# Scrapy中文指南

极客学院出版

# 前言

Scrapy 是一个为了爬取网站数据，提取结构性数据而编写的应用框架。 可以应用在包括数据挖掘，信息处理或存储历史数据等一系列的程序中。本指南是 Scrapy 目前最新的版本，内容涉及安装，使用，开发，API 调试等全部知识点，帮助读者学习使用 Scrapy 框架开发网络爬虫。

## 适用人群

本指南适用于网络爬虫初学者学习，能够通过本指南了解到爬虫框架的原理和实现过程。

## 学习前提

学习本教程前，你需要了解 Python 这门编程语言。

## 版本信息

书中演示代码基于以下版本：

| 语言/框架 | 版本信息 |
| --- | --- |
| Python | Python2.7 |

鸣谢：http://scrapy-chs.readthedocs.org/zh_CN/latest/intro/overview.html

# 目录

# 1

# 初窥 Scrapy

Scrapy 是一个为了爬取网站数据，提取结构性数据而编写的应用框架。 可以应用在包括数据挖掘，信息处理或存储历史数据等一系列的程序中。

其最初是为了页面抓取 (更确切来说，网络抓取)所设计的， 也可以应用在获取 API 所返回的数据(例如 Amazon Associates Web Services ) 或者通用的网络爬虫。

本文档将通过介绍 Scrapy 背后的概念使您对其工作原理有所了解， 并确定Scrapy是否是您所需要的。

当您准备好开始您的项目后，您可以参考入门教程。

## 选择一个网站

当您需要从某个网站中获取信息，但该网站未提供API或能通过程序获取信息的机制时，Scrapy 可以助你一臂之力。

以 Mininova 网站为例，我们想要获取今日添加的所有种子的 URL、名字、描述以及文件大小信息。

今日添加的种子列表可以通过这个页面找到:
http://www.mininova.org/today

## 定义您想抓取的数据

第一步是定义我们需要爬取的数据。在 Scrapy 中， 这是通过 Scrapy Items 来完成的。(在本例子中为种子文件)

我们定义的 Item:

```
import scrapy

class TorrentItem(scrapy.Item):
    url = scrapy.Field()
    name = scrapy.Field()
    description = scrapy.Field()
    size = scrapy.Field()
```

## 编写提取数据的 Spider

第二步是编写一个 spider。其定义了初始 URL(http://www.mininova.org/today)、针对后续链接的规则以及从页面中提取数据的规则。

通过观察页面的内容可以发现，所有种子的 URL 都类似 `http://www.mininova.org/tor/NUMBER` 。其中， `NUMBER` 是一个整数。 根据此规律，我们可以定义需要进行跟进的链接的正则表达式: `/tor/\d+` 。

我们使用 XPath 来从页面的HTML源码中选择需要提取的数据。 以其中一个种子文件的页面为例:

http://www.mininova.org/tor/2676093

观察 HTML 页面源码并创建我们需要的数据(种子名字，描述和大小)的 XPath 表达式。

通过观察，我们可以发现文件名是包含在 `<h1>` 标签中的:

```
<h1>Darwin – The Evolution Of An Exhibition</h1>
```

与此对应的 XPath 表达式:

```
//h1/text()
```

种子的描述是被包含在 `id="description"` 的 `<div>` 标签中:

```
<h2>Description:</h2>

<div id="description">
Short documentary made for Plymouth City Museum and Art Gallery regarding the setup of an exhibit about Charles Darw


...
```

对应获取描述的 XPath 表达式:

```
//div[@id='description']
```

文件大小的信息包含在 `id=specifications` 的 `<div>` 的第二个 `<p>` 标签中:

```
<div id="specifications">

<p>
<strong>Category:</strong>
<a href="/cat/4">Movies</a> &gt; <a href="/sub/35">Documentary</a>
</p>
```

```
<p>
<strong>Total size:</strong>
150.62 megabyte</p>
```

选择文件大小的 XPath 表达式:

```
//div[@id='specifications']/p[2]/text()[2]
```

关于 XPath 的详细内容请参考 XPath 参考 。

最后，结合以上内容给出 spider 的代码:

```
from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors import LinkExtractor

class MininovaSpider(CrawlSpider):

    name = 'mininova'
    allowed_domains = ['mininova.org']
    start_urls = ['http://www.mininova.org/today']
    rules = [Rule(LinkExtractor(allow=['/tor/\d+']), 'parse_torrent')]

    def parse_torrent(self, response):
        torrent = TorrentItem()
        torrent['url'] = response.url
        torrent['name'] = response.xpath("//h1/text()").extract()
        torrent['description'] = response.xpath("//div[@id='description']").extract()
        torrent['size'] = response.xpath("//div[@id='info-left']/p[2]/text()[2]").extract()
        return torrent
```

`TorrentItem` 的定义在 上面 。

## 执行 spider，获取数据

终于，我们可以运行 spider 来获取网站的数据，并以 JSON 格式存入到 scraped_data.json 文件中:

```
scrapy crawl mininova -o scraped_data.json
```

命令中使用了 Feed 导出 来导出 JSON 文件。您可以修改导出格式(XML 或者 CSV)或者存储后端(FTP 或者 Amazon S3)，这并不困难。

同时，您也可以编写 item 管道 将 item 存储到数据库中。

## 查看提取到的数据

执行结束后，当您查看 `scraped_data.json` ，您将看到提取到的 item:

```
[{"url": "http://www.mininova.org/tor/2676093", "name": ["Darwin - The Evolution Of An Exhibition"], "description": ["Short do
 # ... other items ...

]
```

由于 选择器(Selectors) 发挥作用的地方。

## 还有什么?

您已经了解了如何通过 Scrapy 提取存储网页中的信息，但这仅仅只是冰山一角。Scrapy 提供了很多强大的特性来使得爬取更为简单高效，例如:

- HTML，XML 源数据 选择及提取 的内置支持
- 提供了一系列在spider之间共享的可复用的过滤器(即 Item Loaders)，对智能处理爬取数据提供了内置支持。
- 通过 feed 导出 提供了多格式(JSON、CSV、XML)，多存储后端(FTP、S3、本地文件系统)的内置支持
- 提供了 media pipeline，可以 自动下载 爬取到的数据中的图片(或者其他资源)。
- 高扩展性。您可以通过使用 signals，设计好的API(中间件，extensions，pipelines)来定制实现您的功能。
- 内置的中间件及扩展为下列功能提供了支持:
    - cookies and session 处理
    - HTTP 压缩
    - HTTP 认证
    - HTTP 缓存
    - user-agent 模拟
    - robots.txt
    - 爬取深度限制
    - 其他
- 针对非英语语系中不标准或者错误的编码声明, 提供了自动检测以及健壮的编码支持。
- 支持根据模板生成爬虫。在加速爬虫创建的同时，保持在大型项目中的代码更为一致。详细内容请参阅 genspider 命令。
- 针对多爬虫下性能评估、失败检测，提供了可扩展的状态收集工具。
- 提供交互式 shell 终端，为您测试 XPath 表达式，编写和调试爬虫提供了极大的方便
- 提供 System service，简化在生产环境的部署及运行
- 内置 Telnet 终端 ，通过在 Scrapy 进程中钩入 Python 终端，使您可以查看并且调试爬虫
- Logging 为您在爬取过程中捕捉错误提供了方便

- 支持 Sitemaps 爬取
- 具有缓存的 DNS 解析器

## 接下来

下一步当然是 [下载 Scrapyhref="http://scrapy.org/download/) 了， 您可以阅读Scrapy 入门教程并加入社区。感谢您的支持!

# 2

# 安装指南

# 安装 Scrapy

> 注解
>
> 请先阅读平台安装指南。

下列的安装步骤假定您已经安装好下列程序:

- Python2.7
- PythonPackage:pipandsetuptools。现在 pip 依赖 setuptools，如果未安装，则会自动安装setuptools。
- lxml。大多数 Linux 发行版自带了 lxml。如果缺失，请查看 `http://lxml.de/installation.html`
- OpenSSL。除了 Windows(请查看平台安装指南)之外的系统都已经提供。

您可以使用 pip 来安装 Scrapy(推荐使用 pip 来安装 Pythonpackage)。

使用 pip 安装:

```
pip install Scrapy
```

# 平台安装指南

## Windows

- 从 http://python.org/download/上安装 Python2.7。

您需要修改 PATH 环境变量，将 Python 的可执行程序及额外的脚本添加到系统路径中。将以下路径添加到 PATH 中:

```
C:\Python27\;C:\Python27\Scripts\;
```

请打开命令行，并且运行以下命令来修改 PATH:

```
c:\python27\python.exec:\python27\tools\scripts\win_add2path.py
```

关闭并重新打开命令行窗口，使之生效。运行接下来的命令来确认其输出所期望的 Python 版本:

```
python--version
```

- 从 http://sourceforge.net/projects/pywin32/安装 pywin32

请确认下载符合您系统的版本(win32 或者 amd64)

- 从 https://pip.pypa.io/en/latest/installing.html 安装 pip

打开命令行窗口，确认 `pip` 被正确安装:

```
pip--version
```

- 到目前为止 Python2.7 及 `pip` 已经可以正确运行了。接下来安装 Scrapy:

```
pip install Scrapy
```

## Ubuntu9.10 及以上版本

不要使用 Ubuntu 提供的 python-scrapy，相较于最新版的 Scrapy，该包版本太旧，并且运行速度也较为缓慢。

您可以使用官方提供的 Ubuntu Packages。该包解决了全部依赖问题，并且与最新的 bug 修复保持持续更新。

## Archlinux

您可以依照通用的方式或者从 AUR Scrapy package来安装 Scrapy:

```
yaourt-Sscrapy
```

# 3

# Scrapy 入门教程

在本篇教程中，我们假定您已经安装好 Scrapy。 如若不然，请参考 安装指南 。

接下来以 Open Directory Project(dmoz) (dmoz) 为例来讲述爬取。

本篇教程中将带您完成下列任务:

1. 创建一个 Scrapy 项目
2. 定义提取的 Item
3. 编写爬取网站的 spider 并提取 Item
4. 编写 Item Pipeline 来存储提取到的 Item(即数据)

Scrapy 由 Python 编写。如果您刚接触并且好奇这门语言的特性以及 Scrapy 的详情，对于已经熟悉其他语言并且想快速学习 Python 的编程老手，我们推荐 Learn Python The Hard Way，对于想从 Python 开始学习的编程新手，非程序员的 Python 学习资料列表将是您的选择。

## 创建项目

在开始爬取之前，您必须创建一个新的 Scrapy 项目。 进入您打算存储代码的目录中，运行下列命令:

```
scrapy startproject tutorial
```

该命令将会创建包含下列内容的 tutorial 目录:

```
tutorial/
    scrapy.cfg
    tutorial/
        __init__.py
        items.py
        pipelines.py
        settings.py
        spiders/
            __init__.py
            ...
```

这些文件分别是:

- scrapy.cfg: 项目的配置文件
- tutorial/: 该项目的 python 模块。之后您将在此加入代码。
- tutorial/items.py: 项目中的 item 文件。
- tutorial/pipelines.py: 项目中的 pipelines 文件。
- tutorial/settings.py: 项目的设置文件。
- tutorial/spiders/: 放置 spider 代码的目录。

# 定义 Item

Item 是保存爬取到的数据的容器；其使用方法和 python 字典类似， 并且提供了额外保护机制来避免拼写错误导致的未定义字段错误。

类似在 ORM 中做的一样，您可以通过创建一个 `scrapy.Item` 类， 并且定义类型为 `scrapy.Field` 的类属性来定义一个 Item。（如果不了解 ORM, 不用担心，您会发现这个步骤非常简单）

首先根据需要从 dmoz.org 获取到的数据对 item 进行建模。 我们需要从 dmoz 中获取名字，url，以及网站的描述。 对此，在 item 中定义相应的字段。编辑 `tutorial` 目录中的 `items.py` 文件:

```
import scrapy

class DmozItem(scrapy.Item):
    title = scrapy.Field()
    link = scrapy.Field()
    desc = scrapy.Field()
```

一开始这看起来可能有点复杂，但是通过定义 item， 您可以很方便的使用 Scrapy 的其他方法。而这些方法需要知道您的 item 的定义。

# 编写第一个爬虫(Spider)

Spider 是用户编写用于从单个网站(或者一些网站)爬取数据的类。

其包含了一个用于下载的初始 URL，如何跟进网页中的链接以及如何分析页面中的内容， 提取生成 `item` 的方法。

为了创建一个 Spider，您必须继承 `scrapy.Spider` 类， 且定义以下三个属性:

- `name` : 用于区别 Spider。 该名字必须是唯一的，您不可以为不同的 Spider 设定相同的名字。
- `start_urls` : 包含了 Spider 在启动时进行爬取的 url 列表。 因此，第一个被获取到的页面将是其中之一。 后续的 URL 则从初始的 URL 获取到的数据中提取。
- `parse()` 是 spider 的一个方法。 被调用时，每个初始 URL 完成下载后生成的 `Response` 对象将会作为唯一的参数传递给该函数。 该方法负责解析返回的数据(response data)，提取数据(生成 item)以及生成需要进一步处理的 URL 的 `Request 对象。

以下为我们的第一个 Spider 代码，保存在 tutorial/spiders 目录下的 dmoz_spider.py 文件中:

```
import scrapy

class DmozSpider(scrapy.spiders.Spider):
    name = "dmoz"
    allowed_domains = ["dmoz.org"]
    start_urls = [
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/"
    ]

    def parse(self, response):
        filename = response.url.split("/")[-2]
        with open(filename, 'wb') as f:
            f.write(response.body)
```

## 爬取

进入项目的根目录，执行下列命令启动 spider:

```
scrapy crawl dmoz
```

`crawl dmoz` 启动用于爬取 `dmoz.org` 的 spider，您将得到类似的输出:

```
2014-01-23 18:13:07-0400 [scrapy] INFO: Scrapy started (bot: tutorial)
2014-01-23 18:13:07-0400 [scrapy] INFO: Optional features available: ...
2014-01-23 18:13:07-0400 [scrapy] INFO: Overridden settings: {}
2014-01-23 18:13:07-0400 [scrapy] INFO: Enabled extensions: ...
2014-01-23 18:13:07-0400 [scrapy] INFO: Enabled downloader middlewares: ...
2014-01-23 18:13:07-0400 [scrapy] INFO: Enabled spider middlewares: ...
2014-01-23 18:13:07-0400 [scrapy] INFO: Enabled item pipelines: ...
2014-01-23 18:13:07-0400 [dmoz] INFO: Spider opened
2014-01-23 18:13:08-0400 [dmoz] DEBUG: Crawled (200) <GET http://www.dmoz.org/Computers/Programming/Langua
2014-01-23 18:13:09-0400 [dmoz] DEBUG: Crawled (200) <GET http://www.dmoz.org/Computers/Programming/Langua
2014-01-23 18:13:09-0400 [dmoz] INFO: Closing spider (finished)
```

查看包含 `[dmoz]` 的输出，可以看到输出的 log 中包含定义在 `start_urls` 的初始 URL，并且与 spider 中是一一对应的。在 log 中可以看到其没有指向其他页面( ( `referer:None` ) )。

除此之外，更有趣的事情发生了。就像我们 parse 方法指定的那样，有两个包含 url 所对应的内容的文件被创建了: Book，Resources 。

## 刚才发生了什么?

Scrapy 为 Spider 的 start_urls 属性中的每个 URL 创建了 scrapy.Request 对象，并将 parse 方法作为回调函数(callback)赋值给了 Request。

Request 对象经过调度，执行生成 `scrapy.http.Response` 对象并送回给 `spider parse()` 方法。

## 提取 Item

### Selectors 选择器简介

从网页中提取数据有很多方法。Scrapy 使用了一种基于 XPath 和 CSS 表达式机制: `Scrapy Selectors` 。关于 selector 和其他提取机制的信息请参考 `Selector` 文档。

这里给出 XPath 表达式的例子及对应的含义:

- `/html/head/title` : 选择 HTML 文档中 `<head>` 标签内的 `<title>` 元素
- `/html/head/title/text()` : 选择上面提到的 `<title>` 元素的文字
- `//td` : 选择所有的 `<td>` 元素
- `//div[@class="mine"]` : 选择所有具有 class="mine"属性的 div 元素

上边仅仅是几个简单的 XPath 例子，XPath 实际上要比这远远强大的多。 如果您想了解的更多，我们推荐这篇 XPath 教程。

为了配合 XPath，Scrapy 除了提供了 Selector 之外，还提供了方法来避免每次从 response 中提取数据时生成 selector 的麻烦。

Selector 有四个基本的方法(点击相应的方法可以看到详细的 API 文档):

- xpath() : 传入 xpath 表达式，返回该表达式所对应的所有节点的 selector list 列表 。
- css() : 传入 CSS 表达式，返回该表达式所对应的所有节点的 selector list 列表.
- extract() : 序列化该节点为 unicode 字符串并返回 list。
- re() : 根据传入的正则表达式对数据进行提取，返回 unicode 字符串 list 列表。

### 在 Shell 中尝试 Selector 选择器

为了介绍 Selector 的使用方法，接下来我们将要使用内置的 Scrapy shell。Scrapy Shell 需要您预装好 IPython(一个扩展的 Python 终端)。

您需要进入项目的根目录，执行下列命令来启动 shell:

```
scrapy shell "http://www.dmoz.org/Computers/Programming/Languages/Python/Books/"
```

> 注解
>
> 当您在终端运行 Scrapy 时，请一定记得给 url 地址加上引号，否则包含参数的 url(例如 & 字符)会导致 Scrapy 运行失败。

shell 的输出类似:

```
[ ... Scrapy log here ... ]

2015-01-07 22:01:53+0800 [domz] DEBUG: Crawled (200) <GET http://www.dmoz.org/Computers/Programming/Langua
[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0x02CE2530>
[s]   item       {}
[s]   request    <GET http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
[s]   response   <200 http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
[s]   sel        <Selector xpath=None data=u'<html lang="en">\r\n<head>\r\n<meta http-equ'>
[s]   settings   <CrawlerSettings module=<module 'tutorial.settings' from 'tutorial\settings.pyc'>>
[s]   spider     <DomzSpider 'domz' at 0x302e350>
[s] Useful shortcuts:
[s]   shelp()           Shell help (print this help)
```

```
[s]   fetch(req_or_url) Fetch request (or URL) and update local objects
[s]   view(response)    View response in a browser

>>>
```

当 shell 载入后，您将得到一个包含 response 数据的本地 response 变量。输入 response.body 将输出 response 的包体，输出 response.headers 可以看到 response 的包头。

更为重要的是，当输入 response.selector 时， 您将获取到一个可以用于查询返回数据的 selector(选择器)，以及映射到 response.selector.xpath() 、response.selector.css() 的 快捷方法(shortcut): response.xpath() 和 response.css() 。

同时，shell 根据 response 提前初始化了变量 sel 。该 selector 根据 response 的类型自动选择最合适的分析规则(XML vs HTML)。

让我们来试试:

```
In [1]: sel.xpath('//title')
Out[1]: [<Selector xpath='//title' data=u'<title>Open Directory - Computers: Progr'>]

In [2]: sel.xpath('//title').extract()
Out[2]: [u'<title>Open Directory - Computers: Programming: Languages: Python: Books</title>']

In [3]: sel.xpath('//title/text()')
Out[3]: [<Selector xpath='//title/text()' data=u'Open Directory - Computers: Programming:'>]

In [4]: sel.xpath('//title/text()').extract()
Out[4]: [u'Open Directory - Computers: Programming: Languages: Python: Books']

In [5]: sel.xpath('//title/text()').re('(\w+):')
Out[5]: [u'Computers', u'Programming', u'Languages', u'Python']
```

**提取数据**

现在，我们来尝试从这些页面中提取些有用的数据。

您可以在终端中输入 `response.body` 来观察 HTML 源码并确定合适的 XPath 表达式。不过，这任务非常无聊且不易。您可以考虑使用 Firefox 的 Firebug 扩展来使得工作更为轻松。详情请参考 使用 Firebug 进行爬取和借助 Firefox 来爬取。

在查看了网页的源码后，您会发现网站的信息是被包含在 第二个 <ul> 元素中。

我们可以通过这段代码选择该页面中网站列表里所有 <li> 元素:

```
sel.xpath('//ul/li')
```

网站的描述:

```
sel.xpath('//ul/li/text()').extract()
```

网站的标题:

```
sel.xpath('//ul/li/a/text()').extract()
```

以及网站的链接:

```
sel.xpath('//ul/li/a/@href').extract()
```

之前提到过，每个 `.xpath()` 调用返回 selector 组成的 list，因此我们可以拼接更多的 `.xpath()` 来进一步获取某个节点。我们将在下边使用这样的特性:

```
for sel in response.xpath('//ul/li'):
    title = sel.xpath('a/text()').extract()
    link = sel.xpath('a/@href').extract()
    desc = sel.xpath('text()').extract()
    print title, link, desc
```

> 注解
>
> 关于嵌套 selctor 的更多详细信息，请参考 嵌套选择器(selectors) 以及 选择器(Selectors) 文档中的 使用相对 XPaths 部分。

在我们的 spider 中加入这段代码:

```
import scrapy

class DmozSpider(scrapy.Spider):
    name = "dmoz"
    allowed_domains = ["dmoz.org"]
    start_urls = [
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/"
    ]

    def parse(self, response):
        for sel in response.xpath('//ul/li'):
            title = sel.xpath('a/text()').extract()
            link = sel.xpath('a/@href').extract()
            desc = sel.xpath('text()').extract()
            print title, link, desc
```

现在尝试再次爬取 dmoz.org，您将看到爬取到的网站信息被成功输出:

```
scrapy crawl dmoz
```

## 使用 item

Item 对象是自定义的 python 字典。您可以使用标准的字典语法来获取到其每个字段的值。(字段即是我们之前用 Field 赋值的属性):

```
>>> item = DmozItem()
>>> item['title'] = 'Example title'
>>> item['title']
'Example title'
```

一般来说，Spider 将会将爬取到的数据以 Item 对象返回。所以为了将爬取的数据返回，我们最终的代码将是:

```
import scrapy

from tutorial.items import DmozItem

class DmozSpider(scrapy.Spider):
    name = "dmoz"
    allowed_domains = ["dmoz.org"]
    start_urls = [
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/"
    ]

    def parse(self, response):
        for sel in response.xpath('//ul/li'):
            item = DmozItem()
            item['title'] = sel.xpath('a/text()').extract()
            item['link'] = sel.xpath('a/@href').extract()
            item['desc'] = sel.xpath('text()').extract()
            yield item
```

> 注解
>
> 您可以在 dirbot 项目中找到一个具有完整功能的 spider。该项目可以通过 https://github.com/scrapy/dirbot 找到。

现在对 dmoz.org 进行爬取将会产生 `DmozItem` 对象:

```
[dmoz] DEBUG: Scraped from <200 http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
    {'desc': [u' - By David Mertz; Addison Wesley. Book in progress, full text, ASCII format. Asks for feedback. [author web
     'link': [u'http://gnosis.cx/TPiP/'],
     'title': [u'Text Processing in Python']}
[dmoz] DEBUG: Scraped from <200 http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
    {'desc': [u' - By Sean McGrath; Prentice Hall PTR, 2000, ISBN 0130211192, has CD-ROM. Methods to build XML appli
     'link': [u'http://www.informit.com/store/product.aspx?isbn=0130211192'],
     'title': [u'XML Processing with Python']}
```

## 保存爬取到的数据

最简单存储爬取的数据的方式是使用 Feed exports：

```
scrapy crawl dmoz -o items.json
```

该命令将采用 JSON 格式对爬取的数据进行序列化，生成 items.json 文件。

在类似本篇教程里这样小规模的项目中，这种存储方式已经足够。 如果需要对爬取到的 item 做更多更为复杂的操作，您可以编写 Item Pipeline 。 类似于我们在创建项目时对 Item 做的，用于您编写自己的 tutorial/pipelines.py 也被创建。 不过如果您仅仅想要保存 item，您不需要实现任何的 pipeline。

# 下一步

本篇教程仅介绍了 Scrapy 的基础，还有很多特性没有涉及。请查看 初窥 Scrapy 章节中的 还有什么？ 部分,大致浏览大部分重要的特性。

接着，我们推荐您把玩一个例子(查看 例子 )，而后继续阅读 基本概念 。

# 4

# 例子

学习的最好方法就是参考例子，Scrapy 也不例外。Scrapy 提供了一个叫做 dirbot 的样例项目供您把玩学习。其包含了在教程中介绍的 dmoz spider。

您可以通过 https://github.com/scrapy/dirbot 找到 dirbot。其包含了 README 文件，详细介绍了项目的内容。

如果您熟悉 git，您可以 checkout 代码。或者您可以点击 Downloads 来下载项目的 tarball 或者 zip 的压缩包。

Snipplr 上的 scrapy 标签是用来分享 spider，middeware，extension 或者 script 代码片段。欢迎(并鼓励)在那分享您的代码。

5

# 命令行工具(Command line tools)

新版功能。

Scrapy 是通过 scrapy 命令行工具进行控制的。这里我们称之为 “Scrapy tool” 以用来和子命令进行区分。对于子命令，我们称为 “command” 或者 “Scrapy commands”。

Scrapy tool 针对不同的目的提供了多个命令，每个命令支持不同的参数和选项。

## 默认的 Scrapy 项目结构

在开始对命令行工具以及子命令的探索前，让我们首先了解一下 Scrapy 的项目的目录结构。

虽然可以被修改，但所有的 Scrapy 项目默认有类似于下边的文件结构:

```
scrapy.cfg
myproject/
    __init__.py
    items.py
    pipelines.py
    settings.py
    spiders/
        __init__.py
        spider1.py
        spider2.py
        ...
```

`scrapy.cfg` 存放的目录被认为是 项目的根目录 。该文件中包含 python 模块名的字段定义了项目的设置。例如:

```
[settings]
default = myproject.settings
```

# 使用 scrapy 工具

您可以以无参数的方式启动 Scrapy 工具。该命令将会给出一些使用帮助以及可用的命令：

```
Scrapy X.Y - no active project

Usage:
  scrapy <command> [options] [args]

Available commands:
  crawl        Run a spider
  fetch        Fetch a URL using the Scrapy downloader
[...]
```

如果您在 Scrapy 项目中运行，当前激活的项目将会显示在输出的第一行。上面的输出就是响应的例子。如果您在一个项目中运行命令将会得到类似的输出:

```
Scrapy X.Y - project: myproject

Usage:
  scrapy <command> [options] [args]

[...]
```

## 创建项目

一般来说，使用 `scrapy` 工具的第一件事就是创建您的 Scrapy 项目：

```
scrapy startproject myproject
```

该命令将会在 myproject 目录中创建一个 Scrapy 项目。

接下来，进入到项目目录中:

```
cd myproject
```

这时候您就可以使用 scrapy 命令来管理和控制您的项目了。

## 控制项目

您可以在您的项目中使用 `scrapy` 工具来对其进行控制和管理。

比如，创建一个新的 spider：

```
scrapy genspider mydomain mydomain.com
```

有些 Scrapy 命令(比如 `crawl` )要求必须在 Scrapy 项目中运行。 您可以通过下边的 `commands reference` 来了解哪些命令需要在项目中运行，哪些不用。

另外要注意，有些命令在项目里运行时的效果有些许区别。 以 fetch 命令为例，如果被爬取的 url 与某个特定 spider 相关联， 则该命令将会使用 spider 的动作(spider-overridden behaviours)。 (比如 spider 指定的 `user_agent` )。 该表现是有意而为之的。一般来说， `fetch` 命令就是用来测试检查 spider 是如何下载页面。

## 可用的工具命令(tool commands)

该章节提供了可用的内置命令的列表。每个命令都提供了描述以及一些使用例子。您总是可以通过运行命令来获取关于每个命令的详细内容：

```
scrapy <command> -h
```

您也可以查看所有可用的命令：

```
scrapy -h
```

Scrapy 提供了两种类型的命令。一种必须在 Scrapy 项目中运行(针对项目(Project-specific)的命令)，另外一种则不需要(全局命令)。全局命令在项目中运行时的表现可能会与在非项目中运行有些许差别(因为可能会使用项目的设定)。

全局命令：

- `startproject`
- `settings`
- `runspider`
- `shell`
- `fetch`
- `view`
- `version`

项目(Project-only)命令：

- `crawl`
- `check`
- `list`
- `edit`
- `parse`
- `genspider`
- `deploy`
- `bench`

## startproject

- 语法: `scrapy startproject <project_name>`
- 是否需要项目: no

在 project_name 文件夹下创建一个名为 project_name 的 Scrapy 项目。

例子:

```
$ scrapy startproject myproject
```

## genspider

- 语法: `scrapy genspider [-t template] <name> <domain>`
- 是否需要项目：yes

在当前项目中创建 spider。

这仅仅是创建 spider 的一种快捷方法。该方法可以使用提前定义好的模板来生成 spider。您也可以自己创建 spider 的源码文件。

例子:

```
$ scrapy genspider -l
Available templates:
  basic
  crawl
  csvfeed
  xmlfeed

$ scrapy genspider -d basic
import scrapy

class $classname(scrapy.Spider):
    name = "$name"
    allowed_domains = ["$domain"]
    start_urls = (
        'http://www.$domain/',
        )

    def parse(self, response):
```

```
    pass

$ scrapy genspider -t basic example example.com
Created spider 'example' using template 'basic' in module:
  mybot.spiders.example
```

