

- 第1章 Scrapy介绍
  - HelloScrapy
  - 喜爱Scrapy的其它理由
  - 关于此书：目标和用法
  - 掌握自动抓取数据的重要性
  - 开发高可靠高质量的应用 提供真实的开发进度表
  - 快速开发最小化可行产品
  - 网络抓取让你的应用快速成长 —— Google不能使用表格
  - 发现并实践
  - 在充满爬虫的网络世界做守法公民
  - Scrapy不是什么
  - 总结
- 第2章 理解HTML和XPath
  - HTML、DOM树结构和XPath
  - HTML文档
  - 树结构
  - 浏览器中的页面
  - 使用Chrome浏览器获得XPath表达式
  - 常见工作
  - 提前应对网页发生改变
  - 总结
- 第3章 爬虫基础
  - 安装Scrapy
  - UR2IM——基础抓取过程
  - 编写爬虫
- 第4章 从Scrapy到移动应用
  - 选择移动应用框架
  - 创建数据库和集合
  - **用Scrapy导入数据**
  - 创建移动应用
  - 创建数据库接入服务
  - 将数据映射到用户界面
  - 映射数据字段和用户组件
  - 测试、分享、生成app
  - 总结
- 第5章 快速构建爬虫
  - 一个具有登录功能的爬虫

- 使用JSON APIs和AJAX页面的爬虫
- 在响应间传递参数
- 一个加速30倍的项目爬虫
- 可以抓取Excel文件的爬虫
- 总结
- 第6章 Scrapinghub部署
  - 注册、登录、创建项目
  - 部署爬虫并制定计划
  - 访问文件
  - 制定周期抓取
  - 总结
- 第7章 配置和管理
  - 使用Scrapy设置
  - 基本设置
  - 案例1——使用远程登录
  - 性能
  - HTTP缓存和脱机工作
  - 案例2——用缓存离线工作
  - 抓取方式
  - 案例3——下载图片
  - 亚马逊网络服务
  - 案例4——使用代理和Crawlera的智慧代理
  - 更多的设置
  - 和项目相关的设定
  - 扩展Scrapy设置
  - 微调下载
  - 自动限定扩展设置
  - 内存使用扩展设置
  - 登录和调试
  - 总结
- 第8章 Scrapy编程
  - Scrapy是一个Twisted应用
  - 延迟项和延迟链
  - 理解Twisted和非阻塞I/O——Python的故事
  - Scrapy架构概要
  - 案例1——一个简单的pipeline
  - 信号

- 案例2——一个可以测量吞吐量和延迟的扩展
- 进一步扩展中间件
- 总结
- 第9章 使用Pipelines
  - 使用REST APIs
  - 使用treq
  - 一个写入Elasticsearch的pipeline
  - pipeline使用Google Geocoding API进行地理编码
  - 在Elasticsearch进行地理索引
  - 连接数据库与Python客户端
  - 用pipeline写入MySQL
  - 使用Twisted 特定客户端连接服务
  - 用pipeline读写Redis
  - 连接CPU密集型、阻塞或旧方法
  - pipeline进行CPU密集型和阻塞操作
  - pipeline使用二进制和脚本
  - 总结
- 第10章 理解Scrapy的性能
  - Scrapy的引擎——一个直观的方法
  - 串联排队系统
  - 确认瓶颈
  - Scrapy的性能模型
  - 使用远程登录控制组件
  - 评分系统
  - 标准性能模型
  - 解决性能问题
  - 实例1——CPU满负荷
  - 实例2-阻塞代码
  - 实例3-下载器中有“垃圾”
  - 实例4-大量响应造成溢出
  - 实例5-item并发受限/过量造成溢出
  - 实例6-下载器没有充分运行
  - 解决问题的流程
  - 总结
- 第11章 Scrapyd分布式抓取和实时分析
  - 房子的标题如何影响价格?
  - Scrapyd
  - 分布式系统概述
  - 修改爬虫和中间件

- 抓取共享首页
- 批次抓取URL
- 从settings启动URL
- 将项目部署到scrapyd服务器
- 创建自定义监视命令
- 用Apache Spark streaming计算偏移值
- 进行分布式抓取
- 系统性能
- 要点
- 总结

# 第1章 Scrapy介绍

---

欢迎来到Scrapy之旅。通过这本书，我们希望你可以从只会一点或零基础的初学者，达到熟练使用这个强大的框架海量抓取网络和其他资源的水平。在本章里，我们会向你介绍Scrapy，以及Scrapy能做什么。

## HelloScrapy

---

Scrapy是一个健壮的抓取网络资源的框架。作为互联网使用者，你可能经常希望可以将网上的资源保存到Excel中（见第3章），以便离线时使用或进行计算。作为开发者，你可能经常希望将不同网站的资源整合起来，但你清楚这么做的复杂性。Scrapy可以帮助你完成简单和复杂的数据提取。

Scrapy是利用健壮高效的方式提取网络资源的多年经验开发的。使用Scrapy，你只需进行一项设置，就可以抵过其它框架使用多个类、插件和配置。看一眼第7章，你就可以知道仅需几行代码就可以完成大量工作。

从开发者的角度，你会喜欢Scrapy的基于事件的架构（见第8章和第9章）。它可以让我们进行串联操作，清洗、形成、丰富数据，或存入数据库等等，同时不会有太大的性能损耗。从技术上说，基于事件的机制，Scrapy可以让吞吐量摆脱延迟，同时开放数千个连接。举一个极端的例子，假设你要从一个网站提取列表，每页有100个列表项。Scrapy可以轻松的同时处理16个请求，假设每个请求在一秒内完成，每秒就可以抓取16个页面。乘以每页的列表数，每秒就可以抓取1600个列表项。然后，你想将每个列表项写入一个高并发的云存储，每个要花3秒。为了支持每秒16个请求，必须要并行进行4800个写入请求（第9章你会看到更多类似的计算）。对于传统的多线程应用，这需要4800个线程，对你和操作系统都是个挑战。在Scrapy中，4800个并发请求很平常，只要操作系统支持就行。更进一步，Scrapy的内存要求和你要抓取的列表项的数据量相关，而对于多线程应用，每个线程的大小都和一个列表的大小相当。

简而言之，速度慢或不可预测的网站、数据库或远程API不会对Scrapy的性能造成影响，因为你可以进行并发请求，用单线程管理。相比于多线程应用，使用更简单的代码反而可以同时运行几个抓取器和其它应用，这样就可以降低费用。

## 喜爱Scrapy的其它理由

---

Scrapy出现已经有五年多了，现在已经成熟稳定。除了前面提到的性能的优点，以下是Scrapy其它让人喜爱的理由：

- Scrapy可以读懂破损的HTML

你可以在Scrapy上直接使用BeautifulSoup或lxml，但Scrapy提供Selector，一个相比lxml更高级的XPath解析器。它可以有效的处理破损的HTML代码和费解的编码。

- 社区

Scrapy有一个活跃的社区。可以查看Scrapy的邮件列表<https://groups.google.com/forum/#!forum/scrapy-users>和Stack Overflow上的数千个问题<http://stackoverflow.com/questions/tagged/scrapy>。多数问题在数分钟之内就会得到解答。<http://scrapy.org/community/>有更多的社区资源。

- 由社区维护的组织清晰的代码

Scrapy需要用标准的方式组织代码。你用Python来写爬虫和pipelines，就可以自动使引擎的效率提高。如果你在网上搜索，你会发现许多人有使用Scrapy的经验。这意味着，可以方便地找人帮你维护或扩展代码。无论是谁加入你的团队，都不必经过学习曲线理解你特别的爬虫。

- 注重质量的更新

如果查看版本记录（<http://doc.scrapy.org/en/latest/news.html>），你会看到有不断的更新和稳定性/错误修正。

## 关于此书：目标和用法

---

对于此书，我们会用例子和真实的数据教你使用Scrapy。大多数章节，要抓取的都是一个房屋租赁网站。我们选择它的原因是，它很有代表性，并可以进行一定的变化，同时也很简单。使用这个例子，可以让我们专注于Scrapy。

我们会从抓取几百页开始，然后扩展到抓取50000页。在这个过程中，我们会教你如何用Scrapy连接MySQL、Redis和Elasticsearch，使用Google geocoding API找到给定地点的坐标，向Apache Spark传入数据，预测影响价格的关键词。

你可能需要多读几遍本书。你可以粗略地浏览一遍，了解一下结构，然后仔细读一两章、进行学习和试验，然后再继续读。如果你对哪章熟悉的话，可以跳过。如果你熟悉HTML和XPath的话，就没必要在第2章浪费太多时间。某些章如第8章，既是示例也是参考，具有一定深度。它就需要你多读几遍，每章之间进行数周的练习。如果没有完全搞懂第8章的话，也可以读第9章的具体应用。后者可以帮你进一步理解概念。

我们已经尝试调整本书的结构，以让其既有趣也容易上手。但我们做不到用这本书教给你如何使用Python。Python的书有很多，但我建议你在学习的过程中尽量保持放松。Python流行的原因之一是，它很简洁，可以像读英语一样读代码。对于Python初学者和专家，Scrapy都是一个高级框架。你可以称它为“Scrapy语言”。因此，我建议你直接从实例学习，如果你觉得Python语法有困难的话，再进行补充学习，可以是在线的Python教程或Coursera的初级课程。放心，就算不是Python专家，你也可以成为一个优秀的Scrapy开发者。

## 掌握自动抓取数据的重要性

---

对于许多人，对Scrapy这样的新技术有好奇心和满足感，就是学习的动力。学习这个框架的同时，我们可以从数据开发和社区，而不是代码，获得额外的好处。

## 开发高可靠高质量的应用 提供真实的开发进度表

---

为了开发新颖高质量的应用，我们需要真实和大量的数据，如果可能的话，最好在写代码之前就有数据。现在的软件开发都要实时处理海量的瑕疵数据，以获取知识和洞察力。当软件应用到海量数据时，错误和疏忽很难检测出来，就会造成后果严重的决策。例如，在进行人口统计时，很容易忽略一整个州，仅仅是因为这个州的名字太长，它的数据被丢弃了。通过细心的抓取，有高质量的、海量的真实数据，在开发和设计的过程中，就可以找到并修复bug，然后才能做出正确的决策。

另一个例子，假设你想设计一个类似亚马逊的“如果你喜欢这个，你可能也喜欢那个”的推荐系统。如果在开始之前，你就能抓取手机真实的数据，你就可以快速知道一些问题，比如无效记录、打折商品、重复、无效字符、因为分布导致的性能问题。数据会强制你设计健壮的算法以处理被数千人抢购或无人问津的商品。相比较于数周开发之后却碰到现实问题，这两种方法可能最终会一致，但是在一开始就能对整个进程有所掌握，意义肯定是不同的。从数据开始，可以让软件的开发过程更为愉悦和有预测性。

## 快速开发最小化可行产品

---

海量真实数据对初创企业更为重要。你可能听说过“精益初创企业”，这是Eric Ries发明的词，用来描述高度不确定的企业发展阶段，尤其是技术初创企业。它的核心概念之一就是最小化可行产

品（MVP），一个只包含有限功能的产品，快速开发并投放，以检测市场反应、验证商业假设。根据市场反应，初创企业可以选择追加投资，或选择其他更有希望的项目。

很容易忽略这个过程中的某些方面，这些方面和数据问题密切相关，用Scrapy可以解决数据问题。当我们让潜在用户尝试移动App时，例如，作为开发者或企业家，我们让用户来判断完成的App功能如何。这可能对非专家的用户有点困难。一个应用只展示“产品1”、“产品2”、“用户433”，和另一个应用展示“Samsung UN55J6200 55-Inch TV”，用户“Richard S.”给它打了五星评价，并且有链接可以直接打开商品主页，这两个应用的差距是非常大的。很难让人们客观地对MVP进行评价，除非它使用的数据是真实可信的。

一些初创企业事后才想到数据，是因为考虑到采集数据很贵。事实上，我们通常都是打开表格、屏幕、手动输入数据，或者我们可以用Scrapy抓取几个网站，然后再开始写代码。第4章中，你可以看到如何快速创建一个移动App以使用数据。

## 网络抓取让你的应用快速增长 —— Google不能使用表格

---

让我们来看看表格是如何影响一个产品的。假如谷歌的创始人创建了搜索引擎的第一个版本，但要求每个网站站长填入信息，并复制粘贴他们的每个网页的链接。他们然后接受谷歌的协议，让谷歌处理、存储、呈现内容，并进行收费。可以想象整个过程工作量巨大。即使市场有搜索引擎的需求，这个引擎也成为不了谷歌，因为它的成长太慢了。即使是最复杂的算法也不能抵消缺失数据。谷歌使用网络爬虫逐页抓取，填充数据库。站长完全不必做任何事。实际上，想屏蔽谷歌，还需要做一番努力。

让谷歌使用表格的主意有点搞笑，但是一个普通网站要用户填多少表呢？登录表单、列表表单、勾选表单等等。这些表单会如何遏制应用的市场扩张？如果你足够了解用户，你会知道他们还会使用其它什么网站，或许已经有了账户。例如，开发者可能有Stack Overflow和GitHub账户。经过用户同意，你能不能直接用这些账户就自动填入照片、介绍和最近的帖子呢？你能否对这些帖子做文本分析，根据结果设置网站的导航结构、推荐商品或服务呢？我希望你能看到将表格换为自动数据抓取可以更好的为用户服务，使网站快速增长。

## 发现并实践

---

抓取数据自然而然会让你发现和思考你和被抓取目标的关系。当你抓取一个数据源时，自然会有一些问题：我相信他们的数据吗？我相信提供数据的公司吗？我应该和它们正式商谈合作吗？我和他们有竞争吗？从其他渠道获得数据花费是多少？这些商业风险是必然存在的，但是抓取数据可以让我们更早地知道，进行应对。

你还想知道如何反馈给这些网站或社区？给他们免费流量，他们肯定很高兴。另一方面，如果你

的应用不能提供价值，继续合作的可能就会变小，除非找到另外合作的方式。通过从各种渠道获得数据，你可以开发对现有生态更友好的产品，甚至打败旧产品。或者，老产品能帮助你扩张，例如，你的应用数据来自两个或三个不同的生态圈，每个生态圈都有十万名用户，结合起来，你的应用或许就能惠及三十万人。假如你的初创企业结合了摇滚乐和T恤印刷行业，就将两个生态圈结合了起来，你和这两个社区都可以得到扩张。

## 在充满爬虫的网络世界做守法公民

---

开发爬虫还有一些注意事项。不负责任的网络抓取让人不悦，有时甚至是犯罪。两个最重要的要避免的就是拒绝访问攻击（DoS）和侵犯著作权。

对于第一个，普通访问者每隔几秒才访问一个新页面。爬虫的话，每秒可能下载几十个页面。流量超过普通用户的十倍。这会让网站的拥有者不安。使用阻塞器降低流量，模仿普通用户。检测响应时间，如果看到响应时间增加，则降低抓取的强度。好消息是Scrapy提供了两个现成的方法（见第7章）。

对于著作权，可以查看网站的著作权信息，以确认什么可以抓取什么不能抓取。大多数站点允许你处理网站的信息，只要不复制并宣称是你的。一个好的方法是在你请求中使用一个User-Agent字段，告诉网站你是谁，你想用他们的数据做什么。Scrapy请求默认使用你的BOT\_NAME作为User-Agent。如果这是一个URL或名字，可以直接指向你的应用，那么源网站的站长就可以访问你的站点，并知道你用他的数据做什么。另一个重要的地方，允许站长可以禁止爬虫访问网站的某个区域。Scrapy提供了功能（RobotsTxtMiddleware），以尊重源网站列在robots.txt文件的意见（在<http://www.google.com/robots.txt>可以看到一个例子）。最后，最好提供可以让站长提出拒绝抓取的方法。至少，可以让他们很容易地找到你，并提出交涉。

每个国家的法律不同，我无意给出法律上的建议。如果你觉得需要的话，请寻求专业的法律建议。这适用于整本书的内容。

## Scrapy不是什么

---

最后，因为数据抓取和相关的名词定义很模糊，或相互使用，很容易误解Scrapy。我这里解释一下，避免发生误解。

Scrapy不是Apache Nutch，即它不是一个原生的网络爬虫。如果Scrapy访问一个网站，它对网站一无所知，就不能抓取任何东西。Scrapy是用来抓取结构化的信息，并需要手动设置XPath和CSS表达式。Apache Nutch会取得一个原生网页并提取信息，例如关键词。它更适合某些应用，而不适合其它应用。

Scrapy不是Apache Solr、Elasticsearch或Lucene；换句话说，它和搜索引擎无关。Scrapy不是

用来给包含“爱因斯坦”的文档寻找参考。你可以使用Scrapy抓取的数据，并将它们插入到Solr或Elasticsearch，如第9章所示，但这只是使用Scrapy的一种途径，而不是嵌入Scrapy的功能。

最后，Scrapy不是类似MySQL、MongoDB、Redis的数据库。它不存储和索引数据。它只是提取数据。也就是说，你需要将Scrapy提取的数据插入到数据库中，可行的数据库有多种。虽然Scrapy不是数据库，它的结果可以方便地输出为文件，或不进行输出。

## 总结

---

在本章中，我们向你介绍了Scrapy以及它的作用，还有使用这本书的最优方法。通过开发与市场完美结合的高质量应用，我们还介绍了几种自动抓取数据能使你获益的方法。下一章会介绍两个极为重要的网络语言，HTML和XPath，我们在每个Scrapy项目中都会用到。

# 第2章 理解HTML和XPath

---

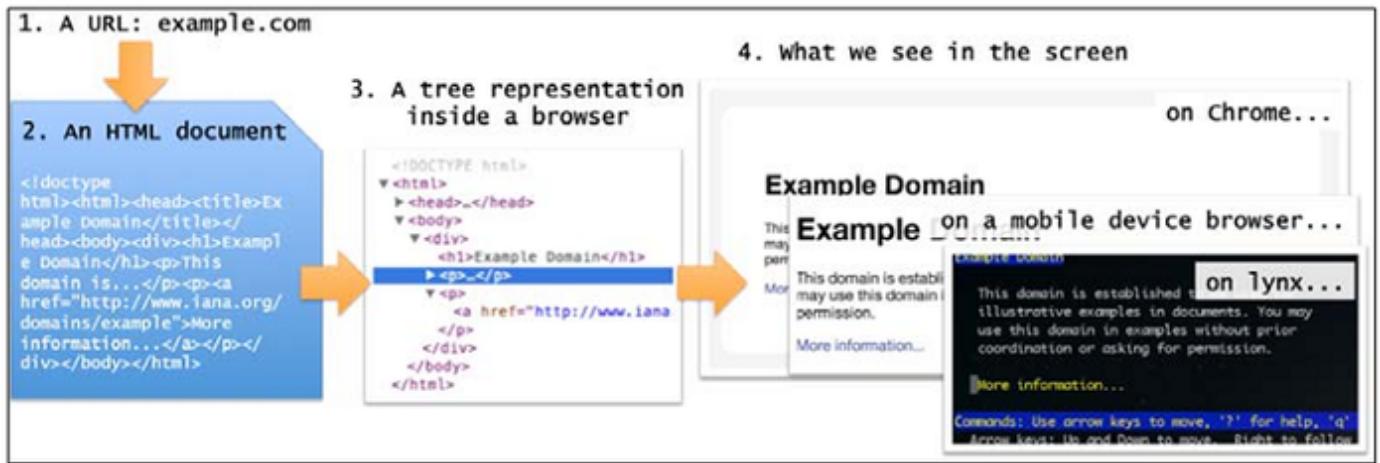
为了从网页提取信息，了解网页的结构是非常必要的。我们会快速学习HTML、HTML的树结构和用来筛选网页信息的XPath。

## HTML、DOM树结构和XPath

---

从这本书的角度，键入网址到看见网页的整个过程可以分成四步：

- 在浏览器中输入网址URL。URL的第一部分，也即域名（例如gumtree.com），用来搜寻网络上的服务器。URL和其他像cookies等数据形成了一个发送到服务器的请求request。
- 服务器向浏览器发送HTML。服务器也可能发送XML或JSON等其他格式，目前我们只关注HTML。
- HTML在浏览器内部转化成树结构：文档对象模型（DOM）。
- 根据布局规范，树结构转化成屏幕上的真实页面。



研究下这四个步骤和树结构，可以帮助定位要抓取的文本和编写爬虫。

## URL

URL包括两部分：第一部分通过DNS定位服务器，例如当你在浏览器输入`https://mail.google.com/mail/u/0/#inbox`这个地址时，产生了一个`mail.google.com`的DNS请求，后者为你解析了一台服务器的IP地址，例如173.194.71.83。也就是说，`https://mail.google.com/mail/u/0/#inbox`转换成了`https://173.194.71.83/mail/u/0/#inbox`。

URL其余的部分告诉服务器这个请求具体是关于什么的，可能是一张图片、一份文档或是触发一个动作，例如在服务器上发送一封邮件。

## HTML文档

服务器读取URL，了解用户请求，然后回复一个HTML文档。HTML本质是一个文本文件，可以用TextMate、Notepad、vi或Emacs等软件打开。与大多数文本文件不同，HTML严格遵循万维网联盟（World Wide Web Consortium）的规定格式。这个格式超出了本书的范畴，这里只看一个简单的HTML页面。如果你打开`http://example.com`，点击查看源代码，就可以看到HTML代码，如下所示：

```
<!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>
```

为了便于阅读，我美化了这个HTML文档。你也可以把整篇文档放在一行里。对于HTML，大多数情况下，空格和换行符不会造成什么影响。

尖括号里的字符称作标签，例如或。是起始标签，是结束标签。标签总是成对出现。某些网页没有结束标签，例如只用标签分隔段落，浏览器对这种行为是容许的，会智能判断哪里该有结束标签。

与之间的内容称作HTML的元素。元素之间可以嵌套元素，比如例子中的标签，和第二个标签，后者包含了一个标签。

有些标签稍显复杂，例如，带有URL的href部分称作属性。

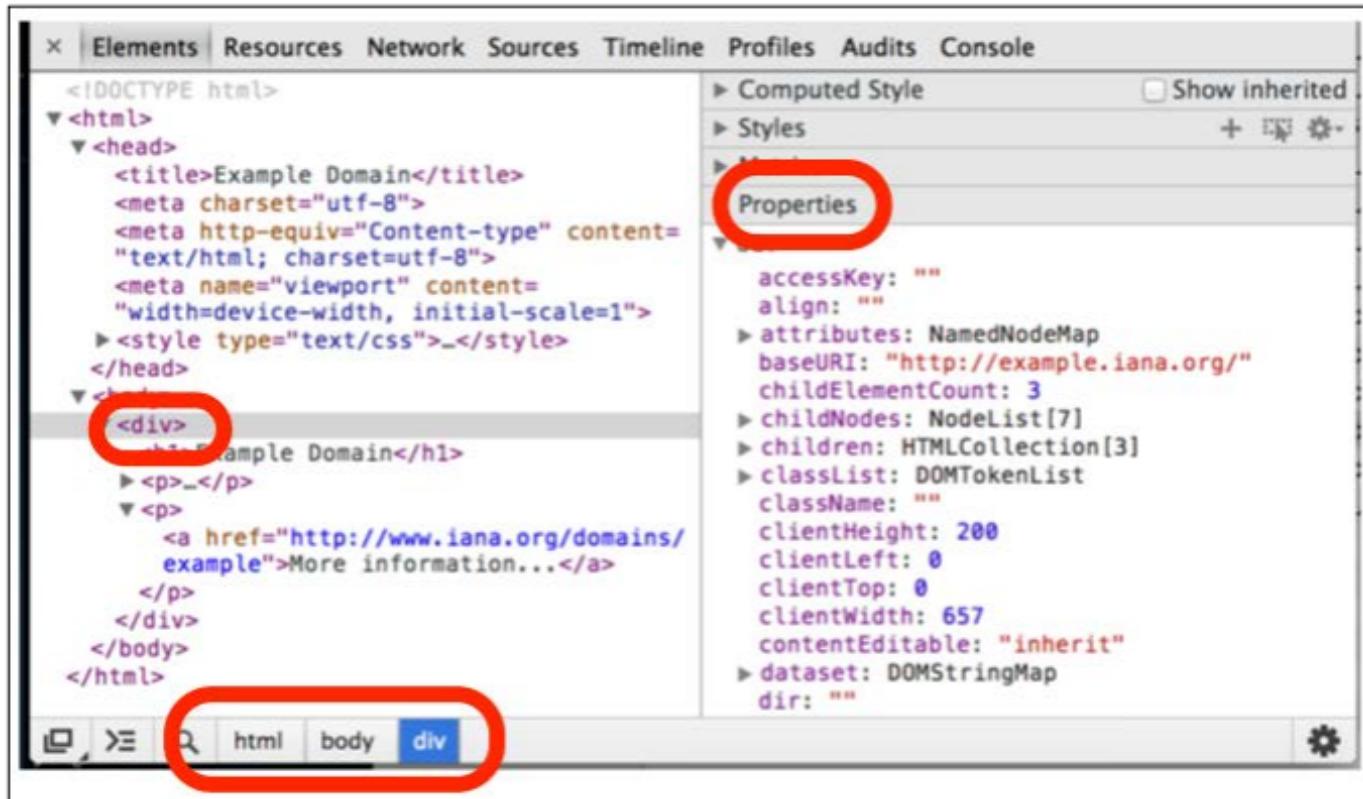
最后，许多标签元素包含有文本，例如标签中的Example Domain。对我们而言，标签之间的可见内容更为重要。头部标签中指明了编码字符，由Scrapy对其处理，就不用我们浪费精力了。

## 树结构

不同的浏览器有不同的借以呈现网页的内部数据结构。但DOM树是跨平台且不依赖语言的，可以被几乎所有浏览器支持。

只需右键点击，选择查看元素，就可以在浏览器中查看网页的树结构。如果这项功能被禁止了，可以在选项的开发者工具中修改。

你看到的树结构和HTML很像，但不完全相同。无论原始HTML文件使用了多少空格和换行符，树结构看起来都会是一样的。你可以点击任意元素，或是改变属性，这样可以实时看到对HTML网页产生了什么变化。例如，如果你双击了一段文字，并修改了它，然后点击回车，屏幕上这段文字就会根据新的设置发生改变。在右边的方框中，在属性标签下面，你可以看到这个树结构的属性列表。在页面底部，你可以看到一个面包屑路径，指示着选中元素的所在位置。



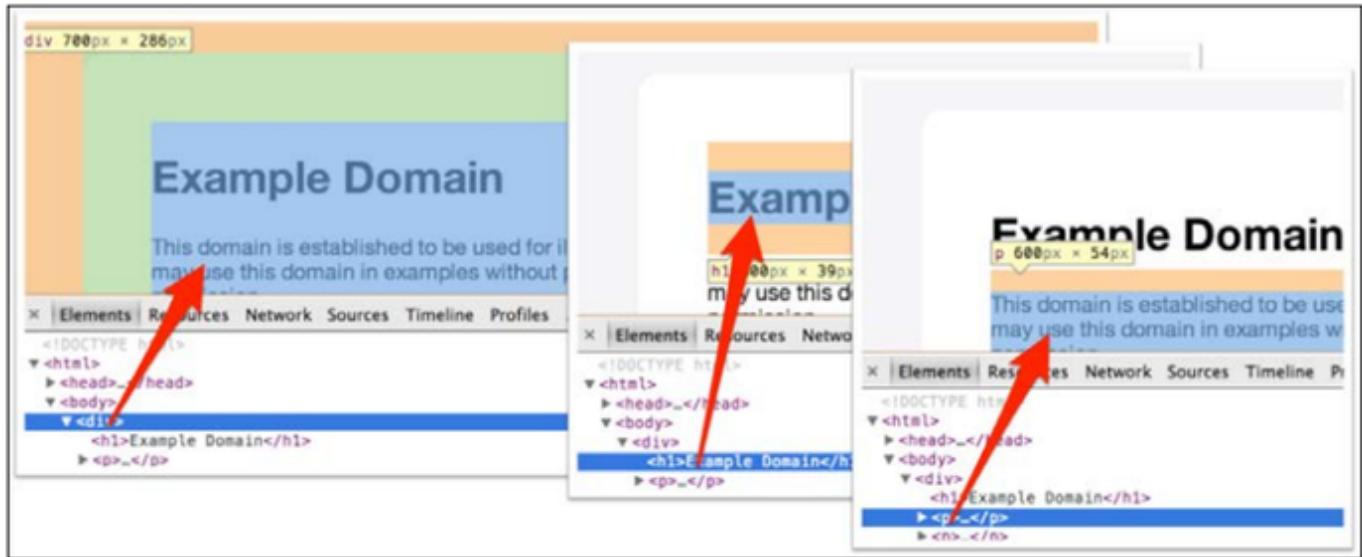
重要的是记住，HTML是文本，而树结构是浏览器内存中的一个对象，你可以通过程序查看、操作这个对象。在Chrome浏览器中，就是通过开发者工具查看。

## 浏览器中的页面

HTML文本和树结构和我们平时在浏览器中看到的页面截然不同。这恰恰是HTML的成功之处。HTML文件就是要具有可读性，可以区分网页的内容，但不是按照呈现在屏幕上的方式。这意味着，呈现HTML文档、进行美化都是浏览器的职责，无论是对于功能齐备的Chrome、移动端浏览器、还是Lynx这样的文本浏览器。

也就是说，网页的发展对网页开发者和用户都提出了极大的开发网页方面的需求。CSS就是这样被发明出来，用以服务HTML元素。对于Scrapy，我们不涉及CSS。

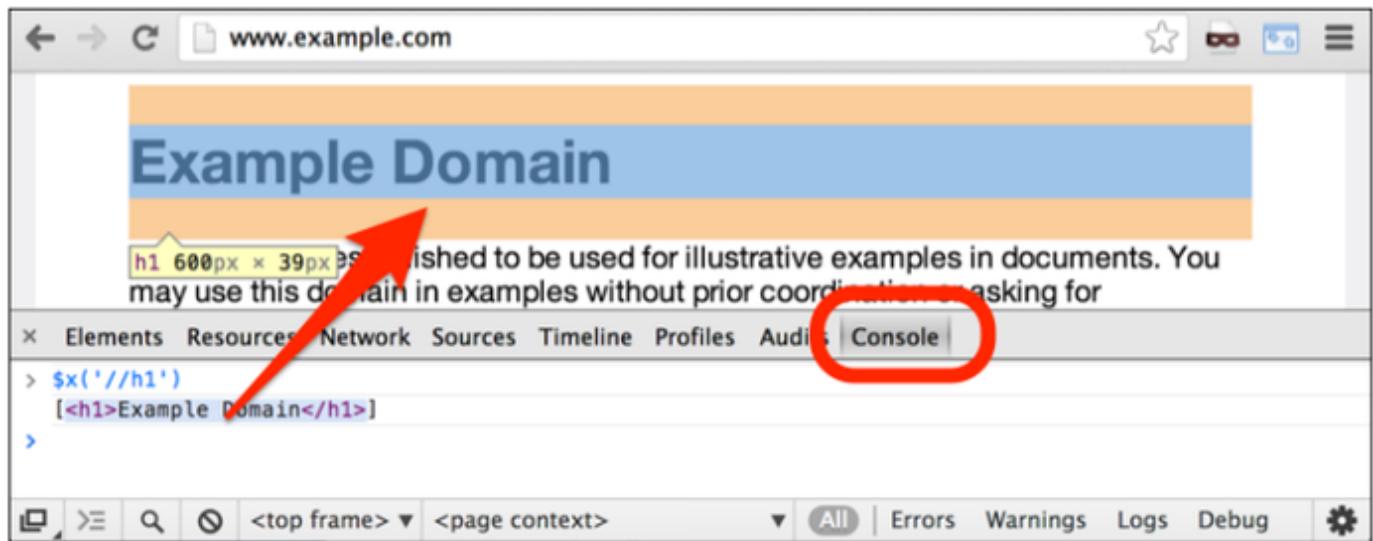
既然如此，树结构对呈现出来的网页有什么作用呢？答案就是盒模型。正如DOM树可以包含其它元素或是文字，同样的，盒模型里面也可以内嵌其它内容。所以，我们在屏幕上看到的网页是原始HTML的二维呈现。树结构是其中的一维，但它是隐藏的。例如，在下图中，我们看到三个DOM元素，一个和两个内嵌的和，出现在浏览器和DOM中：



## 用XPath选择HTML元素

如果你以前接触过传统的软件工程，并不知道XPath，你可能会担心，在HTML文档中查询某个信息，要进行复杂的字符串匹配、搜索标签、处理特殊字符、解析整个树结构等繁琐工作。对于XPath，所有的这些都不是问题，你可以轻松提取元素、属性或是文字。

在Chrome中使用XPath，在开发者工具中点击控制台标签，使用`x`功能。例如，在网页`[http://example.com/](https://link.jianshu.com?t=http://x('//h1'))`，就可以移动到元素，如截图所示：



你在控制台中看到的是一个包含所选元素的JavaScript数组。如果你将光标移动到这个数组上，你可以看到被选择的元素被高亮显示。这个功能很有用。

## XPath表达式

HTML文档的层级结构的最高级是标签，你可以使用元素名和斜杠线选择任意元素。例如，下面的表达式返回了`http://example.com/`上对应的内容：

```
$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> ]
```

注意，标签在标签内有两个，所以会返回两个。你可以用p[1]和p[2]分别返回两个元素。

从抓取的角度，文档的标题或许是唯一让人感兴趣的，它位于文档的头部，可以用下面的额表达式找到：

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

对于大文档，你可能要写很长的XPath表达式，以获取所要的内容。为了避免这点，两个斜杠线//可以让你访问到所有的同名元素。例如，//p可以选择所有的p元素，//a可以选择所有的链接。

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

//a可以用在更多的地方。例如，如果要找到所有标签的链接，你可以使用//div//a。如果a前面只有一个斜杠，//div/a会返回空，因为在上面的例子中标签下面没有。

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

你也可以选择属性。<http://example.com/>上唯一的属性是链接href，可以通过下面的方式找到：

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

你也可以只通过text()函数选择文字：

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

可以使用\*标志选择某层下所有的元素，例如：

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

寻找特定属性，例如@class、或属性有特定值时，你会发现XPath非常好用。例如，//a[@href]可以找到所有链接，//a[@href="http://www.iana.org/domains/example"]则进行了指定的选择。

当属性值中包含特定字符串时，XPath会极为方便。例如，

```
$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>]
```

在[http://www.w3schools.com/xsl/xsl\\_functions.asp](http://www.w3schools.com/xsl/xsl_functions.asp)在线文档中你可以找到更多类似的函数，但并非都常用。

在Scrapy终端中可以使用同样的命令，在命令行中输入

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

终端会向你展示许多写爬虫时碰到的变量。其中最重要的是响应，在HTML中是`HtmlResponse`，这个类可以让你在Chrome使用`xpath()`方法\$`x`。下面是一些例子：

```
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>']
```

这意味着，你可用Chrome浏览器生成XPath表达式，以便在Scrapy爬虫中使用。

## 使用Chrome浏览器获得XPath表达式

Chrome浏览器可以帮助我们获取XPath表达式这点确实对开发者非常友好。像之前演示的那样检查一个元素：右键选择一个元素，选择检查元素。开发者工具被打开，该元素在HTML的树结构中被高亮显示，可以在右键打开的菜单中选择Copy XPath，表达式就复制到粘贴板中了。



你可以在控制台中检测表达式：

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

## 常见工作

下面展示一些XPath表达式的常见使用。先来看看在维基百科上是怎么使用的。维基百科的页面非常稳定，不会在短时间内改变排版。

- 取得id为firstHeading的div下的span的text：

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

- 取得id为toc的div下的ul内的URL：

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

- 在任意class包含ltr和class包含skin-vector的元素之内，取得h1的text，这两个字符串可能在同一class内，或不在。

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

实际应用中，你会在XPath中频繁地使用class。在这几个例子中，你需要记住，因为CSS的板式原因，你会看到HTML的元素总会包含许多特定的class属性。这意味着，有的的class是link，其他导航栏的的class就是link active。后者是当前生效的链接，因此是可见或是用CSS特殊色高亮显示的。当抓取的时候，你通常是对含有某个属性的元素感兴趣的，就像之前的link和link active。XPath的contains( )函数就可以帮你选择包含某一class的所有元素。

- 选择class属性是infobox的table的第一张图片的URL：

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

- 选择class属性是reflist开头的div下面的所有URL链接：

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

- 选择div下面的所有URL链接，并且这个div的下一个相邻元素的子元素包含文字 References：

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

- 取得所有图片的URL：

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

## 提前应对网页发生改变

爬取的目标常常位于远程服务器。这意味着，如果它的HTML发生了改变，XPath表达式就无效了，我们就不得不回过头修改爬虫的程序。因为网页的改变一般就很少，爬虫的改动往往不会很大。然而，我们还是宁肯不要回头修改。一些基本原则可以帮助我们降低表达式失效的概率：

- 避免使用数组序号 Chrome常常会在表达式中加入许多常数

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

如果HTML上有一个广告窗的话，就会改变文档的结构，这个表达式就会失效。解决的方法是，尽量找到离img标签近的元素，根据该元素的id或class属性，进行抓取，例如：

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

- 用class抓取效果不一定好 使用class属性可以方便的定位要抓取的元素，但是因为CSS也要通过class修改页面的外观，所以class属性可能会发生改变，例如下面用到的class：

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

过一段时间之后，可能会变成：

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

- 数据指向的class优于排版指向的class

在上一个例子中，使用thumbnail和green两个class都不好。thumbnail比green好，但这两个都不如departure-time。前面两个是用来排版的，departure-time是有语义的，和div中的内容有关。所以，在排版发生改变的情况下，departure-time发生改变的可能性会比较小。应该说，网站作者在开发中十分清楚，为内容设置有意义的、一致的标记，可以让开发过程收益。

- id通常是最可靠的

只要id具有语义并且数据相关，id通常是抓取时最好的选择。部分原因是，JavaScript和外链锚点总是使用id获取文档中特定的部分。例如，下面的XPath非常可靠：

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

相反的例子是，指向唯一参考的id，对抓取没什么帮助，因为抓取总是希望能够获取具有某个特点的所有信息。例如：

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

这是一个非常差的XPath表达式。还要记住，尽管id最好要有某种特点，但在许多HTML文档中，id都很杂乱无章。

## 总结

编程语言的不断进化，使得创建可靠的XPath表达式从HTML抓取信息变得越来越容易。在本章中，你学到了HTML和XPath的基本知识、如何利用Chrome自动获取XPath表达式。你还学会了如何手工写XPath表达式，并区分可靠和不够可靠的XPath表达式。第3章中，我们会用这些知识来写几个爬虫。

# 第3章 爬虫基础

本章非常重要，你可能需要读几遍，或是从中查找解决问题的方法。我们会从如何安装Scrapy讲起，然后在案例中讲解如何编写爬虫。开始之前，说几个注意事项。

因为我们马上要进入有趣的编程部分，使用本书中的代码段会十分重要。当你看到：

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

是要让你在终端中输入echo hello world（忽略\$），第二行是看到结果。

当你看到：

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

是让你在Python或Scrapy界面进行输入（忽略>>>）。同样的，第二行是输出结果。

你还需要对文件进行编辑。编辑工具取决于你的电脑环境。如果你使用Vagrant（强烈推荐），你可以是用Notepad、Notepad++、Sublime Text、TextMate、Eclipse、或PyCharm等文本编辑器。如果你更熟悉Linux/Unix，你可以用控制台自带的vim或emacs。这两个编辑器功能强大，但是有一定的学习曲线。如果你是初学者，可以选择适合初学者的nano编辑器。

## 安装Scrapy

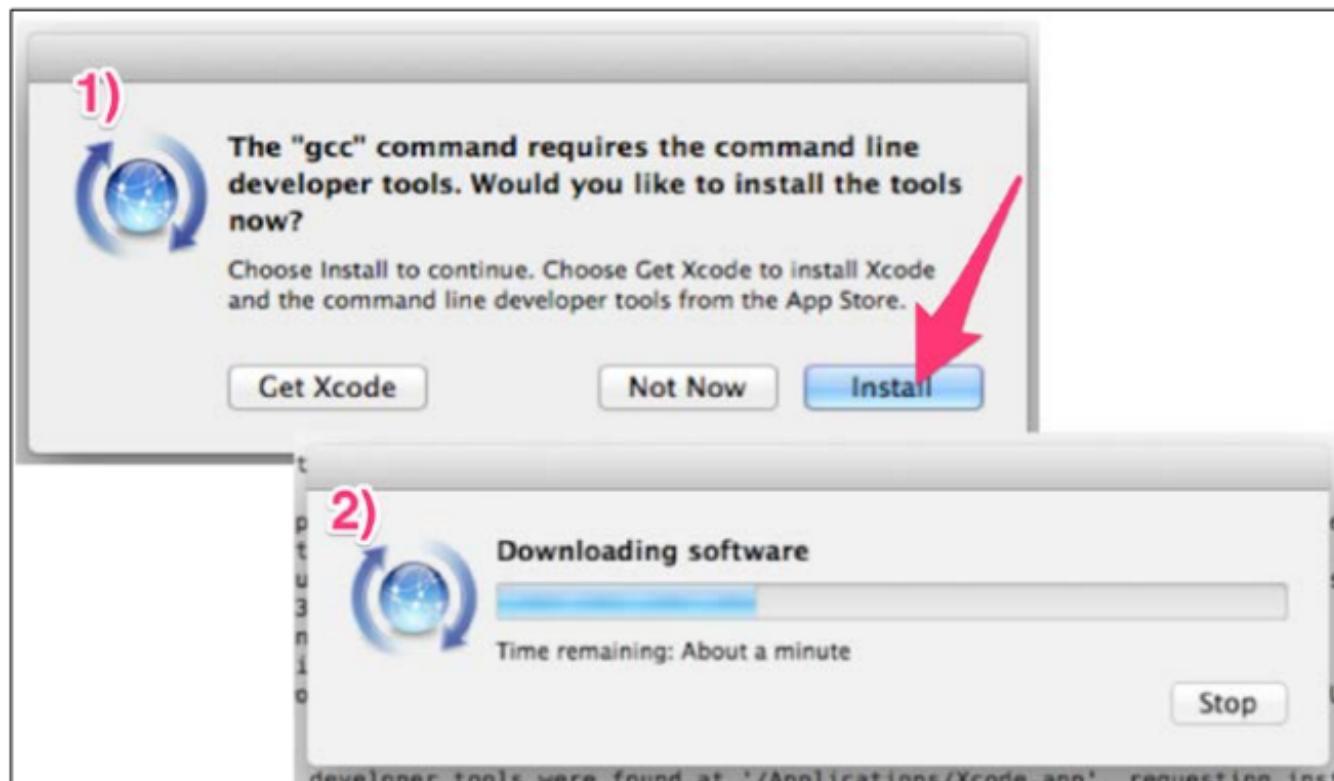
Scrapy的安装相对简单，但这还取决于读者的电脑环境。为了支持更多的人，本书安装和使用Scrapy的方法是用Vagrant，它可以在Linux盒中使用所有的工具，而无关于操作系统。下面提供了Vagrant和一些常见操作系统的指导。

### MacOS

为了轻松跟随本书学习，请参照后面的Vagrant说明。如果你想在MacOS中安装Scrapy，只需控制台中输入：

```
$ easy_install scrapy
```

然后，所有事就可以交给电脑了。安装过程中，可能会向你询问密码或是否安装Xcode，只需同意即可。



## Windows

在Windows中安装Scrapy要麻烦些。另外，在Windows安装本书中所有的软件也很麻烦。我们都为你想到了可能的问题。有Virtualbox的Vagrant可以在所有64位电脑上顺利运行。翻阅相关章节，只需几分钟就可以安装好。如果真要在Windows中安装，请参考本书网站<http://scrapybook.com/>上面的资料。

## Linux

你可能会在多种Linux服务器上安装Scrapy，步骤如下：

提示：确切的安装依赖变化很快。写作本书时，Scrapy的版本是1.0.3（翻译此书时是1.4）。下面只是对不同服务器的建议方法。

### Ubuntu或Debian Linux

为了在Ubuntu（测试机是Ubuntu 14.04 Trusty Tahr - 64 bit）或是其它使用apt的服务器上安装Scrapy，可以使用下面三条命令：

```
$ 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
```

这个方法需要进行编译，可能随时中断，但可以安装PyPI上最新版本的Scrapy。如果想避开编译，安装不是最新版本的话，可以搜索“install Scrapy Ubuntu packages”，按照官方文档安装。

## Red Hat或CentOS Linux

在使用yum的Linux上安装Scrapy也很简单（测试机是Ubuntu 14.04 Trusty Tahr - 64 bit）。只需三条命令：

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

## 从GitHub安装

按照前面的指导，就可以安装好Scrapy的依赖了。Scrapy是纯Python写成的，如果你想编辑源代码或是测试最新版，可以从<https://github.com/scrapy/scrapy>克隆最新版，只需命令行输入：

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

我猜如果你是这类用户，就不需要我提醒安装virtualenv了。

## 升级Scrapy

Scrapy升级相当频繁。如果你需要升级Scrapy，可以使用pip、easy\_install或aptitude：

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

或

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

如果你想降级或安装指定版本的Scrapy，可以：

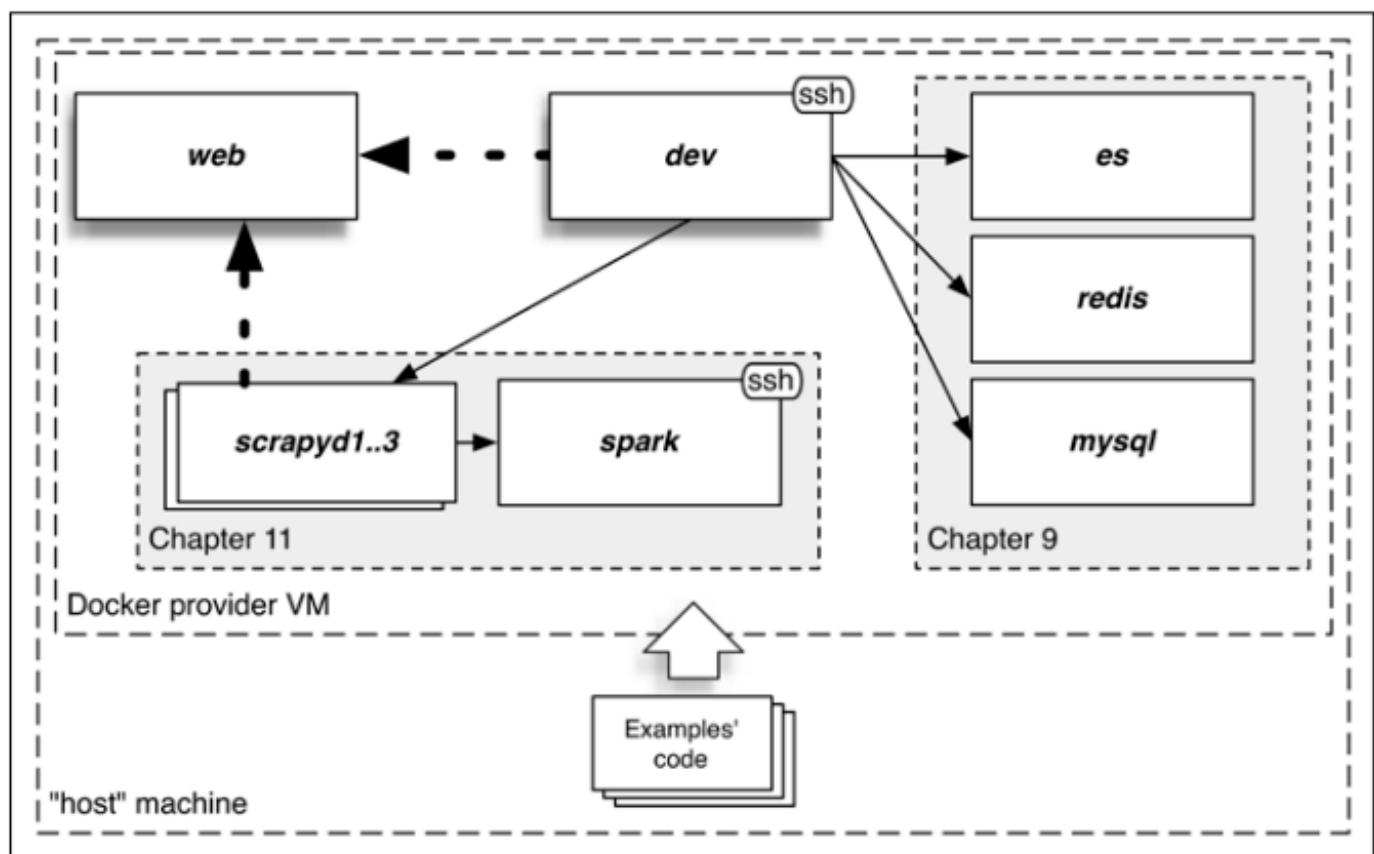
```
$ sudo pip install Scrapy==1.0.0
```

