

使用Python高效爬取页面数据的艺术

精通Python 爬虫框架Scrapy

Learning Scrapy

[美] 迪米特里奥斯 考奇斯-劳卡斯 (Dimitrios Kouzis-Loukas) 著
李斌 译



精通Python 爬虫框架Scrapy

〔美〕迪米特里奥斯 考奇斯-劳卡斯 (Dimitrios Kouzis-Loukas) 著
李斌 译

人民邮电出版社
北京

异步社区会员 Kensuke(18319066421) 专享 尊重版权

本书配套资源获取方式

方法一：



登录本书配套资源网站异步社区www.epubit.com.cn，在搜索栏搜索你想要下载图书书名，进入图书详情页，点击“配套文件下载”即可。

方法二：



关注微信公众号【异步社区】，在后台回复你想要下载的图书书名，按照公众号下载指引操作即可。

图书在版编目（C I P）数据

精通Python爬虫框架Scrapy / (美) 迪米特里奥斯·考奇斯-劳卡斯著；李斌译. -- 北京：人民邮电出版社，2018.2

ISBN 978-7-115-47420-9

I. ①精… II. ①迪… ②李… III. ①软件工具—程序设计 IV. ①TP311.561

中国版本图书馆CIP数据核字(2017)第305803号

版 权 声 明

Copyright © Packt Publishing 2016. First published in the English language under the title Learning Scrapy.
All Rights Reserved.

本书由英国 **Packt Publishing** 公司授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有，侵权必究。

-
- ◆ 著 [美]迪米特里奥斯 考奇斯-劳卡斯
(Dimitrios Kouzis-Loukas)
- 译 李 斌
- 责任编辑 傅道坤
- 责任印制 焦志炜
- ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京市艺辉印刷有限公司印刷
- ◆ 开本：800×1000 1/16
印张：16
字数：225 千字 2018 年 2 月第 1 版
印数：1 - 3 000 册 2018 年 2 月北京第 1 次印刷
- 著作权合同登记号 图字：01-2017-2291 号
-

定价：59.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

内容提要

Scrapy 是使用 Python 开发的一个快速、高层次的屏幕抓取和 Web 抓取框架，用于抓取 Web 站点并从页面中提取结构化的数据。本书以 Scrapy 1.0 版本为基础，讲解了 Scrapy 的基础知识，以及如何使用 Python 和三方 API 提取、整理数据，以满足自己的需求。

本书共 11 章，其内容涵盖了 Scrapy 基础知识，理解 HTML 和 XPath，安装 Scrapy 并爬取一个网站，使用爬虫填充数据库并输出到移动应用中，爬虫的强大功能，将爬虫部署到 Scrapinghub 云服务器，Scrapy 的配置与管理，Scrapy 编程，管道秘诀，理解 Scrapy 性能，使用 Scrapyd 与实时分析进行分布式爬取。本书附录还提供了各种必备软件的安装与故障排除等内容。

本书适合软件开发人员、数据科学家，以及对自然语言处理和机器学习感兴趣的人阅读。

关于作者

Dimitrios Kouzis-Loukas 作为一位顶级的软件开发人员，已经拥有超过 15 年的经验。同时，他还使用自己掌握的知识和技能，向广大读者讲授如何编写优秀的软件。

他学习并掌握了多门学科，包括数学、物理学以及微电子学。他对这些学科的透彻理解，提高了自身的标准，而不只是“实用的解决方案”。他知道真正的解决方案应当是像物理学规律一样确定，像 ECC 内存一样健壮，像数学一样通用。

Dimitrios 目前正在使用最新的数据中心技术开发低延迟、高可用的分布式系统。他是语言无关论者，不过对 Python、C++ 和 Java 略有偏好。他对开源软硬件有着坚定的信念，他希望他的贡献能够造福于各个社区和全人类。

关于审稿人

Lazar Telebak 是一位自由的 Web 开发人员，专注于使用 Python 库/框架进行网络爬取和对网页进行索引。

他主要从事于处理自动化和网站爬取以及导出数据到不同格式（包括 CSV、JSON、XML 和 TXT）和数据库（如 MongoDB、SQLAlchemy 和 Postgres）的项目。

他还拥有前端技术和语言的经验，包括 HTML、CSS、JS 和 jQuery。

前言

让我来做一个大胆的猜测。下面的两个故事之一会和你的经历有些相似。

你与 Scrapy 的第一次相遇是在网上搜索类似“Web scraping Python”的内容时。你快速对其进行了浏览，然后想“这太复杂了吧……我只需要一些简单的东西。”接下来，你使用 Requests 库开发了一个 Python 脚本，并且挣扎于 BeautifulSoup 中，但最终还是完成了很酷的工作。它有些慢，所以你让它整夜运行。你重新启动了几次，忽略了一些不完整的链接和非英文字符，到早上的时候，大部分网站已经“骄傲地”存在你的硬盘中了。然而难过的是，不知什么原因，你不想再看到自己写的代码。当你下一次再想抓取某些东西时，则会直接前往 scrapy.org，而这一次文档给了你很好的印象。现在你可以感受到 Scrapy 能够以优雅且轻松的方式解决了你面临的所有问题，甚至还考虑到了你没有想到的问题。你不会再回头了。

另一种情况是，你与 Scrapy 的第一次相遇是在进行网络爬取项目的研究时。你需要的是健壮、快速的企业级应用，而大部分花哨的一键式网络爬取工具无法满足需求。你希望它简单，但又有足够的灵活性，能够让你为不同源定制不同的行为，提供不同的输出类型，并且能够以自动化的形式保证 24/7 可靠运行。提供爬取服务的公司似乎太贵了，你觉得使用开源解决方案比固定供应商更加舒服。从一开始，Scrapy 就像一个确定的赢家。

无论你是出于何种目的选择了本书，我都很高兴能够在这本专注于 Scrapy 的图书中遇到你。Scrapy 是全世界爬虫专家的秘密。他们知道如何使用它以节省工作时间，提供出色的性能，并且使他们的主机费用达到最低限度。如果你没有太多经验，但是还想实现同样的结果，那么很不幸的是，Google 并没有能够帮到你。网络上大多数 Scrapy 信息

要么太简单低效，要么太复杂。对于那些想要了解如何充分利用 Scrapy 找到准确、易理解且组织良好的信息的人们来说，本书是非常有必要的。我希望本书能够帮助 Scrapy 社区进一步发展，并使其得以广泛应用。

本书内容

第 1 章，Scrapy 简介，介绍本书和 Scrapy，可以让你对该框架及本书剩余部分有一个明确的期望。

第 2 章，理解 HTML 和 XPath，旨在使爬虫初学者能够快速了解 Web 相关技术以及我们后续将会使用的技巧。

第 3 章，爬虫基础，介绍了如何安装 Scrapy，并爬取一个网站。我们通过向你展示每一个行动背后的方法和思路，逐步开发该示例。学习完本章之后，你将能够爬取大部分简单的网站。

第 4 章，从 Scrapy 到移动应用，展示了如何使用我们的爬虫填充数据库并输出给移动应用。本章过后，你将清晰地认识到爬虫在市场方面所带来的好处。

第 5 章，迅速的爬虫技巧，展示了更强大的爬虫功能，包括登录、更快速地抓取、消费 API 以及爬取 URL 列表。

第 6 章，部署到 Scrapinghub，展示了如何将爬虫部署到 Scrapinghub 的云服务器中，并享受其带来的可用性、易部署以及可控性等特性。

第 7 章，配置与管理，以组织良好的表现形式介绍了大量的 Scrapy 功能，这些功能可以通过 Scrapy 配置启用或调整。

第 8 章，Scrapy 编程，通过展示如何使用底层的 Twisted 引擎和 Scrapy 架构对其功能的各个方面进行扩展，将我们的知识带入一个全新的水平。

第 9 章，管道秘诀，提供了许多示例，在这里我们修改了 Scrapy 的一些功能，在不会造成性能退化的情况下，将数据插入到数据库（比如 MySQL、Elasticsearch 及 Redis）、

接口 API，以及遗留应用中。

第 10 章，理解 Scrapy 性能，将帮助我们理解 Scrapy 的时间是如何花费的，以及我们需要怎么做来提升其性能。

第 11 章，使用 Scrapyd 与实时分析进行分布式爬取，这是本书最后一章，展示了如何在多台服务器中使用 Scrapyd 实现横向扩展，以及如何将爬取得到的数据提供给 Apache Spark 服务器以执行数据流分析。

阅读本书的前提

为了使本书代码和内容的受众尽可能广泛，我们付出了大量的努力。我们希望提供涉及多服务器和数据库的有趣示例，不过我们并不希望你必须完全了解如何创建它们。我们使用了一个称为 Vagrant 的伟大技术，用于在你的计算机中自动下载和创建一次性的多服务器环境。我们的 Vagrant 配置在 Mac OS X 和 Windows 上时使用了虚拟机，而在 Linux 上则是原生运行。

对于 Windows 和 Mac OS X，你需要一个支持 Intel 或 AMD 虚拟化技术（VT-x 或 AMD-v）的 64 位计算机。大多数现代计算机都没有问题。对于大部分章节来说，你还需要专门为虚拟机准备 1GB 内存，不过在第 9 章和第 11 章中则需要 2GB 内存。附录 A 讲解了安装必要软件的所有细节。

Scrapy 本身对硬件和软件的需求更加有限。如果你是一位有经验的读者，并且不想使用 Vagrant，也可以根据第 3 章的内容在任何操作系统中安装 Scrapy，即使其内存十分有限。

当你成功创建 Vagrant 环境后，无需网络连接，就可以运行本书几乎全部示例了（第 4 章和第 6 章的示例除外）。是的，你可以在航班上阅读本书了。

本书读者

本书尝试着去适应广泛的读者群体。它可能适合如下人群：

- 需要源数据驱动应用的互联网创业者；
- 需要抽取数据进行分析或训练模型的数据科学家与机器学习从业者；
- 需要开发大规模爬虫基础架构的软件工程师；
- 想要为其下一个很酷的项目在树莓派上运行 **Scrapy** 的爱好者。

就必备知识而言，阅读本书只需要用到很少的部分。在最开始的几章中，本书为那些几乎没有爬虫经验的读者提供了网络技术和爬虫的基础知识。**Python** 易于阅读，对于有其他编程语言基本经验的任何读者来说，与爬虫相关的章节中给出的大部分代码都易于理解。

坦率地说，我相信如果一个人心中有一个项目，并且想使用 **Scrapy** 的话，他就能够修改本书中的示例代码，并在几个小时之内良好地运行起来，即使这个人之前没有爬虫、**Scrapy** 或 **Python** 经验。

在本书的后半部分中，我们将变得更加依赖于 **Python**，此时初学者可能希望在进一步研究之前，先让自己用几个星期的时间丰富 **Scrapy** 的基础经验。此时，更有经验的 **Python/Scrapy** 开发者将学习使用 **Twisted** 进行事件驱动的 **Python** 开发，以及非常有趣的 **Scrapy** 内部知识。在性能章节，一些数学知识可能会有用处，不过即使没有，大多数图表也能给我们清晰的感受。

目录

第 1 章	Scrapy 简介	1
1.1	初识 Scrapy	1
1.2	喜欢 Scrapy 的更多理由	2
1.3	关于本书：目标和用途	3
1.4	掌握自动化数据爬取的重要性	4
1.4.1	开发健壮且高质量的应用，并提供合理规划	4
1.4.2	快速开发高质量最小可行产品	5
1.4.3	Google 不会使用表单，爬取才能扩大规模	6
1.4.4	发现并融入你的生态系统	7
1.5	在充满爬虫的世界里做一个好公民	7
1.6	Scrapy 不是什么	8
1.7	本章小结	9
第 2 章	理解 HTML 和 XPath	10
2.1	HTML、DOM 树表示以及 XPath	10
2.1.1	URL	11
2.1.2	HTML 文档	11
2.1.3	树表示法	13
2.1.4	你会在屏幕上看到什么	14
2.2	使用 XPath 选择 HTML 元素	15
2.2.1	有用的 XPath 表达式	16

2.2.2	使用 Chrome 获取 XPath 表达式	19
2.2.3	常见任务示例	20
2.2.4	预见变化	21
2.3	本章小结	22
第 3 章	爬虫基础	23
3.1	安装 Scrapy	24
3.1.1	MacOS	24
3.1.2	Windows	25
3.1.3	Linux	25
3.1.4	最新源码安装	26
3.1.5	升级 Scrapy	26
3.1.6	Vagrant : 本书中运行示例的官方方式	27
3.2	UR²IM ——基本抓取流程	28
3.2.1	URL	29
3.2.2	请求和响应	31
3.2.3	Item	31
3.3	一个 Scrapy 项目	37
3.3.1	声明 item	38
3.3.2	编写爬虫	40
3.3.3	填充 item	43
3.3.4	保存文件	45
3.3.5	清理—— item 装载器与管理字段	47
3.3.6	创建 contract	50
3.4	抽取更多的 URL	53
3.4.1	使用爬虫实现双向爬取	56
3.4.2	使用 CrawlSpider 实现双向爬取	59
3.5	本章小结	61

第 4 章 从 Scrapy 到移动应用	62
4.1 选择手机应用框架	62
4.2 创建数据库和集合	63
4.3 使用 Scrapy 填充数据库	65
4.4 创建手机应用	68
4.4.1 创建数据库访问服务	69
4.4.2 创建用户界面	69
4.4.3 将数据映射到用户界面	70
4.4.4 数据库字段与用户界面控件间映射	71
4.4.5 测试、分享及导出你的手机应用	72
4.5 本章小结	73
第 5 章 迅速的爬虫技巧	75
5.1 需要登录的爬虫	75
5.2 使用 JSON API 和 AJAX 页面的爬虫	81
5.3 30 倍速的房产爬虫	85
5.4 基于 Excel 文件爬取的爬虫	90
5.5 本章小结	93
第 6 章 部署到 Scrapinghub	94
6.1 注册、登录及创建项目	94
6.2 部署爬虫与计划运行	96
6.3 访问 item	99
6.4 计划定时爬取	100
6.5 本章小结	101
第 7 章 配置与管理	102
7.1 使用 Scrapy 设置	102
7.2 基本设置	103

7.2.1	分析	104
7.2.2	性能	107
7.2.3	提前终止爬取	108
7.2.4	HTTP 缓存和离线运行	108
7.2.5	爬取风格	109
7.2.6	feed	110
7.2.7	媒体下载	111
7.2.8	Amazon Web 服务	113
7.2.9	使用代理和爬虫	113
7.3	进阶设置	114
7.3.1	项目相关设置	115
7.3.2	Scrapy 扩展设置	116
7.3.3	下载调优	116
7.3.4	自动限速扩展设置	117
7.3.5	内存使用扩展设置	117
7.3.6	日志和调试	117
7.4	本章小结	118
第 8 章	Scrapy 编程	119
8.1	Scrapy 是一个 Twisted 应用	119
8.1.1	延迟和延迟链	122
8.1.2	理解 Twisted 和非阻塞 I/O——一个 Python 故事	125
8.2	Scrapy 架构概述	132
8.3	示例 1: 非常简单的管道	135
8.4	信号	136
8.5	示例 2: 测量吞吐量和延时的扩展	138
8.6	中间件延伸	141
8.7	本章小结	144

第 9 章 管道秘诀	145
9.1 使用 REST API	146
9.1.1 使用 <code>treq</code>	146
9.1.2 用于写入 <code>Elasticsearch</code> 的管道	146
9.1.3 使用 <code>Google Geocoding API</code> 实现地理编码的管道	149
9.1.4 在 <code>Elasticsearch</code> 中启用地理解码索引	156
9.2 与标准 Python 客户端建立数据库接口	157
9.3 使用 Twisted 专用客户端建立服务接口	161
9.4 为 CPU 密集型、阻塞或遗留功能建立接口	166
9.4.1 处理 CPU 密集型或阻塞操作的管道	166
9.4.2 使用二进制或脚本的管道	168
9.5 本章小结	172
第 10 章 理解 Scrapy 性能	173
10.1 Scrapy 引擎——一种直观方式	173
10.1.1 级联队列系统	175
10.1.2 定义瓶颈	176
10.1.3 Scrapy 性能模型	176
10.2 使用 telnet 获得组件利用率	178
10.3 基准系统	180
10.4 标准性能模型	182
10.5 解决性能问题	185
10.5.1 案例 #1: CPU 饱和	185
10.5.2 案例 #2: 代码阻塞	187
10.5.3 案例 #3: 下载器中的“垃圾”	188
10.5.4 案例 #4: 大量响应或超长响应造成的溢出	191
10.5.5 案例 #5: 有限/过度 item 并发造成的溢出	193
10.5.6 案例 #6: 下载器未充分利用	194

10.6	故障排除流程	197
10.7	本章小结	198
第 11 章	使用 Scrapy 与实时分析进行分布式爬取	199
11.1	房产的标题是如何影响价格的	200
11.2	Scrapy	200
11.3	分布式系统概述	203
11.4	爬虫和中间件的变化	205
11.4.1	索引页分片爬取	205
11.4.2	分批爬取 URL	207
11.4.3	从设置中获取初始 URL	211
11.4.4	在 Scrapy 服务器中部署项目	213
11.5	创建自定义监控命令	215
11.6	使用 Apache Spark 流计算偏移量	216
11.7	运行分布式爬取	218
11.8	系统性能	220
11.9	关键点	221
11.10	本章小结	221
附录 A	必备软件的安装与故障排除	222

第 1 章

Scrapy 简介

欢迎来到你的 Scrapy 之旅。通过本书，我们旨在将你从一个只有很少经验甚至没有经验的 Scrapy 初学者，打造成拥有信心使用这个强大的框架从网络或者其他源爬取大数据集的 Scrapy 专家。本章将介绍 Scrapy，并且告诉你一些可以用它实现的很棒的事情。

1.1 初识 Scrapy

Scrapy 是一个健壮的网络框架，它可以从各种数据源中抓取数据。作为一个普通的网络用户，你会发现自己经常需要从网站上获取数据，使用类似 Excel 的电子表格程序进行浏览（参见第 3 章），以便离线访问数据或者执行计算。而作为一个开发者，你需要经常整合多个数据源的数据，但又十分清楚获得和抽取数据的复杂性。无论难易，Scrapy 都可以帮助你完成数据抽取的行动。

以健壮而又有效的方式抽取大量数据，Scrapy 已经拥有了多年经验。使用 Scrapy，你只需一个简单的设置，就能完成其他爬虫框架中需要很多类、插件和配置项才能完成的工作。快速浏览第 7 章，你就能体会到通过简单的几行配置，Scrapy 可以实现多少功能。

从开发者的角度来说，你也会十分欣赏 Scrapy 的基于事件的架构（我们将在第 8 章和第 9 章中对其进行深入探讨）。它允许我们将数据清洗、格式化、装饰以及将这些数据存储到数据库中等操作级联起来，只要我们操作得当，性能降低就会很小。在本书中，

你将学会怎样可以达到这一目的。从技术上讲，由于 Scrapy 是基于事件的，这就能够让我们在拥有上千个打开的连接时，可以通过平稳的操作拆分吞吐量的延迟。来看这样一个极端的例子，假设你需要从一个拥有汇总页的网站中抽取房源，其中每个汇总页包含 100 个房源。Scrapy 可以非常轻松地在该网站中并行执行 16 个请求，假设完成一个请求平均需要花费 1 秒钟的时间，你可以每秒爬取 16 个页面。如果将其与每页的房源数相乘，可以得出每秒将产生 1600 个房源。想象一下，如果每个房源都必须在大规模并行云存储当中执行一次写入，每次写入平均需要耗费 3 秒钟的时间（非常差的主意）。为了支持每秒 16 个请求的吞吐量，就需要我们并行运行 $1600 \times 3 = 4800$ 次写入请求（你将在第 9 章中看到很多这样有趣的计算）。对于一个传统的多线程应用而言，则需要转变为 4800 个线程，无论是对你，还是对操作系统来说，这都会是一个非常糟糕的体验。而在 Scrapy 的世界中，只要操作系统没有问题，4800 个并发请求就能够处理。此外，Scrapy 的内存需求和你需要的房源数据量很接近，而对于多线程应用而言，则需要为每个线程增加与房源大小相比十分明显的开销。

简而言之，缓慢或不可预测的网站、数据库或远程 API 都不会对 Scrapy 的性能产生毁灭性的结果，因为你可以并行运行多个请求，并通过单一线程来管理它们。这意味着更低的主机托管费用，与其他应用的协作机会，以及相比于传统多线程应用而言更简单的代码（无同步需求）。

1.2 喜欢 Scrapy 的更多理由

Scrapy 已经拥有超过 5 年的历史了，成熟而又稳定。除了上一节中提到的性能优势外，还有下面这些能够让你爱上 Scrapy 的理由。

- Scrapy 能够识别残缺的 HTML

你可以在 Scrapy 中直接使用 Beautiful Soup 或 lxml，不过 Scrapy 还提供了一种在 lxml 之上更高级的 XPath（主要）接口——**selectors**。它能够更高效地处理残缺的 HTML 代码和混乱的编码。

- 社区

Scrapy 拥有一个充满活力的社区。只需要看看 <https://groups.google.com/forum/#!forum/scrapy-users> 上的邮件列表，以及 Stack Overflow 网站 (<http://stackoverflow.com/questions/tagged/scrapy>) 中的上千个问题就可以知道了。大部分问题都能够在几分钟内得到回应。更多社区资源可以从 <http://scrapy.org/community/> 中获取到。

- 社区维护的组织良好的代码

Scrapy 要求以一种标准方式组织你的代码。你只需编写被称为爬虫和管道的少量 Python 模块，并且还会自动从引擎自身获取到未来的任何改进。如果你在网上搜索，可以发现有多数专业人士拥有 Scrapy 经验。也就是说，你可以很容易地找到人来维护或扩展你的代码。无论是谁加入你的团队，都不需要漫长的学习曲线，来理解你的自定义爬虫中的特别之处。

- 越来越多的高质量功能

如果你快速浏览发布日志 (<http://doc.scrapy.org/en/latest/news.html>)，就会注意到无论是在功能上，还是在稳定性/bug 修复上，Scrapy 都在不断地成长。

1.3 关于本书：目标和用途

在本书中，我们的目标是通过重点示例和真实数据集教你使用 Scrapy。大部分章节将专注于爬取一个示例的房屋租赁网站。我们选择这个例子，是因为它能够代表大多数的网站爬取项目，既能让我们介绍感兴趣的变动，又不失简单。以该示例为主题，可以帮助我们聚焦于 Scrapy，而不会分心。

我们将从只运行几百个页面的小爬虫开始，最终在第 11 章中使用几分钟的时间，将其扩展为能够处理 5 万个页面的分布式爬虫。在这个过程中，我们将向你介绍如何将 Scrapy 与 MySQL、Redis 和 Elasticsearch 等服务相连接，使用 Google 的地理编码 API 找到我们示例属性中的位置坐标，以及向 Apache Spark 提供数据用于预测最影响房价