## crawl

- 语法: `scrapy crawl <spider>`
- 是否需要项目：yes

使用 spider 进行爬取。

例子：

```
$ scrapy crawl myspider
[ ... myspider starts crawling ... ]
```

## check

- 语法： `scrapy check [-l] <spider>`
- 是否需要项目：yes

运行 contract 检查。

例子：

```
$ scrapy check -l
first_spider
  * parse
  * parse_item
second_spider
  * parse
  * parse_item

$ scrapy check
[FAILED] first_spider:parse_item
>>> 'RetailPricex' field is missing

[FAILED] first_spider:parse
>>> Returned 92 requests, expected 0..4
```

## list

- 语法：`scrapy list`
- 是否需要项目：yes

列出当前项目中所有可用的 spider。每行输出一个 spider。

使用例子：

```
$ scrapy list
spider1
spider2
```

## edit

- 语法：`scrapy edit <spider>`
- 是否需要项目：yes

使用 EDITOR 中设定的编辑器编辑给定的 spider

该命令仅仅是提供一个快捷方式。开发者可以自由选择其他工具或者 IDE 来编写调试 spider。

例子：

```
$ scrapy edit spider1
```

## fetch

- 语法：`scrapy fetch <url>`
- 是否需要项目：no

使用 Scrapy 下载器(downloader)下载给定的 URL，并将获取到的内容送到标准输出。

该命令以 spider 下载页面的方式获取页面。例如，如果 spider 有 `USER_AGENT` 属性修改了 User Agent，该命令将会使用该属性。

因此，您可以使用该命令来查看 spider 如何获取某个特定页面。

该命令如果非项目中运行则会使用默认 Scrapy downloader 设定。

例子：

```
$ scrapy fetch --nolog http://www.example.com/some/page.html
[ ... html content here ... ]

$ scrapy fetch --nolog --headers http://www.example.com/
{'Accept-Ranges': ['bytes'],
 'Age': ['1263   '],
 'Connection': ['close     '],
 'Content-Length': ['596'],
 'Content-Type': ['text/html; charset=UTF-8'],
 'Date': ['Wed, 18 Aug 2010 23:59:46 GMT'],
 'Etag': ['"573c1-254-48c9c87349680"'],
 'Last-Modified': ['Fri, 30 Jul 2010 15:30:18 GMT'],
 'Server': ['Apache/2.2.3 (CentOS)']}
```

## view

- 语法： `scrapy view <url>`
- 是否需要项目：no

在浏览器中打开给定的 URL，并以 Scrapy spider 获取到的形式展现。 有些时候 spider 获取到的页面和普通用户看到的并不相同。 因此该命令可以用来检查 spider 所获取到的页面，并确认这是您所期望的。

例子：

```
$ scrapy view http://www.example.com/some/page.html
[ ... browser starts ... ]
```

## shell

- 语法： `scrapy shell [url]`
- 是否需要项目：no

以给定的 URL(如果给出)或者空(没有给出 URL)启动 Scrapy shell。查看 Scrapy 终端(Scrapy shell) 获取更多信息。

例子：

```
$ scrapy shell http://www.example.com/some/page.html
[ ... scrapy shell starts ... ]
```

## parse

- 语法： `scrapy parse <url> [options]`
- 是否需要项目：yes

获取给定的 URL 并使用相应的 spider 分析处理。如果您提供 `--callback` 选项，则使用 spider 的该方法处理，否则使用 `parse` 。

支持的选项：

- `--spider=SPIDER`： 跳过自动检测 spider 并强制使用特定的 spider
- `--a NAME=VALUE`： 设置 spider 的参数(可能被重复)
- `--callback or -c`： spider 中用于解析返回(response)的回调函数
- `--pipelines`： 在 pipeline 中处理 item
- `--rules or -r`： 使用 CrawlSpider 规则来发现用来解析返回(response)的回调函数
- `--noitems`： 不显示爬取到的 item
- `--nolinks`： 不显示提取到的链接
- `--nocolour`： 避免使用 pygments 对输出着色
- `--depth or -d`： 指定跟进链接请求的层次数(默认：1)
- `--verbose or -v`： 显示每个请求的详细信息

例子：

```
$ scrapy parse http://www.example.com/ -c parse_item
[ ... scrapy log lines crawling example.com spider ... ]

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

[{'name': u'Example item',
 'category': u'Furniture',
 'length': u'12 cm'}]

 # Requests  -----------------------------------------------------------------

[]
```

## settings

- 语法：`scrapy settings [options]`
- 是否需要项目：no

获取 Scrapy 的设定

在项目中运行时，该命令将会输出项目的设定值，否则输出 Scrapy 默认设定。

例子：

```
$ scrapy settings --get BOT_NAME
scrapybot
$ scrapy settings --get DOWNLOAD_DELAY
0
```

## runspider

- 语法：`scrapy runspider <spider_file.py>`
- 是否需要项目：no

在未创建项目的情况下，运行一个编写在 Python 文件中的 spider。

例子：

```
$ scrapy runspider myspider.py
[ ... spider starts crawling ... ]
```

## version

- 语法：`scrapy version [-v]`
- 是否需要项目：no

输出 Scrapy 版本。配合 -v 运行时，该命令同时输出 Python，Twisted 以及平台的信息，方便 bug 提交。

## deploy

新版功能。

- 语法：`scrapy deploy [ <target:project> | -l <target> | -L ]`
- 是否需要项目：yes

将项目部署到 Scrapyd 服务。查看部署您的项目。

## bench

新版功能。

- 语法：scrapy bench
- 是否需要项目：no

运行 benchmark 测试。Benchmarking。

## 自定义项目命令

您也可以通过 COMMANDS_MODULE 来添加您自己的项目命令。您可以以 scrapy/commands 中 Scrapy commands 为例来了解如何实现您的命令。

# COMMANDS_MODULE

```
Default: '' (empty string)
```

用于查找添加自定义 Scrapy 命令的模块。

例子:

```
COMMANDS_MODULE = 'mybot.commands'
```


Items

爬取的主要目标就是从非结构性的数据源提取结构性数据，例如网页。Scrapy 提供 `Item` 类来满足这样的需求。

Item 对象是种简单的容器，保存了爬取到得数据。其提供了类似于词典(dictionary-like)的API以及用于声明可用字段的简单语法。

## 声明 Item

Item 使用简单的 class 定义语法以及 `Field` 对象来声明。例如:

```
import scrapy

class Product(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field()
    stock = scrapy.Field()
    last_updated = scrapy.Field(serializer=str)
```

> 注解
>
> 熟悉 Django 的朋友一定会注意到 Scrapy Item 定义方式与 Django Models 很类似, 不过没有那么多不同的字段类型(Field type)，更为简单。

## Item字段(Item Fields)

Field 对象指明了每个字段的元数据(metadata)。例如下面例子中 `last_updated` 中指明了该字段的序列化函数。

您可以为每个字段指明任何类型的元数据。Field 对象对接受的值没有任何限制。也正是因为这个原因，文档也无法提供所有可用的元数据的键(key)参考列表。Field 对象中保存的每个键可以由多个组件使用，并且只有这些组件知道这个键的存在。您可以根据自己的需求，定义使用其他的 Field 键。 设置 Field 对象的主要目的就是在一个地方定义好所有的元数据。一般来说，那些依赖某个字段的组件肯定使用了特定的键(key)。您必须查看组件相关的文档，查看其用了哪些元数据键(metadata key)。

需要注意的是，用来声明 item 的 Field 对象并没有被赋值为 class 的属性。不过您可以通过 Item.fields 属性进行访问。

以上就是所有您需要知道的如何声明 item 的内容了。

## 与 Item 配合

接下来以下边声明的 Product item 来演示一些 item 的操作。您会发现 API 和 dict API 非常相似。

### 创建 item

```
>>> product = Product(name='Desktop PC', price=1000)
>>> print product
Product(name='Desktop PC', price=1000)
```

### 获取字段的值

```
>>> product['name']
Desktop PC
>>> product.get('name')
Desktop PC

>>> product['price']
1000

>>> product['last_updated']
Traceback (most recent call last):
  ...
KeyError: 'last_updated'

>>> product.get('last_updated', 'not set')
not set

>>> product['lala'] # getting unknown field
Traceback (most recent call last):
  ...
KeyError: 'lala'

>>> product.get('lala', 'unknown field')
'unknown field'

>>> 'name' in product  # is name field populated?
True

>>> 'last_updated' in product  # is last_updated populated?
```

```
False

>>> 'last_updated' in product.fields  # is last_updated a declared field?
True

>>> 'lala' in product.fields  # is lala a declared field?
False
```

## 设置字段的值

```
>>> product['last_updated'] = 'today'
>>> product['last_updated']
today

>>> product['lala'] = 'test' # setting unknown field
Traceback (most recent call last):
    ...
KeyError: 'Product does not support field: lala'
```

## 获取所有获取到的值

您可以使用 dict API 来获取所有的值:

```
>>> product.keys()
['price', 'name']

>>> product.items()
[('price', 1000), ('name', 'Desktop PC')]
```

## 其他任务

复制 item:

```
>>> product2 = Product(product)
>>> print product2
Product(name='Desktop PC', price=1000)

>>> product3 = product2.copy()
>>> print product3
Product(name='Desktop PC', price=1000)
```

## 根据 item 创建字典(dict)

```
>>> dict(product) # create a dict from all populated values
{'price': 1000, 'name': 'Desktop PC'}
```

## 根据字典(dict)创建 item

```
>>> Product({'name': 'Laptop PC', 'price': 1500})
Product(price=1500, name='Laptop PC')

>>> Product({'name': 'Laptop PC', 'lala': 1500}) # warning: unknown field in dict
Traceback (most recent call last):
  ...
KeyError: 'Product does not support field: lala'
```

## 扩展 Item

您可以通过继承原始的 Item 来扩展 item(添加更多的字段或者修改某些字段的元数据)。

例如:

```
class DiscountedProduct(Product):
    discount_percent = scrapy.Field(serializer=str)
    discount_expiration_date = scrapy.Field()
```

您也可以通过使用原字段的元数据，添加新的值或修改原来的值来扩展字段的元数据:

```
class SpecificProduct(Product):
    name = scrapy.Field(Product.fields['name'], serializer=my_serializer)
```

这段代码在保留所有原来的元数据值的情况下添加(或者覆盖)了 name 字段的 serializer。

## Item 对象

```
class scrapy.item.Item([arg])
```

返回一个根据给定的参数可选初始化的 item。

Item复制了标准的 dict API。包括初始化函数也相同。Item 唯一额外添加的属性是：

```
fields
```

一个包含了 item 所有声明的字段的字典，而不仅仅是获取到的字段。该字典的 key 是字段(field)的名字，值是 Item 声明中使用到的 `Field` 对象。

## 字段(Field)对象

```
class scrapy.item.Field([arg])
```

Field 仅仅是内置的 dict 类的一个别名，并没有提供额外的方法或者属性。换句话说，Field 对象完完全全就是 Python 字典(dict)。被用来基于类属性(class attribute)的方法来支持 item 声明语法。

# 7

# Spiders

Spider 类定义了如何爬取某个(或某些)网站。包括了爬取的动作(例如：是否跟进链接)以及如何从网页的内容中提取结构化数据(爬取 item)。换句话说，Spider 就是您定义爬取的动作及分析某个网页(或者是有些网页)的地方。

对 spider 来说，爬取的循环类似下文:

1. 以初始的 URL 初始化 Request，并设置回调函数。当该 request 下载完毕并返回时，将生成 respons e，并作为参数传给该回调函数。

   spider 中初始的 request 是通过调用 `start_requests()` 来获取的。`start_requests()` 读取 `start_url s` 中的 URL， 并以 `parse` 为回调函数生成 `Request` 。

2. 在回调函数内分析返回的(网页)内容，返回 Item 对象或者 Request 或者一个包括二者的可迭代容器。 返回的 Request 对象之后会经过 Scrapy 处理，下载相应的内容，并调用设置的 callback 函数(函数可相同)。

3. 在回调函数内，您可以使用 选择器(Selectors) (您也可以使用 BeautifulSoup, lxml 或者您想用的任何解析器) 来分析网页内容，并根据分析的数据生成 item。

4. 最后，由 spider 返回的 item 将被存到数据库(由某些 Item Pipeline 处理)或使用 Feed exports 存入到文件中。

虽然该循环对任何类型的 spider 都(多少)适用，但 Scrapy 仍然为了不同的需求提供了多种默认 spider。 之后将讨论这些 spider。

## Spider 参数

Spider 可以通过接受参数来修改其功能。 spider 参数一般用来定义初始 URL 或者指定限制爬取网站的部分。您也可以使用其来配置 spider 的任何功能。

在运行 `crawl` 时添加 `-a` 可以传递 Spider 参数：

```
scrapy crawl myspider -a category=electronics
```

Spider 在构造器(constructor)中获取参数：

```
import scrapy

class MySpider(Spider):
    name = 'myspider'

    def __init__(self, category=None, *args, **kwargs):
        super(MySpider, self).__init__(*args, **kwargs)
        self.start_urls = ['http://www.example.com/categories/%s' % category]
        # ...
```

Spider 参数也可以通过 Scrapyd 的 `schedule.json API` 来传递。参见 Scrapyd documentation。

# 内置 Spider 参考手册

Scrapy 提供多种方便的通用 spider 供您继承使用。 这些 spider 为一些常用的爬取情况提供方便的特性， 例如根据某些规则跟进某个网站的所有链接、根据 Sitemaps 来进行爬取，或者分析 XML/CSV 源。

下面 spider 的示例中，我们假定您有个项目在 `myproject.items` 模块中声明了 `TestItem` :

```
import scrapy

class TestItem(scrapy.Item):
    id = scrapy.Field()
    name = scrapy.Field()
    description = scrapy.Field()
```

## Spider

### class scrapy.spider.Spider

Spider 是最简单的 spider。每个其他的 spider 必须继承自该类(包括 Scrapy 自带的其他 spider 以及您自己编写的 spider)。Spider 并没有提供什么特殊的功能。其仅仅请求给定的 start_urls/start_requests，并根据返回的结果(resulting responses)调用 spider 的 parse 方法。

#### name

定义 spider 名字的字符串(string)。spider 的名字定义了 Scrapy 如何定位(并初始化)spider，所以其必须是唯一的。 不过您可以生成多个相同的 spider 实例(instance)，这没有任何限制。 name 是 spider 最重要的属性，而且是必须的。

如果该 spider 爬取单个网站(single domain)，一个常见的做法是以该网站(domain)(加或不加后缀)来命名 spider。 例如，如果 spider 爬取 `mywebsite.com` ，该 spider 通常会被命名为 mywebsite 。

#### allowed_domains

可选。包含了 spider 允许爬取的域名(domain)列表(list)。 当 `OffsiteMiddleware` 启用时， 域名不在列表中的 URL 不会被跟进。

#### start_urls

URL 列表。当没有制定特定的 URL 时，spider 将从该列表中开始进行爬取。 因此，第一个被获取到的页面的 URL 将是该列表之一。后续的 URL 将会从获取到的数据中提取。

start_requests()

该方法必须返回一个可迭代对象(iterable)。该对象包含了 spider 用于爬取的第一个 Request。

当 spider 启动爬取并且未制定 URL 时，该方法被调用。 当指定了 URL 时， `make_requests_from_url()` 将被调用来创建 Request 对象。 该方法仅仅会被 Scrapy 调用一次，因此您可以将其实现为生成器。

该方法的默认实现是使用 `start_urls` 的 url 生成 Request。

如果您想要修改最初爬取某个网站的 Request 对象，您可以重写(override)该方法。 例如，如果您需要在启动时以 POST 登录某个网站，你可以这么写:

```
def start_requests(self):
    return [scrapy.FormRequest("http://www.example.com/login",
                               formdata={'user': 'john', 'pass': 'secret'},
                               callback=self.logged_in)]

def logged_in(self, response):
    # here you would extract links to follow and return Requests for
    # each of them, with another callback
    pass
```

make_requests_from_url(url)

该方法接受一个 URL 并返回用于爬取的 Request 对象。该方法在初始化 request 时被 start_requests()调用，也被用于转化 url 为 request。

默认未被复写(overridden)的情况下，该方法返回的 Request 对象中， `parse()` 作为回调函数，dont_filter 参数也被设置为开启。(详情参见 `Request` )。

parse(response)

当 response 没有指定回调函数时，该方法是 Scrapy 处理下载的 response 的默认方法。

parse 负责处理 response 并返回处理的数据以及(/或)跟进的 URL。 Spider 对其他的 Request 的回调函数也有相同的要求。

该方法及其他的 Request 回调函数必须返回一个包含 Request 及(或) Item 的可迭代的对象。

**参数:** response ( `Response` ) – 用于分析的 response

log(message[, level, component])

使用 scrapy.log.msg() 方法记录(log)message。log 中自动带上该 spider 的 name 属性。 更多数据请参见 Logging。

closed(reason)

当 spider 关闭时，该函数被调用。 该方法提供了一个替代调用 signals.connect()来监听 spider_closed 信号的快捷方式。

## Spider 样例

让我们来看一个例子:

```
import scrapy

class MySpider(scrapy.Spider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = [
        'http://www.example.com/1.html',
        'http://www.example.com/2.html',
        'http://www.example.com/3.html',
    ]

    def parse(self, response):
        self.log('A response from %s just arrived!' % response.url)
```

另一个在单个回调函数中返回多个 Request 以及 Item 的例子:

```
import scrapy
from myproject.items import MyItem

class MySpider(scrapy.Spider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = [
        'http://www.example.com/1.html',
        'http://www.example.com/2.html',
        'http://www.example.com/3.html',
    ]

    def parse(self, response):
        sel = scrapy.Selector(response)
        for h3 in response.xpath('//h3').extract():
            yield MyItem(title=h3)

        for url in response.xpath('//a/@href').extract():
            yield scrapy.Request(url, callback=self.parse)
```

## CrawlSpider

class scrapy.contrib.spiders.CrawlSpider

爬取一般网站常用的 spider。其定义了一些规则(rule)来提供跟进 link 的方便的机制。 也许该 spider 并不是完全适合您的特定网站或项目，但其对很多情况都使用。因此您可以以其为起点，根据需求修改部分方法。当然您也可以实现自己的 spider。

除了从 Spider 继承过来的(您必须提供的)属性外，其提供了一个新的属性:

rules
一个包含一个(或多个) Rule 对象的集合(list)。 每个 Rule 对爬取网站的动作定义了特定表现。 Rule 对象在下边会介绍。 如果多个 rule 匹配了相同的链接，则根据他们在本属性中被定义的顺序，第一个会被使用。

该 spider 也提供了一个可复写(overrideable)的方法:

parse_start_url(response)
当 start_url 的请求返回时，该方法被调用。 该方法分析最初的返回值并必须返回一个 Item 对象或者 一个 Request 对象或者 一个可迭代的包含二者对象。

## 爬取规则(Crawling rules)

class scrapy.contrib.spiders.Rule(link_extractor, callback=None, cb_kwargs=None, follow=None, process_links=None, process_request=None)

link_extractor 是一个 Link Extractor 对象。其定义了如何从爬取到的页面提取链接。

callback 是一个 callable 或 string(该 spider 中同名的函数将会被调用)。从 link_extractor 中每获取到链接时将会调用该函数。该回调函数接受一个 response 作为其第一个参数，并返回一个包含 Item 以及(或) Request 对象(或者这两者的子类)的列表(list)。

> 警告
>
> 当编写爬虫规则时，请避免使用 parse 作为回调函数。 由于 CrawlSpider 使用 parse 方法来实现其逻辑，如果 您覆盖了 parse 方法，crawl spider 将会运行失败。

`cb_kwargs` 包含传递给回调函数的参数(keyword argument)的字典。

`follow` 是一个布尔(boolean)值，指定了根据该规则从 response 提取的链接是否需要跟进。 如果 `callback` 为 None， `follow` 默认设置为 `True` ，否则默认为 `False` 。

process_links 是一个 callable 或 string(该 spider 中同名的函数将会被调用)。 从 link_extractor 中获取到链接列表时将会调用该函数。该方法主要用来过滤。

process_request 是一个 callable 或 string(该 spider 中同名的函数将会被调用)。 该规则提取到每个 request 时都会调用该函数。该函数必须返回一个 request 或者 None。 (用来过滤 request)

## CrawlSpider 样例

接下来给出配合 rule 使用 CrawlSpider 的例子:

```
import scrapy
from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors import LinkExtractor

class MySpider(CrawlSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com']

    rules = (
        # 提取匹配 'category.php' (但不匹配 'subsection.php') 的链接并跟进链接(没有 callback 意味着 follow 默认为 True)
        Rule(LinkExtractor(allow=('category\.php', ), deny=('subsection\.php', ))),

        # 提取匹配 'item.php' 的链接并使用 spider 的 parse_item 方法进行分析
        Rule(LinkExtractor(allow=('item\.php', )), callback='parse_item'),
    )

    def parse_item(self, response):
        self.log('Hi, this is an item page! %s' % response.url)

        item = scrapy.Item()
        item['id'] = response.xpath('//td[@id="item_id"]/text()').re(r'ID: (\d+)')
        item['name'] = response.xpath('//td[@id="item_name"]/text()').extract()
        item['description'] = response.xpath('//td[@id="item_description"]/text()').extract()
        return item
```

该 spider 将从 example.com 的首页开始爬取，获取 category 以及 item 的链接并对后者使用 parse_item 方法。当 item 获得返回(response)时，将使用 XPath 处理 HTML 并生成一些数据填入 Item 中。

## XMLFeedSpider

class scrapy.contrib.spiders.XMLFeedSpider

XMLFeedSpider 被设计用于通过迭代各个节点来分析 XML 源(XML feed)。迭代器可以从 iternodes，xml，html 选择。鉴于 xml 以及 html 迭代器需要先读取所有 DOM 再分析而引起的性能问题，一般还是推荐使用 iternodes。不过使用 html 作为迭代器能有效应对错误的 XML。

您必须定义下列类属性来设置迭代器以及标签名(tag name):

iterator
用于确定使用哪个迭代器的 string。可选项有:

- 'iternodes' –一个高性能的基于正则表达式的迭代器
- 'html' –使用 Selector 的迭代器。 需要注意的是该迭代器使用 DOM 进行分析，其需要将所有的 DOM 载入内存， 当数据量大的时候会产生问题。
- 'xml' –使用 Selector 的迭代器。 需要注意的是该迭代器使用 DOM 进行分析，其需要将所有的DOM 载入内存，当数据量大的时候会产生问题。

默认值为 iternodes 。

itertag
一个包含开始迭代的节点名的 string。例如:

```
itertag = 'product'
```

namespaces
一个由 (prefix, url) 元组(tuple)所组成的 list。 其定义了在该文档中会被 spider 处理的可用的 namespace。 prefix 及 uri 会被自动调用 `register_namespace()` 生成 namespace。

您可以通过在 `itertag` 属性中制定节点的 namespace。

例如:

```
class YourSpider(XMLFeedSpider):

    namespaces = [('n', 'http://www.sitemaps.org/schemas/sitemap/0.9')]
    itertag = 'n:url'
    # ...
```

除了这些新的属性之外，该 spider 也有以下可以覆盖(overrideable)的方法:

adapt_response(response)

该方法在 spider 分析 response 前被调用。您可以在 response 被分析之前使用该函数来修改内容(body)。该方法接受一个 response 并返回一个 response(可以相同也可以不同)。

parse_node(response, selector)

当节点符合提供的标签名时(itertag)该方法被调用。 接收到的 response 以及相应的 Selector 作为参数传递给该方法。 该方法返回一个 Item 对象或者 Request 对象 或者一个包含二者的可迭代对象(iterable)。

process_results(response, results)

当 spider 返回结果(item 或 request)时该方法被调用。设定该方法的目的是在结果返回给框架核心(framework core)之前做最后的处理，例如设定 item 的 ID。其接受一个结果的列表(list of results)及对应的 response。其结果必须返回一个结果的列表(list of results)(包含 Item 或者 Request 对象)。

## XMLFeedSpider 例子

该 spider 十分易用。下边是其中一个例子:

```
from scrapy import log
from scrapy.contrib.spiders import XMLFeedSpider
from myproject.items import TestItem

class MySpider(XMLFeedSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com/feed.xml']
    iterator = 'iternodes' # This is actually unnecessary, since it's the default value
    itertag = 'item'

    def parse_node(self, response, node):
        log.msg('Hi, this is a <%s> node!: %s' % (self.itertag, ''.join(node.extract())))

        item = TestItem()
        item['id'] = node.xpath('@id').extract()
        item['name'] = node.xpath('name').extract()
        item['description'] = node.xpath('description').extract()
        return item
```

简单来说，我们在这里创建了一个 spider，从给定的 start_urls 中下载 feed，并迭代 feed 中每个 item 标签，输出，并在 Item 中存储有些随机数据。

## CSVFeedSpider

class scrapy.contrib.spiders.CSVFeedSpider

该 spider 除了其按行遍历而不是节点之外其他和 XMLFeedSpider 十分类似。 而其在每次迭代时调用的是 `parse_row()` 。

delimiter

在 CSV 文件中用于区分字段的分隔符。类型为 string。 默认为 ',' (逗号)。

quotechar

A string with the enclosure character for each field in the CSV file Defaults to "" (quotation mark).

headers

在 CSV 文件中包含的用来提取字段的行的列表。参考下边的例子。

parse_row(response, row)

该方法接收一个 response 对象及一个以提供或检测出来的 header 为键的字典(代表每行)。 该 spider 中，您也可以覆盖 `adapt_response` 及 `process_results` 方法来进行预处理(pre-processing)及后(post-processing)处理。

## CSVFeedSpider 例子

下面的例子和之前的例子很像，但使用了 `CSVFeedSpider` :

```
from scrapy import log
from scrapy.contrib.spiders import CSVFeedSpider
from myproject.items import TestItem

class MySpider(CSVFeedSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com/feed.csv']
    delimiter = ';'
    quotechar = "'"
    headers = ['id', 'name', 'description']

    def parse_row(self, response, row):
        log.msg('Hi, this is a row!: %r' % row)

        item = TestItem()
```

```
    item['id'] = row['id']
    item['name'] = row['name']
    item['description'] = row['description']
    return item
```

## SitemapSpider

### class scrapy.contrib.spiders.SitemapSpider

SitemapSpider 使您爬取网站时可以通过 Sitemaps 来发现爬取的 URL。

其支持嵌套的 sitemap，并能从 robots.txt 中获取 sitemap 的 url。

sitemap_urls
包含您要爬取的 url 的 sitemap 的 url 列表(list)。您也可以指定为一个 robots.txt，spider 会从中分析并提取 url。

sitemap_rules
一个包含 (regex, callback) 元组的列表(list):

- `regex` 是一个用于匹配从 sitemap 提供的 url 的正则表达式。`regex` 可以是一个字符串或者编译的正则对象(compiled regex object)。
- callback 指定了匹配正则表达式的 url 的处理函数。`callback` 可以是一个字符串(spider 中方法的名字)或者是 callable。

例如:

```
sitemap_rules = [('/product/', 'parse_product')]
```

规则按顺序进行匹配，之后第一个匹配才会被应用。

如果您忽略该属性，sitemap 中发现的所有 url 将会被 parse 函数处理。

sitemap_follow
一个用于匹配要跟进的 sitemap 的正则表达式的列表(list)。其仅仅被应用在 使用 *Sitemap index files* 来指向其他 sitemap 文件的站点。

默认情况下所有的 sitemap 都会被跟进。

sitemap_alternate_links
指定当一个 url 有可选的链接时，是否跟进。 有些非英文网站会在一个 url 块内提供其他语言的网站链接。

例如:

```
<url>
    <loc>http://example.com/</loc>
    <xhtml:link rel="alternate" hreflang="de" href="http://example.com/de"/>
</url>
```

当 `sitemap_alternate_links` 设置时，两个 URL 都会被获取。当 `sitemap_alternate_links` 关闭时，只有 `http://example.com/` 会被获取。

默认 `sitemap_alternate_links` 关闭。

## SitemapSpider 样例

简单的例子:使用 parse 处理通过 sitemap 发现的所有 url:

```
from scrapy.contrib.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/sitemap.xml']

    def parse(self, response):
        pass # ... scrape item here ...
```

用特定的函数处理某些 url，其他的使用另外的 callback:

```
from scrapy.contrib.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/sitemap.xml']
    sitemap_rules = [
        ('/product/', 'parse_product'),
        ('/category/', 'parse_category'),
    ]

    def parse_product(self, response):
        pass # ... scrape product ...

    def parse_category(self, response):
        pass # ... scrape category ...
```

跟进 robots.txt 文件定义的 sitemap 并只跟进包含有 `..sitemap_shop` 的 url:

```
from scrapy.contrib.spiders import SitemapSpider

class MySpider(SitemapSpider):
```

```
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/', 'parse_shop'),
    ]
    sitemap_follow = ['/sitemap_shops']

    def parse_shop(self, response):
        pass # ... scrape shop here ...
```

在 SitemapSpider 中使用其他 url:

```
from scrapy.contrib.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/', 'parse_shop'),
    ]

    other_urls = ['http://www.example.com/about']

    def start_requests(self):
        requests = list(super(MySpider, self).start_requests())
        requests += [scrapy.Request(x, self.parse_other) for x in self.other_urls]
        return requests

    def parse_shop(self, response):
        pass # ... scrape shop here ...

    def parse_other(self, response):
        pass # ... scrape other here ...
```