或

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

## Vagrant: 本书案例的运行方法

本书有的例子比较复杂，有的例子使用了许多东西。无论你是什么水平，都可以尝试运行所有例子。只需一句命令，就可以用Vagrant搭建操作环境。



## 本书使用的系统

在Vagrant中，你的电脑被称作“主机”。Vagrant在主机中创建一个虚拟机。这样就可以让我们忽略主机的软硬件，来运行案例了。

本书大多数章节使用了两个服务——开发机和网络机。我们在开发机中登录运行Scrapy，在网络机中进行抓取。后面的章节会使用更多的服务，包括数据库和大数据处理引擎。

根据附录A安装必备，安装Vagrant，直到安装好git和Vagrant。打开命令行，输入以下命令获取本书的代码：

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

打开Vagrant:

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

第一次打开Vagrant会需要些时间，这取决于你的网络。第二次打开就会比较快。打开之后，登录你的虚拟机，通过：

```
$ vagrant ssh
```

代码已经从主机中复制到了开发机，现在可以在book的目录中看到：

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

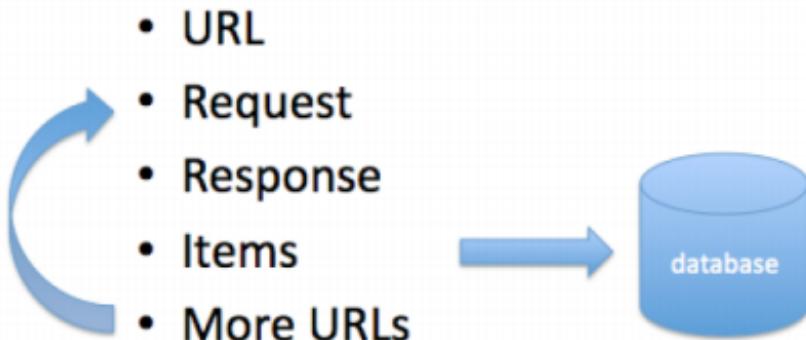
可以打开几个窗口输入vagrant ssh，这样就可以打开几个终端。输入vagrant halt可以关闭系统，vagrant status可以检查状态。vagrant halt不能关闭虚拟机。如果在VirtualBox中碰到问题，可以手动关闭，或是使用vagrant global-status查找id，用vagrant halt 暂停。大多数例子可以离线运行，这是Vagrant的一大优点。

安装好环境之后，就可以开始学习Scrapy了。

## UR2IM——基础抓取过程

每个网站都是不同的，对每个网站进行额外的研究不可避免，碰到特别生僻的问题，也许还要用Scrapy的邮件列表咨询。寻求解答，去哪里找、怎么找，前提是熟悉整个过程和相关术语。Scrapy的基本过程，可以写成字母缩略语UR2IM，见下图。

# The Basic Scraping Equation: UR<sup>2</sup>IM



## The URL

一切都从URL开始。你需要目标网站的URL。我的例子是<https://www.gumtree.com/>, Gumtree分类网站。

例如，访问伦敦房地产首页<http://www.gumtree.com/flats-houses/london>，你就可以找到许多房子的URL。右键复制链接地址，就可以复制URL。其中一个URL可能是这样的：<https://www.gumtree.com/p/studios-bedsits-rent/split-level>。但是，Gumtree的网站变动之后，URL的XPath表达式会失效。不添加用户头的话，Gumtree也不会响应。这个留给以后再说，现在如果你想加载一个网页，你可以使用Scrapy终端，如下所示：

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

要进行调试，可以在Scrapy语句后面添加 -pdb，例如：

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

我们不想让大家如此频繁的点击Gumtree网站，并且Gumtree网站上URL失效很快，不适合做例子。我们还希望大家能在离线的情况下，多多练习书中的例子。这就是为什么Vagrant开发环境内嵌了一个网络服务器，可以生成和Gumtree类似的网页。这些网页可能并不好看，但是从爬虫开发者的角度，是完全合格的。如果想在Vagrant上访问Gumtree，可以在Vagrant开发机上访问<http://web:9312/>，或是在浏览器中访问<http://localhost:9312/>。

让我们在这个网页上尝试一下Scrapy，在Vagrant开发机上输入：

```
$ 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
>>>
```

得到一些输出，加载页面之后，就进入了Python（可以使用Ctrl+D退出）。

## 请求和响应

在前面的输出日志中，Scrapy自动为我们做了一些工作。我们输入了一条地址，Scrapy做了一个GET请求，并得到一个成功响应值200。这说明网页信息已经成功加载，并可以使用了。如果要打印reponse.body的前50个字母，我们可以得到：

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

这就是这个Gumtree网页的HTML文档。有时请求和响应会很复杂，第5章会对其进行讲解，现在只讲最简单的情况。

## 抓取对象

下一步是从响应文件中提取信息，输入到Item。因为这是个HTML文档，我们用XPath来做。首先来看一下这个网页：

The screenshot shows the developer tools in Chrome's inspect element mode. The DOM tree on the left highlights the 'title' element, which corresponds to the page title 'Split level studio with a balcony in Earls Court, Notting Hill, London'. The right panel shows various actions like 'Copy', 'Search Google.com for "UTILITY"', and 'Inspect Element'. A red arrow points from the 'title' element in the tree to the page title. Another red arrow points from the 'telephone' element in the tree to its value '020 7229 9122' on the page.

页面上的信息很多，但大多是关于版面的：logo、搜索框、按钮等等。从抓取的角度，它们不重要。我们关注的是，例如，列表的标题、地址、电话。它们都对应着HTML里的元素，我们要在HTML中定位，用上一章所学的提取出来。先从标题开始。

The screenshot shows the developer tools with the context menu open over an 

# element. The menu has options like 'Add Attribute', 'Edit Attribute', 'Force Element State', 'Edit as HTML', 'Copy as HTML', 'Copy CSS Path', 'Copy XPath', 'Delete Node', 'Break on...', and 'Create Listener...'. A red arrow points to the 'Copy XPath' option. Another red arrow points to the simplified XPath '/h1' shown in the bottom right, with the text '3. Simplify' written above it. The page content includes a house image, a price of £350.00pw, and a contact section.

在标题上右键点击，选择检查元素。在自动定位的HTML上再次右键点击，选择复制XPath。Chrome给的XPath总是很复杂，并且容易失效。我们要对其进行简化。我们只取最后面的h1。这是因为从SEO的角度，每页HTML只有一个h1最好，事实上大多是网页只有一个h1，所以不用担

心重复。

提示：SEO是搜索引擎优化的意思：通过对网页代码、内容、链接的优化，提升对搜索引擎的支持。

让我们看看h1标签行不行：

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

很好，完全行得通。我在h1后面加上了text()，表示只提取h1标签里的文字。没有添加text()的话，就会这样：

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

我们已经成功得到了title，但是再仔细看看，还能发现更简便的方法。

► <h1 **itemprop="name"** **class="space-mbs"**>...</h1>

Gumtree为标签添加了属性，就是itemprop=name。所以XPath可以简化为//**\***[@itemprop="name"]**[1]**/text()。在XPath中，切记数组是从1开始的，所以这里[]里面是1。

选择itemprop="name"这个属性，是因为Gumtree用这个属性命名了许多其他的内容，比如“You may also like”，用数组序号提取会很方便。

接下来看价格。价格在HTML中的位置如下：

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

我们又看到了itemprop="name"这个属性，XPath表达式为//**\***[@itemprop="price"]**[1]**/text()。验证一下：

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

注意Unicode字符（£符号）和价格350.00pw。这说明要对数据进行清理。在这个例子中，我们用正则表达式提取数字和小数点。使用正则方法如下：

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

提取房屋描述的文字、房屋的地址也很类似，如下：

```
//*[@itemprop="description"])[1]/text()  
//*[@itemtype="http://schema.org/Place"])[1]/text()
```

相似的，抓取图片可以用//img[@itemprop="image"][@src]。注意这里没使用text()，因为我们只想要图片的URL。

假如这就是我们要提取的所有信息，整理如下：

目标	XPath表达式
title	//*[@itemprop="name"])[1]/text() Example value: [u'set unique family well']
Price	//*[@itemprop="price"])[1]/text() Example value (using re()):[u'334.39']
description	//*[@itemprop="description"])[1]/text() Example value: [u'website court wareho use\r\npool...']
Address	//*[@itemtype="http://schema.org/Place"])[1]/text() Example value: [u'Angel, Lo ndon']
Image_U RL	//*[@itemprop="image"])[1]/@src Example value: [u'..//images/i01.jpg']

这张表很重要，因为也许只要稍加改变表达式，就可以抓取其他页面。另外，如果要爬取数十个网站时，使用这样的表可以进行区分。

目前为止，使用的还只是HTML和XPath，接下来用Python来做一个项目。

## 一个Scrapy项目

目前为止，我们只是在Scrapy shell中进行操作。学过前面的知识，现在开始一个Scrapy项目，Ctrl+D退出Scrapy shell。Scrapy shell只是操作网页、XPath表达式和Scrapy对象的工具，不要在上面浪费太多，因为只要一退出，写过的代码就会消失。我们创建一个名字是properties的项

目：

```
$ scrapy startproject properties
$ cd properties
$ tree
.
├── properties
│   ├── __init__.py
│   ├── items.py
│   ├── pipelines.py
│   ├── settings.py
│   └── spiders
│       └── __init__.py
└── scrapy.cfg
2 directories, 6 files
```

先看看这个Scrapy项目的文件目录。文件夹内包含一个同名的文件夹，里面有三个文件items.py, pipelines.py, 和settings.py。还有一个子文件夹spiders，里面现在是空的。后面的章节会详谈settings、pipelines和scrapy.cfg文件。

## 定义items

用编辑器打开items.py。里面已经有代码，我们要对其修改下。用之前的表里的内容重新定义class PropertiesItem。

还要添加些后面会用到的内容。后面会深入讲解。这里要注意的是，声明一个字段，并不要求一定要填充。所以放心添加你认为需要的字段，后面还可以修改。

字段	Python表达式
images	pipeline根据image_URL会自动填充这里。后面详解。
Location	地理编码会填充这里。后面详解。

我们还会加入一些杂务字段，也许和现在的项目关系不大，但是我个人很感兴趣，以后或许能用到。你可以选择添加或不添加。观察一下这些项目，你就会明白，这些项目是怎么帮助我找到何地 (server, url)，何时 (date)，还有 (爬虫) 如何进行抓取的。它们可以帮助我取消项目，制定新的重复抓取，或忽略爬虫的错误。这里看不明白不要紧，后面会细讲。

杂务字段	Python表达式
url	response.url Example value: ' <a href="http://web.../property_000000.html">http://web.../property_000000.html</a> '
project	self.settings.get('BOT_NAME') Example value: 'properties'

spider	self.name Example value: 'basic'
server	server socket.gethostname() Example value: 'scrapyserver1'
date	datetime.datetime.now() Example value: datetime.datetime(2015, 6, 25...)

利用这个表修改PropertiesItem这个类。修改文件properties/items.py如下：

```
from scrapy.item import Item, Field

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

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

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

这是我们的第一段代码，要注意Python中是使用空格缩进的。每个字段名之前都有四个空格或是一个tab。如果一行有四个空格，另一行有三个空格，就会报语法错误。如果一行是四个空格，另一行是一个tab，也会报错。空格符指定了这些项目是在PropertiesItem下面的。其他语言有的用花括号{}，有的用begin – end，Python则使用空格。

## 编写爬虫

已经完成了一半。现在来写爬虫。一般的，每个网站，或一个大网站的一部分，只有一个爬虫。爬虫代码来成UR2IM流程。

当然，你可以用文本编辑器一句一句写爬虫，但更便捷的方法是用scrapy genspider命令，如下所示：

```
$ scrapy genspider basic web
```

使用模块中的模板“basic”创建了一个爬虫“basic”：

```
properties.spiders.basic
```

一个爬虫文件basic.py就出现在目录properties/spiders中。刚才的命令是，生成一个名字是basic的默认文件，它的限制是在web上爬取URL。我们可以取消这个限制。这个爬虫使用的是basic这个模板。你可以用scrapy genspider -l查看所有的模板，然后用参数-t利用模板生成想要的爬虫，后面会介绍一个例子。

查看properties/spiders/basic.py file文件，它的代码如下：

```
import scrapy
class BasicSpider(scrapy.Spider):
    name = "basic"
    allowed_domains = ["web"]
start_URL = (
    'http://www.web/',
)
def parse(self, response):
    pass
```

import命令可以让我们使用Scrapy框架。然后定义了一个类BasicSpider，继承自scrapy.Spider。继承的意思是，虽然我们没写任何代码，这个类已经继承了Scrapy框架中的类Spider的许多特性。这允许我们只需写几行代码，就可以有一个功能完整的爬虫。然后我们看到了一些爬虫的参数，比如名字和抓取域字段名。最后，我们定义了一个空函数parse()，它有两个参数self和response。通过self，可以使用爬虫一些有趣的功能。response看起来很熟悉，它就是我们在Scrapy shell中见到的响应。

下面来开始编辑这个爬虫。start\_URL更改为在Scrapy命令行中使用过的URL。然后用爬虫事先准备的log()方法输出内容。修改后的properties/spiders/basic.py文件为：

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

```
'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_URL: %s" % response.xpath(
        '//*[@itemprop="image"]')[1]/@src).extract()
```

总算到了运行爬虫的时间！让爬虫运行的命令是scrapy crawl接上爬虫的名字：

```
$ 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_URL: [u'../images/i01.jpg']
INFO: Closing spider (finished)
...
```

成功了！不要被这么多行的命令吓到，后面我们再仔细说明。现在，我们可以看到使用这个简单的爬虫，所有的数据都用XPath得到了。

来看另一个命令，scrapy parse。它可以让我们选择最合适的爬虫来解析URL。用–spider命令可以设定爬虫：

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

你可以看到输出的结果和前面的很像，但却是关于另一个房产的。

填充一个项目

接下来稍稍修改一下前面的代码。你会看到，尽管改动很小，却可以解锁许多新的功能。

首先，引入类PropertiesItem。它位于properties目录中的item.py文件，因此在模块properties.items中。它的导入命令是：

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

然后我们要实例化，并进行返回。这很简单。在parse()方法中，我们加入声明item = PropertiesItem()，它产生了一个新项目，然后为它分配表达式：

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

最后，我们用return item返回项目。更新后的properties/spiders/basic.py文件如下：

```
import scrapy
from properties.items import PropertiesItem
class BasicSpider(scrapy.Spider):
    name = "basic"
    allowed_domains = ["web"]
    start_URL = (
        'http://web:9312/properties/property_00000.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_URL'] = response.xpath(
            '//*[@itemprop="image"]')[1]/@src).extract()
        return item
```

现在如果再次运行爬虫，你会注意到一个不大但很重要的改动。被抓取的值不再打印出来，没有“DEBUG：被抓取的值”了。你会看到：

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

这是从这个页面抓取的PropertiesItem。这很好，因为Scrapy就是围绕Items的概念构建的，这意味着我们可以用pipelines填充丰富项目，或是用“Feed export”导出保存到不同的格式和位置。

## 保存到文件

试运行下面：

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

不用我们写任何代码，我们就可以用这些格式进行存储。Scrapy可以自动识别输出文件的后缀名，并进行输出。这段代码中涵盖了一些常用的格式。CSV和XML文件很流行，因为可以被Excel直接打开。JSON文件很流行是因为它的开放性和与JavaScript的密切关系。JSON和JSON Line格式的区别是.json文件是在一个大数组中存储JSON对象。这意味着如果你有一个1GB的文件，你可能必须现在内存中存储，然后才能传给解析器。相对的，.jl文件每行都有一个JSON对象，所以读取效率更高。

不在文件系统中存储生成的文件也很麻烦。利用下面例子的代码，你可以让Scrapy自动上传文件到FTP或亚马逊的S3 bucket。

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

注意，证书和URL必须按照主机和S3更新，才能顺利运行。

另一个要注意的是，如果你现在使用scrapy parse，它会向你显示被抓取的项目和抓取中新的请求：

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

当出现意外结果时，scrapy parse可以帮你进行debug，你会更感叹它的强大。

## 清洗——项目加载器和杂务字段

恭喜你，你已经创建成功一个简单爬虫了！让我们让它看起来更专业些。

我们使用一个功能类，ItemLoader，以取代看起来杂乱的extract()和xpath()。我们的parse()进行如下变化：

```
def parse(self, response):
    l = ItemLoader(item=PropertiesItem(), response=response)
    l.add_xpath('title', '//*[@itemprop="name"]る/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"]る/text()')
    l.add_xpath('image_URL', '//*[@itemprop="image"]る/@src')
    return l.load_item()
```

是不是看起来好多了？事实上，它可不是看起来漂亮那么简单。它指出了我们现在要干什么，并且后面的加载项很清晰。这提高了代码的可维护性和自文档化。（自文档化，self-documenting，是说代码的可读性高，可以像文档文件一样阅读）

ItemLoaders提供了许多有趣的方式整合数据、格式化数据、清理数据。它的更新很快，查阅文档可以更好的使用它，<http://doc.scrapy.org/en/latest/topics/loaders>。通过不同的类处理器，ItemLoaders从XPath/CSS表达式传参。处理器函数快速小巧。举一个Join()的例子。//p表达式会选取所有段落，这个处理函数可以在一个入口中将所有内容整合起来。另一个函数MapCompose()，可以与Python函数或Python函数链结合，实现复杂的功能。例如，MapCompose(float)可以将字符串转化为数字，MapCompose(unicode.strip, unicode.title)可以去除多余的空格，并将单词首字母大写。让我们看几个处理函数的例子：

处理函数	功能
Join()	合并多个结果。
MapCompose(unicode.strip)	除去空格。
MapCompose(unicode.strip, unicode.title)	除去空格，单词首字母大写。
MapCompose(float)	将字符串转化为数字。
MapCompose(lambda i: i.replace(',', ''), float)	将字符串转化为数字，逗号替换为空格。
MapCompose(lambda i: urlparse.urljoin(response.url, i))	使用response.url为开头，将相对URL转化为绝对URL。

你可以使用Python编写处理函数，或是将它们串联起来。unicode.strip()和unicode.title()分别用单一参数实现了单一功能。其它函数，如replace()和urljoin()需要多个参数，我们可以使用Lambda函数。这是一个匿名函数，可以不声明函数就调用参数：

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

可以取代下面的函数：

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

使用Lambda函数，打包replace()和urljoin()，生成一个结果，只需一个参数即可。为了更清楚前面的表，来看几个实例。在scrapy命令行打开任何URL，并尝试：

```
>>> 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 c0De'])
[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']
```

要记住，处理函数是对XPath/CSS结果进行后处理的的小巧函数。让我们来看几个我们爬虫中的处理函数是如何清洗结果的：

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

完整的列表在本章后面给出。如果你用scrapy crawl basic再运行的话，你可以得到干净的结果如下：

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

最后，我们可以用add\_value()方法添加用Python（不用XPath/CSS表达式）计算得到的值。我们用它设置我们的“杂务字段”，例如URL、爬虫名、时间戳等等。我们直接使用前面杂务字段表里总结的表达式，如下：

```
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())
```

记得import datetime和socket，以使用这些功能。

现在，我们的Items看起来就完美了。我知道你的第一感觉是，这可能太复杂了，值得吗？回答是肯定的，这是因为或多或少，想抓取网页信息并存到items里，这就是你要知道的全部。这段代码如果用其他语言来写，会非常难看，很快就不能维护了。用Scrapy，只要25行简洁的代码，它明确指明了意图，你可以看清每行的意义，可以清晰的进行修改、再利用和维护。

你的另一个感觉可能是处理函数和ItemLoaders太花费精力。如果你是一名经验丰富的Python开发者，你已经会使用字符串操作、lambda表达构造列表，再学习新的知识会觉得不舒服。然而，这只是对ItemLoader和其功能的简单介绍，如果你再深入学习一点，你就不会这么想了。ItemLoaders和处理函数是专为有抓取需求的爬虫编写者、维护者开发的工具集。如果你想深入学习爬虫的话，它们是绝对值得学习的。

## 创建协议

协议有点像爬虫的单元测试。它们能让你快速知道错误。例如，假设你几周以前写了一个抓取器，它包含几个爬虫。你想快速检测今天是否还是正确的。协议位于评论中，就在函数名后面，协议的开头是@。看下面这个协议：

```
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_URL  
    @scrapes url project spider server date  
    """
```

这段代码是说，检查这个URL，你可以在找到一个项目，它在那些字段有值。现在如果你运行scrapy check，它会检查协议是否被满足：

```
$ scrapy check basic
-----
Ran 3 contracts in 1.640s
OK
如果url的字段是空的（被注释掉），你会得到一个描述性错误：
FAIL: [basic] parse (@scrapes post-hook)

ContractFail: 'url' field is missing
```

当爬虫代码有错，或是XPath表达式过期，协议就可能失效。当然，协议不会特别详细，但是可以清楚的指出代码的错误所在。

综上所述，我们的第一个爬虫如下所示：

```
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_URL = (
        '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_URL
        @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_URL', '//*[@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()

```

## 提取更多的URL

到目前为止，在爬虫的start\_URL中我们还是只加入了一条URL。因为这是一个元组，我们可以向里面加入多个URL，例如：

```

start_URL = (
    'http://web:9312/properties/property_00000.html',
    'http://web:9312/properties/property_00001.html',
    'http://web:9312/properties/property_00002.html',
)

```

不够好。我们可以用一个文件当做URL源文件：

```

start_URL = [i.strip() for i in
open('todo.URL.txt').readlines()]

```

还是不够好，但行得通。更常见的，网站可能既有索引页也有列表页。例如，Gumtree有索引页：<http://www.gumtree.com/flats-houses/london>：

**Listing link**

**next page**

93,495 ads in Property, London

Refine

Keyword: I'm looking for...

Search title & description

Search only:

- Urgent ads
- Feature ads
- Ads with pictures

Clear

Apply

Categories: All Categories

Property

Westminster, London

Earls Court, London

Earls Court, London

E330pw Just now

E350pw Just now

E295pw Just now

1 2 3 4 5 6 ... 4,209 Next >

一个典型的索引页包含许多列表页、一个分页系统，让你可以跳转到其它页面。

**(vertical crawling)**

**(horizontal crawling)**

**Listing pages**

Items

Index pages

93,495 ads in Property, London

Refine

Keyword: I'm looking for...

Search title & description

Search only:

- Urgent ads
- Feature ads
- Ads with pictures

Clear

Apply

Categories: All Categories

All Categories

All Categories

Westminster, London

Earls Court, London

Earls Court, London

E330pw Just now

E350pw Just now

E295pw Just now

1 2 3 4 5 6 ... 4,209 Next >

London

Name / Studio Apartments & Bedsit to Rent

1.00pw

Flat Price frequency per

Rental amount £350.00 Seller type Agency

Date available 13 Sep 2014 Posted Just now

Number of beds 0

Date available 11 Sep 2014 Posted Just now

Number of beds 0

Date available 11 Sep 2014 Posted Just now

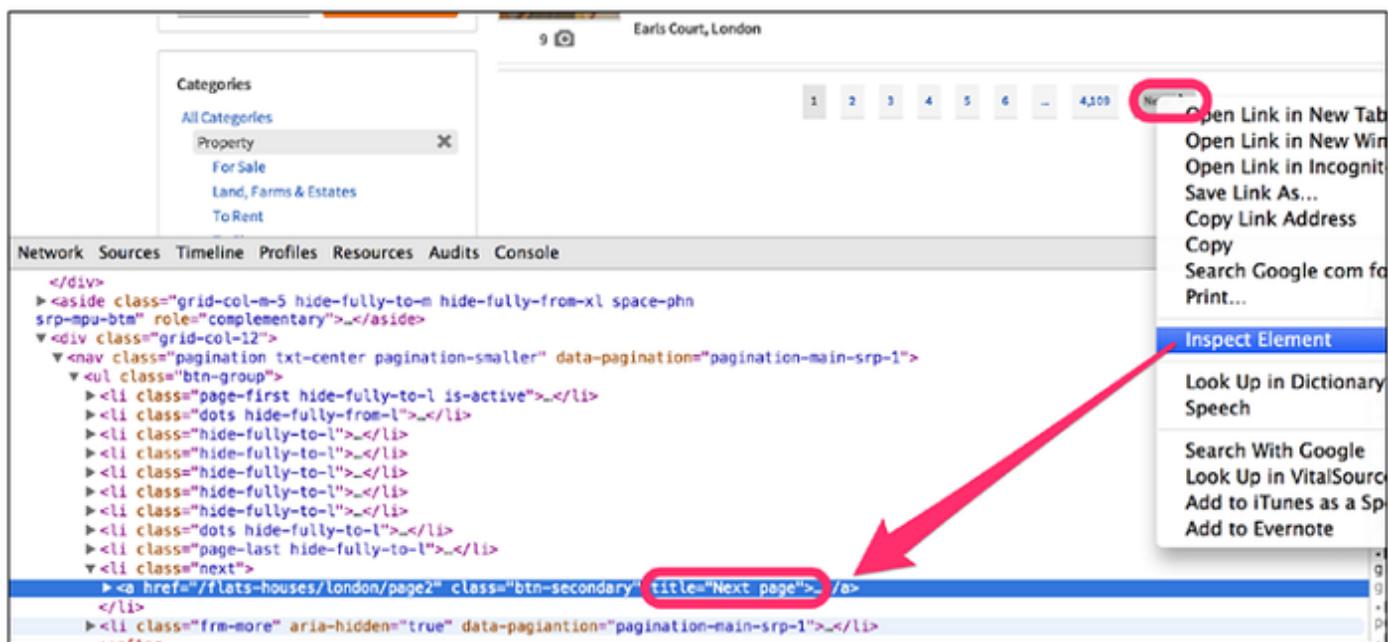
Number of beds 0

因此，一个典型的爬虫在两个方向移动：

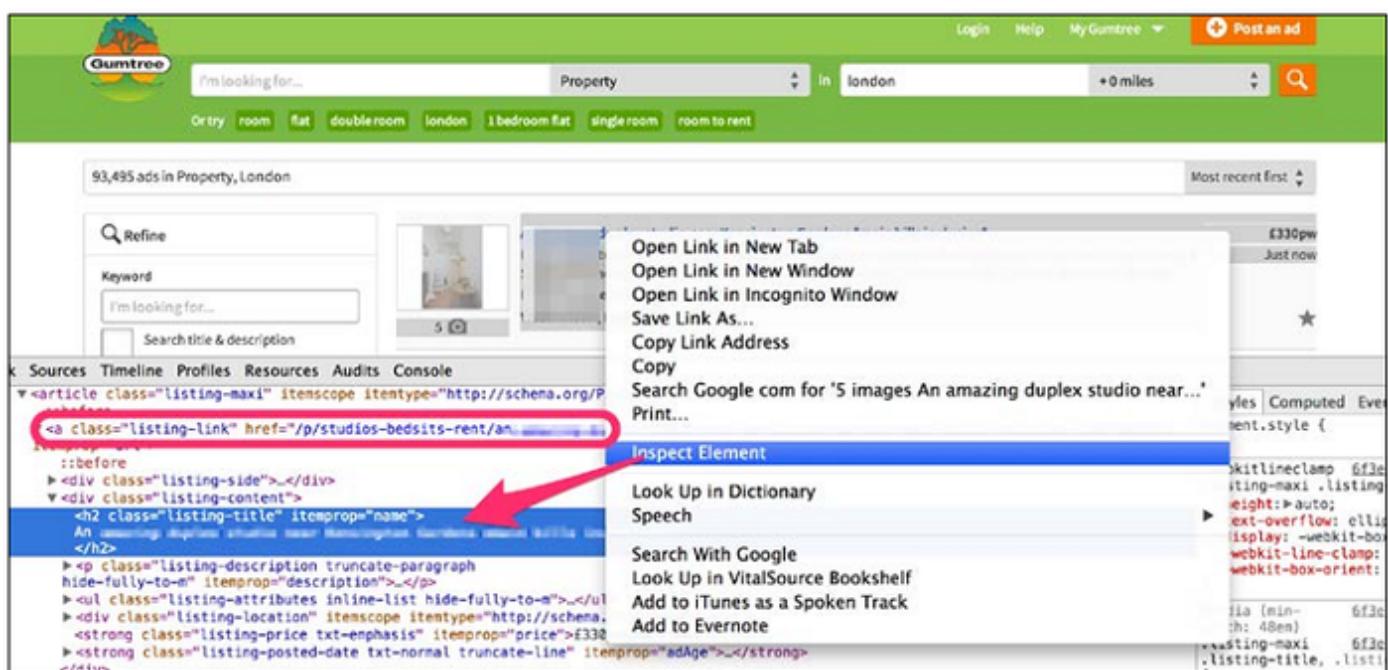
- **水平**——从索引页到另一个索引页
  - **垂直**——从索引页面到列表页面提取项目

在本书中，我们称前者为水平抓取，因为它在同一层次（例如索引）上抓取页面；后者为垂直抓取，因为它从更高层次（例如索引）移动到一个较低的层次（例如列表）。

做起来要容易许多。我们只需要两个XPath表达式。第一个，我们右键点击Next page按钮，URL位于li中，li的类名含有next。因此XPath表达式为`//*[@contains(@class, "next")]/@href`。



对于第二个表达式，我们在列表的标题上右键点击，选择检查元素：



这个URL有一个属性是itemprop="url"。因此，表达式确定为//\*[@itemprop="url"]/@href。打开scrapy命令行进行确认：

```
$ scrapy shell http://web:9312/properties/index_00000.html
>>> URL = response.xpath('//*[contains(@class, "next")]/@href').extract()
>>> URL
[u'index_00001.html']
>>> import urlparse
>>> [urlparse.urljoin(response.url, i) for i in URL]
[u'http://web:9312/scrapybook/properties/index_00001.html']
>>> URL = response.xpath('//*[@itemprop="url"]/@href').extract()
>>> URL
[u'property_00000.html', ... u'property_00029.html']
>>> len(URL)
30
>>> [urlparse.urljoin(response.url, i) for i in URL]
[u'http://..._00000.html', ... /property_00029.html']
```

很好，我们看到有了这两个表达式，就可以进行水平和垂直抓取URL了。

## 使用爬虫进行二维抓取

将前一个爬虫代码复制到新的爬虫manual.py中：

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

在properties/spiders/manual.py中，我们通过添加from scrapy.http import Request引入Request，将爬虫的名字改为manual，将start\_URL改为索引首页，将parse()重命名为parse\_item()。接下来写心得parse()方法进行水平和垂直的抓取：

```
def parse(self, response):
    # Get the next index URL 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 URL 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)
```

提示：你可能注意到了yield声明。它和return很像，不同之处是return会退出循环，而yield不会。从功能上讲，前面的例子与下面很像

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

yield可以大大提高Python编程的效率。

做好爬虫了。但如果让它运行起来的话，它将抓取5万张页面。为了避免时间太长，我们可以通过命令-s CLOSESPIDER\_ITEMCOUNT=90（更多的设定见第7章），设定爬虫在一定数量（例如，90）之后停止运行。开始运行：

```
$ 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_URL': [u'http://web:9312/images/i10.jpg'],
 'price': [223.88],
 'project': ['properties'],
 'server': ['scrapyservice1'],
 '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,
```

查看输出，你可以看到我们得到了水平和垂直两个方向的结果。首先读取了index\_00000.html，然后产生了许多请求。执行请求的过程中，debug信息指明了谁用URL发起了请求。例如，我们看到，property\_000029.html, property\_000028.html ... 和 index\_00001.html都有相同的referer(即index\_00000.html)。然后，property\_000059.html和其它网页的referer是index\_00001，过程以此类推。

这个例子中，Scrapy处理请求的机制是后进先出（LIFO），深度优先抓取。最后提交的请求先被执行。这个机制适用于大多数情况。例如，我们想先抓取完列表页再取下一个索引页。不然的话，我们必须消耗内存存储列表页的URL。另外，许多时候你想用一个辅助的Requests执行一个请求，下一章有例子。你需要Requests越早完成越好，以便爬虫继续下面的工作。

我们可以通过设定Request()参数修改默认的顺序，大于0时是高于默认的优先级，小于0时是低于默认的优先级。通常，Scrapy会先执行高优先级的请求，但不会花费太多时间思考到底先执行哪一个具体的请求。在你的大多数爬虫中，你不会有超过一个或两个的请求等级。因为URL会被多重过滤，如果我们想向一个URL多次请求，我们可以设定参数dont\_filter Request()为True。

## 用CrawlSpider二维抓取

如果你觉得这个二维抓取单调的话，说明你入门了。Scrapy试图简化这些琐事，让编程更容易。完成之前结果的更好方法是使用CrawlSpider，一个简化抓取的类。我们用genspider命令，设定一个-t参数，用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properties/spiders/easy.py文件包含如下所示：

```
...
class EasySpider(CrawlSpider):
    name = 'easy'
    allowed_domains = ['web']
    start_URL = ['http://www.web/']
    rules = (
        Rule(LinkExtractor(allow=r'Items/'),
callback='parse_item', follow=True),
    )
    def parse_item(self, response):
        ...
```

这段自动生成的代码和之前的很像，但是在类的定义中，这个爬虫从CrawlSpider定义的，而不

是Spider。CrawlSpider提供了一个包含变量rules的parse()方法，以完成之前我们手写的内容。

现在将start\_URL设定为索引首页，并将parse\_item()方法替换。这次不再使用parse()方法，而是将rules变成两个rules，一个负责水平抓取，一个负责垂直抓取：

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

两个XPath表达式与之前相同，但没有了a与href的限制。正如它们的名字，LinkExtractor专门抽取链接，默认就是寻找a、href属性。你可以设定tags和attrs自定义LinkExtractor()。对比前面的请求方法Requests(self.parse\_item)，回调的字符串中含有回调方法的名字（例如，parse\_item）。最后，除非设定callback，一个Rule就会沿着抽取的URL扫描外链。设定callback之后，Rule才能返回。如果你想让Rule跟随外链，你应该从callback方法return/yield，或设定Rule()的follow参数为True。当你的列表页既有Items又有其它有用的导航链接时非常有用。

你现在可以运行这个爬虫，它的结果与之前相同，但简洁多了：

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

## 总结

对所有学习Scrapy的人，本章也许是最重要的。你学习了爬虫的基本流程UR2IM、如何自定义Items、使用ItemLoaders、XPath表达式、利用处理函数加载Items、如何yield请求。我们使用Requests水平抓取多个索引页、垂直抓取列表页。最后，我们学习了如何使用CrawlSpider和Rules简化代码。多度几遍本章以加深理解、创建自己的爬虫。

我们刚刚从一个网站提取了信息。它的重要性在哪呢？答案在下一章，我们只用几页就能制作一个移动app，并用Scrapy填充数据。

# 第4章 从Scrapy到移动应用

有人问，移动app开发平台Appery.io和Scrapy有什么关系？眼见为实。在几年前，用Excel向别人展示数据才可以让别人印象深刻。现在，除非你的受众分布很窄，他们彼此之间是非常不同的。接下来几页，你会看到一个快速构建的移动应用，一个最小可行产品。它可以向别人清楚的展示你抓取的数据的力量，为源网站搭建的生态系统带来回报。

我尽量让这个挖掘数据价值的例子简短。要是你自己就有一个使用数据的应用，你可以跳过本章。本章就是告诉你如何用现在最流行的方式，移动应用，让你的数据面向公众。

## 选择移动应用框架

---

使用适当的工具向移动应用导入数据是相当容易的。跨平台开发移动应用的框架很多，例如 PhoneGap、Appcelerator和Appcelerator云服务、jQuery Mobile和Sencha Touch。

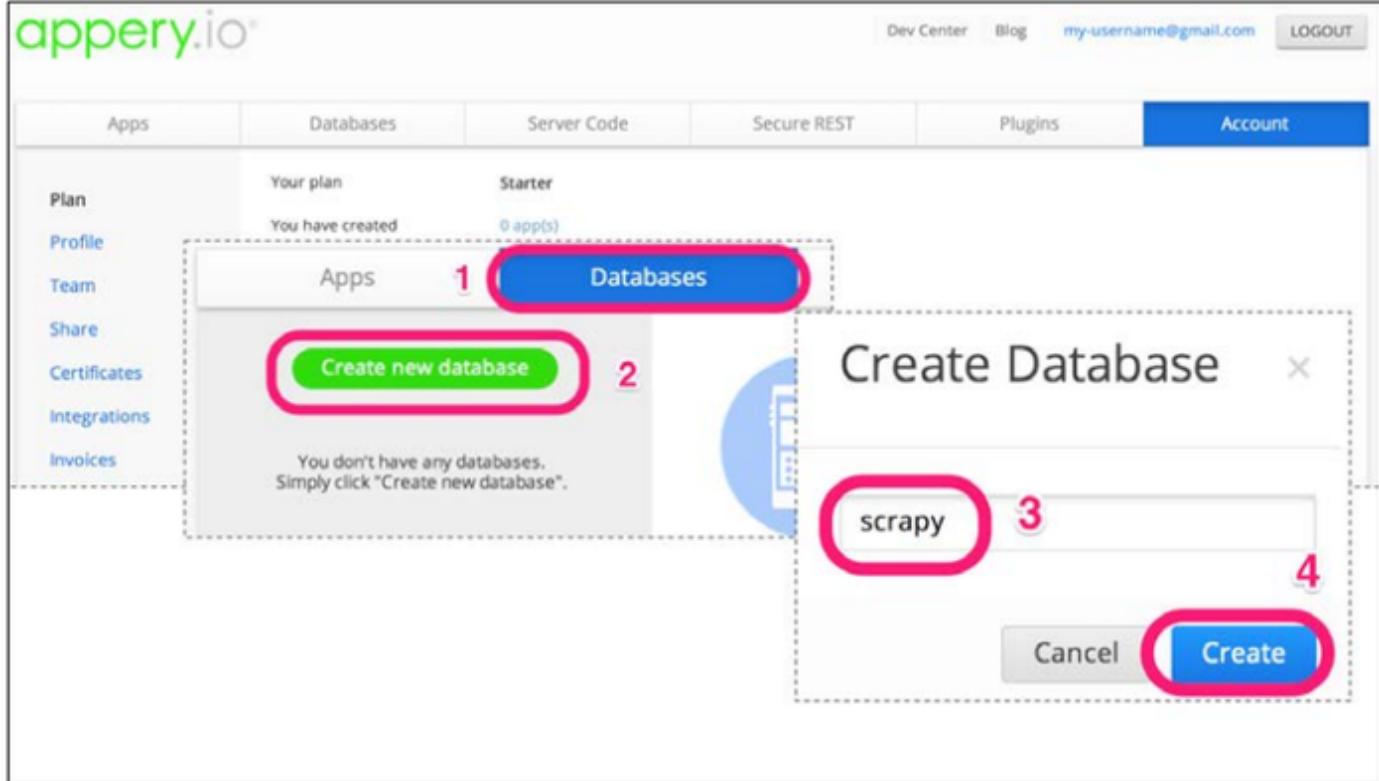
本章会使用Appery.io，因为它可以让我们用PhoneGap和jQuery Mobile快速开发iOS、Android、Windows Phone、HTML5移动应用。我并不是要为Appery.io代言，我鼓励你自己去调研下它是否符合你的需求。Appery.io是一个付费服务，但有14天的试用期。在我看来，即使是外行也可以用Appery.io快速创建一个应用。我选择它的原因是，它提供了移动和后端两个服务，所以我们不用配置数据库、写REST APIs、或在服务器和移动端使用不同的语言。你将看到，我们根本不用写任何代码！我们会使用它的在线工具，你可以随时下载app作为PhoneGap项目，使用PhoneGap的全部特性。

使用Appery.io，你需要连接网络。另外，因为它的网站可能会发生改变，如果和截图不同不要惊讶。

## 创建数据库和集合

---

第一步是注册Appery.io，并选择试用。提供名字、Email密码之后，你的账户就创立了。登录Appery.io工作台，你就可以创建数据库和集合了：



步骤如下：

- 1.点击Databases标签（1）。
- 2.然后点击绿色的Create new database按钮（2）。将新数据库命名为scrapy（3）。
- 3.现在点击Create按钮（4）。自动打开Scrapy数据库工作台，在工作台上可以新建集合。

在Appery.io中，数据库是集合的整合。粗略的讲，一个应用使用一个数据库，这个数据库中有许多集合，例如用户、特性、信息等等。Appery.io已经有了一个Users集合，用来存储用户名和密码（Appery.io有许多内建的功能）。

The screenshot shows the MongoDB Compass interface. On the left, there's a sidebar with 'Predefined collections' (Users, Files, Devices) and a 'Collections' section with 'properties'. In the main area, under 'Users', a row is selected with '\_id' (543ae91...), 'username' (root), and 'password' ((hidden)). A 'Security and permissions' tab is visible. Below this, a 'Create new collection' dialog is open, with 'properties' entered in the name field (circled in red). An 'Add' button is highlighted with a red circle. In the foreground, an 'Add new column' dialog is open for the 'Properties' collection. It shows a table with columns '\_id', 'title' (highlighted with a red circle), 'price' (number type), 'description' (string type), 'url' (string type), and 'image\_urls' (string type). The 'title' column is highlighted with a red circle. The 'Type' dropdown is set to 'String' (circled in red). A 'Create column' button is highlighted with a red circle. The 'Add' button in the 'Create new collection' dialog is also circled in red.

让我们添加一个用户，用户名是root，密码是pass。显然，密码可以更复杂。在侧边栏点击 Users (1)，然后点击+Row (2) 添加user/row。在弹出的界面中输入用户名和密码 (3,4)。

再为Scrapy抓取的数据创建一个集合，命名为properties。点击Create new collection绿色按钮 (5)，命名为properties (6)，点击Add按钮 (7)。现在，我们需要自定义这个集合。点击+Col添加列 (8)。列有一些数据类型可以帮助确认值。大多数要填入的是字符串，除了价格是个数字。点击+Col (8) 再添加几列，填入列的名字 (9)、数据类型 (10)，然后点击Create column按钮 (11)。重复五次这个步骤以创建下表：

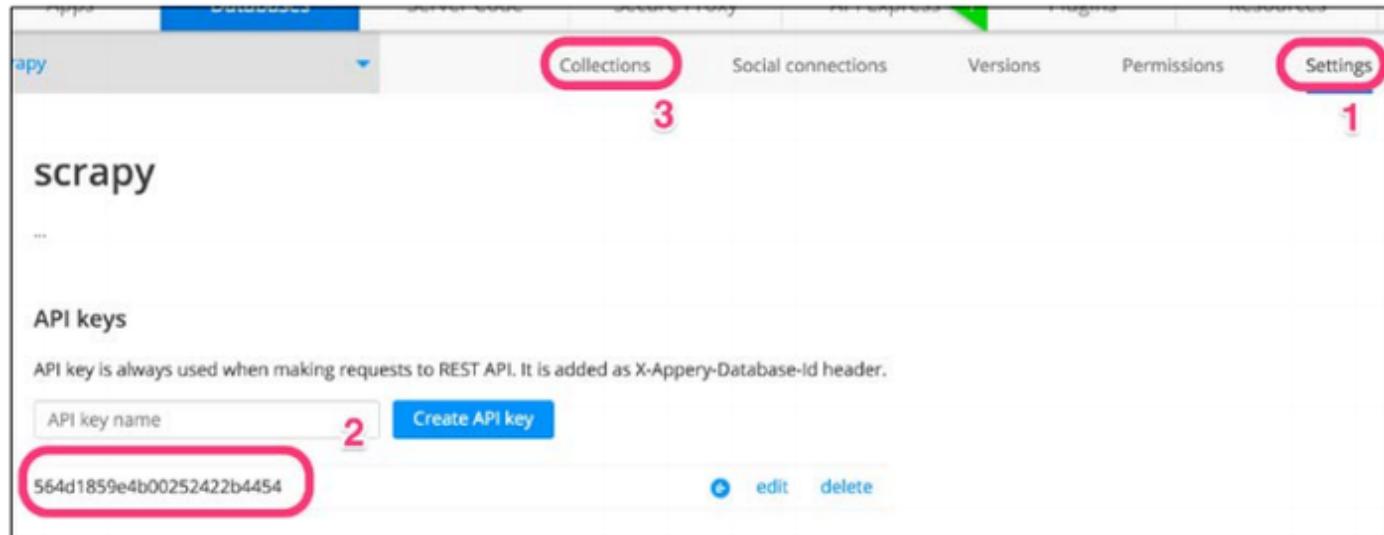
Column	title	price	description	url	image_urls
Type	string	number	string	string	string

创建好所有列之后，就可以导入数据了。

## 用Scrapy导入数据

首先，我们需要API key，在Settings中可以找到 (1)。复制它 (2)，然后点击Collections标签

返回集合 (3) :



现在，修改一下上一章的代码，以导入数据。我们把名字是easy.py的爬虫中的代码复制到名字是tomobile.py的爬虫中：

```
$ 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_URL = (
        'http://scrapybook.s3.amazonaws.com/properties/'
        'index_00000.html',
    )
...
...
```

你可能注意到了，我们没有使用网络服务器<http://web:9312>。我们用的是我托管在<http://scrapybook.s3.amazonaws.com>上的副本。使用它，我们的图片和URL所有人都可以访问，更易分享我们的app。

我们使用Appery.io pipeline导入数据。Scrapy的pipelines是后处理的、简洁的、可以存储items的很小的Python类。第8章中会详细讲解两者。现在，你可以用easy\_install或pip安装，但如果你用Vagrant开发机，因为已经都安装好了，你就不用再安装了：

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

或

