

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

在 `parse_item()` 内部，可以使用该值替代之前使用过的 XPath 表达式。

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

你会发现我们不再调用 `add_xpath()`，而是转为调用 `add_value()`，这是因为我们该字段中将不会再使用到任何 XPath 表达式。现在，可以使用 `scrapy crawl` 运行这个新的爬虫，并且可以在 `PropertyItems` 中看到来自 `api.json` 的标题。

5.3 30 倍速的房产爬虫

有这样一种趋势，当你开始使用一个框架时，做任何事情都可能会使用最复杂的方式。你在使用 **Scrapy** 时也会发现自己在做这样的事情。在疯狂于 XPath 等技术之前，值得停下来想一想：我选择的方式是从网站中抽取数据最简单的方式吗？

如果你能从索引页中抽取出基本相同的信息，就可以避免抓取每个房源页，从而得到数量级的提升。



请记住，很多网站在其索引页中提供了不同的项目数量选择。比如，一个网站可能允许你通过调整参数指定每个索引页显示的房源数是 10、50 还是 100，如 `&show=50`。显然，如果是这样的情况，就可以将该参数设置为允许的最大值。

比如，在房产示例中，我们所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。这就意味着只抓取一个索引页，就能抽取其中的 30 个条目以及前往下一页的链接。通过爬取 100 个索引页，我们只需要 100 个请求，而不是 3000 个请求，就能够得到 3000 个条目。太棒了！

在真实的 **Gumtree** 网站中，索引页的描述信息要比列表页中完整的描述信息稍短一些。不过此时这种抓取方式可能也是可行的，甚至也能令人满意。



在许多情况下，我们将不得不权衡数据质量与请求数量的关系。很多源都会限制大量的请求（后续章节会遇到更多此类问题），因此在索引中获取也可能帮助我们解决其他难题。

在我们的例子中，当查看任何一个索引页的 HTML 代码时，就会发现索引页中的每个房源都有其自己的节点，并使用 `itemtype="http://schema.org/Product"` 来表示。在该节点中，我们拥有与详情页完全相同的方式为每个属性注解的所有信息，如图 5.4 所示。

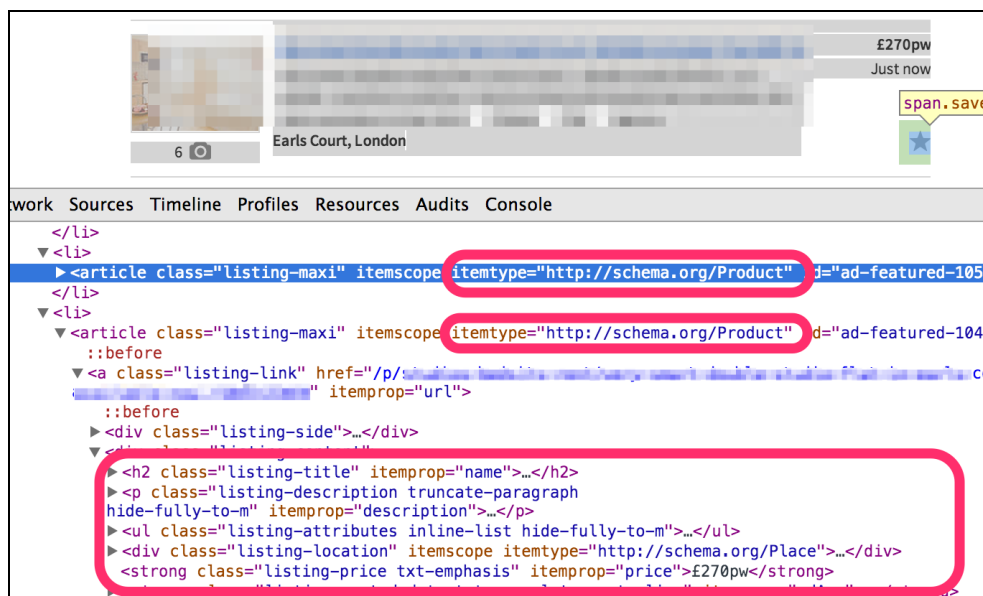


图 5.4 从单一索引页抽取多个房产信息

我们在 Scrapy shell 中加载第一个索引页，并使用 XPath 表达式进行测试。

```
$ scrapy shell http://web:9312/properties/index_00000.html
```

在 Scrapy shell 中，尝试选取所有带有 Product 标签的内容：

```
>>> 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'...]
```

可以看到我们得到了一个包含 30 个 Selector 对象的列表，每个对象指向一个房源。在某种意义上，Selector 对象与 Response 对象有些相似，我们可以在其中使用 XPath 表达式，并且只从它们指向的地方获取信息。唯一需要说明的是，这些表达式应该是相对 XPath 表达式。相对 XPath 表达式与我们之前看到的基本一样，不过在前面增加了一个 '.' 点号。举例说明，让我们看一下使用 `//*[@itemprop="name"][1]/text()` 这个相对 XPath 表达式，从第 4 个房源抽取标题时是如何工作的。

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

可以在 Selector 对象的列表中使用 for 循环，抽取索引页中全部 30 个条目的信息。

为了实现该目的，我们再一次从第 3 章的 `manual.py` 着手，将爬虫重命名为 `"fast"`，并重命名文件为 `fast.py`。我们将复用大部分代码，只在 `parse()` 和 `parse_items()` 方法中进行少量修改。最新方法的代码如下。

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

在代码的第一部分中，对前往下一个索引页的 Request 的 `yield` 操作的代码没有变化。唯一改变的内容在第二部分，不再使用 `yield` 为每个详情页创建请求，而是迭代选择器并调用 `parse_item()`。其中，`parse_item()` 的代码也和原始代码非常相似，如下所示。

```
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_urls', '.*[@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()
```

我们所做的细微变更如下所示。

- ItemLoader 现在使用 selector 作为源，而不再是 Response。这是 ItemLoader API 一个非常便捷的功能，能够让我们从当前选取的部分（而不是整个页面）抽取数据。
- XPath 表达式通过使用前缀点号 (.) 转为相对 XPath。



比较巧合的是，在我们的例子中，索引页和详情页中的 XPath 表达式是一样的。实际情况并不总是这样，你可能需要重新开发 XPath 表达式，以匹配索引页的结构。

- 我们必须自己编辑 Item 的 URL。之前，`response.url` 已经给出了房源页的 URL。而现在，它给出的是索引页的 URL，因为该页面才是我们要爬取的。我们需要使用熟悉的 `//*[@itemprop="url"][1]/@href` 这个 XPath 表达式抽取出房源的 URL，然后使用 MapCompose 处理器将其转换为绝对 URL。

小的改变能够节省巨大的工作量。现在，我们可以使用如下代码运行该爬虫。

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

和预期一样，只用了 3 个请求，就抓取了 90 个条目。如果我们没有在索引页中获取到的话，则需要 93 个请求。这种方式太明智了！

如果你想使用 `scrapy parse` 进行调试，那么现在必须设置 `spider` 参数，如下所示。

```
$ scrapy parse --spider=fast http://web:9312/properties/index_00000.html
...
>>> STATUS DEPTH LEVEL 1 <<<
# Scraped Items -----
[{'address': [u'Angel, London'],
... 30 items...
# Requests -----
[<GET http://web:9312/properties/index_00001.html>]
```

正如期望的那样，`parse()` 返回了 30 个 Item 以及一个前往下一索引页的 Request。请使用 `scrapy parse` 随意试验，比如传输 `--depth=2`。

5.4 基于 Excel 文件爬取的爬虫

大多数情况下，每个源网站只会有一个爬虫；不过在某些情况下，你想要抓取的数据来自多个网站，此时唯一变化的东西就是所使用的 XPath 表达式。对于此类情况，如果为每个网站都使用一个爬虫则显得有些小题大做。那么可以只使用一个爬虫来爬取所有这些网站吗？答案是肯定的。

让我们为该实验创建一个新的爬虫，因为这次爬取的条目会和之前区别很大（实际上我们还没有在该项目中定义任何东西！）。假设此时在 ch05 下的 properties 目录中。让我们向上一层，如下面的代码所示进行操作。

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

我们创建了一个名为 generic 的新项目，以及一个名为 fromcsv 的爬虫。

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

现在，创建一个 .csv 文件，其中包含想要抽取的信息。可以使用一个电子表格程序，比如 Microsoft Excel，来创建这个 .csv 文件。填入如图 5.5 所示的几个 URL 和 XPath 表达式，然后将其命名为 todo.csv，保存到爬虫目录当中（scrapy.cfg 所在目录）。要想保存为 .csv 文件，需要在保存对话框中选择 CSV 文件（Windows）作为文件格式。

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

图 5.5 包含 URL 和 XPath 表达式的 todo.csv

很好！如果一切都已就绪，你就可以在终端上看到该文件。

```
$ 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 有一个用于处理.csv 文件的内置库。只需通过 `import csv` 导入模块，然后就可以使用如下这些直截了当的代码，以字典的形式读取文件中的所有行了。在当前目录下打开 Python 提示符，就可以尝试如下代码。

```
$ 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
```

文件中的第一行会被自动作为标题行处理，并且会根据它们得出字典中键的名称。在接下来的每一行中，会得到一个包含行内数据的字典。我们使用 `for` 循环迭代每一行。当运行前面的代码时，可以得到如下输出。

```
{'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()'}
```

非常好。现在，可以编辑 `generic/spiders/fromcsv.py` 这个爬虫了。我们会用到.csv 文件中的 URL，并且不希望有任何域名限制。因此，首先要做的事情就是移除 `start_urls` 以及 `allowed_domains`，然后读取.csv 文件。

由于我们事先并不知道想要起始的 URL，而是从文件中读取得到的，因此需要实现一个 `start_requests()` 方法。对于每一行，创建 `Request`，然后对其进行 `yield` 操作。此外，还会在 `request.meta` 中存储来自 csv 文件的字段名称和 XPath 表达式，以便在 `parse()` 函数中使用它们。然后，使用 `Item` 和 `ItemLoader` 填充 `Item` 的字段。下面是完整的代码。

```
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()
```

接下来开始爬取，并将结果输出到 out.csv 文件中。

```
$ 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
```

正如爬取得到的结果一样，非常简洁直接！

在代码中，你可能已经注意到了几个事情。由于我们没有为该项目定义系统范围的 `Item`，因此必须像如下代码这样手动为 `ItemLoader` 提供。

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

此外，我们还使用了 `Item` 的成员变量 `fields` 动态添加字段。为了能够动态添加新字段，并通过 `ItemLoader` 对其进行填充，需要实现的代码如下。

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

最后，还可以使代码更加好看。硬编码 `todo.csv` 文件名不是一个非常好的实践。`Scrapy` 提供了一个非常便捷的方法，用于传输参数到爬虫当中。当传输一个命令行参数 `-a` 时（比如：`-a variable=value`），就会为我们设置一个爬虫属性，并且可以通过 `self.variable` 取得该值。为了检查变量，并在未提供该变量时使用默认值，可以使用 `Python` 的 `getattr()` 方法：`getattr(self, 'variable', 'default')`。总之，我们将原来的 `with open...` 语句替换为如下语句。

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

现在，除非明确使用 `-a` 参数设置源文件名，否则将会使用 `todo.csv` 作为其默认值。当给出另一个文件 `another_todo.csv` 时，可以按如下方式运行。

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

5.5 本章小结

本章深入讨论了 `Scrapy` 爬虫的内部机制。我们学习了使用 `FormRequest` 进行登录，使用 `Request/Response` 的 `meta` 属性传输变量，使用相对 `XPath` 表达式和 `Selector`，以及使用 `.csv` 文件作为源等。

接下来，第 6 章会讲解如何将爬虫部署到 `Scrapinghub` 云上，第 7 章将继续深入 `Scrapy` 的设置。

第 6 章

部署到 Scrapinghub

在前面的几章中，我们了解了如何开发 Scrapy 爬虫。当我们对爬虫的功能感到满意时，接下来会有两个选项。如果我们需要的只是使用它们执行简单的抓取工作，那么此时使用开发机运行即可。而另一方面，更常见的情况是需要周期性地运行抓取任务，此时可以使用云服务器，如 Amazon、RackSpace 或其他提供商，不过这些都需要创建、配置和维护工作。此时就是 Scrapinghub 发挥作用的时候了。

Scrapinghub 是 Scrapy 托管的 Amazon 服务器，它是由 Scrapy 开发者创建的 Scrapy 云基础设施提供商。它是一个付费服务，不过也提供了免费方案。如果你想在几分钟内，就能够让 Scrapy 爬虫运行在专业的创建和维护环境中的话，那么本章非常适合你。

6.1 注册、登录及创建项目

第一步是在 <http://scrapinghub.com/> 上面创建账号。我们所需填写的只有邮箱地址和密码。在单击确认邮件的链接后，就可以登录到其服务中。我们可以看到的第一个页面是个人面板。目前，我们还没有任何项目，因此现在单击 **+Service** 按钮（1）来创建一个项目，如图 6.1 所示。

将项目命名为 `properties`（2），然后单击 **Create** 按钮（3）。之后，单击主页的 **new** 链接（4）打开该项目。

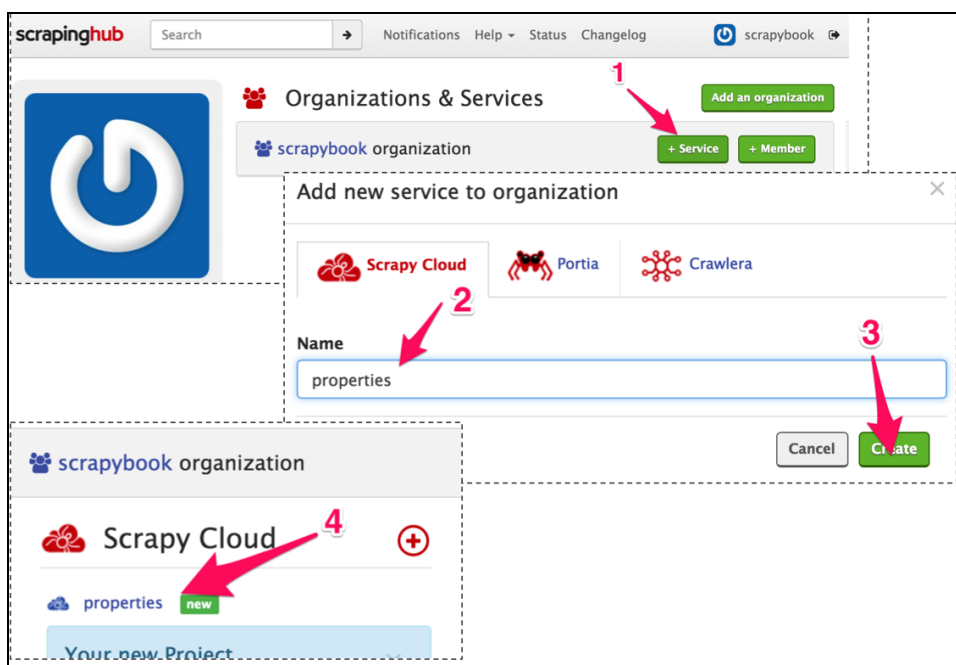


图 6.1 在 scrapinghub 上创建新项目

项目面板是项目中最重要页面。在左侧的菜单中，可以看到几个区域，如图 6.2 所示。**Jobs** 和 **Spiders** 区域分别提供关于运行和爬虫的信息。**Periodic Jobs** 允许我们计划定期爬取任务。而另外 4 个区域目前来说对我们没有那么有用。

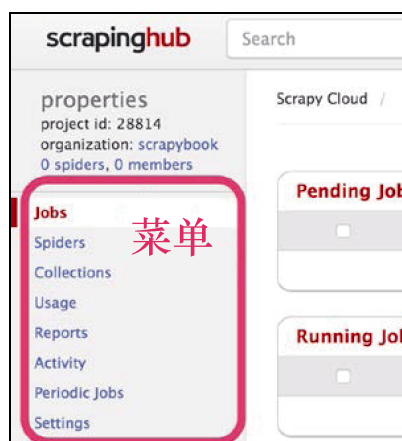


图 6.2 主菜单

我们可以直接前往 **Settings** 区域（1），如图 6.3 所示。与很多网站的设置不同，Scrapinghub 的设置提供了很多功能，需要你十分了解它们。目前，我们的主要关注点是 **Scrapy Deploy** 区域（2）。

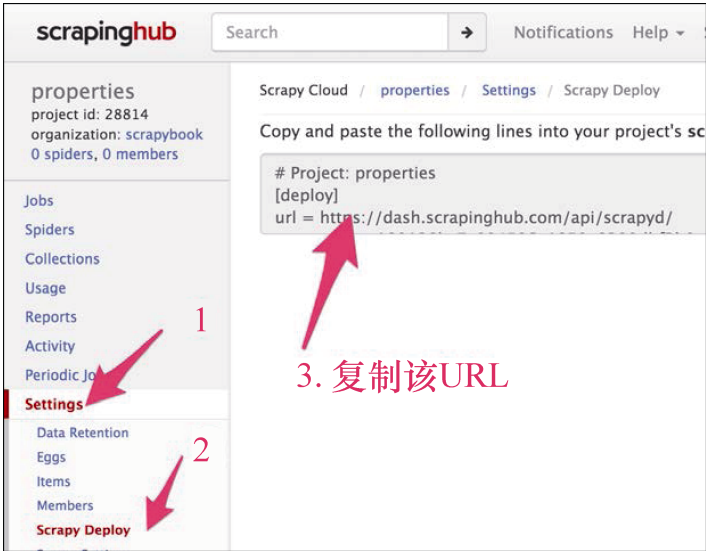


图 6.3 爬虫部署设置

6.2 部署爬虫与计划运行

我们将直接从开发机进行部署。要想实现这一目标，只需将 **Scrapy Deploy** 页面中的代码（3）拷贝到项目中的 `scrapy.cfg` 文件中，替换掉默认的 `[deploy]` 区域即可。你会注意到我们并不需要设置密码。我们将使用第 4 章中的房产项目作为示例，使用该爬虫的原因是目标数据需要能够在网络上访问到，和第 4 章使用的情况一样。在使用它之前，需要恢复原始的 `settings.py` 文件，移除和 Appery.io 管道相关的引用。



本章代码在 `ch06` 目录中。其中，该示例位于 `ch06/properties` 目录中。

```

$ 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, 还需要使用 Scrapinghub 提供的 shub 工具。可以通过 `pip install shub` 安装该工具, 不过我们已经在开发机中已经安装好该工具了。可以使用下述方法登录 Scrapinghub。

```

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

```

我们已经将 API key 复制到 `scrapy.cfg` 文件中了, 不过也可以通过单击 Scrapinghub 网站右上角的用户名, 再单击 **API Key** 找到该值。无论如何, 现在我们已经准备好使用 `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 将本项目中的所有爬虫打包, 并上传到 Scrapinghub 当中。可以注意到, 此时产生了两个新目录和一个新文件。这些只是辅助文件, 如果不需要的话, 可以安全地删除它们, 不过通常情况下没必要在意它们。

```

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

