Scrapy 轻松定制网络爬虫

by pluskid, on 2009-08-14

网络爬虫（Web Crawler, Spider）就是一个在网络上乱爬的机器人。当然它通常并不是一个实体的机器人，因为网络本身也是虚拟的东西，所以这个"机器人"其实也就是一段程序，并且它也不是**乱**爬，而是有一定目的的，并且在爬行的时候会搜集一些信息。例如 Google 就有一大堆爬虫会在 Internet 上搜集网页内容以及它们之间的链接等信息；又比如一些别有用心的爬虫会在 Internet 上搜集诸如 foo@bar.com 或者 foo [at] bar [dot] com 之类的东西。除此之外，还有一些定制的爬虫，专门针对某一个网站，例如前一阵子 JavaEye 的 Robbin 就写了几篇专门对付恶意爬虫的 blog（原文链接似乎已经失效了，就不给了），还有诸如小众软件或者 **LinuxToy** 这样的网站也经常被整个站点 crawl 下来，换个名字挂出来。其实爬虫从基本原理上来讲很简单，只要能访问网络和分析 Web 页面即可，现在大部分语言都有方便的 Http 客户端库可以抓取 Web 页面，而 HTML 的分析最简单的可以直接用正则表达式来做，因此要做一个最简陋的网络爬虫实际上是一件很简单的事情。不过要实现一个高质量的 spider 却是非常难的。

爬虫的两部分，一是下载 Web 页面，有许多问题需要考虑，如何最大程度地利用本地带宽，如何调度针对不同站点的 Web 请求以减轻对方服务器的负担等。一个高性能的 Web Crawler 系统里，DNS 查询也会成为急需优化的瓶颈，另外，还有一些"行规"需要遵循（例如 **robots.txt**）。而获取了网页之后的分析过程也是非常复杂的，Internet 上的东西千奇百怪，各种错误百出的 HTML 页面都有，要想全部分析清楚几乎是不可能的事；另外，随着 AJAX 的流行，如何获取由 Javascript 动态生成的内容成了一大难题；除此之外，Internet 上还有有各种有意或无意出现的 **Spider Trap**，如果盲目的跟踪超链接的话，就会陷入 Trap 中万劫不复了，例如这个网站，据说是之前 Google 宣称 Internet 上的 Unique URL 数目已经达到了 1 trillion 个，因此这个人 *is proud to announce the second trillion* 。

不过，其实并没有多少人需要做像 Google 那样通用的 Crawler，通常我们做一个 Crawler 就是为了去爬特定的某个或者某一类网站，所谓知己知彼，百战不殆，我们可以事先对需要爬的网站结构做一些分析，事情就变得容易多了。通过分析，选出有价值的链接进行跟踪，就可以避免很多不必要的链接或者 Spider Trap，如果网站的结构允许选择一个合适的路径的话，我们可以按照一定顺序把感兴趣的东西爬一遍，这样以来，连 URL 重复的判断也可以省去。