```
$ sudo pip install --upgrade scrapyappyio
```

这时，要在Scrapy的设置文件中添加API key。更多关于设置的内容会在第7章中介绍。现在，我们只需在properties/settings.py文件后面加入如下代码：

```
ITEM_PIPELINES = {'scrapyappyio.ApperyIoPipeline': 300}
APPERYIO_DB_ID = '<<Your API KEY here>>'
APPERYIO_USERNAME = 'root'
APPERYIO_PASSWORD = 'pass'
APPERYIO_COLLECTION_NAME = 'properties'
```

别忘了将APPERYIO\_DB\_ID替换为API key。还要确认你的设置有和Appery.io相同的用户名和密码。要进行向Appery.io注入数据，像之前一样用Scrapy抓取：

```
$ 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)
```

输出的结果略有不同。你可以看到代码的前几行运行了ApperyIoPipeline的项目pipeline；更显著的是，大概抓取了100个项目，有约200个请求/响应。这是因为Appery.io pipeline为写入每个项目，都额外的做了一次请求。这些请求也出现在日志中，带有api.appery.io URL。

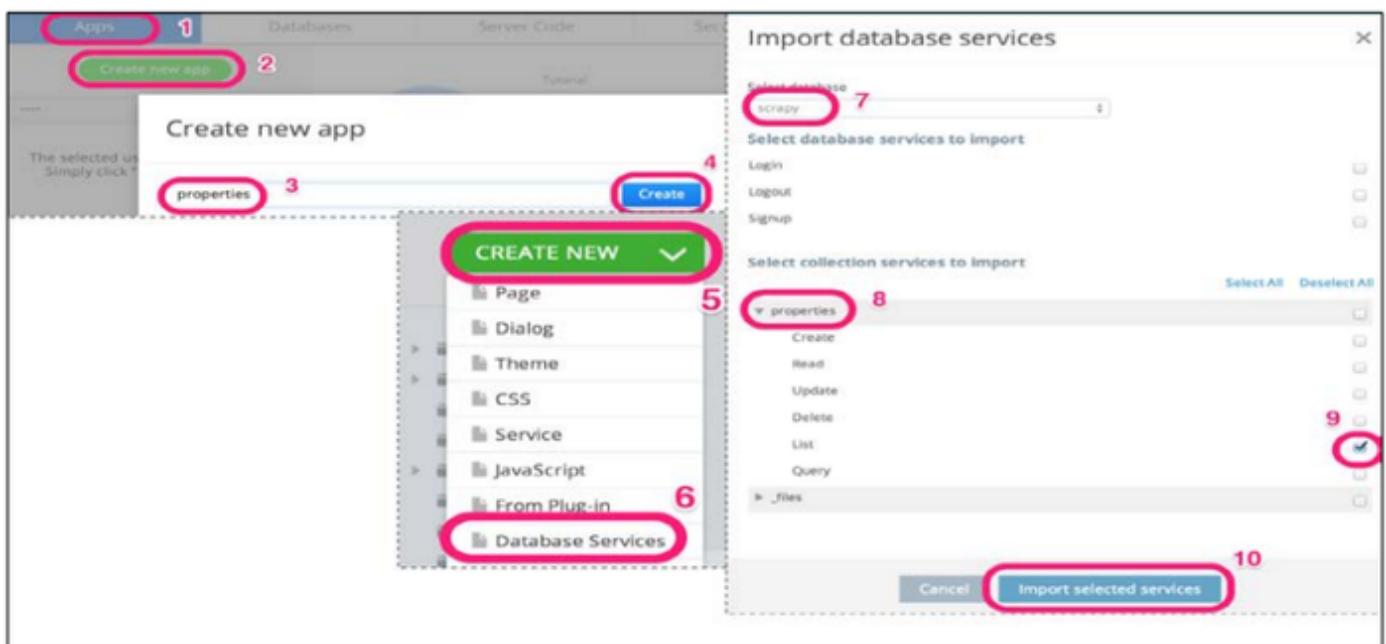
The screenshot shows the Appery.io database interface. On the left, there's a sidebar titled 'Collections' with a list containing 'properties'. A red circle highlights the word 'properties'. At the top, there are several buttons: 'Rename', 'Delete', 'Security and permissions', 'Manage indexes', and 'Change'. A red arrow points to the 'Delete All' button in the top right corner. Below these buttons is a table representing the 'properties' collection with columns: '\_id', 'title', 'price', 'description', 'url', and 'image\_urls'. There are four rows of data.

	_id	title	price	description	url	image_urls
<input type="checkbox"/>	543aea0...	Vegetari...	203.23	semi too...	http://sc...	http://sc...
<input type="checkbox"/>	543aea0...	John Mo...	152.8	set gard...	http://sc...	http://sc...
<input type="checkbox"/>	543aea0...	Caledoni...	197.78	followin...	http://sc...	http://sc...
<input type="checkbox"/>	543aea0...	Morden ...	200.2	underm...	http://sc...	http://sc...

如果返回Appery.io，我们可以properties集合（1）中填入了数据（2）。

## 创建移动应用

创建移动应用有点繁琐。点击Apps标签（1），然后点击Create new app（2）。将这个应用命名为properties（3），再点击Create按钮（4）：

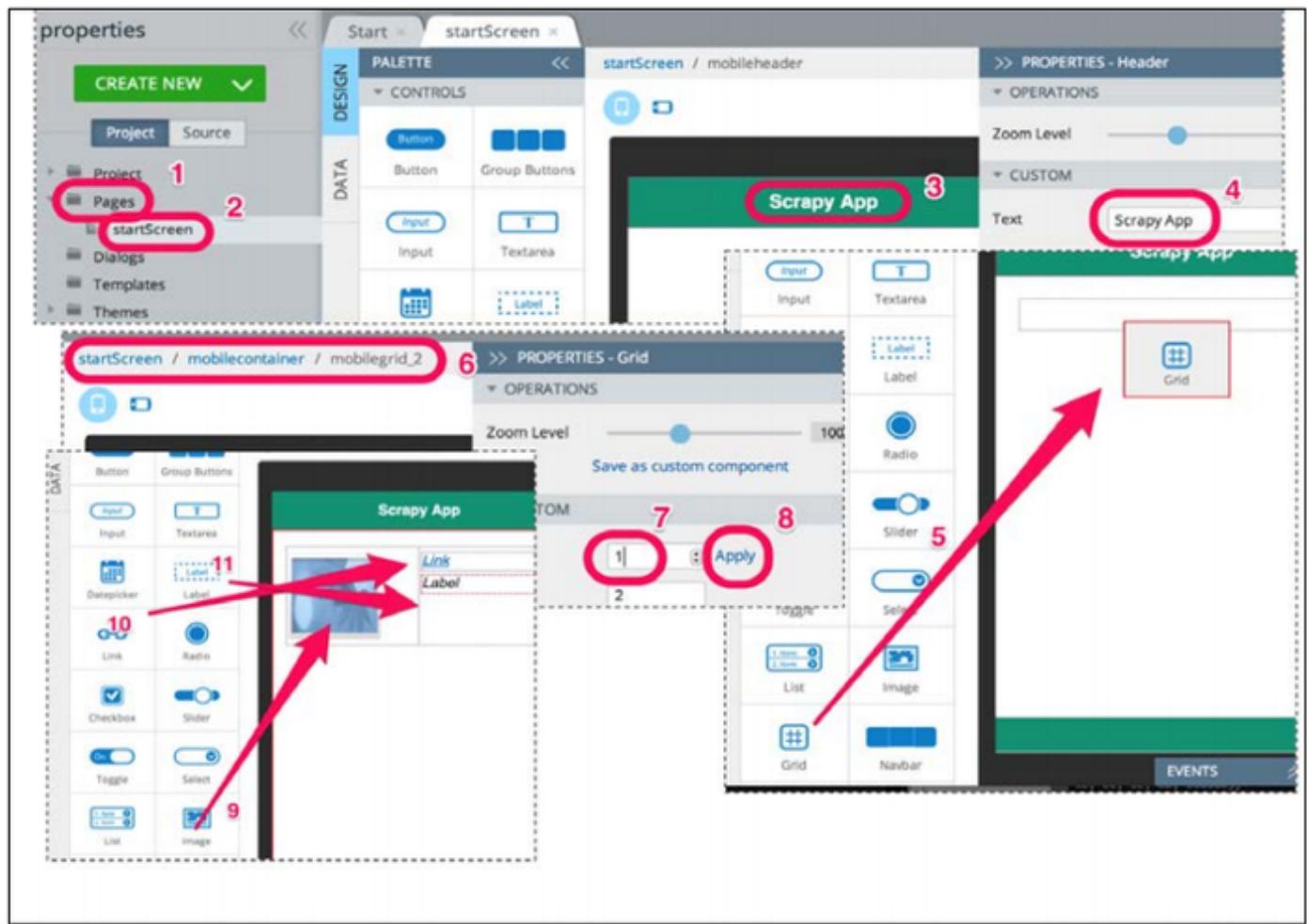


## 创建数据库接入服务

创建应用的选项很多。使用Appery.io应用编辑器可以编写复杂应用，但我们的应用力求简单。让我们的应用连接Scrapy数据库，点击CREATE NEW按钮（5），选择Database Services（6）。弹出一个界面让我们选择连接的对象。我们选择scrapy数据库（7）。点击properties栏（8），选择List（9）。这些操作可以让我们爬到的数据可用于数据库。最后点击Import selected services完成导入（10）。

## 设定用户界面

接下来创建app的界面。我们在DESIGN标签下工作：



在左侧栏中点开Pages文件夹（1），然后点击startScreen（2）。UI编辑器会打开一个页面，我们在上面添加空间。先修改标题。点击标题栏，在右侧的属性栏修改标题为Scrapy App。同时，标题栏会更新。

然后，我们添加格栅组件。从左侧的控制板中拖动Grid组件（5）。这个组件有两行，而我们只要一行。选择这个格栅组件，选中的时候，它在路径中会变为灰色（6）。选中之后，在右侧的属性栏中编辑Rows为1，然后点击Apply（7,8）。现在，格栅就只有一行了。

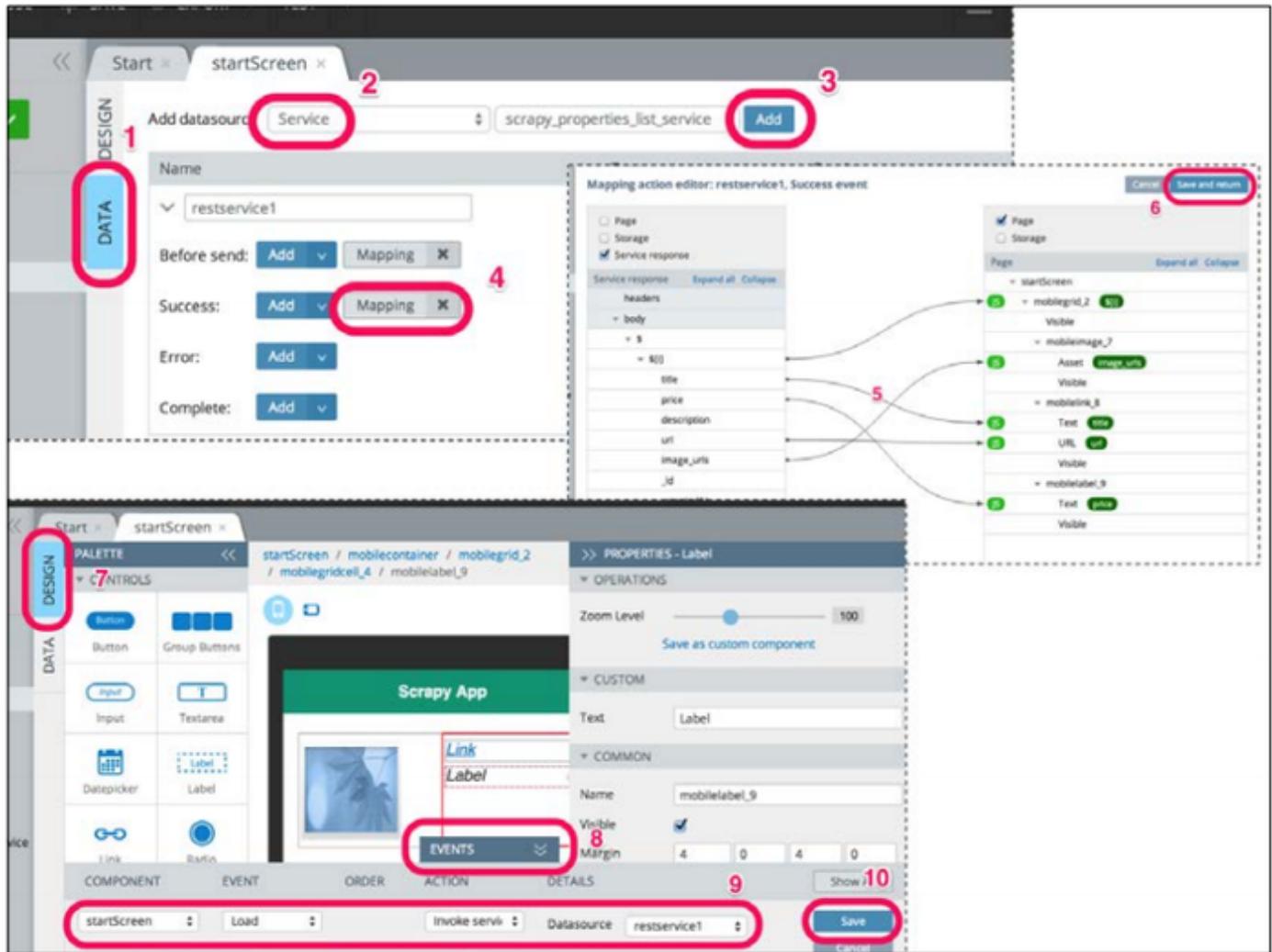
最后，再向格栅中拖进一些组件。先在左边添加一个图片组件（9），然后在右侧添加一个链接（10）。最后，在链接下添加一个标签（11）。

排版结束。接下来将数据从数据库导入用户界面。

## 将数据映射到用户界面

截止目前，我们只是在DESIGN标签下设置界面。为了连接数据和组件，我们切换到DATA标签

(1) :



我们用Service (2) 作为数据源类型，它会自动选择我们之前建立的唯一可用数据。点击Add按钮 (3) 。点击Add之后，可以在下方看到一系列事件，例如Before send和Success。点击Success后面的Mapping可以调用服务，我们现在对它进行设置。

打开Mapping action editor，在上面进行连线。编辑器有两个部分。左边是服务的可用响应，右边是UI组件的属性。两边都有一个Expand all，展开所有的项，以查看可用的。接下来按照下表，用从左到右拖动的方式完成五个映射 (5)：

响应	组件	属性	注释
\$[i]	mobilegrid_2		为每一行建立一个 for 循环
Title	mobilegrid_8	Text	设置链接的文本
Price	mobilegrid_9	Text	设置价格
Image_URL	mobilegrid_7	Asset	从图片容器的 URL 加载图片
url	mobilegrid_8	URL	为 URL 设置链接，当用户点击时， 关联加载页面。

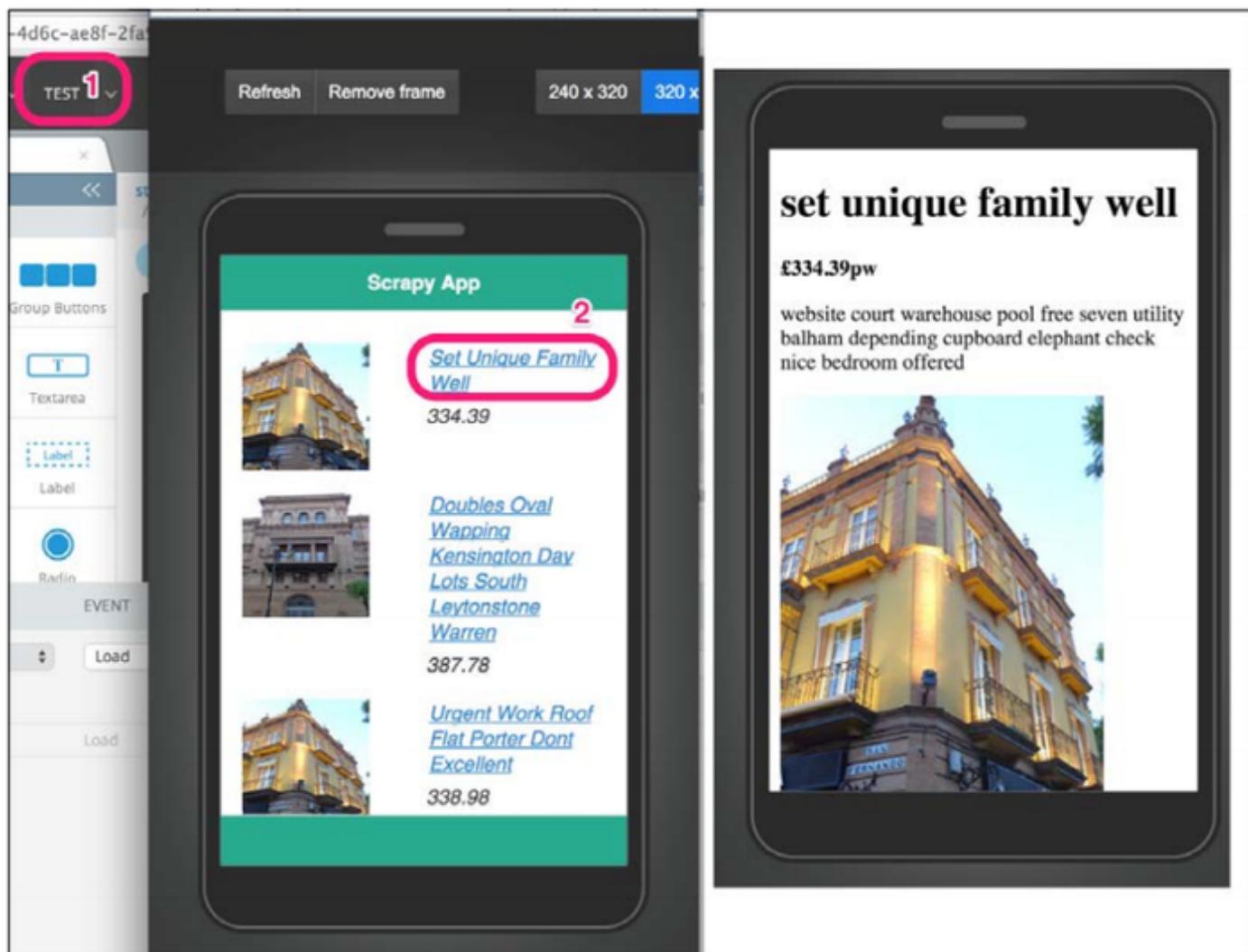
# 映射数据字段和用户组件

前面列表中的数字可能在你的例子中是不同的，但是因为每种组件的类型都是唯一的，所以连线出错的可能性很小。通过映射，我们告诉Appery.io当数据库查询成功时载入数据。然后点击 Save and return (6)。

返回DATA标签。我们需要返回UI编辑器，点击DESIGN标签 (7)。屏幕下方，你会看到 EVENTS区域 (8) 被展开了。利用EVENTS，我们让Appery.io响应UI时间。下面是最后一步，就是加载UI时调用服务取回数据。我们打开startScreen作为组件，事件的默认选项是Load。然后选择Invoke service作为action，然后用Datasource作为默认的restservice1选项 (9)。点击 Save (10)，保存这个移动应用。

## 测试、分享、生成app

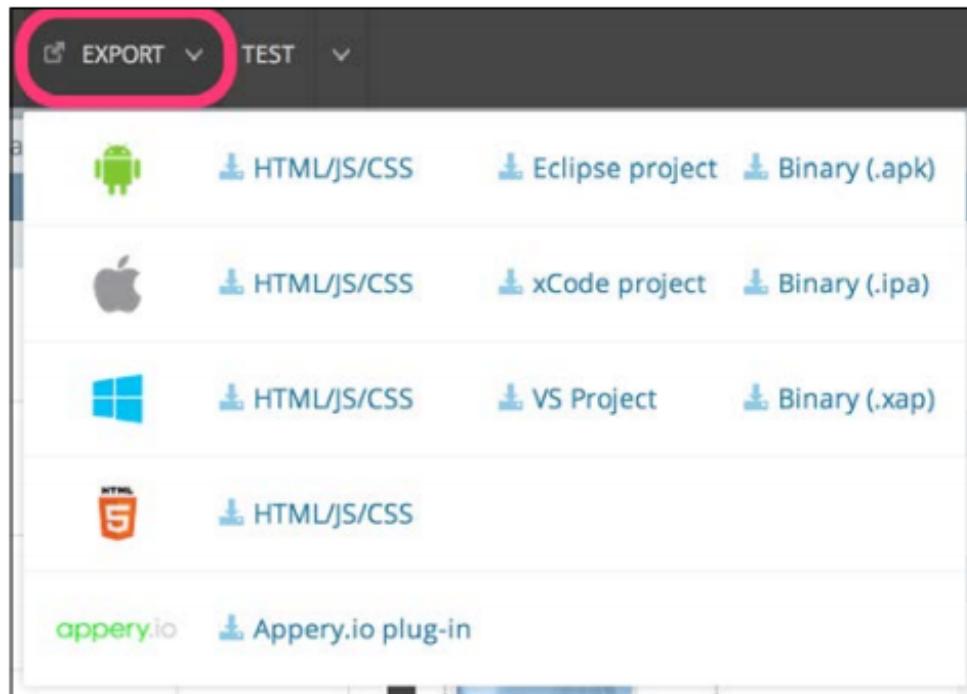
现在准备测试app。我们要做的是点击UI上方的TEST按钮 (1)：



这个应用直接在浏览器中运行。链接 (2) 是启动的，可以进行跳转。你可以设置分辨率和屏幕

的横竖。你还可以点击View on Phone，创建一个二维码，用手机扫描，然后在手机上看。你刚刚创建了一个链接，别人也可以在他们的浏览器中查看。

只需几次点击，我们就用一个移动应用展示了Scrapy抓取的数据。你可以在这个网页，<http://devcenter.appery.io/tutorials/>学习Appery.io教程，继续定制这个应用。当你准备好之后，可以点击EXPORT按钮输出这个app：



你可以输出文档到你喜爱的IDE继续开发，或是生成在各个平台都能运行的app。

## 总结

使用Scrapy和Appery.io两个工具，我们创建了一个爬虫、抓取了一个网站，并将数据存到数据库之中。我们还创建了RESTful API和一个简单的移动端应用。对于更高级的特点和进一步开发，你可以进一步探究这个平台，或将这个应用用于实际或科研。现在，用最少的代码，你就可以用一个小产品展示网络抓取的应用了。

鉴于这么短的开发时间，我们的app就有不错的效果。它有真实的数据，而不是Lorem Ipsum占字符，所有的链接运行良好。我们成功地制作了一个最小可行产品，它可以融合进源网站的生态，提高流量。

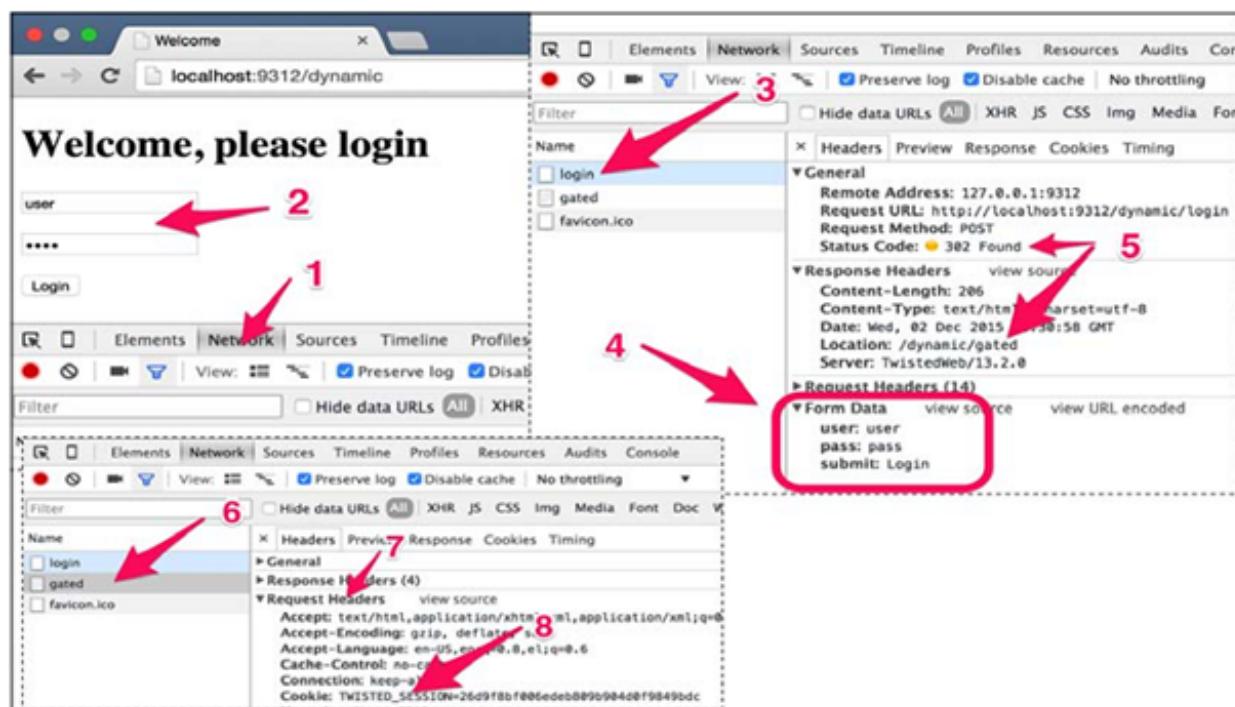
接下来学习在更加复杂的情况下，如何使用Scrapy爬虫提取信息。

# 第5章 快速构建爬虫

第3章中，我们学习了如何从网页提取信息并存储到Items中。大多数情况都可以用这一章的知识处理。本章，我们要进一步学习抓取流程UR2IM中两个R，Request和Response。

## 一个具有登录功能的爬虫

你常常需要从具有登录机制的网站抓取数据。多数时候，网站要你提供用户名和密码才能登录。我们的例子，你可以在<http://web:9312/dynamic>或<http://localhost:9312/dynamic>找到。用用户名“user”、密码“pass”登录之后，你会进入一个有三条房产链接的网页。现在的问题是，如何用Scrapy登录？



让我们使用谷歌Chrome浏览器的开发者工具搞清楚登录的机制。首先，选择Network标签

(1)。然后，填入用户名和密码，点击Login (2)。如果用户名和密码是正确的，你会进入下一页。如果是错误的，会看到一个错误页。

一旦你点击了Login，在开发者工具的Network标签栏中，你就会看到一个发往<http://localhost:9312/dynamic/login>的请求Request Method: POST。

**提示：**上一章的GET请求，通常用来获取静止数据，例如简单的网页和图片。POST请求通常用来获取的数据，取决于我们发给服务器的数据，例如这个例子中的用户名和密码。

点击这个POST请求，你就可以看到发给服务器的数据，其中包括表单信息，表单信息中有你刚才输入的用户名和密码。所有数据都以文本的形式发给服务器。Chrome开发者工具将它们整理好并展示出来。服务器的响应是302 FOUND (5)，然后将我们重定向到新页面：/dynamic/gated。只有登录成功时才会出现此页面。如果没有正确输入用户名和密码就前往<http://localhost:9312/dynamic/gated>，服务器会发现你作弊，并将你重定向到错误页面：<http://>

[localhost:9312/dynamic/error](http://localhost:9312/dynamic/error)。服务器怎么知道你和密码呢？如果你点击左侧的gated (6)，你会发现RequestHeaders (7) 下有一个Cookie (8)。

提示：HTTP cookie是通常是一些服务器发送到浏览器的短文本或数字片段。反过来，在每一个后续请求中，浏览器把它发送回服务器，以确定你、用户和期限。这让你可以执行复杂的需要服务器端状态信息的操作，如你购物车中的商品或你的用户名和密码。

总结一下，单单一个操作，如登录，可能涉及多个服务器往返操作，包括POST请求和HTTP重定向。Scrapy处理大多数这些操作是自动的，我们需要编写的代码很简单。

我们将第3章名为easy的爬虫重命名为login，并修改里面名字的属性，如下：

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

提示：本章的代码github的ch05目录中。这个例子位于ch05/properties。

我们要在<http://localhost:9312/dynamic/login>上面模拟一个POST请求登录。我们用Scrapy中的类FormRequest来做。这个类和第3章中的Request很像，但有一个额外的formdata，用来传递参数。要使用这个类，首先必须要引入：

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

我们然后将start\_URL替换为start\_requests()方法。这么做是因为在本例中，比起URL，我们要做一些自定义的工作。更具体地，用下面的函数，我们创建并返回一个FormRequest：

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

就是这样。CrawlSpider的默认parse()方法，即LoginSpider的基本类，负责处理响应，并如第3章中使用Rules和LinkExtractors。其余的代码很少，因为Scrapy负责了cookies，当我们登录时，Scrapy将cookies传递给后续请求，与浏览器的方式相同。还是用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_URL': [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,
```

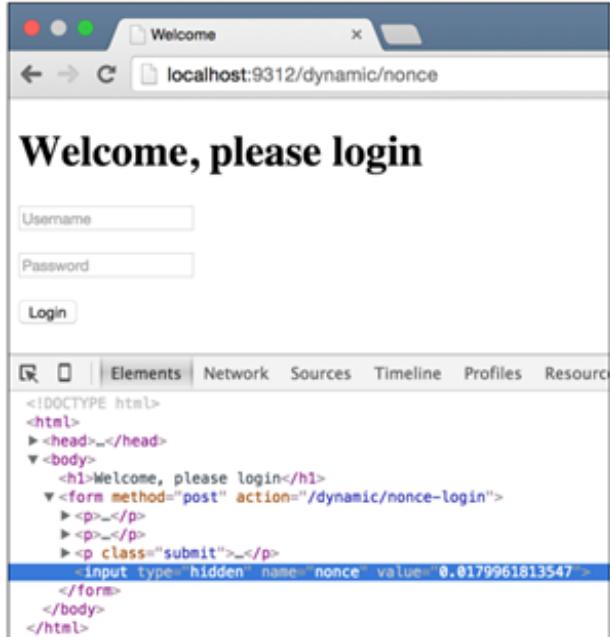
我们注意到登录跳转从dynamic/login到dynamic/gated，然后就可以像之前一样抓取项目。在统计中，我们看到一个POST请求和四个GET请求；一个是dynamic/gated首页，三个是房产网页。

提示：在本例中，我们不保护房产页，而是是这些网页的链接。代码在相反的情况下也是相同的。

如果我们使用了错误的用户名和密码，我们将重定向到一个没有URL的页面，进程并将在那里结束，如下所示：

```
$ 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，演示了基本的登录机制。大多数网站可能有更复杂的机制，但Scrapy也处理的很好。例如一些网站在执行POST请求时，需要通过从表单页面到登录页面传递某种形式的变量以确定cookies的启用，让你使用大量用户名和密码暴力破解时变得困难。



例如，如果你访问<http://localhost:9312/dynamic/nonce>，你会看到一个和之前一样的网页，但如果你使用Chrome开发者工具，你会发现这个页面的表单有一个叫做nonce的隐藏字段。当你提交表单<http://localhost:9312/dynamic/nonce-login>时，你必须既要提供正确的用户名密码，还要提交正确的浏览器发给你的nonce值。因为这个值是随机且只能使用一次，你很难猜到。这意味着，如果要成功登陆，必须要进行两次请求。你必须访问表单、登录页，然后传递数值。和以前一样，Scrapy有内建的功能可以解决这个问题。

我们创建一个和之前相似的NonceLoginSpider爬虫。现在，在start\_requests()中，我们要向表单页返回一个简单的Request，并通过设定callback为名字是parse\_welcome()的方法手动处理响应。在parse\_welcome()中，我们使用FormRequest对象中的from\_response()方法创建FormRequest，并将原始表单中的字段和值导入FormRequest。FormRequest.from\_response()可以模拟提交表单。

**提示：**花时间看from\_response()的文档是十分值得的。他有许多有用的功能如formname和formnumber，它可以帮助你当页面有多个表单时，选择特定的表单。

它最大的功能是，一字不差地包含了表单中所有的隐藏字段。我们只需使用formdata参数，填入user和pass字段，并返回FormRequest。代码如下：

```
# 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"}  
)
```

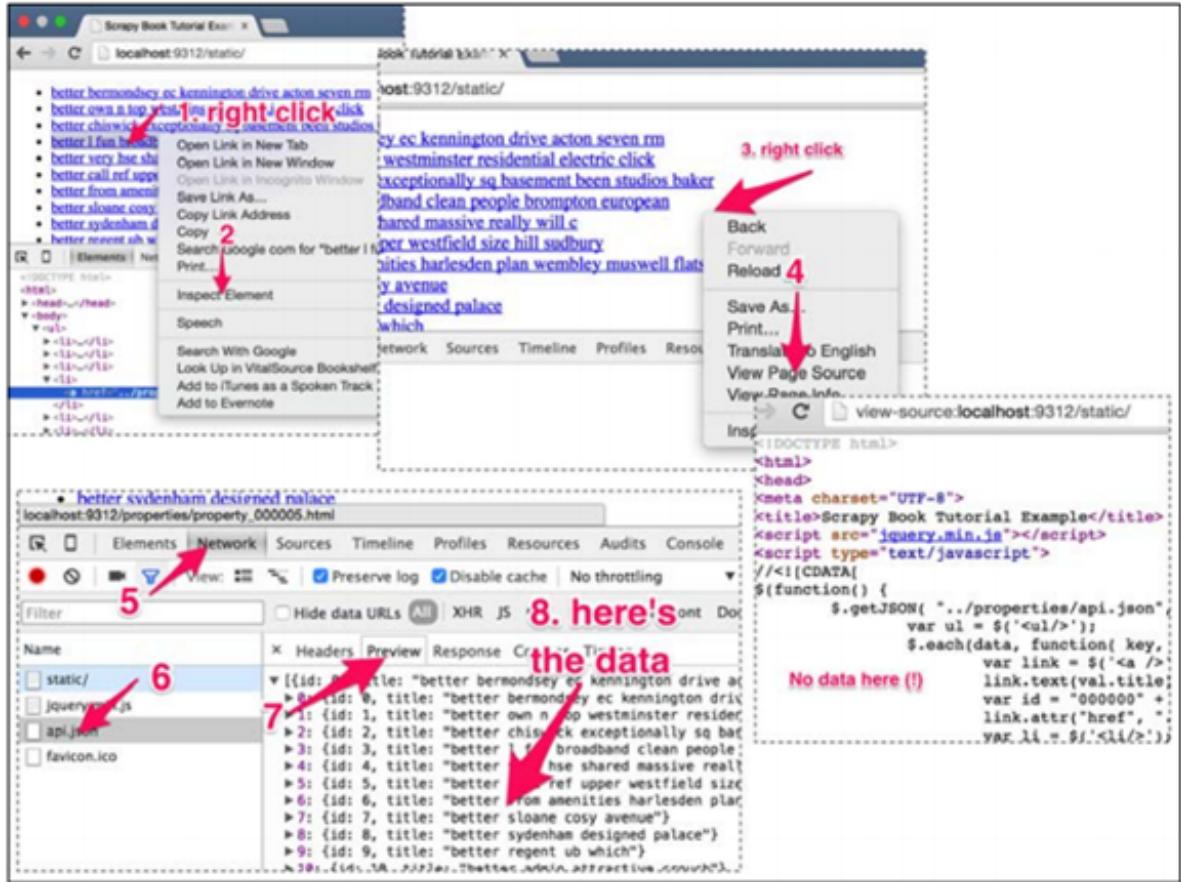
像之前一样运行爬虫：

```
$ 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,
```

我们看到第一个GET请求先到/dynamic/nonce，然后POST，重定向到/dynamic/nonce-login之后，之后像之前一样，访问了/dynamic/gated。登录过程结束。这个例子的登录含有两步。只要有足够的耐心，无论多少步的登录过程，都可以完成。

## 使用JSON APIs和AJAX页面的爬虫

有时，你会发现网页的HTML找不到数据。例如，在<http://localhost:9312/static/>页面上右键点击检查元素（1,2），你就可以在DOM树中看到所有HTML元素。或者，如果你使用scrapy shell或在Chrome中右键点击查看网页源代码（3,4），你会看到这个网页的HTML代码不包含任何和值有关的信息。数据都是从何而来呢？



和以前一样，在开发者工具中打开Network标签（5）查看发生了什么。左侧列表中，可以看到所有的请求。在这个简单的页面中，只有三个请求：static/我们已经检查过了，jquery.min.js是一个流行的JavaScript框架，api.json看起来不同。如果我们点击它（6），然后在右侧点击Preview标签（7），我们可以看到它包含我们要找的信息。事实上，<http://localhost:9312/properties/api.json>包含IDs和名字（8），如下所示：

```
[
  {"id": 0,
    "title": "better set unique family well"},
  ... {
    "id": 29,
    "title": "better portered mile"}
]
```

这是一个很简单的JSON API例子。更复杂的APIs可能要求你登录，使用POST请求，或返回某种数据结构。任何时候，JSON都是最容易解析的格式，因为不需要XPath表达式就可以提取信息。

Python提供了一个强大的JSON解析库。当我们import json时，我们可以使用json.loads (response.body) 解析JSON，并转换成等价的Python对象，语句、列表和字典。

复制第3章中的manual.py文件。这是最好的方法，因为我们要根据JSON对象中的IDs手动创建URL和Request。将这个文件重命名为api.py，重命名类为ApiSpider、名字是api。新的start\_URL变成：

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

如果你要做POST请求或更复杂的操作，你可以使用start\_requests()方法和前面几章介绍的方法。这里，Scrapy会打开这个URL并使用Response作为参数调用parse()方法。我们可以import json，使用下面的代码解析JSON：

```
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)
```

这段代码使用了json.loads (response.body) 将响应JSON对象转换为Python列表，然后重复这个过程。对于列表中的每个项，我们设置一个URL，它包含：base\_url, property\_%06d 和.html.base\_url, .html.base\_url前面定义过的URL前缀。%06d是一个非常有用的Python词，可以让我们结合多个Python变量形成一个新的字符串。在本例中，用id变量替换%06d。id被当做数字（%d的意思就是当做数字进行处理），并扩展成6个字符，位数不够时前面添加0。如果id的值是5，%06d会被替换为000005；id是34322时，%06d会被替换为034322替换。最后的结果是可用的URL。和第3章中的yield一样，我们用URL做一个新的Request请求。运行爬虫：

```
$ 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,
```

最后一共有31次请求，每个项目一次，api.json一次。

## 在响应间传递参数

许多时候，你想把JSON APIs中的信息存储到Item中。为了演示，在我们的例子中，对于一个项，JSON API在返回它的名字时，在前面加上“better”。例如，如果一个项的名字是“Covent Garden”，API会返回“Better Covent Garden”。我们要在Items中保存这些含有“better”的名字。如何将数据从parse()传递到parse\_item()中呢？

我们要做的就是在parse()方法产生的Request中进行设置。然后，我们可以从parse\_item()的Response中取回。Request有一个名为meta的字典，在Response中可以直接访问。对于我们的例子，给字典设一个title值以存储从JSON对象的返回值：

```
title = item["title"]
yield Request(url, meta={"title": title}, callback=self.parse_item)
```

在parse\_item()中，我们可以使用这个值，而不用XPath表达式：

```
l.add_value('title', response.meta['title'],
    MapCompose(unicode.strip, unicode.title))
```

你会注意到，我们从调用add\_xpath()切换到add\_value()，因为对于这个字段不需要使用XPath。我们现在运行爬虫，就可以在PropertyItems中看到api.json中的标题了。

## 一个加速30倍的项目爬虫

当你学习使用一个框架时，这个框架越复杂，你用它做任何事都会很复杂。可能你觉得Scrapy也是这样。当你就要为XPath和其他方法变得抓狂时，不妨停下来思考一下：我现在抓取网页的方法是最简单的吗？

如果你可以从索引页中提取相同的信息，就可以避免抓取每一个列表页，这样就可以节省大量的工作。

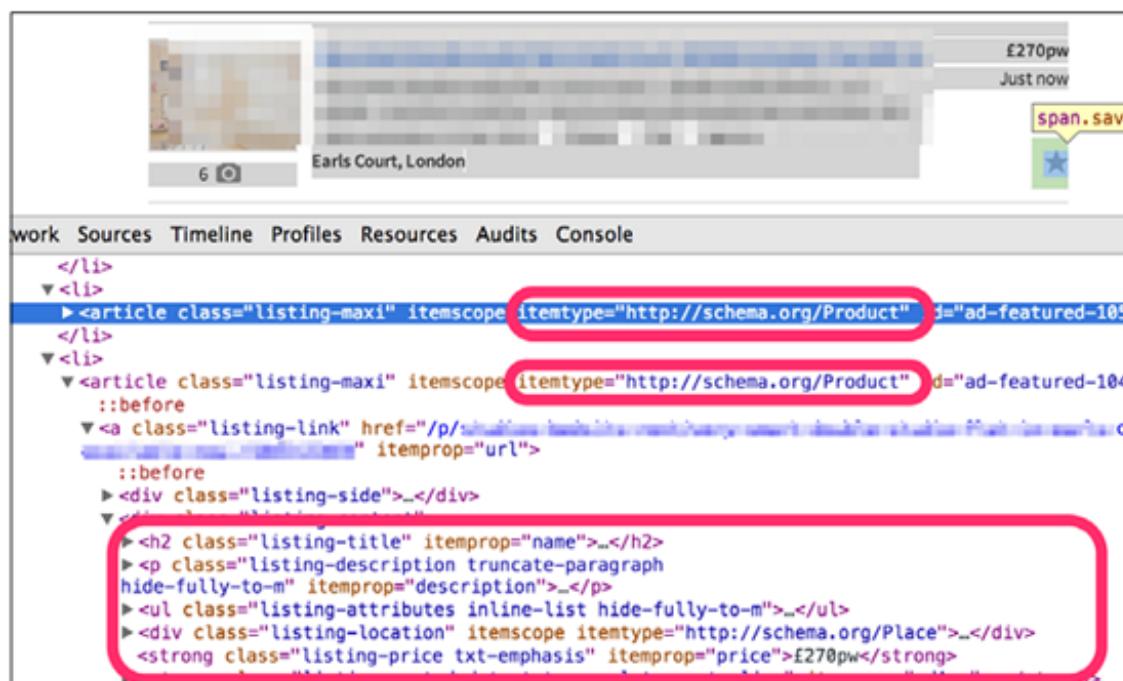
提示：许多网站的索引页提供的项目数量是不同的。例如，一个网站可以通过调整一个参数，例如&show=50，给每个索引页面设置10、50或100个列表项。如果是这样的话，将其设置为可用的最大值。

例如，对于我们的例子，我们需要的所有信息都存在于索引页中，包括标题、描述、价格和图片。这意味着我们抓取单个索引页，提取30个条目和下一个索引页的链接。通过抓取100个索引页，我们得到3000个项，但只有100个请求而不是3000个。

在真实的Gumtree网站上，索引页的描述比列表页的完整描述要短。这是可行的，或者是更推荐的。

**提示：**许多情况下，您不得不在数据质量与请求数量间进行折衷。很多网站都限制请求数量（后面章节详解），所以减少请求可能解决另一个棘手的问题。

在我们的例子中，如果我们查看一个索引页的HTML，我们会发现，每个列表项有自己的节点，itemtype="<http://schema.org/Product>"。节点有每个项的全部信息，如下所示：



The screenshot shows a listing item from the Gumtree website. The item includes a thumbnail image, a title ('Earls Court, London'), a price ('£270pw'), and a timestamp ('Just now'). A 'span.save' button is visible. Below the listing, a portion of the website's navigation bar is shown, followed by the browser's developer tools. The HTML source code is displayed, highlighting several elements with red boxes. These highlighted elements are all 'article' tags with the class 'listing-maxi' and the attribute 'itemtype="http://schema.org/Product"'. One of these highlighted articles is further expanded to show its internal structure, including 'listing-title', 'listing-description', 'listing-attributes', and 'listing-location'.

让我们在Scrapy shell中加载索引首页，并用XPath处理：

```
$ scrapy shell http://web:9312/properties/index_00000.html
While within the Scrapy shell, let's try to select everything with the Product tag:
>>> p=response.xpath('//*[@itemtype="http://schema.org/Product"]')
>>> len(p)
30
>>> p
[<Selector xpath='//*[@itemtype="http://schema.org/Product"]' data=u'<li
class="listing-maxi" itemscopeitemt'...]
```

我们得到了一个包含30个Selector对象的表，每个都指向一个列表。Selector对象和Response对象很像，我们可以用XPath表达式从它们指向的对象中提取信息。不同的是，表达式为有相关性

的XPath表达式。相关性XPath表达式与我们之前见过的很像，不同之处是它们前面有一个点“.”。然我们看看如何用`./[@itemprop="name"]`[1]/text()提取标题的：

```
>>> selector = p[3]
>>> selector
<Selector xpath='//*[@itemtype="http://schema.org/Product"]' ... '>
>>> selector.xpath('.//*[@itemprop="name"]')[1].text()
[u'l fun broadband clean people brompton european']
```

我们可以在Selector对象表中用for循环提取一个索引页的所有30个项目信息。还是从第3章中的maunal.py文件开始，重命名为fast.py。重复使用大部分代码，修改parse()和parse\_item()方法。更新的方法如下所示：

```
def parse(self, response):
    # Get the next index URL and yield Requests
    next_sel = response.xpath('//*[contains(@class, "next")]/@href')
    for url in next_sel.extract():
        yield Request(urlparse.urljoin(response.url, url))
    # Iterate through products and create PropertiesItems
    selectors = response.xpath(
        '//*[@itemtype="http://schema.org/Product"]')
    for selector in selectors:
        yield self.parse_item(selector, response)
```

第一部分中用于产生下一条索引请求的代码没有变动。不同的地方是第二部分，我们重复使用选择器调用parse\_item()方法，而不是用yield创建请求。这和原先使用的源代码很像：

```
def parse_item(self, selector, response):
    # Create the loader using the selector
    l = ItemLoader(item=PropertiesItem(), selector=selector)
    # 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))
make_url = lambda i: urlparse.urljoin(response.url, i)
l.add_xpath('image_URL', './/*[@itemprop="image"][1]/@src',
            MapCompose(make_url))
# Housekeeping fields
l.add_xpath('url', './/*[@itemprop="url"][1]/@href',
            MapCompose(make_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()

```

我们做出的变动是：

- ItemLoader现在使用selector作为源，不使用Response。这么做可以让ItemLoader更便捷，可以让我们从特定的区域而不是整个页面抓取信息。
- 通过在前面添加“.”使XPath表达式变为相关XPath。

**提示：**碰巧的是，在我们的例子中，XPath表达式在索引页和介绍页中是相同的。不同的时候，你需要按照索引页修改XPath表达式。

- 在response.url给我们列表页的URL之前，我们必须自己编辑Item的URL。然后，它才能返回我们抓取网页的URL。我们必须用.//\*[@itemprop="url"][1]/@href提取URL，然后将它用MapCompose转化为URL绝对路径。

这些小小大量的工作的改动可以节省大量的工作。现在，用以下命令运行爬虫：