的关键词。

你需要做好阅读本书多次的准备。你可能需要从略读开始，先理解其架构。然后阅读一到两章，仔细学习、实验一段时间，再进入后面的章节。如果你觉得自己已经熟悉了某一章的内容，那么跳过这一章也无需担心。尤其是如果你已经了解 HTML 和 XPath，那么就没有必要花费太多时间在第 2 章上面了。不用担心，对你来说本书还有很多需要学习的内容。一些章节，比如第 8 章，将参考书和教程的元素结合起来，深入编程概念。这就是一个例子，我们可能会阅读某一章几次，在这中间允许我们有几个星期的时间实践 Scrapy。你在继续阅读后续的章节，比如以应用为主第 9 章之前，不需要完美掌握第 8 章中的内容。阅读后续的内容，有助于你理解如何使用编程概念，如果你愿意的话，可以回过头来反复阅读几次。

为使本书既有趣，又对初学者友好，我们已经试图做了平衡。不过我们不会做的一件事情是，在本书中教授 Python。对于这一主题，目前已经有了很多优秀的书籍，不过我更加建议的是以一种轻松的心态来学习。Python 如此流行的一个原因是因为它比较简单、整洁，并且阅读起来更近似于英文。Scrapy 是一个高级框架，无论是初学者还是专家，都需要学习。你可以将其称之为“Scrapy 语言”。因此，我会推荐你通过材料来学习 Python，如果你发觉自己对于 Python 的语法比较迷惑，那么可以通过一些 Python 的在线教程或 Coursera 等为 Python 初学者开设的免费在线课程予以补充。请放心，即使你不是 Python 专家，也能够成为一名优秀的 Scrapy 开发者。

1.4 掌握自动化数据爬取的重要性

对于大多数人来说，掌握一门像 Scrapy 这样很酷的技术所带来的好奇心和精神上的满足，足以激励我们。令人惊喜的是，在学习这个优秀框架的同时，我们还能享受到开发过程始于数据和社区，而不是代码所带来的好处。

1.4.1 开发健壮且高质量的应用，并提供合理规划

为了开发现代化的高质量应用，我们需要真实的大数据集，如果可能的话，在开始

动手写代码之前就应该进行这一步。现代化软件开发就是实时处理大量不完善数据，并从中提取出知识和有价值的情报。当我们开发软件并应用于大数据集时，一些小的错误和疏忽难以被检测出来，就有可能导致昂贵的错误决策。比如，在做人口统计学研究时，很容易发生仅仅是由于州名过长导致数据被默认丢弃，造成整个州的数据被忽视的错误。在开发阶段，甚至更早的设计探索阶段，通过细心抓取，并使用具有生产质量的真实世界大数据集，可以帮助我们发现和修复错误，做出明智的工程决策。

另外一个例子是，假设你想要设计 Amazon 风格的“如果你喜欢这个商品，也可能喜欢那个商品”的推荐系统。如果你能够在开始之前，先爬取并收集真实世界的数据集，就会很快意识到有关无效条目、停产商品、重复、无效字符以及偏态分布引起的性能瓶颈等问题。这些数据将会强迫你设计足够健壮算法，无论是数千人购买过的商品，还是零销售量的新条目，都能够很好地处理。而孤立的软件开发，可能会在几个星期的开发之后，也要面对这些丑陋的真实世界数据。虽然这两种方法最终可能会收敛，但是为你提供进度预估承诺的能力以及软件的质量，都将随着项目进展而产生显著差别。从数据开始，能够带给我们更加愉悦并且可预测的软件开发体验。

1.4.2 快速开发高质量最小可行产品

对于初创公司而言，大规模真实数据的集甚至更加必要。你可能听说过“精益创业”，这是由 *Eric Ries* 创造的一个术语，用于描述类似技术初创公司这样极端不确定条件下的业务发展过程。该框架的一个关键概念是最小可行产品（**Minimum Viable Product, MVP**），这种产品只有有限的功能，可以被快速开发并向有限的客户发布，用于测试反响及验证业务假设。基于获得的反馈，初创公司可能会选择继续更进一步的投入，也可能是转向其他更有前景的方向。

在该过程中的某些方面，很容易忽视与数据紧密连接的问题，这正是 *Scrapy* 所能为我们做的部分。比如，当邀请潜在的客户尝试使用我们的手机应用时，作为开发者或企业主，会要求他们评判这些功能，想象应用在完成时看起来应该如何。对于这些并非专家的人而言，这里需要的想象有可能太多了。这个差距相当于一个应用只展示了“产品 1”、“产品 2”、“用户 433”，而另一个应用提供了“三星 UN55J6200 55 英寸电视机”、

用户“Richard S”给出了五星好评以及能够让你直达产品详情页面（尽管事实上我们还没有写这个页面）的有效链接等诸多信息。人们很难客观判断一个 MVP 产品的功能性，除非使用了真实且令人兴奋的数据。

一些初创企业将数据作为事后考虑的原因之一是认为收集这些数据需要昂贵的代价。的确，我们通常需要开发表单及管理界面，并花费时间录入数据，但我们也可以在编写代码之前使用 Scrapy 爬取一些网站。在第 4 章中，你可以看到一旦拥有了数据，开发一个简单的手机应用会有多么容易。

1.4.3 Google 不会使用表单，爬取才能扩大规模

当谈及表单时，让我们来看下它是如何影响产品增长的。想象一下，如果 Google 的创始人在创建其引擎的第一个版本时，包含了一个每名网站管理员都需要填写的表单，要求他们把网站中每一页的文字都复制粘贴过来。然后，他们需要接受许可协议，允许 Google 处理、存储和展示他们的内容，并剔除大部分广告利润。你能想象解释该想法并说服人们参与这一过程所需花费的时间和精力会有多大吗？即使市场非常渴望一个优秀的搜索引擎（事实正是如此），这个引擎也不会是 Google，因为它的增长过于缓慢。即使是最复杂的算法，也不能弥补数据的缺失。Google 使用网络爬虫技术，在页面间跳转链接，填充其庞大的数据库。网站管理员则不需要做任何事情。实际上，反而还需要一些努力才能阻止 Google 索引你的页面。

虽然 Google 使用表单的想法听起来有些荒谬，但是一个典型的网站需要用户填写多少表单呢？登录表单、新房源表单、结账表单，等等。这些表单中有多少会阻碍应用增长呢？如果你充分了解你的受众/客户，很可能已经拥有关于他们通常使用并且很可能已经有账号的其他网站的线索了。比如，一个开发者很可能拥有 Stack Overflow 和 GitHub 的账号。那么，在获得他们允许的情况下，你是否能够抓取这些站点，只需他们提供给你用户名，就能自动填充照片、简介和一小部分近期文章呢？你能否对他们最感兴趣的一些文章进行快速文本分析，并根据其调整网站的导航结构，以及建议的产品和服务呢？我希望你能够看到如何使用自动化数据抓取替代表单，从而更好地服务你的受众，增长网站规模。

1.4.4 发现并融入你的生态系统

抓取数据自然会让你发现并考虑与你付出相关的社区的关系。当你抓取一个数据源时,很自然地就会产生一些问题:我是否相信他们的数据?我是否相信获取数据的公司?我是否需要和他们沟通以获得更正式的合作?我和他们是竞争关系还是合作关系?从其他源获得这些数据会花费我多少钱?无论如何,这些商业风险都是存在的,不过抓取过程可以帮助我们尽早意识到这些风险,并制定出缓解策略。

你还会发现自己想知道能够为这些网站和社区带来的回馈是什么。如果你能够给他们带来免费的流量,他们应该会很高兴。另一方面,如果你的应用不能给你的数据源带来一些价值,那么你们的关系可能会很短暂,除非你与他们沟通,并找到合作的方式。通过从不同源获取数据,你需要准备好开发对现有生态系统更友好的产品,充分尊重已有的市场参与者,只有在值得努力时才可以去破坏当前的市场秩序。现有的参与者也可能会帮助你成长得更快,比如你有一个应用,使用两到三个不同生态系统的数据库,每个生态系统有 10 万个用户,你的服务可能最终将这 30 万个用户以一种创造性的方式连接起来,从而使每个生态系统都获益。例如,你成立了一个初创公司,将摇滚乐与 T 恤印花社区关联起来,你的公司最终将成为两种生态系统的融合,你和相应的社区都将从中获益并得以成长。

1.5 在充满爬虫的世界里做一个好公民

当开发爬虫时,还有一些事情需要清楚。不负责任的网络爬虫会令人不悦,甚至在某些情况下是违法的。有两个非常重要的事情是避免类似**拒绝服务 (DoS)** 攻击的行为以及侵犯版权。

对于第一种情况,一个典型的访问者可能每几秒访问一个新的页面。而一个典型的网络爬虫则可能每秒下载数十个页面。这样就比典型用户产生的流量多出了 10 倍以上。这可能会使网站所有者非常不高兴。请使用流量限速将你产生的流量减少到可以接受的普通用户的水平。此外,还应该监控响应时间,如果发现响应时间增加了,就需要降低

爬虫的强度。好消息是 Scrapy 对于这些功能都提供了开箱即用的实现（参见第 7 章）。

对于版权问题，显然你需要看一下你抓取的每个网站的版权声明，并确保你理解其允许做什么，不允许做什么。大多数网站都允许你处理其站点的信息，只要不以自己的名义重新发布即可。在你的请求中，有一个很好的 User-Agent 字段，它可以让网站管理员知道你是谁，你用他们的数据做什么。Scrapy 在制造请求时，默认使用 BOT_NAME 参数作为 User-Agent。如果 User-Agent 是一个 URL 或者能够指明你的应用名称，那么网站管理员可以通过访问你的站点，更多地了解你是如何使用他们的数据的。另一个非常重要的方面是，请允许任何网站管理员阻止你访问其网站的指定区域。对于基于 Web 标准的 robots.txt 文件（参见 <http://www.google.com/robots.txt> 的文件示例），Scrapy 提供了用于尊重网站管理员设置的功能（RobotsTxtMiddleware）。最后，最好向网站管理员提供一些方法，让他们能说明不希望在你的爬虫中出现的東西。至少网站管理员必须能够很容易地找到和你交流及表达顾虑的方式。

1.6 Scrapy 不是什么

最后，很容易误解 Scrapy 可以为你做什么，主要是因为数据抓取这个术语与其相关术语有些模糊，很多术语是交替使用的。我将尝试使这些方面更加清楚，以防止混淆，为你节省一些时间。

Scrapy 不是 Apache Nutch，也就是说，它不是一个通用的网络爬虫。如果 Scrapy 访问一个一无所知的网站，它将无法做出任何有意义的事情。Scrapy 是用于提取结构化信息的，需要人工介入，设置合适的 XPath 或 CSS 表达式。而 Apache Nutch 则是获取通用页面并从中提取信息，比如关键字。它可能更适用于一些应用，但对另一些应用则又更不适合。

Scrapy 不是 Apache Solr、Elasticsearch 或 Lucene，换句话说，就是它与搜索引擎无关。Scrapy 并不打算为你提供包含“Einstein”或其他单词的文档的参考。你可以使用 Scrapy 抽取数据，然后将其插入到 Solr 或 Elasticsearch 当中，我们会在第 9 章的开始部分讲解这一做法，不过这仅仅是使用 Scrapy 的一个方法，而不是嵌入在 Scrapy 内的功能。

最后，Scrapy 不是类似 MySQL、MongoDB 或 Redis 的数据库。它既不存储数据，也不索引数据。它只用于抽取数据。即便如此，你可能会将 Scrapy 抽取得到的数据插入到数据库当中，而且它对很多数据库也都有所支持，能够让你的生活更加轻松。然而 Scrapy 终究不是一个数据库，其输出也可以很容易地更改为只是磁盘中的文件，甚至什么都不输出——虽然我不确定这有什么用。

1.7 本章小结

本章介绍了 Scrapy，给出了它能够帮你做什么的概述，并描述了我们认为的使用本书的正确方式。本章还提供了几种自动化数据抓取的方式，通过帮你快速开发能够与现有生态系统更好融合的高质量应用而获益。下一章将介绍 HTML 和 XPath，这是两个非常重要的 Web 语言，我们在每个 Scrapy 项目中都将用到它们。

第 2 章

理解 HTML 和 XPath

为了从网页中抽取信息，你必须对其结构有更多了解。我们将快速浏览 HTML、HTML 的树状表示，以及在网页上选取信息的一种方式 XPath。

2.1 HTML、DOM 树表示以及 XPath

让我们花费一些时间来了解从用户在浏览器中输入 URL（或者更常见的是，在其单击链接或书签时）到屏幕上显示出页面的过程。从本书的视角来看，该过程包含 4 个步骤，如图 2.1 所示。

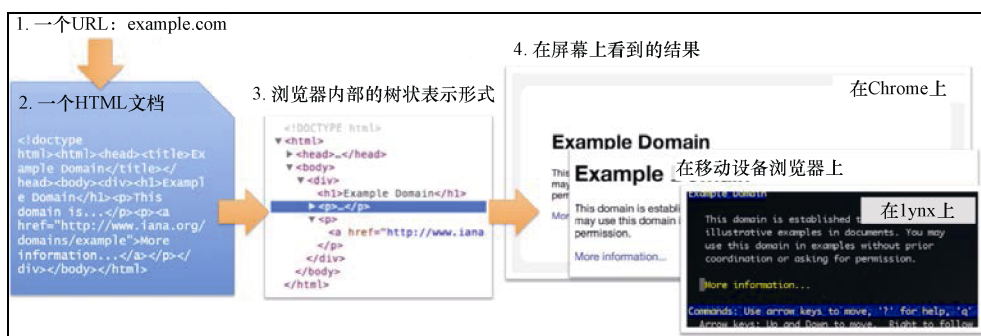


图 2.1

- 在浏览器中输入 URL。URL 的第一部分（域名，比如 gumtree.com）用于在网络上找到合适的服务器，而 URL 以及 cookie 等其他数据则构成了一个请求，

用于发送到那台服务器当中。

- 服务端回应，向浏览器发送一个 HTML 页面。需要注意的是，服务端也可能返回其他格式，比如 XML 或 JSON，不过目前我们只关注 HTML。
- 将 HTML 转换为浏览器内部的树状表示形式：**文档对象模型（Document Object Model, DOM）**。
- 基于一些布局规则渲染内部表示，达到你在屏幕上看到的视觉效果。

下面来看看这些步骤，以及它们所需的文档表示。这将有助于定位你想要抓取并编写程序获取的文本。

2.1.1 URL

对于我们而言，URL 分为两个主要部分。第一个部分通过**域名系统（Domain Name System, DNS）**帮助我们在网络上定位合适的服务器。比如，当在浏览器中发送 `https://mail.google.com/mail/u/0/#inbox` 时，将会创建一个对 `mail.google.com` 的 DNS 请求，用于确定合适的服务器 IP 地址，如 `173.194.71.83`。从本质上来看，`https://mail.google.com/mail/u/0/#inbox` 被翻译为 `https://173.194.71.83/mail/u/0/#inbox`。

URL 的剩余部分对于服务端理解请求是什么非常重要。它可能是一张图片、一个文档，或是需要触发某个动作的东西，比如向服务器发送邮件。

2.1.2 HTML 文档

服务端读取 URL，理解我们的请求是什么，然后回应一个 HTML 文档。该文档实质上就是一个文本文件，我们可以使用 TextMate、Notepad、vi 或 Emacs 打开它。和大多数文本文档不同，HTML 文档具有由万维网联盟指定的格式。该规范当然已经超出了本书的范畴，不过还是让我们看一个简单的 HTML 页面。当访问 `http://example.com` 时，可以在浏览器中选择 **View Page Source**（查看页面源代码）以看到与其相关的 HTML 文件。在不同的浏览器中，具体的过程是不同的；在许多系统中，可以通过右键单击找

到该选项，并且大部分浏览器在你按下 *Ctrl + U* 快捷键（或 Mac 系统中的 *Cmd + U*）时可以显示源代码。

 在一些页面中，该功能可能无法使用。此时，需要通过单击 Chrome 菜单，然后选择 **Tools | View Source** 才可以。

下面是 `http://example.com` 目前的 HTML 源代码。

```
<!doctype html>
<html>
  <head>
    <title>Example Domain</title>
    <meta charset="utf-8" />
    <meta http-equiv="Content-type"
      content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width,
      initial-scale=1" />
    <style type="text/css"> body { background-color: ...
      } }</style>
  <body>
    <div>
      <h1>Example Domain</h1>
      <p>This domain is established to be used for
        illustrative examples examples in documents.
        You may use this domain in examples without
        prior coordination or asking for permission.</p>
      <p><a href="http://www.iana.org/domains/example">
        More information...</a></p>
    </div>
  </body>
</html>
```

我将这个 HTML 文档进行了格式化，使其更具可读性，而你看到的情况可能是所有文本在同一行中。在 HTML 中，空格和换行在大多数情况下是无关紧要的。

尖括号中间的文本（比如 `<html>` 或 `<head>`）被称为标签。`<html>` 是起始标签，而 `</html>` 是结束标签。这两种标签的唯一区别是 `/` 字符。这说明，标签是成对出现的。虽然一些网页对于结束标签的使用比较粗心（比如，为独立的段落使用单一的 `<p>` 标签），

但是浏览器有很好的容忍度，并且会尝试推测结束的`</p>`标签应该在哪里。

`<p>`和`</p>`标签中的所有东西被称为 HTML 元素。请注意，元素中可能还包括其他元素，比如示例中的`<div>`元素，或是包含`<a>`元素的第二个`<p>`元素。

有些标签会更加复杂，比如`<a href="http://www.iana.org/domains/example">`。含有 URL 的 `href` 部分被称为属性。

最后，许多元素还包含文本，比如`<h1>`元素中的 "Example Domain"。

对于我们来说，好消息是这些标签并不都是重要的。唯一可见的东西是 `body` 元素中的元素，即`<body>`和`</body>`标签之间的元素。`<head>`部分对于指明诸如字符编码的元信息来说非常重要，不过 Scrapy 能够处理大部分此类问题，所以很多情况下不需要关注 HTML 页面的这个部分。

2.1.3 树表示法

每个浏览器都有其自身复杂的内部数据结构，凭借它来渲染网页。DOM 表示法具有跨平台、语言无关性等特点，并且被大多数浏览器所支持。

想要在 Chrome 中查看网页的树表示法，可以右键单击你感兴趣的元素，然后选择 **Inspect Element**。如果该功能被禁用，你仍然可以通过单击 Chrome 菜单并选择 **Tools | Developer Tools** 来访问它，如图 2.2 所示。



图 2.2

此时，你可以看到一些看起来和 HTML 表示非常相似但又不完全相同的东西。它就是 HTML 代码的树表示法。如果不管原始 HTML 文档是如何使用空格和换行符的话，它看起来几乎就是一样的。你可以单击每个元素，检查或调整属性等，同时可以在屏幕

上观察这些变动有何影响。比如，当你双击某个文本，修改它，并按下回车键时，屏幕上的文本将会更新为这个新值。在右侧的 **Properties** 标签下，可以看到这个树表示法的属性，并且在底部可以看到一个类似面包屑的结构，它显示出了当前选择的元素在 HTML 元素层次结构中的确切位置，如图 2.3 所示。

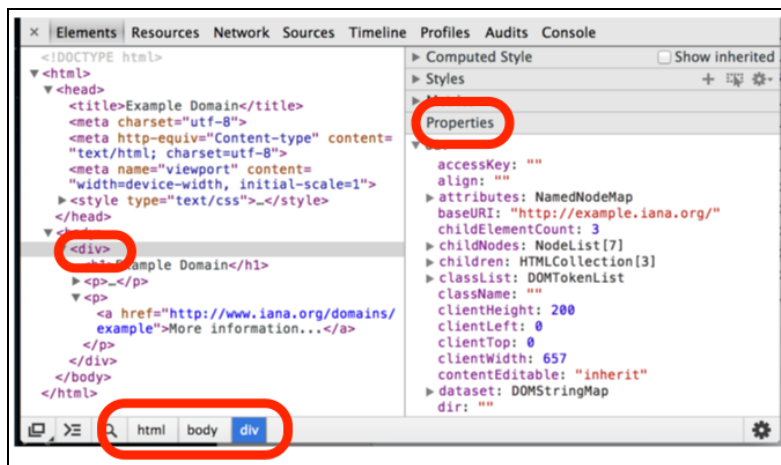


图 2.3

需要注意的一个重要事情是，HTML 只是文本，而树表示法是浏览器内存里的对象，你可以通过编程的方式查看并操纵它，比如在 Chrome 中使用 **Developer Tools**。

2.1.4 你会在屏幕上看到什么

HTML 文本表示和树表示并不包含任何像我们通常在屏幕上看到的那种漂亮视图。这实际上是 HTML 成功的原因之一。它应该是一个由人类阅读的文档，并且可以指定页面中的内容，而不是用于在屏幕中渲染的方式。这意味着选择 HTML 文档并使其更加好看是浏览器的责任，不管它是诸如 Chrome 的全功能浏览器、移动设备浏览器，甚至是诸如 Lynx 的纯文本浏览器。

也就是说，网络的发展促使 Web 开发者和用户对网页渲染的控制产生了巨大需求。CSS 的创建就是为了对 HTML 元素如何渲染给予提示。不过，对于抓取而言，我们并不需要任何和 CSS 相关的东西。

那么，树表示法是如何映射到我们在屏幕上所看到的東西呢？答案就是框模型。正如 DOM 树元素可以包含其他元素或文本一样，默认情况下，当在屏幕上渲染时，每个元素的框表示同样也都包含其嵌入元素的框表示。从这种意义上说，我们在屏幕上所看到的是原始 HTML 文档的二维表示——树结构也以一种隐藏的方式作为该表示的一部分。比如，在图 2.4 中，我们可以看到 3 个 DOM 元素（一个<div>和两个嵌入元素<h1>和<p>）是如何在浏览器和 DOM 中呈现的。

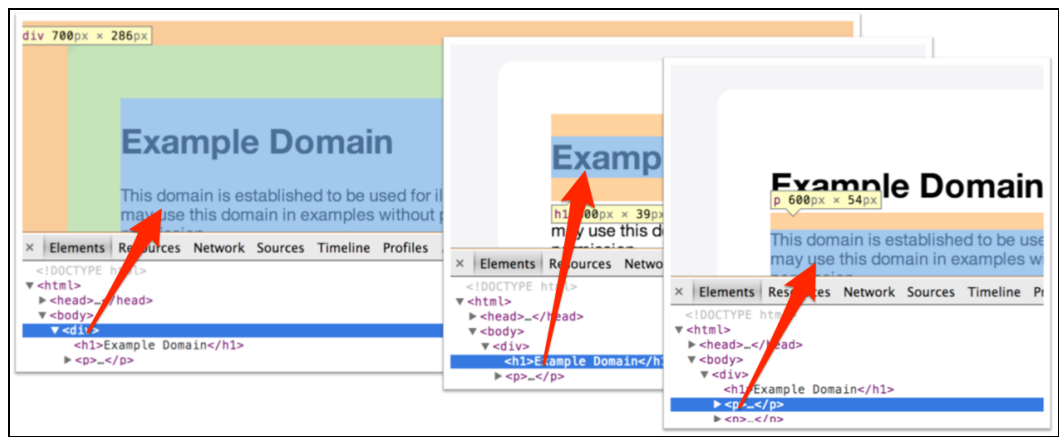


图 2.4

2.2 使用 XPath 选择 HTML 元素

如果你具有传统软件工程背景，并且不了解 XPath 相关知识的话，可能会担心为了访问 HTML 文档中的信息，你将需要做很多字符串匹配、在文档中搜索标签、处理特殊情况等工作，或是需要设法解析整个树表示法以获取你想抽取的东西。有一个好消息是这些工作都不是必需的。你可以通过一种称为 XPath 的语言选择并抽取元素、属性和文本，这种语言正是专门为此而设计的。