```

现在，当单击 Scrapinghub 的 **Spiders** 区域（1）时，可以找到刚刚部署的 **tomobile** 爬虫，如图 6.4 所示。

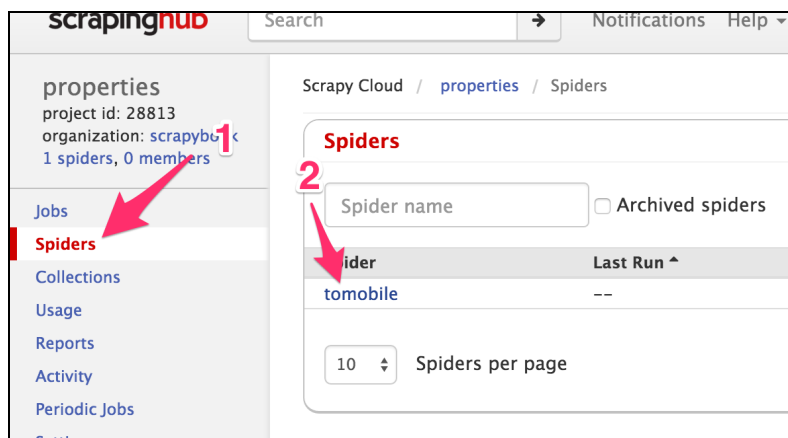


图 6.4 选择爬虫

当单击它时（2），会进入到爬虫面板，如图 6.5 所示。该面板中包含大量信息，不过目前我们需要做的就是单击右上角的 **Schedule** 按钮（3），然后在弹出的对话框中再次单击 **Schedule** 按钮（4）。

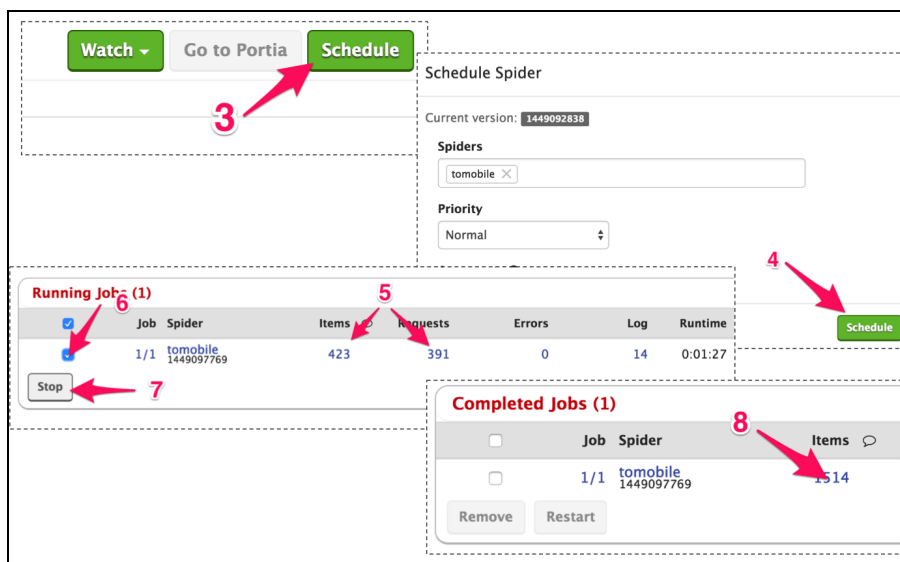


图 6.5 计划爬虫运行

过单击左上方的 **Items** 按钮 (12)，选择合适的选项 (13)，将 item 以常见的 CSV、JSON、JSON 行等格式下载下来。

另一种访问 item 的方式是通过 Scrapinghub 提供的 Items API。我们所需做的就是查看任务或 items 页面中的 URL，类似于下面这样。

```
https://dash.scrapinghub.com/p/28814/job/1/1/
```

在该 URL 中，**28814** 是项目编号（之前在 scrapy.cfg 文件中设置过该值），第一个 1 是该爬虫的编号/ID（即“tomobile”爬虫），而第二个 1 则是任务编号。以上述顺序使用这 3 个数值，并使用我们的用户名/API Key 进行验证，就可以在控制台中使用 curl 建立到 `https://storage.scrapinghub.com/items/<project id>/<spider id>/<job id>` 的请求，获取 item，该过程如下所示。

```
$ 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...
...
```

如果它请求输入密码，我们将其留空即可。允许编程访问数据的特性使得我们可以编写应用，使用 Scrapinghub 作为数据存储后端。不过需要注意的是，这些数据并不是无限期存储的，而是依赖于订阅方案中的存储时间限制（对于免费方案来说该限制为 7 天）。

6.4 计划定时爬取

现在当你听到计划定时爬取任务只需要单击几下鼠标的话，应该不会再感到惊讶了。

该过程如图 6.7 所示。我们只需要前往 **Periodic Jobs** 区域 (1)，单击 **Add** (2)，设置爬虫 (3)，调整爬取频率 (4)，最后单击 **Save** 即可 (5)。

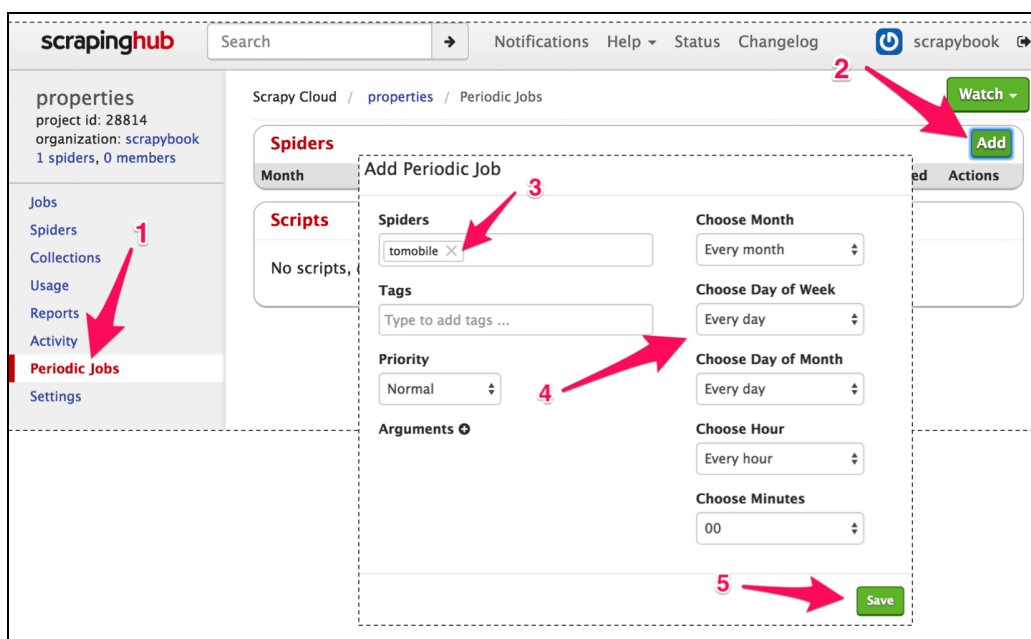


图 6.7 计划定时爬取

6.5 本章小结

在本章中，我们拥有了第一次部署 Scrapy 项目的经验，这里我们使用了 Scrapinghub 将其部署到云端。我们计划运行任务，收集上千个 item，并且可以通过使用 API 的方式非常容易地浏览和抽取它们。在接下来的章节中，我们将会继续提高知识水平，为自己创建一个类似 Scrapinghub 的小型服务器。首先，我们会在下一章中学习配置和管理。

第 7 章

配置与管理

前面章节讲解了使用 Scrapy 开发一个简单爬虫，并用它从网络上抽取数据是多么简单。Scrapy 包含很多工具和功能，可以通过设置使它们可用。对于许多软件框架来说，设置是“令人讨厌的东西”，因为它需要根据系统如何运转进行调整。而对于 Scrapy 来说，设置则是其最重要的基本机制之一，除了调优和配置外，还可以启用功能，以及允许我们扩展框架。我们不打算与优秀的 Scrapy 文档竞争，只想辅助你更快地浏览设置概况，并找出与你最相关的内容。当你准备在生产环境中进行变更之前，请仔细阅读 Scrapy 文档。

7.1 使用 Scrapy 设置

在 Scrapy 中，可以按照 5 个递增的优先级修改设置。我们将会依次看到这 5 个等级。第一级是默认设置，通常不需要修改它，不过 `scrapy/settings/default_settings.py`（在系统的 Scrapy 源代码或 Scrapy 的 GitHub 中可以找到）中的代码确实值得一读。默认设置在命令级别中得以优化。实际上，除非想要实现自定义命令，否则无需考虑它。通常情况下，我们只会在命令级别下一级的项目 `<project_name>/settings.py` 文件中修改设置。这些设置只应用于当前项目。该级别最为方便，因为当我们将项目部署到云服务时，`settings.py` 文件将会打包在其中，并且由于它是一个文件，因此可以使用自己喜欢的文本编辑器轻松调整几十个设置。接下来一级是每个

爬虫的设置。通过在爬虫定义中使用 `custom_settings` 属性，可以轻松地为每个爬虫自定义设置。比如，可以通过该设置为一个指定的爬虫启用或禁用 `Item` 管道。最后，对于一些临时修改，可以使用命令行参数 `-s`，在命令行中传输设置。我们在前面已经使用过几次，比如 `-s CLOSESPIDER_PAGECOUNT=3`，即用于启用爬虫关闭扩展，以便爬虫尽早关闭。在该级别中，我们可能会去设置 `API secrets`、密码等。不要将这些信息写入 `settings.py` 文件中，因为你不会希望它们意外出现在某些公开代码库当中。

在本节中，我们将会研究一些非常重要的常用设置。为了感受不同类型，可以在任意项目中尝试如下命令。

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

你得到的是其默认值。然后，修改项目中的 `<project_name>/settings.py` 文件，为 `CONCURRENT_REQUESTS` 设置一个值，比如 14。此时，前面的 `scrapy settings` 命令将会给出你刚刚设置的那个值，之后不要忘记恢复该值。接下来，尝试从命令行中显式设置该参数，将会得到如下结果。

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

前面的输出提示了一个很有意思的事情。`scrapy crawl` 和 `scrapy settings` 都只是命令。每个命令都能使用刚才描述的加载设置的方法，其示例如下所示。

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

当需要找出项目中某个设置的有效值时，可以使用前面给出的任意一种方法。现在，我们需要更加仔细地了解 `Scrapy` 的设置。

7.2 基本设置

`Scrapy` 包含非常多的设置，因此为其分类成为了一个迫切的需求。我们将会从图 7.1 中总结出的大部分基本设置开始讨论。通过它们了解重要的系统特性，并且我们还将频繁地调整它们。

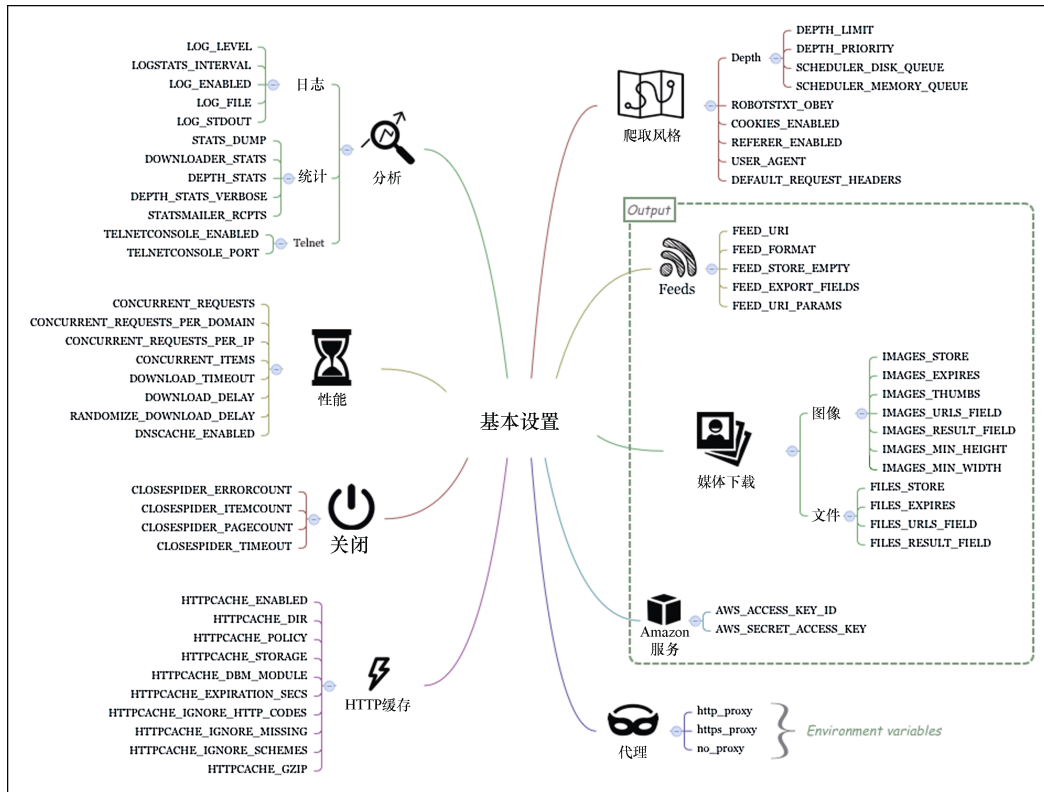


图 7.1 Scrapy 基本设置

7.2.1 分析

使用这些设置，你可以配置 Scrapy，使其通过日志、统计和 Telnet 工具提供性能和调试信息。

1. 日志

Scrapy 基于严重程度，拥有不同的日志等级：DEBUG（最低等级）、INFO、WARNING、ERROR 及 CRITICAL（最高等级）。除此之外，还有一个 SILENT 等级，使用它将不记录任何日志。通过将 LOG_LEVEL 设置为希望日志记录的最低级别，可以限制日志文件只接受指定等级以上的日志。我们一般将该值设为 INFO，因为 DEBUG 级别过于详细。一个非常实用的 Scrapy 扩展是 Log Stats 扩展，该扩展会打印出每分钟抓取的 item 和页

面的数量。日志频率使用 `LOGSTATS_INTERVAL` 进行设置，其默认值为 60 秒。该设置的频率过低，所以在我开发时，会将该值设置为 5 秒，因为大多数运行都很短暂。写入日志的文件可以通过 `LOG_FILE` 设置。除非将 `LOG_ENABLED` 的值设置为 `False` 进行显式禁用，否则日志将会输出到标准错误当中。最后，可以通过设置 `LOG_STDOUT` 为 `True`，告知 Scrapy 将所有标准输出（比如：“print”消息）写入日志。

2. 统计

`STATS_DUMP` 默认是开启的，它会在爬虫结束运行时，将统计信息收集器中的值转存到日志当中。可以通过将 `DOWNLOADER_STATS` 设置为 `False`，控制是否为下载记录统计信息。还可以通过 `DEPTH_STATS` 设置，控制是否收集站点深度的统计信息。要想了解有关深度的更多细节，可以将 `DEPTH_STATS_VERBOSE` 设为 `True`。`STATSMAILER_RCPTS` 是一个邮件列表（比如设置为 `['my@mail.com']`），当爬取完成时，会向该列表中的邮箱发送邮件。无需经常调整这些设置，不过它们偶尔会在调试时帮助你。

3. Telnet

Scrapy 包含一个内置的 Telnet 控制台，可以为你提供正在运行中的 Scrapy 进程的 Python shell。`TELNETCONSOLE_ENABLED` 默认情况下是开启的，而 `TELNETCONSOLE_PORT` 决定了连接到控制台的端口。你可能需要修改该值，以防止端口冲突。

示例 1——使用 Telnet

在某些情况下，需要查看正在运行的 Scrapy 的内部状态。下面让我们看看如何使用 Telnet 控制台完成该操作。



本章代码位于 `ch07` 目录中。其中，本示例在 `ch07/properties` 目录中。

```
$ pwd
/root/book/ch07/properties
```

```
$ ls
properties scrapy.cfg
```

使用如下命令开始爬取。

```
$ scrapy crawl fast
...
[scrapy] DEBUG: Telnet console listening on 127.0.0.1:6023:6023
```

上面的消息意味着 Telnet 已经被激活，并且使用 6023 端口进行监听。现在，可以在另一个终端中，使用 telnet 命令连接它。

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

此时，该控制台会提供一个 Scrapy 内部的 Python 控制台。你可以查看某些组件，比如使用 engine 变量查看引擎，不过为了能够更快地了解状态概况，可以使用 est() 命令。

```
>>> 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.scrapers.slot.active)    : 2
```

第 10 章将会探讨其中的一些度量标准。此时将发现你依然是在 Scrapy 引擎内部运行它。假设使用了如下命令：

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

此时，你会发现在另一个终端中会出现短暂的暂停。显然，该控制台不是用来计算 Pi 值前 100 万位的合适地点。你可以在该控制台中操作的事情还包括暂停、继续和终止爬取。你可能会发现，在远程机器操作 Scrapy 会话时，这些事情和终端通常都很有用。

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

7.2.2 性能

第 10 章将会详细介绍关于性能的设置，这里仅作为一个小结。性能设置可以让我们根据特定的工作负载调整 Scrapy 的性能特性。CONCURRENT_REQUESTS 用于设置同时执行的最大请求数。大多数情况下，该设置用于防止在爬取不同网站（域名/IP）时超出服务器出站容量。除此之外，还可以找到更加严格的 CONCURRENT_REQUESTS_PER_DOMAIN 以及 CONCURRENT_REQUESTS_PER_IP。这两个设置分别通过限制同时对每个域名或 IP 地址发出的请求数，达到保护远程服务器的效果。当 CONCURRENT_REQUESTS_PER_IP 为非零值时，CONCURRENT_REQUESTS_PER_DOMAIN 就会被忽略。这些设置不是以秒为单位的。如果 CONCURRENT_REQUESTS = 16，而请求平均花费 1/4 秒的话，你的限制就是每秒 $16/0.25 = 64$ 个请求。CONCURRENT_ITEMS 用于设置对每个响应同时处理的最大 item 数量。你可能会发现该设置并没有它看起来那么实用，因为很多情况下，每个页面或请求中只有一个 Item。并且，其默认值 100 也比较随意。如果减小该值，比如减小到 10 或者 1，你甚至可能会看到性能提升，这取决于每个请求的 Item 数量，以及管道的复杂程度。还需要注意的是，由于该值是每个请求时的数量，如果限制了 CONCURRENT_REQUESTS = 16、CONCURRENT_ITEMS = 100，那么可能意味着会有 1600 个 item 同时在尝试写入数据库。一般来说，建议将该值设置得更保守一些。

对于下载，DOWNLOAD_TIMEOUT 决定了下载器在取消一个请求之前需要等待的时间，其默认值为 180 秒，这似乎有些偏高（当并发请求数为 16 时，这意味着站点下载的速度大约为 5 页/分钟）。建议降低该值，比如当存在超时问题时，将其降低为 10 秒。默认情况下，Scrapy 将两次下载间的延迟设置为 0，以最大化抓取速度。可以使用 DOWNLOAD_DELAY 设置将其修改为更加保守的下载速度。有些网站会将请求频率作为“机器人”行为的测量指标。通过设置 DOWNLOAD_DELAY，还会在下载延迟中启用一个 $\pm 50\%$ 的随机偏移量。可以通过将 RANDOMIZE_DOWNLOAD_DELAY 设置为 False 来禁用该功能。

最后，为了更快的 DNS 查找，Scrapy 默认使用了 `DNSCACHE_ENABLED` 设置，启用了内存中的 DNS 缓存。

7.2.3 提前终止爬取

Scrapy 的 `CloseSpider` 扩展可以在达成某个条件时，自动终止爬虫爬取。可以分别使用 `CLOSESPIDER_TIMEOUT`（以秒计）、`CLOSESPIDER_ITEMCOUNT`、`CLOSESPIDER_PAGECOUNT` 以及 `CLOSESPIDER_ERRORCOUNT` 这些设置，配置在一段时间后、抓取一定数量 item 后、接收到一定数量响应后或是遇到一定数量错误后，关闭爬虫。通常情况下，你会在运行爬虫时使用命令行的方式设置这些内容，我们已经在前面的几章中做过几次此类操作。

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

7.2.4 HTTP 缓存和离线运行

Scrapy 的 `HttpCacheMiddleware` 组件（默认未激活）为 HTTP 请求和响应提供了一个低级的缓存。当启用该组件时，缓存会存储每个请求及其对应的响应。通过将 `HTTPCACHE_POLICY` 设置为 `scrapy.contrib.httpcache.RFC2616Policy`，可以启用一个遵从 RFC2616 的更复杂的缓存策略。为了启用该缓存，还需要将 `HTTPCACHE_ENABLED` 设置为 `True`，并将 `HTTPCACHE_DIR` 设置为文件系统中的目录（使用相对路径将会在项目的数据文件夹下创建一个目录）。