# 8

# 选择器(Selectors)

当抓取网页时，你做的最常见的任务是从 HTML 源码中提取数据。现有的一些库可以达到这个目的：

- BeautifulSoup 是在程序员间非常流行的网页分析库，它基于 HTML 代码的结构来构造一个 Python 对象， 对不良标记的处理也非常合理，但它有一个缺点：慢。
- lxml 是一个基于 ElementTree(不是 Python 标准库的一部分)的 python 化的 XML 解析库(也可以解析 HTML)。

Scrapy 提取数据有自己的一套机制。它们被称作选择器(seletors)，因为他们通过特定的 XPath 或者 CSS 表达式来“选择” HTML 文件中的某个部分。

XPath 是一门用来在 XML 文件中选择节点的语言，也可以用在 HTML 上。CSS 是一门将 HTML 文档样式化的语言。选择器由它定义，并与特定的 HTML 元素的样式相关连。

Scrapy 选择器构建于 lxml 库之上，这意味着它们在速度和解析准确性上非常相似。

本页面解释了选择器如何工作，并描述了相应的 API。不同于 lxml API 的臃肿，该 API 短小而简洁。这是因为 lxml 库除了用来选择标记化文档外，还可以用到许多任务上。

选择器 API 的完全参考详见 Selector reference

# 使用选择器(selectors)

## 构造选择器(selectors)

Scrapy selector 是以 文字(text) 或 `TextResponse` 构造的 `Selector` 实例。 其根据输入的类型自动选择最优的分析方法(XML vs HTML):

```
>>> from scrapy.selector import Selector
>>> from scrapy.http import HtmlResponse
```

以文字构造:

```
>>> body = '<html><body><span>good</span></body></html>'
>>> Selector(text=body).xpath('//span/text()').extract()
[u'good']
```

以 response 构造:

```
>>> response = HtmlResponse(url='http://example.com', body=body)
>>> Selector(response=response).xpath('//span/text()').extract()
[u'good']
```

为了方便起见，response 对象以.selector 属性提供了一个 selector，您可以随时使用该快捷方法:

```
>>> response.selector.xpath('//span/text()').extract()
[u'good']
```

## 使用选择器(selectors)

我们将使用 Scrapy shell (提供交互测试)和位于 Scrapy 文档服务器的一个样例页面，来解释如何使用选择器:

http://doc.scrapy.org/en/latest/_static/selectors-sample1.html

这里是它的 HTML 源码:

```
<html>
 <head>
  <base href='http://example.com/' />
  <title>Example website</title>
 </head>
 <body>
```

```
 <div id='images'>
  <a href='image1.html'>Name: My image 1 <br /><img src='image1_thumb.jpg' /></a>
  <a href='image2.html'>Name: My image 2 <br /><img src='image2_thumb.jpg' /></a>
  <a href='image3.html'>Name: My image 3 <br /><img src='image3_thumb.jpg' /></a>
  <a href='image4.html'>Name: My image 4 <br /><img src='image4_thumb.jpg' /></a>
  <a href='image5.html'>Name: My image 5 <br /><img src='image5_thumb.jpg' /></a>
 </div>
</body>
</html>
```

首先，我们打开 shell:

```
scrapy shell http://doc.scrapy.org/en/latest/_static/selectors-sample1.html
```

接着，当 shell 载入后，您将获得名为 `response` 的 shell 变量，其为响应的 response，并且在其 `response.selector` 属性上绑定了一个 selector。

因为我们处理的是 HTML，选择器将自动使用 HTML 语法分析。

那么，通过查看 HTML code 该页面的源码，我们构建一个 XPath 来选择 title 标签内的文字:

```
>>> response.selector.xpath('//title/text()')
[<Selector (text) xpath=//title/text()>]
```

由于在 response 中使用 XPath、CSS 查询十分普遍，因此，Scrapy 提供了两个实用的快捷方式: response.xpath()及 response.css():

```
>>> response.xpath('//title/text()')
[<Selector (text) xpath=//title/text()>]
>>> response.css('title::text')
[<Selector (text) xpath=//title/text()>]
```

如你所见，.xpath()及.css()方法返回一个类 SelectorList 的实例，它是一个新选择器的列表。这个 API 可以用来快速的提取嵌套数据。

为了提取真实的原文数据，你需要调用.extract() 方法如下:

```
>>> response.xpath('//title/text()').extract()
[u'Example website']
```

注意 CSS 选择器可以使用 CSS3 伪元素(pseudo-elements)来选择文字或者属性节点:

```
>>> response.css('title::text').extract()
[u'Example website']
```

现在我们将得到根 URL(base URL)和一些图片链接:

```
>>> response.xpath('//base/@href').extract()
[u'http://example.com/']

>>> response.css('base::attr(href)').extract()
[u'http://example.com/']

>>> response.xpath('//a[contains(@href, "image")]/@href').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']

>>> response.css('a[href*=image]::attr(href)').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']

>>> response.xpath('//a[contains(@href, "image")]/img/@src').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']

>>> response.css('a[href*=image] img::attr(src)').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']
```

## 嵌套选择器(selectors)

选择器方法( .xpath() or .css() )返回相同类型的选择器列表，因此你也可以对这些选择器调用选择器方法。下面是一个例子:

```
>>> links = response.xpath('//a[contains(@href, "image")]')
>>> links.extract()
[u'<a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>',
 u'<a href="image2.html">Name: My image 2 <br><img src="image2_thumb.jpg"></a>',
 u'<a href="image3.html">Name: My image 3 <br><img src="image3_thumb.jpg"></a>',
```

```
 u'<a href="image4.html">Name: My image 4 <br><img src="image4_thumb.jpg"></a>',
 u'<a href="image5.html">Name: My image 5 <br><img src="image5_thumb.jpg"></a>']

>>> for index, link in enumerate(links):
        args = (index, link.xpath('@href').extract(), link.xpath('img/@src').extract())
        print 'Link number %d points to url %s and image %s' % args

Link number 0 points to url [u'image1.html'] and image [u'image1_thumb.jpg']
Link number 1 points to url [u'image2.html'] and image [u'image2_thumb.jpg']
Link number 2 points to url [u'image3.html'] and image [u'image3_thumb.jpg']
Link number 3 points to url [u'image4.html'] and image [u'image4_thumb.jpg']
Link number 4 points to url [u'image5.html'] and image [u'image5_thumb.jpg']
```

## 结合正则表达式使用选择器(selectors)

Selector 也有一个.re()方法，用来通过正则表达式来提取数据。然而，不同于使用.xpath()或者.css()方法，.re()方法返回 unicode 字符串的列表。所以你无法构造嵌套式的.re()调用。

下面是一个例子，从上面的 HTML code 中提取图像名字:

```
>>> response.xpath('//a[contains(@href, "image")]/text()').re(r'Name:\s*(.*)')
[u'My image 1',
 u'My image 2',
 u'My image 3',
 u'My image 4',
 u'My image 5']
```

## 使用相对 XPaths

记住如果你使用嵌套的选择器，并使用起始为 `/` 的 XPath，那么该 XPath 将对文档使用绝对路径，而且对于你调用的 `Selector` 不是相对路径。

比如，假设你想提取在 `<div>` 元素中的所有 `<p>` 元素。首先，你将先得到所有的 `<div>` 元素:

```
>>> divs = response.xpath('//div')
```

开始时，你可能会尝试使用下面的错误的方法，因为它其实是从整篇文档中，而不仅仅是从那些 `<div>` 元素内部提取所有的 `<p>` 元素:

```
>>> for p in divs.xpath('//p'):  # this is wrong - gets all <p> from the whole document
...     print p.extract()
```

下面是比较合适的处理方法(注意 .//p XPath 的点前缀):

```
>>> for p in divs.xpath('.//p'):  # extracts all <p> inside
...     print p.extract()
```

另一种常见的情况将是提取所有直系 <p> 的结果:

```
>>> for p in divs.xpath('p'):
...     print p.extract()
```

更多关于相对 XPaths 的细节详见 XPath 说明中的 Location Paths 部分。

## 使用 EXSLT 扩展

因建于 lxml 之上，Scrapy 选择器也支持一些 EXSLT 扩展，可以在 XPath 表达式中使用这些预先制定的命名空间:

| 前缀 | 命名空间 | 用途 |
| --- | --- | --- |
| re | http://exslt.org/regular-expressions | 正则表达式 |
| set | http://exslt.org/sets | 集合操作 |

## 正则表达式

例如在 XPath 的 starts-with() 或 contains() 无法满足需求时， test() 函数可以非常有用。

例如在列表中选择有”class”元素且结尾为一个数字的链接:

```
>>> from scrapy import Selector
>>> doc = """
... <div>
...     <ul>
...         <li class="item-0"><a href="link1.html">first item</a></li>
...         <li class="item-1"><a href="link2.html">second item</a></li>
...         <li class="item-inactive"><a href="link3.html">third item</a></li>
...         <li class="item-1"><a href="link4.html">fourth item</a></li>
...         <li class="item-0"><a href="link5.html">fifth item</a></li>
...     </ul>
... </div>
... """
>>> sel = Selector(text=doc, type="html")
>>> sel.xpath('//li//@href').extract()
[u'link1.html', u'link2.html', u'link3.html', u'link4.html', u'link5.html']
```

```
>>> sel.xpath('//li[re:test(@class, "item-\d$")]//@href').extract()
[u'link1.html', u'link2.html', u'link4.html', u'link5.html']
>>>
```

> 警告
>
> C 语言库 `libxslt` 不原生支持 EXSLT 正则表达式，因此 lxml 在实现时使用了 Python `re` 模块的钩子。 因此，在 XPath 表达式中使用 regexp 函数可能会牺牲少量的性能。

## 集合操作

集合操作可以方便地用于在提取文字元素前从文档树中去除一些部分。

例如使用 itemscopes 组和对应的 itemprops 来提取微数据(来自 http://schema.org/Product 的样本内容):

```
>>> doc = """
... <div itemscope itemtype="http://schema.org/Product">
...   <span itemprop="name">Kenmore White 17" Microwave</span>
...   <img src="kenmore-microwave-17in.jpg" alt='Kenmore 17" Microwave' />
...   <div itemprop="aggregateRating"
...    itemscope itemtype="http://schema.org/AggregateRating">
...    Rated <span itemprop="ratingValue">3.5</span>/5
...    based on <span itemprop="reviewCount">11</span> customer reviews
...   </div>
...
...   <div itemprop="offers" itemscope itemtype="http://schema.org/Offer">
...     <span itemprop="price">$55.00</span>
...     <link itemprop="availability" href="http://schema.org/InStock" />In stock
...   </div>
...
...   Product description:
...   <span itemprop="description">0.7 cubic feet countertop microwave.
...   Has six preset cooking categories and convenience features like
...   Add-A-Minute and Child Lock.</span>
...
...   Customer reviews:
...
...   <div itemprop="review" itemscope itemtype="http://schema.org/Review">
...     <span itemprop="name">Not a happy camper</span> -
...     by <span itemprop="author">Ellie</span>,
...     <meta itemprop="datePublished" content="2011-04-01">April 1, 2011
...     <div itemprop="reviewRating" itemscope itemtype="http://schema.org/Rating">
...       <meta itemprop="worstRating" content = "1">
...       <span itemprop="ratingValue">1</span>/
```

```
...       <span itemprop="bestRating">5</span>stars
...     </div>
...     <span itemprop="description">The lamp burned out and now I have to replace
...     it. </span>
...   </div>
...
...   <div itemprop="review" itemscope itemtype="http://schema.org/Review">
...     <span itemprop="name">Value purchase</span> -
...     by <span itemprop="author">Lucas</span>,
...     <meta itemprop="datePublished" content="2011-03-25">March 25, 2011
...     <div itemprop="reviewRating" itemscope itemtype="http://schema.org/Rating">
...       <meta itemprop="worstRating" content = "1"/>
...       <span itemprop="ratingValue">4</span>/
...       <span itemprop="bestRating">5</span>stars
...     </div>
...     <span itemprop="description">Great microwave for the price. It is small and
...     fits in my apartment.</span>
...   </div>
...   ...
... </div>
... """
>>>
>>> for scope in sel.xpath('//div[@itemscope]'):
...     print "current scope:", scope.xpath('@itemtype').extract()
...     props = scope.xpath("'
...                 set:difference(./descendant::*/@itemprop,
...                                .//*[@itemscope]/*/@itemprop)'")
...     print "    properties:", props.extract()
...     print
...

current scope: [u'http://schema.org/Product']
    properties: [u'name', u'aggregateRating', u'offers', u'description', u'review', u'review']

current scope: [u'http://schema.org/AggregateRating']
    properties: [u'ratingValue', u'reviewCount']

current scope: [u'http://schema.org/Offer']
    properties: [u'price', u'availability']

current scope: [u'http://schema.org/Review']
    properties: [u'name', u'author', u'datePublished', u'reviewRating', u'description']

current scope: [u'http://schema.org/Rating']
    properties: [u'worstRating', u'ratingValue', u'bestRating']
```

```
current scope: [u'http://schema.org/Review']
   properties: [u'name', u'author', u'datePublished', u'reviewRating', u'description']

current scope: [u'http://schema.org/Rating']
   properties: [u'worstRating', u'ratingValue', u'bestRating']

>>>
```

在这里，我们首先在 `itemscope` 元素上迭代，对于其中的每一个元素，我们寻找所有的 `itemprops` 元素，并排除那些本身在另一个 `itemscope` 内的元素。

## 内建选择器的参考

class scrapy.selector.Selector(response=None, text=None, type=None)

Selector 的实例是对选择某些内容响应的封装。

response 是 HtmlResponse 或 XmlResponse 的一个对象，将被用来选择和提取数据。

text 是在 response 不可用时的一个 unicode 字符串或 utf-8 编码的文字。将 text 和 response 一起使用是未定义行为。

type 定义了选择器类型，可以是 "html"，"xml" or None (默认).

如果 type 是 None ，选择器会根据 response 类型(参见下面)自动选择最佳的类型，或者在和 text 一起使用时，默认为 "html" 。

如果 type 是 None ，并传递了一个 response ，选择器类型将从 response 类型中推导如下:

- "html" for HtmlResponse type
- "xml" for XmlResponse type
- "html" for anything else

其他情况下，如果设定了 type ，选择器类型将被强制设定，而不进行检测。

xpath(query)
寻找可以匹配 xpath query 的节点，并返回 `SelectorList` 的一个实例结果，单一化其所有元素。列表元素也实现了 `Selector` 的接口。

query 是包含 XPATH 查询请求的字符串。

> 注解
>
> 为了方便起见，该方法也可以通过 `response.xpath()` 调用

css(query)
应用给定的 CSS 选择器，返回 SelectorList 的一个实例。

`query` 是一个包含 CSS 选择器的字符串。

在后台，通过 cssselect 库和运行 .xpath() 方法，CSS 查询会被转换为 XPath 查询。

> 注解

> 为了方便起见，该方法也可以通过 response.css() 调用

extract()
串行化并将匹配到的节点返回一个 unicode 字符串列表。 结尾是编码内容的百分比。

re(regex)
应用给定的 regex，并返回匹配到的 unicode 字符串列表。

`regex` 可以是一个已编译的正则表达式，也可以是一个将被 `re.compile(regex)` 编译为正则表达式的字符串。

register_namespace(prefix, uri)
注册给定的命名空间，其将在 `Selector` 中使用。 不注册命名空间，你将无法从非标准命名空间中选择或提取数据。参见下面的例子。

remove_namespaces()
移除所有的命名空间，允许使用少量的命名空间 xpaths 遍历文档。参加下面的例子。

__nonzero__()
如果选择了任意的真实文档，将返回 True ，否则返回 False 。 也就是说， Selector 的布尔值是通过它选择的内容确定的。

## SelectorList 对象

### class scrapy.selector.SelectorList

`SelectorList` 类是内建 list 类的子类，提供了一些额外的方法。

xpath(query)
对列表中的每个元素调用.xpath()方法，返回结果为另一个单一化的 SelectorList。

query 和 `Selector.xpath()` 中的参数相同。

css(query)
对列表中的各个元素调用.css() 方法，返回结果为另一个单一化的 SelectorList。

query 和 Selector.css() 中的参数相同。

extract()
对列表中的各个元素调用.extract()方法，返回结果为单一化的 unicode 字符串列表。

re()
对列表中的各个元素调用 .re() 方法，返回结果为单一化的 unicode 字符串列表。

__nonzero__()
列表非空则返回 True，否则返回 False。

## 在 HTML 响应上的选择器样例

这里是一些 Selector 的样例，用来说明一些概念。在所有的例子中，我们假设已经有一个通过 HtmlResponse 对象实例化的 Selector ，如下:

```
sel = Selector(html_response)
```

1. 从 HTML 响应主体中提取所有的<h1>元素，返回:class:Selector 对象(即 SelectorList 的一个对象)的列表:

```
sel.xpath("//h1")
```

1. 从 HTML 响应主体上提取所有<h1>元素的文字，返回一个 unicode 字符串的列表:

```
sel.xpath("//h1").extract()         # this includes the h1 tag
sel.xpath("//h1/text()").extract()  # this excludes the h1 tag
```

1. 在所有<p>标签上迭代，打印它们的类属性:

```
for node in sel.xpath("//p"):
    print node.xpath("@class").extract()
```

## 在 XML 响应上的选择器样例

这里是一些样例，用来说明一些概念。在两个例子中，我们假设已经有一个通过 XmlResponse 对象实例化的 Selector ，如下:

```
sel = Selector(xml_response)
```

1. 从 XML 响应主体中选择所有的 元素，返回 Selector 对象(即 SelectorList 对象)的列表:

```
sel.xpath("//product")
```

1. 从 Google Base XML feed 中提取所有的价钱，这需要注册一个命名空间:

```
sel.register_namespace("g", "http://base.google.com/ns/1.0")
sel.xpath("//g:price").extract()
```

## 移除命名空间

在处理爬虫项目时，完全去掉命名空间而仅仅处理元素名字，写更多简单/实用的 XPath 会方便很多。你可以为此使用 `Selector.remove_namespaces()` 方法。

让我们来看一个例子，以 Github 博客的 atom 订阅来解释这个情况。

首先，我们使用想爬取的 url 来打开 shell:

```
$ scrapy shell https://github.com/blog.atom
```

一旦进入 shell，我们可以尝试选择所有的 <link> 对象，可以看到没有结果(因为 Atom XML 命名空间混淆了这些节点):

```
>>> response.xpath("//link")
[]
```

但一旦我们调用 `Selector.remove_namespaces()` 方法，所有的节点都可以直接通过他们的名字来访问:

```
>>> response.selector.remove_namespaces()
>>> response.xpath("//link")
[<Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 ...
```

如果你对为什么命名空间移除操作并不总是被调用，而需要手动调用有疑惑。这是因为存在如下两个原因，按照相关顺序如下:

1. 移除命名空间需要迭代并修改文件的所有节点，而这对于 Scrapy 爬取的所有文档操作需要一定的性能消耗
2. 会存在这样的情况，确实需要使用命名空间，但有些元素的名字与命名空间冲突。尽管这些情况非常少见。

# 9

# Item Loaders

Item Loaders 提供了一种便捷的方式填充抓取到的: `Items` 。虽然 Items 可以使用自带的类字典形式 API 填充，但是 Items Loaders 提供了更便捷的 API，可以分析原始数据并对 Item 进行赋值。

从另一方面来说，Items 提供保存抓取数据的容器，而 Item Loaders 提供的是 填充 容器的机制。

Item Loaders 提供的是一种灵活，高效的机制，可以更方便的被 spider 或 source format (HTML，XML，etc)扩展，并 override 更易于维护的、不同的内容分析规则。

## Using Item Loaders to populate items

要使用 Item Loader, 你必须先将它实例化。你可以使用类似字典的对象(例如: Item or dict)来进行实例化，或者不使用对象也可以，当不用对象进行实例化的时候，Item 会自动使用 `ItemLoader.default\_item_class` 属性中指定的 Item 类在 Item Loader constructor 中实例化。

然后,你开始收集数值到 Item Loader 时，通常使用 Selectors。你可以在同一个 item field 里面添加多个数值；Item Loader 将知道如何用合适的处理函数来“添加”这些数值。

下面是在 `Spider` 中典型的 Item Loader 的用法，使用 `Items chapter` 中声明的 Product item:

```
from scrapy.contrib.loader import ItemLoader
from myproject.items import Product

def parse(self, response):
    l = ItemLoader(item=Product(), response=response)
    l.add_xpath('name', '//div[@class="product_name"]')
    l.add_xpath('name', '//div[@class="product_title"]')
    l.add_xpath('price', '//p[@id="price"]')
    l.add_css('stock', 'p#stock]')
    l.add_value('last_updated', 'today') # you can also use literal values
    return l.load_item()
```

快速查看这些代码之后,我们可以看到发现 name 字段被从页面中两个不同的 XPath 位置提取:

1. `//div[@class="product_name"]`
2. `//div[@class="product_title"]`

换言之,数据通过用 `add_xpath()` 的方法,把从两个不同的 XPath 位置提取的数据收集起来. 这是将在以后分配给 `name` 字段中的数据?

之后,类似的请求被用于 price 和 stock 字段 (后者使用 CSS selector 和 `add_css()` 方法)， 最后使用不同的方法 `add_value()` 对 `last_update` 填充文本值( `today` )。

最终，当所有数据被收集起来之后, 调用 ItemLoader.load_item() 方法，实际上填充并且返回了之前通过调用 add_xpath()，add_css()，and add_value()所提取和收集到的数据的 Item。

## Input and Output processors

Item Loader 在每个(Item)字段中都包含了一个输入处理器和一个输出处理器? 输入处理器收到数据时立刻提取数据 (通过 add_xpath()，add_css()或者 add_value()方法)之后输入处理器的结果被收集起来并且保存在 ItemLoader 内. 收集到所有的数据后, 调用 ItemLoader.load_item()方法来填充,并得到填充后的 Item 对象。这是当输出处理器被和之前收集到的数据(和用输入处理器处理的)被调用.输出处理器的结果是被分配到 Item 的最终值?

让我们看一个例子来说明如何输入和输出处理器被一个特定的字段调用(同样适用于其他 field)::

```
l = ItemLoader(Product(), some_selector)
l.add_xpath('name', xpath1) # (1)
l.add_xpath('name', xpath2) # (2)
l.add_css('name', css) # (3)
l.add_value('name', 'test') # (4)
return l.load_item() # (5)
```

发生了这些事情:

1. 从 xpath1 提取出的数据,传递给 输入处理器 的 name 字段。输入处理器的结果被收集和保存在 Item Loader 中(但尚未分配给该 Item)?
2. 从 xpath2 提取出来的数据,传递给(1)中使用的相同的 输入处理器。输入处理器的结果被附加到在(1)中收集的数据(如果有的话)?
3. 和之前相似，只不过这里的数据是通过 css CSS selector 抽取，之后传输到在(1)和(2)使用 的 input processor 中。最终输入处理器的结果被附加到在(1)和(2)中收集的数据之后 (如果存在数据的话)。
4. 这里的处理方式也和之前相似，但是此处的值是通过 add_value 直接赋予的， 而不是利用 XPath 表达式或 CSS selector 获取。得到的值仍然是被传送到输入处理器。 在这里例程中，因为得到的值并非可迭代，所以在传输到输入处理器之前需要将其 转化为可迭代的单个元素，这才是它所接受的形式。
5. 在之前步骤中所收集到的数据被传送到 output processor 的 name field 中。输出处理器的结果就是赋到 item 中 name field 的值。

需要注意的是，输入和输出处理器都是可调用对象，调用时传入需要被分析的数据， 处理后返回分析得到的值。因此你可以使用任意函数作为输入、输出处理器。唯一需注意的是它们必须接收一个（并且只是一个）迭代器性质的 positional 参数。

10

# Scrapy 终端(Scrapy shell)

Scrapy 终端是一个交互终端，供您在未启动 spider 的情况下尝试及调试您的爬取代码。 其本意是用来测试提取数据的代码，不过您可以将其作为正常的 Python 终端，在上面测试任何的 Python 代码。

该终端是用来测试 XPath 或 CSS 表达式，查看他们的工作方式及从爬取的网页中提取的数据。 在编写您的 spider 时，该终端提供了交互性测试您的表达式代码的功能，免去了每次修改后运行 spider 的麻烦。

一旦熟悉了 Scrapy 终端后，您会发现其在开发和调试 spider 时发挥的巨大作用。

如果您安装了 IPython，Scrapy 终端将使用 IPython (替代标准 Python 终端)。 IPython 终端与其他相比更为强大，提供智能的自动补全，高亮输出，及其他特性。

我们强烈推荐您安装 IPython，特别是如果您使用 Unix 系统(IPython 在 Unix 下工作的很好)。 详情请参考 IPython installation guide 。

## 启动终端

您可以使用 shell 来启动 Scrapy 终端:

```
scrapy shell <url>
```

<url> 是您要爬取的网页的地址。

# 使用终端

Scrapy 终端仅仅是一个普通的 Python 终端(或 IPython)。其提供了一些额外的快捷方式。

## 可用的快捷命令(shortcut)

- `shelp()` – 打印可用对象及快捷命令的帮助列表
- `fetch(request_or_url)` – 根据给定的请求(request)或 URL 获取一个新的 response，并更新相关的对象
- `view(response)` – 在本机的浏览器打开给定的 response。 其会在 response 的 body 中添加一个 tag ，使得外部链接(例如图片及 css)能正确显示。 注意，该操作会在本地创建一个临时文件，且该文件不会被自动删除。

## 可用的 Scrapy 对象

Scrapy 终端根据下载的页面会自动创建一些方便使用的对象，例如 Response 对象及 Selector 对象(对 HTML 及 XML 内容)。

这些对象有:

- crawler – 当前 Crawler 对象。
- spider – 处理 URL 的 spider。对当前 URL 没有处理的 Spider 时则为一个 Spider 对象。
- request – 最近获取到的页面的 Request 对象。您可以使用 replace() 修改该 request。或者 使用 fetch 快捷方式来获取新的 request。
- response – 包含最近获取到的页面的 Response 对象。
- sel – 根据最近获取到的 response 构建的 Selector 对象。
- settings – 当前的 Scrapy settings

## 终端会话(shell session)样例

下面给出一个典型的终端会话的例子。 在该例子中，我们首先爬取了 http://scarpy.org 的页面，而后接着爬取 http://slashdot.org 的页面。 最后，我们修改了(Slashdot)的请求，将请求设置为 POST 并重新获取， 得到 HTTP 405(不允许的方法)错误。 之后通过 Ctrl-D(Unix)或 Ctrl-Z(Windows)关闭会话。

需要注意的是，由于爬取的页面不是静态页，内容会随着时间而修改， 因此例子中提取到的数据可能与您尝试的结果不同。 该例子的唯一目的是让您熟悉 Scrapy 终端。

首先，我们启动终端:

```
scrapy shell 'http://scrapy.org' --nolog
```

接着该终端(使用 Scrapy 下载器(downloader))获取 URL 内容并打印可用的对象及快捷命令(注意到以 `[s]` 开头的行):

```
[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0x1e16b50>
[s]   item       {}
[s]   request    <GET http://scrapy.org>
[s]   response   <200 http://scrapy.org>
[s]   sel        <Selector xpath=None data=u'<html>\n  <head>\n    <meta charset="utf-8'>
[s]   settings   <scrapy.settings.Settings object at 0x2bfd650>
[s]   spider     <Spider 'default' at 0x20c6f50>
[s] Useful shortcuts:
[s]   shelp()           Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local objects
[s]   view(response)    View response in a browser

>>>
```

之后，您就可以操作这些对象了:

```
>>> sel.xpath("//h2/text()").extract()[0]
u'Welcome to Scrapy'

>>> fetch("http://slashdot.org")
[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0x1a13b50>
[s]   item       {}
[s]   request    <GET http://slashdot.org>
[s]   response   <200 http://slashdot.org>
```

```
[s]   sel       <Selector xpath=None data=u'<html lang="en">\n<head>\n\n\n\n\n<script id="'>
[s]   settings   <scrapy.settings.Settings object at 0x2bfd650>
[s]   spider     <Spider 'default' at 0x20c6f50>
[s] Useful shortcuts:
[s]   shelp()          Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local objects
[s]   view(response)    View response in a browser

>>> sel.xpath('//title/text()').extract()
[u'Slashdot: News for nerds, stuff that matters']

>>> request = request.replace(method="POST")

>>> fetch(request)
[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0x1e16b50>
...

>>>
```

## 在 spider 中启动 shell 来查看 response

有时您想在 spider 的某个位置中查看被处理的 response， 以确认您期望的 response 到达特定位置。

这可以通过 `scrapy.shell.inspect_response` 函数来实现。

以下是如何在 spider 中调用该函数的例子:

```
import scrapy

class MySpider(scrapy.Spider):
    name = "myspider"
    start_urls = [
        "http://example.com",
        "http://example.org",
        "http://example.net",
    ]

    def parse(self, response):
        # We want to inspect one specific response.
        if ".org" in response.url:
            from scrapy.shell import inspect_response
            inspect_response(response, self)

        # Rest of parsing code.
```

当运行 spider 时，您将得到类似下列的输出:

```
2014-01-23 17:48:31-0400 [myspider] DEBUG: Crawled (200) <GET http://example.com> (referer: None)
2014-01-23 17:48:31-0400 [myspider] DEBUG: Crawled (200) <GET http://example.org> (referer: None)
[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0x1e16b50>
...

>>> response.url
'http://example.org'
```

接着测试提取代码:

```
>>> sel.xpath('//h1[@class="fn"]')
[]
```

呃，看来是没有。您可以在浏览器里查看 response 的结果，判断是否是您期望的结果:

```
>>> view(response)
True
```

最后您可以点击 Ctrl-D(Windows 下 Ctrl-Z)来退出终端，恢复爬取:

```
>>> ^D
2014-01-23 17:50:03-0400 [myspider] DEBUG: Crawled (200) <GET http://example.net> (referer: None)
...
```

注意: 由于该终端屏蔽了 Scrapy 引擎，您在这个终端中不能使用 `fetch` 快捷命令(shortcut)。 当您离开终端时，spider 会从其停下的地方恢复爬取，正如上面显示的那样。

# Item Pipeline

当 Item 在 Spider 中被收集之后，它将会被传递到 Item Pipeline，一些组件会按照一定的顺序执行对 Item 的处理。

每个 item pipeline 组件(有时称之为“Item Pipeline”)是实现了简单方法的 Python 类。他们接收到 Item 并通过它执行一些行为，同时也决定此 Item 是否继续通过 pipeline，或是被丢弃而不再进行处理。

以下是 item pipeline 的一些典型应用:

- 清理 HTML 数据
- 验证爬取的数据(检查 item 包含某些字段)
- 查重(并丢弃)
- 将爬取结果保存到数据库中

## 编写你自己的 item pipeline

编写你自己的 item pipeline 很简单，每个 item pipeline 组件是一个独立的 Python 类，同时必须实现以下方法:

process_item(self, item, spider)

每个 item pipeline 组件都需要调用该方法，这个方法必须返回一个 Item (或任何继承类)对象， 或是抛出 DropItem 异常，被丢弃的 item 将不会被之后的 pipeline 组件所处理。

**参数: **
– item (Item 对象) – 被爬取的 item – spider (Spider 对象) – 爬取该 item 的 spider

此外，他们也可以实现以下方法:

open_spider(self, spider)

当 spider 被开启时，这个方法被调用。

参数:

- spider (Spider 对象) – 被开启的 spider

close_spider(spider)

当 spider 被关闭时，这个方法被调用

参数:

spider (Spider 对象) – 被关闭的 spider

from_crawler(cls, crawler)

If present, this classmethod is called to create a pipeline instance from a Crawler. It must return a new instance of the pipeline. Crawler object provides access to all Scrapy core components like settings and signals; it is a way for pipeline to access them and hook its functionality into Scrapy.