为了在 Google Chrome 浏览器中使用 XPath，需要单击 **Developer Tools** 的 **Console** 标签，并使用 \$x 工具函数。比如，你可以尝试在 `http://example.com/` 上使用 `$x('//h1')`。它将会把浏览器移动到<h1>元素上，如图 2.5 所示。

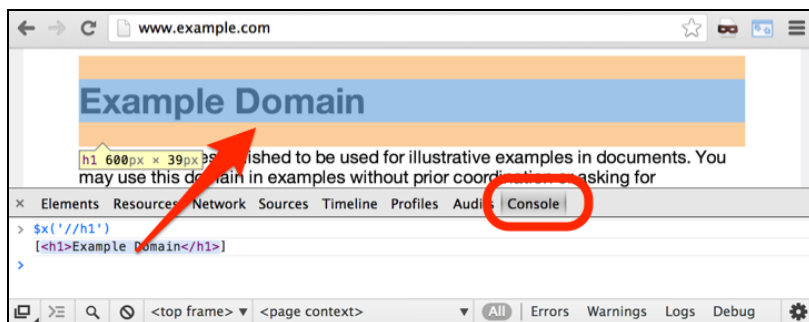


图 2.5

你在 Chrome 的 **Console** 标签中将会看到返回的是一个包含选定元素的 JavaScript 数组。如果将鼠标指针移动到这些属性上，被选取的元素将会在屏幕上高亮显示，这样就会十分方便。

2.2.1 有用的 XPath 表达式

文档的层次结构始于<html>元素，可以使用元素名和斜线来选择文档中的元素。比如，下面是几种表达式从 `http://example.com` 页面返回的结果。

```
$x('/html')
[ <html>...</html> ]
$x('/html/body')
[ <body>...</body> ]
$x('/html/body/div')
[ <div>...</div> ]
$x('/html/body/div/h1')
[ <h1>Example Domain</h1> ]
$x('/html/body/div/p')
[ <p>...</p>, <p>...</p> ]
$x('/html/body/div/p[1]')
[ <p>...</p> ]
$x('/html/body/div/p[2]')
[ <p>...</p> ]
```

需要注意的是，因为在这个特定页面中，<div>下包含两个<p>元素，因此 `html/body/div/p` 会返回两个元素。可以使用 `p[1]` 和 `p[2]` 分别访问第一个和第二个元素。

另外还需要注意的是，从抓取的角度来说，文档标题可能是 head 部分中我们唯一感兴趣的元素，该元素可以通过下面的表达式进行访问。

```
$x('//html/head/title')
[ <title>Example Domain</title> ]
```

对于大型文档，可能需要编写一个非常大的 XPath 表达式以访问指定元素。为了避免这一问题，可以使用 // 语法，它可以让你取得某一特定类型的元素，而无需考虑其所在的层次结构。比如，//p 将会选择所有的 p 元素，而//a 则会选择所有的链接。

```
$x('//p')
[ <p>...</p>, <p>...</p> ]
$x('//a')
[ <a href="http://www.iana.org/domains/example">More
information...</a> ]
```

同样，//a 语法也可以在层次结构中的任何地方使用。比如，要想找到 div 元素下的所有链接，可以使用 //div//a。需要注意的是，只使用单斜线的 //div/a 将会得到一个空数组，这是因为在 example.com 中，'div' 元素的直接下级中并没有任何 'a' 元素：

```
$x('//div//a')
[ <a href="http://www.iana.org/domains/example">More
information...</a> ]
$x('//div/a')
[ ]
```

还可以选择属性。http://example.com/ 中的唯一属性是链接中的 href，可以使用符号 @ 来访问该属性，如下面的代码所示。

```
$x('//a/@href')
[ href="http://www.iana.org/domains/example" ]
```



实际上，在 Chrome 的最新版本中，@href 不再返回 URL，而是返回一个空字符串。不过不用担心，你的 XPath 表达式仍然是正确的。

还可以通过使用 text() 函数，只选取文本。

```
$x('//a/text()')
[ "More information..." ]
```

可以使用*符号来选择指定层级的所有元素。比如：

```
$x('//div/*')
[ <h1>Example Domain</h1>, <p>...</p>, <p>...</p> ]
```

你将会发现选择包含指定属性（比如@class）或是属性为特定值的元素非常有用。可以使用更高级的谓词来选取元素，而不再是前面例子中使用过的 p[1] 和 p[2]。比如，//a[@href] 可以用来选择包含 href 属性的链接，而//a[@href="http://www.iana.org/domains/example"] 则是选择 href 属性为特定值的链接。

更加有用的是，它还拥有找到 href 属性中以一个特定子字符串起始或包含的能力。下面是几个例子。

```
$x('//a[@href]')
[ <a href="http://www.iana.org/domains/example">More information...</a> ]
$x('//a[@href="http://www.iana.org/domains/example"]')
[ <a href="http://www.iana.org/domains/example">More information...</a> ]
$x('//a[contains(@href, "iana")]')
[ <a href="http://www.iana.org/domains/example">More information...</a> ]
$x('//a[starts-with(@href, "http://www.")]')
[ <a href="http://www.iana.org/domains/example">More information...</a> ]
$x('//a[not(contains(@href, "abc"))]')
[ <a href="http://www.iana.org/domains/example">More information...</a> ]
```

XPath 有很多像 not()、contains() 和 starts-with() 这样的函数，你可以在在线文档（http://www.w3schools.com/xsl/xsl_functions.asp）中找到它们，不过即使不使用这些函数，你也可以走得很远。

现在，我还要再多说一点，大家可以在 Scrapy 命令行中使用同样的 XPath 表达式。要打开一个页面并访问 Scrapy 命令行，只需要输入如下命令：

```
scrapy shell http://example.com
```

在命令行中，可以访问很多在编写爬虫代码时经常需要用到的变量（参见下一章）。这其中最重要的就是响应，对于 HTML 文档来说就是 HtmlResponse 类，该类可以让你通过 xpath() 方法模拟 Chrome 中的 \$x。下面是一些示例。

```
response.xpath('/html').extract()
[u'<html><head><title>...</body></html>']
response.xpath('/html/body/div/h1').extract()
```

```
[u'<h1>Example Domain</h1>']
response.xpath('/html/body/div/p').extract()
[u'<p>This domain ... permission.</p>', u'<p><a href="http://www.iana.org/domains/example">More information...</a></p>']
response.xpath('///html/head/title').extract()
[u'<title>Example Domain</title>']
response.xpath('///a').extract()
[u'<a href="http://www.iana.org/domains/example">More information...</a>']
response.xpath('///a/@href').extract()
[u'http://www.iana.org/domains/example']
response.xpath('///a/text()').extract()
[u'More information...']
response.xpath('///a[starts-with(@href, "http://www.")]').extract()
[u'<a href="http://www.iana.org/domains/example">More information...</a>']
```

这就意味着，你可以使用 Chrome 开发 XPath 表达式，然后在 Scrapy 爬虫中使用它们，正如我们在下一节中将要看到的那样。

2.2.2 使用 Chrome 获取 XPath 表达式

Chrome 通过向我们提供一些基本的 XPath 表达式，从而对开发者更加友好。从前文提到的检查元素开始：右键单击想要选取的元素，然后选择 **Inspect Element**。该操作将会打开 **Developer Tools**，并且在树表示法中高亮显示这个 HTML 元素。现在右键单击这里，在菜单中选择 **Copy XPath**，此时 XPath 表达式将会被复制到剪贴板中。上述过程如图 2.6 所示。



图 2.6

你可以和之前一样，在命令行中测试该表达式。

```
$x('/html/body/div/p[2]/a')  
[ <a href="http://www.iana.org/domains/example">More  
information...</a>]
```

2.2.3 常见任务示例

有一些 XPath 表达式，你将会经常遇到。让我们看一些目前在维基百科页面上的例子。维基百科拥有一套非常稳定的格式，所以我认为它们不会很快发生改变，不过改变终究还是会发生的。我们把如下这些表达式作为说明性示例。

- 获取 id 为 "firstHeading" 的 h1 标签下 span 中的 text。

```
//h1[@id="firstHeading"]/span/text()
```

- 获取 id 为 "toc" 的 div 标签内的无序列表 (ul) 中所有链接的 URL。

```
//div[@id="toc"]/ul//a/@href
```

- 获取 class 属性包含 "ltr" 以及 class 属性包含 "skin-vector" 的任意元素内所有标题元素 (h1) 中的文本。这两个字符串可能在同一个 class 中，也可能在不同的 class 中。

```
//*[contains(@class,"ltr") and contains(@class,"skin-vector")]//h1//text()
```

实际上，你将会经常在 XPath 表达式中使用到类。在这些情况下，需要记住由于一些被称为 CSS 的样式元素，你会经常看到 HTML 元素在其 class 属性中拥有多个类。比如，在一个导航系统中，你会看到一些 div 标签的 class 属性是 "link"，而另一些是 "link active"。后者是当前激活的链接，因此会表现为可见或使用一种特殊的颜色（通过 CSS）高亮表示。当抓取时，你通常会对包含有特定类的元素感兴趣，具体来说，就是前面例子中的 "link" 和 "link active"。对于这种情况，XPath 的 contains() 函数可以让你选择包含有指定类的所有元素。

- 选择 class 属性值为 "infobox" 的表格中第一张图片的 URL。

```
//table[@class="infobox"]//img[1]/@src
```

- 选择 class 属性以 "reflist" 开头的 div 标签中所有链接的 URL。

```
//div[starts-with(@class,"reflist")]//a/@href
```

- 选择子元素包含文本"References"的元素之后的div元素中所有链接的URL。

```
//*[text()="References"]/../following-sibling::div//a
```

请注意该表达式非常脆弱并且很容易无法使用，因为它对文档结构做了过多假设。

- 获取页面中每张图片的URL。

```
//img/@src
```

2.2.4 预见变化

抓取时经常会指向我们无法控制的服务器页面。这就意味着如果它们的HTML以某种方式发生变化后，就会使XPath表达式失效，我们将不得不回到爬虫当中进行修正。通常情况下，这不会花费很长时间，因为这些变化一般都很小。但是，这仍然是需要避免发生的情况。一些简单的规则可以帮助我们减少表达式失效的可能性。

- 避免使用数组索引（数值）

Chrome 经常会给你的表达式中包含大量常数，例如：

```
//*[@id="myid"]/div/div/div[1]/div[2]/div/div[1]/div[1]/a/img
```

这种方式非常脆弱，因为如果像广告块这样的东西在层次结构中的某个地方添加了一个额外的div的话，这些数字最终将会指向不同的元素。本案例的解决方法是尽可能接近目标的img标签，找到一个可以使用的包含id或者class属性的元素，如：

```
//div[@class="thumbnail"]/a/img
```

- 类并没有那么好用

使用class属性可以更加容易地精确定位元素，不过这些属性一般是用于通过CSS影响页面外观的，因此可能会由于网站布局的微小变更而产生变化。例如下面的class：

```
//div[@class="thumbnail"]/a/img
```

一段时间后，可能会变成：

```
//div[@class="preview green"]/a/img
```

- 有意义的面向数据的类要比具体的或者面向布局的类更好

在前面的例子中,无论是"thumbnail"还是"green"都是我们所依赖类名的坏示例。虽然"thumbnail"比"green"确实更好一些,但是它们都不如"departure-time"。前面两个类名是用于描述布局的,而"departure-time"更加有意义,与 div 标签中的内容相关。因此,在布局发生变化时,后者更可能保持有效。这可能也意味着该站的开发者非常清楚使用有意义并且一致的方式标注他们数据的好处。

- ID 通常是最可靠的

通常情况下, id 属性是针对一个目标的最佳选择,因为该属性既有意义又与数据相关。部分原因是 JavaScript 以及外部链接锚一般选择 id 属性以引用文档中的特定部分。例如,下面的 XPath 表达式非常健壮。

```
//*[@id="more_info"]//text()
```

例外情况是以编程方式生成的包含唯一标记的 ID。这种情况对于抓取毫无意义。比如:

```
//*[@id="order-F4982322"]
```

尽管使用了 id,但上面的表达式仍然是一个非常差的 XPath 表达式。需要记住的是,尽管 ID 应该是唯一的,但是你仍然会发现很多 HTML 文档并没有满足这一要求。

2.3 本章小结

由于标记的质量不断提高,现在可以更加容易地创建健壮的 XPath 表达式,来抽取 HTML 文档中的数据。在本章中,你学习了 HTML 文档和 XPath 表达式的基础知识。你可以看到如何使用 Google 的 Chrome 浏览器自动获取一些 XPath 表达式,并将其作为我们后续优化的起点。你同样还学到了如何通过审查 HTML 文档,直接创建这些表达式,以及辨别 XPath 表达式是否健壮。现在,我们准备好运用已经学到的所有知识,在第 3 章中使用 Scrapy 编写我们的前几个爬虫。

第 3 章

爬虫基础

这是非常重要的一章，你可能会多次阅读本章，并且经常会在寻找解决方案时回到本章中。我们首先会介绍如何安装 **Scrapy**，然后伴随若干示例及不同的实现，转向开发 **Scrapy** 爬虫的方法论。在开始之前，我们先来看一些重要的概念。

由于我们会快速进入有趣的代码部分，因此使用本书中代码片段的能力非常重要。当你看到如下内容时：

```
$ echo hello world
hello world
```

表示你在终端输入了 `echo hello word`（忽略美元符号），接下来的一行或几行就是你在终端上面看到的输出。



我们将会混用“终端”、“控制台”和“命令行”这几个术语，它们在本书的背景下没有太大区别。请用 Google 搜索并找出如何启动你所使用的平台（Windows、OS X 或其他）中的控制台。你也可以在附录 A 中找到详细的指引。

当你看到如下内容时：

```
>>> print 'hi'
hi
```

表示你在 Python 或 Scrapy 的 shell 提示符中输入了 `print 'hi'`（忽略 `>>>`）。同样地，接下来的一行或几行就是你在终端上面看到的该命令的输出。

在本书中，你还需要编辑文件。你所使用的工具很大程度上依赖于你的环境。如果你使用 Vagrant (强烈推荐)，可以使用电脑或笔记本中诸如 Notepad、Notepad++、Sublime Text、TextMate、Eclipse 或 PyCharm 等编辑器。如果你有更多的 Linux 或 UNIX 使用经验，也可能更喜欢直接使用 Vim 或 Emacs 在控制台中编辑文件。这两种编辑器都很强大，不过需要一定的学习曲线。如果你是一个初学者，并且不得不在控制台中编辑某些东西，那么也可以尝试对初学者更加友好的 nano 编辑器。

3.1 安装 Scrapy

Scrapy 的安装相对来说比较简单，不过它会完全依赖于你从哪里起步。为了能够支持尽可能多的用户，本书中运行和安装 Scrapy 以及所有示例的“官方”方式是通过 Vagrant，该软件能够让你在不考虑宿主操作系统的情况下，运行一个标准的 Linux 系统，在该系统中我们已经安装好所有需要用到的工具。我们将会在接下来的几小节中给出 Vagrant 的使用说明以及一些常用操作系统中的指引。

3.1.1 MacOS

为了更加方便地阅读本书，请按照后面给出的 Vagrant 使用说明操作。如果你想直接在 MacOS 系统中安装 Scrapy，其实也很简单。只需要输入下面的命令即可。

```
$ easy_install scrapy
```

然后，一切都会为你准备好。在过程中，可能会要求你填写密码或安装 Xcode，如图 3.1 所示。这些都没有问题，你可以放心地接受这些请求。

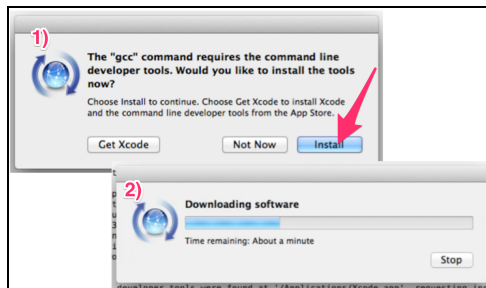


图 3.1

3.1.2 Windows

直接在 Windows 系统中安装 Scrapy 会复杂一些，坦白来说，会有一点痛苦。而且，安装本书中所需的所有软件也需要很大程度的勇气和决心。我们已经为你做好了准备。Vagrant 和 Virtualbox 可以在 Windows 64 位平台中良好运行。直接前往本章后续的相关小节，你可以很快将其安装好并运行起来。如果你必须要在 Windows 系统中直接安装 Scrapy，请查阅本书网站 (<http://scrapybook.com>) 中的资源。

3.1.3 Linux

和前面提及的两个操作系统一样，如果你想按照本书操作，那么 Vagrant 就是最为推荐的方式。

由于在很多场景下，你需要在 Linux 服务器中安装 Scrapy，因此更详尽的指引可能会很有用。



确切的依赖条件经常会发生变更。本书编写时，我们安装的 Scrapy 版本是 1.0.3，下面的内容是针对不同主流系统的操作指南。

1. Ubuntu 或 Debian Linux

为了在 Ubuntu（使用 Ubuntu 14.04 Trust Tahr 64 位版本测试）或其他使用 apt 的发布版本中安装 Scrapy，需要执行如下 3 个命令。

```
$ sudo apt-get update
```

```
$ sudo apt-get install python-pip python-lxml python-crypto python-  
cssselect python-openssl python-w3lib python-twisted python-dev libxml2-  
dev libxslt1-dev zlib1g-dev libffi-dev libssl-dev
```

```
$ sudo pip install scrapy
```

上述过程需要一些编译工作，而且可能会被不时打断，不过它将会为你安装 PyPI

源上最新版本的 Scrapy。如果你想避免某些编译工作，并且能够忍受使用稍微过时一些的话，可以通过 Google 搜索 “install Scrapy Ubuntu packages”，并跟随 Scrapy 官方文档的指引进行操作。

2. Red Hat 或 CentOS Linux

在 Red Hat 或其他使用 yum 的发布版本中安装 Scrapy 相对来说比较容易。你只需按照如下 3 行操作即可。

```
sudo yum update
sudo yum -y install libxslt-devel pyOpenSSL python-lxml python-devel gcc
sudo easy_install scrapy
```

3.1.4 最新源码安装

只要你按照上述指引操作的话，就已经安装好了 Scrapy 目前所需的所有依赖。由于 Scrapy 是纯 Python 应用，因此如果你想修改其源代码或测试最新功能，可以很容易地从 <https://github.com/scrapy/scrapy> 网站中克隆其最新版本。在你的系统中安装 Scrapy，只需输入如下命令。

```
$ git clone https://github.com/scrapy/scrapy.git
$ cd scrapy
$ python setup.py install
```

我猜如果你属于这类 Scrapy 用户，也就不需要我再提及 virtualenv 了。

3.1.5 升级 Scrapy

Scrapy 经常会升级。你会发现自己需要在很短时间内完成升级，此时可以使用 pip、easy_install 或 aptitude 完成这项工作。

```
$ sudo pip install --upgrade Scrapy
```

或

```
$ sudo easy_install --upgrade scrapy
```

如果想降级或选择特定版本，可以通过指定版本号来完成，比如：

```
$ sudo pip install Scrapy==1.0.0
```

或

```
$ sudo easy_install scrapy==1.0.0
```

3.1.6 Vagrant：本书中运行示例的官方方式

本书中会有很多复杂但又有趣的例子，其中一些例子会用到很多服务。无论是处于初学还是进阶阶段，都可以运行本书中的这些示例，这是因为被称为 **Vagrant** 的程序可以让我们仅仅使用简单的命令就能准备好这个复杂的系统。本书中使用的系统如图 3.2 所示。

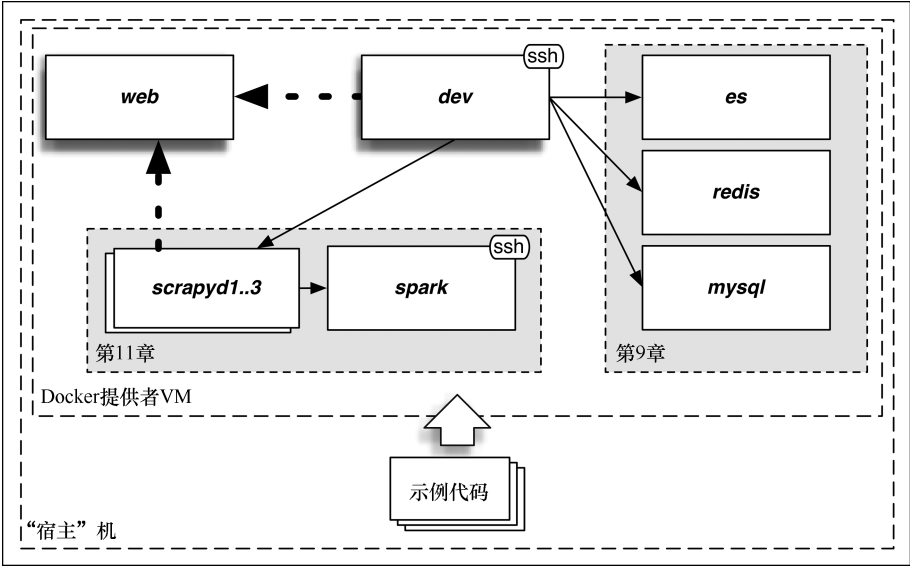


图 3.2 本书使用的系统

在 **Vagrant** 的术语中，你的电脑或笔记本被称为“宿主”机。**Vagrant** 使用宿主机运行 **Docker** 提供者 **VM**（虚拟机）。这些技术可以让我们拥有一个隔离的系统，在其中拥有其私有网络，可以忽略宿主机的软硬件，运行本书中的示例。

大部分章节只使用了两个服务：“dev”机器和“web”机器。我们登录到 **dev** 机器中运行爬虫，抓取 **web** 机器中的页面。后面的一些章节会用到更多的服务，包括数据库和大

数据处理引擎。

请按照附录 A 的说明，在操作系统中安装 Vagrant。到附录 A 的结尾时，你应当已经在操作系统中安装好 git 和 Vagrant 了。打开控制台/终端/命令提示符，现在可以按照如下操作获取本书的代码了。

```
$ git clone https://github.com/scalingexcellence/scrappybook.git
$ cd scrappybook
```

然后可以通过输入如下命令打开 Vagrant 系统。

```
$ vagrant up --no-parallel
```

在首次运行时将会花费一些时间，这取决于你的网络连接状况。在这之后，'vagrant up' 操作将会瞬间完成。当系统运行起来之后，就可以使用如下命令登录 dev 虚拟机。

```
$ vagrant ssh
```

