O'REILLY®

原书第2版

社交网站的数据挖掘与分析

MINING THE SOCIAL WEB

Matthew A. Russell 著
苏统华 魏通 赵逸雪 王烁行 刘智月 译

机械工业出版社
China Machine Press

O’Reilly精品图书系列

社交网站的数据挖掘与分析（原书第2版）

Mining the Social Web，Second Edition

（美）拉塞尔（Russell，M.A.） 著

苏统华 等译

ISBN：978-7-111-48699-2

本书纸版由机械工业出版社于2015年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

斧子钝了，其刃不再锋利，必多费力气；但得智慧指教，方可奏效。

——《传道书》

O’Reilly Media，Inc.介绍

O’Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

让你决定捧起这本书的，极可能是这本书的主题。毋庸置疑，社交网络已经深入人心。社交网络引发了交往方式的历史性变化，这是当今时代最大的一次变革。Facebook、Twitter、LinkedIn、Google+、GitHub等社交网站的非凡意义在于改变了互联网的生态，将自发、无序状态的互联网引入到一个有组织、有阶层甚至实名制的社交网络之中。

社交网站数据蕴藏着大量的价值和见解。这些数据随着岁月流逝，会如同醇酒一般越发芳香。社交网络与物联网也在不断融合，大数据为统计学习方法提供了广阔的舞台。随着数据爆炸不断升级，庞杂的大数据中夹杂的噪声有可能被放大。只有利用合适的工具，才能准确而快速地挖掘出感兴趣的或者有意义的知识。

Matthew A.Russell是挖掘社交网络数据的资深专家。他深谙数据挖掘的各种工具和技术，同时他对数据挖掘初学者的所需所求了如指掌。你正在阅读的这本书是在第1版基础上作了“重大更新”的第2版，其中吸收了第1版读者的大量有建设性的意见。如果你对社交网站数据感兴趣，那么本书是帮助你快速入门的法宝。本书体例完备、特色鲜明，从实用的角度出发，对主流的各种社交网站做了较全面覆盖。本书各个章节之间也保持着一定的独立性，如果你只对特定章节的技术感兴趣，也

可以直接跳到对应内容阅读。不论你关注的是Facebook、Twitter、LinkedIn、Google+、GitHub、邮箱、网页还是语义网，本书都可以传授给你爬取数据、分析数据以及展示数据的技术。特别值得一提的是，本书配套的代码借助了IPython Notebook，让你可以快速配置自己的开发环境并享受交互式学习的乐趣。强烈建议你配合书中的虚拟机来学习！

本书的翻译由苏统华全程组织。魏通、赵逸雪、王烁行以及刘智月协助苏统华完成了全书的译文初稿。其中本书的前言、前四章的初稿由魏通、王烁行和苏统华共同完成，接着的第5章到第7章的初稿由赵逸雪和苏统华共同翻译，第8章的初稿由刘智月和苏统华完成，第9章的初稿由赵逸雪和苏统华协作完成，最后的附录由王烁行和苏统华共同翻译。在初稿的基础上，翻译团队进行了交叉核对，并最终由苏统华统一定稿。

本书从启动翻译到进入出版流程历时整整一年，在此过程中，得到很多同事、朋友和编辑团队的热心帮助，在此表达我们深深的谢意。另外，本书的翻译还得到了多个项目的资助，在此一并致谢。国家自然科学基金（资助号：61203260）、黑龙江省科研启动基金（资助号：LBH-Q13066）、哈尔滨工业大学科研创新基金（资助号：HITNSRIF2015083）对本书的翻译提供了部分资助。最后，黑龙江省教育厅高等教育教学改革项目（资助号：JG2013010224）、哈尔滨工业大学研究生教育教学改革研究项目（资助号：JCJS-201309）也对本书的

翻译提供了大力支持。

本书涉及的技术较广，鉴于译者水平有限，译文中难免存在一些问题，真诚地希望读者朋友将你的意见发到译者邮箱 tonghuasu@gmail.com。

苏统华

哈尔滨工业大学软件学院

2014年10月20日

译者简介

苏统华　博士，硕士生导师，CUDA研究中心以及教学中心负责人。主要研究方向包括：物联网大数据智能信息处理、大规模并行计算、模式识别、智能媒体交互与计算等。作为自然手写中文文本识别的开拓者，四年内代表工作被同行大篇幅他引约300次；他所建立的HIT-MW库为全世界100多家科研院所采用；目前负责国家自然科学基金项目2项。2013年，他领导的研究组在文档分析和识别国际会议（ICDAR’2013）上获得手写汉字识别竞赛的双料冠军；2014年，两项手写文字识别核心技术授权给某高新技术公司，正在为超过200万终端用户提供技术服务。著有英文专著《Chinese Handwriting Recognition：An Algorithmic Perspective》（德国施普林格出版社），出版5本大数据分析方面的译作（机械工业出版社）。

前言

与其说网络是一项技术创新，不如说它是一项社交创举。

我设计它意在延伸社交性（帮助大家一起工作），而不是为了制造一种高科技玩具。网络的终极目标是支持并改进现实世界的网络化生存。现实世界中，我们会组成家庭、组织协会、组建公司。现实世界中，我们会跨越空间的樊篱建立信任，亦会近在咫尺却心生芥蒂。

——Tim Berners-Lee（万维网之父），《Weaving the Web》（Harper）

读者必读

本书经过精心设计，为特定的目标受众提供一段难以忘怀的学习体验。那些影响心情的电子邮件、糟糕的书评或者其他误导，可能让你对本书的范围和目的产生不必要的误解。为了避免这些混乱，本前言的余下内容试图帮助你确定你是否是该书的目标受众。作为一位非常繁忙的职场人士，我认为时间是我最宝贵的财富，并且我认为对你也是这样的。尽管我经常遭遇失败，但是当我走出困境时，我真的尽力向我的邻居致敬。本前言是我试图向你（读者）致敬，我的致敬方式是清楚阐释

本书能否满足你的期望。

管理你的期望

首先，本书假设你希望学习如何挖掘来自流行社交网络资源中的数据，避免在运行示例代码时遇到技术问题并且在过程中获得很多乐趣。尽管你读这本书仅仅可能是为了了解社交网络挖掘可能做什么事情，但你应该知道本书的写作风格。本书组织成让你可以跟随本书尝试许多练习，并且一旦完成了一些安装开发环境的简单步骤就能进入数据挖掘者的行列。如果你以前编写过一些程序，应该会发现可以轻松运行这些示例代码。即使你以前从未编过程但认为自己的技术领悟能力还可以，我敢说你可以将这本书作为一次难忘旅程的出发点，它将以你从未想象的方式扩展你的思想。

为了充分享受本书及其所提供的内容，你需要对挖掘流行社交网络（如Twitter、Facebook、LinkedIn和Google+）存储数据的广阔可能性很感兴趣，需要主动下载一个虚拟机并且在IPython Notebook上重现本书的示例代码。IPython Notebook是一个奇妙的基于网络的工具，每一章的示例代码都基于它。执行这些代码通常和在键盘上按一些键一样容易，因为所有的代码都是以友好的用户接口呈现的。本书将会教你一些乐于学习的事物，并且在你的工具箱中加入了一些独立的工具，但是可

能更加重要的是，它将会告诉你一个故事并且在途中给你带来快乐。这是一个关于与社交网站相关的数据科学的故事，它向你展示这些网站堆积的数据以及一些你能够使用这些数据做到的诱人潜力。

如果从头到尾阅读本书，你会注意到这个故事是按照章节顺序展开的。尽管每一章会大体遵循一个容易理解的模式来介绍一种社交网站、教你如何使用它的API获取数据，并且介绍一些分析数据的技术，该书讲述的内容会越来越广泛，同时也会越来越复杂。本书的前几章花一些时间介绍基本概念，然而后面的章节将会系统地建立在前几章的基础上并且逐渐介绍一系列挖掘社交网络的工具和技术，你可以将其应用到你生活的其他方面。

一些最流行的社交网站最近几年已经从流行转变到主流再转变到家喻户晓，它们改变着我们线上和线下的生活方式，它能够让技术给我们呈现出最好的（有时是最坏的）一面。综合来说，本书的每一章都将社交网站与数据挖掘、分析和可视化技术的内容组织在一起，以探索数据并且回答以下典型的问题：

·谁与谁相识，哪些人是他们社交网络中共有的？

·某人与其他人交流有多频繁？

·哪一个社交网络关系为特定的领域产生了最大的价值？

·在网络世界里，地理位置是如何影响你的社会关系的？

·谁是某个社交网络里最有影响力的人（最流行的人）？

·人们在谈论些什么（这个有价值吗）？

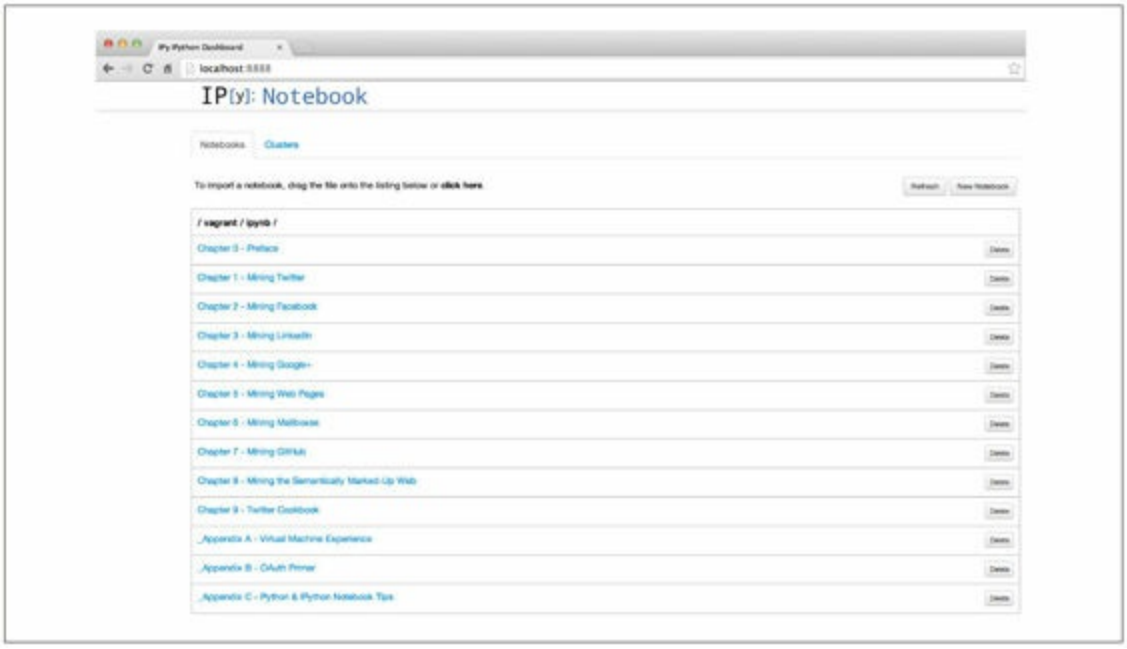
·基于人们在数字世界使用的人类语言，人们感兴趣的是什么？

这些基本问题的答案经常会产生一些有价值的见解，并且为企业家、社会科学家以及其他急于理解一个问题空间并且寻找解决方案的实践者展现盈利的机会。从零开始构建一个一站式的杀手级应用程序（killer app）来回答这些问题，探索远远超出经典可视化库的用法以及构建任何最先进的东西等内容不在本书的讨论范围。如果你购买本书是为了做这些事情，那么你真的会非常失望。然而，本书提供了回答这些问题的基本构造单元，并且为你构建杀手级应用程序或进行学术研究提供助力。自己浏览几章看看，本书涵盖了大量的必备知识。

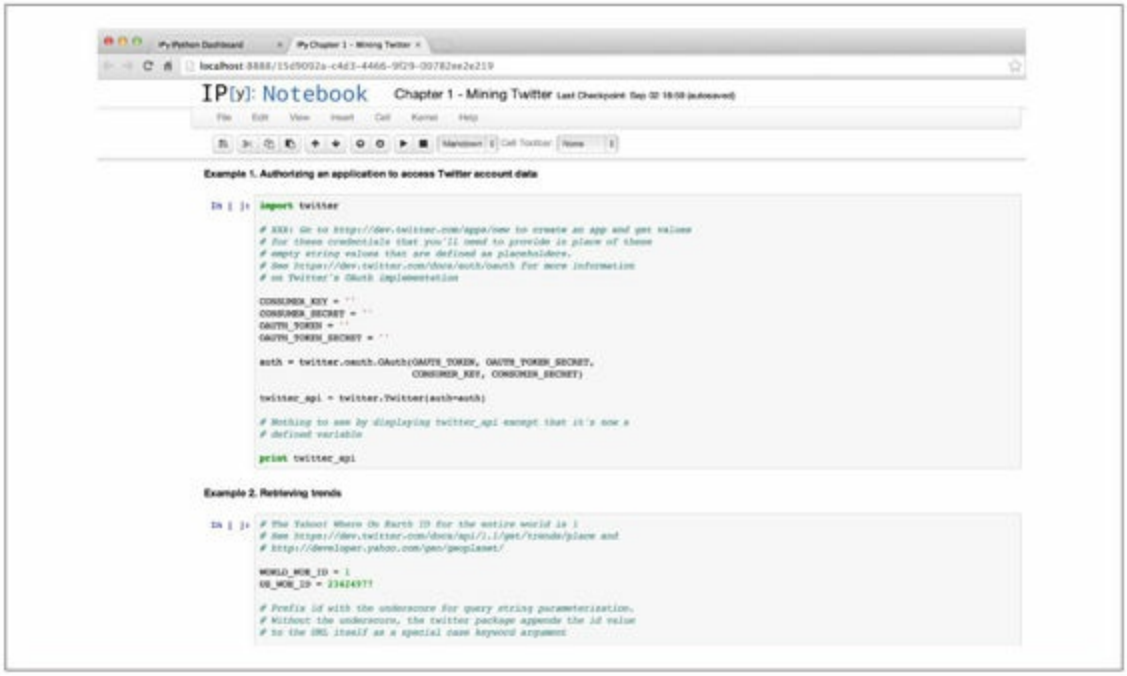
以Python为中心的技术

本书中的所有示例代码特意利用了Python语言的优势。Python直观的语法、迷人的包生态系统，可以最小化API访问和数据操作的复杂性。实际上是JSON（http://bit.ly/1a1kFaF）的核心数据结构使它成为一个出色的教学工具，它不仅强大而且非常容易启动和运行。如果这还不

足以让Python既成为一个伟大的教学选择又成为挖掘社交网络的选择，那么可以借助IPython Notebook（http://bit.ly/1a1kFr4）这个强大的、交互式的Python解释器，它在你的Web浏览器中提供了一个类似笔记（notebook）的用户体验，并且结合了代码执行、代码输出、文本、数学排版、绘图以及更多的功能。我们很难想到更好用户体验的学习环境，因为它将提供样例代码的麻烦最小化了，作为读者你可以跟着它一起执行代码而不会遇到任何麻烦。图P-1提供了一个IPython Notebook体验的图示，它显示了书中每一章Notebook的精简展示（dashboard）。图P-2显示了其中一个Notebook的视图。



图P-1：IPython Notebook概览，其中为Notebook的精简展示



图P-2：IPython Notebook概览，其中为“Chapter 1-Mining Twitter”（本书英文版第1章）的Notebook

书中的每一章都对应一个附带示例代码的IPython Notebook。这使得学习代码、修改bug、按照自己的目的自定义成为一种乐趣。如果你编写过一些程序但是却从来没有看到过Python语法，提前浏览几页一定是你需要的。优秀的文档可以在线获得，如果你正在寻找一个权威的Python编程语言的介绍，那么官方的Python教程（http://bit.ly/1a1kDj8）是很好的一个起点。本书的Python源代码使用Python 2.7编写，它是2.x系列的最新发行版。（尽管可能会碰到点问题，但我们不难想象可以使用一些自动化工具向上转换到Python 3。）

IPython Notebook无疑是很好用的，但是如果你刚接触Python编程，

那么仅仅建议你跟随网上的说明配置你的开发环境可能会有些适得其反（甚至可能是无理的）。为了使你尽可能愉快地体验这本书，一个一站式的虚拟机可能会更合适，它包含IPython Notebook并预装了你重现本书示例所需的其他所有要求条件。你需要做的就是按部就班，大约15分钟就可以运行了。如果你有编程背景，你将能够配置自己的开发环境，但是我希望你会相信：虚拟机体验是更好的出发点。

注意：更多关于本书虚拟机体验的详细信息见附录A。附录C同样也值得你注意：它提供了一些IPython Notebook的提示以及本书源代码中使用的常见Python编程惯用法。

无论你是一位Python新手还是高手，本书最新修复bug的源代码以及附带的用于构建虚拟机的脚本可在GitHub（http://bit.ly/1a1kFHM）上获得。GitHub是一个社交Git仓库（http://bit.ly/16mhOep），它始终反映最新可用的示例代码。我们希望社交编程将会增强那些想要一起工作的志同道合者之间的合作，便于他们扩充示例和不断修改感兴趣的问题。希望你会派生（fork）、扩充（extend）和改进（improve）这些源代码，并且可能会在过程中结识一些新朋友。

注意：官方GitHub库（http://bit.ly/MiningThe SocialWeb2E）包含本书最新、最大程度修复bug的源代码。

第2版的具体改进

当我着手本书第2版的工作时，我完全没有意识到自己将陷入的境地。从一开始就声称的“重大更新”在我认为是对第1版几乎进行了重写。我已经对每一章做了大量更新，高瞻远瞩地增加了新内容，并且我真的相信第2版在各个方面都优于第1版。我真诚希望它比第1版有更广泛的读者。同时希望，通过借助工具、技术以及实战建议并依靠重组和分析社交网站数据来实现我们心中所想，激发广泛的兴趣。如果我成功地达到了这个目标，我们将会看到关于社交网站数据可以做些什么的更广泛的认知以及更多的崭露头角的企业家和热心的爱好者将社交网络数据应用到工作中。

一本书就是一个产品。任何产品的第一个版本都是可以被大大改善的，并不一定是客户想要的。并且如果虚心接受建设性的反馈并用来改进产品，那么会产生巨大的长进。本书也不例外。过去几年中，通过与读者和客户互动获得的针对本书示例代码的反馈和学习体验对于塑造这本书是非常重要的，甚至远远超过了我自己所能创造的。我已经尽可能地纳入这类反馈，它主要可以为读者简化学习体验。

在第2版中简化以不同方式出现。或许，最值得注意的是，本书和上一个版本最大的区别之一在于大大简化了技术工具链，并且我已经通过叫做Vagrant（http://bit.ly/1a1kGeH）的迷人虚拟化技术进行了配置管

理。上一个版本涉及许多存储用的数据库、各种可视化工具，并且假定读者能够通过阅读在线说明弄清楚大部分的安装和配置。

相反，本书不遗余力地介绍了尽可能少的不同的技术依赖，并且将它们全部呈现在虚拟机体验中。它会抽象掉软件安装和配置的复杂性，这有时会比最初看起来更具挑战性。从某个角度看，核心工具箱只包括预装在虚拟机上的IPython Notebook和一些第三方依赖包（这些都是受到版本控制的，这样更新开源软件就不会导致代码受到破坏）。内置的可视化甚至被附加到IPython Notebook上，从IPython Notebook本身呈现并且被整合到一个单独的JavaScript工具D3.js（http://d3js.org），它维持了整章视觉上一致的美感。

继续刚才简化的主题，由于花费更少的时间即可介绍完全不同的技术，这就使得有更多的机会在分析方面进行基本练习。第1版中读者经常诟病的一个内容是应该将更多的时间花在分析和讨论练习的意义上（这确实是一个公正的批评）。我希望通过第2版在现存内容留下空白的地方增加补充说明，从而做到与这些美妙的建议匹配。从某种意义上说，第2版用更精炼的内容做更多的事，因此它也会给读者传递更多的价值。

从结构重组的角度看，你可能已经注意到第2版加入了关于GitHub的一章。GitHub令人感兴趣是有各种各样原因的。你可以从这一章的回顾部分看出它并不都是关于“社交编程”的（尽管这是它的主要部分）。

GitHub是一个非常社交化的网站，它跨越了国界，正在迅速成为一个通用的协作中枢，并且延伸超出了编码范畴，可以被解释为一个兴趣图谱——一幅将人与吸引他们的事物连接的图谱。兴趣图谱（无论是从GitHub得到的还是从其他地方得到的）是这个逐渐发展的Web传奇故事中的重要概念，并且由于有人对社交网络感兴趣，所以你不想忽略它们。

除了关于GitHub的新的一章外，第1版中关于Twitter的“高级的”两章被重构了，扩展成一些更容易接受的Twitter代码配方（recipe），并组织到第9章。尽管本书第1章缓缓地展开并且介绍了社交网络API和数据挖掘的概念作为预热，但是本书最后一章又回到开始的地方并且带来了不同构造单元，你可以基于它们以不同的方式进行组装从而获得无数可能。最后，之前专门介绍微格式的一章已经被并入现在的第8章，它被设计成一种更具前瞻性的关于“带标记的语义网”的精选的某些有趣话题而不是像之前的章节那样的许多编程练习的集合。

注意：建设性的反馈总是受欢迎的，我非常喜欢以书评、发给@SocialWebMining（http://bit.ly/1a1kHzq）的推文或发表在本书Facebook涂鸦墙（http://on.fb.me/1a1kHPQ）的评论方式听到你们的反馈。本书的官方网站和博客http://MiningTheSocialWeb.com允许使用更长形式的内容扩展本书。

本书约定

本书使用了以下排版约定：

斜体（Italic）

用于新术语、URL、电子邮件地址、文件名与文件扩展名。

等宽字体（Constant width）

用于表明程序清单，以及在段落中引用的程序中的元素，如变量、函数名、数据库、数据类型、环境变量、语句、关键字等。

等宽粗体（Constant width bold）

用于表明命令，或者需要读者逐字输入的文本内容。

等宽斜体（Constant width italic）

用于表示需要使用用户提供的值或者由上下文决定的值来替代的文本内容。

注意：表示一个技巧、建议或一般性说明。

警告：表示一个警告或注意事项。

示例代码的使用

本书最新的示例代码在GitHub的http://bit.ly/MiningTheSocialWeb2E维护，这是本书的官方代码库。我们鼓励你关注这个库以便获得最新的修复bug的代码，以及由作者和其他社交编程社区编写的更多示例。如果你阅读的是纸质版，书中的代码示例很可能不是最新的，但是只要你使用本书的GitHub库，你就会获得最新修复bug的示例代码。如果你利用了本书提供的虚拟机，那么你已经获得了最新的源代码，但是如果你选择使用自己的开发环境，请确保直接从GitHub库下载源代码压缩包。

注意：请将关于示例代码的问题记录到GitHub库的问题追踪系统而不是O'Reilly目录的勘误追踪系统。GitHub上源代码的问题被解决时会更新到本书的手稿中，然后定期地作为电子书更新提供给读者。

本书提供代码的目的是帮你快速完成工作。一般情况下，你可以在你的程序或文档中使用本书中的代码，而不必取得我们的许可，除非你想复制书中很大一部分代码。例如，你在编写程序时，用到了本书中的几个代码片段，这不必取得我们的许可。但若将O'Reilly图书中的代码制作成光盘并进行出售或传播，则需获得我们的许可。引用示例代码或书中内容来解答问题无需许可。将书中很大一部分的示例代码用于你个人的产品文档，这需要我们的许可。

如果你引用了本书的内容并标明版权归属声明，我们对此表示感谢，但这不是必需的。版权归属声明通常包括：标题、作者、出版社和ISBN号，例如：“Mining the Social Web，2nd Edition，by Matthew A.Russell.Copyright 2014Matthew A.Russell，978-1-449-36761-9”。

如果你认为你对示例代码的使用已经超出上述范围，或者你对是否需要获得示例代码的授权还不清楚，请随时联系我们：permissions@oreilly.com。

联系我们

有关本书的任何建议和疑问，可以通过下列方式与我们取得联系：

美国：

O’Reilly Media，Inc.

1005Gravenstein Highway North

Sebastopol，CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们会在本书的网页中列出勘误表、示例和其他信息。可以通过访问如下网址获得：

http://bit.ly/mining_social_web_2e

任何关于示例代码的勘误可以作为工单（ticket）通过GitHub的问题追踪系统在如下网址提交：

http://github.com/ptwobrussell/Mining-the-Social-Web/issues

读者可以通过GetSatisfaction从作者或出版商处寻求常见帮助，网址如下：

http://getsatisfaction.com/oreilly

要评论或询问本书的技术问题，请发送电子邮件到：

bookquestions@oreilly.com

想了解关于O’Reilly图书、课程、会议和新闻的更多信息，请访问以下网站：

http://www.oreilly.com.cn

http://www.oreilly.com

第2版致谢

我要重申本书第1版说过的话，编写一本书意味着做出很多牺牲。你远离家人和朋友的时间（多数发生在持续很长时间的晚上和周末）是相当宝贵的而且无法倒流的。你真的需要一定的精神支持才能在保持好关系的同时渡过难关。再次感谢对我非常有耐心的朋友和家人，他们真的可以不再容忍我写另一本书，并且可能认为我有某种痴迷于熬夜工作和周末加班的慢性疾病。如果你能找到治疗痴迷写书这一病症的康复诊所，我保证会去给自己做个检查。

每个项目都需要一个伟大的项目经理。我的编辑Mary Treseler让我很佩服，能与她以及她优秀的印刷团队合作出版本书，我很高兴。编写一本技术书籍是一个漫长而充满压力的事业，至少可以说，能与这么多专业人士一起工作是一个了不起的经验。正是在他们的帮助下，你才能顺利通过这个令人筋疲力尽的旅程并出版一部精致打磨的、让你乐于分享于世的作品。Kristen Brown、Rachel Monaghan和Rachel Head真真切切地把我的付出提升到一个全新的专业水准。

从才华出众的编辑人员和技术专家那里收到的详细反馈令我惊叹不已。这些反馈涵盖从非常技术化的建议到软件工程方面的Python最佳实践，再到转换身份从读者角度如何最大程度地满足目标受众。它们远远超出了我曾经的预期。如果不是这些同行给出的宝贵评议意见，你将要

阅读的这本书很难达到目前的质量。特别向Abe Music、Nicholas Mayne、Robert P.J.Day、Ram Narasimhan、Jason Yee和Kevin Makice致谢，他们针对草稿给出了非常详细的评议意见。他们的意见大大提升了这本书的质量，而我唯一的遗憾是我们没有机会在这个过程中更密切的合作。还要感谢Tate Eskew向我介绍Vagrant（http://bit.ly/1a1kGeH），这一工具为本书建立了一个易于使用和易于维护的虚拟机体验环境。

我还要感谢许多值得称道的Digital Reasoning同事，我们多年来畅谈有关数据挖掘和计算机科学的主题。与他们的谈话帮助我形成专业思维。我很荣幸成为这样有天分与能力的团队的一员。特别感谢Tim Estes和Rob Metcalf，他们一直支持我从事类似写书这样耗时的项目（在Digital Reasoning公司职责之外）。

最后，感谢每一位读者和本书代码的使用者，他们在本书第1版的整个生命周期里提供了建设性的反馈意见。虽然你们的名字多到无法在此列出，但你们的反馈意见已经在塑造第2版的过程中发挥了不可估量的作用。我希望第2版能符合你们的期望，并可以位列你愿意推荐给朋友或同事的有用书单之中。

第1版致谢

毫不夸张地说，编写一本技术书籍需要做出很多牺牲。在家里，我

放弃了与妻子Baseeret和女儿Lindsay Belle相处的很多时间，这比我敢于承认的时间还要多。虽然我的抱负是有朝一日能在一定程度上征服世界（这只是暂时的，坦白地说，我正在尽力摆脱这种状态），但我还是要对她们的爱表示感谢。

我深信你所做的一切决定最终都会影响到你的一生（尤其是你的职业生涯），但是谁也不可能孤独前行，我们要懂得感恩。撰写本书时，我真的很庆幸能与世界上最聪明的一帮人合作，其中包括像Mike Loukides这样聪明的技术编辑，以及O'Reilly这样极富天赋的制作团队，还有帮助我完成本书的很多热心的评审人。我要特别感谢AbeMusic、Pete Warden、Tantek Celik、J.Chris Anderson、Salvatore Sanfilippo、Robert Newson、DJ Patil、Chimezie Ogbuji、Tim Golden、Brian Curtin、Raffi Krikorian、Jeff Hammerbacher、Nick Ducoff和Cameron Marlowe对本书所用材料的评审或者对本书提出的有见地建议，所有这些都帮助提升本书质量。在此我也要感谢Tim O'Reilly的慷慨帮助，他允许我研究他Twitter和Google+上的数据；这些内容必定会为某些章节增趣不少。我不可能一一介绍曾经直接或间接地帮助过我或者帮助过本书出版的人，在此一并表示感谢。

最后，要感谢你考虑阅读本书。如果你在阅读本书，至少你有可能会愿意购买一本。如果你真的购买了本书，虽然我尽了最大努力，但你还是会发现本书存在的一些疏漏之处。然而，我坚信，虽然疏漏在所难

免，但本书定会让你觉得值得你花上几个晚上或几周的时间来细细研读，而且最终你也的确会有所收获。

第一部分　社交网络导引

本书的第一部分命名为“社交网络导引”，因为它提出了若干从一些最流行社交网站获得直接价值的实用技巧。你将学习如何访问API并分析来自Twitter、Facebook、LinkedIn、Google+、网页、博客和订阅、电子邮件以及GitHub账户的社交数据。一般情况下，每个章节都是相对独立的并讲述一个自成体系的故事，但第一部分所有章节串联起来也讲述一个更完整的故事。在进入轻松地讨论与当前社交网络生活有关的语义网知识之前，主题的复杂性渐次增强。

因为复杂性是逐步增加的，我们鼓励你依次阅读每一章，但你也应该精选某些章节并跟随其中的示例。每章的示例代码合并到一个单独的IPython Notebook，每个Notebook都是根据本书中的章节编号命名。

注意：本书的源代码可以在GitHub（http://bit.ly/1a1kNqy）上找到。我们强烈鼓励你借助虚拟机体验，这样你可以在一个预先配置的“恰好工作”（just work）的开发环境中顺利运行示例代码。

序幕

虽然已在前言中提及而且将继续不经意地在后续各章提起，这不是随正文内容配有示例代码库的传统科技书籍。这本书一反常态并为技术书籍定义一个新标准，其中的代码是作为一流的、开源软件项目来管理的，而本书则是对该代码库的“高级”支持。

为了达到这一目标，经过精心考虑，把本书的讨论与代码示例尽可能完美地集成为一个自然的学习体验。经过与第1版读者的多次讨论并针对教训予以反思，很明显，通过运行有虚拟机的服务器的支持并植根于坚实的配置管理的交互式用户界面是本书最可取的方式。没有更简更好的方式能够赋予你对代码的完全控制权，同时确保代码将“恰好工作”，而不必担心你是使用Mac OS、Windows或Linux，你是运行在32位或64位机，是否和第三方软件的依赖关系改变了或是破坏了API。

好好利用这个强大的互动学习环境。

注意：关于建立一个本书第2版中虚拟机过程的更多思考，请进一步阅读“创作最小可行著作的思考”一文（http://bit.ly/1a1kPyJ）。

虽然第1章是开始阅读最合理的地方，但当准备开始运行代码示例时，你应该花点时间来熟悉附录A和附录C。附录A提供了一个在线文档并带有截屏，引导你通过一个快速简便的安装过程建立虚拟机。附录C

给出了在线文档，提供一些背景资料，有助于你从交互式虚拟机体验中获得最大价值。

即使你是一位经验丰富的开发者，能够凭一己之力做好这一切工作，在首次尝试本书时试用虚拟机能让你免受软件安装过程中必然遭遇的小挫折。

第1章　挖掘Twitter：探索热门话题、发现人们的谈论内容等

本章，我们将从Twitter开始揭开挖掘社交网络的序幕。Twitter是社交数据的丰富来源，它对公共消费固有的开放性、简洁且文档完备的API、丰富的开发者工具以及对各行各业用户广泛的吸引力，使它成为进行社交网络挖掘的理想起跑线。Twitter的数据是非常有趣的，因为推文（tweet）可以以最快的速度把想法发布出去，它们几乎是实时的，并且它的多样性代表了国际范围内最广泛的社会阶层。通过简短的对话（通常是有意义的）、将人与其关注事物相连的兴趣图谱（interest graph）以及推文和Twitter的“关注”（following）机制将人们以不同的方式联系到一起。

由于这是本书的第1章，我们将会花些时间让你逐步适应社交网络挖掘的旅程。不过，考虑到Twitter数据很容易得到并且接受公众的审查，第9章以问答的形式提供了一系列既短小精悍又具有普遍性的代码配方，从而进一步阐述了大量数据挖掘的可能性。你也可以将随后章节中的概念应用到Twitter数据中。

注意：一定要从http://bit.ly/MiningTheSocialWeb2E在线获得本章（和其他每一章）最新的错误修正版本的源代码。此外，请务必利用好

本书附录A中描述的虚拟机体验，从而最大化地享用示例代码。

1.1 概述

在这一章，我们将会开始配置最基本的（但是却非常高效的）Python开发环境、研究Twitter API，并使用频率分析从推文中提取一些分析性结论。这一章里你将了解的主题包括：

·Twitter开发者平台以及如何发起API请求。

·推文元数据以及如何使用它。

·从推文中提取实体，例如：提及的用户、主题标签和URL。

·使用Python进行频率分析的相关技术。

·使用IPython Notebook绘制Twitter数据的直方图。

1.2 Twitter风靡一时的原因

大多数的章节并不会从发人深省的讨论开始，但是由于这是全书的第1章并要引入一个往往会被人误解的社交网站，因此从根本的层面审视Twitter似乎更为合适。

你会如何定义Twitter呢？

有很多种方式可以回答这个问题，但我们从宏观的角度来考虑，任何技术只有对我们共有的人性的某些根本方面负责，才能是真正有用并成功的。毕竟，技术是为了提升我们人类的体验。

作为人类，我们希望技术能够帮助我们获得的有哪些呢？

·我们希望被听到。

·我们希望满足自己的好奇心。

·我们希望事情变得容易。

·我们现在就想得到。

在当前讨论的语境下只有部分观察现象属于通常意义的人性。我们对分享自己的想法和体验有一种根深蒂固的需求，这使我们能够与别人

交流、能够被倾听、能够感受到自己的价值和重要性。我们对周围的世界以及如何组织、操纵它充满了好奇，并且通过交流分享自己的看法、提出问题、在困惑中与别人进行有意义的交谈。

最后两点凸显了我们固有的对摩擦的无法容忍。理想情况下，为了满足我们的好奇心或完成某项特定的工作，除了必要的工作外，我们不希望更加辛苦。我们更想做其他事情或者转移到下一件事情，因为我们的时间如此珍贵却又如此短暂。与此类似，我们或许现在就想做某些事情，并且往往会对事情没有按照我们计划的速度执行而感到不耐烦。

一种定义方式是把Twitter描述成微博服务，该服务允许人们使用简短的、符合人们思维的140个字符的消息进行交流。在这方面，你可以把Twitter想象成类似于一个免费、高速、全球性的文本消息服务。换句话说，它是允许快速、简易交流的基础设施的重要部分。然而，这并不是全部。当有超过5亿好奇的用户，而其中超过1亿人每个月定期活跃时（http://bit.ly/1a1kNXR），显现出来的并不能充分满足我们固有的好奇心和价值取向。

除了宏观层面上营销和广告的可能性（面对这么大的用户基数，商机是无限的），潜在的网络动态性产生了对如此多用户的引力，这才是真正有趣的，也正是Twitter的魅力所在。尽管使用户以极快的速度共享简短消息的交流渠道是Twitter平台用户增长和持续参与的必要条件，但它并不是充分条件。使其成为充分条件的其他因素是：Twitter非对称的

关注模型满足了我们的好奇心。正是这种非对称的关注模型将Twitter描述成一种兴趣图谱而不是社交网络，并且API提供了足够的框架使结构或自组织行为从混乱中显现出来。

换句话说，尽管一些社交网站，比如Facebook、LinkedIn，需要关系双方的互相接受（通常表明现实世界的某种关系），然而Twitter的关系模型可以让你关注其他任何用户的最新进展，即使这些用户可能没有选择关注你甚至不知道你的存在。Twitter的关注模型非常简单但是却利用了我们之所以成为人类的基本特性：我们的好奇心。这可能是对名人绯闻的痴迷、对获取自己最喜欢体育团体最新信息的急切、对某个政治话题的浓厚兴趣或者是对与其他人接触的渴望。Twitter提供了无限的机会以满足你的好奇心。

警告：尽管在前一段我已经从“关注”关系的角度介绍了Twitter，关注某个人的行为有时被描述成“加好友”（friending）（尽管这是一种奇怪的单向好友关系）。虽然你将会在官方的Twitter API文档的命名系统中看到“好友”（http://bit.ly/1a1kOuI），然而我们最好认为Twitter是之前描述的关注关系。

我们可以将兴趣图谱想象成对人们与他们的兴趣之间的关系进行建模的一种方式。兴趣图谱在数据挖掘领域提供了大量可能性，这些主要涉及测量事物之间的相关性从而进行智能推荐，或涉及机器学习中的其他应用。例如，你可以使用兴趣图谱测量相关性，从向你推荐Twitter上

关注对象到在线推荐购买什么商品再到推荐应该与谁约会。为了更好地理解Twitter的兴趣图谱这个概念，假设Twitter用户不一定是一个真实的人。它可以是一个人，也可以是一个没有生命的物体、一个公司、一个音乐团队、一个虚构的角色、虚拟的某个人（现存的或死去的）或其他任何事物。

例如，@HomerJSimpson（http://bit.ly/1a1kQD1）这个账号是Homer Simpson的官方账号，他是电视节目The Simpsons中的流行角色。尽管Homer Simpson不是一个真实的人，但他却是一个世界知名的人物，@HomerJSimpson的Twitter用户充当者帮助他（实际上是其创作者）吸引粉丝。类似的，尽管这本书可能无法达到Homer Simpson的知名度，但是@SocialWebMining（http://bit.ly/1a1kHzq）是本书的官方Twitter账号并且为对该书内容感兴趣的团体提供了以多种方式进行联系和参与的手段。当你认识到Twitter允许你创建、联系和探索任何一个感兴趣话题的兴趣团体时，Twitter的力量以及你从挖掘Twitter数据中获得的见解也就变得更加明显。

除了Twitter账号中的一些可以将名人和公众人物识别成“认证账号”的徽章以及Twitter的服务条款协议（http://bit.ly/1a1kRXl）中的基本约束之外（这是使用服务必需的），Twitter账号的内容很少受到控制。这看起来微不足道，但是这是和一些社交网站的重要的不同，在那些社交网站中，账号要么必须对应真实的、活生生的人，要么对应商业公司

或分类系统中类似性质的实体。Twitter对账号的人物并没有特定的约束，并且依赖自组织的行为，例如关注关系和使用由主题标签构成的大众分类法在系统中创建一个特定形式的秩序。

分类法和大众分类法

人类智能的体现之一是渴望对事物进行分类并且获得一个层次结构，其中每个元素“属于”或者是层次结构中上一级父元素的“孩子”。忽略分类学和本体论的一些细微区别（http://bit.ly/1a1kRXy），将分类学看成一个类似树的层次结构并将元素分类成特定的父、子关系，而大众分类法（folksonomy）（http://bit.ly/1a1kU5C）（该术语大约是2004年创建的）描述了出现在Web不同生态系统中的协同标签和社交索引尝试的领域。它将大众（folk）和分类法（taxonomy）这两个词结合了起来。因此从本质上讲，大众分类法只是一种描述去中心化标签领域的绝好方式，当你允许人们使用标签分类内容时，它可以涌现为集体智慧。Twitter中的主题标签的使用如此引人关注是因为大众分类法是有机地聚集共同兴趣的方式，既为探索提供了专业手段，也为发现无数奇妙事物留下无限可能性。

1.3 探索Twitter API

既然我们已经对Twitter进行了适当的框架性介绍，现在我们将注意力转移到获取和分析Twitter数据的问题上。

1.3.1 基本的Twitter术语

Twitter可以被描述成一个实时的、高度社交化的微博服务。它允许用户在时间轴上发布简短的状态更新，这些状态被称为推文（tweet）。在推文140个字符的内容中可能包含一个或多个实体，并且引用一个或多个可以映射到现实世界中某个位置的地点。理解用户、推文和时间轴对高效地使用Twitter API（http://bit.ly/1a1kSKQ）是尤为重要的，因此在使用API获取数据之前，我们有必要对这些基本的Twitter平台对象（http://bit.ly/1a1kSL8）进行简要地介绍。到目前为止，我们已经大量讨论了Twitter用户及其非对称的关注模型，因此这一节我们会简要地介绍推文和时间轴，从而让读者对Twitter有大致的了解。

推文是Twitter的核心。尽管从概念上它被认为是与用户状态更新相关的140字符的文本内容，然而实际上还有其他一些我们并没有看到的元数据。除了推文本身的文本内容外，推文还捆绑了两种需要我们特别注意的元数据：实体和地点。实体大体上是用户引用、主题标签、URL

以及和推文相关的媒体文件，而地点是可能被附加到推文中的现实世界的位置。请注意，地点可能是推文发送时所在的实际位置，但也可能是对推文中描述的地点的指代。

为了使描述具体一些，我们来看一条以下内容的推文：

@ptwobrussell is writing@SocialWebMining，2nd Ed.from his home office in Franklin，TN.Be#social：http://on.fb.me/16WJAf9

这条推文的长度为124个字符并且包含4个推文实体：用户引用@ptwobrussell和@SocialWebMining、主题标签#social以及URL http://on.fb.me/16WJAf9。尽管这条推文中有一个明确提到的地点“Franklin，Tennessee”，然而和推文关联的位置元数据可能包含推文创建时所在的位置，该位置可能并不是“Franklin，Tennessee”。这些大量的元数据被封装到不到140个字符的内容中，从而表明推文语言的强大：它可以明确地引用多个其他的Twitter用户、到网页的链接、使用主题标签相互引用的主题，而这些话题作为聚集点对整个Twitter虚拟空间（Twitterverse）以一种便于搜索的方式进行垂直切分。

最后，时间轴是按时间排序的推文集合。抽象地说，你可以认为时间轴是按时间顺序呈现的特定推文集合。然而，你通常会发现有一些时间轴是非常值得注意的。从任意一个Twitter用户的角度来看，主页时间轴是你登录到账户时看到的视图，并且还能看到所有你关注的用户发送

的推文。而特定的用户时间轴仅仅是某个用户发送的推文集合。

例如，当你登录到Twitter账户时，你的主页时间轴位于https://twitter.com。而任何特定的用户时间轴必须添加一个指定用户的后缀，比如https://twitter.com/SocialWebMining。如果你想从特定用户的角度浏览他的主页时间轴，你可以在URL结尾添加“following”后缀，然后进行访问。例如，Tim O’Reilly登录到他的Twitter之后看到的主页时间轴可以通过https://twitter.com/timoreilly/following访问。

TweetDeck这类应用程序为混乱的推文场景提供了一些可自定义的视图，如图1-1所示。如果你还没有深入了解Twitter.com的用户接口，这个值得你进行尝试。

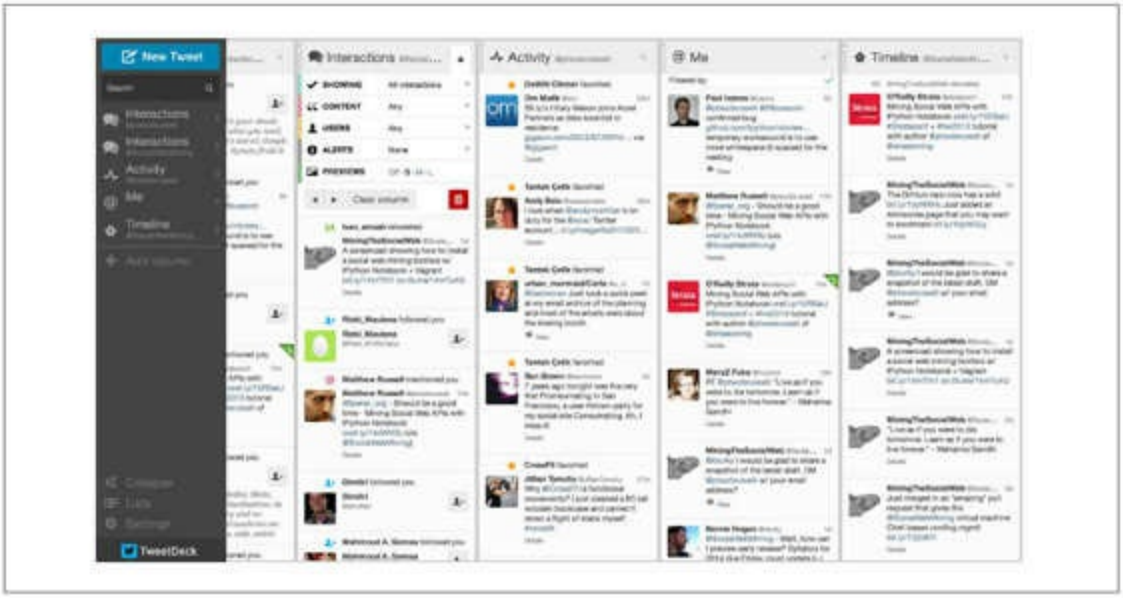


图1-1：TweetDeck提供了一个可高度自定义的用户接口，它对分析Twitter上正在发生的事情以及展现那些可以通过Twitter API访问的数据

是很有帮助的

时间轴是更新速度较慢的推文集合，而流（stream）是一些在Twitter上实时流动的公共推文。在一些被广泛关注的事件期间，例如总统辩论，所有推文的公共信息（public firehose）的峰值为每分钟几十万条（http://bit.ly/1a1kV9N）。从这本书的角度来说，Twitter的公共信息提供的数据超出所考虑的范围，并且提出了有趣的工程性挑战，这至少是许多第三方提供商与Twitter合作将这些数据以可消费的方式带给用户的原因之一。即便如此，对公共时间轴的较小随机采样（http://bit.ly/1a1kVq7）是可用的，它对API开发者提供了足够多的公共数据的过滤访问以便让他们开发出功能强大的应用程序。

本章的余下内容以及本书的第二部分假定你已经拥有了一个Twitter账号，这是API访问所必需的。如果你还没有Twitter账号，可以花些时间新建一个，然后看一看Twitter宽松的服务条款（http://bit.ly/1a1kWKB）、API文档（http://bit.ly/1a1kSKQ）以及开发者守则（http://bit.ly/1a1kX1a）。本章和第二部分的实例代码基本不需要你自己有好友和关注者。但是如果你的账户十分活跃，有很多好友和关注者，那么你会发现第二部分的一些例子会更加有趣，你可以将它作为社交网络数据挖掘的基础。如果你的账户并不活跃，那么现在就是开始添加、填充数据的好时机，从而让你获得数据挖掘的乐趣。

1.3.2 创建一个Twitter API连接

Twitter非常精心地创建了一个直观、易用的API，同时这些API符合优雅而简单的REST（RESTful）（http://bit.ly/1a1kVX5）。即使如此，仍然有很多可用的库可以让我们进一步减少发起API请求时所做的工作。twitter包就是很好的例子，它很好地封装了Twitter API，并完全模拟了公共API的语义。和大多数其他的Python包一样，你可以在终端上通过输入pip install twitter命令，来用pip安装它。

注意：如何安装pip请参考附录C。

Python提示：开发过程中利用pydoc获得高效帮助

我们将通过几个例子说明如何使用twitter包，如果需要帮助，你可以通过几种不同的方式浏览Python包的文档

pydoc（http://bit.ly/1a1kVXg）。如果不使用Python shell，那么使用终端在PYTHONPATH下运行某个包的pydoc是一个很好的选择。例如，在Linux或Mac系统下，可以仅仅通过在终端输入pydoc twitter获得包的文档，而pydoc twitter.Twitter提供了这个包包含的Twitter类的文档。在Windows系统下，尽管方法稍微不同，你仍然可以通过将pydoc作为包来执行，然后获得同样的信息。例如，输入python-mpydoc twitter.Twitter，就会提供twitter.Twitter类的信息。如果你经常查阅某些特定模块的文档，可以选择在pydoc中加上-w选项，然后文档就会被写

成可以在浏览器中保存、收藏的HTML页面。

然而，更可能的是，当你需要帮助时，你正处于工作会话中。内置的help函数接收一个包或类名并且对于普通的Python shell很有用，而IPython（http://bit.ly/1a1kXyf）用户可以在包名或类名后面加上一个问号从而查看内置的帮助。比如，你可以在常规的Python解释器里输入help（twitter）或者help（tiwtter.Twitter），你也可以在IPython或IPython Notebook中输入更加简洁的twitter？或twitter.Twitter？

我们强烈建议你在没有使用IPython Notebook的时候把IPython作为标准的Python shell，因为它提供了很多更加方便的函数，例如制表符补齐、会话历史记录、“魔法函数”（magic function）（http://bit.ly/1a1kXyf）。附录A提供了最简化的细节学习如何使用推荐的开发者工具，如IPython。

注意：我们选择使用Python创建实用的API请求，因为twitter包十分优雅地模拟了满足REST的API。如果你对了解使用HTTP能够创建的原始请求很感兴趣或者想以一种更加交互的方式研究API，那么去看看开发者控制台（http://bit.ly/1a1kWui）或者命令行工具Twurl（http://bit.ly/1a1kZq1）。

在向Twitter创建API请求之前，你需要在https://dev.twitter.com/apps创建一个应用程序。创建应用程序是开发者获取API权限，并且让

Twitter按照需要对第三方平台的开发者进行监控和交互的标准方式。创建一个应用程序的过程是很标准的，你所需要的只是对API的只读访问。

在这个情况下，为了访问你的账号数据，你要创建一个应用程序并对其进行授权，这似乎有点多余：为什么不直接加上账号名和密码对API进行访问呢？尽管这种方法对你来说是很有效的，然而第三方（例如朋友或者同事）可能会对仅仅为了从你的应用程序获取数据而交出自己的账号和密码感到不舒服。放弃凭证永远不会是正确的做法。幸运的是，一些聪明的人很多年前就认识到了这个问题，现在对于更广阔的社交网络有一个标准的协议，称为OAuth（开放授权的缩写）（http://bit.ly/1a1kZWN），其以一种通用的方式服务于这种情况。目前，该协议是社交网络的标准。

如果你没有记住以上任何信息，那么你只要记住OAuth是允许用户对第三方应用程序进行授权，从而可以在不需要共享一些密码之类敏感信息的情况下访问他们账号数据的一种方式。如果你对此感兴趣，附录B提供了一个关于OAuth如何工作的宽泛概述，同时Twitter的OAuth文档（http://bit.ly/1a1kZWW）提供了关于其实现的具体细节[1]。

为了便于开发，你需要从新建的应用程序配置中获取关键信息，包括用户账号、用户密码、访问令牌以及访问令牌密钥。这4组凭证组合起来，提供了保证应用程序最终获得授权（授权过程中涉及请求用户批

准授权的一系列重定向操作）所需的一切，所以它们是与密码一样敏感的信息，要好好保存。

注意：生成一个支持任意用户授权使用他们账号数据的应用程序时，可能需要OAuth 2.0流程支持，相关细节详见附录B。

图1-2显示了获取这些凭证的页面。

事不宜迟，让我们创建一个已认证的Twitter API连接并且通过查阅从GET trends/place资源（http://bit.ly/1a1kYSQ）获取的数据找出人们正在谈论哪些话题。然而，在编写之前你需要收藏官方API文档（http://bit.ly/1a1kSKQ）以及REST API 1.1版资源（http://bit.ly/1a1kZ9i），因为在你学习Twitter虚拟空间里开发者技巧的过程中，你将会经常参考这些文档。

注意：截止到2013年3月，Twitter API运行在1.1版本并且与以前的版本1的API在某些方面有很大的不同。版本1的API已经被弃用半年了并且以后不再运作。本书中的所有示例代码假定API的版本为1.1。

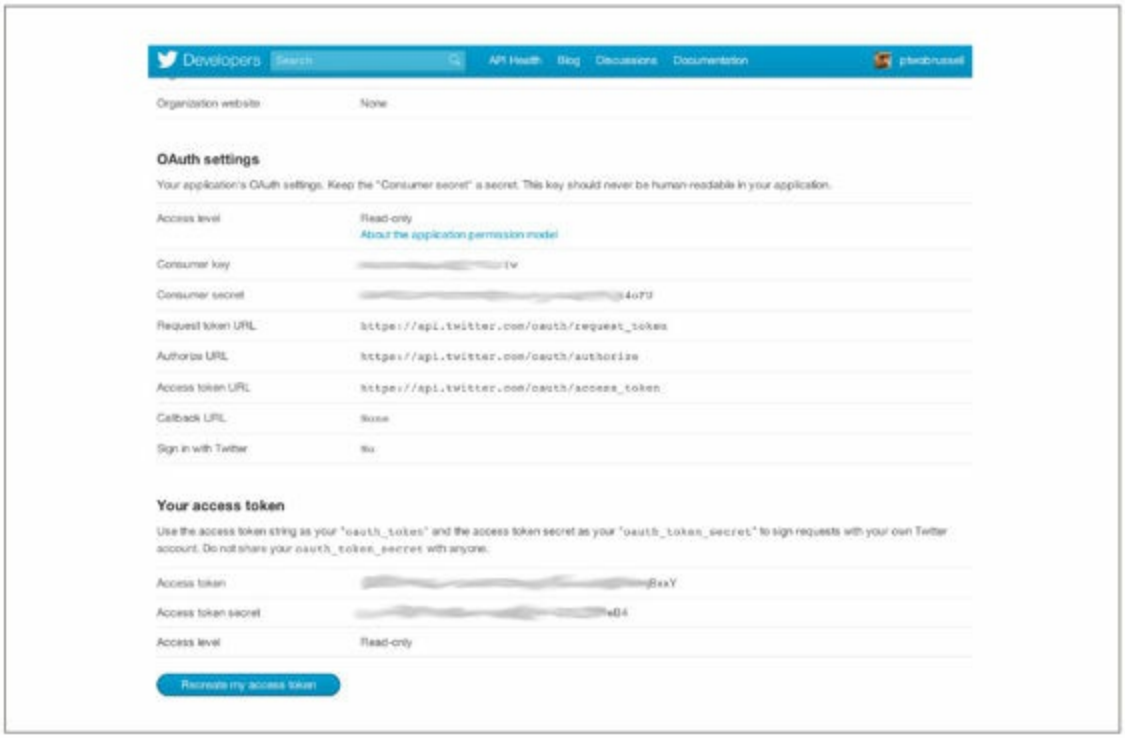


图1-2：在https://dev.twitter.com/apps创建了一个新的Twitter应用程序以获得OAuth凭证以及API访问权限；四个（模糊的）OAuth域是你向Twitter API创建API调用时需要用到的

让我们打开IPython Notebook并且初始化一个搜索。依照示例1-1将代码开头的变量替换成你自己的凭证信息，然后执行创建Twitter API实例的调用。使用你的OAuth凭证创建一个代表你的OAuth授权的名为auth的对象，它可以被传递给一个能够向Twitter API提交查询的名为Twitter的类，随后代码就可以正常工作了。

示例1-1：对一个应用程序授权使其能访问Twitter账号数据

```
    import twitter
# XXX: Go to http://dev.twitter.com/apps/new to create an app and get values
# for these credentials, which you'll need to provide in place of these
```

```
# empty string values that are defined as placeholders.
# See https://dev.twitter.com/docs/auth/oauth for more information
# on Twitter's OAuth implementation.
CONSUMER_KEY = ''
CONSUMER_SECRET = ''
OAUTH_TOKEN = ''
OAUTH_TOKEN_SECRET = ''
auth = twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                           CONSUMER_KEY, CONSUMER_SECRET)
twitter_api = twitter.Twitter(auth=auth)
# Nothing to see by displaying twitter_api except that it's now a
# defined variable
print twitter_api
```

该例子的结果应该只显示我们所创建的twitter_api对象的一个明确表示，例如：

```
<twitter.api.Twitter object at 0x39d9b50>
```

这表明我们已经成功使用OAuth凭证获得了查询Twitter API的授权。

1.3.3 探索热门话题

得到API连接的授权后，你现在可以发起一个请求了。示例1-2显示了如何向Twitter查询目前世界范围内热门的话题，如果你想尝试自己想要的内容，仅仅通过传递API参数就能将话题限定在一些更加具体的领域。限制查询的设备是通过Yahoo！GeoPlanet（http://yhoo.it/1a1kZ9u）的Where On Earth（WOE）ID系统，该系统本身就是一个API，它的目的是提供一种将唯一的标识符映射到地球上（或者是理论上，甚至是虚

拟世界的）特定地点的方法。尝试运行以下的示例代码，这个例子同时收集了全世界和仅仅美国的热门话题集合。

示例1-2：获得热门话题

```
   # The Yahoo! Where On Earth ID for the entire world is 1.
# See https://dev.twitter.com/docs/api/1.1/get/trends/place and
# http://developer.yahoo.com/geo/geoplanet/
WORLD_WOE_ID = 1
US_WOE_ID = 23424977
# Prefix ID with the underscore for query string parameterization.
# Without the underscore, the twitter package appends the ID value
# to the URL itself as a special case keyword argument.
world_trends= twitter_api.trends.place(_id=WORLD_WOE_ID)
us_trends= twitter_api.trends.place(_id=US_WOE_ID)
printworld_trends
print
printus_trends
```

在进一步处理之前，你应该会看到一个无法完全理解的来自API的响应信息，该响应信息是Python词典的列表（跟任何错误消息不同），例如下面截取的结果（随后，我们将会对响应信息进行格式化处理，使其更加易读）。

```
   [{u'created_at': u'2013-03-27T11:50:40Z', u'trends': [{u'url':
  u'http://twitter.com/search?q=%23MentionSomeoneImportantForYou'…
```

请注意，样例结果内容包含了一个热门话题的URL，该话题被表示成与主题标签#MentionSomeoneImportantForYou对应的搜索查询，其中23%是对主题标签符号的编码。在本章剩余的部分中，我们将这个相对正面的主题标签作为随后例子中的统一主题。尽管本书的源代码中有一个包含该主题推文的样例数据文件，然而探索当前正在流行的话题要比

跟随不再流行的固定话题有趣得多。

使用twitter模块的模式十分简单而明确：使用与基本URL对应的对象链来初始化Twitter类，然后调用该对象中与URL上下文对应的方法。例如，twitter_api._trends.place（WORLD_WOE_ID）初始化一个GET https://api.twitter.com/1.1/trends/place.json?id=1的HTTP调用。请注意这个映射到由twitter包创建以发起请求的对象链上的URL，同时注意查询字符串参数是如何作为关键字参数进行传递的。为了将twitter包用在任意的API请求上，你一般要用这种直接的方式构建请求，我们很快就会遇到一些需要注意的地方。

Twitter规定了给定时间间隔内一个应用程序向任何API资源发出请求的速率限制（rate limit）（http://bit.ly/1a1l257）。Twitter的速率限制是明确地记录在文档中的，并且为了方便起见，每一个单独的API资源也给出了其特定的限制。例如，我们刚才为获得热门话题提交的API请求将程序限制在每15分钟的间隔内只能发送最多15次请求（见图1-3）。如果你想获得关于Twitter速率限制更详细的信息，请参考REST API访问速率限制1.1版（http://bit.ly/1a1l2ly）。如果是仅仅运行本章的内容，那么你基本不会受到速率的限制。9.16节（示例9-16）将会介绍应对速率限制的一些实用技巧。

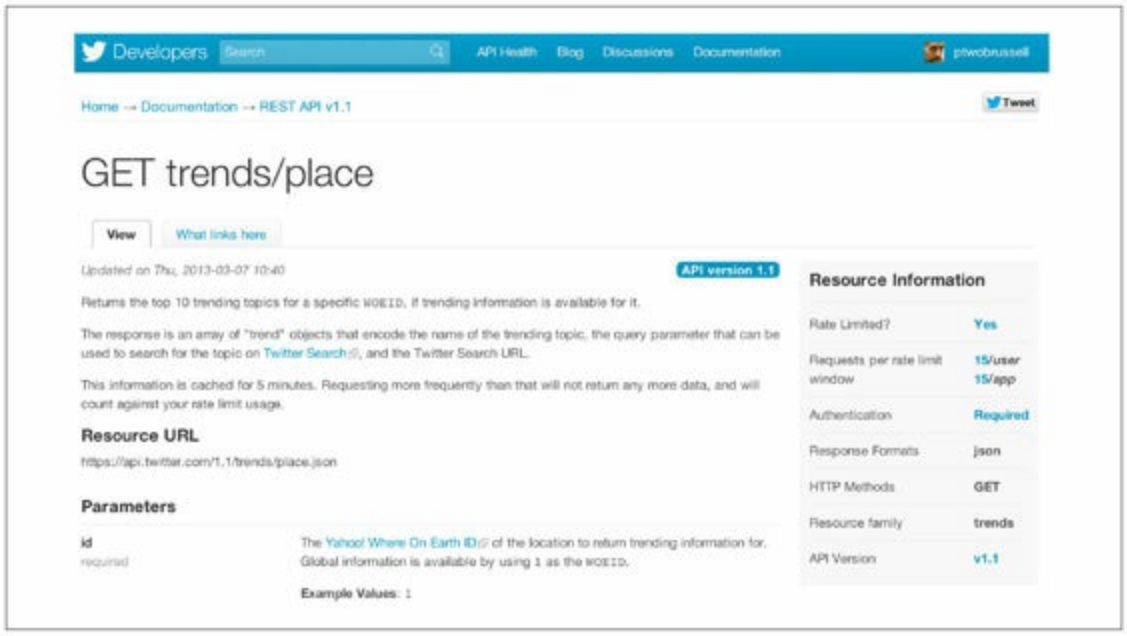


图1-3：在在线文档中，Twitter API资源中每个API调用的速率限制都明确的记录在案，这里显示的特定API资源每个速率限制时间窗内（当前定义为15分钟）允许15次请求

注意：开发者文档指出，热门话题API查询的结果每5分钟才更新一次，因此以更快的频率发起API请求结果是不明智的。

尽管没有明确指出，示例1-2中无法完全可读的输出是打印的原生Python数据结构。尽管IPython解释器会自动为你打印出漂亮的结果，然而IPython Notebook和标准的Python解释器却不会这样做。如果你自己处于这种状况，那么你会发现使用内置的json包进行强制的格式化显示是很方便的，如示例1-3所示。

注意：JSON（http://bit.ly/1a1l2lJ）是你会经常遇到的一种数据交换

格式。简而言之，JSON提供了一种随意存储映射（map）、列表（list）、原始类型（例如数字和字符串）以及它们的组合的方法。换句话说，如果你愿意，你可以使用JSON对任何事物进行建模。

示例1-3：显示API响应信息为优美打印的JSON格式

```
   import json
print json.dumps(world_trends, indent=1)
print
print json.dumps(us_trends, indent=1)
```

使用json.dumps从热门话题API得到的一个简短的响应示例可能是这样的：

```
   [
 {
   "created_at": "2013-03-27T11:50:40Z",
   "trends": [
    {
      "url": "http://twitter.com/search?q=%23MentionSomeoneImportantForYou",
      "query": "%23MentionSomeoneImportantForYou",
      "name": "#MentionSomeoneImportantForYou",
      "promoted_content": null,
      "events": null
     },
     ...
    ]
  }
]
```

尽管浏览这两个热门话题集合并且寻找其中的共性是很简单的，但是现在我们使用Python中的集合set（http://bit.ly/1a1l2Sw），这个数据结构会自动帮我们计算，因为这正是集合适用的场合。在这个例子中，集合指的是存储着一组无序、互异元素的数据结构的一种数学概念，并且可以和其他元素集合进行计算或是执行集合操作。例如，集合的求交操

作计算多个集合中共有的元素，集合的求并操作则将多个集合中的元素合并。集合的求差操作类似一种减法操作，其中凡是出现在一个集合中的元素都会从另外一个集合中删除。

示例1-4显示了如何使用Python的列表解析（list comprehension）（http://bit.ly/1a11hy）从之前查询的结果中解析出热门话题的名字、将这些列表转换成集合，然后计算交集从而找出两者共有的元素。请注意，任意给定的热门话题集合之间可能有很多重叠的内容，也有可能没有，这完全取决于你查询热门话题时实际发生的事情。换句话说，你分析的结果完全取决于你的查询以及其返回的数据。

注意：附录C提供了一些Python常见用法的参考资料，例如列表解析等，这可能会对你有所帮助。

示例1-4：计算两个热门话题集合的交集

```
   world_trends_set= set([trend['name']
                      for trend in world_trends[0]['trends']])
us_trends_set= set([trend['name']
                    for trend in us_trends[0]['trends']])
common_trends= world_trends_set.intersection(us_trends_set)
print common_trends
```

注意：在继续学习本章内容之前，你应该完成示例1-4从而确保你能够访问和分析Twitter数据。你能解释一下你所在国家和世界其他地区的热门话题之间存在的关联吗？

集合论、直觉和可数无穷大

集合操作看上去可能是一种相对原始的分析方式，但是集合论在数学中的衍生物是很深远的，因为它为许多数学原理提供了基础。

一般认为，Georg Cantor正式确立了集合论背后的数学理论，他的论文“On a Characteristic Property of All Real Algebraic Numbers”（1874）确立了集合论并将其作为回答无穷大相关问题工作的一部分。为了理解其中的原理，考虑以下问题：正整数集合的基数是否比同时包含正整数和负整数的集合的基数大？

尽管通常的直觉可能会觉得整数集合的基数是正整数的两倍，然而Cantor的工作表明这两个集合的基数实际上是相等的！从数学的角度，他显示你可以将这两组数字的集合进行映射，从而形成有一个确定的起点并且从一个方向上无限扩展的序列，如：{1，－1，2，－2，3，－3，…}。

由于数字可以被很明确地列举但是却永远不会有终点，因此我们说集合的基数为可数无穷大。换句话说，如果你有足够的时间去数，那么该序列的数字都是可以明确列出的。

1.3.4 搜索推文

我们发现热门话题集合中一个共有的元素是主题标签

#MentionSomeoneImportant-ForYou，因此我们将其作为搜索查询的基础，来获取一些推文从而进行深入的分析。示例1-5显示了如何对某个感兴趣的查询使用GET search/tweets资源（http://bit.ly/1a1l398），包括对于搜索结果使用元数据中的特殊字段的能力，从而对于更多的搜索结果创建额外的请求。Twitter的信息流API（http://bit.ly/1a1l1ya）资源超出了本章的范围，但是我们会在9.8节（示例9-8）介绍它，因为在你想维护一个持续更新推文视图的情况下会更加合适。

注意：示例1-5中使用的函数参数*args和**kwargs分别是Python中表示可变参数和关键字参数的常用语法。这种语法的简要概述见附录C。

示例1-5：收集搜索结果

```
   # XXX: Set this variable to a trending topic,
# or anything else for that matter. The example query below
# was a trending topic when this content was being developed
# and is used throughout the remainder of this chapter.
q= '#MentionSomeoneImportantForYou'
count = 100
# See https://dev.twitter.com/docs/api/1.1/get/search/tweets
search_results= twitter_api.search.tweets(q=q, count=count)
statuses = search_results['statuses']
# Iterate through 5 more batches of results by following the cursor
for _ in range(5):
    print "Length of statuses", len(statuses)
    try:
        next_results= search_results['search_metadata']['next_results']
    except KeyError, e: # No more results when next_results doesn't exist
        break
# Create a dictionary from next_results, which has the following form:
# ?max_id=313519052523986943&q=NCAA&include_entities=1
kwargs= dict([ kv.split('=') for kv in next_results[1:].split("&") ])
search_results= twitter_api.search.tweets(**kwargs)
statuses+= search_results['statuses']
# Show one sample search result by slicing the list...
print json.dumps(statuses[0], indent=1)
```

注意：尽管这里我们只将一个主题标签传递给搜索API，然而值得注意的是它包含了许多强大的操作符（http://bit.ly/1a1l3pN），这些操作符允许你根据各种关键字、推文发送者、推文相关地点等是否存在对查询进行过滤。

实际上，所有代码所做的是重复地向搜索API发送请求。如果你使用过其他的网络API（包括Twitter API版本1），那么最初可能会让你措手不及的是搜索API本身没有明确的分页概念。回顾一下API文档你会发现这样做是有意为之的，并且考虑到Twitter资源的高度动态性，使用游标的方法也有一些好的理由（http://bit.ly/1a1l4K6）。游标在整个Twitter开发者平台上的最佳实践略有不同，其中搜索API对搜索结果提供了比其他资源（例如时间轴）更简单的导航方式。

搜索结果包括一个特殊的search_metadata节点，它嵌入了一个next_results字段，该查询字符串为随后的查询提供了基础。如果我们没有使用twitter这样的库帮我们创建HTTP请求，这种预先构造的查询字符串可能仅仅会被添加到搜索API的URL中。然而，由于我们没有直接地创建HTTP请求，我们必须将查询字符串分解成一组键/值对并且将其作为关键词参数提供。

按照Python的语法，我们正在将字典中的值拆包成函数接收的关键词参数。换句话说，示例1-5中for循环内的函数最终调用了如twitter_api.search.tweets（q='%23MentionSomeoneImportantForYou'，

include_entities=1，max_id=313519052523986943）这样的函数，虽然从源代码上看却是twitter_api.search.tweets（**kwargs），这里kwargs是键/值对组成的字典。

注意：search_metadata字段也包含一个叫做refresh_url的值，如果你想使用经过前一个查询之后才变得可用的新信息对结果集合进行维护和定期更新，那么你可以使用这个值。

下面的推文示例显示了对#MentionSomeoneImportantForYou查询的搜索结果。花些时间仔细阅读一下。像我之前提到的一样，推文的实际内容比我们看到的要多很多。下面的这个推文相当具有代表性，当使用未压缩的JSON格式显示的时候，它包含了超过5KB的内容。这大约是我们通常认为的一个推文140个字符数据量的40多倍！

```
  [
{
 "contributors": null,
 "truncated": false,
 "text": "RT @hassanmusician: #MentionSomeoneImportantForYou God.",
 "in_reply_to_status_id": null,
 "id": 316948241264549888,
 "favorite_count": 0,
 "source": "<a href=\"http://twitter.com/download/android\"...",
 "retweeted": false,
 "coordinates": null,
 "entities": {
  "user_mentions": [
   {
    "id": 56259379,
    "indices": [
     3,
     18
    ],
   "id_str": "56259379",
   "screen_name": "hassanmusician",
   "name": "Download the NEW LP!"
   }
  ],
  "hashtags": [
```

```
  {
   "indices": [
    20,
    50
   ],
   "text": "MentionSomeoneImportantForYou"
  }
 ],
 "urls": []
},
"in_reply_to_screen_name": null,
"in_reply_to_user_id": null,
"retweet_count": 23,
"id_str": "316948241264549888",
"favorited": false,
"retweeted_status": {
 "contributors": null,
 "truncated": false,
 "text": "#MentionSomeoneImportantForYou God.",
 "in_reply_to_status_id": null,
 "id": 316944833233186816,
 "favorite_count": 0,
 "source": "web",
 "retweeted": false,
 "coordinates": null,
 "entities": {
  "user_mentions": [],
  "hashtags": [
   {
    "indices": [
     0,
     30
    ],
    "text": "MentionSomeoneImportantForYou"
   }
  ],
  "urls": []
},
"in_reply_to_screen_name": null,
"in_reply_to_user_id": null,
"retweet_count": 23,
"id_str": "316944833233186816",
"favorited": false,
"user": {
 "follow_request_sent": null,
 "profile_use_background_image": true,
 "default_profile_image": false,
 "id": 56259379,
 "verified": false,
 "profile_text_color": "3C3940",
 "profile_image_url_https": "https://si0.t...",
 "profile_sidebar_fill_color": "95E8EC",
 "entities": {
  "url": {
   "urls": [
    {
     "url": "http://t.co/yRX89YM4J0",
     "indices": [
      0,
      22
     ],
```

```
      "expanded_url": "http://www.datpiff...",
      "display_url": "datpiff.com/mixtapes-detai\u2026"
     }
    ]
   },
   "description": {
    "urls": []
   }
  },
  "followers_count": 105041,
  "profile_sidebar_border_color": "000000",
  "id_str": "56259379",
  "profile_background_color": "000000",
  "listed_count": 64,
  "profile_background_image_url_https": "https://si0.t...",
  "utc_offset": -18000,
  "statuses_count": 16691,
  "description": "#TheseAreTheWordsISaid LP",
  "friends_count": 59615,
  "location": "",
  "profile_link_color": "91785A",
  "profile_image_url": "http://a0.twimg.com/...",
  "following": null,
  "geo_enabled": true,
  "profile_banner_url": "https://si0.twimg.com/pr...",
  "profile_background_image_url": "http://a0.twi...",
  "screen_name": "hassanmusician",
  "lang": "en",
  "profile_background_tile": false,
  "favourites_count": 6142,
  "name": "Download the NEW LP!",
  "notifications": null,
  "url": "http://t.co/yRX89YM4J0",
  "created_at": "Mon Jul 13 02:18:25 +0000 2009",
  "contributors_enabled": false,
  "time_zone": "Eastern Time (US & Canada)",
  "protected": false,
  "default_profile": false,
  "is_translator": false
 },
 "geo": null,
 "in_reply_to_user_id_str": null,
 "lang": "en",
 "created_at": "Wed Mar 27 16:08:31 +0000 2013",
 "in_reply_to_status_id_str": null,
 "place": null,
 "metadata": {
  "iso_language_code": "en",
  "result_type": "recent"
 }
},
"user": {
 "follow_request_sent": null,
 "profile_use_background_image": true,
 "default_profile_image": false,
 "id": 549413966,
 "verified": false,
 "profile_text_color": "3D1957",
 "profile_image_url_https": "https://si0.twimg...",
 "profile_sidebar_fill_color": "7AC3EE",
 "entities": {
```

```
   "description": {
    "urls": []
   }
  },
  "followers_count": 110,
  "profile_sidebar_border_color": "FFFFFF",
  "id_str": "549413966",
  "profile_background_color": "642D8B",
  "listed_count": 1,
  "profile_background_image_url_https": "https:...",
  "utc_offset": 0,
  "statuses_count": 1294,
  "description": "i BELIEVE do you? I admire n adore @justinbieber ",
  "friends_count": 346,
  "location": "All Around The World ",
  "profile_link_color": "FF0000",
  "profile_image_url": "http://a0.twimg.com/pr...",
  "following": null,
  "geo_enabled": true,
  "profile_banner_url": "https://si0.twimg.com/...",
  "profile_background_image_url": "http://a0.tw...",
  "screen_name": "LilSalima",
  "lang": "en",
  "profile_background_tile": true,
  "favourites_count": 229,
  "name": "KoKo :D",
  "notifications": null,
  "url": null,
  "created_at": "Mon Apr 09 17:51:36 +0000 2012",
  "contributors_enabled": false,
  "time_zone": "London",
  "protected": false,
  "default_profile": false,
  "is_translator": false
 },
 "geo": null,
 "in_reply_to_user_id_str": null,
 "lang": "en",
 "created_at": "Wed Mar 27 16:22:03 +0000 2013",
 "in_reply_to_status_id_str": null,
 "place": null,
 "metadata": {
  "iso_language_code": "en",
  "result_type": "recent"
 }
},
 ...
]
```

推文包含一些社交网络中存在的最丰富的元数据，在第9章我们将对其中的许多可能性进行阐述。

[1] 虽然Twitter的OAuth文档是一种实现的细节，但Twitter的1.1版本API

一直使用OAuth 1.0a，这使得它可能没有多大价值，而许多其他的社交网络已经更新到OAuth 2.0。

1.4 分析140字的推文

在线文档总是Twitter平台对象的权威来源，并且其中的推文（http://bit.ly/1a1l3WL）页面是值得收藏的，因为在你熟悉推文基本构造的过程中你会经常参考它。本书并没有直接引用在线文档，但是考虑到你可能会被推文包含的5KB的信息吓到，有一些注释是很有必要的。为了命名上的简化，我们假设已经从搜索结果提取出了一个单独推文，并且将其存储在名为t的变量中。例如，t.keys（）返回这条推文最外层的字段，t['id']得到推文的标识符。

注意：如果这一章你一直在使用IPython Notebook，那么我们用到的推文是被存储在名为t的变量中，这样你就可以交互式地访问它的字段并且更加容易地进行探索。我们当前的讨论也是假定在相同命名环境中的，所以值应该是一一对应的。

·通过t['text']我们可以得到推文可读的文本内容：

```
RT @hassanmusician: # MentionSomeoneImportantForYou God.
```

·推文文本中的实体已经很方便地为你处理好了，你可以通过t['entities']获得：

```
  {
"user_mentions": [
```

```
  {
   "indices": [
    3,
    18
   ],
   "screen_name": "hassanmusician",
   "id": 56259379,
   "name": "Download the NEW LP!",
   "id_str": "56259379"
  }
 ],
 "hashtags": [
  {
   "indices": [
    20,
    50
   ],
   "text": "MentionSomeoneImportantForYou"
  }
 ],
 "urls": []
}
```

·关于推文兴趣度的相关线索可以通过t['favorite_count']和t['retweet_count']获得，它们分别返回该推文被收藏和转推的次数。

·如果一条推文被转推了，那么t['retweet_status']字段提供了关于原始推文本身以及其作者非常详细的信息。请注意，有时候一条推文被转推后其文字内容会发生变化，因为用户添加了回复或对内容进行了其他操作。

·t['retweeted']字段表明认证用户（通过授权的应用程序）是否对该推文进行转推。随特定用户所持观点而变化的字段在Twitter开发者文档表示为perspectival，这意味着它们的值随着用户角度的不同而发生变化。

·此外，注意从API和信息管理的角度看，只有原始的推文被转推。

因此，retweet_count反映了原始推文被转推的次数，并且原始推文和之后的转发中的这个值应该是相同的。换句话说，转推的推文并没有被转推。乍一看这可能有悖常理，但是如果你现在正在转推一条已被转推的推文，实际上你只是在通过代理转推看到的原始推文。1.4.4节对有关于转推和引用推文之间的不同进行更细致的讨论。

警告：通过检查retweet字段的值决定一条推文是否被转推过是我们很容易犯的错误。为了检查一条推文是否被转推，应该看看推文中是否存在一个retweeted_status节点包装器。

在继续之前，你应该再研究一下样例推文并且查阅文档，从而理清遗留的问题。掌握良好的推文构造的实用知识对有效地挖掘Twitter数据是至关重要的。

1.4.1 提取推文实体

接下来，我们将推文实体和文本内容提炼到更加方便的数据结构以供进一步研究。示例1-6从收集的推文中提取了文本、昵称和主题标签，并且引入了一个叫做双重（嵌套）列表解析的Python语法。如果你理解（单重）列表解析，那么下面的代码说明双重列表解析仅仅是从嵌套循环中获得的值的集合而不是单一循环的结果。列表解析是格外强大的，因为它们通常比嵌套列表获得更可观的性能收益并且提供了直观

（一旦你熟悉了）却又简洁的语法。

注意：在整本书中，列表解析经常被使用，如果你想知道更多的背景，可以参考附录C或官方的Python教程（http://bit.ly/1a1l1hy）从而获得更详细的信息。

示例1-6：从推文中提取文本、昵称和主题标签

```
   status_texts = [ status['text']
                    for status in statuses ]
screen_names = [ user_mention['screen_name']
                    for status in statuses
                        for user_mention in status['entities']['user_mentions'] ]
hashtags = [ hashtag['text']
               for status in statuses
                    for hashtag in status['entities']['hashtags'] ]
# Compute a collection of all words from all tweets
words = [ w
          for t in status_texts
              for w in t.split() ]
# Explore the first 5 items for each...
print json.dumps(status_texts[0:5], indent=1)
print json.dumps(screen_names[0:5], indent=1)
print json.dumps(hashtags[0:5], indent=1)
print json.dumps(words[0:5], indent=1)
```

示例的输出如下，它显示了5个状态文本、昵称以及主题标签，从而让我们对数据的内容有大致的认识。

注意：在Python语法中，列表或字符串值后面加上方括号表示切片，例如status_texts[0：5]，你可以很方便地从列表中提取元素或是从字符串中提取子字符串。在这个情况下，[0：5]表明你想要得到列表status_texts中的前5项（对应索引为0到4的元素）。附录C有对Python切片更加详细的描述。

```
   [
 "\u201c@KathleenMariee_: #MentionSomeOneImportantForYou @AhhlicksCruise...,
 "#MentionSomeoneImportantForYou My bf @Linkin_Sunrise.",
 "RT @hassanmusician: #MentionSomeoneImportantForYou God.",
 "#MentionSomeoneImportantForYou @Louis_Tomlinson",
 "#MentionSomeoneImportantForYou @Delta_Universe"
]
[
 "KathleenMariee_",
 "AhhlicksCruise",
 "itsravennn_cx",
 "kandykisses_13",
 "BMOLOGY"
]
[
 "MentionSomeOneImportantForYou",
 "MentionSomeoneImportantForYou",
 "MentionSomeoneImportantForYou",
 "MentionSomeoneImportantForYou",
 "MentionSomeoneImportantForYou"
]
[
 "\u201c@KathleenMariee_:",
 "#MentionSomeOneImportantForYou",
 "@AhhlicksCruise",
 ",",
 "@itsravennn_cx"
]
```

和预期的一样，主题标签的输出结果都是#MentionSomeoneImportantForYou。同时输出也提供了一些经常出现且值得研究的用户昵称。

1.4.2 使用频率分析技术分析推文和推文实体

实际上，几乎所有的分析都可以归结为某种程度的计数行为。本书中我们要做的很多事情是对数据进行处理，达到可以对其计数并且可以更有意义地被进一步处理的目的。

从经验来看，对观察到的事物计数是一切的出发点，因此它也是任何试图从噪声数据中寻找微弱信号的统计滤波或处理的出发点。前面我们仅仅从每个未排序的列表中提取前5项以了解数据的大致内容，现在我们通过计算频率分布并且查找出每个列表中的前10项从而对数据有更深的认识。

截至Python 2.7版本，collections（http://bit.ly/1a1l4tC）模块提供了一个计数器，它使得计算频率分布变得异常容易。示例1-7演示了如何使用Counter计算频率分布得到排过序的元素列表。挖掘Twitter数据的原因很多，其中一个令人信服的原因是试图回答现在人们正在交流些什么。为了回答这个问题，频率分析是你可以使用的最简单的一个技术，正如我们现在进行的。

示例1-7：从推文单词中创建基本的频率分布

```
   from collections import Counter
for item in [words, screen_names, hashtags]:
    c = Counter(item)
    print c.most_common()[:10] # top 10
    print
```

下面是从推文频率分析中得到的一些示例结果：

```
   [(u'#MentionSomeoneImportantForYou', 92), (u'RT', 34), (u'my', 10),
 (u',', 6), (u'@justinbieber', 6), (u'<3', 6), (u'My', 5), (u'and', 4),
 (u'I', 4), (u'te', 3)]
[(u'justinbieber', 6), (u'Kid_Charliej', 2), (u'Cavillafuerte', 2),
 (u'touchmestyles_', 1), (u'aliceorr96', 1), (u'gymleeam', 1), (u'fienas', 1),
 (u'nayely_1D', 1), (u'angelchute', 1)]
[(u'MentionSomeoneImportantForYou', 94), (u'mentionsomeoneimportantforyou', 3),
 (u'NoHomo', 1), (u'Love', 1), (u'MentionSomeOneImportantForYou', 1),
 (u'MyHeart', 1),  (u'bebesito', 1)]
```

频率分析的结果是对应元素以其频率的键/值对映射，因此让我们用表格的形式使结果更加直观、易读。你可以通过在终端输入pip install prettytable安装一个叫做prettytable的包，这个包提供了便于将结果格式化成固定宽度的表格格式的方法，这样我们可以很容易地进行复制和粘贴。

示例1-8展示了使用prettytable显示相同结果的方法。

示例1-8：用prettytable以更漂亮的表格形式显示元组

```
   from prettytable import PrettyTable
for label, data in (('Word', words),
                    ('Screen Name', screen_names),
                    ('Hashtag', hashtags)):
    pt = PrettyTable(field_names=[label, 'Count'])
    c = Counter(data)
    [ pt.add_row(kv) for kv in c.most_common()[:10] ]
    pt.align[label], pt.align['Count'] = 'l', 'r' # Set column alignment
    print pt
```

示例1-8的结果用一系列文本格式的表格显示，便于浏览，如下列输出所示。

```
   +---------------------------------+-------+
| Word                            | Count |
+---------------------------------+-------+
| #MentionSomeoneImportantForYou |    92 |
| RT                              |    34 |
| my                              |    10 |
| ,                               |     6 |
| @justinbieber                   |     6 |
| &lt;3                           |     6 |
| My                              |     5 |
| and                             |     4 |
| I                               |     4 |
| te                              |     3 |
+---------------------------------+-------+
+----------------+-------+
```

```
| Screen Name    | Count |
+----------------+-------+
| justinbieber   |     6 |
| Kid_Charliej   |     2 |
| Cavillafuerte  |     2 |
| touchmestyles_ |     1 |
| aliceorr96     |     1 |
| gymleeam       |     1 |
| fienas         |     1 |
| nayely_1D      |     1 |
| angelchute     |     1 |
+----------------+-------+
+-------------------------------+-------+
| Hashtag                       | Count |
+-------------------------------+-------+
| MentionSomeoneImportantForYou |    94 |
| mentionsomeoneimportantforyou |     3 |
| NoHomo                        |     1 |
| Love                          |     1 |
| MentionSomeOneImportantForYou |     1 |
| MyHeart                       |     1 |
| bebesito                      |     1 |
+-------------------------------+-------+
```

通过快速浏览以上内容，我们至少可以发现一个略微让人吃惊的事：Justin Bieber在这个小型样本数据的实体列表中排名很高。尽管我们不能从这些结果中得出定论，但是考虑到他在Twitter上的人气，他很有可能是这个热门话题中的“最重要的人物”。上面出现的&lt；3也是非常有趣的，因为它是<3的转义形式，代表着心形（和其他表情符号一样旋转了90度），是“love”的常见缩写。尽管最初它可能看起来像垃圾或噪声数据，考虑到查询的本质，我们并会不对&lt；3这样的值感到奇怪。

尽管频率大于2的实体是很有趣的，然而更广泛的结果却揭露其他的方面。例如，“RT”是一个常见的标记，它意味着大量的推文转发（我们将在1.4.4节深入研究这个现象）。最后，正如我们预料的，#MentionSomeoneImportantForYou这个主题标签以及其大小写上的一些

不同形式占据了几乎全部的主题标签。我们在数据处理上应当在绘制频率图表时将每个单词、昵称以及主题标签归一化到小写形式，因为推文中一定存在大小写上的差异。

1.4.3 计算推文的词汇丰富性

一种略加先进的度量方式涉及计算简单的频率，并且能够应用到非结构化文本中，叫做词汇丰富性（lexical diversity）。从数学上看，它是文本中不重复的单词的个数除以文本中所有单词个数得到的表达式，这是非常基本但是却很重要的度量方式。在人际交流中，词汇丰富性是一个非常有趣的概念，因为它为某个人或团体的词汇丰富性提供了定量的度量。例如，设想一下，你正在听某个人重复地说“and stuff”概括的描述信息而不是提供具体的例子的方法去通过细节强调观点。现在，我们将这个说话者和其他很少说“stuff”一词，而是使用具体例子强调观点的人进行对比。重复说“and stuff”的说话者的词汇丰富性可能比使用更多词汇的说话者要低，并且很有可能你会从对话中离开并且认为好像具有更高词汇丰富性的人更加理解主题。

正如被应用在推文或者其他类似的在线交流中一样，词汇丰富性可以作为一种基本统计量回答很多问题，例如某个人或团体讨论的话题是宽泛的还是狭小的。尽管整体的评估可能是很有趣的，然而将分析分解

成具体的时间段可能会得到额外的见解，因为可以对不同的个人或团体进行对比。例如，如果你想研究对比两个饮料公司，比如可口可乐（http://bit.ly/1a1l5xR）和百事可乐（http://bit.ly/1a1l7pt），在Twitter上进行社交化媒体营销活动的有效性，那么以衡量它们之间的词汇丰富性是否存在显著的差异作为研究的切入点将会是非常有趣的。

在对如何使用类似词汇丰富性统计量分析文本内容（比如推文）有了基本的理解之后，我们可以计算当前使用数据集关于状态、昵称和主题标签的词汇丰富性，如示例1-9所示。

示例1-9：计算推文的词汇丰富性

```
   # A function for computing lexical diversity
def lexical_diversity(tokens):
    return 1.0*len(set(tokens))/len(tokens)
# A function for computing the average number of words per tweet
def average_words(statuses):
    total_words = sum([ len(s.split()) for s in statuses ])
    return 1.0*total_words/len(statuses)
print lexical_diversity(words)
print lexical_diversity(screen_names)
print lexical_diversity(hashtags)
print average_words(status_texts)
```

警告：在Python 3.0之前，除法操作符（/）会使用floor舍入函数并且返回一个整型值（除非有一个操作数是浮点值）。可以将分子或分母乘以1.0从而避免这种舍入错误。

示例1-9的结果如下：

```
   0.67610619469
0.955414012739
```

```
0.0686274509804
5.76530612245
```

在这个结果中有一些现象值得我们考虑：

·推文文本中单词的词汇丰富性大约为0.67。解读这个数字的一种方式是：每三个单词中有两个单词是互异的，你也可以说每条状态更新携带了67%的互异的信息。考虑到每条推文平均单词个数为6，这就相当于每条推文有4个互异的单词。这个数据是与我们的直觉相符的，因为#MentionSomeoneImportantForYou这个热门主题标签的特性就是为了查询包含很少单词的响应信息。在任何情况下，0.67的词汇丰富性对于日常人际交往来说已经算很高了，但是考虑到数据的特性，这个值也似乎是很合理的。

·然而，网络昵称的词汇丰富性更高，它的值为0.95，这意味着提及的20个昵称中有19个是互异的。这个现象也是很合理的，因为很多问题的回答就是一个昵称，并且大多数人并不会为某个受关注的主题标签提供相同的回复。

·主题标签的词汇丰富性是非常低的，它的值大约为0.068，这表明除了#MentionSomeoneImportantForYou只有很少的值在结果中出现多次。同样，这也是很合理的，因为大多数的响应是很短的，而且引入主题标签作为谁是对你重要的人这一问题的回答确实不怎么有效。

·推文中单词的平均长度很小，略小于6个。由于主题标签是用来查

询包含很少单词的响应的，因此这个值也是合理的。

现在，更进一步检视这些数据，看看能否从更加定性的分析中得到公共的响应或者其他的见解。考虑到每条推文中单词的平均个数少至6个，用户在140个字符的内容中使用缩写的可能性不大，因此数据中噪声的数量也非常少，额外的频率分析可能会揭露一些意想不到的事情。

1.4.4 检视转推模式

尽管用户界面和很多Twitter客户端已经采用原生的用于填写retweet_count和retweeted_status等状态值的转推API很长时间了，然而一些Twitter用户更喜欢引用推文（http://bit.ly/1a1l7FZ），这就需要通过包括复制和粘贴文本、在前面添加“RT@username”或是加上后缀“/via@username”等流程提供相关属性。

注意：在挖掘Twitter数据时，为了最大化你的分析效率，你可能既想解释推文元数据，又想利用启发规则分析出140字符转推中的特殊记号，例如“RT@username”或者“/via@username”。9.14节是对使用Twitter原生的转推API进行转推和使用传统的方法添加属性引用推文方式之间比较的更详细的讨论。

此时，更好的做法是深入地分析数据从而判断是否存在某条推文被

大量转推了或者仅仅是有很多“一次性的”转推。寻找最流行的转推的方法是简单地对每一条状态更新进行迭代，如果该状态更新是转推，就存储转推数量、转推发起者和转推内容。示例1-10演示了如何使用列表解析获取这些值并且对转推数量进行排序从而显示排名最高的几条结果。

示例1-10：寻找最流行的转推

```
   retweets = [
            # Store out a tuple of these three values ...
            (status['retweet_count'],
             status['retweeted_status']['user']['screen_name'],
             status['text'])
            # ... for each status ...
            for status in statuses
            # ... so long as the status meets this condition.
                if status.has_key('retweeted_status')
           ]
# Slice off the first 5 from the sorted results and display each item in the tuple
pt = PrettyTable(field_names=['Count', 'Screen Name', 'Text'])
[ pt.add_row(row) for row in sorted(retweets, reverse=True)[:5] ]
pt.max_width['Text'] = 50
pt.align= 'l'
print pt
```

示例1-10得到的结果非常有意思：

```
+-------+----------------+----------------------------------------------------+
| Count | Screen Name    | Text                                               |
+-------+----------------+----------------------------------------------------+
| 23    | hassanmusician | RT @hassanmusician: #MentionSomeoneImportantForYou |
|       |                | God.                                               |
| 21    | HSweethearts   | RT @HSweethearts: #MentionSomeoneImportantForYou   |
|       |                | my high school sweetheart ❤                        |
| 15    | LosAlejandro_  | RT @LosAlejandro_: ¿Nadie te menciono en           |
|       |                | "#MentionSomeoneImportantForYou"? JAJAJAJAJAJAJAJA |
|       |                | JAJAJAJAJAJAJAJAJAJAJAJAJAJAJAJAJAJAJA Ven, ...   |
| 9     | SCOTTSUMME     | RT @SCOTTSUMME: #MentionSomeoneImportantForYou My  |
|       |                | Mum. Shes loving, caring, strong, all in one. I    |
|       |                | love her so much ❤❤❤❤                              |
| 7     | degrassihaha   | RT @degrassihaha: #MentionSomeoneImportantForYou I |
|       |                | can't put every Degrassi cast member, crew member, |
|       |                | and writer in just one tweet....                   |
+-------+----------------+----------------------------------------------------+
```

“God”是列表中的第一个，紧随其后的是“my high school sweetheart”，排名第四的是“My Mum”。前五个结果中没有一个可以对应到Twitter用户账号上，尽管我们可能从之前的分析中对此（除@justinbieber之外）有所猜测。继续观察列表中后面的结果会出现一些用户账号，但是我们从查询中获得的样本很小不足以发现任何趋势。在一个较大样本的结果上进行搜索，我们可能会发现有些用户提及的频率大于1，对此进行进一步分析是很有趣的。进一步分析的可能性是无限的，到目前为止，希望你跃跃欲试，多尝试一些自定义的查询。

注意：推荐的练习在这一章的最后。此外，请务必参考作为灵感源泉的第9章：它包括二十多种代码配方并且以实用指南的方式呈现。

在我们继续之前，一个值得注意的是我们转推的原始推文极可能不在我们的示例搜索结果集里（因为这一节中观察的转推的频率相当低）。例如，示例结果中最流行的转推来自于昵称为@hassanmusician的用户并且被转推了23次。然而，对数据进行深入的检视会发现，在我们的搜索结果中我们只收集了23个转推中的一个。原始的推文和其他22个转推都没有出现在数据集中。尽管我们可能会提问，哪22个用户转推了这条状态，但是这却不会导致任何特别的问题。

这种问题的答案是非常有价值的，因为它允许我们选取能代表一个概念的内容，比如这里的“God”，并且能够发现分享相同心情和共同兴趣的用户团体。像之前提到的一样，兴趣图谱是一种非常便于对包含人

物以及他们感兴趣的事物的数据进行建模的方法，它是支持第7章分析的主要数据结构。这些用户可能是宗教人士。对他们的特定推文进行深入分析可能会支持这一推论。示例1-11演示了如何使用GET statuses/retweets/：id API（http://bit.ly/1a1l64H）寻找这些人。

示例1-11：寻找转推过状态的用户

```
   # Get the original tweet id for a tweet from its retweeted_status node
# and insert it here in place of the sample value that is provided
# from the text of the book
_retweets = twitter_api.statuses.retweets(id=317127304981667841)
print [r['user']['screen_name'] for r in _retweets]
```

这里把对转推这条状态给任何宗教或信仰组织的用户进行深入分析的需求留作一个独立的练习。

1.4.5 使用直方图将频率数据可视化

IPython Notebook的一个非常好的特性是它能够生成和插入高质量的可定制数据图，将其作为交互式工作流的一部分。特别是，matplotlib（http://bit.ly/1a1l7Wv）包和其他科学计算工具对于IPython Notebook是可用的，它们是非常强大的，并且一旦理解了基本的流程你就能很轻松地生成复杂的图表。

为了说明matplotlib绘图功能，我们绘制一些数据的图表并将其显示

出来。作为热身，我们考虑将示例1-9定义的words变量中的结果绘制成图表。在Counter的帮助下，我们能够很容易地生成一个排好序的元组列表，其中每一个元组是一个（word，frequency）对。x轴上的值对应元组的索引，y轴的值对应该元组中单词的频率。将每个单词都作为x轴上的标签绘制出来是不实际的，尽管这是x轴所代表的含义。图1-4显示了与之前示例1-8中表格显示的同一单词数据的图表。图中y轴的值对应一个单词出现的次数。尽管每个单词的标签这里没有给出，但实际上x轴的值被排过序了，这样单词频率之间的关系就更加明显了。每个轴的数据都进行了对数比例的调整，从而对显示的曲线进行了压缩。这张图可以直接在IPython Notebook上用示例1-12所示的代码生成。

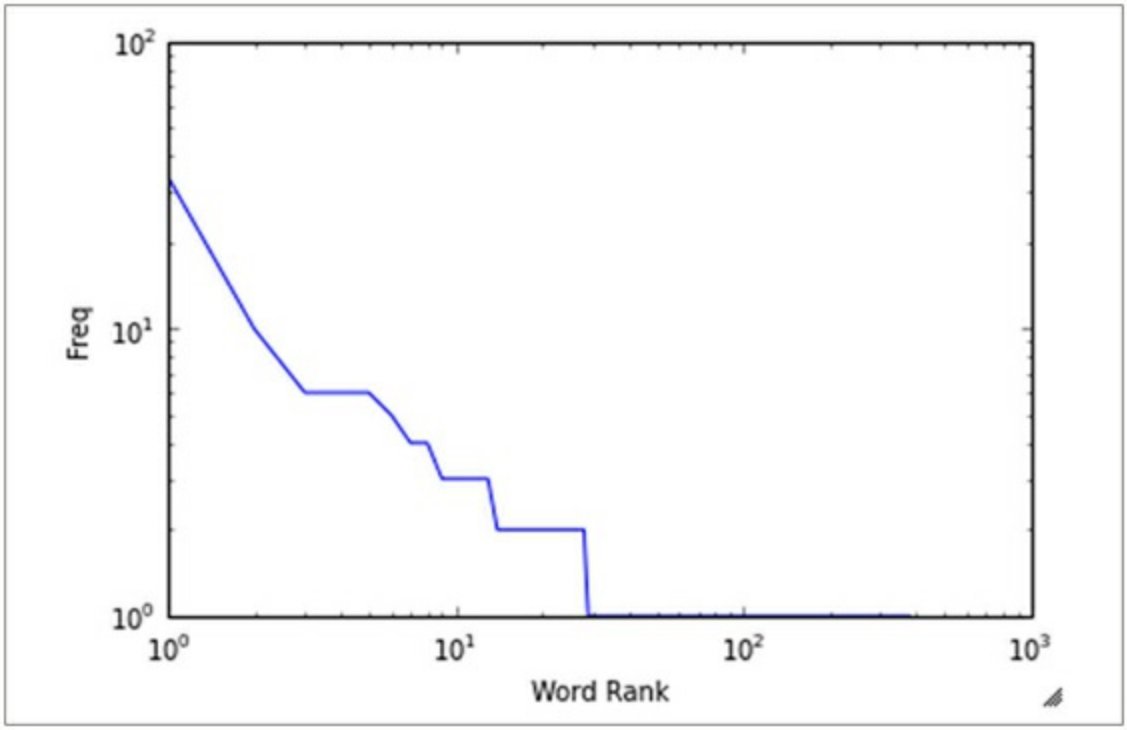


图1-4：单词按频率排序后的频率图，频率数据由示例1-8计算

注意：如果你使用虚拟机，则你的IPython Notebook应该进行配置以利用现成的绘图能力。如果你在使用自己的本地环境，确保在启动IPython Notebook时启用了PyLab（http://bit.ly/1a1l6BN），如下所示：

```
ipython notebook --pylab=inline
```

示例1-12：绘制单词的频率

```
   word_counts = sorted(Counter(words).values(), reverse=True)
plt.loglog(word_counts)
plt.ylabel("Freq")
plt.xlabel("Word Rank")
```

频率绘制成图是直观且方便的，如果可以进一步把数据的值划分成桶则更有用，其中每个桶对应一个频率的范围。例如，从中我们可以回答多少个词出现的频率介于1~5之间、5~10之间以及10~15之间等。直方图（http://bit.ly/1a1l6Sk）就是恰恰为此目的而设计的，它提供一个方便的可视化形式来把制表的频率显示为相邻的矩形。这里每个矩形面积代表落入特定频率范围的数据测量。图1-5和图1-6分别显示了来自示例1-8和示例1-10数据的直方图，尽管这些直方图没有x轴标签来显示该频率处的单词，但直方图本来就不是为此目的设计的。直方图对潜在的频率分布提供洞察力，而x轴的频率范围对应一组单词，每个单词的频率落在该范围，同时y轴代表落入该范围内全部单词的总频率。

当解释图1-5时，回头看下对应的制表数据，并考虑这儿有很多的

单词、昵称或主题标签只有较低的出现频率（文本中较少出现）。然而当我们把这些低频词合在一起，并放入一个命名为“频率介于1~10之间的单词”的范围，我们看到这些低频词的总计数量构成了文本的大部分互异词。更具体的，我们看到大约10个词构成绝大多数的频率，如图中的大矩形面积，而这里只有几个词拥有更高的频率，分别是“#MentionSomeoneImportantForYou”和“RT”，它们在我们制表数据中的频率分别为34和92。

类似地，我们在解释图1-6时看到，有少量的推文被转推的频率大大高于其他仅被单次转推的推文，而后者构成了推文的大多数，见直方图左边的最大矩形。

用于直接在IPython Notebook中产生这些直方图的代码在示例1-13和示例1-14中给出。花些时间探索matplotlib和其他科学计算工具的功能是值得的。

