$t_{overhead}$ • $x + t_{start/stop}$，其中 x = N / CONCURRENT_REQUESTS 和 $y = t_{job}$ • $x + t_{response}$）。使用最小二乘法（Excel 函数为 LINEST）和前面的数据，我们可以计算得到 $t_{overhead}$ = 6ms，而 $t_{start/stop}$ = 3.1s。$t_{overhead}$ 是一个很小的数值，而启动时间却非常显著，不过它支持了数千个 URL 的长时间运行。因此，我们将使用一个非常有用的公式，以请求数/秒为单位近似系统的吞吐量，如下所示。

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

通过运行 N 个请求的长时间任务，我们可以测量出 t_{job} 的汇总时间，然后直接计算 T。

10.5 解决性能问题

现在我们应当对系统预期拥有的性能是什么有了充分的了解，接下来看一下如果没有得到想要的性能时应当如何操作。我们将通过探讨具体症状来展示不同的问题案例，执行示例爬虫进行复现，探讨根本原因，最终提供解决问题的操作。案例展示的顺序从系统顶层问题逐步到低层次的 Scrapy 技术细节。这就意味着更普遍的案例可能会出现在没那么常见的案例之后。在探索你的性能问题之前，请完整阅读本章全部内容。

10.5.1 案例 #1：CPU 饱和

症状：在某些情况下，你增加了并发级别，但没有得到性能提升。当降低并发级别时，一切工作再次回归预期（见图 10.6）。你的下载器可以被充分利用，但是似乎每个请求的平均时间出现了激增。当在 UNIX/Linux 系统中使用 `top` 命令、在 Power Shell 中使用 `ps` 命令或在 Windows 中使用任务管理器查看 CPU 负载如何时，会发现 CPU 负载非常高。

示例：假设运行了如下命令。

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

你得到了其抓取 5000 个 URL 的时间。在表 10.2 中，期望值一列是基于前面得到的

异步社区会员 Kensuke(18319066421) 专享 尊重版权

公式计算所得，而 CPU 负载是通过 top 命令观察得到的（可以在开发机中使用第二个终端运行该命令）。

表 10.2

CONCURRENT_REQUESTS	期望值（秒）	实际值（秒）	期望值与实际值的百分比	CPU 负载
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%

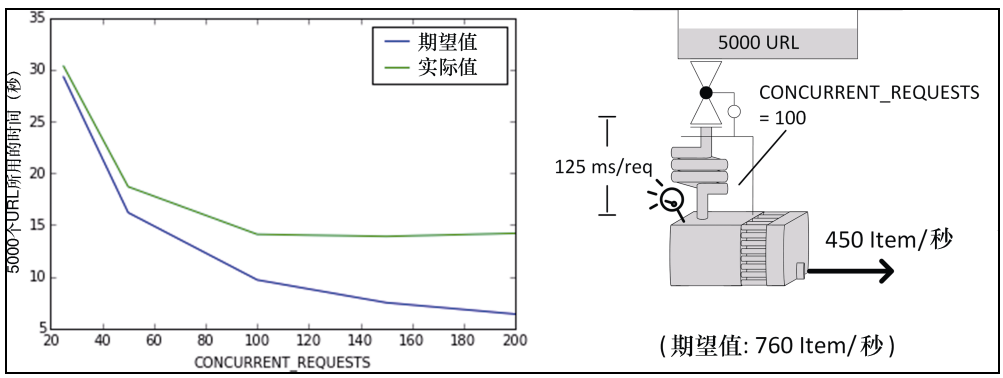


图 10.6　当并发增长到一定程度后，性能趋于平缓

在我们的实验中，由于几乎不执行任何处理，因此能够得到高并发。而在一个更复杂的系统中，很可能会更早地看到该行为。

讨论：Scrapy 重度使用单一线程，当达到很高级别的并发时，CPU 可能会成为瓶颈。假设不使用任何线程池，那么 Scrapy 应当使用的 CPU 负载建议在 80%～90%。请记住你可能在其他系统资源上遇到相似的问题，比如网络带宽、内存或磁盘吞吐量，不过这些都很少见，并且会落入通用系统的管理范畴，因此就不在这里进一步强调了。

解决方案：通常假设你的代码是有效的。你可以通过在同一台服务器上运行多个 Scrapy 爬虫，以使总计并发超过 CONCURRENT_REQUESTS。这可以帮助你利用更多

异步社区会员 Kensuke(18319066421) 专享 尊重版权

可用核心，尤其是当管道的其他服务或其他线程不使用它们的时候。如果需要更多的并发，可以使用多台服务器（参见第 11 章），这种情况下可能还需要更多可用的资金、网络带宽以及磁盘吞吐量。始终检查 CPU 利用率是你的首要约束。

10.5.2 案例 #2：代码阻塞

症状：你所观察到的行为无法说通。和期望值相比，系统非常慢，并且奇怪的是，即使当你改变 CONCURRENT_REQUESTS 的值时，速度也没有显著变化（见图 10.7）。下载器看起来总是空的（少于 CONCURRENT_REQUESTS），而抓取程序却有不少响应。

示例：你可以使用两个基准设置：SPEED_SPIDER_BLOCKING_DELAY 和 SPEED_PIPELINE_BLOCKING_DELAY（它们具有相同的效果），对每个响应启用一个 100ms 的阻塞。在给定并发级别时，我们期望 100 个 URL 应当花费 2～3 秒，但无论 CONCURRENT_REQUESTS 的值是多少，我们总是需要花费大约 13 秒的时间（见表 10.3）。

```
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
```

表 10.3

CONCURRENT_REQUESTS	总时间（秒）
16	13.9
32	13.2
64	12.9

讨论：任何阻塞代码都会立即抵消掉 Scrapy 的并发性，本质上相当于设置 CONCURRENT_REQUESTS = 1。根据上面的简单公式，100URL • 100ms（阻塞延迟）= 10 秒 + $t_{start/stop}$，充分解释了我们所看到的延迟。

无论阻塞代码是在管道中还是在爬虫中，你都会发现抓取程序可以被充分利用，但其前后的模块都是空的。这看起来违背了前面讲过的管道的物理现象，不过由于我们已经不再拥有一个并发系统了，所以管道规则不再适用。该错误非常容易发生（比如使用

异步社区会员 Kensuke(18319066421) 专享 尊重版权

阻塞 API），你一定会在某一时刻出现该错误。你会注意到类似的讨论同样适用于复杂代码的计算。你应当为此类代码使用多线程，正如我们在第 9 章中所看到的；或者是在 Scrapy 之外进行批量处理，我们将会在第 11 章中看到一个相关示例。

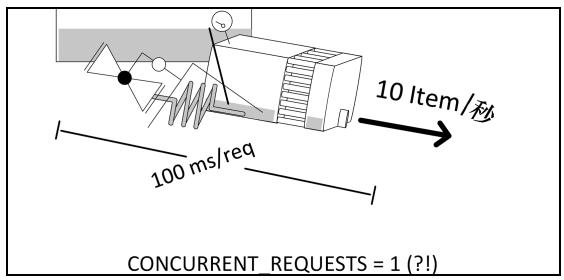


图 10.7　阻塞代码以不可预测的方式使并发无效

解决方案：将假设你继承了基代码，并且不清楚阻塞代码位于何处。如果该系统在没有任何管道的情况下仍然可以工作，那么禁用这些管道，并检查是否仍存在奇怪的行为。如果仍存在，那么阻塞代码位于爬虫中。如果不再存在，那么依次启用管道，观察问题是否开始出现。如果该系统在缺少任何运行中的模块的情况下无法正常运转，那么可以在每个管道阶段的功能之间添加一些日志消息（或插入虚拟管道打印时间戳）。通过检查日志，可以轻松检测出系统在什么地方花费了最多的时间。如果希望有一个更加长期/可复用的解决方案，可以使用虚拟管道跟踪你的请求，在 `Request` 的 `meta` 字段中为每个阶段添加时间戳。最后，hook 到 `item_scraped` 信号，并记录时间戳日志。一旦你发现阻塞代码，则应将其转换为 Twisted/异步代码，或使用 Twisted 的线程池。如果想要查看该转换的效果，可以将 `SPEED_PIPELINE_BLOCKING_DELAY` 替换为 `SPEED_PIPELINE_ASYNC_DELAY`，重新运行前面的示例。性能的变化将十分惊人。

10.5.3　案例 #3：下载器中的“垃圾”

症状：你得到的吞吐量低于预期。下载器看起来有时会有比 `CONCURRENT_ REQUESTS` 更多的请求。

示例：模拟以 0.25 秒响应时间的情况下载 1000 个页面。按照默认的 16 个并发，根据公式需要花费大约 19 秒的时间。我们使用一个管道，用 `crawler.engine.download()`

异步社区会员 Kensuke(18319066421) 专享 尊重版权

制造到伪造 API 的额外 HTTP 请求，其响应时间在 1 秒之内。你可以通过 http://localhost:9312/benchmark/ar:1/api?text=hello 进行尝试（见图 10.8）。让我们运行一个爬取程序。

```
$ 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
...
s/edule  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
```

非常奇怪！我们的任务不但花费了预期的 3 倍时间，还超出了下载器定义的 CONCURRENT_REQUESTS 所设定的 16 个活跃请求数（d/load）。下载器显然是瓶颈，因为它在超负荷工作。我们重新运行爬取程序，并在另一个控制台中打开到 Scrapy 的 telnet 连接。之后，就可以查看下载器中有哪些请求是活跃的了。

```
$ telnet localhost 6023
>>> engine.downloader.active
set([<POST http://web:9312/ar:1/ti:1000/rr:0.25/benchmark/api>, ... ])
```

看起来它处理的大部分是 API 请求，而不是下载正常页面。

讨论：你可能会认为没有人使用 crawler.engine.download()，因为它看起来会比较复杂，不过它在 Scrapy 的基代码中使用了两次，分别是 robots.txt 中间件和多媒体管道。因此，当人们需要使用 Web API 时，它也会被推荐为一种解决方案。因为使用它要比使用阻塞 API 更好，比如我们在前面章节中看到的流行的 Python 包 requests；而且，使用它还会比理解 Twisted 编程和使用 treq 简单一些。现在既然有了咱们这本书，这些就不再是使用它的借口了。另一方面，该错误非常难调试，所以应当在研究性能时主动检查下载器中的活跃请求。如果发现 API 或多媒体 URL 不是你爬取的直接目标，那么就意味着某些管道使用了 crawler.engine.download() 来执行 HTTP 请求。由于我们的 CONCURRENT_REQUESTS 限制不适用于这些请求，也就意味着

异步社区会员 Kensuke(18319066421) 专享 尊重版权

我们很可能看到下载器加载的请求数超过 CONCURRENT_ REQUESTS，乍看起来有些矛盾。除非虚假请求数降低到 CONCURRENT_ REQUESTS 以下，否则调度器不会获取新的正常页面请求。

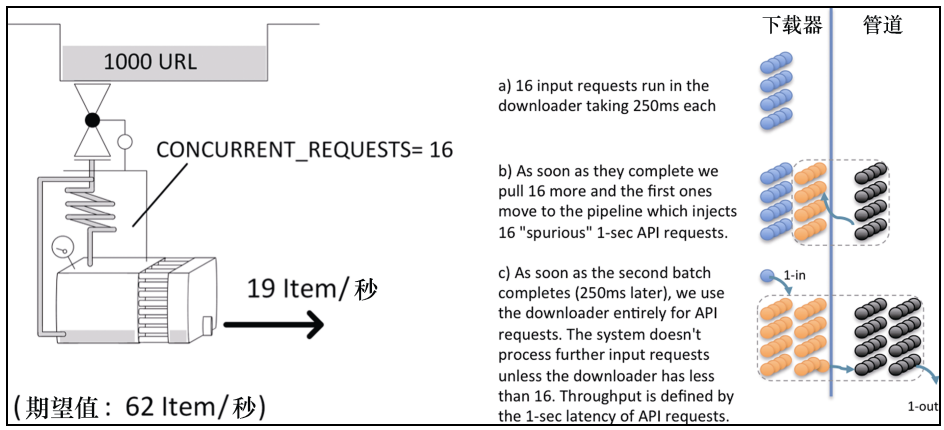


图 10.8　由虚假 API 请求数定义的性能

因此，我们从系统中得到的吞吐量相当于原始请求持续 1 秒（API 延迟），而不是 0.25 秒（页面下载延迟）的吞吐量不是一种巧合。这种情况特别容易令人困惑，因为除非 API 调用比页面请求慢，否则我们不会注意到任何性能下降。

解决方案：我们可以使用 treq 代替 crawler.engine.download() 来解决该问题。你将发现这会使抓取程序的性能突增，这对于 API 架构来说可能是个坏消息。我将从一个低数值的 CONCURRENT_REQUESTS 开始，逐渐增长以确保不会使 API 服务器过载。

下面是和前面相同的运行示例，不过这次使用了 treq。

```
$ 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
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

```
...
real 0m19.922s
```

你会发现一个非常有趣的事情。管道（p/line）似乎包含比下载器（d/load）更多的条目（见图 10.9）。这种情况非常好，并且了解其原因也很有趣。

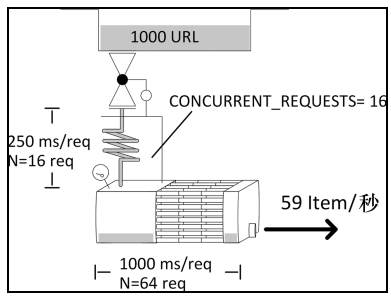


图 10.9 拥有长管道非常完美（在 Google 图片中查看“industrial heat exchanger”）

下载器如预期一样，充分加载了 16 个请求。也就是说，系统吞吐量为 T = N / S = 16 / 0.25= 64 个请求/秒。我们可以通过观察 done 列的增长进行确认。一个请求会在下载器中花费 0.25 秒，但是由于缓慢的 API 请求，它会在管道中花费 1 秒的时间。这意味着在管道中（p/line），我们期望看到平均 N = T • S = 64 • 1 = 64 个 Item。非常好。这表示现在管道有瓶颈吗？不，因为我们没有限制同时在管道中处理的响应数量。只要数值不是无限增加，就能够很好地运行。在下一节中，我们将看到更多关于这个问题的讨论。

10.5.4 案例 #4：大量响应或超长响应造成的溢出

症状：下载器几乎满负荷运转，并且一段时间后关闭。该模式不断重复。抓取程序的内存使用率很高。

示例：此处我们使用了和前面一样的设置（使用了 treq），不过响应会比较大，大约是 120KB 的 HTML。如你所见，此时花费了 31 秒的时间完成，而不是 20 秒左右（见图 10.10）。