还可以选择通过设置存储后端类 `HTTPCACHE_STORAGE` 为 `scrapy.contrib.httpcache.DbmCacheStorage`，为缓存文件指定数据库后端，并且还可以选择调整 `HTTPCACHE_DBM_MODULE` 设置（默认为任意数据库管理系统）。还有一些设置可以用于缓存行为调优，不过默认值已经能够为你很好地服务了。

示例 2——使用缓存的离线运行

假设你运行了如下代码：

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

你会发现大约一分钟后运行可以完成。如果此时无法访问 Web 服务器，可能就无法爬取任何数据。假设你现在使用如下代码，再次运行爬虫。

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

你会注意到此时启用了 `HttpCacheMiddleware`，当查看当前目录下的隐藏目录时，将会发现一个新的 `.scrapy` 目录，目录结构如下所示。

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

现在，如果重新运行爬虫，获取略少于前面数量的 item 时，就会发现即使在无法访问 Web 服务器的情况下，也能完成得更加迅速。

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

我们使用了略少于前面数量的 item 作为限制，是因为当使用 `CLOSESPIDER_ITEMCOUNT` 结束时，一般会在爬虫完全结束前读取更多的页面，但我们不希望命中的页面不在缓存范围内。要想清理缓存，只需删除缓存目录即可。

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

7.2.5 爬取风格

Scrapy 允许我们调整选择优先爬取页面的方式。可以在 `DEPTH_LIMIT` 设置中设定

最大深度，该值为 0 时表示不限制。通过 `DEPTH_PRIORITY` 设置，可以基于请求的深度指定优先级。最值得注意的是，可以将该值设置为正数，以执行广度优先爬取，并将任务队列由 LIFO（后入先出）转为 FIFO（先入先出）：

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

在爬取时进行这些设置非常有用，比如，在一个新闻门户网站中，最近的新闻更应该接近首页，并且每个新闻页都有到其他相关新闻的链接。Scrapy 的默认行为是对首页的前几个新闻报道进行尽可能深地爬取，之后才会继续爬取接下来的头版新闻。而广度优先的顺序则是首先爬取最顶层的新闻，之后才会进一步深入，当结合 `DEPTH_LIMIT` 设置时，比如设为 3，可以让你快速浏览门户网站中最近的新闻。

网站在其根目录下使用 Web 标准的 `robots.txt` 文件，声明它们允许的爬取策略，以及不希望被访问的网站结构。如果将 `ROBOTSTXT_OBEY` 设置为 `True`，Scrapy 将会遵守该约定。如果启用了该设置，请在调试时记住该点，以防发现任何意外的行为。

`CookiesMiddleware` 显然包含了和 `cookie` 相关的所有操作，其中包括会话跟踪、准许登录等。如果你想拥有更“私密”的爬取，可以通过将 `COOKIES_ENABLED` 设置 `False` 以禁用。禁用 `cookie` 还会轻微降低你使用的带宽，并且可能会对你的爬取操作有一点提速，当然它会依赖于你爬取的网站。与 `CookiesMiddleware` 类似，`REFERER_ENABLED` 的默认设置是 `True`，即启用了用于填充 `Referer` 头的 `RefererMiddleware`。可以使用 `DEFAULT_REQUEST_HEADERS` 自定义头部。你可能会发现该设置对于某些奇怪的网站很有用，在这些网站中只有包含了特定请求头的请求才不会被禁止。最后，自动生成的 `settings.py` 文件推荐我们设置 `USER_AGENT`。该设置的默认值是 Scrapy 的版本，而我们需要将其修改为能够让网站所有者联系到我们的信息。

7.2.6 feed

`feed` 可以让你将 Scrapy 抓取得到的数据输出到本地文件系统或远程服务器当中。`FEED_URI.FEED_URI` 决定了 `feed` 的位置，该设置中可能会包含命名参数。比如，`scrapy fast -o "%(name)s_%(time)s.jl"` 将会自动以当前时间和爬虫名称

(fast) 填充输出文件名。如果需要使用一个自定义参数, 比如%(foo)s, 那么 feed 输出器需要你在爬虫中提供 foo 属性。此外, feed 的存储, 如 S3、FTP 或本地文件系统, 也定义在 URI 中。例如, FEED_URI='s3://mybucket/file.json' 将使用你的 Amazon 凭证 (AWS_ACCESS_KEY_ID 和 AWS_SECRET_ACCESS_KEY) 上传文件到 Amazon 的 S3 当中。Feed 的格式 (JSON、JSON Line、CSV 及 XML) 可以使用 FEED_FORMAT 确定。如果没有设定该设置, Scrapy 将会根据 FEED_URI 的扩展名猜测格式。通过将 FEED_STORE_EMPTY 设置为 True, 可以选择输出空的 feed。此外, 还可以使用 FEED_EXPORT_FIELDS 设置, 选择只输出指定的几个字段。该设置对于具有固定标题列的 .csv 文件尤其有用。最后, FEED_URI_PARAMS 用于定义对 FEED_URI 中任意参数进行后置处理的函数。

7.2.7 媒体下载

Scrapy 可以使用图像管道下载媒体内容, 此外还可以将图像转换为不同的格式、生成缩略图以及基于大小过滤图像。

IMAGES_STORE 设置用于设定图像存储的目录 (使用相对路径时, 将会在项目根目录下创建目录)。每个 Item 的图像 URL 应该在 image_urls 字段中设定 (可以被 IMAGES_URLS_FIELD 设置覆写), 而下载图像的文件名则是在一个新的 images 字段中设定 (可以被 IMAGES_RESULT_FIELD 设置覆写)。可以使用 IMAGES_MIN_WIDTH 和 IMAGES_MIN_HEIGHT 设置过滤过小的图像。IMAGES_EXPIRES 决定了图像在过期前保留在缓存中的天数。对于缩略图的生成, 可以使用 IMAGES_THUMBS 设置, 它可以让你按照一种或多种尺寸生成缩略图。比如, 可以让 Scrapy 生成一种图标大小的缩略图以及一种用于每次图像下载时的中等大小缩略图。

1. 其他媒体

可以使用文件管道下载其他媒体文件。与图像类似, FILES_STORE 设置用于确定已下载文件的存放位置, 而 FILES_EXPIRES 设置用于确定文件保留的天数。FILES_URLS_FIELD 以及 FILES_RESULT_FIELD 设置都和对应的 IMAGES_* 设置的功能相似。文件管道和图像管道可以同时激活, 不会产生冲突。

示例 3——下载图像

为了能够使用图像功能，必须使用 `sudo pip install image` 安装图像包。在我们的开发机中，已经为大家安装好该三方包了。想要启用图像管道，只需要编辑项目的 `settings.py` 文件，添加少量设置。首先，需要在 `ITEM_PIPELINES` 中包含 `scrapy.pipelines.images.ImagesPipeline`。然后，设置 `IMAGES_STORE` 为相对路径 `"images"`，此外还可以选择通过 `IMAGES_THUMBS` 设置一些缩略图的描述，相关代码如下所示。

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

我们在 `Item` 中已经包含了合适的 `image_urls` 字段，所以现在可以参照如下命令执行爬虫了。

```
$ scrapy crawl fast -s CLOSESPIDER_ITEMCOUNT=90
...
DEBUG: Scraped from <200 http://http://web:9312/.../index_00003.html/
property_000001.html>{
  'image_urls': [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
...

```

可以看到图像成功下载，并且创建了缩略图。主文件的 JPG 名称按照预期存储在了 `images` 字段当中，因此很容易推测缩略图的路径。如果想要清空图像，我们可以使用 `rm -rf images`。

7.2.8 Amazon Web 服务

Scrapy 对访问 Amazon Web 服务有内置支持。你可以在 `AWS_ACCESS_KEY_ID` 设置中存储 AWS 访问密钥，在 `AWS_SECRET_ACCESS_KEY` 设置中存储私密密钥。默认情况下，这些设置均为空。可以在如下场景中使用：

- 当下载以 `s3://` 开头的 URL 时（而不是 `https://` 等）；
- 当通过媒体管道使用 `s3://` 路径存储文件或缩略图时；
- 当在 `s3://` 目录中存储 Item 的输出 Feed 时。

不要将这些设置存储在 `settings.py` 文件当中，以防未来某天由于任何原因造成代码公开时被泄露。

7.2.9 使用代理和爬虫

Scrapy 的 `HttpProxyMiddleware` 组件允许你使用代理设置，根据 UNIX 约定，这些设置是通过 `http_proxy`、`https_proxy` 以及 `no_proxy` 这几个环境变量定义的。该组件默认是启用状态的。

示例 4——使用代理和 Crawlera 的智能代理

DynDNS（或任何类似的服务）提供了一个免费的在线工具，用于查看当前的 IP 地址。使用 Scrapy shell，我们向 `checkip.dyndns.org` 发送请求，查看响应，获取当前的 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( )
```

想要开始代理请求，需要退出 shell，并使用 `export` 命令设置新的代理。可以通过搜索 HMA 的公开代理列表测试免费代理 (<http://proxylist.hidemyass.com>)。比如，我们从该列表中选择一个 IP 为 10.10.1.1、端口为 80 的代理（非真实存在的代理，请将其替换为你自己的代理地址），可以按照如下操作。

```
$ # 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
```

按照刚才的步骤重新运行 scrapy shell，可以看到执行的请求使用了不同的 IP。此外，你还会发现这些代理通常速度都很慢，而且在一些情况下无法成功，如果遇到这类情况，可以尝试更换为其他的代理。如果想要禁用代理，则需要退出 Scrapy shell，并执行 `unset http_proxy`（或恢复为之前的值）。

Crawlera 是 Scrapinghub 的一项服务，可以为 Scrapy 的开发者提供一个非常智能的代理。除了在后台使用很大的 IP 池路由你的请求外，该代理还会调整延迟和失败重试，让你在保持尽可能快的情况下，获得尽可能多且稳定的成功响应流。它基本上可以使爬虫开发者的梦想成真，并且只需像之前那样，设置 `http_proxy` 环境变量，就可以使用。

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

除了 HTTP 代理外，Crawlera 还可以通过 Scrapy 的中间件组件方式使用。

7.3 进阶设置

现在，我们要探讨一些 Scrapy 中不太常见的方面，以及 Scrapy 扩展的相关设置，后续章节中会详细介绍这些内容。这些进阶设置如图 7.2 所示。

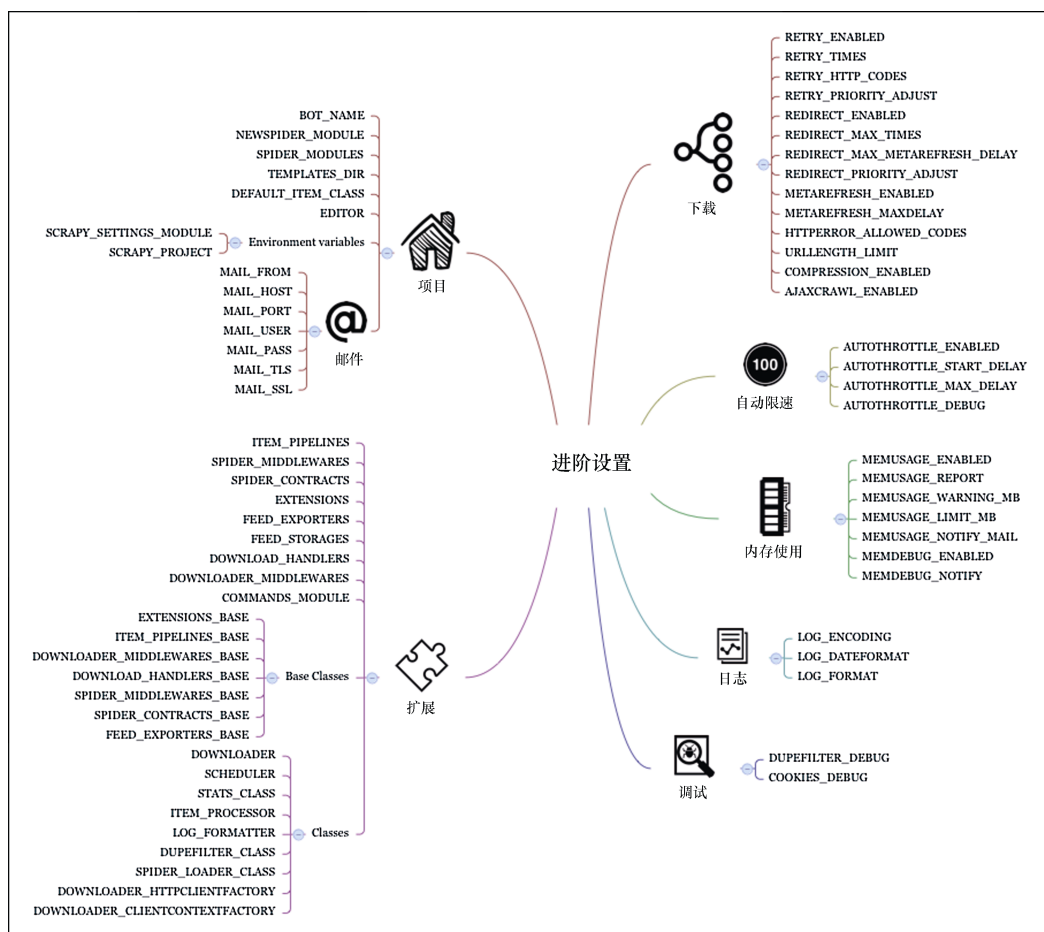


图 7.2 Scrapy 进阶设置

7.3.1 项目相关设置

在这里可以找到一些与具体项目相关的管理设置，如 `BOT_NAME`、`SPIDER_MODULES` 等。你可以快速浏览一下这些设置的文档，因为它们会提升具体用例的生产效率，不过通常情况下，Scrapy 的 `startproject` 和 `genspider` 命令都已经提供了合理的默认值，即使不显式修改它们，也能很好地运行。邮件相关的设置，比如 `MAIL_FROM`，可以让你配置 `MailSender` 类，该类目前用于统计邮件信息（另外参见 `STATSMAILER_RCPTS`）以及内存使用信息（另外参见 `MEMUSAGE_NOTIFY_MAIL`）。还有两个环境变量：`SCRAPY_SETTINGS_MODULE` 以及 `SCRAPY_PROJECT`，可以让你调整 Scrapy 项目与其

他项目集成的方式，比如 Django 项目。scrapy.cfg 还允许你调整设置模块的名称。

7.3.2 Scrapy 扩展设置

这些设置能够让你扩展并修改 Scrapy 的几乎所有方面。这些设置中最重要的当属 ITEM_PIPELINES。它可以让你在项目中使用 Item 处理管道。第 9 章会看到更多的例子。除了管道之外，还可以通过不同的方式扩展 Scrapy，其中一些将会在第 8 章中进行总结。COMMANDS_MODULE 允许我们添加常用命令。比如，可以在 properties/hi.py 文件中添加如下内容。

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

当在 settings.py 文件中添加 COMMANDS_MODULE='properties.hi' 时，就激活了这个小命令，我们可以在 Scrapy 帮助中看到它，并且通过 scrapy hi 运行。在命令的 default_settings 中定义的设置，会被合并到项目的设置当中，并覆盖默认值，不过其优先级低于 settings.py 文件或命令行中设定的设置。

Scrapy 使用 `__BASE` 字典（比如 `FEED_EXPORTERS_BASE`）存储不同框架扩展的默认值，并允许我们在 settings.py 文件或命令行中，通过设置它们的非 `__BASE` 版本（比如 `FEED_EXPORTERS`）进行自定义。

最后，Scrapy 使用 `DOWNLOADER`、`SCHEDULER` 等设置，保存系统基本组件的包/类名。我们可以继承默认的下载器（`scrapy.core.downloader.Downloader`），重载一些方法，然后将 `DOWNLOADER` 设置为自定义的类。这样可以让开发者大胆地对新特性进行实验，并且可以简化自动化测试过程，不过除非你明确了解自己做的事情，否则不要轻易修改这些设置。

7.3.3 下载调优

`RETRY_*`、`REDIRECT_*` 以及 `METAREFRESH_*` 设置分别用于配置重试、重定向以

及元刷新中间件。例如，将 `REDIRECT_PRIORITY_ADJUST` 设为 2，意味着每次发生重定向时，新请求将会在所有非重定向请求完成服务后才会被调度；而将 `REDIRECT_MAX_TIMES` 设置为 20，则表示在执行 20 次重定向后，下载器将会放弃尝试，并返回目前所见到的内容。这些设置在爬取一些运行不太正常的网站时非常有用，不过在大多数情况下，默认值已经可以提供很好的服务了。它同样也适用于 `HTTPERROR_ALLOWED_CODES` 以及 `URLLENGTH_LIMIT`。

7.3.4 自动限速扩展设置

`AUTOTHROTTLE_*` 设置用于启用并配置自动限速扩展。虽然对它有很大期望，但从实践来看，我发现它往往有些保守，不容易调整。它使用下载延迟，来了解我们和目标服务器的负载情况，并据此调整下载器的延迟。如果你很难找到 `DOWNLOAD_DELAY` 的最佳值（默认为 0），就会发现该模块很有用。

7.3.5 内存使用扩展设置

`MEMUSAGE_*` 设置用于启用并配置内存使用扩展。当超出内存限制时，将会关闭爬虫。当运行在共享环境时，该设置非常有用，因为此时需要非常礼貌的行为。大多数情况下，你可能会发现它只有在接收报警邮件时才会有用，此时我们需要将 `MEMUSAGE_LIMIT_MB` 设置为 0，禁用关闭爬虫的功能。该扩展只在类 UNIX 平台上适用。

`MEMDEBUG_ENABLED` 和 `MEMDEBUG_NOTIFY` 用于启用并配置内存调试扩展，在爬虫关闭时打印出仍然存活的引用数量。总之，追踪内存泄露不是一件简单而有趣的事情（好吧，它还是有一些乐趣的）。我们可以阅读 *Debugging memory leaks with trackref* 这篇优秀的文档，了解更多内存泄露排查的方法，不过最重要的建议是，保持你的爬虫相对简短、批量处理，并且需要根据服务器的能力运行。我认为没有什么好的理由可以让我们批量运行超过几千页或几分钟。

7.3.6 日志和调试

最后，还有一些日志和调试功能。`LOG_ENCODING`、`LOG_DATEFORMAT` 和 `LOG_FORMAT` 可以用来调整日志格式，当准备使用日志管理解决方案时（比如 **Splunk**、

Logstash 和 Kibana)，会发现这些设置非常有用。DUPEFILTER_DEBUG 和 COOKIES_DEBUG 将会帮助你调试相对复杂的情况，比如得到的请求数少于预期或会话意外丢失。

7.4 本章小结

通过阅读本章，我相信与从头开始编写爬虫相比，你能体会到使用 Scrapy 功能所带来的深度和广度。如果你想调整或扩展 Scrapy 的功能，可以有很多选项，我们将会在下章中看到它们。

第 8 章

Scrapy 编程

到目前为止，我们编写的爬虫主要用于定义爬取数据源的方式以及如何从中抽取信息。除了爬虫外，**Scrapy** 还提供了能够调整其大多数方面功能的机制。比如，你可能会发现自己经常在处理如下的一些问题。

1. 你需要从同一个项目的其他爬虫中复制、粘贴大量代码。重复的代码与数据更加相关（比如，执行字段计算），而不是数据源。
2. 你需要编写脚本，对 `Item` 进行后处理，执行像删除重复条目或后置处理值的事情。
3. 你在不同的项目中有重复的代码，用于处理基础架构。比如，你可能需要登录并向专有仓库传输文件，向数据库中添加 `Item` 或在爬虫执行完成时触发后置处理操作。
4. 你发现 **Scrapy** 的某个方面与你希望的功能并不完全一致，你想在自己的大部分项目中使用自定义或变通的方案。

Scrapy 开发者所设计的架构，能够为我们解决这些常见的问题。我们将会在本章后续部分研究该架构。不过我们首先介绍支持 **Scrapy** 的引擎，该引擎叫作 **Twisted**。

8.1 Scrapy 是一个 Twisted 应用

Scrapy 是一个内置使用了 **Python** 的 **Twisted** 框架的抓取应用。**Twisted** 确实有些与众

不同，因为它是事件驱动的，并且鼓励我们编写异步代码。习惯它需要一些时间，不过我们将通过只学习和 Scrapy 有关的部分，从而让任务变得相对简单一些。我们还可以在错误处理方面轻松一些。GitHub 上的完整代码会有更加彻底的错误处理，不过在本书中将忽略该部分。

让我们从头开始。Twisted 与众不同是因为它的主要口号。



在任何情况下，都不要编写阻塞的代码。

代码阻塞的影响很严重，而可能造成代码阻塞的原因包括：

- 代码需要访问文件、数据库或网络；
- 代码需要派生新进程并消费其输出，比如运行 shell 命令；
- 代码需要执行系统级操作，比如等待系统队列。

Twisted 提供的方法允许我们执行上述所有操作甚至更多操作时，无需再阻塞代码执行。

为了展示两种方式的不同，我们假设有一个典型的同步抓取应用（见图 8.1）。假设该应用包含 4 个线程，并且在一个给定的时刻，其中 3 个线程处于阻塞状态，用于等待响应，而另一个线程被阻塞，用于执行数据库写访问以保存 Item。在任何给定时刻，很有可能无法找到抓取应用的一个执行其他事情的线程，只能等待一些阻塞操作完成。当阻塞操作完成时，一些计算操作可能占用几微秒，然后线程再次被阻塞，执行其他阻塞操作，这很可能持续至少几毫秒的时间。总体来说，服务器不会是空闲的，因为它运行了几十个应用程序，并使用了上千个线程，因此，在一些细致的调优后，CPU 才能够合理利用。

Twisted/Scrapy 的方式更倾向于尽可能使用单线程。它使用现代操作系统的 I/O 复用功能（参见 `select()`、`poll()` 和 `epoll()`）作为“挂起器”。在通常会有阻塞操作的地方，比如 `result = i_block()`，Twisted 提供了一个可以立即返回的替代实现。不过，它并不是返回真实值，而是返回一个 hook，比如 `deferred = i_dont_block()`，在这里可以挂起任何想要运行的功能，而不用管什么时候返回值可用（比如，`deferred`。

`addCallback (process_result)`)。一个 Twisted 应用是由一组此类延迟运行的操作组成的。Twisted 唯一的主线程被称为 Twisted 事件反应器线程，用于监控挂起器，等待某个资源变为可用（比如，服务器返回响应到我们的 Request 中）。当该事件发生时，将会触发链中最前面的延迟操作，执行一些计算，然后依次触发下面的操作。部分延迟操作可能会引发进一步的 I/O 操作，这样就会造成延迟操作链回到挂起器中，如果可能的话，还会释放 CPU 以执行其他功能。由于我们使用的是单线程，因此不会存在额外线程所需的上下文切换以及保存资源（如内存）所带来的开销。也就是说，我们使用该非阻塞架构时，只需一个线程，就能达到类似使用数千个线程才能达到的性能。

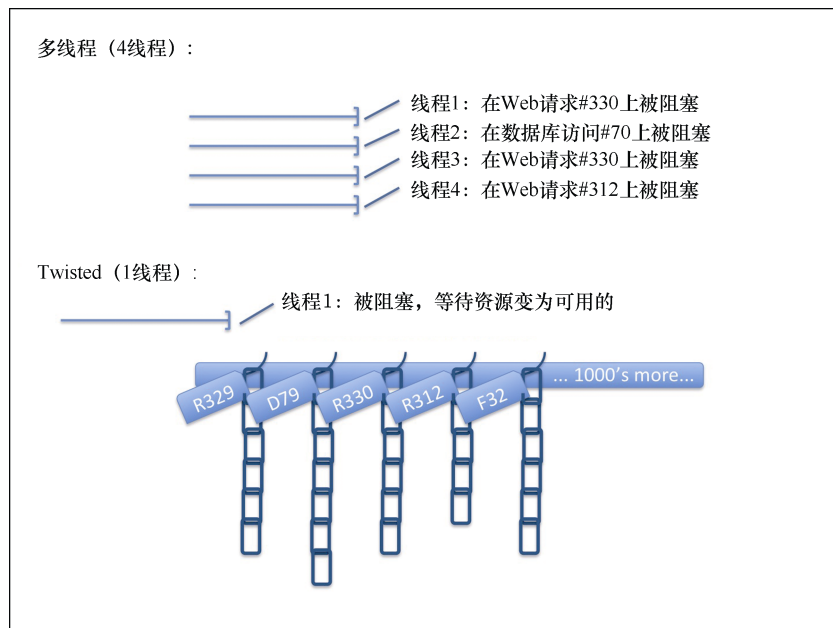


图 8.1 多线程代码和 Twisted 异步代码的对比

坦率地说，操作系统开发人员花费了数十年的时间优化线程操作，以使它们速度更快。性能的争论没有以前那么强烈了。有一件大家都认同的事情是，为复杂应用编写正确的线程安全代码非常困难。当你克服考虑延迟/回调所带来的最初冲击后，会发现 Twisted 代码要比多线程代码简单得多。`inlineCallbacks` 生成器工具使得代码更加简单，我们将会在后继章节进一步讨论它。



可以说,到目前为止,最成功的非阻塞 I/O 系统是 Node.js,主要是因为它以高性能和并发性作为出发点,没有人去争论这是好事还是坏事。每个 Node.js 应用都只用非阻塞 API。在 Java 的世界里,Netty 可能是最成功的 NIO 框架驱动应用,比如 Apache Storm 和 Spark。C++ 11 的 `std::future` 和 `std::promise` (与延迟操作非常类似) 通过使用 `libevent` 或纯 POSIX 这些库,使得编写异步代码更加简单。

8.1.1 延迟和延迟链

延迟机制是 Twisted 提供的最基础的机制,能够帮助我们编写异步代码。Twisted API 使用延迟机制,允许我们定义发生某些事件时所采取的动作序列。下面让我们具体看一下。



你可以从 GitHub 上获取本书的全部源代码。如果想要下载本书代码,可以使用 `git clone https://github.com/scalingexcellence/scrapybook`。本章的完整代码在 `ch08` 目录中,其中本示例的代码在 `ch08/deferreds.py` 文件中,你可以使用 `./deferreds.py 0` 运行该代码。

可以使用 Python 控制台运行如下的交互式实验。

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

可以看到, Deferred 本质上代表的是一个无法立即获取的值。当触发 d 时 (调用其 callback 方法), 其 called 状态变为 True, 而 result 属性被设置为在回调方法中设定的值。

