序列 A. Smith 可能会指代要匹配的人——Anne Smith，但是也可能会指代另外一个人，如 Adam Smith。

同样地，向前查找也可能会降低灵敏度。比如说，假设想匹配所有字符序列 John，那么下面的模式将会匹配一个词边界，然后是想要的字符序列 John，接着会再检查后跟的字符是不是一个空格符：

```
\bJohn(?= )
```

但是，如果测试文本如下所示，那么由于这个向前查找过于特殊则会导致可能匹配的内容匹配失败：

```
I went with John, and Mary on a trip.
```

把这个向前查找修改为 (?=\b) 或 (?=\W) 将会避免由于意料之外的逗号导致的问题。

9.3.4　正则表达式应该做多少

在本书前面的多数例子中，都使用了支持正则表达式功能的工具来演示正则表达式的用途。这些工具用来演示正则表达式是比较合适的，但是当你作为一名开发人员要使用正则表达式时，一般都要在 Java、JavaScript、VB.NET 等语言的代码中使用正则表达式，或者也可能会对取自关系型数据库的数据应用正则表达式。那么，正则表达式应该承担多少数据操作的任务量呢？或者说你能在多大程度上假设其他代码或在数据库中完成的错误检测功能能够保证数据的可靠性呢？

例如，假设有一组包含 IP 地址的 HTML 文档，要修改 IP 地址显示的样式。如果在开始时，IP 地址被嵌套在 HTML 中 b 元素的开始和结束标签中，如下所示：

```
<b>1.12.123.234</b>
```

那么应使用什么样的模式来查找这样的 IP 地址呢？是否可以确定得到的数据都是正确有效的(包括没有超过 255 的值)，或者说是否需要使用一个更复杂的正则表达式来保证只匹配正确有效的 IP 地址呢？

如果假设其中的 IP 地址都正确有效，或者说都已经通过其他代码的检测，那么就可以使用下面相当简单的模式：

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

这个模式也会匹配不是 IP 地址的字符序列，比如：

```
<b>1234.2345.5678.9999999</b>
```

如果能保证数据中不会包含类似的无效内容，这个简单模式就足够了。不用花多少功夫，就可以将这个模式改编成其匹配句点字符前面或后面只有一到三个数字的模式：

```
<b>([0-9]{1,3}(\.[0-9]{1,3})</b>
```

虽然像下面这样不适当的字符序列仍然可能被匹配，但是通过排除多于三个数字的错误匹配，至少能够提高模式的一些特殊性：

```
<b>999.256.789.1</b>
```

然而，如果不能保证数据中的 IP 地址都是正确有效的，可能就要开发一个更长，而且更复杂的模式了。另一种可能是，也许因为某种特定的目的而不必在意假定的 IP 地址是否有效。如果是这种情况的话，那么最简单的正则表达式就是适当的选择。

9.4 了解数据、灵敏度和特殊性

影响灵敏度和特殊性的一个关键性问题就是对要应用正则表达式的数据的了解程度。当然，对使用的语言或工具所支持的正则表达式语法及技术的理解也是非常重要的因素。

但是，如果没有真正理解要处理的数据，那么即便是语法正确的正则表达式也可能因为低下的灵敏度或特殊性而带来不想要的结果。

9.4.1 缩写词

缩写词对一个正则表达式的灵敏度具有极大的潜在威胁。例如，称谓 Dr(没有句点字符)和 Dr.(有一个句点字符)经常作为 Doctor 的缩写词出现。在某些环境下，你可能会十分肯定数据源中只包含其中一种形式。而如果数据中存在所有三种形式，那么就必须使用像下面这样的模式来避免错过某些想要的匹配：

```
(Doctor|Dr.|Dr)
```

类似的问题也会在处理有关资格信息的数据时出现。例如，如果要匹配的是哲学博士学位(Doctor of Philosophy degree)，而这个学位经常会被写成 PhD(没有空格符或句点字符)、Ph.D.(包含两个句点字符)或 Ph. D.(包含一个空格符和两个句点字符)。

要匹配刚才提到的这些可能的缩写词，使用下面的模式可以满足需求：

```
Ph\.? ?D\.?
```

这个模式中包含两个带有 ? 限定符的 \.，它们匹配任何一个可能在一些选项中出现的

可选的句点字符。而根据获得的学位不同，也有可能出现 D.Phil.(包含两个句点字符)和 DPhil(没有句点字符)的形式。如果也要匹配这两种缩写词，则需要像下面这样的模式：

```
(Ph\.? ?D\.?|D\.?Phil\.?)
```

9.4.2　来自其他语言的字符

本书主要介绍在英语(包括美式英语和英式英语)环境中如何使用正则表达式。然而，随着贸易全球化进程的加快，文档中出现其他语言字符或文字的情况也越来越常见，但其中大部分内容往往仍然是英文。

在加拿大，有很多官方文件都使用法文。因此，经常会在其中看到重音符。

在英文文档中，单词的写法不同也会导致差别。例如下面的测试文本：

```
"Nostalgia is not what it used to be." That is my favorite cliche.
```

也可以等价地写成：

```
"Nostalgia is not what it used to be." That is my favorite cliché.
```

下面这一句中结束句子的句点字符之前，有一个重音字符é。要匹配以上两种形式，则需要使用下面的模式：

```
clich(e|é)
```

当 HTML 文档中包含外国字符时，还会导致其他问题。比如在测试文档 EAcute.html 中，使用 é 符号来代替直接量字符：

```
<h2>"Nostalgia is not what it used to be." That is my favorite clich&eacute;.</h2>
```

结果如图 9-8 所示，网页中显示的仍然是正确的字符。

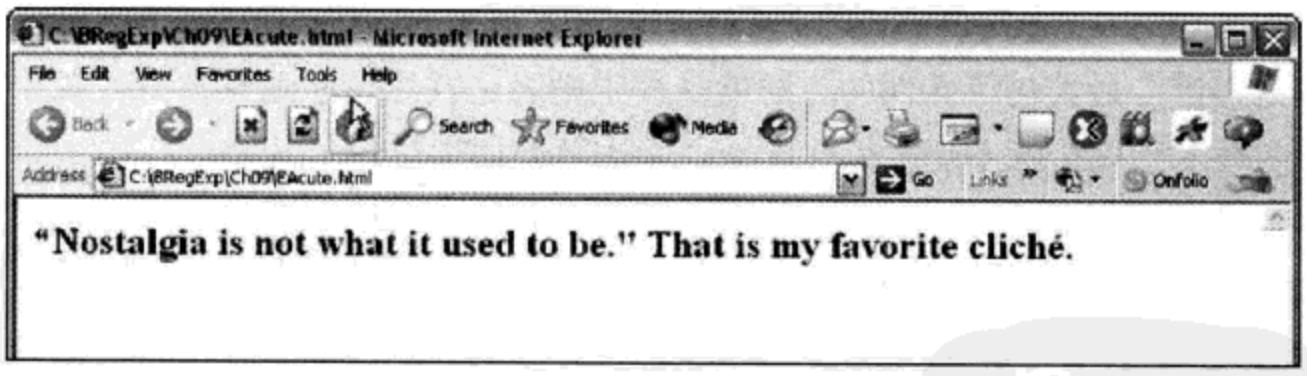


图　9-8

在某些文档中可以使用 Unicode 编码值来表示非英文字符，所以在试图匹配这些字符时记住还要多考虑一下。

9.4.3　名字

由于人类奔走于各地的频率增大(比如，由于工作原因)，一些起源于非罗马语言的名字也经常会成为人力资源以及其他数据中的一部分。比如说，有时会拼写成 Saurav 的印

度人的名字，也可以拼写成 Saurabh 和不太常见的 Surav。那么，模式 Saurav 就不会匹配后两种拼写形式，而有时我们需要匹配这个名字的所有拼写形式。如果要匹配所有拼写形式，则需要使用下面的模式：

```
Sa?ura(v|bh)
```

对于其他非英语语言也要考虑到这一问题。俄罗斯名字彼得 (Peter)有时会被翻译成 Pyotr，也可能会被拼写为 Petr 或 Pëtr，甚至仍然会被翻译成 Peter，有时也可能需要匹配所有这些拼写形成。要匹配所有这些可能的名字形式，可以使用像下面这样的模式：

```
P(yo|e|ë)te?r
```

某些欧洲人的姓也存在多种拼写方式。比如，姓 Van Nistelrooy(带有空格符)也可以拼写成 Van Nistelrooij 或 VanNistelrooy(不带空格符)。因此，需要使用下面的模式来匹配这三种拼写变体：

```
Van *Nistelroo(ij|y)
```

当然，由于这些姓中的首字母有时候会被拼成小写，比如 van 中的 v，所以下面的模式在这种情况下会更具灵敏度：

```
[vV]an *Nistelroo(ij|y)
```

9.4.4 灵敏度及如何最大化

要实现最大化的灵敏度，必须知道要匹配的字符序列的所有变体。

每当向模式中添加一个组件时都会使模式更具特殊性。对于给定要处理的数据，要认真考虑添加组件会不会导致本来匹配的内容变得不再匹配。

9.4.5 特殊性及如何最大化

从理论上讲，最大化特殊性的方法就是使正则表达式尽可能地具体或者说更具有针对性。有很多技术可以排除不想要的匹配项，而本章前面也讨论了其中一些相关的技术。

在尝试最大化特殊性时，最重要的是要仔细考虑所有想匹配的情况，并构造适当的模式从而从结果中排除那些不想要的字符序列。达成高度特殊性还涉及到对正则表达式语法的理解、对可用技术的理解以及对于这些技术如何影响所操纵数据的理解。

9.5 重新分析 Star Training Company 的例子

在第 1 章，我们看到了一个新员工在虚拟公司 Star Training Company 所遇到的挑战。在学习了第 2 章到第 7 章中的技术后，能够做到很好地避免第 1 章中使用简单的查找替换所导致的问题。

为方便起见，我们把测试文件 StarOriginal.txt 的内容重新展示一次：

```
Star Training Company

Starting from May 1st Star Training Company is offering a startling special offer
to our regular customers - a 20% discount when 4 or more staff attend a single Star
Training Company course.

In addition, each quarter our star customer will receive a voucher for a free
holiday away from the pressures of the office. Staring at a computer screen all day
might be replaced by starfish and swimming in the Seychelles.

Once this offer has started and you hear about other Star Training customers
enjoying their free holiday you might feel left out. Don't be left on the outside
staring in. Start right now building your points to allow you to start out on your
very own Star Training holiday.

Reach for the star. Training is valuable in its own right but the possibility of a
free holiday adds a startling new dimension to the benefits of Star Training
training.

Don't stare at that computer screen any longer. Start now with Star. Training is
crucial to your company's wellbeing. Think Star.
```

相应的问题定义可以描述如下：

匹配所有引用 Star Training Company 的字符序列 S、t、a 和 r。将这个字符序列替换成另一个字符序列 M、o、o 和 n。

目标是将所有对虚构的 Star Training Company 的引用替换为对同样虚构的 Moon Training Company 的引用。

当在实践中面对这样的任务时，有必要通过文本编辑器或者带搜索功能的文字处理程序事先查看一些例子文档。在这些工具中，可以输入一个模式查找可能相关的字符序列。本例中，可以使用简单的直接量模式 star(全部小写)，并以不区分大小写的形式进行搜索：

试一试：用 Moon 替换 Star

(1) 在 penOffice.org Writer 中打开文件 StarOriginal.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 复选框，但保持 Match case 复选框的未选中状态。因为我们想要以不区分大小写的方式搜索指定的模式。

(4) 在 Search for 文本框中输入模式 star，并单击 Find All 按钮。

(5) 观察结果，如图 9-9 所示。注意其中所有引用 Star Training Company 的字符序列 star。

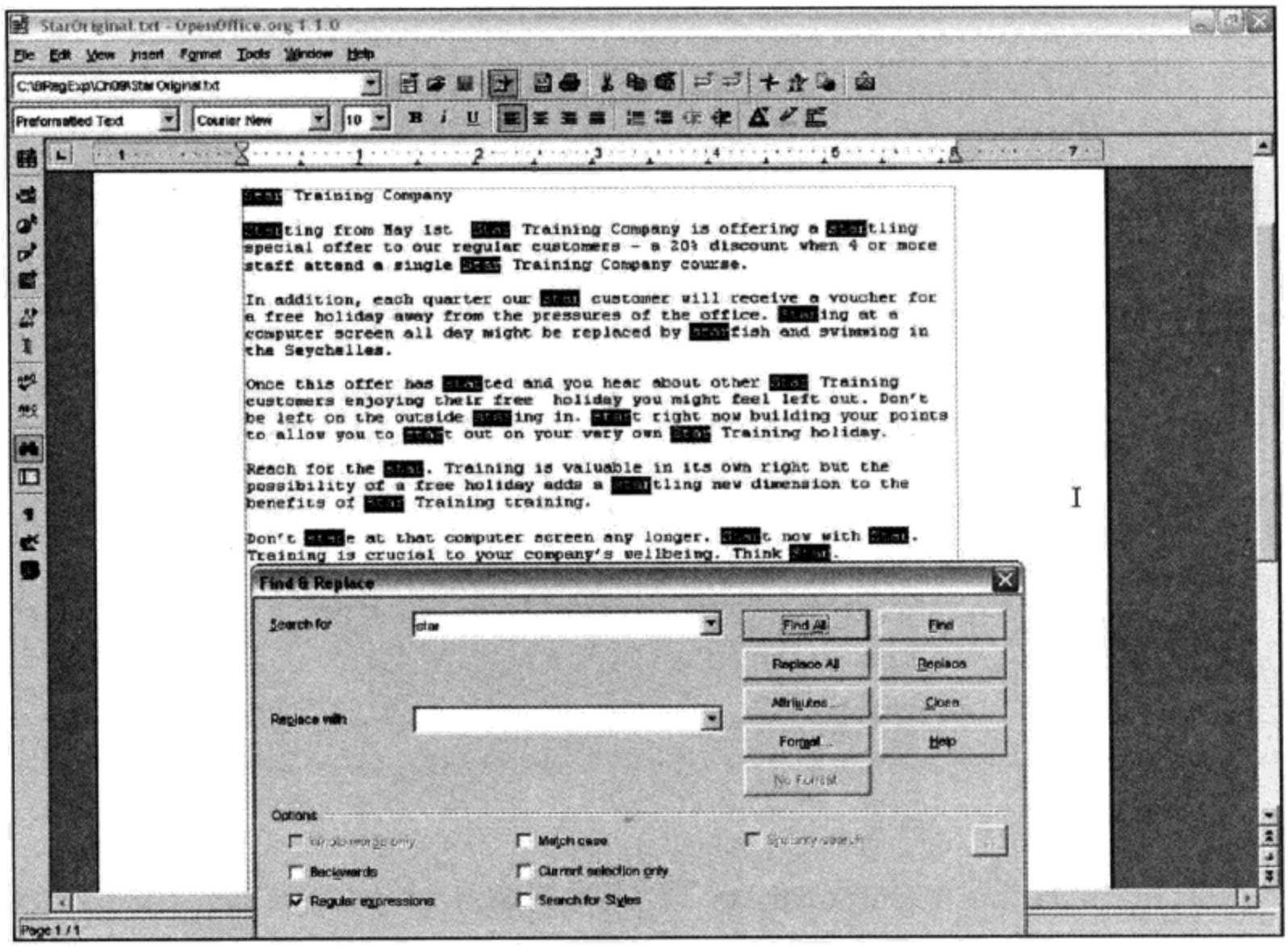


图 9-9

工作原理

在前面的例子中，匹配非常简单，而且也匹配了组成模式的所有直接量字符序列。

对于搜索一个文档来说，使用 OpenOffice.org Writer 是很方便的。然而，当需要处理大量文档时，可以考虑使用 PowerGrep。PowerGrep 可以让你同时查找多个文档中的匹配项，并突出显示每个匹配项。这样就会因为减少了要处理的数据而节省大量时间。

我们来花点时间列举一下你想匹配的字符序列。你想匹配的是位于下面字符序列中的 star：

```
Star Training
Star.
```

(列举想要匹配的字符序列的所有变体，可以避免遗漏，进而提高正则表达式的灵敏度。译者注)。

而你想避免匹配下面字符序列中的 star：

```
Starting
startling
star customer
Staring
starfish
```

```
started
Start right
start out
star.
startling
stare
Start now
```

(列举出不想匹配的类似字符序列，可以使要排除的匹配项更加清楚，进而提高正则表达式的特殊性。译者注)。

这里并没列举出想要的和不想要的匹配项。但是，当希望尽可能达到接近 100% 的灵敏度和 100% 的特殊性时，列出类似这样的列表是很有意义的。

将字符序列分为想要匹配和不想匹配的两个列表，对于将来计算出某个模式的灵敏度和特殊性是最有用的。

如果你认为使用向前查找可以解决问题，那么可以尝试使用下面的模式来取得所有想要的匹配项：

```
Star(?= Training)
```

然而，当检查想要的匹配项列表时，就会立即看到前面的模式不会匹配句子 Think Star 中的 Star。那是一个后跟句点字符的 Star。

下面的模式提供了包含两个向前查找的交替选项，适合在测试文本中要找的所有匹配项：

```
Star((?= Training)|(?=\.))
```

这样，根据测试文本我们可以判断该模式具有 100% 的灵敏度。图 9-10 显示了使用前面的模式测试字符序列 Star 的结果。

这里一个明智的做法就是始终要考虑你所看到的测试数据中并没有完全包含你想要的、合适的或可能的字符序列。本章后面有一道练习题就是要求读者修改前面的模式，以便使它也匹配与我们要求的 Star 的用法相关的其他可能的实例。

一般来说，想要匹配的模式不同于那些不想匹配的模式。因此，要保证模式不会匹配任何不想要的字符序列通常比较容易，但也有一个例外：你想匹配五个字符的序列 Star.(首字母 S 大写)，但不想匹配五个字符的序列 star.(首字符 s 小写)。

如果在区分大小写的情况下使用前面的模式，就没有问题。这时，不想要的字符序列 star. 不会匹配。然而，如果是在不区分大小写的情况下，这个不想要的字符序列 star. 将会匹配，这就降低了所选模式的特殊性。

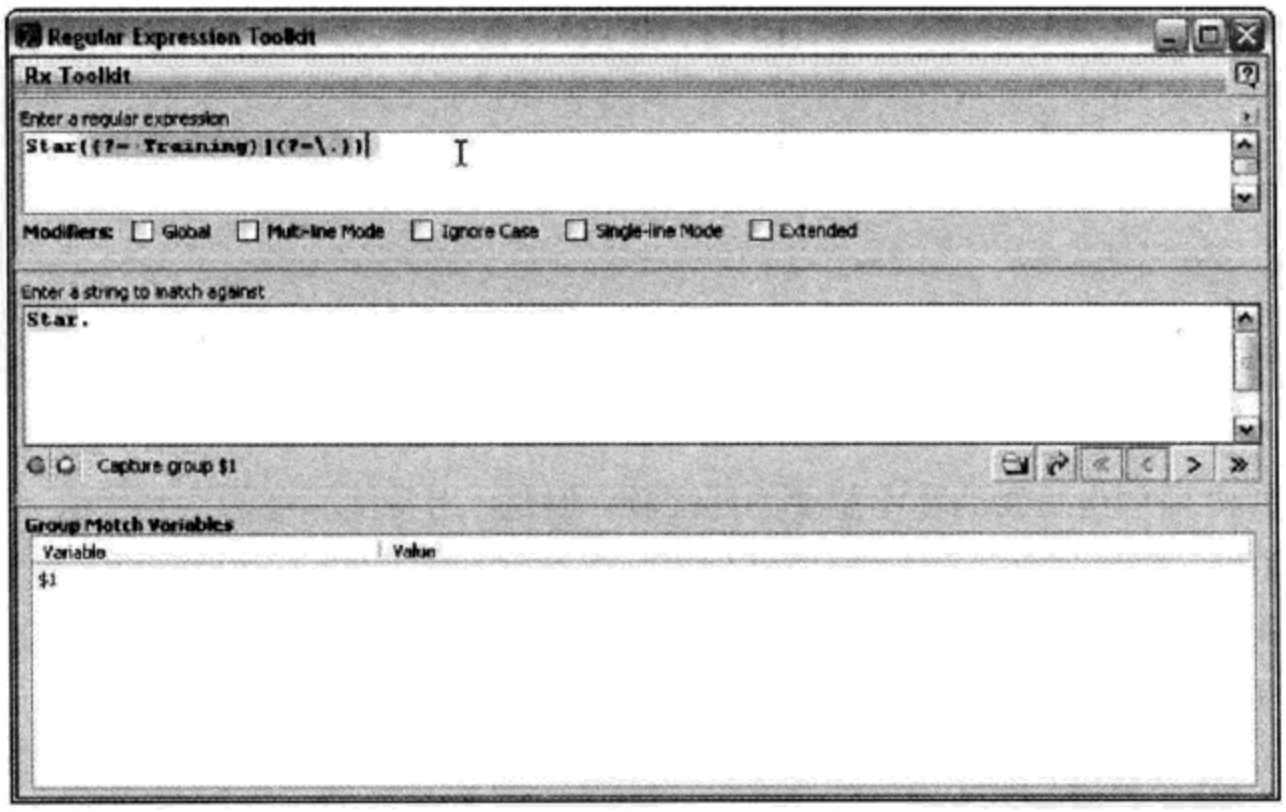


图 9-10

9.6 练习

通过下面的练习可以测试读者对本章所讲内容的理解情况：

1. 修改模式 ^\w*(?<=\w)\.?\w+@(?=[\w\.]+\W)\w+\.\w{3,4}$，这个模式是本章前面介绍的用于匹配电子邮件地址的一个模式，请修改这个模式让它只匹配域名为 .com、.net 或 .org 的电子邮件地址。

2. 修改 Star Training Company 例子中的模式，使其匹配字符序列 Star 并且使它也会在下面的数据中被匹配：

```
What do you think of Star?
The best training company is Star!
```

第10章

说明和调试正则表达式

在阅读本书的过程中，多数人会发现正则表达式很难编写。不仅如此，正则表达式也很难读懂——无论是别人编写的，还是你自己在几天前刚刚编写的。当把编好的正则表达式在项目中使用一段时间后，你又不得不面对第三个事实：正则表达式很难维护。本章的目的就是设法降低以上三大问题的影响。

一个重要的、不该忘记的基本事实是正则表达式从来不会脱离数据而孤立存在。正则表达式总是与要操作的数据密切相关，无论数据是简单还是冗长；它还与所用工具或编程语言的环境密切相关。此外，编写正则表达式的开发人员也会带有一定的目的性，比如某种复杂或微妙的商业目的。

使用正则表达式时遇到的问题，都可以简单地归结为无法编写出能够表达想要的匹配特征的模式。理想的情况下，当把这本书看完后，这种问题就会越来越少见。

在本章中将学习以下内容：

- 如何说明正则表达式
- 如何把握要操纵的数据
- 如何为正则表达式创建测试用例
- 如何调试正则表达式

10.1 说明正则表达式

任何意义重大的编程项目都会从良好的说明注释中受益。良好的说明注释不仅使项目各方面的意图表达得更加清晰明确，而且也有助于将来对代码进行深层开发。对于语法简洁且隐秘的正则表达式，如果能够认真说明创建时使用的手段，以及对正则表达式中每个组件的期望用途，一定会更加有意义。

在许多环境下，只是在很小的范围内使用正则表达式，此时不添加说明也无所谓。有时候，没有说明文档倒是一个明智的选择。例如，在 Microsoft Word 或 OpenOffice.org Writer 中使用正则表达式时，给一个正则表达式添加说明就没有必要。如果只是想查找或替换一个文档中的某个字符序列，则没有必要添加正式的文档。

然而，在一些意义重大的任务或项目中，创建正式的文档就非常有益，它有助于克服含糊不清的表达，使任务的方方面面都清晰而明确。

10.1.1 说明问题定义

问题定义是记录你在设计正则表达式过程中想法的关键所在。我们在前面几章中介绍过，初次对问题进行定义时无需精确。如果是比较复杂的问题，最好能记录下你不想匹配的问题定义，以便几个月后再看这些代码时，能对当初设计正则表达式模式时的需求有更清晰的回忆。

初次对问题进行定义时，可能会不够具体或者不能根据其描述马上定义一个所需的匹配。

在第 1 章的 Star Training Company 示例中，初次的问题定义可能会像下面这样：

用 Moon 替换 Star。

如果根据这个问题定义进行搜索和替换，会出现大量不适当的修改。如果是在没有备份的情况下对大量文档进行这种不适当的修改，那么就需要投入相当多的时间来纠正由于使用这个粗略的直接量正则表达式而导致的问题。

随着对数据理解的加深，可以进一步完善问题定义。假设有下面的文本数据：

```
Star Training Company ...
... I highly recommend Star.
Why not accept this special offer from Star?
... recent course with Star - which was great!
```

你会发现可以用不同的方式来表达要匹配的内容。但只有理解数据后，才能构建出能匹配(然后替换)所有想要内容的模式。

另一方面，可能会存在内容相似的文本，对这些文本需要保持原状：

```
The trainer was good - a real star!
The training was excellent - star training.
Star performer among the trainers ...
```

同样，如果没有花时间理解不想要的可能匹配项，最终会导致对文档进行不适当的修改。

10.1.2 为代码添加注释

为代码添加注释是一项基本任务。注释要尽可能有意义，且要尽量在其中表达出创建模式的意图。

如果是像下面这样的注释，就没有多大意义了——特别是当某天再看这些代码并想知道它为什么没有起作用时：

```
// 以下代码用 Moon 替换 Star
```

怎么使注释更有意义，看下面这个注释的例子：

```
// 以下代码以区分大小写的方式匹配 Star，从而排除像 start 和 star 这样的单词
// 而且，只有当 Star 后跟一个空格符和字符序列 Training 时，
// 或者后跟一个句点字符时，
```

```
// 或者后跟一个问号时才会匹配
```

像这样的注释才能对正则表达式的作用，以及什么情况下会匹配正则表达式模式的组件给出更清晰的概念。

如果在解决问题的方法时，某种方法不适用，那么把这种方法为什么不适用放到注释中也是非常有意义的。虽然承认错误会令人不太舒服，但把问题摆在前面总比浪费几周时间后仍然钻进同一条死胡同更可取。

10.1.3　利用扩展模式

当我们用 JavaScript、Java、Visual Basic .NET 和其他语言编写代码时，总会把代码注释适当地隔开，并对嵌套的代码块进行缩进处理，以便让代码更清晰易读。如果可以避免，笔者从来不会把一大段代码放到一行中，因为这样不好阅读。提高代码的可读性和添加注释是使编码及维护更顺畅的两个最相关的方面。

在普通代码中添加注释的一个关键好处是可以把注释正好放在与其相关的代码组件之后。如果把注释放在相关代码后一、两屏远的位置上，就不太有用。在许多正则表达式实现中，都可能存在类似的问题，就是无法把注释放到与它们相关的代码旁边。

在 Perl、Java 和 PHP 等语言中可以使用扩展模式。扩展模式允许把注释放在相关模式组件的同一行中。这样有助于减少对代码的误解。

在 Perl 中，扩展模式用 m//操作符中的第二个正斜杠后的 x 修饰符来表示。

要匹配两个已知用户的输入，可以使用像 JimOrFred.pl 这样简单的程序：

```
#!/usr/bin/perl -w
use strict;
print "This program will say 'Hello' to Jim or Fred.\n";
my $myPattern = "^(Jim|Fred)\$";
# The pattern matches only 'Jim' or 'Fred'. Nothing else is allowed.
print "Enter your first name here: ";
my $myTestString = <STDIN>;
chomp ($myTestString);
if ($myTestString =~ m/$myPattern/x)
{
print "Hello $myTestString. How are you today?";
}
else
{
print "Sorry I don't know you!";
}
```

这个程序简单地接受用户的输入。如果用户输入的名字是 Jim 或 Fred，会显示欢迎信息；否则，用户就会看到系统不认识该名字的信息。

图 10-1 显示的是输入两个可以接受的名字后的程序运行情况。

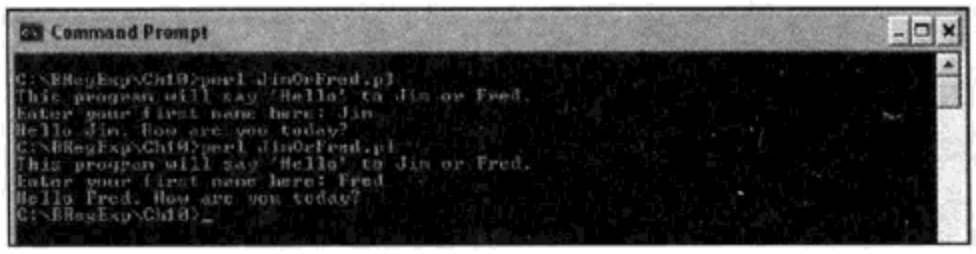


图 10-1

下面的单行注释提供了恰当的信息：

```
# The pattern matches only 'Jim' or 'Fred'. Nothing else is allowed.
```

通过扩展模式，可以在代码内部就模式每个组件的作用添加更详细的注释。

文件 JimOrFred2.pl 显示的是使用扩展模式的同一组代码。注意针对变量 $myPattern 的声明语句分布在多行中：

```
#!/usr/bin/perl -w
use strict;
print "This program will say 'Hello' to Jim or Fred.\n";
my $myPattern = "
^ # Matches the position before the first character on the line
(Jim # Literally matches 'Jim'
| # The alternation character
Fred) # Literally matches 'Fred'
\$" # Matches the position after the last character on the line
;
# The pattern matches only 'Jim' or 'Fred'. Nothing else is allowed.
print "Enter your first name here: ";
my $myTestString = <STDIN>;
chomp ($myTestString);
if ($myTestString =~ m/$myPattern/x)
{
print "Hello $myTestString. How are you today?";
}
else
{
print "Sorry I don't know you!";
}
```

下面这条语句在 JimOrFred.pl 中被写在一行中：

```
my $myPattern = "^(Jim|Fred)\$";
```

而在 JimOrFred2.pl 中则分别写在多行中，而且每一行中的模式组件都包含描述相应组件作用的注释：

```
my $myPattern = "
^  # Matches the position before the first character on the line
(Jim # Literally matches 'Jim'
|  # The alternation character
```

```
Fred) # Literally matches 'Fred'
\$" # Matches the position after the last character on the line
;
```

除了使用扩展模式来支持这种注释，还可以在代码中包含下面的针对整个模式的注释，以便清晰地说明整个正则表达式模式的目的：

```
# The pattern matches only 'Jim' or 'Fred'. Nothing else is allowed.
```

x 修饰符的含义是，JimOrFred2.pl 中变量 $myPattern 内部的空白符可以忽略不计：

```
if ($myTestString =~ m/$myPattern/x)
```

10.2　了解你的数据

使用正则表达式时，需要真正理解所要操作的数据。

下面几节内容将会说明数据中可能存在的几种类型的问题。

10.2.1　缩写词

如果要处理大量文本数据，就会发现某个要匹配的术语的缩写词可能会引起一些问题。

假设想查找有关 Dr. Victor Smith 的信息，可能找到的形式如下：

```
Dr. Smith
Dr. V. Smith
Victor Smith
Doctor V. Smith
Doctor Victor Smith
Dr Victor Smith
```

如上所见，相应的称呼可以被写成 Doctor、Dr(不带句点字符)或 Dr.(带句点字符)。

一些技术性术语也经常会被缩写而导致类似问题。例如，如果想查找有关 Microsoft's Most Valuable Professionals(微软最有价值专家)的信息，那么需要匹配如下这些形式：

```
MVP
MVPs
Most Valuable Professional
```

10.2.2　固有名字

如果数据中有关内容涉及到固有名词，无论是与人有关、与商业有关还是与位置有关，它都会导致一些问题。

例如，如果你对著名画家 Leonardo da Vinci(达芬奇)感兴趣，很可能会在数据中发现下面这一些名字的变体：

```
Leonardo Da Vinci
Leonardo da Vinci
Leonardo DaVinci
```

```
Leonardo daVinci
```

注意这四种变体和 Vinci 前面有无空格符的变化。

仅匹配 Leonardo da Vinci 前面的这几种变体是远远不够的，因为可能还会有下面这样的惯用语：

```
the great da Vinci
sgnr da Vinci
Sgnr da Vinci
Mr. da Vinci
```

可以使用一个非特定的直接量模式，比如 vinci，在不区分大小写的情况下获得要操作的数据中都使用了哪些变体的一个初步印象。当然，这个不具有针对性的模式也会不可避免地返回一些不想要的匹配项，比如 invincible 中的 vinci。在初步试探性阶段，出现不想要的匹配没关系。关键是这样可以看到与想匹配的名字有关的所有可能的形式。

在知道数据中可能存在的拼写变体后，就可以开始着手设计适当的正则表达式模式了。

10.2.3 错误的拼写

人们在拼写单词时难免会出错，即使是拼写熟悉的单词时也不能始终保证百分百正确。除非运气好，否则在操作的大量文本中肯定会存在一些拼错的词。要最大化灵敏度和特殊性，就必须考虑到拼错的情况，至少在处理重要或者大范围的文本时需要如此。

要考虑拼错的情况，可以使用下面的探测模式：

```
\b\w+\s+Training
```

或者

```
Star\s+T\w+g
```

前一个模式会检查 Training 前面的单词。因此，可能会筛选出诸如 Satr 和 Star 之类的变体。后一个模式可以筛选出许多种可能拼错的 Training。

拼写检查可以避免某些问题，但同时也可能引入其他问题。最近，笔者看到有人发表姓 Debate 的作者的一本书的信息。事实上，那个姓是错误的。之所以出现这种问题，是因为拼写检查程序在把它不能识别的姓修改为自己熟悉的另外一个姓。

10.3 创建测试用例

当对多个文档应用正则表达式时，创建测试用例的作用非常明显。在第 2 章中提过理解要操作的数据源很重要。随着要搜索或操作的文档数量的增加，或者数据范围的加大，投入必要的时间对要使用正则表达式搜索的这些数据进行彻底分析也就变得越重要。

能否创建出恰当的测试用例，取决于对数据源的理解。要测试是否能检测到引用 Star Training 的所有 Star，可能需要在测试用例中包含以下内容：

```
Star Training
Star.
Star?
```

为保证不匹配不想要的字符序列，还要考虑以下文本：

```
Star performer
Starting from the beginning
```

确保成功地找出想要匹配的文本和不想匹配的文本，可以增强对模式能否如愿以偿地作用于测试数据，或者暴露出所使用技术未能匹配想要的文本或匹配了不想要的文本等问题的信心。

当模式没有按期望行事时，首先要分析不想要的结果，看是否能迅速发现其行为不符合预期的原因所在。在前几章的例子中，逐个字地解释了匹配的过程。如果能够理解那些解释，就可以分析出希望自己的模式做什么。如果分析不奏效，那么就需要回到起点，再重新建立一个问题定义，并投入更多时间在理解数据源的基础上完善问题定义。

10.4　调试正则表达式

如果可能的话，应尽量避免调试正则表达式。因为调试正则表达式既耗费时间，也容易令人受挫。

在逐步精炼问题定义的过程中投入的时间越多，对使用正则表达式做什么也就越清晰。如果在代码中还清晰地说明了正则表达式每个组件表达的意图，就可以最大限度地减少被正则表达式代码搞得晕头转向的次数。

然而，即使代码十分清晰，也可能突然蹦出几个问题。

10.4.1　叛逆的空白符

空白符是正则表达式中常见的叛逆因子。在现代高分辨率的显示器中，要确定模式中是否存在一个空白符——特别是一个单独的空格符并非易事。同样，在测试文本的某些部分中也可能存在数量不定的空白符。

当模式中缺少必要的空白符或者存在多余的空白符时，由空白符导致的问题就会出现。

缺少经验的正则表达式程序员常会犯的一个错误就是，在模式中用于分隔选项的竖线(|)后面带一个空格符。表面上，这样会提高代码的可读性，但这种可读性是以改变模式的含义为代价换来的。

如果使用本章前面已经介绍过的扩展模式，那么正则表达式中的任何空白符都可以忽略不计。所以，如果模式要匹配包含空白符(如空格符)的字符序列，那么必须使用元字符 \s (它匹配任何空白符)来指定相应的空白符。

在文件 JimOrFred3.pl 中模式的竖线后面有一个不想要的空格符，如下所示：

```
my $myPattern = "^(Jim| Fred)\$";
```

除此之外，JimOrFred3.pl 与 JimOrFred.pl 完全相同。但存在这个空白符的结果是 Jim 仍然匹配，但 Fred 则不会匹配了。因为，此时正则表达式引擎尝试匹配的模式变成了一个空格符后跟字符序列 Fred 了，而用户并没有输入这样的名字，所以匹配失败。图 10-2 显示的是字符序列 Fred 匹配失败的情况。

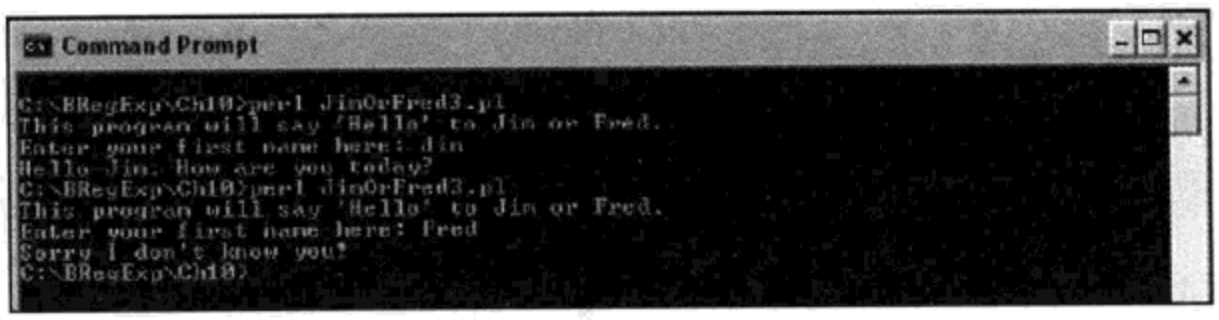


图 10-2

如果电脑中没有安装 Perl，请参考第 26 章中有关下载和安装 Perl 的信息来运行这个代码。

试一试：基本交替选择的例子

本例要求用户输入他或她的名字，然后系统根据能否识别这个名字来显示不同的信息。

本例中的代码使用了简单的交替选择 (Jim| Fred) 来判断用户是否输入了 Jim 或 Fred。而位置元字符 ^ 和 $ 用于指定除所要名字之外的其他输入都不会匹配。

(1) 在文本编辑器中输入下列 Perl 代码，或使用本章的测试文件 JimOrFred3.pl。

```
#!/usr/bin/perl -w
use strict;
print "This program will say 'Hello' to Jim or Fred.\n";
my $myPattern = "^(Jim| Fred)\$";
print "Enter your first name here: ";
my $myTestString = <STDIN>;
chomp ($myTestString);
if ($myTestString =~ m/$myPattern/)
{
print "Hello $myTestString. How are you today?";
}
else
{
print "Sorry I don't know you!";
}
```

(2) 使用 perl JimOrFred3.pl 命令在命令行中运行这些代码。

(3) 在提示状态下，输入 Jim 并按回车键。

(4) 观察显示的结果。

(5) 再次运行这些代码，输入 Fred，然后按回车键。

(6) 观察显示的结果(图 10-2 显示的是这一步的结果)。

工作原理

当模式是 ^(Jim| Fred)$ 时，字符序列 Jim 会匹配，因为它是位于竖线之前的那个字符序列。然而，在竖线字符之后的模式是一个空格符和 Fred。除非用户会输入一个空格符和 Fred，否则不会匹配。

可能你会对 JimOrFred3.pl 中的模式使用 \$ 感到奇怪。我们马上会讨论到这个问题。

由于空白符的存在还会导致一些其他问题。一种可能性就是问题出在用户而非开发人员身上。

假设你再次运行 JimOrFred.pl，在图 10-1 中我们看到该程序匹配 Jim 和 Fred 这两个字符序列。然而，你可能会接到 Fred 打来的电话，说他在登录时被程序拒之门外了。发生这种情况的原因是他输入 Fred Schmidt，然后又删除了 Schmidt，但却留下了中间的空格符。结果这个输入没有被程序匹配，因为程序中不允许存在空格符。而他可能会给你发一个屏幕截图(如图 10-3 所示)，告诉你登录失败。

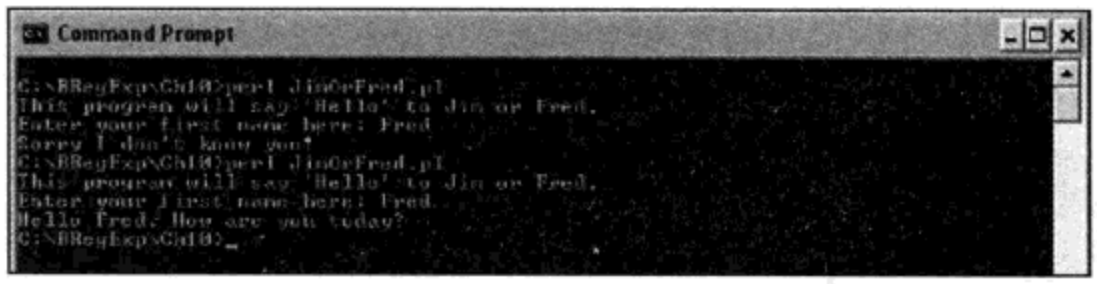


图　10-3

诚然，这个例子有一点牵强。但它要说明的重点是用户的行为难以预料。如果编码时没有考虑到这些问题，就可能会碰到这些永远也想不到的问题。虽然代码没有“错误”，但可惜它不允许用户有意料之外的动作。

10.4.2　反斜杠导致的问题

在某些设置下，漏掉或者增加反斜杠会改变正则表达式的含义。如果模式尝试匹配的字符序列与想要匹配的不一样，就会得不到目的结果。

在 Perl 中使用$元字符时会出现这种情况。例如，当在 JimOrFred3.pl 中给变量 $myPattern 赋值时，写成下面这样：

```
my $myPattern = "^(Jim| Fred)\$";
```

此时，如果漏掉其中的反斜杠意味着代码将不会被编译。然而，这里使用 \$ 来表示 $ 元字符的做法令人迷惑。

但是，在其他情况下可能会发现没有编写错误的向前查找或向后查找居然会失败。你可能会指望正则表达式引擎匹配一个字符，但它却尝试去匹配另外一个字符。结果是正确的模式却匹配失败。

10.4.3　考虑其他原因

毋庸置疑，正则表达式越复杂，导致意想不到的和不想要的结果的可能性也就越大。但是，不能因为一个冗长的正则表达式很复杂或者含义隐晦，就对由于分析过程或代码的

其他地方可能存在缺陷而导致不想要的结果视而不见。

可能出现的问题取决于代码想要实现的功能。切记正则表达式代码的问题是一个涉及到正则表达式模式、数据源和周边代码的复杂交互性问题。如果问题持续存在，就需要系统地检测每种可能性。

第11章

在 Microsoft Word 中使用正则表达式

在 Microsoft Word 中进行搜索或搜索替换时，正则表达式是非常有用的。然而，在 Microsoft Word 中正则表达式的语法与多数编程语言都显著不同。Microsoft Word 不仅把正则表达式称做通配符，而且还包含许多非标准的特性。因此，如果在其他语言或工具中使用过正则表达式，那么在 Microsoft Word 中就必须进行重大调整，一方面是因为 Word 中的语法不标准，另一方面是因为它的功能也相当有限。比如，没有向后或者向前查找功能。

在本章中将学习以下内容：

- 如何在 Microsoft Word 中使用正则表达式功能
- Microsoft Word 支持的正则表达式功能，以及与标准正则表达式语法的差别
- 如何在某些特定的情形下使用 Microsoft Word 的通配符

11.1 用户界面

在 Microsoft Word 中使用正则表达式的界面非常直观。如果想在 Word 2003 中使用正则表达式(通配符)，只须简单地选择 Edit 菜单，然后选择 Find；也可以使用键盘快捷键 Ctrl+F，打开 Find and Replace 对话框。如果从没有在 Word 中使用过正则表达式或者其他高级搜索功能，该对话框的预期外观将如图 11-1 所示。

而要想启用正则表达式功能，请单击 More 按钮。此时，会有更多选项显示出来，如图 11-2 所示。

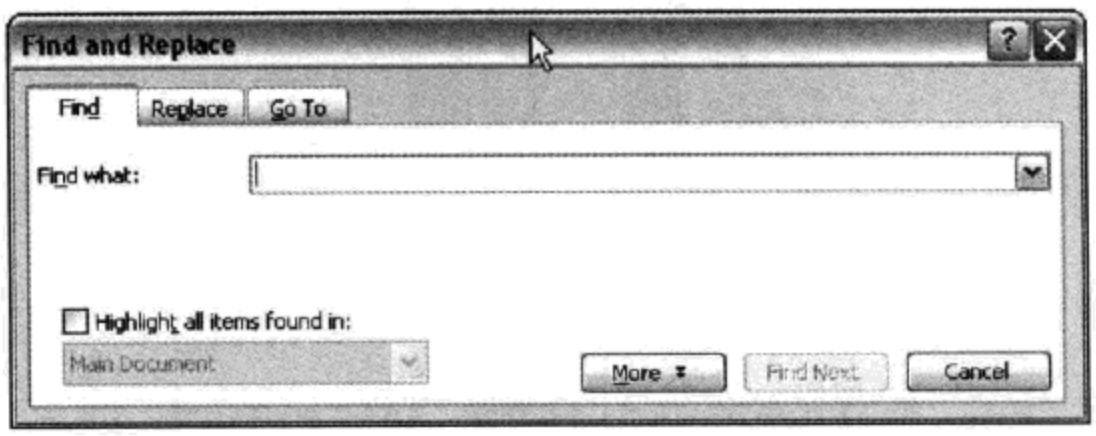


图 11-1

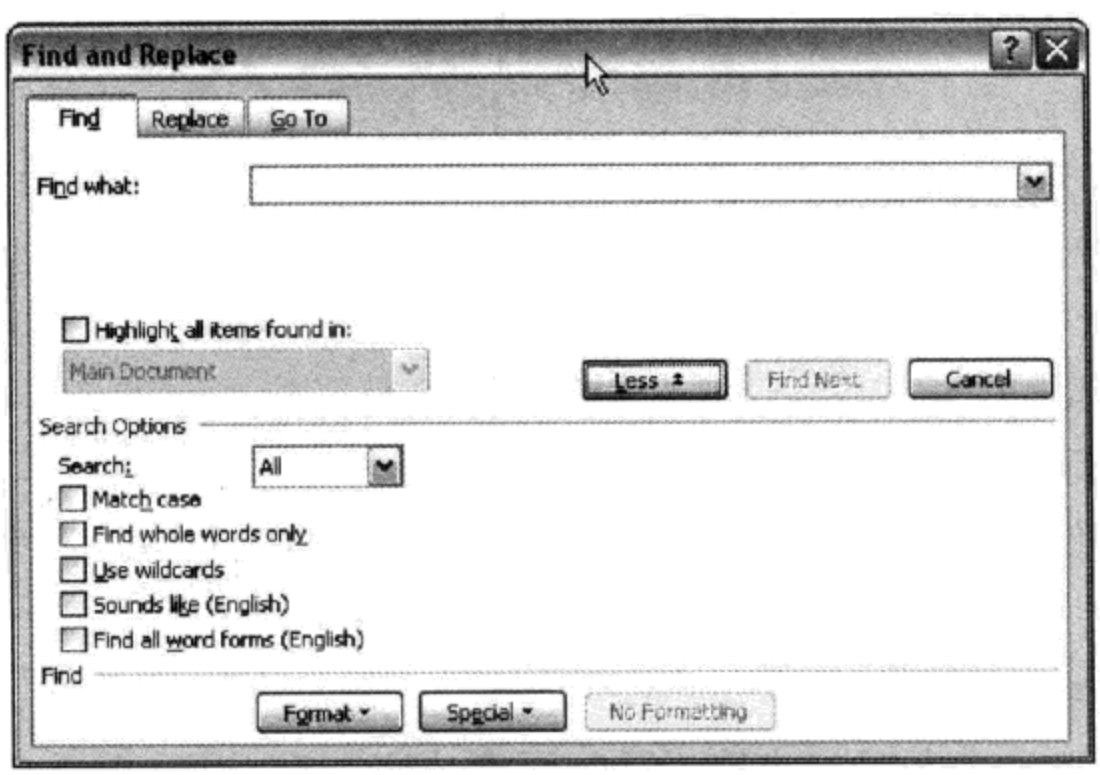


图 11-2

为了使用 Word 的正则表达式功能，选中 Use wildcards 复选框。

试一试：Word 通配符

(1) 在 Microsoft Word 启动后，打开测试文件 Apple.doc。

该测试文件的内容如下：

```
The skin of the apple is green.

The apple's flavor was fantastic.

Apples grow on trees.
```

(2) 按 Ctrl+F 快捷键打开 Find and Replace 对话框。如果必要，单击 More 按钮显示 Use wildcards 复选框。

(3) 选中 Use wildcards 复选框。当单击 Use wildcards 复选框时，Match case 和 Find whole words only 复选框变灰。而 Sounds like(English)和 Find whole words only 复选框一样也不能与 Use wildcards 复选框同时选中。虽然在 Word 中这三个选项显示为复选框，但从

功能上看更类似 Web 表单中的单选按钮，因为任何时候我们只能选择其中一个选项。

(4) 在 Find what 文本框中输入 apple，并单击三次 Find Next 按钮，在每次单击后注意观察突出显示的文本。

图 11-3 显示了在第 4 步中单击两次 Find Next 按钮后的结果。当单击第三次后，会有一个对话框显示出来，表示已经到达文件结尾处了——“Word has finished searching the document(Word 已完成对文档的搜索)”。

工作原理

第一行中的字符序列 apple 显然会被匹配。而第三行中的字符序列 Apple 却没有匹配，这是因为在使用通配符时 Microsoft Word 的默认行为是以区分大小写的方式进行匹配。而由于 Apple 的首字符是大写的 A，但模式中的第一个字符是小写的 a，所以没有匹配。

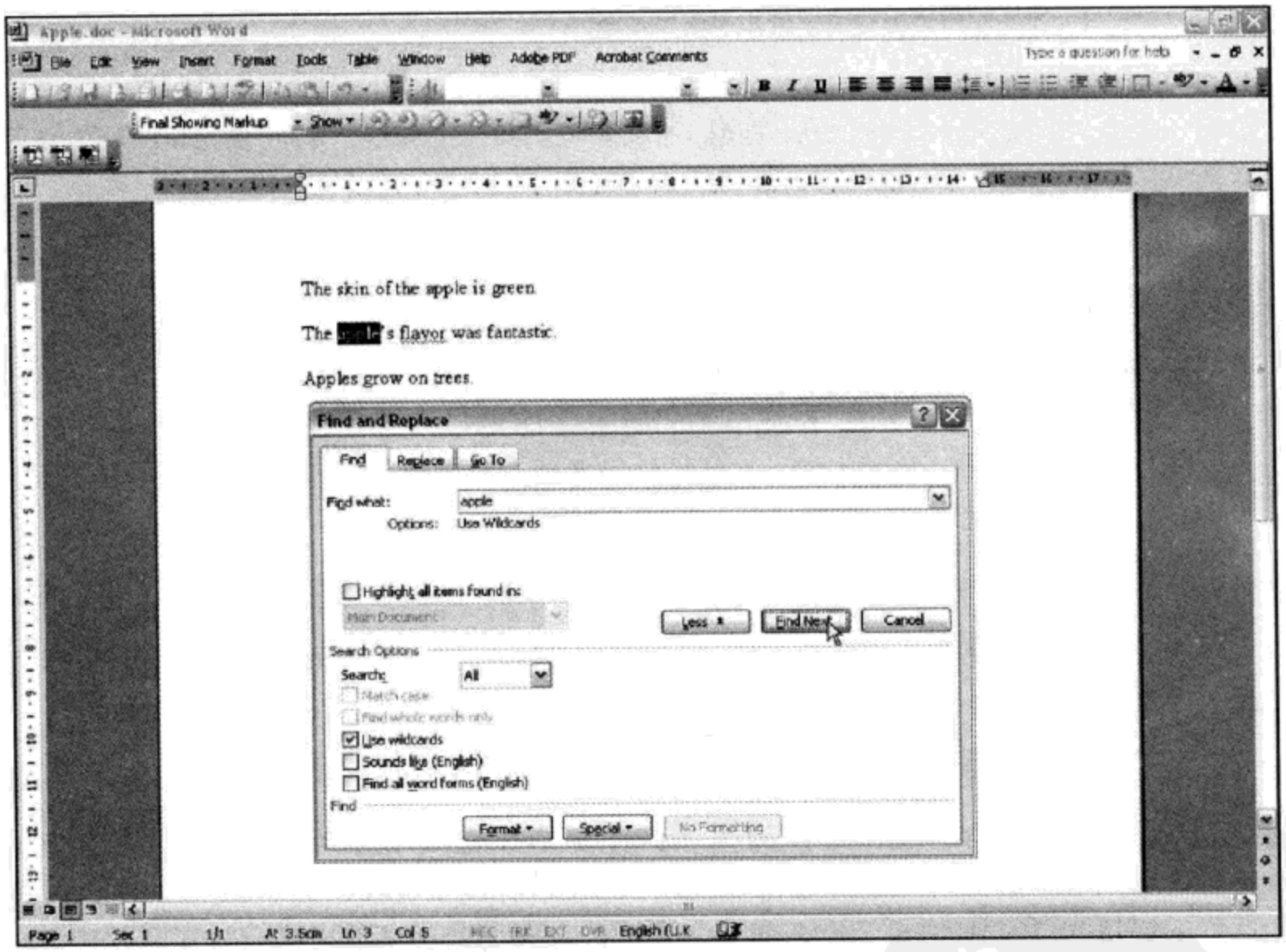


图　11-3

11.2　可用的元字符

下面的表 11-1 中显示了在 Word 2003 中可以使用的元字符。有关这些元字符在 Word 中的功能将在本章后面几节中更详细地介绍。

表 11-1　Word 中可用的元字符

元 字 符	描　　述
?	非标准的用法(在 Word 中，它不是限定符)。匹配一个单独的字符。大致相当于标准语法中的句点元字符(.)
*	非标准的用法。匹配零个或多个字符。大致相当于标准语法中的 .*
<	匹配位于非字母字符与尾随的字母字符之间的位置。事实上，就是匹配词的开始位置的元字符
>	匹配位于字母字符与尾随的非字母字符之间的位置。事实上，就是匹配词的结束位置的元字符
[...]	字符类定界符
!	作为字符类取反元字符用于字符类中
{n,m}	限定符语法
@	限定符。非标准语法，匹配一个或多个前面的字符或组。等价于标准语法中的 + 元字符

11.2.1　限定符

Microsoft Word 并没有把多数正则表达式实现中使用的 ?、* 或 + 元字符作为限定符。在 Microsoft Word 中 ? 和 * 元字符的用法与 DOC 文件名中的用法相同。

? 元字符表示一个单独的字符，并且等价于多数正则表达式实现中的模式 . 或者 \w。因此，下面的模式：

```
so?t
```

在 Microsoft Word 中会匹配 soft 和 sort。然而，? 元字符在 Word 中并不代表可选择的意思。下面的模式：

```
sa?t
```

会匹配 salt，但不会匹配 sat——因为在 sat 的 a 与 t 之间没有字符。? 元字符在 Word 中总要匹配一个真正的字符(包括空白符)。

* 元字符在 Word 中表示零个或多个字符，大致相当于多数正则表达式实现中的 .* 或 \w*。值得注意的是，虽然 ? 元字符没有可选的意思，但 * 却可以表示可选的字符。因此，模式：

```
se*t
```

会匹配 set(e 和 t 之间的字母是可选的)、seat、sect 和 sent。但是，当 * 元字符在 Word 中匹配零个字符时，匹配结果有时也会令人瞠目结舌。例如，下面的模式：

```
s*t
```

会匹配单词 sit 和 sat，但也会匹配单词 first 和 best 中的字符序列 st。图 11-4 显示了匹配 first 中的 st 的情景。相关的测试文本如下：

```
sit
sat
first
```

所用模式为 s*t。

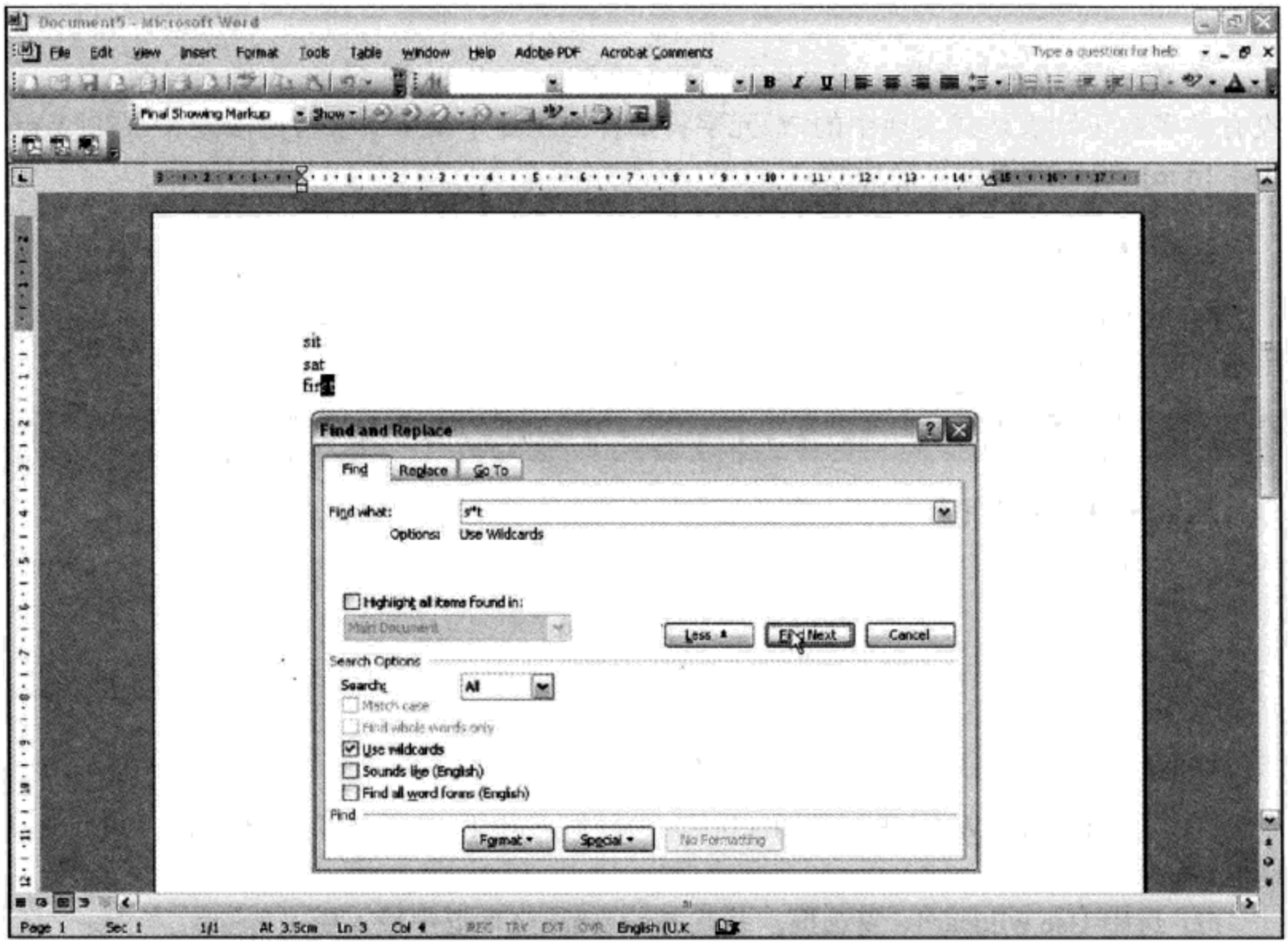


图　11-4

如果想完成一次搜索和替换操作，那么这种意想不到的匹配将会在替换操作之后使文本变得非常古怪。

在 Word 中使用 * 元字符会极大地降低搜索的特殊性。所以，一般而言都要避免使用 * 元字符来完成替换操作。

下面，表 11-2 中总结了 Microsoft Word 对限定符的支持情况。

表 11-2　Microsoft Word 中的限定符

元　字　符	说　　明
?	? 元字符在 Word 中不是一个典型的限定符。它匹配一个字符，因而并不是限定符
*	* 元字符在 Word 中不是一个典型的限定符。它匹配零个或多个字符
@	@元字符
{n,m}	{n,m}语法在 Word 中是限定符

在 Word 中没有与多数正则表达式实现中匹配零个或一个字符的 ? 等价的限定符。也没有像多数正则表达式实现中的 * 元字符那样匹配零个或多个字符的限定符。在 Word 中，{n,m} 也不能提供等价功能。

1. @限定符

Microsoft Word 中唯一的限定符就是 @ 元字符。它等价于多数实现中的 + 元字符，用于匹配一个或多个位于它前面的字符或组。

试一试：@限定符

这里使用测试文件 Hot.doc，其内容如下：

```
hot
hoot
host
holt
shoot
ht
```

(1) 在 Microsoft Word 中打开 Hot.doc，并使用 Ctrl+F 快捷键打开 Find and Replace 对话框。

(2) 选中 Use wildcards 复选框，并在 Find what 文本框中输入模式 ho@t。

(3) 单击三次 Find Next 按钮。在每次单击后，观察突出显示的文本。

图 11-5 显示了第三步后的结果，此时已经单击了三次 Find Next 按钮。在第一次单击后，字符序列 hot 被突出显示。在第二次单击后，字符序列 hoot 被突出显示。而在第三次单击后，shoot 中的字符序列 hoot 被突出显示。而如果再次单击 Find Next 按钮，最后一行的字符序列 ht 也不会突出显示，因为在 ht 的 h 与 t 之间有零个 o，这不匹配模式中的组件 o@。

(4) 新建一个 Word 文档，并在其中输入测试文本 AABABABAAAAAAA。

(5) 打开 Find and Replace 对话框，并选中 Use wildcards 复选框。

(6) 在 Find what 文本框中输入模式 A(AB)@，单击 Find Next 按钮，并观察突出显示的文本，如图 11-6 所示。

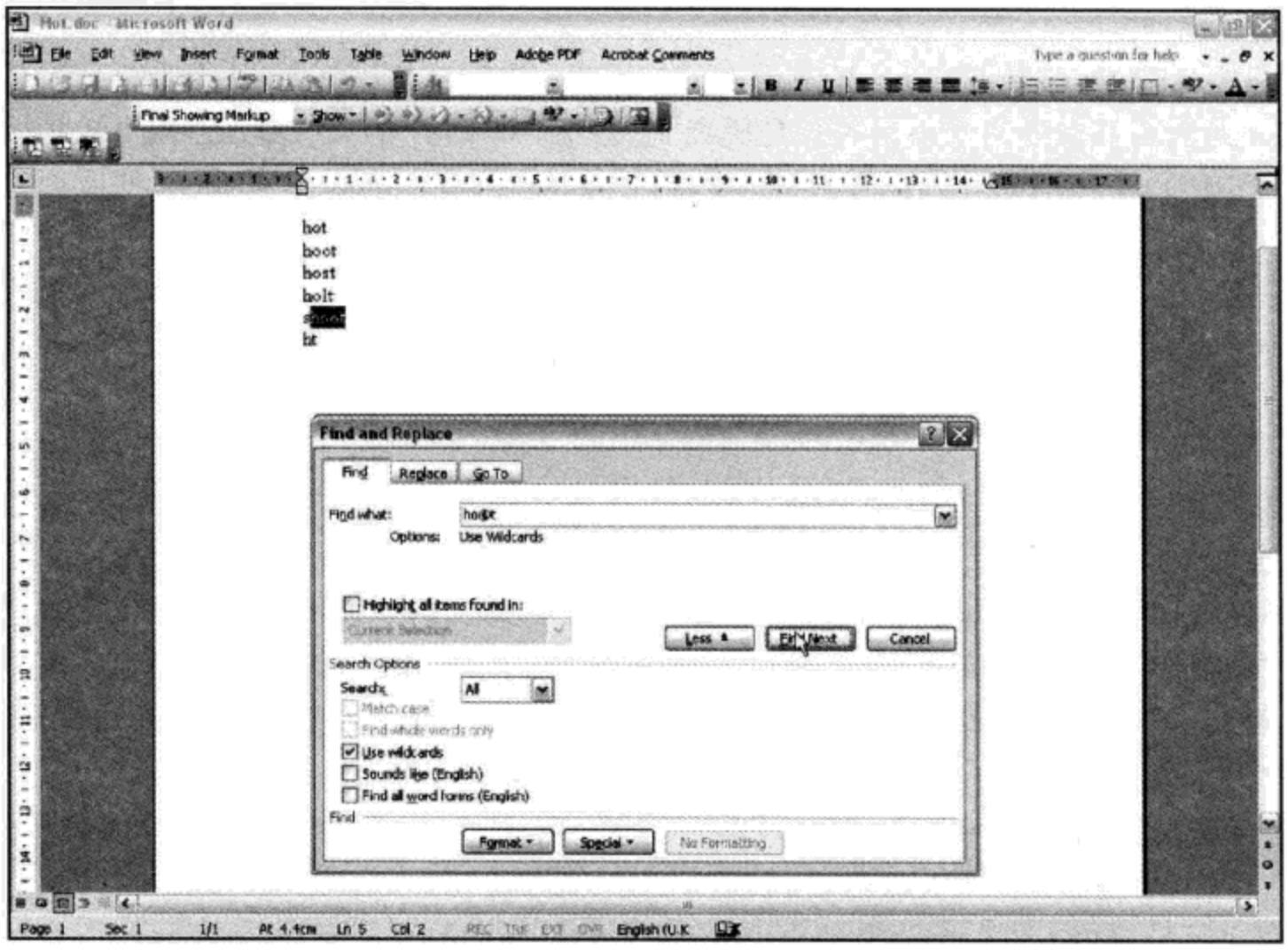


图　11-5

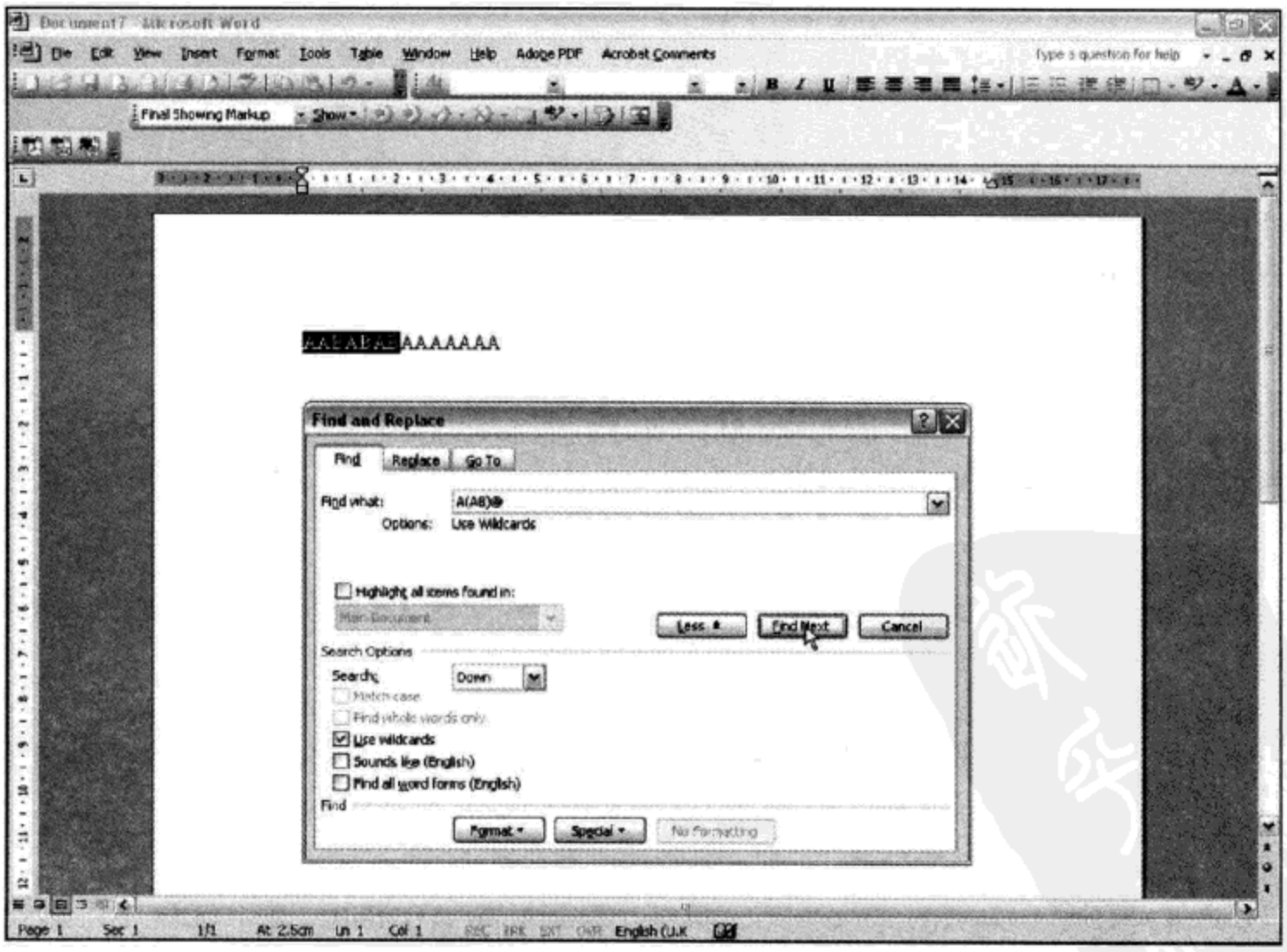


图　11-6

工作原理

模式 o@匹配一个或多个小写 o 的实例。因此，在测试文档 hot.doc 中，第一次单击匹配 hoot 是因为其中的两个小写的 o 与模式 o@匹配。同样，第二次单击匹配 hot 是因为测试文本中的一个小写的 o 匹配模式 o@。而第三次单击匹配 shoot 中的 hoot 的原理与第一次匹配相同。

在使用模式 A(AB)@ 时，@ 限定符引用的是前面的组，而不是一个单独的字符。在这个例子中，模式中的 (AB)@ 组件匹配了字符序列 AB 的连续三个实例。

2. {n,m}语法

虽然在 Word 中可以使用 {n,m} 限定符语法，但遗憾的是，它不能提供与典型的正则表达式语法中的 ? 和 * 元字符等价的限定符。

在 Word 中使用 {n,m} 语法存在一些限制。比如说，不能使用 {0,} 或 {0,*} 创建与典型的正则表达式语法中的 * 元字符等价的限定符。实际上，大括号中第一个数字根本不能是 0。由于缺乏对最少 0 个实例的支持直接限制了通过 {n,m} 语法表达 ? 和 * 元字符的可能性。

试一试：{n,m} 语法

测试文档 Zeros.doc 的内容如下：

```
AB1
AB10
AB100
AB1000
AB10000
```

(1) 在 Word 中打开 Zeros.doc，并用 Ctrl+F 快捷键打开 Find and Replace 对话框。

(2) 在 Find what 文本框中输入模式 AB10{1,100}，单击四次 Find Next 按钮。并在每次单击后观察突出显示的文本。

图 11-7 显示了在第一次单击 Find Next 按钮后文档的外观。

(3) 在 Find what 文本框中将模式修改为 AB10{3,100}，单击三次 Find Next 按钮，并在每次单击后观察突出显示的文本。

工作原理

{n,m}语法与其他实现中的作用相同。n 指定匹配的最少实例数，而 m 指定匹配的最多实例数。

在使用模式 AB10{1,100}时，第一次匹配的是字符序列 AB10，因为模式中的前三个字符分别匹配了相应的直接量，而数字 0 有一个实例，所以满足了最少实例数的限制条件。

另一些与模式 AB10{1,100}匹配的字符序列中数字 0 的个数介于 1~100 之间——分别限定了最少实例数和最多实例数。

当把模式修改为 AB10{3,100} 后，第一次匹配的是 AB1000，其中包含三个数字 0。而字符序列 AB1、AB10 和 AB100 中数字 0 的个数都不到三个，所以没有匹配。

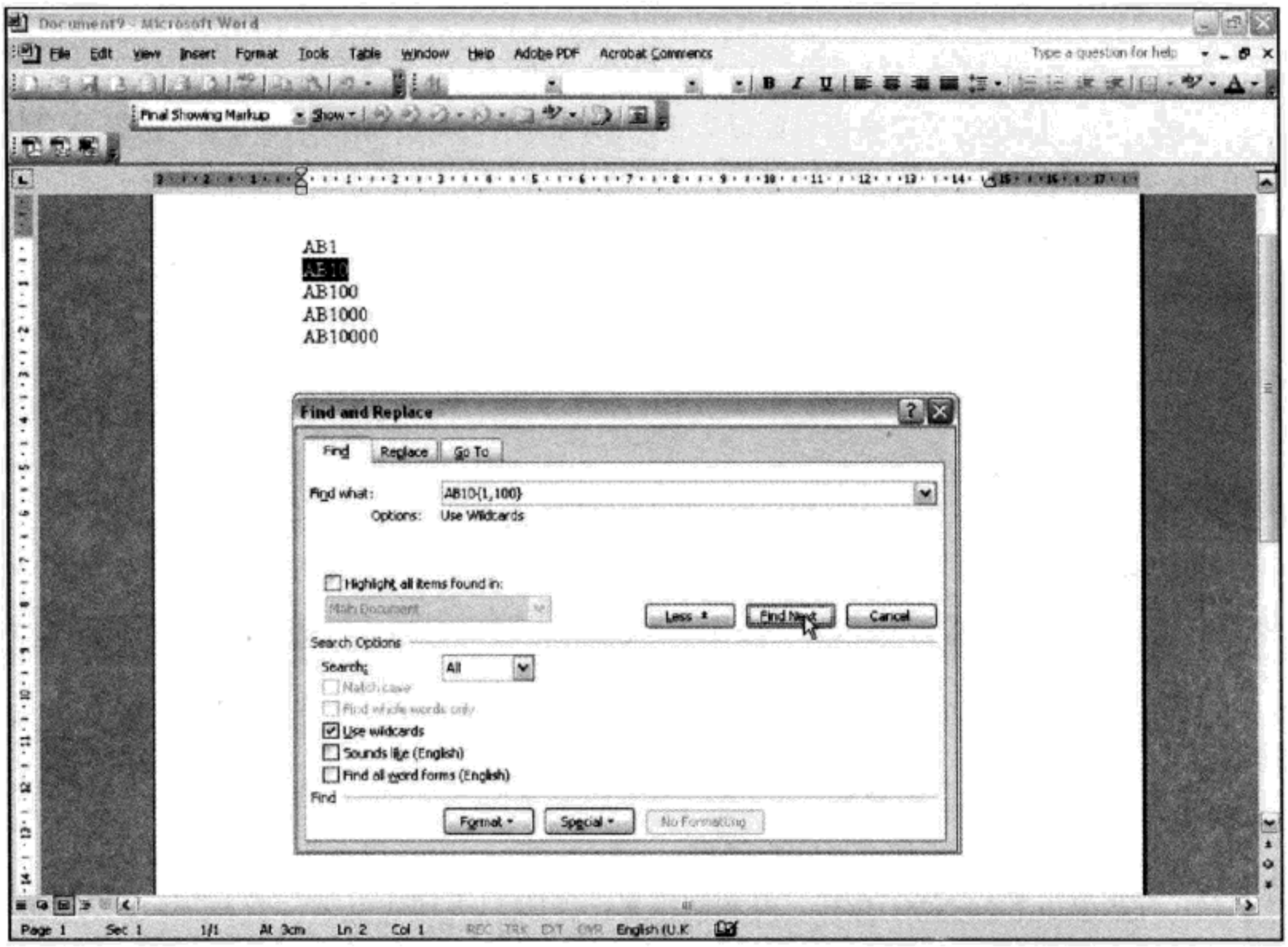


图　11-7

11.2.2　模式

Microsoft Word 中使用模式选项的机会为零。即不可能以不区分大小写的方式来使用通配符功能。

如果把以下模式：

```
Staff
```

应用到测试文档 Staff.doc：

```
Staff is an archaic word for a long walking stick.
The Staff party takes place on Friday.

I spoke yesterday with senior staff.
```

则只会在包含以下文本的行中找到匹配项：

```
Staff is an archaic word for a long walking stick.
```

和

```
The Staff party takes place on Friday.
```

但在下面这行中，则不会有匹配项：

```
I spoke yesterday with senior staff.
```

因为模式 Staff 只匹配以大写 S 开头的字符序列。

不过，有一种办法可以实现不区分大小写的匹配——即在模式中使用字符类匹配直接量字符。虽然这种办法可能会比较麻烦，但在 Word 中是最好的办法了。比如说，对于测试文档 Star.doc，其内容如下：

```
sTar

star

Star

STAR
```

注意观察每行中的某些字符的大小写形式。

试一试：用字符类实现不区分大小写的匹配

(1) 在 Microsoft Word 中打开 Star.doc，并用 Ctrl+F 快捷键打开 Find and Replace 对话框。

(2) 选中 Use wildcards 复选框，并在 Find what 文本框中输入模式 star。

(3) 单击 Find Next 按钮，并观察结果。

(4) 再次单击 Find Next 按钮，并观察结果，如图 11-8 所示。

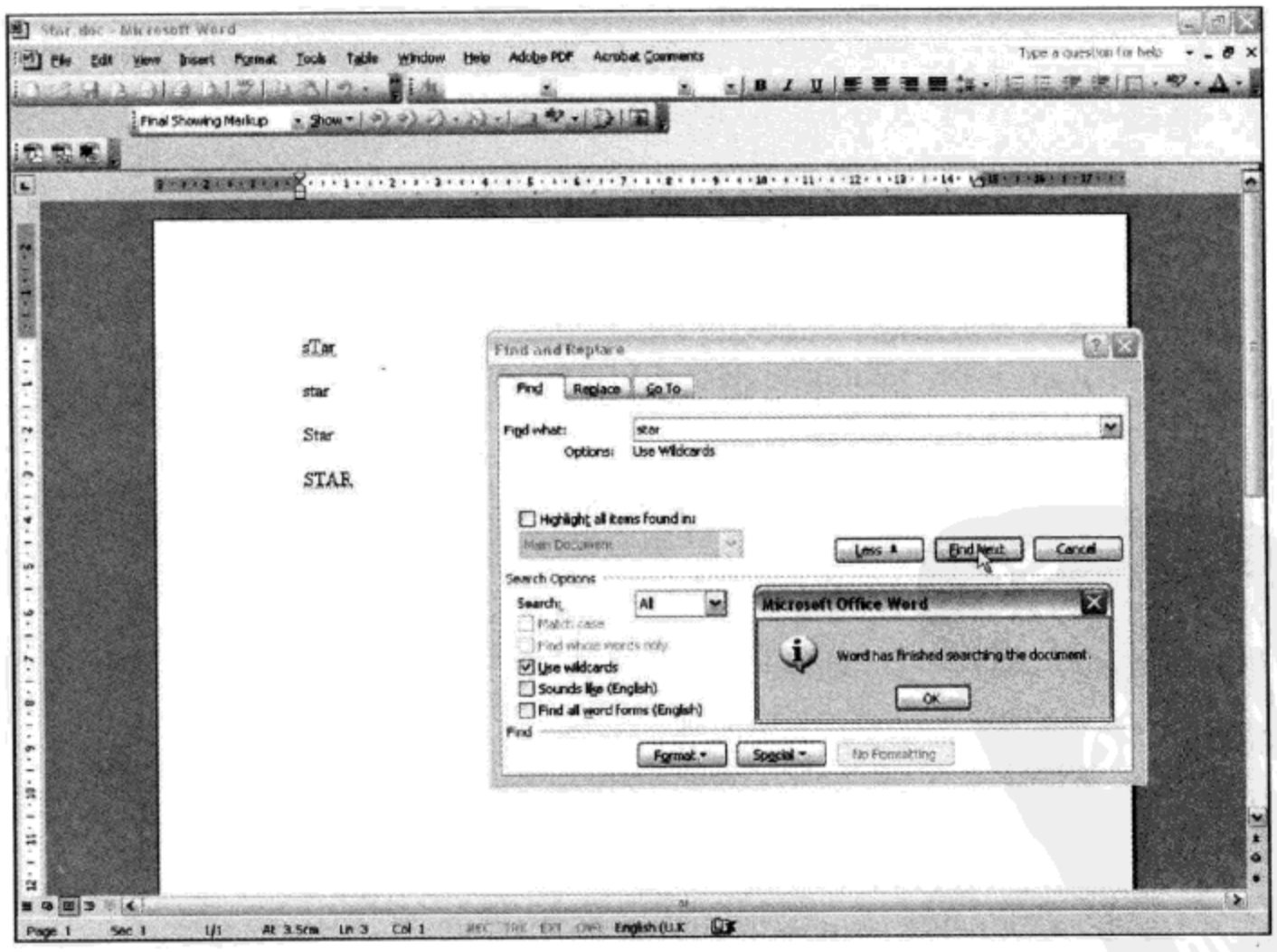


图 11-8

注意只有包含 star(全部小写)的那一行文本中存在匹配项。

当再次单击 Find Next 按钮时，则会显示出图 11-8 中所示的信息窗口。

(5) 在 Find what 文本框中将模式修改为 [Ss][Tt][Aa][Rr]。

(6) 确保光标位于测试文档的开始位置。单击四次 Find Next 按钮。

图 11-9 显示的是在第 6 步中单击一次 Find Next 按钮后的文档外观。

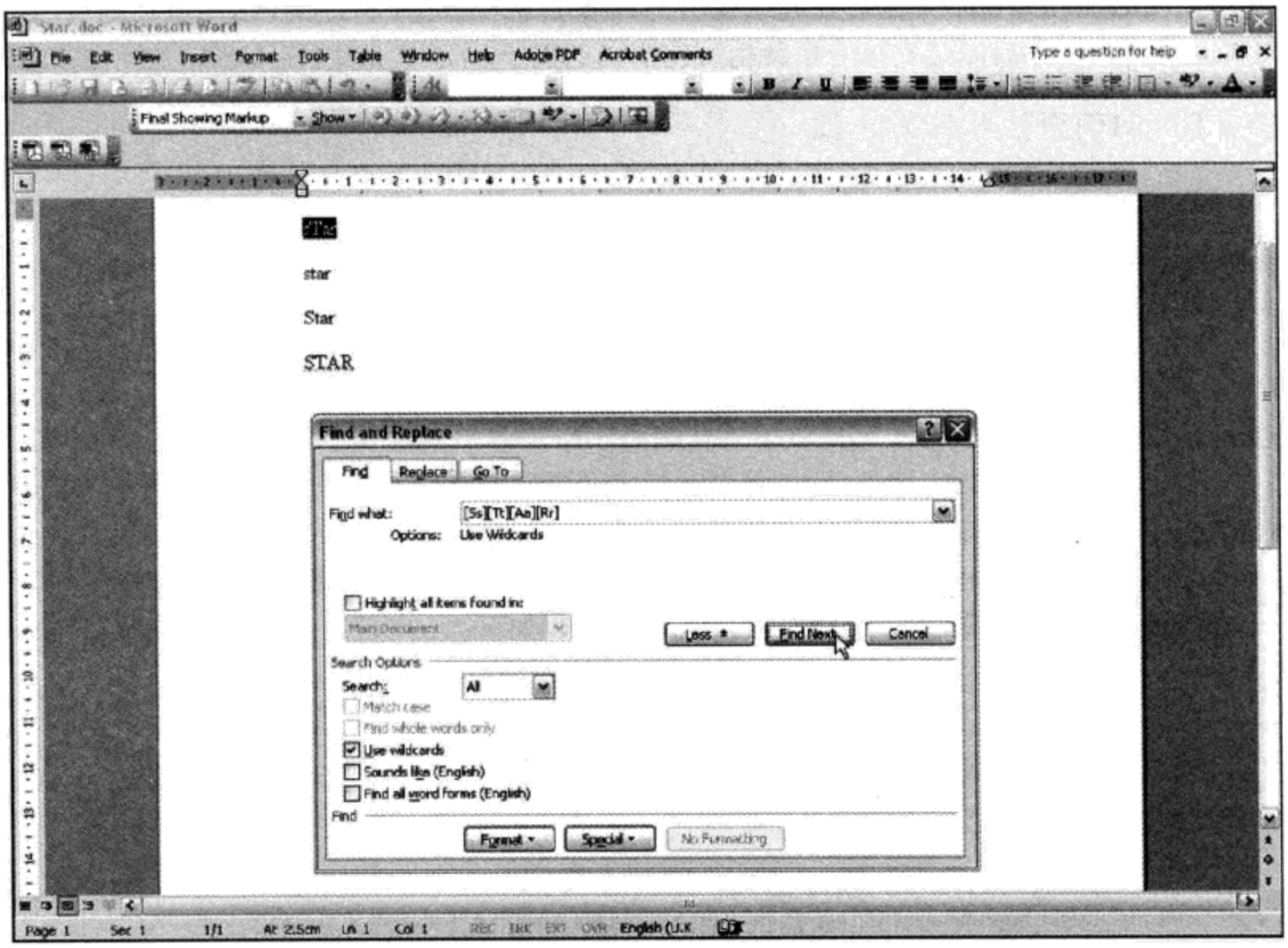


图　11-9

工作原理

模式 star 是以区分大小写的方式进行匹配的。字符序列 sTar 未匹配是因为它的第二个字符是一个大写的 T，在区分大小写的情况下，这与模式中对应的字符(小写的 t)不匹配。

该模式匹配第二行中的字符序列 star 是因为模式中的每一个字符都与测试字符序列中的对应字符匹配。

字符序列 Star 不匹配是因为它的第一个字符是大写的。而字符序列 STAR 不匹配是因为匹配过程在其第一个大写字母处就失败了。

在把模式修改为 [Ss][Tt][Aa][Rr] 后，以上所有测试字符序列都会匹配，因为对于测试字符序列中的每一个字符，模式中都提供了对应的大小写形式。

11.2.3 字符类

Microsoft Word 提供了对字符类的支持。其中区别于标准语法的最典型的变化是使用 ! 元字符来表示对字符类取反(代替了其他实现中的 ^ 元字符)。

11.2.4 反向引用

在 Microsoft Word 中也可以通过使用一对圆括号来分组字符序列，从而成功地使用反向引用功能。本章后面会安排一个使用反向引用的例子。

11.2.5 向前查找和向后查找

Microsoft Word 不支持向前查找和向后查找。

11.2.6 贪婪匹配与懒惰匹配

另一个在 Microsoft Word 中经常出现，而且也与多数正则表达式实现的默认行为不同的是，它默认的行为更倾向于懒惰匹配。而多数正则表达式实现的默认行为都是贪婪匹配(也有许多语言为使用懒惰匹配提供了特殊的语法)。

不过，Word 在某些情况下也会进行贪婪匹配。回忆一下本章前面例子中的模式 A(AB)@，该模式就会匹配尽可能多的实例。

Word 在使用模式时，通常会匹配尽可能少的字符。为了把这个概念讲清楚，我们来看一个例子。所用的测试文件为 ABC123.doc，其内容如下：

```
ABC123

ABC123456

ABC123456
```

试一试：Microsoft Word 中的懒惰匹配

(1) 在 Microsoft Word 中打开文件 ABC123.doc，并使用 Ctrl+F 快捷键打开 Find and Replace 对话框。

(2) 选中 Use wildcards 复选框，并在 Find what 文本框中输入模式 *。

(3) 单击 Find Next 按钮并观察结果。在图 11-10 中，注意只有一个字符(大写的 A)突出显示。

(4) 再单击几次 Find Next 按钮，并在每次单击后观察结果。

(5) 在 Find what 文本框中将模式修改为 A*。

(6) 在文本的开始位置处单击以确保光标处于第一个大写的 A 之前。单击 Find Next 按钮，并观察结果。

(7) 再单击 Find Next 按钮两次，并在每次单击后观察结果。

图 11-11 中显示了在每次单击后，只有相应行中的第一个大写的 A 会突出显示。

(8) 在 Find what 文本框中把模式修改为 A*{2,5}。

(9) 在文本的开始位置处单击以确保光标处于第一个大写的 A 之前。单击 Find Next

按钮，并观察结果，如图 11-12 所示。

(10) 再单击 Find Next 按钮两次，并在每次单击后观察结果。每次单击后，每一行中开头处的字符序列 ABC 会突出显示。

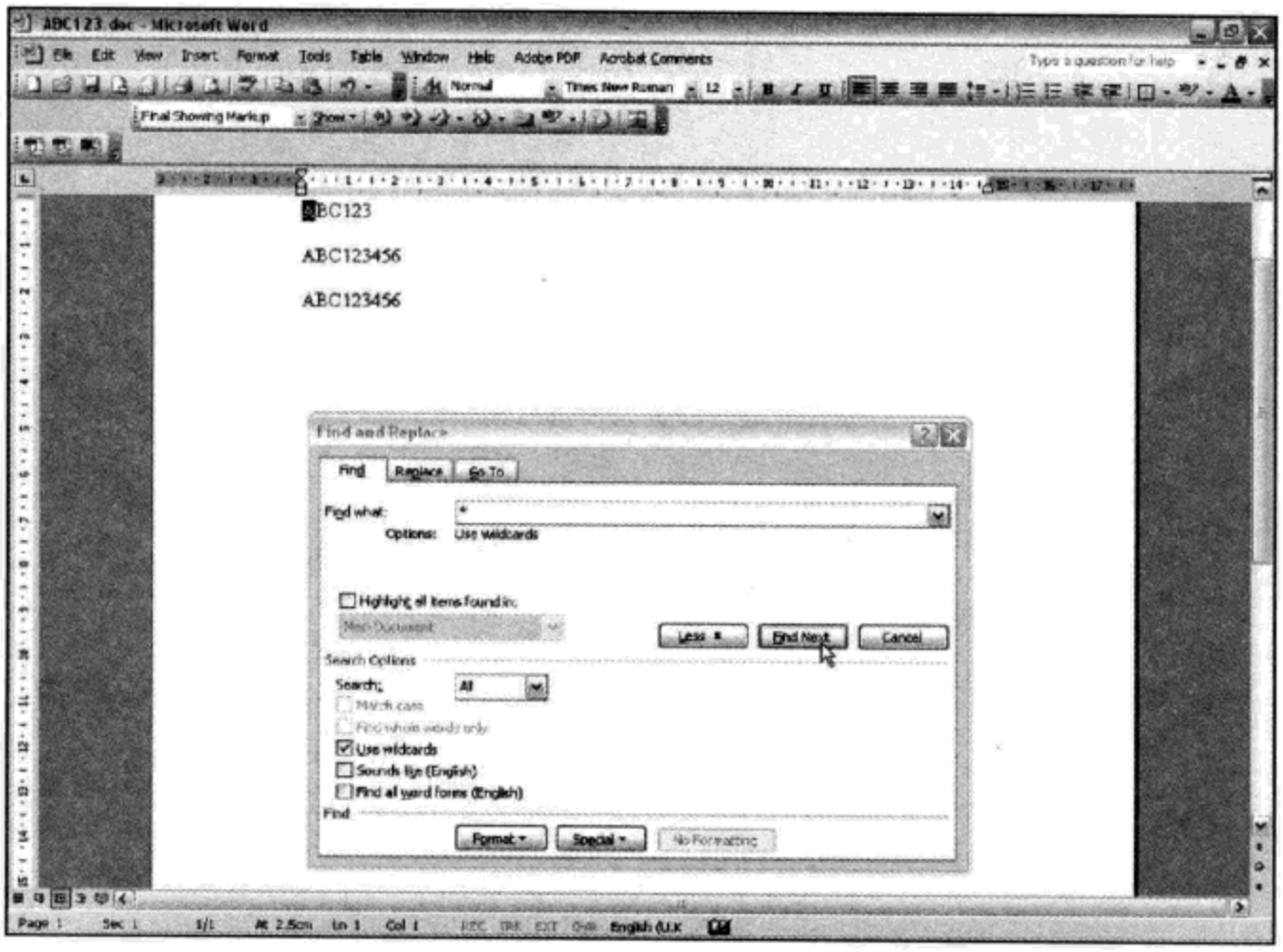


图　11-10

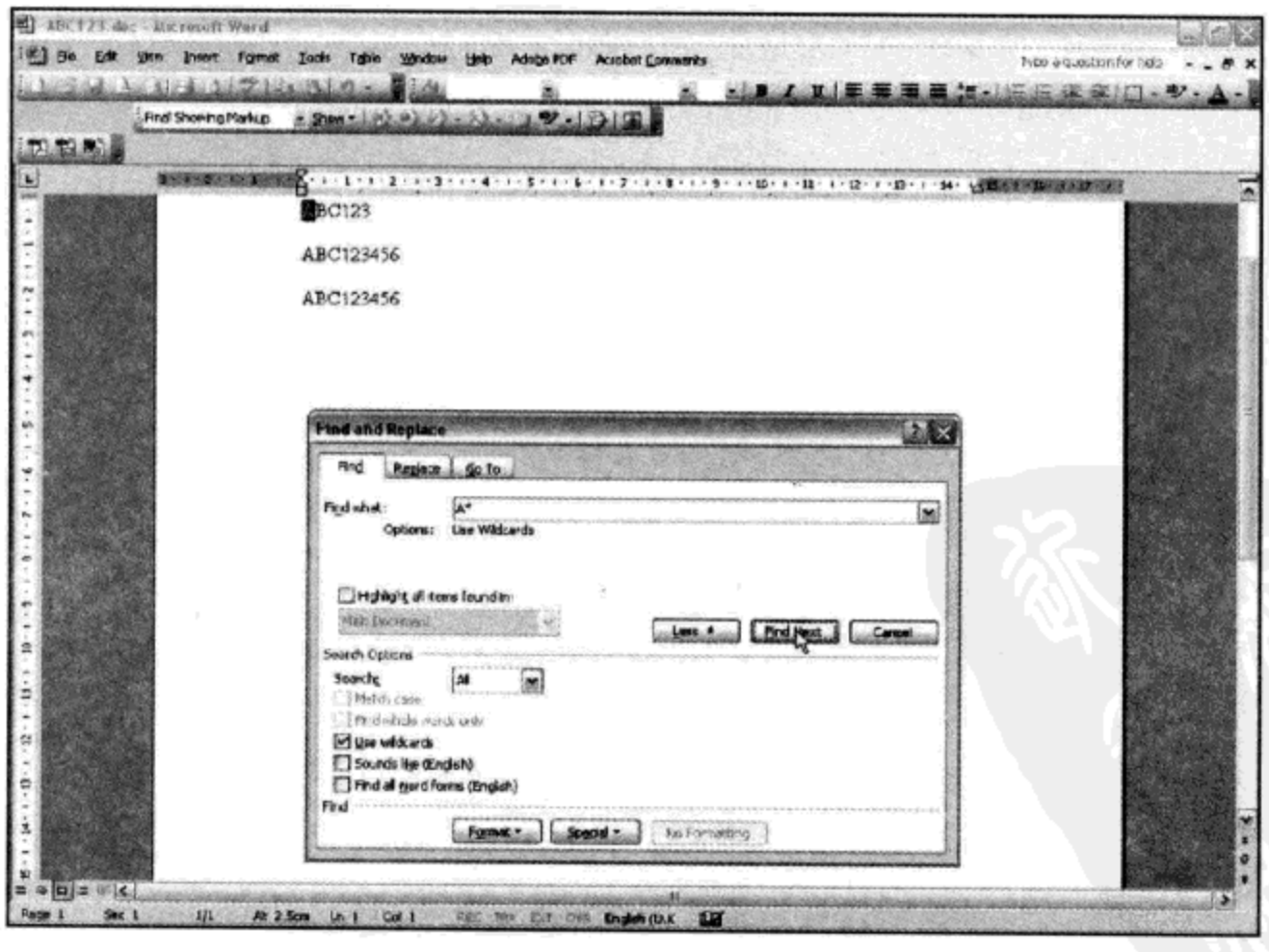


图　11-11

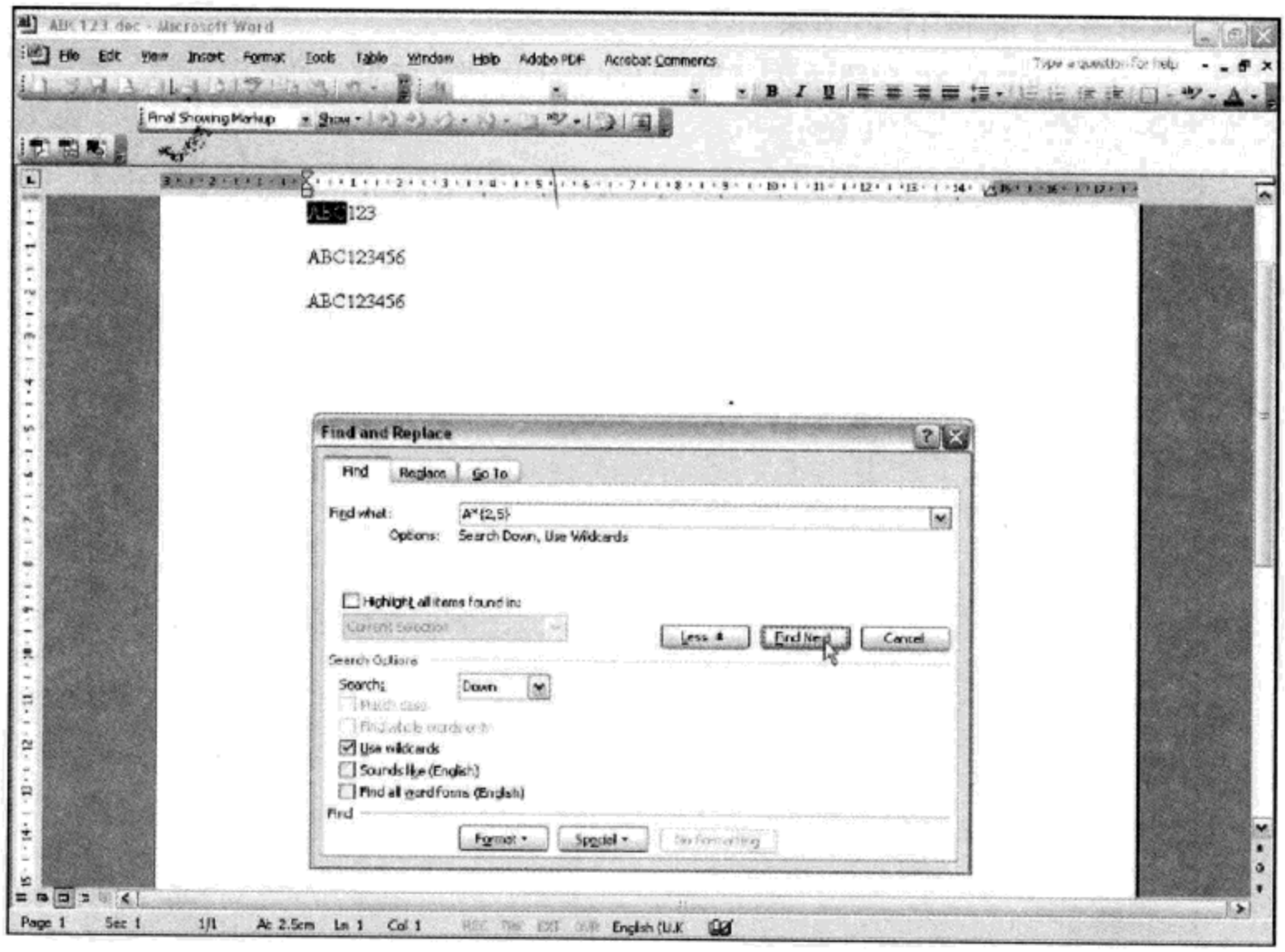


图 11-12

工作原理

在匹配 * 元字符自身时的行为是比较奇怪的。我们都以为 * 元字符在 Word 中匹配零个或多个字符，因此如果对它应用懒惰匹配，那么它应该匹配零个字符。事实上，如我们在图 11-10 中所见，它匹配了一个字符。

如果只使用了 * 元字符来匹配——这种情况不多见，它的行为就是预料之中的懒惰匹配。模式 A* 只会匹配每一行中的首字符 A。假设正则表达式从第一行的起始位置开始匹配，那么模式中的大写字母 A 会与该行测试文本中的第一个字符匹配。然后，又匹配后续字符的最少个数——零个。这就是懒惰匹配。

类似地，在模式 A*{2,5} 中，也只匹配了最少的字符数——BC。

11.3 例子

以下是在 Word 中使用通配符的一些例子。

11.3.1 字符类的例子(包括范围)

Microsoft Word 支持使用字符类的搜索。

Word 中肯定的字符类使用的是标准的正则表达式语法。例如，要查找文档中的字符序列 gray 和 grey，可以使用下面的正则表达式：

```
gr[ae]y
```

测试文档 gray.doc 的内容如下：

```
gray
grey
greying
greyed
grapple
grim
goat
filigree
great
groat
gloat
```

字符类 [ae] 表示要搜索一个字符，可能是 a 或 e。因为没有将匹配结果限制为完整的单词，所以只要 gray.doc 中有包含直接量字符 gr 后跟由这个字符类指定的 a 或 e，再后跟 y 的行，那么该行中的相应字符序列就会被匹配。

11.3.2　全字匹配

搜索组成完整单词的字符序列是一种常见的需求。例如，搜索单词(不仅仅是字符序列)gray 和 grey。要进行全字搜索，可以组合使用词边界元字符<和> 与其他正则表达式组件。

试一试：全字匹配

使用前面的测试文档 Gray.doc。

(1) 在 Microsoft Word 中打开 Gray.doc，并打开 Find and Replace 对话框。

(2) 选中 Using wildcards 复选框，并在 Find what 文本框中输入模式<gr[ae]y>。

(3) 单击三次 Find Next 按钮，并在每次单击后观察匹配或者没有匹配的文本。

图 11-13 显示了第一次单击 Find Next 按钮后的文本。

工作原理

字符序列 gray 匹配，是因为它是一个单词，而不仅仅是一个简单的字符序列。< 元字符匹配 gray 中 g 之前的位置。然后直接量 g 和 r 匹配。字符类 [ae] 包含了与测试文本中第三个字符 a 匹配的字符。最后，模式中的 y 匹配字符序列中最后的字符 y。在 y 匹配之后，> 元字符匹配了行的结束位置。

单词 grey 匹配的原因与上一段解释的相同。但是，这一次单词 greying 和 greyed 没有匹配。它们的前四个字符虽然与正则表达式的组件 <gr[ae]y 匹配，但在匹配 > 元字符时却失败了。因为 greying 和 greyed 中 y 后面的字符都是一个字母字符，不存在一个单词结束处的词边界位置。

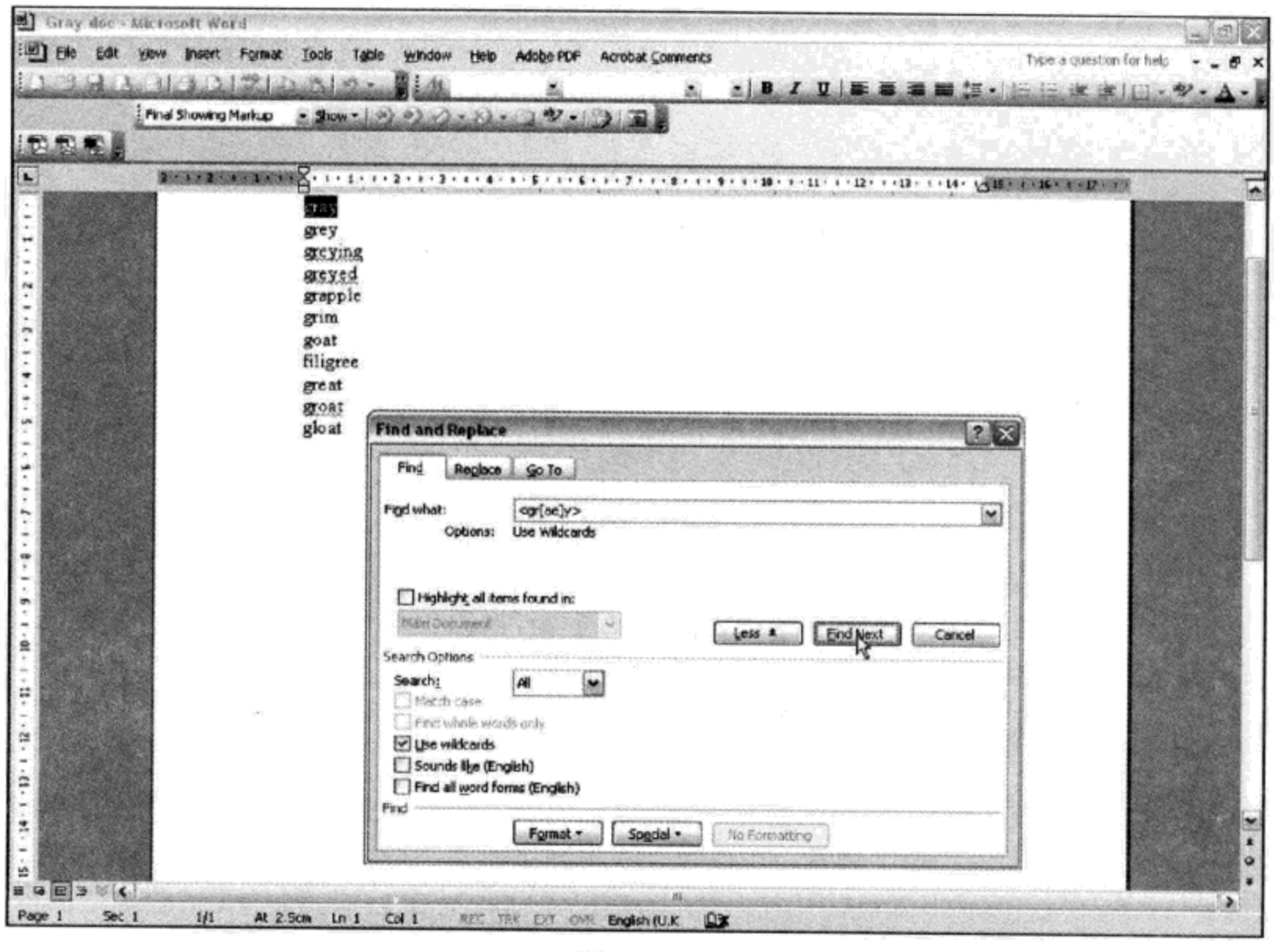


图 11-13

11.4 搜索和替换的例子

下面的例子示范了如何在搜索和替换操作中使用通配符。

11.4.1 使用反向引用改变名字的结构

在 Microsoft Word 中，可以使用通配符来改变文档中所有名字的结构。例如，初始的名字为：

```
Fred Smith
```

在使用通配符功能完成搜索和替换操作后，将变成：

```
Smith, Fred
```

完成这种操作需要用到反向引用。

测试文档 Names.doc 的内容如下所示：

```
Fred Smith
Alice Green
Barbara Kaplan
```

```
Ali Hussein
Paul Simenon
```

其中每个人的名字都是名后跟一个空格，再跟姓的形式。在查找操作中，这种结构的一致性非常重要。

试一试：改变名字的结构

(1) 在 Microsoft Word 中打开 Names.doc，使用 Ctrl+F 快捷键打开 Find and Replace 对话框，并单击 Replace 选项卡。

(2) 如果看不到 Find and Replace 窗口中的 Search Options 部分，单击 More 按钮便可看到。在其中选中 Use wildcards 复选框。

(3) 在 Find what 文本框中，输入模式 (<*>) (<*>)(确保在第一个圆闭括号和第二个圆开括号之间有一个空格符) 。

(4) 在 Replace with 文本框中，输入模式\2, \1(确保在模式中的逗号之后有一个空格符)。

(5) 单击 Find Next 按钮(Fred Smith 便会突出显示)。

(6) 单击 Replace 按钮。观察第一次执行替换操作后的结果(如图 11-14 所示。Fred Smith 被 Smith, Fred 替换)。

(7) 单击 Replace All 按钮，并观察结果，如图 11-15 所示。所有名字都应该变成了下面这种形式：

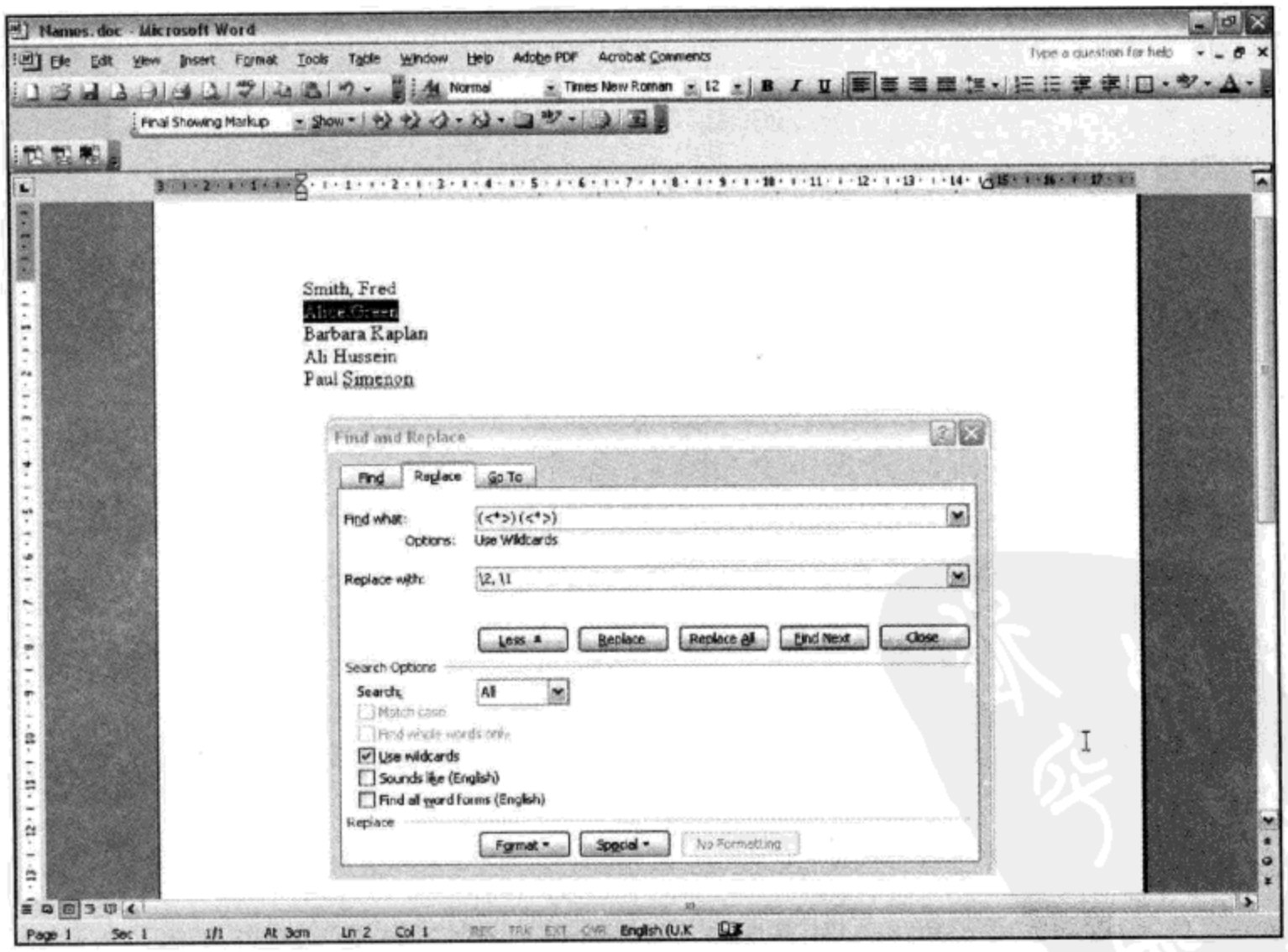


图　11-14

```
姓，名
```

工作原理

我们来分析一下模式 (<*>) (<*>)。其中的两对圆括号用于进行分组。< 元字符匹配字母字符序列的开始位置。* 元字符匹配零个或多个字符(等价于标准正则表达式语法中的 .*)。而 > 元字符匹配字母字符序列的结束位置和非字母字符序列的开始位置。因此，<*> 匹配一个任意长度的词。再把这个组件包含在圆括号中，就构成了组。

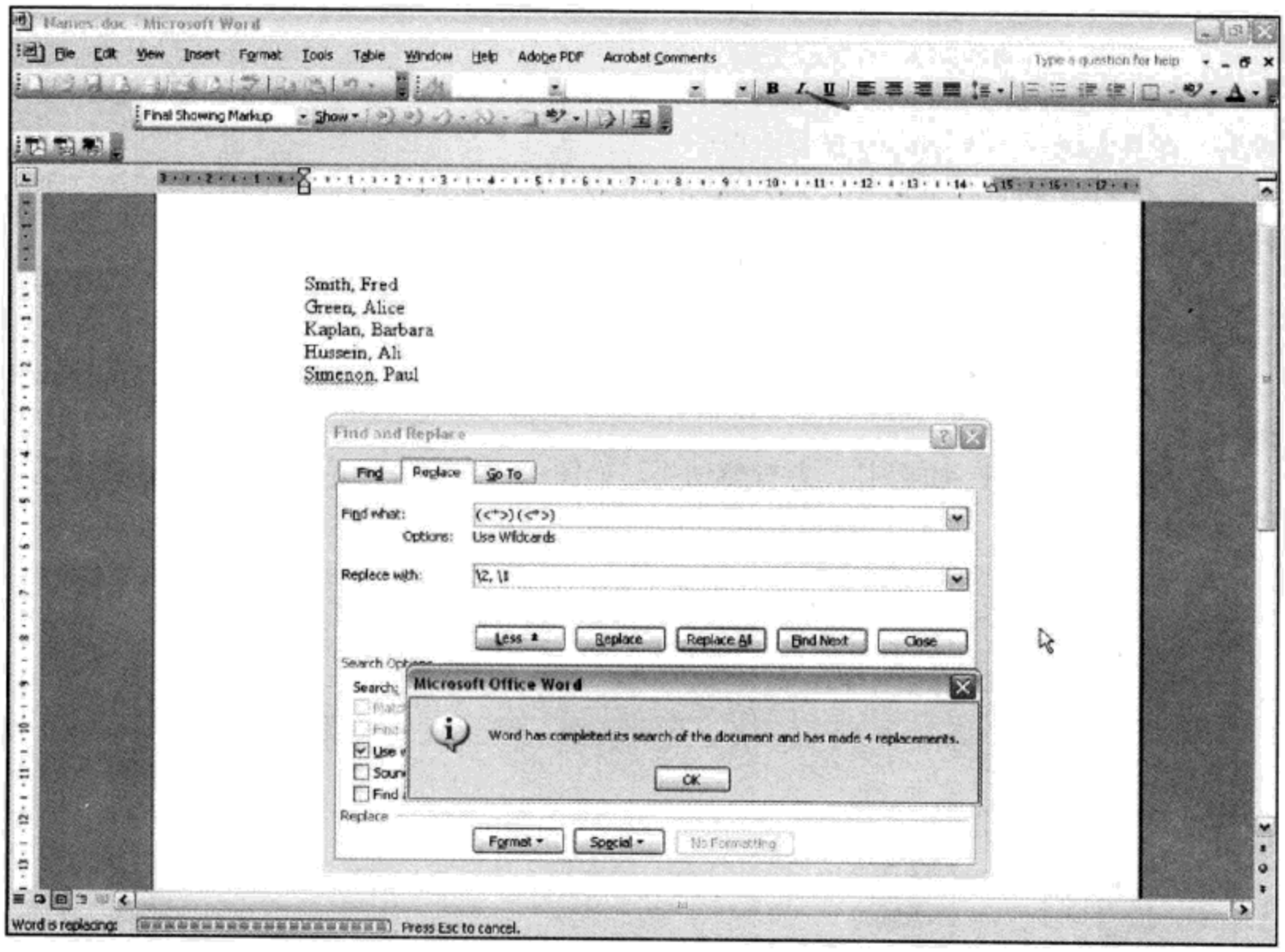


图 11-15

由于在第一和第二个 (<*>) 之间有一个空格符，所以模式：

```
(<*>) (<*>)
```

匹配由一个空格分隔开的两个词，这也正是 Names.doc 中的所有人名的结构形式。同时这个模式还会捕获两个组的匹配项——\1 和 \2，可以将其用于替换操作中。

圆括号创建了两个组：\1 包含空格符之前的词，而 \2 则包含空格符之后的词。因此，模式：

```
\2, \1
```

会将被一个空格符分隔的两个词替换为第二个词(由 \2 表示)后跟一个逗号和一个空格符，再后跟原先的第一个词(由 \1 表示)的形式。

11.4.2　操纵日期

正如不同英语方言中的单词 gray(grey) 和 license(licence)一样，不同语言中的日期结构之间也存在差别。在美国，数字形式的日期通常都写做 MM/DD/YYYY。而在英国，则把日期写成 DD/MM/YYYY 格式。在日本，使用的是 YYYY/MM/DD 格式。以上格式都是使用正斜杠作为分隔符，但是也可以使用连字符(短划线)或句点字符作为日期部件的分隔符。在这个例子当中，假定要操作的日期不会写成 25 December 2005、25 Dec 2005 或其他类似的格式。

本例使用测试文档 Dates.doc，它的内容如下：

```
2005-12-25
12/11/2003
01.25.2006
03-18-2007
07-19-2004
2006/09/18
```

这个测试文档中的日期有一些是国际通用的日期格式 YYYY-MM-DD，还有一些是美国的日期格式 MM/DD/YYYY(涉及几种不同的分隔符)。我们的任务是匹配其中的美国的日期格式，并将其替换成相应的国际通用格式。由于 XML 中使用的是国际通用的日期格式，所以这种格式很可能会被越来越多与日期相关的信息所采用。

我们假设其中没有英国的日期格式，因为第二行中的日期如果采取英国格式会变得不明确。在美国格式中，它的含义是 December 11, 2003，而在英国格式中，则会变成 12 November 2003。

试一试：将日期格式转换成国际通用的格式

(1) 在 Microsoft Word 中打开 Dates.doc。

(2) 使用 Ctrl+F 快捷键打开 Find and Replace 对话框，并选中 Use wildcards 复选框。

首先，我们要尝试创建一个正则表达式来匹配美国的日期格式。

(3) 在 Find what 文本框中输入模式([0-9]{2})[./-]([0-9]{2})[./-]([0-9]{4})，单击四次 Find Next 按钮，并观察突出显示的文本。

如果一切顺利，这个模式应该会匹配以美国格式表示的日期，而不会匹配国际通用格式的日期。我们也假定其中所有日期都是有效的——因为此处的模式也会匹配那些没有意义的字符序列，比如 44/12/5432。

图 11-16 显示了在第 3 步中单击了四次 Find Next 按钮之后的文本。

(4) 切换到 Find and Replace 对话框中的 Replace 选项卡，并在 Replace with 文本框中输入模式\3-\1-\2。

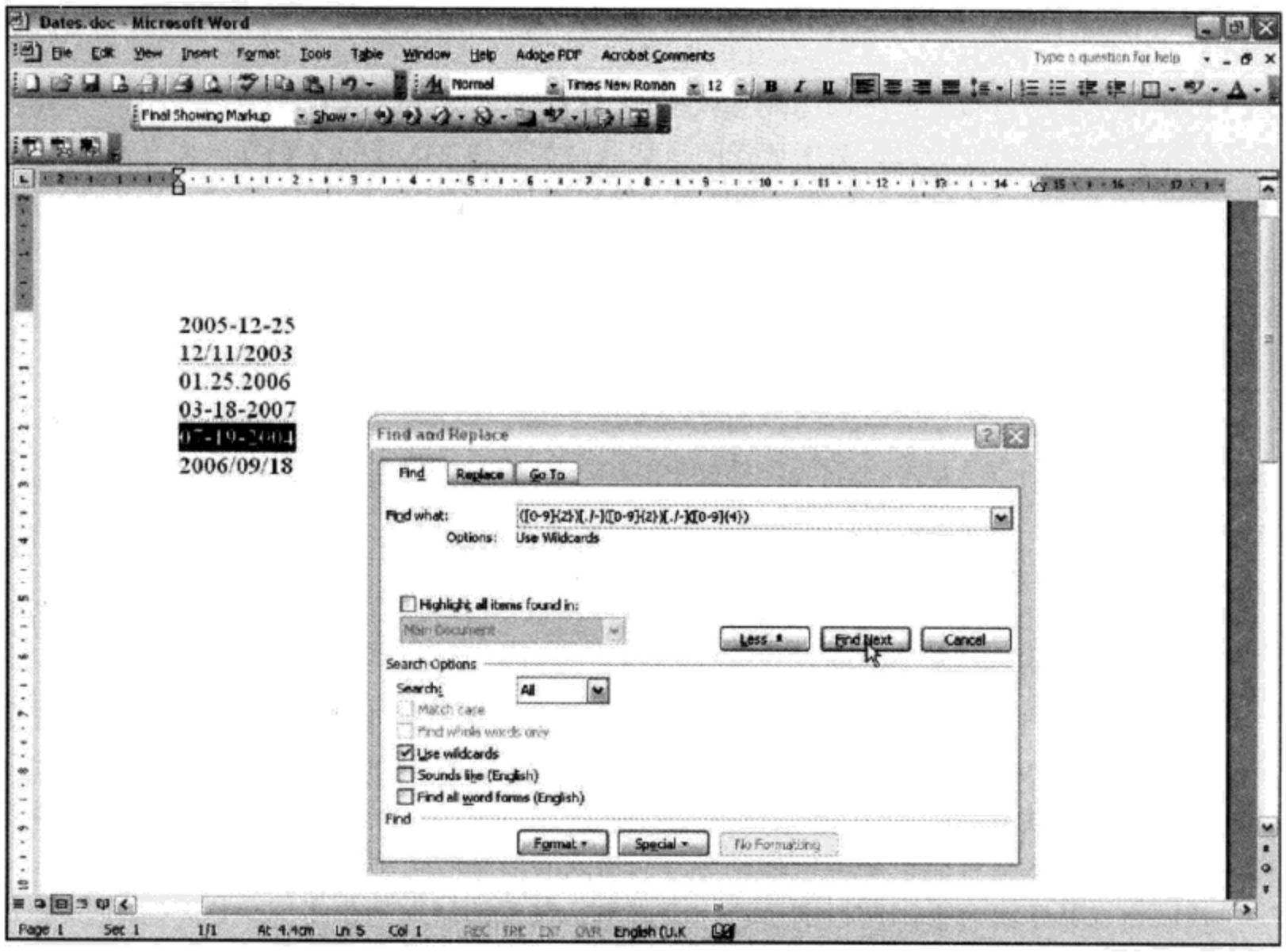


图 11-16

(5) 单击 Find Next 按钮，并单击 Replace 按钮。确认这次替换已经生成了正确格式的日期——即，国际通用格式的日期。

(6) 重复三次第 5 步，以替换所有美国格式的日期。图 11-17 显示了完成第 6 步后的文本。

工作原理

我们把模式([0-9]{2})[./-]([0-9]{2})[./-]([0-9]{4}) 分解成单个组件后就容易理解它的作用了。

组件 ([0-9]{2}) 匹配由两个连续数字组成的字符序列。这里必须使用字符类来指定要匹配的数字，因为 Microsoft Word 中没有专门的数字元字符。而圆括号是创建 \1 的一个组，该组包含着日期中的月份部件。

组件 [./-] 会匹配使用连字符、正斜杠或句点字符作为分隔符的日期。一定要注意别把这个字符类写成 [.-/]，因为写成这种形式后其中的连字符会创建范围，从而导致匹配不想要的结果。

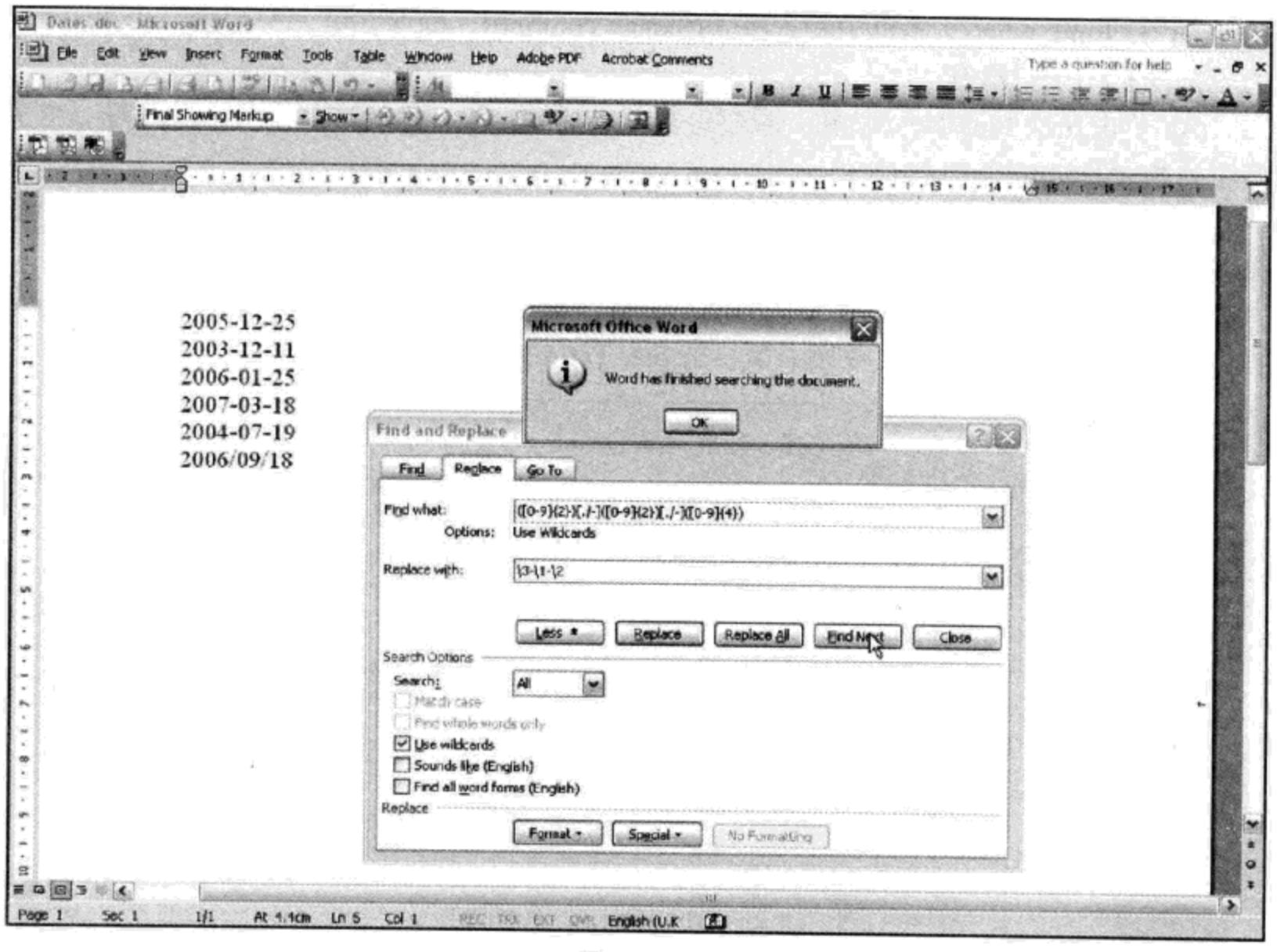


图　11-17

组件([0-9]{2})[./-] 与模式中前一个组件的作用相同，只不过这里的分组圆括号创建的是组 \2，其中包含着日期的天数部件。

而组件 ([0-9]{4}) 匹配四个数字的字符序列。这里的分组圆括号创建了组 \3，其中包含着日期的年份部件。

最后是替换模式 \3-\1-\2，它用于对日期的各个部件进行重新排序。其中包含着年份的 \3 组后跟一个直接量连字符，然后依次是包含月份部件的 \1 组、另外一个直接量连字符和包含日期中天数部件的 \2 组。

11.4.3　Star Training Company 的例子

Microsoft Word 没有向前查找功能，所以不能在 Word 中使用这个功能来完成对 Star Training Company 文档的搜索和替换操作。然而，我们却可以使用反向引用来达到相同的目的，尽管不如使用向前查找那样好用。

试一试：Star Training Company 的例子

(1) 在 Microsoft Word 中打开 StarOriginal.doc，并打开 Find and Replace 对话框。

(2) 切换到 Replace 选项卡。由于缺少交替选择(Word 中没有 | 元字符)，所以必须通过两次替换才能完成。

(3) 在 Find what 文本框中输入模式(Star)(Training)，确保在第二个圆括号中 Training 的 T 前面有一个空格符。

(4) 单击几次 Find Next 按钮，观察每次单击后的结果。确认这个模式会匹配每一个字符序列 Star Training。

(5) 在 Replace with 文本框中输入模式 Moon\2。小心不要在 Moon 和\2 之间插入空格符。因为组 \2 中已经保存了捕获的空格符，所以这里就不需要再添加了。

(6) 使用 Find Next 和 Replace 按钮搜索整个文档，并替换其中的每一个字符序列 Star Training。图 11-18 显示了在替换了第一个 Star Training 后的屏幕外观。

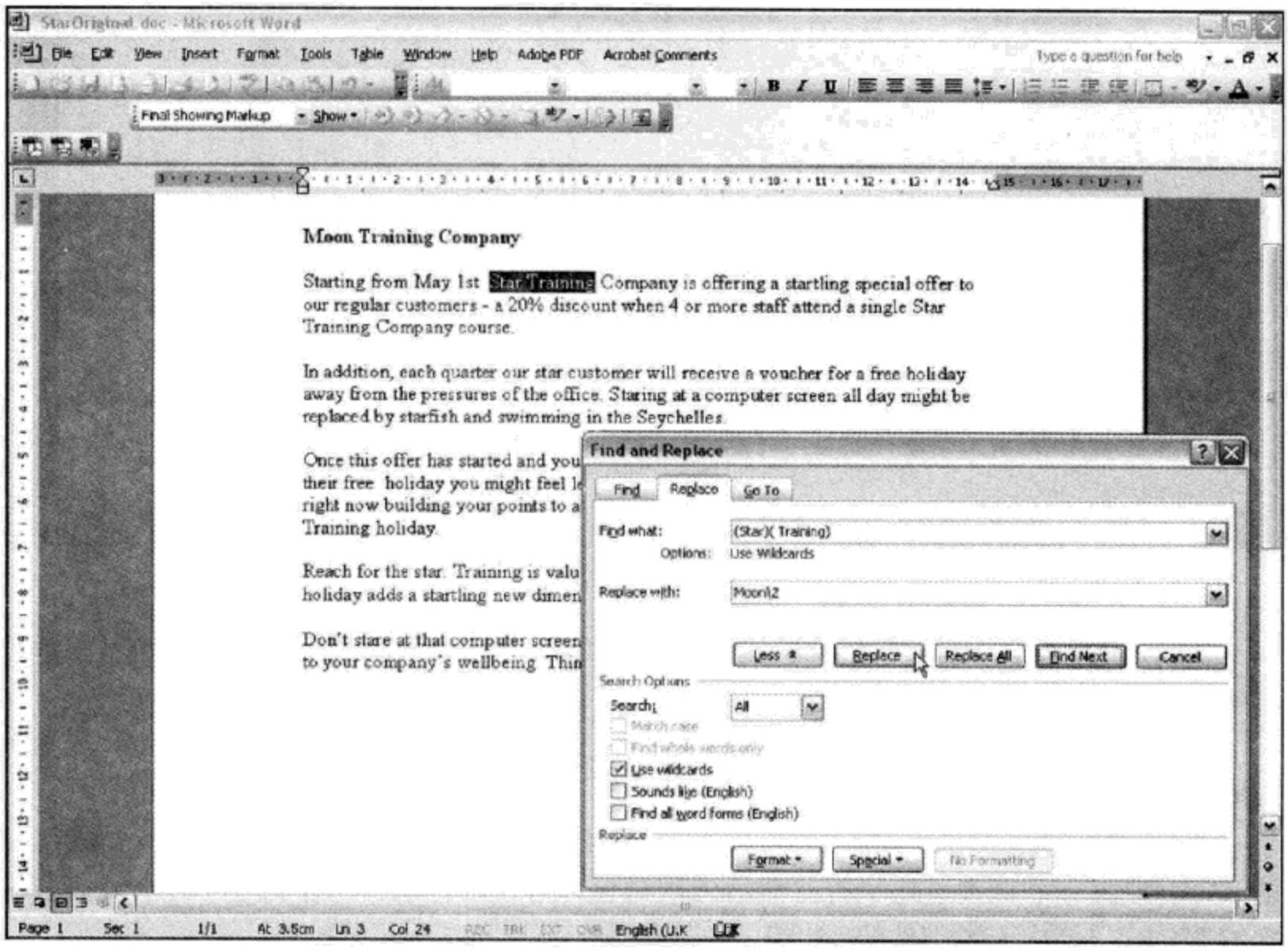


图 11-18

(7) 在 Find what 文本框中把模式修改为(Star)([.])。这个稍微反常的模式在 Word 中是必要的，稍后会解释为什么。

(8) Replace with 文本框中的替换模式不用修改。通过单击 Find Next 和 Replace 按钮用 Moon.来替换 Star.。图 11-19 显示了在第一个字符序列 Star. 被替换后的文档外观。

工作原理

首先，我们来看一下模式 (Star)(Training) 是如何匹配的。这个模式创建了两个组，其中第一个组是(Star)，而第二个组中是一个空格符后跟字符序列 Training。通过把捕获的组 \2 简单地放在替换模式中，就弥补了缺少向前查找功能的不足。这样，字符序列 Star 会被

Moon 替换，而 \2 则被自身替换(保持不变。译者注)。

模式(Star)([.]) 也创建了两个组。第一个组中包含了字符序列 Star，而第二个组中则包含句点字符。在多数正则表达式实现中，你可能不会把这个句点字符放在字符类中，而是使用 \. 转义序列来代替。然而，在 Word 中必须把这个句点字符放入字符类中，以避免触发错误提示。最后，位于一个句点字符之前的 Star 会被 Moon 替换。

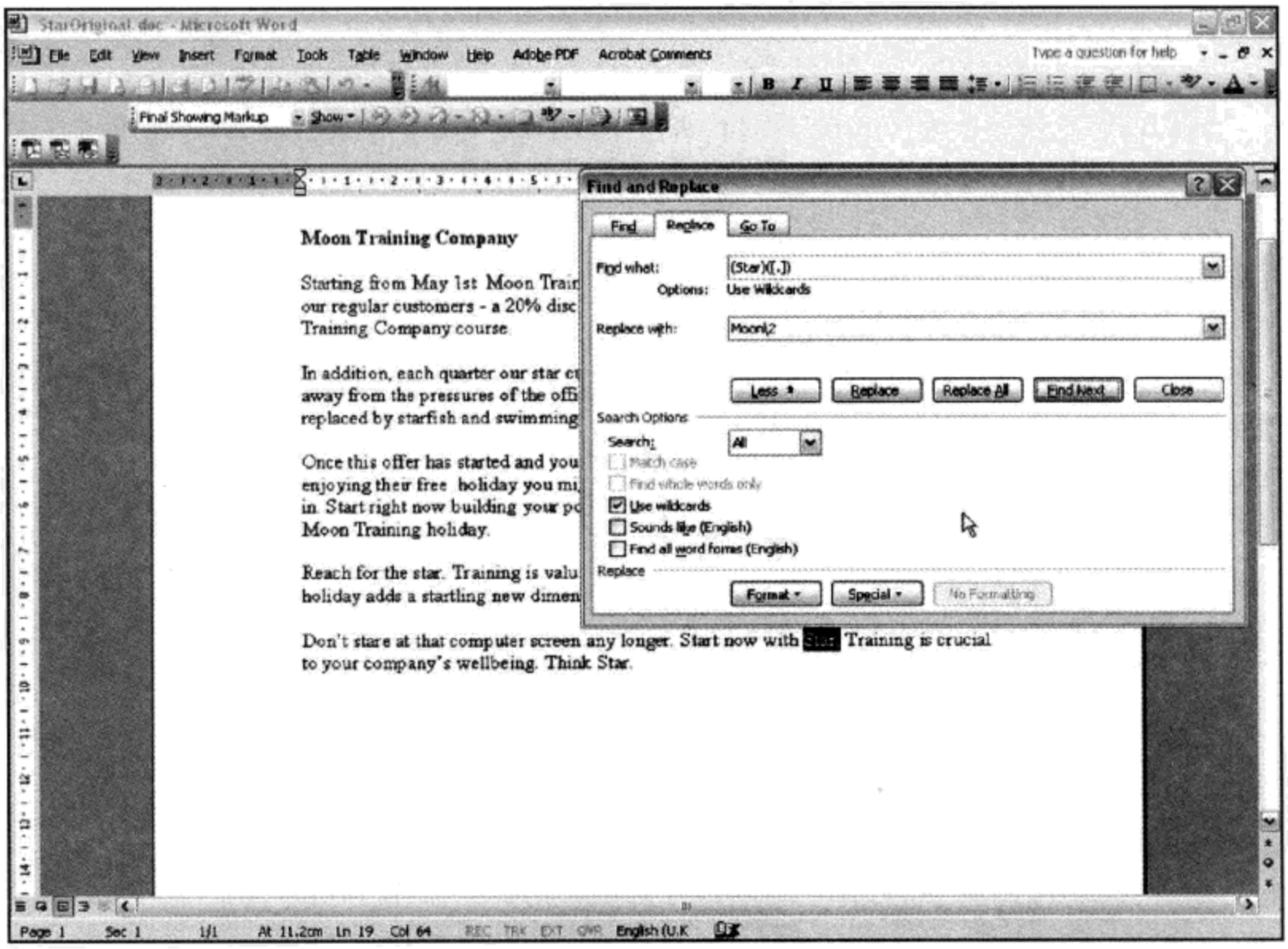


图　11-19

在通过我们刚才介绍的两步操作将字符序列 Star 替换之后，结果文档 StarWhichIsNowMoon.doc 的内容变成如下所示：

```
Moon Training Company

Starting from May 1st Moon Training Company is offering a startling special offer
to our regular customers - a 20% discount when 4 or more staff attend a single Moon
Training Company course.

In addition, each quarter our star customer will receive a voucher for a free
holiday away from the pressures of the office. Staring at a computer screen all day
might be replaced by starfish and swimming in the Seychelles.

Once this offer has started and you hear about other Moon Training customers
enjoying their free holiday you might feel left out. Don't be left on the outside
```

```
staring in. Start right now building your points to allow you to start out on your
very own Moon Training holiday.

Reach for the star. Training is valuable in its own right but the possibility of a
free holiday adds a startling new dimension to the benefits of Moon Training
training.

Don't stare at that computer screen any longer. Start now with Moon. Training is
crucial to your company's wellbeing. Think Moon.
```

我们的确是替换了所有相关的 Star 实例。对于第 1 章中提到的新手而言，能够完成如此高灵敏度和高特殊性的搜索替换操作，无疑是一次较大的进步。

11.5 VBA 中的正则表达式

VBA(Visual Basic for Applications)能够以编程的方式实现与在 Microsoft Word 界面中类似的正则表达式(通配符)功能。下面 Word 的宏 StarToMoon 也能够实现同样的意图：

```
Sub StarToMoon()
'
' StarToMoon Macro
' Macro recorded 19/07/2004 by Andrew Watt
'
    Selection.Find.ClearFormatting
Selection.Find.Replacement.ClearFormatting
With Selection.Find
      .Text = "(Star)( Training)"
      .Replacement.Text = "Moon\2"
      .Forward = True
      .Wrap = wdFindContinue
      .Format = False
      .MatchCase = False
      .MatchWholeWord = False
      .MatchAllWordForms = False

      .MatchSoundsLike = False
      .MatchWildcards = True
End With
Selection.Find.Execute Replace:=wdReplaceAll
With Selection.Find
      .Text = "(Star)(.)"
      .Replacement.Text = "Moon\2"
      .Forward = True
      .Wrap = wdFindContinue
      .Format = False
      .MatchCase = False
      .MatchWholeWord = False
```

```
        .MatchAllWordForms = False
        .MatchSoundsLike = False
        .MatchWildcards = True
    End With
    Selection.Find.Execute
    With Selection
        If .Find.Forward = True Then
            .Collapse Direction:=wdCollapseStart
        Else
            .Collapse Direction:=wdCollapseEnd
        End If
        .Find.Execute Replace:=wdReplaceOne
        If .Find.Forward = True Then
            .Collapse Direction:=wdCollapseEnd
        Else
            .Collapse Direction:=wdCollapseStart
        End If
        .Find.Execute
    End With
    With Selection
        If .Find.Forward = True Then
            .Collapse Direction:=wdCollapseStart
        Else
            .Collapse Direction:=wdCollapseEnd
        End If
        .Find.Execute Replace:=wdReplaceOne
        If .Find.Forward = True Then
            .Collapse Direction:=wdCollapseEnd
        Else
            .Collapse Direction:=wdCollapseStart
        End If
        .Find.Execute
    End With
End Sub
```

教你如何编写 Word 中的宏并不是本章的目的，但是在 Word 的宏里面包含查找和替换功能却是非常有用的。

测试文档 StarOriginalWithMacro.doc 中包含了前面的宏，可以打开该文档执行这个宏。

文档 StarAfterMacro.doc 中包含着使用宏进行文本替换后的内容。

要运行 StarToMoon 宏，可能需要调整 Word 中与宏有关的安全设置。

请注意，创建宏的计算机应该是没有病毒的。然而，最好还是用你自己的防病毒软件检测一下以确保其中不会包含病毒。

11.6 练习

下面的练习题可以让你测试自己对本章讲述内容的理解情况：

1. 请在 Word 中创建一个能够匹配字符序列 peak 和 peek 的模式。

2. 请修改日期例子中的模式，使其能够将英国格式的日期转换成国际通用格式的日期。

第12章

在 StarOffice/OpenOffice.org Writer 中使用正则表达式

多年来，Microsoft Office 一直是 Windows 平台中一款卓越的办公系统。最近几年，由于价格优势等原因，StarOffice 和 OpenOffice.org 办公套件变得日益流行起来，大有将 Microsoft Word 和 Microsoft Excel 取而代之之势。本章主要介绍在 OpenOffice.org 文字处理程序 OpenOffice.org Writer 套件中使用正则表达式的内容。

OpenOffice.org 对正则表达式的支持要好于 Microsoft Word。OpenOffice.org Writer 不仅支持更多的元字符，而且使用的正则表达式语法也比 Microsoft Word 更标准。当然，选择这些流行的文字处理程序时，这些功能上的差别只是诸多考虑因素中的一个而已。

OpenOffice.org Writer 中非常有用的一个功能就是它的 Find All 按钮。在本书前面章节的例子中，已经多次用到过 Find All 按钮了。通过这个按钮，可以更方便快捷地找到与正则表达式模式匹配的字符序列。在少数情况下(本章后面会讨论到)，对于 Find All 按钮返回的结果，还需要仔细地加以分析。

OpenOffice.org 1.0 版不支持正则表达式。如果要使用正则表达式功能，则必须使用 OpenOffice.org 1.1 或更高版本。

在本章中将学习以下内容：

- 如何通过 OpenOffice.org Writer 的界面在搜索和替换文本时使用正则表达式
- OpenOffice.org Writer 支持的正则表达式语法有哪些
- 如何在 OpenOffice.org Writer 中完成搜索和搜索替换操作
- 如何使用 OpenOffice.org Writer 支持的 POSIX 字符类

12.1 用户界面

OpenOffice.org Writer 的用户界面与 Microsoft Office 的用户界面存在许多不同之处。不过，与 Word 中一样，用 Ctrl+F 快捷键可以调出如图 12-1 所示的 Find & Replace 对话框。

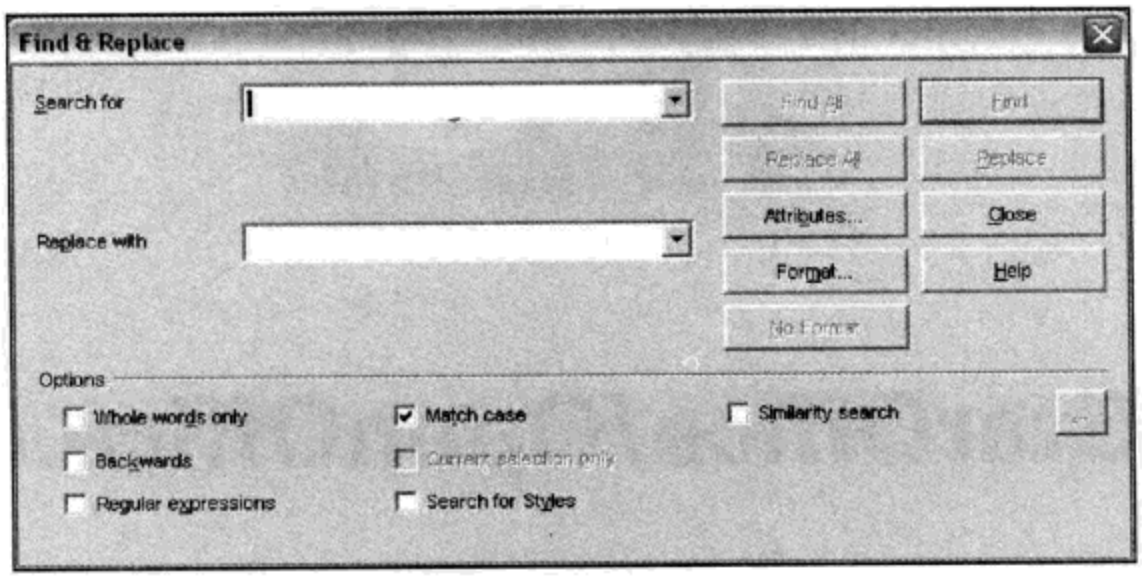


图 12-1

要在 OpenOffice.org Writer 中使用正则表达式，必须选中这个对话框左下角的 Regular expressions 复选框。OpenOffice.org 在默认情况下执行的是不区分大小写的正则表达式搜索。如果要进行区分大小写的正则表达式搜索，必须在选中 Regular expressions 复选框的同时再选中 Match case 复选框。根据之前使用 OpenOffice.org Writer 的情况，当打开 Find & Replace 对话框时，Match case 复选框可能已经被选中了。如果希望完成默认的不区分大小写的匹配，而又没注意到 Match case 复选框被选中，那么就可能出现不想要的匹配结果。第一次打开 Find & Replace 对话框时，Regular expressions 复选框不会被选中。但是，如果打开好几个 OpenOffice.org 窗口，那么在一个文档中的 Find & Replace 设置就有可能带到另一个文档中。

在选中 Regular expressions 复选框的情况下，Whole words only 复选框就会变灰。因为 OpenOffice.org Writer 支持匹配词的开始位置和结束位置的元字符(本章后面会介绍到)，因而可以使用正则表达式来实现只匹配完整的词(更严格来讲，应该是匹配完整的字母字符序列)。

在前面几章的例子中已经看到过，OpenOffice.org Writer 的 Find All 按钮对于查找某个特定正则表达式模式的所有匹配项是非常方便的。然而，在某些情况下则必须仔细解读由 Find All 按钮找到的结果。例如，有多个连续的数字，它们位于下面测试文本(Numbers.txt)的某些行中：

```
A123

2345

9876

12ABC345

999
```

单击 Find All 按钮后，Search for 文本框中的模式 [0-9]+ 会使所有数字匹配，如图 12-2 所示。这说明，使用 Find All 按钮无法分清匹配是贪婪的还是懒惰的——换句话说，难以分辨出一次单独的匹配，匹配的是一个数字还是一串数字。

如果不知道 OpenOffice.org Writer 的匹配总是贪婪的话，就无法确定地解释这些匹配项。而如果不能确定一个模式实际匹配的是什么，那么连续单击 Find 按钮则有助于确认该模式在任何条件下的匹配情况。

这种不确定性只存在于使用简单模式的情况下。对于大多数(也可能是全部)现实中使用的模式而言，都不会存在这种问题。

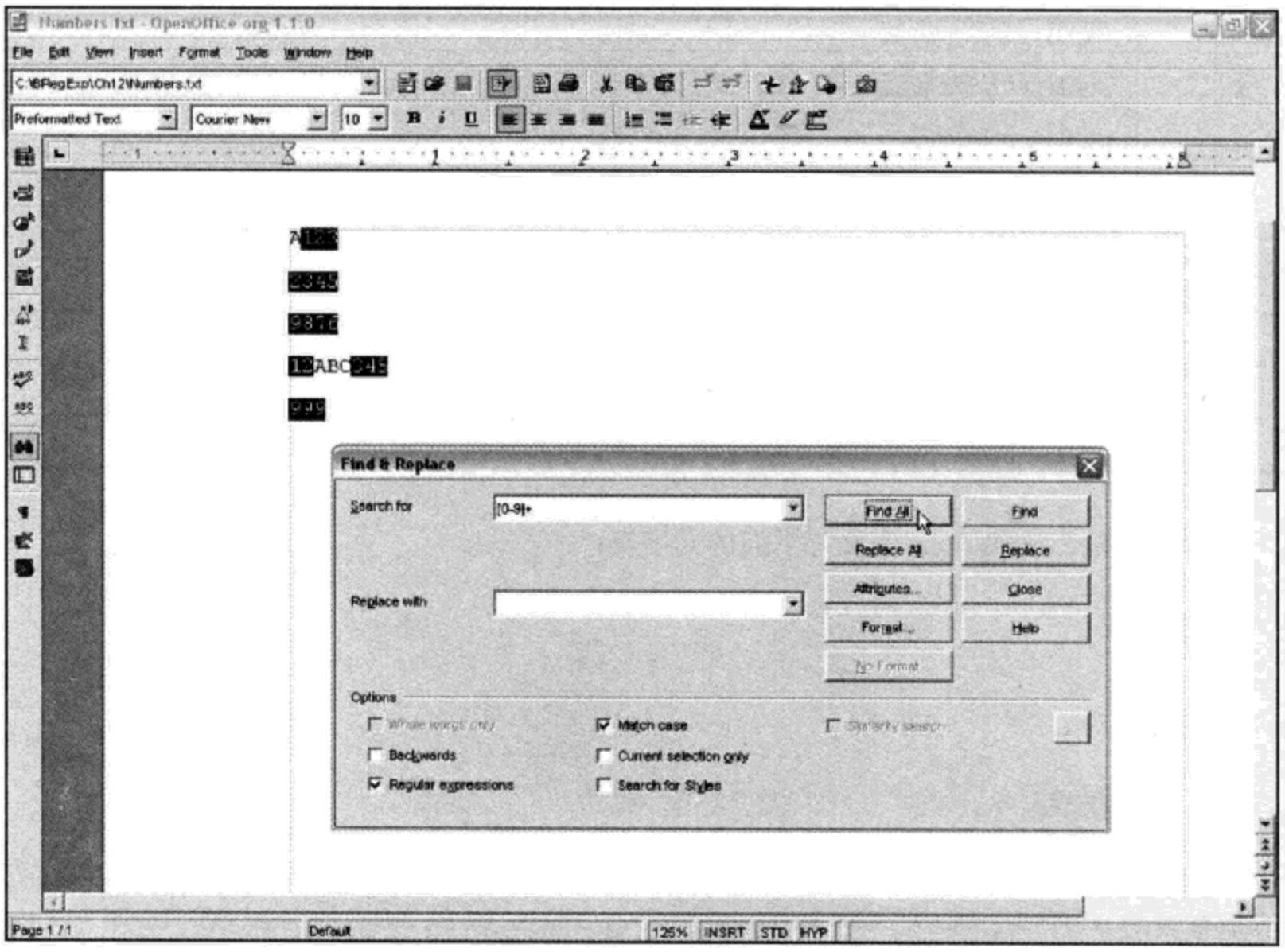


图　12-2

12.2　可用的元字符

OpenOffice.org Writer 中使用的元字符与 Microsoft Word 中使用的元字符有类似的地方，但与 Word 中的那些通配符不同。这些元字符也不同于其他语言(如 Perl)中用于正则表达式的元字符。表 12-1 总结了 OpenOffice.org Writer 支持的元字符。有关 OpenOffice.org Writer 支持的 POSIX 字符类的内容将在后面的一节中单独介绍。

表 12-1 中用到的术语“块”表示一个单独的字符或者一个包含在圆括号中的组。

表 12-1　OpenOffice.org Writer 支持的元字符及其说明

元字符	说　明
	几乎匹配任何字符，包括许多非英语符号和字符。可以单独使用，也可以与 ?、* 或 + 限定符结合使用
?	表示零个或一个前面块的限定符
*	表示零个或多个前面块的限定符
+	表示一个或多个前面块的限定符
\n	非标准用法。只匹配用 Shift+Enter 创建的一个换行符
^	匹配行开始的位置
$	匹配行结尾的位置
\t	匹配一个制表符
\<	匹配词开始的位置
\>	匹配词结束的位置
&	非标准元字符。类似于反向引用
[]	字符类
|	交替选择。匹配 | 元字符前面或后面的块
{n,m}	限定符语法

OpenOffice.org Writer 不支持下面的元字符：

- \b
- \s
- \w
- \d

12.2.1　限定符

OpenOffice.org Writer 中使用的是标准的限定符语法，即 ?、*、+ 和 {n,m} 限定符都是有效的。

测试文件 AandSomeBs.txt 的内容如下：

```
ABC

ABBC

AC

A3C

ABBBBBBC

AbbCC
```

模式 AB?C 可以匹配字符序列 ABC 和 AC，如图 12-3 所示。每个匹配的字符序列都由一个大写的 A 后跟零个或一个大写的 B，再跟一个大写的 C 组成。

如果把模式修改为 AB*C，则匹配的字符序列是 ABC、ABBC、AC 和 ABBBBBBC。每个匹配的字符序列都由一个大写的 A，后跟零个或多个大写的 B，再跟一个大写的 C 组成。

如果把模式修改为 AB+C，则字符序列 AC 不再匹配，因为它不包含一个大写的 B。模式 AB+C 的含义是“匹配一个大写的 A，后跟一个或多个大写的 B，再跟一个大写的 C”。

如果再把模式修改为 AB{2,4}C，那么就只有字符序列 ABBC 匹配。因为它是测试文本中唯一一个由一个大写的 A ，后跟二个到四个大写的 B，再跟一个大写的 C 组成的字符序列。

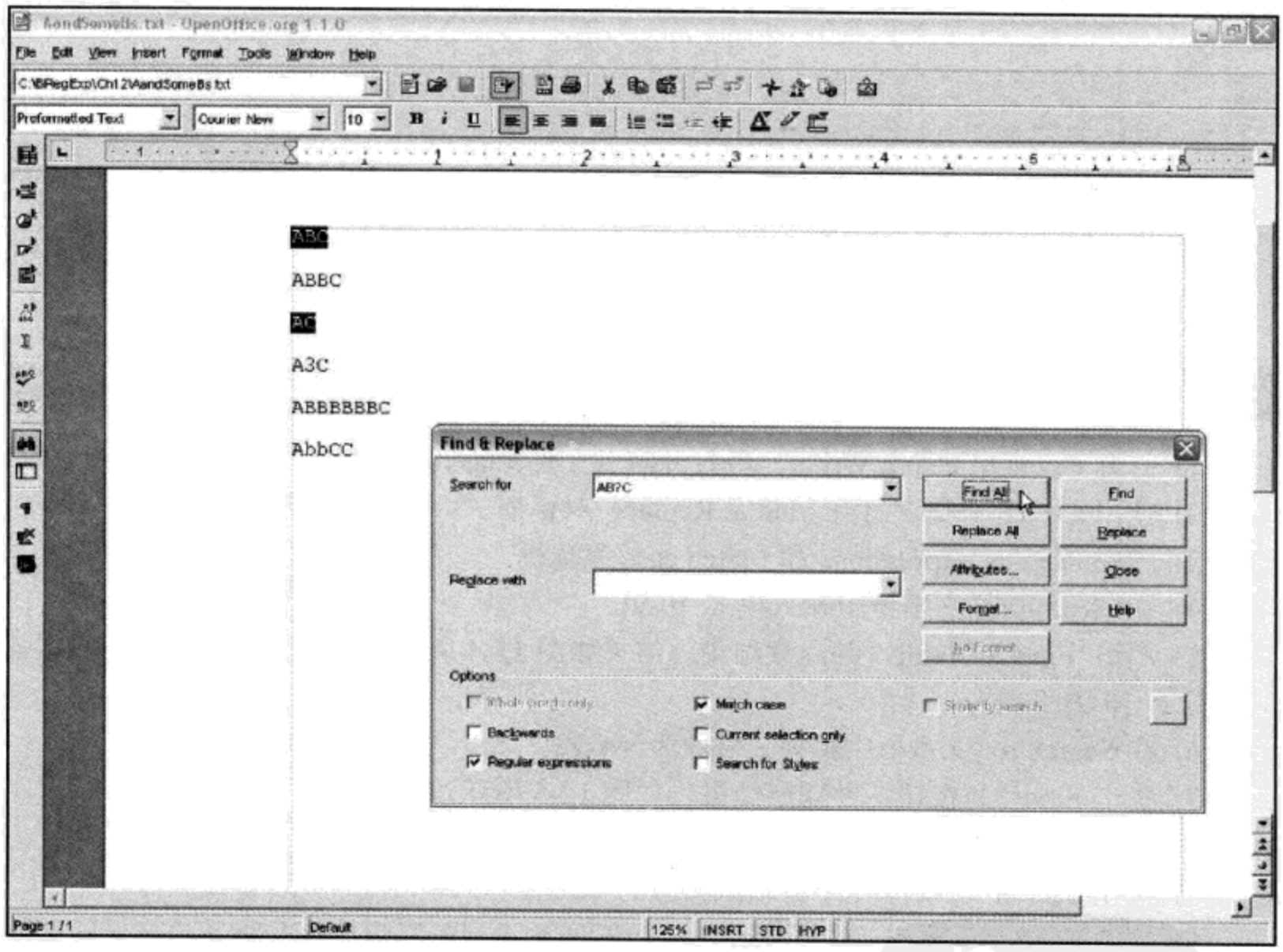


图　12-3

12.2.2　模式

OpenOffice.org Writer 支持不区分大小写(默认)的匹配，也支持区分大小写的匹配。Match Case 复选框就是控制在匹配中所使用模式的界面工具。

12.2.3　字符类

OpenOffice.org Writer 中实现的字符类是相当标准的。既支持范围，也支持对字符类取

反。

OpenOffice.org Writer 不支持匹配数字的 \d 元字符，也不支持匹配字母字符的 \w 元字符。因此，正则表达式开发人员在匹配这些字符时，必须使用相应的字符类，比如用[0-9]匹配数字，用 [A-Za-z] 匹配大小写形式的字母字符。而且，这些字符类也可以由前面“限定符”一节中提到的任何限定符来限定匹配次数。

ClassTest.txt 是下面“试一试”中要用的测试文件，其内容如下：

```
AB1

RD2

K9

993ABC

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

0123456789
```

试一试：字符类

(1) 打开 OpenOffice.org Writer，并打开测试文件 ClassTest.txt。

(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 在 Search for 文本框中输入模式 [0-9]。

(5) 单击 Find all 按钮，并观察结果。结果如图 12-4 所示，测试文本中的所有数字都与字符类 [0-9] 匹配。

(6) 在 Search for 文本框中将模式修改为 [A-Z]。

(7) 单击 Find All 按钮，并观察结果。如图 12-5 所示，测试文本中的所有大写字母字符都与字符类 [A-Z] 匹配。

(8) 再在 Search for 文本框中将模式修改为 [a-z]。

(9) 单击 Find All 按钮，并观察结果。此时所有小写的字母字符作为新模式的匹配项都被突出显示。

(10) 取消对 Match case 复选框的选定。

(11) 单击 Find All 按钮，并观察结果。结果如图 12-6 所示，所有大小写字母作为匹配项全部被突出显示。

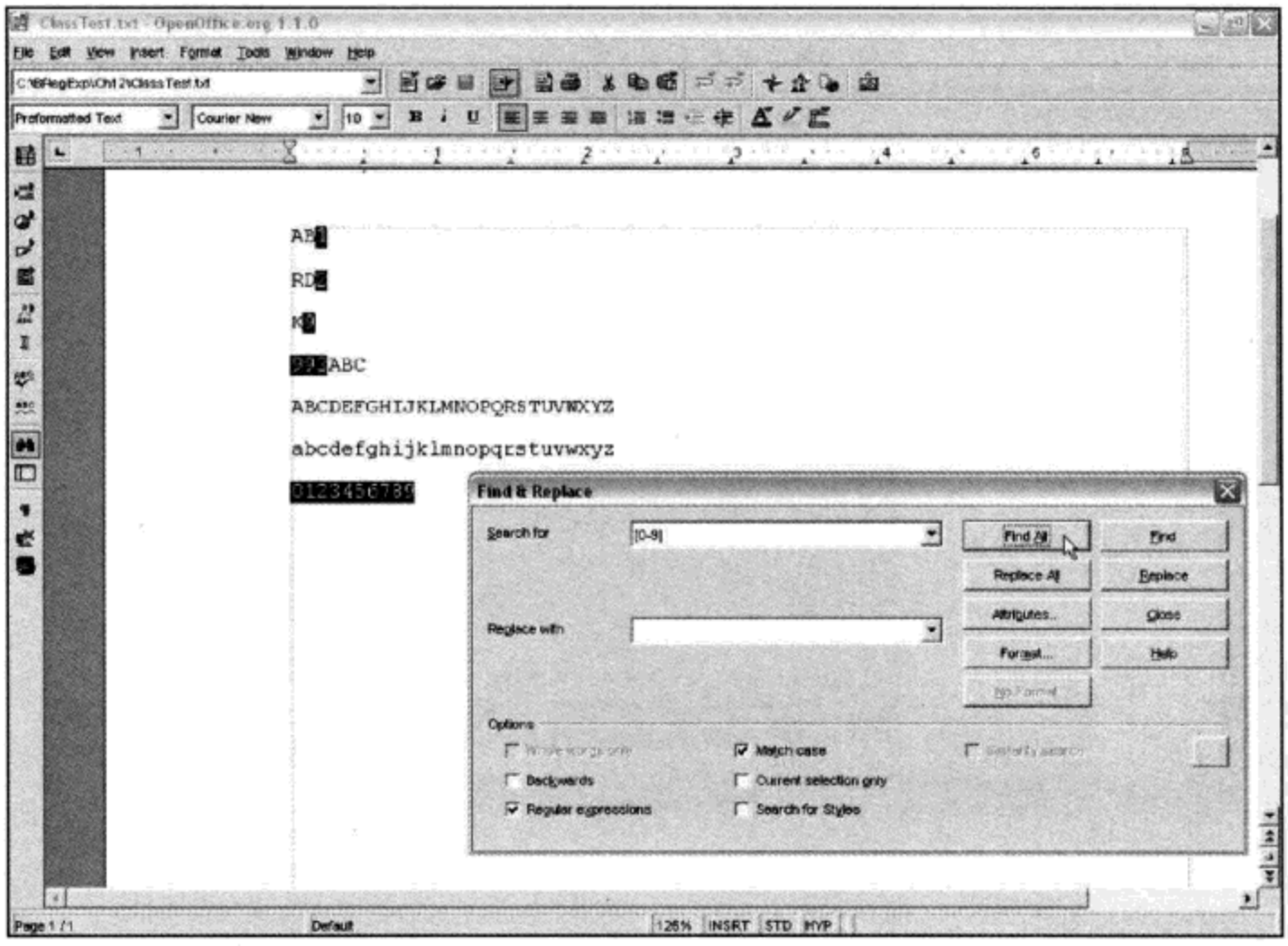


图　12-4

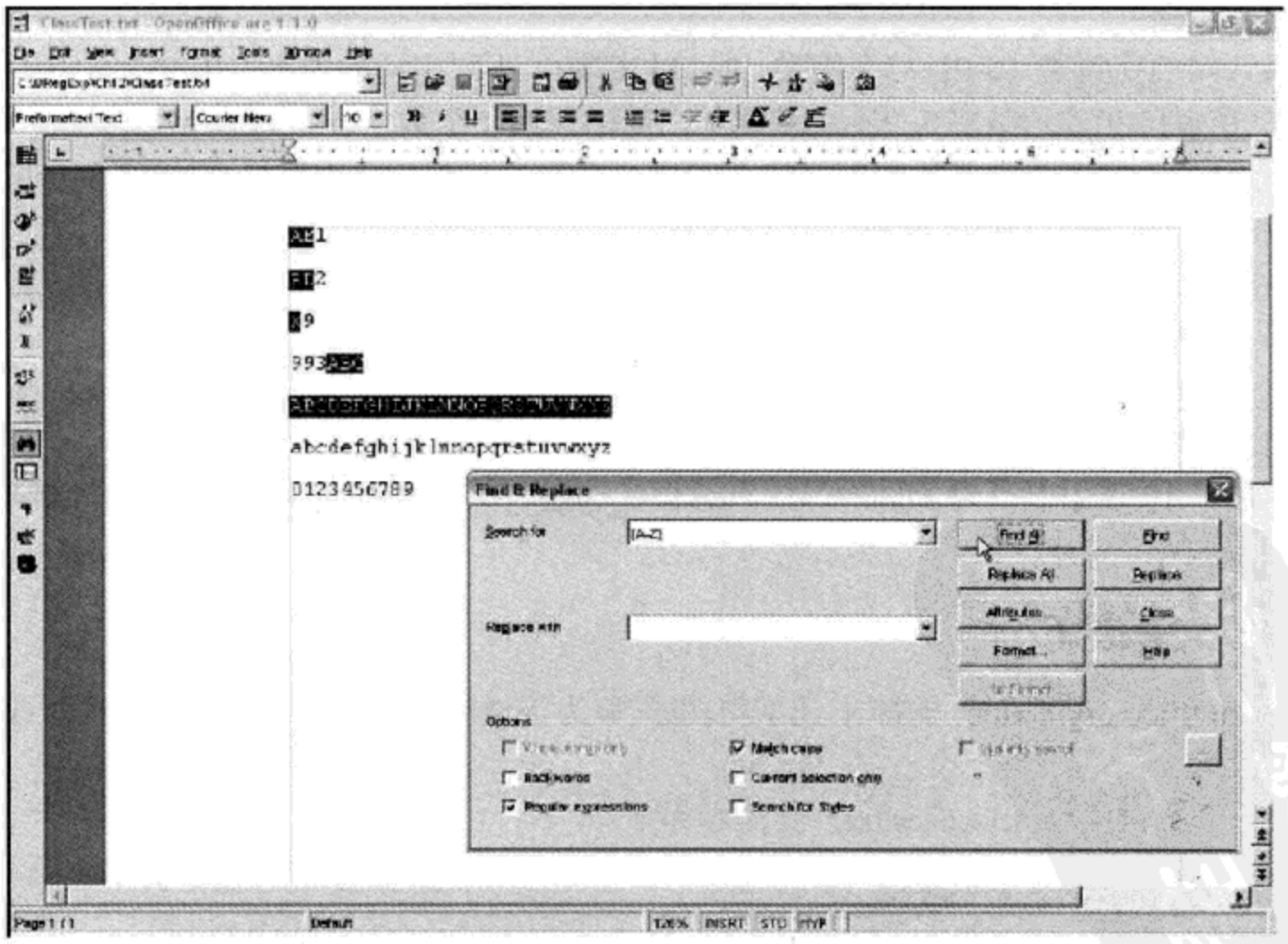


图　12-5

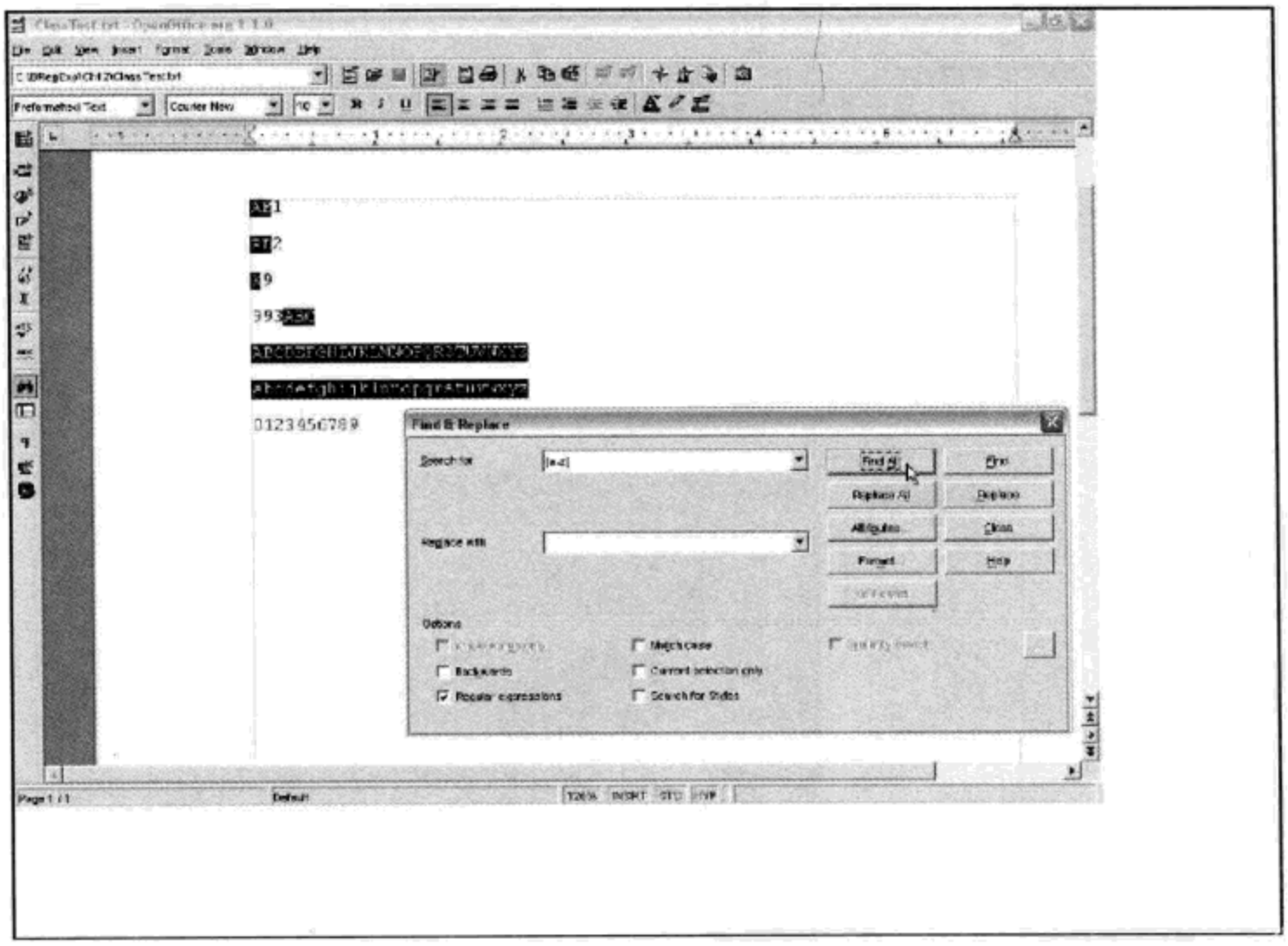


图 11-6

工作原理

最开始的字符类 [0-9] 与字符类 [0123456789] 匹配相同的内容。[0-9]中的短划线表示范围，所以[0-9]表示数字0~9(包括0和9)之间的任何一个数字。由于OpenOffice.org Writer不支持 \d 元字符(\d 在多数正则表达式实现中都用来匹配任意数字)，所以需要使用字符类来匹配数字。