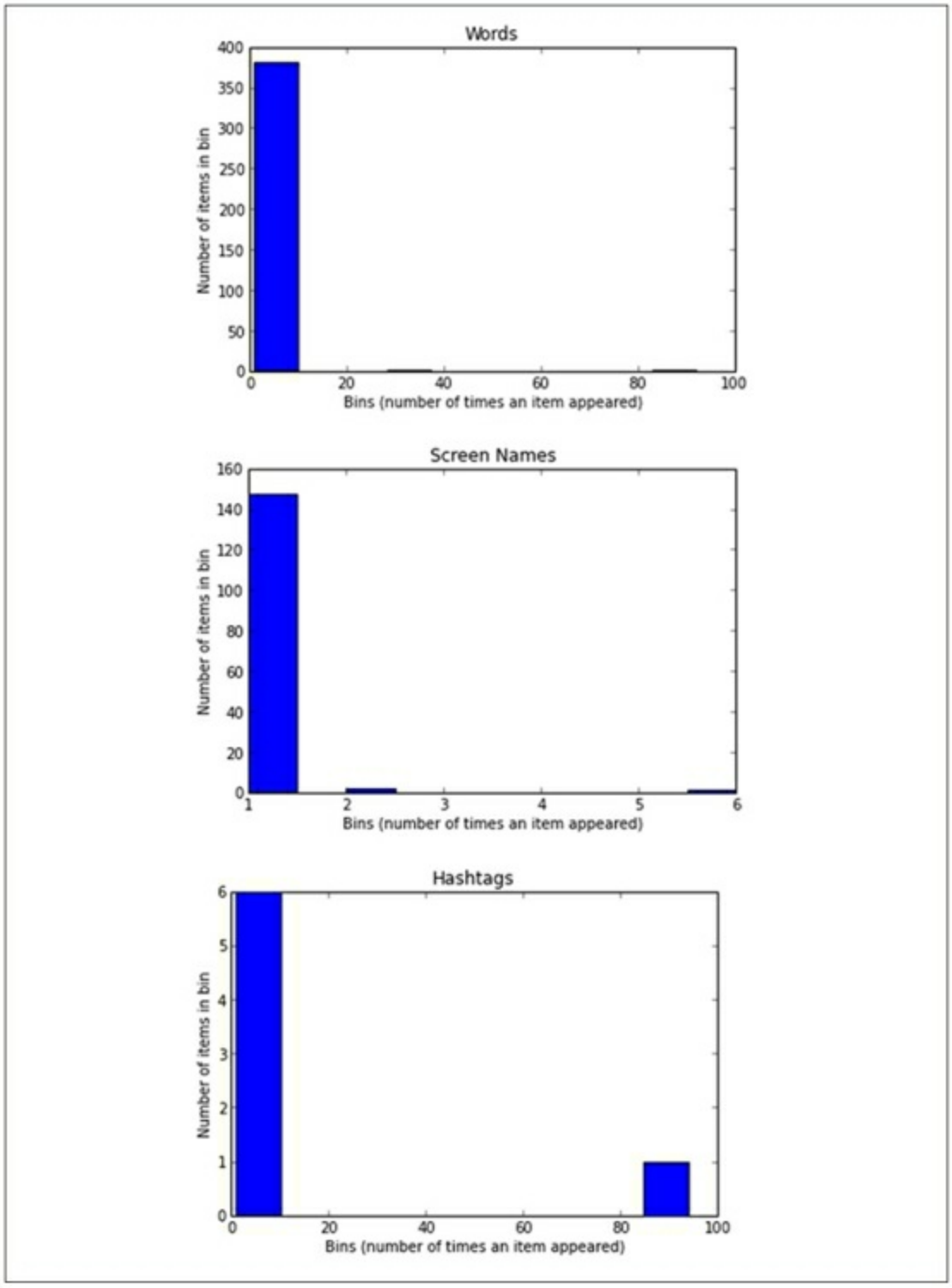


图1-5：针对单词、昵称和主题标签统计的频率数据直方图，每个桶显示一个根据频率划分的特定类型的数据

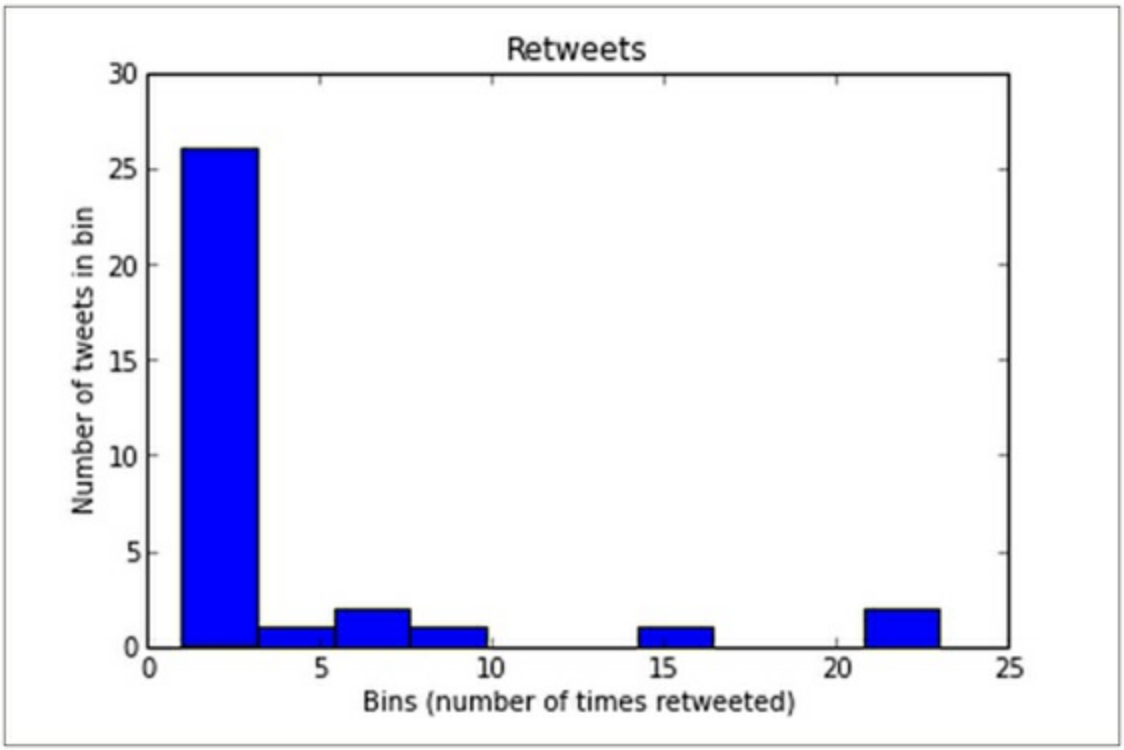


图1-6：转推频率直方图

警告：鉴于它们依赖链中的一些动态载入库的原因，安装如matplotlib这样的工具可能会碰到不少挫折。遇到的麻烦随工具的版本和操作系统的版本而不同。如果你还没有成功安装这些工具，强烈建议你借助本书的虚拟机体验（参见附录A的概述）。

示例1-13：生成单词、昵称和主题标签的直方图

```
for label, data in (('Words', words),
                    ('Screen Names', screen_names),
                    ('Hashtags', hashtags)):
 # Build a frequency map for each set of data
 # and plot the values
 c = Counter(data)
 plt.hist(c.values())
 # Add a title and y-label ...
 plt.title(label)
 plt.ylabel("Number of items in bin")
 plt.xlabel("Bins (number of times an item appeared)")
```

```
    # ... and display as a new figure
    plt.figure()
```

示例1-14：生成转推次数的直方图

```
   # Using underscores while unpacking values in
# a tuple is idiomatic for discarding them
counts = [count for count, _, _ in retweets]
plt.hist(counts)
plt.title("Retweets")
plt.xlabel('Bins (number of times retweeted)')
plt.ylabel('Number of tweets in bin')
print counts
```

1.5 本章小结

在本章中，我们将Twitter作为一个快速增长并且风靡一时的成功技术平台进行介绍，因为它能够满足一些人类与沟通、好奇心以及自组织行为相关的基本期望，这些期望产生于它混乱的网络动态性。这一章的示例代码是让你了解和运行Twitter API，演示如何轻松地使用Python以交互的方式探索和分析Twitter数据，并且提供一些可以用于挖掘推文的初始模板。我们从学习如何创建授权的连接开始了本章的内容，然后演示了如何发现特定地区的热门话题、如何搜索有趣的推文以及如何使用基本却非常有效的基于频率分析和简单统计量的技术对这些推文进行分析等一系列的例子。尽管这些热门话题看起来有点随意，但是结果却带给我们一些有价值的方向以及一些进行深入分析的可能性。

注意：第9章包含一些Twitter代码配方，它们包含了非常广泛的话题，从获得和分析推文到高效的存储存档推文，再到分析关注者获取见解的相关技术。

从分析的角度来看，本章的主要经验是：计数通常是任何有意义的定量分析的第一步。尽管基本的频率分析非常简单，但它是非常强大的工具，不应该仅仅因为直观而被忽视。此外，很多其他高级的统计学也依赖于它。相反，正是因为这样做很直观简单，频率分析和词汇丰富性

等度量应该被经常使用。最简单的技术得到的结果的质量通常（但并不总是这样的）比得上那些更复杂的分析方法得到的结果。对于Twitter虚拟空间中的数据，这些一般的技术通常会使你接近问题的答案，“现在人们正在谈论些什么呢？”现在，这是我们都希望知道的东西，不是吗？

注意：这一章和其他章列出的源代码可以从GitHub（http://bit.ly/1a1kNqy）中获得，它们是以好用的IPython Notebook格式存储的，因此我们强烈建议你在自己的Web浏览器中尝试这些代码。

1.6 推荐练习

·收藏Twitter API文档（http://bit.ly/1a1kSKQ）并且花些时间阅读它。尤其要花一些时间浏览REST API（http://bit.ly/1a1kZ9i）和平台对象（http://bit.ly/1a1kSL8）方面的信息。

·熟练地使用IPython（http://bit.ly/1a1laRY）和IPython Notebook（http://bit.ly/1a1laSf），把它们作为替代传统Python解释器的更高效的选择。在你从事社交网络数据挖掘的职业生涯中，由此节省的时间和提高的生产力是非常可观的。

·如果你的Twitter账号中有大量的推文，那么从账号设置（account setting）（http://bit.ly/1a1lb8D）中请求你的历史推文存档并对它进行分析。导出的账号数据包括以传统的JSON格式存储的按时间组织的文件。详细信息请查看包含在下载的存档中的README.txt文件。你的推文中最常出现的词是哪些？谁经常转发？你有多少条推文被转发了（以及你认为为什么会这样）？

·花些时间，在开发者控制台（http://bit.ly/1a1kWui）下研究Twitter REST API。尽管在这一章我们选择了编程的方式使用Python中的twitter包，然而控制台对于探索API、参数的选择等方面是非常有用的。如果你喜欢在终端下工作，那么命令行工具Twurl（http://bit.ly/1a1kZq1）是

值得你考虑的另一种选择。

·完成一个练习，判断那些转推“God”对他们很重要的这条状态的人是否有精神或宗教信仰，或者依照本章的工作流程分析某一个热门话题或你自己选择的搜索查询。研究一些可用于更加准确查询的高级搜索特性（http://bit.ly/1a1l3pN）。

·研究Yahoo！GeoPlanet的Where On Earth ID API（http://yhoo.it/1a1kZ9u），这样你就可以比较不同地区的热门话题。

·仔细了解matplolib（http://bit.ly/1a1l7Wv）并且学习如何使用IPython Notebook创建漂亮的二维和三维数据图表（http://bit.ly/1a1lccP）。

·研究并应用第9章的一些练习。

1.7 在线资源

下面是本章出现的一些链接，它们对于回顾本章的内容可能会很有用处：

·使用IPython Notebook绘制漂亮的二维和三维数据图（http://bit.ly/1a1lccP）

·IPython的“魔法函数”（http://bit.ly/1a1kXyf）

·json.org（http://bit.ly/1a1l2lJ）

·PyLab（http://bit.ly/1a1l6BN）

·Python列表解析（http://bit.ly/1a1l1hy）

·官方Python教程（http://bit.ly/1a1l1hy）

·OAuth（http://bit.ly/1a1kZWN）

·Twitter API文档（http://bit.ly/1a1kSKQ）

·Twitter API速率访问限制1.1版（http://bit.ly/1a1l2ly）

·Twitter开发者控制台（http://bit.ly/1a1kWui）

·Twitter开发者守则（http://bit.ly/1a1kX1a）

·Twitter的OAuth文档（http://bit.ly/1a1kZWW）

·Twitter搜索API操作符（http://bit.ly/1a1l3pN）

·Twitter信息流API（http://bit.ly/1a1l1ya）

·Twitter服务条款（http://bit.ly/1a1kWKB）

·Twurl（http://bit.ly/1a1kZq1）

·Yahoo！GeoPlanet的Where On Earth ID API（http://yhoo.it/1a1kZ9u）

第2章　挖掘Facebook：分析粉丝页面、查看好友关系等

在这一章，我们将利用（社交）图谱API开始进军到Facebook平台并且探索其中巨大的可能性。由于Facebook的10亿用户[1]中超过一半的人每天活跃地更新状态、上传照片、交流信息、实时对话、在实际地点签到、玩游戏、购物以及参加其他任何你可以想象到的活动，因此Facebook可以说是社交网络的核心，是一个集全部功能于一身的奇迹。从社交网络挖掘的角度看，Facebook存储的关于个人、团体以及产品的数据财富是非常振奋人心的，因为Facebook简洁的API展现了令人难以置信的可能，就是将其整合为信息（世界上最珍贵的商品）并且收集有价值的观点。另一方面，这种强大的能力对应着巨大的责任，Facebook已经提交了一套已知最复杂的在线隐私控制（http://on.fb.me/1a1llg9），从而保护用户免遭入侵。

值得注意的是，尽管Facebook自称为一种社交图谱，但是同时它也在稳定的转型为有价值的兴趣图谱，因为它在Facebook页面和点“赞”（likes）特性中维护了人物与其感兴趣的事物之间的关系。在这方面，你可能会越来越多的听到它被描述为一种“社交兴趣图谱”。在大多数情况下，你可以认定兴趣图谱无疑是存在的，并且可以从多数社交数据源创建。例如，第1章中的Twitter实际上是一种兴趣图谱，因为它

使用了人物、地点或者事物之间非对称的“关注”关系。Facebook为一种兴趣图谱这个概念将会贯穿在这一章的内容中，在第7章我们将会回到从社交数据中明确地创建一个兴趣图谱的构想上。

在本章的余下内容中，我们假设你已拥有一个活跃的Facebook账号（http://on.fb.me/1a1lkcd），这是使用Facebook API必需的。尽管分析公共信息也很有趣，但是从你自己的社交网络中分析数据会更有趣，因此如果你刚刚接触Facebook，那么你值得花些时间添加一些新的好友。

注意：在http://bit.ly/MiningTheSocialWeb2E上可以找到本章（及所有其他章节）最新修订bug的源代码。同时也要利用好本书的虚拟机，如附录A中描述的，来尽可能的享用样例代码。

[1] Internet使用统计（http://bit.ly/1a1ljF8）指出，全球人口总数大约是70亿，而Internet用户估计接近25亿。

2.1 概述

由于这是全书的第2章，因此我们将要涉及的概念要比第1章里的稍微复杂一些，但是对于广大读者来说仍然应该是非常容易接受的。在这一章，你将会学习：

·Facebook的社交图谱API以及如何发起API请求。

·开放图协议（Open Graph protocol）以及它与Facebook社交图谱（Social Graph）的关系。

·从Facebook页面和Facebook好友中分析“赞”。

·用来分析社交图谱的团分析（clique analysis）等技术。

·使用D3JavaScript库对社交图谱进行可视化。

2.2 探索Facebook的社交图谱API

Facebook平台是一个成熟的、健壮的并且文档详尽的门户网站，通过它我们可以接触到有史以来积累的最全面并且经系统化组织的信息存储，无论是从广度还是从深度上来看都是这样的。它的广度体现在其用户群代表了全部现存人口的七分之一，而它的深度体现在对于某些特定用户所能了解的信息量上。Twitter有独特的非对称好友模型，用户可以自由而有条件地关注其他用户而不需要特别的同意，而Facebook的好友模型是对称的并且需要用户之间相互同意才能查看对方的交互和活动。

此外，Twitter中用户间所有的交互除了私人消息之外几乎都是公共状态，而Facebook允许更多细粒度的隐私控制，这样好友关系可以像列表一样进行组织和维护，在不同活动中的好友呈现不同的可视级别。比如，你可能选择分享一个链接或一张照片给某一特定列表中的好友而不是你全部的社交网络。

作为一位社交网络挖掘者，访问一位Facebook用户的账号数据的唯一方法是注册一个应用程序，然后将这个应用程序作为进入Facebook开发者平台的切入点。此外，对于一个应用程序唯一可用的数据是用户特别授权它访问的那些。例如，作为编写一个Facebook应用程序的开发者，你将会是登录到该应用程序的用户，那么它就能够访问你特别对它

授权的那些数据。在这种情况下，作为Facebook用户你可能会认为应用程序有些像你的Facebook好友，因为应用程序可以访问什么最终都受到你的控制并且你可以在任何时候撤销访问授权。Facebook平台政策文档（http://bit.ly/1a1lm3C）是每个Facebook开发者必须要阅读的，因为它为Facebook用户提供了一套完整的权利和义务以及Facebook开发者相关的法律精神和条文。如果你还没有读过，你值得花些时间阅读一下Facebook的开发者政策并且收藏Facebook开发者首页（http://bit.ly/1a1lm3Q），因为它是进入Facebook平台及其文档的入口。

注意：请记住，作为一位挖掘自己的账号的开发者，让你的应用程序访问所有你自己的账号数据可能不会出现问题。然而，如果你立志要开发一个成功的托管应用程序并且该程序需要访问数据量更多的数据，那么请小心，因为很有可能某个用户将会无法识别或信任你的应用程序从而使你的程序无法控制该级别的权限（这是理所当然的）。

尽管随后的章节中我们将会在程序中访问Facebook平台，但是Facebook提供了许多有用的开发者工具（http://bit.ly/1a1lnVf），包括一个图谱API管理工具程序（http://bit.ly/1a1lnVv），我们将会使用它初步熟悉社交图谱。这个程序提供了一种直观并且完整的查询社交图谱的方式，一旦你适应了社交图谱的工作方式，那么将查询翻译成Python代码可以实现自动化而且后续处理就变得十分自然了。尽管我们将要讨论图

谱API，但是你从精心编写的“图谱API入门指南”文档（http://bit.ly/1a1lobU）开始阅读会很受益的。

警告：除了图谱API，你可能也会遇到Facebook查询语言（Facebook Query Language，FQL）（http://bit.ly/1a1lmRd）。现在它是指Legacy REST API（http://bit.ly/1a1lmRo）。注意，尽管FQL仍然被使用并且这一章我们会对其做简短的介绍，但是Legacy REST API已经过时了不久就会被淘汰。在用Facebook进行开发时千万不要使用它。

2.2.1 理解社交图谱API

正如名字所暗示的，Facebook的社交图谱是一个大规模的图（http://bit.ly/1a1loIX）数据结构，它代表了社会交互并且包含了节点和节点之间的连接。图谱API提供了与社交图谱交互的主要方法，熟悉图谱API的最佳方法是花些时间摆弄一下图谱API管理工具（Graph API Explorer）（http://bit.ly/1a1lnVv）。

注意，图谱API管理工具并不是什么特殊的工具。除了能够载入事先保存的访问令牌和调试它们外，它是一种寻常的Facebook应用程序并且使用了与其他开发者应用程序一样的开发者API。实际上，当你有与正在开发的程序的一组特定授权相关联的OAuth令牌并且想运行一些查询作为探索性开发工作或调试周期的一部分时，图谱API管理工具就会

显得非常方便。在编程访问图谱API时我们将会回顾这个总体思路。图2-1到2-4说明了单击加号（+）并且添加连接和字段后得到的一系列图谱API查询。对于这个查询，有一些事项需要我们注意：

访问令牌

出现在应用程序中的访问令牌是一个由登入用户授权的OAuth（http://bit.ly/1a1kZWN）令牌，它和你的应用程序访问数据时所需的OAuth令牌是同一个。在这一章，我们会一直使用这个访问令牌，但是你可以参考附录B中对OAuth的简要介绍，包括为了获得访问令牌在Facebook上实现OAuth流的细节。正如第1章提到的一样，如果这是你第一次遇到OAuth，你只要知道这个协议是一种网络标准，代表开放授权就够了。简而言之，OAuth是一种允许用户对第三方应用程序授权，从而使其能够访问这些用户的账号数据而无需共享密码等敏感信息的一种方式。

注意：实现一个OAuth 2.0流的时候，你的应用程序需要获得某个用户的授权，才能访问其账号的数据，详情见附录B。

节点ID

该查询的基础是ID（标识）为“644382747”节点，它对应着名为“Matthew A.Russell”的人，他作为当前登录用户已被图谱API管理工具预先加载。节点的“id”和“name”值被称为字段。我们以后就会看到查

询的基础可以是其他任何的节点。这样对图进行遍历以及查询其他节点（可能是人或者图书、电视表演等事物）就会非常自然。

关系约束

你可以使用“好友”这个关系对原始查询进行修改，如图2-2所示，单击“+”然后再滚动到“关系”弹出菜单中的“好友”。控制台中出现的“好友”关系代表着那些连接到原始查询节点的节点。现在，你可以在这些节点中单击任何蓝色的ID字段然后以该节点作为基础初始化一个新的查询。在网络科学的术语中，我们得到被称为“自我图”（ego graph）的图，因为它有一个执行者（或个体）作为焦点或逻辑中心，并且连接到周围的其他节点上。如果你画出来的话，个体图很像是一个轮毂连接许多轮辐。

“赞”约束

我们可以对每个好友添加“赞”约束从而对原始查询进行更深入的修改，如图2-3所示。然而，在你能获取好友的“赞”关系之前，你必须通过更新访问令牌然后同意对图谱API管理工具程序进行授权从而使其能够访问好友的“赞”信息，如图2-4所示。图谱API管理工具允许通过单击获取访问令牌（Get Access Token）按钮，然后在好友数据许可标签中选择“friends_likes”复选框从而轻松地对其进行授权。我们得到的仍然是自我图，但是它可能会复杂很多，因为图中可能会存在许多额外的节点

以及节点间可能出现的连接。

调试

调试按钮对基于访问令牌的授权情况下没有按照期望返回数据的那些查询很有帮助。

JSON响应格式

一个图谱API查询的结果是以非常方便的JSON格式返回的，我们可以很容易的对其进行操作和处理。

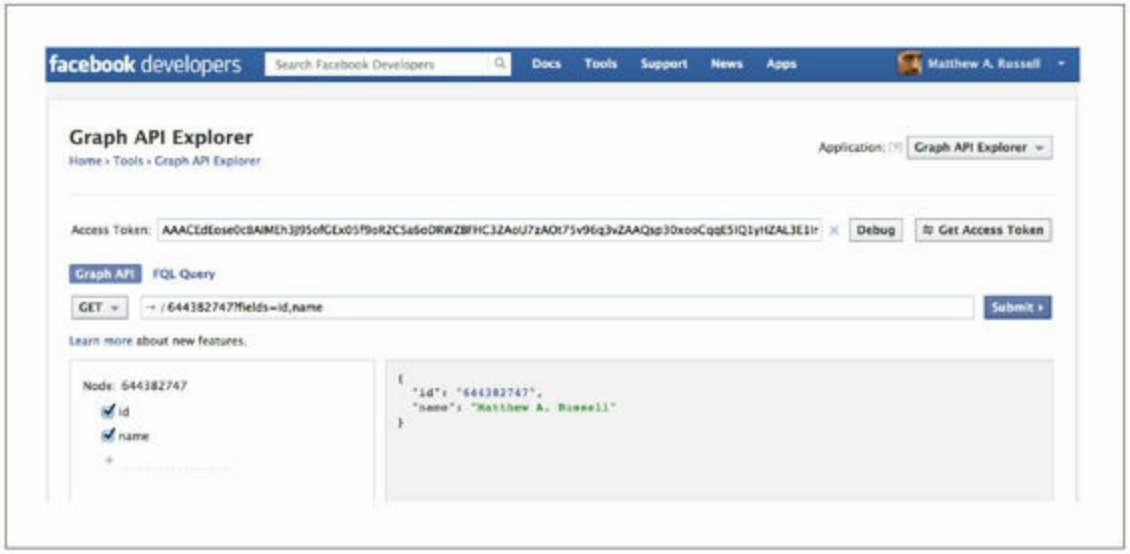


图2-1：使用图谱API管理工具程序逐步构建对好友兴趣的查询：对社交图谱一个节点的查询

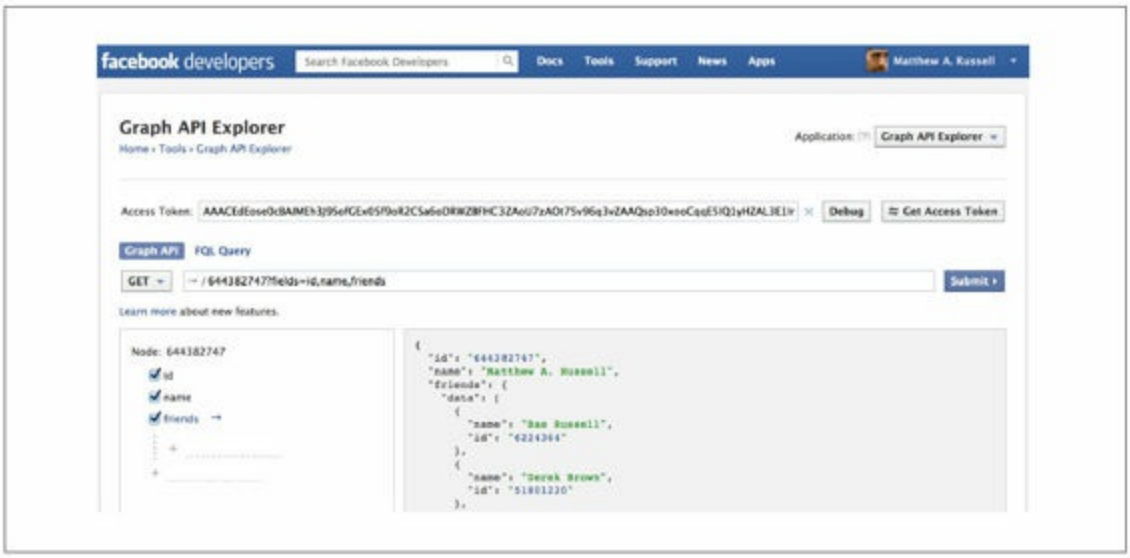


图2-2：使用图谱API管理工具程序逐步构建对好友兴趣的查询：对社交图谱一个节点以及和好友之间关系的查询

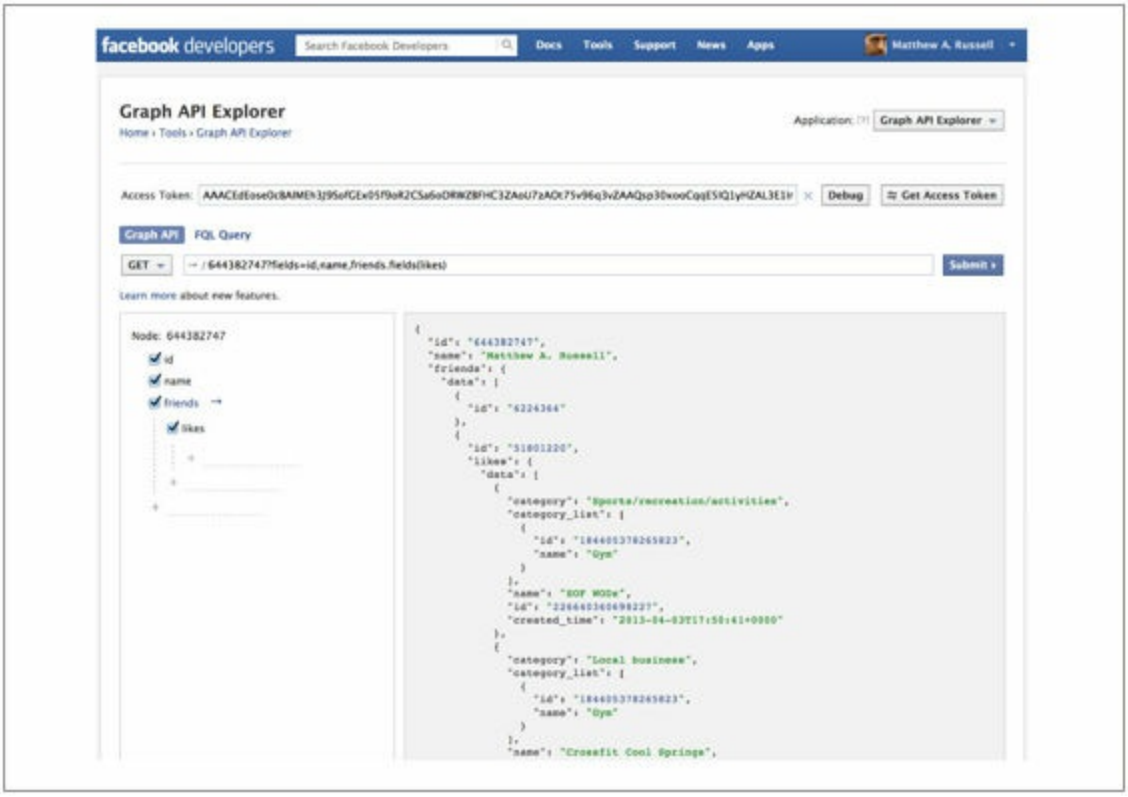


图2-3：使用图谱API管理工具程序逐步构建对好友兴趣的查询：对社交

图谱一个节点、与好友之间的关系以及好友的“赞”的查询

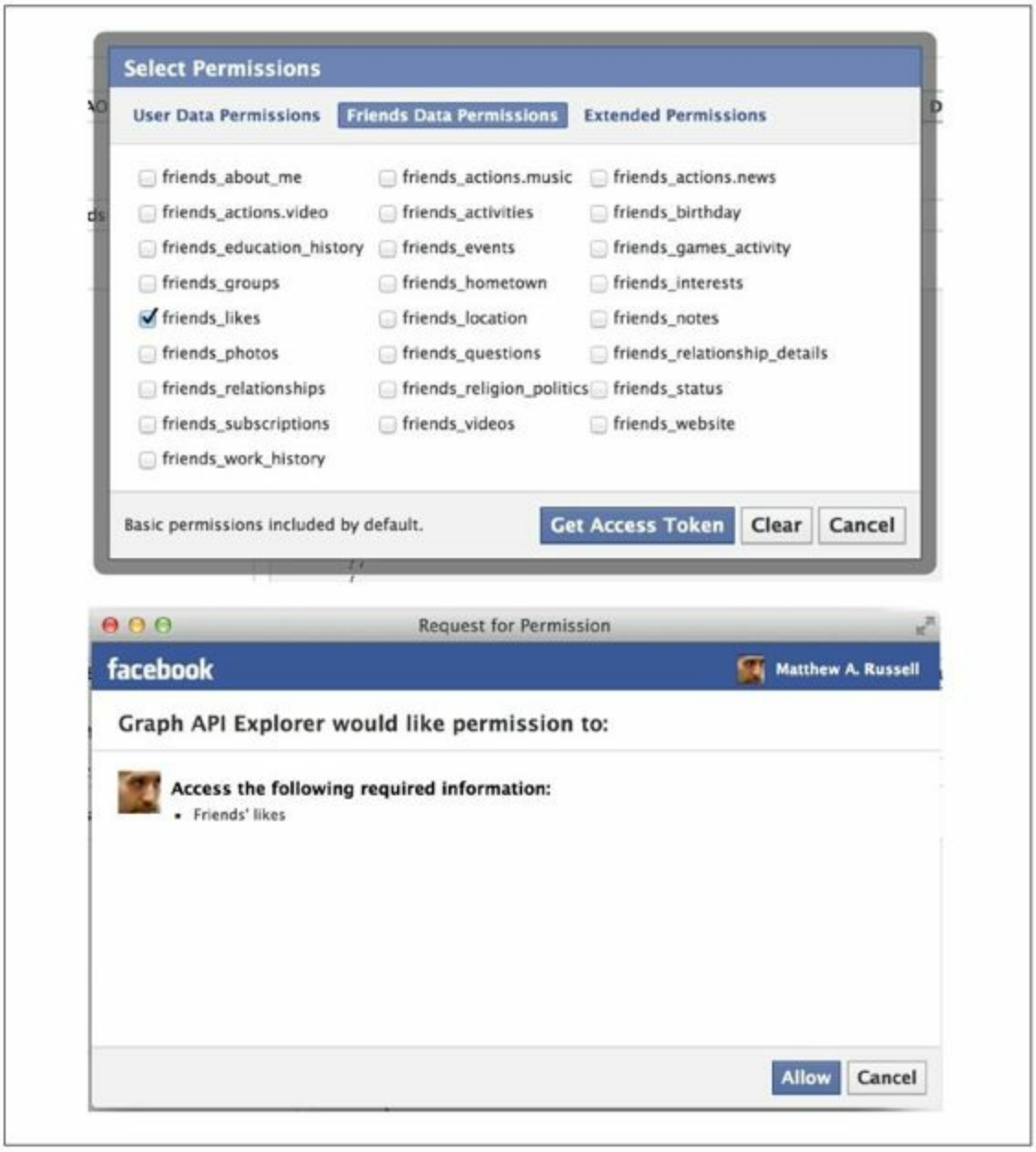


图2-4：Facebook应用程序为了访问用户的账户数据必须申请授权。上图：图谱API管理工具许可面板。下图：Facebook中为了使图谱API管理工具程序能够访问朋友的“赞”数据而提交授权的对话框

Facebook查询语言

除了图谱API之外，FQL为查询Facebook的社交图谱提供了另一种很好的选择，并且它有SQL风格的语法，这对大多数开发者来说是很直观的。尽管有些高级查询只能使用SQL获得而不能通过图谱API获得，但是几乎任何能够使用图谱API查询到的数据，你都可以通过FQL查询到。看起来Facebook的长期计划似乎是确保图谱API和FQL能够实现相同的功能。例如，一些近期的投入帮助图谱API获得许多强大的新特性，包括字段扩展和嵌套（field expansion and nesting）（http://bit.ly/1a1lsIE）等。如果你对学习FQL很感兴趣，可以查看FQL参考手册（http://bit.ly/1a1lmRd），并且使用FQL查询控制台作为图谱API管理工具的替代工具试着执行一些查询。例如，你可以在图谱API管理工具的FQL查询标签栏中使用以下查询查询好友的姓和名：

```
    select first_name, last_name
from user
where uid in (
  select uid2
  from friend
  where uid1 = me()
)
```

尽管随后我们在这一章将使用一个Python包编写程序探索图谱API，然而你可以选择通过图谱API管理工具模拟请求从而更加直接的通过HTTP创建图谱API查询。例如，示例2-1使用requests（http://bit.ly/1a1lrEt）包简化了创建一个获取好友信息以及他们“赞”信息的HTTP请求的过程（而不是使用Python标准库中更加繁琐的包，比如urllib2）。你可以在终端里使用pip install requests命令安装该

包。查询是由字段（fields）参数中的值决定的，这与我们在图谱API管理工具中以交互方式构建的查询是一样的。我们特别感兴趣的是friends.limit（10）.fields（likes.limit（10））这个语法使用了图谱API中相对较新的一个特性——字段扩展（http://bit.ly/1a1lsIE），该特性可以在一个API调用中创建多条查询并对其参数化。

示例2-1：通过HTTP创建图谱API请求

```
   import requests # pip install requests
import json
base_url = 'https://graph.facebook.com/me'
# Get 10 likes for 10 friends
fields = 'id,name,friends.limit(10).fields(likes.limit(10))'
url = '%s?fields=%s&access_token=%s' % \
    (base_url, fields, ACCESS_TOKEN,)
# This API is HTTP-based and could be requested in the browser,
# with a command line utlity like curl, or using just about
# any programming language by making a request to the URL.
# Click the hyperlink that appears in your notebook output
# when you execute this code cell to see for yourself...
print url
# Interpret the response as JSON and convert back
# to Python data structures
content = requests.get(url).json()
# Pretty-print the JSON and display it
print json.dumps(content, indent=1)
```

如果你试图设置fileds='id，name，friends.fields（likes）执行一条获取所有好友“赞”的查询，那么这个脚本或许会终止，这很可能是因为你有很多的好友，而每个好友也有很多“赞”。如果发生了这种问题，你可能需要在查询的字段中添加约束和偏移，像Facebook的字段扩展（http://bit.ly/1a1lsIE）文档描述的那样。然而，随后在本章你将学到的facebook包解决了其中的一些问题，因此我们推荐你先尝试使用它。第一个示例仅仅是为了说明Facebook API是建立在HTTP之上的。下列一

些字段约束、偏移的例子说明了字段选择器的一些可能情况：

```
   # Get all likes for 10 friends
fields = 'id,name,friends.limit(10).fields(likes)'
# Get all likes for 10 more friends
fields = 'id,name,friends.offset(10).limit(10).fields(likes)'
# Get 10 likes for all friends
fields = 'id,name,friends.fields(likes.limit(10))'
```

在写作本书时，查询的默认限制是返回最多5000项，但是通常你并不会创建返回多于5000项的图谱API查询；如果你这样做了，可以参考分页文档（pagination documentation）（http://bit.ly/1a1ltMP）从而了解如何在结果的多个页之间进行导航。

2.2.2 理解开放图协议

你可以使用强大的图谱API，遍历社交图谱以及查询熟悉的Facebook对象，除此之外，你也应该知道Facebook早在2010年4月就推出的开放图协议（Open Graph protocol，OGP）（http://bit.ly/1a1lu3m），它与社交图谱是在同一届的F8会议上公布的。简而言之，OGP是一种允许开发者通过向任何页面注入一些RDFa元数据（http://bit.ly/1a1lujR）从而将网页作为一个Facebook社交图谱对象的机制。因此，除了能够在Facebook的“围墙花园”内访问图谱API参考手册（http://bit.ly/1a1lvEr）中描述的一些对象（用户、图片、视频、签到、链接、状态消息等）外，你也可能在网络中遇到过那些被接入到

社交图谱并代表一定概念的页面。换句话说，OGP是一种通往社交图的方式，在Facebook的开发者文档中这些概念被描述为“开放图”[1]。

实际上，有很多种方法可以使用OGP将网页以有价值的方式接入到社交图谱中，很有可能你已经遇到过这些网页却从未意识到。例如，考虑图2-5，它显示了IMDb.com（http://imdb.com/）中The Rock这部电影的页面。在右侧的边栏中你会看到一个非常熟悉的“赞”按钮以及“19319人对它点了赞，快去成为你朋友圈的第一人”这条消息。IMDb为每个可以纳入社交图谱中的对象URL实现OGP，从而启用了上述功能。页面中有了正确的RDFa元数据，然后Facebook就能够明确地与这些对象进行联系并将它们合并到活动流以及Facebook用户体验的其他关键因素中。

图2-5：一个为The Rock实现OGP的IMDb页面

OGP的展示实现类似于网页上“赞”按钮，如果你过去习惯看到它们，那么你会觉得它们是很自然的，但是Facebook成功地开放其开发平

台并允许任意的纳入网络上的对象的影响是非常深远的。

例如，在2013年初编写这本书的时候，Facebook已经开始了对少数用户推出新的图搜索产品（http://on.fb.me/1a1lv7b）。Google这些公司为了支持搜索，对整个网络进行抓取和建索引，而Facebook的图搜索的基本思路与此不同：你在搜索栏输入任何事物，这和传统的Google用户体验一样，但是你获得的返回结果却会基于Facebook上大量关于你的信息进行定制。OGP已经建立得很完善了，问题是Facebook的图搜索结果并不会局限于Facebook用户体验内的事情，因为来自网络的其他联系也从本质上被结合到社交图谱中了。考虑到Facebook的用户群，思考颠覆性的社交图谱如何给网络带来广泛的影响超出了我们的范围，但是这却是一个值得你花些时间思考的练习。

在讨论图谱API查询之前，让我们简要地看一看实现OGP的要点。OGP文档中的经典例子说明了如何将IMDb中The Rock的页面变成开放图协议中的一个对象，并将该对象作为使用如下命名空间的XHTNL文档的一部分：

```
   <html xmlns:og="http://ogp.me/ns#">
<head>
<title>The Rock (1996)</title>
<meta property="og:title" content="The Rock" />
<meta property="og:type" content="movie" />
<meta property="og:url" content="http://www.imdb.com/title/tt0117500/" />
<meta property="og:image" content="http://ia.media-imdb.com/images/rock.jpg" />
...
</head>
...
</html>
```

一旦大规模地实现了，这些元数据就会有巨大的潜力，因为它们支持使用类似http://www.imdb.com/title/tt0117500的URI以机器可读的方式明确地表示任何网页（无论是人、公司还是产品等），从而进一步推动对语义网的构想。除了能够对The Rock点“赞”之外，用户可以通过自定义操作与该对象进行交互。例如，因为这是一部电影，用户可以表明他们曾经看过The Rock（http://bit.ly/1a1lwrU）。OGP允许用户与对象之间进行一组大范围的、灵活的活动，这是社交图谱的一部分。

注意：如果你还没有浏览该网页的HTML源代码，可以通过http://www.imdb.com/title/tt0117500访问，自己看一看原始的RDFa是什么样的。

使用图谱API查询开放图open Graph对象是非常简单的：在http（s）//graph.facebook.com/上添加一个网页URL或者对象的ID就可以获取该对象的详细信息。例如，在你的浏览器中访问URL http://graph.facebook.com/http://www.imdb.com/title/tt0117500将会返回下面的相应信息：

```
   {
   "id": "114324145263104",
   "name": "The Rock (1996)",
   "picture": "http://profile.ak.fbcdn.net/hprofile-ak-snc4/hs344.snc4/...jpg",
   "link": "http://www.imdb.com/title/tt0117500/",
   "category": "Movie",
   "description": "Directed by Michael Bay.  With Sean Connery, ...",
   "likes" : 3
}
```

如果你查看URL http://www.imdb.com/title/tt0117500的源码，你会发现响应信息中的字段与页面元标签中的数据相对应，这并不是巧合。通过一条简单的查询，返回丰富的元数据，这正是设计OGP的初衷。更加有趣的是你可以通过在请求上添加查询字符串参数metadata=1明确地请求额外的元数据。下面是查询https://graph.facebook.com/114324145263104?metadata=1的示例响应，其中我们使用了ID而不是IMDB网页的URL：

```
{
"id": "114324145263104",
"name": "The Rock (1996)",
"picture": "http://profile.ak.fbcdn.net/hprofile-ak-snc4/..._s.jpg",
"link": "http://www.imdb.com/title/tt0117500",
"category": "Movie",
"website": "http://www.imdb.com/title/tt0117500",
"description": "Directed by Michael Bay.  With Sean Connery,  ...",
"about": "Directed by Michael Bay.  With Sean Connery, Nicolas Cage, ...",
"likes": 8606,
"were_here_count": 0,
"talking_about_count": 0,
"is_published": true,
"app_id": 115109575169727,
"metadata": {
   "connections": {
      "feed": "http://graph.facebook.com/http://www.imdb.com/title/...",
      "posts": "http://graph.facebook.com/http://www.imdb.com/title/...",
      "tagged": "http://graph.facebook.com/http://www.imdb.com/title/...",
      "statuses": "http://graph.facebook.com/http://www.imdb.com/title/...",
      "links": "http://graph.facebook.com/http://www.imdb.com/title/...",
      "notes": "http://graph.facebook.com/http://www.imdb.com/title/...",
      "photos": "http://graph.facebook.com/http://www.imdb.com/title/...",
      "albums": "http://graph.facebook.com/http://www.imdb.com/title/...",
      "events": "http://graph.facebook.com/http://www.imdb.com/title/...",
      "videos": "http://graph.facebook.com/http://www.imdb.com/title/...",
   },
   "fields": [
      {
         "name": "id",
         "description": "The Page's ID. Publicly available. A JSON string."
      },
      {
         "name": "name",
         "description": "The Page's name. Publicly available. A JSON string."
      },
      {
         "name": "category",
         "description": "The Page's category. Publicly available. ..."
```

```
        },
        {
           "name": "likes",
           "description": "\\* The number of users who like the Page..."
        },
        ...
     ]
  },
  "type": "page"
}
```

---

metadata.connections中的项是指向图中其他节点的指针，你可以对它们进行抓取从而获得更多有趣的信息。例如，你可以按照“photos”链接下载与电影相关的图片，或者浏览图片相关的链接发现谁上传了它们或者看一看关于它们的评论。万一他无法正常工作，你也可以把自己作为图中的一个对象。试着访问相同的URL前缀，但是将后面的内容替换成你自己的FacebookID或用户名作为URL的上下文，然后你就可以看到自己了（例如，在你的浏览器中访问 https://graph.facebook.com/<YOUR_FB_ID>）。

注意：试试使用Facebook ID“MiningTheSocialWeb”通过图谱API管理工具获取本书官方Facebook粉丝页面（http://on.fb.me/1a1lAI8）的详细信息。你同样可以修改示例2-1，编写程序对 https://graph.facebook.com/MiningTheSocialWeb查询从而获取基本的页面信息，包括发布到页面上的内容。例如，在URL后添加“?fields=posts”这样的限定符进行查询，这样将会返回该页面发布内容的列表。

在编写程序访问图谱API之前的最后一条建议是：当我们认为OGP

具有前瞻性和创新性的时候，请不要忘记，它仍然在不断发展。由于它和语义网以及一般网络标准相关，“开放”这个词的使用（http://tcrn.ch/1a1lAYF）会让你很自然地产生一些恐慌。规范中的一些问题已经被解决了（http://bit.ly/1a1lAbd）并且有一些将来有可能被解决。可以认为OGP是单一提供商的努力产物，尽管从社会影响看非常不同，但是它的功能和很早之前网络上的meta元素（http://bit.ly/1a1Bma）不分上下。

OGP和图搜索是否某一天能够支配网络是一个具有很大争议性的话题，这种可能性很显然是存在的，种种迹象表明它的成功是趋向于积极方向的，很多令人激动人心的事物将会在未来发生并且创新也将不断的持续。既然现在你已经对社交图谱的背景有了全面的了解，让我们回过头熟悉一下如何访问图谱API。

[1] 贯穿于本节中对OGP实现的描述，术语“社交图谱”通常同时指代社交图谱和开放图，若有例外，会明确说明。

2.3 分析社交图谱联系

图谱API的一个官方Python SDK是Facebook以前维护的代码库（repository）的社区版本并且能够使用pip install facebook-sdk标准命令进行安装。这个包包含了一些非常有用且十分方便的方法，能够使你通过多种方式与Facebook进行交互，包括创建FQL查询、发布状态或照片等功能。然而，为了使用图谱API获取数据你仅仅需要知道GraphAPI类（定义在facebook.py源文件中）中几个少量的关键方法，因此如果你喜欢你可以选择直接使用requests（如示例2-1所示）通过HTTP进行查询。这些方法是：

```
    get_object(self, id, **args)
      用法示例: get_object("me", metadata=1)
get_objects(self, id, **args)
      用法示例: get_objects(["me", "some_other_id"], metadata=1)
get_connections(self, id, connection_name, **args)
      用法示例: get_connections("me", "friends", metadata=1)
request(self, path, args=None, post_args=None)
      用法示例: request("search", {"q": "social web", "type": "page"})
```

注意：和其他社交网络不同，Facebook API的访问次数限制（http://on.fb.me/1a1LDDU）并没有明确公布在指南中。尽管这些API可以很开放的使用，但是你仍然应该小心的设计你的应用程序，尽可能少的使用这些API并且解决任何的错误情况是我们最推荐的做法。

你将会使用最常见（并且经常是唯一的）的关键词参数是

metadata=1，它是为了不光返回对象本身的详细信息还返回与该对象相关的联系。看一下示例2-2，它引入了GraphAPI类并且使用了其中的get_objects方法查询与你相关的信息、好友的相关信息以及与术语社交网络相关的内容。这个示例同样引入了一个叫做pp的帮助函数，在这一章里它是用来将结果显示成格式更加优美的JSON从而免去一些输入。

示例2-2：使用Python查询图谱API

```
   import facebook # pip install facebook-sdk
import json
# A helper function to pretty-print Python objects as JSON
def pp(o):
    print json.dumps(o, indent=1)
# Create a connection to the Graph API with your access token
g = facebook.GraphAPI(ACCESS_TOKEN)
# Execute a few sample queries
print '---------------'
print 'Me'
print '---------------'
pp(g.get_object('me'))
print
print '---------------'
print 'My Friends'
print '---------------'
pp(g.get_connections('me', 'friends'))
print
print '---------------'
print 'Social Web'
print '---------------'
pp(g.request("search", {'q' : 'social web', 'type' : 'page'}))
```

示例2-2查询的示例结果如下所示，比较好理解。如果你使用的是图谱API管理工具，那么结果将会是一样的。在开发中，取决于你的具体目标，将图谱API管理工具与IPython或IPython Notebook协同使用常常是非常方便的。图谱API管理工具的优点是你可以很方便地单击ID的值，通过探索性的尝试产生新的查询。示例2-2的结果如下：

```
   ---------------
Me
---------------
{
"last_name": "Russell",
 "relationship_status": "Married",
 "locale": "en_US",
 "hometown": {
  "id": "104012476300889",
  "name": "Princeton, West Virginia"
 },
 "quotes": "The only easy day was yesterday.",
 "favorite_athletes": [
  {
   "id": "112063562167357",
   "name": "Rich Froning Jr. Fan Site"
  }
 ],
 "timezone": -5,
 "education": [
  {
   "school": {
    "id": "112409175441352",
    "name": "United States Air Force Academy"
   },
   "type": "College",
   "year": {
    "id": "194603703904595",
    "name": "2003"
   }
  }
 ],
 "id": "644382747",
 "first_name": "Matthew",
 "middle_name": "A.",
 "languages": [
  {
   "id": "106059522759137",
   "name": "English"
  },
  {
   "id": "312525296370",
   "name": "Spanish"
  }
 ],
 "location": {
  "id": "103078413065161",
  "name": "Franklin, Tennessee"
 },
 "email": "ptwobrussell@gmail.com",
 "username": "ptwobrussell",
 "bio": "How I Really Feel About Using Facebook (Or: A Disclaimer)...",
"birthday": "06/17/1981",
 "link": "http://www.facebook.com/ptwobrussell",
 "verified": true,
 "name": "Matthew A. Russell",
"gender": "male",
 "work": [
  {
   "position": {
    "id": "135722016448189",
```

```
    "name": "Chief Technology Officer (CTO)"
   },
   "start_date": "0000-00",
   "employer": {
    "id": "372007624109",
    "name": "Digital Reasoning"
   }
  }
 ],
 "updated_time": "2013-04-04T14:09:22+0000",
 "significant_other": {
  "name": "Bas Russell",
  "id": "6224364"
 }
}
---------------
My Friends
---------------
{
 "paging": {
  "next": "https://graph.facebook.com/644382747/friends?...",
 },
 "data": [
  {
   "name": "Bas Russell",
   "id": "6224364"
  },
  ...
  {
   "name": "Jamie Lesnett",
   "id": "100002388496252"
  }
 ]
}
---------------
Social Web
---------------
{
 "paging": {
  "next": "https://graph.facebook.com/search?q=social+web&type=page...",
 },
"data": [
  {
   "category": "Book",
   "name": "Mining the Social Web",
   "id": "146803958708175"
  },
  {
   "category": "Internet/software",
   "name": "Social & Web Marketing",
   "id": "172427156148334"
  },
  {
   "category": "Internet/software",
   "name": "Social Web Alliance",
   "id": "160477007390933"
  },
  ...
  {
   "category": "Local business",
   "name": "Social Web",
```

```
   "category_list": [
    {
     "id": "2500",
     "name": "Local Business"
    }
   ],
   "id": "145218172174013"
  }
 ]
}
```

现在，你已经体验了图谱API管理工具和Python控制台本身以及它们所提供的强大功能。既然我们已经攀越了“围墙花园”，让我们将注意转移到分析其中的数据上去。

2.3.1 分析Facebook页面

虽然开始时Facebook是一种没有社交图谱的纯粹社交网络，难以使商业和其他实体很好的参与，但是它很快适应并利用了市场需求。快速发展了几年，现在商业公司、俱乐部、图书以及其他很多种非个人实体都拥有了包含很大粉丝群的页面。Facebook页面（http://on.fb.me/1a1lCzQ）是商业公司吸引顾客的强大工具，并且Facebook煞费苦心地提供了可以让Facebook页面管理者使用一个小型工具箱理解他们粉丝，这个工具被很恰当地称为“Insights”。

如果你已经是一位Facebook用户了，你很有可能已经“赞”过一个或多个Facebook页面，它们代表着你喜欢的或你认为有趣的事物。在这方面，Facebook页面显著地扩大了社交图谱发展为平台的可能性。通过

Facebook页面、“赞”按钮和社交图谱框架对非个人用户实体的明确支持，为兴趣图谱平台提供了一个强大的资源库并且带来了远大的前景（回头参考的1.2节关于为什么兴趣图谱有如此丰富使用前景的讨论）。

2.3.1.1 分析本书的Facebook页面

由于本书有相应的Facebook页面，该页面正好是搜索“social web”得到的最佳结果。很自然，我们可以将其作为演示的起点，进行有益的分析[1]。下面是关于本书Facebook页面（或者其他任何Facebook页面）的一些值得考虑的问题：

·该页面有多么流行？

·该页面粉丝参与度怎样？

·该页面是否有任何特别积极地参与的粉丝？

·关于该页面最最被经常讨论的话题是什么？

当你在挖掘Facebook页面时，你能从中获得的信息远远超出你能想象到的，并且这些问题能够指导你往正确的方向前进。在整个过程中，我们也将会使用这些问题作为与其他页面比较的基础。

回想一下，我们旅程的起点是对“social web”进行的搜索，它通过下

面的搜索结果项显示了一本名为《Mining the Social Web》（本书的英文名称）的书：

```
   {
   "category": "Book",
   "name": "Mining the Social Web",
   "id": "146803958708175"
  }
```

对于结果中的每一项，我们可以通过一个facebook.GraphAPI实例的get_object方法使用ID作为图查询的基础。如果你没有一个可用的数字字符串ID，可以仅仅使用页面的名称（例如“Mining the Social Web”），当你访问页面时它会出现在浏览器的URL栏中。这个代码仅仅只有一行却会得到如示例2-3所示的结果。

示例2-3：使用图谱API查询“Mining the Social Web”的结果

```
   # Get an instance of Mining the Social Web
# Using the page name also works if you know it.
# e.g. 'MiningTheSocialWeb' or 'CrossFit'
mtsw_id = '146803958708175'
pp(g.get_object(mtsw_id))
```

该查询的样例结果揭示了支撑该对象Facebook页面的数据，如下所示：

```
   {
 "category": "Book",
 "username": "MiningTheSocialWeb",
 "about": "Analyzing Data from Facebook, Twitter, LinkedIn, and Other Social...",
 "talking_about_count": 22,
 "description": "Facebook, Twitter, and LinkedIn generate a tremendous ...",
 "company_overview": "Like It here on Facebook!\n\nFollow @SocialWebMining...",
 "release_date": "January 2011",
 "can_post": true,
```

```
 "cover": {
  "source": "https://sphotos-b.xx.fbcdn.net/...",
  "cover_id": 474206292634605,
  "offset_x": -41,
  "offset_y": 0
 },
 "mission": "Teaches you how to...\n\n* Get a straightforward synopsis of ...",
 "name": "Mining the Social Web",
 "founded": "January 2011",
 "website": "http://amzn.to/d1Ci8A",
 "link": "http://www.facebook.com/MiningTheSocialWeb",
 "likes": 911,
 "were_here_count": 0,
 "general_info": "Analyzing Data from Facebook, Twitter, LinkedIn, ...",
 "id": "146803958708175",
 "is_published": true
}
```


查询响应中得到的有趣分析结果是图书的talking_about_count和like_count。like_count是能非常好的反映页面总体流行度的指标，因此对“该页面有多么流行”这个问题的一个合理回答是：这个页面有911个Facebook粉丝，其中有22个经常参与讨论。考虑到《Mining the Social Web》是一本小众的技术书籍，这个粉丝群看起来是合理的。[2]

然而对于任何的流行度分析来说，开展对比对于理解大环境来说是至关重要的。有很多种方式可以进行对比，但是一组惊人的数据是图书出版商O’Reilly Media（http://on.fb.me/1a1lD6F），它拥有大约34000个“赞”，《Python编程语言》（http://on.fb.me/1a1lD6V）大约有80000个“赞”。因此，《Mining the Social Web》的流行度大约接近出版商全部粉丝群的3%，并且仅仅超过编程语言粉丝群的1%。很明显，尽管本书是小众话题的，但是其流行度还有很大的提升空间。

尽管更佳的对比可能是和《Mining the Social Web》相似的小众书

籍，但是通过查看Facebook页面数据我们很难找到任何合适的对比，因为在编写本书时我们还无法在搜索页面时将其限制为“图书”的类别。例如，你无法将结果集限制为图书搜索页面从而寻找合适的对比物。你不得不搜索页面然后通过类别对结果进行过滤从而仅仅获取图书信息。不过，也有一些选择可以考虑。

一种选择是搜索你知道的O’Reilly出版的其他类似的小众书籍，例如ProgrammingCollective Intelligence，然后看一看结果。查询“Programming Collective Intelligence”的图谱API搜索结果是一个有400赞的社区页面（http://on.fb.me/1a1lEYn）。在其他条件都相同的条件下，一个出版6年的图书仅仅只有《Mining the Social Web》一半的“赞”，而本书作者并没有活跃的对页面进行维护，这是非常有趣的。

另一个考虑的选择是利用Facebook的开放图协议从而进行对比。例如，O’Reilly在线目录包含条目并且为所有O’Reilly的书籍实现了OGP，因此《Mining the Social Web》第2版（http://oreil.ly/1cMLoug）和《Programming Collective Intelligence》都有相应的页面（以及“赞”按钮）。我们可以很容易的向图谱API提交请求，看看哪些数据可用，并且按照如下所示的方式，通过在浏览器中查询URL密切关注这些数据：

对《Mining the Social Web》进行图谱API查询

https://graph.facebook.com/http://shop.oreilly.com/product/0636920030

对《Programming Collective Intelligence》进行图谱API查询

https://graph.facebook.com/http://shop.oreilly.com/product/9780596529

从使用Python编程进行查询的角度来说，URL是我们查询（就像我们之前查询的IMDb中The Rock的URL）的对象，因此在代码中我们可以像示例2-4所示查询这些对象。有一个非常微妙却很重要的区别，请记住虽然《Mining the Social Web》的O'Reilly目录页面和Facebook粉丝页面逻辑上代表同一本书，但是Facebook页面和O'Reilly目录页面对应的节点（以及附带的元数据，例如，"赞"的个数）是完全独立的。只是恰巧它们代表了现实世界中的同一个概念。图2-6显示了在IPython Notebook上使用图谱API的一次探索。

注意：一种完全不同的分析方法是实体解析（或是实体消歧，这取决于你如何设计问题），它是将提到的事物聚合成理想概念的过程。例如，在这种情况下，一个实体解析过程将会观测到开放图中有多个节点，它们指代同一个理想概念《社交网络挖掘》并且创建互相之间的联系，这表明实际上它们是现实世界中相同的实体。实体解析是一个激动人心的研究领域，并且将来它会继续对我们如何使用数据产生深远的影响。

示例2-4：通过URL对开放图对象查询图谱API

```
   # MTSW catalog link
pp(g.get_object('http://shop.oreilly.com/product/0636920030195.do'))
```

```
# PCI catalog link
pp(g.get_object('http://shop.oreilly.com/product/9780596529321.do'))
```

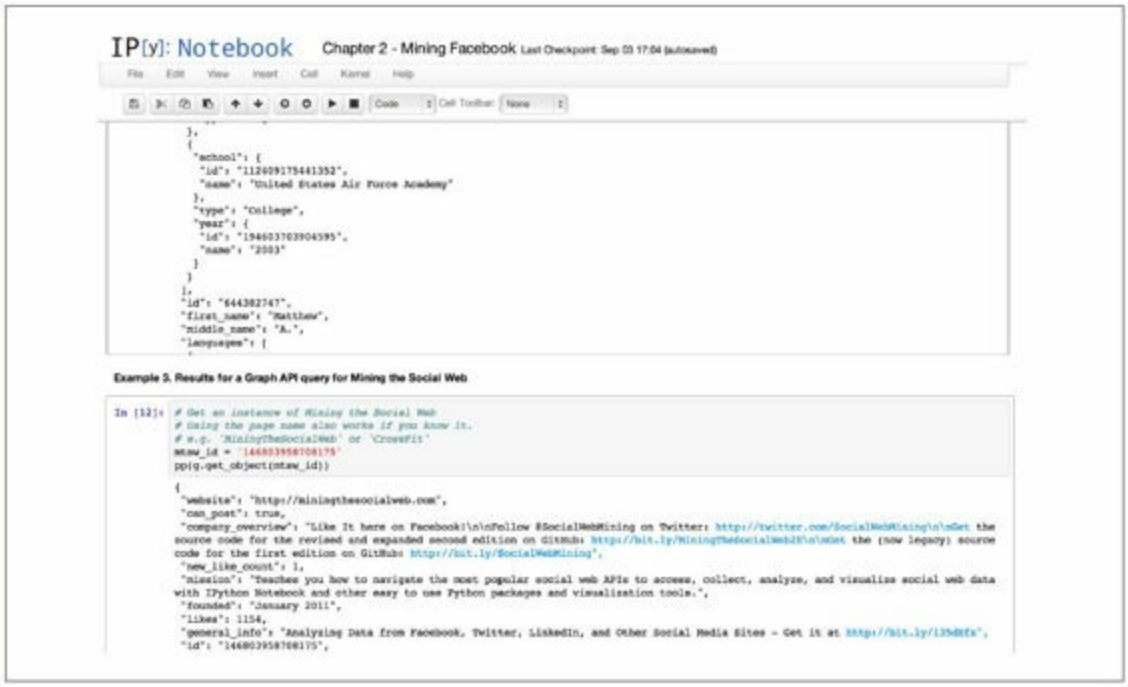


图2-6：在IPython Notebook这样的交互式编程环境的帮助下，探索图谱API就变得轻而易举

尽管大多数情况下你无法在数据挖掘时找到合适的对比从而得到权威的结果，但是仍然可以学到很多。对一个数据集探索足够长的时间会对数据积累很强的直觉，这经常会为你提供一些有益的见解，而该见解是当你第一次遇到问题空间时所需要的。新图谱搜索产品对图谱API作了增强，希望会有利于更加复杂的查询并且在将来降低数据挖掘者的门槛。

2.3.1.2 分析并对比可口可乐和百事可乐的Facebook页面

让我们花些时间将讨论的范围扩大到一些更主流的事物上，看看会得到什么结果，并将其作为分析技术书籍的补充。百事可乐和可口可乐之间无休止的软饮料战争似乎是无害的，反而很可能是一个非常有趣的话题，因此让我们开始根据Facebook决定哪一个是最流行的。正如你所知道的一样，得到的结果仅仅是一组图查询，如示例2-5所示。

示例2-5：比较可口可乐和百事可乐粉丝页的“赞”数

```
   # Find Pepsi and Coke in search results
pp(g.request('search', {'q' : 'pepsi', 'type' : 'page', 'limit' : 5}))
pp(g.request('search', {'q' : 'coke', 'type' : 'page', 'limit' : 5}))
# Use the ids to query for likes
pepsi_id = '56381779049' # Could also use 'PepsiUS'
coke_id = '40796308305'  # Could also use 'CocaCola'
# A quick way to format integers with commas every 3 digits
def int_format(n): return "{:,}".format(n)
print "Pepsi likes:", int_format(g.get_object(pepsi_id)['likes'])
print "Coke likes:", int_format(g.get_object(coke_id)['likes'])
```

结果有些令人惊讶：

```
   Pepsi likes: 9677881
Coke likes: 62735664
```

你是否预想到可口可乐在Facebook上的受欢迎程度几乎是百事可乐的7倍？作为一个可能的调查来源你可能会参考股市信息，然后看看“赞”的数量是否与总市值相关，它可能会反映公司的总体规模。然而，如果你要查找这个信息，结果可能会让你惊讶：在编写本书时（大约是2013年3月），可口可乐（NYSE：KO）的市值大约是178B，而百事可乐（NYSE：PEP）的市值是121B。尽管从金融角度对公司进行分

析是非常复杂的，但是这两个公司总市值大约相差30%，这与在Facebook上受欢迎程度700%的差别相去甚远。我们似乎有理由认为每个公司可能都有相似的手段，并且销售了数量相当的产品。

花些时间进一步深入发掘并试图确定产生这种差异的原因是很有意义的。为了处理这样的问题，请记住，尽管Facebook数据本身可能会有这样的指标，然而这一问题涉及的范围是很大的，除了你在Facebook数据中找到的指标之外还有大量依赖的因素。例如，你是否能在数据中找到特定的指标表明可口可乐推出了大规模广告活动或任何百事可乐没有做过的却能够吸引客户的特别事情？

对这个现象的进一步挖掘留作一个练习。但是，这里给你一个提示：对网络进行初步搜索会在一些著名网站找到一些文章，例如Forbes中名为“Coca-Cola and Procter and Gamble Lead the Way into the New Advertising Era of SocialTV…A Money Machine”和“Coca-Cola Leveraging Social to Drive Leadership in Social Media Marketing，”（http://onforb.es/1a1lIY9）的文章，这些表明可口可乐积极地开展了营销活动，并且大量的使用了社交媒体。

实际上，分析Facebook页面有无限的可能性。尽管我们也可以查询更加具体的过滤器（例如页面的共享链接）进行分析，但是在执行过频率分析之后一个很大的技术升级是从网页的订阅（feed）中检查人类语言数据。我们无法在这么短的一章里解决所有问题，但是一旦你在第

4、5章阅读了一些处理人类语言数据的技术，你就能重新考虑这些问题，利用这些技术更好的理解为什么可口可乐的社交媒体团队吸引了其Facebook粉丝，并且从他们的交流中获得深入的见解。此外，你可以花些时间浏览可口可乐和百事可乐的页面获得直观的感受。毕竟，尽可能在高层次解释数据是编写程序分析的先决条件。

如果你想使用第4章介绍的基本频率分析技术将网页的内容作为一个词袋（bag of words）（http://bit.ly/1a1lHDF）并且从中查看人类语言数据，那么示例2-6为你提供了进行数据收集的切入点。换句话说，你可以仅仅通过使用空格近似单词的边界将文本划分成单词然后将单词作为Counter的输入计算较为频繁的词项，并且将其作为基准。

示例2-6：查询一个页面的“订阅”和“链接”联系

```
   pp(g.get_connections(pepsi_id, 'feed'))
pp(g.get_connections(pepsi_id, 'links'))
pp(g.get_connections(coke_id, 'feed'))
pp(g.get_connections(coke_id, 'links'))
```

如果你选择查看页面中的人类语言数据，这里有些问题供你思考：

·你能通过评论数量和点“赞”数量确定订阅中的哪一个帖子是最受欢迎的吗？

·是否有某一特定类型的帖子比其他的更受欢迎？例如，带有链接的帖子是不是比带有照片的帖子更受欢迎？

·你能找出哪些特征可以让一个帖子广泛传播而不是只有几个“赞”呢？

注意：示例2-6显示了如何查询页面的订阅和链接。订阅、帖子（post）、状态（status）之间的区别开始会让人有点困惑。简而言之，订阅包含了所有用户可能在自己涂鸦墙上看到的东西，帖子包括几乎所有用户创建的并且发布到自己或好友涂鸦墙上的内容，状态仅仅包括用户发布在自己涂鸦墙上的状态更新。详细的信息请参考用户的图谱API文档（http://bit.ly/1a1lHU1）。

2.3.2 查看好友关系

现在让我们使用图谱API的知识在你自己的社交网络里查看好友关系。这里的问题会让你获得一些创意：

·你的社交网络中是否有任何特别明显的话题或特殊兴趣？

·你的社交网络是否包含许多相互的好友关系甚至是更大的互连“团”（http://bit.ly/1a1ljlj）？

·你的社交网络中的人联系是否紧密？

·你是否有一些朋友特别坦率或热衷于任何你也有兴趣了解的事

情？

本节的余下内容会介绍一些涉及分析“赞”以及分析和可视化相互好友关系的练习。尽管这一节我们是对你的社交网络进行分析，但是请记住这里的技巧可以泛化到任何的用户账号并且也能通过你创建的Facebook应用程序实现。

2.3.2.1 分析你好友“赞”的事物

让我们着手研究关于你的社交网络中是否有任何话题或特殊的兴趣，并且从这里开始探索。回答该问题的一个逻辑起点是收集你每个朋友的“赞”然后确定是否有一些出现频率特别高的。示例2-7演示了如何构建一个你的社交网络中“赞”的频率分布，它将作为深入分析的基础。请记住，如果你的任何朋友在隐私设置中选择不共享特定类别的个人信息，例如对应用程序是否喜欢等，你将会得到空的结果而不是任何显式的错误信息。

示例2-7：查询你所有朋友的“赞”

```
   # First, let's query for all of the likes in your social
# network and store them in a slightly more convenient
# data structure as a dictionary keyed on each friend's
# name. We'll use a dictionary comprehension to iterate
# over the friends and build up the likes in an intuitive
# way, although the new "field expansion" feature could
# technically do the job in one fell swoop as follows:
#
# g.get_object('me', fields='id,name,friends.fields(id,name,likes)')
#
# See Appendix C for more information on Python tips such as
# dictionary comprehensions
friends = g.get_connections("me", "friends")['data']
```

```
likes = { friend['name'] : g.get_connections(friend['id'], "likes")['data']
          for friend in friends }
print likes
```

注意：减少预期数据的范围往往会提高响应速度。如果你有很多Facebook好友，那么之前的查询可能要执行一段时间。在对数据进行初步探索时，考虑尝试使用字段扩展创建一个查询或使用类似friends[：100]的列表切片对结果进行约束从而将分析的范围限制在你的100个好友中。

尽管你可能会第一次遇到词典解析，但是收集好友的“赞”以及创建一个很好的数据结构并不是什么的特别棘手的问题。和列表解析一样，词典解析会对列表的每一项进行迭代并收集返回的值（这里是键/值对）。你也可能想尝试使用图谱API的新字段解析特性，然后对所有好友的点“赞”在一个请求中发起一个单独的查询。使用facebook包，你可以这样做：g.get_object（'me'，fields='id，name，friends.fields（id，name，likes）'）。

注意：更多关于词典解析的信息以及其他Python的提示和技巧请参考附录C。

借助likes这个包含了所有好友以及他们“赞”的数据结构，让我们从计算所有好友中最受欢迎的“赞”开始分析。如示例2-8所示，Counter类提供了一种创建频率分布的简单方法，我们将会使用它。我们可以使用prettytable包（如果你还没有安装，请使用pip install prettytable命令）对

结果进行格式化输出，使其更加可读。

示例2-8：计算你好友中最受欢迎的“赞”

```
   # Analyze all likes from friendships for frequency
# pip install prettytable
from prettytable import PrettyTable
from collections import Counter
friends_likes = Counter([like['name']
                         for friend in likes
                           for like in likes[friend]
                               if like.get('name')])
pt = PrettyTable(field_names=['Name', 'Freq'])
pt.align['Name'], pt.align['Freq'] = 'l', 'r'
[ pt.add_row(fl) for fl in friends_likes.most_common(10) ]
print 'Top 10 likes amongst friends'
print pt
```

样例结果如下：

```
   Top 10 likes amongst friends
+-------------------------+------+
| Name                    | Freq |
+-------------------------+------+
| Crossfit Cool Springs   |   14 |
| CrossFit                |   13 |
| The Pittsburgh Steelers |   13 |
| Working Out             |   13 |
| The Bible               |   13 |
| Skiing                  |   12 |
| Star Trek               |   12 |
| Seinfeld                |   12 |
| Jesus                   |   12 |
+-------------------------+------+
```

在社交网络中似乎锻炼/体育是一个常见的主题，而且宗教/基督教也是一个常见的主题。让我们更加深入地进行挖掘，分析社交网络中“赞”的种类来看看是否存在相同的话题。下面示例2-9是之前示例的变形版本：

示例2-9：计算你的好友“赞”中最受欢迎的类别

```
   # Analyze all like categories by frequency
friends_likes_categories = Counter([like['category']
                                    for friend in likes
                                      for like in likes[friend]])
pt = PrettyTable(field_names=['Category', 'Freq'])
pt.align['Category'], pt.align['Freq'] = 'l', 'r'
[ pt.add_row(flc) for flc in friends_likes_categories.most_common(10) ]
print "Top 10 like categories for friends"
print pt
```

查询的样例结果是一个元组，并且与之前的有类似的结构：

```
   Top 10 like categories for friends
+-------------------------+------+
| Category                | Freq |
+-------------------------+------+
| Musician/band           |   62 |
| Book                    |   46 |
| Movie                   |   43 |
| Interest                |   40 |
| Tv show                 |   31 |
| Public figure           |   31 |
| Local business          |   25 |
| Community               |   24 |
| Non-profit organization |   21 |
| Product/service         |   17 |
+-------------------------+------+
```

这里并没有明确的提到体育或宗教，但是有趣的是“音乐家/乐队”或“书籍”这些“未预料到的”类别的频率更高。这可能仅仅是因为社交网络中有很多高度中立、不重叠的爱好。

计算每个好友有多少个“赞”有助于进一步了解现状并且它本身也是引人入胜的。例如，是否有大多数好友有相似数量的“赞”，或者“赞”的数量是否是非常不平均的？对背后的分布有额外的见解有助于在数据聚合时感知到某些要发生的事情。在示例2-10中，我们将计算频率分布，

它显示了每个好友的“赞”数量从而知道之前示例中的类别是如何偏斜的。

注意：示例2-10介绍了operator.itemgetter函数，它通常和sorted函数组合，使用基于元组中某个字段对元组（像对字典实例调用items（）返回的结果一样）组成的列表进行排序。例如，将key=itemgetter（1）传递给sorted函数将会返回基于元组第二项字段进行排序的有序列表。详细内容见附录C。

示例2-10：计算每个好友的“赞”数量，并根据频率进行排序

```
   # Build a frequency distribution of number of likes by
# friend with a dictionary comprehension and sort it in
# descending order
from operator import itemgetter
num_likes_by_friend = { friend : len(likes[friend])
                        for friend in likes }
pt = PrettyTable(field_names=['Friend', 'Num Likes'])
pt.align['Friend'], pt.align['Num Likes'] = 'l', 'r'
[ pt.add_row(nlbf)
  for nlbf in sorted(num_likes_by_friend.items(),
                     key=itemgetter(1),
                     reverse=True) ]
print "Number of likes per friend"
print pt
```

样例结果的格式与包含好友和频率值的元组相似。其中一些结果如下：

```
   Number of likes per friend
+--------------------+-----------+
| Friend             | Num Likes |
+--------------------+-----------+
| Joshua             |       187 |
| Derek              |       146 |
| Heather            |        84 |
| Rick               |        69 |
| Patrick            |        42 |
```

```
| Bryan              |        38 |
| Ray                |        17 |
| Jamie              |        14 |
| ...                |       ... |
| Bas                |         0 |
+--------------------+-----------+
```

花时间理解这些数据会让你有更深的见解。希望你能开始对发生的事情有了更全面的了解。我们现在知道数据中“赞”的分布在很少的几个朋友中发生了明显的偏斜，并且任何一个好友的结果会对总的结果产生巨大的影响并且会打破每个“赞”类别的频率分布。此时，我们可以有很多种选择。其中一种可能是开始比较更小的好友样本寻找某些相似性或深入分析“点赞”。例如，Joshua是否占据了喜欢的电视节目的90%？绝大多数喜欢的音乐是否被Derek占据了？此时，这些问题的答案都在你的掌握之中。

然而，我们要问另外一个问题：哪个朋友更像社交网络中的自我[3]？为了在两者之间进行有效的相似性对比，我们需要一个相似性函数。最简单的概率很可能是最好的基准，因此我们使用“分享的赞的个数”计算自己和好友关系之间的相似度。为了计算两者的相似度，我们所需要的只是把自己的“赞”以及set对象的集合求交操作，这使得我们能够比较两个列表的元素以及计算两者相交的项。示例2-11说明了如何计算网络中自己和好友关系中重叠的“赞”，将其作为在网络中寻找最相似朋友的第一步。

示例2-11：找出社交网络中自己和好友关系共同的“赞”

```
   # Which of your likes are in common with which friends?
my_likes = [ like['name']
             for like in g.get_connections("me", "likes")['data'] ]
pt = PrettyTable(field_names=["Name"])
pt.align = 'l'
[ pt.add_row((ml,)) for ml in my_likes ]
print "My likes"
print pt
# Use the set intersection as represented by the ampersand
# operator to find common likes.
common_likes = list(set(my_likes) & set(friends_likes))
pt = PrettyTable(field_names=["Name"])
pt.align = 'l'
[ pt.add_row((cl,)) for cl in common_likes ]
Print
print "My common likes with friends"
print pt
```

这是社交网络中共有的重叠“赞”结果的简短输出：

```
   My likes
+------------------------------+
| Name                         |
+------------------------------+
| Snatch (weightlifting)       |
| First Blood                  |
| Robinson Crusoe              |
| The Godfather                |
| The Godfather                |
|         ...                  |
| The Art of Manliness         |
| USA Triathlon                |
| CrossFit                     |
| Mining the Social Web        |
+------------------------------+
My common likes with friends
+------------------------------+
| Name                         |
+------------------------------+
| www.SEALFIT.com              |
| Rich Froning Jr. Fan Site    |
| CrossFit                     |
| The Great Courses            |
| The Art of Manliness         |
| Dan Carlin - Hardcore History |
| Mining the Social Web        |
| Crossfit Cool Springs        |
+------------------------------+
```

兜了一圈之后，我们也许不会对体育/锻炼这一常见主题和一些与基督教[4]相关的话题的再一次出现（但是这一次有详细的信息）感到惊

讶。还有更多吸引人的问题要问（并且回答）。让我们通过完成这个查询的第二部分来结束本节的内容，这就是在社交网络中寻找拥有共同爱好的好友。示例2-12显示了如何使双重列表结构对好友关系进行迭代并处理结果从而实现该目标。这也提醒我们，可以充分利用1.4.5节中介绍的matplotlib的画图功能。

注意：如果你正在使用虚拟机，你应该对IPython Notebook进行配置从而能够立刻使用画图功能。如果你在自己的本地环境上运行，请确保在启动IPython Notebook时使用了如下命令启用PyLab（http://bit.ly/1a1l6BN）：ipython notebook pylab=inline。

示例2-12：计算社交网络中与自己最相似的好友

```
   # Which of your friends like things that you like?
similar_friends = [ (friend, friend_like['name'])
                     for friend, friend_likes in likes.items()
                       for friend_like in friend_likes
                         if friend_like.get('name') in common_likes ]
# Filter out any possible duplicates that could occur
ranked_friends = Counter([ friend for (friend, like) in list(set(similar_friends)) ])
pt = PrettyTable(field_names=["Friend", "Common Likes"])
pt.align["Friend"], pt.align["Common Likes"] = 'l', 'r'
[ pt.add_row(rf)
  for rf in sorted(ranked_friends.items(),
                   key=itemgetter(1),
                   reverse=True) ]
print "My similar friends (ranked)"
print pt
# Also keep in mind that you have the full range of plotting
# capabilities available to you. A quick histogram that shows
# how many friends.
plt.hist(ranked_friends.values())
plt.xlabel('Bins (number of friends with shared likes)')
plt.ylabel('Number of shared likes in each bin')
# Keep in mind that you can customize the binning
# as desired. See http://matplotlib.org/api/pyplot_api.html
# For example...
plt.figure() # Display the previous plot
plt.hist(ranked_friends.values(),
         bins=arange(1,max(ranked_friends.values()),1))
```

```
plt.xlabel('Bins (number of friends with shared likes)')
plt.ylabel('Number of shared likes in each bin')
plt.figure() # Display the working plot
```

现在，你应该对这些处理过程很熟悉了。我们已经简单的对构建的变量进行迭代，创建了形式为（friend，friend’s like）的扩展元组列表，然后用它计算频率分布以确定哪些好友拥有最共同的“赞”。该查询的样例结果如下表所示，图2-7以直方图的形式显示了相同的结果：

```
   My Similar friends (ranked)
+----------+--------------+
| Friend   | Common Likes |
+----------+--------------+
| Derek    |            7 |
| Jamie    |            4 |
| Joshua   |            3 |
| Heather  |            3 |
| ...      |          ... |
| Patrick  |            1 |
+----------+--------------+
```

正如你可能想到的一样，只要有了你Facebook好友的一小条数据，我们就可以研究大量的问题。我们已经接触到了冰山一角，但还是希望这些练习能够帮助你进行更深入的探索。为了说明一种可能性，让我们花些时间在结束本章的内容之前以不同的思考方式，查看对一些Facebook好友数据进行有效可视化的方法。

2.3.2.2 使用有向图分析相互好友关系

Twitter本质上是一个开放的网络，你可以在较长的时间段内抓取“好友关系”并且为任何给定的起点构建一个巨大的图。与Twitter不同，Facebook数据更加丰富并且充斥着与个人相关的身份信息和敏感属

性，因此隐私和访问控制使其更加封闭。尽管你可以使用图谱API访问认证用户或其好友的数据，然而你无法为任意用户越权访问数据，除非它被显示为公开可用的。一个特别有趣的图谱API操作是获取你的社交网络中（或认证用户的社交网络）存在的相互好友关系（可以通过mutualfriends（http://bit.ly/1a1lK29）API获得并且在文档中被记录为用户对象（http://bit.ly/1a1lHU1）的一部分）。（换句话说你的哪些好友相互之间也是好友？）从图分析的角度，从一幅自我图出发分析相互好友关系可以被很自然的看成一个团（clique）（http://bit.ly/1a1lKiG）检测问题。

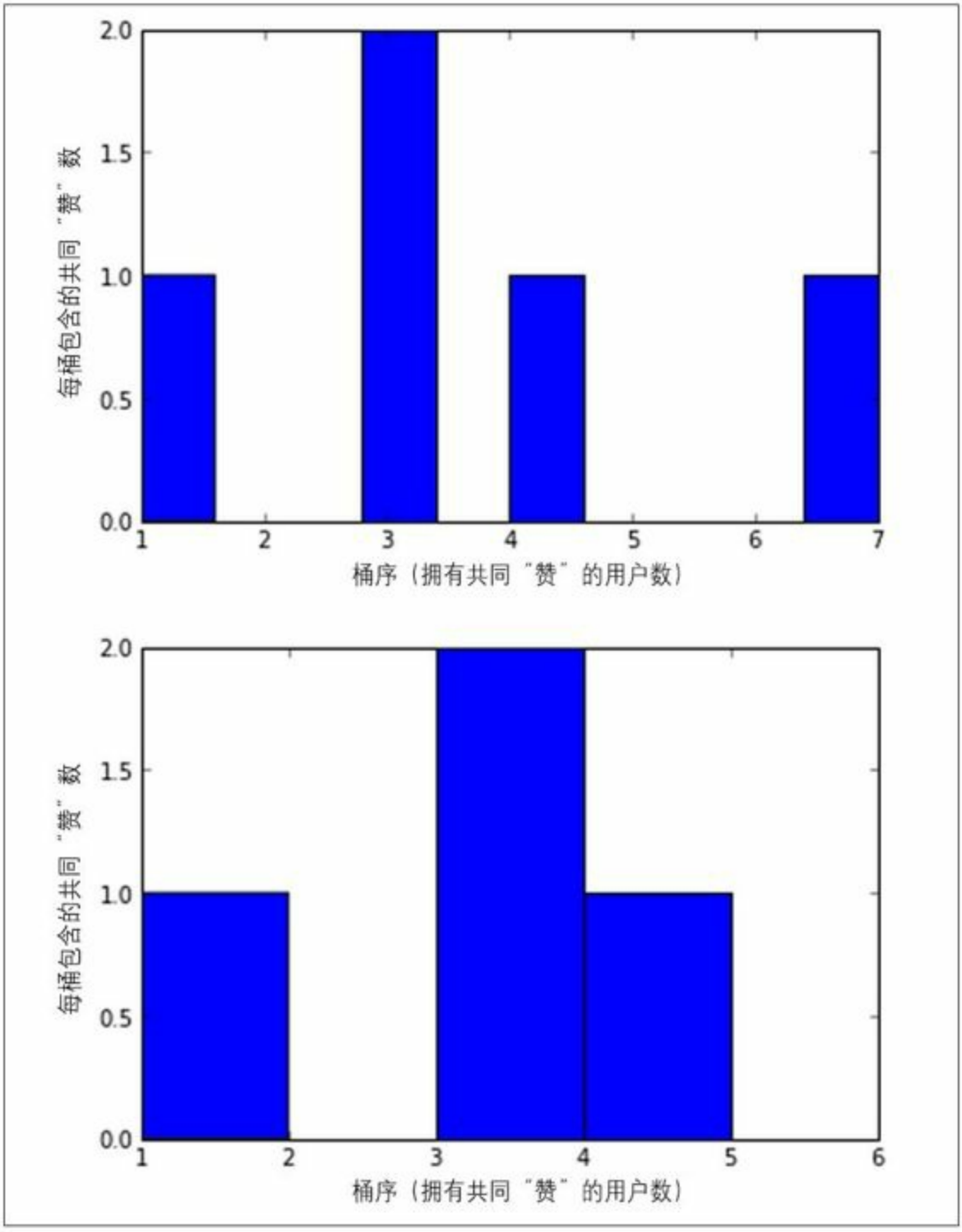


图2-7：显示示例2-12中数据的直方图

例如，如果Abe和Bob、Carol以及Dale三人是好友，同时Bob和

Carol也是好友，那么图中存在的最大的团是由Abe、Bob和Carol构成。如果Abe、Bob、Carol和Dale相互之间是好友，那么该图就是全连接的，最大团的大小为4。向图中添加节点可能会创建额外的团，但是不一定会影响图中最大团的大小。在社交网络的环境中，最大团是非常有趣的，因为它表明图中最大的共同好友的集合。给定两个社交网络，将其最大好友团的大小进行对比可能会为分析群组动态的不同方面提供很好的基础，例如团队合作、信任、生产力。图2-8说明了一幅简单的图，并将最大团突出显示。我们说这幅图的有大小为4的最大团。

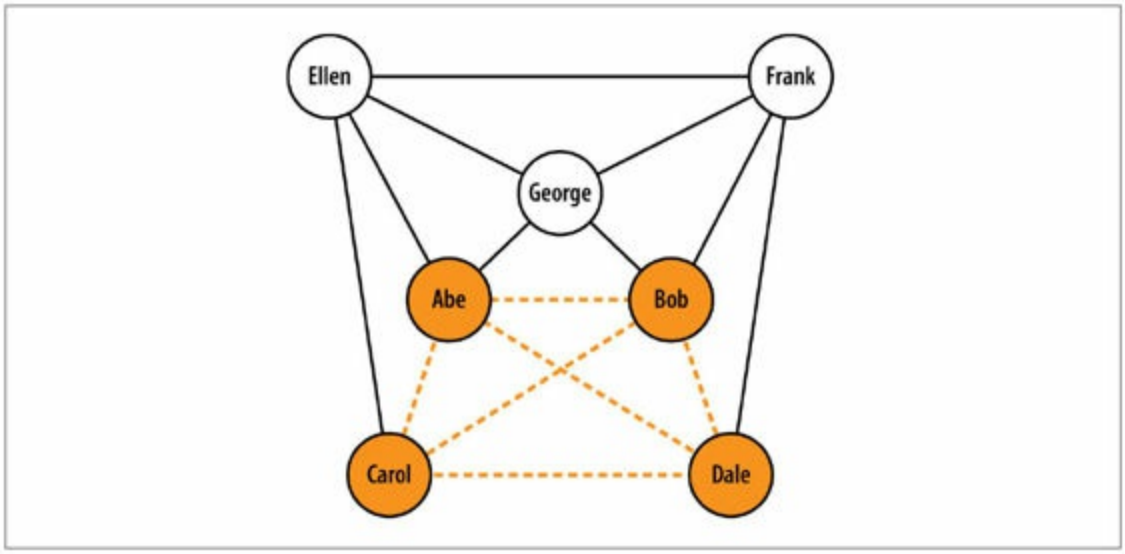


图2-8：包含大小为4的最大团的示例图

注意：从技术上说，极大团和最大团有细微的区别。最大团是图中最大的一个团（或者是大小相等的几个团）。而极大团是那些不是任何其他团子图的团。例如，图2-8显示了大小为4的最大团，但是其中也有一些大小为3的极大团。

寻找团是一个NP完全问题（意味着指数级运行时间（http://bit.ly/1a1lKz9）），但是有一个叫做NetworkX的神奇Python包（发音为“networks”或“network x”）提供了大量的图分析功能，包括一个find_cliques（http://1.usa.gov/1a1lKzk）方法，该方法提供了这个难题的完整实现。要注意，如果图的大小超过合理的范围，那么运行该方法会花费很长时间（上面提到的指数级运行时间）。示例2-13和2-14显示了如何使用Facebook数据构建一个相互好友关系图并且使用NetworkX分析图中的团。你可以在终端输入pip install networkx安装NetworkX。

示例2-13：创建一个相互好友关系图

```
   import networkx as nx # pip install networkx
import requests # pip install requests
friends = [ (friend['id'], friend['name'],)
                for friend in g.get_connections('me', 'friends')['data'] ]
url = 'https://graph.facebook.com/me/mutualfriends/%s?access_token=%s'
mutual_friends = {}
# This loop spawns a separate request for each iteration, so
# it may take a while. Optimization with a thread pool or similar
# technique would be possible.
for friend_id, friend_name in friends:
    r = requests.get(url % (friend_id, ACCESS_TOKEN,) )
    response_data = json.loads(r.content)['data']
    mutual_friends[friend_name] = [ data['name']
                                    for data in response_data ]
nxg = nx.Graph()
[ nxg.add_edge('me', mf) for mf in mutual_friends ]
[ nxg.add_edge(f1, f2)
  for f1 in mutual_friends
      for f2 in mutual_friends[f1] ]
# Explore what's possible to do with the graph by
# typing nxg.<tab> or executing a new cell with
# the following value in it to see some pydoc on nxg
print nxg
```

示例2-14：在相互好友关系图中寻找和分析“团”

```
   # Finding cliques is a hard problem, so this could
```

```
# take a while for large graphs.
# See http://en.wikipedia.org/wiki/NP-complete and
# http://en.wikipedia.org/wiki/Clique_problem.
cliques = [c for c in nx.find_cliques(nxg)]
num_cliques = len(cliques)
clique_sizes = [len(c) for c in cliques]
max_clique_size = max(clique_sizes)
avg_clique_size = sum(clique_sizes) / num_cliques
max_cliques = [c for c in cliques if len(c) == max_clique_size]
num_max_cliques = len(max_cliques)
max_clique_sets = [set(c) for c in max_cliques]
friends_in_all_max_cliques = list(reduce(lambda x, y: x.intersection(y),
                                  max_clique_sets))
print 'Num cliques:', num_cliques
print 'Avg clique size:', avg_clique_size
print 'Max clique size:', max_clique_size
print 'Num max cliques:', num_max_cliques
print
print 'Friends in all max cliques:'
print json.dumps(friends_in_all_max_cliques, indent=1)
print
print 'Max cliques:'
print json.dumps(max_cliques, indent=1)
```

示例2-14的样例输出如下，结果表明图中有4个大小为4的团，其中自我（“me”）和另一个人是社交网络中共有的。尽管在所有团中都出现的另一个人不一定是网络中连接频率第二高的，但是根据这些共同的关系，这个人很可能是最有影响力的人物之一。

```
   Num cliques: 6
Avg clique size: 3
Max clique size: 4
Num max cliques: 4
Friends in all max cliques:
[
 "me",
 "Bas"
]
Max cliques:
[
 [
  "me",
  "Bas",
  "Joshua",
  "Heather"
 ],
 [
  "me",
  "Bas",
  "Ray",
  "Patrick"
```

```
 ],
 [
  "me",
  "Bas",
  "Ray",
  "Rick"
 ],
 [
  "me",
  "Bas",
  "Jamie",
  "Heather"
 ]
]
```

示例2-14可以被修改成很多形式，很显然除了检测团之外我们还可以做很多事。将团中涉及的人的位置画在地图上，然后看看紧密相连的人际网络与地理位置是否有任何相关性，并且分析他们个人档案数据中的信息以及帖子的内容将会是很好的基础练习。在下一节，我们将会学习如何以直观的图形化方式对相互好友关系进行简洁而有效的可视化。

2.3.2.3 可视化相互好友关系的有向图

D3.js（http://bit.ly/1a1kGvo）是一个最新的JavaScript工具，它能够通过一系列数据驱动的变换以一种直观的方式操作对象，在浏览器中呈现漂亮的可视化结果。如果你没有尝试过D3，应该花些时间浏览示例库（http://bit.ly/1a1lMal）感受一下它能够实现哪些效果。它会给你留下深刻印象的。

关于如何使用D3的教程超出了本书的范畴，网上有很多关于如何使用这些激动人心的可视化工具的教程（http://bit.ly/1a1lMHg）和讨论。在结束这一章内容之前，我们将在这一节呈现上一节引入的相互好

友关系图的交互式可视化结果。从抽象的角度看，一个图只是一种数学构造并且没有视觉的描述，但是一些布局算法（layout algorithm）可以用来在二维空间渲染图像，这样就能够显示得相当漂亮（尽管你可能需要经常调整一些布局参数）。

NetworkX能够按照D3的要求输出可供其直接使用的文件，由于IPython Notebook能够通过预置files到路径使用内联的框架呈现本地的内容，因此我们只需少量的工作就能对图进行可视化。示例2-15显示了如何将图进行保存从而进行显示，示例2-16使用IPython Notebook提供了一个网页来显示一个如图2-9所示的交互式图。嵌入了必要样式和脚本的HTML以及本章的IPython Notebook被放在资源viz子文件夹里。

注意：你可以访问文件进行阅读或基于相对或绝对路径用IPython Notebook编写，然而要将文件作为网页，你需要预先把files设置到路径中。

示例2-15：将一幅NetworkX图保存为文件以供D3使用

```
   from networkx.readwrite import json_graph
nld = json_graph.node_link_data(nxg)
json.dump(nld, open('resources/ch02-facebook/viz/force.json','w'))
```

示例2-16：使用D3将一个相互好友关系图可视化

```
   from IPython.display import IFrame
from IPython.core.display import display
# IPython Notebook can serve files and display them into
# inline frames. Prepend the path with the 'files' prefix.
```

```
viz_file = 'files/resources/ch02-facebook/viz/force.html'
display(IFrame(viz_file, '100%', '600px'))
```

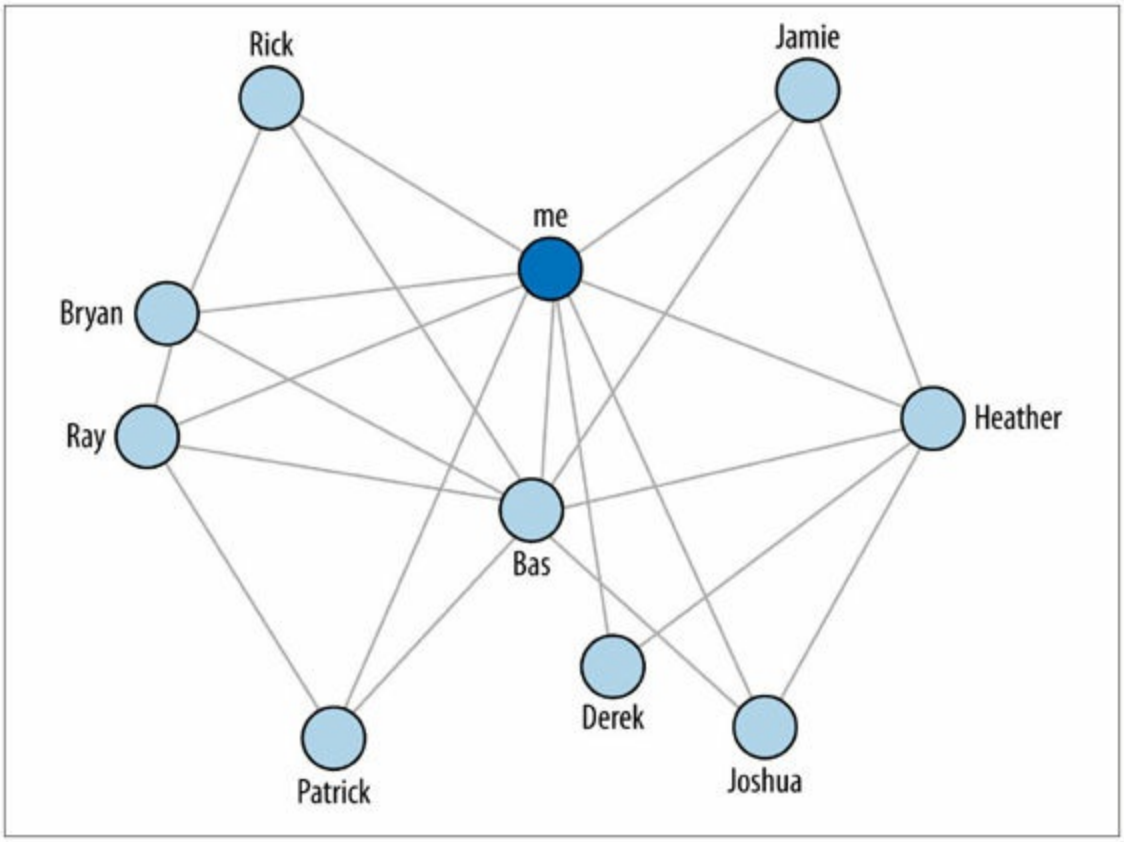


图2-9：一幅Facebook社交网络中相互好友关系的图——你可以在IPython Notebook中跟随示例代码生成一幅类似的图

[1] 提醒，本节查询的返回结果是基于本书第1版的数据，因为第2版还处于写作过程中，只有第1版是可用的。另外，对你的特定查询，得到的结果可能会存在一些出入。

[2] 这一结论基于2013年3月的数据。

[3] 记住，社交网络的“自我”是它的逻辑中心或基础。此时，网络的“自我”是本书作者，这就是我们正查看的社交网络的拥有者。

[4] Rich Froning Jr.是位著名的CrossFit运动员（同时也是坦率的基督

徒）。

2.4 本章小结

本章的目标是教会你如下几件事情：如何使用图谱API、开放图协议如何在任意两个网页和Facebook的社交图之间建立连接，以及如何编程查询社交图从而对Facebook页面和你自己的社交网络获得深入的理解。如果你练习了本章的全部示例，应该很容易就能探测社交图从而回答一些很有价值的问题。注意，当你探索像Facebook社交图这样庞大且有趣的数据集时，你仅仅需要一个好的切入点。当你调查一个初始查询的谜底时，你很有可能会遵循一个自然的探索过程，它将持续提炼你对数据的理解，使你离寻找的答案更进一步。

在Facebook上进行数据挖掘的可能性是巨大的，但是要尊重隐私，并且尽可能的遵守Facebook的服务条款（http://on.fb.me/1a1lMXM）。和Twitter以及其他这些本质上更加开放的数据源中的数据不同，Facebook数据是非常敏感的，特别是当你分析自己的社交网络时。希望这一章清楚地表明了我们可以使用社交数据实现许多激动人心的可能性，并且Facebook隐藏了大量的价值。

注意：本章和其他章节中列出的源代码可以在GitHub（http://bit.ly/1a1kNqy）中获得，它们是非常方便的IPython Notebook格式，我们非常推荐你利用Web浏览器试验这些程序。

2.5 推荐练习

·分析Facebook里一些你感兴趣事物的粉丝页面上的数据并且尝试分析评论流中的自然语言从而获得一些见解。被讨论的最常见的话题是哪些？你能否判断粉丝对一些事物是特别高兴或沮丧吗？

·选择两个相同性质的不同粉丝页面然后进行对比。例如，你能从Chipotle Mexican Grill和Taco Bell识别出哪些相似和不同吗？你能找到任何让人意想不到的事情吗？

·分析你自己的好友关系，试图确定自己的网络中是否有一些自然地聚集点或者共同的爱好。将你的网络绑在一起的共同黏合剂是什么？

·Facebook图谱API可用的对象（http://bit.ly/1a1lNeg）有很多。你是否能够检查类似照片或签到等对象从而发现关于你的网络中某些人的信息。例如，谁发布了最多的图片以及你是否能够根据评论流确定它们是关于什么的吗？你的好友们最经常签到的地方是哪里？

·使用直方图（在1.4.5节引入）深入分析你好友的“赞”数据。

·使用图谱API收集其他类型的数据并寻找合适的D3可视化工具将其呈现。例如，你能否在地图上画出好友居住或者长大的地方吗？哪些好友仍然在家乡居住？

·获得一些Twitter数据，构建一个图，然后使用本章介绍的技术对其进行分析和可视化。

·尝试使用一些不同的相似性标准计算和你最相似的好友。Jaccard指数（http://bit.ly/1a1lNuR）是一个很好的切入点。4.4.4节的信息对此有很大帮助。

2.6 在线资源

下面这个链接列表对复习是很有帮助的：

·“词袋”模型（http://bit.ly/1a1lHDF）

·D3.js示例库（http://bit.ly/1a1lMal）

·D3.js教程（http://bit.ly/1a1lMHg）

·Facebook开发者（http://bit.ly/1a1lm3Q）

·Facebook开发者分页文档（http://bit.ly/1a1ltMP）

·Facebook平台政策（http://bit.ly/1a1lm3C）

·FQL字段扩展和嵌套（http://bit.ly/1a1lsIE）

·FQL参考手册（http://bit.ly/1a1lmRd）

·图谱API入门指南（http://bit.ly/1a1lobU）

·图谱API管理工具（http://bit.ly/1a1lnVv）

·图谱API参考手册（http://bit.ly/1a1lvEr）

·图团（http://bit.ly/1a1lJLJ）

·HTMLmeta元素（http://bit.ly/1a1lBMa）

·NetworkX团算法（http://bit.ly/1a1lWyo）

·NP完全问题（http://bit.ly/1a1lLDd）

·OAuth（http://bit.ly/1a1kZWN）

·开放图协议（http://bit.ly/1a1lu3m）

·PyLab（http://bit.ly/1a1l6BN）

·Python Requests库（http://bit.ly/1a1lrEt）

·RDFa（http://bit.ly/1a1lujR）

第3章　挖掘LinkedIn：分组职位、聚类同行等

本章的技术和讨论用于挖掘存储在LinkedIn中数据的价值，这是一个专注于职业和商业关系的社交网站。尽管LinkedIn初看起来可能和其他的社交网站比较像，但其API提供的数据与它们有本质上的不同。如果你把Twitter比作城镇广场那样繁忙的公共场所，把Facebook比作一个坐满了朋友和亲人的大房间，一起聊着晚间闲聊类的话题，那么你可能把LinkedIn比作一个要求半正式着装的私人聚会，这里每个人都表现其最好的状态，并尝试表达他们可以给职业市场带来的价值和技能专长。