参数:

crawler (Crawler object) – crawler that uses this pipeline

# Item pipeline 样例

## 验证价格，同时丢弃没有价格的 item

让我们来看一下以下这个假设的 pipeline，它为那些不含税( price_excludes_vat 属性)的 item 调整了 price 属性，同时丢弃了那些没有价格的 item:

```
from scrapy.exceptions import DropItem

class PricePipeline(object):

    vat_factor = 1.15

    def process_item(self, item, spider):
        if item['price']:
            if item['price_excludes_vat']:
                item['price'] = item['price'] * self.vat_factor
            return item
        else:
            raise DropItem("Missing price in %s" % item)
```

## 将 item 写入 JSON 文件

以下 pipeline 将所有(从所有 spider 中)爬取到的 item，存储到一个独立地 items.jl 文件，每行包含一个序列化为 JSON 格式的 item:

```
import json

class JsonWriterPipeline(object):

    def __init__(self):
        self.file = open('items.jl', 'wb')

    def process_item(self, item, spider):
        line = json.dumps(dict(item)) + "\n"
        self.file.write(line)
        return item
```

> 注解

> JsonWriterPipeline 的目的只是为了介绍怎样编写 item pipeline，如果你想要将所有爬取的 item 都保存到同一个 JSON 文件， 你需要使用 Feed exports 。

## Write items to MongoDB

In this example we’ll write items to MongoDB using pymongo. MongoDB address and database name are specified in Scrapy settings; MongoDB collection is named after item class.

The main point of this example is to show how to use from_crawler() method and how to clean up the resources properly.

> 注解
>
> Previous example (JsonWriterPipeline) doesn’t clean up resources properly. Fixing it is left as an exercise for the reader. import pymongo

```
class MongoPipeline(object):

    def __init__(self, mongo_uri, mongo_db):
        self.mongo_uri = mongo_uri
        self.mongo_db = mongo_db

    @classmethod
    def from_crawler(cls, crawler):
        return cls(
            mongo_uri=crawler.settings.get('MONGO_URI'),
            mongo_db=crawler.settings.get('MONGO_DATABASE', 'items')
        )

    def open_spider(self, spider):
        self.client = pymongo.MongoClient(self.mongo_uri)
        self.db = self.client[self.mongo_db]

    def close_spider(self, spider):
        self.client.close()

    def process_item(self, item, spider):
        collection_name = item.__class__.__name__
        self.db[collection_name].insert(dict(item))
        return item
```

## 去重

一个用于去重的过滤器，丢弃那些已经被处理过的 item。让我们假设我们的 item 有一个唯一的 id，但是我们 spider 返回的多个 item 中包含有相同的 id:

```
from scrapy.exceptions import DropItem

class DuplicatesPipeline(object):

    def __init__(self):
        self.ids_seen = set()

    def process_item(self, item, spider):
        if item['id'] in self.ids_seen:
            raise DropItem("Duplicate item found: %s" % item)
        else:
            self.ids_seen.add(item['id'])
            return item
```

## 启用一个 Item Pipeline 组件

为了启用一个 Item Pipeline 组件，你必须将它的类添加到 ITEM_PIPELINES 配置，就像下面这个例子:

```
ITEM_PIPELINES = {
    'myproject.pipelines.PricePipeline': 300,
    'myproject.pipelines.JsonWriterPipeline': 800,
}
```

分配给每个类的整型值，确定了他们运行的顺序，item 按数字从低到高的顺序，通过 pipeline，通常将这些数字定义在 0-1000 范围内。

# Feed exports

新版功能。

实现爬虫时最经常提到的需求就是能合适的保存爬取到的数据，或者说，生成一个带有爬取数据的”输出文件”(通常叫做”输出 feed”)，来供其他系统使用。

Scrapy 自带了 Feed 输出，并且支持多种序列化格式(serialization format)及存储方式(storage backends)。

# 序列化方式(Serialization formats)

feed 输出使用到了 Item exporters 。其自带支持的类型有:

- JSON
- JSON lines
- CSV
- XML

您也可以通过 FEED_EXPORTERS 设置扩展支持的属性。

## JSON

- FEED_FORMAT: json
- 使用的 exporter: JsonItemExporter
- 大数据量情况下使用 JSON 请参见 这个警告

## JSON lines

- FEED_FORMAT: jsonlines
- 使用的 exporter: JsonLinesItemExporter

## CSV

- FEED_FORMAT: csv
- 使用的 exporter: CsvItemExporter

## XML

- FEED_FORMAT: xml
- 使用的 exporter: XmlItemExporter

## Pickle

- FEED_FORMAT: pickle
- 使用的 exporter: PickleItemExporter

## Marshal

- FEED_FORMAT: marshal
- 使用的 exporter: MarshalItemExporter

## 存储(Storages)

使用 feed 输出时您可以通过使用 URI(通过 FEED_URI 设置) 来定义存储端。feed 输出支持 URI 方式支持的多种存储后端类型。

自带支持的存储后端有:

- 本地文件系统
- FTP
- S3 (需要 boto)
- 标准输出

有些存储后端会因所需的外部库未安装而不可用。例如，S3 只有在 boto 库安装的情况下才可使用。

## 存储 URI 参数

存储 URI 也包含参数。当 feed 被创建时这些参数可以被覆盖:

- `%(time)s` - 当 feed 被创建时被 timestamp 覆盖
- `%(name)s` - 被 spider 的名字覆盖

其他命名的参数会被 spider 同名的属性所覆盖。例如， 当 feed 被创建时， `%(site_id)s` 将会被 `spider.site_id` 属性所覆盖。

下面用一些例子来说明:

- 存储在 FTP，每个 spider 一个目录:
  - `ftp://user:password@ftp.example.com/scraping/feeds/%(name)s/%(time)s.json`
- 存储在 S3，每一个 spider 一个目录:
  - `s3://mybucket/scraping/feeds/%(name)s/%(time)s.json`

# 存储端(Storage backends)

## 本地文件系统

将 feed 存储在本地系统。

- URI scheme: `file`
- URI 样例: `file:///tmp/export.csv`
- 需要的外部依赖库: `none`

注意: (只有)存储在本地文件系统时，您可以指定一个绝对路径 /tmp/export.csv 并忽略协议(scheme)。不过这仅仅只能在 Unix 系统中工作。

## FTP

将 feed 存储在 FTP 服务器。

- URI scheme: `ftp`
- URI 样例: `ftp://user:pass@ftp.example.com/path/to/export.csv`
- 需要的外部依赖库: `none`

## S3

将 feed 存储在 Amazon S3 。

- URI scheme: s3
- URI 样例:
  - s3://mybucket/path/to/export.csv
  - s3://aws_key:aws_secret@mybucket/path/to/export.csv
- 需要的外部依赖库: boto

您可以通过在 URI 中传递 user/pass 来完成 AWS 认证，或者也可以通过下列的设置来完成:

AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY

## 标准输出

feed 输出到 Scrapy 进程的标准输出。

- URI scheme: stdout
- URI 样例: stdout:
- 需要的外部依赖库: none

## 设定(Settings)

这些是配置 feed 输出的设定:

- FEED_URI (必须)
- FEED_FORMAT
- FEED_STORAGES
- FEED_EXPORTERS
- FEED_STORE_EMPTY

### FEED_URI

Default: `None`

输出 feed 的 URI。支持的 URI 协议请参见 `存储端(Storage backends)` 。

为了启用 feed 输出，该设定是必须的。

### FEED_FORMAT

输出 feed 的序列化格式。可用的值请参见 `序列化方式(Serialization formats)` 。

### FEED_STORE_EMPTY

Default: `False`

是否输出空 feed(没有 item 的 feed)。

### FEED_STORAGES

Default:: `{}`

包含项目支持的额外 feed 存储端的字典。 字典的键(key)是 URI 协议(scheme)，值是存储类(storage clas
s)的路径。

## FEED_STORAGES_BASE

Default:

```
{
    '': 'scrapy.contrib.feedexport.FileFeedStorage',
    'file': 'scrapy.contrib.feedexport.FileFeedStorage',
    'stdout': 'scrapy.contrib.feedexport.StdoutFeedStorage',
    's3': 'scrapy.contrib.feedexport.S3FeedStorage',
    'ftp': 'scrapy.contrib.feedexport.FTPFeedStorage',
}
```

包含 Scrapy 内置支持的 feed 存储端的字典。

## FEED_EXPORTERS

Default:: `{}`

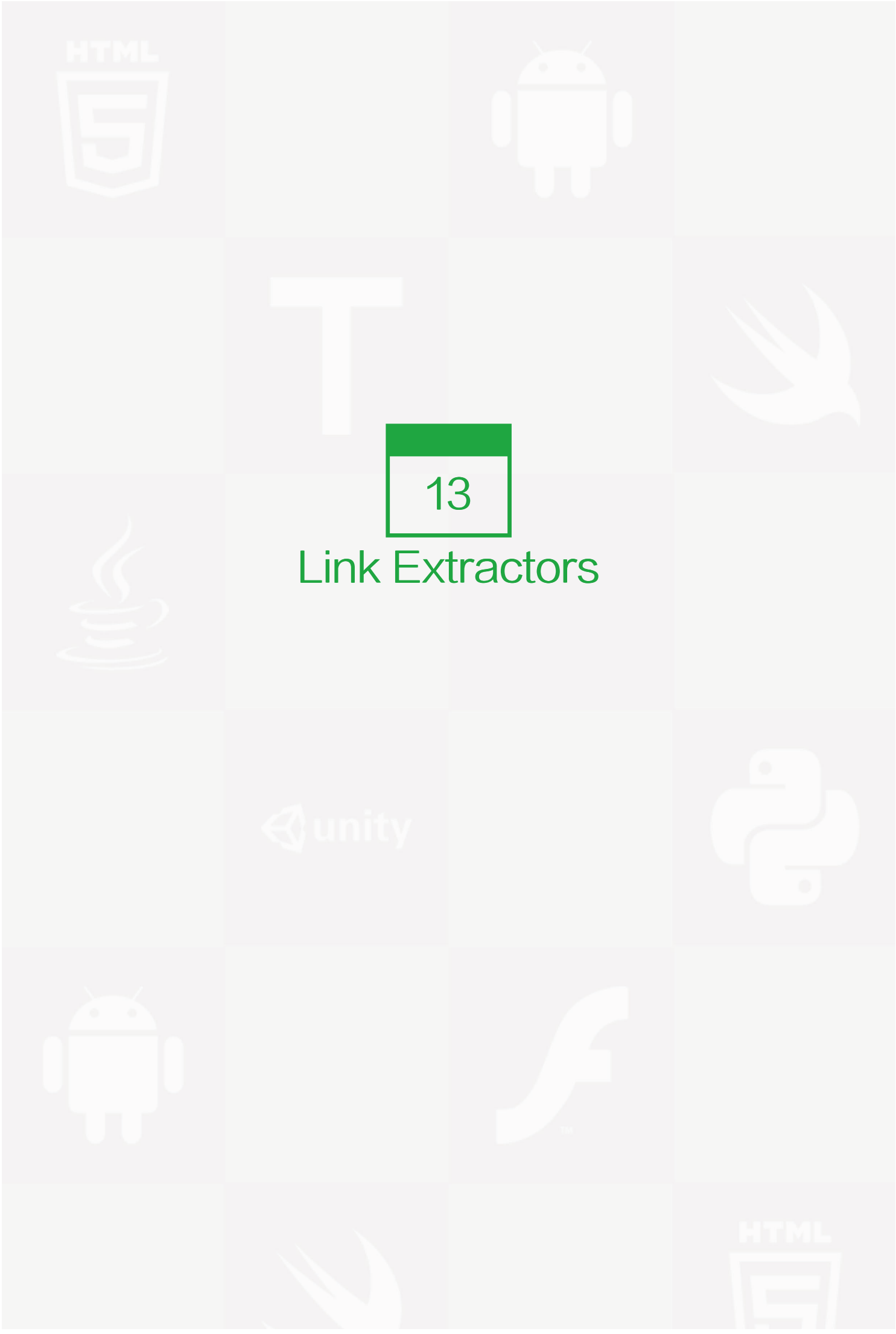
包含项目支持的额外输出器(exporter)的字典。 该字典的键(key)是 URI 协议(scheme)，值是 Item 输出器(exporter) 类的路径。

## FEED_EXPORTERS_BASE

Default:

```
FEED_EXPORTERS_BASE = {
    'json': 'scrapy.contrib.exporter.JsonItemExporter',
    'jsonlines': 'scrapy.contrib.exporter.JsonLinesItemExporter',
    'csv': 'scrapy.contrib.exporter.CsvItemExporter',
    'xml': 'scrapy.contrib.exporter.XmlItemExporter',
    'marshal': 'scrapy.contrib.exporter.MarshalItemExporter',
}
```

包含 Scrapy 内置支持的 feed 输出器(exporter)的字典。



# 13

# Link Extractors

Link Extractors 是用于从网页( scrapy.http.Response )中抽取会被 follow 的链接的对象。

Scrapy 默认提供 2 种可用的 Link Extractor, 但你通过实现一个简单的接口创建自己定制的 Link Extractor 来满足需求? Scrapy 提供了 scrapy.contrib.linkextractors import LinkExtractor ， 不过您也可以通过实现一个简单的接口来创建您自己的 Link Extractor，满足需求。

每个 LinkExtractor 有唯一的公共方法是 extract_links ，其接收 一个 Response 对象， 并返回 scrapy.link.Link 对象? Link Extractors 只实例化一次，其 extract_links 方法会根据不同的 response 被调用多次来提取链接?

Link Extractors 在 CrawlSpider 类(在 Scrapy 可用)中使用, 通过一套规则,但你也可以用它在你的 Spider 中,即使你不是从 CrawlSpider 继承的子类, 因为它的目的很简单: 提取链接?

# 内置 Link Extractor 参考

Scrapy 自带的 Link Extractors 类在 scrapy.contrib.linkextractors 模块提供?

默认的 link extractor 是 LinkExtractor ，其实就是 LxmlLinkExtractor:

```
from scrapy.contrib.linkextractors import LinkExtractor
```

在以前版本的 Scrapy 版本中提供了其他的 link extractor，不过都已经被废弃了。

## LxmlLinkExtractor

class scrapy.contrib.linkextractors.lxmlhtml.LxmlLinkExtractor(allow=(), deny=(), allow_domains=(), deny_domains=(), deny_extensions=None, restrict_xpaths=(), tags=('a', 'area'), attrs=('href', ), canonicalize=True, unique=True, process_value=None)

LxmlLinkExtractor is the recommended link extractor with handy filtering options. It is implemented using lxml’s robust HTMLParser.

**参数:**

- allow (a regular expression (or list of)) – a single regular expression (or list of regular expressions) that the (absolute) urls must match in order to be extracted. If not given (or empty), it will match all links.
- deny (a regular expression (or list of)) – a single regular expression (or list of regular expressions) that the (absolute) urls must match in order to be excluded (ie. not extracted). It has precedence over the allow parameter. If not given (or empty) it won’t exclude any links.
- allow_domains (str or list) – a single value or a list of string containing domains which will be considered for extracting the links
- deny_domains (str or list) – a single value or a list of strings containing domains which won’t be considered for extracting the links
- deny_extensions (list) – a single value or list of strings containing extensions that should be ignored when extracting links. If not given, it will default to the IGNORED_EXTENSIONS list defined in the scrapy.linkextractor module.

- restrict_xpaths (str or list) – is a XPath (or list of XPath’s) which defines regions inside the response where links should be extracted from. If given, only the text selected by those XPath will be scanned for links. See examples below.
- tags (str or list) – a tag or a list of tags to consider when extracting links. Defaults to ('a', 'area').
- attrs (list) – an attribute or list of attributes which should be considered when looking for links to extract (only for those tags specified in the tags parameter). Defaults to ('href',)
- canonicalize (boolean) – canonicalize each extracted url (using scrapy.utils.url.canonicalize_url). Defaults to True.
- unique (boolean) – whether duplicate filtering should be applied to extracted links.
- process_value (callable) –

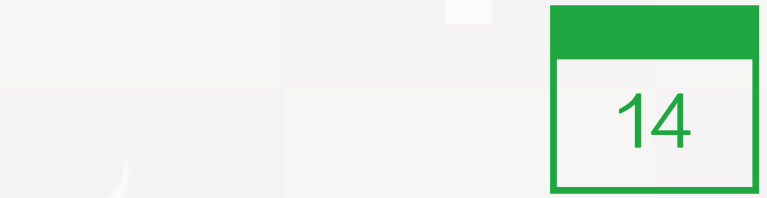
它接收来自扫描标签和属性提取每个值, 可以修改该值, 并返回一个新的，或返回 None 完全忽略链接的功能。如果没有给出，process_value 默认是 lambda x: x。

例如，从这段代码中提取链接:

```
<a href="javascript:goToPage('../other/page.html'); return false">Link text</a>
```

你可以使用下面的这个 `process_value` 函数:

```
def process_value(value):
    m = re.search("javascript:goToPage\('(.*?)'", value)
    if m:
        return m.group(1)
```



# 14
# Logging

crapy 提供了 log 功能。您可以通过 `scrapy.log` 模块使用。当前底层实现使用了 Twisted logging，不过可能在之后会有所变化。

log 服务必须通过显示调用 `scrapy.log.start()` 来开启，以捕捉顶层的 Scrapy 日志消息。 在此之上，每个 crawler 都拥有独立的 log 观察者(observer)(创建时自动连接(attach)),接收其 spider 的日志消息。

## Log levels

Scrapy 提供 5 层 logging 级别:

- CRITICAL - 严重错误(critical)
- ERROR - 一般错误(regular errors)
- WARNING - 警告信息(warning messages)
- INFO - 一般信息(informational messages)
- DEBUG - 调试信息(debugging messages)

## 如何设置 log 级别

您可以通过终端选项(command line option) –loglevel/-L 或 `LOG_LEVEL` 来设置 log 级别。

## 如何记录信息(log messages)

下面给出如何使用 `WARNING` 级别来记录信息的例子:

```
from scrapy import log
log.msg("This is a warning", level=log.WARNING)
```

## 在 Spider 中添加 log(Logging from Spiders)

在 spider 中添加 log 的推荐方式是使用 Spider 的 `log()` 方法。该方法会自动在调用 `scrapy.log.msg()` 时赋值 spider 参数。其他的参数则直接传递给 `msg()` 方法。

## scrapy.log 模块

scrapy.log.start(logfile=None, loglevel=None, logstdout=None)

启动 Scrapy 顶层 logger。该方法必须在记录任何顶层消息前被调用 (使用模块的 msg() 而不是 Spider.log 的消息)。否则，之前的消息将会丢失。

参数:

- logfile (str) – 用于保存 log 输出的文件路径。如果被忽略，LOG_FILE 设置会被使用。如果两个参数都是 None，log 将会被输出到标准错误流(standard error)。
- loglevel – 记录的最低的 log 级别。可用的值有: CRITICAL，ERROR，WARNING，INFO and DEBUG。
- logstdout (boolean) – 如果为 True，所有您的应用的标准输出(包括错误)将会被记录(logged instead)。例如，如果您调用 “print ‘hello’ ”，则’hello’会在 Scrapy 的 log 中被显示。 如果被忽略，则 LOG_STDOUT 设置会被使用。

scrapy.log.msg(message, level=INFO, spider=None)

记录信息(Log a message)

参数:
– message (str) – log 的信息 – level – 该信息的 log 级别. 参考 Log levels. – spider (Spider 对象) – 记录该信息的 spider. 当记录的信息和特定的 spider 有关联时，该参数必须被使用。

scrapy.log.CRITICAL

严重错误的 Log 级别

scrapy.log.ERROR

错误的 Log 级别 Log level for errors

scrapy.log.WARNING

警告的 Log 级别 Log level for warnings

scrapy.log.INFO

记录信息的 Log 级别(生产部署时推荐的 Log 级别)

scrapy.log.DEBUG

调试信息的 Log 级别(开发时推荐的 Log 级别)

## Logging 设置

以下设置可以被用来配置 logging:

- LOG_ENABLED
- LOG_ENCODING
- LOG_FILE
- LOG_LEVEL
- LOG_STDOUT

15

# 数据收集(Stats Collection)

Scrapy 提供了方便的收集数据的机制。数据以 key/value 方式存储，值大多是计数值。 该机制叫做数据收集器(Stats Collector)，可以通过 Crawler API 的属性 `stats` 来使用。在下面的章节常见数据收集器使用方法将给出例子来说明。

无论数据收集(stats collection)开启或者关闭，数据收集器永远都是可用的。因此您可以 import 进自己的模块并使用其 API(增加值或者设置新的状态键(stat keys))。该做法是为了简化数据收集的方法: 您不应该使用超过一行代码来收集您的 spider，Scrpay 扩展或任何您使用数据收集器代码里头的状态。

数据收集器的另一个特性是(在启用状态下)很高效，(在关闭情况下)非常高效(几乎察觉不到)。

数据收集器对每个 spider 保持一个状态表。当 spider 启动时，该表自动打开，当 spider 关闭时，自动关闭。

## 常见数据收集器使用方法

通过 stats 属性来使用数据收集器。 下面是在扩展中使用状态的例子:

```
class ExtensionThatAccessStats(object):

    def __init__(self, stats):
        self.stats = stats

    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler.stats)
```

设置数据:

```
stats.set_value('hostname', socket.gethostname())
```

增加数据值:

```
stats.inc_value('pages_crawled')
```

当新的值比原来的值大时设置数据:

```
stats.max_value('max_items_scraped', value)
```

当新的值比原来的值小时设置数据:

```
stats.min_value('min_free_memory_percent', value)
```

获取数据:

```
>>> stats.get_value('pages_crawled')
8
```

获取所有数据:

```
>>> stats.get_stats()
{'pages_crawled': 1238, 'start_time': datetime.datetime(2009, 7, 14, 21, 47, 28, 977139)}
```

## 可用的数据收集器

除了基本的 `StatsCollector`，Scrapy 也提供了基于 `StatsCollector` 的数据收集器。 您可以通过 `STATS_CLASS` 设置来选择。默认使用的是 `MemoryStatsCollector` 。

### MemoryStatsCollector

class scrapy.statscol.MemoryStatsCollector

一个简单的数据收集器。其在 spider 运行完毕后将其数据保存在内存中。数据可以通过 spider_stats 属性访问。该属性是一个以 spider 名字为键(key)的字典。

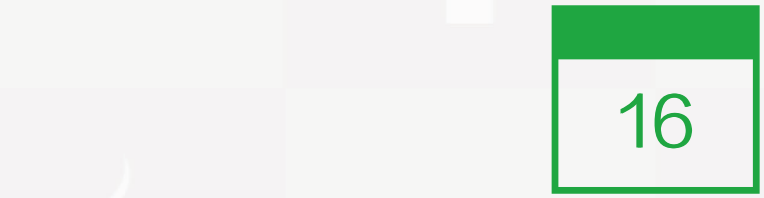
这是 Scrapy 的默认选择。

spider_stats
保存了每个 spider 最近一次爬取的状态的字典(dict)。该字典以 spider 名字为键，值也是字典。

### DummyStatsCollector

class scrapy.statscol.DummyStatsCollector

该数据收集器并不做任何事情但非常高效。您可以通过设置 STATS_CLASS 启用这个收集器，来关闭数据收集，提高效率。 不过，数据收集的性能负担相较于 Scrapy 其他的处理(例如分析页面)来说是非常小的。



# 16

# 发送 email

虽然 Python 通过 smtplib 库使得发送 email 变得很简单，Scrapy 仍然提供了自己的实现。 该功能十分易用，同时由于采用了 Twisted 非阻塞式(non-blocking)IO，其避免了对爬虫的非阻塞式 IO 的影响。 另外，其也提供了简单的 API 来发送附件。 通过一些 settings 设置，您可以很简单的进行配置。

## 简单例子

有两种方法可以创建邮件发送器(mail sender)。您可以通过标准构造器(constructor)创建:

```
from scrapy.mail import MailSender
mailer = MailSender()
```

或者您可以传递一个 Scrapy 设置对象，其会参考 settings:

```
mailer = MailSender.from_settings(settings)
```

这是如何来发送邮件了(不包括附件):

```
mailer.send(to=["someone@example.com"], subject="Some subject", body="Some body", cc=["another@example.com"])
```

## MailSender 类参考手册

在 Scrapy 中发送 email 推荐使用 MailSender。其同框架中其他的部分一样，使用了 Twisted 非阻塞式(non-blocking)IO。

class scrapy.mail.MailSender(smtphost=None, mailfrom=None, smtpuser=None, smtppass=None, smtpport=None)

参数:

- smtphost (str) – 发送 email 的 SMTP 主机(host)。如果忽略，则使用 MAIL_HOST 。
- mailfrom (str) – 用于发送 email 的地址(address)(填入 From:) 。如果忽略，则使用 MAIL_FROM 。
- smtpuser – SMTP 用户。如果忽略,则使用 MAIL_USER 。如果未给定，则将不会进行 SMTP 认证(authentication)。
- smtppass (str) – SMTP 认证的密码
- smtpport (int) – SMTP 连接的短裤
- smtptls – 强制使用 STARTTLS
- smtpssl (boolean) – 强制使用 SSL 连接

classmethod from_settings(settings)
使用 Scrapy 设置对象来初始化对象。其会参考 这些 Scrapy 设置 。

参数:

settings (scrapy.settings.Settings object) – the e-mail recipients

send(to, subject, body, cc=None, attachs=(), mimetype='text/plain')
发送 email 到给定的接收者。

参数:

- to (list) – email 接收
- subject (str) – email 内容
- cc (list) – 抄送的人
- body (str) – email 的内容

- attachs (iterable) – 可迭代的元组 (attach_name, mimetype, file_object) attach_name 是一个在 email 的附件中显示的名字的字符串，mimetype 是附件的 mime 类型， file_object 是包含附件内容的可读的文件对象。
- mimetype (str) – email 的 mime 类型