```
$ 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
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

```
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 0m30.611s
```

讨论：我们可能会天真地尝试将这种延迟解释为“创建、传输、处理页面需要花费更多时间”，不过这并不是此处发生的情况。此处有一个硬编码（编写代码时写入）的对请求总大小的限制：`max_active_size` = 5000000。假设每个请求的大小等于其请求体的大小，并且至少是 1KB。

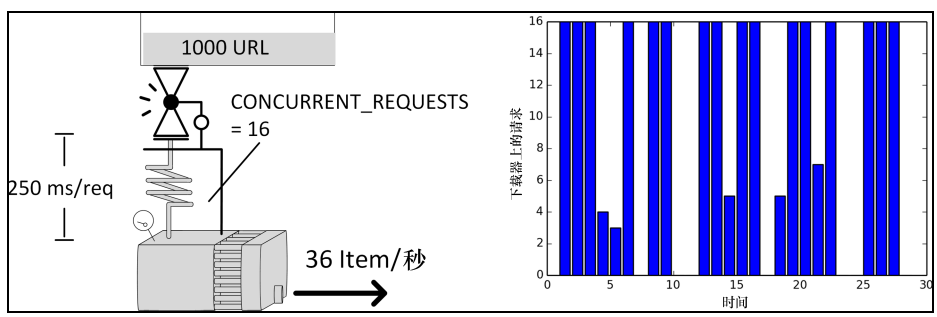


图 10.10　下载器中不规则的请求数表示响应大小限流

一个重要的细节是，该限制可能是 Scrapy 最巧妙且本质的机制，用于防止过慢的爬虫或管道。如果你的任何一个管道的吞吐量比下载器的吞吐量更慢，最终就会发生这种情况。当管道处理时间过长时，即使很小的请求，也很容易触发该限制。下面是一个管道超长的极端案例，80 秒之后就会开始产生问题。

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=10000 -s SPEED_T_
RESPONSE=0.25 -s SPEED_PIPELINE_ASYNC_DELAY=85
```

解决方案：对于已存在的基础架构，针对该问题几乎无计可施。当你不再需要时（比如爬虫之后），清空响应体是个不错的选择，不过在写操作时执行该操作不会重置 Scraper 的计数器。所有你能做的就是降低管道的处理时间，从而有效减少 Scraper 中处理的响应

异步社区会员 Kensuke(18319066421) 专享 尊重版权

数量。可以使用传统的优化手段实现它：检查可能与之交互的 API 或数据库是否能够支持抓取程序的吞吐量；分析抓取程序；将功能管道移动到批处理/后处理系统；使用更强大的服务器或分布式爬取。

10.5.5 案例 #5：有限/过度 item 并发造成的溢出

症状：你的爬虫为每个响应创建了多个 Item。你得到的吞吐量低于预期，并且可能和前面案例中的开/关模式相同。

示例：这里，我们有一个稍微不太一样的设置，我们有 1000 个请求，并且它们的每个返回页面都有 100 个 Item。响应时间是 0.25 秒，Item 管道处理时间为 3 秒。我们设置 CONCURRENT_ITEMS 的值从 10 到 150，执行多次。

```
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
...
```

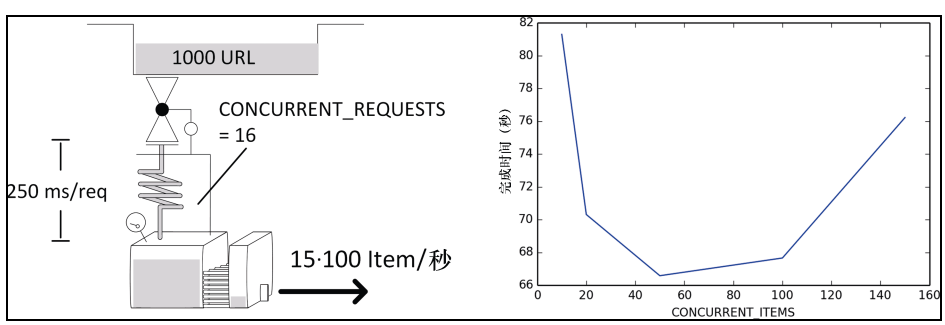


图 10.11 以 CONCURRENT_ITEMS 为变量的爬取时间函数

异步社区会员 Kensuke(18319066421) 专享 尊重版权

讨论：值得再次注意，该情况只适用于爬虫为每个响应生成多个 Item 时。除这种情况外，你应该设置 CONCURRENT_ITEMS = 1，然后忘了它。另外还需注意的是，这是一个虚拟的示例，因为其吞吐量相当大，达到了每秒大约 1300 个 Item。之所以达到如此高的吞吐量，是因为延迟低且稳定、几乎没有真实处理，以及响应的大小很小。这种情况并不常见。

我们首先要注意的事情是，在此之前 scrape 和 p/line 列通常都是相同的数值，而现在 p/line 则是 CONCURRENT_ITEMS · scrape。这是符合预期的，因为 scrape 显示的是响应数，而 p/line 则是 Item 数。

第二个有意思的事情是图 10.11 所示的浴缸形状的性能函数。由于纵轴是缩放的，因此该图表看起来会比实际情况更显著。在左侧，延迟非常高，因为触及了前一节所提到的内存限制。而在右侧，并发过多，造成使用了过多的 CPU。获得最佳效果并不那么重要，因为向左右移动非常容易。

解决方案：检测本案例的两种问题症状非常容易。如果 CPU 使用率过高，那么最好减少 CONCURRENT_ITEMS 的值。如果触及响应的 5MB 限制，那么你的管道无法跟上下载器的吞吐量，增加 CONCURRENT_ITEMS 的值可能能够快速解决该问题。如果修改后没有什么区别，那么应当遵照前面一节给出的建议，再三询问自己系统的其余部分是否能够支持你的抓取程序的吞吐量。

10.5.6　案例 #6：下载器未充分利用

症状：你增加了 CONCURRENT_REQUESTS 的值，但是下载器并未跟上，没能充分利用。调度器是空的。

示例：首先，我们运行一个没有问题的示例。我们将切换到 1 秒的响应时间，因为它能够简化计算量，使下载器的吞吐量 T = N / S = N / 1 = CONCURRENT_REQUESTS。假设按照如下命令运行。

```
$ 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
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

```
...
real 0m10.99s
```

我们得到了一个充分利用的下载器（64 个请求），总时间为 11 秒，与我们以每秒 64 个请求的条件处理 500 个 URL 的模型相匹配（$S = N / T + t_{start/stop} = 500 / 64 + 3.1 = 10.91$ 秒）。

现在，执行相同的爬取，不过不再像前面那些示例那样默认从列表中提供 URL，而是使用索引页通过 `SPEED_START_REQUESTS_STYLE=UseIndex` 抽取 URL。这和我们本书中其他章使用的模式相同。每个索引页默认包含 20 个 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，这和前面的案例不太一样。不知为何，下载器的运行低于其最大能力，并且吞吐量为 $T = N / S - t_{start/stop} = 500 / (32.2 - 3.1) = 17$ 个请求/秒。

讨论：快速浏览 `d/load` 列，可以确信下载器没能充分利用。这是因为我们没有足够的 URL 提供给它。我们的抓取处理生成 URL 的速度比最大消费能力要慢。在本例中，每个索引页会生成 20 个 URL 加上 1 个前往下一索引页的 URL。吞吐量无论如何都无法超过每秒 20 个请求，因为我们无法足够快地得到源 URL。该问题非常隐蔽，容易被忽视。

解决方案：如果每个索引页包含一个以上的下一页的链接，那么可以利用它们加速 URL 的生成。如果可以找到显示更多结果的索引页面（比如 50 个）就更好了。我们可以通过运行几个模拟来观察其行为。

```
$ 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
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

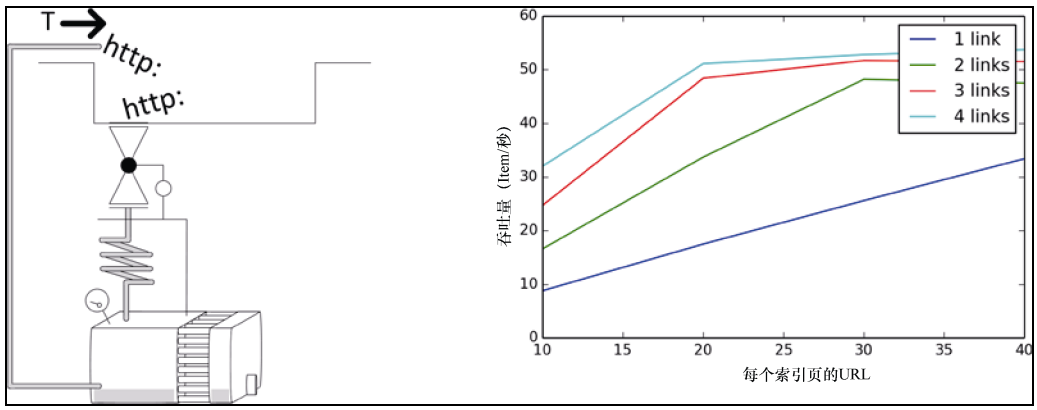


图 10.12　以每个索引页链接的详情页及下一页数量为变量的吞吐量函数

在图 10.12 中，可以看到吞吐量是如何根据这两个参数变化的。我们观察到了线性行为，无论是下一页链接，还是详情页，直到达到系统上限。可以通过重新排列爬取的 `Rule` 进行实验。如果使用 LIFO（默认）顺序，你可能会看到如果先调用索引页请求，最后在列表中抽取它们的话，能够得到较小的改善。你也可以尝试为访问索引页的请求设置高优先级。虽然这两种技术都没有显著的改善，但可以通过分别设置 `SPEED_INDEX_RULE_LAST=1` 和 `SPEED_ INDEX_HIGHER_PRIORITY=1` 来进行尝试。请注意这两种解决方案都会首先下载整个索引页（由于优先级高），因此会在调度器中生成大量 URL，增加内存需求。在它们完成所有索引之前，只会给出少量的结果。对于少量索引还可以接受，但是对于大量索引的情况，就不太可取了。

一个简单而又强大的技术是索引分片。这就需要你使用超过一个初始索引 URL，在它们之间有一个最大距离。比如，如果索引包含 100 页，你可以选取 1 和 51 作为起始索引。然后，爬虫可以以两倍速率使用下一页链接有效遍历索引。如果你能找到一种遍历索引的方式，比如基于产品的品牌或提供给你的任何其他属性，并且可以将其按照大致相等的段进行拆分的话，也可以做到类似的事情。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
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

结果要比前面的技术更好，如果该方法适合你的话，我将会推荐这种方法，因为它更加简单整洁。

10.6 故障排除流程

总结来说，Scrapy 在设计时就将下载器作为瓶颈。从一个低数值的 CONCURRENT_REQUESTS 开始，逐渐增加，直到触及下述限制之一：

- CPU 使用率大于 80%～90%；
- 源网站延迟过度增长；
- 抓取程序中响应达到了 5MB 的内存限制。

同时，执行以下操作：

- 始终保持调度器队列（mqs/dqs）中至少有一定量的请求，避免下载器出现 URL 饥饿；
- 永远不要使用任何阻塞代码或 CPU 密集型代码。

图 10.13 总结了诊断并修复 Scrapy 性能问题的过程。

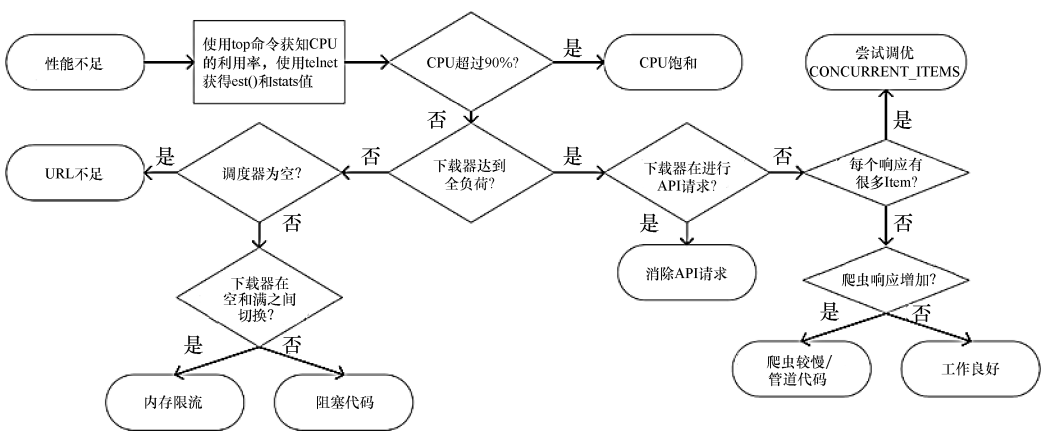


图 10.13 Scrapy 性能问题故障排除

异步社区会员 Kensuke(18319066421) 专享 尊重版权

10.7　本章小结

在本章中，我们尝试通过给出几个有趣的案例，来突出 Scrapy 架构的优秀性能。具体细节可能会在未来版本的 Scrapy 中有所变更，不过本章提供的知识应当会在很长一段时间内保持有效，并且可能会帮助你处理基于 Twisted、Netty Node.js 或类似框架的任何高并发异步系统。

当谈到 Scrapy 的性能问题时，有 3 个有效的答案：我不知道也不介意；我不知道但我会找出来；我知道。正如我们在本章中多次论证的，天真地回答“我们需要更多的服务器/内存/带宽”更有可能与 Scrapy 的性能无关。人们需要真正理解瓶颈在什么地方，并且去提升它。

在最后一章中，我们将进一步专注提升性能，通过在多台服务器上分布式部署爬虫，达到超越单机的能力。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

第 11 章
使用 Scrapyd 与实时分析进行分布式爬取

我们已经走了很长的一段路。我们首先熟悉了两种基础的网络技术——HTML 和 XPath，然后开始使用 Scrapy 爬取复杂网站。接下来，我们深入了解了 Scrapy 通过其设置为我们提供的诸多功能，然后在探讨其 Twisted 引擎的内部架构和异步功能时，更加深入地了解了 Scrapy 和 Python。在上一章中，我们研究了 Scrapy 的性能，并学习了如何解决复杂和经常违背直觉的性能问题。

在最后的这一章中，我将为你指出如何进一步将该技术扩展到多台服务器的一些方向。我们很快就会发现爬取工作经常是一种“高度并发”的问题，因此可以轻松地实现横向扩展，利用多台服务器的资源。为了实现该目标，我们可以像平时那样使用一个 Scrapy 中间件，不过也可以使用 Scrapyd，这是一个专门用于管理运行在远程服务器上的 Scrapy 爬虫的应用。这将允许我们在自己的服务器上，拥有与第 6 章中介绍的相兼容的功能。

最后，我们将使用基于 Apache Spark 的简单系统，对抽取的数据执行实时分析。Apache Spark 是一个非常流行的大数据处理框架。我们将使用 Spark Streaming API 展示在数据收集增多时越来越准确的结果。对于我来说，最终的这个应用展示了 Python 作为一种语言的能力和成熟度，因为我们只需这些，就能编写出富有表现力、简洁并且高效的代码，实现从数据抽取到分析的全栈工作。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

11.1　房产的标题是如何影响价格的

我们尝试解决的示例问题是找出标题是如何与房产价格相关的。我们会认为诸如“Jacuzzi”或“pool”这样的词汇与高价位相关，而类似“discount”这样的词汇与低价位相关。结合位置信息，就可能根据该位置信息和描述，为我们提供房产是否特价的实时报警。

我们所需要计算的是给定词汇在是否存在时的价格差：

$$Shift_{term} = (\overline{Price_{properties-with-term}} - \overline{Price_{properties-without-term}}) / \overline{Price}$$

比如，假设平均租金为$1000，我们观察到包含词汇 jacuzzi 的房产平均价格是$1300，而不包含该词汇的房产平均价格是$995，那么 jacuzzi 的价格差为 $shift_{jacuzzi} = (1300-995) / 1000 = 30.5\%$。如果存在一个包含 jacuzzi 关键词的房产，其价格只比平均价格高出 5%，那么我会非常想要了解它。

请注意，该指标并非微不足道，因为关键词的效果将会被聚合。例如，既包含 jacuzzi 又包含 discount 的标题很可能显示出这些关键词的组合效果。我们收集并分析的数据越多，预估的准确度越高。下面我们将回到该问题上来，讲解如何在一分钟内实现一个流媒体解决方案。

11.2　Scrapyd

现在，我们将要开始介绍 Scrapyd。Scrapyd 这个应用允许我们在服务器上部署爬虫，并使用它们制定爬取的计划任务。让我们来感受一下使用它是多么简单吧。我们在开发机中已经预安装了该应用，所以可以立即使用第 3 章中的代码对其进行测试。我们在之前使用了几乎完全相同的过程，在这里只有一个小的变化。

首先，我们访问 `http://localhost:6800/`，来看一下 Scrapyd 的 Web 界面，如图 11.1 所示。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

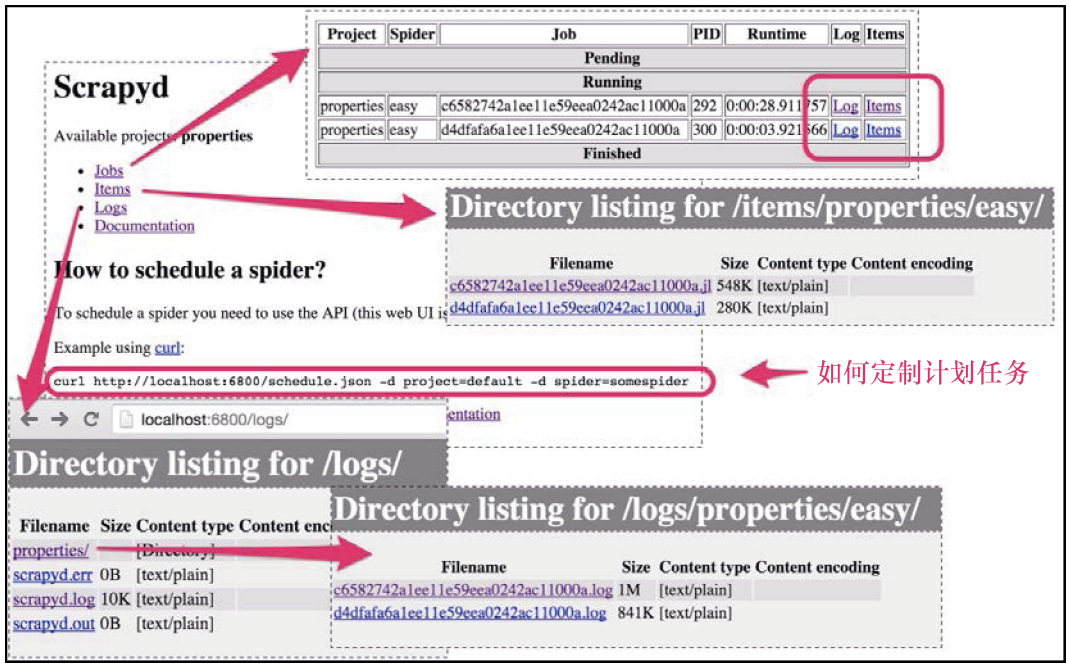


图 11.1 Scrapyd 的 Web 界面

可以看出，Scrapyd 对于 **Jobs**、**Items**、**Logs** 和 **Documentation** 都有不同的区域。此外，它还提供了一些指引，告知我们如何使用其 API 定制计划任务。

为了完成该测试，我们必须先在 Scrapyd 服务器上部署爬虫。第一步是按照如下操作修改 `scrapy.cfg` 配置文件。