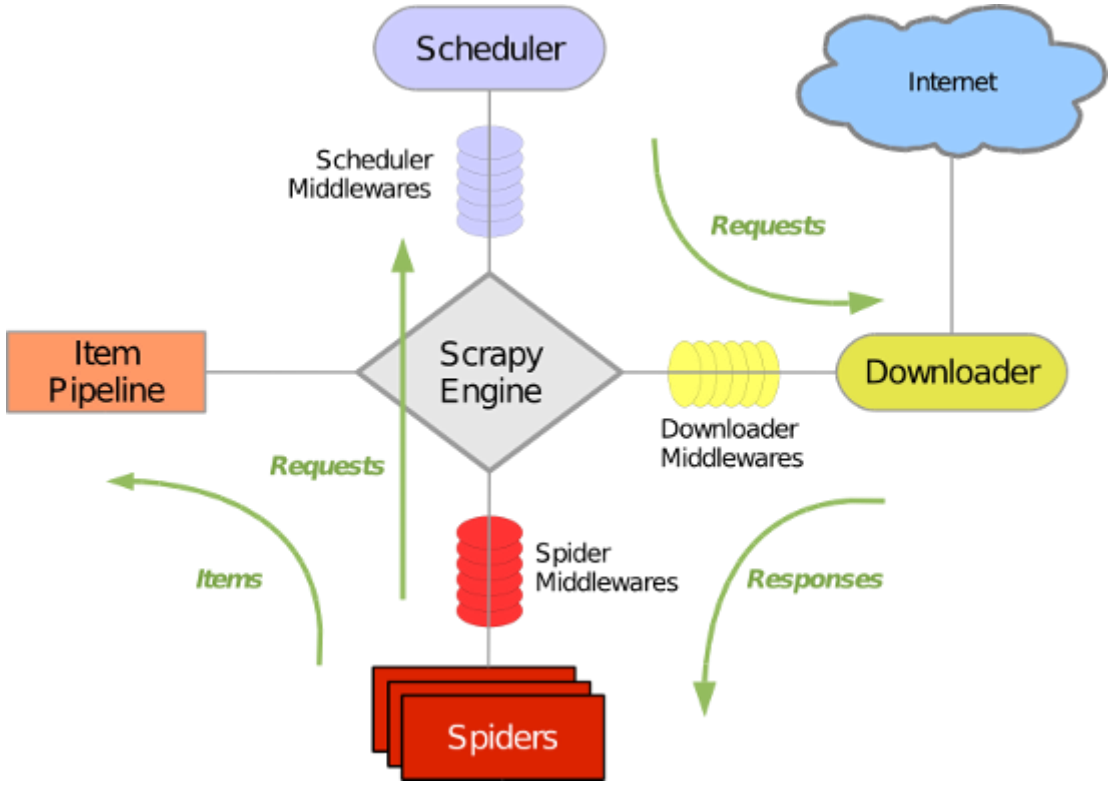
举个例子，假如我们想把 pongba 的 blog **mindhacks.cn** 里面的 blog 文字爬下来，通过观察，很容易发现我们对其中的两种页面感兴趣：

1. 文章列表页面，例如首页，或者 URL 是 `/page/\d+/` 这样的页面，通过 **Firebug** 可以看到到每篇文章的链接都是在一个 `h1` 下的 `a` 标签里的（需要注意的是，在 Firebug 的 HTML 面板里看到的 HTML 代码和 View Source 所看到的也许会有些出入，如果网页中有 Javascript 动态修改 DOM 树的话，前者是被修改过的版本，并且经过 Firebug 规则化的，例如 attribute 都有引号扩起来等，而后者通常才是你的 spider 爬到的原始内容。如果是使用正则表达式对页面进行分析或者所用的 HTML Parser 和 Firefox 的有些出入的话，需要特别注意），另外，在一个 class 为 `wp-pagenavi` 的 `div` 里有到不同列表页面的链接。
2. 文章内容页面，每篇 blog 有这样一个页面，例如 **/2008/09/11/machine-learning-and-ai-resources/**，包含了完整的文章内容，这是我们感兴趣的内容。

因此，我们从首页开始，通过 wp-pagenavi 里的链接来得到其他的文章列表页面，特别地，我们定义一个路径：只 follow *Next Page* 的链接，这样就可以从头到尾按顺序走一遍，免去了需要判断重复抓取的烦恼。另外，文章列表页面的那些到具体文章的链接所对应的页面就是我们真正要保存的数据页面了。

这样以来，其实用脚本语言写一个 ad hoc 的 Crawler 来完成这个任务也并不难，不过今天的主角是 **Scrapy**，这是一个用 Python 写的 Crawler Framework，简单轻巧，并且非常方便，并且官网上说已经在实际生产中在使用了，因此并不是一个玩具级别的东西。不过现在还没有 Release 版本，可以直接使用他们的 **Mercurial 仓库**里抓取源码进行安装。不过，这个东西也可以不安装直接使用，这样还方便随时更新，文档里说得很详细，我就不重复了。

Scrapy 使用 **Twisted** 这个异步网络库来处理网络通讯，架构清晰，并且包含了各种中间件接口，可以灵活的完成各种需求。整体架构如下图所示：



绿线是数据流向，首先从初始 URL 开始，Scheduler 会将其交给 Downloader 进行下载，下载之后会交给 Spider 进行分析，Spider 分析出来的结果有两种：一种是需要进一步抓取的链接，例如之前分析的"下一页"的链接，这些东西会被传回 Scheduler ；另一种是需要保存的数据，它们则被送到 Item Pipeline 那里，那是对数据进行后期处理（详细分析、过滤、存储等）的地方。另外，在数据流动的通道里还可以安装各种中间件，进行必要的处理。

看起来好像很复杂，其实用起来很简单，就如同 Rails 一样，首先新建一个工程：

```
scrapy-admin.py startproject blog_crawl
```