# Mail 设置

这些设置定义了 MailSender 构造器的默认值。其使得在您不编写任何一行代码的情况下，为您的项目配置实现 email 通知的功能。

## MAIL_FROM

默认值: `'scrapy@localhost'`

用于发送 email 的地址(address)(填入 `From` :) 。

## MAIL_HOST

默认值: `'localhost'`

发送 email 的 SMTP 主机(host)。

## MAIL_PORT

默认值: `25`

发用邮件的 SMTP 端口。

## MAIL_USER

默认值: `None`

SMTP 用户。如果未给定，则将不会进行 SMTP 认证(authentication)。

## MAIL_PASS

默认值: `None`

用于 SMTP 认证，与 `MAIL_USER` 配套的密码。

## MAIL_TLS

默认值: `False`

强制使用 STARTTLS。STARTTLS 能使得在已经存在的不安全连接上，通过使用 SSL/TLS 来实现安全连接。

## MAIL_SSL

默认值: `False`

强制使用 SSL 加密连接。

17

# Telnet 终端(Telnet Console)

Scrapy 提供了内置的 telnet 终端，以供检查，控制 Scrapy 运行的进程。 telnet 仅仅是一个运行在 Scrapy 进程中的普通 python 终端。因此您可以在其中做任何事。

telnet 终端是一个自带的 Scrapy 扩展。 该扩展默认为启用，不过您也可以关闭。 关于扩展的更多内容请参考Telnet console 扩展。

## 如何访问 telnet 终端

telnet 终端监听设置中定义的 `TELNETCONSOLE_PORT`，默认为 6023。访问 telnet 请输入：

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

Windows 及大多数 Linux 发行版都自带了所需的 telnet 程序。

## telnet 终端中可用的变量

telnet 仅仅是一个运行在 Scrapy 进程中的普通 python 终端。因此您可以做任何事情，甚至是导入新终端。

telnet 为了方便提供了一些默认定义的变量：

| 快捷名称 | 描述 |
| --- | --- |
| crawler | Scrapy Crawler ( scrapy.crawler.Crawler 对象) |
| engine | Crawler.engine属性 |
| spider | 当前激活的爬虫(spider) |
| slot | the engine slot |
| extensions | 扩展管理器(manager) (Crawler.extensions属性) |
| stats | 状态收集器 (Crawler.stats属性) |
| settings | Scrapy设置(setting)对象 (Crawler.settings属性) |
| est | 打印引擎状态的报告 |
| prefs | 针对内存调试 (参考调试内存溢出) |
| p | pprint.pprint 函数的简写 |
| hpy | 针对内存调试 (参考 调试内存溢出) |

# Telnet console usage examples

下面是使用 telnet 终端的一些例子:

## 查看引擎状态

在终端中您可以使用 Scrapy 引擎的 est()方法来快速查看状态:

```
telnet localhost 6023
>>> est()
Execution engine status

time()-engine.start_time                    : 8.62972998619
engine.has_capacity()                       : False
len(engine.downloader.active)               : 16
engine.scraper.is_idle()                    : False
engine.spider.name                          : followall
engine.spider_is_idle(engine.spider)        : False
engine.slot.closing                         : False
len(engine.slot.inprogress)                 : 16
len(engine.slot.scheduler.dqs or [])        : 0
len(engine.slot.scheduler.mqs)              : 92
len(engine.scraper.slot.queue)              : 0
len(engine.scraper.slot.active)             : 0
engine.scraper.slot.active_size             : 0
engine.scraper.slot.itemproc_size           : 0
engine.scraper.slot.needs_backout()         : False
```

## 暂停，恢复和停止 Scrapy 引擎

暂停:

```
telnet localhost 6023
>>> engine.pause()
>>>
```

恢复:

```
telnet localhost 6023
>>> engine.unpause()
>>>
```

停止：

```
telnet localhost 6023
>>> engine.stop()
Connection closed by foreign host.
```

## Telnet 终端信号

scrapy.telnet.update_telnet_vars(telnet_vars)

在 telnet 终端开启前发送该信号。您可以挂载(hook up)该信号来添加，移除或更新 telnet 本地命名空间可用的变量。您可以通过在您的处理函数(handler)中更新 telnet_vars 字典来实现该修改。

**参数:** telnet_vars (dict) – telnet 变量的字典

# Telnet 设定

以下是终端的一些设定：

## TELNETCONSOLE_PORT

Default: `[6023, 6073]`

telnet 终端使用的端口范围。如果设为 `None` 或 `0`， 则动态分配端口。

## TELNETCONSOLE_HOST

默认: `'127.0.0.1'`

telnet 终端监听的接口(interface)。

# 18

# Web Service

webserver 被移动到另外一个项目中。

托管在:

https://github.com/scrapy/scrapy-jsonrpc

# 19

# 常见问题(FAQ)

## Scrapy 相 BeautifulSoup 或 lxml 比较，如何呢?

BeautifulSoup 及 lxml 是 HTML 和 XML 的分析库。Scrapy 则是 编写爬虫，爬取网页并获取数据的应用框架(application framework)。

Scrapy 提供了内置的机制来提取数据(叫做 选择器(selectors))。 但如果您觉得使用更为方便，也可以使用 BeautifulSoup(或 lxml)。 总之，它们仅仅是分析库，可以在任何 Python 代码中被导入及使用。

换句话说，拿 Scrapy 与 BeautifulSoup (或 lxml) 比较就好像是拿 jinja2 与 Django 相比。

## Scrapy 支持那些 Python 版本?

Scrapy 仅仅支持 Python 2.7。Python2.6 的支持从 Scrapy 0.20 开始被废弃了。

## Scrapy 支持 Python 3 么?

不。但是 Python 3.3+的支持已经在计划中了。现在，Scrapy 支持 Python 2.7。

> 参见
>
> Scrapy 支持那些 Python 版本? 。

## Scrapy 是否从 Django 中”剽窃”了 X 呢?

也许吧，不过我们不喜欢这个词。我们认为 Django 是一个很好的开源项目，同时也是 一个很好的参考对象，所以我们把其作为 Scrapy 的启发对象。

我们坚信，如果有些事情已经做得很好了，那就没必要再重复制造轮子。这个想法，作为 开源项目及免费软件的基石之一，不仅仅针对软件，也包括文档，过程，政策等等。所以，与其自行解决每个问题，我们选择从其他已经很好地解决问题的项目中复制想法(copy idea)，并把注意力放在真正需要解决的问题上。

如果 Scrapy 能启发其他的项目，我们将为此而自豪。欢迎来抄(steal)我们!

## Scrapy 支持 HTTP 代理么?

是的。(从 Scrapy 0.8 开始)通过 HTTP 代理下载中间件对 HTTP 代理提供了支持。参考 `HttpProxyMiddleware` 。

## 如何爬取属性在不同页面的 item 呢?

参考 Passing additional data to callback functions。

## Scrapy 退出，ImportError: Nomodule named win32api

这是个 Twisted bug，您需要安装 pywin32。

## 我要如何在 spider 里模拟用户登录呢?

参考 使用 FormRequest.from_response()方法模拟用户登录。

## Scrapy 是以广度优先还是深度优先进行爬取的呢?

默认情况下，Scrapy 使用 LIFO 队列来存储等待的请求。简单的说，就是深度优先顺序。深度优先对大多数情况下是更方便的。如果您想以 广度优先顺序 进行爬取，你可以设置以下的设定:

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

## 我的 Scrapy 爬虫有内存泄露，怎么办?

参考 调试内存溢出 。

另外，Python 自己也有内存泄露，在 Leaks without leaks 有所描述。

## 如何让 Scrapy 减少内存消耗?

参考上一个问题。

## 我能在 spider 中使用基本 HTTP 认证么?

可以。参考 `HttpAuthMiddleware` .

## 为什么 Scrapy 下载了英文的页面，而不是我的本国语言?

尝试通过覆盖 `DEFAULT_REQUEST_HEADERS` 设置来修改默认的 `Accept-Language` 请求头。

## 我能在哪里找到 Scrapy 项目的例子?

参考 例子 。

## 我能在不创建 Scrapy 项目的情况下运行一个爬虫(spider)么?

是的。您可以使用 `runspider` 命令。例如，如果您有个 spider 写在 `my_spider.py` 文件中，您可以运行:

```
scrapy runspider my_spider.py
```

详情请参考 `runspider` 命令。

## 我收到了 “Filtered offsite request” 消息。如何修复?

这些消息(以 DEBUG 所记录)并不意味着有问题，所以你可以不修复它们。

这些消息由 Offsite Spider 中间件(Middleware)所抛出。 该(默认启用的)中间件筛选出了不属于当前 spider 的站点请求。

更多详情请参见: `OffsiteMiddleware` 。

## 发布 Scrapy 爬虫到生产环境的推荐方式?

参见 `Scrapyd` 。

## 我能对大数据(large exports)使用 JSON 么?

这取决于您的输出有多大。参考 JsonItemExporter 文档中的 这个警告 。

我能在信号处理器(signal handler)中返回(Twisted)引用么? 有些信号支持从处理器中返回引用，有些不行。参考 内置信号参考手册(Built-in signals reference) 来了解详情。

## reponse 返回的状态值 999 代表了什么?

999 是雅虎用来控制请求量所定义的返回值。 试着减慢爬取速度，将 spider 的下载延迟改为 2 或更高:

```
class MySpider(CrawlSpider):

    name = 'myspider'

    download_delay = 2

    # [ ... rest of the spider code ... ]
```

或在 `DOWNLOAD_DELAY` 中设置项目的全局下载延迟。

## 我能在 spider 中调用 pdb.set_trace() 来调试么?

可以，但你也可以使用 Scrapy 终端。这能让你快速分析(甚至修改) spider 处理返回的返回(response)。通常来说，比老旧的 pdb.set_trace() 有用多了。

更多详情请参考 在 spider 中启动 shell 来查看 response 。

## 将所有爬取到的 item 转存(dump)到 JSON/CSV/XML 文件的最简单的方法?

dump 到 JSON 文件:

```
scrapy crawl myspider -o items.json
```

dump 到 CSV 文件:

```
scrapy crawl myspider -o items.csv
```

dump 到 XML 文件:

```
scrapy crawl myspider -o items.xml
```

更多详情请参考 Feed exports

## 在某些表单中巨大神秘的 __VIEWSTATE 参数是什么?

__VIEWSTATE 参数存在于 ASP.NET/VB.NET 建立的站点中。关于这个参数的作用请参考这篇文章。这里有一个爬取这种站点的样例爬虫。

## 分析大 XML/CSV 数据源的最好方法是?

使用 XPath 选择器来分析大数据源可能会有问题。选择器需要在内存中对数据建立完整的 DOM 树，这过程速度很慢且消耗大量内存。

为了避免一次性读取整个数据源，您可以使用 `scrapy.utils.iterators` 中的 `xmliter` 及 `csviter` 方法。 实际上，这也是 feed spider(参考 `Spiders` )中的处理方法。

## Scrapy 自动管理 cookies 么?

是的，Scrapy 接收并保持服务器返回来的 cookies，在之后的请求会发送回去，就像正常的网页浏览器做的那样。

更多详情请参考 `Requests and Responses` 及 `CookiesMiddleware` 。

## 如何才能看到 Scrapy 发出及接收到的 Cookies 呢?

启用 `COOKIES_DEBUG` 选项。

## 要怎么停止爬虫呢?

在回调函数中 raise `CloseSpider` 异常。 更多详情请参见: `CloseSpider` 。

## 如何避免我的 Scrapy 机器人(bot)被禁止(ban)呢?

参考 避免被禁止(ban) 。

## 我应该使用 spider 参数(arguments)还是设置(settings)来配置 spider 呢?

spider 参数 及 设置(settings) 都可以用来配置您的 spider。没有什么强制的规则来限定要使用哪个，但设置(settings)更适合那些一旦设置就不怎么会修改的参数，而 spider 参数则意味着修改更为频繁，在每次 spider 运行都有修改，甚至是 spider 运行所必须的元素 (例如，设置 spider 的起始 url)。

这里以例子来说明这个问题。假设您有一个 spider 需要登录某个网站来 爬取数据，并且仅仅想爬取特定网站的特定部分(每次都不一定相同)。 在这个情况下，认证的信息将写在设置中，而爬取的特定部分的 url 将是 spider 参数。

## 我爬取了一个 XML 文档但是 XPath 选择器不返回任何的 item

也许您需要移除命名空间(namespace)。参见 移除命名空间 。

# 20

# 调试(Debugging)Spiders

本篇介绍了调试 spider 的常用技术。 考虑下面的 spider：

```
import scrapy
from myproject.items import MyItem

class MySpider(scrapy.Spider):
    name = 'myspider'
    start_urls = (
        'http://example.com/page1',
        'http://example.com/page2',
        )

    def parse(self, response):
        # collect `item_urls`
        for item_url in item_urls:
            yield scrapy.Request(item_url, self.parse_item)

    def parse_item(self, response):
        item = MyItem()
        # populate `item` fields
        # and extract item_details_url
        yield scrapy.Request(item_details_url, self.parse_details, meta={'item': item})

    def parse_details(self, response):
        item = response.meta['item']
        # populate more `item` fields
        return item
```

简单地说，该 spider 分析了两个包含 item 的页面(start_urls)。Item 有详情页面，所以我们使用 Request 的 meta 功能来传递已经部分获取的 item。

## Parse 命令

检查 spier 输出的最基本方法是使用 parse 命令。这能让你在函数层(method level)上检查 spider 各个部分的效果。其十分灵活并且易用，不过不能在代码中调试。

查看特定 url 爬取到的 item：

```
$ scrapy parse --spider=myspider -c parse_item -d 2 <item_url>
[ ... scrapy log lines crawling example.com spider ... ]

>>> STATUS DEPTH LEVEL 2 <<<
 # Scraped Items  ------------------------------------------------------------

[{'url': <item_url>}]

 # Requests  -----------------------------------------------------------------

[]
```

使用--verbose 或-v 选项，查看各个层次的状态：

```
$ scrapy parse --spider=myspider -c parse_item -d 2 -v <item_url>
[ ... scrapy log lines crawling example.com spider ... ]

>>> DEPTH LEVEL: 1 <<<
 # Scraped Items  ------------------------------------------------------------

[]

 # Requests  -----------------------------------------------------------------

[<GET item_details_url>]


>>> DEPTH LEVEL: 2 <<<
 # Scraped Items  ------------------------------------------------------------

[{'url': <item_url>}]

 # Requests  -----------------------------------------------------------------

[]
```

检查从单个 start_url 爬取到的 item 也是很简单的:

```
$ scrapy parse --spider=myspider -d 3 'http://example.com/page1'
Scrapy终端(Shell)
```

尽管 `parse` 命令对检查 spider 的效果十分有用，但除了显示收到的 response 及输出外，其对检查回调函数内部的过程并没有提供什么便利。如何调试 parse_detail 没有收到 item 的情况呢?

幸运的是，救世主 shell 出现了(参考 `在 spide r中启动 shell 来查看 response` ):

```
from scrapy.shell import inspect_response

def parse_details(self, response):
    item = response.meta.get('item', None)
    if item:
        # populate more `item` fields
        return item
    else:
        inspect_response(response, self)
```

参考 `在spider中启动shell来查看response` 。

## 在浏览器中打开

有时候您想查看某个 response 在浏览器中显示的效果，这是可以使用 `open_in_browser` 功能。下面是使用的例子：

```
from scrapy.utils.response import open_in_browser

def parse_details(self, response):
    if "item name" not in response.body:
        open_in_browser(response)
```

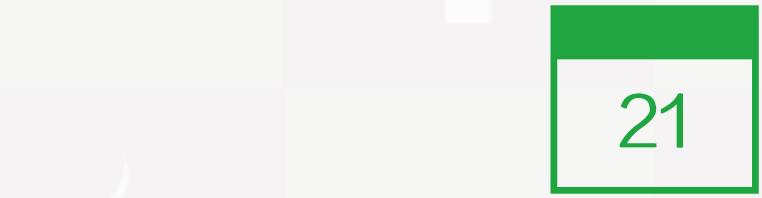
`open_in_browser` 将会使用 Scrapy 获取到的 response 来打开浏览器，并且调整 base tag 使得图片及样式(style)能正常显示。

## Logging

记录(logging)是另一个获取到 spider 运行信息的方法。虽然不是那么方便，但好处是 log 的内容在以后的运行中也可以看到：

```
from scrapy import log

def parse_details(self, response):
    item = response.meta.get('item', None)
    if item:
        # populate more `item` fields
        return item
    else:
        self.log('No item received for %s' % response.url,
            level=log.WARNING)
```

更多内容请参见 `Logging` 部分。



# 21

# Spiders Contracts

新版功能。

> 注解
>
> 这是一个新引入(Scrapy 0.15)的特性，在后续的功能/API 更新中可能有所改变，查看 release notes 来了解更新。

测试 spider 是一件挺烦人的事情，尤其是只能编写单元测试(unit test)没有其他办法时，就更恼人了。 Scrapy 通过合同(contract)的方式来提供了测试 spider 的集成方法。

您可以硬编码(hardcode)一个样例(sample)url， 设置多个条件来测试回调函数处理 repsponse 的结果，来测试 spider 的回调函数。 每个 contract 包含在文档字符串(docstring)里，以 @ 开头。 查看下面的例子：

```
def parse(self, response):
   """ This function parses a sample response. Some contracts are mingled
   with this docstring.

   @url http://www.amazon.com/s?field-keywords=selfish+gene
   @returns items 1 16
   @returns requests 0 0
   @scrapes Title Author Year Price
   """
```

该回调函数使用了三个内置的 contract 来测试：

### class scrapy.contracts.default.UrlContract

该 constract(@url)设置了用于检查 spider 的其他 constract 状态的样例 url。该 contract 是必须的，所有缺失该 contract 的回调函数在测试时将会被忽略：

```
@url url
```

### class scrapy.contracts.default.ReturnsContract

该 contract(@returns)设置 spider 返回的 items 和 requests 的上界和下界。上界是可选的：

```
@returns item(s)|request(s) [min [max]]
```

### class scrapy.contracts.default.ScrapesContract

该 contract(@scrapes)检查回调函数返回的所有 item 是否有特定的 fields：

```
@scrapes field_1 field_2 ...
```

使用 `check` 命令来运行 contract 检查。

## 自定义 Contracts

如果您想要比内置 scrapy contract 更为强大的功能，可以在您的项目里创建并设置您自己的 contract，并使用 `SPIDER_CONTRACTS` 设置来加载：

```
SPIDER_CONTRACTS = {
    'myproject.contracts.ResponseCheck': 10,
    'myproject.contracts.ItemValidate': 10,
}
```

每个 contract 必须继承 `scrapy.contracts.Contract` 并覆盖下列三个方法：

### class scrapy.contracts.Contract(method, *args)

参数：

- method (function) – contract 所关联的回调函数
- args (list) – 传入 docstring 的(以空格区分的)argument 列表(list)

**adjust_request_args(args)**
接收一个 `字典(dict)` 作为参数。该参数包含了所有 `Request` 对象 参数的默认值。该方法必须返回相同或修改过的字典。

**pre_process(response)**
该函数在 sample request 接收到 response 后，传送给回调函数前被调用，运行测试。

**post_process(output)**
该函数处理回调函数的输出。迭代器(Iterators)在传输给该函数前会被列表化(listified)。

该样例 contract 在 response 接收时检查了是否有自定义 header。 在失败时 Raise `scrapy.exceptions.ContractFaild` 来展现错误：

```
from scrapy.contracts import Contract
from scrapy.exceptions import ContractFail

class HasHeaderContract(Contract):
    """ Demo contract which checks the presence of a custom header
        @has_header X-CustomHeader
    """

    name = 'has_header'
```

```
def pre_process(self, response):
    for header in self.args:
        if header not in response.headers:
            raise ContractFail('X-CustomHeader not present')
```

# 22

# 实践经验(Common Practices)

本章节记录了使用 Scrapy 的一些实践经验(common practices)。这包含了很多使用不会包含在其他特定章节的的内容。

# 在脚本中运行 Scrapy

除了常用的 `scrapy crawl` 来启动 Scrapy，您也可以使用 `API` 在脚本中启动 Scrapy。

需要注意的是，Scrapy 是在 Twisted 异步网络库上构建的，因此其必须在 Twisted reactor 里运行。

另外，在 spider 运行结束后，您必须自行关闭 Twisted reactor。这可以通过在 `CrawlerRunner.crawl` 所返回的对象中添加回调函数来实现。

下面给出了如何实现的例子，使用 testspiders 项目作为例子。

```
from twisted.internet import reactor
from scrapy.crawler import CrawlerRunner
from scrapy.utils.project import get_project_settings

runner = CrawlerRunner(get_project_settings())

 # 'followall' is the name of one of the spiders of the project.

d = runner.crawl('followall', domain='scrapinghub.com')
d.addBoth(lambda _: reactor.stop())
reactor.run() # the script will block here until the crawling is finished
Running spiders outside projects it’s not much different. You have to create a generic Settings object and populate it as

Spiders can still be referenced by their name if SPIDER_MODULES is set with the modules where Scrapy should look for

from twisted.internet import reactor
from scrapy.spider import Spider
from scrapy.crawler import CrawlerRunner
from scrapy.settings import Settings

class MySpider(Spider):
    # Your spider definition
    ...

settings = Settings({'USER_AGENT': 'Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)'})
runner = CrawlerRunner(settings)

d = runner.crawl(MySpider)
d.addBoth(lambda _: reactor.stop())
reactor.run() # the script will block here until the crawling is finished
```

> 参见

> Twisted Reactor Overview.

## 同一进程运行多个 spider

默认情况下，当您执行 `scrapy crawl` 时，Scrapy 每个进程运行一个 spider。当然，Scrapy 通过 `内部(internal)API` 也支持单进程多个 spider。

下面以 testspiders 作为例子来说明如何同时运行多个 spider:

```
from twisted.internet import reactor, defer
from scrapy.crawler import CrawlerRunner
from scrapy.utils.project import get_project_settings

runner = CrawlerRunner(get_project_settings())
dfs = set()
for domain in ['scrapinghub.com', 'insophia.com']:
    d = runner.crawl('followall', domain=domain)
    dfs.add(d)

defer.DeferredList(dfs).addBoth(lambda _: reactor.stop())
reactor.run() # the script will block here until all crawling jobs are finished
```

相同的例子，不过通过链接(chaining) deferred 来线性运行 spider:

```
from twisted.internet import reactor, defer
from scrapy.crawler import CrawlerRunner
from scrapy.utils.project import get_project_settings

runner = CrawlerRunner(get_project_settings())

@defer.inlineCallbacks
def crawl():
    for domain in ['scrapinghub.com', 'insophia.com']:
        yield runner.crawl('followall', domain=domain)
    reactor.stop()

crawl()
reactor.run() # the script will block here until the last crawl call is finished
```

> 参见
>
> 在脚本中运行 Scrapy。

# 分布式爬虫(Distributed crawls)

Scrapy 并没有提供内置的机制支持分布式(多服务器)爬取。不过还是有办法进行分布式爬取，取决于您要怎么分布了。

如果您有很多 spider，那分布负载最简单的办法就是启动多个 Scrapyd，并分配到不同机器上。

如果想要在多个机器上运行一个单独的 spider，那您可以将要爬取的 url 进行分块，并发送给 spider。 例如:

首先，准备要爬取的 url 列表，并分配到不同的文件 url 里:

```
http://somedomain.com/urls-to-crawl/spider1/part1.list
http://somedomain.com/urls-to-crawl/spider1/part2.list
http://somedomain.com/urls-to-crawl/spider1/part3.list
```

接着在 3 个不同的 Scrapd 服务器中启动 spider。spider 会接收一个(spider)参数 part，该参数表示要爬取的分块:

```
curl http://scrapy1.mycompany.com:6800/schedule.json -d project=myproject -d spider=spider1 -d part=1
curl http://scrapy2.mycompany.com:6800/schedule.json -d project=myproject -d spider=spider1 -d part=2
curl http://scrapy3.mycompany.com:6800/schedule.json -d project=myproject -d spider=spider1 -d part=3
```

## 避免被禁止(ban)

有些网站实现了特定的机制，以一定规则来避免被爬虫爬取。 与这些规则打交道并不容易，需要技巧，有时候也需要些特别的基础。 如果有疑问请考虑联系商业支持。

下面是些处理这些站点的建议(tips):

- 使用 user agent 池，轮流选择之一来作为 user agent。池中包含常见的浏览器的 user agent(google 一下一大堆)
- 禁止 cookies(参考 `COOKIES_ENABLED` )，有些站点会使用 cookies 来发现爬虫的轨迹。
- 设置下载延迟(2 或更高)。参考 `DOWNLOAD_DELAY` 设置。
- 如果可行，使用 Google cache 来爬取数据，而不是直接访问站点。
- 使用 IP 池。例如免费的 Tor 项目或付费服务(ProxyMesh)。
- 使用高度分布式的下载器(downloader)来绕过禁止(ban)，您就只需要专注分析处理页面。这样的例子有:Crawlera

如果您仍然无法避免被 ban，考虑联系商业支持。

## 动态创建 Item 类

对于有些应用，item 的结构由用户输入或者其他变化的情况所控制。您可以动态创建 class。

```
from scrapy.item import DictItem, Field

def create_item_class(class_name, field_list):
fields = {field_name: Field() for field_name in field_list}

return type(class_name, (DictItem,), {'fields': fields})
```

# 23

# 通用爬虫(Broad Crawls)

Scrapy 默认对特定爬取进行优化。这些站点一般被一个单独的 Scrapy spider 进行处理， 不过这并不是必须或要求的(例如，也有通用的爬虫能处理任何给定的站点)。

除了这种爬取完某个站点或没有更多请求就停止的”专注的爬虫”，还有一种通用的爬取类型，其能爬取大量(甚至是无限)的网站， 仅仅受限于时间或其他的限制。 这种爬虫叫做”通用爬虫(broad crawls)”，一般用于搜索引擎。

通用爬虫一般有以下通用特性:

- 其爬取大量(一般来说是无限)的网站而不是特定的一些网站。
- 其不会将整个网站都爬取完毕，因为这十分不实际(或者说是不可能)完成的。相反，其会限制爬取的时间及数量。
- 其在逻辑上十分简单(相较于具有很多提取规则的复杂的 spider)，数据会在另外的阶段进行后处理(post-processed)
- 其并行爬取大量网站以避免被某个网站的限制所限制爬取的速度(为表示尊重，每个站点爬取速度很慢但同时爬取很多站点)。

正如上面所述，Scrapy 默认设置是对特定爬虫做了优化，而不是通用爬虫。不过， 鉴于其使用了异步架构，Scrapy 对通用爬虫也十分适用。 本篇文章总结了一些将 Scrapy 作为通用爬虫所需要的技巧， 以及相应针对通用爬虫的 Scrapy 设定的一些建议。

## 增加并发

并发是指同时处理的 request 的数量。其有全局限制和局部(每个网站)的限制。

Scrapy 默认的全局并发限制对同时爬取大量网站的情况并不适用，因此您需要增加这个值。 增加多少取决于您的爬虫能占用多少 CPU。 一般开始可以设置为 100 。不过最好的方式是做一些测试，获得 Scrapy 进程占取 CPU 与并发数的关系。 为了优化性能，您应该选择一个能使 CPU 占用率在 80%-90% 的并发数。

## 增加全局并发数

```
CONCURRENT_REQUESTS = 100
```

## 降低 log 级别

当进行通用爬取时，一般您所注意的仅仅是爬取的速率以及遇到的错误。 Scrapy 使用 INFO log 级别来报告这些信息。为了减少 CPU 使用率(及记录 log 存储的要求), 在生产环境中进行通用爬取时您不应该使用 DEBUG log 级别。 不过在开发的时候使用 DEBUG 应该还能接受。

设置 Log 级别：

```
LOG_LEVEL = 'INFO'
```

## 禁止 cookies

除非您 真的 需要，否则请禁止 cookies。在进行通用爬取时 cookies 并不需要，(搜索引擎则忽略 cookies)。禁止 cookies 能减少 CPU 使用率及 Scrapy 爬虫在内存中记录的踪迹，提高性能。

禁止 cookies：

```
COOKIES_ENABLED = False
```

## 禁止重试

对失败的 HTTP 请求进行重试会减慢爬取的效率，尤其是当站点响应很慢(甚至失败)时， 访问这样的站点会造成超时并重试多次。这是不必要的，同时也占用了爬虫爬取其他站点的能力。

禁止重试：

```
RETRY_ENABLED = False
```

## 减小下载超时

如果您对一个非常慢的连接进行爬取(一般对通用爬虫来说并不重要)， 减小下载超时能让卡住的连接能被快速的放弃并解放处理其他站点的能力。

减小下载超时：

```
DOWNLOAD_TIMEOUT = 15
```

## 禁止重定向

除非您对跟进重定向感兴趣，否则请考虑关闭重定向。当进行通用爬取时，一般的做法是保存重定向的地址，并在之后的爬取进行解析。这保证了每批爬取的 request 数目在一定的数量，否则重定向循环可能会导致爬虫在某个站点耗费过多资源。

关闭重定向：

```
REDIRECT_ENABLED = False
```

## 启用 “Ajax Crawlable Pages” 爬取

有些站点(基于 2013 年的经验数据，之多有 1%)声明其为 ajax crawlabl。这意味着该网站提供了原本只有 ajax 获取到的数据的纯 HTML 版本。网站通过两种方法声明：

1. 在 url 中使用 `#!` - 这是默认的方式;
2. 使用特殊的 meta 标签 - 这在”main”，“index” 页面中使用。

Scrapy 自动解决(1)；解决(2)您需要启用 AjaxCrawlMiddleware：

```
AJAXCRAWL_ENABLED = True
```

通用爬取经常抓取大量的 “index” 页面； AjaxCrawlMiddleware 能帮助您正确地爬取。 由于有些性能问题，且对于特定爬虫没有什么意义，该中间默认关闭。

# 24

# 借助 Firefox 来爬取

这里介绍一些使用 Firefox 进行爬取的点子及建议，以及一些帮助爬取的 Firefox 实用插件。

## 在浏览器中检查 DOM 的注意事项

Firefox 插件操作的是活动的浏览器 DOM(live browser DOM)，这意味着当您检查网页源码的时候， 其已经不是原始的 HTML，而是经过浏览器清理并执行一些 Javascript 代码后的结果。 Firefox 是个典型的例子，其会在 table 中添加 `<tbody>` 元素。而 Scrapy 相反，其并不修改原始的 HTML，因此如果在 XPath 表达式中使用 `<tbody>`，您将获取不到任何数据。

所以，当 XPath 配合 Firefox 使用时您需要记住以下几点:

- 当检查 DOM 来查找 Scrapy 使用的 XPath 时，禁用 Firefox 的 Javascrpit。
- 永远不要用完整的 XPath 路径。使用相对及基于属性(例如 `id`， `class`， `width` 等)的路径 或者具有区别性的特性例如 `contains(@href, 'image')` 。
- 永远不要在 XPath 表达式中加入 `<tbody>` 元素，除非您知道您在做什么

# 对爬取有帮助的实用 Firefox 插件

## Firebug

Firebug 是一个在 web 开发者间很著名的工具，其对抓取也十分有用。尤其是检查元素(Inspect Element)特性对构建抓取数据的 XPath 十分方便。当移动鼠标在页面元素时，您能查看相应元素的 HTML 源码。

查看 使用 Firebug 进行爬取 ，了解如何配合 Scrapy 使用 Firebug 的详细教程。

## XPather

XPather 能让你在页面上直接测试 XPath 表达式。

## XPath Checker

XPath Checker 是另一个用于测试 XPath 表达式的 Firefox 插件。

## Tamper Data

Tamper Data 是一个允许您查看及修改 Firefox 发送的 header 的插件。Firebug 能查看 HTTP header，但无法修改。

## Firecookie

Firecookie 使得查看及管理 cookie 变得简单。您可以使用这个插件来创建新的 cookie，删除存在的 cookie，查看当前站点的 cookie，管理 cookie 的权限及其他功能。

25

# 使用 Firebug 进行爬取

> 注解
>
> 本教程所使用的样例站 Google Directory 已经被 Google 关闭了。不过教程中的概念任然适用。 如果您打算使用一个新的网站来更新本教程，您的贡献是再欢迎不过了。 详细信息请参考 Contributing to Scrapy。

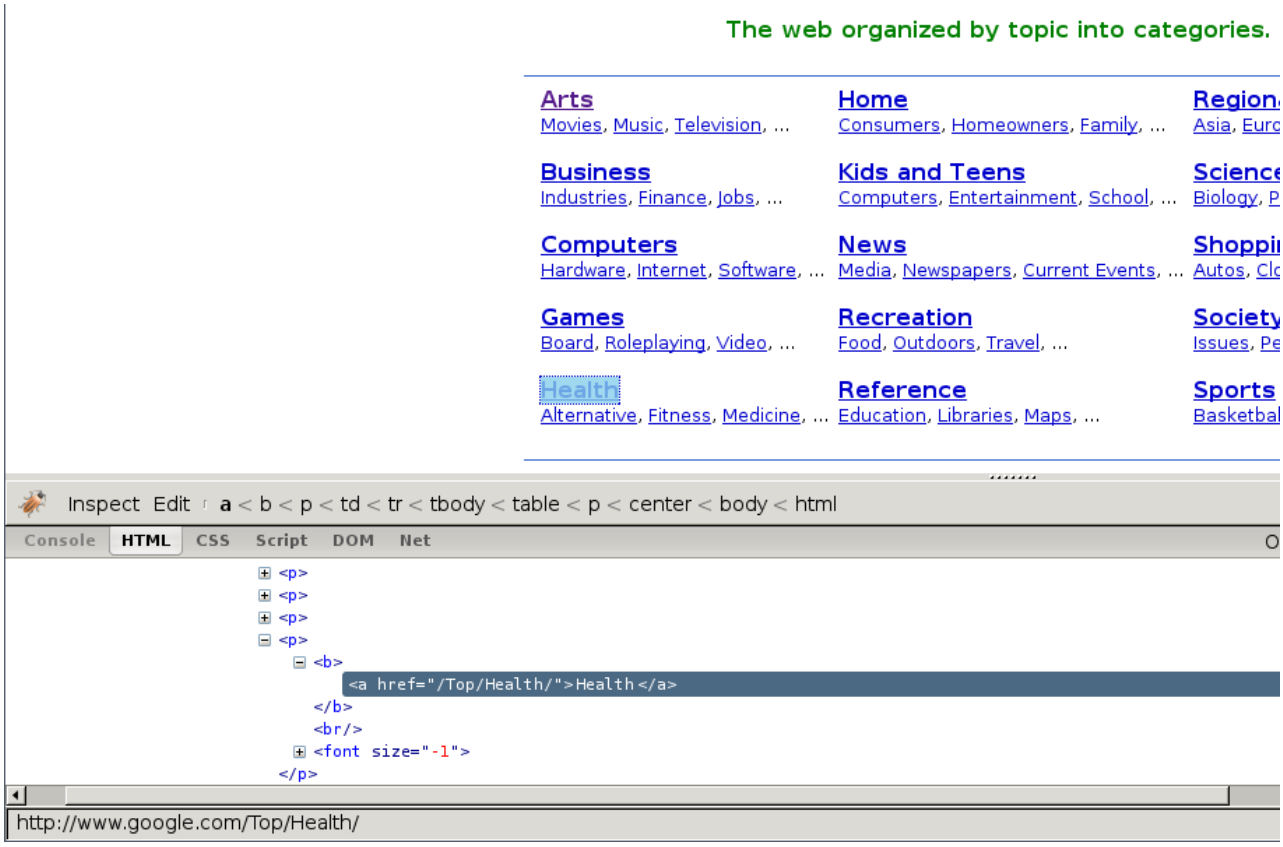
# 介绍

本文档介绍了如何适用 Firebug(一个 Firefox 的插件)来使得爬取更为简单，有趣。更多有意思的 Firefox 插件请参考对爬取有帮助的实用 Firefox 插件。使用 Firefox 插件检查页面需要有些注意事项: 在浏览器中检查 DOM 的注意事项 。

在本样例中将展现如何使用 Firebug 从 Google Directory 来爬取数据。Google Directory 包含了 入门教程 里所使用的 Open Directory Project 中一样的数据，不过有着不同的结构。

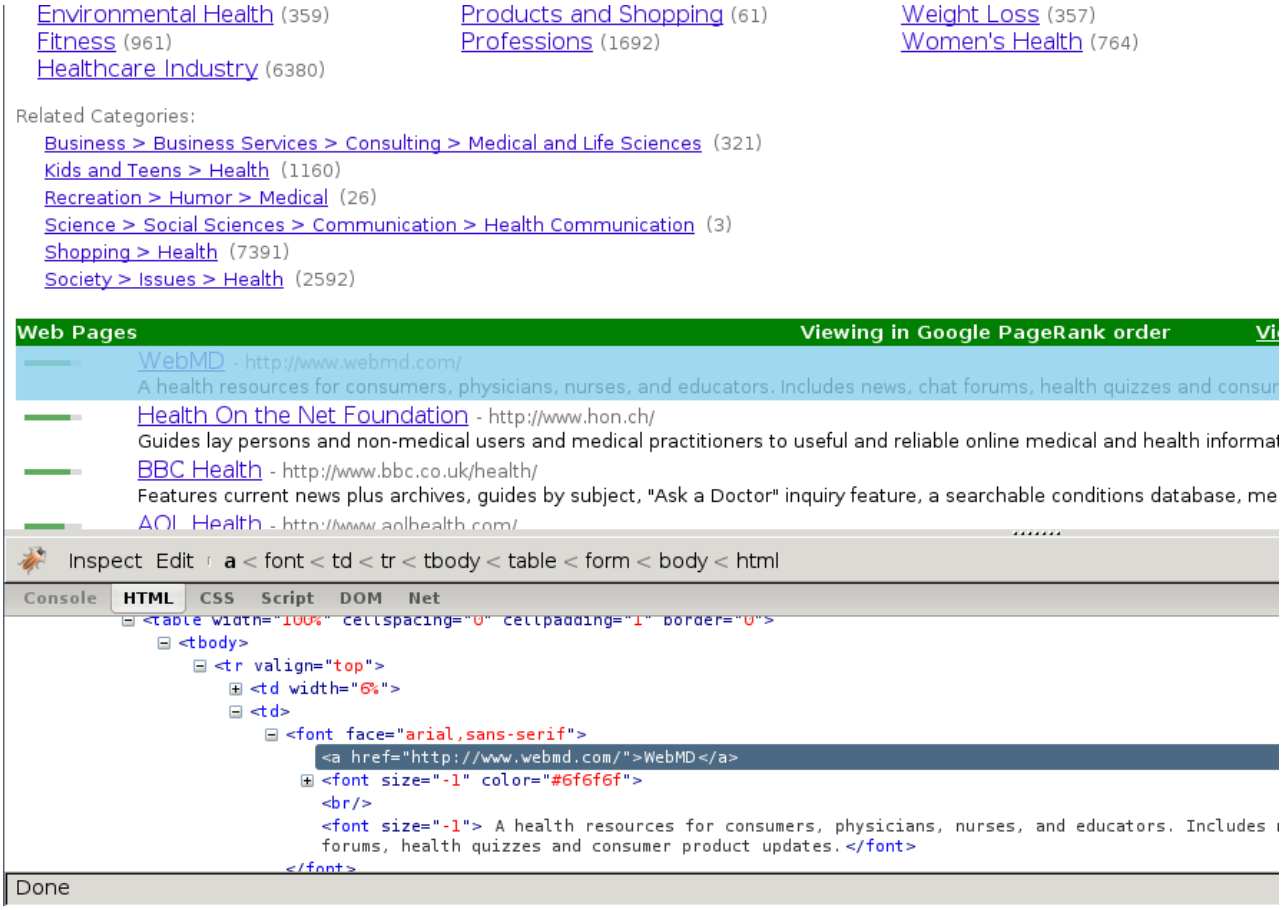
Firebug 提供了非常实用的 检查元素 功能。该功能允许您将鼠标悬浮在不同的页面元素上， 显示相应元素的 HTML 代码。否则，您只能十分痛苦的在 HTML 的 body 中手动搜索标签。

在下列截图中，您将看到 检查元素 的执行效果。




首先我们能看到目录根据种类进行分类的同时，还划分了子类。

不过，看起来子类还有更多的子类，而不仅仅是页面显示的这些，所以我们接着查找：




正如路径的概念那样，子类包含了其他子类的链接，同时也链接到实际的网站中。

## 获取到跟进(follow)的链接

查看路径的 URL，我们可以看到 URL 的通用模式(pattern)：

http://directory.google.com/Category/Subcategory/Another_Subcategory

了解到这个消息，我们可以构建一个跟进的链接的正则表达式：

```
directory\.google\.com/[A-Z][a-zA-Z_/]+$
```

因此，根据这个表达式，我们创建第一个爬取规则：

```
Rule(LinkExtractor(allow='directory.google.com/[A-Z][a-zA-Z_/]+$', ),
    'parse_category',
    follow=True,
),
```

`Rule` 对象指导基于 `CrawlSpider` 的 spider 如何跟进目录链接。 parse_category 是 spider 的方法，用于从页面中处理也提取数据。

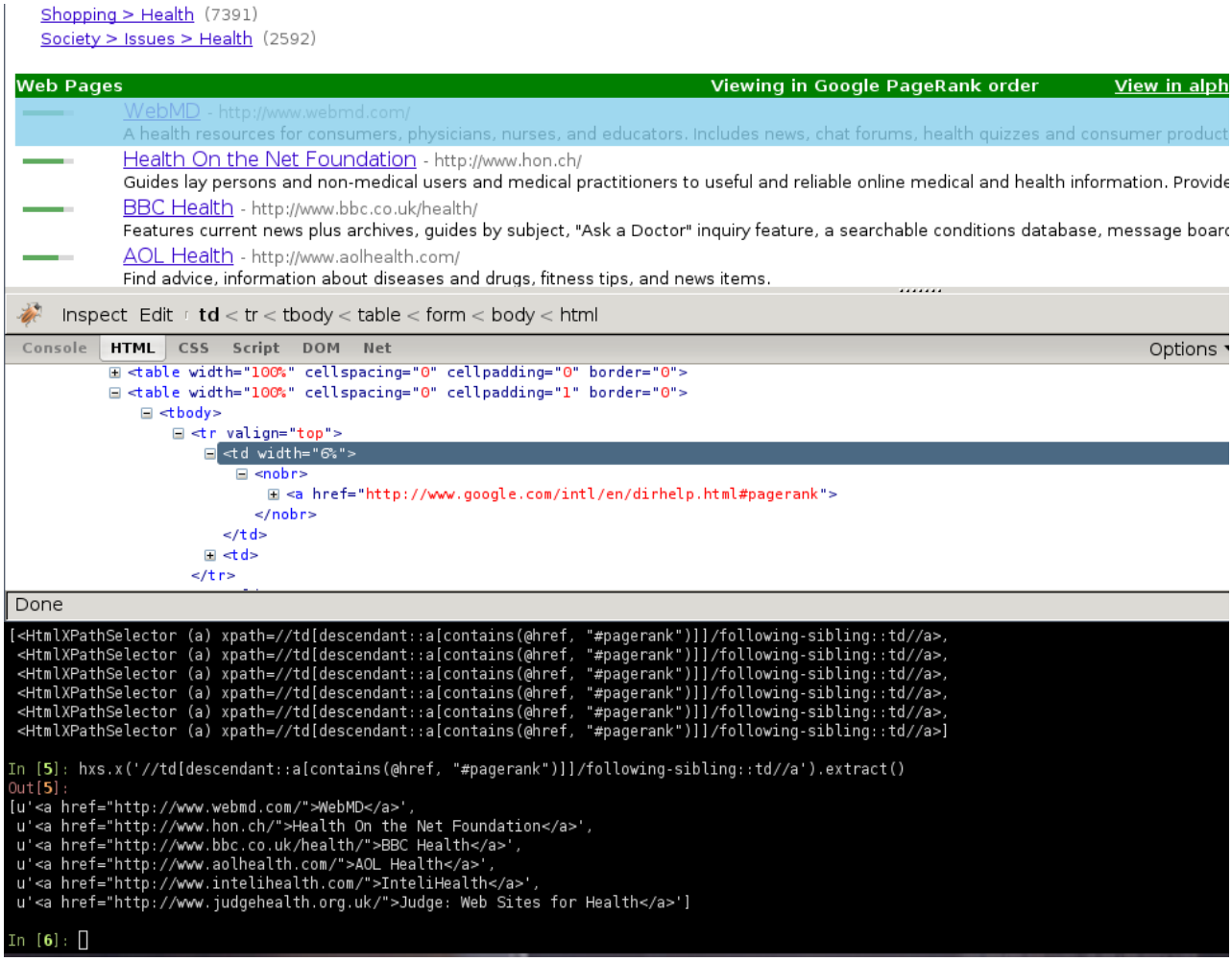
spider 的代码如下：

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule

class GoogleDirectorySpider(CrawlSpider):
    name = 'directory.google.com'
    allowed_domains = ['directory.google.com']
    start_urls = ['http://directory.google.com/']

    rules = (
        Rule(LinkExtractor(allow='directory\.google\.com/[A-Z][a-zA-Z_/]+$'),
            'parse_category', follow=True,
        ),
    )

    def parse_category(self, response):
        # write the category page data extraction code here
        pass
```

## 提取数据

现在我们来编写提取数据的代码。

在 Firebug 的帮助下，我们将查看一些包含网站链接的网页(以 http://directory.google.com/Top/Arts/Awards/为例)，找到使用 Selectors 提取链接的方法。我们也将使用 `Scrapy shell` 来测试得到的 `XPath` 表达式，确保表达式工作符合预期。




正如您所看到的那样，页面的标记并不是十分明显: 元素并不包含 `id`， `class` 或任何可以区分的属性。所以我们将使用等级槽(rank bar)作为指示点来选择提取的数据，创建 XPath。

使用 Firebug，我们可以看到每个链接都在 `td` 标签中。该标签存在于同时(在另一个 `td` )包含链接的等级槽(ranking bar)的 `tr` 中。

所以我们选择等级槽(ranking bar)，接着找到其父节点(tr)，最后是(包含我们要爬取数据的)链接的 td 。

对应的 XPath:

```
//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td//a
```

使用 Scrapy 终端 来测试这些复杂的 XPath 表达式，确保其工作符合预期。

简单来说，该表达式会查找等级槽的 td 元素，接着选择所有 td 元素，该元素拥有子孙 a 元素，且 a 元素的属性 href 包含字符串 #pagerank 。

当然，这不是唯一的 XPath，也许也不是选择数据的最简单的那个。 其他的方法也可能是，例如，选择灰色的链接的 font 标签。

最终，我们编写 parse_category() 方法：

```
def parse_category(self, response):

    # The path to website links in directory page
    links = response.xpath('//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td/font')

    for link in links:
        item = DirectoryItem()
        item['name'] = link.xpath('a/text()').extract()
        item['url'] = link.xpath('a/@href').extract()
        item['description'] = link.xpath('font[2]/text()').extract()
        yield item
```

注意，您可能会遇到有些在 Firebug 找到，但是在原始 HTML 中找不到的元素， 例如典型的 <tbody> 元素，或者 Firebug 检查活动 DOM(live DOM)所看到的元素，但元素由 javascript 动态生成，并不在 HTML 源码中。(原文语句乱了，上面为意译- -: or tags which Therefer in page HTML sources may on Firebug inspects the live DOM )。

# 26

# 调试内存溢出

在 Scrapy 中，类似 RequestsResponse 及 Items 的对象具有有限的生命周期: 他们被创建，使用，最后被销毁。

这些对象中，Request 的生命周期应该是最长的，其会在调度队列(Scheduler queue)中一直等待，直到被处理。更多内容请参考架构概览。

由于这些 Scrapy 对象拥有很长的生命，因此将这些对象存储在内存而没有正确释放的危险总是存在。 而这导致了所谓的”内存泄露”。

为了帮助调试内存泄露，Scrapy 提供了跟踪对象引用的机制，叫做 `trackref` ， 或者您也可以使用第三方提供的更先进内存调试库 `Guppy` (更多内容请查看下面)。而这都必须在 `Telnet 终端` 中使用。

## 内存泄露的常见原因

内存泄露经常是由于 Scrapy 开发者在 Requests 中(有意或无意)传递对象的引用(例如，使用 `meta` 属性或 request 回调函数)，使得该对象的生命周期与 Request 的生命周期所绑定。这是目前为止最常见的内存泄露的原因，同时对新手来说也是一个比较难调试的问题。

在大项目中，spider 是由不同的人所编写的。而这其中有的 spider 可能是有”泄露的”， 当所有的爬虫同时运行时，这些影响了其他(写好)的爬虫，最终，影响了整个爬取进程。

与此同时，在不限制框架的功能的同时避免造成这些造成泄露的原因是十分困难的。因此， 我们决定不限制这些功能而是提供调试这些泄露的实用工具。这些工具回答了一个问题: 哪个 spider 在泄露 。

内存泄露可能存在与一个您编写的中间件，管道(pipeline) 或扩展，在代码中您没有正确释放 (之前分配的)资源。例如，您在 `spider_opened` 中分配资源但在 `spider_closed` 中没有释放它们。

# 使用 trackref 调试内存泄露

trackref 是 Scrapy 提供用于调试大部分内存泄露情况的模块。 简单来说，其追踪了所有活动(live)的 Request，Request，Item 及 Selector 对象的引用。

您可以进入 telnet 终端并通过 prefs() 功能来检查多少(上面所提到的)活跃(alive)对象。 pref() 是 print_live_refs() 功能的引用：

```
telnet localhost 6023

>>> prefs()
Live References

ExampleSpider                1   oldest: 15s ago
HtmlResponse                10   oldest: 1s ago
Selector                     2   oldest: 0s ago
FormRequest                878   oldest: 7s ago
```

正如所见，报告也展现了每个类中最老的对象的时间(age)。 If you’re running multiple spiders per process chances are you can figure out which spider is leaking by looking at the oldest request or response. You can get the oldest object of each class using the get_oldest() function (from the telnet console).

如果您有内存泄露，那您能找到哪个 spider 正在泄露的机会是查看最老的 request 或 response。 您可以使用 get_oldest() 方法来获取每个类中最老的对象， 正如此所示(在终端中)(原文档没有样例)。

## 哪些对象被追踪了?

trackref 追踪的对象包括以下类(及其子类)的对象：

- scrapy.http.Request
- scrapy.http.Response
- scrapy.item.Item
- scrapy.selector.Selector
- scrapy.spider.Spider

## 真实例子

让我们来看一个假设的具有内存泄露的准确例子。

假如我们有些 spider 的代码中有一行类似于这样的代码：

```
return Request("http://www.somenastyspider.com/product.php?pid=%d" % product_id,
    callback=self.parse, meta={referer: response}")
```

代码中在 request 中传递了一个 response 的引用，使得 reponse 的生命周期与 request 所绑定， 进而造成了内存泄露。

让我们来看看如何使用 trackref 工具来发现哪一个是有问题的 spider(当然是在不知道任何的前提的情况下)。

当 crawler 运行了一小阵子后，我们发现内存占用增长了很多。 这时候我们进入 telnet 终端，查看活跃(live)的引用：

```
>>> prefs()
Live References

SomenastySpider                 1   oldest: 15s ago
HtmlResponse                 3890   oldest: 265s ago
Selector                        2   oldest: 0s ago
Request                      3878   oldest: 250s ago
```

上面具有非常多的活跃(且运行时间很长)的 response，而其比 Request 的时间还要长的现象肯定是有问题的。因此，查看最老的 response：

```
>>> from scrapy.utils.trackref import get_oldest
>>> r = get_oldest('HtmlResponse')
>>> r.url
'http://www.somenastyspider.com/product.php?pid=123'
```

就这样，通过查看最老的 response 的 URL，我们发现其属于 somenastyspider.com spider。现在我们可以查看该 spider 的代码并发现导致泄露的那行代码(在 request 中传递 response 的引用)。

如果您想要遍历所有而不是最老的对象，您可以使用 iter_all() 方法：

```
>>> from scrapy.utils.trackref import iter_all
>>> [r.url for r in iter_all('HtmlResponse')]
['http://www.somenastyspider.com/product.php?pid=123',
 'http://www.somenastyspider.com/product.php?pid=584',
...
```

## 很多 spider?

如果您的项目有很多的 spider，prefs() 的输出会变得很难阅读。针对于此，该方法具有 ignore 参数，用于忽略特定的类(及其子类)。例如：

```
>>> from scrapy.spider import Spider
>>> prefs(ignore=Spider)
```

将不会展现任何 spider 的活跃引用。

## scrapy.utils.trackref 模块

以下是 `trackref` 模块中可用的方法。

### class scrapy.utils.trackref.object_ref

如果您想通过 trackref 模块追踪活跃的实例，继承该类(而不是对象)。

### scrapy.utils.trackref.print_live_refs(class_name, ignore=NoneType)

打印活跃引用的报告，以类名分类。

**参数:** ignore (*类或者类的元组*) – 如果给定，所有指定类(或者类的元组)的对象将会被忽略。

### scrapy.utils.trackref.get_oldest(class_name)

返回给定类名的最老活跃(alive)对象，如果没有则返回 None。首先使用 `print_live_refs()` 来获取每个类所跟踪的所有活跃(live)对象的列表。

### scrapy.utils.trackref.iter_all(class_name)

返回一个能给定类名的所有活跃对象的迭代器，如果没有则返回 `None` 。首先使用 `print_live_refs()` 来获取每个类所跟踪的所有活跃(live)对象的列表。

## 使用 Guppy 调试内存泄露

trackref 提供了追踪内存泄露非常方便的机制，其仅仅追踪了比较可能导致内存泄露的对象 (Requests, Response, Items 及 Selectors)。然而，内存泄露也有可能来自其他(更为隐蔽的)对象。 如果是因为这个原因，通过 trackref 则无法找到泄露点，您仍然有其他工具:Guppy library。

如果使用 `setuptools`，您可以通过下列命令安装 Guppy:

```
easy_install guppy
```

telnet 终端也提供了快捷方式(hpy)来访问 Guppy 堆对象(heap objects)。下面给出了查看堆中所有可用的 Python 对象的例子:

```
>>> x = hpy.heap()
>>> x.bytype
Partition of a set of 297033 objects. Total size = 52587824 bytes.
 Index  Count   %     Size   % Cumulative  % Type
     0  22307   8 16423880  31  16423880  31 dict
     1 122285  41 12441544  24  28865424  55 str
     2  68346  23  5966696  11  34832120  66 tuple
     3    227   0  5836528  11  40668648  77 unicode
     4   2461   1  2222272   4  42890920  82 type
     5  16870   6  2024400   4  44915320  85 function
     6  13949   5  1673880   3  46589200  89 types.CodeType
     7  13422   5  1653104   3  48242304  92 list
     8   3735   1  1173680   2  49415984  94 _sre.SRE_Pattern
     9   1209   0   456936   1  49872920  95 scrapy.http.headers.Headers
<1676 more rows. Type e.g. '_.more' to view.>
```

您可以看到大部分的空间被字典所使用。接着，如果您想要查看哪些属性引用了这些字典， 您可以:

```
>>> x.bytype[0].byvia
Partition of a set of 22307 objects. Total size = 16423880 bytes.
 Index  Count   %     Size   % Cumulative  % Referred Via:
     0  10982  49  9416336  57   9416336  57 '.__dict__'
     1   1820   8  2681504  16  12097840  74 '.__dict__', '.func_globals'
     2   3097  14  1122904   7  13220744  80
     3    990   4   277200   2  13497944  82 "['cookies']"
     4    987   4   276360   2  13774304  84 "['cache']"
     5    985   4   275800   2  14050104  86 "['meta']"
     6    897   4   251160   2  14301264  87 '[2]'
     7      1   0   196888   1  14498152  88 "['moduleDict']", "['modules']"
     8    672   3   188160   1  14686312  89 "['cb_kwargs']"
```

```
    9     27   0   155016   1  14841328  90 '[1]'
<333 more rows. Type e.g. '_.more' to view.>
```

如上所示，Guppy 模块十分强大，不过也需要一些关于 Python 内部的知识。关于 Guppy 的更多内容请参考 Guppy documentation。

## Leaks without leaks

有时候，您可能会注意到 Scrapy 进程的内存占用只在增长，从不下降。不幸的是，有时候这并不是 Scrapy 或者您的项目在泄露内存。这是由于一个已知(但不有名)的 Python 问题。Python 在某些情况下可能不会返回已经释放的内存到操作系统。关于这个问题的更多内容请看：

- Python Memory Management
- Python Memory Management Part 2
- Python Memory Management Part 3

改进方案由 Evan Jones 提出，在 这篇文章 中详细介绍，在 Python 2.5 中合并。 不过这仅仅减小了这个问题，并没有完全修复。引用这片文章：

*不幸的是，这个 patch 仅仅会释放没有在其内部分配对象的区域(arena)。这意味着 碎片化是一个大问题。某个应用可以拥有很多空闲内存，分布在所有的区域(arena)中，但是没法释放任何一个。这个问题存在于所有内存分配器中。解决这个问题的唯一办法是 转化到一个更为紧凑(compact)的垃圾回收器，其能在内存中移动对象。这需要对 Python 解析器做一个显著的修改。*

这个问题将会在未来 Scrapy 发布版本中得到解决。我们打算转化到一个新的进程模型，并在可回收的子进程池中运行 spider。

27

# 下载项目图片

Scrapy 提供了一个 Item Pipeline，来下载属于某个特定项目的图片，比如，当你抓取产品时，也想把它们的图片下载到本地。

这条管道，被称作图片管道，在 `ImagesPipeline` 类中实现，提供了一个方便并具有额外特性的方法，来下载并本地存储图片：

- 将所有下载的图片转换成通用的格式（JPG）和模式（RGB）
- 避免重新下载最近已经下载过的图片
- 缩略图生成
- 检测图像的宽/高，确保它们满足最小限制

这个管道也会为那些当前安排好要下载的图片保留一个内部队列，并将那些到达的包含相同图片的项目连接到那个队列中。这可以避免多次下载几个项目共享的同一个图片。

Pillow 是用来生成缩略图，并将图片归一化为 JPEG/RGB 格式，因此为了使用图片管道，你需要安装这个库。Python Imaging Library(PIL) 在大多数情况下是有效的，但众所周知，在一些设置里会出现问题，因此我们推荐使用 Pillow 而不是 PIL。

## 使用图片管道

当使用 `ImagesPipeline`，典型的工作流程如下所示：

1. 在一个爬虫里，你抓取一个项目，把其中图片的 URL 放入 `image_urls` 组内。
2. 项目从爬虫内返回，进入项目管道。
3. 当项目进入 `ImagesPipeline`，`image_urls` 组内的 URLs 将被 Scrapy 的调度器和下载器（这意味着调度器和下载器的中间件可以复用）安排下载，当优先级更高，会在其他页面被抓取前处理。项目会在这个特定的管道阶段保持“locker”的状态，直到完成图片的下载（或者由于某些原因未完成下载）。
4. 当图片下载完，另一个组(`images`)将被更新到结构中。这个组将包含一个字典列表，其中包括下载图片的信息，比如下载路径、源抓取地址（从 `image_urls` 组获得）和图片的校验码。`images` 列表中的图片顺序将和源 `image_urls` 组保持一致。如果某个图片下载失败，将会记录下错误信息，图片也不会出现在 `images` 组中。

## 使用样例

为了使用图片管道，你仅需要启动它并用 `image_urls` 和 `images` 定义一个项目：

```
import scrapy

class MyItem(scrapy.Item):

    # ... other item fields ...
    image_urls = scrapy.Field()
    images = scrapy.Field()
```

如果你需要更加复杂的功能，想重写定制图片管道行为，参见实现定制图片管道。

## 开启你的图片管道

为了开启你的图片管道，你首先需要在项目中添加它 `ITEM_PIPELINES setting`：