类似地，字符类[A-Z]与[ABCDEFGHIJKLMNOPQRSTUVWXYZ]的含义相同，但更简洁。因为选中了 Match case 复选框(参见第 3 步)，所以只有大写的字母被匹配。

字符类[a-z]与[abcdefghijklmnopqrstuvwxyz]也匹配相同的内容。在选中 Match case 复选框的情况下，只有小写字母会被匹配。

当在第 10 步中取消对 Match case 复选框的选定时，模式[a-z]会匹配所有字母字符，包括大写和小写。

12.2.4 交替选择

OpenOffice.org Writer 支持 | 元字符(也常称为管道字符)，该字符表示交替选择或者逻辑或的意思。

有一个测试文档 Licenses.txt，其内容如下：

```
This licence has expired.

Friday is the day that the licensing authority meets.
```

```
Licences are essential before you can do that legally.

License is morally questionable.

Licensed practitioners only should apply.
```

我们的目标是匹配所有 licence、license 或 licensing，而另一种形式 licencing(这个测试文档中没有)的可能性也要考虑到。

这里的问题定义可以宽泛地表达如下：

匹配单词 licence 或 licensing，并考虑每个单词可能的拼写变体。

对这个问题定义改进后，则可以表达如下：

以不区分大小写的方式匹配直接量字符序列 l、i、c、e、n，后跟 c 或 s，然后再跟 e 或字符序列 i、n 和 g。

在不区分大小写的前提下，满足以上问题定义的模式可以如下：

```
licen(c|s)(e|ing)
```

下面我们就来试一试。

试一试：交替选择

(1) 打开 OpenOffice.org Writer，并打开测试文件 Licenses.txt。

(2) 选中 Regular expressions 复选框。因为要进行不区分大小写的匹配，所以要保证 Match case 复选框未被选中。

(3) 在 Search for 文本框中，输入模式 licen(c|s)(e|ing)。

(4) 单击 Find All 按钮，并观察突出显示的文本。图 12-7 显示的是执行第 4 步后的文档外观。

在结果中可以看到，分别位于 Licensed 和 Licenses 结尾处的 d 和 s 都没有匹配。如果希望匹配完整的单词，那么可以通过修改模式来实现。当字符序列 licens 或 licenc 后跟一个 e 时，我们可以允许后跟一个可选的 d 或 s。这样，licence、license、licenced、licensed、licences 和 licenses 就都匹配了。然而，当匹配项是 licensing 或 licencing 时，就不希望匹配后面跟字母 s 或 d。

所以，问题定义可以修改成：

以不区分大小写的方式匹配直接量字符序列 l、i、c、e、n 后跟 c 或 s，然后再后跟 e 及 d 或 s——其中 d 或 s 都是可选的；或者再后跟字符序列 i、n 和 g。

此时，将模式修改为 licen(c|s)(e(s|d)?|ing)来表达上面的问题定义。

从前面的问题定义和模式中可以看到，要明确地表达嵌套的选项比较困难。

(5) 将 Search for 文本框中的模式修改为 licen(c|s)(e(s|d)?|ing)。

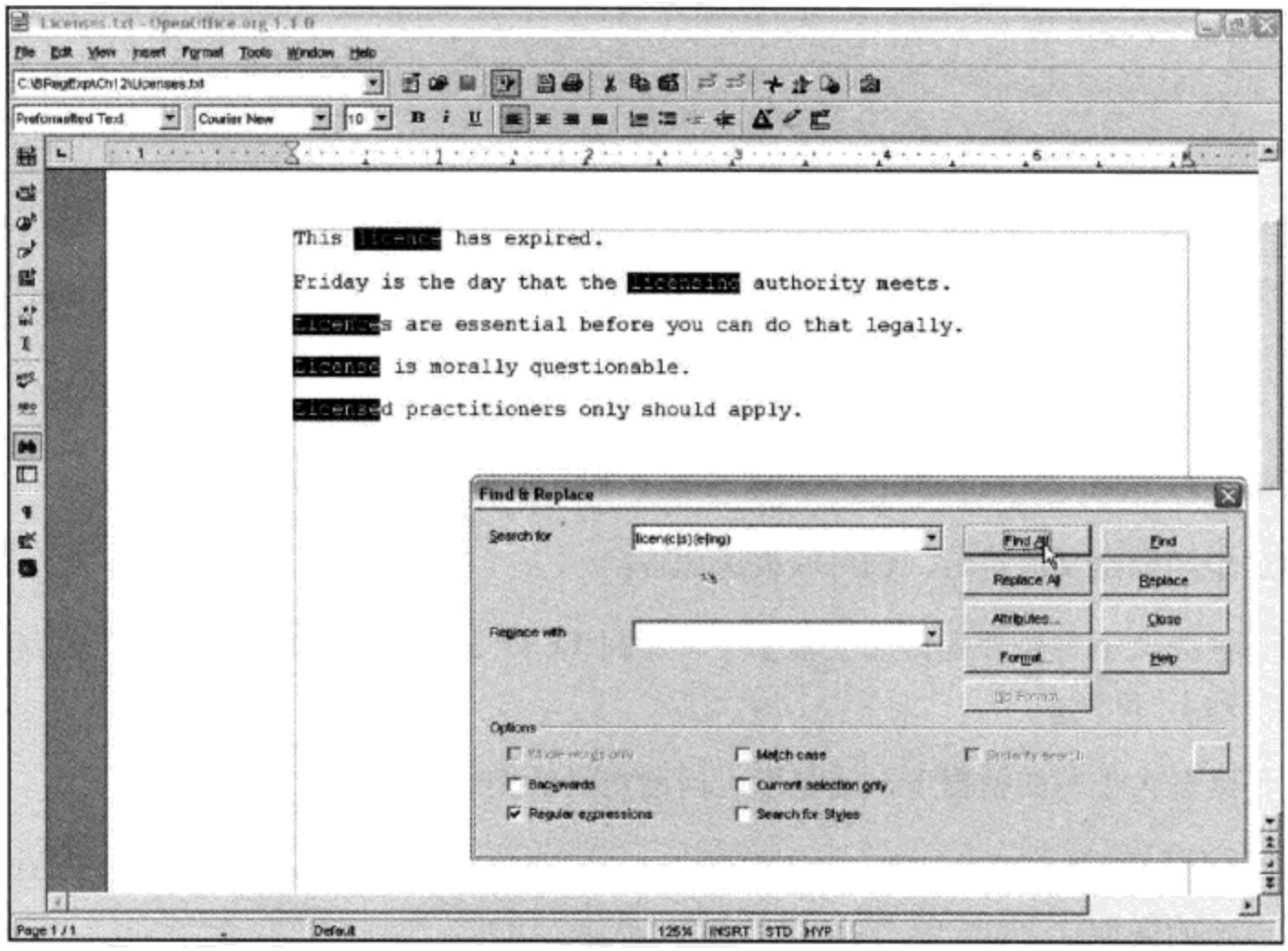


图 12-7

(6) 单击 Find All 按钮，并观察如图 12-8 所示的结果。

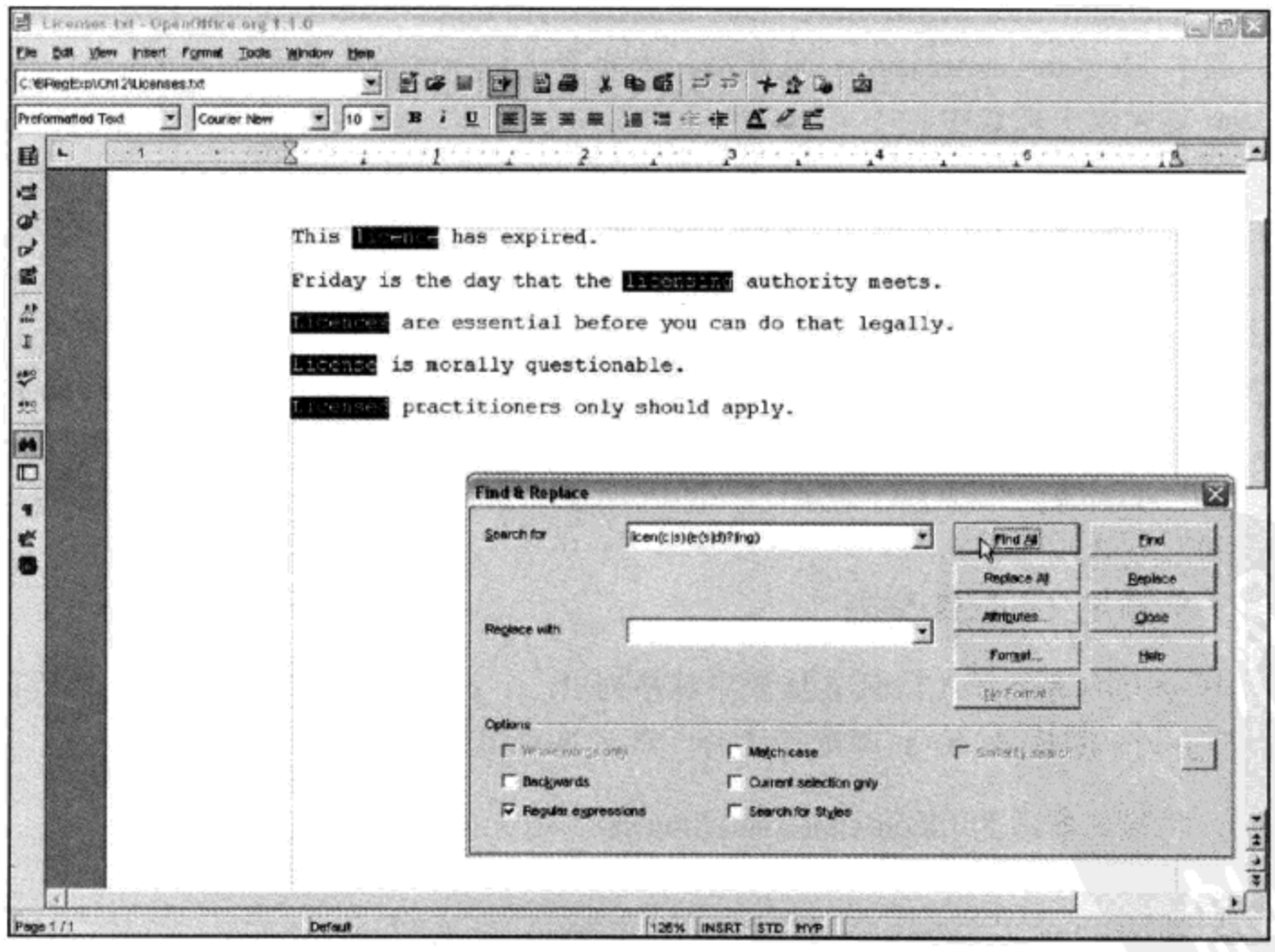


图 12-8

工作原理

下面来分析一下模式 licen(c|s)(e(s|d)?|ing)如何匹配测试文本中每一行。

第一行，匹配的文本是 licence。当正则表达式引擎位于 licence 的首字母 l 之前的位置时，首先会将该单词的前五个字符匹配模式中的前五个直接量字符。然后，licence 中的第二个 c 会与 (c|s) 中的第一个选项匹配。而 licence 中最后的字母 e 则与 (e(s|d)?|ing) 中的第一个选项匹配——换句话说，e(s|d)? 表示一个 e 后跟一个可选的 s 或 d。

第二行，匹配的文本是 licensing。当正则表达式引擎位于 licensing 的首字母 l 之前的位置时，单词的前五个字符首先与模式中的前五个直接量字符匹配。然后，licensing 中的 s 匹配 (c|s) 中的第二个选项。而最后的字符序列 ing 匹配了 (e(s|d)?|ing) 中的第二个选项——同样也是一个直接量字符序列 ing。

第三行，匹配的文本是 Licences。因为匹配是以不区分大小写的方式进行的，所以模式中开始的 licen 匹配单词中开始的字符序列 Licen。然后，Licences 中第二个 c 与 (c|s) 中的第一个选项匹配。而末尾的 es 则匹配了(e(s|d)?|ing) 中的第一选项——即，匹配一个直接量 e 后跟零个或一个 s。

第四行的匹配过程也一样，只不过不存在末尾的 s。因为 (s|d)? 的含义是 s 或 d 都是可选的，所以匹配同样成功。

第五行，匹配的文本是 Licenced。由于匹配不区分大小写，所以模式中开始的 licen 匹配单词中的字符序列 Licen。而 Licenced 中第二个 c 与 (c|s) 的第一个选项匹配。最后的 ed 匹配了 (e(s|d)?|ing) 中的第一个选项。也就是说，它匹配一个直接量 e 后跟零个或一个 d。所以在这一行，e(s|d)? 匹配的是字符序列 ed。

12.2.5　反向引用

OpenOffice.org Writer 不支持标准的反向引用，但它通过 & 元字符提供了一种受限的、类似反向引用的功能，可以用在搜索和替换中。

假设想将所有 walk 和 sulk 等单词修改为 walking 和 sulking 等形式。可以匹配直接量字符序列 lk，然后再将 ing 添加到这些字符序列后面。这一功能可以通过 & 元字符来实现。

测试文件 Walk.txt 的内容如下：

```
Walk

talk

sulk

milk
```

试一试：使用 & 元字符

(1) 打开 OpenOffice.org Writer，并打开测试文件 Walk.txt。

(2) 用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular Expressions 复选框。使 Match case 复选框为未选中状态。

(4) 在 Search for 文本框中输入模式 lk。

(5) 在 Replace with 文本框中输入模式 &ing。

(6) 单击 Replace All 按钮，并观察结果。

图 12-9 显示的是第 6 步之后的文档外观。如你所见，每个带有字符序列 lk 的单词都被添加上了 ing。

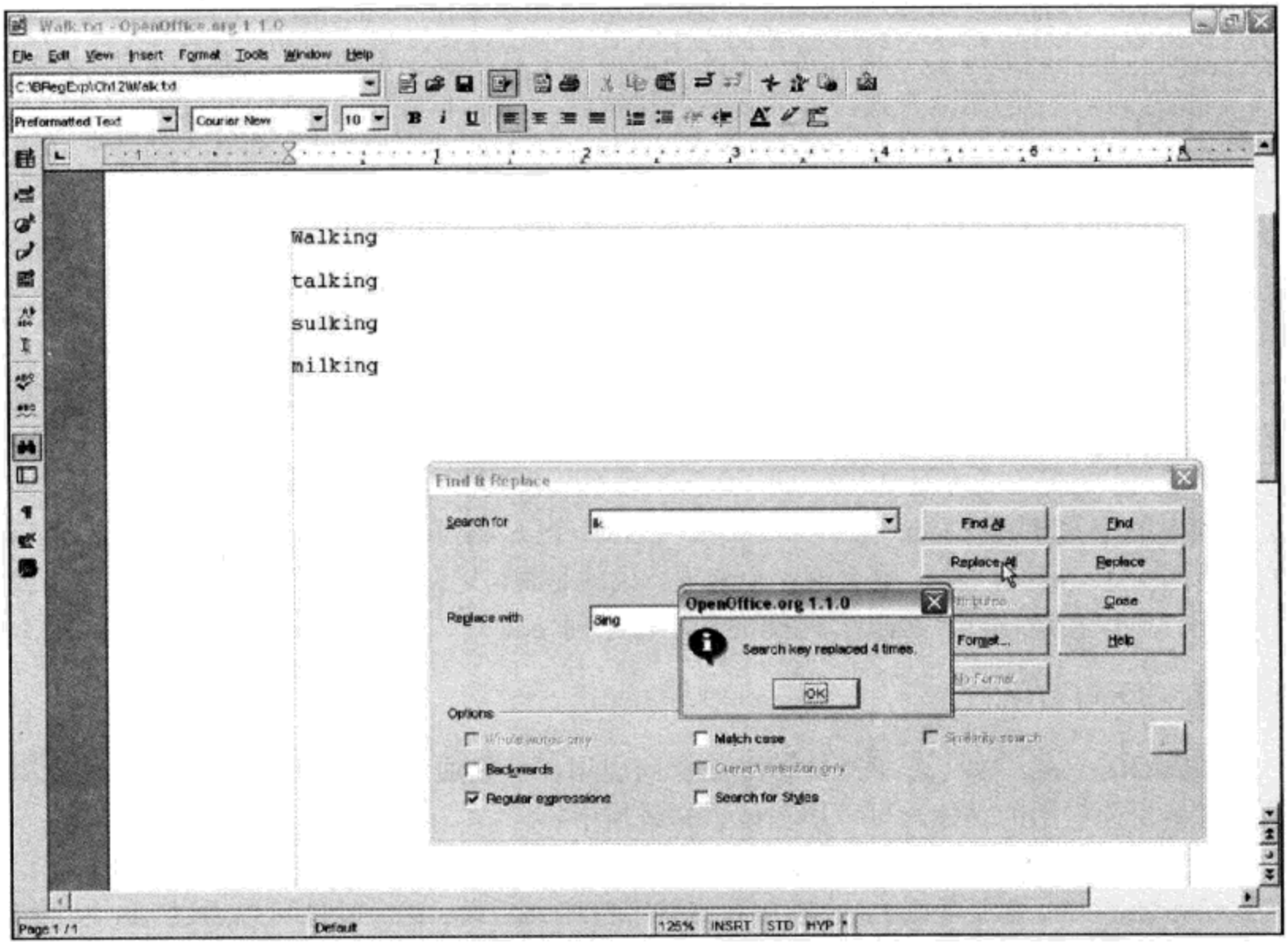


图 12-9

工作原理

& 元字符匹配 Search for 文本框中的模式所匹配的文本。在这个例子中，每次匹配的结果都是字符序列 lk。这个字符序列每次都被自身后跟另一个字符序列 ing 所替换。替换后，sulk 变成了 sulking，而 milk 变成了 milking。无论是什么模式，都必须仔细地评估该模式是否适合于相应的测试数据。如果这里的测试数据中还包含一个单词 walks，那么它将会被替换成 walkings。

12.2.6 向前查找和向后查找

OpenOffice.org Writer 不支持向前查找和向后查找。

12.3　搜索的例子

下面这个搜索的例子要在同一个句子中查找单词(严格来讲，是字符序列)Heaven 和 Hell。

测试文件 Heaven.txt 的内容如下：

```
This sentence contains both the words Heaven and Hell.

This sentence does not contain those two words and therefore is not matched.

This paragraph has Heaven in the first sentence. And Hell in the second.
```

问题定义可以表述如下：

匹配段落的开始位置，匹配零个或多个字符，匹配字符序列 Heaven，匹配零个或多个字符，匹配字符序列 Hell，匹配零个或多个字符，然后匹配一个直接量句点字符。

实现这个问题定义的模式是 ^.*Heaven.*Hell.*\.。

试一试：容易同时出现的词

(1) 打开 OpenOffice.org Writer，并打开测试文件 Heaven.txt。

(2) 用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 复选框。

(4) 在 Search for 文本框中输入模式 ^.*Heaven.*Hell.*\.。

(5) 单击 Find All 按钮，并观察结果。

图 12-10 显示的是第 5 步之后的结果。你可能会对第三段中的两个句子都作为匹配项被突出显示感到惊讶。这一点在稍后的“工作原理”部分将会进行解释。

如果只想匹配同一个句子中的一个实例，那么当前使用的模式就不够具体。可以把这个模式修改为 ^.*Heaven[^.]*Hell.*\.。

(6) 将 Search for 文本框中的模式修改为 ^.*Heaven[^.]*Hell.*\.。

(7) 单击 Find All 按钮，并观察结果。

图 12-11 显示的是第 7 步之后的结果。注意，此时只有第一个段落中的句子作为匹配项被突出显示出来。

(8) 如果只想匹配同一段落中的两个单词，还有一个替代模式可用。将 Search for 文本框中的模式修改为 ^.*Heaven.*Hell.*$。

(9) 单击 Find All 按钮，并观察结果。此时结果中突出显示的文本与图 12-10 所显示的结果相同。

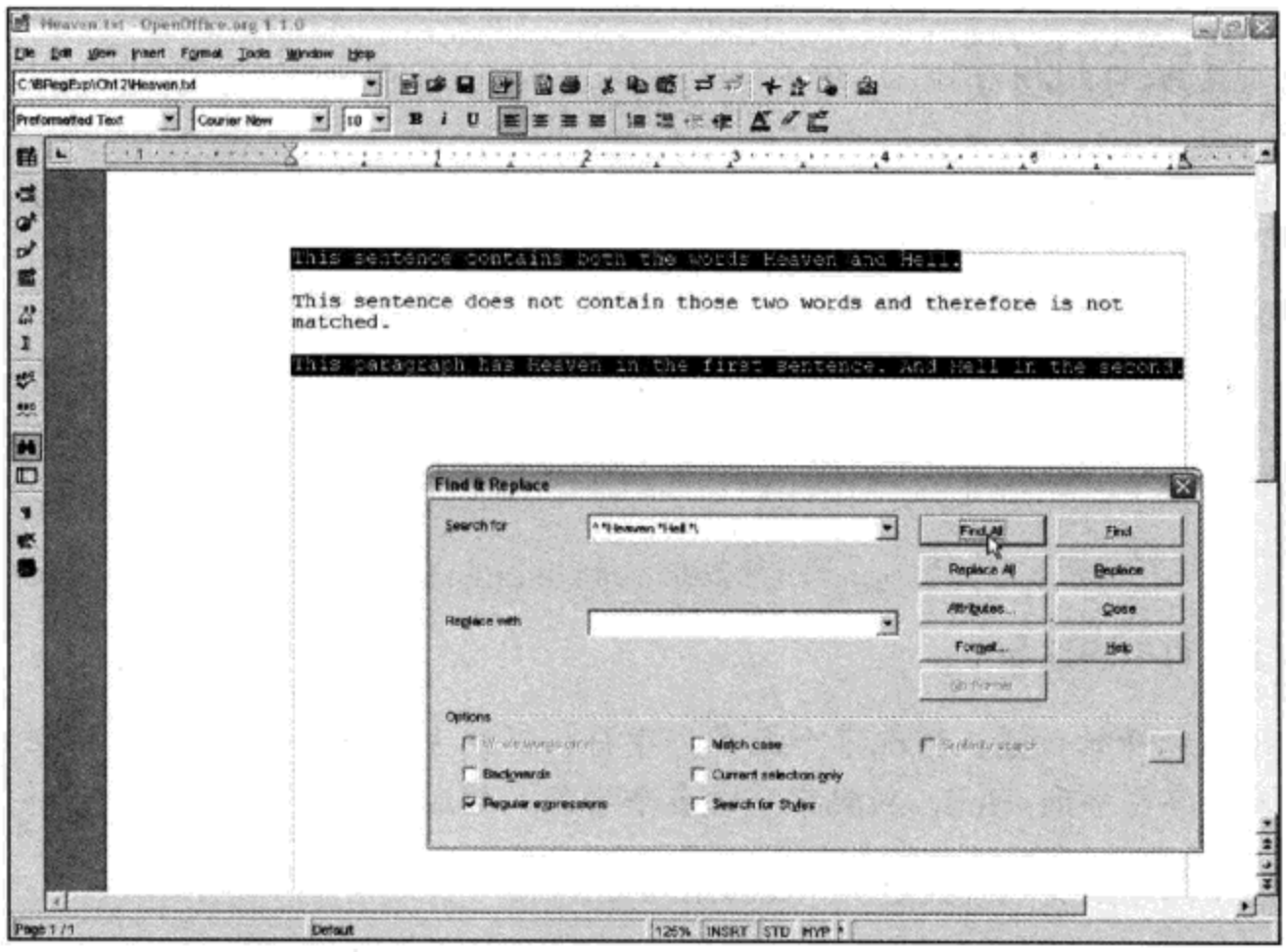


图 12-10

工作原理

在第 5 步之前使用的模式是 ^.*Heaven.*Hell.*\.。其中，^ 元字符匹配段落开始的位置，而 .* 匹配零个或多个字符，Heaven 匹配直接量字符序列 Heaven。之后的 .* 匹配零个或多个字符，而 Hell 也匹配直接量字符序列 Hell。接下来的 .* 匹配零个或多个字符，而 \. 则匹配一个直接量句点字符。

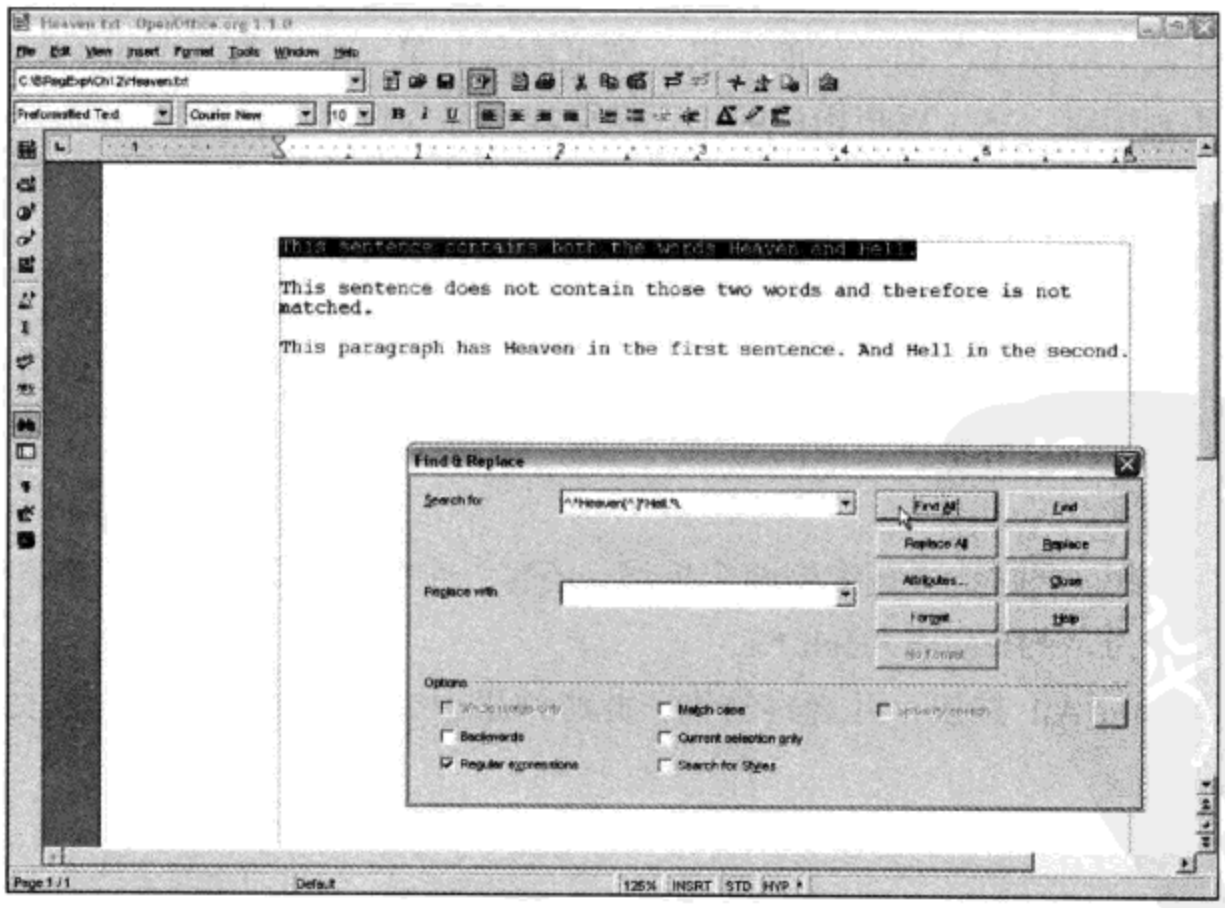


图 12-11

第一段中的匹配过程显而易见。但第三段中的匹配过程就不是那么明显。其中关键是正则表达式中位于 Heaven 之后和 Hell 之前的 .*。由于 OpenOffice.org Writer 的匹配是贪婪匹配，所以 .* 也会匹配第一句末尾的句点字符。因此只要字符序列 Hell 后存在一个句点字符，它就会匹配分别位于两个不同句子中的 Heaven 和 Hell。如果删除第三段中最后的句点字符，那么第三段与模式 ^.*Heaven.*Hell.*\ 就不匹配了。

第 6 步使用的模式是 ^.*Heaven[^.]*Hell.*\.，在 Heaven 和 Hell 之间有一个 [^.]*，这个组件的含义是在字符序列 Heaven 和 Hell 之间只能存在非句点字符。也就是说，只有当这两个字符序列出现在同一个句子中时才会存在匹配项。

第 8 步中使用的模式 ^.*Heaven.*Hell.*$ 使用了 $ 元字符，$ 元字符在 OpenOffice.org Writer 中匹配段落的结束位置，而 ^ 元字符匹配段落的开始位置。组件 .* 匹配零个或多个字符，Heaven 匹配其直接量，后面的 .* 匹配零个或多个字符，Hell 同样匹配其直接量，而 .* 仍然匹配零个或多个字符。最后的 $ 元字符匹配段落结尾的位置。也就是说，只要在段落中 Heaven 位于 Hell 之前就存在匹配项。

12.4　搜索和替换的例子

下面的例子示范了在 OpenOffice.org Writer 中非常实用的正则表达式用法。

在线聊天记录

信息传递工具的发展非常迅速。在线聊天在实时信息交流中占有重要地位。然而，由于聊天记录中存在许多格式不规范的语句，而且其中还混杂着关于加入或者离开之类的消息文本，使得阅读这些聊天记录变得非常吃力。正则表达式可以快速地清理这些记录。

经过高度简化的测试文档 Interesting Chat.sxw 的内容如下：

```
Some interesting chat
A welcome message.
Some interesting information.
Somebody says something interesting.
(Andrew Smith has joined the conversation
(Jane Callander has left the conversation.

Another piece of real chat.

(Harry Danvers has joined the conversation
(Carol Clairvoyant has left the conversation
(Ceridwen Davies has joined the conversation.
Another real comment.
```

例子中的“(”是聊天软件用于标记加入和离开动作的非字母字符表示法。

在人数众多的聊天室中，这些加入或离开信息会比真正的聊天信息还多。例如，在作者撰写本章并在那一天把这项技术应用于一个真实的聊天过程中时，在聊天记录中居然有 1200 多条这样的重复信息。

图 12-12 显示的是测试文档的外观。注意其中位于加入和离开信息开头处的右方向箭头。

我们的目标是将这些有关加入和离开的信息清理掉，以便使聊天的主题更加突出。相应的问题定义如下：

删除包含个人加入或离开聊天室的信息记录行。

聊天软件将所有加入和离开信息放在单独的一行中使得我们的任务很明确。因此，完善这个聊天记录例子的问题定义可以改进如下：

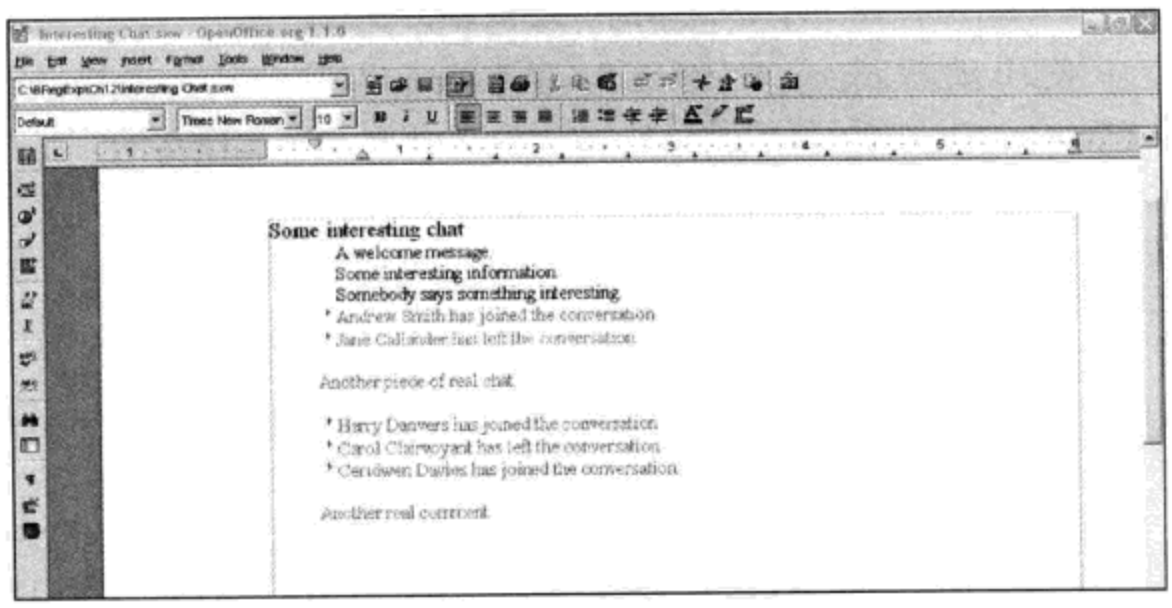


图 12-12

匹配所有以聊天软件使用的特殊符号开头的行，后跟零个或多个任意字符，再跟一个行结尾的位置。用空白替换所有匹配项。

前面提到的现实中的例子是在 Microsoft Word 文档中。由于 Microsoft Word 没有匹配行开始和结束位置的元字符，所以在 OpenOffice.org Writer 中完成搜索和替换会更方便。下面在 OpenOffice.org Writer 中打开了这个文档，并利用 Writer 中对正则表达式更完整的支持来实现作者的意图。

在默认情况下，Writer 是以只读方式打开 Word 文档的。如果想编辑该文档，只需单击工具栏中的 Edit 按钮，就会有对话框跳出来询问是否想编辑当前的文档。选择 Yes 即可打开一个新的 Writer 文档(.sxw)，于是就可以使用 Writer 中的正则表达式来清理其中的文本内容了。清理完毕后，可以使用 Writer 中的 Save As 选项把文档保存为 Word 格式。

试一试：清理在线聊天记录

(1) 打开 OpenOffice.org Writer，然后打开测试文件 Interesting Chat.sxw。

(2) 用 Ctrl+F 快捷键打开 Find & Replace 对话框。

(3) 选中 Regular expressions 和 Match case 复选框。

(4) 突出显示文本行中的右方向箭头符号。

(5) 在 Search for 文本框中输入 ^ 元字符，粘贴右方向箭头符号，并输入 .*$。此时应该看到图 12-13 中 Search for 文本框内的模式。注意，粘贴的右方向箭头显示为一个空心的方块形符号。虽然显示不准确，但匹配过程依然是正确的。使 Replace with 文本框为空。

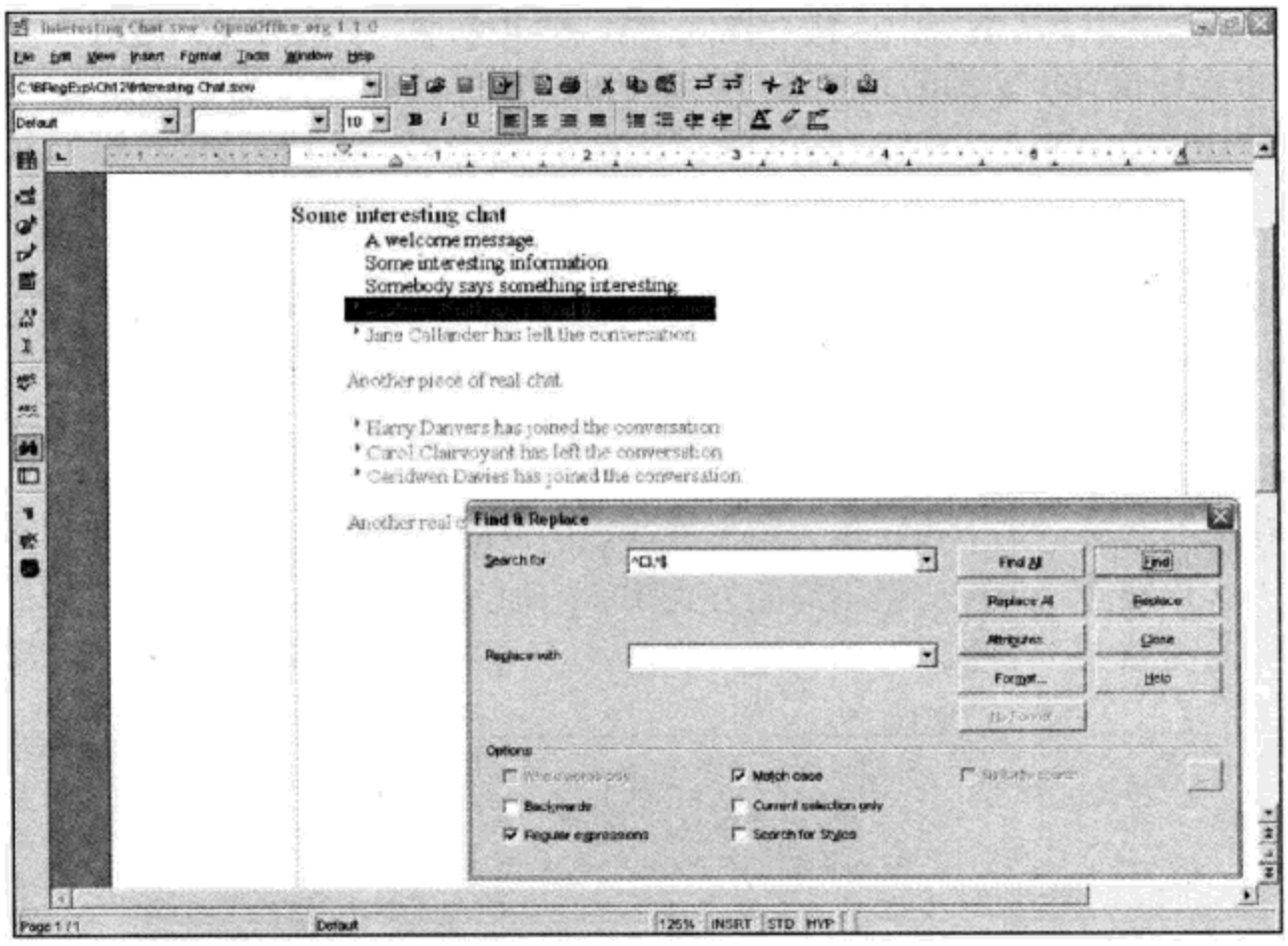


图　12-13

(6) 单击一次 Find 按钮。此时，包含右方向箭头的第一行文本会被突出显示出来。

(7) 单击一次 Replace 按钮。第 6 步中突出显示的那一行不见了。

(8) 单击一次 Replace All 按钮。所有包含右方向箭头符号的行都消失了。

图 12-14 显示的是第 8 步之后的结果。所有以前包含右方向箭头符号的行都被删除了。

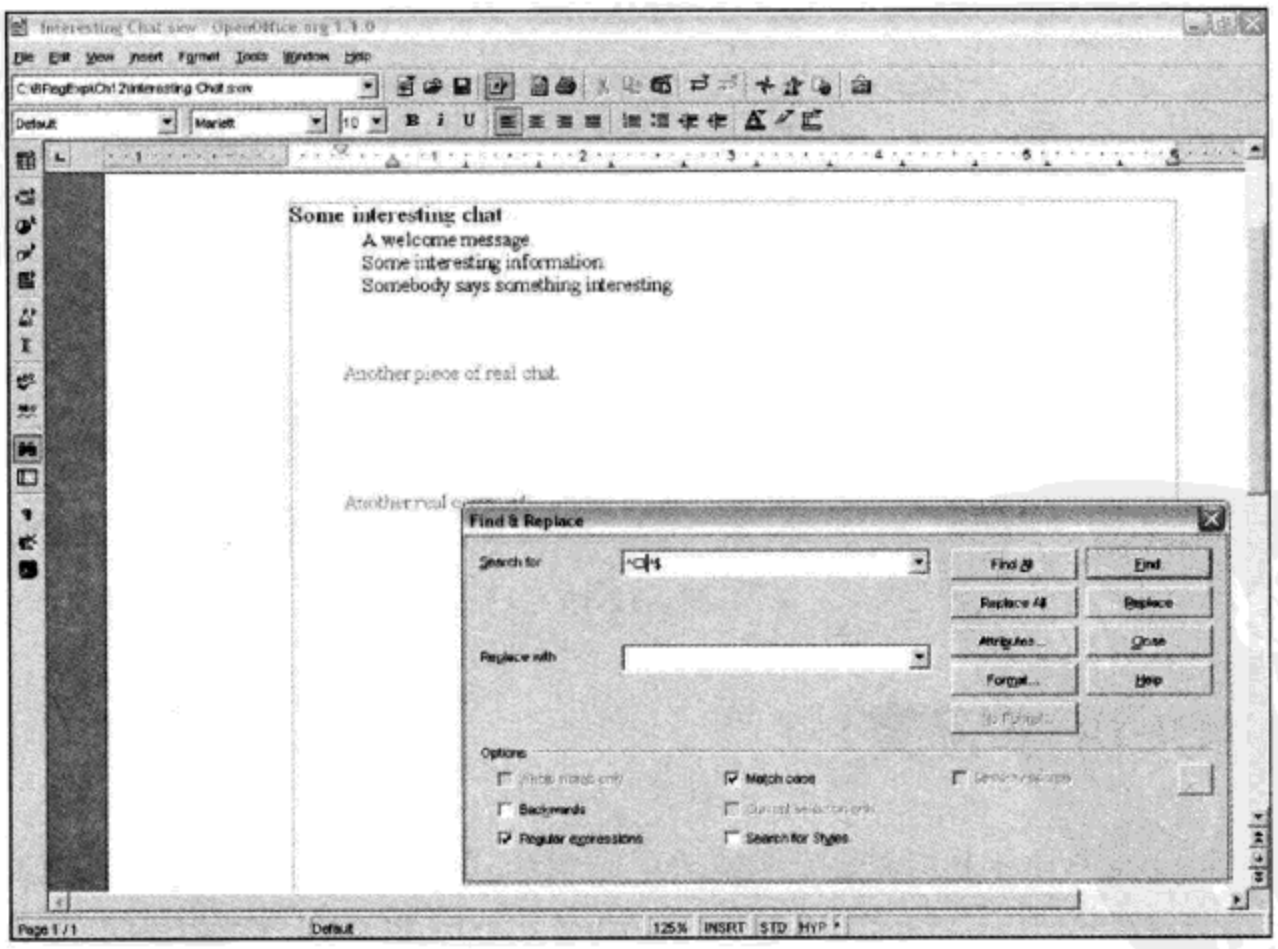


图　12-14

(9) 在 Search for 文本框中，输入模式 ^$。使 Replace with 文本框为空。

(10) 将光标重置于文档的开始处。单击 Find 按钮。第一个空白行被突出显示了出来。

(11) 单击 Replace All 按钮，以替换掉所有的空白行，并观察结果，如图 12-15 所示。现在所有包含右方向箭头符号(也就是包含某人加入或离开信息)的行都被删除了。

工作原理

第 5 步创建的模式匹配任何以右方向箭头符号开头的行。其中，^ 元字符匹配一行的开始位置。而右方向箭头符号匹配自身。模式 .* 匹配零个或多个字符。$ 元字符匹配一行结尾的位置。

此处使用的聊天记录中包含加入或离开信息的每一行都是以一个右方向箭头开始的。而其他聊天客户端程序可能会以不同的方式来表示加入或离开的信息。如果其他程序在右方向箭头的前面加上一个空格，那么也应该在 ^ 元字符的后面插入一个空格符。

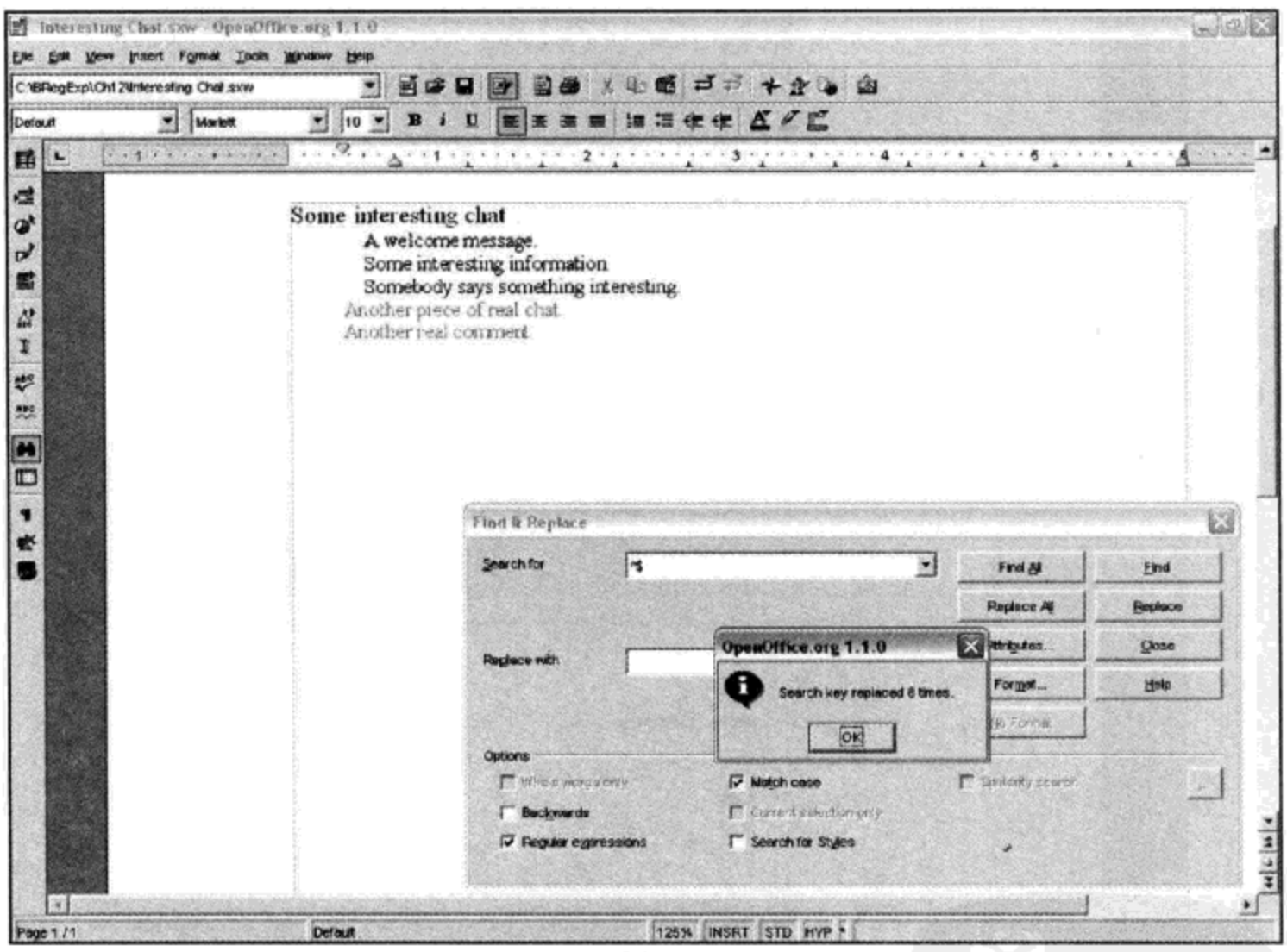


图 12-15

12.5 POSIX 字符类

OpenOffice.org Writer 1.1 除支持常规的正则表达式字符类外，还支持 POSIX 字符类的一个子集。表 12-2 中列出了 OpenOffice.org Writer 支持的 POSIX 字符类及其含义。其中，? 元字符是 POSIX 字符类语法的一个组成部分。但此时，它不再是一个表示前面字符类是可

选的限定符。

表 12-2　OpenOffice.org Writer 支持的 POSIX 字符类及其含义

字　符　类	含　　义
[:digit:]?	单独使用时匹配一个单独的数字。作为较长模式的组件使用时，它匹配一个可选的数字
[:digit:]*	匹配零个或多个数字
[:space:]?	匹配空格符
[:print:]?	匹配一个单独的可打印字符，包括空格符。当作为较长模式的组件使用时，它匹配一个可选的可打印字符
[:alnum:]?	匹配一个字母字符或一个数字。作为较长模式的组件使用时，它匹配一个可选的字母或数字字符
[:alpha:]?	匹配一个字母字符，但不匹配数字
[:lower:]?	在选中 Match case 复选框的情况下，它匹配一个小写的字母；否则，含义与 [:alpha:]? 相同
[:upper:]?	在选中 Match case 复选框的情况下，它匹配一个大写的字母；否则，含义与 [:alpha:]? 相同

匹配数字

如表 12-2 中所提到的，POSIX 字符类在单独使用时具有某些特殊含义。本节的例子将会通过试验一个测试文件，弄清楚 OpenOffice.org Writer 中 POSIX 字符类[:digit:]?的用途。

要使用的测试文件是 ADigitsB.txt，其内容如下：

```
A123B

AB

A8B

A1234567890B
```

如你所见，所有测试字符串都由一个大写的 A 后跟零个或多个数字，再跟一个 B 组成。

试一试：使用 POSIX [:digit:] 字符类

(1) 在 OpenOffice.org Writer 中打开文件 ADigitsB.txt。
(2) 使用 Ctrl+F 快捷键打开 Find & Replace 对话框。
(3) 选中 Regular expressions 复选框。
(4) 在 Search for 文本框中，输入模式 [:digit:]?。
(5) 单击几次 Find 按钮(不是 Find All)，并观察每次单击后突出显示的字符。

每次只看到一个数字被突出显示。单独使用时，模式 [:digit:]? 可匹配一个数字。OpenOffice.org Writer 不会识别单独的模式 [:digit:]，读者可以删除模式中的 ？来试一下。

(6) 单击测试文件中第一个字符之前的位置。将 Search for 文本框中的模式修改为 [:digit:]。

(7) 单击一次 Find 按钮，并观察结果。

此时，会看到如图 12-16 所示的对话框。其中的信息表明 OpenOffice.org Writer 已搜索了整个文档，但没有找到匹配项。

单独使用限定符 * 和 + 时(即代替 [:digit:]? 中的 ？时。译者注)，结果是一样的。也就是说，模式 [:digit:]* 和 [:digit:]+ 都会匹配一个或多个数字(即单独使用 * 和 + 时，相当于使用限定符 +。译者注)。

(8) 再单击文件中第一个字符之前的位置，并将正则表达式模式修改为 [:digit:]*。

(9) 单击几次 Find 按钮，并观察突出显示的字符，直到搜索完整个文档。

(10) 再单击文件中第一个字符之前的位置。将正则表达式模式修改为 [:digit:]+。

(11) 单击几次 Find 按钮，并观察突出显示的字符，直到搜索完整个文档。

(12) 当在较长的模式中使用时，模式 [:digit:] 的行为会稍有不同。单击文件中第一个字符之前的位置，并将正则表达式修改为 A[:digit:]B。

(13) 单击两次 Find 按钮，每次单击后观察结果。此时，[:digit:] 的行为与之前所期望的一样——匹配一个数字。同样地，当 [:digit:] 在构成一个较长的模式时，它后面的 ？限定符表示该字符类是可选的。

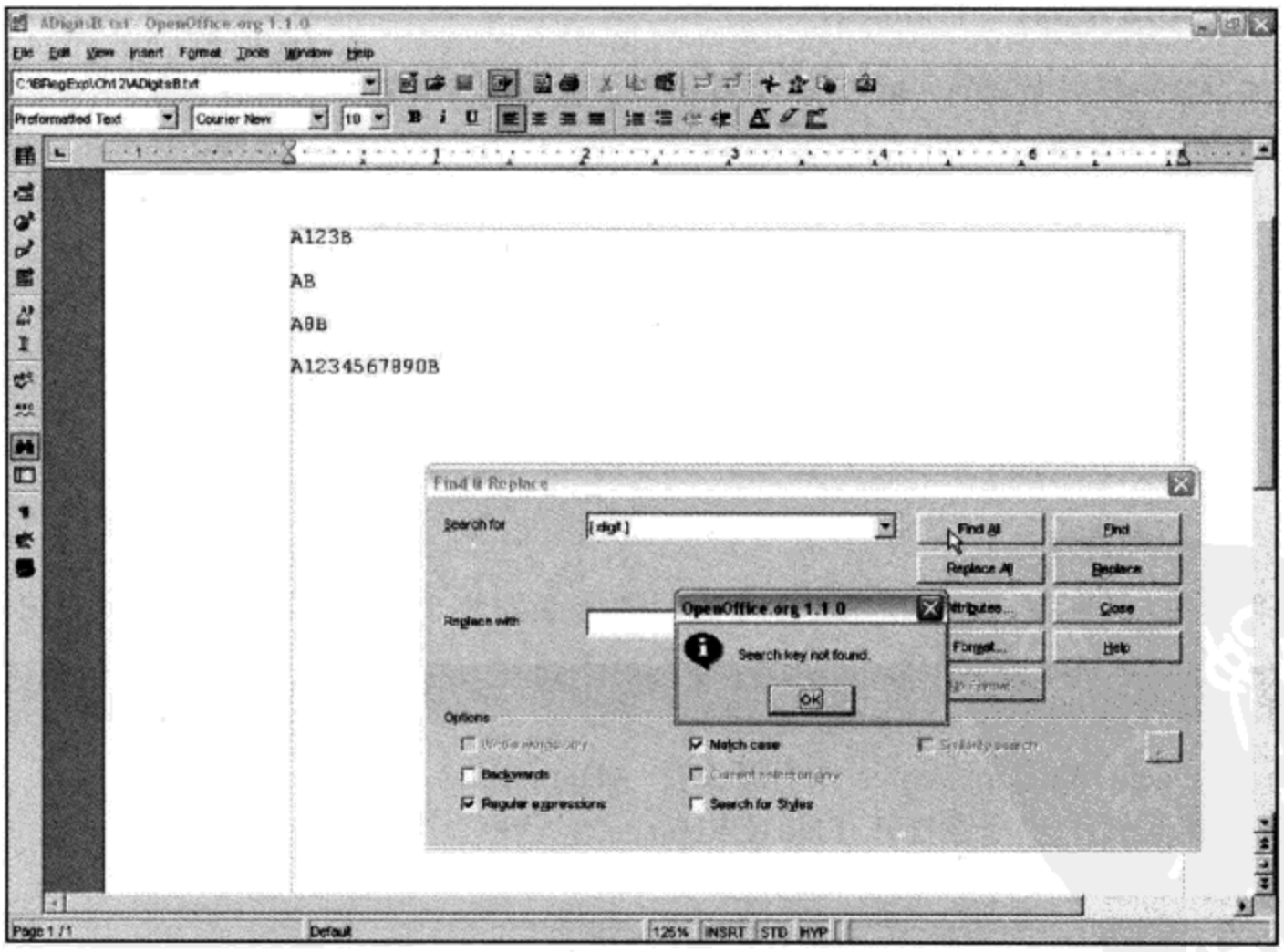


图 12-16

工作原理

在第 4 步中使用的模式是 [:digit:]?。每次单击 Find 按钮时，都会有一个数字被匹配。如果删除了 ?，则不存在匹配项。

当使用模式 [:digit:]* 时，字符序列 123、8 和 1234567890 都会被匹配。因为 * 元字符的作用相当于一个限定符(原文此处描述不恰当。因为单独使用 [:digit:]* 时会匹配一个或多个数字，而 * 并没有起限定符——零个或多个数字——的作用。只有在这些 POSIX 字符类作为较长模式的组件时，?、* 和 + 才作为限定符使用。译者注)。模式 [:digit:]+ 也会匹配这些相同的字符序列。

但是，当把模式修改为 A[:digit:]B 时，POSIX 字符类中不需要 ? 元字符也可以匹配一个数字(这与单独使用 [:digit:] 无法匹配数字的情况恰好相反。译者注)。此时，这个模式仅匹配测试文本中的字符序列 A8B。如果加上 ?，那么模式 A[:digit:]?B 会匹配 A8B 和 AB(此时 ? 作为限定符使用。译者注)。

12.6　练习

下面的练习题旨在测试对本章所学内容的掌握程度：

1. 请指定一个匹配除 W、X、Y 和 Z 外所有大写字母的字符类。
2. 请指定一个匹配 a~h 和 t~z 的小写字母字符类。

第13章

通过 findstr 使用正则表达式

findstr 实用程序是一种命令行实用程序，用来查找包含特定字符串模式的文件。findstr 实用程序能够完成类似 Windows 搜索完成的任务，并且还支持更加具体的搜索，包括使用正则表达式。

findstr 实用程序能够利用命令行中所提供的参数和一些标准或非标准的正则表达式语法。

在本章中将学习以下内容：

- 如何在命令行中使用 findstr
- 如何使用 findstr 支持的正则表达式元字符

13.1 findstr 简介

在比较新的 Windows 版本中都包含了 findstr 实用程序。比如，在 Windows XP 的命令行中就可以不通过设置路径或者环境变量直接使用 findstr。

本章的内容基于 Windows XP Professional 中的 findstr 实用程序来介绍。

确认所使用的 Windows 版本中是否包含 findstr 以及相应的功能，只需在命令行中简单地输入 findstr /?，然后回车即可。如果一切顺利，应该能够看到相当多的帮助信息显示在命令提示符窗口中。图 13-1 中是所显示帮助信息的最后一部分。其余的帮助信息已经滚动到视野之外了。

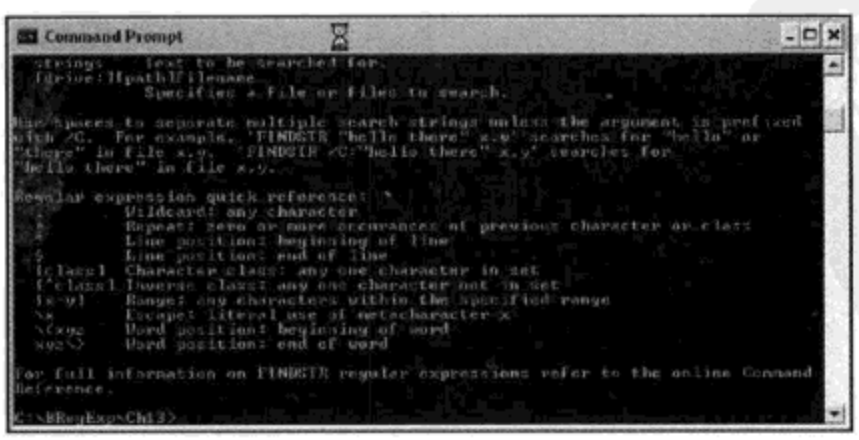


图 13-1

不能在命令行中单独使用以下命令，否则会收到一个出错信息，如图 13-2 所示。

```
findstr
```

图　13-2

在使用 findstr 的过程中，基本上是依靠一个或多个命令行开关(switchs)字符和相应的参数来指示 findstr 实现相应的搜索。

查找直接量文本

findstr 能够完成的一个最简单的任务就是匹配直接量文本。在一个单独的文件中完成简单直接量匹配的 findstr 命令的一般形式如下：

```
findstr "要搜索的文本" 文件名.扩展名
```

严格来讲，这里使用的是一个只包含要匹配的直接量字符的正则表达式模式。

这里要使用测试文件 Hello.txt，其内容如下：

```
Hello world!
Hello with initial upper-case.
hello with initial lower case.
Goodbye!
```

注意，有两行中 Hello 的第一字母是大写的 H，而有一行中 hello 的第一个字母是小写的 h。

试一试：查找直接量文本

(1) 打开命令提示符窗口，并定位到保存测试文件 Hello.txt 的目录下。

(2) 在命令行中输入以下命令：

```
findstr "Hello" Hello.txt
```

(3) 按回车键，并观察由 findstr 返回的结果。图 13-3 显示的是其结果。其中显示了包含 Hello(首字母为大写 H)的两行文本，而包含 hello(首字母为小写 h)的那一行没有显示。这是因为 findstr 默认执行的是区分大小写的匹配。

结果中只显示了两行文本的内容，但没有指出它们所在的文件以及行号。而这些附加信息在搜索多个文件时是非常有用的。

图　13-3

(4) 本例中测试文件 Hello.txt 中的数据都是分行放置的，但并非所有文档都有如此简单的数据结构。因此，通常有必要在显示特定行中文本的同时也显示行号，这样就可以在长文档中找到匹配的行并了解其上下文环境。要通过 findstr 来显示行号，使用 /n 开关。

在命令行中输入以下命令，并按回车键：

```
findstr /n "Hello" Hello.txt
```

(5) 观察在命令中添加了 /n 开关后返回的结果。如图 13-4 所示，现在显示了测试文件中包含匹配文本的每一行的行号。

有时候，如果在命令行中使用 F3 重复显示以前的命令并进行修改，findstr 实用程序会在存在匹配的情况下匹配失败。如果发现这种意想不到的匹配失败，建议重新输入一次命令。这样一般就可以解决问题。

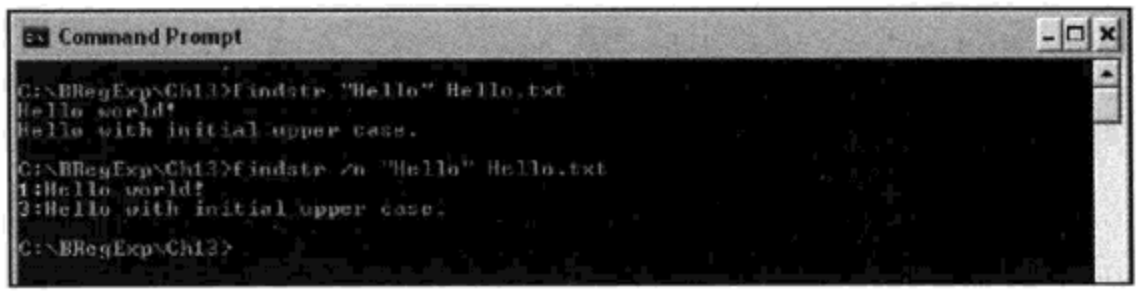


图 13-4

(6) 如果想以不区分大小写的方式完成匹配，可以使用 /i 开关。在命令行中输入以下命令，并按回车键：

```
findstr /i /n "Hello" Hello.txt
```

图 13-5 显示的是其结果。包含 Hello 和 hello 的三行文本全部显示出来了。

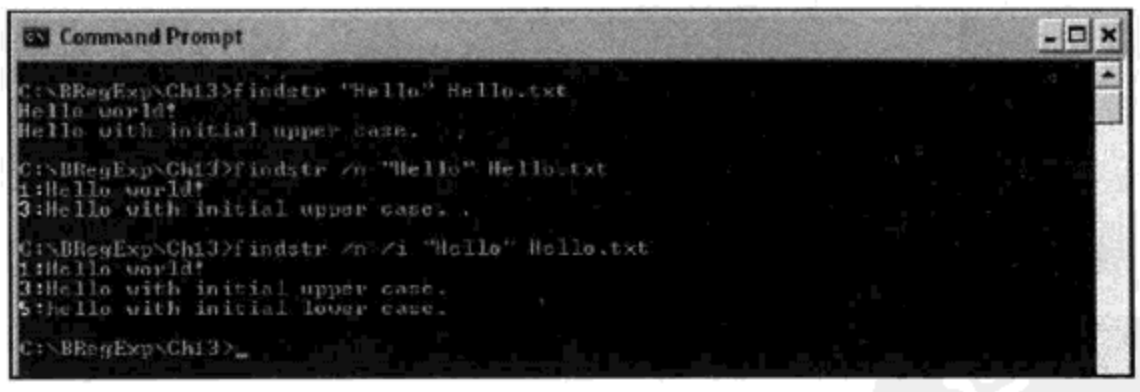


图 13-5

有一些 findstr 命令行开关起到了正则表达式中元字符的作用。这些开关将在下一节讨论 findstr 支持的元字符时再做介绍。

13.2 findstr 支持的元字符

findstr 实用程序支持许多正则表达式元字符，但也许是因为在命令行中使用的缘故，

这些被支持的元字符包含许多非标准的正则表达式语法(参见表 13-1)。

表 13-1　findstr 支持的元字符

元　字　符	说　　明
.	匹配任何字符
*	表示零个或多个实例的限定符
?	不支持
+	不支持
{n,m}	不支持
^	匹配行开始的位置
$	匹配行结尾的位置
[...]	字符类
\<	匹配词开始的位置
\>	匹配词结束的位置

如表 13-1 所示，findstr 实用程序不支持其中一些元字符。表 13-2 中列出了能够完成与其他实现中的正则表达式元字符类似功能的命令行开关，以及表示其他含义的命令行开关。需要带参数的命令行开关将在表 13-3 中介绍。

表 13-2　命令行开关及其含义

命令行开关	等价的元字符或其他含义
/b	当后续的字符位于一行开始的位置时匹配。等价于 ^ 元字符
/e	当后续的字符位于一行结尾的位置时匹配。等价于 $ 元字符
/p	忽略包含不可打印字符的文件
/offline	只处理带有脱机属性集的文件
/o	在每个匹配行前打印字符相对于文件起始位置的偏移量
/m	如果文件包含匹配项，只打印其文件名
/n	显示包含匹配项并显示的行的行号
/v	显示不包含匹配项的行
/x	只有整行与正则表达式匹配才能匹配。类似于在其他实现中使用 ^ 和 $ 元字符
/i	以不区分大小写的方式完成匹配。默认行为是区分大小写的匹配
/s	在当前目录及其所有子目录中搜索符合标准的文件
/r	指定位于一对双引号中的文本作为正则表达式解释。即使不指定 /r 开关，这也是默认行为
/l	不把正则表达式解释为正则表达式，而是匹配其直接量

表 13-3 中的命令行开关都带有影响其行为的参数。

表 13-3　带参数的命令行开关

命令行开关	描　　述
/f:file	参数 file 是一个文件名，相应的文件中包含要搜索的文件列表
/c:string	参数 string 是一个直接使用的搜索字符串
/g:file	参数 file 是一个文件名，相应的文件中包含要查找的字符串列表
/d:dirlist	参数 dirlist 是一个包含要搜索目录的以逗号分隔的列表
/a:colorattribute	参数 colorattribute 是使用两个十六进制数指定的颜色属性值

13.2.1　限定符

findstr 对限定符的支持是有限的。它支持标准的 * 限定符(即匹配零个或多个实例)。然而，却不支持 ?、+ 限定符以及 {n,m} 限定符语法。

下面通过测试文件 Order1.txt 和 Order2.txt 演示如何使用 * 限定符。

Order1.txt 的内容如下：

```
This is an order for Part No. ABC123.
Blah blah. As easy as ABC.
2004/08/20
```

Order2.txt 的内容如下：

```
This is an order for Part No. ABC456.
Blah blah.
2003/07/18
```

为了方便起见，本例只匹配其中的零件编号。在许多正则表达式实现中都可以使用 ABC\d{3}或 ABC[0-9]{3}来匹配三位数字，但 findstr 不支持这样的语法。

试一试：使用 * 限定符

(1) 打开命令提示符窗口，并在命令提示符下输入以下命令：

```
findstr /n "ABC[0-9]*" Order*.txt
```

(2) 观察返回的结果，如图 13-6 所示。有三行都包含匹配项。但第二行不是想要的匹配，因为该行中 ABC 的后面没有数字，因而不是一个零件编号。

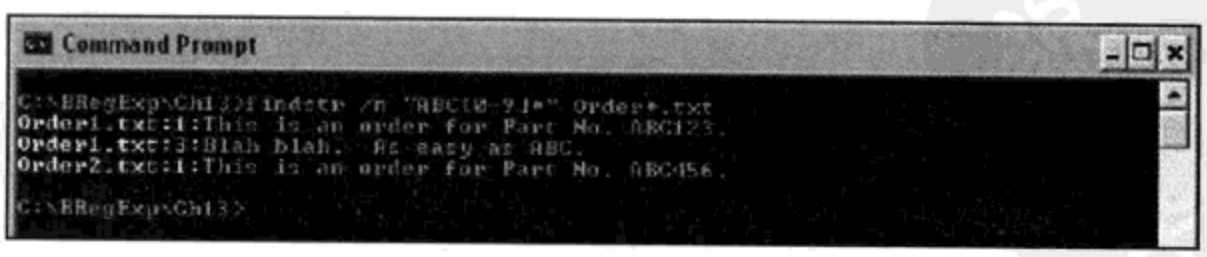


图　13-6

(3) 如果要匹配想要的数字个数(三个)，就要使用下面的模式：

```
ABC[0-9][0-9][0-9]
```

(4) 在命令行中，输入以下命令：

```
findstr /n "ABC[0-9][0-9][0-9]" Orders*.txt
```

图 13-7 显示的是其结果。

图　13-7

工作原理

在第 2 步之后，结果中有两行包含着由字符序列 ABC 后跟三位数字组成的零件编号——这是想要的匹配结果。然而，Orders1.txt 中的第二行也匹配，这是因为模式 [0-9]* 匹配零个或多个匹配数字的字符类，而 ABC 除了 ABC 没有别的，所以带有零个数字的 ABC 与模式 ABC[0-9]* 匹配。

findstr 实用程序不支持反向引用、向前查找和向后查找。

13.2.2　字符类

在本章前面的例子中已经看到，findstr 支持字符类 [0-9]。事实上，如果要使用 findstr 匹配数字，还可以使用字符类 [0123456789]，因为 findstr 不支持 \d 元字符。

下面要用的测试文件是 PartNums.txt，其内容如下：

```
ABC123

DEF890

GHI234

HKO838

RUV991

ILR246

UVW991

ADF274

DRX119
```

findstr 支持范围，也支持对字符类取反。

试一试：使用字符类

(1) 打开命令提示符窗口，并将其定位于包含测试文件 PartNums.txt 的目录下。

(2) 在命令行中输入下列命令：

```
findstr /n "A[A-Z][A-Z][0-9][0-9][0-9]" PartNums.txt
```

(3) 观察如图 13-8 所示的结果。以大写的 A 开头，后跟两个大写字母和三个数字的零件编号所在的行被显示了出来。

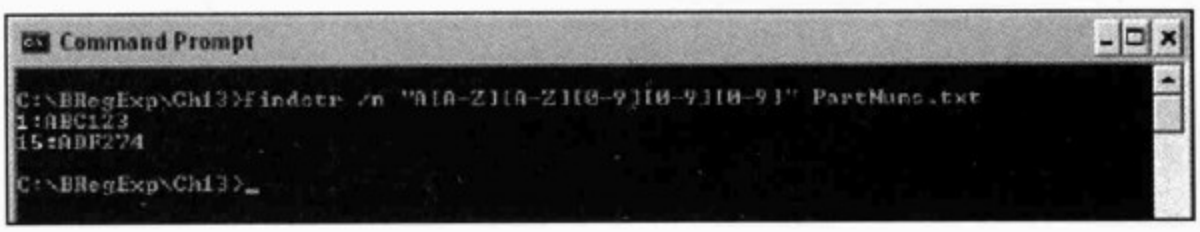


图 13-8

基于 findstr 的匹配方式，对给定的测试数据而言，还可以使用一种更简单的模式 A[A-Z][A-Z][0-9]。但是，如果测试文本中包含 ABC1 这样的零件编号，这个简单的模式也会匹配它，而它并不是我们想要的。

(4) 在命令行中输入下面的命令：

```
findstr /n "A[A-Z][A-Z][0-9]" PartNums.txt
```

(5) 观察结果(仍然匹配相同的那些行)。

(6) 如果想匹配的零件编号以 A 开头，但第二个字符不是大写的 B，那么就需要使用取反的字符类 [^B] 来表达这一含义。

在命令行中，输入下面的命令：

```
findstr /n "A[^B][A-Z][0-9]" PartNums.txt
```

(7) 观察结果(现在第一行不再匹配)，如图 13-9 所示。

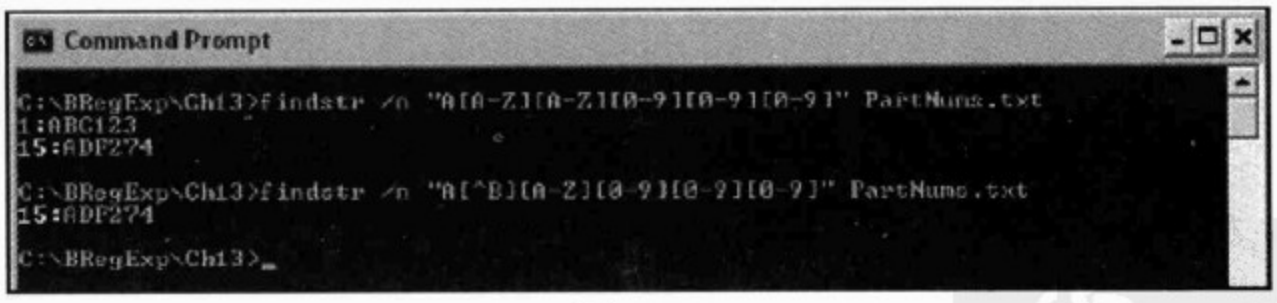


图 13-9

工作原理

在第 2 步中，使用的模式是 A[A-Z][A-Z][0-9][0-9][0-9]。A 匹配大写的直接量 A。因为只有第 1 行和第 15 行包含的零件编号以 A 开头，所以只有这两行可能与其余的正则表达式组件匹配。字符类 [A-Z] 匹配任何字母字符，匹配第 1 行的 B 和第 15 行的 D。第二个字符类 [A-Z] 匹配第 1 行的 C 和第 15 行的 F。而接下来的三个字符类 [0-9][0-9][0-9] 匹配第 1 行和第 15 行中连续的三个数字 123 和 274。因此，第 1 行

和第 15 行都匹配。

在第 6 步中，使用了模式 A[^B][A-Z][0-9]。同以前一样，第一个 A 匹配第 1 行和第 15 行中的 A。而字符类 [^B] 匹配除 B 之外的任何字母字符。所以，第 1 行不能匹配模式 A[^B]。但在第 15 行，D 与 [^B] 是匹配的，所以匹配在第 15 行中仍然可以继续。接着，模式 [A-Z] 匹配第 15 行中的 F，而[0-9][0-9][0-9] 匹配第 15 行中的 274。因此，第 15 行就成为了唯一的匹配项。

在模式中包含 [^B] 这样的字符类存在一定的风险，因为这个字符类几乎可以与点(.)元字符相媲美。所以，如果测试文件中存在不正确的零件编号 A$C123，同样也会匹配模式 [A-Z][^B][A-Z][0-9][0-9][0-9]。如果目的是匹配除 B 之外的任何大写字母，那么更具体的字符类应该是 [AC-Z]。此时相应的正则表达式要写成：

```
A[AC-Z][A-Z][0-9][0-9][0-9]
```

[AC-Z]与[ACDEFGHIJKLMNOPQRSTUVWXYZ]含义相同。

13.3　词边界位置

findstr 实用程序支持匹配词开始位置的元字符和词结束位置的元字符。\< 元字符匹配词的开始位置，\> 元字符匹配词的结束位置。

下面试验中要用的测试文件 Word.txt 的内容如下：

```
Swords are sharp, typically.

Words are powerful things. They can wound.

Churchill is a byword for wartime persistence.

Do you have a favorite word?

His surname is Answord.

Wordsworth was a famous English poet.

Word by word is, typically, not a good method of translation.
```

注意，字符序列 word 多次出现在一个字母字符序列的开头或结尾，或者嵌入在一个较长字符序列的内部。同时，也要注意 word 或 Word 中 w 的大小写形式，因此在采用区分大小写或者不区分大小写的搜索时要当心。

试一试：词的开始和结束位置

(1) 打开命令提示符窗口，并定位至包含测试文件 Word.txt 的目录下。

(2) 在命令提示符中，输入以下命令：

```
findstr /n "Word" Word.txt
```

(3) 观察如图 13-10 所示的结果。七行中只有三个被显示出来。这是因为 findstr 在默认情况下采取区分大小写的搜索方式。

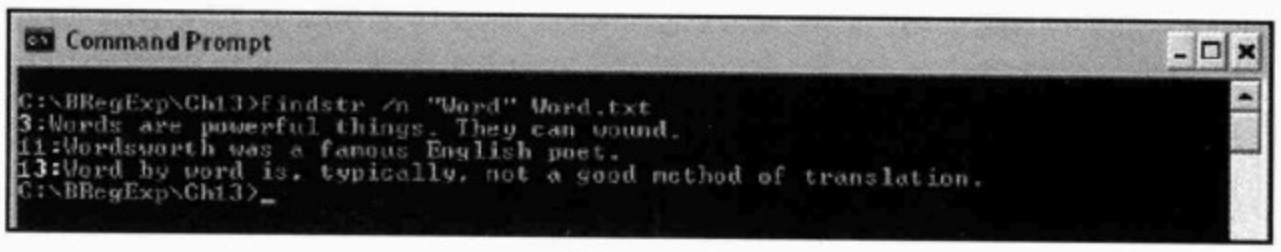


图 13-10

(4) 为保证搜索到包含字符序列 word 的所有行，可以使用 /i 命令行开关。

在命令行中输入下列命令：

```
findstr /n /i "Word" Word.txt
```

(5) 观察结果。现在七行都被显示出来了。因此，可以确认所有字符序列 word 的实例都匹配了。

(6) 接着，我们来看一下开始词边界元字符 \< 的作用。

在命令行中输入以下命令：

```
findstr /n /i "\<Word" Word.txt
```

(7) 观察结果，如图 13-11 所示。七行中只有四行被显示出来。而这四行中都包含字符序列 word 或 Word(不区分大小写)，而它们都位于一个字母字符序列的开始位置(事实上，是在能够称其为“词<word>”的开始位置)。

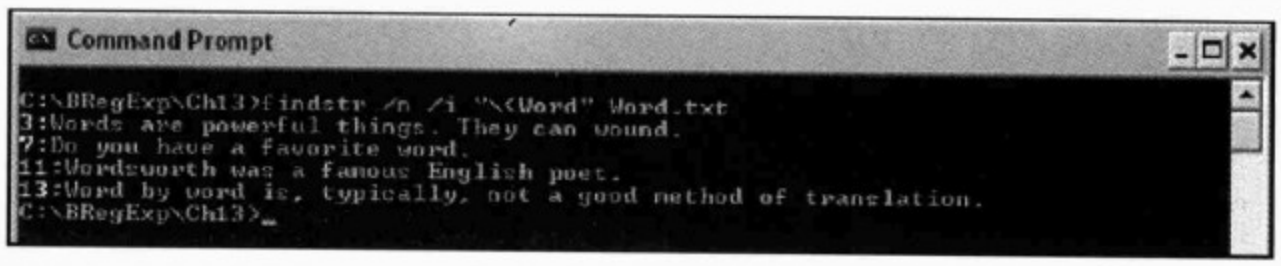


图 13-11

(8) 可以通过向正则表达式中添加表示词结束位置的元字符 \>，使匹配更加具体。即只匹配字符序列 word 或 Word 前面是词开始位置而后面是词结束位置的情况。

在命令提示符中输入以下命令：

```
findstr /n /i "\<Word\>" Word.text
```

(9) 观察结果，如图 13-12 所示。现在只有两行被显示出来，其中每一行中的字符序列 word 实际上都只是一个单词。严格来讲，词开始位置和词结束位置的元字符分别标记的是一个字符序列的开始位置和结束位置。实际上，这两个元字符都用于表示词的开始位置和结束位置。

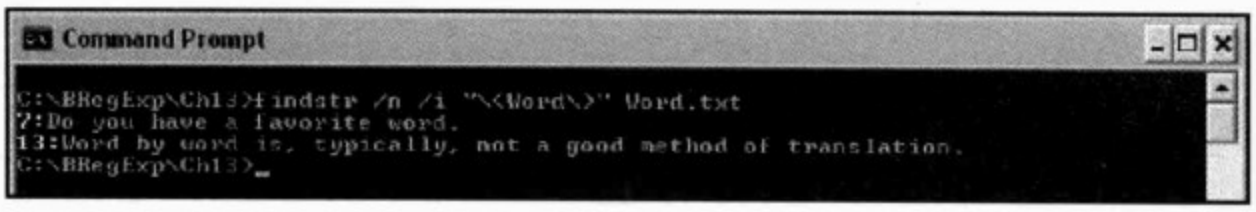


图　13-12

13.4　行开始位置和结束位置

findstr 实用程序提供了两种非常独特的方式来指定发生于行开始和结尾处的匹配。首先，是 /b 和 /e 开关，分别用于匹配行的开始和结束位置。其次，是 ^ 和 $ 元字符。

下面要用到的测试文件 Low.txt 的内容如下：

```
Low is the opposite of high.

A Ferrari isn't usually thought of as slow.

Slow, slow, quick, quick, slow

Slow, slow, quick, quick, slow.

Allow me to to pass please.

Lowering sky over a blackened sea.
```

试一试：匹配行的开始位置和结束位置

(1) 打开命令提示符窗口，并定位于测试文件 Low.txt 所在的目录下。

(2) 在命令提示符下输入以下命令：

```
findstr /n /i "Low" Low.txt
```

(3) 观察结果。六行文本都显示了出来，这是因为字符序列 low(注意 /i 开关表示不区分大小写的匹配)存在于所有这六行中。

(4) 然后，我们来测试 /b 开关，用它可以限制于匹配行的开始位置。

在命令行中输入以下命令：

```
findstr /n /i /b "Low" Low.txt
```

(5) 观察结果，如图 13-13 所示。现在只显示了两行，而每一行的前三个字符都是字符序列 Low。

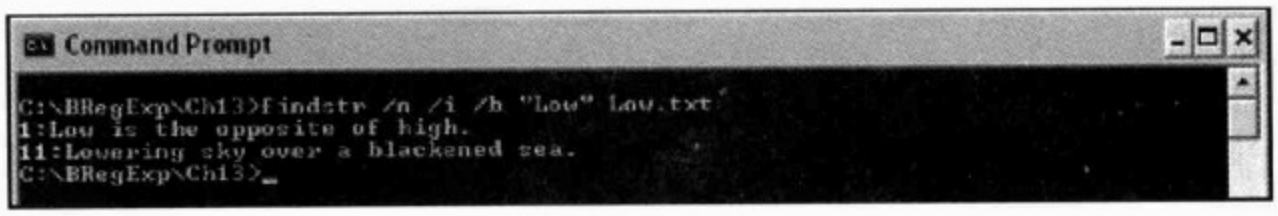


图 13-13

(6) 接着，测试 /e 开关，它将匹配限制于行的结束位置。

在命令行中输入以下命令：

```
findstr /n /i /e "Low" Low.txt
```

(7) 观察结果，结果如图 13-14 所示。

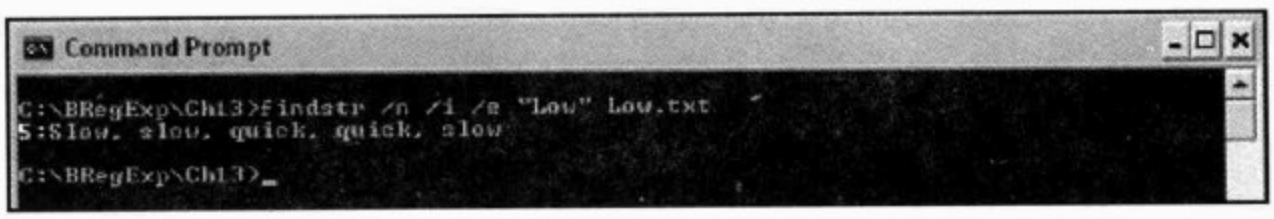


图 13-14

现在只显示了一行。如果觉得应该显示三行，那么就需要再仔细观察一下这些行。在另外两行中，最后的字符序列是 low 后跟一个句点字符——也就是说，low 不是行的结尾。这正是另外两行不匹配的原因所在。

(8) 还可以使用更常规的元字符来完成同样的搜索。要匹配一行的开始位置，可以使用元字符 ^。

在命令行中输入以下命令：

```
findstr /n /i "^Low" Low.txt
```

(9) 观察结果。显示于图 13-13 中的那些行再次显示出来。

(10) 最后，可以试一试使用 $ 元字符匹配行的结束位置。

在命令行中输入下列命令：

```
findstr /n /i "Low$" Low.txt
```

(11) 观察结果。只有一行显示——与图 13-14 相同。另外两行末尾的句点字符阻止了模式 Low$ 的成功匹配。

13.4.1 命令行开关的例子

本节举一些使用 findstr 命令行开关的例子。其中一些开关会直接起到正则表达式的作用——比如，/i 会导致执行不区分大小写的匹配。

13.4.2 /v 开关

/v 开关的作用是只显示那些不匹配的行。这个开关对于测试与标准不一致的数据非常有用。

例如，如果知道零件编号目录中的编号都由三个字母字符后跟三个数字组成，那么要想找到那些格式不正确的零件编号就非常简单。

这里要用的测试文件 PartNums2.txt 的内容如下：

```
ABC876

A2D993

AB2882

AEJ88

KHD945

HEW78R

H
```

在试验这个例子时，需要假设进行区分大小写的匹配。

试一试：使用 /v 开关

(1) 打开命令提示符窗口，并定位至测试文件 PartNums2.txt 所在的目录。

(2) 在命令行提示符中输入以下命令：

```
findstr /n /v "[A-Z][A-Z][A-Z][0-9][0-9][0-9]" PartNums2.txt
```

(3) 观察如图 13-15 所示的结果。注意显示的那些行中包含与模式 [A-Z][A-Z][A-Z][0-9][0-9][0-9]不匹配的零件编号。

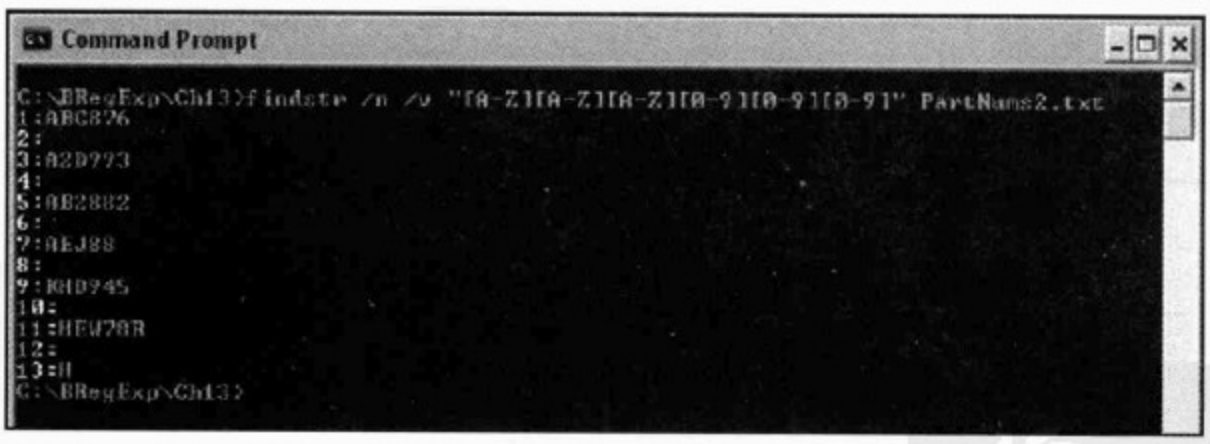


图　13-15

(4) 为了确认所有行中部分匹配而其他不匹配，可以在省略 /v 开关的情况下重新运行 findstr。

在命令行提示符下输入下列命令：

```
findstr /n "[A-Z][A-Z][A-Z][0-9][0-9][0-9]" PartNums2.txt
```

(5) 观察结果，如图 13-16 所示。通过比较图 13-15 和图 13-16，就会发现所有行分别

显示于这两个窗口中，但没有一行同时显示于两个窗口中。

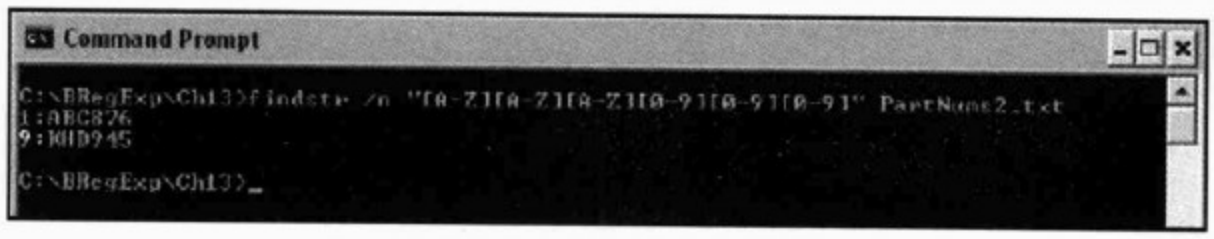


图 13-16

工作原理

使用/v 开关时，会显示一些空白行。因为这些行中都不包含与模式[A-Z][A-Z][A-Z][0-9][0-9][0-9]匹配的内容，它们没有匹配的基础。因此，这些行作为不匹配的实例显示出来。

在第 3 行，文本 A2D993 不匹配，因为它的第二个字符是一个数字，该数字不匹配模式中的第二个组件 [A-Z] 字符类。

在第 5 行，没有匹配文本 AB2882 是因为其第三个字符是数字，这与模式中的第三个组件 [A-Z] 字符类不匹配。

在第 7 行，没有匹配 ABJ88 是因为它只包含两个数字，不匹配第三个字符类[0-9]。

在第 11 行，文本 HEW78R 不匹配，因为其第六个字符是大写的 R，不匹配字符类[0-9]。

在执行第 4 步后得到如图 13-16 所示的结果。此时，第 1 行和第 9 行匹配，是因为它们都包含由三个大写的字母字符后跟三个数字组成的零件编号，而且恰好与模式[A-Z][A-Z][A-Z][0-9][0-9][0-9] 匹配。

13.4.3 /a 开关

/a 开关后跟一个冒号和一个或两个十六进制数。如果是一个十六进制数，则是控制文本(前景)颜色。如果是两个十六进制数，那么第一个十六进制数指定背景颜色，第二个十六进制数指定文本颜色。

表 13-4 十六进制数及其对应的颜色

十六进制数	指定的颜色
0	黑
1	蓝
2	绿
3	水绿
4	红
5	紫
6	黄
7	白
8	灰
9	淡蓝

(续表)

十六进制数	指定的颜色
A	淡绿
B	淡水绿
C	淡红
D	淡紫
E	淡黄
F	亮白

对于有些无意义的组合，白色背景上的白色文本，会被忽略。例如，如果输入以下命令：

```
findstr /n /i /a:00 "ABC[0-9]" Orders*.txt
```

指定在黑色背景上显示黑色文本，那么就会使用正常的黑底白字来显示。其他组合，如下面蓝色背景上的蓝色文本：

```
findstr /n /i /a:11 "ABC[0-9]" Order*.txt
```

是允许的，但基本没有什么用处——除非想让包含匹配项的一行开头显示一种颜色块。

图 13-17 显示的是使用几种 /a 开关的参数所得到的屏幕显示效果。

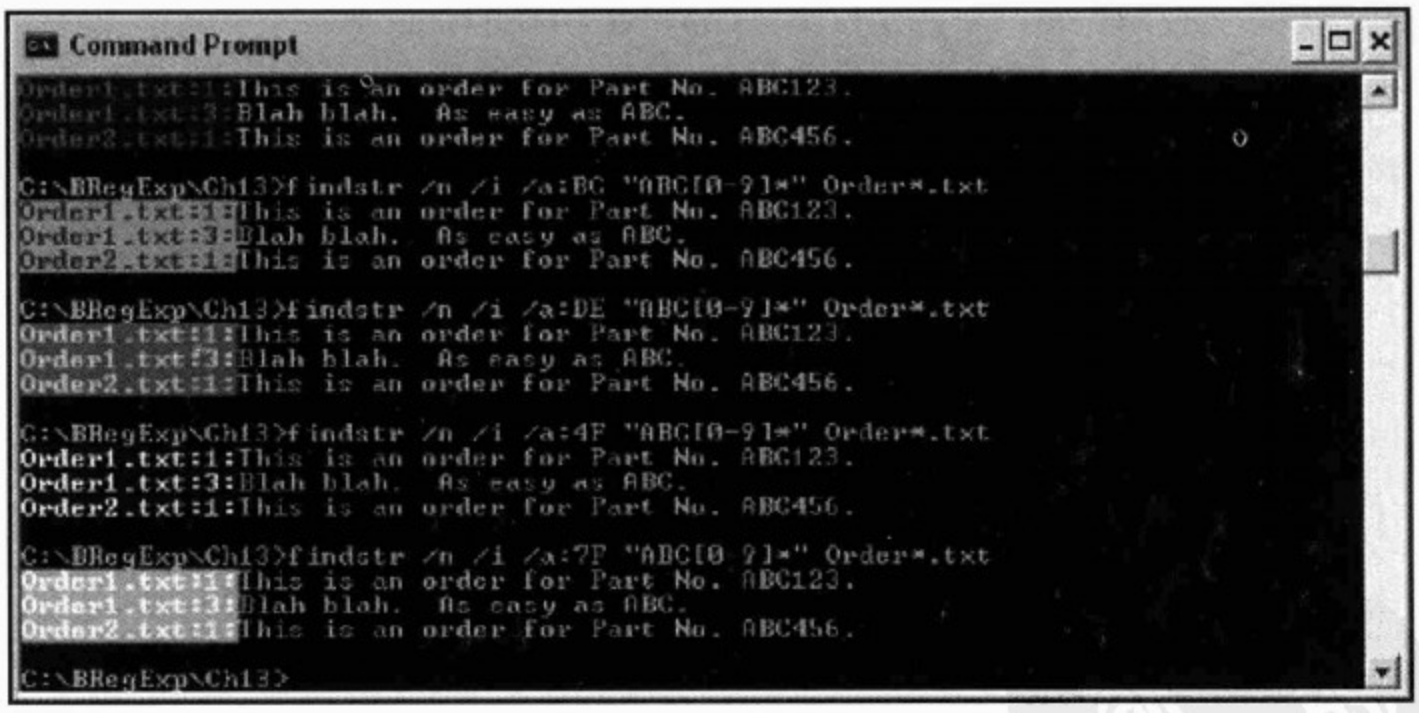


图　13-17

13.5　单个文件的例子

下面的几个例子同样演示了 findstr 实用程序的用法。由于 findstr 对限定符的支持有限，也就限制了其用途。

13.5.1 简单字符类的例子

这个例子使用测试文件 gray.txt 来演示 findstr 实用程序的用法，其内容如下：

```
gray
grey
greying
greyed
grapple
grim
goat
filigree
great
groat
gloat
Gray
Grey
```

问题定义如下：

匹配一个 g 后跟一个 r，后跟一个可选的 e 或 a，再后跟 y。

包含一个简单字符类的模式 gr[ae]y 可以用于匹配想要的文本。

试一试：简单字符类的例子

(1) 打开命令提示符窗口，并定位至 Gray.txt 所在的目录下。

(2) 在命令提示符中，输入以下命令：

```
findstr /n "gr[ae]y" Gray.txt
```

(3) 观察结果，如图 13-18 所示。

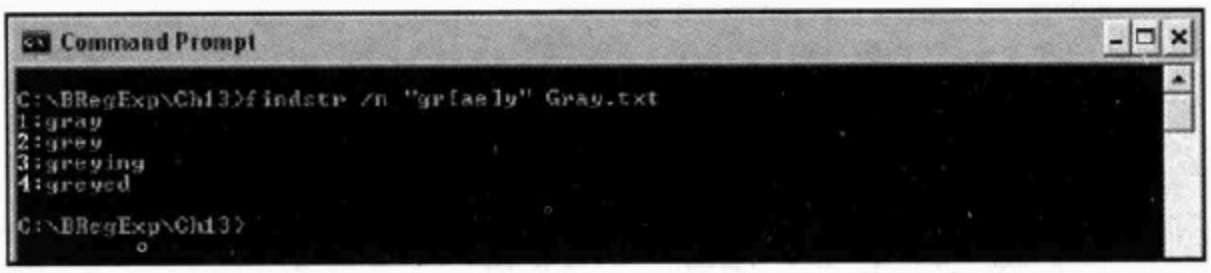


图 13-18

13.5.2 查找协议的例子

本例示范了一种查找 Internet 协议的简单技术。所使用的测试文件 Protocols.txt 内容为：

```
http://www.w3.org/

ftp://www.XMML.com/

mailto:someone@example.org
```

在命令行中，输入如下命令：

```
findstr /n "://" Protocols.txt
```

这会显示出所有包含 Internet 协议的行——本例中，即第 1 行和第 2 行。

13.6　多个文件的例子

findstr 最有价值的地方在于通过命令行可以一次搜索多个文件。这样就会比在编辑器或文字处理程序中每次打开一个文件更有效率。

本例演示如何使用 findstr 在多个文件中查找 HTTP URL。这里有三个简短的测试文件。URL1.txt 包含如下内容：

```
I found interesting information at http://www.w3.org/ on the XQuery specification.
```

URL2.txt 包含如下内容：

```
I wanted to find information about Microsoft SQL Server 2005 and the site at
http://www.microsoft.com/sql/ was very useful.
```

而 URL3.txt 则包含如下内容：

```
This document shouldn't be detected because the protocol, http, is omitted. The
site that I
visited was www.w3.org.
```

相应的问题定义可以描述为：

匹配字符序列 http 后跟一个冒号字符，后跟两个正斜杠字符。

试一试：查找 URL

(1) 打开命令提示符窗口，并定位到文件 URL1.txt、URL2.txt 和 URL3.txt 所在的目录中。

(2) 在命令行中输入下列命令：

```
findstr /n "http://" URL*.txt
```

(3) 观察结果，如图 13-19 所示。注意在图 13-19 中显示出来的 findstr 结果布局中的不足——一个文件的结果与另一个文件的结果前后相连。这种情况会在测试文本并不是整齐地基于行的格式、而是基于段落格式的时候发生。而且，findstr 在结果中显示的是包含匹配项的文本，不是真正的匹配项，这种不严密的反馈可能会导致混乱。如果看到这种结果混在一起的显示，就需要使用 /a 开关(前面有一个相关的例子)来使结果更清楚一些。

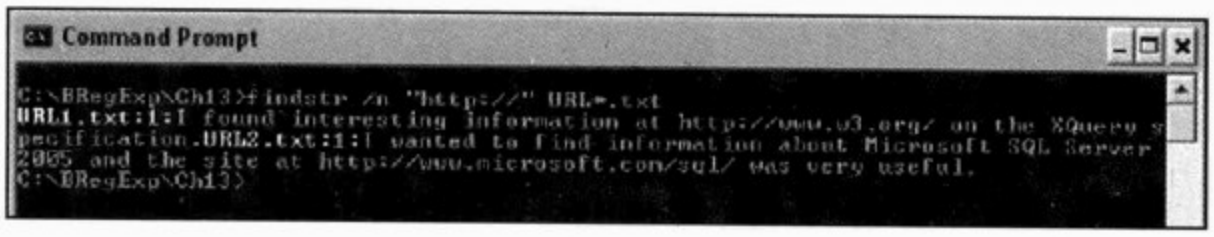


图 13-19

13.7 文件列表的例子

迄今为止，本章在命令行中使用文件名时都用到了一个通配符(比如在 URL*.txt 中)，但这是相对简单的。有时，可能需要搜索一些无法使用通配符的文件。而 /f 命令行开关就是为此准备的。

测试文件 Targets.txt 中，包含着一个文件列表：

```
URL1.txt
URL2.txt
URL3.txt
```

而文件 Data.txt 中则包含了一个非常简单的正则表达式：

```
http://
```

要把这两个文件结合起来，就需要用到 /g 命令行开关和 /f 命令行开关。开关 /g 的参数是一个文件名，该文件中包含着要搜索的数据(或正则表达式模式。译者注)。开关 /f 的参数也是一个文件名，这个文件中包含着被搜索的文件列表。另外，如果在文件列表中存在包含匹配项的文件，还会把相应的文件列表写入一个结果文件中。

试一试：使用 /g 开关和 /f 开关

(1) 打开命令提示符窗口，并定位至包含 Data.txt、Targets.txt、URL1.txt、URL2.txt 和 URL3.txt 文件的目录下。

(2) 在命令行提示符下输入以下命令：

```
findstr /g:Data.txt /f:Targets.txt > Results.txt
```

(3) 再在命令行提示符中输入下列命令：

```
Type Results.txt
```

(4) 观察结果。结果与前面的例子相同，但这次它们被传送到另一个结果文件中，而不像前一次那样只是简单地被输出到了屏幕上。

工作原理

参数 /g:Data.txt 表示正则表达式包含在文件 Data.txt 中。参数 /f:Targets.txt 表示文件 Targets.txt 中包含着要搜索的文件名。而结果则如命令行中的 > Results.txt 所指示的，被重新定向到另一个文件 Results.txt 中。

13.8　练习

下面的练习题旨在巩固本章所介绍的知识：

1. 假设零件编号的格式是三个字母字符后跟三个数字，而要检测的文件可以表示为“文件名*.扩展名”的形式。那么使用什么样的 findstr 命令能够显示出所包含的零件编号中第二个字符不是 L、M 或 N (而是其他任意大写字母)的那些行？

2. 请给出两个可能的 findstr 命令，以显示以 the 或 The 开头的行。

第14章

PowerGREP

PowerGREP 是一款功能强大的正则表达式工具。它是一款基于 Windows 平台的商业化产品，具有图形用户界面(Graphical User Interface，GUI)，并可实现 Unix 和 Linux 平台中的 grep、egrep 以及类似工具所具有的功能。可以在其网站 www.powergrep.com 上了解有关 PowerGERP 的更多信息。

与 findstr 实用程序相比，使用 PowerGREP 不必学习命令字符串以及相关参数。而且，PowerGREP 更完整地实现了正则表达式的功能。此外，它所能完成的替换操作也大大超出了 findstr 的功能范围。

在本章中将学习如下内容：

- 如何使用 PowerGREP 的用户界面
- 如何使用 PowerGREP 所支持的众多正则表达式功能
- 如何使用 PowerGREP 完成搜索或搜索及替换操作(包括对多个文件)

本章中的例子已经在 PowerGREP 2.3.1 中进行了测试。

14.1 PowerGREP 的界面

如果第一次使用 PowerGREP，打开这个程序的界面将会如图 14-1 所示。如果以前用过 PowerGREP，那么打开这个程序后会保留上一次用到的正则表达式模式、文件夹选项以及文件掩码(mask)。如果刚用过 PowerGREP，则会发现结果面板中还残留着上一次搜索的结果，如图 14-1 所示。

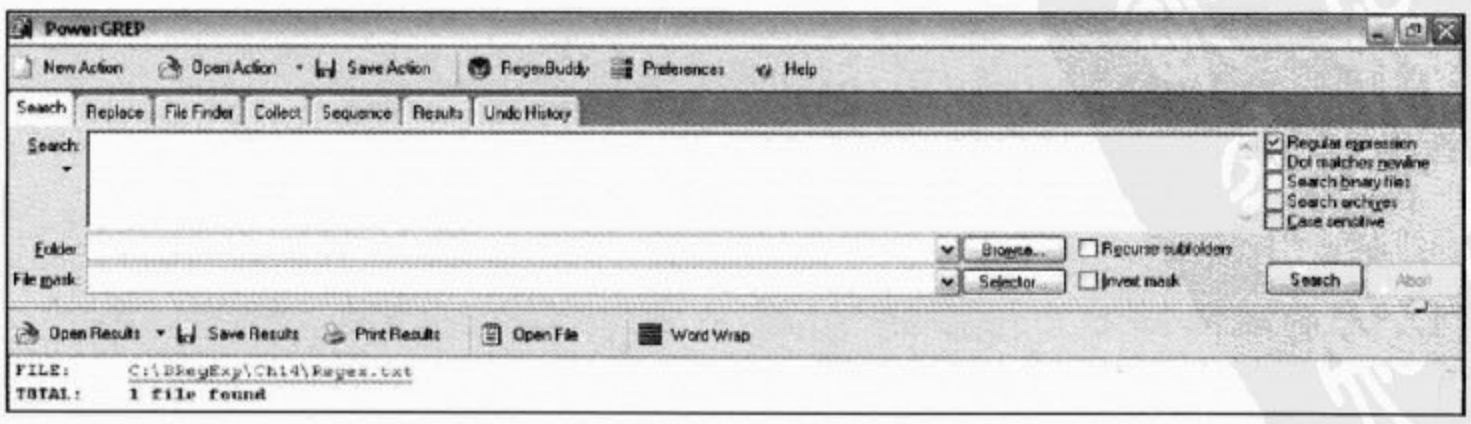


图 14-1

14.1.1　简单查找的例子

下面例子所用的测试文件是 Regex.txt，其内容如下：

```
This is regular but not an expression.

Here is a simple regular expression pattern: \d.

Regex is an abbreviation for regular expression.

Some people use the abbreviation regexp.

The plural of regex is regexes.
```

这段测试文本中包含着多处引用正则表达式的单词。有时使用的是术语 regular expression，有时使用的则是缩写词 regex，而且还会使用不常见的 regexp。

相应的问题定义如下：

匹配任何正则表达式或其缩写词的文本。

当然，如果想更好地转换成正则表达式，必须要对这个问题定义进一步提炼，使其表述更加精确。比如可以进一步表述如下：

匹配任何下列字符序列：

- 匹配直接量字符序列 regular expression
- 匹配直接量字符序列 regex
- 匹配直接量字符序列 regexp

要将此时的问题定义表达为正则表达式有多种方式。其中一种就是使用简单的交替选择：

```
(regular expression|regex|regexp)
```

这种方式的好处在于简洁易读。

另一种方式是从想要的匹配项中提取出公共的字符，即：

```
reg(ular expression|ex|exp)
```

这个模式稍短一些，但可读性较差。

如果要使用最简单的模式，那么还有一种就是：

```
reg(ular expression|(ex)p?)
```

不过，相比使用简单交替选择的较长模式而言，这个模式的可读性也最差。

试一试：简单的查找

(1) 打开 PowerGREP，并在 Search 文本区中输入(regular expression|regex|regexp)。

(2) 确保选中 Regular expression 复选框。

(3) 在 Folder 文本框中，输入 C:\BRegExp\Ch14(假设下载的测试文件解压在 C 盘的 BRegExp 目录下。如果把文件解压缩在其他地方，调整此处的路径)。

(4) 在 File mask 文本区中，输入 Regex.txt 并单击 Search 按钮。

(5) 观察结果，如图 14-2 所示。共找到六个匹配项。如果将 PowerGREP 显示的结果与 Regex.txt 的内容进行比较，就会发现所有字符序列 regular expression、regex 或 regexp 都被匹配了。

(6) 在 Search 文本区中输入另一个模式 reg(ular expression|ex|exp)，并观察结果。

(7) 在 Search 文本区中输入第三个模式 reg(ular expression|(ex)p?)，并观察结果。

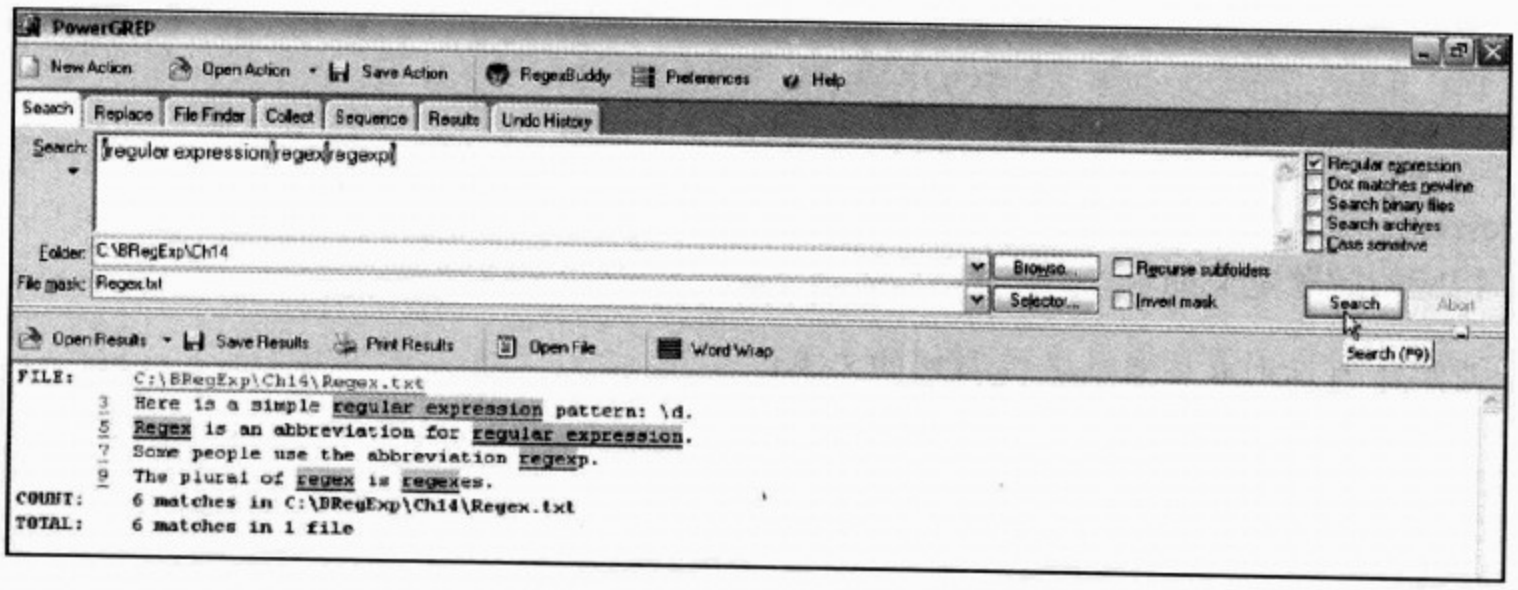


图 14-2

使用后面两个模式搜索得到的结果应与图 14-2 显示的结果相同，但第 6 步和第 7 步中在 Search 文本区中输入的正则表达式是不相同的。

工作原理

先来看第一个正则表达式 (regular expression|regex|regexp)。匹配过程非常直观，因为可以匹配的直接量字符串有三个，每一个都是可选的。正则表达式引擎首先尝试匹配 regular expression，如果没有成功，会尝试匹配 regex，如果还未成功，则尝试匹配 regexp。

在第 1 行中，字符序列 regular 和 expression 都存在，但它们之间有其他字符，所以没有与模式匹配。

第 3 行和第 5 行中的字符序列 regular expression 与模式的第一个选项匹配。

在第 5 行(一次)、第 7 行(一次)和第 9 行(两次)匹配的是 regex。

模式 regexp 则没有找到匹配(参考本节后面的评注)。

现在再来看看第二次使用的正则表达式 reg(ular expression|ex|exp) 和第三次使用的正则表达式 reg(ular expression|(ex)p?)的情况。

在第 1 行中，字符序列 reg 匹配，但却没有能与模式中后三个选项匹配的字符序列。所以第 1 行没有匹配项。

在第 3 行和第 5 行中，字符序列 reg 匹配，因而会继续测试三个选项。第一个选项 ular expression 也匹配这两行中的后续字符。

在第 5 行(一次)、第 7 行(一次)和第 9 行(两次)，虽然字符序列 reg 匹配，但第一个选项 ular expression 不匹配，不过第二个选项 ex 或(ex)却匹配。所以这几行中的字符序列 regex 都被匹配。

本例中使用的匹配策略存在一些缺陷。如果在试验这个例子的过程中发现了这个缺陷，那么将有机会在本章后面的练习中纠正这个问题。

14.1.2　Replace 选项卡

在 PowerGREP 的标签中有一个 Replace 选项卡，通过它用户可以定义文本如何替换。

图 14-3 显示的是刚运行完前面例子后的 Replace 选项卡。其中在 Find 选项卡中查找到的结果仍然显示着。这样就可以方便地了解有哪些匹配项，需要替换哪些匹配项。

下面的练习会以字符序列 regex 替换文件中出现的字符序列 regular expression、regex 或 regexp。

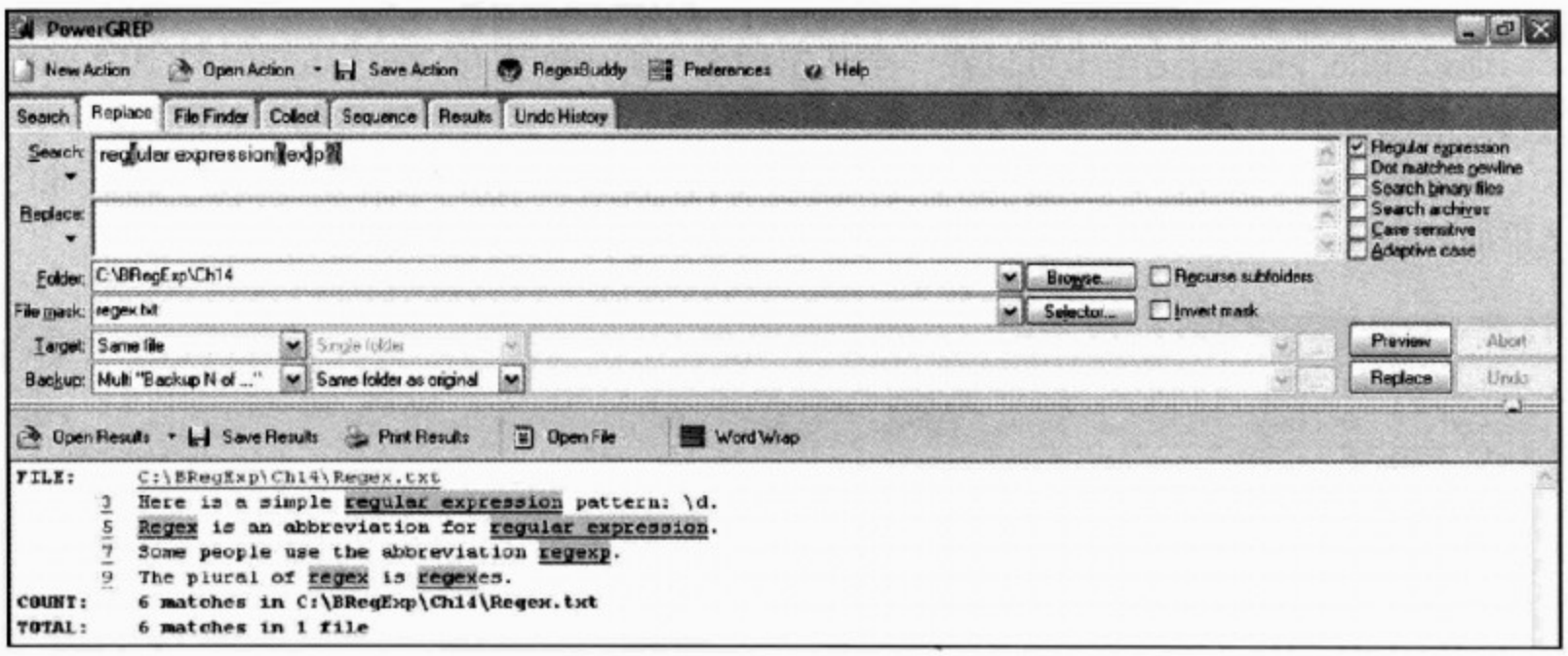


图　14-3

试一试：Replace 选项卡

(1) 单击一下 Replace 选项卡。此时界面如图 14-3 所示。

(2) 在 Replace 文本区中，输入文本 regex，并单击 Preview 按钮。

(3) 观察结果面板中显示的情况，如图 14-4 所示。如果是在屏幕中查看结果，那么原始的文本默认会以黄色显示，而作为替换的文本会以绿色显示。比如，在第 3 行中，原始文本 regular expression 是以黄色被突出显示的，而作为替换的文本 regex 则以绿色被突出显示。

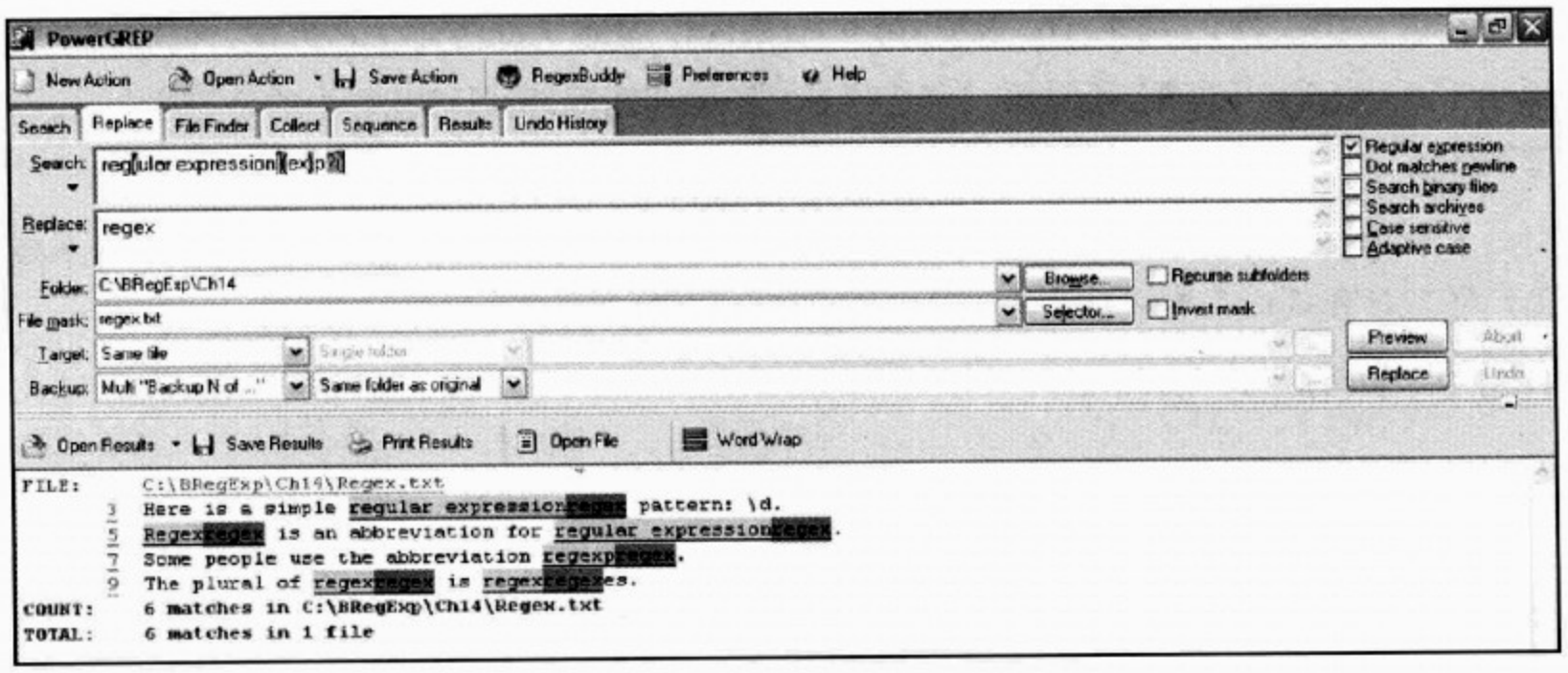


图 14-4

14.1.3 File Finder 选项卡

通过 File Finder 选项卡可以在一个选定的文件夹中搜索包含与指定正则表达式匹配的内容的文件。

在运行 Replace 选项卡这个例子后，File Finder 的界面会如图 14-5 所示。然而，此时结果面板中的结果并不像我们期望的那样，它显示的是前一次操作的结果。

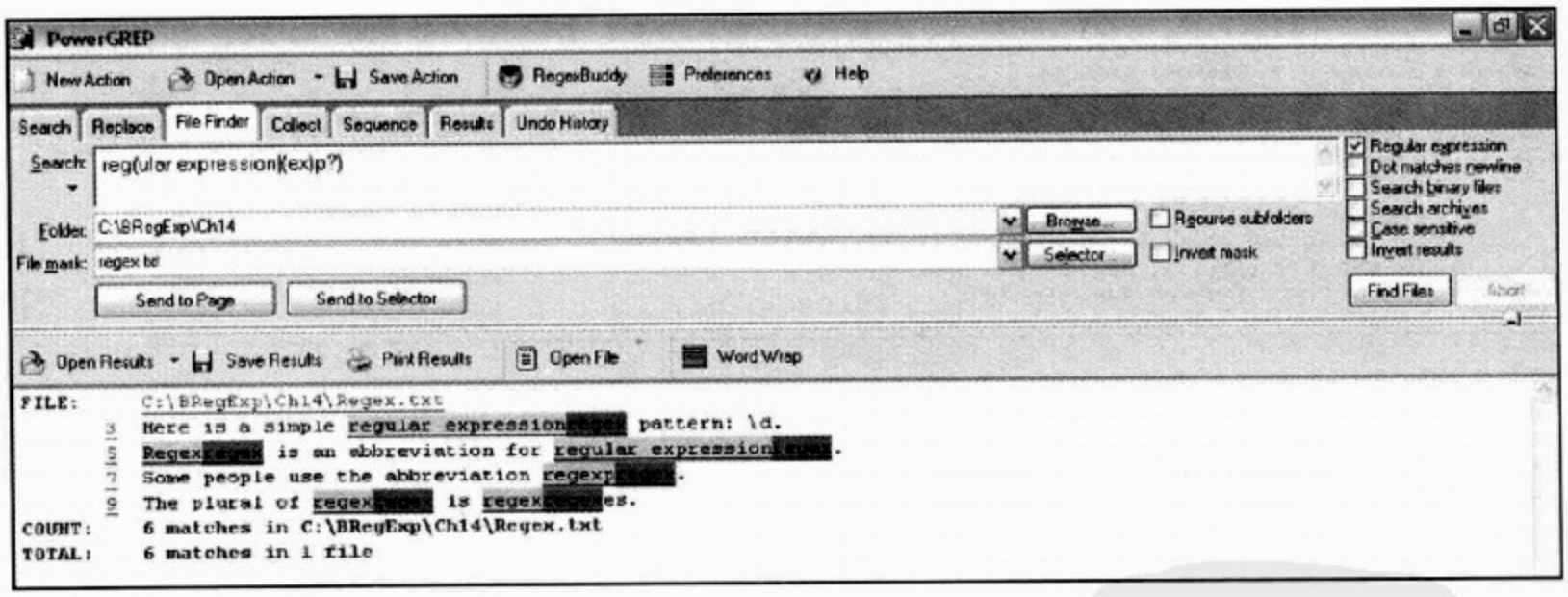


图 14-5

此时，只需单击一次 Find Files 按钮，与 File Finder 选项卡对应的结果就会显示出来，如图 14-6 所示。

在这个例子中，只有一个文件 Regex.txt，它包含与正则表达式 reg(ular expression|(ex)p?) 匹配的内容。

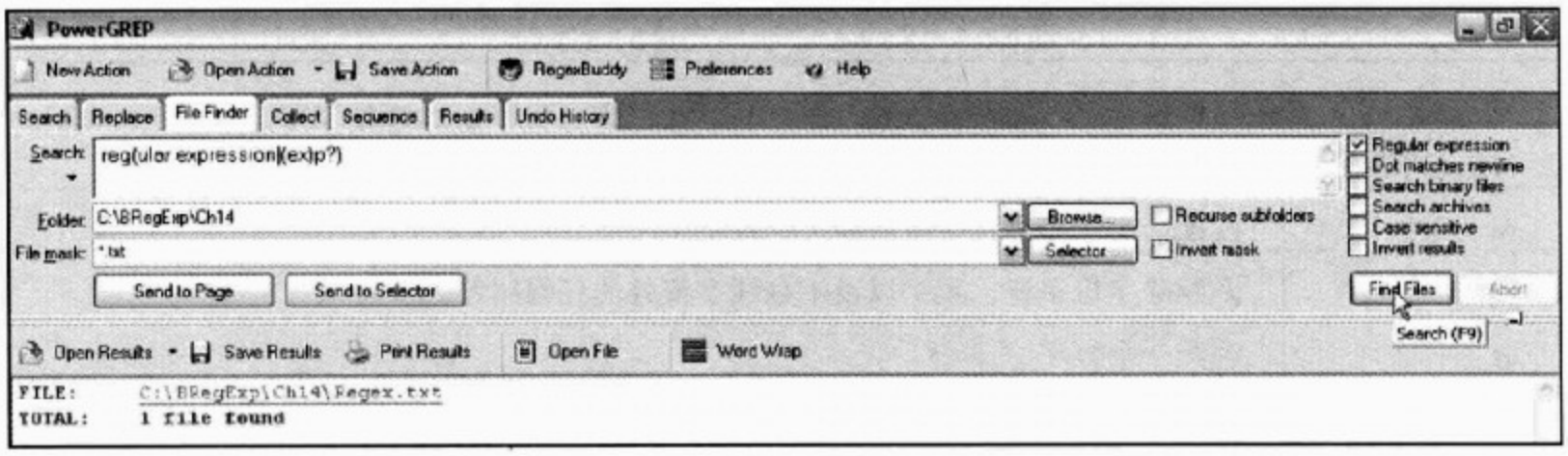


图　14-6

14.1.4　语法着色

此处简单介绍 PowerGREP 中的一个配置选项——语法着色，关闭它会对使用这个工具更有帮助。在默认情况下，PowerGREP 会为正则表达式应用语法着色。本章中的一些屏幕截图中就带有相应的语法着色效果。但是，当正则表达式比较复杂时，对语法着色则会导致混乱。

要关闭语法着色功能，单击 PowerGREP 工具栏中的 Preferences 图标。并在 Search Boxes 选项卡中取消对 Apply syntax coloring to regular expressions 复选框的选定。这样，正则表达式就会以纯文本形式显示了。

14.1.5　其他选项卡

PowerGREP 中的 Collect 选项卡和 Sequence 选项卡中包含以其他方式使用正则表达式的选项，但本章不会更多地介绍它们。Results 选项卡只是简单地显示结果，同样的结果也会显示在其他选项卡的结果面板中。Undo History 选项卡可以根据备份情况，对替换后的结果进行撤销。同样，这些选项卡的使用本章也不介绍。但是，这些附加的功能可以让你不必熟悉本书后面介绍的编程语言，就能熟练地使用正则表达式。

14.2　PowerGREP 支持的元字符

与 Microsoft Word 相比，PowerGREP 对正则表达式的支持更强。而且，它对正则表达式的支持也强于 OpenOffice.org Writer 和 findstr 实用程序。它对正则表达式的支持，能与 Komodo Regular Expression Toolkit 相提并论。但，PowerGREP 的优势在于你可以像使用一个实用工具一样使用它，无须编程就能用正则表达式来完成实际任务。

表 14-1 总结了 PowerGREP 支持的元字符。其中多数元字符将在本节或本章后面进行深入的介绍，或者通过例子进行验证。

表 14-1　PowerGREP 支持的元字符

元　字　符	说　　明
(句点字符)	几乎匹配任何字符
\w	匹配一个字母字符、数字和下划线字符
\W	匹配除字母字符、数字以及下划线字符外的任何字符
\d	匹配一个数字
?	限定符——匹配出现零次或一次的字符或块
*	限定符——匹配出现零次或多次的字符或块
+	限定符——匹配出现一次或多次的字符或块
{n,m}	限定符——匹配至少出现 n 次，最多出现 m 次的字符或块
{n,}	限定符——匹配至少出现 n 次，最多无限次的字符或块
[...]	字符类
[^ ...]	取反的字符类
(|)	交替选择
\1 ……	反向引用
^	位置元字符——匹配一行开头的位置
$	位置元字符——匹配一行结尾的位置
\b	词边界的位置
\<	词开始的位置——不支持
\>	词结束的位置——不支持

14.2.1　数字和字母字符

PowerGREP 支持 \w、\W 和 \d 元字符。

下面实验中要使用测试文件 AlphaNumTest.txt，其内容如下：

```
This line contains numbers, 1 2 3, and text . . . Blah, blah.

ABC

DEF 890

The next line has nonalphabetic characters.

?! "£$%^&*()_

1234567890
```

试一试：匹配数字和字母字符

(1) 打开 PowerGREP，并切换到 Search 选项卡。选中 Regular expression 复选框。

(2) 在 Search 文本区中输入简单的正则表达式 \w。

(3) 在 Folder 文本框中输入 C:\BRegExp\Ch14(假设把下载的文件解压缩在 C 盘的 BRegExp 目录中。如果未解压缩到此位置，请将此处的路径调整为正确的路径)。

(4) 在 File mask 文本框中，输入 AlphNumTest.txt，单击 Search 按钮并观察结果。如图 14-7 所示。所有字母字符、数字和下划线字符都与 \w 元字符匹配。每个匹配项都是一个单独的字符或数字。

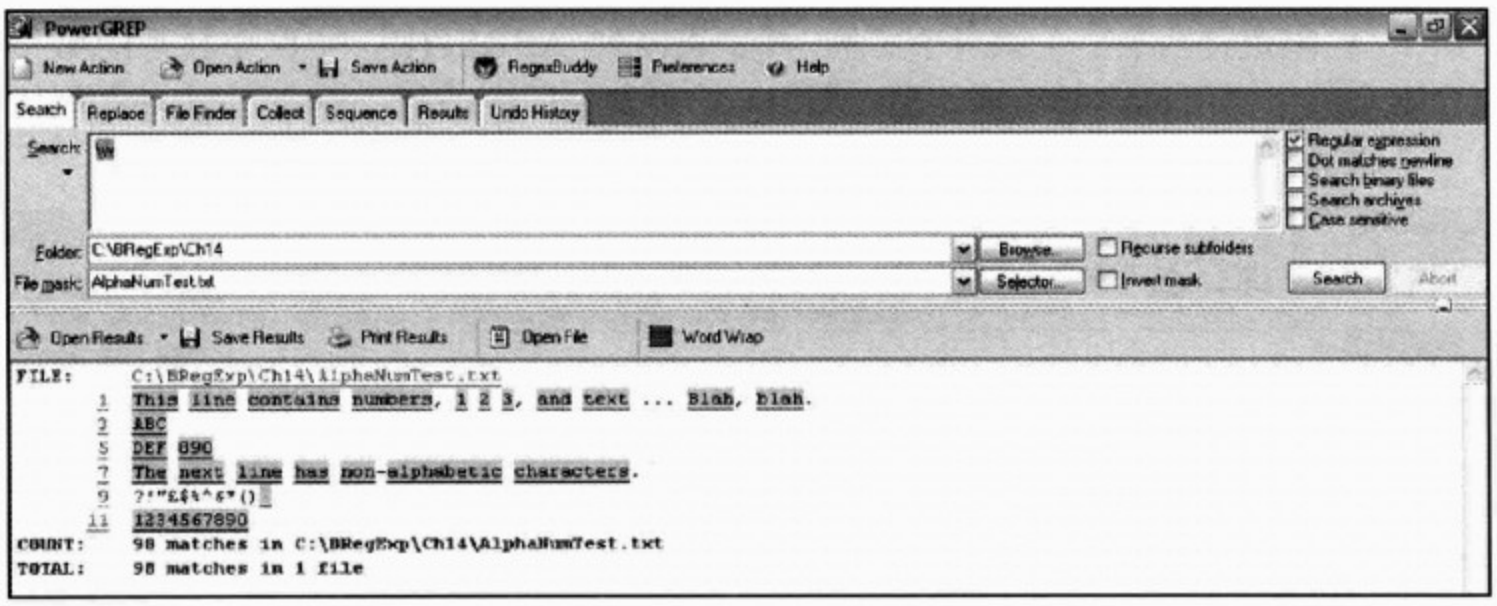


图　14-7

(5) 将 Search 区域的模式修改为 \d，并观察结果。此时，\d 元字符匹配的是测试文件中的数字，每个匹配项都是一个单个数字。

14.2.2　限定符

PowerGREP 支持 ?、* 和 + 限定符，也支持 {n,m} 语法。

下面将会用到的测试文件是 ABDEF.txt，其内容如下：

```
AB123DEF

AB1DEF

ABDEF

AB12DEF

AB1234567890DEF
```

注意数据的每一行中都包含字符序列 AB 后跟零个或多个数字，再跟字符序列 DEF。

试一试：使用限定符

(1) 打开 PowerGREP，并切换到 Search 选项卡。选中 Regular expression 复选框。

(2) 在 Search 文本框中，输入 AB\d?DEF。

(3) 在 Folder 文本框中输入 C:\BRegExp\Ch14(假设下载的文件解压缩在 C 盘的 BRegExp 目录中。如果未解压缩到此位置，请将此处的路径调整为正确的路径)。

(4) 在 File Mask 文本框中输入 ABDEF.txt，单击 Search 按钮并观察结果。结果中显示的匹配项为 ABDEF (零个数字)和 AB1DEF(一个数字)。

(5) 将 Search 文本区中的模式修改为 AB\d*DEF，单击 Search 按钮，并观察结果，如图 14-8 所示。

注意，测试文本中所有行都匹配了。因为每一行中都包含字符序列 AB 后跟零个或多个数字，再后跟字符序列 DEF。

(6) 将 Search 文本区的模式修改为 AB\d+DEF，单击 Search 按钮，并观察结果。

由于 + 限定符匹配一个或多个实例，而字符序列 ABDEF(图 14-8 中显示的第 5 行)中包含零个数字，所以这次不再匹配。图 14-8 中其他匹配的行仍然会匹配，因为它们都包含一个或多个数字。

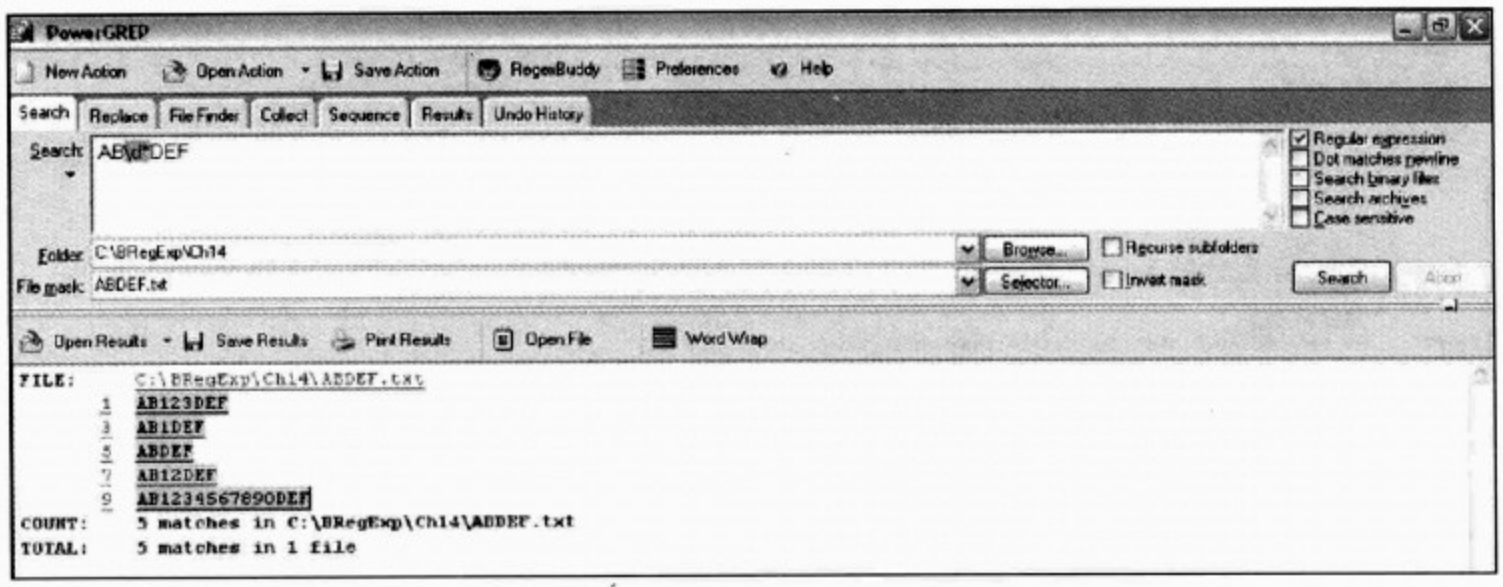


图 14-8

(7) 将 Search 文本区的模式修改为 AB\d{0,3}DEF，单击 Search 按钮，并观察结果(图 14-9)。

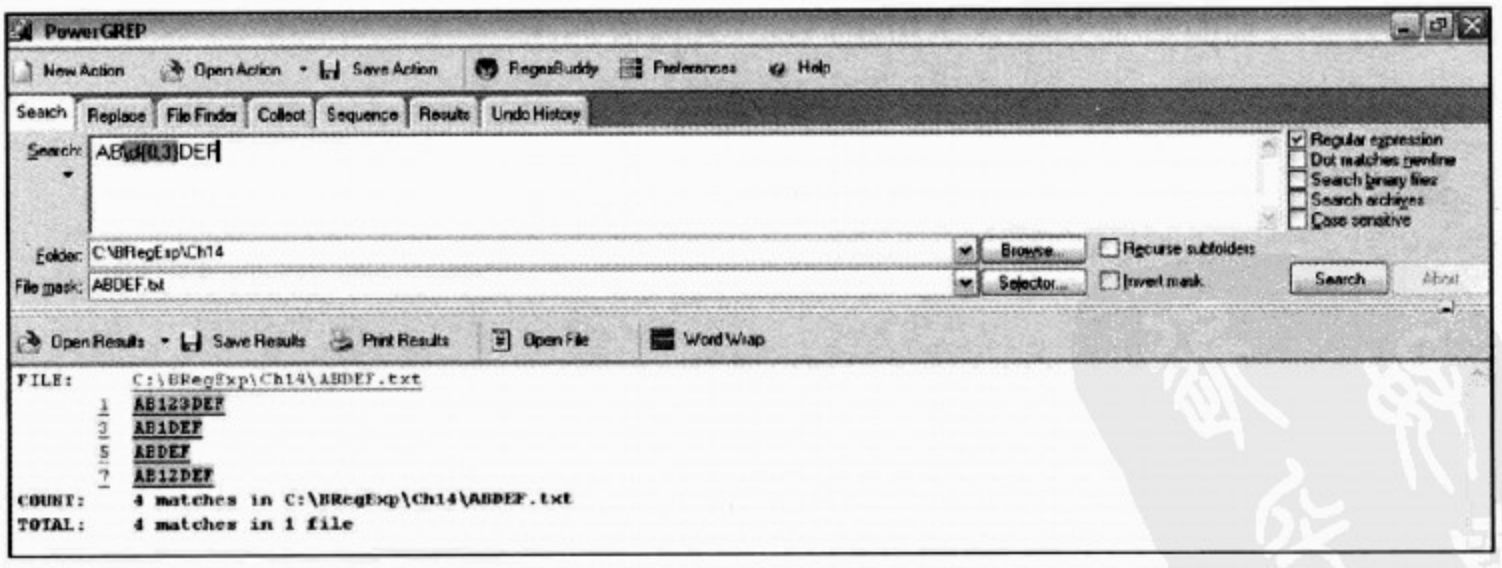


图 14-9

注意，图 14-9 中，没有匹配第 9 行，因为 AB1234567890DEF 中包含 10 个数字，而模式 AB\d{0,3}DEF 要求最多出现三次数字。

(8) 将 Search 文本区的模式修改为 AB\d{2,}DEF，单击 Search 按钮，并观察结果。

此时匹配的文本是 AB123DEF、AB12DEF 和 AB1234567890DEF。其中每个匹配都最少包含两个数字，最多的数字个数没有限制。

14.2.3　反向引用

PowerGREP 支持反向引用。也就是说，正则表达式中的每一对圆括号都会创建一个组。正则表达式中用圆括号括住的组会按照数字顺序进行捕获，并通过 \1、\2 等来引用。

下面这个例子使用 PowerGREP 和反向引用将 Star Training 中的 Star 全部替换成 Moon。为方便起见，测试文件 StarOriginal.txt 的内容在这里再展示一次。注意其中第一个 Star Training 出现的位置。

```
Star Training Company

Starting from May 1st Star Training Company is offering a startling special offer
to our regular customers - a 20% discount when 4 or more staff attend a single Star
Training Company course.

In addition, each quarter our star customer will receive a voucher for a free
holiday away from the pressures of the office. Staring at a computer screen all day
might be replaced by starfish and swimming in the Seychelles.

Once this offer has started and you hear about other Star Training customers
enjoying their free holiday you might feel left out. Don't be left on the outside
staring in. Start right now building your points to allow you to start out on your
very own Star Training holiday.

Reach for the star. Training is valuable in its own right but the possibility of a
free holiday adds a startling new dimension to the benefits of Star Training
training.

Don't stare at that computer screen any longer. Start now with Star. Training is
crucial to your company's wellbeing. Think Star.
You replace Star only when it precedes Training.
```

试一试：用反向引用完成替换

(1) 打开 PowerGREP，并切换到 Replace 选项卡。确保选中 Regular expression 复选框。

(2) 在 Search 文本区中输入(Star)(*)(Training)；在 Replace 文本区中输入 Moon\2\3。

(3) 在 Folder 文本框中输入 C:\BRegExp\Ch14(假设把下载的文件解压缩在 C 盘的 BRegExp 目录中。如果未解压缩到此位置，请将此处的路径调整为正确的路径)。

(4) 在 File mask 文本框中输入 StarOriginal.txt 并单击 Preview 按钮。这里如图 14-10 所示。可能需要水平滚动窗口区域才能看到全部匹配项。在屏幕中，匹配的文本以黄色显示，而作为替换的文本则以绿色显示。如图 14-10 所示，所有将要替换的地方都没有问题。所以可放心地进行下一步——完成替换。

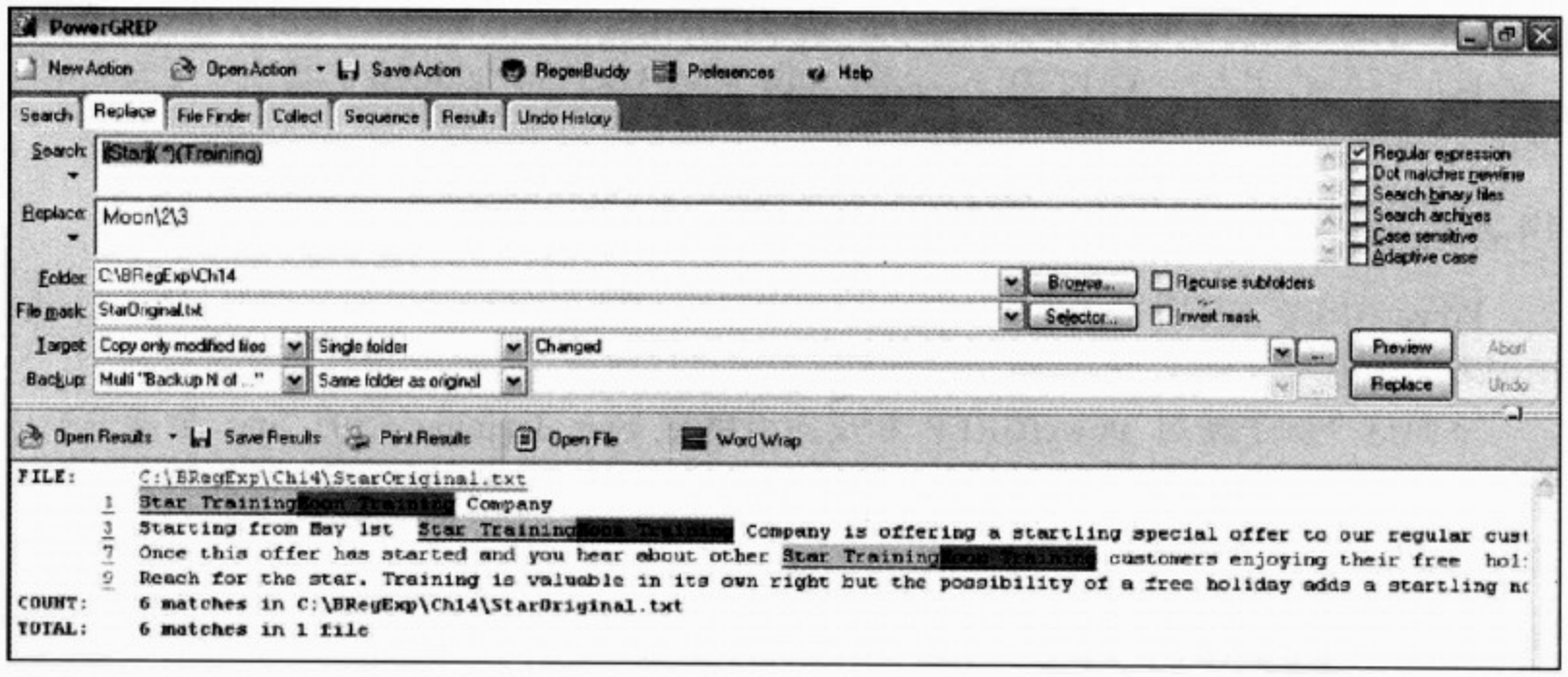


图 14-10

PowerGREP 中的术语“target(目标)”指的是搜索和替换操作后的一个文件。 其中还有许多选项，这里仅示范其中几个。

(5) 在 Target 区域的第一个下拉列表中，选择 Copy only modified files。

(6) 在 Target 区域的第二个下拉列表中，选择 Single folder。

(7) 在 Target 区域的第三个文本区下拉列表中，输入 C:\BRegExp\Ch14\Changed。此处，也可以通过浏览选择其他目标文件夹。

(8) 在 Backup 区域的第一个下拉列表中，选择 *.bak。

(9) 在 Backup 区域的第二个下拉列表中，选择 Single folder。

(10) 在 Backup 区域的第三个文本区下拉列表中输入 C:\BRegExp\Ch14\Changed。图 14-11 显示的是这一步后的外观图。

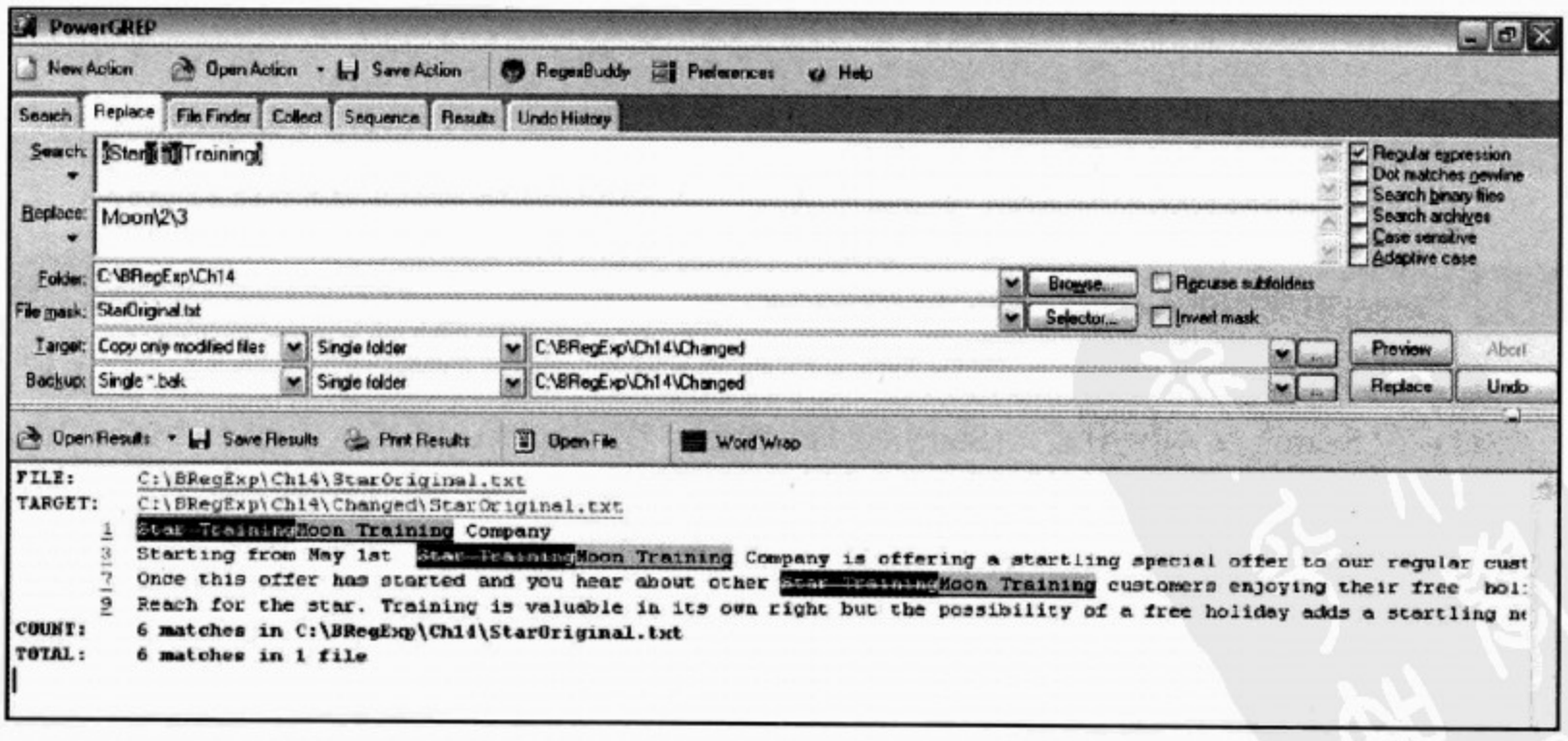


图 14-11

第 5 步到第 7 步会把修改后的文件放到 C:\BRegExp\Ch14\Changed 目录中，而且文件名与源文件名保持一致。

第 8 步到第 10 步会把原始文件备份到 C:\BRegExp\Ch14\Changed 目录中，并为其加上 .bak 扩展名。图 14-12 中显示的是添加到 Changed 文件夹中的文件。

PowerGREP 中对于目标和备份文件所使用的相对路径相当不明确。如果在此使用 . 缩写词，并认为这会将备份文件保存到的源文件相同的目录中，那就错了。使用 . 缩写词会导致备份文件被放到 PowerGREP 的安装目录中。

所以，最安全的方法就是放弃使用相对路径，只使用绝对路径。

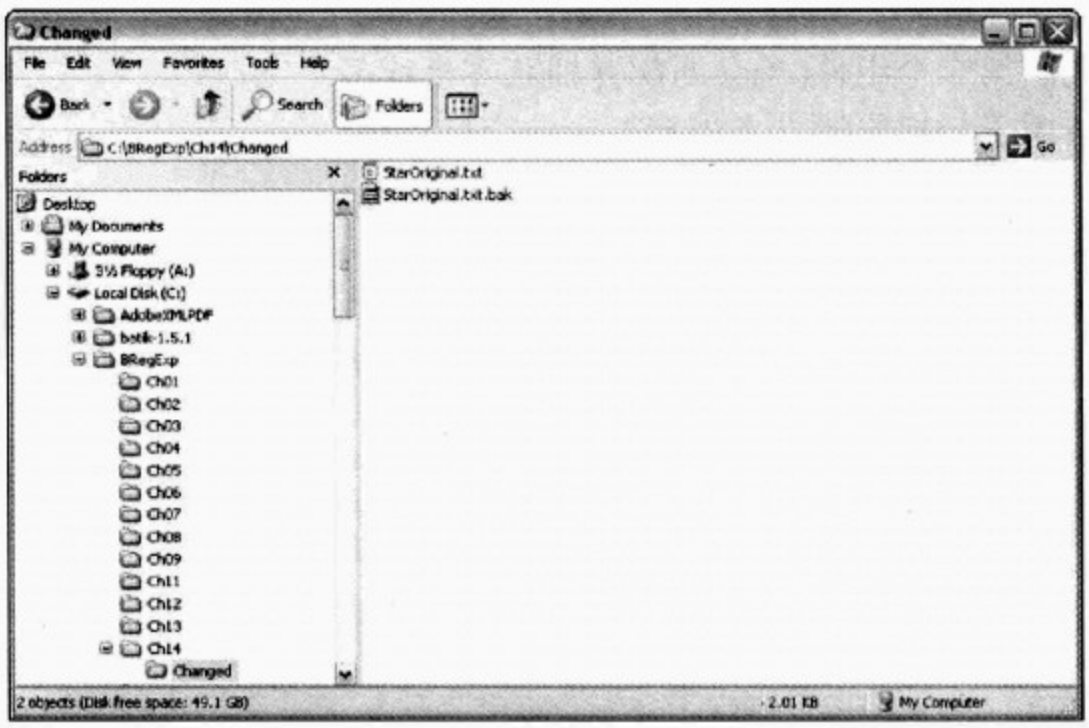


图　14-12

替换后的内容被保存到 C:\BRegExp\Ch14\Changed\StarOriginal.txt 中。该文件的内容以 Komodo 2.5 编辑器显示(为显示完整的路径)的结果如图 14-13 所示。

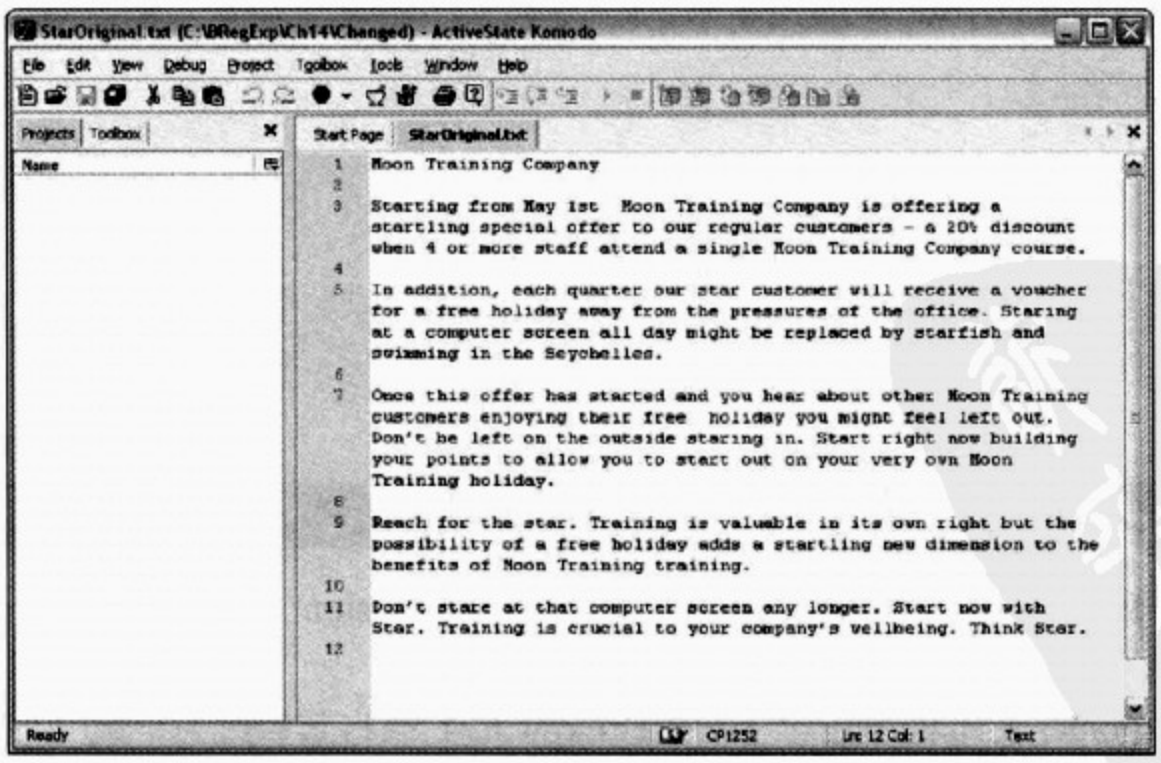


图　14-13

注意，在预览(Preview)时看到的所有字符序列 Star 都已被替换成字符序列 Moon。

工作原理

本节的主要内容是反向引用的工作原理。与 PowerGREP 特定相关的内容在前面“试一试”部分中已有所说明。

本例中使用的匹配模式是 (Star)(*)(Training)。这个模式会匹配字符序列 Star 后跟零个或多个(在测试文本中的可能是一个或多个)空白符和字符序列 Training。

这个正则表达式的第一个组件(Star)匹配字符序列 Star，同时将 Star 捕获并保存到 \1 中。但在搜索和替换的替换部分中不能用 \1。因为字符序列 Moon 会在替换模式中占据 \1 的位置(表示替换)。

正则表达式的第二个组件(*)会匹配并捕获零个或多个空白符，将它们保存在 \2 中。捕获到的空白符用于替换原始文本中相对应的空格。另一种方法是用一个文本空格字符替换一个或多个空格字符。

正则表达式的第三个组件(Training)匹配并捕获字符序列 Training，将其保存在\3 中。\3 在替换过程中只是简单地从原始文本中将字符序列 Training 复制到替换后的文本中。

最终的结果就是如果 Star 出现在零个或多个空格及字符序列 Training 之前，那么就会被替换成字符序列 Moon。

14.2.4 交替选择

PowerGREP 支持交替选择。本章的第一个例子就是一个交替选择的例子(Regex.txt 作为测试文件)。

14.2.5 行位置元字符

PowerGREP 支持 ^ 和 $ 这两种行边界元字符。^ 元字符匹配行中第一个字符前的位置，而 $ 元字符则匹配行中最后一个字符后的位置。

测试文件 LineEndTest.txt 的内容如下：

```
This is here.

Look at this.

This is a theatre.

Theatre is as stimulating as this.
```

试一试：使用行位置元字符

(1) 打开 PowerGREP，并确保选中了 Regular expression 复选框。

(2) 在 Search 文本区中输入模式 ^This。

(3) 确保 Folder 文本框中包含 C:\BRegExp\Ch14 或将其修改为下载文件的保存目录。

(4) 在 File mask 文本框中，输入 LineEndTest.txt。单击 Search 按钮，并观察结果，

如图 14-14 所示。记住，PowerGREP 中的默认匹配是不区分大小写的。注意，只有位于一行开头处的字符序列 This 才被匹配。

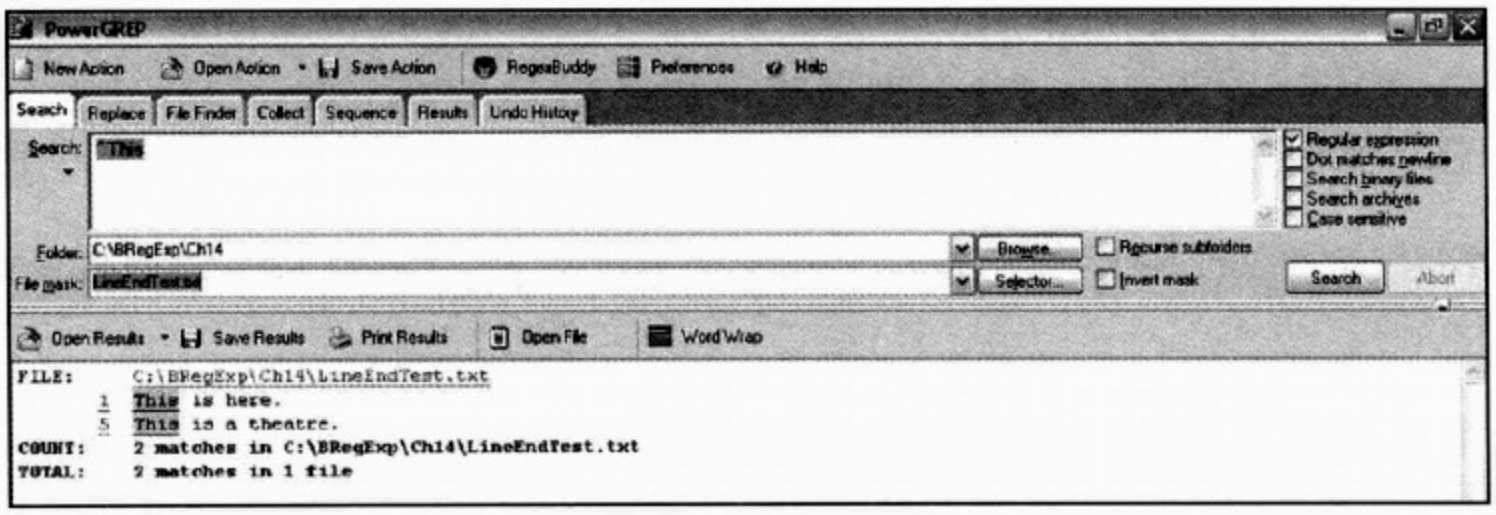


图　14-14

(5) 将 Search 文本区中的模式修改为 this\.$。如果一行的最后一个字符是句点，那么模式 this$将不会与本例测试文件中的 this 匹配。

(6) 单击 Search 按钮，并观察结果，如图 14-15 所示。两个位于行尾的字符序列 this. 都被匹配了。

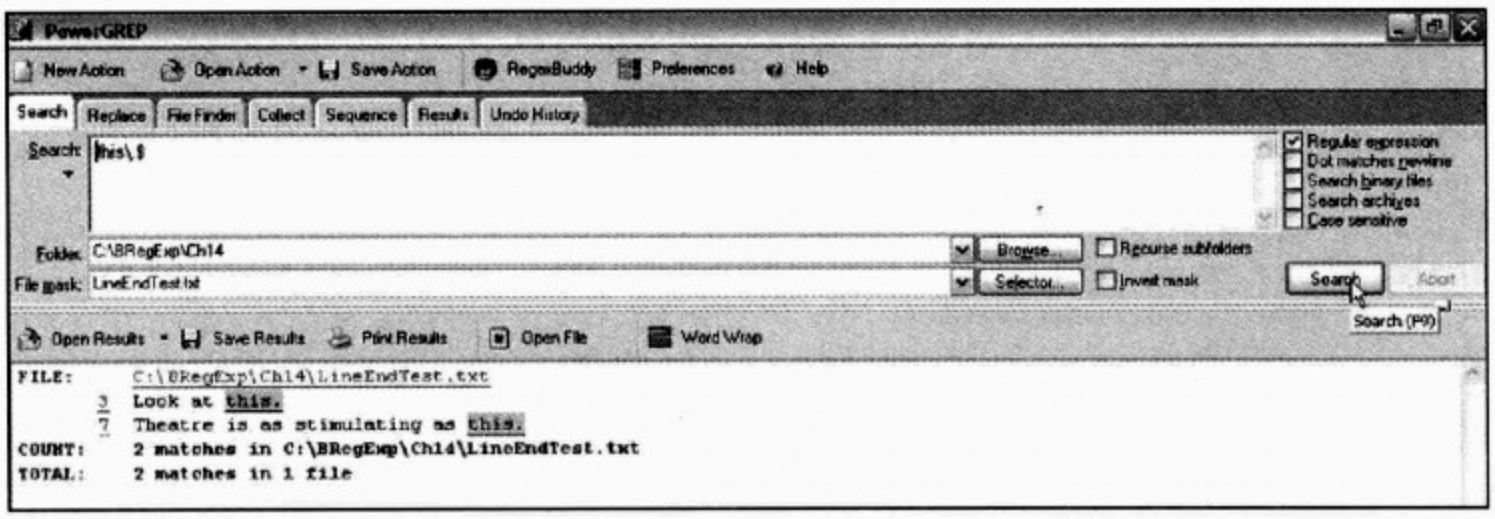


图　14-15

工作原理

首先，看一下模式 ^This 的匹配过程。这个模式首先匹配一行第一个字符之前的位置，然后尝试匹配字符序列 This。也就是说，它匹配位于一行开始处的字符序列 This。

改成模式 this\.$ 后，this 后跟一个句点再跟一个$元字符，此时，字符序列 this 后跟一个句点会被匹配，然后再尝试匹配行结尾的位置。只有 this. 是一行最后五个字符的情况才会匹配。

14.2.6　词边界元字符

PowerGREP 只支持 \b 词边界元字符。它不支持 \< 和 \> 这两个词边界元字符。

测试文件 Cat.txt 的内容如下：

```
Catalonia is a region of Spain.
```

```
"Scat," he said.

A caterpillar later becomes a butterfly.

I love my cat.

The cat sat on the mat.
```