```
>>> # 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
```

延迟机制最强大的功能就是可以在设定值时串联其他要被调用的操作。在上面的例子中, 添加了一个 foo() 函数作为 d 的回调函数。当通过调用 callback(3) 触发 d 时, 会调用函数 foo(), 打印消息, 并将其返回值设为 d 最终的 result 值。

```
>>> # 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()
>>> a.addCallback(b_callback).addCallback(on_done)
>>> status(a, b)
```

```
[('N/A', 2), ('N/A', 0)]
>>> a.callback(3)
b_callback called with arg = 3
>>> status(a, b)
[(<Deferred at 0x10e7209e0>, 1), ('N/A', 1)]
>>> b.callback(4)
on_done called with arg = 4
>>> status(a, b)
[(4, 0), (None, 0)]
```

该示例展示了更加复杂的延迟行为。我们看到该示例中有一个普通的延迟 `a`，和之前例子中创建的一样，不过这次它有两个回调方法。第一个是 `b_callback()`，返回值是另一个延迟 `b`，而不是一个值。第二个是 `on_done()` 打印函数。我们还有一个 `status()` 函数，用于打印延迟状态。在两个延迟完成初始化之后，得到了相同的状态：`[('N/A', 2), ('N/A', 0)]`，这意味着两个延迟都还没有被触发，并且第一个延迟有两个回调，而第二个没有回调。然后，当触发第一个延迟时，我们得到了一个奇怪的 `[(<Deferred at 0x10e7209e0>, 1), ('N/A', 1)]` 状态，可以看出现在 `a` 的值是一个延迟（实际上就是 `b` 延迟），并且目前它还有一个回调，这种情况是合理的，因为 `b_callback()` 已经被调用，只剩下了 `on_done()`。意外的情况是现在 `b` 也包含了一个回调。实际上是在后台注册了一个回调，一旦触发 `b`，就会更新它的值。当其发生时，`on_done()` 依然会被调用，并且最终状态会是 `[(4, 0), (None, 0)]`，和我们预期的一样。

```
>>> # 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)]
```

而另一方面,如果像 Experiment 3.b 所示,b 先于 a 被触发,状态将会变为 [('N/A', 2), (4, 0)], 然后当 a 被触发时,两个回调都会被调用,最终状态与之前一样。有意思的是,不管顺序如何,最终结果都是相同的。两个例子唯一的不同是,在第一个例子中,b 值保持延迟的时间更长一些,因为它是第二个被触发的,而在第二个例子中,b 首先被触发,并且从该时刻起,它的值就会在需要时被立即使用。

此时,你应该已经对什么是延迟、它们是如何串联起来表示尚不可用的值,有了不错的理解。我们将通过第 4 个例子结束这一部分的研究,在该示例中,将展示如何触发依赖于多个其他延迟的方法。在 Twisted 的实现中,将会使用 `defer.DeferredList` 类。

```
>>> # 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)]
```

可以注意到,尽管 for 循环语句只触发了 5 个延迟中的 4 个, `on_done()` 仍然需要等到列表中所有延迟都被触发后才会调用,也就是说,要在最后的 `deferreds[4].callback()` 之后调用。`on_done()` 的参数是一个元组组成的列表,每个元组对应一个延迟,其中包含两个元素,分别是表示成功的 `True` 或表示失败的 `False`,以及延迟的值。

8.1.2 理解 Twisted 和非阻塞 I/O——一个 Python 故事

既然我们已经掌握了原语,接下来让我告诉你一个 Python 的小故事。该故事中所有人物均为虚构,如有雷同纯属巧合。

```
# ~~ 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"])
```

运行该代码。

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

我们得到的是顺序的执行。4 位客户，每人执行 3 秒，意味着总共需要 12 秒的时间。这种方式的扩展性不是很好，因此我们将在第二个例子中添加多线程。

```
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)])

```

按照下述方式运行这段代码。

```

$ ./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

```

在这段代码中，使用了 5 个线程并行执行。15 个客户，每人 3 秒，总共需要执行 45 秒，而当使用 5 个并行的线程时，最终只花费了 9 秒钟。不过代码有些难看。现在代码的一部分只用于管理并发性，而不是专注于算法或业务逻辑。另外，输出也变得混乱并且可读性很差。即使是让很简单的多线程代码正确运行，也有很大难度，因此我们将转为使用 Twisted。

```

# 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()
```

现在运行该代码。

```
$ ./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
```

此时，我们在没有使用多线程的情况下，就获得了良好运行的代码，以及漂亮的输出结果。我们并行处理了所有的 15 位客户，也就是说，应当执行 45 秒的计算只花费了 3 秒钟！技巧就是将所有阻塞调用的 `sleep()` 替换为 Twisted 对应的 `task.deferLater()` 以及回调函数。由于处理现在发生在其他地方，因此可以毫不费力地同时为 15 位客户服务。



刚才提到前面的处理此时是在其他地方执行的。这是在作弊吗？答案当然不是。算法计算仍然在 CPU 中处理，不过与磁盘和网络操作相比，CPU 操作速度很快。因此，将数据传给 CPU、从一个 CPU 发送或存储数据到另一个 CPU 中，占据了大部分时间。我们使用非阻塞的 I/O 操作，为 CPU 节省了这些时间。这些操作，尤其是像 `task.deferLater()` 这样的操作，会在数据传输完成后触发回调函数。

另一个需要非常注意的地方是 Goodmorning from Twisted developer 以及 Bye from Twisted developer! 消息。在代码启动时，它们就都被立即打印了出来。如果代码过早地到达该点，那么应用实际是什么时候运行的呢？答案是 Twisted 应用（包括 Scrapy）完全运行在 reactor.run() 上！当调用该方法时，必须拥有应用程序预期使用的所有可能的延迟链（相当于前面故事中建立 CRM 系统的步骤和流程）。你的 reactor.run()（故事中的秘书）执行事件监控以及触发回调。



reactor 的主要规则是：只要是快速的非阻塞操作就可以做任何事。

非常好！不过虽然代码没有了多线程时的混乱输出，但是这里的回调函数还是有一些难看！因此，我们将引入下一个例子。

```
# 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()
```

以如下方式运行该代码。

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

上述代码和之前那个版本的代码看起来基本一样，不过更加优雅。inlineCallbacks 生成器使用了一些 Python 机制让 inline_install() 的代码能够暂停和恢复。inline_install() 变为延迟函数，并且为每位客户并行执行。每当执行 yield 时，执行会在当前的 inline_install() 实例上暂停，当 yield 的延迟函数触发时再恢复。

现在唯一的问题是，如果不是只有 15 个客户，而是 10000 个，该代码会无耻地同时启动 10000 个处理序列（调用 HTTP 请求、数据库写操作等）。这样可能会正常运行，也可能造成各种各样的失败。在大规模并发应用中，比如 Scrapy，一般需要将并发量限制到可接受的水平。在本例中，可以使用 `task.Cooperator()` 实现该限制。Scrapy 使用了同样的机制在 item 处理管道中限制并发量（`CONCURRENT_ITEMS` 设置）。

```
@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 ~*~
```

运行该代码。

```
$ ./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
```

可以看到，现在有类似于 5 个客户的处理槽。如果想要处理一个新的客户，只有在存在空槽时才可以开始，实际上，在这个例子中客户处理的时间总是相同的（3 秒），因此会造成 5 位客户会在同一时间被批量处理的情况。最后，我们得到了和多线程示例中相同的性能，不过现在只使用了一个线程，代码更加简单并且更容易正确编写。

祝贺你，坦白地说，现在你得到了对于 Twisted 和非阻塞 I/O 编程的一份非常严谨的介绍。

8.2 Scrapy 架构概述

图 8.2 所示为 Scrapy 的架构。

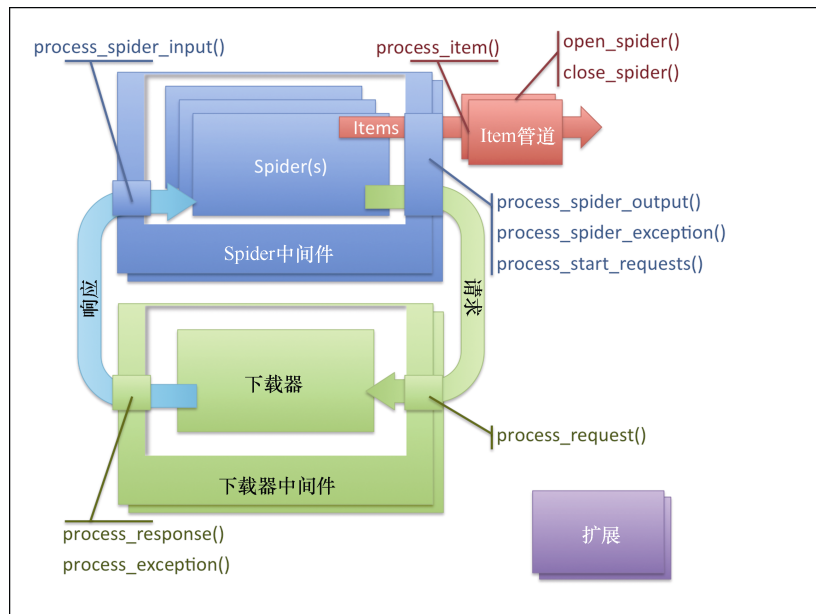


图 8.2 Scrapy 架构

你可能已经注意到，该架构运行在我们熟悉的三类对象之上：Request、Response 以及 Item。我们的爬虫就在架构的核心位置，它们创建 Request，处理 Response，生成 Item 和更多的 Request。

爬虫生成的每个 Item 都使用其 `process_item()` 方法由 Item 管道序列执行后置处理。通常情况下，`process_item()` 会修改 Item，然后以返回这些 Item 的方式将其传给后续的管道。有时候（比如冗余或非法数据的情况），我们可能需要放弃一个 Item，此时可以通过抛出 `DropItem` 异常的方式实现。这种情况下，后续的管道将不会再接收该 Item。如果我们提供了 `open_spider()` 和/或 `close_spider()` 方法，那么爬虫会对应地在开始和结束爬虫时调用该方法。这里是我们进行初始化和清理工作的时机。Item 管道通常用于执行问题域名或基础结构的操作，比如清理数据、向数据库插入 Item 等。你还会发现自己会在项目之间很大程度地复用它们，尤其是当处理基础架构细节时。第 4 章中使用过的 Appery.io 管道，即通过少量配置上传 Item 到 Appery.io 的工作，就是用 Item 管道执行基础架构工作的一个例子。

我们通常会从爬虫发送 Request，并得到返回的 Response，来进行工作。Scrapy 以透明的方式负责 Cookie、权限认证、缓存等，我们所需要的就是偶尔调整一些设置。这其中大部分功能是在下载器中间件中实现的。它们通常都非常复杂，在处理 Request/Response 内部构件时有着很高的技巧。你可以创建自定义的中间件，以使 Scrapy 按照你要求的方式处理 Request。通常，成功的中间件可以在多个项目中复用，并且可以向其他 Scrapy 开发者提供有用的功能，因此向社区分享是个不错的选择。你没有必要经常编写下载器中间件。如果你想了解默认的下载器中间件，可以查看 Scrapy 的 Github 仓库中 `settings/default_settings.py` 文件的 `DOWNLOADER_MIDDLEWARES_BASE` 设置。

下载器是真正执行下载的引擎。除非你是 Scrapy 的代码贡献者，否则不要修改它。

有时候，你可能需要编写爬虫中间件（见图 8.3）。这些中间件在爬虫之后且所有下载器中间件之前处理 Request；而在处理 Response 时，则是相反的顺序。使用下载器中间件，可以做很多事情，比如重写所有 URL，使用 HTTPS 代替 HTTP，而不用管爬虫从页面中抽取出来的内容是什么。它可以实现特定于项目需求的功能，并分享给所有

的爬虫。下载器中间件和爬虫中间件最主要的区别是，当下载器中间件获取一个 Request 时，只会返回一个 Response。而爬虫中间件可以在对某些 Request 不感兴趣时舍弃掉它们，或者对每个输入的 Request 都发出多个 Request，用来完成你的应用程序的目标。可以说爬虫中间件是针对 Request 和 Response 的，而 Item 管道是针对 Item 的。爬虫中间件同样也接收 Item，不过通常情况下不会对其进行修改，因为在 Item 管道中进行这些操作更加容易。如果你了解默认的爬虫中间件，可以在 Scrapy 的 Git 上查看 `settings/default_settings.py` 文件的 `SPIDER_MIDDLEWARES_BASE` 设置。

最后还有一个部分是扩展。扩展非常常见，实际上其常见程度仅次于 Item 管道。它们是在爬取工作启动时加载的普通类，可以访问设置、爬虫、注册回调信号以及定义自己的信号。信号是一类基础的 Scrapy API，它可以让回调函数在系统中发生某些事情时进行调用，比如 Item 被抓取、丢弃时或爬虫开启时。有很多非常有用的预定义信号，我们将会在后面见到其中的一部分。某种意义上讲，扩展有些博而不精，它能够让你写出任何可以想到的工具，但又无法给你实际的帮助（比如像 Item 管道的 `process_item()` 方法）。我们必须将其 hook 到信号上，自己实现需要的功能。例如，在达到指定页数或 Item 个数后停止爬取，就是通过扩展实现的。如果想要了解默认的扩展，可以从 Scrapy 的 Git 上查看 `settings/default_settings.py` 文件的 `EXTENSIONS_BASE` 设置。

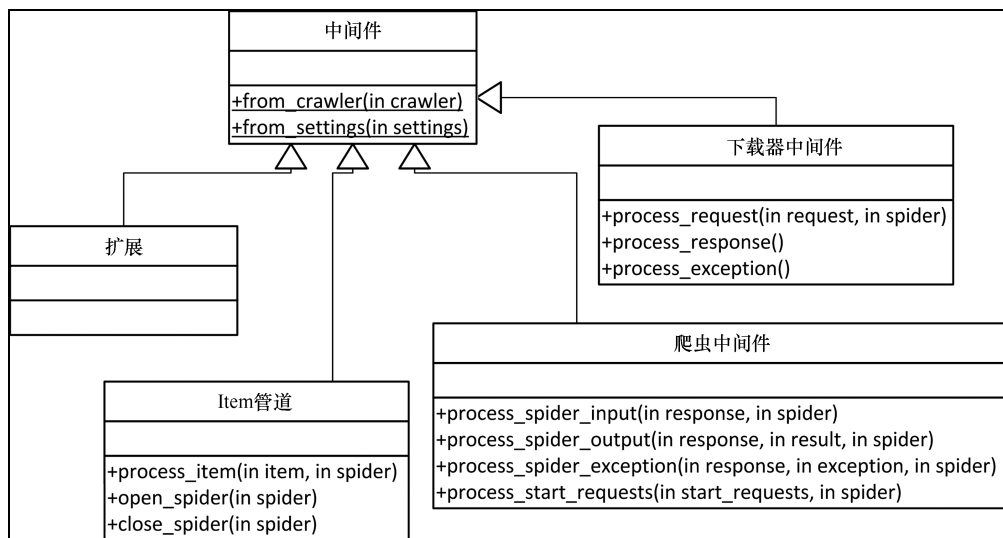


图 8.3 中间件架构

更严格地说, Scrapy 把所有这些类都当作是中间件 (通过 Middleware Manager 类的子类管理), 允许我们通过实现 `from_crawler()` 或 `from_settings()` 类方法, 分别从 Crawler 或 Settings 对象初始化它们。由于 Settings 可以从 Crawler 中轻松获取 (`crawler.settings`), 因此 `from_crawler()` 是更加流行的方式。如果不需要 Settings 或 Crawler, 可以不去实现它们。

表 8.1 可以帮助你在针对指定问题时决定最好的机制。

表 8.1

问题	解决方案
一些只针对于我正在爬取的网站的内容	修改爬虫
修改或存储 Item——特定领域, 可能在项目间复用	编写 Item 管道
修改或丢弃 Request/Response——特定领域, 可能在项目间复用	编写爬虫中间件
执行 Request/Response——通用, 比如支持一些定制化登录模式或处理 Cookie 的特定方式	编写下载器中间件
所有其他问题	编写扩展

8.3 示例 1: 非常简单的管道

假设我们有一个包含几个爬虫的应用, 以 Python 常见格式提供爬取日期。数据库需要将其转为字符串格式, 以便进行索引。我们不想编辑爬虫, 因为爬虫的数量比较多。此时可以怎么做呢? 使用一个非常简单的管道对 Item 进行后置处理, 执行需要的转换即可。让我们看看它是如何工作的。