```
ITEM_PIPELINES = {'scrapy.contrib.pipeline.images.ImagesPipeline': 1}
```

并将 `IMAGES_STORE` 设置为一个有效的文件夹，用来存储下载的图片。否则管道将保持禁用状态，即使你在 `ITEM_PIPELINES` 设置中添加了它。

比如：

```
IMAGES_STORE = '/path/to/valid/dir'
```

# 图片存储

文件系统是当前官方唯一支持的存储系统，但也支持（非公开的）Amazon S3。

## 文件系统存储

图片存储在文件中（一个图片一个文件），并使用它们 URL 的 SHA1 hash 作为文件名。

比如，对下面的图片 URL：

```
http://www.example.com/image.jpg
```

它的 SHA1 hash 值为：

```
3afec3b4765f8f0a07b78f98c07b83f013567a0a
```

将被下载并存为下面的文件：

```
<IMAGES_STORE>/full/3afec3b4765f8f0a07b78f98c07b83f013567a0a.jpg
```

其中：

- `<IMAGES_STORE>` 是定义在 `IMAGES_STORE` 设置里的文件夹
- `full` 是用来区分图片和缩略图（如果使用的话）的一个子文件夹。详情参见缩略图生成。

## 额外的特性

### 图片失效

图像管道避免下载最近已经下载的图片。使用 `IMAGES_EXPIRES` 设置可以调整失效期限，可以用天数来指定：

```
# 90 天的图片失效期限

IMAGES_EXPIRES = 90
```

### 缩略图生成

图片管道可以自动创建下载图片的缩略图。

为了使用这个特性，你需要设置 `IMAGES_THUMBS` 字典，其关键字为缩略图名字，值为它们的大小尺寸。

比如：

```
IMAGES_THUMBS = {
    'small': (50, 50),
    'big': (270, 270),
}
```

当你使用这个特性时，图片管道将使用下面的格式来创建各个特定尺寸的缩略图：

```
<IMAGES_STORE>/thumbs/<size_name>/<image_id>.jpg
```

其中：

- `<size_name>` 是 `IMAGES_THUMBS` 字典关键字（small，big，等）
- `<image_id>` 是图像 url 的 SHA1 hash

例如使用 `small` 和 `big` 缩略图名字的图片文件：

```
<IMAGES_STORE>/full/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/small/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/big/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
```

第一个是从网站下载的完整图片。

## 滤出小图片

你可以丢掉那些过小的图片，只需在 :setting:*IMAGES_MIN_HEIGHT* 和 IMAGES_MIN_WIDTH 设置中指定最小允许的尺寸。

比如：

```
IMAGES_MIN_HEIGHT = 110
IMAGES_MIN_WIDTH = 110
```

注意：这些尺寸一点也不影响缩略图的生成。

默认情况下，没有尺寸限制，因此所有图片都将处理。

## 实现定制图片管道

下面是你可以在定制的图片管道里重写的方法：

### class scrapy.contrib.pipeline.images.ImagesPipeline

#### get_media_requests(item, info)

在工作流程中可以看到，管道会得到图片的 URL 并从项目中下载。为了这么做，你需要重写 `get_media_requests()` 方法，并对各个图片 URL 返回一个 Request：

```
def get_media_requests(self, item, info):
    for image_url in item['image_urls']:
        yield scrapy.Request(image_url)
```

这些请求将被管道处理，当它们完成下载后，结果将以 2-元素的元组列表形式传送到 `item_completed()` 方法：

- `success` 是一个布尔值，当图片成功下载时为 `True` ，因为某个原因下载失败为 `False`
- `image_info_or_error` 是一个包含下列关键字的字典（如果成功为 `True` ）或者出问题时为 Twisted Failure。
    - `url` - 图片下载的 url。这是从 `get_media_requests()` 方法返回请求的 url。
    - `path` - 图片存储的路径（类似 `IMAGES_STORE` ）
    - `checksum` - 图片内容的 MD5 hash `item_completed()` 接收的元组列表需要保证与 `get_media_requests()` 方法返回请求的顺序相一致。下面是 results 参数的一个典型值：

```
[(True,
  {'checksum': '2b00042f7481c7b056c4b410d28f33cf',
   'path': 'full/7d97e98f8af710c7e7fe703abc8f639e0ee507c4.jpg',
   'url': 'http://www.example.com/images/product1.jpg'}),
 (True,
  {'checksum': 'b9628c4ab9b595f72f280b90c4fd093d',
   'path': 'full/1ca5879492b8fd606df1964ea3c1e2f4520f076f.jpg',
   'url': 'http://www.example.com/images/product2.jpg'}),
 (False,
  Failure(...))]
```

默认 `get_media_requests()` 方法返回 `None` ，这意味着项目中没有图片可下载。

item_completed(results, items, info)

当一个单独项目中的所有图片请求完成时（要么完成下载，要么因为某种原因下载失败），`ImagesPipeline.item_completed()` 方法将被调用。

`item_completed()` 方法需要返回一个输出，其将被送到随后的项目管道阶段，因此你需要返回（或者丢弃）项目，如你在任意管道里所做的一样。

这里是一个 item_completed() 方法的例子，其中我们将下载的图片路径（传入到 results 中）存储到 image_paths 项目组中，如果其中没有图片，我们将丢弃项目：

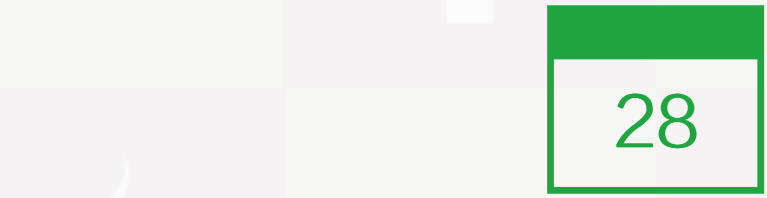
```
from scrapy.exceptions import DropItem

def item_completed(self, results, item, info):
    image_paths = [x['path'] for ok, x in results if ok]
    if not image_paths:
        raise DropItem("Item contains no images")
    item['image_paths'] = image_paths
    return item
```

默认情况下，`item_completed()` 方法返回项目。

## 定制图片管道的例子

下面是一个图片管道的完整例子，其方法如上所示：

```
import scrapy
from scrapy.contrib.pipeline.images import ImagesPipeline
from scrapy.exceptions import DropItem

class MyImagesPipeline(ImagesPipeline):

    def get_media_requests(self, item, info):
        for image_url in item['image_urls']:
            yield scrapy.Request(image_url)

    def item_completed(self, results, item, info):
        image_paths = [x['path'] for ok, x in results if ok]
        if not image_paths:
            raise DropItem("Item contains no images")
        item['image_paths'] = image_paths
        return item
```



# 28

# Ubuntu 软件包

新版功能。

Scrapinghub 发布的 apt-get 可获取版本通常比 Ubuntu 里更新，并且在比 Github 仓库 (master & stable branches)稳定的同时还包括了最新的漏洞修复。

用法：

- 把 Scrapy 签名的 GPG 密钥添加到 APT 的钥匙环中：

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 627220E7
```

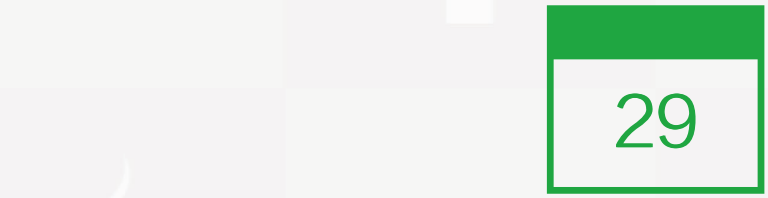
- 执行如下命令，创建/etc/apt/sources.list.d/scrapy.list 文件：

```
echo 'deb http://archive.scrapy.org/ubuntu scrapy main' | sudo tee /etc/apt/sources.list.d/scrapy.list
```

- 更新包列表并安装 scrapy-0.25:

```
sudo apt-get update && sudo apt-get install scrapy-0.25
```

> 注解
>
> 如果你要升级 Scrapy，请重复步骤 3。
>
> 警告
>
> debian 官方源提供的 python-scrapy 是一个非常老的版本且不再获得 Scrapy 团队支持。



# 29 Scrapyd

Scrapyd 被移动成为一个单独的项目。 其文档当前被托管在:

http://scrapyd.readthedocs.org/

# 30

# 自动限速(AutoThrottle)扩展

该扩展能根据 Scrapy 服务器及您爬取的网站的负载自动限制爬取速度。

## 设计目标

1. 更友好的对待网站，而不使用默认的下载延迟 0。
2. 自动调整 scrapy 来优化下载速度，使得用户不用调节下载延迟及并发请求数来找到优化的值。 用户只需指定允许的最大并发请求数，剩下的都交给扩展来完成。

## 扩展是如何实现的

在 Scrapy 中，下载延迟是通过计算建立 TCP 连接到接收到 HTTP 包头(header)之间的时间来测量的。

注意，由于 Scrapy 可能在忙着处理 spider 的回调函数或者无法下载，因此在合作的多任务环境下准确测量这些延迟是十分苦难的。 不过，这些延迟仍然是对 Scrapy(甚至是服务器)繁忙程度的合理测量，而这扩展就是以此为前提进行编写的。

## 限速算法

算法根据以下规则调整下载延迟及并发数：

1. spider 永远以 1 并发请求数及 AUTOTHROTTLE_START_DELAY 中指定的下载延迟启动。
2. 当接收到回复时，下载延迟会调整到该回复的延迟与之前下载延迟之间的平均值。

> 注解
>
> AutoThrottle 扩展尊重标准 Scrapy 设置中的并发数及延迟。这意味着其永远不会设置一个比 DOWNLOAD_DELAY 更低的下载延迟或者比 CONCURRENT_REQUESTS_PER_DOMAIN 更高的并发数 (或 CONCURRENT_REQUESTS_PER_IP，取决于您使用哪一个)。

## 设置

下面是控制 AutoThrottle 扩展的设置：

- AUTOTHROTTLE_ENABLED
- AUTOTHROTTLE_START_DELAY
- AUTOTHROTTLE_MAX_DELAY
- AUTOTHROTTLE_DEBUG
- CONCURRENT_REQUESTS_PER_DOMAIN
- CONCURRENT_REQUESTS_PER_IP
- DOWNLOAD_DELAY

更多内容请参考限速算法。

### AUTOTHROTTLE_ENABLED

默认： `False`

启用 AutoThrottle 扩展。

### AUTOTHROTTLE_START_DELAY

默认： `5.0`

初始下载延迟(单位：秒)。

### AUTOTHROTTLE_MAX_DELAY

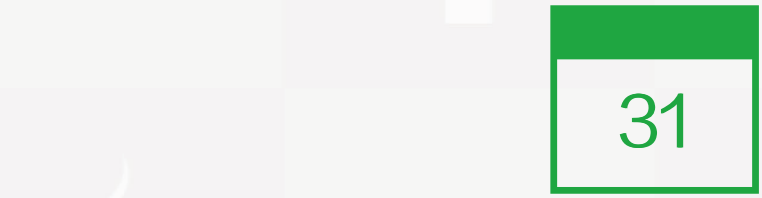
默认： `60.0`

在高延迟情况下最大的下载延迟(单位秒)。

## AUTOTHROTTLE_DEBUG

默认：`False`

起用 AutoThrottle 调试(debug)模式，展示每个接收到的 response。您可以通过此来查看限速参数是如何实时被调整的。



# 31

# Benchmarking

Scrapy 提供了一个简单的性能测试工具。其创建了一个本地 HTTP 服务器，并以最大可能的速度进行爬取。该测试性能工具目的是测试 Scrapy 在您的硬件上的效率，来获得一个基本的底线用于对比。其使用了一个简单的 spider，仅跟进链接，不做任何处理。

运行：

```
scrapy bench
```

您能看到类似的输出：

```
2013-05-16 13:08:46-0300 [scrapy] INFO: Scrapy 0.17.0 started (bot: scrapybot)
2013-05-16 13:08:47-0300 [follow] INFO: Spider opened
2013-05-16 13:08:47-0300 [follow] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:48-0300 [follow] INFO: Crawled 74 pages (at 4440 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:49-0300 [follow] INFO: Crawled 143 pages (at 4140 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:50-0300 [follow] INFO: Crawled 210 pages (at 4020 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:51-0300 [follow] INFO: Crawled 274 pages (at 3840 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:52-0300 [follow] INFO: Crawled 343 pages (at 4140 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:53-0300 [follow] INFO: Crawled 410 pages (at 4020 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:54-0300 [follow] INFO: Crawled 474 pages (at 3840 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:55-0300 [follow] INFO: Crawled 538 pages (at 3840 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:56-0300 [follow] INFO: Crawled 602 pages (at 3840 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:57-0300 [follow] INFO: Closing spider (closespider_timeout)
2013-05-16 13:08:57-0300 [follow] INFO: Crawled 666 pages (at 3840 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:57-0300 [follow] INFO: Dumping Scrapy stats:
    {'downloader/request_bytes': 231508,
     'downloader/request_count': 682,
     'downloader/request_method_count/GET': 682,
     'downloader/response_bytes': 1172802,
     'downloader/response_count': 682,
     'downloader/response_status_count/200': 682,
     'finish_reason': 'closespider_timeout',
     'finish_time': datetime.datetime(2013, 5, 16, 16, 8, 57, 985539),
     'log_count/INFO': 14,
     'request_depth_max': 34,
     'response_received_count': 682,
     'scheduler/dequeued': 682,
     'scheduler/dequeued/memory': 682,
     'scheduler/enqueued': 12767,
     'scheduler/enqueued/memory': 12767,
     'start_time': datetime.datetime(2013, 5, 16, 16, 8, 47, 676539)}
2013-05-16 13:08:57-0300 [follow] INFO: Spider closed (closespider_timeout)
```

这说明了您的 Scrapy 能以 3900 页面/分钟的速度爬取。注意，这是一个非常简单，仅跟进链接的 spider。任何您所编写的 spider 会做更多处理，从而减慢爬取的速度。减慢的程度取决于 spider 做的处理以及其是如何被编写的。

未来会有更多的用例会被加入到性能测试套装中，以覆盖更多常见的情景。

# 32

# Jobs:暂停，恢复爬虫

有些情况下，例如爬取大的站点，我们希望能暂停爬取，之后再恢复运行。

Scrapy 通过如下工具支持这个功能：

- 一个把调度请求保存在磁盘的调度器
- 一个把访问请求保存在磁盘的副本过滤器[duplicates filter]
- 一个能持续保持爬虫状态(键/值对)的扩展

## Job 路径

要启用持久化支持，你只需要通过 JOBDIR 设置 job directory 选项。这个路径将会存储 所有的请求数据来保持一个单独任务的状态(例如：一次 spider 爬取(a spider run))。必须要注意的是，这个目录不允许被不同的 spider 共享，甚至是同一个 spider 的不同 jobs/runs 也不行。也就是说，这个目录就是存储一个 单独 job 的状态信息。

## 怎么使用

要启用一个爬虫的持久化，运行以下命令：

```
scrapy crawl somespider -s JOBDIR=crawls/somespider-1
```

然后，你就能在任何时候安全地停止爬虫(按 Ctrl-C 或者发送一个信号)。恢复这个爬虫也是同样的命令：

```
scrapy crawl somespider -s JOBDIR=crawls/somespider-1
```

## 保持状态

有的时候，你希望持续保持一些运行长时间的蜘蛛的状态。这时您可以使用 spider.state 属性，该属性的类型必须是 dict。scrapy 提供了内置扩展负责在 spider 启动或结束时，从工作路径(job directory)中序列化、存储、加载属性。

下面这个例子展示了使用 spider state 的回调函数(callback)(简洁起见，省略了其他的代码)：

```
def parse_item(self, response):
    # parse item here
    self.state['items_count'] = self.state.get('items_count', 0) + 1
```

# 持久化的一些坑

如果你想要使用 Scrapy 的持久化支持,还有一些东西您需要了解：

## Cookies 的有效期

Cookies 是有有效期的(可能过期)。所以如果你没有把你的爬虫及时恢复，那么他可能在被调度回去的时候 就不能工作了。当然如果你的爬虫不依赖 cookies 就不会有这个问题了。

## 请求序列化

请求是由 *pickle* 进行序列化的，所以你需要确保你的请求是可被 pickle 序列化的。 这里最常见的问题是在在 request 回调函数中使用 `lambda` 方法，导致无法序列化。

例如，这样就会有问题：

```
def some_callback(self, response):
    somearg = 'test'
    return scrapy.Request('http://www.example.com', callback=lambda r: self.other_callback(r, somearg))

def other_callback(self, response, somearg):
    print "the argument passed is:", somearg
```

这样才对：

```
def some_callback(self, response):
    somearg = 'test'
    return scrapy.Request('http://www.example.com', meta={'somearg': somearg})

 #这里的实例代码有错，应该是(译者注)

 #   return scrapy.Request('http://www.example.com', meta={'somearg': somearg}, callback=self.other_callback)


def other_callback(self, response):
    somearg = response.meta['somearg']
    print "the argument passed is:", somearg
```

# 33

# DjangoItem

`DjangoItem` 是一个 item 的类，其从 Django 模型中获取字段(field)定义。 您可以简单地创建一个 `DjangoItem` 并指定其关联的 `Django` 模型。

除了获得您 item 中定义的字段外， DjangoItem 提供了创建并获得一个具有 item 数据的 Django 模型实例(Django model instance)的方法。

## 使用 DjangoItem

DjangoItem 使用方法与 Django 中的 ModelForms 类似。您创建一个子类，并定义其 django_model 属性。这样，您就可以得到一个字段与 Django 模型字段(model field)一一对应的 item 了。

另外，您可以定义模型中没有的字段，甚至是覆盖模型中已经定义的字段。

让我们来看个例子：

创造一个 Django 模型：

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=255)
    age = models.IntegerField()
```

定义一个基本的 `DjangoItem`：

```
from scrapy.contrib.djangoitem import DjangoItem

class PersonItem(DjangoItem):
    django_model = Person
```

`DjangoItem` 的使用方法和 `Item` 类似：

```
>>> p = PersonItem()
>>> p['name'] = 'John'
>>> p['age'] = '22'
```

要从 item 中获取 Django 模型，调用 `DjangoItem` 中额外的方法 `save()`：

```
>>> person = p.save()
>>> person.name
'John'
>>> person.age
'22'
>>> person.id
1
```

当我们调用 `save()` 时，模型已经保存了。我们可以在调用时带上 `commit=False` 来避免保存， 并获取到一个未保存的模型：

```
>>> person = p.save(commit=False)
>>> person.name
'John'
>>> person.age
'22'
>>> person.id
None
```

正如之前所说的，我们可以在 item 中加入字段：

```
import scrapy
from scrapy.contrib.djangoitem import DjangoItem

class PersonItem(DjangoItem):
    django_model = Person
    sex = scrapy.Field()
```

```
>>> p = PersonItem()
>>> p['name'] = 'John'
>>> p['age'] = '22'
>>> p['sex'] = 'M'
```

> 注解
>
> 当执行 save()时添加到 item 的字段不会有作用(taken into account)。

并且我们可以覆盖模型中的字段：

```
class PersonItem(DjangoItem):
    django_model = Person
    name = scrapy.Field(default='No Name')
```

这在提供字段属性时十分有用，例如您项目中使用的默认或者其他属性一样。

## DjangoItem 注意事项

DjangoItem 提供了在 Scrapy 项目中集成 DjangoItem 的简便方法，不过需要注意的是，如果在 Scrapy 中爬取大量(百万级)的 item 时，Django ORM 扩展得并不是很好(not scale well)。这是因为关系型后端对于一个密集型(intensive)应用(例如 web 爬虫)并不是一个很好的选择，尤其是具有大量的索引的数据库。

## 配置 Django 的设置

在 Django 应用之外使用 Django 模型(model)，您需要设置 `DJANGO_SETTINGS_MODULE` 环境变量以及 - 大多数情况下 - 修改 `PYTHONPATH` 环境变量来导入设置模块。

完成这个配置有很多方法，具体选择取决您的情况及偏好。 下面详细给出了完成这个配置的最简单方法。

假设您项目的名称为 `mysite` ，位于 `/home/projects/mysite` 且用 `Person` 模型创建了一个应用 `myapp` 。 这意味着您的目录结构类似于：

```
/home/projects/mysite
 ├─── manage.py
 ├─── myapp
 |    ├─── __init__.py
 |    ├─── models.py
 |    ├─── tests.py
 |    └─── views.py
 └─── mysite
    ├─── __init__.py
    ├─── settings.py
    ├─── urls.py
    └─── wsgi.py
```

接着您需要将 `/home/projects/mysite` 加入到 `PYTHONPATH` 环境变量中并将 `mysite.settings` 设置为 `DJANGO_SETTINGS_MODULE` 环境变量。 这可以在 Scrapy 设置文件中添加下列代码：

```
import sys
sys.path.append('/home/projects/mysite')

import os
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
```

注意，由于我们在 python 运行环境中，所以我们修改 `sys.path` 变量而不是 `PYTHONPATH` 环境变量。 如果所有设置正确，您应该可以运行 `scrapy shell` 命令并且导入 `Person` 模型(例如 `from myapp.models import Person` )。

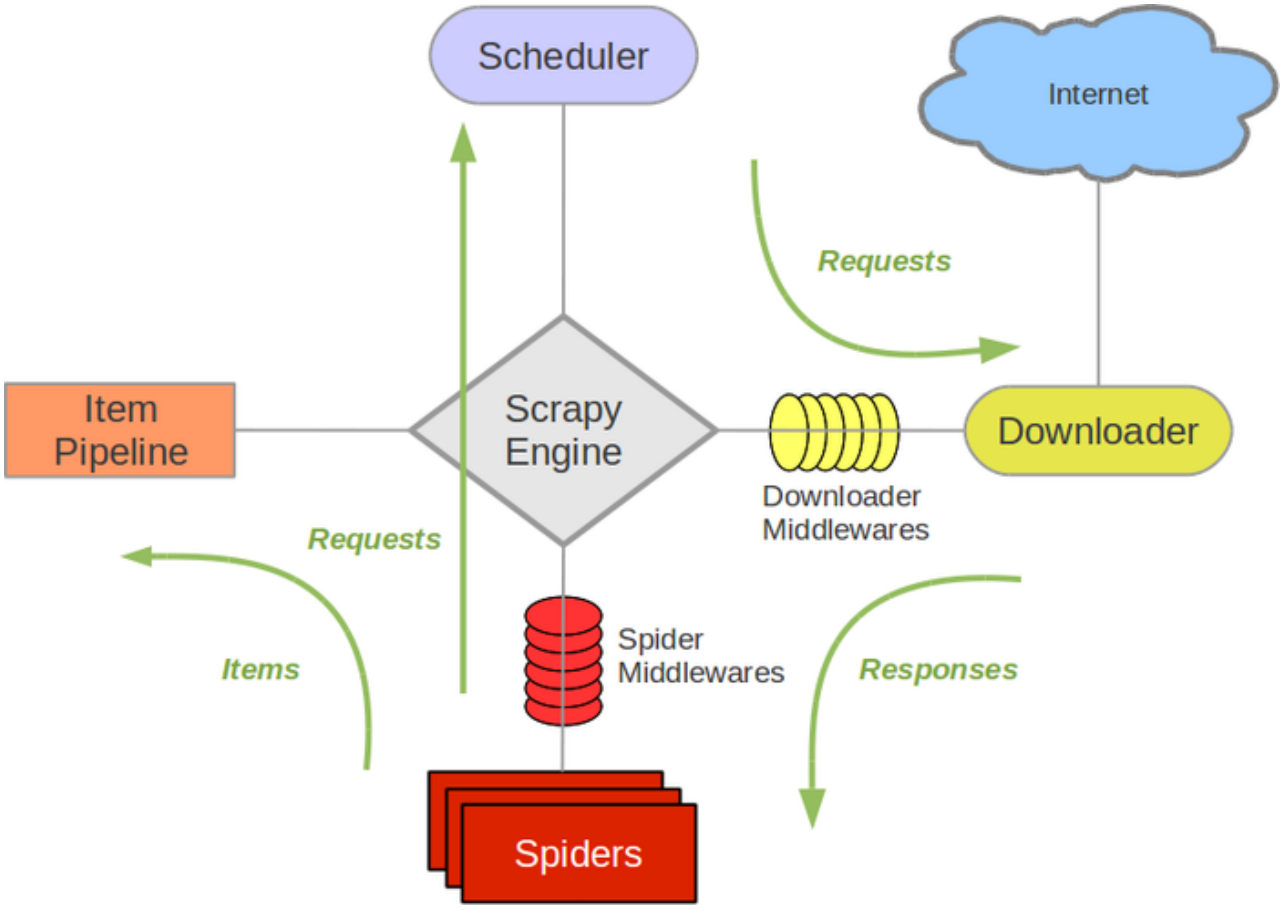
# 34

# 架构概览

本文档介绍了 Scrapy 架构及其组件之间的交互。

## 概述

接下来的图表展现了 Scrapy 的架构，包括组件及在系统中发生的数据流的概览(绿色箭头所示)。下面对每个组件都做了简单介绍，并给出了详细内容的链接。数据流如下所描述。



# 组件

## Scrapy Engine

引擎负责控制数据流在系统中所有组件中流动，并在相应动作发生时触发事件。 详细内容查看下面的数据流(Data Flow)部分。

## 调度器(Scheduler)

调度器从引擎接受 request 并将他们入队，以便之后引擎请求他们时提供给引擎。

## 下载器(Downloader)

下载器负责获取页面数据并提供给引擎，而后提供给 spider。

## Spiders

Spider 是 Scrapy 用户编写用于分析 response 并提取 item(即获取到的 item)或额外跟进的 URL 的类。 每个 spider 负责处理一个特定(或一些)网站。更多内容请看 Spiders。

## Item Pipeline

Item Pipeline 负责处理被 spider 提取出来的 item。典型的处理有清理、验证及持久化(例如存取到数据库中)。更多内容查看 Item Pipeline。

## 下载器中间件(Downloader middlewares)

下载器中间件是在引擎及下载器之间的特定钩子(specific hook)，处理 Downloader 传递给引擎的 response。其提供了一个简便的机制，通过插入自定义代码来扩展 Scrapy 功能。更多内容请看下载器中间件(Downloader Middleware)。

## Spider 中间件(Spider middlewares)

Spider 中间件是在引擎及 Spider 之间的特定钩子(specific hook)，处理 spider 的输入(response)和输出(items 及 requests)。其提供了一个简便的机制，通过插入自定义代码来扩展 Scrapy 功能。更多内容请看 Spider 中间件(Middleware)。

## 数据流(Data flow)

Scrapy 中的数据流由执行引擎控制，其过程如下:

1. 引擎打开一个网站(open a domain)，找到处理该网站的 Spider 并向该 spider 请求第一个要爬取的 URL(s)。
2. 引擎从 Spider 中获取到第一个要爬取的 URL 并在调度器(Scheduler)以 Request 调度。
3. 引擎向调度器请求下一个要爬取的 URL。
4. 调度器返回下一个要爬取的 URL 给引擎，引擎将 URL 通过下载中间件(请求(request)方向)转发给下载器(Downloader)。
5. 一旦页面下载完毕，下载器生成一个该页面的 Response，并将其通过下载中间件(返回(response)方向)发送给引擎。
6. 引擎从下载器中接收到 Response 并通过 Spider 中间件(输入方向)发送给 Spider 处理。
7. Spider 处理 Response 并返回爬取到的 Item 及(跟进的)新的 Request 给引擎。
8. 引擎将(Spider 返回的)爬取到的 Item 给 Item Pipeline，将(Spider 返回的)Request 给调度器。
9. (从第二步)重复直到调度器中没有更多地 request，引擎关闭该网站。

## 事件驱动网络(Event-driven networking)

Scrapy 基于事件驱动网络框架 Twisted 编写。因此，Scrapy 基于并发性考虑由非阻塞(即异步)的实现。

关于异步编程及 Twisted 更多的内容请查看下列链接：

- Introduction to Deferreds in Twisted
- Twisted - hello, asynchronous programming

35

# 下载器中间件(Downloader Middleware)

下载器中间件是介于 Scrapy 的 request/response 处理的钩子框架。 是用于全局修改 Scrapy request 和 response 的一个轻量、底层的系统。

## 激活下载器中间件

要激活下载器中间件组件，将其加入到 DOWNLOADER_MIDDLEWARES 设置中。该设置是一个字典(dict)，键为中间件类的路径，值为其中间件的顺序(order)。

这里是一个例子：

```
DOWNLOADER_MIDDLEWARES = {
    'myproject.middlewares.CustomDownloaderMiddleware': 543,
}
```

DOWNLOADER_MIDDLEWARES 设置会与 Scrapy 定义的 DOWNLOADER_MIDDLEWARES_BASE 设置合并(但不是覆盖)，而后根据顺序(order)进行排序，最后得到启用中间件的有序列表: 第一个中间件是最靠近引擎的，最后一个中间件是最靠近下载器的。

关于如何分配中间件的顺序请查看 DOWNLOADER_MIDDLEWARES_BASE 设置，而后根据您想要放置中间件的位置选择一个值。由于每个中间件执行不同的动作，您的中间件可能会依赖于之前(或者之后)执行的中间件，因此顺序是很重要的。

如果您想禁止内置的(在 DOWNLOADER_MIDDLEWARES_BASE 中设置并默认启用的)中间件，您必须在项目的 DOWNLOADER_MIDDLEWARES 设置中定义该中间件，并将其值赋为 *None*。例如，如果您想要关闭 user-agent 中间件：