现在，你已经处于开发控制台当中，在这里可以按照本书的其他说明操作。代码已经从你的宿主机复制到 dev 机器当中，可以在 book 目录下找到这些代码。

```
$ cd book
$ ls
ch03 ch04 ch05 ch07 ch08 ch09 ch10 ch11 ...
```

打开几个控制台并执行 vagrant ssh，可以获得多个可供操作的 dev 终端。可以使用 vagrant halt 关闭系统，使用 vagrant status 查看系统状态。请注意，vagrant halt 不会关掉 VM。如果出现问题，则需要打开 VirtualBox 然后手动关闭它，或者使用 vagrant global-status 找到其 id（名为"docker-provider"），然后使用 vagrant halt <ID> 停掉它。即使你处于离线状态，大部分示例仍然能够运行，这也是使用 Vagrant 的一个很好的副作用。

现在，我们已经正确地创建好了系统，下面就该准备学习 Scrappy 了。

3.2 UR²IM——基本抓取流程

每个网站都是不同的，如果发现某些不常见的情况，则需要一些额外的学习，或是

在 Scrapy 的邮件列表中咨询一些问题。不过，为了知道在哪里和如何搜索，重要的是对其流程有一个整体的了解，并且清楚相关的术语。和 Scrapy 打交道时，你所遵循的最通用的流程是 UR²IM 流程，如图 3.3 所示。

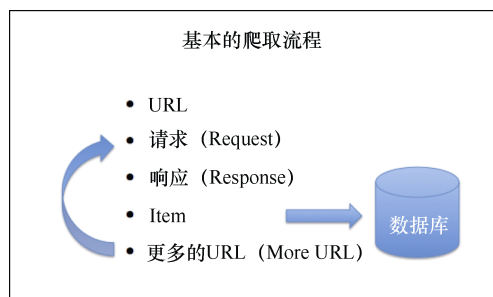


图 3.3 UR²IM 流程

3.2.1 URL

一切始于 URL。你需要从准备抓取的网站中选择几个示例 URL。我将使用 Gumtree 分类广告网站 (<https://www.gumtree.com>) 作为示例进行演示。

比如，通过访问 Gumtree 上的伦敦房产主页（链接为 <http://www.gumtree.com/flats-houses/london>），你能够找到一些房产的示例 URL。可以通过右键单击分类列表，选择 Copy Link Address（复制链接地址）或你浏览器中同样的功能，来复制这些链接。比如，其中一个可能类似于 <https://www.gumtree.com/p/studios-bedsits-rent/split-level>。虽然可以在真实网站中使用这些 URL 来操作，但不幸的是，经过一段时间后，真实的 Gumtree 网站可能会发生变化，造成 XPath 表达式无法正常工作。此外，除非设置一个用户代理头，否则 Gumtree 不会回应你的请求。稍后我们会对此进行更进一步的讲解，不过就现在而言，如果想加载它们的某个页面，可以在 scrapy shell 中使用如下命令。

```
scrapy shell -s USER_AGENT="Mozilla/5.0" <your url here e.g. http://www.gumtree.com/p/studios-bedsits-rent/...>
```

如果想要在使用 scrapy shell 时调试问题，可以使用 `--pdb` 参数启用交互式调试，以避免发生异常。例如：

```
scrapy shell --pdb https://gumtree.com
```



scrapy shell 是一个非常有用的工具，能够帮助我们使用 Scrapy 开发。

很显然，我们并不鼓励你在学习本书内容时访问 Gumtree 的网站，我们也不希望本书的示例在不久之后就无法使用。此外，我们还希望即使无法连接互联网，你仍然能够开发和使用我们的示例。这就是为什么你的 Vagrant 开发环境中包含一个提供了类似于 Gumtree 网站页面的 Web 服务器的原因。虽然它们可能不如真实网站那么漂亮，但是从爬虫角度来说，它们其实是一样的。即便如此，我们在本章中的所有截图还是来自真实的 Gumtree 网站。在你 Vagrant 的 dev 机器中，可以通过 `http://web:9312/` 访问该 Web 服务器，而在你的浏览器中，可以通过 `http://localhost:9312/` 来访问。

在 scrapy shell 中打开服务器中的一个网页，并且在 dev 机器上输入如下内容进行操作。

```
$ scrapy shell http://web:9312/properties/property_000000.html
...
[s] Available Scrapy objects:
[s]   crawler   <scrapy.crawler.Crawler object at 0x2d4fb10>
[s]   item      {}
[s]   request   <GET http:// web:9312/.../property_000000.html>
[s]   response  <200 http://web:9312/.../property_000000.html>
[s]   settings  <scrapy.settings.Settings object at 0x2d4fa90>
[s]   spider    <DefaultSpider 'default' at 0x3ea0bd0>
[s] Useful shortcuts:

[s]   shelp()           Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local...
[s]   view(response)    View response in a browser
>>>
```

我们得到了一些输出，现在可以在 Python 提示符下，用它来调试刚才加载的页面（一般情况下，可以使用 `Ctrl + D` 退出）。

3.2.2 请求和响应

大家可能注意到在前面的日志中，`scrapy shell` 本身已经为我们做了一些工作。我们给出了一个 URL，然后它执行了一个默认的 GET 请求，并得到了一个状态码为 200 的响应。这就意味着，页面信息已经加载完毕，可以使用了。如果想要打印 `response.body` 的前 50 个字符，可以按如下命令操作。

```
>>> response.body[:50]
'<!DOCTYPE html>\n<html>\n<head>\n<meta charset="UTF-8"'
```



`[:50]` 是什么？这是 Python 从文本变量（本例为 `response.body`）中抽取最前面 50 个字符（如果存在）的方式。如果你之前并不了解 Python，请保持冷静，继续向前。很快，你就会熟悉并享受所有这些语法技巧了。

这是 Gumtree 上指定页面的 HTML 内容。请求和响应部分不会给我们带来太多麻烦。不过，在很多情况下，你需要做一些工作才能保证其正确。第 5 章中讲到这些内容。就目前来说，我们尽量保持简单，直接进入下一部分——Item。

3.2.3 Item

下一步是尝试从响应中将数据抽取到 Item 的字段中。因为该页面的格式是 HTML，因此可以使用 XPath 表达式进行操作。首先，让我们看一下这个页面，如图 3.4 所示。

在图 3.4 中有大量的信息，但其中大部分都是布局：logo、搜索框、按钮等。虽然这些信息都很有用，但是爬虫并不会对其产生兴趣。我们可能感兴趣的字段，比如说包括房源的标题、位置或代理商的电话号码，它们都具有对应的 HTML 元素，我们需要定位到这些元素，然后使用前一节中所描述的流程抽取数据。那么，先从标题开始吧（如图 3.5 所示）。

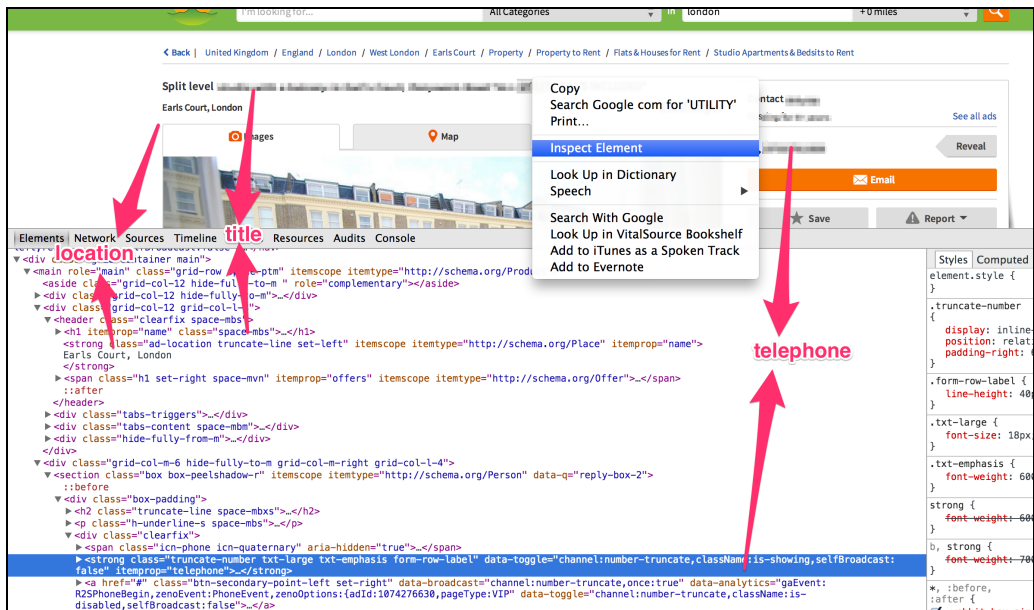


图 3.4 页面、感兴趣的字段及其 HTML 代码

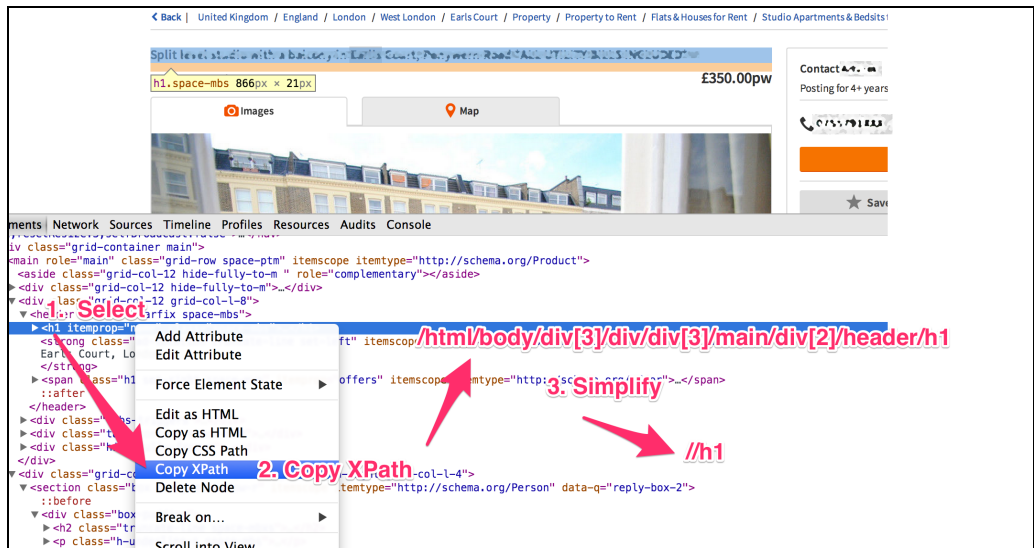


图 3.5 抽取标题

右键单击页面上的标题，并选择 **Inspect Element**。这样就可以看到相应的 HTML 源代码了。现在，尝试通过右键单击并选择 **Copy XPath**，抽取标题的 XPath 表达式。你

会发现 Chrome 浏览器给我们的 XPath 表达式很精确，但又十分复杂，因此该表达式是非常脆弱的。我们将对其进行一些简化，只使用最后的一部分，通过使用表达式 `//h1`，选择在页面中可以看到的任何 H1 元素。尽管这种方式有些误导，因为我们并不是真的需要页面中的每一个 H1，不过实际上这里只有标题使用了 H1；而作为优秀的 SEO 实践，每个页面应当只有一个 H1 元素，并且大部分网站确实是这样的。



SEO 是 Search Engine Optimization（搜索引擎优化）的缩写，即通过优化网站代码、内容和出入站链接的流程，实现提供给搜索引擎的最佳方式。

我们来检查下该 XPath 表达式能否在 scrapy shell 中良好运行。

```
>>> response.xpath('//h1/text()').extract()
[u'set unique family well']
```

非常好，完美工作。你应该已经注意到我在 `//h1` 表达式的结尾处添加了 `/text()`。如果想要只抽取 H1 元素所包含的文本内容，而不是 H1 元素自身的话，就需要使用到它。我们通常都会使用 `/text()` 来获得文本字段。如果忽略它，就会得到整个元素的文本，包括并不需要的标记。

```
>>> response.xpath('//h1').extract()
[u'<h1 itemprop="name" class="space-mbs">set unique family well</h1>']
```

此时，我们就得到了抽取本页中第一个感兴趣的属性（标题）的代码，不过如果你观察得更仔细的话，就会发现还有一种更好更简单的方法也可以做到。

Gumtree 通过微数据标记注解它们的 HTML。比如，我们可以看到，在其头部有一个 `itemprop="name"` 的属性，如图 3.6 所示。非常好，这样我们就可以使用一个更简单的 XPath 表达式，而不再包含任何可视化元素了，此时得到的表达式为 `//*[@itemprop="name"][1]/text()`。你可能会奇怪为什么我们选择了包含 `itemprop="name"` 的第一个元素。



图 3.6 Gumtree 拥有微数据标记



稍等！你是说第一个？如果你是一个经验丰富的程序员，可能已经将 `array[1]` 作为数组的第二个元素了。令人惊讶的是，XPath 是从 1 开始的，因此 `array[1]` 是数组的第一个元素。

我们这么做，不只是因为 `itemprop="name"` 在许多不同的上下文中作为微数据来使用，还因为 Gumtree 在其页面的“你可能还喜欢……”部分为其他属性使用了嵌套的信息，以这种方式阻止我们对其轻易识别。尽管如此，这并不是一个大问题。我们只需要选择第一个，而且我们也将使用同样的方式处理其他字段。

让我们来看一下价格。价格被包含在如下的 HTML 结构当中。

```
<strong class="ad-price txt-xlarge txt-emphasis" itemprop="price">
£334.39pw</strong>
```

我们又一次看到了 `itemprop="name"` 这种形式，太棒了。此时，XPath 表达式会是 `//*[@itemprop="price"][1]/text()`。我们来试一下吧。

```
>>> response.xpath('//*[@itemprop="price"][1]/text()').extract()
[u'\xa3334.39pw']
```

我们注意到，这里包含一些 Unicode 字符（英镑符号£），然后是 334.39pw 的价格。这表明数据并不总是像我们希望的那样干净，所以可能还需要对其进行一些清洗的工作。比如，在本例中，我们可能需要使用一个正则表达式，以便只选择数字和点号。可以使用 `re()` 方法做到这一要求，并使用一个简单的正则表达式替代 `extract()`。

```
>>> response.xpath('//*[@itemprop="price"][1]/text()').re('[.0-9]+')
[u'334.39']
```



这里使用了一个 `response` 对象，并调用了它的 `xpath()` 方法来抽取感兴趣的值。不过，`xpath()` 返回的值是什么呢？如果在一个简单的 XPath 表达式中，不使用 `.extract()` 方法，将会得到如下的显示输出：

```
>>> response.xpath('.')
[<Selector xpath='.' data=u'<html>\n<head>\n<meta
charse'>]
```

`xpath()` 返回了网页内容预加载的 Selector 对象。我们目前只使用了 `xpath()` 方法, 不过它还有另一个有用的方法: `css()`。`xpath()` 和 `css()` 都会返回选择器, 只有当调用 `extract()` 或 `re()` 方法的时候, 才会得到真实的文本数组。这种方式非常好用, 因为这样就可以将 `xpath()` 和 `css()` 操作串联起来了。比如, 可以使用 `css()` 快速抽取正确的 HTML 元素。

```
>>> response.css('.ad-price')
[<Selector xpath=u"descendant-or-self::*[@class and contains(concat(' ', normalize-space(@class), ' '), 'ad-price ')]" data=u'<strong class="ad-price txt-xlarge txt-e">]
```

请注意, 在后台中 `css()` 实际上编译了一个 `xpath()` 表达式, 不过我们输入的内容要比 XPath 自身更加简单。接下来, 串联一个 `xpath()` 方法, 只抽取其中的文本。

```
>>> response.css('.ad-price').xpath('text()')
[<Selector xpath='text()' data=u'\xa3334.39pw'>]
```

最后, 还可以通过 `re()` 方法, 串联上正则表达式, 以抽取感兴趣的值。

```
>>> response.css('.ad-price').xpath('text()').re('[.0-9]+')
[u'334.39']
```

实际上, 这个表达式与原始表达式相比, 并无好坏之差。请把它当作一个引起思考的说明性示例。在本书中, 我们将尽可能保持事物简单, 同时也会尽可能多地使用虽然有些老旧但仍然好用的 XPath。关键点是记住 `xpath()` 和 `css()` 返回的 Selector 对象是可以被串联起来的。为了获取真实值, 可以使用 `extract()`, 也可以使用 `re()`。在 Scrapy 的每个新版本当中, 都会围绕这些类添加新的令人兴奋且高价值的功能。相关的 Scrapy 文档部分为 <http://doc.scrapy.org/en/latest/topics/selectors.html>。该文档非常优秀, 相信你可以从中找到抽取数据的最有效的方式。



描述文本的抽取也是相似的。有一个 `itemprop="description"` 的属性用于标示描述。其 XPath 表达式为 `//*[ @itemprop="description" ][1]/text()`。相似地，住址部分使用 `itemtype="http://schema.org/Place"` 注解；因此，XPath 表达式为 `//*[ @itemtype="http:// schema.org/Place" ][1]/text()`。

同理，图片使用了 `itemprop="image"`。因此使用 `//img [ @itemprop="image" ][1]/@src`。这里需要注意的是，我们没有使用 `/text()`，这是因为我们并不需要任何文本，而是只需要包含图片 URL 的 `src` 属性。

假设这些是我们想要抽取的全部信息，我们可以将其总结到表 3.1 中。

表 3.1

基本字段	XPath 表达式
title	<code>//*[@itemprop="name"][1]/text()</code> 示例值: [u'set unique family well']
price	<code>//*[@itemprop="price"][1]/text()</code> 示例值 (使用 <code>re()</code>): [u'334.39']
description	<code>//*[@itemprop="description"][1]/text()</code> 示例值: [u'website court warehouse\r\npool...']
address	<code>//*[@itemtype="http://schema.org/Place"][1]/text()</code> 示例值: [u'Angel, London']
image_urls	<code>//*[@itemprop="image"][1]/@src</code> 示例值: [u'../images/i01.jpg']

现在，表 3.1 就变得非常重要了，因为如果我们有许多包含相似信息的网站，则很可能需要创建很多类似的爬虫，此时只需改变前面的这些表达式。此外，如果想要抓取大量网站，也可以使用这样一张表格来拆分工作量。

到目前为止，我们主要在使用 HTML 和 XPath。接下来，我们将开始编写一些真正的 Python 代码。

3.3 一个 Scrapy 项目

到目前为止，我们只是在通过 scrapy shell “小打小闹”。现在，既然已经拥有了用于开始第一个 Scrapy 项目的所有必要组成部分，那么让我们按下 *Ctrl + D* 退出 scrapy shell 吧。需要注意的是，你现在输入的所有内容都将丢失。显然，我们并不希望在每次爬取某些东西的时候都要输入代码，因此一定要谨记 scrapy shell 只是一个可以帮助我们调试页面、XPath 表达式和 Scrapy 对象的工具。不要花费大量时间在这里编写复杂代码，因为一旦你退出，这些代码就都会丢失。为了编写真实的 Scrapy 代码，我们将使用项目。下面创建一个 Scrapy 项目，并将其命名为"properties"，因为我们正在抓取的数据是房产。

```
$ scrapy startproject properties
$ cd properties
$ tree
.
├── properties
│   ├── __init__.py
│   ├── items.py
│   ├── pipelines.py
│   ├── settings.py
│   └── spiders
│       └── __init__.py
└── scrapy.cfg
```

2 directories, 6 files



提醒一下，你可以从 GitHub 中获得本书的全部源代码。要下载该代码，可以使用如下命令：

```
git clone https://github.com/scalingexcellence/scrapybook
```

本章的代码在 ch03 目录中，其中该示例的代码在 ch03/properties 目录中。

我们可以看到这个 Scrapy 项目的目录结构。命令 scrapy startproject properties

创建了一个以项目名命名的目录，其中包含 3 个我们感兴趣的文件，分别是 `items.py`、`pipelines.py` 和 `settings.py`。这里还有一个名为 `spiders` 的子目录，目前为止该目录是空的。在本章中，我们将主要在 `items.py` 文件和 `spiders` 目录中工作。在后续的章节里，还将对设置、管道和 `scrapy.cfg` 文件有更多探索。

3.3.1 声明 item

我们使用一个文件编辑器打开 `items.py` 文件。现在该文件中已经包含了一些模板代码，不过还需要针对用例对其进行修改。我们将重定义 `PropertiesItem` 类，添加表 3.2 中总结出来的字段。

我们还会添加几个字段，我们的应用在后续会用到这些字段（这样之后就不需要再修改这个文件了）。本书后续的内容会深入解释它们。需要重点注意的一个事情是，我们声明一个字段并不意味着我们将在每个爬虫中都填充该字段，或是全部使用它。你可以随意添加任何你感觉合适的字段，因为你可以在之后更正它们。

表 3.2

可计算字段	Python 表达式
<code>images</code>	图像管道将会基于 <code>image_urls</code> 自动填充该字段。可以在后续的章节中了解更多相关内容
<code>location</code>	我们的地理编码管道将会在后面填充该字段。可以在后续的章节中了解更多相关的内容

我们还会添加一些管理字段（见表 3.3）。这些字段不是特定于某个应用程序的，而是我个人感兴趣的字段，可能会在未来帮助我调试爬虫。你可以在项目中选择其中的一些字段，当然也可以不选择。如果你仔细观察这些字段，就会明白它们可以让我清楚地（`server`、`url`）、何时（`date`）、如何（`spider`）执行的抓取。它们还可以自动完成一些任务，比如使 `item` 失效、规划新的抓取迭代或是删除来自有问题的爬虫的 `item`。如果你还不能理解所有的表达式，尤其是 `server` 的表达式，也不用担心。当我们进入到后面的章节时，这些都会变得越来越清楚。

表 3.3

管理字段	Python 表达式
url	response.url 示例值: 'http://web.../property_000000. html'
project	self.settings.get('BOT_NAME') 示例值: 'properties'
spider	self.name 示例值: 'basic'
server	socket.gethostname() 示例值: 'scrapyserver1'
date	datetime.datetime.now() 示例值: datetime.datetime(2015, 6, 25...)