```
from datetime import datetime

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

如你所见, 这里只有一个包含 `process_item()` 方法的简单类。这是我们为了这

个简单管道所需要做的所有事情。我们可以复用第 3 章中的爬虫，将前面的代码写入 `pipelines` 目录的 `tidyup.py` 文件中。

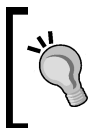


可以将这个 Item 管道的代码放到任何地方，不过为其创建一个单独的目录是一个好主意。

现在，需要编辑项目的 `settings.py` 文件，将 `ITEM_PIPELINES` 设置为

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

前面代码字典中的数字 100，用于定义管道连接的顺序。如果另一个管道有更小的数值，它将在该管道之前优先处理 Item。



本示例的完整代码位于 `ch08/properties` 目录中。

现在，可以运行该爬虫了。

```
$ 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'],
```

和我们期望的一样，日期现在被格式化为 ISO 字符串了。

8.4 信号

信号提供了一种为系统中发生的事件添加回调的机制，比如当爬虫开启时或当 item 被抓取时。可以使用 `crawler.signals.connect()` 方法 hook 到它们上（下一节将会给出使用它的一个示例）。信号总共有 11 个，理解它们的最简单方式可能就是在实践中看到它们。我创建了一个项目，在其中创建了一个扩展，hook 了所有可以使用的信号。

另外，我还创建了一个 Item 管道、一个下载器和一个爬虫中间件，可以记录所有的方法调用。该项目使用的爬虫非常简单，只对两个 item 进行 yield 操作，然后抛出异常。

```
def parse(self, response):
    for i in range(2):
        item = HooksasyncItem()
        item['name'] = "Hello %d" % i
        yield item
    raise Exception("dead")
```

在第二个 item 中，我配置了 Item 管道，以抛出 DropItem 异常。



本示例的完整代码可以从 `ch08/hooksasync` 得到。

使用该项目，我们可以更好地理解某个信号是什么时候发出的。请查看如下执行中日志行之间的注释（为了简短起见，省略了部分行）。

```
$ 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
```

虽然只有 11 个信号，可能会感觉比较有限，但是每个 Scrapy 的默认中间件都是只使用它们实现的，因此它们肯定够用。请注意，除了 `spider_idle`、`spider_error`、`request_scheduled`、`response_received` 和 `response_downloaded` 以外的所有其他信号，都可以返回延迟，而不是真实值。

8.5 示例 2：测量吞吐量和延时的扩展

当我们在第 9 章中添加管道后，测量吞吐量（每秒的 item 数）和延时（计划后和下载后的时间）的变化是一件很有意思的事情。

Scrapy 扩展中已经包含了一个测量吞吐量的扩展，即日志统计扩展（`scrapy/extensions/logstats.py`），我们将会以此为起点。要想测量延时，需要 hook 一些信号，包括 `request_scheduled`、`response_received` 和 `item_scraped`。我们对每个信号记录时间戳，并通过累计多次取平均值的方式减去适当的计算延时。通过观察这些信号提供的回调参数，会发现一些讨厌的东西。`item_scraped` 只在 `Response` 中获得，`request_scheduled` 只在 `Request` 中获得，而 `response_received` 则是两者中都有。幸运的是，我们不需要任何特殊的技巧来传递这些值。每个 `Response` 都有一个 `Request` 成员，回指其 `Request`，更好的是它拥有我们在第 5 章中看到的 `meta` 字典，它和原始 `Request` 中的一样，而不管是否存在重定向。非常好，我们可以在这里存储时间戳了！



实际上,这并不是我的主意。同样的机制已经在 AutoThrottle 扩展 (scrapy/extensions/throttle.py) 中使用了。在该扩展中,使用了 request.meta.get ('download_latency'), 其中 download_latency 是在 scrapy/core/downloader/webclient.py 下载器中进行计算的。在编写中间件时,最快的改善方式就是让自己熟悉 Scrapy 默认的中间件代码。

下面是扩展的代码。

```
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
```

前两个方法非常重要，因为它们很通用。它们使用 Crawler 对象初始化中间件。你会发现这些代码几乎出现在每个重要的中间件当中。from_crawler(cls, crawler) 是获取 Crawler 对象的方式。然后，可以注意到在 __init__() 方法中，访问了 crawler.settings，并且会在其未设置时抛出 NotConfigured 异常。你会看到很多 FooBar 扩展，用于检查相应的 FOOBAR_ENABLED 设置，如果没有设置或者设置为 False 时，将会抛出异常。这是一种非常常见的模式，是为了方便将中间件包含在对应的 settings.py 设置中（比如 ITEM_PIPELINES），但是默认情况下是禁用的，除非通过其对应的设置显式启用。许多默认的 Scrapy 中间件（比如 AutoThrottle 或 HttpCache）都使用了这种模式。在本例中，我们的扩展会保持 LATENCIES_INTERVAL 的禁用状态，除非已经对其进行了设置。

在 __init__() 方法的后面一部分代码中，我们使用 crawler.signals.connect()，为所有感兴趣的信号都注册了回调，并初始化了一些成员变量。这个类的剩余部分实现了信号处理器。在 _spider_opened() 中，我们初始化了一个计时器，会每隔 LATENCIES_INTERVAL 秒调用 _log() 方法；在 _spider_closed() 中，我们停止了该计时器。在 _request_scheduled() 和 _response_received() 中，我们在 request.meta 中存储了时间戳；而在 _item_scraped() 中，我们累计两次延时（从计划/接收开始直到当前时间），并递增抓取到的 Item 的数量。在 _log() 方法中，我们计算了平均值，格式化并打印出消息，重置累加器以开始另一个采样周期。



任何在多线程上下文中编写类似代码的人，都会意识到上述代码中没有使用互斥锁。本例可能还不是特别复杂，不过编写单线程代码仍然要更加简单，并且在更加复杂的场景下可以很好地扩展。

我们可以将该扩展的代码添加到 `latencies.py` 模块中，放到和 `settings.py` 同级的目录下。如果想要启用该扩展，只需在 `settings.py` 文件中添加如下两行。

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

我们可以像平时那样运行它。

```
$ 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: Scraped 0 items at 0.0 items/sec, average latency: 0.00 sec and
average time in pipelines: 0.00 sec
INFO: Scraped 115 items at 23.0 items/s, avg latency: 0.84 s and avg time
in pipelines: 0.12 s
INFO: Scraped 125 items at 25.0 items/s, avg latency: 0.78 s and avg time
in pipelines: 0.12 s
```

日志的第一行来自日志统计扩展，而接下来的各行来自我们的扩展。可以看到吞吐量是每秒 25 个 item，平均时延是 0.78 秒，我们在下载后几乎没有花费时间处理。通过利特尔法则，我们得到系统中 item 的数量为 $N = S \cdot T = 43 \cdot 0.45 \cong 19$ 。无论设置的 `CONCURRENT_REQUESTS` 和 `CONCURRENT_REQUESTS_PER_DOMAIN` 是多少，即便没有触及 100% 的 CPU，出于某些原因，也不应该使其超过 30。我们可以在第 10 章中了解更多相关内容。

8.6 中间件延伸

本节是为好奇的读者提供的，而不再是开发者。如果只是编写基础或中级的 Scrapy

扩展的话，你并不需要了解这些内容。

如果查看 `scrapy/settings/default_settings.py` 文件，就会发现在默认设置中有很多类名。Scrapy 大量使用了依赖注入机制，可以让我们自定义和扩展许多内部对象。例如，一些人可能希望支持除了文件、HTTP、HTTPS、S3 以及 FTP 这些在 `DOWNLOAD_HANDLERS_BASE` 设置中定义好的协议以外的更多协议。要想实现这一点，只需要创建一个下载处理器类，并在 `DOWNLOAD_HANDLERS` 设置中添加映射即可。最困难的部分是找出你的自定义类必须包含哪些接口（即需要实现哪些方法），因为大部分接口都不是显式的。你必须阅读源代码，查看这些类是如何使用的。最好的办法是从已有的实现开始，将其修改为令自己满意的版本。不过，这些接口在近期的 Scrapy 版本中已经逐渐趋于稳定，因此我将尝试在图 8.4 中将它们和 Scrapy 核心类一起记录成文档（这里省略了前面已经提及的中间件架构）。

核心类位于图 8.4 的左上角。当人们使用 `scrapy crawl` 时，Scrapy 就会使用 `CrawlerProcess` 对象创建我们熟悉的 `Crawler` 对象。`Crawler` 对象是最重要的 Scrapy 类。它包括 `settings`、`signals` 以及 `spider`。在名为 `extensions.crawler.engine` 的 `ExtensionManager` 对象中，还包含所有的扩展，这将带领我们来到另一个非常重要的类——`ExecutionEngine`。在该类中，包含了 `Scheduler`、`Downloader` 以及 `Scraper`。`URL` 通过 `Scheduler` 进行计划，通过 `Downloader` 下载，通过 `Scraper` 进行后置处理。毫无疑问，`Downloader` 包含 `DownloaderMiddleware` 和 `DownloadHandler`，而 `Scraper` 包含 `SpiderMiddleware` 和 `ItemPipeline`。4 个 `MiddlewareManager` 也都拥有其自己的小架构。在 Scrapy 中，`feed` 输出是以扩展的形式实现的，即 `FeedExporter`。它包含两个独立的结构，一个用于定义输出格式，而另一个用于存储类型。这就允许我们可以通过调整输出的 `URL` 将 S3 的 XML 文件导出为命令行上的 `Pickle` 编码输出。这两个结构还可以使用 `FEED_STORAGES` 和 `FEED_EXPORTERS` 设置进行独立扩展。最后，`scrapy check` 命令使用的 `contract` 也有其自身的结构，可以使用 `SPIDER_CONTRACTS` 设置进行扩展。

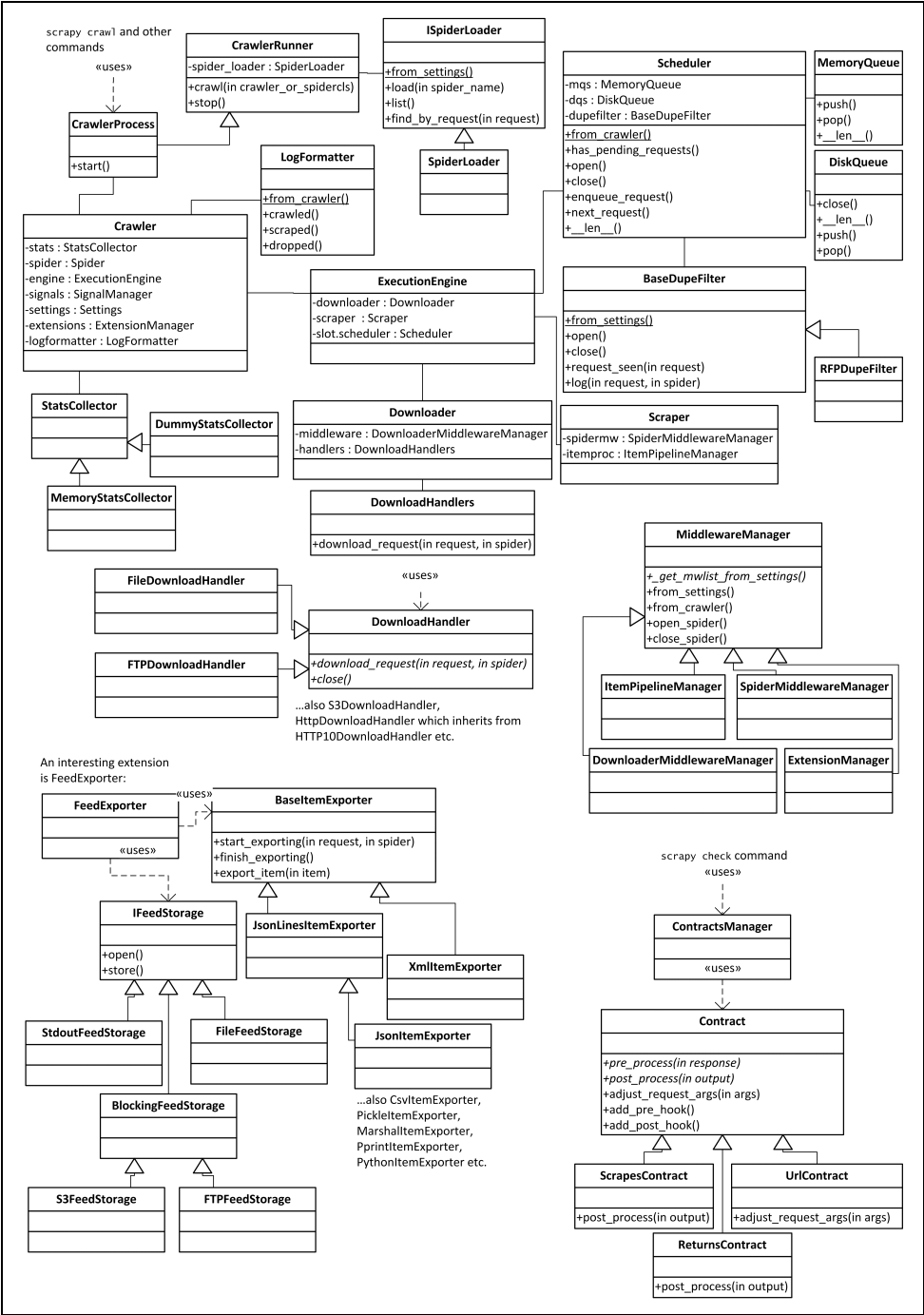


图 8.4 Scrapy 接口和核心对象

8.7 本章小结

恭喜你，你已经对 Scrapy 和 Twisted 编程有了深入了解。你可能还会多次阅读本章，并将本章作为参考使用。到目前为止，我们需要的最流行的扩展是 Item 处理管道。下一章会用它解决一些常见的问题。

第 9 章

管道秘诀

上一章讨论了使用 **Scrapy** 中间件的编程技术。本章将通过展示各种常见用例（包括消费 REST API、数据库接口、处理 CPU 密集型任务以及与遗留服务的接口），重点关注编写正确而高效的管道。

在本章中，我们将会使用几个新的服务器，你可以在图 9.1 的右侧看到这些服务器。

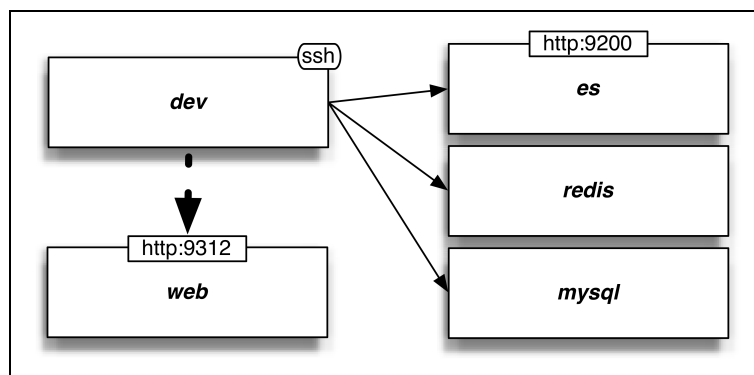


图 9.1 本章使用的系统

Vagrant 应该已经为我们创建好了这些服务器，我们可以从 **dev** 服务器中使用其主机名进行 **ping** 操作，例如 **ping es** 或 **ping mysql**。话不多说，让我们从 REST API 开始探索吧。

9.1 使用 REST API

REST 是一套用于创建现代 Web 服务的技术，其主要优点是比 SOAP 或专有 Web 服务机制更加简单，更加轻量级。软件开发人员观察发现，Web 服务经常提供的 **CRUD**（创建、读取、更新、删除[**Create、Read、Update、Delete**]）功能与 HTTP 基本操作（GET、POST、PUT、DELETE）具有相似性。另外，他们还发现典型的 Web 服务调用其所需的大部分信息时，都可以将其压缩到资源 URL 上。例如，`http://api.mysite.com/customer/john` 是一个资源 URL，它可以让我们确定目标服务器（`api.mysite.com`），实际上我正在尝试在服务器上执行和 `customers`（表）相关的操作，更具体的说就是执行和 `john`（行——主键）相关的操作。当它与其他 Web 概念（如安全认证、无状态、缓存、使用 XML 或 JSON 作为载荷等）结合时，能够通过一种强大而又简单、熟悉且可以轻松跨平台的方式，提供和使用 Web 服务。难怪 REST 可以掀起软件行业的一场风暴。