考虑到隐藏在LinkedIn中数据的敏感特性，它的API有其自己的微妙之处，使其不同于我们在本书中看到过的许多其他API。人们加入LinkedIn主要是对商业机遇感兴趣，它提供的这些机遇与随意的社会交往不同，它需要一直提供关于商业关系、工作记录等敏感细节。例如，虽然通常你可以访问你的联系人的教育背景和之前工作职位的所有细节，但是你不能像Facebook那样，确定任意两个人是否相连。这样的API方法是故意未提供的。它的API无法把它打造成像Facebook或Twitter那样的社会图谱，因此需要你对可利用的数据提各种问题。

本章剩下的内容会教会你用LinkedIn API来读取数据，同时也会介绍一些基本的数据挖掘技术，以帮助你根据相似度来聚类不同的行业，这样就能回答下面的几种问题：

·你的哪些“联系人”（connection）最符合你的标准（例如职位）？

·你的哪些“联系人”曾在你想去的公司工作过？

·你的大多数“联系人”住在什么地方？

所有这些情形，使用聚类技术的分析模式在本质上是相同的：在同行的档案数据中提取一些特征，定义一个相似度来比较每个档案中的特征，并用聚类技术将足够相似的同行分在同一组。这种方法用来聚类LinkedIn数据是可行的，并且你也能对遇到的几乎任何其他数据应用这些技术。

注意：在http://bit.ly/MiningTheSocialWeb2E上可以找到本章（及所有其他章节）最新修订bug的源代码。同时也要利用好本书的虚拟机，如附录A中描述的，来尽可能地享用样例代码。

3.1 概述

本章介绍的内容是机器学习的基础，并且一般情况下会比前两章要高级一些。建议你在阅读本章材料之前，牢固掌握前两章的知识。本章你会学习到：

·LinkedIn的开发平台以及发起API请求。

·三种常用的聚类方法。聚类是一个基础的机器学习问题，它可以应用在几乎任何问题领域。

·数据清洗以及标准化。

·地理编码。它是一种从描述一个地点的文本参考中获得一系列坐标的方法。

·用Google地图和统计地图来可视化地理数据。

3.2 探索LinkedIn API

你需要一个LinkedIn账号，并且要求在你的职业人脉网络中有一些同行，这样你跟着本章的样例进行操作会很有意义。如果你没有LinkedIn账号，你仍旧可以应用将要学到的基本的聚类技术到其他的领域，但是无法充分参与本章的学习，因为如果你没有自己的LinkedIn数据，就不能顺着本章的例子来进行操作。LinkedIn账号是你职业生活中有价值的投入，如果你还没有账号，那么现在就开始构建你的LinkedIn职业人脉网络吧。

尽管本章绝大多数的分析都是针对LinkedIn联系人的逗号分隔值（Comma-Separated Value，CSV）文件进行操作的（该文件可以下载到），本节为保证与其他章的一致性，也先提供LinkedIn API的概述。如果你对LinkedIn API不感兴趣，你可以直接跳转到3.3.3节的分析部分，在之后需要了解API使用细节时再回头来阅读。

3.2.1 发起LinkedIn API请求

和其他社交网络的情形相同（如我们在前面章节讨论的Twitter和Facebook），获得LinkedIn API使用权的第一步就是创建一份申请。你可以在https://www.LinkedIn.com/secure/developer上创建一个申请；记下

你所提交申请的API Key、Secret Key、OAuth User Token和OAuth User Secret这些认证信息，你会在编程调用API的时候用到它们。图3-1显示了创建一个申请后的情形。

图3-1：为调用LinkedIn API，

在https://www.LinkedIn.com/secure/developer上创建一个申请，并记录4个OAuth认证信息（图中挡住的部分），在申请的详细页面可以看到它们

有了必需的OAuth认证信息后，使用API读取自己的档案数据的过程很像Twitter的操作，在使用Twitter API的过程中，你需要把这些认证信息提供给程序库，它允许你发起API请求。如果你没有借助本书的虚拟机，你需要在终端输入pip install python-LinkedIn命令来安装。

注意：实现OAuth 2.0的细节可以参考附录B，你需要用它来建立一个申请，这个申请需要使用者授权来读取账户数据。

示例3-1阐述了一个样例脚本，它用你的LinkedIn认证信息来创建一个LinkedInApplication类的实例，该类可以读取你的账户数据。注意该脚本的最后一行检索了你的基本资料信息，包括你的名字和概要。在进一步研究前，你应该花些时间浏览它的REST文档（http://linkd.in/1a1lZuj），以了解哪些LinkedIn API操作对开发者可用，该文档提供了你可以做的事情的概况。尽管我们可以通过包装了HTTP请求的Python包来调用API，但是核心API的文档总是最权威的参考手册，并且大多数好的程序包都模仿它的风格。

注意：如果你需要从你的申请或任何其他OAuth申请中取消账户使用权，你可以在账户设置中（http://linkd.in/1a1lZKN）进行操作。

示例3-1：用LinkedIn OAuth验证信息来获取开发和读取数据的权限

```
   from LinkedIn import linkedin # pip install python-linkedin
# Define CONSUMER_KEY, CONSUMER_SECRET,
# USER_TOKEN, and USER_SECRET from the credentials
# provided in your LinkedIn application
CONSUMER_KEY = ''
CONSUMER_SECRET = ''
USER_TOKEN = ''
USER_SECRET = ''
RETURN_URL = '' # Not required for developer authentication
# Instantiate the developer authentication class
auth = linkedin.LinkedInDeveloperAuthentication(CONSUMER_KEY, CONSUMER_SECRET,
                                  USER_TOKEN, USER_SECRET,
                                  RETURN_URL,
                                  permissions=linkedin.PERMISSIONS.enums.values())
# Pass it in to the app...
app = linkedin.LinkedInApplication(auth)
# Use the app...
```

```
app.get_profile()
```

简而言之，使用LinkedInApplication实例的调用和使用REST API（http://linkd.in/1a1lZuj）调用是一样可行的。并且GitHub上的python LinkedIn文档（http://bit.ly/1a1m2Gk）提供了许多查询来帮助你起步。最值得关注的一对API是联系人API（Connections API）和查询API（Search API）。回忆下本书开始的简要讨论，你不能获得“朋友的朋友”（在LinkedIn语法中为联系人的联系人），但是联系人API会返回一个你联系人的列表，它提供了一个获得档案信息的起点。查询API提供了对LinkedIn上人、公司或工作查询的一种方式。

其他API也是可以使用的，并且值得你花时间来熟悉它们。你职业网络中可用数据的质量是十分高的，因为它可能包括了完整的工作经历、现在公司的详细信息、所处位置的地理信息等。

示例3-2展示了如何使用app（一个LinkedInApplication的实例）对你的联系人检索[1]扩展了的档案信息，并将检索的数据存入一个文件，以避免任何不必要的API请求，这些请求的速率流上限（http://linkd.in/1a1m2WP）限制类似于Twitter API的限制。

警告：当使用LinkedIn API的时候要小心：速率上限限制直到世界标准时间（Coordinated Universal Time，UTC）的午夜才会重置，并且如果你不够小心，一个有bug的循环会打乱你下一个24小时的计划。

示例3-2：检索你的LinkedIn联系人并将他们存储在磁盘中

```
   import json
connections = app.get_connections()
connections_data = 'resources/ch03-linkedin/linkedin_connections.json'
f = open(conections_data, 'w')
f.write(json.dumps(connections, indent=1))
f.close()
# You can reuse the data without using the API later like this...
# connections = json.loads(open(connections_data).read())
```

检查你的联系人数据的第一步，让我们用在前面章节介绍的prettytable包以漂亮排版的表格形式来显示你的联系人和他们的工作地点，如示例3-3所展示的那样。如果你没有借助本书前面配置好的虚拟机，为了运行本章大多数的例子，你需要在终端中输入pip install prettytable；这个包可以生成精心排版的表格输出。

示例3-3：完美输出你的LinkedIn联系人数据

```
   from prettytable import PrettyTable # pip install prettytable
pt = PrettyTable(field_names=['Name', 'Location'])
pt.align = 'l'
[ pt.add_row((c['firstName'] + ' ' + c['lastName'], c['location']['name']))
  for c in connections['values']
      if c.has_key('location')]
print pt
```

样例（匿名）结果展示了你的联系人并根据他们的档案显示他们现在所处的位置。

```
   +---------------------------------+------------------------------------------+
| Name                            | Location                                 |
+---------------------------------+------------------------------------------+
| Laurel A.                       | Greater Boston Area                      |
| Eve A.                          | Greater Chicago Area                     |
| Jim A.                          | Washington D.C. Metro Area               |
| Tom A.                          | San Francisco Bay Area                   |
```

```
|    ...                          |          ...                                |
+---------------------------------+---------------------------------------------+
```

对从联系人API中返回的档案信息的完整扫描表明这些信息是优美而简洁的，但是如果可能，你可以用在“档案字段”（Profile Field）在线文档（http://linkd.in/1a1m3ds）中概括的字段选择器（field selector）来检索额外的细节。例如，示例3-4显示了如何获得一个联系人的工作职位史。

示例3-4：为你的档案和一个联系人的档案展示工作职位史

```
   import json
# See http://developer.linkedin.com/documents/profile-fields#fullprofile
# for details on additional field selectors that can be passed in for
# retrieving additional profile information.
# Display your own positions...
my_positions = app.get_profile(selectors=['positions'])
print json.dumps(my_positions, indent=1)
# Display positions for someone in your network...
# Get an id for a connection. We'll just pick the first one.
connection_id = connections['values'][0]['id']
connection_positions = app.get_profile(member_id=connection_id,
                                       selectors=['positions'])
print json.dumps(connection_positions, indent=1)
```

样本的输出显示每个职位的一些有趣细节，包括公司的名字、行业、简介和雇佣日期：

```
    {
   "positions": {
   "_total": 10,
   "values": [
    {
     "startDate": {
     "year": 2013,
     "month": 2
    },
    "title": "Chief Technology Officer",
    "company": {
     "industry": "Computer Software",
```

```
     "name": "Digital Reasoning Systems"
    },
    "summary": "I lead strategic technology efforts...",
    "isCurrent": true,
    "id": 370675000
   },
   {
    "startDate": {
    "year": 2009,
    "month": 10
    }
    ...
   }
  ]
 }
}
```

正如预期的那样，一些API返回值可能不一定包括你想知道的所有信息，另一些返回值也可能包含比你需要的更多的信息。你应该利用好字段选择器的语法（http://linkd.in/1a1m3tV）来定制你所需要返回的细节，而不是调用多次API并将信息拼接在一起或武断地去除你不需要的信息。示例3-5显示你该如何对公司信息只检索其name、industry和id三个字段，作为回应档案职位查询的一部分。

示例3-5：用字段选择器的语法在API中请求附加的细节

```
   # See http://developer.linkedin.com/documents/understanding-field-selectors
# for more information on the field selector syntax
my_positions = app.get_profile(selectors=['positions:(company:(name,industry,id))'])
print json.dumps(my_positions, indent=1)
```

一旦你熟悉了对你有用的基本API，有了一些便利的文档标签，还多次调用过API来熟悉一些基础技术，你就可以开启你的LinkedIn之旅了。

3.2.2 下载LinkedIn的联系人并保存为CSV文件

尽管API提供了用编程的方式读取一个授权用户能看到的位于http://LinkedIn.com的所有事情，你还可以以CSV文件格式导出你的LinkedIn联系人的通讯录，里面包含了本章所需要的所有工作职位细节。导出的初始步骤是从联系人菜单中选择联系人菜单项来操作你的LinkedIn联系人页，随后从你的LinkedIn账户中选择“导出联系人”的链接。或者你可以直接使用图3-2中显示的输出LinkedIn联系人对话框（http://linkd.in/1a1m4ho）。

在本章的后面内容中，我们将使用作为Python标准库一部分的csv模块来分析输出的数据，因此为了保证以后代码清单的一致性，在可用选项中选择Outlook CSV选项。

[1] 如果你的任何联系人不允许被LinkedIn API读取，他的姓和名会出现“private”，进一步的细节信息也难以获得。

3.3 数据聚类速成

现在你已经对如何调用LinkedIn API有了基本理解，我们下面采用聚类技术，对聚类给予相对更具体的分析。聚类[1]是一个无监督的机器学习技术，它在任何数据挖掘的工具包中都是重要的组成部分。聚类涉及获取一批条目，并根据一些启发式将它们分为更小集合或类，设计这些启发式通常需要比较集合中的条目。

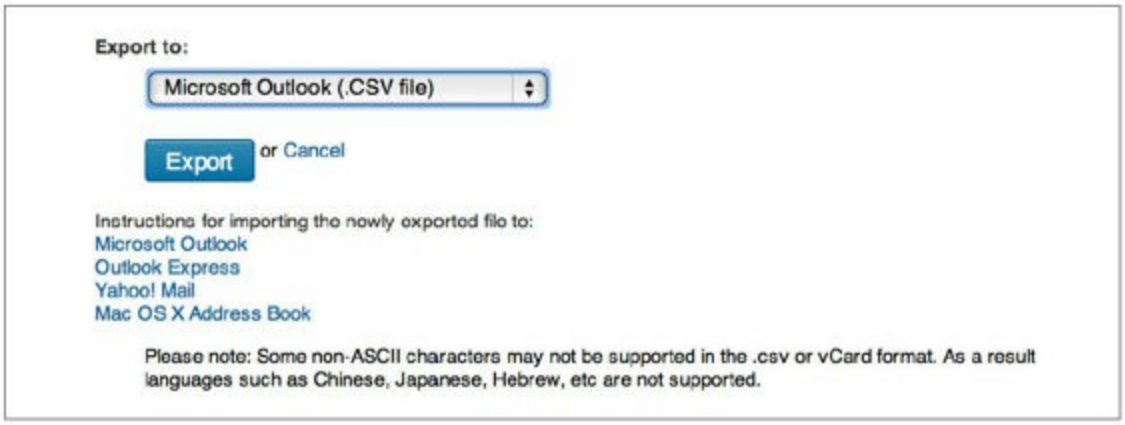


图3-2：LinkedIn的一个几乎不为人知的特性就是你可以在http://www.LinkedIn.com/people/export-settings上方便且便捷地导出你所有的联系人为CSV格式文件

注意：聚类是一个基础的数据挖掘技术，作为介绍的一部分，本章包括了对问题背后的数学原理的一些脚注和讨论。尽管你最后应该尝试去理解这些细节，但是为成功使用聚类技术，你不需要掌握所有的要点，尤其当你第一次看到它们的时候你无需对掌握它们而感到压力。如

果你没有必要的数学基础，你或许需要一点时间来消化这些讨论。

例如，如果你正考虑地理信息的重新划分，你可能会发现将你的LinkedIn联系人聚类成几个地理区域来更好地发现可用的商业机会是很有用的。我们随后会再次谈到这个概念，但是首先我们要花一点时间来简略地讨论一些有关聚类的细节问题。

当实现可解决LinkedIn或其他地方聚类的问题时，你将会重复地遇到聚类分析过程中的至少两个主要问题（参见附注框内的“降维在聚类中的作用”对第三个问题的讨论）：

数据标准化

即使你正用一个很好的API来检索数据，提供给你的数据通常也不是你想要的那个格式，往往需要花一些工夫将数据转为适合分析的形式。例如，LinkedIn会员可以在文本输入框中描述他们的职位名称，这样你就不能总是得到完全标准化的职位名称。一个总经理可能会选择“Chief Technology Officer”这样的名称，然而另外一个总经理可能会选择更含糊的名称“CTO”，并且仍然会有其他各种各样的名称来表达相同的职位。我们将会再次讨论数据标准化的问题，并随后实现一种对LinkedIn数据的特定方面进行处理的模式。

相似度计算

假设你拥有良好标准化的条目，你需要计算它们中任两个的相似度，它们可能是职位名称、公司名字、职业兴趣、地理坐标或是你能在文本中输入变量的其他方面，因此你需要定义一个能粗略估计任意两个值相似度的启发式。在一些情形下，计算相似度的启发式会十分明显，但是另一些情况下会是困难的。例如，比较2个人的工作年头是十分简单的，只需要加法操作，但是用完全自动化的方式来比较像领导能力这样的职业元素会是一个挑战。

降维在聚类中的作用

尽管数据标准化和相似度计算是聚类在抽象层次上会遇到的两个首要的主题，但是一旦你使用的数据规模变得很大的时候，降维就会成为第三个重要主题。为了用相似度度量的方法将集合中的所有条目聚类，你会将每一项与其他项相比较。这样，对于一个有n个成员的数据集，最坏的情况下你的算法中需要n^2次相似度计算，因为你需要将n个成员的每一个和其他n－1个相比较。

计算机科学家称这个问题为n平方问题，通常用O（n^2）来表示这种问题；通俗地讲，你会称其为“n平方的大O”问题。对于很大的n值，O（n^2）问题会很棘手，大多数情况下，很棘手意味着要想解决方案计算完毕，你要等“很长”时间。根据问题的性质和问题的限制，“很长”可能是几分钟，几年或者永世。

数据降维技术的探究超出了我们现在要讨论的范畴，但是知道一点

即可：一个经典的数据降维技术涉及用一个函数将“足够相似”的项放到固定数目的桶内，这样在每个桶内的项彼此可以更充分地相互比较。数据降维通常既是艺术又是科学，并且那些应用降维方法成功获得竞争优势的机构经常把它当做产权信息或商业秘密。

聚类技术是任何数据挖掘者所必备工具的基本部分，因为从国防情报（defense intelligence）到银行部门的欺诈检测（fraud detection），再到景观美化的几乎任何工业的任何一个部门中，都有大量的半标准化的相关数据需要分析。在过去几年，数据科学家职位的增加也确实证实了这点。

通常，一个公司会建立一个数据库来收集某种信息，但不是每个字段都落入预先定义好的有效答案范围中。或者是因为应用的用户界面逻辑设计的不正确（一些字段没有指示出他们已经确定的值）或者因为用户体验很关键，所以允许用户在文本框中输入他们喜欢的任意内容，但结果总是相同的：你最终获得了大量半标准化的数据，或是“脏记录”。虽然对一特定字段可能会有N个不同的字符串表示，但是有些字符串实际上表示相同的意思。冗余的产生可能由多种原因造成，例如错误的拼写、缩写或速记，还有单词大小写方面的区别。

尽管这种情况可能不明显，但这确实是我们在挖掘LinkedIn数据时所面对的典型情况，也就是LinkedIn的会员可以在文本中自由的输入他们的职业信息，这样就会导致一定数量不可避免的差别。例如，如果你

想检测你的职业网络，并试着发现你的联系人大都在哪工作，你就需要考虑公司名字的常见变化。即使是最简单的公司名字也有一些常见的变化，这几乎是肯定要遇到的情况。例如，很明显对大多数人来说“Google”是“Google，Inc.”的缩写形式，但是即使是命名规范中这些简单变化也要在标准化过程中做出明确的说明。在标准化公司名字过程中，最好是首先考虑后缀，如LLC和Inc。

3.3.1 聚类来提高用户体验

即便是针对刚刚得到的工作职位这一简单的结果进行利用，简单的聚类技术也会创造出令人难以置信的用户体验。图3-3通过简单的树控件给出了一种强大的数据展示方式，这个树可以被用作导航栏或分组窗口展示的一部分，用于过滤搜索条件。假设你选择相似度已经得到了有意义的群组，一个简洁的分层对数据进行逻辑归类，每类给出个体数目，可以对几乎任何应用简化查找数据和随机查找的过程，否则要找到我们需要的结果需要浏览大量数据。

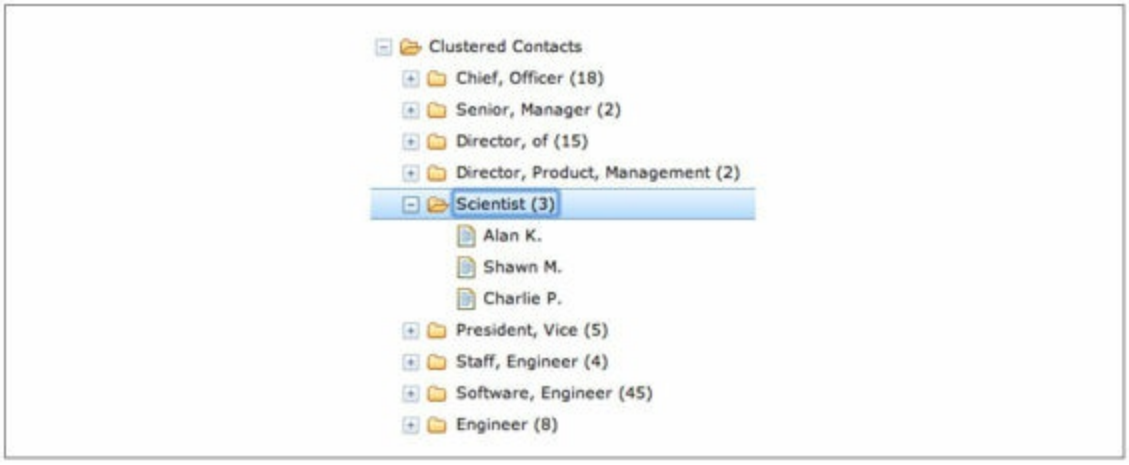


图3-3：合理聚类的数据适合以分组窗口的方式展示并能获得难以置信的用户体验

注意：从你的LinkedIn联系人中创建一个分组窗口展示的代码包含在本章用IPython Notebook写出的样例中。

如果使用成熟的Ajax工具包和其他的UI库，创建一个具有简单导航展示的代码可以十分容易，它展示数据的直观方式可以强化流程，对创建用户体验有不可思议的意义。一些简单如精巧的分级展示也能促使用户在网站上花更多的时间、使用户发现比平常更多的信息，并且最终在网站提供的服务中实现更多价值。

3.3.2 标准化数据以便分析

标准化是建立一个实用聚类算法必须且有效的过程，让我们来探索你可能在标准化LinkedIn数据过程中遇到的一些常见情形。本节我们将

实现一个标准化公司名字和职位的常见模式。一个更高级的练习，我们也将简单地转向并讨论LinkedIn档案信息中地理信息的二义性和坐标化的问题。（换句话说，我们将要尝试将LinkedIn档案中像“Greater Nashville Area”这样的标签转换为可以在地图上标出的坐标。）

注意：数据标准化的主要目的是允许你能够计数并分析数据的重要特征，并能够使用像聚类这样先进的数据挖掘技术。例如对LinkedIn的数据，我们会检视像公司的职位和地理位置这样的内容。

3.3.2.1 标准化及计算公司的数量

让我们尝试标准化职业网络中公司的名字。回忆一下你读取LinkedIn数据的两种基本方式，分别是使用LinkedIn API以编程的方式来检索相关字段，或通过使用较少人知道的机制将你的职业网络导出为包含姓名、工作职位、公司和联系人这类基本信息的通讯录数据。

假设你有从LinkedIn导出的联系人的CSV文件，你可以正规化该数据，并输出从柱状图中选择的实体，如示例3-6所示。

注意：你可能会在示例3-6的代码清单的公开评论中注意到，你需要复制并重新对你的LinkedIn联系人的CSV文件命名，该文件处于你源代码检查点下的某个目录，每一步均已在3.2.2节予以介绍。

示例3-6：简单标准化通讯录数据中的公司后缀

```
   import os
import csv
from collections import Counter
from operator import itemgetter
from prettytable import PrettyTable
# XXX: Place your "Outlook CSV" formatted file of connections from
# http://www.LinkedIn.com/people/export-settings at the following
# location: resources/ch03-linkedin/my_connections.csv
CSV_FILE = os.path.join("resources", "ch03-linkedin", 'my_connections.csv')
# Define a set of transforms that converts the first item
# to the second item. Here, we're simply handling some
# commonly known abbreviations, stripping off common suffixes,
# etc.
transforms = [(', Inc.', ''), (', Inc', ''), (', LLC', ''), (', LLP', ''),
              (' LLC', ''), (' Inc.', ''), (' Inc', '')]
csvReader = csv.DictReader(open(CSV_FILE), delimiter=',', quotechar='"')
contacts = [row for row in csvReader]
companies = [c['Company'].strip() for c in contacts if c['Company'].strip() != '']
for i, _ in enumerate(companies):
    for transform in transforms:
        companies[i] = companies[i].replace(*transform)
pt = PrettyTable(field_names=['Company', 'Freq'])
pt.align = 'l'
c = Counter(companies)
[pt.add_row([company, freq])
 for (company, freq) in sorted(c.items(), key=itemgetter(1), reverse=True)
    if freq > 1]
print pt
```

下面显示了频率分析的典型结果：

```
   +--------------------------------+------+
| Company                        | Freq |
+--------------------------------+------+
| Digital Reasoning Systems      | 31   |
| O拎eilly Media                 | 19   |
| Google                         | 18   |
| Novetta Solutions              | 9    |
| Mozilla Corporation            | 9    |
| Booz Allen Hamilton            | 8    |
| ...                            | ...  |
+--------------------------------+------+
```

注意：Python允许通过解引用列表和词典传递参数给函数，有时这会很方便，如示例3-6所示。例如，只要args定义为[1，7]，kw定义为{'x'：23}，调用f（*args，**kw）和调用f（1，7，x=23）是相同的。更多Python的技巧可以查看附录C。

记住，你需要更巧妙地处理更复杂的情况，如你可能会遇到公司名称的各种表示形式，如O’Reilly Media这个名字是经过多年变化后得到的。例如，你可能会看到该公司的名字表示为O’Reilly & Associates、O’Reilly Media、O’Reilly Inc.或只是O’Reilly。[2]

3.3.2.2 标准化及计算职位的数量

可能正如所期望的那样，当考虑职位名称的时候出现了和标准化公司名称相同的问题，不同的是这个问题变得更糟糕了，因为职位的名称有更多的变化。表3-1列出了你可能会在软件公司遇到的一些职位名称，它包含了一些很自然的变化。下面列出的10个职位名称有多少是不同意思的呢？

表3-1：技术公司的职位样例

工作职位
Chief Executive Officer
President/CEO
President & CEO
CEO
Developer
Software Developer
Software Engineer
Chief Technical Officer
President
Senior Software Engineer

尽管可以定义一个别名或缩写的列表来将像CEO和Chief Executive

Officer的名称等同起来，但是要人工定义列表将所有可能领域的一般情况下的职位名称（如Software Engineer和Developer）等同起来是不切实际的。然而，即便是在最混乱领域的最糟糕的情况下，实现将数据压缩到易于让专业人士进行复查并随后反馈给程序的程度不是很困难的，程序可以用与专家以前做过的几乎相同的方式来处理数据。大多数情况下，这也是各个机构所偏爱的方法，因为它能够让人简短地介入主循环并执行质量控制。

回想一下，当处理任何数据集的时候，一个最明显的起始点就是记录事件发生的次数，并且这种情形都是相同的。让我们重新使用标准化公司名称的相同思想来实现标准化常见的职位名称，随后对这些名称做一个基本的频率分析作为聚类的基础。假设你已经导出了一个合理数量的联系人资料，你遇到的职位名称间的细小差别实际上可能会是令人吃惊的，但是我们在开始之前，先介绍一些样例代码来建立一些标准化记录的模式，以及获得根据词频排序的基本表单。

示例3-7检查职位名称并打印职位和其中单个词项的频率信息。

示例3-7：标准化常见的职位名称并计算它们的频率

```
   import os
import csv
from operator import itemgetter
from collections import Counter
from prettytable import PrettyTable
# XXX: Place your "Outlook CSV" formatted file of connections from
# http://www.linkedin.com/people/export-settings at the following
# location: resources/ch03-linkedin/my_connections.csv
CSV_FILE = os.path.join("resources", "ch03-linkedin", 'my_connections.csv')
```

```
transforms = [
    ('Sr.', 'Senior'),
    ('Sr', 'Senior'),
    ('Jr.', 'Junior'),
    ('Jr', 'Junior'),
    ('CEO', 'Chief Executive Officer'),
    ('COO', 'Chief Operating Officer'),
    ('CTO', 'Chief Technology Officer'),
    ('CFO', 'Chief Finance Officer'),
    ('VP', 'Vice President'),
    ]
csvReader = csv.DictReader(open(CSV_FILE), delimiter=',', quotechar='"')
contacts = [row for row in csvReader]
# Read in a list of titles and split apart
# any combined titles like "President/CEO."
# Other variations could be handled as well, such
# as "President & CEO", "President and CEO", etc.
titles = []
for contact in contacts:
    titles.extend([t.strip() for t in contact['Job Title'].split('/')
                           if contact['Job Title'].strip() != ''])
# Replace common/known abbreviations
for i, _ in enumerate(titles):
    for transform in transforms:
        titles[i] = titles[i].replace(*transform)
# Print out a table of titles sorted by frequency
pt = PrettyTable(field_names=['Title', 'Freq'])
pt.align = 'l'
c = Counter(titles)
[pt.add_row([title, freq])
for (title, freq) in sorted(c.items(), key=itemgetter(1), reverse=True)
     if freq > 1]
print pt
# Print out a table of tokens sorted by frequency
tokens = []
for title in titles:
    tokens.extend([t.strip(',') for t in title.split()])
pt = PrettyTable(field_names=['Token', 'Freq'])
pt.align = 'l'
c = Counter(tokens)
[pt.add_row([token, freq])
for (token, freq) in sorted(c.items(), key=itemgetter(1), reverse=True)
    if freq > 1 and len(token) > 2]
print pt
```

简而言之，该代码读入CSV格式的记录，将用斜杠连在一起的名称分开（如“President/CEO”）并替换缩写词尝试着将名称标准化。除此之外，它仅仅显示了职位全称和包含在职位名称中的单个词项的频率分布结果。

这与之前用公司名称的练习没多少区别，但是它是一个有用的初始模板，并提供给你如何分解数据的一些合理洞见。

样本的结果如下：

```
   +-------------------------------------+------+
| Title                                 | Freq |
+-------------------------------------+------+
| Chief Executive Officer               | 19   |
| Senior Software Engineer              | 17   |
| President                             | 12   |
| Founder                               | 9    |
| ...                                   | ...  |
+-------------------------------------+------+
+---------------+------+
| Token         | Freq |
+---------------+------+
| Engineer      | 43   |
| Chief         | 43   |
| Senior        | 42   |
| Officer       | 37   |
| ...           | ...  |
+---------------+------+
```

对于样例结果值得注意的一件事是，使用精确匹配时最常见的职位名称是“Chief Executive Officer”，紧随其后的是其他高级职位，如“President”和“Founder”。因此，该职业网络的拥有者可以很好地接触到企业家和商业领袖。职位名称中最常用的词项是“Engineer”和“Chief”。“Chief”跟我们之前所设想的一样，是与公司高层相关的，而“Engineer”对职业网络的本质提供了略微不同的暗示。尽管“Engineer”不是最常出现职位名称的一部分，但它确实出现在大量职位名称中，如“Senior Software Engineer”和“Software Engineer”，这些词也出现在职位名称列表的前几位。因此，该网络的拥有者看起来也与技术开发人员有联系。

在职位名称或通讯录的数据分析中，这种方法促进了对近似匹配和聚类算法的需求。下一节将进一步研究。

3.3.2.3 标准化及计算位置数量

尽管LinkedIn包括了一个大概的地理区域，这个区域通常与你的每个联系人所在的一个城市区域相同，但是如果没有额外的信息是不能够在地图上标出正确位置的。知道某人工作在“Greater Nashville Area”是有用的，并且通过一些额外知识，会知道他所在的位置可能是Tennessee州Nashville城区。然而，写代码来实现将“Greater Nashville Area”转换为可以在地图上找到的一系列坐标要比听起来难，尤其是当一个区域的可认读标签十分宽泛的情况下。

作为一个广义问题，找准地理位置的实际指代十分困难。纽约城的人口十分多，以至于你可以合理地推断纽约指的就是纽约州的纽约城，但是“Smithville”呢？在美国有成百上千个Smithville，大多数的州都会有几个Smithville，因而要做出正确的决定我们需要除了周边州以外的上下文地理信息。像“Greater Smithville Area”这样十分模糊的信息在LinkedIn上不会出现，但是它演示了如何找准地理位置实际指代的普遍问题，这样地理信息就可以转变到特定坐标中。

找出并地理编码LinkedIn联系人的位置要比最广义的问题形式容易些，因为大多数的职业人员倾向于和比较大的都市区域有联系，并且这

样的区域是相对有限的。尽管不总是这样，但是你通常可以粗略的假定LinkedIn档案中的位置信息都是相对知名的地方，并且地点的名字可能就是人口最多的都市区域。

你可以通过使用命令pip install geopy来安装一个叫做geopy的Python包；它提供了一个普遍的方法来传递位置的标注，并传回可能匹配的坐标列表。geopy包是多个web服务供应商（如Bing和Google）的代理，用来执行地理编码，使用该包的优势之一就是该包提供了多种地理编码服务的标准API接口，这样你就不需要手动的设计请求并解析反馈。geopy在GitHub的代码库（http://bit.ly/1a1m7Ka）是阅读在线文档的起点。

示例3-8展示了如何利用Microsoft的Bing来使用geopy，Bing提供了丰富的API但需要账号，该API只做教育用途，如本书对它的学习。运行该脚本，你需要向Bing发送API密匙请求（http://bit.ly/1a1m5lq）。

注意：对本书使用geopy的练习，推荐Bing作为地理编码器。鉴于产品战略的一些调整，出现了一个叫做Yahoo！BOSS Geo（http://yhoo.it/1a1m5C4）服务的新产品，导致在写作本书之际Yahoo！地理编码服务是不可操作的。尽管Google地图（v3）API也是可用的，但它每天可请求的最大次数远比Bing允许的少。

示例3-8：用Microsoft Bing地理编码位置

```
   from geopy import geocoders
GEO_APP_KEY = '' # XXX: Get this from https://www.bingmapsportal.com
```

```
g = geocoders.Bing(GEO_APP_KEY)
print g.geocode("Nashville", exactly_one=False)
```

关键词参数exactly_one=False告诉地理编码器如果有不止一个可能的结果不要报错，这种情况是很普遍的。该脚本的样例结果演示了使用一个像“Nashville”这样的模糊标签来决定一系列坐标的实质过程。

```
   [(u'Nashville, TN, United States', (36.16783905029297, -86.77816009521484)),
 (u'Nashville, AR, United States', (33.94792938232422, -93.84703826904297)),
 (u'Nashville, GA, United States', (31.206039428710938, -83.25031280517578)),
 (u'Nashville, IL, United States', (38.34368133544922, -89.38263702392578)),
 (u'Nashville, NC, United States', (35.97433090209961, -77.96495056152344))]
```

Bing地理编码服务会在结果列表中首先返回人口最多的位置，因此假定LinkedIn通常在档案中公开的位置是大都市地区，我们会倾向简单地选择列表中的第一项作为我们的返回值。然而，在我们地理编码前，我们不得不回到数据标准化的问题上，因为传入像“Greater Nashville Area”这样的值到地理编码器中不会给我们返回值（你可以自己试试看）。一种模式是，我们转换位置的形式，这样通常就会去除常见的前缀和后缀，如示例3-9所示。

示例3-9：用Microsoft Bing地理编码LinkedIn联系人的位置

```
   from geopy import geocoders
GEO_APP_KEY = '' # XXX: Get this from https://www.bingmapsportal.com
g = geocoders.Bing(GEO_APP_KEY)
transforms = [('Greater ', ''), (' Area', '')]
results = {}
for c in connections['values']:
    if not c.has_key('location'): continue
    transformed_location = c['location']['name']
    for transform in transforms:
        transformed_location = transformed_location.replace(*transform)
    geo = g.geocode(transformed_location, exactly_one=False)
    if geo == []: continue
```

```
    results.update({ c['location']['name'] : geo })
print json.dumps(results, indent=1)
```

地理编码练习的样例结果如下：

```
   {
 "Greater Chicago Area": [
  "Chicago, IL, United States",
  [
   41.884151458740234,
   -87.63240814208984
  ]
 ],
 "Greater Boston Area": [
  "Boston, MA, United States",
  [
   42.3586311340332,
   -71.05670166015625
  ]
 ],
 "Bengaluru Area, India": [
  "Bangalore, Karnataka, India",
  [
   12.966970443725586,
   77.5872802734375
  ]
 ],
 "San Francisco Bay Area": [
  "CA, United States",
  [
   37.71476745605469,
   -122.24223327636719
  ]
 ],
 ...
}
```

本章的后面部分，我们将使用地理编码的坐标作为聚类算法的一部分，聚类算法是分析你职业网络的很好的方法。同时，另外一种叫做统计地图的可视化技术可以将你的人脉网络可视化，引发你的兴趣。

3.3.2.4 用统计地图可视化位置信息

统计地图（cartogram）（http://bit.ly/1a1m5Ss）是一种可视化技

术。它根据控制参数按比例缩放地理的边界来展示地理信息。例如，一幅美国地图可能将每个州的大小按比例缩放，这个比例是根据像肥胖率、贫困水平、百万富翁的数量等参数决定的。可视化结果不需要呈现完全整合的地理内容，因为每个州缩放比例不一致将不再适合整合在一起。你可能还会想到对控制每个州进行比例缩放的参数的总体状态做些文章。

一个叫Dorling Cartogram（http://stanford.io/1a1m5SA）统计地图的特殊变化是用像圆圈这样的图形来替代地图上每个区域，这些图形是根据控制参数，按其原区域所在大概位置及大小比例进行设定的。表述Dorling Cartogram的另外一个方式是将其作为“地理聚合的泡泡图”。它是一个重要的可视化工具，因为它允许你用直觉来确定信息可以出现在2D地图表面的什么位置，你也能用非常直观的图形属性（如面积和颜色）来编码参数。

假设Bing地理编码服务返回的结果包括了每个城市都被地理编码的州，让我们来利用好这个信息并建立你人脉的Dorling Cartgogram统计地图，你可以根据在每个州你所有的人脉数量来对其进行比例缩放。在第2章介绍的先进可视化工具包D3包括了Dorling Cartogram的大部分机制，并提供了高度可定制的途径，如果你有需要，它可以将你的可视化扩展加入进其他的一些参数。D3也包括了可以表达地理信息的一些其他可视化方式，如heatmaps、symbol maps和choropleth maps，它们可以

很容易地表达工作数据的内容。

为了按州来将你的通讯录可视化，你还需要执行最后一个数据重组的步骤，就是要解析地理编码器返回的州的信息。总的来讲，包括各州的文本信息的返回值可能会有一些细微的不同，但是作为一般的模式，每个州通常是用两个连续的大写字母来表示，正则表达式（http://bit.ly/1a1m6pJ）是解析文本中模式的一个好方法。

示例3-10阐明了如何使用Python标准库的re包来解析地理编码器的返回值，并将其写到JSON文件中，该文件可以通过D3支持的Dorling Cartogram可视化来读取。正则表达式的内容不在我们现在要讨论的范畴中，但是我们查找的文本中2个连续的大写字母的核心模式是'.*（[A-Z]{2}）.*'，用.*通配符代表的符号可以在任何文本的前面或后面。小括号是用来获取（或用正则表达式语法叫做“tag”）我们感兴趣的组合，这样我们就可以很容易地检索出需要的内容。

示例3-10：用正则表达式解析Bing地理编码器结果中州的名称信息

```
   import re
# Most results contain a response that can be parsed by
# picking out the first two consecutive upper case letters
# as a clue for the state
pattern = re.compile('.*([A-Z]{2}).*')
def parseStateFromBingResult(r):
    result = pattern.search(r[0][0])
    if result == None:
        print "Unresolved match:", r
        return "???"
    elif len(result.groups()) == 1:
        print result.groups()
        return result.groups()[0]
    else:
        print "Unresolved match:", result.groups()
```

```
        return "???"
transforms = [('Greater ', ''), (' Area', '')]
results = {}
for c in connections['values']:
    if not c.has_key('location'): continue
    if not c['location']['country']['code'] == 'us': continue
        transformed_location = c['location']['name']
        for transform in transforms:
        transformed_location = transformed_location.replace(*transform)
    geo = g.geocode(transformed_location, exactly_one=False)
    if geo == []: continue
    parsed_state = parseStateFromBingResult(geo)
    if parsed_state != "???":
        results.update({c['location']['name'] : parsed_state})
print json.dumps(results, indent=1)
```

样例结果如下，它表明了该技术是有效的：

```
   {
 "Greater Chicago Area": "IL",
 "Greater Boston Area": "MA",
 "Dallas/Fort Worth Area": "TX",
 "San Francisco Bay Area": "CA",
 "Washington D.C. Metro Area": "DC",
 ...
}
```

用从你的LinkedIn通讯录中提取可靠的州名称缩写的技术，我们现在可以计算每个州出现的频率，这正是我们要用D3来进行Dorling Cartogram可视化所需要的内容。一个职业网络的可视化样例如图3-4所示。注意，在许多统计地图中，阿拉斯加和夏威夷经常是在可视化图（许多地图也是将它们作为镶嵌物在地图中显示）的左下角位置展示。尽管可视化仅仅是地图上的一些精心展示的圆圈，但是每个圆圈所对应的州是比较明显的。鼠标停留在圆圈上就会默认弹出含有州名称的工具提示，并且通过观察标准的D3最优范例，添加额外的定制信息是不难的。产生用于D3可视化的输出结果的过程，不过是获取各州出现的频率分布并将其输出到JSON文件中。

注意：为了简便起见，本节省略了根据你的LinkedIn联系人创建Dorling Cartogram的一些代码，但是完整的代码包含在本章IPython Notebook的完整样例中。

3.3.3 测量相似度

之前我们对数据进行了标准化的操作，现在我们将注意力转向计算相似度的问题，这是聚类问题的核心基础。以有效的方式聚类一系列字符串（本例中是指职位名称）中需要我们做出最多的决定是，使用哪种相似度度量方法。有许多有用的字符串相似度度量方法，根据你的目标选择更适合你的方法。

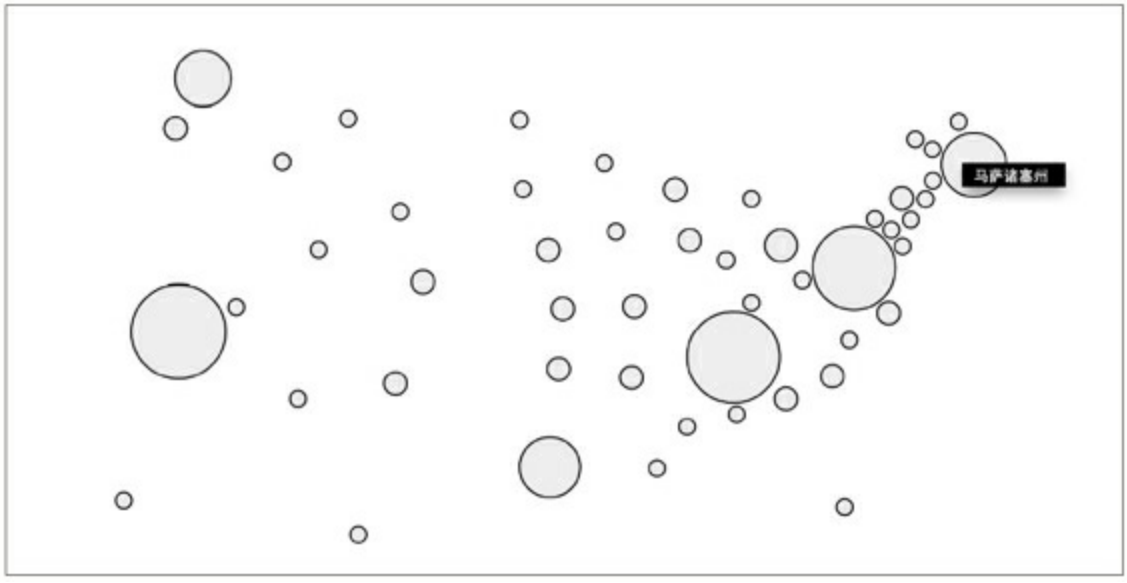


图3-4：根据解析自LinkedIn职业网络的位置信息生成的Dorling

Cartogram统计地图，鼠标放在圆圈上（在本图中，鼠标放在了马萨诸塞州的上面），会出现含有该州名字的工具提示

尽管这些相似度的计算方法不难定义和计算，但是正好借这个机会来介绍自然语言工具箱（Natural Language Toolkit，NLTK）（http://bit.ly/1a1mc0m），它是一个Python工具箱，是挖掘社交网络的“新武器”。和其他的Python包安装方法相同，只需要运行pip install nltk命令就可以安装。

注意：根据你对NLTK的使用情况，你可能还需要下载一些附加的数据集，这些数据没有和NLTK打包在一起。如果你没有使用本书的虚拟机，可以运行nltk.download（）命令来下载NLTK附加的数据集。你可在http://bit.ly/1a1mcgV上阅读更多关于安装NLTK数据的内容。

这里是一些常见的相似度度量方法，它们可能对NLTK中实现的职位名称的比较有帮助：

编辑距离

编辑距离（edit distance）也叫做Levenshtein距离（http://bit.ly/1a1mcNO），是对从一个字符串转为另一个字符串所需的插入、删除和替换次数的简单测量。例如，将dad转换为bad所需的花费为一次替换操作（将第一个d替换为b），同时产生值为1。NLTK通过nltk.metrics.distance.edit_distance函数来实现编辑距离的度量。

两个字符串之间实际的编辑距离不同于计算编辑距离所需的操作数量；对于长度为M和N的两个字符串计算编辑距离通常需要M×N次操作。换句话说，计算编辑距离是一个计算复杂度很高的操作，因此针对一般的数据需要巧妙的使用该方法。

n元文法相似度

n元文法是一种简洁表达来自文本的连续n个符号的每一种排列的可能性，也提供了计算搭配词所需的基本数据结构。n元文法相似度有许多种变形，但是我们考虑最直接的方式：计算2个字符串中所有词的所有可能二元文法，并通过计算字符串间相同的二元文法数来对字符串的相似度打分，如示例3-11所示。

注意：n元文法和搭配的扩展讨论在4.4.4节。

示例3-11：用NLTK来计算二元文法

```
   ceo_bigrams = nltk.bigrams("Chief Executive Officer".split(), pad_right=True,
                                                                  pad_left=True)
cto_bigrams = nltk.bigrams("Chief Technology Officer".split(), pad_right=True,
                                                                   pad_left=True)
print ceo_bigrams
print cto_bigrams
print len(set(ceo_bigrams).intersection(set(cto_bigrams)))
```

下面的样例结果就是二元文法的计算以及两个不同的职位名称间相同二元文法的数量：

```
   [(None, 'Chief'), ('Chief', 'Executive'), ('Executive', 'Officer'),
```

```
('Officer', None)]
[(None, 'Chief'), ('Chief', 'Technology'), ('Technology', 'Officer'),
('Officer', None)]
2
```

关键词参数pad_right和pad_left的使用是为了允许开头和结尾字符也进行匹配。该操作允许二元文法（None，'Chief'）的出现，这对于职位名称的对比是很重要的。NLTK通过定义在nltk.metrics.association模块中的BigramAssociationMeasures和TrigramAssociationMeasure类来提供使用广泛的二元文法和三元文法的打分函数。

Jaccard距离

通常情况下，我们可以计算两个集合（其中集合是指无序元素的群组）之间的相似度。Jaccard相似度度量就是表达两个集合之间的相似度，并定义为集合之间相交与相并的比值。数学上来说，Jaccard相似度可以写为：

$$\frac{|Set1 \cap Set2|}{|Set1 \cup Set2|}$$

也就是两个集合中共有元素的数量（交集的基）除以两个集合中截然不同元素的数量（并集的基）。该比值的直觉思想在于该计算是一种获得归一化相似度的合理方法。总的来说，你可以使用n元文法（包括一元文法）计算Jaccard相似度来测量两个字符串的相似度。

鉴于Jaccard相似度测量了两个集合之间的紧密度，那么你可以通过

用1.0减去该值来获得我们所知道的Jaccard距离。

注意：除了这些方便的相似度度量方法和许多其他的实用工具，NLTK还提供了一个类，你可以通过nltk.FreqDist来访问。这是一个频率分布，它和我们一直使用的Python标准库中的collections.Counter使用方式相似。

计算相似度是任何聚类算法的关键部分，一旦你对正在挖掘的数据有更好想法，你可以很容易尝试不同的相似度启发式，它是你在数据科学领域工作的一部分。下一节我们将编写用Jaccard相似度来聚类职位名称的脚本。

3.3.4 聚类算法

有了我们之前的数据标准化以及相似度启发式，现在我们从LinkedIn上收集一些真实的数据，并进行一些有意义的聚类来洞察你职业人脉的更多信息。不论你是想客观地看看你增添人脉的技巧是否在帮助你遇到“正确的人群”、你是想用一种特殊的商业调查或建议来找到最可能符合特定社会经济体的人，或者你是想确定是否有一个更好的地方供你居住或者开一个远程办事处来推广商业业务，使用大量高质量的数据一定会在职业人脉中出现有价值的事物。本节剩下的部分会通过更深入地思考分组相似的职位名称的问题来展示一些不同的聚类方法。

3.3.4.1 贪心聚类

假定我们认为职位中的重叠部分很重要，我们通过计算Jaccard距离将职位名称互相比较（示例3-7的延伸）来将它们聚类。示例3-12将相似的名称聚类然后展示相应的通讯录。浏览代码，尤其是调用DISTANCE函数的嵌套循环，然后再讨论。

示例3-12：用贪心启发式来聚类职位名称

```
   import os
import csv
from nltk.metrics.distance import jaccard_distance
# XXX: Place your "Outlook CSV" formatted file of connections from
# http://www.linkedin.com/people/export-settings at the following
# location: resources/ch03-linkedin/my_connections.csv
CSV_FILE = os.path.join("resources", "ch03-linkedin", 'my_connections.csv')
# Tweak this distance threshold and try different distance calculations
# during experimentation
DISTANCE_THRESHOLD = 0.5
DISTANCE = jaccard_distance
def cluster_contacts_by_title(csv_file):
    transforms = [
        ('Sr.', 'Senior'),
        ('Sr', 'Senior'),
        ('Jr.', 'Junior'),
        ('Jr', 'Junior'),
        ('CEO', 'Chief Executive Officer'),
        ('COO', 'Chief Operating Officer'),
        ('CTO', 'Chief Technology Officer'),
        ('CFO', 'Chief Finance Officer'),
        ('VP', 'Vice President'),
        ]
    separators = ['/', 'and', '&']
    csvReader = csv.DictReader(open(csv_file), delimiter=',', quotechar='"')
    contacts = [row for row in csvReader]
    # Normalize and/or replace known abbreviations
    # and build up a list of common titles.
    all_titles = []
    for i, _ in enumerate(contacts):
        if contacts[i]['Job Title'] == '':
            contacts[i]['Job Titles'] = ['']
            continue
        titles = [contacts[i]['Job Title']]
        for title in titles:
            for separator in separators:
                if title.find(separator) >= 0:
                    titles.remove(title)
                    titles.extend([title.strip() for title in title.split(separator)
```

```
                                if title.strip() != ''])
            for transform in transforms:
                titles = [title.replace(*transform) for title in titles]
            contacts[i]['Job Titles'] = titles
            all_titles.extend(titles)
        all_titles = list(set(all_titles))
        clusters = {}
        for title1 in all_titles:
            clusters[title1] = []
            for title2 in all_titles:
                if title2 in clusters[title1] or clusters.has_key(title2) and title1 \
                    in clusters[title2]:
                    continue
                distance = DISTANCE(set(title1.split()), set(title2.split()))
                if distance < DISTANCE_THRESHOLD:
                    clusters[title1].append(title2)
            # Flatten out clusters
            clusters = [clusters[title] for title in clusters if len(clusters[title]) > 1]
            # Round up contacts who are in these clusters and group them together
            clustered_contacts = {}
            for cluster in clusters:
                clustered_contacts[tuple(cluster)] = []
                for contact in contacts:
                    for title in contact['Job Titles']:
                        if title in cluster:
                            clustered_contacts[tuple(cluster)].append('%s %s'
                                % (contact['First Name'], contact['Last Name']))
        return clustered_contacts
    clustered_contacts = cluster_contacts_by_title(CSV_FILE)
    print clustered_contacts
    for titles in clustered_contacts:
        common_titles_heading = 'Common Titles: ' + ', '.join(titles)
        descriptive_terms = set(titles[0].split())
        for title in titles:
            descriptive_terms.intersection_update(set(title.split()))
        descriptive_terms_heading = 'Descriptive Terms: ' \
            + ', '.join(descriptive_terms)
        print descriptive_terms_heading
        print '-' * max(len(descriptive_terms_heading), len(common_titles_heading))
        print '\n'.join(clustered_contacts[titles])
        print
```

上述代码首先利用常用连词的列表将连在一起的名称分开，然后将常用职位名称标准化。其次用一个嵌套循环迭代所有的职位名称，并根据Jaccard相似度（DISTANCE定义的）的阈值将它们聚类，这里采取把jaccard_distance分配给DISTANCE的方式，我们可以很容易替换为不同的距离计算方式来做实验。这个紧凑的循环是代码清单中大部分的计算真正发生的地方：每个名称都和其他的相比较。

如果任何两个名称的距离（根据相似度启发式而定）是“足够接近的”，我们贪心地将它们分为一组。这里“贪心”的意思也就是我们第一次能够认定一个条目适合一个类别时，我们就将其分到一类而不再进一步考虑其是否有更适合的类别，如果后面出现更适合的我们也不再考虑了。尽管该方法很实用，也不影响其得到合理的结果。很明显，选择有效的相似度启发式是成功的关键，但是考虑到嵌套循环的特性，我们调用打分函数的次数越少，代码执行的速度就越快（主要关心对于大量数据的操作）。下一节我们将谈论更多这方面的考虑，但是注意，我们尽可能用一些条件判断来避免重复的不必要的计算。

剩下的代码仅仅是查找通讯录，把具有特定职位名称的联系人分为一组展示，但是在聚类的过程中有另一个需要考虑的细节：你通常需要为每一类分配一个有意义的标签。该方法的实现是考虑每类职位名称中的相交词项来计算其类别，这是最明显而常见的思路，看起来是合理的。如果用其他的方法来计算距离肯定会有不同。

你从这个代码中得到的结果类型是有用的，因为它将在工作中可能有相同职责的人分在了一起。正如我们前面提到的，出于多个原因，该信息可能会是有用的，如你在计划一个有“CEO Panel”的活动，尝试找到最能帮助你做出下一步职业规划的人，或是考虑到你自己的职业责任和对未来的愿望，来试图确定你是否跟其他有相似职业的人有足够好的联系。对于一个样例职业人脉聚类的简略结果如下：

```
   Common Titles: Chief Technology Officer,
                  Founder,
                  Chief Technology Officer,
                  Co-Founder,
                  Chief Technology Officer
Descriptive Terms: Chief, Technology, Officer
-------------------------------------------
Damien K.
Andrew O.
Matthias B.
Pete W.
...
Common Titles: Founder,
               Chief Executive Officer,
               Chief Executive Officer
Descriptive Terms: Chief, Executive, Officer
-------------------------------------------
Joseph C.
Janine T.
Kabir K.
Scott S.
Bob B.
Steve S.
John T. H.
...
```

运行时间分析

注意：本节包括对聚类计算细节的相对深入的讨论，可以选择性的阅读，因为不是所有人都对它感兴趣。如果这是你第一次阅读本章，可以跳过本节，当你再次遇到该问题的时候再详细阅读。

最差的情况下，示例3-12中的嵌套循环执行DISTANCE函数，需要花费我们之前提到的O（n^2）的时间复杂度，换句话说，它将被调用len（all_titles）*len（all_titles）次。对于非常大的n值，嵌套循环为达到聚类目的而将每个元素与其他的元素相比较，这不是一个具有良好扩展性的方法，但是考虑到你的职业人脉中不同职位名称的数量不可能非常大，它应该不会造成太大的性能限制。这可能看起来不是一个大问

题，毕竟只是一个嵌套循环，但是复杂度为O（n^2）的算法的症结在于，处理输入数据集合所需的比较次数随集合中的元素数量增加而成指数倍增加。例如，对于含有100个职位名称的小输入集合只需要10000次打分操作，然后对于10000个职位名称的输入集合将会需要100000000次打分操作。这种操作从数学上来讲不会运行很好，并最终会遇到问题，即使是你用很多硬件来进行操作。

当遇到看起来是不可扩展的窘境时，你的最初反应可能是尽量缩小n值。但是大多数情况下随着输入数据的增加，你不可能将其减小很多而使你的解决方案可扩展，因为你的算法复杂度仍然是O（n^2）。我们真正想要做的是设计一个复杂度是O（k×n）的算法，这里k是一个比n小得多的数，并且是易控数值，也就是随着n值的增长，k值的增长要慢得多。和任何其他工程决策一样，在真实世界的每个角落都有性能和质量因素的权衡，并且获得恰当的平衡是十分具有挑战的事情。事实上，许多成功实现可扩展的匹配分析的数据挖掘公司，都将其实现方式作为知识产权（商业秘密），因为这些方法会带来确定的商业优势。

对于复杂度是O（n^2）的算法是不可接受的，你可能尝试重写上述基本方法的嵌套循环，随机的选择样本来进行打分，这样就会使复杂度减为O（k×n），这里k是抽样的大小。然而随着抽样的大小接近n，其运行的时间也开始接近O（n^2）的运行时间。下面对于示例3-12的修改显示了抽样技术在代码中的具体实现；与之前代码的不同之处用粗体来

突出。对于每个外层循环，我们执行的内层循环是小了很多的一个固定次数：

```
   # ... snip ...
all_titles = list(set(all_titles))
clusters = {}
for title1 in all_titles:
    clusters[title1] = []
    for sample in range(SAMPLE_SIZE):
        title2 = all_titles[random.randint(0, len(all_titles)-1)]
        if title2 in clusters[title1] or clusters.has_key(title2) and title1 \
            in clusters[title2]:
            continue
        distance = DISTANCE(set(title1.split()), set(title2.split()))
        if distance < DISTANCE_THRESHOLD:
            clusters[title1].append(title2)
# ... snip ...
```

另外一个方法是随机地将数据抽样到n个桶中（n是一个通常不大于数据集中元素数平方根的数），然后在每个桶中执行聚类操作，随后根据情况合并输出结果。例如，如果你有100万个数据，一个复杂度为O（n^2）的算法就需要执行1万亿次逻辑操作，然而将100万的数据放入1000个桶中，每个桶中有1000个数据，聚类每个桶只需要10亿次操作。（1000个桶，每个桶进行1000×1000次操作）。10亿仍然是一个很大的数字，但是比1万亿小3个数量级，这也是一个巨大的进步（尽管在一些情况下，该进步还不够大）。

除了抽样和分桶的方法，文献中还有许多其他的方法，可以更好地降低问题的规模。例如，理论上你应该比较数据集中的每个数据，你所使用的特定技术对于较大n值应该避免复杂度是O（n^2）的情形，相反要基于现实世界的限制和你想获得的结果而不同，通过实验和特定领域的

知识来选取不同的技术。当你在考虑可实施性的时候，记住机器学习领域提供了许多技术来应对这类问题，可以使用不同的概率模型和成熟的采样技术来解决。在后面的k-means聚类部分，我们将会介绍这个直观且著名的聚类算法，它是一个聚类多维空间的通用的无监督方法。我们将在后面使用该技术针对地理坐标来聚类你的通讯录。

3.3.4.2 层次聚类

示例3-12介绍了一个直观、贪心的聚类方法。它是作为练习让你了解问题的根本所在。现在有了对基本问题的正确认知，我们来介绍2个常用的聚类算法，分别是层次聚类和k-means聚类。这两个算法在你的数据挖掘职业生涯中会经常遇到并应用到不同的场合。

层次聚类表面上与我们一直在使用的贪心启发式相似，而k-means聚类确实有根本上的不同。本章的剩下章节我们重点关注k-means聚类方法，但是有必要简单地介绍下这两个方法的基本理论，因为你很有可能在阅读文章和做研究的时候遇到它们。可以使用cluster模块完美地实现这两个方法，该模块可以通过命令pip install cluster来安装。

层次聚类是确定性的技术，它计算了所有数据之间的距离并得到一个全矩阵（注：全矩阵的计算意味着多项式的运行时间。对于聚合聚类，运行时间的复杂度通常是O（n^3）。），然后遍历该矩阵将满足最小距离阈值的数据归为一类。遍历矩阵是分层次的，在遍历矩阵的过程

中对数据聚类，会产生一个树结构，来表达数据间的相对距离。在文献中，你可能会看到这种技术叫做“聚合”（agglomerate），因为它先通过将每个数据项分到不同的类别来建立一个树结构，然后这些类分层次地归并到其他类中直至所有的数据集都聚集到树顶。树的叶节点表示正在被聚类的数据，而树的中间节点则分层次的将这些数据聚合在不同类中。

为使凝合的想法概念化，我们先来看看图3-5，会注意到叫“Andrew O.”和“Matthias B.”的联系人是聚类后树的叶子，而如“Chief，Technology，Officer”的节点将这些叶子聚合成一类。尽管树的结构只有两层，但是不难想象如果再聚合一层，也就是使用如“Chief，Officer”之类的标签概念化商业主管，它可以将“Chief，Technology，Officer”和“Chief，Executive，Officer”节点聚合在一起。

聚合技术和示例3-12中的方法相似，但它们的本质是不同的，示例3-12是用贪心启发式来聚类，而不是连续地构建层级。因此，运行层次聚类的代码可能会花费更长的时间，你可能还需要调整打分函数和相应的距离阈值（注：动态规划（http://bit.ly/1a1maFO）的使用和其他巧妙的统计技术会省去大量执行时间，使用实现好的工具包的一个优势就是这些巧妙的优化通常都已经帮你实现好了。例如，像职位名称这样的两个元素之间的距离几乎肯定是对称的，你只需要计算一半的距离矩阵，而不是全部的。这样，即使算法的复杂度仍旧是O（n^2），也只有$n^2/2$

的运行次数而不是n^2次。）。通常，聚合聚类不适合大的数据集，因为它会花费不可估量的时间。

如果我们要用cluster包来重写示例3-12，执行聚类中DISTANCE计算的嵌套循环可以用下面的代码来替代：

```
   # ... snip ...
# Define a scoring function
def score(title1, title2):
    return DISTANCE(set(title1.split()), set(title2.split()))
# Feed the class your data and the scoring function
hc = HierarchicalClustering(all_titles, score)
# Cluster the data according to a distance threshold
clusters = hc.getlevel(DISTANCE_THRESHOLD)
# Remove singleton clusters
clusters = [c for c in clusters if len(c) > 1]
# ... snip ...
```

如果你对层次聚类的变形感兴趣，一定要查看一些HierarchicalClustering类中的setLinkageMethod方法，该方法对于计算类之间的距离有些许变形。例如，你可以详细的说明类别之间的聚类是由两类之间的最短距离、最长距离还是平均距离来决定的。根据你数据的分布，选择不同的链接方法可能会得到十分不同的结果。

图3-5和图3-6分别展示一部分职业网络的树状图和节点连接的树结构，它们是由我们之前介绍的可视化工具D3（http://bit.ly/1a1kGvo）制作而成的。节点连接的布局更有效的利用了空间，可能使用该方案对于这个数据集来说是更好的选择，然而如果你想在更复杂的数据集中找到树的每一层之间的联系，树状图（http://bit.ly/1a1md4B）是个非常好的选择（也就是相当于层次聚类中每层的聚合）。如果展开的层次更深，

树状图会有明显的优势，但是当前的聚类方法只有两层，因此对于该数据集来说，相比于其他方式，节点连接展开的布局看起来更美观。记住我们在这展示的两种可视化方法实质上只是将图3-3的内容用交互树来形象化。如这些可视化效果图所展示的，当你能看到你的职业人脉的一个简图时，会出现许多令人吃惊的信息。

注意：为了简洁起见，这里省略了上述两种可视化的方法的代码，但是它们会包含在本章的IPython Notebook中。

3.3.4.3 k-means聚类

层次聚类是一个确定性技术，它会遍历所有的可能，因而通常会有O（n^3）的计算复杂度，而k-means聚类通常只需要O（k×n）次计算。即使对于小的k值，节省的计算次数也是非常大的。计算性能的提高是以对结果近似为代价的，但是结果仍有可能是非常棒的。在包含n个点的多维空间，该算法要聚集k类所需的步骤如下：

1.随机的在数据空间中选择k个点做初值，它们会被用来计算k个类：K_1，K_2，…，K_k。

2.对于每个点，找到距离最近的K_i，并将该点归入K_i所在的类，这样只需要k×n次比较就有效地将它们分成了k类。

3.对于k类中每个类，计算其中心值（http://bit.ly/1a1mbcW）或均

值，并将K_i替换为该值。（这样，算法的每次迭代就会计算“k个均值”）。

4.重复上述2~3步，直至两个迭代周期之间每类的成员不再发生改变。一般来说，只需若干次迭代该算法就会收敛。

k-means第一眼看起来可能不是太直观，图3-7展示了算法每次迭代得到的结果，如网上的“聚类算法教程”（http://bit.ly/1a1mbtp）所显示的那样，该图是用一个交互的Java小程序来实现的。使用的样本参数涉及了100个数据点，k的值设为3，也就是说该算法会产生3类。每次迭代需要重点注意的是方框的位置以及这些数据点的归类情况。该算法只需要9次迭代就完成了。

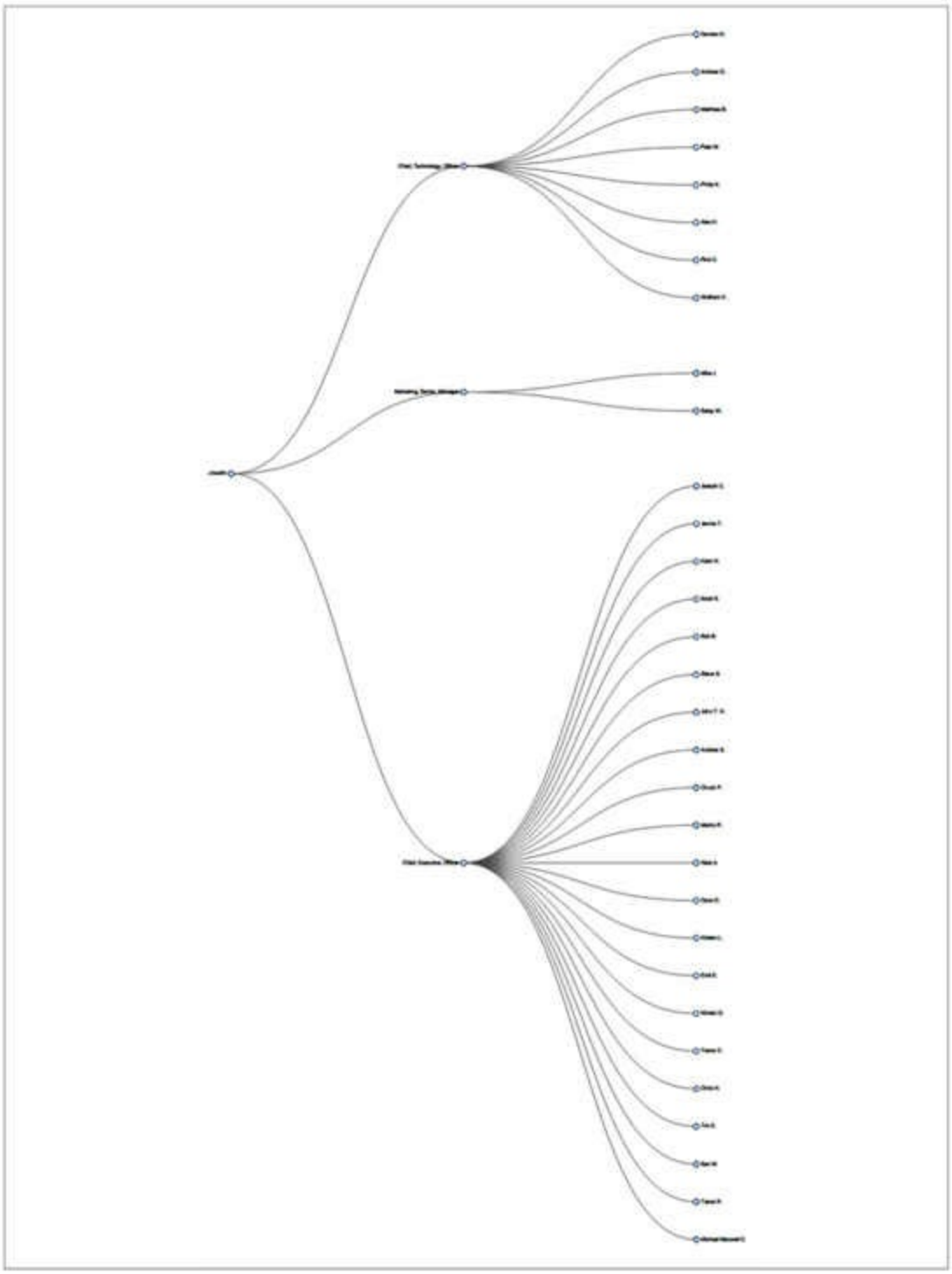


图3-5：通过职位名称聚类得到的通讯录的树状图结构，树状图通常是用显式分层方式来显示的

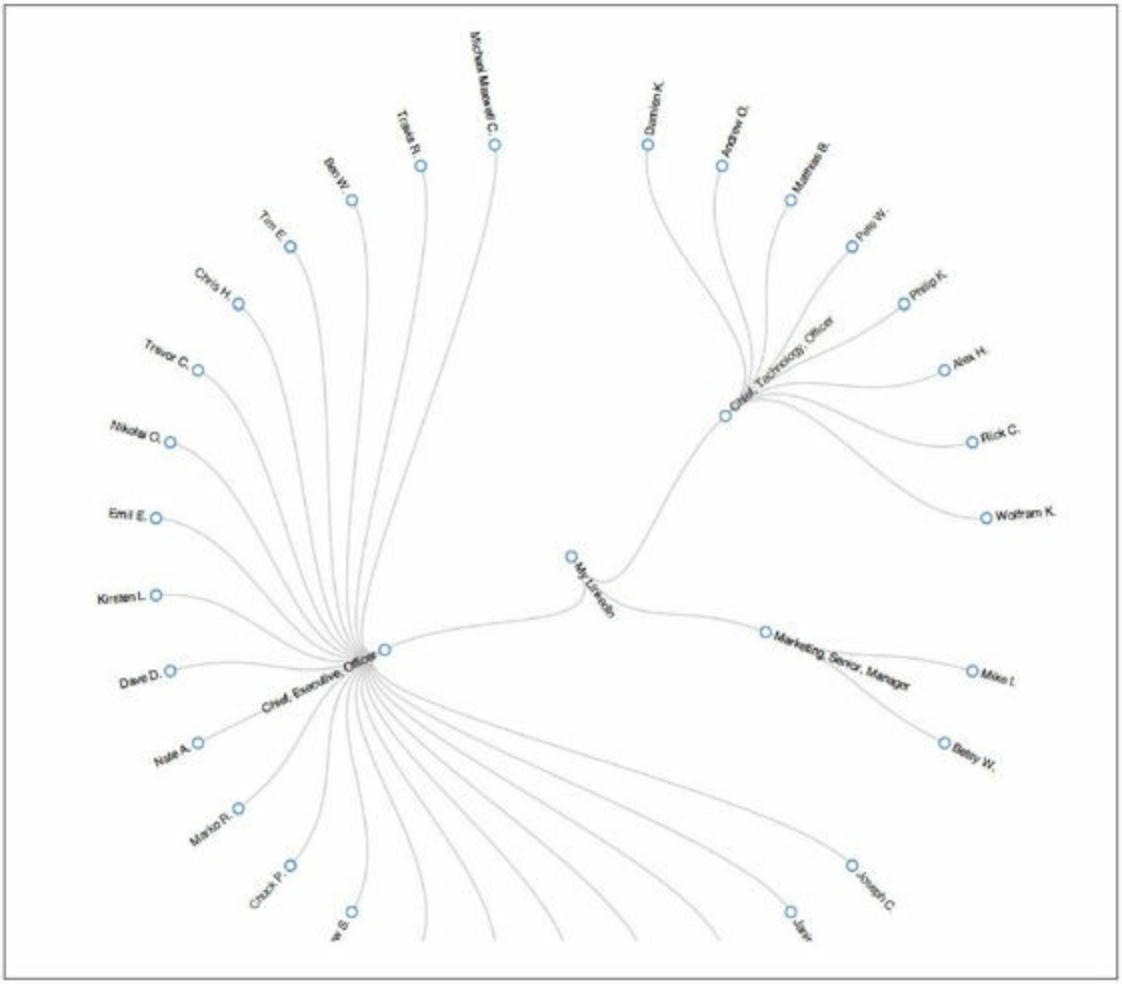


图3-6：通过职业名称聚类得到的通讯录的节点连接树的结构，它和图3-5所表达的信息相同，节点连接树与上一种方法相比较是一种更美观且令人满意的结构

尽管你可以在2维或2000维的点上运行k-means方法，但是通常情况下维度的范围都在几十，然而最最常见的情况是2维或3维。当你工作的空间维度相对比较小的时候，k-means可以是一个有效的聚类技术，因为它会运行的相当快并能够产生十分合理的结果。然而你需要选择适当的k值，该值一般不是显而易见的。

本节剩下的内容会演示如何使用k-means方法对你的职业人脉进行地理上的聚类并将其可视化，同时用Google地图（http://bit.ly/1a1mdRV）或Google地球（http://bit.ly/1a1meFC）输出结果。

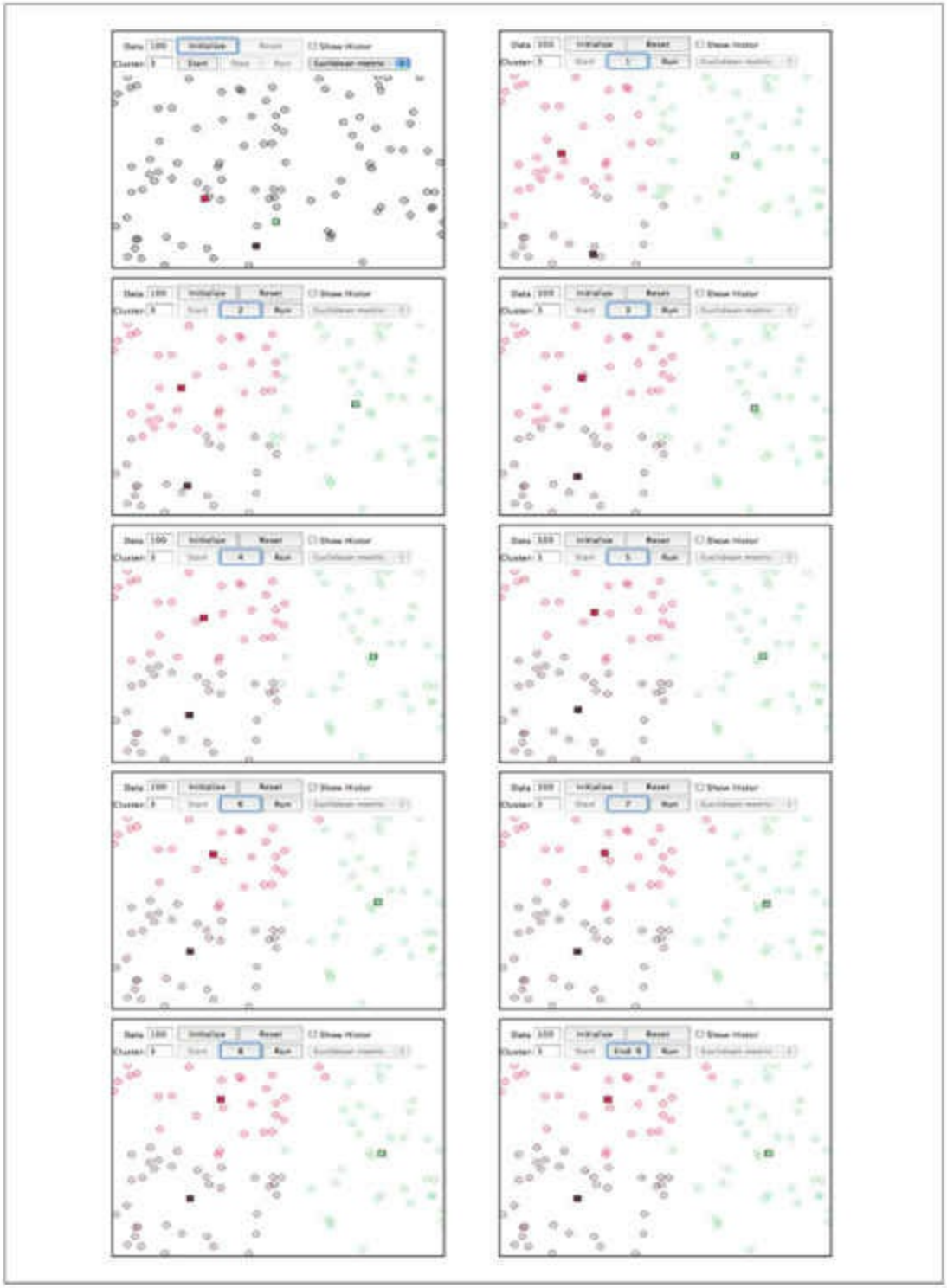


图3-7：对100个点进行k-means聚类的过程，k值为3。注意在该算法前几次迭代中类别形成的速度很快，剩下的迭代只会影响每类边界附近的

数据点

3.3.4.4 用Google地图将地理位置上的群组可视化

观察k-means是否有效的一个有用方式是通过聚类职业LinkedIn人脉将数据点画在二维空间中来可视化。通过可视化除了直观获得你联系人的分布情况以及记录其模式或异常现象外，你也可以用你的联系人、你联系人的不同的雇主或联系人主要居住的不同的都市区域来分析这些聚出的类别。这三种方式可能会分别得出出于不同的目的有用结果。

记得通过LinkedIn API能够获得描述大都会地区的位置信息，如“Greater Nashville Area”，我们将把这些位置信息地理编码为坐标，并存为特定的格式（如KML格式）（http://bit.ly/1a1meWb），通过如Google地球这样的工具能够绘制出来以提供交互性的用户体验。

注意：出于可视化的目的，Google新的地图引擎也提供了多种上传数据（http://bit.ly/1a1mep1）的方法。

为了将你的LinkedIn通讯录转换为像KML这样的格式，你必须做的一些基本的事情包括分析你的每个联系人档案中的地理位置，并为如Google地球这样的可视化工作构建KML文件。示例3-9示范了如何地理编码档案信息并提供收集我们所需要数据的基础工作。cluster包中的KMeansClustering类可以为我们进行聚类计算，这样我们剩下的工作就是处理数据和将聚类结果存为KML格式，使用XML工具这个过程也是

相对固定的。

跟示例3-12中一样，涉及可视化聚类结果的大部分工作都是围绕着数据处理的。最有趣的细节也都在KMeansClustering的getclusters的函数调用中实现，示例3-13的k-means聚类也是一样的。该方法的过程是先将通讯录的位置分组，然后将它们聚类，随后计算聚类的结果的中心。图3-8是运行示例3-13代码的结果。

注意：示例3-13中提供createKML函数的LinkedIn_kml_utility是一个固定的模式，为简洁明了，其过程我们在这里省略了，但是它包含在本章的IPython Notebook样例中。

示例3-13：根据通讯录的位置聚类LinkedIn的职业人脉，并输出KML格式以借助Google地球将其可视化

```
   import os
import sys
import json
from urllib2 import HTTPError
from geopy import geocoders
from cluster import KMeansClustering, centroid
# A helper function to munge data and build up an XML tree.
# It references some code tucked away in another directory, so we have to
# add that directory to the PYTHONPATH for it to be picked up.
sys.path.append(os.path.join(os.getcwd(), "resources", "ch03-linkedin"))
from linkedin__kml_utility import createKML
# XXX: Try different values for K to see the difference in clusters that emerge
K = 3
# XXX: Get an API key and pass it in here. See https://www.bingmapsportal.com.
GEO_API_KEY = ''
g = geocoders.Bing(GEO_API_KEY)
# Load this data from where you've previously stored it
CONNECTIONS_DATA = 'resources/ch03-linkedin/linkedin_connections.json'
OUT_FILE = "resources/ch03-linkedin/viz/linkedin_clusters_kmeans.kml"
# Open up your saved connections with extended profile information
# or fetch them again from linkedin if you prefer
connections = json.loads(open(CONNECTIONS_DATA).read())['values']
locations = [c['location']['name'] for c in connections if c.has_key('location')]
```

```
# Some basic transforms may be necessary for geocoding services to function properly
# Here are a couple that seem to help.
transforms = [('Greater ', ''), (' Area', '')]
# Step 1 - Tally the frequency of each location
coords_freqs = {}
for location in locations:
    if not c.has_key('location'): continue
    # Avoid unnecessary I/O and geo requests by building up a cache
    if coords_freqs.has_key(location):
        coords_freqs[location][1] += 1
        continue
    transformed_location = location
    for transform in transforms:
        transformed_location = transformed_location.replace(*transform)
        # Handle potential I/O errors with a retry pattern...
        while True:
            num_errors = 0
            try:
                results = g.geocode(transformed_location, exactly_one=False)
                break
            except HTTPError, e:
                num_errors += 1
                if num_errors >= 3:
                    sys.exit()
                print >> sys.stderr, e
                print >> sys.stderr, 'Encountered an urllib2 error. Trying again...'
        for result in results:
            # Each result is of the form ("Description", (X,Y))
            coords_freqs[location] = [result[1], 1]
            break # Disambiguation strategy is "pick first"
# Step 2 - Build up data structure for converting locations to KML
# Here, you could optionally segment locations by continent or country
# so as to avoid potentially finding a mean in the middle of the ocean.
# The k-means algorithm will expect distinct points for each contact, so
# build out an expanded list to pass it.
expanded_coords = []
for label in coords_freqs:
    # Flip lat/lon for Google Earth
    ((lat, lon), f) = coords_freqs[label]
    expanded_coords.append((label, [(lon, lat)] * f))
# No need to clutter the map with unnecessary placemarks...
kml_items = [{'label': label, 'coords': '%s,%s' % coords[0]} for (label,
            coords) in expanded_coords]
# It would also be helpful to include names of your contacts on the map
for item in kml_items:
    item['contacts'] = '\n'.join(['%s %s.' % (c['firstName'], c['lastName'])
        for c in connections if c.has_key('location') and
                                c['location']['name'] == item['label']])
# Step 3 - Cluster locations and extend the KML data structure with centroids
cl = KMeansClustering([coords for (label, coords_list) in expanded_coords
                       for coords in coords_list])
centroids = [{'label': 'CENTROID', 'coords': '%s,%s' % centroid(c)} for c in
             cl.getclusters(K)]
kml_items.extend(centroids)
# Step 4 - Create the final KML output and write it to a file
kml = createKML(kml_items)
f = open(OUT_FILE, 'w')
f.write(kml)
f.close()
print 'Data written to ' + OUT
```

简单地将你的人脉可视化就会有新的发现，但是计算你职业人脉的地理中心也能获得一些有趣的可能性。例如，你可能想要计算一系列地区研讨会或会议的候选位置。或者，如果你从事咨询业务并且出差行程很紧张，你可能会想寻找一些好的地点来租像家一样温馨的房子。或者你可能想根据他们的职位职责在地图上标出你人脉中的职业人员，或者根据他们的职位名称和经验标出他们可能适合的社会经济体。通过可视化你的职业人脉的位置数据，除了获得更多选择外，地理聚类也会获得许多其他的可能性，例如提供连锁管理和旅行推销员（http://bit.ly/1a1mhkF）类的问题，这些问题都涉及不同位置间旅行或运输物资，因而需要将支出最小化。

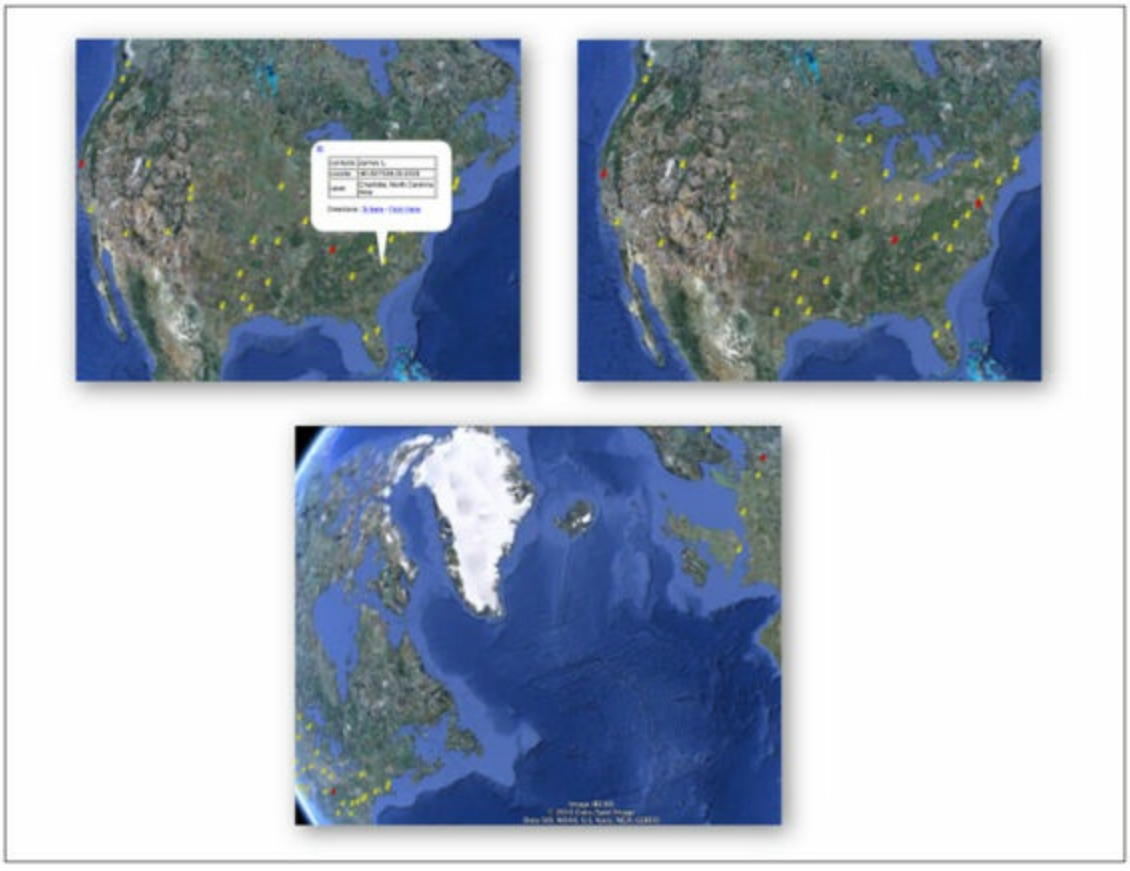


图3-8：从左上角的图顺时针方向介绍：1）按位置聚类通讯录，这样你就会容易看到哪个人生活或工作在哪个城市；2）通过k-means的计算找到3个聚类的中心；3）当想要找到理想的会议地点时，不要忘记聚类是可以跨国家甚至是跨州的

[1] 不考虑技术的微小差别，它通常也被称作近似匹配、模糊匹配或冗余消除等。

[2] 如果你认为这就开始复杂了，考虑下Dun & Bradstreet（http://bit.ly/1a1m4Om）在公司信息的“Who’s Who”上付出的工作量，他们参与了维护全世界的工商名录的挑战，该名录可以识别全世界跨越不同语言的公司。

3.4 本章小结

本章涉及了一些重要的基础知识。介绍了聚类的基本概念并展示了一些将其应用到LinkedIn上职业人脉的数据中的方法。就核心内容而言，本章毫无疑问比前面章节的内容更高级，因为本章开始处理一些常见的问题，如（多少有些）杂乱的数据的标准化、对标准化数据相似度的计算以及常见数据挖掘技术计算效率的考虑。尽管通过一遍阅读就学会所有的内容可能有些难，但千万不要气馁。本章内容的一些细节可能需要多阅读几遍才能消化理解。

同时也记住，使用聚类技术的知识不需要对其背后的理论有深刻的理解。一般来说，尽管你应该尽力地理解基础知识来加强你在挖掘社交网络时使用的技术。和其他章一样，我们仅仅接触到了冰山的一角；对LinkedIn的数据还有许多其他有趣的事可以做，很多方法只会涉及基本的频率分析而不要求进行聚类操作。也就是说，你现在已经有了处理数据的强大工具。

注意：本章和其他章的源码在GitHub（http://bit.ly/1a1kNqy）上都有，并以方便的IPython Notebook格式存储，强烈建议你打开网页浏览器来试试这些内容。

3.5 推荐练习

·花些时间来探索你拥有的扩展档案信息。试着将人们的工作地点和上学地点建立联系会很有趣，或者分析下人们是否倾向于迁入或迁出某地。

·试着用D3的其他可视化方法，如choropleth map（http://bit.ly/1a1mg0a），将你的职业人脉可视化。

·阅读新的令人兴奋的GeoJSON规范（http://bit.ly/1a1mggF），学习如何通过产生GeoJSON数据，在GitHub上轻松地创建交互式的可视化。试着将这个技术应用到你的职业人脉中来替代使用Google地球。

·用LinkedIn Labs InMaps（http://bit.ly/1a1mgx2）将你的LinkedIn人脉可视化。它用一些额外的信息来创建你的职业网络的图示，这些额外的信息不是直接通过API获得的，它会生成令人侧目的可视化效果。

·看一下geodict（http://bit.ly/1a1mgxd），在数据科学工具包（Data Science Toolkib）（http://bit.ly/1a1mgNK）中有一些其他的地理属性。你能从随机的文章中提取出位置信息并用有意义的方式来将它们可视化以获得数据中存在信息而不必进行通篇阅读么？

·挖掘Twitter或Facebook档案中的地理信息并用有意义的方式将其可

视化。推文和Facebook帖子通常包括地理编码，这是它们结构化元数据的一部分。

·LinkedIn API提供了检索联系人的Twitter接口的方法。在你的LinkedIn联系人中多少人有和他们职业档案相关联的Twitter账户？多少账户是活跃的？从潜在的雇主角度来看，他们网上体现的Twitter个性的职业程度有多少？

·将本章的聚类技术应用到推文上。假定你有一个用户的推文，能够提取有用的tweet信息、定义有意义的相似度并以有意义的方式将推文聚类么？

·将本章的聚类技术应用到Facebook数据中，如喜好或帖子。假定你收集了一个朋友的Facebook的“赞”数据，你能定义有意义的相似度计算并将喜好进行有意义的聚类么？假定你有你所有朋友的所有喜好，你能够将这些“赞”（或你的朋友）进行有意义的聚类么？

3.6 在线资源

下面是本章链接清单，或许对回顾本章内容有所帮助：

·Bing地图接口（http://bit.ly/1a1m5lq）

·中心（http://bit.ly/1a1mbcW）

·D3.js样例库（http://bit.ly/1a1lMal）

·数据科学工具包（http://bit.ly/1a1lmghk）

·树状图（http://bit.ly/1a1md4B）

·输出LinkedIn联系人（http://linkd.in/1a1m4）

·geopy的GitHub代码库（http://bit.ly/1a1m7Ka）

·锁孔标记语言（KML）（http://bit.ly/1a1meWb）

·编辑距离（http://bit.ly/1a1mcNO）

·LinkedIn API速率流上限（http://linkd.in/1a1m2WP）

·LinkedIn API字段选择器语法
（http://developer.LinkedIn.com/documents/understanding-field-selectors）

·LinkedIn开发者（http://linkd.in/1a1lZuj）

·LinkedIn InMaps（http://bit.ly/1a1mgx2）

·在GitHub上对GeoJSON文件进行转换（http://bit.ly/1a1mp3J）

·python-LinkedIn在GitHub上的代码库（http://bit.ly/1a1m2Gk）

·旅行推销员问题（http://bit.ly/1a1mhkF）

·聚类算法教程（http://bit.ly/1a1mbtp）

第4章　挖掘Google+：计算文档相似度、提取搭配等

本章介绍文本挖掘中的一些基本概念[1]，并且本章是这本书的一个转折点。我们在本书的开始部分对Twitter数据进行了一些基本的频率分析，随后我们对LinkedIn档案中更杂乱的数据进行了更复杂些的聚类分析，本章将会通过介绍像TF-IDF、余弦相似度和搭配提取这样的信息检索（Information Retrieval，IR）理论基础来对文档中的文本信息进行分析。相应地，本章的内容会比前面的章节稍微复杂些，仔细阅读前面的章节可能会对本章的学习大有帮助。

Google+（http://bit.ly/1a1mqES）是本章数据的主要来源，因为它本质上具有社交特性。它包含的内容通常以更长的文本内容（note）来进行记录，有点像博客文章，它现在是社交网络中已建立的主要产品。不难发现，在一些情况下，Google+的“动态”（activity）填补了Twitter和博客之间的一些空白。通过Google+多样而强大的API，前面章节的概念也可以同样应用到Google+的数据中，尽管我们选择将这些操作的样例（大部分）留给积极主动的读者作为练习。

只要有可能，我们不会白费力气去做重复的工作或从头实现一些分析工具。但是当遇到一些对理解文本挖掘来说必不可少的基本主题时，

我们会深入分析。自然语言工具箱（NLTK）包含一些强大的技术，可以查看第3章的相关内容；它提供的许多工具我们都会在本章用到。它丰富的API初看起来会很复杂，但是不用担心：尽管文本分析是个十分多样且复杂的研究领域，但是有很多强大的基本原理可以帮助你学习，而不需要再花费大量的精力进行研究。本章及后面章节的目的就是让你学会这些基本原理。（对NLTK的详细介绍不在本书的范畴，但是你可以在NLTK的网站（http://bit.ly/1a1mtAk）查看“用Python实现自然语言处理：用自然语言工具箱来分析文本[O’Reilly]”的详细内容。）

注意：在http://bit.ly/MiningTheSocialWeb2E上可以找到本章（及所有其他章节）最新修订bug的源代码。同时也要利用好本书的虚拟机，如附录A中描述的，以尽可能地享用样例代码。

[1] 本书不对一些常用短语做区别， 如文本挖掘（text mining） 、非结构化数据分析（Unstructured Data Analytics，UDA）或信息检索，我们只简单地把它们视为同一概念。

4.1 概述

本章我们将用Google+开启分析人类语言数据的旅程。本章你将会学到：

·Google+API以及发送API请求；

·词频-逆文档频率（Term Frequency-Inverse Document Frequency，TF-IDF），这一分析文档中单词的基本技术；

·如何将NLTK应用到理解人类语言问题中；

·如何将余弦相似度应用到按关键词查询文档等常见问题中；

·如何通过检测搭配模式从人类语言数据中提取有用的词组。

4.2 探索Google+API

只要拥有Gmail账号，你就可以创建一个Google+账号，并开始和朋友们沟通。从产品的角度来看，Google+发展得非常迅速，并且运用了一些社交网络平台（如Twitter和Facebook）最有吸引力的特征，来形成它自己的一系列功能。正如本书中介绍的其他社交网站一样，对Google+全面的介绍并不在本章范畴之中，但是你可以很容易地在网上阅读到这类内容。最佳的学习方式是创建账号并花费一些时间来探索。大部分情况下，如果有用户接口的特性，就会为你提供API来利用这些特性。可以这样说，Google+已经利用了现存社交网络中的可靠特性，如用话题标签标注内容和根据可定制的隐私设置维护一个配置文件；除此之外，还包括其他一些新颖的功能，例如叫做“圈子”（circle）的新鲜内容分享功能，称为“环聊”（hangouts）的视频交流功能以及许多和其他Google服务（如Gmail联系人）整合在一起的功能。借用Google+API（http://bit.ly/1a1mtR0）的说法，社交互动按照人员、动态、评论和生活片段来组织。

在线API文档一直都是权威的指导来源，但是一个较为简洁的概述可能有助于你思考Google+与其他平台（如Twitter或者Facebook）的区别：

人员（http://bit.ly/1a1mu7B）

人员（People）是Google+的用户。从编程的角度上看，你可以通过使用搜索类型的API来发现用户。如果是“名人”[1]，也可以通过个性网址（personalized URL）（http://bit.ly/1a1muEo）进行查找，或者你还可以在浏览器的网址中分离出他们的Google+ID，使用这个内容探索他们的个人信息。在本章的后面的内容中，我们将使用搜索类型的API在Google+上寻找用户。

动态（http://bit.ly/1a1muV4）

动态（Activities）是人们在Google+上所做的事情。动态本质上是一条记录，它可长可短，这完全取决作者的喜好：它可以和博客一样长，也可以没有任何文本意义（例如仅仅用来分享链接或多媒体内容）。给定一个Google+用户，你可以很容易地获取到他的一系列活动，我们将在本章后面介绍这些。像推文一样，一条动态包含很多有趣的元数据，例如动态被重新分享的次数等。

评论（http://bit.ly/1a1mvrX）

发表评论（Comments）是Google+用户彼此进行交互的方式。对Google+上的评论做简单的统计分析会非常的有趣，并且可能会揭示一个人的社交圈的大量信息或内容的传播特点。例如，哪些用户最频繁地对动态发表评论？哪些动态得到的评论最多（为什么）？

生活片段（http://bit.ly/1a1mvIx）

生活片段（Moments）是相对而言最近才增加到Google+上的，并且代表了一种捕捉用户和Google+应用之间交互的方式。生活片段和Facebook的社交图谱（social graph stories）（http://bit.ly/1a1my7k）类似，它们是用来获取或创造用户和应用交互的机会，这些应用可以展示在时间轴上。例如，如果你想在某个应用中购买商品、上传图片或者看YouTube视频，这些都会被应用程序捕捉为生活片段（按照时间顺序做的一些事情）并且显示在你的行为历史记录中，或者以一系列动态的方式分享给朋友。

本章我们将关注获取和分析Google+动态数据，这些数据是文本形式的，并且与你之前遇到的推文、博客或者Facebook状态更新表达相似的含义。换句话说，我们将尝试分析人类语言数据。如果你还没有注册Google+，那么注册一个账号是很有必要的，你也可以花些时间熟悉自己的Google+个人资料。在Google+上查找特定用户最容易的一个方法就是在http://plus.google.com上搜索，并且从这里开始探索这个平台。

当你在探索Google+的时候，请记住它有一些独特的功能。如果直接和其他社交网络的特性比较，会觉得有些奇怪。如果你希望有个直观的比较，你会在这找到惊喜。它和Twitter类似，因为它提供了一种“关注”（following）模式。你可以把某些人添加到自己的某个Google+圈子中，关注他们发生的事情而不需要通过他们的验证。再者，Google+平

台还提供了丰富的与其他Google网络特性进行集成的功能、支持强大的视频会议功能，并且它还有和Facebook相似的API，帮助用户分享内容和交互。闲话不多说，让我们开始探索这个API并用来挖掘一些数据。

4.2.1 发起Google+API请求

从软件开发的角度来看，和其他的社交网络一样，Google+利用OAuth启用你将创建的代表用户访问数据的应用程序，因此你需要注册一个应用程序从而获得正确的访问Google+平台的凭证。Google API控制台（http://bit.ly/1a1mwMP）提供了一种注册应用程序（在Google API控制台中叫工程）的方式来获得OAuth凭证，但是同时也获得了API密匙，这个密匙可以用来进行简单的API访问。这个API密匙是我们在本章通过程序访问Google+平台和其他几乎所有Google服务所需要的。

一旦你已经创建了一个应用程序，你还需要一个额外的步骤特别允许它能够使用Google+。图4-1提供了一个Google+API控制台的截图，同时也展示了当允许应用程序进行Google+API访问后的界面。

你可以使用命令pip install google-api-python-client安装一个叫google-api-python-client的Python包以使用Google的API。这是一种标准的基于Python访问Google+数据的方式。在线的google-api-python-client文档（http://bit.ly/1a1mzYI）对你熟悉它的功能并没有太大的帮助，但

是总的来说，你将只是用Python的包将一些官方Google+API文档中的参数填充到可预测的访问模式中。一旦你完成一些练习，你就会发现这个过程是非常简单的。

注意：不要忘记，在学习阶段，pydoc对于你在终端上收集关于包、类或方法等信息是很有帮助的。在标准的Python解释器中，help函数也很好用。回想一下，IPython中在方法名后面加上？是显示该方法文档字符串（docstrings）的快捷方法。

作为第一次练习，我们考虑一个Google+上的寻人问题。和任何其他的社交网络API一样，Google+API提供了搜索的方式。尤其是，我们关注People：search API（http://bit.ly/1a1mzrQ）。示例4-1演示了如何用Google+API搜索一个人。由于Tim O’Reilly是个知名人士并有一个活跃且受关注的Google+账户，因此我们先来搜索他。

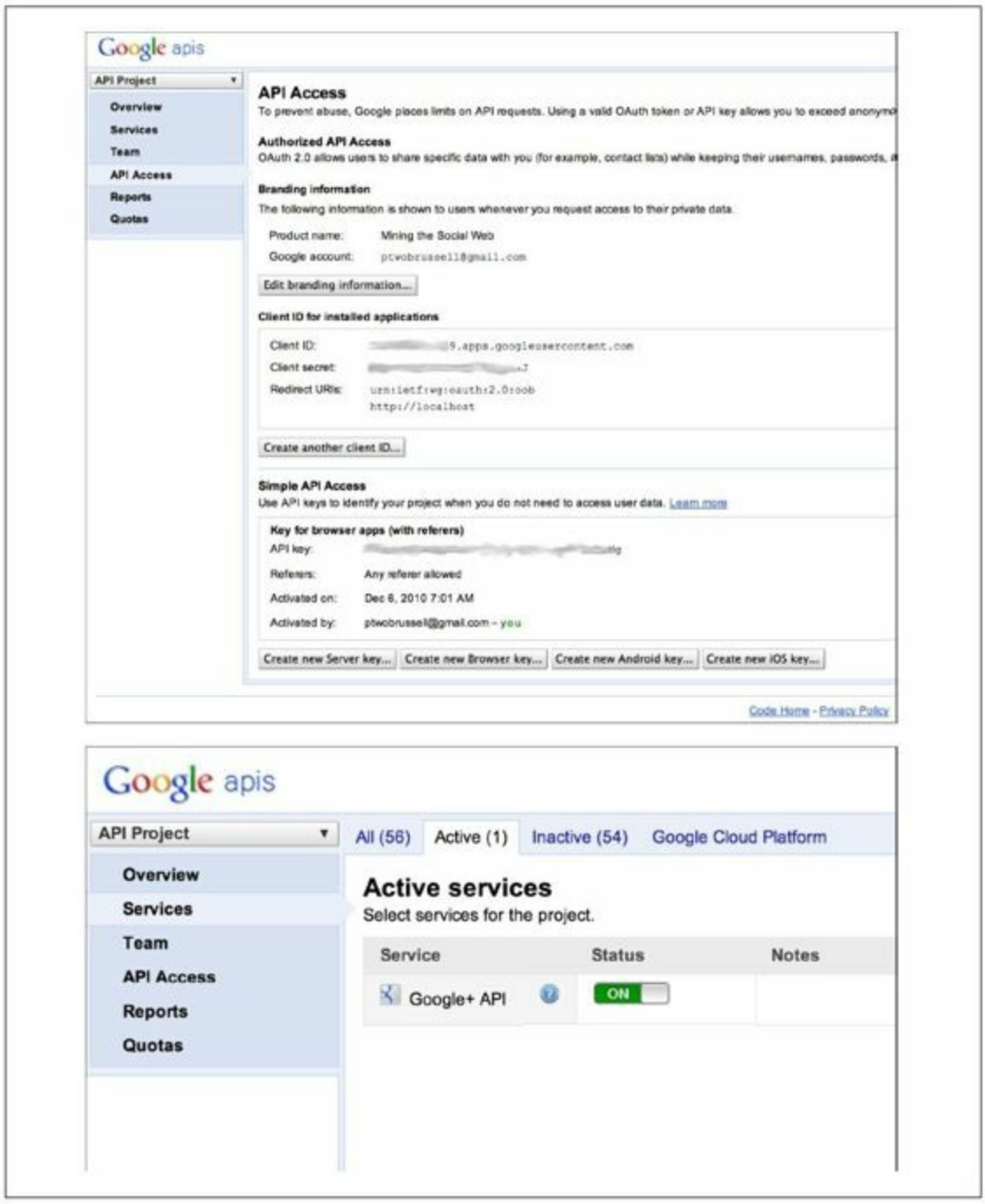


图4-1：使用Google API控制台注册一个应用程序，从而可以使用API访问Google服务，别忘了将Google+API访问作为一个服务选项启用

使用Python客户端常用的基本模式：用你的API密钥创建一个参数

化的服务实例来访问Google+，然后可以将其为特定的平台服务进行初始化。这里我们通过调用service.people（）来建立与人员API的连接，然后将一些其他的API操作加入其中，这些操作可以通过查阅在线API文档获得。随后我们查询“动态”数据，你们将会看到它也使用相同的基本模式。