```
$ pwd
/root/book/ch03/properties
$ cat scrapy.cfg
...
[settings]
default = properties.settings

[deploy]
url = http://localhost:6800/
project = properties
```

基本上，我们所有需要做的就是去除 **url** 一行的注释。默认的设置已经很合适了。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

现在，要想部署爬虫，需要使用 scrapyd-client 提供的 scrapyd-deploy 工具。scrapyd-client 曾经是 Scrapy 的一部分，不过现在已经独立为一个单独的模块，该模块可以使用 pip install scrapyd- client 安装（已经在开发机中安装好了该模块）。

```
$ 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"}
```

当部署成功后，可以在 Scrapyd 的 Web 界面主页的 **Available projects** 区域看到该项目。现在，可以按照提示在该页面提交一个任务。

```
$ curl http://localhost:6800/schedule.json -d project=properties -d
spider=easy
{"status": "ok", "jobid": " d4df...", "node_name": "dev"}
```

如果回到 Web 界面的 **Jobs** 区域，可以看到任务正在运行。稍后可以使用 schedule.json 返回的 jobid，通过 cancel.json 取消该任务。

```
$ curl http://localhost:6800/cancel.json -d project=properties -d
job=d4df...
{"status": "ok", "prevstate": "running", "node_name": "dev"}
```

请一定记住执行取消操作，否则你会浪费一段时间的计算机资源。

非常好！当访问 **Logs** 区域时，可以看到日志；而当访问 **Items** 区域时，可以看到刚才爬取的 Item。这些都会在一定周期之后清空以释放空间，因此在几次爬取操作后这些内容可能就不再可用。

如果有合理的理由，比如冲突，那么我们可以使用 http_port 修改端口，这是 Scrapyd 提供的诸多设置之一。通过访问 http://scrapyd. readthedocs.org/来了解 Scrapyd 的文档是非常值得的。在本章中，我们需要修改的一个重要设置是 max_proc。如果将该设置保留为默认值 0 的话，Scrapyd 将在 Scrapy 任务运行时允许 4

异步社区会员 Kensuke(18319066421) 专享 尊重版权

倍于 CPU 数量的并发。由于我们将运行多个 Scrapyd 服务器，并且大部分可能是在虚拟机当中的，因此我们将会设置该值为 4，即允许至多 4 个任务并发运行。这与本章的需求有关，而在实际部署当中，一般情况下使用默认值就能够良好运行。

11.3 分布式系统概述

对我来说，设计该系统是一个非常棒的经历（见图 11.2）。起初，我增加了功能和复杂性，以至于不得不要求读者拥有高端硬件才能运行这些示例。这就造成之后的一个紧迫需求成为简化——无论是为了保持硬件需求更加实际，还是确保本章能够保持专注在 Scrapy 上。

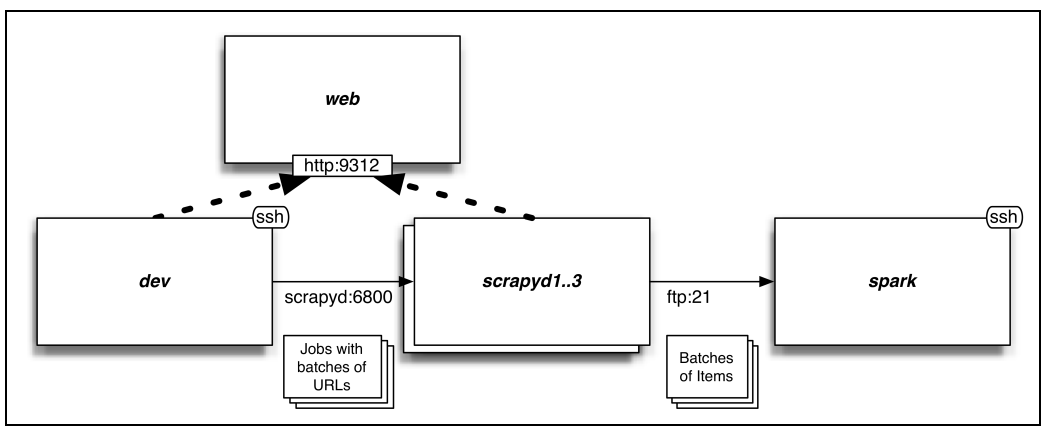


图 11.2 系统概述

最后，本章将要使用的系统包含我们的开发机以及几个其他服务器。我们将使用开发机执行索引页面的垂直抓取，并从中批量抽取 URL。之后，将以轮询的方式将这些 URL 分发到 Scrapyd 节点当中执行爬取。最后，包含 `Item` 的 `.jl` 文件将会通过 FTP 传输到运行 Apache Spark 的服务器中。什么？FTP？是的，我选择 FTP 和本地文件系统，而不是 HDFS 或 Apache Kafka 的原因是因为其内存需求很低，并且 Scrapy 后端的 FEED_URI 能够直接支持。请注意，通过简单修改 Scrapyd 和 Spark 的配置，我们可以使用 Amazon S3 来存储这些文件，享受其带来的冗余性、扩展性等诸多特性。不过，这里不会有更多有意思的相关话题来学习任何奇技淫巧。

异步社区会员 Kensuke(18319066421) 专享 尊重版权



使用 FTP 的一个风险是 Spark 可能会在其上传过程中看到不完整的文件。为了避免发生该问题，我们将使用 Pure-FTPd 以及一个回调脚本，在上传完成后立即将上传的文件移动到 `/root/items` 目录中。

每隔几秒，Spark 将会检测该目录（`/root/items`），读取任何新文件，形成小批次，并执行分析。我们使用 Apache Spark 是因为它支持 Python 作为其编程语言，并且还支持流。到目前为止，我们可能已经使用了一些生命周期相对较短的爬取工作，不过现实世界中许多爬取工作永远都不会结束。爬取工作 24/7 不间断运行，并提供用于分析的数据流，数据越多其结果就越精确。正因如此，我们将使用 Apache Spark 进行展示。



使用 Apache Spark 和 Scrapy 并没有什么特殊之处。你也可以选择使用 Map-Reduce、Apache Storm 或任何其他适合你需求的框架。

在本章中，我们并不会将 `Item` 插入到诸如 ElasticSearch 或 MySQL 等数据库当中。第 9 章中介绍的技术在这里同样适用，不过其性能会很糟糕。当你每秒钟执行数千次写入操作时，只有极少数的数据库系统能够运行良好，但这正是我们的管道将会做的事情。如果我们想要向数据库中插入数据，则需要遵循与使用 Spark 相似的流程，即批量导入生成的 `Item` 文件。你可以修改我们的 Spark 示例流程，批量导入到任意数据库当中。

最后需要注意的是，该系统并没有良好的弹性。我们假设各节点都是健康的，并且任何失败都不会产生严重的业务影响。Spark 拥有弹性配置，能够提供高可用性。而除了 Scrapyd 的持久化队列外，Scrapy 并没有提供任何相关的内建功能，这就意味着失败的任务需要在节点恢复后才能重新启动。这种方式对于你的需求来说，也许适合，也许不适合。如果对你而言弹性十分重要，那么你需要搭建监控和分布式队列方案（如基于 Kafka 或 RabbitMQ），来重启失败的爬取工作。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

11.4 爬虫和中间件的变化

为了构建该系统，我们需要稍微对 Scrapy 爬虫进行修改，并且需要开发爬虫中间件。更具体地说，我们必须执行如下操作：

- 调整索引页爬取，以最大速率执行；
- 编写中间件，分批发送 URL 到 Scrapyd 服务器；
- 使用相同中间件，允许在启动时使用批量 URL。

我们将尝试使用尽可能小的改动来实现这些变化。理想情况下，整个操作应该清晰、易理解并且对其依赖的爬虫代码透明。这应该是一个基础架构层级的需求，如果想对爬虫（可能数百个）进行修改来实现它则是一个坏主意。

11.4.1 索引页分片爬取

我们的第一步是优化索引页爬取，使其尽可能更快。在开始之前，先来设置一些期望。假设爬虫爬取并发量是 16，并且我们测量得到其与源网站服务器的延迟大约为 0.25 秒。此时得到的吞吐量最多为 16 / 0.25 = 64 页/秒。索引页数量为 50000 个详情页 / 每个索引页 30 个详情页链接 = 1667 索引页。因此，我们期望索引页下载花费的时间大约为 1667 / 64 = 26 秒多一点。

让我们以第 3 章中名为 `easy` 的爬虫开始。先把执行垂直抓取的 Rule 注释掉（`callback='parse_item'`的那个），因为现在只需要爬取索引页。



你可以在 GitHub 中获取到本书的全部代码。下载该代码，可以访问：`git clone https://github.com/scalingexcellence/scrapybook`。

本章中的完整代码位于 `ch11` 目录当中。

如果我们在进行任何优化之前对 `scrapy crawl` 只爬取 10 个页面的情况进行计时，

异步社区会员 Kensuke(18319066421) 专享 尊重版权

可以得到如下结果。

```
$ 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 个页面就花费了 4 秒时间，那么就不可能在 26 秒时间内完成 1,700 个页面。通过查看日志，我们发现每个页面都来自于前一个页面的下一页链接，也就是说在任意时刻都只有至多一个页面正在执行爬取。我们的有效并发为 1。我们希望并行处理，得到想要的并发数量（16 个并发请求）。我们将对索引分片，并允许一些额外的分片，以确保爬虫中的 URL 不会不足。我们将会把索引分为 20 个段。实际上，任何超过 16 的数值都能够增加速度，不过在超过 20 之后所得到的回报呈递减趋势。我们将通过如下表达式计算每个分片的起始索引 ID。

```
>>> map(lambda x: 1667 * x / 20, range(20))
[0, 83, 166, 250, 333, 416, 500, ... 1166, 1250, 1333, 1416, 1500, 1583]
```