9.1.1 使用 `treq`

`treq` 是一个 Python 包，相当于基于 Twisted 应用编写的 Python `requests` 包。它可以帮助我们轻松执行 GET、POST 以及其他 HTTP 请求。想要安装该包，可以使用 `pip install treq`，不过它已经在我们的开发机中预先安装好了。

我们更倾向于选择 `treq` 而不是 `Scrapy` 的 `Request/crawler.engine.download()` 的原因是，虽然它们都很简单，但是在性能上 `treq` 更有优势，我们将会在第 10 章中看到更详细的介绍。

9.1.2 用于写入 Elasticsearch 的管道

首先，我们要编写一个将 Item 存储到 **ES**（**Elasticsearch**）服务器的爬虫。你可能会觉得从 ES 开始，甚至先于 MySQL，作为持久化机制进行讲解有些不太寻常，不过其实它是我们可以做的最简单的事情。ES 可以是无模式的，也就是说无需任何配置就能够使用它。对于我们这个（非常简单的）用例来说，`treq` 也已经足够使用。如果想要使

用更高级的 ES 功能，则需要考虑使用 `txes2` 或其他 Python/Twisted ES 包。

在我们的开发机中，已经包含正在运行的 ES 服务器了。下面登录到开发机中，验证其是否正在正常运行。

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

在宿主机浏览器中，访问 `http://localhost:9200`，也可以看到同样的结果。当访问 `http://localhost:9200/properties/property/_search` 时，可以看到返回的响应表示 ES 进行了全局性的尝试，但是没有找到任何与房产信息相关的索引。恭喜你，刚刚已经使用了 ES 的 REST API。



在本章，我们将在 `properties` 集合中插入房产信息。你可能需要重置 `properties` 集合，此时可以使用 `curl` 执行 DELETE 请求：

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

本章中管道实现的完整代码包含很多额外的细节，如更多的错误处理等，不过我将通过凸显关键点的方式，保持这里的代码简洁。



本章在 `ch09` 目录当中，其中本示例的代码为 `ch09/properties/properties/pipelines/es.py`。

从本质上说，爬虫代码只包含如下 4 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)
```

其中，前两行用于定义标准的 `process_item()` 方法，可以在其中 `yield` 延迟操作（参考第 8 章）。

第3行用于准备要插入的 data。首先，我们将 Item 转化为字典。然后使用 `json.dumps()` 将其编码为 JSON 格式。`ensure_ascii=False` 的目的是通过不转义非 ASCII 字符，使得输出更加紧凑。然后，将这些 JSON 字符串编码为 UTF-8，即 JSON 标准中的默认编码。

最后一行使用 `treq` 的 `post()` 方法执行 POST 请求，将文档插入到 ElasticSearch 中。`es_url` 存储在 `settings.py` 文件当中（`ES_PIPELINE_URL` 设置），如 `http://es:9200/properties/property`，可以提供一些基本信息，如 ES 服务器的 IP 和端口（`es:9200`）、集合名称（`properties`）以及想要写入的对象类型（`property`）。

要想启用该管道，需要将其添加到 `settings.py` 文件的 `ITEM_PIPELINES` 设置当中，并且使用 `ES_PIPELINE_URL` 设置进行初始化。

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

完成上述工作后，我们可以进入到适当的目录当中。

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

然后，开始运行爬虫。

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

如果现在再次访问 `http://localhost:9200/properties/property/_search`，可以在响应的 `hits/total` 字段中看到已经插入的条目数量，以及前 10 条结果。我们还可以通过添加 `?size=100` 参数取得更多结果。在搜索 URL 中添加 `q=` 参数时，可以在全部或特定字段中搜索指定关键词。最相关的结果将会出现在最前面。例

如, `http://localhost:9200/properties/property/_search?q=title:london`, 将会返回标题中包含"London"的房产信息。对于更加复杂的查询, 可以查阅 ES 的官方文档, 网址为: `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html`。

ES 不需要配置的原因是它可以根据我们提供的第一个属性自动检测模式(字段类型)。通过访问 `http://localhost:9200/properties/`, 可以看到其自动检测的映射关系。

让我们快速查看一下性能, 使用上一章结尾处给出的方式重新运行 `scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000`。平均延时从 0.78 秒增长到 0.81 秒, 这是因为管道的平均时间从 0.12 秒增长到了 0.15 秒。吞吐量仍然保持在每秒大约 25 个 Item。



使用管道将 Item 插入到数据库当中是不是一个好主意呢? 答案是否定的。通常情况下, 数据库提供的批量插入条目的方式可以有几个数量级的效率提升, 因此我们应当使用这种方式。也就是说, 应当将 Item 打包批量插入, 或在爬虫结束时以后置处理的步骤执行插入。我们将在最后一章中看到这些方法。不过, 许多人仍然使用 Item 管道插入数据库, 此时使用 Twisted API 而不是通用/阻塞的方法实现该方案才是正确的方式。

9.1.3 使用 Google Geocoding API 实现地理编码的管道

每个房产信息都有地区名称, 因此我们想对其进行地理编码, 也就是说找到它们对应的坐标(经度、纬度)。我们可以使用这些坐标将房产信息放到地图上, 或是根据它们到某个位置的距离对搜索结果进行排序。开发这种功能需要复杂的数据库、文本匹配以及空间计算。而使用 Google 的 Geocoding API, 可以避免上面提到的几个问题。可以通过浏览器或 `curl` 打开下述 URL 以获取数据。

```
$ 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"
  ]
}
```

我们可以看到一个 JSON 对象，当搜索"location"时，可以很快发现 Google 提供的是伦敦中心坐标。如果继续搜索，会发现同一文档中还包含其他位置。其中，第一个坐标位置是最相关的。因此，如果存在 `results[0].geometry.location` 的话，它就是我们所需要的信息。

Google 的 Geocoding API 可以使用之前用过的技术 (`treq`) 进行访问。只需几行，就可以找出一个地址的坐标位置（查看 `pipeline` 目录的 `geo.py` 文件），其代码如下。

```
@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"]})
```

该函数使用了一个和前面用过的 URL 相似的 URL，不过在这里将其指向到一个假的实现，以使其执行速度更快，侵入性更小，可离线使用并且更加可预测。可以使用 `endpoint = 'https://maps.googleapis.com/maps/api/geocode/json'` 来访问 Google 的服务器，不过需要记住的是 Google 对请求有严格的限制。`address` 和

sensor 的值都通过 `treq` 的 `get()` 方法的 `params` 参数进行了自动 URL 编码。`treq.get()` 方法返回了一个延迟操作，我们对其执行 `yield` 操作，以便在响应可用时恢复它。对 `response.json()` 的第二个 `yield` 操作，用于等待响应体加载完成并解析为 Python 对象。此时，我们可以得到第一个结果的位置信息，将其格式化为字典后，使用 `defer.returnValue()` 返回，该方法是从使用 `inlineCallbacks` 的方法返回值的适当的方式。如果任何地方存在问题，该方法会抛出异常，并通过 `Scrapy` 报告给我们。

通过使用 `geocode()`，`process_item()` 可以变为一行代码，如下所示。

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

我们可以在 `ITEM_PIPELINES` 设置中添加并启用该管道，其优先级数值应当小于 `ES` 的优先级数值，以便 `ES` 获取坐标位置的值。

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

我们启用调试数据，运行一个快速的爬虫。

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

```

现在，可以看到 `Item` 中包含了 `location` 字段。太好了！不过当使用真实的 Google API 的 URL 临时运行它时，很快就会得到类似下面的异常。

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

这是我们在完整代码中放入的一个检查，用于确保 `Geocoding API` 的响应中 `status` 字段的值是 `OK`。如果该值非真，则说明我们得到的返回数据不是期望的格式，无法被安

全使用。在本例中，我们得到了 `OVER_QUERY_LIMIT` 状态，可以清楚地说明在什么地方发生了错误。这可能是我们在许多案例中都会面临的一个重要问题。由于 Scrapy 的引擎具备较高的性能，缓存和资源请求的限流成为了必须考虑的问题。

可以访问 Geocoder API 的文档来了解其限制：“免费用户 API：每 24 小时允许 2500 个请求，每秒允许 5 个请求”。即使使用了 Google Geocoding API 的付费版本，仍然会有每秒 10 个请求的限流，这就意味着该讨论仍然是有意义的。



下面的实现看起来可能会比较复杂，但是它们必须在上下文中进行判断。而在典型的多线程环境中创建此类组件需要线程池和同步，这样就会产生更加复杂的代码。

下面是使用 Twisted 技术实现的一个简单而又足够好用的限流引擎。

```
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)
```

该代码中，延迟操作排队进入列表中，每次调用 `_allow_one()` 时依次触发它们；`_allow_one()` 检查队列是否为空，如果不是，则调用最旧的延迟操作的 `callback()`（先入先出，FIFO）。我们使用 Twisted 的 `task.LoopingCall()` API 周期性调用 `_allow_one()`。使用 `Throttler` 非常简单。我们可以在管道的 `__init__` 中对其进行初始化，并在爬虫结束时对其进行清理。

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

    def close_spider(self, spider):
        self.throttler.stop()
```

在使用想要限流的资源之前（在本例中为在 `process_item()` 中调用 `geocode()` ），需要对限流器的 `throttle()` 方法执行 `yield` 操作。

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

在第一个 `yield` 时，代码将会暂停，等待足够的时间过去之后再恢复。比如，某个时刻共有 11 个延迟操作在队列中，我们的速率限制是每秒 5 个请求，我们的代码将会在队列清空时恢复，大约为 $11/5=2.2$ 秒。

使用 `Throttler` 后，我们不再会发生错误，但是爬虫速度会变得非常慢。通过观察发现，示例的房产信息中只有有限的几个不同位置。这是使用缓存的一个非常好的机会。我们可以使用一个简单的 `Python` 字典来实现缓存，不过这种情况下将会产生竞态条件，导致不正确的 `API` 调用。下面是一个没有该问题的缓存，此外还演示了一些 `Python` 和 `Twisted` 的有趣特性。

```
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)

```

该缓存看起来和人们通常期望的有些不同。它包含两个组成部分。

- `self.deferreds_waiting`: 这是一个延迟操作的队列，等待指定键的值。
- `self.records`: 这是已经出现的键-操作对的字典。

如果查看 `find()` 实现的中间部分，就会发现如果没有在 `self.records` 中找到一个键，则会调用一个预定义的 `callback` 函数，取得缺失值 (`yield self.key_not_found_callback(key)`)。该回调函数可能会抛出一个异常。我们要如何在 Python 中以紧凑的方式存储这些值或异常呢？由于 Python 是一种函数式语言，我们可以根据是否出现异常，在 `self.records` 中存储调用延迟操作的 `callback` 或 `errback` 的小函数 (`lambda`)。在定义时，该值或异常被附加到 `lambda` 函数中。函数中对变量的依赖被称为闭包，这是大多数函数式编程语言最显著和强大的特性之一。



缓存异常有些不太常见，不过这意味着如果在第一次查找某个键时，`key_not_found_callback(key)` 抛出了异常，那么接下来对相同键再次查询时仍然会抛出同样的异常，不需要再执行额外的调用。

`find()` 实现的剩余部分提供了避免竞态条件的机制。如果要查询的键已经在进程当中，将会在 `self.deferreds_waiting` 字典中有记录。在这种情况下，我们不再额外调用 `key_not_found_callback()`，只是添加到延迟操作列表中，等待该键。当 `key_not_found_callback()` 返回，并且该键的值变为可用时，触发每个等待该键的延迟操作。我们可以直接执行 `action(d)`，而不是使用 `reactor.callFromThread()`，

不过这样就必须处理所有抛出的异常，并且会创建一个不必要的长延迟链。

使用缓存非常简单。只需在 `__init__()` 中对其初始化，并在执行 API 调用时设置回调函数即可。在 `process_item()` 中，按照如下代码使用缓存。

```
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)
```

本例的完整代码包含了更多的错误处理代码，能够对限流导致的错误重试调用（一个简单的 `while` 循环），并且还包含了更新爬虫状态的代码。



本例的完整代码文件地址为：ch09/properties/
properties/pipelines/geo2.py。

要想启用该管道，需要禁用（注释掉）之前的实现，并且在 `settings.py` 文件的 `ITEM_PIPELINES` 中添加如下代码。

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

然后，可以按照如下代码运行该爬虫。

```
$ 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,

```

可以看到，爬取延时最初由于填充缓存的原因非常高，但是很快就回到了之前的值。统计显示总共有 35 次未命中，这正是我们所用的示例数据集内不同位置的数量。显然，在本例中总共有 $1019 - 35 = 984$ 次命中缓存。如果使用真实的 Google API，并将每秒对 API 的请求数量稍微增加，比如通过将 `Throttler(5)` 改为 `Throttler(10)`，把每秒请求数从 5 增加到 10，就会在 `geo_pipeline/retries` 统计中得到重试的记录。如果发生任何错误，比如使用 API 无法找到一个位置，将会抛出异常，并且会在 `geo_pipeline/errors` 统计中被捕获到。如果某个位置的坐标已经被设置（后面的小节中看到），则会在 `geo_pipeline/already_set` 统计中显示。最后，当访问 `http://localhost:9200/properties/_property/_search`，查看房产信息的 ES 时，可以看到包含坐标位置值的条目，比如 `{... "location": { "lat": 51.5269736, "lon": -0.0667204 } ...}`，这和我们所期望的一样（在运行之前清理集合，确保看到的不是旧值）。

9.1.4 在 Elasticsearch 中启用地理编码索引

既然已经拥有了坐标位置，现在就可以做一些事情了，比如根据距离对结果进行排序。下面是一个 HTTP POST 请求（使用 `curl` 执行），返回标题中包含 "Angel" 的房产信息，并按照它们与点 `{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"
  }}]}'
```

唯一的问题是当尝试运行它时，会发现运行失败，并得到了一个错误信息：“failed to find mapper for [location] for geo distance based sort”。这说明位置字段并不是执行空间操作的适当格式。要想设置为合适的类型，则需要手动重写其默认类型。首先，将其自动检测的映射关系保存到文件中。

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

然后编辑 property.txt 的如下代码。

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

将该行的代码修改为如下代码。

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

另外，我们还删除了文件尾部的{"properties":{"mappings": and two }}。对该文件的修改到此为止。现在可以按如下代码删除旧类型，使用指定的模式创建新类型。

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

现在可以再次运行该爬虫，并且可以重新运行本节前面的 curl 命令，此时将会得到按照距离排序的结果。我们的搜索返回了房产信息的 JSON，额外包含了一个 sort 字段，该字段的值是到搜索点的距离，单位为千米。

9.2 与标准 Python 客户端建立数据库接口

有很多重要的数据库遵从 Python 数据库 API 规范 2.0 版本，包括 MySQL、PostgreSQL、Oracle、Microsoft SQL Server 和 SQLite。它们的驱动一般都比较复杂且久经考验，如果为 Twisted 重新实现的话则是巨大的浪费。人们可以在 Twisted 应用中使用这些数据库客户端，比如在 Scrapy 使用 twisted.enterprise.adbapi 库。我们将使用 MySQL 作为示例演示其使用，不过对于任何其他兼容的数据库来说，也可以应用

相同的原则。

9.2.1 用于写入 MySQL 的管道

MySQL 是一个非常强大且流行的数据库。我们将编写一个管道，将 item 写入到其中。我们已经在虚拟环境中运行了一个 MySQL 实例。现在只需使用 MySQL 命令行工具执行一些基本管理即可，同样该工具也已经在开发机中预安装好了，下面执行如下操作打开 MySQL 控制台。

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

这将会得到 MySQL 的提示符，即 `mysql>`，现在可以创建一个简单的数据库表，其中包含一些字段，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)
```

非常好，现在拥有了一个 MySQL 数据库，以及一张名为 `properties` 的表，其中包含了一些字段，此时可以准备创建管道了。请保持 MySQL 的控制台为开启状态，因为之后还会回来检查是否正确插入了值。如果想退出控制台，只需要输入 `exit` 即可。



在本节，我们将会向 MySQL 数据库中插入房产信息。如果你想擦除它们，可以使用如下命令：

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

我们将使用 Python 的 MySQL 客户端。我们还将安装一个名为 `dj-database-url` 的小工具模块，帮助我们解析连接的 URL（仅用于为我们在 IP、端口、密码等不同设置中切换节省时间）。可以使用 `pip install dj-database-url MySQL-python` 安

装这两个库，不过我们已经在开发环境中安装好它们了。我们的 MySQL 管道非常简单，如下所示。

```
from twisted.enterprise import adbapi
...
class MysqlWriter(object):
    ...
    def __init__(self, mysql_url):
        conn_kwargs = 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)
```



本示例的完整代码地址为 `ch09/properties/properties/pipeline/mysql.py`。

本质上，大部分代码仍然是模板化的爬虫代码。我们省略的代码用于将 `MYSQL_PIPELINE_URL` 设置中包含的 `mysql://user:pass@ip/database` 格式的 URL 解析为独立参数。在爬虫的 `__init__()` 中，我们将这些参数传给 `adbapi.ConnectionPool()`，使用 `adbapi` 的基础功能初始化 MySQL 连接池。第一个参数是想要导入的模块名称。在该 MySQL 示例中，为 `MySQLdb`。我们还为 MySQL 客户端设置了一些额外的参数，用于处理 `Unicode` 和超时。所有这些参数会在每次 `adbapi` 需要打开新连接时，前往底层的 `MySQLdb.connect()` 函数。当爬虫关闭时，我们为该连接池调用 `close()` 方法。

我们的 `process_item()` 方法实际上包装了 `dbpool.runInteraction()`。该方法将稍后调用的回调方法放入队列，当来自连接池的某个连接的 `Transaction` 对象变为可用时，调用该回调方法。`Transaction` 对象的 API 与 DB-API 游标相似。在本例中，回调方法为 `do_replace()`，该方法在后面几行进行了定义。`@staticmethod` 意味着该方法指向的是类，而不是具体的类实例，因此，可以省略平时使用的 `self` 参数。当不使用任何成员时，将方法静态化是个好习惯，不过即使忘记这么做，也没有问题。该方法准备了一个 SQL 字符串和几个参数，调用 `Transaction` 的 `execute()` 方法执行插入。我们的 SQL 语句使用了 `REPLACE INTO` 来替换已经存在的条目，而不是更常见的 `INSERT INTO`，原因是如果条目已经存在，可以使用相同的主键。在本例中这种方式非常便捷。如果想使用 SQL 返回数据，如 `SELECT` 语句，可以使用 `dbpool.runQuery()`。如果想要修改默认游标，可以通过设置 `adbapi.ConnectionPool()` 的 `cursorclass` 参数来实现，比如设置 `cursorclass=MySQLdb.cursors.DictCursor`，可以让数据获取更便捷。

要想使用该管道，需要在 `settings.py` 文件的 `ITEM_PIPELINES` 字典中添加它，另外还需要设置 `MYSQL_PIPELINE_URL` 属性。

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

执行如下命令。

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

该命令运行后，可以回到 MySQL 提示符下，按如下方式查看数据库中的记录。