试一试：使用词边界位置元字符

(1) 打开 PowerGREP，并确保选中了 Regular expression 复选框。

(2) 在 Search 文本区中输入模式 \bcat。

(3) 在 Folder 文本框中输入 C:\BRegExp\Ch14(如果把下载的文件放在其他位置，应该适当修改此处的路径)。

(4) 在 File mask 文本框中输入 Cat.txt，单击 Search 按钮，观察结果，如图 14-16 所示。注意，只有作为字母字符序列——通俗来讲，就是一个单词——的前三个字符的 cat 才会匹配。

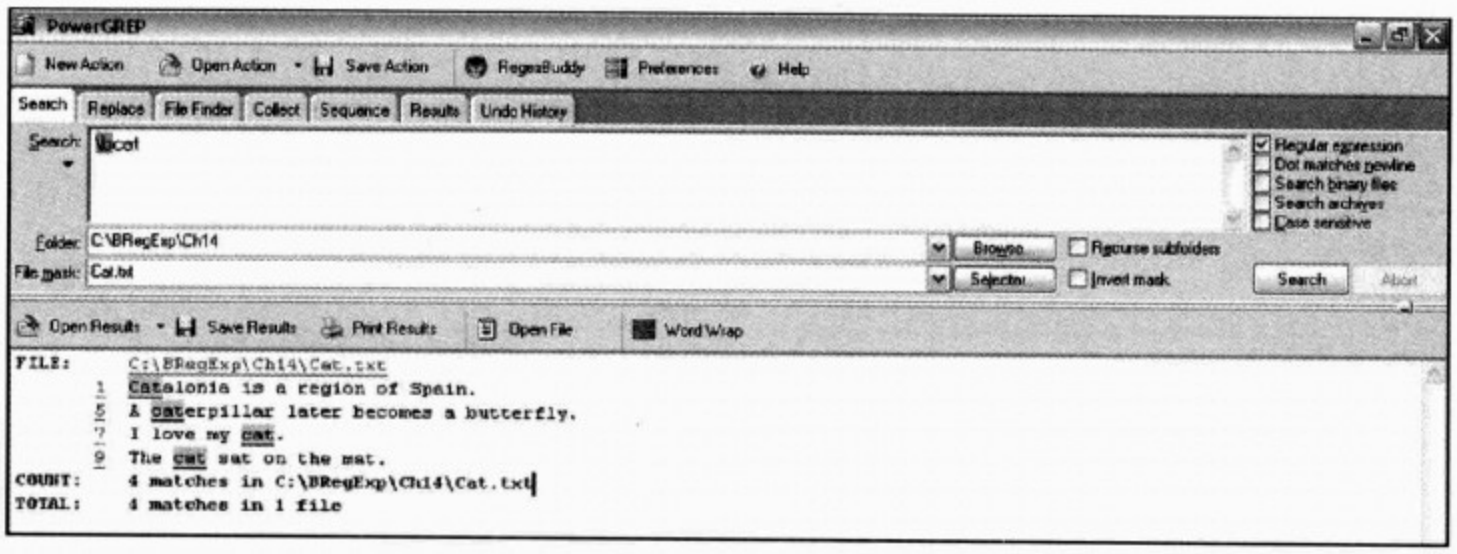


图 14-16

(5) 将 Search 文本区的模式修改为 cat\b，单击 Search 按钮，并观察结果，如图 14-17 所示。此时，只有当字符序列 cat 是一个字母字符序列的最后三个字符时才匹配。

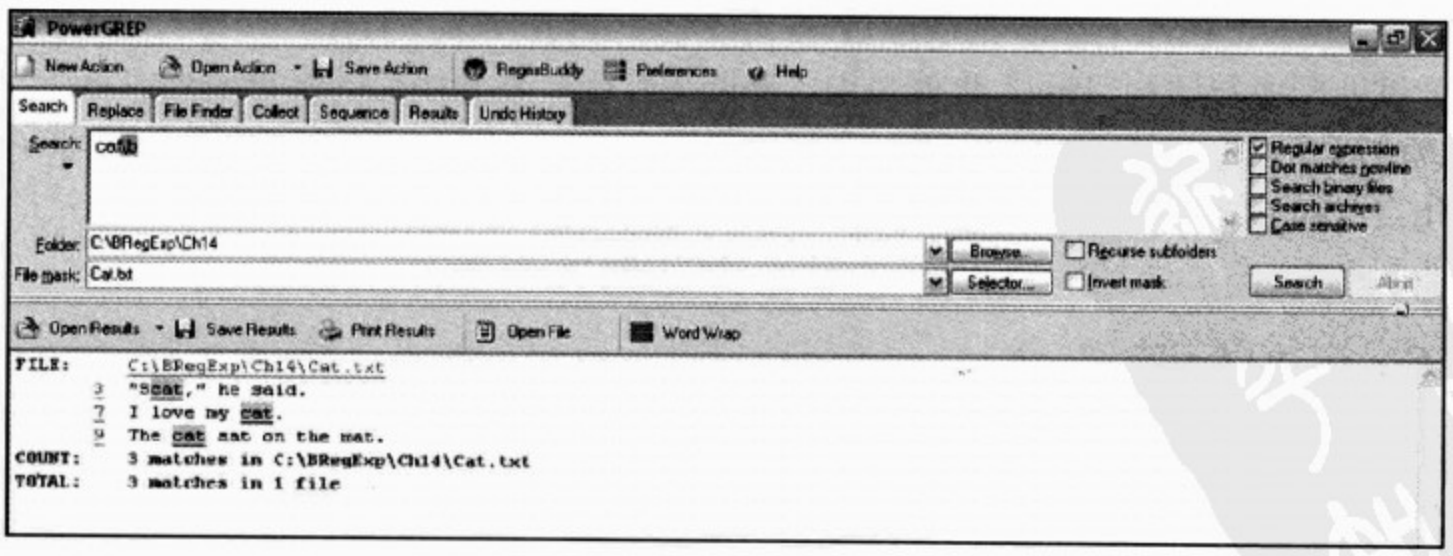


图 14-17

(6) 将 Search 文本区的模式修改为\bcat\b，单击 Search 按钮，并观察结果，如图 14-18 所示。

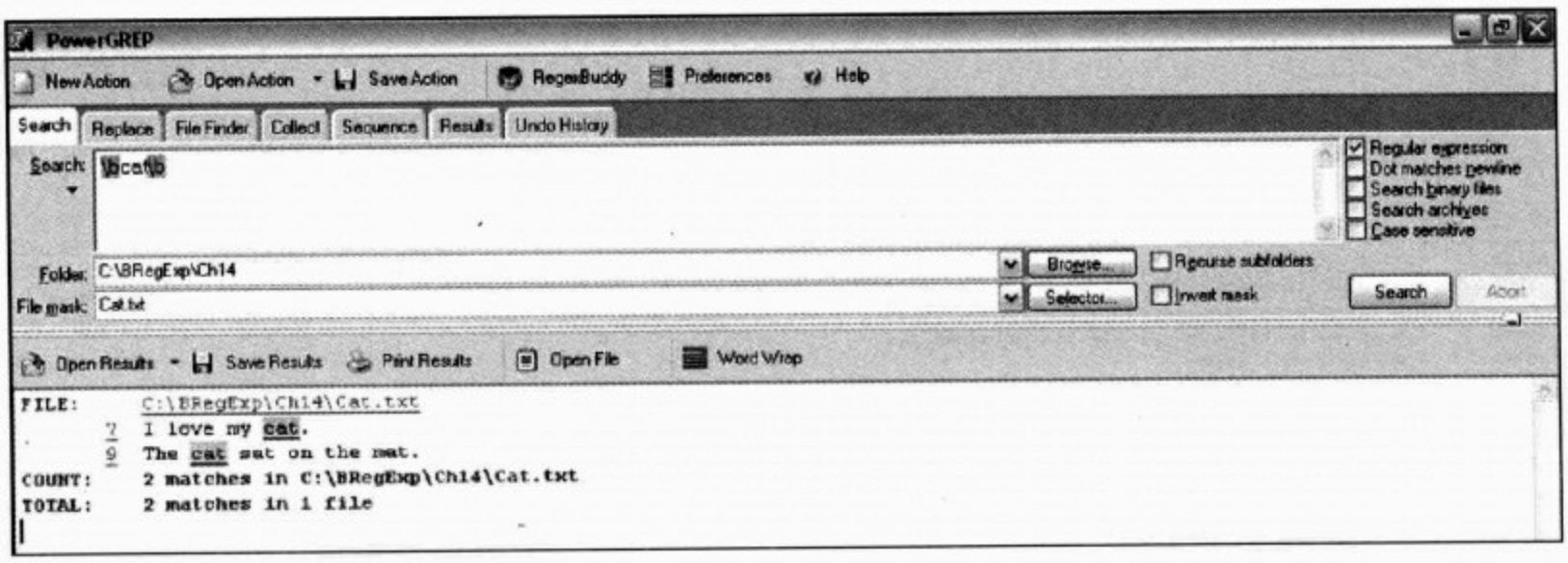


图　14-18

工作原理

模式 \bcat 首先会匹配一个位于非字母字符和字母字符之间的边界位置。因为 \b 元字符后跟一个字母字符序列，这种情况下，它的功能就是一个匹配词的开始位置的元字符，所以，\bcat 匹配位于一个单词开始位置的三个字符序列 cat。

模式 cat\b 匹配字符序列 cat 后跟一个词边界的位置。因此，cat\b 会匹配位于一个单词(如 cat 或 Scat)结尾处的 cat。

模式 \bcat\b 只匹配字符序列 cat 之前和之后同时是一个词边界位置的情况。换句话说，模式 \bcat\b 只匹配单词 cat。

14.2.7　向前查找和向后查找

PowerGREP 支持向前查找和向后查找。

下面的例子使用 StarOriginal.txt 作为测试文件，这个文件在本章前面使用过。

试一试：使用向前查找和向后查找

(1) 打开 PowerGREP，确保选中了 Regular expression 复选框。

(2) 在 Search 文本区中输入模式 Star(?=.)。

(3) 在 Folder 文本框中输入 C:\BRegExp\Ch14 或根据下载的测试文件的保存位置调整此处的路径。

(4) 在 File mask 文本框中输入 StarOriginal.txt。单击 Search 按钮，并观察结果，如图 14-19 所示。

(5) 将 Search 文本区中的模式修改为 (?<=with)Star，单击 Search 按钮，并观察结果。

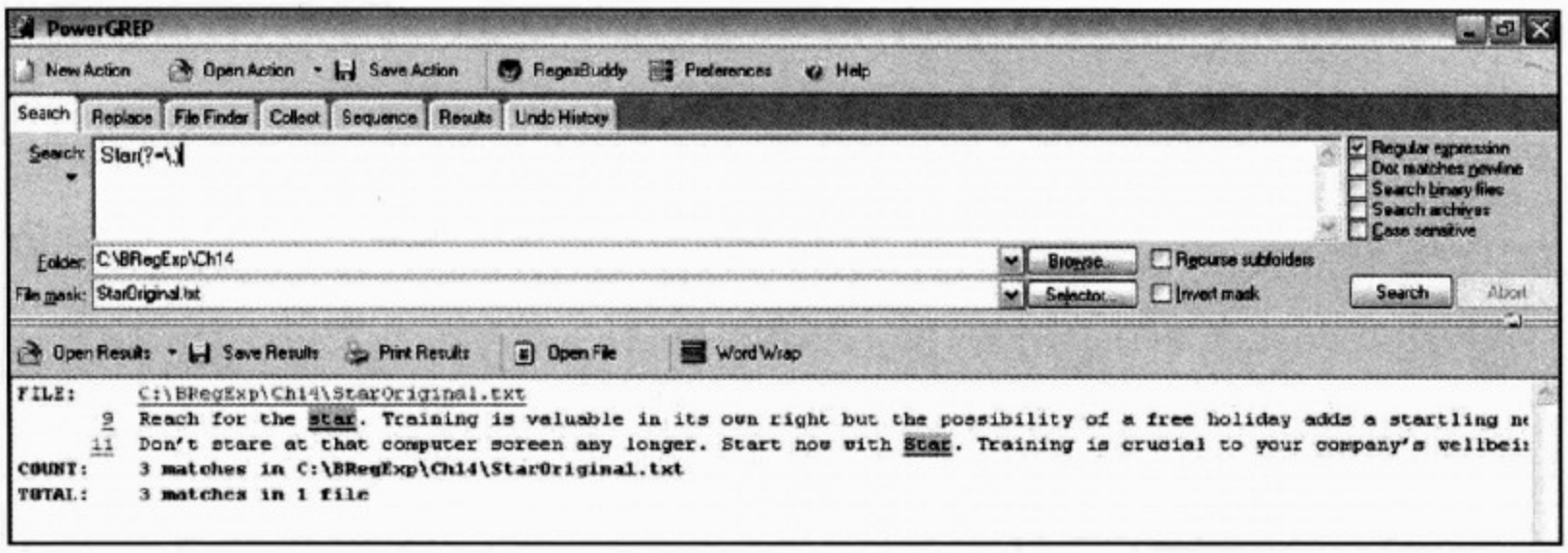


图 14-19

工作原理

首先，我们来分析模式 Star(?=\.)。模式开始处的 Star 会与其直接量匹配。由于 PowerGREP 默认以不区分大小写的方式进行匹配，所以字符序列 star 和 Star 都会匹配。事实上，STAR 和 sTar 也会匹配(但它们没出现在测试文本中)。但之后的向前查找则对匹配提出了条件。模式 (?=\.) 作为一个向前查找组件，含义是在 Star 与其直接量匹配后，如果 Star(或其他匹配项)后面并不直接跟着一个句点(以 \. 元字符表示)，那么匹配最终会失败。

模式 (?<=with)Star 中包含的是一个向后查找组件，它表示只有当字符序列 Star(或其他匹配项)前面是字符序列 with 后跟一个空格的时候才会匹配成功。

14.3 复杂一点的例子

本节介绍在 PowerGREP 中使用正则表达式功能的比较复杂一些的例子。PowerGREP 中最有用的部分就是它能够跨文件查找匹配项。为节省版面，本例只使用两个文件，且每个都很短。

14.3.1 查找 HTML 中的水平线(<hr>)元素

本例的目标是从多个文档中查找所有 HTML 水平线(<hr>)元素的标签。

第一次问题定义可能会如下：

匹配所有 HTML/XHTML 的水平线元素。

很明显，必须在理解 hr 元素标签结构的基础上进一步优化问题定义。

hr 标签可能会简单如下：

```
<hr>
```

也可以采用大写形式——在HTML 中通常如此。或者可能会带有 HTML 式的属性——属性值不加双引号：

```
<hr width=50% color=#990066 size=4 />
```

或者会如 XHMTL 中那样给属性值加上双引号：

```
<hr width="50%" color="#990066" size="4" />
```

或者也可能使用单引号：

```
<hr width='50 %' color='#990066' size='4' />
```

同时注意，在 XHTML 风格的标签中，该元素的右尖括号之前还会有一个正斜杠。更全面的问题定义如下：

匹配一个 < 字符后跟字符序列 hr(任意大小写)，后跟可选的空白符，后跟零个或多个字符，后跟可选的空白符，后跟一个可选的正斜杠，再后跟一个 > 字符。

与前面问题定义对应的模式可以写成这样：

```
<hr *.* */?>
```

下面是例子中要用到的测试文档。首先是 HorizRule1.html 的内容：

```
<html>
<head>
<title>Horizontal Rule 1</title>
</head>
<body>
<p>This file contains a horizontal rule with no attributes.</p>
<hr />
</body>
</html>
```

再是 HorizRule2.html 的内容：

```
<html>
<head>
<title>Horizontal Rule 1</title>
</head>
<body>
<p>This file contains a horizontal rule with three attributes.</p>
<hr width="50%" color="#990066" size="4" />
</body>
</html>
```

试一试：查找水平线(hr 元素)

(1) 打开 PowerGREP，并确保选中 Regular expression 复选框。

(2) 在 Search 文本区中输入模式 <hr *.* */?>。

(3) 确保 Folder 文本框中包含 C:\BRegExp\Ch14 或者根据存放下载文件的位置调整此处的路径。

(4) 在 File mask 文本框中输入 Horiz*.html，单击 Search 按钮，并观察结果，如图 14-20 所示。

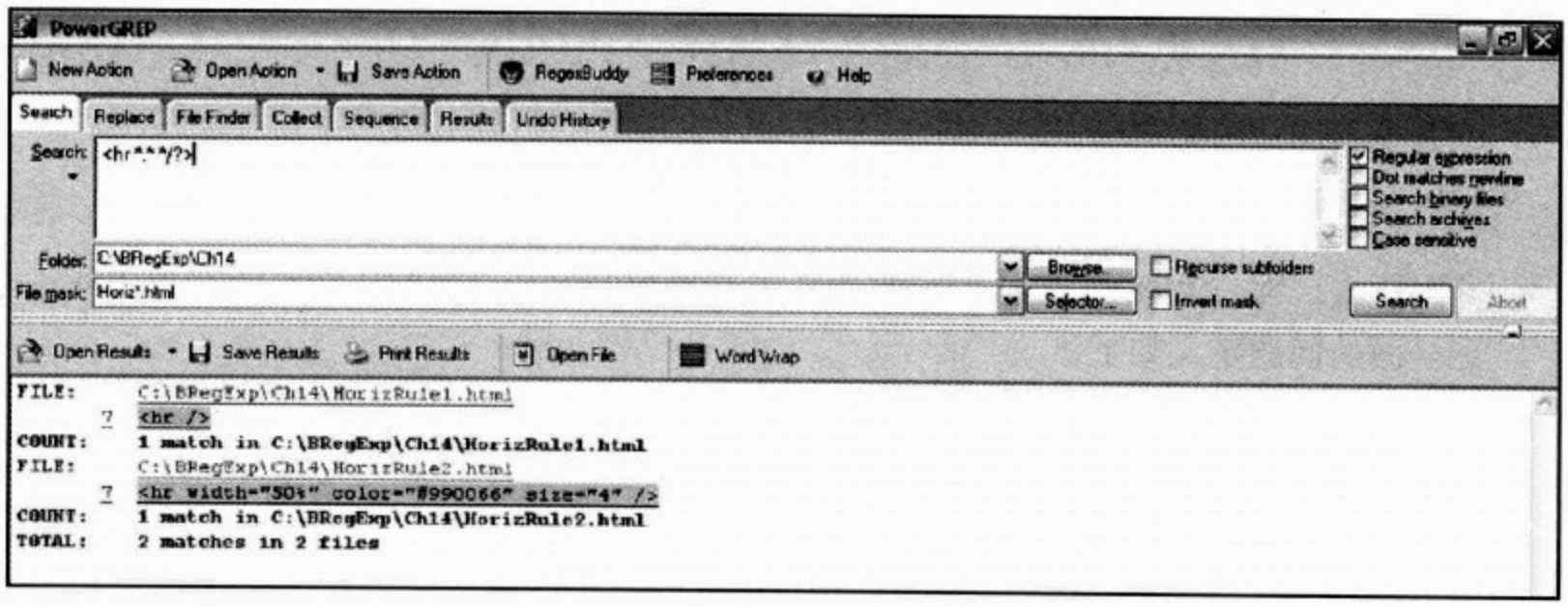


图 14-20

工作原理

模式 <hr *.* */?> 匹配一个 < 字符后跟一个字符序列 hr(任意大小写)，后跟可选的空白符，后跟零个或多个字符，后跟一个可选的正斜杠，再后跟一个 > 字符。

如果只是想查找所有结构正确的 hr 元字符，那么这个模式的灵敏度几乎能够达到 100%。但是，如果有的 hr 元素像下面这样被拆开写在了几行中：

```
<hr
width="50%"
color="#990066"
size="4" />
```

那么，需把模式修改成如下形式才能继续有效：

```
<hr\s*.*\s*/?>
```

模式中的 \s 元字符可以确保制表符或换行符也会被匹配。

但这个模式也会匹配某些格式不正确的字符序列，并将其当作 hr 元素。比如：

```
<hr width=="50%" />
```

其中包含两个连续的 =，这是不规范的。但使用模式 .* 会匹配各种各样不合法的字符序列。虽然这降低了模式的特殊性，但它却能够确保匹配所有 hr 元素，即使该元素包含轻微的语法错误。

如果要查找的文件中包含的是 HTML 或 XHTML 标记，这种特殊性的损失不会造成什么大问题。

14.3.2 匹配时间的例子

本例着眼于匹配构成一天中时间的数据。相应的初次问题定义可能会表达如下：

匹配一天中的任何时间，无论是 12 小时制还是 24 小时制的时间。

为更好地定义问题，就要理解每种时间的表示法以及相应的书写格式。

采用 12 小时制表示法的时间值如下：

```
9:31 am
```

或

```
09:31am
```

或

```
09:31 pm
```

或

```
09:31pm
```

在 12 小时制下，第一个可选的数字在 0 和 1 之间。如果第一个数字是 0，那么第二个数字则一定是 0～9 中的一个数字；但如果第一个数字是 1，则第二个数字则只能是 0、1 或 2。

下面的模式会匹配小时为 09 以内的时间：

```
[0]?[0-9]
```

10～12 的小时数则可以用下面的模式来匹配：

```
1[0-2]
```

这样，位于时间冒号之前的部分，就可以用下面模式来表示：

```
([0]?[0-9]|1[0-2])
```

如果基于不合法的“时间”——如 18:88pm——来测试这个模式，它也会匹配。但是，当在模式后面加上一个冒号时，这个问题就会消失。

匹配 12 小时制下的其余时间部件的模式比较直接明了：

```
:[0-5][0-9] ?[ap]m
```

把这两个时间部件对应的模式组合到一起，就有了匹配 12 小时制下时间表示法的模式：

```
\b([0]?[0-9]|1[0-2]):[0-5][0-9] ?[ap]m
```

对于 24 小时制下的时间可以使用下面的模式来匹配：

```
([01][0-9]|2[0-4]):[0-5][0-9]
```

再将这两个模式通过交替选择组合起来，得到如下模式：

```
(\b([0]?[0-9]|1[0-2]):[0-5][0-9] ?[ap]m|([01][0-9]|2[0-4]):[0-5][0-9])
```

测试文件 Time1.txt 中包含一些 12 小时制的时间值：

```
08:22 pm
08:37 am
19:88 am
12:00 am
11:39pm
7:28 am
8:19 am
```

测试时间 Time2.txt 中包含一些 24 小时制的时间值：

```
06:31
19:15
18:12
23:59
00:03
19:54
03:00
10:49
```

试一试：匹配时间值

(1) 打开 PowerGREP，并确保选中了 Regular expression 复选框。首先，匹配的是 Time1.txt 中的 12 小时制时间值。

(2) 在 Search 文本区中输入模式 \b([0]?[0-9]|1[0-2]):[0-5][0-9] ?[ap]m。

(3) 在 Folder 文本框中输入 C:\BRegExp\Ch14 或者根据保存下载文件的位置调整此处的路径。

(4) 在 File mask 文本框中输入 Time1.txt，单击 Search 按钮，并观察结果，如图 14-21 所示。测试文件中所有合法时间值都被匹配了，而不合法的“时间值”——19:88 没有匹配。

接下来，匹配 Time2.txt 中的 24 小时制时间值。

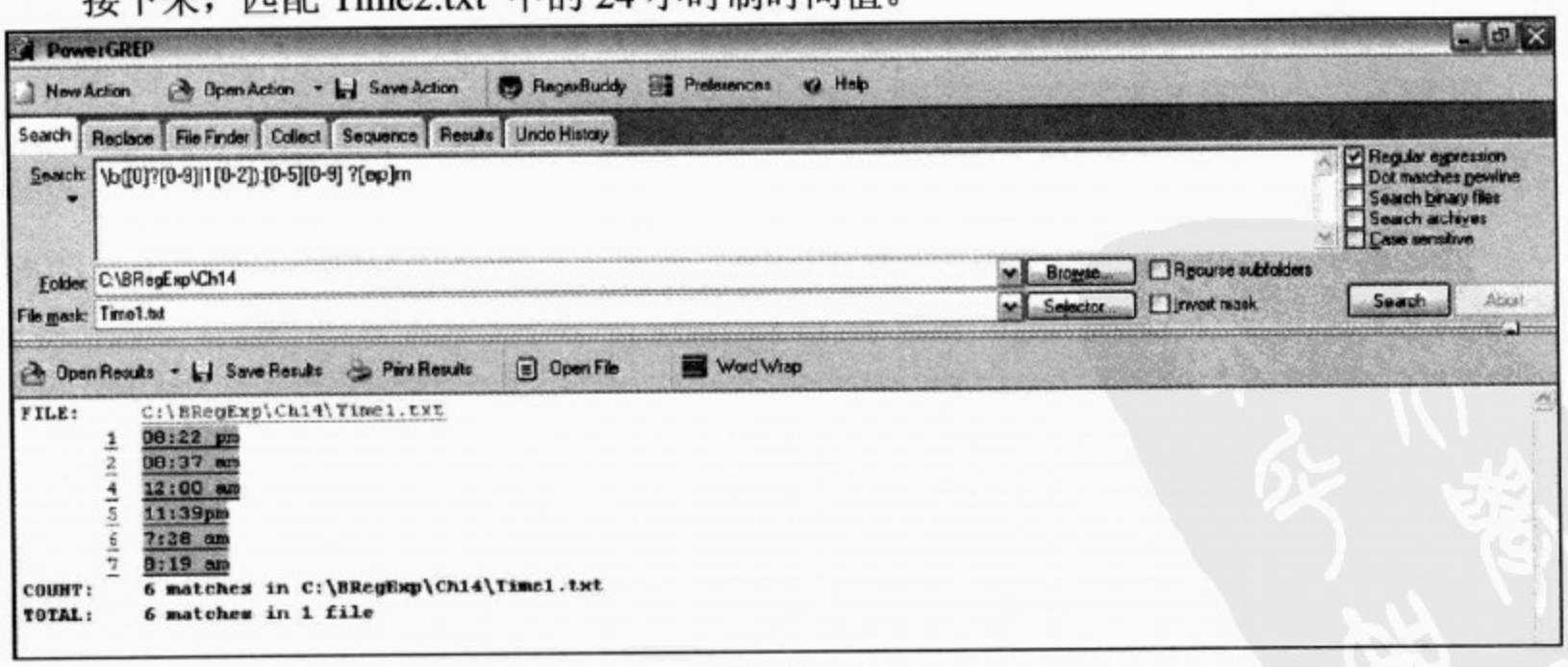


图 14-21

(5) 将 Search 文本区中的模式修改为 ([01][0-9]|2[0-4]):[0-5][0-9]。

(6) 将文件掩码(file mask)修改为 Time2.txt。单击 Search 按钮，并观察结果，如图 14-22 所示。测试文件中的所有 24 小时制时间值都匹配了。

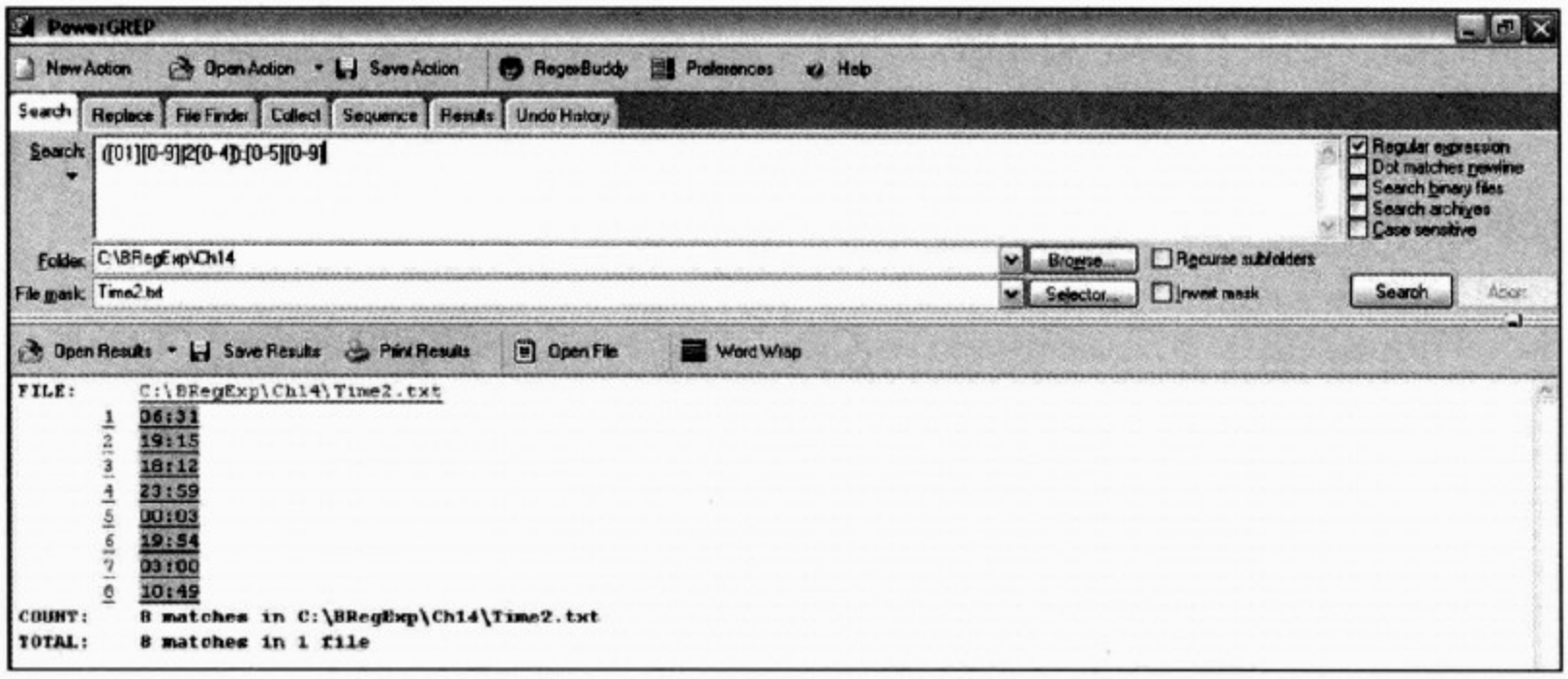


图　14-22

最后，匹配 Time1.txt 和 Time2.txt 这两个测试文件中的时间格式。

(7) 将 Search 文本区的模式修改为 (\b([0]?[0-9]|1[0-2]):[0-5][0-9]?[ap]m|([01][0-9]|2[0-4]):[0-5][0-9])。

(8) 将文件掩码修改为 Time*.txt。单击 Search 按钮，并观察结果。如图 14-23 所示，测试文件中的所有 12 小时制和 24 小时制的时间值都被匹配了。

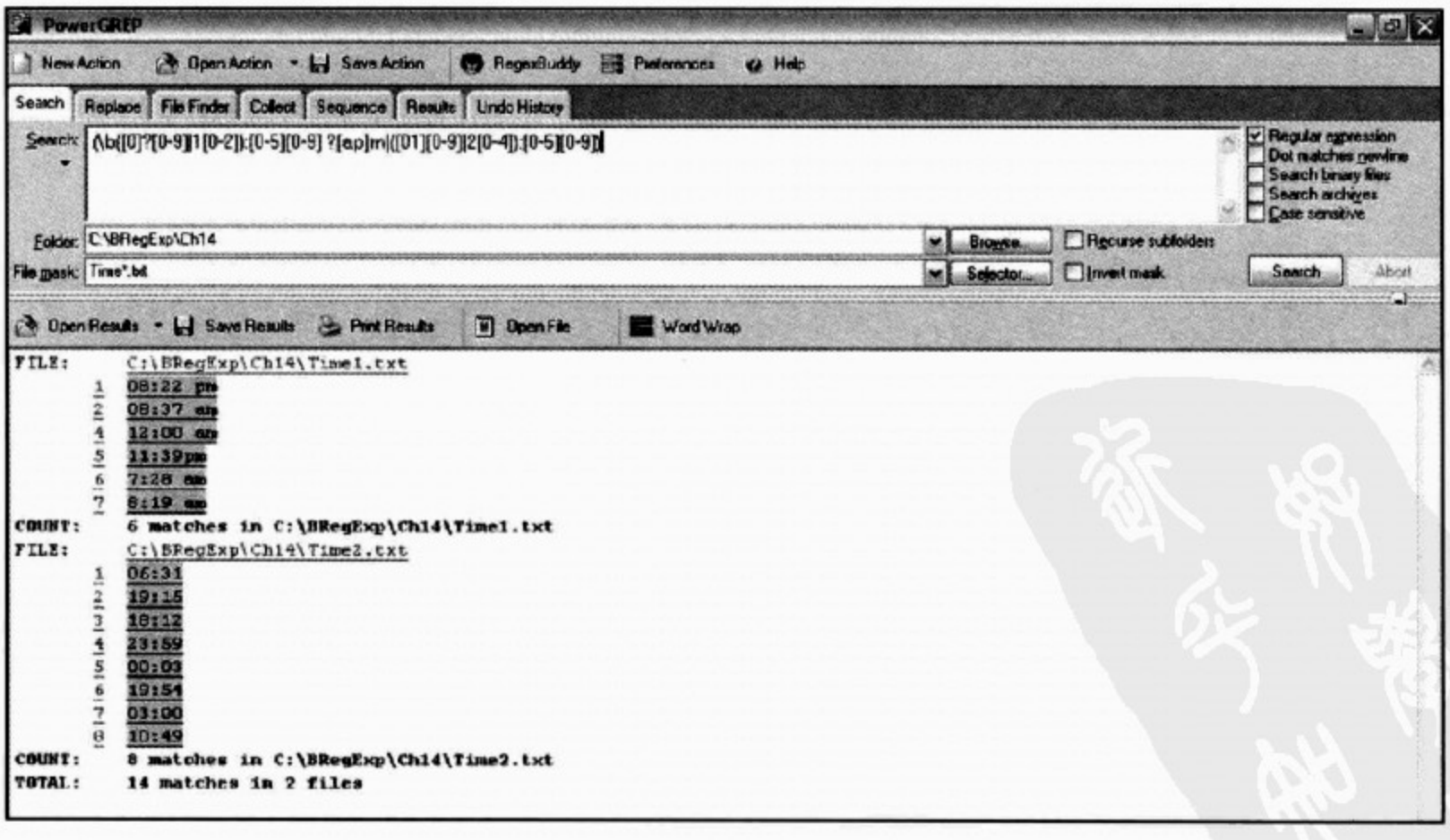


图　14-23

14.4 练习

1. 在本章的第一个例子中，虽然测试文件 Regex.txt 中包含 regexp，而且 regexp 也是测试模式中的一个选项，但 regexp 却没有被匹配。请给出一个改进后的模式以匹配字符序列 regexp 的所有实例。

2. 请创建一个匹配美元金额的模式，假设每个金额数值的小数点前面和后面都只有两位数，例如：$88.23。

第15章

Microsoft Excel 中的通配符

Microsoft Office Excel 是 Microsoft Office 套件中最成功的应用程序之一。很多保存在 Excel 工作表中的数据都是可以搜索的文本。Excel 也提供了基于工作表或工作簿中的公式、值或批注进行搜索的工具。

Excel 中并没有提供对正则表达式完整的支持。与 Microsoft Word 类似，Excel 也是通过通配符对正则表达式提供有限的支持。Excel 中的通配符功能明显比 Word 更有限，但在本章中也会看到，当 Excel 中的通配符功能与其他 Excel 工具结合使用时，同样也可以提高工作效率。

在本章中将学习以下内容：

- 在 Excel 的 Find and Replace 对话框中使用通配符的界面
- Excel 支持的通配符
- 如何在搜索中使用通配符
- 如何在记录单中使用通配符
- 如何在筛选中使用通配符

本章所介绍的通配符功能是基于 Microsoft Office Excel 2003 进行测试的。

15.1 Excel 的查找界面

Excel 中使用通配符的界面与 Microsoft Word 中的界面类似。如图 15-1 所示，Find and Replace 对话框是整个过程的核心。

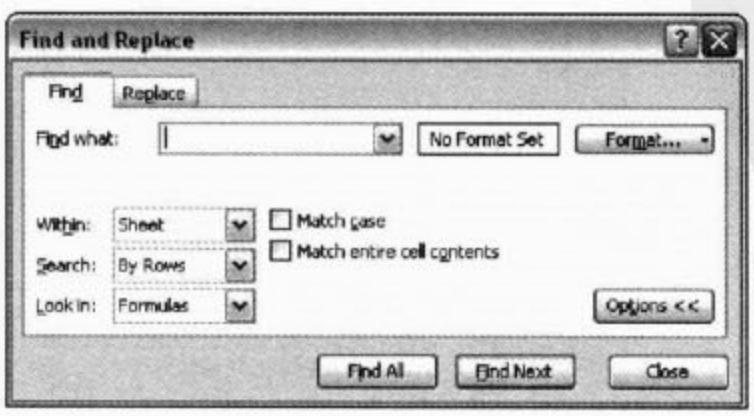
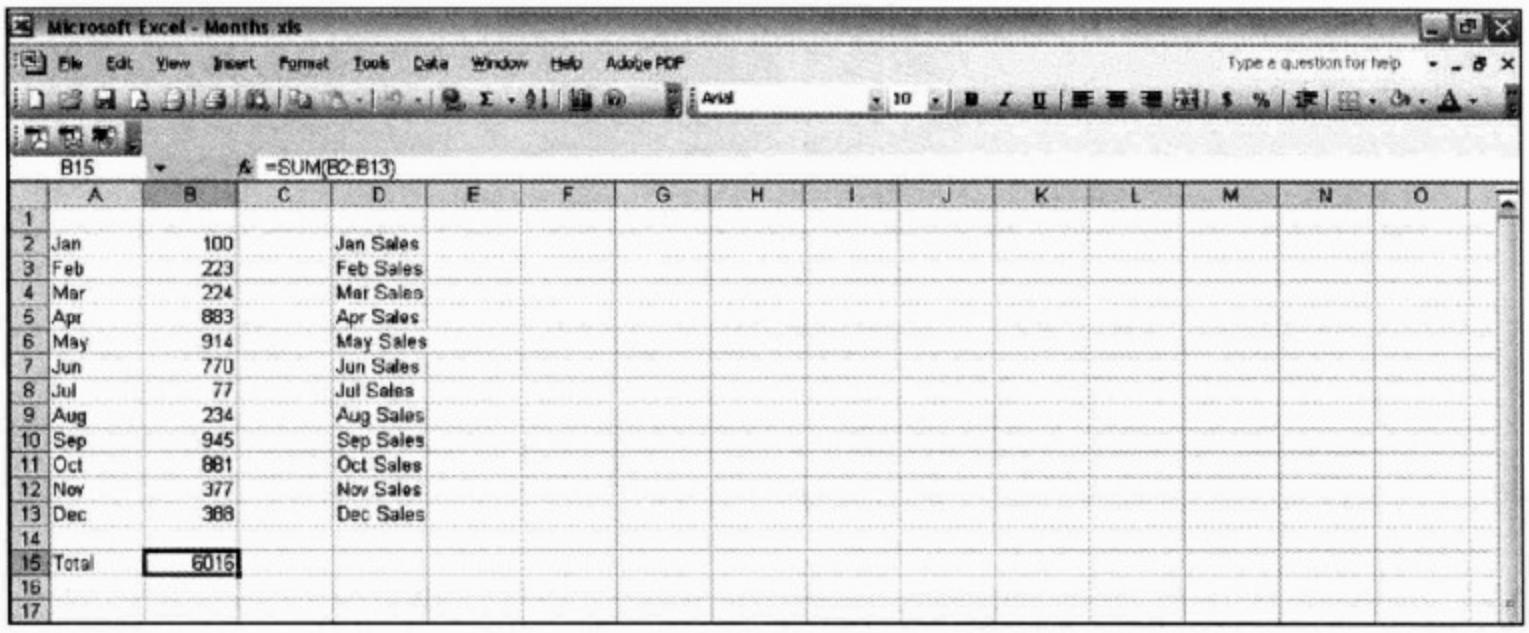


图 15-1

在介绍如何使用 Find and Replace 对话框时，将使用电子数据表 Months.xls，如图 15-2 所示。注意，在这个电子数据表的单元格中包含着一些简单的文本值，而 B15 单元格中则包含着这些值的和。



	A	B	C	D
1				
2	Jan	100		Jan Sales
3	Feb	223		Feb Sales
4	Mar	224		Mar Sales
5	Apr	883		Apr Sales
6	May	914		May Sales
7	Jun	770		Jun Sales
8	Jul	77		Jul Sales
9	Aug	234		Aug Sales
10	Sep	945		Sep Sales
11	Oct	881		Oct Sales
12	Nov	377		Nov Sales
13	Dec	388		Dec Sales
14				
15	Total	6016		
16				
17				

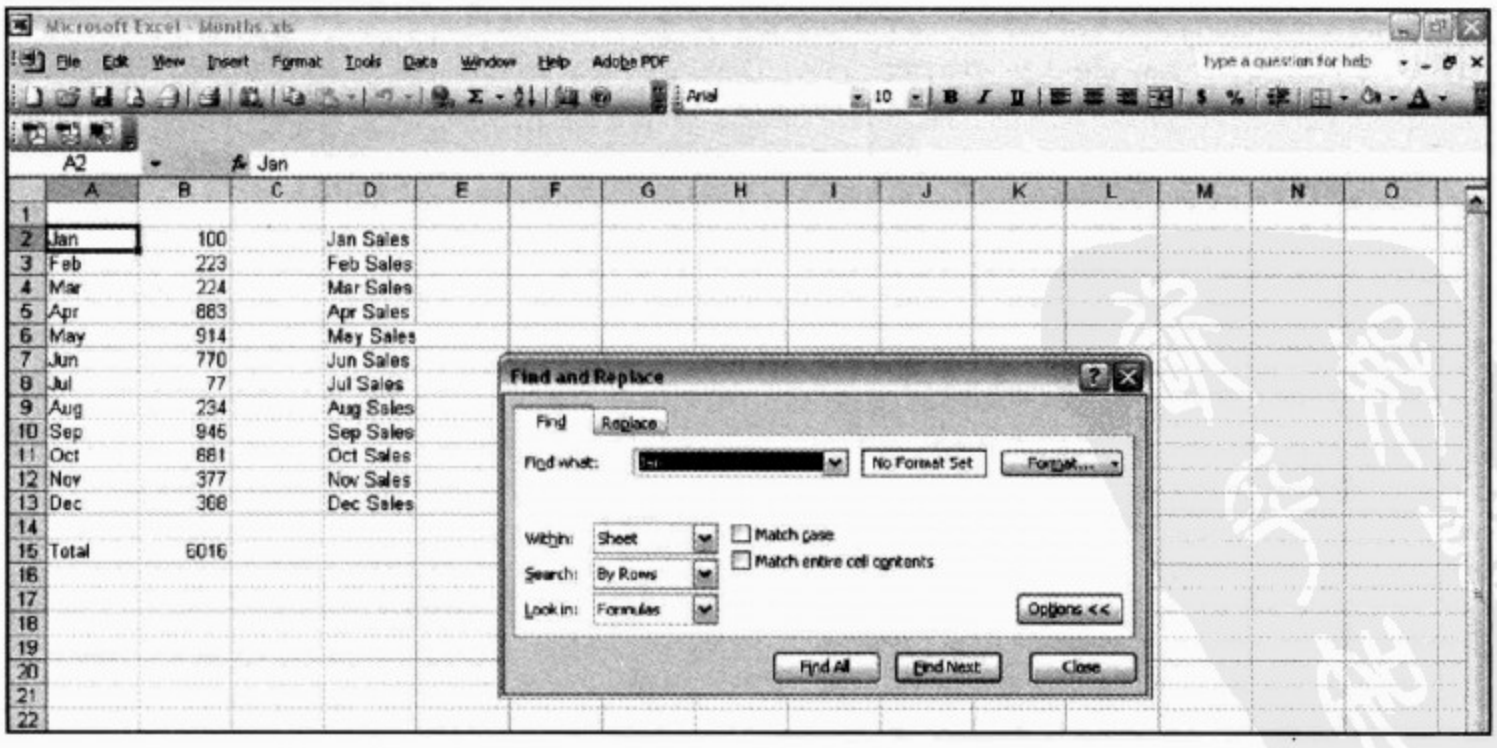
图 15-2

试一试：Find and Replace 界面

(1) 在 Excel 中打开工作簿 Months.xls，并使用 Ctrl+F 快捷键打开 Find and Replace 对话框。

(2) 在 Find what 文本框中，输入字符序列 Jan，单击 Find Next 按钮，并观察结果，结果如图 15-3 所示。其中包含直接量字符序列 Jan 的单元格 A2 突出显示。

(3) Excel 的 Find and Replace 对话框中有一个 Find All 按钮，这个按钮是 Microsoft Word 中所没有的。单击 Find All 按钮，并观察结果，结果如图 15-4 所示。注意，在 Find and Replace 对话框的下方显示了一个匹配项的列表。只有一个匹配项突出显示——在搜索次序中的第一个单元格，此时仍然是单元格 A2。



	A	B	C	D
1				
2	Jan	100		Jan Sales
3	Feb	223		Feb Sales
4	Mar	224		Mar Sales
5	Apr	883		Apr Sales
6	May	914		May Sales
7	Jun	770		Jun Sales
8	Jul	77		Jul Sales
9	Aug	234		Aug Sales
10	Sep	945		Sep Sales
11	Oct	881		Oct Sales
12	Nov	377		Nov Sales
13	Dec	388		Dec Sales
14				
15	Total	6016		
16				
17				
18				
19				
20				
21				
22				

图 15-3

在 Find and Replace 对话框下方显示出列表的原因是某些匹配项可能在当前屏幕中并没有被显示出来。特别是对于大型的电子数据表，其中或许包含多个匹配项，而这些匹配项可能会分散在几个屏幕中。

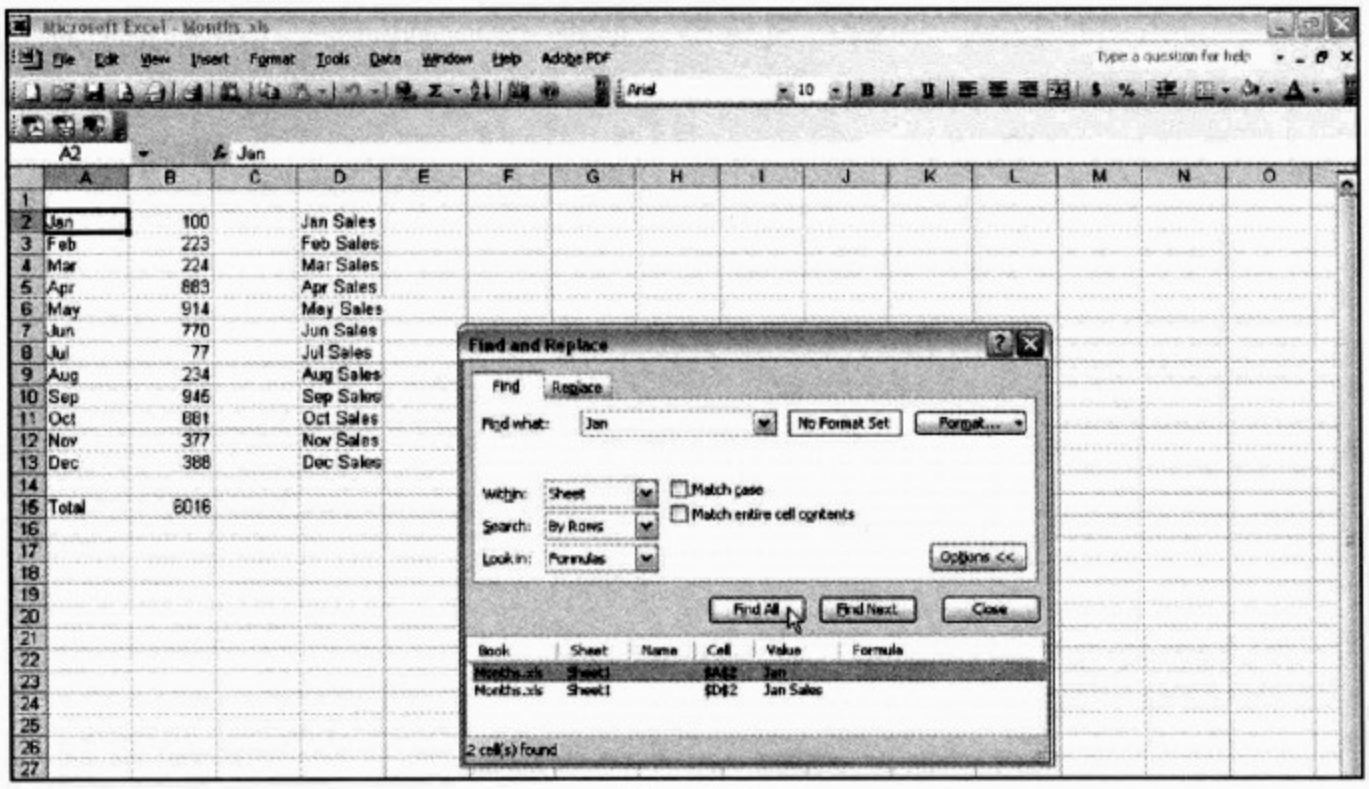


图　15-4

无论多个匹配项是否显示于当前屏幕中，通过 Find and Replace 对话框下方的匹配项列表都可以轻松地导航到你所关注的匹配项中。在本例中，仅有两个匹配项。

(4) 在 Find and Replace 下方的列表中单击下面的匹配项。图 15-5 显示的是这一步之后的屏幕外观。此时单元格 D2 突出显示，因为它也包含着字符序列 Jan。

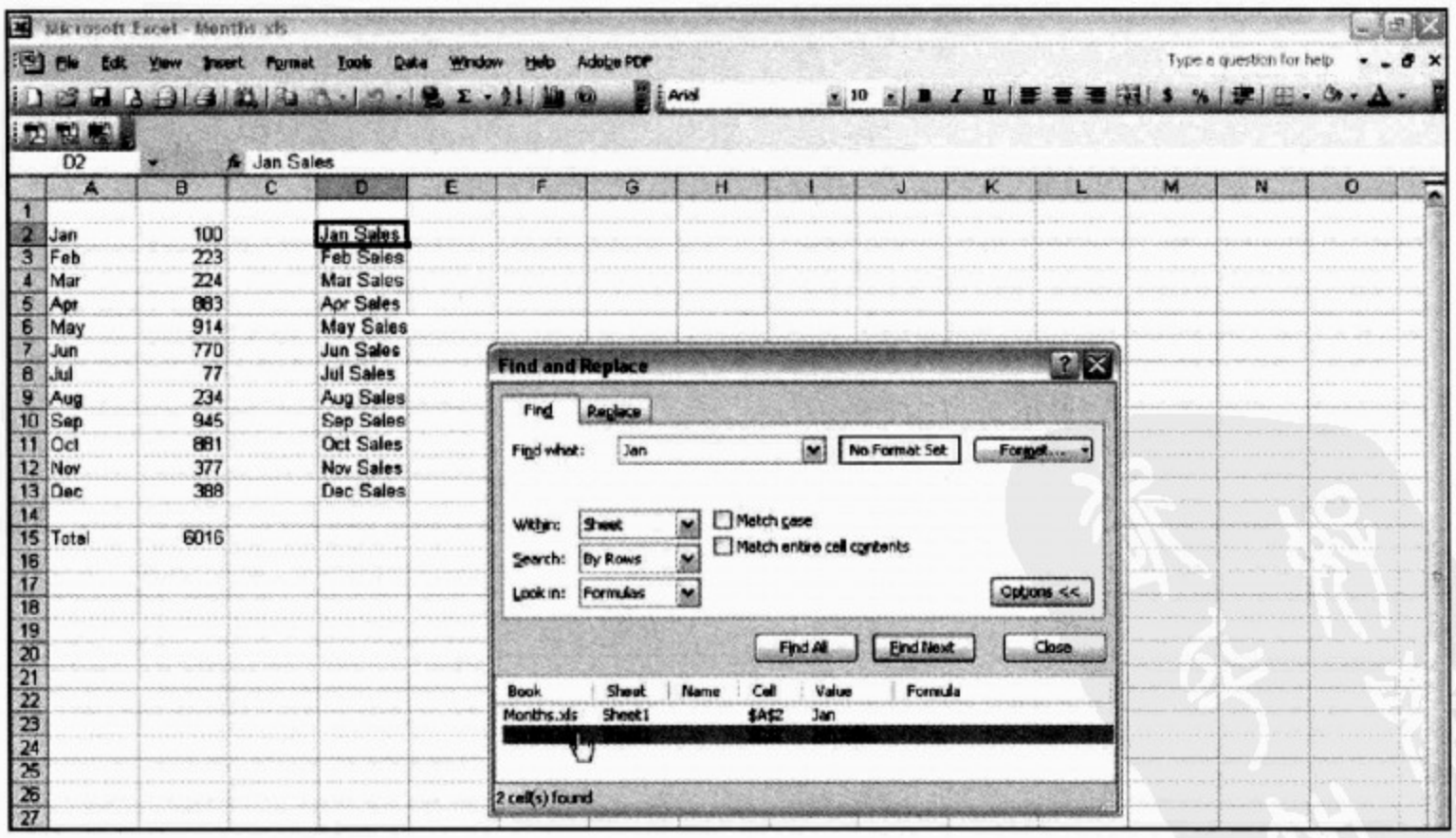


图　15-5

在 Find and Replace 对话框中有一个 Match case 复选框。如果选中它，就会将默认的不区分大小写的匹配转换为区分大小写的匹配。如果选中 Match entire cell contents 复选框，则意味着直接量模式 Jan 只有在 Jan 是一个单元格中的全部内容时才会匹配。

(5) 选中 Match entire cell contents 复选框，单击 Find All 按钮，并观察结果，结果如图 15-6 所示。注意，现在只列出一个匹配项。由于 Jan 只是单元格 D2 中的一部分内容，所以这次单元格 D2 没有匹配。

在了解 Within、Search 和 Look in 这几个下拉列表的作用之前，先来看一看由 Excel 通配符提供的类似正则表达式功能的有限范围。

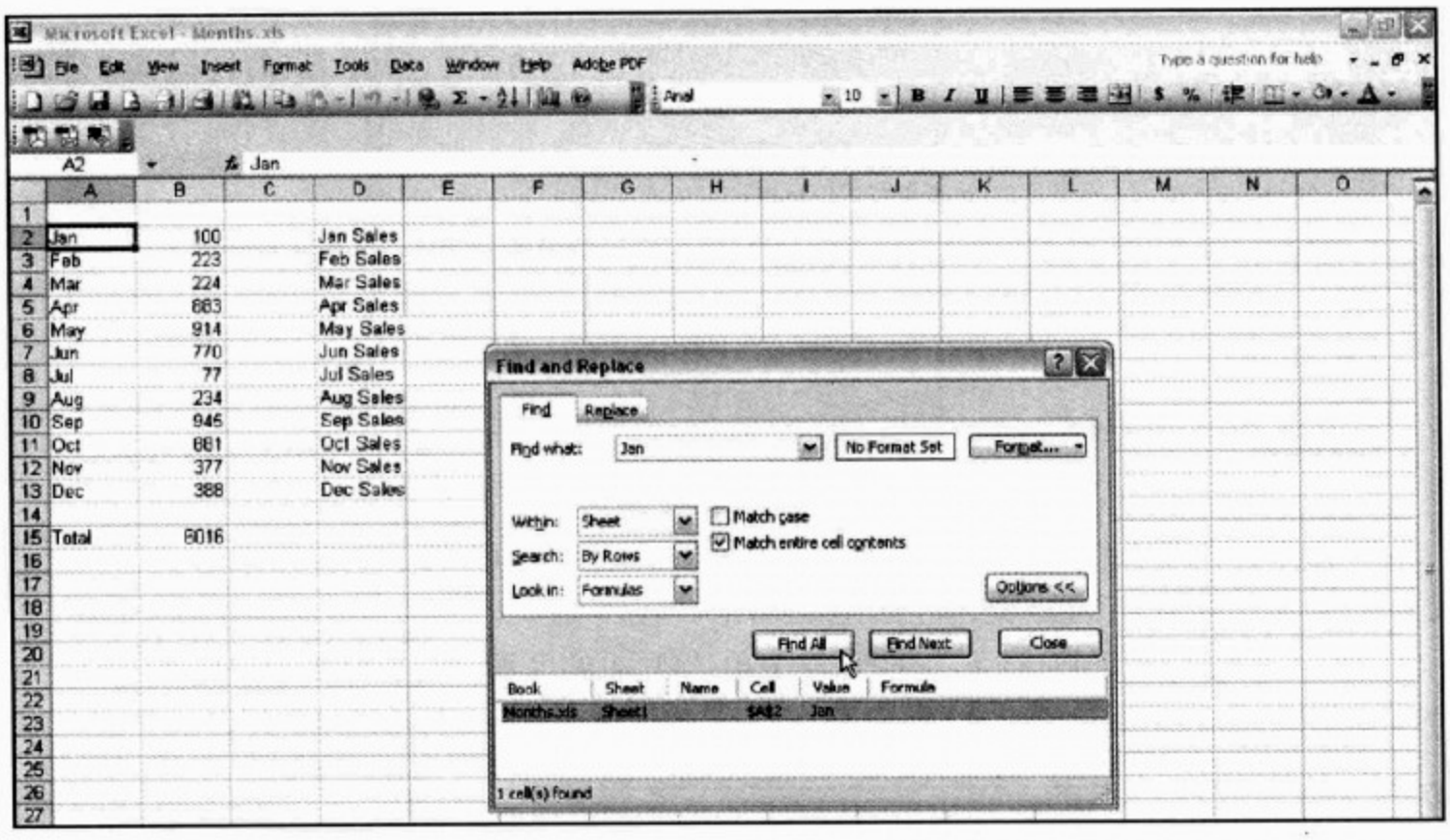


图 15-6

15.2 Excel 支持的通配符

Excel 支持的通配符是本书所介绍的工具中最少的。它仅支持三个元字符，这三个元字符及其含义如表 15-1 所示。

表 15-1 Excel 支持的元字符

元 字 符	含 义
?	匹配任何单独的字符
*	匹配任何零个或多个字符的序列
~	转义字符

试一试：Excel 中的通配符

(1) 在 Excel 中打开 Months.xls。确保 Match entire cell contents 复选框没有被选中。

(2) 在 Find 文本框中输入模式 J?n。这个模式将会匹配字符序列，如 Jan 或 Jun，它们在 Months.xls 中同样都有两个实例。

(3) 单击 Find All 按钮，并观察显示于 Find and Replace 下方的结果，如图 15-7 所示。

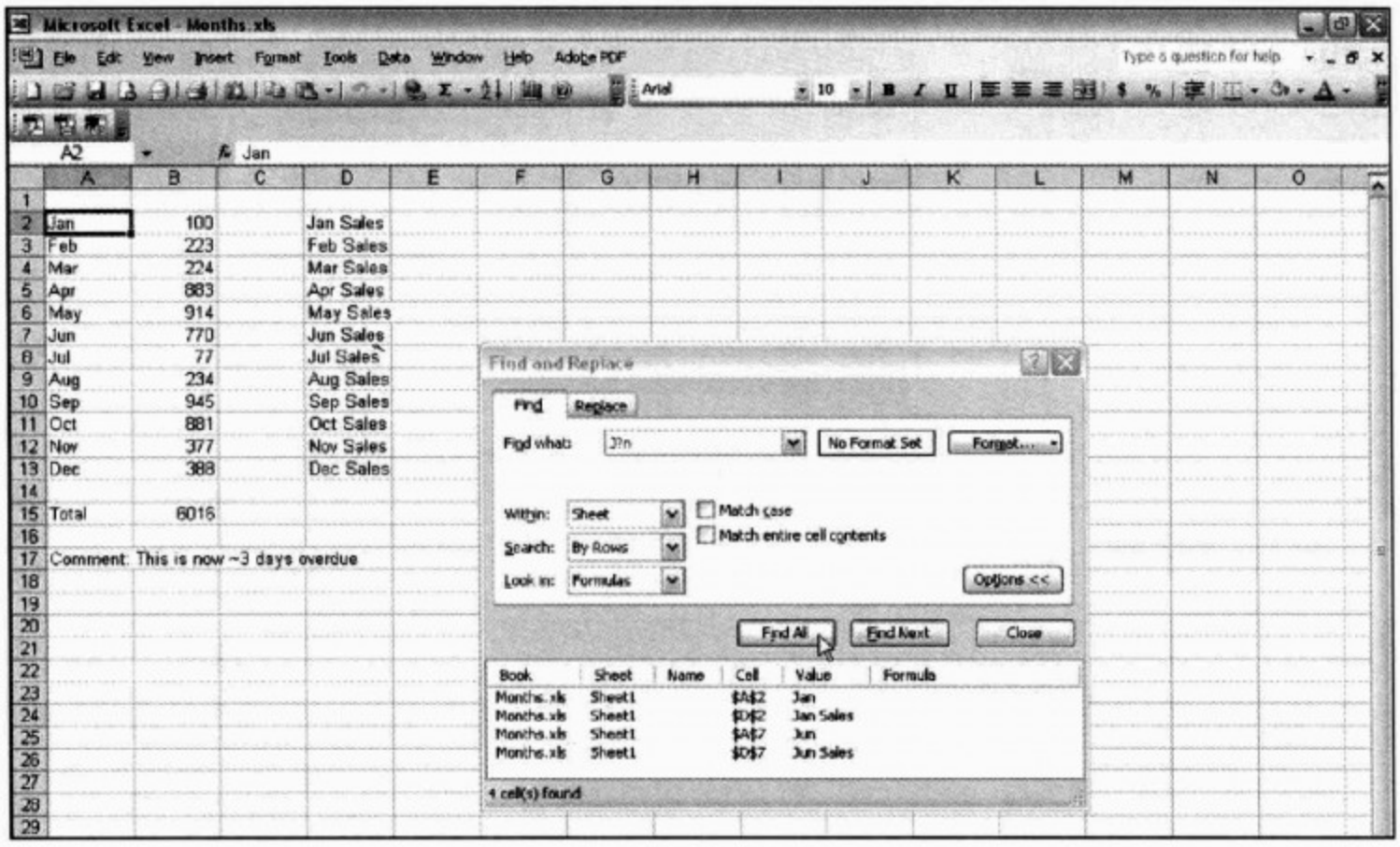


图　15-7

单击 Find and Replace 下方匹配列表中的任何一个匹配项，都可以定位到相应的值，而不用管它在当前屏幕中是否可见。

(4) 单击第二行，这个匹配项的值是 Jan Sales。图 15-8 显示的是被选择的单元格被突出显示。

(5) 单击 Find and Replace 对话框下方匹配列表中的其他匹配项以确定能够定位到相应的单元格。

(6) 在 Find what 文本框中将模式修改为 A*，单击 Find All 按钮，并观察如图 15-9 所示的结果。其中所有字母字符 A 或 a 都被匹配。

即便在 Months.xls 中数据量很少的情况下，与模式 A* 匹配的内容数量仍然多得让人无法接受。在 Excel 中没有位置元字符的概念，比如说表示行开始位置的元字符 ^ 等。因此，无法使用这种技术来限定匹配的范围。在这个例子中，可以通过指定进行区分大小写的匹配来排除由于单词 sales 导致的多余匹配项。在 Excel 中，默认进行的是不区分大小写的匹配。

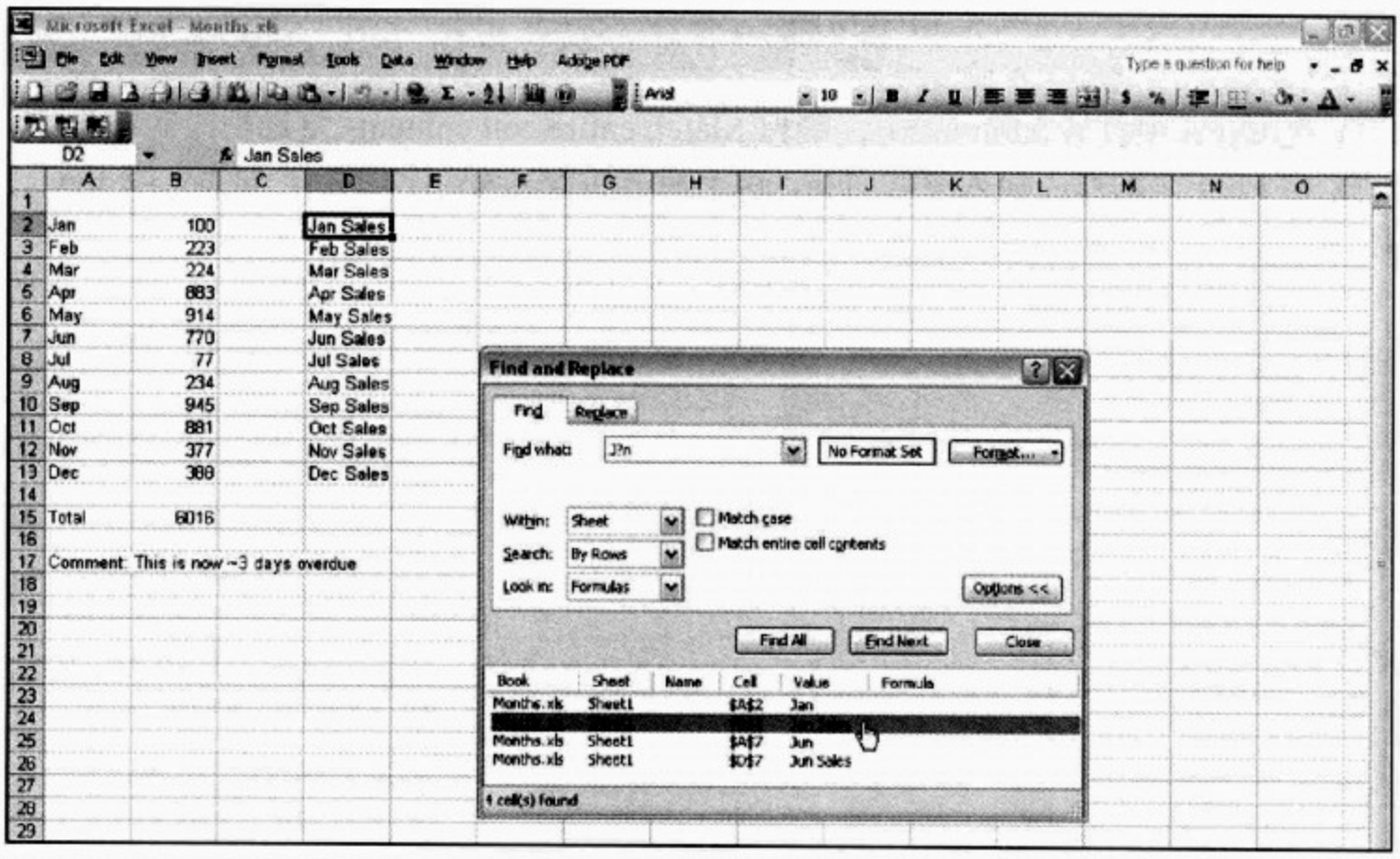


图 15-8

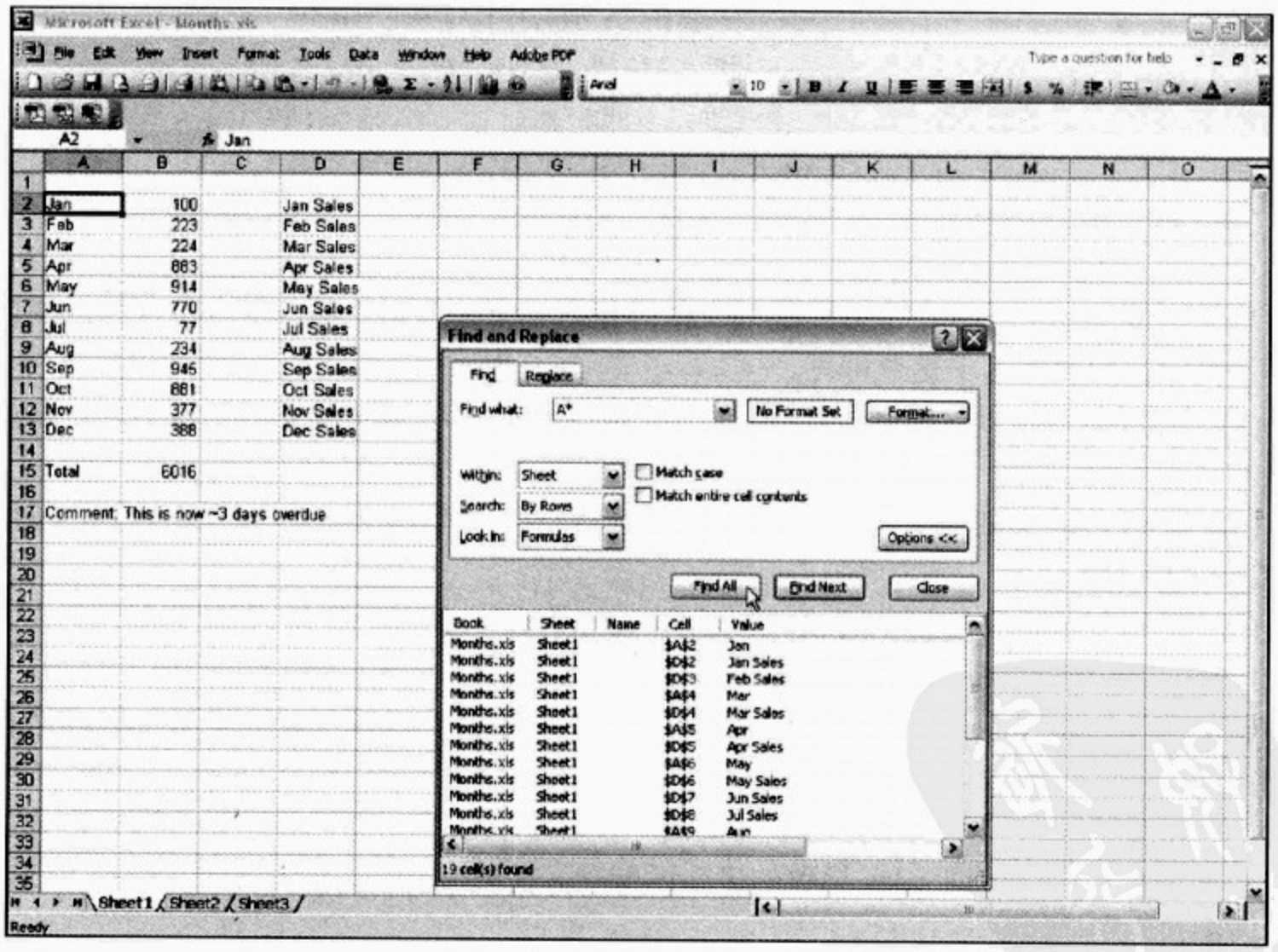


图 15-9

(7) 选中 Match case 复选框，单击 Find All 按钮，并观察如图 15-10 所示的结果。此时只会匹配包含一个大写 A 的单元格。

本例演示了在 Excel 中使用有限的通配符功能所存在的一个普遍问题，即有时候特殊性会非常低。

不要关闭 Excel，因为下一个例子将会从这里继续。

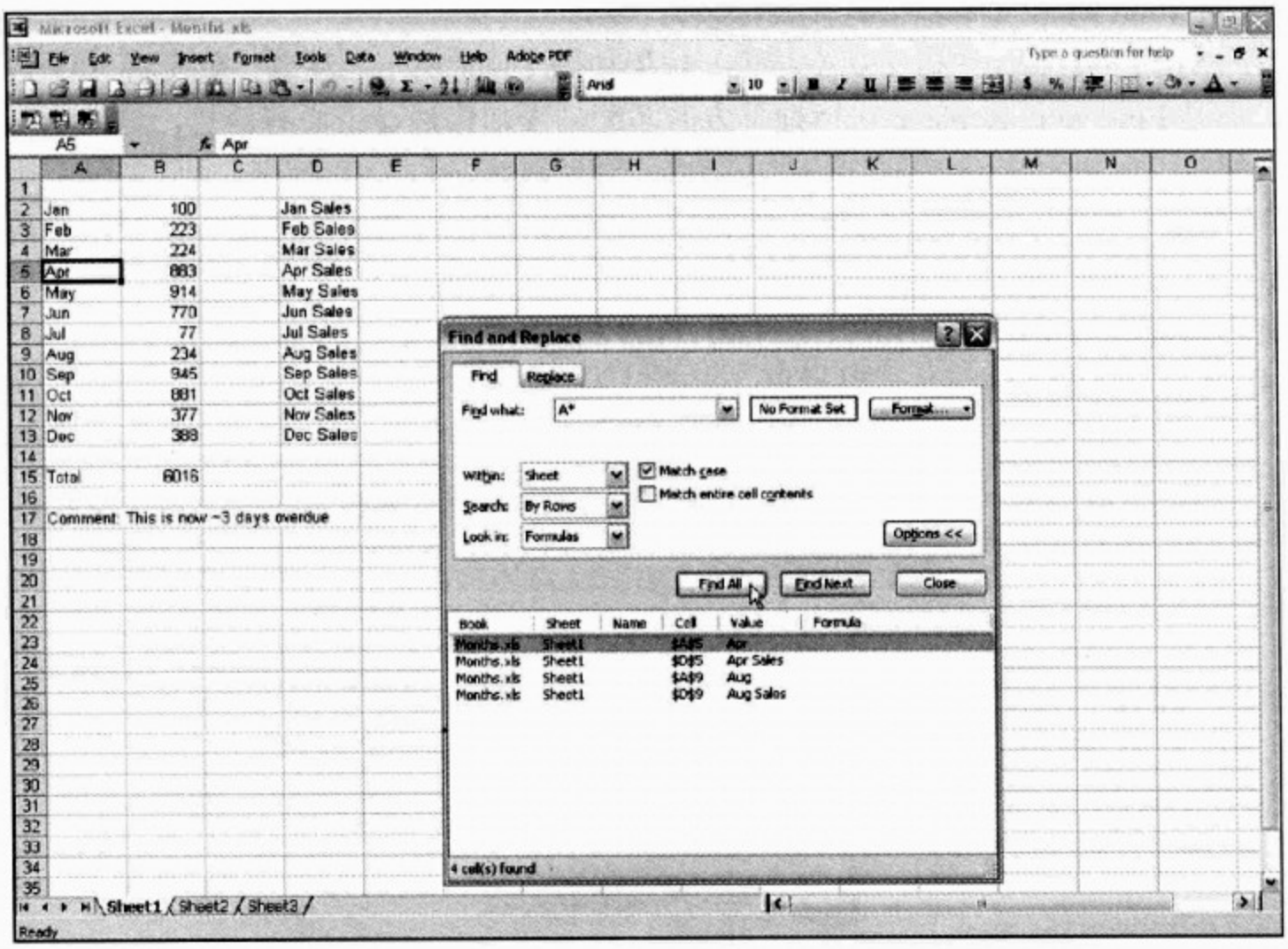


图　15-10

工作原理

模式 J?n 匹配任何以 J 开头、后跟任意字符再跟一个 n 的字符序列。在这个例子中，字符序列 Jan 和 Jun 匹配。

在未选中 Match Case 复选框的情况下，模式 A* 会匹配任何包含 A 或 a 并后跟任何数量字符的字符序列。此时，不想要匹配的最大来源就是单词 sales，该单词出现了 12 次。之所以会匹配它们是因为 sales 中包含一个 a，而且 a 也不是其最后一个字母。所以，在不区分大小写的情况下就会成功匹配。

对于大型的工作表而言，如果使用的模式就像 A* 一样不具体，那么匹配的数量会多到难以想象的程度。另一种限定匹配范围的技术就是在模式中添加更多的字符——例如，Ap* 会匹配 April 但不会匹配 August。还有一种选择是使用多个 ? 元字符，每个 ? 都会匹配一个单独字符，所以可以指定想要匹配的数量。比如说，模式 Ap??? 会匹配 April。

转义通配符

在某些情况下，可能想要匹配 * 或 ? 直接量字符，而不是将它们作为元字符使用。在 Excel 中需要使用 ~ 字符(波形符号)来转义这些元字符。

因此，要匹配直接量 ? 需使用 ~?，而要匹配直接量 * 就要使用模式 ~*。若要匹配

直接量 ~，需要使用模式 ~~。

试一试：转义通配符

(1) 在 Find what 文本框中将模式修改为 ~~。

(2) 单击 Find Next 按钮并观察结果，结果如图 15-11 所示。注意，单元格 A17 被突出显示。但是 Find and Replace 对话框下方的匹配列表却仍然显示原有信息，这在无形中会导致误解。

(3) 在上一步中使用 Find Next 按钮导致的结果滞留问题可以通过在新模式下再次单击 Find All 按钮来解决。如果存在多个匹配项，则可以使用 Find and Replace 对话框下方的导航列表定位到想要的单元格中。单击 Find All 按钮。

(4) 观察 Find and Replace 对话框，以确认滞留的结果已经被最新的匹配结果所取代。

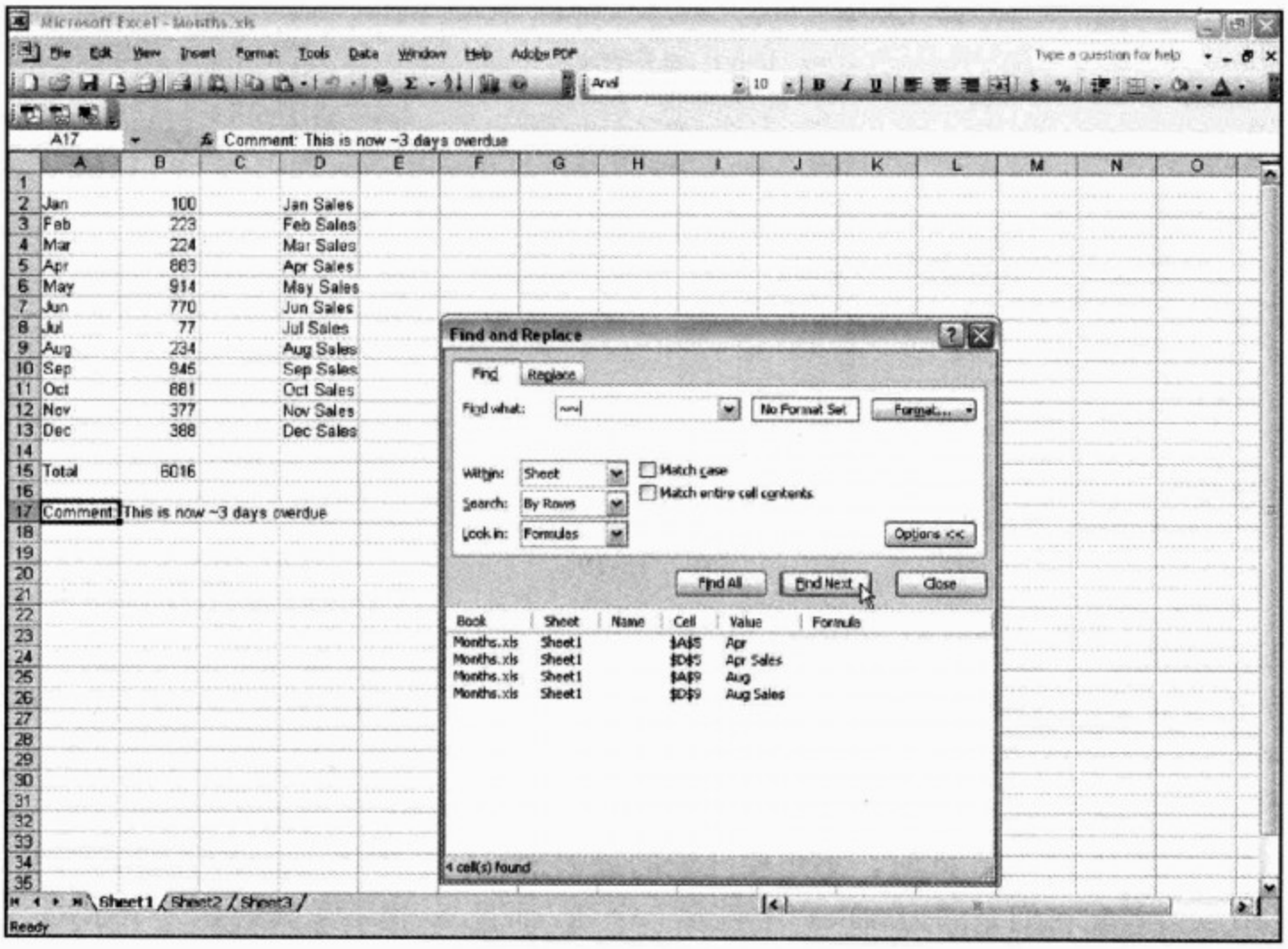


图 15-11

15.3 在记录单中使用通配符

Excel 中的表格数据可以保存在一个列表中。示例文件 Names.xls 中包含了一个保存着一些姓(LastName)、名(FirstName)及出生日期(DateOfBirth)的简单列表。

试一试：在记录单中使用通配符

(1) 在 Excel 中打开 Names.xls。

(2) 在 Data 菜单中选择 Form。观察显示的列表数据。此时打开标题为 Sheet1 的记录单，注意到记录单的中间有一个 Slider 控件。

(3) 单击 Criteria 按钮。图 15-12 显示的是这一步之后的屏幕外观。此时 Criteria 按钮被 Form 按钮所替换。而现在可以在记录单中输入匹配数据的条件了。

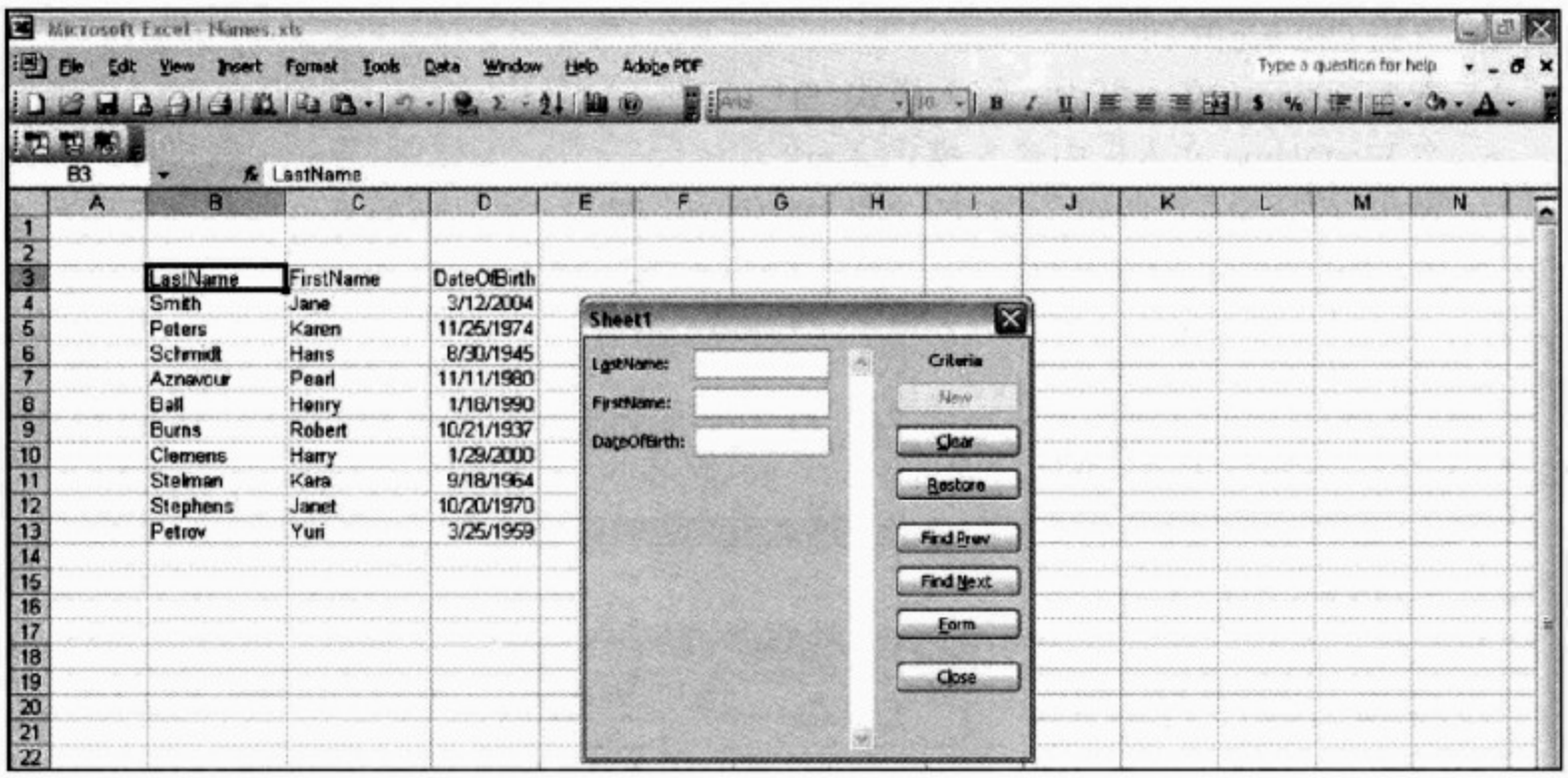


图　15-12

(4) 要匹配以字符序列 St 开头的姓，可以在记录单中的 LastName 文本框中输入模式 St*。现在就在 LastName 文本框中输入模式 St*，然后单击 Find Next 按钮并观察结果，结果如图 15-13 所示。注意，记录单中间的 Slider 控件处于接近底部的位置。这是因为第一次匹配项出现在 10 行中的第 8 行。同时，请注意随着数据显示出来，Criteria 按钮也再次出现了。

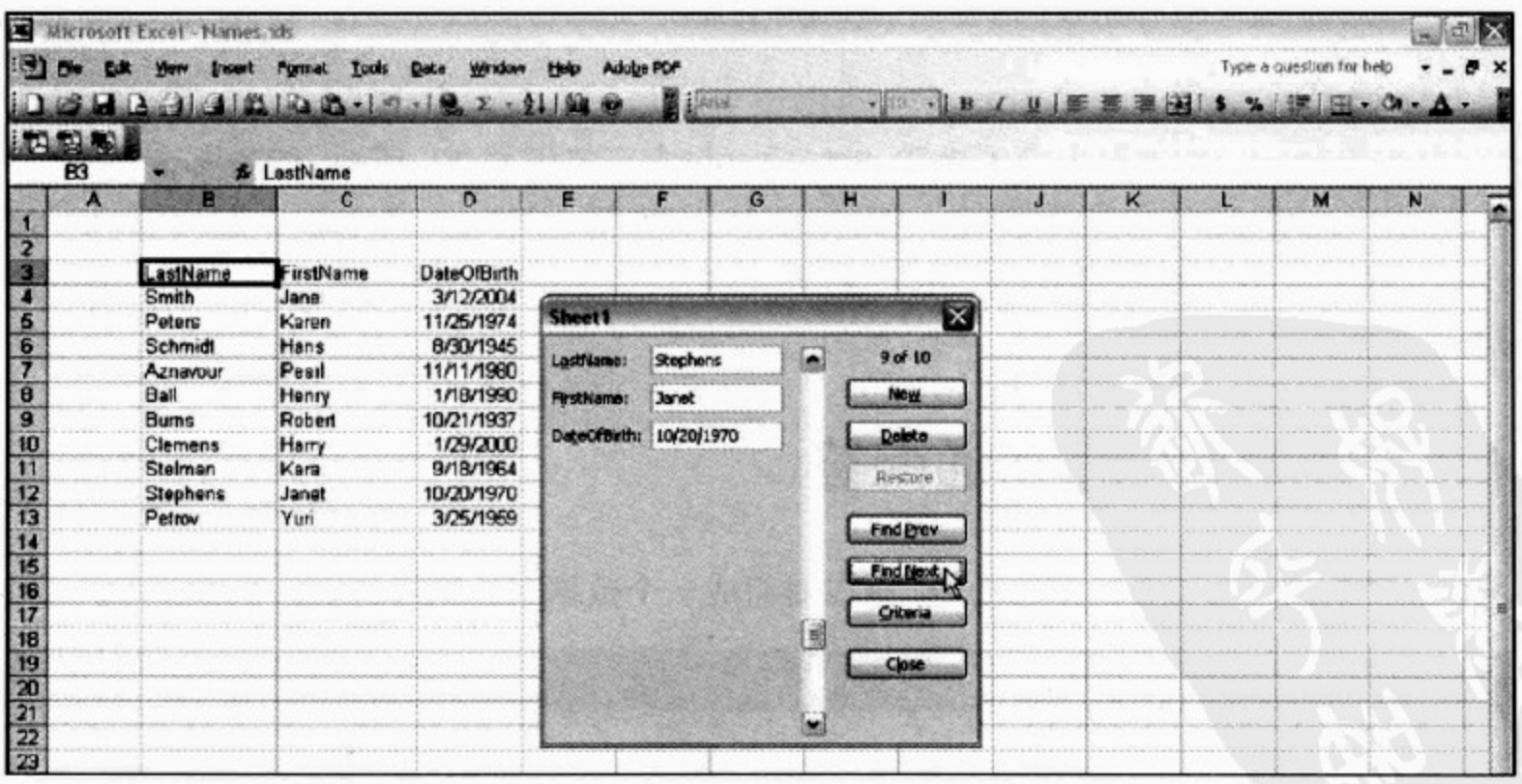


图　15-13

(5) 再次单击 Find Next 按钮，并确认第 9 行被匹配。

通过 Find Prev 按钮可以重新定位到第 8 行。

(6) 可以在文本框中以组合的形式使用通配符。比如说，可以通过下列步骤搜索姓中包含 St 而名字中包含 Kar 的人：

a) 单击 Criteria 按钮。

b) 在 LastName 文本框中，输入模式 St*。

c) 在 FirstName 文本框中输入模式 Kar?。

d) 单击 Find Next 按钮，并观察记录单的外观。图 15-14 显示的是这一步之后的外观。在这个有限的数据集合中只找到了一个匹配项——Kara Stelman。

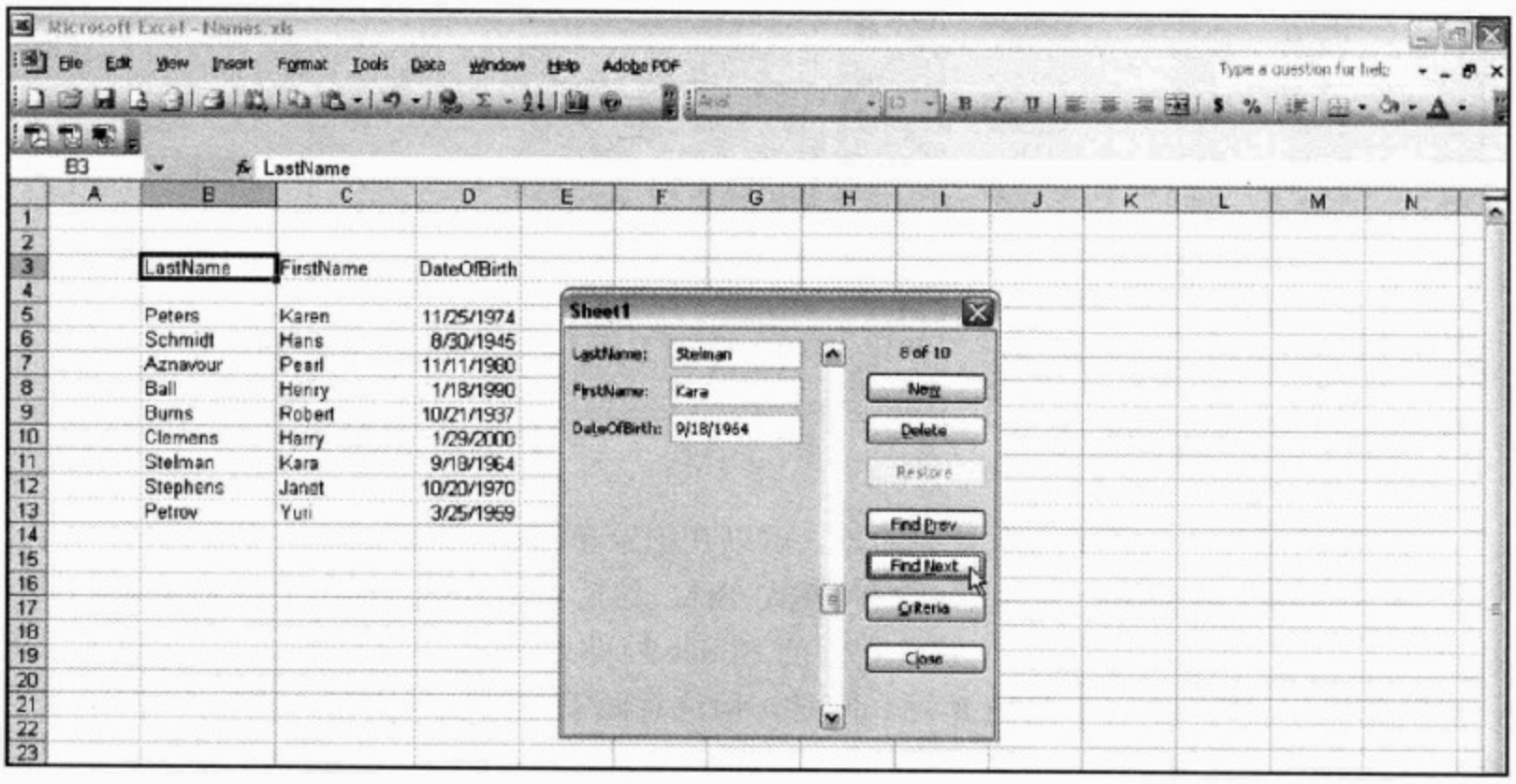


图 15-14

工作原理

在 LastName 文本框中输入模式 St* 后，包含字符序列 St 或 st 的姓都会被匹配。

当在 LastName 文本框中输入模式 St* 而在 FirstName 文本框中输入模式 Kar? 后，则只有名字中包含字符序列 Kar 后跟一个字符，并且姓中包含字符序列 St 后跟任意数量字符的人名才会被匹配。

15.4 在筛选中使用通配符

Excel 中包含着强大的筛选功能，可以对组成一个数据表的表格数据执行各种筛选操作。

试一试：在 Excel 筛选中使用通配符

(1) 在 Excel 中打开 Names.xls。在 Data 菜单中选择 Filter，并选择 AutoFilter。此时，应该会看到如图 15-15 所示的类似结果。如果在进行自动筛选时遇到了问题，可以使用文

件 NamesWithFilter.xls。

注意包含在单元格 B3、C3 和 D3 中的下拉列表。通过它们可以使用自动筛选功能。

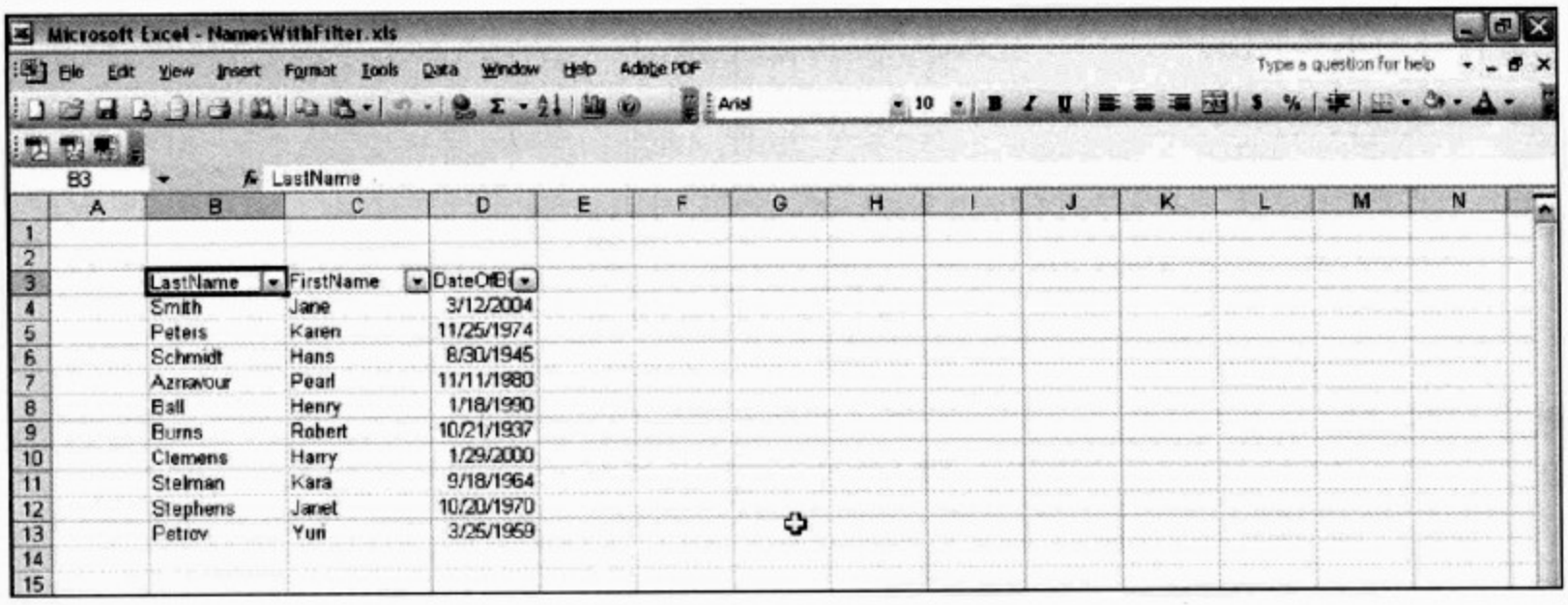


图　15-15

(2) 单击带有 LastName 值的单元格(B3)中的下拉列表，并在列表中选择 Custom 选项。

这一步后 Excel 的界面外观应该如图 15-16 所示。通过 Custom AutoFilter 对话框可以显示选择的行。

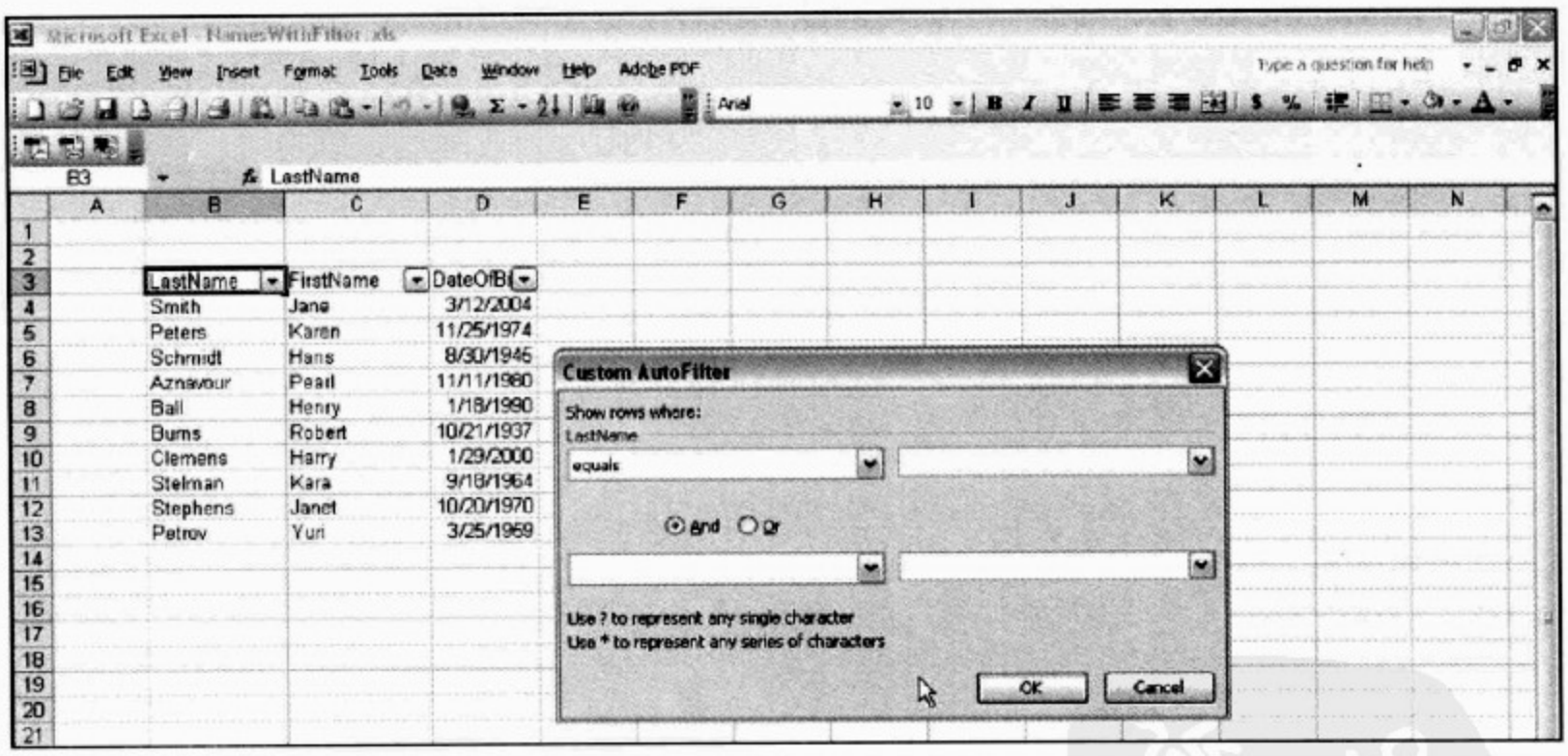


图　15-16

(3) 在左上角的下拉列表中，选择 Begins with 选项(必须向下滚动列表才能选择到)。

(4) 在右上角的文本框中输入模式 St*，单击 OK 按钮，并观察结果。结果如图 15-17 所示。此时，只显示出了 10 行中的两行。而显示的这两行都是姓氏以字符序列 St 开头的数据行。

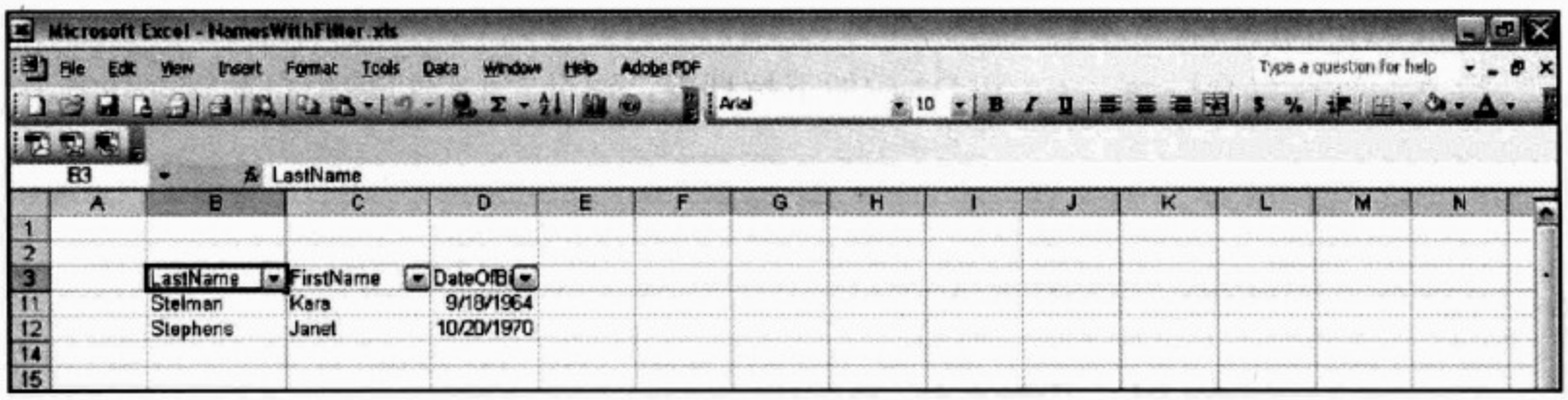


图 15-17

Custom AutoFilter 对话框在某些方面要优于记录单。例如，它可以让用户指定模式匹配单元格中值的开头或结尾。这就提供了类似于 ^ 或 $ 元字符的功能。另外，也可以使用“与 AND”和“或 OR”逻辑来创建两条规则。

然而，Custom AutoFilter 毕竟只能通过模式定义一列数据。而通过记录单中的条件按钮，则可以基于数据表中任何列的组合进行模式定义。

15.5 练习

1. 在 NamesWithFilter.xls 中，怎样才能只显示名字以 Kar 开头的人名？
2. 在 Months.xls 中，描述一下如何匹配只包含字符序列 Jun 或 Jul 的单元格。

第16章

SQL Server 2000 中的正则表达式功能

浩如烟海的商业及其他数据都保存于 Microsoft 的旗舰关系型数据库产品—— SQL Server 之中。大多数使用 SQL Server 的管理员和开发人员都能能对从大型数据库中检索数据时可能出现的问题了如指掌。开发人员或用户对数据了解越多，检索数据的效果也就越好。而通过使用正则表达式功能，能提高从 SQL Server 数据库中检索到目标数据的灵敏度和特殊性。

在 SQL Server 中可以通过 WHERE 子句中的 LIKE 关键字，或者全文索引和搜索来实现类似于正则表达式的功能。

本章将介绍 SQL Server 2000 中支持的正则表达式功能。beta 版(在本书写作时。译者注)的 SQL Server 2005 也支持类似功能。

在本章中将学习以下内容：

- SQL Server 2000 支持哪些元字符
- 如何通过 LIKE 关键字使用被支持的元字符
- 如何使用全文搜索来实现类似正则表达式的功能

本章中提供的例子假设你在本地安装了 SQL Server 2000 的未命名实例。如果你是将 SQL Server 2000 作为一个命名实例来运行，在必要的情况下需修改相应的连接信息。

16.1 支持的元字符

SQL Server 2000 支持有限的四个元字符，其中一些元字符的用法和含义也不标准。这四个元字符均可以在 LIKE 关键字中使用。

表 16-1 列出了 SQL Server 2000 支持的元字符。

表 16-1　SQL Server 支持的元字符及其含义

元 字 符	含　义
%	匹配零个或多个字符。% 不是限定符
_	下划线字符匹配单个字符。而且它也不是限定符
[...]	匹配字符类。可以支持字符类范围
[^...]	取反的字符类，即匹配任何不在字符类中的字符

在使用 LIKE 关键字的情况下，许多正则表达式的特性都不被支持。表 16-2 列出了不被支持的正则表达式特性。

表 16-2　不被支持的正则表达式特性

元字符或功能	说　明
\d	不支持
\w	不支持
反向引用	不支持
?	不支持
*	不支持；% 元字符不是限定符
+	不支持
{n,m}	不支持
向前查找	不支持

16.2　在 LIKE 中使用正则表达式

LIKE 关键字是 SELECT 语句中 WHERE 子句的一部分。它能使 WHERE 子句基于一个正则表达式进行筛选，而不仅仅是使用简单的直接量字符序列。

下面的“试一试”部分展示了使用 SQL Server 2000 所支持的有限元字符的一个例子。

16.2.1　%元字符

%元字符匹配零个或多个字符。它等价于许多标准正则表达式实现中的元序列 .* 。

试一试：使用 % 元字符

(1) 打开 Query Analyzer。在 Windows XP 中选择 Start | All Programs | SQL Server | Query Analyzer。

(2) 在 Connect to SQL Server 对话框中，连接到适当的 SQL Server。 Query Analyzer 打开后的界面如图 16-1 所示。此时的界面与上一次使用 Query Analyzer 时的选项设置有很大关系。

(3) 在第一次查询中，在 pubs 测试数据库中选择姓氏以 B 开头的作者。可以用模式 B%

实现这一点，其中 % 元字符匹配任意数量的字符或字符组合。

在 Query Analyzer 的查找面板中输入下列 Transact-SQL 代码：

```
USE pubs
SELECT au_lname, au_fname FROM dbo.authors
WHERE au_lname LIKE 'B%'
ORDER BY au_lname
```

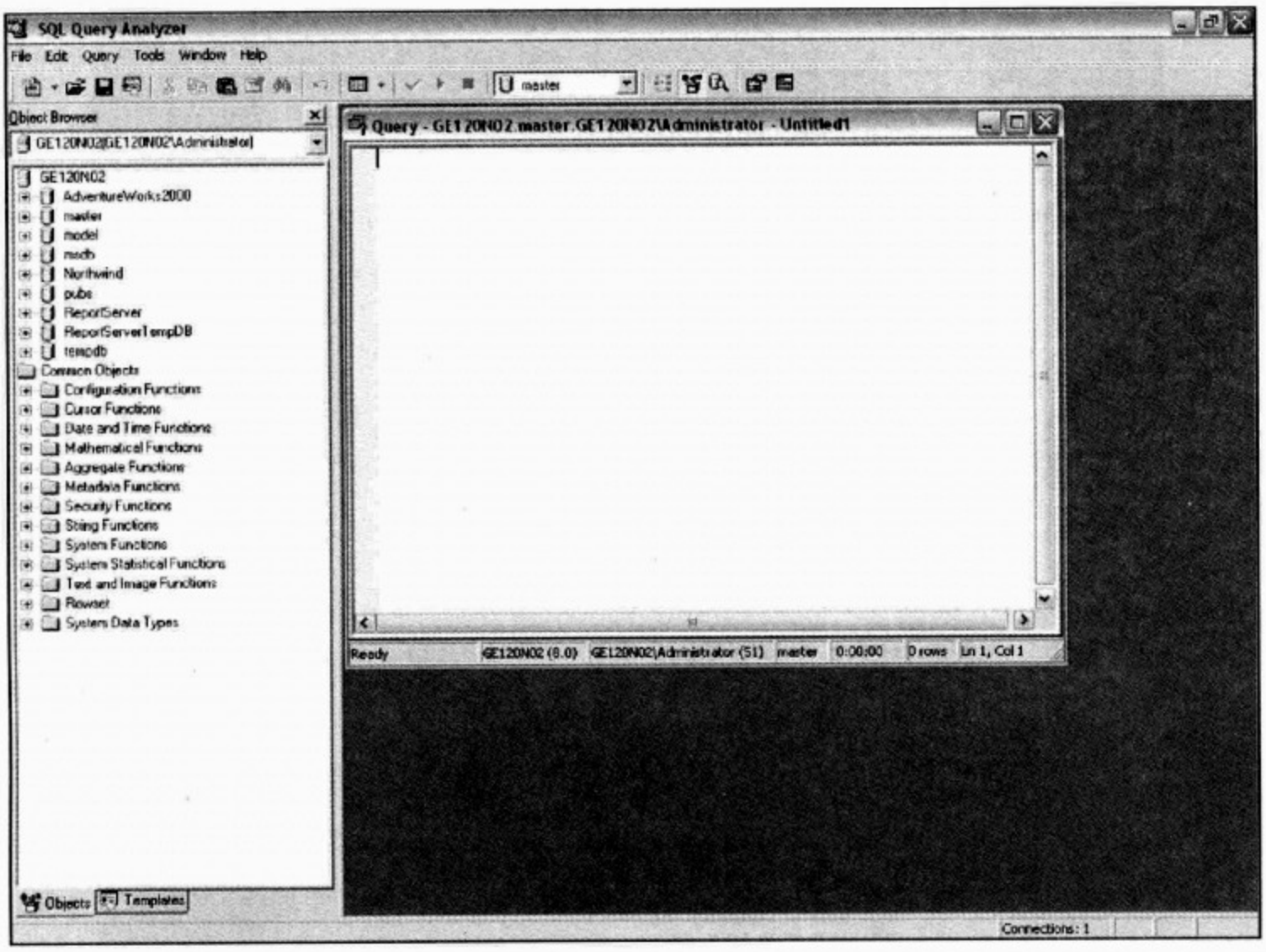


图　16-1

(4) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-2 显示的是第 4 步之后的界面。在结果面板中显示了姓氏是以 B 开头的作者名。如果输入的 Transact-SQL 代码中有错误，则在结果面板中会出现错误信息。如果没有发现错误，那么输入的代码应该与 BSurnames.sql 中的代码相同。

(5) 要匹配姓氏中包含字母 B，不区分大小写，且出现在姓氏的开头或后面都可以，那么可以使用模式 %b%。

在 Query Analyzer 的查询窗口中输入以下 Transact-SQL 代码：

```
USE pubs
SELECT au_lname, au_fname FROM dbo.authors
WHERE au_lname LIKE '%b%'
ORDER BY au_lname
```

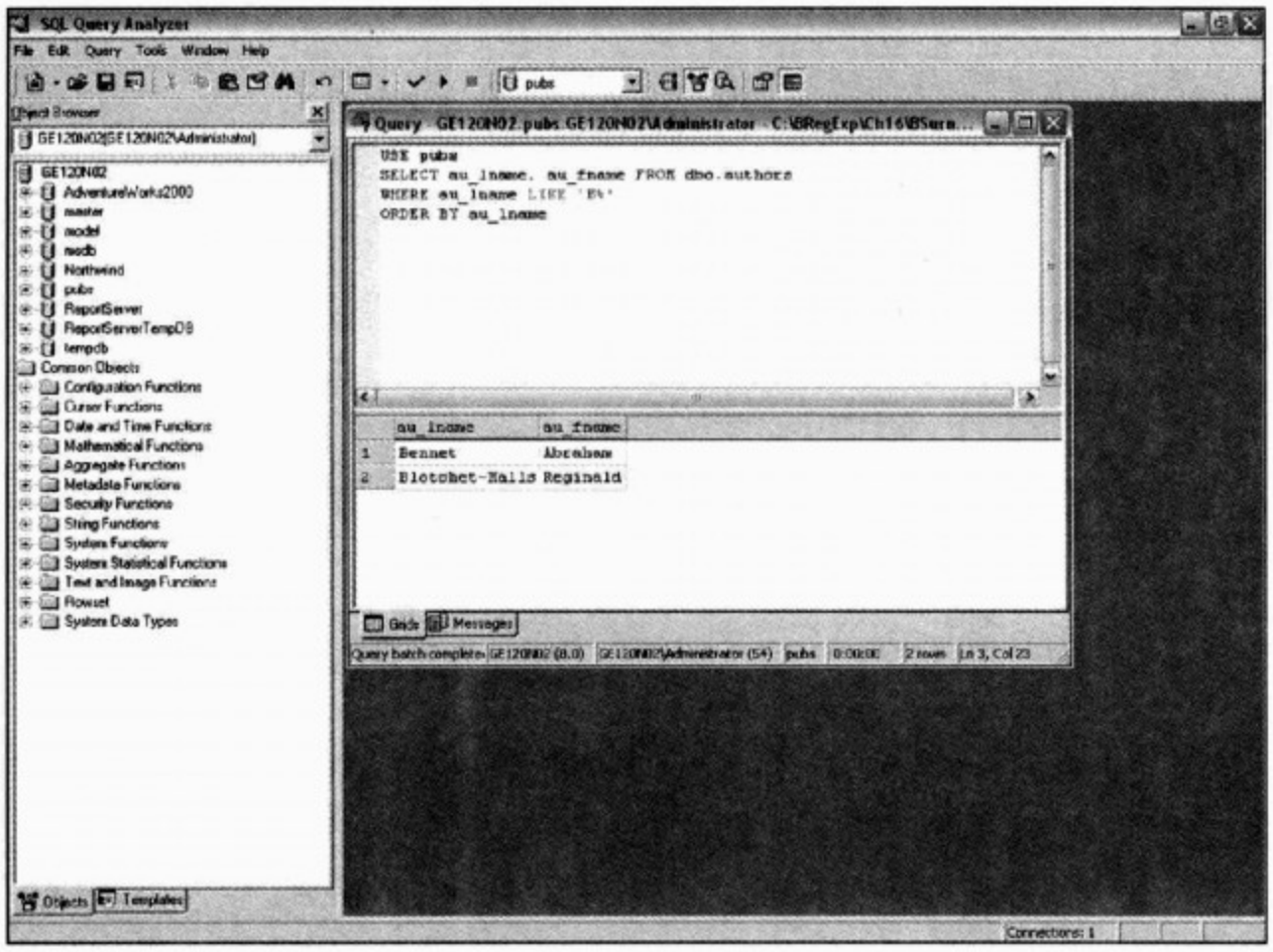


图 16-2

(6) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-3 中显示了结果。结果中每个作者的姓氏都包含一个字母 b，有的是在单词的前面，有的在后面。

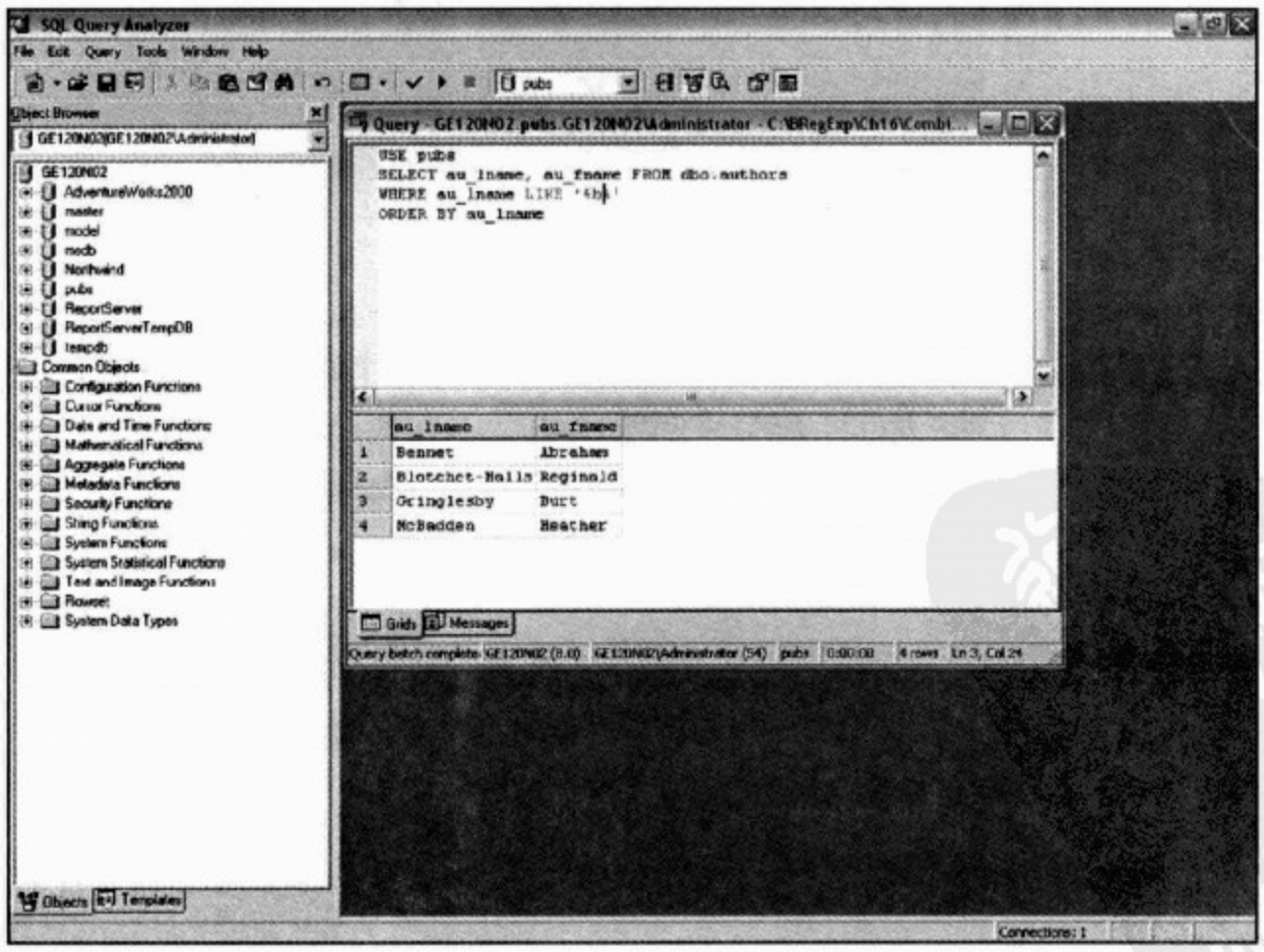


图 16-3

如果输入代码有困难，可以使用文件 BAnywhere.sql 作为替代来运行同样的代码。

(7) 要查找姓氏中包含字符序列 nt 的作者可以使用模式 %nt%。

在 Query Analyzer 的查询窗口中输入以下 Transact-SQL 代码：

```
USE pubs
SELECT au_lname, au_fname FROM dbo.authors
WHERE au_lname LIKE '%nt%'
ORDER BY au_lname
```

(8) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-4 显示执行代码后的结果。其中每个姓氏都包含字符序列 nt。

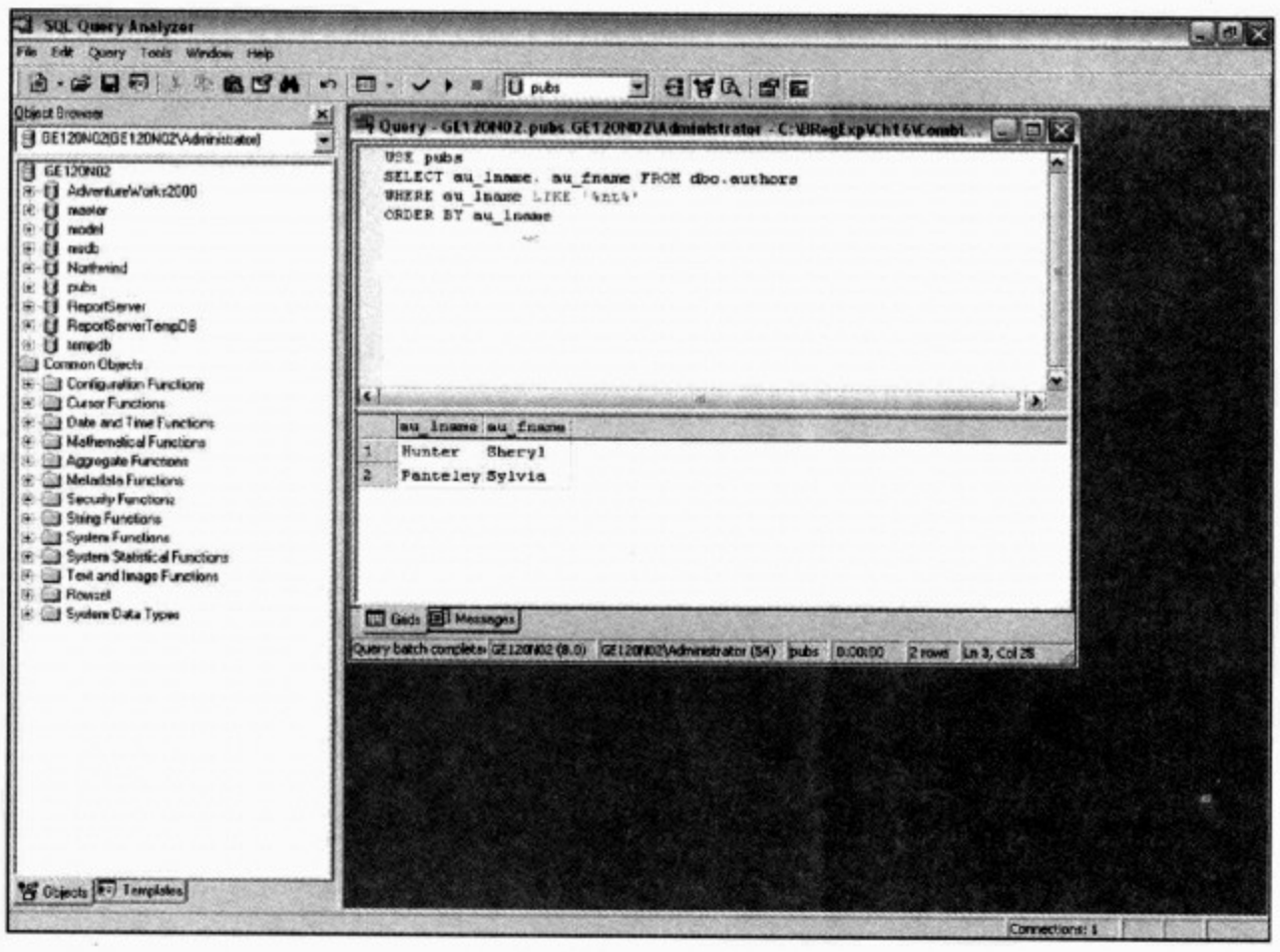


图　16-4

如果不愿在查询窗口中输入代码，可以使用 NTanywhere.sql 中包含的代码。

(9) 在 WHERE 子句中可以多次使用 LIKE 关键字。例如，如果想查找姓氏是以 R 开头并且名字是以字符序列 Al 开头的作者，可以在 au_lname 列中使用模式 R%，在 au_fname 列中使用模式 Al% 。

在 Query Analyzer 的查询窗口中输入以下 Transact-SQL 代码：

```
USE pubs
SELECT au_lname, au_fname FROM dbo.authors
WHERE au_lname LIKE 'R%' AND au_fname LIKE 'Al%'
ORDER BY au_lname
```

(10) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-5 显示的是执行代码后的结果。此时，只返回了一个同时满足两个条件的名字。

(11) 可以将 LIKE 关键字与 NOT 关键字组合使用。例如，要选择姓氏不以 B 开头的

作者，可以使用如下的 WHERE 子句：

```
WHERE au_lname NOT LIKE 'B%'
```

在 Query Analyzer 的查询窗口中输入以下 Transact-SQL 代码：

```
USE pubs
SELECT au_lname, au_fname FROM dbo.authors
WHERE au_lname NOT LIKE 'B%'
ORDER BY au_lname
```

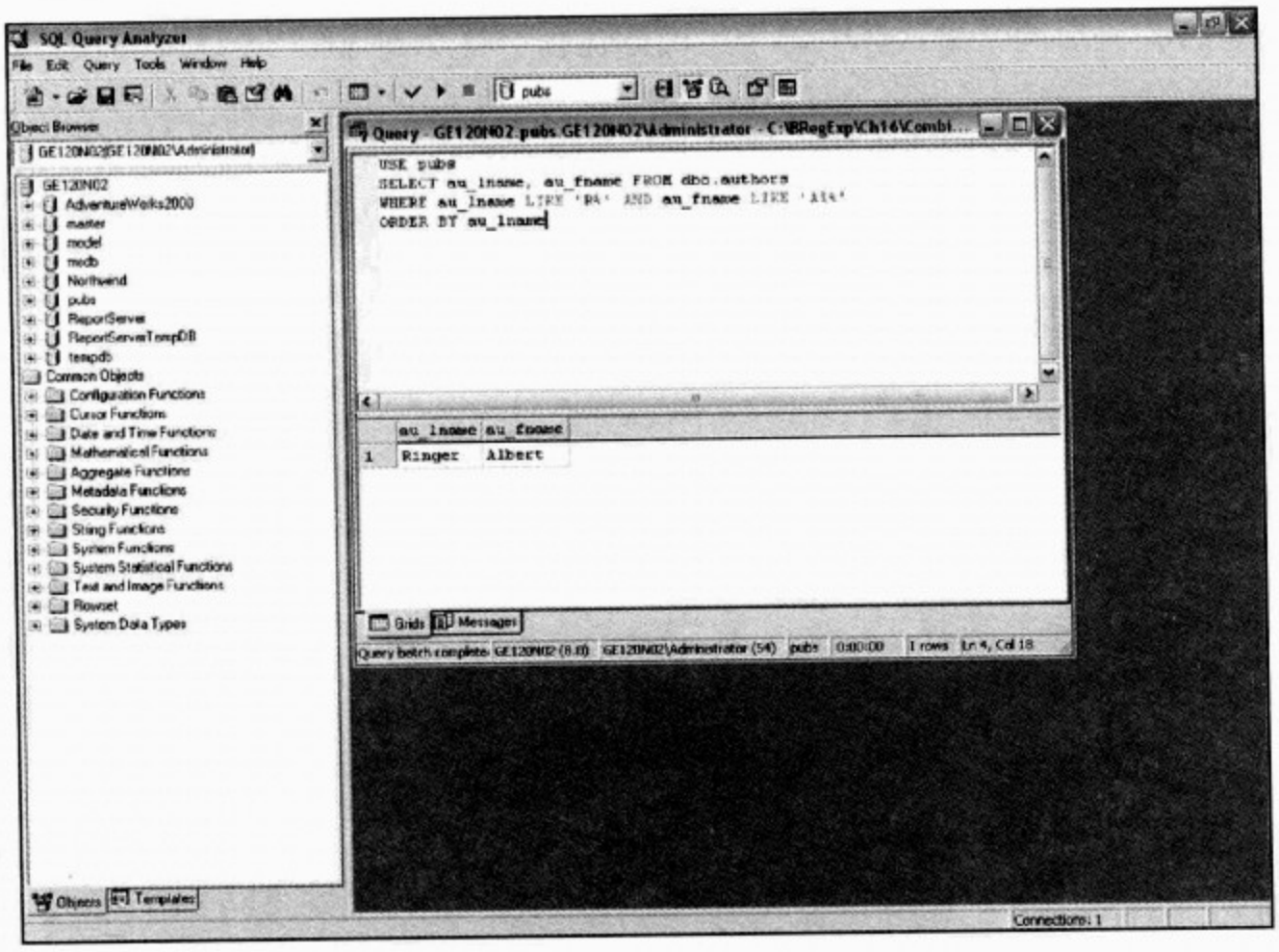


图 16-5

(12) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。代码执行后，pubs 数据库中所有姓氏不以 B 开头的作者全部作为结果返回。

工作原理

模式 B% 匹配位于一个值开头的字符 B 或 b 并且后跟任意数量的任何字符。也就是说，只要字符 b 位于单词的开始处，不论大小写，都会匹配。

模式 %b% 匹配位于姓氏中任何位置的字符 B 或 b。%b% 开始的 % 匹配零个或多个字符。在匹配零个字符的情况下，它匹配以 B 开头的姓。

模式%nt% 组合了字符序列 nt 和前后两个 % 元字符。它匹配包含字符序列 nt 的姓。

当用模式 R% 匹配 au_lname 列的值而用模式 Al% 匹配 au_fname 列的值时，首先 au_lname 列中以 R 开头的记录会匹配。然后，在这些匹配项中只有 au_fname 列中以 Al 开头的记录才会最终匹配。也就是说，只有这两次匹配都成功的记录才会作为结果被显示出来。

当同时使用 NOT 和 LIKE 关键字时，比如下面的 WHERE 子句中，则只有姓与其模式不匹配的记录才会匹配：

```
WHERE au_lname NOT LIKE 'B%'
```

此例中模式匹配以 B 开头的姓。所以，只有不以字符 B 开头的姓才符合匹配条件。

16.2.2　_ 元字符

_ 元字符匹配一个字符。它有点类似于更标准的正则表达式语法中的点(.)元字符。

试一试：使用 _ 元字符

(1) 打开 Query Analyzer。在 Query Analyzer 的查询窗口中输入以下 Transact-SQL 代码：

```
USE Northwind
SELECT SupplierID, ProductID, ProductName FROM dbo.products
WHERE SupplierID LIKE '1%'
ORDER BY SupplierID
```

注意，如 Transact-SQL 代码第一行所示，现在使用的是 Northwind 测试数据库。第三行中的模式 1% 则表示要查找的是由字符 1 构成的 SupplierID 的值。

(2) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-6 显示的是这一步之后的外观。其中第一个记录的 SupplierID 值仅由字符 1 组成。当我们在这个“试一试”的后面使用 _ 元字符时，这些记录将被排除在外。

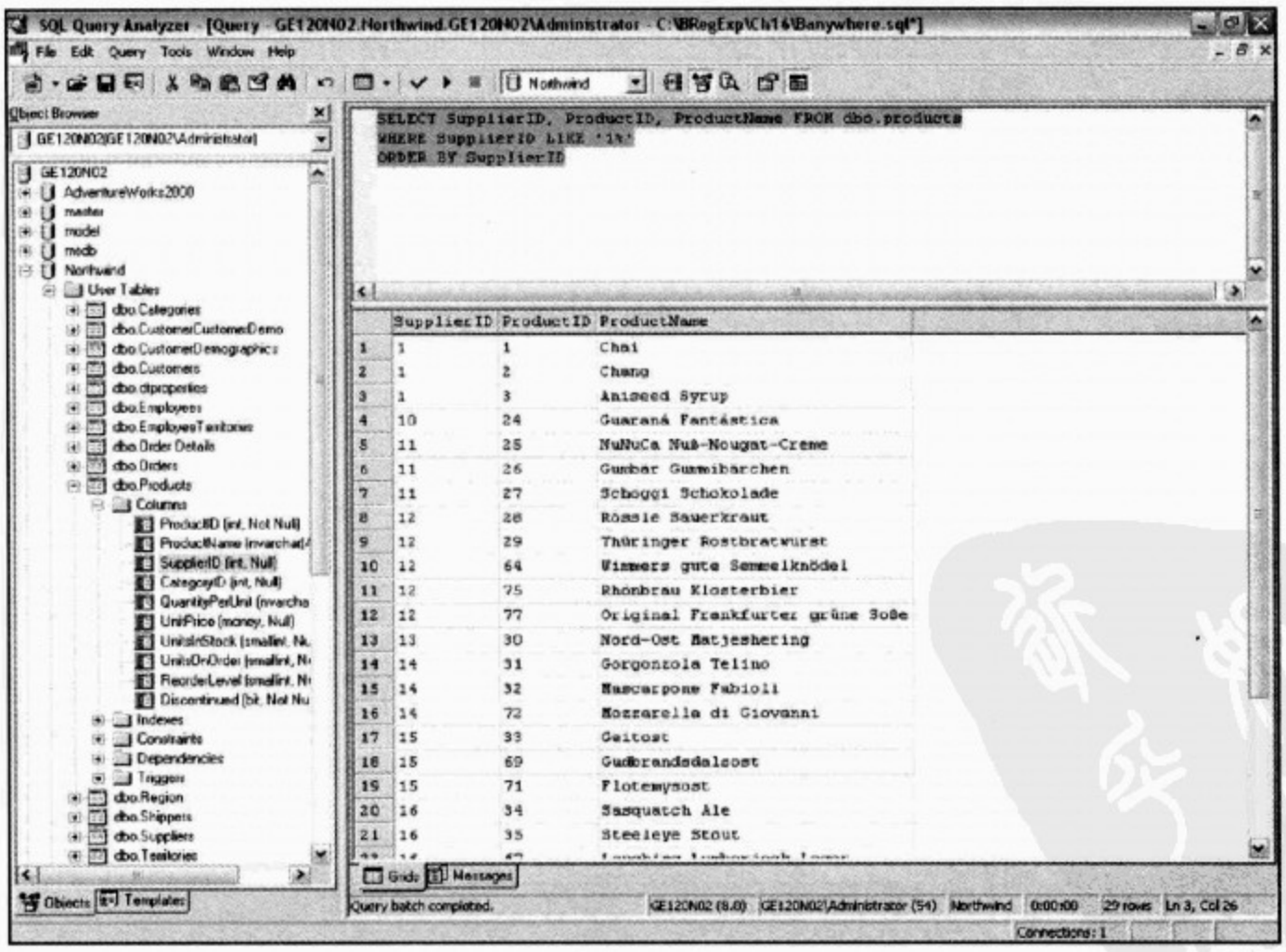


图　16-6

(3) 在 Query Analyzer 的查询窗口中输入以下 Transact-SQL 代码：

```
USE Northwind
SELECT SupplierID, ProductID, ProductName FROM dbo.products
WHERE SupplierID LIKE '1_'
ORDER BY SupplierID
```

在 Transact-SQL 中可以用模式 1_ 替换模式 1%。

(4) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-7 显示的是这一步之后的界面。注意图 16-6 中显示的三行记录此时已经不见了。

工作原理

模式 1% 匹配以数字 1 后跟零个或多个字符开头的 SupplierID 值。所以，它匹配 1 和 10 这样的值。

模式 1_ 则只匹配开头数字 1 后跟一个字符的 SupplierID 值。因此，值 1 不再匹配，因为 1 后不跟字符。但像 10 这样的值依然会匹配，因为 1 后面恰好是一个字符。

16.2.3 字符类

SQL Server 2000 对字符类的支持提供了与其他实现中类似的功能。而且它也支持在字符类中使用范围。

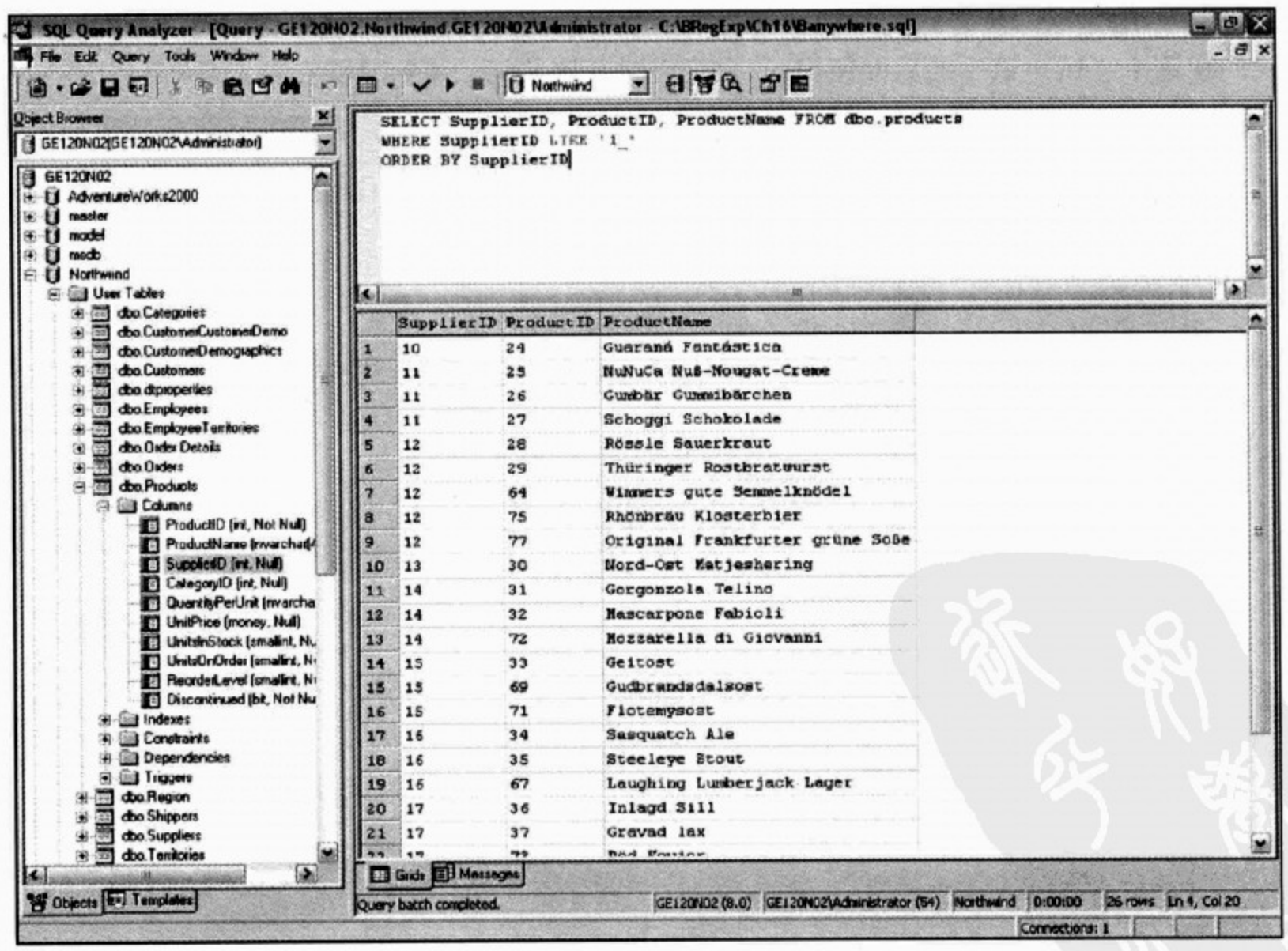


图 16-7

字符类为 % 和 _ 元字符提供了更有用的功能。比如，可以使用模式 [ABC]% 中的字符类 [ABC] 来匹配姓氏是以 A、B 或 C 开头的作者。

试一试：使用字符类

(1) 打开 Query Analyzer，并在 Query Analyzer 的查询窗口中输入以下代码：

```
USE pubs
SELECT au_lname, au_fname from dbo.authors
WHERE au_lname LIKE '[ABC]%'
ORDER BY au_lname
```

(2) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-8 显示的是这一步之后的界面。由于没有姓氏以 A 开头的作者，所以只显示姓氏是以 B 或 C 开头的作者。

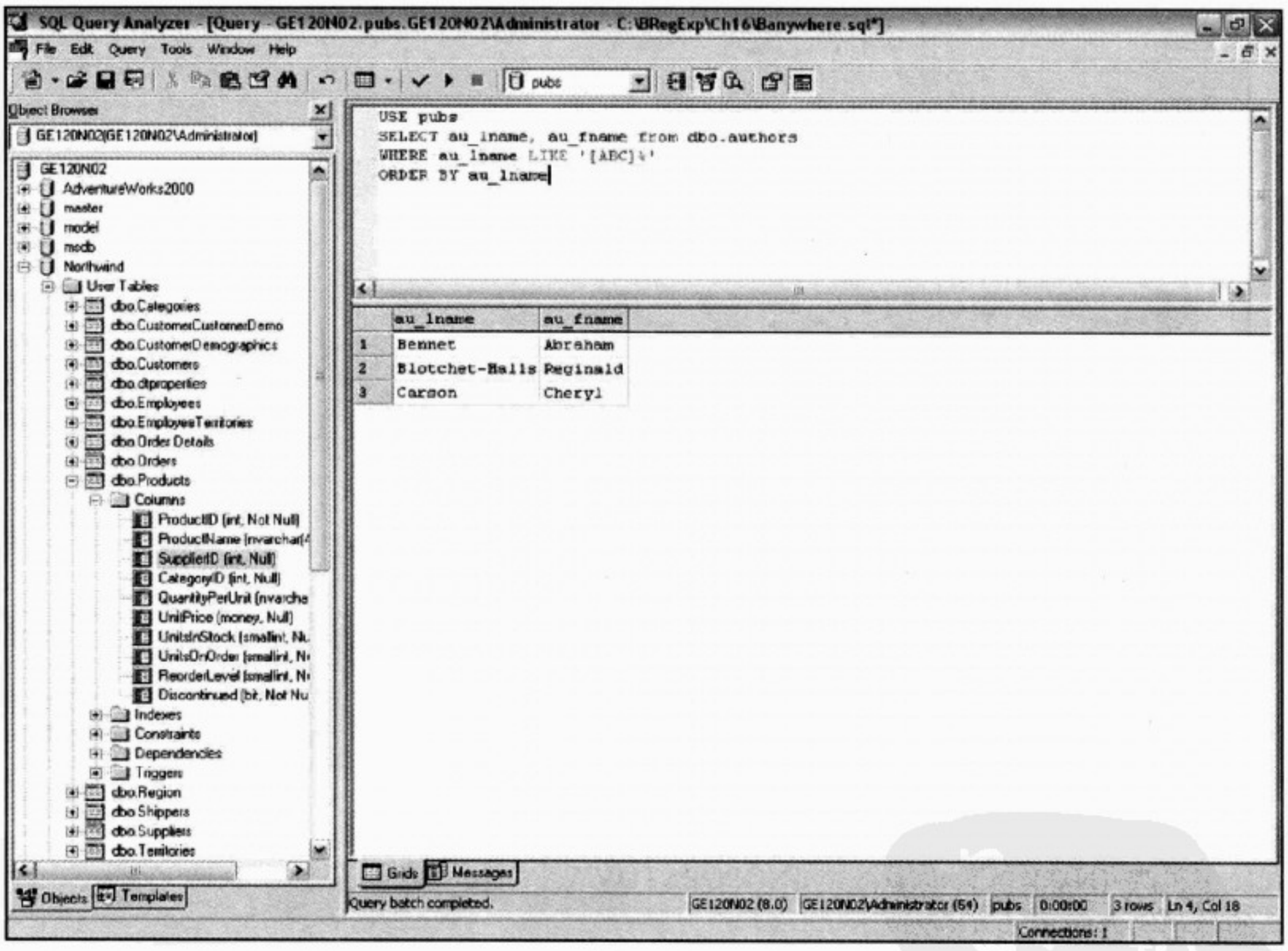


图　16-8

(3) 可以在字符类中使用范围。例如，要显示姓氏中包含 N~Z 的作者，可以在模式 [N-Z]% 中使用字符类 [N-Z]。

在 Query Analyzer 的查询窗口中输入以下代码：

```
USE pubs
SELECT au_lname, au_fname from dbo.authors
```

```
WHERE au_lname LIKE '[N-Z]%'
ORDER BY au_lname
```

(4) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-9 显示的是此步之后的界面。

工作原理

字符类 [ABC] 匹配字符 A、B 或 C，因此模式[ABC]% 等价于模式 A%、B%和 C%的组合。而模式 A%匹配姓氏以 A 开头的作者。模式 B% 和 C% 分别匹配姓氏以 B 和 C 开头的作者。将这三个模式放到一起，模式 [ABC]% 就会匹配姓氏以 A、B 或 C 开头的作者。

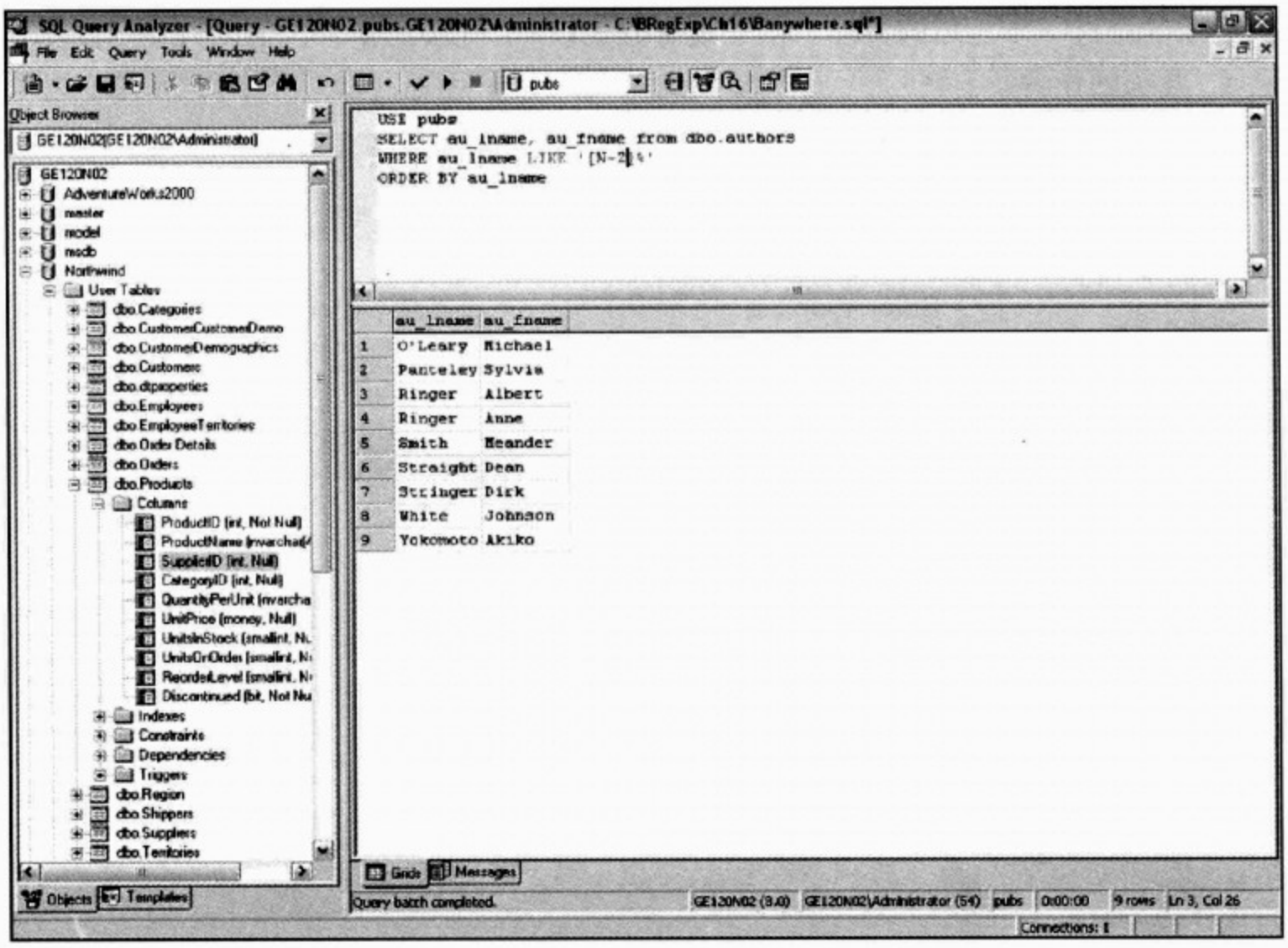


图 16-9

字符类 [N-Z] 等价于字符类 [NOPQRSTUVWXYZ]。这样，任何以 N ~ Z 的字符开头的姓都会匹配模式 [N-Z]%。

16.3 对字符类取反

SQL Server 2000 中也支持对字符类取反。^ 元字符在对字符类取反时应该直接放在开始的方括号后面。

对字符类取反的同时也可以在字符类中使用范围。

试一试：使用取反的字符类

(1) 打开 Query Analyzer，并在查询窗口中输入下列 Transact-SQL 代码：

```
USE AdventureWorks2000
SELECT LastName, FirstName from dbo.contact
WHERE LastName LIKE 'Ad%'
ORDER BY LastName, FirstName
```

这个例子中使用了 AdventureWorks2000 数据库。注意，数据库名 AdventureWorks2000 采取了首字母大写的形式，同时也不要在数据库名中插入空格。前面的代码会匹配以 Ad 开头的姓。稍后使用取反的字符类时，会看到原先匹配的这些行将不再匹配。

(2) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-10 显示的是这一步之后的界面。其中四行姓氏为 Adams 的记录被显示出来了。

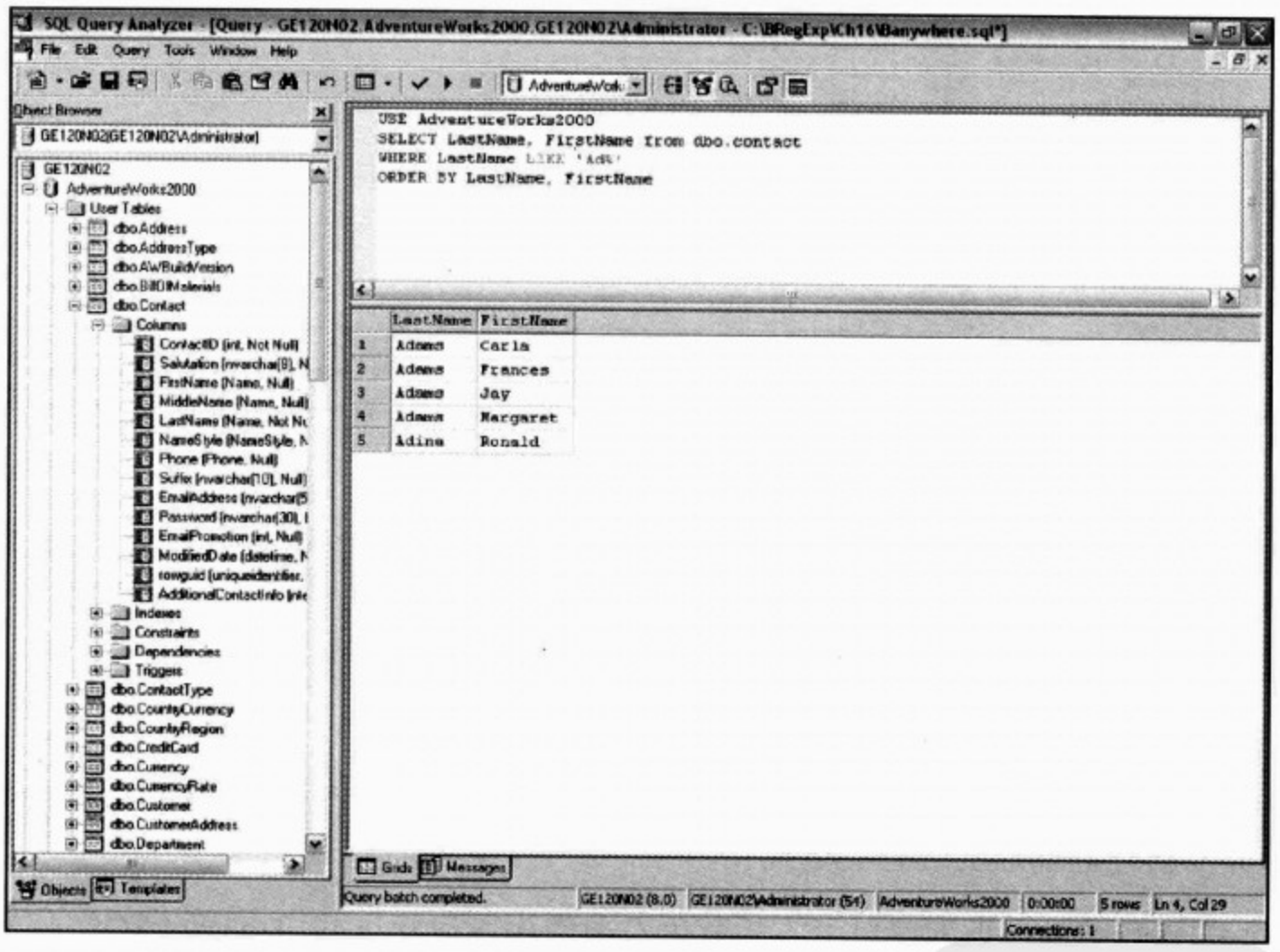


图　16-10

(3) 在查询窗口中输入下面代码：

```
USE AdventureWorks2000
SELECT LastName, FirstName from dbo.contact
WHERE LastName LIKE 'A[^d]%'
ORDER BY LastName, FirstName
```

(4) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-11 显示的是这一步之后的界面。此时包含姓氏为 Adams 的记录没有显示。在使用

ORDER BY 子句的情况下，如果它们匹配将会显示在第 6 行和第 7 行之间。

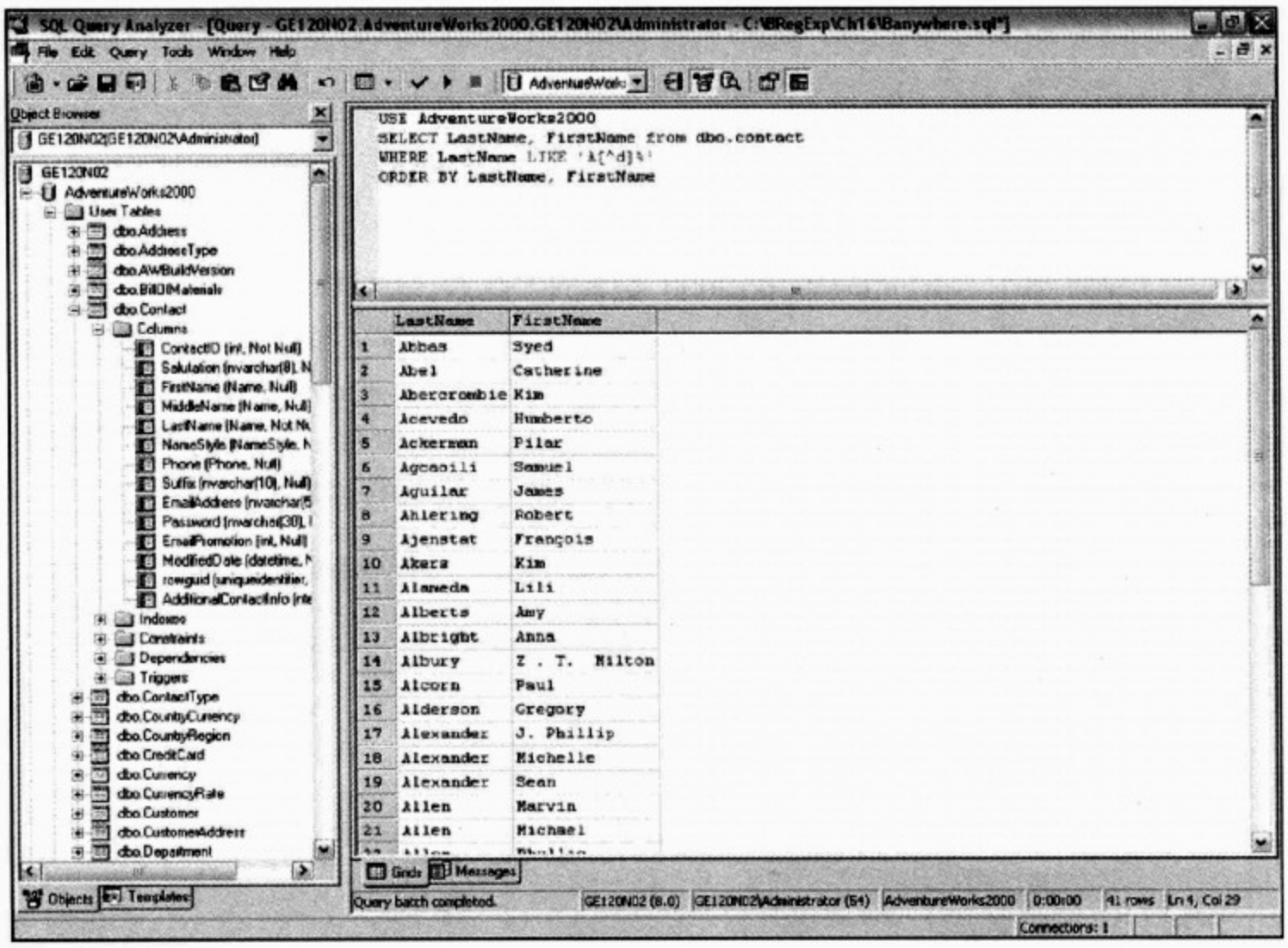


图 16-11

(5) 取反的字符类中也可以使用范围。例如，如果想匹配以 A 开头的姓，而且后续的字符不在 a~k 的范围之内，则可以使用模式 A[^a-k]%。

在查询窗口中输入下面的代码：

```
USE AdventureWorks2000
SELECT LastName, FirstName from dbo.contact
WHERE LastName LIKE 'A[^a-k]%'
ORDER BY LastName, FirstName
```

(6) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-12 中显示的是这一步之后的界面。结果面板中不包含姓氏以 A 开头而第二个字母处于 a~k 这个范围之内的作者记录。

工作原理

模式 Ad% 匹配以字符 A 后跟字符 d 开头的姓。

模式 A[^d]% 匹配以字符 A 后跟任何不是 d 的字母开头的姓。字符类 [^d] 表示除 d 之外的任何字符。

模式 A[^a-k]% 匹配以 A 字符后跟不在范围 a ~ k 之内的第二个字符开头的姓。% 元字符匹配位于姓中第二个字符之后的零个或多个任意字符。

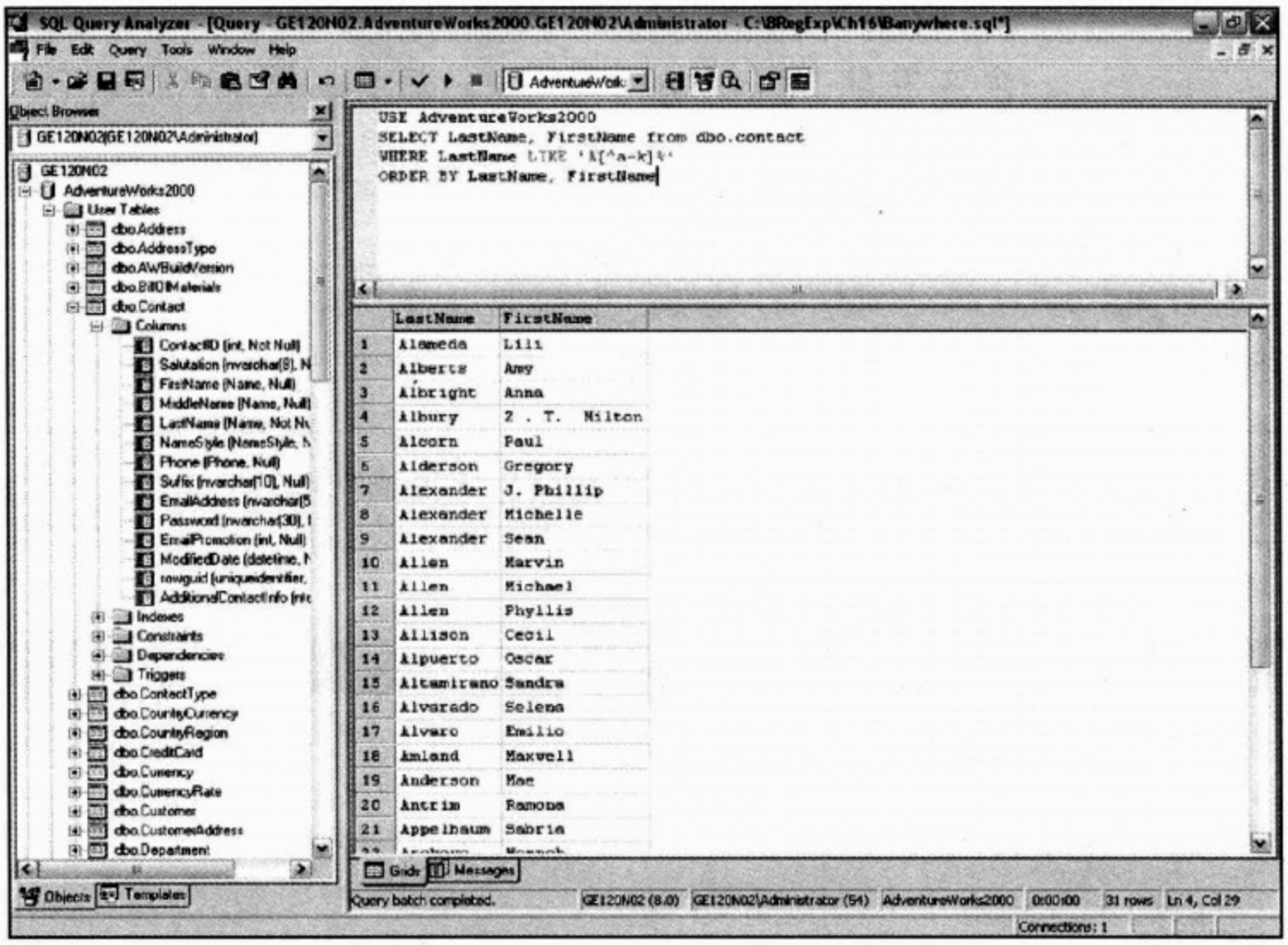


图　16-12

16.4　使用全文搜索

SQL Server 的全文搜索功能(Full-Text Search)是在 SQL Server 7.0 中开始引入的。全文搜索并非 SQL Server 2000 中的主要部分，因为它使用衍生自 Indexing Server(索引服务器)的 MSSearch 服务组件来创建 SQL Server 的外部索引。由于全文索引位于 SQL Server 外部，所以它不会像常规的索引那样被自动更新。

全文索引被包含在全文目录(full-text catalog)中。全文目录则被保存在文件系统，而不是保存在 SQL Server 的数据库中，虽然这些目录及索引仍然是通过数据库来管理的。一个 SQL Server 2000 实例的全文目录必须作为本地的 SQL Server 实例驻留在硬盘中。

表 16-3 中总结了全文索引与 SQL Server 索引之间的比较。

表 16-3　全文索引与 SQL Server 索引

全 文 索 引	SQL Server 2000 索引
位于全文目录的组中。每个全文索引的 SQL Server 数据库只有一个目录	不分组
保存在文件系统中。通过与其关联的关系型数据库管理	与其关联的数据库一同保存

(续表)

全 文 索 引	SQL Server 2000 索引
每张 SQL Server 表最多一个全文索引	每张表允许多个索引
全文索引可以定时发生以响应数据的改变，也可以通过手动来实现	当插入、删除或更新数据时自动更新
可以使用 SQL Server 2000 Enterprise Manager、Wizard 或存储过程来创建、管理或删除全文索引	可以使用 SQL Server 2000 Enterprise Manager、Wizard 或 Transact-SQL 语句来创建或删除常规索引

使用全文搜索功能，首先必须创建相关的全文索引。在这个例子中，将创建一个与 pubs 数据库相关的全文索引。可以使用几种不同的方式来创建全文索引，这里使用 SQL Server 2000 Enterprise Manager 来创建。

试一试：启用和创建全文索引

(1) 打开 SQL Server 2000 Enterprise Manager，在左侧面板中，选中 pubs 数据库，如图 16-13 所示。

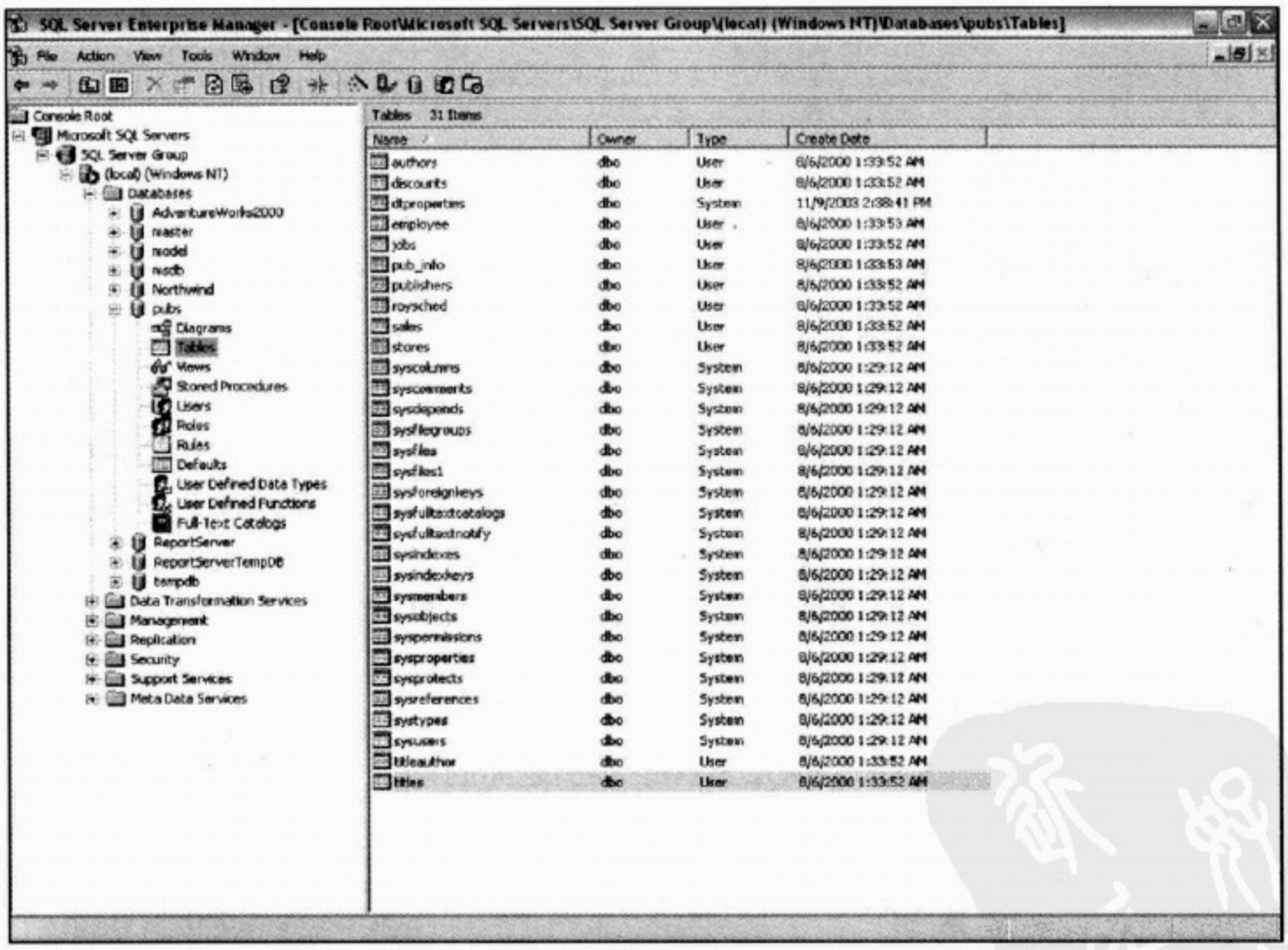


图 16-13

(2) 双击 Full-Text Catalog 图标以确认是否存在基于 pubs 数据库的全文目录。结果是还没有。

(3) 单击位于 Enterprise Manager 界面左上角的返回按钮(左方向箭头)，并单击 titles 表。

(4) 在 Tools 菜单中，选择 Full-Text Indexing 选项。图 16-14 显示的是 Full-Text Indexing Wizard 的初始界面。