示例4-1：使用Google+API寻人

```
   import httplib2
import json
import apiclient.discovery # pip install google-api-python-client
# XXX: Enter any person's name
Q = "Tim O’Reilly"
# XXX: Enter in your API key from https://code.google.com/apis/console
API_KEY = ''
service = apiclient.discovery.build('plus', 'v1', http=httplib2.Http(),
                                    developerKey=API_KEY)
people_feed = service.people().search(query=Q).execute()
print json.dumps(people_feed['items'], indent=1)
```

下面是搜索Tim O’Reilly得到的示例结果：

```
   [
 {
  "kind": "plus#person",
  "displayName": "Tim O’Reilly",
  "url": "https://plus.google.com/+TimOReilly",
  "image": {
   "url": "https://lh4.googleusercontent.com/-J8..."
  },
  "etag": "\"WIBkkymG3C8dXBjiaEVMpCLNTTs/wwgOCMn...\"",
  "id": "107033731246200681024",
  "objectType": "person"
 },
 {
  "kind": "plus#person",
  "displayName": "Tim O’Reilly",
  "url": "https://plus.google.com/11566571170551...",
  "image": {
   "url": "https://lh3.googleusercontent.com/-yka..."
 },
 "etag": "\"WIBkkymG3C8dXBjiaEVMpCLNTTs/0z-EwRK7...\"",
 "id": "115665711705516993369",
 "objectType": "person"
```

```
 },
 ...
]
```

结果确实返回了名为Tim O’Reilly的人员的列表，但是我们如何知道哪个是我们要找的那个在科技界享有盛名的Tim O’Reilly呢？一种方法是对于结果中的每一条结果查看它的个人资料或“动态”信息，尝试人工区分它们。另外一个方法是将结果中包含的头像显示出来，在IPython Notebook中将头像作为图片显示出来是很容易实现的。示例4-2演示了如何对每个查询的结果显示其头像和相应的ID，它通过生成HTML并把头像的显示内嵌到notebook中实现的。

示例4-2：在IPython Notebook中显示Google+头像为我们提供了一种快速分辨搜索结果，发现我们要找的人的方式

```
   from IPython.core.display import HTML
html = []
for p in people_feed['items']:
    html += ['<p><img src="%s" /> %s: %s</p>' % \
             (p['image']['url'], p['id'], p['displayName'])]
HTML(''.join(html))
```

示例结果如图4-2所示，它为我们搜索O’Reilly Media公司的那个Tim O’Reilly提供了快捷的方法。

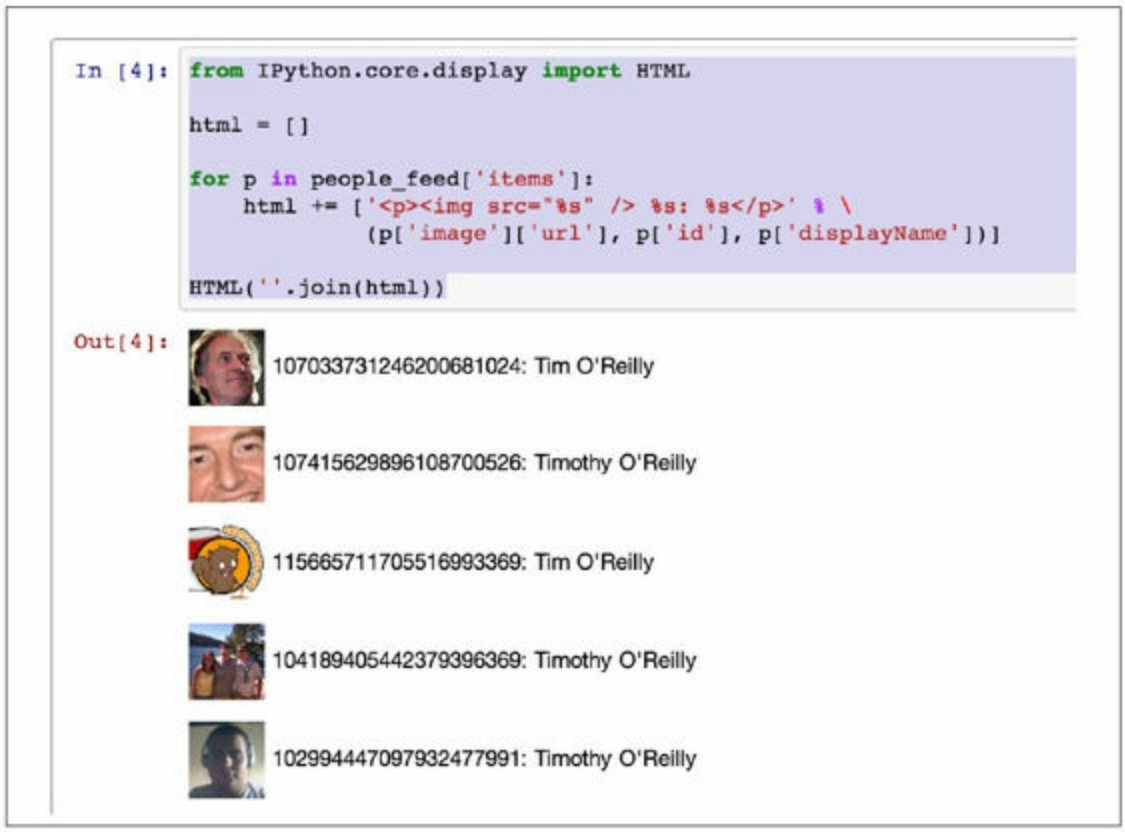


图4-2：将Google+的头像以图片形式显示，我们可以快速的扫描搜索结果，从而识别我们要找的人

尽管我们用人员API可以做许多事情，然而本章我们关注的是分析账号的文本内容。因此让我们将注意力转向如何获取与账号相关的“动态”信息。你将会发现，Google+“动态”是Google+内容的关键，它包含与账号相关的各种各样的丰富内容，并为平台的其他对象（如评论）提供了逻辑支点。为获得一些“动态”，我们需要修改在搜索人员时使用的设计模式，代码如示例4-3所示。

示例4-3：获取特定Google+用户近期的“动态”

```
   import httplib2
import json
import apiclient.discovery
USER_ID = '107033731246200681024' # Tim O’Reilly
# XXX: Re-enter your API_KEY from https://code.google.com/apis/console
# if not currently set
# API_KEY = ''
service = apiclient.discovery.build('plus', 'v1', http=httplib2.Http(),
                                    developerKey=API_KEY)
activity_feed = service.activities().list(
  userId=USER_ID,
  collection='public',
  maxResults='100' # Max allowed per API
).execute()
print json.dumps(activity_feed, indent=1)
```

示例结果中的第一项（activity_feed['items'][0]）是Google+活动的基本特征。

```
   {
 "kind": "plus#activity",
 "provider": {
  "title": "Google+"
 },
 "title": "This is the best piece about privacy that I've read in a ...",
 "url": "https://plus.google.com/107033731246200681024/posts/78UeZ1jdRsQ",
 "object": {
  "resharers": {
   "totalItems": 191,
   "selfLink": "https://www.googleapis.com/plus/v1/activities/z125xvy..."
  },
  "attachments": [
   {
    "content": "Many governments (including our own, here in the US) ...",
    "url": "http://www.zdziarski.com/blog/?p=2155",
    "displayName": "On Expectation of Privacy | Jonathan Zdziarski's Domain",
    "objectType": "article"
   }
  ],
  "url": "https://plus.google.com/107033731246200681024/posts/78UeZ1jdRsQ",
  "content": "This is the best piece about privacy that I&#39;ve read ...",
  "plusoners": {
   "totalItems": 356,
   "selfLink": "https://www.googleapis.com/plus/v1/activities/z125xvyid..."
  },
  "replies": {
  "totalItems": 48,
   "selfLink": "https://www.googleapis.com/plus/v1/activities/z125xvyid..."
   },
   "objectType": "note"
  },
   "updated": "2013-04-25T14:46:16.908Z",
   actor": {
   "url": "https://plus.google.com/107033731246200681024",
   "image": {
```

```
   "url": "https://lh4.googleusercontent.com/-J8nmMwIhpiA/AAAAAAAAAAI/A..."
  },
  "displayName": "Tim O扑eilly",
  "id": "107033731246200681024"
 },
 "access": {
  "items": [
   {
    "type": "public"
   }
  ],
  "kind": "plus#acl",
  "description": "Public"
 },
 "verb": "post",
 "etag": "\"WIBkkymG3C8dXBjiaEVMpCLNTTs/d-ppAzuVZpXrW_YeLXc5ctstsCM\"",
 "published": "2013-04-25T14:46:16.908Z",
 "id": "z125xvyidpqjdtol423gcxizetybvpydh"
}
```

每一个“动态”对象都遵循一个三元组的形式（actor，verb，object）。在这个帖子中，（Tim O’Reilly，post，note）告诉我们结果中的这一项是一条note，它本质上是使用文本内容进行的状态更新。更进一步地看这个结果，会发现该内容是Tim O’Reilly十分关注的，正如其标题所说的一样“这是我这么长时间读到关于隐私的最好内容”，请注意这条note是活跃的，我们可以从分享和评论的个数中看出来。

如果你重新仔细地检视输出结果，你会发现“动态”的content字段包含HTML标记，比如出现了HTML实体I&#39；ve。总地来说，你应该假定，Google+“动态”的文本数据包括一些基本的标注，例如标签
和用单引号转义的HTML实体，因此最好的做法是进行额外过滤，将它们清理干净。示例4-4提供了如何通过引入cleanHtml从一条记录的content字段中提取出纯文本内容的例子。它利用了NLTK提供的clean_html函数以及另一个非常方便的用来处理HTML的包

BeautifulSoup，它将HTML内容转换为纯文本。如果你还没有使用过BeautifulSoup，一旦你将它放到你的工具箱中，你就再也离不开它了。即使HTML无效并且违反了标准或者存在其他可预期的异常（web数据的形式s），它也能合理的进行处理。这个包可以合理地处理HTML，即使处理的内容是无效的并且违反了标准或者有其他的异常。如果你还没有使用过它，可以通过命令pip install nltk beautifulsoup4来安装。

示例4-4：通过去除HTML标签清理Google+content中的HTML，并将HTML实体还原为纯文本表示

```
   from nltk import clean_html
from BeautifulSoup import BeautifulStoneSoup
# clean_html removes tags and
# BeautifulStoneSoup converts HTML entities
def cleanHtml(html):
  if html == "": return ""
  return BeautifulStoneSoup(clean_html(html),
          convertEntities=BeautifulStoneSoup.HTML_ENTITIES).contents[0]
print activity_feed['items'][0]['object']['content']
print
print cleanHtml(activity_feed['items'][0]['object']['content'])
```

一旦使用cleanHtml进行清理，记录的content得到的输出会是纯文本，这样我们就不再需要额外关注记录内容中的噪声了，可以直接对数据进行处理。正如我们将要在本章和后续关于文本挖掘的章节学到的一样，减少文本内容中的噪声是提高准确度的关键方法。下面是记录内容在处理前后的效果。

这里是activity_feed['items'][0]['object']['content']的原始内容：

```
   This is the best piece about privacy that I&#39;ve read in a long time!
If it doesn&#39;t change how you think about the privacy issue, I&#39;ll be
surprised. It opens:<br /><br />&quot;Many governments (including our own,
here in the US) would have its citizens believe that privacy is a switch (that
is, you either reasonably expect it, or you don't). This has been demonstrated
in many legal tests, and abused in many circumstances ranging from spying
on electronic mail, to drones in our airspace monitoring the movements of
private citizens. But privacy doesn't work like a switch  at least it shouldn't
for a country that recognizes that privacy is an inherent right. In fact,
privacy, like other components to security, works in layers...&quot;<br /><br />
Please read!
```

下面是通过cleanHtml（activity_feed['items'][0]['object']['content']）处理后的内容：

```
   This is the best piece about privacy that I've read in a long time! If it
doesn't change how you think about the privacy issue, I'll be surprised. It
opens: "Many governments (including our own, here in the US) would have its
citizens believe that privacy is a switch (that is, you either reasonably expect it,
or you don't). This has been demonstrated in many legal tests, and abused
in many circumstances ranging from spying on electronic mail, to drones in our
airspace monitoring the movements of private citizens. But privacy doesn't work like
a switch  at least it shouldn't for a country that recognizes that privacy is an
inherent right. In fact, privacy, like other components to security, works in
layers..." Please read!
```

从Google+中得到纯净文本的能力是本章其余文本挖掘练习的基础，但是在我们将注意力转移到其他地方之前，你可能会发现一种用于获取多页内容的模式是非常有用的。

虽然之前的例子获取了100条“动态”（这是一次查询结果的最大数量），但是很有可能你想对一个“动态”源（activity feed）进行迭代并且获取比每一页最大“动态”个数更多的内容。Google+API概述（http://bit.ly/1a1mAfm）中强调了分页的模式，并且Python客户端封装器会负责处理大部分的这些困扰。

示例4-5显示了当“动态”是记录类型并且内容有意义时如何获取多页的“动态”，并且从中提取出文本信息。

示例4-5：对多页的Google+“动态”进行循环，并从记录中提取出纯净的文本

```
   import os
import httplib2
import json
import apiclient.discovery
from BeautifulSoup import BeautifulStoneSoup
from nltk import clean_html
USER_ID = '107033731246200681024' # Tim O扑eilly
# XXX: Re-enter your API_KEY from https://code.google.com/apis/console
# if not currently set
# API_KEY = ''
MAX_RESULTS = 200 # Will require multiple requests
def cleanHtml(html):
   if html == "": return ""
   return BeautifulStoneSoup(clean_html(html),
            convertEntities=BeautifulStoneSoup.HTML_ENTITIES).contents[0]
service = apiclient.discovery.build('plus', 'v1', http=httplib2.Http(),
                                    developerKey=API_KEY)
activity_feed = service.activities().list(
  userId=USER_ID,
  collection='public',
  maxResults='100' # Max allowed per request
)
activity_results = []
while activity_feed != None and len(activity_results) < MAX_RESULTS:
  activities = activity_feed.execute()
  if 'items' in activities:
    for activity in activities['items']:
      if activity['object']['objectType'] == 'note' and \
         activity['object']['content'] != '':
         activity['title'] = cleanHtml(activity['title'])
         activity['object']['content'] = cleanHtml(activity['object']['content'])
         activity_results += [activity]
  # list_next requires the previous request and response objects
  activity_feed = service.activities().list_next(activity_feed, activities)
# Write the output to a file for convenience
f = open(os.path.join('resources', 'ch04-googleplus', USER_ID + '.json'), 'w')
f.write(json.dumps(activity_results, indent=1))
f.close()
print str(len(activity_results)), "activities written to", f.name
```

我们知道了如何探索Google+API以及如何从“动态”内容中获取一些

有趣的人类语言数据，现在让我们将注意力转到分析内容的问题上来。

[1] Google+在初期仅为知名人士提供个性网址，例如https://plus.google.com/+TimOReilly，目前这一功能正在推广到所有用户。除非你被Google选中或者具备了申请的资格，否则你将持续使用更长的用数字进行标识的网址，例如https://plus.google.com/107033731246200681024。

4.3 TF-IDF简介

为了获得对文本数据最深刻的理解，使用严格的方法来进行自然语言处理（NLP）是十分必要的，其中包括句子切分（sentence segmentation）、分词（tokenization）、单词组合（word chunking）和实体检测（entity detection）。但是先从信息检索（Information Retrieval，IR）理论来介绍一些基础知识也是很有帮助的。本章剩下的内容会介绍其中更基础的方面，包括TF-IDF，余弦相似度度量和一些搭配检测（collocation detection）背后的理论。第5章将作为这里的延续，提供对NLP更深的讨论。

注意：如果你想进一步研究IR理论，《信息检索简介》（Introduction to Information Retrieval）提供了这个领域你想知道的所有信息。该书的作者是Christopher Manning、Prabhakar Raghavan和Hinrich Schiltze，由剑桥大学出版社出版。该书可以在线获得（http://stanford.io/1a1mAvP）。

信息检索非常广泛，它横跨了多个学科。这里我们只讨论TF-IDF，它是从语料库（集合）中提取相关文档最基本的技术之一。TF-IDF的全称是词频-逆向文档频率，可以通过计算表示文档中词语相对重要性的归一化得分来查询语料库。

从数学的角度来说，TF-IDF表示为词频和逆向文档频率的乘积，即tf_idf=tf*idf，tf表示一个词语在具体文档中的重要性，idf表示一个词语在整个语料库中的重要性。将这两项相乘得到可以同时说明这两个因素的得分，并且在某些方面成为了每个主流搜索引擎不可缺少的一部分。为了更直观的了解TF-IDF是如何工作的，让我们来看看计算总分时涉及的所有运算。

4.3.1 词频

为了方便说明，假设你有一个包含三个样本文档的语料库，词语是按照空格简单划分得到的，如示例4-6所示。

示例4-6：本章后续内容用于演示用途的样本数据结构

```
   corpus = {
 'a' : "Mr. Green killed Colonel Mustard in the study with the candlestick. \
Mr. Green is not a very nice fellow.",
 'b' : "Professor Plum has a green plant in his study.",
 'c' : "Miss Scarlett watered Professor Plum's green plant while he was away \
from his office last week."
}
terms = {
 'a' : [ i.lower() for i in corpus['a'].split() ],
 'b' : [ i.lower() for i in corpus['b'].split() ],
 'c' : [ i.lower() for i in corpus['c'].split() ]
}
```

词频可以简单的表示为它在文档中出现的次数，但是更普遍的用法是通过考虑文本中词的总数对它进行归一化。这样总分就能表示文档长

度与词频的相对关系。例如，“green”（已经被标准化为小写）在corpus['a']中出现了两次，在corpus['b']中只出现一次，因此如果词频是唯一的评判标准的话，那么该词在corpus['a']中的得分会更高。然而，如果根据文档长度进行归一化，即便“green”更常出现在corpus['a']中，“green”在corpus['b']的词频得分（1/9）将会略高于corpus['a']（2/19），因为corpus['b']的长度相对较短。计算“Mr.Green”这类复合查询得分的常用技术是将文档中每个查询词的词频相加，返回按总体词频得分排序的文档。

上一段其实很好理解，例如，查询语料库每个文档中的“Mr.Green”会得到表4-1所示的归一化得分。

表4-1：“Mr.Green”的样本词频得分

文档	tf(mr.)	tf(green)	总计
corpus['a']	2/19	2/19	4/19 (0.2105)
corpus['b']	0	1/9	1/9 (0.1111)
corpus['c']	0	1/16	1/16 (0.0625)

对于这个例子，计算累计词频得分的方案是有效的并会返回corpus['a']，因为只有corpus['a']包含复合词“Mr.Green”。然而，这也出现了许多问题，因为词频得分模型把所有文档都看做一个无序的单词集。例如，即使“Green Mr.”或“Green Mr.Foo”这两个复合词都没有出现在样

例语句中，但查询他们将与查询“Mr.Green”返回相同的结果。另外，我们很容易想到，当词后面的标点符号没有正确处理和计算时没有考虑目标词周围的上下文时，词频排序技术会得到很差的结果。

对文档评分时，只考虑词频是常见的错误，因为它没有考虑到很多文档中都很常见的词，即停用词（stopword）[1]。也就是说，所有词都均等地加权，而没有考虑它们实际的重要性。例如，“the green plant”包含停用词“the”，它提高了corpus['a']的总词频得分，因为“the”和“green”都在文档中出现了两次。相反，“green”和“plant”在corpus['c']中只出现了一次。

最终得分会被分解为如表4-2所示的那样。其中corpus['a']的排名比corpus['c']更相关，即使是直觉上我们也会相信结果不应该是这样的（然而，幸运的是，corpus['b']的排名仍然是最高的）。

表4-2：“the green plant”的样本词频得分

文档	tf(the)	tf(green)	tf(plant)	总计
corpus['a']	2/19	2/19	0	4/19 (0.2105)
corpus['b']	0	1/9	1/9	2/9 (0.2222)
corpus['c']	0	1/16	1/16	1/8 (0.125)

4.3.2 逆文档频率

NLTK这样的工具包提供了停用词列表，可以用它来过滤“the”、“a”和“the”这类词。但是请记住，也有很多词不包含在最佳停用词列表中，这类情况在专业领域仍然是很常见的。逆文档频率提供的计算结果表示语料库的通用归一化度量。通常情况下，它通过考虑文档集合中出现了所考察常见词的文档数量，来计算该词在一个文档集合的出现次数。

对于该度量的直观理解是如果某个词在语料库中不常出现，那么它就会产生一个更大的值，这有助于解释我们刚才研究的停用词问题。例如，对样本文档语料库中“green”的查询返回的逆文档频率得分应该低于“candlestick”，因为每个文档中都有“green”，而“candlestick”只出现在一个文档。从数学角度来说，计算逆文档频率时唯一不同的是使用对数函数将结果压缩到某一范围内，因为我们通常会将它作为比例因子与词频相乘。正如你看到的一样，当输入值变大时，log函数的增长十分慢，这对输入进行了有效的压缩。

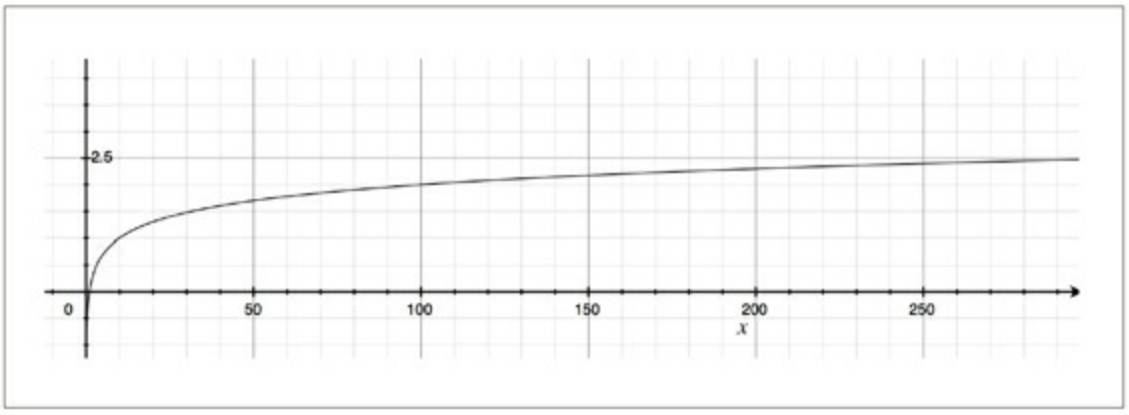


图4-3：对数函数将大范围的数值压缩到一个更窄空间——注意当x值增

加时y值增加的非常缓慢

表4-3展示了前面讲到的与词频相对应的逆文档频率。下一节中的例4-7展示了计算这些得分的源代码。同时，你可以认为词的IDF得分是文档的总数除以语料库中包含该词的文本数量的商的对数。鉴于词频的得分是基于每个文档计算的，看到这些表的同时，请记住词的IDF分数是基于整个语料库计算的。IDF意在对整个语料库常见单词进行归一化，期望这样做会有效果。

表4-3：出现在“mr.green”和“the green plant”中词的样例逆文档频率得分

idf(mr.)	idf(green)	idf(the)	idf(plant)
1+ log(3/1) = 2.0986	1 + log(3/3) = 1.0	1 + log(3/1) = 2.0986	1 + log(3/2) = 1.4055

4.3.3 TF-IDF

现在，我们回到原点并设计一种对多个词查询的得分进行计算的方法。这种得分可以表示文档中单词出现的频率，含有特定单词的文档长度，以及整个语料库中单词在各个文档中的整体独特性。我们可以将词频和逆文档频率相乘得到一个得分，即TF-IDF=TF*IDF。示例4-7是这个问题的简单实现，它有助于理解我们描述的概念。花一些时间来阅读它，随后我们将讨论一些样例查询。

示例4-7：在样本数据上计算TF-IDF

```
   from math import log
# XXX: Enter in a query term from the corpus variable
QUERY_TERMS = ['mr.', 'green']
def tf(term, doc, normalize=True):
    doc = doc.lower().split()
    if normalize:
        return doc.count(term.lower()) / float(len(doc))
    else:
        return doc.count(term.lower()) / 1.0
def idf(term, corpus):
    num_texts_with_term = len([True for text in corpus if term.lower()
                               in text.lower().split()])
    # tf-idf calc involves multiplying against a tf value less than 0, so it's
    # necessary to return a value greater than 1 for consistent scoring.
    # (Multiplying two values less than 1 returns a value less than each of
    # them.)
    try:
        return 1.0 + log(float(len(corpus)) / num_texts_with_term)
    except ZeroDivisionError:
        return 1.0
def tf_idf(term, doc, corpus):
    return tf(term, doc) * idf(term, corpus)
corpus = \
    {'a': 'Mr. Green killed Colonel Mustard in the study with the candlestick. \
Mr. Green is not a very nice fellow.',
    'b': 'Professor Plum has a green plant in his study.',
    'c': "Miss Scarlett watered Professor Plum's green plant while he was away \
from his office last week."}
for (k, v) in sorted(corpus.items()):
    print k, ':', v
print
# Score queries by calculating cumulative tf_idf score for each term in query
query_scores = {'a': 0, 'b': 0, 'c': 0}
for term in [t.lower() for t in QUERY_TERMS]:
    for doc in sorted(corpus):
        print 'TF(%s): %s' % (doc, term), tf(term, corpus[doc])
    print 'IDF: %s' % (term, ), idf(term, corpus.values())
    print
    for doc in sorted(corpus):
        score = tf_idf(term, corpus[doc], corpus.values())
        print 'TF-IDF(%s): %s' % (doc, term), score
        query_scores[doc] += score
    print
print "Overall TF-IDF scores for query '%s'" % (' '.join(QUERY_TERMS), )
for (doc, score) in sorted(query_scores.items()):
    print doc, score
```

样例输出如下：

```
   a : Mr. Green killed Colonel Mustard in the study...
b : Professor Plum has a green plant in his study.
c : Miss Scarlett watered Professor Plum's green...
TF(a): mr. 0.105263157895
TF(b): mr. 0.0
TF(c): mr. 0.0
IDF: mr. 2.09861228867
```

```
TF-IDF(a): mr. 0.220906556702
TF-IDF(b): mr. 0.0
TF-IDF(c): mr. 0.0
TF(a): green 0.105263157895
TF(b): green 0.111111111111
TF(c): green 0.0625
IDF: green 1.0
TF-IDF(a): green 0.105263157895
TF-IDF(b): green 0.111111111111
TF-IDF(c): green 0.0625
Overall TF-IDF scores for query 'mr. green'
a 0.326169714597
b 0.111111111111
c 0.0625
```

尽管我们是对很小规模的数据进行计算，但是该方法也同样适用于更大的数据集。表4-4是对如下三个样本查询（涉及4个不同的词）的程序输出的统一改写：

·“green”

·“Mr.green”

·“the green plant”

虽然词语的IDF计算是针对整个语料库的，我们还是在每个文档的TF值之后均显示了它们，这样就可以跳过若干行并把两个值相乘从而很容易对TF-IDF结果进行验证。而且，值得你花几分钟时间熟悉表中的数据，这样可以了解计算的具体过程了。鉴于不考虑文档中单词的邻近和顺序，当完成查询时你会发现TF-IDF是十分强大的。

表4-4：TF-IDF样本查询涉及的计算过程，它是由示例4-7计算的

文档	tf(mr.)	tf(green)	tf(the)	tf(plant)
corpus['a']	0.1053	0.1053	0.1053	0
corpus['b']	0	0.1111	0	0.1111
corpus['c']	0	0.0625	0	0.0625

idf(mr.)	idf(green)	idf(the)	idf(plant)
2.0986	1.0	2.099	1.4055

	tf-idf(mr.)	tf-idf(green)	tf-idf(the)	tf-idf(plant)
corpus['a']	0.1053*2.0986 = 0.2209	0.1053*1.0 = 0.1053	0.1053*2.099 = 0.2209	0*1.4055 = 0
corpus['b']	0*2.0986 = 0	0.1111*1.0 = 0.1111	0*2.099 = 0	0.1111*1.4055 = 0.1562
corpus['c']	0*2.0986 = 0	0.0625*1.0 = 0.0625	0*2.099 = 0	0.0625*1.4055 = 0.0878

与每个查询相同的结果也同样展示在表4-5中，TF-IDF值是在每个文档上进行累计得到的。

表4-5：由示例4-7计算的样本查询的总TF-IDF值（加粗部分分别是三个查询的最大值）

查询	corpus['a']	corpus['b']	corpus['c']
green	0.1053	0.1111	0.0625
Mr. Green	0.2209 + 0.1053 = 0.3262	0 + 0.1111 = 0.1111	0 + 0.0625 = 0.0625
the green plant	0.2209 + 0.1053 + 0 = 0.3262	0 + 0.1111 + 0.1562 = 0.2673	0 + 0.0625 + 0.0878 = 0.1503

从定性的角度来说，查询结果是非常合理的。corpus['b']文档对于“green”的查询是最好的，其次是corpus['a']。在这种情况下，决定因素是corpus['b']的长度比corpus['a']小得多：归一化的TF得分倾向于只出

现了一次“green”的corpus['b']，即使“green”在corpus['a']中出现了两次。由于“green”在三个文档中均有出现，因此IDF对计算结果不产生影响。

然而，一定要注意，如果在IDF计算“green”的结果中返回了0.0而不是1.0，正如一些实现中所做的那样，那么由于要在TF上乘以0，所以三个文档中“green”的TF-IDF得分都是0.0。根据不同的情况，IDF得分返回为0.0可能要比返回为1.0好。例如，如果有10万个文档，所有这些文档中都出现了“green”，你几乎可以肯定地认为它是停用词，而且应该完全去掉它对查询的影响。

对于查询“Mr.Green”，最好的是corpus['a']文档。然而，该文档对于查询“the green plant”也有最好的得分。我们可以考虑一下为什么对于该查询corpus['a']得到最高的分数，而不是第一眼看起来会得到高分的corpus['b']。

我们需要注意的最后一点，示例4-7中的示例实现将IDF得分在对数函数计算后额外加了1.0，这样是为了便于演示，因为我们处理的文档集过于简单了。不在计算中加上1.0，idf函数可能会返回小于1.0的值，这会导致在TF-IDF计算中要进行相乘的两个因子都是小于1.0的分数。这样两个因子的乘法结果比它们本身还要小，所以它其实是TF-IDF计算中容易被忽视的边缘情况。这里的调整是呼应了TF-IDF计算的初衷：希望将两项相乘之后，能够将相关度高的查询比不太相关的查询产生较大的TF-IDF得分。

[1] 停用词是那些频繁出现于文本却携带很少信息的词。常见的停用词包括a、an、the以及其他限定词。

4.4 用TF-IDF查询人类语言数据

让我们将之前章节中学习的理论应用到实际工作中。在这一部分，我们将正式的介绍NLTK，它是处理自然语言的强大工具包，我们将要用它来分析从Google+上获得的人类语言数据。

4.4.1 自然语言工具包概述

编写NLTK是为了让你在无前期大规模投入的情况下轻松地研究数据，并开始形成一些印象。然而，在跳过这一部分内容之前，考虑跟随示例4-8的解释器会话来了解NLTK提供的强大功能。由于你之前可能没有使用过NLTK，不要忘了，在需要时你可以随时使用内置的help函数获得更多的信息。例如，help（nltk）会提供NLTK包中的文档。

NLTK中并不是所有功能都是面向产品型软件的，因为其输出是被写到控制台的，不能被链表这样的数据结构获取。这样，nltk.text.concordance这样的方法被认为是演示功能。说到这个话题，许多NLTK模块都有demo函数，你可以调用demo函数来了解如何使用它们提供的功能，这些demo函数的源代码是学习如何使用新API的一个好的起点。例如，可以运行解释器中的nltk.text.demo（）来了解更多关于nltk.text模块提供的功能。

示例4-8的演示可以作为探索数据的良好起点，它的样例输出也包含在数据中作为交互式解释器会话的一部分，同样用来探索数据的一些命令也都包含在本章IPython的Notebook里。请跟随这个例子并检查它每一步的输出。您是否能够跟得上并理解解释器会话的处理流程？请先看一看，我们随后会讨论它的一些细节。

注意：下一个例子包括停用词，正如我们之前讲的，它是在文本中经常出现，但却传达非常少信息的词（如a、an和the等）。

示例4-8：用NLTK探索Google+数据

```
   # 通过探索数据来探索NLTK的一些功能
# Here are some suggestions for an interactive interpreter session.
import nltk
# 如果没有安装，下载辅助nltk包
nltk.download('stopwords')
all_content = " ".join([ a['object']['content'] for a in activity_results ])
# 文本的大小
print len(all_content)
tokens = all_content.split()
text = nltk.Text(tokens)
# 出现单词"open"的例子
text.concordance("open")
# 文本中的搭配（通常是有意义的词组）
text.collocations()
# 对该兴趣词的词频分析
fdist = text.vocab()
fdist["open"]
fdist["source"]
fdist["web"]
fdist["2.0"]
# 文本中的单词数
len(tokens)
# 文本中罕见单词的数量
len(fdist.keys())
# 不是停用词的普通单词
[w for w in fdist.keys()[:100] \
   if w.lower() not in nltk.corpus.stopwords.words('english')]
# 不是URL的长单词
[w for w in fdist.keys() if len(w) > 15 and not w.startswith("http")]
# URL的数量
len([w for w in fdist.keys() if w.startswith("http")])
# 枚举频率分布
```

```
for rank, word in enumerate(fdist):
    print rank, word, fdist[word]
```

注意：本章的例子（包括前面的例子）都使用split方法来分词。然而，分词并不意味着简单地用空格来划分单词，第5章会介绍更复杂的适应于一般情形的分词方法。

解释器会话（interpreter session）的最后一个命令列出了单词的频率分布，以按频率进行排序。毫不惊奇，像the、to和of这些停用词是最常出现的词，但是频率分布会急速下降并有一个长长的拖尾。尽管我们现在是对一个小的文本样例进行处理，但是此特点适用于任何自然语言的频率分析。

Zipf法则（http://bit.ly/1a1mCUD）是自然语言中著名的经验法则，该法则认为一个语料库中单词的频率和它在频率表中的排序成反比。这句话的意思是如果最常出现的词在语料库中占整个词的N%，那么第二高频率的词会占（N/2）%，第三高频率的词占（N/3）%，依此类推。当制作成图后，这样的分布就像图4-4中紧贴每个坐标轴行走的曲线那样（即便是对一个小样本量的数据）。

尽管开始不够明显，但这种分布的大部分区域都在它的尾部。并且从一门语言合理采样所得到充分大的语料库，尾部也总是很长。如果在图中绘制这种分布，且坐标轴用对数函数进行变换，对足够大的代表样本绘制的曲线会接近直线。

Zipf法则让你看到了语料库中单词频率的分布，并且提供了对预测频率有用的经验法则。例如，如果你知晓一个语料库中有一百万个单词，并假设频率最高的词（对英语来说，通常是the）占整个词的7%[1]，这时你要在频率分布中考虑一些特殊的单词，你就可以计算出一个算法所需要的逻辑计算数量。有时这种简单的算法就是对长时间运行程序的检测，或者用来确定在大数据集上的计算是否可行。

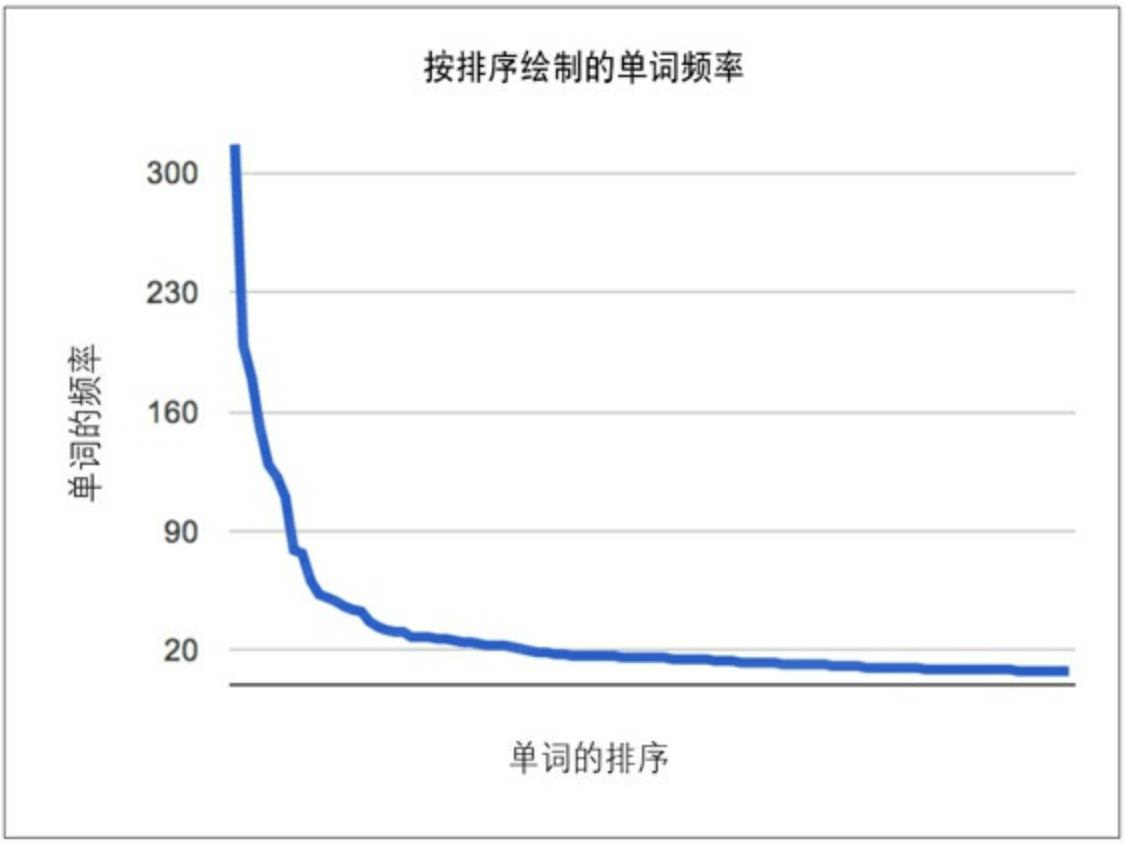


图4-4：Google+数据中一个小样本的频率分布，该分布紧贴着坐标轴；如果用双重对数图来绘制该分布，会得到一个很接近负数斜率的直线

注意：你能用第1章介绍的IPython的画图函数结合本章介绍的技

术，为数百个Google+动态，画出类似图4-4的相同曲线么？

4.4.2 对人类语言使用TF-IDF

让我来对之前收集的Google+数据应用TF-IDF，并看其作为查询数据的工具是被如何运用的。NLTK提供了一些可以使用的函数，而不用自己编写，因此在你理解了基本理论后，要做的事情就不多了。示例4-9的内容是假定你将本章之前使用的Google+数据保存为JSON文件，该代码可以让你进行多个查询来对相关文档打分。

示例4-9：用TF-IDF来查询Google+数据

```
   import json
import nltk
# 从你保存的文件中导入人类语言数据
DATA = 'resources/ch04-googleplus/107033731246200681024.json'
data = json.loads(open(DATA).read())
# XXX: 这里提供你自己的查询
QUERY_TERMS = ['SOPA']
activities = [activity['object']['content'].lower().split() \
              for activity in data \
                if activity['object']['content'] != ""]
# TextCollection 提供tf、idf和tf_idf的计算，这样我们就不需要自己计算
tc = nltk.TextCollection(activities)
relevant_activities = []
for idx in range(len(activities)):
    score = 0
    for term in [t.lower() for t in QUERY_TERMS]:
        score += tc.tf_idf(term, activities[idx])
    if score > 0:
        relevant_activities.append({'score': score, 'title': data[idx]['title'],
                                    'url': data[idx]['url']})
# 按分数排序，并显示结果
relevant_activities = sorted(relevant_activities,
                             key=lambda p: p['score'], reverse=True)
for activity in relevant_activities:
    print activity['title']
    print '\tLink: %s' % (activity['url'], )
    print '\tScore: %s' % (activity['score'], )
    print
```

对Tim O’Reilly的Google+数据执行“SOPA”（对立法提案的争论）样本查询的结果：

```
   I think the key point of this piece by +Mike Loukides, that PIPA and SOPA provide
a "right of ext...
    Link: https://plus.google.com/107033731246200681024/posts/ULi4RYpvQGT
    Score: 0.0805961208217
Learn to Be a Better Activist During the SOPA Blackouts +Clay Johnson has put
together an awesome...
    Link: https://plus.google.com/107033731246200681024/posts/hrC5aj7gS6v
    Score: 0.0255051015259
SOPA and PIPA are bad industrial policy There are many arguments against SOPA
and PIPA that are b...
    Link: https://plus.google.com/107033731246200681024/posts/LZs8TekXK2T
    Score: 0.0227351539694
Further thoughts on SOPA, and why Congress shouldn't listen to lobbyists
Colleen Taylor of GigaOM...
    Link: https://plus.google.com/107033731246200681024/posts/5Xd3VjFR8gx
    Score: 0.0112879721039
...
```

给定一个搜索词，当分析非结构化的文本数据时，能够集中在根据相关性排序的三个Google+内容结果上，会有巨大的好处。试试其他查询并定性的检查TF-IDF度量标准的效果如何。记住，分数的绝对值并不是十分重要，根据相关性找到并排序文档才是我们所关心的。随后，开始考虑各种调整或放大度量方法使其能更有效。一个很明显的改进是去除动词的变化（给读者留作练习），这样元素中如时态和语法的变化就可以归为一类，并可以用相似的计算来精确地解释。nltk.stem模块实现了几个好用的词干提取算法。

现在我们来使用新的工具，并将它们应用到寻找相似文档的基本问题上。总而言之，一旦你确定了感兴趣的文档，下一个步骤就是去发现

其他可能感兴趣的内容。

4.4.3 寻找相似文档

在你查询并发现感兴趣文档后，你想做的下一件事可能是找到相似文档。尽管根据搜索单词的TF-IDF可以提供方法缩小语料库的范围，但余弦相似度是比较两个文档相似度的最常用方法之一，这正是寻找相似文档的精髓所在。理解余弦相似度需要对向量空间模型（vector space model）做一个简介，下面就来讨论这个主题。

4.4.3.1 向量空间模型和余弦相似度理论

虽然我们已经强调过，TF-IDF是将文档建模为无序的单词集合，但另一个对文档建模的合适方法是向量空间模型。向量空间模型的基本理论是认为你有一个很大的多维空间，这个空间对每个文档都有一个向量，向量的距离就表示文档间的相似度。向量空间模型最美妙的地方在于你可以将查询表示成一个向量，并通过寻找与查询向量距离最近的文档向量，来发现最相关的文档。

尽管仅依靠本节短短的内容不可能完成这个课题，但是如果你对文本挖掘或者IR有兴趣，就应当对向量空间模型有个基本的理解。如果你对理论不感兴趣，并想直接跳到细节的实现，请直接跳到下一节。

警告：这部分假定你对三角法有基本的理解。如果你对三角法不熟，本章会是温习高中数学的一个绝佳机会。如果你对它不感兴趣，可以直接跳过该部分。请放心，我们用来查找相似文档的相似度计算是有一些精确的数学作支撑的。

首先，我们要弄清单词“向量”的意义，因为它对不同的领域会有很多细微的差别。总的来说，向量是一个数组的链表，它可以表示方向和大小，大小就是到原点的距离。在N维空间中，向量可以表示为一个原点到空间中一点的线段。

为了演示目的，想象有一个文档，该文档由两个单词（“Open”和“Web”）来定义，相应的向量为（0.45，0.67），该向量的值可以是单词的TF-IDF分数。在向量空间中，该文档可以表示为二维空间（0，0）点到（0.45，0.67）点的线段。就x/y坐标平面而言，x轴表示“Open”，y轴表示“Web”，从（0，0）到（0.45，0.67）的向量将表示我们讨论的文档。重要的文档通常最少包含成百上千的单词，但是对更高维度文档建模的方法是相同的，只是难于可视化。

试试将两维向量可视化文档的方法应用到有三个维度的文档中，例如（“Open”，“Web”，和“Government”）。然后考虑接受很难可视化的多维向量。如果你可以做到，你就会相信，对二维空间向量的操作也可以应用到10维或367维的空间中。图4-5展示了三维空间中的一个样例向量。

假设可以将文档建模为面向单词的向量，文档中的每个单词都由相应的TF-IDF分数表示，我们的任务是决定最能代表两个文档相似度的度量。事实证明，两个向量夹角的余弦值是比较两个文档相似度的有效度量，称为向量余弦相似度。尽管不够直接，但是多年的科学研究表明，计算文档间单词向量的余弦相似度是一个非常有效的度量方式（尽管还有许多问题，参考4.5节的概要介绍）。对余弦相似度的严密证明超出本书的范畴，但是其核心内容是任何两个向量夹角的余弦表明它们之间的相似度，并且等于它们单位向量的点乘（http://bit.ly/1a1mBjn）。

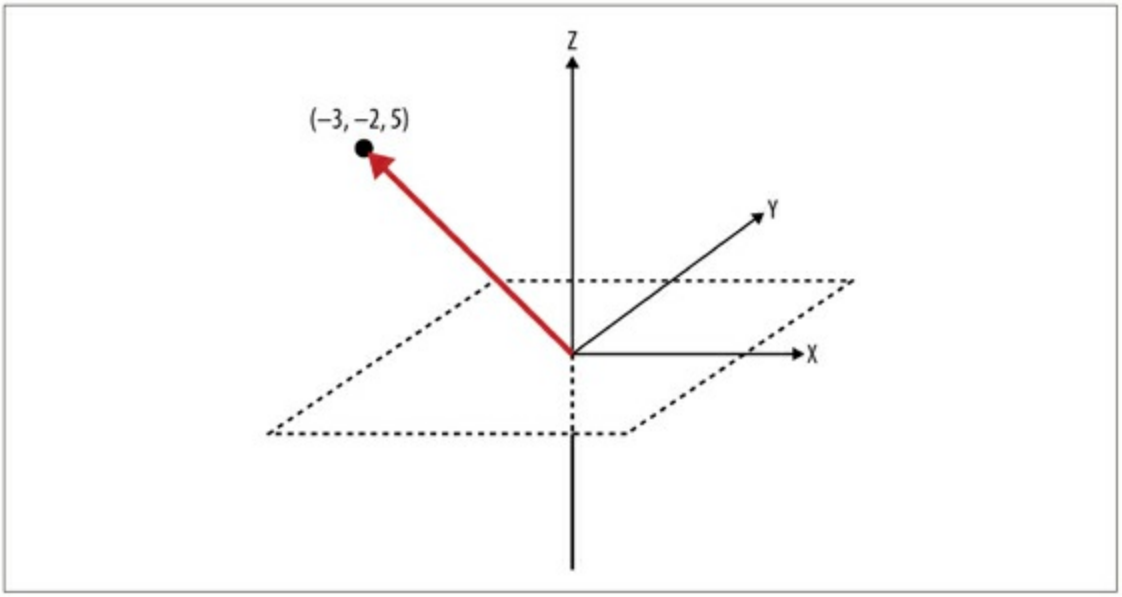


图4-5：三维空间上值为（—3，—2，5）的样例向量；从原点左移3个单位、后移2个单位并上移5个单位即得到该点

直观来看，可以这样想，如果两个向量越接近，它们之间的夹角也就越小，这样它们之间夹角的余弦值也就越大。两个相同向量之间的夹

角是0度，因此相似度为1，而正交的两个向量的夹角是90度，相似度为0。下面的简表用于演示这一点：

$\overrightarrow{doc1} \cdot \overrightarrow{doc2} = \|doc1\| \cdot \|doc2\| \cdot \cos\Theta$	已知（来自三角学原理）
$\frac{\overrightarrow{doc1} \cdot \overrightarrow{doc2}}{\|doc1\| \cdot \|doc2\|} = \cos\Theta$	使用除法
$\hat{doc1} \cdot \hat{doc2} = \cos\Theta$	根据“单位向量”的定义
$\hat{doc1} \cdot \hat{doc2} = Similarity\ (doc1, doc2)$	使用变量代换 （假设：$\cos\Theta = Similarity\ (doc1, doc2)$

假设单位向量的长度是1.0（根据定义），你可以看出用单位向量来计算文档的相似度的好处是，它们已经进行了标准化，而不会在长度上有很大的差异。我们将在后面章节中使用该方法。

4.4.3.2 用余弦相似度来归类帖子

前面讨论的最重要的一点是，要计算两个文档的相似度，你只需要把每个文档转换成一个单词的向量，并对这些文档计算单位向量的点积。为方便起见，NLTK提供nltk.cluster.util.cosine_distance（v1，v2）函数来计算余弦相似度，这样就很容易比较两个文档的相似度。如示例4-10所显示的，所有的工作都是为了产生正确的单词向量；简而言之，就是通过将TF-IDF分数分配到向量中，从而对给定的文档计算单词向量。因为两个文档的词表可能不同，所以对在一个文档中不存在而在另外一个文档中存在的单词，要用0.0在向量中占位。这样的效果是，两

个向量有相同的长度，并且分量按相同的顺序排序，这样就可以进行向量操作。

例如，假设文档1有单词（A，B，C），并有相应的TF-IDF权重（0.10，0.15，0.12），同时文档2有单词（C，D，E），并有相应的权重（0.05，0.10，0.09）。因此得到对文档1的向量为（0.10，0.15，0.12，0.0，0.0），对文档2的向量为（0.0，0.0，0.05，0.10，0.09）。这些向量都可以传给NLTK的cosine_distance函数，并得到余弦相似度。函数内部，cosine_distance调用numpy模块来高效的计算单位向量的点积并得出结果。尽管这一部分的代码重用了我们之前介绍的TF-IDF的计算，但是确切的打分函数可以替换为任何有用的度量方式。而TF-IDF（或对它的一些变形）常用于各种实现，能够提供一个极好的起点。

示例4-10阐述了用余弦相似度在Google+数据的语料库中找到最相似的文档的方法。它也可以应用到其他类型的人类语言数据，例如博客帖子或书籍。

示例4-10：用余弦相似度找到相似的文档

```
   import json
import nltk
# 读入保存的人类语言数据
DATA = 'resources/ch04-googleplus/10703373124620068102 4.json'
data = json.loads(open(DATA).read())
# 只考虑有1000个单词以上的内容
data = [ post for post in json.loads(open(DATA).read())
         if len(post['object']['content']) > 1000 ]
all_posts = [post['object']['content'].lower().split()
```

```
             for post in data ]
# 提供tf，idf和tf_idf打分函数
tc = nltk.TextCollection(all_posts)
# 计算单词-文档矩阵，例如td_matrix[doc_title][term]，返回文档中单词的tf-idf分数
td_matrix = {}
for idx in range(len(all_posts)):
    post = all_posts[idx]
    fdist = nltk.FreqDist(post)
    doc_title = data[idx]['title']
    url = data[idx]['url']
    td_matrix[(doc_title, url)] = {}
    for term in fdist.iterkeys():
        td_matrix[(doc_title, url)][term] = tc.tf_idf(term, post)
#建立向量，使相同单词的分数在相同的位置
distances = {}
for (title1, url1) in td_matrix.keys():
    distances[(title1, url1)] = {}
    (min_dist, most_similar) = (1.0, ('', ''))
    for (title2, url2) in td_matrix.keys():
        # 注意不要改变原来的数据结构
        # 因为我们在循环中，需要用到初始值许多次
        terms1 = td_matrix[(title1, url1)].copy()
        terms2 = td_matrix[(title2, url2)].copy()
        # 填补无单词的位置，这样相同长度的向量就可以用来计算
        for term1 in terms1:
            if term1 not in terms2:
                terms2[term1] = 0
        for term2 in terms2:
            if term2 not in terms1:
                terms1[term2] = 0
        # 对单词映射建立向量
        v1 = [score for (term, score) in sorted(terms1.items())]
        v2 = [score for (term, score) in sorted(terms2.items())]
        # 计算文件间的相似度
        distances[(title1, url1)][(title2, url2)] = \
            nltk.cluster.util.cosine_distance(v1, v2)
        if url1 == url2:
            #打印distances[(title1, url1)][(title2, url2)]
            continue
        if distances[(title1, url1)][(title2, url2)] < min_dist:
            (min_dist, most_similar) = (distances[(title1, url1)][(title2,
                                        url2)], (title2, url2))
    print '''Most similar to %s (%s)
\t%s (%s)
\tscore %f
''' % (title1, url1,
            most_similar[0], most_similar[1], 1-min_dist)
```

假设你觉得余弦相似度的讨论比较有趣，当你发现查询一个向量空间和计算文件之间的相似度是相同的操作，不同之处无非在于用查询向量和文档的向量之间的比较替代文档之间的比较，你会觉得这是十分神奇的。考虑一下：这是一个十分有意义的方式，用数学的方法就能够解

决。

然而，就实现一个程序来计算整个语料库的相似度而言，注意，最简单的方式就是构建一个含有你查询单词的向量，并和语料库中每个文档相比较。很明显，直接将查询向量与每一个可能的文档向量比较不是一个好的方法，即使是对一个中等大小的语料库。你需要做出一些工程性的决定，这涉及正确的使用索引以获得可扩展的解决方案。在第3章，我们简单地介绍了聚类中需要降维的基本问题，这里我们看到了相同的问题。不管什么时候，一旦遇到相似度计算，你会迫切需要降维来使它变得容易计算。

4.4.3.3 用矩阵图表示可视化文件相似度

本节将单词之间相似度可视化的方法是使用图表的结构，该结构中文档之间的链接表示它们的相似度。这样就会带来一个极好的机会来更多的介绍D3（http://bit.ly/1a1kGvo）可视化，D3是在第2章中介绍的可视化工具包。D3是根据数据科学家的兴趣来设计的，它采用大家熟知的声明性语法，并获得高层接口和底层接口之间的平衡点（middle ground）。

为适用于示例4-10，最小程度的改变就是需要收集节点和边，它们用来产生类似于D3例子（http://bit.ly/1a1lMal）中的视觉效果。一个嵌套循环可以计算Google+数据的样本语料库中文档之间的相似度，且单

词之间的联系根据一个简单的统计阈值标准来决定。

注意：涉及重组数据和输出结果（以支持可视化）的细节不在这里介绍，但是本章的样例代码在IPython Notebook中提供。

图4-6中，代码产生了一个矩阵图表。尽管这张图表的文本分类还不能看到，但你可以观察模型矩阵，并在你的浏览器中用交互可视化来查看分类标签。例如，在一个元素上停留可能会使用该标签作为工具提示。

和基于图的布局相比，矩阵图表的优势在于没有边（表示连接）之间的杂乱覆盖，因此你可以通过这一显示方式来避免众所周知的“毛线球”问题。然而，行和列的顺序会影响矩阵的模式，因此需要细心的思考行和列最佳的排序。通常，行和列用它们附加的属性来进行排序，这样能更容易确定数据的模式。本章的一个推荐练习是花些时间提高这一矩阵图表的表现力。

图4-6：表示Google+动态之间链接的矩阵图表

4.4.4 分析人类语言中的二元文法

如前面提到的，一个在非结构化的文本处理中常被忽略的问题是，当你能一次观察不止一个词项的时候，你会获得大量的信息，这是因为我们表达的概念很多都是词组，而不仅仅是独立的单词。例如，如果有人要告诉你，一张海报中最常出现的几个词是“open”，“source”，和“government”，你能说这个文本可能是关于“opensource”或“open government”的么？如果你有对作者或内容的先验知识，你可能会猜对。但是如果你全靠机器来分类一个文档是关于协同软件发展的，还是关于转型政府的，你需要回到文本中，并确定哪个词在“open”后出现的

频率最高，也就是你要找到以“open”开头的搭配。

回顾一下第3章中的相关内容，n元文法仅仅是一个简洁的表示方法，来表示文本中n个连续词的每一种可能的顺序，并且n元文法提供了用于计算搭配的基本数据结构。对于任意n，总有（n-1）个二元文法，如果你要考虑["Mr."，"Green"，"killed"，"Colonel"，"Mustard"]所有的二元文法合，你会有四种可能[（"Mr."，"Green"），（"Green"，"killed"），（"killed"，"Colonel"），（"Colonel"，"Mustard"）]。你需要比我们的样句更大的文本样例来决定搭配，但是假设你有背景知识或者有另外的文本，下一步应该做的是统计分析二元文法，来决定它们中哪些可能是搭配。

N元文法的存储要求

保存一个n元文法模型需要存储（T－1）*n个词（实际上是T×n），这里T是文档中单词的数量，n是n元文法的大小。例如，假设文档中包含1000个单词，需要约8KB的存储空间。存储文本中所有的二元文法需要两倍于原文本大小的空间，也就是16KB，也就是要存储999×2个单词。存储所有的三元文法（998×3）需要3倍于原大小空间，也就是24KB。这样，如果没有特殊的数据结构或压缩方法，对n元文法的存储花销将是原存储开销的n倍。

n元文法在用于将常见词聚类的问题时，十分简单，却又非常有效。如果你计算所有的n元文法（即使n的值不大），你会发现文本中一

些有趣的模式，这里不需要再做其他的工作。（具有代表性的是，你会经常看到二元文法和三元文法在数据挖掘的练习中使用）。例如，对一个足够长的文本考虑二元文法，你可能会发现如“Mr.Green”和“Colonel Mustard”这样的专有名以及如“open source”和“open government”这样的概念等。事实上，用这种方法计算二元文法得到的结果，和用之前运行的collocations函数所产生的结果是一样的，只是另外需要对不常见单词进行统计分析。当你考虑常用的三元文法或n元文法（n比3略大）的时候会有类似的模型。正如你已经在示例4-8中知晓的，NLTK可以很好的处理n元文法计算任务，发现文本中的搭配，发现一个或多个单词使用的语境。如示例4-11所示。

示例4-11：对一个句子用NLTK来计算二元文法和搭配

```
   import nltk
sentence = "Mr. Green killed Colonel Mustard in the study with the " + \
           "candlestick. Mr. Green is not a very nice fellow."
print nltk.ngrams(sentence.split(), 2)
txt = nltk.Text(sentence.split())
txt.collocations()
```

使用内置函数如nltk.Text.collocations的一个缺点是，这些函数通常不会返回你可以存储和操作的数据结构。一旦你遇到这样的情形，只要看看实际的源码即可，你可以很容易的从中学到一些东西，并将其修改为你所需要的代码。示例4-12展示了如何对一些词来计算搭配和相应的索引，并保存对结果的控制。

注意：在Python解释器中，你通常可以通过调用包的__file__属性来查找该包的源路径。例如，试试打印出nltk.__file__的值来查找NLTK源代码所在磁盘中的位置。在IPython或IPython Notebook中，你可以通过运行nltk来使用“双问号”magic（“double question mark magic”）函数，以便立即预览源代码。

示例4-12：以与nltk.Text.collocation样例函数相似的方式，用NLTK来计算搭配

```
   import json
import nltk
# Load in human language data from wherever you've saved it
DATA = 'resources/ch04-googleplus/10703731246200681024.json'
data = json.loads(open(DATA).read())
# Number of collocations to find
N = 25
all_tokens = [token for activity in data for token in activity['object']['content'
              ].lower().split()]
finder = nltk.BigramCollocationFinder.from_words(all_tokens)
finder.apply_freq_filter(2)
finder.apply_word_filter(lambda w: w in nltk.corpus.stopwords.words('english'))
scorer = nltk.metrics.BigramAssocMeasures.jaccard
collocations = finder.nbest(scorer, N)
for collocation in collocations:
    c = ' '.join(collocation)
    print c
```

总之，该实现大致仿效了NLTK的collocations演示功能。它过滤掉了出现次数小于最小值（这里是2）的二元文法，然后用打分度量将结果排序。这种情况下，打分函数是我们在第3章讨论的著名的雅卡尔系数（Jacard Index），它用nltk.metrics.BigramAssocMeasures.jaccard来定义。BigramAssocMeasures类用相依表（contingency table）来排序单词的共现（collocation），它是用任意给定的二元文法与在二元文法中出

现其他单词的概率相比较而得到。从概念上说，雅尔卡系数衡量了集合的相似度，在这种情形下，样本集是用来与文本中出现的二元文法来做具体的比较。

计算相依表和雅卡尔值的细节属于更高级的主题，但是下面的小节提供了对那些细节的延伸讨论，因为它们是更深入理解搭配检测的基础。

同时，我们从Tim O’Reilly的Google+数据来检验一些输出，很明显，返回带打分的二元文法要比只返回带打分的单词要有用得多，因为多出的上下文赋予单词更确切的意义：

```
   ada lovelace
jennifer pahlka
hod lipson
pine nuts
safe, welcoming
1st floor,
5 southampton
7ha cost:
bcs, 1st
borrow 42
broadcom masters
building, 5
date: friday
disaster relief
dissolvable sugar
do-it-yourself festival,
dot com
fabric samples
finance protection
london, wc2e
maximizing shareholder
patron profiles
portable disaster
rural co
vat tickets:
```

记住，这里没有使用特殊的启发方法或策略以根据题目大小写来检

查文本的专有名词，并且令人惊讶的是有许多专有名词和常见的短语也被过滤掉了。例如，AdaLovelace是一个相当知名的历史人物，Mr.O’Reilly有时会写关于她的内容（鉴于她与计算的联系）。Jennifer Pahlka因她的“Code for America”计划而知名，Mr.O’Reilly也从事这方面的工作。Hod Lipson是康奈尔大学的机器人学教授。尽管你可以通读全文并找出那些名字，但是机器可以为你做这件事，这样你就能投入精力到更值得关注的分析过程中去。

结果中仍然有一些不可避免的噪声，因为我们还没有从提取的单词中去除标点，但是对于我们已经提交的少量工作，得出的结果是极佳的。现在可以说，即使采用相当好的自然语言处理，也很难从文本分析的结果中去除所有的噪声。你需要做的是适应噪声并找到启发方法来控制噪声，直到机器能获得一个“完美的”结果。这里的“完美的”是根据受良好教育的人从文本中选出的结果界定的。

希望你得到的观察结果只花费很少精力和时间，我们已经能使用另一个基本技术从一些自由文本数据中抽取一些有用信息，并且结果看起来符合我们的预期。这是令人鼓舞的，因为这表明，对其他人的Google+数据（或其他非结构化文本）使用相同的技术，会得到一些重要信息，让你能够快速的看到正在讨论的关键词。同等重要的是，尽管这种情形下的数据可能证实了你所知道关于Tim O’Reilly的事情，但你可能会了解到更多新的事情，这可以从搭配表前面几项出现的人来印

证。尽管使用索引、正则表达式或者Python的find方法来找与“ada lovelace”相关的内容很简单，但是我们现在将利用我们之前在例4-9中开发的代码，并使用TF-IDF来查询“ada lovelace”。这是返回结果：

```
   I just got an email from +Suw Charman about Ada Lovelace Day,
and thought I'd share it here, sinc...
    Link: https://plus.google.com/107033731246200681024/posts/1XSAkDs9b44
    Score: 0.198150014715
```

这样你达到了目的：“ada lovelace”查询将我们引向关于“Ada Lovelace Day”的一些内容。你实际上已经从对文本词汇的理解开始，用搭配分析的方法来关注感兴趣的主题，并用TF-IDF在文本中搜索其中的主题。你当然也可以使用余弦相似度，来找到与Ada Lovelace（或是任意你想要找的）最相似的文档。

4.4.4.1 相依表和打分函数

注意：本节细述支撑BigramCollocationFinder运行的技术：示例4-12中使用的雅卡尔打分函数。如果这是你首次阅读本章或者你对这些细节不感兴趣，可以跳过这个部分，以后再回过头学习。毫无疑问这是一个深入的主题，你不需要充分的理解并在本章有效地掌握该技术。

一个用来计算关于二元文法度量方法的常见数据结构是相依表。相依表的目的是紧凑的显示关于二元文法不同组合的概率情况。观察表4-6的黑体部分，token1表示二元文法中有token1出现，~token1表示它没有出现在二元文法中。

表4-6：相依表的例子：斜体的数值表示条件概率，黑体部分表示的是在二元文法变形下的频率

	token1	*~token1*	
token2	frequency(token1, token2)	frequency(~token1, token2)	*frequency(*, token2)*
~token2	frequency(token1, ~token2)	frequency(~token1, ~token2)	
	*frequency(token1, *)*		*frequency(*, *)*

尽管在细节上一些单元格对不同的计算会有不同的重要性，但不难看到表中间的四个单元格，表达了不同二元文法中出现的频率。这些单元格的值可以计算各种相似性度量方法，它们可以用来打分，并按可能的重要性来排序二元文法，正如前面所介绍的雅卡尔系数那样，我们一会儿会仔细分析这个问题。我们先简要的讨论一下相依表的内容是如何计算的。

相依表中不同词条的计算方式，直接与你预先计算的或已经可用的数据结构相关。如果你假设只拥有文本中出现的各个二元文法的频率分布，那么计算频率（token1，token2）的方式就是直接查表，但是frequency（~token1，token2）又该是多少呢？在没有其他可用信息的情况下，你需要将每一出现token2的二元文法累加，并减去频率（token1，token2）。（如果这看起来不够明显，你可以花时间证明一下它的正确性。）

然而，如果假设你有一个可用的频率分布，它除了包括一个二元文

法的频率分布，还计算了文本中每个单词出现的次数（即一元文法），那么对于两个查找操作和一个算术操作，你可以用一个低开销的快捷方法。将一元组中出现token2的次数减去二元文法（token1，token2）出现的次数，剩下的就是二元文法（~token1，token2）出现的次数。例如，如果二元文法（“mr.”，“green”）出现三次，一元文法（“green”）出现4次（~“mr.”表示不出现“mr.”）。表4-6中，frequence（*，token2）表示一元文法token2是边缘频率，因为它被记录在表的边缘，frequence（token1，*）的值同样有助于计算frequence（token1，~token2），表达式frequence（*，*）表示任何一元组，也就相当于文本中出现的单词总数。假定有frequence（token1，token2）、frequence（token1，~token2）和frequence（~token1，token1），要计算frequence（toke*，*）的值则要计算frequence（~token1，~token1）的值。

尽管这里对相依表的讨论看起来可能有些离题，但这是理解不同打分函数的重要基础。例如，让我们回过头来考虑第3章的雅卡尔系数。理论上来说，该系数表达了两个集合的相似度，其定义为：

$$\frac{|Set1 \cap Set2|}{|Set1 \cup Set2|}$$

换句话说，这个定义就是两个集合相同单词的个数除以两个集合的并集里的不同的单词个数。如果Set1和Set2是相同的，两个集合的交集

和并集会是相同的，所有结果会得1.0。如果两个集合完全不同，公式的分子会是0，这样得到的结果就是0。

一个二元文法的雅尔卡系数表示，该二元文法的频率与所有含有感兴趣单词的二元文法出现的频率之和的比率。对该衡量标准的理解是，比率越高，（token1，token2）出现在文档中的可能性越大，搭配“token1token2”越有可能表达有意义的概念。

选择最恰当的打分函数通常基于隐藏在数据背后的特征和一些直觉，有时还会有点运气。在nltk.metrics.associations中定义的大多数度量方法都在《Foundations of Statistical Natural Language Processing》（MIT Press）第5章进行了讨论。该书可以在线（http://stanford.io/1a1mBQy）下载，可并作为后面描述内容的参考资料。

正态分布重要吗？

统计学中一个最基础的概念就是正态分布。正态分布因其形状，通常被称为钟形曲线。该分布之所以叫“正态”分布是因为它经常是其他分布所对比的对象。它是一个对称分布，该分布可能是统计学中应用最广泛的分布。它的意义如此深远的一个原因是，它能为现实世界遇到的许多自然现象提供一个模型，这些现象的范围很广，从人口特性到制造过程缺陷以及抛骰子的预测都有涉及。

表述正态分布有效性的经验法则叫做68-95-99.7规则

（http://bit.ly/1a1mEfo）。这是一个很简单的启发式方法，可以用来回答近似正态分布的许多问题。对于一个正态分布，可以证明99.7%的数据在均值的3个标准差范围内，95%的数据在2个标准差内，68%的数据在一个标准差内。这样，如果你知道可以用近似正态分布来解释现实中的一些现象，并知道该分布的均值和标准差，你就可以用它来回答许多有用的问题。图4-7给出了68-95-99.7规则。

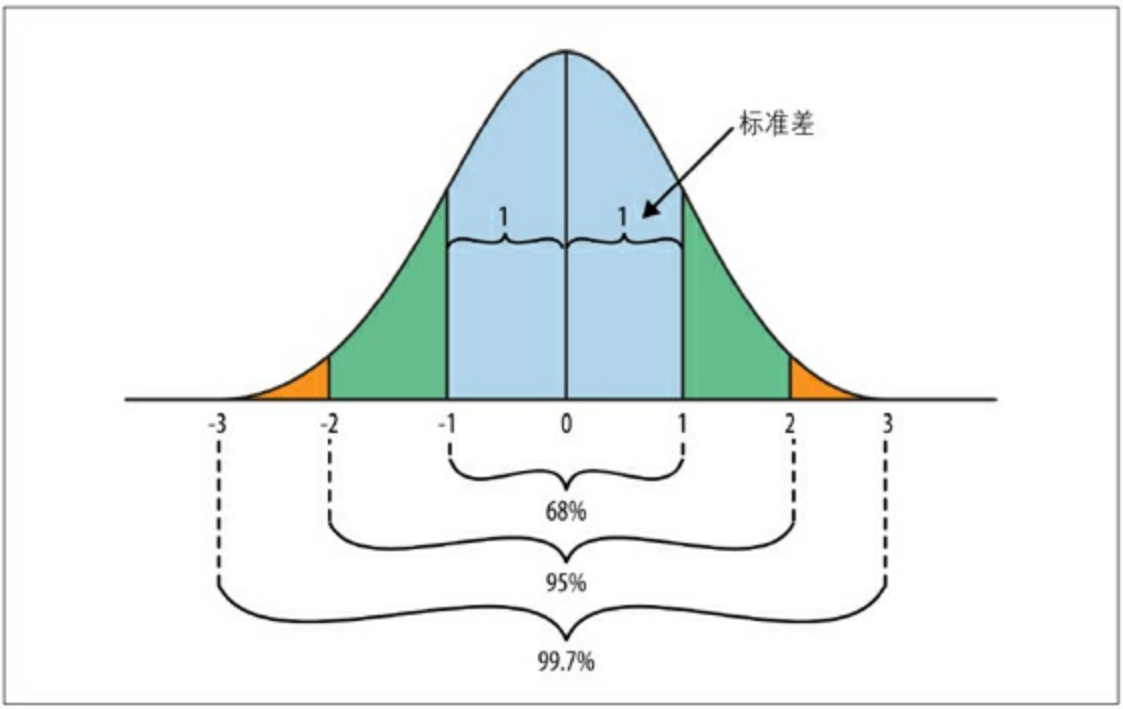


图4-7：正态分布是统计数学中的重要分布，因为它可以对许多自然现象的变化进行建模

Khan Academy的“正态分布简介”（http://bit.ly/1a1mCnm）提供了关于正态分布30分钟的综述；你也可以观看中心极限定理（http://bit.ly/1a1mCnA）中截取的十分钟内容片段，这一定理也是统计

学中同样高深的概念，它表明正态分布会以令人惊讶的方式涌现。

对这些度量方法的深刻讨论已超出本书的范围，但是刚刚提到的章节，用深入的例子提供了一个度量方法的清单。如果你需要建立自己的搭配检测器，雅卡尔系数、Dice系数，以及似然率都是好的开端。它们和一些其他的关键术语一起在下面予以描述：

原始频率

一如其名，原始频率（Raw frequency）是一个n元文法频率与所有n元文法频率的比率。它对检测文本中特定搭配的整体频率有用。

雅卡尔系数

雅卡尔系数（Jaccard Index）是用来测量集合之间的相似度的比率。如应用在搭配中，它被定义为一个搭配的频率除以搭配（在感兴趣的搭配中包含至少一个单词）的总数。它对判断给定的词汇是否能构成搭配很有用，同时也可以对可能的搭配进行排序。使用与先前描述一致的符号，可以用数学表达式定义为：

$$\frac{freq(term1, term2)}{freq(term1, term2) + freq(\sim term1, term2) + freq(term1, \sim term2)}$$

Dice系数

Dice系数（Dice's coefficient）与雅卡尔系数极为相似。基本的不同

是集合之间相似度的权重值是雅卡尔系数的两倍。它的数学公式定义如下：

$$\frac{2 * freq(term1, term2)}{freq(*, term2) + freq(term1, *)}$$

它也可以简化为：

$$Dice = \frac{2 * Jaccard}{1 + Jaccard}$$

当你想要增加集合重叠部分的分数时，可以用该度量方法来代替雅卡尔系数。当集合间有一个或多个不同，且不同的分数很高的时候，使用该方法很方便。因为当集合间的差集合大小不断增大时，雅卡尔系数会随之减小（因为集合的并操作处于雅卡尔分数的分母上）。

学生t分数

传统意义上，学生t分数（Student’s t-score）适用于假设检验。如应用到n元文法分析那样，t分数可以用来测试两个词是否搭配。该计算的统计过程对每个规范的t测试使用了标准分布。相对原始频率，t分数的一个优势是，t分数考虑了一个二元文法相对其构成的分量的频率。这个特性有助于排序搭配的强弱。t测试不好的地方在于它需要假定搭配的概率分布是正态分布，而这种情形不是经常满足。

卡方检验

就像学生t分数，卡方检验（Chi-square）这种度量方法通常用来测试两个变量之间的独立性，并可以用来测试是否为搭配的两个单词在Pearson卡方检验上是否满足统计显著性。总的来说，应用t测试和卡方测试的区别不是很大。卡方检验的优势在于它不像t测试，它没有假设变量背后的潜在分布是正态分布，因此，卡方测试使用更普遍。

似然率

似然率（Likelihood ratio）是另一个假设检验的度量方法，它用来测量可能会形成搭配的单词的独立性。一般情况下，相对卡方检验它是一个更正确的用来发现搭配的方法。它在一些数据上应用的很好，这些数据也包括许多不常用的搭配。涉及对搭配计算似然估计的方法（如NLTK实现的那样）假设它服从二项分布（binomial distribution）（http://bit.ly/1a1mEMj），决定该分布的参数是基于搭配和单词出现的次数来计算的。

逐点互信息

当你知道一个单词的相邻单词的值时，我们可以使用逐点互信息（Pointwise Mutual Information，PMI）来测量从这个单词获得了多少信息。换句话说，就是你从一个单词中可以得到另外一个单词的多少信息。具有讽刺意味的是，涉及PMI的计算使得高频词汇的分数反而低于低频词汇，这正好与我们想要的效果相反。因此，这是测量独立性而不

是相关性的好方法（例如，它不是对搭配打分的最理想选择）。事实还表明，稀疏的数据会阻碍PMI打分，而如似然率这样的方法效果会更好。

评估并选择特定情况下的最好方法通常既是一门科学又是一门艺术。一些问题已经被研究透彻，可以提供指导性方案，然而有一些情形通常需要更多创新的研究和实验。对于最困难的问题，应该查找最近的科学文献（不论是通过Google学术（http://bit.ly/1a1mHYk）搜索到的书还是文章），来确认正试图解决的问题是否被充分研究过。

4.4.5 分析人类语言数据的反思

本章已经介绍了多种工具和处理方法来分析人类语言数据，结束前的一些反思会对综合其内容有所帮助：

上下文驱动其意义

虽然TF-IDF是一个容易使用的强大工具，在具体实现的时候会有一些限制。为讨论方便我们之前将其忽略了，但是你应该将其考虑在内。最基本的一个限制是，该方法将文档看做是一个单词包，这也就意味着文本和查询中的单词顺序被忽略了。例如，如果我们不将查询词的顺序考虑在内，或将查询的内容解释为词组而不是独立的单词，那么查

询“Green Mr.”与查询“Mr.Green”会返回相同的结果。但显而易见，单词出现的顺序至关重要。

在进行n元文法分析来解释搭配和单词顺序时，我们仍会遇到潜在的问题，也就是TF-IDF假设所有相同文本值的词项都表示同一个事物。然而，很明显，情况并不是这样。同音同形异义词（homonym）（http://bit.ly/1a1mFzJ）是有相同的拼写和发音的词，它的意思是由上下文决定的，并且你选择的任何同音同形异义词都会是一个反例。同音同形异义词如book、match、cave和cool就是一些例子，它们表明在决定一个单词的意思时上下文的重要性。

余弦相似度有许多和TF-IDF一样的缺点。它也没有将文本的内容或n元文法分析中单词的顺序考虑在内，并且它假设向量空间中彼此靠近的词汇会是相似的。然而实际并不总是这样。正如TF-IDF一样，明显的反例就是同音同形异义词。我们对余弦相似度的实现，也依靠TF-IDF打分作为计算文档中单词相对重要性的方法，因此TF-IDF错误会有级联效应。

人类语言遭遇上下文过载

你或许已经注意到，在分析非结构化文本的时候会有许多讨厌的细节，而这些细节又十分重要。例如，字符串的对比会有大小写的问题，因此将单词正规化会很重要，这样就能尽可能正确的计算频率。然而，

盲目的将其变为小写也会使问题变的复杂，因为特定单词和词组的使用也是很重要的。

“Mr.Green”和“Web 2.0”是两个值得考虑的例子。对于例子“Mr.Green”保留“Green”会有优势，因为它能够提供线索来说明这个词不是形容词，可能是名词短语中的一部分。我们在第5章（讨论NLP）会再次讨论这个主题，因为用单词包的方法会丢失上下文信息，而用NLP更高级的词性标注技术会保留上下文信息。

从人类语言解析上下文不是件容易的事

另一个值得关注的问题，更多源于我们的具体实现，而不是TF-IDF的通用框架本身。我们用split来分割单词的时候，会在分割的单词中留下标点，这会影响对频率的统计。例如，在例4-6中，语料库['b']结束是“study.”；这和出现在['a']中的“study”不一样。在这种情形下，后面的标点会影响TF和IDF的计算。有时，我们看起来处理句子末尾的句号上下文内容很容易，但是对机器来说，要做出相同水准的判断会困难很多。

编写程序来帮助机器更好理解出现在人类语言数据的单词内容，是一个充满活力的研究领域，而且对未来的搜索技术和Web技术有巨大的潜在价值。

[1] 单词the在Brown语料库是占7%的词项，在不知道其他信息的条件

下，这为了解一个语料库提供了很不错的起点。

4.5 本章小结

本章介绍了Google+API以及如何收集、清洗人类语言数据，这是练习查询个人Google+动态的一部分。我们随后学习了IR理论、TF-IDF、余弦相似度和搭配的一些基础知识来分析我们收集的数据。最后，我考虑了任何搜索引擎为打造成功的技术产品都会考虑的一些问题。然而，即使我希望，通过本章让你们知道如何从非结构化的文本中提取有用的信息，但是对大多数基础概念来说（无论出于理论还是工程方面的考虑），这仅仅是皮毛。信息检索毫不夸张地讲是数十亿的产业，因此不难想象如Google和Bing这样可以支撑搜索引擎的大公司在理论和实现方面做了多么巨大的投入。

考虑如Google这样的搜索提供商的巨大能量，很容易忘记这些基本的搜索技术还存在。然而，明白这些技术会帮助理解搜索现状的假设和限制，也有助于清楚的理解今后出现的核心技术。第5章介绍对本章一些技术的基本范式转移。对于那些属于技术驱动型的、能有效分析人类语言数据的公司，有许多令人兴奋的机会。

注意：本章和其他章节的源代码都可以在GitHub（http://bit.ly/1a1kNqy）上以IPython Notebook的格式（鼓励用该方式而不是你习惯的浏览器阅读）找到。

4.6 推荐练习

·利用第1章介绍的IPythonNotebook的画图特性，为语料库中的单词绘制Zipf曲线。

·挖掘评论内容，并尝试根据评论的频率来判断趋势。例如，对像Tim O'Reilly这样的Google+用户，谁是对其数百次的动态评论频率最高的人。

·挖掘Google+动态来发现哪些是最流行的活动。出发点可以是评论和分享的数量。

·从Google+动态中的链接获取内容，并用本章的cleanHtml函数来提取网页的文本信息。分享的链接中有相同的主题么？文本中最常出现的词是什么？

·如果你想对网页应用本章的技术，你可以查一下Scrapy（http://bit.ly/1a1mG6P），它是方便使用并且成熟的网页爬取框架，可以帮助你收集网页的信息。

·花一些时间对本章的矩阵表增添交互功能。当文本被单击时，你能增加事件句柄来自动的进入发布页面么？你能想到任何有意义的方式，来排序行和列以便更容易的鉴别模式么？

·更新发布JSON的代码，让JSON来驱动矩阵图表，以便它能够计算不同的相似度。因此这种使文档相关的方式不同于默认的实现方式。

·你能想到文本中其他的特征，以便更准确的计算文档之间的相似度么？

·花些时间钻研本章介绍的基本IR理论。

4.7 在线资源

下面是本章的链接清单，或许对回顾有所帮助：

·68-95-99.7规则（http://bit.ly/1a1mEf0）

·二项分布（http://bit.ly/1a1mEMj）

·Brown语料库（http://bit.ly/1a1mB2X）

·中心极限定理（http://bit.ly/1a1mCnA）

·D3.js样例集（http://bit.ly/1a1lMal）

·Google+API控制台（http://bit.ly/1a1mwMP）

·google-api-python-client（http://bit.ly/1a1mzYI）

·Google+API参考手册（http://bit.ly/1a1mtR0）

·HTTP API概述（http://bit.ly/1a1mAfm）

·信息检索简介（http://stanford.io/1a1mAvP）

·正态分布简介（http://bit.ly/1a1mCnm）

·Manning and Schütze（第5章搭配）（http://stanford.io/1a1mBQy）

·NLTK在线文档（http://bit.ly/1a1mtAk）

·Scrapy（http://bit.ly/1a1mG6P）

·社交图谱（http://bit.ly/1a1my7k）

·Zipf法则（http://bit.ly/1a1mCUD）

第5章　挖掘网页：使用自然语言处理理解人类语言、总结博客内容等

本章延续前面章节的讲解，尝试使用自然语言处理（NLP）的方法对网页信息进行挖掘，并且将其应用到在社交网络（或其他地方）遇到的大量人类语言数据[1]。上一章介绍了信息检索（IR）理论中的基本技术，大体上将文本视为以文档为中心的“词袋”（无序的单词集合），并可以用向量来建模和处理。尽管这些模型在大多数情况下表现很好，但它们通常没有将当前上下文赋予单词意义的线索最大化。

本章采用以上下文为驱动的不同技术来对人类语言数据的语义进行更深层次的挖掘。社交网络的API能够返回符合良好定义的模式的数据是很重要的，不过人类还是以自然语言的方式进行最基本的交流，比如你在本页读到的内容、Facebook上发布的内容、链在推文（tweet）中的网页等等。到目前为止，人类语言对我们来说是最常见的，未来以数据为驱动的创新很大程度上依赖于我们是否能够有效地使机器具备理解数字形式的人类交流的能力。

注意：我们强烈推荐你在学习这章之前对前面章节的内容有了很好的掌握。需要对NLP有良好理解，例如对TF-IDF、向量空间模型等的基本优缺点的评价及其应用知识的理解。从某种程度上来说，本章和上一

章比其他章节在这方面的耦合度更高。

本着前几章的精神，我们将尝试用最少的细节来让你对本身很复杂的话题有基本的了解，同时我们也钻研了会使用到的技术，这样你就可以立即进行数据挖掘了。虽然我们经常走捷径，通过使用20%的关键技术便可以完成80%的工作（任何一本书或者小型多卷集书籍的任何一章都不可能完全介绍NLP的内容），但是这一章的内容是一个实际的介绍，会给你足够的信息来对你在社交网络上获得的人类语言数据做一些有趣的事情。尽管我们将专注于从网页和订阅中提取人类语言数据，但是不要忘记社交网络几乎都会提供API来返回人类语言，所以这些技术可以针对几乎所有的社交网络。

注意：在线获取本章（或其他章节）修复bug的源码地址为http://bit.ly/MiningTheSocialWeb2E。好好利用本书虚拟机的经验，比如附录A中描述的那样，来增加你对示例代码的兴趣。

[1] 在本章中，人类语言数据指自然语言处理的对象，意在传达与自然语言数据或非结构化数据相同的意思。如果不是数据本身表达出明确的不同，那么对用词的选择不会造成区别。

5.1 概述

本章继续分析人类语言数据，并以网页和订阅为基础。在本章中你会学到：

·获取网页并从中提取人类语言数据。

·利用NLTK完成自然语言处理中的基本问题。

·在NLP中使用上下文驱动的方法进行分析。

·使用NLP来解决分析性问题，比如生成文档摘要。

·度量涉及预测分析领域质量的准则。

5.2 抓取、解析、爬取网页

尽管使用编程语言或者终端工具，比如curl或者wget来获取网页很繁琐，但提取你想从网页中得到的独立文本并没有这么繁琐。文本显然已经在网页中了，但还有许多样板（boilerplate）的内容，比如导航栏、头、脚注、广告等其他你不关心的内容。因此，并不是去掉HTML标签再处理剩下的文本就行了，因为去掉HTML标签不会对去掉那些样板有什么作用。有些时候，网页中会有更多的样板给你造成干扰。

好消息是，近年来帮你识别感兴趣内容的工具越来越成熟，并且对你想要进行文本挖掘的材料进行隔离的方法也有了很多出色的可供选择。另外，相对普遍的订阅，比如RSS和Atom，可以帮助检索出干净的没有网页上那些典型的没用信息的文本，如果你对于获取订阅内容有先见之明，可以使用它们。

注意：通常订阅只发布“最近的”内容，所以即使有了订阅，有时你也需要处理网页。如果给你选择，或许你更倾向于订阅而不是任意的网页，但你需要为这二者都做好准备。

一个网页抓取（从网页提取文本的过程）的出色工具是基于Java的boilerpipe库（http://bit.ly/1a1mMLA），它是为了识别和移除网页中的样板而设计的。boilerpipe库基于一篇发表的论文，题目是“Boilerplate

Detectios Using Shallow Text Features”（http://bit.ly/1a1mN21），阐释了使用监督式学习（http://bit.ly/1a1mPHr）技术来分离样板和页面内容的效力。监督式学习技术包含创建在它的领域中有代表性的训练样本的预测模型的过程，因此boilerpipe是可以定制的，你可以调整它来提高精确度。

虽然这个库是基于Java的，它很有用也很流行，一个叫做python-boilerpipe（http://bit.ly/1a1mSD4）的Python包封装了它。安装这个包可以使用pip install boilerpipe命令。确保你的Java版本是相对新的，这就是使用boilerpipe的所有要求了。

示例5-1展示了提取文章的正文内容的直接使用，由ArticleExtractor参数表示，传到Extractor构造器中。你也可以在线尝试boilerpipe的托管版本（http://bit.ly/1a1mSTF）来看看其他提取器的不同，比如LargestContentExtractor或者DefaultExtractor。

默认的提取器可以在一般的情况下工作，一个为了包含文章的网页训练好的提取器和一个为了提取页面上大规模的正文的提取器，它们很适合于那些只有一个大规模正文的网页。在任何情况下，仍会有对文本进行少量的后处理的需求，这取决于你能识别的其他特征是起干扰的作用还是需要注意的内容，但是让boilerpipe来做这些困难的工作却是很简单的。

示例5-1：使用boilerpipe从网页中提取文本

```
   fromboilerpipe.extract import Extractor
URL='http://radar.oreilly.com/2010/07/louvre-industrial-age-henry-ford.html'
extractor = Extractor(extractor='ArticleExtractor', url=URL)
printextractor.getText()
```

尽管网页抓取过去常常被视为获取网站内容的唯一方式，现在有一种潜在的更简单的方法来获取内容，尤其是当内容来自新的源、博客、或其他聚合的源。但是在谈论这些以前，让我们先开始一个快速的记忆回顾。

如果你已经使用网络很长时间了，你也许记得在20世纪90年代末，那时新闻读者还不存在。如果你想知道一个网站的最新的改变，你不得不到那个网站去亲自看有没有什么改变。之后，聚合格式利用了自己发布的博客和格式，比如RSS（Really Simple Syndication，简易信息聚合）和Atom生成的进化的XML（http://bit.ly/18RFKaW）规格。现在在处理内容提供者发布内容和用户订阅方面都越来越流行了。分析订阅是比较简单的问题，因为订阅是符合（http://bit.ly/1a1mTqE）发布标准的格式固定的（http://bit.ly/1a1mQLr）XML数据，尽管网页不一定是格式良好的、有效的或者是遵循最佳实践的。

通常用来处理订阅的Python包是feed parse，它是处理订阅的重要工具。你可以在终端中使用标准的pip命令pip install feedparser来安装它。示例5-2展示了提取文本、标题、RSS订阅的入口的源URL的最简单的使

用方法。

示例5-2：使用feedparser从RSS和Atom订阅中提取文本（和其他域）

```
   import feedparser
FEED_URL='http://feeds.feedburner.com/oreilly/radar/atom'
fp = feedparser.parse(FEED_URL)
for e in fp.entries:
    printe.title
    printe.links[0].href
    printe.content[0].value
```

HTML、XML和XHTML

在早期网络演化的过程中，将网页的内容从展示层分离出来的困难很快成为了一个亟待解决的问题，XML（部分上）是当时的一种解决方法。内容的创建者将数据以XML的格式发布，并使用样式表将它转换成可向终端用户展示的XHTML。XHTML是将HTML用格式良好的XML写出来的语言：每个标签都是小写，组成树形结构，并且标签都是自封闭的（比如
）或者每个开始标签（比如 ）都有相对应的结束标签（ ）。

在网页抓取的上下文中，这些规定有许多好处，比如让分析器对每个网页更好处理。在设计方面，XML正是网络所需要的。这个主张有很多好处并且几乎没有坏处：格式良好的XHTML内容可以证明对XML模式有效，而且能享受XML的所有其他优点，如使用命名空间（RDFa这类语义网技术依赖的设计）的自定义属性。

问题是它并没有引人注意。结果，我们仍然生活在一个基于HTML 4.01标准的语义标记的世界里（虽然经过了十余年，但HTML 4.01标准仍然在蓬勃发展），而XHTML和基于XHTML的技术（比如RDFa）仍然不是主流。（事实上，像BeautifulSoup（http://bit.ly/1a1mRit）这样的库是为了处理格式不规范的HTML而设计的。）大多数网络开发者正屏住呼吸，期待HTML5（http://bit.ly/1a1mRz5）能够创造一个期待已久的融合，比如microdata（http://bit.ly/1a1mRPA）技术的流行和发布工具的现代化。如果你对这段历史感兴趣的话，维基百科中关于HTML的文章（http://bit.ly/1a1mS66）很值得一读。

爬取网站是对和这一节提到的相同的概念的逻辑扩展：它通常包括获得页面、提取页面中的超链接，然后系统的获得所有超链接的页面。这个过程根据你的目标可以重复任意多层。这也是最早的搜索引擎的工作方法，并且也是现在大多数的索引网页的搜索引擎所使用的。尽管爬取网页并不在我们的讨论范围之内，但解决这个问题所应用的知识很有用，所以让我们简单的思考一下获得所有页面的计算复杂性。

如果你想实现自己的网络爬取，Scrapy（http://bit.ly/1a1mG6P）是一个很好的基于Python的网页爬取框架。实现网页爬取并不是本章讨论的范围，不过Scrapy的在线文档可以指导你不费吹灰之力就能爬取目标网页。下一节简单的讨论了一般情况下实现网页爬取的计算复杂性，这样你就能更好的理解你要做的东西。

注意：如今我们可以从比如亚马逊的Common Crawl Corpus（http://amzn.to/1a1mXXb）这样的源，来获得一个符合大部分科研目的周期性更新的网页爬取，记录了超过50亿的网页和超过81兆兆字节的数据！

5.2.1 网页爬取中的广度优先搜索

注意：这一节包括细节的内容以及对如何实现网页爬取的分析，对于你理解本章的内容影响不大（尽管你很可能觉得这一节很有趣也很有启发意义）。如果这是你第一次读这一章，可以把本节留作下次阅读。

网页爬取最基本的算法是广度优先搜索（http://bit.ly/1a1mYdG），这种搜索是有固定开始节点和一定约束条件的树状或者图形结构。在我们的网页爬取方案中，开始节点是最初的网页，相邻节点是超链接的网页。

其他的方案中，深度优先搜索（http://bit.ly/1a1mVPd）是常见的代替广度优先搜索的策略。选择哪一种策略取决于可用的计算资源、特定的领域知识，甚至理论层面的考虑。广度优先搜索在示例5-3中用伪代码展示了它是如何工作的，图5-1也形象地展示了搜索的过程。

示例5-3：广度优先搜索的伪代码

```
   Create an empty graph
Create an empty queue to keep track of nodes that need to be processed
Add the starting point to the graph as the root node
Add the root node to a queue for processing
Repeat until some maximum depth is reached or the queue is empty:
  Remove a node from the queue
  For each of the node's neighbors:
    If the neighbor hasn't already been processed:
    Add it to the queue
    Add it to the graph
    Create an edge in the graph that connects the node and its neighbor
```

通常我们不会花这么长时间来分析一个方法，但是广度优先搜索是你需要了解的非常重要的一种方法。一般有两种对算法的度量标准：效率和效力（或者说性能和质量）。

通常对算法的性能分析包括最坏情况的时间和空间复杂性——换句话说，就是程序的执行时间和在处理大规模数据的时候对内存的要求。我们爬取网页的广度优先方法本质上就是广度优先搜索，除非没有要查找的东西，因为扩展图时，除了达到最大层数或遍历完所有的节点，并没有终止条件。如果我们要搜索特定的东西而不是任意的爬取链接，就需要考虑实际的广度优先搜索。因此，通常将广度优先搜索演变为有边界的广度优先搜索，正如示例中一样，会对搜索的最大层数进行限制。

对于广度优先搜索（或广度优先爬取）来说，最坏情况下的时间和空间复杂性都能限制在b^d，b是图的分支数，d是层数。你可以像图5-1那样在纸上画一个草图，经过简单思考，结果就会变得很显然。

如果图中的每个节点都有5个相邻节点，如果只有一层，你结束时遍历了6个节点：根节点和它的5个相邻节点。如果这5个相邻节点又都

有5个相邻节点，再扩展一层，结束时总共有31个节点：根节点、根节点的5个相邻节点以及这5个相邻节点的5个相邻节点。表5-1展示了b^d在b和d的规模都比较小时是如何增长的。

表5-1：不同层数的图的分支数的计算

分支数	节点数 层数=1	节点数 层数=2	节点数 层数=3	节点数 层数=4	节点数 层数=5
2	3	7	15	31	63
3	4	13	40	121	364
4	5	21	85	341	1365
5	6	31	156	781	3906
6	7	43	259	1555	9331

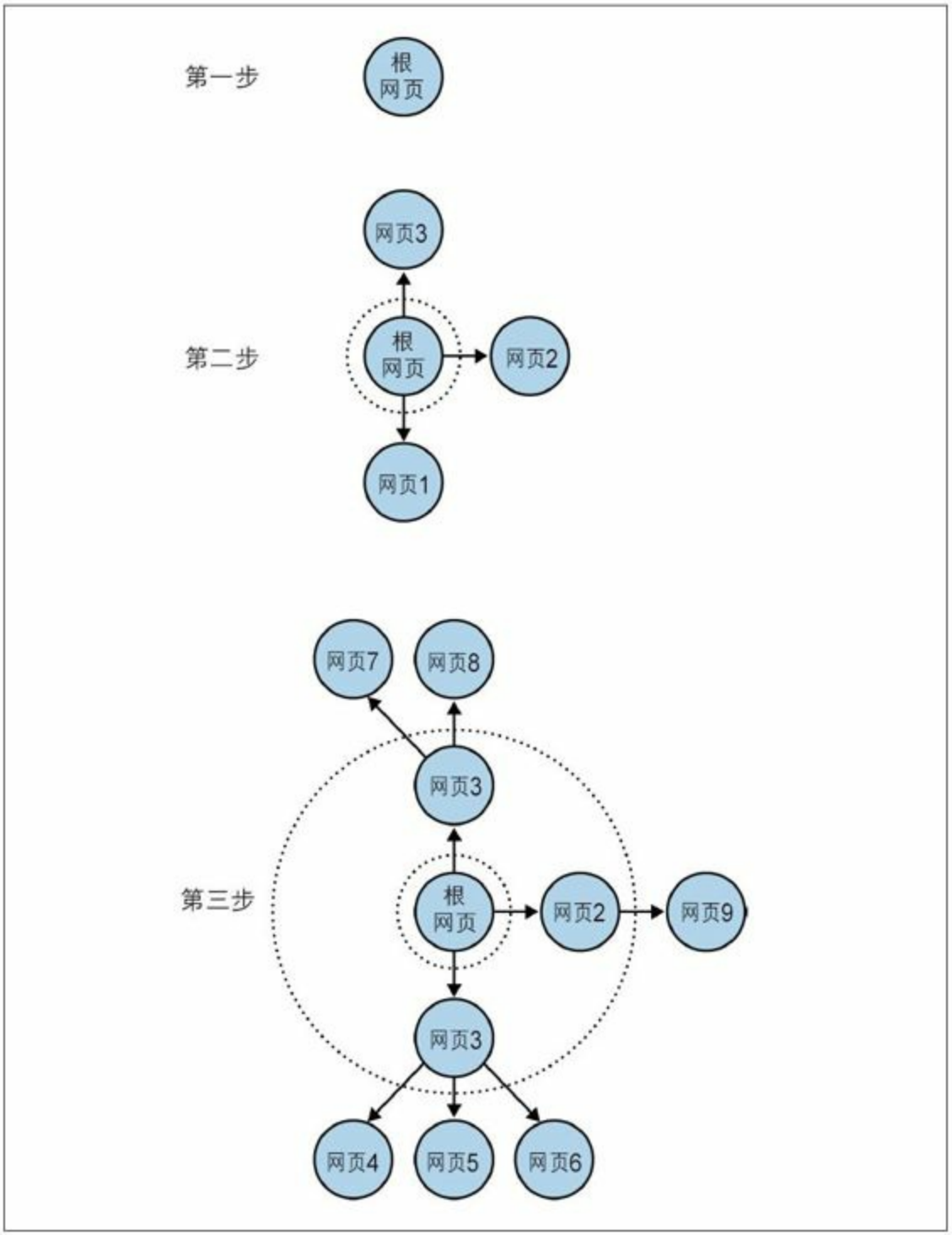


图5-1：在广度优先搜索中，每进行一步就扩展一层，直到达到最大层数或者满足其他的终止条件

图5-2形象地展示了表5-1中的值。

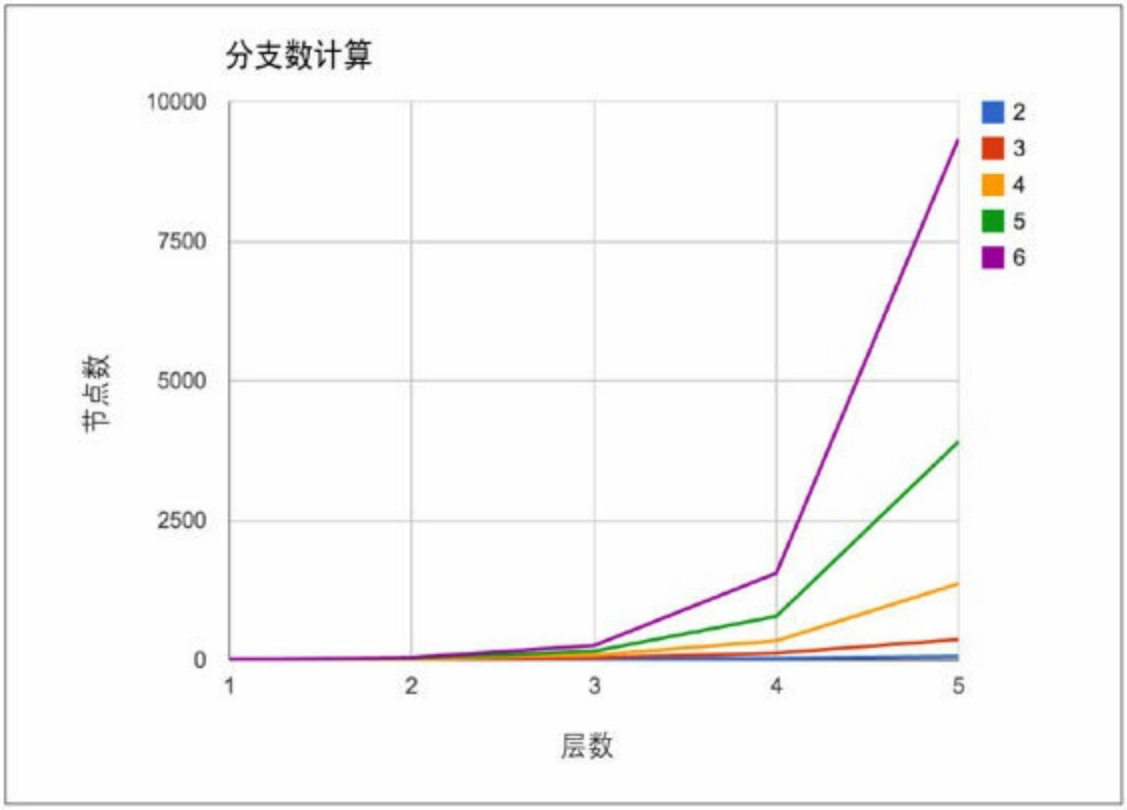


图5-2：随着广度优先搜索的层数增加而增加节点数

尽管前面主要讲了算法的理论界限，但值得注意的最终的考虑因素是：对于一个固定大小的数据集，该算法的实际性能到底如何。对代码的简单分析表明，从“绝大多数时间都用在等待对库的调用来返回处理的内容”的观点来看，它主要是I/O约束。这种情况下，线程池技术是提高性能的普遍方法。

5.3 通过解码语法来探索语义

回顾前面的章节，TF-IDF和余弦相似度最基本的缺点是这两个模型本身就不要求深入了解数据的语义，并且抛开了许多关键的上下文环境。然而，那一章的示例能利用非常基本的句法，用空格分割单词词项（token）来把其他不透明的文档分解为“词袋”（bag of words，http://bit.ly/1a1lHDF），并使用频率和简单的统计相似性度量来确定数据中的哪些单词（token）可能是重要的。虽然你可以使用这些技术做一些非常了不起的事情，但是它们并没有真正告诉你某个文档上下文中出现的单词的意义。包含“fish”或“bear”甚至“google”这类同形异义词（http://bit.ly/1a1mWCL）[1]的句子就是一个很好的例子；它们既可以是名词，也可以是动词。

NLP本身就很复杂，并且很难做到完美，在常用语言的大型集合中完全掌握它可能会是本世纪的问题。很多人认为这个问题离解决还有很远的距离，然而我们对网络的“更深理解”的兴趣已经有所提高了。比如Google的知识图谱（Google’s Knowledge Graph，http://bit.ly/1a1mZyk），正被推举为未来的搜索。毕竟，对NLP的完全掌握实质上是掌握图灵测试（http://bit.ly/1a1mZON）的合理策略，对于最仔细的观察者来说，实现这种“理解”的计算机程序，虽然是用软件对大脑建模而不是真正的大脑，也说明了人工智能的不可思议。

虽然结构化或半结构化源基本上是记录的集合，同时这些记录已经为可以被直接分析的每个字段预设了一些意义，但是即使是最简单的任务，处理人类语言数据也有更多细致的考虑。例如，让我们假设给了你一个文档，并要求你统计其中句子的总数。如果你是人类并且基本了解英语语法的话，这是非常简单的任务，但是对于机器来说，这就是完全另一回事了，它要求复杂而详细的指令集来完成这个任务。

值得庆幸的是，只要数据的结构相对合理，机器就可以检测出其断句的位置，不但速度很快，而且准确率也很高。即使你已经准确地检测到了所有的句子，你仍然不了解这些句子中词或短语的用法。想想说反话或者讽刺用语这样的例子，即使对数据的结构信息的了解已经相当完全了，你还需要这个句子之外的上下文来准确的理解它。

因此，作为极度广泛的概括，我们可以说NLP基本上是关于使用由符号的有序集组成的不透明的文档，这些符号遵循正确的句法和定义明确的语法，并最终推导出与这些符号相关的语义。

让我们回到大多数NLP流程的第一步：检测句子来说明NLP的复杂性。似乎很容易高估简单的基于规则的启发式方法的作用，但重要的是通过一个练习来让你了解关键问题是什么，不要把时间浪费在尝试重蹈覆辙上。

解决句子检测问题的最初尝试可能只是统计句子中句号、问号和感

叹号的数量。这可能是开始时最明显的启发式方法，但是它相当不成熟，可能会产生很大的误差范围。看一下下面的（非常明确的）控诉：

Mr.Green killed Colonel Mustard in the study with the candlestick.Mr.Green is not a very nice fellow.