给出字段列表之后, 再去修改并自定义 scrapy startproject 为我们创建的 PropertiesItem 类, 就会变得很容易。在文本编辑器中, 修改 properties/items.py 文件, 使其包含如下内容:

```
from scrapy.item import Item, Field

class PropertiesItem(Item):
    # Primary fields
    title = Field()
    price = Field()
    description = Field()
    address = Field()
    image_urls = Field()

    # Calculated fields
    images = Field()
    location = Field()

    # Housekeeping fields
    url = Field()
    project = Field()
    spider = Field()
    server = Field()
    date = Field()
```

由于这实际上是我们在文件中编写的第一个 Python 代码，因此需要重点指出的是，Python 使用缩进作为其语法的一部分。在每个字段的起始部分，会有精确的 4 个空格或 1 个制表符，这一点非常重要。如果你在其中一行使用了 4 个空格，而在另一行使用了 3 个空格，就会出现语法错误。如果你在其中一行使用了 4 个空格，而在另一行使用了制表符，同样也会产生语法错误。这些空格在 PropertiesItem 类下，将字段声明组织到了一起。其他语言一般使用大括号（{}）或特殊的关键词（如 begin-end）来组织代码，而 Python 使用空格。

3.3.2 编写爬虫

我们已经在半路上了。现在，我们需要编写爬虫。通常，我们会为每个网站或网站的一部分（如果网站非常大的话）创建一个爬虫。爬虫代码实现了完整的 UR²IM 流程，我们很快就可以看到。



什么时候使用爬虫，什么时候使用项目呢？项目是由 Item 和若干爬虫组成的。如果有很多网站，并且需要从中抽取相同类型的 Item，比如：房产，那么所有这些网站都可以使用同一个项目，并且为每个源/网站使用一个爬虫。反之，如果要处理图书及房产这两种不同的源时，则应该使用不同的项目。

当然，可以在文本编辑器中从头开始创建一个爬虫，不过为了减少一些输入，更好的方法是使用 scrapy genspider 命令，如下所示。

```
$ scrapy genspider basic web
Created spider 'basic' using template 'basic' in module:
properties.spiders.basic
```

现在，如果再次运行 tree 命令，就会注意到与之前相比唯一的不同是在 properties/spiders 目录中增加了一个新文件 basic.py。前面的命令所做的工作就是创建了一个名为“basic”的“默认”爬虫，并且该爬虫被限制为只能爬取 web 域名下的 URL。如果需要的话，可以很容易地移除这个限制，不过目前来说没有问题。爬虫使用“basic”模板创建。你可以通过输入 scrapy genspider -l 来查看其他可用的模板，然后在执行 scrapy genspider 时，通过 -t 参数，使用任意其他模板创建爬虫。

在本章稍后的部分，我们将会看到一个示例。



Scrapy 有许多子目录。我们一般假设你位于包含 `scrapy.cfg` 文件的目录中。这是项目的“顶级”目录。现在，每当我们引用 Python “包”和“模块”时，它们就是以映射目录结构的方式设置的。比如，输出提到了 `properties.spiders.basic`，就是指 `properties/spiders` 目录中的 `basic.py` 文件。我们早前定义的 `PropertiesItem` 类是在 `properties.items` 模块中，该模块对应的就是 `properties` 目录中的 `items.py` 文件。

如果查看 `properties/spiders/basic.py` 文件，可以看到如下代码。

```
import scrapy

class BasicSpider(scrapy.Spider):
    name = "basic"
    allowed_domains = ["web"]
    start_urls = (
        'http://www.web/',
    )

    def parse(self, response):
        pass
```

`import` 语句能够让我们使用 Scrapy 框架中已有的类。下面是扩展自 `scrapy.Spider` 的 `BasicSpider` 类的定义。通过“扩展”的方式，尽管我们实际上没有写任何代码，但是该类已经“继承”了 Scrapy 框架中的 `Spider` 类的相当一部分功能。这样，就可以只额外编写少量的代码行，而获得一个完整运行的爬虫了。然后，我们可以看到一些爬虫的参数，比如它的名字以及我们允许其爬取的域名。最后是空函数 `parse()` 的定义，该函数包含了两个参数，分别是 `self` 和 `response` 对象。通过使用 `self` 引用，我们就可以使用爬虫中感兴趣的功能了。而另一个对象 `response`，我们应该很熟悉，它就是我们在 `scrapy shell` 中使用过的 `response` 对象。



这是你的代码——你的爬虫。不要害怕修改它，你不会真的把事情搞砸的。即使在最坏的情况下，你还可以使用 `rmproperties/spiders/basic.py` 删除文件，然后再重新生成。尽情发挥吧！

好了，让我们开始改造吧。首先，要使用在 `scrapy shell` 中使用过的那个 URL，对应地设置到 `start_urls` 参数中。然后，将使用爬虫预定义的方法 `log()`，输出在基本字段表中总结的所有内容。修改后，`properties/spiders/basic.py` 的代码如下所示。

```
import scrapy

class BasicSpider(scrapy.Spider):
    name = "basic"
    allowed_domains = ["web"]
    start_urls = (
        'http://web:9312/properties/property_000000.html',
    )

    def parse(self, response):
        self.log("title: %s" % response.xpath(
            '//*[ @itemprop="name" ][1]/text() ').extract())
        self.log("price: %s" % response.xpath(
            '//*[ @itemprop="price" ][1]/text() ').re('[.0-9]+'))
        self.log("description: %s" % response.xpath(
            '//*[ @itemprop="description" ][1]/text() ').extract())
        self.log("address: %s" % response.xpath(
            '//*[ @itemtype="http://schema.org/"
            'Place" ][1]/text() ').extract())
        self.log("image_urls: %s" % response.xpath(
            '//*[ @itemprop="image" ][1]/@src ').extract())
```



我将会不时地修改格式，以便在屏幕和纸张中都能很好地显示。这并不意味着它有什么特殊的含义。

等了这么久，终于到了运行爬虫的时候了。我们可以使用命令 `scrapy crawl` 以及爬虫的名称来运行爬虫。

```
$ scrapy crawl basic
INFO: Scrapy 1.0.3 started (bot: properties)
...
INFO: Spider opened
DEBUG: Crawled (200) <GET http://...000.html>
DEBUG: title: [u'set unique family well']
DEBUG: price: [u'334.39']
DEBUG: description: [u'website...']
DEBUG: address: [u'Angel, London']
DEBUG: image_urls: [u'../images/i01.jpg']
INFO: Closing spider (finished)
...
```

非常好！不要被大量的日志行吓倒。我们将会在后续的章节中更详细地研究其中的一部分，不过对于现在而言，只需要注意到所有使用 XPath 表达式收集到的数据确实能够通过这个简单的爬虫代码抽取出来就可以了。

让我们再来试验一下另一个命令：scrapy parse。它允许我们使用“最合适”的爬虫来解析参数中给定的任意 URL。我不喜欢抱有侥幸心理，所以我们使用它结合 --spider 参数来设置爬虫。

```
$ scrapy parse --spider=basic http://web:9312/properties/property_000001.html
```

你会看到输出和之前是相似的，只不过现在是另一套房产。



scrapy parse 同样也是一个相当方便的调试工具。在任何情况下，如果你想“认真”抓取的话，应当使用主命令 scrapy crawl。

3.3.3 填充 item

我们将会对前面的代码进行少量修改，以填充 PropertiesItem。你将会看到，尽管修改非常轻微，但是会“解锁”大量的新功能。

首先，需要引入 PropertiesItem 类。如前所述，它在 properties 目录的 items.py 文件中，也就是 properties.items 模块中。我们回到 properties/spiders/

basic.py 文件，使用如下命令引入该模块。

```
from properties.items import PropertiesItem
```

然后需要进行实例化，并返回一个对象。这非常简单。在 parse() 方法中，可以通过添加 item = PropertiesItem() 语句创建一个新的 item，然后可以按如下方式为其字段分配表达式。

```
item['title'] =  
response.xpath('//*[ @itemprop="name" ][1]/text() ').extract()
```

最后，使用 return item 返回 item。最新版的 properties/spiders/basic.py 代码如下所示。

```
import scrapy  
from properties.items import PropertiesItem  
  
class BasicSpider(scrapy.Spider):  
    name = "basic"  
    allowed_domains = ["web"]  
    start_urls = (  
        'http://web:9312/properties/property_000000.html',  
    )  
  
    def parse(self, response):  
        item = PropertiesItem()  
        item['title'] = response.xpath(  
            '//*[ @itemprop="name" ][1]/text() ').extract()  
        item['price'] = response.xpath(  
            '//*[ @itemprop="price" ][1]/text() ').re('[.0-9]+')  
        item['description'] = response.xpath(  
            '//*[ @itemprop="description" ][1]/text() ').extract()  
        item['address'] = response.xpath(  
            '//*[ @itemtype="http://schema.org/'  
            'Place" ][1]/text() ').extract()  
        item['image_urls'] = response.xpath(  
            '//*[ @itemprop="image" ][1]/@src ').extract()  
        return item
```

现在，如果你再像之前那样运行 scrapy crawl basic，就会发现一个非常小但很重要的区别。我们不再在日志中记录抓取值（所以没有包含字段值的 DEBUG:行了），而是看到如下的输出行。

```

DEBUG: Scraped from <200
http://...000.html>
{'address': [u'Angel, London'],
 'description': [u'website ... offered'],
 'image_urls': [u'../images/i01.jpg'],
 'price': [u'334.39'],
 'title': [u'set unique family well']}

```

这是从本页面抓取得到的 `PropertiesItem`。非常好，因为 Scrapy 是围绕着 `Items` 的概念构建的，也就是说你现在可以使用后续章节中介绍的管道，对其进行过滤和丰富了，并且可以通过“Feed exports”将其以不同的格式导出存储到不同的地方。

3.3.4 保存文件

请尝试如下爬取示例。

```

$ scrapy crawl basic -o items.json
$ cat items.json
[{"price": ["334.39"], "address": ["Angel, London"], "description":
["website court ... offered"], "image_urls": ["../images/i01.jpg"],
"title": ["set unique family well"]}]]

$ scrapy crawl basic -o items.jl
$ cat items.jl
{"price": ["334.39"], "address": ["Angel, London"], "description":
["website court ... offered"], "image_urls": ["../images/i01.jpg"],
"title": ["set unique family well"]}

$ scrapy crawl basic -o items.csv
$ cat items.csv
description,title,url,price,spider,image_urls...
"...offered",set unique family well,,334.39,..../images/i01.jpg
$ scrapy crawl basic -o items.xml
$ cat items.xml
<?xml version="1.0" encoding="utf-8"?>
<items><item><price><value>334.39</value></price>...</item></items>

```

我们不需要编写任何额外的代码，就可以保存为这些不同的格式。Scrapy 在幕后识别你想要输出的文件扩展名，并以适当的格式输出到文件中。前面的格式覆盖了一些最常见的用例。CSV 和 XML 文件非常流行，因为类似微软 Excel 的电子表格程序可以直

接打开它们。JSON 文件在网上非常流行，原因是它们富有表现力而且与 JavaScript 的关系相当密切。JSON 与 JSON 行（JSON Line）格式的轻微不同是，.json 文件是在一个大数组中存储 JSON 对象的。这就意味着如果你有一个 1GB 的文件，你可能不得不在使用典型的解析器解析之前，将其全部存入内存当中。而 .jl 文件则是每行包含一个 JSON 对象，所以它们可以被更高效地读取。

将你生成的文件保存到文件系统之外的地方也很容易。比如，通过使用如下命令，Scrapy 可以自动将文件上传到 FTP 或 S3 存储桶中。

```
$ scrapy crawl basic -o "ftp://user:pass@ftp.scrapybook.com/items.json"
$ scrapy crawl basic -o "s3://aws_key:aws_secret@scrapybook/items.json"
```

需要注意的是，除非凭证和 URL 都更新为与有效的主机/S3 提供商相匹配，否则该示例无法工作。



我的 MySQL 驱动在哪里？起初，我也对 Scrapy 缺少针对 MySQL 或其他数据库的内置支持感到惊讶。而实际上，没有什么内置的，这与 Scrapy 的思考方式是完全违背的。Scrapy 的目标是快速和可扩展。它使用了很少的 CPU，以及尽可能高的入站带宽。从性能的角度来看，将数据插入到大部分关系型数据库将会是一场灾难。当需要将 item 插入到数据库时，必须将其先存储到文件当中，然后再使用批量加载机制导入它们。在第 9 章中，我们将会看到多种高效的方式，用来将独立的 item 导入到数据库中。

这里需要注意的另一件事是，如果你现在尝试使用 scrapy parse，它会向你显示已经抓取的 item，以及你的爬取生成的新请求（本例中没有）。

```
$ scrapy parse --spider=basic http://web:9312/properties/property_000001.html
INFO: Scrapy 1.0.3 started (bot: properties)
...
INFO: Spider closed (finished)

>>> STATUS DEPTH LEVEL 1 <<<
# Scraped Items -----
```

```
[{'address': [u'Plaistow, London'],
  'description': [u'features'],
  'image_urls': [u'../images/i02.jpg'],
  'price': [u'388.03'],
  'title': [u'belsize marylebone...deal']}]
# Requests -----
[]
```

在调试给出意料之外的结果的 URL 时，你会更加感激 scrapy parse。

3.3.5 清理——item 装载器与管理字段

恭喜，你在创建基础爬虫方面做得不错！下面让我们做得更专业一些吧。

首先，我们使用一个强大的工具类——ItemLoader，以替代那些杂乱的 extract() 和 xpath() 操作。通过使用该类，我们的 parse() 方法会按如下进行代码变更。

```
def parse(self, response):
    l = ItemLoader(item=PropertiesItem(), response=response)

    l.add_xpath('title', '//*[@itemprop="name"][1]/text()')
    l.add_xpath('price', '//*[@itemprop="price"]'
                '[1]/text()', re='[.0-9]+')
    l.add_xpath('description', '//*[@itemprop="description"]'
                '[1]/text()')
    l.add_xpath('address', '//*[@itemtype="http://schema.org/Place"][1]/text()')
    l.add_xpath('image_urls', '//*[@itemprop="image"][1]/@src')

    return l.load_item()
```

好多了，是不是？不过，这种写法并不只是在视觉上更加舒适，它还非常明确地声明了我们意图去做的事情，而不会将其与实现细节混淆起来。这就使得代码具有更好的可维护性以及自描述性。

ItemLoader 提供了许多有趣的结合数据及对数据进行格式化和清洗的方式。请注意，此类功能的开发非常活跃，因此请查阅 Scrapy 优秀的官方文档来发现使用它们的更高效的方式，文档地址为 <http://doc.scrapy.org/en/latest/topics/loaders.html>。Itemloaders 通过不同的处理类传递 XPath/CSS 表达式的值。处理

器是一个快速而又简单的函数。处理器的一个例子是 `Join()`。假设你已经使用类似 `//p` 的 XPath 表达式选取了很多个段落, 该处理器可以将这些段落结合成一个条目。另一个非常有意思的处理器是 `MapCompose()`。通过该处理器, 你可以使用任意 Python 函数或 Python 函数链, 以实现复杂的功能。比如, `MapCompose(float)` 可以将字符串数据转换为数值, 而 `MapCompose(Unicode.strip, Unicode.title)` 可以删除多余的空白符, 并将字符串格式化为每个单词均为首字母大写的样式。让我们看一些处理器的例子, 如表 3.4 所示。

表 3.4

处 理 器	功 能
<code>Join()</code>	把多个结果连接在一起
<code>MapCompose(unicode.strip)</code>	去除首尾的空白符
<code>MapCompose(unicode.strip, unicode.title)</code>	与 <code>MapCompose(unicode.strip)</code> 相同, 不过还会使结果按照标题格式
<code>MapCompose(float)</code>	将字符串转为数值
<code>MapCompose(lambda i: i.replace(',', ' '), float)</code>	将字符串转为数值, 并忽略可能存在的','字符
<code>MapCompose(lambda i: urlparse.urljoin(response.url, i))</code>	以 <code>response.url</code> 为基础, 将 URL 相对路径转换为 URL 绝对路径

你可以使用任何 Python 表达式作为处理器。可以看到, 我们可以很容易地将它们一个接一个地连接起来, 比如, 我们前面给出的去除首尾空白符以及标题化的例子。`unicode.strip()` 和 `unicode.title()` 在某种意义上来说比较简单, 它们只有一个参数, 并且也只有一个返回结果。我们可以在 `MapCompose` 处理器中直接使用它们。而另一些函数, 像 `replace()` 或 `urljoin()`, 就会稍微有点复杂, 它们需要多个参数。对于这种情况, 我们可以使用 Python 的“lambda 表达式”。lambda 表达式是一种简洁的函数。比如下面这个简洁的 lambda 表达式。

```
myFunction = lambda i: i.replace(',', ' ')
```

可以代替:

```
def myFunction(i):  
    return i.replace(',', ' ')
```

通过使用 `lambda`, 我们将类似 `replace()` 和 `urljoin()` 这样的函数包装在只有一个参数及一个返回结果的函数中。为了能够更好地理解表 3.4 中的处理器, 下面看几个使用处理器的例子。使用 `scrapy shell` 打开任意 URL, 然后尝试如下操作。

```
>>> from scrapy.loader.processors import MapCompose, Join
>>> Join()(['hi', 'John'])
u'hi John'
>>> MapCompose(unicode.strip)([u' I', u' am\n'])
[u' I', u'am']
>>> MapCompose(unicode.strip, unicode.title)([u'nIce cOdE'])
[u'Nice Code']
>>> MapCompose(float)(['3.14'])
[3.14]
>>> MapCompose(lambda i: i.replace(',', ''), float)(['1,400.23'])
[1400.23]
>>> import urlparse
>>> mc = MapCompose(lambda i: urlparse.urljoin('http://my.com/test/abc',
i))
>>> mc(['example.html#check'])
['http://my.com/test/example.html#check']
>>> mc(['http://absolute/url#help'])
['http://absolute/url#help']
```

这里要解决的关键问题是, 处理器只是一些简单小巧的功能, 用来对我们的 XPath/CSS 结果进行后置处理。现在, 在爬虫中使用几个这样的处理器, 并按照我们想要的方式输出。

```
def parse(self, response):
    l.add_xpath('title', '//*[@itemprop="name"][1]/text()',
                MapCompose(unicode.strip, unicode.title))
    l.add_xpath('price', '//*[@itemprop="price"][1]/text()',
                MapCompose(lambda i: i.replace(',', ''), float),
                re='[, .0-9]+')
    l.add_xpath('description', '//*[@itemprop="description"]'
                '[1]/text()', MapCompose(unicode.strip), Join())
    l.add_xpath('address',
                '//*[@itemtype="http://schema.org/Place"][1]/text()',
                MapCompose(unicode.strip))
    l.add_xpath('image_urls', '//*[@itemprop="image"][1]/@src',
                MapCompose(
                    lambda i: urlparse.urljoin(response.url, i)))
```

完整列表将会在本章后续部分给出。当你使用我们目前开发的代码运行 `scrapy crawl basic` 时，可以得到更加整洁的输出值。

```
'price': [334.39],  
'title': [u'Set Unique Family Well']
```

最后，我们可以通过使用 `add_value()` 方法，添加 Python 计算得出的单个值（而不是 XPath/CSS 表达式）。我们可以用该方法设置“管理字段”，比如 URL、爬虫名称、时间戳等。我们还可以直接使用管理字段表中总结出来的表达式，如下所示。

```
l.add_value('url', response.url)  
l.add_value('project', self.settings.get('BOT_NAME'))  
l.add_value('spider', self.name)  
l.add_value('server', socket.gethostname())  
l.add_value('date', datetime.datetime.now())
```

为了能够使用其中的某些函数，请记得引入 `datetime` 和 `socket` 模块。

好了！我们现在已经得到了非常不错的 `Item`。此刻，你的第一感觉可能是所做的这些都很复杂，你可能想要知道这些工作是不是值得付出努力。答案当然是值得的——这是因为，这就是你为了从页面抽取数据并将其存储到 `Item` 中几乎所有需要知道的东西。如果你从零开始编写，或者使用其他语言，该代码通常都会非常难看，而且很快就会变得不可维护。而使用 `Scrapy` 时，只需要仅仅 25 行代码。该代码十分简洁，用于表明意图，而不是实现细节。你清楚地知道每一行代码都在做什么，并且它可以很容易地修改、复用及维护。

你可能产生的另一个感觉是所有的处理器以及 `ItemLoader` 并不值得去努力。如果你是一个经验丰富的 Python 开发者，可能会觉得有些不舒服，因为你必须去学习新的类，来实现通常使用字符串操作、`lambda` 表达式以及列表推导式就可以完成的操作。不过，这只是 `ItemLoader` 及其功能的简要概述。如果你更加深入地了解它，就不会再回头了。`ItemLoader` 和处理器是基于编写并支持了成千上万个爬虫的人们的抓取需求而开发的工具包。如果你准备开发多个爬虫的话，就非常值得去学习使用它们。

3.3.6 创建 contract

`contract` 有点像为爬虫设计的单元测试。它可以让你快速知道哪里有运行异常。例如，

假设你在几个星期之前编写了一个抓取程序，其中包含几个爬虫，今天想要检查一下这些爬虫是否仍然能够正常工作，就可以使用这种方式。`contract` 包含在紧挨着函数名的注释（即文档字符串）中，并且以`@`开头。下面来看几个 `contract` 的例子。

```
def parse(self, response):
    """ This function parses a property page.

    @url http://web:9312/properties/property_000000.html
    @returns items 1
    @scrapes title price description address image_urls
    @scrapes url project spider server date
    """
```

上述代码的含义是，检查该 URL，并找到我列出的字段中有值的一个 Item。现在，当你运行 `scrapy check` 时，就会去检查 `contract` 是否能够满足。

```
$ scrapy check basic
```

```
-----
Ran 3 contracts in 1.640s
OK
```

如果将 `url` 字段留空（通过注释掉该行来设置），你会得到一个失败描述。

```
FAIL: [basic] parse (@scrapes post-hook)
```

```
-----
ContractFail: 'url' field is missing
```

`contract` 失败的原因可能是爬虫代码无法运行，或者是你要检查的 URL 的 XPath 表达式已经过时了。虽然结果并不详尽，但它是抵御坏代码的第一道灵巧的防线。

综合上面的内容，下面给出我们的第一个基础爬虫的代码。

```
from scrapy.loader.processors import MapCompose, Join
from scrapy.loader import ItemLoader
from properties.items import PropertiesItem
import datetime
import urlparse
import socket
import scrapy

class BasicSpider(scrapy.Spider):
    name = "basic"
    allowed_domains = ["web"]
```

```
# Start on a property page
start_urls = (
    'http://web:9312/properties/property_000000.html',
)

def parse(self, response):
    """ This function parses a property page.
    @url http://web:9312/properties/property_000000.html
    @returns items 1
    @scrapes title price description address image_urls
    @scrapes url project spider server date
    """
    # Create the loader using the response
    l = ItemLoader(item=PropertiesItem(), response=response)

    # Load fields using XPath expressions
    l.add_xpath('title', '//*[@itemprop="name"][1]/text()',
                MapCompose(unicode.strip, unicode.title))
    l.add_xpath('price', '//*[@itemprop="price"][1]/text()',
                MapCompose(lambda i: i.replace(',', ''),
                            float),
                re='[,.0-9]+')
    l.add_xpath('description', '//*[@itemprop="description"]'
                '[1]/text()',
                MapCompose(unicode.strip), Join())
    l.add_xpath('address',
                '//*[@itemtype="http://schema.org/Place"]'
                '[1]/text()',
                MapCompose(unicode.strip))
    l.add_xpath('image_urls', '//*[@itemprop="image"]'
                '[1]/@src', MapCompose(
                    lambda i: urlparse.urljoin(response.url, i)))

    # Housekeeping fields
    l.add_value('url', response.url)
    l.add_value('project', self.settings.get('BOT_NAME'))
    l.add_value('spider', self.name)
    l.add_value('server', socket.gethostname())
    l.add_value('date', datetime.datetime.now())

    return l.load_item()
```