因此，我们使用如下代码设置 start_urls。

```
start_urls = ['http://web:9312/properties/index_%05d.html' % id
              for id in map(lambda x: 1667 * x / 20, range(20))]
```

这可能和你的索引有很大的不同，因此我们没必要在此处实现得更漂亮。如果还设定了并发设置（CONCURRENT_REQUESTS、CONCURRENT_ REQUESTS_PER_DOMAIN）为 16，那么当运行爬虫时，将会得到如下结果。

```
$ time scrapy crawl easy -s CONCURRENT_REQUESTS=16 -s CONCURRENT_
REQUESTS_PER_DOMAIN=16
...
real 0m32.344s
```

该结果已经与期望值非常接近了。我们的下载速度为 1667 个页面 / 32 秒 = 52 个索

异步社区会员 Kensuke(18319066421) 专享 尊重版权

引页/秒，这就意味着每秒可以生成 52×30 = 1560 个详情页 URL。现在，可以将垂直抓取的 Rule 的注释取消掉，保存文件作为新爬虫分发。我们不需要对爬虫代码进行更多修改，这显示出我们即将开发的中间件的强大以及非侵入性。如果只使用开发服务器运行 scrapy crawl，假设处理详情页的速度和索引页处理时一样快，那么它将花费不少于 50000 / 52 = 16 分钟时间完成爬取。

本节有两个关键内容。在学习完第 10 章之后，我们已经可以实现真正的工程。我们能够精确计算出系统期望得到的性能，并且确保在达到该性能之前不会停止（在合理范围内）。第二个要记住的重要事情是，由于索引页爬取提供了详情页，爬取的总吞吐量将会是其吞吐量的最小值。如果我们生成的 URL 比 Scrapyd 能够消费得更快，那么 URL 将会堆积在其队列当中。反过来，如果生成的 URL 太慢，Scrapyd 将会拥有过剩的无法利用的能力。

11.4.2 分批爬取 URL

现在，我们准备开发处理详情页 URL 的基础架构，目的是对其进行垂直爬取、分批并分发到多台 Scrapyd 节点中，而不是在本地爬取。

如果查看第 8 章中的 Scrapy 架构，就可以很容易地得出结论，这是爬虫中间件的任务，因为它实现了 process_spider_output()，在到达下载器之前，在此处处理请求，并能够中止它们。我们在实现中限制只支持基于 CrawlSpider 的爬虫，另外还只支持简单的 GET 请求。如果需要更加复杂，比如 POST 或有权限验证的请求，那么需要开发更复杂的功能来扩展参数、请求头，甚至可能在每次批量运行后重新登录。

在开始之前，先来快速浏览一下 Scrapy 的 GitHub。我们将回顾 SPIDER_MIDDLEWARES_BASE 设置，以查看 Scrapy 提供的参考实现，以便尽最大可能复用它。Scrapy 1.0 包含如下爬虫中间件：HttpError Middleware、OffsiteMiddleware、RefererMiddleware、UrlLength Middleware 以及 DepthMiddleware。在快速了解它们的实现之后，我们发现 OffsiteMiddleware（只有 60 行代码）与想要实现的功能很相似。它根据爬虫的 allowed_domains 属性，把 URL 限制在某些特定域名中。我们可以使用相似的模式吗？和 OffsiteMiddleware 实现中丢弃 URL 不同，我

异步社区会员 Kensuke(18319066421) 专享 尊重版权

们将对这些 URL 进行分批并发送到 Scrapyd 节点中。事实证明这是可以的。下面是实现的部分代码。

```
def __init__(self, crawler):
    settings = crawler.settings
    self._target = settings.getint('DISTRIBUTED_TARGET_RULE', -1)
    self._seen = set()
    self._urls = []
    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._urls.append(url)
        if len(self._urls) >= self._batch_size:
            self._flush_urls(spider)
```

process_spider_output()既处理 Item 也处理 Request。我们只想处理 Request，因此我们对其他所有内容执行 yield 操作。如果查看 CrawlSpider 的源代码，就会注意到将 Request / Response 映射到 Rule 的方式是通过其 meta 字典的名为'rule'的整型字段。我们检查该数值，如果它指向目标的 Rule（DISTRIBUTED_TARGET_RULE 设置），则会调用_add_to_batch()添加 URL 到当前批次。然后，丢弃该 Request。对其他所有 Request 执行 yield 操作，比如下一页链接、无变化的链接。_add_to_batch()方法实现了一个去重机制。不过很遗憾的是，由于前一节中描述

异步社区会员 Kensuke(18319066421) 专享 尊重版权

的分片流程，我们可能对少数 URL 抽取两次。我们使用_seen 集合检测并丢弃重复值。然后，把这些 URL 添加到_urls 列表中，如果其大小超过_batch_size（DISTRIBUTED_BATCH_SIZE 设置），就会触发调用_flush_urls()。该方法提供了如下的关键功能。

```
    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_urls(self, spider):
        if not self._urls:
            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_urls = json.dumps(self._urls)
        data.append(("setting", "DISTRIBUTED_START_URLS=%s" % json_urls))

        d = treq.post("http://%s/schedule.json" % target,
                      data=data, timeout=5, persistent=False)

        self._scrapyd_submits_to_wait.append(d)

        self._urls = []
        self._batch += 1
```

首先，它使用一个批次计数器（_batch）来决定要将该批次发送到哪个 Scrapyd 服务器中。我们在_targets（DISTRIBUTED_TARGET_HOSTS 设置）中保持更新可用的服务器。然后，构造 POST 请求到 Scrapyd 的 schedule.json。这比之前通过 curl

异步社区会员 Kensuke(18319066421) 专享 尊重版权

执行的更加高级，因为它传递了一些精心挑选的参数。基于这些参数，Scrapyd 可以有效地计划运行任务，类似如下所示。

```
scrapy crawl distr \
-s DISTRIBUTED_START_URLS='[".../property_000000.html", ... ]' \
-s FEED_URI='ftp://anonymous@spark/%(batch)s_%(name)s_%(time)s.jl' \
-a batch=1
```

除了项目和爬虫名外，我们还向爬虫传递了一个 FEED_URI 设置。我们可以从 DISTRIBUTED_TARGET_FEED_URL 设置中获取该值。

由于 Scrapy 支持 FTP，我们可以让 Scrapyd 通过匿名 FTP 的方式将爬取到的 Item 文件上传到 Spark 服务器中。格式包含爬虫名（%(name)s）和时间（%(time)s）。如果只使用这些，那么当两个文件的创建时间相同时，最终会产生冲突。为了避免意外覆盖，我们还添加了一个%(batch)s 参数。默认情况下，Scrapy 不知道任何关于批次的事情，因此我们需要找到一种方式来设置该值。Scrapyd 中 schedule.json 这个 API 的一个有趣特性是，如果参数不是设置或少数几个已知参数的话，它将会被作为参数传给爬虫。默认情况下，爬虫参数将会成为爬虫属性，未知的 FEED_URI 参数将会去查阅爬虫的属性。因此，通过传递 batch 参数给 schedule.json，我们可以在 FEED_URI 中使用它以避免冲突。

最后一步是使用编码为 JSON 的该批次详情页 URL 编译为 DISTRIBUTED_START_URLS 设置。除了熟悉和简单之外，使用该格式并没有什么特殊的理由。任何文本格式都可以做到。



通过命令行向 Scrapy 传输大量数据丝毫也不优雅。在一些时候，你想要将参数存储到数据存储中（比如 Redis），并且只向 Scrapy 传输 ID。如果想要这样做，则需要在_flush_urls()和 process_start_requests()中做一些小的改变。

我们使用 treq.post()处理 POST 请求。Scrapyd 对持久化连接处理得不是很好，因此使用 persistent=False 禁用该功能。为了安全起见，我们还设置了一个 5 秒的超时时间。有趣的是，我们为该请求在_scrapyd_ submits_to_wait 列表中存储了

异步社区会员 Kensuke(18319066421) 专享 尊重版权

延迟函数，后续内容中将会进行讲解。关闭该函数时，我们将重置_urls 列表，并增加当前的_batch 值。

出人意料的是，我们在关闭操作处理器中发现了如下所示的诸多功能。

```
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 URLs
    self._flush_urls(spider)

    yield defer.DeferredList(self._scrapyd_submits_to_wait)
```

_close()将会在我们按下 *Ctrl + C* 或爬取完成时被调用。无论哪种情况，我们都不希望丢失属于最后一个批次的任何 URL，因为它们还没有被发送出去。这就是为什么我们在_close()方法中首先要做的是调用_flush_ urls(spider)清空最后的批次的原因。第二个问题是，作为非阻塞代码，任何 treq.post()在停止爬取时都可能完成或没有完成。为了避免丢失任何批次，我们将使用之前提及的 scrapyd_submits_ to_wait 列表，来包含所有的 treq.post()的延迟函数。我们使用 defer.DeferredList()进行等待，直到全部完成。由于_close()使用了@defer.inlineCallbacks，我们只需对其执行 yield 操作，并在所有请求完成之后进行恢复即可。

总结来说，在 DISTRIBUTED_START_URLS 设置中包含批量 URL 的任务将被送往 Scrapyd 服务器，并在这些 Scrapyd 服务器中运行相同的爬虫。很明显，我们需要某种方式以使用该设置初始化 start_urls。

11.4.3 从设置中获取初始 URL

当你注意到爬虫中间件提供的用于处理爬虫给我们的 start_requests 的 process_start_requests()方法时，就会感受到爬虫中间件是怎样满足我们的需求的。我们检测 DISTRIBUTED_START_URLS 设置是否已被设定，如果是的话，则解码

异步社区会员 Kensuke(18319066421) 专享 尊重版权

JSON 并使用其中的 URL 对相关的 Request 进行 yield 操作。对于这些请求，我们设置 CrawlSpider 的_response_ download()方法作为回调，并设置 meta['rule'] 参数，以便其 Response 能够被适当的 Rule 处理。坦白来说，我们查阅了 Scrapy 的源代码，发现 CrawlSpider 创建 Request 的方式使用了相同的方法。在本例中，代码如下所示。

```
def __init__(self, crawler):
    ...
    self._start_urls = settings.get('DISTRIBUTED_START_URLS', None)
    self.is_worker = self._start_urls 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_urls):
            yield Request(url, spider._response_downloaded,
                          meta={'rule': self._target})
```

我们的中间件已经准备好了。可以在 settings.py 中启用它并进行设置。

```
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",
]
```

一些人可能会认为 DISTRIBUTED_TARGET_RULE 不应该作为设置，因为不同爬虫之间可能是不一样的。你可以将其认为是默认值，并且可以在爬虫中使用 custom_settings 属性进行覆写，比如：

```
custom_settings = {
    'DISTRIBUTED_TARGET_RULE': 3
}
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

不过在我们的例子中并不需要这么做。我们可以做一个测试运行，爬取作为设置提供的单个页面。

```
$ scrapy crawl distr -s \
DISTRIBUTED_START_URLS='["http://web:9312/properties/property_000000.html"]'
```

当爬取成功后，可以尝试更进一步，爬取页面后使用 FTP 传输给 Spark 服务器。

```
scrapy crawl distr -s \
DISTRIBUTED_START_URLS='["http://web:9312/properties/property_000000.
html"]' \
-s  FEED_URI='ftp://anonymous@spark/%(batch)s_%(name)s_%(time)s.jl'  -a
batch=12
```

如果你通过 ssh 登录到 Spark 服务器中（稍后会有更多介绍），将会看到一个文件位于`/root/items` 目录中，比如 `12_distr_date_time.jl`。

上述是使用 Scrapyd 实现分布式爬取的中间件的示例实现。你可以使用它作为起点，实现满足自己特殊需求的版本。你可能需要适配的事情包括如下内容。

- 支持的爬虫类型。比如，一个不局限于 `CrawlSpider` 的替代方案可能需要你的爬虫通过适当的 `meta` 以及采用回调命名约定的方式来标记分布式请求。
- 向 Scrapyd 传输 URL 的方式。你可能希望使用特定域名信息来减少传输的信息量。比如，在本例中，我们只传输了房产的 ID。
- 你可以使用更优雅的分布式队列解决方案，使爬虫能够从失败中恢复，并允许 Scrapyd 将更多的 URL 提交到批处理。
- 你可以动态填充目标服务器列表，以支持按需扩展。

11.4.4 在 Scrapyd 服务器中部署项目

为了能够在我们的 3 台 Scrapyd 服务器中部署爬虫，我们需要将这 3 台服务器添加到 `scrapy.cfg` 文件中。该文件中的每个`[deploy:target-name]`区域都定义了一个新的部署目标。

```
$ pwd
/root/book/ch11/properties
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

```
$ cat scrapy.cfg
...
[deploy:scrapyd1]
url = http://scrapyd1:6800/
[deploy:scrapyd2]
url = http://scrapyd2:6800/
[deploy:scrapyd3]
url = http://scrapyd3:6800/
```

可以通过 scrapyd-deploy -l 查询当前可用的目标。

```
$ scrapyd-deploy -l
scrapyd1              http://scrapyd1:6800/
scrapyd2              http://scrapyd2:6800/
scrapyd3              http://scrapyd3:6800/
```

通过 scrapyd-deploy <target-name>，可以很容易地部署任意服务器。

```
$ 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"}
```

该过程会留给我们一些额外的目录和文件（build、project.egg-info、setup.py），我们可以安全地删除它们。本质上，scrapyd-deploy 所做的事情就是打包你的项目，并使用 addversion.json 上传到目标 Scrapyd 服务器当中。

之后，当我们使用 scrapyd-deploy -L 查询单台服务器时，可以确认项目是否已经被成功部署，如下所示。

```
$ scrapyd-deploy -L scrapyd1
properties
```

我还在项目目录中使用 touch 创建了 3 个空文件(scrapyd1-3)。使用 scrapyd* 扩展为文件名称，同样也是目标服务器的名称。之后，你可以使用一个 bash 循环部署所有服务器：for i in scrapyd*; do scrapyd- deploy $i; done。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