会创建一个 blog_crawl 目录，里面有个 scrapy-ctl.py 是整个项目的控制脚本，而代码全都放在子目录 blog_crawl 里面。为了能抓取 mindhacks.cn ，我们在 spiders 目录里新建一个 mindhacks_spider.py ，定义我们的 Spider 如下：

```
from scrapy.spider import BaseSpider

class MindhacksSpider(BaseSpider):
    domain_name = "mindhacks.cn"
    start_urls = ["http://mindhacks.cn/"]

    def parse(self, response):
        return []

SPIDER = MindhacksSpider()
```

我们的 MindhacksSpider 继承自 BaseSpider （通常直接继承自功能更丰富的 scrapy.contrib.spiders.CrawlSpider 要方便一些，不过为了展示数据是如何 parse 的，这里还是使用 BaseSpider 了），变量 domain_name 和 start_urls 都很容易明白是什么意思，而 parse 方法是我们需要定义的回调函数，默认的 request 得到 response 之后会调用这个回调函数，我们需要在这里对页面进行解析，返回两种结果（需要进一步 crawl 的链接和需要保存的数据），让我感觉有些奇怪的是，它的接口定义里这两种结果竟然是混杂在一个 list 里返回的，不太清楚这里为何这样设计，难道最后不还是要费力把它们分开？总之这里我们先写一个空函数，只返回一个空列表。另外，定义一个"全局"变量 SPIDER，它会在 Scrapy 导入这个 module 的时候实例化，并自动被 Scrapy 的引擎找到。这样就可以先运行一下 crawler 试试了：

```
./scrapy-ctl.py crawl mindhacks.cn
```

会有一堆输出，可以看到抓取了 http://mindhacks.cn，因为这是初始 URL，但是由于我们在 parse 函数里没有返回需要进一步抓取的 URL，因此整个 crawl 过程只抓取了主页便结束了。接下来便是要对页面进行分析，Scrapy 提供了一个很方便的 Shell （需要 IPython ）可以让我们做实验，用如下命令启动 Shell ：

```
./scrapy-ctl.py shell http://mindhacks.cn
```

它会启动 crawler ，把命令行指定的这个页面抓取下来，然后进入 shell ，根据提示，我们有许多现成的变量可以用，其中一个就是 hxs ，它是一个 HtmlXPathSelector ，mindhacks 的 HTML 页面比较规范，可以很方便的直接用 **XPath** 进行分析。通过 Firebug 可以看到，到每篇 blog 文章的链接都是在 h1 下的，因此在 Shell 中使用这样的 XPath 表达式测试：

```
In [1]: hxs.x('//h1/a/@href').extract()
Out[1]:
[u'http://mindhacks.cn/2009/07/06/why-you-should-do-it-yourself/',
 u'http://mindhacks.cn/2009/05/17/seven-years-in-nju/',
 u'http://mindhacks.cn/2009/03/28/effective-learning-and-memorization/',
 u'http://mindhacks.cn/2009/03/15/preconception-explained/',
 u'http://mindhacks.cn/2009/03/09/first-principles-of-programming/',
 u'http://mindhacks.cn/2009/02/15/why-you-should-start-blogging-now/',
 u'http://mindhacks.cn/2009/02/09/writing-is-better-thinking/',
 u'http://mindhacks.cn/2009/02/07/better-explained-conflicts-in-intimate-relationship/',
 u'http://mindhacks.cn/2009/02/07/independence-day/',
 u'http://mindhacks.cn/2009/01/18/escape-from-your-shawshank-part1/']
```

这正是我们需要的 URL ，另外，还可以找到“下一页”的链接所在，连同其他几个页面的链接一同在一个 div 里，不过“下一页”的链接没有 title 属性，因此 XPath 写作

```
//div[@class="wp-pagenavi"]/a[not(@title)]
```

不过如果向后翻一页的话，会发现其实“上一页”也是这样的，因此还需要判断该链接上的文字是那个下一页的箭头 u'\xbb' ，本来也可以写到 XPath 里面去，但是好像这个本身是 unicode escape 字符，由于编码原因理不清楚，直接放到外面判断了，最终 parse 函数如下：

```
def parse(self, response):
    items = []
    hxs = HtmlXPathSelector(response)
    posts = hxs.x('//h1/a/@href').extract()
    items.extend([self.make_requests_from_url(url).replace(callback=self.parse_post)
                  for url in posts])

    page_links = hxs.x('//div[@class="wp-pagenavi"]/a[not(@title)]')
    for link in page_links:
        if link.x('text()').extract()[0] == u'\xbb':
            url = link.x('@href').extract()[0]
            items.append(self.make_requests_from_url(url))

    return items
```