```

$ scrapy crawl fast -s CLOSESPIDER_PAGECOUNT=3
...
INFO: Dumping Scrapy stats:
'downloader/request_count': 3, ...
'item_scraped_count': 90, ...

```

就像之前说的，我们用三个请求，就抓取了90个项目。不从索引开始的话，就要用93个请求。

如果你想用scrapy parse来调试，你需要如下设置spider参数：

```

$ scrapy parse --spider=fast http://web:9312/properties/index_00000.html
...
>>> STATUS DEPTH LEVEL 1 <<<
# Scrapped Items -----

```

```
[{'address': [u'Angel, London'],
... 30 items...
# Requests -----
[<GET http://web:9312/properties/index_00001.html>]
```

正如所料，parse()返回了30个Items和下一个索引页的请求。你还可以继续试验scrapy parse，例如，设置-depth=2。

## 可以抓取Excel文件的爬虫

大多数时候，你每抓取一个网站就使用一个爬虫，但如果要从多个网站抓取时，不同之处就是使用不同的XPath表达式。为每一个网站配置一个爬虫工作太大。能不能只使用一个爬虫呢？答案是可以。

新建一个项目抓取不同的东西。当前我们是在ch05的properties目录，向上一级：

```
$ pwd
/root/book/ch05/properties
$ cd ..
$ pwd
/root/book/ch05
```

新建一个项目，命名为generic，再创建一个名为fromcsv的爬虫：

```
$ scrapy startproject generic
$ cd generic
$ scrapy genspider fromcsv example.com
```

新建一个.csv文件，它是我们抓取的目标。我们可以用Excel表建这个文件。如下表所示，填入URL和XPath表达式，在爬虫的目录中（有scrapy.cfg的文件夹）保存为todo.csv。保存格式是csv：

	A	B	C
1	url	name	price
2	<a href="http://web:9312/static/a.html">http://web:9312/static/a.html</a>	//*[@id="itemTitle"]/text()	//*[@id="prclsum"]/text()
3	<a href="http://web:9312/static/b.html">http://web:9312/static/b.html</a>	//h1/text()	//span/strong/text()
4	<a href="http://web:9312/static/c.html">http://web:9312/static/c.html</a>	//*[@id="product-desc"]/span/text()	

一切正常的话，就可以在终端看见这个文件：

```
$ cat todo.csv
url,name,price
a.html,"//*[@id=""itemTitle""]/text()",//*[@id=""prcIsum""]/text()
b.html,//h1/text(),//span/strong/text()
c.html,"//*[@id=""product-desc""]/span/text()"
```

Python中有csv文件的内建库。只需import csv，就可以用后面的代码一行一行以dict的形式读取这个csv文件。在当前目录打开Python命令行，然后输入：

```
$ pwd
/root/book/ch05/generic2
$ python
>>> import csv
>>> with open("todo.csv", "rU") as f:
    reader = csv.DictReader(f)
    for line in reader:
        print line
```

文件的第一行会被自动作为header，从而导出dict的键名。对于下面的每一行，我们得到一个包含数据的dict。用for循环执行每一行。前面代码的结果如下：

```
{"url": ' http://a.html', 'price': '//*[@id="prcIsum"]/text()', 'name': '//*[@id="itemTitle"]/text()'}
{"url": ' http://b.html', 'price': '//span/strong/text()', 'name': '//h1/text()'}
{"url": ' http://c.html', 'price': '', 'name': '//*[@id="product-desc"]/span/text()'
'}
```

很好。现在编辑generic/spiders/fromcsv.py爬虫。我们使用.csv文件中的URL，并且不希望遇到域名限制的情况。因此第一件事是移除start\_URL和allowed\_domains。然后再读.csv文件。

因为从文件中读取的URL是我们事先不了解的，所以使用一个start\_requests()方法。对于每一行，我们都会创建Request。我们还要从request,meta的csv存储字段名和XPath，以便在我们的parse()函数中使用。然后，我们使用Item和ItemLoader填充Item的字段。下面是所有代码：

```
import csv
import scrapy
from scrapy.http import Request
from scrapy.loader import ItemLoader
from scrapy.item import Item, Field
```

```

class FromcsvSpider(scrapy.Spider):
    name = "fromcsv"
def start_requests(self):
    with open("todo.csv", "rU") as f:
        reader = csv.DictReader(f)
        for line in reader:
            request = Request(line.pop('url'))
            request.meta['fields'] = line
            yield request
def parse(self, response):
    item = Item()
    l = ItemLoader(item=item, response=response)
    for name, xpath in response.meta['fields'].iteritems():
        if xpath:
            item.fields[name] = Field()
            l.add_xpath(name, xpath)
    return l.load_item()

```

运行爬虫，输出文件保存为csv：

```

$ scrapy crawl fromcsv -o out.csv
INFO: Scrapy 0.0.3 started (bot: generic)
...
DEBUG: Scraped from <200 a.html>
{'name': [u'My item'], 'price': [u'128']}
DEBUG: Scraped from <200 b.html>
{'name': [u'Getting interesting'], 'price': [u'300']}
DEBUG: Scraped from <200 c.html>
{'name': [u'Buy this now']}
...
INFO: Spider closed (finished)
$ cat out.csv
price,name
128,My item
300,Getting interesting
,Buy this now

```

有几点要注意。项目中没有定义一个整个项目的Items，我们必须手动向ItemLoader提供一个：

```

item = Item()
l = ItemLoader(item=item, response=response)

```

我们还用Item的fields成员变量添加了动态字段。添加一个新的动态字段，并用ItemLoader填充，使用下面的方法：

```
item.fields[name] = Field()  
l.add_xpath(name, xpath)
```

最后让代码再漂亮些。硬编码todo.csv不是很好。Scrapy提供了一种便捷的向爬虫传递参数的方法。如果我们使用-a参数，例如，-a variable=value，就创建了一个爬虫项，可以用self.variable取回。为了检查变量（没有的话，提供一个默认变量），我们使用Python的getattr()方法：getattr(self, 'variable', 'default')。总之，原来的with open...替换为：

```
with open(getattr(self, "file", "todo.csv"), "rU") as f:
```

现在，todo.csv是默认文件，除非使用参数-a，用一个源文件覆盖它。如果还有一个文件，another\_todo.csv，我们可以运行：

```
$ scrapy crawl fromcsv -a file=another_todo.csv -o out.csv
```

## 总结

在本章中，我们进一步学习了Scrapy爬虫。我们使用FormRequest进行登录，用请求/响应中的meta传递变量，使用了相关的XPath表达式和Selectors，使用.csv文件作为数据源等等。

接下来在第6章学习在Scrapinghub云部署爬虫，在第7章学习关于Scrapy的设置。

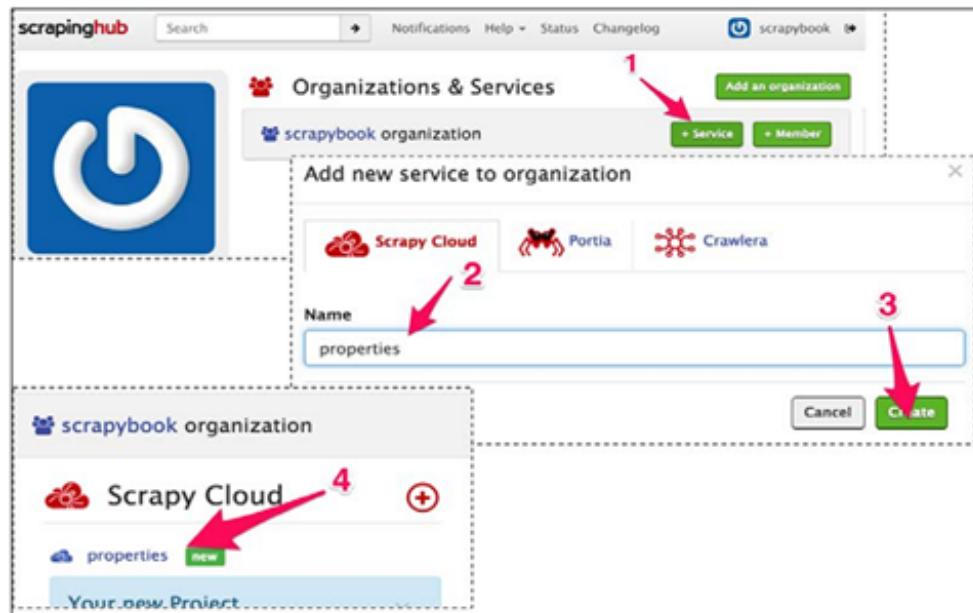
# 第6章 Scrapinghub部署

前面几章中，我们学习了如何编写爬虫。编写好爬虫之后，我们有两个选择。如果是做单次抓取，让爬虫在开发机上运行一段时间就行了。或者，我们往往需要周期性的进行抓取。我们可以用Amazon、RackSpace等服务商的云主机，但这需要一些设置、配置和维护。这时候就需要Scrapinghub了。

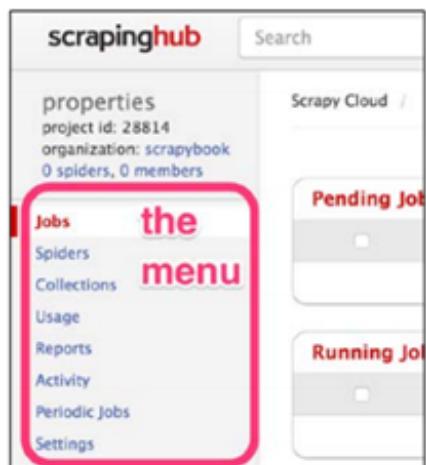
Scrapinghub是Scrapy高级开发者托管在Amazon上面的云架构。这是一个付费服务，但提供免费使用。如果想短时间内让爬虫运行在专业、有维护的平台上，本章内容很适合你。

# 注册、登录、创建项目

第一步是在`http://scrapinghub.com`注册一个账户，只需电子邮件地址和密码。点击确认邮件的链接之后，就登录了。首先看到的是工作台，目前还没有任何项目，点击+Service按钮（1）创建一个：



将项目命名为properties（2），点击Create按钮（3）。然后点击链接new（4）打开这个项目。



项目的工作台是最重要的界面。左侧栏中可以看到一些标签。Jobs和Spiders提供运行和爬虫的信息。Periodic Jobs可以制定周期抓取。其它四项，现在对我们不重要。

The screenshot shows the Scrapinghub project settings interface for the 'properties' project. On the left sidebar, under the 'Settings' section, the 'Scrapy Deploy' option is highlighted with a red arrow and labeled '1'. Below it, the 'Scrapy Deploy' link is also highlighted with a red arrow and labeled '2'. In the main content area, there is a code block with a red arrow pointing to the 'url' line, which is highlighted with a red box and labeled '3. Copy this'. The URL shown is `https://dash.scrapinghub.com/api/scrapyd/`.

进入Settings (1)。和许多网站的设置不同，Scrapinghub提供许多非常有用的设置项。

现在，先关注下Scrapy Deploy (2)。

## 部署爬虫并制定计划

我们从开发机直接部署。将Scrapy Deploy页上的url复制到我们项目的scrapy.cfg中，替换原有的 [deploy]部分。不必设置密码。我们用第4章中的properties爬虫作例子。我们使用这个爬虫的原因是，目标数据可以从网页访问，访问的方式和第4章中一样。开始之前，我们先恢复原有的 settings.py，去除和Appery.io pipeline有关的内容：

提示：代码位于目录ch06。这个例子在ch06/properties中。

```
$ pwd  
/root/book/ch06/properties  
$ ls  
properties  scrapy.cfg  
$ cat scrapy.cfg  
...  
[settings]  
default = properties.settings  
# Project: properties  
[deploy]  
url = http://dash.scrapinghub.com/api/scrapyd/  
username = 180128bc7a0.....50e8290dbf3b0  
password =  
project = 28814
```

为了部署爬虫，我们使用Scrapinghub提供的shub工具，可以用pip install shub安装。我们的开发机中已经有了。我们shub login登录Scrapinghub，如下所示：

```
$ shub login  
Insert your Scrapinghub API key : 180128bc7a0.....50e8290dbf3b0  
Success.
```

我们已经在scrapy.cfg文件中复制了API key，我们还可以点击Scrapinghub右上角的用户名找到API key。弄好API key之后，就可以使用shub deploy部署爬虫了：

```
$ shub deploy  
Packing version 1449092838  
Deploying to project "28814"in {"status": "ok", "project": 28814,  
"version": "1449092838", "spiders": 1}  
Run your spiders at: https://dash.scrapinghub.com/p/28814/
```

Scrapy打包了所有爬虫文件，并上传到了Scrapinghub。我们可以看到两个新目录和一个文件，可以选择删除或不删除。

```
$ ls  
build project.egg-info properties scrapy.cfgsetup.py  
$ rm -rf build project.egg-info setup.py
```

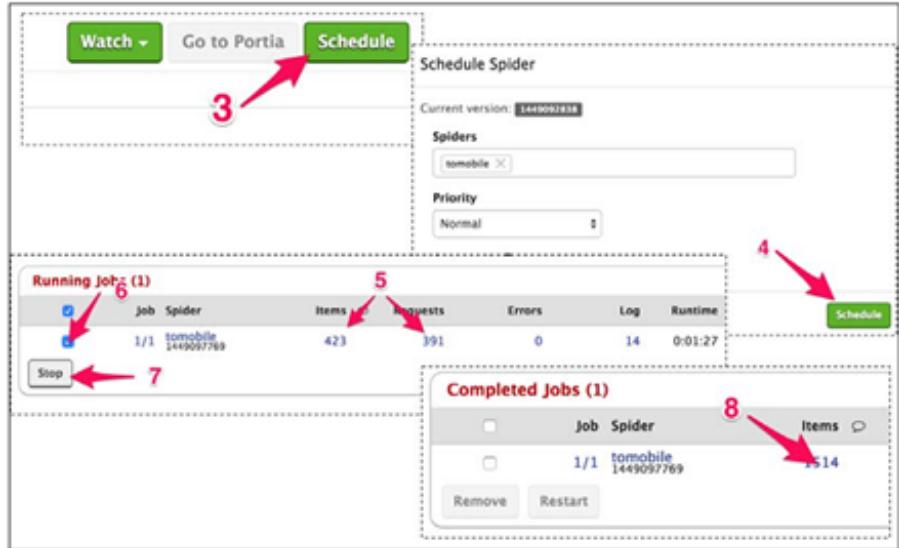
现在，如果我们在Scrapinghub点击Spiders栏（1），我们可以看到上传的tomobile爬虫：

The screenshot shows the Scrapinghub dashboard interface. On the left, there is a sidebar with links: 'Jobs' (highlighted with a red arrow), 'Spiders' (highlighted with a red arrow and labeled '1'), 'Collections', 'Usage', 'Reports', 'Activity', 'Periodic Jobs', and 'Cronjobs'. The main content area is titled 'Scrapy Cloud / properties / Spiders'. It has a search bar and a checkbox for 'Archived spiders'. Below is a table with one row:

Spider	Last Run
tomobile	--

At the bottom, there is a pagination control '10 Spiders per page'.

如果我们点击它（2），可以转到爬虫的工作台。里面的信息很多，但我们要做的是点击右上角的Schedule按钮（3），在弹出的界面中再点击Schedule（4）。



几秒钟之后，Running Jobs栏会出现新的一行，再过一会儿，Requests和Items的数量开始增加。

**提示：**你或许不会限制抓取速度。Scrapinghub使用算法估算在不被封的情况下，你每秒的最大请求数。

运行一段时间后，勾选这个任务（6），点击Stop（7）。

几秒之后，可以在Completed Jobs看到抓取结束。要查看抓取文件，可以点击文件数（8）。

## 访问文件

来到任务的工作台。这里，可以查看文件（9），确认它们是否合格。我们还可以用上面的条件过滤结果。当我们向下翻动时，更多的文件被加载进来。

The screenshot shows the 'Items' page for a specific job. At the top, there's a breadcrumb navigation: 'Scrapy Cloud / properties / Spiders / tomobile / Job 1'. Below it are buttons for 'Job', 'Items (7999)', 'Requests (1620)', 'Log (22)', and 'Stats'. A red arrow labeled '11' points to the breadcrumb. Another red arrow labeled '10' points to the 'Items' button. On the right, there's a dropdown menu with options like 'Get as CSV', 'JSON', 'JSON Lines', 'XML', 'Sample', 'Random', and 'Latest'. A red arrow labeled '12' points to the 'Get as CSV' option. A red arrow labeled '13' points to the dropdown menu itself. A red arrow labeled '9' points to the table below, which lists item details such as description, price, url, address, date, image\_uris, project, and server.

如果有错的话，我们可以在Items的上方找到有用的关于Requests和Log的信息（10）。用上方的面包屑路径（11）可以返回爬虫或项目主页。当然，可以点击左上的Items按钮（12）下载文件，选择合适的选项（13），保存格式可以是CSV、JSON和JSON Lines。

另一种访问文件的方法是通过Scrapinghub的Items API。我们要做的是查看任务页或文件页的

URL。应该看起来和下面很像：

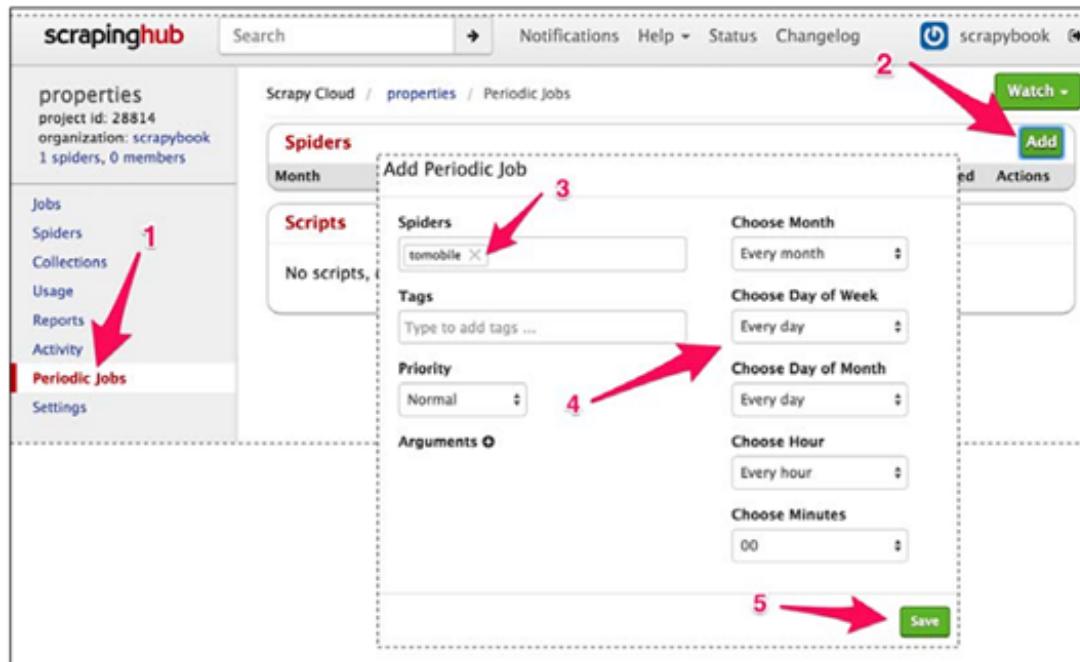
<https://dash.scrapinghub.com/p/28814/job/1/1/>

在这个URL中，28814是项目编号（scrapy.cfg中也设置了它），第一个1是爬虫“tomobile”的ID编号，第二个1是任务编号。按顺序使用这三个数字，我们可以在控制台中用curl取回文件，请发送到<https://storage.scrapinghub.com/items///>，并使用用户名/API key验证，如下所示：

```
$ curl -u 180128bc7a0.....50e8290dbf3b0: https://storage.scrapinghub.com/items/28814/1/1
{
  "_type": "PropertiesItem",
  "description": [
    "same\r\nsmoking\r\nnr..."
  ],
  "_type": "PropertiesItem",
  "description": [
    "british bit keep eve..."
  ],
  ...
}
```

如果询问密码的话，可以不填。用程序取回文件的话，可以使用Scrapinghub当做数据存储后端。存储的时间取决于订阅套餐的时间（免费试用是七天）。

## 制定周期抓取



只需要点击Periodic Jobs栏（1），点击Add（2），设定爬虫（3），调整抓取频率（4），最后点击Save（5）。

## 总结

本章中，我们首次接触了将Scrapy项目部署到Scrapinghub。定时抓取数千条信息，并可以用API方便浏览和提取。后面的章节中，我们继续学习设置一个类似Scrapinghub的小型服务器。下一章先学习配置和管理。

# 第7章 配置和管理

我们已经学过了用Scrapy写一个抓取网络信息的简单爬虫是多么容易。通过进行设置，Scrapy还有许多用途和功能。对于许多软件框架，用设置调节系统的运行，很让人头痛。对于Scrapy，设置是最基础的知识，除了调节和配置，它还可以扩展框架的功能。这里只是补充官方Scrapy文档，让你可以尽快对设置有所了解，并找到能对你有用的东西。在做出修改时，还请查阅文档。

## 使用Scrapy设置

在Scrapy的设置中，你可以按照五个等级进行设置。第一级是默认设置，你不必进行修改，但是scrapy/settings/default\_settings.py文件还是值得一读的。默认设置可以在命令级进行优化。一般来讲，除非你要插入自定义命令，否则不必修改。更经常的，我们只是修改自己项目的settings.py文件。这些设置只对当前项目管用。这么做很方便，因为当我们把项目部署到云主机时，可以连带设置文件一起打包，并且因为它是文件，可以用文字编辑器进行编辑。下一级是每个爬虫的设置。通过在爬虫中使用custom\_settings属性，我们可以自定义每个爬虫的设置。例如，这可以让我们打开或关闭某个特定蜘蛛的Pipelines。最后，要做最后的修改时，我们可以在命令行中使用-s参数。我们做过这样的设置，例如-s CLOSESPIDER\_PAGECOUNT=3，这可以限制爬虫的抓取范围。在这一级，我们可以设置API、密码等等。不要在settings.py文件中保存这些设置，因为不想让它们在公共仓库中失效。

这一章，我们会学习一些非常重要且常用的设置。在任意项目中输入以下命令，可以了解设置都有多少类型：

```
$ scrapy settings --get CONCURRENT_REQUESTS  
16
```

你得到的是默认值。修改这个项目的settings.py文件的CONCURRENT\_REQUESTS的值，比如，14。上面命令行的结果也会变为14，别忘了将设置改回去。在命令行中设置参数的话：

```
$ scrapy settings --get CONCURRENT_REQUESTS -s CONCURRENT_REQUESTS=19  
19
```

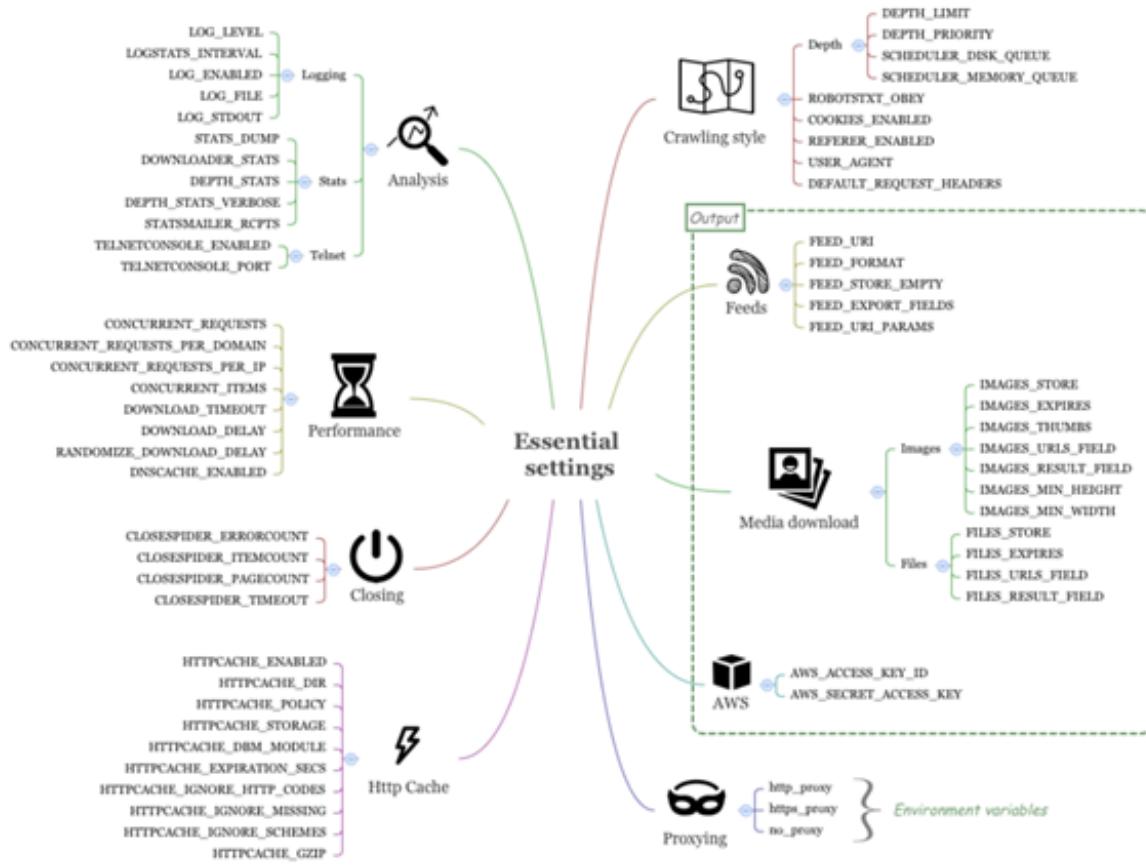
这个结果暗示scrapy crawl和scrapy settings都是命令。每个命令都使用这样的方法加载设置。再举一个例子：

```
$ scrapy shell -s CONCURRENT_REQUESTS=19  
>>> settings.getint('CONCURRENT_REQUESTS')  
19
```

当你想确认设置文件中的值时，你就可以才用以上几种方法。下面详细学习Scrapy的设置。

## 基本设置

Scrapy的设置太多，将其分类很有必要。我们从下图的基本设置开始，它可以让你明白重要的系统特性，你可能会频繁使用。



## 分析

通过这些设置，可以调节Scrapy的性能、调试信息的日志、统计、远程登录设备。

## 日志

Scrapy有不同的日志等级：DEBUG（最低），INFO，WARNING，ERROR，和CRITICAL（最高）。除此之外，还有一个SILENT级，没有日志输出。Scrapy的有用扩展之一是Log Stats，它

可以打印出每分钟抓取的文件数和页数。LOGSTATS\_INTERVAL设置日志频率，默认值是60秒。这个间隔偏长。我习惯于将其设置为5秒，因为许多运行都很短。LOG\_FILE设置将日志写入文件。除非进行设定，输出会一直持续到发生标准错误，将LOG\_ENABLED设定为False，就不会这样了。最后，通过设定LOG\_STDOUT为True，你可以让Scrapy在日志中记录所有的输出（比如print）。

## 统计

STATS\_DUMP是默认开启的，当爬虫运行完毕时，它把统计收集器（Stats Collector）中的值转移到日志。设定DOWNLOADER\_STATS，可以决定是否记录统计信息。通过DEPTH\_STATS，可以设定是否记录网站抓取深度的信息。若要记录更详细的深度信息，将DEPTH\_STATS\_VERBOSE设定为True。STATSMAILER\_RCPTS是一个当爬虫结束时，发送email的列表。你不用经常设置它，但有时调试时会用到它。

## 远程登录

Scrapy包括一个内建的远程登录控制台，你可以在上面用Python控制Scrapy。

TELNETCONSOLE\_ENABLED是默认开启的，TELNETCONSOLE\_PORT决定连接端口。在发生冲突时，可以对其修改。

# 案例1——使用远程登录

有时，你想查看Scrapy运行时的内部状态。让我们来看看如何用远程登录来做：

笔记：本章代码位于ch07。这个例子位于ch07/properties文件夹中。

```
$ pwd  
/root/book/ch07/properties  
$ ls  
properties  scrapy.cfg  
Start a crawl as follows:  
$ scrapy crawl fast  
...  
[scrapy] DEBUG: Telnet console listening on 127.0.0.1:6023:6023
```

这段信息是说远程登录被激活，监听端口是6023。然后在另一台电脑，使用远程登录的命令连接：

```
$ telnet localhost 6023  
>>>
```

现在，这台终端会给你一个在Scrapy中的Python控制台。你可以查看某些组件，例如用`engine`变量查看引擎，可以用`est()`进行快速查看：

```
>>> est()
Execution engine status
time() - engine.start_time : 5.73892092705
engine.has_capacity() : False
len(engine.downloader.active) : 8
...
len(engine.slot.inprogress) : 10
...
len(engine.scrape.slot.active) : 2
```

我们在第10章中会继续学习里面的参数。接着输入以下命令：

```
>>> import time
>>> time.sleep(1) # Don't do this!
```

你会注意到，另一台电脑有一个短暂停。你还可以进行暂停、继续、停止爬虫。使用远程机器时，使用远程登录的功能非常有用：

```
>>> engine.pause()
>>> engine.unpause()
>>> engine.stop()
Connection closed by foreign host.
```

## 性能

第10章会详细介绍这些设置，这里只是一个概括。性能设定可以让你根据具体的工作调节爬虫的性能。`CONCURRENT_REQUESTS`设置了并发请求的最大数。这是为了当你抓取很多不同的网站（域名/IPs）时，保护你的服务器性能。不是这样的话，你会发现`CONCURRENT_REQUESTS_PER_DOMAIN`和`CONCURRENT_REQUESTS_PER_IP`更多是限制性的。这两项分别通过限制每一个域名或IP地址的并发请求数，保护远程服务器。如果`CONCURRENT_REQUESTS_PER_IP`是非零的，`CONCURRENT_REQUESTS_PER_DOMAIN`则被忽略。这些设置不是按照每秒。如果`CONCURRENT_REQUESTS = 16`，请求平均消耗四分之一秒，最大极限则为每秒 $16/0.25 = 64$ 次请求。`CONCURRENT_ITEMS`设定每次请求并发处理的最大文件数。你可能会觉得这个设置没什么用，因为每个页面通常只有一个抓取项。它的默认值

是100。如果降低到，例如10或1，你可能会觉得性能提升了，取决于每次请求抓取多少项和pipelines的复杂度。你还会注意到，当这个值是关于每次请求的，如果CONCURRENT\_REQUESTS = 16, CONCURRENT\_ITEMS = 100意味每秒有1600个文件同时要写入数据库。我一般把这个值设的比较小。

对于下载，DOWNLOADS\_TIMEOUT决定了取消请求前，下载器的等待时间。默认是180秒，这个时间太长，并发请求是16时，每秒的下载数是5页。我建议设为10秒。默认情况下，各个下载间的间隔是0，以提高抓取速度。你可以设置DOWNLOADS\_DELAY改变下载速度。有的网站会测量请求频率以判定是否是机器人行为。设定DOWNLOADS\_DELAY的同时，还会有±50%的随机延迟。你可以设定RANDOMIZE\_DOWNLOAD\_DELAY为False。

最后，若要使用更快的DNS查找，可以设定DNSCACHE\_ENABLED打开内存DNS缓存。

### 提早结束抓取

Scrapy的CloseSpider扩展可以在条件达成时，自动结束抓取。你可以用CLOSESPIDER\_TIMEOUT(in seconds), CLOSESPIDER\_ITEMCOUNT, CLOSESPIDER\_PAGECOUNT, 和CLOSESPIDER\_ERRORCOUNT分别设置在一段时间、抓取一定数量的文件、发出一定数量请求、发生一定数量错误时，提前关闭爬虫。你会在运行爬虫时频繁地做出这类设置：

```
$ scrapy crawl fast -s CLOSESPIDER_ITEMCOUNT=10
$ scrapy crawl fast -s CLOSESPIDER_PAGECOUNT=10
$ scrapy crawl fast -s CLOSESPIDER_TIMEOUT=10
```

## HTTP缓存和脱机工作

Scrapy的HttpCacheMiddleware中间件（默认关闭）提供了一个低级的HTTP请求响应缓存。如果打开的话，缓存会存储每次请求和对应的响应。通过设定HTTPCACHE\_POLICY为scrapy.contrib.httpcache.RFC2616Policy，我们可以使用一个更为复杂的、按照RFC2616遵循网站提示的缓存策略。打开这项功能，设定HTTPCACHE\_ENABLED为True, HTTPCACHE\_DIR指向一个磁盘路径（使用相对路径的话，会存在当前文件夹内）。

你可以为缓存文件指定数据库后端，通过设定HTTPCACHE\_STORAGE为scrapy.contrib.httpcache.DbmCacheStorage，还可以选择调整HTTPCACHE\_DBM\_MODULE。（默认为anydbm）还有其它微调缓存的设置，但按照默认设置就可以了。

## 案例2——用缓存离线工作

运行以下代码：

```
$ scrapy crawl fast -s LOG_LEVEL=INFO -s CLOSESPIDER_ITEMCOUNT=5000
```

一分钟之后才结束。如果你无法联网，就无法进行任何抓取。用下面的代码再次进行抓取：

```
$ scrapy crawl fast -s LOG_LEVEL=INFO -s CLOSESPIDER_ITEMCOUNT=5000 -s HTTPCACHE_ENABLED=1
...
INFO: Enabled downloader middlewares:...*HttpCacheMiddleware*
```

你会看到启用了HttpCacheMiddleware，如果你查看当前目录，会发现一个隐藏文件夹，如下所示：

```
$ tree .scrapy | head
.scrapy
└── httpcache
    └── easy
        └── 00
            └── 002054968919f13763a7292c1907caf06d5a4810
                ├── meta
                ├── pickled_meta
                ├── request_body
                ├── request_headers
                └── response_body
...
...
```

当你再次运行不能联网的爬虫时，抓取稍少的文件，你会发现运行变快了：

```
$ scrapy crawl fast -s LOG_LEVEL=INFO -s CLOSESPIDER_ITEMCOUNT=4500 -s
HTTPCACHE_ENABLED=1
```

抓取稍少的文件，是因为使用CLOSESPIDER\_ITEMCOUNT结束爬虫时，爬虫实际上会多抓取几页，我们不想抓取不在缓存中的内容。清理缓存的话，只需删除缓存目录：

```
$ rm -rf .scrapy
```

# 抓取方式

Scrapy允许你设置从哪一页开始爬。设置DEPTH\_LIMIT，可以设置最大深度，0代表没有限制。根据深度，通过DEPTH\_PRIORITY，可以给请求设置优先级。将其设为正值，可以让你实现广度优先抓取，并在LIFO和FIFO间切换：

```
DEPTH_PRIORITY = 1
SCHEDULER_DISK_QUEUE = 'scrapy.squeue.PickleFifoDiskQueue'
SCHEDULER_MEMORY_QUEUE = 'scrapy.squeue.FifoMemoryQueue'
```

这个功能十分有用，例如，当你抓取一个新闻网站，先抓取离首页近的最近的新闻，然后再是其它页面。默认的Scrapy方式是顺着第一条新闻抓取到最深，然后再进行下一条。广度优先可以先抓取层级最高的新闻，再往深抓取，当设定DEPTH\_LIMIT为3时，就可以让你快速查看最近的新闻。

有的网站在根目录中用一个网络标准文件robots.txt规定了爬虫的规则。当设定ROBOTSTXT\_OBEY为True时，Scrapy会参考这个文件。设定为True之后，记得调试的时候碰到意外的错误时，可能是这个原因。

CookiesMiddleware负责所有cookie相关的操作，开启session跟踪的话，可以实现登录。如果你想进行秘密抓取，可以设置COOKIES\_ENABLED为False。使cookies无效减少了带宽，一定程度上可以加快抓取。相似的，REFERER\_ENABLED默认是True，可使RefererMiddleware生效，用它填充Referer headers。你可以用DEFAULT\_REQUEST\_HEADERS自定义headers。你会发现当有些奇怪的网站要求特定的请求头时，这个特别有用。最后，自动生成的settings.py文件建议我们设定USER\_AGENT。默认也可以，但我们应该修改它，以便网站所有者可以联系我们。

## Feeds

Feeds可以让你导出用Scrapy抓取的数据到本地或到服务器。存储路径取决于FEED\_URI.FEED\_URI，其中可能包括参数。例如scrapy crawl fast -o "%(name)s\_%(time)s.jl"，可以自动将时间和名字填入到输出文件。如果你需要你个自定义参数，例如%(foo)s，feed输出器希望在爬虫中提供一个叫做foo的属性。数据的存储，例如S3、FTP或本地，也是在URI中定义。例如，FEED\_URI='s3://mybucket/file.json'可以使用你的Amazon证书（AWS\_ACCESS\_KEY\_ID和AWS\_SECRET\_ACCESS\_KEY），将你的文件存储到Amazon S3。存储的格式，JSON、JSON Lines、CSV和XML，取决于FEED\_FORMAT。如果没有指定的话，Scrapy会根据FEED\_URI的后缀猜测。你可以选择输出为空，通过设定FEED\_STORE\_EMPTY为True。你还可以选择输出指定字段，通过设定FEED\_EXPORT\_FIELDS。这对.csv文件特别有用，可以固定header的列数。最后FEED\_URI\_PARAMS用于定义一个函数，对传递给FEED\_URI的参数进行后

处理。

## 下载媒体文件

Scrapy可以用Image Pipeline下载媒体文件，它还可以将图片转换成不同的格式、生成面包屑路径、或根据图片大小进行过滤。

IMAGES\_STORE设置了图片存储的路径（选用相对路径的话，会存储在项目的根目录）。每个图片的URL存在各自的image\_URL字段（它可以被IMAGES\_URL\_FIELD设置覆盖），下载下来的图片的文件名会存在一个新的image字段（它可以被IMAGES\_RESULT\_FIELD设置覆盖）。你可以通过IMAGES\_MIN\_WIDTH和IMAGES\_MIN\_HEIGHT筛选出小图片。IMAGES\_EXPIRES可以决定图片在缓存中存储的天数。IMAGES\_THUMBS可以设置一个或多个缩略图，还可以设置缩略图的大小。例如，你可以让Scrapy生成一个图标大小的缩略图或为每个图片生成一个中等的缩略图。

## 其它媒体文件

你可以使用Files Pipelines下载其它媒体文件。与图片相同FILES\_STORE决定了存储地址，FILES\_EXPIRES决定存储时间。FILES\_URL\_FIELD和FILES\_RESULT\_FIELD的作用与之前图片的相似。文件和图片的pipelines可以同时工作。

## 案例3——下载图片

为了使用图片功能，我们必须安装图片包，命令是pip install image。我们的开发机已经安装好了。要启动Image Pipeline，你需要编辑settings.py加入一些设置。首先在ITEM\_PIPELINES添加scrapy.pipelines.images.ImagesPipeline。然后，将IMAGES\_STORE设为相对路径"images"，通过设置IMAGES\_THUMBS，添加缩略图的描述，如下所示：

```
ITEM_PIPELINES = {
...
    'scrapy.pipelines.images.ImagesPipeline': 1,
}
IMAGES_STORE = 'images'
IMAGES_THUMBS = { 'small': (30, 30) }
```

我们已经为Item安排了image\_URL字段，然后如下运行：

```
$ scrapy crawl fast -s CLOSESPIDER_ITEMCOUNT=90
...
DEBUG: Scraped from <200 http://http://web:9312/.../index_00003.html/
property_000001.html>{
    'image_URL': [u'http://web:9312/images/i02.jpg'],
```

```
'images': [{'checksum': 'c5b29f4b223218e5b5beece79fe31510',
             ''path': 'full/705a3112e67...a1f.jpg',
             ''url': 'http://web:9312/images/i02.jpg'}],
```

```
...
$ tree images
images
└── full
    ├── 0abf072604df23b3be3ac51c9509999fa92ea311.jpg
    └── 1520131b5cc5f656bc683ddf5eab9b63e12c45b2.jpg
...
└── thumbs
    └── small
        ├── 0abf072604df23b3be3ac51c9509999fa92ea311.jpg
        └── 1520131b5cc5f656bc683ddf5eab9b63e12c45b2.jpg
...
```

我们看到图片成功下载下来，还生成了缩略图。Images文件夹中存储了jpg文件。缩略图的路径可以很容易推测出来。删掉图片，可以使用命令rm -rf images。

## 亚马逊网络服务

Scrapy内建支持亚马逊服务。你可以将AWS的access key存储到AWS\_ACCESS\_KEY\_ID，将secret key存到AWS\_SECRET\_ACCESS\_KEY。这两个设置默认都是空的。使用方法如下：

- 当你用开头是s3://（注意不是http://）下载URL时
- 当你用media pipelines在s3://路径存储文件或缩略图时
- 当你在s3://目录存储输出文件时，不要在**settings.py**中存储这些设置，以免有一天这个文件要公开。

### 使用代理和爬虫

Scrapy的HttpProxyMiddleware组件可以让你使用代理，它包括http\_proxy、https\_proxy和no\_proxy环境变量。代理功能默认是开启的。

## 案例4——使用代理和Crawlera的智慧代理

DynDNS提供了一个免费检查你的IP地址的服务。使用Scrapy shell，我们向checkip.dyndns.org发送一个请求，检查响应确定当前的IP地址：

```
$ scrapy shell http://checkip.dyndns.org
>>> response.body
'<html><head><title>Current IP Check</title></head><body>Current IP
```

```
Address: xxx.xxx.xxx.xxx</body></html>\r\n'
>>> exit()
```

要使用代理请求，退出shell，然后使用export命令设置一个新代理。你可以通过搜索HMA的公共代理列表 (<http://proxylist.hidemyass.com/>) 测试一个免费代理。例如，假设我们选择一个代理IP是10.10.1.1，端口是80（替换成你的），如下运行：

```
$ # First check if you already use a proxy
$ env | grep http_proxy
$ # We should have nothing. Now let's set a proxy
$ export http_proxy=http://10.10.1.1:80
```

再次运行Scrapy shell，你可以看到这次请求使用了不同的IP。代理很慢，有时还会失败，这时可以选择另一个IP。要关闭代理，可以退出Scrapy shell，并使用unset http\_proxy。

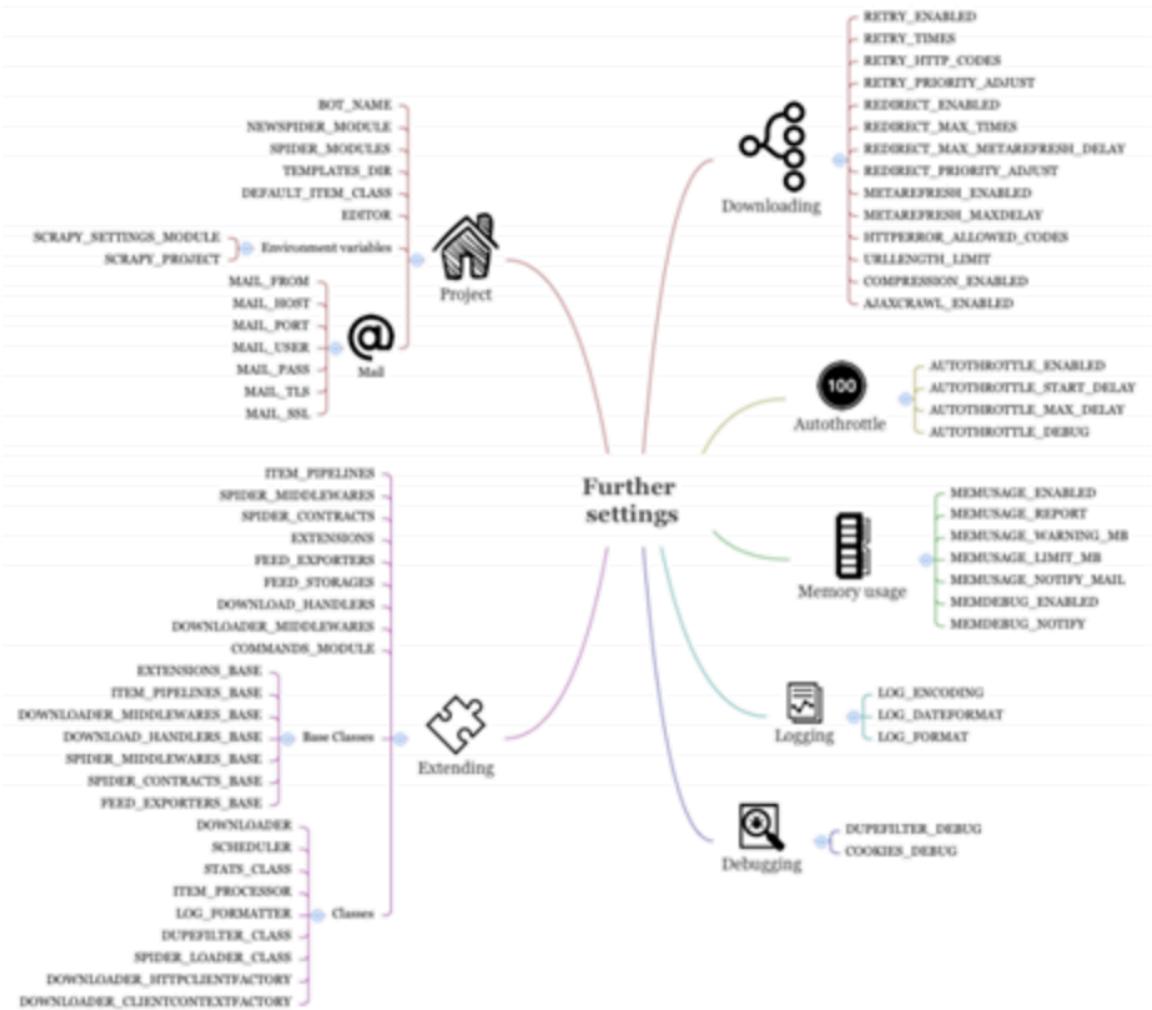
Crawlera是Scrapinghub的一个服务。除了使用一个大的IP池，它还能调整延迟并退出坏的请求，让连接变得快速稳定。这是爬虫工程师梦寐以求的产品。使用它，只需设置http\_proxy的环境变量为：