11.5 创建自定义监控命令

如果想监控多台 Scrapyd 服务器的爬虫进程，则需要手动执行。这是一个很好的机会，能够让我们练习到目前为止所见到的一切知识，创建一个原始的 Scrapy 命令——scrappy monitor，用于监控一组 Scrapyd 服务器。我们将该文件命名为 monitor.py，并且在 settings.py 文件中添加 COMMANDS_MODULE = 'properties.monitor'。通过快速浏览 Scrapyd 的文档，我们发现 listjobs.json 这个 API 可以为我们提供任务相关的信息。如果想要找到给定目标的基础 URL，可以猜到它一定在 scrapyd-deploy 代码中的某个地方，从而可以让我们在单个文件中找到它。如果查看 https://github.com/scrapy/scrapyd-client/blob/master/scrapyd-client/scrapyd-deploy，很快就会发现 _get_target() 函数（由于其实现没有添加太多值，因此我会忽略它），在该函数中将会给我们提供目标名称及其基础 URL。太棒了！我们开始实现该命令的第一部分吧，其代码如下所示。

```
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()
```

目前我们所看到的实现还是很简单的。它使用目标名称和我们想要监控的 API 地址填充 _to_monitor 字典。然后，我们使用 task.LoopingCall() 计划到 _monitor()

异步社区会员 Kensuke(18319066421) 专享 尊重版权

方法的定期调用。_monitor()方法使用了 treq 和延迟操作，而我们使用了@defer.inlineCallbacks 来简化其实现。下面是其代码（已忽略一些错误处理和装饰）。

```
@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']))
```

上面这些行已经包含了我们知道的几乎所有 Twisted 技术。我们使用 treq 调用 Scrapyd 的 API，并且使用 defer.DeferredList 立即处理所有响应。当我们的所有结果进入到 all_resp 之后，则开始迭代并获取其 JSON 对象。treq Response 的 json()方法将会返回延迟操作，而不是真实值，我们对其执行了 yield 操作，并会在未来的某个时间点恢复其真实值。最后一步，我们打印出结果。JSON 响应包含待处理、运行中及已完成任务列表的信息，我们将打印出它们的长度。

11.6　使用 Apache Spark 流计算偏移量

此刻，我们的 Scrapy 系统功能齐全。现在，让我们快速看一下 Apache Spark 的功能。

在本章最开始介绍的公式 $shift_{term}$ 非常简单好用，但是无法有效实现。我们可以通过两个计数器计算 $\overline{Price}$，使用 $2 \cdot n_{terms}$ 个计数器计算 $\overline{Price_{with}}$，每个新价格只需更新其中的 4 个。不过计算 $\overline{Price_{without}}$ 则是一个很大的问题，因为对于每个新价格来说，都需要更新 $2 \cdot (n_{terms}-1)$个计数器。比如，我们需要添加 jacuzzi 的价格到每个 $Price_{without}$ 计数器中，而不是只有 jacuzzi 这一个。这会造成算法由于包含大量条件而不可行。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

为了解决该问题，我们所能注意到的是，如果我们将带某个条件的房产价格，与不带相同条件的房产价格相加，将会得到所有房产的价格（很明显！），即 $\Sigma Price = \Sigma Price|_{with} + \Sigma Price|_{without}$。因此，不带某个条件的房产平均价格可以使用如下的代价很小的操作进行计算。

$$\overline{Price_{without}} = \frac{\sum Price_{without}}{n_{without}} = \frac{\sum Price - \sum Price|_{without}}{n - n_{with}}$$

使用该公式，偏移公式变为如下所示。

$$Shift_{term} = \left(\frac{\sum Price|_{with}}{n_{with}} - \frac{\sum Price - \sum Price|_{with}}{n - n_{with}} \right) \Big/ \frac{\sum Price}{n}$$

现在让我们看看如何实现该公式。请注意此处不是 Scrapy 的代码，因此感到有些陌生是很正常的，不过你仍然可以不费太多力气就能阅读并理解该代码。你可以在 boostwords.py 中找到该应用。请记住该代码中包含很多复杂的测试代码，你可以安全地忽略它们。其核心代码如下所示。

```
# 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 使用所谓的 DStream 表示数据流。textFileStream() 方法监控文件系统的目录，当它检测到新文件时，将会从中获取数据流。preprocess() 函数将其转变为条件/价格对的数据流。我们通过 Spark 的 updateStateByKey() 方法，使用 update_state_function() 函数，在运行的计数器中聚合这些条件/价格对。最后，通过运行 to_shifts() 计算偏移量，并使用 print_shifts() 函数打印出最佳结果。我们的大部分功能都很简单，它们只是按照对 Spark 高效的方式形成数据。最有意思的例外是我们的 to_shifts() 函数。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

```
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)
```

它如此紧密地遵循公式，令人印象非常深刻。除了其简单性之外，Spark 的 `mapValues()` 使我们的（可能多台）Spark 服务器能够以最小网络开销高效运行 `calculate_shift`。

11.7　运行分布式爬取

我通常使用 4 个终端查看爬取的完成进度。为了使本节自成一体，因此我还为你提供了打开到相关服务器终端的 `vagrant ssh` 命令（见图 11.3）。

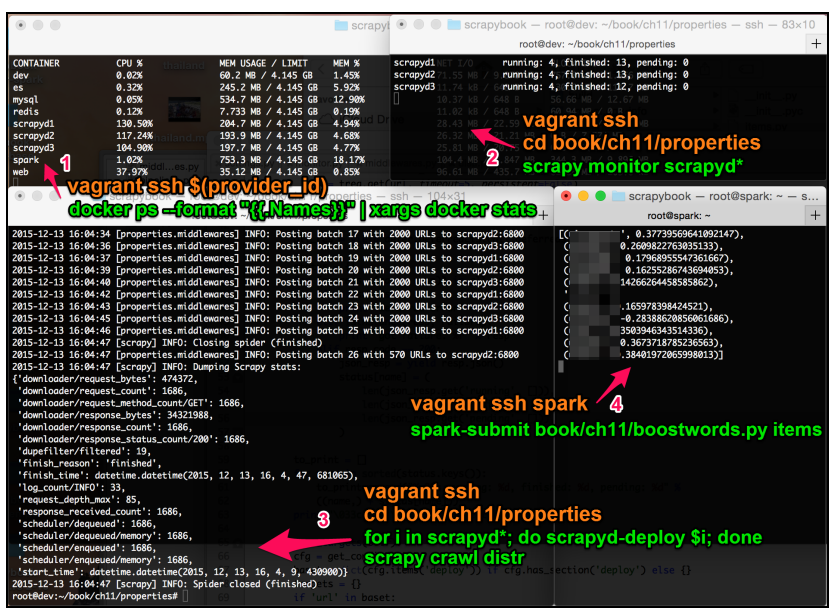


图 11.3　使用 4 个终端监控爬取

异步社区会员 Kensuke(18319066421) 专享 尊重版权

在终端 1 中，我喜欢监控多台服务器的 CPU 和内存使用率。这有助于识别和修复潜在问题。要想启动它，可运行如下命令。

```
$ alias provider_id="vagrant global-status --prune | grep 'docker-
provider' | awk '{print \$1}'"
$ vagrant ssh $(provider_id)
$ docker ps --format "{{.Names}}" | xargs docker stats
```

前面两行稍微复杂的代码允许通过 ssh 登录到 docker provider VM 中。如果使用的不是虚拟机，而是运行在 docker 驱动的 Linux 机器上，那么只需要最后一行。

第 2 个终端同样用于诊断，一般按照如下命令使用它运行 `scrapy monitor`。

```
$ vagrant ssh
$ cd book/ch11/properties
$ scrapy monitor scrapyd*
```

请记住使用 `scrapyd*`以及以服务器名称命名的空文件，`scrapy monitor scrapyd*`将被扩展为 `scrapy monitor scrapyd1 scrapyd2 scrapyd3`。

第 3 个终端是我们的开发机，我们在这里启动爬虫。除此之外，大部分时间是空闲的。如果想要启动一个新的爬虫，可以执行如下命令。

```
$ vagrant ssh
$ cd book/ch11/properties
$ for i in scrapyd*; do scrapyd-deploy $i; done
$ scrapy crawl distr
```

最后两行是最基本的。首先，我们使用 `for` 循环及 `scrapyd-deploy` 部署爬虫到服务器中。然后，使用 `scrapy crawl distr` 启动爬取操作。我们也可以运行更少的爬取操作，比如 `scrapy crawl distr -s CLOSESPIDER_PAGECOUNT=100`，以爬取大约 100 个索引页，相当于大概 3000 个详情页。

最后的第 4 个终端用于连接 Spark 服务器，我们将使用它运行数据流分析任务。

```
$ vagrant ssh spark
$ pwd
/root
$ ls
book items
$ spark-submit book/ch11/boostwords.py items
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

只有最后一行是最基本的，在该行中运行了 `boostwords.py`，并将我们本地的 `items` 目录提供给监控。有时，我还会使用 `watch ls -l items` 来关注 Item 文件的到达情况。

究竟哪些关键词对价格影响最大呢？我把它作为惊喜，留给那些一直跟随下来的读者们。

11.8　系统性能

在性能方面，结果很大程度上取决于我们的硬件情况，以及我们给虚拟机的 CPU 数量和内存大小。在实际部署中，我们可以获得水平的伸缩性，可以让我们以服务器允许的最快速度运行爬取。

对于给定设置情况下的理论最大值是：3 个服务器 • 4 个处理器/服务器 • 16 个并发请求 • 4 个页面/秒（通过页面下载延迟定义）= 768 个页面/秒。

实践时，在 Macbook Pro 中使用分配了 4GB 内存以及 8 核 CPU 的 VirtualBox 虚拟机，我可以在 2 分 40 秒的时间内获取 50,000 个 URL，也就是大约 315 个页面/秒。在拥有 2 个 vCPU 和 8GB 内存的 Amazon EC2 m4.large 实例中，由于有限的 CPU 能力，花费了 6 分 12 秒的时间，即 134 个页面/秒。在拥有 16 个 vCPU 和 64GB 内存的 Amazon EC2 m4.4xlarge 实例中，爬取完成时间是 1 分 44 秒，即 480 个页面/秒。在同一台机器中，我将 Scrapyd 的实例数量加倍，即增加到 6 个（只需编辑 `Vagrantfile`、`scrapy.cfg` 以及 `settings.py`），此时爬虫完成时间为 1 分 15 秒，即其速度为 667 个页面/秒。在最后一种情况下，我们的 Web 服务器似乎遇到了瓶颈（在实际中意味着中断）。

我们得到的性能与理论最大值之间的距离是合理的。有很多小的延迟在我们的粗略计算中是没有考虑进去的。尽管我们之前声称有 250ms 的页面加载延迟，但是在前面的章节中可以看到该延迟实际上更大，因为至少还有 Twisted 和操作系统的延迟。另外，还有一些其他延迟，比如 URL 从开发机传输到 Scrapyd 服务器的时间、我们爬取的 `Item` 通过 FTP 传给 Spark 的时间以及 Scrapyd 发现和计划任务所花费的时间（平均 2.5 秒——参考 Scrapyd 的 `poll_interval` 设置）。此外，还有开发机以及 Scrapyd

异步社区会员 Kensuke(18319066421) 专享 尊重版权

爬取的启动时间没有计算进来。我将不会尝试改善这些延迟中的任何一个，除非能确定它们可以提升吞吐量。我的下一步是增加爬取的大小（比如 50 万个页面）、负载均衡几个 Web 服务器实例以及在我们的扩展尝试中发现下一个有趣的挑战。

11.9 关键要点

本章最重要的要点是，如果你想运行分布式爬虫，则应当使用合适的批次大小。

根据源网站的响应速度，你可能有数百、数千甚至数万个 URL。你会希望它们足够大，达到几分钟的级别，以便能够分摊启动成本。而另一方面，你又不希望它们过大，因为这将会使机器故障成为主要风险。在容错分布式系统中，你可以重试失败的批次，但你不会希望这将给你带来几个小时的工作量。

11.10 本章小结

我希望你能喜欢这本关于 Scrapy 的书，就像我编写它那样。你现在已经对 Scrapy 的能力有了非常丰富的了解，并且能够使用它实现或简单或复杂的爬虫场景。你也会对使用这样一个高性能系统并充分利用它进行开发的复杂性有所了解。使用爬虫，你可以通过自己的应用及时获取现实世界中的大规模数据集。我们已经看到了使用 Scrapy 数据集构建手机应用及实现有趣分析的方式。希望你能使用 Scrapy 开发出优秀、创新的应用，让我们的世界变得更好。祝你好运！

异步社区会员 Kensuke(18319066421) 专享 尊重版权

附录 A
必备软件的安装与故障排除

A.1 必备软件的安装

本书使用了庞大的虚拟服务器系统演示现实中多服务器部署环境下的 Scrapy 使用。我们使用了行业标准工具——Vagrant 和 Docker，来搭建该系统。由于本书严重依赖于网站内容和布局，如果我们使用不可控的网站，那么我们的例子将会在几个月的时间之后无法使用。Vagrant 和 Docker 为我们提供了一个独立的环境，在这里我们的示例无论现在还是以后都能正常运行。作为附带的好处，我们不会访问任何远程服务器，因此就不会对任何网站管理者造成不便。即使我们破坏了某些东西，造成示例无法工作，也可以使用两个命令：`vagrant destroy` 和 `vagrant up --no-parallel`，销毁并重建系统，继续运行。

在开始之前，我需要说明一下，该基础架构是专门为本书读者的需求定制的。尤其是有关 Docker 的部分，普遍共识是每个 Docker 容器应当是只运行单一进程的微服务。我们并没有这么做。我们的很多 Docker 容器都比较重，我们可以使用 `vagrant ssh` 连接它们并执行各种操作。尤其是我们的开发机看起来一点也不像微服务。这是我们去往该隔离系统的用户友好的网关，我们将其视为功能齐全的 Linux 机器。如果我们不使用这种方式改变规则，就必须使用大量的 Vagrant 和 Docker 命令，更加深入地排查故障，在这种情况下本书将很快变为 Vagrant/Docker 书籍。我希望 Docker 爱好者能够原谅我们，

异步社区会员 Kensuke(18319066421) 专享 尊重版权

并且每位读者能够享受到 Vagrant 和 Docker 带给我们的方便和益处。



本书中的容器不适用于生产环境。

我们不可能测试每个软件/硬件的配置。假设某些地方无法工作，如果可以的话，请修复它并在 GitHub 中向我们发送一个 Pull Request。如果你不知道如何修复，那么请在 GitHub 上搜索相关 issue，如果不存在的话请打开一个新的 issue。