图　16-14

(5) 单击 Next 按钮。由于 titles 表只能创建一种索引，所以下拉列表中没有其他选项。图 16-15 显示的界面是一个让你选择使用哪一列创建索引的列表(如前所述，在 pubs 数据库的 titles 表中，只有一种可能的选项)。

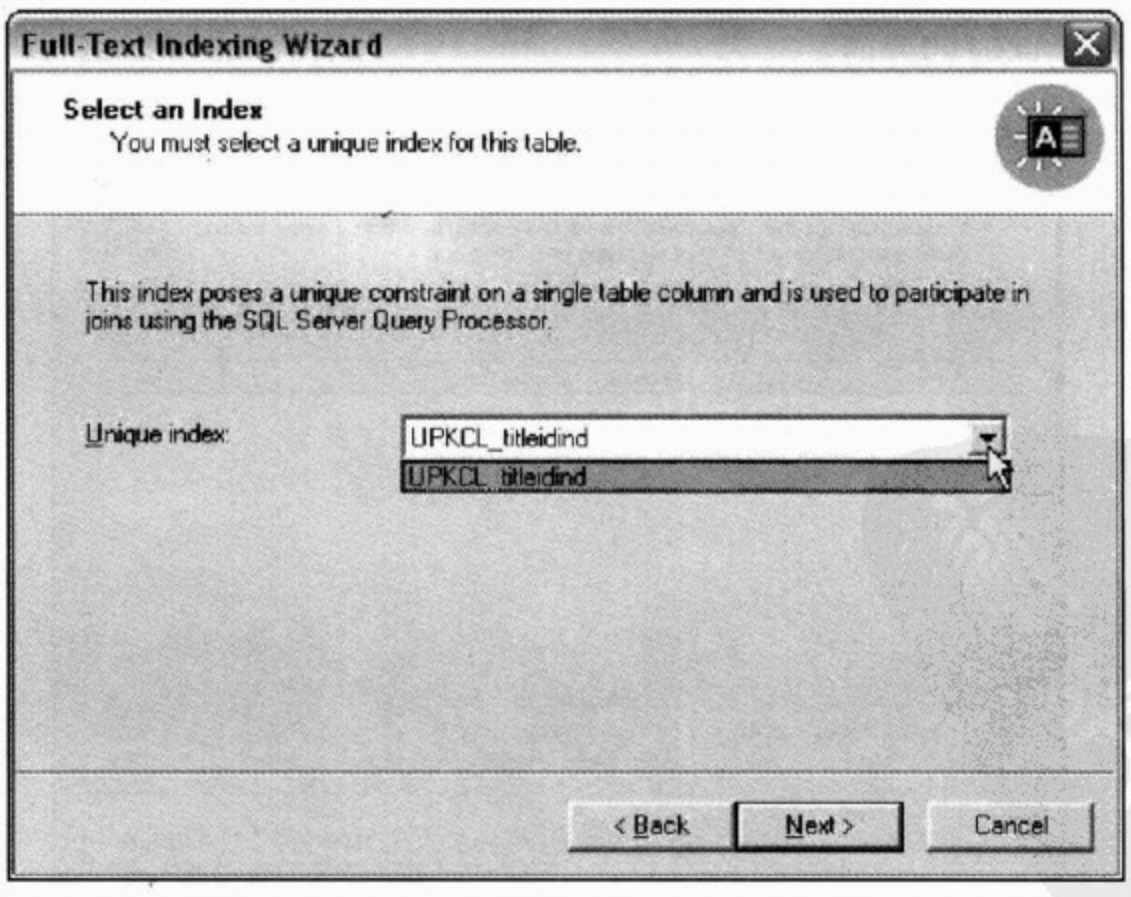


图　16-15

(6) 在下一步显示的界面中，选中 title 和 notes 列前面的复选框，并单击 Next 按钮。图 16-16 显示的是已经选中要创建索引的列的结果。

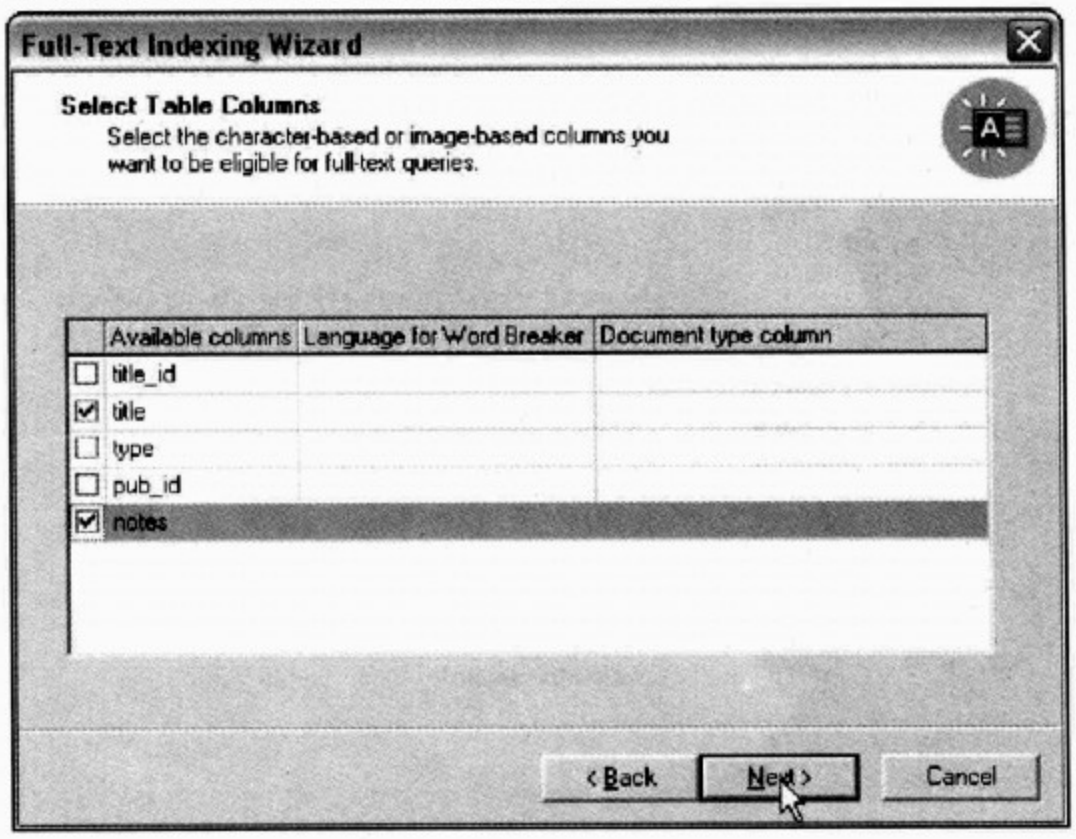


图 16-16

(7) 创建目录名。此处目录命名为 Chap16。

(8) 另外，可以选择一个与默认位置不同的位置来保存创建的目录。

(9) 单击 Next 按钮。现在，我们已经指定了要创建的目录和索引，以及要包含的列。接下来需要指定何时同步(populate)索引信息。

(10) 在下一个界面中，会要求选择同步索引的选项。单击 New Catalog Schedule 按钮。图 16-17 显示的是此时的界面。

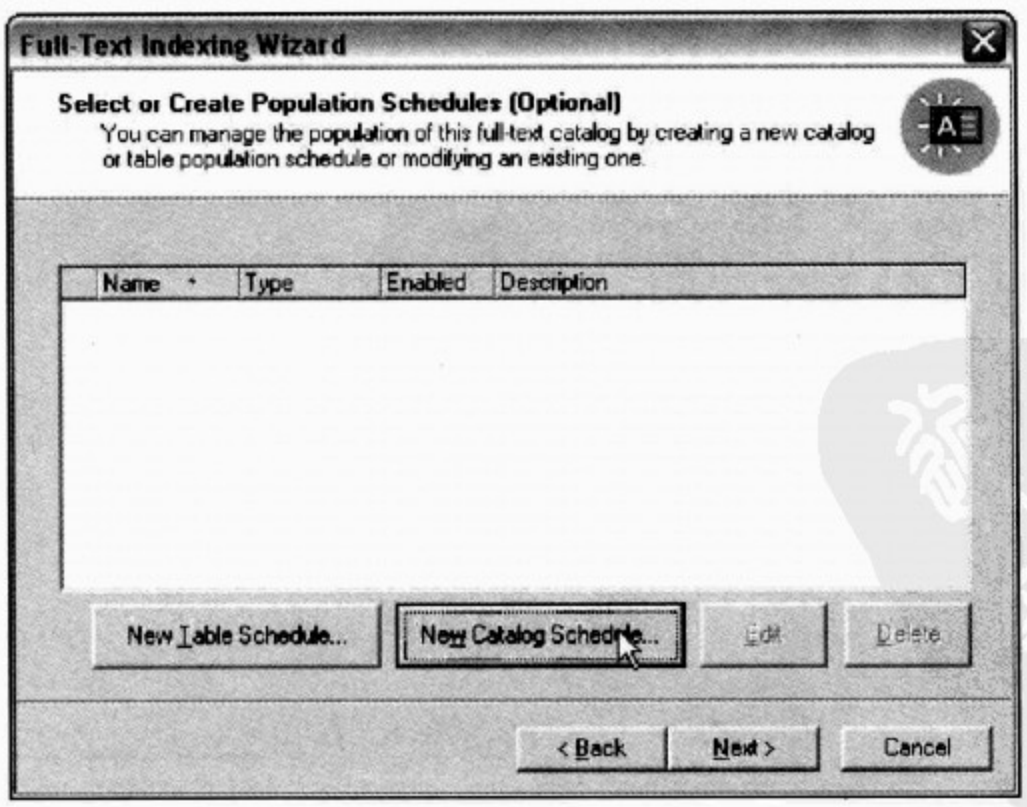


图 16-17

(11) 在下一个窗口中，指定更新索引的时间。选择 Full population 和 One time 单选按钮。

(12) 假设希望马上同步索引，可以选择当前时间几分钟之后的一个适当时间。图 16-18 显示的是此时的界面。

(13) 单击 OK 按钮，然后单击 Next 按钮，最后单击 Finish 按钮。

当 Full-Text Indexing Wizard 尝试去实现在前面几个界面中所选择的设置时，会看到一些提示信息。假设没有错误发生，最后会看到如图 16-19 所示的界面。

此时的信息告诉你还没有对索引进行同步。假设在第 11 步中选择了在几分钟后开始完全同步操作，那么可以等到同步开始。

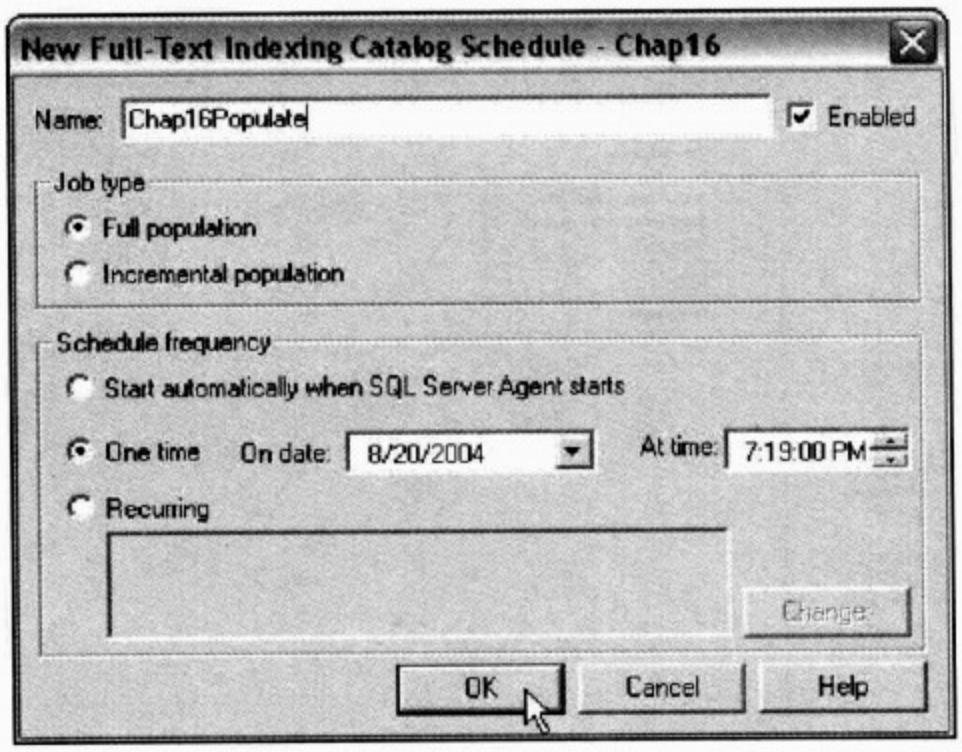


图　16-18

图　16-19

(14) 在 Enterprise Manager 中选中 pubs 数据库的情况下，双击全文索引选项。如果正确创建了 Chap16(或者为索引取的其他名字)，它就会在此时显示出来。如果还没有开始同步，会看到 Status 栏中显示 Idle(空闲)，而 Last Population Date 一栏是空白的。

此时，可以通过手动方式来执行完全同步操作。

(15) 右击 Chap16 目录，在关联菜单中，单击 Start Full Population 选项。此时的界面如图 16-20 所示。

由于 pubs 数据库很小，所以在速度很快的电脑中，完全同步只需瞬间就能完成。当目录同步完成后，就会看到 Last Population Date 栏中显示出相应的日期和时间。图 16-21 显示是在完成同步操作后的界面。

由于若不能正确地创建目录和索引，将无法继续下面的例子，所以以上有关建立全文索引的描述是非常详尽的。假设一切顺利，接下来就可以继续探索 SQL Server 2000 中的全文搜索功能了。

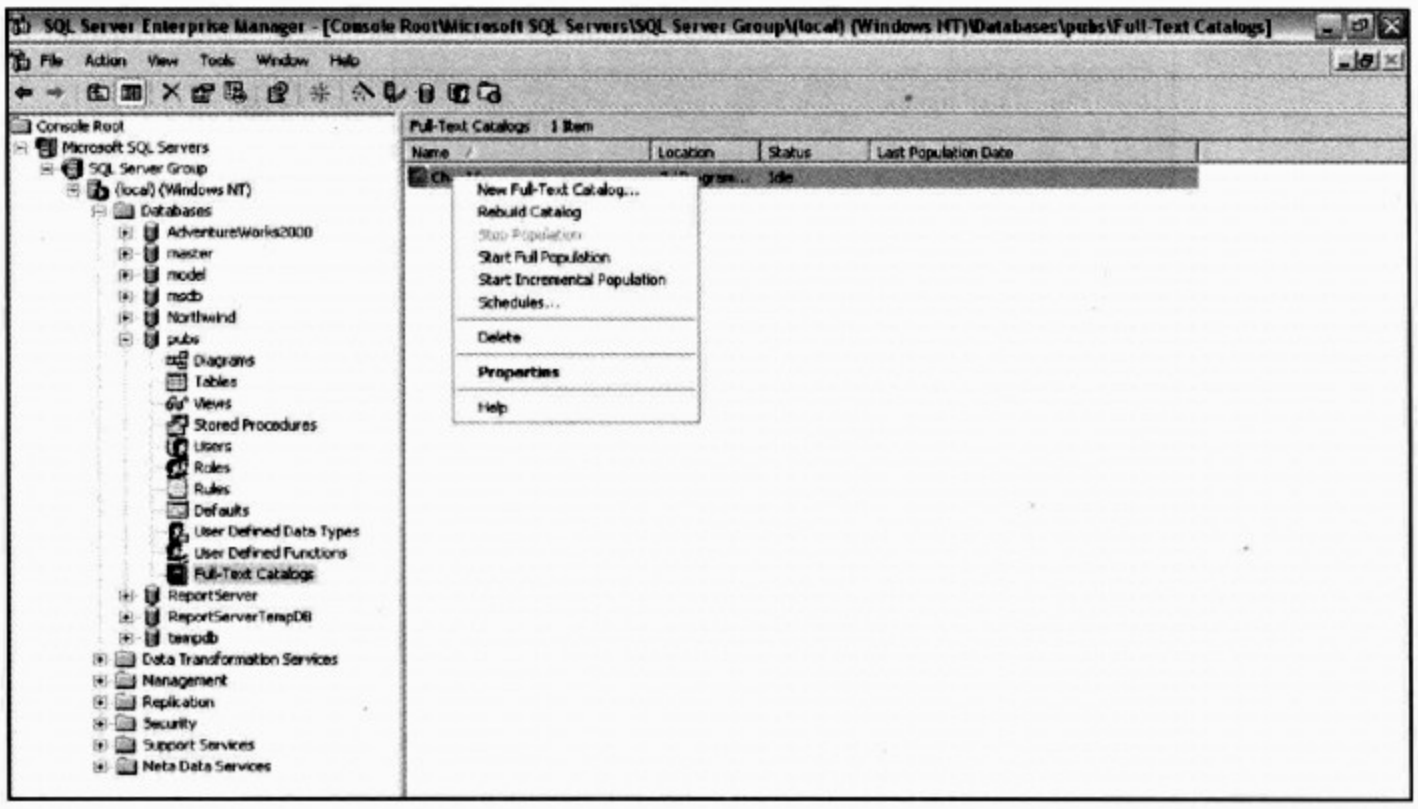


图 16-20

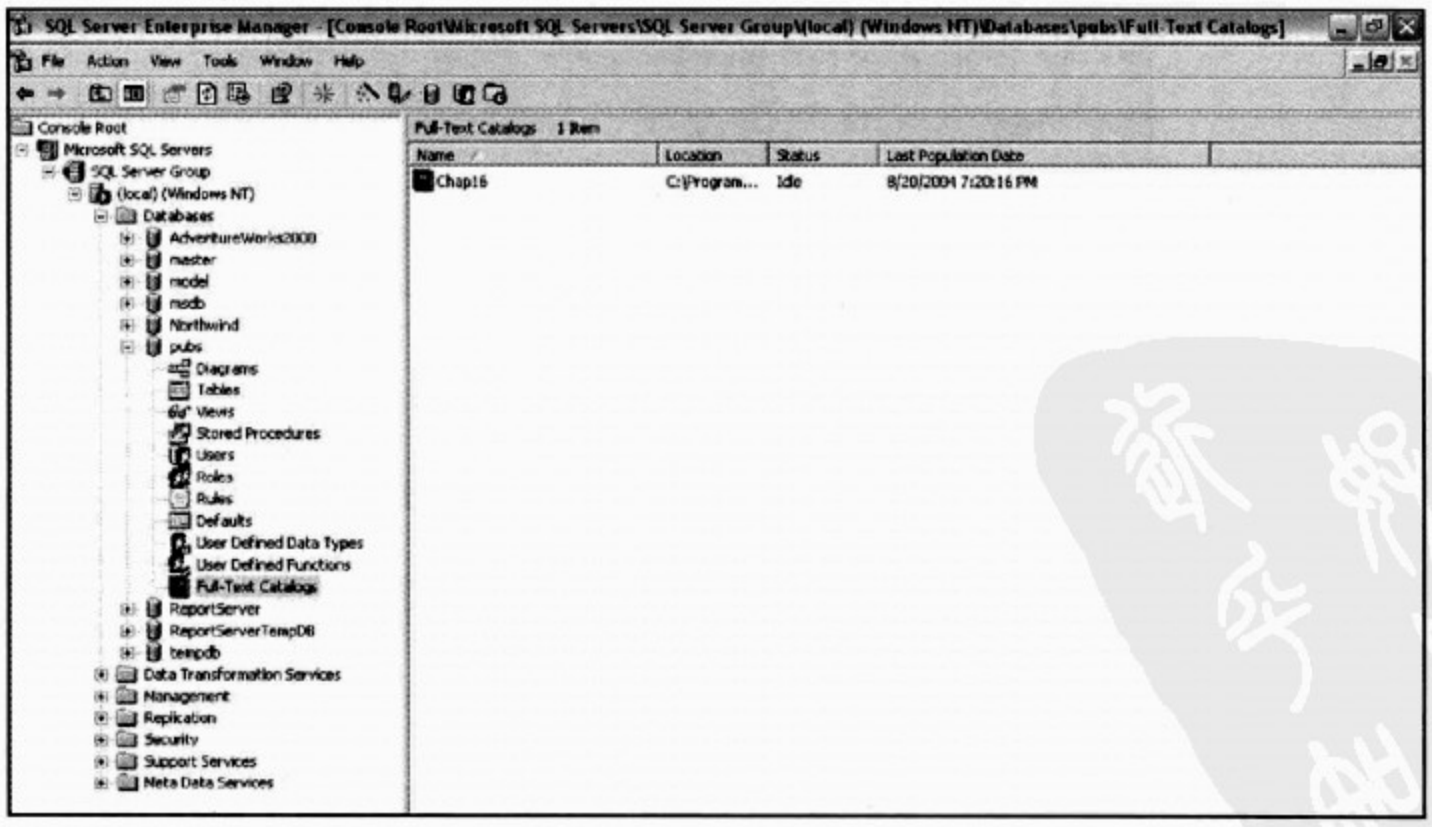


图 16-21

使用 CONTAINS 谓词

作为 SQL Server 全文搜索功能的一个组成部分，CONTAINS 谓词用于 Transact-SQL SELECT 语句的 WHERE 子句中。

我们将基于 titles 表使用 CONTAINS 谓词，需特别注意 title 和 notes 列。为了查看这些列中的内容，可以使用下列 Transact-SQL 代码：

```
USE pubs
SELECT title_id, title, notes FROM titles
```

图 16-22 显示的是执行前面代码之后的界面。

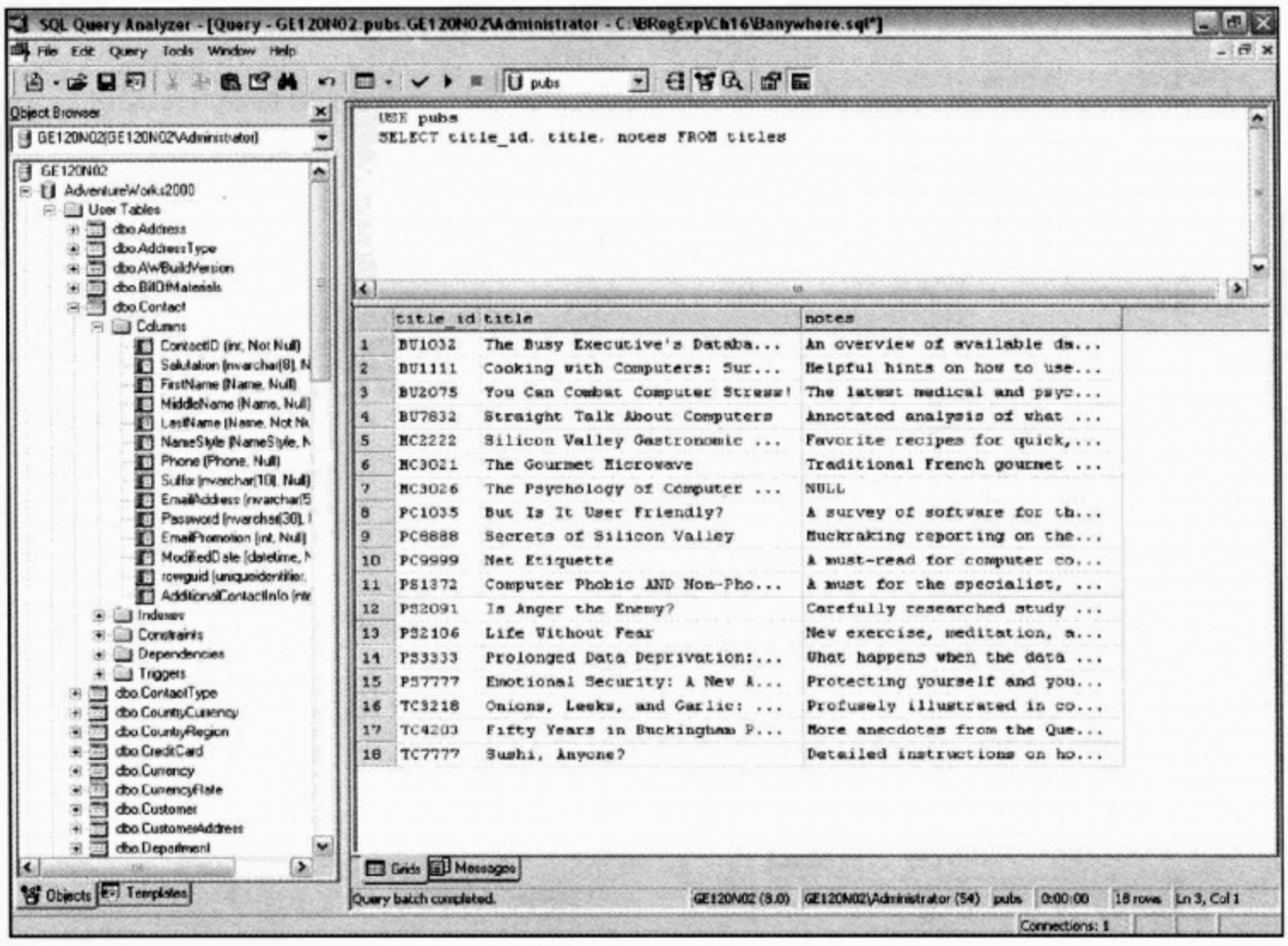


图　16-22

下面的例子搜索 title 列中包含单词 computer 的记录。

试一试：使用 CONTAINS 谓词

(1) 如果没有打开 Query Analyzer，打开它并在查询窗口中输入下列代码：

```
USE pubs
SELECT title_id, title, notes FROM titles
WHERE CONTAINS(title, ' "computer" ')
```

(2) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些 Transact-SQL 代码。图 16-23 显示的是这一步之后的界面。此时结果中只包含三条记录。如图 16-23 中所示，每条记录的 title 列中都包含单词 computer。

CONTAINS 谓词使用两个参数。第一个参数是要搜索的列名，第二个参数是全文搜索的条件。在这个例子中，全文搜索条件非常简单——是一个直接量字符串。

也可以像下面代码中所示的那样使用 LIKE 关键字得到类似的结果：

```
USE pubs
SELECT title_id, title, notes FROM titles
WHERE title LIKE '%computer%'
```

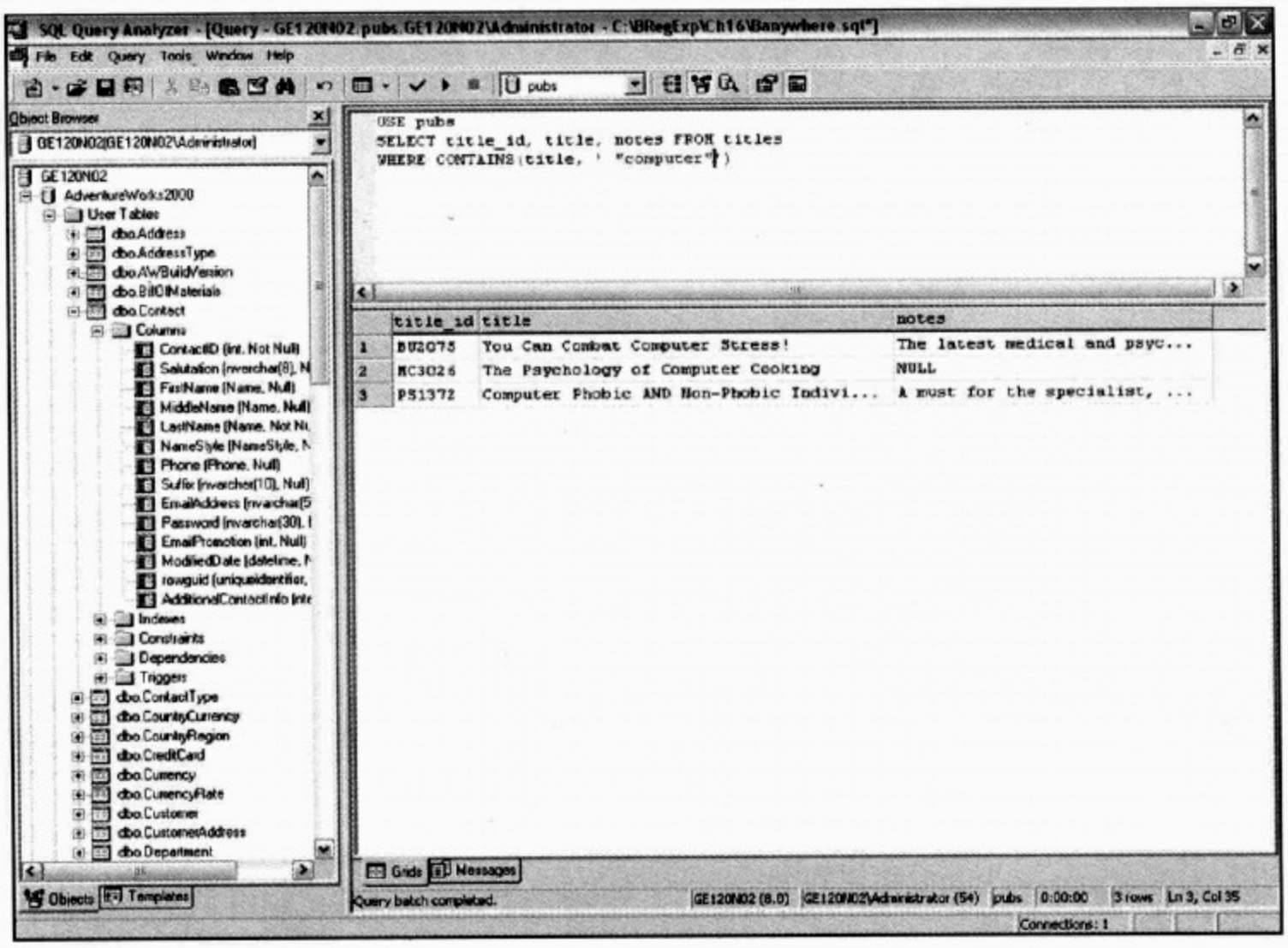


图 16-23

(3) 在 Query Analyzer 的查询窗口中输入前面的代码，并按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些代码。

结果返回五条记录，其中三条包含单词 computer，两条包含复数形式的 computers，它们都与模式%computer%匹配。

CONTAINS 谓词中可以使用 AND 关键字组合两个搜索条件。例如，假设想查找 title 列中同时包含 computer 和 psychology 的记录。就可以使用下列 WHERE 子句：

```
WHERE CONTAINS(title, ' "computer" AND "psychology" ')
```

(4) 在查询窗口中输入下列代码：

```
USE pubs
SELECT title_id, title, notes FROM titles
WHERE CONTAINS(title, ' "computer" AND "psychology" ')
```

(5) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些代码。图 16-24 显示的是此步之后的界面。结果显示只有一行记录匹配，该记录的 title 列中同时包含了单词 computer 和 psychology。

类似地，也可以使用 OR 关键字。

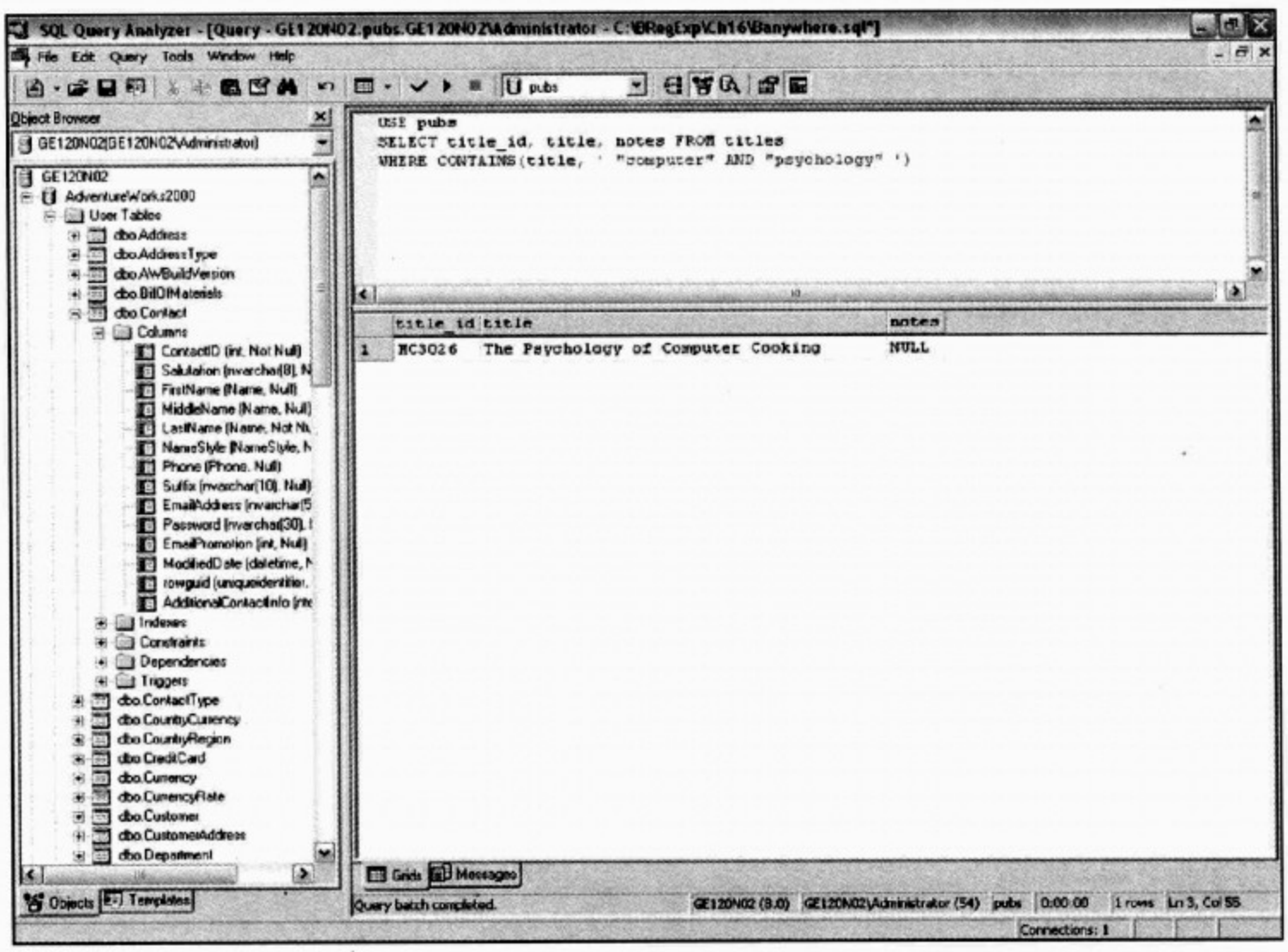


图　16-24

(6) 在查询窗口中输入下列代码：

```
USE pubs
SELECT title_id, title, notes FROM titles
WHERE CONTAINS(title, ' "computer" OR "busy" ')
```

(7) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些代码。三条包含 computer 的 title 列和一条包含 busy 的 title 列的记录会显示出来。而且，还可以匹配以特殊字符(串)开头的单词。例如，下面的 WHERE 子句会匹配包含以 computer 为开头的单词的 title 列：

```
WHERE CONTAINS(title, ' "computer*" ')
```

这样，titles 表中包含 computer 或 computers 的 title 列会显示出来。

使用 NEAR 关键字也可以测试相邻的情况。

(8) 在查询窗口输入下列代码：

```
USE pubs
SELECT title_id, title, notes FROM titles
WHERE CONTAINS(title, ' "computer" NEAR "phobic" ')
```

(9) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些代码。结果同时包含 computer 和 phobic 的 title 列会显示出来。

可以使用 FORMSOF 关键字来测试单词变形(Inflectional)的情况。单词变形也包括复数形式。

(10) 在查询窗口输入下列代码：

```
USE pubs
SELECT title_id, title, notes FROM titles
WHERE CONTAINS(title, ' FORMSOF(INFLECTIONAL,computer)')
```

(11) 按 F5 或单击查询窗口上方的蓝色右向箭头以运行这些代码。如图 16-25 所示，结果中包含 computer 或 computers 的 title 列显示了出来。

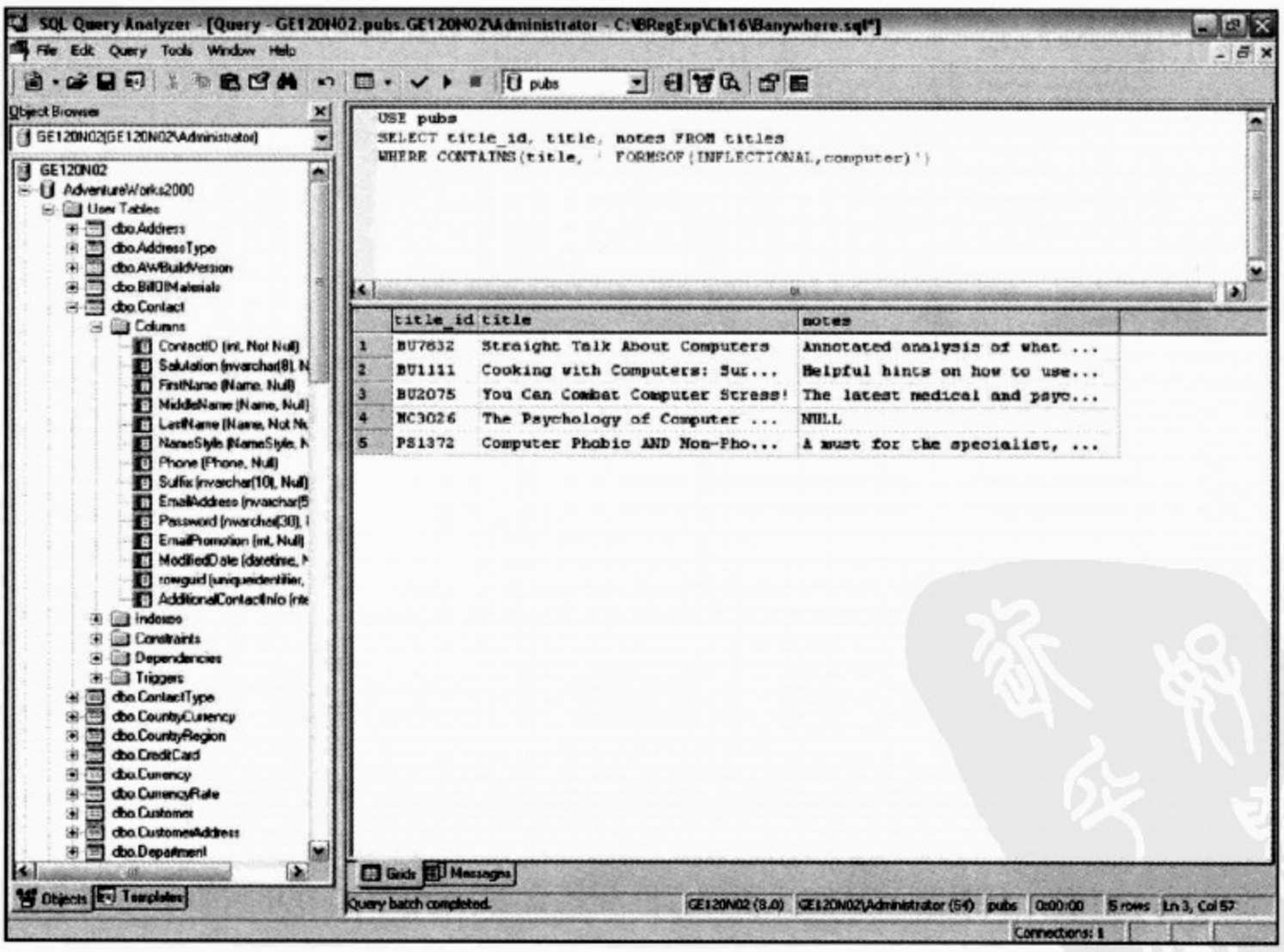


图 16-25

全文搜索的功能极其强大，尤其是在搜索大量的文本数据时更加突出。而通过 pubs 数据库只能在一定限度内测试全文搜索功能。

工作原理

CONTAINS 谓词会查找完整的单词，比如下例中的 computer：

```
WHERE CONTAINS(title, ' "computer" ')
```

CONTAINS 谓词中使用 AND 关键字可以搜索单词的组合：

```
WHERE CONTAINS(title, ' "computer" AND "psychology" ')
```

OR 关键字则提供了与标准的正则表达式中交替选择相似的功能：

```
WHERE CONTAINS(title, ' "computer" OR "busy" ')
```

CONTAINS 谓词可以匹配包含由 * 元字符指定的字符序列的单词：

```
WHERE CONTAINS(title, ' "computer*" ')
```

通过 CONTAINS 谓词可以实现搜索相邻单词，这与标准的双向查找有类似之处。

```
WHERE CONTAINS(title, ' "computer" NEAR "phobic" ')
```

此外，通过 INFLECTIONAL 关键字可以在仅指定一种单词形式的情况下，运用对单词变形的知识查找所有相关的单词：

```
WHERE CONTAINS(title, ' FORMSOF(INFLECTIONAL,computer)')
```

这与标准正则表达式不同，标准正则表达式匹配字符序列不需要理解英语单词的各种必要形式。而 INFLECTIONAL 关键字则是在理解英语单词中相应变形的前提下完成操作的。

16.5　图像字段中的筛选器

SQL Server 2000 可以在图像字段中存储多种类型的文档。比如说，可以将一些 Microsoft Word 文档保存在这种字段中。而且，SQL Server 还有几个内置的筛选器，可以保留图像字段中的文档，这样它们的文本化内容就可以编制成索引以供搜索。

将多个文档存储在一个图像字段中，对通过全文搜索功能来搜索多个文档十分有用。

16.6　练习

下列练习题可以测试对本章所介绍的新内容的理解程度。

1. 使用 pubs 数据库创建相应的 Transact-SQL 代码，只匹配姓氏为 Green 和 Greene 的记录。提示：使用模式 G%可以找 pubs 数据库中姓氏以 G 开头的记录。

2. 使用 pubs 数据库创建相应的 Transact-SQL 代码，匹配图书标题中包含字符序列 data 的记录。提示：图书标题包含在 pubs 数据库中 dbo.titles 表的 title 列中。

第17章

在 MySQL 中使用正则表达式

MySQL 是一种关系型数据库，它与久经考验的商业化关系型数据库管理系统(比如 IBM 的 DB2 和 Microsoft 的 SQL Server)相互竞争。虽然 MySQL 还缺乏企业级关系型数据库管理系统市场中主流产品的一些特性，但它仍不失为一款功能强大而且极具灵活性的数据库管理系统。

MySQL 对正则表达式具有广泛支持，因而可以对 MySQL 数据库中的文本化数据实现强大而且富有灵活性的搜索。

在本章中将学习以下内容：

- MySQL 支持哪些元字符
- 如何使用 SQL 元字符 _ 和 %
- 如何在 MySQL 中使用 REGEXP 功能

本章所介绍的功能基于 MySQL 4.0 版，该版本也是在写作本书时推荐使用的产品。然而，beta 版的 MySQL 4.1 和 MySQL 5.0 的 alpha 版已开始开发了。虽然可以预见在 4.1 和 5.0 版中仍然会继续支持本章介绍的正则表达式功能，但软件开发过程中通常也会存在不确定的因素。

17.1 MySQL 简介

MySQL 的数据库产品可以从 www.mysql.com 下载。在写作本书时，MySQL 的下载页面是 http://dev.mysql.com/downloads。

如果想在 Windows 中安装 MySQL，则下载时必须选择适合 Windows 平台的相应版本(成品版或 alpha、beta 版)。本章的例子在 MySQL 4.0 中运行测试通过。

下载完成后，将下载的文件解压缩到一个临时目录中。在这个临时目录中，运行 Setup.exe 文件。本章中的例子假设将 MySQL 安装在 c:\mysql 目录中。如果安装在其他目录中，则需要在后面的例子中调整相应的路径。

在 Windows XP 中，MySQL 是作为一个 Windows 服务运行的。根据是否安装过 MySQL 的早期版本，可能需要手工启动 MySQL 服务。为此，需要单击 Start(开始)按钮，选择 Control Panel(控制面板)。假设在使用经典视图的状态下，选择 Administrative Tools(管

理工具)，选择 Services(服务)，然后找到 MySql 服务。此时会显示该服务的当前状态，如果 MySQL 服务的状态栏是空白的，选中 MySQL 服务，然后单击服务窗口左上角附近的 Start(启动)链接来启动 MySQL 服务。

打开命令提示符窗口。假设将 MySQL 安装在 c:\mysql，定位到 c:\mysql\bin 目录，在命令行输入以下命令：

```
mysql
```

mysql 实用程序会启动。在本章后面的例子中，都将通过这个 mysql 实用程序的命令行来下达 SQL 命令。

如果安装了 MySQL 且启动了 MySQL 服务，此时会看到与图 17-1 类似的屏幕界面。

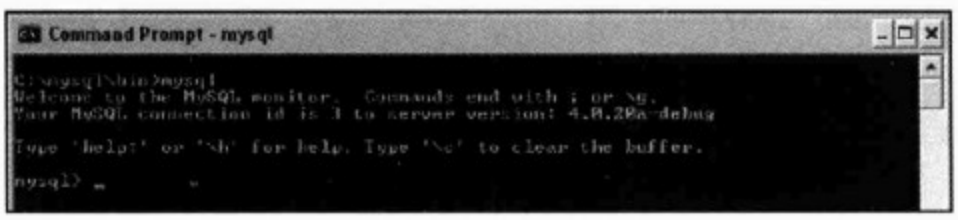


图　17-1

本章的例子将使用一个名为 BRegExp 的数据库。首先，需要创建该数据库。

在 Windows 平台的 MySQL 中，除数据库本身外数据库对象的名称是区分大小写的。这是因为 MySQL 数据库以操作系统中文件的形式保存，而 Windows 操作系统不支持区分大小写的文件名，所以 Windows 平台中的 MySQL 好像是不区分数据库名称的大小写一样。而在 Unix 和 Linux 平台中，MySQL 的数据库名称是区分大小写的，因而与其他 MySQL 数据库对象能够保持一致。

要创建 BRegExp 数据库，在 mysql 的命令行提示符中执行以下命令：

```
CREATE DATABASE BRegExp;
```

一定要在命令的结尾处包含分号，否则 mysql 实用程序会一直等，直到输入分号才会执行该命令。对于较复杂的 SQL 命令，建议将各个子句分别写在几行中， 这样既方便又能增强可读性。

MySQL 4.0 增强版较其早期版本已经修改了数据库对象的默认权限。虽然目的是增强安全性，但同时也牺牲了易用性。因此，可能需要花点时间研究一下下载的版本中包含的权限文档。本章内容假设你已经按照自己使用版本的权限文档对 MySQL 的权限进行了适当的配置。

如果成功创建了 BRegExp 数据库，则会看到与图 17-2 类似的界面。

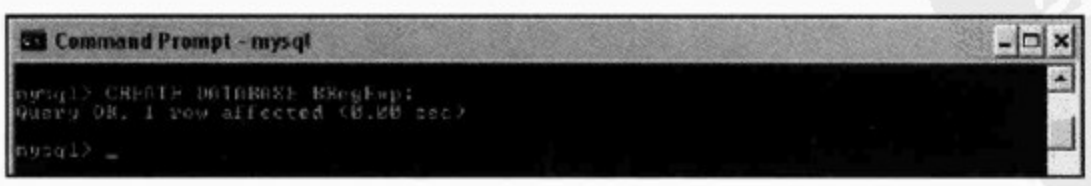


图　17-2

在 mysql 命令行中，执行以下命令切换到使用 BRegExp 数据库的状态：

```
USE BRegExp;
```

此时，会看到如下反馈信息：

```
Database Changed
```

现在，可以在数据库 BRegExp 中创建数据表，以便对其中包含正则表达式的数据应用 SQL 查询。

在 mysql 命令行中输入 EXIT 命令退出 mysql 实用程序。

BRegExp 中将要添加的第一个表用于测验一个简单的 SQL 正则表达式结构。

通过执行 SQL 脚本——People.sql，可以在 BRegExp 数据库中创建一个 People 表，同时向表中插入一些数据。将 ID 列的值设置为 NULL 可以让 MySQL 为 ID 列提供自动递增的值。该脚本的内容如下：

```
USE BRegExp;
CREATE TABLE People
 (ID INT PRIMARY KEY AUTO_INCREMENT,
  LastName VARCHAR(20),
  FirstName VARCHAR(20),
  DateOfBirth DATE);
INSERT INTO People
 (ID, LastName, FirstName, DateOfBirth)
  VALUES
    (NULL, 'Smith', 'George', '1959-11-11'),
    (NULL, 'Armada', 'Francis', '1971-03-08'),
    (NULL, 'Schmidt', 'Georg', '1981-10-09'),
    (NULL, 'Clingon', 'David', '1944-11-01'),
    (NULL, 'Dalek', 'Eve', '1953-04-04'),
    (NULL, 'Bush', 'Harold', '1939-11-08'),
    (NULL, 'Burns', 'Geoffrey', '1960-08-02'),
    (NULL, 'Builth', 'Wellstone', '1947-10-05'),
    (NULL, 'Thomas', 'Dylan', '1984-07-07'),
    (NULL, 'LLareggub', 'Dai', '1950-11-02'),
    (NULL, 'Barns', 'Samuel', '1944-06-01'),
    (NULL, 'Claverhouse', 'Henry', '1931-08-12'),
    (NULL, 'Litmus', 'Susie', '1954-11-03');
```

下面的命令假设已将上面的脚本文件下载到 c:\BRegExp\Ch17 目录，并且已经通过 MySQL 安装目录的 bin 目录打开了命令行窗口。在命令行提示符窗口中执行以下命令：

```
mysql <c:\BRegExp\Ch17\People.sql
```

命令中的 < 字符表示 mysql 实用程序执行的 SQL 脚本的位置。

如果脚本执行成功，则命令行提示符窗口中不会显示错误信息。

可以通过运行 mysql 实用程序来确认已经成功创建了表。启动 mysql 实用程序后，在命令行中执行以下命令：

```
USE BRegExp;
```

然后再执行：

```
SELECT * FROM People;
```

如果脚本执行成功，会看到如图 17-3 所示的屏幕外观，其中显示了 People 表的内容。

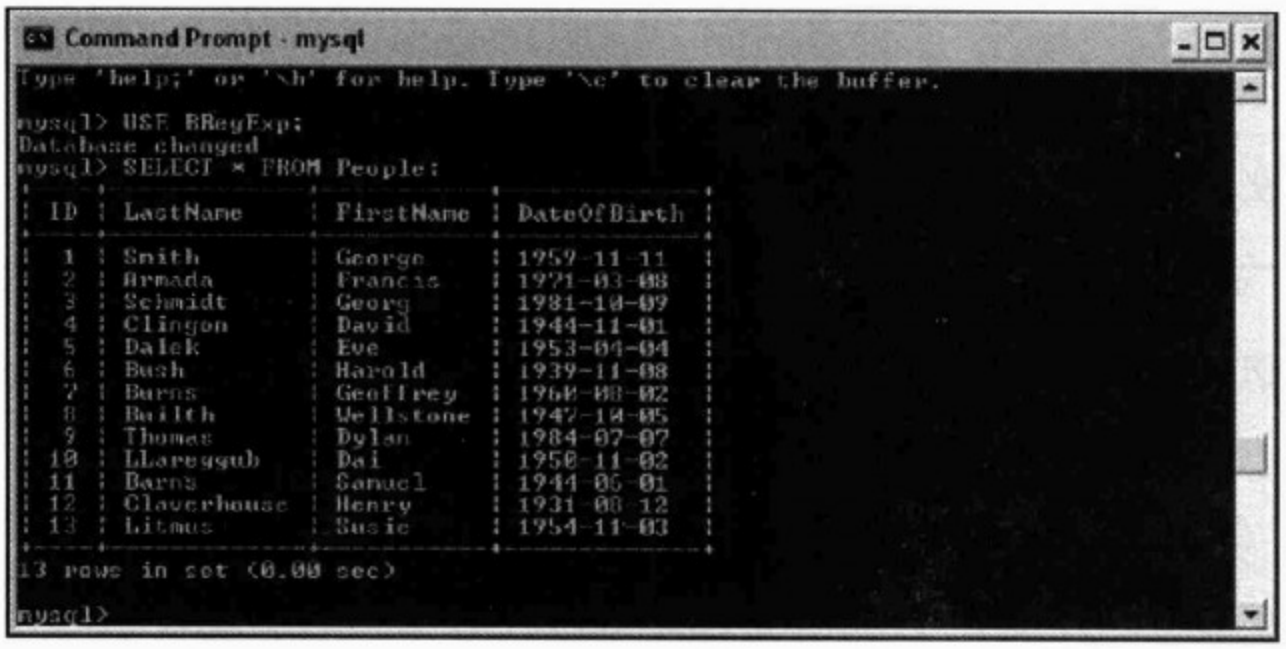


图　17-3

17.2　MySQL 支持的元字符

MySQL 支持很多有用的元字符，其中一些借鉴了 SQL 语法，而另一些使用了正则表达式语法。

表 17-1 和 17-2 总结了 MySQL 4.0 支持的正则表达式功能。表 17-1 列出了与 LIKE 关键字结合使用的 SQL 元字符。表 17-2 中描述了与 REGEXP 关键字结合使用的正则表达式元字符。

下列元字符可以在 SQL WHERE 子句中与 LIKE 关键字一起使用。

表 17-1　与 LIKE 关键字结合使用的 SQL 元字符

元　字　符	说　　明
_	匹配任何单个字符
%	匹配零个或多个字符

下列元字符可以在 WHERE 子句中与 REGEXP 关键字一起使用。

表 17-2　与 REGEXP 关键字结合使用的元字符

元　字　符	说　　明
^	匹配字段(列)开始位置的元字符
$	匹配字段(列)结束位置的元字符

(续表)

元 字 符	说 明
[...]	字符类。同时也支持范围
[^...]	取反的字符类
?	限定符。表示前面的字符或组是可选的
*	限定符。表示匹配前面的字符或组零次或多次
+	限定符。表示匹配前面的字符或组一次或多次
{n,m}	限定符。表示匹配前面的字符或至少 n 次，最多 m 次
|	交替选择。| 元字符分隔互斥的选项

MySQL 4.0 不支持向前查找、向后查找和反向引用。

17.2.1 使用 _ 和 % 元字符

_ 和 % 是 SQL 元字符。它们在 WHERE 子句中与 LIKE 关键字连用。_ 元字符匹配一个单独的字符，因而与标准正则表达式语法中的句点元字符含义类似。而 % 元字符匹配零个或多个字符，等价于标准正则表达式语法中的 .*。

在 MySQL 中，使用 _ 和 % 元字符匹配是不区分大小写的。

试一试：使用 _ 和 % 元字符

下面步骤中的指示假设已经打开命令行提示符窗口，而且当前目录为 MySQL 安装目录中的 bin 目录。

(1) 启动 mysql 实用程序，在 mysql 命令行中执行下面的命令以切换到 BRegExp 数据库：

```
USE BRegExp;
```

首先，使用 _ 元字符选择姓(LastName)字段中以 B 开头，后跟任意单个字符和字符序列 rns 的记录。

(2) 在 mysql 命令行提示符中执行下列命令：

```
SELECT LastName, FirstName
FROM People
WHERE LastName LIKE 'B_rns'
;
```

图 17-4 显示的是这一步之后的结果。

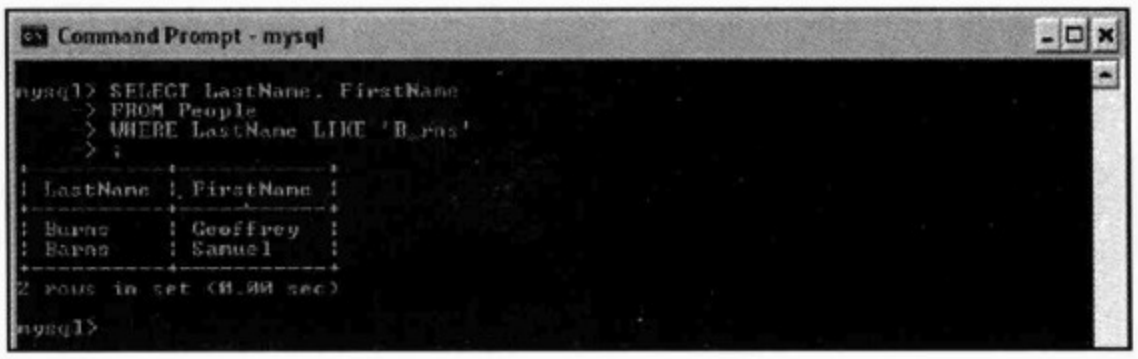


图　17-4

(3) 使用 _ 元字符时也隐含地使用了字段开始元字符和结束元字符。如果从模式中删除最后的 s 就能够验证这一点。在 mysql 命令行提示符中执行下列命令：

```
SELECT LastName, FirstName
FROM People
WHERE LastName LIKE 'B_rn'
;
```

如图 17-5 所示，现在的结果是空记录集。换句话说，没有匹配项。这说明 MySQL 正则表达式引擎将模式 B_rn 当成 ^B_rn$。

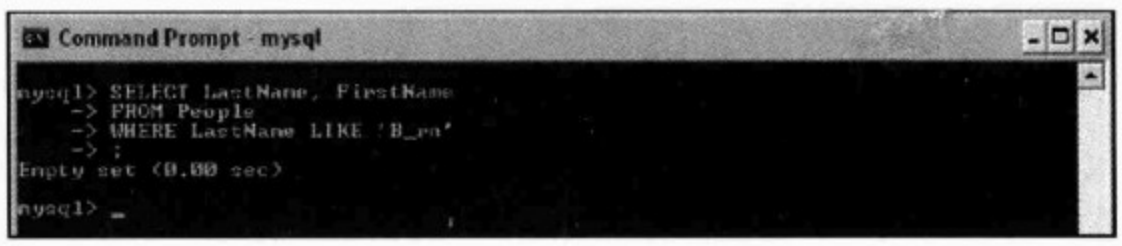


图　17-5

(4) 通过 % 元字符可以找到姓以字符序列 Cl 开头的人的记录。在 mysql 命令行提示符中执行下列命令：

```
SELECT LastName, FirstName
FROM People
WHERE LastName LIKE 'Cl%'
;
```

如图 17-6 所示，只返回了姓氏以字符序列 Cl 开头的记录。

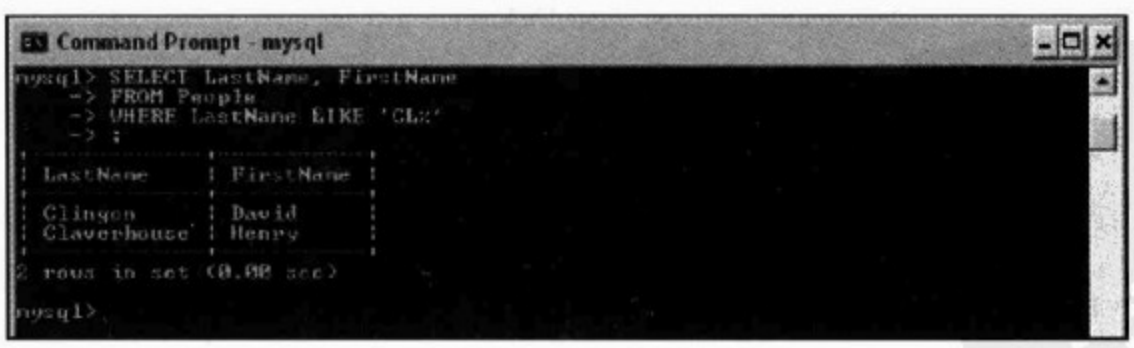


图　17-6

(5) 也可以用 % 元字符匹配 DATE 类型的字段值。比如模式 195%就可以匹配对应的年份。此时，ORDER BY 子句可以使记录按日期的顺序返回。

在 mysql 命令行提示符中输入下列命令：

```
SELECT FirstName, LastName, DateOfBirth
FROM People
WHERE DateOfBirth LIKE '195%'
ORDER BY DateOfBirth
;
```

(6) 观察如图 17-7 所示的结果。其中只显示出生日期以字符序列 195 开头的人的记录。

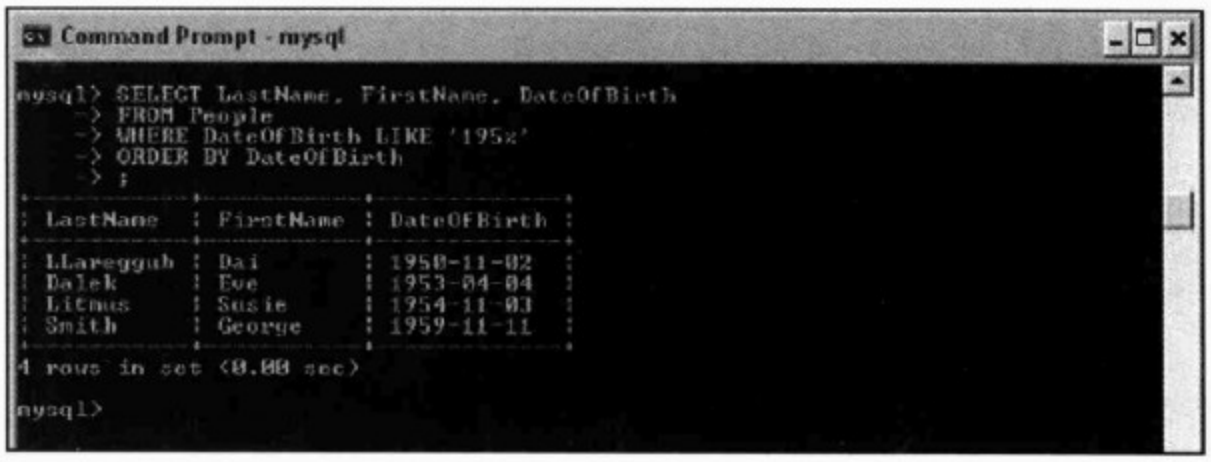


图 17-7

17.2.2 直接量测试匹配：_ 和 % 元字符

与使用 LIKE 关键字和 _ 或 % 元字符匹配保存在数据库中的数据相同，也可以直接测试一个字符序列与一个模式是否匹配。这样就可以测试想找的字符串是否与构建的模式匹配。

相应的语法如下：

```
SELECT "TheString" LIKE "The Pattern";
```

字符串和模式的定界符可以是前面所用的一对双引号，也可以是下面所示的一对单引号：

```
SELECT 'TheString' LIKE 'The Pattern';
```

试一试：选择匹配的直接量

检查模式 Fr% 是否匹配字符序列 Fred。

(1) 在 mysql 命令行中输入下列命令：

```
SELECT "Fred" LIKE "Fr%";
```

(2) 在 mysql 命令行中再输入下列命令：

```
SELECT "Bid" LIKE "B_d"
```

(3) 观察结果。图 17-8 显示的是第 1 步和第 2 步之后的结果。结果中的数字 1 表示存在一个匹配项，与布尔值 True 对应。

图　17-8

如果匹配不成功，结果中会显示数字 0。

(4) 在 mysql 命令行中输入下列命令：

```
SELECT "Fred" LIKE "B_d";
```

(5) 观察结果。根据匹配失败后返回的数字 0，可以确定字符序列 Fred 与模式 B_d 不匹配。如果希望测试的正则表达式找到匹配项，还需要对该正则表达式加以改进。

17.3　使用 REGEXP 关键字和元字符

MySQL 为应用正则表达式功能提供了另一个关键字——REGEXP 关键字。与 LIKE 关键字类似，REGEXP 关键字在 SELECT 语句的 WHERE 子句中使用。

关键字 RLIKE 是 REGEXP 的一个别名。

为完成本节的例子，需扩充前面创建的 People 表并创建一个新的 Employees 表，该表中包含 SSN、Department、Skills 和 Comments 列。创建并迁移 Employees 表的脚本，如下所示：

```
USE BRegExp;
CREATE TABLE Employees
  (ID INT PRIMARY KEY AUTO_INCREMENT,
   LastName VARCHAR(20),
   FirstName VARCHAR(20),
   DateOfBirth DATE,
   SSN VARCHAR(11),
   Department VARCHAR(18),
   Skills VARCHAR(50),
   Comments VARCHAR(100));
 INSERT INTO Employees
  (ID, LastName, FirstName, DateOfBirth, SSN, Department, Skills, Comments)
    VALUES
     (NULL, 'Smith', 'George', '1959-11-11', '123-45-6789', 'Data Management',
'Analysis Services, Business Intelligence, Data Transformation Services', 'Good
skills in SQL Server 2000. Can be grumpy at times.'),
(NULL, 'Armada', 'Francis', '1971-03-08', '881-32-8913', 'Sales', NULL,
'Effective salesman. Particularly good at relating to the business needs of
```

```
    clients.'),
       (NULL, 'Schmidt', 'Georg', '1981-10-09', '456-12-1234', 'Admin', NULL,
    'Effective head of Admin Department. Good communicator.'),
       (NULL, 'Clingon', 'David', '1944-11-01', '234-59-3489', 'Data Management',
'DBA,
    SQL DMO', 'Good database administrator. Lots of experience.'),
       (NULL, 'Dalek', 'Eve', '1953-04-04', '345-19-8822', 'Sales', NULL, 'Good sales
    record. Technically informed.'),
       (NULL, 'Bush', 'Harold', '1939-11-08', '378-12-0021', 'Public Relations', NULL,
    'An old hand. Handled virus crisis excellently last year.'),
       (NULL, 'Burns', 'Geoffrey', '1960-08-02', '000-12-3872', 'Development', 'C#,
    .NET', 'Good .NET programmer. Can lack vision of bigger picture at times.'),
       (NULL, 'Builth', 'Wellstone', '1947-10-05', '009-348-234', 'Development',
    'VB.NET, .NET, ADO.NET', 'Sound. Useful member of team.'),
       (NULL, 'Thomas', 'Dylan', '1984-07-07', '310-23-3891', 'Data Management',
'DTS',
    'Great guy for those data transformation jobs.'),
       (NULL, 'LLareggub', 'Dai', '1950-11-02', '210-23-4578', 'Data Processing',
'Data
    Transformation Services, SQL DMO', 'Good guy. Could be more proactive.'),
       (NULL, 'Barns', 'Samuel', '1944-06-01', '238-12-9999', 'International Sales',
    'Good French and German skills.', 'Good salesman.'),
       (NULL, 'Claverhouse', 'Henry', '1931-08-12', '723-123-234', 'International
    Sales', NULL, 'Semi-retired now. Still effective though.'),
       (NULL, 'Litmus', 'Susie', '1954-11-03', '123-34-4888', 'Admin', 'Good
    organizer.', 'Deputy to Georg Schmidt.');
```

要运行 Employees.sql 脚本，在操作系统命令行中输入以下命令：

```
mysql <C:\BRegExp\Ch17\Employees.sql
```

如果脚本运行成功，提示符中不会显示错误信息。

为了运行试验限定符的例子，需要一个由 Parts sql 脚本创建的 Parts 表，相应的脚本内容如下：

```
USE BRegExp;
CREATE TABLE Parts
 (ID INT PRIMARY KEY AUTO_INCREMENT,
  PartNum VARCHAR(12),
  Description VARCHAR(50));
INSERT INTO Parts
 (ID, PartNum, Description)
  VALUES
    (NULL, 'ABC123', 'A basic widget.'),
    (NULL, 'AAC123', 'A special widget.'),
    (NULL, 'ABBC1234', 'A green widget.'),
    (NULL, 'AAAAAAC2345', 'A purple thing.'),
    (NULL, 'AAAAADD8899', 'A tartan widget'),
    (NULL, 'BC123', 'A thin widget'),
    (NULL, 'ART987', 'An artistic widget'),
```

```
    (NULL, 'XYZ345', 'A recent widget'),
    (NULL, 'AB123', 'A super widget'),
    (NULL, 'AC123', 'An exercise widget'),
    (NULL, 'ABCD234567', 'A long widget'),
    (NULL, 'STUV234', 'A late widget'),
    (NULL, 'VWX7656', 'An automatic widget'),
    (NULL, 'NOP278', 'An opinionated widget'),
    (NULL, 'A2345', 'An numeric widget');
```

在操作系统的命令行提示符窗口中，定位于 MySQL 安装目录中的 bin 目录，执行以下命令：

```
mysql <C:\BRegExp\Ch17\Parts.sql
```

如果运行脚本后没有出现错误提示，则说明可能已经成功地创建了 Parts 表。但为了保险起见，还是有必要测试一下 Employees 和 Parts 这两个表是否确实创建并迁移成功。

运行 mysql 实用程序程序。在 mysql 提示符中执行以下命令切换到包含 Employees 表和 Parts 表的 BRegExp 数据库。

```
USE BRegExp;
```

由于 Employees 表中列数较多，而且 Skills 和 Comments 列的内容也很长，所以分两步显示其中的数据。在 mysql 命令行提示符中执行以下命令：

```
SELECT ID, LastName, FirstName, DateOfBirth
FROM Employees
;
```

如果该表创建成功，那么应该显示包含所有选择列的 13 行记录，包括自动编号的 ID 列，如图 17-9 所示。

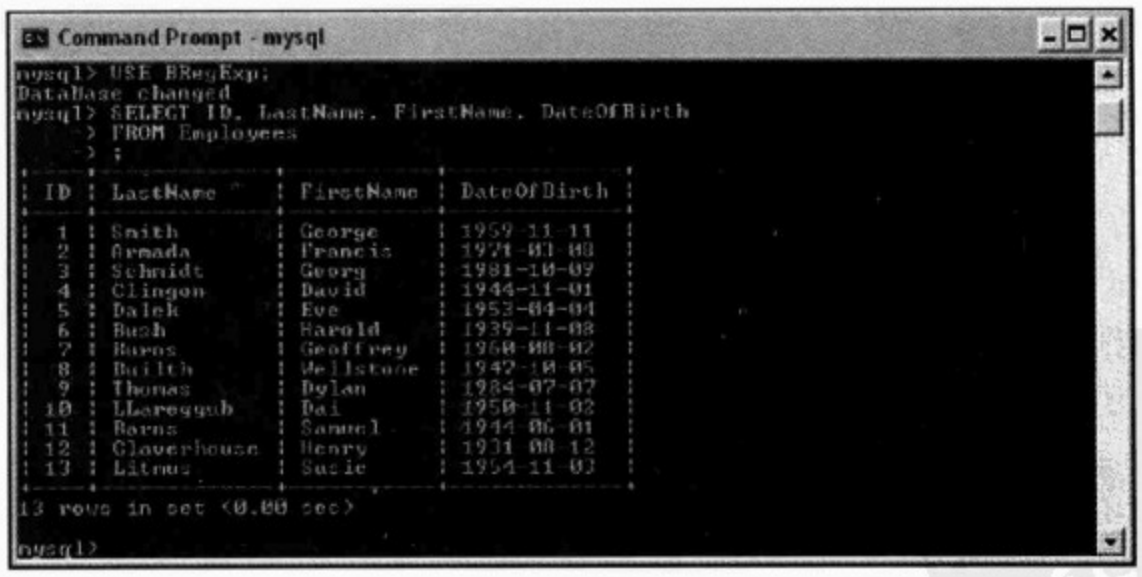


图　17-9

运行下面的命令来测试其他列。首先运行：

```
SELECT ID, SSN, Department, Skills
FROM Employees
```

```
;
```

然后运行：

```
SELECT ID, Comments
FROM Employees
;
```

由于 Skills 列和 Employees 列中的数据内容较长，有些数据可能会在屏幕中折行显示，导致表格较乱。此时，有必要滚动屏幕以确保所有列的数据都成功迁移。

要确定 Parts 表的数据也迁移成功，需要执行以下命令：

```
SELECT *
FROM Parts
;
```

此时没有必要切换数据库，因为刚刚就是在 BRegExp 数据库下测试 Employees 表。图 17-10 显示的是 Parts 表已成功创建并正确迁移。

图 17-10

17.3.1 使用位置元字符

MySQL 既支持匹配字段(列)开始位置的元字符 ^，也支持匹配字段结束位置的元字符 $。如果 WHERE 子句中的模式不包含位置元字符，那么该模式会匹配出现在字段任何位置的相关字符序列。

下面的练习将使用 Employees 表中的数据。

试一试：使用位置元字符

首先，运行不包含位置元字符的代码。

(1) 在 mysql 命令行提示符中执行以下命令：

```
USE BRegExp;
```

(2) 然后再执行以下命令：

```
SELECT ID, LastName, FirstName, Skills
FROM Employees
WHERE Skills REGEXP 'Data'
;
```

结果会显示 Skills 列中包含字符序列 Data 的记录行。图 17-11 显示的是第 2 步之后的结果。结果中包含两条记录：George Smith 和 Dai LLareggub 的记录。

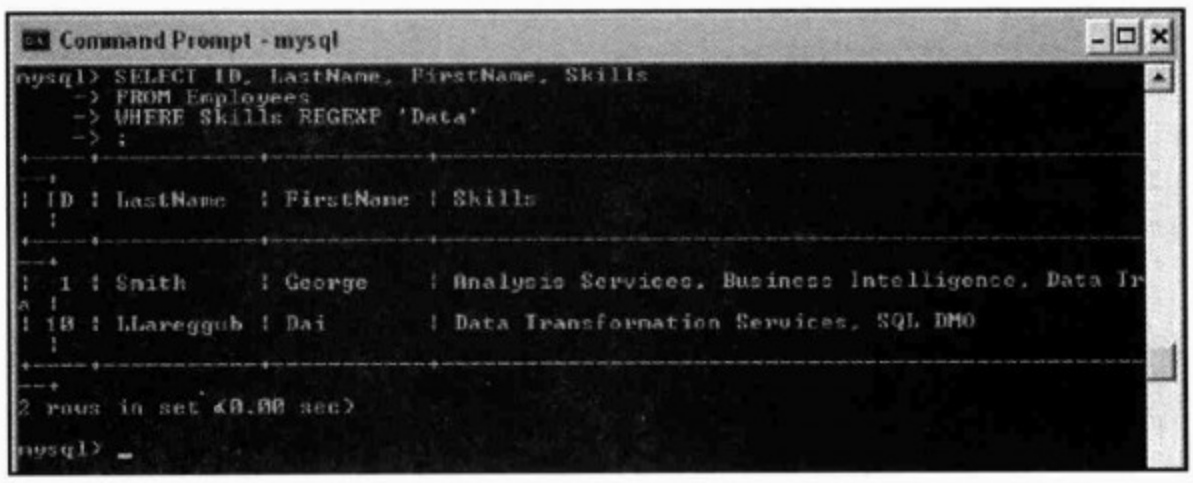


图　17-11

注意，其中只有 Dai LLareggub 记录中的 Data 位于字段的开始位置。所以，如果在模式中加入位置元字符 ^，将只会显示 Dai LLareggub 的记录。

(3) 在 mysql 命令行提示符中执行以下 SQL 命令：

```
SELECT ID, LastName, FirstName, Skills
FROM Employees
WHERE Skills REGEXP '^Data'
```

注意，图 17-12 中只显示出了一条数据——Dai LLareggub 的记录。由于 ^ 元字符指定的字段开始位置后面不是字符序列 Data，所以 George Smith 的记录就不会显示了。

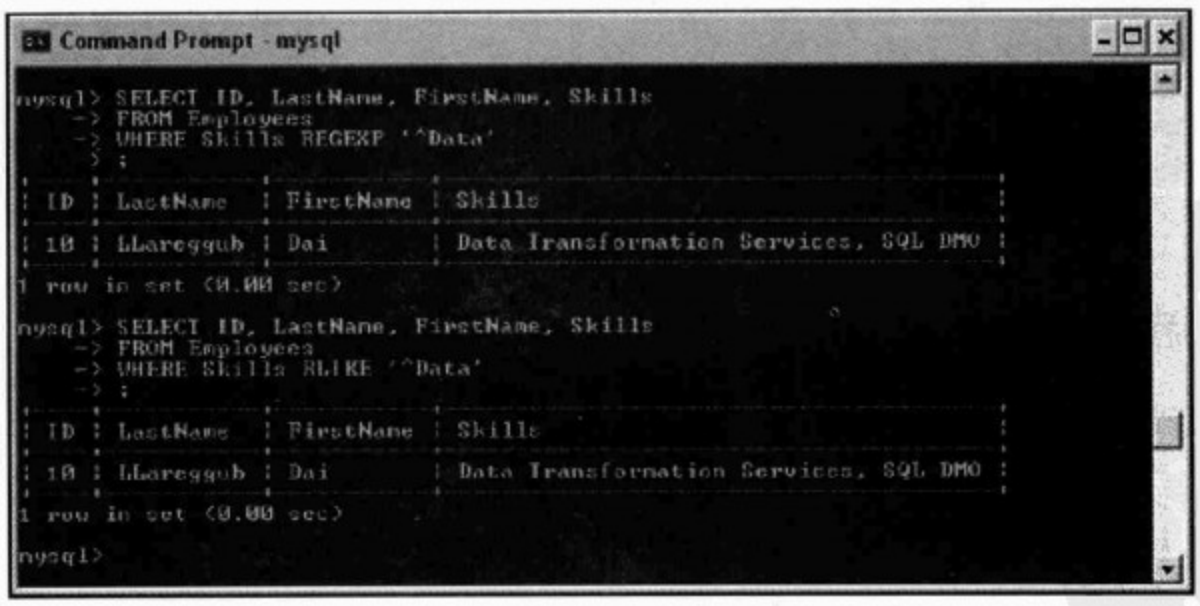


图　17-12

(4) 接着，用与 REGEXP 关键字相同的方式来演示 RLIKE 关键字。在 mysql 命令行提

示符中输入以下命令：

```
SELECT ID, LastName, FirstName, Skills
FROM Employees
WHERE Skills RLIKE '^Data'
;
```

第 4 步的结果也如图 17-12 所示。

(5) $ 元字符的作用是匹配字段(列)结束的位置。首先在没有 $ 元字符的情况下匹配字符序列 Tra。也就是说，匹配 Skills 列中包含处于任意位置的字符序列 Tra 的行。

在 mysql 命令行提示符中执行以下命令：

```
SELECT ID, LastName, FirstName, Skills
FROM Employees
WHERE Skills REGEXP 'Tra'
;
```

结果返回两条记录，分别是 George Smith 和 Dai LLareggub 的记录。当在模式中使用 $ 元字符时，将只剩下一条——George Smith 的记录。

(6) 在 mysql 命令行提示符中执行以下命令：

```
SELECT ID, LastName, FirstName, Skills
FROM Employees
WHERE Skills REGEXP 'Tra$'
;
```

由于只有 George Smith 的记录中包含位于字段结尾的字符序列 Tra，所以只显示该条记录，如图 17-13 所示。

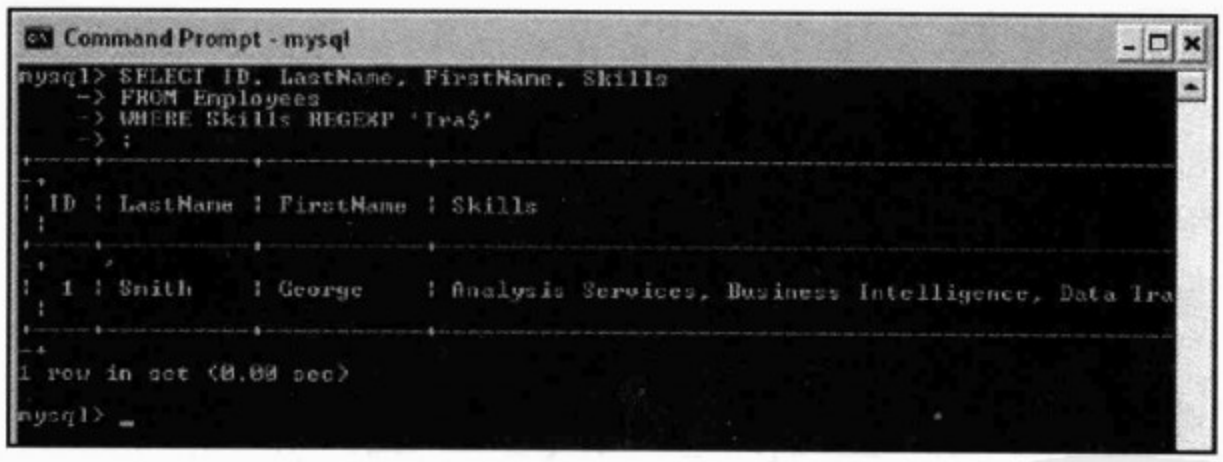


图 17-13

17.3.2 使用字符类

MySQL 支持字符类和取反的字符类，其语法是标准的。比如，模式 [ABCDE] 指定的是包含大写的字母字符 A、B、C、D 和 E 的字符类。字符类 [A-E] 的含义相同，只是在字符类中使用了范围。

试一试：使用字符类

本例使用 Parts 表。

(1) 在 mysql 命令行提示符中输入以下命令：

```
USE BRegExp;
```

(2) 执行以下命令选择以 A 后跟 A、B、C 或 D 开头的零件编号：

```
SELECT ID, PartNum, Description
FROM Parts
WHERE PartNum REGEXP 'A[ABCD]'
;
```

图 17-14 显示的是第 2 步后的结果。结果中包含 8 条记录。每一条记录中的零件编号都是以 A 后跟 A、B、C 或 D 开头的。

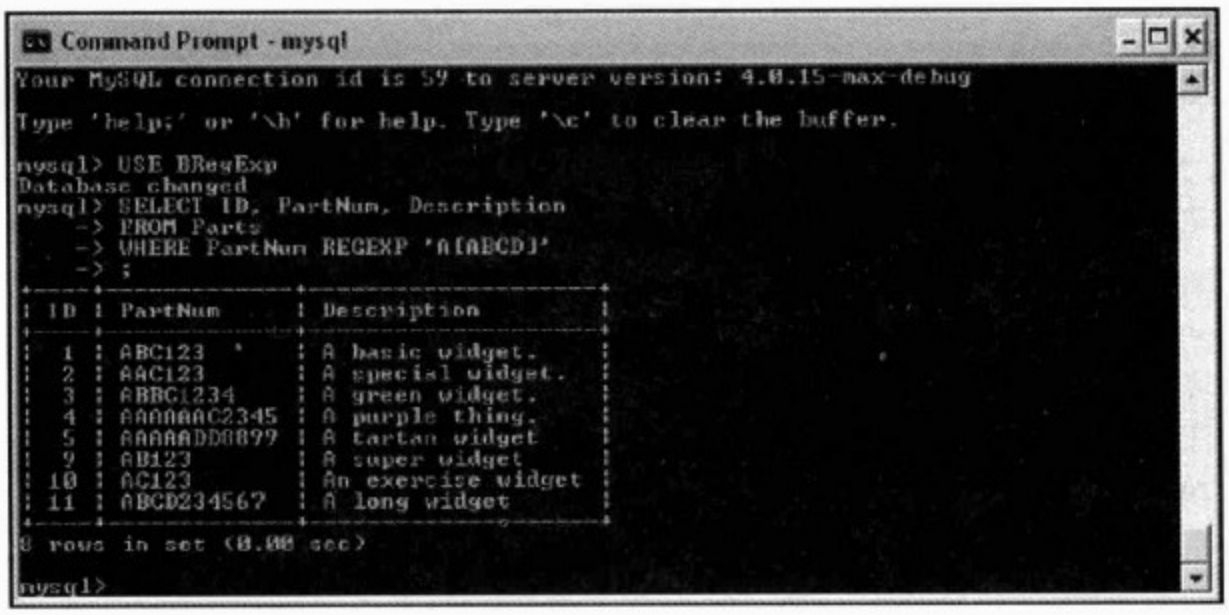


图　17-14

(3) MySQL 支持在字符类中使用范围。所以可以尝试使用带范围的字符类返回与第 2 步相同的结果。

在 mysql 命令行提示符中执行以下命令：

```
SELECT ID, PartNum, Description
FROM Parts
WHERE PartNum REGEXP 'A[A-D]'
;
```

执行以上代码后的结果与图 17-14 所示的结果相同。

(4) MySQL 也支持取反的字符类。因此，可以通过对前面的字符类取反来返回以 A 和不匹配模式 [A-D] 的第二个字母开头的记录。

在 mysql 命令行提示符中执行以下 SQL 命令：

```
SELECT ID, PartNum, Description
FROM Parts
WHERE PartNum REGEXP 'A[^A-D]'
;
```

(5) 观察结果，并与上面字符类返回的结果进行比较。图 17-15 显示的是运行前面代码后的结果。注意，结果中没有出现包含肯定字符类[ABCD]或[A-D]的代码返回的记录。

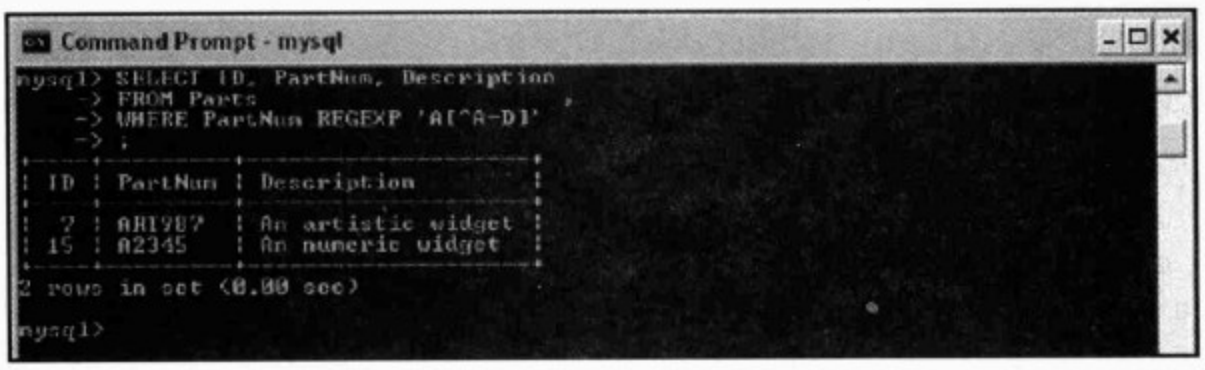


图 17-15

17.3.3 限定符

MySQL 支持多种限定符，包括 ?、 * 、 + 元字符和 {n,m} 表示法。

试一试：在 MySQL 中使用限定符

这个例子使用 Parts 表。

? 元字符表示前面的字符或组是可选的。* 元字符匹配前面零个或多个字符或组。而 + 元字符匹配前面一个或多个字符或组。

(1) 启动 mysql 实用程序，输入以下命令以切换到 BRegExp 数据库：

```
USE BRegExp;
```

(2) 在 mysql 命令行提示符中输入以下命令：

```
SELECT ID, PartNum, Description
FROM Parts
WHERE PartNum REGEXP '^AA?'
```

图 17-16 显示的是第 2 步后的结果。模式 ^AA? 的含义是 A 是第一个字符，后跟一个可选的 A。换句话说，这个模式匹配任何以 A 开头的零件编号。

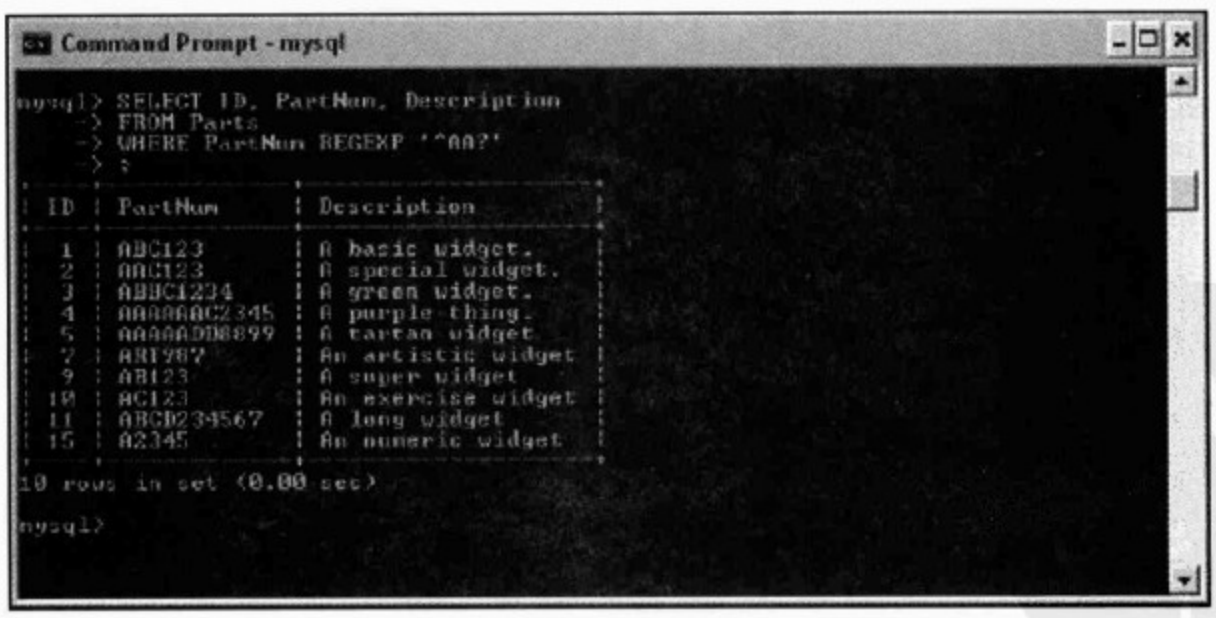


图 17-16

(3) 模式^AA* 会匹配零个或多个第二个 A 的实例。这个模式与模式 ^AA? 会返回相同的记录，因为后一个模式相当于匹配零个或一个的 ^AA*。

在 mysql 命令行提示符中执行以下命令来确认上述推断：

```
SELECT ID, PartNum, Description
FROM Parts
WHERE PartNum REGEXP '^AA*'
;
```

(4) 观察结果，并与图 17-16 的结果进行比较。结果相同。然而，如果把模式修改为 ^AA+ 则是指定了位于初始的 A 后面必须要有一个或多个 A 的实例。所以，零件编号中第二个字母不是 A 的记录将不会再匹配。

(5) 在 mysql 命令行提示符中执行以下命令：

```
SELECT ID, PartNum, Description
FROM Parts
WHERE PartNum REGEXP '^AA+'
;
```

(6) 观察结果，并与第 4 步后的结果进行比较。图 17-17 显示的是第 5 步之后的结果。现在只有三条记录匹配。所有之前匹配、但第二个字母不是 A 的记录没有再匹配。因为模式 ^AA+ 的含义是在初始的 A 后面必须至少还要有一个 A 。

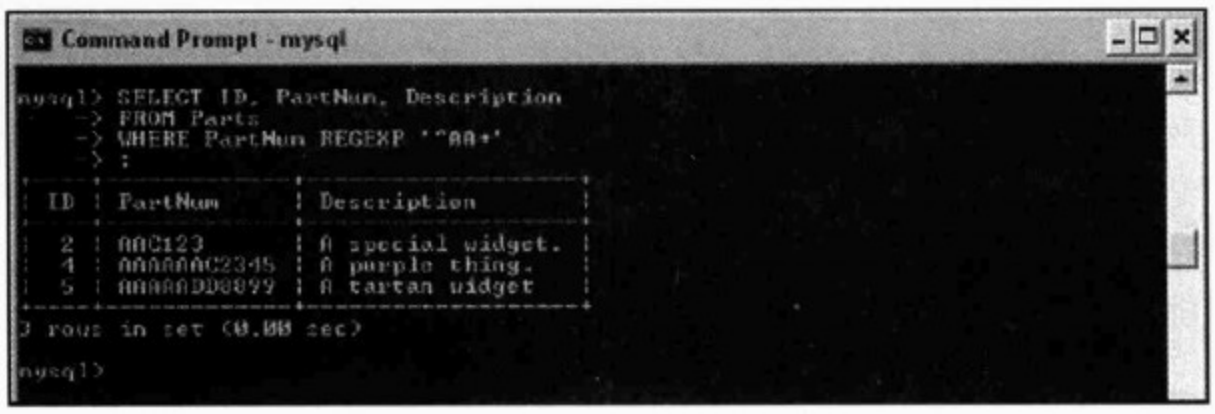


图　17-17

MySQL 还支持完整的 {n,m}　限定符语法。因此，{0,} 与 * 限定符会产生相同的结果，因为它们的含义相同。同样，模式 ^AA{1,} 也会返回与模式 ^AA+ 相同的结果，因为 A{1,} 和 A+ 的含义相同。

(7) 在 MySQL 中即便同时指定 {n,m} 中的 n 和 m 也同样是有效的。作为练习，我们来指定初始的 A 后面跟 1~3 个 A 的实例的模式。在 mysql 命令行提示符中执行以下命令：

```
SELECT ID, PartNum, Description
FROM Parts
WHERE PartNum REGEXP '^AA{1,3}'
;
```

(8) 观察返回的结果，并将其与前面例子中的结果进行比较。

结果中包含三条记录，这与图 17-17 所示的结果相同。这是因为在本例的模式中只指定了要匹配的字符序列而没有指定后续的字符序列。因此，如果在初始的 A 之后有四个字母 A ，限定符中三个的上限依然会匹配。也就是说，存在四个 A 并不影响匹配前三个 A。

17.4 社会保险号的例子

在测试正则表达式时，美国社会保险号(SSN)是一个经典的例子。美国 SSN 的历史很复杂。很多美国人在计算机系统普及之前就拥有了分配给他们的 SSN。

可以匹配 SSN 的一个相对简单的模式如下：

```
[0-9]{3}-[0-9]{2}-[0-9]{4}
```

MySQL 不支持用 \d 元字符匹配数字，因此需要使用字符类 [0-9] 来代替。

可以使用以上模式来查看是否所有员工记录中都包含了有效的社会保险号。

试一试：匹配美国社会保险号码

下面的例子假设启动了 mysql 实用程序并且 BRegExp 是当前数据库。

(1) 在 mysql 命令行提示符中执行以下命令：

```
SELECT ID, LastName, FirstName, SSN
FROM Employees
WHERE SSN REGEXP '^[0-9]{3}-[0-9]{2}-[0-9]{4}$'
;
```

因为我们只想检查整个字段是否包含有效的 SSN，所以需要用 ^ 和 $ 元字符将前面的模式包围起来。

(2) 观察结果中没有显示的 ID 号。图 17-18 显示的结果中每一条记录的 SSN 值都符合三位数字，一个连字符，两位数字，一个连字符和四位数字这样的模式。因此可以确定该模式能够匹配有效的 SSN。

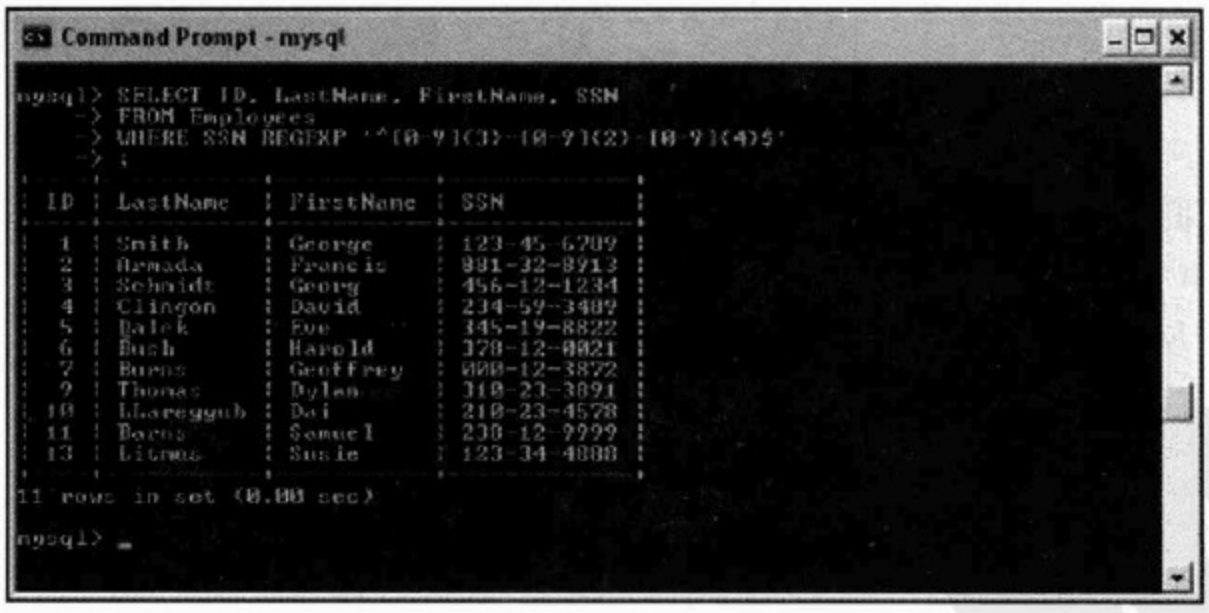


图 17-18

由于不能和 REGEXP 同时使用 NOT 关键字，所以不能直接查找出 SSN 列中包含无效值的记录。

而且，因为 MySQL 不支持向前查找和向后查找，也进一步限制了对 SSN 模式的改进。如果支持向前查找，那么就可以通过 ^(?!000) 这样的模式来限制 SSN 的前三个数字

不能是 000。根据 www.ssa.gov/foia/stateweb.html 中的官方说明，一个有效的 SSN 的前三位数字不可能是 000。如该 URL 中的文档所指出的，将来还会投入使用更多的地区号码(前三位数字)。如果希望创建一个具有高度特殊性的、匹配 SSN 的模式，则需要跟踪该 URL 的更新以保证了解现行的编号规则。

17.5　练习

1. 请编写一行 SQL 代码来检测模式 195% 是否匹配日期 1950-01-01。

2. 请编写几行 SQL 代码来找出具有 .NET 技能的员工。要求显示员工的 ID、姓(LastName)、名(FirstName)以及能力的描述(Skills)。

第18章

正则表达式与 Microsoft Access

经过十多年的发展，Microsoft Access 已经成为小型商务应用和其他个人应用中一种流行的数据库管理系统。

Access 的一项基本工作就是保存用户关心的数据。当数据量较小时，它可以一次返回全部数据使用户通过眼睛来浏览关心的数据。但是，随着数据量的增加，依靠 Access 中的查询功能则会更有效率，也更可靠。Access 中的查询功能是通过一些通配符(类似正则表达式中的元字符)来选择并显示目标数据的。这些通配符有助于过滤由查询检索到的数据，提升 Access 在多种常规的商务数据检索中的价值。

在本章中将学习如下内容：

- Access 支持的元字符
- 如何在 Access 查询中使用元字符
- 如何创建应用 Access 通配符的 select 查询和参数查询

要在 Access 中使用通配符，至少要理解一些相关的内容。因此，本章的内容并非按部就班、平铺直叙。

18.1 Microsoft Access 中元字符的用法

Access 中通配符的首要用途就是在查询过程中匹配文本模式。通配符可以用于需要频繁执行某个特殊查询的硬编码(hard-wired)查询中，也可以用于由用户输入参数并执行通配符搜索的参数查询中。

本章将介绍这两种使用 Access 中不同元字符的方法并结合众多的例子加以解释说明。

本章的例子中使用了一个名为 AuctionPurchases 的测试数据库，它保存着一位图书收藏家从一家名为 dBeach 的虚拟在线拍卖行购书的记录。这些实例中主要关注的是 ItemTitle 字段和 ItemAuthor 字段，这两个字段中分别包含着所购图书的书名和作者信息。其中书名及作者都是真实的。而数据库中的其他数据则是虚构的——包括日期和价格。

本章中的例子已经在 Microsoft Access 2003 中测试通过。而且，例子中截取的屏幕界面也是 Access 2003 的界面。Access 其他版本的界面可能会略有不同。

18.1.1　创建一个硬编码的查询

本例假想查找名字中包含字符序列 Hill 或 hill 的图书作者。由于 Access 中的匹配是不区分大小写的，所以对于直接量匹配而言使用哪种模式都没有问题。

下面步骤中的指示是按照 Access 2003 给出的。更低版本的界面可能存在细微的不同。

试一试：创建一个硬编码的查询

(1) 打开 Access 2003，并从 File 菜单中选择 Open 选项。

(2) 找到 C:\BRegExp\Ch18，选择 AuctionPurchases.mdb 数据库，并单击 OK 按钮。如果下载的测试数据库放在了其他位置，则在相应的目录中查找。

图 18-1 显示的是在第 2 步中单击 OK 按钮后的界面。

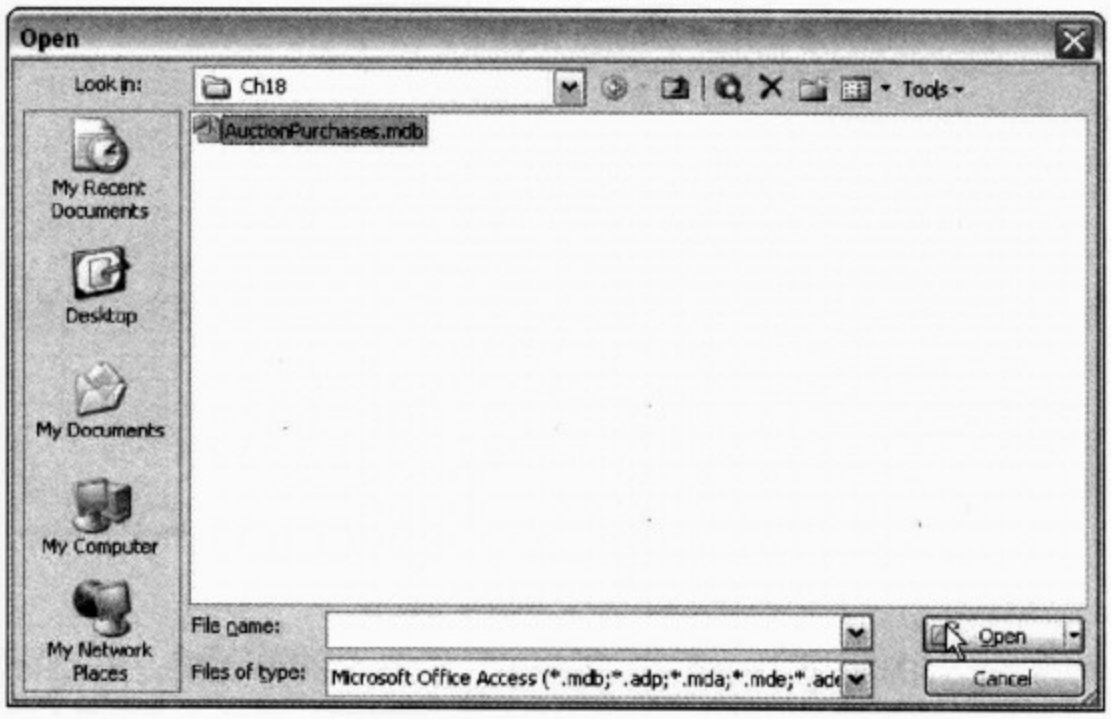


图　18-1

(3) 此时 AuctionPurchases 测试数据库打开。在 Database Object 窗口(如图 18-2 所示)单击左侧面板中的 Queries，在右侧的窗格中会显示 Create query in Design View 和 Create query by using wizard 选项。这里选用 Design View 更合适。

(4) 双击 Create Query in Design View 选项。

(5) 在 Design View 中，会打开 Show Table 对话框(如图 18-3 所示)。选中 dBeach Purchases 表，单击 Add 按钮。

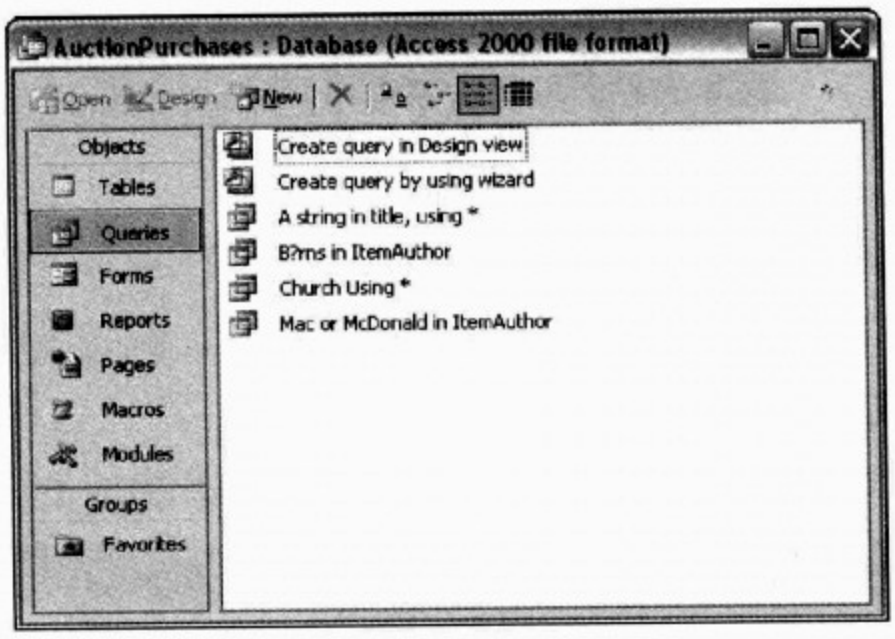


图 18-2

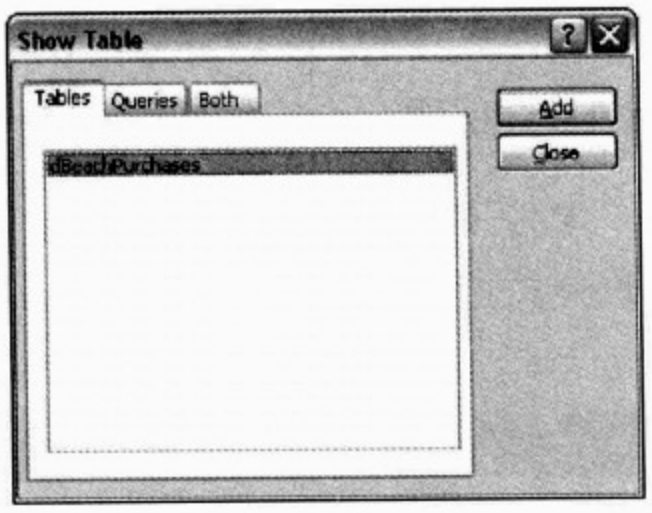


图 18-3

(6) 单击 Close 按钮隐藏 Show Table 对话框。此时，dBeachPurchases 表将显示在如图 18-4 所示的设计窗口的上半部分。在设计窗口的下半部分可以从 dBeachPurchases 表中添加要搜索的列。

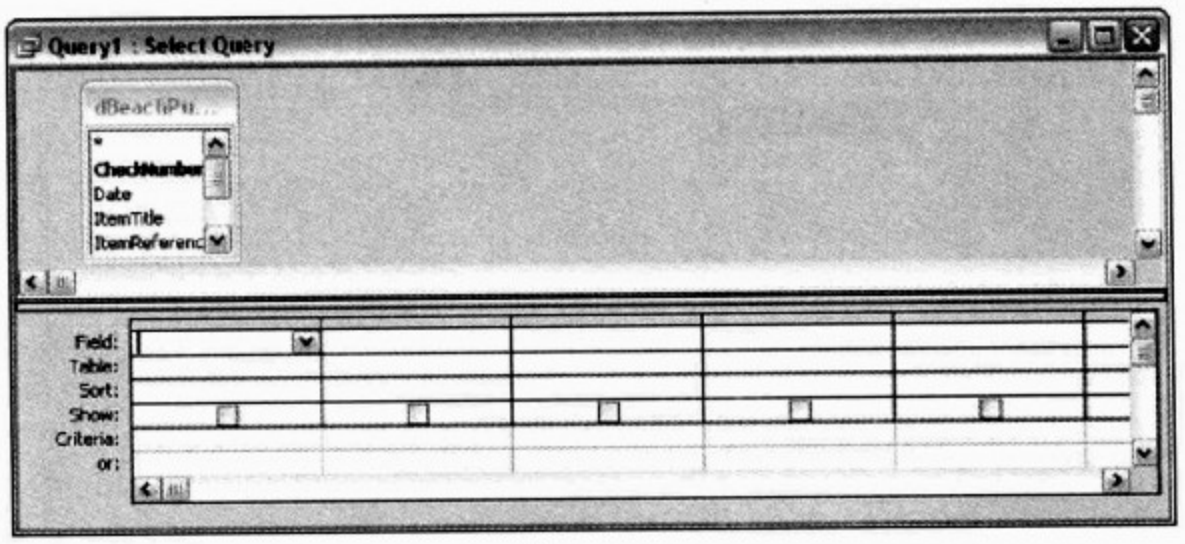


图 18-4

(7) 在设计窗口下方的网格中，单击 Field 行最左端的单元格。在显示的下拉菜单中选择 ItemTitle 字段。

图 18-5 显示的是在第 7 步中调出下拉菜单的界面。

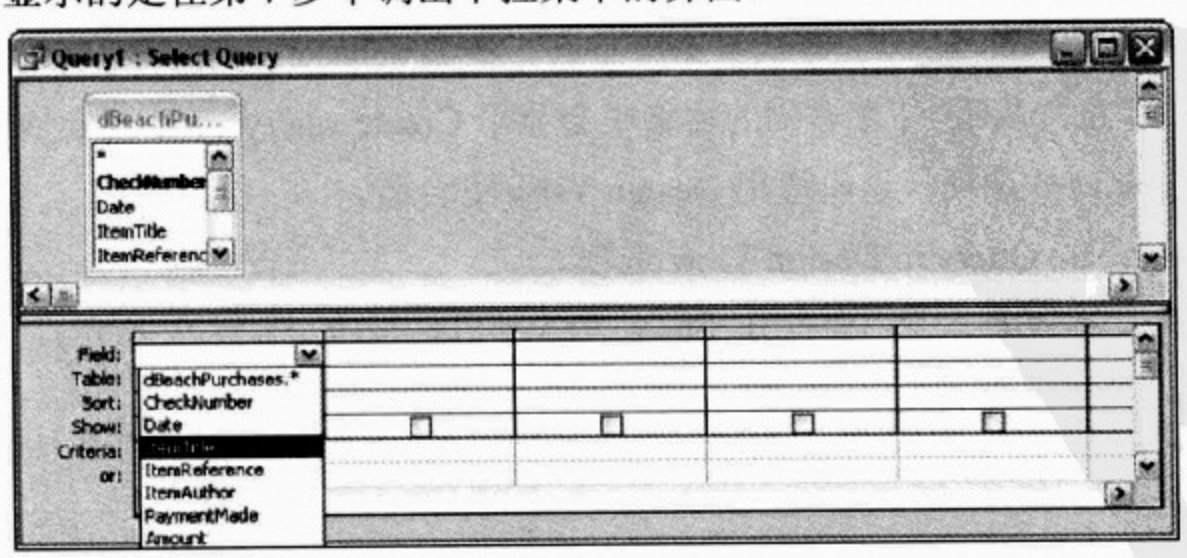


图 18-5

(8) 在下一列中，从下拉菜单中选择 ItemAuthor 字段。图 18-6 显示的是选择 ItemAuthor 时的界面。

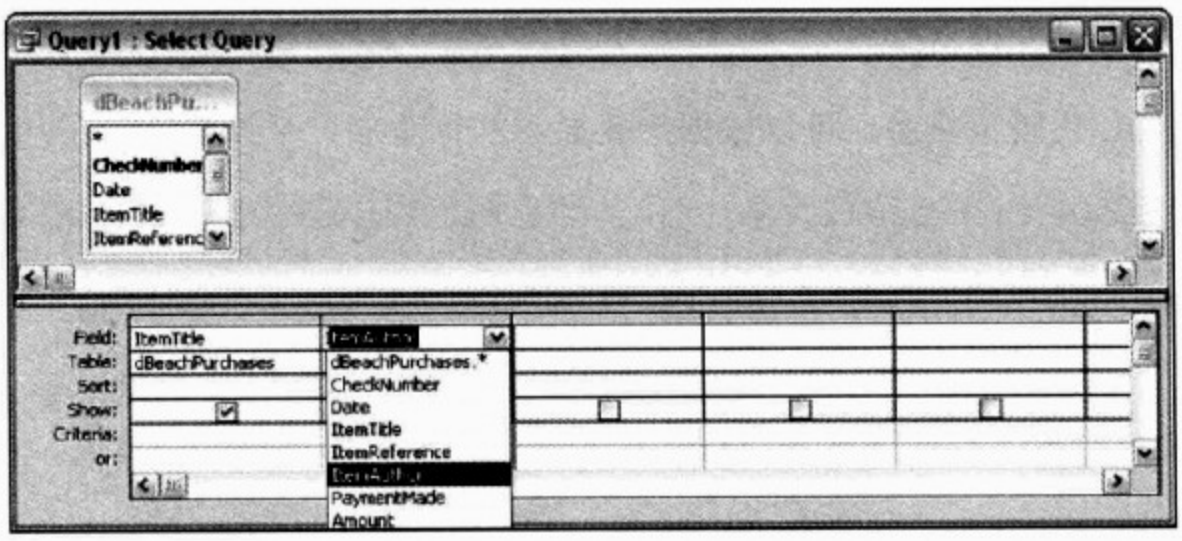


图　18-6

(9) 通过图 18-7 所示的菜单切换到 SQL View(该菜单位于设计窗口的左侧上方)。如果没有发现该菜单，则可能需要显示 Query Design 工具栏。为此，可以打开 View 菜单选择 Toolbars，并选中 Query Design Toolbar 选项。

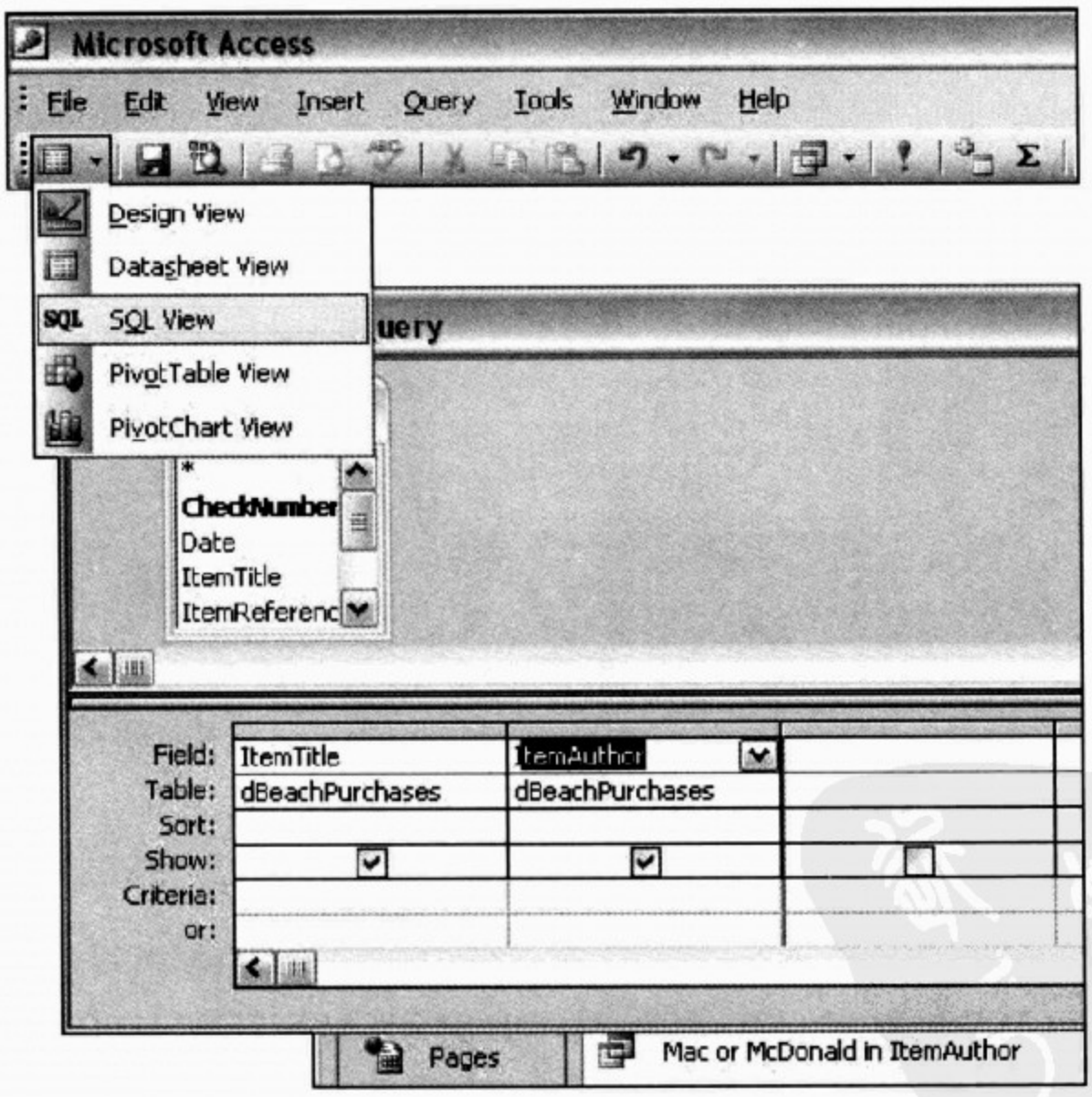


图　18-7

如图 18-8 中所示，已经创建了一个包含简单的 SELECT 语句的 SQL 代码。

在第 7 步和第 8 步中所做的两项选择会生成以下 SQL 代码：

```
SELECT dBeachPurchases.ItemTitle, dBeachPurchases.ItemAuthor
FROM dBeachPurchases;
```

如果对 SQL 代码很熟悉，可以根据想要显示的行添加一个 WHERE 子句，比如：

```
SELECT dBeachPurchases.ItemTitle, dBeachPurchases.ItemAuthor
FROM dBeachPurchases
WHERE dBeachPurchases.ItemTitle LIKE "*Hill*";
```

有时候，在 Access 中编写 SQL 代码需要添加额外的引号或者圆括号。通常这些引号或圆括号都不会影响到查询结果，但偶尔也会显著地影响到返回的结果。

不过，对于本例而言，我们会在设计窗口中添加正则表达式过滤条件。

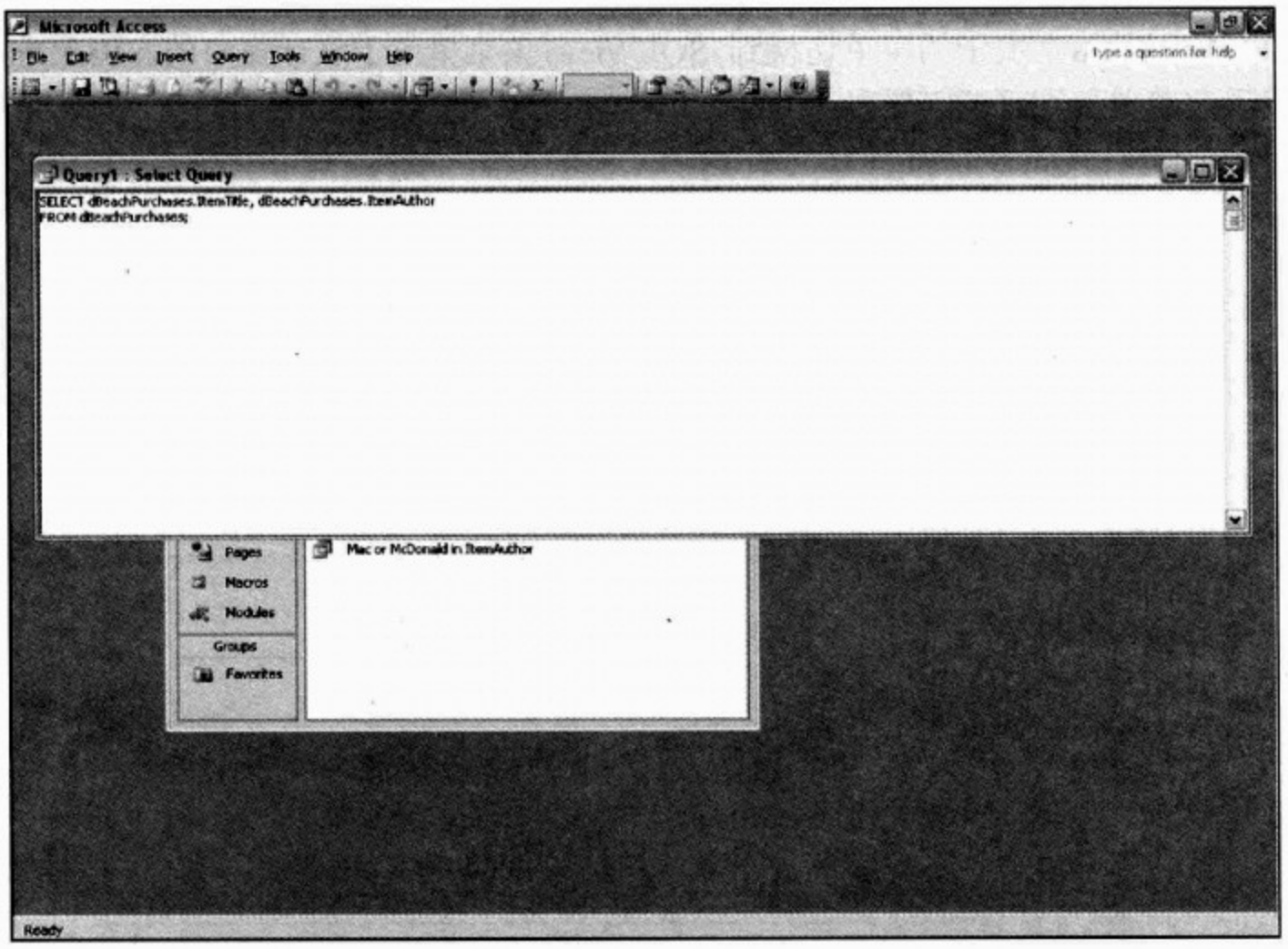


图 18-8

(10) 在设计窗口的 Criteria 行输入下列代码：

```
Like "*Hill*"
```

*元字符匹配零个或多个字符。因此，模式*Hill*会匹配字符序列 Hill(任何大小写组合，因为默认的匹配不区分大小写)之前和之后存在任何数量字符的情况，如图 18-9 所示。

(11) 保存查询，将其命名为 Hill in ItemTitle。

(12) 打开 SQL View，观察由 Access 设计器刚刚创建的代码：

```
SELECT dBeachPurchases.ItemTitle, dBeachPurchases.ItemAuthor
FROM dBeachPurchases
WHERE (((dBeachPurchases.ItemTitle) Like "*Hill*"));
```

注意，Access 添加到代码中的圆括号大多是不必要的。

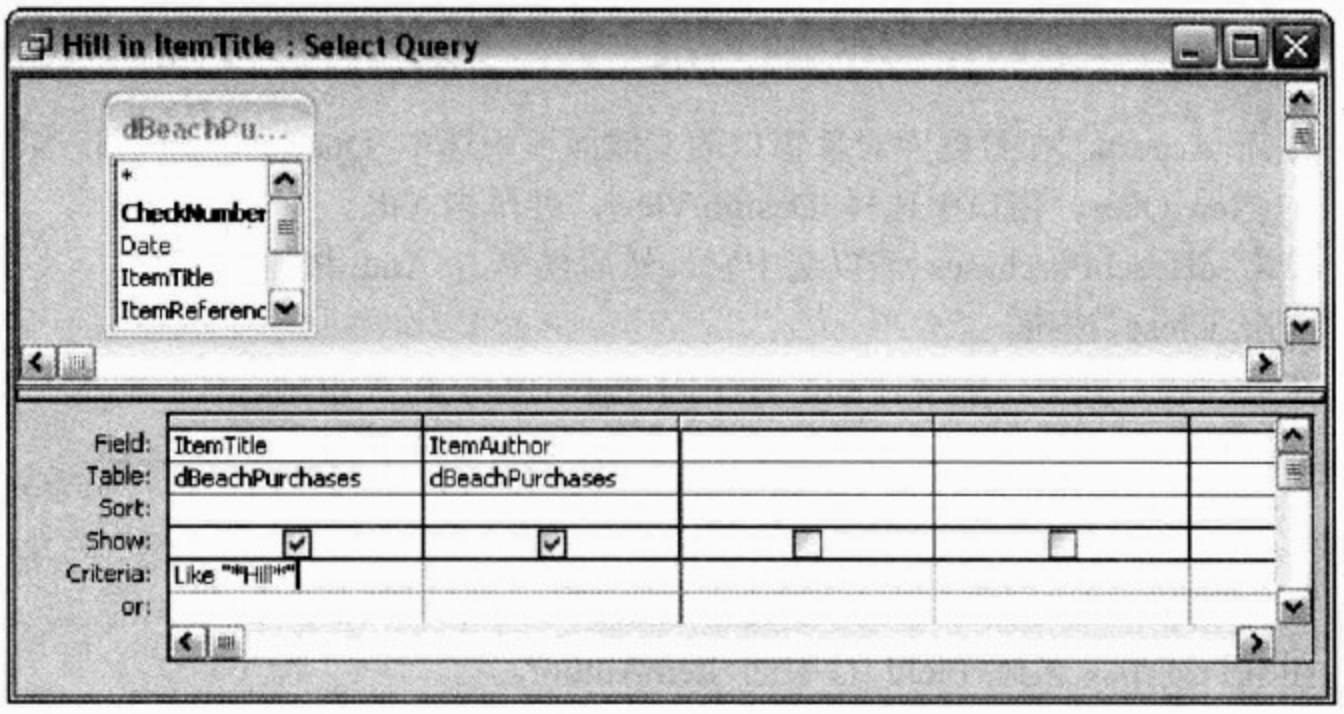


图　18-9

(13) 关闭查询。

(14) 在数据库对象窗口中双击 Hill in ItemTitle。结果如图 18-10 所示。

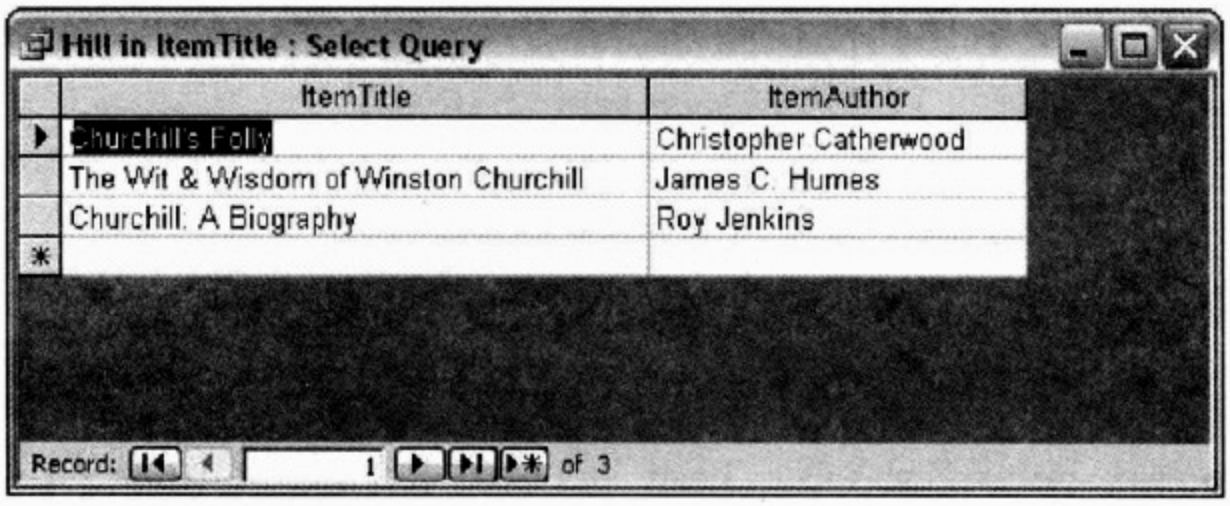


图　18-10

显示的三行记录中都包含字符序列 hill。在测试数据中，每个 hill 都是字符序列 Churchill(两次)或 Churchill's 的一部分。如果通过在 ItemAuthor 列中指定条件 LIKE "*Hill*" 创建一个类似的查询，则会因为 Hill 和 Hills 中包含 Hill 而得到两行记录。可用测试数据库中提供的 Hill in ItemAuthor 查询来验证这一点。

18.1.2　创建一个参数查询

在一些设置中，上一节所介绍的硬编码查询很有用，尤其是在比上面所介绍的简单示例更复杂一点的情况下就更有用了。例如，对特定时期内销售数据的查询可能会被多次使用。

然而，除硬编码正则表达式模式的直接量部分外，还可以让用户来指定要查找的字符序列，使查询更具灵活性。创建参数查询所需的大多数步骤与上一节介绍的创建硬编码的查询所需的步骤是类似的。为此，下面例子对这些类似的地方描述较少。如有必要，可以在创建查询时参考上一节中的详细描述和图示说明。

试一试：创建一个参数查询

(1) 打开 Access，在数据库对象窗口的左侧面板中选中 Queries，并单击 New 按钮。

(2) 在 New Query 窗口中选择 Design View，并单击 OK。

(3) 单击 dBeachPurchases 表以选中它，然后再单击 Add 按钮。

(4) 单击 Close 按钮。

(5) 在最左边一列中，选择 Field 行中的 ItemTitle，在 Criteria 行中输入下列代码：

```
LIKE "*" & [Enter a character sequence to search for:] & "*"
```

注意，不能将以上代码的中间部分包围在一对引号中。换句话说，不要在方括号外面再加引号。如果插入了引号，则无法创建参数查询。

(6) 在下一列中，选择 Field 行中的 ItemAuthor。

(7) 保存查询，并将其命名为 Find a sequence in ItemTitle。然后关闭查询。

(8) 在数据库对象窗口中选择左侧面板中的 Queries，双击 Find a sequence in ItemTitle 查询。

Enter Parameter Value 对话框会打开，如图 18-11 所示。

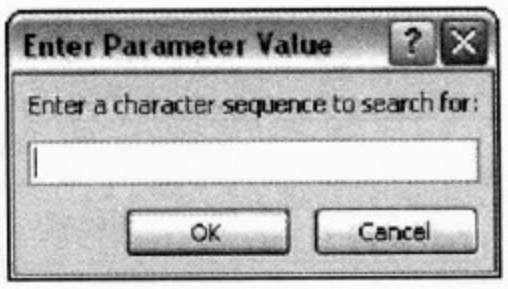


图 18-11

(9) 在 Enter Parameter Value 窗口的文本框中输入 Love，并单击 OK。图 18-12 显示了这一步后的界面。其中书名包含单词 Love 和 Lovers 的记录被显示了出来。

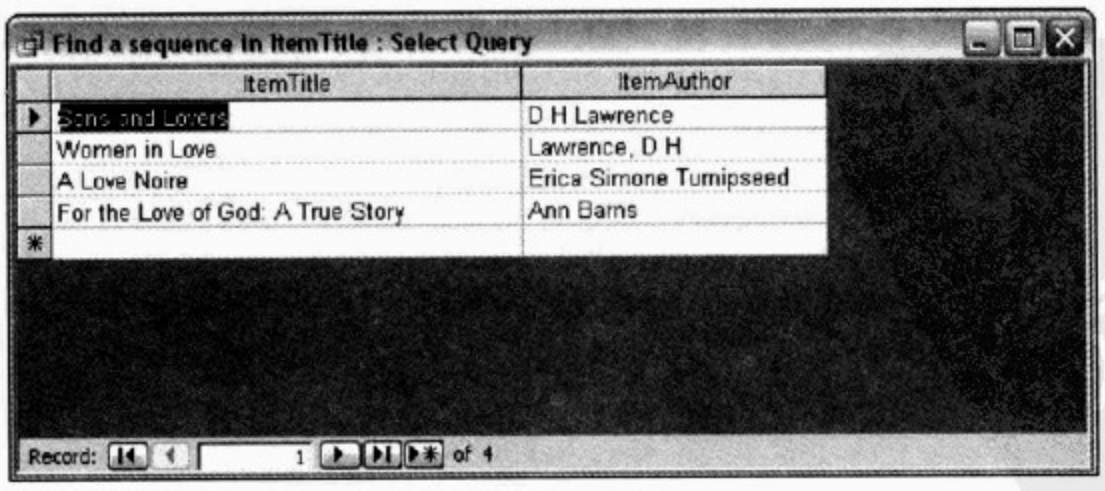


图 18-12

(10) 双击 Find a sequence in ItemTitle 查询，在 Enter Parameter Value 对话框中输入字符序列 Men，并单击 OK。图 18-13 显示的是此时的结果。

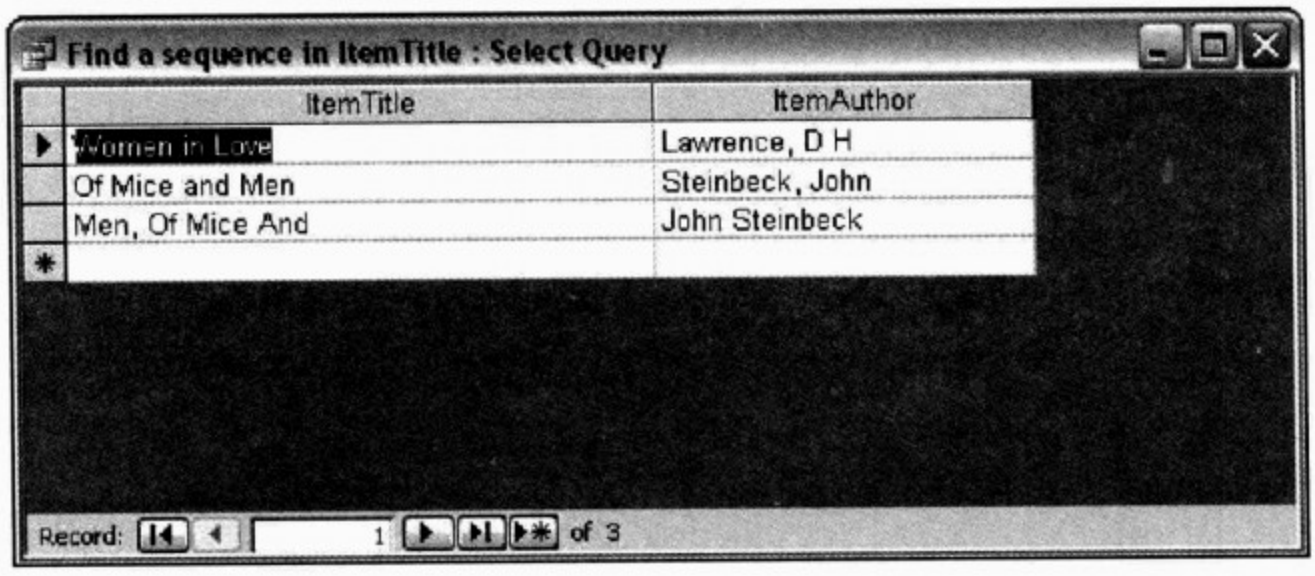


图　18-13

从图中可看到有两条不同的记录，每条记录中的书名都是 Of Mice and Men，只不过书名的记录方式不同。此时的模式 *Men*(后台创建的模式)会匹配位于 ItemTitle 字段中任何位置的字符序列 Men，包括结果记录中位于单词 women 中的 men。

18.2　Access 支持的元字符

与 MySQL 这样的数据库管理系统相比，Access 对正则表达式的支持相当有限。然而，在完成查询时它也提供一些有用的过滤功能。

表 18-1 总结了 Access 2003 支持的元字符。

如果用 ADO 访问 Access 中的数据，可以使用另外一组不同的元字符(本章不做详细介绍)。即使用 % 代替 *，使用 _ 代替 ?。

表 18-1　Access 2003 支持的元字符及其说明

元　字　符	说　　明
?	匹配单个字符
*	匹配零个或多个字符
#	匹配任何单个数字
[...]	匹配方括号中列出的任何单个字符
[! ...]	匹配方括号中未列出的任何单个字符

以下介绍和练习的例子全部使用 AuctionPurchases.mdb 数据库。

18.2.1　使用 ？元字符

？元字符匹配一个单独的字符。

下面的例子将创建一个查询，用于显示作者名中包含 H 后跟一个任意字符，再后跟字符序列 ll 的记录。该查询中使用的模式是 H?ll。

试一试：使用 ？元字符

(1) 打开 Access，在数据库对象窗口左侧的面板中选中 Queries，并单击 New 按钮。
(2) 在 New Query 窗口中，选择 Design View，并单击 OK。
(3) 单击选中 dBeachPurchases 表，然后单击 Add 按钮。
(4) 单击 Close 按钮。
(5) 在最左边一列中，选择 Field 行中的 ItemTitle 字段。
(6) 在下一列中，选择 Field 行中的 ItemAuthor 字段。
(7) 在 Criteria 行中，输入下列代码：

```
Like "*H?ll*"
```

(8) 保存并关闭刚创建的查询，将其命名为 Find H?ll in ItemAuthor。
(9) 双击 Find H?ll in ItemAuthor 运行查询。

图 18-14 显示的是其结果。其中，Heller、Hill、Hills 和 Hall 都与模式 *H?ll* 匹配，因为这些名字中都包含一个 H 字符后跟任意字符(匹配 ？元字符)，再后跟字符序列 ll。模式两边的 * 元字符表示匹配的四个字符的序列两边可以跟任何字符序列，也可以什么都没有。

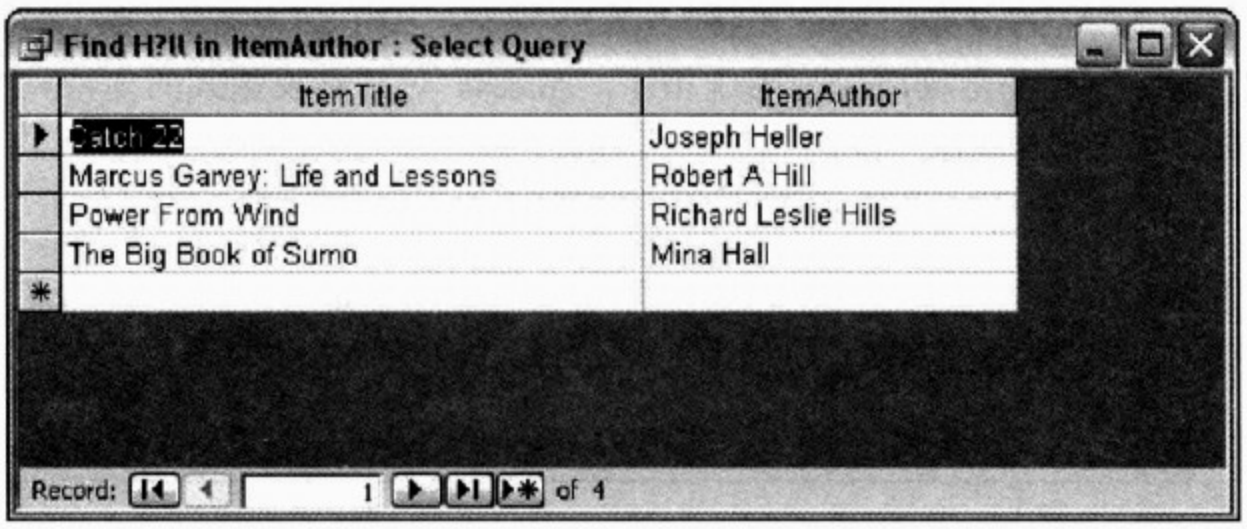


图 18-14

18.2.2 使用 * 元字符

* 元字符匹配零个或多个字符。

查询 Find a sequence in ItemAuthor 是一个参数查询，它允许用户输入一个全部或部分匹配作者名字的字符串。建立这个查询的过程与本章前面详细介绍的创建参数查询 Find a sequence in ItemTitle 的过程非常相似。考虑到篇幅，这里不再逐步介绍其创建过程了。

在设计窗口中左数第二列中，应该选中 ItemAuthor 列，并在 Criteria 行中输入下列代码：

```
Like "*" & [Enter a character sequence:] & "*"
```

这样，当用户在 Enter Parameter Value 对话框的文本框中输入了一个字符序列后，就会动态生成正则表达式模式。因此，如果用户输入字符序列 Hill，则会生成模式 *Hill*。或者，如果用户输入字符序列 Mark，那么生成的模式就是 *Mark*。

双击 Find a sequence in ItemAuthor 查询。在 Enter Parameter Value 对话框的文本框中输入字符序列 Mark，并单击 OK。结果如图 18-15 所示。

返回的每一条记录的 ItemAuthor 列中都包含字符序列 Mark。凑巧的是，这两条结果都包含字符序列 Mark Twain。即模式 *Mark* 中的第一个 * 匹配零个字符，而第二个 * 元字符匹配了空格符后跟字符序列 Twain 。

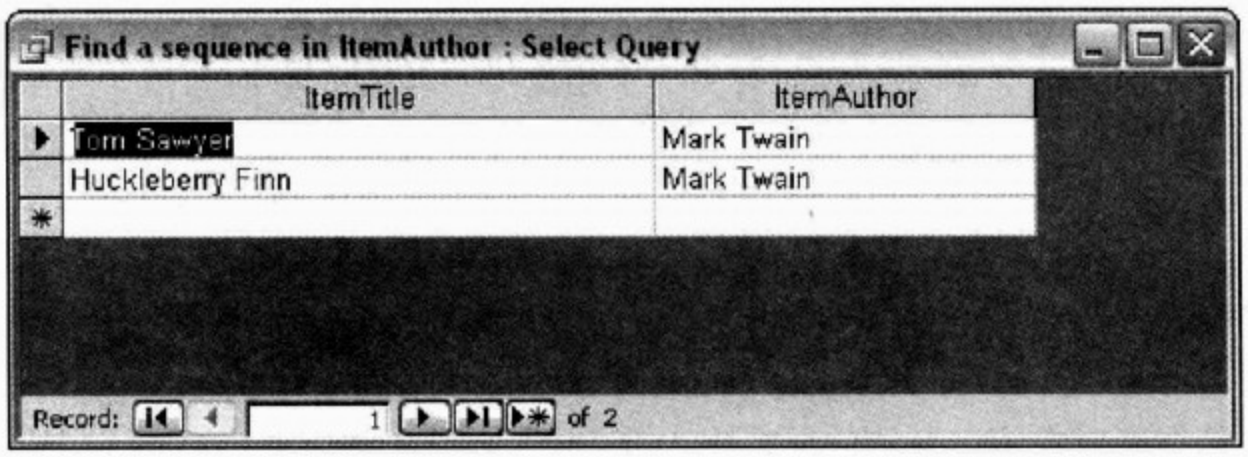


图　18-15

18.3　使用 # 元字符

元字符匹配一个数字，等价于更标准的正则表达式语法中的 \d 元字符。

查询 Number in ItemTitle 示范了 # 元字符的用法。

试一试：匹配一个数字

(1) 打开 Access，并打开 AuctionPurchases.mdb 数据库。

(2) 在数据库对象窗口左侧的面板中，选择 Queries，然后单击 New 按钮。

(3) 选择 dBeachPurchases 表，单击 Add 按钮。然后单击 Close 按钮。

(4) 在设计窗口最左边一列的 Field 行中，选择 ItemTitle。

(5) 在该列的 Criteria 行中输入下列代码：

```
LIKE "*#*"
```

元字符将匹配任何包含一个数字的 ItemTitle 列。

(6) 在下一列的 Field 行中选择 ItemAuthor，然后将查询保存为 Number in ItemTitle。

(7) 在设计视图中关闭这个查询。双击 Number in ItemTitle 查询，并观察如图 18-16 所示的结果。

每个结果的书名中都包含一个或多个数字。由于模式 *#* 中只有一位数字，所以记录中只要有一位数字就可以完成匹配，然后匹配的记录被显示出来。

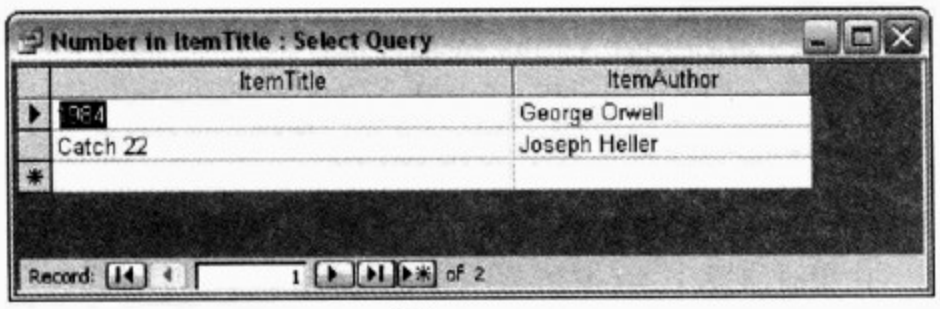


图 18-16

18.4 使用 # 字符匹配日期/时间数据

在 Access 查询中，# 元字符还有另一个用途——匹配日期/时间值。严格来讲，这种情况下并不是将 # 当做正则表达式字符使用。但这种技术对于匹配感兴趣的数据还是很有用的。

试一试：使用 # 字符匹配日期/时间数据

(1) 打开 Access，并打开 AuctionPurchases.mdb 数据库。
(2) 在数据库对象窗口左侧的面板中，选择 Queries，然后单击 New 按钮。
(3) 在 Show Table 对话框中，选择 dBeachPurchases 表，并单击 Add 按钮。
(4) 在网格左边一列的 Field 行中选择 ItemTitle 列。
(5) 在下一列中选择 ItemAuthor 列。
(6) 再在下一列中选择 Date 列。
(7) 在该列的 Criteria 行中输入下列代码：

```
Between #4/1/2003# And #4/30/2003#
```

前面的代码假设要匹配美国式的日期——即，月份后跟日期再后跟年份。Access 就是这样工作的，它只需采取 MM/DD/YY 的格式而不必考虑本地化的设置。

(8) 将查询保存为 April 2003 Purchases，然后在设计视图中关闭该查询。
(9) 在数据库对象窗口中，双击 April 2003 Purchases，并观察结果。

图 18-17 显示的是购买于 2003 年 4 月份的图书列表。从 Date 列中，可以确定只有 2003 年 4 月份的购书记录被显示出来。

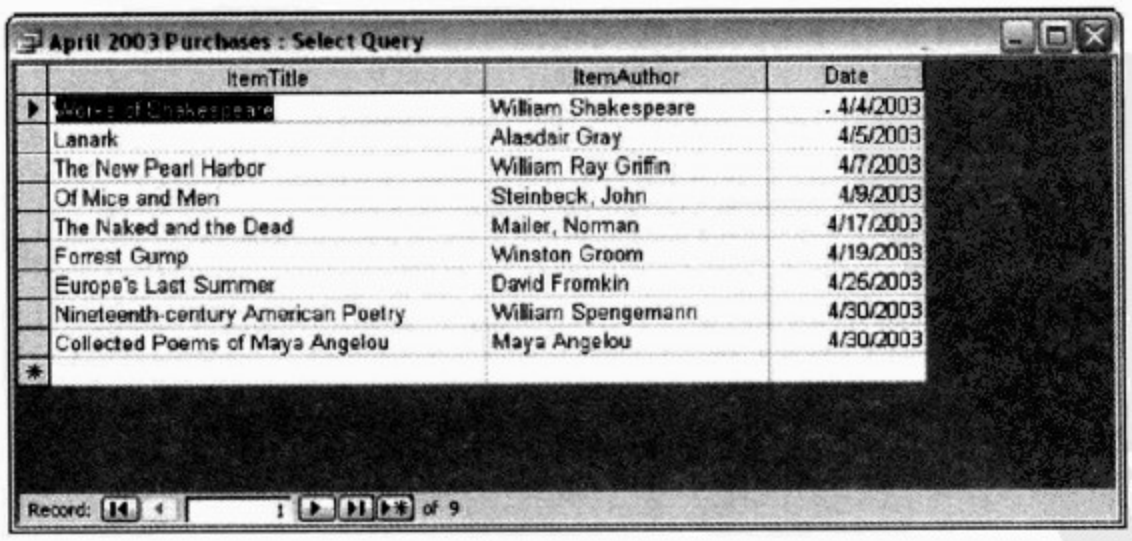


图 18-17

18.5　在 Access 中使用字符类

Microsoft Access 支持字符类，包括范围和对字符类取反。Access 同样也使用以方括号来包含字符类的正规语法。不过，对字符类取反则是以第一个方括号后面的感叹号来表示的。也就是说，如果不想匹配字符 N～Z，那么用取反的字符类来表示就是 [!N-Z]。在表示字符类的方括号外部，感叹号只是一个简单的直接量字符。

本书使用术语字符类来表示包含在方括号中的字符集合。在 Access 中表示字符类时，也可能会用到术语字符列表。

试一试：使用肯定的字符类

(1) 在 Access 中打开 AuctionPurchases.mdb 数据库，并在数据库对象窗口的左侧面板中选择 Queries。

(2) 单击 New 按钮，并从提供的选项中选择 Design View。选择 dBeachPurchases 表。

(3) 在左侧的列中选择 ItemTitle。

(4) 在 Criteria 行中，输入下列代码：

```
LIKE "[A-D]*"
```

(5) 在下一列中，选择 ItemAuthor。

(6) 将查询保存为 Titles Beginning A to D，然后关闭查询。

(7) 在查询面板中双击 Titles Beginning A to D 来运行查询，并观察结果，如图 18-18 所示。每条记录中书名的首字母都位于 A~D 的范围之内。

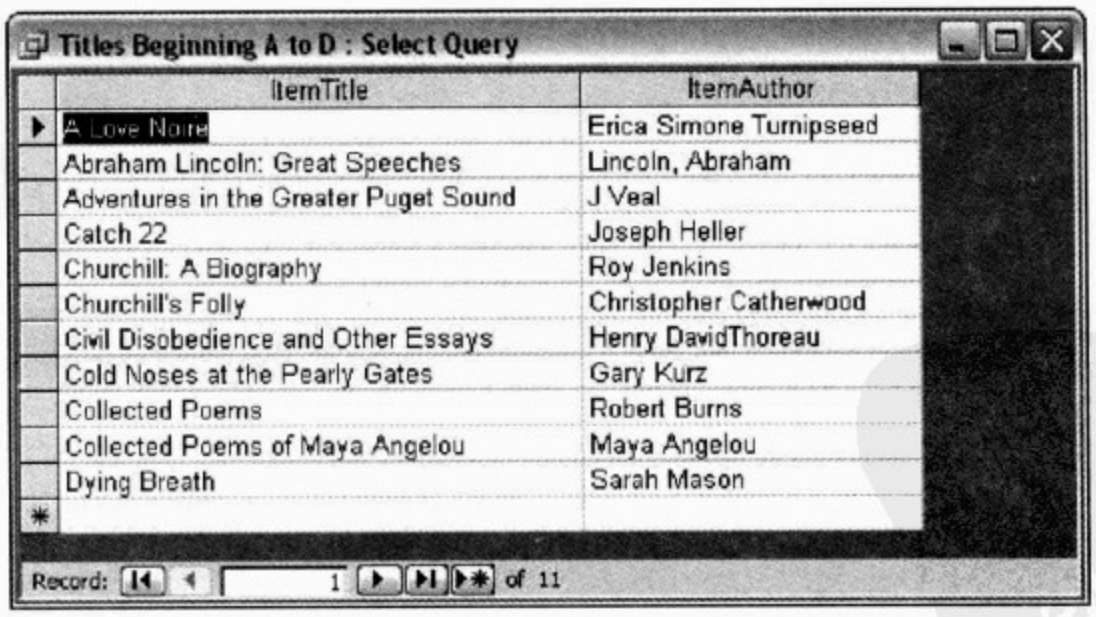


图　18-18

添加一个 ORDER BY 子句更有助于安装结果，从而确定数据以一定的形式组织。

(8) 右击 Titles Beginning A to D，从关联菜单中选择 Design View 选项。

(9) 打开设计视图窗口后，在工具栏的下拉列表中切换到 SQL View。此时的代码如下所示：

```
SELECT dBeachPurchases.ItemTitle, dBeachPurchases.ItemAuthor
FROM dBeachPurchases
WHERE (((dBeachPurchases.ItemTitle) Like "[A-D]*"));
```

(10) 在 SQL 代码结尾的分号之前，插入下列代码：

```
ORDER BY dBeachPurchases.ItemTitle
```

完成的代码应该如下所示：

```
SELECT dBeachPurchases.ItemTitle, dBeachPurchases.ItemAuthor
FROM dBeachPurchases
WHERE (((dBeachPurchases.ItemTitle) Like "[A-D]*"))
ORDER BY dBeachPurchases.ItemTitle;
```

(11) 保存(使用 Ctrl+S)改进后的查询，并关闭查询。

(12) 在数据库对象窗口中，双击 Titles Beginning A to D 查询。如图 18-19 所示，书名按字母顺序排列显示。现在就能更明显地看出只有书名的首字母位于 A～D 之间的记录被显示出来。

Titles Beginning A to D : Select Query

ItemTitle	ItemAuthor
A Love Noire	Erica Simone Turnipseed
Abraham Lincoln: Great Speeches	Lincoln, Abraham
Adventures in the Greater Puget Sound	J Veal
Catch 22	Joseph Heller
Churchill: A Biography	Roy Jenkins
Churchill's Folly	Christopher Catherwood
Civil Disobedience and Other Essays	Henry DavidThoreau
Cold Noses at the Pearly Gates	Gary Kurz
Collected Poems	Robert Burns
Collected Poems of Maya Angelou	Maya Angelou
Dying Breath	Sarah Mason

Record: 1 of 11

图 18-19

试一试：使用取反的字符类

在前面例子的基础上再做下列修改。

在第 4 步中，输入下面的代码：

```
LIKE "[!A-D]*"
```

这样就创建了一个取反的字符类。

在第 6 步中，将查询保存为 Titles NOT beginning A to D。

在随后的几步中，将提到 Titles beginning A to D 的地方，用这个新查询取而代之。

图 18-20 显示的是在运行新查询 Titles NOT beginning A to D 之后的结果。

Titles NOT beginning A to D : Select Query

ItemTitle	ItemAuthor
1984	George Orwell
Europe's Last Summer	David Fromkin
For the Love of God: A True Story	Ann Barns
Forrest Gump	Winston Groom
Gertrude & Claudius	John Updike
Grapes of Wrath	John Steinbeck
Great Expectations	Charles Dickens
Huckleberry Finn	Mark Twain
I Claudius	Robert Graves
Lanark	Alasdair Gray
Lander's Kingdom	Tom Harper
Land's End	Michael Cunningham
Last Breath	Peter Stark
Marcus Garvey: Life and Lessons	Robert A Hill
Men, Of Mice And	John Steinbeck
Narrative of the Life of Frederick Douglass	Frederick Douglass
Nineteenth-century American Poetry	William Spengemann
Of Mice and Men	Steinbeck, John
Pearls in the Shell	J S Soong
Perpetual War for Perpetual Peace	Gore Vidal
Pickwick Papers	Charles Dickens
Powdersmoke Range	William Smoke MacDonald
Power From Wind	Richard Leslie Hills
Pride and Prejudice	Jane Austen
Sense and Sensibility	Jane Austen
Signs of Belonging: Luther's Marks of the Ch	Mary E Hinkle
Sons and Lovers	D H Lawrence
Texas Writers of Today	F E Barns
The Big Book of Sumo	Mina Hall

Record: 1 of 44

图　18-20

18.6　练习

1. 请写出以下查询的 SQL 代码：该查询用于查找 dBeachPurchases 数据库的 ItemAuthor 列中包含 Burns 或 Barns 的记录。

2. 建立一个查询，它能显示 ItemTitle 字段中包含姓 McDonald 或 MacDonald 的记录的 ItemTitle 和 ItemAuthor 字段。

第19章

JScript 和 JavaScript 中的正则表达式

JavaScript 和 JScript 都是实现了 ECMAScript 官方标准(ECMA 262)的语言分支，是继最初的 Netscape JavaScript 之后的新一代脚本语言。

Netscape 和 Mozilla 浏览器使用 JavaScript(当前版本为 1.5)，而 Microsoft Internet Explorer 使用 JScript(当前版本为 5.6)。这两种语言中的大部分功能也存在于其他语言中。而且，它们都提供了可以应用于浏览器环境的正则表达式功能。然而，这两种 ECMAScript 的语言分支之间也存在着某些差别。本章主要着眼于这两种语言分支中共有的功能。此外，在 Internet Explorer 浏览器和 Mozilla 家族浏览器(包括 Firefox)所支持的对象模型间也存在着一些区别。

要运行 JScript 和 JavaScript 代码，必须要有相应的宿主环境来解释代码。当在 Web 浏览器中运行 JScript 和 JavaScript 时，浏览器中的 JScript/JavaScript 解释器会运行相应的脚本代码。而在服务器端，JScript 代码可以通过 Active Server Page(ASP)来运行。另外，JScript 代码也可以通过 Windows Script Host(WSH)中的解释器来运行。

JScript 和 JavaScript 常用于验证在 Web 表单中输入的数据。客户端的表单数据验证比服务器端的验证速度快，而且良好的客户端代码的响应性也会比旧式的验证技术(只能在服务器端完成验证)为用户带来更佳的体验。通常，在现代的表单脚本中，服务器端的验证和处理代码都是作为客户端验证脚本的补充。开发这样双重验证的代码不仅会提高相关的成本，而且也增加了为保证客户端代码与服务器端代码不冲突的维护工作量。

> **W3C 针对 XForms 建立了一个规范，XForms 使用基于 XML 的验证为表单数据提供了更新的验证技术。XForms 使用 W3C XML Schema 进行验证。在本书写作时，XForms 的应用比客户端 JavaScript 或 JScript 代码的应用要少得多。**

在本章中将学习以下内容：

- 如何在 JavaScript 和 JScript 中使用正则表达式
- JavaScript 和 JScript 支持的元字符
- 如何在网页的例子中使用元字符

本章中出现的例子都通过了 Internet Explorer 6.0 和 Mozilla Firefox 0.9.3 中的相应测试。由于所支持对象模型的差异，并不是所有代码都能在 Firefox 0.9.3 中运行。

JScript 与 JScript .NET 并不是同一种语言。后者使用的是 Microsoft .NET Framework，而不是 JavaScript/JScript 解释器。本章不涉及 JScript .NET 的内容。

19.1　在 JavaScript 和 JScript 中使用正则表达式

JScript 和 JavaScript 正则表达式的一个主要用途是处理由终端用户在 Web 表单中输入的信息。

出于安全原因，当在 Web 浏览器中运行 JavaScript 和 JScript 代码时，它们都不能访问保存在本地文件系统中的文件。如果在浏览器中运行的脚本代码能够访问和读写本地文件，那么当这些脚本代码构成网页时，就有可能导致重大的安全威胁。因为在很多情况下，那些脚本的作者都是无法识别、甚至是不应该信任的。事实上，即使对脚本代码的用途做了这些限制，许多 Web 浏览器仍然提供了禁止运行任何脚本代码的选项，以便为有需要的用户提供额外的保护。

如果想运行本章中的例子，那么就需要在选择的 Web 浏览器中至少允许一些网页启用对 JavaScript 代码的支持。本章例子中的指示都是针对当前流行的两种浏览器—— Firefox 0.9.3 和 Internet Explorer 6.0 给出的。

在 Firefox 0.9.3 中可以通过选择 Tools | Options 命令来检查是否启用了对 JavaScript 的支持。在 Options 窗口的左侧面板中，单击 Web Features，此时的屏幕外观将与图 19-1 类似。

如果想运行本章的例子代码，那么要确保选中 Enable JavaScript 复选框。

Internet Explorer 则具有更精细的安全设置。下面的指示假设以本地方式下载并运行测试代码。

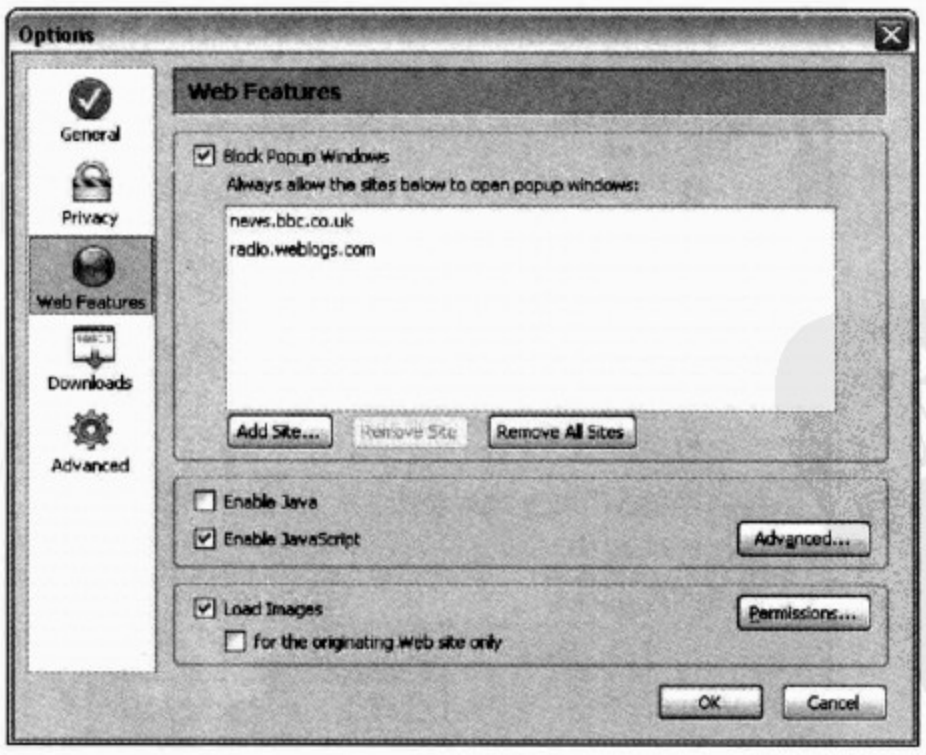


图　19-1

在 Internet Explorer 6 中选择 Tools | Internet Options。在 Internet Options 对话框中，选择 Security 选项卡，并单击上方的 Local Intranet 选项(如图 19-2 所示)。

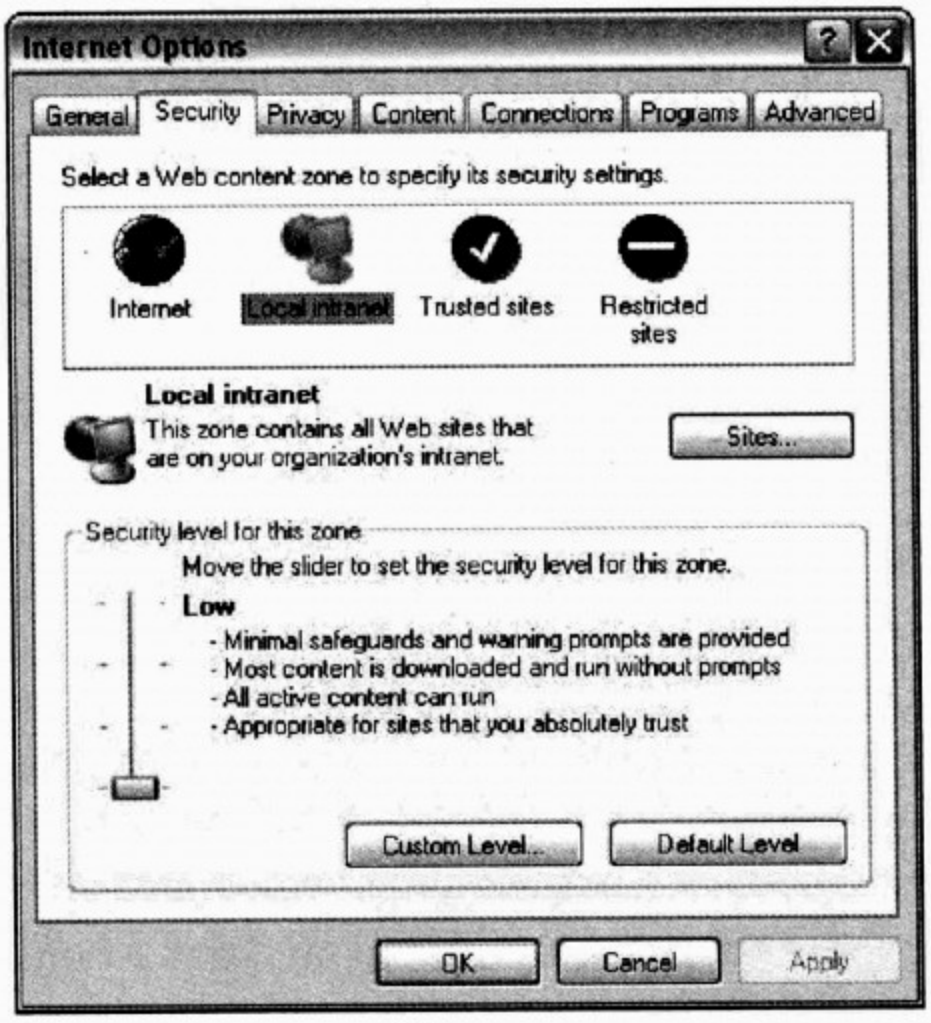


图 19-2

然后，单击 Security 选项卡底部 Custom Level 按钮。Security Settings 窗口打开。在该窗口中向下滚动右侧的滚动条，直到看到如图 19-3 所示的脚本选项。

图 19-3

如果想运行本章的例子，要确定启用如图 19-3 所示的脚本选项。要运行本章的测试代码并不需要启用 Java applets 脚本。

JavaScript 和 JScript 中正则表达式的用途取决于 RegExp 对象和 String 对象的正则表达式功能。这里将完整地介绍 RegExp 对象和 String 对象中与正则表达式有关的内容。

19.1.1　RegExp 对象

可以通过 RegExp 对象的实例来使用模式，而创建这些实例的方式有两种。

- 将一个正则表达式模式指定给一个变量。例如，通过下列代码可以将模式 t$ 指定给变量 myPattern：

```
myPattern = /t$/;
```

- 通过 RegExp 构造函数来创建一个新的 RegExp 对象的实例：

```
myPattern = new RegExp("t$");
```

模式 t$ 将会匹配任何以字符 t 结尾的字符串。

在 JScript 中，可以将一条语句写在一行中而不必使用分号作为语句终止符。然而，在 JavaScript 中一条语句后则必须要有分号(事实上，在 JavaScript 中如果写在一行中的代码是一条完整的语句，也可以省略分号。但 JavaScript 解释器会自动为该行加上分号，因此在 JavaScript 中一行中的语句必须完整。译者注)。因此，考虑到跨平台兼容性的问题，本章中的所有代码都将使用分号作为语句终止符。