3.4 抽取更多的 URL

到目前为止，我们使用的只是设置在爬虫的 `start_urls` 属性中的单一 URL。而该属性实际为一个元组，我们可以硬编码写入更多的 URL，如下所示。

```
start_urls = (  
    'http://web:9312/properties/property_000000.html',  
    'http://web:9312/properties/property_000001.html',  
    'http://web:9312/properties/property_000002.html',  
)
```

这种写法可能不会让你太激动。不过，我们还可以使用文件作为 URL 的源，写法如下所示。

```
start_urls = [i.strip() for i in  
open('todo.urls.txt').readlines()]
```

这种写法其实也不那么令人激动，但它确实管用。更经常发生的情况是感兴趣的网站中包含一些索引页及房源页。比如，Gumtree 就包含了如图 3.7 所示的索引页，其地址为 `http://www.gumtree.com/flats-houses/london`。

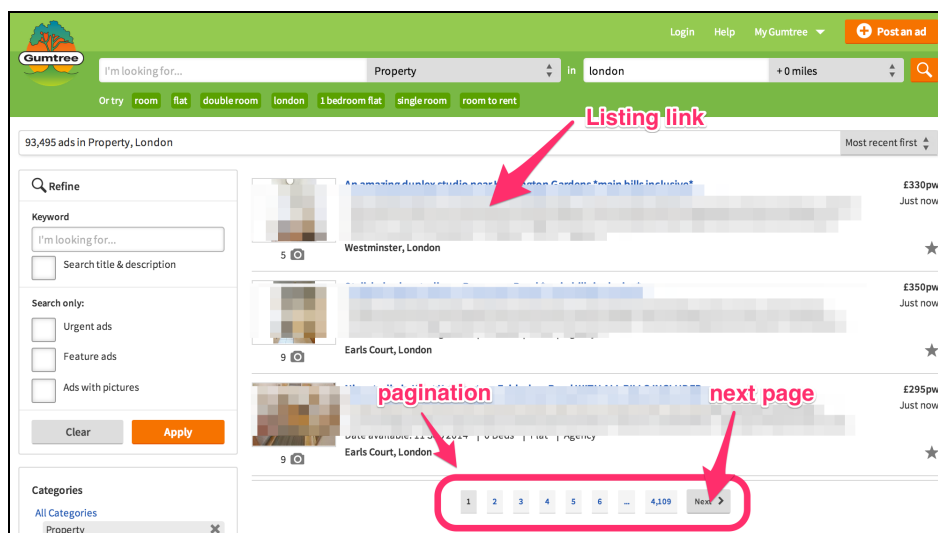


图 3.7 Gumtree 的索引页

一个典型的索引页会包含许多到房源页面的链接，以及一个能够让你从一个索引页前往另一个索引页的分页系统。

因此，一个典型的爬虫会向两个方向移动（见图 3.8）：

- 横向——从一个索引页到另一个索引页；
- 纵向——从一个索引页到房源页并抽取 Item。



图 3.8 向两个方向移动的典型爬虫

在本书中，我们将前者称为**水平爬取**，因为这种情况下是在同一层级下爬取页面（比如索引页）；而将后者称为**垂直爬取**，因为该方式是从一个更高的层级（比如索引页）到一个更低的层级（比如房源页）。

实际上，它比听起来更加容易。我们所有需要做的事情就是再增加两个 XPath 表达式。对于第一个表达式，右键单击 **Next Page** 按钮，可以注意到 URL 包含在一个链接中，而该链接又是在一个拥有类名 `next` 的 `li` 标签内，如图 3.9 所示。因此，我们只需使用一个实用的 XPath 表达式 `//*[contains (@class, "next")]/@href`，就可以完美

运行了。

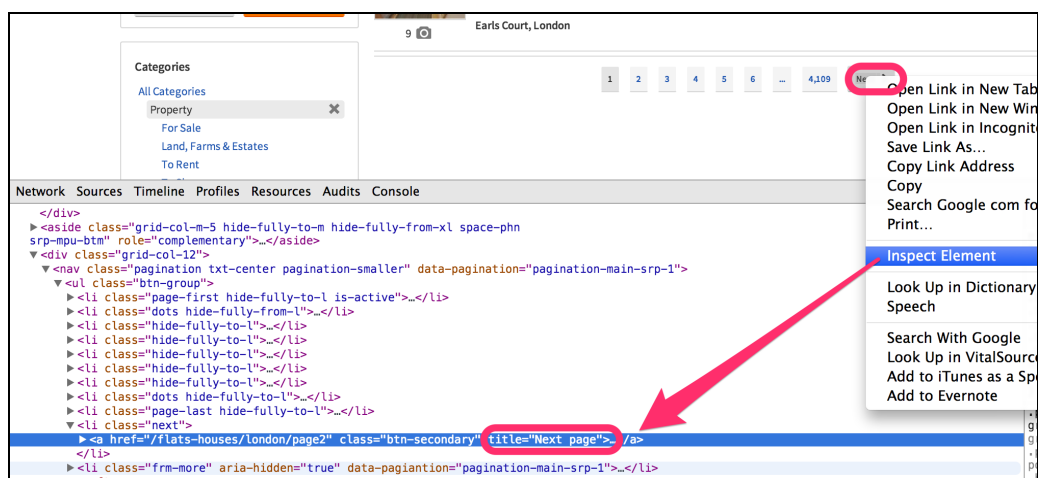


图 3.9 查找下一个索引页 URL 的 XPath 表达式

对于第二个表达式，右键单击页面中的列表标题，并选择 **Inspect Element**，如图 3.10 所示。

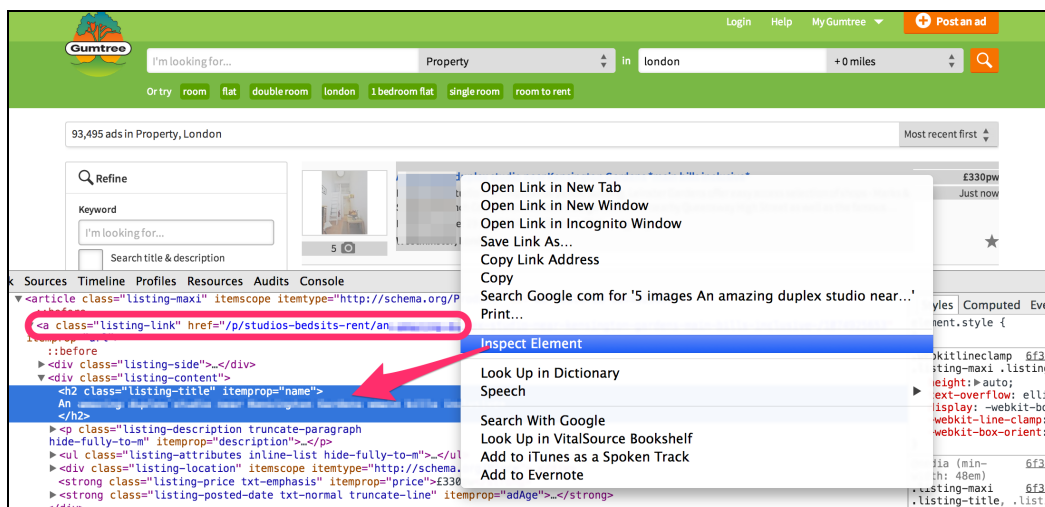


图 3.10 查找列表页 URL 的 XPath 表达式

请注意，URL 中包含我们感兴趣的 `itemprop="url"` 属性。因此，表达式

`//*[@itemprop="url"]/@href` 就可以正常运行。现在，打开一个 scrapy shell 来确认这两个表达式是否有效：

```
$ scrapy shell http://web:9312/properties/index_00000.html
>>> urls = response.xpath('//*[contains(@class,"next")]//@href').extract()
>>> urls
[u'index_00001.html']
>>> import urlparse
>>> [urlparse.urljoin(response.url, i) for i in urls]
[u'http://web:9312/scrapybook/properties/index_00001.html']
>>> urls = response.xpath('//*[itemprop="url"]/@href').extract()
>>> urls
[u'property_000000.html', ... u'property_000029.html']
>>> len(urls)
30
>>> [urlparse.urljoin(response.url, i) for i in urls]
[u'http://..._000000.html', ... /property_000029.html']
```

非常好！可以看到，通过使用之前已经学习的内容及这两个 XPath 表达式，我们已经能够按照自身需求使用水平抓取和垂直抓取的方式抽取 URL 了。

3.4.1 使用爬虫实现双向爬取

我们将之前的爬虫拷贝到一个新文件中，并命名为 `manual.py`。

```
$ ls
properties scrapy.cfg
$ cp properties/spiders/basic.py properties/spiders/manual.py
```

在 `properties/spiders/manual.py` 文件中，通过添加 `from scrapy.http import Request` 语句引入 Request 模块，将爬虫的 `name` 参数改为 `'manual'`，修改 `start_urls` 以使用第一个索引页，并将 `parse()` 方法重命名为 `parse_item()`。好了！现在开始编写一个新的 `parse()` 方法，来实现水平和垂直两种抓取方式。

```
def parse(self, response):
    # Get the next index URLs and yield Requests
    next_selector = response.xpath('//*[contains(@class, '
                                   '"next")]//@href')
    for url in next_selector.extract():
```

```

yield Request(urlparse.urljoin(response.url, url))

# Get item URLs and yield Requests
item_selector = response.xpath('//*[ @itemprop="url" ]/@href')
for url in item_selector.extract():
    yield Request(urlparse.urljoin(response.url, url),
                  callback=self.parse_item)

```



你可能已经注意到了前面例子中的 `yield` 语句。`yield` 与 `return` 在某种意义上来说有些相似，都是将返回值提供给调用者。不过，和 `return` 不同的是，`yield` 不会退出函数，而是继续执行 `for` 循环。从功能上来说，前面的例子与下面的代码大体相当：

```

next_requests = []
for url in...
    next_requests.append(Request(...))
for url in...
    next_requests.append(Request(...))
return next_requests

```

`yield` 是 Python “魔法”的一部分，它可以使日常的高效编程工作更加轻松。

我们现在已经准备好运行该爬虫了。不过如果让该爬虫以当前的方式运行的话，则会抓取网站完整的 5 万个页面。为了避免运行时间过长，可以通过命令行参数：`-s CLOSESPIDER_ITEMCOUNT=90`，告知爬虫在爬取指定数量（如 90 个）的 Item 后停止运行（更多细节参见第 7 章）。现在，我们可以运行了。

```

$ scrapy crawl manual -s CLOSESPIDER_ITEMCOUNT=90
INFO: Scrapy 1.0.3 started (bot: properties)
...
DEBUG: Crawled (200) <...index_00000.html> (referer: None)
DEBUG: Crawled (200) <...property_000029.html> (referer: ...index_00000.html)
DEBUG: Scraped from <200 ...property_000029.html>
{'address': [u'Clapham, London'],
 'date': [datetime.datetime(2015, 10, 4, 21, 25, 22, 801098)],

```

```
'description': [u'situated camden facilities corner'],
'image_urls': [u'http://web:9312/images/i10.jpg'],
'price': [223.88],
'project': ['properties'],
'server': ['scrapyserver1'],
'spider': ['manual'],
'title': [u'Portered Mile'],
'url': ['http://.../property_000029.html']]
DEBUG: Crawled (200) <...property_000028.html> (referer: ...index_00000.
html)
...
DEBUG: Crawled (200) <...index_00001.html> (referer: ...)
DEBUG: Crawled (200) <...property_000059.html> (referer: ...)
...
INFO: Dumping Scrapy stats: ...
      'downloader/request_count': 94, ...
      'item_scraped_count': 90,
```

如果仔细查看前面的输出，就会发现我们同时获得了水平抓取和垂直抓取的结果。第一个 `index_00000.html` 读取后，派生出了许多请求。当它们执行时，调试信息通过 `referer` URL 指出是谁发起的请求。比如，可以看到，`property_000029.html`、`property_000028.html`……及 `index_00001.html` 都有相同的 `referer`(`index_00000.html`)。而 `property_000059.html` 及其他请求则是以 `index_00001.html` 为 `referer` 的，并且该过程还在持续。

从该示例中还可以观察到，Scrapy 在处理请求时使用的是**后入先出(LIFO)**策略(即深度优先爬取)。用户提交的最后一个请求会被首先处理。在大多数情况下，这种默认的方式非常方便。比如，我们想要在移动到下一个索引页之前处理每一个房源页时。否则，我们将会填充一个包含待爬取房源页 URL 的巨大队列，无谓地消耗内存。另外，在许多情况中，你可能需要辅助的请求来完成单个请求，我们将会在后面的章节中遇到这种情况。你需要这些辅助的请求能够尽快完成，以腾出资源，并且让被抓取的 Item 能够稳定流动。

我们可以通过设置 `Request()` 的优先级参数修改默认顺序，大于 0 表示高于默认的优先级，小于 0 表示低于默认的优先级。通常来说，Scrapy 的调度器会首先执行高优

先级的请求，不过不要花费太多时间来考虑具体的哪个请求应该被首先执行。很可能在你的应用中，不会使用超过 1 个或 2 个请求优先级。此外还需要注意的是，URL 还会被执行去重操作，这在大部分时候也是我们想要的功能。不过如果我们需要多次执行同一个 URL 的请求，可以设置 `dont_filter_request()` 参数为 `true`。

3.4.2 使用 CrawlSpider 实现双向爬取

如果感觉上面的双向爬取有些冗长，则说明你确实发现了关键问题。Scrapy 尝试简化所有此类通用情况，以使其编码更加简单。最简单的实现同样结果的方式是使用 `CrawlSpider`，这是一个能够更容易地实现这种爬取的类。为了实现它，我们需要使用 `genspider` 命令，并设置 `-t crawl` 参数，以使用 `crawl` 爬虫模板创建一个爬虫。

```
$ scrapy genspider -t crawl easy web
Created spider 'crawl' using template 'crawl' in module:
properties.spiders.easy
```

现在，文件 `properties/spiders/easy.py` 包含如下内容。

```
...
class EasySpider(CrawlSpider):
    name = 'easy'
    allowed_domains = ['web']
    start_urls = ['http://www.web/']

    rules = (
        Rule(LinkExtractor(allow=r'Items/'),
            callback='parse_item', follow=True),
    )

    def parse_item(self, response):
        ...
```

当你阅读这段自动生成的代码时，会发现它和之前的爬虫有些相似，不过在此处的类声明中，会发现爬虫是继承自 `CrawlSpider`，而不再是 `Spider`。`CrawlSpider` 提供了一个使用 `rules` 变量实现的 `parse()` 方法，这与我们之前例子中手工实现的功能一致。



你可能会感到疑惑，为什么我首先给出了手工实现的版本，而不是直接给出捷径。这是因为你在手工实现的示例中，学会了使用回调的 `yield` 方式的请求，这是一个非常有用和基础的技术，我们将会在后文的章节中不断使用它，因此理解该内容非常值得。

现在，我们要把 `start_urls` 设置成第一个索引页，并且用我们之前的实现替换预定义的 `parse_item()` 方法。这次我们将不再需要实现任何 `parse()` 方法。我们将预定义的 `rules` 变量替换为两条规则，即水平抓取和垂直抓取。

```
rules = (  
    Rule(LinkExtractor(restrict_xpaths='//*[contains(@class,"next")]')),  
    Rule(LinkExtractor(restrict_xpaths='//*[@itemprop="url"]',  
                        callback='parse_item'))  
)
```

这两条规则使用的是和我们之前手工实现的示例中相同的 XPath 表达式，不过这里没有了 `a` 或 `href` 的限制。顾名思义，`LinkExtractor` 正是专门用于抽取链接的，因此在默认情况下，它们会去查找 `a`（及 `area`）`href` 属性。你可以通过设置 `LinkExtractor()` 的 `tags` 和 `attrs` 参数来进行自定义。需要注意的是，回调参数目前是包含回调方法名称的字符串（比如 `'parse_item'`），而不是方法引用，如 `Request(self.parse_item)`。最后，除非设置了 `callback` 参数，否则 `Rule` 将跟踪已经抽取的 URL，也就是说它将会扫描目标页面以获取额外的链接并跟踪它们。如果设置了 `callback`，`Rule` 将不会跟踪目标页面的链接。如果你希望它跟踪链接，应当在 `callback` 方法中使用 `return` 或 `yield` 返回它们，或者将 `Rule()` 的 `follow` 参数设置为 `true`。当你的房源页既包含 `Item` 又包含其他有用的导航链接时，该功能可能会非常有用。

运行该爬虫，可以得到和手工实现的爬虫相同的结果，不过现在使用的是一个更加简单的源代码。

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=90
```

3.5 本章小结

本章可能是大家开始学习 Scrapy 时最重要的一章。你刚刚学习了开发爬虫最基本的方法：UR²IM。你学会了如何自定义适合需求的 Item，使用 ItemLoader、XPath 表达式和处理器加载 Item，以及如何对 Request 使用 yield 操作。我们使用 Request 横向到达不同的索引页，纵向到达房源页并抽取 Item。最后，我们看到了如何使用 CrawlSpider 和 Rule，以很少的代码行创建非常强大的爬虫。如果你想要更深入地理解这些概念，请尽可能多地阅读本章，当然，也可以在你开发自己的爬虫时使用本章作为参考。

我们刚刚从网站中得到了一些信息。为什么它这么重要呢？我想答案会在下一章中变得明朗起来，在下一章中，通过简单的几页内容，我们将会开发一个简单的手机应用，并使用 Scrapy 填充其中的数据。我想，结果会令大家印象深刻。

第 4 章

从 Scrapy 到移动应用

我能够听到人们的尖叫声：“Appery.io 是什么，一个手机应用的专用平台，它和 Scrapy 有什么关系？”那么，眼见为实吧。你可能还会对几年前在 Excel 电子表格上给某个人（朋友、管理者或者客户）展示数据时的场景印象深刻。不过现如今，除非你的听众都十分老练，否则他们的期望很可能会有所不同。在接下来的几页里，你将看到一个简单的手机应用，这是一个只需几次单击就能够创建出来的最小可视化产品，其目的是向利益相关者传达抽取所得数据的力量，并回到生态系统中，以源网站网络流量的形式展示它能够带来的价值。

我将尽量保持简短的启发式示例，在这里它们将展示如何充分利用你的数据。只有当你有一个具体的应用用于消费数据时，才可以安全地略过本章。本章将会向你展示如何以当下最流行的方式——手机应用，向公众展示你的数据。

4.1 选择手机应用框架

借助于适当的工具向手机应用提供数据将是非常容易的事情。目前有许多优秀的跨平台手机应用开发框架，如 PhoneGap、使用 Appcelerator 云服务的 Appcelerator、jQuery Mobile 和 Sencha Touch。

本章将使用 Appery.io，因为它可以让我们使用 PhoneGap 和 jQuery Mobile 快速创建 iOS、Android、Windows Phone 以及 HTML5 手机应用。我和 Scrapy 都与 Appery.io 无任

何利益关联。我会鼓励你独立进行调研，看看除了本章中提出的功能外，它是否也能符合你的需求。请注意这是一个付费服务，你可以有 14 天的试用期，不过在我看来，它可以让人无需动脑就能快速开发出原型，尤其是对于那些不是网络专家的人来说，为此付费是值得的。我选择该服务的主要原因是它既能提供手机应用，也能提供后端服务，也就是说我们不需要再去配置数据库、编写 REST API 或服务端及手机应用使用其他一些语言。你将看到，我们一行代码都不用去编写！我们将会使用它们的在线工具；在任何时候，你都可以下载该应用，并作为 PhoneGap 项目，使用 PhoneGap 的所有功能。

在本章中，你需要接入互联网连接，以便使用 Appery.io。同时，还需要注意的是该网站的布局可能在未来会有所变化。请将我们的截屏作为参考，而不要在发现该网站外观不同时感到惊讶。

4.2 创建数据库和集合

第一步是通过单击 Appery.io 网站上的 **Sign-Up** 按钮并选取免费方案，来注册免费的 Appery.io 方案。你需要提供用户名、邮箱地址以及密码，然后就会创建好新账户了。等待几秒钟后，账户完成激活。然后就可以登录到 Appery.io 的仪表盘了。现在，开始准备创建新的数据库以及集合，如图 4.1 所示。

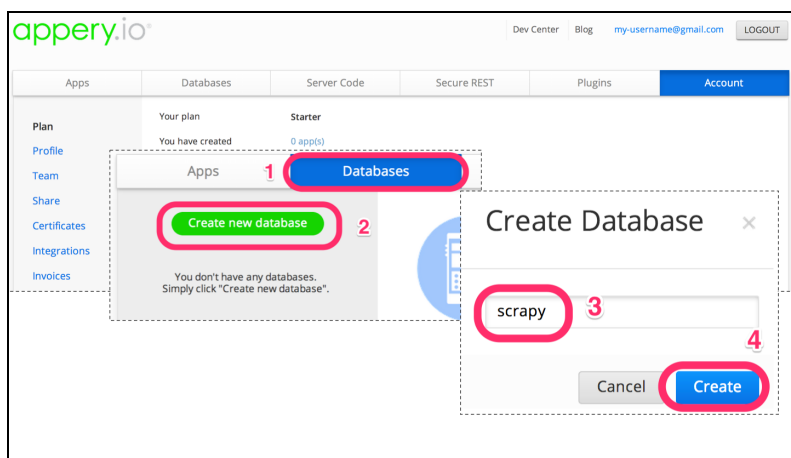


图 4.1 使用 Appery.io 创建新数据库及集合

为了完成该操作，需要按照如下步骤执行。

1. 单击 **Databases** 选项卡 (1)。
2. 然后单击绿色的 **Create new database** (2) 按钮。将新数据库命名为 scrapy (3)。
3. 现在，单击 **Create** 按钮 (4)。此时会自动打开 Scrapy 数据库的仪表盘，在这里，你可以创建新的集合。

在 Appery.io 的术语中，一个数据库是由一组集合组成的。大致来说，一个应用使用一个单独的数据库（至少在最初时是这样），每个数据库中包含多个集合，比如用户、房产、消息等。Appery.io 默认已经提供了一个 **Users** 集合，其中包括用户名和密码（它们有很多内置功能）。图 4.2 所示为创建集合的过程。

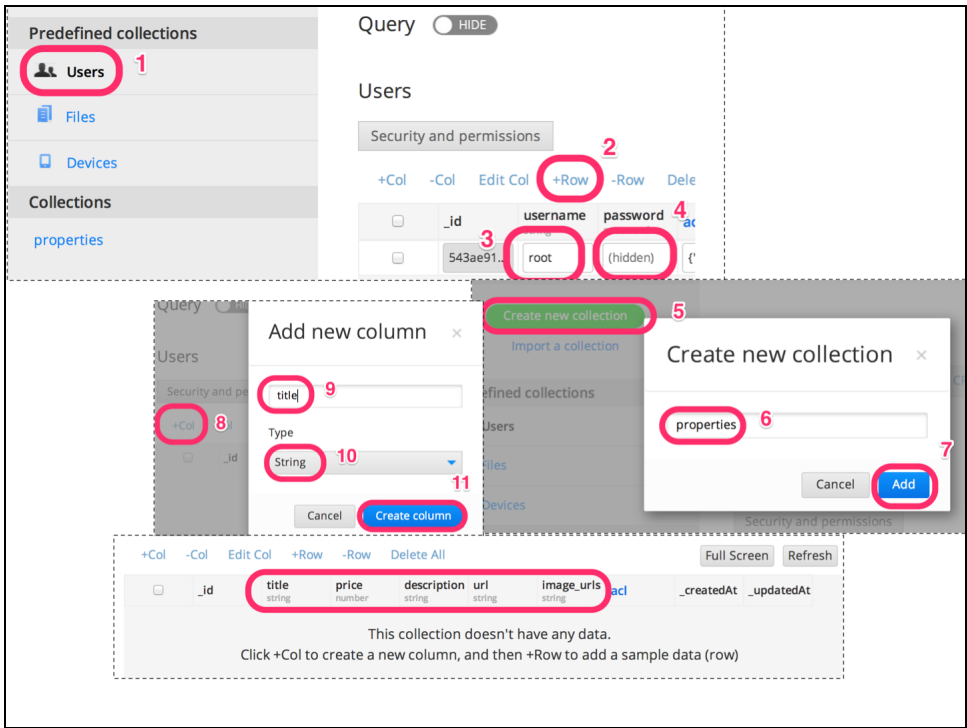


图 4.2 使用 Appery.io 创建新数据库及集合

现在，我们添加一个用户，用户名为 root，密码为 pass。当然，你也可以选择更加安全的用户名和密码。为实现该目的，请单击侧边栏的 **Users** 集合 (1)，然后单击 **+Row**

添加用户/行 (2)。在出现的两个字段中填入用户名和密码 (3) 和 (4)。

我们还需要创建一个新的集合，用于存储 Scrapy 抓取到的房产数据，并将该集合命名为 `properties`。通过单击绿色的 **Create new collection** 按钮 (5)，将其命名为 `properties` (6)，然后单击 **Add** 按钮 (7)，就可以创建新的集合了。现在，我们还必须对该集合进行一些定制化处理。单击 **+Col** 添加数据库列 (8)。每个数据库列都有其类型，用于对值进行校验。除了价格是数值类型外，大部分字段都是简单的字符串类型。我们将通过单击 **+Col** 添加几个列 (8)，并填充列名 (9)，如果不是字符串类型的话，还需要选择类型 (10)，然后单击 **Create column** 按钮 (11)。重复该过程 5 次，创建表 4.1 中展示的列。

表 4.1

列	title	price	description	url	image_urls
类型	string	number	string	string	string

在集合创建的最后，你应该已经将所需的所有列都创建完成了，就像表 4.1 中所示的那样。现在已经准备好从 Scrapy 中导入一些数据了。

4.3 使用 Scrapy 填充数据库

首先，我们需要一个 API key。我们可以在 **Settings** 选项卡 (1) 中找到它。复制该值 (2)，然后单击 **Collections** 选项卡 (3) 回到房产集合中，过程如图 4.3 所示。

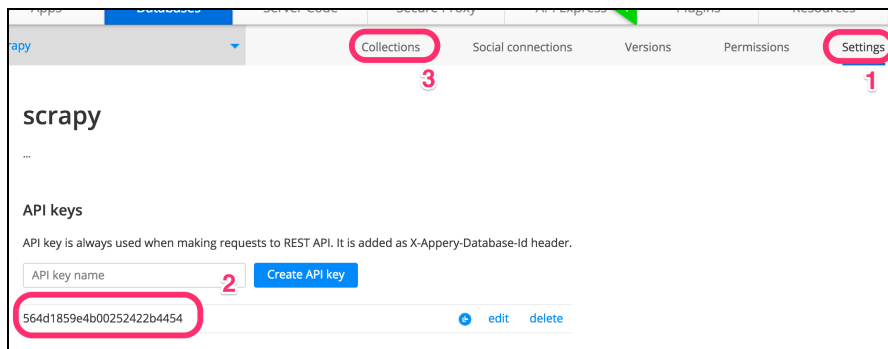


图 4.3 使用 Appery.io 创建新数据库及集合

非常好！现在需要修改在第 3 章中创建的应用，将数据导入到 Appery.io 中。我们先将项目以及名为 easy 的爬虫（easy.py）复制过来，并将该爬虫重命名为 tomobile（tomobile.py）。同时，编辑文件，将其名称设为 tomobile。