```
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)
```

延时和吞吐量等性能和之前保持相同，相当不错。

9.3 使用 Twisted 专用客户端建立服务接口

到目前为止，我们看到了如何通过 `treq` 使用类 REST API。Scrapy 还可以和许多其他使用 Twisted 专用客户端的服务建立接口。比如，我们想要与 MongoDB 建立接口，当搜索 "MongoDB Python" 时，将会得到 `PyMongo`，该库是阻塞/同步的，不能和 Twisted 一起使用，除非使用后续小节中的方法，在管道中描述线程，处理阻塞操作。如果搜索 "MongoDB Twisted Python"，将会得到 `txmongo`，该库可以在 Twisted 和 Scrapy 中完美运行。通常情况下，Twisted 客户端背后的社区都很小，但相比自行编写客户端，这仍然是一个更好的选择。我们将使用一个类似的 Twisted 专用客户端作为接口，处理 Redis 键值对存储。

9.3.1 用于读写 Redis 的管道

Google Geocoding API 是按照 IP 进行限制的。我们可以利用多个 IP（例如使用多台服务器）进行缓解，此时需要避免重复请求其他机器上已经完成地理编码的地址。这种情况也适用于之前运行中曾见到过的地址。我们不想浪费宝贵的限额。



请与 API 供应商沟通，确保在他们的策略下这种做法是可行的。比如，你可能必须每隔几分钟/小时就要丢弃缓存记录，或者根本不允许缓存。

我们可以使用 Redis 的键值对缓存，从本质上说，它是一个分布式的字典。我们已经在 `vagrant` 环境中运行了一个 Redis 实例，可以使用 `redis-cli` 命令，从开发机连接它并执行基本操作。

```
$ 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 Google 搜索"Redis Twisted"，我们找到了 `txredisapi` 库。其本质区别是它不再是同步 Python 库的包装，而是适用于 Twisted 的库，它使用 `reactor.connectTCP()` 连接 Redis、实现 Twisted 协议等。使用该库的方式与其他库类似，不过在 Twisted 应用中使用它时，其效率肯定会更高一些。我们在安装它时可以再附带一个工具库——`dj_redis_url`，该工具库用于解析 Redis 配置 URL，我们可以使用 `pip` 进行安装(`sudo pip install txredisapi dj_redis_url`)，和往常一样，在我们的开发机中也已经预先安装好了这些库。

可以按如下代码初始化 `RedisCache`。

```

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)

```

该管道非常简单。为了连接 Redis 服务器，我们需要主机地址、端口等参数，由于这些参数是以 URL 格式存储的，因此需要使用 `parse_redis_url()` 方法解析该格式（为简洁起见已经省略）。为键设置前缀作为命名空间的行为非常常见，在本例中，我们将其存储在 `redis_nm` 中。然后，使用 `txredisapi` 的 `lazyConnectionPool()`，打开到服务器的连接。

最后一行使用了一个很有意思的函数。我们的目的是将地理编码管道与该管道包装起来。如果在 Redis 中没有某个值，我们将不会设置该值，我们的地理编码管道将像之前那样使用 API 对地址进行地理编码。在该操作完成之后，需要有一种方式在 Redis 中缓存这些键值对，在这里是通过连接到 `signals.item_scraped` 信号的方式实现的。我们定义的回调（`item_scraped()` 方法，将很快看到）在非常靠后的位置被调用，此时坐标位置将会被设置。



本示例的完整代码位于 `ch09/properties/properties/pipelines/redis.py`。

我们通过查找和记录每个 `Item` 的地址和位置，保持了缓存的简单性。这对 Redis 来说是很有意义的，因为它经常运行在同一个服务器当中，这使得它运行速度非常快。如果不是这种情况，那么可能需要添加一个基于字典的缓存，与我们在地理编码管道中的实现类似。下面是处理传入的 `Item` 的方法。

```
@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)
```

和大家的期望相同。我们得到了地址，为其添加前缀，然后使用 `txredisapi` `connection` 的 `get()` 方法在 Redis 中查询。我们在 Redis 中存储的值是 JSON 编码的对象。如果值已经设定，则使用 JSON 对其进行解码，并将其设为坐标位置。

当一个 Item 到达所有管道的结尾时，我们重新捕获它，确保存储到 Redis 的位置值当中。下面是实现代码。

```
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)
```

这里同样没有什么惊喜。如果我们找到一个位置，就可以得到地址，为其添加前缀，并使用它们作为键值对，用于 `txredisapi` 连接的 `set()` 方法。你会发现该函数没有使用 `@defer.inlineCallbacks`，这是因为在处理 `signals.item_scraped` 时并不支持该装饰器。这就意味着无法再对 `connection.set()` 使用非常便捷的 `yield` 操作，不过我们可以做的工作是返回延迟操作，**Scrapy** 可以用它串联任何未来的信号进行监听。无论何种情况，如果到 Redis 的连接无法执行 `connection.set()`，就会抛出一个异常。可以通过添加自定义错误处理到 `connection.set()` 返回的延迟操作中，静默忽略该异常。在该错误处理中，我们将失败作为参数传递，并告知它们对任何

ConnectionError 执行 `trap()` 操作。这是 Twisted 的延迟操作 API 的一个非常好用的功能。通过在预期的异常中使用 `trap()`，我们能够以紧凑的方式静默忽略它们。

为了启用该管道，我们所需做的就是将其添加到 `ITEM_PIPELINES` 设置中，并在 `settings.py` 文件中提供一个 `REDIS_PIPELINE_URL`。为该管道设置一个比地理编码管道更小的优先级值非常重要，否则其运行就会太迟，无法起到作用。

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

我们可以像平时那样运行该爬虫。第一次运行将会和之前类似，不过接下来的每次运行都会像下面这样。

```
$ 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,
```

可以看到 `GeoPipeline` 和 `RedisCache` 都已经启用，并且 `RedisCache` 会首先进行。另外，还可以注意到 `geo_pipeline/already_set` 统计值是 106。这些是 `GeoPipeline` 从 Redis 缓存中找到的预先填充好的 item，并且它们都不需要请求 Google API 调用。如果 Redis 缓存为空，你会看到一些键依然会使用 Google API 进行处理。在性能方面，我们注意到 `GeoPipeline` 引发的初始行为现在没有了。实际上，由于目前使用了缓存，因此绕过了每秒 5 个请求的 API 限制。当使用 Redis 时，还应当考虑使用过期键，使系统可以周期性地刷新缓存数据。

9.4 为 CPU 密集型、阻塞或遗留功能建立接口

本章最后一节讨论的是访问大多数非 Twisted 的工作。尽管有高效的异步代码所带来的巨大收益，但为 Twisted 和 Scrapy 重写每个库，既不现实也不可行。使用 Twisted 的线程池和 `reactor.spawnProcess()` 方法，我们可以使用任何 Python 库甚至其他语言编写的二进制包。

9.4.1 处理 CPU 密集型或阻塞操作的管道

第8章讲到，`reactor` 对于简短、非阻塞的任务非常理想。如果必须要执行一些更复杂或是涉及阻塞的事情，该怎么做呢？Twisted 提供了线程池，可以使用 `reactor.callInThread()` API 调用，在一些线程中执行慢操作，而不是在主线程中执行（Twisted 的 `reactor`）。这就意味着 `reactor` 会持续运行其处理过程，并在计算发生时响应事件。请注意，在线程池中的处理不是线程安全的。这就是说当你使用全局状态时，又会出现多线程编程中所有的传统同步问题。让我们从该管道的一个简单版本起步，逐渐编写出完整的代码。

```
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)
```

在前面的管道中，我们看到了实际运行的基本原语。对于每个 Item，我们抽取其

价格，并希望使用 `_do_calculation()` 方法处理它。该方法使用了一个阻塞操作 `time.sleep()`。我们将使用 `reactor.callInThread()` 调用把它放到另一个线程中运行。其中，被调用的函数以及传给该函数的任意数量的参数将会作为参数。显然，我们不只传递了 `price`，还创建并传递了一个名为 `out` 的延迟操作。当 `_do_calculation()` 完成计算时，我们将使用 `out` 回调返回值。在下一步中，我们对这个延迟操作执行了 `yield` 处理，并为价格设置了新值，最后返回 `Item`。

在 `_do_calculation()` 中，注意到有一个简单的计算——价格自增 1，然后是 100 毫秒的睡眠。这是非常多的时间，如果在 `reactor` 线程中调用，它将使我们每秒处理的页数无法超过 10 页。通过使其在其他线程中运行，就不再有这个问题了。任务将会在线程池中排队，等待出现可用的线程，一旦进入线程执行，该线程就将睡眠 100 毫秒。最后一步是触发 `out` 回调。正常情况下，可以使用 `out.callback(new_price)`，不过由于现在处于另一个线程中，这种方法不再安全。如果这样做，会导致延迟操作的代码和 `Scrapy` 的功能会从另一个线程调用，迟早会出现错误的结果。替代方案是使用 `reactor.callFromThread()`，同样，也是将函数作为参数，并将任意数量的额外参数传到函数中。该函数将会排队，由 `reactor` 线程调用；而另一方面，会解除 `process_item()` 对象 `yield` 操作的阻塞，为该 `Item` 恢复 `Scrapy` 操作。

如果有全局状态（比如计数器、移动平均值等）的话，那么在 `_do_calculation()` 中使用它们会发生什么呢？例如，我们添加两个变量——`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)...
```

上面的代码存在问题，我们会得到断言错误。这是因为如果一个线程在 `self.beta`

和 `self.delta` 之间切换，而另一个线程使用这些 `beta/delta` 的值恢复计算价格，那么会发现它们处于不一致的状态（`beta` 比 `delta` 大），因此，会计算出错误的结果。短暂的睡眠使该问题更容易产生，不过即便没有它，竞态条件也将很快出现。为了避免此类问题发生，必须使用锁，比如使用 Python 的 `threading.RLock()` 递归锁。当使用锁时，我们可以确信不会存在两个线程同时执行其保护的临界区的情况。

```
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 ...
```

前面的代码现在是正确的。请记住我们并不需要保护整段代码，只需覆盖全局状态的使用就够了。



本示例的完整代码位于 `ch09/properties/properties/pipelines/computation.py` 文件中。

要想使用该管道，只需在 `settings.py` 文件中将其添加到 `ITEM_PIPELINES` 设置即可，如下所示。

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

可以按照平时那样运行该爬虫。按照预期，管道延时显著增长了 100 毫秒，不过我们惊喜地发现吞吐量几乎保持不变，即每秒 25 个 `item` 左右。

9.4.2 使用二进制或脚本的管道

对于一个遗留功能来说，最不可知的接口就是独立的可执行程序或脚本。它可能需

要几秒钟时间启动（比如从数据库中加载数据），不过在这之后，它可能会在一小段延期内处理许多值。即使对于这种情况，Twisted 仍然能够覆盖。我们可以使用 `reactor.spawnProcess()` API 以及相关的 `protocol.ProcessProtocol` 运行任何类型的可执行程序。来看一个例子，该示例的脚本如下所示。

```
#!/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。当它启动后，会禁用 `Ctrl + C`。这是为了解决 `Ctrl + C` 派生到子进程后过早终止，导致 Scrapy 自身无法停止，无限等待子进程返回结果的系统特性。禁用 `Ctrl + C` 后，脚本将会睡眠 3 秒钟，以模拟启动时间。然后脚本会从输入中读取行，等待 250 毫秒，再返回结果价格，该计算使用 Linux 的 `awk` 命令将原值乘以 1.2 倍。该脚本的最大吞吐量是每秒 4 个 Item。可以使用一个简短的会话对其进行测试，如下所示。

```
$ 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
```

由于 `Ctrl + C` 被禁用，我们必须使用 `Ctrl + D` 终止会话。不错！那么，我们要如何在 Scrapy 中使用该脚本呢？仍然从一个简化的版本起步。

```
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)
```

我们可以在这里找到名为 `CommandSlot` 的 `ProcessProtocol` 的定义，以及 `Pricing` 爬虫。在 `__init__()` 中，我们创建了新的 `CommandSlot`，其构造方法初始化的一个空队列，并使用 `reactor.spawnProcess()` 启动了一个新的进程。该调用将从进程中传输和接收数据的 `ProcessProtocol` 作为第一个参数。在本例中，该值为 `self`，因为 `spawnProcess()` 是在 `protocol` 类中进行调用的。第二个参数是可执行程序名称。第三个参数 `args` 将该二进制程序的所有命令行参数作为字符串列表保留。

在管道的 `process_item()` 中，基本上将所有工作都委托给 `CommandSlot` 的 `legacy_calculate()` 方法，它将返回一个延迟操作，并执行 `yield` 操作。`legacy_calculate()` 创建了一个延迟操作，使其排队，然后使用 `transport.write()` 将价格写入到进程当中。`transport` 由 `ProcessProtocol` 提供，用于让我们和进程进行通信。无论我们何时从进程中接收到数据，都会调用 `outReceived()`。通过延迟操作排队，以及按顺序处理的 `shell` 脚本，我们可以从队列中只弹出最旧的延迟操作，使用接收到的值触发它。到此为止。我们可以通过在 `ITEM_PIPELINES` 中添加它的方式，启动该管道，并像平时那样运行。

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

如果我们运行一次，就会发现其性能非常糟糕。如我们所料，我们的处理成为瓶颈，限制了吞吐量只能达到每秒 4 个 Item。要想增长吞吐量，我们所能做的就是对管道进行一些修改，允许该类并行运行多个，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"][0] = yield
            slot.legacy_calculate(item["price"][0])
        defer.returnValue(item)
```

我们将其修改为启动 16 个实例，并以轮询的方式为每个实例发送价格。该管道现在提供了每秒 $16 \times 4 = 64$ 个 item 的吞吐量。我们可以通过一个快速爬取来确认，如下所示。