下面这两个例子对于终端用户来说是没有区别的。但是，在为变量指定 RegExp 对象时，其中一个例子使用正斜杠语法，而另一个则使用的是 RegExp() 构造函数。而且，一个例子使用 Internet Explorer 的屏幕截图，而另一个则使用 Firefox 的屏幕截图。但对最终用户而言，它们的功能都是相同的。

试一试：使用正斜杠语法创建一个 RegExp 对象的实例

(1) 在文本编辑器中输入以下代码，或者直接打开测试文件 FinalT.html：

```
<html>
<head>
<title>Check for Final t in a string</title>
<script language="javascript" type="text/javascript">
var myRegExp = /t$/;

function Validate(entry){
return myRegExp.test(entry);
} // end function Validate()

function ShowPrompt(){
var entry = prompt("This script tests for matches for the regular expression
pattern: " + myRegExp + ".\nType in a string and click on the OK button.", "Type
your text here.");
```

```
if (Validate(entry)){
alert("There is a match!\nThe regular expression pattern is: " + myRegExp + ".\n
The string that you entered was: '" + entry + "'.");
} // end if
else{
 alert("There is no match in the string you entered.\n" + "The regular expression
pattern is " + myRegExp + "\n" + "You entered the string: '" + entry + "'." );
} // end else

} // end function ShowPrompt()

</script>
</head>
<body>
<form name="myForm">
<br />
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
</form>
</body>
</html>
```

(2) 在 Internet Explorer 中打开 FinalT.html。根据在机器中设置的文件关联程序，只需双击这个文件就可以直接在 Internet Explorer 中打开。也可以在 Windows Explorer 中右击该文件，并通过关联菜单中的 Open with 选项，选择 Internet Explorer。如果在 Open with 菜单中没有 Internet Explorer，那么单击 Choose program 选项，然后根据屏幕提示找到 Internet Explorer。还可以打开 Internet Explorer 并使用 File 菜单的 Open 选项来打开 FinalT.html。图 19-4 显示的是在 Internet Explorer 中打开 FinalT.html 后的屏幕外观。

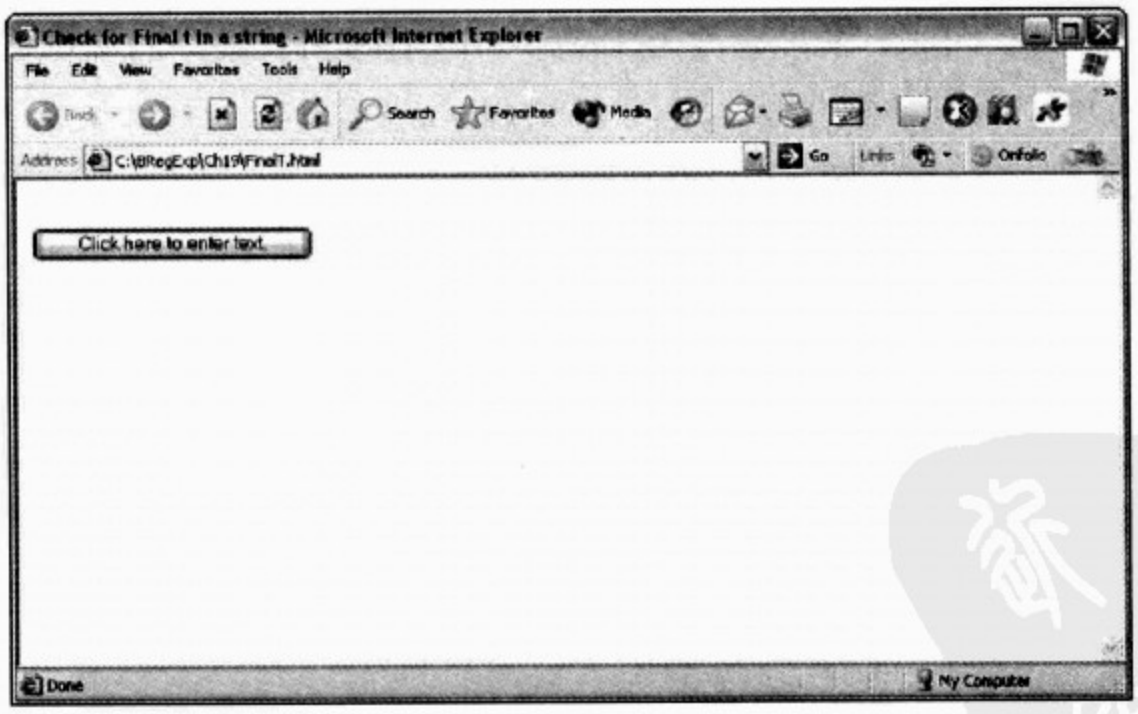


图 19-4

(3) 单击 Click here to enter text 按钮。图 19-5 显示的是此时的屏幕外观。

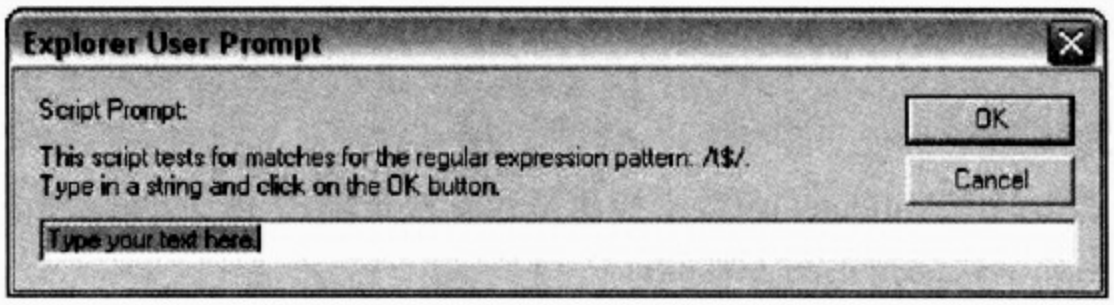


图　19-5

(4) 在文本框中输入字符序列 Test，然后单击 OK 按钮。

图 19-6 显示的是单击 OK 按钮后的屏幕外观。因为测试字符序列 Test 是以 t 结尾，所以它与模式 t$ 匹配。

图　19-6

如果输入的字符序列不匹配，则会显示没有发现匹配项的信息。

(5) 再次单击OK 按钮，图 19-6 中显示信息的对话框消失。然后，单击 Click here to enter text 按钮。

(6) 输入不匹配的字符序列 Tess，并单击 OK 按钮。图 19-7 显示的是提示信息文本框的外观，表明测试字符串不匹配。

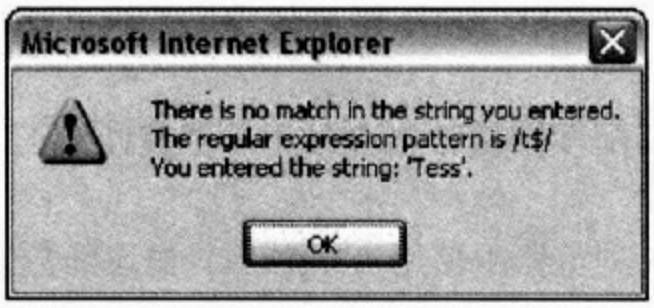


图　19-7

工作原理

首先声明一个全局变量 myRegExp，并给这个变量指定一个由一对正斜杠包围的正则表达式模式 t$：

```
var myRegExp = /t$/;
```

模式 t$ 匹配一个位于用户输入的字符串末尾的字符 t。一对正斜杠是正则表达式模式的定界符，类似于作为字符串定界符的一对双引号。

单击 Click here to enter text 按钮后，会调用 ShowPrompt()函数：

```
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
```

ShowPrompt()函数可以接受用户输入的文本：

```
var entry = prompt("This script tests for matches for the regular expression
pattern: " + myRegExp + ".\nType in a string and click on the OK button.", "Type
your text here.");
```

然后，if 语句调用 Validate()函数来检测是否有符合模式 t$ 的匹配项。如果有匹配项，则会显示一个由如下代码指定的警告框：

```
if (Validate(entry)){
alert("There is a match!\nThe regular expression pattern is: " + myRegExp + ".\n
The string that you entered was: '" + entry + "'.");
} // end if
```

alert() 函数的参数将直接量信息文本与正则表达式模式的值(包含在变量 myRegExp 中)、以及用户输入的值(包含在变量 entry 中)连接起来。

如果没有匹配项，else 子句则会用 alert 警告框显示没有找到匹配项的信息：

```
else{
alert("There is no match in the string you entered.\n" + "The regular expression
pattern is " + myRegExp + "\n" + "You entered the string: '" + entry + "'." );
} // end else
```

Validate() 函数通过下面的代码来测试用户在提示框中输入的文本是否与变量 myRegExp 中的正则表达式匹配：

```
function Validate(entry){
return myRegExp.test(entry);
} // end function Validate()
```

RegExp 对象的 test() 方法以一个字符串作为参数，并测试该字符串中是否存在符合正则表达式模式的匹配项。

如前所述，如果存在匹配项则警告框会显示出存在匹配的信息。如果没有匹配项，则会显示不存在匹配的信息。

下面，我们在 Firefox 浏览器中使用 RegExp() 构造函数来实现同样的验证逻辑。

试一试：使用 RegExp() 构造函数

(1) 打开 Firefox 浏览器(如果没有安装 Firefox，在 Internet Explorer 中也可以运行本例中的代码)。

(2) 在 File 菜单中选择 Open，然后找到文件 FinalTConstructor.html 并打开它。该文件的内容如下：

```
<html>
<head>
<title>Check for Final t in a string</title>
<script language="javascript" type="text/javascript">
var myRegExp = new RegExp("t$");

function Validate(entry){
return myRegExp.test(entry);
```

```
} // end function Validate()

function ShowPrompt(){
var entry = prompt("This script tests for matches for the regular expression
pattern: " + myRegExp + ".\nType in a string and click on the OK button.", "Type
your text here.");
if (Validate(entry)){
alert("There is a match!\nThe regular expression pattern is: " + myRegExp + ".\n
The string that you entered was: '" + entry + "'.");
} // end if
else{
  alert("There is no match in the string you entered.\n" + "The regular expression
pattern is " + myRegExp + "\n" + "You entered the string: '" + entry + "'." );
} // end else

} // end function ShowPrompt()

</script>
</head>
<body>
<form name="myForm">
<br />
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
</form>
</body>
</html>
```

(3) 单击 Click here to enter text 按钮，并输入字符序列 Test。

(4) 单击 OK 按钮并观察显示存在匹配项信息的警告框，如图 19-8 所示。

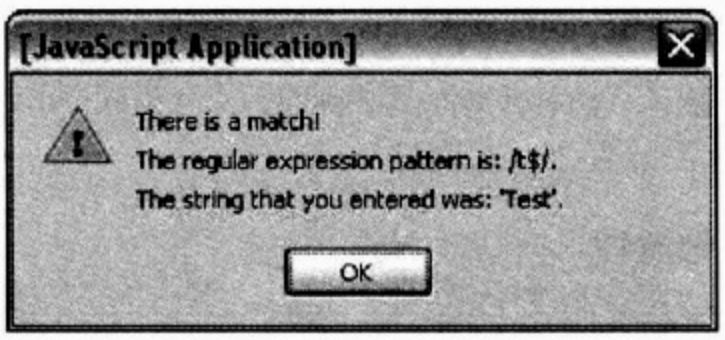


图　19-8

(5) 单击 OK 按钮退出警告框，然后再单击 Click here to enter text 按钮并输入字符序列 Test.。注意在测试字符序列的结尾处有一个句点字符。

(6) 单击 OK 按钮，观察警告框中显示的信息。

工作原理

下面的代码使用 RegExp() 构造函数创建了包含模式 t$ 的正则表达式对象，并将该对象指定给变量 myRegExp：

```
var myRegExp = new RegExp("t$");
```

其余的函数代码如上一个“试一试”部分所述。当用户输入的字符串结尾是一个小写的 t 时，则存在匹配项；否则，不存在匹配项。

个人认为，下面的语法更简洁一些：

```
var myRegExp = /t$/;
```

但以上两种创建正则表达式对象的语法是等价的，如果你更愿意使用 RegExp() 构造函数，那么只需将本章后面例子中使用一对正斜杠的语法替换掉即可。

1. RegExp 对象的属性

下面的代码(在前面看到过)使用 RegExp 对象的默认设置将模式 t$ 指定给变量 myRegExp ：

```
var myRegExp = /t$/;
```

通过这样一个赋值语句的一般形式，也可以附加 RegExp 对象的属性。其一般的形式是：

```
var myVariable = /pattern/attributes
```

其中“模式”是一个正则表达式模式，而“属性”则可能是包含字符 m、g 或 i 的字符串。属性 m 表示多行匹配，属性 i 表示不区分大小写的匹配，而属性 g 表示全局匹配。在多行匹配模式下，^ 和 $ 元字符匹配整个测试文件的开始和结束位置 ——即使该文件分布在多行中。

如果使用 RegExp() 构造函数，那么相应的语法是：

```
var myVariable = new RegExp(/pattern/attributes)
```

RegExp 对象的三个属性也对应着 m、g 和 i。RegExp 对象的 global 属性表示是否指定了 g 属性，RegExp 对象的 ignoreCase 属性则对应所指定的 i 属性。最后，RegExp 对象的 multiline 属性表示是否设置了 m 属性。

2. RegExp 对象的其他属性

除 global、ignoreCase 和 multiline 属性之外，RegExp 对象还有其他属性。

lastIndex 属性表示最后一个匹配项出现的位置。当在一个字符串中找到多个匹配项时常会用到 lastIndex 属性。而 source 属性中保存着正则表达式的源文本——即保存着正则表达式模式。

通过下面的测试文件 NumericDigitsOthersAllowed.html 可以体会一下如何使用刚才提到的属性。NumericDigitsOthersAllowed.html 的代码如下：

```
<html>
<head>
<title>RegExp Object Properties</title>
<script language="javascript" type="text/javascript">
var myRegExp = /\d+/;
```

```
function Validate(entry){
return myRegExp.test(entry);
} // end function Validate()

function ShowPrompt(){
var entry = prompt("This script tests for matches for the regular expression
pattern: " + myRegExp + ".\nType in a string and click on the OK button.", "Type
your text here.");
if (Validate(entry)){
displayString = "";
displayString += "There is a match!\nThe regular expression pattern is: "
+
myRegExp + ".\nThe string that you entered was: '" + entry +"\n";
displayString += "The global property contained: " + myRegExp.global + "\n";
displayString += "The ignoreCase property contained: " + myRegExp.ignoreCase +
"\n";
displayString += "The multiline property contained: " + myRegExp.multiline + "\n";
displayString += "The source property contained: " + myRegExp.source + "\n";
displayString += "The lastIndex property contained: " + myRegExp.lastIndex;
alert(displayString);
} // end if
else{
  alert("There is no match in the string you entered.\n" + "The regular expression
pattern is " + myRegExp + "\n" + "You entered the string: '" + entry + "'." );
} // end else
} // end function ShowPrompt()

</script>
</head>
<body>
<form name="myForm">
<br />
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
</form>
</body>
</html>
```

试一试：探索 RegExp 对象的属性

(1) 在 Internet Explorer 中打开文件 NumericDigitsOthersAllowed.html。

(2) 单击 Click here to enter text 按钮，在文本框中输入文本 Hello 99。

(3) 单击 OK 按钮，观察警告框中显示的信息，如图 19-9 所示。

图 19-9

(4) 单击 OK 按钮退出警告框，然后再单击 Click here to enter text 按钮，在文本框中输入 99 Hello。

(5) 单击 OK 按钮，并观察如图 19-10 所示的警告框中显示的信息。

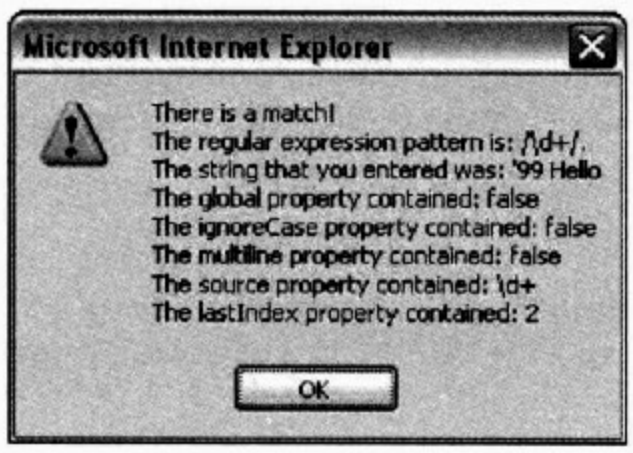


图 19-10

(6) 比较图 19-9(输入的文本是 Hello 99)与图 19-10(输入的文本是 99 Hello)。注意 RegExp 对象在这两种情况下不同的 lastIndex 属性值。

工作原理

变量 myRegExp 是作为一个全局变量声明的：

```
var myRegExp = /\d+/;
```

指定给它的是一个包含模式 \d+ 的 RegExp 对象，该模式匹配一个或多个数字。

当单击 Click here to enter text 按钮时，会调用函数 ShowPrompt()。

```
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
```

而 ShowPrompt()函数中又会调用 Validate()函数：

```
function Validate(entry){
return myRegExp.test(entry);
} // end function Validate()
```

在验证 entry 变量后，会构建一个包含指定给变量 myRegExp 的 RegExp 对象相关信息的字符串：

```
if (Validate(entry)){
displayString = "";
displayString += "There is a match!\nThe regular expression pattern is: " +
myRegExp + ".\nThe string that you entered was: '" + entry +"\n";
displayString += "The global property contained: " + myRegExp.global + "\n";
displayString += "The ignoreCase property contained: " + myRegExp.ignoreCase +
"\n";
displayString += "The multiline property contained: " + myRegExp.multiline + "\n";
displayString += "The source property contained: " + myRegExp.source + "\n";
displayString += "The lastIndex property contained: " + myRegExp.lastIndex;
alert(displayString);
} // end if
```

由于创建 myRegExp 变量时并没有指定相关的属性，所以 global、ignoreCase 和 multiline 属性包含的都是布尔值 false。

source 属性中包含着模式的值——\d+，该值是通过赋值语句指定给变量 myRegExp 的。

lastIndex 属性中包含着最后一个匹配项后面的字符位置值。当测试文本是 Hello 99 时，lastIndex 属性的值是 8——因为匹配项(字符序列 99)的位置是 6 和 7。当测试文本是 99 Hello 时，lastIndex 属性的值是 2——因为匹配项 99 的字符位置分别是 0 和 1(字符串位置的索引值从 0 开始)。

3. RegExp 对象的 test()方法

RegExp 对象的 test() 方法用于测试一个字符串是否与模式匹配。如果至少存在一个匹配项，test()方法就会返回布尔值 true。如果一个匹配项都没有，test()方法就会返回布尔值 false。

在前面的例子中，test() 方法返回的是 true。因为字符序列 Hello 99 和 99 Hello 中都包含两个数字，因此与匹配一个或多个数字的模式 \d+ 相匹配。

如果输入的字符序列中不包含数字，那么 test() 方法在匹配模式 \d+ 时就会返回布尔值 false。

4. RegExp 对象的 exec()方法

RegExp 对象的 exec() 方法既灵活强大，又容易使人迷惑。

首先，我们看一下当模式使用 g 属性(即 RegExp 对象的 global 属性会包含布尔值 true)时 exec() 方法的用途。

试一试：全局匹配(global 属性为 true)

本例中使用测试文件 RegExpExecExample.html，其中的标记和代码如下：

```
<html>
<head>
<title>RegExp exec() Method Example with global attribute set.</title>
<script language="javascript" type="text/javascript">
var myRegExp = /\sthe/ig;
var entry;
```

```
    function PatternProcess(entry){
    var displayString = "";

    while ((result = myRegExp.exec(entry)) != null ){
    displayString += "Matched '" + result;
    displayString += "' at position " + result.index + "\n";
    displayString += "The next match attempt begins at position " + myRegExp.lastIndex;
    alert(displayString);
    displayString = "";
    } // end while loop
    } // end function Process(entry)
    function ShowPrompt(){
    entry = prompt("This script tests for matches for the regular expression pattern: "
    + myRegExp + ".\nType in a string and click on the OK button.", "Type your text
    here.");
    PatternProcess(entry);
    } // end function ShowPrompt()

    </script>
    </head>
    <body>
    <form name="myForm">
    <br />
    <button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
    </form>
    </body>
    </html>
```

(1) 在 Internet Explorer 中打开 RegExpExecExample.html，然后单击 Click here to enter text 按钮。

(2) 在文本框中输入以下文本：Hello there, the theatre is nice.，其中包含符合模式\sthe 的三人匹配项。

图 19-11 显示的是在提示对话框中输入测试文本的屏幕截图。

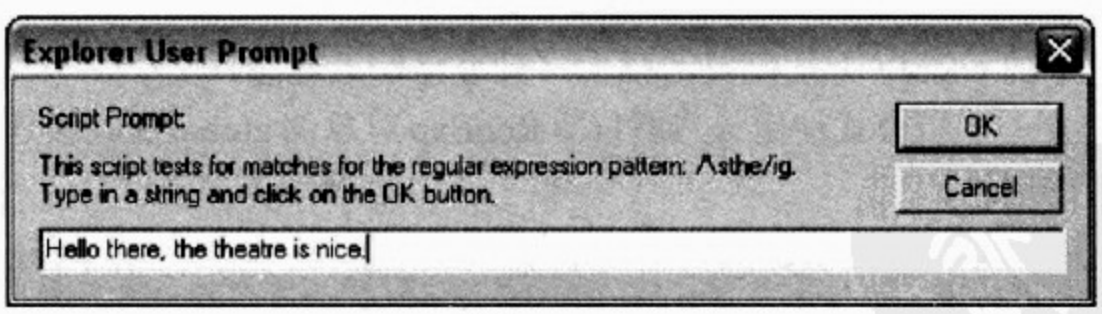


图 19-11

(3) 单击 OK 按钮，并观察警告框中显示的结果，如图 19-12 所示。此时匹配的文本是 the，其开始位置为 6。对于给定的测试字符串 Hello there, the theatre is nice.，注意匹配的是 there 中的前三个字母。

图　19-12

(4) 单击 OK 按钮，并观察如图 19-13 所示的警告框中显示的信息。此时匹配文本的开始位置是 12，这表示匹配项 the 是测试字符串中的单词 the。

(5) 单击 OK 按钮，并观察警告框中显示的信息，如图 19-14 所示。现在匹配文本的开始位置是 16，说明匹配项 the 是测试字符串中 theatre 的前三个字母。

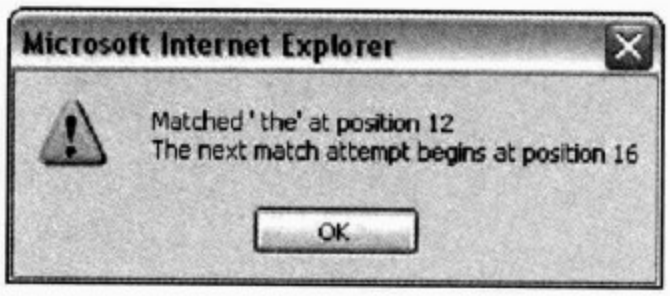


图　19-13

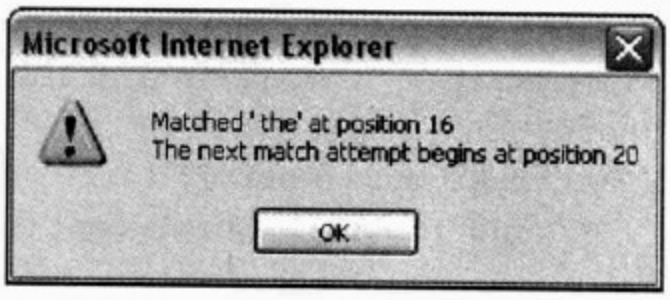


图　19-14

工作原理

根据代码的逻辑，程序会以单独的警告框来显示所有匹配项的信息。如果没有找到匹配项，则不显示任何信息。

要匹配的模式是 \sthe，即一个空白符后跟字符序列 the。

在测试字符串 Hello there，the theatre is nice.中包含三个与模式 \sthe 符合的匹配项。

第一个匹配项是单词 Hello 后面的空格符，后跟单词 there 中的字符序列 the。字符位置的编号从 0 开始，所以位置 5 位于匹配的第一个字符(空格符)之前。之后的匹配都要从 the 中 e 后面的位置 9 开始。

第二个匹配项是单词 there 后面的逗号之后的空格符。后面的字符序列 the 是单词 the。

第三个匹配项是单词 the 后面的空格符，后跟单词 theatre 中的前三个字符。

如果是在非全局匹配的模式下，RegExp 对象的 exec()方法只匹配一次。并且，它会把符合圆括号内模式的匹配项返回到一个数组中。下面来看一个实际的例子。

试一试：在非全局匹配模式下使用 exec() 方法

本例子中使用的测试文件 RegExpExecNonGlobal.html 所包含的内容如下：

```
<html>
<head>
<title>RegExp exec() Method Example with no global attribute.</title>
<script language="javascript" type="text/javascript">
var myRegExp = /((A|B)(\d{3}))/i;
var entry;
```

```
function PatternProcess(entry){
var displayString = "";
result = myRegExp.exec(entry);
for (n=0; n<5; n++){
displayString += "Matched '" + result[n];
displayString += "' in result[" + n + "].\n";
} // end for loop

alert(displayString);displayString = "";
} // end function Process(entry)

function ShowPrompt(){
entry = prompt("This script tests for matches for the regular expression pattern: "
+ myRegExp + ".\nType in a string and click on the OK button.", "Type your text
here.");
PatternProcess(entry);
} // end function ShowPrompt()

</script>
</head>
<body>
<form name="myForm">
<br />
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
</form>
</body>
</html>
```

(1) 在 Internet Explorer 中打开 RegExpExecNonGlobal.html，然后单击 Click here to enter text 按钮。

(2) 在提示对话框的文本框中输入零件编号 A234，然后单击 OK 按钮。

(3) 观察警告框中显示的结果，如图 19-15 所示。

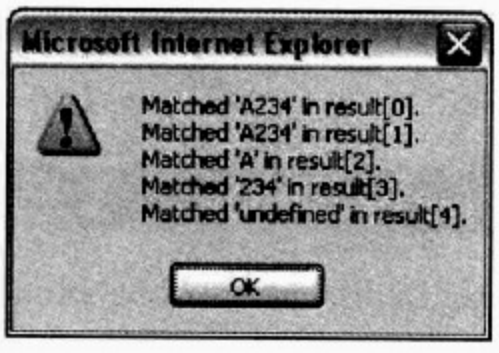


图 19-15

工作原理

在 result[0] 中，保存着匹配的整个字符序列—— A234。

在 result[1] 中，保存着匹配最外面圆括号中模式的字符序列。在这个例子中，也是

A234。

在 result[2] 中，保存着与嵌套的第一个圆括号中的模式匹配的字符序列，此时是 A。

在 result[3] 中，保存着与嵌套的下一个圆括号中的模式匹配的字符序列，此时是 234。

数组元素 result[4] 用来演示当没有下一对圆括号时，exec() 方法会返回值 undefined。

下面的“试一试”将把上面介绍的两种情况放到一起，以了解使用圆括号和全局匹配模式的更多内容：

试一试：在使用圆括号和全局匹配的情况下使用 exec()

本例中的测试文件是 RegExpExecExample2.html，其内容如下：

```
<html>
<head>
<title>RegExp exec() Method Example with global attribute set.</title>
<script language="javascript" type="text/javascript">
var myRegExp = /((A|B|C)(\d{3}))/ig;
var entry;

function PatternProcess(entry){
var displayString = "";

while ((result = myRegExp.exec(entry)) != null ){
displayString += "Matched '" + result;
displayString += "' at position " + result.index + "\n";
displayString += "The next match attempt begins at position " + myRegExp.lastIndex
+ "\n";
displayString += "The whole matching string is " + result[0] + "\n";
displayString += "The content of the outer parentheses is " + result[1] + "\n";
displayString += "The content of the first nested parentheses is " + result[2] +
"\n";
displayString += "The content of the second nested parentheses is " + result[3] +
"\n";
alert(displayString);
displayString = "";
} // end while loop
} // end function Process(entry)

function ShowPrompt(){
entry = prompt("This script tests for matches for the regular expression pattern: "
+ myRegExp + ".\nType in a string and click on the OK button.", "Type your text
here.");
PatternProcess(entry);
} // end function ShowPrompt()

</script>
</head>
<body>
<form name="myForm">
<br />
```

```
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
</form>
</body>
</html>
```

此时的匹配是全局性的，因为在声明 myRegExp 变量时已经指定了 g 属性。模式中也包含圆括号。

(1) 在 Internet Explorer 中打开 RegExpExecExample2.html，然后单击 Click here to enter text 按钮。

(2) 在文本框中输入文本 A123, B456, C789，然后单击 OK 按钮。

(3) 观察第一个警告框中显示的结果，如图 19-16 所示(此时的结果将在下面“工作原理”中讨论)。

(4) 单击 OK 按钮退出第一个警告框，并观察第二个警告框显示的结果，如图 19-17 所示。

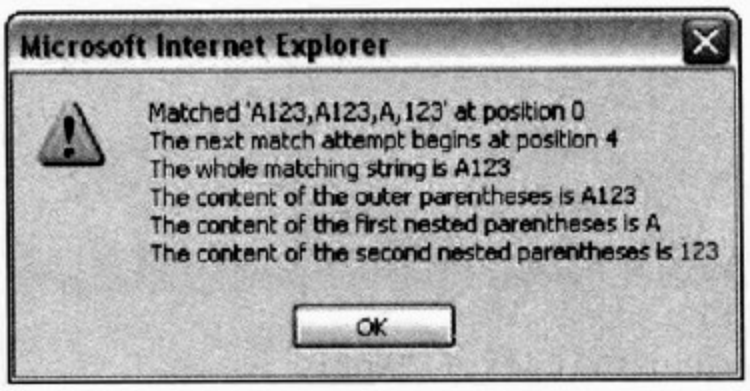


图 19-16

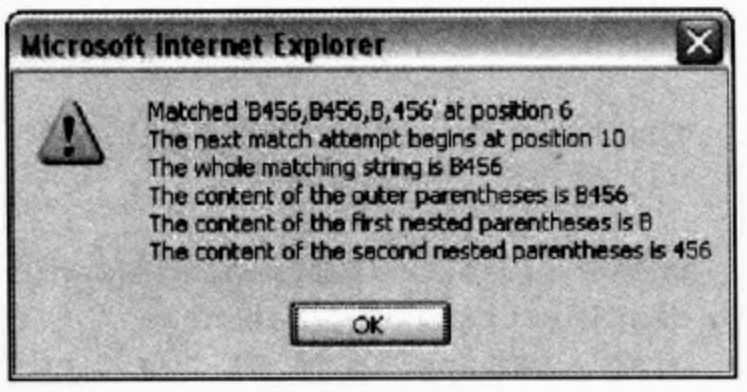


图 19-17

(5) 单击 OK 按钮退出第二个警告框，并观察第三个警告框中显示的结果，如图 19-18 所示。

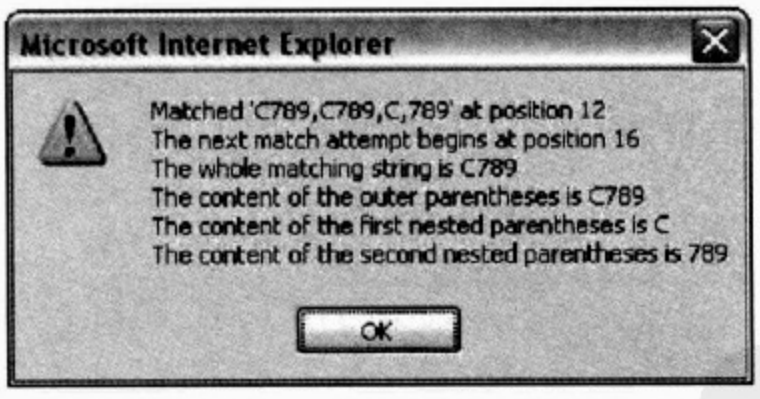


图 19-18

工作原理

下面的语句用来声明 myRegExp 变量：

```
var myRegExp = /((A|B|C)(\d{3}))/ig;
```

这个模式将会匹配以 A、B 或 C 开头的，并后跟三个数字的字符序列。虽然并不需要这么多圆括号，但它们有助于演示 exec() 方法返回到数组中的结果。

下面的测试字符串中包含符合模式 ((A|B|C)(\d{3})) 的三个匹配项：A123、B456 和 C789。

```
A123, B456, C789
```

首先，仔细看一下第一个警告框中显示的信息(如图 19-16 所示)。

该警告框中显示的信息是由下面代码构建而成的：

```
displayString += "Matched '" + result;
displayString += "' at position " + result.index + "\n";
displayString += "The next match attempt begins at position " + myRegExp.lastIndex
+ "\n";
displayString += "The whole matching string is " + result[0] + "\n";
displayString += "The content of the outer parentheses is " + result[1] + "\n";
displayString += "The content of the first nested parentheses is " + result[2] +
"\n";
displayString += "The content of the second nested parentheses is " + result[3] +
"\n";
```

而下面的代码则用来显示信息：

```
alert(displayString);
```

警告框中的第一行信息显示了由逗号分隔的 result 数组中的每一个值。如图 19-16 所示，该行信息是 Matched 'A123, A123, A, 123' at position 0。此时，逗号分隔的是 result[0]、result[1]、result[2] 和 result[3] 的值，匹配从位置 0 开始。

图 19-16 中警告框接下来的四行清楚地说明了结果数组的构成。

第二个匹配项是 B456，对应的结果如图 19-17 中的警告框所示。现在的位置变成了 6，result 数组中的元素也按照刚才介绍的顺序显示。

第三个匹配项是 C789，对应的结果显示于图 19-18 中。此时的位置是 12，result 数组中的元素同样按照本“工作原理”部分一开始所介绍的顺序显示。

19.1.2　String 对象

在 JavaScript/JScript 中，字符串指的是由一对双引号或一对单引号包围的 Unicode 字符序列。JavaScript/JScript 的 String 对象表示一个字符串。JavaScript 字符串的值与字符序列的值是相同的。String 对象则是这样一个字符序列的编程接口。

下面的每一行代码中都包含了一个 JavaScript/JScript 的字符串：

```
"Test"
'This is a multicharacter string enclosed in paired apostrophes.'
"99.31"
"This string has two \n lines."
```

字符串必须写在一行中。不过，对于多行字符串可以使用 \n 转义序列来表示换行符。

String 对象中与正则表达式有关的方法有三个：

- match()

- replace()
- search()

其中，match() 方法需要一个 RegExp 对象作为参数，并测试相应的字符串是否匹配与该对象关联的模式。

试一试：使用 String.match() 方法

在 String.match()方法的例子中使用的测试文件是 StringObjectMatch.html，它包含的代码如下所示：

```
<html>
<head>
<title>The match() Method of the String Object.</title>
<script language="javascript" type="text/javascript">
var myRegExp = /\d+\.\d+/;
var entryString;
var displayString = "";

function StringProcess(){
if (entryString.match(myRegExp) != null ){
var result = entryString.match(myRegExp);
displayString += "Matched '" + result + ".\n";
displayString += "result[0] is " + result[0] + ".\n";
displayString += "result[1] is " + result[1] + ".\n";
alert(displayString);
displayString = "";
} // end if statement
else
alert("The string you entered did not match the pattern " + myRegExp);
} // end function StringProcess()

function ShowPrompt(){
entryString = prompt("Type a string which is or contains a decimal number.\nType
and click on the OK button.", "Type a pattern here.");
StringProcess();
}

</script>
</head>
<body>
<form name="myForm">
<br />
<button type="Button" onclick="ShowPrompt()">Click here to enter a decimal
value.</button>
</form>
</body>
</html>
```

本例子中的模式可以匹配包含或者由小数组成的字符串。要匹配模式 \d+\.\d+，小数

点之前和之后都必须至少有一位数字。

(1) 在 Internet Explorer 中打开 StringObjectMatch.html，然后单击 Click here to enter a decimal value 按钮。

(2) 在提示对话框的文本框中输入字符序列 My score is 91.23，然后单击 OK 按钮。

(3) 观察如图 19-19 所示的警告框中显示的信息。

图　19-19

工作原理

下面的赋值语句将正则表达式模式\d+\.\d+赋给了变量 myRegExp：

```
var myRegExp = /\d+\.\d+/;
```

这个模式将用于匹配小数点前后至少都有一位数字的小数。

单击 Click here to enter a decimal value 按钮会调用函数 ShowPrompt()，该函数会显示提示对话框，请求用户输入。用户的输入会指定给变量 entryString：

```
entryString = prompt("Type a string which is or contains a decimal number.\nType
and click on the OK button.", "Type a pattern here.");
```

然后，调用 ProcessString() 函数。

首先，通过一个 if 语句来测试是否存在匹配项：

```
if (entryString.match(myRegExp) != null )
```

如果没有匹配项，entryString.match(myRegExp) 将返回 null，而函数的执行过程将前进到 else 子句(稍后介绍)。如果存在匹配项， entryString.match(myRegExp) 的返回值就不是 null，if 语句中的代码会继续执行。

变量 result 是一个数组(因为知道存在匹配项)，其元素 result[0] 包含匹配的字符序列。result[1] 及其之后的元素则包含由正则表达式模式中的圆括号表示的匹配项的各个组成部分。这与 RegExp 对象的 exec()方法的行为类似：

```
if (entryString.match(myRegExp) != null ){
var result = entryString.match(myRegExp);
displayString += "Matched '" + result + ".\n";
displayString += "result[0] is " + result[0] + ".\n";
displayString += "result[1] is " + result[1] + ".\n";
alert(displayString);
displayString = "";
```

声明一个作为空字符串的全局变量 displayString：

```
var displayString = "";
```

然后，在 if 语句内，其内容使用结果的值——result[0] 和 result[1] 来构建。如图 19-19 所示，result[1] 的值是 undefined ——因为模式 \d+\.\d+ 中不包含圆括号。最后，当退出警告框后，displayString 的值又被重置回空字符串。因为不想将它保存的数据提交保存，所以这一步是必要的。

如果 entryString.match(myRegExp)返回 null，就会直接跳到 else 子句，该子句显示没有找到匹配项的信息。

在使用 String.match()方法时指定 g 属性也会如前面使用 RegExp.exec() 方法一样产生多个结果。如果模式中有圆括号，结果将是元素中包含每一个匹配项的数组，这也同 RegExp.exec()方法类似。

String.search()方法以 RegExp 对象作为参数。该方法返回字符串中第一个匹配项的位置，如果没有匹配项则返回 -1。

String.replace()方法需要两个参数——第一个是 RegExp 对象，第二个是替换 String 对象中匹配文本的字符串。这个替换字符串可以直接指定，也可以使用函数调用指定。

19.2 JavaScript 和 JScript 中的元字符

JavaScript 和 JScript 中的正则表达式基于 Perl 语言中的正则表达式。所以，随着 Perl 对正则表达式支持的改进，JScript 和 JavaScript 对正则表达式的支持也在不断进步。

下面的表 19-1 总结了 JavaScript 1.5 和 JScript 5.6 支持的元字符。

表 19-1 JavaScript 1.5 和 JScript 5.6 支持的元字符

元 字 符	说 明
\d	匹配单个数字
.	(句点元字符)匹配除了换行符或其他表示换行的 Unicode 字符之外的任何字符
\w	匹配任何 ASCII 字母字符——即 [A-Za-z0-9]
\W	匹配任何 \w 不匹配的字符
\s	匹配一个空白符
\S	匹配任何 \s 不匹配的字符
\D	匹配任何 \d 不匹配的字符——即 \D 等价于 [^0-9]
[...]	字符类。匹配方括号中的任何单个字符。也支持范围
[^...]	取反的字符类
?	限定符。表示匹配前面字符或组的零个或一个实例
*	限定符。表示匹配前面字符或组的零个或多个实例
+	限定符。表示匹配前面字符或组的一个或多个实例
{n,m}	限定符。表示匹配前面字符或组最少 n 个、最多 m 个实例

19.3　说明 JavaScript 正则表达式

许多语言都支持在正则表达式中使用扩展的空白符。遗憾的是，JavaScript 和 JScript 并不支持这一特性。

建议除非常一般的正则表达式模式外，最好在为声明的变量指定 RegExp 对象的赋值语句之前或之后的注释行中说明模式的细节。

例如，假设声明了以下变量：

```
var myRegExp = /\d{3}-\d{2}-\d{4}/;
```

那么可以这样来对它加以说明：

```
var myRegExp = /\d{3}-\d{2}-\d{4}/;
// \d{3} matches three numeric digits
// - matches a hyphen
// \d{2} matches two numeric digits
// - matches a hyphen
// \d{4} matches four numeric digits
```

也可像这样来说明：

```
var myRegExp = /\d{3}-\d{2}-\d{4}/;
/* \d{3} matches three numeric digits
- matches a hyphen
\d{2} matches two numeric digits
- matches a hyphen
\d{4} matches four numeric digits
*/
```

不论采取哪种方式，如果能够在代码文件中逐步地说明模式开发人员的意图，对于将来维护这些复杂的代码都是非常有用的。而且，像这样添加说明对于编写代码也很有价值，比如在说明中可以记载以前没有注意到的问题。养成这种良好的习惯一定会避免许多编码错误。

19.4　验证 SSN 的例子

可以使用 JavaScript 和 JScript 通过正则表达式来验证在网页表单中输入的信息。下面这个例子将验证一个美国社会保险号(SSN)格式的有效性。SSN 的格式是三位数字后跟一个连字符，后跟两位数字，后跟一个连字符，再后跟四位数字。

这里使用的测试文件 SSNValidation.html 是为 Internet Explorer 设计的：

```
<html>
<head>
<title>Processing an SSN</title>
<script language="javascript" >
```

```
myRegExp = /\d{3}-\d{2}-\d{4}/;
var entry;

function Validate(){
entry = simpleForm.SSNBox.value;
if (myRegExp.test(entry)) {
alert("The value you entered, " + entry + "\n matches the regular expression, "
+ myRegExp + ". It is a valid SSN." );
} // end the if statement
else
{
alert("The value you entered," + entry + ",\nis not a valid SSN. Please try
again.");
} // end of else clause
} // end Validate() function

function ClearBox(){
simpleForm.SSNBox.value = "";
// The above line clears the texbox when it receives focus
} // end ClearBox() function
</script>
</head>
<body>
<form name="simpleForm" >
<table>
<tr>
<td width="40%">Enter a valid SSN here:</td>
<td><input name="SSNBox" onfocus="ClearBox()" type="text" value="Enter an SSN
here"></input></td>
</tr>
<tr>
<td><input name="Submit" type="submit" value="Check the SSN"
onclick="Validate()"></input></td>
</tr>
</table>
</form>
</body>
</html>
```

工作原理

指定的模式将匹配三位数字后跟一个连字符，后跟两位数字，再后跟一个连字符和四位数字：

```
myRegExp = /\d{3}-\d{2}-\d{4}/;
```

通过 RegExp 对象的 test()方法来判定是否存在匹配项：

```
if (myRegExp.test(entry)) {
```

如果匹配成功，会显示包含用户输入文本的信息。如果匹配失败，则会告知用户输入的值不是一个 SSN。

19.5　练习

下面的练习题旨在测试对本章内容的理解情况：

1. 请修改 FinalT.html 中的代码 ，使其中的模式匹配至少包含一位数字和只包含一位数字的用户输入文本。

2. 请修改 SSNValidation.html 中的代码，使其匹配一个 16 位的信用卡号码。信用卡号码的格式是四位数字一组，中间以空格符分隔。

第20章

正则表达式与 VBScript

VBScript 是从 5.0 版开始引入正则表达式功能的。在 VBScript 中，正则表达式可以用于解析字符序列(字符串)，以及提供灵活的替换功能。

在 Internet Explorer 浏览器提供 VBScript 解释程序的情况下，VBScript 可以用在客户端的网页代码中。此外，VBScript 也可以用在 Windows 脚本主机(Windows Script Host，WSH)中。

VBScript 解释程序(vbscript.dll)不允许存取文件。但当在 WSH 中使用 VBScript 时，相关的文件——scrrun.dll，则允许 VBScript 存取文件并操纵目录。

在本章中将学习以下内容：

- 如何使用 RegExp 对象的属性和方法
- 如何使用 Match 对象和 Matches 集合
- VBScript 支持的元字符及其用法
- 如何使用 VBScript 正则表达式解决某些文本处理问题

20.1 RegExp 对象及其用法

RegExp 对象、Match 对象和 Matches 集合都与在 VBScript 中使用正则表达式有关。本节集中讨论 RegExp 对象。

RegExp 对象有三个属性和三个方法。其属性如下：

- Pattern 属性
- Global 属性
- IgnoreCase 属性

下面几节中会详细介绍这些属性。三个方法如下：

- Execute 方法
- Replace 方法
- Test 方法

这三个方法也会在后面几节中介绍并通过例子进行示范。对于完成简单的匹配而言，Pattern 属性和 Test()方法是最常用的。

20.1.1　RegExp 对象的 Pattern 属性

VBScript 的 RegExp 对象与本书前面讨论的工具和语言存在着功能上的差别。例如，在 VBScript 中没有像下面这样的 JScript 声明和赋值语句的语法：

```
var myRegExp = /\d{3}/;
```

相反，VBScript 使用 RegExp 对象的 Pattern 属性来保存一个字符串值——即正则表达式模式。

因此，等价于前面 JScript 代码的 VBScript 代码如下所示：

```
Dim myRegExp
Set myRegExp = new RegExp
myRegExp.Pattern = "\d{3}"
```

下面的例子描述一个非常简单的函数，该函数使用 Pattern 属性进行了简单的搜索操作。

试一试：简单的匹配操作

测试文件 TestForA.html 描述了如何使用 Pattern 属性：

```
<html>
<head>
<title>Test For Upper Case A</title>
<script language="vbscript" type="text/vbscript">
Function MatchTest
Dim myRegExp, TestString
Set myRegExp = new RegExp
myRegExp.Pattern = "A"
TestString = "Andrew"
If myRegExp.Test(TestString) = True Then
  MsgBox "The test string '" & TestString & "' matches the pattern '" &
myRegExp.Pattern & "'."
Else
  MsgBox "There is no match."
End If
End Function

</script>
</head>
<body onload="MatchTest">

</body>
</html>
```

在 Internet Explorer 中打开 TestForA.html，当页面载入完成时注意消息框中显示的信息。

图 20-1 是显示的消息框界面。它正确地指出在字符序列 Andrew 中存在一个与模式 A 匹配的字符。

图 20-1

工作原理

当 HTML 页面载入完成时，会调用 MatchTest()函数：

```
<body onload="MatchTest">
```

所有功能都包含在了 MatchTest() 函数中：

```
Function MatchTest
```

首先，定义变量 myRegExp(在它的 Pattern 属性中包含正则表达式模式)和 TestString(用来保存测试字符串)：

```
Dim myRegExp, TestString
```

然后，在 Set 语句中，将一个新的 RegExp 对象指定给变量 myRegExp：

```
Set myRegExp = new RegExp
```

接着，把一个直接量字符——A，指定给 myRegExp 对象的 Pattern 属性：

```
myRegExp.Pattern = "A"
```

之后，再把字符序列 Andrew 指定给变量 TestString：

```
TestString = "Andrew"
```

匹配成功则 If 语句会显示一个消息框。在 If 语句的逻辑测试中使用了 myRegExp 变量的 Test()方法。Test()方法测试一个字符串(该方法的唯一参数)是否包含与同一个 RegExp 对象(本例中指的是变量 myRegExp)的 Pattern 属性值符合的匹配项：

```
If myRegExp.Test(TestString) = True Then
```

要显示的信息由文本、变量 TestString 的值和 myRegExp.Pattern 属性的值组成：

```
MsgBox "The test string '" & TestString & "' matches the pattern '" &
myRegExp.Pattern & "'."
```

同时，也会有一个 Else 子句用于显示不存在匹配项的信息。不过，在本例中 Else 子句根本没有用到 myRegExp.Pattern 和 TestString 的值，在后面的例子中，在用户应用测试字符串时均需要在该处加一条 Else 从句，如下所示：

```
Else
MsgBox "There is no match."
```

最后，End If 和 End Function 语句结束 MatchTest 函数：

```
End If
End Function
```

20.1.2　RegExp 对象的 Global 属性

RegExp 对象的 Global 属性可以设置为布尔值 True 或 False，但 Global 属性的默认值是 False。这表示 Pattern 属性的值只能找到一个匹配项。当 Global 属性的值是 True 时，匹配会尝试搜索整个字符串，因此可能会返回多个匹配项。

试一试：使用 Global 属性

本例使用测试文件 MatchGlobal.html，其内容如下：

```
<html>
<head>
<title>Carry out a non-global replace and a global replace.</title>
<script language="vbscript" type="text/vbscript">
Dim myRegExp, InputString, ChangedString

Function MatchGlobal
Set myRegExp = new RegExp
myRegExp.Pattern = "A"
DoReplaceDefault
DoReplaceGlobal
End Function

Function DoReplaceDefault
InputString = InputBox("Enter a string. It will be tested once to see if it
contains" &VBCrLf & "any 'A' characters. Any 'A' will be replaced by 'B'")
myRegExp.Global = False
ChangedString = myRegExp.Replace(InputString, "B")
If myRegExp.Test(InputString) = True Then
MsgBox "The test string '" & InputString & "' matches the pattern '" &
myRegExp.Pattern & "'." _& VBCrLf
& "The changed string is " & ChangedString
Else
MsgBox "There is no match. '" & InputString & "' does not match " &VBCrLf _
& "the pattern '" & myRegExp.Pattern & "'."
End If
End Function

Function DoReplaceGlobal
InputString = InputBox("Enter a string. It will be tested to see if it contains"
&VBCrLf & "any 'A' characters. Any 'A' will be replaced by 'B'")
myRegExp.Global = True
ChangedString = myRegExp.Replace(InputString, "B")
If myRegExp.Test(InputString) = True Then
  MsgBox "The test string '" & InputString & "' matches the pattern '" &
```

```
myRegExp.Pattern & "'." & VBCrLf _
  & "The changed string is " & ChangedString
Else
  MsgBox "There is no match. '" & InputString & "' does not match " &VBCrLf _
  & "the pattern '" & myRegExp.Pattern & "'."
End If
End Function

</script>
</head>
<body onload="MatchGlobal">

</body>
</html>
```

在使用 Global 属性的同时，代码中也使用了 RegExp 对象的 Replace()方法。

当打开 MatchGlobal.html 时，会显示两个消息框。第一个是在只查找一个匹配项的情况下显示的，而第二个则是在找到测试字符串中所有匹配项的情况下显示的。

(1) 在 Internet Explorer 中打开 MatchGlobal.html。

(2) 在显示的输入框中输入字符序列 THE APPLE IS A TASTY FRUIT。由于在 VBScript 中默认执行的是区分大小写的匹配，所以要确保全部大写。

(3) 单击 OK 按钮，并观察显示的消息框。

图 20-2 显示的是第 3 步后的界面。结果是只替换了一个大写字母 A。

图 20-2

(4) 单击消息框中的 OK 按钮，并在输入框中重新输入相同的字符串 THE APPLE IS A TASTY FRUIT。

(5) 单击 OK 按钮，并观察消息框中显示的结果。

图 20-3 显示的是第 5 步后的界面。此时测试字符串中的每个大写的 A 都被替换成了大写的 B。

图 20-3

工作原理

本例中的代码分为三个函数：MatchGlobal、DoReplaceDefault 和 DoReplaceGlobal。因为三个函数中都用到 myRegExp 变量和其他变量，所以将其声明为全局变量：

```
Dim myRegExp, InputString, ChangedString
```

MatchGlobal 函数使用 Set 语句创建 RegExp 对象的一个引用，将其保存在 myRegExp 变量中：

```
Function MatchGlobal
Set myRegExp = new RegExp
myRegExp.Pattern = "A"
DoReplaceDefault
DoReplaceGlobal
End Function
```

通过把字符串值 A 指定给 myRegExp 对象的 Pattern 属性的值，使其成为一个需要匹配的直接量模式。

下面的 DoReplaceDefault 函数能接受用户通过输入框输入的一个字符串：

```
Function DoReplaceDefault
InputString = InputBox("Enter a string. It will be tested once to see if it
contains" &VBCrLf & "any 'A' characters. Any 'A' will be replaced by 'B'")
myRegExp.Global = False
ChangedString = myRegExp.Replace(InputString, "B")
If myRegExp.Test(InputString) = True Then
  MsgBox "The test string '" & InputString & "' matches the pattern '" &
myRegExp.Pattern & "'." _& VBCrLf
  & "The changed string is " & ChangedString
Else
  MsgBox "There is no match. '" & InputString & "' does not match " &VBCrLf
_
  & "the pattern '" & myRegExp.Pattern & "'."
End If
End Function
```

当使用 VBScript 的 InputBox()函数时，显示输入框。而且，输入框中的信息会提示用户输入的字符串只会在测试中被匹配一次。

要对用户在输入框中输入的字符串中的任意(第一个) A 进行替换，就要用到 RegExp 对象的 Replace() 方法：

```
ChangedString = myRegExp.Replace(InputString, "B")
```

注意，没有必要为 Replace() 方法指定一个要匹配的模式作为参数。因为相应的模式已经在前面指定给了 myRegExp 的 Pattern 属性：

```
myRegExp.Pattern = "A"
```

Replace() 方法的两个参数分别是要执行替换操作的字符串(即 InputString 变量的值)

和用于替换匹配 Pattern 属性值(即模式 A)的第一个匹配项的字符序列。

在 THE APPLE IS A TASTY FRUIT 中，第一个 A 的实例是 APPLE 的首字母。这个 A 被 Replace()方法的第二个参数 B 所替换。因此，下面的 Changed String 变量值是 THE BPPLE IS A TASTY FRUIT，字符序列 BPPLE 替换了字符序列 APPLE：

```
ChangedString = myRegExp.Replace(InputString, "B")
```

在这种情况下，匹配并替换一次是默认行为。

下面的 DoReplaceGlobal 函数要做的差不多一样，只是没有限制对输入字符串的匹配次数：

```
Function DoReplaceGlobal
InputString = InputBox("Enter a string. It will be tested to see if it contains"
&VBCrLf & "any 'A' characters. Any 'A' will be replaced by 'B'")
myRegExp.Global = True
ChangedString = myRegExp.Replace(InputString, "B")
If myRegExp.Test(InputString) = True Then
   MsgBox "The test string '" & InputString & "' matches the pattern '" &
myRegExp.Pattern & "'." & VBCrLf _
   & "The changed string is " & ChangedString
Else
   MsgBox "There is no match.'"& InputString & "' does not match " &VBCrLf _
   & "the pattern '" & myRegExp.Pattern & "'."
End If
End Function
```

由于将 RegExp 对象的 Global 属性值设置为 True，所以改变了默认的匹配行为：

```
myRegExp.Global = True
```

此时，每当 Pattern 属性的值——大写的 A 找到匹配项，相应的匹配项就会被大写的 B 替换，最终的替换结果保存在变量 ChangedString 中：

```
ChangedString = myRegExp.Replace(InputString, "B")
```

因此，输入字符串 THE APPLE IS A TASTY FRUIT 中每一个 A 都会被替换成 B，变量 ChangedString 保存的值将变成 THE BPPLE IS B TBSTY FRUIT。

20.1.3 RegExp 对象的 IgnoreCase 属性

RegExp 对象的 IgnoreCase 属性可以允许完成不区分大小写的匹配。

在前面使用 MatchGlobal.html 的例子中，输入的字符串全部采取大写形式。更自然的写法并非 THE APPLE IS A TASTY FRUIT，而可能是 The apple is a tasty fruit.。

通过下面的测试文件 CaseReplace.html，可以试一试区分大小写(默认)和不区分大小写的替换操作：

```
<html>
<head>
<title>Carry out a case-sensitive replace and a case-insensitive replace.</title>
```

```
<script language="vbscript" type="text/vbscript">
Dim myRegExp, InputString, ChangedString
Function MatchCaseOptions
Set myRegExp = new RegExp
myRegExp.Pattern = "A"
myRegExp.Global = True
DoReplaceSensitive
DoReplaceInsensitive
End Function

Function DoReplaceSensitive
InputString = InputBox("Enter a string. It will be tested once to see if it
contains" &VBCrLf & "any 'A' characters. Any 'A' will be replaced by 'B'")
ChangedString = myRegExp.Replace(InputString, "B")
If myRegExp.Test(InputString) = True Then
  MsgBox "The test string '" & InputString & "' matches the pattern '" &
myRegExp.Pattern & "'." & VBCrLf _
  & "The changed string is " & ChangedString
Else
  MsgBox "There is no match. '" & InputString & "' does not match " &VBCrLf
_
  & "the pattern '" & myRegExp.Pattern & "'."
End If
End Function

Function DoReplaceInsensitive
myRegExp.IgnoreCase = True
InputString = InputBox("Enter a string. It will be tested to see if it contains"
&VBCrLf & "any 'A' characters. Any 'A' will be replaced by 'B'")
ChangedString = myRegExp.Replace(InputString, "B")
If myRegExp.Test(InputString) = True Then
  MsgBox "The test string '" & InputString & "' matches the pattern '" &
myRegExp.Pattern & "'." & VBCrLf _
  & "The changed string is " & ChangedString
Else
  MsgBox "There is no match. '" & InputString & "' does not match " &VBCrLf
_
  & "the pattern '" & myRegExp.Pattern & "'."
End If
End Function

</script>
</head>
<body onload="MatchCaseOptions">

</body>
</html>
```

试一试：使用 RegExp 对象的 IgnoreCase 属性

首先，尝试使用 IgnoreCase 属性的默认值来匹配输入的字符串。

(1) 在 Internet Explorer 中打开 CaseReplace.html，在显示的输入框中输入测试字符串 The apple is a tasty fruit.。

(2) 单击 OK 按钮，并观察消息框中的信息。如图 20-4 所示，此时没有找到匹配项。

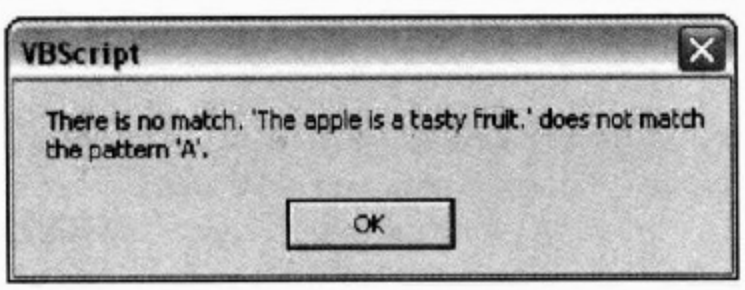


图 20-4

下面，再尝试在将 IgnoreCase 属性的值设置为 True 的情况下来匹配输入的字符串。换句话说，就是进行不区分大小写的匹配。

(3) 单击 OK 按钮退出如图 20-4 所示的消息框。

(4) 在显示的输入框中，输入测试字符串 The apple is a tasty fruit.。

(5) 单击 OK 按钮，并观察消息框中显示的信息，如图 20-5 所示。

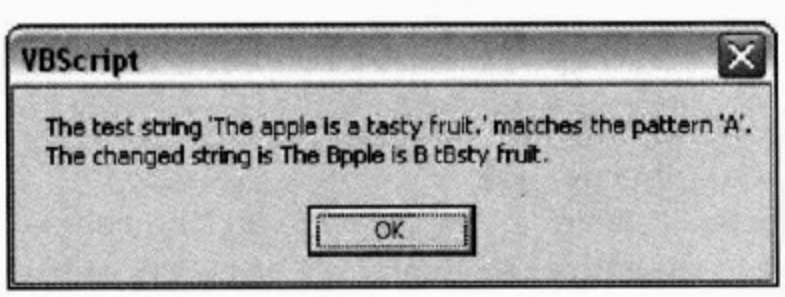


图 20-5

工作原理

当 CaseReplace.html 载入完成后，就会调用函数 MatchCaseOptions：

```
<body onload="MatchCaseOptions">
```

在 MatchCaseOptions 中，Global 属性的值被设置为 True，因此所有的匹配项都将被替换：

```
Function MatchCaseOptions
Set myRegExp = new RegExp
myRegExp.Pattern = "A"
myRegExp.Global = True
DoReplaceSensitive
DoReplaceInsensitive
End Function
```

函数 DoReplaceSensitive 会执行一个与前面例子中所看到的类似的全局性匹配。

此时函数 DoReplaceInsensitive 同样进行全局性匹配，但不区分大小写——因为此时

IgnoreCase 属性的值被设置为 True：

```
Function DoReplaceInsensitive
myRegExp.IgnoreCase = True
```

因此，如图 20-5 所示，字符串 The apple is a tasty fruit. 中所有的字母 A(无论大小写)都被大写的 B 所替换。

20.1.4　RegExp 对象的 Test() 方法

Test() 方法根据指定的字符串来尝试匹配正则表达式并返回布尔值。由 Test() 方法返回的布尔值可以用于逻辑测试。例如，在前面例子中曾看到过下面的代码，这些代码测试 TestString 变量中的测试字符串是否匹配 Pattern 属性的值，并且根据返回的布尔值相应地显示匹配成功或失败的消息。

```
If myRegExp.Test(TestString) = True Then
  MsgBox "The test string '" & TestString & "' matches the pattern '" &
myRegExp.Pattern & "'."
Else
  MsgBox "There is no match."
End If
```

20.1.5　RegExp 对象的 Replace() 方法

Replace()方法可以将与保存在 RegExp 对象的 Pattern 属性中的模式匹配的部分字符串替换成其他字符串。在替换过程中，被替换的字符串是 Replace()方法的第一个参数，而替换字符串则是第二个参数。

当在 Replace()方法中使用分组的圆括号时，可以颠倒第一个参数的字符串的相应组成部分的顺序。匹配字符串中被捕获的组可以用做替换字符串来实现顺序颠倒。

试一试：使用 Replace() 方法实现顺序颠倒

测试文件 ReverseName.html 的内容如下：

```
<html>
<head>
<title>Reverse Surname and First Name</title>
<script language="vbscript" type="text/vbscript">
Function ReverseName
Dim myRegExp, TestName, Match
Set myRegExp = new RegExp
myRegExp.Pattern = "(\S+)(\s+)(\S+)"
TestString = InputBox("Enter your name below, in the form" & VBCrLf & _
  "first name, then a space then last name." & VBCrLf & "Don't enter an initial or
middle name.")
Match = myRegexp.Replace(TestString, "$3,$2$1")
If Match <> "" Then
  MsgBox "Your name in last name, first name format is:" & VBCrLf & Match
Else
  MsgBox "You didn't enter your name." & VBCrLF & "Press OK then F5 to run the
```

```
example again."
End If
End Function

</script>
</head>
<body onload="ReverseName">

</body>
</html>
```

(1) 在 Internet Explorer 中打开 ReverseName.html。

(2) 在显示的输入框中以“名一姓”的格式输入名字，名和姓之间至少要有一个空格。图 20-6 显示的是按照要求的格式输入的名字 John Smith。

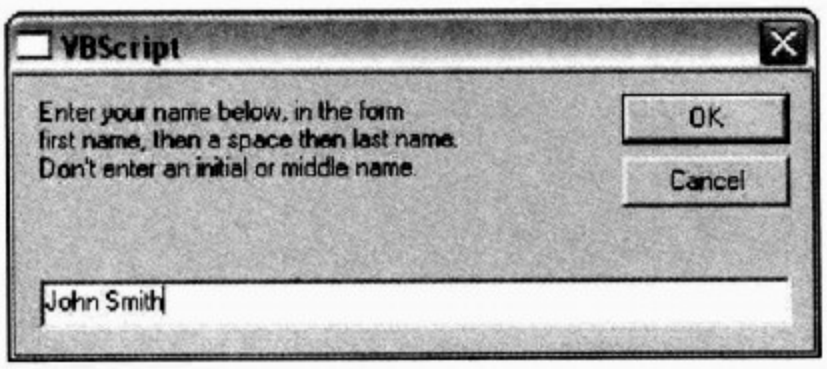


图 20-6

(3) 单击 OK 按钮，并观察结果，如图 20-7 所示。

如果是按照要求的格式(名 姓)输入名字，那么显示在消息框中的名字的格式就会变成“姓,名”。

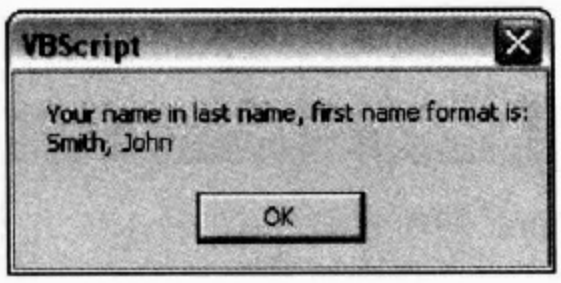


图 20-7

工作原理

当网页载入完成后，会调用 ReverseName 函数：

```
<body onload="ReverseName">
```

模式 (\S+)(\s+)(\S+) 被指定给 myRegExp 变量的 Pattern 属性：

```
myRegExp.Pattern = "(\S+)(\s+)(\S+)"
```

该模式捕获三个组。第一个，由 (\S+) 指定，匹配并捕获一个或多个非空白字符。第二个，由 (\s+) 指定，匹配并捕获一个或多个空白符。第三个，由 (\S+) 指定，匹配并捕获一个或多个非空白字符。如果用户按照要求的格式输入自己名字，自己名字后面跟一个

空格，再跟姓，那么变量 TestString 中就会包含一个匹配项。与三个组匹配的内容分别保存在专用变量 $1、$2 和 $3 中：

```
TestString = InputBox("Enter your name below, in the form" & VBCrLf & _
"first name, then a space then last name." & VBCrLf & "Don't enter an initial or
middle name.")
```

可以将变量 $3、$2 和 $1 用在像下面这样的替换操作中：

```
Match = myRegexp.Replace(TestString, "$3,$2$1")
```

因为姓包含在 $3 而名包含在 $1 中，所以可以通过模式 $3,$2$1 来颠倒名和姓的顺序并插入一个逗号。如果保证输出中恰好包含一个空格(而不是反映用户在输入框中输入的空白符)，则可以将替换模式修改为 $3, $1。

If 语句控制着是显示成功匹配的消息，还是显示对用户输入的名字不符合要求的提示：

```
If Match <> "" Then
  MsgBox "Your name in last name, first name format is:" & VBCrLf & Match
Else
  MsgBox "You didn't enter your name." & VBCrLF & "Press OK then F5 to run the
example again."
End If
```

20.1.6　RegExp 对象的 Execute()方法

RegExp 对象的 Execute() 方法会根据指定的字符串来执行正则表达式匹配。正则表达式模式保存在同一个 RegExp 对象的 Pattern 属性中。Execute() 方法的返回值是一个包含匹配项的集合，它包含测试文本中每个匹配项的(Match)对象。

试一试：使用 Execute() 方法

测试文件 ExecuteDemo.html 的内容如下：

```
<html>
<head>
<title>Demo of the Execute() Method</title>
<script language="vbscript" type="text/vbscript">
Function ExecuteDemo
Dim myRegExp, TestName, Match, Matches, displayString
displayString = ""
Set myRegExp = new RegExp
myRegExp.Pattern = "[A-Z]\d"
myRegExp.IgnoreCase = True
myRegExp.Global = False
TestString = InputBox("Enter characters and numbers in the text box below.")
Set Matches = myRegexp.Execute(TestString)
  For Each Match in Matches
    displayString = displayString & "Match found at position " & Match.FirstIndex &
VBCrLf
    displayString = displayString & "The match value is '" & Match.Value & "'."
```

```
    MsgBox displayString
    displayString = ""
 Next
End Function
</script>
</head>
<body onload="ExecuteDemo">

</body>
</html>
```

(1) 在 Internet Explorer 中打开 ExecuteDemo.html，并在输入框中输入字符序列 A9。

(2) 单击 OK 按钮，并观察消息框中显示的文本，如图 20-8 所示。其中提到在位置 0 找到匹配项，说明字符位置是从零开始计算的。

图 20-8

(3) 单击 OK，并按 F5 重新载入页面以再次运行脚本。

(4) 在输入框中输入字符序列 A9 A10 A11 A12 A13。

(5) 单击 OK 按钮，并观察消息框中显示的文本，如图 20-9 所示。结果与上一次相同。这是因为 myRegExp.Global 属性的值被设置成 False(后面还要再用到这个例子，不过届时 myRegExp.Global 属性的值将被设置为 True)。

图 20-9

工作原理

当页面载入后，会调用 ExecuteDemo 函数：

```
<body onload="ExecuteDemo">
```

首先，定义 ExecuteDemo 中要使用的变量：

```
Function ExecuteDemo
Dim myRegExp, TestName, Match, Matches, displayString
```

将变量 displayString 的值设置为空字符串，并设置一个新 RegExp 对象的引用：

```
    displayString = ""
    Set myRegExp = new RegExp
```

将匹配一个字母字符后跟一个数字的正则表达式模式 [A-Z]\d 指定给 myRegExp 对象的 Pattern 属性：

```
    myRegExp.Pattern = "[A-Z]\d"
```

将 IgnoreCase 属性设置为 True，表示匹配过程将不区分大小写：

```
    myRegExp.IgnoreCase = True
```

将 Global 属性的值设置为 False，表示 Execute()方法将只返回一个匹配项：

```
    myRegExp.Global = False
```

使用 InputBox()函数收集用户输入，并把输入指定给 TestString 变量：

```
    TestString = InputBox("Enter characters and numbers in the text box below.")
```

通过 Execute()方法返回一个包含匹配项的集合：

```
    Set Matches = myRegexp.Execute(TestString)
```

Matches 集合中的每一个 Match 对象都会在 For Each 循环中被处理。如果没有找到匹配项，该循环将不会产生输出。变量 displayString 用于构建要显示的字符串：

```
    For Each Match in Matches
     displayString = displayString & "Match found at position " & Match.FirstIndex &
    VBCrLf
        displayString = displayString & "The match value is '" & Match.Value & "'."
```

然后，使用 MsgBox 函数来显示 displayString 变量的值：

```
    MsgBox displayString
```

最后，再次将 displayString 变量设置为空字符串，以备下一次 For Each 循环迭代时使用。如果不将 displayString 变量重置为空字符串，那么每个匹配项的信息连接到一起就会变得过长。有时候比较喜欢这种情况，比如当把 Global 属性的值设为 True 时，要在同一个消息框中显示出 Matches 集合中所有 Match 对象的相关信息：

```
        displayString = ""
    Next
End Function
```

下面的例子修改了上面例子中的代码，使测试字符串中包含的所有匹配项都能返回。

试一试：当 Global 属性设置为 True 时使用 Execute()方法

测试文件 ExecuteDemoGlobal.html 中的代码如下：

```
<html>
<head>
<title>Demo of the Execute() Method</title>
```

```
<script language="vbscript" type="text/vbscript">
Function ExecuteDemo
Dim myRegExp, TestName, Match, Matches, displayString
displayString = ""
Set myRegExp = new RegExp
myRegExp.Pattern = "[A-Z]\d"
myRegExp.IgnoreCase = True
myRegExp.Global = True
TestString = InputBox("Enter characters and numbers in the text box below.")
Set Matches = myRegexp.Execute(TestString)
  For Each Match in Matches
   displayString = displayString & "Match found at position " & Match.FirstIndex &
"."
   displayString = displayString & "The match value is '" & Match.Value & "'." &
VBCrLf
  'displayString = ""
 Next
 MsgBox displayString
End Function

</script>
</head>
<body onload="ExecuteDemo">

</body>
</html>
```

(1) 在 Internet Explorer 中打开 ExecuteDemoGlobal.html。

(2) 在输入框中输入测试字符串 A9 A10 A11 A12 A13。

(3) 单击 OK 按钮，并观察消息框中显示的结果，如图 20-10 所示。现在，Matches 集合中的每个匹配项的信息都显示在消息框中。

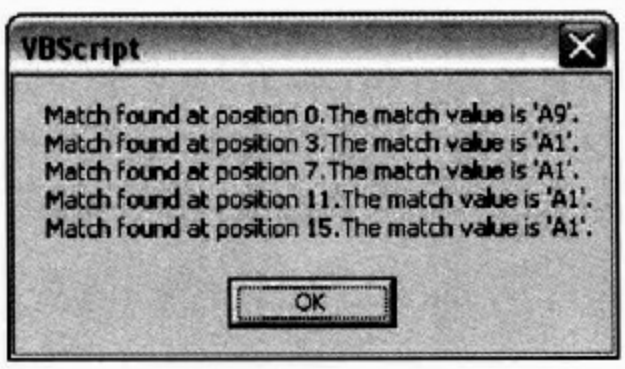


图 20-10

工作原理

这个测试文件中的代码与 ExecuteDemo.html 中的代码的工作原理类似。关键的区别在于 myRegExp 变量的 Global 属性被设置为 True：

```
myRegExp.Global = True
```

这说明 Execute()方法将返回所有找到的匹配项。对于每个匹配项，都会在 Matches 集合中保存一个相应的 Match 对象。如图 20-10 所示，此时包含五个匹配项。

20.2　使用 Match 对象和 Matches 集合

Matches 集合以及它所包含的 Match 对象，只能通过上一节介绍的 Execute()方法来创建(返回)。

每个 Match 对象都有三个只读属性：

- FirstIndex——匹配项中第一个字符的位置
- Length——匹配项的长度
- Value——匹配项的值

这些属性中包含着匹配项的值、匹配项中第一个字符的位置以及匹配的字符序列的长度等相关信息。

有关 FirstIndex 和 Value 属性的用法在上一节使用 Execute() 方法的例子中已经演示过了。

本例中使用测试文件 MatchLength.html，其内容如下：

```
<html>
<head>
<title>The Length Property of a Match Object</title>
<script language="vbscript" type="text/vbscript">
Function MatchLength
Dim myRegExp, TestName, Match, Matches, displayString
displayString = ""
Set myRegExp = new RegExp
myRegExp.Pattern = "[A-Z]\d+"
myRegExp.IgnoreCase = True
myRegExp.Global = True
TestString = InputBox("Enter characters and numbers in the text box below.")
Set Matches = myRegexp.Execute(TestString)
  For Each Match in Matches
   displayString = displayString & "Match found at position " & Match.FirstIndex &
"."
   displayString = displayString & "The match value is '" & Match.Value & "'." &
VBCrLf
  displayString = displayString & "Its length is " & Match.Length & "
characters." &VBCrLf & VBCrLf
  'displayString = ""
  Next
  MsgBox displayString
End Function

</script>
</head>
<body onload="MatchLength">

</body>
</html>
```

试一试：使用 Length 属性

(1) 在 Internet Explorer 中打开 MatchLength.html。

(2) 在输入框中输入字符序列 A9 B10 C110 D1123456 E1234567890 A3。

(3) 单击 OK 按钮，并观察消息框中显示的结果，如图 20-11 所示。

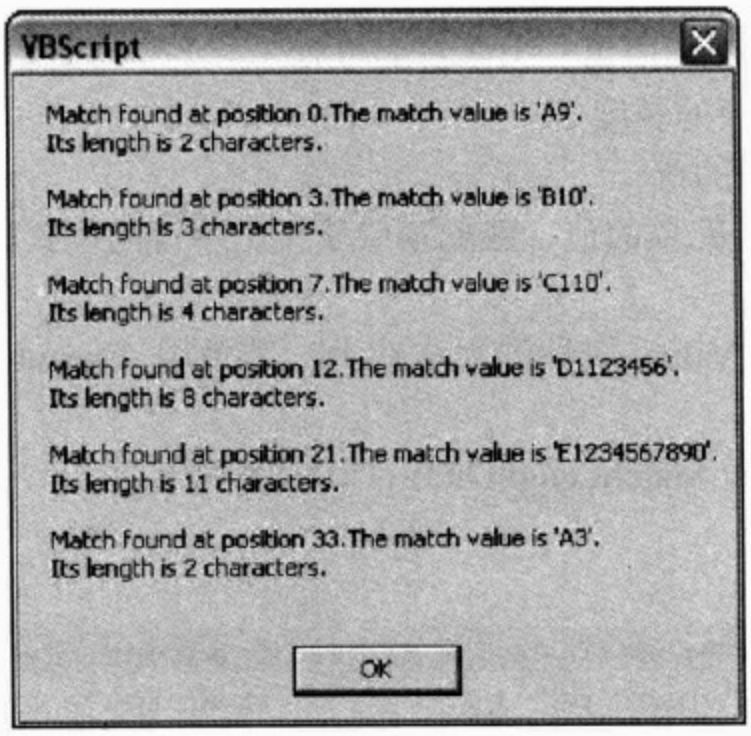


图 20-11

工作原理

当网页载入完成后，会调用 MatchLength 函数：

```
<body onload="MatchLength">
```

将正则表达式模式显示给 Pattern 属性：

```
myRegExp.Pattern = "[A-Z]\d+"
```

该模式匹配一个任意大小写的字母字符(因为 IgnoreCase 属性被设置为 True)，后跟一个或多个数字：

```
myRegExp.IgnoreCase = True
```

测试字符串 A9 B10 C110 D1123456 E1234567890 A3 中，包含 6 个匹配项。因为 myRegExp 变量的 Global 属性值被设置为 True，所以全部 6 个匹配项都会以 Matches 集合中的 Match 对象来表示：

```
myRegExp.Global = True
```

执行下面这行代码时 Matches 集合会被创建：

```
Set Matches = myRegexp.Execute(TestString)
```

本例中，Execute()方法返回的 Matches 集合中包含 6 个 Match 对象，每个对象在 For Each

循环中都以相同的方式被处理：

```
For Each Match in Matches
  displayString = displayString & "Match found at position " & Match.FirstIndex &
"."
  displayString = displayString & "The match value is '" & Match.Value & "'." &
VBCrLf
  displayString = displayString & "Its length is " & Match.Length & "
characters." &VBCrLf & VBCrLf
  'displayString = ""
Next
```

变量 displayString 的值开始是空字符串。每执行完一次 For Each 循环，相应匹配项的位置信息(保存在 FirstIndex 属性中)和匹配的字符序列的长度信息(保存在 Length 属性中)就被添加到 displayStirng 变量中。

最后，displayString 变量的值在消息框中被显示出来，这包含每个匹配项的位置和长度的信息：

```
MsgBox displayString
```

20.3　VBScript 支持的元字符

表 20-1 总结了 VBScript 支持的元字符。

表 20-1　VBScript 支持的元字符及其说明

元 字 符	说　明
^	匹配输入字符串的开始位置
$	匹配输入字符串的结束位置
?	限定符。匹配前面字符或组的零个或一个实例
*	限定符。匹配前面字符或组的零个或多个实例
+	限定符。匹配前面字符或组的一个或多个实例
{n,m}	限定符表示法。它匹配前面的字符或组最少 n 个，最多 m 个实例
(...)	分组的圆括号
(?: ...)	非分组(捕获)的圆括号
(?= ...)	肯定式向前查找
(?! ...)	否定式向前查找
|	交替选择
[...]	字符类。支持在字符类中使用范围
[^ ...]	取反的字符类
\b	匹配字母字符和非字母字符之间的边界。实际上，它匹配一个“单词”开始和结束的边界

(续表)

元 字 符	说 明
\B	匹配 \b 不匹配的位置
\d	匹配一个数字。等价于字符类 [0-9]
\D	匹配一个非数字。等价于字符类 [^0-9]
\s	匹配任何空白符
\S	匹配任何 \s 不匹配的字符
\t	匹配一个制表符
\w	匹配任何字母(构成单词的)字符。等价于字符类 [A-Za-z0-9_]
\W	匹配任何 \w 不匹配的字符。等价于字符类 [^A-Za-z0-9_]

20.3.1 限定符

VBScript 支持全部限定符——即 ?、 *、+ 元字符，以及 {n,m} 表示法。

而且，在 VBScript 中这些限定符的用法都是标准的。

20.3.2 位置元字符

VBScript 支持用 ^ 元字符，它匹配字符序列中第一个字符前的位置。也支持用 $ 元字符，它匹配字符序列中最后一个字符后的位置。

下面例子中使用测试文件 Positional.html，其内容如下：

```
<html>
<head>
<title>Positional Metacharacters</title>
<script language="vbscript" type="text/vbscript">
Dim myRegExp, TestString, displayString, MatchOrNot

Function FindMatch
displayString = ""
Set myRegExp = new RegExp
myRegExp.Pattern = "[A-Z]\d{2}"
myRegExp.IgnoreCase = False
myRegExp.Global = False
TestString = InputBox("Enter one alphabetic character and two numbers in the text
box below.")
MatchOrNot = myRegexp.Test(TestString)
If MatchOrNot Then
displayString = "When the pattern is '" & myRegExp.Pattern & "' the input '"_
 & TestString & "' contains a match."
Else
displayString = "When the pattern is '" & myRegExp.Pattern & "' the input '"_
 & TestString & "' does not contain a match."
End If
```

```
myRegExp.Pattern = "^[A-Z]\d{2}$"
MatchOrNot = myRegexp.Test(TestString)
If MatchOrNot Then
displayString = displayString & VBCrLf & "When the pattern is '" & myRegExp.Pattern
& "' the input '" _
  & TestString & "' contains a match."
Else
displayString = displayString & VBCrLf & "When the pattern is '" & myRegExp.Pattern
& "' the input '" _
& TestString & "' does not contain a match."

End If
MsgBox displayString
End Function

</script>
</head>
<body onload="FindMatch">

</body>
</html>
```

以上代码将根据两个模式来匹配输入框中输入的字符序列。第一个模式中不包含位置元字符 ^ 和 $。第二个模式中包含这两个元字符。

试一试：使用位置元字符

(1) 在 Internet Explorer 中打开 Positional.html。

(2) 在输入框中输入字符序列 A99。

(3) 单击 OK 按钮，并观察如图 20-12 所示的消息框中显示的信息。消息框中显示了在模式中没有位置元字符和同时带有两个位置元字符时尝试匹配的结果。这两种情况都找到了匹配项。

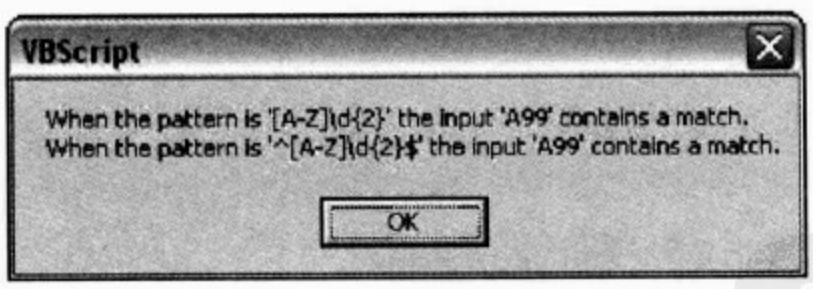


图　20-12

(4) 单击 OK 按钮退出消息框，然后按 F5 重新载入网页。

(5) 在文本框中输入字符序列 A999。

(6) 单击 OK 按钮，并观察如图 20-13 所示的消息框中显示的信息。此时，在模式中不包含位置元字符的情况下，找到了一个匹配项；而在包含位置元字符的情况下，没有找到匹配项。

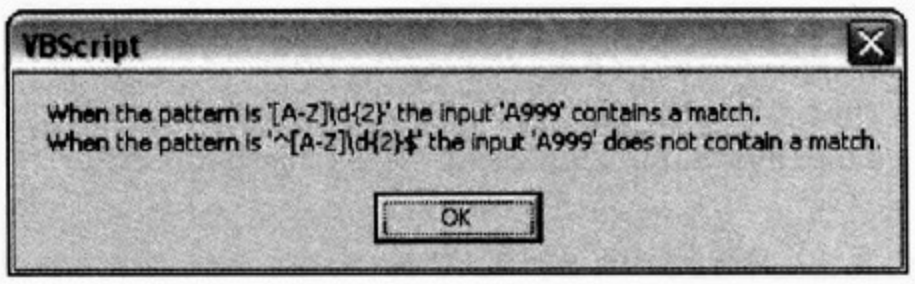


图 20-13

(7) 单击 OK 按钮退出消息框，再按 F5 重新载入网页。

(8) 在文本框中输入字符序列 A2A。

(9) 单击 OK 按钮，并观察如图 20-14 所示的消息框中显示的信息。此时，任何一个模式都没有匹配项。

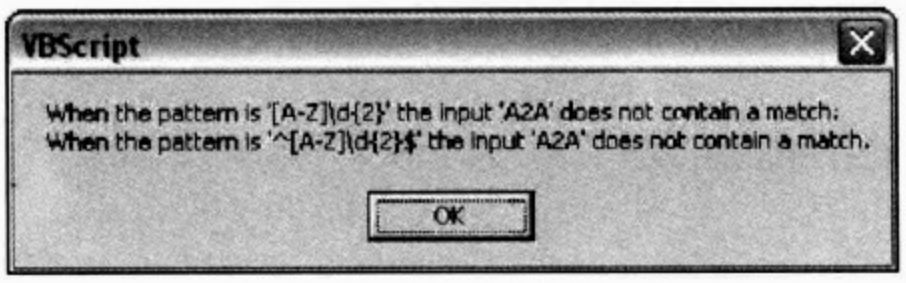


图 20-14

工作原理

当页面载入完成后，会调用 FindMatch 函数：

```
<body onload="FindMatch">
```

FindMatch 函数中使用了两次 RegExp 对象的 Test()方法来匹配用户输入的字符串。

开始时，指定给 Pattern 属性的模式是[A-Z]\d{2}，它匹配一个字母字符后跟两个数字：

```
myRegExp.Pattern = "[A-Z]\d{2}"
```

由于 IgnoreCase 和 Global 属性都被设置为 False，所以字母字符必须要使用大写形式，并只会进行一次匹配：

```
myRegExp.IgnoreCase = False
myRegExp.Global = False
```

用户输入的字符串被指定给变量 TestString：

```
TestString = InputBox("Enter one alphabetic character and two numbers in the
text box below.")
```

而以 TestString 变量作为参数的 Test()方法返回的结果(一个布尔值)被指定给变量 MatchOrNo。MatchOrNot 中要么会包含一个非零长度的字符串(等价于布尔值 True)，要么会包含一个空字符串(等价于布尔值 False)：

```
MatchOrNot = myRegexp.Test(TestString)
```

如果存在匹配项，模式与测试字符串将会被一起输出：

```
If MatchOrNot Then
displayString = "When the pattern is '" & myRegExp.Pattern & "' the input '"_
 & TestString & "' contains a match."
```

如果不存在匹配项，则会输出模式与输入的字符串不匹配的消息：

```
Else
displayString = "When the pattern is '" & myRegExp.Pattern & "' the input '"_
 & TestString & "' does not contain a match."
End If
```

现在，变量 displayString 中包含着第一次匹配的结果。此时，修改 Pattern 属性的值，使其包含 ^ 元字符和 $ 元字符，以便进行第二次匹配：

```
myRegExp.Pattern = "^[A-Z]\d{2}$"
```

同样，在第二次匹配完成后，Test()方法返回的布尔值仍然指定给变量 MatchOrNot：

```
MatchOrNot = myRegexp.Test(TestString)
```

如果存在匹配项，MatchOrNot 变量就包含一个等价于布尔值 True 的值，因此会将包含模式与匹配项的消息添加到 displayString 变量中：

```
If MatchOrNot Then
displayString = displayString & VBCrLf & "When the pattern is '" & myRegExp.Pattern
& "' the input '" _
 & TestString & "' contains a match."
```

但是，如果不存在匹配项，则会将不包含匹配项的消息添加到 displayString 变量中：

```
Else
displayString = displayString & VBCrLf & "When the pattern is '" & myRegExp.Pattern
& "' the input '" _
 & TestString & "' does not contain a match."
End If
```

最后，displayString 变量的值(包含用两个不同的 Pattern 属性值进行两次匹配的结果)会在消息框中显示出来：

```
MsgBox displayString
End Function
```

当用户输入测试字符串 A99 时，它匹配模式[A-Z]\d{2}，同时也匹配模式 ^[A-Z]\d{2}。

当用户输入测试字符串 A999 时，它匹配模式[A-Z]\d{2}，因为它包含一个字母字符后跟两位数字。但是，该字符串不匹配第二个模式 ^[A-Z]\d{2}$，因为它包含三位数字，这与模式所要求的在匹配 $ 元字符的行结束位置前应该有两位数字不符。

测试字符串 A2A 与任何一个模式都不匹配，因为它不像两个模式 [A-Z]\d{2} 和 ^[A-Z]\d{2}$ 所要求的那样，一个字母字符后跟两位数字。

20.3.3 字符类

VBScript 也支持所有的字符类。不过，在 VBScript 文档中称字符类为字符集。

因此，如果要匹配任何 A~L 的字符，可以使用字符类 [ABCDEFGHIJKL]。等价地，也可以使用包含范围的 [A-L]。

当然，也可以使用取反的字符类。比如，字符类 [^A-D]匹配除 A~D 外的任意字符。

20.3.4 词边界

VBScript 支持 \b 元字符匹配单词(字母)字符序列的开始或结束的位置。通常，由字母字符构成的序列在人们看来都是一个单词，但正则表达式却没有单词的概念。\b 元字符匹配下面任何一种情况：

- 匹配前面的字符包含在 [A-Za-z0-9_] 而后面的字符包含在 [^A-Za-z0-9_] 中的一个位置。等价于单词的结束位置。
- 匹配前面的字符包含在 [^A-Za-z0-9_] 而后面的字符包含在 [A-Za-z0-9_] 中的一个位置。等价于单词的开始位置。

20.3.5 向前查找

VBScript 支持向前查找。而且，支持肯定式向前查找和否定式向前查找。肯定式向前查找的语法是(?=the Lookahead)，而否定式向前查找的语法是(?!theNegativeLookahead)

下面的例子同时示范了肯定式向前查找和否定式向前查找。使用的测试文件是 Lookaheads.html，包含如下代码：

```
<html>
<head>
<title>Positive and Negative Lookahead</title>
<script language="vbscript" type="text/vbscript">
Function MatchLookaheads
Dim myRegExp, TestName, Match, Matches, displayString
displayString = ""
Set myRegExp = new RegExp
myRegExp.Pattern = "the(?=atre)" 'matches, for example, the in theatre
myRegExp.IgnoreCase = True
myRegExp.Global = True
TestString = InputBox("Enter characters and numbers in the text box below.")
Set Matches = myRegexp.Execute(TestString)
displayString = displayString & "MATCH ATTEMPT 1: 'the' in 'theatre'" & VBCrLf
  For Each Match in Matches
    displayString = displayString & "Match found at position " & Match.FirstIndex &
"."
    displayString = displayString & "The match value is '" & Match.Value & "'." &
VBCrLf
  Next
displayString = displayString & VBCrLf & VBCrLf
'Begin a new match which produces a new Match collection.
myRegExp.Pattern = "the(?!atre)" 'matches the NOT in theatre
```

```
Set Matches = myRegexp.Execute(TestString)
displayString = displayString & "MATCH ATTEMPT 2: 'the' not in 'theatre'" & VBCrLf
  For Each Match in Matches
    displayString = displayString & "Match found at position " & Match.FirstIndex &
"."
    displayString = displayString & "The match value is '" & Match.Value & "'." &
VBCrLf
  Next
  MsgBox displayString
End Function

</script>
</head>
<body onload="MatchLookaheads">

</body>
</html>
```

试一试：肯定式向前查找和否定式向前查找

(1) 在 Internet Explorer 中打开 Lookaheads.html。

(2) 在输入框中，输入字符序列 They love the theatre theatrically.。

(3) 单击 OK 按钮，并观察如图 20-15 所示的消息框中显示的结果。消息框中的 Match 1 部分是使用模式 the(?=atre)时的匹配结果，Match 2 部分则是使用模式 the(?!atre)时的匹配结果。

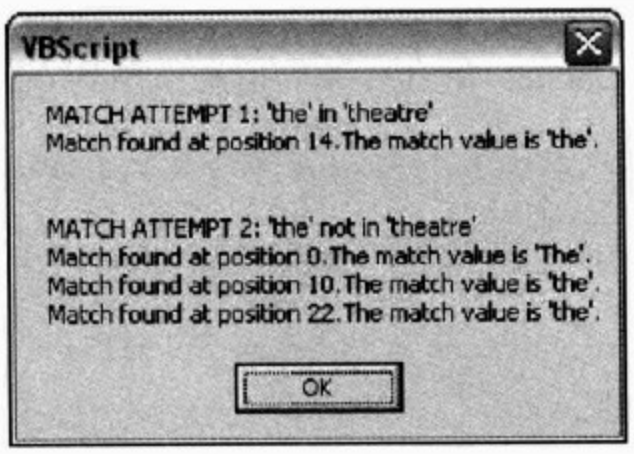


图　20-15

工作原理

当页面载入完成后，会调用函数 MatchLookaheads：

```
<body onload="MatchLookaheads">
```

其中使用了两次 RegExp 对象的 Execute()方法。第一次，Pattern 属性中的值如下：

```
myRegExp.Pattern = "the(?=atre)" 'matches, for example, the in theatre
```

然后执行 Execute()方法：

```
Set Matches = myRegexp.Execute(TestString)
```

此时，为 displayString 变量指定一个表示当前是第一次匹配的标签：

```
displayString = displayString & "MATCH ATTEMPT 1: 'the' in 'theatre'" & VBCrLf
```

对于 Matches 集合中的每一个 Match 对象(此时只有一个匹配项)，与其有关的信息都将被添加到 displayString 变量中：

```
For Each Match in Matches
  displayString = displayString & "Match found at position " & Match.FirstIndex &
"."
   displayString = displayString & "The match value is '" & Match.Value & "'." &
VBCrLf
  'displayString = ""
 Next
displayString = displayString & VBCrLf & VBCrLf
```

被匹配的模式是 the(?=atre)，它匹配后跟字符序列 atre 的字符序列 the。因为这是一个肯定式向前查找，所以它匹配 theatre 或 theatres 中 the。

然后开始进行第二次匹配。此时指定给 Pattern 属性的值是包含否定式向前查找的模式。这说明只有后面不是字符序列 atre 的字符序列 the 才能匹配：

```
myRegExp.Pattern = "the(?!atre)" 'matches the NOT in theatre
```

再次执行 Execute()方法意味着上一次返回的 Matches 集合被一个新集合所取代。不过，与上一次返回的集合有关的信息已经被捕获并保存在 displayString 变量中，以便稍后显示：

```
Set Matches = myRegexp.Execute(TestString)
```

现在，又把与第二个 Matches 集合有关的信息添加到 displayString 变量中：

```
displayString = displayString & "MATCH ATTEMPT 2: 'the' not in 'theatre'" & VBCrLf
  For Each Match in Matches
    displayString = displayString & "Match found at position " & Match.FirstIndex &
"."
   displayString = displayString & "The match value is '" & Match.Value & "'." &
VBCrLf

  Next
```

第一次匹配使用的模式 the(?=atre) 中包含肯定式向前查找。对于测试字符序列 They love the theatre theatrically. 而言，它匹配 theatre 中的字符序列 the。因此返回的 Matches 集合中包含一个 Match 对象。

第二次匹配使用的模式 the(?!atre)中包含否定式向前查找。对于测试字符序列 They love the theatre theatrically. 而言，会找到三个匹配项。因此，第二次返回的 Matches 集合中包含三个 Match 对象，分别是 They 中的 The、单词 the 和单词 theatrically 中的 the。由于匹配不区分大小写，所以 They 的首字母 T 仍然是大写的。

20.3.6　分组和非分组(捕获)的圆括号

VBScript 同时支持分组和非分组的圆括号。分组的圆括号写做：

```
(the Group)
```

而非分组的圆括号写做：

```
(?:not Grouped)
```

下面的例子修改了测试文件 ReverseName.html 中的模式，因而不再捕获名与姓之间的空白符。

相关的测试文件是 NonGrouping.html，其中的代码如下：

```
<html>
<head>
<title>Reverse Surname and First Name, using non-grouping parentheses</title>
<script language="vbscript" type="text/vbscript">
Function ReverseName2
Dim myRegExp, TestName, Match
Set myRegExp = new RegExp
myRegExp.Pattern = "(\S+)(?:\s+)(\S+)"
TestString = InputBox("Enter your name below, in the form" & VBCrLf & _
  "first name, then a space then last name." & VBCrLf & "Don't enter an initial or
middle name.")
Match = myRegexp.Replace(TestString, "$2, $1")
If Match <> "" Then
  MsgBox "Your name in last name, first name format is:" & VBCrLf & Match
Else
  MsgBox "You didn't enter your name." & VBCrLF & "Press OK then F5 to run the
example again."
End If
End Function

</script>
</head>
<body onload="ReverseName2">

</body>
</html>
```

试一试：分组和非分组的圆括号

(1) 在 Internet Explorer 中打开 NonGrouping.html。

(2) 在输入框中输入字符串 John Smith。

(3) 单击 OK 按钮，并观察消息框中显示的信息，如图 20-16 所示。结果与使用测试文件 ReverseName.html 时的结果相同。但是后台模式中的分组却不一样，“工作原理”部分会对此进一步做出解释。

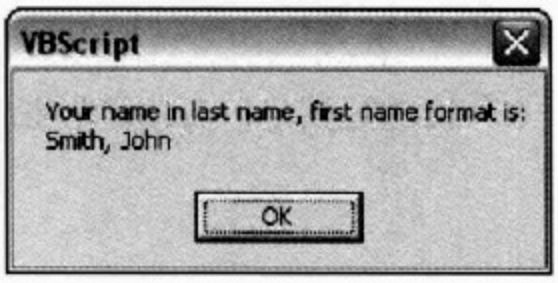


图 20-16

工作原理

本例中使用的 ReverseName2 函数不同于本章前面例子中的 ReverseName 函数，关键的区别在于：它在定义模式时，在名和姓之间使用了非分组的圆括号来匹配用户输入的空白符：

```
myRegExp.Pattern = "(\S+)(?:\s+)(\S+)"
```

这意味着匹配并捕获的名会被保存在 $1 中(跟以前一样)，而姓则被保存在 $2 中(以前是被保存在 $3 中)。因此，当使用 Replace() 方法时，必须对替换模式进行相应调整，以确保首先显示姓(被保存在$2)，然后是逗号和空格符，然后才显示名(被保存在$1)：

```
Match = myRegexp.Replace(TestString, "$2, $1")
```

20.4 练习

1. 请修改测试文件 TestForA.html 中的模式，用它来测试用户是否输入格式为 MM/DD/YYYY 的日期值。

2. 请修改测试文件 ReverseName.html 中的模式，使其只匹配一个字母字符序列后跟一个或多个空白符，再跟一个字母字符序列。如果用户在输入框中输入了额外的字符，那么应该给用户一个错误提示，让用户按照规定的格式输入。提示：不仅要修改 Pattern 属性的值，而且也要修改在没有找到匹配项时要显示的信息内容。

第21章

Visual Basic .NET 与正则表达式

Microsoft Visual Basic .NET 提供了功能强大且富有灵活性的正则表达式功能。Visual Basic .NET 对正则表达式的支持与前面介绍的 VBScript 对正则表达式的支持，既存在十分相似的地方，也有比较大的区别。总而言之，Visual Basic .NET 对正则表达式的支持更强大而且更灵活。

在 Visual Basic .NET 中进行正则表达式编程的基础构建于 System.Text.RegularExpressions 命名空间之上，该命名空间属于 .NET 架构类库(Framework Class Library)的一部分。

在本章中将学习以下内容：

- 如何使用包含在 System.Text.RegularExpressions 命名空间中的类和对象
- Visual Basic .NET 支持的元字符的含义

本章中出现的例子都在 Visual Studio 2003 和 .NET Framework 1.1 中测试通过。假设读者有权访问一个 Visual Studio 2003 的副本，并具有使用 Visual Basic .NET 的基本常识。本章不会详细介绍如何使用 Visual Studio .NET 2003。不过，如果读者没有访问 Visual Studio 2003 某个副本的权限，可以运行测试文件中提供的 .exe 文件的副本，但不能查看和编辑其中的 Visual Basic .NET 代码。

21.1 System.Text.RegularExpressions 命名空间

Visual Basic .NET 中的正则表达式功能包含在 System.Text.RegularExpressions 命名空间的对象中。在介绍具体的类之前，先看一个非常简单的 Visual Basic .NET 正则表达式的例子。

在 Visual Basic .NET 中，可以通过几种方式使用正则表达式，下面的例子使用其中一种方式实现了简单的匹配。

21.1.1 一个简单的 Visual Basic .NET 的例子

这个例子测试用户输入的字符串是否与一个直接量正则表达式模式 Fred 匹配。

测试文件是 FindFred 项目中的 Module1.vb，它包含如下代码：

```
Imports System.Text.RegularExpressions
Module Module1

    Sub Main()
        Dim myInput, myRegex
        myRegex = New Regex("Fred")
        Console.WriteLine("Enter a test string")
        Console.WriteLine("Then press the Enter key to continue.")
        myInput = Console.ReadLine()
        Console.WriteLine("The string you entered was: " & myInput)
        Console.WriteLine("The match is: " & myRegex.Match(myInput).Value)
        Console.WriteLine("Press the Return key to continue.")
        Console.ReadLine()
    End Sub
End Module
```

在某些例子中，原始文件中写在一行的代码可能会在印刷的页面中变成多行。这种差异是由印刷页面的宽度限制所决定的。

下面步骤中的指示假设电脑中已安装 Visual Studio 2003。

(1) 打开 Visual Studio 2003，在 File 菜单中选择 New，然后选择 Project。

图 21-1 显示的是打开的对话框界面。其中第 2 步~第 4 步中要选的选项已经选择完成了。

(2) 在如图 21-1 所示的对话框界面的 Project Type 窗格中，选择 Visual Basic Projects。

(3) 在 Templates 窗格中选择 Console Application。然后，在 Name 文本框中输入文本 FindFred。

(4) 在 Location 文本框中，输入 C:\BRegExp\Ch21。也可以选择其他位置保存项目文件。单击 OK 按钮。

图 21-2 显示的是在单击 OK 按钮后的屏幕外观。由于 Visual Studio 2003 中业已存在的自定义设置不同，真正的界面可能会与插图中的界面不大一样。

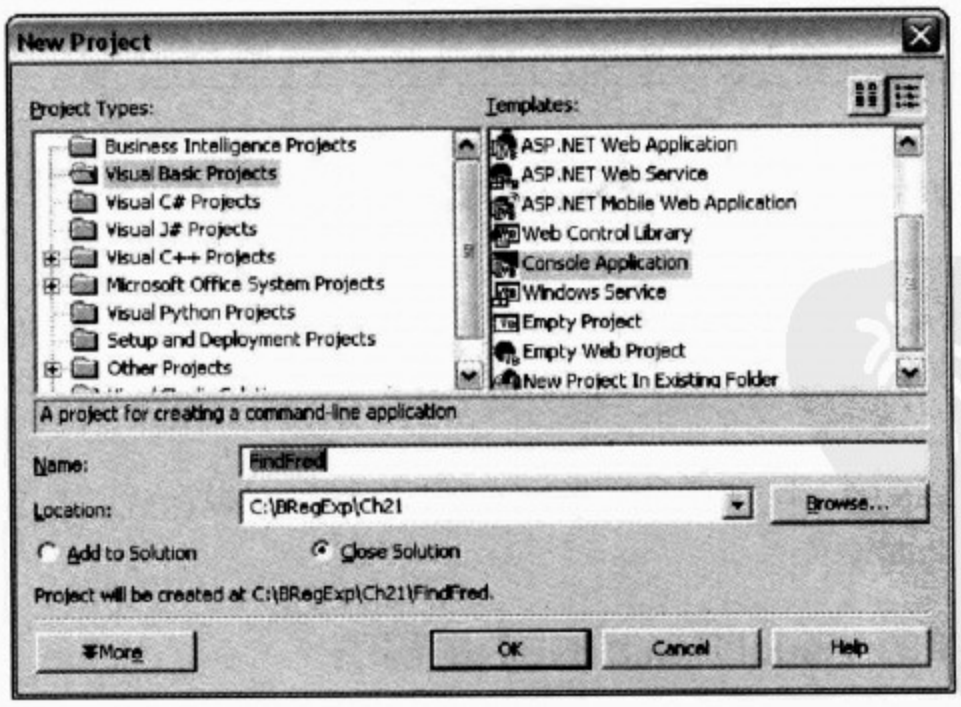


图 21-1

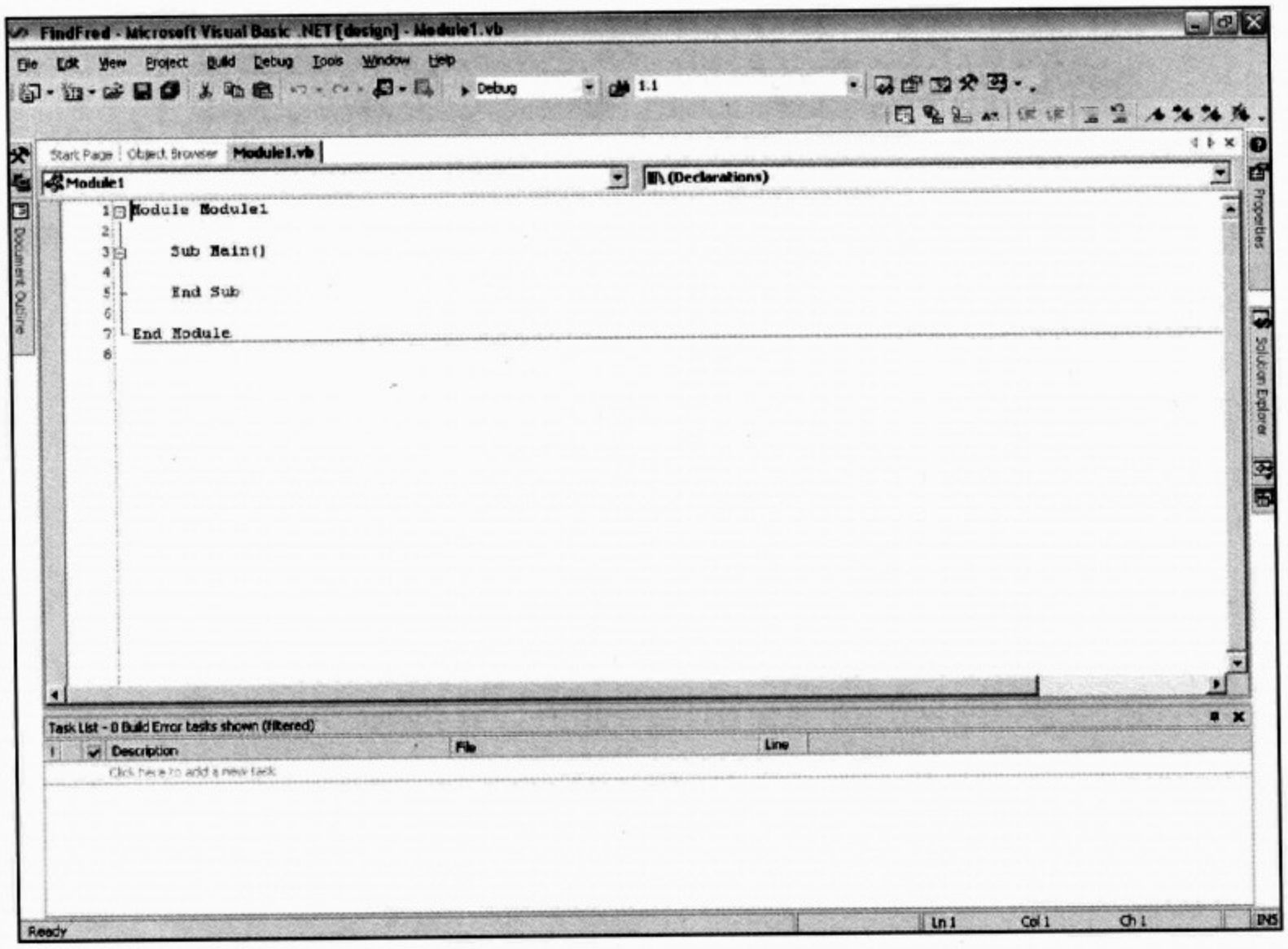


图　21-2

Regex 对象属于 System.Text.RegularExpressions 命名空间。使用 Imports 语句引入 System.Text.RegularExpressions 命名空间，可以使代码更简洁。

(5) 在代码窗口中添加下列代码，注意将它写在包含 Module Module1 的行之前：

```
Imports System.Text.RegularExpressions
```

(6) 在包含 Sub Main()的行后面添加下列代码：

```
Dim myInput, myRegex
myRegex = New Regex("Fred")
Console.WriteLine("Enter a test string")
Console.WriteLine("Then press the Enter key to continue.")
myInput = Console.ReadLine()
Console.WriteLine("The string you entered was: " & myInput)
Console.WriteLine("The match is: " & myRegex.Match(myInput).Value)
Console.WriteLine("Press the Return key to continue.")
Console.ReadLine()
```

(7) 打开 File 菜单，选择 Save All(也可以使用 Ctrl+Shift+S 快捷键)。

(8) 按 F5 键运行代码。如果输入的代码正确，应该看到如图 21-3 所示的控制台窗口。

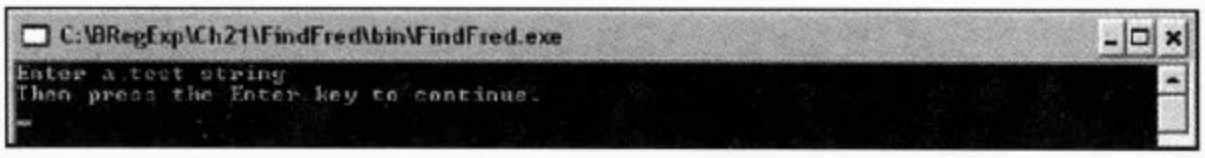


图 21-3

(9) 在命令行中，输入 Does anyone here know Fred?，然后按回车键并观察显示的信息，如图 21-4 所示。

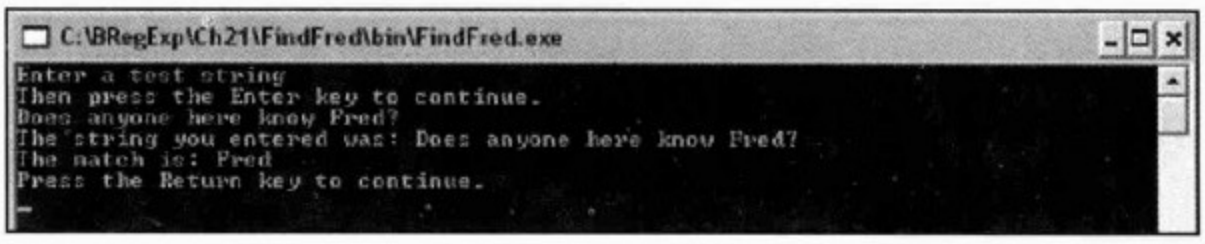


图 21-4

工作原理

第 1 步~第 4 步创建了一个控制台(命令窗口)应用程序的框架。

第 5 步添加的代码可以使我们直接通过类名引用包含在 System.Text.Regular Expressions 命名空间中的类，而不用使用完整的限定名，如 System.Text.RegularExpressions.Regex：

```
Imports System.Text.RegularExpressions
```

在这个例子中，只使用了一次 Regex 对象，引入 System.Text.RegularExpressions 命名空间似乎并没有节省时间。但如果是在更复杂的例子中，引入命名空间的确有助于节省开发人员的时间，因为可以这样编写代码：

```
myRegex = New Regex("Fred")
```

而不用再写成：

```
myRegex = New System.Text.RegularExpressions.Regex("Fred")
```

以上两行代码的含义相同，但第一行代码更少而且更具可读性。

添加到 Sub Main() 内部的代码定义了这个控制台程序的功能。首先，定义两个变量 myInput 和 myRegex。如果 OPTION STRICT 被设置为 ON，那么还必须要指定每个变量的类型：

```
Dim myInput, myRegex
```

接着，通过给构造函数传递直接量模式 Fred 实例化一个 Regex 对象。实例化一个 Regex 对象是在 Visual Basic .NET 中使用正则表达式功能的方式之一：

```
myRegex = New Regex("Fred")
```

然后，使用两次 Console.WriteLine()，向用户显示简单的指令：

```
Console.WriteLine("Enter a test string")
Console.WriteLine("Then press the Enter key to continue.")
```

当用户按下回车键时，Console.ReadLine()方法会从命令行中读取用户的输入，并将输入的值指定给变量 myInput：

```
myInput = Console.ReadLine()
```

随后，包含在变量 myInput 中(用户输入)的字符串，又通过 Console.WriteLine()被发送回命令行窗口：

```
Console.WriteLine("The string you entered was: " & myInput)
```

Regex 对象的 Match()方法用于匹配保存在 myInput 变量中的字符序列。而 Console.WriteLine()方法则用于显示一些说明性的文本以及匹配模式(本例中，Fred)的结果：

```
Console.WriteLine("The match is: " & myRegex.Match(myInput).Value)
```

然后显示一条信息告诉用户如何继续运行程序：

```
Console.WriteLine("Press the Return key to continue.")
```

让命令行窗口等待用户输入，可以使前次程序的输出在用户准备退出程序之前始终显示：

```
Console.ReadLine()
```

21.1.2　System.Text.RegularExpressions 中的类

表 21-1 中列出了包含于 System.Text.RegularExpressions 命名空间中的类。其中列出的类的属性和方法将在本章后面详细介绍，同时还会提供如何使用这些属性和方法的例子。

表 21-1　System.Text.RegularExpressions 中的类

类	说　明
Capture	表示由一对圆括号包围的子表达式所捕获的文本
CaptureCollection	表示一个 Capture 对象的集合
Group	表示由一对圆括号构成的单个捕获组的结果
GroupCollection	表示一个 Group 对象的集合
Match	表示单个正则表达式匹配的结果
MatchCollection	表示一个 Match 对象的集合
Regex	该类中包含正则表达式模式
RegexCompilationInfo	提供编译器用于将正则表达式编译为独立程序集的信息

可以把 Regex 对象想象为一个包含与正则表达式有关的所有信息的对象。而 MatchCollection 对象则包含匹配过程发现的所有匹配项，以及包含在 Match 对象中的有关每个匹配项的信息。GroupCollection 对象则包含一次匹配过程中所有组的信息，而且每个组都以一个 Group 对象表示。CaptureCollection 对象包含一个组中所有捕获的信息，以及被保存在 Capture 对象中的每个捕获的信息。

下面几节分别详细讨论 System.Text.RegularExpressions 命名空间中的这几个成员。

在下面几节中，对类将简单地称呼其类名(如，Regex)，而不使用完整的限定名，比如 System.Text.RegularExpressions.Regex。

21.1.3 Regex 对象

通过实例化 Regex 对象，就能够以编程的方式访问和操纵其属性和方法。另外，还可以使用 Regex 类的三个共享方法。有关这三个共享方法的作用将在稍后介绍并示范。

Regex 对象有两个公共属性，如表 21-2 所示。

表 21-2 Regex 对象的两个公共属性

属　性	说　明
Options	包含传递给 Regex 对象的选项信息
RightToLeft	返回一个布尔值，表示是否采取了从右往左的搜索方式。True 表示从右往左的匹配

如果已经习惯于在 JScript 或 VBScript 中创建正则表达式对象，即使用 RegExp 对象，那么在拼写 .NET 中的 Regex 对象时一定要多加注意。

表 21-3 总结了 Regex 对象的方法。其中一些 Regex 对象的概念和技术在标准的正则表达式中是没有的。因此，表 21-3 中相关概念的摘要信息将有助于在后面学习这些方法及其用途时有一个清晰的认识。

表 21-3 Regex 对象的方法

方　法	说　明
CompileToAssembly	将正则表达式编译为一个独立程序集(默认行为是不把正则表达式编译为一个独立程序集)
Equals	确定两个对象是否相等
Escape	转义一组元字符，即用相应的转义字符替换元字符
GetGroupNames	返回一个包含捕获组名称的数组
GetGroupNumbers	返回一个包含捕获组编号的数组
GetHashCode	从 Object 类继承的方法
GetType	取得当前实例的类型
GroupNameFromNumber	取得与作为参数提供的一个组编号所对应的组名称
GroupNumberFromName	取得与作为参数提供的一个组名称所对应的组编号
IsMatch	返回表示正则表达式模式与作为参数传递给 IsMatch() 方法的字符串是否匹配的布尔值
Match	返回零个或一个 Match 对象，取决于作为参数传递给该方法的字符串中是否包含一个匹配项

方　法	说　明
Matches	返回一个包含零个或多个 Match 对象的 MatchCollection 对象，其中包含作为 Matches()方法参数的字符串中的所有匹配项(或一个也没有)
Replace	用指定的字符序列替换符合某个正则表达式模式的所有匹配项
Split	将输入字符串拆分成一个子字符串数组，拆分的位置由一个正则表达式模式指定
ToString	返回一个字符串，其中包含由构造函数传递给 Regex 对象的正则表达式模式
Unescape	取消输入字符串中的任何转义字符

1. 使用 Match 对象和 Matches 集合

如果存在一个匹配项，则 Match()方法会返回一个 Match 对象。如果有多个潜在匹配项，Match()只匹配一次后就停止。

试一试：使用 Match() 方法

下面的测试代码包含在 MatchMethodDemo 项目的 Module1.vb 文件中：

```
Imports System.Text.RegularExpressions
Module Module1
    Dim myRegex = New Regex("[A-Z]\d")
    Sub Main()
        Console.WriteLine("Enter a string on the following line:")
        Dim inputString = Console.ReadLine()
        Dim myMatch = myRegex.Match(InputString)
        Console.WriteLine("The match, '" & myMatch.Value & "' was found.")
        Console.WriteLine("Press Return to close this application.")
        Console.ReadLine()
    End Sub

End Module
```

(1) 在 Visual Studio 2003 中创建一个新项目。如果从未创建过新项目，请参考本章第一个例子中的详细说明。

(2) 将这个项目命名为 MatchMethodDemo。

(3) 编辑 Module1.vb，使其内容如前面代码所示。

(4) 保存这个项目并按 F5 运行它。如果输入的代码正确，应该看到一个显示有下列文本的命令窗口：

```
Enter a string on the following line:
```

(5) 输入文本 Hello K9 and K10，然后按回车，观察结果。

输入的字符串中有两个与模式 [A-Z]\d 匹配的潜在匹配项：K9 和 K1。图 21-5 中则

只显示了一个匹配项——K9，它是输入字符串中的第一个匹配项。

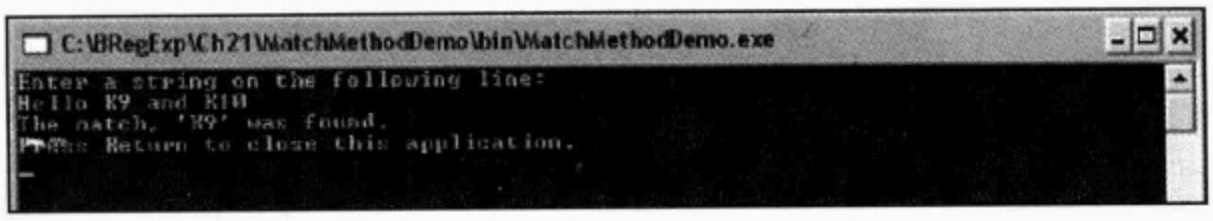


图 21-5

工作原理

首先定义变量 myRegex 并将以模式 [A-Z]\d 实例化的 Regex 对象指定给它，这个模式匹配一个大写的字母字符后跟一个数字：

```
Dim myRegex = New Regex("[A-Z]\d")
```

从命令行中读取到测试字符串后，把 myRegex 变量的 Match() 方法应用到 InputString，而将结果指定给变量 myMatch：

```
Dim myMatch = myRegex.Match(InputString)
```

变量 InputString 的值是字符串 Hello K9 and K10.。对于模式 [A-Z]\d 而言，该字符串中存在两个匹配项：字符序列 K9 和 K1。

一些说明性的文本与 myMatch 对象的 Value 属性连接起来构成要显示的消息：

```
Console.WriteLine("The match, '" & myMatch.Value & "' was found.")
```

此时，只显示第一个匹配项 K9。第二个潜在的匹配项 K1 没有匹配也没有显示。如果要匹配并显示测试字符串中的所有匹配项，需要使用 Regex 对象的 Matches() 方法。

不过，Matches() 方法返回的是一个 MatchCollection 对象，该对象会包含与测试字符串中所有匹配项对应的 Match 对象。在下面的例子中，我们来看一看如何使用 Matches() 方法。

试一试：使用 Matches() 方法

本例的测试代码包含在 MatchesMethodDemo 项目中的 Module1.vb 文件中：

```
Imports System.Text.RegularExpressions
Module Module1
    Sub Main()
        Dim myRegex = New Regex("[A-Z]\d")
        Console.WriteLine("Enter a string on the following line:")
        Dim inputString = Console.ReadLine()
        Dim myMatchCollection = myRegex.Matches(inputString)
        Console.WriteLine()
        Console.WriteLine("There are {0} matches.", myMatchCollection.Count)
        Console.WriteLine()
        Dim myMatch As Match
        For Each myMatch In myMatchCollection
        Console.WriteLine("At position {0}, the match '{1}' was found",
```

```
    myMatch.Index, myMatch.ToString)
            Next
            Console.WriteLine()
            Console.WriteLine("Press Return to close this application.")
            Console.ReadLine()
      End Sub
    End Module
```

(1) 在 Visual Studio 2003 中创建一个新的 Visual Basic .NET 控制台项目，并命名为 MatchesMethodDemo。

(2) 编辑 Module1.vb 中的代码，使其内容如前面的代码所示。

(3) 按 Ctrl+S 快捷键保存代码，然后按 F5 运行。

(4) 在命令窗口中，输入 Hello K9, K10 and K21.，然后按回车键。

(5) 观察如图 21-6 所示的结果。现在有三个匹配项：K9、K1 和 K2。

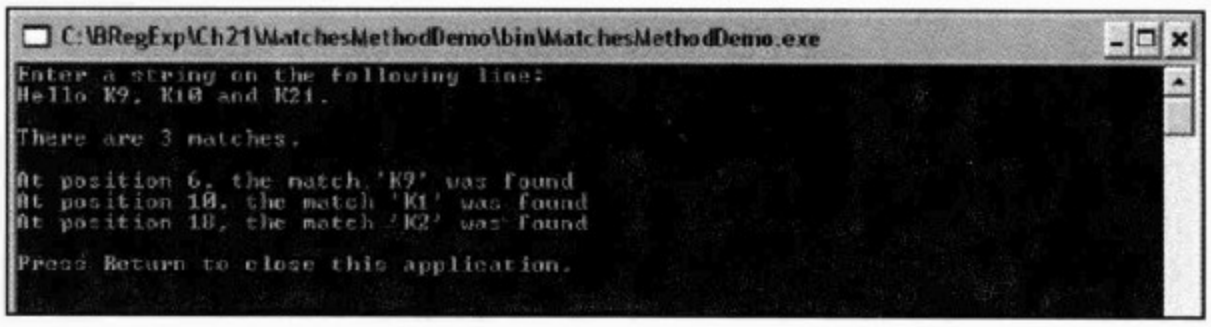


图　21-6

工作原理

首先，定义变量 myRegex，并将通过模式 [A-Z]\d 实例化的一个新 Regex 对象指定给该变量：

```
Dim myRegex = New Regex("[A-Z]\d")
```

在邀请用户输入字符串后，会声明一个 myMatchCollection 变量，然后将以 inputString 作为参数的 Matches()方法返回的匹配项集合指定给该变量：

```
Dim myMatchCollection = myRegex.Matches(inputString)
```

通过(由 myMatchCollection 引用的)MatchCollection 对象的 Count 属性，显示 Match 对象的数量：

```
Console.WriteLine("There are {0} matches.", myMatchCollection.Count)
```

将 myMatch 变量定义为一个 Match 对象：

```
Dim myMatch As Match
```

然后，使用 For Each 循环以相同的方式处理 myMatchCollection 变量中的每个 Match 对象：

```
For Each myMatch In myMatchCollection
```

使用 Match 对象的 Index 属性显示每个匹配项在字符串中的位置。而 Match 对象的 ToString()方法则返回一个包含在 Match 对象中的与模式匹配的字符序列：

```
        Console.WriteLine("At position {0}, the match '{1}' was found",
myMatch.Index, myMatch.ToString)
        Next
```

2. 使用 Match.Success 属性和 Match.NextMatch 方法

刚才介绍的 MatchCollection 对象提供了一种迭代测试字符串中匹配项的方法。另一种循环遍历测试字符串中所有匹配项的方法是使用 Match 对象的 Success 属性结合 Match 对象的 NextMatch 方法。

试一试：使用 Match.Success 属性和 Match.NextMatch 方法

NextMatchDemo 项目的 Module1.vb 文件中的内容如下：

```
Imports System.Text.RegularExpressions
Module Module1

    Sub Main()
        Dim myRegex = New Regex("[A-Z]\d")
        Console.WriteLine("Enter a string on the following line:")
        Dim inputString = Console.ReadLine()
        Dim myMatch = myRegex.Match(inputString)
        Dim myMatchCount As Integer
        Console.WriteLine("The first match is {0}.", myMatch.ToString)
        Do While myMatch.Success
            myMatchCount += 1
            Console.WriteLine("At position {0}, the match '{1}' was found",
myMatch.Index, myMatch.ToString)
            myMatch = myMatch.NextMatch
        Loop
        Console.WriteLine("There were {0} matches.", myMatchCount)
        Console.WriteLine()
        Console.WriteLine("Press Return to close this application.")
        Console.ReadLine()

    End Sub

End Module
```

(1) 打开 Visual Studio，创建一个名为 NextMatchDemo 的新项目。

(2) 在 Module1.vb 的代码窗口中，编辑其中的内容使其如前面的 Module1.vb 文件所示。

(3) 保存 Module1.vb 文件，并按 F5 运行这些代码。

(4) 在命令行中输入测试文本 Hello K9, K10 and K21.，然后按回车键。观察如图 21-7 所示的结果。

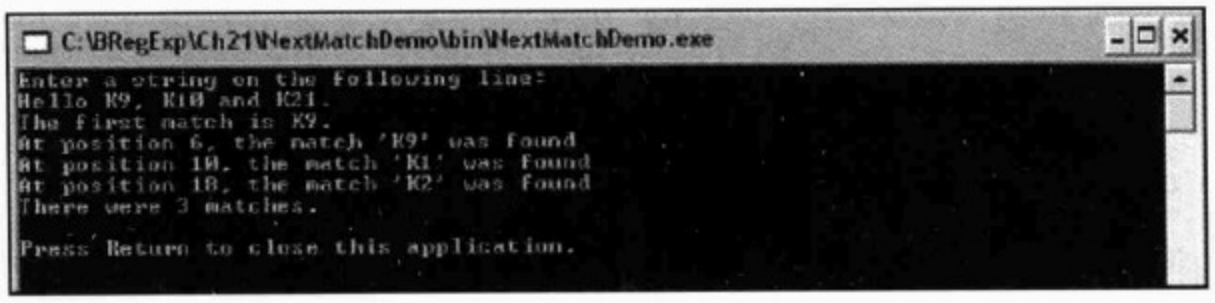


图　21-7

工作原理

本例使用了与前面例子中相同的正则表达式模式来实例化了一个 Regex 对象，并将该对象指定给 myRegex 变量：

```
Dim myRegex = New Regex("[A-Z]\d")
```

测试字符串仍然来自用户的输入，输入的字符串指定给变量 inputString。然后，使用 myRegex 的 Match() 方法将 inputString 与模式 [A-Z]\d 进行匹配：

```
Dim myMatch = myRegex.Match(inputString)
```

定义变量 myMatchCount，并用它来记录匹配项的数量：

```
Dim myMatchCount As Integer
```

输出第一个匹配项的值：

```
Console.WriteLine("The first match is {0}.", myMatch.ToString)
```

根据 myMatch 变量的 Success 属性，使用 Do While 循环来处理每一个成功的匹配项：

```
Do While myMatch.Success
```

循环计数器变量 myMatchCount 每次加 1：

```
myMatchCount += 1
```

然后，使用 Index 属性和 ToString()方法来显示匹配项的位置和值：

```
Console.WriteLine("At position {0}, the match '{1}' was found",
myMatch.Index, myMatch.ToString)
```

最后使用 NextMatch()测试是否还存在其他匹配项。如果存在另一个匹配项，那么会对 Do While 语句中的 myMatch.Success 求值并返回布尔值 True，循环会重新开始。如果不存在其他匹配项，myMatch.Success 会返回布尔值 False，因而退出循环：

```
myMatch = myMatch.NextMatch
Loop
```

退出循环后，使用 myMatchCount 变量的值来显示匹配项的数量：

```
Console.WriteLine("There were {0} matches.", myMatchCount)
```

这种技术为迭代遍历一个测试字符串中存在的匹配项集合时使用 MatchCollection 对象提供了另一种替代方案。

21.1.4 GroupCollection 和 Group 类

本章前面几个例子中使用的模式都非常简单，没有包含模式中常用的圆括号，而圆括号在模式中用于创建组。一次匹配过程中的所有组都保存在 GroupCollection 对象中。而集合中的每个组则保存在一个 Group 对象中。

试一试：使用 GroupCollection 和 Group 类

GroupsDemo 项目中的文件 Module1.vb 中包含如下测试代码：

```
Imports System.Text.RegularExpressions
Module Module1
    Sub Main()
        Dim myRegex = New Regex("([A-Z])(\d+)")
        Console.WriteLine("Enter a string on the following line:")
        Dim inputString = Console.ReadLine()
        Dim myMatchCollection = myRegex.Matches(inputString)
        Console.WriteLine()
        Console.WriteLine("There are {0} matches.", myMatchCollection.Count)
        Console.WriteLine()
        Dim myMatch As Match
        Dim myGroupCollection As GroupCollection
        Dim myGroup As Group
        For Each myMatch In myMatchCollection
            Console.WriteLine("At position {0}, the match '{1}' was found",
myMatch.Index, myMatch.ToString)
            myGroupCollection = myMatch.Groups
            For Each myGroup In myGroupCollection
                Console.WriteLine("Group containing '{0}' found at position
'{1}'.", myGroup.Value, myGroup.Index)
            Next
            Console.WriteLine()
        Next
        Console.WriteLine()
        Console.WriteLine("Press Return to close this application.")
        Console.ReadLine()
    End Sub
End Module
```

(1) 在 Visual Studio 2003 中创建一个新项目，并命名为 GroupsDemo。
(2) 在 Module1.vb 的代码窗口中，输入以上所示的测试代码。
(3) 保存代码并按 F5 运行这些代码。
(4) 在命令窗口中，输入字符串 Hello K9, K10, K21 and K999。
(5) 按回车键，并观察如图 21-8 所示的结果。

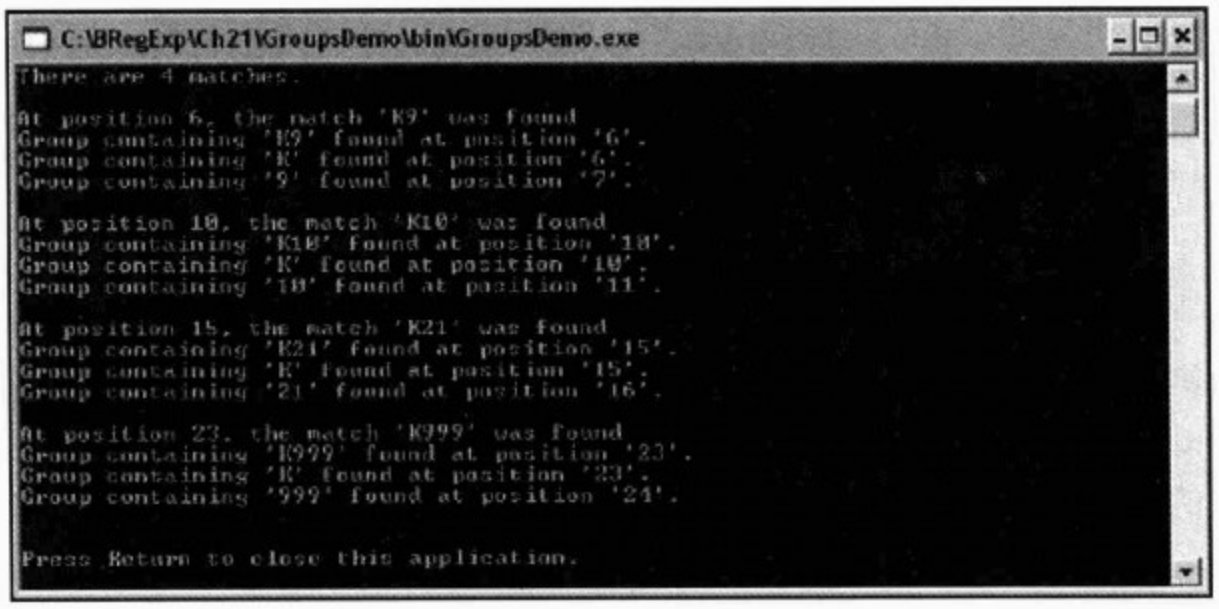


图　21-8

工作原理

本例中的正则表达式模式包含两对圆括号。而且，模式将匹配一个或多个数字，而不再是匹配一个数字：

```
Dim myRegex = New Regex("([A-Z])(\d+)")
```

与前面的例子一样，也是请用户输入测试字符串，然后使用 Regex 对象的 Matches() 方法来匹配包含在 myRegex 中的模式匹配变量 inputString：

```
Dim myMatchCollection = myRegex.Matches(inputString)
```

定义一个 GroupCollection 类型的变量 myGroupCollection：

```
Dim myGroupCollection As GroupCollection
```

将变量 myGroup 定义为一个 Group 对象：

```
Dim myGroup As Group
```

使用两个 For Each 循环。外部的 For Each 循环负责循环遍历 MatchCollection 集合中的 Match 对象：

```
For Each myMatch In myMatchCollection
```

对每一个 Match 对象，分别使用其 Index 属性和 ToString()方法来显示该匹配项的位置和值：

```
        Console.WriteLine("At position {0}, the match '{1}' was found",
myMatch.Index, myMatch.ToString)
```

将 Match 对象的 Groups 属性指定给 myGroupCollection 变量：

```
myGroupCollection = myMatch.Groups
```

内部的 For Each 循环用来处理包含在 GroupCollection 集合中的每一个 Group 对象：

```
For Each myGroup In myGroupCollection
```

使用 Group 对象的 Value 属性和 Index 属性来显示每个组的值和位置信息：

```
                Console.WriteLine("Group containing '{0}' found at position
'{1}'.", myGroup.Value, myGroup.Index)
            Next
            Console.WriteLine()
        Next
```

输出的第一个组是包含与整个正则表达式模式匹配的组。该组存在于所有成功匹配的过程中。GroupCollection 集合中的另外的一些组对应着正则表达式模式中成对的圆括号。因为本例中的正则表达式模式中包含两对圆括号，因此每个 GroupCollection 集合中包含三个组。通过图 21-8 可以看出来，每次匹配都显示三个组。

21.1.5 CaptureCollection 和 Capture 类

CaptureCollection 对象中包含由一个或多个 Capture 对象组成的集合。每个 Capture 对象表示由成对圆括号构成的一个捕获组的内容。

Capture 对象实例没有公共构造函数。而且，Capture 对象是不可变的(immutable)。它只能通过匹配过程创建，而每一个 Capture 对象都是 CaptureCollection 集合的一个元素。

试一试：使用 CaptureCollection 和 Capture 类

本例中的测试代码包含在 CapturesDemo 项目中的文件 Module1.vb 中：

```
Imports System.Text.RegularExpressions
Module Module1
    Sub Main()
        Dim myRegex = New Regex("([A-Z])+(\d)+")
        Console.WriteLine("Enter a string on the following line:")
        Dim inputString = Console.ReadLine()
        Dim myMatchCollection = myRegex.Matches(inputString)
        Console.WriteLine()

        Console.WriteLine("There are {0} matches.", myMatchCollection.Count)
        Console.WriteLine()
        Dim myMatch As Match
        Dim myGroupCollection As GroupCollection
        Dim myGroup As Group
        For Each myMatch In myMatchCollection
            Console.WriteLine("At position {0}, the match '{1}' was found",
                myMatch.Index, myMatch.ToString)
            Console.WriteLine("This match has {0} groups.", myMatch.Groups.Count)
            myGroupCollection = myMatch.Groups
            For Each myGroup In myGroupCollection
                Dim myCaptureCollection As CaptureCollection = myGroup.Captures
            Dim myCapture As Capture
            Console.WriteLine("Group containing '{0}' found at position
                '{1}'.", myGroup.Value, myGroup.Index)
            For Each myCapture In myCaptureCollection
```

```
                Console.WriteLine(" Capture: '{0}' at position '{1}'.",
                  myCapture.Value, myCapture.Index)
                  Next
              Next
              Console.WriteLine()
          Next
          Console.WriteLine()
          Console.WriteLine("Press Return to close this application.")
          Console.ReadLine()
      End Sub
End Module
```

(1) 在 Visual Studio 2003 中基于控制台程序模板创建一个新项目，并命名为 Captures Demo。

(2) 在默认模块的代码窗口中输入前面的代码。保存代码，并按 F5 运行代码。

(3) 在命令窗口中，输入测试字符串 ABC1 A123。

(4) 按回车键并观察如图 21-9 所示的结果。

图　21-9

工作原理

本例中使用的正则表达式模式很关键，它在前面例子中模式的基础上经过了细微的修改：

```
Dim myRegex = New Regex("([A-Z])+(\d)+")
```

注意，模式中将字符类 [A-Z] 包含在一对圆括号中，并且使用了 + 限定符限定了该组。这意味着存在捕获单个字符的组。如果测试字符串中有一个大写的字母字符，就会创建一个组和一个捕获。而如果有多个大写的字母字符，则会创建多个组和多个捕获。

在模式的数字部分也体现出与前面同样的考虑，因为根据测试字符串的内容不同，(\d) 可能会创建一个出现一次或多次的组。而模式 (\d)+ 会为每一个捕获的数字创建一个组，它与模式(\d+)不一样，后者会创建一个组，但组中可能包含一个，也可能包含更多个数字。

在接受用户输入的测试字符串 ABC1 A123 后，匹配、组和捕获分别在三层嵌套的 For

Each 循环中被处理：

```
For Each myMatch In myMatchCollection
```

首先，显示每一个匹配项的值：

```
Console.WriteLine("At position {0}, the match '{1}' was found",
myMatch.Index, myMatch.ToString)
```

然后，显示该匹配项中的组数：

```
Console.WriteLine("This match has {0} groups.", myMatch.Groups.Count)
```

并将 myMatch 变量的 Groups 属性指定给变量 myGroupCollection：

```
myGroupCollection = myMatch.Groups
```

接下来，处理 GroupCollection 集合中的每一个 Group 对象：

```
For Each myGroup In myGroupCollection
```

将每个组的捕获指定给变量 myCaptureCollection：

```
Dim myCaptureCollection As CaptureCollection = myGroup.Captures
Dim myCapture As Capture
```

并显示每个组。当你看到有多个实例的组中只显示最后一个实例时可能会感到惊讶(实际上是正常的，因为模式中的组件([A-Z])+ 和 (\d)+ 会为匹配的每个字母或数字分别创建一个组和一个捕获。而在创建多个捕获的过程中，组的引用结果始终只有一个，因此在显示组的值时，只有最后一次匹配的字母或数字被保留。译者注)：

```
    Console.WriteLine("Group containing '{0}' found at position
'{1}'.", myGroup.Value, myGroup.Index)
```

然后，显示捕获集合中的每一个捕获的值和位置。因为每对圆括号只捕获一个单独的字符，所以像 ABC 这样的字符序列将会导致显示三次捕获：

```
            For Each myCapture In myCaptureCollection
                Console.WriteLine(" Capture: '{0}' at position '{1}'.",
myCapture.Value, myCapture.Index)
```

每个 For Each 循环中最后都包含一个 Next 语句：

```
        Next
    Next
    Console.WriteLine()
Next
```

21.1.6 RegexOptions 枚举

在 System.Text.RegularExpressions 命名空间中包含一个 RegexOptions 枚举，用于控制正则表达式匹配操作的模式。

下面的表 21-4 中总结了 RegexOptions 枚举的特性。

表 21-4　Regex Options 的特性

选　　项	说　　明
None	指定没有设置选项
IgnoreCase	指定匹配不区分大小写
Multiline	将每一行视为独立的字符串进行匹配。因此，^ 元字符的意义就不一样了(现在匹配每一行的开始位置)，$ 元字符也不一样了(现在匹配每一行的结束位置)(^ 元字符和 $ 元字符本来分别匹配一个字符串的开始和结束位置。译者注)
ExplicitCapture	改变圆括号的捕获行为
Compiled	指定是否将正则表达式编译为独立的程序集
SingleLine	改变句点元字符的含义，使其匹配全部字符。正常情况下，句点元字符匹配除 \n 外的全部字符
IgnorePatternWhitespace	不将未转义的空白符作为模式的一部分。允许前置 # 的嵌入式(inline)注释
RightToLeft	指定模式的搜索从右向左进行
ECMAScript	启用(受限的)ECMAScript 兼容性
CultureInvariant	指定忽略语言中的文化差异

1. 不区分大小写的匹配：使用 IgnoreCase 选项

在 Visual Basic .NET 的正则表达式中，默认的匹配模式是区分大小写的。要指定匹配以不区分大小写的模式进行，则需要使用 IgnoreCase 选项。

试一试：进行不区分大小写的匹配

本例中使用的测试代码包含在 IgnoreCaseDemo 项目中的 Module1.vb 文件中：

```
Imports System.Text.RegularExpressions
Module Module1
    Sub Main()
        Dim myPattern As String = "[A-Z]+\d+"
        Console.WriteLine("Enter a string on the following line:")
        Dim inputString = Console.ReadLine()
        Dim myMatchCollection = Regex.Matches(inputString, myPattern)
        Console.WriteLine("This is case sensitive matching.")
        Console.WriteLine("There are {0} matches.", myMatchCollection.Count)
        Console.WriteLine()
        Dim myMatch As Match
        For Each myMatch In myMatchCollection
            Console.WriteLine("At position {0}, the match '{1}' was found",
```

```
    myMatch.Index, myMatch.ToString)
            Next
            Console.WriteLine()
            myMatchCollection = Regex.Matches(inputString, myPattern,
            RegexOptions.IgnoreCase)
    Console.WriteLine("This is case insensitive matching.")
            Console.WriteLine("There are {0} matches.", myMatchCollection.Count)
            Console.WriteLine()
            For Each myMatch In myMatchCollection
                Console.WriteLine("At position {0}, the match '{1}' was found",
    myMatch.Index, myMatch.ToString)
            Next
            Console.WriteLine()
            Console.WriteLine("Press Return to close this application.")
            Console.ReadLine()

        End Sub

    End Module
```

(1) 在 Visual Studio 2003 中使用控制台程序模板创建一个新项目，并命名为 CaseInsensitiveDemo。

(2) 输入前面所示的代码。

(3) 保存代码，然后按 F5 键运行这些代码。

(4) 在命令窗口中，输入测试文本 ABC123 abc123 DeF234。

(5) 按回车键，并观察如图 21-10 所示的结果。当以区分大小写的模式进行匹配时，有两个匹配项；而当以不区分大小写的模式进行匹配时，则有三个匹配项。同时，在区分大小写时匹配的是 DeF234 中的 F234，而在不区分大小写时匹配的则是 DeF234。

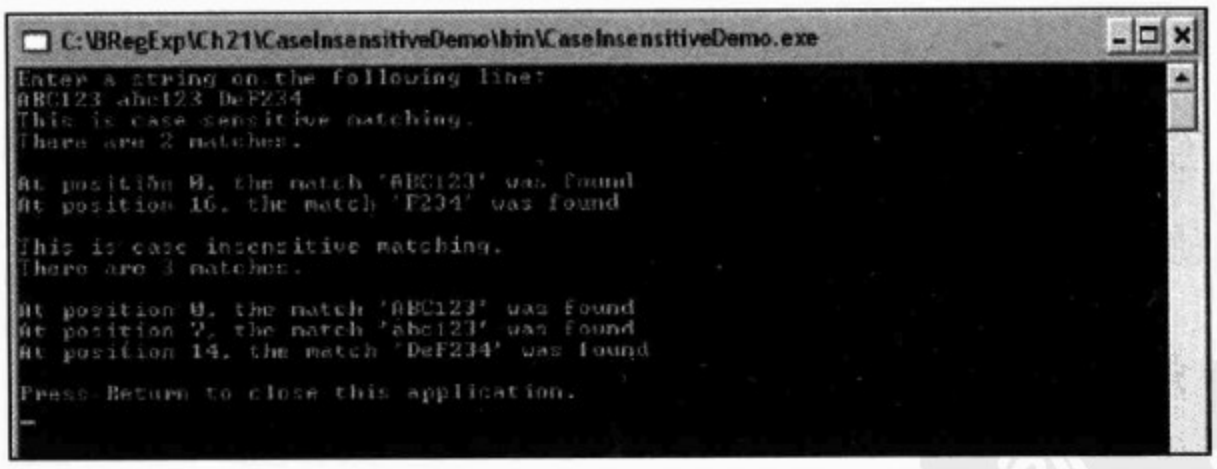


图 21-10

工作原理

指定给变量 myPattern 的模式字符串(注意，此处没有实例化 Regex 对象)是 [A-Z]+\d+。在使用 .NET 中默认的区分大小写的模式时，这个模式只会匹配用户输入的大写字母字符：

```
    Dim myPattern As String = "[A-Z]+\d+"
```

与 myMatchCollection 变量对应的 MatchCollection 对象是由 Regex 类的共享方法 Matches()创建的。此时没有实例化 Regex 对象。注意，Matches()方法在这种情况下(应用不区分大小写的匹配模式时)带有两个参数，第二个参数是包含正则表达式模式的字符串：

```
Dim myMatchCollection = Regex.Matches(inputString, myPattern)
```

字符序列 ABC123 匹配是因为该字符序列中只包含大写的字母字符。而按照相同的标准，字符序列 abc123 不匹配。在字符序列 DeF234 中，只有字符序列 F234 匹配，是因为其中包含一个大写字母字符后跟三个数字。

在显示了区分大小写的匹配结果后，会将另一个新的 Match 对象的集合指定给 myMatchCollection 变量。在这种情况下，Matches() 方法接受了三个参数。第三个参数指定了 RegexOptions 对象的一个属性——IgnoreCase：

```
myMatchCollection = Regex.Matches(inputString, myPattern,
RegexOptions.IgnoreCase)
```

当以不区分大小写的模式进行匹配时，找到了三个匹配的字符序列—— ABC123、abc123 和 DeF234 都匹配模式 [A-Z]+\d+。

如果想指定多个选项，则必须以单词 Or 来分隔每个选项。因此，如果要指定不区分大小写的、从右往左的匹配，可以使用下面的代码：

```
myMatchCollection = Regex.Matches(inputString, myPattern, _
RegexOptions.IgnoreCase Or RegexOptions.RightToLeft)
```

2. 多行匹配：对 ^ 和 $ 元字符的影响

^ 元字符在正常情况下匹配一个字符串开始处第一个字符之前的位置，而 $ 元字符在正常情况下匹配字符序列结尾处最后一个字符之后的位置。当使用多行匹配模式时，^ 元字符则改为匹配每行开始处第一个字符之前的位置，而 $ 元字符变成了匹配每行结尾处最后一个字符之后的位置。

21.1.7　使用 IgnorePatternWhitespace 选项添加嵌入式说明

IgnorePatternWhitespace 选项允许使用嵌入式注释，可以对正则表达式模式的每一部分都进行清晰的说明。

通常，在使用一个正则表达式模式匹配时，其中的任何空白符都是有意义的。例如，模式中的一个空格符会被解释为要匹配一个字符。当设置了 IgnorePatternWhitespace 选项后，模式中包含的所有空白符都将被忽略，包括空格符和换行符。这样，就可以允许对单个模式进行分行排布以增进可读性，并借以添加注释，对维护正则表达式模式提供支持。而此时若要匹配一个空白符，可以使用 \s 元字符。

在 Visual Basic .NET 中，添加嵌入式注释的语法有些繁琐。如果想使用 myRegex 变量匹配一个字母字符后跟一个数字，那么可以将代码写成下面这样：

```
Dim myRegex = New Regex("[A-Z]\d")
```

然而，在使用 IgnorePatternWhitespace 选项指定相同的正则表达式模式并包括嵌入式

注释的情况下，则必须写成下面这样：

```
Dim myRegex = New Regex( _
   "[A-Z] (?# A character class to match an uppercase alphabetic character)" & _
   "\d (?# followed by a numeric digit)", & _
   RegexOptions.IgnorePatternWhitespace)
```

其中的嵌入式注释前置了字符序列 (?# 并后跟一个 2) 字符。Visual Basic .NET 中的连接字符——&，用于连接模式的各个组件。而行延续字符(下划线)则用于表示一条语句将在下一行中继续。

试一试：使用 IgnorePatternWhitespace 选项

本例将匹配美国社会保障号码(SSN)。要使用的测试代码位于 IgnorePatternWhitespace Demo 项目的 Module1.vb 文件中：

```
Imports System.Text.RegularExpressions
Module Module1
    Dim myRegex = _
    New Regex _
       ("^ (?# match the position before the first character)" & _
       "\d{3} (?# Three numeric digits, followed by)" & _
       "-  (?# a literal hyphen)" & _
       "\d{2} (?# then two numeric digits)" & _
       "-  (?# then a literal hyphen)" & _
       "\d{4} (?# then two numeric digits)" & _
       "$  (?# match the position after the last character)", _
       RegexOptions.IgnorePatternWhitespace)
    Sub Main()
        Console.WriteLine("Enter a string on the following line:")
        Dim inputString = Console.ReadLine()
        Dim myMatch = myRegex.Match(inputString)
        If myMatch.ToString.Length Then
            Console.WriteLine("The match, '" & myMatch.Value & "' was found.")
        Else
           Console.WriteLine("There was no match")
        End If
        Console.WriteLine("Press Return to close this application.")
        Console.ReadLine()

    End Sub

End Module
```

(1) 在 Visual Studio 2003 中创建一个新的控制台程序项目，并命名为 IgnorePatternWhitespaceDemo。

(2) 在代码窗口中输入前面 Module1.vb 文件中的代码。保存代码，并按 F5 运行它们。

(3) 在命令行中，输入测试字符串 123-12-1234。按回车键，并观察如图 21-11 所示的结果。结果显示有一个成功的匹配。

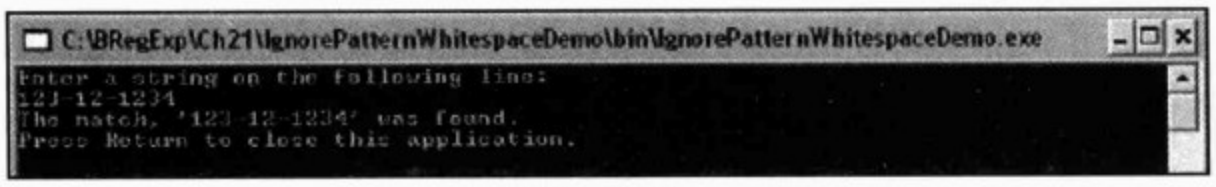


图　21-11

(4) 在 Visual Studio 2003 中，按 F5 再次运行这些代码。

(5) 在命令行中输入测试字符串 A123-12-1234。按回车键，并观察如图 21-12 所示的结果。

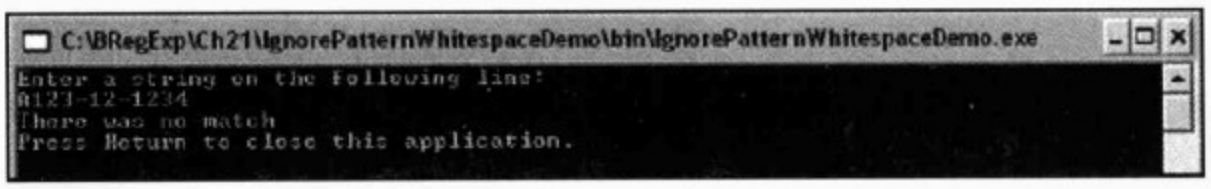


图　21-12

工作原理

如果将要匹配的模式写在一行中，如下所示：

```
^\d{3}-\d{2}-\d{4}$
```

可能一眼就能看出这个简单的模式会匹配美国社会保险号(SSN)。

在这个例子中，该模式是在定义变量 myRegex 时与嵌入式注释写在一起的。以这种方式写模式会更复杂一些，但其中嵌入的注释却能够使其他开发人员清楚地知道这个模式的用途。

但是，Visual Basic .NET 中为嵌入式注释准备的语法——(?# ...)与 Visual C# .NET 中简单的 # 结构比较起来的确是不够简洁。Visual Basic .NET 中的这种语法反而会影响注释的可读性。因此，可以将每行中左边的圆括号对齐以最大限度地提高 Visual Basic .NET 中嵌入式注释的可读性：

```
New Regex _
   ("^   (?# match the position before the first character)" & _
   "\d{3} (?# Three numeric digits, followed by)" & _
   "-   (?# a literal hyphen)" & _
   "\d{2} (?# then two numeric digits)" & _
   "-   (?# then a literal hyphen)" & _
   "\d{4} (?# then two numeric digits)" & _
   "$   (?# match the position after the last character)", _
   RegexOptions.IgnorePatternWhitespace)
```

以上模式相当于 ^\d{3}-\d{2}-\d{4}$ ，因而会匹配一个 SSN。所以，当测试字符串是 123-12-1234 时，匹配成功，如图 21-11 所示。这个结果的显示是由下面代码中的 If 语句控制的。当 Length 属性不是 0 时，说明找到了一个匹配项，所以 myMatch 变量的 Value 属性中会包含匹配的字符序列：

```
If myMatch.ToString.Length Then
Console.WriteLine("The match, '" & myMatch.Value & "' was found.")
```

而当 myMatch.ToString 的 Length 属性是 0 时，则说明没有找到匹配项，此时会通过 Else 子句输入一条表示没有匹配项的消息：

```
Else
   Console.WriteLine("There was no match")
End If
```

从右向左匹配：使用 RightToLeft 选项

在英语中，一行中的字符正常情况下是从左往右排列的。而在其他一些语言中，字符的排列顺序则是从右往左的。为了在这种语言中支持使用正则表达式，.NET 架构中提供了相应的功能以实现从右往左的匹配。但遗憾的是，当使用 RightToLeft 选项时，匹配的结果并不完全可靠。

21.2 Visual Basic .NET 支持的元字符

Visual Basic .NET 大概是本书到目前为止介绍的所有工具中拥有最完整、最全面的正则表达式实现的一门语言。Visual Basic .NET 中支持的许多正则表达式功能都可以合情合理地称之为标准。但是，就如同许多其他 Microsoft 的技术一样，这些标准的语法和技术中也存在某些需要扩展或改进的地方。

下面的表 21-5 中总结了 Visual Basic .NET 所支持的元字符。

表 21-5 Visual Basic .NET 中支持的元字符

元 字 符	说 明
\d	匹配一个数字
\D	匹配任何非数字
\w	相当于字符类 [A-Za-z0-9_]
\W	相当于字符类 [^A-Za-z0-9_]
\b	匹配一个 \w 字符序列的开始位置或一个 \w 字符序列的结束位置。通俗地说，\b 就是匹配词边界的元字符
\B	匹配一个非 \b 位置的位置
\t	匹配一个制表符
\n	匹配一个换行符
\040	匹配一个以八进制记数法表示的 ASCII 字符。\040 元字符匹配一个空格符
\x020	匹配一个以十六进制记数法表示的 ASCII 字符。\x020 元字符匹配一个空格符
\u0020	匹配一个以带有四位数字的十六进制记数法表示的 Unicode 字符。\u0020 元字符匹配一个空格符

(续表)

元 字 符	说　明
[...]	匹配字符类中的任何一个字符
[^...]	匹配字符类中未指定的任何一个字符
\s	匹配一个空白符
\S	匹配任何一个非空白符
^	如果设置了 MultiLine 选项，它匹配一行中第一个字符之前的位置；否则，匹配一个字符串中第一个字符之前的位置
$	如果设置了 MultiLine 选项，它匹配一行中最后一个字符之后的位置；否则，匹配一个字符串中最后一个字符之后的位置
$number	替代与组号为 number 的组匹配的最后一个子字符序列
${name}	替代与组名为 name 的组匹配的最后一个子字符序列
\A	匹配一个字符串中第一个字符之前的位置。它的行为不受设置 MultiLine 选项的影响
\Z	匹配一个字符串中最后一个字符之后的位置。它的行为不受设置 MultiLine 选项的影响
\G	指定匹配必须是连贯的，即不能包含任何居间的非匹配字符
?	限定符。匹配前面字符或组的零个或一个实例
*	限定符。匹配前面字符或组的零个或多个实例
+	限定符。匹配前面字符或组的一个或多个实例
{n}	限定符。匹配前面字符或组的 n 个实例
{n,m}	限定符。匹配前面字符或组最少 n 个、最多 m 个实例
(substring)	捕获包含的子字符串
(?<name>substring)	捕获包含的子字符串，并为其指定一个名称
(?:substring)	一个非捕获组
(?=...)	肯定式向前查找
(?!...)	否定式向前查找
(?<=...)	肯定式向后查找
(?<!...)	否定式向后查找
\N (其中 N 为数字)	对编号组的反向引用
\k<name>	引用一个命名的反向引用的反向引用(与下同)
\k'name'	引用一个命名的反向引用的反向引用(与上同)
|	交替选择
(?imnsx-imnsx)	一种内置地指定 RegexOptions 设置的替换技术

向前查找和向后查找

Visual Basic .NET 对肯定式、否定式的向前查找和向后查找的支持非常好。四种查找方式都支持。

肯定式向前查找使用如下的语法：

```
(?=向前查找模式)
```

要匹配后面跟一个空格符和字符序列 Training 的单词 Star，可以使用下面的代码：

```
Dim myRegex = New Regex("Star(?= Training)")
Dim myMatch = myRegex.Match("The Star Training Company carries out great
training.")
```

否定式向前查找使用如下的语法：

(?!向前查找模式)

要匹配后面不跟一个空格符和字符序列 Training 的字符序列 Star，可以使用下面的代码：

```
Dim myRegex = New Regex("Star(?! Training)")
Dim myMatch = myRegex.Match("The Star Training Company carries out great
training.")
```

肯定式向后查找使用如下的语法：

(?<=向后查找模式)

要匹配前面是一个字符序列 Star 后跟一个空格符的字符序列 Training，可以使用下面的代码：

```
Dim myRegex = New Regex("(?<=Star )Training)")
Dim myMatch = myRegex.Match("The Star Training Company carries out great
training.")
```

否定式向后查找使用下面这样的语法：

(?<!向后查找模式)

要匹配前面不是字符序列 Star 后跟一个空格符的字符序列 Training，可以使用下面的代码：

```
Dim myRegex = New Regex("(?<!Star )Training)")
Dim myMatch = myRegex.Match("The Star Training Company carries out great
training.")
```

21.3　练习

1. 请编写一个模式，使其只匹配单词 cold 或 bold 中的字符序列 old。提示：有两种方案，其中一种是使用向后查找和向前查找。

2. 请创建一个控制台程序，用字符序列 Dr. 替换字符序列 Doctor 或 Doc。

第22章

C#和正则表达式

Microsoft Visual C# .NET 对正则表达式提供了功能全面、强大、灵活的支持。Visual C# .NET 不仅提供了与 Perl 5.0 相同的支持，还增加了本质上是 .NET 架构专有的一些扩展(例如，从右往左匹配)。正则表达式的实现基本上是一种与时俱进的过程，很可能其他一些语言也将实现诸如从右往左匹配这样的特性。

在本章中将学习如下内容：

- 使用 System.Text.RegularExpresssions 命名空间中的对象
- 使用 C#支持的元字符

本章的例子都在 Visual Studio 2003 和 .NET Framework 1.1 中测试通过。本章假设你拥有访问一个 Visual Studio 2003 副本的权限，而且至少具有使用 Visual C# .NET 的基本知识。本章不会详细介绍如何使用 Visual Studio .NET 2003。

不过，如果没有访问一个 Visual Studio 2003 副本的权限，可以运行本书测试文件中提供的 .exe 文件。但是无法查看和编辑这些 .exe 文件中的 Visual C# .NET 代码。

C# 中的正则表达式功能是以 System.Text.RegularExpressions 命名空间中的类为基础的。本章将详细介绍这些类，并提供使用这些类及其属性和方法的例子。

22.1 System.Text.RegularExpressions 命名空间中的类

.NET 架构类库中对正则表达式的支持包含在 System.Text.RegularExpressions 命名空间中。

22.1.1 介绍性的例子

这个例子示范了在 C#中使用正则表达式的一种方式。其他方式将在本章后面介绍，届时也将对 System.Text.RegularExpressions 命名空间中的类及其成员做更详细的讨论。

试一试：一个介绍性的 C# 控制台应用的例子

下面的代码包含在 SimpleMatch 项目的文件 Class1.cs 中：

```
using System;
using System.Text.RegularExpressions;

namespace SimpleMatch
{
 /// <summary>
 /// This is a simple regular expression example which uses the Regex object.
 /// </summary>
 class Class1
 {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
               Console.WriteLine(@"This will find a match for the regular
               expression '[A-Z]\d'.");
               Console.WriteLine("Enter a test string now.");
               Regex myRegex = new Regex(@"[A-Z]\d", RegexOptions.IgnoreCase);
               string inputString;
               inputString = Console.ReadLine();
               Match myMatch = myRegex.Match(inputString);
               Console.WriteLine("You entered the string: '" + inputString +
               "'.");
               if (myMatch.Success)
               Console.WriteLine("The match '" + myMatch.ToString() + "' was found
               in the string you entered.");
               Console.ReadLine();
        }
  }
}
```

以下的说明示范了在 Visual Studio 2003 中使用 Visual C# .NET 创建一个简单的控制台程序的步骤。如果曾经使用 C#编过很多程序，那么这些步骤对你来说一点也不陌生。

(1) 打开 Visual Studio 2003，在 File 菜单中选择 New，然后选择 Project 创建包含一个项目的新方案。

图 22-1 显示了新建项目的界面，但已经选择第 2 步到第 5 步中指定的选项。

(2) 在 Project Types 面板中，选择 Visual C# Projects。

(3) 在 Templates 面板中，选择 Console Application。

(4) 在 Name 文本框中，输入 SimpleMatch 作为项目名称。

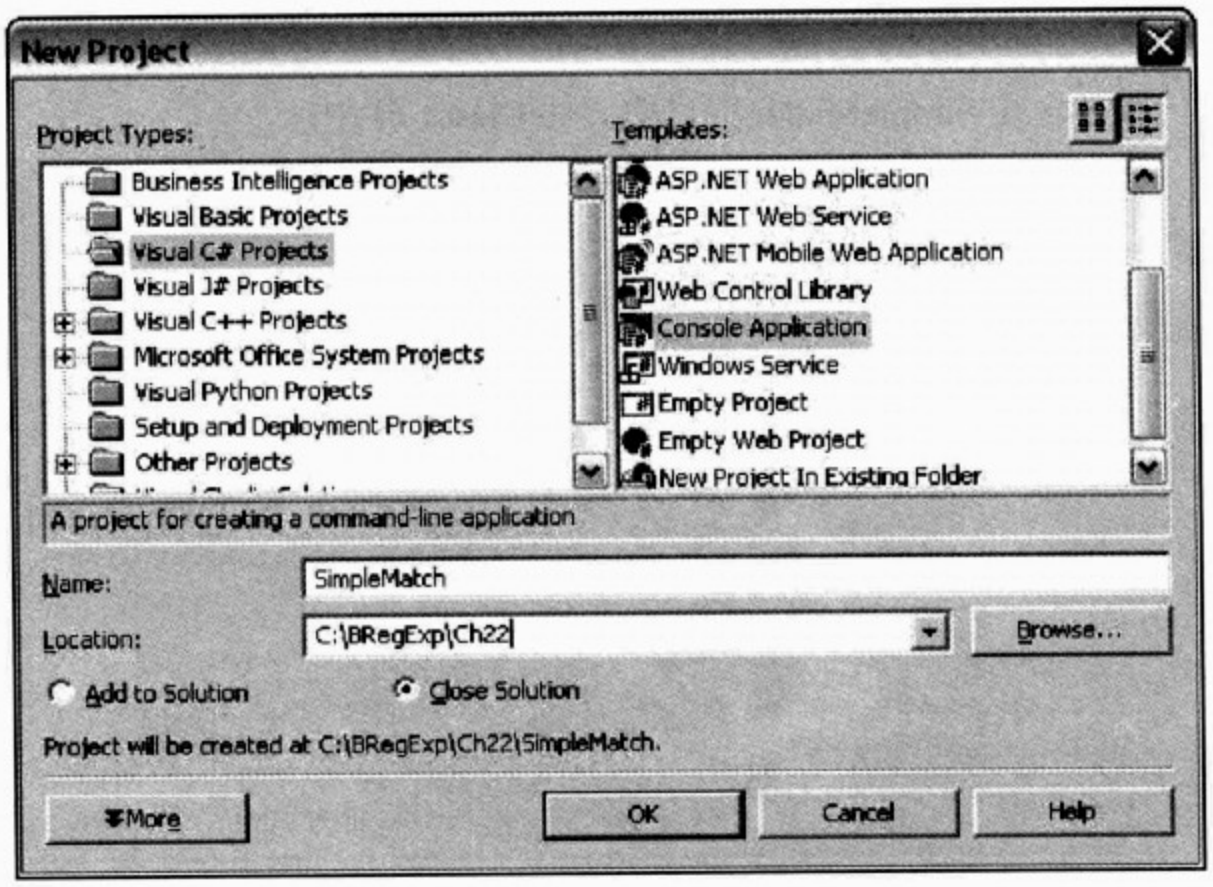


图 22-1

(5) 在 Location 文本框中，输入 C:\BRegExp\Ch22 作为保存项目的位置(也可以选择其他位置)。

(6) 单击 OK 按钮。在短暂的停顿期间 Visual Studio 2003 会准备好新项目所需的文件，然后代码编辑器打开，出现下列模板代码：

```
using System;

namespace SimpleMatch
{
 /// <summary>
 /// Summary description for Class1.
 /// </summary>
 class Class1
 {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
              //
              // TODO: Add code to start application here
              //
        }
 }
}
```

(7) 编辑前面的模板代码，使它包含如 Class1.cs 文件中的代码。

(8) 使用 Ctrl+Shift+S 快捷键保存代码。按 F5 运行这些代码。

(9) 在命令行中，输入测试文本 K9，然后按回车键。观察如图 22-2 所示的结果。

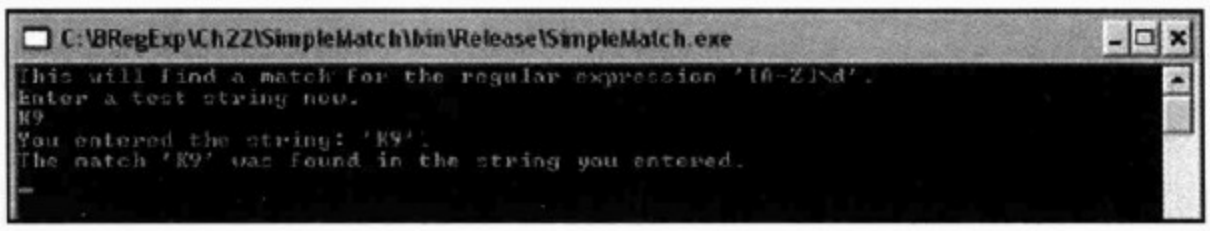


图　22-2

工作原理

使用 C#时，必须指明使用的.NET 架构类库中的组件。Visual Studio 2003 会在创建文件 Class1.cs 时自动添加下面一行代码：

```
using System;
```