```
$ ls
properties scrapy.cfg
$ cat properties/spiders/tomobile.py
...
class ToMobileSpider(CrawlSpider):
    name = 'tomobile'
    allowed_domains = ["scrapybook.s3.amazonaws.com"]

    # Start on the first index page
    start_urls = (
        'http://scrapybook.s3.amazonaws.com/properties/'
        'index_00000.html',
    )
...
```



本章代码可以在 GitHub 的 ch04 目录下找到。

你可能已经注意到的一个问题是，这里并没有使用之前章节中用过的 Web 服务器（http://web:9312），而是使用了该站点的一个公开可用的副本，这是我存放在 http://scrapybook.s3.amazonaws.com 上的副本。之所以在本章中使用这种方式，是因为这样可以使图片和 URL 都能够公开可用，此时就可以非常轻松地分享应用了。

我们将使用 Appery.io 的管道来插入数据。Scrapy 管道通常是一个很小的 Python 类，拥有后置处理、清理及存储 Scrapy Item 的功能。第 8 章将会更深入地介绍这部分的内容。就目前来说，你可以使用 easy_install 或 pip 安装它，不过如果你使用的是我们的 Vagrant dev 机器，则无需进行任何操作，因为我们已经将其安装好了。

```
$ sudo easy_install -U scrapyapperyio
```

或

```
$ sudo pip install --upgrade scrapyapperyio
```

此时，你需要对 Scrapy 的主设置文件进行一些小修改，将之前复制的 API key 添加

进来。第 7 章将会更加深入地讨论设置。现在，我们所需要的就是将如下行添加到 `properties/settings.py` 文件中。

```
ITEM_PIPELINES = {'scrapyapperyio.ApperyIoPipeline': 300}

APPERYIO_DB_ID = '<<Your API KEY here>>'
APPERYIO_USERNAME = 'root'
APPERYIO_PASSWORD = 'pass'
APPERYIO_COLLECTION_NAME = 'properties'
```

不要忘记将 `APPERYIO_DB_ID` 替换为你的 API key。此外，还需要确保设置中的用户名和密码，要和你在 Appery.io 中创建数据库用户时使用的相同。要想向 Appery.io 的数据库中填充数据，请像平常那样启动 scrapy crawl。

```
$ scrapy crawl tomobile -s CLOSESPIDER_ITEMCOUNT=90
INFO: Scrapy 1.0.3 started (bot: properties)
...
INFO: Enabled item pipelines: ApperyIoPipeline
INFO: Spider opened
...
DEBUG: Crawled (200) <GET https://api.appery.io/rest/1/db/login?username=
root&password=pass>
...
DEBUG: Crawled (200) <POST https://api.appery.io/rest/1/db/collections/
properties>
...
INFO: Dumping Scrapy stats:
  {'downloader/response_count': 215,
   'item_scraped_count': 105,
   ...}
INFO: Spider closed (closespider_itemcount)
```

这次的输出会有些不同。可以看到在最开始的几行中，有一行是用于启用 ApperyIoPipeline 这个 Item 管道的；不过最明显的是，你会发现尽管抓取了 100 个 Item，但是却有 200 次请求/响应。这是因为 Appery.io 的管道对每个 Item 都执行了一个到 Appery.io 服务端的额外请求，以便写入每一个 Item。这些带有 `api.appery.io` 这个 URL 的请求同样也会在日志中出现。

当回到 Appery.io 时，可以看到在 **properties** 集合（1）中已经填充好了数据（2），

如图 4.4 所示。

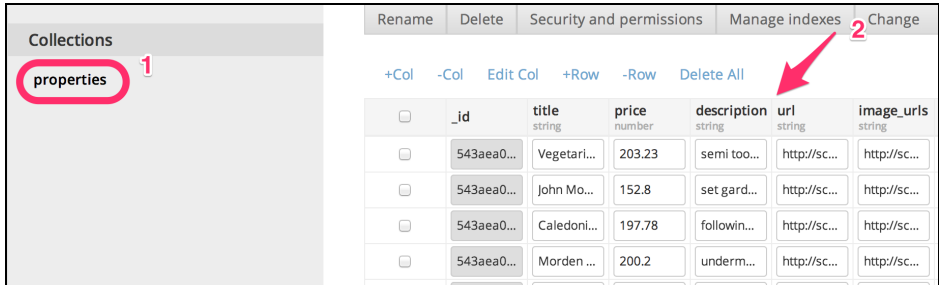


图 4.4 使用数据填充 properties 集合

4.4 创建手机应用

创建一个新的手机应用非常简单。我们只需单击 **Apps** 选项卡 (1)，然后单击绿色的 **Create new app** 按钮 (2)。填写应用名称为 **properties** (3)，然后单击 **Create** 按钮进行创建就可以了，该过程如图 4.5 所示。

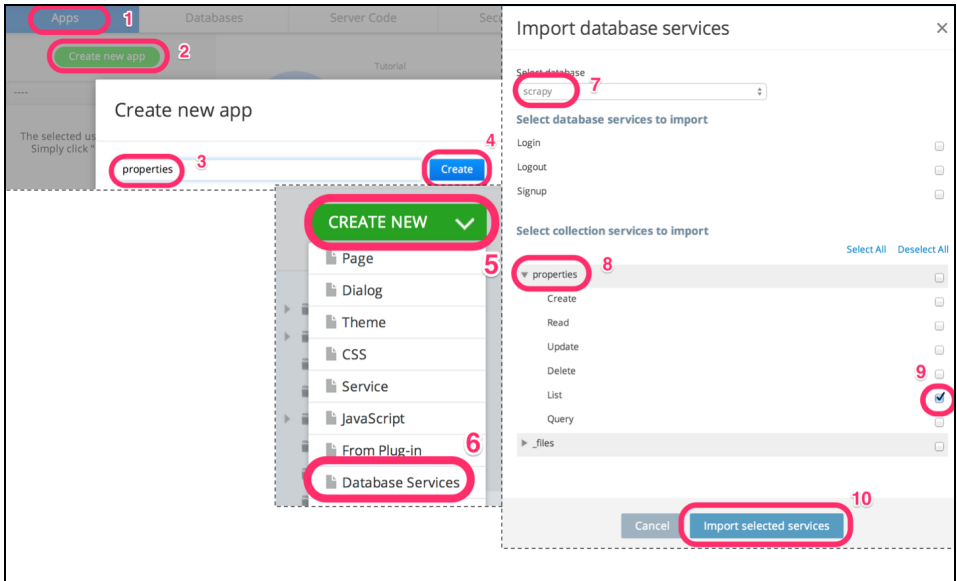


图 4.5 创建新手机应用及数据库集合

4.4.1 创建数据库访问服务

创建新应用时的选项数量可能会有些多。使用 Appery.io 的应用编辑器，可以写出复杂的应用，不过我们将尽可能保持事情简单。我们最初需要的就是创建一个服务，能够让我们从应用中访问 Scrapy 数据库。为了达到这一目的，需要单击长方形的绿色按钮 **CREATE NEW** (5)，然后选择 **Database Services** (6)。这时会弹出一个新的对话框，让我们选择想要连接的数据库。选择 **scrapy** 数据库 (7)。这个菜单中的大部分选项都不会用到，现在只需要单击展开 **properties** 区域 (8)，然后选择 **List** (9)。在后台，它会为我们编写代码，使得我们使用 Scrapy 爬取的数据可以在网络上使用。最后，单击 **Import selected services** 按钮完成 (10)。

4.4.2 创建用户界面

下面将要开始创建应用所有的可视化元素了，这将会使用编辑器中的 **DESIGN** 选项卡来实现，如图 4.6 所示。

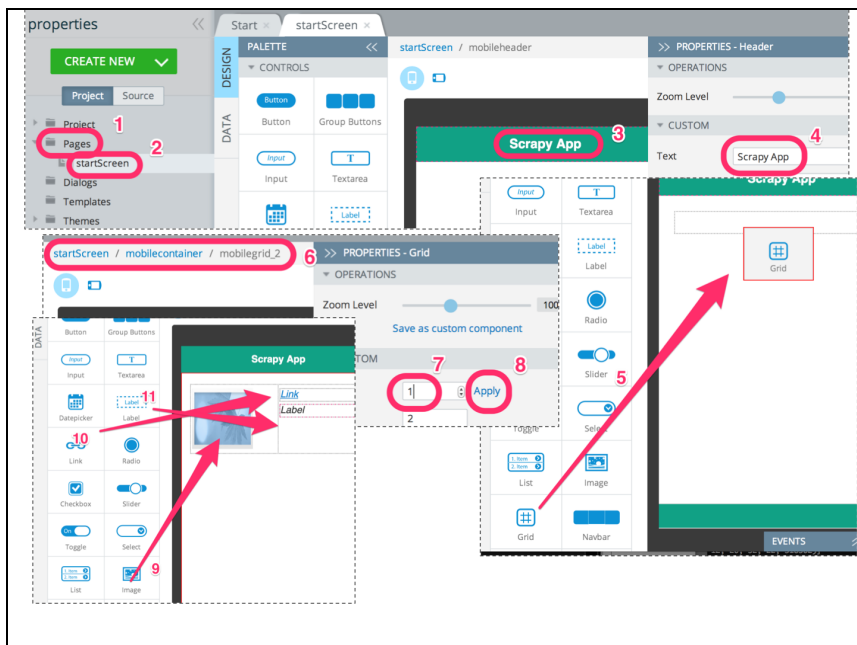


图 4.6 创建用户界面

从页面左侧的树中，展开 **Pages** 文件夹（1），然后单击 **startScreen**（2）。UI 编辑器将会打开该页面，我们可以在其中添加一些控件。下面使用编辑器编辑标题，以便对其更加熟悉。单击头部标题（3），然后会发现屏幕右侧的属性区域会变为显示标题的属性，其中包含一个 **Text** 属性，将该属性值修改为 **Scrapy App**，屏幕中间的标题也会相应地更新。

然后，需要添加一个网格组件，从左侧面板（5）中拖曳 **Grid** 控件即可实现。该控件有两行，而根据我们的需求，只需要一行即可。选择刚刚添加的网格。当手机视图顶部的缩略图区域（6）变灰时，就可以知道该网格已经被选取了。如果没有被选取，单击该网格以便选中。然后右侧的属性栏会更新为网格的属性。这里只需要将 **Rows** 属性设置为 1，然后单击 **Apply** 即可（7）和（8）。现在，该网格就会被更新为只有一行了。

最后，拖拽另外一些控件到网格中。首先要在网格左侧添加图片控件（9），然后在网格右侧添加链接（10），最后在链接下面添加标签（11）。

就布局而言，此时已经足够。接下来将从数据库中向用户界面输入数据。

4.4.3 将数据映射到用户界面

目前为止，我们花费了大量时间在 **DESIGN** 选项卡中，以创建应用的可视化效果。为了将可用的数据链接到这些控件中，需要切换到 **DATA** 选项卡（1），如图 4.7 所示。

选择 **Service**（2）作为数据源类型。由于前面创建的服务是唯一可用的服务，因此它会被自动选取。然后可以继续单击 **Add** 按钮（3），此时服务属性将会在其下方列出。只要按下了 **Add** 按钮，就会看到像 **Before send** 以及 **Success** 这样的事件。我们可以通过单击 **Success** 后面的 **Mapping** 按钮，定制服务成功调用后要做的的事情。

此时会打开 **Mapping action editor**，我们可以在这里完成连线。该编辑器有两侧。左侧是服务响应中可用的字段，而在右侧中可以看到前面步骤中添加的 UI 控件的属性。两侧都有一个 **Expand all** 链接，单击该链接可以看到所有可用的数据和控件。接下来，需要按照表 4.2 中给出的 5 个映射，从左侧向右侧拖曳。

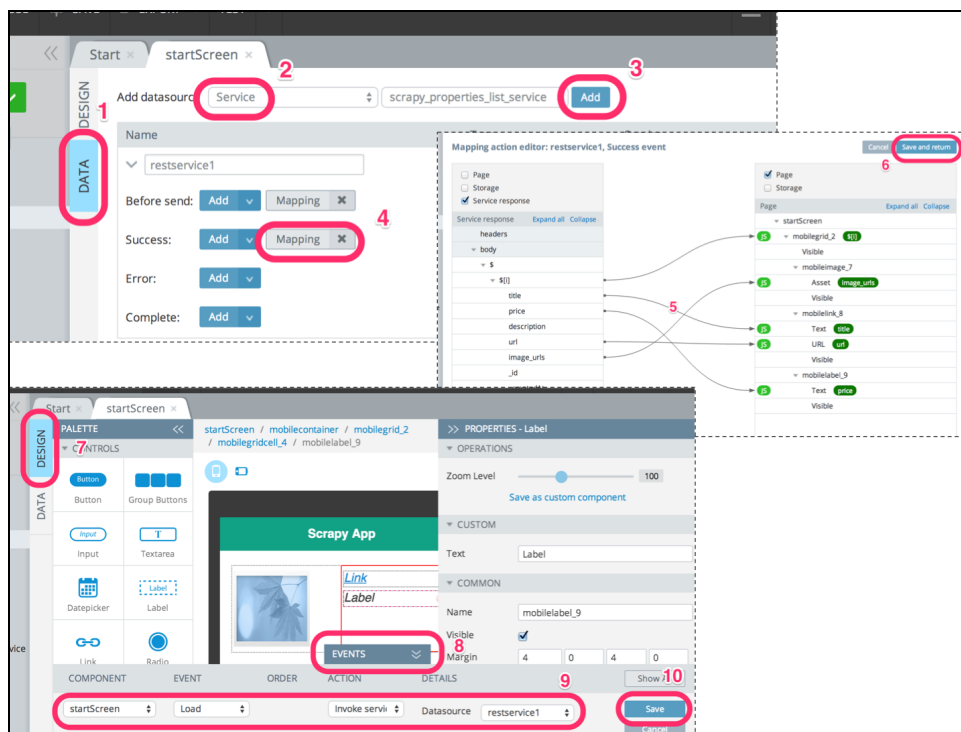


图 4.7 将数据映射到用户界面

表 4.2

响应	组件	属性	备注
<code>\$[i]</code>	mobilegrid_2		使用 for 循环创建每一行
title	mobilelink_8	Text	设置链接文本
price	mobilelabel_9	Text	在文本域中设置价格
image_urls	mobileimage_7	Asset	从图片容器的 URL 中加载图片
url	mobilelink_8	URL	为链接设置 URL。当用户单击时，将会加载关联的页面

4.4.4 数据库字段与用户界面控件间映射

表 4.2 中项的数量可能会与你的情况有些许差别，不过由于每种控件都只有一个，因此出错的可能性非常小。通过设置这些映射，我们通知 Appery.io 在后台编写所有代码，

以便在数据库查询成功时使用数据库中的值加载控件。下面，可以单击 **Save and return** 按钮（6）继续。

此时又回到了 **DATA** 选项卡，如图 4.7 所示。由于还需要返回到 UI 编辑器当中，因此需要单击 **DESIGN** 选项卡（7）。在屏幕下方，你会发现一个 **EVENTS** 区域（8），尽管该区域一直存在，但它刚刚才被展开。在 **EVENTS** 区域中，我们让 Appery.io 做一些事情，作为对 UI 事件的响应。这是我们需要执行的最后一个步骤。它会让应用在 UI 加载完成后立即调用服务取回数据。为了实现该功能，我们需要选择 **startScreen** 作为组件，并将事件保持为默认的 **Load** 选项。然后选择 **Invoke service** 作为 **action**，保持 **Datasource** 为默认的 **restservice1** 选项（9）。最后，单击 **Save**（10），这就是我们为创建这个手机应用所做的所有事情了。

4.4.5 测试、分享及导出你的手机应用

现在，可以测试这个应用了。我们所需要做的事情就是单击 UI 生成器顶部的 **TEST** 按钮（1），如图 4.8 所示。

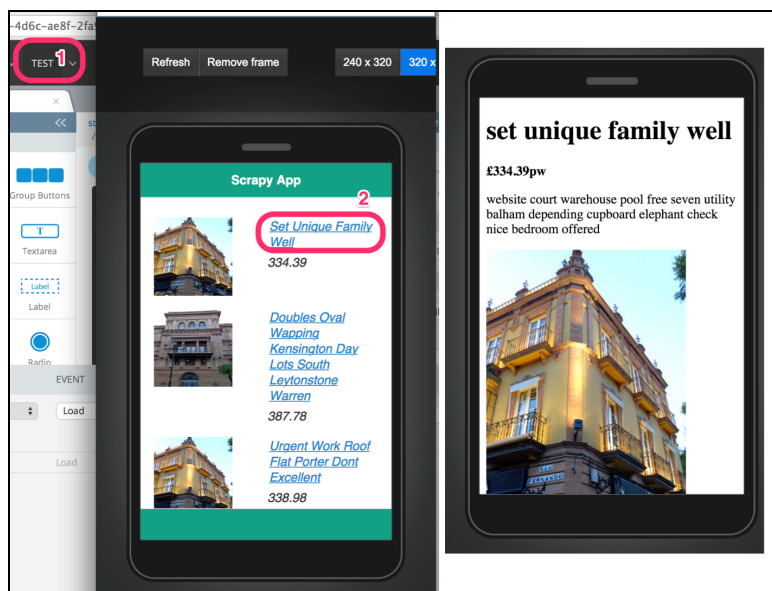


图 4.8 运行在你浏览器中的手机应用

手机应用将会在浏览器中运行。这些链接都是有效的 (2)，可以浏览。可以预览不同的手机屏幕方案以及设备方向，也可以单击 **View on Phone** 按钮，此时会显示一个二维码，你可以使用移动设备扫描该二维码，并预览该应用。你只需分享其生成的链接，其他人也可以在他们的浏览器中尝试该应用。

只需单击几下，我们就可以将 **Scrapy** 抓取的数据组织起来，并展示在手机应用中。如果你需要更进一步地定制该应用，可以参考 **Appery.io** 提供的教程，其网址为 <http://devcenter.appery.io/tutorials/>。当一切准备就绪时，就可以通过 **EXPORT** 按钮导出该应用了，**Appery.io** 提供了非常丰富的导出选项，如图 4.9 所示。

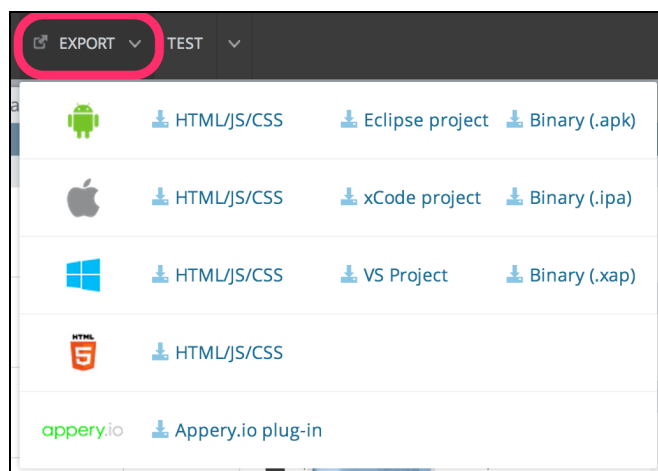


图 4.9 你可以将应用导出到大部分主流移动平台

你可以导出项目文件，在自己喜欢的 IDE 中进一步开发；也可以获得二进制文件，发布到各个平台的手机市场当中。

4.5 本章小结

使用 **Scrapy** 和 **Appery.io** 这两个工具，我们拥有了一个可以抓取网站并且能够将数据插入到数据库中的系统。此外，我们还得到了 **RESTful API**，以及一个简单的可以用于 **Android** 和 **iOS** 的手机应用。对于高级特性和进一步开发，你可以更加深入到这些平

台中，将其中部分开发工作外包给领域专家，或是研究替代方案。现在，你只需要最少的编码，就能够拥有一个可以演示应用理念的最小产品。

你会注意到，在如此短的开发时间中，我们的应用看起来还不错。这是因为它使用了真实的数据，而不是占位符，并且所有链接都是可用且有意义的。我们成功创建了一个尊重其生态（源网站）的最小可用产品，并以流量的形式将价值回馈给源网站。

现在，我们可以开始学习如何使用 Scrapy 爬虫在更加复杂的场景下抽取数据了。

第 5 章

迅速的爬虫技巧

第 3 章关注的是如何从页面中抽取信息，并将其存储到 `Items` 中。我们所学习的内容已经覆盖了大部分常见的 `Scrapy` 用例，足够你创建并运行爬虫了。而在本章中，我们将看到更多特殊的例子，以便让你更加熟悉 `Scrapy` 的两个最重要的类——`Request` 和 `Response`，即我们在第 3 章中提到的 UR²IM 抓取模型中的两个 R。

5.1 需要登录的爬虫

通常情况下，你会发现自己想要抽取数据的网站存在登录机制。大部分情况下，网站会要求你提供用户名和密码用于登录。你可以从 `http://web:9312/dynamic`（从 `dev` 机器访问）或 `http://localhost:9312/dynamic`（从宿主机浏览器访问）找到我们要使用的例子。如果使用 "user" 作为用户名，"pass" 作为密码的话，你就可以访问到包含 3 个房产页面链接的网页。不过现在的问题是，要如何使用 `Scrapy` 执行相同的操作？

让我们使用 Google Chrome 浏览器的开发者工具来尝试理解登录的工作过程（见图 5.1）。首先，打开 **Network** 选项卡（1）。然后，填写用户名和密码，并单击 **Login**（2）。如果用户名和密码正确，你将会看到包含 3 个链接的页面。如果用户名和密码不匹配，将会看到一个错误页。

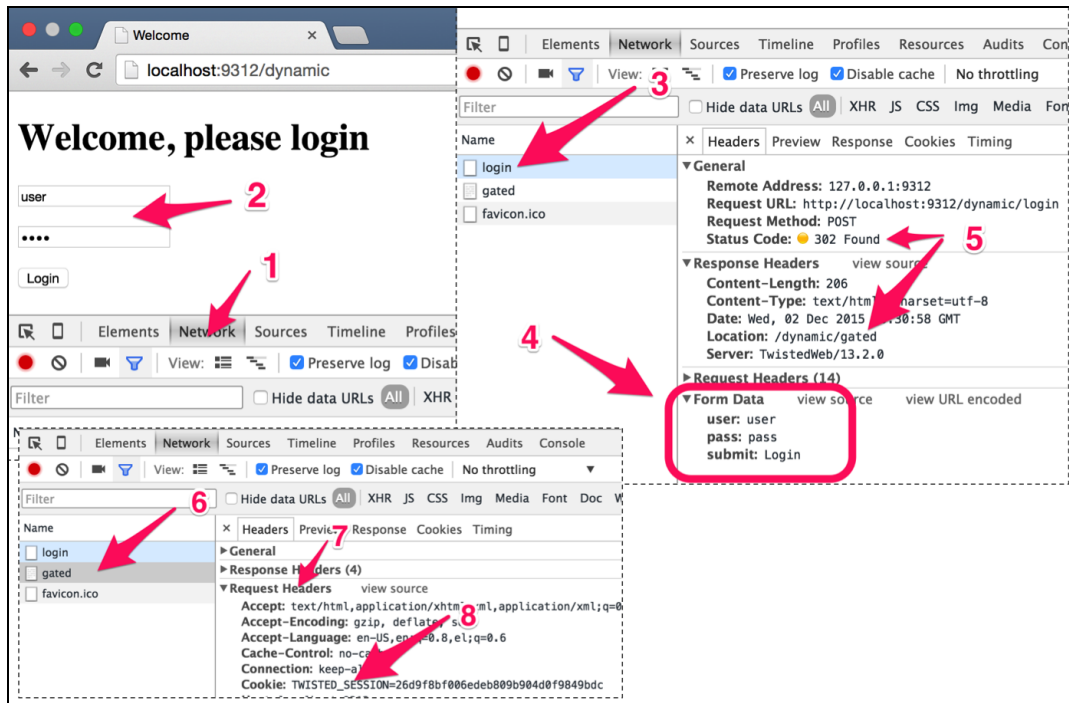


图 5.1 登录网站时的请求和响应

当按下 **Login** 按钮时，会在 Google Chrome 浏览器开发者工具的 **Network** 选项卡中看到一个包含 **Request Method: POST** 的请求，其目的地址为 `http://localhost:9312/dynamic/login`。



前面章节中的请求都是 GET 类型的请求，一般用于获取不会改变的数据，比如简单的网页、图像等。而 POST 类型的请求通常用于获取那些依赖于传送给服务器内容的数据，比如本例中的用户名和密码。

当你单击该请求时（3），可以看到发送给服务端的数据，包括 **Form Data**（4），其中包含了我们输入的用户名和密码。这些数据都是以文本形式传输给服务端的。Chrome 浏览器只是将其组织起来，向我们更好地显示这些数据。服务端的响应是 **302 Found**（5），使我们跳转到一个新的页面：`/dynamic/gated`。该页面只有在登录成功后才会出现。

如果尝试直接访问 `http://localhost:9312/dynamic/gated`，而不输入正确的用户名和密码的话，服务端会发现你在作弊，并跳转到错误页，其地址是 `http://localhost:9312/dynamic/error`。服务端是如何知道你和你密码的呢？如果你单击开发者工具左侧的 **gated** (6)，就会发现在 **Request Headers** 区域下面 (7) 设置了一个 **Cookie** 值 (8)。



HTTP Cookie 是一些服务端发送给浏览器的文本或数值，通常都很短。相应地，浏览器会在随后的每个请求中将其返回给服务端，用于标识你、用户和会话。这样你就可以执行需要服务端状态信息的复杂操作了，比如购物车里的商品或你的用户名和密码。

总之，即使是一个单一的操作，比如登录，也可能涉及包括 POST 请求和 HTTP 跳转的多次服务端往返。Scrapy 能够自动处理大部分操作，而我们需要编写的代码也很简单。

我们从第 3 章中名为 `easy` 的爬虫开始，创建一个新的爬虫，命名为 `login`，保留原有文件，并修改爬虫中的 `name` 属性（如下所示）：