```
$ export http_proxy=myusername:mypassword@proxy.crawlera.com:8010
```

除了HTTP代理，还可以通过它给Scrapy设计的中间件使用Crawlera。

## 更多的设置

接下来看一些Scrapy不常用的设置和Scrapy的扩展设置，后者在后面的章节会详细介绍。



## 和项目相关的设定

这个小标题下，介绍和具体项目相关的设置，例如`BOT_NAME`、`SPIDER_MODULES`等等。最好在文档中查看一下，因为它们在某些具体情况下可以提高效率。但是通常来讲，Scrapy的`startproject`和`genspider`命令的默认设置已经是合理的了，所以就不必另行设置了。和邮件相关的设置，例如`MAIL_FROM`，可以让你配置`MailSender`类，它被用来发送统计数据（还可以查看`STATSMAILER_RCPTS`）和内存使用（还可以查看`MEMUSAGE_NOTIFY_MAIL`）。还有两个环境变量`SCRAPY_SETTINGS_MODULE`和`SCRAPY_PROJECT`，它们可以让你微调Scrapy项目的整合，例如，整合一个Django项目。`scrapy.cfg`还可以让你修改设置模块的名字。

## 扩展Scrapy设置

这些设定允许你扩展和修改Scrapy的几乎各个方面。最重要的就是`ITEM_PIPELINES`。它允许你在项目中使用Item Processing Pipelines。我们会在第9章中看到更多的例子。除了pipelines，还可以用多种方式扩展Scrapy，第8章总结了一些方式。`COMMANDS_MODULE`允许我们设置自定

义命令。例如，假设我们添加了一个properties/hi.py文件：

```
from scrapy.commands import ScrapyCommand
class Command(ScrapyCommand):
    default_settings = {'LOG_ENABLED': False}
    def run(self, args, opts):
        print("hello")
```

一旦我们在settings.py加入了COMMANDS\_MODULE='properties.hi'，就可以在Scrapy的help中运行hi查看。在命令行的default\_settings中定义的设置会与项目的设置合并，但是与settings.py文件的优先级比起来，它的优先级偏低。

Scrapy使用-\_BASE字典（例如，FEED\_EXPORTERS\_BASE）来存储不同扩展框架的默认值，然后我们可以在settings.py文件和命令行中设置non-\_BASE版本进行切换（例如，FEED\_EXPORTERS）。

最后，Scrapy使用设置，例如DOWNLOADER或SCEDULER，保管系统基本组件的包和类的名。我们可以继承默认的下载器（scrapy.core.downloader.Downloader），加载一些方法，在DOWNLOADER设置中自定义我们的类。这可以让开发者试验新特性、简化自动检测，但是只推荐专业人士这么做。

## 微调下载

RETRY\_、REDIRECT\_和METAREFRESH\_\*设置分别配置了Retry、Redirect、Meta-Refresh中间件。例如，REDIRECT\_PRIORITY设为2，意味着每次有重定向时，都会在没有重定向请求之后，预约一个新的请求。REDIRECT\_MAX\_TIMES设为20意味着，在20次重定向之后，下载器不会再进行重定向，并返回现有值。当你抓取一些有问题的网站时，知道这些设置是很有用的，但是默认设置在大多数情况下就能应付了。HTTPERROR\_ALLOWED\_CODES和URLLENGTH\_LIMIT也类似。

## 自动限定扩展设置

AUTOTHROTTLE\_\*设置可以自动限定扩展。看起来有用，但在实际中，我发现很难用它进行调节。它使用下载延迟，并根据加载和指向服务器，调节下载器的延迟。如果你不能确定DOWNLOAD\_DELAY（默认是0）的值，这个模块会派上用场。

## 内存使用扩展设置

`MEMUSAGE_*`设置可以配置内存使用扩展。当超出内存上限时，它会关闭爬虫。在共享环境中这会很有用，因为抓取过程要尽量小心。更多时候，你会将`MEMUSAGE_LIMIT_MB`设为0，将自动关闭爬虫的功能取消，只接收警告email。这个扩展只在类Unix平台有。

`MEMDEBUG_ENABLED`和`MEMDEBUG_NOTIFY`可以配置内存调试扩展，可以在爬虫关闭时实时打印出参考的个数。阅读用`trackref`调试内存泄漏的文档，更重要的，我建议抓取过程最好简短、分批次，并匹配服务器的能力。我认为，每批次最好一千个网页、不超过几分钟。

## 登录和调试

---

最后，还有一些登录和调试的设置。`LOG_ENCODING`, `LOG_DATEFORMAT`和`LOG_FORMAT`可以让你微调登录的方式，当你使用登录管理，比如Splunk、Logstash和Kibana时，你会觉得它很好用。`DUPEFILTER_DEBUG`和`COOKIES_DEBUG`可以帮助你调试相对复杂的状况，比如，当你的请求数比预期少，或丢失session时。

## 总结

---

通过阅读本章，你一定会赞叹比起以前手写的爬虫，Scrapy的功能更具深度和广度。如果你想微调或扩展Scrapy的功能，可以有大量的方法，见下面几章。

# 第8章 Scrapy编程

---

到目前为止，我们创建爬虫的目的是抓取数据，并提取信息。除了爬虫，scrapy可以让我们微调它的功能。例如，你会经常碰到以下状况：

你在同一个项目的爬虫间复制粘贴了很多代码。重复的代码更多是关于处理数据，而不是关于数据源。

你必须写脚本，好让Items复制入口或后处理数值。

你要在项目中架构中使用重复代码。例如，你要登录，并将文件传递到私有仓库，向数据库添加Items，或当爬虫结束时触发后处理操作。

你发现Scrapy有些方面不好用，你想在自己的项目中自定义Scrapy。

Scrapy的开发者设计的架构允许我们解决上述问题。我们会在本章后面查看Scrapy架构。现在，首先让我们来看Scrapy的引擎，Twisted。

# Scrapy是一个Twisted应用

---

Scrapy是一个用Twisted Python框架构建的抓取应用。Twisted很不寻常，因为它是事件驱动的，并且鼓励我们编写异步代码。完全弄懂需要一些时间，我们只学习和Scrapy相关的部分。我们还会在处理错误中学习。Scrapy在GitHub上的代码有更多的错误处理，我们会跳过它。

让我们从头开始。Twisted的不同之处在于它自身的结构。

**提示：**在任何时候，都不要让代码发生阻塞。

这个提示很重要。发生阻塞的代码包括：

- 访问文件、数据库或网络的代码
- 产生新进程并占用输出的代码，例如，运行命令行
- 执行系统级操作的代码，例如，在系统中排队

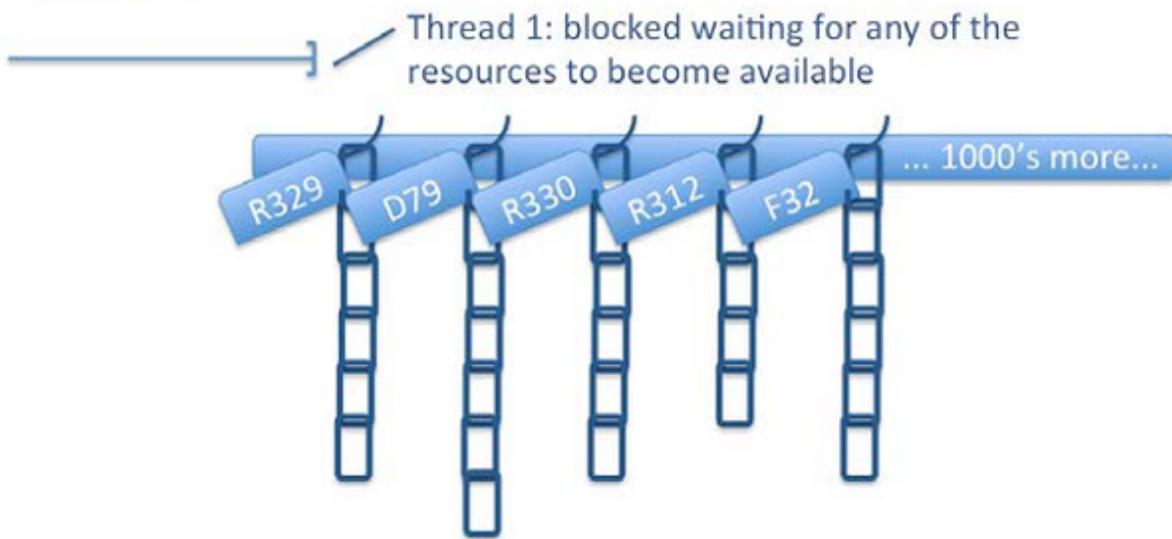
Twisted可以在不发生阻塞的情况下，执行以上操作。

为了展示不同，假设我们有一个典型的同步抓取应用。假设它有四个线程，在某个时刻，其中三个在等待响应而被阻塞，另一个在数据库中向Item文件写入而被阻塞。这时候，只能等待阻塞结束。阻塞结束时，又会有其它应用在几微妙之后占用了线程，又会发生阻塞。整体上，服务器并没有空闲，因为它上面运行着数十个程序、使用了数千个线程，因此，在微调之后，CPUs的利用率照样很高。

## Multithreading (4 threads):



## Twisted (1 thread):



Twisted/Scrapy的方法尽量使用一个线程。它使用操作系统的I/O多线路函数（见`select()`、`poll()`和`epoll()`）作为“挂架”。要发生阻塞时，例如，`result = i_block()`，Twisted会立即返回。然而，它不是返回实际值，而是返回一个钩子，例如`deferred = i_dont_block()`。我们可以在值变得可用时，例如`deferred.addCallback(process_result)`，将值返回到任何可以用到该值的进程。Twisted就是延迟操作链组成的。Twisted的单线程被称作Twisted事件反应器，它负责监视“挂架”是否有资源可用（例如，一个服务器响应了我们的请求）。当可用时，事件反应器会将排在最前面的延迟项执行，它执行完之后，会调用下一个。一些延迟项可能引发更多的I/O操作，它会将延迟链继续挂起来，让CPU执行别的操作。因为是单线程，我们不需要其它线程切换上下文和保存资源。换句话，使用这种非阻塞的结构，我们使用一个线程，就相当于有数千个线程。

OS开发者在数十年中不断优化线程操作。但是收效甚微。为一个复杂应用写出正确的多线程代码确实很难。当你搞明白延迟和回调，你会返现Twisted代码比线程代码简单多了。  
`inlineCallbacks`生成器可以让代码更简单，下面会继续介绍。

笔记：可能目前最成功的非阻塞I/O系统是Node.js，这主要因为从一开始Node.js就要求高性能和并发。每个Node.js只是用非阻塞的APIs。在Java中，Netty可能是最成功的NIO框架，例如Apache Storm和Spark。C++11的`std::future`和`std::promise`（与延迟项相似）可以用库，例如libevent或plain POSIX写异步代码。

# 延迟项和延迟链

延迟项是Twisted写出异步代码的最重要机制。Twisted APIs使用延迟项让我们定义事件发生时产生动作的顺序。

提示：本章代码位于ch08。这个例子位于ch08/deferreds.py file，你可以用./deferreds.py 0 运行。

你可以用Python控制台如下运行：

```
$ python
>>> from twisted.internet import defer
>>> # Experiment 1
>>> d = defer.Deferred()
>>> d.called
False
>>> d.callback(3)
>>> d.called
True
>>> d.result
3
```

我们看到，延迟项本质代表一个值。当我们触发d时（调用callback方法），延迟项的called状态变为True，result属性变为调用的值：

```
>>> # Experiment 2
>>> d = defer.Deferred()
>>> def foo(v):
...     print "foo called"
...     return v+1
...
>>> d.addCallback(foo)
<Deferred at 0x7f...>
>>> d.called
False
>>> d.callback(3)
foo called
>>> d.called
True
>>> d.result
4
```

延迟项的最强大之处是，当值确定时，可以在延迟链上添加新的项。在上面的例子中，我们使用foo()作为d的回调。当我们调用callback(3)时，函数foo()被调用并打印出信息。返回值作为d的最后结果：

```
>>> # Experiment 3
>>> def status(*ds):
...     return [(getattr(d, 'result', "N/A"), len(d.callbacks)) for d in
... ds]
>>> def b_callback(arg):
...     print "b_callback called with arg =", arg
...     return b
>>> def on_done(arg):
...     print "on_done called with arg =", arg
...     return arg
>>> # Experiment 3.a
>>> a = defer.Deferred()
>>> b = defer.Deferred()
```

这个例子演示了延迟项更复杂的情况。我们看到了一个正常的延迟项a，但它有两个调回。第一个是b\_callback()，返回的是b而不是a。第二个是，on\_done()打印函数。我们还有一个status()函数，它可以打印延迟项的状态。对于两个调回，刚建立时，有两个相同的状态[('N/A', 2), ('N/A', 0)]，意味着两个延迟项都没有被触发，第一个有两个调回，第二个没有调回。然后，如果我们先触发a，我们进入一个奇怪的状态 [(1, ('N/A', 1))，它显示a现在有一个值，这是一个延迟值（实际上就是b），它只有一个调回，因为b\_callback()已经被调回，只留下on\_done()。意料之外的是吗，现在b[(4, 0), (None, 0)]，这正是我们想要的：

```
>>> # Experiment 3.b
>>> a = defer.Deferred()
>>> b = defer.Deferred()
>>> a.addCallback(b_callback).addCallback(on_done)
>>> status(a, b)
[('N/A', 2), ('N/A', 0)]
>>> b.callback(4)
>>> status(a, b)
[('N/A', 2), (4, 0)]
>>> a.callback(3)
b_callback called with arg = 3
on_done called with arg = 4
>>> status(a, b)
[(4, 0), (None, 0)]
```

另一方面，在设a为3之前就触发b， b的状态变为 `[('N/A', 2), (4, 0)]`，然后当a被触发时，两个调用都会被调用，最后的状态和前一个例子一样。无论触发的顺序，结果都是一样的。两者的区别是，在第一种情况中，b的值被延迟更久，因为它是后触发的。而在第二种情况下，先触发b，然后它的值立即被使用。

这时，你应该可以理解什么是延迟项，它们是怎么构成链的和表达值得。我们用第四个例子说明触发取决于其它延迟项的数量，通过使用Twisted中的类`defer.DeferredList`:

```
>>> # Experiment 4
>>> deferreds = [defer.Deferred() for i in xrange(5)]
>>> join = defer.DeferredList(deferreds)
>>> join.addCallback(on_done)
>>> for i in xrange(4):
...     deferreds[i].callback(i)
>>> deferreds[4].callback(4)
on_done called with arg = [(True, 0), (True, 1), (True, 2),
                            (True, 3), (True, 4)]
```

我们看到`for`声明`on_done()`触发了五个中的四个，它们并没有被调用，直到所有延迟项都被触发，在最后的调用`deferreds[4].callback()`之后。`on_done()`的参数是一个元组表，每个元组对应一个延迟项，包含`True`是成功/`False`是失败，和延迟项的值。

## 理解Twisted和非阻塞I/O——Python的故事

现在我们已经有了一个大概的了解，现在让我给你讲一个Python的小故事。所有的角色都是虚构的，如有巧合纯属雷同：

```
# ~*~ Twisted - A Python tale ~*~
from time import sleep
# Hello, I'm a developer and I mainly setup Wordpress.
def install_wordpress(customer):
    # Our hosting company Threads Ltd. is bad. I start installation
    # and...
    print "Start installation for", customer
    # ...then wait till the installation finishes successfully. It is
    # boring and I'm spending most of my time waiting while consuming
    # resources (memory and some CPU cycles). It's because the process
    # is *blocking*.
    sleep(3)
    print "All done for", customer
# I do this all day long for our customers
def developer_day(customers):
```

```
for customer in customers:  
    install_wordpress(customer)  
developer_day(["Bill", "Elon", "Steve", "Mark"])
```

让我们运行它：

```
$ ./deferreds.py 1  
----- Running example 1 -----  
Start installation for Bill  
All done for Bill  
Start installation  
...  
* Elapsed time: 12.03 seconds
```

结果是顺序执行的。4名顾客，每人3秒，总和就是12秒。时间有些长，所以我们在第二个例子中，添加线程：

```
import threading  
# The company grew. We now have many customers and I can't handle  
the  
# workload. We are now 5 developers doing exactly the same thing.  
def developers_day(customers):  
    # But we now have to synchronize... a.k.a. bureaucracy  
    lock = threading.Lock()  
    #  
    def dev_day(id):  
        print "Goodmorning from developer", id  
        # Yuck - I hate locks...  
        lock.acquire()  
        while customers:  
            customer = customers.pop(0)  
            lock.release()  
            # My Python is less readable  
            install_wordpress(customer)  
            lock.acquire()  
        lock.release()  
        print "Bye from developer", id  
    # We go to work in the morning  
    devs = [threading.Thread(target=dev_day, args=(i,)) for i in  
range(5)]  
    [dev.start() for dev in devs]  
    # We leave for the evening  
    [dev.join() for dev in devs]
```

```
# We now get more done in the same time but our dev process got more
# complex. As we grew we spend more time managing queues than doing dev
# work. We even had occasional deadlocks when processes got extremely
# complex. The fact is that we are still mostly pressing buttons and
# waiting but now we also spend some time in meetings.
developers_day(["Customer %d" % i for i in xrange(15)])
```

如下运行：

```
$ ./deferreds.py 2
----- Running example 2 -----
Goodmorning from developer 0Goodmorning from developer
1Start installation forGoodmorning from developer 2
Goodmorning from developer 3Customer 0
...
from developerCustomer 13 3Bye from developer 2
* Elapsed time: 9.02 seconds
```

你用5名工人线程并行执行。15名顾客，每人3秒，单人处理要45秒，但是有5名工人的话，9秒就够了。代码有些复杂。不再关注于算法和逻辑，它只考虑并发。另外，输出结果变得混乱且可读性变差。把简单的多线程代码写的好看也十分困难，现在我们 Twisted 怎么来做：

```
# For years we thought this was all there was... We kept hiring more
# developers, more managers and buying servers. We were trying harder
# optimising processes and fire-fighting while getting mediocre
# performance in return. Till Luckily one day our hosting
# company decided to increase their fees and we decided to
# switch to Twisted Ltd.!
from twisted.internet import reactor
from twisted.internet import defer
from twisted.internet import task
# Twisted has a slightly different approach
def schedule_install(customer):
    # They are calling us back when a Wordpress installation completes.
    # They connected the caller recognition system with our CRM and
    # we know exactly what a call is about and what has to be done
    # next.
    #
    # We now design processes of what has to happen on certain events.
def schedule_install_wordpress():
    def on_done():
        print "Callback: Finished installation for", customer
    print "Scheduling: Installation for", customer
```

```

        return task.deferLater(reactor, 3, on_done)
#
def all_done(_):
    print "All done for", customer
#
# For each customer, we schedule these processes on the CRM
# and that
# is all our chief-Twisted developer has to do
d = schedule_install_wordpress()
d.addCallback(all_done)
#
return d
# Yes, we don't need many developers anymore or any synchronization.
# ~~ Super-powered Twisted developer ~~
def twisted_developer_day(customers):
    print "Goodmorning from Twisted developer"
    #
    # Here's what has to be done today
    work = [schedule_install(customer) for customer in customers]
    # Turn off the lights when done
    join = defer.DeferredList(work)
    join.addCallback(lambda _: reactor.stop())
    #
    print "Bye from Twisted developer!"

# Even his day is particularly short!
twisted_developer_day(["Customer %d" % i for i in xrange(15)])
# Reactor, our secretary uses the CRM and follows-up on events!
reactor.run()

```

让我们运行它：

```

$ ./deferreds.py 3
----- Running example 3 -----
Goodmorning from Twisted developer
Scheduling: Installation for Customer 0
....
Scheduling: Installation for Customer 14
Bye from Twisted developer!
Callback: Finished installation for Customer 0
All done for Customer 0
Callback: Finished installation for Customer 1
All done for Customer 1
...
All done for Customer 14
* Elapsed time: 3.18 seconds

```

我们没用线程就得到了十分漂亮的结果。我们并行处理了15名顾客，45秒的工作在3秒内完成。我们的方法是让阻塞的调用进行sleep()，而采用task.deferLater()和调用函数。在其它地方进行处理时，我们可以轻松送出应付15名顾客。

笔记：我之前提到在其它地方进行处理。这是作弊吗？不是。计算仍在CPUs中进行。与磁盘和网络操作比起来，如今的CPU运算非常快。CPUs接收发送数据或存储才是最花时间的。通过使用非阻塞I/O操作，我们为CPUs节省了这个时间。与task.deferLater()相似，当数据传输完毕时，触发再进行调用。

另一个重点是Goodmorning from Twisted developer和Bye from Twisted developer!消息。当运行代码时，它们立即就被打印出来。如果代码到达此处这么早，应用什么时候真正运行起来的呢？答案是Twisted应用全部都是在reactor.run()中运行的。当你调用某个方法时，你必须有每个可能要用到的延迟项（相当于前面的故事里，在CRM系统中设定步骤和过程）。你的reactor.run()监控事件并触发调回。

笔记：反应器的最主要规则是，只要是非阻塞操作就可以执行。

虽然没有线程了，调回函数还是有点不好看。看下面的例子：

```
# Twisted gave us utilities that make our code way more readable!
@defer.inlineCallbacks
def inline_install(customer):
    print "Scheduling: Installation for", customer
    yield task.deferLater(reactor, 3, lambda: None)
    print "Callback: Finished installation for", customer
    print "All done for", customer
def twisted_developer_day(customers):
    ... same as previously but using inline_install()
    instead of schedule_install()
twisted_developer_day(["Customer %d" % i for i in xrange(15)])
reactor.run()
```

运行如下：

```
$ ./deferreds.py 4
... exactly the same as before
```

这段代码的功能和之前的一样，但是好看很多。inlineCallbacks生成器用Python机制暂停和继续

`inline_install()`中的代码。`inline_install()`变成了一个延迟项，而后对每名顾客并行执行。每次yield时，暂停当前的`inline_install()`，被触发时再继续。

唯一的问题是，当我们不是有15名顾客，而是10000名时，这段代码会同时发起10000个进程（可以是HTTP请求、写入数据库等等）。这可能可以运行，或者会产生严重的问题。在大并发应用中，我们通常会限制并发数。在这个例子中。Scrapy使用了相似的机制，在`CONCURRENT_ITEMS`设置中限制并发数：

```
@defer.inlineCallbacks
def inline_install(customer):
    ... same as above
# The new "problem" is that we have to manage all this concurrency to
# avoid causing problems to others, but this is a nice problem to have.
def twisted_developer_day(customers):
    print "Goodmorning from Twisted developer"
    work = (inline_install(customer) for customer in customers)
    #
    # We use the Cooperator mechanism to make the secretary not
    # service more than 5 customers simultaneously.
    coop = task.Cooperator()
    join = defer.DeferredList([coop.coiterate(work) for i in xrange(5)])
    #
    join.addCallback(lambda _: reactor.stop())
    print "Bye from Twisted developer!"
twisted_developer_day(["Customer %d" % i for i in xrange(15)])
reactor.run()
# We are now more lean than ever, our customers happy, our hosting
# bills ridiculously low and our performance stellar.
# ~*~ THE END ~*~
```

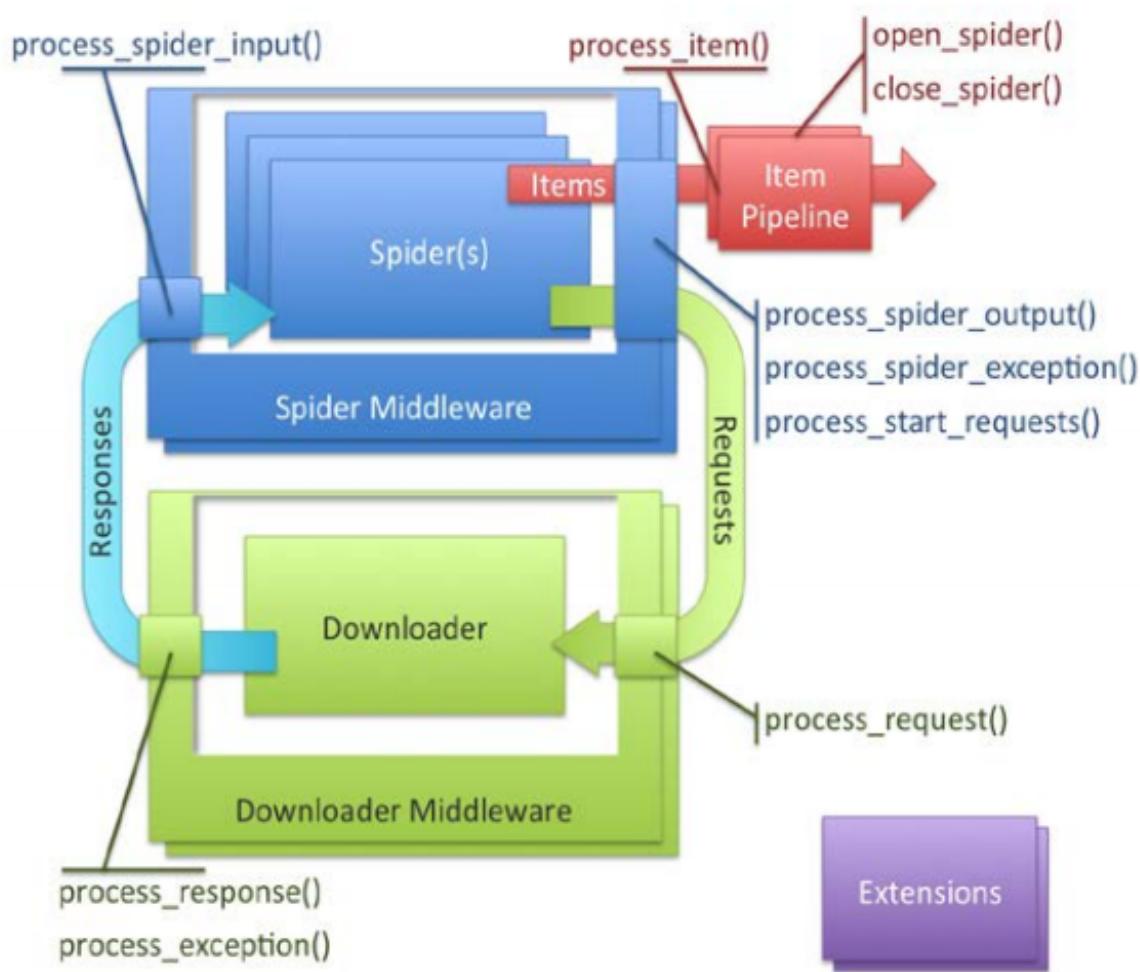
运行如下：

```
$ ./deferreds.py 5
----- Running example 5 -----
Goodmorning from Twisted developer
Bye from Twisted developer!
Scheduling: Installation for Customer 0
...
Callback: Finished installation for Customer 4
All done for Customer 4
Scheduling: Installation for Customer 5
...
Callback: Finished installation for Customer 14
All done for Customer 14
```

\* Elapsed time: 9.19 seconds

我们现在看到，一共有五个顾客的处理窗口。只有存在空窗口时，才能服务新顾客。因为处理每名顾客都是3秒，每批次可以处理5名顾客。最终，我们只用一个线程就达到了相同的性能，而且代码很简单。

## Scrapy架构概要



在架构操作的对象中有三个很眼熟，即Requests、Responses和Items。我们的爬虫位于架构的核心。爬虫产生请求、处理响应、生成Items和更多的请求。

爬虫生成的每个Item都按照Item Pipelines的`process_item()`方法指定的顺序，进行后处理。一般情况下，`process_item()`修改Items之后，将它们返回到随后的pipelines。特殊情况时（例如，有两个重复的无效数据），我们需要丢掉一个Item，我们要做的是加入DropItem例外。这时，后继的pipelines就不会接收Item了。如果我们还提供`open_spider()`和/或`close_spider()`，将会在爬虫开启和关闭时调用。这时可以进行初始化和清洗。Item Pipelines主要是用来处理问题和底层操作，例如清洗数据或将Items插入到数据库。你还会在项目之间重复使用它，尤其是涉及底层操作

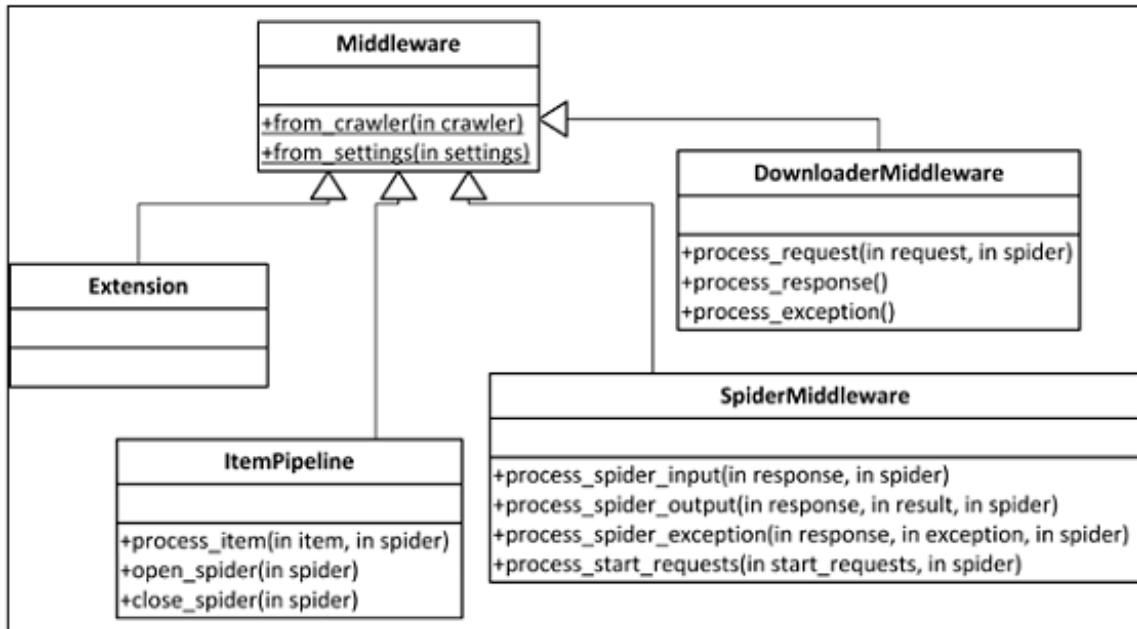
时。第4章中，我们使用的Appery.io pipeline就是用来做底层操作，用最少的配置将Items上传到Appery.io。

我们一般从爬虫发出请求，并得到返回的响应。Scrapy负责了cookies、认证、缓存等等，我们要做的只是偶尔进行设置。其中大部分都是靠下载器中间件完成的。下载器中间件通常很复杂，运用高深的方法处理请求响应间隔。你可以自定义下载器中间件，让请求处理可以按照自己的想法运行。好用的中间件可以在许多项目中重复使用，最好能在开发者社区中分享。如果你想查看默认的下载器中间件，可以在Scrapy的GitHub里的settings/default\_settings.py中，找到DOWNLOADER\_MIDDLEWARES\_BASE。

下载器是实际下载的引擎。你不必对其进行修改，除非你是Scrapy贡献者。

有时，你可能不得不写一个爬虫中间件。它们要在爬虫之后、其它下载器中间件之前处理请求，按相反的顺序处理响应。例如，利用下载器中间件，你想重写所有的URL使用HTTPS而不是HTTP，不管爬虫从网页抓取到什么。中间件专门为你的项目需求而设，并在爬虫间共享。下载器中间件和爬虫中间件的区别是，当下载器中间件有一个请求时，它必须回复一个单一的响应。另一方面，爬虫中间件不喜欢某个请求的话，可以丢掉这个请求，例如，忽略每一个输入请求，如果忽略对应用是有好处的话。你可以认为爬虫中间件是专为请求和响应的，item pipelines是专为Items的。爬虫中间件也可以接收Items，但通常不进行修改，因为用item pipeline修改更容易。如果你想查看默认的爬虫中间件，可以在Scrapy的GitHub里的settings/default\_settings.py中，找到SPIDER\_MIDDLEWARES\_BASE设置。

最后，来看扩展。扩展很常见，是仅次于Item Pipelines常见的。它们是抓取启动时加载的类，可以接入设置、爬虫、注册调用信号、并定义它们自己的信号。信号是一个基本的Scrapy API，它可以允许系统中有事情发生时，进行调用，例如，当一个Item被抓取、丢弃，或当一个爬虫打开时。有许多有用的预先定义的信号，我们后面会讲到。扩展是一个万金油，因为它可以让你写任何你能想到的功能，但不会提供任何实质性的帮助（例如Item Pipelines的process\_item()）。我们必须连接信号，并植入相关的功能。例如，抓取一定页数或Items之后关闭爬虫。如果你想查看默认的扩展，可以在Scrapy的GitHub里的settings/default\_settings.py中，找到EXTENSIONS\_BASE设置。



严格一点讲，Scrapy将所有的中间件当做类处理（由类MiddlewareManager管理），允许我们通过执行from\_crawler()或from\_settings()类方法，分别启用爬虫或Settings对象。因为可以从爬虫轻易获取设置（crawler.settings），from\_crawler()更流行一些。如果不需要Settings或Crawler，可以不引入它们。

下面的表可以帮助你确定，给定一个问题时，最佳的解决方案是什么：

问题	方案
和抓取的网站有关。	修改爬虫。
在特定域修改或存储 Items，可能在整个项目中使用。	写一个 Item Pipeline。
在特定域修改或丢弃请求或相应，可能在整个项目中使用。	写一个爬虫中间件。
执行请求响应，例如，支持自定义登录或特别处理 cookies。	写一个下载器中间件。
其它问题。	写一个扩展。

## 案例1——一个简单的pipeline

假设我们有一个含有若干蜘蛛的应用，它用通常的Python格式提供抓取日期。我们的数据库需要字符串格式以便索引它。我们不想编辑爬虫，因为它们有很多。我们该怎么做呢？一个很简单的pipelines可以后处理items和执行我们需要的转换。让我们看看它是如何做的：

```

from datetime import datetime
class TidyUp(object):
    def process_item(self, item, spider):
        item['date'] = map(datetime.isoformat, item['date'])

```

```
    return item
```

你可以看到，这就是一个简单的类加一个process\_item()方法。这就是我们需要的pipeline。我们可以再利用第3章中的爬虫，在tidyup.py文件中添加上述代码。

笔记：我们将pipeline的代码放在任何地方，但最好是在一个独立目录中。

我们现在编辑项目的settings.py文件，将ITEM\_PIPELINES设为：

```
ITEM_PIPELINES = {'properties.pipelines.tidyup.TidyUp': 100}
```

前面dict中的100设定了连接的pipelines的等级。如果另一个pipeline有更小的值，会优先将Items连接到这个pipeline。

提示：完整代码位于文件夹ch8/properties。

现在运行爬虫：

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=90
...
INFO: Enabled item pipelines: TidyUp
...
DEBUG: Scraped from <200 ...property_000060.html>
...
'date': ['2015-11-08T14:47:04.148968'],
```

和预想的一样，日期现在的格式是ISO字符串了。

## 信号

信号提供了一个可以给系统中发生的事件添加调用的机制，例如、当打开爬虫时，或是抓取一个Item时。你可以使用crawler.signals.connect()方法连接它们（例子见下章）。信号有11种，最好在实际使用中搞清它们。我建了一个项目，其中我创建了一个扩展，让它连接了每种可能的信号。我还建了一个Item Pipeline、一个下载器和一个爬虫中间件，它能记录每个使用过的方法。这个爬虫非常简单，只生成两个items，还有一个例外：

```
def parse(self, response):
    for i in range(2):
```

```
item = HooksasyncItem()
item['name'] = "Hello %d" % i
yield item
raise Exception("dead")
```

对于第二个Item，我通过Item Pipeline配置了一个DropItem例外。

提示：完整代码位于ch08/hooksasync。

使用这个项目，，我们可以更好地理解特定信号何时发送。看下面的命令行之间的注释（为了简洁起见，进行了省略）：

```
$ scrapy crawl test
... many lines ...
# First we get those two signals...
INFO: Extension, signals.spider_opened fired
INFO: Extension, signals.engine_started fired
# Then for each URL we get a request_scheduled signal
INFO: Extension, signals.request_scheduled fired
...# when download completes we get response_downloaded
INFO: Extension, signals.response_downloaded fired
INFO: DownloaderMiddlewareprocess_response called for example.com
# Work between response_downloaded and response_received
INFO: Extension, signals.response_received fired
INFO: SpiderMiddlewareprocess_spider_input called for example.com
# here our parse() method gets called... and then SpiderMiddleware used
INFO: SpiderMiddlewareprocess_spider_output called for example.com
# For every Item that goes through pipelines successfully...
INFO: Extension, signals.item_scraped fired
# For every Item that gets dropped using the DropItem exception...
INFO: Extension, signals.item_dropped fired
# If your spider throws something else...
INFO: Extension, signals.spider_error fired
# ... the above process repeats for each URL
# ... till we run out of them. then...
INFO: Extension, signals.spider_idle fired
# by hooking spider_idle you can schedule further Requests. If you don't
# the spider closes.
INFO: Closing spider (finished)
INFO: Extension, signals.spider_closed fired
# ... stats get printed
# and finally engine gets stopped.
INFO: Extension, signals.engine_stopped fired
```

你可能会觉得只有11的信号太少了，但每个默认的中间件都是用它们实现的，所以肯定足够了。请注意，除了spider\_idle、spider\_error、request\_scheduled、response\_received和response\_downloaded，你还可以用其它的信号返回的延迟项。

## 案例2——一个可以测量吞吐量和延迟的扩展

用pipelines测量吞吐量（每秒的文件数）和延迟（从计划到完成下载的时间）的变化十分有趣。

Scrapy已经有了一个可以测量吞吐量的扩展，Log Stats（见Scrapy的GitHub页scrapy/extensions/logstats.py），我们用它作为起点。为了测量延迟，我们连接信号request\_scheduled、response\_received和item\_scraped。我们给每个盖上时间戳，通过相减计算延迟，然后再计算平均延迟。通过观察信号的调用参数，我们发现了一些问题。item\_scraped只得到了Responses，request\_scheduled只得到了Requests，response\_received两个都取得了。我们不必破解就可以传递参数。每个Response都有一个Request成员，它指向回Request，更好的是，无论是否有重定向，它都有一个meta dict，并与原生的Requests的meta dict相同。所以可以将时间戳存在里面。

笔记：事实上，这不是我的主意。扩展AutoThrottle也使用了相同的机制（scrapy/extensions/throttle.py），它使用了request.meta.get ('download\_latency')。其中，通过计算器scrapy/core/downloader/webclient.py 求得download\_latency。提高写中间件速度的方法是，熟悉Scrapy默认中间件的代码。

以下是扩展的代码：

```
class Latencies(object):
    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler)
    def __init__(self, crawler):
        self.crawler = crawler
        self.interval = crawler.settings.getfloat('LATENCIES_INTERVAL')
        if not self.interval:
            raise NotConfigured
        cs = crawler.signals
        cs.connect(self._spider_opened, signal=signals.spider_opened)
        cs.connect(self._spider_closed, signal=signals.spider_closed)
        cs.connect(self._request_scheduled, signal=signals.request_
scheduled)
        cs.connect(self._response_received, signal=signals.response_-
received)
        cs.connect(self._item_scraped, signal=signals.item_scraped)
        self.latency, self.proc_latency, self.items = 0, 0, 0
    def _spider_opened(self, spider):
```

```

    self.task = task.LoopingCall(self._log, spider)
    self.task.start(self.interval)
def _spider_closed(self, spider, reason):
    if self.task.running:
        self.task.stop()
def _request_scheduled(self, request, spider):
    request.meta['schedule_time'] = time()
def _response_received(self, response, request, spider):
    request.meta['received_time'] = time()
def _item_scraped(self, item, response, spider):
    self.latency += time() - response.meta['schedule_time']
    self.proc_latency += time() - response.meta['received_time']
    self.items += 1
def _log(self, spider):
    irate = float(self.items) / self.interval
    latency = self.latency / self.items if self.items else 0
    proc_latency = self.proc_latency / self.items if self.items else 0
    spider.logger.info(("Scraped %d items at %.1f items/s, avg
latency: "
    "%.2f s and avg time in pipelines: %.2f s") %
    (self.items, irate, latency, proc_latency))
    self.latency, self.proc_latency, self.items = 0, 0, 0

```

头两个方法非常重要，因为它们具有普遍性。它们用一个Crawler对象启动中间件。你会发现每个重要的中间件都是这么做的。用from\_crawler(cls, crawler)是取得crawler对象。然后，我们注意init()方法引入crawler.settings并设置了一个NotConfigured例外，如果没有设置的话。你可以看到许多FooBar扩展控制着对应的FOOBAR\_ENABLED设置，如果后者没有设置或为False时。这是一个很常见的方式，让settings.py设置（例如，ITEM\_PIPELINES）可以包含相应的中间件，但它默认是关闭的，除非手动打开。许多默认的Scrapy中间件（例如，AutoThrottle或HttpCache）使用这种方式。在我们的例子中，我们的扩展是无效的，除非设置LATENCIES\_INTERVAL。

而后在init()中，我们用crawler.signals.connect()给每个调用设置了信号，并且启动了一些成员变量。其余的类由信号操作。在\_spider\_opened()，我们启动了一个定时器，每隔LATENCIES\_INTERVAL秒，它会调用\_log()方法。在\_spider\_closed()，我们关闭了定时器。在\_request\_scheduled()和\_response\_received()，我们在request.meta存储了时间戳。在\_item\_scraped()，我们得到了两个延迟，被抓取的items数量增加。我们的\_log()方法计算了平均值、格式，然后打印消息，并重设了累加器以开始下一个周期。

笔记：任何在多线程中写过相似代码的人都会赞赏这种不使用互斥锁的方法。对于这个例子，他们的方法可能不会特别复杂，但是单线程代码无疑更容易，在任何场景下都不会太大。

我们可以将这个扩展的代码添加进和settings.py同级目录的latencies.py文件。要使它生效，在settings.py中添加两行：

```
EXTENSIONS = { 'properties.latencies.Latencies': 500, }
LATENCIES_INTERVAL = 5
```

像之前一样运行：

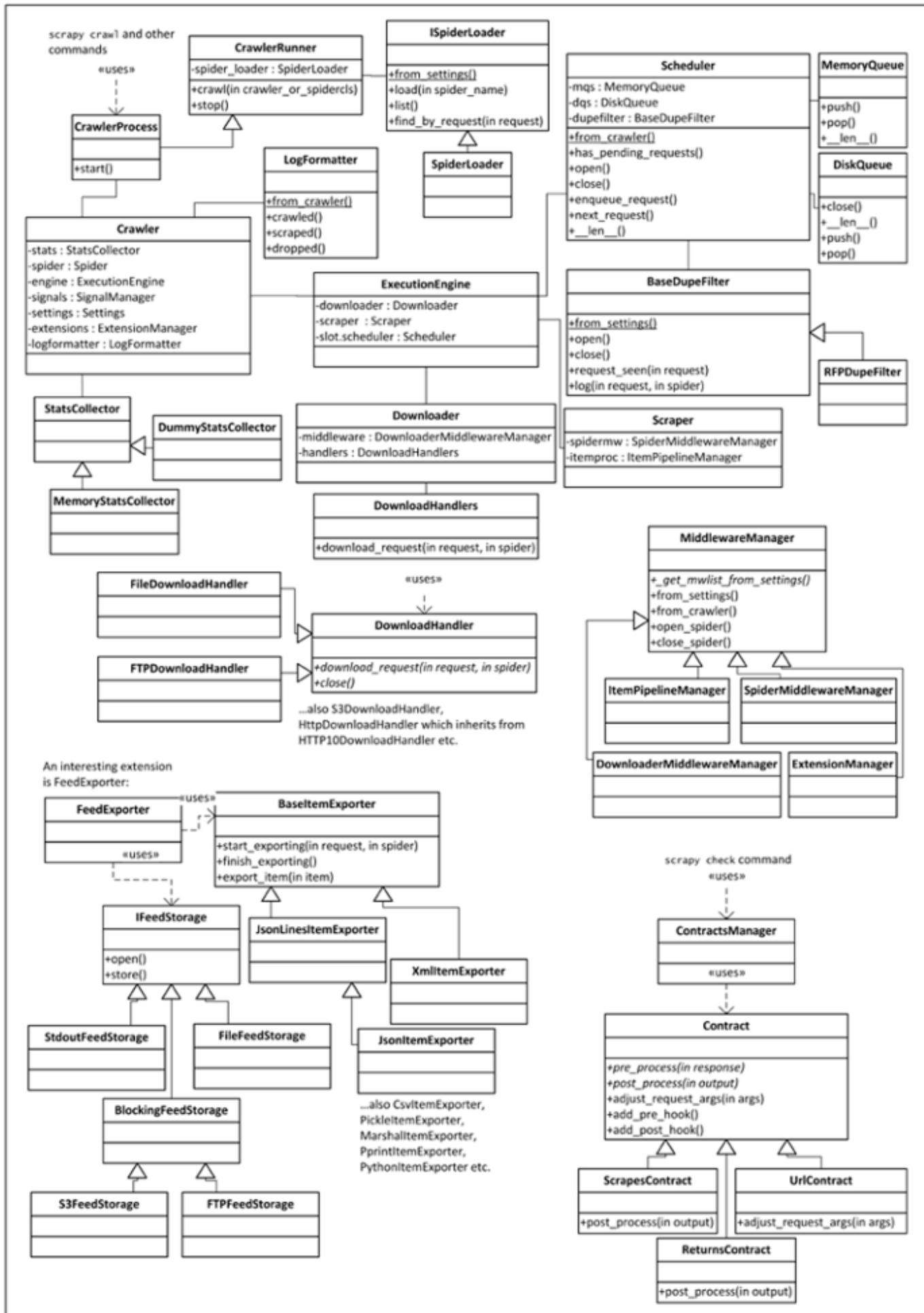
```
$ pwd
/root/book/ch08/properties
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000 -s LOG_LEVEL=INFO
...
INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min)
INFO: Scrapped 0 items at 0.0 items/sec, average latency: 0.00 sec and
average time in pipelines: 0.00 sec
INFO: Scrapped 115 items at 23.0 items/s, avg latency: 0.84 s and avg time
in pipelines: 0.12 s
INFO: Scrapped 125 items at 25.0 items/s, avg latency: 0.78 s and avg time
in pipelines: 0.12 s
```

日志的第一行来自Log Stats扩展，剩下的来自我们的扩展。我们可以看到吞吐量是每秒24个文件，平均延迟是0.78秒，下载之后，我们对其处理的时间很短。Little定律给系统中文件赋值为 $N=ST=430.45 \approx 19$ 。无论我们设置CONCURRENT\_REQUESTS和CONCURRENT\_REQUESTS\_PER\_DOMAIN是什么，尽管我们没有达到100% CPU，这个值很奇怪没有上过30。更多关于此处的内容请见第10章。

## 进一步扩展中间件

这一部分是为感兴趣的读者写的。只写简单和中级的扩展，可以不用看。

如果你看一眼scrapy/settings/default\_settings.py，你会看到很少的类名。Scrapy广泛使用了类似依赖注入的机制，允许我们自定义和扩展它的大部分内部对象。例如，除了DOWNLOAD\_HANDLERS\_BASE设置中定义的文件、HTTP、HTTPS、S3、和FTP协议，有人还想要支持更多的URL协议。要实现的话，只要创建一个DOWNLOAD\_HANDLERS类，并在DOWNLOAD\_HANDLERS设置中添加映射。这里的难点是，你自定义的类的接口是什么（即引入什么方法），因为大多数接口都不清晰。你必须阅读源代码，查看这些类是如何使用的。最好的方法是，采用一个现有的程序，然后改造成你的。随着Scrapy版本的进化，接口变得越来越稳定，我尝试将它们和Scrapy的核心类整理成了一篇文档（我省略了中间件等级）。