因为要使用包含在 System.Text.RegularExpresssions 命名空间中的 Regex 类，所以还需要使用 using 语句添加对该命名空间的引用：

```
using System.Text.RegularExpressions;
```

当然，也可以在引用对象时使用完整的限定名。不过，代码中有 using System.Text.RegularExpressions 语句，就可以使用下面更简洁的代码来声明 myRegex 对象并给它赋值：

```
Regex myRegex = new Regex(@"[A-Z]\d", RegexOptions.IgnoreCase);
```

如果代码中没有 using System.Text.RegularExpressions 语句却尝试运行以上代码，则会得到一堆错误信息——包括下面的这些信息，因为在 Visual Studio 2003 默认模板代码中声明的 System 命名空间中找不到 Regex 类：

```
The type or namespace name 'Regex' could not be found.
```

由于 Regex 和 RegexOptions 类包含在 System.Text.RegularExpressions 命名空间中，所以，要声明 myRegex 变量并给它赋值，就要使用下面这些包含完整限定名的代码：

```
System.Text.RegularExpressions.Regex myRegex = new
System.Text.RegularExpressions.Regex(@"[A-Z]\d",
System.Text.RegularExpressions.RegexOptions.IgnoreCase);
```

同样，要声明 myMatch 对象变量并给它赋值，则需要使用下面的代码：

```
System.Text.RegularExpressions.Match myMatch = myRegex.Match(inputString);
```

可以看出，其中大部分都是重复的代码。如果使用 using System.Text. RegularExpressions 语句，可以使代码更容易编写同时也具有更高的可读性。

在 Cass1.cs 中会自动添加注释生成占位程序(stubs)，同时也会自动生成与项目名称和类名对应的命名空间——默认是 Class1。

Main()方法的内容就是这个简单例子的核心所在：

```
static void Main(string[] args)
{
```

首先，使用 Console 对象的 WriteLine() 方法把一条消息写入到命令窗口。Console 类是 System 命名空间的一个成员，由于已经使用 using System 语句引用了该命名空间，所以可以简单地写成 Console.Writeline()，并把相应的内容放在圆括号中：

```
Console.WriteLine(@"This will find a match for the regular expression
'[A-Z]\d'.");
```

注意：WriteLine()方法的圆括号中的第一个字符是 @ 字符。如果没有这个字符，程序就会报错——因为，C# 不能识别字符序列 \d。如果不加上 @ 字符，那么就必须将双引号中的字符串写成 “This will find a match for the regular expression ‘[A-Z]\\d’.”。换句话说，必须写成 \\d 才能让 C# 认出它是一个正则表达式元字符 \d。

个人认为，使用@字符更好，因为这样就可以使用其他语言中熟悉的正则表达式语法了。由于经常要在多种语言和工具中使用正则表达式，所以作者倾向于避免使用双反斜杠：

接着，输出一个简单的信息字符串：

```
Console.WriteLine("Enter a test string now.");
```

然后，声明一个继承自 Regex 类的对象变量 myRegex。如前所述，Regex 是对完整限定名 System.Text.RegularExpressions.Regex 的缩写。正则表达式模式 [A-Z]\d 是 Regex() 构造函数的第一个参数，它指定了据以匹配的模式。Regex() 构造函数的第二个参数指定匹配要使用的是不区分大小写的选项：

```
Regex myRegex = new Regex(@"[A-Z]\d", RegexOptions.IgnoreCase);
```

接着，再声明一个字符串变量 inputString：

```
string inputString;
```

Console 类的 ReadLine()方法用于读取用户输入的文本。将读取的值指定给 inputString 变量：

```
inputString = Console.ReadLine();
```

声明一个继承自 Match 类的对象变量 myMatch。myMatch 变量的值由 Regex 类的 Match()方法指定，Match()方法的第一个参数就是 inputString 变量。也就是说，myMatch 变量中包含着以前面指定给 myRegex 变量的正则表达式模式 [A-Z]\d 在 inputString 变量中找到的第一个匹配项：

```
Match myMatch = myRegex.Match(inputString);
```

在取得了匹配项之后，首先输出 inputString 变量的值以提示或告知用户由 Console.ReadLine()方法捕获的字符串：

```
Console.WriteLine("You entered the string: '" + inputString + "'.");
```

然后，使用 if 语句来测试 myMatch 对象变量的 Success 属性。如果确实存在匹配项(根据 Success 属性的值判定)，则使用 Console.WriteLine() 方法输出一个字符串以告知用户匹配的内容：

```
    if (myMatch.Success)
    Console.WriteLine("The match '" + myMatch.ToString() + "' was found in the
string
    you entered.");
```

使用 ReadLine()方法可以让显示的匹配信息在用户按回车键之前始终保持显示在屏幕中：

```
    Console.ReadLine();
    }
```

22.1.2　System.Text.RegularExpressions 的类

表 22-1 中列出了 System.Text.RegularExpressions 命名空间中包含的类。本章后面将详细介绍其中一些类的属性和方法，同时给出例子示范如何使用它们。

表 22-1　System.Text.RegularExpressions 命名空间中包含的类

类	说　明
Capture	表示由一对包围子表达式的圆括号捕获的文本
CaptureCollection	表示 Capture 对象的一个集合
Group	表示一对单独的圆括号捕获组的结果
GroupCollection	表示 Group 对象的一个集合
Match	表示一个单独的正则表达式匹配的结果
MatchCollection	表示 Match 对象的一个集合
Regex	这个类中包含正则表达式模式
RegexCompilationInfo	提供编译器用来将正则表达式编译为一个程序集的信息

Regex 类是 System.Text.RegularExpressions 命名空间中最重要的类。

22.1.3　Regex 类

可以通过实例化一个 Regex 对象，使其属性和方法能够通过编程来操纵。此外，Regex 类还有三个静态方法。有关这三个共享方法的使用将在后面介绍和示范。

Regex 对象有两个公共属性，表 22-2 简单地说明了这两个属性。

表 22-2　Regex 对象的两个公共属性

属　性	说　明
Options	包含传递给 Regex 对象的选项信息
RightToLeft	返回一个布尔值，表示采取的匹配方式是否为从右向左搜索。若值为 True 则表示从右向左搜索

如果习惯于在 JScript 或 VBScript 中创建正则表达式对象时使用 RegExp 对象，那么在.NET 中要注意正确拼写 Regex 对象。

1. Regex 类的 Options 属性

Options 属性中所包含的值都是与某个选项对应的一个值。每个值要么是 0 要么是 1，这两种状态表示是否选中一个特定的选项。例如，将等价于 RegexOptions.None 的默认值传递给 Match()方法，那么该选项的值就是 0。可用的选项将在下面介绍 RegexOptions 类时说明。

2. Regex 类的 RightToLeft 属性

.NET 架构中允许使用从右往左的匹配。当匹配英语或其他从左往右书写的语言时，正则表达式的匹配过程是也可以从左往右匹配。而当在匹配阿拉伯或希伯来文时，由于这些语言都是从右往左书写的，那么 Regex 类的 RightToLeft 属性就可以支持(或者被设计得支持)从右往左匹配。

但是，并非所有的匹配过程都是相反的。比如，向前查找仍然会在当前匹配位置的右侧查找字符序列，而向后查找也仍会在匹配位置的左侧再查找字符序列(以确定当前匹配项。译者注)。在实践中，已经证实了使用 RightToLeft 属性并非如我们想象的那样可靠，所以这里不再做示范。

3. Regex 类的方法

表 22-3 总结了 Regex 对象可以使用的方法。其中一些 Regex 对象的概念和技术不在标准正则表达式的范畴内。因此，表 22-3 中总结的一些概念将有助于理解后续几节中对这些方法的介绍及示范。

表 22-3 Regex 类的方法

方　法	说　明
CompileToAssembly	将正则表达式编译为一个程序集(默认的行为是不把正则表达式编译为一个程序集)
Equals	判断两个对象是否相等
Escape	转义一组元字符，即以转义后的字符替换相应的元字符
GetGroupNames	返回一个包含捕获组名的数组
GetGroupNumbers	返回一个包含捕获组号的数组
GetHashCode	继承 Object 类
GetType	取得当前实例的类型
GroupNameFromNumber	取得与作为参数组号对应的组名
GroupNumberFromName	取得与作为参数组名对应的组号

(续表)

方　法	说　明
IsMatch	返回一个布尔值，表示正则表达式模式是否在一个字符串中找到匹配项，该字符串是 IsMatch() 方法的参数
Match	根据作为参数传递给这个方法的字符串中是否包含一个匹配项，返回零个或一个 Match 对象
Matches	返回一个包含零个或多个 Match 对象的 MatchCollection 对象，它包含作为 Matches() 方法参数的字符串中的所有匹配项(或一个也没有)
Replace	以指定的字符序列替换符合一个正则表达式模式的所有匹配项
Split	将一个输入字符串拆分为一个子字符串数组。拆分的位置由一个正则表达式模式指定
ToString	返回通过构造函数传递给 Regex 对象的正则表达式模式字符串
Unescape	取消输入字符串中转义字符的转义

4. CompileToAssembly()方法

Regex 类的 CompileToAssembly()方法接受两个参数：RegexCompilationInfo 对象(它是 System.Text.RegularExpressions 命名空间的成员，包含用于指定如何完成编译的必要信息)和要创建的程序集的名称。

使用 CompileToAssembly()方法后，启动时间会变长，但运行速度会加快。

5. GetGroupNames()方法

GetGroupNames()可以取得与一个 Match 对象相关的任何命名组的名称。GetGroupNames()方法没有参数。

6. GetGroupNumbers()方法

GetGroupNumbers()可以取得与一个 Match 对象相关的任何编号组的编号。至少会有一个组，即匹配整个正则表达式模式。如果正则表达式模式中还包含成对的圆括号，那么还将有额外的编号组。GetGroupNumbers()方法无参数。

7. GroupNumberFromName()方法和 GroupNameFromNumber()方法

GroupNumberFromName()方法用于取得与作为参数的组名对应的组号。作为参数的组名以字符串形式提供。GroupNameFromNumber()方法用于取得与作为参数的组号对应的组名。作为参数的组号以整数形式提供。

8. IsMatch()方法

Regex 对象的 IsMatch()方法接受一个字符串参数，并测试该字符串是否包含与正则

表达式模式匹配的内容。它返回布尔值。IsMatch() 方法还可以接受可选的第二个参数——一个整数值，该参数用于指定字符串参数中开始匹配的位置。

试一试：使用 IsMatch() 方法

(1) 打开 Visual Studio 2003，选择控制台程序模板并创建一个项目，将新项目命名为 IsMatchDemo。

(2) 在 using System 语句后面添加下列语句：

```
using System.Text.RegularExpressions;
```

(3) 将下列代码添加为 Main()方法的内容：

```
Console.WriteLine(@"This will find a match for the regular expression
'[A-Z]\d'.");
          Console.WriteLine("Enter a test string now.");
          Regex myRegex = new Regex(@"[A-Z]\d", RegexOptions.IgnoreCase);
          string inputString;
          inputString = Console.ReadLine();
          Match myMatch = myRegex.Match(inputString);
         string outputString = "The following option(s) are set: ";
          Console.WriteLine(outputString + myRegex.Options.ToString());
          Console.WriteLine("You entered the string: '" + inputString + "'.");
          if (myRegex.IsMatch(inputString))
                    Console.WriteLine("The match '" + myMatch.ToString() + "' was
found in the string you entered.");
          else
               Console.WriteLine("No match was found.");
          Console.ReadLine();
```

(4) 保存代码，并按 F5 运行代码。

(5) 在命令行提示符中输入字符串 J88，按回车键，并观察如图 22-3 所示的信息。

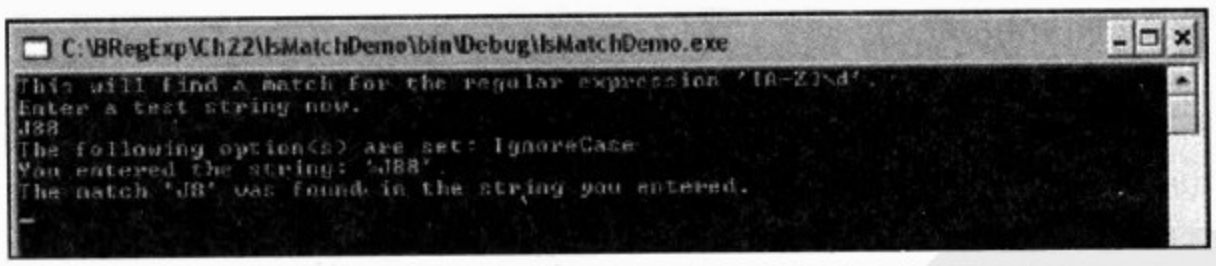


图 22-3

IsMatch()方法也可以被静态重载，使得使用该方法可以不必实例化 Regex 对象。

工作原理

在本章的第一个例子中，使用正则表达式模式 [A-Z]\d 实例化了一个 Regex 对象并将其指定给对象变量 myRegex。由于模式中包含一个带反斜杠的元字符，所以在 Regex()构造函数的字符串参数前放置了一个 @ 符号，因此不需要使用两个反斜杠：

```
Regex myRegex = new Regex(@"[A-Z]\d", RegexOptions.IgnoreCase);
```

当用户输入字符串后，通过 IsMatch() 方法来判定输入的字符串中是否包含匹配项：

```
if (myRegex.IsMatch(inputString))
```

如果测试字符串中包含一个匹配项，则返回布尔值 True，在这个例子中，就会执行 if 语句中的代码；如果没有发现匹配项，则返回布尔值 False，此时将执行 else 语句中的代码：

```
else

Console.WriteLine("No match was found.");
```

由于正则表达式模式是[A-Z]\d，所以测试字符串 J88 中的匹配项是 J8。

9. Match()方法

Match() 方法具有下列重载的方法：

```
public Match Match(string, inputString);
```

和

```
public Match Match(string inputString,
                   int startAt);
```

和

```
public Match Match(string inputString,
                   int startAt,
                   int length);
```

通过对参数 inputString 进行测试，可以知道其中是否存在与包含在 Regex 对象中的正则表达式模式匹配的内容。

在本章第一个例子中已经看到，Match()方法返回一个 Match 对象：

```
Match myMatch = myRegex.Match(inputString);
```

Match()方法用来查找一个测试字符串中是否包含匹配项。通过将 Regex 对象的 Match()方法与 Match 对象的 NextMatch()方法结合使用，可以枚举出一个测试字符串中的所有匹配项。关于如何连用这两个方法将在本章稍后进一步讨论。

Match()方法也可作为静态方法使用，本章后面也将讨论到。

10. Matches()方法

如果想找到一个测试字符串中所有的匹配项，则可以使用 Matches()方法。

试一试：使用 Matches() 方法

(1) 打开 Visual Studio 2003，在 Window Application 模板中创建一个新项目，并将它命名为 MatchesDemo。图 22-4 显示了此时的屏幕外观。根据 Visual Studio 2003 中设置的选项不同，在屏幕中看到的外观有可能会稍有不同。

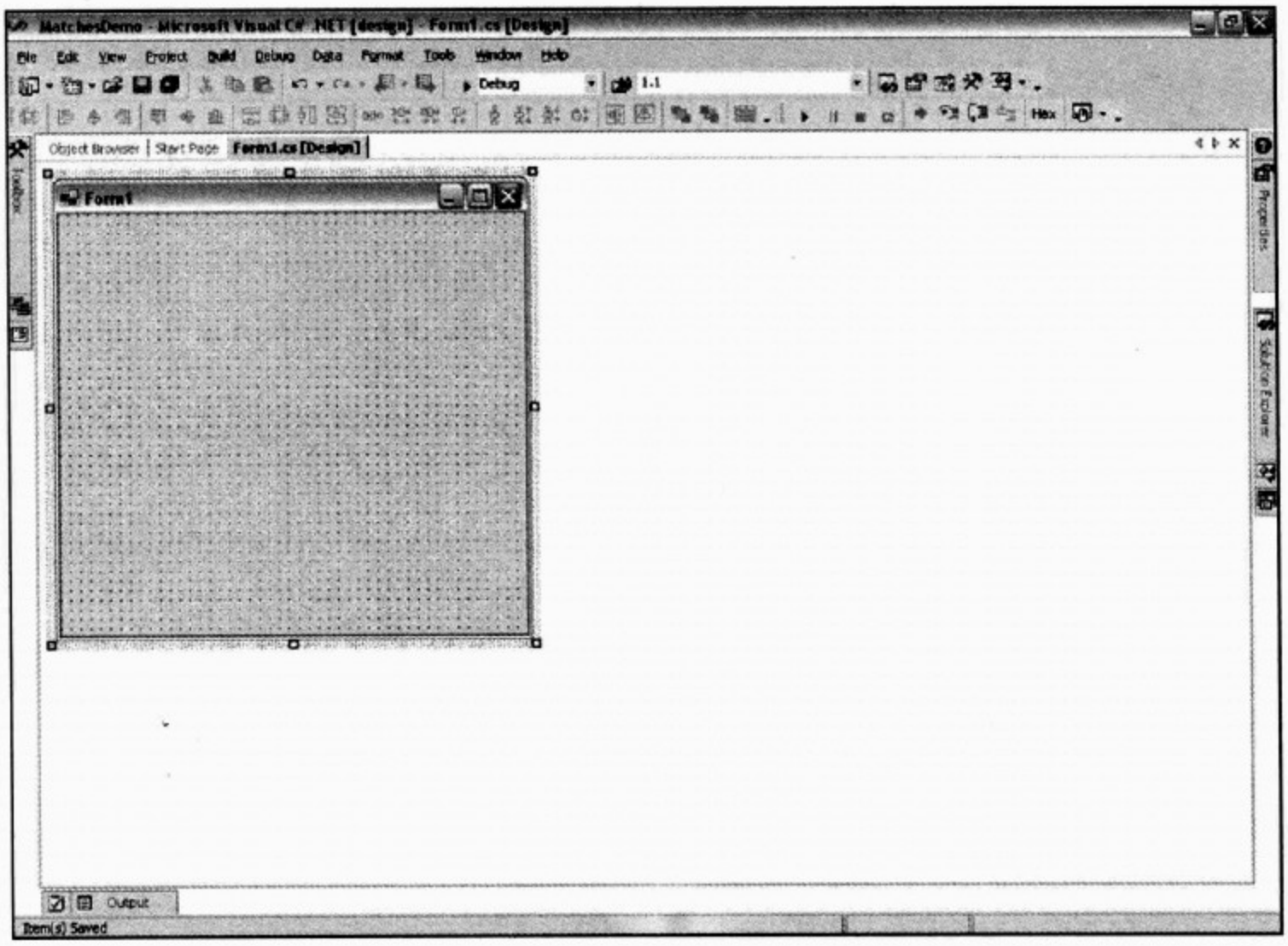


图 22-4

(2) 从 ToolBox 中拖出一个 Label 控件并放置在接近窗体顶部的位置，然后将其 Text 属性修改为 This form tests against the pattern '[A-Z]\d'。

(3) 从 ToolBox 中拖出一个 Label 控件放在窗体上，将其 Text 属性修改为 Enter a test string:。

(4) 从 ToolBox 中拖出一个 TextBox 控件放在窗体的设计界面上，将其 Text 属性置空。

(5) 向窗体中拖入一个 Button 控件，将其 Text 属性修改为 Click to Find Matches。

(6) 向窗体中拖入一个 TextBox 控件，并将其 Multline 属性修改为 True，将其 Text 属性置空。图 22-5 显示的是完成这一步后的界面。可能需要调整一下各个控件的位置和大小，以便形成与图 22-5 类似的外观。

在此阶段，窗体看上去比较整洁，但没有与之相关的功能。现在必须首先指定要使用 System.Text.RegularExpressions 命名空间的代码。

(7) 在 Solution Explorer 中，右击 Form1.cs 并选择 View Code 以打开代码编辑器窗口。如有必要，向上滚动窗口区，将会看到如下所示的 using 语句：

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

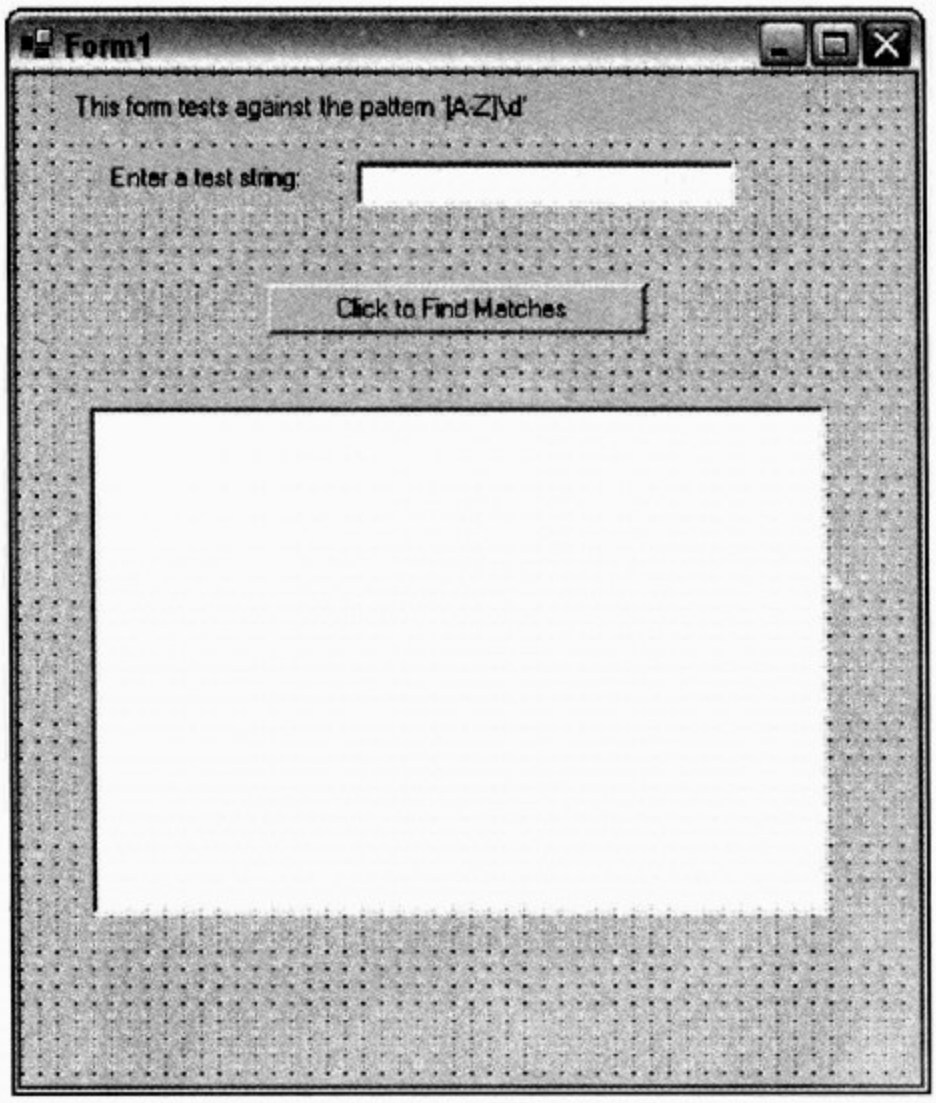


图　22-5

(8) 在自动添加的最后一条 using 语句之后，加入下面这行代码：

```
using System.Text.RegularExpressions;
```

(9) 返回设计界面，双击 Click to Find Matches 按钮，在打开的代码编辑器窗口中将包含下列自动创建的代码：

```
private void button1_Click(object sender, System.EventArgs e)
{
}
```

这个 button1_Click 事件处理程序会响应对该按钮的单击。现在需要为这个单击事件创建一些功能。

(10) 在代码编辑器窗口中，将下列代码添加到 button1_Click 事件处理程序的两个大括号之间：

```
Regex myRegex = new Regex(@"[A-Z]\d");
                string inputString;
                inputString = this.textBox1.ToString();
                MatchCollection myMatchCollection = myRegex.Matches(inputString);
                this.textBox2.Text = "The matches are:" + Environment.NewLine;
                foreach(Match myMatch in myMatchCollection)
                {
```

```
                    this.textBox2.Text += myMatch.ToString() +
Environment.NewLine;
                }
```

(11) 保存代码，按 F5 运行这些代码。如果代码没有运行，可以看一下错误任务列表中的错误信息。注意第一个错误中所提到的行号，找到该行并纠正相应的错误。然后按 F5 再次运行代码看问题有没有得到纠正。如果输入的代码没有问题，那么应该看到一个如图 22-6 所示的窗体。实际的外观取决于摆放窗体控件的位置及窗体的大小。

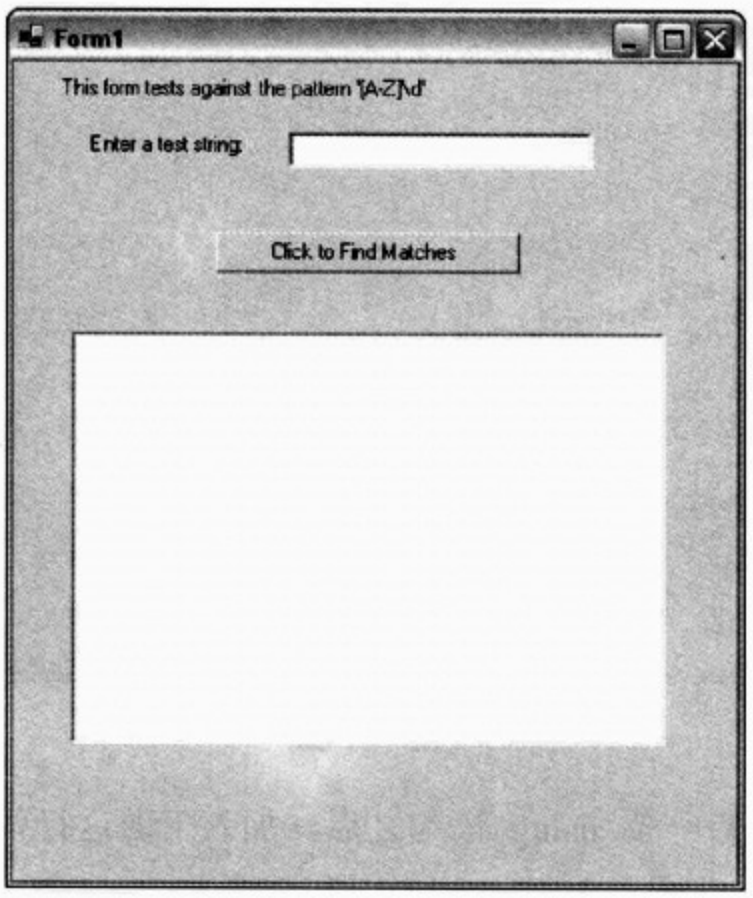


图 22-6

(12) 在上方的文本框中输入测试字符串 K99 L00 M11。

(13) 单击 Click to Find Matches 按钮，并观察显示在下方文本框中的结果，如图 22-7 所示。结果显示找到三个匹配项。

工作原理

为了使用 System.Text.RegularExpressions 命名空间中的类，必须添加适当的 using 指令：

```
using System.Text.RegularExpressions;
```

这个简单程序的任务是由 button1_Click 函数中的代码完成的。首先，声明一个继承自 Regex 类的对象变量 myRegex，并将以正则表达式模式 [A-Z]\d 实例化的 Regex 对象指定给它，注意在构造函数中使用了 @ 语法以省去在双引号中使用两个反斜杠的做法。

```
Regex myRegex = new Regex(@"[A-Z]\d");
```

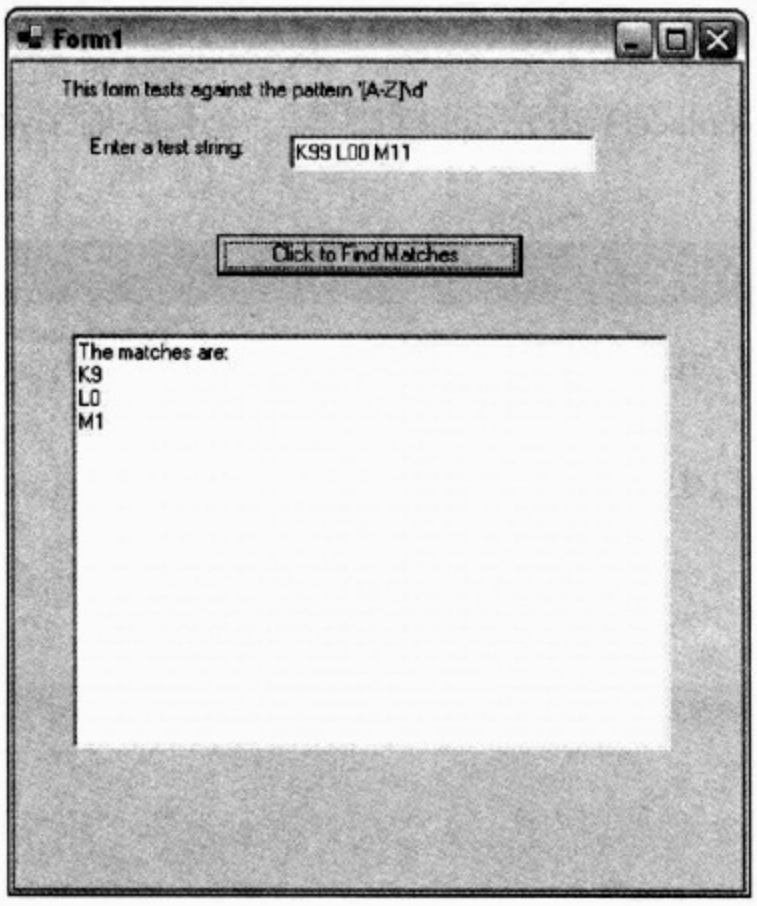


图　22-7

接着，声明一个字符串变量 inputString：

```
string inputString;
```

接着，将输入到 textBox1(上方的单行文本框)中的值指定给 inputString 变量：

```
inputString = this.textBox1.ToString();
```

接着，声明一个继承自 MatchCollection 类的对象变量 myMatchCollection。一个 MatchCollection 对象可以包含零个或多个 Match 对象。使用 myRegex 变量的 Matches() 方法，以 inputString 变量作为它的参数，将返回的值指定给 myMatchCollection 变量：

```
MatchCollection myMatchCollection = myRegex.Matches(inputString);
```

接着，指定一些直接量文本给 textBox2 的 Text 属性。其中的 Environment.Newline 用来将显示内容移至下一行：

```
this.textBox2.Text = "The matches are:" + Environment.NewLine;
```

然后使用 foreach 语句将 myMatchCollection 变量中包含的每一个 Match 对象的内容添加到 textBox2 中：

```
foreach(Match myMatch in myMatchCollection)
{
this.textBox2.Text += myMatch.ToString() + Environment.NewLine;
}
```

11. Replace()方法

通过 Regex 类的 Replace() 方法可以将匹配一个模式的字符序列替换成指定的模式或其他字符序列。

试一试：使用 Replace() 方法

(1) 在 Visual Studio 2003 中创建一个新的控制台程序，并将这个新项目命名为 SimpleReplace。

(2) 在代码编辑器窗口中，输入内容使之匹配下面的 Class1.cs：

```
using System;
using System.Text.RegularExpressions;

namespace SimpleReplace
{
  class Class1
  {
        [STAThread]
        static void Main(string[] args)
        {
    Console.WriteLine(@"This will find a match for the regular expression 'wrox'");
    Console.WriteLine(@"and replace it with 'Wrox'.");
    Console.WriteLine("Enter a test string now.");
  Regex myRegex = new Regex(@"wrox", RegexOptions.IgnoreCase);
    string inputString;
    inputString = Console.ReadLine();
    string newString = myRegex.Replace(inputString, "Wrox");
    Console.WriteLine("You entered the string '" + inputString + "'.");
   Console.WriteLine("After replacement the new string is '" + newString + "'.");
                Console.ReadLine();
        }
  }
}
```

确保要包含 using System.Text.RegularExpressions 指令。保存代码，并按 F5 来运行这些代码。

(3) 在命令窗口中，输入测试文本 This book is published by wrox.，然后按回车键并观察如图 22-8 所示的结果。结果显示字符序列 wrox(首字母小写)被替换成 Wrox(首字母大写)。

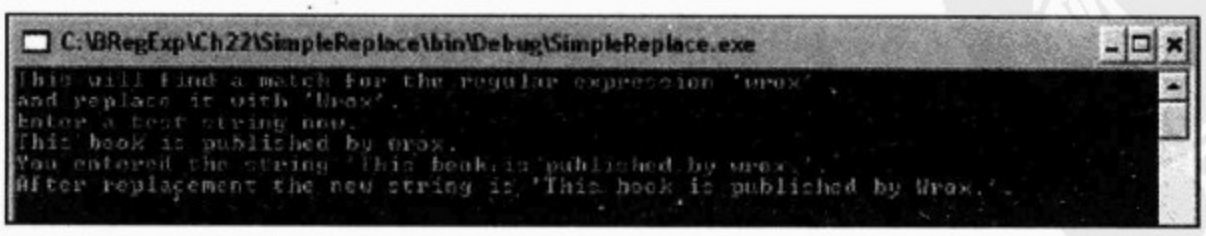


图 22-8

(4) 按回车键关闭命令窗口。

(5) 在 Visual Studio 中，按 F5 再次运行这些代码。

(6) 在命令窗口中，输入测试字符串 This book is published by WROX.，按回车键并观察结果。由于 IgnoreCase 选项所指定的——匹配不区分大小写，所以字符序列 WROX 也会匹配进而也会被替换成字符序列 Wrox。

工作原理

本例中的代码和以前的一样，也包含了 using System.Text.RegularExpressions 指令。

首先，显示一条消息告知用户该程序的意图是什么：

```
Console.WriteLine(@"This will find a match for the regular expression 'wrox'");
Console.WriteLine(@"and replace it with 'Wrox'.");
```

将一个简单的直接量模式 wrox 指定给 myRegex 对象变量。由于同时还指定了 IgnoreCase 选项，所以 wrox、Wrox 和 WROX 等这些字符序列都会被匹配：

```
Regex myRegex = new Regex(@"wrox", RegexOptions.IgnoreCase);
```

然后用户根据提示输入字符串，该字符串被指定给变量 inputString 。

使用 myRegex 对象的 Replace()方法将变量 inputString 中第一个 wrox 的实例(不区分大小写的匹配)替换成字符序列 Wrox：

```
string newString = myRegex.Replace(inputString, "Wrox");
Console.WriteLine("You entered the string '" + inputString + "'.");
Console.WriteLine("After replacement the new string is '" + newString + "'.");
```

当输入字符串中包含字符序列 wrox 时，它会被替换成 Wrox。而当输入字符串中包含 WROX 时，也会被替换成 Wrox。

12. Split()方法

Regex 类的 Split()方法会按一个正则表达式模式指定的位置拆分字符串。

可以在实例化的 Regex 对象中使用 Split()方法，也可以将其作为一个静态方法使用。

试一试：使用 Regex.Split() 方法

(1) 在 Visual Studio 2003 中使用 Windows Application 模板创建一个新项目。

(2) 向窗体中拖入一个标签，将其 Text 属性修改为 This demonstrates the Regex Split() method.。

(3) 向窗体中稍靠下的位置拖入一个标签，将其 Text 属性修改为 This will split a string when a comma is matched.。

(4) 向窗体中比第二个标签位置更低一些的位置拖入第三个标签，并将其 Text 属性修改为 Enter a string which includes commas:。

(5) 向窗体中拖入一个文本框，将其 Text 属性置空。

(6) 向窗体中拖入一个按钮，将其 Text 属性修改为 Click to split the string.。

(7) 整理窗体中控件的布局，使其与图 22-9 所示的界面类似。窗体外观可能会与此稍有不同，但并不会影响功能。

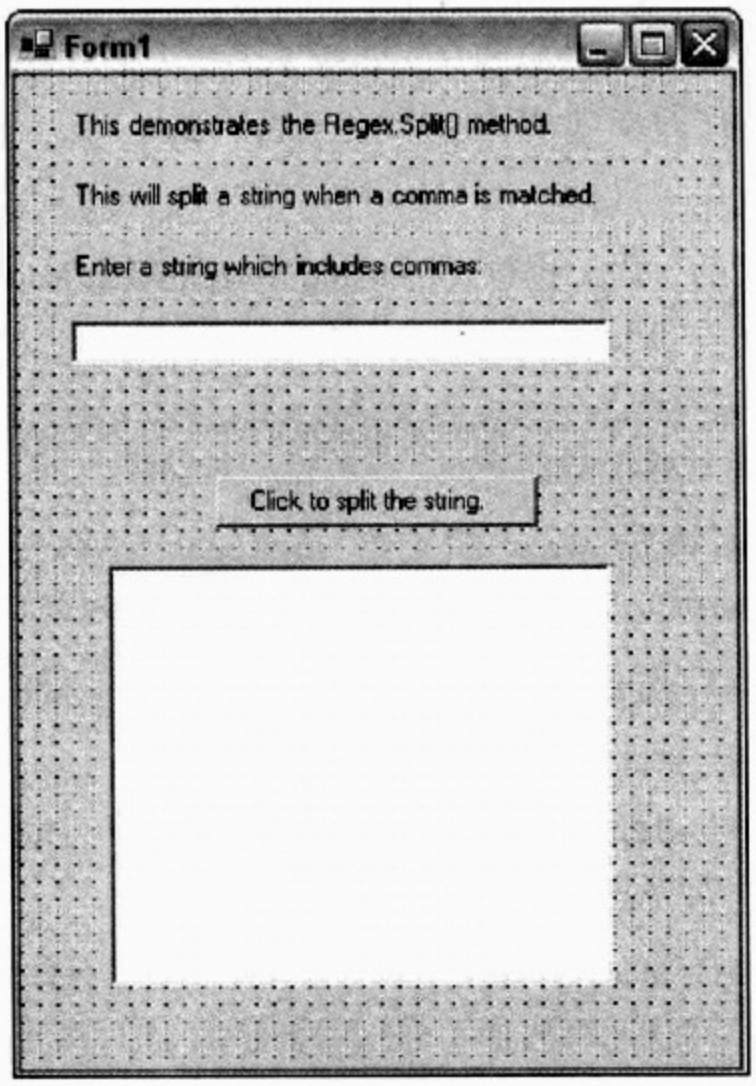


图 22-9

(8) 双击按钮，此时打开的代码编辑器窗口中会显示如下自动生成的代码：

```
private void button1_Click(object sender, System.EventArgs e)
{

}
```

(9) 向上滚动到代码窗口的顶端，在自动创建的 using 指令下方，插入下列代码：

```
using System.Text.RegularExpressions;
```

(10) 向下滚动到 button1_Click()函数，并在其中添加下列代码：

```
Regex myRegex = new Regex(",");
string inputString = this.textBox1.Text;
string[] splitResults;
splitResults = myRegex.Split(inputString);
this.textBox2.Text = "The string contained the following elements:" +
Environment.NewLine + Environment.NewLine;
foreach (string stringElement in splitResults)
this.textBox2.Text += stringElement + Environment.NewLine;
```

(11) 保存代码，按 F5 运行这些代码。

(12) 在上方的文本框中，输入文本 A1,B2,C12,D13，单击下方的按钮，并观察显示于

下方(多行)文本框中的结果，如图 22-10 所示。

工作原理

在 button1_Click()函数的代码中，首先声明一个 myRegex 变量，并赋予它一个逗号值。也就是说，myRegex 将匹配一个逗号。在这个例子中要以匹配正则表达式模式的位置来拆分测试字符串。

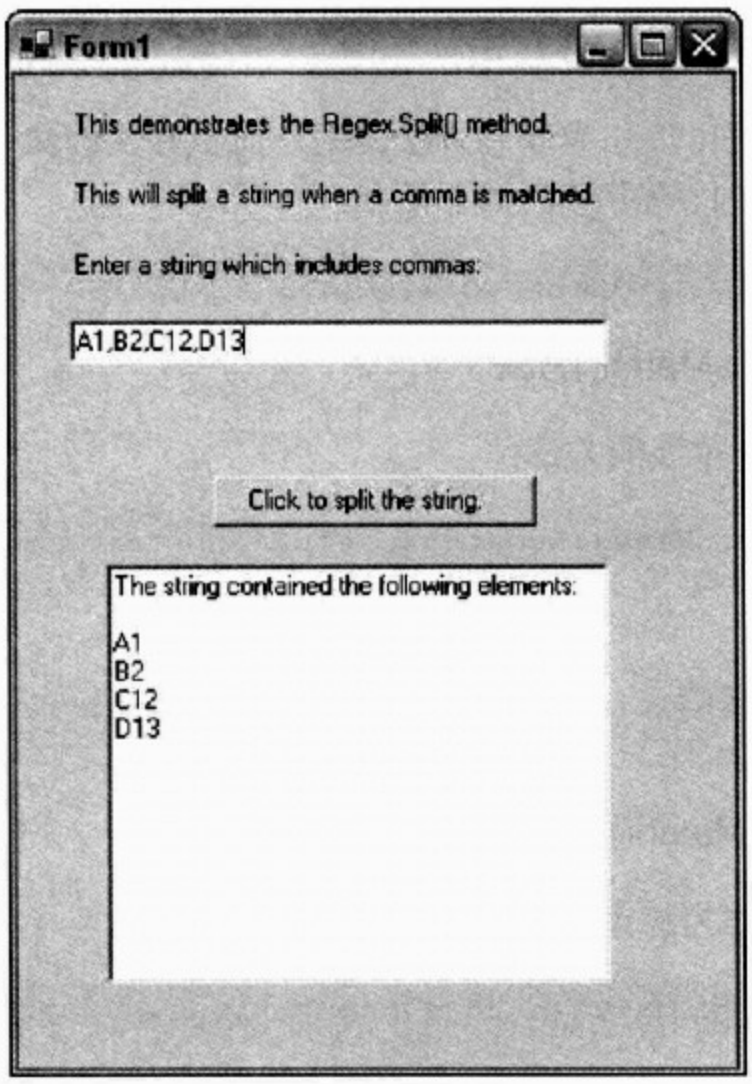


图　22-10

```
Regex myRegex = new Regex(",");
```

接着，声明变量 inputString 并将在上方文本框中输入的内容指定给该变量：

```
string inputString = this.textBox1.Text;
```

接着，声明一个字符串数组 splitResults：

```
string[] splitResults;
```

然后，将 inputString 变量应用 Split()方法后的结果指定给 splitResults 数组。该数组中的每一个元素分别包含着原始字符串中被逗号分隔的相邻的子字符串：

```
splitResults = myRegex.Split(inputString);
```

将一些基本的显示信息指定给 textBox2 的 Text 属性：

```
this.textBox2.Text = "The string contained the following elements:" +
Environment.NewLine + Environment.NewLine;
```

使用 foreach 循环将 splitResults 数组中的每个字符串值添加为 textBox2 的 Text 属性的值。由于使用了 Environment.Newline，数组的每一个元素值都会显示在单独的行中：

```
foreach (string stringElement in splitResults)
this.textBox2.Text += stringElement + Environment.NewLine;
```

22.1.4 使用 Regex 类的静态方法

Regex 类的一些方法可以作为静态方法使用，而不必实例化一个 Regex 类的对象。以下几个小节都假设代码中包含如下指令：

```
using System.Text.RegularExpressions;
```

1. 作为静态方法的 IsMatch()方法

IsMatch()方法有两个重载的方法：

```
public static bool Regex.IsMatch(string inputString, string pattern);
```

和

```
public static bool Regex.IsMatch(string inputString, string pattern, RegexOptions
options);
```

2. 作为静态方法的 Match()方法

Match()方法有两个作为静态方法的重载方法：

```
public static Match Match(string inputString, string pattern);
```

和

```
public static Match Match(string inputString, string pattern, RegexOptions
options);
```

3. 作为静态方法的 Matches()方法

Matches()方法有两个作为静态方法的重载方法：

```
public static MatchCollection Matches(string inputString, string pattern);
```

和

```
public static MatchCollection Matches(string inputString, string pattern,
RegexOptions options);
```

4. 作为静态方法的 Replace()方法

Replace()方法有两个作为静态方法的重载方法：

```
public static string Regex.Replace(string inputString, string pattern, string
```

```
replacementString);
```

和

```
public static string Regex.Replace(string inputString, string pattern, string
replacementString, RegexOptions options);
```

5. 作为静态方法的 Split()方法

Split()方法有两个作为静态方法的重载方法：

```
public static string[] Regex.Split(string inputString, string pattern);
```

和

```
public static string[] Regex.Split(string inputString, string pattern, RegexOptions
options);
```

22.1.5　Match 和 Matches 类

Match 类中包含一个独立的匹配项。而 MatchCollection 类中包含的是一个匹配项的集合。

Match 类

Match 类没有公共的构造函数。因此，该类必须通过其他类来存取。例如，可以通过 Regex 类的 Match()方法返回一个 Match 对象。

Match 对象有一个 Groups 属性。每个 Match 对象至少包含一个组。而 Match 对象等价于 Match.Groups[0]，因为第 0 个组中包含的是整个模式的匹配项。

Match 类的属性通过表 22-4 来介绍。

表 22-4　Match 类的属性

属　性	说　明
Captures	取得由捕获组捕获的文本集合。一个匹配项中可能包含零个或多个捕获文本
Empty	返回匹配失败
Groups	取得正则表达式匹配项的组的集合。假设存在一个匹配项，则该集合中至少存在一个组
Index	字符串中第一个成功匹配的字符所在的位置
Length	匹配的子字符串的长度
Success	表示匹配是否成功的值
Value	匹配的子字符串

表 22-5 罗列了 Match 对象的方法。这些方法并非都与使用正则表达式直接有关。

表 22-5　Match 对象的方法

方　法	说　明
Equals	判断两个对象是否相等
GetHashCode	取得散列码
GetType	取得当前对象实例的类型
NextMatch	如果存在一个匹配项，查找测试字符串中的下一个匹配项
Result	返回替换后的值
Synchronized	返回可以在线程间共享的一个 Match 对象
ToString	取得测试字符串中匹配的子字符串

NextMatch()方法可以与 Match()方法结合使用来完成对一个测试字符串中的匹配项进行迭代遍历。

试一试：使用 Match()方法和 NextMatch()方法

(1) 打开 Visual Studio 2003，使用 Windows Application 模板创建一个新项目，并将其命名为 MatchNextMatchDemo。

(2) 向窗体中拖入一个标签，将其 Text 属性修改为 Demo of the Match() and NextMatch() methods.。

(3) 向窗体设计界面中拖入一个标签，并将其 Text 属性修改为 This finds matches for the pattern '[A-Z]\d'.。

(4) 向窗体设计界面中拖入一个标签，并将其 Text 属性修改为 Enter a string in the text box below:。

(5) 向窗体设计界面中拖入一个文本框，并置空其 Text 属性。

(6) 向窗体设计界面中拖入一个按钮，将其 Text 属性修改为 Click to find all matches.。

(7) 向窗体设计界面中拖入一个文本框。将其 Multiline 属性设置为 True，并置空其 Text 属性。

(8) 调整窗体中控件的大小和位置，使当前的界面如图 22-11 所示。

(9) 在其他 using 指令下面添加如下代码：

```
using System.Text.RegularExpressions;
```

(10) 双击按钮创建 button1_Click()函数，并在函数中添加如下代码：

```
Regex myRegex = new Regex(@"[A-Z]\d");
string inputString = this.textBox1.Text;
Match myMatch = myRegex.Match(inputString);
this.textBox2.Text = "Here are all the matches:" + Environment.NewLine;
while (myMatch.Success)
{
  this.textBox2.Text += myMatch.ToString()+ Environment.NewLine;
  myMatch = myMatch.NextMatch();
```

(11) 保存代码，并按 F5 运行这些代码。

(12) 在上方的文本框中，输入文本 A11 B22 C33 D44。单击按钮，观察显示在下方文本框中的结果，如图 22-12 所示。

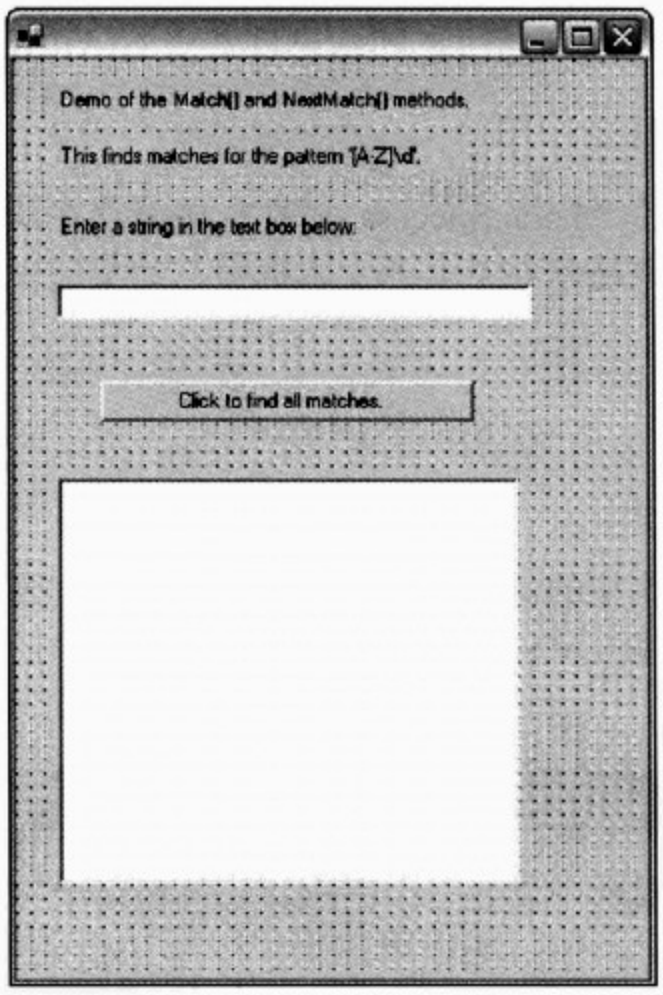


图　22-11

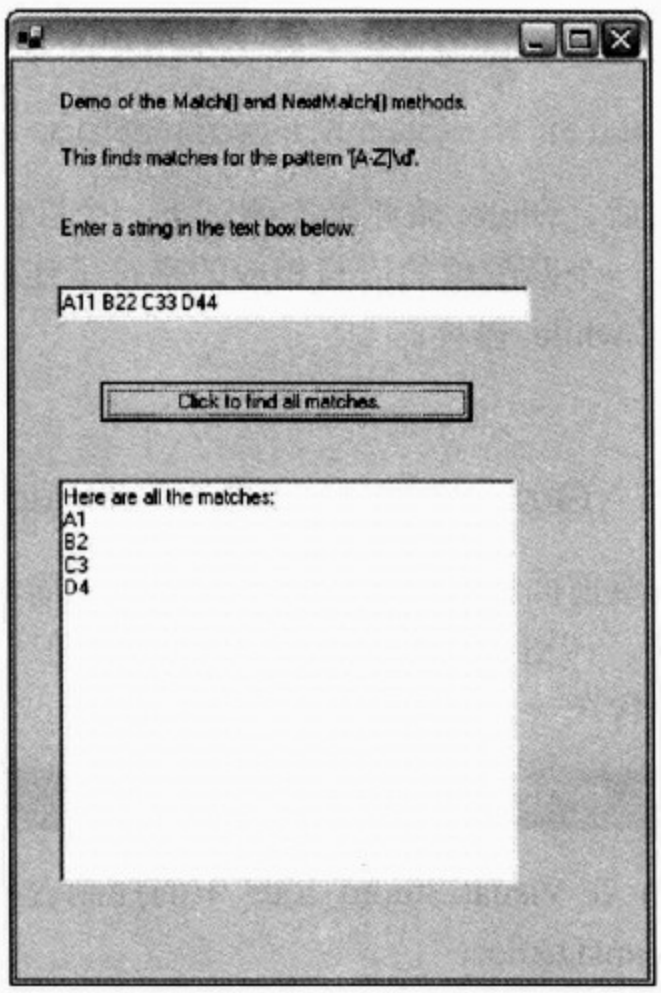


图　22-12

工作原理

下面介绍 button1_Click()函数中代码的运行过程。

声明 myRegex 变量，并用模式[A-Z]\d 实例化该对象：

```
Regex myRegex = new Regex(@"[A-Z]\d");
```

声明 inputString 变量，并将用户在上方文本框中输入的值指定给它：

```
string inputString = this.textBox1.Text;
```

声明 myMatch 变量，并将由 inputString 变量的 Match()方法返回的匹配项指定给它：

```
Match myMatch = myRegex.Match(inputString);
```

将一些说明性的文本指定给下方(多行)文本框的 Text 属性：

```
this.textBox2.Text = "Here are all the matches:" + Environment.NewLine;
```

利用 while 循环根据 Match 对象的 Success 属性来判断是否存在匹配项：

```
while (myMatch.Success)
{
```

在 while 循环内部，使用 ToString()方法取得匹配项的值并与相应的字符串连接起来显示于下方的文本框中：

```
this.textBox2.Text += myMatch.ToString()+ Environment.NewLine;
```

NextMatch()方法将测试字符串中的下一个匹配项并将返回结果指定给 myMatch 变量：

```
myMatch = myMatch.NextMatch();
```

然后，while 循环再次测试下一个匹配项的 Match.Success 属性，如果存在匹配项，则将下一个匹配项的值与相应的字符串连接起来显示在下方的文本框中。如果测试失败，则退出 while 循环：

```
}
```

22.1.6 GroupCollection 类和 Group 类

本章前面例子中使用的模式都比较简单。典型的情况下，模式中还会包含创建分组的圆括号。一次匹配中所有的组都被包含在一个 GroupCollection 对象中。该集合中的每个组又被包含在一个 Group 对象中。

试一试：使用 GroupCollection 类和 Group 类

(1) 在 Visual Studio 2003 中的控制台程序模板中创建一个新项目，并将这个项目命名为 GroupsDemo。

(2) 在代码编辑器窗口中，将下列代码输入到 Main()方法中。注意：正则表达式的第一行中使用了两对圆括号，它们将会创建两个捕获组。

```
Regex myRegex = new Regex(@"([A-Z])(\d+)");
Console.WriteLine("Enter a string on the following line:");
string inputString = Console.ReadLine();
MatchCollection myMatchCollection = myRegex.Matches(inputString);
Console.WriteLine();
Console.WriteLine("There are {0} matches.", myMatchCollection.Count);
Console.WriteLine();
GroupCollection myGroupCollection;

foreach (Match myMatch in myMatchCollection)
  {
  Console.WriteLine("At position {0}, the match '{1}' was found",
Match.Index,myMatch.ToString());
  myGroupCollection = myMatch.Groups;
  foreach (Group myGroup in myGroupCollection)
    {
    Console.WriteLine("Group containing '{0}' found at position '{1}'.",
 myGroup.Value, myGroup.Index);
    }
  Console.WriteLine();
 }
```

```
Console.WriteLine();
Console.WriteLine("Press Return to close this application.");
Console.ReadLine ();
```

(3) 保存代码，按 F5 运行这些代码。在命令窗口中，输入测试字符串 A12 B23 C34。

(4) 按回车键，并观察命令窗口中显示的结果，如图 22-13 所示。

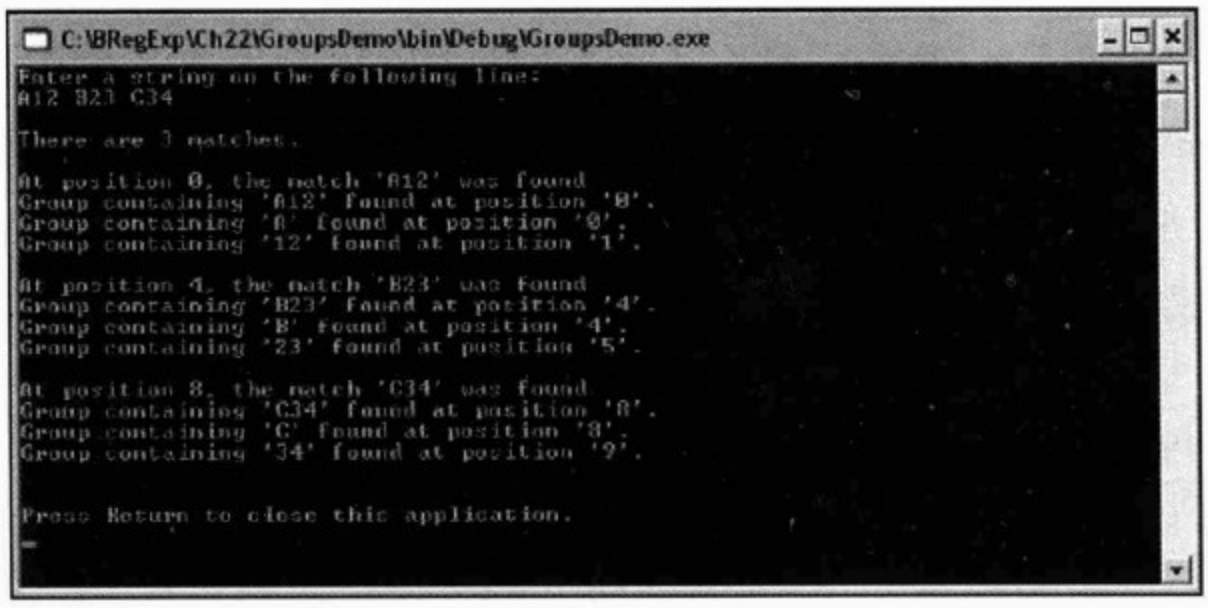


图　22-13

工作原理

详细分析一下在 Main() 方法中添加的代码。首先声明变量 myRegex 并为该对象指定了模式 [A-Z]\d+，该模式匹配一个大写的字母字符后跟一个或多个数字：

```
Regex myRegex = new Regex(@"([A-Z])(\d+)");
```

在向用户显示完输入测试字符串的提示后，声明 myMatchCollection 变量，并将由 inputString 变量的 Matches()方法返回的结果指定给 myMatchCollection 变量：

```
Console.WriteLine("Enter a string on the following line:");
string inputString = Console.ReadLine();
MatchCollection myMatchCollection = myRegex.Matches(inputString);
```

向命令窗口中写入一个空行后，显示保存在 myMatchCollection 对象的 Count 属性中的匹配项数量：

```
Console.WriteLine();
Console.WriteLine("There are {0} matches.", myMatchCollection.Count);
```

在显示了另一个空行后，声明一个继承自 GroupCollection 类的 myGroupCollection 变量：

```
Console.WriteLine();
GroupCollection myGroupCollection;
```

然后使用嵌套的 foreach 循环来处理每个匹配项以及包含在其中的每个组：

```
foreach (Match myMatch in myMatchCollection)
{
```

Match 对象的 Index 属性用来显示测试字符串中每个匹配项的位置，同时显示组成匹配项的子字符串：

```
Console.WriteLine("At position {0}, the match '{1}' was found", myMatch.Index,
myMatch.ToString());
```

在每个匹配项的 Groups 属性中，包含着与该匹配项对应的所有组的信息。将该属性指定给 myGroupCollection 变量：

```
myGroupCollection = myMatch.Groups;
```

然后，使用嵌套的 foreach 循环来处理 myGroupCollection 变量中的每个 Group 对象：

```
foreach (Group myGroup in myGroupCollection)
{
```

通过 Group 对象的 Value 属性显示每个组的内容，而 Group 对象的 Index 属性则显示该组在字符串中所处的位置：

```
Console.WriteLine("Group containing '{0}' found at position '{1}'.",
myGroup.Value,
myGroup.Index);
}
```

每个匹配中的第 0 个组包含整个匹配项的值。因为模式([AZ])(\d+) 包含两个额外的组(由两对圆括号创建)，所以每次处理内部的 foreach 循环时，都显示额外两个组的信息。也就是说，由于模式中包含两个可见的捕获组(由两对圆括号创建)，所以每个匹配项都会对应地显示三个组的信息。

最后，在每个执行外部的 foreach 循环时输出一个空行：

```
Console.WriteLine();
}
```

22.1.7 RegexOptions 类

RegexOptions 类是 System.Text.RegularExpressions 命名空间的一个成员，用来指定是否设置某个选项。

表 22-6 中总结了 RegexOptions 类可以指定的选项。

表 22-6 RegexOptions 类的选项

选　项	说　明
None	指定不设置选项
IgnoreCase	指定匹配不区分大小写
Multiline	将每一行都当做匹配过程中的独立字符串。因而 ^ 元字符(匹配每行的开始位置)和 $ 元字符(匹配每行的结束位置)的含义都会相应改变

(续表)

选　项	说　明
ExplicitCapture	改变圆括号的捕获行为
Compiled	指定是否将正则表达式编译为一个程序集
SingleLine	改变句点字符的含义，使其匹配任意一个字符。正常情况下，它匹配除 \n 外的任意一个字符
IgnorePatternWhitespace	不把模式中未转义的空格符解释为模式的组件。允许使用前置 # 的嵌入式注释
RightToLeft	指定匹配过程是从右往左搜索
ECMAScript	启用(受限制的)ECMAScript 兼容性
CultureInvariant	指定忽略语言间的文化差别

22.1.8　IgnorePatternWhitespace 选项

通过使用 IgnorePatternWhitespace 选项，允许使用嵌入式注释来详细说明正则表达式模式中每个组件的含义。

正常情况下，使用正则表达式模式匹配时，其中的任何空白符都是有意义的。例如，模式中的一个空格字符会作为一个字符来匹配。通过设置 IgnorePatternWhitespace 选项，模式中包含的所有空格符都将被忽略，包括空格符和换行符。这样，就可以为增进可读性而将单个模式分写在几行中，同时可以在其中添加注释，为将来维护正则表达式模式提供支持。

下面的介绍假定代码前面已经包含了 using System.Text.RegularExpressions 指令。

在 C#中，如果使用 myRegex 变量来匹配模式[A-Z]\d，那么可以像下面这样声明变量：

```
Regex myRegex = new Regex(@"[A-Z]\d");
```

但如果使用 IgnorePatternWhitespace 选项，则可以写成这样：

```
Regex myRegex = new Regex(

@"[A-Z]  # Matches a single upper case alphabetic character

\d       # Matches a single numeric digit",

RegexOptions.IgnorePatternWhitespace);
```

如上所示，可以将正则表达式模式的每个逻辑组件分别写在单独一行中，并通过前置 # 字符来为每个组件添加具有针对性的注释，从而使模式的构成更加清晰。当正则表达式模式既长又复杂时，使用这一选项来添加分行的注释尤其有用。

试一试：使用 IgnorePatternWhitespace 选项

(1) 在 Visual Studio 2003 中使用 Console Application 模板创建一个新项目，并将其命名为 IgnorePatternWhitespaceDemo。

(2) 在代码编辑器窗口中，将下列代码添加到默认的 using 语句后面：

```
using System.Text.RegularExpressions;
```

(3) 在 Main()方法中输入以下代码：

```
Regex myRegex = new Regex(
@"^ # match the position before the first character
\d{3} # Three numeric digits, followed by
-# a literal hyphen
\d{2} # then two numeric digits
-# then a literal hyphen
\d{4} # then four numeric digits
$# match the position after the last character",
RegexOptions.IgnorePatternWhitespace);
Console.WriteLine("Enter a string on the following line:");
string inputString = Console.ReadLine();
Match myMatch = myRegex.Match(inputString);
if (myMatch.ToString().Length != 0)
  {
  Console.WriteLine("The match, '" + myMatch.Value + "' was found.");
  }
else
{
  Console.WriteLine("There was no match");
}
Console.WriteLine("Press Return to close this application.");
Console.ReadLine();
```

(4) 保存代码，按 F5 来运行这些代码。

(5) 在命令行中，输入文本 123-12-1234(一个美国社会保险号)。按回车键，并观察如图 22-14 所示的结果。

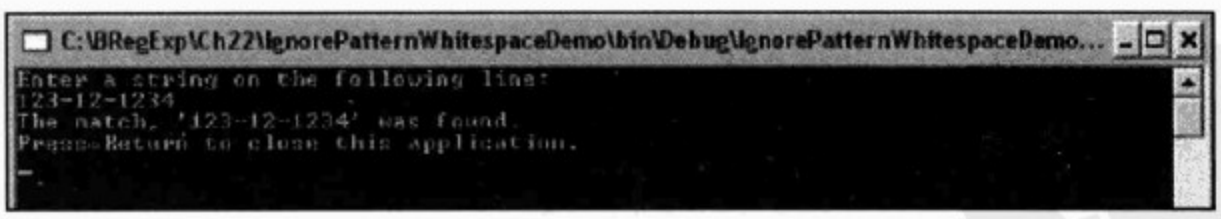


图 22-14

(6) 按回车键关闭程序。在 Visual Studio 2003 中按 F5 再次运行这些代码。

(7) 在命令行中输入文本 123-12-1234A，按回车键，并观察显示的消息。此时在输入的字符串中没有找到匹配项。

工作原理

本例中的代码寻找一个匹配的美国社会保险号，而且同一行中不能包含其他字符。

代码中有意思的部分是在设置 IgnorePatternWhitespace 选项的情况下如何编写正则表达式模式。

首先声明 myRegex 变量。不是写成：

```
Regex myRegex = new Regex(@"^\d{3}-\d{2}-\d{4}$";
```

当设置 IgnorePatternWhitespace 选项后，该模式可以写在多行中。其中的 @ 字符可以避免在 \d 组件前面再添加一个反斜杠。模式中的每个组件都可以写在单独的一行中，在 # 字符后可添加注释来详细说明每个组件的作用：

```
Regex myRegex = new Regex(
@"^ # match the position before the first character
\d{3} # Three numeric digits, followed by
-# a literal hyphen
\d{2} # then two numeric digits
-# then a literal hyphen
\d{4} # then four numeric digits
$# match the position after the last character",
```

最后，指定 IgnorePatternWhitespace 选项：

```
RegexOptions.IgnorePatternWhitespace);
```

模式 ^\d{3}-\d{2}-\d{4}$ 匹配一行的开始位置后跟三个数字、一个连字符、两个数字、一个连字符、四个数字，然后是一行的结束位置。匹配这一模式的字符串类似美国的社会保险号。还有更具体的模式能排除匹配这一模式但却无效的字符序列。因为本例的目的是示范 IgnorePatternWhitespace 选项的用途，所以在此就不再深究这个问题了。

22.2　Visual C# .NET 支持的元字符

Visual C# .NET 拥有非常完整和全面的正则表达式实现，其正则表达式的功能超过本书前几章介绍的许多工具。

Visual C# .NET 中支持的大多数正则表达式功能很多是标准的。然而，与许多其他 Microsoft 技术一样，某些标准的语法和技术也可能被扩展或修改。

表 22-7 中总结了 Visual C# .NET 支持的多种元字符。

表 22-7　Visual C# .NET 支持的元字符

元 字 符	说　明
\d	匹配一个数字
\D	匹配除数字外的任何字符
\w	等价于字符类 [A-Za-z0-9_]

(续表)

元 字 符	说 明
\W	等价于字符类 [^A-Za-z0-9_]
\b	匹配一个 \w 字符序列的开始或结束处的位置。通俗地说，\b 是一个匹配词边界的元字符
\B	匹配一个不是 \b 位置的位置
\t	匹配一个制表符
\n	匹配一个换行符
\040	匹配一个以八进制数表示的 ASCII 字符。元字符 \040 匹配一个空格符
\x020	匹配一个以十六进制表示的 ASCII 字符。\x020 元字符匹配一个空格符
\u0020	匹配一个以带有 4 位数字的十六进制表示的 Unicode 字符。\u0020 元字符匹配一个空格符
[...]	匹配字符类中指定的任何一个字符
[^...]	匹配字符类中未指定的任何一个字符
\s	匹配一个空白符
\S	匹配任何一个非空白符
^	如果设置 MultiLine 选项，它匹配一行中第一个字符前的位置；否则，匹配一个字符串中第一个字符前的位置
$	如果设置了 MultiLine 选项，它匹配一行中最后一个字符后的位置；否则，匹配一个字符串中最后一个字符后的位置
$number	替代与组号为 number 的组匹配的最后一个子字符序列
${name}	替代与组名为 name 的组匹配的最后一个子字符序列
\A	匹配一个字符串中第一个字符之前的位置。它的行为不受 MultiLine 选项的影响
\Z	匹配一个字符串中最后一个字符之后的位置。它的行为不受 MultiLine 选项的影响
\G	指定匹配必须是连贯的，即不包含任何居间的非匹配字符
?	限定符。匹配前面字符或组的零个或一个实例
*	限定符。匹配前面字符或组的零个或多个实例
+	限定符。匹配前面字符或组的一个或多个实例
{n}	限定符。匹配前面字符或组的 n 个实例
{n,m}	限定符。匹配前面字符或组最少 n 个、最多 m 个实例
(substring)	捕获包含的子字符串
(?<name>substring)	捕获包含的子字符串，并为其指定一个名称
(?:substring)	一个非捕获组
(?=...)	肯定式向前查找

(续表)

元 字 符	说　明
(?!...)	否定式向前查找
(?<=...)	肯定式向后查找
(?<!...)	否定式向后查找
\N (N 为数字)	对编号组的反向引用
\k<name>	引用一个命名的反向引用的反向引用(与下同)
\k'name'	引用一个命名的反向引用的反向引用(与上同)
|	交替选择
(?imnsx-imnsx)	一种指定 RegexOptions 设置的替换技术

22.2.1　使用命名的组

.NET 架构支持一种其他许多正则表达式实现都不支持的特性——命名的组。其语法是

```
(?<组名>模式)
```

对字符组命名比使用组编号更容易理解和维护。例如，分析下面的模式：

```
${lastName}, ${firstName}
```

在替换字符串中使用这种模式的命名组，要比在同样的替换操作中使用下面这样的编号组更容易理解：

```
${1}, ${2}
```

下面的例子使用命名的组来颠倒名和姓的位置。

试一试：使用命名的组

(1) 在 Visual Studio 2003 中使用 Console Application 模板创建一个新项目，并将其命名为 NamedGroupsDemo。

(2) 在代码编辑器窗口中，将下列代码添加到默认的 using 语句后面：

```
using System.Text.RegularExpressions;
```

(3) 在 Main()方法的两个大括号间输入下列代码：

```
  Console.WriteLine(@"This will find a match for the regular
expression '^(?<firstName>\w+)\s+(?<lastName>\w+)$'.");
  Console.WriteLine("Enter a test string consisting of a first name
then a last name.");
  string inputString;
  inputString = Console.ReadLine();
  string outputString = Regex.Replace(inputString,
@"^(?<firstName>\w+)\s+(?<lastName>\w+)$", "${lastName}, ${firstName}");
```

```
    Console.WriteLine("You entered the string: '" + inputString +
"'.");
    Console.WriteLine("The replaced string is '" + outputString +
"'.");
    Console.ReadLine();
```

(4) 保存代码，按 F5 运行这些代码。

(5) 在命令行中，输入测试字符串 John Smith，观察如图 22-15 所示的结果。

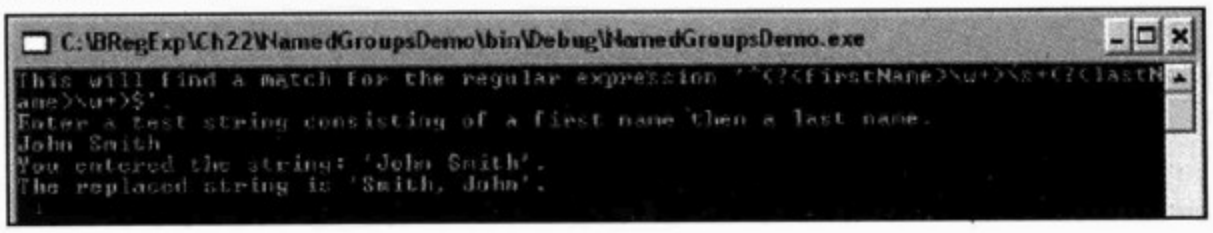


图 22-15

工作原理

下面解释 Main()中的代码。

首先，显示要匹配的模式，并请用户输入一个名和一个姓。要匹配的模式中包含两个命名的组，分别以 (?<firstName>\w+) 和 (?<lastName>\w+) 表示：

```
Console.WriteLine(@"This will find a match for the regular
expression '^(?<firstName>\w+)\s+(?<lastName>\w+)$'.");
Console.WriteLine("Enter a test string consisting of a first name
then a last name.");
```

声明 inputString 变量，并使用 Console.ReadLine() 方法捕获用户输入的字符串。将该字符串值指定给 inputString 变量：

```
string inputString;
inputString = Console.ReadLine();
```

Regex 类的 Replace()静态方法接受三个参数。第一个参数指定要替换的字符串——即，由 inputString 变量指定的字符串。第二个参数是用于匹配的模式—— 即，模式 ^(?<firstName>\w+)\s+(?<lastName>\w+)$。第三个参数是一个字符串值，其中使用语法 ${组名} 来表示每一个命名的组。

毋庸置疑，${firstName}组包含首先输入的字母字符序列，而 ${lastName}组则包含随后输入的字母字符序列：

```
string outputString = Regex.Replace(inputString,
@"^(?<firstName>\w+)\s+(?<lastName>\w+)$", "${lastName}, ${firstName}");
```

然后，向用户展示了他们输入的字符串和应用 Replace()方法之后生成的字符串：

```
Console.WriteLine("You entered the string: '" + inputString +
"'.");
Console.WriteLine("The replaced string is '" + outputString +
"'.");
Console.ReadLine();
```

22.2.2　使用反向引用

C# .NET 也支持反向引用。反向引用的一个典型用途就是查找重复的单词并删除它们。下面的例子示范如何使用反向引用。

试一试：使用反向引用

(1) 在 Visual Studio 2003 中使用 Console Application 模板创建一个新项目，并将该项目命名为 BackReferenceDemo。

(2) 添加一个 using System.Text.RegularExpressions 语句。

(3) 在代码编辑器窗口中，将下列代码添加到 Main()方法的方法体内：

```
  Console.WriteLine("This example will find a doubled word.");
  Console.WriteLine("Using a backreference and the Replace() method
the doubled word will be removed.");
  Console.WriteLine("Enter a test string containing a doubled
word.");
  string inputString;
  inputString = Console.ReadLine();
  string outputString = Regex.Replace(inputString, @"(\w+)\s+(\1)",
"${1}");
  Console.WriteLine("You entered the string: '" + inputString +
"'.");
  Console.WriteLine("The replaced string is '" + outputString +
"'.");
  Console.ReadLine();
```

(4) 保存代码，并按 F5 运行这些代码。

(5) 输入测试字符串 Paris in the the Spring(注意测试字符串中重复的 the)。按回车键，并观察如图 22-16 所示的结果中显示的信息。

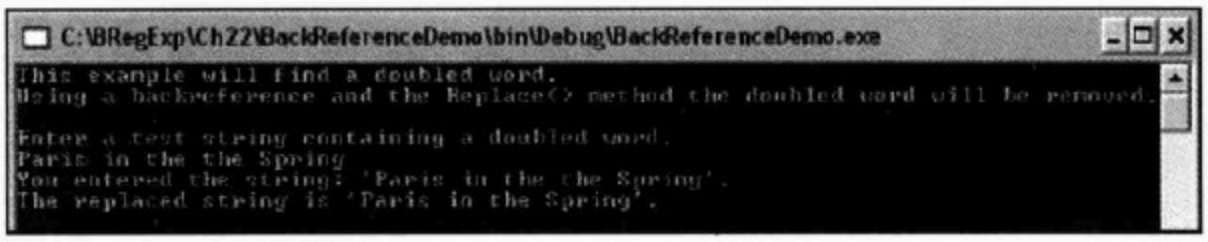


图　22-16

(6) 按回车键关闭程序。在 Visual Studio 中按 F5 再次运行代码。

(7) 输入测试字符串 Hello Hello，按回车键，并观察显示的信息。同样地，重复的单词被找到并被替换成一个单词。

工作原理

Main() 方法中的代码一开始就向用户显示使用反向引用的信息，并请用户输入包含重复单词的测试字符串：

```
  Console.WriteLine("This example will find a doubled word.");
  Console.WriteLine("Using a backreference and the Replace() method
```

```
the doubled word will be removed.");
Console.WriteLine("Enter a test string containing a doubled
word.");
```

声明 inputString 变量，并将用户输入的字符序列指定给它：

```
string inputString;
inputString = Console.ReadLine();
```

调用 Regex 类的静态 Replace() 方法，并将返回的结果指定给 outputString 变量。要匹配的正则表达式 (\w+)\s+(\1) 作为 Replace() 方法的第二个参数。该模式匹配一个等价于字符类 [A-Za-z0-9_] 的单词字母字符序列后跟一个或多个空白符，然后跟由反向引用 \1 表示的、前面已经匹配的同一个单词字符序列。换句话说，此模式匹配由空白符分隔的两个相同的单词。

Replace() 方法的第三个参数是用于替换的匹配文本的模式。匹配文本应该包含两个重复的单词(如果存在)。替换文本使用与反向引用对应的编号组——${1}，以一个单词替换两个单词：

```
string outputString = Regex.Replace(inputString, @"(\w+)\s+(\1)",
"${1}");
```

然后，将原始字符串和修改后的字符串显示给用户：

```
Console.WriteLine("You entered the string: '" + inputString +
"'.");
Console.WriteLine("The replaced string is '" + outputString +
"'.");
Console.ReadLine();
```

22.3 练习

1. 哪个 RegexOptions 选项是用于指定不区分大小写的匹配？

第23章

PHP 和正则表达式

PHP，即 PHP 超文本处理程序(PHP Hypertext Processor)，是在基于 Web 的应用程序中被广泛使用的一门语言。在基于 Web 的应用程序中，无论使用何种语言，在将数据写入到关系型数据库之前，都要在客户端或者服务器端对用户输入的数据进行验证。

PHP 通常用于开发服务器端的应用程序，与 ASP 和 ASP.NET 类似。要运行本章中的例子，需要在 Web 服务器中安装 PHP。

在本章中将学习以下内容：

- 如何使用 PHP 5.0
- PHP 中的组件如何支持正则表达式
- 如何使用 ereg()函数族
- PHP 的 Perl 兼容正则表达式(Perl Compatible Regular Expressions，PCRE)中支持哪些元字符
- 如何匹配公共需求的用户记录

本章介绍 PHP 5.0 所支持的正则表达式功能。

23.1 PHP 5.0 入门

要运行本章中的例子，必须在 Web 服务器中安装 PHP。因为本书主要介绍 Windows 平台下的正则表达式应用，所以要在 Windows IIS 服务器中安装 PHP。

随着 PHP 5.0 的出现，安装 PHP 的建议方法已经较之以前在 www.php.net 中推荐的方法有了显著不同。

PHP 的网站 www.php.net 是发布 PHP 最新消息的官方站点。本章主要介绍 PHP 5.0 的功能，但安装和配置的建议可能会改变。所以，建议查看上面的 URL 以便获悉当前的情况。

如果需要 PHP 4 而非 PHP 5，在本书写作之时，仍然能够从 www.php.net/download.php 中下载到。但是，如果出于兼容性的原因而需要 PHP 3，那么恐怕就要到 http://museum.php.net/去下载了。

下面的指示介绍了如何使用 Windows 安装程序包来安装 PHP 5.0.1，并且假设你已经安装了 IIS。Windows 安装程序包是在 Windows 中安装 PHP 的最简单的方式。虽然也能够使用以.zip 文件格式压缩的 PHP 安装文件手工安装 PHP，而且也允许对安装过程的完全控制，但因为本章内容介绍的是如何使用 PHP 中的正则表达式功能，而不是如何在 Web 服务器端安装 PHP，所以没有提供对如何下载、安装和配置 .zip 文件的说明。

下面的指示将帮你建立起运行 PHP 程序的环境。但是，没有考虑到安全安装 PHP 的问题。如果希望在一个开发服务器中使用 PHP，请确保投入足够的时间理解与在互联网中使用 PHP 相关的安全问题。

试一试：使用 Windows 安装程序安装 PHP

(1) 从 www.php.net 中的下载页面(在本书写作时，位于 www.php.net/downloads.php)下载 Windows 安装程序。下载完成后，双击 Windows 安装程序包。图 23-1 显示的是 PHP 5.0.1 安装程序包的初始界面。

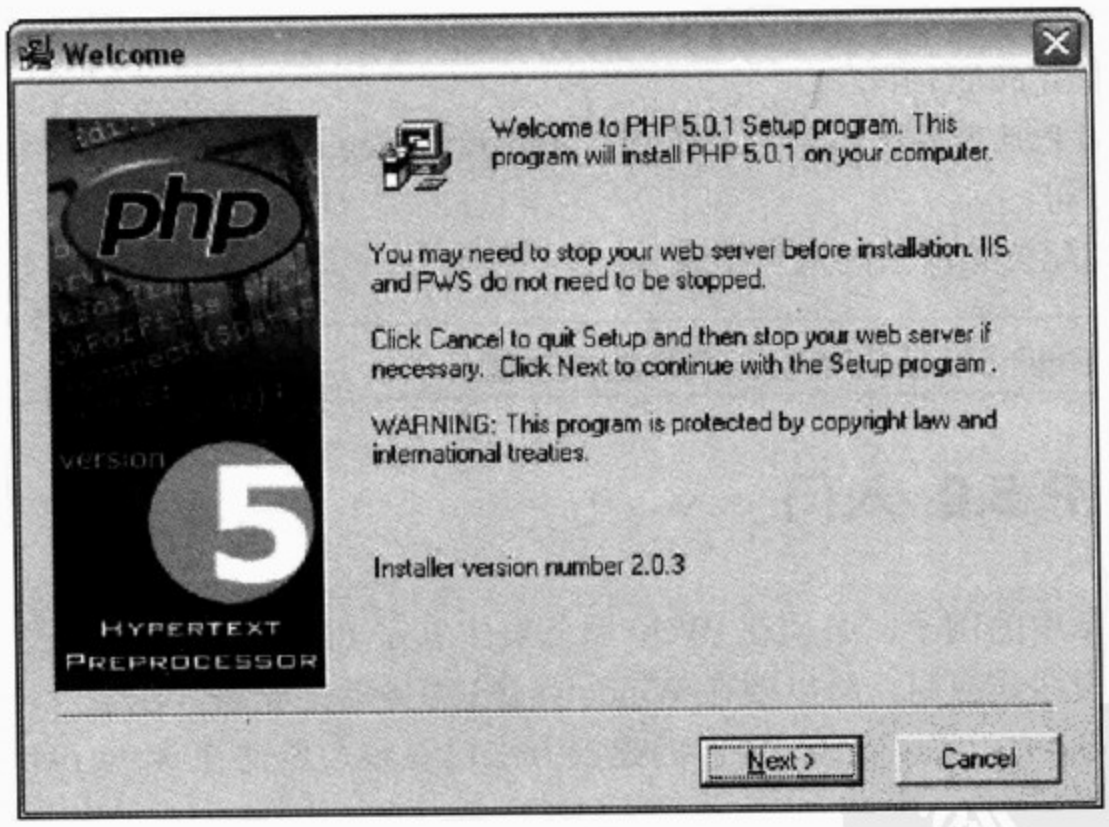


图 23-1

(2) 单击 Next 按钮。阅读 License Agreement 并单击 I Agree 按钮。如果不同意协议中的内容，将无法继续安装 PHP 5.0.1。

(3) 在下一个界面中，会显示 Standard 和 Advanced 两个安装选项，以供选择。选择 Advanced，并单击 Next 按钮。

(4) 选择安装位置，如 C:\PHP 5.0.1。单击 Next 按钮。

(5) 然后系统会问是否希望备份在安装过程中被替换的文件。使用默认选项——Yes，然后再单击 Next 按钮。

(6) 接受默认的上传目录，并单击 Next 按钮。接受默认的会话信息存放目录，并单击 Next 按钮。

(7) 接受 localhost 作为 SMTP 服务器位置或者进行适当修改。出于测试安装的原因，建议接受 localhost，然后单击 Next 按钮。

(8) 接受默认的警告和错误选项，并单击 Next 按钮。

(9) 在下一个界面中，安装程序可能会识别出所安装的 IIS 或 Personal Web Server(PWS)。除非有其他需要，否则还是接受默认选择并单击 Next 按钮。

(10) 选择与 PHP 关联的文件扩展名。建议将扩展名限定为 .php(除非有特殊需求)。单击 Next 按钮。

(11) 在下一个界面中，会看到有关安装程序已经拥有了完成安装所必需的信息的提示。单击 Next 按钮，安装程序会显示有关安装过程的信息。如果一切顺利，则会看到如图 23-2 所示的信息。

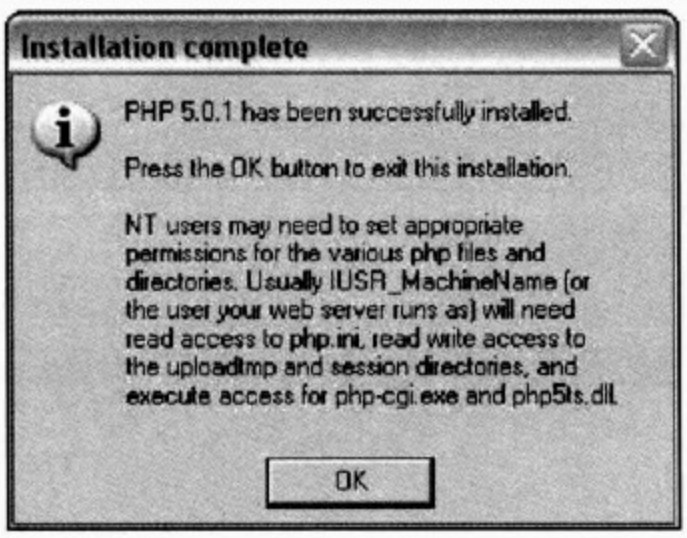


图　23-2

现在，PHP 已经安装完成。接下来需要测试它能否与 IIS 正确地协同工作。下面的指示假设你安装了在本地机器中运行的 IIS，而且默认的 IIS 内容目录位于 C:\inetpub\wwwroot\。如果你的设置不一样，可以相应地调整下面的指示。

(12) 在“记事本”程序或者其他文本编译器中，输入以下代码：

```
<?php
phpinfo()
?>
```

(13) 在 C:\inetpub\wwwroot\ 或其他目录中创建 PHP 目录。这样就可通过 URL http://localhost/PHP/加上相关的 PHP 文件名来访问 PHP 代码了。

(14) 将该文件命名为 phpinfo.php 并保存到 PHP 目录中。如果是在“记事本”程序中保存文件，确保文件名加上一对双引号，否则“记事本”程序会将文件保存为 phpinfo.php.txt，这样使用 Web 浏览器打开该文件时就不能正确运行了。

(15) 打开 Internet Explorer 或其他浏览器，在地址栏中输入 URL http://localhost/PHP/phpinfo.php 并按回车。

图 23-3 显示的是在这一步之后应该看到的结果。当然，如果没有使用 Internet Explorer 6.0 或 PHP 5.0.1，界面肯定与图 23-3 不一样。然而，只要看到类似于图 23-3 所示的结果，就说明已经成功安装了 PHP。

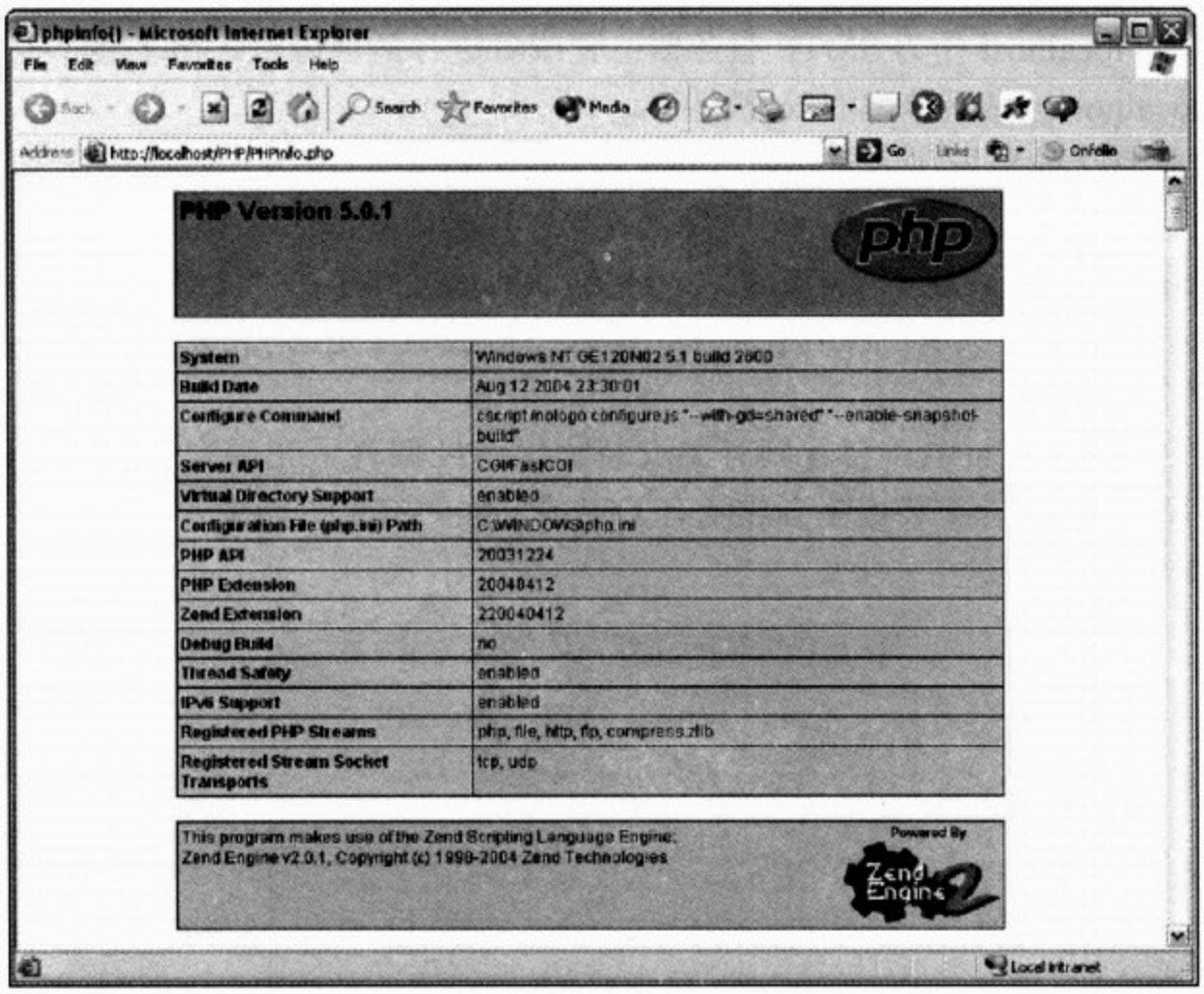


图 23-3

(16) 使用 Ctrl+F 快捷键在网页中搜索文本 PCRE。如图 23-4 所示，使用 Windows 安装程序安装的 PHP 5.0.1 默认支持 PCRE 功能。由于本章中有些例子依赖于 PCRE 功能，所以确认该功能启用非常重要。

在 PHP 安装成功之后，就可以开始研究 PHP 5.0 是如何支持正则表达式的了。

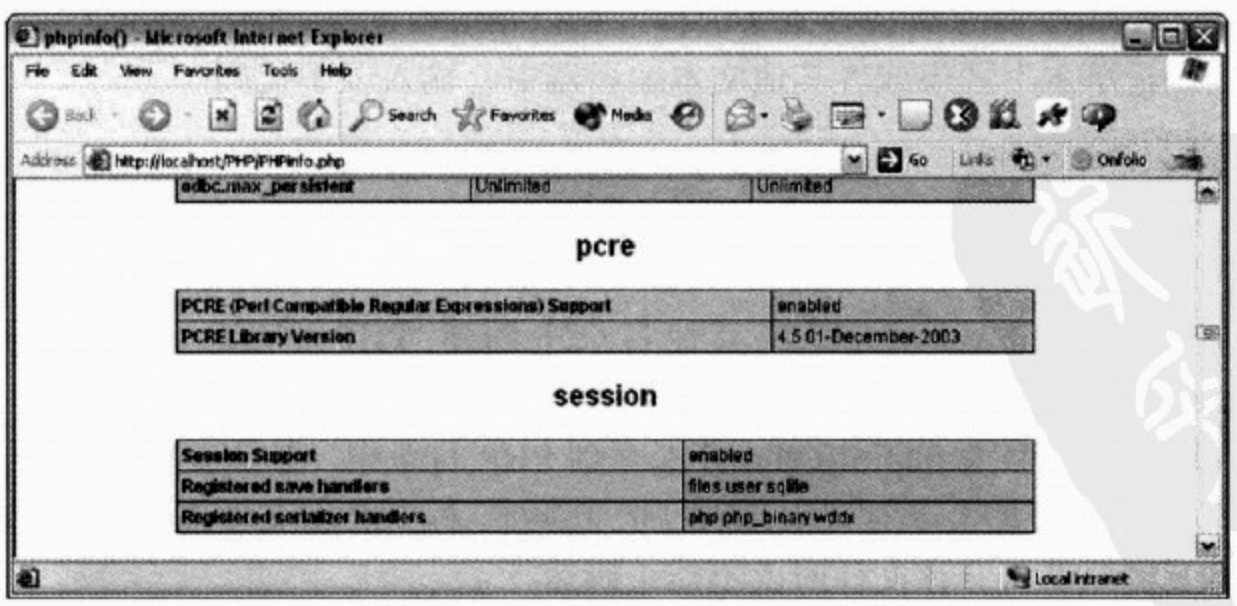


图 23-4

23.2　PHP 组件如何支持正则表达式

PHP 通过两组不同的函数来提供对正则表达式的支持。第一组，ereg() 及相关函数，已经在 PHP 中使用一段时间了(可以在 PHP 3 Web 服务器中使用该函数)。另一组，是新加入 PHP 中的 Perl 兼容正则表达式(PCRE)，具有更多的正则表达式功能。本章的大部分内容和例子都涉及到 PCRE 功能。

如果需要兼容 PHP 较早的版本，那么 ereg() 函数集可能是唯一的选择。而另一方面，如果需要使用 ereg() 函数族中缺少的功能，PCRE 有可能是最佳的选择。当然如有必要的话，首先升级 Web 服务器中的 PHP 版本。

23.2.1　ereg() 函数集

ereg() 函数集以 POSIX 正则表达式为基础。下面的表 23-1 中总结了 ereg()函数集所包含的函数。

表 23-1　ereg()函数集中包含的函数

函　数	说　明
ereg()	以区分大小写的方式将一个字符串与正则表达式模式进行匹配
eregi()	以不区分大小写的方式将一个字符串与正则表达式模式进行匹配
ereg_replace()	以区分大小写的方式匹配一个正则表达式模式，如果存在匹配项则替换匹配项
eregi_replace()	以不区分大小写的方式匹配一个正则表达式模式，如果存在匹配项则替换匹配项
split()	基于在区分大小写的情况下匹配正则表达式，将字符串拆分为一个子字符串数组
spliti()	基于在不区分大小写的情况下匹配正则表达式，将字符串拆分为一个子字符串数组
sql_regcase()	根据指定的字符串创建一个在不区分大小写的情况下与之匹配的有效正则表达式模式

1. ereg()函数

ereg()函数进行区分大小写的匹配。它接受两个或三个参数。当 ereg() 使用两个参数时，第一个参数是一个字符串值，表示正则表达式模式；第二个参数是测试字符串。

例如，如果想查找在测试字符串 The theatre is a favorite of thespians. 中是否存在与直接量模式 the 匹配的内容，那么就可以使用下列代码：

```
ereg('the'', "The theatre is a favorite of thespians");
```

因为 ereg()执行区分大小写的匹配，所以会找到两个匹配项：theatre 和 thespians 的前三个字符。

在介绍带三个参数的 ereg()函数之前，先来试验一个接受两个参数的例子。这个例子示范了通过 ereg()函数使用直接量正则表达式模式 Hel 来匹配直接量字符串值 Hello world!

的简单过程。

试一试：一个简单的 ereg() 函数的例子

(1) 在文本编辑器中输入下列代码。如果没有其他编辑器，使用“记事本”程序也可以。

```
<html>
<head>
<title>Simple ereg() Regex Test</title>
</head>
<body>
<?php
if (ereg("Hel", "Hello world!")) echo "<p>A match was found.</p>"
?>
</body>
</html>
```

(2) 将文件保存为 C:\inetpub\PHP\SimpleRegexTest.php。如果 Web 服务器不在本地机器中或者 PHP 目录在其他地方，请修改这里的路径。

(3) 在 Web 浏览器地址栏中，输入 URL http://localhost/PHP/ SimpleRegexTest.php，按回车键并观察网页显示的内容。如图 23-5 所示，会看到 A match was found.。

图 23-5

工作原理

在 SimpleRegexTest.php 中包含 HTML/XHTML 标记和 PHP 代码。PHP 代码非常简单：

```
<?php
if (ereg("Hel", "Hello world!")) echo "<p>A match was found.</p>"
?>
```

<?php 标记表示 PHP 代码的开始，而 ?> 标记表示 PHP 代码的结束。中部的代码行则是 PHP 代码本身。

代码中的 if 语句测试了 ereg()函数的返回值。此时的 ereg()函数带有两个参数，第一个是字符串 Hel，该字符串将被解释为一个直接量正则表达式模式；第二个参数是字符串 Hello world!，也就是测试字符串。换句话说，PHP 处理器会尝试在 Hello world! 中查找与模式 Hel 匹配的内容。结果没有什么意外——找到了一个匹配项，所以 ereg()函数返回 1，

表示存在一个匹配项。而值 1 被解释为等价于 True，所以由 if 控制的代码会被执行：

```
echo "<p>A match was found.</p>"
```

通常，该 PHP 语句会以一个分号结尾。因为在这个简单的例子中只有一行 PHP 代码，所以不需要加分号。而在本章其他例子中，如果忽略了 PHP 结尾处的分号，那么将不能正确运行代码。

echo 语句会将一个字符串插入到 PHP 代码在网页 SimpleRegexTest.php 中的相应位置。也就是说，程序会在服务器端将包含 HTML/XHTML 标记的代码插入到开始的<body>和结束的</body>标记之间。而当把网页交付给客户端时，其源代码如图 23-6 所示。

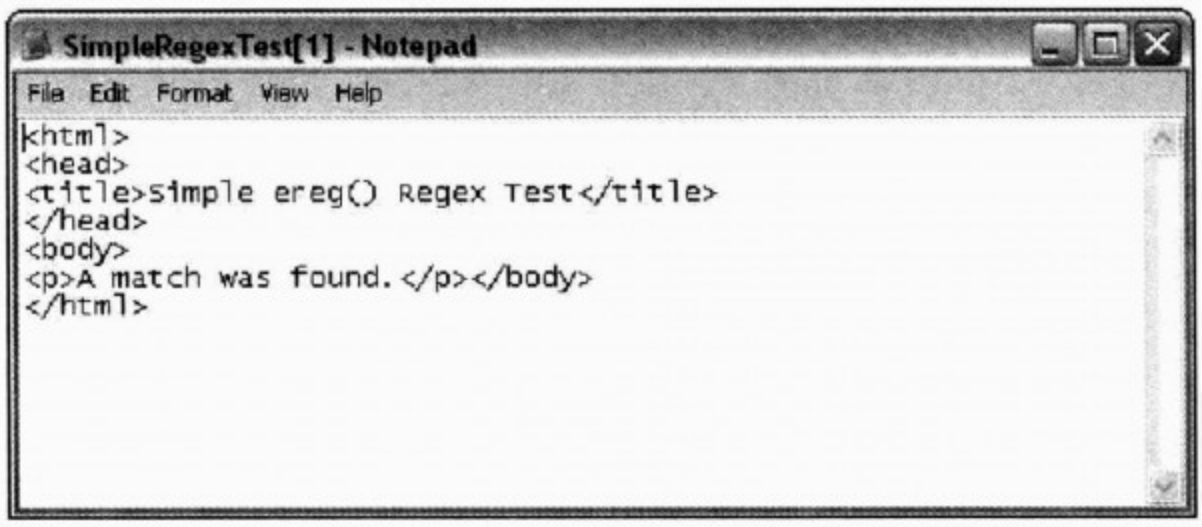


图　23-6

本例简单地使用由 ereg()函数返回的布尔值来控制是否向网页中添加直接量文本。在下一个例子中，将看到如何通过使用 ereg()函数来操纵个别匹配项的内容。

2. 带三个参数的 ereg()函数

当 ereg()函数接受三个参数时，第一个参数仍然是由字符串值表示的正则表达式模式，第二个参数仍然是测试字符串，而第三个参数指定的则是一个用于保存匹配项结果(包括捕获组值)的数组。

试一试：使用带有三个参数的 ereg()函数

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>Splitting a date using ereg()</title>
</head>
<body>
<?php
$myPattern = '([12][0-9]{3}) ([01][0-9]) ([0123][0-9])';
$testString = gmdate("Y m d");
echo "<p>The date is now: $testString</p>";
$myResult = ereg($myPattern, $testString, $matches);
```

```
if ($myResult)
{
echo "<p>A match was found when testing case sensitively.</p>";
echo "<p>Expressed in MM/DD/YYYY format the date is now,
$matches[2]/$matches[3]/$matches[1].</p>";
}
else
{
echo "<p>No match was found when testing case sensitively.</p>";
}
?>
</body>
</html>
```

2. 将文件命名为 SplitDate.php 并保存到 IIS 安装目录下的 PHP 目录中。也就是说，将代码保存为 C:\inetpub\wwwroot\PHP\SplitDate.php。

3. 在 Internet Explorer 的地址栏中输入 URL http://localhost/PHP/SplitDate.php，按回车键并观察结果。如图 23-7 所示，代码运行于 2004 年 9 月 22 日。在页面中的最后一行，该日期被显示为 09/22/2004，即已经对由 gmdate()函数返回的原始格式的日期值 2004 09 22 进行了重新排列。

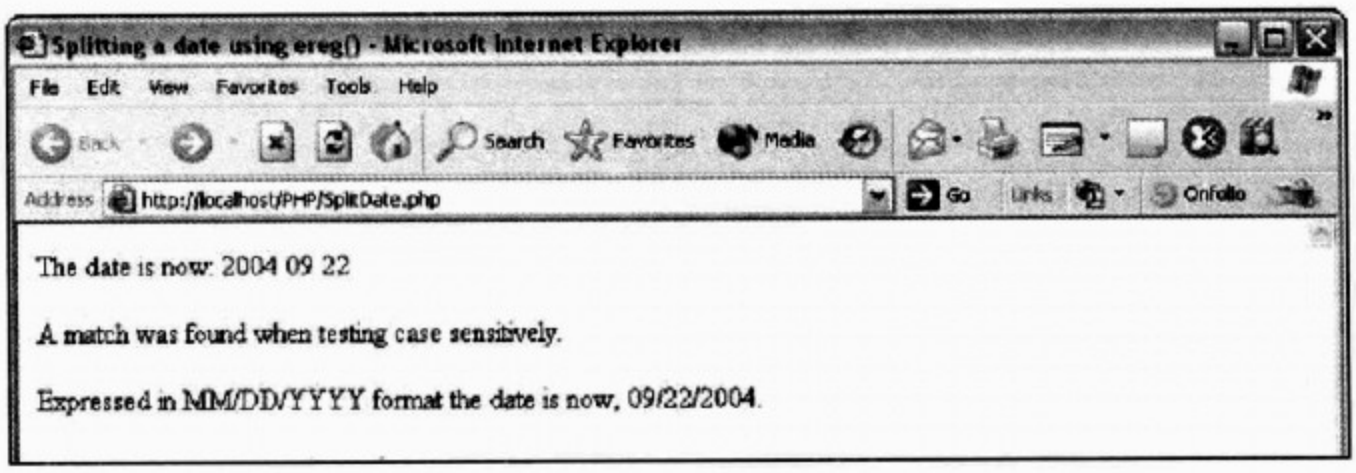


图 23-7

工作原理

首先，看一下本例使用的模式：

```
([12][0-9]{3}) ([01][0-9]) ([0123][0-9])
```

因为日期由 PHP 的 gmdate() 函数生成，实际上不用担心该函数返回日期的有效性。所以，这里可以使用比较粗略的模式。

该模式首先匹配一个数字 1 或 2。这样就可以满足 20 世纪或 21 世纪年份的第一位数字。然后，匹配三个数字。模式的这一部分包含在一对圆括号中，构成捕获组。稍后会看到如何取得并处理这个捕获组的值。也就是说，模式中的这个组件匹配 20 世纪或 21 世纪中四位数字的年份。

然后是一个空格符。接着又是一个捕获组，该组以 0 或 1 开头，然后再跟一个数字。这个组件用于匹配月份的值。从理论上讲，这个组件也会匹配一年中的第 19 个“月”。不过，由于 gmdate() 函数不可能返回这样的日期值，所以在本例中无须担心这个问题。

然后还是一个空格符和第三个捕获组，该组以 0、1、2 或 3 开头，后跟一个数字。这个组件匹配一月中的天数。从理论上讲，该组件会匹配一月份的第 39 “天”。不过，还是那个原因——gmdate() 不可能返回这种无效日期，所以不用为此担心。

首先，将这个模式作为字符串值指定给变量 $myPattern：

```
$myPattern = '([12][0-9]{3}) ([01][0-9]) ([0123][0-9])';
```

然后，将 gmdate() 函数的返回值指定给变量 $testString。传递给 gmdate() 函数的参数指定了要返回的日期格式。本例中，Y 表示返回四位数的年份 2004；m 表示使用 1～12 之间的数字来表示月份；而 d 表示用 1～31 之间的数字来表示一月中的天数：

```
$testString = gmdate("Y m d");
```

然后，使用 echo 语句显示格式化的当前日期。如果习惯使用 JavaScript 语言，那么此刻可能会觉得直接把变量 $testString 放到双引号中，而不是使用 + 连接操作符有点不可思议。但在 PHP 中把变量放在双引号内部时，PHP 处理器会知道其意图就是要使用该变量的值：

```
echo "<p>The date is now: $testString</p>";
```

然后，将接受三个参数的 ereg() 函数的返回值指定给变量 $myResult。如果存在一个或多个匹配项，变量 $myResult 会包含布尔值 True(1)；若没有匹配项，该变量则包含布尔值 False(0)。

和以前一样，第一个参数是模式，第二个参数是测试字符串。而第三个参数是包含匹配文本组件的数组。该数组中的元素由捕获圆括号决定，这些捕获组在分析正则表达式模式时介绍过：

```
$myResult = ereg($myPattern, $testString, $matches);
```

测试是否存在匹配项：

```
if ($myResult)
```

如果存在，则使用两个 echo 语句将内容添加到网页中：

```
{
```

简单地声明一下找到了匹配项：

```
echo "<p>A match was found when testing case sensitively.</p>";
```

然后，说明将以 MM/DD/YYYY 的格式返回日期，也就是说必须对日期部件进行重新排序。通过使用数组元素的组合 $matches[2]/$matches[3]/$matches[1] 就可实现这一点。

整个匹配项的值保存在数组的第 0 个元素中。模式匹配的日期包含三对捕获圆括号，所以第一个捕获组的值被保存在 $matches[1] 中，第二个捕获组的值被保存在 $matches[2] 中，依此类推。结果，$matches[1] 保存年份，$matches[2] 保存月份，而 $matches[3] 保存天数。

因为要返回的格式为 MM/DD/YYYY，所以使用序列 $matches[2]/$matches[3]/$matches[1]。这意味着要返回 $matches[2] 中的值(月份)后跟一个直接量正斜杠，后跟 $matches[3] 中的值(天数)，后跟一个直接量正斜杠，再后跟 $matches [1] 中的值(年份)。

这样，最初显示为 2004 09 20 的日期值就会转换成 09/20/2004：

```
echo "<p>Expressed in MM/DD/YYYY format the date is now,
$matches[2]/$matches[3]/$matches[1].</p>";
```

如果不存在匹配项，只须简单地显示一条相关信息即可：

```
}
else
{
echo "<p>No match was found when testing case sensitively.</p>";
}
```

也就是说，如果为 ereg() 函数指定了一个变量作为其第三个参数，那么使用圆括号捕获的组就会分别保存在该变量数组的一个元素中。

23.2.2 eregi()函数

eregi() 函数除了执行不区分大小写的匹配之外，与 ereg() 函数几乎没有什么不同。下面的例子显示了对相同的测试文本应用 ereg() 和 eregi() 的结果。

试一试：使用 eregi() 函数

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>ereg() and eregi() Test to match [A-Z][0-9]</title>
</head>
<body>
<?php
$myPattern = '[A-Z][0-9]';
$testString = "a9";
$myResult = ereg($myPattern, $testString);
if ($myResult)
{
echo "<p>A match was found when testing case sensitively.</p>";
}
else
{
echo "<p>No match was found when testing case sensitively.</p>";
}
$myResult2 = eregi($myPattern, $testString);
if ($myResult2)
{
echo "<p>A match was found when testing case insensitively.</p>";
}
```

```
else
{
echo "<p>No match was found when testing case insensitively.</p>";
}
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\EregEregiTest.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/EregEregiTest.php。按回车键，并观察如图 23-8 所示的结果。根据显示的第一条信息，我们知道在使用 ereg()函数时，没有找到匹配项。而根据第二条信息，我们知道在使用 eregi()时，存在一个匹配项。

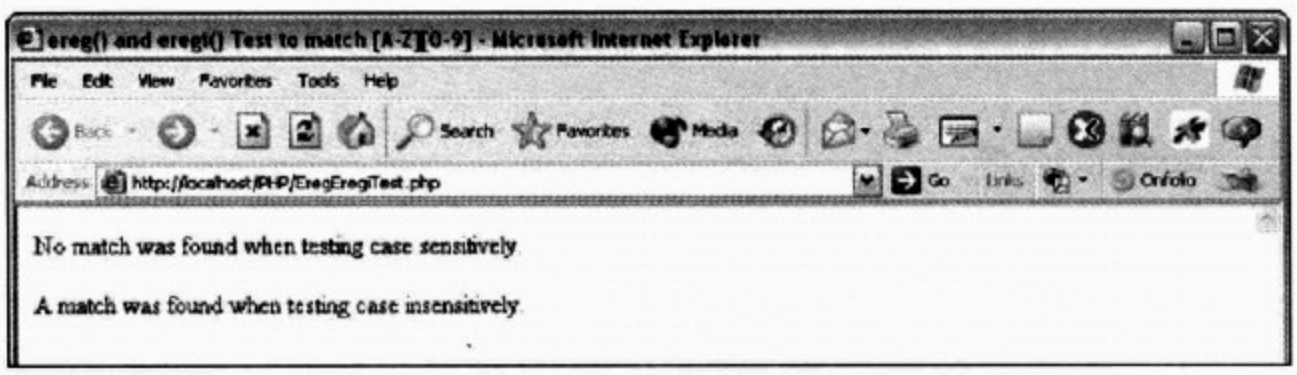


图　23-8

工作原理

将模式 [A-Z][0-9] 指定给变量 $myPattern。此处，由于 ereg()函数不支持 \d 元字符，所以不能使用 \d 来匹配数字：

```
$myPattern = '[A-Z][0-9]';
```

将测试字符串 a9 指定给变量 $testString：

```
$testString = "a9";
```

ereg()函数接受两个参数，第一个参数是$myPattern，第二个参数是$testString。将该函数返回的结果指定给变量$myResult。这样，就可以测试模式[A-Z][0-9] 在测试字符串 a9 中是否存在匹配项：

```
$myResult = ereg($myPattern, $testString);
```

如果存在匹配项，$myResult 中包含布尔值 True(1)。如果没有匹配项，则会包含布尔值 False(0)。通过这两个布尔值来决定显示给用户的信息：

```
if ($myResult)
```

测试字符串 a9 中的第一个字符，不包含在字符类[A-Z]中，因为匹配过程区分大小写。

当使用 ereg()函数时，变量 $myResult 会返回 False。也就是说，在区分大小写的匹配中没有发现匹配项，所以会显示 No match was found when testing case sensitively.：

```
else
{
echo "<p>No match was found when testing case sensitively.</p>";
}
```

但是，当使用不区分大小写的 eregi() 函数时，测试字符串 a9 的第一个字符匹配字符类 [A-Z]，而该测试字符串的第二个字符也匹配字符类 [0-9]。因为模式的每一个组件都找到了匹配项，所以变量 $myResult2 中包含的是布尔值 True。

由于$myResult2 中包含布尔值 True，则会显示 A match was found when testing case insensitively：

```
if ($myResult2)
{
echo "<p>A match was found when testing case insensitively.</p>";
}
```

1. ereg_replace()函数

ereg_replace() 函数尝试以区分大小写的方式匹配一个模式。如果找到了一个匹配项，该匹配项将会被指定的替换文本替换。如果存在多个匹配项，每个匹配项都会被替换。

ereg_replace() 函数接受三个参数。第一个参数是用于匹配的模式；第二个参数是替换匹配项的字符串；第三个参数则是要替换其中匹配项的文本。

试一试：使用 ereg_replace() 函数

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>ereg_replace() Demo</title>
</head>
<body>
<?php
$myPattern = "Hello";
$myReplacement = "Hi";
$myString = "Hello world!";
echo "<p>The original string was '$myString'.</p>";
echo "<p>The pattern is '$myPattern'.</p>";
echo "<p>The replacement text is '$myReplacement'.</p>";
$replacedString = ereg_replace($myPattern, $myReplacement, $myString);
$displayString = "<p>After replacement the string becomes: ' ";
$displayString = $displayString . $replacedString . "'.</p>";
echo $displayString;
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\ereg_replaceDemo.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/ereg_replaceDemo.php。按回车

键，并观察显示的结果。如图 23-9 所示，原始字符序列中的 Hello 被替换成了 Hi。

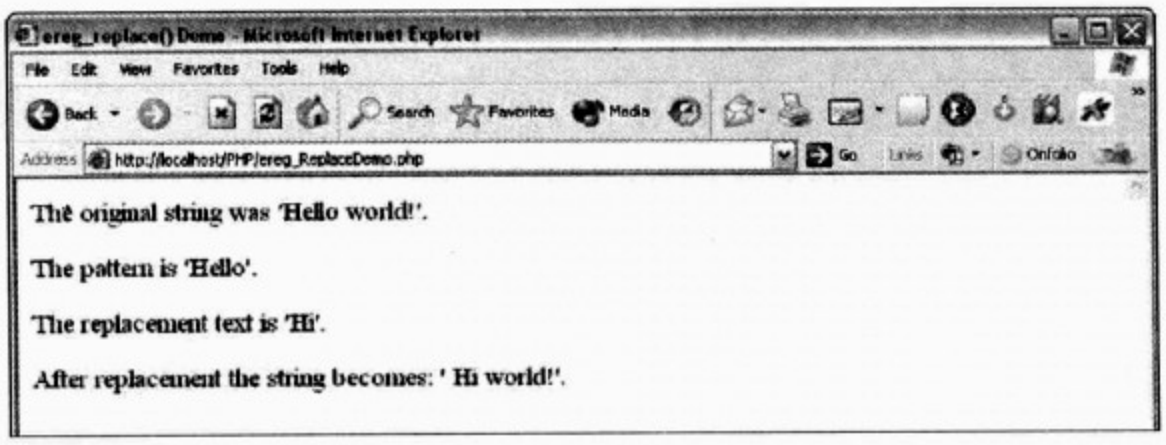


图　23-9

工作原理

声明三个变量$myPattern、$myReplacement 和$myString，并分别为它们赋予字符串值：

```
$myPattern = "Hello";
$myReplacement = "Hi";
$myString = "Hello world!";
```

然后，通过 echo 语句将它们的原始值返回给用户：

```
echo "<p>The original string was '$myString'.</p>";
echo "<p>The pattern is '$myPattern'.</p>";
echo "<p>The replacement text is '$myReplacement'.</p>";
```

声明变量$replacedString，并将 ereg_replace()函数的返回值指定给它。注意传递给函数 ereg_replace()的三个参数的顺序，其中测试字符串是第三个参数，不再是传递给 ereg()函数时的第二个：

```
$replacedString = ereg_replace($myPattern, $myReplacement, $myString);
```

最后，声明$displayString 变量。该变量的值由直接量文本和 $replacedString 变量的值连接而成。注意，在 PHP 中句点字符用于连接字符串值：

```
$displayString = "<p>After replacement the string becomes: ' ";
$displayString = $displayString . $replacedString . "'.</p>";
echo $displayString;
```

如果该模式没有匹配项，则仍然会返回原始的字符串。

2. eregi_replace()函数

eregi_replace()函数以不区分大小写的方式匹配并用替换字符串替换在测试字符串中找到的匹配项。

eregi_replace()函数接受三个参数。第一个是以字符串值表示的正则表达式模式；第二个是用于替换测试字符串中匹配项的字符串；第三个是测试字符串。

试一试：使用 eregi_replace() 函数

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>eregi_replace() Demo</title>
</head>
<body>
<?php
$myPattern = "Doctor";
$myReplacement = "Dr.";
$myString = "Doctor Smith spoke with another doctor.";
echo "<p>The original string was '$myString'.</p>";
echo "<p>The pattern is '$myPattern'.</p>";
echo "<p>The replacement text is '$myReplacement'.</p>";
$replacedString = eregi_replace($myPattern, $myReplacement, $myString);
$displayString = "<p>After replacment the string becomes: ' ";
$displayString = $displayString . $replacedString . "'.</p>";
echo $displayString;
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\eregi_replaceDemo.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/eregi_replaceDemo.php，按回车键并观察网页中显示的结果。如图 23-10 所示，两个单词 Doctor 的实例(一个是 Doctor，另一个是 doctor)被替换了。但多数情况下，第二个实例可能并非人们想替换的。

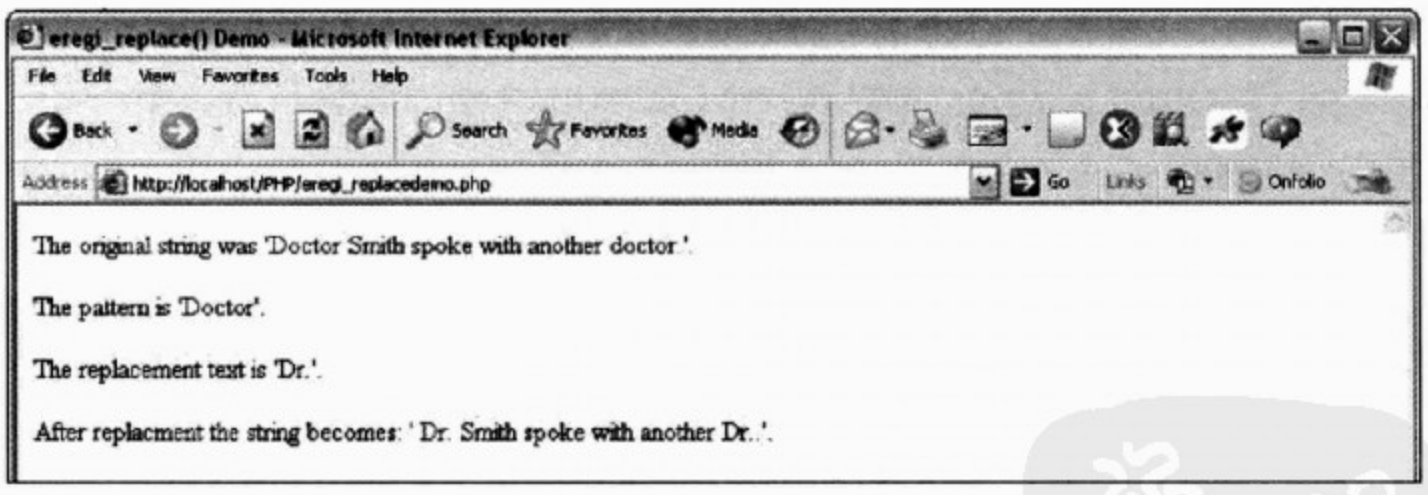


图 23-10

工作原理

本例的目标是以不区分大小写的方式进行匹配并将所有字符串 Doctor 的实例替换成其缩写形式 Dr.。

首先声明变量$myPattern、$myReplacement 和$myString，并为其赋值：

```
$myPattern = "Doctor";
```

```
$myReplacement = "Dr.";
$myString = "Doctor Smith spoke with another doctor.";
```

在将上述变量的值显示给用户之后，使用 eregi_replace() 函数来以不区分大小写的方式匹配并以 Dr. 替换所有 Doctor 的实例：

```
$replacedString = eregi_replace($myPattern, $myReplacement, $myString);
```

这会导致(测试字符串中的)Doctor 和 doctor 都被替换，因为模式 Doctor 的匹配是不区分大小写的。这降低了匹配过程的特殊性，因为诸如 DocTOr、dOCtor 之类的字符序列也将被匹配。但在多数情况下，把 doctor 替换成 Dr. 都是不合适的。所以，对于特定的任务而言，可能使用区分大小写的 ereg_replace() 函数会更合适一些。但本例的目的只是示范不区分大小写的匹配过程。

3. split()函数

通过 split()函数，可以根据与一个正则表达式匹配的字符(序列)或位置将字符串拆分成独立的子字符串。这个函数在处理文件中以逗号分隔的值或使用其他不同分隔符的数据时非常有用。下面的例子将示范如何使用 split()函数拆分以连字符、正斜杠以及句点字符分隔的日期部件。

split()函数可以接受两个或三个参数。第一个参数是表示拆分点的正则表达式模式。如果存在匹配项，则该匹配项会被丢弃。这种行为作为默认行为是有意义的，例如对于以逗号分隔的数据，一般而言我们想要的只是数据而不是逗号。第二个参数是测试字符串。第三个参数(可选)是一个整数值，该值用于指定匹配和拆分的最大次数。

试一试：使用 split() 函数

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>split() Function Demo</title>
</head>
<body>
<?php
$myString1 = "2004/09/23";
$myString2 = "2004.09.23";
$myString3 = "2004-09-23";
$myPattern = "[-/.]";
list($year, $month, $day) = split($myPattern, $myString1);
echo "<p>String was: $myString1. <br />Year: $year <br />Month: $month <br />Day:
$day</p>";
list($year, $month, $day) = split($myPattern, $myString2);
echo "<br /><br /><p>String was: $myString2. <br />Year: $year <br />Month: $month
<br />Day: $day</p>";
list($year, $month, $day) = split($myPattern, $myString3);
echo "<br /><br /><p>String was: $myString3. <br />Year: $year <br />Month: $month
<br />Day: $day</p>";
```

```
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\splitdemo.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/splitdemo.php，按下回车键并观察如图 23-11 所示的结果。

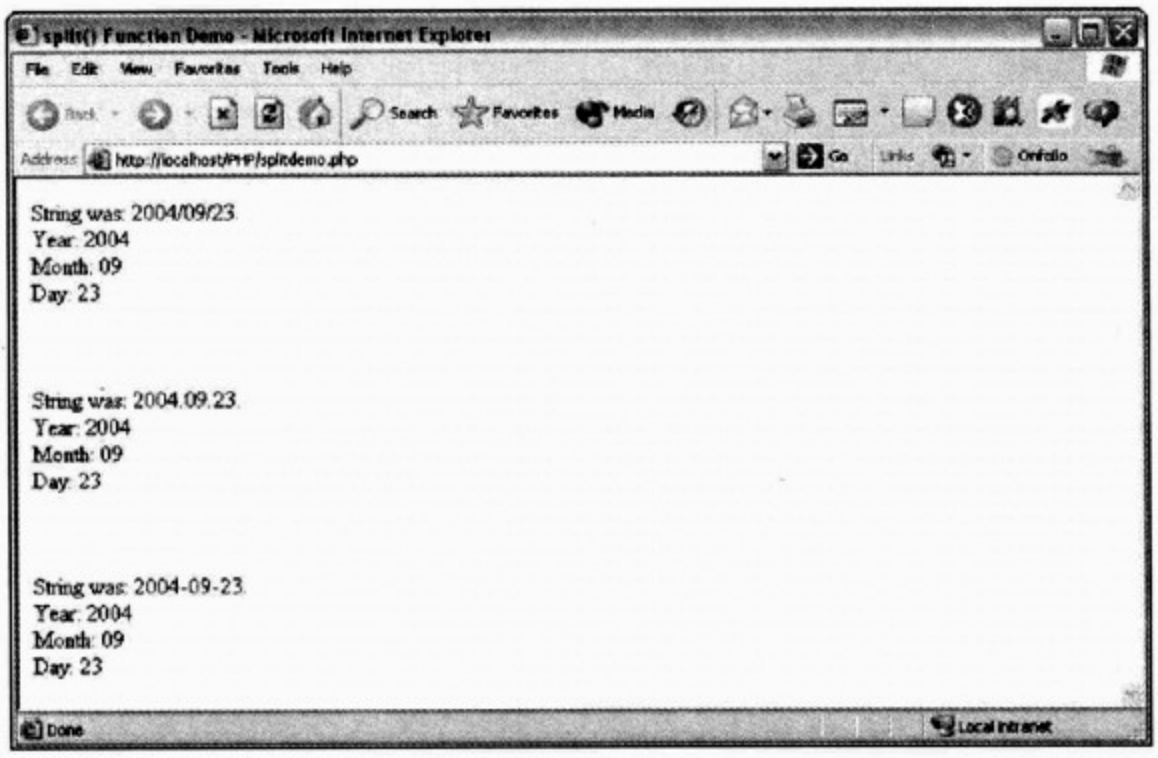


图 23-11

工作原理

本例示范了如何把三个表示日期值的字符串，按照不同的分隔符——正斜杠、句点字符和连字符进行拆分。

首先，声明三个变量，每个变量都保存一个 2004 年 9 月 23 日的日期值。但三个变量 $myString1、$myString2 和 $myString3 中使用的分隔符都不相同：

```
$myString1 = "2004/09/23";
$myString2 = "2004.09.23";
$myString3 = "2004-09-23";
```

然后，声明一个变量 $myPattern 并将一个表示字符类的字符串指定给它，该字符类中包含连字符、正斜杠以及句点字符。注意将连字符作为字符类中的第一个字符，这样它就不会指定一个范围了：

```
$myPattern = "[-/.]";
```

创建一个列表，使其包含 myString1 所表示的日期的各个部件。由于正斜杠会匹配两次，所以字符串 myString1 将被拆分为三个部分，分别保存在列表中的$year、$month 和$day 变量中：

```
list($year, $month, $day) = split($myPattern, $myString1);
```

输出变量 $year、$month 和 $day：

```
echo "<p>String was: $myString1. <br />Year: $year <br />Month: $month <br />Day:
$day</p>";
```

接下来创建包含变量 $myString2(使用句点字符作为分隔符)和 $myString3(使用连字符作为分隔符)中日期部件的列表，并输出相应部件的值：

```
list($year, $month, $day) = split($myPattern, $myString2);
echo "<br /><br /><p>String was: $myString2. <br />Year: $year <br />Month: $month
<br />Day: $day</p>";
list($year, $month, $day) = split($myPattern, $myString3);
echo "<br /><br /><p>String was: $myString3. <br />Year: $year <br />Month: $month
<br />Day: $day</p>";
```

4. spliti()函数

除了不区分大小写之外，spliti() 函数与 split() 函数的作用是相同的。

如果将一个字母字符作为分隔符，那么可以使用 spliti() 进行不区分大小写的匹配，使用 split() 进行区分大小写的匹配。但是，由于最常见的分隔符是逗号、连字符、正斜杠、分号以及句点字符，所以在实践中使用这两个函数往往没有什么区别。

spliti() 函数与 split() 函数类似，也可以接受两到三个参数。第一个参数是匹配拆分位置的正则表达式模式。第二个参数是测试字符串。而第三个参数(可选)是一个整数值，该整数值用于指定匹配和拆分的最大次数。在不指定第三个参数的情况下，匹配和拆分会在整个字符串范围内进行。

5. sql_regcase()函数

sql_regcase()函数与其他 ereg()函数族中的函数不同，它用于创建正则表达式模式，而不是使用正则表达式模式。

试一试：使用 sql_regcase() 函数

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>sql_regcase() Demo</title>
</head>
<body>
<?php
$sequenceToMatch = "Doctor";
$myPattern = sql_regcase($sequenceToMatch);
echo "<p>To match '$sequenceToMatch' the sql_regcase() function produces:
'$myPattern'.</p>";
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\sql_regcaseDemo.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/sql_regcaseDemo.php。按下回车键并观察网页中显示的结果。如图 23-12 所示，结果生成了一个在不区分大小写的方式下与指定字符串匹配的字符类序列(正则表达式模式)。

图 23-12

工作原理

对于$sequenceToMatch 变量中包含的每一个字符，该函数都为其创建一个包含大小写字符的字符类。比如，对于 doctor 中 D，会创建字符类[Dd]。

23.2.3 Perl 兼容正则表达式

Perl 兼容正则表达式(PCRE)提供了比 ereg()及其相关函数更流行也更强大的正则表达式支持。

下面的表 23-2 中总结了 PHP 中的 PCRE 所支持的函数族。其中每个函数都将在本章稍后加以详细介绍。

表 23-2 PCRE 支持的函数

函　数	说　明
preg_match()	在指定的测试字符串中尝试匹配指定的模式
preg_match_all()	在指定的测试字符串中尽可能多地尝试匹配指定的模式
preg_grep()	该函数用于在数组中查找一个正则表达式模式的匹配项
preg_quote()	该函数可用于将一个正则表达式模式中的每个字符都使用反斜杠转义
preg_replace()	在指定的测试字符串中尝试匹配一个正则表达式模式，并使用指定的替换字符串替换找到的所有匹配项
preg_replace_callback()	与 preg_replace() 类似，但替换字符串由一个回调函数定义
preg_split()	在与指定的正则表达式模式匹配的位置将测试字符串拆分为相应的子字符串数组

1. PCRE 中的模式定界符

PCRE 支持由开发人员指定的正则表达式模式定界符。默认的定界符是一对正斜杠字符。若要指定一个正则表达式，必须将模式包含在一对定界符中，而定界符则必须包含在一对双引号中。

可以使用任意非字母字符作为定界符，或者使用通常都成对出现的字符，比如{}、<>或()。

试一试：使用多种定界符

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>Simple preg_match() Regex Test</title>
</head>
<body>
<?php
if (preg_match("/Hel/", "Hello world!")) echo "<p>A match was found using paired
'/' as delimiter.</p>";
if (preg_match(".Hel.", "Hello world!")) echo "<p>A match was found using paired
'.' as delimiter.</p>";
if (preg_match("{Hel}", "Hello world!")) echo "<p>A match was found using matched
'{' and '}' as delimiters.</p>";
if (preg_match("(Hel)", "Hello world!")) echo "<p>A match was found using matched
'(' and ')' as delimiters.</p>";
if (preg_match("<Hel>", "Hello world!")) echo "<p>A match was found using matched
'<' and '>' as delimiters.</p>";
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\DelimiterTest.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/DelimiterTest.php。按下回车键，并观察如图 23-13 所示的结果。

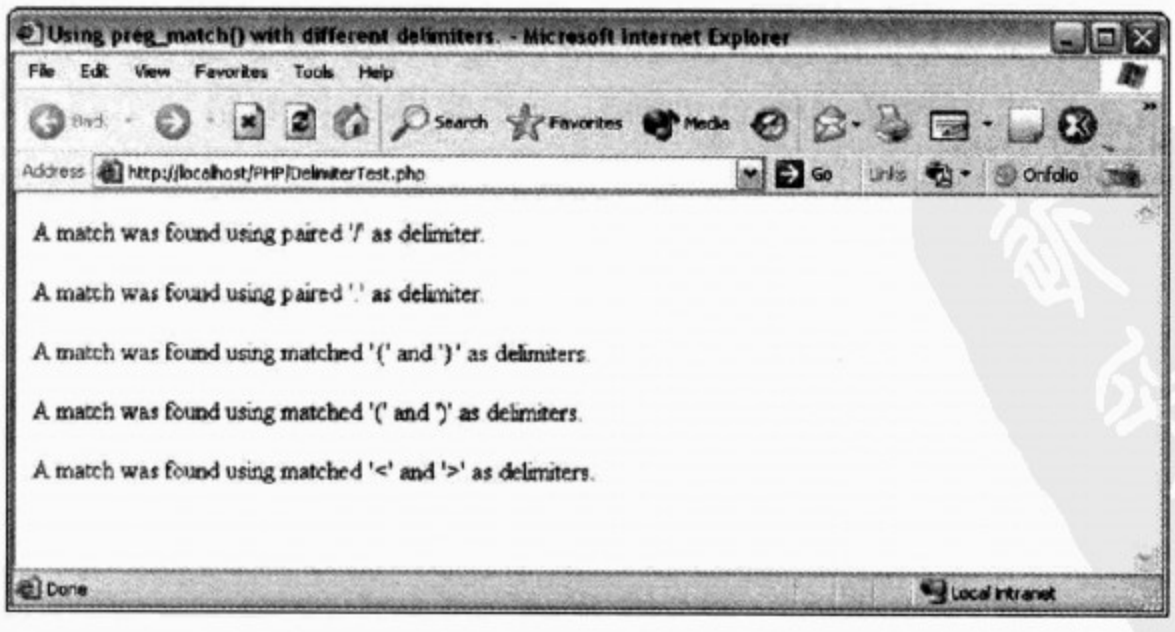


图　23-13

工作原理

本例中使用了五次 preg_match()函数。每一次都基于相同的字符串 Hello world!，测试其中是否包含与直接量模式 Hel 匹配的内容。而如果有(任何一次匹配中)，则会显示存在匹配项的信息以及使用了什么定界符。

第一个 if 语句中使用了一对正斜杠作为定界符。这是 Perl 中默认的定界符：

```
if (preg_match("/Hel/", "Hello world!")) echo "<p>A match was found using paired
'/' as delimiter.</p>";
```

不过，也可以使用其他成对的非字母字符作为定界符。第二个 if 语句就使用了一对句点字符作为模式的定界符：

```
if (preg_match(".Hel.", "Hello world!")) echo "<p>A match was found using paired
'.' as delimiter.</p>";
```

其他三个 if 语句中使用的都是通常会成对出现的字符。比如，左大括号 { 及对应的右大括号 }。最后两个模式中使用了常见的左右圆括号 (和) 以及左右尖括号 < 和 >。

```
if (preg_match("{Hel}", "Hello world!")) echo "<p>A match was found using matched
'{' and '}' as delimiters.</p>";
if (preg_match("(Hel)", "Hello world!")) echo "<p>A match was found using matched
'(' and ')' as delimiters.</p>";
if (preg_match("<Hel>", "Hello world!")) echo "<p>A match was found using matched
'<' and '>' as delimiters.</p>";
```

如果没有其他特殊的原因，最好使用正斜杠作为定界符。因为大多数开发人员对使用正斜杠作为定界符的正则表达式模式都非常熟悉，所以使用正斜杠至少不会引起其他人的误解。假如必须使用其他定界符，建议对你的选择添加相应的注释，以便将来维护代码的人不会对此产生歧义。

2. 转义模式定界符

当在 PHP 中使用诸如正斜杠之类的字符作为正则表达式模式的定界符时，如果希望匹配该字符的直接量，那么就必须对这个用做定界符的字符进行转义。例如，如果将正斜杠用做模式的定界符，而希望匹配包含 com、net、org、info 以及 biz 等顶级域名的 HTTP URL，就必须在模式中转义每一个正斜杠，才能将其作为第一个参数传递给函数 preg_match()。例如：

```
preg_match('/http:\/\/.*\.(com|net|org|info|biz)\/.*/', $testString)
```

3. PCRE 中的匹配修饰符

在基于 Perl 的 Perl 兼容正则表达式中，匹配修饰符可能会让人感到有些好奇。下面的表 23-3 中总结了 PHP 的 PCRE 中有效的匹配修饰符。

表 23-3　PCRE 中有效的匹配修饰符

匹配修饰符	说　明
i	使匹配不区分大小写
m	多行。改变 ^ 和 $ 元字符的作用。在多行匹配模式下，^ 匹配一行开始的位置，而 $ 匹配一行结束的位置
s	更改句点元字符的含义。在使用 s 修饰符的情况下，句点字符将匹配任何字符；如果不使用 s 修饰符，句点字符匹配除换行符之外的任何字符
x	更改模式中空白符的处理方式。在使用 x 修饰符的情况下，未转义的空白符会被忽略
A	强制匹配测试字符串的开始位置
D	如果设置了 D 修饰符，$ 元字符则只匹配测试字符串的结束位置。如果设置了 m 修饰符，D 修饰符会被忽略
S	影响到模式如何被处理。如果设置了 S 修饰符，则会引发如何以最佳方式处理正则表达式的深度分析
U	U 修饰符用于修改模式的“贪婪性”。设置 U 修饰符后，默认的行为将变成懒惰的。通过 (? ...) 可以使匹配变得贪婪
X	开启 Perl 兼容的行为
e	只应用于 preg_replace() 函数

匹配修饰符应放在第二个成对或匹配的定界符之后、第二个双引号字符之前。因此，要通过 preg_match() 函数以不区分大小写的方式基于测试字符串 Hello world! 来匹配模式 Hel，应该写成下面这样：

```
preg_match("/Hel/i", "Hello world!")
```

本章后面的例子中还会用到这里的一些修饰符。

4. 使用 preg_match()函数

preg_match()能够基于测试字符串匹配一个模式，与前面使用的 ereg()函数类似。不过，preg_match()通常比 ereg()的速度更快，因此当使用支持 PCRE 功能的 PHP 时，应该首选 preg_match()。

preg_match()函数可以接受两到三个参数。第一个参数是正则表达式模式。但是，与在使用 ereg()时写成下面这样不同：

```
ereg("Hel", "Hello world!");
```

要匹配字符串 Hello world! 中的字符序列 Hel，在使用 preg_match() 函数时必须将模式包含在一对定界符当中——比如一对正斜杠定界符，而且这对定界符也要包含在一对双引号当中。因此，在使用 preg_match()时，应该使用下面的代码来表示要匹配 Hello world! 中的字符序列 Hel：

```
preg_match("/Hel/", "Hello world!");
```

第二个参数是测试字符串。可选的第三个参数是一个变量，在存在匹配项的情况下，会返回给该变量一个包含匹配项的数组。

试一试：一个使用 preg_match() 的例子

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>Simple preg_match() Regex Test</title>
</head>
<body>
<?php
if (preg_match("/Hel/", "Hello world!")) echo "<p>A match was found.</p>"
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\SimplePregTest.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/SimplePregTest.php，按下回车键并观察如图 23-14 所示的结果。由于存在与模式 Hel 匹配的字符序列，所以会看到 A match was found.被显示出来。

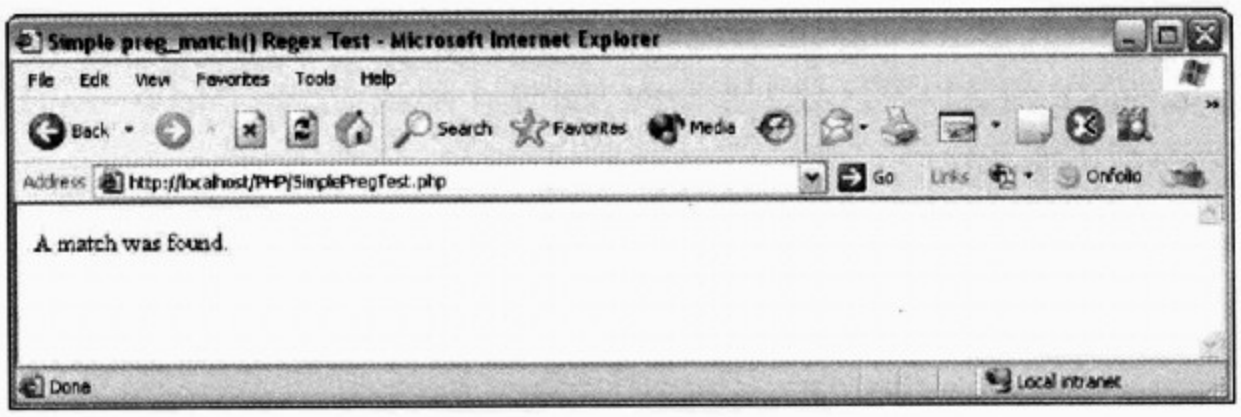


图 23-14

工作原理

在此，preg_match()函数用于生成一个可以被 if 语句测试的布尔值：

```
if (preg_match("/Hel/", "Hello world!")) echo "<p>A match was found.</p>"
```

注意，在传递给 preg_match()函数的第一个参数中，使用一对正斜杠作为定界符将正则表达式模式包含在其中。

下面这个例子使用带有三个参数的 preg_match()函数，并操纵由该函数第三个参数返回的数组元素。数组元素是从零开始编号的。而返回的数组元素[0]中包含的是整个匹配项。

试一试：在 preg_match() 函数中使用分组的模式

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>Using preg_match() with 3 arguments</title>
</head>
<body>
<?php
$dateToday = gmdate("Y m d");
$myDate = preg_match("/(\d{4})\s(\d{2}) (\d{2})/", $dateToday, $dateComponents);
if ($myDate)
{
echo "<p>The original date was: $dateToday</p>";
echo "<p>In MM/DD/YYYY format today is:
$dateComponents[2]/$dateComponents[3]/$dateComponents[1]</p>";
}
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\preg_match_3args.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/preg_match_3args.php，然后按下回车键并观察如图 23-15 所示的网页结果。在网页中，上面的一行中显示的是日期的原始格式，而下面一行中显示的则是 MM/DD/YYYY 式的日期格式。下面显示的新日期格式是通过使用 $dateComponents 数组创建的，该数组是 preg_match() 函数的第三个参数。

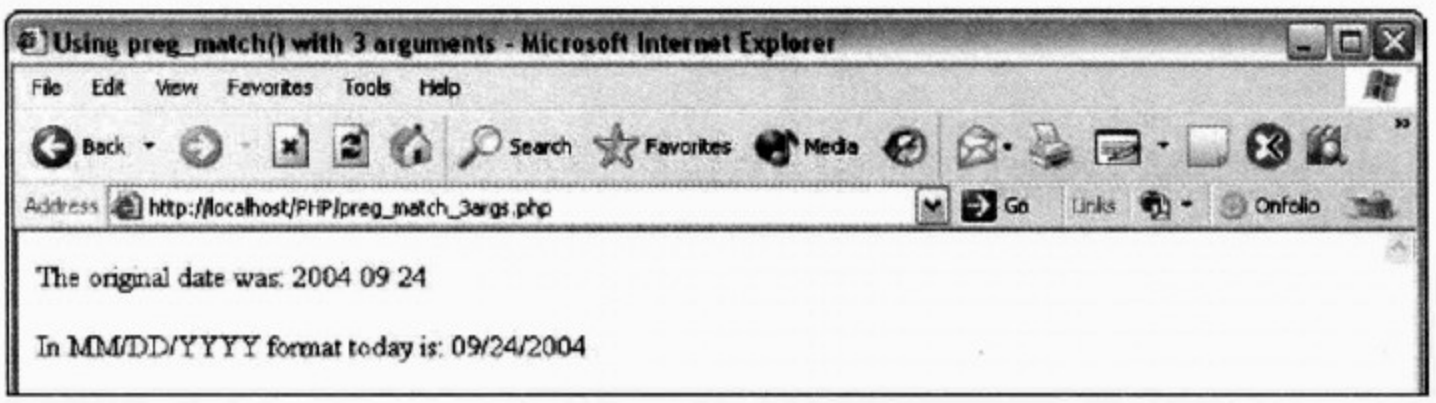


图　23-15

工作原理

将 gmdate()函数的返回值指定给变量 $dateToday。传递给 gmdate()函数的参数表示返回 YYYY MM DD 格式的日期值，每个值都是数字形式的：

```
$dateToday = gmdate("Y m d");
```

将 preg_match() 函数的返回值指定给变量 $myDate。由于在 $dateToday 的值中包含模式 (\d{4})\s(\d{2}) (\d{2}) 的匹配项，所以 $myDate 变量中包含的值等价于布尔值 True。注意在第二组和第三组圆括号之间有一个空格符。

本例中的 preg_match()函数带有三个参数，作为第一个参数的模式中包含着通过圆括号创建的分组。而第三个参数$dateComponents 作为数组变量，其数组元素中将包含与每个分组匹配的字符序列：

```
$myDate = preg_match("/(\d{4})\s(\d{2}) (\d{2})/", $dateToday, $dateComponents);
```

由于存在匹配项，所以$myDate 变量的值为 True，因而会执行包含在 if 语句中的代码：

```
if ($myDate)
```

首先，显示保存在变量 $dateToday 中日期的原始格式。然后，对$dateComponents 数组中的元素进行重新排列构成 MM/DD/YYYY 式的新日期格式：

```
{
echo "<p>The original date was: $dateToday</p>";
echo "<p>In MM/DD/YYYY format today is:
$dateComponents[2]/$dateComponents[3]/$dateComponents[1]</p>";
}
```

虽然 preg_match()函数能够回答是否存在匹配项的问题。但是，如果想匹配并操纵多个匹配项(如果测试字符串中存在多个匹配项)，就需要使用 preg_match_all()函数。

5. 使用 preg_match_all()函数

preg_match_all() 函数能够返回测试字符串中与指定模式匹配的所有匹配项。

试一试：使用 preg_match_all() 函数

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>Using preg_match_all()</title>
</head>
<body>
<?php
$testString = "A99 B888 C234 D123 E45678 f2345";
$myMatches = preg_match_all("/[A-Z]\d{1,5}/", $testString, $partNumbers);
if ($myMatches)
{
 for($counter=0; $counter < $myMatches; $counter++)
 {
  echo "<p>" . $partNumbers[0][$counter]. "</p>";
 }
}
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\preg_match_all.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/preg_match_all.php，然后按下回车键并观察网页中显示的结果，如图 23-16 所示。

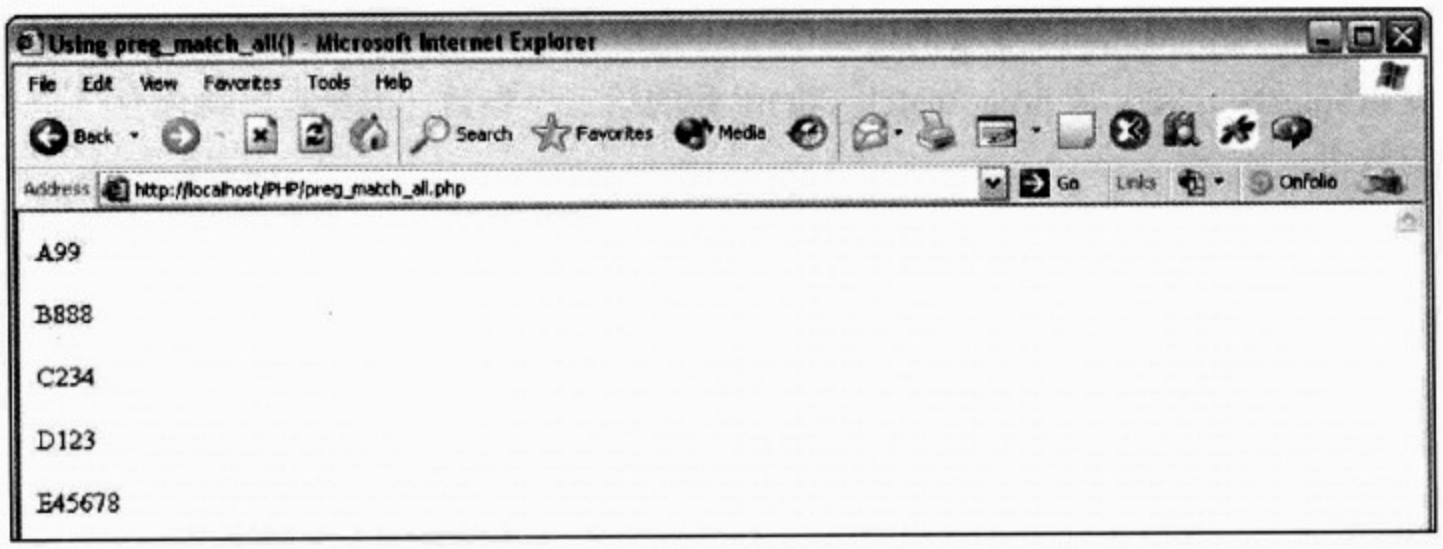


图　23-16

(4) 修改下列代码行：

```
$myMatches = preg_match_all("/[A-Z]\d{1,5}/", $testString, $partNumbers);
```

使其执行不区分大小写的匹配：

```
$myMatches = preg_match_all("/[A-Z]\d{1,5}/i", $testString, $partNumbers);
```

(5) 将修改后的代码保存为 C:\inetpub\wwwroot\PHP\preg_match_all_Insensitive.php。

(6) 在 Internet Explorer 中输入 URL http://localhost/PHP/preg_match_all_Insensitive.php，然后按回车键，观察网页中显示的结果，如图 23-17 所示。与上一次比较，多了一个匹配项——f2345。

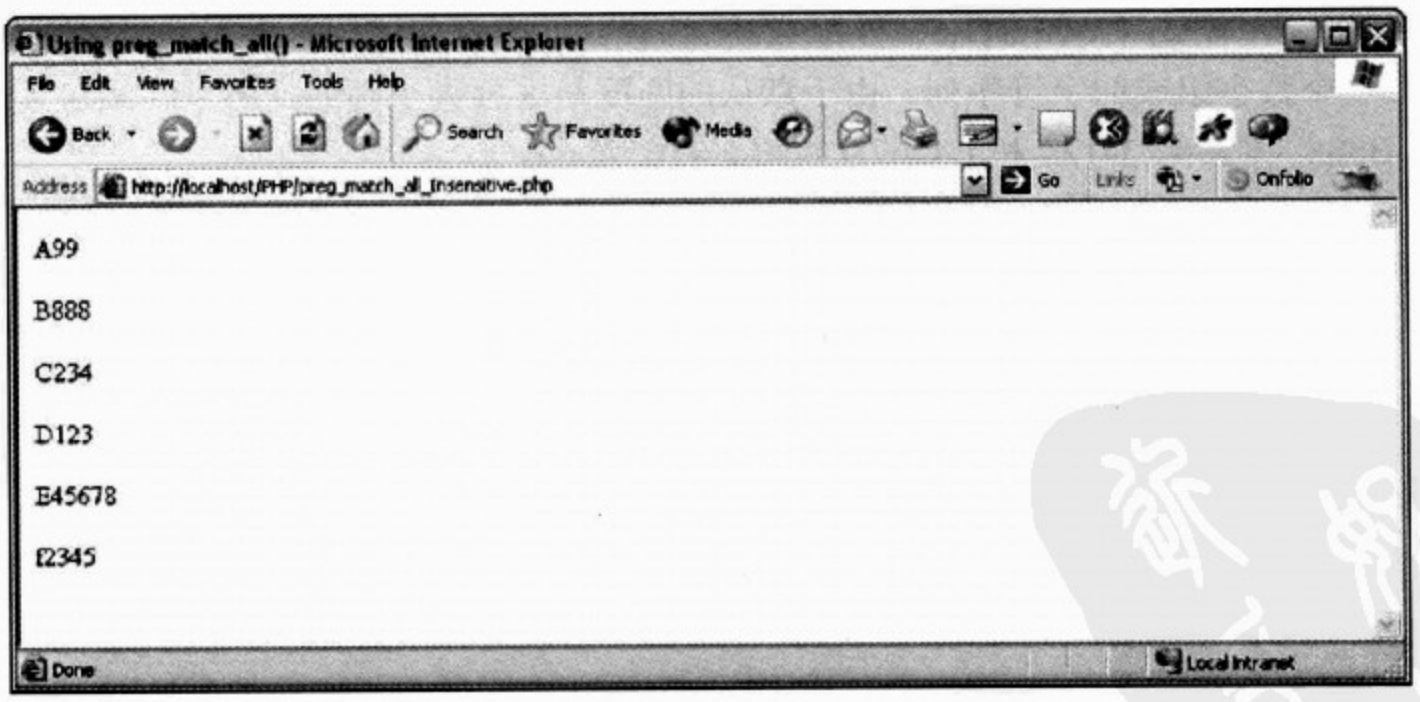


图　23-17

工作原理

将测试字符串指定给变量$testString：

```
$testString = "A99 B888 C234 D123 E45678 f2345";
```

将 preg_match_all()函数返回的 int 值指定给变量$myMatches。稍后，当循环遍历数组的值时会用到 $myMatches 变量的值。

变量$partNumbers 是 preg_match_all()函数的第三个参数，该变量中将保存一个包含匹配项的数组：

```
$myMatches = preg_match_all("/[A-Z]\d{1,5}/", $testString, $partNumbers);
```

为显示全部匹配项，使用了 if 语句以及嵌入其中的 for 循环。if 语句用于测试变量 $myMatches 是否不等于 0——换句话说，至少存在一个匹配项——如果是，则 if 语句的测试就会通过：

```
if ($myMatches)
{
```

for 循环从 0 开始计数，一直累加到等于$myMatches 变量的值为止：

```
for($counter=0; $counter < $myMatches; $counter++)
{
```

在 for 循环中，使用 echo 语句用于连接直接量文本和一个由$counter 及更深一层的直接量文本决定的数组元素的值。

我们希望返回整个匹配项，所以使用每个匹配项数组中的元素 0，而数组中的另一个维则由 $counter 变量的当前值决定：

```
echo "<p>" . $partNumbers[0][$counter]. "</p>";
}
```

在第 4 步，模式 [A-Z]\d{1,5} 匹配成功。该模式匹配任何以一个大写字母字符开头的、后跟 1~5 个数字组成的字符序列。由于默认的匹配是区分大小写的，所以匹配结果分别是 A99、B888、C234、D123 和 E45678(如图 23-16 所示)。

当把 preg_match_all() 函数的第一个参数修改为“/[A-Z]\d{1,5}/i”之后，匹配变成了不区分大小写的。所以模式 [A-Z]\d{1,5} 将匹配由一个大写或小写的字母字符开头的、后跟 1~5 个数字的字符序列。上一次匹配的字符序列仍然匹配，但除此之外还有一个匹配项——f2345，这个字符序列的第一个字符是小写字母。

6. 使用 preg_grep()函数

preg_grep()函数能够基于数组中的元素来匹配正则表达式模式。preg_grep() 函数接受两个参数。第一个参数是正则表达式模式，第二个参数是一个数组。

因此，若要将数组$myArray 中与模式$myPattern 匹配的值指定给变量$myMatches，可以使用下面的代码：

```
$myMatches = preg_grep($myPattern, $myArray);
```

试一试：使用 preg_grep()函数

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>A preg_grep() Test</title>
</head>
<body>
<?php
$myArray = array("Hello", "Help", "helper", "shell", "satchel", "Camera");
$myMatchesSensitive = preg_grep("/Hel/", $myArray);
echo "<p>Matching case sensitively:</p>";
if ($myMatchesSensitive)
{
print_r (array_values($myMatchesSensitive));
}
$myMatchesInsensitive = preg_grep("/Hel/i", $myArray);
echo "<br /><p>Matching case insensitively:</p>";
if ($myMatchesInsensitive)
{
print_r (array_values($myMatchesInsensitive));
}
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\preg_grep.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/preg_grep.php，然后按下回车键，观察网页中显示的结果，如图 23-18 所示。比较一下区分大小写匹配和不区分大小写的匹配有何不同。

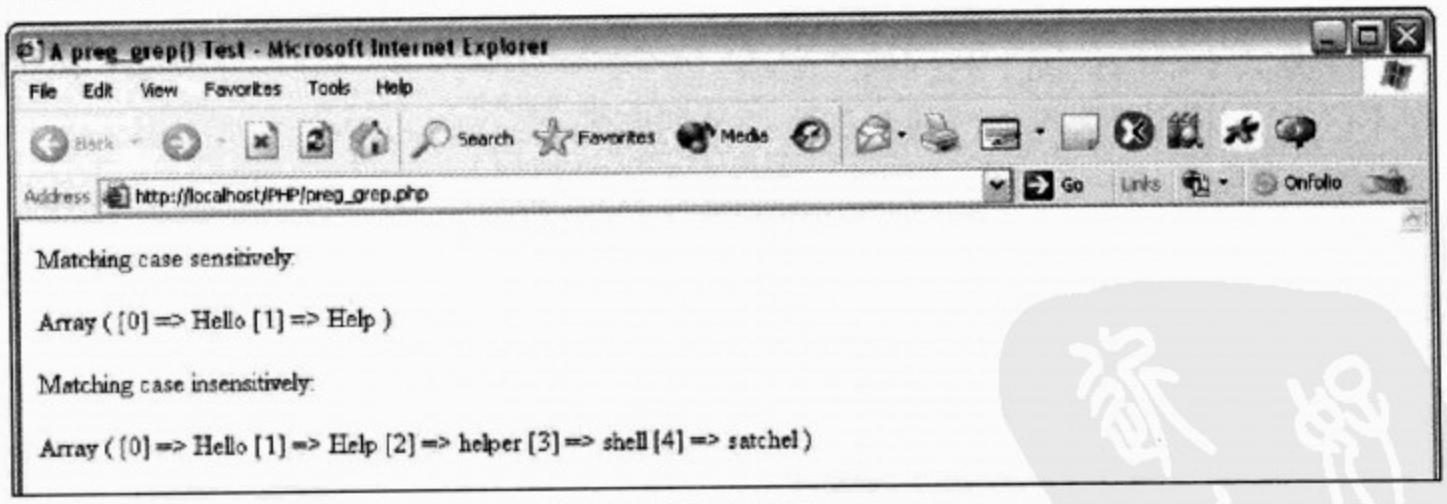


图　23-18

工作原理

本例使用了两次 preg_grep()函数：第一次区分大小写(默认)，第二次不区分大小写。要匹配的数组是$myArray，它包含的数组元素如下所示：

```
$myArray = array("Hello", "Help", "helper", "shell", "satchel", "Camera");
```

首先，进行区分大小写的匹配。将直接量模式“/Hel/”作为 preg_grep()函数的第一个参数，将 preg_grep()函数返回的数组指定给变量 $myMatchesSensitive：

```
$myMatchesSensitive = preg_grep("/Hel/", $myArray);
```

显示以区分大小写方式进行匹配的信息：

```
echo "<p>Matching case sensitively:</p>";
```

然后，将变量$myMatchesSensitive 的值用于 if 语句测试。如果变量 $myMatches Sensitive 中包含一个非空数组，则执行 if 语句中的代码：

```
if ($myMatchesSensitive)
```

使用 print_r()函数显示$myArray 中数组元素的键和值。print_r()函数用于输出人类可读的函数信息。

在区分大小写的情况下，只有两个元素包含匹配项，Hello 和 Help：

```
{
print_r (array_values($myMatchesSensitive));
}
```

在进行不区分大小写的匹配时，仍然重复以上过程。将 preg_grep()函数返回的数组指定给变量$myMatchesInsensitive。注意，该函数的第一个参数中使用了 i 修饰符以指定不区分大小写的匹配：

```
$myMatchesInsensitive = preg_grep("/Hel/i", $myArray);
```

显示以不区分大小写的方式进行匹配的信息：

```
echo "<br /><p>Matching case insensitively:</p>";
```

然后，if 语句测试 $myMatchesInsensitive 中是否包含空数组。如果不是，使用 print_r()输出在不区分大小写的情况下找到的匹配项。

在不区分大小写的情况下，找到三个额外的匹配项：helper、shell 和 satchel。

```
if ($myMatchesInsensitive)
{
print_r (array_values($myMatchesInsensitive));
}
```

7. 使用 preg_quote()函数

preg_quote() 用于转义运行时生成的字符串中需要转义的字符。例如，可以使用 preg_quote() 匹配一个句点字符。在输出字符串中，句点会被转义成 \.。

8. 使用 preg_replace()函数

preg_replace()函数会在测试字符串中尽可能多地匹配一个正则表达式，然后将每一个匹配的子字符串替换成指定的替换文本。

preg_replace()函数可以接受三个或四个参数。第一个参数是一个正则表达式模式。第二个参数是替换文本。第三个参数是测试字符串。而可选的第四个参数是一个用于指定最多替换次数的 int 值。

通过使用 e 匹配修饰符可以使替换文本中的反向引用被解释为 PHP 代码，并使用相应的结果替换测试字符串。

试一试：使用 preg_replace() 函数

下面的例子示范了如何使用 preg_replace()函数将直接量文本 Star 替换成直接量文本 Moon。

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>A preg_replace() Demo</title>
</head>
<body>
<?php
$myString = "Star Training Company.";
$newString = preg_replace("/Star/", "Moon", $myString);
echo "<p>The original string was: '$myString'.</p>";
echo "<p>After replacement the string is: '$newString'.</p.>";
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\replaceDemo.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/replaceDemo.php，并观察结果，如图 23-19 所示。

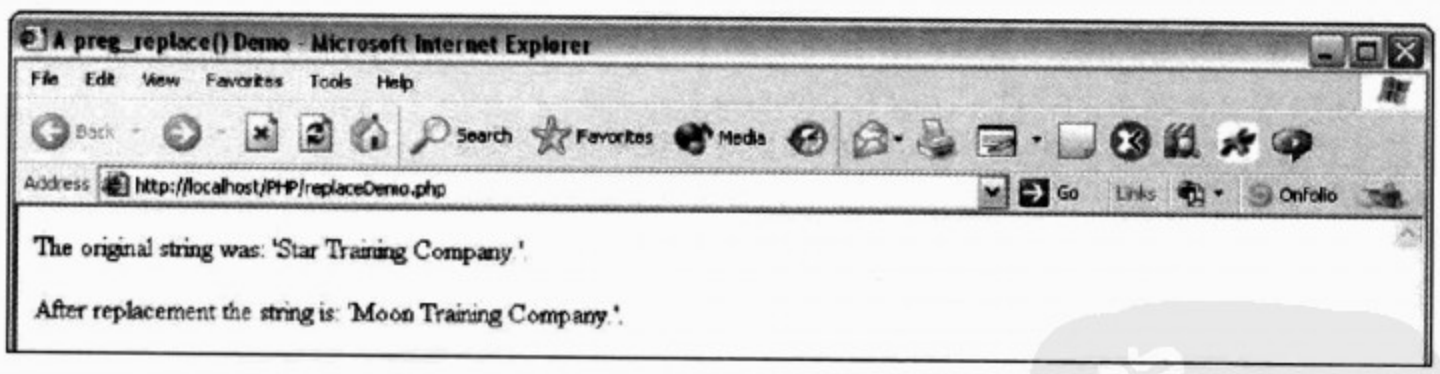


图　23-19

工作原理

将原始字符串 Star Training Company 指定给变量$myString：

```
$myString = "Star Training Company.";
```

将 preg_replace()函数返回的字符串值指定给变量$newString。传递给函数 preg_replace()的模式是一个直接量模式 Star，替换文本是 Moon，而替换发生在通过变量 $myString 指定的字符串中。本质上来讲，任何 Star 的实例都会被 Moon 所替换：

```
$newString = preg_replace("/Star/", "Moon", $myString);
```

将原始字符串及替换后的字符串显示给用户：

```
echo "<p>The original string was: '$myString'.</p>";
echo "<p>After replacement the string is: '$newString'.</p.>";
```

9. 使用 preg_replace_callback()函数

preg_replace_callback()函数与 preg_replace()函数本质上是相同的——只不过 preg_replace_callback()函数的第二个参数是一个回调函数。这个回调函数可以是用户定义的任何函数。

10. 使用 preg_split()函数

preg_split()函数能够根据一个正则表达式模式来拆分测试字符串。该函数返回一个由拆分后的子字符串组成的数组。preg_split() 函数接受两个强制性的参数和两个可选的参数。两个强制性的参数是正则表达式模式和测试字符串。可选的第三个参数是一个指定最大拆分次数的整数值，可选的第四个参数是表示所设置的标志的 int 值。

试一试：使用 preg_split() 函数

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>A preg_split() Example</title>
</head>
<body>
<?php
$myCSV = "Oranges, Apples, Bananas, Kiwi Fruit, Mangos";
$myArray = preg_split("/,/", $myCSV);
echo "<p>The original string was: '$myCSV'.</p>";
echo "<p>After splitting the array contains the following values:</p.><br />";
print_r(array_values($myArray));
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\preg_split.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/preg_split.php，观察网页中显示的结果，如图 23-20 所示。

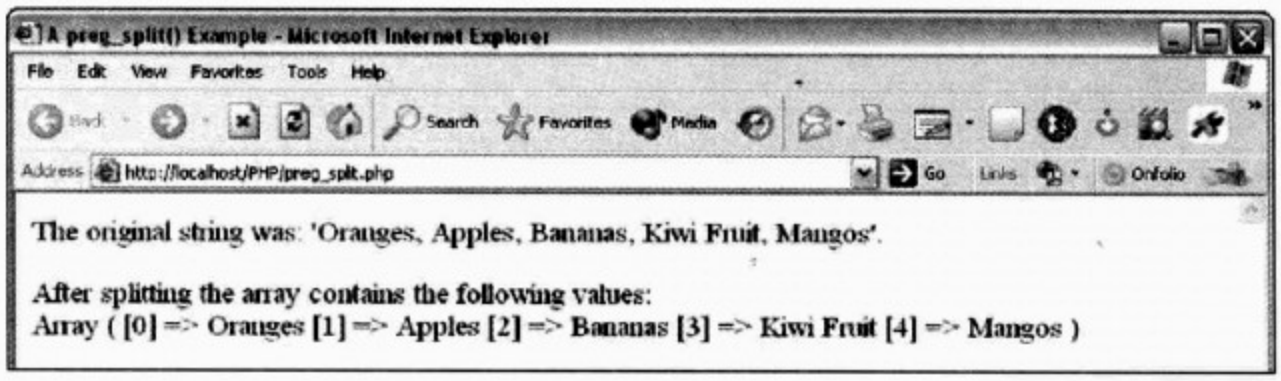


图　23-20

工作原理

本例将包含在一个字符串中以逗号分隔的值的列表拆分为一个数组。

首先，将测试字符串指定给变量 $myCSV：

```
$myCSV = "Oranges, Apples, Bananas, Kiwi Fruit, Mangos";
```

然后，将 preg_split() 函数返回的数组指定给变量 $myArray。preg_split() 函数的第一个参数是包含一个逗号的正则表达式模式，也就是说将使用逗号来拆分测试字符串 $myCSV。因为没有为 preg_split() 函数指定第三个参数，所以拆分会发生在每一个逗号所在的位置：

```
$myArray = preg_split("/,/", $myCSV);
```

使用两个 echo 语句向用户显示原始字符串和相关信息。而 print_r()函数与 array_values()函数连用是为了显示$myArray 变量的值：