通过标点符号（本例中是句号）来简单地切分这个句子，会产生下面的效果：

```
   >>> txt = "Mr. Green killed Colonel Mustard in the study with the \
... candlestick. Mr. Green is not a very nice fellow."
>>> txt.split(".")
['Mr', 'Green killed Colonel Mustard in the study with the candlestick',
'Mr', 'Green is not a very nice fellow', '']
```

显而易见的是：不综合考虑上下文或者高级信息，只是盲目地按句号断句来执行句子检测是不够的。在本例中，问题是英语中常见的缩写词“Mr.”的使用。虽然前面的章节对该样本的n元语法分析已经说明“Mr.Green”实际上是一个被称为搭配或词块的复合标记，但是如果要分析数量更大的文本，就不难想象难以检测以搭配出现为基础的其他边界情况。提前想一下，同样值得指出的是：使用简单的逻辑在句子中查找关键主题也并不容易完成。作为聪明的人类，你可以很容易地推断出样本中的关键主题可能是“Mr.Green”、“Colonel Mustard”、“the study”和“the candlestick”，但是训练机器得出相同的结果是一项复杂的任务。

注意：在继续讨论剩余部分之前，花时间想一下怎么写程序解决这个问题。

可能会出现一些明显的可能性，如使用正则表达式检测“首字母大写”，构建常见缩写词列表解析出专有名词，将这个逻辑的一些变形应用到查找句末（EOS）边界的问题中，以防你陷入困境。

当然，这些努力会对一些示例有效，但是对于任意的英语文本，误差范围会是怎样的呢？算法能在多大程度上容忍用法不标准的英语文本；文本消息或tweet这类高度浓缩的信息；或（夹杂）其他浪漫的语言，如西班牙语、法语或意大利语？这里没有简单的答案，这就是文本分析在数字化的人类语言数据几乎每秒都在增加的时代中如此重要的原因。

5.3.1 逐步讲解自然语言处理

我们准备通过检查一系列示例来讲解使用NLTK的NLP。我们要检查的NLP流程有下面几步：

1.句末检测

2.切词

3.词性标记

4.分块

5.提取

我们继续使用下面的样例文本来说明：“Mr.Green killed Colonel Mustard in the study with the candlestick.Mr.Green is not a very nice fellow.”请记住，即使你已经阅读了该文本，了解了它的语法结构，但是对于机器来说，现在它只是一个不透明的字符串值。让我们来更详细地查看这些步骤。

注意：下面的NLP流程按照在Python解释器中展开的形式来给出，是为了清晰简明的展示每一步的输入和预期输出。不过，流程的每一步都预先装载进本章的IPython Notebook中了，所以你可以和其他的例子一起继续学习。

这五步介绍如下。

EOS检测

这一步把一段文本分解成一个有意义的句子集合。因为句子通常是代表思维的逻辑单元，所以它们往往都包含非常适合于进一步分析的可预见句法。你看到的大多数NLP流程都是从这一步开始的，因为切词（下一步）操作的是单句。虽然把文本分解成段或节可能会增加某些分

析类型的值，但是它不可能帮助EOS检测整体任务。在解释器中，使用NLTK解析句子的过程如下：

```
   >>> import nltk
>>> txt = "Mr. Green killed Colonel Mustard in the study with the \
... candlestick. Mr. Green is not a very nice fellow."
>>> txt = "Mr. Green killed Colonel Mustard in the study with the \
... candlestick. Mr. Green is not a very nice fellow."
>>> sentences = nltk.tokenize.sent_tokenize(txt)
>>> sentences
['Mr. Green killed Colonel Mustard in the study with the candlestick.',
'Mr. Green is not a very nice fellow.']
```

我们会在下一节更多的讨论sent_tokenize底层发生了什么。现在，我们将会接受出现在任何文本中的正确的句子检测的字面意义——对在可能是标点符号的字符处停顿的明显改善。

切词

这一步操作的是单个句子，把它们分割成单词（token）。仿照示例解释器会话，需要执行以下操作：

```
   >>> tokens = [nltk.tokenize.word_tokenize(s) for s in sentences]
>>> tokens
[['Mr.', 'Green', 'killed', 'Colonel', 'Mustard', 'in', 'the', 'study',
  'with', 'the', 'candlestick', '.'],
['Mr.', 'Green', 'is', 'not', 'a', 'very', 'nice', 'fellow', '.']]
```

请注意，对于这个简单的示例，切词和在空格上分割所做的操作一样，只是它能正确地区分句末标记（句号）。在后面一节中我们可以看到，如果我们给它机会的话，它能做的其实更多。而且我们已经知道，句号是句子结束的标记还是作为缩写词的一部分，它们之间的差别有时

候也是重要的。有趣的是，有些书面语言，比如象形文字，并不需要空格去分割句子中的单词，而是需要读者（或者机器）来区分边界。

POS标记

这一步把词性（part-of-speech，POS）信息分配给每个单词。在示例解释器会话中，通过另外一步的运行用标签来标识它们：

```
   >>> pos_tagged_tokens = [nltk.pos_tag(t) for t in tokens]
>>> pos_tagged_tokens
[[('Mr.', 'NNP'), ('Green', 'NNP'), ('killed', 'VBD'), ('Colonel', 'NNP'),
  ('Mustard', 'NNP'), ('in', 'IN'), ('the', 'DT'), ('study', 'NN'),
  ('with', 'IN'), ('the', 'DT'), ('candlestick', 'NN'), ('.', '.')],
[('Mr.', 'NNP'), ('Green', 'NNP'), ('is', 'VBZ'), ('not', 'RB'),
 ('a', 'DT'), ('very', 'RB'), ('nice', 'JJ'), ('fellow', 'JJ'),
 ('.', '.')]]
```

你可能无法直观的了解所有这些标签，但是它们确实表示POS信息。例如，'NNP'表示这个单词是作为名词短语一部分的一个名词，'VBD'表示一般过去时中的一个动词，'JJ'表示一个形容词。宾州树库项目（The PennTredank Project）（http://bit.ly/1a1mXqf）提供了可返回的POS标签的全面总结（http://bit.ly/1a1n05o）。随着POS标记的完成，分析显然会变得非常强大。例如，通过使用POS标签，可以把名词组块作为名词短语的一部分，然后试着讨论它们可能是哪类实体（例如，人、地方、组织等）。如果你从未想过应用这些小学中就有的词性练习，仔细想想：这对自然语言处理的正确应用是至关重要的。

分块

这一步包含分析一个句子中所有被标记的单词，聚集表达逻辑概念的复合单词——这是和统计并分析搭配词完全不同的方法。可以通过NLTK的chunk.RegexpParser来定义一种自定义语法，但是这部分内容不在本书的范围内；查看O'Reilly出版的《Natural Language Processing with Pthon》的第9章（http://bit.ly/1a1n0Cl）可以获得全部细节。此外，NLTK还提供了把分块和命名实体提取结合起来的一个函数，它是下一步的内容。

提取

这一步包含每个分块的分析，以及把这些分块进一步标记为命名实体。如人、组织、位置等。解释器中NLP的继续执行如下：

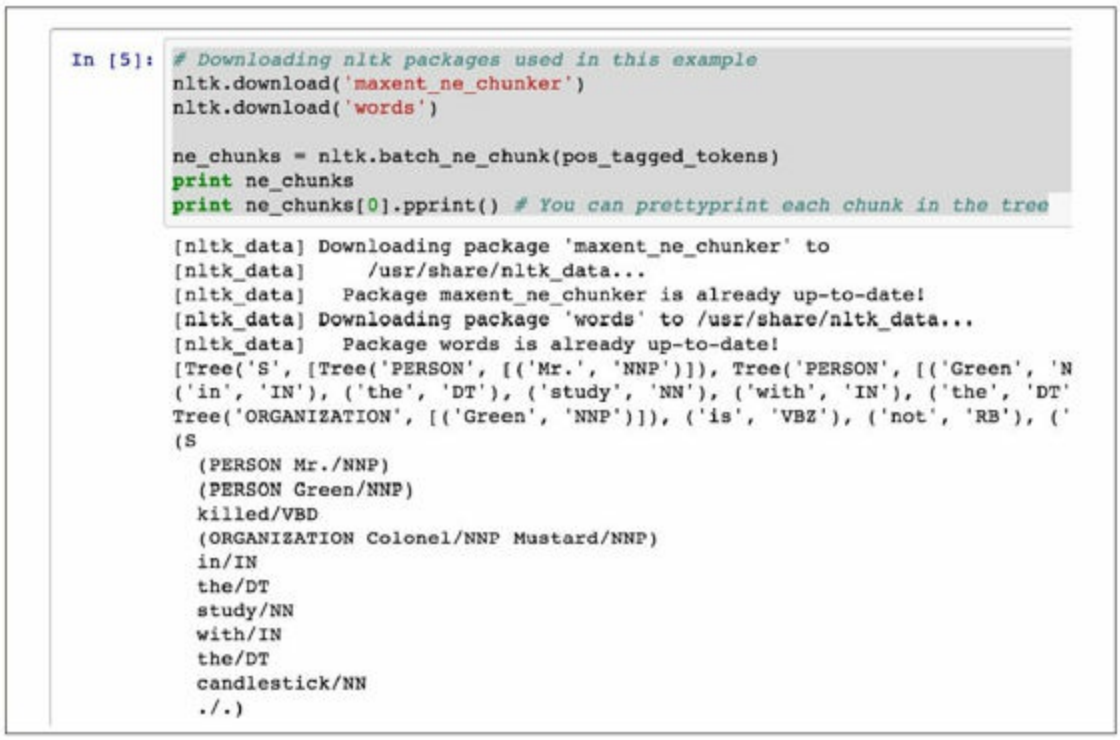
```
   >>> ne_chunks = nltk.batch_ne_chunk(pos_tagged_tokens)
>>> print ne_chunks
[Tree('S', [Tree('PERSON', [('Mr.', 'NNP')]),
Tree('PERSON', [('Green', 'NNP')]), ('killed', 'VBD'),
Tree('ORGANIZATION', [('Colonel', 'NNP'), ('Mustard', 'NNP')]),
    ('in', 'IN'), ('the', 'DT'), ('study', 'NN'), ('with', 'IN'),
    ('the', 'DT'), ('candlestick', 'NN'), ('.', '.')]),
Tree('S', [Tree('PERSON', [('Mr.', 'NNP')]),
                  Tree('ORGANIZATION',
                      [('Green', 'NNP')]), ('is', 'VBZ'), ('not', 'RB'),
                       ('a', 'DT'), ('very', 'RB'), ('nice', 'JJ'),
                       ('fellow', 'JJ'), ('.', '.')])]
>>> print ne_chunks[0].pprint() # You can pretty-print each chunk in the tree
(S
(PERSON Mr./NNP)
(PERSON Green/NNP)
killed/VBD
(ORGANIZATION Colonel/NNP Mustard/NNP)
in/IN
the/DT
study/NN
with/IN
the/DT
candlestick/NN
./.)
```

不要太专注于试图精确解释上面的树输出的含义。简而言之，它把一些标记组成块，并且试着把它们分成某些类型的实体。（你可以看出，它把“Mr.Green”识别为人，遗憾的是，它把“Colonel Mustard”分类为组织。）图5-3说明了在IPython Notebook中的输出。

和继续使用NLTK探索自然语言一样，这种投入程度虽然值得却不是我们在这里真正的目的。这一节是为了让你认识到任务的难度，并鼓励你回顾NLTK book（http://bit.ly/1a1mtAk）或其他丰富的在线资源来深入探索。

我们知道NLTK的一些方面是可以被定制的，除非另有说明，本章剩余的部分都假设你也“按现在的样子”使用NLTK。

有了对NLP的简单介绍，我们可以开始挖掘一些博客数据了。



```
In [5]: # Downloading nltk packages used in this example
        nltk.download('maxent_ne_chunker')
        nltk.download('words')

        ne_chunks = nltk.batch_ne_chunk(pos_tagged_tokens)
        print ne_chunks
        print ne_chunks[0].pprint() # You can prettyprint each chunk in the tree

[nltk_data] Downloading package 'maxent_ne_chunker' to
[nltk_data]     /usr/share/nltk_data...
[nltk_data]   Package maxent_ne_chunker is already up-to-date!
[nltk_data] Downloading package 'words' to /usr/share/nltk_data...
[nltk_data]   Package words is already up-to-date!
[Tree('S', [Tree('PERSON', [('Mr.', 'NNP')]), Tree('PERSON', [('Green', 'N
('in', 'IN'), ('the', 'DT'), ('study', 'NN'), ('with', 'IN'), ('the', 'DT'
Tree('ORGANIZATION', [('Green', 'NNP')]), ('is', 'VBZ'), ('not', 'RB'), ('
(S
  (PERSON Mr./NNP)
  (PERSON Green/NNP)
  killed/VBD
  (ORGANIZATION Colonel/NNP Mustard/NNP)
  in/IN
  the/DT
  study/NN
  with/IN
  the/DT
  candlestick/NN
  ./.)
```

图5-3：NLTK拥有图形界面，这样你就能更加直观地在图形化界面中检查分块的输出，而不是你在解释器中看到的原始数据那样

5.3.2 人类语言数据的句子检测

当创建NLP栈时，句子检测可能是你要考虑的第一个任务，所以从创建栈开始就是非常有意义的。即使你没有完成流程中剩余的任务，只靠EOS检测也能得到一些不错的结果（如文档摘要），我们将会在下一节中把它作为后续练习。但是首先，我们需要抓取一些整洁的人类语言数据。让我们使用行之有效的feedparser包，还有前几章介绍的基于nltk和BeautifulSoup的工具，来整理从O’Reilly Radar博客

（http://oreil.ly/1a1n3Oz）抓取的HTML格式的内容。示例5-4的清单抓取了一些文章，并把它们以JSON格式保存在本地文件中。

示例5-4：通过解析源来获取博客数据

```
   import os
import sys
import json
import feedparser
from BeautifulSoup import BeautifulStoneSoup
fromn ltk import clean_html
FEED_URL = 'http://feeds.feedburner.com/oreilly/radar/atom'
def cleanHtml(html):
    return BeautifulStoneSoup(clean_html(html),
                convertEntities=BeautifulStoneSoup.HTML_ENTITIES).contents[0]
fp = feedparser.parse(FEED_URL)
print "Fetched %sentries from '%s'" % (len(fp.entries[0].title), fp.feed.title)
blog_posts = []
for e in fp.entries:
    blog_posts.append({'title': e.title, 'content'
                       : cleanHtml(e.content[0].value), 'link': e.links[0].href})
out_file = os.path.join('resources', 'ch05-webpages', 'feed.json')
f = open(out_file, 'w')
f.write(json.dumps(blog_posts, indent=1))
f.close()
print 'Wrote output file to %s' % (f.name, )
```

从有信誉的来源获取人类语言数据可以给我们提供假设良好的英语语法优势；但愿这也意味着NLTK的即装即用句子检测器可以正常运行。找不到比代码更好的方法来看看到底发生了什么，所以让我们继续前进，看一下示例5-5中的代码清单。它提供了sent_tokenize和word_tokenize方法，它们分别是NLTK目前推荐的句子检测器（sentence detector）和分词器（word tokenizer）的别名。稍后我们会简单讨论一下这个清单。

示例5-5：使用NLTK的NLP工具处理博客数据中的人类语言

```
   import json
import nltk
# Download nltk packages used in this example
nltk.download('stopwords')
BLOG_DATA = "resources/ch05-webpages/feed.json"
blog_data = json.loads(open(BLOG_DATA).read())
# Customize your list of stopwords as needed. Here, we add common
# punctuation and contraction artifacts.
stop_words = nltk.corpus.stopwords.words('english') + [
    '.',
    ',',
    '--',
    '\'s',
    '?',
    ')',
    '(',
    ':',
    '\'',
    '\'re',
    '"',
    '-',
    '}',
    '{',
    u'-',
    ]
for post in blog_data:
    sentences = nltk.tokenize.sent_tokenize(post['content'])
    words = [w.lower() for sentence in sentences for w in
             nltk.tokenize.word_tokenize(sentence)]
    fdist = nltk.FreqDist(words)
    # Basic stats
    num_words = sum([i[1] for i in fdist.items()])
    num_unique_words = len(fdist.keys())
    # Hapaxes are words that appear only once
    num_hapaxes = len(fdist.hapaxes())
    top_10_words_sans_stop_words = [w for w in fdist.items() if w[0]
                                    not in stop_words][:10]
    print post['title']
    print '\tNum Sentences:'.ljust(25), len(sentences)
    print '\tNum Words:'.ljust(25), num_words
    print '\tNum Unique Words:'.ljust(25), num_unique_words
    print '\tNumHapaxes:'.ljust(25), num_hapaxes
    print '\tTop 10 Most Frequent Words (sans stop words):\n\t\t',\
            '\n\t\t'.join(['%s(%s)'
            % (w[0], w[1]) for w in top_10_words_sans_stop_words])
    print
```

你想知道的第一件事可能是sent_tokenize和word_tokenize调用。NLTK提供了一些“分词”选项，它通过这些别名提供了最好的“建议”。在写作这部分内容时（在任何时候你可以使用在IPython或IPythonNoteBook中的pydoc或者像命令行那样的

nltk.tokenize.sent_tokenize？再次确认一下），句子检测器是PunktSentenceTokenizer，分词器是TreebankWordTokenizer。让我们简单地介绍一下它们吧！

PunktSentenceTokenizer在很大程度上依赖能够把缩写词检测为搭配模式的一部分，它通过考虑标点符号用法的公用模式，使用一些正则表达式来试着智能的解析句子。对PunktSentenceTokenizer内部结构逻辑的完全解释超出了本书的范围，但是Tibor Kiss和Jan Strunk的学术性论文“Unsupervised Multilingual Sentence Boundary Detection”（http://bit.ly/1a1n3OI）讨论了它的方法，具有很强的可读性，它值得你花点时间来仔细阅读一下。

正如我们一会儿看到的，使用尝试提高它精确度的示例文本来实例化PunktSentenceTokenizer是可能的；所使用的基础算法的类型是非监督式学习算法（unsupervised learning algorithm）；它不要求你以任何方式显示标记示例训练数据。相反，该算法检查出现在文本本身的某些特性，如大写字母的使用、共同出现的标记等，来获取将文本分解成句子的合适参数。

虽然NLTK的WhitespaceTokenizer会是我们介绍的最简单的分词器，它通过按空格分解文本创建单词，但是你已经熟悉盲目按空格分解的缺点了。相反，NLTK目前推荐TreebankWordTokenizer作为分词器，它对句子进行操作并使用与宾州书库项目（http://bit.ly/1a1mXqf）相同

的约定[2]。可能会让你措手不及的一件事是TreebankWordTokenizer的分词器（http://bit.ly/1a1n4ly）会做一些不太明显的事，如在缩写和所有格形式的名词上分别标记要素。例如，分析句子“I’m hungry”，将会产生不同的要素“I”和“m”，保留主语和“I’m”动词之间的区别。和你想象的一样，是时候开始仔细检查句子中主语和动词之间关系的高级分析了，对这类语法信息的“细粒度”访问可能非常有价值。

已知句子解析器（sentence tokenizer）和分词器，首先可以把文本解析为句子，然后把每个句子都解析为单词（token）。虽然这种方法非常直观，但是它有一个致命点，因为句子检测器产生的错误会向前传播，可能会约束其他NLP栈产生的质量上限。例如，如果句子解析器错误地分解了出现在“Mr.Green killed Colonel Mustard in the study with the candlestick”这段文本中的“Mr.”后面某个句子的句号，那么可能无法从文本中提取出“Mr.Green”实体，除非存在专门的修复逻辑。它完全依赖整个NLP栈的文本复杂度，以及它如何解释误差传播。

即装即用的PunktSentenceTokenizer是在宾州树库项目语料库中训练的，可以良好执行。语法分析的最终目标是实例化一个便捷的nltk.FreqDist对象（很像稍微复杂些的collections.Counter），该对象需要一个单词列表。示例5-5中剩余的代码是一些常用的NLTK API的简单的用法。

注意：如果你在使用先进的分词器（如NLTK的

TreebankWordTokenizer或PunktWordTokenizer）方面有很多困难，那么最好使用默认的WhitespaceTokenizer，直到你确定是否值得投入使用更先进的分词器为止。事实上，有时候使用更简单的分词器可能很有利。例如，对经常内联URL的数据使用先进的分词器可能不是一个好主意。

这一节旨在让你熟悉创建NLP流程的第一步。与此同时，我们开发了一些试图表示博客数据的度量。我们的流程并不包含词性标记或分块，但是它可以让你基本了解一些概念，让你思考包含的某些细微问题。虽然我们确实可以简单地在空格上分割、统计词数、统计结果和从数据中获取很多信息，但是不久以后你就会很高兴你采取这些初始步骤来深入了解数据。为了说明你刚才学到的一种可能的应用，下一节会介绍一个简单的文档摘要算法，它只依赖于句子分割和频率分析。

5.3.3 文档摘要

执行相当好的句子检测作为挖掘非结构化数据的NLP方法的一部分，使得一些非常强大的文本挖掘功能成为可能，如对文档摘要的虽然不成熟但是合理的尝试。有很多可能性和方法，但是全面开始使用数据的其中一种最简单的方法来自《IBM研究与发展年报》（IBM Journal）1958年4月的专题。在“The Automatic Creation of Literature Abstracts”（http://bit.ly/1a1n4Cj）这篇文章中，H.P.Luhn描述了归结为

过滤句子的技术，这些句子包含最常出现的单词，这些单词是彼此邻近的。

这篇学术论文很容易理解，而且非常有趣；Luhn实际上描述了它如何准备穿孔卡，以便使用不同的参数运行各种测试！令人惊讶的是在一个廉价的商业硬件中，我们用几十行Python代码可以实现什么，他可能用了好长时间把这段代码编写到庞大的主机中。示例5-6提供了文档摘要的Luhn算法的基本实现。下一节会对该算法进行简要分析。在跳到这个讨论之前，让我们先花一点时间浏览这段代码，看看你是否能确定它的工作原理。

注意：示例5-6使用numpy包（高度优化的数字运算的集合），它应该和nltk一起安装。如果因为某些原因你没有使用虚拟机并且需要安装它，使用pip install numpy命令即可。

示例5-6：基于句子检测和句中频率分析的文档摘要算法

```
   import json
import nltk
import numpy
BLOG_DATA = "resources/ch05-webpages/feed.json"
N = 100  # Number of words to consider
CLUSTER_THRESHOLD = 5  # Distance between words to consider
TOP_SENTENCES = 5  # Number of sentences to return for a "top n" summary
# Approach taken from "The Automatic Creation of Literature Abstracts" by H.P. Luhn
def _score_sentences(sentences, important_words):
    scores = []
    sentence_idx = -1
    for s in [nltk.tokenize.word_tokenize(s) for s in sentences]:
        sentence_idx += 1
        word_idx = []
        # For each word in the word list...
        for w in important_words:
            try:
```

```
                # Compute an index for where any important words occur in the
sentence.
                word_idx.append(s.index(w))
            exceptValueError, e: # w not in this particular sentence
                pass
        word_idx.sort()
        # It is possible that some sentences may not contain any important words at
all.
        iflen(word_idx)== 0: continue
        # Using the word index, compute clusters by using a max distance threshold
        # for any two consecutive words.
        clusters = []
        cluster = [word_idx[0]]
        i = 1
        whilei<len(word_idx):
            if word_idx[i] - word_idx[i - 1] < CLUSTER_THRESHOLD:
                cluster.append(word_idx[i])
            else:
                clusters.append(cluster[:])
                cluster = [word_idx[i]]
            i += 1
        clusters.append(cluster)
        # Score each cluster. The max score for any given cluster is the score
        # for the sentence.
        max_cluster_score = 0
        for c in clusters:
            significant_words_in_cluster = len(c)
            total_words_in_cluster = c[-1] - c[0] + 1
            score = 1.0 * significant_words_in_cluster\
                * significant_words_in_cluster / total_words_in_cluster
            if score >max_cluster_score:
                max_cluster_score = score
        scores.append((sentence_idx, score))
    return scores
def summarize(txt):
    sentences = [s for s in nltk.tokenize.sent_tokenize(txt)]
    normalized_sentences = [s.lower() for s in sentences]
    words = [w.lower() for sentence in normalized_sentences for w in
             nltk.tokenize.word_tokenize(sentence)]
    fdist = nltk.FreqDist(words)
    top_n_words = [w[0] for w in fdist.items()
            if w[0] not in nltk.corpus.stopwords.words('english')][:N]
    scored_sentences = _score_sentences(normalized_sentences, top_n_words)
    # Summarization Approach 1:
    # Filter out nonsignificant sentences by using the average score plus a
    # fraction of the std dev as a filter
    avg = numpy.mean([s[1] for s in scored_sentences])
    std = numpy.std([s[1] for s in scored_sentences])
    mean_scored = [(sent_idx, score) for (sent_idx, score) in scored_sentences
                   if score >avg + 0.5 * std]
    # Summarization Approach 2:
    # Another approach would be to return only the top N ranked sentences
    top_n_scored = sorted(scored_sentences, key=lambda s: s[1])[-TOP_SENTENCES:]
    top_n_scored = sorted(top_n_scored, key=lambda s: s[0])
    # Decorate the post object with summaries
    returndict(top_n_summary=[sentences[idx] for (idx, score) in top_n_scored],
               mean_scored_summary=[sentences[idx] for (idx, score) in mean_scored])
blog_data = json.loads(open(BLOG_DATA).read())
for post in blog_data:
    post.update(summarize(post['content']))
    print post['title']
```

```
    print '=' * len(post['title'])
    print
    print 'Top N Summary'
    print '-------------'
    print ' '.join(post['top_n_summary'])
    print
    print 'Mean Scored Summary'
    print '-------------------'
    print ' '.join(post['mean_scored_summary'])
    print
```

我们会使用Tim O’Reilly的Radar文章“The Louvre of the Industrial Age”（http://oreil.ly/1a1n4SO）来作为示例输入/输出。它大约有460个单词，在这里转载了它，这样就可以比较清单中两个摘要尝试的示例输出了：

This morning I had the chance to get a tour of The Henry Ford Museum in Dearborn，MI，along with Dale Dougherty，creator of Make：and Makerfaire，and Marc Greuther，the chief curator of the museum.I had expected a museum dedicated to the auto industry，but it’s so much more than that.As I wrote in my first stunned tweet，“it’s the Louvre of the Industrial Age.”

When we first entered，Marc took us to what he said may be his favorite artifact in the museum，a block of concrete that contains Luther Burbank’s shovel，and Thomas Edison’s signature and footprints.Luther Burbank was，of course，the great agricultural inventorwho created such treasures as the nectarine and the Santa Rosa plum.Ford was a farm boy who became an industrialist；Thomas Edison was his friend and mentor.The

museum，opened in 1929，was Ford’s personal homage to the transformation of the world that he was so much a part of.This museum chronicles that transformation.

The machines are astonishing—steam engines and coal-fired electric generators as big as houses，the first lathes capable of making other precision lathes（the makerbot of the 19th century），a ribbon glass machine that is one of five that in the 1970s made virtually all of the incandescent lightbulbs in the world，combine harvesters，railroad locomotives，cars，airplanes，even motels，gas stations，an early McDonalds'restaurant and other epiphenomena of the automobile era.

Under Marc’s eye，we also saw the transformation of the machines from purely functional objects to things of beauty.We saw the advances in engineering—the materials，the workmanship，the design，over a hundred years of innovation.Visiting The Henry Ford，as they call it，is a truly humbling experience.I would never in a hundred years have thought of making a visit to Detroit just to visit this museum，but knowing what I know now，I will tell you confidently that it is as worth your while as a visit to Paris just to see the Louvre，to Rome for the Vatican Museum，to Florence for the Uffizi Gallery，to St.Petersburg for the Hermitage，or to Berlin for the Pergamon Museum.This is truly one of the world’s great museums，and

the world that it chronicles is our own.

I am truly humbled that the Museum has partnered with us to hold Makerfaire Detroit on their grounds.If you are anywhere in reach of Detroit this weekend，I heartily recommend that you plan to spend both days there.You can easily spend a day at Makerfaire，and you could easily spend a day at The Henry Ford.P.S.Here are some of my photos from my visit.（More to come soon.Can't upload many as I'm currently on a plane.）

使用平均分和标准差过滤句子，会产生大约170个单词的摘要：

This morning I had the chance to get a tour of The Henry Ford Museum in Dearborn，MI，along with Dale Dougherty，creator of Make：and Makerfaire，and Marc Greuther，the chief curator of the museum.I had expected a museum dedicated to the auto industry，but it’s so much more than that.As I wrote in my first stunned tweet，it's the Louvre of the Industrial Age.This museum chronicles that transformation.The machines are astonishing-steam engines and coal fired electric generators as big as houses，the first lathes capable of making other precision lathes（the makerbot of the 19th century），a ribbon glass machine that is one of five that in the 1970s made virtually all of the incandescent lightbulbs in the world，combine harvesters，railroad locomotives，cars，airplanes，even motels，gas stations，an early McDonalds？restaurant and other

epiphenomena of the automobile era.You can easily spend a day at Makerfaire，and you could easily spend a day at The Henry Ford.

另一种摘要算法会产生约90个单词的缩略结果，它只考虑了前N个句子（在本例中，N=5）。它更简洁，但是仍然可以说它是一个很好的信息提取：

This morning I had the chance to get a tour of The Henry Ford Museum in Dearborn，MI，along with Dale Dougherty，creator of Make：and Makerfaire，and Marc Greuther，the chief curator of the museum.I had expected a museum dedicated to the auto industry，but it’s so much more than that.As I wrote in my first stunned tweet，it’s the Louvre of the Industrial Age.This museum chronicles that transformation.You can easily spend a day at Makerfaire，and you could easily spend a day at The Henry Ford.

和其他涉及分析的情况一样，可以从与全文有关的摘要的视觉检查中了解它。

输出几乎可以被所有Web浏览器打开的标记格式像调整该代码最后一部分一样简单，它通过执行输出来替换字符串。示例5-7说明了一种全文对文档摘要输出可视化的可能性，它的方法是展示全文时，如果是摘要中的句子就作为黑体出现，这样就能很轻易地看出来是不是摘要中

的句子了。这个脚本将HTML文件的集合保存在磁盘中，这样你就能在IPython Notebook中进行查看或者在浏览器中打开而不需要服务器。

示例5-7：用HTML输出将文档摘要结果可视化

```
   import os
import json
import nltk
import numpy
from IPython.display import IFrame
from IPython.core.display import display
BLOG_DATA = "resources/ch05-webpages/feed.json"
HTML_TEMPLATE = """<html>
    <head>
        <title>%s</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    </head>
    <body>%s</body>
</html>"""
blog_data = json.loads(open(BLOG_DATA).read())
for post in blog_data:
    # Uses previously defined summarize function.
    post.update(summarize(post['content']))
    # You could also store a version of the full post with key sentences marked up
    # for analysis with simple string replacement...
    for summary_type in ['top_n_summary', 'mean_scored_summary']:
        post[summary_type + '_marked_up'] = '<p>%s</p>' % (post['content'], )
        for s in post[summary_type]:
        post[summary_type + '_marked_up'] =\
        post[summary_type + '_marked_up'].replace(s, '<strong>%s</strong>' % (s, ))
    filename = post['title'].replace("?", "") + '.summary.' + summary_type + '.html'
    f = open(os.path.join('resources', 'ch05-webpages', filename), 'w')
    html = HTML_TEMPLATE % (post['title'] +\
        ' Summary', post[summary_type + '_marked_up'],)
    f.write(html.encode('utf-8'))
    f.close()
    print "Data written to", f.name
# Display any of these files with an inline frame. This displays the
# last file processed by using the last value of f.name...
print "Displaying %s:" % f.name
display(IFrame('files/%s' % f.name, '100%', '600px'))
```

输出结果是文档的全文，其中组成摘要的句子以粗体形式高亮表示，如图5-4所示。和探讨摘要的替代技术一样，浏览器标签之间的快速浏览可以让你直观地了解摘要技术之间的相似性。这里说明的主要差

别是文档中间很长（和描述性）的句子，以“The machines are astonishing”开始。

下一节是关于Luhn算法的简短讨论。

5.3.3.1 对Luhn摘要算法的分析

注意：这一章提供了对Luhn摘要算法的分析。它的目标是扩大你对人类语言处理技术的理解，但并不是挖掘社交网络所必需的。如果你觉得你在细节中有些迷茫，可以先跳过本节，之后再来阅读它。

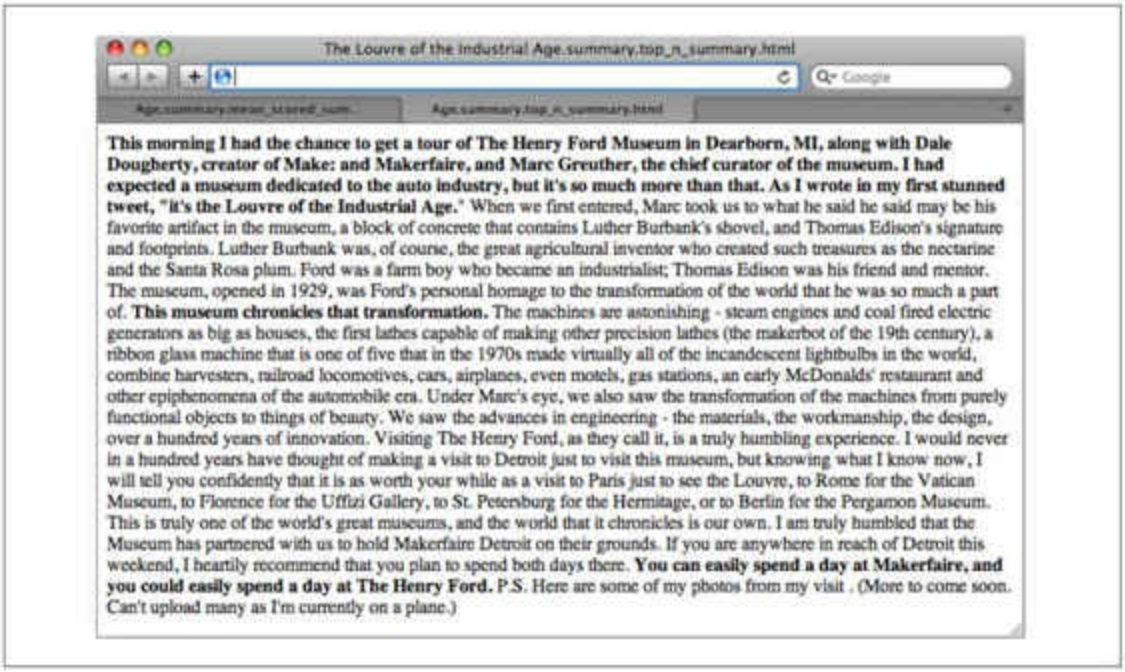


图5-4：O’Reilly Radar博文中的文本的可视化，其中摘要算法确定的重点句子由粗体字表示

Luhn算法的基本前提是文档中的重点句子包含经常出现的单词的句

子。不过还是值得指出一些细节。首先，并不是所有经常出现的单词都是重要的；一般来说，停用词是填充器，一般不值得分析。请记住，虽然我们确实可以在样例实现中过滤常见的停用词，但是对于给定的博客或范围，使用附加先验知识创建一个停用词自定义列表是可能的，这可能会进一步增强这个算法或假设已经过滤了停用词的其他算法的效力。例如，专门谈论棒球的博客可能会经常使用“棒球”这个词，应该考虑将它加入一个停用词列表中，即使它不是通用的停用词。（作为旁注，对某个数据源，把TF-IDF合并到计分函数中作为说明这个领域中常用词用法的方法会很有趣。）

假设已经尝试消除了停用词，算法的下一步是选择合适的N值，选择前N个词作为分析的基础。这个算法潜在的假设是这前N个词能充分的描述文档的特性，对于文档中的任何两个句子来说，包含更多这些单词的句子会被认为更具描述性。确定了文档中的“重点词”后，剩余的工作就是对每个句子应用启发式方法，过滤句子的一些子集来把它作为文档的概要或摘要。需要使用score_sentences函数来计算每个句子的得分。这是代码清单中最有趣的部分。

为了计算每个句子的得分，score_sentences中的算法对聚类单词应用了简单的距离阈值，根据下面的公式计算每个聚类的得分：

$$\frac{(\text{聚类中有意义的词})^2}{\text{聚类中的总词数}}$$

每个句子的最终得分与出现在句子中的任何聚类的最高得分相等。让我们考虑一个例句的score_sentences中涉及的大体步骤，来看看该方法是如何执行的：

输入：例句

```
   ['Mr.', 'Green', 'killed', 'Colonel', 'Mustard', 'in', 'the',
'study', 'with', 'the', 'candlestick', '.']
```

输入：重点词列表

```
   ['Mr.', 'Green', 'Colonel', 'Mustard', 'candlestick']
```

输入/假设：聚类阈值（距离）

```
   3
```

中间计算：聚类检测

```
   [ ['Mr.', 'Green', 'killed', 'Colonel', 'Mustard'], ['candlestick'] ]
```

中间计算：聚类得分

```
   [ 3.2, 1 ] # Computation: [ (4*4)/5, (1*1)/1]
```

输出：句子得分

```
   3.2 # max([3.2, 1])
```

在score_sentences中完成的实际工作只是记账来检测句子中的聚类。一个聚类被定义为包含两个或更多重要词的句子，其中每个重要词都是与其最近的邻居的距离阈值。虽然Luhn的论文建议距离阈值取为4或5，但为了简单起见，我们在示例中用了3；因此，'Green'和'Colonel'之间的距离被充分的桥接起来了，第一个检测的阈值由句子中的前5个词组成。'study'也出现在了重要词列表中，整个句子都作为聚类（除了最后的标点符号）。

计算每个句子的得分之后，剩下的就是确定把哪些句子作为原文摘要返回。样例实现提供了两种方法。第一种方法通过计算得分的均值和标准差，使用统计阈值来过滤句子，而后一种方法只是简单的返回了前N个句子。根据数据的性质，里程可能会有所不同，但是使用任意一种方法，你都应该可以调整参数来得到合理的结果。使用前N个句子的好处是你会深入了解摘要的最大长度。如果很多句子的得分相对接近，使用均值和标准差可能会返回更多的句子。

Luhn的算法容易实现，对被描述为整个文档中经常出现的词有效。然后，请记住，和前面的章节讨论的很多IR方法一样，Luhn算法并没有尝试从更深的语义层次上理解数据——虽然它确实不仅仅依赖于“词袋”。它将直接计算摘要作为经常出现词的函数，而且它在如何计算句子得分方面也并不十分复杂，但是（和TF-IDF的情况一样），这使得可执行的以及似乎可以在随机抽取的博客数据中执行操作更惊人。

当你权衡实现一个更复杂的方法的利弊时，值得反思一下改进Luhn算法产生的这类合理摘要需要的工作量。有时候，一个不成熟的启发式方法是你实现你的目标真正需要的。然而，平时你可能需要一些更新的技术。棘手的部分是计算从不成熟的启发式方法到最新的解决方案的成本效益分析。我们很多人往往会对包含的相对工作量过于乐观。

[1] 同音异义词是同形异义词的特殊情况。如果两个词拼写相同，它们就是同形异义词。如果两个词拼写和发音都相同，它们就是同音异义词。由于某种原因，即使是“同形异义词”的误用，语法中更常见的还是“同音异义词”。

[2] “treebank”（树库）是一个专有名词，指的是使用先进的语言信息专门标记的一个语料库。事实上，把这个语料库称为“树库”的原因是为了强调它是由很多句子组成的一个库（集合），已经把它们解析为遵循特定语法的树。

5.4 以实体为中心的分析：范式转换

本章一直在暗示：更深入了解数据的分析方法可能比只是把每个单词作为不透明符号的方法更强大。但是对数据的“更深入的了解”到底是什么意思呢？

一种解释是能够检测文档中的实体，并以这些实体作为分析的基础，而不是以文档为中心的分析方法，其中包含关键字搜索或将搜索输入解释成特定类型的实体并相应地定制结果。虽然你没有从这方面考虑过它，但这恰恰是WolframAlpha这类新兴技术在表示层所做的。例如，在WolframAlpha中搜索“tim O’Reilly”的结果返回表示了“搜索的实体是一个人”这样的理解，不是只返回一个包含关键字的文档列表（参阅图5-5）。无论使用哪种内部技术来实现这个目标，用户体验都会更好，因为这些结果更符合用户期望的格式。

尽管我们在当前的讨论中，无法考虑所有以实体为中心的分析的各种可能性，介绍一种从文档中提取实体的方法还是非常合适的，稍后会用它做各种分析。假设使用本章前面介绍的NLP示例流程，你可以轻易地从文档中提取所有的名词和名词短语——重要的潜在假设是名词和名词短语（或是一些精心构造的子集）可以作为我们感兴趣的实体。正如后面的示例清单所表达的那样，这确实是合理的假设，而且是以实体为

中心的分析的良好起点。请注意，按照宾州树库约定注解的结果，以“NN”开头的任何标记都是名词或名词短语。宾州树库标签（http://bit.ly/1a1n59j）的完整清单可以在线获得。

示例5-8分析了应用于单词以及识别作为实体的名词和名词短语。以数据挖掘术语来说，依赖于你想完成的细微差别，在文档中寻找实体称为实体提取或命名实体识别。

示例5-8：使用NLTK从文本中提取实体

```
   import nltk
import json
BLOG_DATA = "resources/ch05-webpages/feed.json"
blog_data = json.loads(open(BLOG_DATA).read())
for post in blog_data:
    sentences = nltk.tokenize.sent_tokenize(post['content'])
    tokens = [nltk.tokenize.word_tokenize(s) for s in sentences]
    pos_tagged_tokens = [nltk.pos_tag(t) for t in tokens]
```

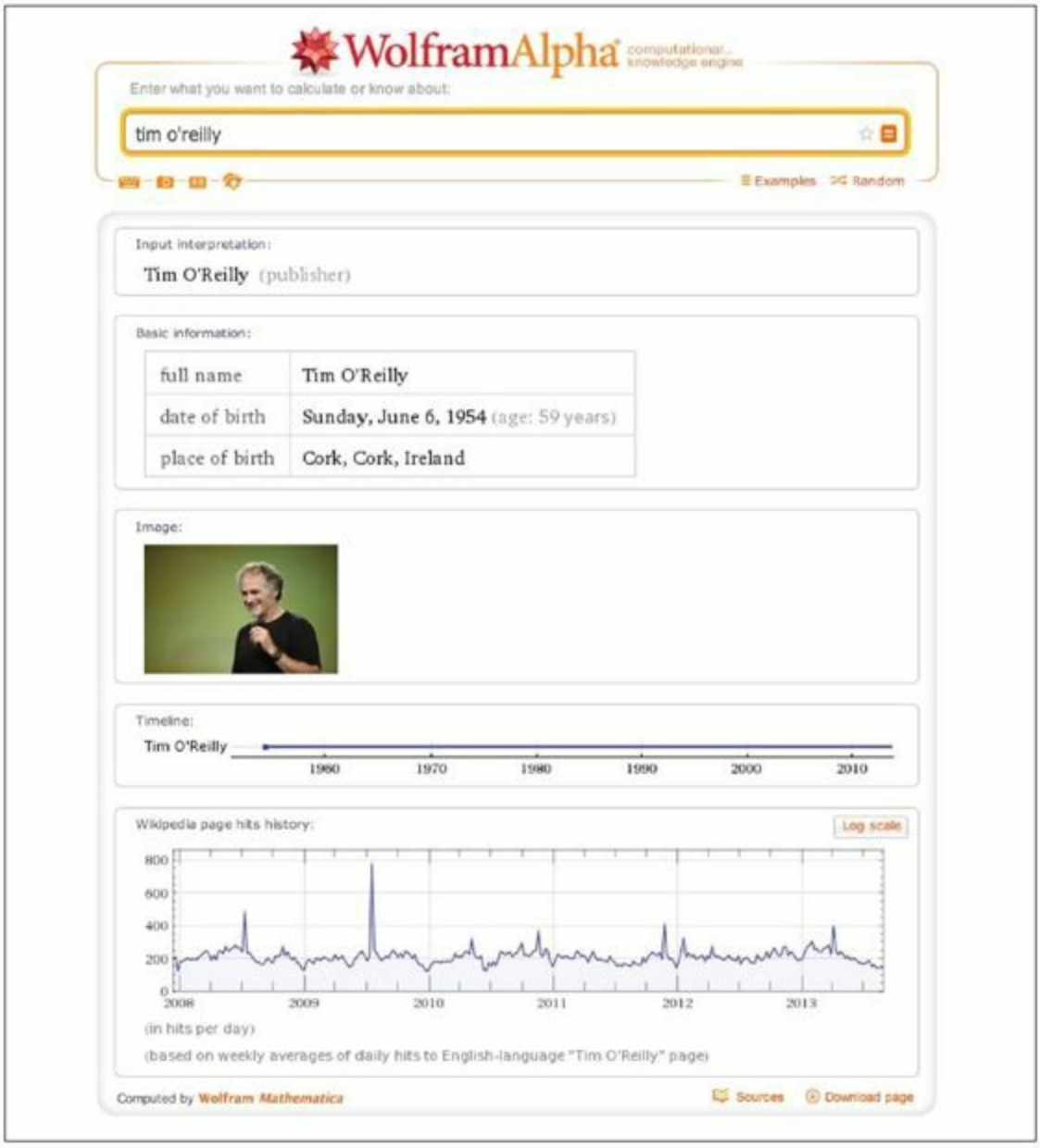


图5-5：使用WolframAlpha查询“tim o’reilly”的示例结果

```
   # Flatten the list since we're not using sentence structure
# and sentences are guaranteed to be separated by a special
# POS tuple such as ('.', '.')
pos_tagged_tokens = [token for sent in pos_tagged_tokens for token in sent]
all_entity_chunks = []
previous_pos = None
current_entity_chunk = []
for (token, pos) in pos_tagged_tokens:
    if pos == previous_pos and pos.startswith('NN'):
        current_entity_chunk.append(token)
    elifpos.startswith('NN'):
        if current_entity_chunk != []:
```

```
                # Note that current_entity_chunk could be a duplicate when appended,
                # so frequency analysis again becomes a consideration
                all_entity_chunks.append((' '.join(current_entity_chunk), pos))
            current_entity_chunk = [token]
        previous_pos = pos
    # Store the chunks as an index for the document
    # and account for frequency while we're at it...
    post['entities'] = {}
    for c in all_entity_chunks:
        post['entities'][c] = post['entities'].get(c, 0) + 1
    # For example, we could display just the title-cased entities
    print post['title']
    print '-' * len(post['title'])
    proper_nouns = []
    for (entity, pos) in post['entities']:
        if entity.istitle():
            print '\t%s(%s)' % (entity, post['entities'][(entity, pos)])
    print
```

注意：从5.3.1节中对“提取”的描述可以看到，NLTK提供了nltk.batch_ne_chunk函数来尝试从POS标记的单词中提取命名实体。欢迎你直接使用这个功能，但你可能会发现，你得到的好处可能会随着NLTK实现时提供的即装即用的模块而变化。

示例5-8的输出如下所示，并传达了可以被用在很多方面的有意义的结果。例如，它们通过像WordPress插件这样的智能博客平台来给出对标签的建议。

```
   The Louvre of the Industrial Age
--------------------------------
    Paris (1)
    Henry Ford Museum (1)
    Vatican Museum (1)
    Museum (1)
    Thomas Edison (2)
    Hermitage (1)
    Uffizi Gallery (1)
    Ford (2)
    Santa Rosa (1)
    Dearborn (1)
    Makerfaire (1)
    Berlin (1)
    Marc (2)
    Makerfaire (1)
    Rome (1)
```

```
Henry Ford (1)
Ca (1)
Louvre (1)
Detroit (2)
St. Petersburg (1)
Florence (1)
Marc Greuther (1)
Makerfaire Detroit (1)
Luther Burbank (2)
Make (1)
Dale Dougherty (1)
Louvre (1)
```

统计学结果通常有不同的目的和受众。一个文本摘要是为了阅读，而像前面那样提取的实体列表，是用来快速浏览寻找模式的。对于比这个例子更大的语料，标签云（http://bit.ly/1a1n5pO）是可视化数据的明显的结果。

注意：尝试从网页http://oreil.ly/1a1n4SO抓取文本来重现这个结果。

通过更加盲目的分析句子中词法特点（如大写的使用），我们就已经算是发现了与专有名词相同的列表了吗？也许是这样，但请记住，该技术也可以抓取没有明确表明大小写的名词和名词短语。大小写确实是文本中值得探讨的重要特征，但示例文本中也有全部是小写字母的实体（比如“chief curator”、“locomotives”和“lightbulbs”）。

尽管实体列表并不能像我们之前计算的摘要那样有效的传达文本的大意，标识这些实体对分析极其重要，因为它们有语义层次的含义，而且不是经常出现的词。事实上，在示例输出中出现的大多数专有名词的频率都很低。尽管如此，它们还是很重要的，因为它们在文本中有实质

的含义，即它们是人、地点、事情或想法，通常都是数据中的实质性信息。

5.4.1 领会人类语言数据

现在把动词考虑进去并计算主谓宾这个三元组是向前迈出的又一个重要步骤，这样你就能知道哪些实体和哪些实体交互，以及这些交互的本质。这些三元组可以帮助文档的对象图可视化，比起原本的文档，我们可以更快的浏览。更好的是，想象采用来自文档集中的多幅对象图，并合并它们来获得更大的语料库。这项技术是非常活跃的研究领域，并且几乎能够应用于任何信息过载的问题。但需要说明的是，一般情况下，它是一个极其折磨人的问题。

假设一个POS标签器已经从句子中标识出以下词性，并生成如下输出：[（'Mr.'，'NNP'），（'Green'，'NNP'），（'killed'，'VBD'），（'Colonel'，'NNP'），（'Mustard'，'NNP'），...]，存储（'Mr.Green'，'killed'，'Colonel Mustard'）这种格式的主谓宾的三元祖的索引很容易计算。然而，这种情况的实质是使用这种简单的级别，你不太可能遇到真正的POS标记数据——除非你是想挖掘少儿读物（对于初学者来说，这并不是一个坏主意）。例如，考虑用NLTK产生的本章前面的博文的第一句话的标签，并实现你想转化为对象图的部分数据：

This morning I had the chance to get a tour of The Henry Ford Museum in Dearborn，MI，along with Dale Dougherty，creator of Make：and Makerfaire，and Marc Greuther，the chief curator of the museum.

你想从上面的句子中提取出来的最简单的三元组可能是（'I'，'get'，'tour'），但即使你得到这个结果，也不能说明Dale Dougherty也参加了这次参观，或者Marc Greuther也参加了。POS标记的数据也能很清楚地说明得到这些解释并不那么简单，因为这句话有着非常丰富的结构：

```
   [(u'This', 'DT'), (u'morning', 'NN'), (u'I', 'PRP'), (u'had', 'VBD'),
(u'the', 'DT'), (u'chance', 'NN'), (u'to', 'TO'), (u'get', 'VB'),
(u'a', 'DT'), (u'tour', 'NN'), (u'of', 'IN'), (u'The', 'DT'),
(u'Henry', 'NNP'), (u'Ford', 'NNP'), (u'Museum', 'NNP'), (u'in', 'IN'),
(u'Dearborn', 'NNP'), (u',', ','), (u'MI', 'NNP'), (u',', ','),
(u'along', 'IN'), (u'with', 'IN'), (u'Dale', 'NNP'), (u'Dougherty', 'NNP'),
(u',', ','), (u'creator', 'NN'), (u'of', 'IN'), (u'Make', 'NNP'), (u':', ':'),
(u'and', 'CC'), (u'Makerfaire', 'NNP'), (u',', ','), (u'and', 'CC'),
(u'Marc', 'NNP'), (u'Greuther', 'NNP'), (u',', ','), (u'the', 'DT'),
(u'chief', 'NN'), (u'curator', 'NN'), (u'of', 'IN'), (u'the', 'DT'),
(u'museum', 'NN'), (u'.', '.')]
```

这种情况下，高质量的开源NLP工具可能会产生有意义的三元组是值得怀疑的，考虑到谓语“had a chance to get a tour”的复杂特性，参与这次参观的其他人被列在了句子最后的短语中。

注意：如果你想寻求构造这些三元组的方法，你应该使用较为准确的POS标记信息作为最初的尝试。操作人类语言数据的高级任务还有很多，但结果都很令人满意，并且都可能产生令人欣喜若狂的结果（好的方面）。

好消息是，正如前面说的那样，通过从文本中提取实体并把它们作为分析的基础，你能做许多有趣的事情。你能够轻易地从以每个句子为基础的文本中产生三元组，其中每个三元组的“谓语”是表示主语和宾语“交互”的语言关系的概念。示例5-9是对示例5-8的重构，它收集了以每个句子为基础的实体，这对计算使用作为上下文窗口的句子的实体的交互非常有用。

示例5-9：查找实体间的交互

```
   import nltk
import json
BLOG_DATA = "resources/ch05-webpages/feed.json"
def extract_interactions(txt):
    sentences = nltk.tokenize.sent_tokenize(txt)
    tokens = [nltk.tokenize.word_tokenize(s) for s in sentences]
    pos_tagged_tokens = [nltk.pos_tag(t) for t in tokens]
    entity_interactions = []
    for sentence in pos_tagged_tokens:
        all_entity_chunks = []
        previous_pos = None
        current_entity_chunk = []
        for (token, pos) in sentence:
            if pos == previous_pos and pos.startswith('NN'):
                current_entity_chunk.append(token)
            elif pos.startswith('NN'):
                if current_entity_chunk != []:
                    all_entity_chunks.append((' '.join(current_entity_chunk),
                            pos))
                current_entity_chunk = [token]
            previous_pos = pos
        if len(all_entity_chunks) > 1:
            entity_interactions.append(all_entity_chunks)
        else:
            entity_interactions.append([])
    assert len(entity_interactions) == len(sentences)
    return dict(entity_interactions=entity_interactions,
                sentences=sentences)
blog_data = json.loads(open(BLOG_DATA).read())
# Display selected interactions on a per-sentence basis
for post in blog_data:
    post.update(extract_interactions(post['content']))
    print post['title']
    print '-' * len(post['title'])
    for interactions in post['entity_interactions']:
        print '; '.join([i[0] for i in interactions])
    print
```

下面清单的结果强调了非结构化数据分析本质中非常重要的内容：它很混乱！

```
   The Louvre of the Industrial Age
--------------------------------
morning; chance; tour; Henry Ford Museum; Dearborn; MI; Dale Dougherty; creator;
Make; Makerfaire; Marc Greuther; chief curator
tweet; Louvre
"; Marc; artifact; museum; block; contains; Luther Burbank; shovel; Thomas Edison
Luther Burbank; course; inventor; treasures; nectarine; Santa Rosa
Ford; farm boy; industrialist; Thomas Edison; friend
museum; Ford; homage; transformation; world
machines; steam; engines; coal; generators; houses; lathes; precision; lathes;
makerbot; century; ribbon glass machine; incandescent; lightbulbs; world; combine;
harvesters; railroad; locomotives; cars; airplanes; gas; stations; McDonalds;
restaurant; epiphenomena
Marc; eye; transformation; machines; objects; things
advances; engineering; materials; workmanship; design; years
years; visit; Detroit; museum; visit; Paris; Louvre; Rome; Vatican Museum;
Florence; Uffizi Gallery; St. Petersburg; Hermitage; Berlin
world; museums
Museum; Makerfaire Detroit
reach; Detroit; weekend
day; Makerfaire; day
```

结果中一定量的“噪声”几乎是不可避免的，但认识到结果是十分智能并且有用的——即使它们确实包含不少“噪声”——是很有价值的目标。实现几乎无噪音的原始结果需要的工作量是巨大的。事实上，大多数情况下，因为自然语言固有的复杂性和现有的大多数可用工具包（比如NLTK）的局限性，这几乎是不可能实现的。如果你能提出数据领域的某种假设或有含有“噪声”的专业知识，你或许就能设计有效的启发式方法，而不必承担某些信息丢失的风险——但这比较难以实现。

而且，这些交互确实提供了一些有价值的“要点”。例如，你对“morning；chance；tour；Henry Ford Museum；Dearborn；MI；Dale

Dougherty；creator；Make；Makerfaire；Marc Greuther；chief curator”的解释会多接近原句的含义呢？

和我们之前对摘要的研究一样，显示可以直观浏览的标记也是很方便的。对示例5-9输出的简单修改，正如示例5-10中的那样，就是产生图5-6的结果所需要的全部工作了。

示例5-10：采用HTML输出对实体间的交互可视化

```
   import os
import json
import nltk
from IPython.display import IFrame
from IPython.core.display import display
BLOG_DATA = "resources/ch05-webpages/feed.json"
HTML_TEMPLATE = """<html>
    <head>
        <title>%s</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    </head>
    <body>%s</body>
</html>"""
blog_data = json.loads(open(BLOG_DATA).read())
for post in blog_data:
    post.update(extract_interactions(post['content']))
    # Display output as markup with entities presented in bold text
    post['markup'] = []
    for sentence_idx in range(len(post['sentences'])):
        s = post['sentences'][sentence_idx]
        for (term, _) in post['entity_interactions'][sentence_idx]:
            s = s.replace(term, '<strong>%s</strong>' % (term, ))
        post['markup'] += [s]
    filename = post['title'].replace("?", "") + '.entity_interactions.html'
    f = open(os.path.join('resources', 'ch05-webpages', filename), 'w')
    html = HTML_TEMPLATE % (post['title'] + ' Interactions',
                            ' '.join(post['markup']),)
    f.write(html.encode('utf-8'))
    f.close()
    print "Data written to", f.name
    # Display any of these files with an inline frame. This displays the
    # last file processed by using the last value of f.name...
print "Displaying %s:" % f.name
display(IFrame('files/%s' % f.name, '100%', '600px'))
```

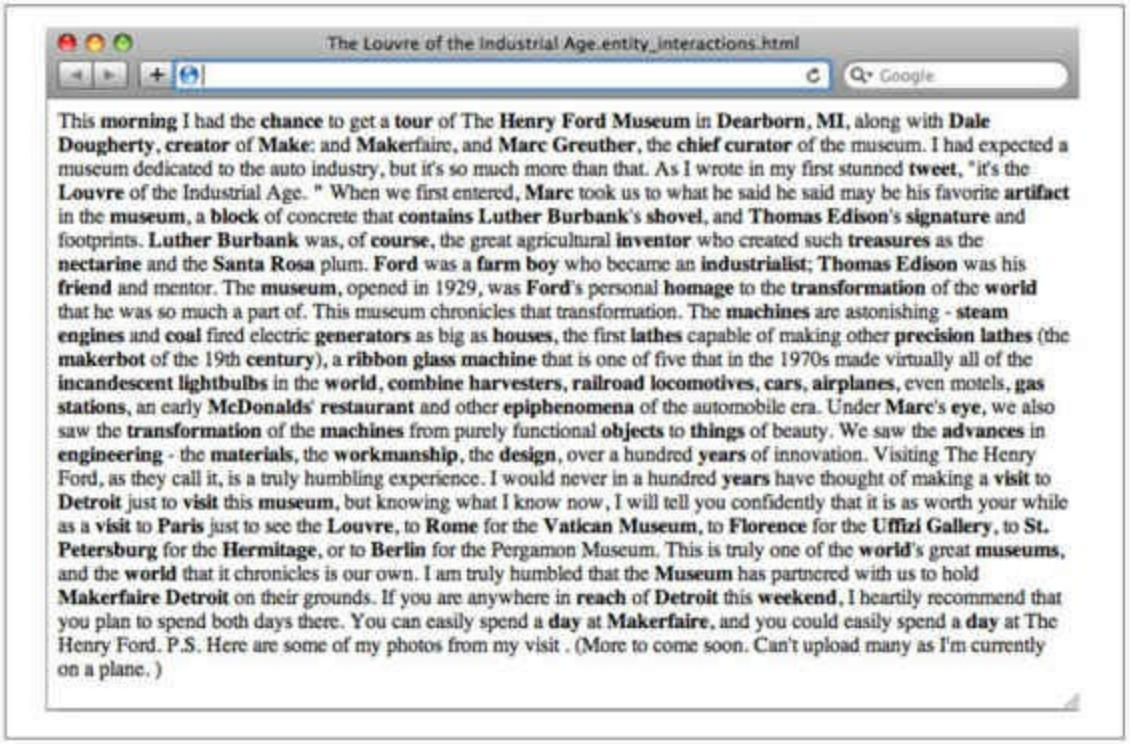


图5-6：示例HTML输出，使用粗体字标识出实体，这样就可以轻易地浏览它的关键概念的内容了

标识大段文本的交互集合并查找交互中的并发事件是很有意义的。参与2.3.2.3节中力导向图的代码是一个很好的可视化的开始模板，就算不了解交互的特定属性，了解主语和宾语也很有价值。如果你觉得不能满足，可以填上缺少的动词来完善元组。

5.5 人类语言数据处理分析的质量

就算只做了很少一部分数据挖掘的工作，你也会很想开始量化你的分析的质量。对句末的检测有多精确？词性标注是否准确？例如，你可能开始定制从非结构化文本中提取实体的基本算法，那你又如何知道你的算法得到的结果的质量的好坏呢？尽管对于小的语料库，你可以人工检查结果并调整算法直到自己满意，但你仍然需要花时间确定你的分析在更大的语料库或不同类的文档中是否运行良好——因此，我们需要更自动化的过程。

明显的起点是：随机抽取一些文档，创建一个实体“黄金集合”，要保证你相信从它们中提取的信息并可以用来评估算法的质量。根据你对自己的要求，你可以计算样本误差，并使用叫做置信区间（http://bit.ly/1a1n8BW）的统计概念来预测真正的误差。然而，为了计算精确度，以你的提取结果和“黄金集合”为基础的计算到底是什么呢？一种计算精确度的很常见的计算叫做F1得分（F1Score），是根据准确率和召回率[1]两个概念所定义的：

$$F = 2 \times \frac{准确率 \times 召回率}{准确率 + 召回率}$$

其中，

$$准确率 = \frac{TP}{TP + FP}$$

还有：

$$召回率 = \frac{TP}{TP + FN}$$

在当前的上下文中，准确率表示误报率（False Positive，FP）正确性的度量，召回率是表示判断为真的正确率（True Positive，TP）完整性的度量。下面的列表说明了与现在的讨论相关的术语的含义，以免你对它们不熟悉而造成混淆：

判断为真的正确率（TP）

正确地标识为术语的实体。

误报率（FP）

标识为实体，却不是实体的术语。

判断为假的正确率（TN）

没有标识为实体，实际上也不是实体的术语。

漏报率（FN）

是实体，却没有被标识的术语。

由于准确率是量化“误报率”正确性的度量，它被定义为TP/（TP+FP）。直观的看，如果“误报率”的值为0，那这个算法就很完美，并且准确率的值是1.0。相反，如果“误报率”很高，达到或超过了“判断为真的正确率”，准确率就很低，这个比例也接近于0。作为完整性的度量，召回率被定义为TP/（TP+FN），如果“漏报率”为0，那么召回率的值就为1.0，表明召回率很完美。当误报率增加时，召回率会接近于0。根据定义，当准确率和召回率都很完美时，F1的值为1.0；当准确率和召回率都很低时，F1的值会接近于0。

当然，当你想提高准确率或者召回率时需要对二者进行权衡，因为通常二者难以兼得。仔细想想，这很容易理解，因为误报率和漏报率就是无法兼得的（见图5-7）。

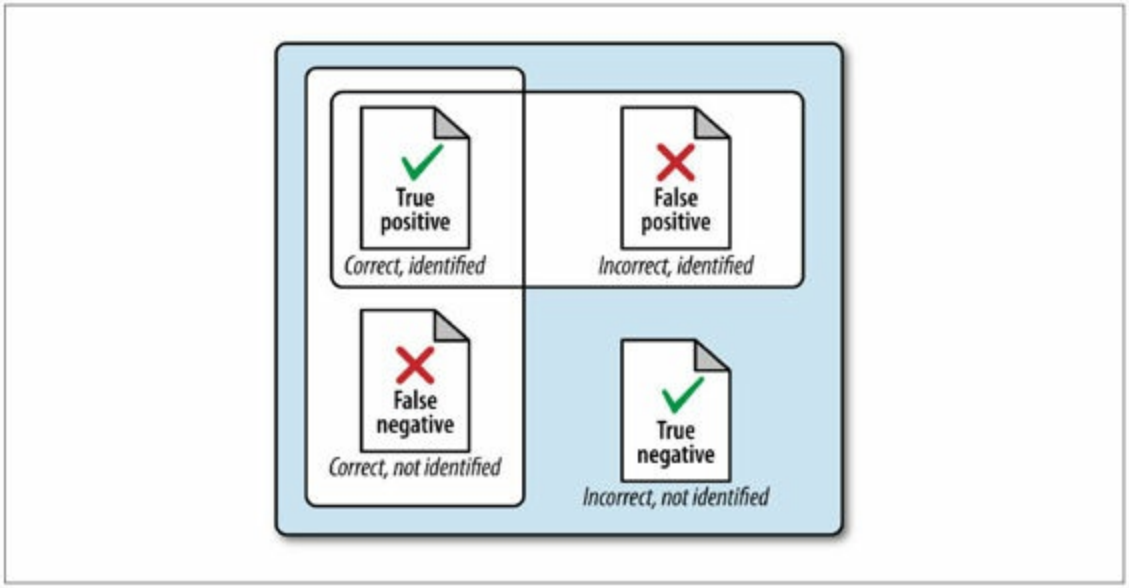


图5-7：站在预测分析的立场上，对判断为真的正确率、误报率，和判

断为假的正确率和漏报率的直观感受

为了深入理解，让我们考虑经典的句子“Mr.Green killed Colonel Mustard in the study with the candlestick”，并假设专家已经确定了这句话的关键实体是“Mr.Green”、“Colonel Mustard”、“study”和“candlestick”。假设你的算法只识别出了这4个术语，那么你的判断为真的正确率为4，误报率为0，判断为假的正确率
（“killed”、“with”、“the”、“in”和“the”）为5，漏报率为0。这样完美的准确率和召回率产生的F1的值为1.0。如果这是你第一次遇到这些术语，把不同的值带入准确率和召回率公式是很有价值的练习。

注意：如果你的算法识别出的
是“Mr.Green”、“Colonel”、“Mustard”和“candlestick”，那么准确率、召回率和F1得分分别是多少呢？

很多最具竞争力的技术栈被NLP领域的商业公司采用，它们使用了先进的统计模型，根据监督式学习算法来处理自然语言。正如我们在本章前面讨论的那样，监督式学习方法是你提供包括输入和预期输出的训练样本的重要方法，这样该模型就能够较为精确的预测三元组了。棘手的部分是确保受过训练的模型在遇到从未见过的输入时，也可以很好地概括。如果该模型在训练数据中运行良好，在从未见过的样本中无法良好运行，通常说它遇到了过度拟合训练数据的问题。一种常见的度量模型效力的方法叫做交叉验证（cross-validation）。在这种方法中，训练

数据的一部分内容（比如三分之一）专门用于测试模型，剩余的数据用于训练模型。

[1] 更确切地说，F1被认为是准确率和召回率的调和平均数，其中任何两个数：x和y的调和平均数定义为：H=2×xy/（x＋y）可以通过阅读调和平均数（http://bit.ly/1a1n6tJ）的定义来了解“调和”是什么意思。

5.6 本章小结

本章介绍了先进的非结构化数据的本质，并说明了如何使用NLTK将剩余的NLP流程和从文本中提取实体结合起来。理解人类语言数据是学科交叉的，虽然我们进行过很多尝试，它仍然处于起步阶段，并且这世界上最常用的语言的NLP问题仍是本世纪争论的问题（或者说至少是21世纪的前50年）。

一旦把NLTK利用到了极致，当你仍想提高性能和质量时，你需要亲自深入研究一些学术著作。这刚开始确实是一项艰巨的任务，但如果你有兴趣解决它，这将非常有价值。本章能教给你的仅限于此，但它开启了巨大的可能性。开源工具是很好的起点，对于那些能掌握自然语言处理的理论和艺术的人来说，前途是非常光明的。

注意：本章和其他章节的源代码都能在GitHub（http://bit.ly/1a1kNqy）中找到，并且是方便的IPythonNotebook的格式，鼓励你从自己的浏览器开始尝试。

5.7 推荐练习

·应用本章的代码，从几百个高质量的文章或博客中收集数据并总结内容。

·使用例如Google App Engine的这类工具建立一个在线总结工具的网页应用。（比如Yahoo！最近收购了为读者总结新闻的Summly公司（http://tcrn.ch/1a1n70L），你会发现这个练习非常有启发意义。）

·考虑使用NLTK的word-stemming工具，参考示例5-9的代码，尝试计算（实体，谓语词干，实体）的三元组。

·学习WordNet（http://bit.ly/1a1n7hj），它是你在NLP过程中，发现谓语短语的额外含义时一定会遇到的工具。

·用标签云（tag cloud，http://bit.ly/1a1n5pO）展示从文本中提取的实体。

·基于可以编码成规则的逻辑，使用if-then语句，尝试自己写一个句末检测器，这就是一个确定的分析器。请比较和NLTK中的方法的区别。使用确定规则对语言建模是合适的吗？

·使用Scrapy爬取少量新闻或博客数据并提取文本以供处理。

·探索NLTK的贝叶斯分类器（http://bit.ly/1a1n9Wt），这是可以用来标记训练样本（比如文档）的监督式学习技术。你可以训练一个分类器来把用Scrapy抓取的文档标记为“sports”、“editorial”和“other”吗？使用F1得分的方法来衡量你的精确度。

·有没有可能准确率和召回率的调和平均数很不令人满意呢？什么时候你愿意牺牲召回率来换取更高的准确率呢？什么时候你又愿意牺牲准确率来换取更高的召回率呢？

·你能把本章学到的技术应用到Twitter的数据上吗？GATE Twitter词性标记器（http://bit.ly/1a1nad2）和Carnegie Mellon的Twitter NLP和Part-of-Speech Tagging（http://bit.ly/1a1n84Y）库是很好的开始。

5.8 在线资源

下面这些本章中的链接的清单很值得回顾：

·文献摘要的自动生成（The Automatic Creation of Literature Abstracts，http://bit.ly/1a1n4Cj）

·“词袋”模型（“Bag of words”model，http://bit.ly/1a1lHDF）

·贝叶斯分类器（Bayesian classifier，http://bit.ly/1a1n9Wt）

·BeautifulSoup（http://bit.ly/1a1mRit）

·Boilerplate Detection Using Shallow Text Features（http://bit.ly/1a1mN21）

·广度优先搜索（Breadth-first search，http://bit.ly/1a1mYdG）

·Carnegie Mellon的Twitter NLP和part-of-speech tagger（http://bit.ly/1a1n84Y）

·常见的爬虫语料库（Common Crawl Corpus，http://amzn.to/1a1mXXb）

·置信区间（Confidence interval，http://bit.ly/1a1n8BW）

·d3-cloud的GitHub库（http://bit.ly/1a1n5pO）

·深度优先搜索（http://bit.ly/1a1mVPd）

·GATE Twitter词性标记器（GATE Twitter part-of-speech tagger，http://bit.ly/1a1nad2）

·boilerpipe的托管版本（Hosted version ofboilerpipe，http://bit.ly/1a1mSTF）

·HTML5（http://bit.ly/1a1mRz5）

·Microdata（http://bit.ly/1a1mRPA）

·NLTK book（http://bit.ly/1a1mtAk）

·宾州树库项目（Penn Treebank Project，http://bit.ly/1a1mXqf）

·python-boilerpipe（http://bit.ly/1a1mSD4）

·Scrapy（http://bit.ly/1a1mG6P）

·监督式机器学习（Supervised machine learning，http://bit.ly/1a1mPHr）

·线程池（Thread pool，http://bit.ly/1a1mW5M）

·图灵测试（Turing Test，http://bit.ly/1a1mZON）

·Unsupervised Multilingual Sentence Boundary Detection（http://bit.ly/1a1n3OI）

·WordNet（http://bit.ly/1a1n7hj）

第6章　挖掘邮箱：分析谁和谁说什么以及说的频率等

邮件数据虽有争议，但也是社交网站数据和早期线上社交网络的基础。邮件数据无处不在，同时每条消息本质上都是来自社会的，涉及对话以及多人之间的交互。并且每条消息都包括明确的人类语言数据，含有结构化的元数据域，可以用特定的时间间隔和明确的身份描述人类语言数据。挖掘邮箱数据显然为整合你在前面章节学到的概念提供了机会，并能增加发现有价值的见解的机会。

不管你是对挖掘邮箱列表很感兴趣并且想分析公司通讯的趋势和模式的公司CIO，还是只是想探寻自己邮箱的规律性模式以达到部分的自我量化（http://bit.ly/1a1niJw）的个人，下面的讨论都是一个好的开始。这一章介绍了一些探索邮箱的基本工具和技术，以回答以下类似的问题：

·谁给谁发邮件（次数/频率）？

·一天中是否存在某个特定的时间（或是一周中的几天）邮件最多？

·哪个人给别人发送的邮件最多？

·讨论最热烈的话题是什么？

虽然社交媒体网站积累了越来越多的接近实时的社交数据，但仍存在重大的缺点。社交网络数据是由服务提供者集中管理的，他们可以创建规则，规定访问它的方式以及你能做什么和不能做什么。而邮件数据并不是集中管理的，它以丰富的邮件列表的方式分散在网络上，包括对一连串话题的讨论，或者自己的账号中成千上万的邮件。当你回过头来想想，有效的挖掘邮件数据似乎是你的数据挖掘工具箱中最重要的能力之一了。

尽管找到能作为例证的真实的社交网络数据集并不简单，然而这一章我们将以相对备受研究的Enron语料库（http://bit.ly/1a1nj01）作为基础，以便有最多的不需考虑法律[1]和隐私问题而进行分析的机会。我们将把数据集标准化为众所周知的Unix邮箱（mbox）格式，这样就能使用很多常见的工具来处理数据了。最后，尽管我们可以选择处理存储在文件中的JSON格式的数据，我们还是利用邮件以文档为中心的本质属性，并学习使用MongoDB（http://bit.ly/1a1nlVK）这种更强大而有趣的方式来存储和分析数据，包括频率分析和关键字查找等多种方式。

作为综合的存储和查询任意JSON数据的数据库，MongoDB很难被打败，因为它功能多样且强大，你在很多情况下都会需要它。（尽管我们在这一章之前都选择避免使用外部依赖，比如数据库，但当它成为必需时，你很快就会意识到，采用MongoDB存储任意我们检索和访问到

的社交网络数据都和使用JSON文件一样简单。）

注意：要经常在线获取本章的（或其他章节的）最新修复了bug的源代码，网址是http://bit.ly/MiningTheSocialWeb2E。并且好好利用本书附录A中描述的虚拟机知识，来最大化你对示例代码的使用。

[1] 如果你想分析邮件列表数据，要知道如果使用API，大多数服务提供商（比如Google、雅虎）会限制你使用这些数据，不过你能通过订阅列表并等待数据累积，轻易的获取和保存邮件列表数据。你也可以让列表所有者提供给你一套存档内容作为另一种选择。

6.1 概述

邮件数据异常丰富，你可以用从本书中学到的知识对邮箱数据进行分析。在本章中你将会学到：

·将邮件数据规范化为方便处理的格式的方法。

·MongoDB，这个强大的文档型数据库是存储邮件数据或其他形式的社交网络数据的理想方法。

·Enron语料库，包含了Enron丑闻时期雇员邮箱内容的公开数据集。

·使用MongoDB以多种方式查询Enron语料库。

·访问和导出你自己的邮箱数据进行分析的工具。

6.2 获取和处理邮件语料库

这一节介绍了如何获取邮件语料库，将其转换为mbox，并将mbox导入MongoDB，它提供了通用的存储和查询数据的API。我们将从分析小型的虚构的邮箱开始，然后再处理Enron语料库。

6.2.1 Unix邮箱指南

mbox是大型的连接了很多邮件消息的文本文件，这些数据能够轻易地由基于文本操作的工具获得。邮件工具和协议的发展很快，在很久以前就超出了mbox的范畴，但很多时候你还是可以使用这种格式来简单的处理数据，也可以方便的共享或分发数据。实际上，大多数邮件客户端都提供“导出”或者“另存为”功能让你将数据导出为这种格式（尽管有时很多余），正如6.5节中的图6-2展现的那样。

下面说明详细的格式要求，每封邮件的开头都是由特殊的From行标记，并格式化为“From user@example.com asctime”模式，其中asctime（http://bit.ly/1a1nmcl）是一种标准的固定长度的时间戳表示，比如Fri Dec 2500：06：422009这样的形式。两封邮件的边界是由两个新行前面的（除了第一次出现）的From行决定的。（直观地，正如下面显示的那样，From行前面有一个空白行。）一个虚构的mbox的一小部

分包括两封邮件，如下面所示：

```
   From santa@northpole.example.org Fri Dec 25 00:06:42 2009
Message-ID: <16159836.1075855377439@mail.northpole.example.org>
References: <88364590.8837464573838@mail.northpole.example.org>
In-Reply-To: <194756537.0293874783209@mail.northpole.example.org>
Date: Fri, 25 Dec 2001 00:06:42 -0000 (GMT)
From: St. Nick <santa@northpole.example.org>
To: rudolph@northpole.example.org
Subject: RE: FWD: Tonight
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
Sounds good. See you at the usual location.
Thanks,
-S
-----Original Message-----
From: Rudolph
Sent: Friday, December 25, 2009 12:04 AM
To: Claus, Santa
Subject: FWD: Tonight
Santa -
Running a bit late. Will come grab you shortly.Standby.
Rudy
Begin forwarded message:
> Last batch of toys was just loaded onto sleigh.
>
> Please proceed per the norm.
>
> Regards,
> Buddy
>
> --> Buddy the Elf
> Chief Elf
> Workshop Operations
> North Pole
> buddy.the.elf@northpole.example.org
From buddy.the.elf@northpole.example.org Fri Dec 25 00:03:34 2009
Message-ID: <88364590.8837464573838@mail.northpole.example.org>
Date: Fri, 25 Dec 2001 00:03:34 -0000 (GMT)
From: Buddy <buddy.the.elf@northpole.example.org>
To: workshop@northpole.example.org
Subject: Tonight
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
Last batch of toys was just loaded onto sleigh.
Please proceed per the norm.
Regards,
Buddy
--
Buddy the Elf
Chief Elf
Workshop Operations
North Pole
buddy.the.elf@northpole.example.org
```

在前面的示例中，我们看到了两封邮件，但能够推测出还有另一封被回复的邮件可能保存在mbox的其他地方。按照时间顺序，第一封邮件是由一个叫做Buddy的人写的，并发送到workshop@northpole.example.org说玩具刚被装载好。另一封mbox中的邮件是Santa对Rudolph的回复。没有在示例中显示的是一封由Rudolph转发给Santa的邮件，是来自Buddy的中间邮件，内容是说他来晚了。尽管我们可以通过人为的阅读方式，根据邮件的上下文的内容推断出这些内容，但Message-ID、References和In-Reply-To邮件头中也提供了可供分析的重要线索。

这些邮件头很直观，并且为显示同主题的邮件的算法提供了基础。稍后我们会研究在电子邮件的主题聚合中使用了这些字段的一个著名算法，重点是每封邮件都有独特的邮件ID，包含对这封正在被回复的邮件的引用而且可以在回复链中引用多封其他邮件，这样就知道这些邮件是正在进行的讨论的一部分了。

注意：因为我们会使用Python模块来做大部分琐碎的工作，我们不需要讨论电子邮件的细节，比如多部分邮件、MIME类型（http://bit.ly/1a1nmsJ）、7位内容传输编码等。

这些邮件头极为重要。即使是这个简单的例子，你也能看到当你解析一封邮件的正文时，事情会变得多么混乱：Rudolph的客户端使用“>”符号引用转发的内容，但Santa的客户端总是什么都不引用就直接

回复，不过却使用了可读的邮件头。

大多数邮件客户端都有允许查看平常无法看到的扩展邮件头的选项，当你想查看这些邮件时，如果你感兴趣的是比研究原始存储更方便的技术的话，图6-1显示了Apple Mail的示例邮件头。

图6-1：大多数邮件客户端都允许通过选项菜单查看扩展邮件头

幸运的是，你有很多可以做的，而不用重新实现一个邮件客户端。另外，如果你想做的只是浏览邮箱，只需简单地将它导入邮件客户端中浏览，对吧？

注意：你值得花些时间探索邮件客户端是否提供以mbox的格式导入/导出数据的功能，这样你才能够使用本章的工具来操作它。

为了了解数据处理的过程，示例6-1给出一个程序，它对mbox提出了多个简化假设以引入mailbox包和email包。这两个包都是Python标准库中的一部分。

示例6-1：将mailbox转换为JSON格式

```
   import mailbox
import email
import json
MBOX = 'resources/ch06-mailboxes/data/northpole.mbox'
# A routine that makes a ton of simplifying assumptions
# about converting an mbox message into a Python object
# given the nature of the northpole.mbox file in order
# to demonstrate the basic parsing of an mbox with mail
# utilities
def objectify_message(msg):
    # Map in fields from the message
    o_msg = dict([ (k, v) for (k,v) in msg.items() ])
    # Assume one part to the message and get its content
    # and its content type
    part = [p for p in msg.walk()][0]
    o_msg['contentType'] = part.get_content_type()
    o_msg['content'] = part.get_payload()
    return o_msg
# Create an mbox that can be iterated over and transform each of its
# messages to a convenient JSON representation
mbox = mailbox.UnixMailbox(open(MBOX, 'rb'), email.message_from_file)
messages = []
while 1:
    msg = mbox.next()
    if msg is None: break
    messages.append(objectify_message(msg))
print json.dumps(messages, indent=1)
```

尽管这个处理mbox文件的小脚本很简洁，并能够获得合理的结果，但解析任意邮件数据或者确定任何邮箱中对话的确切流程都很棘手。很多因素都会有影响，比如二义性、人们在回复链中回复或评论的差异、邮件客户端处理邮件和回复的不同等等。

为了强调这一点，表6-1显示了邮件流，并且明确地包含了

northpole.mbox中引用过却没有出现的第三封邮件。从脚本中截取的示例输出如下：

```
  [
 {
  "From": "St. Nick <santa@northpole.example.org>",
  "Content-Transfer-Encoding": "7bit",
  "content": "Sounds good. See you at the usual location.\n\nThanks,...",
  "To": "rudolph@northpole.example.org",
  "References": "<88364590.8837464573838@mail.northpole.example.org>",
  "Mime-Version": "1.0",
  "In-Reply-To": "<194756537.0293874783209@mail.northpole.example.org>",
  "Date": "Fri, 25 Dec 2001 00:06:42 -0000 (GMT)",
  "contentType": "text/plain",
  "Message-ID": "<16159836.1075855377439@mail.northpole.example.org>",
  "Content-Type": "text/plain; charset=us-ascii",
  "Subject": "RE: FWD: Tonight"
 },
 {
  "From": "Buddy <buddy.the.elf@northpole.example.org>",
  "Subject": "Tonight",
  "Content-Transfer-Encoding": "7bit",
  "content": "Last batch of toys was just loaded onto sleigh. \n\nPlease...",
  "To": "workshop@northpole.example.org",
  "Date": "Fri, 25 Dec 2001 00:03:34 -0000 (GMT)",
  "contentType": "text/plain",
  "Message-ID": "<88364590.8837464573838@mail.northpole.example.org>",
  "Content-Type": "text/plain; charset=us-ascii",
  "Mime-Version": "1.0"
 }
]
```

表6-1：northpole.mbox的邮件流

时间	邮件活动
Fri, 25 Dec 2001 00:03:34 -0000 (GMT)	Buddy向workshop发送了邮件
Friday, December 25, 2009 12:04 AM	Rudolph向Santa转发了Buddy的邮件，并添加了内容
Fri, 25 Dec 2001 00:06:42 -0000 (GMT)	Santa回复Rudolph

有了对邮箱的基本的理解，让我们将注意力转移到将Enron语料库（http://bit.ly/1a1nmsU）转换成mbox，这样就能最大化的利用Python标准库。

6.2.2 获得Enron数据

完整Enron数据集（http://bit.ly/1a1nmsU）的原始形式能够以多种格式获取，不过需要大量的预处理过程。我们选择从最原始的未处理数据集开始，它实际上是根据人或文件夹对一系列邮箱进行组织的文件夹集合。数据的规范化和清洗是一个常规步骤，本章将提供一些思路和做法。

如果你好好的利用本书中的虚拟机知识，本章的IPython Notebook提供了将数据下载到合适位置的脚本，来让你顺顺利利地学习这些示例。完整的Enron语料的压缩版本大约有450MB，你可以下载它来跟随这些练习学习。如果你的网速正常而且电脑较新，那么可能最多只需要10分钟来下载和解压这些数据。

遗憾的是，如果你使用的是虚拟机，Vagrant将上千未归档文件同步到主机的时间可能会达到两小时。如果时间对你来说是很重要的因素并且不能找到合适的时机运行这个脚本，你可以跳过下载和前期的处理步骤，因为处理过的数据（就像从示例6-3中得到的一样）和源代码可以在ipynb/resources/ch06-mailboxes/data/enron.mbox.json.bz2找到。欲查看更详细的信息，参见随本章IPython Notebook的注意事项。

警告：下载和解压文件的时间比Vagrant同步大量在主机上解压的

文件要快，并且写作本书时还没有一个变通的方法可以在所有的平台上对此进行加速。Vagrant可能要花超过一小时的时间来同步上千个解压的文件。

下面终端输出显示了你下载并解压后的语料库的基本结构。这些数据值得你花些时间在终端中对它们进行研究从而熟悉它们以及浏览它们。

注意：如果你使用Windows系统而不熟悉终端操作，你可以浏览一下theipynb/resources/ch06-mailboxes/data文件夹，如果你能够利用本书中的虚拟机知识，你就能够将它同步到你的主机上。

```
   $ cd enron_mail_20110402/maildir # Go into the mail directory
maildir $ ls # Show folders/files in the current directory
allen-p       crandell-s     gay-r          horton-s
lokey-t       nemec-g        rogers-b       slinger-r
tycholiz-b    arnold-j       cuilla-m       geaccone-t
hyatt-k       love-p         panus-s        ruscitti-k
smith-m       ward-k         arora-h        dasovich-j
germany-c     hyvl-d         lucci-p        parks-j
sager-e       solberg-g      watson-k       badeer-r
corman-s      gang-l         holst-k        lokay-m
            ...directory listing truncated...
neal-s        rodrique-r     skilling-j     townsend-j
$ cd allen-p/ # Go into the allen-p folder
allen-p $ ls # Show files in the current directory
_sent_mail       contacts            discussion_threads notes_inbox
sent_items       all_documents       deleted_items      inbox
sent             straw
allen-p $ cd inbox/ # Go into the inbox for allen-p
inbox $ ls # Show the files in the inbox for allen-p
1.  11. 13. 15. 17. 19. 20. 22. 24. 26. 28. 3.  31. 33. 35. 37. 39. 40.
42. 44. 5.  62. 64. 66. 68. 7.  71. 73. 75. 79. 83. 85. 87. 10. 12. 14.
16. 18. 2.  21. 23. 25. 27. 29. 30. 32. 34. 36. 38. 4.  41. 43. 45. 6.
63. 65. 67. 69. 70. 72. 74. 78. 8.  84. 86. 9.
inbox $ head -20 1. # Show the first 20 lines of the file named "1."
Message-ID: <16159836.1075855377439.JavaMail.evans@thyme>
Date: Fri, 7 Dec 2001 10:06:42 -0800 (PST)
From: heather.dunton@enron.com
To: k..allen@enron.com
Subject: RE: West Position
Mime-Version: 1.0
```

```
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
X-From: Dunton, Heather </O=ENRON/OU=NA/CN=RECIPIENTS/CN=HDUNTON>
X-To: Allen, Phillip K. </O=ENRON/OU=NA/CN=RECIPIENTS/CN=Pallen>
X-cc:
X-bcc:
X-Folder:\Phillip_Allen_Jan2002_1\Allen, Phillip K.\Inbox
X-Origin: Allen-P
X-FileName: pallen (Non-Privileged).pst
Please let me know if you still need Curve Shift.
Thanks,
```

终端中最后的命令说明了邮件消息被组织成了文件，包含能够和内容一起被处理的邮件头格式的元数据。这些数据的格式统一，但并不一定就是能被强大的工具处理的常见格式。所以，我们需要对数据进行预处理，把一部分转换为众所周知的Unix的mbox格式，以便更好地说明将邮件语料规范化为常见并易于处理的格式的一般过程。

6.2.3 将邮件语料转换为Unix邮箱

示例6-2演示了查找Enron语料库中“收件箱”（inbox）文件夹的目录结构和在其中添加消息生成enron.mbox文件作为输出的方法。为了运行这个脚本，你需要下载Enron语料库并且解压到脚本中MAILDIR指定的目录下。

这个脚本使用了dateutil包来将数据转换为标准格式。我们之前并没有转换，考虑到变化消耗的空间，这比想象的要难一些。你可以使用pip install python_dateutil命令来安装这个包。（在这种特殊情况下，pip命令安装时的包名和你在代码里导入的名字可能会略有不同。）否则，

这个脚本只是使用Python标准库中的一些工具将数据重组到一个mbox。虽然从解析上来说并不是很有趣，但这个脚本提示了如何使用正则表达式以及后面将会用到的email包，并阐述了其他对数据处理有用的概念。确保你理解了脚本的工作原理从而扩展了你的综合应用知识和数据挖掘的工具链。

警告：这个脚本应该会在完整的Enron语料库上运行10到15分钟，这取决于你电脑的硬件。处理数据时，IPython Notebook会在界面的右上角显示“Kernel Busy”（内核繁忙）的信息。

示例6-2：将Enron语料转换为标准的mbox格式

```
   import re
import email
from time import asctime
import os
import sys
from dateutil.parser import parse # pip install python_dateutil
# XXX: Download the Enron corpus to resources/ch06-mailboxes/data
# and unarchive it there.
MAILDIR = 'resources/ch06-mailboxes/data/enron_mail_20110402/' +\
          'enron_data/maildir'
# Where to write the converted mbox
MBOX = 'resources/ch06-mailboxes/data/enron.mbox'
# Create a file handle that we'll be writing into...
mbox = open(MBOX, 'w')
# Walk the directories and process any folder named 'inbox'
for (root, dirs, file_names) in os.walk(MAILDIR):
    if root.split(os.sep)[-1].lower() != 'inbox':
        continue
    # Process each message in 'inbox'
    for file_name in file_names:
        file_path = os.path.join(root, file_name)
        message_text = open(file_path).read()
        # Compute fields for the From_ line in a traditional mbox message
        _from = re.search(r"From: ([^\r]+)", message_text).groups()[0]
        _date = re.search(r"Date: ([^\r]+)", message_text).groups()[0]
        # Convert _date to the asctime representation for the From_ line
        _date = asctime(parse(_date).timetuple())
        msg = email.message_from_string(message_text)
        msg.set_unixfrom('From %s %s' % (_from, _date))
        mbox.write(msg.as_string(unixfrom=True) + "\n\n")
```

```
mbox.close()
```

如果你查看刚创建的mbox文件，你会发现它除了遵循众所周知的规范外，还是一个单独的文件，它和我们之前看到的邮件格式非常相似。

注意：如果你倾向于分析更集中的Enron语料库的子集，你只需简单地给每个人或是特定的一组人创建单独的mbox文件。

6.2.4 将Unix邮箱转换为JSON

mbox文件非常方便，因为有很多现成的跨计算平台和编程语言的工具可以处理它。在这一小节中，我们会抛开示例6-1中的许多简单假设，这样就能使程序更加健壮的处理Enron邮箱，并考虑许多处理邮箱数据时会碰到的常见问题。Python标准库中含有mbox工具，示例6-3中的脚本介绍了将mbox数据转换为行分隔的JSON格式的方法，并且能够导入文档型数据库中，比如MongoDB（http://bit.ly/1a1nlVK）。我们稍后将更多的讨论MongoDB以及为什么它很适合存储像邮件数据这样的内容。然而现在，只需了解它存储的数据是类JSON格式的并提供强大的功能来索引和操作数据就足够了。

MongoDB还有一个便利之处，就是我们将每封邮件的日期标准化为epoch格式（从1970年1月1日开始的毫秒数），并和特殊的标记一同

传递到MongoDB中，这样MongoDB就能采用标准的方法来解释日期字段。尽管我们可以在将数据加载进MongoDB中之后再做这件事，然而这项繁琐的工作被归于“数据整合”类别下，允许我们在数据加载之后可以立刻使用每封邮件的日期字段进行统一查询。

最终，为了获得能够导入MongoDB的数据，我们需要给每个MongoDB的文档写一个每行包含一个JSON对象的文件。重申一下，就算你对分析不感兴趣，这个脚本还是演示了一些数据清洗和处理过程中的若干实际问题——比如，邮件数据可能不是像UTF-8那样的特定编码，并且可能包含需要进行分离的HTML格式。

警告：示例6-3中的很多地方都有decode（'utf-8'，'ignore'）函数。当你处理以文本为基础的数据（比如邮件数据或者网页数据）时，由于意外的字符编码，很可能遇到令人讨厌的UnicodeDecodeError，但错误并不明显，找到修复方法也不是很容易。你可以在任意的字符串上使用decode函数，该函数的第二个参数指明遇到UnicodeDecodeError时该如何处理。默认值是'strict'，会抛出异常，不过你可以根据自己的需要使用'ignore'或者'replace'来代替默认值。

示例6-3：将mbox转换为适合导入MongoDB的JSON结构

```
   import sys
import mailbox
import email
import quopri
import json
import time
```

```
from BeautifulSoup import BeautifulSoup
from dateutil.parser import parse
MBOX = 'resources/ch06-mailboxes/data/enron.mbox'
OUT_FILE = 'resources/ch06-mailboxes/data/enron.mbox.json'
def cleanContent(msg):
    # Decode message from "quoted printable" format
    msg = quopri.decodestring(msg)
    # Strip out HTML tags, if any are present.
    # Bail on unknown encodings if errors happen in BeautifulSoup.
    try:
        soup = BeautifulSoup(msg)
    except:
        return ''
    return ''.join(soup.findAll(text=True))
# There's a lot of data to process, and the Pythonic way to do it is with a
# generator. See http://wiki.python.org/moin/Generators.
# Using a generator requires a trivial encoder to be passed to json for object
# serialization.
class Encoder(json.JSONEncoder):
    def default(self, o): return  list(o)
# The generator itself...
def gen_json_msgs(mb):
    while 1:
        msg = mb.next()
        if msg is None:
            break
        yield jsonifyMessage(msg)
def jsonifyMessage(msg):
    json_msg = {'parts': []}
    for (k, v) in msg.items():
        json_msg[k] = v.decode('utf-8', 'ignore')
    # The To, Cc, and Bcc fields, if present, could have multiple items.
    # Note that not all of these fields are necessarily defined.
    for k in ['To', 'Cc', 'Bcc']:
        if not json_msg.get(k):
            continue
        json_msg[k] = json_msg[k].replace('\n', '').replace('\t', '').replace('\r',
''')\
                                  .replace(' ', '').decode('utf-8',
'ignore').split(',')
    for part in msg.walk():
        json_part = {}
        if part.get_content_maintype() == 'multipart':
            continue
        json_part['contentType'] = part.get_content_type()
        content = part.get_payload(decode=False).decode('utf-8', 'ignore')
        json_part['content'] = cleanContent(content)
        json_msg['parts'].append(json_part)
    # Finally, convert date from asctime to milliseconds since epoch using the
    # $date descriptor so it imports "natively" as an ISODate object in MongoDB
    then = parse(json_msg['Date'])
    millis = int(time.mktime(then.timetuple())*1000 + then.microsecond/1000)
    json_msg['Date'] = {'$date' : millis}
    return json_msg
mbox = mailbox.UnixMailbox(open(MBOX, 'rb'), email.message_from_file)
# Write each message out as a JSON object on a separate line
# for easy import into MongoDB via mongoimport
f = open(OUT_FILE, 'w')
for msg in gen_json_msgs(mbox):
    if msg != None:
        f.write(json.dumps(msg, cls=Encoder) + '\n')
```

```
f.close()
```

总会有更多的数据整合可以进行，但是我们已经处理了一些最常见的问题，比如解码可打印字符引用编码的文本或分离HTML标签的基本机制。（quopri包就是用来处理可打印字符引用编码格式的，这是一种能在7位数据通路[1]上传输8位数据的编码。）下面是在Enron的mbox文件上运行示例6-3所得到输出中的一行，用它来说明输出的基本格式：

```
   {
    "Content-Transfer-Encoding": "7bit",
    "Content-Type": "text/plain; charset=us-ascii",
    "Date": {
        "$date": 988145040000
    },
    "From": "craig_estes@enron.com",
    "Message-ID": "<24537021.1075840152262.JavaMail.evans@thyme>",
    "Mime-Version": "1.0",
    "Subject": "Parent Child Mountain Adventure, July 21-25, 2001",
    "X-FileName": "jskillin.pst",
    "X-Folder": "\\jskillin\\Inbox",
    "X-From": "Craig_Estes",
    "X-Origin": "SKILLING-J",
    "X-To": "",
    "X-bcc": "",
    "X-cc": "",
    "parts": [
        {
            "content": "Please respond to Keith_Williams...",
            "contentType": "text/plain"
        }
    ]
}
```

这个简单的脚本对于去除噪声数据、解析邮件中有用的信息、创建能够导入MongoDB的文件都很有效。这才是真正乐趣开始的地方。有了刚刚获得的将邮件数据整合和处理成可访问格式的能力，你很自然地会迫不及待地想分析这些数据。在下一节，我们会将数据导入MongoDB并开始数据分析。

注意：如果你选择不下载原始的Enron数据并且遵循预处理的步骤，你仍然可以根据本章IPython Notebook中的笔记，从示例6-3获得输出，并且从这里开始继续每个标准的探讨。

6.2.5 将JSON化的邮件语料导入MongoDB

使用合适的工具能够使数据分析的工作更加轻松。尽管Python已经是相对简单的处理JSON数据的语言了，但将JSON数据导入文档型数据库（如MongoDB）中似乎更加容易。

如果只是以实际应用为目的，只需把MongoDB想象为存储和操作JSON数据的很简单的数据库。你可以将数据组织到集合中，使用有效的方式遍历或查询数据，比如全文索引等。在当前分析Enron语料的上下文中，MongoDB提供了相应的API来操作数据，允许我们创建索引、对JSON文档的任意字段进行查询、甚至进行全文检索。

在我们的练习中，你可以在本地机器上运行MongoDB的实例，当数据增加时，也可以扩展MongoDB到集群上运行。它拥有出色的管理工具，并且如果需要技术支持，也有专业的服务公司。关于MongoDB的更为详尽的讨论超出了本书的范围，不过即使在你阅读本章之前从未听说过MongoDB也无妨，跟随本节内容直接了解它就够了。MongoDB的在线文档和教程（http://bit.ly/1a1nlVK）非常丰富，既然都给了链

接，你应该花些时间来查看。

不管你使用什么操作系统，你都应该直接安装MongoDB而不是使用虚拟机，你可以轻松地找到在线教程（http://bit.ly/1a1nqc9）；所有主流平台的包都可以找到。你只需确认你使用的是2.4或者更高的版本，因为本章的部分练习依赖全文索引，它是2.4版本中引入的新特性。和虚拟机一起预安装的MongoDB是作为一个服务安装和管理的（http://bit.ly/1a1nokx），只能通过在配置文件（位置是/etc/mongodb.conf）中设置参数来支持全文检索的索引。