A.2　系统

本节用于参考。你可以先跳过本节内容，当想要更好地理解本书系统的构成方式时，可以返回来阅读本节。我们在相关章节中重复了本节中的部分信息。

我们使用 Vagrant 构建了如下系统（见图 A.1）。

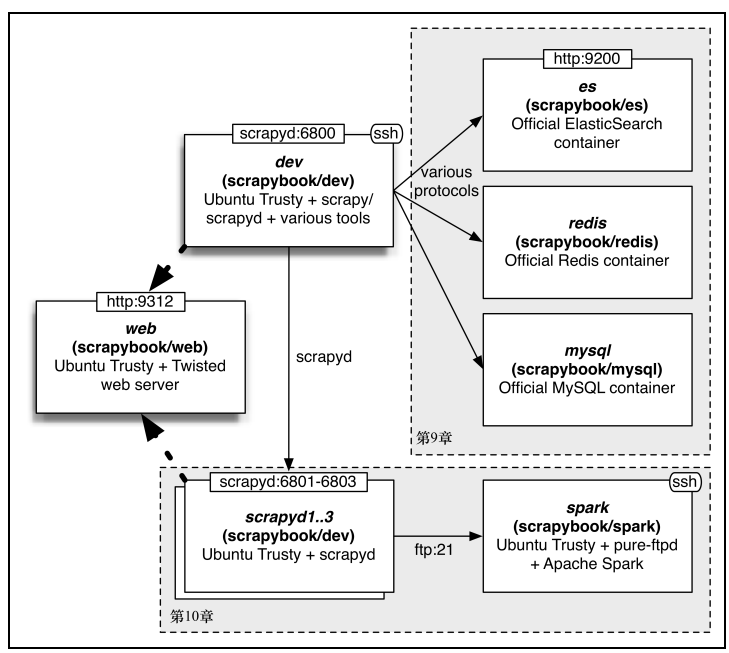


图 A.1　本书使用的系统

异步社区会员 Kensuke(18319066421) 专享 尊重版权

在图 A.1 中，每个方框表示一台服务器，主机名是其标题的第一部分（dev、Web、es 等）。标题的第二部分是其使用的 Docker 镜像（scrapybook/dev、scrapybook/web、scrapybook/es 等）。下面是运行在该服务器上的软件的简要描述。线段表示不同服务器之间的链接，其协议写在线段旁边。Docker 所提供的隔离的一部分是不允许超出显式声明的连接。也就是说，比如你想在 Spark 服务器上使用 1234 端口监听某些东西，除非你在 Vagrant 文件中添加相关声明暴露该端口，否则没有人能连接到该端口。请记住这一点，以避免在其他服务器中安装自定义软件时出现问题。

在大部分章节中，我们只会使用到两个机器：dev 和 web。vagrant ssh 可以让我们连接到开发机中。我们可以从这里使用主机名很轻松地访问其他机器（mysql、web 等）。我们可以通过执行如 ping web 的操作来确认能否访问 web 机器。我们在每章中使用并解释了很多命令。第 9 章演示了如何推送数据到不同的数据库。第 11 章使用了 3 个运行 Scrapyd 的 Docker 容器（实际上与开发机相同，以减少下载大小），这些机器的主机名分别是 scrapyd1-3。我们还使用了一个主机名为 spark 的服务器，用于运行 Apache Spark 以及 FTP 服务。可以使用 vagrant ssh spark 连接该服务器，并运行 Spark 任务。

可以在 GitHub 顶级目录的 Vagrantfile 中找到该系统的描述。当输入 vagrant up --no-parallel 时，系统将开始构建。这将会花费几分钟时间，尤其是在第一次构建时，我们将会在后面的 FAQ 中了解到更详细的介绍。可以看到，本书代码是挂载在~/book 目录当中的。任何时候我们在宿主机修改其中的内容时，变更都会自动传播。这样我们就可以使用文本编辑器或 IDE 修改文件，并且可以在开发机中快速查看变化了。

最后，一些监听端口被转发到我们的宿主机中，并暴露了相关的服务。比如，你可以使用一个简单的 Web 浏览器来访问它们。如果你已经在计算机中使用了其中某个端口，那么会产生冲突，导致系统构建无法成功。我们将会在后面的 FAQ 中告知你如何解决这种情况。表 A.1 是转发的端口列表。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

表 A.1

机器和服务	从开发机访问的地址	从你的（宿主）机访问的地址
Web - web 服务器	`http://web:9312`	`http://localhost:9312`
dev—scrapyd	`http://dev:6800`	`http://localhost:6800`
scrapyd1—scrapyd	`http://scrapyd1:6800`	`http://localhost:6801`
scrapyd2—scrapyd	`http://scrapyd2:6800`	`http://localhost:6802`
scrapyd3—scrapyd	`http://scrapyd3:6800`	`http://localhost:6803`
es—Elasticsearch API	`http://es:9200`	`http://localhost:9200`
spark—FTP	`ftp://spark:21 & 30000-9`	`ftp://localhost:21 & 30000-9`
Redis—Redis API	`redis://redis:6379`	`redis://localhost:6379`
MySQL - MySQL 数据库	`mysql://mysql:3306`	`mysql://localhost:3306`

部分机器的 `ssh` 也是暴露的，Vagrant 负责为我们重定向并转发这些端口，以避免冲突。我们所需要做的就是运行 `vagrant ssh <hostname>`来访问想要连接的机器。

A.3　安装概述

我们所需安装的必要软件如下：

- Vagrant；
- git；
- VirtualBox（Windows 或 Mac 主机）或 Docker（Linux 主机）。

在 Windows 中，可能还需要启用 `git ssh` 客户端。你可以访问它们的网站，并遵照它们对你所使用的平台描述的步骤操作。在下面几节中，我们将尝试提供逐步指引方案，目前来说这些方法是有效的，不过它们肯定会在未来某个时间失效，因此也请随时关注其官方文档。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

A.4　在 Linux 上安装

我们之所以首先介绍如何在 Linux 上安装系统是因为它是最简单的。我将以 Ubuntu 14.04.3 LTS (Trusty Tahr)进行演示，不过该过程在其他分发版本中也会十分相似，当然分发版本越不常见，你就越能了解如何填补其中的差距。为了安装 Vagrant，需要访问 Vagrant 的网站：`https://www.vagrant.com/`，并浏览其下载页。右键单击 **Debian package, 64-bit version**。复制链接地址，如图 A.2 所示。



图 A.2

我们将使用终端安装 Vagrant，因为这是最通用的方式，尽管可以在 Ubuntu 上通过几下单击达成相同目的。为了打开终端，需要单击屏幕左上角的 Ubuntu 图标来打开 **Dash**。另一种方案是，按下 **Windows** 按键。然后输入 `terminal`，并单击 **Terminal** 图标以打开它。

我们输入 `wget`，并粘贴从 Vagrant 页面中得到的链接。几秒后，将会下载一个 `.deb` 文件。输入 `sudo dpkg -I <name of the .deb file you just downloaded>` 以安装文件。到这里为止，Vagrant 已经被安装好了。

安装 `git` 只需要在终端中输入如下两行命令。

```
$ sudo apt-get update
$ sudo apt-get install git
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

现在，让我们来安装 Docker。我们将按照 https://docs.docker.com/engine/installation/ubuntulinux/的指南进行安装。在终端中，输入如下命令。

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D

$ echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main" | sudo
tee /etc/apt/sources.list.d/docker.list

$ sudo apt-get update
$ sudo apt-get install docker-engine
$ sudo usermod -aG docker $(whoami)
```

我们登出并再重新登录以应用分组变化，此时，应该可以没有问题地使用 docker ps 命令了。现在，我们可以下载本书的代码，并享受本书内容。

```
$ git clone https://github.com/scalingexcellence/scrapybook.git
$ cd scrapybook
$ vagrant up --no-parallel
```

A.5　在 Windows 或 Mac 上安装

Windows 和 Mac 环境中的安装过程是相似的，因此我们将一起介绍这两种环境下的安装，并凸显它们之间的区别。

A.5.1　安装 Vagrant

为了安装 Vagrant，我们需要访问 Vagrant 的网站：https://www.vagrantup.com/，并浏览其下载页。选择自己的操作系统，并使用安装向导进行安装，如图 A.3 所示。

几次单击之后，Vagrant 将会安装好。要想访问它，需要打开命令行或终端。

A.5.2　如何访问终端

在 Windows 中，可以按下 *Ctrl* + *Esc* 或 *Win* 键打开应用菜单，并搜索 cmd。而在 Mac 中，可以按下 *Cmd* + *Space*，并搜索 terminal。上述访问方式如图 A.4 所示。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

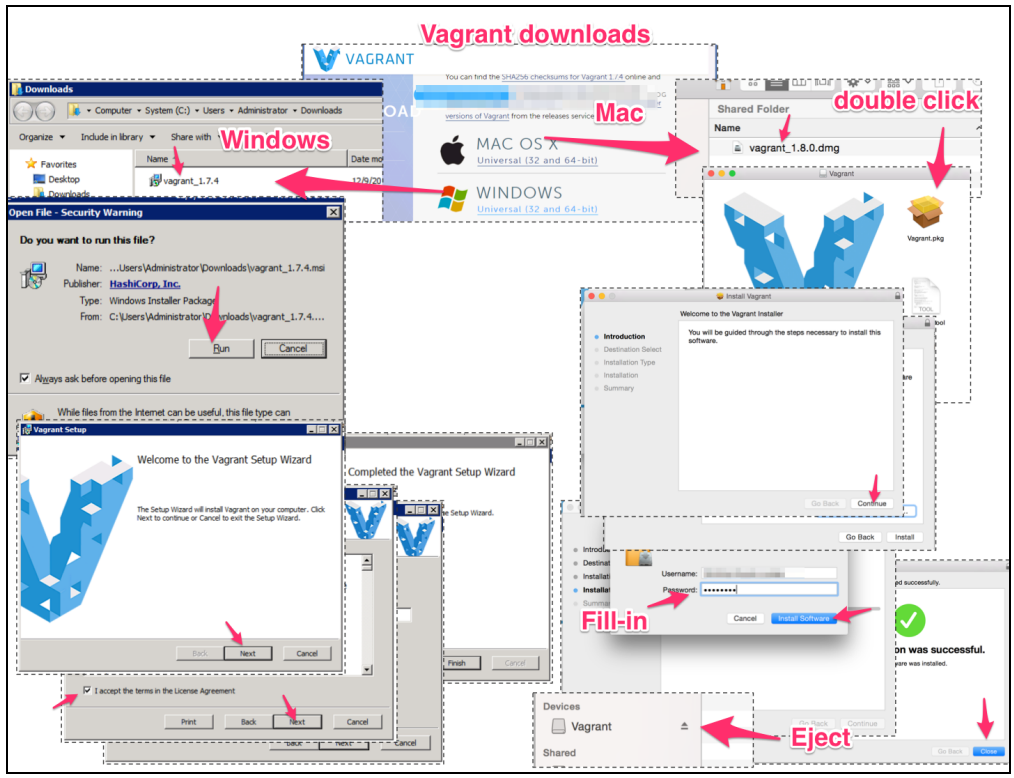


图 A.3

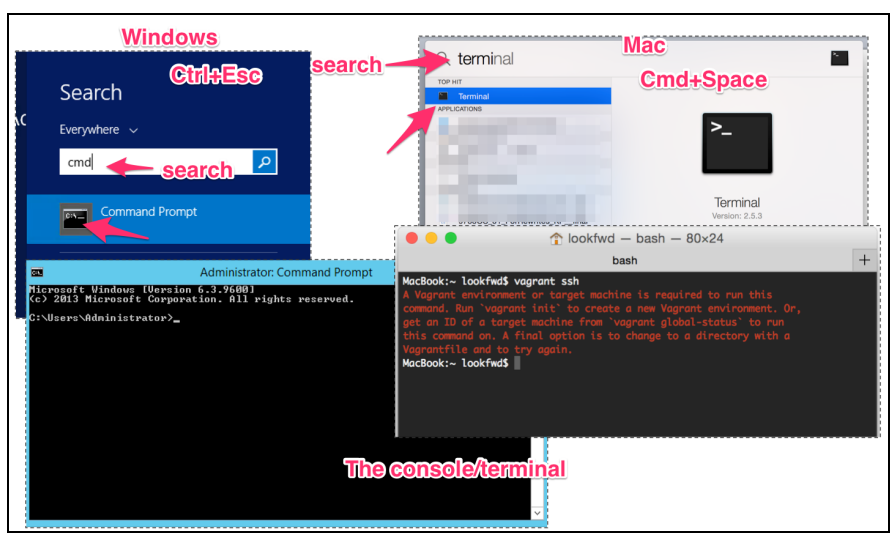


图 A.4

异步社区会员 Kensuke(18319066421) 专享 尊重版权

无论哪种情况，我们都得到了一个控制台窗口，当我们输入 `vagrant` 时，将会打印出一些说明。这就是我们现在所需要做的所有事情。

A.5.3　安装 VirtualBox 和 Git

为了简化该步骤，我们将安装 Docker Toolbox，在其中已经包含了 Git 和 VirtualBox。如果我们使用 Google 搜索 *docker toolbox install*，可以找到 `https://www.docker.com/docker-toolbox`，在这里可以下载适用于我们操作系统的版本。安装过程像 Vagrant 一样简单，如图 A.5 所示。