```
echo "<p>The original string was: '$myCSV'.</p>";
echo "<p>After splitting the array contains the following values:</p.><br />";
print_r(array_values($myArray));
```

23.3　PHP 支持的元字符

下面将按照 ereg()和 preg()函数族分别介绍 PHP 支持的元字符。这样，如果只想使用某一族的函数，就很容易找到该族函数所支持的功能。

23.3.1　ereg() 函数族支持的元字符

下面的表 23-4 中总结了在 PHP 中使用 ereg() 函数族时支持的元字符。

表 23-4　PHP 中使用 ereg()函数族时支持的元字符

元　字　符	说　　明
\d	不支持。需要使用字符类 [0-9] 代替
\D	不支持。需要使用相反的字符类 [^0-9] 代替
\w	不支持。需要使用字符类 [A-Za-z0-9_] 或 POSIX 字符类 [:alnum:] 代替 (POSIX 字符类 [:alnum:] 不包含下划线。译者注)

(续表)

元 字 符	说 明
\W	不支持。需要使用取反的字符类 [^A-Za-z0-9_] 代替
?	限定符。匹配前面字符或组的零个或一个实例
*	限定符。匹配前面字符或组的零个或多个实例
+	限定符。匹配前面字符或组的一个或多个实例
{n,m}	限定符。匹配前面字符或组的至少 n 个、最多 m 个实例
.(句点字符)	匹配除换行符之外的任何字符
^	匹配字符串的开始位置
$	匹配字符串的结束位置
[...]	字符类。匹配包含在方括号内的任何字符的单个实例
[^...]	取反的字符类。匹配不包含在方括号内的任何字符的单个实例

23.3.2 在 PHP 中使用 POSIX 字符类

ereg() 函数族的功能是基于 POSIX 的。因此，可以在 ereg()函数及相关函数中使用 POSIX 字符类，如[:alnum:]。

下面的表 23-5 中总结了最常用的 POSIX 字符类。

表 23-5 常用的 POSIX 字符类

字 符 类	说 明
[:alnum:]	匹配字母或数字字符
[:alpha:]	匹配字母字符
[:space:]	匹配空白符
[:blank:]	匹配空格符或制表符
[:digit:]	匹配数字
[:lower:]	匹配小写的字母字符
[:upper:]	匹配大写的字母字符

试一试：在 ereg() 函数中使用 [:alnum:] 字符类

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>ereg() [:alnum:] Test</title>
</head>
<body>
<?php
```

```
$match = ereg('[[:alnum:]]+', "Hello world!", $matches);
if ($match)
{
echo "<p>A match was found.</p>";
echo "<p>$matches[0]</p>";
}
?>
</body>
</html>
```

(2) 将文件保存为 EregAlnumTest.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/EregAlnumTest.php，并观察页面中显示的结果，如图 23-21 所示。

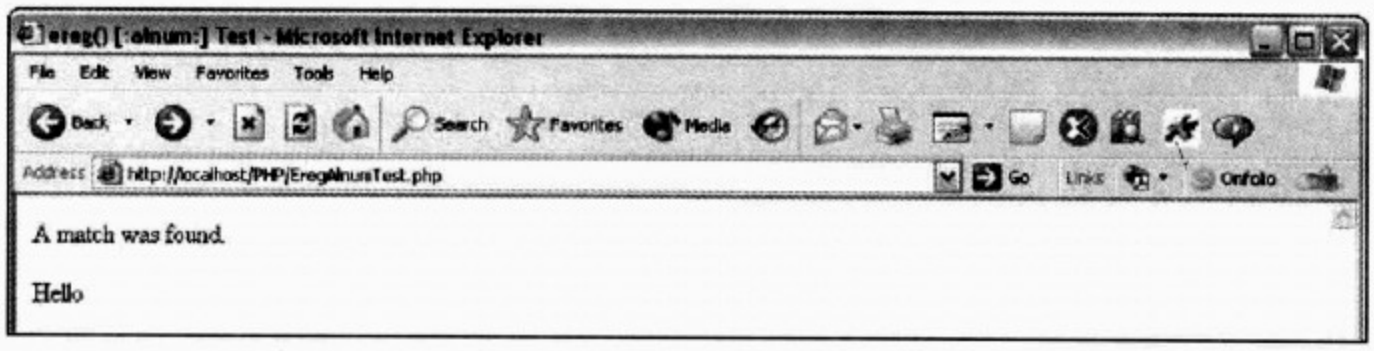


图　23-21

工作原理

POSIX 字符类 [:alnum:] 匹配字母字符和数字。与 \w 元字符不同，该字符类不匹配下划线。

ereg() 函数的第一个参数匹配一个或多个字母数字字符。注意，在 PHP 中 [:alnum:] 字符类要包含在另一对方括号中。

匹配是基于测试字符串 Hello world! 进行的。ereg() 函数带有三个参数，因此第一个匹配项(如果有)中分组捕获的值会返回到数组变量 $matches 中：

```
$match = ereg('[[:alnum:]]+', "Hello world!", $matches);
```

if 语句通过测试$match 变量值来判断是否存在一个匹配项：

```
if ($match)
```

在本例中，会通知用户存在一个匹配项：

```
{
echo "<p>A match was found.</p>";
```

匹配的文本包含在 $matches 数组的第 0 个元素中。再将匹配的字符序列显示给用户：

```
echo "<p>$matches[0]</p>";
}
```

下面的例子将使用 POSIX 的 [:space:]字符类，该字符类用于匹配空白符。

试一试：使用 [:space:] 字符类

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>ereg() [:space:] Test</title>
</head>
<body>
<?php
$match = ereg('o[[:space:]]+w', "Hello world!", $matches);
if ($match)
{
 echo "<p>A match was found.</p>";
 echo "<p>$matches[0]</p>";
}
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\EregSpaceTest.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/EregSpaceTest.php，并观察网页中显示的结果，如图 23-22 所示。

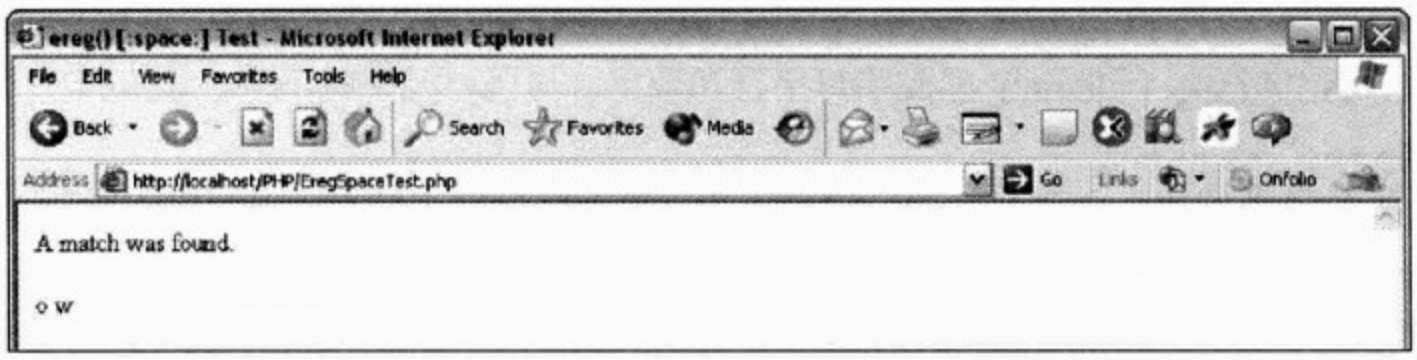


图 23-22

工作原理

ereg()函数的第一个参数用于匹配直接量 o 后跟一个或多个空白符，再后跟一个直接量 w。在测试字符串 Hello world! 中，匹配的文本是 Hello 最后的 o 后跟一个空格符，后跟 world 的首字母 w:

```
$match = ereg('o[[:space:]]+w', "Hello world!", $matches);
```

通知用户存在一个匹配项，同时将 $matches 数组第 0 个元素的值显示给用户：

```
echo "<p>A match was found.</p>";
echo "<p>$matches[0]</p>";
```

23.3.3 PCRE 支持的元字符

下面的表 23-6 中总结了 PCRE 支持的元字符。通过比较表 23-6 与 23-4，会发现 PCRE 支持的元字符更多一些。

表 23-6　PCRE 支持的元字符

元 字 符	说　明
^	匹配字符串(或多行模式中的行)的开始位置
$	匹配字符串(或多行模式中的行)的结束位置
.(句点字符)	匹配除换行符之外的任何字符(默认)
[...]	字符类。匹配方括号中包含的任意字符一次
[^...]	取反的字符类。匹配不包含在方括号中的任意字符一次
|	交替选择
()	以一对圆括号将与模式匹配的字符序列进行分组
?	限定符。匹配前面字符或组的零个或一个实例
*	限定符。匹配前面字符或组的零个或多个实例
+	限定符。匹配前面字符或组的一个或多个实例
{n,m}	限定符。匹配前面字符或组的至少 n 个、最多 m 个实例
\d	匹配一个数字
\D	匹配除数字之外的任何一个字符
\s	匹配一个空白符
\S	匹配除空白符之外的任何一个字符
\w	匹配任何 Perl 认可的"单词"字符。相当于字符类 [A-Za-z0-9_]
\W	匹配任何 Perl 认可的"单词"字符之外的字符。相当于取反字符类[^A-Za-z0-9_]
\b	匹配位于 \w 字符和 \W 字符之间的位置
\B	匹配不位于 \w 字符和 \W 字符之间的位置
\A	匹配一个字符串的开始位置。其行为不受多行模式的影响
\z	匹配一个字符串的结束位置。其行为不受多行模式的影响

23.3.4　位置元字符

^ 和 $ 元字符可以分别用于匹配一个测试字符串的开始位置和结束位置。在使用 m 匹配修饰符的情况下，^ 和 $ 元字符则分别匹配一行的开始位置和结束位置。

23.3.5　PHP 中的字符类

PHP 中的 PCRE 功能中包含了对字符类的完整支持——也包括范围及取反的字符类。

下面的例子示范了如何通过取反的字符类来测试一个字符串中是否只包含希望的字符类型。

试一试：使用取反的字符类

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>Negated Character Class Example</title>
</head>
<body>
<?php
$sequenceToMatch1 = "12345";
$sequenceToMatch2 = "123 45";
$negCharClass = "/[^0-9]/";
$nonNumMatch = preg_match($negCharClass, $sequenceToMatch1);
if ($nonNumMatch)
{
echo "<p>There was a non-numeric character in $sequenceToMatch1.</p>";
}
else
{
echo "<p>All characters were numeric in $sequenceToMatch1.</p>";
}
$nonNumMatch = preg_match($negCharClass, $sequenceToMatch2);
if ($nonNumMatch)
{
echo "<p>A non-numeric character was found in $sequenceToMatch2.</p>";
}
else
{
echo "<p>All characters were numeric in $sequenceToMatch2.</p>";
}
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\NegatedCharacterClass.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/NegatedCharacterClass.php，并观察网页中显示的结果，如图 23-23 所示。

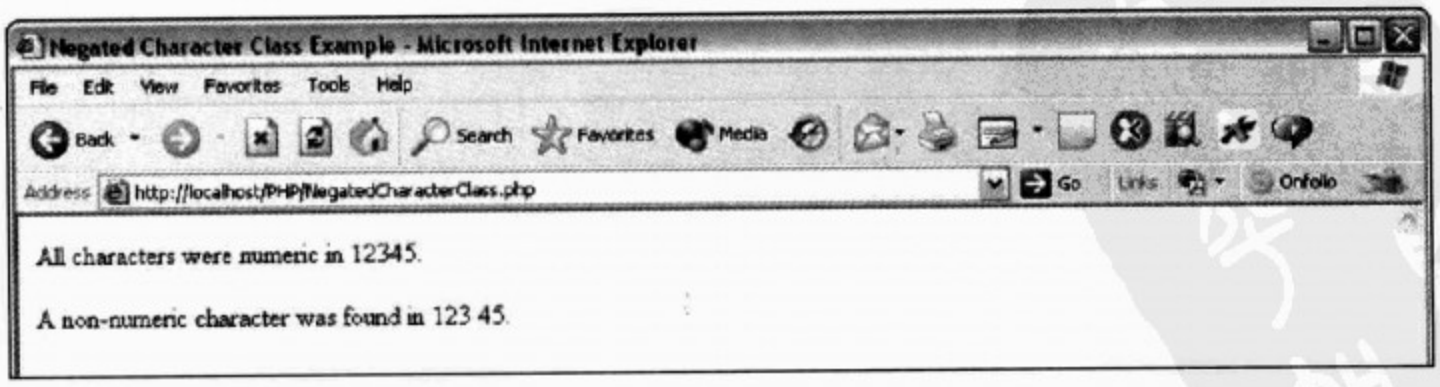


图 23-23

工作原理

本例通过测试两个字符串来查看其中是否包含不想要的字符类型——即一个非数字字符。

首先，将两个测试字符串分别指定给变量$sequenceToMatch1 和$sequenceToMatch2。如你所见，$sequenceToMatch2 中包含一个空格符——一个非数字字符。而变量$sequenceToMatch1 中则只包含数字：

```
$sequenceToMatch1 = "12345";
$sequenceToMatch2 = "123 45";
```

指定给变量 $negCharClass 的正则表达式模式由一个取反的字符类 [^0-9] 构成，该字符类匹配任何非数字字符：

```
$negCharClass = "/[^0-9]/";
```

首先使用 preg_match()函数来测试变量 $sequenceToMatch1 中是否包含一个非数字字符。因为没有找到非数字字符，所以会执行 else 语句中的代码：

```
$nonNumMatch = preg_match($negCharClass, $sequenceToMatch1);
if ($nonNumMatch)
{
echo "<p>There was a non-numeric character in $sequenceToMatch1.</p>";
}
else
{
echo "<p>All characters were numeric in $sequenceToMatch1.</p>";
}
```

接着，再使用 preg_match()函数测试变量 $sequenceToMatch2。而该字符串中包含一个非数字字符，也就是说 $nonNumMatch 中存在一个匹配项：

```
$nonNumMatch = preg_match($negCharClass, $sequenceToMatch2);
```

因此，if 语句的测试结果返回 True。于是，用户会收到在$sequenceToMatch2 中存在一个非数字字符的通知，同时该变量的值也会显示给用户：

```
if ($nonNumMatch)
{
echo "<p>A non-numeric character was found in $sequenceToMatch2.</p>";
}
else
{
echo "<p>All characters were numeric in $sequenceToMatch2.</p>";
}
```

23.3.6　为 PHP 中的正则表达式添加说明

可以使用 x 匹配修饰符来让匹配引擎忽略(作为函数——例如 preg_match()函数——第一个参数的)正则表达式模式中的空白符。

试一试：通过使用 x 匹配修饰符添加说明

(1) 在文本编辑器中输入下列代码：

```
<html>
<head>
<title>The x matching modifier in use.</title>
</head>
<body>
<?php
$US_SSN = "123-12-1234";
$myMatch = preg_match("/
\d{3} # Matches three numeric digits
- # Matches a literal hyphen
\d{2} # Matches two numeric digits
- # Matches a literal hyphen
\d{4} # Matches four numeric digits
/x", "123-12-1234", $theMatch);
echo "<p>The test string was: '$US_SSN'.</p>";
echo "<p>This matches the pattern /\d{3}-\d{2}-\d{4}/</p>";
?>
</body>
</html>
```

(2) 将代码保存为 C:\inetpub\wwwroot\PHP\XModifier.php。

(3) 在 Internet Explorer 中输入 URL http://localhost/PHP/XModifier.php，并观察网页中显示的结果，如图 23-24 所示。

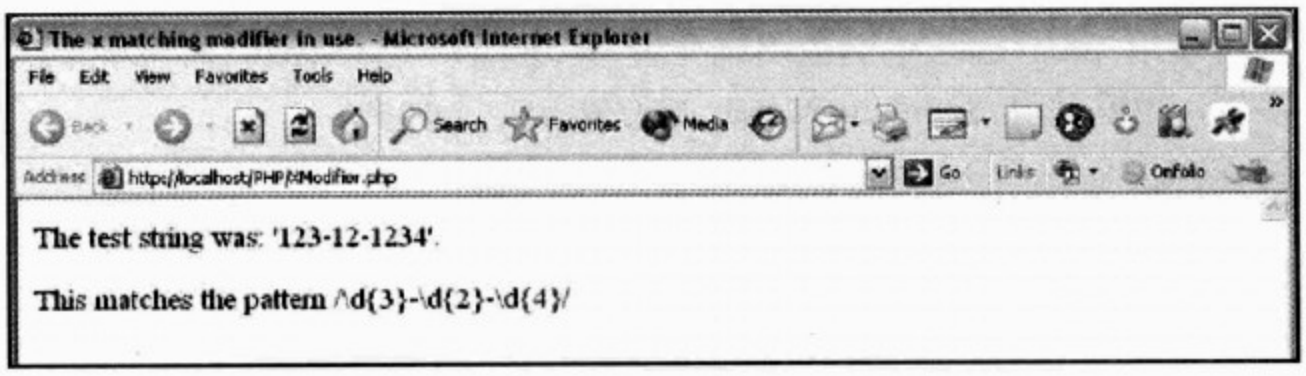


图 23-24

工作原理

首先，将一个可以用做美国社会保险号(SSN)的值指定给变量$US_SSN：

```
$US_SSN = "123-12-1234";
```

使用 preg_match() 函数来确定是否存在与模式 \d{3}-\d{2}-\d{4} 匹配的字符序列。但是，由于使用了 x 匹配修饰符，可以将模式分开在几行中来写，然后在每行中以 # 字符打头输入与模式相应的单个组件有关的含义说明：

```
$myMatch = preg_match("/
\d{3} # Matches three numeric digits
```

```
- # Matches a literal hyphen
\d{2} # Matches two numeric digits
- # Matches a literal hyphen
\d{4} # Matches four numeric digits
```

x 修饰符要放在第二个正斜杠——即正则表达式模式最后的限定符——的后面。同时将匹配的值返回给$theMatch 数组：

```
/x", "123-12-1234", $theMatch);
```

然后，通知用户这次匹配的结果：

```
echo "<p>The test string was: '$US_SSN'.</p>";
echo "<p>This matches the pattern /\d{3}-\d{2}-\d{4}/</p>";
```

23.4　练习

1. 在任何版本的 PHP 中都可以使用 Perl 兼容的正则表达式吗？
2. 哪个匹配修饰符可以允许在模式中使用多行注释？

第24章

W3C XML Schema 中的正则表达式

如今，以可扩展标记语言(eXtensible Markup Language，XML)来存储和传输的数据量与日俱增。因此，确定数据是否有效非常必要。另外，当 XML 格式的数据结构中保存有诸如信用卡号、社会保险号或邮政编码之类的数据时，可以通过正则表达式来检验 XML 数据结构中的内容是否合乎要求。W3C XML Schema 为控制 XML 文档内容的构成提供了一些有用的方法。

在本章中将学习以下内容：

- W3C XML Schema 的基本原理以及 W3C XML Schema 文档与 XML 实例文档如何关联
- 如何通过正则表达式和其他方法来表示对 XML 实例文档内容的约束
- Unicode 编码如何影响 W3C XML Schema 的使用
- W3C XML Schema 支持哪些元字符

在本章中，术语 W3C XML Schema 指的是万维网联盟(World Wide Web Consortium，W3C)指定的 XML 模式(schema)定义语言。

有关 W3C XML Schema 规范的细节和入门性的介绍可以在 www.w3.org/TR/xmlschema-0、www.w3.org/TR/xmlschema-1 和 www.w3.org/TR/xmlschema-2 中找到。除了基本的概念外，如果还想参考与 W3C XML Schema 有关的书籍，可翻阅由 R.Allen Wyke 和 Andrew Watt 合著的 *XML Schema Essential*。

24.1 W3C XML Schema 基础

当 1998 年早期发布 XML 1.0 版时，就已经存在一种与之相关的模式(schema)语言了。这个针对 XML 1.0 的模式就是文档类型定义(Document Type Definition，DTD)。但 DTD 本身具有某些局限性，表现在不仅能够指定的 XML 数据类型有限，而且也缺乏深入约束

XML 内容的功能。

模式，在 XML 文档中，是指用于规定一类 XML 文档的可用结构和内容的文档(也就是所说的文档的文档。译者注)。

W3C XML Schema 可以通过两种基本的方式来约束值。一种是约束“值空间(value space)”，另一种是约束“词法空间(lexical space)”。为区分这两个概念，使用 100 这个数值做例子。这个值与 100.0、100.00 和 100.000 等都是相同的。这说明在值空间中，它有一个值；而在词法空间中，它却有三个(只是这个例子列出的，实际应该更多)值。W3C XML Schema 中的正则表达式所操作的是词法空间，而不是值空间。

24.1.1　使用 W3C XML Schema 的工具

本章将展示通过 XML 编辑器来创建 XML 实例文档以及相应的 W3C XML Schema 文档。在基于模式验证 XML 实例文档的过程中，就会看到 W3C XML Schema 对正则表达式的支持。

虽然通过简单的文本编辑器也可以创建 XML 文档及相应的 W3C XML Schema 文档，但使用专门的 XML 编辑器可以提供下列一部分或全部功能：语法着色、格式检查、XML 实例文档验证、XML 实例文档与模式文档关联，以及根据 XML 实例文档创建 W3C XML Schema 文档等。

本章所使用的 XML 文档及相关的 W3C XML Schema 文档都是使用 XMLSpy、XMLWriter 和 Stylus Studio 创建的。其他 XML 编辑器也具有类似的功能，比如根据一个实例文档创建 W3C XML Schema 文档(或者从头创建一份模式文档)、测试一个 XML 实例文档是否符合某个模式(DTD 或 W3C XML Schema 文档)等。

使用 XMLSpy 或 Stylus Studio 时，可以创建一个 XML 实例文档，然后再根据这个 XML 实例文档创建一个 W3C XML Schema 文档。当然，根据 XML 实例文档的典型性，可能需要编辑一下生成的 W3C XML Schema 文档。

XMLSpy、XMLWriter 和 Stylus Studio 测试版的下载地址分别是：www.xmlspy.com/download.html 、 www.xmlwriter.com/download/download.shtml 和 www.stylusstudio.com/xml_download.html。

24.1.2　XML Schema 和 DTD 的比较

对于下面所示的这个简单的 XML 文档——PersonDataForDTD.xml，带有 DOCTYPE 声明的那一行中显示与这个 XML 实例文档关联的 DTD 的位置：

```
<!DOCTYPE PersonData SYSTEM "C:\BRegExp\Ch24\PersonData.dtd">
<PersonData>
   <Person>
      <LastName>Smith</LastName>
      <FirstName>John</FirstName>
   </Person>
</PersonData>
```

如果对 DOCTYPE 声明的语法不熟悉，但是有像 XMLSpy 或 Stylus Studio 这样的工具，那么可以使用这些软件来创建 DTD 并将 XML 实例文档与 DTD 关联起来。

PersonDataForDTD.xml 中的第一行引用了位于 C:\BRegExp\Ch24\PersonData.dtd 的一个 DTD。如果下载的文件被放到不同的位置，那么就需要修改这一行代码才能在 XMLSpy 或类似的 XML 编辑器中验证 XML。

上面 XML 实例文档所引用的 DTD——PersonData.dtd 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
<!ELEMENT Person (LastName, FirstName)>
<!ELEMENT PersonData (Person)>
```

其中，PersonData 元素出现在最后一行，它包含 Person 元素。然后，在倒数第二行中，Person 元素包含 LastName 和 FirstName 元素。在第二行和第三行中，FirstName 和 LastName 包含 PCDATA(可解析的字符数据)。从本质上讲，其含义就是说 FirstName 和 LastName 元素包含的是将被 XML 解析程序解析的 Unicode 字符序列。

然而，DTD 不能指定某个元素包含字符序列，如有效的信用卡号、电话号码和电子邮件地址等。DTD 的这种局限性是促成开发 W3C XML Schema 的原因之一。

XMLSpy 和 Stylus Studio 可以根据需求创建一个 W3C XML Schema 文档来反映 XML 实例文档中的结构。在 XMLSpy 中，创建 W3C XML Schema 文档是自动发生的。

图 24-1 显示了如何在 XMLSpy 中根据 XML 实例文档——PersonDataForSchema.xml，来创建相应的模式文档。PersonDataForSchema.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonData >
   <Person>
      <LastName>Smith</LastName>
      <FirstName>John</FirstName>
   </Person>
</PersonData>
```

当单击如图 24-1 所示的菜单选项后，会打开如图 24-2 所示的对话框。要创建 W3C XML Schema 文档，则选择 W3C Schema 左侧的单选按钮。

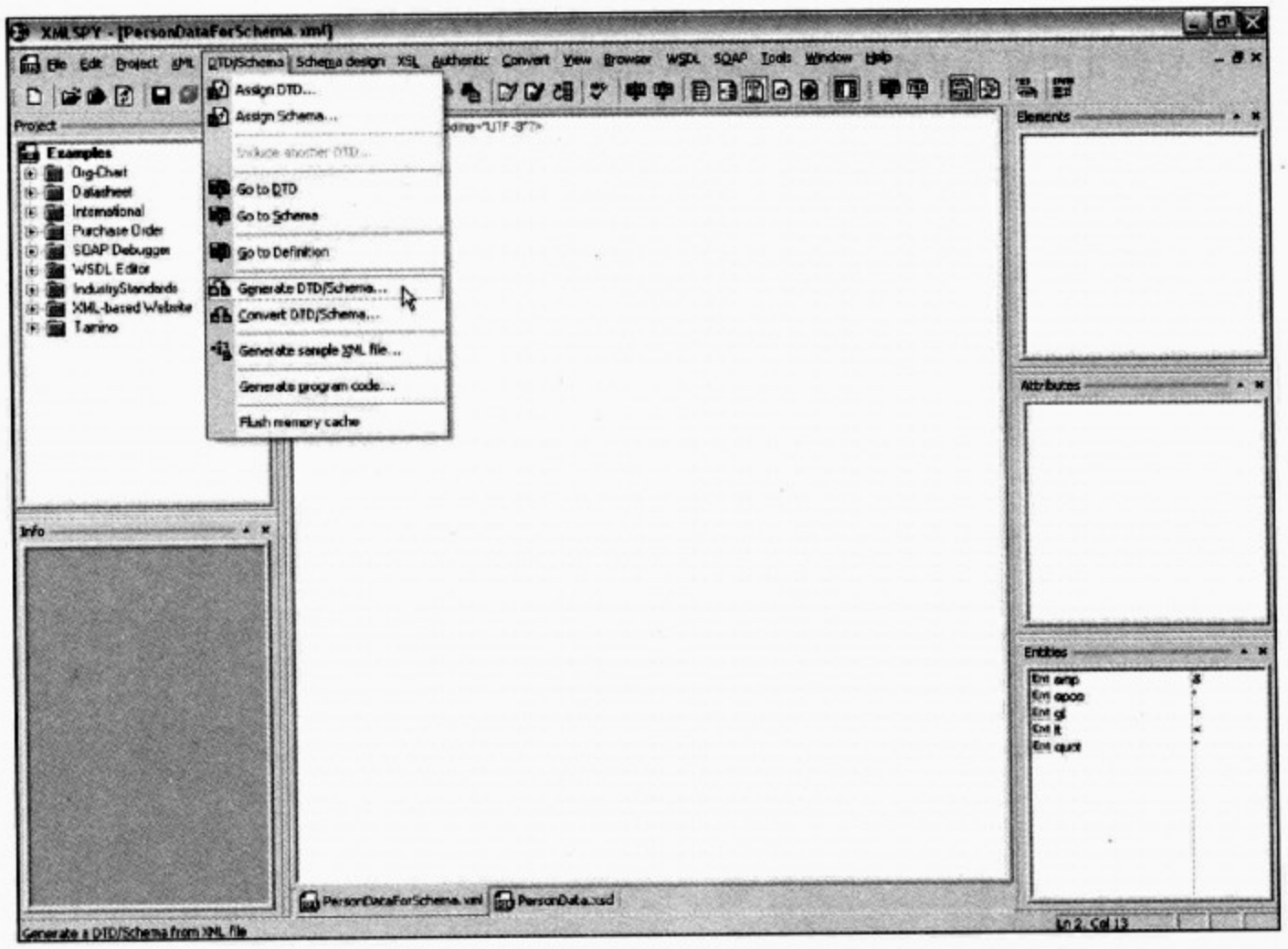


图　24-1

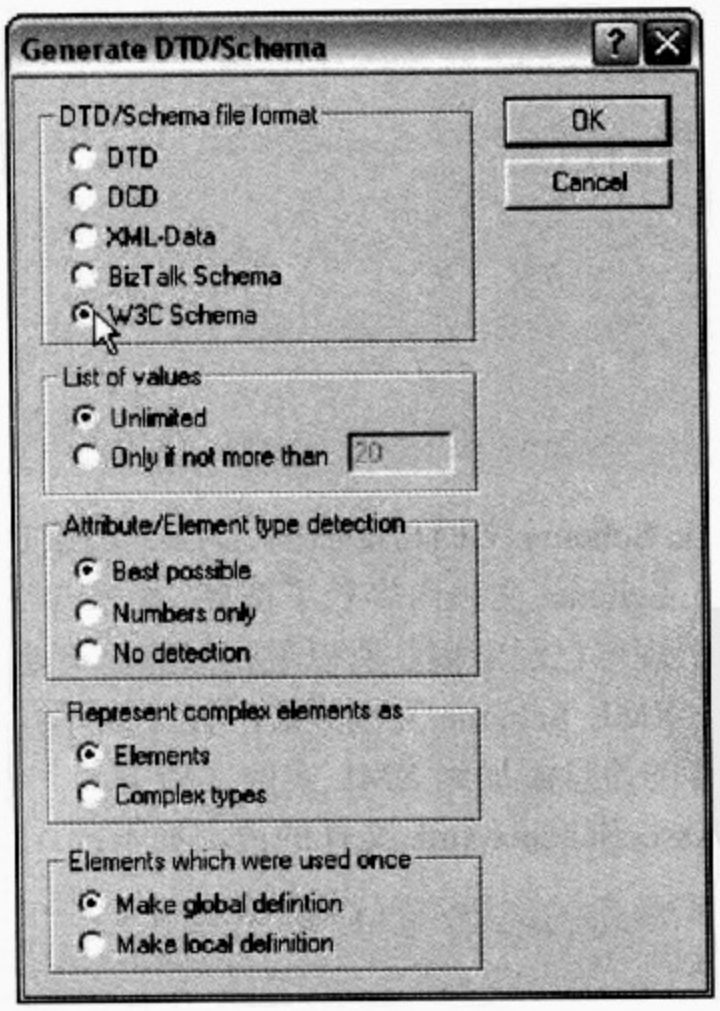


图　24-2

XMLSpy 会询问是否想将刚才创建的 W3C XML Schema 文档与 XML 实例文档关联。

图 24-3 显示了这个对话框。如果单击 Yes 按钮，XMLSpy 就会在 PersonData 元素中添加必要的代码，以便让内置于 XMLSpy、XMLWriter 以及 Stylus Studio 中的验证解析程序找到 W3C XML Schema 文档并完成验证。

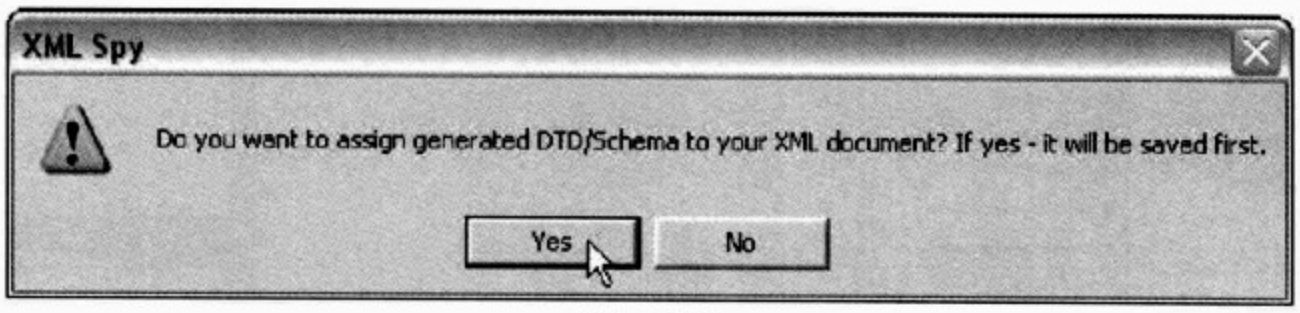


图 24-3

创建完成的 W3C XML Schema 文档——PersonData.xsd 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="FirstName" type="xs:string"/>
  <xs:element name="LastName" type="xs:string"/>
  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="LastName"/>
        <xs:element ref="FirstName"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="PersonData">
  <xs:complexType>
    <xs:sequence>
       <xs:element ref="Person"/>
     </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

如果将这个 W3C XML Schema 文档与前面的 DTD 文档 PersonData.dtd 进行比较，就会发现相应的 W3C XML Schema 文档内容长了很多。虽然 W3C XML Schema 文档的冗长招致了很多批评，但由于规范已经定稿，所以现在只能简单地接受这一现实了。

之所以要先保存 W3C XML Schema 文档(如图 24-3 所示)，是因为需要将有关 W3C XML Schema 文档保存位置的信息添加到 XML 实例文档中。

修改后的 PersonDataAssocSchema.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonData xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\PersonData.xsd">
    <Person>
        <LastName>Smith</LastName>
        <FirstName>John</FirstName>
```

```
    </Person>
</PersonData>
```

XMLSpy 为 XML Schema 的实例命名空间添加了一个命名空间声明：

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

XML Schema 实例命名空间中的属性 xsi:noNamespaceSchemaLocation 以及它的值(表示 W3C XML Schema 文档位置的 URI——即 C:\BRegExp\Ch24\PersonData.xsd)也被添加到了文档元素中。如果想要验证的 XML 实例文档和模式文档保存在其他位置，则需要适当地修改一下 xsi:noNamespaceSchemaLocation 属性的值：

```
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\PersonData.xsd"
```

在 XMLSpy 将 W3C XML Schema 文档与 XML 实例文档建立关联之后，就可以使用 XMLSpy 来验证这个 XML 实例文档了。图 24-4 中的光标正位于一个相关的工具栏按钮之上。在图 24-4 的下方，可以看到该文档相对于模式而言是有效的信息。

同样，可以在 Stylus Studio(如图 24-4 所示)或 XMLWriter(如图 24-5 所示)中验证 XML 实例文档 PersonDataAssocSchema.xml。每幅图中的箭头光标所在的位置就是用于验证 XML 实例文档的相关工具栏按钮。

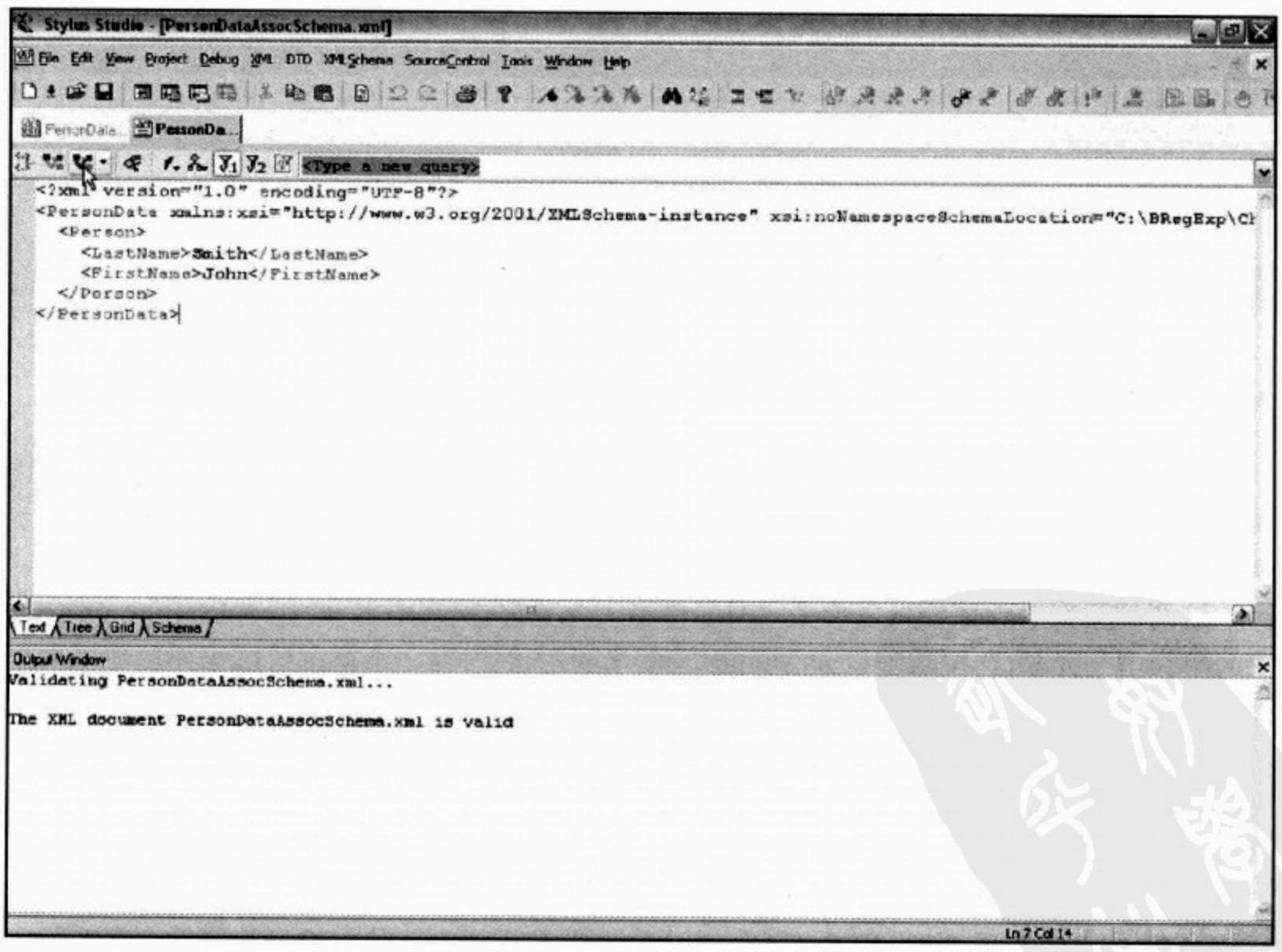


图　24-4

无论是否拥有一个 XML 编辑器，或者选择下载使用 XMLSpy、Stylus Studio 和 XMLWriter 的测试版，现在都可以基于一个模式文档来验证 XML 实例文档了。接下来，我们就开始尝试本章的例子。

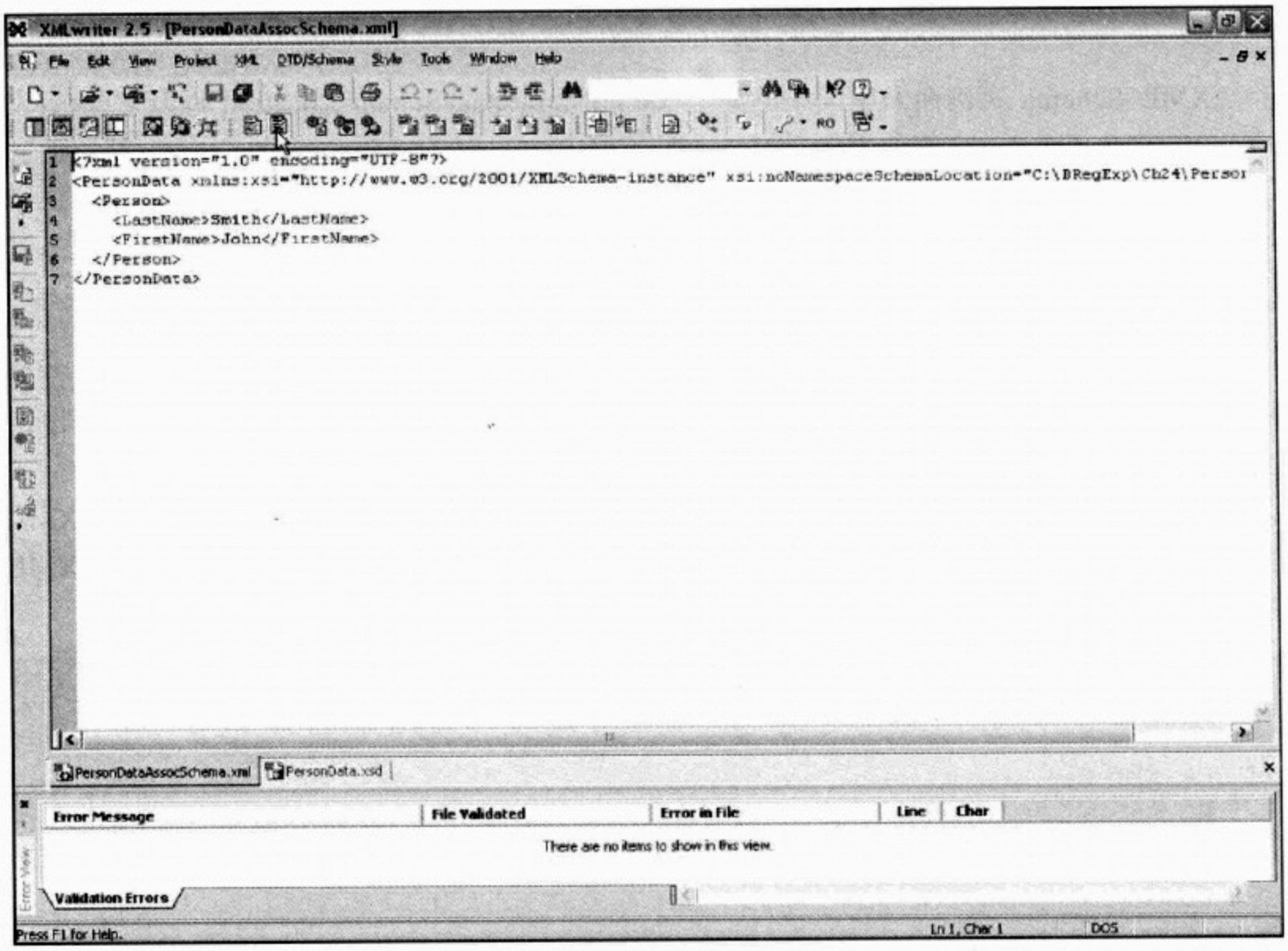


图 24-5

24.1.3 W3C XML Schema 如何表示约束

从某种意义上说，W3C XML Schema 的任务就是应用各种约束。一种约束形式就是限制应用在某个模式属于一类 XML 文档的实例文档中的元素和属性如何组织。另一种约束就是限制只有被允许的内容才能作为元素或属性的值。

这两种类型的约束都可以作为“复杂类型”元素(在模式中以 xs:complexType 表示)和“简单类型”元素(在模式中以 xs:simpleType 来表示)的内容。本章主要讨论如何通过简单类型来约束一个 XML 实例文档的值。

24.1.4 W3C XML Schema 中的数据类型

在本书已经介绍的正则表达式应用中，正则表达式都是应用于一个字符串值。在 W3C XML Schema 中，则可以将正则表达式用在其他数据类型中。

表 24-1 中总结了 W3C XML Schema 中内置的数据类型。数据类型前面的命名空间前缀 xs 表示它们属于 XML 命名空间 http://www.w3.org/2001/XMLSchema。数据类型分为原

始类型和派生类型。

表 24-1　W3W XML Schema 中内置的数据类型

数 据 类 型	说　　明
xs:anyType	功能上类似于类型层次中的根元素。由 xs:anyType 派生的类型可以是复杂类型或者简单类型
xs:anySimpleType	所有简单类型的基础类型
xs:string	有限长度的 XML 字符序列
xs:boolean	表示二进制逻辑的 true 和 false
xs:base64Binary	表示 Base64 编码的二进制数据
xs:hexBinary	表示十六进制编码的二进制数据
xs:float	表示一个 IEEE 单精度 32 位浮点数
xs:decimal	表示任意精度的十进制数
xs:double	表示一个 IEEE 双精度 64 位浮点数
xs:anyURI	表示绝对或相对的统一资源标识符，可能包括一个片断(fragment)标识符
xs:QName	一个 XML 命名空间限定(namespace-qualified)名
xs:NOTATION	表示一个 XML 1.0 中的 NOTATION 类型
xs:duration	表示一个由公历年、月、日、时、分、秒组成的时间段
xs:dateTime	表示一个特定的时间
xs:time	表示一个每天都重现的特定时间
xs:date	表示一个特定的日历天数
xs:gYearMonth	表示 xs:dateTime 中的年和月部分
xs:gMonthDay	表示一年中特定的月和日，如 9 月 25 日
xs:gDay	表示一月中特定的某一天，如 25 日
xs:gMonth	表示公历的某个月

除已列出的数据类型外，还有一些直接或间接派生自 xs:string 和 xs:decimal 的数据类型。表 24-2 中总结了派生自 xs:string 的数据类型。

表 24-2　派生自 xs:string 的数据类型

派生数据类型	说　　明
xs:normalizedString	基础类型是 xs:string。而 xs:normalizedString 类型是不包含回车(#xD)、换行(#xA)和制表(#x9)符的字符串集合
xs:token	基础类型是 xs:string。此数据类型是不包含换行(#xA)和制表符(#x9)，也不包含首尾的空格符(#x20)或内部双空格符的字符串集合
xs:language	基础类型是 xs:token。此数据类型是 XML 1.0(第二版)规范中语言标识符的一组 xs:token 值的集合

(续表)

派生数据类型	说　明
xs:Name	基础类型是 xs:token。此数据类型是由 XML 1.0(第二版)定义的合法 XML 名称的字符串集合
xs:NCName	基础类型是 xs:Name。此数据类型是不包含冒号的 XML 名称的字符串集合
xs:ID	基础类型是 xs:NCName。此数据类型表示同属于 NCName 的 ID 类型值
xs:IDREF	基础类型是 xs:NCName。此数据类型表示属于 NCName 的 IDREF 类型值的字符串集合
xs:IDREFS	项目类型是 xs:IDREF。此数据类型是一个由空格分隔的 xs:IDREF 类型值的列表
xs:NMTOKEN	基础类型是 xs:token。此数据类型是符合 XML 1.0(第二版)规范中 NMTOKEN 定义的一组 xs:token 值的集合
xs:NMTOKENS	项目类型是 xs:NMTOKEN。此数据类型是一个由空格分隔的 xs:NMTOKEN 类型值的列表
xs:ENTITY	基础类型是 xs:NCName。此数据类型表示 XML 1.0(第二版)定义的 ENTITY 类型的值
xs:ENTITIES	项目类型是 xs:ENTITY。此数据类型是一个由空格分隔的 xs:ENTITY 类型值的列表

表 24-3 中总结了直接或间接派生自 xs:decimal 的内置数据类型。

表 24-3　派生自 xs:decimal 的内置数据类型

派生数据类型	说　明
xs:integer	基础类型是 xs:decimal。此数据类型表示正、负整数值
xs:nonPositiveInteger	基础类型是 xs:integer。此数据类型表示负整数和零
xs:negativeInteger	基础类型是 xs:nonPositiveInteger。此数据类型表示负整数
xs:long	基础类型是 xs:integer。此数据类型表示 -9 223 372 036 854 775 808 ~9 223 372 036 854 775 807 之间的整数值
xs:int	基础类型是 xs:long。此数据类型表示 -2 147 483 648~2 147 483 647 之间的整数值。包含两端值
xs:short	基础类型是 xs:int。此数据类型表示 -32 768~32 767 之间的整数值。包含两端值
xs:byte	基础类型是 xs:short。此数据类型表示 -128~127 之间的整数值。包含两端值
xs:nonNegativeInteger	基础类型是 xs:integer。此数据类型表示正整数和零
xs:unsignedLong	基础类型是 xs:nonNegativeInteger。此数据类型表示 0~18 446 744 073 709 551 615 之间的整数值

(续表)

派生数据类型	说　明
xs:unsignedInt	基础类型是 xs:unsignedLong。此数据类型表示 0~4 294 967 295 之间的整数值。包含两端值
xs:unsignedShort	基础类型是 xs:unsignedInt。此数据类型表示 0~65 535 之间的整数值。包含两端值
xs:unsignedByte	基础类型是 xs:unsignedShort。此数据类型表示 0~255 之间的整数值。包含两端值
xs:positiveInteger	基础类型是 xs:nonNegativeInteger。此数据类型表示大于等于 1 的整数值

有关内置数据类型更加完整详细的定义，可以查看 XML Schema 的第二部分(www.w3.org/TR/2001/REC-xmlschema-2-20010502)、XML 1.0(第二版)(www.w3.org/TR/2000/WD-xml-2e-20000814)和 XML 中的命名空间(www.w3.org/TR/REC-xml-names)。

编程人员可以通过下列三种机制中的任何一种来基于内置类型创建自定义类型：

- 通过限制派生——通过限制可接受的值约束已有数据类型的值。
- 通过列表派生——即一个内置或用户定义的数据类型值的列表。
- 通过合并派生——即用户定义的数据类型是其他两种数据类型(内置或用户定义的数据类型)的交集。

24.1.5　通过限制派生

使用 W3C XML Schema 时，通常会存在几种方法都能够指定一种特定的结构。在前面列表提到的派生方法中，最常用的就是通过限制派生。

一种限制方法就是枚举。与下面的 XML 实例文档 BookEnum.xml 关联的就是一个包含枚举的 W3C XML Schema 文档。

```
<?xml version="1.0" encoding="UTF-8"?>
<Book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\BookEnum.xsd">
  <Chapter number="1">Some content</Chapter>
  <Chapter number="2">Some content</Chapter>
  <Chapter number="3">Some content</Chapter>
  <Chapter number="4">Some content</Chapter>
  <Chapter number="5">Some content</Chapter>
</Book>
```

相应的 W3C XML Schema 文档 BookEnum.xsd 是通过 XMLSpy 创建的，其中对 Chapter 元素的 number 属性值就是采取枚举 1~5 的值来进行约束的。

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Book">
```

```
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Chapter" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="Chapter">
      <xs:complexType>
        <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="number" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:NMTOKEN">
                <xs:enumeration value="1"/>
                <xs:enumeration value="2"/>
                <xs:enumeration value="3"/>
                <xs:enumeration value="4"/>
                <xs:enumeration value="5"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

number 属性值是个简单类型值。这个由 XMLSpy 创建的模式文档使用了 xs:NMTOKEN 数据类型，因为 XML 实例文档中的示例值涉及到该数据类型的 1、2、3、4 和 5。但是，对值的相同约束，也可以使用如下 BookPattern.xsd 中的 xs:pattern 元素来达到目的：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Book">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Chapter" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Chapter">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="number" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
            <xs:pattern value="(1|2|3|4|5)" />
```

```
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

在下载的测试代码中，提供了一个与 BookPattern.xsd 相关联的 XML 实例文档 BookPattern.xml。该文档与 BookEnum.xml 的唯一区别就是 xsi:noNamespaceSchemaLocation 属性指向 BookPattern.xsd 文件：

```
<Book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\BookPattern.xsd">
```

这个 xs:pattern 元素是本章后面内容中出现频率最高的元素，因为它是在 W3C XML Schema 中使用正则表达式的元素。xs:pattern 元素的 value 属性值是一个正则表达式模式(pattern)——因此，该元素的名称就是 pattern。

前面代码所包含的限制属性值的模式 (1|2|3|4|5) 是一个非常简单的交替选择的例子，也就是说允许值是 1、2、3、4 或 5 中的任何一个。

在介绍 W3C XML Schema 支持的元字符以及如何使用这些元字符之前，我们先来看一下与 W3C XML Schema 文档中的正则表达式相关的 Unicode 字符。

24.1.6　Unicode 与 W3C XML Schema

XML 文档是由 Unicode 字符序列组成的。Unicode 中包含数千个字符，而事实上，只有少数程序能显示全部 Unicode 字符，而且也只有极少数人能完全理解所有 Unicode 字符。为了更方便地处理 Unicode 字符，将字符分成 Unicode 字符类和 Unicode 块(blocks)。本节后面会讨论这两种类型。

> 有关 Unicode 的完整信息可以在 www.unicode.org 中找到。在本书写作时，Unicode 的最新标准是 4.0.1 版。有关 Unicode 标准的完整信息可访问 www.unicode.org/standard/standard.html。

24.1.7　Unicode 概述

Unicode 标准定义的是通用字符集。Unicode 的目标是让地球上所有语言的文本内容都能够无障碍地进行交换。Unicode 为大多数语言中的大多数字符规定了文本编码，同时也包括增进旧字符编码可用性的编码方案。

Windows Character Map 实用工具提供了查看 Unicode 编码的方便途径。图 24-6 显示了一个大写的 A 被选中的情形。注意，程序界面的底部显示出了大写 A 的 Unicode 编码是 U+0041。位于 U 和+后面的数字必须是 4 位数字。而且，这 4 位数字是十六进制数。在本例中，大写 A 的编码是十六进制的 0041，也就是十进制的 65。

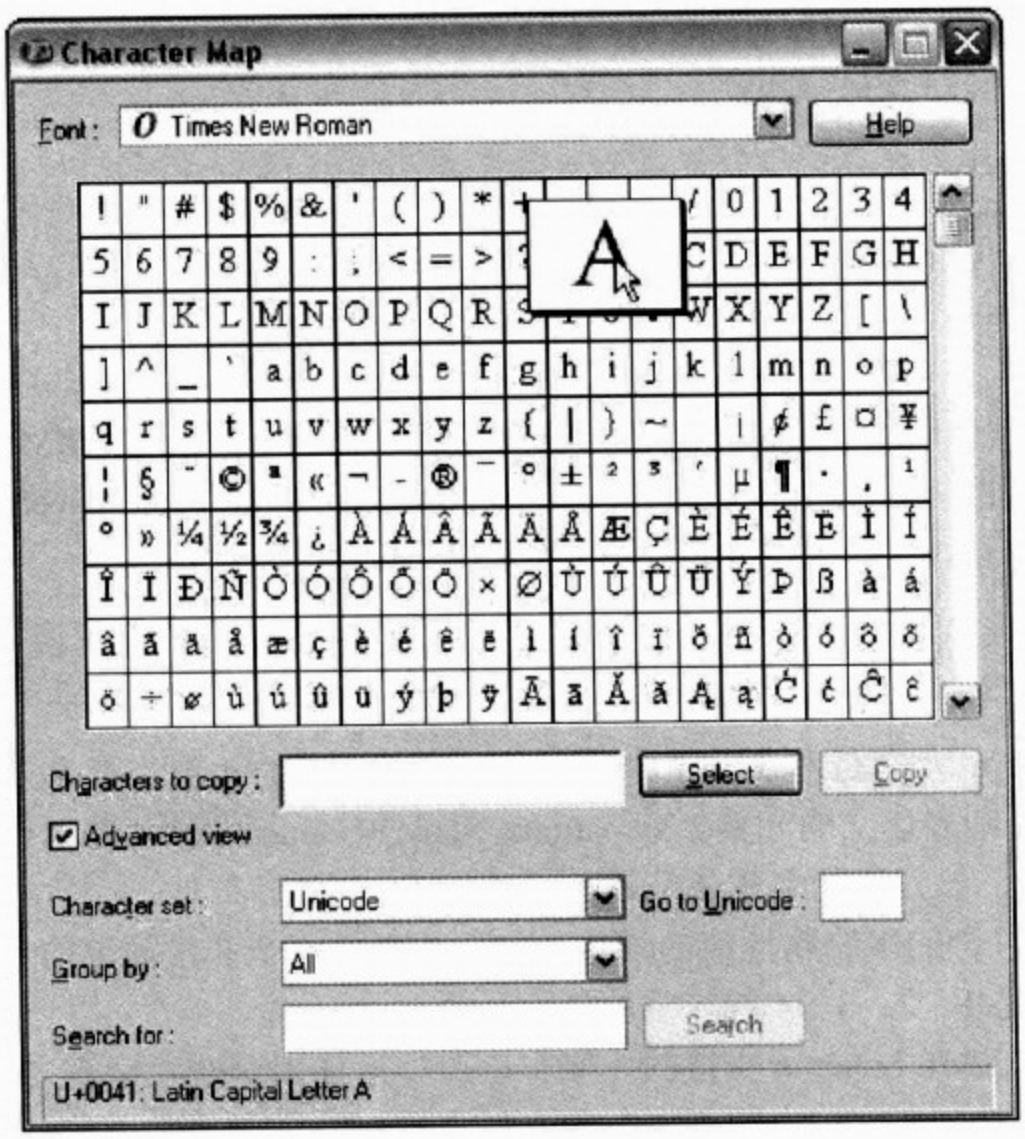


图 24-6

在 XML 中，大写的 A 也可写成 A。但在多数情况下，使用英文字母文体来表示更简单。

所谓 Unicode 字符类，指的是一组字符的使用类型(也就是具有某种共同特征的字符。译者注)——例如，小写字母。而 Unicode 字符块则表示与该字符块相关的一种语言或其他表示方法。

24.1.8 使用 Unicode 字符类

在 W3C XML Schema 文档中使用 Unicode 字符类时，字符类通过如下方式来指定：

```
\p{字符类}
```

表 24-4 中总结了 W3C XML Schema 支持的 Unicode 字符类。

表 24-4 W3C XML Schema 支持的 Unicode 字符类

Unicode 字 符 类	说　明
C	其他字符
Cc	控制字符
Cf	格式字符
Cn	未指定的编码点

(续表)

Unicode 字 符 类	说　　明
L	字母
Ll	小写字母
Lm	修饰字母
Ln	其他字母
Lt	标题格(Title-case)字母
Lu	大写字母
M	所有标记(mark)
Mc	组合空格标记
Me	封装标记
Mn	非空格标记
N	数值
Nd	十进制数
Nl	数字字母
No	其他数字
P	标点符号
Pc	连接器标点符号
Pd	破折号
Pe	结束标点符号
Pf	后引号
Pi	前引号
Po	其他形式的标点符号
Ps	开始的标点符号
S	符号(symbols)
Sc	货币符号
Sk	修饰符号
Sm	数学符号
So	其他符号
Z	分隔符
Zl	换行符
Zp	分段符
Zs	空格符

下面几节简单示范 Unicode 字符类的用法。

1. 匹配十进制数

Nd 字符类匹配十进制数。对于下面所示的测试文件 DocumentUnicode.xml，可以用 Unicode 字符类来指定 Section 元素的 number 属性的取值范围：

```
<?xml version="1.0" encoding="UTF-8"?>
<Document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\DocumentUnicode.xsd">
  <Section number="1">Content</Section>
  <Section number="2">Content</Section>
  <Section number="3">Content</Section>
</Document>
```

相应的模式文档 DocumentUnicode.xsd 中使用了 Unicode 字符类 Nd：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Document">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Section" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Section">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="number" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
              <xs:pattern value="\p{Nd}" />
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

其中，xs:pattern 元素的 value 属性值是\p{Nd}，它指定 Section 元素的 number 属性值是一个单个的十进制数。

2. 将 Unicode 字符类与其他元字符混合使用

同一个正则表达式中可以混合使用 Unicode 字符类和其他元字符。下面的例子示范了如何(以人为设计的方式)混合使用 Unicode 字符类和其他元字符匹配美国社会保险号。XML 实例文件 PersonsSSNUnicode.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonsSSN xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\PersonsSSNUnicode.xsd">
  <Person>
    <Name>Peter Schmidt</Name>
    <SSN>123-45-6789</SSN>
  </Person>
  <Person>
    <Name>Yasmin Brown</Name>
    <SSN>987-65-4321</SSN>
  </Person>
</PersonsSSN>
```

相应的 W3C XML Schema 文档 PersonsSSNUnicode.xsd 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Name" type="xs:string"/>
  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name"/>
        <xs:element ref="SSN"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="PersonsSSN">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Person" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="SSN">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="\p{Nd}{3}-[0-9]{2}-\d{4}" />
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

注意作为 xs:pattern 元素 value 属性值的模式，该模式使用了三种不同的方式来表示数字：Unicode 字符类、正则表达式字符类以及元字符 \d。其中，\p{Nd}{3}使用 Unicode 字符类匹配三个数字，然后是一个直接量连字符。然后，[0-9]{2} 通过在一个常规的字符类中使用范围来匹配两个数字。接着，又是一个直接量连字符。最后，\d{4}匹配四个数字。

24.1.9 Unicode 字符块

Unicode 字符块指与特定用途有关的一批 Unicode 字符。一个 Unicode 字符块可以表示一种语言或一组语言，或者表示某种特定的用途，如方块元素(box drawing)或几何图形。

表 24-5 列出了一些常用的 Unicode 字符块。

表 24-5 一些常用的 Unicode 字符块

块 名 称	起 始 编 码	结 束 编 码
基本拉丁语字符	#x0000	#x007F
拉丁语字符-1 增补	#x0080	#x00FF
拉丁语字符扩充 A	#x0100	#x017F
古斯拉夫语字符	#x0400	#x04FF
希伯来语字符	#x0590	#x05FF
阿拉伯字符	#x0600	#x06FF
希腊字符	#x0370	#x03FF
切罗基族字符	#x13A0	#x13FF
上标和下标	#x2070	#x209F
数学运算符	#x2200	#x22FF

使用 Unicode 字符块

这个例子示范了组合使用 Unicode 字符块和 Unicode 字符类的效果。

试一试：使用 Unicode 字符块

(1) 输入下列 XML 标记，或者打开下载的 WordUnicode.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<Word xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\WordUnicode.xsd">Führer</Word>
```

(2) 输入下列 W3C XML Schema 文档内容或打开下载的 WordUnicode.xsd 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Word" type="UnicodeType"/>
  <xs:simpleType name="UnicodeType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\w+"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

(3) 尝试用 WordUnicode.xsd 来验证 WordUnicode.xml 的有效性。图 24-7 显示的是在 XMLSpy 中执行验证的界面外观。在窗格下部可以看到，这个 XML 实例文档符合与其关联的模式文档。

(4) 输入下列 XML 标记或打开下载的 WordUnicode2.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<Word xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\WordUnicode2.xsd">Führer</Word>
```

(5) 输入下列 W3C XML Schema 文档内容或打开下载的 WordUnicode2.xsd 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Word" type="UnicodeLetterType"/>
  <xs:simpleType name="UnicodeLetterType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\p{L}+"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

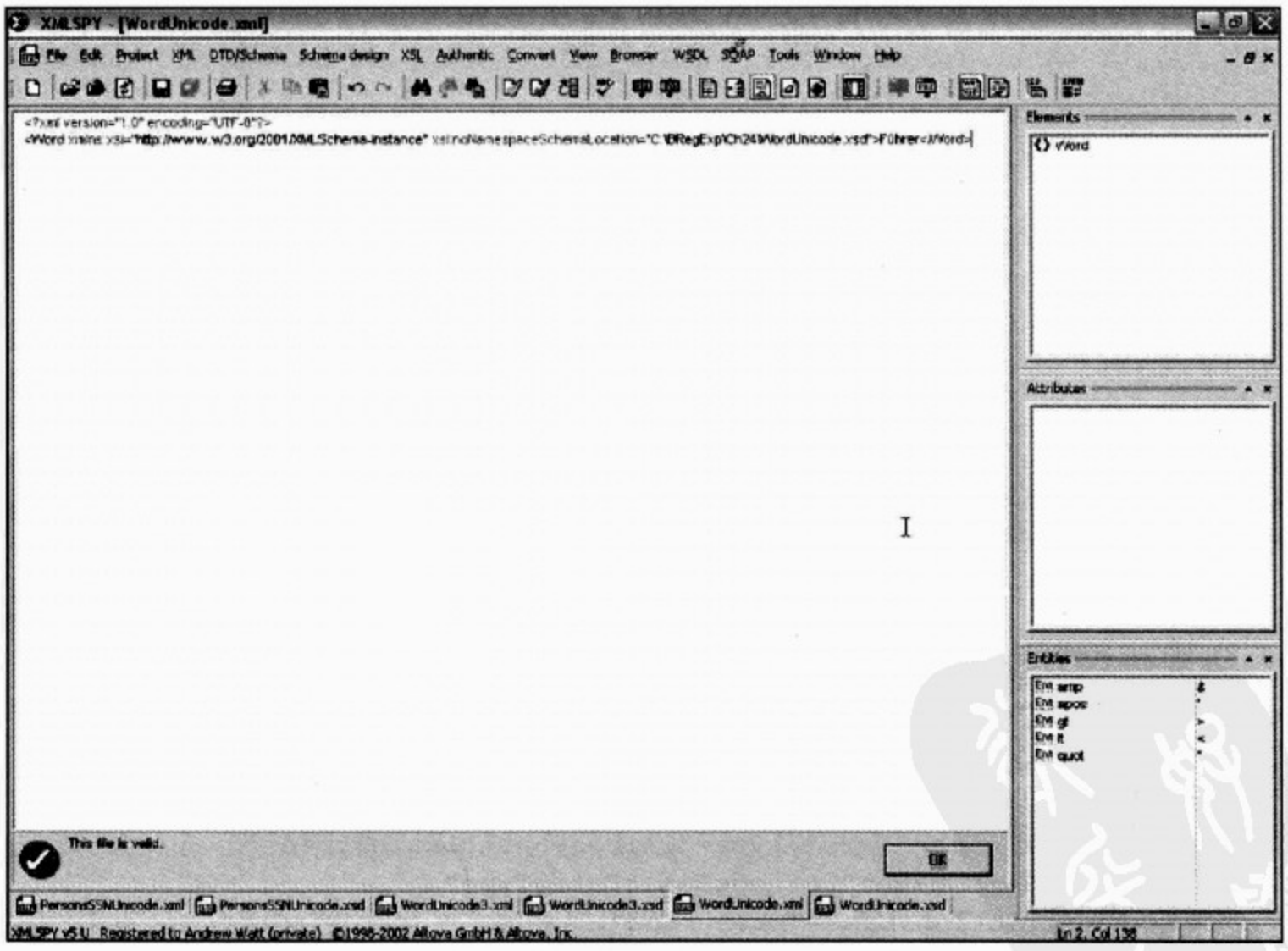


图　24-7

(6) 尝试用 WordUnicode2.xsd 来验证 WordUnicode2.xml 的有效性。图 24-8 显示了此时的界面外观。此次验证中根据模式\p{L}来验证 Führer，由于模式表示所有 Unicode 字母，

所以匹配成功。

接着，尝试用单词 Führer 来匹配基本拉丁语字母，结果没有匹配。因为字符 ü 的 Unicode 编码是 U+00FC，它位于基本拉丁语字符块编码范围 U+0000～U+007F 之外。

(7) 输入下列 XML 标记或打开下载文件 WordUnicode3.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<Word xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\WordUnicode3.xsd">Führer</Word>
```

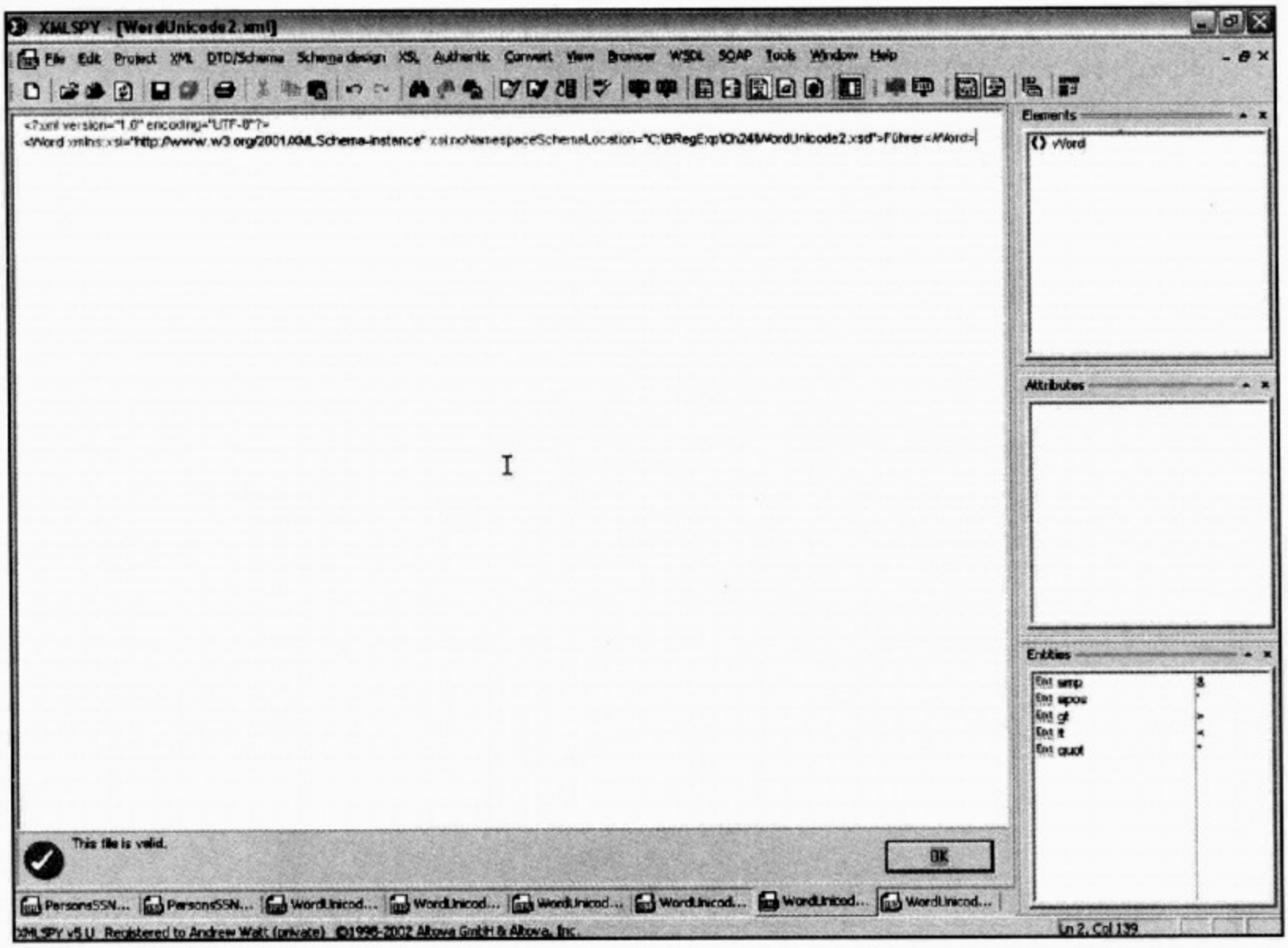


图 24-8

(8) 输入下列 W3C XML Schema 文档内容或打开下载文件 WordUnicode3.xsd：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Word" type="UnicodeBasicLatinType" />
  <xs:simpleType name="UnicodeBasicLatinType" >
    <xs:restriction base="xs:string">
    <xs:simpleType>
      <xs:restriction base="xs:string">
       <xs:pattern value="\p{IsBasicLatin}" />
      </xs:restriction>
    </xs:simpleType>
```

```
      <xs:pattern value="\p{L}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

注意如何使用由模式\p{L}指定的 Unicode 字符类和由模式\p{IsBasicLatin}指定的 Unicode 字符块。

(9) 尝试用 WordUnicode3.xsd 来验证 WordUnicode3.xml 的有效性(如图 24-9 所示)。此时，不匹配。

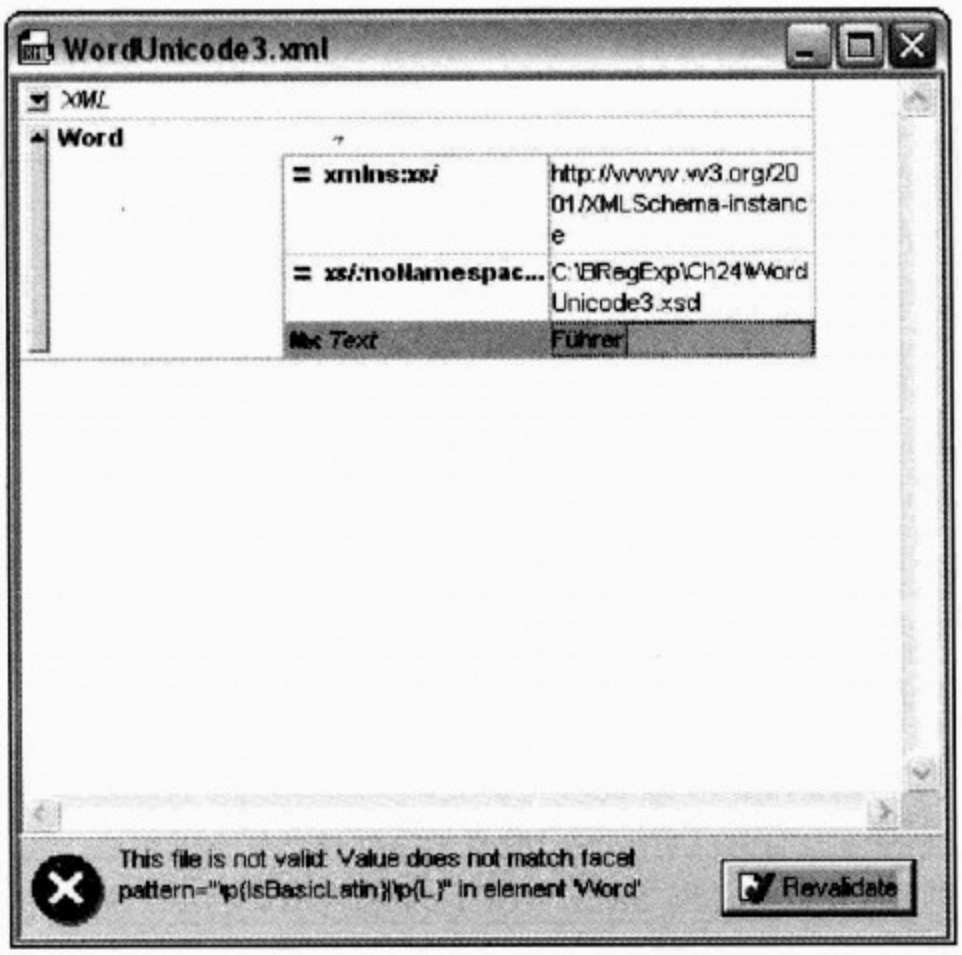


图　24-9

工作原理

文件 WordUnicode.xml 与 WordUnicode.xsd 尝试用模式 \w+ 来验证德语单词 Führer(领导者)。说明在 W3C XML Schema 中，元字符能匹配一些英语中不使用的字母。

文件 WordUnicode2.xml 与 WordUnicode2.xsd 尝试用模式 \p{L} 来验证德语单词 Führer(领导者)。由于单词 Führer 全部由 Unicode 字母组成，所以匹配成功。

当文件 WordUnicode3.xml 与 WordUnicode3.xsd 尝试用模式\p{L}来验证德语单词 Führer(领导者)时，同时也使用了 Unicode 字符块——基本拉丁语字符(\p{IsBasicLatin})。由于单词 Führer 中包含字母 ü，而它不在范围 U+0000 ~ U+007F 之间(ü的编码是 U+00FC)，所以不匹配，验证失败。

24.1.10　W3C XML Schema 支持的元字符

W3C XML Schema 支持的元字符中包含一些直接与 XML 相关的元字符，而这些元字符在大多数其他正则表达式实现中都是没有的。

表 24-6 总结了 W3C XML Schema 1.0 版支持的元字符。同时，参见上一节中有关 W3C XML Schema 支持的 Unicode 的内容。

表 24-6　W3C XML Schema 1.0 版本支持的元字符

元 字 符	说　明
^	不支持在取反的字符类之外使用(参见讨论位置元字符的内容)
$	不支持(参见讨论位置元字符的内容)
\d	匹配一个数字
\D	匹配一个非数字字符
\s	匹配一个空白字符
\S	匹配一个非空白字符
\w	匹配一个“单词”字符
\W	匹配一个非“单词”字符
| (竖线)	交替选择。允许在前后的两个或多个组或字符之间选择一项
?	限定符。匹配前面字符或组的零个或一个实例
*	限定符。匹配前面字符或组的零个或多个实例
+	限定符。匹配前面字符或组的一个或多个实例
{n,m}	限定符。匹配前面字符或组至少 n 个、最多 m 个实例
.(句点字符)	匹配除换行符外的任何字符
[...]	字符类。匹配包含在方括号中的字符一次
[^...]	取反的字符类。匹配不包含在方括号中的字符一次
\i	匹配一个允许作为 XML 名称首字符的字符。相当于字符类 [A-Za-z_]
\I	匹配一个不允许作为 XML 名称首字符的字符。相当于字符类 [^A-Za-z_]
\c	匹配一个 XML 1.0 名称字符。包括字符类 [A-Za-z0-9.:_]
\C	匹配一个非 XML 1.0 名称字符

24.1.11　位置元字符

由于 W3C XML Schema 与其他正则表达式实现的匹配环境不同，W3C XML Schema 不支持匹配一行(或一个字符串)开始或结束位置的 ^ 和 $ 元字符。

在很多正则表达式实现中，模式[A-Z][0-9]会匹配包含一个大写字母字符后跟一个数字的任何字符串。然而，在 W3C XML Schema 中，只有整个字符串与模式匹配才意味着存在一个匹配项。换句话说，在 W3C XML Schema 中，模式 [A-Z][0-9] 会被解释成类似其他正则表达式实现中的 ^[A-Z][0-9]$。

由于在 W3C XML Schema 中，所有正则表达式模式都按照这种好像是存在 ^ 和 $ 元字符的方式被解释，所以不单独支持 ^ 和 $ 元字符。

不过，^ 元字符仍然可以用于对字符类取反。

24.1.12　匹配数字

\d 元字符可以用于匹配数字。例如，在测试文件 Document.xml 中包含一个值必须为一位数字的 number 属性：

```
<?xml version="1.0" encoding="UTF-8"?>
<Document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\Document.xsd">
  <Section number="1">Content</Section>
  <Section number="2">Content</Section>
  <Section number="3">Content</Section>
</Document>
```

相应的 W3C XML Schema 文档 Document.xsd 就在 xs:pattern 元素中使用 \d 元字符来指定其 Section 元素的 number 属性值必须为一位数字：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Document">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Section" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Section">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="number" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
              <xs:pattern value="\d" />
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

xs:restriction 元素的基本属性值是类型 xs:NMTOKEN，但也可以使用其他的类型，比如 xs:byte 等。

24.1.13　交替选择

W3C XML Schema 支持交替选择。本章前面所使用的测试文件 BookPattern.xml 和 BookPattern.xsd 已显示了如何在 xs:pattern 元素中使用交替选择。

24.1.14 使用\w 和\s 元字符

\w 元字符表示的是单词(字母)字符，包括 A～Z 的大小写形式。\s 元字符表示一个空白符。

模式 \w+\s+\w+可以用于表示名后跟一个或多个空格符，再跟姓的名字。测试文件 Name.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<Names xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\Name.xsd">
  <Name>John Smith</Name>
  <Name>Alicia Manton</Name>
  <Name>Pierre Laval</Name>
</Names>
```

对应的模式文档 Name.xsd，使用模式\w+\s+\w+来指定 Name 元素的值应该如何构成：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Names">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" maxOccurs="unbounded">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:pattern value="\w+\s+\w+" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

这个模式匹配一个单词字符序列后跟一个或多个空白符，再跟一个单词字符序列。

但是，这个模式不会匹配像 Maria Von Trapp 这样的字符序列，因为在 W3C XML Schema 中模式 \w+\s+\w+ 的含义就相当于其他正则表达式实现中的模式 ^\w+\s+\w+$。

24.1.15 转义元字符

如果要匹配的字符是正则表达式模式中的元字符，则需要使用一个前置的反斜杠字符来转义这个字符。

表 24-7 总结了 W3C XML Schema 中的转义字符组合和在使用转义字符组合时才会匹配的字符。

表 24-7　W3C XML Schema 中的转义字符组合及其匹配的字符

转义字符组合	匹配的字符
\n	换行符
\r	回车符
\\	\(反斜杠)
\\|	|(竖线)
\.	.(句点)
\-	-(连字符)
\^	^(脱字字符)
\?	?
*	*
\+	+
\(	(
\)	)
\[	[
\]	]
\{	{
\}	}

24.2　练习

1. 请修改 Name.xsd，使得能用它来验证下面所示的 Name2.xml 文档。注意，最后两个 Name 元素的值与当前的模式 \w+\s+\w+ 不匹配。文件 Name2.xsd 中通过 Names 元素的 xsi:noNamespaceSchemaLocation 属性值提供了一种方案：

```
<?xml version="1.0" encoding="UTF-8"?>
<Names xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\Name2.xsd">
   <Name>John Smith</Name>
   <Name>Alicia Manton</Name>
   <Name>Pierre Laval</Name>
   <Name>Maria Von Trapp</Name>
   <Name>John James Manton</Name>
</Names>
```

2. 请使用 Unicode 字符类指定一个能够匹配下列零件编号的模式：

- A99
- BC9933
- DEF88125

- Z1

参考测试文件 PartNumbers.xml，其内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<PartNumbers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\BRegExp\Ch24\PartNumbers.xsd">
  <PartNumber>A99</PartNumber>
  <PartNumber>BC9933</PartNumber>
  <PartNumber>DEF88125</PartNumber>
  <PartNumber>Z1</PartNumber>
</PartNumbers>
```

第25章

Java 中的正则表达式

Java 是一门被广泛使用的编程语言，可以在包括 Windows 在内的多种平台上使用。有一些以 Java 或为 Java 编写的包中支持正则表达式功能。但是，因为现已成为 Java 2 一部分的 java.util.regex 包极好地支持了正则表达式功能，所以本章仅就 java.util.regex 这个 Sun Java 官方下载包来讨论 Java 中的正则表达式功能。

Java 中的正则表达式可以用来验证文本，也可以对文本进行搜索和替换。Java 支持的字符类相当广泛，包括标准的正则表达式字符类、POSIX 字符类以及 Unicode 字符类。此外，java.util.regex 包也提供其他方面的丰富功能。

本章假设读者对 Java 编程有基本的理解。所提供的例子旨在示范在 Java 中如何使用正则表达式功能。而且，所有例子都刻意保持了简短的特点。如果读者具有任何编程语言的经验，理解本章所提供的 Java 例子也不是件难事。但是，如果读者丝毫没有 Java 编程经验，建议通过学习相关的图书——比如 Ivor Horton 编著的《Beginning Java 2(Wrox Press 2002)》——来掌握必要的基本知识。

在本章中将学习以下内容：

- Java 2 标准版中的 java.util.regex 包
- java.util.regex 包支持的元字符
- 如何使用众多的元字符匹配和替换文本
- 如何使用 String 类的方法实现正则表达式功能

本章中的例子都在 Java 5.0 中测试通过。Java 5.0 中的正则表达式功能较之以前版本没有变化。

25.1 java.util.regex 包简介

java.util.regex 包是在 J2SE 1.4 Java 2 Standard Edition Version 1.4 中引入的。因此，本章中的例子不能在 Java 1.4 之前的版本中运行。

java.util.regex 包中有三个类——Pattern、Matcher 和 PatternSyntaxException。本节稍后

将分别介绍这三个类。首先，来看一看如何获取并通过配置使用 Java 支持 java.util.regex 包。

25.1.1 获取并安装 Java

如果机器中没有安装 Java，那么要运行本章中的例子就必须先下载并安装一个最新的 Java 2 标准版，因为该版本支持 java.util.regex。在写作本书时，可以选择下载 Java 1.4.2 或 Java 5.0。这两个版本都属于 Java 2 的范畴。

Java 2 标准版可以从 Sun Java 的网站 http://java.sun.com 中下载。在写作本书时，当前可用的 Java 2 标准版的信息可以在 http://java.sun.com/j2se/ 中找到。

Sun 的 Java 网站中提供了适合 32 位和 64 位平台安装的说明。在本书写作时，可以在 http://java.sun.com/j2se/1.5.0/install.html 中找到相关的安装信息。

Java 5.0 和 1.5 的命名在 Sun 的文档中是互相矛盾的。例如，在前面的网页中使用 1.5.0 来表示名为 Java 5.0 的网站页面。事实上，Java 1.5 和 Java 5.0 指的都是同一个 Java 版本。

在 Windows 平台中安装 Java 很简单，只需简单地执行程序并做几个选择即可。在写作本书时，可以在 http://java.sun.com/j2se/1.5.0/download.jsp 中下载到安装程序。在该 URL 中，还可以下载到 Java 5.0 文档的扩展包。

25.1.2 Pattern 类

java.util.regex.Pattern 表示编译后的正则表达式。Pattern 类没有公共的构造函数。要创建一个模式(pattern)对象，必须使用该类的静态 compile()方法。

正则表达式模式是以一个字符串表示的。该正则表达式通过 compile()方法编译成一个 Pattern 类的实例。而这个 Pattern 对象可以用于创建一个 Matcher 对象，通过 Matcher 对象就能依照在 Pattern 对象中定义的正则表达式来匹配任意字符序列。

典型的 Pattern 和 Matcher 对象的用法如下：

```
Pattern myPattern = Pattern.compile("正则表达式");
Matcher myMatcher = myPattern.matcher("测试字符串");
boolean myBoolean = myMatcher.matches();
```

前面的代码假定代码中已经存在下列 import 语句：

```
import java.util.regex.*;
```

Pattern 类的实例是不可变的，因此在多线程环境下使用是安全的。

25.1.3 使用静态方法 matches()

如果只想使用一次正则表达式，可以选择静态的 matches()方法。使用静态方法 matches()对于只执行一次匹配非常方便。

matches()方法接受两个参数。第一个参数是正则表达式模式——以一个字符串表示；第二个参数是一个字符序列(CharSequence)——即匹配测试的字符串。

在使用静态 matches() 方法时，可以参照如下代码：

```
Pattern.matches(正则表达式模式, 字符序列);
```

假设想用模式 [A-Z] 来匹配字符串 George W. Bush and John Kerry were the US Presidential candidates in 2004 for the two main political parties，那么可以这样编写代码：

```
boolean myBoolean = Pattern.matches("[A-Z]", "George W. Bush and John Kerry
were the US Presidential candidates in 2004 for the two main political parties");
```

因为 matches() 方法的第二个参数，即测试字符序列中包含不止一个大写的字母字符，所以变量 myBoolean 的值为 true。

25.1.4　两个简单的 Java 例子

第一个例子的目标是查找字符序列 the 以及包含它的单词中的后续字符。相应的测试字符串如下：

```
The theatre is the greatest form of live entertainment according to thespians.
```

问题定义可以这样描述：

匹配包含字符序列 t 后跟 h，后跟 e，以及后跟其他字符的单词，直到遇到一个词边界为止。

能够解决以上问题的模式如下：

```
the[a-z]*\b
```

首先，该模式简单地匹配直接量字符序列 the。然后，匹配以 [a-z]* 表示的零个或多个小写的字母字符。最后，匹配以 \b 表示的一个词边界。

当在一个赋值语句中使用模式 the[a-z]*\b 时，必须对其中的\b 元字符进行转义。因此，要将模式写成 the[a-z]*\\b。但对于从一个文本文件中取得的模式，则不必进行此类转义。

下面的指示假设已安装 Java 并且可以在计算机的任何目录中运行 Java 代码(即已正确设置了环境变量。译者注)。

试一试：使用 Pattern 类和 Matcher 类

(1) 在文本编辑器中，输入下列 Java 代码：

```
import java.util.regex.*;

public class Find_the{
  public static void main(String args[])
  throws Exception{

  String myTestString = "The theatre is the greatest form of live entertainment
according to thespians.";

  String myRegex = "the[a-z]*\\b";
```

```
    Pattern myPattern = Pattern.compile(myRegex);

    Matcher myMatcher = myPattern.matcher(myTestString);
    String myGroup = "";

    System.out.println("The test string was: '" + myTestString + "'.");
    System.out.println("The regular expression was '" + myRegex + "'.");
    while (myMatcher.find())
          {
           myGroup = myMatcher.group();
           System.out.println("A match '" + myGroup + "' was found.");
          } // end while

           if (myGroup == ""){
             System.out.println("There were no matches.");
           } // end if
   } // end main()
}
```

(2) 将代码保存为 Find_the.java。

(3) 在命令行中，输入命令 javac Find_the.java 并回车，将源代码编译为一个类文件。

(4) 在命令行中，输入命令 java Find_the 并回车来运行这些代码，同时观察如图 25-1 所示结果。

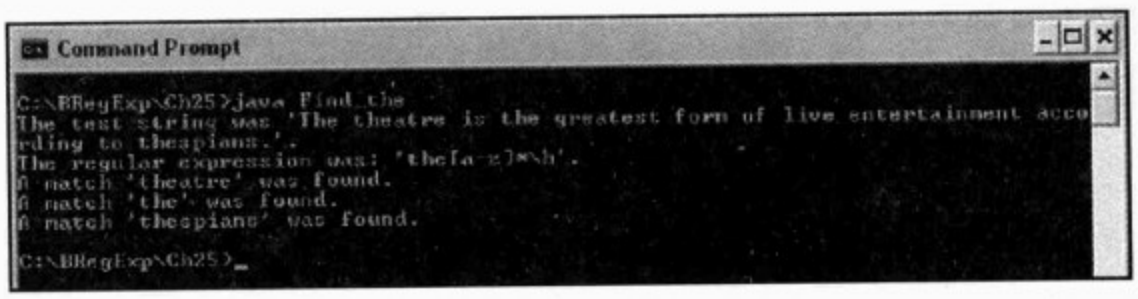


图 25-1

工作原理

Java 编译器 javac 用于编译代码。在编译代码时，一定要正确地输入文件名——包括 .java 扩展名。否则，代码很可能不会被编译。

Java 解释器 java 用于运行编译后的代码。

为方便地使用 java.util.regex 包中的类，通常在代码中导入该包：

```
import java.util.regex.*;
```

这样，开发人员就可以像下面这样来调用该包中的类及其成员了：

```
Pattern myPattern = Pattern.compile(myRegex);
```

假如没有上面的 import 语句，就必须在每行代码中都使用 Pattern 类的完整限定名，如下：

```
java.util.regex.Pattern myPattern = java.util.regex.Pattern.compile(myRegex);
```

即使像这样简单的代码，短代码行的易读性也是显而易见的。

将测试字符串指定给变量 myTestString：

```
    String myTestString = "The theatre is the greatest form of live entertainment
according to thespians.";
```

将一个字符串值指定给变量 regex：

```
    String myRegex = "the[a-z]*\\b";
```

此时编写的正则表达式模式的语法在本书之前的程序和语言中还没有出现过。\b 元字符匹配的是位于一个单词字符和一个非单词字符之间的位置。然而，要告诉 Java 编译器你想使用 \b，就必须对其中的反斜杠进行转义——写成 \\b。

如果试图声明 myRegex 变量并给它指定下面的值：

```
    String myRegex = "the[a-z]*\b";
```

结果会与你的想法背道而驰。因为 \b 将被解释为一个退格符。图 25-2 显示的是在编译运行测试文件 UnescapedFind_the.java 中这样的 Java 代码后的结果。

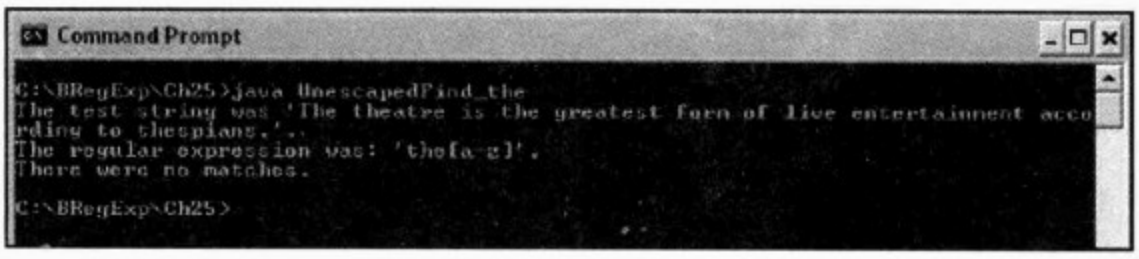


图　25-2

myPattern 变量是作为一个 Pattern 对象声明的，而指定给它的值则是由以 myRegex 变量为参数的 compile()方法返回的：

```
    Pattern myPattern = Pattern.compile(myRegex);
```

然后声明 myMatcher 变量，它是一个 Matcher 对象，指定给它的值是由以 myTestString 变量作为其参数的、myPattern 对象的 matcher()方法返回的。由于没有创建 Mathcer 对象的公共构造函数，所以如果想创建一个 Matcher 对象，就必须使用如下方式：

```
    Matcher myMatcher = myPattern.matcher(myTestString);
```

通过 System.out 中的 println()方法将包含在变量 myTestString 中的测试字符串的值和包含在变量 myRegex 中的正则表达式显示出来：

```
    System.out.println("The test string was '" + myTestString + "'.");
    System.out.println("The regular expression was: '" + myRegex + "'.");
```

然后，通过一个 while 循环来测试是否存在匹配项。如果存在匹配项，则 myMatcher.find()方法的返回值为 true。因此，包含在 while 循环中的代码会在每找到一个匹配项时执行一次：

```
    while (myMatcher.find())
    {
```

将 group() 方法返回的值指定给变量 myGroup：

```
myGroup = myMatcher.group();
```

然后使用 println() 方法显示每一次循环中找到的匹配项的值：

```
System.out.println("A match '" + myGroup + "' was found.");
} // end while
```

如果没有找到匹配项，myGroup 变量的值是空字符串。那么，使用 println()方法来显示没有找到匹配项的信息：

```
if (myGroup == ""){
System.out.println("There were no matches.");
} // end if
```

以上介绍的代码可以显示测试字符串中所有以 the 开头的字符序列。

如果仔细观察变量 myTestString 的值，会发现存在四个可能的匹配项—— The、theatre、the 和 thespians。

```
String myTestString = "The theatre is the greatest form of live entertainment
according to thespians.";
```

这说明在 Java 中，默认的匹配是区分大小写的，因此，由于字符序列 The 的第一个字符是大写的，所以它不是匹配项。

但是，单词 theatre 会匹配。因为模式中的组件 [a-z]* 匹配字符序列 atre。而单词 the 匹配，是因为组件 [a-z]* 匹配零个字符。而单词 thespians 匹配，则是因为组件 [a-z]* 匹配了字符序列 spians。

第二个例子使用一个文本文件来保存正则表达式模式，而使用另一个文本文件来保存测试文本。

试一试：从文件中取得数据

(1) 在文本编辑器中输入下列代码：

```
import java.io.*;
import java.util.regex.*;

public final class RegexTester {
 private static String myRegex;
 private static String testString;
 private static BufferedReader myPatternBufferedReader;
 private static BufferedReader myTestStringBufferedReader;
 private static Pattern myPattern;
 private static Matcher myMatcher;
 private static boolean foundOrNot;

 public static void main(String[] argv) {
  findFiles();
  doMatching();
```

```
    tidyUp(); }

    private static void findFiles() {
     try {
      myPatternBufferedReader = new BufferedReader(new FileReader("Pattern.txt"));
         }
     catch (FileNotFoundException fnfe) {
      System.out.println("Cannot find the Pattern input file! "+fnfe.getMessage());

      System.exit(0); }
      try { myRegex = myPatternBufferedReader.readLine();
       }
       catch (IOException ioe) {}
// Find and open the file containing the test text
       try {
     myTestTextBufferedReader = new BufferedReader(new FileReader("TestText.txt"));
        }
    catch (FileNotFoundException fnfe) {
     System.out.println("Cannot locate Test Text input file! "+fnfe.getMessage());
     System.exit(0); }
     try {
      testString = myTestTextBufferedReader.readLine();
      }

     catch (IOException ioe) {}
    myPattern = Pattern.compile(myRegex);
    myMatcher = myPattern.matcher(testString);
    System.out.println("The regular expression is: " + myRegex);
    System.out.println("The test text is: " + testString);
    } // end of findFiles()

    private static void doMatching()
    {
     while(myMatcher.find())
      {
       System.out.println("The text \""
        + myMatcher.group() + "\" was found, starting at index "
        + myMatcher.start() + " and ending at index "
        + myMatcher.end() + ".");
       foundOrNot = true; }
       if(!foundOrNot){ System.out.println("No match was found.");
       }
     } // end of doMatching()

       private static void tidyUp()
       {
        try{
            myPatternBufferedReader.close();
            myTestTextBufferedReader.close();
       }catch(IOException ioe){}
```

```
        } // end of tidyUp()
    }
```

(2) 将代码保存为 RegexTester.java。在命令行中输入命令 javac RegexTester.java 并回车来编译代码。

(3) 在文本编辑器中输入下列代码，并将其保存为 Pattern.txt。

```
\d\w
```

然后新建一个文本文件，在其中输入下列代码，并保存为 TestText.txt。

```
3D 2A 5R
```

(4) 在命令行中输入命令 java RegexTester 来运行这些代码。如图 25-3 所示，TestText.txt 中的三个字符序列都匹配。

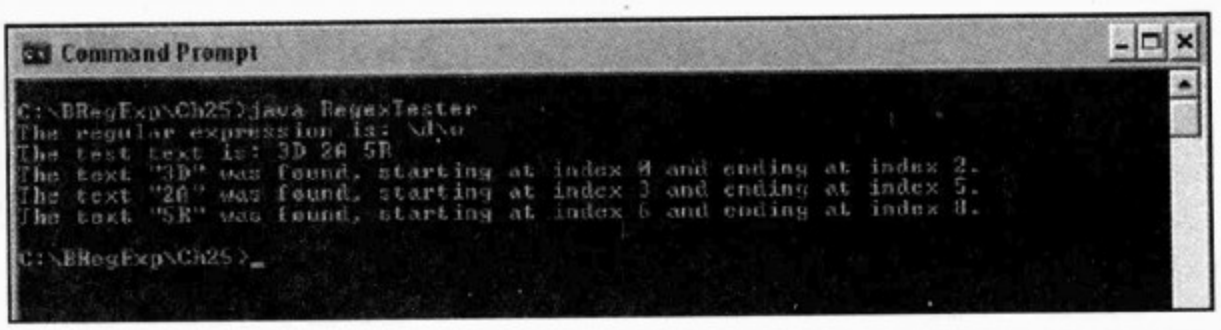


图 25-3

工作原理

这个例子从文本文件中读取数据，因此需要导入 java.io 包和 java.util.regex 包：

```
import java.io.*;
import java.util.regex.*;
```

集中声明一批后面代码中将会用到的变量：

```
private static String myRegex;
private static String testString;
private static BufferedReader myPatternBufferedReader;
private static BufferedReader myTestTextBufferedReader;
private static Pattern myPattern;
private static Matcher myMatcher;
private static boolean foundOrNot;
```

main()方法由三个方法组成：findFiles()、doMatching()和 tidyUp()。

```
public static void main(String[] argv) {
 findFiles();
 doMatching();
 tidyUp(); }
```

findFiles()方法使用了 try...catch 块来测试文件 Pattern.txt 是否存在：

```
private static void findFiles() {
```

```
try {
 myPatternBufferedReader = new BufferedReader(new FileReader("Pattern.txt"));
    }
```

如果不存在，会显示一个错误信息，程序终止：

```
catch (FileNotFoundException fnfe) {
 System.out.println("Cannot find the Pattern input file! "+fnfe.getMessage());
 System.exit(0); }
```

如果找到 Pattern.txt 文件(也就是说没有因为出错而中断程序执行)，则使用 myPattern BufferedReader 对象(BufferedReader 类的一个实例)的 readLine()方法读取 Pattern.txt 中的一行文本并将该行文本指定给变量 myRegex：

```
try { myRegex = myPatternBufferedReader.readLine();
```

同样地，myTestTextBufferedReader 对象用于处理测试文本文件 TestText.txt。其第一行内容会被指定给变量 testString。

在将读取的值赋给变量 myRegex 和 testString 之后，再使用 Pattern 类的 compile()方法创建一个 Pattern 对象——myPattern：

```
myPattern = Pattern.compile(myRegex);
```

然后，再使用 myPattern 对象的 matcher()方法来创建一个 Matcher 对象——myMatcher：

```
myMatcher = myPattern.matcher(testString);
```

最后，在显示变量 myRegex 和 testString 的值以表示两个文件都加载成功后，findFiles()方法执行完毕：

```
System.out.println("The regular expression is: " + myRegex);
System.out.println("The test text is: " + testString);
}
```

然后执行 doMatching()方法：

```
private static void doMatching()
{
```

这个方法使用了 while 循环来处理找到的每一个匹配项：

```
while(myMatcher.find())
 {
```

对于每个匹配项，分别使用 myMatcher 对象的 group()、start()和 end()方法显示匹配项、匹配项的开始位置和结束位置：

```
System.out.println("The text \\""
 + myMatcher.group() + "\" was found, starting at index "
 + myMatcher.start() + " and ending at index "
 + myMatcher.end() + ".");
```

只要找到匹配项，while 循环的最后一行就会把变量 foundOrNot 的值设置为 true：

```
    foundOrNot = true; }
```

然后，再使用 if 语句来测试变量 foundOrNot 的值。如果不是 true，则显示没有找到匹配项的信息：

```
      if(!foundOrNot){ System.out.println("No match found.");
      }
    }
```

最后，通过 tidyUp() 方法来完成清理工作。

本例中使用的是在文件 Pattern.txt 中定义的模式：

```
    \d\w
```

该模式匹配一个数字后跟一个单词字符(也就是说，一个任意大小写的字母字符、下划线或数字)。

测试字符串则放在文件 TestText.txt 中：

```
    3D 2A 5R
```

对于模式 \d\w 而言，有三个匹配项：3D、2A 和 5R。

1. Pattern 类的属性(字段)

下面的表 25-1 中总结了 Pattern 类的属性(字段)。

表 25-1 Pattern 类的属性(字段)

属性(字段)	说　明
CANON_EQ	在匹配时启用规范等价(canonical equivalence)
CASE_INSENSITIVE	不区分大小写的匹配
COMMENTS	对模式中空白符和注释的支持(即忽略模式中的空白符。译者注)
DOTALL	设置此标志后，.(句点)元字符匹配所有字符
MULTILINE	设置此标志后，会修改 ^(脱字字符)和 $(美元符号)位置元字符的含义
UNICODE_CASE	设置此标志后，会对所有 Unicode 字母字符应用不区分大小写的匹配(若适合)
UNIX_LINES	设置此标志后，只有 \n 行终止符会影响 .(句点)、^(脱字字符)和 $(美元符号)元字符的行为

2. CASE_INSENSITIVE 标志

CASE_INSENSITIVE 标志只对美国 ASCII 字符有效。如果需要在对其他字符的匹配中使用不区分大小写的方式，可能需要使用 UNICODE_CASE 标志。

也可以通过嵌入式标志表达式(?i)来指定 CASE_INSENSITIVE 标志。

3. 使用 COMMENTS 标志

当设置 COMMENTS 标志时，在正则表达式中包含的空白符将被忽略，即不会与测试字符序列中的空白符匹配。这样，就可以使代码中的模式(以及说明模式中组件作用的注释)能够以人类容易阅读和理解的方式来显示。

注释要以 # 字符开头。所有 # 字符之后的字符都将被正则表达式引擎忽略(直至发现匹配项 <as far as matching is concerned>)。

也可以通过嵌入式标志表达式 (?x) 来启用注释模式。

下面的例子示范的是在匹配美国邮政编码的过程中，如何通过设置 Pattern.COMMENTS 标志来使用注释。

试一试：使用 COMMENTS 标志

(1) 在文本编辑器中输入下列代码：

```
import java.util.regex.*;

public class MatchZipComments{
  public static void main(String args[])
  throws Exception{

  String myTestString = "12345-1234 23456 45678 01234-1234";

  // Attempt to match US Zip codes.
  // The pattern matches five numeric digits followed by a hyphen followed by four
numeric digits.
  String myRegex =
   "\\d{5} " +
   "# Matches five numeric digits" +
   "\n(-\\d{4})* " +
   "# Matches four numeric digits and a hyphen, all of which are optional";

  Pattern myPattern = Pattern.compile(myRegex, Pattern.COMMENTS);

  Matcher myMatcher = myPattern.matcher(myTestString);
  String myMatch = "\";

  System.out.println("The test string was '" + myTestString + "'.");
  System.out.println("The pattern was '\\d{5}-\\d{4}'.");
  while (myMatcher.find())
        {
         myMatch = myMatcher.group();
         System.out.println("A match '" + myMatch + "'was found.");
        } // end while

         if (myMatch == ""){
```

```
            System.out.println("There were no matches.");
          } // end if
    } // end main()
}
```

(2) 将代码保存为 MatchZipComments.java。在命令行中输入 javac MatchZipComments.java 并回车编译代码。

(3) 运行代码。在命令行中输入 java MatchZipComments 并回车，观察如图 25-4 所示的结果。

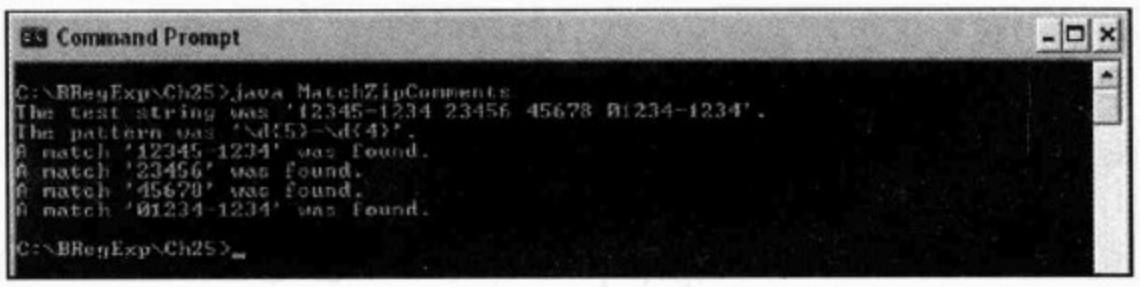


图 25-4

工作原理

将一个包含四个字符序列的可能是美国邮政编码的字符串指定给变量 myTestString：

```
String myTestString = "12345-1234 23456 45678 01234-1234";
```

可以使用常规的 Java 注释来说明这个正则表达式的用途：

```
// Attempt to match US Zip codes(尝试匹配美国邮政编码).
```

类似地，也可以使用常规的 Java 注释来说明模式的构成：

```
// The pattern matches five numeric digits followed by a hyphen followed
by four numeric digits(该模式匹配五个数字后跟一个连字符，再后跟四个数字).
```

因为下面语句中设置了 Pattern.COMMENTS 标志，所以 myRegex 变量的值可以跨行编写，而且注释与正则表达式模式的组件被混合在一起了。但注释都是以 # 字符开头：

```
String myRegex =
 "\\d{5} " +
 "# Matches five numeric digits" +
 "\n(-\\d{4})* " +
 "# Matches four numeric digits and a hyphen, all of which are optional";
```

在将 Pattern 类的 compile()方法的返回值指定给变量 myPattern 时，同时为 compile()方法传递了第二个参数 Pattern.COMMENTS，以设置 COMMENTS 标志。在设置了 COMMENTS 标志后，模式中的空白符将被忽略，从 # 字符开始到下一个行终止符前的字符都将被视为注释：

```
Pattern myPattern = Pattern.compile(myRegex, Pattern.COMMENTS);
```

匹配过程发生在以变量 myTestString 作为参数调用 myPattern 对象的 matcher() 方法时：

```
Matcher myMatcher = myPattern.matcher(myTestString);
```

在变量 myTestString 中有四个匹配项。其中，当可选的组件 (-\d{4})* 匹配一次时，字符序列 12345-1234 和 01234-1234 匹配；而当 (-\d{4})* 匹配零次时，字符序列 23456 和 45678 匹配。

4. DOTALL 标志

默认情况下，.(句点)元字符匹配除行终止符外的任何字符。在 Java 正则表达式中，所谓行终止符(组合)是指下面列出的那些字符(组合)。当设置 DOTALL 标志后，.(句点)元字符匹配所有字符，包括下面这些行终止符：

- \n ——一个新行(换行)符
- \r\n ——一个回车符后跟一个换行符
- \r ——一个回车符(后面没有换行符)
- \u0085 ——一个表示下一行的字符
- \u2028 ——一个行分隔符
- \u2029 ——一个段分隔符

也可以使用嵌入式标志表达式 (?s) 来设置 DOTALL 模式。

5. MULTILINE 标志

默认情况下，位置元字符 ^ 和 $ 分别匹配测试字符序列中第一个字符之前和最后一个字符之后的位置。而当设定了 MULTILINE 模式后，^ 元字符则匹配一行中第一个字符之前的位置，$ 元字符则匹配一行中最后一个字符之后的位置(忽略行终止符)。

也可以通过嵌入式标志表达式 (?m) 来设置 MULTILINE 标志。

6. UNICODE_CASE 标志

设置 CASE_INSENSITIVE 标志可以使匹配美国 ASCII 字符时不区分大小写。而要在匹配其他字符时也不区分大小写，可以设置 UNICODE_CASE 标志。但使用 UNICODE_CASE 标志可能会损失性能，所以建议只在实现必要的正则表达式匹配时才用。

也可以使用嵌入式的标志表达式 (?u) 来指定 UNICODE_CASE 标志。

7. UNIX_LINES 标志

当使用源自 Unix 或其他相关操作系统的多行文本(这些文本中只使用 \n 作为行终止符)时，需要设置 UNIX_LINES 标志。这样，就只有 \n 才能够影响 .(句点)、^(脱字字符)和 $(美元符号)元字符的行为。

也可以使用嵌入式标志表达式 (?d) 来指定 UNIX_LINES 标志。

25.1.5　Pattern 类的方法

表 25-2 中总结了 Pattern 类所特有的方法。其中不包含继承自 Object 类的方法。

表 25-2　Pattern 类特有的方法

方　法	说　明
compile()	这个静态方法用于将一个正则表达式模式编译为一个 Pattern 对象
flags()	返回在 Pattern 对象中设置的标志
matcher()	创建一个根据测试字符串匹配正则表达式的 Matcher 对象
matches()	这个静态方法用于根据一个测试字符串来匹配正则表达式
pattern()	返回据以编译 Pattern 对象的正则表达式模式
split()	在与一个正则表达式匹配的每个实例的位置拆分测试字符串

1. compile()方法

compile()方法有两种形式，这两种形式都是静态方法。一种形式接受一个参数——包含一个正则表达式模式的 String 值。其中任何元字符(如 \d)都必须转义(如写成 \\d)。该方法抛出一个 PatternSyntaxException 异常。

第二种形式接受两个参数：第一个参数是一个包含正则表达式模式的 String 值。其中任何元字符(如 \d)都必须转义(如写成 \\d)。第二个参数是一个用于表示要设置标志的 int 值。如果正则表达式无效，该方法抛出一个 PatternSyntaxException 异常。如果 int 值不对应于一个允许的标志组合，则该方法抛出一个 IllegalArgumentException 异常。

2. flags()方法

flags()不接受参数。它返回与编译 Pattern 对象时设置的标志(如果有)对应的一个 int 值。

3. matcher()方法

matcher()接受一个参数。该参数是一个 CharSequence 值，即测试字符串。如果这个 CharSequence 参数中存在与指定给 Pattern 对象的正则表达式模式匹配的内容，该方法会返回一个新的 Matcher 对象。

4. matches()方法

这个静态方法接受两个参数。第一个参数是包含正则表达式模式的一个 String 值。第二个参数是一个包含测试字符串的 CharSequence 值。matches()会返回一个表示匹配是否成功的布尔值。该方法抛出一个 PatternSyntaxException 异常。

5. pattern()方法

pattern()方法不接受参数，它返回在编译 Pattern 对象时所使用的包含正则表达式模式的 String 值。

6. split()方法

split()方法有两种使用方式。第一种方式以一个 CharSequence 值作为参数，该

CharSequence 值包含一个测试字符串。结果返回一个 String[]数组。作为参数的 CharSequence 值会被在每一个匹配正则表达式模式的匹配项处进行拆分。如果正则表达式模式匹配 CharSequence 的最后一个字符(序列)，那么返回的字符串数组中将不会包含空字符串值。

第二种使用方式与前一种类似，但多了一个 int 值作为它的第二个参数。这个 int 值指定的是对 CharSequence 值进行拆分最多的次数。

25.1.6　Matcher 类

Matcher 类是承担主要工作的类。Matcher 对象负责解释正则表达式并完成匹配操作。

Matcher 类没有公共的构造函数。要创建一个 Matcher 对象，必须在一个 Pattern 对象中调用公共的 matcher()方法(如前所述)：

```
Matcher myMatcher = myPattern.matcher("someString");
```

这个 matcher()方法接受一个参数——测试字符串。

表 25-3 中总结了 Matcher 类的方法。

表 25-3　Matcher 类的方法

方　法	说　明
appendReplacement()	当找到一个匹配项时将一个替换字符串追加到字符串缓冲器中
appendTail()	当找到最后的匹配项时将剩余的字符序列(如果没有找到匹配项，是整个字符序列)追加到字符串缓冲器中
end()	返回最后匹配字符的索引(加 1)
find()	搜索测试字符串中与正则表达式模式匹配的子字符串
group()	不带参数时，返回匹配的子字符串；带一个参数时，返回与指定捕获组对应的子字符串
groupCount()	返回一个正则表达式模式中的捕获组数
lookingAt()	在测试字符串中搜索正则表达式的匹配项
matches()	尝试用正则表达式匹配整个测试字符串
pattern()	返回匹配中使用的 Pattern 对象
replaceAll()	返回一个用替换字符串替换了所有与正则表达式模式匹配的实例后的字符串
replaceFirst()	返回一个用替换字符串替换了第一个与正则表达式模式匹配的实例之后的字符串
reset()	重置一个 Matcher 对象
start()	返回匹配项中第一个字符的索引

1. appendReplacement()方法

appendReplacement()方法与 find()方法和 appendTail()方法共同使用。下面的例子使用 appendReplacement()、find()和 appendTail()方法以及一个 StringBuffer 对象，用字符序列 Moon 来替换字符序列 Star。

试一试：使用 appendReplacement() 方法

(1) 在文本编辑器中输入下列代码：

```
import java.io.*;
import java.util.regex.*;

public final class MatcherMethods {
 private static String myRegex;
 private static String testString;
 private static BufferedReader myBufferedReader;
 private static Pattern myPattern;
 private static Matcher myMatcher;
 public static void main(String[] argv) {
  initResources();
  processTest();
  closeResources(); }

  private static void initResources() {
   try {
    myBufferedReader = new BufferedReader(new FileReader("MatcherMethods.txt"));
    }
  catch (FileNotFoundException fnfe) {
   System.out.println("Cannot locate input file! "+fnfe.getMessage());
   System.exit(0); }
   try { myRegex = myBufferedReader.readLine();
    testString = myBufferedReader.readLine();
    }
    catch (IOException ioe) {}
    myPattern = Pattern.compile(myRegex);
    myMatcher = myPattern.matcher(testString);
    System.out.println("Current myRegex is: "+myRegex);
    System.out.println("Current testString is: "+testString);
    }

    private static void processTest()
    {
     StringBuffer myStringBuffer = new StringBuffer();
     while (myMatcher.find())
     {
     myMatcher.appendReplacement(myStringBuffer, "Moon");
     } // end while loop
     myMatcher.appendTail(myStringBuffer);
     System.out.println();
```

```
            System.out.println(myStringBuffer.toString());
        }

            private static void closeResources()
            {
             try{ myBufferedReader.close();
            }catch(IOException ioe){}
        }
    }
```

(2) 将代码保存为 MatcherMethods.java。在命令行中输入 javac MatcherMethods.java 并按回车编译代码。

(3) 在文本编辑器中输入下列文本：

```
    Star
    Star Training Company is well known for high quality training. Star Training
is currently offering special value training packages.
```

前面的文本除第一个单词外在文件 MatcherMethods.txt 中都处于一行。此处因印刷方便显示为两行。

(4) 将文本保存为 MatcherMethods.txt。

(5) 运行 Java 代码。在命令行中输入 java MatcherMethods 并按回车，然后观察结果。如图 25-5 所示，测试字符串中的每一个 Star 都被替换成了 Moon。

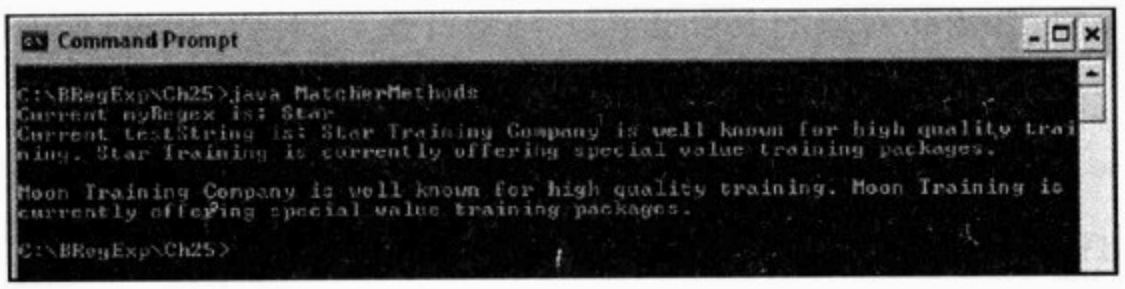


图　25-5

工作原理

在调用 Matcher 对象的方法前，本例的关键部分是 appendReplacement()、find()和 appendTail()方法的用法。

变量 myBufferedReader 用于接受来自文件 MatcherMethods.txt 的内容：

```
    myBufferedReader = new BufferedReader(new FileReader
        ("MatcherMethods.txt"));
```

processTest()方法的代码中使用了 appendReplacment()、find() 和 appendTail()方法。

声明变量 myStringBuffer 并将一个新 StringBuffer 对象的实例指定给它：

```
    StringBuffer myStringBuffer = new StringBuffer();
```

通过 while 循环来处理整个测试字符串。如果找到匹配项，myMatcher.find()方法返回布尔值 true，while 循环中的代码会被执行：

```
while (myMatcher.find())
{
```

在使用 appendReplacement()方法处理测试字符串时，该方法会把字符添加到 StringBuffer 中。如果找到了匹配项，构成匹配项的字符则不会被添加到 StringBuffer 中。反之，会把替换文本添加到字符串缓冲器中。然后，从该匹配项后的位置开始继续查找测试字符串中其余的匹配项：

```
myMatcher.appendReplacement(myStringBuffer, "Moon");
```

当找不到匹配项时，则退出 while 循环：

```
} // end while loop
```

此时，字符串缓冲器中包含的是从测试字符串开头到最后一个匹配项之间的字符。因此，要使用 appendTail() 方法将测试字符串中其余的、非匹配字符添加到 StringBuffer 中：

```
myMatcher.appendTail(myStringBuffer);
```

输出一个空行来分隔原始的测试字符序列和替换后的字符序列(假设找到了匹配项)。如果没有找到匹配项，那么原始的测试字符序列和“替换后”的字符序列将包含相同的字符：

```
System.out.println();
```

调用 myStringBuffer 对象的 toString()方法将替换后的字符串通过 println()方法显示出来：

```
System.out.println(myStringBuffer.toString());
```

2. appendTail()方法

appendTail()方法与 appendReplacement()和 find()方法结合使用。在前一节示范 appendReplacement()方法的例子中也用到过 appendTail()方法。

3. end()方法

end()方法可以不带参数，也可以带一个参数。当不带参数时，end()方法返回匹配的最后一个字符的索引(或位置)加 1 后的值。当带一个参数时，end()方法的这个 int 类型的参数表示的是正则表达式中的一个捕获组。结果返回匹配的组中最后一个字符的索引加 1 的值。

在不带参数或带一个参数的情形下，如果没有发现匹配项或者最近一次匹配失败，end()方法会抛出一个 IllegalStateException 异常。在带一个参数的情形下，如果模式中没有与指定的 int 参数对应的捕获组，那么 end()方法会抛出一个 IndexOutOfBoundsException 异常。

4. find()方法

find()方法用来匹配测试字符串中的下一个子字符串。如果找到一个匹配项，返回布尔值 true。如果没有找到(下一个)匹配项，则返回布尔值 false。如果匹配成功，则可以通过

start()、end()和 group()方法访问有关匹配的信息。

调用 find()方法时可以传递零个或一个参数。在没有参数时，匹配从测试字符串中第一个字符之前的位置开始；或者，如果已找到一个匹配项，则从前一个匹配项最后一个字符之后的位置开始。

在使用一个参数时，该参数应该是一个 int 值，它是表示匹配应该从该处开始的一个索引值，Matcher 对象会被重置，而索引则从测试字符串开始处计算。如果 int 值大于测试字符串的长度，find()方法会抛出一个 IndexOutOfBoundsException 异常。

5. group()方法

group()方法可以不接受参数或者接受一个参数。当不为其传递参数时，group()方法返回由上一次匹配操作找到的匹配项。返回的值是一个 String 值。在实践中，如果模式指定的字符或者元字符都是可选的话，group()方法可能会返回一个空字符串。如果匹配尚未进行过或者上一次匹配失败，group()方法会抛出一个 IllegalStateException 异常。

当为其传递一个参数时，该参数是一个 int 值，表示的是上一次匹配中包含的捕获组。此时，group()方法会返回由相应的编号组捕获的子字符串。如果匹配尚未进行过或者上一次匹配失败，它会抛出一个 IllegalStateException 异常。如果作为参数的 int 值没有对应的捕获组，则会抛出一个 IndexOutOfBoundsException 异常。

6. groupCount()方法

groupCount()方法不接受参数，它返回一个 int 值。这个返回的值表示正则表达式中捕获组的数量，但不包括第零个组——该组对应着整个匹配项。

7. lookingAt()方法

lookingAt()方法会尝试在测试字符串中查找一个正则表达式的匹配项。匹配从字符串中第一个字符之前的位置开始。lookingAt()方法不接受参数。如果找到了一个匹配项，就可以通过 Matcher 对象的 start()、end()和 group()方法访问该匹配项的有关信息。如果测试字符串中存在与正则表达式模式匹配的字符序列，lookingAt()方法会返回一个布尔值 true。

下面的例子会尝试将命令行中输入的一个名字与一个模式进行匹配，这个模式会匹配一个单词后跟一个逗号，后跟一个或多个空格符，再后跟另一个单词的字符序列。也就是说，该模式会匹配以“姓, 名”这样的格式输入的名字。

试一试：使用 lookingAt() 方法

(1) 在文本编辑器中输入下列代码：

```
import java.util.regex.*;

public class lookingAt{
 public static void main(String args[]){
  isMatchPresent(args[0]);
 } // end main()
public static boolean isMatchPresent(String testString){
```

```
    boolean testResult = false;
    String LastNameFirstName = "\\w+,\\s+\\w+";

    Pattern myPattern = Pattern.compile(LastNameFirstName);
    Matcher myMatcher = myPattern.matcher(testString);
    testResult = myMatcher.lookingAt();
    String matchIs = myMatcher.group();

    System.out.println("The test string is: " + testString);
    System.out.println("It is " + testString.length() + " characters long.");

    if (testResult){
     System.out.println("There was a match: " + myMatcher.group() );
     System.out.println("It started at: " + myMatcher.start() );
     System.out.println("It ended at: " + myMatcher.end() );
    }
    else
    {
     System.out.println("No match was found.");
    }
   return testResult;
   } // end isMatchPresent()
}
```

(2) 将代码保存为 lookingAt.java。在命令行中输入 javac lookingAt.java 并按回车键编译代码。

(3) 运行代码。在命令行中输入 java lookingAt "Smith, John" 并按回车。别忘了在逗号后面插入一个空格符，否则会看到错误信息。

(4) 观察显示的结果。然后再次运行代码。

(5) 在命令行中，输入 java lookingAt "Smith, John James" 然后按回车并观察结果。如图 25-6 所示，第 4 步后的结果仍显示在命令行窗口的上方。

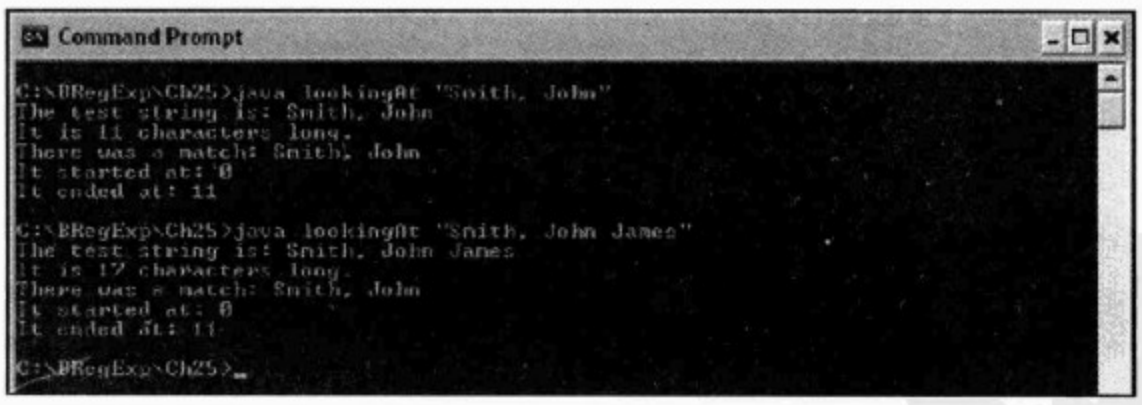


图 25-6

工作原理

本例捕获并处理一个在命令行中输入的字符串参数。同以前一样，main()方法接受 String 对象数组作为参数。而在 main()内部，则会将其中的第一个字符串参数——args[0]，作为 isMatchPresent()方法的参数：

```
public static void main(String args[]){
isMatchPresent(args[0]);
} // end main()
```

isMatchPresent()方法使用了 Matcher 类中的一些方法。

正如该方法的声明所示，传递给 isMatchPresent()方法的 rgs[0]参数，在 isMatchPresent()方法内部是通过 testString 来引用的：

```
public static boolean isMatchPresent(String testString){
```

首先，声明一个布尔变量 testResult 并给它赋默认值 false：

```
boolean testResult = false;
```

然后，将要匹配的模式指定给字符串变量 LastNameFirstName。其中的元字符 \w 和 \s 都要写成 \\w 和 \\s：

```
String LastNameFirstName = "\\w+,\\s+\\w+";
```

使用 Pattern 类的 compile()方法创建一个 Pattern 对象并将其指定给变量 myPattern。然后，使用 myPattern 的 matcher()方法创建一个 Matcher 对象——myMatcher：

```
Pattern myPattern = Pattern.compile(LastNameFirstName);
Matcher myMatcher = myPattern.matcher(testString);
```

将 lookingAt()方法返回的布尔值指定给变量 testResult(该变量的值是以前设置的 false。如果存在匹配项，则该变量的值变成 true)：

```
testResult = myMatcher.lookingAt();
```

将 myMatcher.group()返回的值指定给变量 matchIs。如果存在一个匹配项，该匹配项会被保存在 matchIs 中：

```
String matchIs = myMatcher.group();
```

显示原始字符串及通过 String 类的 length()方法取得的该字符串的长度值。当在命令行中输入的参数是 Smith, John 时，其长度为 11 字符。当输入的参数是 Smith, John James 时，长度是 17 字符：

```
System.out.println("The test string is: " + testString);
System.out.println("It is " + testString.length() + " characters long.");
```

然后，if 语句根据变量 testResult 的值决定如何显示有关匹配的信息。如果显示的是一个匹配项的信息，那么 lookingAt()方法返回的是布尔值 true：

```
if (testResult){
```

显示匹配项使用的是 Matcher 类的 group()方法返回的值。对于两次在命令行中输入的参数—— Smith, John 和 Smith, John James—— 来说，返回的结果是一样的。因为模式 \w+,\s+\w+ 匹配到 John 中最后的 n 就停止了。

```
System.out.println("There was a match: " + myMatcher.group() );
```

Matcher 类的 start()方法返回匹配项中第一个字符的位置：

```
System.out.println("It started at: " + myMatcher.start() );
```

Matcher 类的 end()方法返回匹配项中最后一个字符的位置加 1 的值：

```
System.out.println("It ended at: " + myMatcher.end() );
}
```

如果变量 testResult 的值是 false，则会显示下列信息：

```
else
{
 System.out.println("No match was found.");
}
return testResult;
```

8. matches()方法

matches()方法会尝试将正则表达式模式与整个测试字符串进行匹配。如果整个测试字符串与正则表达式模式匹配，该方法会返回 true。

9. pattern()方法

pattern()方法不接受参数。它返回一个 Pattern 对象。这个返回的 Pattern 对象中包含着 Matcher 对象在匹配中使用的正则表达式模式。

10. replaceAll()方法

replaceAll()方法会用指定的替换字符串替换测试字符串中所有与正则表达式模式匹配的字符序列。替换字符串是 replaceAll()方法唯一的参数。replaceAll()方法的返回值是一个 String。

试一试：使用 replaceAll() 方法

(1) 在文本编辑器中输入下列代码：

```
import java.util.regex.*;

public class replaceAll{
 public static void main(String args[]){
  myReplace(args[0]);
} // end main()

public static boolean myReplace(String testString){
 String myMatch = "Star";

  Pattern myPattern = Pattern.compile(myMatch);
  Matcher myMatcher = myPattern.matcher(testString);
```

```
    String testResult = myMatcher.replaceAll("Moon");

    System.out.println("The test string is: \n'" + testString + "'.");
    System.out.println();

   if (testResult.length() > 0)
    {
     System.out.println("After replacement the string is: \n'" + testResult + "'." );
    }
    else
    {
      System.out.println("No match was found.");
    }
   return true;
   } // end myReplace()
}
```

(2) 将代码保存为 replaceAll.java。然后在命令行中输入 javac replaceAll.java，按回车编译代码。

(3) 运行代码。在命令行中，输入 java replaceAll "Star training is great. Star Training Company is well known." 后按回车并观察结果。如图 25-7 所示，每个字符序列 Star 都被 Moon 所替换。

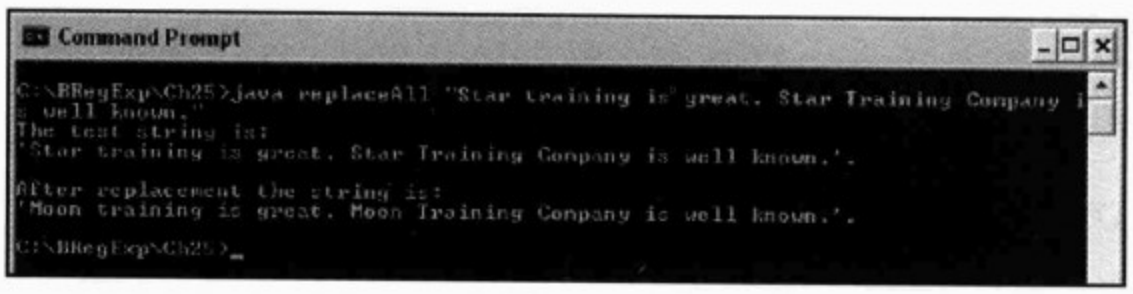


图　25-7

工作原理

在 main()方法中调用 myReplace()方法时，将命令行中输入的字符串传递给它作为参数：

```
myReplace(args[0]);
```

在 myReplace()方法中，将直接量模式 Star 指定给变量 myMatch：

```
String myMatch = "Star";
```

然后分别创建 Pattern 和 Matcher 对象：

```
Pattern myPattern = Pattern.compile(myMatch);
Matcher myMatcher = myPattern.matcher(testString);
```

在调用 Matcher 对象的 replaceAll()方法时为其传递替换字符串 Moon 作为参数，并将替换后的结果指定给变量 testResult：

```
String testResult = myMatcher.replaceAll("Moon");
```

显示原始的测试字符串及一个作为分隔的空白行：

```
System.out.println("The test string is: \n'" + testString + "'.");
System.out.println();
```

String 类的 length()方法用来确定变量 testResult 值的长度。如果存在一个匹配项，那么 testResult 的值将大于零个字符。因此，if 语句的测试返回 true，进而显示替换字符串的信息。

```
if (testResult.length() > 0)
 {
 System.out.println("After replacement the string is: \n'" + testResult + "'." );
}
```

如果 testResult 的长度是零个字符，则显示未找到匹配项的信息：

```
else
{
  System.out.println("No match was found.");
}
return true;
} // end myReplace()
```

11. replaceFirst()方法

replaceFirst()方法会用指定的替换字符串替换测试字符串中的第一个匹配项。replaceFirst()方法唯一的参数就是替换字符串。该方法的返回值是一个 String。

12. reset()方法

reset()方法用于重置 Matcher 对象的状态信息。该方法可以不带参数，也可以带一个参数。当不给 reset()方法传递参数时，会重置状态信息。随后的任何匹配都从测试字符串第一个字符之前的位置开始。测试字符串保持与以前相同。返回的值是一个 Matcher 对象。

当给 reset()方法传递一个参数时，会重置状态信息。参数是一个 String，在重置后，这个字符串参数就成为任何后续匹配的测试字符串。它返回的值是一个 Matcher 对象。

13. start()方法

start()方法不接受参数。它返回一个表示最近一次匹配项中第一个字符索引的 int 值。

如果匹配尚未进行过或者最近一次匹配失败，start()方法会抛出一个 IllegalStateException 异常。

25.1.7 PatternSyntaxException 类

PatternSyntaxException 对象是一个表示正则表达式模式语法中存在错误的未经检查的异常。

PatternSyntaxException 类拥有一些可以访问异常信息的方法。表 25-4 中总结了 PatternSyntaxException 类中的这些方法。

表 25-4　PatternSyntaxException 类中的方法

方　法	说　明
getDescription()	取得错误描述信息
getIndex()	取得模式中的错误索引
getMessage()	返回一个包含错误描述、模式及索引的多行字符串
getPattern()	取得导致错误的正则表达式模式

25.2　java.util.regex 包中支持的元字符

java.util.regex 包中支持的元字符非常广泛，这些元字符将在本节中通过几个表格来简单介绍。其中一些元字符和字符类将在本节后面通过例子更加详细地介绍。

表 25-5 中总结了 Java 的 java.util.regex 包中所支持的元字符。java.util.regex 包中支持的 POSIX 元字符将在本章稍后的小节中介绍。

表 25-5　java.util.regex 包支持的元字符

元　字　符	说　明
.(句点字符)	匹配任何字符。可匹配或不匹配行终止符
\d	匹配一个数字。相当于字符类 [0-9]
\D	匹配一个非数字。相当于取反字符类 [^0-9]
\s	匹配一个空白符
\S	匹配一个非空白符
\w	匹配一个单词字符。相当于字符类 [A-Za-z0-9_]
\W	匹配一个非单词字符。相当于取反字符类 [^A-Za-z0-9_]
[...]	字符类。匹配位于方括号中的任何一个字符
[a-d[w-z]]	两个字符类的联合。相当于 [a-dw-z]
[a-m[h-z]]	两个字符类的交集。相当于 [h-m]
[a-z[^h-m]]	一个字符类与另一个取反的字符类的交集。整体效果是从一个字符类中减去另一个字符类。相当于 [a-gn-z]
[^...]	取反的字符类。匹配任何不包含在方括号内的一个字符

25.2.1　使用\d 元字符

\d 元字符匹配一个数字。下面的例子尝试匹配一个美国邮政编码。通过简单的模式 \d{5}-\d{4}* 能够匹配简短的及扩展的美国邮政编码。

试一试：使用 \d 元字符

(1) 在文本编辑器中输入下列代码：

```
import java.util.regex.*;

public class MatchZip{
  public static void main(String args[])
  throws Exception{
  String myTestString = "12345-1234 23456 45678 01234-1234";

  // Attempt to match US Zip codes.
  // The pattern matches five numeric digits followed by a hyphen followed by four
numeric digits.
  String myRegex = "\\d{5}(-\\d{4})*";

  Pattern myPattern = Pattern.compile(myRegex);

  Matcher myMatcher = myPattern.matcher(myTestString);
  String myMatch = "";

  System.out.println("The test string was '" + myTestString + "'.");
System.out.println("The pattern was '" + myRegex + "'.");
  while (myMatcher.find())
        {
        myMatch = myMatcher.group();
        System.out.println("The match '" + myMatch + "' was found.");
        } // end while
        if (myMatch == ""){
        System.out.println("There were no matches.");
        } // end if
  } // end main()
}
```

(2) 将代码保存为 MatchZip.java。

(3) 在命令行中输入 javac MatchZip.java，并按回车来编译代码。

(4) 在命令行中输入 java MatchZip，并按回车来运行代码，然后观察如图 25-8 所示的结果。

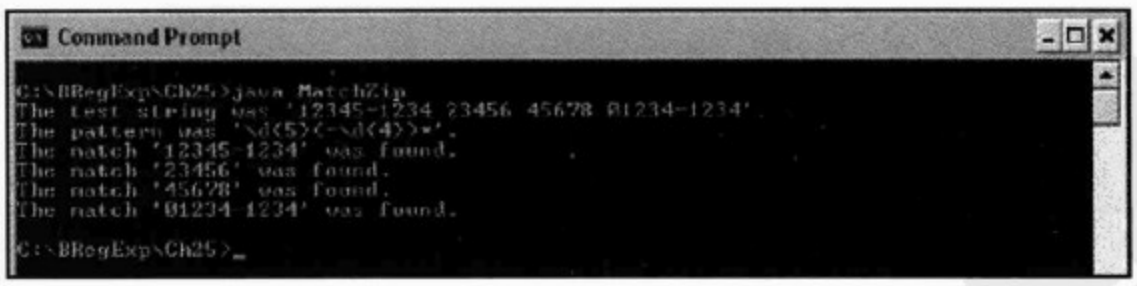


图 25-8

工作原理

将测试字符串指定给变量 myTestString：

```
String myTestString = "12345-1234 23456 45678 01234-1234";
```

将一个能匹配美国邮政编码的模式指定给变量 myRegex：

```
String myRegex = "\\d{5}(-\\d{4})*";
```

通过变量 myRegex 和 myTestString 创建一个 Pattern 对象——myPattern 和一个 Matcher 对象——myMatcher：

```
Pattern myPattern = Pattern.compile(myRegex);
Matcher myMatcher = myPattern.matcher(myTestString);
```

将一个空字符串指定给变量 myMatch，它在后面将用于测试是否存在匹配项。如果变量 myMatch 仍然是空字符串，则说明没有匹配项：

```
String myMatch = "";
```

显示测试字符串 myTestString 的原始值和正则表达式模式 myRegex 的值：

```
System.out.println("The test string was '" + myTestString + "'.");
System.out.println("The pattern was '" + myRegex + "'.");
```

while 循环中测试 find() 方法的返回值用来确定是否找到匹配项。如果根本不存在匹配项，那么 while 循环中的代码永远不会被执行。如果存在匹配项，则会显示每个匹配项的值。在显示最后一个匹配项之后，再测试 myMatcher.find()将返回 false，退出 while 循环：

```
while (myMatcher.find())
        {
         myMatch = myMatcher.group();
         System.out.println("The match '" + myMatch + "' was found.");
        } // end while
```

如果 myMatch 仍然是空字符串，则表示没有匹配项，因此可以放心地输出一个表示没有匹配项的信息。因为如果有一个匹配项，那么变量 myMatch 的值会是一个不为空的字符串，那么就不会显示下列信息：

```
if (myMatch == ""){
System.out.println("There were no matches.");
} // end if
```

25.2.2　字符类

java.util.regex 包中所支持的字符类包括常规字符类、取反字符类和字符类中的范围(在本书介绍性的章节中讨论过)。除此之外，java.util.regex 包还支持许多正则表达式实现中都不支持的一些有用功能。

下面的例子示范基本的字符类功能。

试一试：使用字符类

(1) 在文本编辑器中输入下列代码：

```
import java.util.regex.*;
```

```
public class CharClass{
 public static void main(String args[]){
  findMatches(args[0]);
 } // end main()
public static boolean findMatches(String testString){
 String myRegex = "[A-D]\\d";

 Pattern myPattern = Pattern.compile(myRegex);
 Matcher myMatcher = myPattern.matcher(testString);
 String myMatch = null;

 System.out.println("The test string was: " + testString);
 System.out.println("The regular expression pattern was:" + myRegex);
 while (myMatcher.find())
        {
         myMatch = myMatcher.group();
         System.out.println("Match found: " + myMatch);
        } // end while

         if (myMatch == null){
         System.out.println("There were no matches.");
         } // end if

 return true;
 } // findMatches()
}
```

(2) 将代码保存为 CharClass.java。

(3) 编译代码。在命令行中输入 javac CharClass.java 并按回车。

(4) 运行代码。在命令行中输入 java CharClass "A1 B2 C3 D9 E8 F3 G5" 并按回车，然后观察结果。如图 25-9 所示，字符序列 E8、F3 和 G5 不匹配。

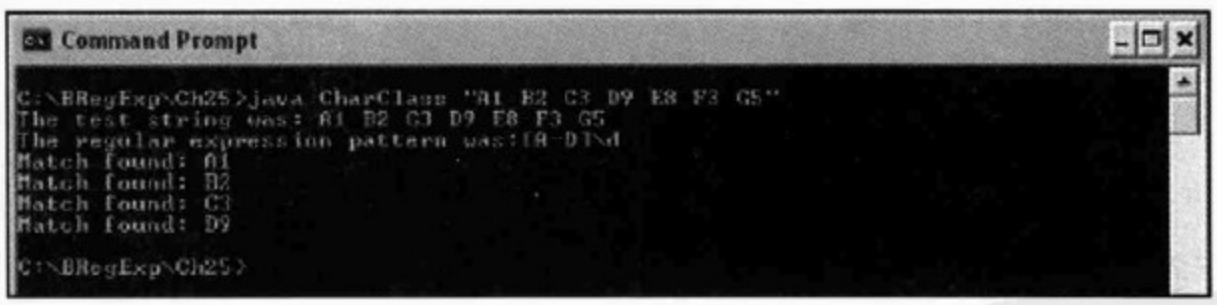


图 25-9

工作原理

将模式 [A-D]\d 指定给变量 myRegex：

```
String myRegex = "[A-D]\\d";
```

在命令行中输入的测试字符串 A1 B2 C3 D9 E8 F3 G5 将指定给 args[0] 数组元素，该元素被用做 findMatches()方法的参数：

```
findMatches(args[0]);
```

然后，通过 while 循环来显示找到的每个匹配项。当 while 语句测试返回 true 时，while 循环中的代码就会被执行。当没有匹配项时，find()方法将返回布尔值 false，于是，退出 while 循环：

```
while (myMatcher.find())
       {
       myMatch = myMatcher.group();
       System.out.println("Match found: " + myMatch);
       } // end while
```

Java 支持字符类的联合，下面就来看一个例子。

试一试：使用字符类的联合

(1) 在文本编辑器中输入下列代码：

```
import java.util.regex.*;

public class CharClassUnion{
 public static void main(String args[]){
  findMatches(args[0]);
 } // end main()

 public static boolean findMatches(String testString){
  String myRegex = "[A-D[H-M]]\\d";
  Pattern myPattern = Pattern.compile(myRegex);
  Matcher myMatcher = myPattern.matcher(testString);
  String myMatch = null;

  System.out.println("The test string was: " + testString);
  System.out.println("The regular expression pattern was:" + myRegex);
  while (myMatcher.find())
         {
          myMatch = myMatcher.group();
          System.out.println("Match found: " + myMatch);
         } // end while
          if (myMatch == null){
           System.out.println("There were no matches.");
         } // end if
  return true;
  } // findMatches()
}
```

(2) 将代码保存为 CharClassUnion.java。

(3) 编译代码。在命令行中输入 javac CharClassUnion.java 并按回车。

(4) 运行代码。在命令行中输入 java CharClassUnion "A1 B2 C3 D4 E5 H9 I2 J3 N4" 并按回车。然后观察结果。如图 25-10 所示，字符序列 E5 和 N4 不匹配。

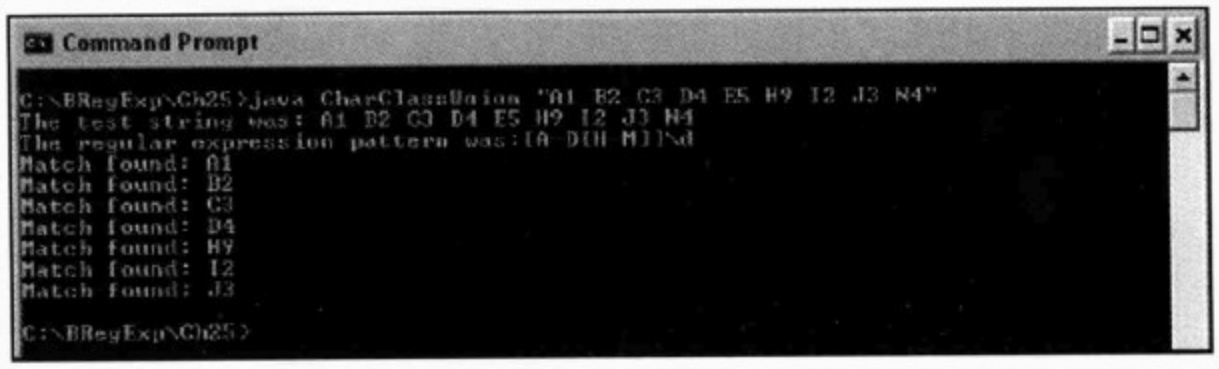


图 25-10

工作原理

指定给变量 myRegex 的字符串是两个字符类 [A-D] 和 [H-M] 的联合。联合操作符是隐含的。这个字符类等价于 [A-DH-M]：

```
String myRegex = "[A-D[H-M]]\\d";
```

如前所述，通过 while 循环来显示每一个匹配项。

如果字符类 [A-DH-M] 中的一个字母字符与一个数字在一起，则存在匹配项。

试一试：使用字符类的交集

(1) 在文本编辑器中输入下列代码：

```
import java.util.regex.*;

public class CharClassSubtraction{
public static void main(String args[]){
String testString = args[0];
 findMatches(testString);
} // end main()

public static boolean findMatches(String testString){
 String myRegex = "[A-Z&&[^H-M]]\\d";
 Pattern myPattern = Pattern.compile(myRegex);
 Matcher myMatcher = myPattern.matcher(testString);
 String myMatch = null;
 System.out.println("The test string was: " + testString);
 System.out.println("The regular expression pattern was: " + myRegex);
 while (myMatcher.find())
        {
         myMatch = myMatcher.group();
         System.out.println("Match found: " + myMatch);
        } // end while

        if (myMatch == null){
         System.out.println("There were no matches.");
        } // end if
 return true;
 } // findMatches()
}
```

(2) 将代码保存为 CharClassSubtraction.java。

(3) 编译代码。在命令行中输入 javac CharClassSubtraction.java 并按回车。

(4) 运行代码。在命令行中输入 java CharClassSubtraction "A1 B2 H3 I2 J4 M5 N6" 并按回车，然后观察结果。结果如图 25-11 所示，字符序列 H3、I2 和 J4 不匹配。

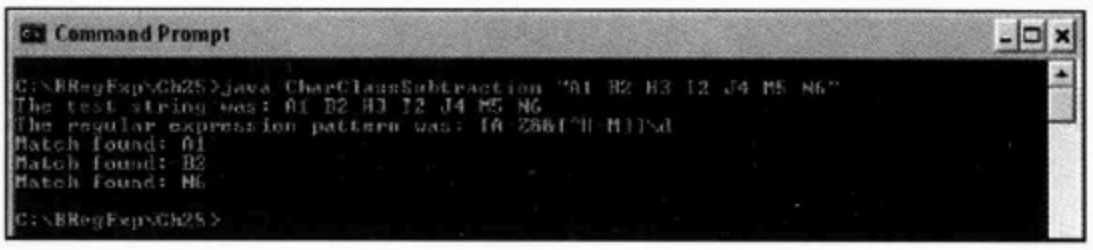


图　25-11

工作原理

在这个例子中，正则表达式是 [A-Z&&[^H-M]]\d。其中的 && 操作符的作用是计算两个字符类([A-Z] 和 [^H-M])的交集。交集是不在 H 到 M 之间的大写字母字符。因此，模式 [A-Z&&[^H-M]] 等价于 [A-GN-Z]：

```
String myRegex = "[A-Z&&[^H-M]]\\d";
```

字符序列 H3、I2 和 J4 匹配失败的原因是从 H 到 M 的字母字符不匹配组合后的字符类。

25.2.3　java.util.regex 包中的 POSIX 字符类

Java 的 java.util.regex 包也支持一些 POSIX 字符类，但所使用的语法与之前看到某些工具(如 OpenOffice.org)中的不同。使用 java.util.regex 所支持的 POSIX 字符类，与在 W3C XML Schema 中使用 Unicode 字符类和字符块的语法有些类似。表 25-6 中列出了 java.util.regex 包支持的 POSIX 字符类。

表 25-6　java.util.regex 包支持的 POSIX 字符类

元 字 符	说　明
\p{Lower}	等价于字符类 [a-z]
\p{Upper}	等价于字符类 [A-Z]
\p{ASCII}	匹配所有 ASCII 字符。相当于 U+0000 到 U+007F
\p{Alpha}	匹配任何字母字符。相当于字符类 [\p{Upper}\p{Lower}] 或 [A-Za-z]
\p{Digit}	等价于字符类 [0-9]
\p{Punct}	等价于字符类 [!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~]
\p{Graph}	匹配可见字符。等价于字符类 [\p{Alpha}\p{Punct}]
\p{Print}	匹配可打印字符。等价于字符类 [\p{Graph}]
\p{Blank}	匹配一个空格符或一个制表符
\p{Cntrl}	匹配一个控制符。等价于字符类 [\x00-\x1F\x7F]
\p{XDigit}	匹配一个十六位数字。等价于字符类 [0-9a-fA-F]
\p{Space}	匹配一个空白符。等价于字符类 [\t\n\x0B\f\r]

25.2.4 Unicode 字符类和字符块

Java 中的字符串都是由 Unicode 字符组成的。其中每个字符都是一个两字节的数。如果不熟悉如何将英文字符以及其他字符映射为 Unicode 编码，那么 Windows Character Map 实用工具是一个好帮手。图 25-12 显示的是一个带有抑扬符号的 e 在 Character Map 中被选中的情形。注意在该窗体左下角显示的该字符的 Unicode 编码值为 U+00EA。而在 Java 中，则应该写成 \u00EA。

举例来说，要匹配拉丁字符块中的字符，可以像模式 \p{InBasicLatin}一样使用小写字母 p。

而要匹配不在拉丁字符块中的字符，可以像模式\P{InBasicLatin}一样使用大写字母 P。

有关 Unicode 的完整信息位于 www.unicode.org 网站中。在本书写作时，Unicode 标准的版本还是 4.0.1。更多有关 Unicode 标准的资料可以在 www.unicode.org/standard/standard.html 中找到。

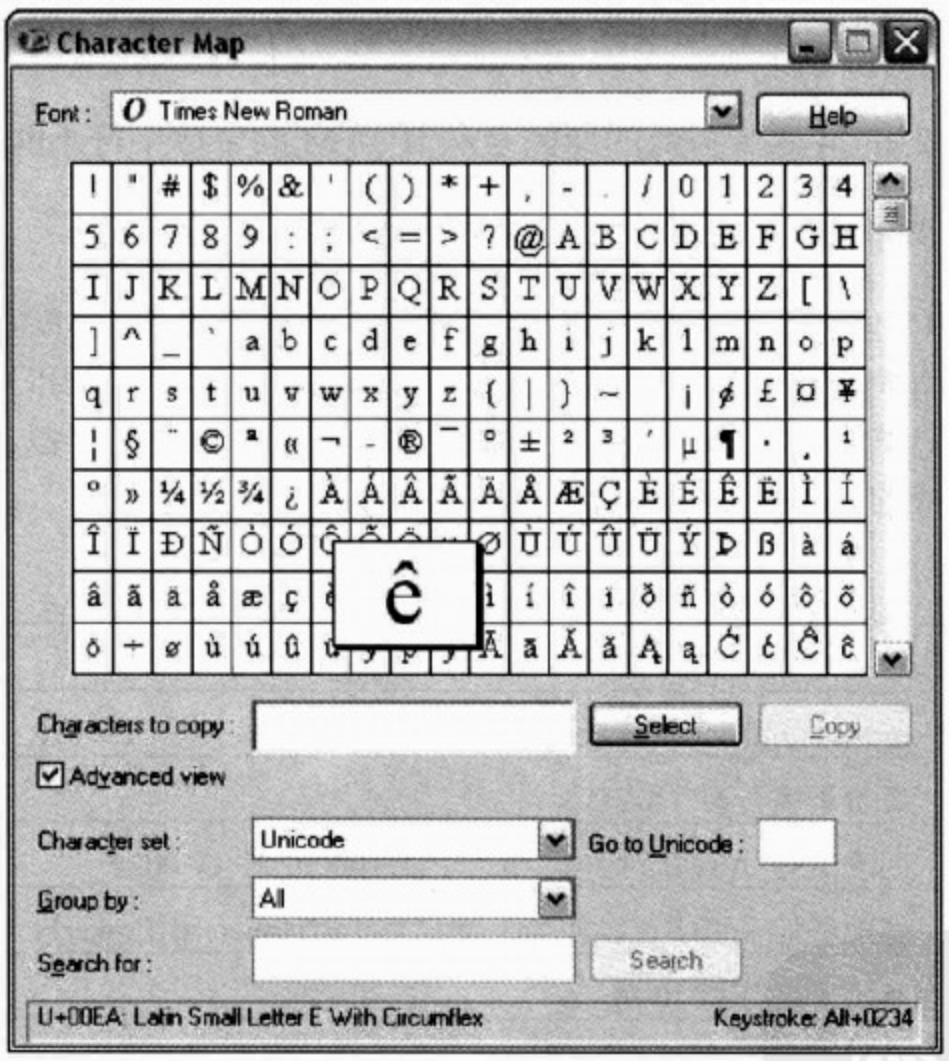


图 25-12

25.2.5 使用转义字符

要匹配特殊目的的正则表达式元字符，必须要进行转义。表 25-7 中列出了 Java 中比较常用的转义字符。

表 25-7　Java 中常用的转义字符

转义字符序列	匹　　配
\\	\(反斜杠)
\(	((左圆括号)
\)	)(右圆括号)
\[	[(左方括号)
\]	](右方括号)
\^	^(脱字字符)；只能在字符类外面使用
\$	$(美元符号)
\?	?(问号)
*	*(星号)
\+	+(加号)
\.	.(句点字符)

还可以通过另外一种方式来匹配元字符而不用对其进行上述转义。可以将元字符或元字符序列包括在两个专门的元字符——\Q (表示一个引用字符的开始)和 \E(表示一个引用字符的结束)——之间。

25.3　使用 String 类的方法

String 类中有一些方法支持使用正则表达式功能。其中的 matches() 方法用于测试在字符串中是否存在一个正则表达式模式的匹配项。而 replaceFirst()和 replaceAll()方法会以指定的替换字符串替换与正则表达式匹配的一个或全部子字符串。String 类中也有一个 replace()方法，但该方法与正则表达式无关，它只能简单地匹配和替换直接量字符串。

表 25-8 中总结了 String 类中支持正则表达式功能的方法及相关信息。当然，String 类中还有很多不使用正则表达式的其他方法(这里不做介绍)。

表 25-8　String 类中支持正则表达式功能的方法及相关信息

方　　法	说　　明
matches()	测试一个字符串中是否包含与给定的正则表达式匹配的内容
replaceFirst()	用指定的替换字符串替换匹配正则表达式模式的第一个子字符串
replaceAll()	用指定的替换字符串替换匹配正则表达式模式的所有子字符串
split()	在与指定的正则表达式匹配的位置处将一个字符串拆分为子字符串

25.3.1　使用 matches()方法

String 类的 matches()方法用于测试整个字符串是否匹配一个正则表达式。matches()方

法接受一个参数，即表示正则表达式模式的 String。

试一试：使用 String 类的 matches() 方法

(1) 在文本编辑器中输入下列代码：

```
import java.util.regex.*;

public class stringMatches{
 public static void main(String args[]){
  findMatch(args[0]);
 } // end main()

 public static boolean findMatch(String testString){
  String myRegex = "\\b[a-z]*hip[a-z]*\\b";
  boolean testResult = testString.matches(myRegex);

  System.out.println("The string was: " + testString);
  System.out.println("The regular expression pattern was:" + myRegex);
  if (testResult)
     {
     System.out.println("There was a match.");
     } // end if
     else
     {
     System.out.println("There was no match.");
     }
     return true;
 } // findMatch()
}
```

(2) 将代码保存为 stringMatches.java。

(3) 编译代码。在命令行中，输入 javac stringMatches.java 并按回车。

(4) 运行代码。在命令行中，输入 java stringMatches "ship" 并按回车，然后观察显示的结果。

(5) 再次运行代码。在命令行中，输入 java stringMatches "The ship was large." 并按回车，观察结果。如图 25-13 所示，字符序列 ship 匹配，但字符序列 The ship was large. 不匹配。

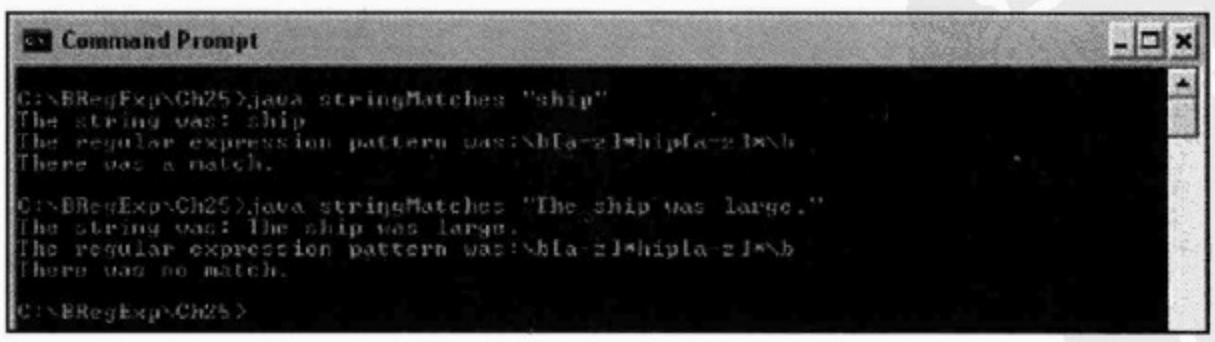


图 25-13

工作原理

指定给 myRegex 变量的正则表达式模式匹配包含字符序列 hip 的任何单词：

```
String myRegex = "\\b[a-z]*hip[a-z]*\\b";
```

String 类的 matches()方法将返回的布尔值指定给变量 testResult。它的参数是指定给变量 myRegex 的正则表达式模式：

```
boolean testResult = testString.matches(myRegex);
```

由于 matches()方法返回一个布尔值，因此可以测试变量 testResult 的值是 true 还是 false。在本例中，testResult 变量的值用于控制显示是否存在匹配项的信息：

```
if (testResult)
        {
        System.out.println("There was a match.");
        } // end if
        else
        {
        System.out.println("There was no match.");
        }
```

因为模式 \b[a-z]*hip[a-z]*\b 是区分大小写的，它会匹配字符序列 ship 但不会匹配 Ship。而且，由于 String 类的 matches() 方法必须匹配整个字符串，所以该模式不会匹配包含 ship 的字符串 The ship was large.。

25.3.2　使用 replaceFirst()方法

String 类的 replaceFirst() 方法接受两个参数。第一个参数是包含正则表达式模式的 String。第二个参数是包含替换字符串的 String。

如果正则表达式模式中存在错误，那么 replaceFirst()方法会抛出一个 PatternSyntax Exception 异常。

下面的例子通过 replaceFirst()方法用字符序列 TWINKLE 来替换字符序列 twinkle。

试一试：使用 String 类的 replaceFirst() 方法

(1) 在文本编辑器中输入下列代码：

```
import java.util.regex.*;

public class stringReplaceFirst{
 public static void main(String args[]){
  myReplaceFirst(args[0]);
} // end main()
public static boolean myReplaceFirst(String testString){
  String myRegex = "twinkle";
  String testResult = testString.replaceFirst(myRegex, "TWINKLE");

  System.out.println("The string was: '" + testString + "'.");
```

```
        System.out.println("The regular expression pattern was: '" + myRegex +"'.");
        System.out.println("After replacement the string was: '" + testResult +"'.");
            return true;
    } // myReplaceFirst()
}
```

(2) 将代码保存为 stringReplaceFirst.java。

(3) 编译代码。在命令行中输入 javac stringReplaceFirst.java 并按回车。

(4) 运行代码。在命令行中输入 java stringReplaceFirst "twinkle, twinkle little star" 并按回车。然后观察如图 25-14 所示的结果。

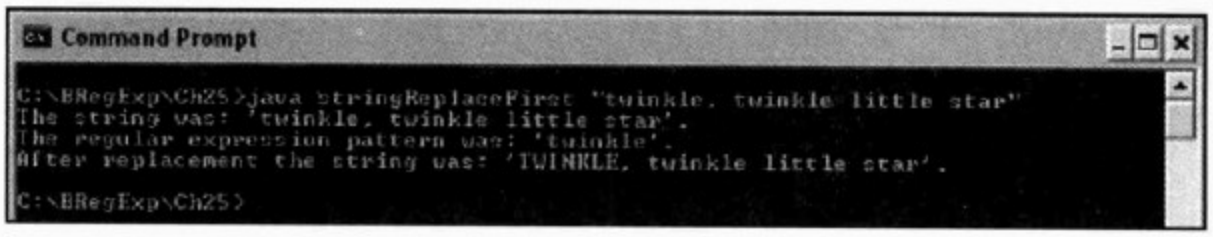


图 25-14

工作原理

将一个简单的直接量模式 twinkle 指定给变量 myRegex：

```
String myRegex = "twinkle";
```

接着，通过 replaceFirst()方法为变量 testResult 指定一个 String 值。除第一个参数正则表达式模式外，该方法接受的第二个参数将字符序列 TWINKLE 指定为替换字符串：

```
String testResult = testString.replaceFirst(myRegex, "TWINKLE");
```

然后，显示字符串、正则表达式和替换后的字符串的值：