核心对象位于左上角。当有人使用scrapy crawl，使用CrawlerProcess对象来创建Crawler对象。Crawler对象是最重要的Scrapy类。它包含settings、signals和spider。在一个名为extensions.crawler的ExtensionManager对象中，它还包括所有的扩展。engine指向另一个非常重要的类ExecutionEngine。它包含了Scheduler、Downloader和Scraper。Scheduler可以对URL进行计划、Downloader用来下载、Scraper可以后处理。Downloader包含了DownloaderMiddleware和DownloadHandler，Scraper包含了SpiderMiddleware和ItemPipeline。这四个MiddlewareManager有等级的区别。Scrapy的输出feeds被当做扩展执行，即FeedExporter。它使用两个独立的层级，一个定于输出类型，另一个定义存储类型。这允许我们，通过调整输出URL，将S3的XML文件中的任何东西输出到Pickle编码的控制台中。两个层级可以进行独立扩展，使用FEED\_STORAGES和FEED\_EXPORTERS设置。最后，通过scrapy check命令，让协议有层级，并可以通过SPIDER\_CONTRACTS设置进行扩展。

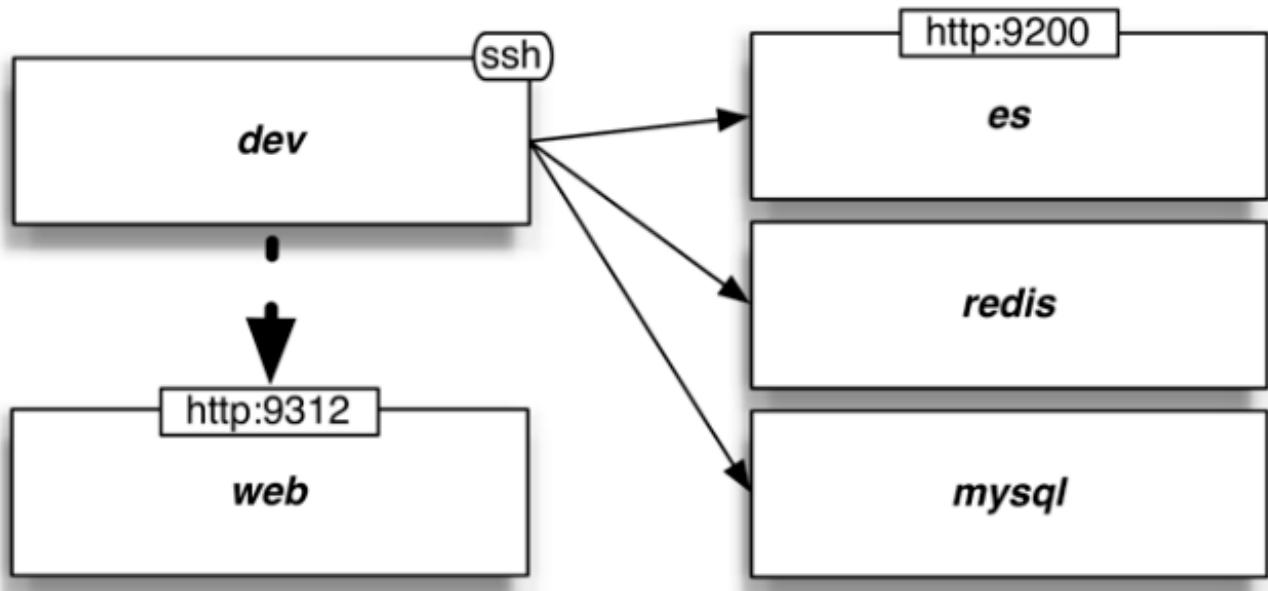
## 总结

你刚刚深度学习了Scrapy和Twisted编程。你可能要多几遍本章，将这章作为参考。目前，最流行的扩展是Item Processing Pipeline。下章学习如何使用它解决许多常见的问题。

# 第9章 使用Pipelines

在上一章，我们学习了如何辨析Scrapy中间件。在本章中，我们通过实例学习编写pipelines，包括使用REST APIs、连接数据库、处理CPU密集型任务、与老技术结合。

我们在本章中会使用集中新的数据库，列在下图的右边：



Vagrant已经配置好了数据库，我们可以从开发机向其发送ping，例如ping es或ping mysql。让

我们先来学习REST APIs。

## 使用REST APIs

---

REST是用来一套创建网络服务的技术集合。它的主要优点是，比起SOAP和专有web服务，REST更简单和轻量。软件开发者注意到了web服务的CRUD（Create、Read、Update、Delete）和HTTP操作（GET、POST、PUT、DELETE）的相似性。它们还注意到传统web服务调用需要的信息可以再URL源进行压缩。例如，<http://api.mysite.com/customer/john>是一个URL源，它可以让我们分辨目标服务器，，更具体的，名字是john的服务器（行的主键）。它与其它技术结合时，比如安全认证、无状态服务、缓存、输出XML或JSON时，可以提供一个强大但简单的跨平台服务。REST席卷软件行业并不奇怪。

Scrapy pipeline的功能可以用REST API来做。接下来，我们来学习它。

## 使用treq

---

treq是一个Python包，它在Twisted应用中和Python的requests包相似。它可以让我们做出GET、POST、和其它HTTP请求。可以使用pip install treq安装，开发机中已经安装好了。

比起Scrapy的Request/crawler.engine.download() API，我们使用treq，因为后者具有性能优势，详见第10章。

## 一个写入Elasticsearch的pipeline

---

我们从一个向ES服务器（Elasticsearch）写入Items的爬虫开始。你可能觉得从ES开始，而不是MySQL，有点奇怪，但实际上ES是最容易的。ES可以是无模式的，意味着我们可以不用配置就使用它。treq也足以应付需要。如果想使用更高级的ES功能，我们应该使用txes2和其它Python/Twisted ES包。

有了Vagrant，我们已经有个一个运行的ES服务器。登录开发机，验证ES是否运行：

```
$ curl http://es:9200
{
  "name" : "Living Brain",
  "cluster_name" : "elasticsearch",
  "version" : { ... },
  "tagline" : "You Know, for Search"
}
```

在浏览器中登录<http://localhost:9200>也可以看到相同的结果。如果访问[http://localhost:9200/properties/property/\\_search](http://localhost:9200/properties/property/_search)，我们可以看到一个响应，说ES已经进行了全局尝试，但是没有找到索引页。

笔记：在本章中，我们会在项集合中插入新的项，如果你想恢复原始状态的话，可以用下面的命令：

```
$ curl -XDELETE http://es:9200/properties
```

本章中的pipeline完整代码还有错误处理的功能，但我尽量让这里的代码简短，以突出重点。

提示：本章位于目录ch09，这个例子位于ch09/properties/properties/pipelines/es.py。

本质上，这个爬虫只有四行：

```
@defer.inlineCallbacks
def process_item(self, item, spider):
    data = json.dumps(dict(item), ensure_ascii=False).encode("utf- 8")
    yield treq.post(self.es_url, data)
```

前两行定义了一个标准process\_item()方法，它可以产生延迟项。（参考第8章）

第三行准备了插入的data。ensure\_ascii=False可使结果压缩，并且没有跳过非ASCII字符。我们然后将JSON字符串转化为JSON标准的默认编码UTF-8。

最后一行使用了treq的post()方法，模拟一个POST请求，将我们的文档插入ElasticSearch。es\_url，例如<http://es:9200/properties/property>存在settings.py文件中（ES\_PIPELINE\_URL设置），它提供重要的信息，例如我们想要写入的ES的IP和端口（es:9200）、集合名(properties) 和对象类型(property)。

为了是pipeline生效，我们要在settings.py中设置ITEM\_PIPELINES，并启动ES\_PIPELINE\_URL设置：

```
ITEM_PIPELINES = {
    'properties.pipelines.tidyup.TidyUp': 100,
    'properties.pipelines.es.EsWriter': 800,
}
ES_PIPELINE_URL = 'http://es:9200/properties/property'
```

这么做完之后，我们前往相应的目录：

```
$ pwd  
/root/book/ch09/properties  
$ ls  
properties  scrapy.cfg
```

然后运行爬虫：

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=90  
...  
INFO: Enabled item pipelines: EsWriter...  
INFO: Closing spider (closespider_itemcount)...  
'item_scraped_count': 106,
```

如果现在访问[http://localhost:9200/properties/property/\\_search](http://localhost:9200/properties/property/_search)，除了前10条结果，我们可以在响应的hits/total字段看到插入的文件数。我们还可以添加参数?size=100以看到更多的结果。通过添加q= URL搜索中的参数，我们可以在全域或特定字段搜索关键词。相关性最强的结果会首先显示出来。例如，[http://localhost:9200/properties/property/\\_search?q=title:london](http://localhost:9200/properties/property/_search?q=title:london)，可以让标题变为London。对于更复杂的查询，可以在<https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html>查询ES文档。

ES不需要配置，因为它根据提供的第一个文件，进行模式（字段类型）自动检测的。通过访问<http://localhost:9200/properties/>，我们可以看到它自动检测的映射。

再次运行crawl easy -s CLOSESPIDER\_ITEMCOUNT=1000。因为pipelines的平均时间从0.12变为0.15秒，平均延迟从0.78变为0.81秒。吞吐量仍保持每秒约25项。

笔记：用pipelines向数据库插入Items是个好方法吗？答案是否定的。通常来讲，数据库更简单的方法以大量插入数据，我们应该使用这些方法大量批次插入数据，或抓取完毕之后进行后处理。我们会在最后一章看到这些方法。然后，还是有很多人使用pipelines向数据库插入文件，相应的就要使用Twisted APIs。

## pipeline使用Google Geocoding API进行地理编码

我们的房子有各自所在的区域，我们还想对它们进行地理编码，即找到相应的坐标（经度、纬度）。我们可以将坐标显示在地图上，或计算距离。建这样的数据库需要复杂的数据库、复杂的文本匹配，还有复杂的空间计算。使用Google Geocoding API，我们可以避免这些。在浏览器中

打开它，或使用curl取回以下URL的数据：

```
$ curl "https://maps.googleapis.com/maps/api/geocode/json?sensor=false&address=london"
{
  "results" : [
    ...
    {
      "formatted_address" : "London, UK",
      "geometry" : {
        ...
        "location" : {
          "lat" : 51.5073509,
          "lng" : -0.1277583
        },
        "location_type" : "APPROXIMATE",
        ...
      },
      "status" : "OK"
    }
  ]
}
```

我们看到一个JSON对象，如果搜索一个location，我们可以快速获取伦敦中心的坐标。如果继续搜索，我们可以看到相同文件中海油其它地点。第一个是相关度最高的。因此如果存在results[0].geometry.location的话，它就是我们要的结果。

可以用前面的方法（treq）使用Google Geocoding API。只需要几行，我们就可以找到一个地址的坐标（目录pipelines中的geo.py），如下所示：

```
@defer.inlineCallbacks
def geocode(self, address):
    endpoint = 'http://web:9312/maps/api/geocode/json'
    parms = [('address', address), ('sensor', 'false')]
    response = yield treq.get(endpoint, params=parms)
    content = yield response.json()
    geo = content['results'][0]['geometry']['location']
    defer.returnValue({"lat": geo["lat"], "lon": geo["lng"]})
```

这个函数做出了一条URL，但我们让它指向一个可以离线快速运行的假程序。你可以使用endpoint = 'https://maps.googleapis.com/maps/api/geocode/json'连接Google服务器，但要记住它对请求的限制很严格。address和sensor的值是URL自动编码的，使用treq的方法get()的参数params。对于第二个yield，即response.json()，我们必须等待响应主题完全加载完毕对解析为Python对象。此时，我们就可以找到第一个结果的地理信息，格式设为dict，使用

`defer.returnValue()`返回，它使用了inlineCallbacks。如果发生错误，这个方法会扔出例外，Scrapy会向我们报告。

通过使用`geocode()`, `process_item()`变成了一行语句：

```
item["location"] = yield self.geocode(item["address"][0])
```

设置让pipeline生效，将它添加到ITEM\_PIPELINES，并设定优先数值，该数值要小于ES的，以让ES获取坐标值：

```
ITEM_PIPELINES = {
...
'properties.pipelines.geo.GeoPipeline': 400,
```

开启数据调试，然后运行：

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=90 -L DEBUG
...
{'address': [u'Greenwich, London'],
...
'image_URL': [u'http://web:9312/images/i06.jpg'],
'location': {'lat': 51.482577, 'lon': -0.007659},
'price': [1030.0],
...}
```

我们现在可以看到Items里的location字段。如果使用真正的Google API的URL运行，会得到例外：

```
File "pipelines/geo.py" in geocode (content['status'], address)
Exception: Unexpected status="OVER_QUERY_LIMIT" for
address="*London"
```

这是为了检查我们在完整代码中插入了地点，以确保Geocoding API响应的status字段有OK值。除非是OK，否则我们取回的数据不会有设定好的格式，进而不能使用。对于这种情况，我们会得到OVER\_QUERY\_LIMIT状态，它指明我们在某处做错了。这个问题很重要，也很常见。应用Scrapy的高性能引擎，进行缓存、限制请求就很必要了。

我们可以在Geocoder API的文档，查看它的限制，“每24小时，免费用户可以进行2500次请求，每秒5次请求”。即使我们使用付费版本，仍有每秒10次请求的限制，所以这里的分析是有意义的。

笔记：后面的代码看起来可能有些复杂，复杂度还要取决于实际情况。在多线程环境中创建这样的组件，需要线程池和同步，这样代码就会变复杂。

这是一个简易的运用Twisted技术的限制引擎：

```
class Throttler(object):
    def __init__(self, rate):
        self.queue = []
        self.looping_call = task.LoopingCall(self._allow_one)
        self.looping_call.start(1. / float(rate))
    def stop(self):
        self.looping_call.stop()
    def throttle(self):
        d = defer.Deferred()
        self.queue.append(d)
        return d
    def _allow_one(self):
        if self.queue:
            self.queue.pop(0).callback(None)
```

这可以让延迟项在一个列表中排队，逐个触发，调用\_allow\_one()；\_allow\_one()检查队列是否为空，如果不是，它会调用第一个延迟项的callback()。我们使用Twisted的task.LoopingCall() API，周期性调用\_allow\_one()。使用Throttler很容易。我们在pipeline的init初始化它，当爬虫停止时清空它：

```
class GeoPipeline(object):
    def __init__(self, stats):
        self.throttler = Throttler(5) # 5 Requests per second
    def close_spider(self, spider):
        self.throttler.stop()
```

在使用限定源之前，我们的例子是在process\_item()中调用geocode()，必须yield限制器的throttle()方法：

```
yield self.throttler.throttle()
item["location"] = yield self.geocode(item["address"][0])
```

对于第一个yield，代码会暂停一下，一段时间之后，会继续运行。例如，当某时有11个延迟项时，限制是每秒5次请求，即时间为 $11/5=2.2$ 秒之后，队列变空，代码会继续。

使用Throttler，不再有错误，但是爬虫会变慢。我们看到示例中的房子只有几个不同的地址。这时使用缓存非常好。我们使用一个简单的Python dict来做，但这么可能不会有竞争条件，这样会造成伪造的API请求。下面是一个没有此类问题的缓存方法，展示了Python和Twisted的特点：

```
class DeferredCache(object):
    def __init__(self, key_not_found_callback):
        self.records = {}
        self.deferreds_waiting = {}
        self.key_not_found_callback = key_not_found_callback
    @defer.inlineCallbacks
    def find(self, key):
        rv = defer.Deferred()
        if key in self.deferreds_waiting:
            self.deferreds_waiting[key].append(rv)
        else:
            self.deferreds_waiting[key] = [rv]
            if not key in self.records:
                try:
                    value = yield self.key_not_found_callback(key)
                    self.records[key] = lambda d: d.callback(value)
                except Exception as e:
                    self.records[key] = lambda d: d.errback(e)
            action = self.records[key]
            for d in self.deferreds_waiting.pop(key):
                reactor.callFromThread(action, d)
        value = yield rv
        defer.returnValue(value)
```

这个缓存看起来有些不同，它包含两个组件：

- self.deferreds\_waiting：这是一个延迟项的队列，等待给键赋值
- self.records：这是键值对中出现过的dict

在find()方法的中间，如果没有在self.records找到一个键，我们会调用预先定义的callback函数，以取回丢失的值（yield self.key\_not\_found\_callback(key)）。这个回调函数可能会扔出一个例外。如何在Python中压缩存储值或例外呢？因为Python是一种函数语言，根据是否有例外，我们在self.records中保存小函数（lambdas），调用callback或errback。lambda函数定义时，就将值或例外附着在上面。将变量附着在函数上称为闭包，闭包是函数语言最重要的特性之一。

笔记：缓存例外有点不常见，但它意味着首次查找key时，key\_not\_found\_callback(key)返回了一个例外。当后续查找还找这个key时，就免去了调用，再次返回这个例外。

find()方法其余的部分提供了一个避免竞争条件的机制。如果查找某个键已经在进程中，会在self.deferreds\_waiting dict中有记录。这时，我们不在向key\_not\_found\_callback()发起另一个调用，只是在延迟项的等待列表添加这个项。当key\_not\_found\_callback()返回时，键有了值，我们触发所有的等待这个键的延迟项。我们可以直接发起action(d)，而不用reactor.callFromThread()，但需要处理每个扔给下游的例外，我们必须创建不必要的很长的延迟项链。

使用这个缓存很容易。我们在init()对其初始化，设定回调函数为API调用。在process\_item()中，使用缓存查找的方法如下：

```
def __init__(self, stats):
    self.cache = DeferredCache(self.cache_key_not_found_callback)
@defer.inlineCallbacks
def cache_key_not_found_callback(self, address):
    yield self.throttler.enqueue()
    value = yield self.geocode(address)
    defer.returnValue(value)
@defer.inlineCallbacks
def process_item(self, item, spider):
    item["location"] = yield self.cache.find(item["address"][0])
    defer.returnValue(item)
```

提示：完整代码位于ch09/properties/properties/pipelines/geo2.py。

为了使pipeline生效，我们使前一个方法无效，并添加当前的到settings.py的ITEM\_PIPELINES：

```
ITEM_PIPELINES = {
    'properties.pipelines.tidyup.TidyUp': 100,
    'properties.pipelines.es.EsWriter': 800,
    # DISABLE 'properties.pipelines.geo.GeoPipeline': 400,
    'properties.pipelines.geo2.GeoPipeline': 400,
}
```

运行爬虫，用如下代码：

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000
...
Scraped... 15.8 items/s, avg latency: 1.74 s and avg time in pipelines:
```

```
0.94 s
Scraped... 32.2 items/s, avg latency: 1.76 s and avg time in pipelines:
0.97 s
Scraped... 25.6 items/s, avg latency: 0.76 s and avg time in pipelines:
0.14 s
...
: Dumping Scrapy stats:...
'geo_pipeline/misses': 35,
'item_scraped_count': 1019,
```

当填充缓存时，我们看到抓取的延迟变高。缓存结束时，延迟降低。数据还显示有35个遗漏，正好是数据集中不同地点的数目。很明显，上例中一共有 $1019 - 35 = 984$ 次API请求。如果我们使用真正的Google API，并提高每秒的API请求数，例如通过改变Throttler(5)到Throttler(10)，使从5提高到10，我们可以将重试添加到geo\_pipeline/retries stat记录中。如果有错误的话，例如，使用API找不到某个地点，会扔出一个例外，这会被geo\_pipeline/errors stat记录。如果地点通过什么方式已经存在了，会在geo\_pipeline/already\_set stat中指明。最后，如果我们访问[http://localhost:9200/properties/property/\\_search](http://localhost:9200/properties/property/_search)，以检查ES中的房子，我们可以看到包括地点的记录，例如{... "location": {"lat": 51.5269736, "lon": -0.0667204}...}。（运行前确保清空集合，去除旧的值）

## 在Elasticsearch进行地理索引

我们已经有了地点，我们可以将它们按距离排序。下面是一个HTTP POST请求，返回标题中包含Angel的房子，按照离点{51.54, -0.19}的距离进行排序：

```
$ curl http://es:9200/properties/property/_search -d '{
  "query" : { "term" : { "title" : "angel" } },
  "sort": [{"_geo_distance": {
    "location": {"lat": 51.54, "lon": -0.19},
    "order": "asc",
    "unit": "km",
    "distance_type": "plane"
  }}]}
}'
```

唯一的问题是如果我们运行它，我们会看到一个错误信息"failed to find mapper for [location] for geo distance based sort"。它指出，我们的location字段没有正确的空间计算的格式。为了设定正确的格式，我们要手动覆盖默认格式。首先，我们将自动检测的映射保存起来，将它作为起点：

```
$ curl 'http://es:9200/_mapping/property' > property.txt
```

然后，我们如下所示编辑property.txt：

```
"location": {"properties": {"lat": {"type": "double"}, "lon": {"type": "double"}}}
```

我们将这行代码替换为：

```
"location": {"type": "geo_point"}
```

我们还在文件最后删除了{"properties":{"mappings": and two }}。文件现在就处理完了。我们现在可以删除旧的类型，并用下面的schema建立新的类型：

```
$ curl -XDELETE 'http://es:9200/_mapping/property'
$ curl -XPUT 'http://es:9200/_mapping/property'
$ curl -XPUT 'http://es:9200/_mapping/property' --data
@property.txt
```

我们现在可以用之前的命令，进行一个快速抓取，将结果按距离排序。我们的搜索返回的是房子的JSONs对象，其中包括一个额外的sort字段，显示房子离某个点的距离。

## 连接数据库与Python客户端

可以连接Python Database API 2.0的数据库有许多种，包括MySQL、PostgreSQL、Oracle、Microsoft、SQL Server和SQLite。它们的驱动通常很复杂且进行过测试，为Twisted再进行适配会浪费很多时间。可以在Twisted应用中使用数据库客户端，例如，Scrapy可以使用twisted.enterprise.adbapi库。我们使用MySQL作为例子，说明用法，原则也适用于其他数据库。

## 用pipeline写入MySQL

MySQL是一个好用又流行的数据库。我们来写一个pipeline，来向其中写入文件。我们的虚拟环境中，已经有了一个MySQL实例。我们用MySQL命令行来做一些基本的管理操作，命令行工具

已经在开发机中预先安装了：

```
$ mysql -h mysql -uroot -ppass
```

mysql> 提示MySQL已经运行，我们可以建立一个简单的含有几个字段的数据表，如下所示：

```
mysql> create database properties;
mysql> use properties
mysql> CREATE TABLE properties (
    url varchar(100) NOT NULL,
    title varchar(30),
    price DOUBLE,
    description varchar(30),
    PRIMARY KEY (url)
);
mysql> SELECT * FROM properties LIMIT 10;
Empty set (0.00 sec)
```

很好，现在已经建好了一个包含几个字段的MySQL数据表，它的名字是properties，可以开始写pipeline了。保持MySQL控制台打开，我们过一会儿会返回查看是否有差入值。输入exit，就可以退出。

笔记：在这一部分中，我们会向MySQL数据库插入properties。如果你想删除，使用以下命令：

```
mysql> DELETE FROM properties;
```

我们使用MySQL的Python客户端。我们还要安装一个叫做dj-database-url的小功能模块（它可以帮助我们设置不同的IP、端口、密码等等）。我们可以用pip install dj-database-url MySQL-python，安装这两项。我们的开发机上已经安装好了。我们的MySQL pipeline很简单，如下所示：

```
from twisted.enterprise import adbapi
...
class MysqlWriter(object):
    ...
    def __init__(self, mysql_url):
        conn_kwarg = MysqlWriter.parse_mysql_url(mysql_url)
        self.dbpool = adbapi.ConnectionPool('MySQLdb',
```

```

        charset='utf8',
        use_unicode=True,
        connect_timeout=5,
        **conn_kwargs)

def close_spider(self, spider):
    self.dbpool.close()
@defer.inlineCallbacks
def process_item(self, item, spider):
    try:
        yield self.dbpool.runInteraction(self.do_replace, item)
    except:
        print traceback.format_exc()
    defer.returnValue(item)
@staticmethod
def do_replace(tx, item):
    sql = """REPLACE INTO properties (url, title, price, description) VALUES (%s,%s,%s,%s)"""
    args = (
        item["url"][0][:100],
        item["title"][0][:30],
        item["price"][0],
        item["description"][0].replace("\r\n", " ")[ :30]
    )
    tx.execute(sql, args)

```

提示：完整代码位于ch09/properties/properties/pipelines/mysql.py。

本质上，这段代码的大部分都很普通。为了简洁而省略的代码将一条保存在 MySQL\_PIPELINE\_URL、格式是mysql://user:pass@ip/database的URL，解析成了独立的参数。在爬虫的init()中，将它们传递到adbapi.ConnectionPool()，它使用adbapi的底层结构，初始化MySQL连接池。第一个参数是我们想要引入的模块的名字。对于我们的MySQL，它是 MySQLdb。我们为MySQL客户端另设了几个参数，以便正确处理Unicode和超时。每当adbapi 打开新连接时，所有这些参数都要进入底层的MySQLdb.connect()函数。爬虫关闭时，我们调用连接池的close()方法。

我们的process\_item()方法包装了dbpool.runInteraction()。这个方法给回调方法排队，会在当连接池中一个连接的Transaction对象变为可用时被调用。这个Transaction对象有一个和DB-API指针相似的API。在我们的例子中，回调方法是do\_replace()，它定义在后面几行。@staticmethod 是说这个方法关联的是类而不是具体的类实例，因此，我们可以忽略通常的self参数。如果方法不使用成员的话，最好设其为静态，如果你忘了设为静态也不要紧。这个方法准备了一个SQL字符串、几个参数，并调用Transaction的execute()函数，以进行插入。我们的SQL使用REPLACE INTO，而不用更常见的INSERT INTO，来替换键相同的项。这可以让我们的案例简化。如果我们想用SQL返回数据，例如SELECT声明，我们使用dbpool.runQuery()，我们可能还需要改变默

认指针，方法是设置adbapi.ConnectionPool()的参数cursorclass为cursorclass=MySQLdb.cursors，这样取回数据更为简便。

使用这个pipeline，我们要在settings.py的ITEM\_PIPELINES添加它，还要设置一下MYSQL\_PIPELINE\_URL：

```
ITEM_PIPELINES = { ...
    'properties.pipelines.mysql.MysqlWriter': 700,
...
MYSQL_PIPELINE_URL = 'mysql://root:pass@mysql/properties'
```

执行以下命令：

```
scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000
```

运行这条命令后，返回MySQL控制台，可以看到如下记录：

```
mysql> SELECT COUNT(*) FROM properties;
+-----+
| 1006 |
+-----+
mysql> SELECT * FROM properties LIMIT 4;
+-----+-----+-----+-----+
| url          | title           | price   | description
+-----+-----+-----+-----+
| http://...0.html | Set Unique Family Well | 334.39 | website c
| http://...1.html | Belsize Marylebone Shopp | 388.03 | features
|
| http://...2.html | Bathroom Fully Jubilee S | 365.85 | vibrant own
| http://...3.html | Residential Brentford Ot | 238.71 | go court
+-----+-----+-----+
4 rows in set (0.00 sec)
```

延迟和吞吐量的性能和之前相同。结果让人印象深刻。

## 使用Twisted 特定客户端连接服务

目前为止，我们学习了如何用treq使用类REST APIs。Scrapy可以用Twisted特定客户端连接许多

其它服务。例如，如果我们想连接MongoDB，通过搜索“MongoDB Python”，我们可以找到PyMongo，它是阻塞/同步的，除非我们使用pipeline处理阻塞操作中的线程，我们不能在Twisted中使用PyMongo。如果我们搜索“MongoDB Twisted Python”，可以找到txmongo，它可以完美适用于Twisted和Scrapy。通常的，Twisted客户端群体很小，但使用它比起自己写一个客户端还是要方便。下面，我们就使用这样一个Twisted特定客户端连接Redis键值对存储。

## 用pipeline读写Redis

Google Geocoding API是按照每个IP进行限制的。如果可以接入多个IPs（例如，多台服务器），当一个地址已经被另一台机器做过地理编码，就要设法避免对发出重复的请求。如果一个地址之前已经被查阅过，也要避免再次查阅。我们不想浪费限制的额度。

笔记：与API商家联系，以确保这符合规定。你可能，必须每几分钟/小时，就要清空缓存记录，或者根本就不能缓存。

我们可以使用Redis键值缓存作为分布式dict。Vagrant环境中已经有了一个Redis实例，我们现在可以连接它，用redis-cli作一些基本操作：

```
$ redis-cli -h redis
redis:6379> info keyspace
# Keyspace
redis:6379> set key value
OK
redis:6379> info keyspace
# Keyspace
db0:keys=1,expires=0,avg_ttl=0
redis:6379> FLUSHALL
OK
redis:6379> info keyspace
# Keyspace
redis:6379> exit
```

通过搜索“Redis Twisted”，我们找到一个txredisapi库。它最大的不同是，它不仅是一个Python的同步封装，还是一个Twisted库，可以通过reactor.connectTCP()，执行Twisted协议，连接Redis。其它库也有类似用法，但是txredisapi对于Twisted效率更高。我们可以通过安装库dj\_redis\_url可以安装它，这个库通过pip可以解析Redis配置URL（sudo pip install txredisapi dj\_redis\_url）。和以前一样，开发机中已经安装好了。

我们如下启动RedisCache pipeline：

```

from txredisapi import lazyConnectionPool
class RedisCache(object):
...
    def __init__(self, crawler, redis_url, redis_nm):
        self.redis_url = redis_url
        self.redis_nm = redis_nm
        args = RedisCache.parse_redis_url(redis_url)
        self.connection = lazyConnectionPool(connectTimeout=5,
                                              replyTimeout=5,
                                              **args)
        crawler.signals.connect(
            self.item_scraped, signal=signals.item_scraped)

```

这个pipeline比较简单。为了连接Redis服务器，我们需要主机、端口等等，它们全都用URL格式存储。我们用parse\_redis\_url()方法解析这个格式。使用命名空间做键的前缀很普遍，在我们的例子中，我们存储在redis\_nm。我们然后使用txredisapi的lazyConnectionPool()打开一个数据库连接。

最后一行有一个有趣的函数。我们是想用pipeline封装geo-pipeline。如果在Redis中没有某个值，我们不会设定这个值，geo-pipeline会用API像之前一样将地址进行地理编码。完毕之后，我们必须要在Redis中缓存键值对，我们是通过连接signals.item\_scraped信号来做的。我们定义的回调（即item\_scraped()方法，马上会讲）只有在最后才会被调用，那时，地址就设置好了。

**提示：**完整代码位于ch09/properties/properties/pipelines/redis.py。

我们简化缓存，只寻找和存储每个Item的地址和地点。这对Redis来说是合理的，因为它通常是在单一服务器上的，这可以让它很快。如果不是这样的话，可以加入一个dict结构的缓存，它与我们在geo-pipeline中用到的相似。以下是我们如何处理入库的Items：

```

process incoming Items:
@defer.inlineCallbacks
def process_item(self, item, spider):
    address = item["address"][0]
    key = self.redis_nm + ":" + address
    value = yield self.connection.get(key)
    if value:
        item["location"] = json.loads(value)
    defer.returnValue(item)

```

和预期的相同。我们得到了地址，给它添加前缀，然后使用txredisapi connection的get()在Redis进行查找。我们将JSON编码的对象在Redis中保存成值。如果一个值设定了，我们就使用JSON

解码，然后将其设为地点。

当一个Item到达pipelines的末端时，我们重新取得它，将其保存为Redis中的地点值。以下是我们做法：

```
from txredisapi import ConnectionError
def item_scraped(self, item, spider):
    try:
        location = item["location"]
        value = json.dumps(location, ensure_ascii=False)
    except KeyError:
        return
    address = item["address"][0]
    key = self.redis_nm + ":" + address
    quiet = lambda failure: failure.trap(ConnectionError)
    return self.connection.set(key, value).addErrback(quiet)
```

如果我们找到了一个地点，我们就取得了地址，添加前缀，然后使用它作为txredisapi连接的set()方法的键值对。set()方法没有使用@defer.inlineCallbacks，因为处理signals.item\_scraped时，它不被支持。这意味着，我们不能对connection.set()使用yield，但是我们可以返回一个延迟项，Scrapy可以在它后面排上其它信号对象。任何情况下，如果Redis的连接不能使用用connection.set()，它就会抛出一个例外。在这个错误处理中，我们把传递的错误当做参数，我们让它trap()任何ConnectionError。这是Twisted的延迟API的优点之一。通过用trap()捕获错误项，我们可以轻易忽略它们。

使这个pipeline生效，我们要做的是在settings.py的ITEM\_PIPELINES中添加它，并提供一个REDIS\_PIPELINE\_URL。必须要让它的优先级比geo-pipeline高，以免太晚就不能使用了：

```
ITEM_PIPELINES = { ...
    'properties.pipelines.redis.RedisCache': 300,
    'properties.pipelines.geo.GeoPipeline': 400,
...
REDIS_PIPELINE_URL = 'redis://redis:6379'
```

像之前一样运行。第一次运行时和以前很像，但随后的运行结果如下：

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=100
...
INFO: Enabled item pipelines: TidyUp, RedisCache, GeoPipeline,
MysqlWriter, EsWriter
...
```

```
Scraped... 0.0 items/s, avg latency: 0.00 s, time in pipelines: 0.00 s
Scraped... 21.2 items/s, avg latency: 0.78 s, time in pipelines: 0.15 s
Scraped... 24.2 items/s, avg latency: 0.82 s, time in pipelines: 0.16 s
...
INFO: Dumping Scrapy stats: {...  
    'geo_pipeline/already_set': 106,  
    'item_scraped_count': 106,
```

我们看到GeoPipeline和RedisCache都生效了，RedisCache第一个输出。还注意到在统计中geo\_pipeline/already\_set: 106。这是GeoPipeline发现的Redis缓存中填充的数目，它不调用Google API。如果Redis缓存是空的，你会看到Google API处理了一些键。从性能上来讲，我们看到GeoPipeline引发的初始行为消失了。事实上，当我们开始使用内存，我们绕过了每秒只有5次请求的API限制。如果我们使用Redis，应该考虑使用过期键，让系统周期刷新缓存数据。

## 连接CPU密集型、阻塞或旧方法

最后一部分讲连接非Twisted的工作。尽管异步程序的优点很多，并不是所有库都专门为Twisted和Scrapy写的。使用Twisted的线程池和reactor.spawnProcess()方法，我们可以使用任何Python库和任何语言写的编码。

## pipeline进行CPU密集型和阻塞操作

我们在第8章中强调，反应器适合简短非阻塞的任务。如果我们不得不处理复杂和阻塞的任务，又该怎么做呢？Twisted提供了线程池，有了它可以使用reactor.callInThread() API在分线程而不是主线程中执行慢操作。这意味着，反应器可以一直运行并对事件反馈，而不中断计算。但要记住，在线程池中运行并不安全，当你使用全局模式时，会有多线程的同步问题。让我们从一个简单的pipeline开始，逐渐做出完整的代码：

```
class UsingBlocking(object):
    @defer.inlineCallbacks
    def process_item(self, item, spider):
        price = item["price"][0]
        out = defer.Deferred()
        reactor.callInThread(self._do_calculation, price, out)
        item["price"][0] = yield out
        defer.returnValue(item)
    def _do_calculation(self, price, out):
        new_price = price + 1
        time.sleep(0.10)
        reactor.callFromThread(out.callback, new_price)
```

在前面的pipeline中，我们看到了一些基本用法。对于每个Item，我们提取出价格，我们相用`_do_calucation()`方法处理它。这个方法使用`time.sleep()`，一个阻塞操作。我们用`reactor.callInThread()`调用，让它在另一个线程中运行。显然，我们传递价格，我们还创建和传递了一个名为out的延迟项。当`_do_calucation()`完成了计算，我们使用out调回值。下一步，我们执行延迟项，并未价格设新的值，最后返回Item。

在`_do_calucation()`中，有一个细微之处，价格增加了1，进而睡了100ms。这个时间很多，如果调用进反应器主线程，每秒就不能抓取10页了。通过在另一个线程中运行，就不会再有这个问题。任务会在线程池中排队，每次处理耗时100ms。最后一步是触发调回。一般的，我们可以使用`out.callback(new_price)`，但是因为我们现在是在另一个线程，这么做不安全。如果这么做的话，延迟项的代码会被从另一个线程调用，这样迟早会产生错误的数据。不这样做，转而使用`reactor.callFromThread()`，它也可以将函数当做参数，将任意其余参数传递到函数。这个函数会排队并被调回主线程，主进程反过来会打开`process_item()`对象`yield`，并继续Item的操作。

如果我们用全局模式，例如计数器、滑动平均，又该怎么使用`_do_calucation()`呢？例如，添加两个变量，`beta`和`delta`，如下所示：

```
class UsingBlocking(object):
    def __init__(self):
        self.beta, self.delta = 0, 0
    ...
    def _do_calculation(self, price, out):
        self.beta += 1
        time.sleep(0.001)
        self.delta += 1
        new_price = price + self.beta - self.delta + 1
        assert abs(new_price-price-1) < 0.01
        time.sleep(0.10)...
```

这段代码是断言失败错误。这是因为如果一个线程在`self.beta`和`self.delta`间切换，另一个线程继续计算使用`beta/delta`计算价格，它会发现它们状态不一致（`beta`大于`delta`），因此，计算出错误的结果。短暂的睡眠可能会造成竞争条件。为了不发生这些状况，我们要使一个锁，例如Python的`threading.RLock()`递归锁。使用它，可以确保没有两个线程在同一时间操作被保护代码：

```
class UsingBlocking(object):
    def __init__(self):
        ...
        self.lock = threading.RLock()
```

```
...
def _do_calculation(self, price, out):
    with self.lock:
        self.beta += 1
    ...
    new_price = price + self.beta - self.delta + 1
assert abs(new_price-price-1) < 0.01 ...
```

代码现在就正确了。记住，我们不需要保护整段代码，就足以处理全局模式。

提示：完整代码位于ch09/properties/properties/pipelines/computation.py。

要使用这个pipeline，我们需要把它添加到settings.py的ITEM\_PIPELINES中。如下所示：

```
ITEM_PIPELINES = { ...
    'properties.pipelines.computation.UsingBlocking': 500,
```

像之前一样运行爬虫，pipeline延迟达到了100ms，但吞吐量没有发生变化，大概每秒25个items。

## pipeline使用二进制和脚本

最麻烦的借口当属独立可执行文件和脚本。打开需要几秒（例如，从数据库加载数据），但是后面处理数值的延迟很小。即便是这种情况，Twisted也预料到了。我们可以使用reactor.spawnProcess() API和相关的protocol.ProcessProtocol来运行任何执行文件。让我们来看一个例子。脚本如下：

```
#!/bin/bash
trap "" SIGINT
sleep 3
while read line
do
    # 4 per second
    sleep 0.25
    awk "BEGIN {print 1.20 * $line}"
done
```

这是一个简单的bash脚本。它运行时，会使Ctrl + C 无效。这是为了避免系统的一个奇怪的错误，将Ctrl + C增值到子流程并过早结束，导致Scrapy强制等待流程结果。在使Ctrl + C无效之

后，它睡眠三秒，模拟启动时间。然后，它阅读输入的代码语句，等待250ms，然后返回结果价格，价格的值乘以了1.20，由Linux的awk命令计算而得。这段脚本的最大吞吐量为每秒1/250ms=4个Items。用一个短session检测：

```
$ properties/pipelines/legacy.sh
12 <- If you type this quickly you will wait ~3 seconds to get results
14.40
13 <- For further numbers you will notice just a slight delay
15.60
```

因为Ctrl + C失效了，我们用Ctrl + D必须结束session。我们该如何让Scrapy使用这个脚本呢？再一次，我们从一个简化版开始：

```
class CommandSlot(protocol.ProcessProtocol):
    def __init__(self, args):
        self._queue = []
        reactor.spawnProcess(self, args[0], args)
    def legacy_calculate(self, price):
        d = defer.Deferred()
        self._queue.append(d)
        self.transport.write("%f\n" % price)
        return d
    # Overriding from protocol.ProcessProtocol
    def outReceived(self, data):
        """Called when new output is received"""
        self._queue.pop(0).callback(float(data))
class Pricing(object):
    def __init__(self):
        self.slot = CommandSlot(['properties/pipelines/legacy.sh'])
    @defer.inlineCallbacks
    def process_item(self, item, spider):
        item["price"][0] = yield self.slot.legacy_calculate(item["price"][0])
        defer.returnValue(item)
```

我们在这里找到了一个名为CommandSlot的ProcessProtocol和Pricing爬虫。在`__init__()`中，我们创建了新的CommandSlot，它新建了一个空的队列，并用`reactor.spawnProcess()`开启了一个新进程。它调用收发数据的ProcessProtocol作为第一个参数。在这个例子中，是`self`的原因是`spawnProcess()`是被从类`protocol`调用的。第二个参数是可执行文件的名字，第三个参数`args`，让二进制命令行参数成为字符串序列。

在pipeline的`process_item()`中，我们用CommandSlot的`legacy_calculate()`方法代表所有工作，

CommandSlot可以返回产生的延迟项。legacy\_calculate()创建延迟项，将其排队，用transport.write()将价格写入进程。ProcessProtocol提供了transport，可以让我们与进程沟通。无论何时我们从进程收到数据，outReceived()就会被调用。通过延迟项，进程依次执行，我们可以弹出最老的延迟项，用收到的值触发它。全过程就是这样。我们可以让这个pipeline生效，通过将它添加到ITEM\_PIPELINES：

```
ITEM_PIPELINES = {...  
    'properties.pipelines.legacy.Pricing': 600,
```

如果运行的话，我们会看到性能很差。进程变成了瓶颈，限制了吞吐量。为了提高性能，我们需要修改pipeline，允许多个进程并行运行，如下所示：

```
class Pricing(object):  
    def __init__(self):  
        self.concurrency = 16  
        args = ['properties/pipelines/legacy.sh']  
        self.slots = [CommandSlot(args)  
                     for i in xrange(self.concurrency)]  
        self.rr = 0  
    @defer.inlineCallbacks  
    def process_item(self, item, spider):  
        slot = self.slots[self.rr]  
        self.rr = (self.rr + 1) % self.concurrency  
        item["price"][] = yield  
            slot.legacy_calculate(item["price"][])  
        defer.returnValue(item)
```