```
class LoginSpider(CrawlSpider):
    name = 'login'
```



本章代码在 GitHub 的 `ch05` 目录下，其中本示例为 `ch05/properties`。

我们需要通过执行到 `http://localhost:9312/dynamic/login` 的 POST 请求，发送登录的初始请求。这将通过 Scrapy 的 `FormRequest` 类实现该功能。该类与第 3 章中使用的 `Request` 类相似，不过该类额外包含一个 `formdata` 参数，可以使用该参数传输表单数据（`user` 和 `pass`）。要想使用该类，首先需要引入如下模块。

```
from scrapy.http import FormRequest
```

然后，将 `start_urls` 语句替换为 `start_requests()` 方法。这样做是因为在本例中，我们需要从一些更加定制化的请求开始，而不仅仅是几个 URL。更确切地说就是，我

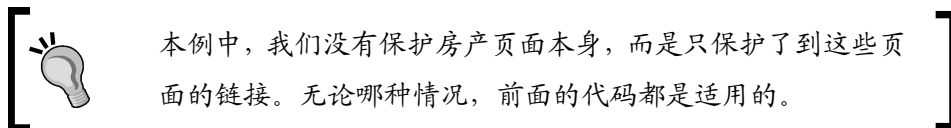
我们从该函数中创建并返回一个 `FormRequest`。

```
# Start with a login request
def start_requests(self):
    return [
        FormRequest(
            "http://web:9312/dynamic/login",
            formdata={"user": "user", "pass": "pass"}
        )
    ]
```

虽然听起来不可思议,但是 `CrawlSpider(LoginSpider 的基类)` 默认的 `parse()` 方法确实处理了 `Response`, 并且仍然能够使用第3章中的 `Rule` 和 `LinkExtractor`。我们只编写了非常少的额外代码,这是因为 `Scrapy` 为我们透明处理了 `Cookie`, 并且一旦我们登录成功,就会在后续的请求中传输这些 `Cookie`, 就和浏览器执行的方式一样。接下来可以像平常一样,使用 `scrapy crawl` 运行。

```
$ scrapy crawl login
INFO: Scrapy 1.0.3 started (bot: properties)
...
DEBUG: Redirecting (302) to <GET .../gated> from <POST .../login >
DEBUG: Crawled (200) <GET .../data.php>
DEBUG: Crawled (200) <GET .../property_000001.html> (referer: .../data.php)
DEBUG: Scraped from <200 .../property_000001.html>
{'address': [u'Plaistow, London'],
 'date': [datetime.datetime(2015, 11, 25, 12, 7, 27, 120119)],
 'description': [u'features'],
 'image_urls': [u'http://web:9312/images/i02.jpg'],
 ...
INFO: Closing spider (finished)
INFO: Dumping Scrapy stats:
{...
  'downloader/request_method_count/GET': 4,
  'downloader/request_method_count/POST': 1,
  ...
  'item_scraped_count': 3,
```

我们可以在日志中看到从 `dynamic/login` 到 `dynamic/gated` 的跳转,然后就会像平时那样抓取 `Item` 了。在统计中,可以看到1个 `POST` 请求和4个 `GET` 请求(一个是前往 `dynamic/gated` 索引页,另外3个是房产页面)。



如果使用了错误的用户名和密码，将会跳转到一个没有任何项目的页面，并且此时爬取过程会被终止，如下面的执行情况所示。

```
$ scrapy crawl login
INFO: Scrapy 1.0.3 started (bot: properties)
...
DEBUG: Redirecting (302) to <GET .../dynamic/error > from <POST .../dynamic/login>
DEBUG: Crawled (200) <GET .../dynamic/error>
...
INFO: Spider closed (closespider_itemcount)
```

这是一个简单的登录示例，用于演示基本的登录机制。大多数网站都会拥有一些更加复杂的机制，不过 Scrapy 也都能够轻松处理。比如，一些网站要求你在执行 POST 请求时，将表单页中的某些表单变量传输到登录页，以便确认 Cookie 是启用的，同样也会让你在尝试暴力破解成千上万次用户名/密码的组合时更加困难。图 5.2 所示即为此种情况的一个示例。

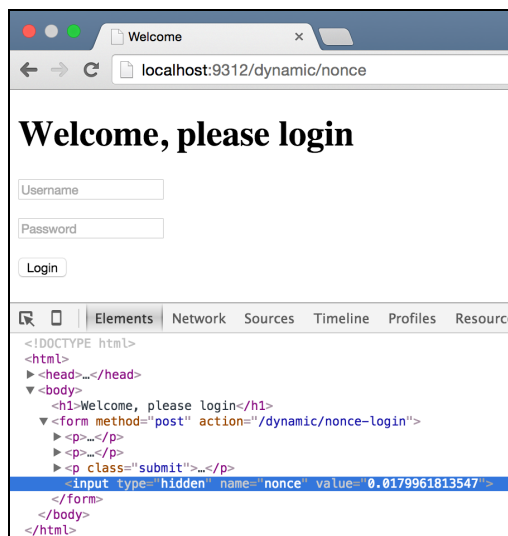


图 5.2 使用一次性随机数的一个更加高级的登录示例的请求和响应情况

比如,当访问 `http://localhost:9312/dynamic/nonce` 时,你会看到一个看起来一样的页面,但是当使用 Chrome 浏览器的开发者工具查看时,会发现页面的表单中有一个叫作 **nonce** 的隐藏字段。当提交该表单时(提交到 `http://localhost:9312/dynamic/nonce-login`),除非你既传输了正确的用户名/密码,又提交了服务端在你访问该登录页时给你的 nonce 值,否则登录不会成功。你无法猜测该值,因为它通常是随机且一次性的。这就表示要想成功登录,现在就需要请求两次了。你必须先访问表单页,然后再访问登录页传输数据。当然,Scrapy 同样拥有内置函数可以帮助我们实现这一目的。

我们创建了一个和之前相似的 `NonceLoginSpider` 爬虫。现在,在 `start_requests()` 中,将返回一个简单的 `Request` (不要忘记引入该模块)到表单页面中,并通过设置其 `callback` 属性为处理方法 `parse_welcome()` 手动处理响应。在 `parse_welcome()` 中,使用了 `FormRequest` 对象的辅助方法 `from_response()`,以创建从原始表单中预填充所有字段和值的 `FormRequest` 对象。`FormRequest.from_response()` 粗略模拟了一次在页面的第一个表单上的提交单击,此时所有字段留空。



花费一些时间让自己熟悉 `from_response()` 的文档是值得的。它有很多非常有用的功能,如 `formname` 和 `formnumber` 可以帮助你拥有多个表单的页面上选择其中某个表单。

该方法对于我们来说非常有用,因为它能够毫不费力地原样包含表单中的所有隐藏字段。我们所需要的就是使用 `formdata` 参数填充 `user` 和 `pass` 字段以及返回 `FormRequest`。下面是其相关代码。

```
# Start on the welcome page
def start_requests(self):
    return [
        Request(
            "http://web:9312/dynamic/nonce",
            callback=self.parse_welcome)
    ]
```

```
# Post welcome page's first form with the given user/pass
def parse_welcome(self, response):
    return FormRequest.from_response(
        response,
        formdata={"user": "user", "pass": "pass"}
    )
```

我们可以像平时一样运行爬虫。

```
$ scrapy crawl noncelogin
INFO: Scrapy 1.0.3 started (bot: properties)
...
DEBUG: Crawled (200) <GET .../dynamic/nonce>
DEBUG: Redirecting (302) to <GET .../dynamic/gated > from <POST .../
dynamic/login-nonce>
DEBUG: Crawled (200) <GET .../dynamic/gated>
...
INFO: Dumping Scrapy stats:
{...
  'downloader/request_method_count/GET': 5,
  'downloader/request_method_count/POST': 1,
  ...
  'item_scraped_count': 3,
```

可以看到，第一个 GET 请求前往 /dynamic/nonce 页面，然后是 POST 请求，跳转到 /dynamic/nonce-login 页面，之后像前面的例子一样跳转到 /dynamic/gated 页面。关于登录的讨论就到这里。该示例使用两个步骤完成登录。只要有足够的耐心，就可以形成任意长链，来执行几乎所有的登录操作。

5.2 使用 JSON API 和 AJAX 页面的爬虫

有时，你会发现自己在页面寻找的数据无法从 HTML 页面中找到。比如，当访问 <http://localhost:9312/static/> 时（见图 5.3），在页面任意位置右键单击 **inspect element**(1,2)，可以看到其中包含所有常见 HTML 元素的 DOM 树。但是，当你使用 scrapy shell 请求，或是在 Chrome 浏览器中右键单击 **View Page Source** (3,4) 时，则会发现该页面的 HTML 代码中并不包含关于房产的任何信息。那么，这些数据是从哪里来的呢？

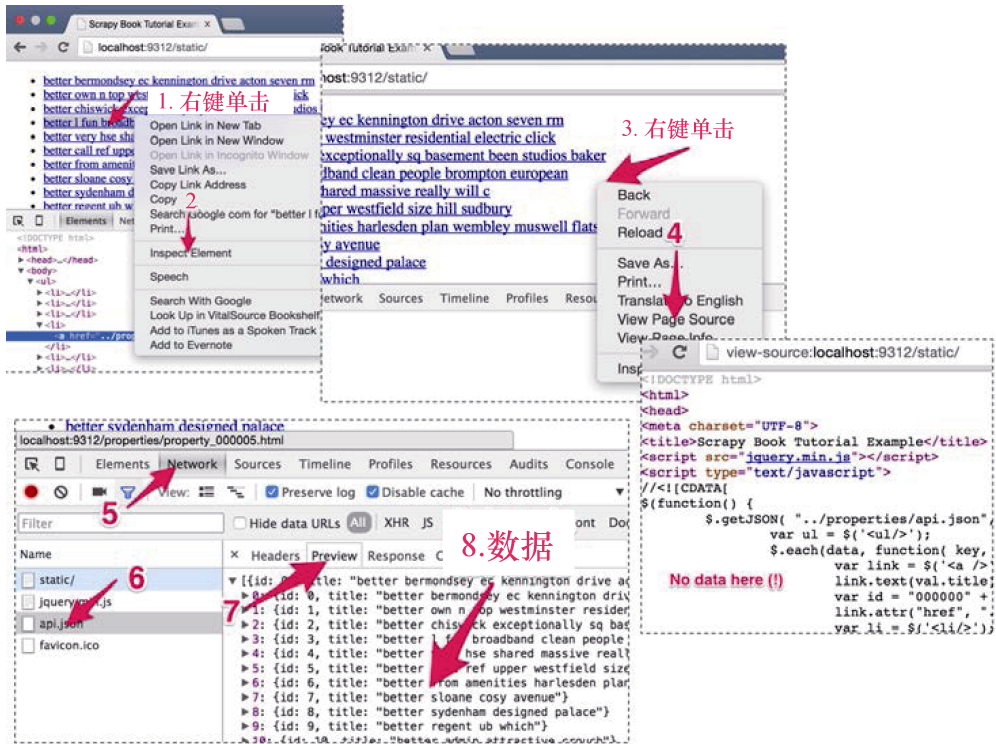


图 5.3 动态加载 JSON 对象时的页面请求与响应

与平常一样，遇到这类例子时，下一步操作应当是打开 Chrome 浏览器开发者工具的 **Network** 选项卡，来看看发生了什么。在左侧的列表中，可以看到加载本页面时 Chrome 执行的请求。在这个简单的页面中，只有 3 个请求：**static/**是刚才已经检查过的请求；**jquery.min.js** 用于获取一个流行的 Javascript 框架的代码；而 **api.json** 看起来会让我们产生兴趣。当单击该请求（6），并单击右侧的 **Preview** 选项卡（7）时，就会发现这里面包含了我们正在寻找的数据。实际上，`http://localhost:9312/properties/api.json` 包含了房产的 ID 和名称（8），如下所示。

```
{
  "id": 0,
  "title": "better set unique family well"
},
... {
```

```
"id": 29,  
"title": "better portered mile"  
}]
```

这是一个非常简单的 JSON API 的示例。更复杂的 API 可能需要你登录, 使用 POST 请求, 或返回更有趣的数据结构。无论在哪种情况下, JSON 都是最简单的解析格式之一, 因为你不需要编写任何 XPath 表达式就可以从中抽取数据。

Python 提供了一个非常好的 JSON 解析库。当我们执行 `import json` 时, 就可以使用 `json.loads(response.body)` 解析 JSON, 将其转换为由 Python 原语、列表和字典组成的等效 Python 对象。

我们将第 3 章的 `manual.py` 拷贝过来, 用于实现该功能。在本例中, 这是最佳的起始选项, 因为我们需要通过在 JSON 对象中找到的 ID, 手动创建房产 URL 以及 Request 对象。我们将该文件重命名为 `api.py`, 并将爬虫类重命名为 `ApiSpider`, `name` 属性修改为 `api`。新的 `start_urls` 将会是 JSON API 的 URL, 如下所示。

```
start_urls = (  
    'http://web:9312/properties/api.json',  
)
```

如果你想执行 POST 请求, 或是更复杂的操作, 可以使用前一节中介绍的 `start_requests()` 方法。此时, **Scrapy** 将会打开该 URL, 并调用包含以 `Response` 为参数的 `parse()` 方法。可以通过 `import json`, 使用如下代码解析 JSON 对象。

```
def parse(self, response):  
    base_url = "http://web:9312/properties/"  
    js = json.loads(response.body)  
    for item in js:  
        id = item["id"]  
        url = base_url + "property_%06d.html" % id  
        yield Request(url, callback=self.parse_item)
```

前面的代码使用了 `json.loads(response.body)`, 将 `Response` 这个 JSON 对象解析为 Python 列表, 然后迭代该列表。对于列表中的每一项, 我们将 URL 的 3 个部分 (`base_url`、`property_%06d` 以及 `.html`) 组合到一起。`base_url` 是在前面定义的 URL 前缀。`%06d` 是 Python 语法中非常有用的一部分, 它可以让我们结合 Python

变量创建新的字符串。在本例中，`%06d` 将会被变量 `id` 的值替换（本行结尾处`%`后面的变量）。`id` 将会被视为数字（`%d` 表示视为数字），并且如果不满 6 位，则会在前面加上 0，扩展成 6 位字符。比如，`id` 值为 5，`%06d` 将会被替换为 000005，而如果 `id` 为 34322，`%06d` 则会被替换为 034322。最终结果正是我们房产页面的有效 URL。我们使用该 URL 形成一个新的 Request 对象，并像第 3 章一样使用 `yield`。然后可以像平时那样使用 `scrapy crawl` 运行该示例。

```
$ scrapy crawl api
INFO: Scrapy 1.0.3 started (bot: properties)
...
DEBUG: Crawled (200) <GET ...properties/api.json>
DEBUG: Crawled (200) <GET .../property_000029.html>
...
INFO: Closing spider (finished)
INFO: Dumping Scrapy stats:
...
    'downloader/request_count': 31, ...
    'item_scraped_count': 30,
```

你可能会注意到结尾处的状态是 31 个请求——每个 Item 一个请求，以及最初的 `api.json` 的请求。

5.2.1 在响应间传参

很多情况下，在 JSON API 中会有感兴趣的信息，你可能想要将它们存储到 Item 中。在我们的示例中，为了演示这种情况，JSON API 会在给定房产信息的标题前面加上 "better"。比如，房产标题是 "Covent Garden"，API 就会将标题写为 "Better Covent Garden"。假设我们想要将这些 "better" 开头的标题存储到 Items 中，要如何将信息从 `parse()` 方法传递到 `parse_item()` 方法呢？

不要感到惊讶，通过在 `parse()` 生成的 Request 中设置一些东西，就能实现该功能。之后，可以从 `parse_item()` 接收到的 Response 中取得这些信息。Request 有一个名为 `meta` 的字典，能够直接访问 Response。比如在我们的例子中，可以在该字典中设置标题值，以存储来自 JSON 对象的标题。