```
System.out.println("The string was: '" + testString + "'.");
System.out.println("The regular expression pattern was: '" + myRegex +"'.");
System.out.println("After replacement the string was: '" + testResult +"'.");
```

变量 testString 的值是 twinkle, twinkle little star。twinkle 的第一个实例会被替换。因此，替换后的字符串就是 TWINKLE, twinkle little star。

25.3.3 使用 replaceAll()方法

String 类的 replaceAll()方法会以指定的替换字符串替换一个正则表达式的所有匹配项。replaceAll()方法的第一个参数是一个包含正则表达式模式的 String。第二个参数是一个包含替换字符串的 String。

25.3.4 使用 split()方法

String 类的 split()方法可以接受一个或两个参数。该方法用于将字符串拆分为子字符串。拆分的位置由正则表达式指定。该方法返回一个 String 数组。

当使用一个参数时，这个参数是包含一个正则表达式模式的 String。在模式的每个匹配项处，将测试字符串拆分为子字符串。当使用两个参数时，第一个参数是包含一个正则

表达式的 String，而第二个参数是一个用于指定最多拆分次数的 int 值。在使用第二个参数加以限制的情况下，测试字符串会在每个模式匹配项处被拆分为子字符串。如果正则表达式模式中包含错误，那么 split()方法会抛出一个 PatternSyntaxException 异常。

25.4　练习

下面的练习可以测试对本章中某些内容的理解程度：

1. 请修改 replaceAll.java 代码，使其只替换正则表达式模式的第一个匹配项。
2. 请修改 stringReplaceFirst.java 代码，使其能替换所有字符序列 twinkle 的实例。

第26章

Perl 中的正则表达式

Perl 是一门强大而又有点隐晦的脚本语言，尤其在文本操作、系统管理和动态生成 Web 内容方面。Perl 对于正则表达式的支持是该语言的一个关键特长，为 Perl 强大的文本操作功能奠定良好的基础。

Perl 的语法很简洁，对于稍有或没有 Perl 编程经验的程序员而言，它显得有点神秘莫测、难以捉摸。在没有准备的情况下，同时面对语法简洁而隐晦的 Perl 和正则表达式，可能会令人感到窒息。然而，如果将 Perl 和正则表达式语法分开来对待，那么这种压迫感也将不复存在。学习完本章的内容和例子之后，将在 Perl 中通过正则表达式实现强大的文本操作方面取得实质性的进步。

在本章中将学习如下内容：

- 下载 Perl 并在 Windows 中安装
- 在 Perl 中使用基本的正则表达式
- 使用 Perl 正则表达式操作符
- Perl 支持的元字符
- 使用各种 Perl 支持的元字符
- 在 Perl 中指定并使用正则表达式模式

26.1 下载并安装 Perl

虽然本书主要介绍基于 Windows 平台的应用，但 Perl 能在许多平台中使用。

要获得面向各种操作系统的当前版本 Perl 的一个副本，可以访问 www.perl.com/download.csp 或 http://www.perl.org/get.html，其中列出了当前有效的 Perl 下载包。如果使用的是非 Windows 平台，可以从中选择相应的下载并按照所提供的安装说明安装。

基于 Windows 平台的 Perl 5.8 下载包可以从 www.activestate.com/ActivePerl/下载。ActivePerl 是一款由 ActiveState.com 开发的商业软件，同时也是 Perl.com 所推荐的 Perl 脚本开发环境。在本书写作时，该软件是免费下载的。要下载 ActivePerl 安装程序，首先必须提供一个用户名和一个电子邮件地址。

下载完 ActivePerl MSI 安装程序后，只需简单地双击运行并按照屏幕提示操作即可完成安装。如果使用的是 Windows XP 或 Windows Server 2003，并且拥有管理员权限，安装

过程非常简单。如果使用的是其他版本的 Windows，那么可以在 http://aspn.activestate.com/ASPN/docs/ActivePerl/install.html 中查找相关的安装信息。在安装程序运行的界面中，保持其默认选项即可(除非有其他特殊原因，否则不用修改)，如图 26-1 所示。

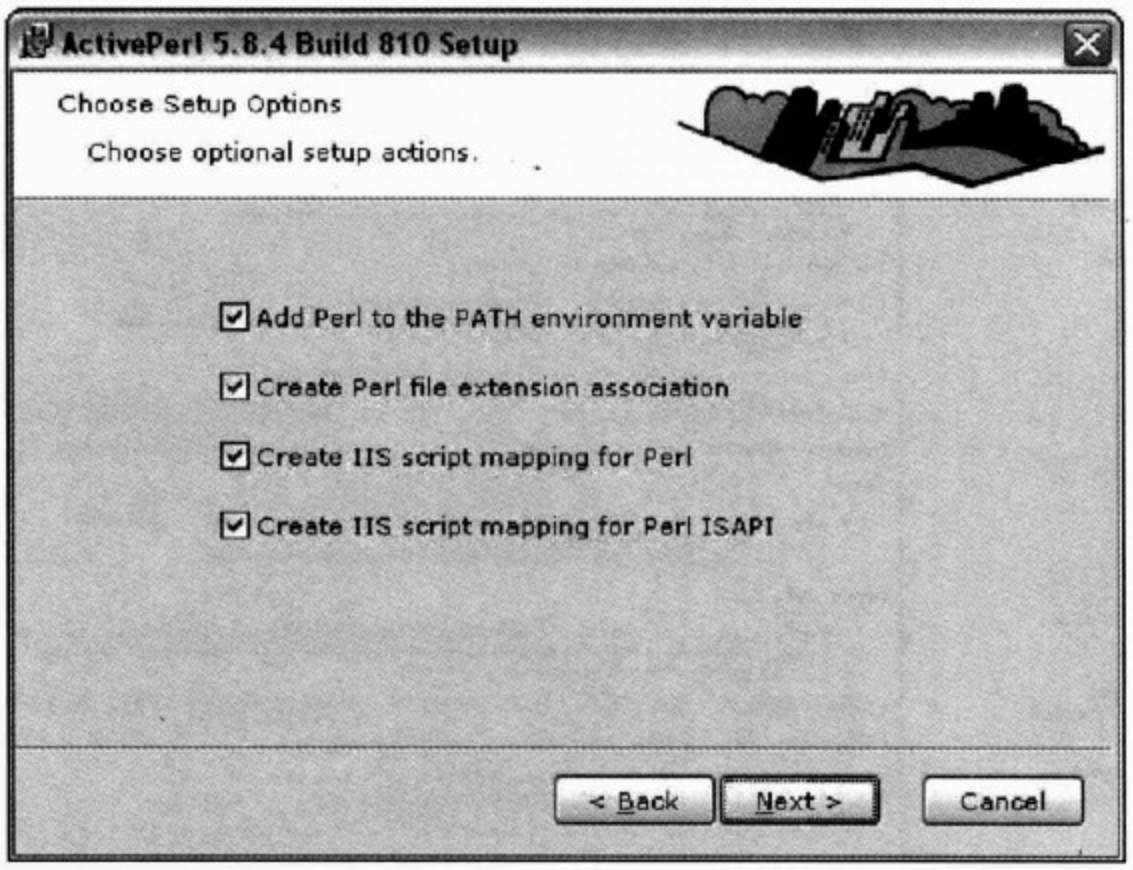


图　26-1

有关安装的说明假设是在一台开发机器中安装 Perl，并仅用于测试目的。如果想在一台产品服务器中安装 Perl，那么最好花些时间理解相关的安全问题。

ActivePerl 将安装在 C:\Perl (安装过程中没有提供安装到其他位置的选择)目录中。如果曾经安装过 Perl 的某个早期版本，那么该版本会被覆盖。除非想单独比较不同版本 ActivePerl 的区别，否则这个默认行为不会导致任何问题。

安装 ActivePerl 的同时也会安装许多 Perl 文档。可以用浏览器打开位于 C:\Perl\html\index.html 中的 ActivePerl User Guide，图 26-2 显示的是在 Firefox 浏览器中打开该指南的界面。该用户指南的左侧窗格中有很多垂直显示的导航信息，可以方便地向下滚动。Perl 核心文档的链接也位于左侧的窗格中。还有一种访问 ActivePerl User Guide 的方式，在 Windows XP 中，选择 Start | All Programs | ActiveState ActivePerl 5.8 | Documentation 后，ActivePerl User Guide 就会在浏览器中打开。

如果喜欢使用 perldoc 实用程序而不是 ActivePerl User Guide 来阅读文档，只需在 Windows 命令行中简单地输入相关的 perldoc 命令即可。因为安装程序已经把必需的信息添加到 PATH 环境变量中了。图 26-3 显示了从 perldoc 中通过 perldoc strict 命令调出的有关 Perlstrict 编译指示的相关信息。此时，要前进一屏，按空格键；要前进一行，按回车键。

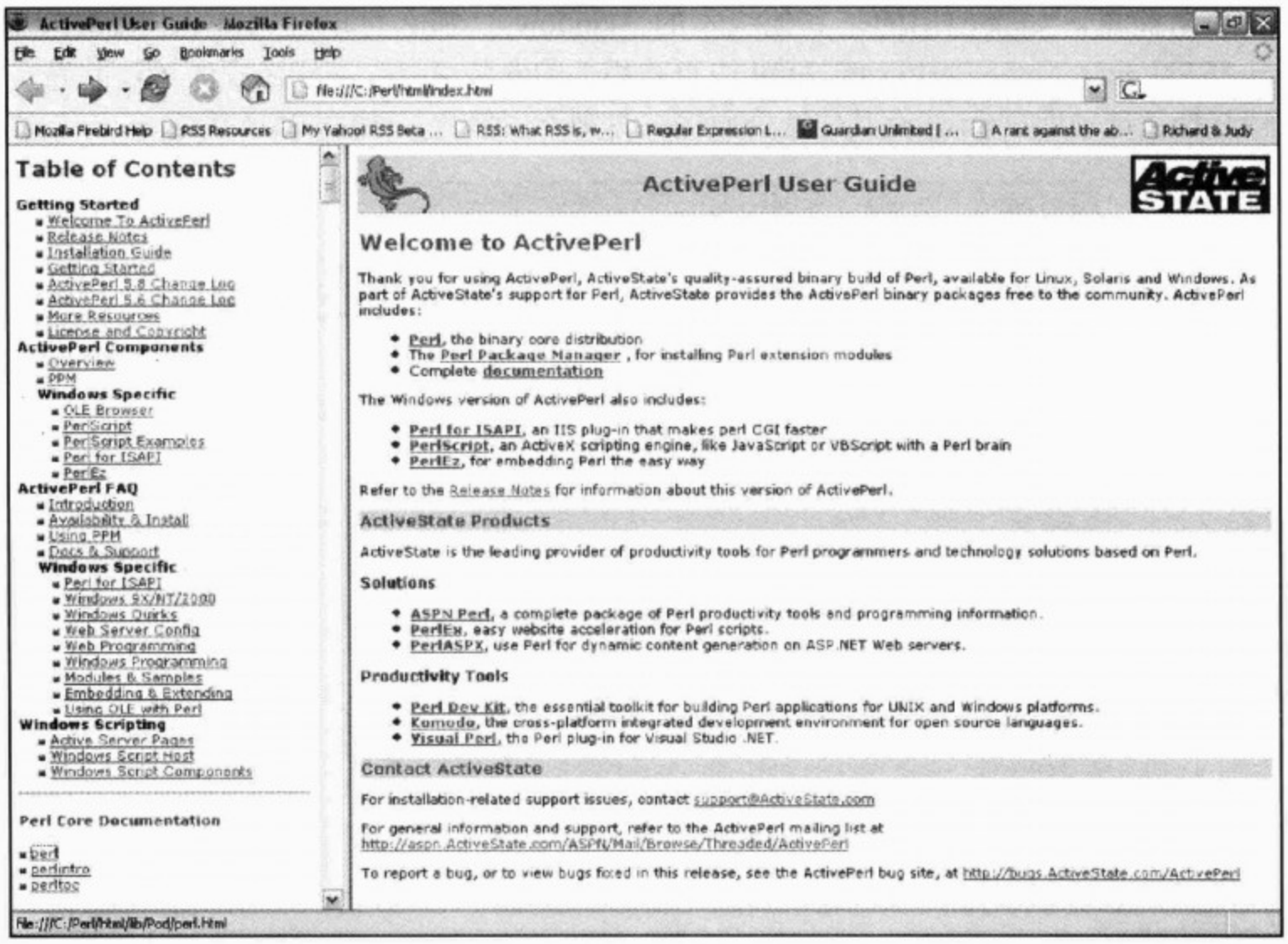


图 26-2

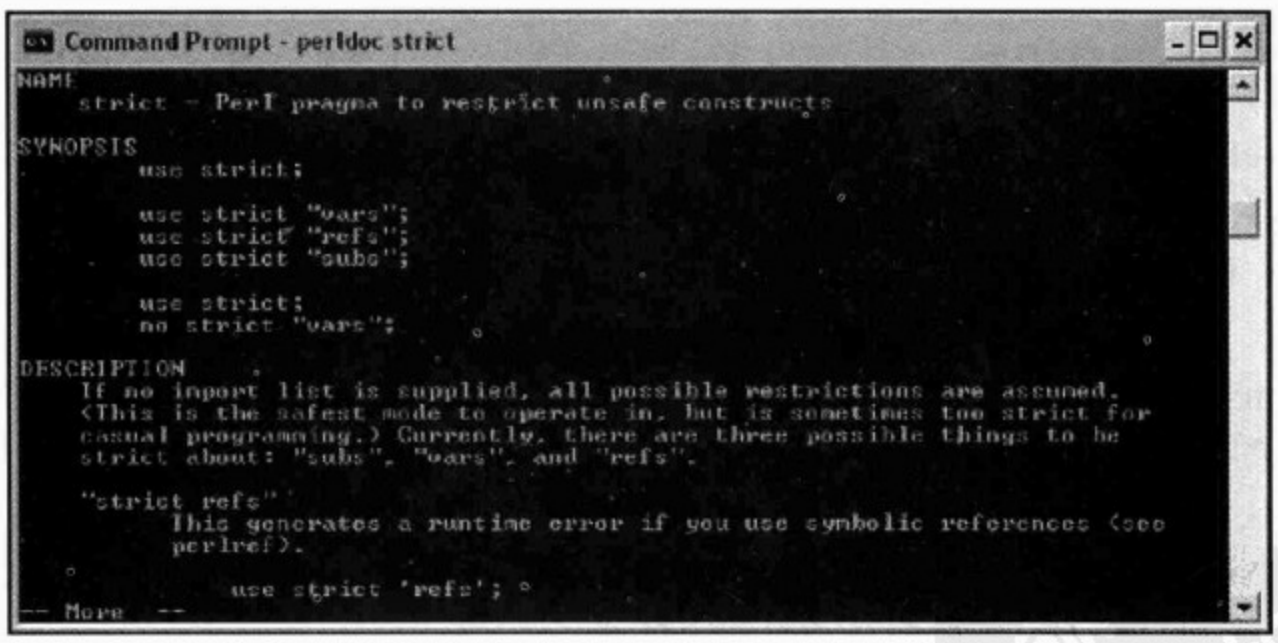


图 26-3

在知道了安装 ActivePerl 后到哪里能找到 Perl 文档以后，现在还需要确认是否正确地安装了 Perl。可以在命令行中输入 perl -v 命令，如果执行该命令后显示出有关安装的 Perl 版本的信息，则说明安装成功。图 26-4 显示的是 ActivePerl 5.8.4 安装后的版本信息。

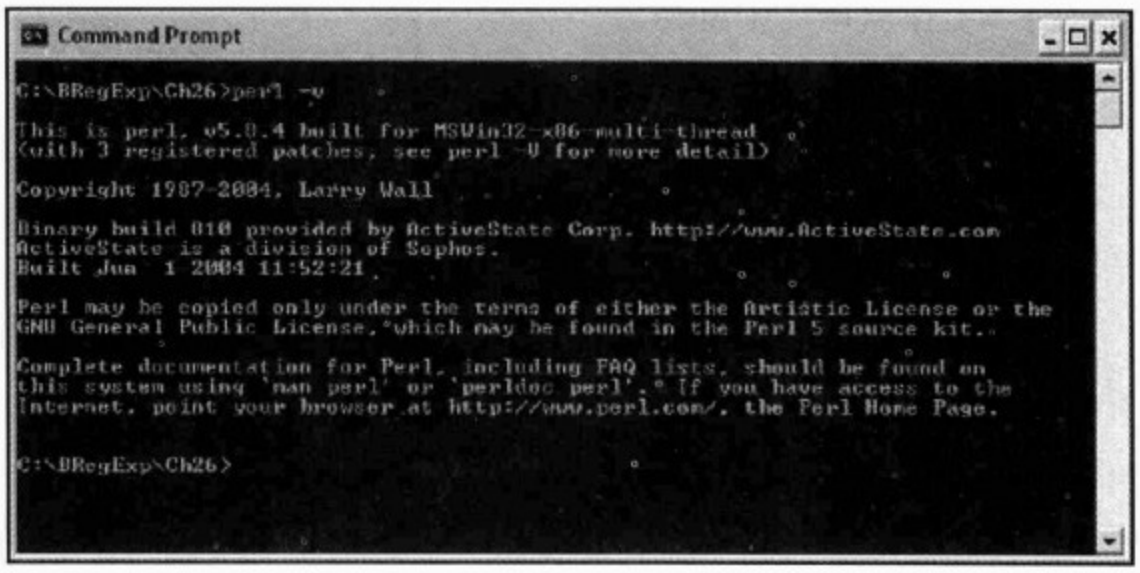


图　26-4

可以在任何文本编辑器或者集成开发环境中编写 Perl 代码。本章中的例子是使用 ActiveState Komodo 3.0 开发环境创建的。可以从 www.activestate.com/Products/ Komodo 下载到一个 Komodo 的限时免费测试版。要下载评估版，必须提供用户名和电子邮件地址。其中电子邮件地址一定要保证正确，因为需要通过一个小程序创建 Komodo 的功能版本，而下载这个小程序的 URL 会发送到你提供的电子邮件地址中。在写作本书时，可以下载到 Komodo 3.0 的 MSI 安装程序。安装 Komodo 是一个遵照屏幕提示就能完成安装的简单过程。

创建一个简单的 Perl 程序

现在可以使用 Komodo 3.0 和 ActivePerl 来创建第一个简单的 Perl 程序了。如果更愿意使用其他文本编辑器，那么假设你熟悉如何在该编译器中输入命令和编辑 Perl 代码。

试一试：创建一个简单的 Perl 程序

(1) 在 Komodo 3.0 中，从 File 菜单中选择 New，然后选择 New File。

(2) 在 New File 对话框(如图 26-5 所示)中，选择 Perl 选项，然后单击 Open 按钮。

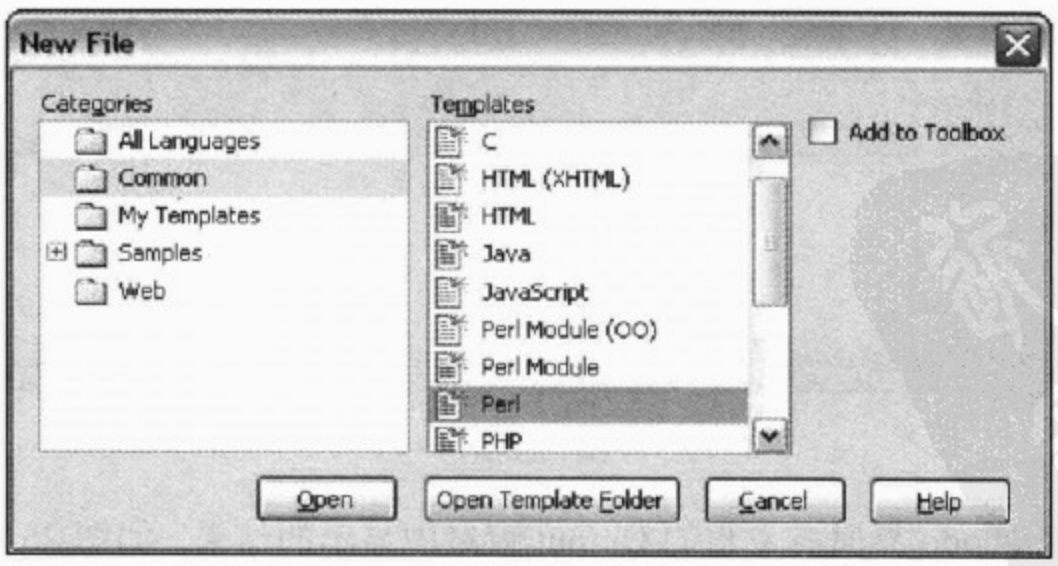


图　26-5

如图 26-6 所示，Komodo 3.0 会打开一个带有简单模板代码的 Perl-1.pl 文件。其中代

码行的含义在后面的“工作原理”中解释。

(3) 在文件中输入下列 Perl 代码：

```
#!/usr/bin/perl -w
use strict;
my $myString = "Hello world!";
print "$myString\n";
print "The preceding line printed the variable \$myString.\n";
print "This program has run without errors.";
```

(4) 将代码保存在 C:\BRegExpr\Ch26\Simple.pl。

(5) 在 Komodo 3.0 中按 F5，然后再按回车键(接受默认的无脚本参数)在调试模式中运行代码。

如果这是在 Komodo 中创建的第一个程序，那么按照前面的指示做就可以了。而假如在 Komodo 中运行过其他代码，可能在按 F5 和回车键中间还需要选择要运行的文件。此时单击 Browse 按钮可以选择要运行的文件。

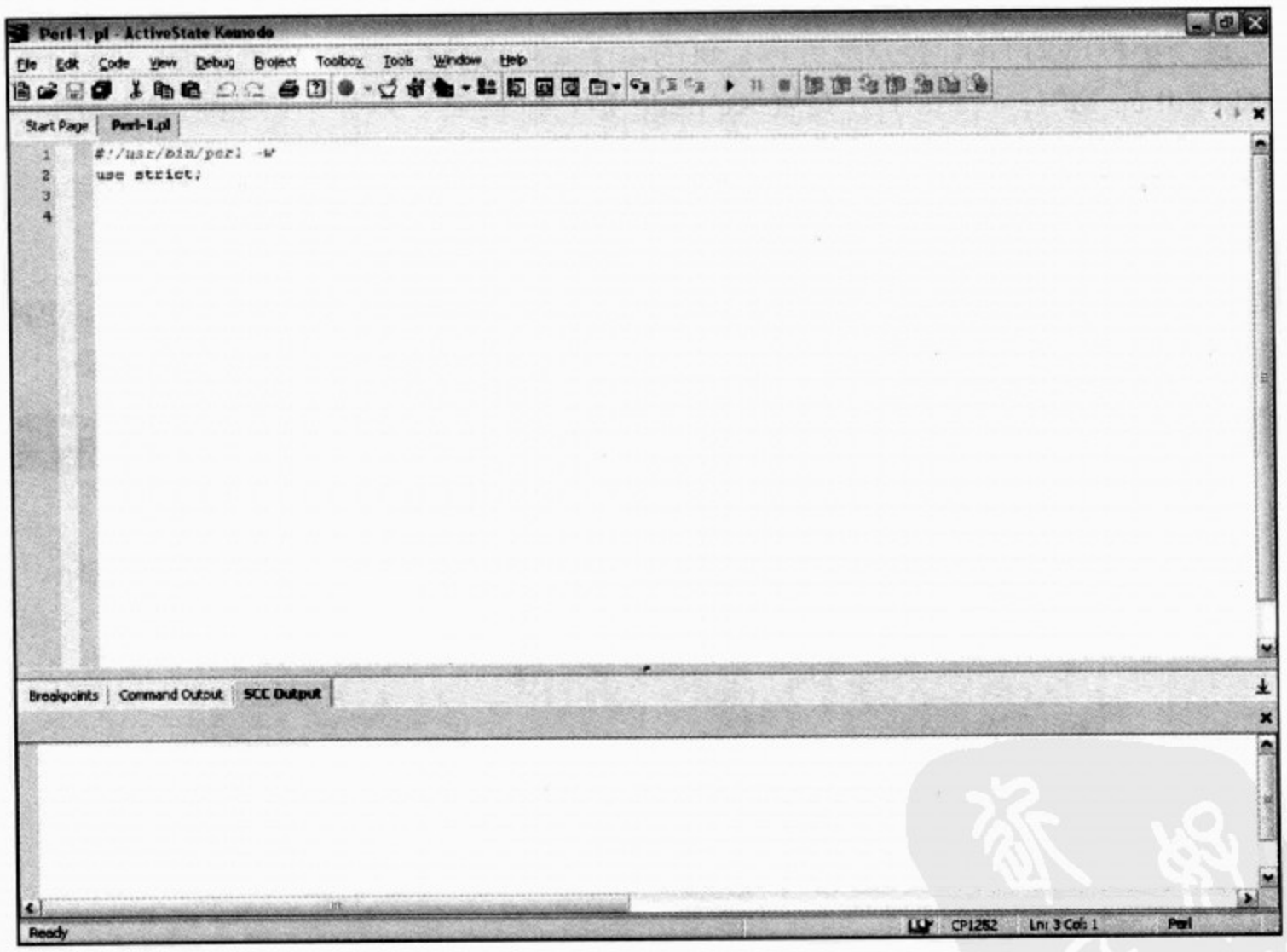


图 26-6

(6) 观察在 Komodo 界面右下角的 Output 窗格中显示的结果，如图 26-7 所示。注意到在 Output 窗格中显示了三行文本。

本章中 Perl 的例子使用 ActivePerl 5.8.4 测试通过。这些例子很可能无须修改就可以在所有 Perl 5.8 版本中运行，但并没有测试过。

如果使用的不是 Komodo 3.0 开发环境，那么可以在命令行中运行这个命名为 Simple.pl 的文件。假定已经安装了 ActivePerl(即设置了 PATH 环境变量)或者已将 perl 引擎添加到机器的路径中，那么输入下列命令并按回车，文件中的代码就会运行：

```
perl Simple.pl
```

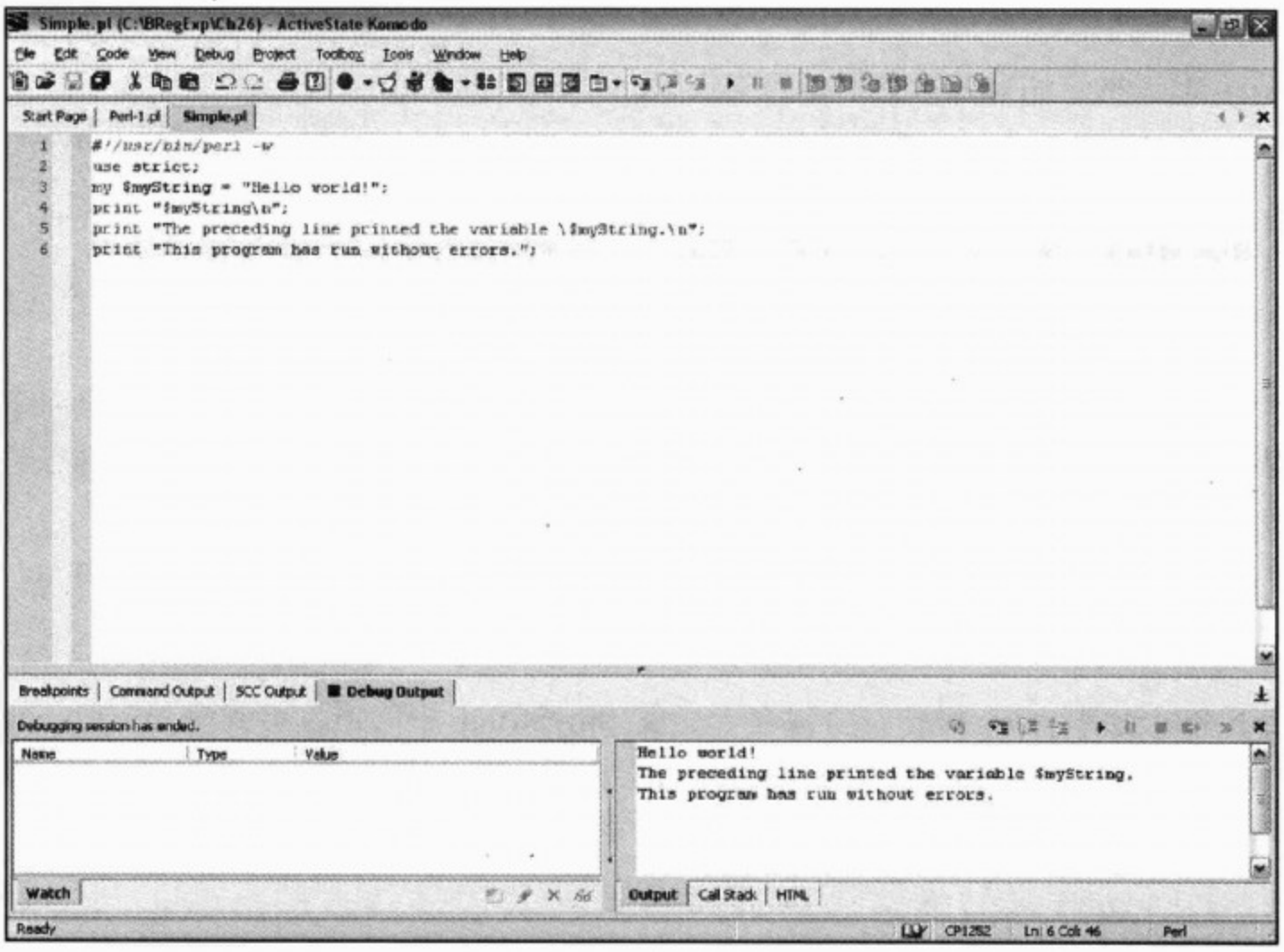


图　26-7

图 26-8 显示的是在 Windows XP 命令窗口中运行 Simple.pl 文件的结果。

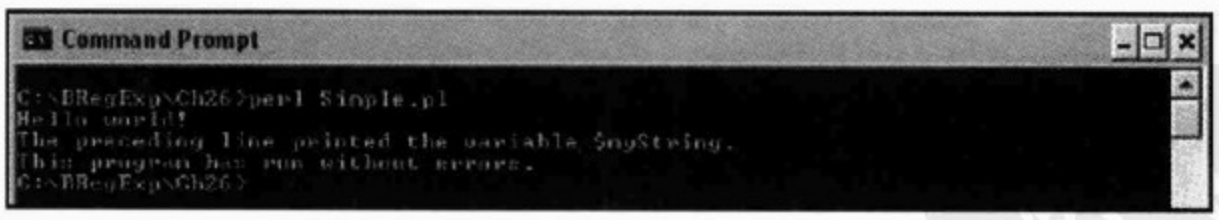


图　26-8

工作原理

由 Komodo 3.0 自动加入到 Perl 文件中的第一行代码如下：

```
#!/usr/bin/perl -w
```

字符表示该行是一个 Perl 的注释。从 # 字符开始到行尾的全部字符在正常情况下

都被视为 Perl 的注释。但是，当在 Unix 平台中文件首行的前两字符是 #! 时，它表示的是运行代码的程序所在的位置。如果是在 Windows 平台中运行 Perl，那么可以忽略这一行。但若是在其他类 Unix 的操作系统中，且 perl 引擎的位置不在当前显示的目录下，那么就需要对其进行适当修改。

Komodo 3.0 自动添加的另一行代码是 strict 编译指示。该编译指示会通知 Perl 引擎不允许使用不规范的编程习惯。该行代码的这种形式还表示应用 strict 编译指示的所有选项。简而言之，perl 引擎将会指出代码中可能存在的潜在问题：

```
use strict;
```

strict 编译指示还有三个更细化的限定版：

- use strict “vars”;——如果访问没有通过 our 或 use vars 声明、没有通过 my 局部化或没有完全限定的变量，那么会生成错误。
- use strict “refs”;——使用符号引用时会产生错误。
- use strict “subs”;——使用不是子程序的裸字标识符(bareword identifier)会产生编译错误。

在实践中，strict 编译指示的作用在于避免直接使用如$myString 这样的变量而不声明其作用域的情况出现，所以本例中使用了 my。这个 my 关键字声明的是一个包含在块或文件中的局部变量。

下面的代码行将字符串值 Hello world! 指定给具有局部作用域的变量$myString：

```
my $myString = "Hello world!";
```

接下来，使用 print 操作符将保存于变量 $myString 中的值输出到命令窗口，然后输出一个换行符(由 \n 表示)。此时可能会发现变量 $myString 包含在一对双引号里。在 Perl 中，如果想输出一个变量的值，可以简单地将该变量包含在一对引号中：

```
print "$myString\n";
```

这样就带来了如何输出 $ 符号的问题。事实上，通过在该符号前面添加一个反斜杠字符，就可以显示输出的是哪个变量的值了：

```
print "The preceding line printed the variable \$myString.\n";
```

也可以使用 print 操作符显示直接量文本：

```
print "This program has run without errors.";
```

当然，Perl 的内容要比这个简单的例子多得多，但通过它能获得在 Perl 中如何输出信息的一个感观印象。

26.2 使用 Perl 正则表达式的基本条件

对于不熟悉 Perl 的读者而言，本节内容只示范 Perl 中正则表达式的简单用法。本章也

不会提供如何使用 Perl 的完整教程。如果对 Perl 了解不多，但想要在真正的 Perl 程序中使用正则表达式，建议找一本不错的书，如 Paul Hoffman 的 *Perl For Dummies*(Wiley 2003)。

要使用 Perl 中的正则表达式功能，必须使用一个或多个正则表达式操作符。

26.3　使用 Perl 正则表达式操作符

Perl 正则表达式操作符与正则表达式模式密切相关。表 26-1 中列出了 Perl 正则表达式操作符及其简要说明。

表 26-1　Perl 正则表达式操作符及其说明

操　作　符	说　　明
m//	当根据一个正则表达式来匹配字符串时使用
s///	当匹配后替换一个模式时使用
q//等	一般引用
split//	将一个字符串拆分为一个子字符串列表

最简单的操作符是 m//，它用于测试一个字符串中是否存在与正则表达式匹配的内容。稍后你会看到，m// 操作符中的 m 在 Perl 代码中并不是必需的。但是，最好不要省略它，这样有助于更好地把握匹配过程。

26.3.1　使用 m//操作符

m// 操作符与 =~ 操作符一起使用可以测试一个字符串中是否存在指定正则表达式的匹配项。

试一试：使用 m// 操作符

(1) 在 Komodo 3.0 或其他文本编辑器中创建一个新的 Perl 文件，并输入下列代码：

```
#!/usr/bin/perl -w
use strict;
my $myString = "Hello world!";
if ($myString =~ m/world/)
{
  print "There was a match.";
}

else
{
print "There was no match.";
}
```

(2) 将代码保存为 SimpleMatch.pl。

(3) 按 F5 键，通过 Browse 按钮找到并选择 SimpleMatch.pl，然后按回车在调试模式下

运行代码。

(4) 观察 Output 窗格(在 Komodo 界面右下角)中显示的结果。如图 26-9 所示，显示的信息表明存在一个匹配项。

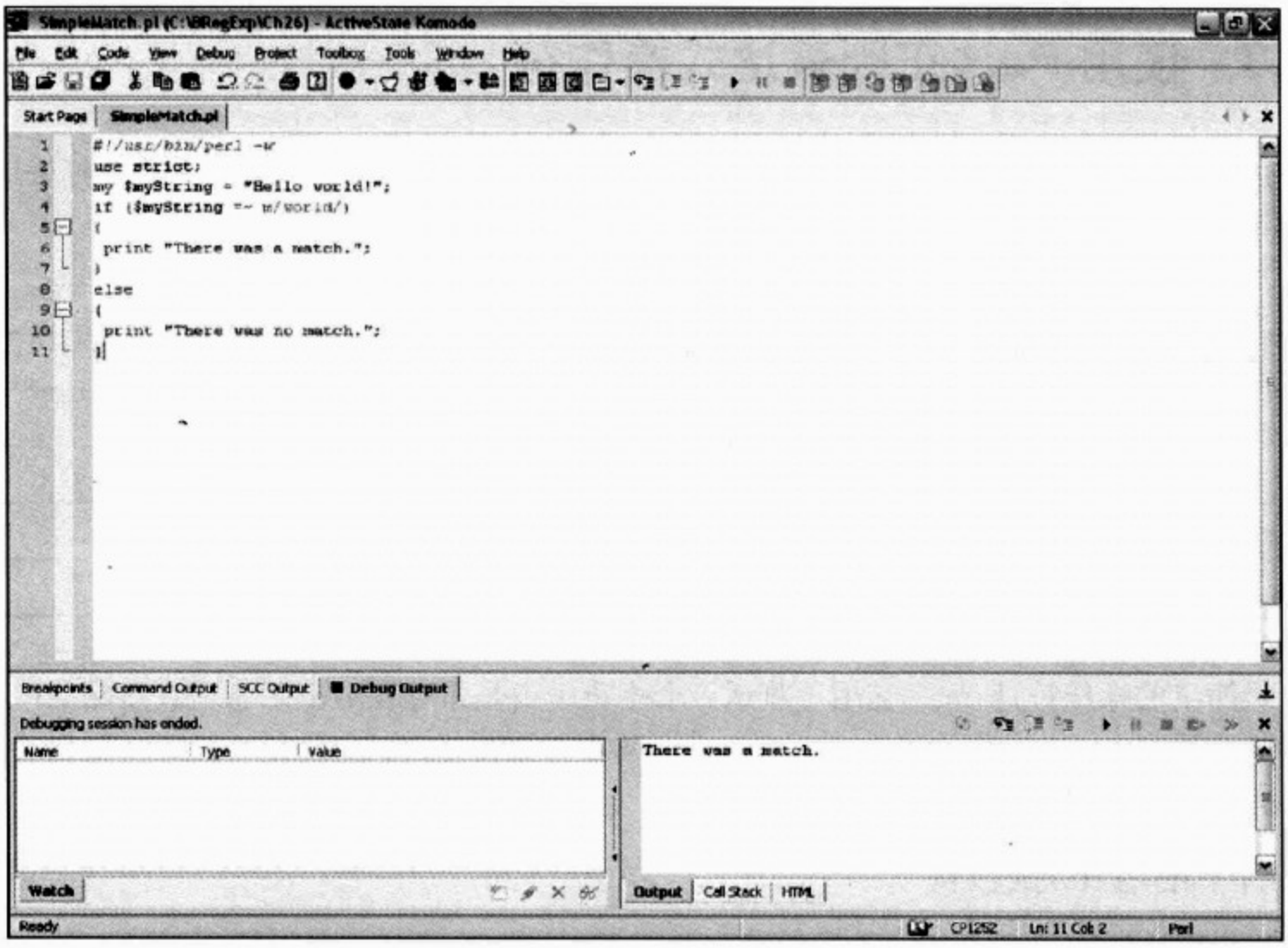


图 26-9

工作原理

首先，将字符串值 Hello world! 指定给变量 $myString。因为 strict 编译指示是有效的，所以要在变量名称前加上 my：

```
my $myString = "Hello world!";
```

然后，通过 if 语句来决定显示匹配成功还是显示匹配失败的信息。if 语句测试的是变量 $myString 中是否包含直接量正则表达式模式 world 的匹配项。其中 =~ 操作符与 m// 操作符的组合可以读作“匹配”。

虽然 Perl 中没有布尔值数据类型，但这两个操作符结合的结果却酷似该类型：

```
if ($myString =~ m/world/)
```

在默认情况下，Perl 中的匹配是区分大小写的。

如果找到一个匹配项(本例中的确存在一个匹配项)，则会显示匹配成功的信息。在 Perl 中，需要使用一对大括号来包括测试结果等价于 true 时执行的语句块，即使只有一条语句也不能省略大括号：

```
{
print "There was a match.";
}
```

如果没有找到匹配项，也会显示相关的信息。同样，此时 else 语句的大括号也是必需的，也就是说，即使 else 语句中只有一行代码，也不能省略这对大括号：

```
else
{
print "There was no match.";
}
```

m// 操作符可以与 Perl 支持的任何正则表达式匹配模式结合使用。下面的例子就示范了如何完成不区分大小写的匹配。不区分大小写的模式以小写的 i 跟在作为正则表达式模式限定符的一对正斜杠中第二个正斜杠的后面来表示：

```
$myTestString =~ m/world/i;
```

本例还介绍了一个非常有用的函数 chomp，它通常在代码中用来接收来自用户的输入。

试一试：不区分大小写的匹配

(1) 在 Komodo 或所选择的文本编辑器中输入下列代码，并将文件保存为 MatchInsensitive.pl：

```
#!/usr/bin/perl -w
use strict;
print "Enter a string. It will be matched against the pattern '/Star/i'.\n\n";
my $myTestString = <STDIN>;
chomp($myTestString);
if ($myTestString =~ m/Star/i)
{

 print "There is a match for '$myTestString'.";
}
else
{
 print "No match was found in '$myTestString'.";
}
```

(2) 运行代码。在 Komodo 3.0 中按 F5 并使用 Browse 按钮选择文件，再按回车键，或在命令行中输入 PerlMatchInsensitive.pl 并按回车键。

(3) 在第一次运行这些代码时，输入测试字符串 Startle，然后按回车键。观察如图 26-10 所示的结果。

在 Komodo 3.0 的 Output 面板中输入文本时，要保证光标处于正确的位置。如果光标当前位于 Code 面板而不是 Output 面板中，那么可能无意间输入的字符就会成为下次运行代码时出错的根源。

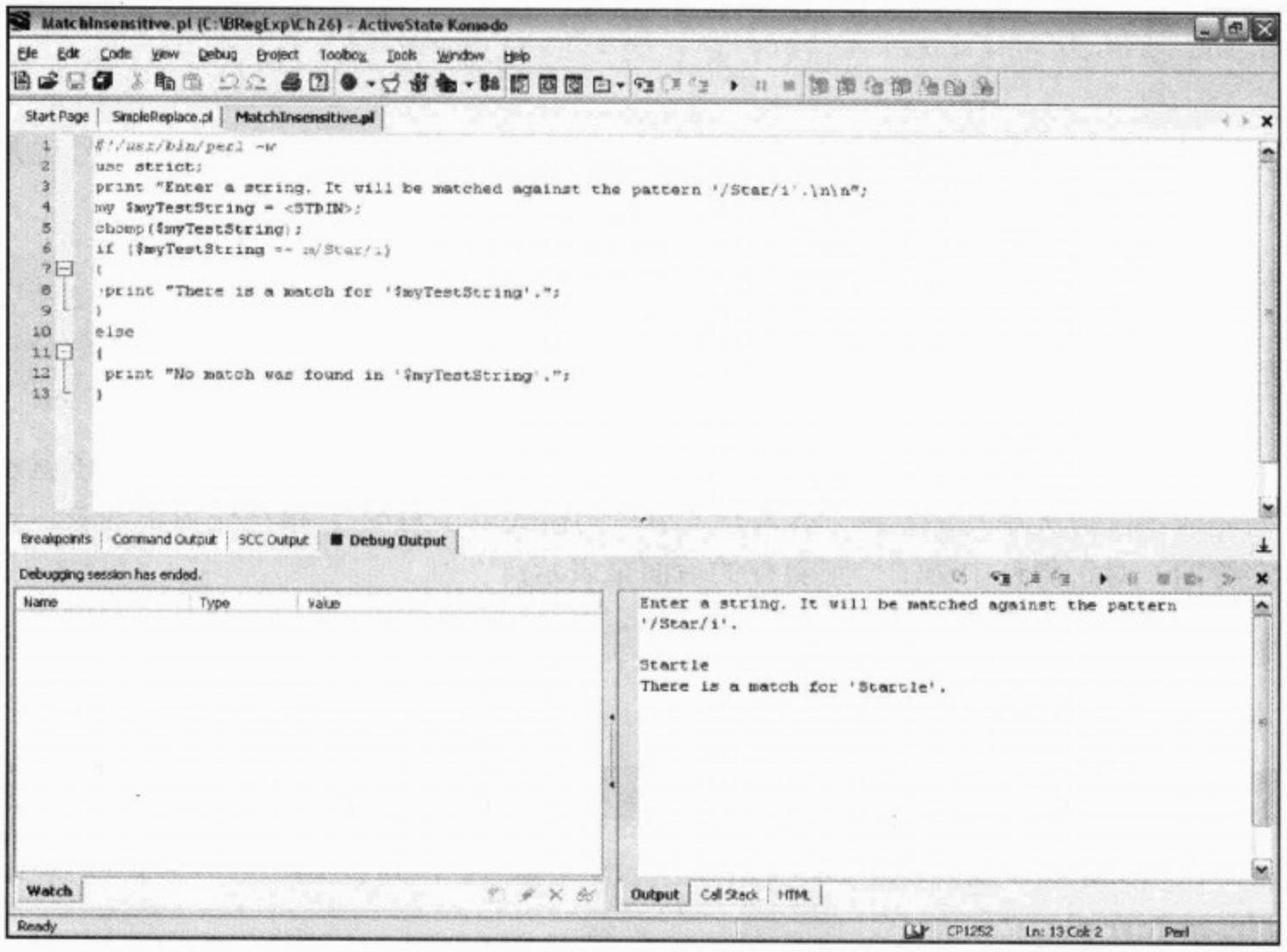


图 26-10

(4) 再次运行代码。输入测试字符串 startle，按回车键并观察结果。这次同样存在一个匹配项。因为当模式 Star 以不区分大小写的方式匹配时，会匹配 startle 开始位置的 star。

(5) 再次运行代码。输入测试字符串 Hello，然后按回车键并观察如图 26-11 所示的结果。

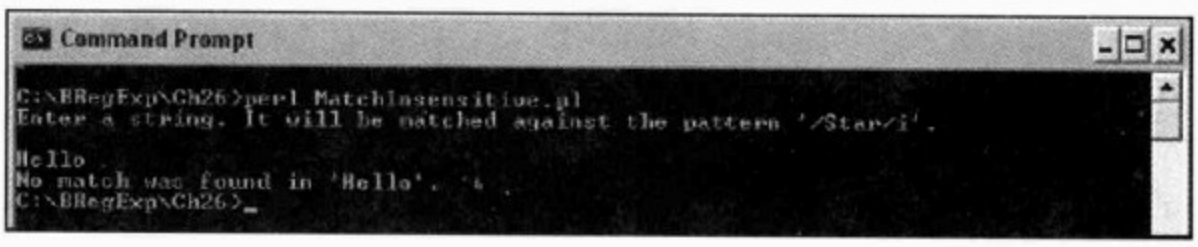


图 26-11

工作原理

首先，通过 print 操作符显示请用户输入测试字符串的信息：

```
print "Enter a string. It will be matched against the pattern '/Star/i'.\n\n";
```

然后，将用户在命令行中输入的字符序列指定给变量 $myTestString。其中的 <STDIN> 操作符会从标准输入中读取一行字符。一般来说，标准输入设备指的就是键盘。因此 <STDIN> 会从键盘中读取一行用户输入的字符，直到按回车键为止。由标准输入提供的这行字符中的一个小问题是它包含换行符。Perl 会将换行符作为匹配的目标字符序列的一

部分。因此，需要删除其中的换行符以保证匹配的准确：

```
my $myTestString = <STDIN>;
```

Perl 为删除从标准输入中取得的字符序列中的换行符提供了 chomp 函数：

```
chomp($myTestString);
```

下面显示的代码文件 MatchInsensitiveLengths.pl 也包含在本书的源代码压缩包中，在调用 chomp()函数的前后都显示了$myTestString 变量的长度。当测试字符串是 Startle 时，显示的长度是 8，比可见字符数多一个。末尾的换行符就是第 8 个字符：

```
#!/usr/bin/perl -w
use strict;
print "Enter a string. It will be matched against the pattern '/Star/i'.\n\n";
my $myTestString = <STDIN>;
my $myLength = length($myTestString);
print "The length before chomp() is $myLength.\n\n";
chomp($myTestString);
$myLength = length($myTestString);
print "The length after chomp() is $myLength.\n\n";
if ($myTestString =~ m/Star/i)
{
print "There is a match for '$myTestString'.\n\n";
}
else
{
print "No match was found in '$myTestString'.";
}
```

图 26-12 显示了在命令行中运行 MatchInsensitiveLengths.pl 的屏幕外观。注意在调用 chomp()前后所显示的$myTestString 变量的长度变化。

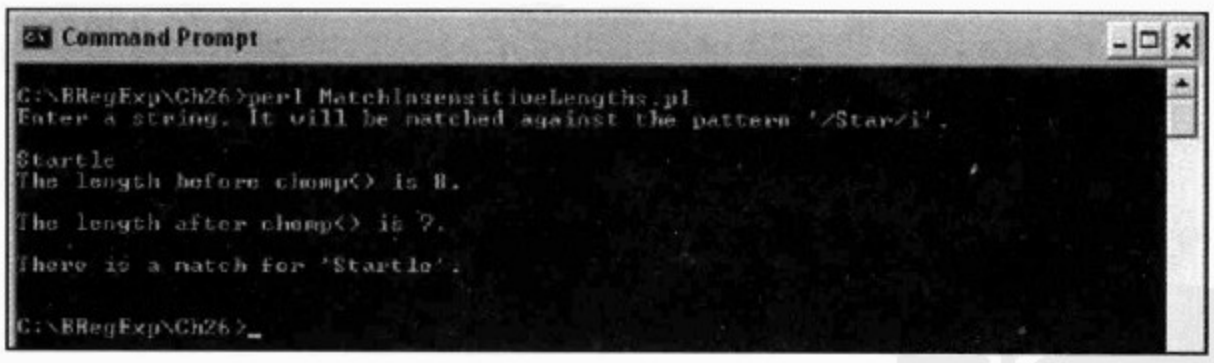


图　26-12

对于初学者而言，掌握 Perl 的难点之一就是许多结构都有多种表示(编写)方式。下面的两个例子就展示了一些类似的多变结构的用法，当处理由他人编写的代码时就有可能遇到这种情况。

事实上，m// 操作符中的 m 是可选的。但为了清晰起见(因为 m 能起到提示匹配的作用)，建议像下面的例子一样使用 m// 而不要用 //。

试一试：可选的“m”

(1) 在 Komodo 3.0 或选择的文本编辑器中输入下列代码，并将文件保存为 SimpleMatchNoM.pl：

```
#!/usr/bin/perl -w
use strict;
my $myString = "Hello world!";
if ($myString =~ /world/)
{
  print "There was a match.";
}
else
{
  print "There was no match.";
}
```

(2) 按 F5 然后再按回车键来运行这些代码。

(3) 观察结果。由于使用 // 和 m// 进行匹配的结果没有什么不同，所以结果与图 26-9 是一样的。

其中 chomp()函数的使用频率比较高，它的作用就是删除用户输入的一行字符结尾处的换行符。下面的例子示范了使用 chomp()函数的另一种方式，这种方式对于不常使用 Perl 的人而言有点隐晦，但却更加简洁。

试一试：另一种使用 chomp()的语法

(1) 在 Komodo 3.0 或其他文本编辑器中输入下列代码，并将文件保存为 MatchAlternativeChomp.pl：

```
#!/usr/bin/perl -w
use strict;
print "Enter a string. It will be matched against the pattern '/Star/i'.\n\n";
chomp (my $myTestString = <STDIN>);
if ($myTestString =~ m/Star/i)
{
  print "There is a match for '$myTestString'.";
}
else
{
  print "No match was found in '$myTestString'.";
}
```

(2) 在 Komodo 中按 F5 或通过在命令行中输入 Perl MatchAlternativeChomp.pl 来运行代码。

(3) 输入测试字符串 Star Training，然后按回车键。观察如图 26-13 所示的结果。

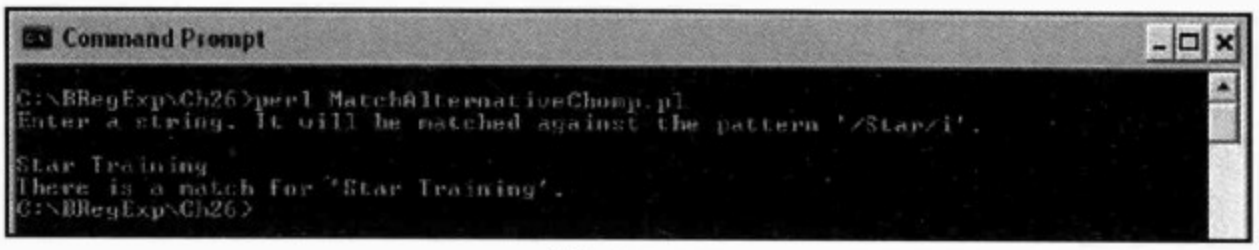


图　26-13

工作原理

下面这行代码：

```
chomp (my $myTestString = <STDIN>);
```

在功能上等价于：

```
my $myTestString = <STDIN>;
chomp ($myTestString);
```

前者的赋值操作符 = 的含义是先进行赋值操作，然后再调用 chomp() 函数。

```
print "Enter a string. It will be matched against the pattern '/Star/i'.\n\n";
chomp (my $myTestString = <STDIN>);
```

print 函数也有多种用法。比如可以像下面这样有条件地使用 print 操作符。下面的代码包含在下载的文件 MatchAlternativeChomp2.pl 中：

其中 if 语句与 print 操作符同在一行，而且位于要输出的字符串之后：

```
print "There is a match for '$myTestString'." if ($myTestString =~ m/Star/i);
```

if 语句测试中使用的 !~ 操作符的含义是“没有匹配项”：

```
print "There is no match for '$myTestString'." if ($myTestString !~ m/Star/i);
```

换句话说，并不是只能用模式来匹配字符串，也可以用模式匹配变量。当想要在代码中多次使用同一个模式时，匹配变量是非常有用的。

试一试：匹配变量

(1) 在文本编辑器中输入下列代码，并将文件保存为 MatchUsingVariable.pl：

```
#!/usr/bin/perl -w
use strict;
my $myPattern = "^\\d{5}(-\\d{4})?\$";
print "Enter a US Zip Code: ";
my $myTestString = <STDIN>;
chomp ($myTestString);
print "You entered a Zip code.\n\n" if ($myTestString =~ m/$myPattern/);
print "The value you entered wasn't recognized as a US Zip code." if ($myTestString
!~ m/$myPattern/);
```

(2) 在 Komodo 中或在命令行中运行这些代码。在出现提示时，输入测试字符串 12345，然后按回车并观察结果。

(3) 再次运行代码(如果使用 Windows 命令行，可以使用 F3)。当出现提示时，输入测试字符串 12345-6789，按回车然后观察显示的结果。

(4) 再次运行代码。当出现提示时，输入测试字符串 Hello world! 按回车并观察结果，如图 26-14 所示。

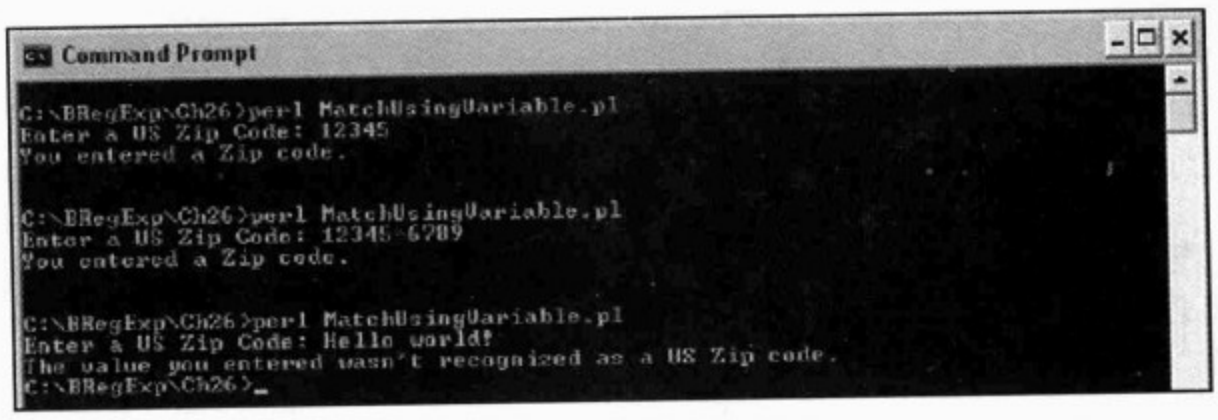


图 26-14

工作原理

首先，声明变量$myPattern 并将模式 ^\d{5}(-\d{4})?$ 指定给它。注意当使用\d 元字符和$元字符时，必须在前面加上一个额外的反斜杠。

这个模式使用位置元字符 ^ 和 $ 来表示模式匹配整个测试字符串。而该模式可以匹配由五个数字组成的测试字符串(\d{5})——美国邮政编码的简短形式；也可以匹配五个数字组成的字符串后面可选地跟着一个连字符和四个数字((-\d{4})?)——美国邮政编码的扩展版格式。其中，-\d{4}是位于一对圆括号中的组，因此 ? 限定符表示全部 -\d{4} 都是可选的：

```
my $myPattern = "^\\d{5}(-\\d{4})?\$";
```

接着，请用户输入一个邮政编码。使用 <STDIN> 来捕获在标准输入中输入的内容。然后，再使用 chomp()删除 $myTestString 尾端的换行符：

```
print "Enter a US Zip Code: ";
my $myTestString = <STDIN>;
chomp ($myTestString);
```

然后，使用了两个 print 语句，每个都带有 if 语句和是否显示有关信息的测试表达式。下面第一行中 if 语句的含义是，如果有匹配项则显示前面的信息。下一行中 if 语句的含义是如果没有匹配项则显示前面的信息：

```
print "You entered a Zip code.\n\n" if ($myTestString =~ m/$myPattern/);
print "The value you entered wasn't recognized as a US Zip code." if ($myTestString !~
m/$myPattern/);
```

26.3.2 使用其他正则表达式定界符

Perl 的灵活性还体现在能够指定其他字符作为正则表达式模式的定界符。

Perl 中默认的正则表达式模式定界符是一对正斜杠，如下所示：

```
my $myTestString = "Hello world!";
```

```
$myTestString =~ /world/;
```

但是，Perl 也允许开发人员使用其他字符作为正则表达式的定界符——但必须指定 m。从个人角度来说，任何时候都只使用一对正斜杠是最简单明了的。虽然 Perl 提供了使用其他字符的可能性，但如果不知道 Perl 的这种灵活性，则很容易导致分不清是用于匹配的字符，还是用于替代正斜杠作为定界符的字符。

下面的例子展示了如何将可以作为目标匹配的大括号、一对感叹号和一对句点字符用做正则表达式的定界符。

试一试：使用非默认的定界符

(1) 在文本编辑器中输入下列代码，并将文件保存为 NonDefaultDelimiters.pl：

```
#!/usr/bin/perl -w
use strict;
print "This example uses delimiters other than the default /somePattern/.\n\n";
my $myTestString = "Hello world!";
print "It worked using paired { and }\n\n" if $myTestString =~ m{world};
print "It worked using paired ! and !\n\n" if $myTestString =~ m!world!;
print "It worked using paired . and .\n\n" if $myTestString =~ m.world.;
```

(2) 在 Komodo 中按 F5 或通过在命令行中输入 Perl NonDefaultDelimiters.pl 并按回车运行代码。

(3) 观察显示的结果。如图 26-15 所示，从完成匹配的角度讲，可以作为匹配项目标的 { 和 } 或一对 ! 和 !，亦或是一对句点字符，都可以用做定界符。

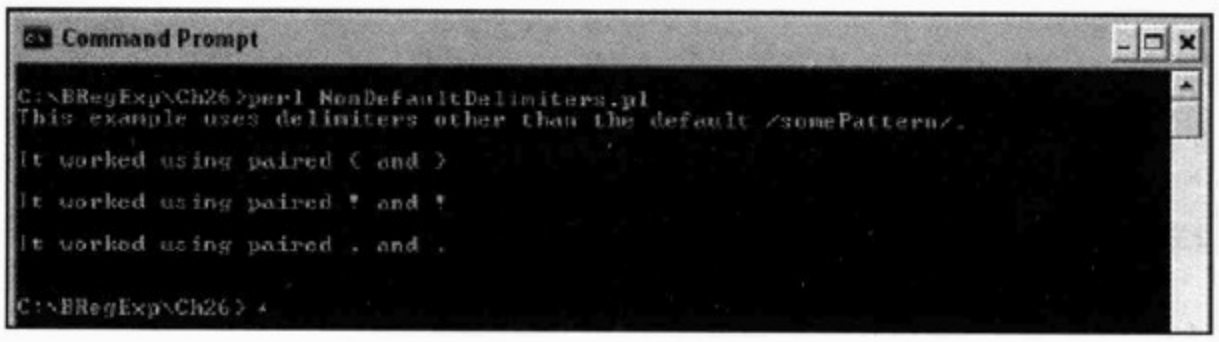


图　26-15

工作原理

在输出简单的信息之后，将字符串 Hello world! 指定给变量$myTestString：

```
my $myTestString = "Hello world!";
```

然后，使用了三次 print 操作符来输出使用自定义的定界符时的匹配信息。当然，if 语句的测试条件必须满足。

26.3.3　使用置入变量匹配

如果使用 Perl 编程的时间不长，在看到包含在双引号中的变量时可能会感到奇怪。但更加令人惊奇的是，在正则表达式模式中也可以包含变量。

根据变量是否出现在模式的末尾，有两种在模式中包含变量的方式可供选择。
如果变量出现在模式的末尾，可以写成下面这样：

```
/some characters$myPattern/
```

但是，如果想在模式中任何其他位置上使用变量，则需要写成下面这样才行：

```
/${myPattern}some other characters/
```

试一试：使用置入变量匹配

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
my $myTestString = "shells";
my $myPattern = "she";
print "$myPattern is found in $myTestString.\n\n" if ($myTestString =~
m/${myPattern}ll/);
$myTestString = "scar";
$myPattern = "car";
print "$myPattern is found in $myTestString.\n\n" if ($myTestString =~
m/s$myPattern/);
```

(2) 将代码保存为 MatchingVariableSubstitution.pl。
(3) 运行代码并观察如图 26-16 所示的结果。

图 26-16

工作原理

首先，看一下能在模式中任何位置置入变量的语法。为变量$myTestString 和$myPattern 分别赋值：

```
my $myTestString = "shells";
my $myPattern = "she";
```

下面这一行代码是为了印刷方便才分开在两行中的。注意 if 语句测试部分的正则表达式模式中的语法，变量$myPattern 被放到了模式中并写成${myPattern}。这对大括号确保了明白无误地定义模式：

```
print "$myPattern is found in $myTestString.\n\n" if ($myTestString =~
m/${myPattern}ll/);
```

本例中第二部分使用的语法则只能用在模式的结尾处。此时，变量 $myPattern 仍然写成$myPattern。由于使用正斜杠来界定模式的结尾，所以含义是明确的：

```
$myTestString = "scar";
$myPattern = "car";
print "$myPattern is found in $myTestString.\n\n" if ($myTestString =~
m/s$myPattern/);
```

我们在本节匹配模式的例子中已经体验到，Perl 中实现匹配的语法的确具有极大的灵活性。

26.3.4　使用 s///操作符

s/// 操作符用于将测试字符串中匹配的子字符串替换(或转换)成指定的替换字符串。搜索和替换的语法采取下列常规形式：

```
s/模式/替换字符串/修饰符
```

如果存在匹配项，s/// 返回匹配项的数量。匹配项的数量取决于是否对 s/// 操作符应用了 g(global)修饰符。如果指定了 g 修饰符，正则表达式引擎就会尝试找到测试字符串中所有的匹配项。

在下面的例子中，直接量模式 Star 会被替换(转换)成字符串 Moon。

试一试：使用 s/// 操作符

(1) 在 Komodo 或其他文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
my $myString = "I attended a Star Training Company training course.";
my $oldString = $myString;
$myString =~ s/Star/Moon/;
print "The original string was: \n'$oldString'\n\n";
print "After replacement the string is: \n'$myString'\n\n";
if ($oldString =~ m/Star/)
{
print "The string 'Star' was matched and replaced in the old string";
}
```

(2) 将代码保存为 SimpleReplace.pl。

(3) 在 Komodo 3.0 中按 F5 运行或在命令行中输入 Perl SimpleReplace.pl 命令运行这些代码(假设源代码文件被保存在当前目录或者机器的 PATH 环境变量指定的目录中)。观察如图 26-17 显示的结果。

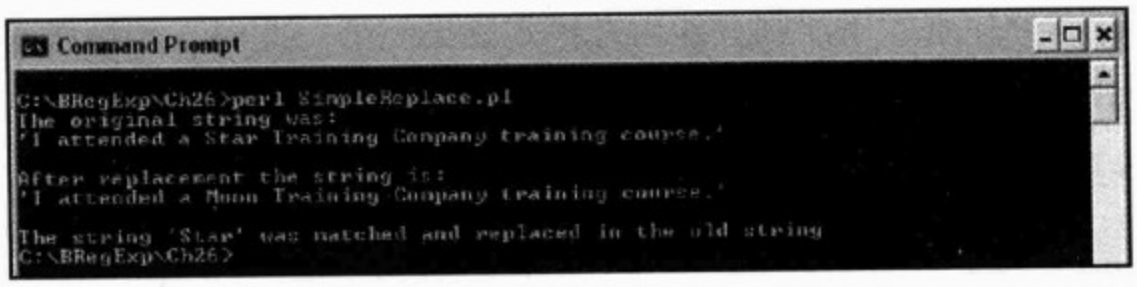


图 26-17

工作原理

将测试字符串指定给变量$myString:

```
my $myString = "I attended a Star Training Company training course.";
```

使用变量$oldString 来保存原始值，以备后面显示使用：

```
my $oldString = $myString;
```

下面的代码将测试字符串中的字符序列 Star 替换成字符序列 Moon：

```
$myString =~ s/Star/Moon/;
```

让用户看到原始的和替换之后的字符串：

```
print "The original string was: \n'$oldString'\n\n";
print "After replacement the string is: \n'$myString'\n\n";
if ($oldString =~ m/Star/)
{
print "The string 'Star' was matched and replaced in the old string";
}
```

26.3.5 使用带全局修饰符的 s///

通常，我们都希望替换测试字符串中某个字符序列的所有实例。本书前面举的那个 Star Training Company 的例子就是一个典型的例证。要想替换所有匹配项，则需要使用全局修饰符 g。

要进行全局性的替换，需要使用下面的代码：

```
$myTestString =~ s/模式/替换字符串/g
```

在第三个正斜杠之后的修饰符 g 表示要进行全局性的替换操作。

试一试：使用带全局修饰符的 s/// 操作符

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
print "This example uses the global modifier, 'g'\n\n";
my $myTestString = "Star Training Company courses are great. Choose Star for your
training needs.";
my $myOnceString = $myTestString;
```

```
my $myGlobalString = $myTestString;
my $myPattern = "Star";
my $myReplacementString = "Moon";
$myOnceString =~ s/$myPattern/$myReplacementString/;
$myGlobalString =~ s/$myPattern/$myReplacementString/g;
print "The original string was '$myTestString'.\n\n";
print "After a single replacement it became '$myOnceString'.\n\n";
print "After global replacement it became '$myGlobalString'.\n\n";
```

(2) 将代码保存为 GlobalReplace.pl。

(3) 运行代码并观察结果。如图 26-18 所示，在没有 g 修饰符的情况下，只有一个字符序列 Star 被替换。而在使用了 g 修饰符之后，所有的 Star(本例中为两个)都被替换了。

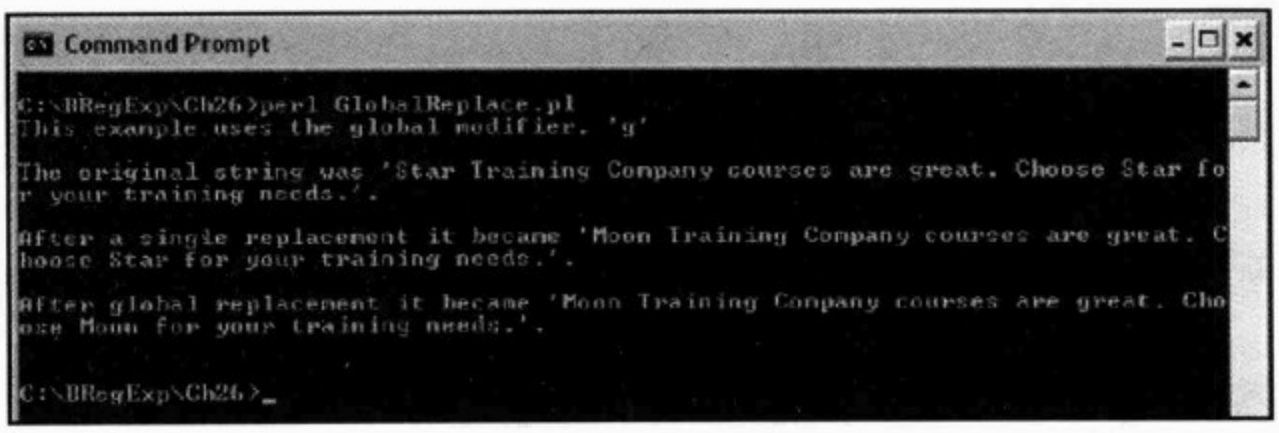


图　26-18

工作原理

将测试字符串指定给变量$myTestString：

```
my $myTestString = "Star Training Company courses are great. Choose Star for your
training needs.";
```

将原始测试字符串的值复制到变量$myOnceString 和 $myGlobalString 中：

```
my $myOnceString = $myTestString;
my $myGlobalString = $myTestString;
```

将模式 Star 指定给变量$myPattern：

```
my $myPattern = "Star";
```

将替换字符串 Moon 指定给变量$myReplacementString：

```
my $myReplacementString = "Moon";
```

变量$myOnceString 中保存的是替换一个 Star 后的结果：

```
$myOnceString =~ s/$myPattern/$myReplacementString/;
```

由于使用了 g 修饰符，变量 $myGlobalString 中保存的是替换了所有匹配项(本例中为两个)之后的结果：

```
$myGlobalString =~ s/$myPattern/$myReplacementString/g;
```

然后分别输出了原始字符串、替换一次后的字符串和执行全局替换后的字符串：

```
print "The original string was '$myTestString'.\n\n";
print "After a single replacement it became '$myOnceString'.\n\n";
print "After global replacement it became '$myGlobalString'.\n\n";
```

26.3.6 使用 s///与默认变量

可以使用 s/// 来搜索替换保存在默认变量 $_ 中的值。

在使用 s/// 和默认变量 $_ 时，有两种语法形式。一种是正常的 s/// 语法，使用变量名、=~ 操作符、模式以及替换文本：

```
$_ =~ s/模式/替换字符串/修饰符;
```

另一种更简洁的语法则允许省略默认变量和=~操作符。因此可以简单地写成如下这样：

```
s/模式/替换字符串/修饰符;
```

试一试：使用 s/// 和默认变量

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
$_ = "I went to a training course from Star Training Company.";
print "The default string, \$_, contains '$_'.\n\n";
if (s/Star/Moon/)
{
print "A replacement has taken place using the default variable.\n";
print "The replaced string in \$_ is now '$_'.";
}
```

(2) 将代码保存为 ReplaceDefaultVariable.pl。

(3) 运行代码，并观察如图 26-19 所示的显示结果。

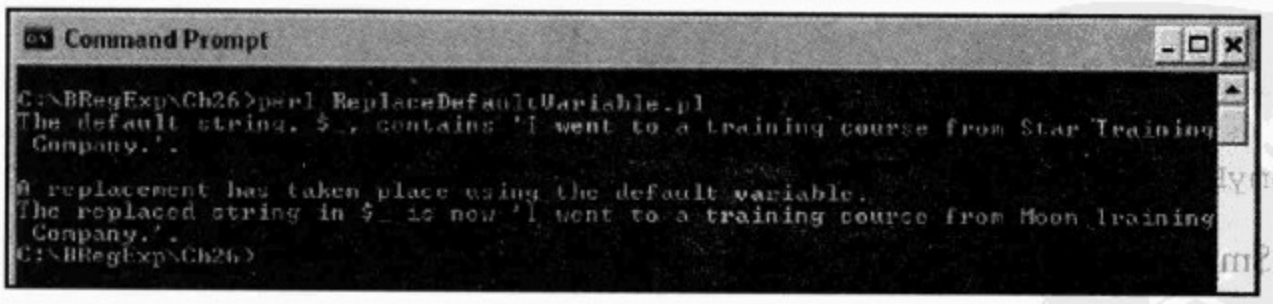


图 26-19

工作原理

将测试字符串指定给默认变量$_：

```
$_ = "I went to a training course from Star Training Company.";
```

输出包含在默认变量中的值：

```
print "The default string, \$_, contains '$_'.\n\n";
```

if 语句的测试部分使用了替换默认变量中匹配项的简洁语法：

```
if (s/Star/Moon/)
```

可能完整的语法看起来更直观一些：

```
if ($_ =~ s/Star/Moon/)
```

无论使用哪种语法，都会以同样的结果通知用户替换操作已经完成，而且会显示替换操作之后的结果字符串：

```
print "A replacement has taken place using the default variable.\n";
print "The replaced string in \$_ is now '$_'.";
```

26.3.7　使用 split 操作符

split 操作符用于根据正则表达式的匹配项来拆分测试字符串。

下面的例子示范了如何将一个逗号分隔的数据序列拆分成各自独立的部分。

试一试：使用 split 操作符

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
my $myTestString = "A, B, C, D";
print "The original string was '$myTestString'.\n";
my @myArray = split/,\s?/, $myTestString;
print "The string has been split into four array elements:\n";
print "$myArray[0]\n";
print "$myArray[1]\n";
print "$myArray[2]\n";
print "$myArray[3]\n";
print "Displaying array elements using the 'foreach' statement:\n";
foreach my $mySplit (split/,\s?/, $myTestString)
{
print "$mySplit\n";
}
```

(2) 将代码保存为 SplitDemo.pl。

(3) 运行代码并观察如图 26-20 所示的结果。

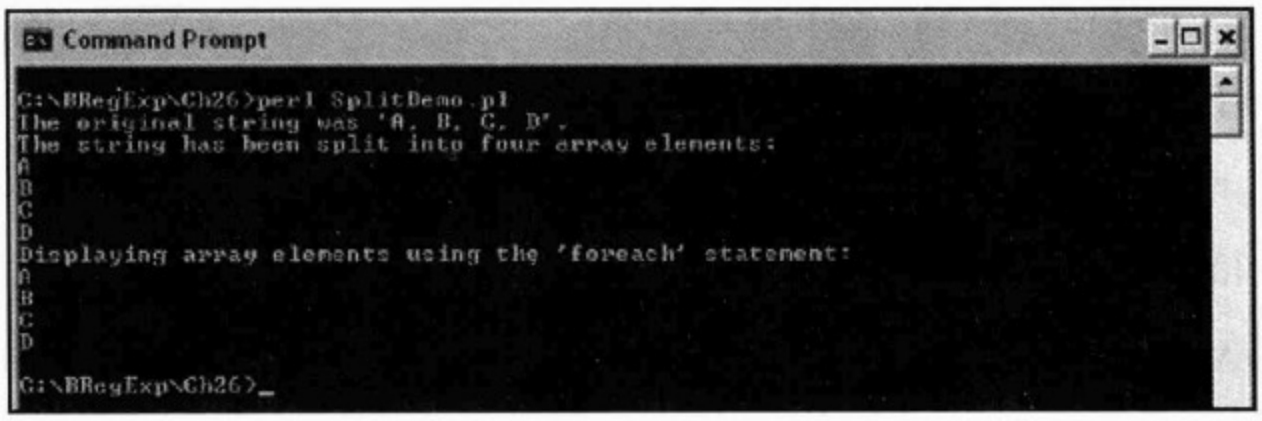


图 26-20

工作原理

将一个由逗号和一个空格分隔的数据序列指定给变量$myTestString：

```
my $myTestString = "A, B, C, D";
```

首先显示原始测试字符串的值：

```
print "The original string was '$myTestString'.\n";
```

通过 split 操作符将拆分后的结果指定给@myArray 数组。所用的模式匹配一个逗号和一个可选的空格符。而 split 操作符的目标是变量$myTestString：

```
my @myArray = split/,\s?/, $myTestString;
```

然后，通过数组索引来显示由拆分测试字符串得到的各个独立数据项的值：

```
print "The string has been split into four array elements:\n";
print "$myArray[0]\n";
print "$myArray[1]\n";
print "$myArray[2]\n";
print "$myArray[3]\n";
```

或者，更美观一点，使用一条 foreach 语句来显示拆分$myTestString 变量得到的每个独立数据项：

```
print "Displaying array elements using the 'foreach' statement:\n";
foreach my $mySplit (split/,\s?/, $myTestString)
{
print "$mySplit\n";
}
```

26.4 Perl 支持的元字符

Perl 支持大量有用的元字符，如表 26-2 所示。

表 26-2　Perl 支持的元字符

元　字　符	说　　明
.(句点字符)	匹配任何字符(唯一的例外是根据模式不同，可能不匹配换行符)
\w	匹配一个字母字符、数字或者一个下划线字符。有时也称为一个“单词字符”。相当于字符类 [A-Za-z0-9_]
\W	匹配一个除字母字符、数字以及下划线字符之外的字符。相当于字符类 [^A-Za-z0-9_] 或 [^\w]
\s	匹配一个空白符
\S	匹配一个非空白符
\d	匹配一个数字。相当于字符类 [0-9]
\D	匹配一个非数字。相当于字符类 [^0-9]
?	限定符。匹配前面字符或组出现零个或一次的情况
*	限定符。匹配前面字符或组出现零个或多次的情况
+	限定符。匹配前面字符或组出现一次或多次的情况
{n,m}	限定符。匹配前面字符或组出现至少 n 次、最多 m 次的情况
(...)	捕获圆括号(组)
$1 等	可以访问捕获组的变量
|	交替选择字符
\b	匹配一个词边界——即匹配位于一个单词字符([A-Za-z0-9_])和一个非单词字符之间的位置
[...]	字符类。匹配方括号内一组字符中的一个字符
[^...]	取反的字符类。匹配不在方括号内的一个任意字符
\A	一个匹配测试字符串第一个字符之前位置的位置元字符
\Z	一个匹配一行或一个字符串最后一个非换行符之后位置的位置元字符
\z	一个匹配字符串最后一个字符之后位置的位置元字符。与模式无关
(?= ...)	肯定式向前查找
(?! ...)	否定式向前查找
(?<= ...)	肯定式向后查找
(?<! ...)	否定式向后查找
\p{charClass}	匹配一个指定的 Unicode 字符类或字符块中的字符
\P{charClass}	匹配一个非指定的 Unicode 字符类或字符块中的字符

26.4.1　在 Perl 中使用限定符

Perl 支持很多相当典型的限定符。

? 元字符匹配前面字符或组的零个或一个实例。换句话说，前面的字符或组是可选的。

比如，要匹配 bat 和 bats，可以使用模式 bats?。此时 ? 元字符表示 s 是可选的。

* 元字符匹配前面字符或组的零个或多个实例。换句话说，前面的字符或组可以出现零次(一次也不出现)或者任意多次。因此，模式 AB* 将匹配以下字符序列：A、AB、ABB 和 ABBB 等。

+ 元字符匹配前面字符或组的一个或多个实例。换句话说，前面的字符或组必须至少出现一次，并且可以是大于一的任意多次。因此，模式 AB+ 将匹配以下字符序列：AB、ABB 和 ABBB 等。但这个模式不匹配 A，因为要匹配成功必须至少要有一个字符 B。

要匹配任何 ?、* 或 + 元字符自身，只需在这些限定符前面加一个反斜杠即可。也就是说，要分别写成 \?、* 和 \+。

同样，使用大括号的限定符语法也是有效的。比如，模式 [A-Z]\d{3} 匹配跟在一个大写字母字符后面的三个数字。而模式 [A-Z]\d{1,3} 则匹配跟在一个大写字母字符后面的 1~3 个数字，也就是说，它匹配 A1、A12 和 A123。

模式 [A-Z]\d{2,} 将匹配一个大写的字母字符后跟两个或多个数字。因此，它匹配 A12、A123、A1234 和 A12345 等。但它不匹配 A1，因为要成功匹配必须至少要有两个数字。

26.4.2 使用位置元字符

Perl 支持 ^ 和 $ 这两个位置元字符。^ 元字符匹配一行或一个字符串第一个字符之前的位置。而 $ 元字符匹配一行或一个字符串最后一个非换行符之后的位置。

\A 位置元字符匹配一个字符串开始之前的位置。

\z 位置元字符匹配一个字符串最后一个字符之后的位置。

试一试：使用位置元字符

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
print "\nThis example demonstrates the use of the ^ and \$ positional
metacharacters.\n\n";
my $myPattern = "cape";
my $myTestString = "escape";
print "In '$myTestString' there is a match for the pattern '$myPattern'.\n\n" if
($myTestString =~ m/$myPattern/);
$myPattern = "^cape";
print "When the pattern is '$myPattern' there is no match for '$myTestString'.\n\n"
if ($myTestString !~ m/$myPattern/);
$myPattern = "cape\$";
print "But there is a match for '$myTestString' when the pattern is
'$myPattern'.\n\n" if ($myTestString =~ m/$myPattern/);
```

(2) 将代码保存为 PositionalMetacharacters.pl。

(3) 运行代码并观察如图 26-21 所示的结果。

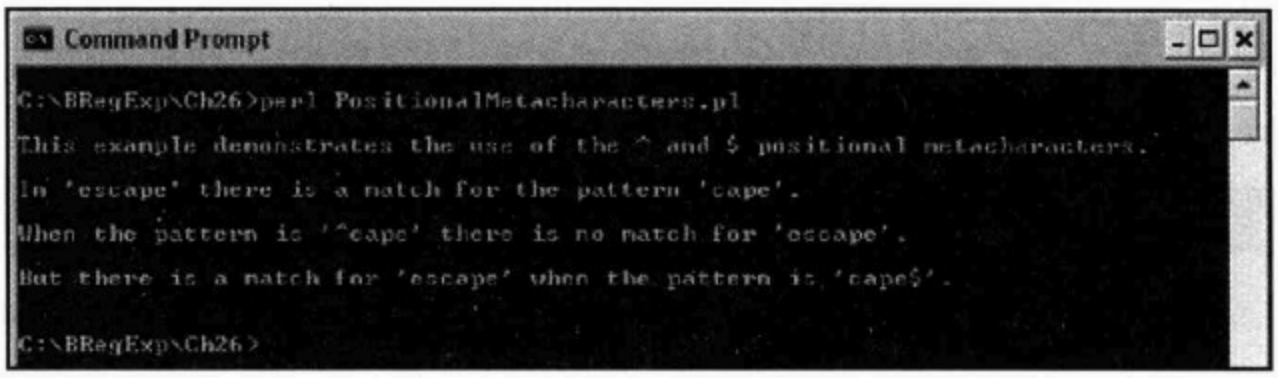


图　26-21

工作原理

首先，将一条简单的信息显示给用户：

```
print "\nThis example demonstrates the use of the ^ and \$ positional
metacharacters.\n\n";
```

然后，定义要使用的第一个模式。它由简单的字符序列组成，不包含任何位置元字符：

```
my $myPattern = "cape";
```

定义测试字符串：

```
my $myTestString = "escape";
```

执行匹配。if 语句确保了只有匹配成功时才会显示前面的信息：

```
print "In '$myTestString' there is a match for the pattern '$myPattern'.\n\n" if
($myTestString =~ m/$myPattern/);
```

修改模式使其包含一个 ^ 位置元字符，它将只匹配以 cape 开头的字符序列：

```
$myPattern = "^cape";
```

结果会执行下面这条语句，显示匹配失败的信息：

```
print "When the pattern is '$myPattern' there is no match for '$myTestString'.\n\n"
if ($myTestString !~ m/$myPattern/);
```

再次修改模式，使其只匹配位于测试字符串结尾处的字符序列 cape：

```
$myPattern = "cape\$";
```

当用模式匹配 escape 时，找到了一个匹配项，所以显示匹配成功的信息：

```
print "But there is a match for '$myTestString' when the pattern is
'$myPattern'.\n\n" if ($myTestString =~ m/$myPattern/);
```

26.4.3　Perl 中的捕获组

在 Perl 中，捕获组由一对圆括号定义。第一个捕获组就是模式中带有最左边的开始圆括号的组。其他的捕获组也由相应的成对圆括号定义，而所有捕获组会根据各自开始圆括

号的先后顺序进行编号。

可以在正则表达式外部通过编号的变量如 $1、$2 等来访问捕获组。

在 Perl 中，完整的匹配项保存在变量 $& 中。

试一试：Perl 中基本的捕获组

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
my $myPattern = "([A-Z])(\\d)";
my $myTestString = "B99";
$myTestString =~ m/$myPattern/;
print "The pattern is '$myPattern'.\n";
print "The test string is '$myTestString'.\n";
print "The whole match is '$&', contained in the \$& variable.\n";
print "The first captured group is '$1', contained in '\$1'.\n";
print "The second captured group is '$2', contained in '\$2'\n";
```

(2) 将代码保存为 CapturedGroups.pl。

(3) 运行代码并观察结果。如图 26-22 所示，与模式(([A-Z])(\d)) 匹配的整个匹配项是通过变量 $& 取得的。

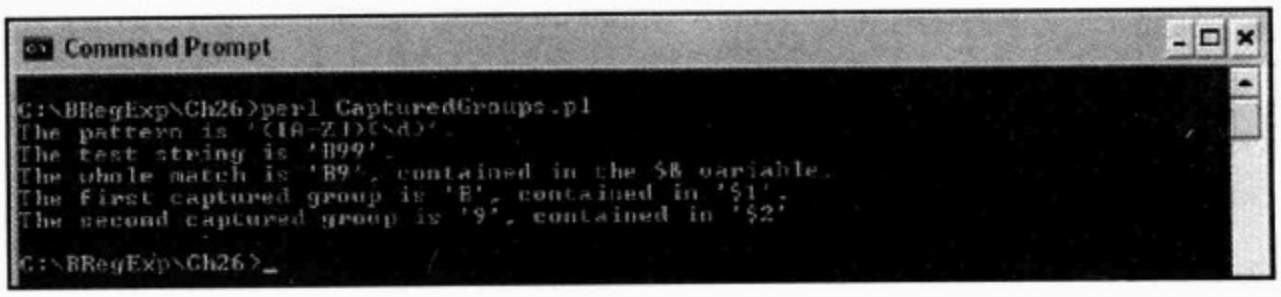


图 26-22

工作原理

将据以匹配的模式指定给变量 $myPattern：

```
my $myPattern = "([A-Z])(\\d)";
```

将测试字符串指定给变量 $myTestString：

```
my $myTestString = "B99";
```

变量 $myTestString 与 $myPattern 进行匹配：

```
$myTestString =~ m/$myPattern/;
```

将测试字符串及模式的值显示给用户：

```
print "The pattern is '$myPattern'.\n";
print "The test string is '$myTestString'.\n";
```

将 $& 变量中保存的整个匹配项(本例中是字符序列 B9)显示给用户：

```
print "The whole match is '$&', contained in the \$& variable.\n";
```

第一对圆括号对应的捕获组匹配字符类 [A-Z]。因此，变量 $1 中保存了单个字符 B：

```
print "The first captured group is '$1', contained in '\$1'.\n";
```

第二对圆括号对应的捕获组匹配元字符 \d。因此，变量 $2 中保存数字 9：

```
print "The second captured group is '$2', contained in '\$2'\n";
```

26.4.4　在 Perl 中使用反向引用

Perl 支持反向引用，即可以在正则表达式模式内部使用对捕获组的引用。

有关使用反向引用的一个经典的例子就是识别并纠正文本中的重复单词。下面的例子示范了如何使用反向引用实现这一目标。

试一试：使用反向引用检测重复的单词

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
my $myPattern = "(\\w+)(\\s+\\1\\b)";
my $myTestString = "Paris in the the Spring Spring.";
print "The original string was '$myTestString'.\n";
$myTestString =~ s/$myPattern/$1/g;
print "The captured group was: '$1'.\n";
print "Any doubled word has now been removed.\n";
print "The string is now '$myTestString'.\n";
```

(2) 将代码保存为 DoubledWord.pl。

(3) 运行代码并观察结果。如图 26-23 所示，原始测试字符串中有两个重复的单词：the 和 Spring。而在替换后的字符串中，这两对重复的单词都被删除了。

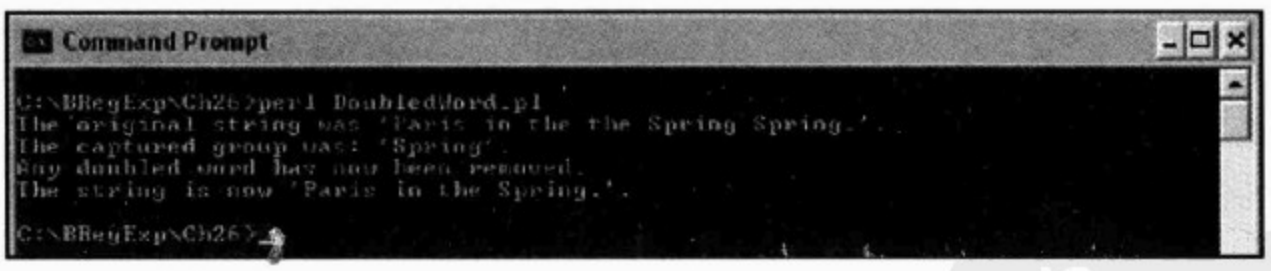


图　26-23

工作原理

本例所用的模式指定给了变量 $myPattern。模式中的元字符 \w、\s 和 \b 都必须加一个额外的反斜杠。

注意，反向引用 \1 同样也必须加一个额外的反斜杠：

```
my $myPattern = "(\\w+)(\\s+\\1\\b)";
```

在测试字符串 Paris in the the Spring Spring. 中，有两对重复的单词：

```
my $myTestString = "Paris in the the Spring Spring.";
```

将包含两对重复单词的原始字符串显示给用户：

```
print "The original string was '$myTestString'.\n";
```

在 s/// 中需要使用反向引用(编号变量) $1：

在这个模式中，组件 (\w+) 捕获的第一个单词保存在 $1 中，另一个匹配项则保存在 $2 中——这个实例被丢弃。

g 修饰符的含义是替换所有重复的单词：

```
$myTestString =~ s/$myPattern/$1/g;
```

向用户分别显示第一个捕获组的内容、替换的效果和替换后的结果：

```
print "The captured group was: '$1'.\n";
print "Any doubled word has now been removed.\n";
print "The string is now '$myTestString'.\n";
```

26.4.5 使用交替选择

使用交替选择可以匹配某一个特定的选项。竖线字符 | 用于表示交替选择。

试一试：使用交替选择

(1) 在文本编辑器中输入下列代码，并将文件保存为 Alternation.pl：

```
#!/usr/bin/perl -w
use strict;
my $myPattern = "(Jim|Fred|Alice)";
print "Enter your first name here: \n";
my $myTestString = <STDIN>;
chomp($myTestString);
if ($myTestString =~ m/$myPattern/)
{
print "Hello $&. How are you?";
}
else
{
print "I am sorry, $myTestString. I don't know you.";
}
```

(2) 运行代码，输入名字 Alice 并观察显示的结果。

(3) 再次运行代码，并输入名字 Andrew，然后观察如图 26-24 所示的结果。

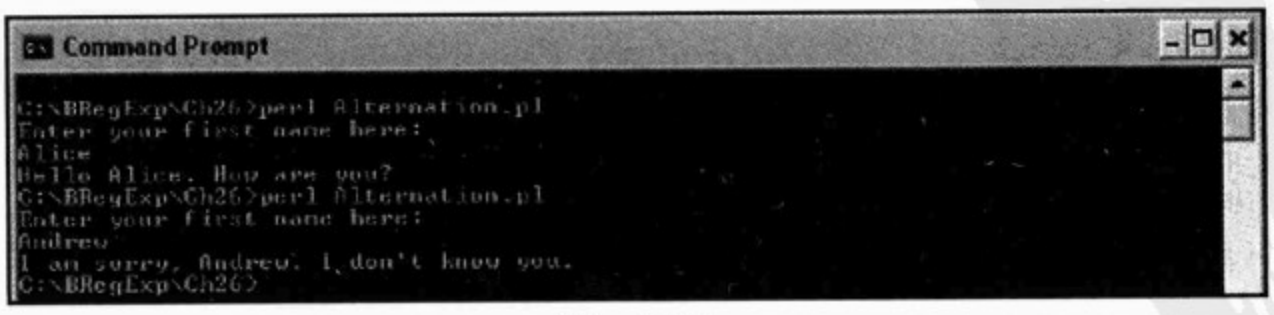


图 26-24

工作原理

给变量 $myPattern 指定一个通过竖线字符将三个直接量模式作为选项的模式：

```
my $myPattern = "(Jim|Fred|Alice)";
```

请用户输入自己的名字：

```
print "Enter your first name here: \n";
```

将自标准输入获取的一行字符指定给变量 $myTestString：

```
my $myTestString = <STDIN>;
```

再使用 chomp() 函数删除变量 $myTestString 末尾的换行符：

```
chomp($myTestString);
```

如果用户输入的名字匹配指定的三个选项中的任何一个，则向用户显示问候信息：

```
if ($myTestString =~ m/$myPattern/)
{
print "Hello $&. How are you?";
}
```

但是，如果输入的名字不是这三个选项中的任何一个，则告诉用户我们不认识他(她)：

```
else
{
print "I am sorry, $myTestString. I don't know you.";
}
```

26.4.6　在 Perl 中使用字符类

Perl 支持相当广泛的字符类功能。如果想指定要匹配的单个字符，可以简单地将这些字符放到一个字符类中。

字符类中的元字符与在字符类之外的含义不同。比如，在字符类外面，元字符 ^ 匹配一个字符串或一行(取决于设置)中第一个字符之前的位置。而在字符类内部，当 ^ 元字符位于左边的方括号之后时，它的含义是对字符类取反。那样，所有 ^ 字符之后的字符都不会匹配。

试一试：使用字符类

(1) 在文本编辑器中输入下列代码，并将文件保存为 CharacterClass.pl：

```
#!/usr/bin/perl -w
use strict;
print "Enter a character class to be used as a pattern: ";
my $myPattern = <STDIN>;
print "\n\nEnter a string to test against the character class: ";
my $myTestString = <STDIN>;
chomp ($myPattern);
```

```
chomp ($myTestString);
print "\n\nThe string you entered was: '$myTestString'.\n";
print "The pattern you entered was: '$myPattern'.\n";
if ($myTestString =~ m/$myPattern/)
{
 print "There was a match: '$&'.\n";
}
 else
{
 print "There was no match.";
}
```

(2) 运行代码。

(3) 输入模式 [A-Z][a-z]*，按回车键。

(4) 输入测试字符串 Hello world!，按回车并观察结果。

(5) 再次运行代码。

(6) 输入模式 [A-E][a-z]*，按回车键。

(7) 输入测试字符串 Hello Ethel. How are you?，按回车并观察如图 26-25 所示的结果。

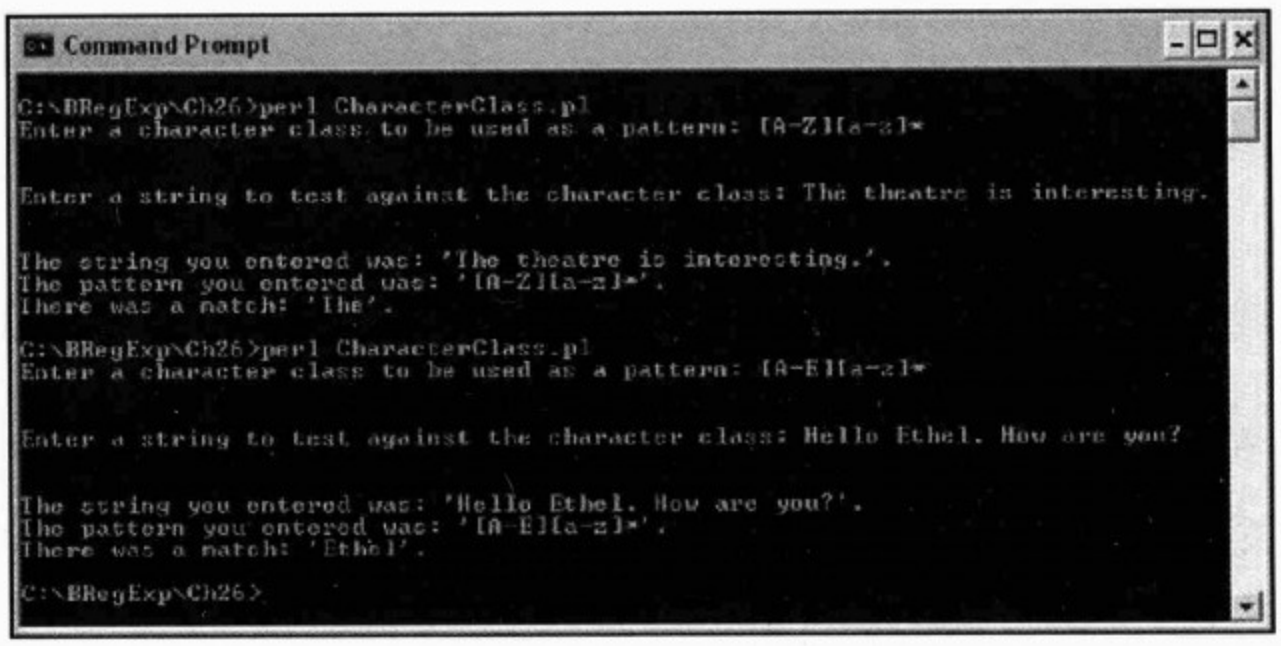


图 26-25

工作原理

首先，请用户输入一个正则表达式模式：

```
print "Enter a character class to be used as a pattern: ";
```

将从标准输入中获得的这一行字符指定给变量 $myPattern：

```
my $myPattern = <STDIN>;
```

再请用户输入测试字符串。将从标准输入中捕获到的测试字符串指定给变量 $myTestString：

```
print "\n\nEnter a string to test against the character class: ";
my $myTestString = <STDIN>;
```

调用 chomp()函数删除变量$myPattern 和$myTestString 末尾的换行符：

```
chomp ($myPattern);
chomp ($myTestString);
```

显示用户输入的测试字符串和模式，以便用户参照。用户也可以从中发现某些错误的输入：

```
print "\n\nThe string you entered was: '$myTestString'.\n";
print "The pattern you entered was: '$myPattern'.\n";
```

然后，if 语句根据匹配过程来确定显示匹配成功还是匹配失败的信息：

```
if ($myTestString =~ m/$myPattern/)
{
```

如果匹配成功，则显示保存在变量 $& 中的匹配内容：

```
print "There was a match: '$&'.\n";
}
```

如果匹配不成功，则会执行 else 子句包含的代码：

```
else
{
print "There was no match.";
}
```

当用户输入模式 [A-Z][a-z]* 时，该模式会匹配一个大写字母字符后跟零个或多个小写字母字符。在测试字符串 Hello world! 中，第一个匹配的字符序列是 Hello。此时的匹配是贪婪的，它会匹配尽可能多的字符。

当用户输入模式 [A-E][a-z]* 时，开始的大写字母字符必须在 A 到 E 之间。因此，Hello 中的 H 不匹配。但是，Ethel 中的 E 与字符类 [A-E] 匹配。而 E 后跟着小写字母字符，所以整个匹配项是 Ethel，如图 26-25 所示。

取反的字符类，指定的是匹配任何一个不包含在方括号中的字符。如果 ^ 元字符是开始方括号后面的第一个字符，那么它的作用就是对字符类取反。

试一试：使用取反的字符类

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
my $myPattern = "[^A-D]\\d{2}";
my $myTestString = "A99 B23 C34 D45 E55";
print "The test string is: '$myTestString'.\n";
print "The pattern is: '$myPattern'.\n";
if ($myTestString =~ m/$myPattern/)
{
 print "There was a match: '$&'.\n";
```

```
}
 else
{
 print "There was no match.";
}
```

(2) 将代码保存为 NegatedCharacterClass.pl。

(3) 运行代码并观察如图 26-26 所示的显示结果。

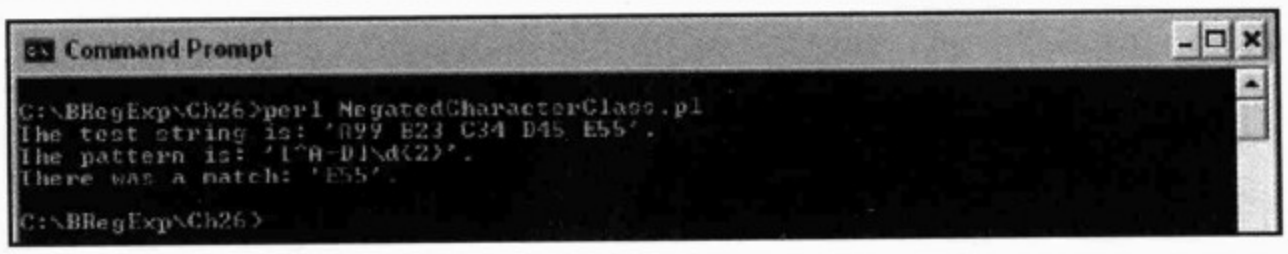


图 26-26

工作原理

指定给变量 $myPattern 的模式是 [^A-D]\d{2}。记住，为保证正确地被识别，必须对 \b 元字符使用双反斜杠。模式 [^A-D]\d{2} 匹配一个不在 A～D 之间的大写字母字符后跟两个数字：

```
my $myPattern = "[^A-D]\\d{2}";
```

将测试字符串指定给变量 $myTestString。注意，测试字符串中前四个字符序列都包含位于 A～D 之间的大写字母字符，而这些字符与取反的字符类不匹配：

```
my $myTestString = "A99 B23 C34 D45 E55";
```

显示测试字符串和模式：

```
print "The test string is: '$myTestString'.\n";
print "The pattern is: '$myPattern'.\n";
```

if 语句通过测试确定是否存在匹配项：

```
if ($myTestString =~ m/$myPattern/)
```

由于取反的字符类 [^A-D] 不会匹配大写的字母字符 A ~ D，所以第一个匹配项是 E55。通过 $& 变量将其显示出来：

```
print "There was a match: '$&'.\n";
```

在本章前面已经看到过在模式中使用置入变量(variable substitution)的例子。置入变量也可以用于字符类中。

试一试：在字符类中使用置入变量

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
```

```
my $toBeSubstituted = "A-D";
my $myPattern = "[$toBeSubstituted]\\d{2}";
my $myTestString = "A99 B23 C34 D45 E55";
print "The test string is: '$myTestString'.\n";
print "The pattern is: '$myPattern'.\n";
if ($myTestString =~ m/$myPattern/)
{
 print "There was a match: '$&'.\n";
}
 else
{
 print "There was no match.";
}
```

(2) 将代码保存为 VariableSubstitutionCharClass.pl。

(3) 运行代码，并观察显示的结果。如图 26-27 所示，匹配项是 A99。

图　26-27

工作原理

本例与 NegatedCharacterClass.pl 类似。不过，字符类的值是由置入的变量 $toBeSubstituted 所提供的。首先，一个能够在方括号中解析出来的值是变量 $toBeSubstituted 的值：

```
my $toBeSubstituted = "A-D";
```

然后，通过将变量 $toBeSubstituted 放在模式开始位置的字符类中将这个值指定给变量 $myPattern：

```
my $myPattern = "[$toBeSubstituted]\\d{2}";
```

本例其余部分的代码与 NegatedCharacterClass.pl 中的代码相同。但是，因为此时使用的字符类是 [A-D]，所以与模式 [A-D]\d{2} 匹配的字符序列是 A99。

26.4.7　使用向前查找

向前查找测试的是跟随在一个模式中某些组件之后的字符序列。Perl 支持肯定式向前查找，也支持否定式向前查找。

肯定式向前查找的语法 (?=……) 用于指定在正则表达式模式另一个组件(即实际要匹配的内容。译者注)的匹配项后面要查找什么。位于向前查找中的字符不会被捕获。

否定式向前查找的语法 (?!……) 用于指定在另一个组件后面必须没有什么，正则表达式才会匹配。

试一试：使用肯定式向前查找

(1) 在文本编辑器中输入下列代码，并将其保存为 LookAhead.pl：

```
#!/usr/bin/perl -w
use strict;
print "Enter a test string here: ";
my $myTestString = <STDIN>;
chomp($myTestString);
if ($myTestString =~ m/Star(?= Training)/)
{
 print "There was a match which was '$&'.";
}
 else
{
print "There was no match.";
}
```

(2) 运行代码。输入测试字符串 I work for Star.，然后按回车，并观察结果。

(3) 再次运行代码。输入测试字符串 I work for Star Training.，然后按回车，并观察结果。如图 26-28 所示，当输入测试字符串 I work for Star. 时没有匹配项，但当输入测试字符串 I work for Star Training. 时，则找到了一个匹配项，即字符序列 Star。

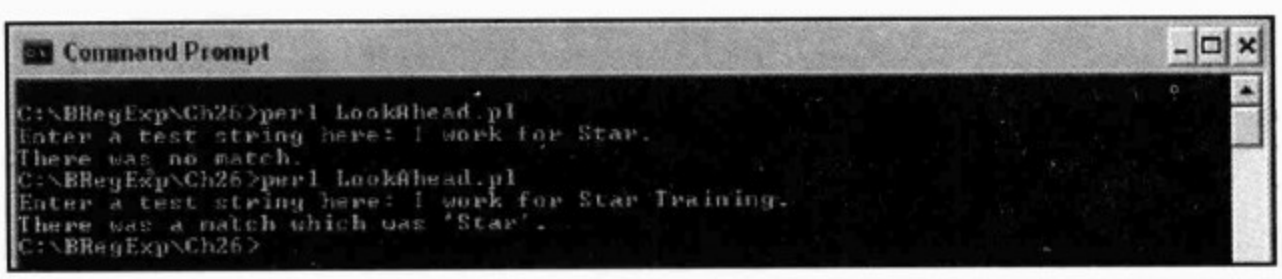


图 26-28

工作原理

用户输入的测试字符串会被指定给变量$myTestString：

```
print "Enter a test string here: ";
my $myTestString = <STDIN>;
```

通过 chomp() 函数删除测试字符串末尾的换行符：

```
chomp($myTestString);
```

if 语句测试变量 $myTestString 的值是否与模式 Star(?= Training) 匹配：

```
if ($myTestString =~ m/Star(?= Training)/)
```

如果字符序列 Star 匹配(在本例中是)，那么向前查找 (?= Training) 会测试 Star 后面是否有一个空格符，而且后跟字符序列 Training。如果是，则匹配成功($& 中只包含匹配项 Star)。

下面的例子示范了如何使用否定式向前查找。

试一试：使用否定式向前查找

(1) 在文本编辑器中输入下列代码，并将其保存为 NegativeLookAhead.pl：

```
#!/usr/bin/perl -w
use strict;
print "Enter a test string here: ";
my $myTestString = <STDIN>;
chomp($myTestString);
if ($myTestString =~ m/Star(?! Training)/)
{
 print "There was a match which was '$&'.";
}
else
{
 print "There was no match.";
}
```

(2) 运行代码。输入测试字符串 I work for Star. 并按回车键，观察结果。

(3) 再次运行代码。输入测试字符串 I work for Star Training. 并按回车键。观察如图 26-29 所示的结果。这一次，第一个测试字符串匹配而第二个测试字符串不匹配。之所以如此是因为(其实也没有什么可奇怪的)，否定式向前查找与肯定式向前查找的结果相反。

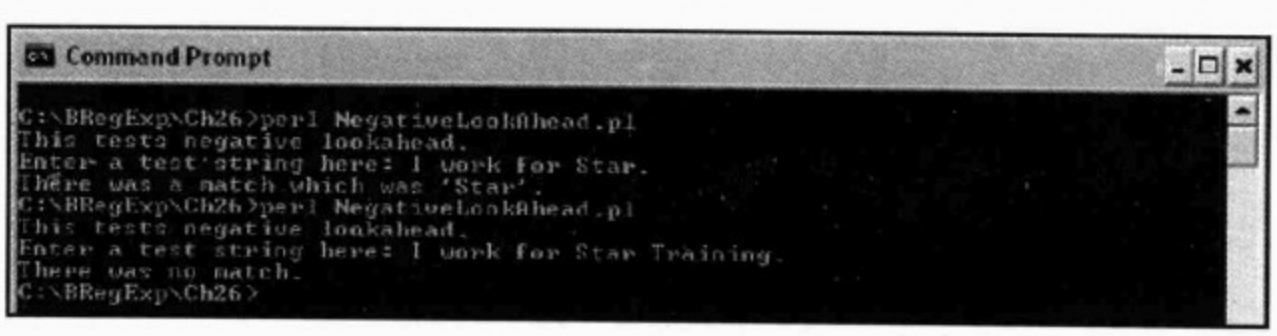


图　26-29

工作原理

本例的关键区别是修改了模式代码——使用了否定式向前查找：

```
if ($myTestString =~ m/Star(?! Training)/)
```

当测试字符串是 I work for Star. 时匹配成功，是因为字符序列 Star 后面没有空格和字符序列 Training。但是，当测试字符串是 I work for Star Training. 时匹配失败，因为发生了向前查找不允许的情况(Star 后面是空格和字符序列 Trainiing)。

26.4.8　使用向后查找

向后查找与向前查找原理相似，只不过向后查找所关心的焦点是位于正则表达式中另一个组件(即实际要匹配的内容。译者注)前面的字符序列。

肯定式向后查找通过语法(?<=……)来指定。否定式向后查找则通过(?<!……)来指定。

试一试：使用向后查找

(1) 在文本编辑器中输入下列代码，并将其保存为 LookBehind.pl：

```
#!/usr/bin/perl -w
use strict;
print "This tests positive lookbehind.\n";
print "Enter a test string here: ";
my $myTestString = <STDIN>;
chomp($myTestString);
if ($myTestString =~ m/(?<=Star )Training/)
{
 print "There was a match which was '$&'.";
}
 else
{
print "There was no match.";
}
```

(2) 运行代码。输入测试字符串 Training is great!，并按回车键，观察结果。

(3) 再次运行代码。输入测试字符串 Star Training is great!，并按回车键，观察如图 26-30 所示的结果。结果正如模式中的肯定式向后查找所指定的那样，只有当字符序列 Training 前面是字符序列 Star 后跟一个空格符时它才会匹配。

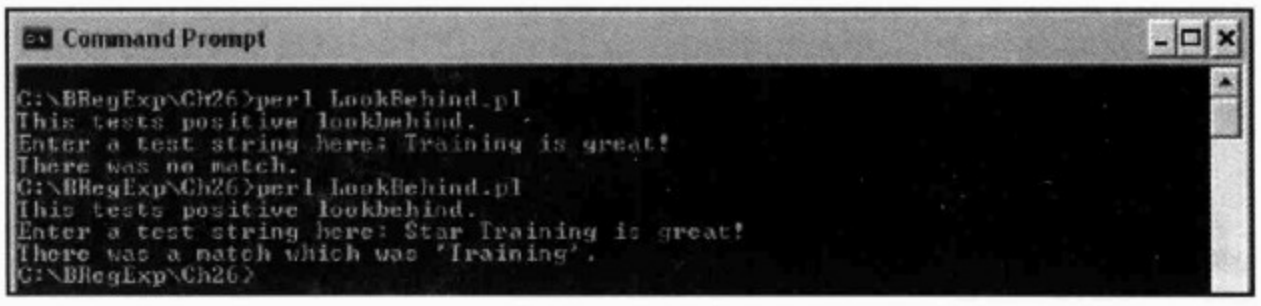


图 26-30

工作原理

本例与前面例子的关键不同在于修改了匹配的模式。注意模式中的向后查找组件 (?<=Star) 位于字符序列 Training 的前面：

```
if ($myTestString =~ m/(?<=Star )Training/)
```

当测试字符串是 Star Training is great! 时存在匹配项，因为在字符序列 Training 前面存在必需的字符序列(Star 后跟一个空格符)。

26.5 在 Perl 中使用正则表达式匹配模式

正则表达式匹配模式使开发人员可以控制如何更有效地应用正则表达式。

表 26-3 中总结了 Perl 中的正则表达式匹配模式。

表 26-3　Perl 中的正则表达式匹配模式

模　式	说　明
i	匹配不区分大小写
x	允许忽略空白符
g	执行全局性匹配
m	将测试字符串当做多行文本匹配
s	将测试字符串当做单行文本匹配

在本章前面的例子中，已经看到了 i(不区分大小写的匹配)和 g(全局性匹配)这两个修饰符的作用了。下面的例子示范如何使用 x 修饰符来辅助说明复杂的正则表达式。

试一试：使用 x 修饰符

(1) 在文本编辑器中输入下列代码，并将其保存为 xModifier.pl：

```
#!/usr/bin/perl -w
use strict;
print "This matches a US Zip code.\n";
print "Enter a test string here: ";
my $myTestString = <STDIN>;
chomp($myTestString);
if ($myTestString =~
    m/\d{5} # Match five numeric digits
    (-\d{4})? # Optionally match a hyphen followed by four numeric digits
    /x)
{
 print "There was a match which was '$&'.";
}
 else
{
 print "There was no match.";
}
```

(2) 运行代码。输入测试字符串 12345 并按回车键。观察显示的结果。

(3) 再次运行代码。输入测试字符串 12345-6789 并按回车键，观察如图 26-31 所示的结果。

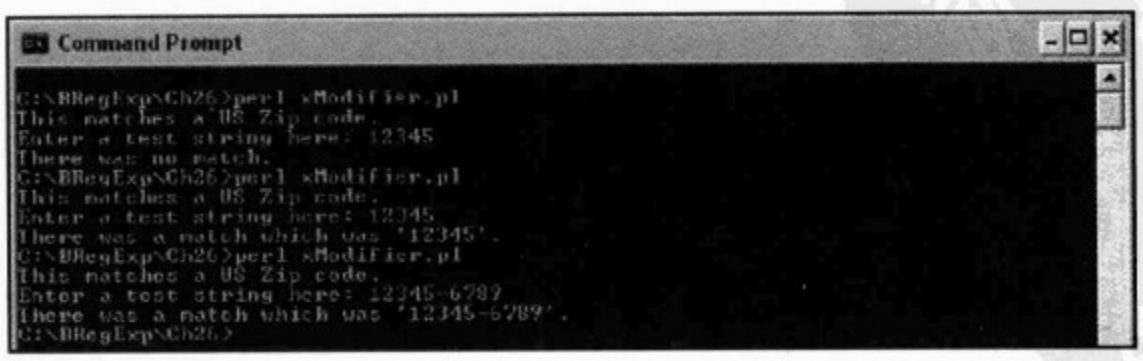


图　26-31

工作原理

xModifier.pl 中的关键是代码中的 m//操作符。注意，下面代码的最后一行中指定了 x 修饰符。它的含义是位于 m// 一对正斜杠中未转义的空白符将被忽略。同样，从 # 开始到该行结尾处的任何字符都将被视为注释。

这样，就可以将模式分散写在多行中，而且在每一行中可以为相应的正则表达式模式的组件添加说明性的注释：

```
if ($myTestString =~
    m/\d{5} # Match five numeric digits
    (-\d{4})? # Optionally match a hyphen followed by four numeric digits
    /x)
```

转义元字符

如果要匹配被用做元字符的字符直接量，那么必须在这些元字符前面加上反斜杠进行转义。

表 26-3 中总结了常见的、必须转义的元字符，假设一对正斜杠被用于正则表达式模式的定界符。

表 26-3 常见的转义元字符

转义的元字符	未转义的元字符
\/ (反斜杠后跟一个正斜杠)	/(正斜杠)
\?	?
*	*
\+	+

在 Perl 中，由于可以自定义正则表达式模式的定界符，所以根据使用的定界符不同，转义的用法也会改变。请看下面的例子。

试一试：使用转义元字符

(1) 在文本编辑器中输入下列代码：

```
#!/usr/bin/perl -w
use strict;
my $myTestString = "http://www.w3.org/";
print "The test string is '$myTestString'.\n";
print "There is a match.\n\n" if ($myTestString =~ m/http:\/\/.*/);
print "The test string hasn't changed but the pattern has.\n";
print "Also the delimiter character is now paired '!' characters.\n";
print "There is a match.\n\n" if ($myTestString =~ m!http://!);
print "The test string hasn't changed and the pattern is the original one.\n";
print "Also the delimiter character is still paired '!' characters.\n";
print "There is a match.\n\n" if ($myTestString =~ m!http:\/\/!);
```

(2) 将代码保存为 EscapedMetacharacters.pl。

(3) 运行代码，并观察如图 26-32 所示的结果。

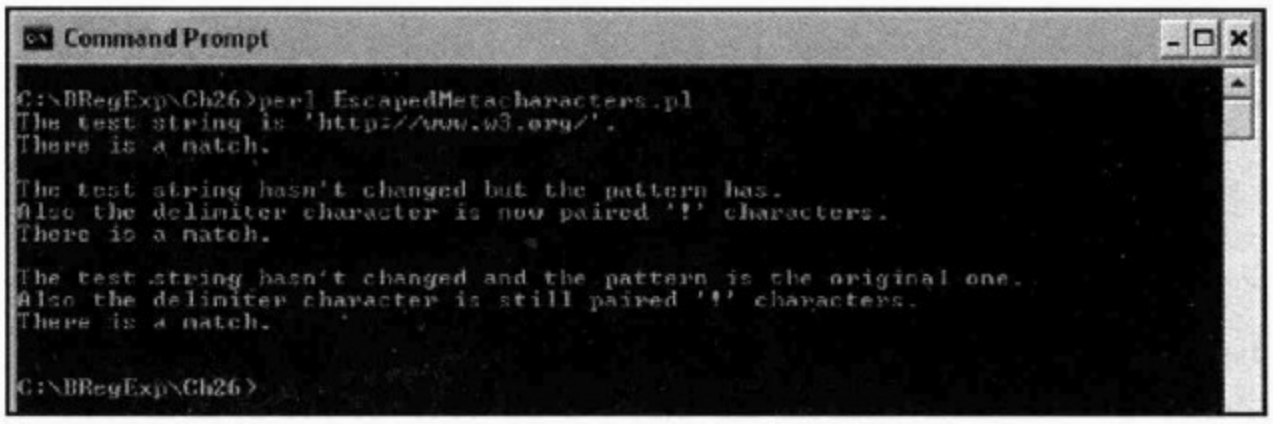


图　26-32

工作原理

第一行代码将一个 URL(指向万维网联盟的网站)指定给变量 $myTestString。注意，与许多 URL 一样，这个 URL 中也包含正斜杠：

```
my $myTestString = "http://www.w3.org/";
```

将测试字符串输出给用户：

```
print "The test string is '$myTestString'.\n";
```

如果指定的模式在测试字符串中存在一个匹配项，那么就会显示相应的信息。注意观察这些模式的构成。其中的每个正斜杠都通过前置一个反斜杠进行了转义。如果尝试在代码中使用没有转义正斜杠的模式 http://.*，一旦运行就会出错：

```
print "There is a match.\n\n" if ($myTestString =~ m/http:\/\/.*/);
print "The test string hasn't changed but the pattern has.\n";
print "Also the delimiter character is now paired '!' characters.\n";
print "There is a match.\n\n" if ($myTestString =~ m!http://!);
print "The test string hasn't changed and the pattern is the original one.\n";
print "Also the delimiter character is still paired '!' characters.\n";
print "There is a match.\n\n" if ($myTestString =~ m!http:\/\/!);
```

26.6　一个简单的 PerlRegex 测试程序

现在已经掌握了一些使用 Perl 正则表达式的技术。你可能会觉得要是有一个简单的 Perl 工具能够根据测试字符串来测试正则表达式会比较方便。RegexTester.pl 恰好提供了这样一种简单的功能。

RegexTester.pl 中包含的代码如下(该文件也在本书的源代码压缩包中)：

```
#!/usr/bin/perl -w
use strict;
print "This is a simple Regular Expression Tester.\n";
print "First, enter the pattern you want to test.\n";
```

```
print "Remember NOT to escape metacharacters like \\d with an extra \\ when you
supply a pattern on the command line.\n";
print "Enter your pattern here: ";
my $myPattern = <STDIN>;
chomp($myPattern);
print "The pattern being tested is '$myPattern'.";
print "Enter a test string:\n";
while (<>)
{
 chomp();
 if (/$myPattern/)
 {
  print "Matched '$&' in '$_'\n";
  print "\nEnter another test string (or Ctrl+C to terminate):";
 }
 else
 {
  print "No match was found for '$myPattern' in '$_'.\n";
  print "\nEnter another test string (or Ctrl+C to terminate):";
 }
}
```

试一试：使用简单的 PerlRegex 测试程序

(1) 在命令行中使用命令 PerlRegexTester.pl 来运行 RegexTester.pl 中的代码。

(2) 输入模式 \d{5}-\d{4}，该模式匹配扩展的美国邮政编码，但不匹配美国邮政编码的简短形式。

(3) 输入测试字符串 12345-6789 并按回车，观察显示的结果。

(4) 再输入测试字符串 12345 并按回车，观察如图 26-33 所示的结果。

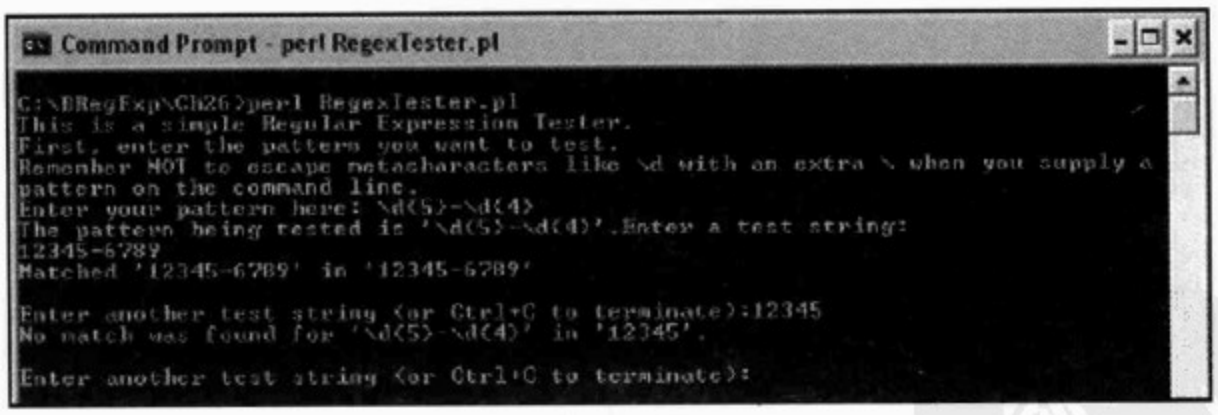


图 26-33

工作原理

首先，显示一些说明性的信息，告诉用户这个程序的用途：

```
print "This is a simple Regular Expression Tester.\n";
print "First, enter the pattern you want to test.\n";
```

有点匪夷所思的是，下面的信息告诉用户在输入模式时，不要转义元字符，比如 \d ，

不过必须对其转义才能使其正确显示。同样，对于反斜杠元字符也是如此：

```
print "Remember NOT to escape metacharacters like \\d with an extra \\ when you
supply a pattern on the command line.\n";
```

然后，指示用户输入一个模式：

```
print "Enter your pattern here: ";
```

使用 <STDIN> 来捕获用户输入的内容。它包含用户输入的模式加一个换行符：

```
my $myPattern = <STDIN>;
```

使用 chomp() 删除不想要的换行符：

```
chomp($myPattern);
```

显示用户其输入的模式，以便用户参照并找到可能存在的输入错误，避免浪费时间用他(她)不想匹配的模式来进行测试：

```
print "The pattern being tested is '$myPattern'.";
```

请用户输入一个测试字符串：

```
print "Enter a test string:\n";
```

使用 < > 表示在用户输入另一行内容时保持循环(< > 将捕获的用户输入保存在默认变量 $_ 中。译者注)：

```
while (<>)
{
```

用户输入的测试字符串末尾有一个不想要的换行符，所以使用 chomp()函数去掉这个换行符(未给 chomp() 函数传递参数意味着操作默认变量 $_。译者注)：

```
chomp();
```

然后，通过 if 语句来测试(在调用 chomp()函数之后)输入的内容中是否存在模式 $myPattern 的匹配项：

```
if (/$myPattern/)
{
```

如果存在匹配项，那么特殊的变量$&中就会包含这个匹配项的值。因此，可以告诉用户哪个字符序列与模式匹配：

```
print "Matched '$&' in '$_'\n";
```

然后，请用户输入另一个测试字符串或者终止程序：

```
 print "\nEnter another test string (or Ctrl+C to terminate):";
}
```

如果没有找到匹配项，则会执行 else 子句中的代码。相应地，用户也会看到一条信息，告诉他(她)们没有找到匹配项，可以继续输入另一个测试字符串或者终止程序：

```
else
{
 print "No match was found for '$myPattern' in '$_'.\n";
 print "\nEnter another test string (or Ctrl+C to terminate):";
}
}
```

26.7 练习

1. 创建一个匹配 16 位信用卡号码的模式，并允许用户选择将数字拆分为四组。在本练习中，假设所有出现在适当位置的数字都是可以接受的。然后，使用 RegexTester.pl 来测试字符串 1234 5678 9012 3456 和 1234567890123456。

2. 修改例子 LookBehind.pl 中的代码，使字符序列 Training 在其前面不是字符序列 Star 后跟一个空格符的情况下匹配。确保编写的代码能够在使用测试字符串 Training is great! 和 Star Training is great!. 测试时显示正确的结果。

附 录

练 习 答 案

第 3 章

1. 可以使用模式 ss 或 s{2}来匹配两个连续的 s。要匹配两个连续的 m，则可以使用模式 mm 或 m{2}。

2. 模式 AB\d\d 或 AB\d{2}能用来匹配指定的文本。

3. 只需修改 UpperL.html 中的一行就可以实现想要的结果。为保持一致，还应该修改 title 元素中的内容以反映更改后的功能。

修改后的文件 UpperLmodified.html 如下所示。其中突出显示的行是经过修改的。突出显示的第二行为变量 myRegExp 指定了模式 the：

```
<html>
<head>
<title>Check for character sequence 'the'.</title>
<script language="javascript" type="text/javascript">
var myRegExp = /the/;

function Validate(entry){
return myRegExp.test(entry);
} // end function Validate()

function ShowPrompt(){
var entry = prompt("This script tests for matches for the regular expression
pattern: " + myRegExp + ".\nType in a string and click on the OK button.",
"Type your text here.");
if (Validate(entry)){
alert("There is a match!\nThe regular expression pattern is: " + myRegExp +
".\n The string that you entered was: '" + entry + "'.");
} // end if
else{
alert("There is no match in the string you entered.\n" + "The regular
expression pattern is " + myRegExp + "\n" + "You entered the string: '" +
entry + "'." );
} // end else
```

```
} // end function ShowPrompt()

</script>
</head>
<body>
<form name="myForm">
<br />
<button type="Button" onclick="ShowPrompt()">Click here to enter text.</button>
</form>
</body>
</html>
```

第 4 章

1. 元字符 . 匹配除换行符外的所有字符。而元字符 \w 则只匹配 ASCII 字母字符(大小写的 A～Z)。

2. 修改过的模式的代码行如下：

```
var myRegExp = /<Person DateOfBirth\s*=\s*".*"\s*>/;
```

修改后的模式在 = 字符两侧添加了模式 \s*。

第 5 章

1. 匹配 license 和 licence 的问题定义大致如下：

匹配一个小写的 l 后跟一个小写的 i，后跟一个小写的 c，后跟一个小写的 e，最后跟一个小写的 n，后跟一个小写的 s 或 c，最后跟一个小写的 e。

下面两个正则表达式模式都是正确的：

```
licen[cs]e
```

或

```
licen[sc]e
```

它们的含义相同。

2. 把这个问题的解决方案分为两部分。

对于月份，有两种情况：第一个数字是 0 或者第一个数字是 1。

当第一个数字是 0 时，那么第二个数字应该是 1～9。反映这种情况的模式可以写成：

```
0[1-9]
```

当月份的第一个数字是 1 时，第二个数字必须是 0、1 或 2。反映这种情况的模式可以写成：

```
1[012]
```

或

```
1[0-2]
```

因为同时需要这两种选项(情况)，所以可以使用圆括号和一个放在这两个选项中间的竖线(|)来指定月份 01～12：

```
(0[1-9]|1[012])
```

对于问题的天数部分，首先是第一个数字为 0 的情况。此时，第二个数字必须位于 1～9 之间。适合的模式是：

```
0[1-9]
```

当天数的第一个数字是 1 时，第二个数字必须是 0～9。适合的模式是：

```
1[0-9]
```

同样，当第一个数字是 2 时，第二个数字也必须是 0～9。适合的模式是：

```
2[0-9]
```

可以将后两种情况组合成下面的模式：

```
[12][0-9]
```

最后，当天数的第一个数字是 3 时，第二个数字必须是 0 或 1。适合的模式是：

```
3[01]
```

以上分析的匹配天数的模式都是互斥的选项，所以可以使用竖线(|)来组合它们：

```
(0[1-9]|[12][0-9]|3[01])
```

因此，整个模式就变成了下面这样：

```
(20|19)[0-9]{2}[-./](0[1-9]|1[012])[-./](0[1-9]|[12][0-9]|3[01])
```

虽然这个模式比题目中的模式有所改进，但不算完美。比如，它不能识别出像 2004/02/30 这样错误的日期(二月份不会有 30 天，所以不允许出现这样的匹配)。是否要继续改进这个模式取决于排除所有错误日期的必要性。

第 6 章

1. 模式 the\> 匹配位于一个单词末尾的字符序列 the。使用 \b 元字符的对应模式是 the\b。

2. 要求编写的模式不能匹配位于字符序列 the 之前的词边界(因为它不匹配 then 中的 the)，而且也不能匹配位于字符序列 the 之后的词边界(因为它不匹配 lathe 中的 the)。因此，

需要一个匹配之前和之后都不是词边界的字符序列 the。

这个问题的定义可以表达如下：

匹配一个不是词边界的位置，后跟字符序列 t、h 和 e，后跟一个不是词边界的位置。

适合的模式为：

```
\Bthe\B
```

图 A-1 显示的是这个模式在 Komodo Regular Expression Toolkit 中匹配的结果。

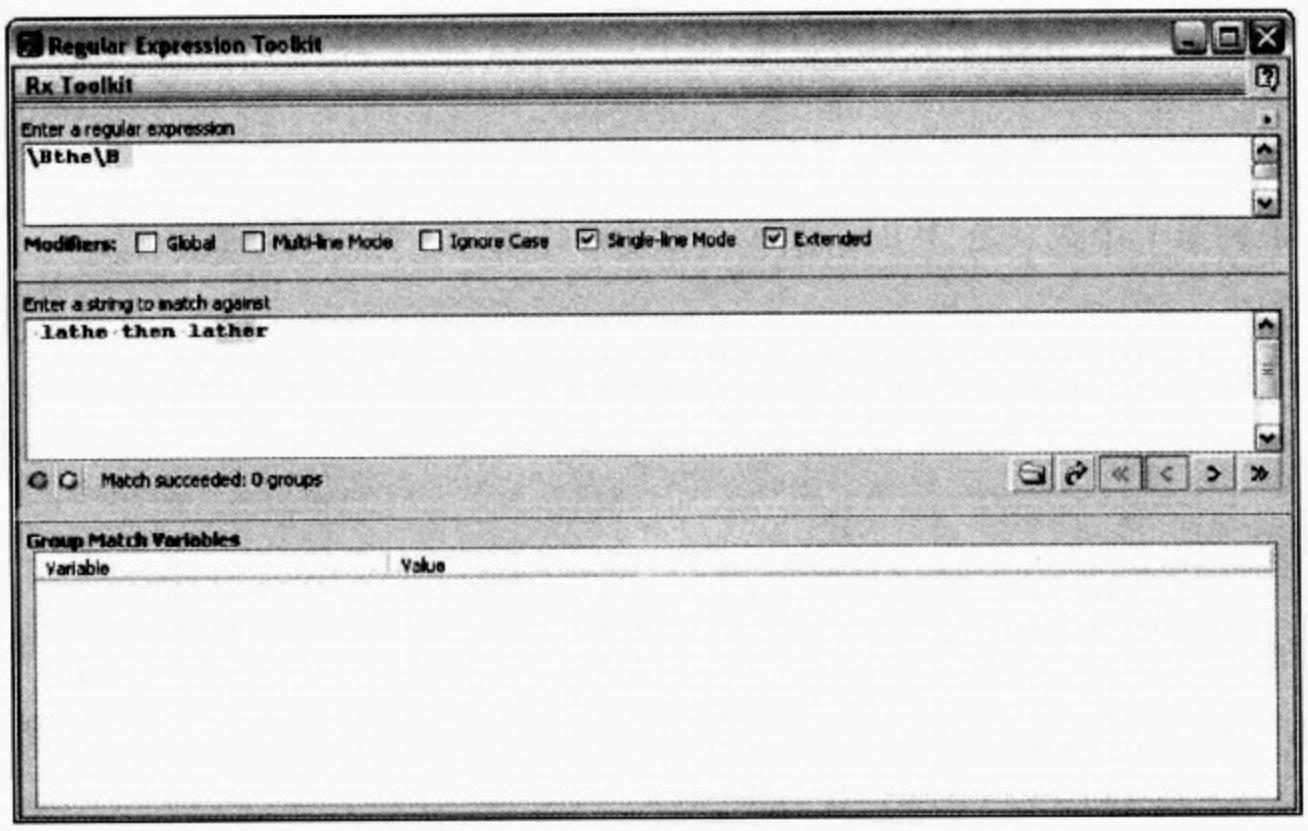


图 A-1

第 7 章

1. 能匹配这两个字符序列的模式有如下三个：

```
(licence|license)
```

或

```
licen(c|s)e
```

或

```
licen[cs]e
```

对于以上这三个模式，交换圆括号或方括号中包含的两个选项也是正确的答案。

2. 下面的模式是唯一的解决方案：

```
(Fear|fear) \1
```

下面的模式只能捕获首字母 f 或 F，因此不能检测到第二个单词是 fear 还是 Fred：

```
(F|f)ear \1
```

下面的模式无效，因为模式中没有圆括号——即什么也不会被捕获。因此反向引用 \1 不起作用：

```
[Ff]ear \1
```

第 8 章

1. 假设不区分大小写，那么前面的字母字符序列可以使用模式 [A-Za-z]+ 来匹配。要指定该字符序列后面必须是一个逗号，可以简单地把直接量逗号放在适当的(肯定式)向前查找圆括号中，如：(?=,)。所以，完整的模式就是 [A-Za-z]+(?=,)。

2. 模式 (?<=\W)sheep(?=\W) 会匹配单词 sheep。向后查找 (?<=\W) 指定 sheep 中 s 前面的字符不能是一个单词字符。而向前查找 (?=\W) 则指定了 sheep 中 p 后面的字符不能是一个单词字符。同时使用向前查找和向后查找的结果就是只能匹配一个单词 sheep。

另外，模式 (?<=[^A-Za-z])sheep(?=[^A-Za-z]) 也可以匹配单词 sheep。

第 9 章

1. 最初的模式是 ^\w*(?<=\w)\.?\w+@(?=[\w\.]+\W)\w+\.\w{3,4}$。其中必须修改的组件很简单，不能再用 \w{3,4} 来匹配位于电子邮件地址中句点字符后面的三个或四个字符，而必须使用交替选择来限定能够出现的特殊域名选项。

在本题中，可以使用交替选择模式 (com|net|org)。注意，不要在圆括号中包含任何空格符，否则可能会丢掉匹配项。因为测试数据中的电子邮件地址不包含空格符。

因此，完整的模式应该是：

```
^\w*(?<=\w)\.?\w+@(?=[\w\.]+\W)\w+\.(com|net|org)$
```

2. 一种方法是在向前查找中添加更多的交替选择选项。下面的模式就会匹配测试文本中的额外字符：

```
Star((?= Training)|(?=\.)|(?=\?)|(?=!))
```

第 11 章

1. 模式 pe[ae]k 可以满足要求。因为 Word 支持字符类。

2. 假设使用模式([0-9]{2})[./-]([0-9]{2})[./-]([0-9]{4})来匹配日期，那么只需替换英国的日期格式即可。

英国的日期格式是 DD/MM/YY。所以要输出国际化的日期格式，可以使用下面这个模式：

```
\3-\2-\1
```

即将最初数据的部件按照想要的顺序进行重排。

第 12 章

1. 模式 [A-V] 是最便捷的解决方案。
2. 模式 [a-ht-z] 是一个理想的解决方案。

图 A-2 显示的是将这个模式应用到测试文件 ClassTest.txt 的结果。

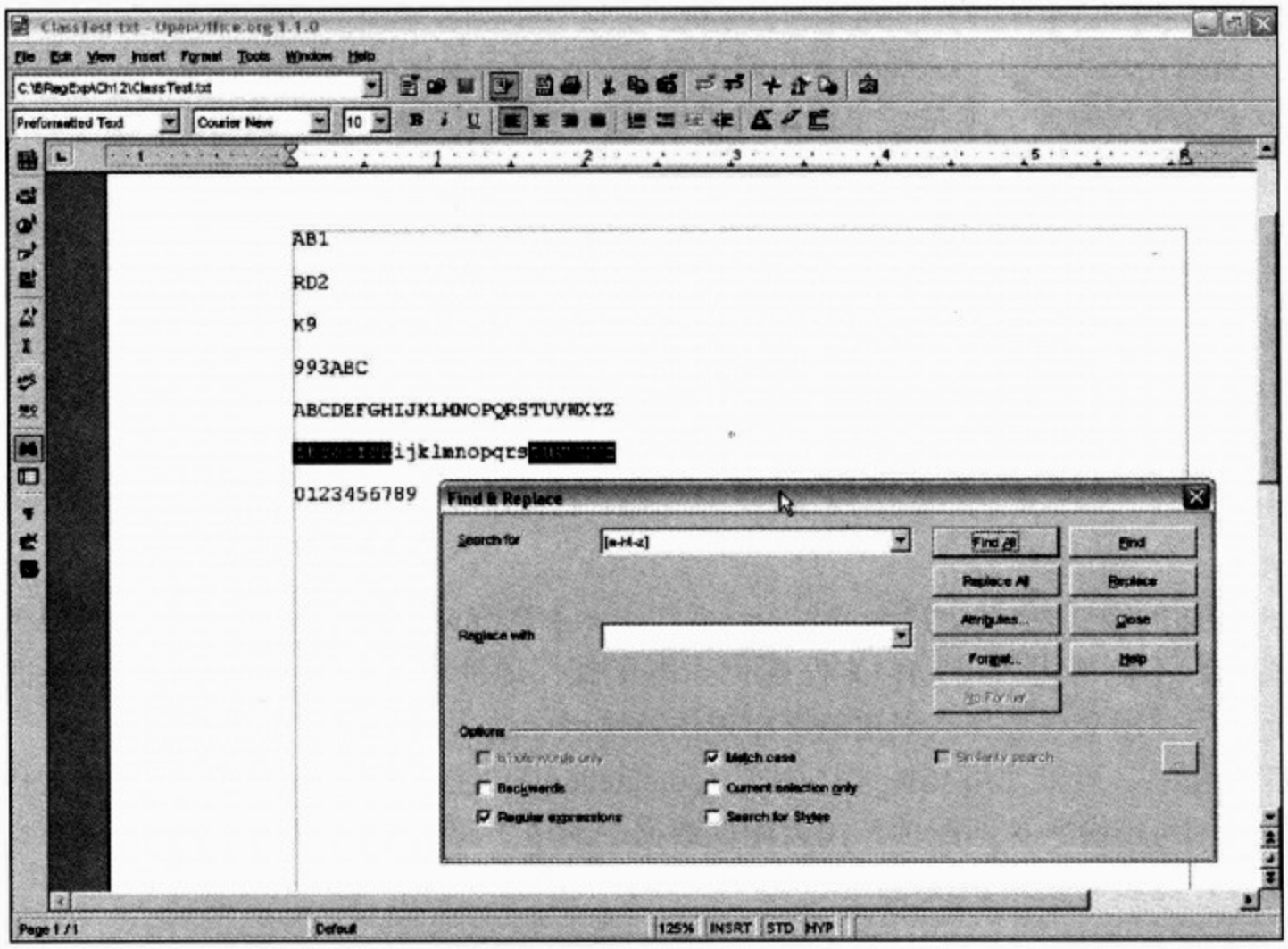


图 A-2

第 13 章

1. 下面的命令能找到包含想要匹配项的行：

```
findstr "[A-Z][A-KO-Z][A-Z][0-9][0-9][0-9]" filename*.extension
```

2. 下面任何一个命令都能找到想要的匹配项：

```
findstr /i "^the" filename*.extension
```

或

```
findstr /i /b "the" filename*.extension
```

第 14 章

1. regexp 不匹配的原因是选项 regexp 在下面选项和其他正则表达式中之前：

```
(regular expression|regex|regexp)
```

如果出现字符序列 regexp，则会匹配 regex。所以模式 regexp 永远不会用到。
要解决这个问题非常简单，只需像下面这样把模式 regexp 挪到前面即可：

```
(regular expression|regexp|regex)
```

同样简单地重新排序也可以解决第二个模式中的问题，如下所示：

```
reg(ular expression|exp|ex)
```

但是，不能以同样的方式来解决第三个模式中的问题：

```
reg(ular expression|(ex)p?)
```

不过可以将其修改为前面出现过的选项：

```
reg(ular expression|exp|ex)
```

如果提供的选项非常接近，要避免此类问题，则一定要保证更"难以匹配"的选项位于更简单的选项之前。

2. 使用下面的模式将会匹配：

```
\$\d{2}\.\d{2}
```

注意，如果不对 $ 元字符转义，它匹配的是行结束的位置。因此，要匹配美元符号，必须使用 \$。另外，也要记住匹配小数点的 \. 只匹配句点字符。

如果使用下面的模式，其中的句点元字符(未转义)将匹配类似 $12345 这样的金额，那不是想要的：

```
\$\d{2}.\d{2}
```

第 15 章

1. 假设已创建筛选器，从 firstname 下拉列表(单元格 C3)选择 Custom 菜单项，在 Custom AutoFilter 对话框左上方的下拉列表中选择 Begins With 选项。在右上方的文本框中输入模式 Kar*，然后单击 OK 按钮。只有 Karen Peters 和 Kara Stelman 所在的行会显示出来。

2. 使用 Ctrl+F 打开 Find and Replace 对话框。在 Find What 文本框中输入模式 Ju?。选中 Match Entire Cell Contents 复选框。单击 Find All 按钮。

第 16 章

1. 在 authors 表中只有一个不是匹配目标的姓 Gringlesby。对于现有的数据来说，可以使用以下几种方案，第一种是：

```
USE pubs
SELECT au_lname, au_fname FROM dbo.authors
WHERE au_lname LIKE 'Gre%'
ORDER BY au_lname
```

模式 Gre%会匹配 Green 和 Greene 但不匹配 Gringlesby。

其他的方案还有：

```
WHERE au_lname LIKE 'Gree%'
```

或者

```
WHERE au_lname LIKE 'Green%'
```

2. 使用下面的 Transact-SQL 代码将得到指定的结果：

```
USE pubs
SELECT title_id, title, pubdate FROM dbo.titles
WHERE title LIKE '%data%'
ORDER BY title
```

其中模式%data%匹配 title 列中相应的值，这些值必须包含字符序列 data，而且 data 前后都可以有零个或多个字符。换句话说，包含字符序列 data 的标题(title)会被匹配。

第 17 章

1. 下面两行代码的功能相同，每个都可以作为本题的答案：

```
SELECT "1950-01-01" LIKE "195%";
```

和

```
SELECT '1950-01-01' LIKE '195%';
```

2. 下面的代码能够实现相应功能：

```
USE BRegExp;
SELECT ID, LastName, FirstName, Skills
FROM Employees
WHERE Skills REGEXP '\.NET'
;
```

注意要对句点字符进行转义——否则，(取决于数据)可能会得到像 Ethernet 这样不想

要的结果。因为句点字符(如果不转义)会匹配 Ethernet 中的 r。

第 18 章

1. 下列代码能够满足题目的要求：

```
SELECT dBeachPurchases.ItemTitle, dBeachPurchases.ItemAuthor
FROM dBeachPurchases
WHERE dBeachPurchases.ItemAuthor LIKE '*B?rns*';
```

这个查询已在 AuctionPurchases.mdb 数据库 B?rns in ItemAuthor 列中创建。

作为替代，? 元字符也可换成 _ 元字符，而且(或者)* 元字符也可以替换成 % 元字符。如果同时替换上述两个元字符，那么会得到如下代码：

```
SELECT dBeachPurchases.ItemTitle, dBeachPurchases.ItemAuthor
FROM dBeachPurchases
WHERE dBeachPurchases.ItemAuthor LIKE '%B_rns%';
```

2. 当回答这个问题时，可能会注意到 Access 缺少一个用于指定前面的字符是可选的限定符。在许多正则表达式实现中，模式 Ma?cDonald 可以匹配 McDonald 或 MacDonald。由于 Access 中不存在这个指定可选项的元字符，必须另想办法。

我们需要匹配 ItemAuthor 列任何记录中的 McDonald 的代码，也需要匹配 ItemAuthor 列任何记录中的 MacDonald 的代码。下面 WHERE 子句是满足第一个要求的代码：

```
WHERE dBeachPurchases.ItemAuthor LIKE "*McDonald*"
```

而下面的 WHERE 子句是能够满足第二个要求的代码：

```
WHERE dBeachPurchases.ItemAuthor LIKE "*MacDonald*"
```

因为我们想同时满足两种可能性，所以可以使用 OR 关键字将前面两个条件联合起来构成一个复合的 WHERE 子句：

```
WHERE dBeachPurchases.ItemAuthor LIKE "*McDonald*" OR dBeachPurchases.ItemAuthor
LIKE "*MacDonald*"
```

如果 WHERE 子句是 SQL 代码中的最后一行，那么还需要在该行结尾处添加表示终止的分号：

a. 在 Database Objects 窗口中，选择 Queries，然后单击 New 按钮。

b. 在 New Query 对话框中选择 Design View。图 A-3 显示的是此时的屏幕外观。

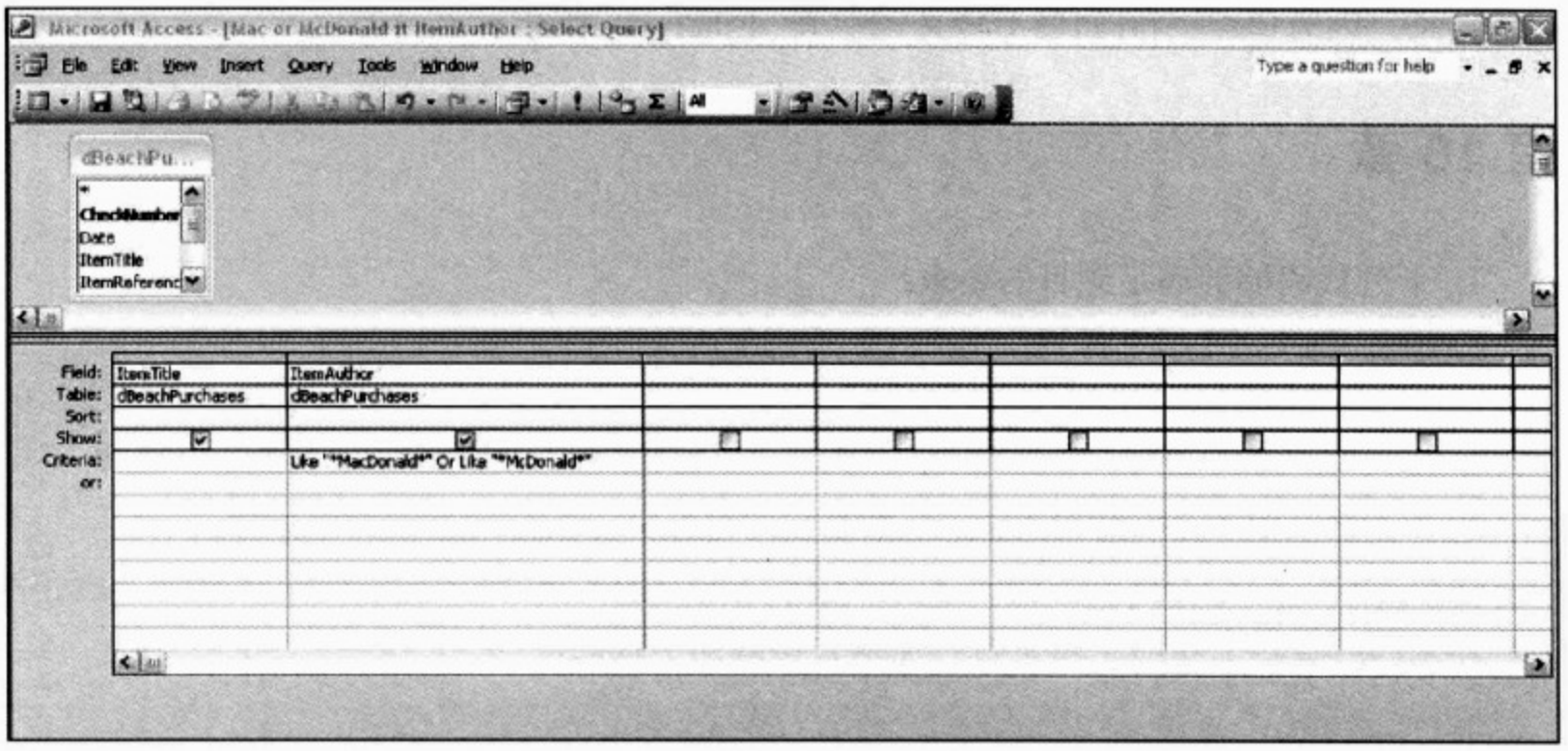


图 A-3

c. 单击 Add 按钮使 dBeachPurchases 表成为查询的对象。在网格中，单击 Field 行最左边的一列，从下拉列表中选择 ItemTitle。

d. 单击 Field 行中的下一列并从下拉列表中选择 ItemAuthor。在同一列中的 Criteria 单元格中输入下列代码：

```
LIKE "*McDonald*" OR LIKE "*MacDonald"
```

e. 使用 Ctrl+S 保存新查询，并将其命名为 Mac or McDonald in ItemAuthor。关闭查询的窗口。

如果稍后在 Design 视图中打开这个窗口，将看到其中的 LIKE 已经被 Access 替换成 like。由于 SQL 是不区分大小写的，因此 Access 的这个替换完全是多此一举，而且如果用户始终都使用大写的 SQL 关键字，此举还可能会导致混乱。

类似地，根据编写的或在 Design 视图中生成的 SQL 代码，Access 还可能会为其添加圆括号。严格来讲，这些圆括号并不是必需的。比如，像下面的代码就是 SQL 视图中显示的添加了圆括号的代码：

```
SELECT dBeachPurchases.ItemTitle, dBeachPurchases.ItemAuthor
FROM dBeachPurchases
WHERE (((dBeachPurchases.ItemAuthor) Like "*MacDonald*")) OR
(((dBeachPurchases.ItemAuthor) Like "*McDonald*"));
```

第 19 章

1. 在 FinalT.html 中，唯一需要修改的地方就是声明变量 myRegExp 的那行代码：

```
var myRegExp = /^\d+$/;
```

模式 ^\d+$ 匹配字符串开始位置(由 ^ 元字符指定)，然后是一个或多个(由 + 元字符

指定)数字(由 \d 元字符指定)，后跟字符串结束位置(由 $ 元字符指定)。

修改后的完成文件 NumericDigitsOnly.html 包含在本书的压缩源文件中。

2. 除了要修改一些次要的细节(比如，修改网页中的 title 元素，使其表达修改后的功能)，我们只需要修改为全局变量 myRegExp 赋值的语句：

这个问题的定义可以表达如下：

匹配一个四位数字符序列，后跟一个空白符，后跟一个四位数字的字符序列，后跟一个空白符，后跟一个四位数的字符序列，后跟一个空白符，再后跟一个四位数的字符序列。

与此定义对应的模式如下所示：

```
\d{4}\s\d{4}\s\d{4}\s\d{4}
```

因此，声明变量 myRegExp 的语句连同注释应该如下所示：

```
var myRegExp = /\d{4}\s\d{4}\s\d{4}\s\d{4}/;
// \d{4} match four numeric digits
// \s match a whitespace character
// \d{4} match four numeric digits
// \s match a whitespace character
// \d{4] match four numeric digits
// \s match a whitespace character
// \d{4} match four numeric digits
```

修改后的文件 CreditCard.html 包含在本书的源代码中。为方便起见，将其内容展示在下面：

```
<html>
<head>
<title>Processing a 16 digit credit card number.</title>
<script language="ecmascript" >
var myRegExp = /\d{4}\s\d{4}\s\d{4}\s\d{4}/;
// \d{4} match four numeric digits
// \s match a whitespace character
// \d{4} match four numeric digits
// \s match a whitespace character
// \d{4] match four numeric digits
// \s match a whitespace character
// \d{4} match four numeric digits
var entry;
function Validate(){
entry = document.simpleForm.CCNBox.value;
if (myRegExp.test(entry)) {
alert("The value you entered, " + entry + "\nmatches the regular expression, " +
myRegExp + ". \nIt is a valid 16 digit credit card number." );
} // end of the if statement
else
{
alert("The value you entered," + entry + ",\nis not a valid 16 digit credit card
```

```
number. Please try again.");
} // end of else clause
} // end Validate() function
function ClearBox(){
document.simpleForm.CCNBox.value = "";
// The above line clears the textbox when it receives focus
} // end ClearBox() function
</script>
</head>
<body>
<form name="simpleForm" >
<table>
<tr>
<td width="50%">Enter a valid 16 digit credit card number here, separating the
groups of four numbers by spaces:</td>
<td width="40%"><input name="CCNBox" width="50" onfocus="ClearBox()" type="text"
value="Enter a credit card number here"></input></td>
</tr>
<tr>
<td><input name="Submit" type="submit" value="Check the Credit Card Number"
onclick="Validate()" ></input></td>
</tr>
</table>
</form>
</body>
</html>
```

第 20 章

1. 修改后的文件 TestForDate.html 的内容如下：

```
<html>
<head>
<title>Test For A Date</title>
<script language="vbscript" type="text/vbscript">
Function MatchDate
Dim myRegExp, TestString, InputString
Set myRegExp = new RegExp
InputString = InputBox("Enter a Date in the format MM/DD/YYYY")
myRegExp.Pattern = "(0[1-9]|1[012])[-/.](0[1-9]|[12][0-9]|3[01])[-/.]\d{4}"
TestString = InputString
If myRegExp.Test(TestString) = True Then
  MsgBox "The test string '" & TestString & "' matches the pattern '" &
myRegExp.Pattern & "'."
Else
  MsgBox "There is no match. '" & InputString & "' does not match " &VBCrLf _
  & "the pattern '" & myRegExp.Pattern & "'."
End If
End Function
```

```
</script>
</head>
<body onload="MatchDate">

</body>
</html>
```

这只是满足练习题目要求的解决方案之一，还有很多其他的方案。

将函数的名称修改为 MatchDate，并额外声明一个变量 InputString 用于接收 VBScript 的 InputBox()函数返回的输入文本：

```
InputString = InputBox("Enter a Date in the format MM/DD/YYYY")
```

InputBox() 函数只是简单地接受一个用户输入的字符串，同时显示给用户应该输入的格式为 MM/DD/YYYY 的指示信息。

为 myRegExp 对象的 Pattern 属性指定一个匹配目标格式日期的模式：

```
myRegExp.Pattern = "(0[1-9]|1[012])[-/.](0[1-9]|[12][0-9]|3[01])[-/.]\d{4}"
```

日期的月份部件必须匹配下列模式：

```
(0[1-9]|1[012])
```

这个模式的第一个选项中包含了 1～9，第二个选项中包含了 10～12。

日期各部件间的分隔符可能是一个连字符、正斜杠或一个句点字符，可以通过下面的字符类表示：

```
[-/.]
```

注意，不要使用下面这个字符类，因为该字符类中的连字符表示的是一个范围，而这不是想要的值：

```
[/-.]
```

日期的天数部件必须匹配下面的模式：

```
(0[1-9]|[12][0-9]|3[01])
```

其中第一个选项匹配 1～9，第二个选项匹配 10～29，第三个选项匹配 30～31。

在指定给 Pattern 属性的正则表达式中，日期的年份部件简单地通过 \d{4} 来表示，该组件会匹配 0000 ～ 9999。因此，很可能希望把允许的日期范围限定在更接近当前的日期范围内，同时为匹配日期中的年份部件提供一个更具体的模式。

2. 下面显示的文件 NameReverseStricter.html 中的内容，是对本题的一种答案。在这些代码中有一些无关紧要的小改动，这里不做介绍了。我们的焦点是如何构建正则表达式以及如何和 RegExp 对象的属性结合使用这个模式：

```
<html>
<head>
```

```
<title>Reverse Surname and First Name</title>
<script language="vbscript" type="text/vbscript">
Function ReverseName
Dim myRegExp, TestName, Match
Set myRegExp = new RegExp
myRegExp.Pattern = "^([A-Za-z]+)(\s+)([A-Za-z]+)$"
TestString = InputBox("Enter your name below, in the form" & VBCrLf & _
"first name, then a space then last name." & VBCrLf & "Don't enter an initial or
middle name."_
& "Any extra information will result in an error.")
Match = myRegexp.Replace(TestString, "$3,$2$1")
If Match <> TestString Then
MsgBox "Your name in last name, first name format is:" & VBCrLf & Match
Else
MsgBox "You didn't enter your name in the format requested." & VBCrLF _
& "You may have entered no data, omitted part of your name," & VBCrLf _
& "or entered extra data." & VBCrLf & VBCrLf _
& "Press OK then F5 to run the example again."
End If
End Function

</script>
</head>
<body onload="ReverseName">

</body>
</html>
```

主要修改的是指定给 myRegExp 对象的 Pattern 属性的字符序列：

```
myRegExp.Pattern = "^([A-Za-z]+)(\s+)([A-Za-z]+)$"
```

本题规定，只允许匹配字母字符序列。由于 \w 元字符也匹配数字和下划线字符(某些假定的人名中包含该字符)，所以它是不合适的。如果只想匹配字母字符，那么可以使用前面所示的模式。

另一种替代方法是将 IgnoreCase 属性的值设置为 True，同时使用只包含一种大小写形式的字母字符类(或者各使用一次)，比如下面的代码中，就使用了两次 [A-Z] 字符类：

```
myRegExp.IgnoreCase = True
myRegExp.Pattern = "^([A-Z]+)(\s+)([A-Z]+)$"
```

还需要修改下面这一行代码：

```
If Match <> TestString Then
```

该行代码以前是：

```
If Match <> "" Then
```

通过将控制 If 语句的条件修改为比较变量 Match 和 TestString 的值，就可以确定是否存在一个匹配项，并且根据情况显示重新组合的名字(当存在匹配项时)，或者显示输入

的格式不正确的信息(当不存在匹配项时)。

第 21 章

1. 为了练习，复杂的解决方法中同时使用了向前查找和向后查找。

模式 (?<=\b[A-Za-z])old\b 只有当字符序列 old 前面是一个单独的字母字符(任意大小写)并且后面没有任何字母字符(即，后跟一个 \b 元字符所匹配的词边界)时才会匹配。因此，该模式只会匹配如 bold、cold、fold、gold 和 hold 等这样的单词。

为了使它也匹配像 scold 或 scolds 这样的单词，可以使用模式 (?<=\b[A-Za-z]+)old(?=[A-Zaz]*\b)。

如果还要匹配单词 golden 中的 old，那么可以再将模式修改为 (?<=\b[A-Za-z]*)old(?=[A-Za-z]*\b)。

但是，更简单的解决方案是使用 \B 元字符，该元字符匹配一个不位于单词字符和非单词字符之间的位置(即非词边界位置或单词内部字母间的位置。译者注)。因此，模式 \Bold 匹配一个非词边界位置后跟字符序列 old。由于 \B 匹配一个非词边界的位置，所以 old 中 o 之前的字符必须是一个单词字符，因此单词 cold 和 bold 中的 old 都将匹配。

2. 通过反向引用可以有多种方式解决这个问题。按照下面的步骤就可以找出本题的答案：

a. 在 Visual Studio 2003 中创建一个新控制台程序的项目，并将项目命名为 Replace Demo。

b. 将代码修改成如下所示：

```
Imports System.Text.RegularExpressions
Module Module1
    Dim myPattern As String = "(Doctor|Doc)"
    Dim myRegex = New Regex(myPattern)
    Sub Main()
        Console.WriteLine("This example replaces 'Doctor' or 'Doc' with 'Dr.'.")
        Console.WriteLine("Enter a string on the following line:")
        Dim inputString = Console.ReadLine()
        Dim myMatch = myRegex.Match(inputString)
        Dim myReplacedString = myRegex.Replace(inputString, "Dr.")
        Console.WriteLine("The match '" & myMatch.Value & "' was found.")
        Console.WriteLine("The amended string is: " & myReplacedString)
        Console.WriteLine("Press Return to close this application.")
        Console.ReadLine()
    End Sub

End Module
```

c. 保存代码并按 F5 运行代码。

d. 用字符串 I know Doc Smith. 和 I know Doctor Smith 来测试代码。

在这个包含交替选择的模式中，要确保把较长的模式放在前面；否则，将只会匹配

Doc，Doctor 将永远不会被匹配：

```
Dim myPattern As String = "(Doctor|Doc)"
```

通过调用 Regex 对象的 Replace()方法来完成替换操作：

```
Dim myReplacedString = myRegex.Replace(inputString, "Dr.")
```

在定义变量 myRegex 时将模式指定给该变量：

```
Dim myPattern As String = "(Doctor|Doc)"
```

当以这种方式来使用 Replace()方法时，需要传递给该方法的两个参数是输入字符串和替换文本。

第 22 章

1. IgnoreCase 选项用于指定不区分大小写的匹配。

第 23 章

1. 不是。Perl 兼容正则表达式是从 PHP 3.0.9 才开始引入的。但到 PHP 4.2 时，才默认启用了 Perl 兼容正则表达式。

2. x 修饰符会使模式中未转义的空白符被忽略。这样就可以允许模式写在多行中，同时可以在每一行中添加注释，以说明该行中模式组件的意图和作用。

第 24 章

1. 下面所示的文件 Name2.xsd 是一种可能的方案，它允许像 Maria Von Trapp 和 John James Manton 这样的名字作为 Name 元素的内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="Names">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" maxOccurs="unbounded">
          <xs:simpleType>
           <xs:restriction base="xs:string">
            <xs:pattern value="\w+\s+\w+(\s+\w+)?" />
           </xs:restriction>
          </xs:simpleType>
          </xs:element>
        </xs:sequence>
```

```
      </xs:complexType>
    </xs:element>
</xs:schema>
```

代码中已经将模式修改为 \w+\s+\w+(\s+\w+)?。其中，(\s+\w+)? 匹配一个可选的、包含一个或多个空白符并后跟一个或多个单词字符的字符序列。当 Name 元素的内容是像 John Smith 这样的名字时，模式 (\s+\w+)? 匹配零长度字符串。当 Name 元素的内容是像 John James Manton 这样的名字时，模式 (\s+\w+)? 则匹配 James 后面的空白符和字符序列 Manton。

2. 示例中的零件编号显示，它们都由 1~3 个字母字符后跟 1~5 个数字组成。

一个使用 Unicode 字符类的 W3C XML Schema 文档—— PartNumbers.xsd(内容如下)，可以作为答案：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="PartNumber">
    <xs:simpleType>
      <xs:restriction base="xs:string">
<xs:pattern value="\p{L}{1,3}\p{Nd}{1,5}" />
      </xs:restriction>
    </xs:simpleType>
</xs:element>
<xs:element name="PartNumbers">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="PartNumber" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

模式 \p{L}{1,3}\p{Nd}{1,5} 假设大小写字母都是可接受的。其中，\p{L} 匹配任何 Unicode 字母，包括大小写。而 \p{Nd} 匹配一个数字。通过前面代码中所示的限定符，整个模式将能够匹配所有零件编号。

假设只能接受大写的字母字符，那么模式 \p{Lu}{1,3}\p{Nd}{1,5} 将成为一个可接受的答案。

假设只能接受基本拉丁语字符，那么可以像本章前面所示范的那样使用\p{IsBasicLatin}和\p{L}{1,3}\p{Nd}{1,5}或\p{Lu}{1,3}\p{Nd}{1,5}的交集。

第 25 章

1. replaceAll.java 中的代码使用 Matcher 类的 replaceAll()方法来替换所有的匹配项。如果只想替换第一个匹配项，一个简单的方法是使用 replaceFirst()方法，而不是使用 replaceAll()方法。

因此，只需修改下面这一行代码：

```
String testResult = myMatcher.replaceAll("Moon");
```

修改为使用 replaceFirst()方法：

```
String testResult = myMatcher.replaceFirst("Moon");
```

将修改后的代码保存为 replaceFirst.java，因此，也需要修改类名：

```
public class replaceFirst{
```

运行这些代码之后的结果如图 A-4 所示，其中只有第一个字符序列 Star 被替换了。

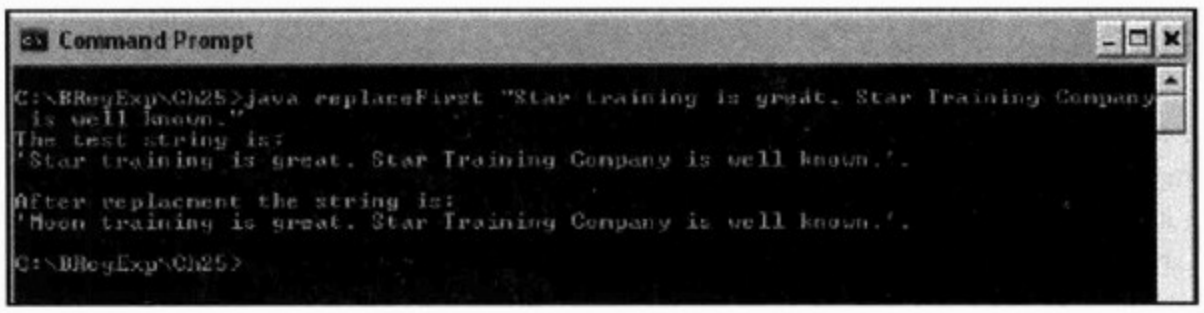


图 A-4

2. 同样，还是只需要修改一行代码(不算其他次要细节的修改)，即将 String 类的 replaceFirst()方法替换成 replaceAll()方法。

要修改的关键代码行如下：

```
String testResult = testString.replaceAll(myRegex, "TWINKLE");
```

修改代码后的运行结果是正确的。不过，除修改方法名称外还修改了其他地方(比如类名。译者注)以反映修改后的功能。

下面是修改后的代码。将这些代码保存为 stringReplaceAll.java：

```
import java.util.regex.*;

public class stringReplaceAll{
 public static void main(String args[]){
  myReplaceAll(args[0]);
 } // end main()

 public static boolean myReplaceAll(String testString){
   String myRegex = "twinkle";
   String testResult = testString.replaceAll(myRegex, "TWINKLE");

   System.out.println("The string was: '" + testString + "'.");
   System.out.println("The regular expression pattern was: '" + myRegex +"'.");
   System.out.println("After replacement the string was: '" + testResult + "'.");
          return true;
 } // myReplaceAll()

}
```

如果在编译完 stringReplaceAll.java 后，在命令行中输入下列命令执行代码，相应的显示结果将如图 A-5 所示。

```
java stringReplaceAll "twinkle, twinkle little star."
```

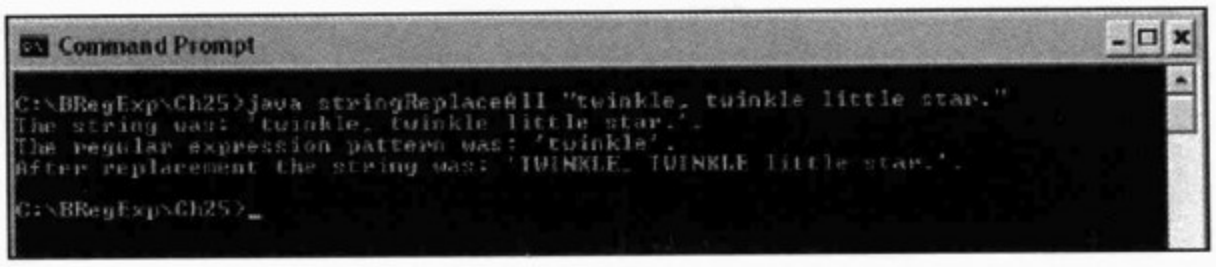


图 A-5

第 26 章

1. 有助于解决这个问题的第一件事就是将题目的要求分解为相应的问题定义。那么，可行的问题定义可以描述如下：

匹配一个四位数字的字符序列后跟一个可选的空格符，后跟四位数字，后跟一个可选的空格符，后跟四位数字，后跟一个可选的空格符，后跟四位数字。

问题定义描述了一个明显重复的模式——“一个四位数字的字符序列后跟一个可选的空格符”，该模式重复了三次。对应这个重复的正则表达式组件的模式如下所示：

```
(\d{4}\s*){3}
```

正则表达式的最后一个组件是简单的四位数字：

```
\d{4}
```

由于除了信用卡号码，不想接收别的字符，所以可以使用 ^ 元字符和 $ 元字符将整个模式包围起来，以便得到满足要求的字符序列。

将以上所有组件放到一起，就得到了如下模式：

```
^(\d{4}\s*){3}\d{4}$
```

图 A-6 显示的是测试完题目要求的两个测试字符串之后的屏幕外观，同时还进一步测试了在两组四位数中间包含两个连续空格符的测试字符串。

2. 要产生满足题目要求的代码，只需要修改 LookBehind.pl 代码中的一个字符。

这行代码是：

```
if ($myTestString =~ m/(?<=Star )Training/)
```

在 LookBehind.pl 需要将其改成：

```
if ($myTestString =~ m/(?<!Star )Training/)
```

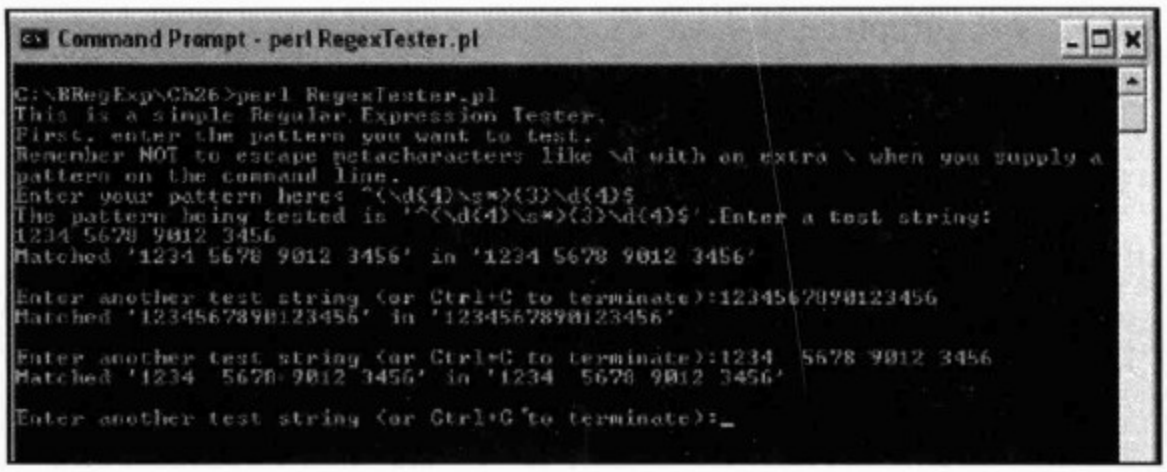


图 A-6

这样就变成了否定式向后查找。

包含在本书源代码中的修改后的代码文件是 NegativeLookBehind.pl。图 A-7 显示的是在命令行中运行这些代码并测试题目要求的两个测试字符串之后的界面外观。

图 A-7

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

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

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

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

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

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

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

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

PROGRAMMER TO PROGRAMMER™

Beginning Regular Expressions

正则表达式能够帮助用户和开发人员更加有效地查找和操纵文本内容。而且，正则表达式已经得到了许多脚本语言、编程语言和数据库的良好支持。这本示例丰富的教程将打破所谓正则表达式难以掌握的传统神话。本书详细解释了正则表达式的各个组成部分、这些组成部分的含义、如何使用它们，以及在编写正则表达式时如何避免常见的错误。

通过逐章地讲解如何在流行的 Windows 平台的软件——包括数据库、跨平台的脚本语言和编程语言中使用正则表达式，你将学习到如何有效地驾驭正则表达式所提供的强大功能，并且全面理解正则表达式的高度灵活性和无限潜能。

本书主要内容

- 正则表达式的基本概念以及如何编写正则表达式
- 如何分解文本操作问题并构建符合逻辑的正则表达式模式
- 如何在不同的脚本或编程语言以及软件包中使用正则表达式
- 当前各种正则表达式实现之间存在的差别
- 可以解决日常问题的、可重用的正则表达式示例代码

本书读者对象

本书适用于那些需要解决文本操作问题，但还不了解正则表达式的开发人员。虽然一些基本的编程或脚本编写经验是有用的，但并不是必需的。

本书技术支持

从 Web 站点 www.wrox.com 和 www.tupwk com.cn/downpage 上可以获取本书的源代码和 Wrox 技术支持。

Wrox Beginning guides are crafted to make learning programming languages and technologies easier than you think, providing a structured, tutorial format that will guide you through all the techniques involved.

p2p.wrox.com The programmer's resource center

www.wrox.com

ISBN 978-7-302-18382-2

9 787302 183822 >

定价：79.99 元

wrox™ An Imprint of WILEY

上架建议 | 编程技术 | 软件开发