这无非是开启16个实例，将价格以轮转的方式发出。这个pipeline的吞吐量是每秒 $16 \times 4 = 64$ 。我们可以用下面的爬虫进行验证：

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000  
...  
Scraped... 0.0 items/s, avg latency: 0.00 s and avg time in pipelines:  
0.00 s  
Scraped... 21.0 items/s, avg latency: 2.20 s and avg time in pipelines:  
1.48 s  
Scraped... 24.2 items/s, avg latency: 1.16 s and avg time in pipelines:  
0.52 s
```

延迟增加了250 ms，但吞吐量仍然是每秒25。

请记住前面的方法使用了`transport.write()`让所有的价格在脚本shell中排队。这个可能对你的应用不适用，，尤其是当数据量很大时。Git的完整代码让值和回调都进行了排队，不想脚本发送值，除非收到前一项的结果。这种方法可能看起来更友好，但是会增加代码复杂度。

## 总结

---

你刚刚学习了复杂的Scrapy pipelines。目前为止，你应该就掌握了所有和Twisted编程相关的知识。并且你学会了如何在进程中执行复杂的功能，用Item Processing Pipelines存储Items。我们看到了添加pipelines对延迟和吞吐量的影响。通常，延迟和吞吐量是成反比的。但是，这是在恒定并发数的前提下（例如，一定数量的线程）。在我们的例子中，我们一开始的并发数为 $N=ST=250.77 \approx 19$ ，添加pipelines之后，并发数为 $N=25*3.33 \approx 83$ ，并没有引起性能的变化。这就是Twisted的强大之处！下面学习第10章，Scrapy的性能。

# 第10章 理解Scrapy的性能

---

通常，很容易将性能理解错。对于Scrapy，几乎一定会把它的性能理解错，因为这里有许多反直觉的地方。除非你对Scrapy的结构有清楚的了解，你会发现努力提升Scrapy的性能却收效甚微。这就是处理高性能、低延迟、高并发环境的复杂之处。对于优化瓶颈，Amdahl定律仍然适用，但除非找到真正的瓶颈，吞吐量并不会增加。要想学习更多，可以看Dr.Goldratt的《目标》这本书，其中用比喻讲到了更多关于瓶颈、吞吐量的知识。本章就是来帮你确认Scrapy配置的瓶颈所在，让你避免明显的错误。

请记住，本章相对较难，涉及到许多数学。但计算还算比较简单，并且有图表示意。如果你不喜欢数学，可以直接忽略公式，这样仍然可以搞明白Scrapy的性能是怎么回事。

## Scrapy的引擎——一个直观的方法

---

并行系统看起来就像管道系统。在计算机科学中，我们使用队列符表示队列并处理元素（见图1的左边）。队列系统的基本定律是Little定律，它指明平衡状态下，队列系统中的总元素个数（N）等于吞吐量（T）乘以总排队/处理时间（S），即 $N=T*S$ 。另外两种形式， $T=N/S$ 和 $S=N/T$ 也十分有用。

## Queueing theory



~



Little's law:  $N = T \cdot S$

$$N=8 R$$

$$S=.25 s$$

$$T=32 R/s$$

## Pipes



$$N=16 R$$

$$S=.25 s$$

$$T=64 R/s$$

$$N=32 R$$

$$S=.25 s$$

$$T=128 R/s$$

管道（图1的右边）在几何学上也有一个相似的定律。管道的体积（V）等于长度（L）乘以横截面积（A），即 $V=L \cdot A$ 。

如果假设L代表处理时间S（ $L \approx S$ ），体积代表总元素个数（ $V \approx N$ ），横截面积A代表吞吐量（ $A \approx T$ ），Little定律和体积公式就是相同的。

提示：这个类比合理吗？答案是基本合理。如果我们想象小液滴在管道中以匀速流过，那么 $L \approx S$ 就完全合理，因为管道越长，液滴流过的时间也越长。 $V \approx N$ 也是合理的，因为管道越大，它能容下的液滴越多。但是，我们可以通过增大压力的方法，压入更多的液滴。 $A \approx T$ 是真正的类比。在管道中，吞吐量是每秒流进/流出的液滴总数，被称为体积流速，在正常的情况下，它与 $A^2$ 成正比。这是因为更宽的管道不仅意味更多的液体流出，还具有更快的速度，因为管壁之间的空间变大了。但对于这一章，我们可以忽略这一点，假设压力和速度是不变的，吞吐量只与横截面积成正比。

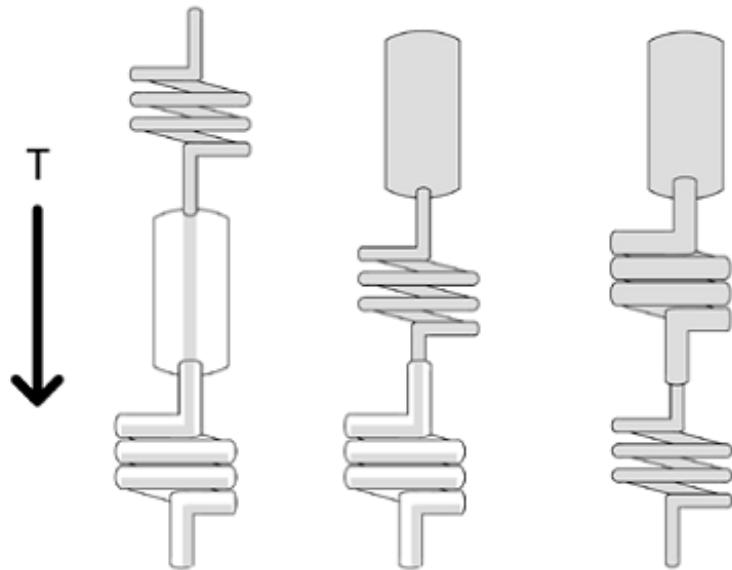
Little定律与体积公式十分相似，所以管道模型直观上是正确的。再看看图1中的右半部。假设管道代表Scrapy的下载器。第一个十分细的管道，它的总体积/并发等级（N）=8个并发请求。长度/延迟（S）对于一个高速网站，假设为 $S=250\text{ms}$ 。现在可以计算横街面积/吞吐量 $T=N/S=8/0.25=32\text{请求/秒}$ 。

可以看到，延迟率是手远程服务器和网络延迟的影响，不受我们控制。我们可以控制的是下载器的并发等级（N），将8提升到16或32，如图1所示。对于固定的管道长度（这也不受我们控制），我们只能增加横截面积来增加体积，即增加吞吐量。用Little定律来讲，并发如果是16个请求，就有 $T=N/S=16/0.25=64\text{请求/秒}$ ，并发32个请求，就有 $T=N/S=32/0.25=128\text{请求/秒}$ 。貌似如果并发数无限大，吞吐量也就无限大。在得出这个结论之前，我们还得考虑一下串联排队系统。

## 串联排队系统

当你将横截面积/吞吐量不同的管道连接起来时，直观上，人们会认为总系统会受限于最窄的管道

(最小的吞吐量T) , 见图2。



你还可以看到最窄的管道（即瓶颈）放在不同的地方，可以影响其他管道的填充程度。如果将填充程度类比为系统内存需求，瓶颈的摆放就十分重要了。最好能将填充程度达到最高，这样单位工作的花费最小。在Scrapy中，单位工作（抓取一个网页）大体包括下载器之前的一条URL（几个字节）和下载器之后的URL和服务器响应。

提示：这就是为什么，Scrapy把瓶颈放在下载器。

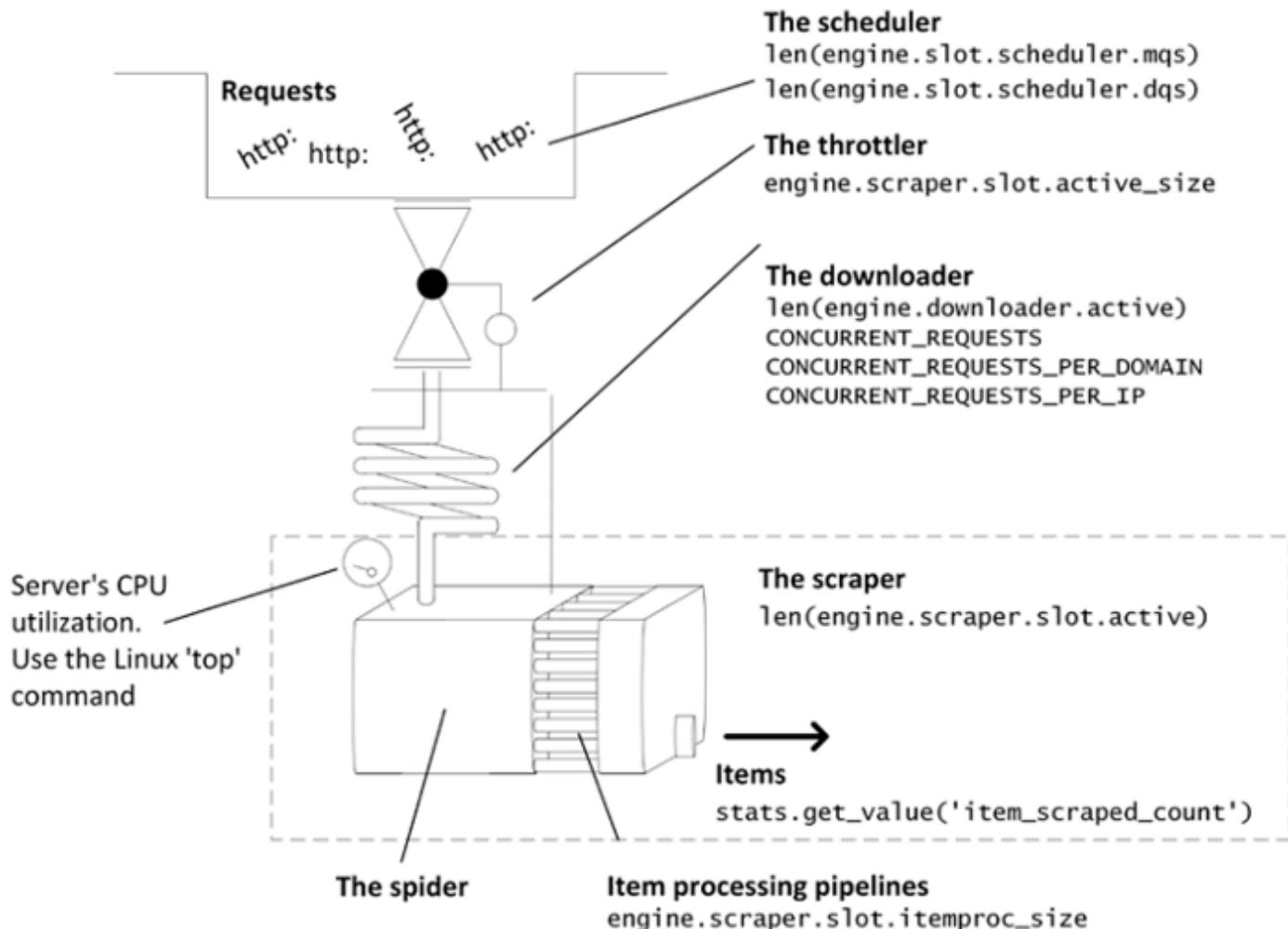
## 确认瓶颈

用管道系统的比喻，可以直观的确认瓶颈所在。查看图2，你可以看到瓶颈之前都是满的，瓶颈之后就不是满的。

对于大多数系统，可以用系统的性能指标监测排队系统是否拥挤。通过检测Scrapy的队列，我们可以确定出瓶颈的所在，如果瓶颈不是在下载器的话，我们可以通过调整设置使下载器成为瓶颈。瓶颈没有得到优化，吞吐量就不会有优化。调整其它部分只会使系统变得更糟，很可能将瓶颈移到别处。所以在修改代码和配置之前，你必须找到瓶颈。你会发现在大多数情况下，包括本书中的例子，瓶颈的位置都和预想的不同。

## Scrapy的性能模型

让我们回到Scrapy，详细查看它的性能模型，见图3。



Scrapy包括以下部分：

- **调度器**: 大量的Request在这里排队，直到下载器处理它们。其中大部分是URL，因此体积不大，也就是说即便有大量请求存在，也可以被下载器及时处理。
- **阻塞器**: 这是抓取器由后向前进行反馈的一个安全阀，如果进程中的响应大于5MB，阻塞器就会暂停更多的请求进入下载器。这可能会造成性能的波动。
- **下载器**: 这是对Scrapy的性能最重要的组件。它用复杂的机制限制了并发数。它的延迟（管道长度）等于远程服务器的响应时间，加上网络/操作系统、Python/Twisted的延迟。我们可以调节并发请求数，但是对其它延迟无能为力。下载器的能力受限于CONCURRENT\_REQUESTS\*设置。
- **爬虫**: 这是抓取器将Response变为Item和其它Request的组件。只要我们遵循规则来写爬虫，通常它不是瓶颈。
- **Item Pipelines**: 这是抓取器的第二部分。我们的爬虫对每个Request可能产生几百个Items，只有CONCURRENT\_ITEMS会被并行处理。这一点很重要，因为，如果你用pipelines连接数据库，你可能无意地向数据库导入数据，pipelines的默认值（100）就会看起来很少。

爬虫和pipelines的代码是异步的，会包含必要的延迟，但二者不会是瓶颈。爬虫和pipelines很少

会做繁重的处理工作。如果是的话，服务器的CPU则是瓶颈。

## 使用远程登录控制组件

为了理解Requests/Items是如何在管道中流动的，我们现在还不能真正的测量流动。然而，我们可以检测在Scrapy的每个阶段，有多少个Requests/Responses/Items。

通过Scrapy运行远程登录，我们就可以得到性能信息。我们可以在6023端口运行远程登录命令。然后，会在Scrapy中出现一个Python控制台。注意，如果在这里进行中断操作，比如`time.sleep()`，就会暂停爬虫。通过内建的`est()`函数，可以查看一些有趣的信息。其中一些或是非常专业的，或是可以从核心数据推导出来。本章后面会展示后者。下面运行一个例子。当我们运行一个爬虫时，我们在开发机打开第二台终端，在端口6023远程登录，然后运行`est()`。

提示：本章代码位于目录ch10。这个例子位于ch10/speed。

在第一台终端，运行如下命令：

```
$ pwd  
/root/book/ch10/speed  
$ ls  
scrapy.cfg speed  
$ scrapy crawl speed -s SPEED_PIPELINE_ASYNC_DELAY=1  
INFO: Scrapy 1.0.3 started (bot: speed)  
...
```

现在先不关注`scrapy crawl speed`和它的参数的意义，后面会详解。在第二台终端，运行如下代码：

```
$ telnet localhost 6023  
>>> est()  
...  
len(engine.downloader.active) : 16  
...  
len(engine.slot.scheduler.mqs) : 4475  
...  
len(engine.scrape.slot.active) : 115  
engine.scrape.slot.active_size : 117760  
engine.scrape.slot.itemproc_size : 105
```

然后在第二台终端按Ctrl+D退出远程登录，返回第一台终端按Ctrl+C停止抓取。

提示：我们现在忽略dqs。如果你通过设置JOBDIR打开了持久支持，你会得到非零的dqs (len(engine.slot.scheduler.dqs))，你应该将它添加到mq的大小中。

让我们查看这个例子中的数据的意义。mq指出调度器中等待的项目很少（4475个请求）。  
len(engine.downloader.active)指出下载器现在正在下载16个请求。这与我们在CONCURRENT\_REQUESTS的设置相同。len(engine.scrape.slot.active)说明现在正有115个响应在抓取器中处理。(engine.scrape.slot.active\_size)告诉我们这些响应的大小是115kb。除了响应，105个Items正在pipelines(engine.scrape.slot.itemproc\_size)中处理，这说明还有10个在爬虫中。经过总结，我们看到瓶颈是下载器，在下载器之前有很长的任务队列(mq)，下载器在满负荷运转；下载器之后，工作量较高并有一定波动。

另一个可以查看信息的地方是stats对象，抓取之后打印的内容。我们可以以dict的形式访问它，只需通过via stats.get\_stats()远程登录，用p()函数打印：

```
$ p(stats.get_stats())
{'downloader/request_bytes': 558330,
...
'item_scraped_count': 2485,
...}
```

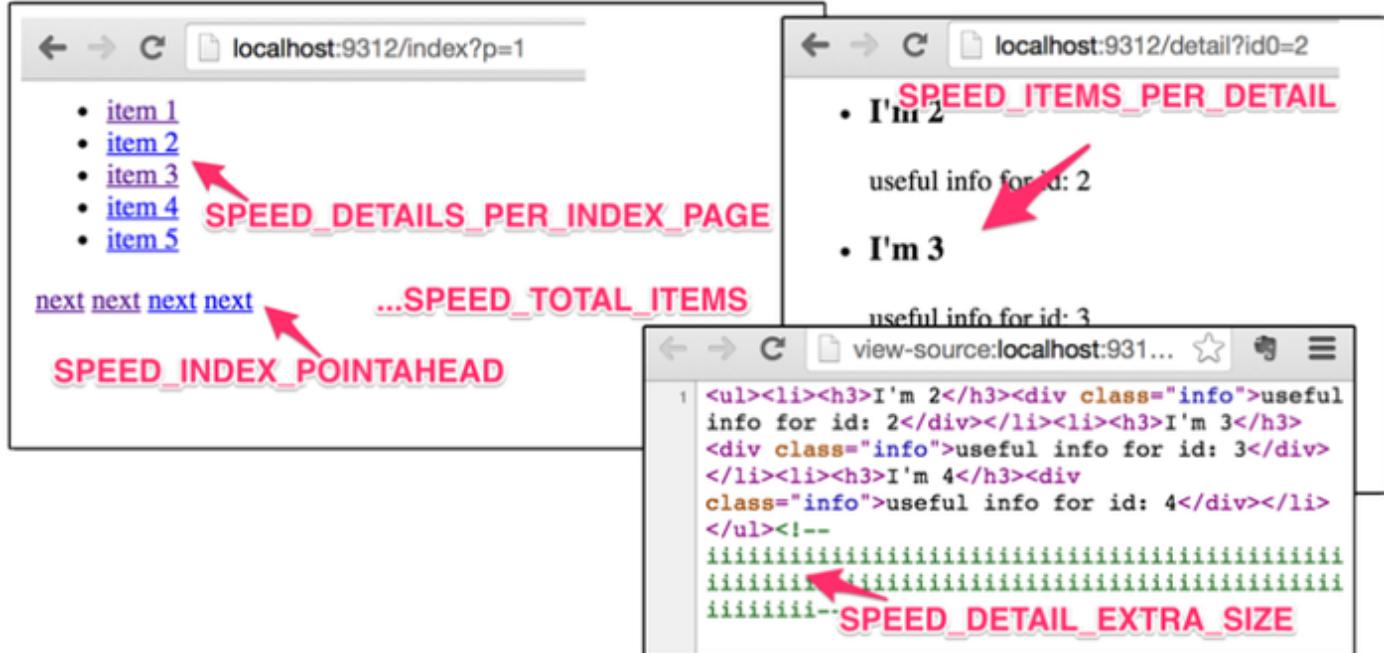
这里对我们最重要的是item\_scraped\_count，它可以通过stats.get\_value('item\_scraped\_count')之间访问。它告诉我们现在已经抓取了多少个items，以及增长的速率，即吞吐量。

## 评分系统

我为本章写了一个简单的评分系统，它可以让我们评估在不同场景下的性能。它的代码有些复杂，你可以在speed/spiders/speed.py找到，但我们不会深入讲解它。

这个评分系统包括：

- 服务器上<http://localhost:9312/benchmark/>...的句柄(handlers)。我们可以控制这个假网站的结构（见图4），通过调节URL参数/Scrapy设置，控制网页加载的速度。不用在意细节，我们接下来会看许多例子。现在，先看一下<http://localhost:9312/benchmark/index?p=1>和<http://localhost:9312/benchmark/id:3/rr:5/index?p=1>的不同。第一个网页在半秒内加载完毕，每页只含有一个item，第二个网页加载用了五秒，每页有三个items。我们还可以在网页上添加垃圾信息，降低加载速度。例如，查看<http://localhost:9312/benchmark/ds:100/detail?id=0>。默认条件下（见speed/settings.py），页面渲染用时SPEED\_T\_RESPONSE = 0.125秒，假网站有SPEED\_TOTAL\_ITEMS = 5000个Items。



- 爬虫，SpeedSpider，模拟用几种方式取回被SPEED\_START\_REQUESTS\_STYLE控制的start\_requests()，并给出一个parse\_item()方法。默认下，用crawler.engine.crawl()方法将所有起始URL提供给调度器。
- pipeline，DummyPipeline，模拟了一些处理过程。它可以引入四种不同的延迟类型。阻塞/计算/同步延迟(SPEED\_PIPELINE\_BLOCKING\_DELAY—很差)，异步延迟(SPEED\_PIPELINE\_ASYNC\_DELAY—不错)，使用远程treq库进行API调用(SPEED\_PIPELINE\_API\_VIA\_TREQ—不错)，和使用Scrapy的crawler.engine.download()进行API调用(SPEED\_PIPELINE\_API\_VIA\_DOWNLOADER—不怎么好)。默认时，pipeline不添加延迟。
- settings.py中的一组高性能设置。关闭任何可能使系统降速的项。因为只在本地服务器运行，我们还关闭了每个域的请求限制。
- 一个可以记录数据的扩展，和第8章中的类似。它每隔一段时间，就打印出核心数据。

在上一个例子，我们已经用过了这个系统，让我们重新做一次模拟，并使用Linux的计时器测量总共的执行时间。核心数据打印如下：

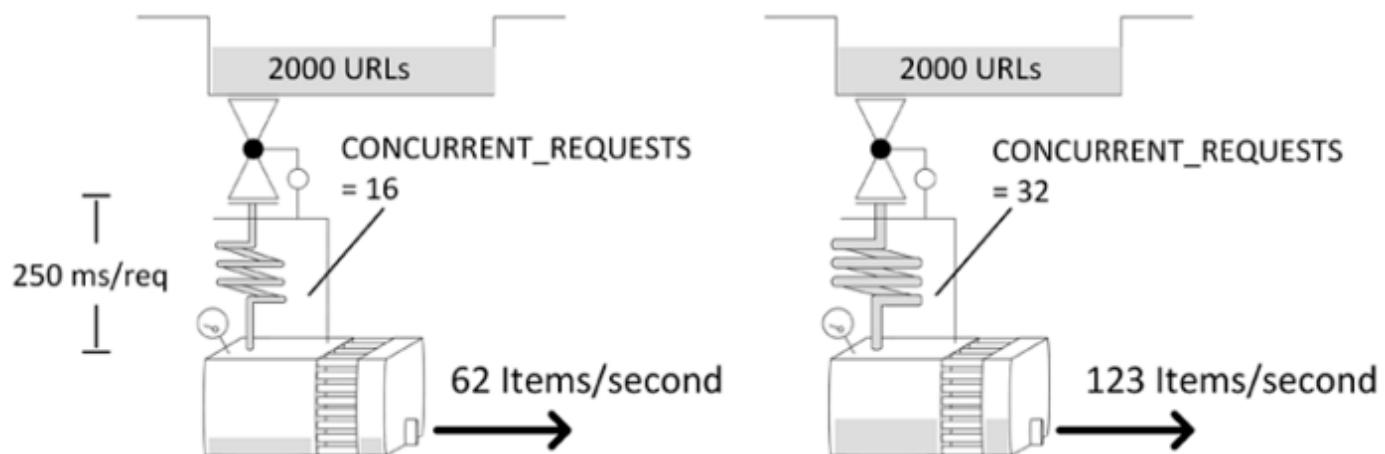
```
$ time scrapy crawl speed
...
INFO: schedule d/load scrape p/line done mem
INFO: 0 0 0 0 0 0
INFO: 4938 14 16 0 32 16384
INFO: 4831 16 6 0 147 6144
...
INFO: 119 16 16 0 4849 16384
INFO: 2 16 12 0 4970 12288
...
real 0m46.561s
```

Column	Metric
schedule	<code>len(engine.slot.scheduler.mqs)</code>
download	<code>len(engine.downloader.active)</code>
scrape	<code>len(engine.scraping.slot.active)</code>
pipeline	<code>engine.scraping.slot.itemproc_size</code>
done	<code>stats.get_value('item_scraped_count')</code>
mem	<code>engine.scraping.slot.active_size</code>

结果这样显示出来效果很好。调度器中初始有5000条URL，结束时done的列也有5000条。下载器全负荷下并发数是16，与设置相同。抓取器主要是爬虫，因为pipeline是空的，它没有满负荷运转。它用46秒抓取了5000个Items，并发数是16，即每个请求的处理时间是 $46*16/5000=147\text{ms}$ ，而不是预想的125ms，满足要求。

## 标准性能模型

当Scrapy正常运行且下载器为瓶颈时，就是Scrapy的标准性能模型。此时，调度器有一定数量的请求，下载器满负荷运行。抓取器负荷不满，并且加载的响应不会持续增加。



三项设置负责控制下载器的性能：CONCURRENT\_REQUESTS, CONCURRENT\_REQUESTS\_PER\_DOMAIN和CONCURRENT\_REQUESTS\_PER\_IP。第一个是宏观上的控制，无论任何时候，并发数都不能超过CONCURRENT\_REQUESTS。另外，如果是单域或几个域，CONCURRENT\_REQUESTS\_PER\_DOMAIN也可以限制活跃请求数。如果你设置了CONCURRENT\_REQUESTS\_PER\_IP，CONCURRENT\_REQUESTS\_PER\_DOMAIN就会被忽略，活跃请求数就是每个IP的请求数量。对于共享站点，比如，多个域名指向一个服务器，这可以帮助你降低服务器的载荷。

为了更简明的分析，现在把per-IP的限制关闭，即使CONCURRENT\_REQUESTS\_PER\_IP为默认值（0），并设置CONCURRENT\_REQUESTS\_PER\_DOMAIN为一个超大值（1000000）。这样就可以无视其它的设置，让下载器的并发数完全受CONCURRENT\_REQUESTS控制。

我们希望吞吐量取决于下载网页的平均时间，包括远程服务器和我们系统（Linux、Twisted/Python）的延迟， $t_{download}=t_{response}+t_{overhead}$ 。还可以加上启动和关闭的时间。这包括从取得响应到Items离开pipeline的时间，和取得第一个响应的时间，还有空缓存的内部损耗。

总之，如果你要完成N个请求，在爬虫正常的情况下，需要花费的时间是：

$$t_{job} = \frac{N \cdot (t_{response} + t_{overhead})}{\text{CONCURRENT\_REQUESTS}} + t_{start/stop}$$

所幸的是，我们只需控制一部分参数就可以了。我们可以用一台更高效的服务器控制overhead，和tstart/stop，但是后者并不值得，因为每次运行只影响一次。除此之外，最值得关注的就是CONCURRENT\_REQUESTS，它取决于我们如何使用服务器。如果将其设置成一个很大的值，在某一时刻就会使服务器或我们电脑的CPU满负荷，这样响应就会不及时，tresponse会急剧升高，因为网站会阻塞、屏蔽进一步的访问，或者服务器会崩溃。

让我们验证一下这个理论。我们抓取2000个items， $t_{response} \in \{0.125s, 0.25s, 0.5s\}$ ， $\text{CONCURRENT\_REQUESTS} \in \{8, 16, 32, 64\}$ ：

```
$ for delay in 0.125 0.25 0.50; do for concurrent in 8 16 32 64; do
    time scrapy crawl speed -s SPEED_TOTAL_ITEMS=2000 \
    -s CONCURRENT_REQUESTS=$concurrent -s SPEED_T_RESPONSE=$delay
done; done
```

在我的电脑上，我完成2000个请求的时间如下：

并发数	125ms/req	250ms/req	500ms/req
8	36.1	67.3	129.7
16	19.4	35.3	66.1
32	11.1	19.3	34.7
64	7.4	11.1	19.0

接下来复杂的数学推导，可以跳过。在图5中，可以看到一些结果。将上一个公式变形为 $y=t_{overhead} \cdot x + t_{start/stop}$ ，其中 $x=N/\text{CONCURRENT\_REQUESTS}$ ， $y=t_{job} \cdot x + t_{response}$ 。使用最小二乘法（LINEST Excel函数）和前面的数据，可以计算出 $t_{overhead}=6ms$ ， $t_{start/stop}=3.1s$ 。 $t_{overhead}$ 可以忽略，但是开始时间相对较长，最好是在数千条URL时长时间运行。因此，可以估算出吞吐量公式是：

$$T = \frac{N}{t_{job} - t_{start/stop}}$$

处理N个请求，我们可以估算 $t_{job}$ ，然后可以直接求出T。

## 解决性能问题

现在我们已经明白如何使Scrapy的性能最大化，让我们来看看如何解决实际问题。我们会通过探究症状、运行错误、讨论原因、修复问题，讨论几个实例。呈现的顺序是从系统性的问题到Scrapy的小技术问题，也就是说，更为常见的问题可能会排在后面。请阅读全部章节，再开始处理你自己的问题。

## 实例1——CPU满负荷

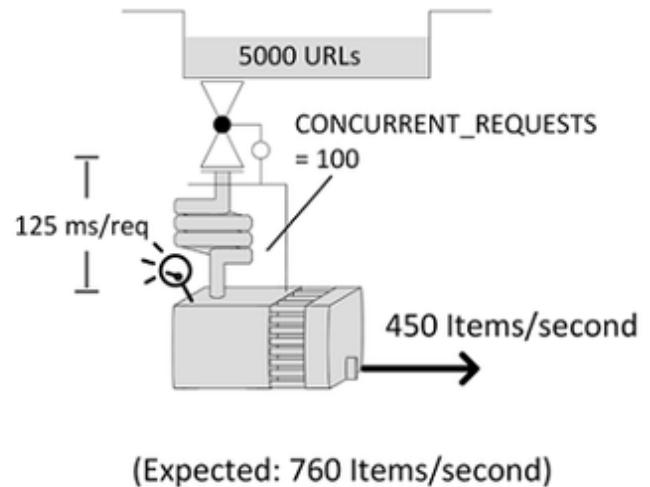
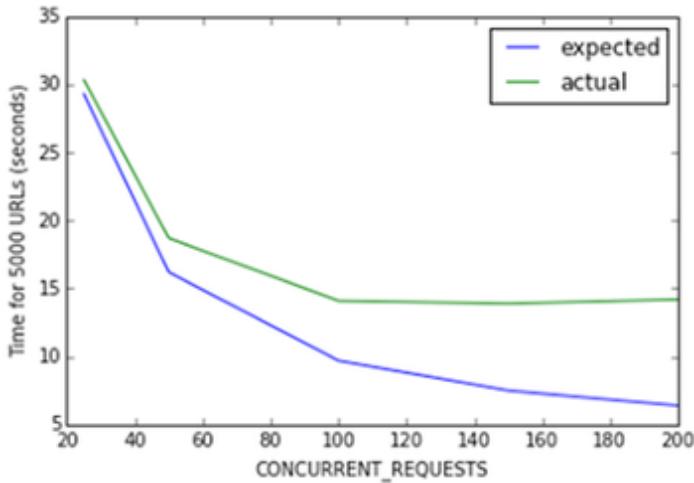
**症状：**当你提高并发数时，性能并没有提高。当你降低并发数，一切工作正常。下载器没有问题，但是每个请求花费时间太长。用Unix/Linux命令ps或Windows的任务管理器查看CPU的情况，CPU的占用率非常高。

**案例：**假设你运行如下命令：

```
$ for concurrent in 25 50 100 150 200; do
    time scrapy crawl speed -s SPEED_TOTAL_ITEMS=5000 \
    -s CONCURRENT_REQUESTS=$concurrent
done
```

求得抓取5000条URL的时间。预计时间是用之前推导的公式求出的，CPU是用命令查看得到的（可以在另一台终端运行查看命令）：

并发请求数	预计时间 (秒)	实际时间 (秒)	占比	CPU 载荷
25	29.3	30.34	97%	52%
50	16.2	18.7	87%	78%
100	9.7	14.1	69%	92%
150	7.5	13.9	54%	100%
200	6.4	14.2	45%	100%



在我们的试验中，我们没有进行任何处理工作，所以并发数可以很高。在实际中，很快就可以看到性能趋缓的情况发生。

讨论：Scrapy使用的是单线程，当并发数很高时，CPU可能会成为瓶颈。假设没有使用线程池，CPU的使用率建议是80-90%。可能你还会碰到其他系统性问题，比如带宽、内存、硬盘吞吐量，但是发生这些状况的可能性比较小，并且不属于系统管理，所以就不赘述了。

解决：假设你的代码已经是高效的。你可以通过在一台服务器上运行多个爬虫，使累积并发数超过CONCURRENT\_REQUESTS。这可以充分利用CPU的性能。如果还想提高并发数，你可以使用多台服务器（见11章），这样就可以使用更多的内存、带宽和硬盘吞吐量。检查CPU的使用情况是你的首要关切。

## 实例2-阻塞代码

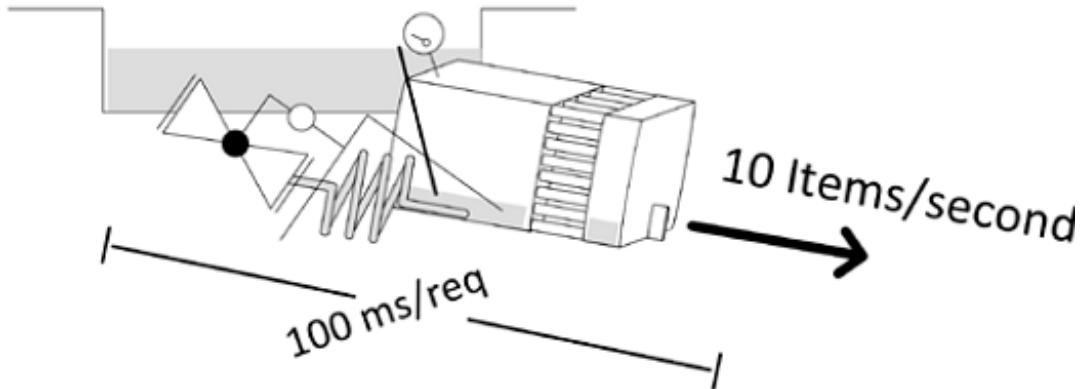
症状：系统的运行得十分奇怪。比起预期的速度，系统运行的十分缓慢。改变并发数，也没有效果。下载器几乎是空的（远小于并发数），抓取器的响应数很少。

案例：使用两个评分设置，SPEED\_SPIDER\_BLOCKING\_DELAY和SPEED\_PIPELINE\_BLOCKING\_DELAY（二者效果相同），使每个响应有100ms的阻塞延迟。在给定的并发数下，100条URL大概要2到3秒，但结果总是13秒左右，并且不受并发数影响：

```
for concurrent in 16 32 64; do
    time scrapy crawl speed -s SPEED_TOTAL_ITEMS=100 \
    -s CONCURRENT_REQUESTS=$concurrent -s SPEED_SPIDER_BLOCKING_DELAY=0.1
done
```

并发数	总时间 (秒)
16	13.9
32	13.2
64	12.9

讨论：任何阻塞代码都会是并发数无效，并使得CONCURRENT\_REQUESTS=1。公式：  
 $100\text{URL} * 100\text{ms}(\text{阻塞延迟}) = 10\text{秒} + t_{\text{start/stop}}$ ，完美解释了发生的状况。



### CONCURRENT\_REQUESTS = 1 (?!)

无论阻塞代码位于pipelines还是爬虫，你都会看到抓取器满负荷，它之前和之后的部分都是空的。看起来这违背了我们之前讲的，但是由于我们并没有一个并行系统，pipeline的规则此处并不适用。这个错误很容易犯（例如，使用了阻塞APIs），然后就会出现之前的状况。相似的讨论也适用于计算复杂的代码。应该为每个代码使用多线程，如第9章所示，或在Scrapy的外部批次运行，第11章会看到例子。

解决：假设代码是继承而来的，你并不知道阻塞代码位于何处。没有pipelines系统也能运行的话，使pipeline无效，看系统能否正常运行。如果是的话，说明阻塞代码位于pipelines。如果不是的话，逐一恢复pipelines，看问题何时发生。如果必须所有组件都在运行，整个系统才能运行的话，给每个pipeline阶段添加日志消息（或者插入可以打印时间戳的伪pipelines），就可以发现哪一步花费的时间最多。如果你想要一个长期可重复使用的解决方案，你可以在每个meta字段添加时间戳的伪pipelines追踪请求。最后，连接item\_scraped信号，打印出时间戳。一旦找到阻塞代码，将其转化为Twisted/异步，或使用Twisted的线程池。要查看转化的效果，将SPEED\_PIPELINE\_BLOCKING\_DELAY替换为SPEED\_PIPELINE\_ASYNC\_DELAY，然后再次运行。可以看到性能改进很大。

## 实例3-下载器中有“垃圾”

症状：吞吐量比预期的低。下载器的请求数貌似比并发数多。

案例：模拟下载1000个网页，每个响应时间是0.25秒。当并发数是16时，根据公式，整个过程大概需要19秒。我们使用一个pipeline，它使用crawler.engine.download()向一个响应时间小于一秒的伪装API做另一个HTTP请求，。你可以在<http://localhost:9312/benchmark/ar:1/api?text=hello>尝试。下面运行爬虫：

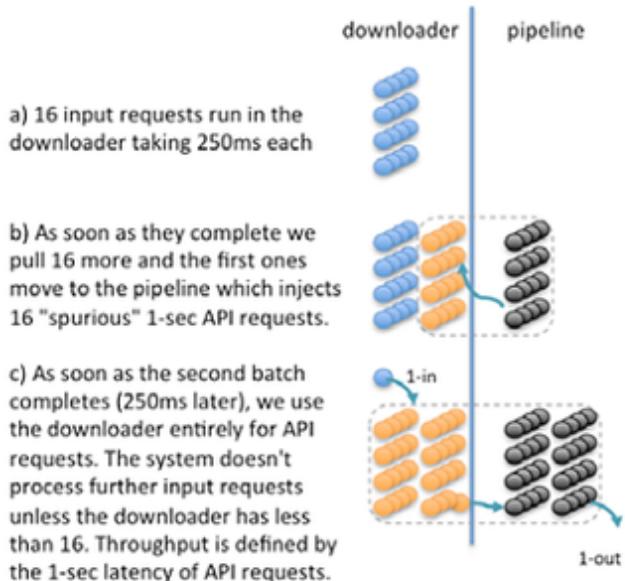
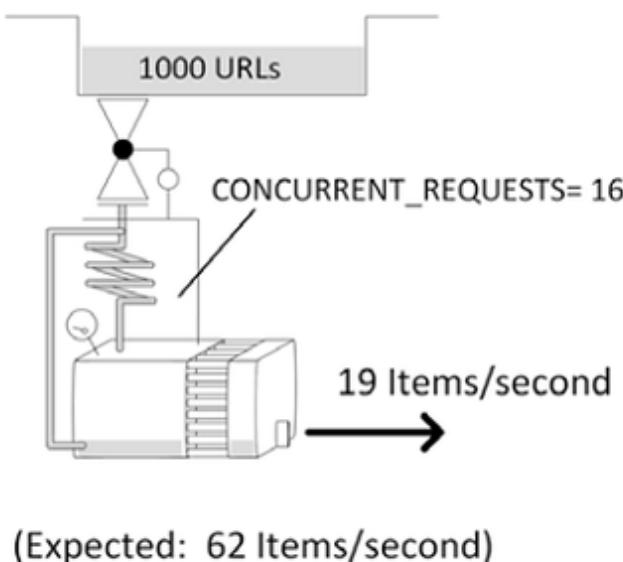
```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=1000 -s SPEED_T_RESPONSE=0.25 -s SPEED_API_T_RESPONSE=1 -s SPEED_PIPELINE_API_VIA_DOWNLOADER=1
...
schedule    d/load    scrape    p/line      done      mem
    968        32        32        32        0    32768
    952        16         0         0       32        0
    936        32        32        32       32    32768
...
real 0m55.151s
```

很奇怪，不仅时间多花了三倍，并发数也比设置的数值16要大。下载器明显是瓶颈，因为它已经过载了。让我们重新运行爬虫，在另一台终端，远程登录Scrapy。然后就可以查看下载器中运行的Requests是哪个：

```
$ telnet localhost 6023
>>> engine.downloader.active
set([<POST http://web:9312/ar:1/ti:1000/rr:0.25/benchmark/api>, ...])
```

貌似下载器主要是在做APIs请求，而不是下载网页。

讨论：你可能希望没人使用crawler.engine.download()，因为它看起来很复杂，但在Scrapy的robots.txt中间件和媒体pipeline，它被使用了两次。因此，当人们需要处理网络APIs时，自然而然要使用它。使用它远比使用阻塞APIs要好，例如前面看过的流行的Python的requests包。比起理解Twisted和使用treq，它使用起来也更简单。这个错误很难调试，所以让我们转而查看下载器中的请求。如果看到有API或媒体URL不是直接抓取的，就说明pipelines使用了crawler.engine.download()进行了HTTP请求。我们的ONCURRENT\_REQUESTS限制部队这些请求生效，所以下载器中的请求数总是超过设置的并发数。除非伪请求数小于CONCURRENT\_REQUESTS，下载器不会从调度器取得新的网页请求。



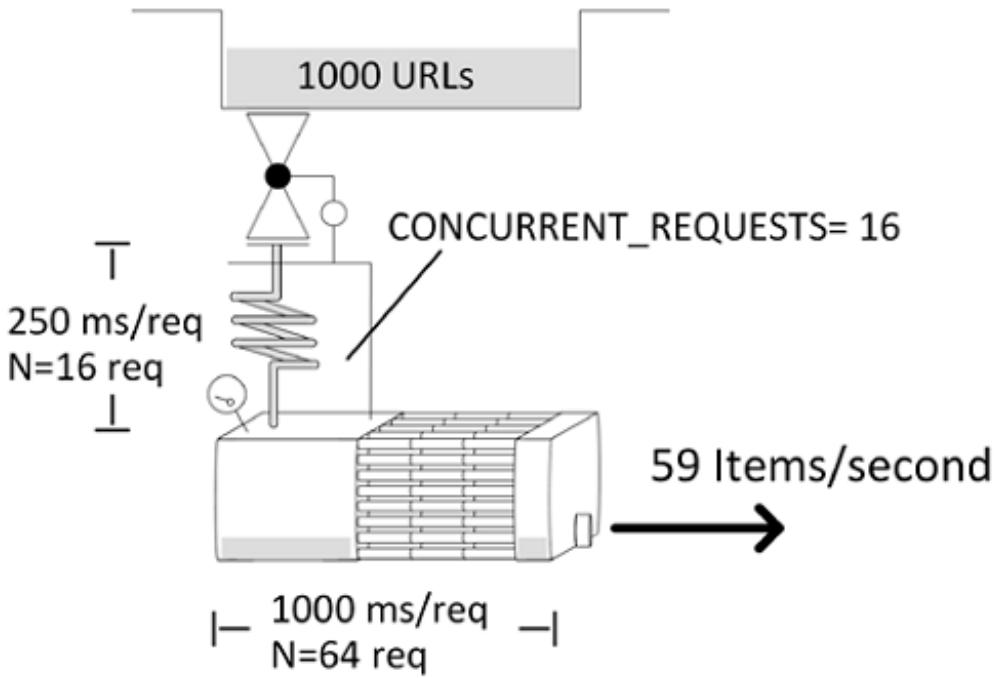
因此，当原始请求持续1秒（API延迟）而不是0.25秒时（页面下载延迟），吞吐量自然会发生变化。这里容易让人迷惑的地方是，要是API的调用比网页请求还快，我们根本不会观察到性能的下降。

解决：我们可以使用treq而不是crawler.engine.download()解决这个问题，你可以看到抓取器的性能大幅提高，这对API可能不是个好消息。我先将CONCURRENT\_REQUESTS设置的很低，然后逐步提高，以确保不让API服务器过载。

下面是使用treq的例子：

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=1000 -s SPEED_T_RESPONSE=0.25 -s SPEED_API_T_RESPONSE=1 -s SPEED_PIPELINE_API_VIA_TREQ=1
...
s/edule  d/load  scrape  p/line      done      mem
  936      16      48      32       0     49152
  887      16      65      64      32    66560
  823      16      65      52      96    66560
...
real 0m19.922s
```

可以看到一个有趣的现象。pipeline (p/line)的items似乎比下载器(d/load)的还多。这并不是一个问题，弄清楚它是很有意思的。



和预期一样，下载器中有16条请求。这意味着系统的吞吐量是 $T = N/S = 16/0.25 = 64$ 请求/秒。  
done这一列逐渐升高，可以确认这点。每条请求在下载器中耗时0.25秒，但它在pipelines中会耗时1秒，因为较慢的API请求。这意味着在pipeline中，平均的 $N = T * S = 64 * 1 = 64$  Items。这完全合理。这是说pipelines是瓶颈吗？不是，因为pipelines没有同时处理响应数量的限制。只要这个数字不持续增加，就没有问题。接下来会进一步讨论。

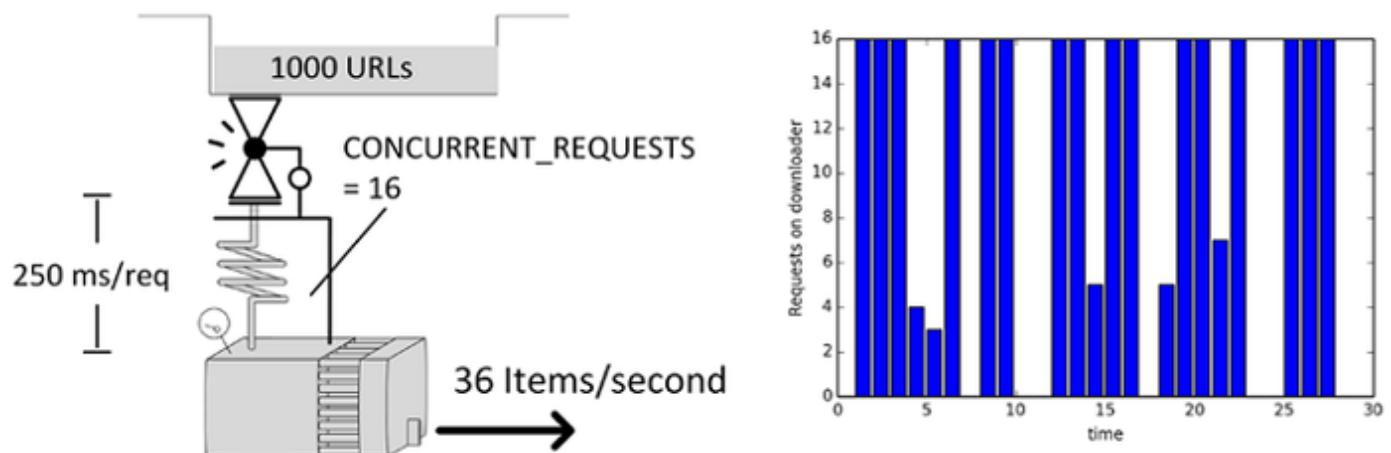
## 实例4-大量响应造成溢出

症状：下载器几乎满负荷运转，一段时间后关闭。这种情况循环发生。抓取器的内存使用很高。

案例：设置和以前相同（使用treq），但响应很高，有大约120kB的HTML。可以看到，这次耗时31秒而不是20秒：

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=1000 -s SPEED_T_
RESPONSE=0.25 -s SPEED_API_T_RESPONSE=1 -s SPEED_PIPELINE_API_VIA_TREQ=1
-s SPEED_DETAIL_EXTRA_SIZE=120000
s/edule d/load scrape p/line done mem
 952    16    32    32      0  3842818
 917    16    35    35     32  4203080
 876    16    41    41     67  4923608
 840      4    48    43    108  5764224
 805      3    46    27    149  5524048
...
real 0m30.61s
```

讨论：我们可能简单的认为延迟的原因是“需要更多的时间创建、传输、处理网页”，但这并不是真正的原因。对于响应的大小有一个强制性的限制，`max_active_size = 5000000`。每一个响应都和响应体的大小相同，至少为1kB。



这个限制可能是Scrapy最基本的机制，当存在慢爬虫和pipelines时，以保证性能。如果pipelines的吞吐量小于下载器的吞吐量，这个机制就会起作用。当pipelines的处理时间很长，即便是很小的响应也可能触发这个机制。下面是一个极端的例子，pipelines非常长，80秒后出现问题：

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=10000 -s SPEED_T_RESPONSE=0.25 -s SPEED_PIPELINE_ASYNC_DELAY=85
```

解决：对于这个问题，在底层结构上很难做什么。当你不再需要响应体的时候，可以立即清除它。这可能是在爬虫的后续清除响应体，但是这么做不会重置抓取器的计数器。你能做的是减少pipelines的处理时间，减少抓取器中的响应数量。用传统的优化方法就可以做到：检查交互中的APIs或数据库是否支持抓取器的吞吐量，估算下载器的能力，将pipelines进行后批次处理，或使用性能更强的服务器或分布式抓取。

## 实例5-item并发受限/过量造成溢出

症状：爬虫对每个响应产生多个Items。吞吐量比预期的小，和之前的实例相似，也呈现出间歇性。

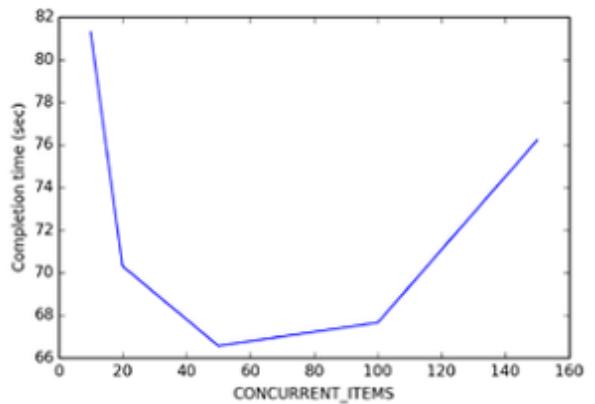
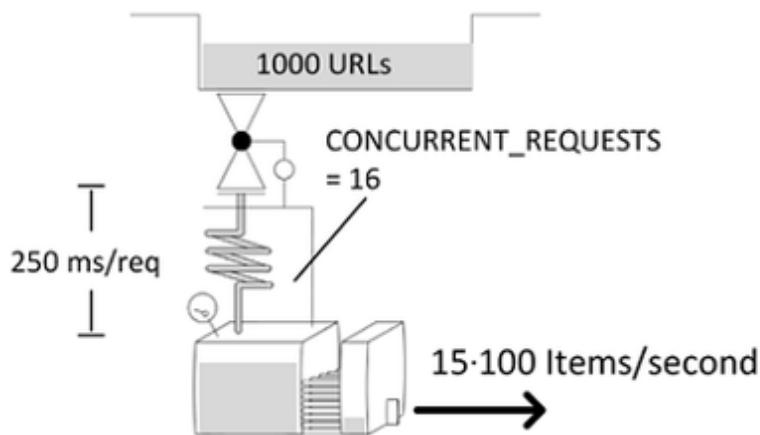
案例：我们有1000个请求，每一个会返回100个items。响应时间是0.25秒，pipelines处理时间是3秒。进行几次试验，`CONCURRENT_ITEMS`的范围是10到150：

```
for concurrent_items in 10 20 50 100 150; do
    time scrapy crawl speed -s SPEED_TOTAL_ITEMS=100000 -s \
    SPEED_T_RESPONSE=0.25 -s SPEED_ITEMS_PER_DETAIL=100 -s \
```

```

SPEED_PIPELINE_ASYNC_DELAY=3 -s \
CONCURRENT_ITEMS=$concurrent_items
done
...
s/edule d/load scrape p/line done mem
952     16      32    180     0  243714
920     16      64    640     0  487426
888     16      96   960     0  731138
...