通过运行示例6-4、示例6-5和示例6-6可以确保Enron数据已经加载、建立了全文索引，并且可以分析了。这些示例利用了对subprocess（http://bit.ly/1a1nrgb）包的轻量封装，叫做Envoy（http://bit.ly/1a1nrwL），它能让你轻易地在Python程序中执行终端命令并获得标准输出和标准错误。依据标准协议，你可以在终端中使用pip install envoy命令来安装envoy。

示例6-4：从IPython Notebook中获得mongoimport命令的选项

```
   import envoy # pip install envoy
r = envoy.run('mongoimport')
print r.std_out
print r.std_err
```

示例6-5：使用mongoimport从IPython Notebook向MongoDB中加载数据

```
   import os
import sys
import envoy
data_file = os.path.join(os.getcwd(), 'resources/ch06-mailboxes/data/enron.mbox.json')
# Run a command just as you would in a terminal on the virtual machine to
# import the data file into MongoDB.
r = envoy.run('mongoimport --db enron --collection mbox ' +\
              '--file %s' % data_file)
# Print its standard output
print r.std_out
print sys.stderr.write(r.std_err)
```

示例6-6：在IPython Notebook中模拟运行MongoDB shell

```
   # We can even simulate a MongoDB shell using envoy to execute commands.
# For example, let's get some stats out of MongoDB just as though we were working
# in a shell by passing it the command and wrapping it in a printjson function to
# display it for us.
def mongo(db, cmd):
    r = envoy.run("mongo %s--eval 'printjson(%s)'" % (db, cmd,))
    print r.std_out
    if r.std_err: print r.std_err
mongo('enron', 'db.mbox.stats()')
```

示例6-6的样例输出如下，它和你在MongoDB shell中使用命令看到的一模一样。很简洁！

```
   MongoDB shell version: 2.4.3
connecting to: enron
{
  "ns" : "enron.mbox",
  "count" : 41299,
  "size" : 157744000,
  "avgObjSize" : 3819.5597956366983,
  "storageSize" : 185896960,
  "numExtents" : 10,
  "nindexes" : 1,
  "lastExtentSize" : 56438784,
  "paddingFactor" : 1,
  "systemFlags" : 1,
  "userFlags" : 0,
  "totalIndexSize" : 1349040,
  "indexSizes" : {
  "_id_" : 1349040
  },
  "ok" : 1
}
```

注意：通过虚拟机终端来加载JSON数据可以像示例6-5中演示的那样，使用下面的mongoimport命令来实现：

```
    mongoimport --db enron --collection mbox --file
/home/vagrant/share/ipynb/resources/ch06-mailboxes
/data/enron.mbox.json
```

MongoDB安装后，最后一步是使用命令pip install pymongo来安装Python客户端pymongo（http://bit.ly/1a1nqZE）包，因为我们马上要使用Python客户端来连接MongoDB并访问Enron数据。

警告：要知道MongoDB在32位系统上最多只支持2GB（http://bit.ly/1a1ns3L）的数据库。虽然这对本章要使用的Enron数据集来说并不是什么问题，但为了防止将来在32位系统上出现问题，你还是应该记住这一点。

6.2.5.1 MongoDB shell

尽管在本章的练习中我们使用Python编写程序，但如果你喜欢使用终端，MongoDB还有一个很方便的shell。这一小节就简单地介绍一下。如果要好好利用本书的虚拟机知识，你需要在secure shell登录虚拟机。在包含Vagrantfile的首层检查点文件夹输入vagrant ssh，能够让你自动登录到虚拟机中。

如果你使用Mac OS X或者Linux系统，SSH客户端在系统中本来就

存在，你直接就可以使用vagrant ssh命令。如果你是Windows用户并且遵循附录A中推荐的Git for Windows（http://bit.ly/1a1nubC）安装方法，只要在安装过程中选择了安装SSH客户端，vagrant ssh命令也将可以使用。如果你是Windows用户并且倾向于使用PuTTY（http://bit.ly/1a1nsAG），输入vagrant ssh能看到如何配置它的信息：

```
   $ vagrant ssh
Last login: Sat Jun 1 04:18:57 2013 from 10.0.2.2
vagrant@precise64:~$ mongo
MongoDB shell version: 2.4.3
connecting to: test
> show dbs
enron 0.953125GB
local 0.078125GB
> use enron
switched to db enron
> db.mbox.stats()
{
  "ns" : "enron.mbox",
  "count" : 41300,
  "size" : 157756112,
  "avgObjSize" : 3819.7605811138014,
  "storageSize" : 174727168,
  "numExtents" : 11,
  "nindexes" : 2,
  "lastExtentSize" : 50798592,
  "paddingFactor" : 1,
  "systemFlags" : 0,
  "userFlags" : 1,
  "totalIndexSize" : 221471488,
  "indexSizes" : {
  "_id_" : 1349040,
  "TextIndex" : 220122448
  },
  "ok" : 1
}
> db.mbox.findOne()
{
  "_id" : ObjectId("51968affaada66efc5694cb7"),
  "X-cc" : "",
  "From" : "heather.dunton@enron.com",
  "X-Folder" : "\\Phillip_Allen_Jan2002_1\\Allen, Phillip K.\\Inbox",
  "Content-Transfer-Encoding" : "7bit",
  "X-bcc" : "",
  "X-Origin" : "Allen-P",
  "To" : [
    "k..allen@enron.com"
  ],
```

```
  "parts" : [
    {
      "content" : " \nPlease let me know if you still need...",
          "contentType" : "text/plain"
    }
  ],
  "X-FileName" : "pallen (Non-Privileged).pst",
  "Mime-Version" : "1.0",
  "X-From" : "Dunton, Heather </O=ENRON/OU=NA/CN=RECIPIENTS/CN=HDUNTON>",
  "Date" : ISODate("2001-12-07T16:06:42Z"),
  "X-To" : "Allen, Phillip K. </O=ENRON/OU=NA/CN=RECIPIENTS/CN=Pallen>",
  "Message-ID" : "<16159836.1075855377439.JavaMail.evans@thyme>",
  "Content-Type" : "text/plain; charset=us-ascii",
  "Subject" : "RE: West Position"
}
```

shell中的命令显示了可用的数据库、将当前使用的数据库设为enron、显示了enron数据库中的数据并任意获得了一个文档的数据。我们在本章并不会花更多时间在MongoDB上，不过你操作数据时会发现它很有用，所以对它的简单介绍是很有必要的。可以在MongoDB的在线文档“The Mongo Shell”（http://bit.ly/1a1nus8）中查看更多信息。

6.2.6 在程序中使用Python访问MongoDB

成功将Enron语料（或其他数据）加载到MongoDB中之后，你就会想要使用一种编程语言编程访问和操作这些数据。MongoDB无疑针对很多编程语言都提供了丰富的库可以选择，包括PyMongo（http://bit.ly/1a1nqZE），它是使用Python操作MongoDB的推荐方法。使用pip install pymongo命令就可以安装PyMongo了；示例6-7包含一个简单的脚本来演示它是如何工作的。MongoDB多种多样的find（http://bit.ly/1a1nt7Q）函数提供了不同的查询的方法，这是大多数

MongoDB查询的基础，是非常有用的。

示例6-7：使用Python中的PyMongo访问MongoDB

```
   import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo
# Connects to the MongoDB server running on
# localhost:27017 by default
client = pymongo.MongoClient()
# Get a reference to the enron database
db = client.enron
# Reference the mbox collection in the Enron database
mbox = db.mbox
# The number of messages in the collection
print "Number of messages in mbox:"
print mbox.count()
print
# Pick a message to look at...
msg = mbox.find_one()
# Display the message as pretty-printed JSON. The use of
# the custom serializer supplied by PyMongo is necessary in order
# to handle the date field that is provided as a datetime.datetime
# tuple.
print "A message:"
print json.dumps(msg, indent=1, default=json_util.default)
```

下面是简短的示例输出，这说明了使用pymongo和使用Mongo shell是差不多的，只不过需要特别的考虑对象序列化的关系。

```
   Number of messages in mbox:
41299
A message:
{
  "X-cc": "",
  "From": "craig_estes@enron.com",
  "Content-Transfer-Encoding": "7bit",
  "X-bcc": "",
  "parts": [
   {
    "content": "Please respond to Keith_Williams\"A YPO International...",
    "contentType": "text/plain"
   }
  ],
  "X-Folder": "\\jskillin\\Inbox",
  "X-Origin": "SKILLING-J",
  "X-FileName": "jskillin.pst",
  "Mime-Version": "1.0",
  "Message-ID": "<24537021.1075840152262.JavaMail.evans@thyme>",
```

```
  "X-From": "Craig_Estes",
  "Date": {
  "$date": 988145040000
 },
 "X-To": "",
 "_id": {
  "$oid": "51a983dae391e8ff964bc4c4"
 },
 "Content-Type": "text/plain; charset=us-ascii",
 "Subject": "Parent Child Mountain Adventure, July 21-25, 2001"
}
```

通过上面的学习，现在你应该对以下内容有所了解：如何获取邮件数据、将它处理为标准的Unix邮箱格式、将标准化数据加载到MongoDB中以及查询数据。分析真实数据和这些步骤是很相似的（除了它们各有自己的曲折），所以如果你能够跟上之前的学习，你的数据科学工具包中将会增加一些强有力的工具。

30秒学习Map-Reduce

Map-Reduce是一种计算模式，包含两个主要的方法：map和reduce。map（映射）函数用来收集文档并对每个文档的新的键值对建立映射，而reduce（化简）函数则是收集文档并将它们使用某些方法进行化简。例如，计算算术平方和的函数，f（x）=$x_1^2+x_2^2+...+x_n^2$，它可以表示为对每个值进行平方的映射函数，对每个输入值的操作都是一样的，而化简函数是对映射函数的输出进行简单的求和，并化简为一个值。这种编程模式对繁琐的并行问题非常有效，但并不是对每个问题都很适用（高性能）。

MongoDB和其他文档型数据库（比如CouchDB和Riak）在单机或者

分布式计算环境下都支持map-reduce。如果你对“大数据”或者在MongoDB上使用高度自定义的查询感兴趣，你可以对Map-Reduce进行更深的学习。当今，这是一项大数据计算模式的重要技术。

[1] 可以查看维基百科（http://bit.ly/1a1nngg）获得概览，而如果你对它的工作原理感兴趣的话，可以查看RFC 2045（http://bit.ly/1a1nnNp）。

6.3 分析Enron语料库

我们在将Enron数据转换为可以方便查询的格式的问题上投入了大量的精力，下面就开始试着去理解这些数据。正如前面的章节那样，当面临任何类型的新数据集时，计算问题是你初步探索时想要考虑的任务，因为它会让你事半功倍。这一节介绍了许多MongoDB的find操作来查询邮箱数据，包括对不同数据字段的组合查询等。这是对应用讨论的扩展，并不会花费太多力气。

Enron丑闻概述

虽然这不是很有必要的，不过如果你知道Enron丑闻，你或许想了解更多，因为这是我们分析的邮件数据的主题。下面这些与Enron有关的事实对我们在本章分析数据时理解上下文有很大帮助：

·Enron是一家位于美国德克萨斯州的能源公司，它从1985年建立以来，成长为一个数百亿美元的公司。直到2001年10月，它的丑闻被曝光了。

·Kenneth Lay是Enron公司的CEO，也是许多有关Enron公司的讨论中的热门话题。

·Enron丑闻的实质是采用财政手段（被称为raptors）来有效的隐藏

资产负债。

·Arthur Andersen曾经是著名的会计师事务所，它在参政审计上也负有责任。在Enron丑闻之后很快就关闭了。

·在丑闻被披露后很短的时间内，Enron申请破产超过60亿美元；这成为当时美国历史上最大的破产企业。

维基百科中关于Enron scandal（http://bit.ly/1a1nuZo）的词条对背景和关键事件做了简单易读的介绍，你只需花几分钟就能了解整个事件。如果你还想深入了解，可以观看纪录片《Enron：The Smartest Guys in the Room》（http://imdb.to/1a1nvwd）。

注意：这个网址http://www.enron-mail.com是Enron邮件数据的一个版本，随着你开始熟悉Enron语料库你会发现这是很有用的。

6.3.1 根据日期/时间范围查询

我们已经把经过处理的数据导入了MongoDB，所以每个对象的Date字段会被数据库正确的解析出来，但对日期/时间范围查询会有些繁琐。事实上，示例6-8是对前面示例的扩展，代码中建立了和数据库的连接并发起了如下的用一些参数进行限制的find请求：

```
mbox.find({"Date" :
```

```
            {
             "$lt" : end_date,
             "$gt" : start_date
            }
          }).sort("date")
```

请求中的表达为：“找到所有有Date域的信息，并且保证Date在start_date之后，在end_date之前。获得结果后，将结果排序并返回”。以美元符号为开头（例如$lt和$gt）的字段是MongoDB中特殊的操作符。在上面的示例中，它们分别表示“小于”和“大于”。你可以在完善的在线文档中阅读到其他的MongoDB操作符（http://bit.ly/1a1nvN1）。

另一个需要记住的问题是：除非数据已经在你要求的特定排序规则下建立了索引，否则排序数据经常会花费额外的时间。在示例6-8中，我们使用指定的日期范围来查询，基于前面findOne返回的结果，随机选取时间范围内的数据，会发现邮箱中的数据大约有2001条。

示例6-8：通过日期/时间范围查询MongoDB

```
   import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo
from datetime import datetime as dt
client = pymongo.MongoClient()
db = client.enron
mbox = db.mbox
# Create a small date range here of one day
start_date = dt(2001, 4, 1) # Year, Month, Day
end_date = dt(2001, 4, 2) # Year, Month, Day
# Query the database with the highly versatile "find" command,
# just like in the MongoDB shell.
msgs = [ msg
        for msg in mbox.find({"Date" :
                                {
                                 "$lt" : end_date,
                                 "$gt" : start_date
                                }
                             }).sort("date")]
# Create a convenience function to make pretty-printing JSON a little
```

```
# less cumbersome
def pp(o, indent=1):
    print json.dumps(msgs, indent=indent, default=json_util.default)
print "Messages from a query by date range:"
pp(msgs)
```

下面的示例输出显示了这些数据集中，只有一条数据满足指定的时间范围：

```
   Messages from a query by date range:
[
 {
  "X-cc": "",
  "From": "spisano@sprintmail.com",
  "Subject": "House repair bid",
  "To": [
   "kevin.ruscitti@enron.com"
  ],
  "Content-Transfer-Encoding": "7bit",
  "X-bcc": "",
  "parts": [
  {
   "content": "\n \n - RUSCITTI BID.wps \n\n",
   "contentType": "text/plain"
  }
 ],
 "X-Folder": "\\Ruscitti, Kevin\\Ruscitti, Kevin\\Inbox",
 "X-Origin": "RUSCITTI-K",
 "X-FileName": "Ruscitti, Kevin.pst",
 "Message-ID": "<8472651.1075845282216.JavaMail.evans@thyme>",
 "X-From": "Steven Anthony Pisano <spisano@sprintmail.com>",
 "Date": {
  "$date": 986163540000
  },
  "X-To": "KEVIN.RUSCITTI@ENRON.COM",
  "_id": {
   "$oid": "51a983dfe391e8ff964c5229"
  },
  "Content-Type": "text/plain; charset=us-ascii",
  "Mime-Version": "1.0"
 }
]
```

因为我们在查询之前，已经将数据经过处理导入到MongoDB中了，这基本是想要根据日期或（和）时间范围查询数据要做的所有事情了。尽管这看起来像是“不劳而获”，但查询的简易程度取决于你在重组

和导入阶段想要运行哪种类型的查询。如果你没有利用MongoDB将日期作为特殊字段的方式来导入数据，那你在查询前还需要做一些繁琐的工作。

另一件值得注意的事是Python的datetime（http://bit.ly/1a1nwjM）函数——由年、月、日组成——可以扩展到小时、分钟、秒，甚至微秒，还有可选的时区信息，都能够作为对元组的额外约束。小时的范围是0到23。比如（2013，12，25，0，23，5）表示2013年12月25日凌晨00：23：05。尽管它不是使用起来最方便的包，但是datetime绝对值得尝试，因为根据日期/时间范围来查询数据集是数据分析中最常见的工作。

6.3.2 发件人/收件人通信的分析模式

其他统计指标，比如某人最初写了多少封邮件、两个特定组之间发生了多少次直接通信，都是进行电子邮件分析的高度相关数据。在你开始分析谁和谁在交流之前，你或许需要枚举所有的发件人和收件人，也可以对查询进行限制，比如邮件的来源和发送地字段。首先，我们需要计算发送或接收的电子邮件地址的个数，参见示例6-9。

示例6-9：枚举邮件的发件人和收件人

```
import json
```

```
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo
client = pymongo.MongoClient()
db = client.enron
mbox = db.mbox
senders = [ i for i in mbox.distinct("From") ]
receivers = [ i for i in mbox.distinct("To") ]
cc_receivers = [ i for i in mbox.distinct("Cc") ]
bcc_receivers = [ i for i in mbox.distinct("Bcc") ]
print "Num Senders:", len(senders)
print "Num Receivers:", len(receivers)
print "Num CC Receivers:", len(cc_receivers)
print "Num BCC Receivers:", len(bcc_receivers)
```

上面数据集的示例输出如下：

```
   Num Senders: 7665
Num Receivers: 22162
Num CC Receivers: 6561
Num BCC Receivers: 6561
```

没有其他信息的话，这些对发件人和收件人的统计是很值得考虑的。平均来说，每封邮件发给了3个人，其中很大一部分是抄送（CC）和密送（BCC）。所以下一步就是过滤数据并使用基本的集合（set）操作（http://bit.ly/1a1l2Sw）（第1章中有所介绍），使用不同的规则的组合来决定哪些数据是重复的。我们只需要将每个特殊值的列表投影到集合上，这样我们就能使用不同类型的集合操作（http://bit.ly/1a1nxo2）了，包括交、并、差。表6-2显示了对少量发件人和收件人的基本操作，来说明集合操作在数据上是如何工作的：

```
   发件人 = {Abe, Bob}，收件人 = {Bob, Carol}
```

表6-2：集合操作示例

操作	操作名称	结果	注释
发件人∪收件人	并	Abe, Bob, Carol	所有不重复的发件人和收件人
发件人∩收件人	交	Bob	也是收件人的发件人
发件人－收件人	差	Abe	不是收件人的发件人
收件人－发件人	差	Carol	不是发件人的收件人

示例6-10演示了如何使用Python中的集合操作来计算数据。

示例6-10：使用集合操作分析发件人和收件人

```
   senders = set(senders)
receivers = set(receivers)
cc_receivers = set(cc_receivers)
bcc_receivers = set(bcc_receivers)
# Find the number of senders who were also direct receivers
senders_intersect_receivers = senders.intersection(receivers)
# Find the senders that didn't receive any messages
senders_diff_receivers = senders.difference(receivers)
# Find the receivers that didn't send any messages
receivers_diff_senders = receivers.difference(senders)
# Find the senders who were any kind of receiver by
# first computing the union of all types of receivers
all_receivers = receivers.union(cc_receivers, bcc_receivers)
senders_all_receivers = senders.intersection(all_receivers)
print "Num senders in common with receivers:", len(senders_intersect_receivers)
print "Num senders who didn't receive:", len(senders_diff_receivers)
print "Num receivers who didn't send:", len(receivers_diff_senders)
print "Num senders in common with *all* receivers:", len(senders_all_receivers)
```

下面脚本的示例输出显示了对邮件数据本质的详细理解：

```
   Num senders in common with receivers: 3220
Num senders who didn't receive: 4445
Num receivers who didn't send: 18942
Num senders in common with all receivers: 3440
```

在这种特殊情况下，收件人比发件人多很多，在7665个发件人当中，只有大约3220（少于一半）也是收件人。对于任意的邮件数据，乍一看似乎很令人吃惊，因为有太多的收件人并没有发过邮件，但记住我

们只是分析一个大公司中的一小组人的邮件数据而已。很多员工都会收到高级管理人员或者其他公司通讯中的群发邮件，却并不需要回复原始的发件人，这是很合理的。

另外，尽管我们的邮件（收到的和接收的）不仅仅局限于Enron公司，也包括全世界的，我们仍然只有一小部分数据。考虑到我们只是查看Enron员工中的一小组人的邮箱，我们并没有办法访问其他邮件域名的发件人，比如bob@example1.com或者jane@example2.com。

后来的判断引出一个有趣的问题，这是很好的后续练习，能够让我们更好地理解对收件箱的请求：基于Enron员工的邮件地址结尾是@enron.com这个假设，我们判断一下有多少发件人和收件人是Enron的员工。示例6-11演示了一种方法。

示例6-11：找到是Enron员工的发件人和收件人

```
   # In a Mongo shell, you could try this query for the same effect:
# db.mbox.find({"To" : {"$regex" : /.*enron.com.*/i} },
#             {"To" : 1, "_id" : 0})
senders = [ i
            for i in mbox.distinct("From")
                if i.lower().find("@enron.com") > -1 ]
receivers = [ i
              for i in mbox.distinct("To")
                  if i.lower().find("@enron.com") > -1 ]
cc_receivers = [ i
                 for i in mbox.distinct("Cc")
                     if i.lower().find("@enron.com") > -1 ]
bcc_receivers = [ i
                  for i in mbox.distinct("Bcc")
                      if i.lower().find("@enron.com") > -1 ]
print "Num Senders:", len(senders)
print "Num Receivers:", len(receivers)
print "Num CC Receivers:", len(cc_receivers)
print "Num BCC Receivers:", len(bcc_receivers)
```

这个脚本的示例输出如下：

```
   Num Senders: 3137
Num Receivers: 16653
Num CC Receivers: 4890
Num BCC Receivers: 4890
```

新的数据表明在原始的7665个发件人中，有3137个Enron员工，这说明剩下的发件人的邮件地址是其他的域名。数据也表明了这大约3000个发件人总共联系了大约17000个员工。《USA Today》对Enron的分析“The Enron scandal by the numbers”（http://usat.ly/1a1nxEp）揭露了当时Enron公司有大约20600个员工，所以我们的数据库中有多达80%的员工数据。

此时，我们的下一步应该是在通信中针对特定的电子邮件地址来分析。比如，数据集中原始的发件人是Enron的CEO Kenneth Lay的邮件有多少？通过研读我们脚本到现在为止枚举的输出，我们可以猜测出他的邮箱地址应该是kenneth.lay@enron.com。然而，更详细的调查显示，一些额外的别名也是我们需要考虑的。示例6-12提供了便于细致观察[1]的初始模板，并说明了如何使用MongoDB的$in操作符来查找满足特定条件的值。

示例6-12：计算发送/接收特定地址的电子邮件的邮件

```
   import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo
client = pymongo.MongoClient()
```

```
db = client.enron
mbox = db.mbox
aliases = ["kenneth.lay@enron.com", "ken_lay@enron.com", "ken.lay@enron.com",
           "kenneth_lay@enron.net", "klay@enron.com"] # More possibilities?
to_msgs = [ msg
            for msg in mbox.find({"To" : { "$in" : aliases } })]
from_msgs = [ msg
         for msg in mbox.find({"From" : { "$in" : aliases } })]
print "Number of message sent to:", len(to_msgs)
print "Number of messages sent from:", len(from_msgs)
```

这个脚本的示例输出有些令人吃惊。在我们加载的语料的子集中，几乎没有邮件是从Kenneth Lay的不同的邮箱地址中发送的：

```
   Number of message sent to: 1326
Number of messages sent from: 7
```

看起来似乎Enron语料库中有很多邮件发给了Enron的CEO，而由CEO发出的却很少——或者说，至少在我们考虑的收件箱（inbox）文件夹中是这样的[2]。（记住我们只选择加载Enron语料库里收件箱文件夹中的部分数据。加载更多的数据，比如发送项（sent items）的邮件，给读者留作练习以便深入探索）。下面两条是留给那些想对Kenneth Lay的邮件数据进行深入分析的读者的：

·大公司的总经理都拥有助手来进行多方面的沟通。Kenneth Lay的助手是Rosalee Fleming，他的邮件地址是rosalee.fleming@enron.com。尝试查找他的助手作为代理人的邮件。

·由于关联因素或（律师的）特权，法律案件极可能导致大量数据被篡改和删除。

如果你一直阅读的很认真，那么现在你可能会有疑问，也许你手边有工具能够回答大部分问题——尤其是在应用了前面章节的原理后。这些问题包括：

·从邮件的正文内容看，这些邮件是关于什么的？

·一封邮件的收件人最多有多少？（邮件是关于什么的？）

·哪两个人最经常互发邮件？（他们都讨论什么？）

·有多少邮件是一对一的？（一个发件人发给一个收件人，或者是一个发件人发给几个收件人的邮件，但不是像公司公告之类的群发。）

·多少邮件是在最长的回复链中的？（是关于什么的？）

Enron语料库是很多学术出版物分析的主题。有了对MongoDB的灵活使用，尤其是它提供的find操作符（http://bit.ly/1a1nt7Q），还有前面章节提到的数据聚合框架（http://bit.ly/1a1nA2Z）、索引功能（http://bit.ly/1a1nA3d）和其他文本挖掘技术，都给了你充足的工具来回答许多有趣的问题。当然，我们在这里的讨论就只能这么多了。

6.3.3 编写高级查询

MongoDB在2.2版本中引入了聚合框架（http://bit.ly/1a1nA2Z）。之

所以叫做聚合框架是因为它可以让你计算大量的集合体（对比于只能对数据进行原始的查询），包括分组、排序等操作的过程，这些都在MongoDB数据库里，而不需要使用Python脚本来调度规划多个查询。和任何一个框架一样，这种框架也有局限性，但是在你尝试过一两个例子之后就会感觉查询MongoDB的大体模式就很顺其自然了。聚合框架在建立一个查询时是一步一步进行的，因为一个聚合的查询过程就是一系列步骤。让我们来看一个查找一条被发出的信息的收件人的例子。在一个MongoDB的shell里，我们可以这样实现该查询：

```
  > db.mbox.aggregate(
 {"$match" : {"From" : "kenneth.lay@enron.com"} },
 {"$project" : {"From" : 1, "To" : 1} },
 {"$group" : {"_id" : "$From", "recipients" : {"$addToSet" : "$To" } } }
)
```

这个查询过程由三步构成，第一步是使用$match（http://bit.ly/1a1nAA5）操作符来寻找任何由特定电子邮件地址发送的消息。请记住我们可以像示例6-12中那样使用MongoDB的$in（http://bit.ly/1a1nAQA）操作符来提供一个扩充的选项列表用于匹配。通常来说，在聚合过程中尽早使用$match是最佳的方式，因为它可以减少后面每一步中的可能选项个数，从而减少总体的计算量。

下一步是使用$project（http://bit.ly/1a1nAQO）操作符来抽取每一条消息中From和To字段，因为我们只需要知道邮件的发件人和收件人。并且在接下来的步骤里我们会观察到From字段是使用$group操作符来进

行分组的，从而有效地把结果缩小成一个列表。

最后，就像上面提及的，$group（http://bit.ly/1a1nB7j）操作符规定From字段是分组的基础，分组结果中包含的To字段的值需要使用$addToSet操作符（http://bit.ly/1a1nCbg）将其赋给一个集合。紧接着有一个精简的结果集合来说明查询的最终形式。需要注意的是最终只有一个结果对象——一个包含_id和一个recipients字段的单独文档。recipients字段是记录了每一个和发件人（由_id标记）通信过的收件人列表：

```
  {
  "result" : [
    {
      "_id" : "kenneth.lay@enron.com",
      "recipients" : [
        [
          "j..kean@enron.com",
          "john.brindle@enron.com"
        ],
        [
          "e..haedicke@enron.com"
        ],
              ...2 more very large lists...
        [
        "mark.koenig@enron.com",
        "j..kean@enron.com",
        "pr<.palmer@enron.com>",
        "james.derrick@enron.com",
        "elizabeth.tilney@enron.com",
        "greg.whalley@enron.com",
        "jeffrey.mcmahon@enron.com",
        "raymond.bowen@enron.com"
        ],
        [
          "tim.despain@enron.com"
        ]
      ]
    }
  ],
  "ok" : 1
}
```

第一次见到使用聚合框架的查询的时候你可能会感到一点陌生和畏

缩，但是在MongoDB的shell里随便看看是不会帮助你了解它的工作原理的。最好的方式就是花一些时间去试着运行一下聚合查询的每一步，了解每一步中数据是怎么从上一步传输过来的。

你可以轻松地使用Python来操作MongoDB计算出的数据结构，但是现在让我们对相同的查询考虑另一个做法。主要是为了引入聚合框架中的一个新的操作符$unwind：

```
  > db.mbox.aggregate(
{"$match" : {"From" : "kenneth.lay@enron.com"} },
{"$project" : {"From" : 1, "To" : 1} },
{"$unwind" : "$To"},
{"$group" : {"_id" : "From", "recipients" : {"$addToSet" : "$To"}} }
)
```

这个查询的示例结果：

```
  {
"result" : [
  {
    "_id" : "kenneth.lay@enron.com",
    "recipients" : [
      "john.brindle@enron.com",
      "colleen.sullivan@enron.com",
      "richard.shapiro@enron.com",
              ...many more results...
      "juan.canavati.@enron.com",
      "jody.crook@enron.com"
    ]
  }
],
"ok" : 1
}
```

虽然我们最初的查询没有使用$unwind来得到对应每条消息特定收件人的分组，但是$unwind建立了一个如下格式的中间步骤，并且后来会传递到$group操作符中：

```
   {
  "result" : [
    {
      "_id" : ObjectId("51a983dae391e8ff964bc85e"),
      "From" : "kenneth.lay@enron.com",
      "To" : "tim.despain@enron.com"
    },
    {
      "_id" : ObjectId("51a983dae391e8ff964bdbc1"),
      "From" : "kenneth.lay@enron.com",
      "To" : "mark.koenig@enron.com"
    },
        ...many more results...
    {
      "_id" : ObjectId("51a983dde391e8ff964c241b"),
      "From" : "kenneth.lay@enron.com",
      "To" : "john.brindle@enron.com"
    }
  ],
  "ok" : 1
}
```

实际上，$unwind把列表中的每一项展开并和文档中其他字段组合。在这个特定的例子里，唯一需要注意的字段就是我们刚刚介绍过的From字段。在展开后，$group操作符可以对From字段进行分组并且有效地把所有收件人记录进一个列表中。这个列表与传递进来的结果数目相同，这也意味着最终结果里可能有一些重复。但是关键的是下一步的$group操作中$addToSet会消除这些重复，因为它把自己创建的列表当作一个集合，所以查询的结果是包含各不相同的收件人的列表。示例6-13使用PyMongo演示这些查询来简要展示MongoDB的聚合框架以及体现它能如何帮助你分析数据：

示例6-13：使用MongoDB的数据聚合框架

```
   import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo
```

```
# The basis of our query
FROM = "kenneth.lay@enron.com"
client = pymongo.MongoClient()
db = client.enron
mbox = db.mbox
# Get the recipient lists for each message
recipients_per_message = db.mbox.aggregate([
  {"$match" : {"From" : FROM} },
  {"$project" : {"From" : 1, "To" : 1} },
  {"$group" : {"_id" : "$From", "recipients" : {"$addToSet" : "$To" } } }
])['result'][0]['recipients']
# Collapse the lists of recipients into a single list
all_recipients = [recipient
                  for message in recipients_per_message
                      for recipient in message]
# Calculate the number of recipients per sent message and sort
recipients_per_message_totals =\
  sorted([len(recipients)
   for recipients in recipients_per_message])
# Demonstrate how to use $unwind followed by $group to collapse
# the recipient lists into a single list (with no duplicates
# per the $addToSet operator)
unique_recipients = db.mbox.aggregate([
  {"$match" : {"From" : FROM} },
  {"$project" : {"From" : 1, "To" : 1} },
  {"$unwind" : "$To"},
  {"$group" : {"_id" : "From", "recipients" : {"$addToSet" : "$To"}} }
])['result'][0]['recipients']
print "Num total recipients on all messages:", len(all_recipients)
print "Num recipients for each message:", recipients_per_message_totals
print "Num unique recipients", len(unique_recipients)
```

这个例子的结果会有些意想不到，结果表明只有少量的消息被从这个个体（正如我们之前提到过的）发送，包括从一些与一个单独个体的通信到一个群发给近1000人的电子邮件：

```
   Num total recipients on all messages: 1043
Num recipients for each message: [1, 1, 2, 8, 85, 946]
Num unique recipients 916
```

注意到奇怪的一点：大批邮件群发的收件人个数（946）比不重复的收件人的总数（916）还多。能猜猜这是为什么吗？再仔细地观察这些数据我们发现群发邮件的946个收件人里包含65个重复的地址。一个可能的解释是对于一个巨大的邮件地址列表的管理工作更容易出错，所

以重复地址的出现也不足为奇。另一个可能的解释是其中一些消息被发给多个收件人列表，而有些收件人在多个列表中出现。

6.3.4 根据关键词查找邮件

MongoDB有一系列强大的索引功能（http://bit.ly/1a1nA3d），而且2.4版引入了一个新的全文检索功能（http://bit.ly/1a1nEzO），它对在文档可索引字段上进行关键词搜索提供了简单而强大的接口。即使在2.4版本中这只是一项处于早期开发阶段的新功能并且有一些缺陷，但是对它进行介绍仍然是很有必要的，因为它可以帮助你搜索邮件消息和其他部分的内容。这本书的虚拟机和MongoDB在一起使用并且已经具有全文搜索特性，你所需要做的就是在你数据库中的集合上使用ensureIndex方法。

在MongoDB的shell里，你可以简单的输入以下两条命令建立一个对所有字段的全文索引，然后观察数据库的统计数据来确认索引是否真正被创建以及索引的规模。有一个特殊的$**字段是指“所有字段”，索引类型是“文本”，名字是“TextIndex”。仅仅占用了220MB的硬盘空间我们就有了一个全文索引来运行任何可以返回文档的查询，从而有助于关注一系列感兴趣的地方。

```
   > db.mbox.ensureIndex({"$**" : "text"}, {name : "TextIndex"})
> db.mbox.stats()
```

```
{
  "ns" : "enron.mbox",
  "count" : 41299,
  "size" : 157744000,
  "avgObjSize" : 3819.5597956366983,
  "storageSize" : 185896960,
  "numExtents" : 10,
  "nindexes" : 2,
  "lastExtentSize" : 56438784,
  "paddingFactor" : 1,
  "systemFlags" : 0,
  "userFlags" : 1,
  "totalIndexSize" : 221463312,
  "indexSizes" : {
    "_id_" : 1349040,
    "TextIndex" : 220114272
  },
  "ok" : 1
}
```

除了MongoDB shell的ensureIndex和PyMongo驱动的ensure_index（http://bit.ly/1a1nEA3）语法稍有不同外，创建索引和查询数据所做的与使用PyMongo大致相同，如示例6-14所示。此外，查询数据库统计数据或者运行文本索引查询的方法变化不大并且涉及PyMongo的command（http://bit.ly/1a1nCIq）方法。通常，command方法把命令作为第一个参数，把集合名称作为第二个参数，如果你想要把它的语法和MongoDB shell对应起来还需要一些相关的关键词作为额外参数（这样可以带来一些语法上的简便）。

示例6-14：使用PyMongo在MongoDB文档里创建一个文本索引

```
   import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo
client = pymongo.MongoClient()
db = client.enron
mbox = db.mbox
# Create an index if it doesn't already exist
mbox.ensure_index([("$**", "text")], name="TextIndex")
# Get the collection stats (collstats) on a collection
# named "mbox"
```

```
print json.dumps(db.command("collstats", "mbox"), indent=1)
# Use the db.command method to issue a "text" command
# on collection "mbox" with parameters, remembering that
# we need to use json_util to handle serialization of our JSON
print json.dumps(db.command("text", "mbox",
                            search="raptor",
                            limit=1),
                 indent=1, default=json_util.default)
```

MongoDB的全文检索功能非常强大，你可以查看一下文本检索文档（http://bit.ly/1a1nDfb）来鉴别什么功能是可以做到的。你可以从一个术语列表中查找任何一个术语、查找特定的短语、统计禁止搜索结果中某个特定术语的出现。在一开始所有字段有着相同的权重，但是为了调整搜索结果也可以赋予不同字段不同的权重（http://bit.ly/1a1nFDO）。

比如你正在Enron公司语料库搜索一个电子邮件地址，你可能希望给予To：字段和From：字段更大的权重，而给Cc：（抄送）和Bcc：（密送）字段较小的权重，从而得到一个排序更优的结果。如果你需要搜索关键字，你可能希望这些关键字出现在消息的主题里会比出现在正文里有更大的权重。

在Enron公司的内容里，从会计学角度来看raptors是指用来隐藏巨额负债的经济手段。下面是一些在MongoDB shell中对于这个贬义词raptor（http://bit.ly/1a1nFE6）使用文本检索的简略示例结果：

```
   > db.mbox.runCommand("text", {"search" : "raptor"})
{
  "queryDebugString" : "raptor||||||",
    "language" : "english",
    "results" : [
        {
```

```
 "score" : 2.0938471502590676,
 "obj" : {
   "_id" : ObjectId("51a983dfe391e8ff964c63a7"),
   "Content-Transfer-Encoding" : "7bit",
   "From" : "joel.ephross@enron.com",
   "X-Folder" : "\\SSHACKL (Non-Privileged)\\Shackleton, Sara\\Inbox",
   "Cc" : [
     "mspradling@velaw.com"
   ],
   "X-bcc" : "",
   "X-Origin" : "Shackleton-S",
   "Bcc" : [
     "mspradling@velaw.com"
   ],
   "X-cc" : "'mspradling@velaw.com'",
   "To" : [
     "maricela.trevino@enron.com",
     "sara.shackleton@enron.com",
     "mary.cook@enron.com",
     "george.mckean@enron.com",
     "brent.vasconcellos@enron.com"
   ],
   "parts" : [
     {
       "content" : "Maricela, attached is a draft of one of the...",
       "contentType" : "text/plain"
     }
   ],
   "X-FileName" : "SSHACKL (Non-Privileged).pst",
   "Mime-Version" : "1.0",
   "X-From" : "Ephross, Joel </O=ENRON/OU=NA/CN=RECIPIENTS/CN=JEPHROS>",
   "Date" : ISODate("2001-09-21T12:25:21Z"),
   "X-To" : "Trevino, Maricela </O=ENRON/OU=NA/CN=RECIPIENTS/CN=Mtr...",
   "Message-ID" : "<28660745.1075858812819.JavaMail.evans@thyme>",
   "Content-Type" : "text/plain; charset=us-ascii",
   "Subject" : "Raptor"
 },
     ...73 more results...
     {
"score" : 0.5000529829394935,
"obj" : {
  "_id" : ObjectId("51a983dee391e8ff964c363b"),
  "X-cc" : "",
  "From" : "sarah.palmer@enron.com",
  "X-Folder" : "\\ExMerge - Martin, Thomas A.\\Inbox",
  "Content-Transfer-Encoding" : "7bit",
  "X-bcc" : "",
  "X-Origin" : "MARTIN-T",
  "To" : [
    "sarah.palmer@enron.com"
  ],
  "parts" : [
    {
      "content" : "\nMore than one Enron official warned company...",
      "contentType" : "text/plain"
    }
  ],
  "X-FileName" : "tom martin 6-25-02.PST",
  "Mime-Version" : "1.0",
  "X-From" : "Palmer, Sarah </O=ENRON/OU=NA/CN=RECIPIENTS/CN=SPALME2>",
  "Date" : ISODate("2002-01-18T14:32:00Z"),
```

```
        "X-To" : "Palmer, Sarah </O=ENRON/OU=NA/CN=RECIPIENTS/CN=Spalme2>",
        "Message-ID" : "<8664618.1075841171256.JavaMail.evans@thyme>",
        "Content-Type" : "text/plain; charset=ANSI_X3.4-1968",
        "Subject" : "Enron Mentions -- 01/18/02"
       }
     }
  ],
  "stats" : {
    "nscanned" : 75,
    "nscannedObjects" : 0,
    "n" : 75,
    "nfound" : 75,
    "timeMicros" : 230716
  },
  "ok" : 1
}
```

既然你已经对与Enron公司相关的raptor熟悉了，你可能会发现搜索结果中一个排名靠前的消息的前几行很有用：

对于在Raptor结构中套期保值的资产的季度性评估是通过标准季度评估流程来的。业务单元RAC和安达信会计师事务所都已经在最初的评估中签署生效。所有Raptor里的投资都在MPR并且被这些业务单元操控，基于这些信息我们已经准备了Raptor职位报告……

如果你徘徊在众多消息中不知从哪里开始，简单地搜索一个关键字一定会给你带来一个好的开端。刚才引用的消息的主题是“RE：Raptor Debris”。去了解一下还有谁在这个讨论组或其他讨论组中进行关于Raptor的讨论难道不是很好吗？你现在有了工具和如何找到它的方法。

B-树是最好的吗？

B-树（http://bit.ly/1a1nFUz）是MongoDB和多数其他数据库系统的基本数据结构，因为从长远来看，即使不断面临最坏的状况它们也显示

出对核心操作（搜索、插入、更新、删除）的高效率的表现。在计算机科学的术语中，他们对于核心操作的性能被称为“对数性能”，使用第3章介绍的“大O”法表示为O（log n）。B-树必须保持平衡，并保持数据有序排列。

这些特点会产生高效的查找，因为底层实现只需要最低限度的磁盘读取。由于仍然可以快速地对传统的基于磁盘的硬盘驱动器进行寻道（数毫秒左右），这就意味着同样可以快速地访问B-树存储的大量数据。MongoDB强烈依赖B-树，不仅仅为了在特定字段或者字段组合上建立二级索引（http://bit.ly/1a1nA3d），还为了支持它的地理空间索引（http://bit.ly/1a1nGrt）和文本检索索引（http://bit.ly/1a1nEzO），这些会在这个部分中介绍。

人们对于B-树这个词的来源并没有统一的认识，但是普遍认同B代表B-树的发明者Bayer。网上有更多B-树和它们常见变形的信息。如果你决定深入研究MongoDB，那么你应该尽可能多了解它们的理论和实现，因为在MongoDB的设计中它们是不可或缺的。

[1] 在这个特定的语境中，“细致观察”就是在ipynb/resources06-mailboxes/data/enron_mail_20110402/enron_data/maildir/lay-k/inboxdirectory中，对“lay@enron”的查找（在Unix或者Linux终端中是grep 'lay@enron'*命令）。这表明还有一些电子邮件地址是可能存在的。

[2] 对kenneth.lay@enron.com（在Unix或者Linux中是grep-R"From:kenneth.lay@enron.com"*命令）和其他邮件别名（也可能在Enron语料库的ipynb/resources06-mailboxes/data/enron_mail_20110402/enron_data/maildir/lay-k文件夹里的邮件标头中出现）的查找显示出了新的结果。这说明我们关注的Enron语料库的部分数据中，确实没有过多的发送邮件。

6.4 探索和可视化时序趋势

可视化邮件数据的方法有很多，这也是很多出版物和开源项目的主题，你可以查找一下寻求灵感。至今为止我们使用的可视化方法也是一个不错的选择。作为一个开始，我们先实现一种之前提到的考虑频域分析的方法，并且用IPython Notebook来获得一个有意义的方法。举个例子，我们可以以时间为标准来把数据以图或列表的形式表示，协助分析它的趋势，比如一周中的哪天或者一天的哪段时间电子邮件收发最多。另外可能做的事包括建立一个代表发件人和收件人关系的图表，并且根据关键词来过滤信息内容或主线，或者计算能比我们之前完成的初级计数反应更深层次的内在关系的直方图。

示例6-15是一个演示如何使用MongoDB以日期/时间因素来对消息计数的聚合查询。这个查询分为三步。第一步把日期拆解成其构成部分的子文档；第二步根据各个字段和其_id分组，并且使用内置的$sum（http://bit.ly/1a1nGYq）函数进行计数求和，这个函数经常与$group结合使用；第三步是根据年月排序。

示例6-15：以日期/时间对消息计数的聚合查询

```
   import json
import pymongo # pip install pymongo
from bson import json_util # Comes with pymongo
client = pymongo.MongoClient()
db = client.enron
```

```
mbox = db.mbox
results = mbox.aggregate([
{
  # Create a subdocument called DateBucket with each date component projected
  # so that these fields can be grouped on in the next stage of the pipeline
  "$project" :
  {
     "_id" : 0,
     "DateBucket" :
     {
       "year" : {"$year" : "$Date"},
       "month" : {"$month" : "$Date"},
       "day" : {"$dayOfMonth" : "$Date"},
       "hour" : {"$hour" : "$Date"},
     }
  }
},
{
  "$group" :
  {
    # Group by year and date by using these fields for the key.
    "_id" : {"year" : "$DateBucket.year", "month" : "$DateBucket.month"},
    # Increment the sum for each group by 1 for every document that's in it
    "num_msgs" : {"$sum" : 1}
  }
},
{
  "$sort" : {"_id.year" : 1, "_id.month" : 1}
}
])
print results
```

以年月排序后的示例查询结果为：

```
   {u'ok': 1.0,
 u'result': [{u'_id': {u'month': 1, u'year': 1997}, u'num_msgs': 1},
  {u'_id': {u'month': 1, u'year': 1998}, u'num_msgs': 1},
  {u'_id': {u'month': 12, u'year': 2000}, u'num_msgs': 1},
  {u'_id': {u'month': 1, u'year': 2001}, u'num_msgs': 3},
  {u'_id': {u'month': 2, u'year': 2001}, u'num_msgs': 3},
  {u'_id': {u'month': 3, u'year': 2001}, u'num_msgs': 21},
  {u'_id': {u'month': 4, u'year': 2001}, u'num_msgs': 811},
  {u'_id': {u'month': 5, u'year': 2001}, u'num_msgs': 2118},
  {u'_id': {u'month': 6, u'year': 2001}, u'num_msgs': 1650},
  {u'_id': {u'month': 7, u'year': 2001}, u'num_msgs': 802},
  {u'_id': {u'month': 8, u'year': 2001}, u'num_msgs': 1538},
  {u'_id': {u'month': 9, u'year': 2001}, u'num_msgs': 3538},
  {u'_id': {u'month': 10, u'year': 2001}, u'num_msgs': 10630},
  {u'_id': {u'month': 11, u'year': 2001}, u'num_msgs': 9219},
  {u'_id': {u'month': 12, u'year': 2001}, u'num_msgs': 4541},
  {u'_id': {u'month': 1, u'year': 2002}, u'num_msgs': 3611},
  {u'_id': {u'month': 2, u'year': 2002}, u'num_msgs': 1919},
  {u'_id': {u'month': 3, u'year': 2002}, u'num_msgs': 514},
  {u'_id': {u'month': 4, u'year': 2002}, u'num_msgs': 97},
  {u'_id': {u'month': 5, u'year': 2002}, u'num_msgs': 85},
  {u'_id': {u'month': 6, u'year': 2002}, u'num_msgs': 166},
  {u'_id': {u'month': 10, u'year': 2002}, u'num_msgs': 1},
```

```
{u'_id': {u'month': 12, u'year': 2002}, u'num_msgs': 1},
{u'_id': {u'month': 2, u'year': 2004}, u'num_msgs': 26},
{u'_id': {u'month': 12, u'year': 2020}, u'num_msgs': 2}]}
```

正如看到的，这个查询计算了每一年每一月的消息数目，但是你可以用多种方式对其稍加修改从而探索通信的模式。例如，你可以使用$DateBucket.day或者$DateBucket.hour来显示每周的哪一天或者每天的哪个小时消息数目最多。如果你觉得日期或时间的范围也值得考虑，那么你可以使用$gt和$lt操作符。

也可以使用模运算来把数值分解到固定范围里，比如每天的小时数。下面这个例子是MongoDB查询文档的一部分。

```
"hour" : {"$subtract" : [
                        {"$hour" : "$Date"},
                        {"$mod" : [{"$hour" :"$Date"} , 2]}
                    ]
```

这个查询执行从日期中提取出小时部分，如果其数值不能被2整除就减1，从而将小时分成两位的间隔。仔细研究一下这段代码然后看看你能从数据中发现什么。这类聚合查询的优点就是MongoDB帮你做了所有的工作，而不是简单地返回给你未处理过的数据。

通常对这种信息最简单的显示方式就是表格。示例6-16展示了怎么使用前面章节介绍过的prettytable包来让数据变的直观易懂。

示例6-16：把时序结果变成整齐的表格

```
   from prettytable import PrettyTable
pt = PrettyTable(field_names=['Year', 'Month', 'Num Msgs'])
pt.align['Num Msgs'], pt.align['Month'] = 'r', 'r'
[ pt.add_row([ result['_id']['year'], result['_id']['month'], result['num_msgs'] ])
  for result in results['result'] ]
print pt
```

下面这个表格显示了每个月邮件数量[1]，并且突出了一个异常状况：2001年十月的邮件数据量是之前任何一个月的3倍左右！原因是2001年10月Enron公司的丑闻被揭露了，引起了巨大数目的通信并且直到两个月后才逐渐消减。

```
   +------+-------+----------+
| Year | Month | Num Msgs |
+------+-------+----------+
| 1997 |     1 |        1 |
| 1998 |     1 |        1 |
| 2000 |    12 |        1 |
| 2001 |     1 |        3 |
| 2001 |     2 |        3 |
| 2001 |     3 |       21 |
| 2001 |     4 |      811 |
| 2001 |     5 |     2118 |
| 2001 |     6 |     1650 |
| 2001 |     7 |      802 |
| 2001 |     8 |     1538 |
| 2001 |     9 |     3538 |
| 2001 |    10 |    10630 |
| 2001 |    11 |     9219 |
| 2001 |    12 |     4541 |
| 2002 |     1 |     3611 |
| 2002 |     2 |     1919 |
| 2002 |     3 |      514 |
| 2002 |     4 |       97 |
| 2002 |     5 |       85 |
| 2002 |     6 |      166 |
| 2002 |    10 |        1 |
| 2002 |    12 |        1 |
| 2004 |     2 |       26 |
| 2020 |    12 |        2 |
+------+-------+----------+
```

使用一些分析人类语言数据（如之前章节中介绍过的）的技术来对2001年10月和11月数据进行分析，我们可以从发件人和收件人的角度以

及语言用词角度发现一些交流方式本质上的区别。

还有更多的其他方式来实现邮件数据的可视化，在本章的推荐习题中也有提及。使用IPython Notebook的表格库可能是下一步要考虑的方式。

[1] 有两封写于2020年的邮件是初始导出邮件数据时出现的bug，它不在我们的控制范围之内。

6.5 分析你自己的邮件数据

Enron邮件数据在邮件分析这一章中是非常好的例子，不过你或许想试着分析自己的邮件数据。幸运的是，许多流行的邮件客户端都提供“导出为mbox”的功能，有了这种可供分析的格式，就让使用本章的技术来分析邮件变得很简单了。

比如，在Apple Mail中，你可以选择一些邮件，点选“File”（文件）菜单中的“Save As”（另存为），然后选择“Raw Message Source”（原始信息源）作为格式选择，来将邮件导出为mbox文件（参见图6-2）。在网上搜索一下，就能找到大多数主流客户端是如何操作的了。

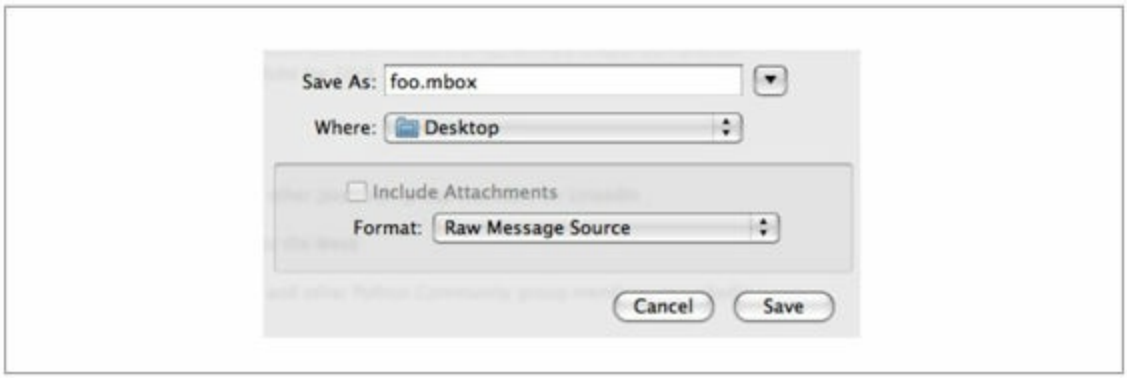


图6-2：大多数邮件客户端提供导出你的邮件数据到mbox文件的选项

如果你只使用在线邮件客户端，你可以选择将你的数据传输到一个邮件客户端，并导出它。但是你可能更愿意直接从服务器中导出数据，完全自动化mbox文件的创建过程。几乎所有的在线邮件服务都支持

POP3（Post Office Protocol version 3，邮局协议版本3）。大多数服务也支持IMAP（Internet Message Access Protocol，Internet邮件访问协议），导出邮件的Python脚本写起来并不困难。

一个非常强大的命令行工具是用Python编写的getmail（http://bit.ly/1a1nKaL），它可以用于从任何地方导出数据。Python标准库中包含两个模块：poplib（http://bit.ly/1a1nI2G）和imaplib（http://bit.ly/1a1nIj5），它们提供了很好的基础。如果你在网上搜索，也可能会发现很多有用的脚本。getmail是很容易启动和运行的。例如，为了导出你的Gmail收件箱数据，你只需要下载并安装它，然后使用一些基本选项来建立getmailrc配置文件。

下面的示例配置展示了对*nix环境的设置。Windows用户需要将[destination]中的path和[options]中的message_log值改为有效路径，不过别忘了，如果你需要*nix环境，你可以选择在虚拟机上运行本书的脚本：

```
   [retriever]
type = SimpleIMAPSSLRetriever
server = imap.gmail.com
username = ptwobrussell
password = xxx
[destination]
type = Mboxrd
path = /tmp/gmail.mbox
[options]
verbose = 2
message_log = ~/.getmail/gmail.log
```

修改好配置后，接下来只需要从终端调用getmail就可以了。一旦有

了本地的mbox文件，就可以使用本章学过的技术来分析它了。下面是你的邮件数据通过使用getmail命令之后的示例数据：

```
   $ getmail
getmail version 4.20.0
Copyright (C) 1998-2009 Charles Cazabon.  Licensed under the GNU GPL version 2.
SimpleIMAPSSLRetriever:ptwobrussell@imap.gmail.com:993:
  msg  1/10972 (4227 bytes) from ... delivered to Mboxrd /tmp/gmail.mbox
  msg  2/10972 (3219 bytes) from ... delivered to Mboxrd /tmp/gmail.mbox
...
```

6.5.1 通过OAuth访问你的Gmail

在2010年早期，Google宣布了OAuth access to IMAP and SMTP in Gmail（通过OAuth访问Gmail的IMAP和SMTP）（http://bit.ly/1a1nIzH）。这非常有意义，因为这是官方将“Gmail as a platform”（Gmail平台）开放出去，允许第三方开发者使用Gmail数据来开发应用，而不需要用户名和密码。这一节并不会详细介绍Xoauth的原理以及Google是如何实现OAuth（http://bit.ly/1a1nKHV）的（参见附录B中对OAuth的概述）；相反，我们只是访问Gmail数据，这只需要简单的几步：

1.在Gmail账户设置中，选中“Forwarding and POP/IMAP”（转发和POP/IMAP）选项卡中的“Enable IMAP”（允许IMAP）选项。

2.访问Google Mail Xoauth工具的wiki页（http://bit.ly/1a1nKYa），

下载xoauth.py命令行工具，根据指示生成“匿名”用户[1]的OAuth token和secret。

3.使用pip install oauth2命令安装python-oauth2（http://bit.ly/1a1nJnh），使用示例6-17的模板来建立连接。

示例6-17：通过Xoauth连接Gmail

```
   import sys
import oauth2 as oauth
import oauth2.clients.imap as imaplib
# See http://code.google.com/p/google-mail-xoauth-tools/wiki/
# XoauthDotPyRunThrough for details on obtaining and
# running xoauth.py to get the credentials
OAUTH_TOKEN = ''  # XXX: Obtained with xoauth.py
OAUTH_TOKEN_SECRET = ''  # XXX: Obtained with xoauth.py
GMAIL_ACCOUNT = ''  # XXX: Your Gmail address - example@gmail.com
url = 'https://mail.google.com/mail/b/%s/imap/' % (GMAIL_ACCOUNT, )
# Standard values for Gmail's Xoauth
consumer = oauth.Consumer('anonymous', 'anonymous')
token = oauth.Token(OAUTH_TOKEN, OAUTH_TOKEN_SECRET)
conn = imaplib.IMAP4_SSL('imap.googlemail.com')
conn.debug = 4  # Set to the desired debug level
conn.authenticate(url, consumer, token)
conn.select('INBOX')
# Access your INBOX data
```

当你做到可以通过程序访问你的邮箱，下一步就是获取解析邮件数据。我们将格式化数据并导出，遵循与本章前面相同的标准，这样你所有的脚本和工具都能在Enron语料库和自己的邮件数据中使用了！

6.5.2 通过IMAP获取和解析邮件

IMAP协议比较复杂，但好消息是，只是查找和获取邮件的话，你

并不需要了解过多的细节。另外，遵循imaplib的示例可以在线获得（http://bit.ly/1a1nJDG）。

常见的操作之一是查找邮件。建立IMAP请求有很多不同的方法。例如，查找特定用户的邮件是conn.search（None，'（FROM"me"）'），其中None是可选的参数，表示字符集，'（FROM"me"）'是查找你自己发送的邮件的命令（Gmail中将“me”视为当前授权的用户）。查找包含“foo”的邮件的命令是'（SUBJECT"foo"）'，许多额外的信息可以从6.4.4中的RFC 3501（http://bit.ly/XSB5cQ）中获得，它定义了IMAP标准。imaplib返回的是元祖形式的查找结果，包括状态码和空格分隔的邮件ID列表，如（'OK'，['506527566']）。你可以解析这些邮件ID值来获得RFC 822-compliant（http://bit.ly/1a1nMzx）邮件数据。但是要将邮件数据的内容解析为可用的格式还需要额外的工作。

幸运的是，我们可以复用示例6-3中的使用email模块来解析邮件数据为更可用格式的代码，这些代码也处理了有些看似无用的邮件信息，因为它们可能对获取有用的内容很有必要，参见示例6-18。

示例6-18：请求Gmail收件箱并将结果存储为JSON格式

```
   import sys
import mailbox
import email
import quopri
import json
import time
from BeautifulSoup import BeautifulSoup
from dateutil.parser import parse
# What you'd like to search for in the subject of your mail.
```

```
# See Section 6.4.4 of http://www.faqs.org/rfcs/rfc3501.html
# for more SEARCH options.
Q = "Alaska" # XXX
# Recycle some routines from Example 6-3 so that you arrive at the
# very same data structure you've been using throughout this chapter
def cleanContent(msg):
    # Decode message from "quoted printable" format
    msg = quopri.decodestring(msg)
    # Strip out HTML tags, if any are present.
    # Bail on unknown encodings if errors happen in BeautifulSoup.
    try:
        soup = BeautifulSoup(msg)
    except:
        return ''
    return ''.join(soup.findAll(text=True))
def jsonifyMessage(msg):
    json_msg = {'parts': []}
    for (k, v) in msg.items():
        json_msg[k] = v.decode('utf-8', 'ignore')
    # The To, Cc, and Bcc fields, if present, could have multiple items.
    # Note that not all of these fields are necessarily defined.
    for k in ['To', 'Cc', 'Bcc']:
        if not json_msg.get(k):
            continue
        json_msg[k] = json_msg[k].replace('\n', '').replace('\t', '')\
                                 .replace('\r', '').replace(' ', '')\
                                 .decode('utf-8', 'ignore').split(',')
    for part in msg.walk():
        json_part = {}
        if part.get_content_maintype() == 'multipart':
            continue
        json_part['contentType'] = part.get_content_type()
        content = part.get_payload(decode=False).decode('utf-8', 'ignore')
        json_part['content'] = cleanContent(content)
        json_msg['parts'].append(json_part)
    # Finally, convert date from asctime to milliseconds since epoch using the
    # $date descriptor so it imports "natively" as an ISODate object in MongoDB.
    then = parse(json_msg['Date'])
    millis = int(time.mktime(then.timetuple())*1000 + then.microsecond/1000)
    json_msg['Date'] = {'$date' : millis}
    return json_msg
# Consume a query from the user. This example illustrates searching by subject.
(status, data) = conn.search(None, '(SUBJECT "%s")' % (Q, ))
ids = data[0].split()
messages = []
for i in ids:
    try:
        (status, data) = conn.fetch(i, '(RFC822)')
        messages.append(email.message_from_string(data[0][1]))
    except Exception, e:
        print e
        print 'Print error fetching message %s. Skipping it.' % (i, )
print len(messages)
jsonified_messages = [jsonifyMessage(m) for m in messages]
# Separate out the text content from each message so that it can be analyzed.
content = [p['content'] for m in jsonified_messages for p in m['parts']]
# Content can still be quite messy and contain line breaks and other quirks.
filename = os.path.join('resources/ch06-mailboxes/data',
                        GMAIL_ACCOUNT.split("@")[0] + '.gmail.json')
f = open(filename, 'w')
f.write(json.dumps(jsonified_messages))
```

```
f.close()
print>> sys.stderr, "Data written out to", f.name
```

一旦你成功的解析了Gmail邮件的正文，还需要一些额外的工作来处理文本，使之能够友好的展示或者是像第5章中高级的NLP那样。然而，对于搭配分析等的处理也并不需要花费过多的工夫。事实上，示例6-18的结果可以直接提供给示例4-12，来从搜索结果中产生一个搭配列表。很有价值的可视化练习是创建一个图形，根据自定义的指标基于共有的二元组个数绘制消息之间链接的强度。

6.5.3 “Graph Your Inbox”Chrome扩展对Gmail的可视化模式

分析网页邮箱有几个有用的工具包，最近几年出现了一个Graph Your Inbox Chrome extension（Chrome扩展程序）（http://bit.ly/1a1nMQ2）非常有前景。为了使用这个扩展，只需要安装并授权它访问你的邮件数据、运行一些Gmail查询，其余部分就可以自动处理了。可以搜索“pizza”这样的关键字和“2010”这样的时间值，或者运行更高级的查询，如“from：matthew@zaffra.com”和“label：Strata”。图6-3显示了一个示例截图。

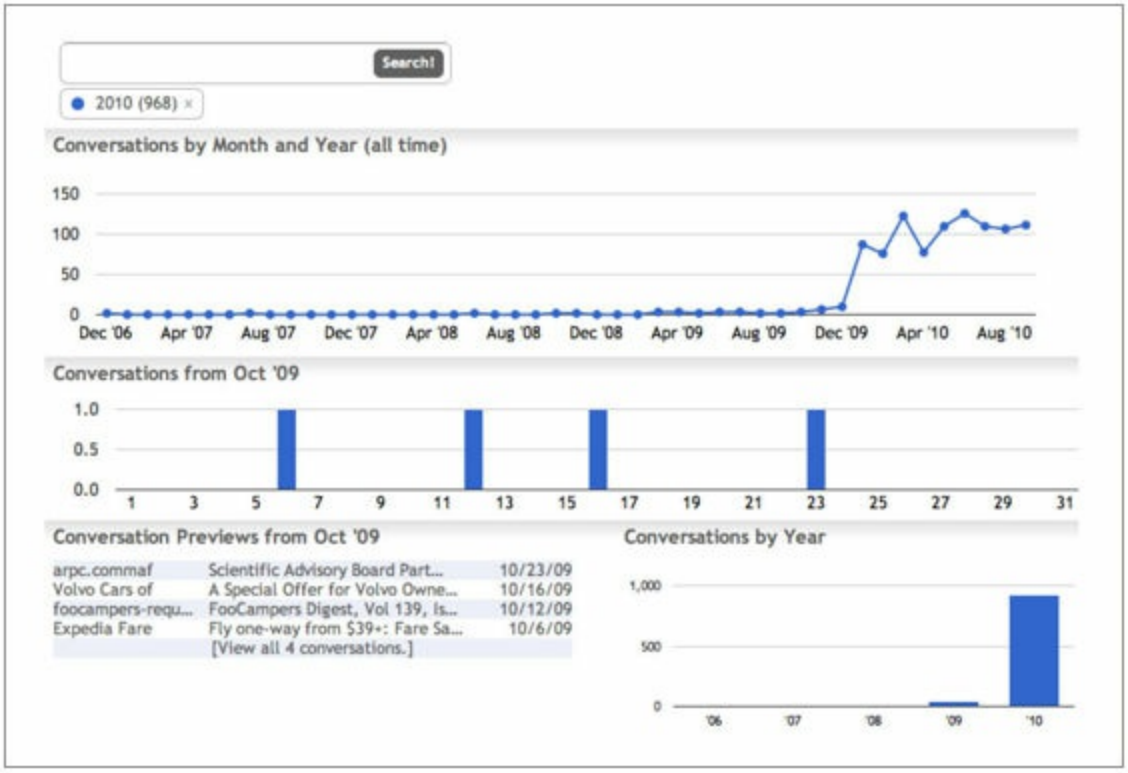


图6-3：Graph Your Inbox Chrome扩展程序提供了Gmail活动的简明摘要

你可以使用这章学习的方法来对这个扩展提供的所有分析举一反三，比如使用IPython Notebook中的Javascript的可视化库D3.js或者matplotlib的绘制工具。你的工具箱中有很多脚本和技术都可以应用在数据域中，不管是对邮箱、网页，还是推文的集合，都可以产生精简展示结果。你当然可以考虑设计一款综合应用，这样会提供良好的用户体验。为用户进行数据科学和分析的构建块已在你的掌握之中。

[1] 如果你只是获取自己的Gmail邮件数据，使用xoauth.py生成的匿名用户证书即可；之后如果需要，你可以随时注册并建立一个“可信的”客户端应用（http://bit.ly/1a1nJ6N）。

6.6 本章小结

本章讨论了很多基础知识——比目前为止任何一章都基础，但却是值得期待的。每一章都是建立在前一章的基础上，尝试讲解关于数据分析的故事，而我们现在已经学习了本书的一大半了。尽管我们只是学习了邮件数据的皮毛，你却可以利用前面章节的知识来探索更多，比如根据你的个人邮件数据探索社交联系和个人生活，这会给数据分析增添许多乐趣。

我们的焦点聚集在mbox上，这是一个简单而方便的文件格式。它有很高的可移植性，并能够用很多Python工具包来分析。当处理复杂的数据，比如邮件数据时，你还有希望领会到使用标准的跨平台格式的价值。有很多挖掘mbox数据的开源技术，而Python是一个提供从不同角度分析数据的很棒工具。在这些工具和合适存储媒体（比如MongoDB）上的少量投资，具有深远的影响，让你专注于手头的问题而不是工具本身。

注意：本章和其他章节的源代码在GitHub（http://bit.ly/1a1kNqy）上有方便的IPython Notebook格式，强烈建议你使用自己的浏览器去尝试一下。

6.7 推荐练习

·从Enron语料库里选取一些素材分析一下作为练习。例如，通过在线阅读资料或者看纪录片来研究一下Enron公司的案例，然后选取10~15个感兴趣的邮箱，使用本章节介绍的技术看看你能发现什么通信交流的模式。

·用之前章节介绍的方法对邮件内容进行文本分析。你能把他们说的话关联起来吗？与之前章节介绍的信息检索概念比起来，全文索引有什么优势和不足？

·研究一下MongoDB的Map-Reduce框架（http://bit.ly/1a1nN6o）。

·研究一下MongoDB的地理空间索引。你能存储例如LinkedIn数据或者Twitter数据里的地理坐标这样的位置，然后有效的查询它们吗？

·阅读了解一下MongoDB的全文索引在内部是怎样实现的（http://bit.ly/1a1nPuY）。特别地，复习一下它使用的Snowball词干提取器（http://bit.ly/1a1nNDk）。

·复习一下用于从邮箱重建邮件会话的有效的启发式算法——电子邮件聚合算法（http://bit.ly/1a1nQ23）。一个示例实现可以在本书第1版的部分（遗留下来的）源代码中找到（http://bit.ly/1a1nQ2e）。

·使用SIMILE时间轴（http://bit.ly/1a1nQz3）工程对上述的邮件聚合算法中的聚合结果进行可视化。有很多时间轴的在线（http://bit.ly/1a1nOr1）展示，还有丰富的文档说明。这个简单绘制邮件时间轴的示例只是一个简单的开始。

·在Google学术上对“Enron”（http://bit.ly/1a1nR6c）进行搜索，阅读并学习大量的相关文献。这些文献会对你的自身学习有所启发。

6.8 在线资源

下面是本章中有用的链接的列表：

·B-树（http://bit.ly/1a1nFUz）

·可下载的Enron语料库（http://bit.ly/1a1nmsU）

·Enron语料库（http://bit.ly/1a1nj01）

·http://www.enron-mail.com

·Enron丑闻（http://bit.ly/1a1nuZo）

·Google学术上的Enron的白皮书（http://bit.ly/1a1nR6c）

·Envoy的GitHub仓库（http://bit.ly/1a1nrwL）

·getmail（http://bit.ly/1a1nKaL）

·Google Mail Xoauth工具的wiki页（http://bit.ly/1a1nKYa）

·Graph Your Inbox的Chrome扩展程序（http://bit.ly/1a1nMQ2）

·Git for Windows（http://bit.ly/1a1nubC）

·MongoDB全文索引的工作原理（http://bit.ly/1a1nPLr）

·JWZ邮件聚合算法（http://bit.ly/1a1nQ23）

·Map-Reduce（http://bit.ly/1a1nN6o）

·MongoDB（http://bit.ly/1a1nlVK）

·MongoDB数据聚合框架（http://bit.ly/1a1nA2Z）

·MongoDB全文检索（http://bit.ly/1a1nEzO）

·MongoDB索引（http://bit.ly/1a1nA3d）

·SIMILE时间轴的在线展示（http://bit.ly/1a1nOr1）

·PyMongo文档（http://bit.ly/1a1nqZE）

·RFC 2045（http://bit.ly/1a1nnNp）

·SIMILE Timeline（SIMILE时间轴）（http://bit.ly/1a1nQz3）

·Snowball词干提取器（http://bit.ly/1a1nNDk）

·subprocess（http://bit.ly/1a1nrgb）

·Xoauth（http://bit.ly/1a1nKHV）

第7章　挖掘GitHub：检查软件协同习惯、构建兴趣图谱等

GitHub近年来演化为极其重要的社交编程平台，看似简单的前提是：它能够为开发者提供优异的主机解决方案。该方案用来建立和维护开源软件项目，其中这些项目使用叫做Git（http://bit.ly/16mhOep）的开源分布式版本控制（http://bit.ly/1a1o1u8）系统。不像其他的版本控制系统那样，比如CVS（http://bit.ly/1a1nZCI）或者Subversion（http://bit.ly/1a1o21g），Git本身并不需要像传统方式那样复制代码库。所有的副本都是工作副本（working copy），开发者可以在工作时的副本上提交本地的变化，而不需要连接中央服务器。

分布式版本控制模式在GitHub的社交编程概念中表现得尤其好，因为当开发者对一个工程的改造感兴趣时，GitHub允许开发者派生（fork）资源库的工作副本，这样就可以立即开始对代码进行改造了，就和代码的原始拥有者的工作方式一模一样。Git不止记录了允许资源库任意被fork的语义信息，同时也使合并派生的子资源库和父资源库的变更相对容易。通过GitHub用户接口，这个工作流程被称之为拉取请求（pull request）。

虽然这个概念看似简单，开发者在开销很小的工作流程中建立项目

并共同协作的能力改善了许多琐碎的细节，这些细节阻碍了开源发展中的创新（一旦你理解一些Git工作原理的基本细节），包括超越了数据可视化和与其他系统的协作能力的一些便利。换句话说，把GitHub想象为开源软件开发的使用者。同样的，尽管数十年来，开发者在编程项目上都有合作，像GitHub这样的主机平台促进了合作，并且采用前所未有的方式进行了创新。它将建立项目、共享源代码、维护反馈、追踪问题、接受更新和bug修复的补丁等，都变得简单了许多。近年来，GitHub越来越面向“无开发者”（http://bit.ly/1a1o2OZ）——成为最炙手可热的主流协作社交平台之一。

为了更加清楚的说明，本章并不提供教你如何使用Git或者GitHub这个分布式版本控制系统的教程，也不会讨论任何层面的Git软件架构。（可以查看许多出色的Git在线参考资料，比如gitscm.com（http://bit.ly/1a1o2hZ）来获取这类内容。）本章尝试教你如何探索GitHub的API来挖掘软件开发的社交协作模式。

注意：在http://bit.ly/MiningTheSocialWeb2E上可以找到本章（及所有其他章节）最新修订bug的源代码。同时也要利用好本书的虚拟机，如附录A中描述的，来尽可能的享用样例代码。

7.1 概述

本章提供了对GitHub这个社交编程平台的介绍以及使用NetworkX对图的分析。在本章中，你会学习到如何利用GitHub丰富的数据，可以通过建立数据的图模型，这在很多方面都很有用。特别地，我们会将GitHub用户、资源库和编程语言的关系视为兴趣图谱（interest graph，http://bit.ly/1a1o3Cu），这是一种表达节点和图中的关联的方法，主要来自于人以及人感兴趣的事物。这些年，黑客、企业家、网络专家对Web的未来是否基于兴趣图谱的某些概念的讨论有很多，所以现在正是加入图谱的快车，了解和学习图谱知识的好时机。

总的来说，本章和前面章节的模板相同，将会涵盖以下内容：

·GitHub开发者平台和如何发起API请求。

·图模式和如何使用NetworkX来对属性图建模。

·兴趣图谱的概念和如何用GitHub数据建立兴趣图谱。

·使用NetworkX来请求属性图。

·图中心度算法，包括点度中心度（degree centrality）、中介中心度（betweenness centrality）和接近中心度（closeness centrality）。

7.2 探索GitHub的API

正如本书其他社交网站的属性特征一样，GitHub的开发者网站（http://bit.ly/1a1o49k）提供了它API的丰富的文档，说明了使用这些API、示例代码等的服务条件。尽管API非常丰富，我们只会关注一小部分API调用，这是我们为建立兴趣图谱采集数据所需要的，同软件开发者、项目、编程语言还有软件开发的其他方面紧密相连。API或多或少的提供了你在建立丰富的用户体验（比如github.com本身）时需要的东西，并且你能用这些API开发很多优秀且能盈利的应用。

GitHub最基本的是用户和项目。在阅读这一页时，你可能已经拉取过本书GitHub项目页中的源代码（http://bit.ly/1a1kNqy），所以我们假设你至少已经访问过一部分GitHub项目页并到处浏览过，你已经对GitHub提供的东西有了基本的了解。

一个GitHub用户会有公开的资料，通常包括一个或多个资源库，可以是自己建立的，也可以是派生别的用户的。比如，GitHub用户ptwobrussell（http://bit.ly/1a1o4GC）有很多GitHub资源库，一个叫做Mining-the-Social-Web（http://bit.ly/1a1o6Ow），还有一个叫做Mining-the-Social-Web-2nd-Edition（http://bit.ly/1a1kNqy）。ptwobrussell也为了开发目的派生了一些资源库来获取特定代码的快照，这些被派生的项目

也在他的公开资料中显示出来了。

GitHub中很强大的地方是ptwobrussell可以免费地对那些派生的项目进行随心所欲的改造（遵循那些软件的许可），其他任何人也可以对派生的项目做同样的事情。当用户派生了一个代码资源库，他可以有效地拥有同样的工作副本并在该副本上加以改造，可以彻底的查看所有代码并建立长久的与原始项目的派生关系，甚至可以永远不和原始父资源库合并。尽管大多数派生来的项目从未进一步加入派生者自己的工作，但对于源代码管理来说，建立派生需要的工作量是微不足道的。派生的项目或者是短暂的通过拉取请求合并到父代码，或者变成长期的独立项目，已经和自己的社区完全脱离了。在GitHub上，阻碍你为开源软件做贡献的障碍是非常小的。

除了在GitHub上派生项目，用户还可以收藏（bookmark）项目或给项目加星（star），并成为这个项目的观星人（stargazer）。收藏项目和收藏网页或者推文一样重要。你对这个项目很感兴趣，这个项目会出现在你的GitHub收藏列表中来快速访问。你可能会注意到的是，收藏项目的人比派生项目的人要多得多。在超过十年的网络冲浪时代，收藏是一个简单又容易理解的概念，尽管派生代码代表你可能想修改或在某种程度上对代码做贡献。在本章剩余的章节中，我们主要关注使用项目的观星人列表作为建立兴趣图谱的基础。

7.2.1 建立GitHub API连接

和其他社交网络的属性一样，GitHub实现了OAuth，获得API访问的步骤包括建立账户，使用下面两种方式：建立一个应用作为API的用户，或者建立“个人的”访问令牌，这会直接关联到你的账户。在本章中，我们选择使用个人的访问令牌，这和在你的账户的应用程序（http://bit.ly/1a1o7lw）菜单中的个人访问API令牌项中单击一个按钮一样简单，如图7-1所示。（参见附录B来获取更多的OAuth知识。）

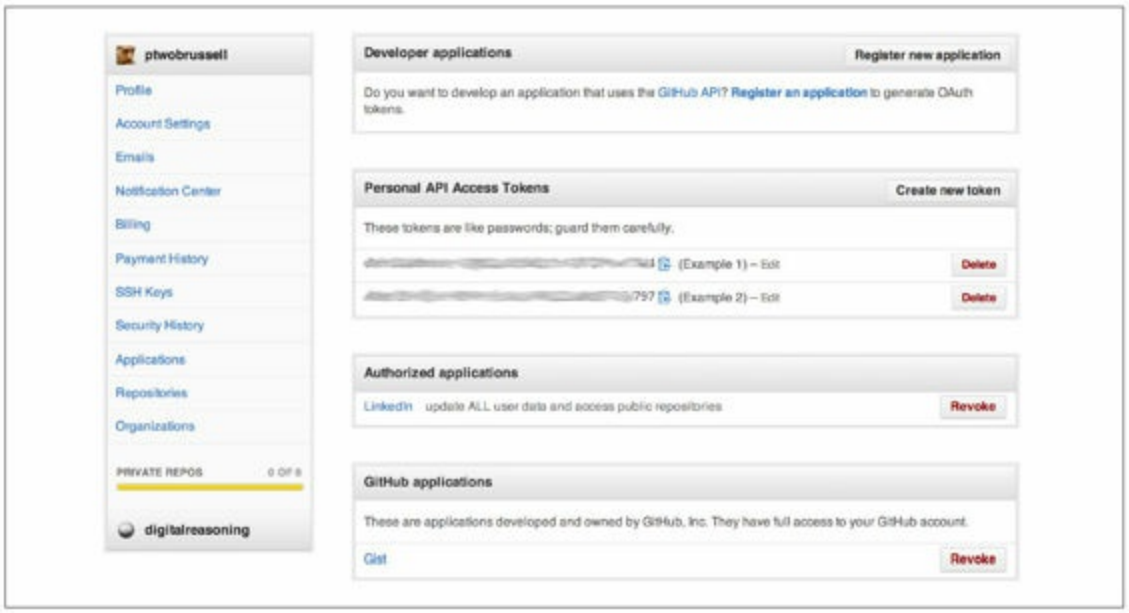


图7-1：在你账户的应用程序菜单中创建“个人访问API令牌”（Personal API Access Token）并使用有意义的注释，这样你就能记住它的目的

在程序中获得访问令牌而不是使用GitHub的用户界面来创建的方法在示例7-1中有所展现，是GitHub的帮助页中“创建一个OAuth令牌供命

令行使用”（Creating an OAuth token for command-line use）（http://bit.ly/1a1o7lG）的改编。（如果你没有好好利用本书附录A中的虚拟机知识，你将需要在运行这个例子之前，在终端输入pip install requests命令。）

示例7-1：在程序中获得访问GitHub API的个人API访问令牌

```
   import requests
from getpass import getpass
import json
username= '' # Your GitHub username
password= '' # Your GitHub password
# Note that credentials will be transmitted over a secure SSL connection
url= 'https://api.github.com/authorizations'
note= 'Mining the Social Web, 2nd Ed.'
post_data = {'scopes':['repo'],'note': note }
response= requests.post(
    url,
    auth= (username, password),
    data= json.dumps(post_data),
    )
print"API response:", response.text
print
print"Your OAuth token is", response.json()['token']
# Go to https://github.com/settings/applications to revoke this token
```

和许多其他的社交网络属性一样，GitHub API是建立在HTTP协议之上的，并且可以使用任何能够发出HTTP请求的编程语言来访问，包括终端的命令行工具。遵循前面章节的先例，我们选择利用Python库，这样就能避免一些繁琐的工作，比如构造请求、解析返回结果、处理页码标注等。在这个例子中，我们将使用PyGithub（http://bit.ly/1a1o7Ca），可以毫无悬念的通过pip install PyGithub命令来安装。我们将以一些如何构造GitHub API请求的例子开始，然后再过渡到图模型。

让我们以Mining-the-Social-Web（http://bit.ly/1a1o6Ow）GitHub资源库的兴趣图谱为切入点，建立它和其他观星人的联系。使用显示观星人API（http://bit.ly/1a1o9dd）可以获得资源库的观星人列表。你可以尝试复制并粘贴下面的URL到你的浏览器中，通过这个API请求来了解返回类型是什么样的：https://api.github.com/repos/ptwobrussell/Mining-the-Social-Web/stargazers。

注意：尽管你正在阅读本书第2版，但在写这部分的时候，第1版的源代码资源库仍然比第2版更丰富，所以第1版的源代码资源库将作为本章示例的基础。分析任何的源代码资源库，包括本书第2版的源代码资源库，都很容易完成，只需要像示例7-3中介绍的那样，将原始工程的名字进行更改即可。

用这种方式发起一个未授权的请求正如你在API中看到的那样非常方便，每个小时中，60个未授权请求的数量限制对于探索来说足够了。你可以追加一个表单的查询字符串？access_token=xxx，其中xxx是你的access token，这样这个请求就是授权的方式了。GitHub的授权数量限制是每小时5000个请求，正如访问速率上限限制（rate limiting）的开发者文档（http://bit.ly/1a1oblo）中描述的那样。示例7-2展示了一个请求和响应的例子。（记住，这只是请求了结果的第一页，正如分页的开发者文档（developer documentation for pagination）（http://bit.ly/1a1o9Ki）中描述的那样，导航结果页的元数据信息被包含在HTTP头中。）

示例7-2：对GitHub的API建立直接的HTTP请求

```
   import json
import requests
# An unauthenticated request that doesn't contain an ?access_token=xxx query string
url= "https://api.github.com/repos/ptwobrussell/Mining-the-Social-Web/stargazers"
response= requests.get(url)
# Display one stargazer
printjson.dumps(response.json()[0], indent=1)
print
# Display headers
for(k,v) in response.headers.items():
    printk, "=>", v
```

示例的输出如下：

```
   {
 "following_url": "https://api.github.com/users/rdempsey/following{/other_user}",
 "events_url": "https://api.github.com/users/rdempsey/events{/privacy}",
 "organizations_url": "https://api.github.com/users/rdempsey/orgs",
 "url": "https://api.github.com/users/rdempsey",
 "gists_url": "https://api.github.com/users/rdempsey/gists{/gist_id}",
 "html_url": "https://github.com/rdempsey",
 "subscriptions_url": "https://api.github.com/users/rdempsey/subscriptions",
 "avatar_url": "https://1.gravatar.com/avatar/8234a5ea3e56fca09c5549ee...png",
 "repos_url": "https://api.github.com/users/rdempsey/repos",
 "received_events_url": "https://api.github.com/users/rdempsey/received_events",
 "gravatar_id": "8234a5ea3e56fca09c5549ee5e23e3e1",
 "starred_url": "https://api.github.com/users/rdempsey/starred{/owner}{/repo}",
 "login": "rdempsey",
 "type": "User",
 "id": 224,
 "followers_url": "https://api.github.com/users/rdempsey/followers"
}
status => 200 OK
access-control-allow-credentials => true
x-ratelimit-remaining => 58
x-github-media-type => github.beta
x-content-type-options => nosniff
access-control-expose-headers => ETag, Link, X-RateLimit-Limit,
                                 X-RateLimit-Remaining, X-RateLimit-Reset,
                                 X-OAuth-Scopes, X-Accepted-OAuth-Scopes
transfer-encoding => chunked
x-github-request-id => 73f42421-ea0d-448c-9c90-c2d79c5b1fed
content-encoding => gzip
vary => Accept, Accept-Encoding
server => GitHub.com
last-modified => Sun, 08 Sep 2013 17:01:27 GMT
x-ratelimit-limit => 60
link =><https://api.github.com/repositories/1040700/stargazers?page=2>;
            rel="next",
            <https://api.github.com/repositories/1040700/stargazers?page=30>;
            rel="last"
```

```
etag => "ca10cd4edc1a44e91f7b28d3fdb05b10"
cache-control => public, max-age=60, s-maxage=60
date => Sun, 08 Sep 2013 19:05:32 GMT
access-control-allow-origin => *
content-type => application/json; charset=utf-8
x-ratelimit-reset => 1378670725
```

正如你看到的那样，很多GitHub返回给我们的有用的信息并不是在HTTP返回的正文中，而是像开发者文档中列出的那样，是在HTTP头中显示的。你应该了解不同的头表达的意思是什么，不过很少一部分包含状态头，这能告诉我们请求返回的是200状态码，即OK状态；包含数量限制的头，比如x-ratelimit-remaining；还有链接头，这包含了像下面这样的值：

```
   https://api.github.com/repositories/1040700/stargazers?page=2; rel="next",
https://api.github.com/repositories/1040700/stargazers?page=29; rel="last".
```

链接头的值给了我们改造的URL，可以用来获得下一页的结果和结果的总页数。

7.2.2 建立GitHub API请求

尽管使用像requests这样的库并通过解析建立请求都不复杂，像PyGithub这样的库会使处理GitHub API的实现细节变得简单，使我们能使用纯净的Python的API。更好的是，如果GitHub变更了API的实现，我们仍然可以使用PyGithub并且我们的代码不会有任何问题。

在使用PyGithub建立请求之前，应该花些时间查看响应的内容。它包含了丰富的信息，而我们最感兴趣的域是login，这是GitHub中正在注视（stargaze）自己感兴趣库的用户的用户名。这个信息是发起许多其他对GitHub API的请求的基础，正如“被加星的库列表”（List repositories being starred）（http://bit.ly/1a1oc8X），是一个返回用户加星过的所有的库的列表的API。这是很关键的，因为在我们给任意一个库加星并请求获得感兴趣的用户列表之后，我们就可以查询那些用户来获得额外的感兴趣的库，并很可能有新发现。

比如，了解所有收藏了Mining-the-Social-Web的用户中被最多收藏的库不是很有趣吗？问题的答案或许是GitHub用户希望的智能推荐的基础，想象不同的智能推荐的领域也不困难，这会（并且经常会）提高用户体验，像Amazon和Netflix一样。作为核心，兴趣图谱本身就能做智能推荐，这也是近年来兴趣图谱成为关注焦点的原因之一。

示例7-3提供使用PyGithub来获取某个库的所有观星人的例子，并为兴趣图谱打下基础。

示例7-3：使用PyGithub来请求特定库的观星人

```
   from github import Github
# XXX: Specify your own access token here
ACCESS_TOKEN = ''
# Specify a username and repository of interest for that user.
USER = 'ptwobrussell'
REPO = 'Mining-the-Social-Web'
client= Github(ACCESS_TOKEN, per_page=100)
user= client.get_user(USER)
repo= user.get_repo(REPO)
```

```
# Get a list of people who have bookmarked the repo.
# Since you'll get a lazy iterator back, you have to traverse
# it if you want to get the total number of stargazers.
stargazers= [ s for s in repo.get_stargazers() ]
print"Number of stargazers", len(stargazers)
```

在后台，PyGithub照顾到了API的实现细节，并且为查询操作提供了方便的对象。在这种情况下，我们同GitHub建立连接，使用per_page关键字参数来说明我们想收到的结果的最大值（100）而不是每页返回的数据默认值（30）。然后，我们获得特定用户的资源库并查询该库的观星人。可能用户的资源库会拥有相同的名称，所以通过名称来查询并不是一个好方法。因为用户名和资源库名可能重复，在使用GitHub的API时，如果你使用名称作为标识的话，你需要格外注意你所使用的对象的类型。在建立图时我们也会考虑到这个问题，因为节点的名称也需要指明是用户名还是资源库名。

最后，PyGithub通常提供“延迟迭代”（lazy iterator）作为结果，这里说明了当请求发起后，它并没有立即尝试获取所有的29页结果。相反，当迭代检索该页之前的数据时，它会一直等待，直到这个特定页发起请求。因此，如果想要借助API获得观星人的真实数目，我们需要使用列表解析来充分利用延迟迭代进行计算。

PyGithub的文档（http://bit.ly/1a1odd3）非常有用，它的API模仿了GitHub的API，你会经常使用它的pydoc，比如通过Python解释器中的dir（）和help（）函数。另外，IPython或IPython Notebook中的制表补全和“问号”魔法功能跟你在搞清楚哪些方法在调用哪些对象时是可用的

情况是一样的。花些时间在GitHub API和PyGithub上是很值得的，会让你在深入研究前熟悉更多的功能。为了测试你掌握的技能，你能迭代Mining-the-Social-Web的观星人（或一些子集）并做一些基础的频率分析来找到哪些其他的资源库有着相似的兴趣吗？你可能会找到Python的collections.Counter或NLTK的nltk.FreqDist来方便的计算频率数据。

7.3 使用属性图为数据建模

你可能想到了2.3.2节第2条中介绍的图，用来展示、分析并可视化Facebook的社交网络数据。这一节提供了更加全面的讨论，并对图计算来说是很有用的指南。尽管不易察觉，但是图对于真实世界很多现象的建模是一种很自然的抽象，所以图计算发展非常迅猛。图在数据表示上非常灵活，和其他方法（比如关系型数据库）相比，在数据实验和分析上有很大的优势。以图为中心的分析当然不是万能的，但学习如何使用图结构来对数据建模是很有用的。

注意：对图论的介绍超出了本章的范围，这里的讨论只是对一些关键概念的简单介绍。如果在继续之前想积累一些背景知识，你可以通过观看一个短小的YouTube视频“图论简介”（Graph Theory—An Introduction）（http://bit.ly/1a1odto）来了解。

本节剩下的内容主要介绍属性图这种类型，目的是使用Python包NetworkX（http://bit.ly/1a1ocFV）对GitHub数据建模出兴趣图谱。属性图是一种数据结构，通过节点表示实体，通过边表示实体间的关系。每个顶点都有唯一标识，属性间的映射定义为键值对和边的集合。同样的，连接节点的边也是可以唯一识别的，并且包含属性。

图7-2展示了平凡属性图的例子，有两个唯一标识为X和Y的节点，

之间有未描述出来的关系。这种图被称作有向图，因为它的边是有方向的。除非边的方向性根植于所建模的领域，上述规则才有破例。

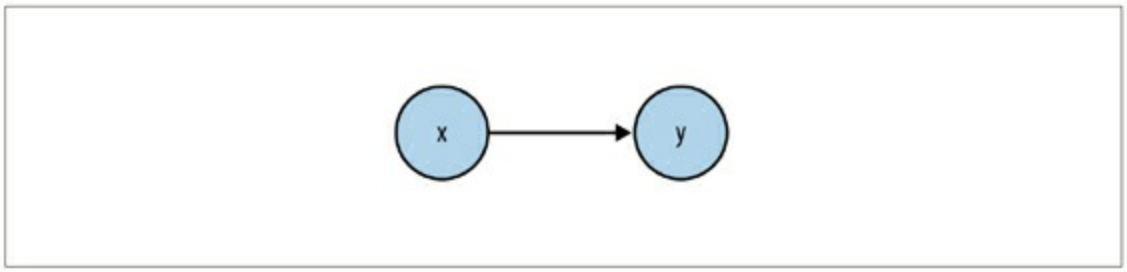


图7-2：拥有有向边的平凡属性图

NetworkX中的代码表达的属性图可以像示例7-4中那么构造。（如果你没有运用本书的虚拟机知识，你可以使用pip install networkx命令来安装这个包。）

示例7-4：构造平凡属性图

```
   import networkx as nx
# Create a directed graph
g = nx.DiGraph()
# Add an edge to the directed graph from X to Y
g.add_edge('X', 'Y')
# Print some statistics about the graph
print nx.info(g)
print
# Get the nodes and edges from the graph
print "Nodes:", g.nodes()
print "Edges:", g.edges()
print
# Get node properties
print "X props:", g.node['X']
print "Y props:", g.node['Y']
# Get edge properties
print "X=>Y props:", g['X']['Y']
print
# Update a node property
g.node['X'].update({'prop1' : 'value1'})
print "X props:", g.node['X']
print
# Update an edge property
g['X']['Y'].update({'label' : 'label1'})
```

```
print "X=>Y props:", g['X']['Y']
```

本例的样本结果如下：

```
   Name:
Type: DiGraph
Number of nodes: 2
Number of edges: 1
Average in degree: 0.5000
Average out degree: 0.5000
Nodes: ['Y', 'X']
Edges: [('X', 'Y')]
X props: {}
Y props: {}
X=>Y props: {}
X props: {'prop1': 'value1'}
X=>Y props: {'label': 'label1'}
```

在这个特定的例子中，有向图的add_edge方法会从X标识的节点到Y标识的节点间添加一条边，形成一个有两个节点一条边的图。这些唯一标识可以通过元祖（X，Y）来表示，因为它连接的每个节点都是唯一的。注意，添加一条从Y到X的边算是建立了图中的第二条边，并且这第二条边会包含自己的边属性的集合。通常，你不会想添加这第二条边，因为你可以获得一个节点的入边或出边，并有效地在任一方向上遍历边，不过有时包含这条额外的边会更加方便。

图中一个节点的度是与该节点连接的边的个数，而对于有向图来说，由于边是有方向的，所以注意区别入度和出度。平均入度和平均出度的值为图提供了常模分数，代表了有入边和出边的节点的数量。在上述情况下，这个有向图只有一条有向边，所以有一个节点拥有一条出边，另一个节点拥有一条入边。

节点的入度和出度是图论的基础概念。假设你知道图中的顶点个数，平均度可以度量图的密度：实际边的个数比上全连通图的边的个数。在全连通图中，每个节点都和其他节点相连接，如果是有向图的话，代表每个节点都有来自所有其他节点的入边。

通过对每个节点的入度求和，再除以图中节点的总数（示例7-4中是1除以2），就能够计算出图的平均入度。平均出度的计算方法也是这样，只不过是对每个节点的出度求和。有向图中，入边和出边的个数总是相同的，因为每条边只连接两个节点[1]，并且平均入度和平均出度也是相同的。

注意：通常情况下，一个图中的平均入度和出度的最大值会比节点的个数少1。想想如何证明它，可以通过考虑一个全连通图中边的个数来入手。

下一节中，我们会使用同样的属性图原语来构造兴趣图谱，并且会展示处理真实数据所需要的方法。首先，花些时间向图中添加节点、边以及属性。如果你是第一次接触图并且想获得额外的指南，NetworkX文档（http://bit.ly/1a1ocFV）提供了有用的介绍性示例。

大型图数据库的崛起

本章介绍了属性图，是一种能够用于建模有节点和边的复杂网络的数据结构。我们会基于直觉根据灵活的图模式来对数据建模，对于某个

狭小的专一领域，这种程序化的方法通常已经足够了。在本章剩余部分我们会看到，属性图对建模并查询复杂数据的灵活性和多样性都是很好的。

NetworkX，这种本书一直使用的基于Python的图工具，提供了强大的工具箱来对属性图建模。请注意，NetworkX是一种内存中的图数据库。对你做的工作的限制是和你机器上运行时消耗的内存多少成比例的。在很多情况下，你可以通过限制领域来使用数据的子集，或者使用运行内存更大的机器来避免这些限制。现在“大数据”的情况越来越多，它的新兴生态系统包括Hadoop和NoSQL数据库。不管怎样，内存中的图并不能简单地说就是一种解决方案。

还有一种新兴的生态系统叫做“大型图数据库”（big graph database），推动了NoSQL数据库的存储并提供了属性图的语义，这是很值得注意的。Titan（http://bit.ly/1a1og8D）是一个很有前途的领跑者，其他采用属性图模型的大型图数据库展示出脱离语义网栈（semantic web stack）（http://bit.ly/1a1oegY）的端倪，包括RDF Schema（http://bit.ly/1a1ogpe）、OWL（http://bit.ly/1a1ogFz）和SPARQL（http://bit.ly/1a1ogW5）等技术。这些技术背后有关于语义网栈的想法，提供了表示数据和从复杂领域中整合数据的机制，因此为标准化能够被有意义查询的词汇表提供了可能。遗憾的是，在网络规模上应用该技术有很多历史上的挑战。Titan的关键之一就是它考虑到了规

模，因此它是为了有效的管理分级存储器而设计的。

看看将来基于NoSQL数据库的大型图数据库和属性图模型是很令人兴奋的，其中属性图模型会融合更加传统的语义网工具链的思想和技术。下一章介绍了当前网络创新中的微格式，这是迈向更加具有语义特性的网络的一个步骤；结尾是一个小例子，基于类似语义网栈的技术小子集，对简单图的推断予以展示。

[1] 更抽象些的图叫做超图（hypergraph）（http://bit.ly/1a1ocWm），它包含可以连接任意数量的顶点的超边（hyperedges）。

7.4 分析GitHub兴趣图谱

现在有了查询GitHub的API和对数据建模为图的技能，我们可以开始测试并创建和分析兴趣图谱了。我们将从能够代表一组GitHub用户的共同兴趣的库开始，使用GitHub的API来找到该库的观星人。从这开始，我们会使用其他的API来对GitHub用户的社交关系进行建模。

我们也会学习一些分析图的基本技术，叫做中心度度量（centrality measure）。尽管一个图的可视化界面非常有用，但许多图过大或者过于复杂，以至于无法进行有效的可视化查看，这样中心度度量在网络结构的分析性度量方面很有帮助。（但不用担心，在本章结束之前，我们仍然会对图进行可视化。）

7.4.1 初始化一个兴趣图谱

回想一下，兴趣图谱和社交图并不是一回事。尽管社交图谱（social graph，http://bit.ly/1a1ofl4）的主要焦点是展现人与人间的联系，它通常需要参与的当事人的相互关系。一个兴趣图谱连接着人和兴趣，使用单向的边来连接。虽然它们两个并不是完全脱离的概念，但不要混淆GitHub用户关注另一个GitHub用户的社交联系——它是“对XX感兴趣”的联系，因为并没有相互接受的标准在里面。

注意：Facebook是一个可以称为社交兴趣图谱的杂交图经典的例子。它主要从基于社交图谱概念的技术平台开始，但赞按钮的加入非常混搭，使之可以被称为社交兴趣图谱。它清楚的展示了人与人之间的关系，以及人与他们感兴趣的事物之间的关系。Twitter也总会有些兴趣图谱的味道，它有不对称的“关注”模型，这可以被解释为人和他们感兴趣的事物（可以是其他人）的关系。

示例7-5和示例7-6介绍了构建初始的用户和资源库的“注视”关系的代码示例，展示了如何研究出现的图结构。初始建立的图可以被称为“自我图”（ego graph），因为它有一个中心点（自我），这是大多数（在这种情况下是全部）边的基础。自我图有时被称作“轴辐式图”（hub and spoke graph）或者“星图”（star graph），因为它很像一个有散发的轮辐的轮轴，也很像视觉中呈现的星星。

从图模式的立场上来看，正如图7-3展示的那样，图包含了两种节点和一种边。

注意：我们会从图7-3中的图模式开始，并在本章的剩余部分中对它进行修改。

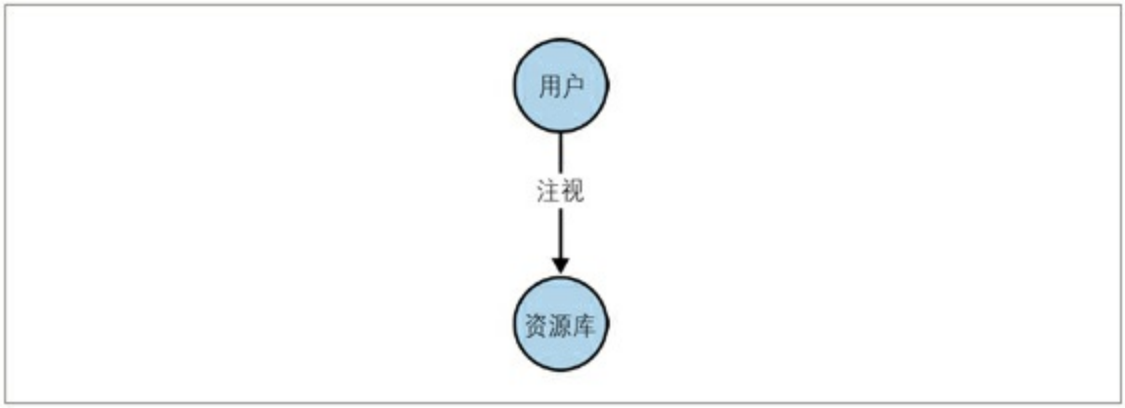


图7-3：包含对资源库感兴趣的GitHub用户的图模式的基础

在数据建模中，有一个细微却重要的限制，那就是要避免命名冲突：用户名和资源库名会（并且经常会）相互冲突。比如，也许有一个GitHub用户名为“ptwobrussell”，还可能有许多资源库的名称也是“ptwobrussell”。回想一下，add_edge方法使用传入的前两个参数作为唯一标识，我们可以添加“（user）”或者“（repo）”来保证所有图中的节点都是唯一的。对于使用NetworkX建模来说，为节点添加类型说明的方法通常可以解决这个问题。

同样，不同用户的资源库也许有相同的名字，不管它们是派生自相同的还是完全不同的代码基本库。不过目前我们还不必关心这个细节，但一旦我们开始添加其他GitHub用户stargaze的资源库时，这种命名冲突的概率就会提高。

允许这种冲突或者实现一种避免该冲突的构建图的策略是设计时要考虑的，并且要考虑到特定的结果。比如，不同人派生的同一个资源库

在图中使用一个节点来表示更好，而不是表现为不同的资源库，但你肯定也不想让名称相同的不同项目使用同一个节点。

注意：鉴于该问题范围是有限的，并且它本来是针对于一个特定的资源库的，所以我们选择避开命名二义性引入的复杂性问题。

有了这些，我们可以看一下示例7-5，它建立了一个资源库和观星人的自我图。示例7-6介绍了一些很有用的对图的操作。

示例7-5：建立一个资源库和观星人的自我图

```
   # Expand the initial graph with (interest) edges pointing each direction for
# additional people interested. Take care to ensure that user and repo nodes
# do not collide by appending their type.
import networkx as nx
g = nx.DiGraph()
g.add_node(repo.name + '(repo)', type='repo', lang=repo.language, owner=user.login)
for sg in stargazers:
    g.add_node(sg.login + '(user)', type='user')
    g.add_edge(sg.login + '(user)', repo.name + '(repo)', type='gazes')
```