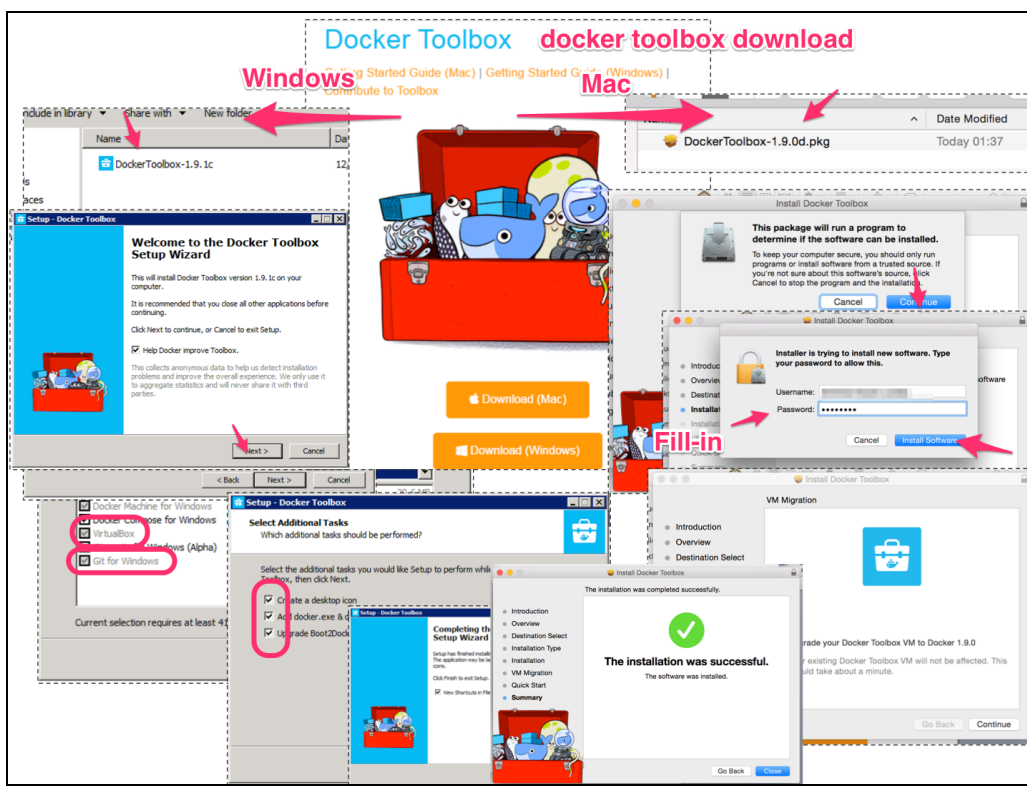


图 A.5

A.5.4　确保 VirtualBox 支持 64 位镜像

安装好 Docker Toolbox 之后，可以在 Windows 桌面或 Mac 的启动器（按下 F4 打开）

异步社区会员 Kensuke(18319066421) 专享 尊重版权

中找到 VirtualBox 的图标。尽早检查 VirtualBox 是否支持 64 位镜像非常重要，检查过程如图 A.6 所示。

打开 VirtualBox，单击 **New** 按钮来创建一个新的虚拟机。查看版本下拉菜单，检查其中的选项，然后单击 **Cancel**。我们现在还不需要真正创建一个虚拟机。



如果下拉菜单中包含 64 位镜像，那么我们可以跳过本节接下来的部分。

如果下拉菜单中没有包含 64 位镜像，或者当我们尝试运行一个 64 位虚拟机时得到类似 **VT-x/AMD-V hardware acceleration is not available on your system** 的错误信息的话，我们可能就有一些麻烦了。

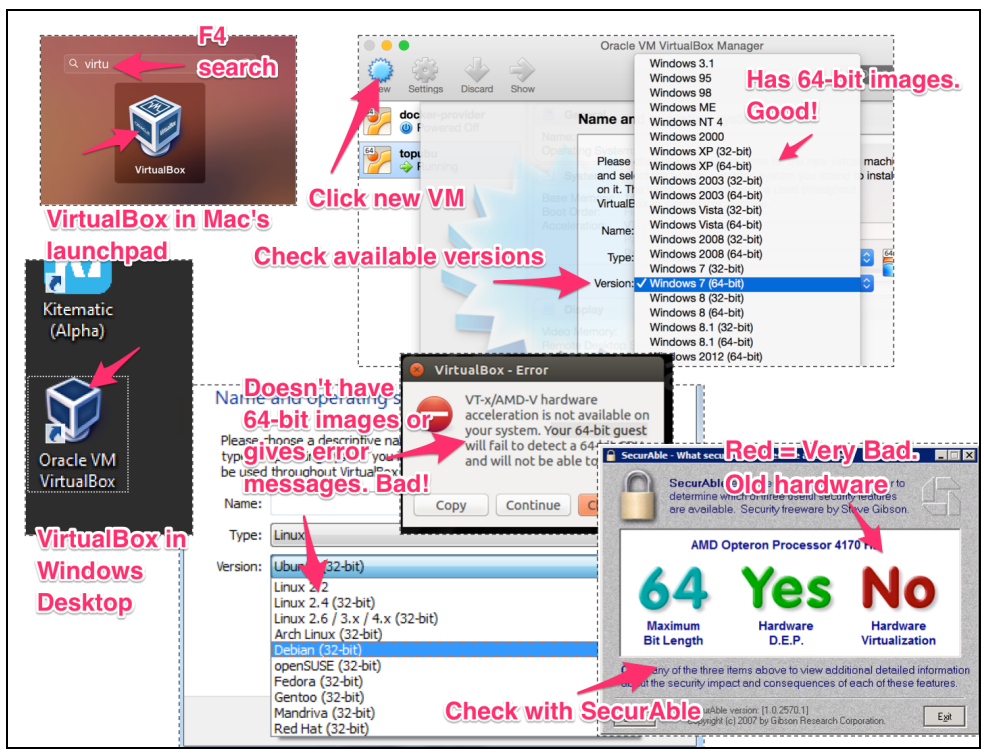


图 A.6

异步社区会员 Kensuke(18319066421) 专享 尊重版权

这意味着 VirtualBox 无法检测到我们电脑中的 VT-x 或 AMD-V 扩展。如果我们的硬件过旧，那么这种情况是合理且符合预期的。但是如果是新硬件，那么很可能是由于这些扩展在 BIOS 中被禁用了。如果我们使用的是 Windows 系统（很大可能），一个简单的方式是通过名为 SecurAble 的工具进行检查，该工具可以从 `https://www.grc.com/securable.htm` 中下载。如果 **Hardware Virtualization** 为红色且提示为 **No** 的话，就意味着我们的 CPU 不支持必要的虚拟扩展。在这种情况下，我们将无法运行 Vagrant/Docker，不过我们仍然可以安装 Scrapy，并且使用在线网站（`scrapybook.s3.amazonaws.com`）作为源来运行这些示例。我们可以从第 4 章中的爬虫开始使用，该爬虫是可以直接拿来使用的，并且是针对在线网站构建的。

如果 **Hardware Virtualization** 为绿色，我们很可能可以从 BIOS 中启用该扩展。使用 Google 搜索你的电脑机型，以及如何变更 BIOS 中关于 VT-x 或 AMD-V 的设置。通常情况下，我们可以在重启时按下某个按键以访问 BIOS。在这里，我们需要进入安全相关的菜单，然后启用 **Virtualization Technology (VTx)**或其他类似写法的选项。重启过后，我们将能够从该计算机运行 64 位的虚拟机。

A.5.5　在 Windows 中启用 ssh 客户端

如果我们使用的是 Mac，将不需要本步，可以直接跳到下一节中。如果我们使用的是 Windows，则没有提供给我们默认的 `ssh` 客户端。幸运的是，Git（我们刚才安装的）有一个 `ssh` 客户端，我们可以通过添加 Windows Path 的方式激活它，如图 A.7 所示。

默认情况下，`ssh` 的二进制文件位于 `C:\Program Files\Git\usr\bin` 中（图 A.7 所示的 1 区域）。我们需要添加 `C:\Program Files\Git\usr\bin` 和 `C:\Program Files\Git\bin` 到路径当中。为了实现该目的，我们需要将它们复制到记事本中，并在每个路径前添加;来连接它们（如图 A.7 所示的 3 区域）。最终结果如下所示：

```
;C:\Program Files\Git\bin;C:\Program Files\Git\usr\bin
```

现在，按下 *Ctrl* + *Esc* 或 *Win* 按键，打开开始菜单，然后找到 **Computer**（**计算机**）选项。右键单击它（图 A.7 所示的 4 区域），并选择 **Properties**（**属性**）。在弹出的窗口中，选择 **Advanced System Settings**（**高级系统设置**）。然后，单击 **Environment Variables**

异步社区会员 Kensuke(18319066421) 专享 尊重版权

（**环境变量**）。这里是我们用于编辑 **Path** 的表单。单击 **Path** 以编辑它。在 **Edit User Variable**（**编辑用户变量**）对话框中，我们在结尾处粘贴在记事本中连接的两个新路径。应当小心不要意外覆盖追加路径；之前的任何值。然后单击几次 **OK**（**确定**），退出所有对话框，此刻必备软件已经全部安装完毕。

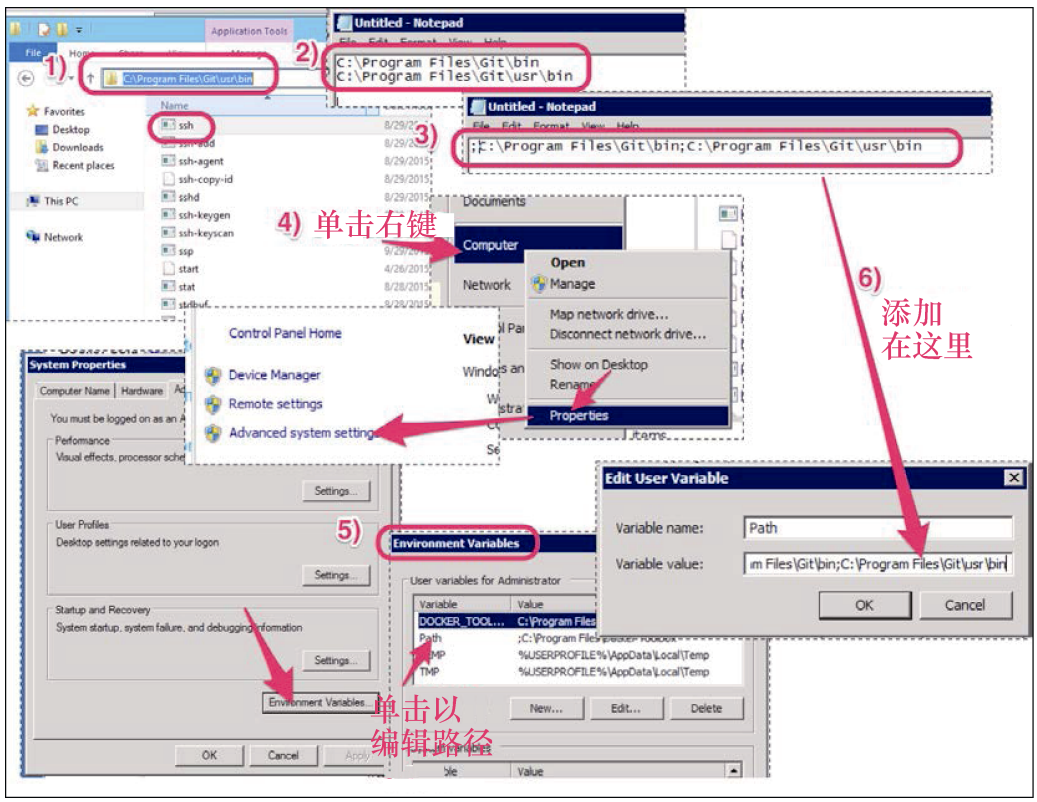


图 A.7

A.5.6　下载本书代码并创建系统

现在，我们已经拥有了一个功能齐全的 Vagrant 系统，接下来打开一个新的控制台/终端/命令行（我们已经在前面见过如何打开），输入如下命令，享受本书所带来的乐趣。

```
$ git clone https://github.com/scalingexcellence/scrapybook.git
$ cd scrapybook
$ vagrant up --no-parallel
```


异步社区会员 Kensuke(18319066421) 专享 尊重版权

A.6　系统创建与操作 FAQ

接下来是你在首次使用 Scrapy 工作时可能遇到的问题的解决方案。

A.6.1　我应该下载什么以及需要花费多少时间

当我们运行 `vagrant up --no-parallel` 之后，就没有那么多的可见度了。所经过的时间与我们的下载速度及网络连接质量密切相关。图 A.8 所示为当网络连接能力达到每秒下载 5MB（38Mbit/s）内容时的期望时间。

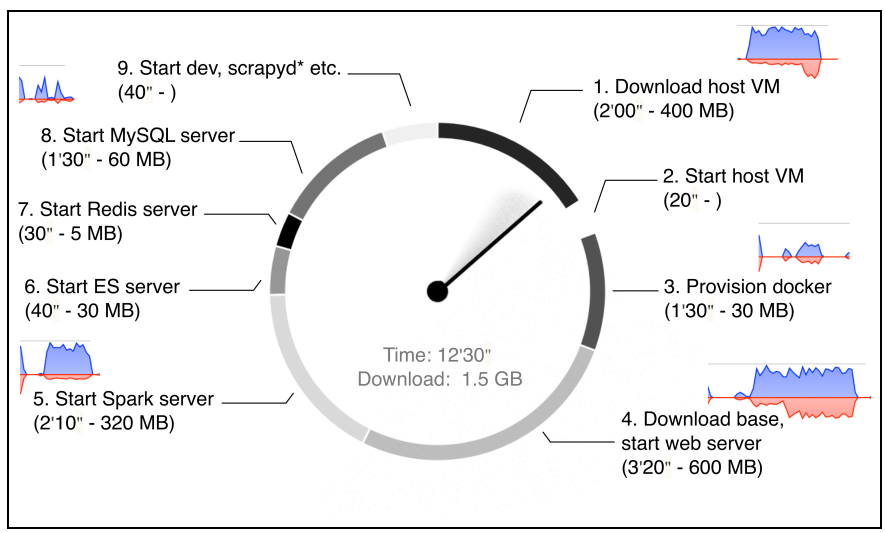


图 A.8

如果我们使用的是 Linux 环境，或是 Docker 已经被安装好，那么前三步就不是必要的，这样可以为我们节省 4 分钟的时间以及 450MB 的下载量。

请注意，上述所有步骤只与用于下载全部内容的 `vagrant up --no- parallel` 命令的第一次运行相关。后续运行在通常情况下只会花费不到 10 秒的时间。

A.6.2　如果 Vagrant 无法响应应该怎么办

可能会有很多原因导致 Vagrant 无法响应，我们所需要做的就是按下 *Ctrl* + *C* 两次

异步社区会员 Kensuke(18319066421) 专享 尊重版权

从中退出。然后再次尝试 `vagrant up --no-parallel`，此时应当能够恢复。我们可能需要这样做几次，这取决于网络连接的速度和质量。如果打开 **Windows Task Manager（Windows 任务管理器）**或 Mac 的 **Activity Monitor（活动监视器）**，可以更清晰地看到 Vagrant 正在做什么，如图 A.9 所示。