前半部分是解析需要抓取的 blog 正文的链接，后半部分则是给出“下一页”的链接。需要注意的是，这里返回的列表里并不是一个个的字符串格式的 URL 就完了，Scrapy 希望得到的是 Request 对象，这比一个字符串格式的 URL 能携带更多的东西，诸如 Cookie 或者回调函数之类的。可以看到我们在创建 blog 正文的 Request 的时候替换掉了回调函数，因为默认的这个回调函数 parse 是专门用来解析文章列表这样的页面的，
而 parse_post 定义如下：

```
def parse_post(self, response):
    item = BlogCrawlItem()
    item.url = unicode(response.url)
    item.raw = response.body_as_unicode()
    return [item]
```

很简单，返回一个 BlogCrawlItem ，把抓到的数据放在里面，本来可以在这里做一点解析，例如，通过 XPath 把正文和标题等解析出来，但是我倾向于后面再来做这些事情，例如 Item Pipeline 或者更后面的 Offline 阶段。BlogCrawlItem 是 Scrapy 自动帮我们定义好的一个继承自 ScrapedItem 的空类，在 items.py 中，这里我加了一点东西：

```
from scrapy.item import ScrapedItem

class BlogCrawlItem(ScrapedItem):
    def __init__(self):
        ScrapedItem.__init__(self)
```

```
        self.url = ''

    def __str__(self):
        return 'BlogCrawlItem(url: %s)' % self.url
```

定义了 __str__ 函数，只给出 URL ，因为默认的 __str__ 函数会把所有的数据都显示出来，因此会看到 crawl 的时候控制台 log 狂输出东西，那是把抓取到的网页内容输出出来了。-.-bb

这样一来，数据就取到了，最后只剩下存储数据的功能，我们通过添加一个 Pipeline 来实现，由于 Python 在标准库里自带了 Sqlite3 的支持，所以我使用 Sqlite 数据库来存储数据。用如下代码替换 pipelines.py 的内容：

```
1  import sqlite3
2  from os import path
3
4  from scrapy.core import signals
5  from scrapy.xlib.pydispatch import dispatcher
6
7  class SQLiteStorePipeline(object):
8      filename = 'data.sqlite'
9
10     def __init__(self):
11         self.conn = None
12         dispatcher.connect(self.initialize, signals.engine_started)
13         dispatcher.connect(self.finalize, signals.engine_stopped)
14
15     def process_item(self, domain, item):
16         self.conn.execute('insert into blog values(?,?,?)',
17                           (item.url, item.raw, unicode(domain)))
18         return item
19
20     def initialize(self):
21         if path.exists(self.filename):
22             self.conn = sqlite3.connect(self.filename)
23         else:
24             self.conn = self.create_table(self.filename)
25
26     def finalize(self):
27         if self.conn is not None:
28             self.conn.commit()
29             self.conn.close()
30             self.conn = None
31
32     def create_table(self, filename):
33         conn = sqlite3.connect(filename)
```

```
34          conn.execute("""create table blog
35                      (url text primary key, raw text, domain text)""")
36          conn.commit()
37          return conn
```

在 `__init__` 函数中，使用 dispatcher 将两个信号连接到指定的函数上，分别用于初始化和关闭数据库连接（在 `close` 之前记得 `commit`，似乎是不会自动 `commit` 的，直接 `close` 的话好像所有的数据都丢失了 dd-.-）。当有数据经过 pipeline 的时候，`process_item` 函数会被调用，在这里我们直接讲原始数据存储到数据库中，不作任何处理。如果需要的话，可以添加额外的 pipeline ，对数据进行提取、过滤等，这里就不细说了。

最后，在 `settings.py` 里列出我们的 pipeline ：

```
ITEM_PIPELINES = ['blog_crawl.pipelines.SQLiteStorePipeline']
```

再跑一下 crawler ，就 OK 啦！ 最后，总结一下：一个高质量的 crawler 是极其复杂的工程，但是如果有好的工具的话，做一个专用的 crawler 还是比较容易的。Scrapy 是一个很轻便的爬虫框架，极大地简化了 crawler 开发的过程。另外，Scrapy 的文档也是十分详细的，如果觉得我的介绍省略了一些东西不太清楚的话，推荐看他的 **Tutorial** 。