```
$ 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
```

延时和预期一样，增长到 250 毫秒，不过吞吐量仍然是每秒 25 个 item。

请注意，前面的方法中使用了 `transport.write()` 将 shell 脚本输入中的所有价格排入队列。对于你的应用而言，这种方式可能合适，也可能不合适，尤其是当它使用了更多的数据而不仅仅是几个数字时。本例完整代码会将所有值和回调排入队列，并且只有在前一次结果被接收后，才会向脚本发送新值。你会发现这种方式对你的遗留应用更加友好，不过也增添了一些复杂度。

9.5 本章小结

本章讲解了一些复杂的 Scrapy 管道。到目前为止，我们已经学习了 Twisted 编程方面所有可能需要的内容，并且知道了如何实现进程、使用 Item 进程管道等复杂功能。我们通过在延时和吞吐量方面添加更多管道阶段，看到了性能是如何变化的。通常情况下，延时和吞吐量被认为是成反比的，不过这是建立在常数并发的假设下的（例如线程的数例有限）。在我们的例子中，我们从 $N = S \cdot T = 25 \cdot 0.77 \cong 19$ 开始，在添加管道后，最终达到 $N = 25 \cdot 3.33 \cong 83$ ，并且没有任何性能问题。这就是 Twisted 编程的力量！现在我们可以进入第 10 章，使 Scrapy 的性能更加完美。

第 10 章

理解 Scrapy 性能

通常情况下，性能很容易出现问题。对于 Scrapy 来说，性能就不只是容易出现问题了，而是几乎肯定会出现，因为它有很多有悖常理的行为。除非你对 Scrapy 内部有非常好的理解，否则你会发现，即使非常努力地优化性能，也很可能得不到收益。这是使用高性能、低延迟以及高并发环境复杂性的一部分。在优化瓶颈性能时，阿姆达尔定律仍然是正确的，不过除非你能指明真正的瓶颈所在，否则在系统其他任何部分的优化都无法增长每秒能够抓取的 item 数量（吞吐量）。我们可以从 Goldratt 博士经典的 *The Goal* 一书中获得更多的感知，这本商务书籍通过优秀的隐喻对瓶颈、延迟和吞吐量的理念进行了阐释。相同的理念同样也适用于软件。本章将帮助你找出 Scrapy 配置中的瓶颈，以及避免出现明显的错误。

请注意本章是一个进阶章节，其中会涉及一些数学知识。计算将会比较简单，并且会附有用于展示相同概念的图表。如果你不喜欢数学，只需忽略掉公式即可，你仍然能够获得 Scrapy 性能如何工作的重要领悟。

10.1 Scrapy 引擎——一种直观方式

并行系统看起来与管道系统很相似。在计算机科学中，我们使用队列符号来表示队列以及处理中的元素（见图 10.1 左侧）。队列系统的基本法则是利特尔法则，该法则认为在稳定状态下，队列系统中的元素数量（N）等于系统吞吐量（T）乘以总排队/

服务时间 (S)，即 $N = T \cdot S$ 。另外两种形式是： $T = N / S$ 以及 $S = N / T$ ，在计算中同样有用。

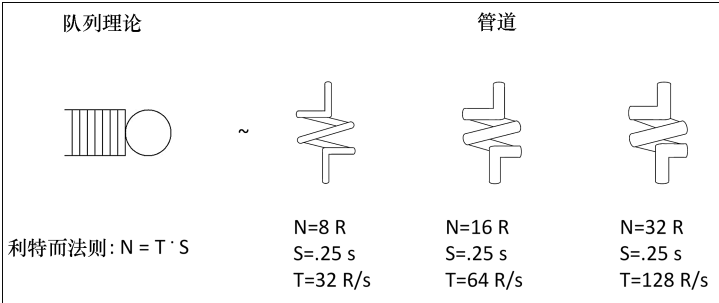


图 10.1 利特尔法则、队列系统以及管道

在管道的几何形状中也有相似的法则（见图 10.1 右侧）。管道容量 (V) 等于管道长度 L 乘以横截面面积 (A)，即 $V = L \cdot A$ 。

如果我们想象长度表示服务时间 ($L \sim S$)，容量表示处理系统的元素数量 ($V \sim N$)，横截面面积表示吞吐量 ($A \sim N$)，那么利特尔法则和容量公式实际是相同的事情。



这个类比有道理吗？答案是差不多。如果我们将工作单位想象为小滴液体，以恒定速率在管道内部移动，那么 $L \sim S$ 绝对有意义，因为管道越长，水滴移动花费的时间越多。 $V \sim N$ 同样有意义，因为管道越大，能够容纳的水滴越多。烦人的是，我们还可以通过施加更大压力的方式压入更多水滴。 $A \sim T$ 是不太满足类比的一点。在管道中，实际吞吐量，即每秒进出管道的水滴数量，被称为“体积流量”，除非满足特定条件（孔口），否则其与 A^2 成正比，而不是 A。这是因为更宽的管道不只意味着有更多的液体流出，还会使液体流动更快，因为管壁之间存在更大的空间。不过为了本章的学习，我们可以忽略这些技术细节，而是假设生活在一个理想的世界中，在这里压力和速度都是常量，并且吞吐量与横截面面积直接成正比。

利特尔法则和这个简单的体积公式非常相似，这就使得该“管道模型”非常直观有用。让我们更详细地看一下图 10.1 中的示例（右侧）。假设管道系统表示 Scrapy 的下载器。第一个非常“细”的下载器，其总体积/并发级别（ N ）可能是 8 个并发请求。管道长度/延迟（ S ）对于一个快速的网站来说，可能 $S=250\text{ms}$ 。在给定 N 和 S 时，现在可以计算处理元素的体积/吞吐量，每秒请求数为 $T = N / S = 8 / 0.25 = 32$ 。

你会发现延迟经常是我们无法控制的，因为它依赖于远端服务器的性能以及网络的延迟。我们比较容易控制的是下载器中并发（ N ）的级别，可以将其从 8 增长到 16 或 32 个并发请求，即 10.1 图中的第二个和第三个管道。对于常量的长度（超出我们控制范围之外），可以通过只增加横截面面积的方式增长体积，也就是说增加吞吐量！按照利特尔法则，16 个并发请求时，我们得到的每秒请求数为 $T = N / S = 16 / 0.25 = 64$ 个，而在 32 个并发请求时，我们得到的每秒请求数是 $T = N / S = 32 / 0.25 = 128$ 个。太好了！我们似乎可以通过增加并发的方式，使系统无限快。在急于得出这样的结论之前，还需要考虑队列系统级联的影响。

10.1.1 级联队列系统

当将不同横截面面积/吞吐量的几个管道依次连接起来时，可以很直观地理解整个系统的流量将由最窄的（最小吞吐量： T ）管道所限制（见图 10.2）。

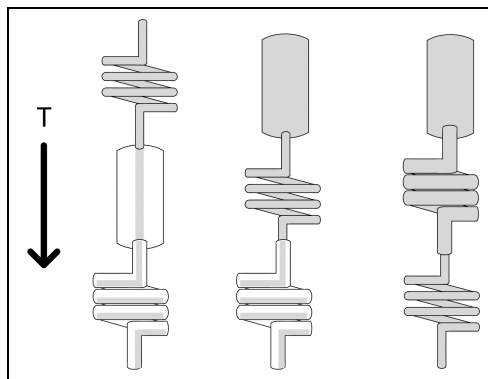


图 10.2 不同容量的级联队列系统

你还可以观察到最窄管道（即瓶颈）的位置，决定了其他管道是如何“填满”的。

如果考虑到与系统内存需求相关的填充，就会意识到瓶颈的位置是非常重要的。我们最好通过配置保持管道充满，且单个工作单元的花销最少。在 Scrapy 中，一个工作单元（爬取一个页面）主要是由下载器前的 URL（几个字节）以及下载后的 URL 加上服务器响应（较大）组成。



这就是为什么在 Scrapy 系统中，通常将瓶颈放置在下载器中。

10.1.2 定义瓶颈

使用管道系统作为类比的一个非常重要的好处是，它在定义瓶颈的过程中更加直观。如果观察图 10.2 就会发现，“瓶颈”前的所有地方都是满的，而之后的所有地方都不是。

好消息是，在大多数系统中，可以相对容易地使用系统度量监控队列系统是如何填满的。通过仔细检查 Scrapy 的队列，我们可以了解瓶颈在什么地方，如果发现不在下载器中，则可以调整设置让其变为下载器。没有改善瓶颈的任何改进都不会带来吞吐量的收益。如果修改系统其他部分，只会让事情变得更糟，很有可能将瓶颈转移到别的地方。这个感觉有点像追尾，可能需要很长时间，并且会令你感到绝望。你必须遵循系统方法，定义瓶颈，并且需要在修改任何代码或配置之前，“知道锤子应该击中哪里”。你在大部分例子中（包括本书的大多数例子）可以看到，瓶颈不是总在人们期望的地方出现。

10.1.3 Scrapy 性能模型

让我们回到 Scrapy，详细看一下其性能模型（见图 10.3）。

Scrapy 包含如下组成部分。

- **调度器**：在这里，多个请求会排队等待下载器处理。它们主要由 URL 组成，因此会十分紧凑，这就意味着即使拥有大量 URL 也不会对系统有很大伤害，并且可以让我们在传入不规则请求流的情况下能够充分利用下载器。
- **限流器**：这是抓取过程（大储水池）反馈的安全阀，如果正在执行的响应的总计

大小超过 5MB，那么它会让前往下载器的后续请求停止。这可能会导致不可预料的性能起伏。

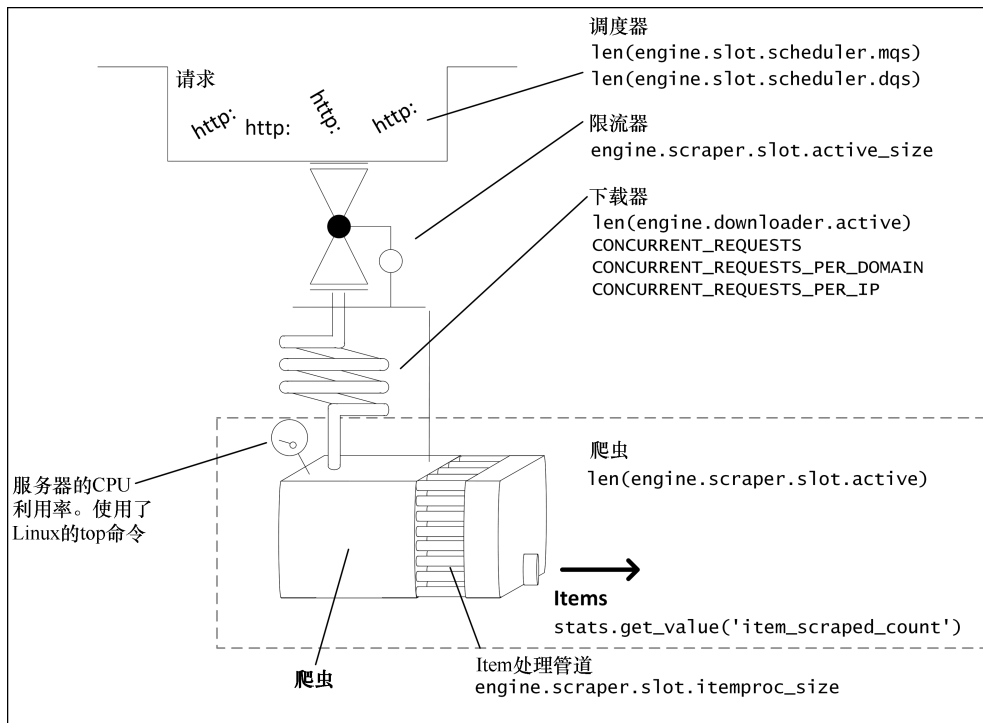


图 10.3 Scrapy 性能模型

- **下载器：**这是 Scrapy 关于性能最重要的组成部分。它对能够并行执行的请求的数量有着复杂的限制。其延迟（管道长度）等于远程服务器响应的时间，加上所有网络/操作系统以及 Python/Twisted 的延迟。我们可以调整并行请求的数量，不过通常情况下，我们几乎无法控制延迟。下载器的容量由 `CONCURRENT_REQUESTS*` 设置限制，我们将会很快看到。
- **爬虫：**这是抓取过程中将响应转为 Item 和后续请求的部分。同时这也是我们编写的部分，通常情况下，只要遵照规则，它们就不会是性能瓶颈。
- **Item 管道：**这是我们编写的代码的第二个部分。我们的爬虫可以对每个请求生成上百个 Item，同一时刻只会处理 `CONCURRENT_ITEMS` 个。该值十分重要，

因为假设你在管道中要处理数据库访问，那么使用默认值（100）就可能会过高，从而在无意间拖垮数据库。

爬虫和管道都应该使用异步代码，并且在必要时引发更多的延迟，但不应因此成为瓶颈。极少情况下，我们的爬虫/管道会处理非常繁重的事情。如果发生此种情况，那么服务器的 CPU 可能会成为瓶颈。

10.2 使用 telnet 获得组件利用率

想要理解 Request/Item 流是如何通过管道的，我们不会真得去测量流量（尽管这可能会是一个很棒的功能），而是使用更容易的方式测量 Scrapy 的每个处理阶段中存在多少流体，即 Request/Response/Item。

我们可以通过 Scrapy 运行的 Telnet 服务获取性能信息。首先，通过使用 telnet 命令连接到 6023 端口。然后，将会在 Scrapy 中得到一个 Python 提示符。需要小心的是，如果你在这里执行了某些阻塞操作，例如 `time.sleep()`，它将会中止爬虫功能。内置的 `est()` 函数可以打印出一些感兴趣的度量。其中一些或者很专用，或者能够从几个核心度量推断出来。在本章剩余部分只会展示后者。让我们从一个示例运行中了解它们。当运行爬虫时，可以在开发机中打开第二个终端，通过 telnet 命令连接 6023 端口，并运行 `est()`。



本章代码位于 ch10 目录，其中本例位于 ch10/speed 目录。

在第一个终端中，运行如下代码。

```
$ 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)
```

现在先不用管 scrapy crawl speed 是什么，以及其参数表示什么。本章后续部分会详细解释这些。现在，在第二个终端上，运行如下命令：

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

然后在第二个终端按下 *Ctrl + D* 退出 Telnet，回到第一个终端，按下 *Ctrl + C* 停止爬虫。

 我们在这里忽略了 dqs。如果通过 JOBDIR 设置启用了持久化支持的话，还会得到非零的 dqs (`len(engine.slot.scheduler.dqs)`)，你需要将其添加到 mqs 的大小中，以继续后续分析。

我们来看一下本例中的这些核心度量都表示什么。mqs 表示目前在调度器中还有很多等待（4475 个请求）。还可以。`len(engine.downloader.active)` 表示目前有 16 个请求正在下载器中被下载。这和我们在爬虫 `CONCURRENT_REQUESTS` 设置中设定的值相同，所以此处非常好。`len(engine.scrapers.slot.active)` 告知我们正在进行抓取处理的响应有 115 个。通过 `(engine.scrapers.slot.active_size)`，我们知道这些响应大小总计为 115kb。在这些响应中，有 105 个 Item 此时正在通过管道处理，可以从 `(engine.scrapers.slot.itemproc_size)` 看出来，这就意味着剩余的 10 个请求目前正在爬虫中处理。总体来说，我们可以看出瓶颈似乎在下载器中，在其之前的工作队列（mqs）非常庞大，但下载器已经满负荷利用了；而在其之后，我们有着数量很高但又比较稳定的任务（可以通过多次执行 `est()` 来确认此项）。

我们感兴趣的另一个信息元是 stats 对象，即通常在爬取完成后打印的信息。我们

可以在 Telnet 中, 通过 `stats.get_stats()`, 以字典的形式在任何时间访问它, 并且可以通过 `p()` 函数打印更优雅的模式。

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

对我们来说, 目前最感兴趣的度量是 `item_scraped_count`, 它可以通过 `stats.get_value('item_scraped_count')` 直接访问。该度量告知我们到目前为止有多少 item 已经被抓取, 它应当以系统吞吐量 (Item/秒) 的速率增长。

10.3 基准系统

为了第 10 章, 我编写了一个简单的基准系统, 可以让我们在不同场景下评估性能。该系统的代码比较复杂, 你可以在 `speed/spiders/speed.py` 中找到它, 但我不会详细讲解该代码。

该系统包含如下功能。

- 我们的 Web 服务器上 `http://localhost:9312/benchmark/...` 目录的处理器。可以通过调整 URL 参数/Scrapy 设置控制伪站点的结构 (见图 10.4) 以及页面加载速度。无需担心细节, 我们很快就会看到更多示例。现在, 可以观察 `http://localhost:9312/benchmark/index?p=1` 与 `http://localhost:9312/benchmark/ id:3/rr:5/index?p=1` 的区别。第一个页面加载时间在半秒之内, 并且每个详情页中有一个条目; 而第二个页面需要 5 秒时间加载, 但每个详情页中包含 3 个条目。我们还可以向页面中添加一些隐藏的垃圾数据, 使其更大一些。比如, `http://localhost:9312/ benchmark/ds:100/detail?id0=0`。默认情况下 (参见 `speed/ settings.py`), 页面渲染在 `SPEED_T_RESPONSE = 0.125` 秒内, 伪站点包含 `SPEED_TOTAL_ITEMS = 5000` 个 Item。

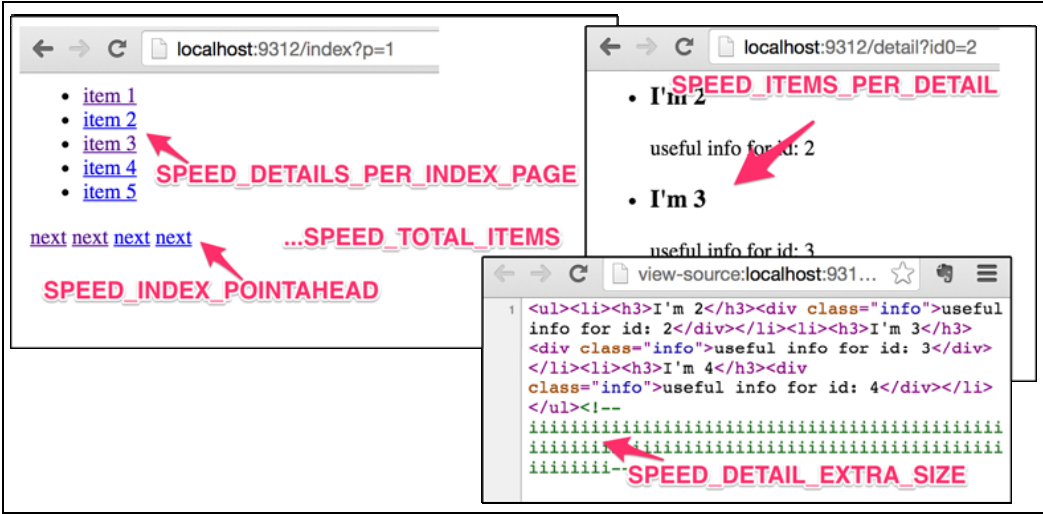


图 10.4 我们的基准系统创建的具有可调整结构的伪站点

- 爬虫 SpeedSpider, 通过控制 SPEED_START_REQUESTS_STYLE 设置伪造一些获取 start_requests() 的方式, 并提供了一个简单的 parse_item() 方法。默认情况下, 我们使用 crawler.engine.crawl() 方法直接将所有启动 URL 提供给 Scrapy 的调度器。
- 管道 DummyPipeline 伪造一些处理。它包含该处理可能导致的 4 种延迟类型: 阻塞/计算/同步延迟 (SPEED_PIPELINE_BLOCKING_DELAY, 这是一种不好的方式)、异步延迟 (SPEED_PIPELINE_ASYNC_DELAY, 这是一种可以接受的方式)、使用 treq 库的远程 API 调用 (SPEED_PIPELINE_API_VIA_TREQ, 这是一种可以接受的方式) 以及使用 Scrapy 的 crawler.engine.download() 的远程 API 调用 (SPEED_PIPELINE_API_VIA_DOWNLOADER, 这是一种不太好的方式)。默认情况下, 该管道不会添加任何延迟。
- 在 settings.py 中包含了一组高性能设置。所有可能会造成系统有任何减慢的设置都已经被禁用。由于我们只访问本地服务器, 因此针对单域名请求数的限制也被禁用了。
- 与第 8 章类似的少量度量捕获扩展。它将周期性地打印出核心度量指标。

我们已经在前面的例子中使用了该系统，不过让我们重新运行一次模拟，并使用 Linux 的时间工具测量完整的执行时间。可以在如下代码中看到被打印出来的核心度量指标。

```
$ time scrapy crawl speed
...
INFO: s/edule 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
s/edule len(engine.slot.scheduler.mqs)
d/load len(engine.downloader.active)
scrape len(engine.scrapers.slot.active)
p/line engine.scrapers.slot.itemproc_size
done stats.get_value('item_scraped_count')
mem engine.scrapers.slot.active_size
```

这种级别的透明度是非常明显的。我缩短了列名，不过它们应该仍然能够清楚说明含义。初始时，在调度器中有 5000 个 URL，而在结束时，完成列中也有 5000 个 item。下载器作为瓶颈，已经被充分利用，根据设置始终会有 16 个活跃的请求。抓取操作主要是爬虫，因为如我们在 p/line 列所见，管道是空的，由于它通常是在瓶颈之后，因此虽然一定程度上被利用了，但是没有充分利用。抓取 5000 个 Item 花费了 46 秒的时间，使用的并发请求 $N = 16$ ，即每个请求的平均时间是 $46 \cdot 16 / 5000 = 147\text{ms}$ ，而不是我们期望的 125ms，不过这也还可以接受。

10.4 标准性能模型

标准性能模型在 Scrapy 功能正常且下载器为性能瓶颈时成立。在这种情况下，可以

在调度器中看到一些请求，而在下载器中则是并发请求数的最大值（见图 10.5）。抓取程序（爬虫和管道）被轻度加载，并且处理中的响应数不会持续增长。

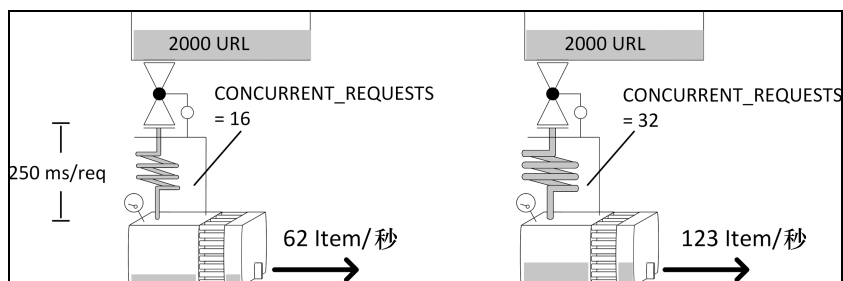


图 10.5 标准性能模型及一些实验结果

有 3 个主要设置用于控制下载器能力：CONCURRENT_REQUESTS、CONCURRENT_REQUESTS_PER_DOMAIN 以及 CONCURRENT_REQUESTS_PER_IP。其中第一个是粗调控制。无论如何都不会在同一时间有超过 CONCURRENT_REQUESTS 数量的请求处于活跃状态。而如果你的目标是单个域名或相对较少的几个域名，CONCURRENT_REQUESTS_PER_DOMAIN 可能会进一步限制活跃请求的数量。如果设置了 CONCURRENT_REQUESTS_PER_IP，那么 CONCURRENT_REQUESTS_PER_DOMAIN 就会被忽略，此时有效的限制将会是针对单个（目标）IP 的请求数。比如，当目标是一些共享主机站点时，多个域名可能会指向同一台服务器，该设置可以帮助你不会过度攻击该服务器。

为了保持现在的性能探索尽可能简单，我们通过使 CONCURRENT_REQUESTS_PER_IP 保留为默认值（0）以禁用每个 IP 的限制，并且设置 CONCURRENT_REQUESTS_PER_DOMAIN 的值为非常大的数值（1000000）。这样的组合可以有效禁用针对 IP 和域名的限制，下载器的并发数量可以完全由 CONCURRENT_REQUESTS 来控制。

我们希望系统吞吐量依赖于下载页面所花费的平均时间，包括远程服务器部分以及我们的系统（Linux、Twisted/Python）的延迟（ $t_{\text{download}} = t_{\text{response}} + t_{\text{overhead}}$ ）。如果能够考虑一些启动和结束时间也是很好的。它包括你得到一个响应的时间与其 Item 从管道另一端出来的时间之间的间隔，以及在缓存冷启动时，你得到第一个响应之前的时间及性能较差时的时间。

总之，如果你需要完成 N 个请求的任务，并且我们的爬虫已经得到了适当的调整，那么你应该会在下述公式所得的时间内完成。

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

我们无法控制这些参数中的大部分，这多少让人有些遗憾。我们可以使用一台更强的服务器来稍微控制 $t_{overhead}$ ，类似情况还有 t_{start}/t_{stop} （该参数几乎不值得为之努力，因为我们只会在每次运行时才会花费该时间）。除了对 N 个请求的给定工作量有少许改善外，我们所能细心调整的数值只有 `CONCURRENT_REQUESTS`，它通常依赖于我们访问远程服务器的困难程度。如果我们将它设定为一个非常大的数值，在某一时刻，会使服务器的 CPU 能力或远程服务器及时响应的能力达到饱和，也就是说， $t_{response}$ 将会突增，因为目标网站对我们实施了限速、封禁，或者我们造成了目标网站宕机。

让我们运行一个实验来检查我们的理论。我们将以 $t_{response} \in \{0.125s, 0.25s, 0.5s\}$ 、`CONCURRENT_REQUESTS` $\in \{8, 16, 32, 64\}$ 的条件爬取 2000 个 item，如下所示。

```
$ 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
```

在我的笔记本上，完成 2000 个请求的时间如表 10.1 所示（以秒为单位）。

表 10.1

CONCURRENT_REQUESTS	125ms/请求	250ms/请求	500ms/请求
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

警告：接下来将会是令人讨厌的计算！你可以略读本段内容。我们可以在图 10.5 中看到部分结果。通过重新排列最后的公式，我们可以将其转换为更加简单的形式（即 $y =$