示例7-6：介绍一些方便的图操作

```
   # Poke around in the current graph to get a better feel for how NetworkX works
print nx.info(g)
print
print g.node['Mining-the-Social-Web(repo)']
print g.node['ptwobrussell(user)']
print
print g['ptwobrussell(user)']['Mining-the-Social-Web(repo)']
# The next line would throw a KeyError since no such edge exists:
# print g['Mining-the-Social-Web(repo)']['ptwobrussell(user)']
print
print g['ptwobrussell(user)']
print g['Mining-the-Social-Web(repo)']
print
print g.in_edges(['ptwobrussell(user)'])
print g.out_edges(['ptwobrussell(user)'])
print
print g.in_edges(['Mining-the-Social-Web(repo)'])
print g.out_edges(['Mining-the-Social-Web(repo)'])
```

下面的示例（缩减版）输出展示了基于图操作的一些可用的方法：

```
   Name:
Type: DiGraph
Number of nodes: 852
Number of edges: 851
Average in degree: 0.9988
Average out degree: 0.9988
{'lang': u'JavaScript', 'owner': u'ptwobrussell', 'type': 'repo'}
{'type': 'user'}
{'type': 'gazes'}
{u'Mining-the-Social-Web(repo)': {'type': 'gazes'}}
{}
[]
[('ptwobrussell(user)', u'Mining-the-Social-Web(repo)')]
[(u'gregmoreno(user)', 'Mining-the-Social-Web(repo)'),
 (u'SathishRaju(user)', 'Mining-the-Social-Web(repo)'),
  ...
]
[]
```

有了初始的兴趣图谱，我们可以富有创意的决定下一步怎样做才可能是最有趣的。我们目前了解的是，在社交网络挖掘中，大约有850个用户拥有共同的兴趣，正如ptwobrussell的Mining-the-Social-Web项目的加星关系展示的一样。图中边的个数果然比节点的个数少1。原因是观星人和资源库是一一对应的（每个观星人和资源库之间必须有一条边连接）。

如果你还记得平均入度和平均出度产生的标准值提供了对图的密度的度量，0.9988的值应该能证实我们的直觉。我们有851个代表观星人的节点，每个节点的出度为1，还有一个节点代表资源库，它的入度为851。换句话说，图中边的个数比节点的个数少1。图中边的密度非常低，因为这个例子中的平均度的最大值是851。

考虑图的拓扑结构很有用，由于知道了它看起来像是星星，可以尝试建立与0.9988这个值之间的联系。我们的图中确实是有一个节点和其他所有节点都有联系，但基于这个单节点，尝试与大约为1的平均度建立联系是不对的。这和将这851个节点可以使用别的配置来达到0.9988的数值的方法一样简单。为了支持这个结论，我们还需要考虑额外的一些分析，比如下一节将要介绍的中心度度量。

7.4.2 计算图的中心度度量

中心度度量是图分析的基础，它提供了对图中一个特定节点的相对重要性的分析。考虑下面的中心度度量，能够帮助我们更仔细的检查图以便对网络有更好的理解：

点度中心度

在图中，节点的点度中心度是一个节点的边数。可以将中心度度量看做一种为节点上的边的频率制表的方式，目的是提供统一度量、找到有最多或最少的关联边（incident edge）的节点或者基于连接个数找到网络拓扑本质的模式。节点的中心度对分析它在网络中的作用很有用，不过这只是一个方面，它还可以识别和图中其他节点的联系中的极值和异常，这是一个很好的开端。我们还能从先前的结论中得到，平均中心度会告诉我们整个图的密度。NetworkX提供了networkx.degree_centrality

方法，它是计算图的点度中心度的内置函数。它会返回一个将每个节点的ID映射到它的点度中心度的字典。

中介中心度

节点的中介中心度是节点连接其他图中节点的中心度度量。你或许将中介中心度理解为一个节点在作为中介或途径来连接其他节点的关键程度。尽管不经常发生，移除中介中心度高的节点会干扰图中能量[1]的流动。有些情况下，移除高中介中心度的节点可以把原图分解为子图。NetworkX提供了networkx.betweenness_centrality方法，它是计算图的中介中心度的内置函数。它会返回一个将每个节点的ID映射到它的中介中心度的字典。

接近中心度

接近中心度是一个节点与图中所有其他节点的联系紧密程度的度量。这种中心度度量可以预测图中的最短路径和一个节点与另一个特定节点的紧密程度。接近中心度并不像节点的中介中心度那样，能够表达作为中介或途径与其他节点联系的完整程度，它通常在有向的联系中考虑。可以把接近度想象为节点将能量分散给其他节点的能力。NetworkX提供了networkx.closeness_centrality方法，它是计算图的接近中心度的内置函数。它会返回一个将每个节点的ID映射到它的接近中心度的字典。

注意：NetworkX在它的在线文档中提供了许多强大的中心度度量（http://1.usa.gov/1a1ofBF）。

图7-4展示了Krackhardt风筝图（http://bit.ly/1a1oixa），它是一个社交网络中被广泛研究的图。该图展示了本节介绍的不同的中心度度量。它被称为“风筝图”的原因是它看起来像一个风筝。

示例7-7展示了从NetworkX中加载图并计算它的中心度度量的代码，在表7-1中有所再现。尽管这对计算并没有影响，请注意这个特定的图经常被用作社交网络的参考。正如上面那样，边并不是有向的，因为社交网络中的联系是相互的。在NetworkX中，它是networkx.Graph的实例，而不是networkx.DiGraph的。

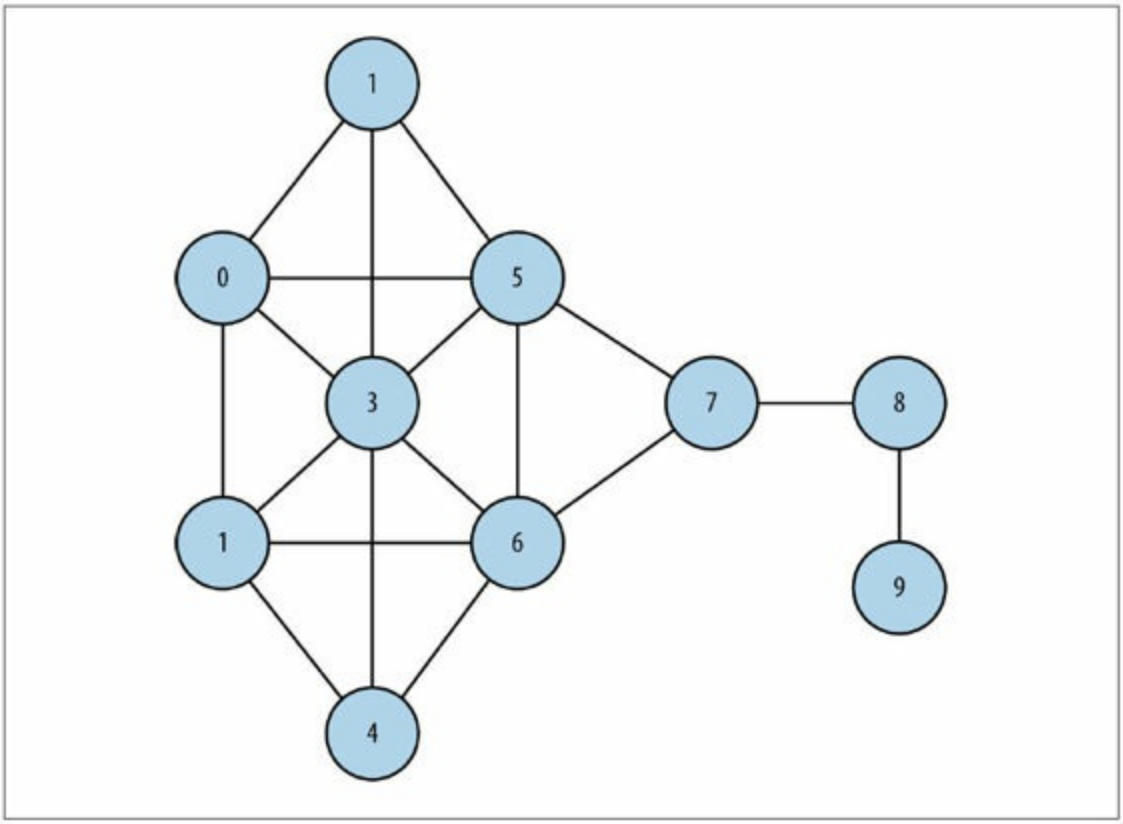


图7-4：Krackhardt风筝图用来展示点度中心度、中介中心度和接近中心度度量

示例7-7：在Krackhardt风筝图中计算点度中心度、中介中心度和接近中心度度量

```
   from operator import itemgetter
from IPython.display import HTML
from IPython.core.display import display
display(HTML('<img src="files/resources/ch07-github/kite-graph.png" width="400px">'))
# The classic Krackhardt kite graph
kkg = nx.generators.small.krackhardt_kite_graph()
print "Degree Centrality"
print sorted(nx.degree_centrality(kkg).items(),
            key=itemgetter(1), reverse=True)
print
print "Betweenness Centrality"
print sorted(nx.betweenness_centrality(kkg).items(),
             key=itemgetter(1), reverse=True)
print
print "Closeness Centrality"
print sorted(nx.closeness_centrality(kkg).items(),
```

```
            key=itemgetter(1), reverse=True)
```

表7-1：Krackhardt风筝图的点度中心度、中介中心度和接近中心度度量（每一列的最大值都用黑体标出了，这样能更方便地和图7-4结合来看）

节点	点度中心度	中介中心度	接近中心度
0	0.44	0.02	0.53
1	0.44	0.02	0.53
2	0.33	0.00	0.50
3	**0.67**	0.10	0.60
4	0.33	0	0.50
5	0.55	0.2	**0.64**
6	0.55	0.2	**0.64**
7	0.33	**0.39**	0.60
8	0.22	0.22	0.43
9	0.11	0.00	0.31

在学习下一节之前，不要忘了花些时间学习Krackhardt的风筝图和它的中心度度量，在本章的剩余内容中，中心度度量一直是很有用的工具。

7.4.3 为用户添加“关注”边来扩展兴趣图谱

除了加星和派生资源库，GitHub还有像Twitter那样的“关注”其他用户的功能。在本节中，我们会使用GitHub的API向图中添加“关注”关系。基于先前关于Twitter兴趣图谱的讨论（比如1.2节中），添加这种关系是捕捉更多的兴趣关系的基本方式，因为“关注”关系和途中的“感兴趣”本质上是一样的。

资源库的拥有者很可能在社区中很受欢迎，因为有很多人加星他的资源库，那还有哪些人很受欢迎呢？这个问题的答案非常重要，并且为未来的分析提供了有用的基础。我们可以通过使用GitHub的用户粉丝API（http://bit.ly/1a1oixo）来查询图中每个用户的粉丝，并添加边来展示关注关系。对于我们的图模型来说，这只是向图中增加了新的边；而不需要添加新的节点。

尽管向图中添加所有的关注关系是可行的，但现在我们将分析对象局限在对初始图中的资源库有明确兴趣的用户。示例7-8展示了向图中添加关注边的示例代码，图7-5描绘了包含关注关系的更新的图。

注意：GitHub的授权部分限制每小时最多请求5000次，这样如果要超标每分钟需要请求80次以上。考虑到每个请求的一些潜在因素，这么高的频率并不太可能发生，所以本章的代码示例中并不包含对频率限制

的处理。

示例7-8：通过“关注”关系向图中添加新的兴趣边

```
   # Add (social) edges from the stargazers' followers. This can take a while
# because of all of the potential API calls to GitHub. The approximate number
# of requests for followers for each iteration of this loop can be calculated as
# math.ceil(sg.get_followers() / 100.0) per the API returning up to 100 items
# at a time.
import sys
for i, sg in enumerate(stargazers):
    # Add "follows" edges between stargazers in the graph if any relationships exist
    try:
        for follower in sg.get_followers():
            if follower.login + '(user)' in g:
                g.add_edge(follower.login + '(user)', sg.login + '(user)',
                           type='follows')
    except Exception, e: #ssl.SSLError
        print >> sys.stderr, "Encountered an error fetching followers for", \
                           sg.login, "Skipping."
        print >> sys.stderr, e
    print "Processed", i+1, " stargazers. Num nodes/edges in graph", \
        g.number_of_nodes(), "/", g.number_of_edges()
    print "Rate limit remaining", client.rate_limiting
```

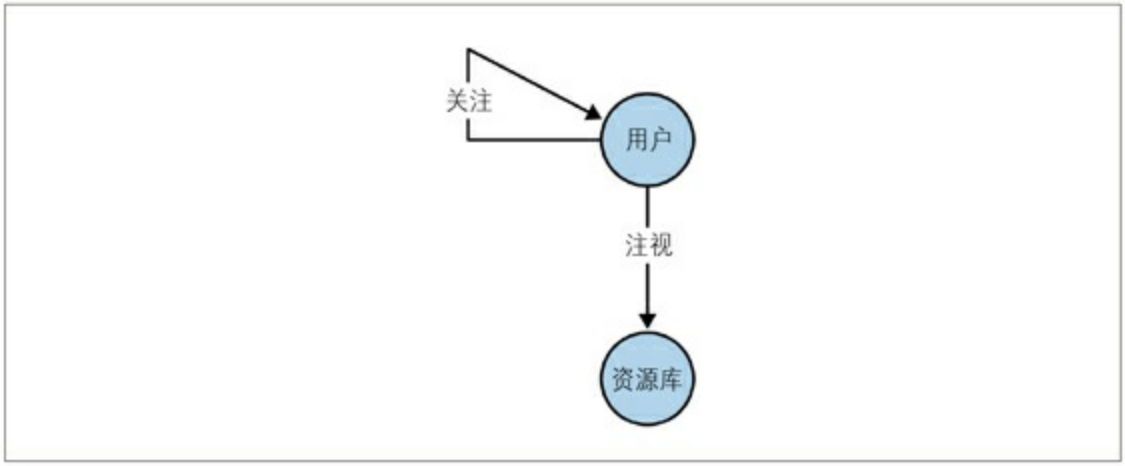


图7-5：包含GitHub中对资源库感兴趣的用户以及其他用户的图模式的示例

图中有了新添加的兴趣数据，分析数据变得更加有趣了。我们可以通过图中的加入“关注”边的个数来计算特定用户的受欢迎程度，如示例

7-9中展示的那样。这种分析最强大的地方是可以让我们找到在特定兴趣领域中最具影响力的用户。

由于我们使用Mining-the-Social-Web资源库来构成图，很可能的假设是对这个话题感兴趣的用户应该对数据挖掘很感兴趣，或者对Python编程很感兴趣，因为我们的代码大部分都是用Python写的。正如示例7-9那样，我们可以探索出最受欢迎的用户是否对Python编程感兴趣。

示例7-9：探索更新了“关注”边的图

```
   from operator import itemgetter
from collections import Counter
# Let's see how many social edges we added since last time.
print nx.info(g)
print
# The number of "follows" edges is the difference
print len([e for e in g.edges_iter(data=True) if e[2]['type'] == 'follows'])
print
# The repository owner is possibly one of the more popular users in this graph.
print len([e
           for e in g.edges_iter(data=True)
               if e[2]['type'] == 'follows' and e[1] == 'ptwobrussell(user)'])
print
# Let's examine the number of adjacent edges to each node
print sorted([n for n in g.degree_iter()], key=itemgetter(1), reverse=True)[:10]
print
# Consider the ratio of incoming and outgoing edges for a couple of users with
# high node degrees...
# A user who follows many but is not followed back by many.
print len(g.out_edges('hcilab(user)'))
print len(g.in_edges('hcilab(user)'))
print
# A user who is followed by many but does not follow back.
print len(g.out_edges('ptwobrussell(user)'))
print len(g.in_edges('ptwobrussell(user)'))
print
c = Counter([e[1] for e in g.edges_iter(data=True) if e[2]['type'] == 'follows'])
popular_users = [ (u, f) for (u, f) in c.most_common() if f >1 ]
print "Number of popular users", len(popular_users)
print "Top 10 popular users:", popular_users[:10]
```

示例的输出如下：

```
   Name:
Type: DiGraph
Number of nodes: 852
Number of edges: 1417
Average in degree: 1.6631
Average out degree: 1.6631
566
89
[(u'Mining-the-Social-Web(repo)', 851),
 (u'hcilab(user)', 121),
 (u'ptwobrussell(user)', 90),
 (u'kennethreitz(user)', 88),
 (u'equus12(user)', 71),
 (u'hammer(user)', 16),
 (u'necolas(user)', 16),
 (u'japerk(user)', 15),
 (u'douglas(user)', 11),
 (u'jianxioy(user)', 11)]
118
3
1
89
Number of popular users 95
Top 10 popular users: [(u'ptwobrussell(user)', 89),
                       (u'kennethreitz(user)', 84),
                       (u'necolas(user)', 15),
                       (u'japerk(user)', 14),
                       (u'hammer(user)', 13),
                       (u'isnowfy(user)', 6),
                       (u'kamzilla(user)', 6),
                       (u'acdha(user)', 6),
                       (u'tswicegood(user)', 6),
                       (u'albertsun(user)', 5)]
```

我们或许可以猜到，资源库的拥有者构建了最原始的兴趣图谱，ptwobrussell（http://bit.ly/1a1o4GC）是图中最受欢迎的用户，不过另一个用户（kennethreitz）和他很接近，有84个关注者，另外还有排名前十位的一些用户拥有数目较多的关注者。且不说别的，kennethreitz（http://bit.ly/1a1ojkT）是流行的Python包requests的作者，这个包在整本书中都有使用。我们也看到了hcilab关注了许多人，但并没有被很多人关注。（一会儿我们会回到这个观察到的问题。）

7.4.3.1 中心度度量的应用

在进行其他的工作之前，我们需要存储一下图的内容，这样我们对目前的状态有稳定的备份，以便将来改进图或者恢复到这个版本，还能对数据进行序列化和分享。示例7-10展示了如何保存并使用NetworkX的内部方法来存储图。

示例7-10：获得图的状态的快照（pickling）并存于磁盘中

```
   # Save your work by serializing out (pickling) the graph
nx.write_gpickle(g, "resources/ch07-github/data/github.gpickle.1")
# How to restore the graph...
# import networkx as nx
# g = nx.read_gpickle("resources/ch07-github/data/github.gpickle.1")
```

把我们的工作存储于磁盘进行备份之后，现在可以将前面章节的中心度度量应用于图中，并获得结果。由于我们知道Mining-the-Social-Web（repo）是图中的超节点，并且连接着大多数用户（该例中是所有用户），所以我们为了获得更好的可视化效果，将要从图中移除这个点。这就只留下了GitHub用户和他们之间的“关注”边。示例7-11展示了在分析起步时的一些代码。

示例7-11：将中心度度量应用于兴趣图谱中

```
   from operator import itemgetter
# Create a copy of the graph so that we can iteratively mutate the copy
# as needed for experimentation
h = g.copy()
# Remove the seed of the interest graph, which is a supernode, in order
# to get a better idea of the network dynamics
h.remove_node('Mining-the-Social-Web(repo)')
# XXX: Remove any other nodes that appear to be supernodes.
# Filter any other nodes that you can by threshold
# criteria or heuristics from inspection.
# Display the centrality measures for the top 10 nodes
dc = sorted(nx.degree_centrality(h).items(),
```

```
            key=itemgetter(1), reverse=True)
print "Degree Centrality"
print dc[:10]
print
bc = sorted(nx.betweenness_centrality(h).items(),
            key=itemgetter(1), reverse=True)
print "Betweenness Centrality"
print bc[:10]
print
print "Closeness Centrality"
cc = sorted(nx.closeness_centrality(h).items(),
             key=itemgetter(1), reverse=True)
print cc[:10]
```

示例的结果如下：

```
    点度中心度
[(u'hcilab(user)', 0.1411764705882353),
 (u'ptwobrussell(user)', 0.10470588235294116),
 (u'kennethreitz(user)', 0.10235294117647058),
 (u'equus12(user)', 0.08235294117647059),
 (u'hammer(user)', 0.01764705882352941),
 (u'necolas(user)', 0.01764705882352941),
 (u'japerk(user)', 0.016470588235294115),
 (u'douglas(user)', 0.011764705882352941),
 (u'jianxioy(user)', 0.011764705882352941),
 (u'mt3(user)', 0.010588235294117647)]
中介中心度
[(u'hcilab(user)', 0.0011790110626111459),
 (u'douglas(user)', 0.0006983995011432135),
 (u'kennethreitz(user)', 0.0005637543592230768),
 (u'frac(user)', 0.00023557126030624264),
 (u'equus12(user)', 0.0001768269145113876),
 (u'acdha(user)', 0.00015935702903069354),
 (u'hammer(user)', 6.654723137782793e-05),
 (u'mt3(user)', 4.988567865308668e-05),
 (u'tswicegood(user)', 4.746068038522 83e-05),
 (u'stonegao(user)', 4.068058318733853e-05)]
接近中心度
[(u'hcilab(user)', 0.14537589026642048),
 (u'equus12(user)', 0.1161965001054185),
 (u'gawbul(user)', 0.08657147291634332),
 (u'douglas(user)', 0.08576408341114222),
 (u'frac(user)', 0.059923888224421004),
 (u'brunojm(user)', 0.05970317408731448),
 (u'empjustine(user)', 0.04591901349775037),
 (u'jianxioy(user)', 0.012592592592592593),
 (u'nellaivijay(user)', 0.012066365007541477),
 (u'mt3(user)', 0.011029411764705881)]
```

在我们前面的分析中，不出所料，用户ptwobrussell和kennethreitz是

点度中心度很高的两个。不过用户hcilab在所有的中心度度量中都位居第一。回顾我们以前的分析，用户hcilab关注了其他许多用户，从该用户的资料（https://github.com/hcilab）能看出这个账号可能是数据挖掘本身的一部分！它被称为“GitHub研究账号”并且近几年只有一天是活动的。基于前面的分析，该用户有资格成为超节点，从图中移除该节点并重新进行中心度度量会改变网络的动态性，也会让分析更加清晰。

另一个观察到的结果是接近中心度和点度中心度比中介中心度要高很多，可以通过0的数量看出来。在“关注”关系中，这说明图中并没有用户可以充当有效连接其他用户的中介。可以这么处理，是因为原始图中的是资源库，提供有共同的兴趣。如果作出与示例相反的假设，真的找到一个有效连接其他用户的用户中介，也未必能得到出其不意的结果。兴趣图谱的基础是特定用户，动态性可能会是不同的。

最后，尽管用户ptwobrussell和kennethreitz都是图中很受欢迎的用户，他们在接近中心度中的排名却排不到前十。反而是一些其他的用户有较高的接近中心度，这是很值得检查的。截止到2013年8月（写作本部分的时间），用户equus12用400个以上的关注者，但只有两个资源库被派生了，并且最近没有公开的活动。用户gawbul拥有44个关注者，许多活动的资源库，个人活动也相当活跃。对拥有较高度中心度和接近中心度的用户进行深入的探索留作独立的练习。记住动态性在不同社区中是不一样的。

注意：对比两个不同社区的网络动态性是很有价值的，比如Ruby on Rails社区和Django社区。你可能也会尝试对比以Microsoft为中心的社区和面向Linux的社区。

7.4.3.2 向兴趣图谱中添加更多的资源库

总而言之，对图中的“关注”边的分析是很有趣的，当我们回顾兴趣图谱的种子时，世界各地吸引不同用户的资源库并不那么令人吃惊。下一步值得做的是通过遍历来努力找到图中每个用户更多的兴趣，然后把它们的加星标的资源库添加到图中。添加这些加星的资源库至少会给我们两个有价值的观点：还有什么别的资源库是涉及这种以社交网络挖掘为基础的社区（在一个较小的程度上，如Python），在这个社区中什么编程语言比较流行，因为GitHub尝试着索引资源库并且判定使用了哪种编程语言。

向图中添加资源库和“注视”边的过程只是对我们本章之前工作的简单扩展。GitHub的“列出被加星的资源库列表”API（http://bit.ly/1a1oc8X）让找回某个特定用户加星的资源库列表变得非常容易，并且我们可以在这些结果上遍历并把与之前相同类型的节点和边加入到图中。示例7-12是实现这种功能的示例代码。它向内存中的图添加了数量可观的数据，这需要花一些时间来执行。如果你在处理有几十个以上的观星人的资源库，那么你就需要一些耐心来等待。

示例7-12：向图中添加带星标的资源库

```
   # Let's add each stargazer's additional starred repos and add edges
# to find additional interests.
MAX_REPOS = 500
for i, sg in enumerate(stargazers):
    print sg.login
    try:
        for starred in sg.get_starred()[:MAX_REPOS]: # Slice to avoid supernodes
            g.add_node(starred.name + '(repo)', type='repo', lang=starred.language, \
                       owner=starred.owner.login)
            g.add_edge(sg.login + '(user)', starred.name + '(repo)', type='gazes')
    except Exception, e: #ssl.SSLError:
        print "Encountered an error fetching starred repos for", sg.login, "Skipping."
    print "Processed", i+1, "stargazers' starred repos"
    print "Num nodes/edges in graph", g.number_of_nodes(), "/", g.number_of_edges()
    print "Rate limit", client.rate_limiting
```

对于创建这个图还有一个小顾虑，就是尽管大多数用户会对合理数目的资源库加星，但是某些用户可能会对非常多的资源库加星，导致远远超出平均数并且向图中引入了高度失衡数目的节点和边。如之前提及的，一个具有很多边数的节点，一个在边界外的离群数据，叫做超节点。通常来说我们都不希望建立带有超节点的图（尤其是内存中的图，比如使用NetworkX实现的图）因为在最好情况下，他们会让遍历等分析变得非常困难，在最差情况下，他们会引起内存溢出的错误。你可以根据自己的特殊情况和目的来决定要不要引入超节点。

一个用来避免向示例7-12的图中引入超节点的合理方法可以是简单地给每个用户的资源库的数目设立一个上限。在这种特殊情况下，我们通过削减在for循环get_starred（）中被遍历的数据结果来限定资源库的数目为一个比较合适的较大数值（500）。如果后来我们想要重新访问超节点我们仅需要查询具有比较多的外向边的节点来发现他们。

警告：Python，包括IPython Notebook服务器内核，在你向图中添加数据时会持续增加内存使用量。如果你尝试去创建一个你的操作系统无法运行的大图，一个内核管理进程可能会结束掉正在进行的Python进程。查看附录C中的“使用Vagrant和IPython Notebook监控并修复内存占用”在线版本来获取如何监控和增加IPython Notebook的内存使用。

既然已经建立好了一个包含额外资源库的图，我们可以开始实际地查询这个图。除了通过对数据结果进行计算来获取这个图的整体规模，我们现在还可以提出和回答一些问题，比如我们可以着重注意拥有最多资源库的用户，这可能会是很有趣的。可能最迫切的问题之一就是，除了用来创建原始的兴趣图谱的资源库外，图里使用最多的资源库是哪个。示例7-13展示了一个解决这个问题和提供一个未来分析出发点的示例程序。

注意：一些其他有用的属性来自PyGithub的get_starred API调用（对GitHub的“列出被加星的资源库”（http://bit.ly/1a1oc8X）API的封装），这些你可能会为了将来的实验而考虑。确定你阅读了API文档，以便了解你在探索该方面要涉及的所有有用信息。

示例7-13：对添加额外加星标的资源库更新后的图进行探索

```
   # Poke around: how to get users/repos
from operator import itemgetter
print nx.info(g)
print
# Get a list of repositories from the graph.
repos= [n for n in g.nodes_iter() if g.node[n]['type'] == 'repo']
```

```
# Most popular repos
print "Popular repositories"
print sorted([(n,d)
               for (n,d) in g.in_degree_iter()
                   if g.node[n]['type'] == 'repo'], \
               key=itemgetter(1), reverse=True)[:10]
print
# Projects gazed at by a user
print "Respositories that ptwobrussell has bookmarked"
print [(n,g.node[n]['lang'])
      for n in g['ptwobrussell(user)']
          if g['ptwobrussell(user)'][n]['type'] == 'gazes']
print
# Programming languages for each user
print "Programming languages ptwobrussell is interested in"
print list(set([g.node[n]['lang']
                for n in g['ptwobrussell(user)']
                    if g['ptwobrussell(user)'][n]['type'] == 'gazes']))
print
# Find supernodes in the graph by approximating with a high number of
# outgoing edges
print "Supernode candidates"
print sorted([(n, len(g.out_edges(n)))
               for n in g.nodes_iter()
                   if g.node[n]['type'] == 'user' and len(g.out_edges(n)) > 500], \
               key=itemgetter(1), reverse=True)
```

示例的输出如下：

```
   Name:
Type: DiGraph
Number of nodes: 48857
Number of edges: 116439
Average in degree: 2.3833
Average out degree: 2.3833
Popular repositories
[(u'Mining-the-Social-Web(repo)', 851),
 (u'bootstrap(repo)', 273),
 (u'd3(repo)', 194),
 (u'dotfiles(repo)', 166),
 (u'node(repo)', 139),
 (u'storm(repo)', 139),
 (u'impress.js(repo)', 125),
 (u'requests(repo)', 122),
 (u'html5-boilerplate(repo)', 114),
 (u'flask(repo)', 106)]
Respositories that ptwobrussell has bookmarked
[(u'Legal-Forms(repo)', u'Python'),
 (u'python-LinkedIn(repo)', u'Python'),
 (u'ipython(repo)', u'Python'),
 (u'Tweet-Relevance(repo)', u'Python'),
 (u'PyGithub(repo)', u'Python'),
 (u'Recipes-for-Mining-Twitter(repo)', u'JavaScript'),
 (u'wdb(repo)', u'JavaScript'),
 (u'networkx(repo)', u'Python'),
 (u'twitter(repo)', u'Python'),
 (u'envoy(repo)', u'Python'),
```

```
 (u'Mining-the-Social-Web(repo)', u'JavaScript'),
 (u'PayPal-APIs-Up-and-Running(repo)', u'Python'),
 (u'storm(repo)', u'Java'),
 (u'PyStratus(repo)', u'Python'),
 (u'Inquire(repo)', u'Python')]
Programming languages ptwobrussell is interested in
[u'Python', u'JavaScript', u'Java']
Supernode candidates
[(u'hcilab(user)', 614),
 (u'equus12(user)', 569),
 (u'jianxioy(user)', 508),
 (u'mcroydon(user)', 503),
 (u'umaar(user)', 502),
 (u'rosco5(user)', 502),
 (u'stefaneyr(user)', 502),
 (u'aljosa(user)', 502),
 (u'miyucy(user)', 501),
 (u'zmughal(user)', 501)]
```

一个最初的观察是新图中边的数目比之前的图多3倍，节点数目是之前的两倍。这正是由于复杂的网络动态性导致分析变得很有趣。但是，复杂的网络动态性也意味着NetworkX需要花费较长时间来建立全局图数据。请记住图在内存中并不意味着所有的计算会很快速地进行。这正是需要一些基础计算原理的知识的地方。

7.4.3.3 计算的考虑

注意：这一小节包含一些相对高级的讨论，比如运行图算法时的数学复杂性。推荐你阅读这些内容，不过如果你是第一次阅读本章，你也可以稍后回过头来读这部分。

对于那三种中心度的计算，我们知道对点度中心度的计算相对简单快速，只需要简单地通过不同的节点计算关联边的数量。中介中心度和接近中心度，却需要计算最小生成树（minimum spanning tree）

（http://bit.ly/1a1omgr）。NetworkX最小生成树算法（http://1.usa.gov/1a1omx6）实现了Kruskal的算法（http://bit.ly/1a1on3X），是计算机科学教育中很基本的一个算法。关于运行时复杂性，需要的时间为O（E logE），E代表图中边的个数。这个算法的复杂性是很有效的，不过100000*log（100000）大约需要一百万个操作，所以一个完整的分析会花费很多时间。

对于超节点的移除对于达到合理的网络图算法的运行时间很关键，提取感兴趣的子图（http://bit.ly/1a1oo80）来更加全面的分析是值得考虑的。例如，你可以通过过滤规则（比如关注者数）选择用户，这为判断他们在整个网络中的重要性提供了基础。你也可以考虑基于最少观星人数的阈值来筛选资源库。

当分析大型图时，建议你每次检视中心度度量中的一个，这样你能更快的遍历结果。为了达到更合理的运行时间从图中移除超节点也很关键，因为超节点在网络图算法中会占据大部分的时间。根据图的大小，有时扩大虚拟机可用的内存也是很有益的。参考附录C以获得更多的使用Vagrant压缩内存和配置的相关细节。

7.4.4 以节点为中心获得更高效的查询

另一个需要考虑的数据特性是用户使用编程语言的流行程度。很可

能用户会对使用他们熟悉的编程语言来实现的项目进行加星标注。即使现在图里存在可以分析用户和常用编程语言的数据和工具，我们的计划仍有一个缺点。因为一种编程语言在资源库中被模拟为一种属性，所以如果需要解决一些重要的问题我们需要扫描所有资源库节点并且根据这个属性来提取或者过滤。

例如，如果我们想要知道一个用户在当前计划中使用哪种编程语言，我们需要查找所有用户注视的资源库，提取出lang属性，然后计算出频率分布。这看起来好像不太繁琐，但是如果我们想要知道有多少用户在使用某种编程语言呢？虽然使用现在的方案结果是可计算的，但是它需要扫描所有资源库节点和计算所有的“注视”入边。通过对图的模式进行修改，回答这个问题可以和仅仅访问图中的一个节点一样简单。这个修改就是在图中为每种编程语言创建一个节点，它拥有programs入边来连接用户和编程语言，还拥有implements出边来连接资源库。

图7-6展示了我们最终的图的模式，它包含了编程语言，以及用户、编程语言、资源库之间的边。这种模式改变带来的总体效果就是我们从一个节点取出一个属性，然后在图中把之前隐式的关系创建成明显的关系。从完整性角度来看，没有新的数据产生，但是我们现在拥有的数据可以对某些查询进行更高效的计算了。虽然我们现在这种模式比较简单，但是所有可能与之相关的图可以很轻松地创建和挖掘，这带来的知识是巨大的。

相比于使用很多个节点拥有的属性来代表一种编程语言，使用一个单独的节点对应一种语言有很大优势，那就是一个节点是聚集的自然点。使用中央聚集点可以简化很多查询，比如找到最大值，这在2.3.2.2节有所描述。例如，使用NetworkX的团检测算法（clique detection algorithms）（http://bit.ly/1a1lWyo）可以更有效地找到关注其他用户并使用特定编程语言的用户的最大团，因为区域内某种编程语言节点的要求严重地限制了搜索性能。

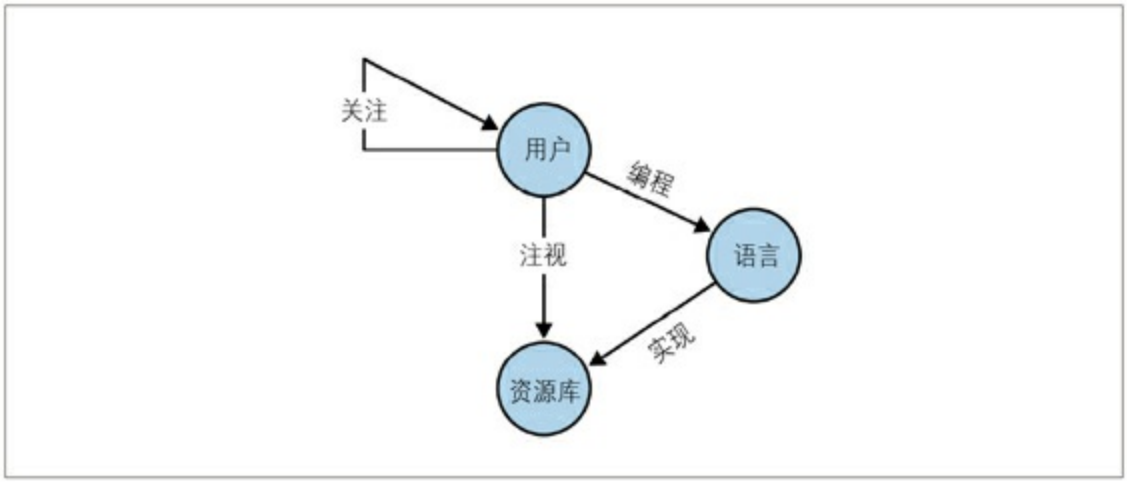


图7-6：包含GitHub用户、资源库和编程语言的图例

示例7-14介绍了一些实现最终图例中的改动的示例代码。由于所有用于创建额外节点和边的有用信息已经被展示在现有的图里（因为我们已经把编程语言作为一个属性储存在资源库节点里了），就没有对GitHub API的额外需求了。

示例7-14：升级图为包含编程语言节点的图

```
   # Iterate over all of the repos, and add edges for programming languages
# for each person in the graph. We'll also add edges back to repos so that
# we have a good point to "pivot" upon.
repos= [n
        for n in g.nodes_iter()
            if g.node[n]['type'] == 'repo']
for repo in repos:
    lang = (g.node[repo]['lang'] or "") + "(lang)"
    stargazers = [u
                  for (u, r, d) in g.in_edges_iter(repo, data=True)
                       if d['type'] == 'gazes'
                  ]
for sg in stargazers:
    g.add_node(lang, type='lang')
    g.add_edge(sg, lang, type='programs')
    g.add_edge(lang, repo, type='implements')
```

我们最终图的方案可以解决很多问题。有一些问题适合在这里提出来研究：

·特定用户使用哪种语言来编程？

·有多少用户使用某种特定语言来编程？

·哪些用户使用多种语言来编程，比如Python和JavaScript？

·哪个编程者会用最多的编程语言？

·特定的编程语言中存在更密切的联系吗？（比如，如果已知一个用户使用Python编程，那么基于图中数据，他更可能还会使用JavaScript还是Go语言？）

示例7-15提供了一些示例代码，可以作为回答这些问题和类似问题的一个好的起点。

示例7-15：对最终图的示例查询

```
   # Some stats
print nx.info(g)
print
# What languages exist in the graph?
print [n
       for n in g.nodes_iter()
           if g.node[n]['type'] == 'lang']
print
# What languages do users program with?
print [n
       for n in g['ptwobrussell(user)']
           if g['ptwobrussell(user)'][n]['type'] == 'programs']
# What is the most popular programming language?
print "Most popular languages"
print sorted([(n, g.in_degree(n))
 for n in g.nodes_iter()
     if g.node[n]['type'] == 'lang'], key=itemgetter(1), reverse=True)[:10]
print
# How many users program in a particular language?
python_programmers = [u
                      for(u, l) in g.in_edges_iter('Python(lang)')
                          if g.node[u]['type'] == 'user']
print "Number of Python programmers:", len(python_programmers)
print
javascript_programmers = [u for
                          (u, l) in g.in_edges_iter('JavaScript(lang)')
                           if g.node[u]['type'] == 'user']
print "Number of JavaScript programmers:", len(javascript_programmers)
print
# What users program in both Python and JavaScript?
print "Number of programmers who use JavaScript and Python"
print len(set(python_programmers).intersection(set(javascript_programmers)))
# Programmers who use JavaScript but not Python
print "Number of programmers who use JavaScript but not Python"
print len(set(javascript_programmers).difference(set(python_programmers)))
# XXX: Can you determine who is the most polyglot programmer?
```

示例的输出如下：

```
   Name:
Type: DiGraph
Number of nodes: 48952
Number of edges: 174180
Average in degree: 3.5582
Average out degree: 3.5582
[u'PHP(lang)', u'Clojure(lang)', u'ActionScript(lang)', u'Logtalk(lang)',
 u'Scilab(lang)', u'Processing(lang)', u'D(lang)', u'Pure Data(lang)',
 u'Java(lang)', u'SuperCollider(lang)', u'Julia(lang)', u'Shell(lang)',
 u'Haxe(lang)', u'Gosu(lang)', u'JavaScript(lang)', u'CLIPS(lang)',
 u'Common Lisp(lang)', u'Visual Basic(lang)', u'Objective-C(lang)',
 u'Delphi(lang)', u'Objective-J(lang)', u'PogoScript(lang)',
```

```
 u'Scala(lang)', u'Smalltalk(lang)', u'DCPU-16 ASM(lang)',
 u'FORTRAN(lang)', u'ASP(lang)', u'XML(lang)', u'Ruby(lang)',
 u'VHDL(lang)', u'C++(lang)', u'Python(lang)', u'Perl(lang)',
 u'Assembly(lang)', u'CoffeeScript(lang)', u'Racket(lang)',
 u'Groovy(lang)', u'F#(lang)', u'Opa(lang)', u'Fantom(lang)',
 u'Eiffel(lang)', u'Lua(lang)', u'Puppet(lang)', u'Mirah(lang)',
 u'XSLT(lang)', u'Bro(lang)', u'Ada(lang)', u'OpenEdge ABL(lang)',
 u'Fancy(lang)', u'Rust(lang)', u'C(lang)', '(lang)', u'XQuery(lang)',
 u'Vala(lang)', u'Matlab(lang)', u'Apex(lang)', u'Awk(lang)', u'Lasso(lang)',
 u'OCaml(lang)', u'Arduino(lang)', u'Factor(lang)', u'LiveScript(lang)',
 u'AutoHotkey(lang)', u'Haskell(lang)', u'HaXe(lang)', u'DOT(lang)',
 u'Nu(lang)', u'VimL(lang)', u'Go(lang)', u'ABAP(lang)', u'ooc(lang)',
 u'TypeScript(lang)', u'Standard ML(lang)', u'Turing(lang)', u'Coq(lang)',
 u'ColdFusion(lang)', u'Augeas(lang)', u'Verilog(lang)', u'Tcl(lang)',
 u'Nimrod(lang)', u'Elixir(lang)',u'Ragel in Ruby Host(lang)', u'Monkey(lang)',
 u'Kotlin(lang)', u'C#(lang)',u'Scheme(lang)', u'Dart(lang)', u'Io(lang)',
 u'Prolog(lang)', u'Arc(lang)', u'PowerShell(lang)', u'R(lang)',
 u'AppleScript(lang)', u'Emacs Lisp(lang)', u'Erlang(lang)']
[u'JavaScript(lang)', u'Java(lang)', u'Python(lang)']
Most popular languages
[(u'JavaScript(lang)', 851), (u'Python(lang)', 715), ('(lang)', 642),
 (u'Ruby(lang)', 620), (u'Java(lang)', 573), (u'C(lang)', 556),
 (u'C++(lang)', 508), (u'PHP(lang)', 477), (u'Shell(lang)', 475),
 (u'Objective-C(lang)', 435)]
Number of Python programmers: 715
Number of JavaScript programmers: 851
Number of programmers who use JavaScript and Python
715
Number of programmers who use JavaScript but not Python
136
```

虽然图例从概念上来说很简单，由于额外的编程语言节点导致边数已经增加了近50%！正如我们从一些示例查询的输出中看到的，有相当多的编程语言在被使用，其中JavaScript和Python最为常用。最初的感兴趣资源库的初始源代码是用Python编写的，所以JavaScript是用户中更为常用的语言可能暗示着有一批网页开发的用户。当然，JavaScript本身就是一种常用的编程语言，并且经常会使用JavaScript来编写客户端程序，用Python来编写服务端的程序。Ruby也是一种常用编程语言，通常被用于作为网页服务端的开发。它在排名中排第四。‘（lang）’成为第三常用的编程语言表示着有642个GitHub无法指明编程语言的资源库，所以把它们聚集起来归为这单独一类。

分析表达人们对别人、资源库中的开源项目和编程语言的兴趣的图潜力是巨大的。无论你选择做什么分析，仔细考虑问题的本质然后仅从图中取出相关数据来分析——对使用NetworkX图的subgraph方法提取出的一系列节点进行校正，或者根据类型或频率的阈值来过滤出节点。

注意：由于用户和编程语言本质的内在联系，使用二分分析（bipartite analysis）（http://bit.ly/1a1oooP）会很有价值。一个二分图包含两个不相交的顶点集合，通过两个集合间的边连接起来。此时你可以轻易地从图中删除资源库节点来极大地提高计算全局图数据的效率（边的数目会减少超过100000）。

7.4.5 兴趣图谱的可视化

虽然对图的可视化很令人兴奋，一张图有时比1000个词还有价值，但是要记住，并不是所有的图都能很容易地进行可视化。然而，想一想，你通常可以提取出子图来可视化，直到它对你正解决问题提供了见解为止。正如你从本章中学到的，图只是一种数据结构并且没有明确的视觉呈现方式。为了可视化，特定的布局算法会得到应用，以便将节点和边映射到二维或三维空间以便可视化。

我们会专注于本书中一直使用的关键工具，并依赖NetworkX导出JSON的功能，这就能通过JavaScript工具D3（http://bit.ly/1a1kGvo）来呈

现；不过还有很多其他的可视化工具。

Graphviz（http://bit.ly/1a1ooVG）是可高度配置的经典工具，能将复杂的图进行布局并以位图的形式呈现。它曾经和其他的命令行工具一样，在终端中配置使用，不过现在在大多数平台上使用它带有界面的版本。另一种选择是Gephi（http://bit.ly/1a1opc5），这是另一个流行的开源项目，它提供了强大的交互可能；Gephi在过去几年中成长迅速，是很值得考虑的一种工具。

示例7-16展示了提取注视我们初始图的种子（Mining-the-Social-Web资源库）的用户的子图和他们之间的“关注”关系。这提取出了图中有共同兴趣的用户，并对他们之间的“关注边”进行了可视化。记住本章构建的整张图非常大，包含成千上万个节点和成百上千条边，所以为了使用像Gephi这样的工具来进行合理的可视化，你需要花些时间来更好的理解。

示例7-16：社交网络的初始兴趣图谱的可视化

```
   import os
import json
from IPython.display import IFrame
from IPython.core.display import display
from networkx.readwrite import json_graph
print "Stats on the full graph"
print nx.info(g)
print
# Create a subgraph from a collection of nodes. In this case, the
# collection is all of the users in the original interest graph
mtsw_users = [n for n in g if g.node[n]['type'] == 'user']
h = g.subgraph(mtsw_users)
print "Stats on the extracted subgraph"
print nx.info(h)
# Visualize the social network of all people from the original interest graph.
d = json_graph.node_link_data(h)
```

```
json.dump(d, open('resources/ch07-github/force.json', 'w'))
# IPython Notebook can serve files and display them into
# inline frames. Prepend the path with the 'files' prefix.
# A D3 template for displaying the graph data.
viz_file = 'files/resources/ch07-github/force.html'
# Display the D3 visualization.
display(IFrame(viz_file, '100%', '600px'))
```

图7-7展示了运行示例代码的示例结果。

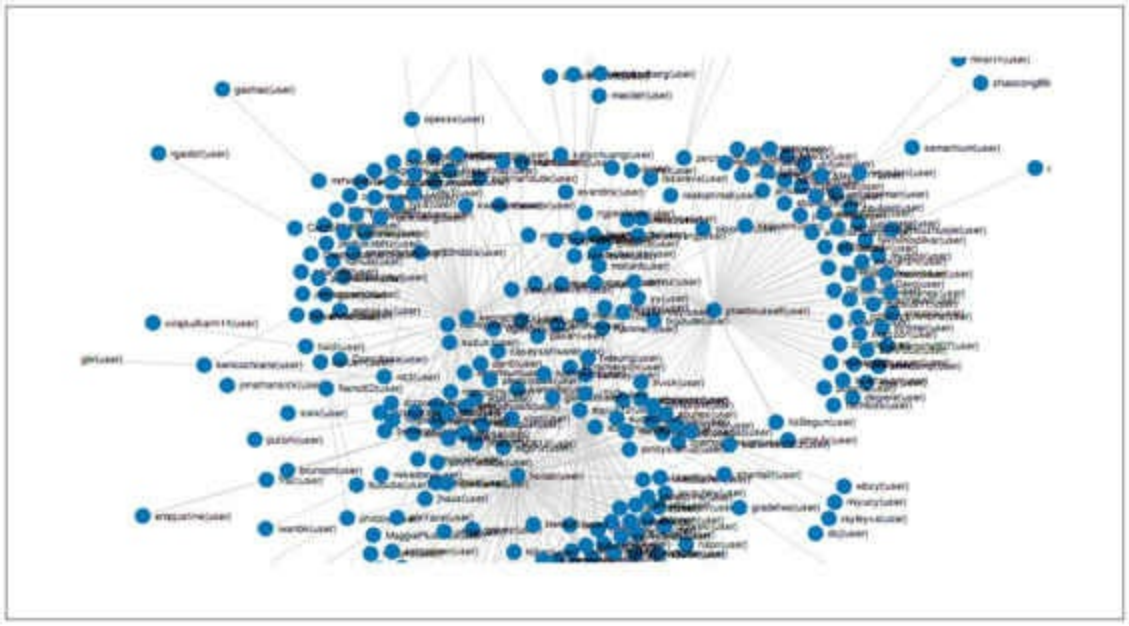


图7-7：对GitHub用户的兴趣图谱中的“关注”边的交互式可视化——注意该图的可视化布局的模式符合本章前面介绍的中心度度量

[1] 在当前的讨论中，术语“能量”用来描述抽象图中的流。

7.5 本章小结

注意：本章和其他章节的源代码在GitHub（http://bit.ly/1a1kNqy）中有方便的IPython Notebook格式，强烈推荐你去尝试。

尽管很多种类的图在本书前面章节都提到过，但本章重点介绍了它们的用途，并展示了GitHub用户网络和它们在特定的软件项目资源库和编程语言上的共同感兴趣的灵活的数据格式。GitHub丰富的API和NetworkX简单实用的API对于挖掘有吸引力的和常被忽略的社交网络数据都很有用，这是广泛使用的很明显的“社交”网络属性之一。兴趣图谱的观点并不完全是新的，但它在社交网络上的应用确实是近几年才发展起来的。尽管兴趣图谱（或者类似的展示）曾经被广告商用于有效地展示广告，但它们现在被企业或软件开发者有效地利用，用来获取用户的兴趣并进行智能地推荐，这可以提高产品对用户的相关性。

和本书其他章节一样，本章仅仅是图形建模、兴趣图谱、GitHub API还有你用这些技术能做的工作的指南。你应该可以简单地将本章的图形建模技术应用在其他社交网络属性中，比如Twitter或者Facebook并获得引人入胜的分析结果，还可以将其他形式的分析应用在GitHub API提供的丰富的数据上。这通常都会充满无限可能。我最大的希望是你能在本章的练习中获得享受，并学习到你在未来的数据挖掘之路上能够用

上的新知识。

7.6 推荐练习

·重复本章的练习，但要使用不同的资源库作为开始。本章的发现大体上是正确的吗？还是你实验的结果是有所不同的？

·GitHub发布了关于编程语言的关系（http://bit.ly/1a1or3Y）的数据。回顾并探索这些数据。和你用GitHub API获取的数据有什么不同吗？

·NetworkX提供了丰富的图的遍历算法（http://bit.ly/1a1orkk）。回顾文档并选择不同的算法运行数据。中心度度量、团和二分算法也许是一个不错的开始。你能计算出图中用户的最大团吗？能计算出有共同兴趣（比如特定的编程语言）的用户的最大团吗？

·GitHub档案（http://bit.ly/1a1orAK）中提供了丰富的GitHub全局的活动数据。调查这些数据，使用推荐的“大数据”工具来探索它们。

·对比两个相似的GitHub项目的数据。Mining-the-Social-Web和Mining-the-Social-Web-2nd-Edition有千丝万缕的联系，所以这两个项目是很合适作为分析的开始的。哪些人收藏或派生了其中一个而不是另一个？如何比较兴趣图谱？你能创建并分析包含对这两个版本都感兴趣的用户的兴趣图谱吗？

·使用相似度度量，比如Jaccard相似度（参考nltk.metrics.jaccard_distance，以及4.4.4.1节）来计算两个任意GitHub用户基于特征（比如加星的相同的资源库、编程语言或其他你能用GitHub API获取的特征）的相似度。

·已知用户和已有的兴趣，你能设计一个为其他用户推荐兴趣的算法吗？可以考虑应用4.3节的代码，它使用了余弦相似度作为预测相关性的方法。

·使用柱状图直观的感受本章兴趣图谱的一个方面，比如编程语言的流行度。

·探索图的可视化工具，比如Graphviz和Gephi，对图的可视化进行布局。

·你可能知道Google在2013年中期停止了它的Google Reader产品。留下的讨论主题之一就是订阅图（http://tcrn.ch/1a1opZV）这一更加去中心化并面向订阅内容的“关注”机制是否已经出现了。你认为将来Web的未来是否会有将相同兴趣的人更简单的聚集起来的技术模式改革？现在已经发生了吗？

·探索Friendster社交网络和ground-truth社区（http://stanford.io/1a1orRr）数据并使用NetworkX算法来分析它。

7.7 在线资源

下面链接的列表是本章有用的在线资源：

·二分图（http://bit.ly/1a1oooP）

·中心度度量（http://bit.ly/1a1osEM）

·为命令行使用创建GitHub OAuth令牌（http://bit.ly/1a1o7lG）

·D3.js（http://bit.ly/1a1kGvo）

·Friendster社交网络和ground-truth社区（http://stanford.io/1a1orRr）

·Gephi（http://bit.ly/1a1opc5）

·GitHub档案（http://bit.ly/1a1orAK）

·GitHub开发者（http://bit.ly/1a1o49k）

·GitHub分页的开发者文档（http://bit.ly/1a1o9Ki）

·GitHub访问速率上限限制的开发者文档（http://bit.ly/1a1oblo）

·gitscm.com（在线Git文档，http://bit.ly/1a1o2hZ）

·图论简介——YouTube视频（http://bit.ly/1a1odto）

·Graphviz（http://bit.ly/1a1ooVG）

·超图（http://bit.ly/1a1ocWm）

·兴趣图谱（http://bit.ly/1a1o3Cu）

·Krackhardt风筝图（http://bit.ly/1a1oixa）

·Kruskal算法（http://bit.ly/1a1on3X）

·最小生成树（http://bit.ly/1a1omgr）

·NetworkX（http://bit.ly/1a1ocFV）

·NetworkX文档（http://bit.ly/1a1ocFV）

·NetworkX图的遍历算法（http://bit.ly/1a1orkk）

·PyGithub的GitHub资源库（http://bit.ly/1a1o7Ca）

·Python线程同步：锁、可重入锁、信号量、条件变量、事件和队列（http://bit.ly/1a1oAUQ）

·线程池模式（http://bit.ly/1a1mW5M）

·线程安全（http://bit.ly/1a1oE73）

第8章　挖掘带标记语义网：提取微格式、推断资源描述框架等

前面的章节概述了社交网络方面的一些热门网站，接下来本章将进一步针对语义网进行务实的讨论。语义网（semantic web）作为Web的一种，与你们已经体会的Web很像。有一点不同的是，它演变成由机器编程提取大量的信息，且能够基于这些信息进行推理并自动做出决策。

语义网本身就值得作为主题写成一本书，但本章不打算单纯讨论它的技术，而是只撷取一些有趣的技术话题。本章尝试将十分实际的社交网络与更宏伟的语义网联系在一起。本章提出了比之前的章节还多的假设，也涉及了许多重要的技术。

本章中你将从实践者的角度学习到的一个重要的主题是微格式（microformat）。它是一种很简单的技术，会在网页中嵌入含义明确的结构化数据。机器能够方便的对其进行解析并用于各种自动化推理。当你进一步学习后，会发现微格式是网络发展历程中浓墨重彩的一笔。IndieWeb（http://bit.ly/15ANScQ）以及一些大公司正通过Shema.org积极地研究它。微格式对社交网络挖掘有巨大的影响。

在本章后面几节，我们将简单介绍使用归纳逻辑针对事实进行推理的技术，讨论它们如何把微格式融入语义网的前景。在本章结束时，你

应该清晰认识到，嵌入语义元数据到网页中的技术在主流Web中所处的地位，以及它与最先进的技术进行比较会怎样。

注意：在线网址http://bit.ly/MiningTheSocialWeb2E可以查看本章最新修订的（以及其他章节）源码。此外，一定要利用本书的虚拟机体验，具体在附录A中有描述，以最大限度地学习样例代码。

8.1 概述

本章对比前面的章节提出了更多假设和一些对Web未来的有益的观点。本章你将学习到：

·微格式的一般形式。

·如何从网页中识别和操纵一般的微格式。

·Web中微格式的最新工具。

·语义网及其技术的概述。

·使用Python工具包FuXi推断语义网数据。

8.2 微格式：易于实现的元数据

在网络的不断发展中，微格式是向前迈出的重要一步。因为它为嵌入“智能数据”到网页提供了一个有效的机制，并且对于内容开发者来说易于实现。简单来说，微格式（http://bit.ly/1a1oEEd）是一种简单的规定，它以一种完全增值的方式精准地将元数据（metadata）嵌入到网页中。本章首先简要地介绍微格式的全貌，接着深入研究一些具体使用geo（http://bit.ly/1a1oFb6）、hRecipe（http://bit.ly/1a1oHjn）和hResume（http://bit.ly/1a1oFIf）微格式的实例。我们将看到其中的一些微格式是如何建立在其他更基础微格式（如hCard（http://bit.ly/1a1oFYG），本章也会对它进行探讨）上的。

虽然看起来把微格式的数据（如geo或是hRecipe），称作“社交数据”多少有点夸张，事实上，它在社会数据聚合（mashup）中扮演着越来越重要的作用。本书的第1版是在2011年初发行的，当时的情况是这样的：近一半的Web开发者报告了一些微格式的使用情况（http://bit.ly/1a1oInm）；微格式社区microformats.org（http://bit.ly/1a1oIDL）刚刚庆祝了它的5岁生日（http://bit.ly/1a1oJaY）；Google指出有94%的富摘要（rich snippets）中包含了微格式（http://bit.ly/1a1oJaY）。从那时起，Google的富摘要项目不断地获得新动力。接着，Schema.org（http://bit.ly/1a1oEnv）出现了，

它试图保证各种供应商能够共享词汇，实现与语义标记相媲美的技术，如微格式、HTML5微数据（microdata）和RDFa。因此，你应该会看到语义标记整体在未来的持续增长，虽然其中各部分可能并不是按照同一速度增长。此外，市场动态、企业政治以及技术趋势也都发挥着不可或缺的作用。

注意：Web Data Commons（http://bit.ly/1a1oJI1）从Common Crawl（http://bit.ly/1a1oM6A）网页语料库中提取像微格式一样的结构化数据，然后大众才能进一步学习使用它们。特别是2012年8月的语料库（http://bit.ly/1a1oKf0）中的信息提供了一些有趣的统计数据，说明由Schema.org发起的联合行动的作用十分显著。

微格式在很大程度上用于弥补描述相当不明确的网与更富含语义的网之间的差距。前种网主要基于人类可识别的HTML 4.01（http://bit.ly/1a1oKvt），后种网对机器解析更为明确和友好。微格式的魅力在于，给那些与生活和社交活动（如日程表、简历、食物、产品、产品评价等）相关的结构化数据提供了一种嵌入网页的方式。目前，它们以完全向后兼容的方式嵌入到HTML标签中。表8-1提供了一部分你很可能会在网站遇到的比较流行的微格式和相关项目。想要查看更多例子，详见网址http://bit.ly/1a1oKLV。

表8-1：嵌入结构化数据到网页的一些热门技术

技术	目的	流行程度	标签	类型
XFN	表示超链接中人可识别的关系	21世纪初广泛使用在博客平台来隐式表示社交图。在Google的社交图API淡出后迅速活跃。目前，用在IndieWeb的RelMeAuth项目中，提供网络登录服务	语义HTML、XHTML	微格式
geo	嵌入人和物体的地理坐标信息	广泛使用，尤其是用于像OpenStreetMap和Wikipedia一类的网站	语义HTML、XHTML	微格式
hCard	识别人、公司和其他联系人信息	广泛使用并且被其他热门的微格式包含，如hResume	语义HTML、XHTML	微格式
hCalendar	嵌入日历数据	继续平稳增长（*http://bit.ly/1a1oOeJ*）	语义HTML、XHTML	微格式
hResume	嵌入简历信息	广泛应用于像LinkedIn网站，为不少于2亿全球用户（*http://linkd.in/1a1oP2m*）以hResume形式提供通用简历（*http://bit.ly/1a1oRap*）	语义HTML、XHTML	微格式
hRecipe	识别菜谱	广泛应用于食品网站，如about.com（*http://bit.ly/1a1oPzm*）的子域（thaifood.about.com（*http://abt.cm/1a1oPPX*））	语义HTML、XHTML	微格式

表8-1：嵌入结构化数据到网页的一些热门技术（续）

技术	目的	流行程度	标签	类型
微数据	嵌入HTML5授权网页的键值对	作为HTML5一部分出现的技术，并且因为Google的富摘要项目和Schema.org稳定地增长着其交易份额	HTML5	W3C项目
RDFa	根据领域专家创建的专业词汇嵌入精确事实到XHTML网页中	Facebook的开放图协议（*http://bit.ly/1a1lu3m*）和开放图概念的基础，热度迅猛增长。然而，根据特定词汇多多少少有些击中或者错过	XHTML[a]	W3C项目
开放图协议	嵌入真实世界事物的轮廓到XHTML网页中	迅猛增长，考虑到Facebook不少于100亿用户（*http://on.wsj.com/1a1oTit*），仍有很大潜力	XHTML (RDFa-based)	Facebook平台项目

a：成文之际，嵌入RDFa到语义标记和HTML5中的行动正在持续升温，详见W3C HTML+RDFa 1.1工作草案（http://bit.ly/1a1oSLx）。

如果你十分了解Web短暂的历史的话，你会发现创新频现，同时也会明白网站运营的高度分散化的方式并不利于应对突如其来的变革。然而实际上，变化是以一种渐进的方式连续而平稳地发生的。比如，由于博客链接的热度下降，并且像Facebook，Twitter等一些大的社交网，它们在社交数据方面着重研究自己的项目而未采用XFN，因此2013年中期的XFN（http://bit.ly/1a1oTyX）（XHTML的朋友网）似乎丧失了展现社会图的大部分动力。重大的技术投资，如Google的社交图API，使用XFN作为基础并且使它的热度渐渐地回升起来。然而，Google的社交图API在2012年初的退出（http://googleblog.blogspot.com/2012/01/renewing-old-resolutions-for-new-year.html）又使这热度相对减弱了。许多社交网

技术的实践者们直接利用Google社交图API，把它作为XFN等微格式的代理者，而并不选择直接利用XFN。

我们回过头来看，虽然直接建立在XFN上的方式是个较安全的方式，但把Google社交图API作为代理方式的方便性更占优势，并且多少有点讽刺的是它导致了XFN的“软复位”。然而现在，XFN属于IndieWeb（http://bit.ly/1a1oWuT）的叫RelMeAuth举措（http://bit.ly/1a1oUD4）（为个人网站或是其他社交网站的后台登录身份验证（http://bit.ly/1a1oUDd）提供了开源标准的技术）的一部分，它正在蓬勃发展。

注意：就像XFN为适应网络新需求而平稳地前进一样，微格式也在为了填补网络上“智慧数据”这一空白而不断变化，同时也为大众架起了现有技术和新兴技术间的桥梁。

你还会遇到很多其他类型的微格式，虽然经验告诉我们应当多观察像Google、必应、Facebook等大公司的新举措，但我们也不能忽视那些定义开放标准的群体的动作。微格式或是语义从有显著影响力的玩家那里获得的支持越多，它成功的可能性也就越大，同时对数据挖掘也更有益处，但永远不要低估那些支持非特定法人实体的有目的的社区领袖的影响。参与Schema.org（http://bit.ly/1a1oEnv）行动的供应商们以及IndieWeb和W3C的一些举措将受益于这些近距离的深入观察。

注意：请查看5.2节的“HTML、XML和XHTML”边注了解其中与语义标记相关的部分。

8.2.1 地理坐标：一个几乎万能的常见思路

使用微格式的影响是微妙且深远的：当你正在阅读一篇关于像富兰克林、田纳西州这样的地方的文章时，你很容易知道地图上的点代表着城镇的位置，但对于机器人来说，如果它没有配置不同模式的专门逻辑，就很难得到这个结论。然而，对于人来说，在整个页面上的全面扫描是杂乱无章的，因为当你以为你找到了全部的可能性时，你也许还会发现你遗漏了一种可能性。恰当的语义嵌入页面，以一种甚至能够让Robby机器人（http://bit.ly/1a1oXii）理解的方式，对非结构化的数据进行标记。这样能够消除二义性，同时也能降低爬取者和开发者的扫描难度。无论是对于生产者还是消费者来说，这都是一个双赢的局面，并且我们也期望它的最终效果是所有人都有所创新。

虽然地理数据本身并没有社交属性，但分散数据中的那些相似的地理环境内容却建立了人们之间许多重要但并不明显的联系。

地理数据无处不在，它在社交聚合（social mashup）中发挥着很重要的作用，因为空间里一个特定的地理位置就像胶水一样将分散的人聚集于一处。它缩小了现实生活和网络生活之间的鸿沟，一旦它与现实世

界中的一个特定个体绑定起来，任何形式的数据都可以相互交流了。举例来说，你可以通过住址和喜爱的食物来区分人们。本节包含挖掘、分析、可视化地理微格式数据的几部分内容。

网页中嵌入地理位置信息的微格式中最简单且应用最广泛的一种叫做geo（http://bit.ly/1a1oFb6），它的规范的灵感来源于vCard（http://bit.ly/1a1oVqL）中与之同名的一个属性，它提供了一种精确定位的手段。使用geo的嵌入微格式有两种方式，下面这段HTML脚本用这两种技术方法来描述富兰克林、田纳西州这两地：

```
   <!-- The multiple class approach -->
<span class="geo">
    <span class="latitude">36.166</span>
    <span class="longitude">-86.784</span>
</span>
<!-- When used as one class, the separator must be a semicolon -->
<span class="geo">36.166; -86.784</span>
```

正如所见，微格式简单地用相应的类名标记了精度和纬度值，并在最外层用geo类标签包住了它们。许多热门网站，包括Wikipedia（http://bit.ly/1a1oVXJ）和OpenStreetMap（http://bit.ly/1a1oYms），使用geo和其他微格式来显示结构化的数据。

注意：geo的一个常见应用就是对用户隐藏编码信息。传统CSS有两种方式，style="display：none"和style="visibility：hidden"。前者清除了页面上的全部元素使得布局也消失了，后者则是隐藏了具体内容却保

留它所占据的空间。

示例8-1说明了一个简单的程序，它从Wikipedia网页中解析geo微格式数据，来为你展示如何从应用geo微格式的内容中抽取坐标信息。需要注意的是Wikipedia定义了一个bot政策（http://bit.ly/1a1oZ9T），其内容就是优先检索像下面内容一样的脚本内容。重点则是你需要下载一些Wikipedia定期更新且并不通过编程从热门网站中借用重要数据的数据档案。（为了教育目的如此做事很有意义的。）

警告：应当时常认真查看网站的服务条款，并确保你运行的脚本遵循最新的指导方针。

示例8-1：从Wikipedia网页中抽取geo微格式的数据

```
   import requests # pip install requests
from BeautifulSoup import BeautifulSoup # pip install BeautifulSoup
# XXX: Any URL containing a geo microformat...
URL = 'http://en.wikipedia.org/wiki/Franklin,_Tennessee'
# In the case of extracting content from Wikipedia, be sure to
# review its "Bot Policy," which is defined at
# http://meta.wikimedia.org/wiki/Bot_policy#Unacceptable_usage
req = requests.get(URL, headers={'User-Agent' : "Mining the Social Web"})
soup = BeautifulSoup(req.text)
geoTag = soup.find(True, 'geo')
if geoTag and len(geoTag) > 1:
    lat = geoTag.find(True, 'latitude').string
    lon = geoTag.find(True, 'longitude').string
    print 'Location is at', lat, lon
elif geoTag and len(geoTag) == 1:
    (lat, lon) = geoTag.string.split(';')
    (lat, lon) = (lat.strip(), lon.strip())
    print 'Location is at', lat, lon
else:
    print 'No location found'
```

下面是例子的结果，说明了输出结果正如我们所想的是一组坐标

值：

```
   Location is at 35.92917  -86.85750
```

为了使输出结果更有意思一点，你可以使用内联框架的形式直接显示在IPython Notebook中，如示例8-2所示。

示例8-2：在IPython Notebook中用Google地图形式显示geo微格式

```
   from IPython.display import IFrame
from IPython.core.display import display
# Google Maps URL template for an iframe
google_maps_url = "http://maps.google.com/maps?q={0}+{1}&" + \
  "ie=UTF8&t=h&z=14&{0},{1}&output=embed".format(lat, lon)
display(IFrame(google_maps_url, '425px', '350px'))
```

执行上面例子在IPython Notebook中显示的结果见图8-1。

Example 2. Displaying geo microformats with Google Maps in IPython Notebook

```
In [38]: from IPython.display import IFrame
         from IPython.core.display import display

         # Google Maps URL template for an iframe

         google_maps_url = "http://maps.google.com/maps?q={0}+{1}&amp;" + \
           "ie=UTF8&amp;t=h&amp;z=14&amp;{0},{1}&amp;output=embed".format(lat, lon)

         display(IFrame(google_maps_url, '425px', '350px'))
```

图8-1：IPython Notebook显示内联框架的这一功能增添了许多交互活动，也方便了你的数据分析实验

一旦你找到一个嵌入了有趣的地理信息数据的网页，可能你想要做的第一件事就是将它可视化。比如说，让我们来看看Wikipedia上叫“List of National Parks of the United States”的文章（http://bit.ly/1a1oYTl），它显示了国家公园漂亮的表格视图，并且用geo格式标记了它们，但是快速将数据加载到交互式工具上不是更方便查看吗？

microform.at（http://microform.at）是一个了不起的小型Web服务，它可以从给定的URL上提取几种微格式，并且以各种有用的格式将它们传递

回来。它显示了探测和与网页中微格式数据交互的各种选择，见图8-2。

图8-2：Wikipedia名为“List of National Parks of the United States”的文章在microform.at上的显示结果

如果你可以选择，KML（http://bit.ly/1a1p1Pb）（Keyhole标记语言）输出是一种最常见的可视化geo数据的方式。你可以通过下载Google Earth并且在本地载入KML文件，或者只需直接在Google Maps搜索栏中输入包含KML数据的URL。在microform.at显示的结果中单击“KML”链接会触发下载该文件，你可以在Google Earth中使用该文件，但你也可以将它复制到剪贴板上，再将他传递给Google Earth。

图8-3展现了Google Earth对http://microform.at/?type=geo&url=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FList_of_U.S._

的可视化——前面提到的那篇文章的KML结果，这只是使用type和url查询字符串参数的基地址URL http://microform.at。

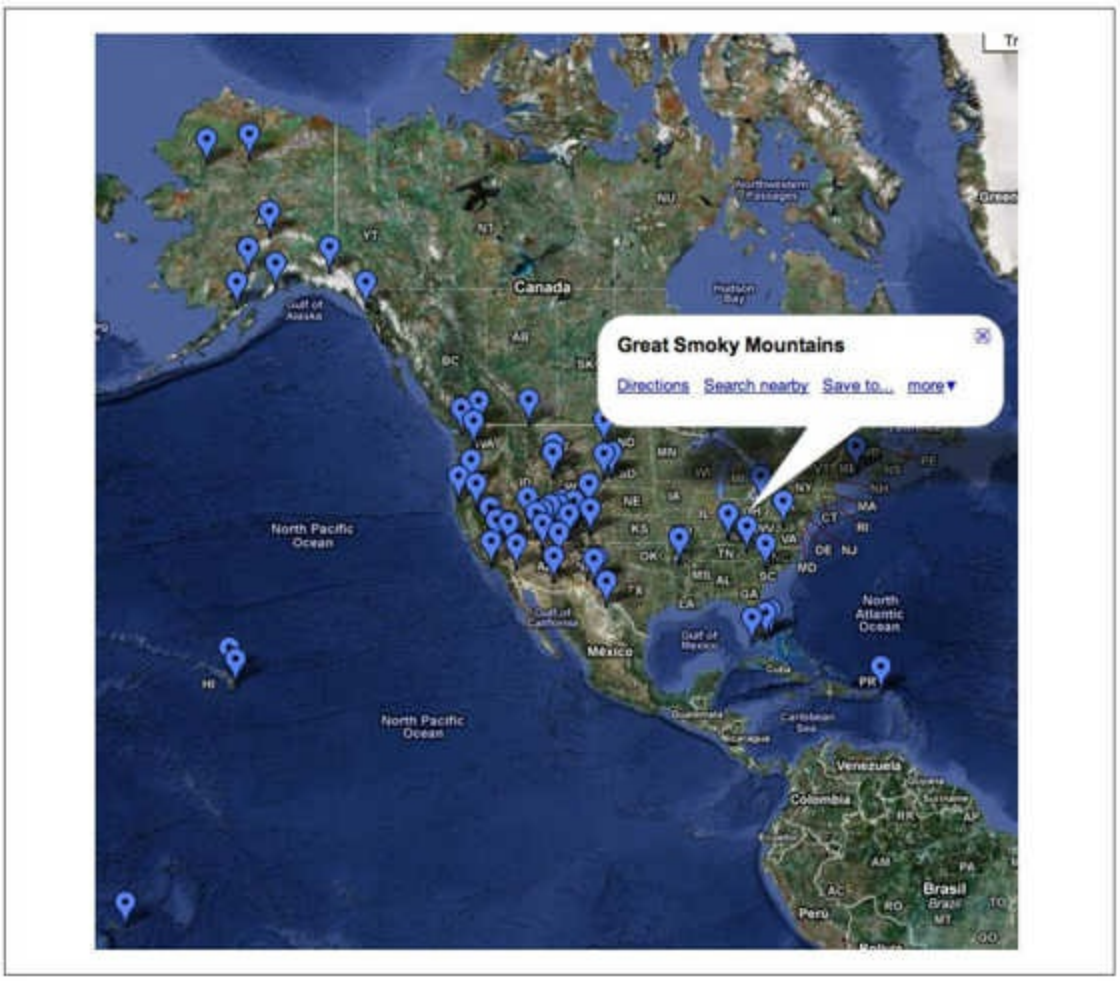


图8-3：当传递了microform.at中的KML结果后，显示的所有美国国家公园的Google Maps结果

从“包含了像geo数据这类语义标记”的Wikipedia的文章开始，并将这些信息可视化需要很强的分析能力，因为它能轻而易举地快速传递出它的内涵。例如火狐浏览器的插件（http://mzl.la/1a1p2Cy）等浏览器拓展就是为了进一步减少这过程中的代价。一个章节只能简要说这些，你

可以花一个小时左右把本节中国家公园的数据和LinkedIn职业网络获得的联系人信息结合在一起，看看如何会在你下一次（可能是人为的）出差中发现更多乐趣。（查看3.3.4.4节，了解如何通过应用k-means的方法寻找聚类，并计算这些聚类的中心来收集并分析geo数据）。

8.2.2 使用菜谱数提高线上相亲成功率

自从Google的富摘要举措的起步引发了人们对微格式前所未有的关注，许多热门的美食网站进一步使用hRecipe和hReview来显示菜谱和评论。考虑到虚拟网上约会服务的潜力，它获得了博客和其他社交群的芳心，人们正努力实现网络上的配对儿相亲。人们可能会看到这样的情况，使用大量链接到具体某一人的geo和hRecipe信息，会对他首次相亲的成功率产生巨大的影响。

可以通过以下两种标准来给人们配对：他们居住的地方有多远和他们喜欢吃什么。例如你可以想象两个都喜欢有机原料煮的素食的人之间的晚餐约会会比烧烤爱好者和素食主义者间的约会效果更好。饮食喜好和是否使用某种过敏原或有机原料等这些信息都是成功配对的有用线索，可以强化经营理念。虽然我们并不打算推出新的在线约会服务，但万一你决定采纳这个想法并进一步实现它，我们就会开始这样做。

About.com（http://abt.cm/1a1p11C）是最为流行的使用微格式改进

全部网络的在线网址之一，它用hRecipe微格式（http://bit.ly/1a1oHjn）的形式展现菜谱信息，并且在菜谱的评论中使用了hReview微格式（http://bit.ly/1a1p1yq）；由于Schema.org行动利用这种信息来进行Web搜索带来了优势，epicurious（http://epi.us/1a1p1yH）和其他一些热门网站也沿袭了这个套路。本节简要说明搜索引擎（或是你）如何从网页中用于索引或是分析的菜谱及其评论信息中解析结构化数据。示例8-3显示了如何将示例8-1中信息解析成hRecipe格式的数据。

警告：虽然微格式的细则已经规定得很完善了，但实际微格式的实施却有微妙的不同。下面代码只给出了一个并不能健壮地满足全部规则的解析网页的初步样例。如果你想要了解更为健壮的微格式的解决方案，可以查看2013年年初在GitHub上使用Node.js实现的微格式解析（http://bit.ly/1a1p3GL）。

示例8-3：从网页中提取hRecipe数据

```
   import sys
import requests
import json
import BeautifulSoup
# Pass in a URL containing hRecipe...
URL = 'http://britishfood.about.com/od/recipeindex/r/applepie.htm'
# Parse out some of the pertinent information for a recipe.
# See http://microformats.org/wiki/hrecipe.
def parse_hrecipe(url):
    req = requests.get(URL)
    soup = BeautifulSoup.BeautifulSoup(req.text)
    hrecipe = soup.find(True, 'hrecipe')
    if hrecipe and len(hrecipe) > 1:
        fn = hrecipe.find(True, 'fn').string
        author = hrecipe.find(True, 'author').find(text=True)
        ingredients = [i.string
                       for i in hrecipe.findAll(True, 'ingredient')
                           if i.string is not None]
        instructions = []
```

```
        for i in hrecipe.find(True, 'instructions'):
            if type(i) == BeautifulSoup.Tag:
                s = ''.join(i.findAll(text=True)).strip()
            elif type(i) == BeautifulSoup.NavigableString:
                s = i.string.strip()
            else:
                continue
            if s != '':
                instructions += [s]
        return {
            'name': fn,
            'author': author,
            'ingredients': ingredients,
            'instructions': instructions,
            }
    else:
        return {}
recipe = parse_hrecipe(URL)
print json.dumps(recipe, indent=4)
```

比如，对于一个广受欢迎的苹果派的菜谱的样例URL，你可以得到下面的（截取了一部分的）结果：

```
   {
    "instructions": [
        "Method",
        "Place the flour, butter and salt into a large clean bowl...",
        "The dough can also be made in a food processor by mixing the flour...",
        "Heat the oven to 425&#176;F/220&#176;C/gas 7.",
        "Meanwhile simmer the apples with the lemon juice and water..."
    ],
    "ingredients": [
        "Pastry",
        "7 oz/200g all purpose/plain flour",
        "pinch of salt",
        "1 stick/ 110g butter, cubed or an equal mix of butter and lard",
        "2-3 tbsp cold water",
        "Filling",
        "1 &#189; lbs/700g cooking apples, peeled, cored and quartered",
        "2 tbsp lemon juice",
        "&#189; cup/ 100g sugar",
        "4 - 6 tbsp cold water",
        "1 level tsp ground cinnamon ",
        "&#188; stick/25g butter",
        "Milk to glaze"
    ],
    "name": "\t\t\t\t\t\t\tTraditional Apple Pie Recipe\t\t\t\t\t",
    "author": "Elaine Lemm"
}
```

除了时间和空间因素，食物可能是下一个使人们聚集在一起的最重

要的因素，此外，进一步研究社交分析和分析与人、食物、时间、空间关联的数据将会是十分有趣的，也很有意义。例如，你可能会通过分析同一菜谱的各种变化，来查看一些特定成分的含量和菜谱的评分之间是否有关联。然后，基于此信息，更好地锁定可以为其推荐商品和服务的目标群体，或是假设约会的成功与恰当的选择第一餐有很大关联，那么这些信息甚至可以为约会网站建立原型。

通过观察不同的人的苹果派菜谱，我们可以确定菜谱中哪些原料更为常见而哪些是不常见的。那么，你能把这些成分与否出现与特定的地理位置联系起来吗？是不是英国苹果派与北美的苹果派中都各自有一些特殊成分呢？你又会怎样利用这种饮食喜好以及地理信息来给人们配对相亲呢？

下一节介绍了对像我们之前讨论过的线上虚拟相亲服务的另一种想法。

8.2.2.1 检索菜谱评论

本节通过简要介绍hReview-aggregate（http://bit.ly/1a1p3Xa）（一种hReview微格式的变体，能够显示机器易解析的结构化数据的综合评价）来总结所有关于微格式的简短的研究。About.com的菜谱采用hReview-aggregate方式，评论能够优化查询结果，使网页的用户也有更好的用户体验。示例8-4展示了如何提取hReview信息。

示例8-4：解析一个菜谱的hReview-aggregate微格式数据

```
   import requests
import json
from BeautifulSoup import BeautifulSoup
# Pass in a URL that contains hReview-aggregate info...
URL = 'http://britishfood.about.com/od/recipeindex/r/applepie.htm'
def parse_hreview_aggregate(url, item_type):
    req = requests.get(URL)
    soup = BeautifulSoup(req.text)
    # Find the hRecipe or whatever other kind of parent item encapsulates
    # the hReview (a required field).
    item_element = soup.find(True, item_type)
    item = item_element.find(True, 'item').find(True, 'fn').text
    # And now parse out the hReview
    hreview = soup.find(True, 'hreview-aggregate')
    # Required field
    rating = hreview.find(True, 'rating').find(True, 'value-title')['title']
    # Optional fields
    try:
        count = hreview.find(True, 'count').text
    except AttributeError: # optional
        count = None
    try:
        votes = hreview.find(True, 'votes').text
    except AttributeError: # optional
        votes = None
    try:
        summary = hreview.find(True, 'summary').text
    except AttributeError: # optional
        summary = None
    return {
        'item': item,
        'rating': rating,
        'count': count,
        'votes': votes,
        'summary' : summary
    }
# Find hReview aggregate information for an hRecipe
reviews = parse_hreview_aggregate(URL, 'hrecipe')
print json.dumps(reviews, indent=4)
```

下面是截取的示例8-4的部分结果：

```
   {
    "count": "7",
    "item": "Traditional Apple Pie Recipe",
    "votes": null,
    "summary": null,
    "rating": "4"
}
```

当你把怪才和食物数据联合起来，你会发现很多有趣的事。令人赞不绝口的畅销书《Cooking for Geeks》（O'Reilly）（http://bit.ly/1a1p519）就是很好的证明。因为美食网站的不断发展提供了额外的API，我们就可以在这个领域中看到更多的创新。图8-4显示了hReview-aggregate应用的样例网页及其网页源代码的截图。

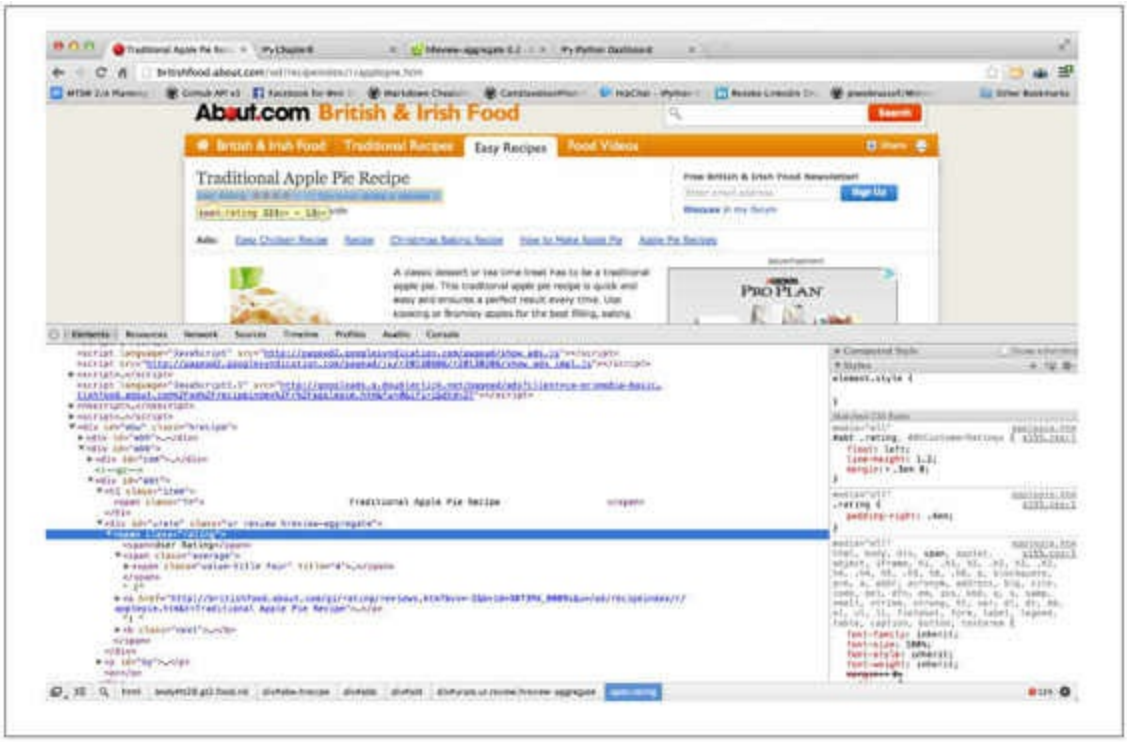


图8-4：如果你对微格式应用的（常用）细节感兴趣的话，你可以查看网页源代码

注意：现在大多数浏览器都本地支持CSS查询选择器（http://bit.ly/1a1p5hG），你还可以使用document.querySelectorAll在开发者控制台中查看JavaScript形式的微格式。例如，运行document.querySelectorAll（".hrecipe"）来查询使用hRecipe类的全部节

点。

8.2.3 使用LinkedIn的2亿在线简历

LinkedIn面向它的2亿用户实施hResume（http://bit.ly/1a1oFIf）（建立在hCard微格式和hCalendar微格式的基础上），本节举个搜索引擎和其他机器如何将富数据识别为结构化数据的简单例子。hResume是信息很丰富的，因为你可以从中找到嵌入hCalendar和hCard信息构成的联系方式、职业经历、教育、隶属机构、出版物等信息。

鉴于LinkedIn的应用相当广泛，在这里，我们难以写出一个应用于所有一般性案例的健壮微格式解析器。因此，在这短短的一节中，我们只介绍一个Python解析器——Google的结构化数据测试工具（http://bit.ly/1a1p5yl）。Google工具可以插入到任意一个URL中，它将提取语义标记，并为你预览搜索结果。鉴于网页上的信息数据量大，结果可能是很广泛的。比如，相对完整的LinkedIn资料产生多屏结果，这些结果能够被之前提到的所有领域解析。可以确定的一点是，LinkedIn的hResume应用是彻底的，正如图8-5所示。如果你想挑战这一部分，可以先看看这个针对之前给出的样例代码的解析hResume数据的例子。

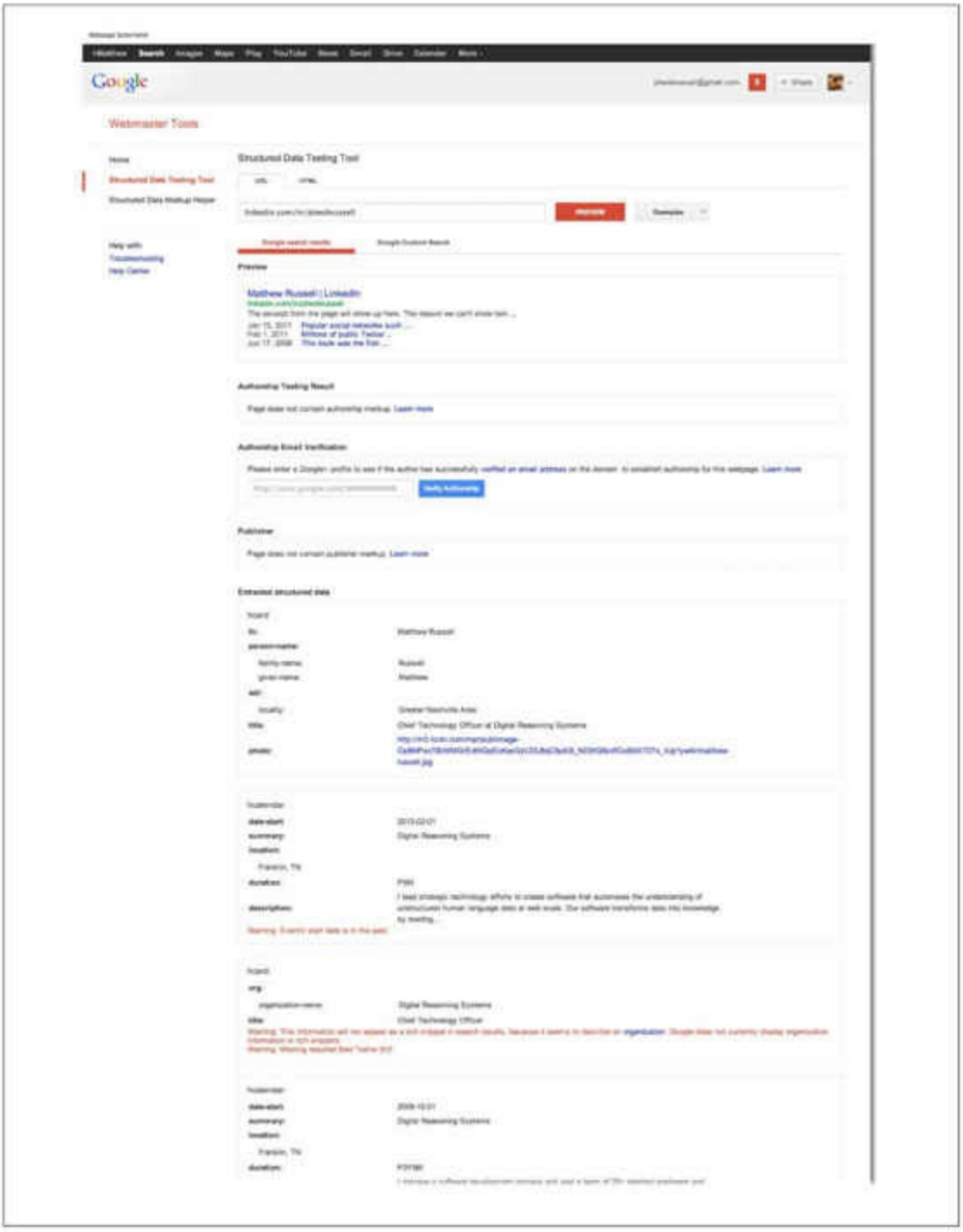


图8-5：Google的结构化数据测试工具提取网页中的语义标记的样例结果

注意：你可以在任意的URL上使用Google的结构化数据测试工具（http://bit.ly/1a1p5yl）来查看隐藏的结构化数据。事实上，这将是更好地帮助你debug编译器的第一步，因为这正是这个工具的一个潜在目的。

8.3 从语义标记过渡到语义网：一个小插曲

为了更好地过渡到语义网，我们要回顾一下之前的讨论，先来看看到目前为止我们都学到了什么。首先，我们了解了一些致力于使机器为了一些常见事物（如：简历、食谱、地理位置坐标）提取网页中的结构化内容的项目。撰写这些内容的作者可以为搜索引擎提供用来提高相关性排名的机器可读数据，并为用户显示更多信息，否则目前来看，它们不能有效地满足用户。这一句强调“目前”很重要，因为实际上我们更为宏大的语义网可能是并无根据地定义的，这样的话，跟所有个体都企图达到的是完全不同的。

实际上，微格式的全部限制是Web前进的重要一步，撰写内容的作者以及消费者都从中受益，同时他们的关注也很可能引起Web的进一步发展。

虽然长远看来，此方式的扩展性让人质疑，但作为当前网络的一个相关目标，我们应当感谢那些公司政客和开放网络的拥护者能够为了网络持续健康发展对此一同努力。然而，你可能会对这样的想法不满意，为了使机器更好地理解网页中的元数据，未来网络可能仅仅依靠小团队内容供应商来发布它们（或者编写精细的脚本来发布页面中的元数据）。我们也许要花上很多年才会看到它的成果，但要记住，像前面章

节提到的自然语言处理（NLP）等技术一样，它将会受到学术界和工业界的越来越多的关注。

最后，在技术的不断进化的过程中，我们可以更好地理解人类语言，终有一天我们会实现机器人深层次地读懂人类语言的网络。与此同时，为了网络持续发展改善，任何形式的努力都是十分必要的。我们期待的网络中机器人可以理解非结构化元数据描述的人类语言，并且能够有效将其翻译成结构化的数据。然而，这样的网络目前并没有。因此，我们可以断定，鉴于机器自动理解自然语言对人类生活各方面的巨大影响，它是目前最值得研究的问题。

8.4 语义网：发展中的变革

不同人对语义网有不同的理解，因此我们首先来解释一下这个词。由于Web是关于共享信息的，而且语义网的实际定义是“有导致某种行为的足够的解释”[1]，那么我们似乎可以断言语义网主要是以某种有意义的方式表示知识，但这些信息又由谁来解读呢？首先先假设排除人类使用它表示的信息，我们考虑这样一种可能性，如果信息以机器可理解的方式共享——一种不太明确的方式，像机器人等用户代理者可以从中提取、翻译并使用这些信息来做决策。

在这个方向上我们已经学习了很多：比如，我们在前面几节讨论过微格式如何使限制内容实现这种可能的，然后在第2章中我们学习了Facebook如何利用开放图协议（详见2.2.2节）向Web中引入了一个明晰的图结构。为了更好地理解这部分，我们有必要回顾一下前面的知识。

Internet[2]仅仅是网络的网络，从技术角度上，它吸引我们的是建立在低级协议之上的高层次的协议最终是如何实现容错的全球计算基础结构的。在线上活动中，我们每天依赖于各种各样的协议，却从来没深入地思考过这些协议。然而，有一个我们很难不去考虑的常见的协议，也就是HTTP，你需要输入到浏览器的每个URL的前缀，支持超文本文档（HTML页面）及其聚集形成Web的各个链接的协议。但正如你长期以

来了解的那样，Web不仅仅是超文本，它还包括各种各样的嵌入式技术，比如JavaScript、Flash，还有新兴的HTML5的特质（如音频流、视频流）。

诚然，我们对通过现代浏览器（包括移动或平板设备中的浏览器）在HTTP上交互的文档、平台、应用的虚拟网络世界的定义是模糊的，但大多数人把这当作Web。在一定程度上来说，2004年兴起的Web 2.0的想法的背后动机就是更准确的定义什么是Web以及它将会变成什么样子。沿着这条思路，有的人认为Web从开始出现，到高度交互的Web应用程序和用户协作的Web 1.0，到富互联网应用（Rich Internet Application，RIA）和协作的Web 2.x，再到尚未到来的语义时代的Web 3.0（详见表8-2）。

目前，对于Web 3.0的定义还没有真正的共识，但大多数的讨论包括“语义网”一词以及目前网络尚未实现的机器识别和使用信息的概念。机器仍旧很难提取并推断网络中文档包含的事实。关键字查询和启发式学习可以提供相关的查询结果列表，但仍需人类智慧来翻译这些文档中的信息并将其汇总。Web 3.0和语义网是否相同仍待争论，然而，我们通常认为语义网很像我们所喜爱且熟知的网络，但同时也演变成了机器可以以粒度级别提取和使用文档中信息的网络。

在这方面，我们可以回顾微格式的发展过程，从而了解发展过程如何取得革命性的进步。

表8-2：各种各样的Web阶段及其特征

显示/时代	特征
Internet	应用程序协议，如SMTP、FTP、BitTorrent、HTTP等
Web 1.0	多数静态HTML网页和超链接
Web 2.0	平台、协作、富用户体验

表8-2：各种各样的Web阶段及其特征（续）

显示/时代	特征
社交网 (Web 2.x)	人和他们虚拟和现实交互的联系和活动
语义标记网 (Web 2.x)	越来越多的机器可读内容，如微格式、RDFa和microdata
Web 3.0 (语义网)	机器可理解的大量内容

8.4.1 人不可能只靠事实生活

语义网表示知识的基本构造称作三元组（triple），它是一种非常直观且自然地描述事实的方式。比如，我们之前多次考虑过的一个句子——“在书房里格林先生用烛台杀了马斯塔德上校”（“Mr.Green killed Colonel Mustard in the study with the candlestick”）——被表示为三元组（Mr.Green，killed，Colonel Mustard），其中的元素是句子中的主语、谓语和宾语。

资源描述框架（RDF）是语义网用于定义和交换三元组的模型。RDF有高度的可扩展性，因为它提供了表示知识的基础，同时也可用来定义称作本体（ontology）的专有词汇，它为特定领域建模提供了精确

语义。

我们不只是简单了解下像RDF（http://bit.ly/1a1p6lR）、RDFa（http://bit.ly/1a1lujR）、资源描述框架计划（RDF Schema）（http://bit.ly/1a1p8Ky）等语义网技术，在最后提到的OWL（http://bit.ly/1a1p9Os）会深入的多，我们还将研究一个高级示例，它说明了语义网的一些相关内容。

在你阅读这节剩余的部分时，切记资源描述框架（RDF）仅是表示知识的一种方式。它可能由大量人力通过网页间交互有价值的元数据建立，也可能由能够处理自然语言并自动从人类语言中提取元组信息的少量机器人群体（如Web代理）执行。无论是哪种方式，RDF都提供了知识建模的基础，并作为可被推理得到答案的图表。在下一节，我们将用一些代码来具体解释。

开放世界假说与封闭世界假说

在像Prolog[3]等逻辑编程语言中的推理方式和像RDF栈这类与之相对的其他的技术之间的一个有趣的区别就是，它们是否作出关于宇宙的开放世界或是封闭世界的假说。像Prolog等逻辑编程语言和多数传统的数据库系统假设封闭的世界，而另一边，RDF技术一般假设开放的世界。

在一个封闭的世界里，任何不能清楚地解释关于宇宙的说法的都是

错误的，而在开放的世界里，你所不知道的一切都被当做是未定义的（“未知”的另一种说法）。区别是，假设开放世界的推理不会排除没有明确在知识库中表示的事实，而对于假设封闭世界的推理会排除这样的事实。此外，在假设封闭世界的系统中，合并矛盾的知识通常会导致错误，而开放世界的系统会尝试产生新推论，从而在某种程度上调和矛盾信息。你可以想象得到，开放世界系统更加灵活，同时也可能会导致一些难题，尤其是当不同知识库合并的时候。

你可能会本能地这样想：基于封闭世界的推理会假设它们给出的数据都是完整的，当有新事实加入的时候，并不是每一个之前的事实都可以保持不变。相反，开放的世界系统不做这样关于数据完整性和不变性的假设。正如你可能会想到的，这会引起比较两个假设价值的争论。如果你对语义网感兴趣，你至少要了解这一点。因为这个问题与资源描述框架（RDF）紧密相关，W3C文档官方指导这样说[4]：

为了便于互联网范围内的操作，资源描述框架（RDF）作为一个开放世界的框架，允许任何人描述任何资源。一般来说，它并不假设可以得到所有资源的完整信息。RDF并不阻止任何人作出无意义或者与其他陈述，或当前人们看到的世界不一样的断言。使用RDF的应用程序的设计者要了解这一点，并能容忍不完整或不一致的信息来源。

如果你对这个主题感兴趣的话，你还可以查看下Peter Patel-Schneider和Ian Horrocks的“Position Paper：A Comparison of Two

Modelling Paradigms in the Semantic Web”（http://bit.ly/1a1p9yb），无论你是否决定深入研究这个主题，都要记住：网络上可以利用的数据都是不完整的，假设封闭世界（也就是说，把所有未知信息当做完全错误的）迟早会导致严重的后果。

8.4.2 推断开放世界

设计像RDF Schema和OWL等基础语言的目的是使用精确的词汇，以机器可读的方式来表达类似三元组（Mr.Green，killed，Colonel Mustard）等事实，而且这是语义网实现的必要（非充分）条件。一般来说，一旦已有一些事实存在，下一步就是执行对事实的推理，并得出基于这些事实的结论。形式推理的概念最早追溯到古希腊亚里士多德的三段论，并且在过去的50年左右的时间里，对人工智能感兴趣的研究者们一直积极关注着机器如何利用这一点。在Java领域，充斥着Jena（http://bit.ly/1a1paCa）和Sesame（http://bit.ly/1a1pcKf）这类企业级工具，它们似乎是多数重量级举措的选择。但幸运的是，在使用Python的领域，我们也有些可靠的选择。

你可能会遇到的具有推理能力的支持Python的最佳选择是FuXi（http://bit.ly/1a1pb96），它是语义网络中一种强大的逻辑推理系统。它使用转发链技术（forward chaining）（http://bit.ly/1a1pdh9）从现

有信息中推导新信息，它从事实出发，通过应用一系列逻辑法则，从已有的事实中挖掘新事实，重复这样的过程直到某一结论被证实或是反驳，或是无法推断出更多新事实来。这种技术是可靠且完整（因为任何一个新产生的事实都是正确的，任何一个正确的事实最终都会被证明）。对这部分命题的成熟讨论和一阶逻辑很容易就能写满一本书，因此，如果你对此感兴趣，可参考Stuart Russell和Peter Norvig编写的经典教材《Artificial Intelligence：A Modern Approach》（http://amzn.to/1a1pdhg）。

为了证明像FuXi一类的系统的推理能力类型，让我们来看个著名的示例——亚里士多德的三段论[5]，首先我们给出一些包括“苏格拉底是人类”和“所有人都是凡人”这样的事实的知识库，由此可以推断“苏格拉底是凡人”。虽然这问题看起来很简单，但请记住，当有更多可用的重要的事实且这些事实再产生新事实的时候，产生新事实“苏格拉底是凡人”的确定性算法也会以相同的方式运作。我们考虑一个稍微复杂一点的知识库，它包含以下几个额外事实：

·苏格拉底是人类。

·所有人都是凡人。

·只有上帝住在奥林匹斯山上。

·所有凡人喝威士忌。

·查克·诺理斯住在奥林匹斯山。

如果我们给出以上的知识库，然后提问“苏格拉底喝威士忌吗？”。你一定会在得到最终答案前先推断事实：你会先推理“苏格拉底是凡人”，然后才能断定事实“苏格拉底喝威士忌”。为了说明这些是如何在代码中运作的，你可以看看使用N3（http://bit.ly/1a1pdxO）的例子，N3是一个简单却十分强大的语义，它描述了RDF中的事实和规则，如下所示：

```
   #Assign a namespace for logic predicates
@prefix log: <http://www.w3.org/2000/10/swap/log#> .
#Assign a namespace for the vocabulary defined in this document
@prefix : <MiningTheSocialWeb#> .
#Socrates is a man
:Socrates a :Man.
@forAll :x .
#All men are mortal: Man(x) => Mortal(x)
{ :x a :Man } log:implies { :x a :Mortal } .
#Only gods live at Mt Olympus: Lives(x, MtOlympus) <=> God(x)
{ :x :lives :MtOlympus } log:implies { :x a :god } .
{ :x a :god } log:implies { :x :lives :MtOlympus } .
#All mortals drink whisky: Mortal(x) => Drinks(x, whisky)
{ :x a :Man } log:implies { :x :drinks :whisky } .
#Chuck Norris lives at Mt Olympus: Lives(ChuckNorris, MtOlympus)
:ChuckNorris :lives :MtOlympus .
```