```



讨论：只有每个响应产生多个Items时才出现这种情况。这个案例的人为性太强，因为吞吐量达到了每秒1300个Items。吞吐量这么高是因为稳定的低延迟、没进行处理、响应很小。这样的条件很少见。

我们首先观察到的是，以前scrape和p/line两列的数值是相同的，现在p/line显示的是shows CONCURRENT\_ITEMS \* scrape。这是因为scrape显示Responses，而p/line显示Items。

第二个是图11中像一个浴缸的函数。部分原因是纵坐标轴造成的。在左侧，有非常高延迟，因为达到了内存极限。右侧，并发数太大，CPU使用率太高。取得最优化并不是那么重要，因为很容易向左或向右变动。

解决：很容易检测出这个例子中的两个错误。如果CPU使用率太高，就降低并发数。如果达到了5MB的响应限制，pipelines就不能很好的衔接下载器的吞吐量，提高并发数就可以解决。如果不能解决问题，就查看一下前面的解决方案，并审视是否系统的其它部分可以支撑抓取器的吞吐量。

## 实例6-下载器没有充分运行

症状：提高了CONCURRENT\_REQUESTS，但是下载器中的数量并没有提高，并且没有充分利用。调度器是空的。

案例：首先运行一个没有问题的例子。将响应时间设为1秒，这样可以简化计算，使下载器吞吐量 $T = N/S = N/1 = \text{CONCURRENT\_REQUESTS}$ 。然后运行如下代码：

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=500 \
-s SPEED_T_RESPONSE=1 -s CONCURRENT_REQUESTS=64
s/edule d/load scrape p/line done mem
 436      64      0      0      0      0
...
real 0m10.99s
```

下载器满状态运行（64个请求），总时长为11秒，和500条URL、每秒64请求的模型相符， $S=N/T+t_{\text{start}}/t_{\text{stop}}=500/64+3.1=10.91$ 秒。

现在，再做相同的抓取，不再像之前从列表中提取URL，这次使用 `SPEED_START_REQUESTS_STYLE=UseIndex` 从索引页提取URL。这与其它章的方法是一样的。每个索引页有20条URL：

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=500 \
-s SPEED_T_RESPONSE=1 -s CONCURRENT_REQUESTS=64 \
-s SPEED_START_REQUESTS_STYLE=UseIndex
s/edule d/load scrape p/line done mem
 0      1      0      0      0      0
 0      21     0      0      0      0
 0      21     0      0      20     0
...
real 0m32.24s
```

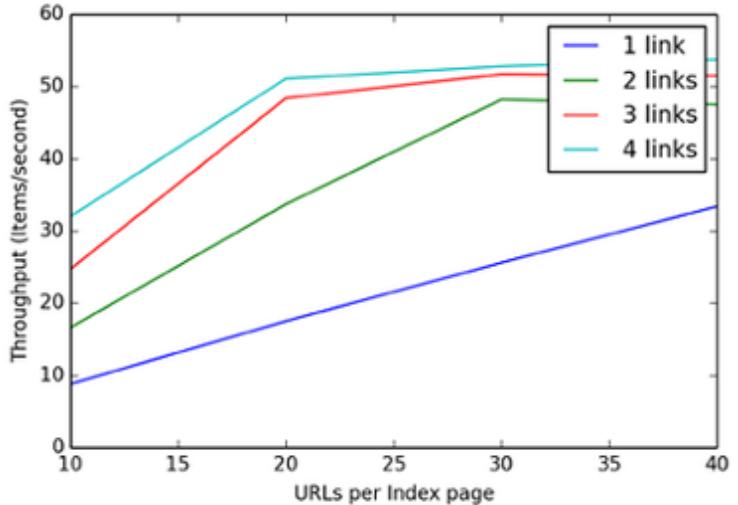
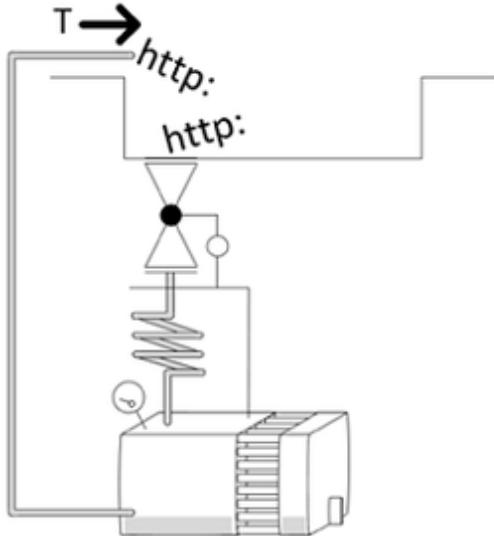
很明显，与之前的结果不同。下载器没有满负荷运行，吞吐量为 $T=N/S-t_{\text{start}}/t_{\text{stop}}=500/(32.2-3.1)=17$ 请求/秒。

讨论：d/load列可以确认下载器没有满负荷运行。这是因为没有足够的URL进入。抓取过程产生URL的速度慢于处理的速度。这时，每个索引页会产生20个URL+下一个索引页。吞吐量不可能超过每秒20个请求，因为产生URL的速度没有这么快。

解决：如果每个索引页有至少两个下一个索引页的链接，那么我们就可以加快产生URL的速度。如果可以找到能产生更多URL（例如50）的索引页面则会更好。通过模拟观察变化：

```
$ for details in 10 20 30 40; do for nxtlinks in 1 2 3 4; do
time scrapy crawl speed -s SPEED_TOTAL_ITEMS=500 -s SPEED_T_RESPONSE=1 \
-s CONCURRENT_REQUESTS=64 -s SPEED_START_REQUESTS_STYLE=UseIndex \
-s SPEED_DETAILS_PER_INDEX_PAGE=$details \
```

```
-s SPEED_INDEX_POINTAHEAD=$nxtlinks
done; done
```



在图12中，我们可以看到吞吐量是如何随每页URL数和索引页链接数变化的。初始都是线性变化，直到到达系统限制。你可以改变爬虫的规则进行试验。如果使用LIFO（默认项）规则，即先发出索引页请求最后收回，可以看到性能有小幅提高。你也可以将索引页的优先级设置为最高。两种方法都不会有太大的提高，但是你可以通过分别设置SPEED\_INDEX\_RULE\_LAST=1和SPEED\_INDEX\_HIGHER\_PRIORITY=1，进行试验。请记住，这两种方法都会首先下载索引页（因为优先级高），因此会在调度器中产生大量URL，这会提高对内存的要求。在完成索引页之前，输出的结果很少。索引页不多时推荐这种做法，有大量索引时不推荐这么做。

另一个简单但高效的方法是分享首页。这需要你使用至少两个首页URL，并且它们之间距离最大。例如，如果首页有100页，你可以选择1和51作为起始。爬虫这样就可以将抓取下一页的速度提高一倍。相似的，对首页中的商品品牌或其他属性也可以这么做，将首页大致分为两个部分。你可以使用-s SPEED\_INDEX\_SHARDS设置进行模拟：

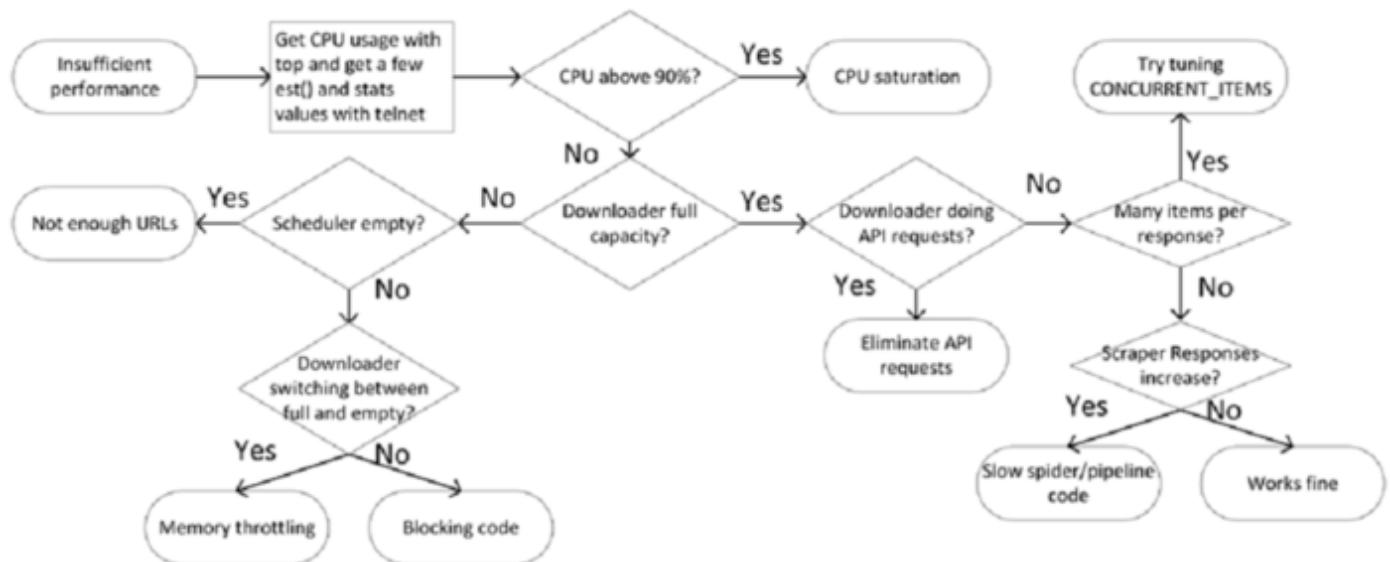
```
$ for details in 10 20 30 40; do for shards in 1 2 3 4; do
time scrapy crawl speed -s SPEED_TOTAL_ITEMS=500 -s SPEED_T_RESPONSE=1 \
-s CONCURRENT_REQUESTS=64 -s SPEED_START_REQUESTS_STYLE=UseIndex \
-s SPEED_DETAILS_PER_INDEX_PAGE=$details -s SPEED_INDEX_SHARDS=$shards
done; done
```

这次的结果比之前的方法要好，并且更加简洁。

## 解决问题的流程

总结一下，Scrapy的设计初衷就是让下载器作为瓶颈。使CONCURRENT\_REQUESTS从小开始，逐渐变大，直到发生以下的限制：

- CPU利用率 > 80-90%
- 源网站延迟急剧升高
- 抓取器的响应达到内存5Mb上限 同时，进行如下操作：
- 始终保持调度器（mqs/dqs）中有一定数量的请求，避免下载器是空的
- 不使用阻塞代码或CPU密集型代码



## 总结

在本章中，我们通过案例展示了Scrapy的架构是如何影响性能的。细节可能会在未来的Scrapy版本中变动，但是本章阐述的原理在相当长一段时间内可以帮助你理解以Twisted、Netty Node.js等为基础的异步框架。

谈到具体的Scrapy性能，有三个确定的答案：我不知道也不关心、我不知道但会查出原因，和我知道。本章已多次指出，“更多的服务器/内存/带宽”不能提高Scrapy的性能。唯一的方法是找到瓶颈并解决它。

在最后一章中，我们会学习如何进一步提高性能，不是使用一台服务器，而是在多台服务器上分布多个爬虫。

## 第11章 Scrapy分布式抓取和实时分析

我们已经学了很多东西。我们先学习了两种基础的网络技术，HTML和XPath，然后我们学习了使用Scrapy抓取复杂的网站。接着，我们深入学习了Scrapy的设置，然后又进一步深入学习了

Scrapy和Python的内部架构和Twisted引擎的异步特征。在上一章中，我们学习了Scrapy的性能和以及处理复杂的问题以提高性能。

在本章中，我将展示如何在多台服务器上进一步提高性能。我们会发现抓取通常是一个并行问题；因此，我们可以水平延展至多台服务器。为了这么做，我们会使用一个Scrapy中间件，我们还会使用Scrapyd，一个用来管理远程服务器爬虫的应用。它可以使我们像第6章那样进行抓取。

我们最后用Apache Spark对提取的数据进行实时分析。Spark一个非常流行的大数据处理框架。收集的数据越多、结果就变得越准确，我们使用Spark Streaming API展示结果。最后的结果展示了Python的强大和成熟，单单用Python的简明代码就全栈开发了从抓取到分析的全过程。

## 房子的标题如何影响价格？

---

我们要研究个问题是房子的标题和价格有什么关系。我们预计像“按摩浴缸”和“游泳池”可能和高价相关，而“打折”会和低价有关。将标题与地点结合，例如，可以根据地点和描述，实时判断哪个房子最划算。

我们想计算的就是特定名词对价格造成的偏移：

$$shift_{term} = \left( \frac{Price_{properties-with-term}}{Price_{properties-without-term}} - 1 \right) / Price$$

例如，如果平均租金是1000，我们观察到带有按摩浴缸的房子的平均价格是1300，没有的价格是\$995，因此按摩浴缸的偏移值为 $shift_{jacuzzi}=(1300-995)/1000=30.5\%$ 。如果一个带有按摩浴缸的房子的价格直逼平均价格高5%，那么它的价格就很划算。

因为名词效应会有累加，所以这个指标并不繁琐。例如，标题同时含有按摩浴缸和打折会有一个混合效果。我们收集分析的数据越多，估算就会越准确。稍后回到这个问题，接下来讲一个流媒体解决方案。

## Scrapyd

---

现在，我们来介绍Scrapyd。Scrapyd是一个应用，使用它，我们可以将爬虫附属到服务器上，并对抓取进行规划。我们来看看它的使用是多么容易，我们用第3章的代码，只做一点修改。

我们先来看Scrapyd的界面，在`http://localhost:6800/`。

The screenshot shows the ScrapyD web interface with several sections:

- Scrapyd**: A sidebar with links to [Jobs](#), [Items](#), [Logs](#), and [Documentation](#).
- Available projects: properties**: A list of projects.
- How to schedule a spider?**: Instructions and an example curl command: `curl http://localhost:6800/schedule.json -d project=default -d spider=somespider`.
- Directory listing for /items/properties/easy/**: A table showing item logs:
 

Filename	Size	Content type	Content encoding
c6582742a1ee11e59eca0242ac11000a.jl	548K	[text/plain]	
d4dfa6a1ee11e59eca0242ac11000a.jl	280K	[text/plain]	
- Directory listing for /logs/**: A table showing log files:
 

Filename	Size	Content type	Content encoding
properties/ [Directory]			
scrapyd.err	0B	[text/plain]	
scrapyd.log	10K	[text/plain]	
scrapyd.out	0B	[text/plain]	
- Directory listing for /logs/properties/easy/**: A table showing specific log files:
 

Filename	Size	Content type	Content encoding
c6582742a1ee11e59eca0242ac11000a.log	1M	[text/plain]	
d4dfa6a1ee11e59eca0242ac11000a.log	841K	[text/plain]	

你可以看到，它有几个部分，有Jobs、Items、Logs和Documentation。它还给出了如何规划抓取工作的API方法。

为了这么做，我们必须首先将爬虫部署到服务器上。第一步是修改scrapy.cfg，如下所示：

```
$ pwd
/root/book/ch03/properties
$ cat scrapy.cfg
...
[settings]
default = properties.settings
[deploy]
url = http://localhost:6800/
project = properties
```

我们要做的就是取消url的注释。默认的设置适合我们。现在，为了部署爬虫，我们使用scrapyd-client提供的工具scrapyd-deploy。scrapyd-client以前是Scrapy的一部分，但现在是一个独立的模块，可以用pip install scrapyd-client进行安装（开发机中已经安装了）：

```
$ scrapyd-deploy
Packing version 1450044699
Deploying to project "properties" in http://localhost:6800/addversion.
```

```
json
Server response (200):
{"status": "ok", "project": "properties", "version": "1450044699",
"spiders": 3, "node_name": "dev"}
```

部署好之后，就可以在Scrapyd的界面的Available projects看到。我们现在可以根据提示，在当前页提交一个任务：

```
$ curl http://localhost:6800/schedule.json -d project=properties -d spider=easy
{"status": "ok", "jobid": "d4df...", "node_name": "dev"}
```

如果我们返回Jobs，我们可以使用jobid schedule.json，它可以在之后用cancel.json取消任务：

```
$ curl http://localhost:6800/cancel.json -d project=properties -d job=d4df...
{"status": "ok", "prevstate": "running", "node_name": "dev"}
```

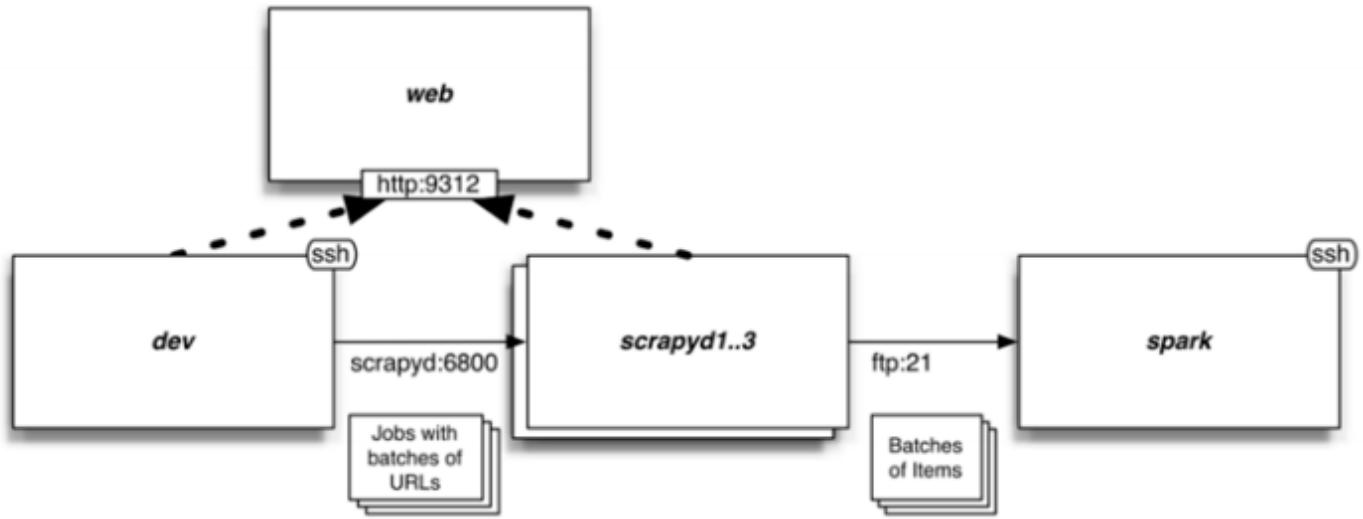
一定要取消进程，否则会浪费计算资源。

完毕之后，访问Logs，我们可以看到日志，在Items我们可以看到抓取过的items。这些数据会被周期清空以节省空间，所以一段时间后就会失效。

如果发生冲突或有其它理由的话，我们可以通过http\_port修改端口，它是Scrapyd的诸多设置之一。最好阅读文档<http://scrapyd.readthedocs.org/>，多了解下。我们的部署必须要设置的是max\_proc。如果使用默认值0，任务的并行数量最多可以是CPU核心的四位。因为我们可能会在虚拟机中运行多个Scrapyd服务器，我们将max\_proc设为4，可以允许4个任务同时进行。在真实环境中，使用默认值就可以。

## 分布式系统概述

设计这个系统对我是个挑战。我一开始添加了许多特性，导致复杂度升高，只有高性能的机器才能完成工作。然后，又不得不进行简化，既对硬件性能要求不那么高，也可以让本章的重点仍然是Scrapy。



最后，系统中会包括我们的开发机和几台服务器。我们用开发机进行首页的水平抓取，提取几批URL。然后用轮训机制将URL分发到Scrapyd的结点，并进行抓取。最后，通过FTP传递.jl文件和Items到运行Spark的服务器上。我选择FTP和本地文件系统，而不是HDFS或Apache Kafka，是因为FTP内存需求少，并且作为FEED\_URI被Scrapy支持。请记住，只要简单设置Scrapyd和Spark的配置，我们就可以使用亚马逊S3存储这些文件，获得冗余度和可伸缩性等便利，而不用再使用其它技术。

**笔记：**FTP的缺点之一是，上传过程可能会造成文件不完整。为了避免这点，一旦上传完成，我们便使用Pure-FTPD和调回脚本将文件上传到/root/items。

每过几秒，Spark都读一下目录/root/items，读取任何新文件，取一个小批次进行分析。我们使用Spark是因为它支持Python作为编程语言，也支持流分析。到现在，我们使用的爬虫都比较短，实际中有的爬虫是24小时运行的，不断发出数据流并进行分析，数据越多，分析的结果越准确。我们就是要用Spark进行这样的演示。

**笔记：**除了Spark和Scrapy，你还可以使用MapReduce，Apache Storm或其它框架。

在本章中，我们不向数据库中插入items。我们在第9章中用的方法也可以在这里使用，但是性能很糟。很少有数据库喜欢每秒被pipelines写入几千个文件。如果想进行写入的话，应该用Spark专用的方法，即批次导入Items。你可以修改我们Spark的例子，向任何数据库进行批次导入。

还有，这个系统的弹性很差。我们假设每个结点都是健康的，任何一个损坏的话，也不会对总系统造成影响。Spark提供高可用性的弹性配置。Scrapy不提供此类内建的功能，除了Scrapyd的“持续排队”功能，即当结点恢复时，可以继续失败的任务。这个功能不一定对你有用。如果弹性对你十分重要，你就得搭建一个监督系统和分布式排队方案（例如，基于Kafka或RabbitMQ），可以重启失败的任务。

## 修改爬虫和中间件

为了搭建这个系统，我们要稍稍修改爬虫和中间件。更具体地，我们要做如下工作：

- 微调爬虫，使抓取索引页的速度达到最大
- 写一个中间件，可以将URL批次发送给scrapyd服务器。
- 使用相同的中间件，使系统启动时就可以将URL分批

我们尽量用简明的方式来完成这些工作。理想状态下，整个过程应该对底层的爬虫代码简洁易懂。这是一个底层层面的要求，通过破解爬虫达到相同目的不是好主意。

## 抓取共享首页

第一步是优化抓取首页的速度，速度越快越好。开始之前，先明确一下目的。假设爬虫的并发数是16，源网站的延迟大概是0.25秒。这样，最大吞吐量是 $16/0.25=64$ 页/秒。首页有50000个子页，每个索引页有30个子页，那就有1667个索引页。预计下载整个首页需要， $1667/64=26$ 秒。

将第3章中的爬虫重命名为easy。我们将首先进行垂直抓取的Rule（含有`callback='parse_item'`的一项）注释掉，因为现在只想抓取索引页。

提示：本章的代码位于目录ch11。

在进行优化之前，我们让scrapy crawl只抓取10个页面，结果如下：

```
$ ls
properties  scrapy.cfg
$ pwd
/root/book/ch11/properties
$ time scrapy crawl easy -s CLOSESPIDER_PAGECOUNT=10
...
DEBUG: Crawled (200) <GET ...index_00000.html> (referer: None)
DEBUG: Crawled (200) <GET ...index_00001.html> (referer: ...index_00000.
html)
...
real  0m4.099s
```

如果10个页面用时4秒，26秒内是不可能完成1700个页面的。通过查看日志，我们看到每个索引页都来自前一个页面，也就是说，任何时候最多是在处理一个页面。实际上，并发数是1。我们需要将其并行化，使达到并发数16。我们将索引页相互共享，即URL互相连接，再加入一些其他的链接，以免爬虫中没有URL。我们将首页分为20个部分。实际上，任何大于16的数，都可以提速，但是一旦超过20，速度反而会下降。我们用下面的方法计算每个部分的起始索引页：

```
>>> map(lambda x: 1667 * x / 20, range(20))
```

```
[0, 83, 166, 250, 333, 416, 500, ... 1166, 1250, 1333, 1416, 1500, 1583]
```

据此，设置start\_URL如下：

```
start_URL = ['http://web:9312/properties/index_%05d.html' % id  
            for id in map(lambda x: 1667 * x / 20, range(20))]
```

这可能会和你的情况不同，所以就不做美化了。将并发数(CONCURRENT\_REQUESTS, CONCURRENT\_REQUESTS\_PER\_DOMAIN)设为16，再次运行爬虫，运行如下：

```
$ time scrapy crawl easy -s CONCURRENT_REQUESTS=16 -s CONCURRENT_  
REQUESTS_PER_DOMAIN=16  
...  
real 0m32.344s
```

结果接近了我们的目标。下载速度是1667页面/32秒=52页面/秒，也就是说，每秒可以产生 $52 \times 30 = 1560$ 个子页面。我们现在可以注释掉垂直抓取的Rule，将文件保存成一个爬虫。我们不需要进一步修改爬虫代码，而是用一个功能强大的中间件继续来做。如果只用开发机运行爬虫，假设可以像抓取索引页一样抓取子页，可以在 $50000 / 52 = 16$ 分钟内完成抓取。

这里有两个要点。在学习完第10章之后，我们在做的都是工程项目。我们可以想方设法计算出系统确切的性能。第二点是，抓取索引页会产生子页，但实际的吞吐量不大。如果产生URL的速度快过scrapyd处理URL的速度，URL就会在scrapyd排队。或者，如果产生URL的速度太慢，scrapyd就会空闲。

## 批次抓取URL

现在来处理子页面的URL，并把它们分批，然后直接发送给scrapyd，而不是继续抓取。

如果检查Scrapy的架构，我们可以明白这么做就是为了做一个中间件，它可以执行process\_spider\_output()，在Requests到达下载器之前就可以进行处理或取消。我们限定中间件只支持CrawlSpider的爬虫，并且只支持简单的GET请求。如果要提高复杂度，例如，POST或认证请求，我们必须开发更多的功能，以传递参数、头文件、每个批次进行重新登陆。

打开Scrapy的GitHub，查看SPIDER\_MIDDLEWARES\_BASE设置，看看能否重利用哪个程序。Scrapy 1.0有以下中间件：HttpErrorMiddleware、OffsiteMiddleware、RefererMiddleware、UrlLengthMiddleware和DepthMiddleware。我们看到OffsiteMiddleware（只有60行）好像使我

们需要的。它根据爬虫属性allowed\_domains限定URL。我们可以用相同的方法吗？不是丢弃URL，我们转而将它们分批，发送给scrapyds。我们确实可以这么做，部分代码如下：

```
def __init__(self, crawler):
    settings = crawler.settings
    self._target = settings.getint('DISTRIBUTED_TARGET_RULE', -1)
    self._seen = set()
    self._URL = []
    self._batch_size = settings.getint('DISTRIBUTED_BATCH_SIZE', 1000)
    ...
def process_spider_output(self, response, result, spider):
    for x in result:
        if not isinstance(x, Request):
            yield x
        else:
            rule = x.meta.get('rule')
            if rule == self._target:
                self._add_to_batch(spider, x)
            else:
                yield x
def _add_to_batch(self, spider, request):
    url = request.url
    if not url in self._seen:
        self._seen.add(url)
        self._URL.append(url)
        if len(self._URL) >= self._batch_size:
            self._flush_URL(spider)
```

process\_spider\_output()处理Item和Request。我们只需要Request，其它就不考虑了。如果查看CrawlSpider的源代码，我们看到将Request/Response映射到Rule的方式是用一个meta dict中的名为“rule”的整数字段。我们检查这个数字，如果它指向我们想要的Rule（DISTRIBUTED\_TARGET\_RULE设置），我们调用\_add\_to\_batch()，将它的URL添加到这个批次里面。我们然后取消这个Request。我们接着产生出其他的请求，例如下一页的链接，不进行改动。The `addto_batch()`方法起到去重的作用。但是，我们前面描述的碎片化过程，意味着有的URL可能要提取两次。我们使用\_seen set检测并去除重复项。然后将这些URL添加到\_URL列表，如果它的大小超过了\_batch\_size（根据DISTRIBUTED\_BATCH\_SIZE设置），就会调用\_flush\_URL()。这个方法提供了一下功能：

```
def __init__(self, crawler):
    ...
    self._targets = settings.get("DISTRIBUTED_TARGET_HOSTS")
    self._batch = 1
    self._project = settings.get('BOT_NAME')
```

```

self._feed_uri = settings.get('DISTRIBUTED_TARGET_FEED_URL', None)
self._scrapyd_submits_to_wait = []
def _flush_URL(self, spider):
    if not self._URL:
        return
    target = self._targets[(self._batch-1) % len(self._targets)]
    data = [
        ("project", self._project),
        ("spider", spider.name),
        ("setting", "FEED_URI=%s" % self._feed_uri),
        ("batch", str(self._batch)),
    ]
    json_URL = json.dumps(self._URL)
    data.append(("setting", "DISTRIBUTED_START_URL=%s" % json_URL))
    d = treq.post("http://%s/schedule.json" % target,
                  data=data, timeout=5, persistent=False)
    self._scrapyd_submits_to_wait.append(d)
    self._URL = []
    self._batch += 1

```

首先，它使用了批次计数器（\_batch）来决定发向哪个scrapyd服务器。可用服务器保存在\_targets（见DISTRIBUTED\_TARGET\_HOSTS设置）。我们然后向scrapyd的schedule.json做一个POST请求。这比之前用过的curl方法高级，因为它传递了经过仔细选择的参数。基于这些常熟，scrapyd就规划了一次抓取，如下所示：

```

scrapy crawl distr \
-s DISTRIBUTED_START_URL='[".../property_00000.html", ... ]' \
-s FEED_URI='ftp://anonymous@spark/%(batch)s_%(name)s_%(time)s.jl' \
-a batch=1

```

除了项目和爬虫的名字，我们想爬虫传递了一个FEED\_URI设置。它的值是从DISTRIBUTED\_TARGET\_FEED\_URL得到的。

因为Scrapy支持FTP，我们可以让scrapyd用一个匿名FTP将抓取的Item文件上传到Spark服务器。它的格式包括爬虫的名字（%(name)s）和时间（%(time)s）。如果只有这两项的话，那么同一时间创建出来的两个文件就会有冲突。为了避免覆盖，我们加入一个参数%(batch)s。Scrapy默认是不知道批次的，所以我们必须给设定一个值。scrapyd的schedule.json API的特点之一是，每个不是设置的参数或已知的参数都被传递给了爬虫。默认时，爬虫的参数成为了爬虫的属性，然后在爬虫的属性中寻找未知的FEED\_URI参数。因此，将一批参数传递给schedule.json，我们就可以在FEED\_URI中使用它，以避免冲突。

最后是将DISTRIBUTED\_START\_URL和这一批次的子页URL编译为JSON，因为JSON是最简洁的文本格式。

笔记：用命令行将大量数据传递到Scrapy并不可取。如果你想将参数存储到数据库（例如Redis），只传递给Scrapy一个ID。这么做的话，需要小幅修改\_flush\_URL()和process\_start\_requests()。

我们用treq.post()来做POST请求。Scrapyd处理持续连接并不好，因此我们用persistent=False取消它。我们还设置了一个5秒的暂停。这个请求的的延迟项被保存在\_scrapyd\_submits\_to\_wait列表。要关闭这个函数，我们重置\_URL列表，并加大当前的\_batch。

奇怪的是，关闭操作中会出现许多方法，如下所示：

```
def __init__(self, crawler):
    ...
    crawler.signals.connect(self._closed, signal=signals.spider_closed)
@defer.inlineCallbacks
def _closed(self, spider, reason, signal, sender):
    # Submit any remaining URL
    self._flush_URL(spider)
    yield defer.DeferredList(self._scrapyd_submits_to_wait)
```

调用\_closed()可能是因为我们按下了Ctrl + C或因为抓取结束。两种情况下，我们不想失去任何最后批次的还未发送的URL。这就是为什么在\_closed()中，第一件事是调用\_flush\_URL(spider)加载最后的批次。第二个问题是，因为是非阻塞的，停止抓取时，[treq.post\(\)](#)可能结束也可能没结束。为了避免丢失最后批次，我们要使用前面提到过的scrapyd\_submits\_to\_wait列表，它包括所有的[treq.post\(\)](#)延迟项。我们使用defer.DeferredList()等待，直到全部完成。因为\_closed()使用了@defer.inlineCallbacks，当所有请求完成时，我们只yield它并继续。

总结一下，DISTRIBUTED\_START\_URL设置中的批次URL会被发送到scrapyd，scrapyd上面运行着相同的爬虫。很明显，我们需要使用这个设置以启动start\_URL。

## 从settings启动URL

中间件还提供了一个process\_start\_requests()方法，使用它可以处理爬虫提供的start\_requests。检测是否设定了DISTRIBUTED\_START\_URL，设定了的话，用JSON解码，并使用它的URL产生相关的请求。对于这些请求，我们设定CrawlSpider的\_response\_downloaded()方法作为回调函数，再设定参数meta['rule']，以让恰当的Rule处理响应。我们查看Scrapy的源码，找到CrawlSpider创建请求的方法，并依法而做：

```

def __init__(self, crawler):
    ...
    self._start_URL = settings.get('DISTRIBUTED_START_URL', None)
    self.is_worker = self._start_URL is not None
def process_start_requests(self, start_requests, spider):
    if not self.is_worker:
        for x in start_requests:
            yield x
    else:
        for url in json.loads(self._start_URL):
            yield Request(url, spider._response_downloaded,
                          meta={'rule': self._target})

```

中间件就准备好了。我们在settings.py进行设置以启动它：

```

SPIDER_MIDDLEWARES = {
    'properties.middlewares.Distributed': 100,
}
DISTRIBUTED_TARGET_RULE = 1
DISTRIBUTED_BATCH_SIZE = 2000
DISTRIBUTED_TARGET_FEED_URL = ("ftp://anonymous@spark/"
                               "%(batch)s_%(name)s_%(time)s.jl")
DISTRIBUTED_TARGET_HOSTS = [
    "scrapyd1:6800",
    "scrapyd2:6800",
    "scrapyd3:6800",
]

```

有人可能认为DISTRIBUTED\_TARGET\_RULE不应该作为设置，因为它会使爬虫差异化。你可以认为这是个默认值，你可以在你的爬虫中使用属性custom\_settings覆盖它，例如：

```

custom_settings = {
    'DISTRIBUTED_TARGET_RULE': 3
}

```

我们的例子并不需要这么做。我们可以做一个测试运行，只抓取一个页面：

```

$ scrapy crawl distr -s \
DISTRIBUTED_START_URL='["http://web:9312/properties/property_000000.html"]'

```

这个成功之后，我们进一步，抓取一个页面之后，用FTP将它传送到Spark服务器：

```
scrapy crawl distr -s \
DISTRIBUTED_START_URL='["http://web:9312/properties/property_00000.html"]' \
-s FEED_URI='ftp://anonymous@spark/%(batch)s_%(name)s_%(time)s.jl' -a batch=12
```

用ssh连接Spark服务器，你可以看到一个文件，例如/root/items下的12\_distr\_date\_time.jl。这个中间件的例子可以让你完成scrapyd的分布式抓取。你可以将它当做起点，进行改造。你可能要做如下修改：

- 爬虫的类型。除了CrawlSpider，你必须让爬虫用恰当的meta标记分布式的请求，用惯用命名法执行调回。
- 向scrapyd传递URL的方式。你可能想限定域名，减少传递的量。例如，你只想传递IDs。
- 你可以用分布式排队方案，让爬虫可以从失败恢复，让scrapyd执行更多的URL批次。
- 你可以动态扩展服务器的规模，以适应需求。

## 将项目部署到scrapyd服务器

为了将爬虫附属到三台scrapyd服务器上，我们必须将它们添加到scrapy.cfg文件。文件上的每个[deploy:target-name]定义了一个新的部署目标：

```
$ pwd
/root/book/ch11/properties
$ cat scrapy.cfg
...
[deploy:scrapyd1]
url = http://scrapyd1:6800/
[deploy:scrapyd2]
url = http://scrapyd2:6800/
[deploy:scrapyd3]
url = http://scrapyd3:6800/
```

你可以用scrapyd-deploy -l查询可用的服务器：

```
$ scrapyd-deploy -l
scrapyd1          http://scrapyd1:6800/
scrapyd2          http://scrapyd2:6800/
scrapyd3          http://scrapyd3:6800/
```

用scrapyd-deploy 进行部署：

```
$ scrapyd-deploy scrapyd1
Packing version 1449991257
Deploying to project "properties" in http://scrapyd1:6800/addversion.json
Server response (200):
{"status": "ok", "project": "properties", "version": "1449991257",
 "spiders": 2, "node_name": "scrapyd1"}
```

这个过程会产生一些新的目录和文件（build、project.egg-info、setup.py），可以删掉。其实，scrapyd-deploy做的就是打包你的项目，并用addversion.json，传递到目标服务器上。

之后，如果我们用scrapyd-deploy -L查询服务器，我们可以确认项目被成功部署了：

```
$ scrapyd-deploy -L scrapyd1
properties
```

我还用touch在项目的目录创建了三个空文件夹，scrapyd1-3。这样可以将scrapyd的名字传递给下面的文件，同时也是服务器的名字。然后可以用bash loop将其部署服务器：for i in scrapyd\*; do scrapyd-deploy \$i; done。

## 创建自定义监视命令

如果你想在多台scrapyd服务器上监视抓取的进程，你必须亲自编写程序。这是一个练习所学知识的好机会，写一个原生的Scrapy命令，scrapy monitor，用它监视一组scrapyd服务器。文件命名为monitor.py，在settings.py中添加COMMANDS\_MODULE = 'properties.monitor'。快速查看scrapyd的文档，listjobs.json API给我们提供了关于任务的信息。如果我们想找到给定目标的根URL，我们可以断定，它只能是在scrapyd-deploy的代码中。如果查看<https://github.com/scrapy/scrapyd-client/blob/master/scrapyd-client/scrapyd-deploy>，我们可以发现一个\_get\_targets()函数（执行它不会添加许多值，所以略去了），它可以给出目标的名字和根URL。我们现在就可以执行命令的第一部分了，如下所示：

```
class Command(ScrapyCommand):
    requires_project = True
    def run(self, args, opts):
        self._to_monitor = {}
```

```

for name, target in self._get_targets().iteritems():
    if name in args:
        project = self.settings.get('BOT_NAME')
        url = target['url'] + "listjobs.json?project=" + project
        self._to_monitor[name] = url
l = task.LoopingCall(self._monitor)
l.start(5) # call every 5 seconds
reactor.run()

```

这段代码将名字和想要监视的API的终点提交给dict `_tomonitor`。我们然后使用`task.LoopingCall()`规划向`_monitor()`方法发起递归调用。`_monitor()`使用`treq`和`deferred`, 我们使用`@defer.inlineCallbacks`对它进行简化。方法如下（省略了一些错误处理和代码美化）：

```

@defer.inlineCallbacks
def _monitor(self):
    all_deferreds = []
    for name, url in self._to_monitor.iteritems():
        d = treq.get(url, timeout=5, persistent=False)
        d.addBoth(lambda resp, name: (name, resp), name)
        all_deferreds.append(d)
    all_resp = yield defer.DeferredList(all_deferreds)
    for (success, (name, resp)) in all_resp:
        json_resp = yield resp.json()
        print "%-20s running: %d, finished: %d, pending: %d" %
            (name, len(json_resp['running']),
             len(json_resp['finished']), len(json_resp['pending']))

```

这几行代码包括了目前我们学过的所有Twisted方法。我们使用`treq`调用scrapyd的API和`defer.DeferredList`, 立即处理所有的响应。当`all_resp`有了所有结果之后, 我们重复这个过程, 取回它们的JSON对象。`treq Response.json()`方法返回延迟项, 而不是实际值, 以与后续的实际值继续任务。我们最后打印出结果。JSON响应的列表信息包括悬挂、运行中、结束的任务, 我们打印出它的长度。

## 用Apache Spark streaming计算偏移值

我们的Scrapy系统现在就功能完备了。让我们来看看Apache Spark的使用。

本章开头的  $Shift_{term}$  公式很简洁，但是不能有效执行。我们可以用两项轻易计算  $\overline{Price}$ ，用  $2 \cdot n_{term}$  个项计算  $\overline{Price}_{with}$ ，对于每个新的价格，必须要更新这四个参数。计算  $\overline{Price}_{with}$  很复杂，因为对于每个新价格，要更新  $2 \cdot (n_{term} - 1)$  项。例如，我们需要向每个  $\overline{Price}_{without}$  加入按摩浴缸的价格。当数据量很大时，这就使得计算变得不可行。

要解决这个问题，我们想到将具有某事物的房子和没有某事物的房子的价格相加，就是所有房子的价格总和，即  $\sum Price = \sum Price|_{with} + \sum Price|_{without}$ 。因此，没有某事物的房子的平均价格可以如下计算：

$$\overline{Price}|_{without} = \frac{\sum Price|_{without}}{n_{without}} = \frac{\sum Price - \sum Price|_{with}}{n - n_{with}}$$

使用这个公式，偏移值公式为：

$$shift_{term} = \left( \frac{\sum Price|_{with}}{n_{with}} - \frac{\sum Price - \sum Price|_{with}}{n - n_{with}} \right) \Bigg/ \frac{\sum Price}{n}$$

让我来看如何执行。请记住这不是Scrapy代码，所以看起来会觉得陌生，但是是可以看懂的。你可以在 boostwords.py 文件找到这个应用，这个文件包括了复杂的测试代码，可以忽略。它的主要代码如下：

```
# Monitor the files and give us a DStream of term-price pairs
raw_data = raw_data = ssc.textFileStream(args[1])
word_prices = preprocess(raw_data)
# Update the counters using Spark's updateStateByKey
running_word_prices = word_prices.updateStateByKey(update_state_
function)
# Calculate shifts out of the counters
shifts = running_word_prices.transform(to_shifts)
# Print the results
shifts.foreachRDD(print_shifts)
```

Spark 使用 DStream 代表数据流。textFileStream() 方法监督文件系统的一个目录，当检测到新文件时，就传出来。我们的 preprocess() 函数将它们转化为 term/price 对。我们用 update\_state\_function() 函数和 Spark 的 updateStateByKey() 方法累加这些 term/price 对。我们最后通过运行 to\_shifts() 计算偏移值，并用 print\_shifts() 函数打印出极值。大多我们的函数修改不大，只是高效重塑了数例据。例外的是 shifts() 函数：

```
def to_shifts(word_prices):
    (sum0, cnt0) = word_prices.values().reduce(add_tuples)
```

```
avg0 = sum0 / cnt0
def calculate_shift((isum, icnt)):
    avg_with = isum / icnt
    avg_without = (sum0 - isum) / (cnt0 - icnt)
    return (avg_with - avg_without) / avg0
return word_prices.mapValues(calculate_shift)
```

这段代码完全是按照公式做的。尽管很简单，Spark的mapValues()可以让calculate\_shift在Spark服务器上用最小开销高效运行。

## 进行分布式抓取

我进行四台终端进行抓取。我想让这部分尽量独立，所以我还提供了vagrant ssh命令，可以在终端使用。



使用四台终端进行抓取

用终端1来检测集群的CPU和内存的使用。这可以确认和修复问题。设置方法如下：

```
$ alias provider_id="vagrant global-status --prune | grep 'docker-provider' | awk '{print \$1}'"
$ vagrant ssh $(provider_id)
$ docker ps --format "{{.Names}}" | xargs docker stats
```

前两行可以让我们用ssh打开docker provider VM。如果没有使用VM，只在docker Linux运行，我们只需要最后一行。

终端2用作诊断，如下运行 scrapy monitor：

```
$ vagrant ssh
$ cd book/ch11/properties
$ scrapy monitor scrapyd*
```

使用scrapyd\*和空文件夹，空文件夹名字是scrapy monitor，这会扩展到scrapy monitor scrapyd1 scrapyd2 scrapyd3。

终端3，是我们启动抓取的终端。除此之外，它基本是闲置的。开始一个新的抓取，我们操作如

下：

```
$ vagrant ssh  
$ cd book/ch11/properties  
$ for i in scrapyd*; do scrapyd-deploy $i; done  
$ scrapy crawl distr
```

最后两行很重要。首先，我们使用一个for循环和scrapyd-deploy，将爬虫部署到服务器上。然后我们用scrapy crawl distr开始抓取。我们随时可以运行小的抓取，例如，scrapy crawl distr -s CLOSESPIDER\_PAGECOUNT=100，来抓取100个索引页，它会产生大概3000个子页。终端4用来连接Spark服务器，我们用它进行实时分析：

```
$ vagrant ssh spark  
$ pwd  
/root  
$ ls  
book items  
$ spark-submit book/ch11/boostwords.py items
```

只有最后一行重要，它运行了boostwords.py，将本地items目录传给了监视器。有时，我还使用watch ls -1 items来监视item文件。

到底哪个词对价格的影响最大呢？这个问题留给读者。

## 系统性能

系统的性能极大地依赖于硬件、CPU的数量、虚拟机分配内存的大小。在真实情况下，我们可以进行水平扩展，使抓取提速。

理论最大吞吐量是3台服务器4个CPU16并发数\*4页/秒=768页/秒。

实际中，使用分配了4G内存、8CPU的虚拟机的Macbook Pro，2分40秒内下载了50000条URL，即315页/秒。在一台亚马逊EC2 m4.large，它有2个vCPUs、8G内存，因为CPU频率低，用时6分12秒，即134页/秒。在一台台亚马逊EC2 m4.4xlarge，它有16个vCPUs、64G内存，用时1分44秒，即480页/秒。在同一台机器上，我将scrapyd的数量提高到6（修改Vagrantfile、scrapy.cfg和settings.py），用时1分15秒，即667页/秒。在最后的例子中，网络服务器似乎是瓶颈。

实际和理论计算存在差距是合理的。我们的粗略计算中没有考虑许多小延迟。尽管我们声明了每个页有250ms的延迟，我们在前几章已经看到，实际延迟要更高，这是因为我们还有额外的

Twisted和操作系统延迟。还有开发机向scrapyd传递URL的时间，FTP向Spark传递Items的时间，还有scrapyd发现新文件和规划任务的时间（平均要2.5秒，根据scrapyd的poll\_interval设置）。还没计算开发机和scrapyd的启动时间。如果不能确定可以提高吞吐量的话，我是不会试图改进这些延迟的。我的下一步是扩大抓取的规模，比如500000个页面、网络服务器的负载均衡，在扩大的过程中发现新的挑战。

## 要点

---

本章的要点是，如果要进行分布式抓取，一定要使用大小合适的批次。

取决于源网站的响应速度，你可能有数百、数千、上万个URL。你希望它们越大越好（在几分钟的水平），这样就可以分摊启动的费用。另一方面，你也不希望它们太大，以免造成机器故障。在一个有容错的分布式系统中，你需要重试失败的批次，而且重试不要浪费太多时间。

## 总结

---

希望你能喜欢这本关于Scrapy的书。现在你对Scrapy应该已经有深入的了解了，并可以解决简单或复杂的问题了。你还学到了Scrapy复杂的结构，以及如何发挥出它的最大性能。通过抓取，你可以在应用中使用庞大的数据资源。我们已经看到了如何在移动应用中使用Scrapy抓取的数据并进行分析。希望你能用Scrapy做出更多强大的应用，为世界做出贡献。祝你好运！