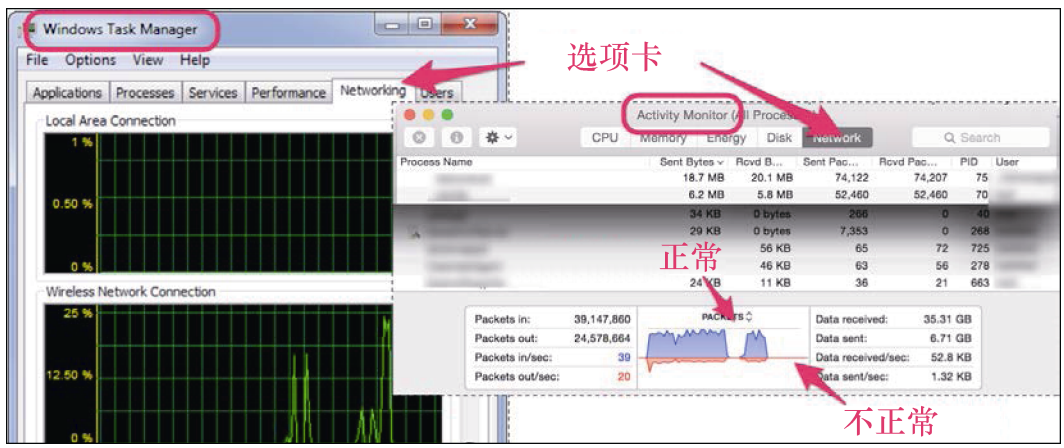


图 A.9

在下载期间或之后不超过 60 秒的短暂无法响应是可以预期的，因为此时软件正在进行安装。而更长时间的无法响应则很有可能意味着出现了某些问题。

当我们中断后再恢复时，`vagrant up --no-parallel` 可能会执行失败，并返回类似下面所述的错误信息。

```
Vagrant cannot forward the specified ports on this VM... The forwarded
port to 21 is already in use on the host machine.
```

这同样是一个临时性的问题。如果我们再次运行 `vagrant up --no-parallel`，则应该能够成功恢复。

假设我们见到了如下的失败信息。

```
... Command: "docker" "ps" "-a" "-q" "--no-trunc"
Stderr: bash: line 2: docker: command not found
```

如果发生该情况，请按照下一个问题所显示的方法关闭并恢复虚拟机。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

A.6.3　如何快速关闭/恢复虚拟机

当使用虚拟机时，最快的关闭方式是进入节能状态，具体来说就是打开 VirtualBox，选择虚拟机，按下 *Ctrl*+ *V* 或 *Cmd* + *V*，或右键单击菜单并选择 **Save State（保存状态）**，如图 A.10 所示。



图 A.10

我们可以通过运行 `vagrant up --no-parallel` 恢复虚拟机。开发和 Spark 服务器的`~/book` 目录都应该可以正常工作。

A.6.4　如何完全重置虚拟机

如果我们想要变更核心数量、内存大小或虚拟机的端口映射，则需要进行完全重置。为了达到该目的，我们仍然需要按照前一个答案的步骤操作，不过现在要选择的是 **Power Off（关闭电源）**，或者按下 *Ctrl* + *F* 或 *Cmd* + *F*。我们也能通过编程方式完成此事，其执行语句是 `vagrant global- status --prune`。我们可以找到名为“docker-provider”的虚拟主机的 ID（比如 95d1234），然后使用 `vagrant halt` 停止它，比如 `vagrant halt 957d887`。

然后，可以使用 `vagrant up --no-parallel` 重启系统。不过很遗憾的是，开发和 Spark 机器很可能已经清空了其`~/book` 目录。要想解决该问题，可以运行 `vagrant`

异步社区会员 Kensuke(18319066421) 专享 尊重版权

destroy -f dev spark，然后重新运行 vagrant up --no-parallel。这将解决此类问题。

A.6.5　如何调整虚拟机大小

我们可能想要改变虚拟机的大小，比如将使用的内存从 2GB 调整为 1GB，将使用的 8 核调整为 4 核。我们可以通过编辑 Vagrantfile.dockerhost 的 vb.memory 及 vb.cpus 设置来进行调整。然后，按照上一个答案的流程完全重置虚拟机。

A.6.6　如何解决端口冲突

有时，在主机上运行的一些服务可能占用了该系统需要的端口。首先，请注意如果我们打开了这两个机器的 Vagrantfile，请移除其中所有的 forwarded_port 语句，按照后面讲到的方法重置，此时仍然能够运行本书中的示例。我们可能刚好不太容易检查宿主机上这些端口运行的服务（通常通过 Web 浏览器）。

也就是说，我们可以通过重新映射冲突端口的方式更适当地解决冲突。让我们使用 Web 服务器 9312 端口的冲突作为示例。根据我们运行的是原生 Linux 还是虚拟机，过程会有些许不同。

Linux 环境使用原生 Docker

该问题将表现为如下所示的错误信息。

```
Stderr: Error: Cannot start container a22f...: failed to create
endpoint web on network bridge: Error starting userland proxy: listen
tcp 0.0.0.0:9312: bind: address already in use
```

打开 Dockerfile，编辑 Web 服务器中 forwarded_port 语句的 host 值。之后，使用 vagrant destroy web 销毁 Web 服务器，并通过 vagrant up web 重启，如果问题发生在初始化加载阶段，则使用 vagrant up --no-parallel 恢复加载。

Windows 或 Mac 环境使用虚拟机

此时，我们会得到不同的错误信息。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

```
Vagrant cannot forward the specified ports on this VM, since they
would collide... The forwarded port to 9312 is already in use
on the host machine...
```

为了修复该问题，我们需要打开 Vagrantfile.dockerhost，移除已有的包含端口号的行。然后在下面添加自定义端口转发语句，比如：config.vm.network "forwarded_port", guest: 9312, host: 9316。此时将会修改为使用 9316 端口。接下来，按照“如何完全重置虚拟机”这一问题的答案流程重置虚拟机，一切又都会正常工作了。

A.6.7　如何隐藏在公司代理背后工作

有一些简单代理和 TLS 拦截代理。简单代理需要我们在请求到达互联网之前，转发到代理服务器上。它们可能需要权限验证，也可能不需要，不过无论哪种情况，我们需要使用的信息就是 URL，该 URL 可以从我们的 IT 部门获取到。它大概形如 http://user:pass@proxy.com:8080/。如果我们使用的是 Linux，而不是虚拟机，很可能已经完全正确配置，不再需要进一步的调整。不过如果我们使用的是虚拟机，则需要使代理服务器在 Vagrant、Docker provider VM、Ubuntu 的 APT 下载以及 Docker 服务自身都应当可用。所有这些操作都已经在 Vagrantfile.dockerhost 中进行了处理，我们只需要移除定义 proxy_url 行的注释，并正确设置其值即可。

假设遇到了如下的 SSL 相关的问题。

```
SSL certificate problem: unable to get local issuer certificate
...
If you'd like to turn off curl's verification of the certificate, use
the -k (or --insecure) option.
```

无论是 Vagrant 还是部署的 Docker，我们都很可能需要处理 TLS 拦截代理的问题。这种代理旨在以一种“中间人”的角色监控所有安全和不安全流量。它们代表我们执行 https 请求，在必要时验证证书；而我们执行到它们的 https 连接，验证它们的证书。我们的 IT 部门很可能会提供给我们一个证书，通常情况下是.crt 文件的形式。我们将该文件的副本放到本书主目录下（Vagrantfile 所在的目录）。接下来，按照前面例子设置 proxy_url，然后更进一步取消掉定义 crt_filename 变量所在行的注释，将其值

异步社区会员 Kensuke(18319066421) 专享 尊重版权

设置为我们的证书文件的名称。

A.6.8　如何连接 Docker provider 虚拟机

如果我们处于 Linux 环境中，并且没有使用虚拟机，那么我们的机器已经是 Docker provider，此时无需做任何事情。如果我们使用的是虚拟机，那么可以通过运行 `vagrant global-status --prune` 得到 Docker provider 的 ID，然后找到名为 `docker-provider` 的机器。我们可以在 Linux 或 Mac 环境中，使用别名的方式对其实现自动化。

```
$ alias provider_id="vagrant global-status --prune | grep 'docker-
provider' | awk '{print \$1}'"
```

我们可以使用 `vagrant ssh <provider id>`，或者在已设置别名的情况下使用 `vagrant ssh $(provider_id)` 来连接 Docker provider。在这里是 Ubuntu Trusty 64 位虚拟机。

A.6.9　每个服务器使用了多少 CPU/内存

如果我们使用了原生 Docker，或者按照前一个答案描述的方法连接到了 provider，那么可以通过 `docker stats`，看到每台独立 Docker 容器所消耗的资源，如下所示。

```
$ docker ps --format "{{.Names}}" | xargs docker stats
```

图 A.11 所示为运行第 11 章代码时的示例输出，此时是 Scrapyd 从 Web 服务器集中下载的时间。

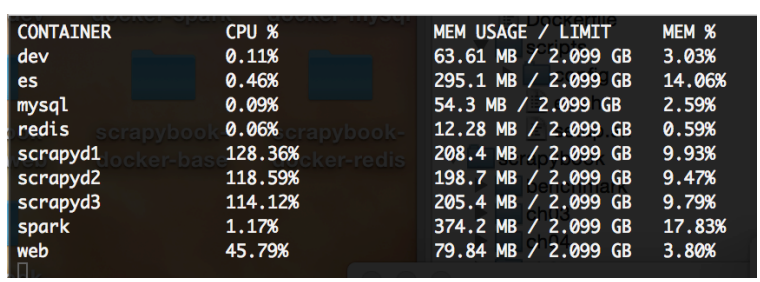


图 A.11

A.6.10　如何查看 Docker 容器镜像的大小

如果我们使用了原生 Docker，或者按照之前答案中看到的方法连接到了 provider，

异步社区会员 Kensuke(18319066421) 专享 尊重版权

那么可以使用如下命令查看 Docker 镜像大小。

```
$ docker images
```

本书的容器都是基于一个镜像，每个变体上安装的其他软件都很少。因此，我们看到的 GB 级的大小是虚拟大小，而不是真实占用的磁盘空间。如果我们想要查看镜像的构建层次以及个体大小，可以为很长的 dockviz 命令创建一个别名，然后按照如下所示进行使用。

```
$ alias dockviz="docker run --rm -v /var/run/docker.sock:/var/run/docker.
sock nate/dockviz"
$ dockviz images -t
```

A.6.11　当 Vagrant 无法响应时，如何重置系统

即使最终处于一个连 Vagrant 也无法重置的混乱状态，我们也可以对系统进行完全重置。我们可以在不重置虚拟主机的情况下做到这一点，当然这种方式需要花费一些时间来完成。我们所需要做的就是连接到 docker provider 机器，强行停止所有容器，移除它们的镜像，然后重启 Docker。具体命令如下所示。

```
$ docker stop $(docker ps -a -q)
$ docker rm $(docker ps -a -q)
$ sudo service docker restart
```

也可以使用如下命令。

```
$ docker rmi $(docker images -a | grep "<none>" | awk "{print $3}")
```

我们使用这种方式移除了下载的所有 Docker 层内容，这就意味着下一次执行 vagrant up --no-parallel 时将会花费一些时间用于下载。

A.7　有一个无法解决的问题，怎么办

我们可以随时使用 VirtualBox 以及从 osboxes.org（http://www.osboxes.org/ubuntu/）下载得到的 Ubuntu 14.04.3（Trusty Tahr）镜像，按照 Linux 的安装过程操作。代码将会完全运行在虚拟机里。我们唯一会忽略的事情是端口转发和同步文件夹，这意味着要么我们手动设置它们，要么在虚拟机中进行开发。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

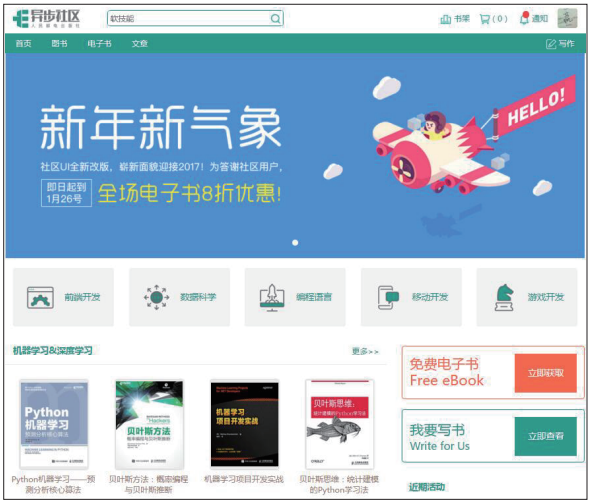
异步社区
人民邮电出版社
www.epubit.com.cn

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户账户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 

里填入可使用的积分数值，即可扣减相应金额。

异步社区会员 Kensuke(18319066421) 专享 尊重版权

特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 57AWG 使用优惠码，然后点击“使用优惠码”，即可享受电子书 8 折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区	微信服务号	微信订阅号	官方微博	QQ 群：436746675

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@ 人邮异步社区，@ 人民邮电出版社 - 信息技术分社

投稿 & 咨询：contact@epubit.com.cn

异步社区会员 Kensuke(18319066421) 专享 尊重版权

精通Python爬虫框架Scrapy



Scrapy是一个开源的Python爬虫框架，可以用来轻松提取页面数据。Scrapy具有丰富的特性，可通过简单的编码或配置来访问，从而可以节省开发人员的开发时间，并精确、高效地提取所需数据。Scrapy有一个高度活跃且迅速增长的社区，而且已经成为黑客、创业者和Web爬取专家的首选框架。

本书讲解了Scrapy的基础知识，讨论了如何从任意源提取数据，如何清理数据，以及如何使用Python和第三方API进行处理，以满足自身需求。本书还讲解了如何将爬取的数据高效地馈入数据库、搜索引擎和流数据处理系统（比如Apache Spark）。在学习完本书后，你将对数据爬取胸有成竹，并将数据应用在自己的应用程序中。

关于译者

李斌，毕业于北京科技大学计算机科学与技术专业，获得硕士学位。曾任职于阿里巴巴，当前供职于凡普金科，负责应用安全工作。热爱Python编程和Web安全，希望以更加智能和自动化的方式提升网络安全。博客地址为pythoner.com。

本书内容：

- 使用HTML和Xpath提取所需的数据；
- 使用Python编写Scrapy爬虫，并在网络上进行爬取操作；
- 将数据推送到任意数据库、搜索引擎或分析系统的方法；
- 配置爬虫，使其下载文件和图形，以及使用代理；
- 创建用来限流数据的高效管道；
- 使用Twitsted实践驱动的API并发处理数百个Item；
- 让爬虫更快速，让内存使用率更高，以及对Scrapy性能进行调优的技巧；
- 使用Scrapyd和Scrapinghub执行大规模分布式爬取操作的最佳方法。

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

ISBN 978-7-115-47420-9

分类建议：计算机／程序设计／软件开发
人民邮电出版社网址：www.ptpress.com.cn

异步社区会员 Kensuke(18319066421) 专享 尊重版权