虽然有很多表达RDF的不同的形式，许多语义网工具选择N3，因为它的可读性和可利用性使它容易理解。快速浏览文件，我们会看到许多为了给标记打基础而建的命名空间和一些之前提到的断言。让我们来看看当从命令行运行FuXi时会发生什么，然后区分翻译来自之前介绍过的样例知识库的事实，以及使用下面的命令累计额外的事实。

```
   $ FuXi --rules=chuck-norris.n3 --ruleFacts --naive
```

注意：如果在你自己的机器上第一次安装FuXi，最简单且快速的选择是采取这些步骤（http://bit.ly/1a1pcd2）。当然，如果你按照这本书中虚拟机部分的IPython Notebook来操作，这种安装方式（像其他方式一样）也考虑到了这一点，并且IPython Notebook中的相应样例代码也同样有效。

如果你在命令行中运行叫chuck-norris.n3的文件，它包含之前N3知识库，你会看到与下面相似的输出：

```
   ('Parsing RDF facts from ', 'chuck-norris.n3')
('Time to calculate closure on working memory: ', '1.66392326355 milli seconds')
<Network: 3 rules, 6 nodes, 3 tokens in working memory, 3 inferred tokens>
@prefix : <file:///.../ipynb/resources/ch08-semanticweb/MiningTheSocialWeb#> .
@prefix iw: <http://inferenceweb.stanford.edu/2004/07/iw.owl#> .
@prefix log: <http://www.w3.org/2000/10/swap/log#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix skolem: <http://code.google.com/p/python-dlp/wiki/SkolemTerm#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
:ChuckNorris a :god .
:Socrates a :Mortal ;
    :drinks :whisky .
```

这段程序的输出可以看出之前知识库中没有清楚地给出的一些信息：

·查克·诺理斯是上帝。

·苏格拉底是凡人。

·苏格拉底喝威士忌。

虽然对于我们来说，推导出这些事实很容易，但对于机器来说却很困难——这也是这个问题令人振奋的一点。同样切记，为了从原始知识库中根据逻辑推断出来结果，这些已有的或是可轻而易举地推断的事实不需要在现实世界中有任何真正的意义。

警告：对查克·诺理斯（甚至是在包含科幻世界的教育性内容中）不合理的言论可能会对你的计算机健康有影响，甚至可能对你个人有影响。[6]

如果你对这个简单的例子感兴趣，可以深入了解FuXi和语义网的潜力。这个例子只是涉及了它们表面上的能力。你还会遇到用于数据挖掘和其他技术工具链的大量数据集。语义网是比社交网更为复杂先进的技术，因此对它的进一步研究是十分有意义的——尤其是你对逻辑推断引发的社交数据的各种可能性十分感兴趣。未来社交数据和包含它的其他技术的革新都将加强语义网的潜能，然而，我们的努力决定了对语义网探索的终点。

[1] 在《Programming the Semantic Web》（O'Reilly）中定义的。

[2] 互联网意味着"共有或合作网络"。

[3] 我们非常鼓励你试试Prolog这类真正的基于逻辑的编程语言，因为它是在专门设计的规范中编写的，这样你就可以表示知识，并从现有的事实中推断新信息了。GNU　Prolog（http://bit.ly/1a1pa53）是个良好的学习起点。

[4] 详见http://www.w3.org/TR/rdf-concepts/。

[5] 现代的说法中，三段论更普遍被称为一个“含义”。

[6] 见http://www.chucknorrisfacts.com。

8.5 本章小结

本章试图为你提供一盘关于“语义网”这一话题的开胃菜。我们提到了一些Web的历史知识，各种版本的语义网，以及它所涉及的周边技术中的竞争哲学。本章仅是一个大的框架，目的在于激发你的兴趣，并为你未来的研究和创业介绍一些基础知识。

如果你并没有从本章学到什么，也请记住微格式是用元数据装饰标记的一种方式，它能显示一些特定类型的结构化信息，如食谱、联系人信息和其他的一些常见数据种类。因为微格式能够使内容出版商把现有的内容转化成可被机器理解并推理的且满足共享标准的数据，它推动了语义网的发展。我们期望看到语义网显著的逐年累月的发展。像微格式和IndieWeb等团体的一些项目是十分典型的，也是值得我们效仿的。

虽然由于本章前面故意忽略了一些对语义网的讨论，可能给你留下了社会网和语义网之间具有严格明确的分别的印象，但实际上这一差别是相当模糊的，也是不断变化的。Web上大量的社交数据的集合，从Wikipedia到About.com、LinkedIn、Facebook的开放图协议和规定开源标准的群体等一系列机构发布的微格式的项目，以及数据黑客的努力共同促进了不同于之前20年炒作的语义网的实现。

大量的社交数据有能力成为语义网发展的催化剂，语义网将作为代

理人代表他们作重要决策。我们虽未实现Epicurus的至理名言，却十分认可他的话，他曾说过“不要让你没得到的东西破坏你已经拥有的，但记住你现在拥有的也是你曾经期待的”。

注意：本章中的以及其他章的源代码都可在GitHub中以较为方便的IPython Notebook形式查看，我们强烈建议你用浏览器实践这些代码。

8.6 推荐的练习

·参与#微格式IRC（Internet Relay Chat，http://bit.ly/1a1phNZ）频道活动并成为社区会员。

·看一些微格式社区领袖Tantek elik（http://bit.ly/1a1pgcS）介绍的关于微格式的概述的优秀视频幻灯片（http://bit.ly/1a1pi4G）。

·回顾wiki微格式（microformats wiki）上的例子（http://bit.ly/1a1oKLV）并考虑如何实践并为此做些贡献。

·探索一些实施微格式的网站，并用开发人员控制台和内置document.querySelector All功能查看HTML源码。

·回顾并跟着一些全面的FuXi在线教程学习（http://bit.ly/1a1pgtA）。

·回顾Schema.org的起步教程（http://bit.ly/1a1piSe）。

·回顾RelMeAuth（http://microformats.org/wiki/RelMeAuth）并尝试使用网络登录服务（http://bit.ly/1a1oUDd）。你会发现Python基础的GitHub的relmeauth项目（http://bit.ly/1a1pkJB）作为起步学习是很有帮助的。

·看一看网页支付的开源案例PaySwarm（http://mzl.la/1a1pl0h）（虽然有点过时，但却是十分重要的一个包含钱的案例，它本身就是一个社交实体）。

·做一个语义网（http://bit.ly 1a1pn8o.htm）作为一个独立的研究项目的调研。

·看一下DBPedia（http://bit.ly/1a1plNG），它能够从Wikipedia中提取结构化数据，从而实现语义推理的目的。想想你能用FuXi和DBPedia数据做些什么呢？

8.7 在线资源

如下本章链接列表以备复习之用：

·《人工智能：现代的方法》（Artificial Intelligence：A Modern Approach）（http://bit.ly/1a1pdhg?cc=baf4541d780e287ff5052d53372c1521）

·共享的爬取语料库（Common Crawl corpus）（http://bit.ly/1a1oM6A）

·CSS查询选择器（CSS query selectors）（http://bit.ly/1a1p5hG）

·DBPedia（http://bit.ly/1a1plNG）

·火狐浏览器插件（Firefox Operator add-on）（http://bit.ly/1a1p2Cy?cc=96f925d9a7cfbafe1b30c2ee9974e86e）

·转发链逻辑算法（Forward chaining logic algorithm）（http://bit.ly/1a1pdh9）

·FuXi（http://bit.ly/1a1pb96）

·FuXi教程（http://bit.ly/1a1pgtA）

·GNU Prolog（http://bit.ly/1a1pa53）

·Google的结构化数据测试工具（Google’s Structured Data Testing Tool）（http://bit.ly/1a1p5yl）

·HTML 4.01（http://bit.ly/1a1oKvt）

·IndieWeb（http://bit.ly/1a1oWuT）

·KML（http://bit.ly/1a1p1Pb）

·Microdata（http://bit.ly/1a1oSeo）

·microform.at（http://bit.ly/1a1oZGX）

·Tantek Celik’s“microformats2&HTML5”presentation（http://bit.ly/1a1pi4G）

·microformats.org（http://bit.ly/1a1oEEd）

·Notation3（N3）（http://bit.ly/1a1pdxO）

·开放图协议（Open Graph protocol）（http://bit.ly/1a1lu3m）

·OWL（http://bit.ly/1a1p9Os）

·PaySwarm（http://mzl.la/1a1pl0h）

·“Position Paper：A Comparison of Two Modelling Paradigms in the Semantic Web”（http://bit.ly/1a1p9yb）

·RDF（http://bit.ly/1a1p6lR）

·RDFa（http://bit.ly/1a1oQmR）

·RDF Schema（http://bit.ly/1a1p8Ky）

·RelMeAuth（http://bit.ly/1a1oUD4）

·Schema.org（http://bit.ly/1a1oEnv）

·语义网（Semantic web）（http://bit.ly/1a1pn8o）

·Web Data Commons（http://bit.ly/1a1oJI1）

·网络登录服务（Web sign-in）（http://bit.ly/1a1oUDd）

·XFN（http://bit.ly/1a1oNaG）

第二部分　Twitter实用指南

本书的第一部分针对精选的一些社交网络资源给出了一个宽泛的导引，转了一圈之后，我们重回到第一部分的Twitter。本部分组织成实用指南，并提供超过20个供挖掘Twitter数据之用的简短代码方法（recipe）。Twitter因其API、与生俱来的开放性以及世界范围内的流行度，而成为我们关注的理想社交网站，而本书此部分意在创造一些具有高度适应能力的基本构建块，服务于多种目的。

为方便查阅，就像任何其他的技术指南，这些代码配方被组织成先提出问题再给出解决方案的结构。在学习它们的过程中，你肯定能想到调整和修改它们的有趣思路。本书第一部分提供了许多社交网络资源的宽泛概述，而第二部分旨在把关注点聚焦到常见的一组小问题，并提出一些移植到其他社交网络资源的模式。

我们强烈鼓励你好好利用这些代码配方，从中获得尽可能多的乐趣。当你提出属于你自己的任何更好代码配方时，考虑通过向其GitHub库（http://bit.ly/1a1kNqy）发送拉取请求分享回本书社区、发布相关推文（如果你想转推，请提及@SocialWebMining），或发布帖子到本书的Facebook页面（http://on.fb.me/1a1kHPQ）。

第9章　Twitter实用指南

这个实用指南是关于挖掘Twitter数据的代码方法的。每种解决特定问题的配方在设计时都尽可能简单，这样多种配方就能方便地组合起来，以实现更加复杂的功能。可以将每种配方想象为一块积木，尽管本身就很有用，但和其他积木组合起来会实现更加复杂的分析单元。和前面讲解内容多于代码内容的章节不同，这个指南并没有提供很多的讲解，而是用代码来说话。这一章的思想是，你可以用多种方式操作和组合代码以达到特定的目标。

尽管大多数方法都是发起参数化的API调用并将返回结果预处理为方便操作的格式，但有些方法很简单（只有短短的几行代码），也有相当复杂的其他方法。这个实用指南会提供一些常见的问题和对应解决方案来帮助你。在有些情况下，你想要的数据真的只需要几行代码来实现可能并不常见。所提供这些代码的价值在于，你就能做些简单的改造，满足自己的需求。

本章的所有方法都需要依赖一个基本的包——twitter包，你可以使用终端的pip install twitter命令来安装。其他软件依赖会在单独的配方中进行介绍。如果你好好利用了本书的虚拟机（强烈推荐），twitter包和其他依赖的包将会预安装好。

正如你从第1章学到的那样，Twitter 1.1版本的API中的所有请求都需要授权，所以每个配方都假定你已经利用了9.1节或9.2节的知识，首先来获得授权的API连接器以供其他配方使用。

注意：本章（和其他章节）最新的bug修复的源代码在http://bit.ly/MiningTheSocialWeb2E中可以在线查看。确保你已经利用了本书附录A中介绍的虚拟机体验，来最大化示例代码的乐趣。

9.1 访问Twitter的API（开发目的）

9.1.1 问题

你想要挖掘你自己的账号的数据，或者以开发为目的使用快速简单的API访问数据。

9.1.2 解决方案

使用twitter包和应用设置中提供的OAuth 1.0a证书来获得API访问你自己数据的权限，不需要HTTP的重定向。

9.1.3 讨论

Twitter实现了OAuth 1.0a（http://bit.ly/1a1pBwg），这是明确设计出的授权机制，这样用户可以授权第三方访问他们的数据而不需要做想象不到的事——提供用户名和密码。你当然可以利用Twitter的OAuth实现，你需要用户授权你的应用访问他们的账号，你也可以使用你的应用设置中的证书来立即以开发目的访问数据或者挖掘你自己账号的数据。

使用你的Twitter账号在http://dev.twitter.com/apps注册应用，记下用户账号、用户密码、访问令牌和访问令牌密码，这些构成了任意OAuth 1.0a的四个证书——让应用最终能够访问账户。图9-1是Twitter应用设置界面的截图。有了这些证书，你可以使用任意OAuth 1.0a库来访问Twitter的支持REST的API（http://bit.ly/1a1pDEq），但是我们选择使用twitter包，它提供了抽象的Python的对Twitter支持REST的API接口的封装。当注册应用时，你不需要明确的回调地址，因为我们有效的避开了完整的OAuth流程，并且简单地使用证书来实现对API的访问。示例9-1展示了如何使用这些证书来建立对API的连接器的例子。

示例9-1：以开发目的来访问Twitter的API

```
   importtwitter
def oauth_login():
    # XXX: Go to http://twitter.com/apps/new to create an app and get values
    # for these credentials that you'll need to provide in place of these
    # empty string values that are defined as placeholders.
    # See https://dev.twitter.com/docs/auth/oauth for more information
    # on Twitter's OAuth implementation.
    CONSUMER_KEY = ''
    CONSUMER_SECRET = ''
    OAUTH_TOKEN = ''
    OAUTH_TOKEN_SECRET = ''
    auth= twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                              CONSUMER_KEY, CONSUMER_SECRET)
    twitter_api = twitter.Twitter(auth=auth)
    returntwitter_api
# Sample usage
twitter_api = oauth_login()
# Nothing to see by displaying twitter_api except that it's now a
# defined variable
print twitter_api
```

警告：记住用来建立连接的证书像用户名、密码一样是有效的，所以仔细的保管好并在应用设置中指定访问需要的最低要求。只读的访问

对于挖掘你自己账号的数据已经足够了。

尽管访问自己的数据很方便，但这种捷径对于写一个客户端来访问别人的数据来说并没有用。这种情况下，你需要进行完整的OAuth流程，正如示例9-2中展示的那样。

9.2 使用OAuth访问Twitter的API（产品目的）

9.2.1 问题

你想使用OAuth使你的应用能够访问别的用户的数据。

9.2.2 解决方案

使用twitter包实现“OAuth流程”。

9.2.3 讨论

twitter包提供了所谓的OAuth舞蹈的内部实现，是为控制台应用程序服务的。对于不在浏览器中运行的应用，它通过实现带外的（Out of Band，OoB）OAuth流程（比如Python程序），能够安全地获得这四个证书来访问API并允许你简单地请求访问特定用户的数据，作为标准的“带外的”功能。然而，如果你想要写一个网页应用来访问其他用户的数据，你需要简单地改造这种实现方式。

虽然真正从IPython Notebook实现一个OAuth舞蹈并没有很多实用的

理由（除非也许你在运行被别人使用的宿主的IPython Notebook服务），这种方法使用Flask作为嵌入式网站服务器，来展示使用和本书剩余章节相同的工具链的方法。由于概念相同，它很容易改造，以适应任意你选择的网页应用框架。

图9-1是Twitter应用设置页面的截图。在OAuth 1.0a流程中，在9.1节中介绍的用户账号和用户密码的值能够唯一标识你的应用。当请求访问用户数据时，你需要提供这些值，这样Twitter就能提示用户关于你的请求类型的信息。假设用户授权了你的应用，Twitter会重定向到你在应用设置中指定的回调URL并包含OAuth verifier，它需要用来交换访问令牌和访问令牌密码，它们和用户账号和用户密码结合使用，就能最终让你的应用访问到用户的数据。（对于oob OAuth流程，并不需要回调URL；Twitter为用户提供了PIN码作为OAuth verifier，它必须被人为地复制/粘贴回应用中。）可以参考附录B来获取更多的OAuth 1.0a流程的信息。

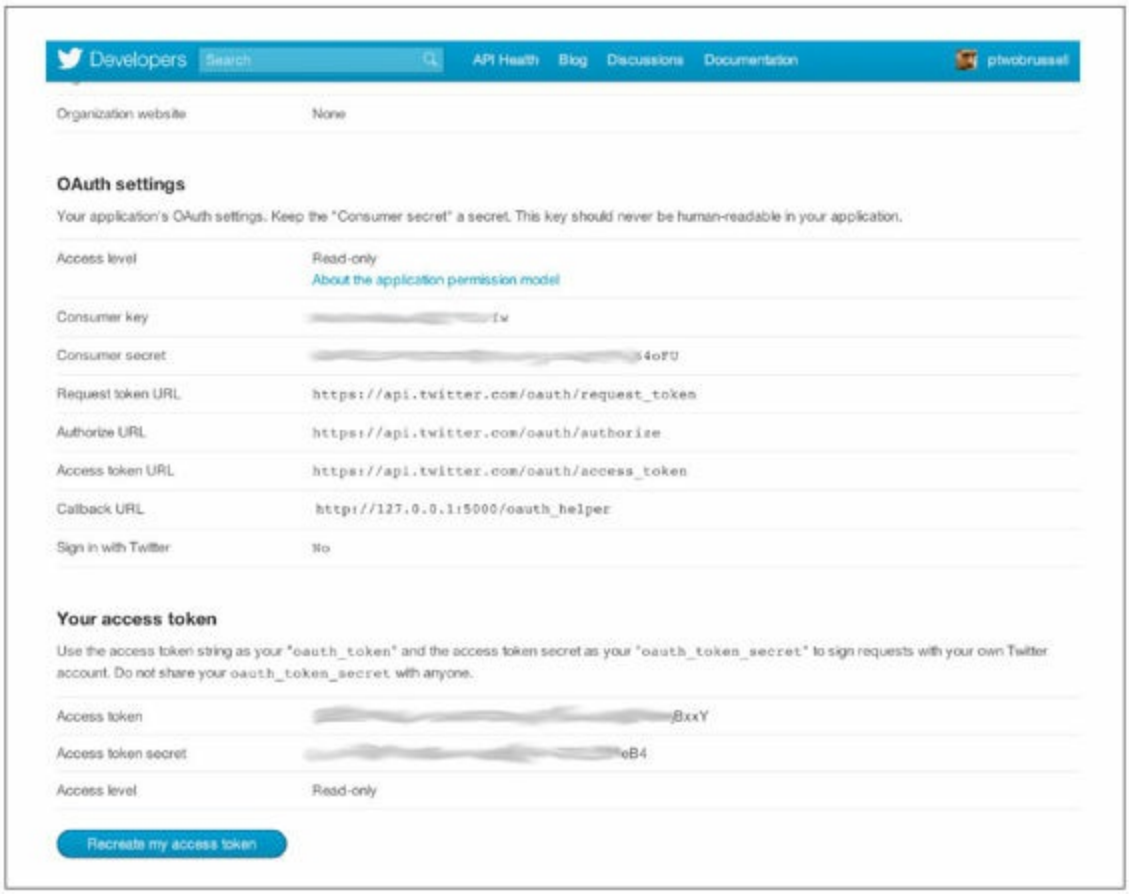


图9-1：Twitter应用的OAuth设置示例

示例9-2展示了如何使用用户账号、用户密码和twitter包通过OAuth来访问用户的数据。访问令牌和访问令牌密码被写在磁盘上，以便简化将来的授权。Twitter的开发问题集（Twitter's Development FAQ）（http://bit.ly/1a1pC3o）中说到，目前Twitter的访问令牌并不会到期，这说明你可以放心大胆地将它们存储起来并用来无限期地代表用户，只要你遵循相应的服务条款（terms of service）（http://bit.ly/1a1pEbj）即可。

示例9-2：以产品为目的通过OAuth来访问Twitter的API

```
   import json
from flask import Flask, request
import multiprocessing
from threading import Timer
from IPython.display import IFrame
from IPython.display import display
from IPython.display import Javascript as JS
import twitter
from twitter.oauth_dance import parse_oauth_tokens
from twitter.oauth import read_token_file, write_token_file
# Note: This code is exactly the flow presented in the _AppendixB notebook
OAUTH_FILE = "resources/ch09-twittercookbook/twitter_oauth"
# XXX: Go to http://twitter.com/apps/new to create an app and get values
# for these credentials that you'll need to provide in place of these
# empty string values that are defined as placeholders.
# See https://dev.twitter.com/docs/auth/oauth for more information
# on Twitter's OAuth implementation, and ensure that *oauth_callback*
# is defined in your application settings as shown next if you are
# using Flask in this IPython Notebook.
# Define a few variables that will bleed into the lexical scope of a couple of
# functions that follow
CONSUMER_KEY = ''
CONSUMER_SECRET = ''
oauth_callback = 'http://127.0.0.1:5000/oauth_helper'
# Set up a callback handler for when Twitter redirects back to us after the user
# authorizes the app
webserver= Flask("TwitterOAuth")
@webserver.route("/oauth_helper")
def oauth_helper():
    oauth_verifier = request.args.get('oauth_verifier')
    # Pick back up credentials from ipynb_oauth_dance
    oauth_token, oauth_token_secret = read_token_file(OAUTH_FILE)
    _twitter = twitter.Twitter(
        auth=twitter.OAuth(
            oauth_token, oauth_token_secret, CONSUMER_KEY, CONSUMER_SECRET),
        format='', api_version=None)
    oauth_token, oauth_token_secret = parse_oauth_tokens(
        _twitter.oauth.access_token(oauth_verifier=oauth_verifier))
    # This web server only needs to service one request, so shut it down
    shutdown_after_request = request.environ.get('werkzeug.server.shutdown')
    shutdown_after_request()
    # Write out the final credentials that can be picked up after the following
    # blocking call to webserver.run().
    write_token_file(OAUTH_FILE, oauth_token, oauth_token_secret)
    return "%s %s written to %s" % (oauth_token, oauth_token_secret, OAUTH_FILE)
# To handle Twitter's OAuth 1.0a implementation, we'll just need to implement a
# custom "oauth dance" and will closely follow the pattern defined in
# twitter.oauth_dance.
def ipynb_oauth_dance():
    _twitter = twitter.Twitter(
        auth=twitter.OAuth('', '', CONSUMER_KEY, CONSUMER_SECRET),
        format='', api_version=None)
    oauth_token, oauth_token_secret = parse_oauth_tokens(
            _twitter.oauth.request_token(oauth_callback=oauth_callback))
    # Need to write these interim values out to a file to pick up on the callback
    # from Twitter that is handled by the web server in /oauth_helper
    write_token_file(OAUTH_FILE, oauth_token, oauth_token_secret)
    oauth_url = ('http://api.twitter.com/oauth/authorize?oauth_token=' + oauth_token)
    # Tap the browser's native capabilities to access the web server through a new
    # window to get user authorization
```

```
    display(JS("window.open('%s')" % oauth_url))
# After the webserver.run() blocking call, start the OAuth Dance that will
# ultimately cause Twitter to redirect a request back to it. Once that request
# is serviced, the web server will shut down and program flow will resume
# with the OAUTH_FILE containing the necessary credentials.
Timer(1, lambda: ipynb_oauth_dance()).start()
webserver.run(host='0.0.0.0')
# The values that are read from this file are written out at
# the end of /oauth_helper
oauth_token, oauth_token_secret = read_token_file(OAUTH_FILE)
# These four credentials are what is needed to authorize the application
auth= twitter.oauth.OAuth(oauth_token, oauth_token_secret,
                          CONSUMER_KEY, CONSUMER_SECRET)
twitter_api = twitter.Twitter(auth=auth)
print twitter_api
```

你应该观察到了你的应用检索到的访问令牌和访问令牌密码和应用设置中的相同，这并不是巧合。仔细保存好这些值，因为它们和用户名、密码组合起来一样有效。

9.3 探索流行话题

9.3.1 问题

你想知道在一个特定的地域内（比如美国、其他国家、一组国家，甚至是整个世界），Twitter上流行什么话题。

9.3.2 解决方案

Twitter的Trends API（http://bit.ly/1a1kYSQ）使你能够获得特定地域内的流行话题。这些地域是由雅虎设计并维护的Where On Earth（WOE）（http://yhoo.it/1a1kZ9u）ID来指定的。

9.3.3 讨论

地点（place）是Twitter开发平台中很重要的一个概念，因为流行话题会被地理位置约束，这样才能提供更好的API来查询流行话题（正如示例9-3中那样）。和其他API类似，该API返回的流行话题是JSON格式的，并能够转换为标准的Python对象，然后可以使用列表推导式（list

comprehension）或相似的技术来操作数据。这说明探索API响应是相当容易的事情了。你可以尝试多种多样的WOE ID来对比不同地域的流行话题。例如，对比两个不同国家的流行话题，或者对比特定国家和整个世界的流行话题。

注意：为了访问并查找Where On Earth（WOE）ID，你需要简单地完成雅虎的注册，因为这是雅虎开发的产品之一。这个步骤很简单却很值得。

示例9-3：探索流行话题

```
   import json
import twitter
def twitter_trends(twitter_api, woe_id):
    # Prefix ID with the underscore for query string parameterization.
    # Without the underscore, the twitter package appends the ID value
    # to the URL itself as a special-case keyword argument.
    return twitter_api.trends.place(_id=woe_id)
# Sample usage
twitter_api = oauth_login()
# See https://dev.twitter.com/docs/api/1.1/get/trends/place and
# http://developer.yahoo.com/geo/geoplanet/ for details on
# Yahoo! Where On Earth ID
WORLD_WOE_ID = 1
world_trends = twitter_trends(twitter_api, WORLD_WOE_ID)
print json.dumps(world_trends, indent=1)
US_WOE_ID = 23424977
us_trends = twitter_trends(twitter_api, US_WOE_ID)
print json.dumps(us_trends, indent=1)
```

9.4 查找推文

9.4.1 问题

你想通过特定的关键字和查询条件来查找Twitter中的推文。

9.4.2 解决方案

使用搜索API进行个性化查询。

9.4.3 讨论

示例9-4展示了如何使用搜索API（http://bit.ly/1a1l398）来对整个Twitter虚拟空间（Twitterverse）进行个性化查询。和搜索引擎的工作原理类似，Twitter的搜索API分批返回结果，并且你可以通过count关键字参数配置每次返回的数目（最多为200条）。很可能你的查询结果大于200（或者你自己用count指定的最大值），根据Twitter的API的要求，你将需要使用cursor来定位到下一批结果。

cursor（http://bit.ly/1a1pFMD）是在Twitter API 1.1版本中新增加

的，它提供了比1.0版本中的分页模式更加健壮的模式。原来的分页模式需要指定页码以及每页的结果限制。cursor模式中关键的一点是它能够更好地适应Twitter平台的动态实时性。例如，Twitter API的cursor设计时考虑了在你定位一批结果时，数据实时更新的可能性。换句话说，在你定位一批结果时，可能更多的你想要的相关数据出现了，这些数据会包含在你当前的结果中，你并不需要再次发起一个新的查询。

示例9-4展示了如何使用搜索API并通过返回结果中的cursor来获取多批结果。

示例9-4：查找推文

```
   def twitter_search(twitter_api, q, max_results=200, **kw):
    # See https://dev.twitter.com/docs/api/1.1/get/search/tweets and
    # https://dev.twitter.com/docs/using-search for details on advanced
    # search criteria that may be useful for keyword arguments
    # See https://dev.twitter.com/docs/api/1.1/get/search/tweets
    search_results = twitter_api.search.tweets(q=q, count=100, **kw)
    statuses = search_results['statuses']
    # Iterate through batches of results by following the cursor until we
    # reach the desired number of results, keeping in mind that OAuth users
    # can "only" make 180 search queries per 15-minute interval. See
    # https://dev.twitter.com/docs/rate-limiting/1.1/limits
    # for details. A reasonable number of results is ~1000, although
    # that number of results may not exist for all queries.
    # Enforce a reasonable limit
    max_results = min(1000, max_results)
    for_ in range(10): # 10*100 = 1000
        try:
            next_results = search_results['search_metadata']['next_results']
        except KeyError, e: # No more results when next_results doesn't exist
            break
        # Create a dictionary from next_results, which has the following form:
        # ?max_id=313519052523986943&q=NCAA&include_entities=1
        kwargs= dict([ kv.split('=')
                       for kv in next_results[1:].split("&") ])
        search_results = twitter_api.search.tweets(**kwargs)
        statuses += search_results['statuses']
        if len(statuses) > max_results:
            break
    return statuses
# Sample usage
```

```
twitter_api = oauth_login()
q = "CrossFit"
results = twitter_search(twitter_api, q, max_results=10)
# Show one sample search result by slicing the list...
print json.dumps(results[0], indent=1)
```

9.5 构造方便的函数调用

9.5.1 问题

你想要绑定某些参数到函数调用中，并传递引用到函数中以简化你的编程模式。

9.5.2 解决方案

使用Python的functools.partial来构造完整或部分的绑定函数，使之能够直接获得参数传递并被其他代码使用，而不需要传递额外的参数。

9.5.3 讨论

尽管functools.partial并不是针对Twitter API才专有的设计模式，但它和twitter包以及本手册中的许多模式还有你编过的其他Python程序相结合是非常方便的。比如，你可能觉得向需要授权的Twitter API连接器（twitter_api，正如这些方法中展示的那样，是大多数函数的第一个参数）连续的传递引用很麻烦，所以你会想构造一个仅满足部分参数的函

数，这样你通过简单地传递剩下的参数就能调用该函数了。

另一个例子展示了绑定部分参数的便利，你也许想在Trends API中绑定Twitter API连接器和地域的WOE ID，这样函数调用就能简单地传递剩下的参数了。你也可能决定例行的输入json.dumps（{…}，indent=1）很麻烦，所以你可以应用关键字参数，将函数进行简短的重命名，比如pp（pretty-print）来节省输入的时间。

其他的可能性还有很多，尽管你可能选择Python的def关键字来定义函数，但有时使用functools.partial会更简单。示例9-5展示了一些有用的可能性。

示例9-5：构造方便的函数调用

```
   from functools import partial
pp = partial(json.dumps, indent=1)
twitter_world_trends = partial(twitter_trends, twitter_api, WORLD_WOE_ID)
print pp(twitter_world_trends())
authenticated_twitter_search = partial(twitter_search, twitter_api)
results = authenticated_twitter_search("iPhone")
print pp(results)
authenticated_iphone_twitter_search = partial(authenticated_twitter_search, "iPhone")
results = authenticated_iphone_twitter_search()
print pp(results)
```

9.6 使用文本文件存储JSON数据

9.6.1 问题

你想要存储从Twitter API获取的少量数据，以便重复分析或者以存档为目的。

9.6.2 解决方案

将数据以方便的JSON格式存储于文本文件中。

9.6.3 讨论

虽然文本文件并不适用于所有情况，但如果你只是想把一些数据存储在磁盘上以便实验和分析，那么这种方式是很方便的选择。事实上，这种方式也是一种最佳实践，因为你可以最小化请求Twitter API的次数，避免可能遇到的频率限制问题。毕竟，一遍又一遍地获取重复的数据并没有什么意义。

示例9-6展示了Python的io包的使用，以确保你读/写的所有数据都

是使用UTF-8编解码的，这样可以避免（经常令人担忧又不容易被发现）UnicodeDecodeError异常，这种异常经常出现在Python 2.x应用对文本数据的序列化和反序列化当中。

示例9-6：使用文本文件存储JSON数据

```
   import io, json
def save_json(filename, data):
    with io.open('resources/ch09-twittercookbook/{0}.json'.format(filename),
                 'w', encoding='utf-8') as f:
        f.write(unicode(json.dumps(data, ensure_ascii=False)))
def load_json(filename):
    with io.open('resources/ch09-twittercookbook/{0}.json'.format(filename),
                 encoding='utf-8') as f:
        returnf.read()
# Sample usage
q = 'CrossFit'
twitter_api = oauth_login()
results = twitter_search(twitter_api, q, max_results=10)
save_json(q, results)
results = load_json(q)
print json.dumps(results, indent=1)
```

9.7 使用MongoDB存储和访问JSON数据

9.7.1 问题

你想要存储和访问Twitter API返回的JSON数据。

9.7.2 解决方案

使用文档型数据库（比如MongoDB）以方便的JSON格式存储数据。

9.7.3 讨论

虽然一个包含一小部分正确编码的JSON文件的文件夹可以用来存储数据，但你很快就会积累许多文件，这种方式就不合适了。幸运的是，文档型数据库（比如MongoDB）是理想的存储Twitter API返回的数据的方式，因为这种数据库就是为了高效地存储JSON数据而设计的。

MongoDB是一个健壮的文档型数据库，不管对少量还是大量数据来说。它提供了强大的查询操作和索引机制，这样就大幅度简化了你在

Python代码中的分析。

在大多数情况下，检索和查询数据时，MongoDB的索引机制和在磁盘上高效的BSON（http://bit.ly/1a1pG34）表示会做得比你个性化的操作更好。参考第6章获取更多的MongoDB的知识，比如存储（JSON化的邮箱）数据和使用MongoDB的聚合框架（aggregation framework）（http://bit.ly/1a1pGjv）繁琐地执行查询。示例9-7展示了如何连接运行时的MongoDB数据库来存储和装载数据。

注意：MongoDB安装起来很方便，并且含有出色的在线文档，内容包括安装/配置和查询/索引操作。如果你想直接开始，本书的虚拟机已经完成了安装。

示例9-7：使用MongDB存储和访问JSON数据

```
   import json
import pymongo # pip install pymongo
def save_to_mongo(data, mongo_db, mongo_db_coll, **mongo_conn_kw):
    # Connects to the MongoDB server running on
    # localhost:27017 by default
    client = pymongo.MongoClient(**mongo_conn_kw)
    # Get a reference to a particular database
    db = client[mongo_db]
    # Reference a particular collection in the database
    coll = db[mongo_db_coll]
    # Perform a bulk insert and return the IDs
    return coll.insert(data)
def load_from_mongo(mongo_db, mongo_db_coll, return_cursor=False,
                    criteria=None, projection=None, **mongo_conn_kw):
    # Optionally, use criteria and projection to limit the data that is
    # returned as documented in
    # http://docs.mongodb.org/manual/reference/method/db.collection.find/
    # Consider leveraging MongoDB's aggregations framework for more
    # sophisticated queries.
    client = pymongo.MongoClient(**mongo_conn_kw)
    db = client[mongo_db]
    coll= db[mongo_db_coll]
    if criteria is None:
```

```
        criteria = {}
    if projection is None:
        cursor = coll.find(criteria)
    else:
        cursor= coll.find(criteria, projection)
    # Returning a cursor is recommended for large amounts of data
    if return_cursor:
        return cursor
    else:
        return[ item for item in cursor ]
# Sample usage
q = 'CrossFit'
twitter_api = oauth_login()
results = twitter_search(twitter_api, q, max_results=10)
save_to_mongo(results, 'search_results', q)
load_from_mongo('search_results', q)
```

9.8 使用信息流API对Twitter数据管道抽样

9.8.1 问题

你想要分析用户实时发表的推文，而不是通过搜索API查询可能稍微（或非常）过时的信息。或者，你想要积累一个特定话题数据，以便将来的分析。

9.8.2 解决方案

使用Twitter的信息流API（http://bit.ly/1a1l1ya）对Twitter数据管道的公共数据抽样。

9.8.3 讨论

Twitter中有1%的推文可以通过随机抽样技术获取，它展示了更多的推文，并通过信息流API获得。除非你想要通过第三方提供商，比如GNIP（http://bit.ly/1a1pIrB）或DataSift（http://bit.ly/1a1pGQE，大多数情况下它都值得你为它花费）。尽管你可能觉得1%微不足道，但你要

认识到在峰值时期，每秒就会有成千上万条推文发出。对于很广泛的话题，存储所有抽样的推文比你想象的麻烦。访问1%的公共推文是非常大的工作。

虽然搜索API查询“历史”信息（考虑到流行产生和消失的速度，在Twitter虚拟空间中“历史”信息代表几分钟或几小时前的信息）更加简单，但信息流API提供了更接近于实时的从世界范围的信息中抽样的方法。twitter包使用更简单的方法调用信息流API，你可以根据关键字限制过滤掉数据管道，这是方便直观的访问数据的方法。与构造twitter.Twitter连接器不同，你构造的是twitter.TwitterStream连接器，它使用和9.1节与9.2节介绍的twitter.oauth.OAuth相同类型的关键字参数。示例9-8展示了如何使用Twitter的信息流API。

示例9-8：使用信息流API对Twitter数据管道抽样

```
   # Finding topics of interest by using the filtering capablities it offers.
import twitter
# Query terms
q = 'CrossFit' # Comma-separated list of terms
print >> sys.stderr, 'Filtering the public timeline for track="%s"' % (q,)
# Returns an instance of twitter.Twitter
twitter_api = oauth_login()
# Reference the self.auth parameter
twitter_stream = twitter.TwitterStream(auth=twitter_api.auth)
# See https://dev.twitter.com/docs/streaming-apis
stream = twitter_stream.statuses.filter(track=q)
# For illustrative purposes, when all else fails, search for Justin Bieber
# and something is sure to turn up (at least, on Twitter)
for tweet in stream:
    print tweet['text']
    # Save to a database in a particular collection
```

9.9 采集时序数据

9.9.1 问题

你想要定期查询Twitter API来获取特定的结果或者流行的话题，并且存储数据以便时序分析。

9.9.2 解决方案

如果9.8节介绍的信息流API无法使用，那么就使用Python内置的time.sleep函数，放在无限循环中发起查询并将数据存储于数据库（比如MongoDB）中。

9.9.3 讨论

虽然在特定时间内对特定关键字的逐点查询很简单，但抽样从一段时间采集的数据和查找流行趋势的能力让我们得以使用经常被忽视的强大的分析形式。每次回过头来我们总会说“我希望我知道……”这是潜在的能够让你先发制人的收集将来可能有用的数据或者预测未来（如果可

以）的机会。

对Twitter数据的时序分析非常有趣，因为可以了解潮流的消失、话题的变化和更新。尽管很多情况下从数据管道抽样并把结果存储在文档型数据库（比如MongoDB）中很有用，但有时定期发起查询并将结果按照时间间隔记录更加简单。例如，你可能会查询很多地域全天的流行话题、度量话题的变化率、对比不同地域的变化率，并找到长时间流行的话题和短时间流行的话题，等等。

关于Twitter中表达的观点和股市的关系，是另一个被活跃探索的有趣主题。对关键字、标签、流行话题的放大还有联系这些数据和真实的股市变化很简单，这可能是预测市场和商品的初始步骤。

示例9-9是9.1节、示例9-3和示例9-7的灵活运用，展示了如何使用原子性的方法来构造复杂的方法，这只需要一点点创新和复制/粘贴。

示例9-9：采集时序数据

```
   import sys
import datetime
import time
import twitter
def get_time_series_data(api_func, mongo_db_name, mongo_db_coll,
                         secs_per_interval=60, max_intervals=15, **mongo_conn_kw):
    # Default settings of 15 intervals and 1 API call per interval ensure that
    # you will not exceed the Twitter rate limit.
    interval= 0
    whileTrue:
        # A timestamp of the form "2013-06-14 12:52:07"
        now = str(datetime.datetime.now()).split(".")[0]
        ids = save_to_mongo(api_func(), mongo_db_name, mongo_db_coll + "-" + now)
        print >> sys.stderr, "Write {0} trends".format(len(ids))
        print >> sys.stderr, "Zzz..."
        print >> sys.stderr.flush()
```

```
        time.sleep(secs_per_interval) # seconds
        interval += 1
        if interval >= 15:
            break
# Sample usage
get_time_series_data(twitter_world_trends, 'time-series', 'twitter_world_trends')
```

9.10 提取推文实体

9.10.1 问题

你想要提取实体，比如@username的提到信息、#hashtags主题标签信息以及推文中的URL以作分析。

9.10.2 解决方案

从推文的entities域中提取实体。

9.10.3 讨论

Twitter的API现在提供了推文实体（http://bit.ly/1a1pIYA）作为大多数API返回结果中都有的域。示例9-10中提到的entities域，包括用户提到信息、标签信息、URL的引用信息、媒体对象（图片或者视频），还有财务信息，比如股票行情。目前，并不是所有的域在所有情况下都会出现。比如，media域只会当用户使用Twitter客户端嵌入媒体时才会出现，该客户端使用特定的API来嵌入内容；简单地复制/粘贴YouTube的

视频链接是不会出现media域的。

参考Tweet实体API文档（Tweet Entities API documentation）（http://bit.ly/1a1pIYA）获取更多的信息，包括每个类型的实体中包含的其他的域的信息。比如，对于URL来说，Twitter提供了一些变量，包括缩短或扩展的形式，还有某些情况下在用户界面中能够更友好地展示的值。

示例9-10：提取推文实体

```
    def extract_tweet_entities(statuses):
        # See https://dev.twitter.com/docs/tweet-entities for more details on tweet
        # entities
        if len(statuses) == 0:
            return[], [], [], [], []
        screen_names = [ user_mention['screen_name']
                             for status in statuses
                                 for user_mention in status['entities']['user_mentions'] ]
        hashtags= [ hashtag['text']
                        for status in statuses
                            for hashtag in status['entities']['hashtags'] ]
        urls= [ url['expanded_url']
                        for status in statuses
                            for url in status['entities']['urls'] ]
        symbols= [ symbol['text']
                    for status in statuses
                        for symbol in status['entities']['symbols'] ]
        # In some circumstances (such as search results), the media entity
        # may not appear
        if status['entities'].has_key('media'):
            media= [ media['url']
                            for status in statuses
                                for media in status['entities']['media'] ]
        else:
            media= []
        return screen_names, hashtags, urls, media, symbols
# Sample usage
q = 'CrossFit'
statuses = twitter_search(twitter_api, q)
screen_names, hashtags, urls, media, symbols = extract_tweet_entities(statuses)
# Explore the first five items for each...
print json.dumps(screen_names[0:5], indent=1)
print json.dumps(hashtags[0:5], indent=1)
print json.dumps(urls[0:5], indent=1)
print json.dumps(media[0:5], indent=1)
print json.dumps(symbols[0:5], indent=1)
```

9.11 特定的推文范围内查找最流行的推文

9.11.1 问题

你想要在查询到的推文的部分结果中（或者其他一些推文，比如某个用户的推文）查找最流行的推文。

9.11.2 解决方案

分析推文的retweet_count域，看看这个推文有没有被转推以及被转推的次数。

9.11.3 讨论

像示例9-11那样，分析retweet_count域，这或许是最直观的分析推文流行度的方法了，因为流行的推文总会被分享给别人。根据对“流行”的解释，另一个应该考虑的值是favorite_count，它代表了用户对该推文点赞的个数。

比如，对于被转推也被点赞的推文，你可能给retweet_count分配1.0

的权重，给favorite_count分配0.1的权重。权重如何分配完全取决于解决特定问题时，你认为哪个值更加重要。另一种可能性是，比如指数级衰减（exponential decay）（http://bit.ly/1a1pHEe）会受时间影响，时间越新的推文更加活跃，这在有些分析中需要注意。

注意：同时参考9.14节和9.15节获取更多讨论信息，这可能有助于在分析和应用转推属性时进行查找，这可能比刚开始看起来要复杂一些。

示例9-11：在特定的推文范围内查找最流行的推文

```
   import twitter
def find_popular_tweets(twitter_api, statuses, retweet_threshold=3):
    # You could also consider using the favorite_count parameter as part of
    # this heuristic, possibly using it to provide an additional boost to
    # popular tweets in a ranked formulation
    return [ status
                for status in statuses
                    if status['retweet_count'] > retweet_threshold ]
# Sample usage
q = "CrossFit"
twitter_api = oauth_login()
search_results = twitter_search(twitter_api, q, max_results=200)
popular_tweets = find_popular_tweets(twitter_api, search_results)
for tweet in popular_tweets:
    print tweet['text'], tweet['retweet_count']
```

警告：推文中的retweeted属性并不是告诉你推文有没有被转推的捷径。所谓的“perspectival”属性可以告诉你授权的用户（如果你在分析自己的数据，那该用户就是你）被转推的状态，这在用户界面中的快速标记很方便。它被称为perspectival属性，因为它提供了授权用户的信息。

9.12 特定的推文范围内查找最流行的推文实体

9.12.1 问题

你想要查找有没有流行的推文实体，比如@username的提到信息、#hashtags标签信息或者URL，这会让你看到推文的本质。

9.12.2 解决方案

提取推文实体并列出，对它们进行计数并过滤掉那些没有达到最小阈值的推文实体。

9.12.3 讨论

Twitter的API提供了直接对推文实体元数据值的访问，它是通过entities域实现的，9.10节中有提到。提取实体之后，可以计算每个实体的频率，并通过collections.Counter（见示例9-12）简单的提取常见的实体，这是Python的标准库中主要的功能，在用Python进行频率分析时非常方便。有了排序过的推文实体的集合，剩下的就是对推文集合进行过

滤或使用其他阈值标准，以便找到特定的推文实体。

示例9-12：在特定的推文范围内查找最流行的推文实体

```
   import twitter
from collections import Counter
def get_common_tweet_entities(statuses, entity_threshold=3):
    # Create a flat list of all tweet entities
    tweet_entities = [ e
                       for status in statuses
                           for entity_type in extract_tweet_entities([status])
                               for e in entity_type
                     ]
    c = Counter(tweet_entities).most_common()
    # Compute frequencies
    return [ (k,v)
            for(k,v) in c
                if v >= entity_threshold
          ]
# Sample usage
q = 'CrossFit'
twitter_api = oauth_login()
search_results = twitter_search(twitter_api, q, max_results=100)
common_entities = get_common_tweet_entities(search_results)
print "Most common tweet entities"
print common_entities
```

9.13 对频率分析制表

9.13.1 问题

你想要对频率分析的结果制表，以便轻易地掠过某些结果或者使结果以更方便阅读的格式展示。

9.13.2 解决方案

使用prettytable包可以轻易地建立能够加载一行一行的信息的对象，每列还能以固定宽度展示表格。

9.13.3 讨论

prettytable包使用起来非常方便，尤其是在构造易读的能够复制/粘贴到报表或文本文件（参考示例9-13）中的文本输出方面非常有用。使用pip install prettytable命令来安装这个包。一起使用prettytable.PrettyTable和collections.Counter或其他从三元组列表中提取的数据结构很方便，这些三元组能够被排序以便分析。

注意：如果你对电子制表软件的数据存储感兴趣，你可以查看csv包的文档（documention on the csv package，http://bit.ly/1a1pHUH）。然而，你需要注意那些已知的问题（正如记载的那样）——关于对Unicode的支持的问题。

示例9-13：对频率分析制表

```
   from prettytable import PrettyTable
# Get some frequency data
twitter_api = oauth_login()
search_results = twitter_search(twitter_api, q, max_results=100)
common_entities = get_common_tweet_entities(search_results)
# Use PrettyTable to create a nice tabular display
pt = PrettyTable(field_names=['Entity', 'Count'])
[ pt.add_row(kv) for kv in common_entities ]
pt.align['Entity'], pt.align['Count'] = 'l', 'r' # Set column alignment
print pt
```

9.14 查找转推了状态的用户

9.14.1 问题

你想查找所有曾经转推过某条状态的用户。

9.14.2 解决方案

使用GET retweeters/ids API来查找哪些用户曾经转推过某条状态。

9.14.3 讨论

尽管GET retweeters/ids API（http://bit.ly/1a1pJvI）返回转推过某个状态的用户的ID，但有些东西你还是需要了解。特别的，你要记住这个API返回的用户只是那些使用Twitter的原生的转推API转推状态的用户，而不是复制/粘贴推文并在前面使用“RT”，并附加上“（via@exampleUser）”属性或其他常见的东西的用户。

大部分Twitter应用（包括twitter.com的网页）使用原生的转推API，但有些用户可能选择通过“迂回”（working around）原生的API来分享状

态，目的是附加额外的评论或将它们自己插入作为中介来广播的对话中。比如，用户可能以“<AWESOME！”作为推文的后缀，以表达自己喜爱的程度。尽管用户可能把这视为转推，但在Twitter API的角度，它是在引用这个推文。至少引用和转推存在混乱的原因部分是Twitter并不经常提供原生的转推API。事实上，转推的概念是演化出来的现象，Twitter最终在2010年底通过提供优秀的API支持来做出回应。

下面的说明或许能够讲清楚这个技术细节：假设@fperez_org发表了一个状态，然后@SocialWebMining转推了这条状态。这时，@fperez_org发表的状态的retweet_count的值会变为1，@SocialWebMining会在他的状态中包含这条状态，这表明了对@fperez_org的状态的转推。

现在让我们假设@jyeee在twitter.com中或者TweetDeck（http://bit.ly/1a1pIbh）这样的应用中，通过@SocialWebMining的转推注意到了@fperez_org的状态，并也转发了这条状态。这时，@fperez_org的状态中，retweet_count的值会变为2，@jyeee也会在他的状态中包含这条状态（和@SocialWebMining的最后一条状态一样），这表明了对@fperez_org的转推。

一个需要理解的重要部分是：对任何浏览@jyeee的状态的用户来说，@Social WebMining作为中介对@fperez_org和@jyeee的联系就会消失。换句话说，@fperez_org会收到原始推文的属性，无论哪些包含流

行状态的多重中介的反应链都会被抵消。

有了任何转推推文的用户的ID，使用GET users/lookup API获取用户信息是非常简单的。参考9.17节获取更多的信息。

虽然示例9-14可能无法完全满足你的需求，但你要认真考虑9.15节中额外的步骤，这样你才能找到状态的广播者。如果你在处理历史推文或者想要检查属性信息的内容，它提供了示例说明使用正则表达式分析140个字符的推文内容并提取引用的推文的属性信息。

示例9-14：查找转推了状态的用户

```
   import twitter
twitter_api = oauth_login()
print """User IDs for retweeters of a tweet by @fperez_org
that was retweeted by @SocialWebMining and that @jyeee then retweeted
from @SocialWebMining's timeline\n"""
print twitter_api.statuses.retweeters.ids(_id=334188056905129984)['ids']
print json.dumps(twitter_api.statuses.show(_id=334188056905129984), indent=1)
print
print "@SocialWeb's retweet of @fperez_org's tweet\n"
print twitter_api.statuses.retweeters.ids(_id=345723917798866944)['ids']
print json.dumps(twitter_api.statuses.show(_id=345723917798866944), indent=1)
print
print "@jyeee's retweet of @fperez_org's tweet\n"
print twitter_api.statuses.retweeters.ids(_id=338835939172417537)['ids']
print json.dumps(twitter_api.statuses.show(_id=338835939172417537), indent=1)
```

注意：一些Twitter用户故意引用推文，而不是使用转推API，是为了将他们自己加入对话并且转推自己，查看RT和via功能都很常见。事实上，流行的应用（比如TweetDeck）包括辨别“Edit and RT”还有原生的“Retweet”（转推）功能，见图9-2。

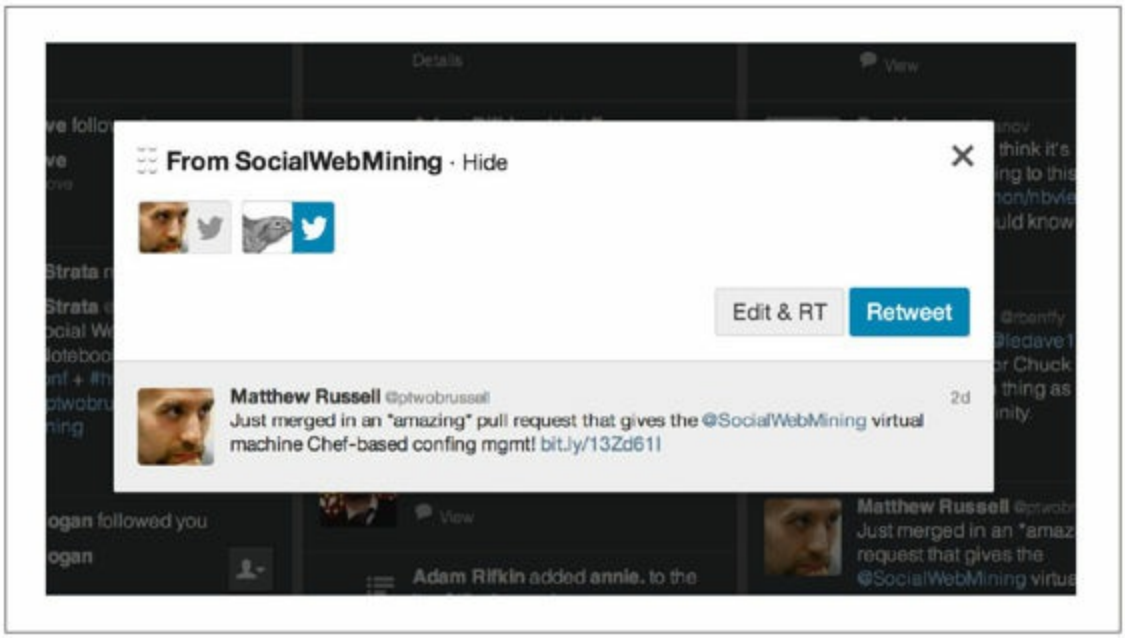


图9-2：流行的应用（比如Twitter自己的TweetDeck）提供了“Edit andRT”功能来“引用”推文，还有更新和原生的“Retweet”（转推）功能

9.15 提取转推的属性

9.15.1 问题

你想要查看推文原始的属性。

9.15.2 解决方案

使用分析140个字符的推文的正则表达式启发式方法来分析“RT@SocialWebMining”或“（via@SocialWebMining）”这样的展示格式。

9.15.3 讨论

探索9.14节中描述的Twitter原生的转推API的结果，这能提供一些推文原始的属性，但肯定不是全部。有时使用这种方法的用户会因为一些原因把自己加入对话，所以为了探索原始的属性，分析某些特定的推文很有必要。示例9-15展示了如何使用Python的正则表达式来探索常用的展示形式。这些形式在Twitter原生的转推API发布之前就采用了，现

在还很常见。

示例9-15：提取转推的属性

```
   import re
def get_rt_attributions(tweet):
    # Regex adapted from Stack Overflow (http://bit.ly/1821y0J)
    rt_patterns = re.compile(r"(RT|via)((?:\b\W*@\w+)+)", re.IGNORECASE)
    rt_attributions = []
    # Inspect the tweet to see if it was produced with /statuses/retweet/:id.
    # See https://dev.twitter.com/docs/api/1.1/get/statuses/retweets/%3Aid.
    if tweet.has_key('retweeted_status'):
        attribution= tweet['retweeted_status']['user']['screen_name'].lower()
        rt_attributions.append(attribution)
    # Also, inspect the tweet for the presence of "legacy" retweet patterns
    # such as "RT" and "via", which are still widely used for various reasons
    # and potentially very useful. See https://dev.twitter.com/discussions/2847
    # and https://dev.twitter.com/discussions/1748 for some details on how/why.
    try:
        rt_attributions += [
                        mention.strip()
                        for mention in rt_patterns.findall(tweet['text'])[0]
[1].split()
                    ]
    except IndexError, e:
        pass
    # Filter out any duplicates
    return list(set([rta.strip("@").lower() for rta in rt_attributions]))
# Sample usage
twitter_api = oauth_login()
tweet = twitter_api.statuses.show(_id=214746575765913602)
print get_rt_attributions(tweet)
print
tweet = twitter_api.statuses.show(_id=345723917798866944)
print get_rt_attributions(tweet)
```

9.16 创建健壮的Twitter请求

9.16.1 问题

在收集用于分析的数据过程中，你会碰到一些需要按每个实际情况来处理意想不到的HTTP错误，从超出访问速率上限限制（429错误）到一向很棘手的“fail whale”（503错误）。

9.16.2 解决方案

编写一个通用的API封装函数来提供可以有效控制各种HTTP错误代码的抽象逻辑。

9.16.3 讨论

即使Twitter的频率限制对于多数应用来说是够用的，但是他们对于数据挖掘却是不够的，所以通常来说你需要控制一段固定时间内请求的数目并且考虑其他种类的HTTP错误，例如很常见的“fail whale”或者别的突发网络故障。一个解决方案，如示例9-16中展示的，就是来编写一

个封装函数来把这种混乱的逻辑抽象化，以允许你简单的编写脚本，就好像频率限制和HTTP错误在大部分情况下不存在。

注意：参考9.5节来获得如何使用标准库中的functools.partial函数来简化某些场合下封装函数的编写。同样的，请参考complete listing of Twitter's HTTP error codes（所有的Twitter HTTP错误码，http://bit.ly/1a1pL6O）。9.19节提供了可以说明如何使用名为make_twitter_request的函数来简化你在获取Twitter数据过程中可能遇到的HTTP错误的具体实现。

示例9-16：创建健壮的Twitter请求

```
   import sys
import time
from urllib2 import URLError
from httplib import BadStatusLine
import json
import twitter
def make_twitter_request(twitter_api_func, max_errors=10, *args, **kw):
    # A nested helper function that handles common HTTPErrors. Return an updated
    # value for wait_period if the problem is a 500 level error. Block until the
    # rate limit is reset if it's a rate limiting issue (429 error). Returns None
    # for 401 and 404 errors, which requires special handling by the caller.
    def handle_twitter_http_error(e, wait_period=2, sleep_when_rate_limited=True):
        if wait_period > 3600: # Seconds
            print >> sys.stderr, 'Too many retries. Quitting.'
            raise e
        # See https://dev.twitter.com/docs/error-codes-responses for common codes
        if e.e.code == 401:
            print >> sys.stderr, 'Encountered 401 Error (Not Authorized)'
            return None
        elif e.e.code == 404:
            print >> sys.stderr, 'Encountered 404 Error (Not Found)'
            return None
        elif e.e.code == 429:
            print >> sys.stderr, 'Encountered 429 Error (Rate Limit Exceeded)'
            if sleep_when_rate_limited:
                print >> sys.stderr, "Retrying in 15 minutes...ZzZ..."
                sys.stderr.flush()
                time.sleep(60*15 + 5)
                print >> sys.stderr, '...ZzZ...Awake now and trying again.'
                return2
```

```
            else:
                raise e # Caller must handle the rate limiting issue
        elif e.e.code in (500, 502, 503, 504):
            print >> sys.stderr, 'Encountered %i Error. Retrying in %i seconds' % \
                (e.e.code, wait_period)
            time.sleep(wait_period)
            wait_period *= 1.5
            return wait_period
        else:
            raise e
    # End of nested helper function
    wait_period = 2
    error_count = 0
    while True:
        try:
            return twitter_api_func(*args, **kw)
        except twitter.api.TwitterHTTPError, e:
            error_count = 0
            wait_period = handle_twitter_http_error(e, wait_period)
            if wait_period is None:
                return
        except URLError, e:
            error_count += 1
            print >> sys.stderr, "URLError encountered. Continuing."
            if error_count > max_errors:
                print >> sys.stderr, "Too many consecutive errors...bailing out."
                raise
        except BadStatusLine, e:
            error_count += 1
            print >> sys.stderr, "BadStatusLine encountered. Continuing."
            if error_count > max_errors:
                print >> sys.stderr, "Too many consecutive errors...bailing out."
                raise
# Sample usage
twitter_api = oauth_login()
# See https://dev.twitter.com/docs/api/1.1/get/users/lookup for
# twitter_api.users.lookup
response = make_twitter_request(twitter_api.users.lookup,
                                screen_name="SocialWebMining")
print json.dumps(response, indent=1)
```

9.17 获取用户个人资料信息

9.17.1 问题

你想要根据一个或多个用户的ID或昵称获取他们的个人资料信息。

9.17.2 解决方案

通过GET users/lookup API，一次使用100个ID或昵称来获取用户完整的个人资料。

9.17.3 讨论

许多API（比如GET friends/ids和GET followers/ids）需要用户名或其他个人信息来返回含糊的ID值，以便进行有意义的分析。Twitter提供的GET users/lookup API可以每次获取100个ID或昵称，并且可以使用简单的模式来遍历大量的数据。尽管它在逻辑上稍增加了一些复杂性，但可以构造一个简单的接受用户名或ID的关键字参数的函数来获取用户个人资料。示例9-17展示了能够应用在很多场合的函数，并提供了在你想

获得用户ID的情况下的附加支持。

示例9-17：获取用户个人资料信息

```
   def get_user_profile(twitter_api, screen_names=None, user_ids=None):
    # Must have either screen_name or user_id (logical xor)
    assert (screen_names != None) != (user_ids != None), \
    "Must have screen_names or user_ids, but not both"
    items_to_info = {}
    items = screen_names or user_ids
    while len(items) >0:
        # Process 100 items at a time per the API specifications for /users/lookup.
        # See https://dev.twitter.com/docs/api/1.1/get/users/lookup for details.
        items_str = ','.join([str(item) for item in items[:100]])
        items = items[100:]
        if screen_names:
            response = make_twitter_request(twitter_api.users.lookup,
                                            screen_name=items_str)
        else: # user_ids
            response= make_twitter_request(twitter_api.users.lookup,
                                            user_id=items_str)
        for user_info in response:
            if screen_names:
            items_to_info[user_info['screen_name']] = user_info
        else: # user_ids
            items_to_info[user_info['id']] = user_info
    return items_to_info
# Sample usage
twitter_api = oauth_login()
print get_user_profile(twitter_api, screen_names=["SocialWebMining", "ptwobrussell"])
#print get_user_profile(twitter_api, user_ids=[132373965])
```

9.18 从任意的文本中提取推文实体

9.18.1 问题

你想要分析任意文本并且提取推文实体，例如@username提到的信息，#hashtags主题标签信息以及其中可能出现的URL。

9.18.2 解决方案

使用第三方的包比如twitter_text来从任意文本中提取推文实体，比如历史推文档案可能不像现在v1.1API提供的那样，它不包含推文实体。

9.18.3 讨论

Twitter并没有一直提取推文实体，但是你可以通过名为twitter_text的第三方包来自己轻易获得，正如示例9-18中展示的那样。你可以使用pip命令的pip install twitter-text-py来安装twitter-text。

示例9-18：从任意的文本中提取推文实体

```
   import twitter_text
# Sample usage
txt = "RT @SocialWebMining Mining 1M+ Tweets About #Syria http://wp.me/p3QiJd-1I"
ex = twitter_text.Extractor(txt)
print "Screen Names:", ex.extract_mentioned_screen_names_with_indices()
print "URLs:", ex.extract_urls_with_indices()
print "Hashtags:", ex.extract_hashtags_with_indices()
```

9.19 获得用户所有的好友和关注者

9.19.1 问题

你想要获得Twitter用户（可能是很受欢迎的用户）所有的好友和关注者。

9.19.2 解决方案

使用9.16节介绍的make_twitter_request函数，考虑到有时关注者人数可能超过预先定义的限制，所以简化对ID获取的处理。

9.19.3 讨论

GET followers/ids和GET friends/ids提供了能够导航到特定用户所有的好友和关注者的ID的API，但由于每个API请求每次最多会返回5000个ID，所以获取所有ID的逻辑并不是非常简单。尽管大多数用户不会有5000个好友或关注者，但一些让人很有兴趣分析的名人用户会有成百上千甚至是百万个关注者。获取所有的ID很有挑战性，因为我们需要用

cursor遍历每次返回的结果，而且还要考虑到这个过程中可能的HTTP错误。幸运的是，应用make_twitter_request并像前面介绍的那样使用cursor系统获取所有的ID并不是很难。

示例9-19中介绍相似的技术可以和9.17节提供的模板结合，来构造一个健壮的函数。该函数提供中间的步骤，比如通过用户名获取ID的子集（或所有的）。建议将结果存储在文档型数据库（比如9.7.1节中介绍的MongoDB）中，这样在大量数据操作时，就不会有数据因为不可预料的错误而丢失了。

注意：有时最好支付像DataSift（http://bit.ly/1a1pKje）这样的第三方以便能够更加快速的访问某些特定的数据，比如名人用户（如@ladygaga）的所有关注者的完整的个人资料。在你尝试收集大量数据之前，至少要想出算法并了解这需要花费多久，考虑在这个长时间运行的过程中可能的（不可预料的）错误，并想想是否从其他途径获取数据会不会更好。花钱的事情往往会节省你的时间。

示例9-19：获得用户所有的好友和关注者

```
   from functools import partial
from sys import maxint
def get_friends_followers_ids(twitter_api, screen_name=None, user_id=None,
                              friends_limit=maxint, followers_limit=maxint):
    # Must have either screen_name or user_id (logical xor)
    assert (screen_name != None) != (user_id != None), \
    "Must have screen_name or user_id, but not both"
    # See https://dev.twitter.com/docs/api/1.1/get/friends/ids and
    # https://dev.twitter.com/docs/api/1.1/get/followers/ids for details
    # on API parameters
    get_friends_ids = partial(make_twitter_request, twitter_api.friends.ids,
                              count=5000)
```

```
    get_followers_ids = partial(make_twitter_request, twitter_api.followers.ids,
                                count=5000)
    friends_ids, followers_ids = [], []
    for twitter_api_func, limit, ids, label in [
                [get_friends_ids, friends_limit, friends_ids, "friends"],
                [get_followers_ids, followers_limit, followers_ids, "followers"]
            ]:
    if limit == 0: continue
    cursor = -1
    while cursor != 0:
        # Use make_twitter_request via the partially bound callable...
        if screen_name:
            response= twitter_api_func(screen_name=screen_name, cursor=cursor)
        else: # user_id
            response= twitter_api_func(user_id=user_id, cursor=cursor)
        if response is not None:
            ids += response['ids']
            cursor = response['next_cursor']
        print >> sys.stderr, 'Fetched {0} total {1} ids for {2}'.format(len(ids),
                                                label, (user_id or screen_name))
        # XXX: You may want to store data during each iteration to provide an
        # an additional layer of protection from exceptional circumstances
        if len(ids) >= limit or response is None:
            break
    # Do something useful with the IDs, like store them to disk...
    return friends_ids[:friends_limit], followers_ids[:followers_limit]
# Sample usage
twitter_api = oauth_login()
friends_ids, followers_ids = get_friends_followers_ids(twitter_api,
                                                       screen_name="SocialWebMining",
                                                       friends_limit=10,
                                                       followers_limit=10)
print friends_ids
print followers_ids
```

9.20 分析用户的好友和关注者

9.20.1 问题

你想要对用户的好友和关注者做一比较，进行基本的分析。

9.20.2 解决方案

使用集合操作，例如集合求交和集合求差，分析用户的好友和关注者。

9.20.3 讨论

获得所有用户的好友和关注者后，你可以只使用ID值本身参与集合操作，比如求交集和求差集，进行一些基本的分析，如示例9-20中那样。

给定两个集合，则集合的交集返回它们具有共有的条目，而集合间的差集是把在一个集合中出现的条目都从另一个集合中移除掉，留下不同的条目。回想一下，求交集是一个可交换操作，而求差集却是不可交

换的[1]。

在分析好友和关注者这一任务下，两个集合的交集可以解释成“互为好友”或你关注的人也关注你，而两个集合的差集根据操作数的次序可以解释成关注你而你未反过来关注他的人或者你关注他而他未关注你的人。

给定一份好友和关注者ID的完整列表，计算这些集合操作是一种很自然的切入点，同时也可以是后续分析的跳板。例如，它可能不需要使用GET users/lookup类的API来获取数以百万计关注者的档案作为分析的中间过程。

你可能反而选择计算集合操作的结果，如互为好友（其中有可能存在更强的亲密性），并在进一步扩大范围之前专注于这些用户ID的档案。

示例9-20：分析用户的好友和关注者

```
    def setwise_friends_followers_analysis(screen_name, friends_ids, followers_ids):
     friends_ids, followers_ids = set(friends_ids), set(followers_ids)
     print '{0} is following {1}'.format(screen_name, len(friends_ids))
     print '{0} is being followed by {1}'.format(screen_name, len(followers_ids))
     print '{0} of {1} are not following {2} back'.format(
             len(friends_ids.difference(followers_ids)),
             len(friends_ids), screen_name)
     print '{0} of {1} are not being followed back by {2}'.format(
             len(followers_ids.difference(friends_ids)),
             len(followers_ids), screen_name)
     print '{0} has {1} mutual friends'.format(
             screen_name, len(friends_ids.intersection(followers_ids)))
# Sample usage
screen_name = "ptwobrussell"
twitter_api = oauth_login()
friends_ids, followers_ids = get_friends_followers_ids(twitter_api,
```

```
                                                    screen_name=screen_name)
setwise_friends_followers_analysis(screen_name, friends_ids, followers_ids)
```

[1] 一个可交换操作是指操作数的次序不影响操作的结果，即操作数可交换。如加法和乘法都是可交换的操作。

9.21 获取用户的推文

9.21.1 问题

你想要获得所有用户的最新推文，以备分析之用。

9.21.2 解决方案

使用GET statuses/user_timeline的API入口来检索一个用户多达3200的最近推文。考虑到这一系列API请求可能会超出访问速率上限或者会在过程中遇到HTTP错误，最好有一个强大的API包装函数，如make_twitter_request来协助（如9.16节所介绍的）。

9.21.3 讨论

时间轴是Twitter开发者生态系统的一个基本概念，并且Twitter提供了一个方便的API入口来通过“用户时间轴”获取一个用户的推文。获取用户的推文，如示例9-21中所展示，是一个有意义分析的切入点，因为推文是这个生态系统中最基本的组成单元。收集得到的某一特定用户的

大量推文，为推断这个人所谈论（进而关心）的事物提供了令人难以置信的洞察力。借助某用户的数百个推文档案，你往往需要发起很少的API访问就可以进行几十个实验。把推文存储在文档级别的数据库中，例如MongoDB，在实验过程中可以很方便地进行存储和访问数据。对于注册时间更长的Twitter用户，执行时间序列分析来探索兴趣或情绪随时间变化的规律可能是值得尝试的练习。

示例9-21：获取用户的推文

```
   def harvest_user_timeline(twitter_api, screen_name=None, user_id=None,
max_results=1000):
    assert (screen_name != None) != (user_id != None), \
    "Must have screen_name or user_id, but not both"
    kw = {  # Keyword args for the Twitter API call
        'count': 200,
        'trim_user': 'true',
        'include_rts' : 'true',
        'since_id' : 1
        }
    if screen_name:
        kw['screen_name'] = screen_name
    else:
        kw['user_id'] = user_id
    max_pages = 16
    results = []
    tweets = make_twitter_request(twitter_api.statuses.user_timeline, **kw)
    if tweets is None: # 401 (Not Authorized) - Need to bail out on loop entry
        tweets = []
    results += tweets
    print >> sys.stderr, 'Fetched %i tweets' % len(tweets)
    page_num = 1
    # Many Twitter accounts have fewer than 200 tweets so you don't want to enter
    # the loop and waste a precious request if max_results = 200.
    # Note: Analogous optimizations could be applied inside the loop to try and
    # save requests. e.g. Don't make a third request if you have 287 tweets out of
    # a possible 400 tweets after your second request. Twitter does do some
    # post-filtering on censored and deleted tweets out of batches of 'count', though,
    # so you can't strictly check for the number of results being 200. You might get
    # back 198, for example, and still have many more tweets to go. If you have the
    # total number of tweets for an account (by GET /users/lookup/), then you could
    # simply use this value as a guide.
    if max_results == kw['count']:
        page_num = max_pages # Prevent loop entry
    while page_num < max_pages and len(tweets) > 0 and len(results) < max_results:
        # Necessary for traversing the timeline in Twitter's v1.1 API:
        # get the next query's max-id parameter to pass in.
```

```
        # See https://dev.twitter.com/docs/working-with-timelines.
        kw['max_id'] = min([ tweet['id'] for tweet in tweets]) - 1
        tweets = make_twitter_request(twitter_api.statuses.user_timeline, **kw)
        results += tweets
        print >> sys.stderr, 'Fetched %i tweets' % (len(tweets),)
        page_num += 1
    print >> sys.stderr, 'Done fetching tweets'
    return results[:max_results]
# Sample usage
twitter_api = oauth_login()
tweets = harvest_user_timeline(twitter_api, screen_name="SocialWebMining", \
                               max_results=200)
# Save to MongoDB with save_to_mongo or a local file with save_json...
```

9.22 爬取好友关系图

9.22.1 问题

你想要获取用户的关注者的ID、关注者的关注者的ID、关注者的关注者的那些关注者的ID以此类推，作为网络分析的一部分。本质上也就是要爬取一个Twitter上的“关注”好友关系图。

9.22.2 解决方案

使用广度优先搜索策略，系统地获取好友信息，可以相当容易地解释为在网络分析中的图谱。

9.22.3 讨论

广度优先搜索是一种常用于图搜索的技术，是你切入问题并建立关于关系的多级语境的标准做法之一。给定一个起点和一个深度，一个广度优先遍历系统地探索空间，使得它可以保证最终返回处在此深度下的所有节点，并且在搜索空间时，每个深度完成探索之后才进入下一深度

的探索（见示例9-22）。

请记住，很可能在探索Twitter的好友关系图时会遇到超级节点。超级节点是那些具有很高出度的节点，它消耗计算机资源以及API请求以致到达访问速率上限的速度很快。至少在你初始分析时，你控制图中每个待获取用户的关注者的最大数量是可取的。因此你可以知道你面对的困难是什么，并可以断定在解决特定问题时，那些超级节点是否值得付出时间和精力。探索一个未知的图是一项复杂的（和令人兴奋的）问题，并且其他类型的多种工具，如采样技术，可以巧妙地结合进来以进一步提高搜索的效率。

示例9-22：爬取一个好友关系图

```
    def crawl_followers(twitter_api, screen_name, limit=1000000, depth=2):
        # Resolve the ID for screen_name and start working with IDs for consistency
        # in storage
        seed_id = str(twitter_api.users.show(screen_name=screen_name)['id'])
        _, next_queue = get_friends_followers_ids(twitter_api, user_id=seed_id,
                                                  friends_limit=0, followers_limit=limit)
        # Store a seed_id => _follower_ids mapping in MongoDB
        save_to_mongo({'followers' : [ _id for _id in next_queue ]}, 'followers_crawl',
                      '{0}-follower_ids'.format(seed_id))
        d = 1
        while d < depth:
            d += 1
            (queue, next_queue) = (next_queue, [])
            for fid in queue:
                follower_ids = get_friends_followers_ids(twitter_api, user_id=fid,
                                                         friends_limit=0,
                                                         followers_limit=limit)
                # Store a fid => follower_ids mapping in MongoDB
                save_to_mongo({'followers' : [ _id for _id in next_queue ]},
                              'followers_crawl', '{0}-follower_ids'.format(fid))
                next_queue += follower_ids
# Sample usage
screen_name = "timoreilly"
twitter_api = oauth_login()
crawl_followers(twitter_api, screen_name, depth=1, limit=10)
```

9.23 分析推文内容

9.23.1 问题

给定一组推文，你想要针对每个140个字符的内容做一些粗略的分析，以获得更好的感知讨论主旨以及推文本身表达的观点。

9.23.2 解决方案

使用简单的统计，如词汇丰富性和每个推文的平均词数，在理解语言本质过程中发现最基本的观点。

9.23.3 讨论

除了分析推文实体内容并简单地分析常用词的频率，你也可以检查推文的词汇丰富性以及计算其他简单的统计量，如每个推文的平均词长，以更好地理解数据（参见示例9-23）。词汇丰富性是一个简单的统计量，它定义为不同的词数除以语料中全部词数；根据定义，词汇丰富性1.0意味着语料中所有词都是不同的，而词汇丰富性接近0.0意味着更

多的重复词。

根据语境的不同，词汇丰富性可以有稍微不同的解释。例如，文学语境下，比较两位作者的词汇丰富性可能被用来衡量和对比他们的语言的丰富性和表现力。虽然就其本身而言，通常这不是最终目标，但是它对考察词汇的丰富性往往提供了宝贵的初始见解（通常与频率分析相结合），可以用来更好地了解可能的后续步骤。

在Twitter虚拟空间（Twittersphere），如果比较两个Twitter用户，词汇的丰富性可能以类似的方式解释，但它也可能暗示了很多关于正在讨论整体内容的相对丰富性，可能是跟一个人谈论技术而跟另一个人谈到更广泛的主题。在类似于多个作者发布同一主题的一组推文的语境下（如在检查由搜索API或流API返回的推文集合的情况下），远低于预期的词汇丰富性也可能意味着有很多“集体思维”正在涌现。另一种可能性是很多转推（retweeting），其中相同的信息被或多或少地复述。与任何其他的分析类似，任何统计量都无法脱离语境进行孤立地解释。

示例9-23：分析推文内容

```
def analyze_tweet_content(statuses):
 if len(statuses) == 0:
     print "No statuses to analyze"
     return
 # A nested helper function for computing lexical diversity
 def lexical_diversity(tokens):
     return 1.0*len(set(tokens))/len(tokens)
 # A nested helper function for computing the average number of words per tweet
 def average_words(statuses):
     total_words = sum([ len(s.split()) for s in statuses ])
     return 1.0*total_words/len(statuses)
 status_texts = [ status['text'] for status in statuses ]
```

```
    screen_names, hashtags, urls, media, _ = extract_tweet_entities(statuses)
    # Compute a collection of all words from all tweets
    words = [ w
          for t in status_texts
              for w in t.split() ]
    print "Lexical diversity (words):", lexical_diversity(words)
    print "Lexical diversity (screen names):", lexical_diversity(screen_names)
    print "Lexical diversity (hashtags):", lexical_diversity(hashtags)
    print "Averge words per tweet:", average_words(status_texts)
# Sample usage
q = 'CrossFit'
twitter_api = oauth_login()
search_results = twitter_search(twitter_api, q)
analyze_tweet_content(search_results)
```

9.24 提取链接目标摘要

9.24.1 问题

你想要粗略地理解链接目标（如提取推文实体的URL）所表达的内容以获得推文的本质和Twitter用户兴趣上的见解。

9.24.2 解决方案

将链接指向的内容概括为几句话，达到可以很快浏览（或从另外角度进行更扼要的分析）而不用阅读整个网页的目的。

9.24.3 讨论

在尝试理解网页中的人类语言数据时，只有你想不到，没有你做不到。示例9-24尝试提供一个处理和提取网页内容为简洁形式的模板，从而可以快速翻阅或者采用其他手段予以分析。简而言之，它展示了如何获取一个网页、分割出网页中的有意义内容（与题头、脚注、边注等大量公式化内容不同）、移除残留其中的HTML标记并借助简单的文摘技

术分离出内容中最重要的句子。

自动文摘技术基于如下基本假设：按时序表述时，最重要的句子能够很好地总结内容，并且，你可以通过辨识那些频繁出现的搭配紧密的词来发现最重要的句子。尽管这种自动文摘做法有点粗糙，它却在相对规范的网页内容上出奇的好用。

示例9-24：总结链接目标的内容

```
   import sys
import json
import nltk
import numpy
import urllib2
from boilerpipe.extract import Extractor
def summarize(url, n=100, cluster_threshold=5, top_sentences=5):
    # Adapted from "The Automatic Creation of Literature Abstracts" by H.P. Luhn
    #
    # Parameters:
    # * n  - Number of words to consider
    # * cluster_threshold - Distance between words to consider
    # * top_sentences - Number of sentences to return for a "top n" summary
    # Begin - nested helper function
    def score_sentences(sentences, important_words):
        scores = []
        sentence_idx = -1
        for s in [nltk.tokenize.word_tokenize(s) for s in sentences]:
            sentence_idx += 1
            word_idx = []
            # For each word in the word list...
            for w in important_words:
                try:
                    # Compute an index for important words in each sentence
                    word_idx.append(s.index(w))
                except ValueError, e: # w not in this particular sentence
                    pass
            word_idx.sort()
            # It is possible that some sentences may not contain any important words
            if len(word_idx)== 0: continue
            # Using the word index, compute clusters with a max distance threshold
            # for any two consecutive words
            clusters = []
            cluster = [word_idx[0]]
            i = 1
            while i < len(word_idx):
                if word_idx[i] - word_idx[i - 1] < cluster_threshold:
                    cluster.append(word_idx[i])
                else:
```

```
                    clusters.append(cluster[:])
                    cluster = [word_idx[i]]
                i += 1
            clusters.append(cluster)
            # Score each cluster. The max score for any given cluster is the score
            # for the sentence.
            max_cluster_score = 0
            for c in clusters:
                significant_words_in_cluster = len(c)
                total_words_in_cluster = c[-1] - c[0] + 1
                score = 1.0 * significant_words_in_cluster \
                    * significant_words_in_cluster / total_words_in_cluster
                if score > max_cluster_score:
                    max_cluster_score = score
            scores.append((sentence_idx, score))
        return scores
    # End - nested helper function
    extractor = Extractor(extractor='ArticleExtractor', url=url)
    # It's entirely possible that this "clean page" will be a big mess. YMMV.
    # The good news is that the summarize algorithm inherently accounts for handling
    # a lot of this noise.
    txt = extractor.getText()
    sentences = [s for s in nltk.tokenize.sent_tokenize(txt)]
    normalized_sentences = [s.lower() for s in sentences]
    words = [w.lower() for sentence in normalized_sentences for w in
             nltk.tokenize.word_tokenize(sentence)]
    fdist = nltk.FreqDist(words)
    top_n_words = [w[0] for w in fdist.items()
            if w[0] not in nltk.corpus.stopwords.words('english')][:n]
    scored_sentences = score_sentences(normalized_sentences, top_n_words)
    # Summarization Approach 1:
    # Filter out nonsignificant sentences by using the average score plus a
    # fraction of the std dev as a filter
    avg = numpy.mean([s[1] for s in scored_sentences])
    std = numpy.std([s[1] for s in scored_sentences])
    mean_scored = [(sent_idx, score) for (sent_idx, score) in scored_sentences
                   if score > avg + 0.5 * std]
    # Summarization Approach 2:
    # Another approach would be to return only the top N ranked sentences
    top_n_scored = sorted(scored_sentences, key=lambda s: s[1])[-top_sentences:]
    top_n_scored = sorted(top_n_scored, key=lambda s: s[0])
    # Decorate the post object with summaries
    return dict(top_n_summary=[sentences[idx] for (idx, score) in top_n_scored],
                mean_scored_summary=[sentences[idx] for (idx, score) in mean_scored])
# Sample usage
sample_url = 'http://radar.oreilly.com/2013/06/phishing-in-facebooks-pond.html'
summary = summarize(sample_url)
print "-------------------------------------------------"
print "                'Top N Summary'"
print "-------------------------------------------------"
print " ".join(summary['top_n_summary'])
print
print
print "-------------------------------------------------"
print "             'Mean Scored' Summary"
print "-------------------------------------------------"
print " ".join(summary['mean_scored_summary'])
```

9.25 分析用户收藏的推文

9.25.1 问题

你想要通过检查一个人收藏的推文，了解更多关于这个人在意的见解。

9.25.2 解决方案

使用GET favorites/list API入口获取一个用户收藏的推文并随后使用技术手段来检测、提取和计算推文实体以刻画其内容。

9.25.3 讨论

考虑到并不是所有的Twitter用户以书签方式收藏喜欢的推文，所以你在专注于获得感兴趣内容和主题时，不能认为它是完全可靠的技术。但如果碰到了习惯把喜欢的推文予以收藏的Twitter用户，那你就太幸运了，你将常常发现井然有序内容的宝藏。尽管示例9-25给出一个基于之前配方来创建推文实体表的分析，你可以应用更高级的技术到推文自

身。一些思路可能包括把内容分离到不同主题、分析一个人的喜好是如何随时间变化或者演化的或者绘制出一个人在何时以及多久收藏一次推文等规律。

记住，除了收藏行为，所有被用户转推的推文也同样是有潜力的分析候选，甚至分析一些特殊行为模式，比如用户是否喜欢转推（以及转推频率）、收藏推文（以及收藏频率）或者两者都喜欢，本身就是很有启发意义的调研。

示例9-25：分析用户收藏的推文

```
    def analyze_favorites(twitter_api, screen_name, entity_threshold=2):
        # Could fetch more than 200 by walking the cursor as shown in other
        # recipes, but 200 is a good sample to work with.
        favs = twitter_api.favorites.list(screen_name=screen_name, count=200)
        print "Number of favorites:", len(favs)
        # Figure out what some of the common entities are, if any, in the content
        common_entities = get_common_tweet_entities(favs,
                                                    entity_threshold=entity_threshold)
        # Use PrettyTable to create a nice tabular display
        pt = PrettyTable(field_names=['Entity', 'Count'])
        [ pt.add_row(kv) for kv in common_entities ]
        pt.align['Entity'], pt.align['Count'] = 'l', 'r' # Set column alignment
        print
        print "Common entities in favorites..."
        print pt
        # Print out some other stats
        print
        print "Some statistics about the content of the favorities..."
        print
        analyze_tweet_content(favs)
        # Could also start analyzing link content or summarized link content, and more.
# Sample usage
twitter_api = oauth_login()
analyze_favorites(twitter_api, "ptwobrussell")
```

注意：查看http://favstar.fm中给出的一个意在帮助你找到“最佳推文”的流行网站示例，它是通过追踪和分析Twitter上那些正在被收藏和

转推的推文来实现的。

9.26 本章小结

相较于上百上千处理和挖掘Twitter数据的代码配方而言，这个实用指南真的只算乏善可陈，希望它为你提供了一个很好的跳板和一些想法的雏形，你可以在此基础上以多种形式好好吸取经验并善用它们。你可以利用Twitter数据（和大多数其他社交数据）做的事情是宽广、强大、充满趣味的。

注意：热烈欢迎和鼓励你拉取请求其他一些代码配方（以及对这些配方的增强），它的批准比较宽松。请从GitHub的资料库（http://bit.ly/1a1kNqy）派生本书的源代码、提交一个代码配方到本章的IPython Notbook并提交拉取请求！希望这个代码配方集合将不断增长、为社交数据挖掘者提供宝贵的切入点并围绕它形成一个充满活力的社区。

9.27 推荐练习

·深入检视Twitter平台API（http://bit.ly/1a1kSKQ）。是否有什么API的出现（或未出现）是让你感到惊奇的？

·分析所有你曾经转推的推文。你是否对你曾经转推的内容或者你的兴趣如何随时间演化的而大吃一惊？

·对比你原创的推文与你转推的推文。它们一般属于同一主题吗？

·写一个代码配方让它从MongoDB载入好友关系图数据并使用NetworkX转成一个真正的图谱表示，然后又采用NetworkX的内置算法（如中心度度量或团分析）进行图谱挖掘。第7章提供了NetworkX的概述，你会发现先复习这部分内容再完成本练习会大有裨益。

·写一个代码配方，改写前面章节的可视化代码，用于显示Twitter数据。例如，改变图谱可视化可以显示好友关系图，改写IPython Notebook以显示推文的模式或者某个用户的趋势，再或者使用推文内容填充一个标签云（如Word Cloud Layout，http://bit.ly/1a1n5pO）。

·写一个代码配方，根据推文内容辨识那些关注你又值得你去关注但你却没有关注的关注者。本书3.3.3节介绍了几个相似度度量，可能是不错的切入点。

·写一个代码配方来基于他们推文的内容计算两个用户的相似度。

·回顾Twitter的列表API，尤其是/lists/list（http://bit.ly/1a1pMrv）和/lists/memberships（http://bit.ly/1a1pOj5）API入口，它们分别告诉你一个用户订阅的列表和一个其成员已被其他用户添加了的列表。你能从订阅列表中获知用户的什么信息？你又能从被其他用户添加的列表获知用户的哪些信息呢？

·尝试运用技术来处理推文中的人类语言。卡耐基·梅隆有一个项目是跟Twitter的NLP和词性标注（http://bit.ly/1a1n84Y）有关的，它是一个很好的切入点。

·如果你关注许多Twitter账户，对你来说不太可能跟上所有的动态。写一个算法，按照推文的重要性而不是时间顺序排序在你的主页时间轴上。你是否能够有效滤除噪声并获得更强的信号？你可以根据你个人的兴趣对一天中排在最前面推文计算一个有意义的摘要？

·开始积累其他社交网站（如Facebook、LinkedIn、Google）的代码配方。

9.28 在线资源

下面链接的列表是本章有用的在线资源：

·BSON（http://bit.ly/1a1pG34）

·d3-cloud GitHub资源库（http://bit.ly/1a1n5pO）

·指数级衰减（http://bit.ly/1a1pHEe）

·MongoDB的数据聚合框架（http://bit.ly/1a1pGjv）

·OAuth 1.0a（http://bit.ly/1a1pBwg）

·Twitter API HTTP错误码（http://bit.ly/1a1pL6O）

·Twitter开发者API（http://bit.ly/1a1kSKQ）

·Twitter开发者问答（http://bit.ly/1a1pC3o）

·Twitter自然语言处理和词性标注（http://bit.ly/1a1n84Y）

·Twitter信息流API（http://bit.ly/1a1l1ya）

·Yahoo！Where On Earth（WOE）ID（http://yhoo.it/1a1kZ9u）

第三部分　附录

本书的附录展示了一些丰富的材料，它对前面很多内容作了扩展。

·附录A简要地概述了贯穿本书的虚拟机使用技术，并简要地讨论了虚拟机的使用范围及目的。

·附录B简短地讨论了开放授权（OAuth），它是一个工业协议，支持使用一个API从几乎所有著名的社交网站中读取社交数据。

·附录C是对你会在本书的源代码中遇到的一些常见的Python风格的简短描述；它强调了关于IPython Notebook的一些细节，了解它们会对你很有帮助。

附录A　关于本书虚拟机体验的信息

正如本书的每一章都有相应的IPython Notebook，每个附录也有相应的IPython Notebook。不论是什么目的，所有的Notebook都保存在本书的GitHub源代码库（http://bit.ly/1a1kNqy）中。你正在读的这个“打印版”附录可以与IPython Notebook相互参照，提供了关于如何安装和配置书中虚拟机的每一步操作。

建议你安装虚拟机作为开发环境，而不是用你已有的Python配置，因为有一些重要的配置管理问题涉及安装IPython Notebook和科学计算所需要的全部依赖。本书前后共使用了各种不同的第三方Python包以及为了支持不同平台用户，只会加剧基本环境搭建及运行的复杂度。因此，本书用虚拟机为所有读者和使用者提供源码，能够让大家在操作样例时遭遇最少的异常。即使你对Python开发工具非常熟悉，你仍然会想在第一次阅读本书的过程中通过利用本书的虚拟机来节省时间。试一下，你会很高兴使用它。

注意：附录A（http://bit.ly/1a1pSiI）的只读IPython Notebook保存在本书GitHub源代码库（http://bit.ly/1a1kNqy）中，并且包含了逐步的详细指导。

附录B　OAuth入门

正如本书的每一章都有相应的IPython Notebook，每个附录也有相应的IPython Notebook。不论是什么目的，所有的Notebook都保存在本书的GitHub源代码库（http://bit.ly/1a1kNqy）中。你正在读的这个“打印版”附录可以与IPython Notebook互相参照，IPython Notebook提供的代码示例展示了交互式OAuth的过程，它涉及了明确的用户授权，如果你要实现面向用户的应用就需要该授权。

本附录的剩下部分意在提供对OAuth的简明讨论。对于像Twitter、Facebook和LinkedIn这样流行网站的OAuth使用样例代码在相应的IPython Notebook中，可以在本书的源代码中找到。

注意：和其他附录一样，本附录也有相应的IPython Notebook，其名为附录B：OAuth入门（http://bit.ly/1a1pW1V），你可以在线阅读。

概述

OAuth表示“开放授权”，它通过一个API为用户提供了一个途径来授权应用程序去读取他们的账户数据，而不需要交出像用户名和密码这样敏感的认证信息。尽管OAuth被放在社交网络的背景下，记住，在任

何用户想要授权应用程序代表他们来进行具体操作的时候，该规范都可广泛使用。总的来说，用户可以控制第三方应用程序读取数据的权限（取决于供应商实现的API对权限的划分）并且可以在任何时候收回这个权限。例如，考虑Facebook的例子，该例中实现了极其细致的权限设定，让用户可以允许第三方应用程序访问十分具体的一些敏感账户信息。

考虑如Twitter、Facebook、LinkedIn和Google+这些大受欢迎的平台和在这些社交网络平台上开发的第三方应用的大量工具，不出所料，它们已经采用OAuth的途径来开放平台。然而，像任何其他的规范或协议一样，不同社交网络内容的OAuth实现在规范版本上存在区别，并且有时在一些特定的实现中会出现一些个人风格。本节剩余部分提供了OAuth 1.0a（如RFC 5849（http://bit.ly/1a1pWio）定义的）和OAuth 2.0（如RFC 6749（http://bit.ly/1a1pWiz）定义的）的简要概述，当你在挖掘社交网络和进行一些其他平台API编码时，你就会遇到OAuth。

OAuth 1.0a

OAuth 1.0[1]定义了一个协议，它可以让网络客户读取服务器上资源拥有者保护的资源，该协议在OAuth 1.0指南（http://bit.ly/1a1pYHe）中十分详尽的予以描述。正如你已经知道的，该协议的存在就是为了避免

用户（资源拥有者）将密码共享给网络应用的问题，尽管其在相当受限的范围内定义，但它在其声明的事情上做的确实很好。正如它所证明的，开发者对OAuth 1.0的一个主要抱怨（起初阻碍了协议的实施）是它实施起来冗长乏味，因为考虑到OAuth 1.0没有假定信任证书是使用HTTPS协议通过安全的SSL连接来进行交换的，它会涉及多种加密的细节（例如HMAC签名（http://bit.ly/1a1pZe1））。换句话说，OAuth 1.0用信息加密来保证通过电线传输过程中的安全性。

尽管我们在这里的讨论不够正式，但是你要知道在OAuth的语法中，请求读取数据的应用程序通常叫做客户（有时也叫做消费者），装载受保护资源的社交网站或服务叫做服务器（有时也叫做服务供应商），授权他人读取数据的用户叫做资源所有者。由于在这个协议中涉及3种角色，他们之间的一系列变化通常称为“三角流动”，或者更通俗地来说叫做“OAuth舞蹈”。尽管协议的实现及其安全的细节有些复杂，但实质上来看在OAuth舞蹈中只有一些基本的步骤，这些步骤最终可以使客户在服务器上代表资源所有者来读取受保护的数据：

1.客户从服务供应商获得未被授权的请求令牌。

2.资源所有者授权请求令牌。

3.客户用请求令牌换取访问令牌。

4.客户用访问令牌代表资源所有者读取受保护的资源。

就其详细的认证信息来说，客户从用户账号和密码开始，到“OAuth舞蹈”末尾获取访问令牌和令牌密码来读取受保护的资源结束。整体考虑，OAuth 1.0开始是使客户应用程序从资源所有者那里安全的获得授权，进而从服务器上读取用户的资源，尽管可能会有一些冗长的实现细节，但它提供了一个广泛接受的协议，能够很好地达到目的。OAuth 1.0可能暂时看起来够用了。

注意：“OAuth介绍”（http://bit.ly/1a1pXD7）阐述了终端用户（资源所有者）如何授权一个短网址的服务，例如bit.ly（客户）自动地将链接发布到Twitter上（服务供应商）。本节出现的抽象概念值得回味并深刻理解。

OAuth 2.0

鉴于OAuth 1.0可以对网络上的应用程序提供实用的授权流程（尽管范围有限），OAuth 2.0的初衷是打算通过在安全方面完全依靠SSL为网页应用开发者提供大大简化的实现细节，并满足更广的用例。这样的用例范围包括从支持移动设备到满足企业的需求，甚至在将来“物联网”的需求，例如可能出现在你家里的设备。

Facebook是其早期的使用者，它打算应用OAuth 2.0的计划要追溯到2011年OAuth 2.0的早期草稿，它也是很快完全依靠部分OAuth 2.0协议

的平台，而LinkedIn等到2013年初才开始支持OAuth 2.0（http://linkd.in/1a1q1mk）。尽管Twitter标准的基于用户的授权仍然是直接依靠OAuth 1.0a，但是它在2013年初实现了基于应用程序的授权（http://bit.ly/1a1q0Ph），该方法是在OAuth 2.0协议的客户证书授权（http://bit.ly/1a1q3KT）流程上建立的模型。最后，Google最近实现了OAuth 2.0的服务，如Google+（http://bit.ly/1a1q29X），并在从2012年4月起放弃支持OAuth 1.0。正如你所看到的，对OAuth 2.0的反应各有不同，并不是所有的社交网站都在OAuth 2.0发布的时候立即争抢着去实现。

OAuth 2.0是否会像起初设想的那样成为新的工业标准，现在仍旧不明朗。一个题为“OAuth 2.0and the Road to Hell”（http://bit.ly/1a1q4yr）的博客（它相应的Hacker News讨论（http://bit.ly/1a1q2Xg））总结了许多问题，值得阅读。该博客是由Eran Hammer所写，他在从事OAuth的工作多年后，自2012年中旬辞去OAuth 2.0协议的领导者和编辑的职位。看起来好像在大的开放性企业问题上，“设计委员会”模式打消了开发组的一些热情并阻碍了它的发展，尽管该协议在2012年末发布，关于它是否能够提供一个实际的说明或蓝图还不清楚。OAuth 1.0开发中，访问API是让开发者感到痛苦的，幸运的是，在过去的几年中，出现了许多极好的OAuth框架，缓解大多数的痛苦。尽管受OAuth 1.0初始的牵绊，开发者仍旧坚持创新。对于本书前面章节所使用的Python包，你不需要知道或者关心任何涉及

OAuth 1.0a实现的复杂细节；你只需要明白它工作的要旨。然而，尽管我们在OAuth 2.0上裹足不前和行动迟缓，但能清楚地看到的是它的几个流程看起来已经定义良好，以至于大的社交网络供应商正在趋向于使用它们。

正如你现在知道的，OAuth 1.0含有相当严格的实现步骤，与此不同，OAuth 2.0的实现根据特定的用例可以有些不同。然而，一个典型的OAuth 2.0流程确实利用了SSL，并且在足够高的层面上实质只包含一些改变，这些改变和我们之前提到的涉及OAuth 1.0流程的一系列步骤看来也差不了太多。例如，Twitter最近的应用程序唯一授权（http://bit.ly/1a1q0Ph）仅仅涉及应用程序用其用户账号和密码来换取安全的SSL链接的访问令牌。重申一下，其实现步骤会根据用例情况而变化，并且尽管其实现不是那么容易读懂，但是如果你对它的一些细节感兴趣，OAuth 2.0规范第4节（http://bit.ly/1a1q3uv）是相当易懂的。回顾该内容时，只要记住OAuth 1.0和OAuth 2.0的一些术语是不同的，因此专注于理解一个规范会比同时学习两个更容易些。

注意：Jonathan LeBlanc所写的《Programming Social Applications》（O'Reilly）一书在第9章针对建立社交网络应用需求很好地讨论了OAuth 1.0和OAuth 2.0。

OAuth的特性，OAuth 1.0和OAuth 2.0潜在的实现流程对于进行社交网络挖掘的你来说并不是那么重要。这里只是简单地讨论了一些相关的

背景，这样你就对涉及的一些关键概念有一个基本的理解，并且如果你想进一步学习和研究，这些内容是不错的起点。正如你可能已经知道的，细节才是真正难的。幸运的是，好用的第三方函数库大大降低了我们要对那些细节知晓的程度，尽管有时这些细节迟早会用到。本附录的在线代码既有关于OAuth 1.0流程的也有关于OAuth 2.0流程的，你可以用它们来钻研所有你想知道的细节。

[1] 在本次讨论中，考虑到OAuth 1.0修订版A淘汰了OAuth 1.0并已成为广泛采用的标准，我们所用“OAuth 1.0”一词技术上来讲就是指“OAuth 1.0a”的意思。

附录C　Python和IPython Notebook的使用技巧

正如本书的每一章都有相应的IPython Notebook，每一个附录也有相应的IPython Notebook。如附录A，这个“打印版”附录可以用来与IPython Notebook互相参照，IPython Notebook保存在本书的GitHub源代码库（http://bit.ly/1a1kNqy）中，它还包括了Python的一些常用语法和一些使用IPython Notebook的使用技巧。

注意：本附录（http://bit.ly/1a1q6GI）的IPython Notebook包括了附加的常见Python用法的例子，这对你的学习是很有用的。它还包括了一些使用IPython Notebook时有用的技巧，这会帮你节省一些时间。

即使经常听到Python是“可执行的伪代码”，但是对于不熟悉Python的读者来说，将Python作为通用编程语言来简单地回顾一下会很有帮助。如果你感觉会从Python编程语言的通用介绍中受益，请考虑依照Python教程（http://bit.ly/1a1q6Xe）前8节学习。Python值得你花时间进行学习，它会让你更好地体验本书学习的乐趣。

作者简介

Matthew Russell（@ptwobrussell），Digital Reasoning公司（http://www.digital reasoning.com/）首席技术官（CTO）、Zaffra公司负责人。他同时是多本技术书籍的作者。他热衷于开源软件开发、数据挖掘和创造技术以扩展人类智能。Matthew学习计算机科学出身，毕业于美国空军学院。除了解决技术难题，他喜欢练习比克拉姆式热瑜伽、全面健身（CrossFit）并参加铁人三项运动。

封面介绍

本书封面上的动物是土拨鼠（学名Marmota monax），又称美洲旱獭（woodchuck，该名字源自其Algonquin叫法“wuchak”）。土拨鼠与美国、加拿大2月2日的土拨鼠节相关。民俗认为，如果这一天土拨鼠从它的洞里出来，而且能看到它的影子，那么冬天还会持续6周。该说法的支持者说，啮齿动物预测的准确性为75%~90%。很多城市中都有著名的土拨鼠天气预报员，包括美国宾夕法尼亚州庞克瑟托克小镇上的菲尔（比尔·默瑞于1993年在电影《土拨鼠节》中所扮演的主角）。

这个传说可能源于土拨鼠是整个冬季都冬眠的少数几个物种之一。作为食草动物，土拨鼠主要依靠植物、浆果、坚果、昆虫和人类园林作物在夏天贮存脂肪，这也导致很多人认为它们属于害虫。它们会在冬天挖洞穴，从10月到来年的3月它们一直待在里面（但在温带地区，它们可能会早点出来，如果它们早出来，将成为土拨鼠节的关注焦点）。

土拨鼠是松鼠家族中最大的成员，长大约16~26英寸，重4~9磅。它们带弯的厚爪子是理想的挖掘工具；它们有两层皮毛为自身提供保护，内层为密集的灰色绒毛，外层则是更长的浅色毛发。

土拨鼠遍及加拿大大部分地区和美国北部地区，生活在开阔的空间和林地丰富的地方。它们能爬树和游泳，但经常生活在离洞穴不远的地

面上，洞穴用来睡觉、养育后代并保护自己。这些洞穴通常有2~5个入口，通常是长达46英尺的地道。