```
DOWNLOADER_MIDDLEWARES = {
    'myproject.middlewares.CustomDownloaderMiddleware': 543,
    'scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware': None,
}
```

最后，请注意，有些中间件需要通过特定的设置来启用。更多内容请查看相关中间件文档。

## 编写您自己的下载器中间件

编写下载器中间件十分简单。每个中间件组件是一个定义了以下一个或多个方法的 Python 类：

### class scrapy.contrib.downloadermiddleware.DownloaderMiddleware

process_request(request, spider)
当每个 request 通过下载中间件时，该方法被调用。

process_request() 必须返回其中之一： 返回 `None` 、返回一个 Response 对象、返回一个 Request 对象或 raise IgnoreRequest。

如果其返回 `None` ，Scrapy 将继续处理该 request，执行其他的中间件的相应方法，直到合适的下载器处理函数(download handler)被调用，该 request 被执行(其 response 被下载)。

如果其返回 Response 对象，Scrapy 将不会调用 任何 其他的 process_request()或 process_exception()方法，或相应地下载函数； 其将返回该 response。已安装的中间件的 process_response()方法则会在每个 response 返回时被调用。

如果其返回 Request 对象，Scrapy 则停止调用 process_request 方法并重新调度返回的 request。当新返回的 request 被执行后， 相应地中间件链将会根据下载的 response 被调用。

如果其 raise 一个 IgnoreRequest 异常，则安装的下载中间件的 process_exception() 方法会被调用。如果没有任何一个方法处理该异常， 则 request 的 errback( `Request.errback` )方法会被调用。如果没有代码处理抛出的异常， 则该异常被忽略且不记录(不同于其他异常那样)。

**参数：**

- request ( `Request` 对象) – 处理的 request
- spider ( `Spider` 对象) – 该 request 对应的 spider

### process_response(request, response, spider)

process_request() 必须返回以下之一： 返回一个 Response 对象、返回一个 Request 对象或 raise 一个 IgnoreRequest 异常。

如果其返回一个 Response (可以与传入的 response 相同，也可以是全新的对象) 该 response 会被在链中的其他中间件的 process_response()方法处理。

如果其返回一个 Request 对象，则中间件链停止，返回的 request 会被重新调度下载。处理类似于 process_request()返回 request 所做的那样。

如果其抛出一个 IgnoreRequest 异常，则调用 request 的 errback( `Request.errback` )。如果没有代码处理抛出的异常，则该异常被忽略且不记录(不同于其他异常那样)。

参数:

- request ( `Request` 对象) – response 所对应的 request
- response ( `Response` 对象) – 被处理的 response
- spider ( `Spider` 对象) – response 所对应的 spider

process_exception(request, exception, spider)

当下载处理器(download handler)或 process_request()(下载中间件)抛出异常(包括 IgnoreRequest 异常)时，Scrapy 调用 process_exception() 。

process_exception()应该返回以下之一： 返回 `None` 、一个 Response 对象、或者一个 Request 对象。

如果其返回 `None` ，Scrapy 将会继续处理该异常，接着调用已安装的其他中间件的 process_exception()方法，直到所有中间件都被调用完毕，则调用默认的异常处理。

如果其返回一个 Response 对象，则已安装的中间件链的 process_response()方法被调用。Scrapy 将不会调用任何其他中间件的 process_exception() 方法。

如果其返回一个 Request 对象， 则返回的 request 将会被重新调用下载。这将停止中间件的 process_exception()方法执行，就如返回一个 response 的那样。

参数:

- **request **(是 `Request` 对象) – 产生异常的 request
- **exception **( `Exception` 对象) – 抛出的异常
- **spider **( `Spider` 对象) – request 对应的 spider

# 内置下载中间件参考手册

本页面介绍了 Scrapy 自带的所有下载中间件。关于如何使用及编写您自己的中间件，请参考 downloader middleware usage guide。

关于默认启用的中间件列表(及其顺序)请参考 `DOWNLOADER_MIDDLEWARES_BASE` 设置。

## CookiesMiddleware

class scrapy.contrib.downloadermiddleware.cookies.CookiesMiddleware

该中间件使得爬取需要 cookie(例如使用 session)的网站成为了可能。其追踪了 web server 发送的 cookie，并在之后的 request 中发送回去，就如浏览器所做的那样。

以下设置可以用来配置 cookie 中间件：

- COOKIES_ENABLED
- COOKIES_DEBUG

## 单 spider 多 cookie session

Scrapy 通过使用 `cookiejar` Request meta key 来支持单 spider 追踪多 cookie session。 默认情况下其使用一个 cookie jar(session)，不过您可以传递一个标示符来使用多个。

例如：

```
for i, url in enumerate(urls):
    yield scrapy.Request("http://www.example.com", meta={'cookiejar': i},
        callback=self.parse_page)
```

需要注意的是 cookiejar meta key 不是”黏性的(sticky)”。您需要在之后的 request 请求中接着传递。例如：

```
def parse_page(self, response):
    # do some processing
    return scrapy.Request("http://www.example.com/otherpage",
        meta={'cookiejar': response.meta['cookiejar']},
        callback=self.parse_other_page)
```

## COOKIES_ENABLED

默认： `True`

是否启用 cookies middleware。如果关闭，cookies 将不会发送给 web server。

## COOKIES_DEBUG

默认： `False`

如果启用，Scrapy 将记录所有在 request(Cookie 请求头)发送的 cookies 及 response 接收到的 cookies(Set-Cookie 接收头)。

下边是启用 `COOKIES_DEBUG` 的记录的样例：

```
2011-04-06 14:35:10-0300 [diningcity] INFO: Spider opened
2011-04-06 14:35:10-0300 [diningcity] DEBUG: Sending cookies to: <GET http://www.diningcity.com/netherlands/index.h
    Cookie: clientlanguage_nl=en_EN
2011-04-06 14:35:14-0300 [diningcity] DEBUG: Received cookies from: <200 http://www.diningcity.com/netherlands/inde
    Set-Cookie: JSESSIONID=B~FA4DC0C496C8762AE4F1A620EAB34F38; Path=/
    Set-Cookie: ip_isocode=US
    Set-Cookie: clientlanguage_nl=en_EN; Expires=Thu, 07-Apr-2011 21:21:34 GMT; Path=/
2011-04-06 14:49:50-0300 [diningcity] DEBUG: Crawled (200) <GET http://www.diningcity.com/netherlands/index.html>
[...]
```

## DefaultHeadersMiddleware

class scrapy.contrib.downloadermiddleware.defaultheaders.DefaultHeadersMiddleware

该中间件设置 `DEFAULT_REQUEST_HEADERS` 指定的默认 request header。

## DownloadTimeoutMiddleware

class scrapy.contrib.downloadermiddleware.downloadtimeout.DownloadTimeoutMiddleware

该中间件设置 `DOWNLOAD_TIMEOUT` 或 spider 的 `download_timeout` 属性指定的 request 下载超时时间。

> 注解

> 您也可以使用 `download_timeout` Request.meta key 来对每个请求设置下载超时时间。这种方式在 DownloadTimeoutMiddleware 被关闭时仍然有效。

## HttpAuthMiddleware

class scrapy.contrib.downloadermiddleware.httpauth.HttpAuthMiddleware

该中间件完成某些使用 Basic access authentication (或者叫 HTTP 认证)的 spider 生成的请求的认证过程。

在 spider 中启用 HTTP 认证，请设置 spider 的 `http_user` 及 `http_pass` 属性。

样例:

```
from scrapy.contrib.spiders import CrawlSpider

class SomeIntranetSiteSpider(CrawlSpider):

    http_user = 'someuser'
    http_pass = 'somepass'
    name = 'intranet.example.com'

    # .. rest of the spider code omitted ...
```

## HttpCacheMiddleware

class scrapy.contrib.downloadermiddleware.httpcache.HttpCacheMiddleware

该中间件为所有 HTTP request 及 response 提供了底层(low-level)缓存支持。其由 cache 存储后端及 cache 策略组成。

Scrapy 提供了两种 HTTP 缓存存储后端:

- Filesystem storage backend (默认值)
- DBM storage backend

您可以使用 `HTTPCACHE_STORAGE` 设定来修改 HTTP 缓存存储后端。您也可以实现您自己的存储后端。

Scrapy 提供了两种了缓存策略:

- RFC2616 策略

- Dummy 策略(默认值)

您可以使用 `HTTPCACHE_POLICY` 设定来修改 HTTP 缓存存储后端。您也可以实现您自己的存储策略。

## Dummy 策略(默认值)

该策略不考虑任何 HTTP Cache-Control 指令。每个 request 及其对应的 response 都被缓存。 当相同的 request 发生时，其不发送任何数据，直接返回 response。

Dummpy 策略对于测试 spider 十分有用。其能使 spider 运行更快(不需要每次等待下载完成)， 同时在没有网络连接时也能测试。其目的是为了能够回放 spider 的运行过程，*使之与之前的运行过程一模一样*。

使用这个策略请设置:

- `HTTPCACHE_POLICY` 为 `scrapy.contrib.httpcache.DummyPolicy`

## RFC2616 策略

该策略提供了符合 RFC2616 的 HTTP 缓存，例如符合 HTTP Cache-Control，针对生产环境并且应用在持续性运行环境所设置。该策略能避免下载未修改的数据(来节省带宽，提高爬取速度)。

实现了:

- 当 no-store cache-control 指令设置时不存储 response/request。
- 当 no-cache cache-control 指定设置时不从 cache 中提取 response，即使 response 为最新。
- 根据 max-age cache-control 指令中计算保存时间(freshness lifetime)。
- 根据 Expires 指令来计算保存时间(freshness lifetime)。
- 根据 response 包头的 *Last-Modified* 指令来计算保存时间(freshness lifetime)。(Firefox 使用的启发式算法)。
- 根据 response 包头的 *Age* 计算当前年龄(current age)。
- 根据 *Date* 计算当前年龄(current age)。
- 根据 response 包头的 *Last-Modified* 验证老旧的 response。
- 根据 response 包头的 *ETag* 验证老旧的 response。
- 为接收到的 response 设置缺失的 *Date* 字段。

目前仍然缺失:

- Pragma: no-cache 支持 http://www.mnot.net/cache_docs/#PRAGMA
- Vary 字段支持 http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.6
- 当 update 或 delete 之后失效相应的 response http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.10
- ... 以及其他可能缺失的特性 ..

使用这个策略，设置：

- `HTTPCACHE_POLICY` 为 `scrapy.contrib.httpcache.RFC2616Policy`

## Filesystem storage backend (默认值)

文件系统存储后端可以用于 HTTP 缓存中间件。

使用该存储端，设置：

- `HTTPCACHE_STORAGE` 为 `scrapy.contrib.httpcache.FilesystemCacheStorage`

每个 request/response 组存储在不同的目录中，包含下列文件：

- `request_body` - the plain request body
- `request_headers` - the request headers (原始 HTTP 格式)
- `response_body` - the plain response body
- `response_headers` - the request headers (原始 HTTP 格式)
- `meta` - 以 Python `repr()` 格式(grep-friendly 格式)存储的该缓存资源的一些元数据。
- `pickled_meta` - 与 `meta` 相同的元数据，不过使用 pickle 来获得更高效的反序列化性能。

目录的名称与 request 的指纹(参考 `scrapy.utils.request.fingerprint` )有关，而二级目录是为了避免在同一文件夹下有太多文件 (这在很多文件系统中是十分低效的)。目录的例子：

```
/path/to/cache/dir/example.com/72/72811f648e718090f041317756c03adb0ada46c7
```

## DBM storage backend

同时也有 DBM 存储后端可以用于 HTTP 缓存中间件。

默认情况下，其采用 anydbm 模块，不过您也可以通过 `HTTPCACHE_DBM_MODULE` 设置进行修改。

使用该存储端，设置：

`HTTPCACHE_STORAGE 为 scrapy.contrib.httpcache.DbmCacheStorage`

## LevelDB storage backend

A LevelDB storage backend is also available for the HTTP cache middleware.

This backend is not recommended for development because only one process can access LevelDB databases at the same time, so you can’t run a crawl and open the scrapy shell in parallel for the same spider.

In order to use this storage backend:

- set `HTTPCACHE_STORAGE` to `scrapy.contrib.httpcache.LeveldbCacheStorage`
- install `LevelDB python bindings` like `pip install leveldb`

## HTTPCache 中间件设置

`HttpCacheMiddleware` 可以通过以下设置进行配置：

## HTTPCACHE_ENABLED

新版功能。

默认： `False`

HTTP 缓存是否开启。

在 0.11 版更改: 在 0.11 版本前，是使用 `HTTPCACHE_DIR` 来开启缓存。

## HTTPCACHE_EXPIRATION_SECS

默认： `0`

缓存的 request 的超时时间，单位秒。

超过这个时间的缓存 request 将会被重新下载。如果为 0，则缓存的 request 将永远不会超时。

在 0.11 版更改： 在 0.11 版本前，0 的意义是缓存的 request 永远超时。

## HTTPCACHE_DIR

默认： `'httpcache'`

存储(底层的)HTTP 缓存的目录。如果为空，则 HTTP 缓存将会被关闭。 如果为相对目录，则相对于项目数据目录(project data dir)。更多内容请参考默认的 Scrapy 项目结构。

## HTTPCACHE_IGNORE_HTTP_CODES

新版功能。

默认： `[]`

不缓存设置中的 HTTP 返回值(code)的 request。

## HTTPCACHE_IGNORE_MISSING

默认： `False`

如果启用，在缓存中没找到的 request 将会被忽略，不下载。

## HTTPCACHE_IGNORE_SCHEMES

新版功能。

默认： `['file']`

不缓存这些 URI 标准(scheme)的 response。

## HTTPCACHE_STORAGE

默认： `'scrapy.contrib.httpcache.FilesystemCacheStorage'`

实现缓存存储后端的类。

## HTTPCACHE_DBM_MODULE

新版功能。

默认： `'anydbm'`

在 `DBM 存储后端` 的数据库模块。 该设定针对 DBM 后端。

## HTTPCACHE_POLICY

新版功能。

默认： `'scrapy.contrib.httpcache.DummyPolicy'`

实现缓存策略的类。

## HttpCompressionMiddleware

class scrapy.contrib.downloadermiddleware.httpcompression.HttpCompressionMiddleware

该中间件提供了对压缩(gzip, deflate)数据的支持。

## HttpCompressionMiddleware Settings

## COMPRESSION_ENABLED

默认： `True`

Compression Middleware(压缩中间件)是否开启。

## ChunkedTransferMiddleware

class scrapy.contrib.downloadermiddleware.chunked.ChunkedTransferMiddleware

该中间件添加了对 chunked transfer encoding 的支持。

## HttpProxyMiddleware

新版功能。

class scrapy.contrib.downloadermiddleware.httpproxy.HttpProxyMiddleware

该中间件提供了对 request 设置 HTTP 代理的支持。您可以通过在 `Request` 对象中设置 `proxy` 元数据来开启代理。

类似于 Python 标准库模块 urllib 及 urllib2，其使用了下列环境变量:

- http_proxy
- https_proxy
- no_proxy

## RedirectMiddleware

class scrapy.contrib.downloadermiddleware.redirect.RedirectMiddleware

该中间件根据 response 的状态处理重定向的 request。

通过该中间件的(被重定向的)request 的 url 可以通过 `Request.meta` 的 `redirect_urls` 键找到。

`RedirectMiddleware` 可以通过下列设置进行配置(更多内容请参考设置文档):

- REDIRECT_ENABLED
- REDIRECT_MAX_TIMES

如果 `Request.meta` 中 `dont_redirect` 设置为 True，则该 request 将会被此中间件忽略。

## RedirectMiddleware settings

## REDIRECT_ENABLED

新版功能。

默认： `True`

是否启用 Redirect 中间件。

## REDIRECT_MAX_TIMES

默认： `20`

单个 request 被重定向的最大次数。

## MetaRefreshMiddleware

class scrapy.contrib.downloadermiddleware.redirect.MetaRefreshMiddleware

该中间件根据 meta-refresh html 标签处理 request 重定向。

`MetaRefreshMiddleware` 可以通过以下设定进行配置 (更多内容请参考设置文档)。

- METAREFRESH_ENABLED
- METAREFRESH_MAXDELAY

该中间件遵循 `RedirectMiddleware` 描述的 `REDIRECT_MAX_TIMES` 设定， `dont_redirect` 及 redirect_urls meta key。

## MetaRefreshMiddleware settings

## METAREFRESH_ENABLED

新版功能。

默认： `True`

Meta Refresh 中间件是否启用。

## REDIRECT_MAX_METAREFRESH_DELAY

默认： `100`

跟进重定向的最大 meta-refresh 延迟(单位：秒)。

## RetryMiddleware

class scrapy.contrib.downloadermiddleware.retry.RetryMiddleware

该中间件将重试可能由于临时的问题，例如连接超时或者 HTTP 500 错误导致失败的页面。

爬取进程会收集失败的页面并在最后，spider 爬取完所有正常(不失败)的页面后重新调度。 一旦没有更多需要重试的失败页面，该中间件将会发送一个信号(retry_complete)， 其他插件可以监听该信号。

`RetryMiddleware` 可以通过下列设定进行配置 (更多内容请参考设置文档)：

- RETRY_ENABLED
- RETRY_TIMES
- RETRY_HTTP_CODES

关于 HTTP 错误的考虑：

如果根据 HTTP 协议，您可能想要在设定 `RETRY_HTTP_CODES` 中移除 400 错误。该错误被默认包括是由于这个代码经常被用来指示服务器过载(overload)了。而在这种情况下，我们想进行重试。

如果 `Request.meta` 中 `dont_retry` 设为 True，该 request 将会被本中间件忽略。

## RetryMiddleware Settings

## RETRY_ENABLED

新版功能。

默认： `True`

Retry Middleware 是否启用。

## RETRY_TIMES

默认： `2`

包括第一次下载，最多的重试次数

## RETRY_HTTP_CODES

默认： `[500, 502, 503, 504, 400, 408]`

重试的 response 返回值(code)。其他错误(DNS 查找问题、连接失败及其他)则一定会进行重试。

## RobotsTxtMiddleware

class scrapy.contrib.downloadermiddleware.robotstxt.RobotsTxtMiddleware

该中间件过滤所有 robots.txt eclusion standard 中禁止的 request。

确认该中间件及 `ROBOTSTXT_OBEY` 设置被启用以确保 Scrapy 尊重 robots.txt。

> 警告
>
> 记住，如果您在一个网站中使用了多个并发请求，Scrapy 仍然可能下载一些被禁止的页面。这是由于这些页面是在 robots.txt 被下载前被请求的。这是当前 robots.txt 中间件已知的限制，并将在未来进行修复。

## DownloaderStats

class scrapy.contrib.downloadermiddleware.stats.DownloaderStats

保存所有通过的 request、response 及 exception 的中间件。

您必须启用 `DOWNLOADER_STATS` 来启用该中间件。

## UserAgentMiddleware

class scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware

用于覆盖 spider 的默认 user agent 的中间件。

要使得 spider 能覆盖默认的 user agent，其 user_agent 属性必须被设置。

## AjaxCrawlMiddleware

class scrapy.contrib.downloadermiddleware.ajaxcrawl.AjaxCrawlMiddleware

根据 meta-fragment html 标签查找 ‘AJAX 可爬取’ 页面的中间件。查看 https://developers.google.com/webmasters/ajax-crawling/docs/getting-started 来获得更多内容。

> 注解
>
> 即使没有启用该中间件，Scrapy 仍能查找类似于 `'http://example.com/!#foo=bar'` 这样的’AJAX 可爬取’页面。AjaxCrawlMiddleware 是针对不具有 `'!#'` 的 URL，通常发生在’index’或者’main’页面中。

## AjaxCrawlMiddleware 设置

## AJAXCRAWL_ENABLED

新版功能。

默认： `False`

AjaxCrawlMiddleware 是否启用。您可能需要针对通用爬虫启用该中间件。

# 36

# Spider 中间件(Middleware)

Spider 中间件(Middleware) 下载器中间件是介入到 Scrapy 的 spider 处理机制的钩子框架，您可以添加代码来处理发送给 Spiders 的 response 及 spider 产生的 item 和 request。

## 激活 spider 中间件

要启用 spider 中间件，您可以将其加入到 SPIDER_MIDDLEWARES 设置中。该设置是一个字典，键位中间件的路径，值为中间件的顺序(order)。

样例:

```
SPIDER_MIDDLEWARES = {
    'myproject.middlewares.CustomSpiderMiddleware': 543,
}
```

SPIDER_MIDDLEWARES 设置会与 Scrapy 定义的 SPIDER_MIDDLEWARES_BASE 设置合并(但不是覆盖)， 而后根据顺序(order)进行排序，最后得到启用中间件的有序列表: 第一个中间件是最靠近引擎的，最后一个中间件是最靠近 spider 的。

关于如何分配中间件的顺序请查看 SPIDER_MIDDLEWARES_BASE 设置，而后根据您想要放置中间件的位置选择一个值。 由于每个中间件执行不同的动作，您的中间件可能会依赖于之前(或者之后)执行的中间件，因此顺序是很重要的。

如果您想禁止内置的(在 SPIDER_MIDDLEWARES_BASE 中设置并默认启用的)中间件，您必须在项目的 SPIDER_MIDDLEWARES 设置中定义该中间件，并将其值赋为 None。例如，如果您想要关闭 off-site 中间件:

```
SPIDER_MIDDLEWARES = {
    'myproject.middlewares.CustomSpiderMiddleware': 543,
    'scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware': None,
}
```

最后，请注意，有些中间件需要通过特定的设置来启用。更多内容请查看相关中间件文档。

## 编写您自己的 spider 中间件

编写 spider 中间件十分简单。每个中间件组件是一个定义了以下一个或多个方法的 Python 类:

### class scrapy.contrib.spidermiddleware.SpiderMiddleware

process_spider_input(response, spider)
当 response 通过 spider 中间件时，该方法被调用，处理该 response。

`process_spider_input()` 应该返回 None 或者抛出一个异常。

如果其返回 `None` ，Scrapy 将会继续处理该 response，调用所有其他的中间件直到 spider 处理该 response。

如果其跑出一个异常(exception)，Scrapy 将不会调用任何其他中间件的 `process_spider_input()` 方法，并调用 request 的 errback。 errback 的输出将会以另一个方向被重新输入到中间件链中，使用 `process_spider_output()` 方法来处理，当其抛出异常时则带调用 `process_spider_exception()` 。

**参数:**

- response ( `Response` 对象) - 被处理的 response
- spider ( `Spider` 对象) - 该 response 对应的 spider

process_spider_output(response, result, spider)
当 Spider 处理 response 返回 result 时，该方法被调用。

`process_spider_output()` 必须返回包含 `Request` 或 `Item` 对象的可迭代对象(iterable)。

**参数:**

- response ( `Response` 对象) - 生成该输出的 response
- result (包含 `Request` 或 `Item` 对象的可迭代对象(iterable)) - spider 返回的 result
- spider ( `Spider` 对象) - 其结果被处理的 spider

process_spider_exception(response, exception, spider)
当 spider 或(其他 spider 中间件的) `process_spider_input()` 跑出异常时， 该方法被调用。

`process_spider_exception()` 必须要么返回 `None` ， 要么返回一个包含 `Response` 或 `Item` 对象的可迭代对象(iterable)。

如果其返回 None ，Scrapy 将继续处理该异常，调用中间件链中的其他中间件的 process_spider_exception() 方法，直到所有中间件都被调用，该异常到达引擎(异常将被记录并被忽略)。

如果其返回一个可迭代对象，则中间件链的 process_spider_output() 方法被调用， 其他的 process_spider_exception() 将不会被调用。

参数:

- response ( Response 对象) - 异常被抛出时被处理的 response
- exception ( Exception 对象) - 被跑出的异常
- spider ( Spider 对象) - 抛出该异常的 spider

process_start_requests(start_requests, spider)
该方法以 spider 启动的 request 为参数被调用，执行的过程类似于 process_spider_output() ，只不过其没有相关联的 response 并且必须返回 request(不是 item)。

其接受一个可迭代的对象(start_requests 参数)且必须返回另一个包含 Request 对象的可迭代对象。

> 注解
>
> 当在您的 spider 中间件实现该方法时， 您必须返回一个可迭代对象(类似于参数 start_requests)且不要遍历所有的 start_requests 。该迭代器会很大(甚至是无限)，进而导致内存溢出。Scrapy 引擎在其具有能力处理 start request 时将会拉起 request，因此 start request 迭代器会变得无限，而由其他参数来停止 spider( 例如时间限制或者 item/page 记数)。

**参数: **

- start_requests (包含 Request 的可迭代对象) - start requests
- spider ( Spider 对象) - start requests 所属的 spider

# 内置 spider 中间件参考手册

本页面介绍了 Scrapy 自带的所有 spider 中间件。关于如何使用及编写您自己的中间件，请参考 spider middleware usage guide。

关于默认启用的中间件列表(及其顺序)请参考 `SPIDER_MIDDLEWARES_BASE` 设置。

## DepthMiddleware

class scrapy.contrib.spidermiddleware.depth.DepthMiddleware

DepthMiddleware 是一个用于追踪每个 Request 在被爬取的网站的深度的中间件。 其可以用来限制爬取深度的最大深度或类似的事情。

`DepthMiddleware` 可以通过下列设置进行配置(更多内容请参考设置文档):

- `DEPTH_LIMIT` – 爬取所允许的最大深度，如果为 0，则没有限制。
- `DEPTH_STATS` – 是否收集爬取状态。
- `DEPTH_PRIORITY` – 是否根据其深度对 requet 安排优先级

## HttpErrorMiddleware

class scrapy.contrib.spidermiddleware.httperror.HttpErrorMiddleware

过滤出所有失败(错误)的 HTTP response，因此 spider 不需要处理这些 request。 处理这些 request 意味着消耗更多资源，并且使得 spider 逻辑更为复杂。

根据 HTTP 标准，返回值为 200-300 之间的值为成功的 resonse。

如果您想处理在这个范围之外的 response，您可以通过 `spider 的 handle_httpstatus_list` 属性或 `HTTPERROR_ALLOWED_CODES` 设置来指定 spider 能处理的 response 返回值。

例如，如果您想要处理返回值为 404 的 response 您可以这么做:

```
class MySpider(CrawlSpider):
    handle_httpstatus_list = [404]
```

`Request.meta` 中的 `handle_httpstatus_list` 键也可以用来指定每个 request 所允许的 response code。

不过请记住，除非您知道您在做什么，否则处理非 200 返回一般来说是个糟糕的决定。

更多内容请参考:HTTP Status Code 定义。

HttpErrorMiddleware settings

## HTTPERROR_ALLOWED_CODES

默认: `[]`

忽略该列表中所有非 200 状态码的 response。

## HTTPERROR_ALLOW_ALL

默认: `False`

忽略所有 response，不管其状态值。

## OffsiteMiddleware

class scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware

过滤出所有 URL 不由该 spider 负责的 Request。

该中间件过滤出所有主机名不在 spider 属性 `allowed_domains` 的 request。

当 spide 返回一个主机名不属于该 spider 的 request 时， 该中间件将会做一个类似于下面的记录:

```
DEBUG: Filtered offsite request to 'www.othersite.com': <GET http://www.othersite.com/some/page.html>
```

为了避免记录太多无用信息，其将对每个新发现的网站记录一次。因此，例如，如果过滤出另一个 `www.othersite.com` 请求，将不会有新的记录 但如果过滤出 `someothersite.com` 请求，仍然会有记录信息(仅针对第一次)。

如果 spider 没有定义 `allowed_domains` 属性，或该属性为空，则 offsite 中间件将会允许所有 request。

如果 request 设置了 `dont_filter` 属性，即使该 request 的网站不在允许列表里，则 offsite 中间件将会允许该 request。

## RefererMiddleware

class scrapy.contrib.spidermiddleware.referer.RefererMiddleware

根据生成 Request 的 Response 的 URL 来设置 Request `Referer` 字段。

RefererMiddleware settings

## REFERER_ENABLED

默认: `True`

是否启用 referer 中间件。

## UrlLengthMiddleware

class scrapy.contrib.spidermiddleware.urllength.UrlLengthMiddleware

过滤出 URL 长度比 URLLENGTH_LIMIT 的 request。

`UrlLengthMiddleware` 可以通过下列设置进行配置(更多内容请参考设置文档):

- `URLLENGTH_LIMIT` – 允许爬取 URL 最长的长度。

# 37

# 扩展(Extensions)

扩展框架提供一个机制，使得你能将自定义功能绑定到 Scrapy。

扩展只是正常的类，它们在 Scrapy 启动时被实例化、初始化。

## 扩展设置(Extension settings)

扩展使用 Scrapy settings 管理它们的设置，这跟其他 Scrapy 代码一样。

通常扩展需要给它们的设置加上前缀，以避免跟已有(或将来)的扩展冲突。比如，一个扩展处理 Google Sitemaps，则可以使用类似 *GOOGLESITEMAP_ENABLED*、*GOOGLESITEMAP_DEPTH* 等设置。

## 加载和激活扩展

扩展在扩展类被实例化时加载和激活。 因此，所有扩展的实例化代码必须在类的构造函数( `__init__` )中执行。

要使得扩展可用，需要把它添加到 Scrapy 的 `EXTENSIONS` 配置中。在 `EXTENSIONS` 中，每个扩展都使用一个字符串表示，即扩展类的全 Python 路径。比如:

```
EXTENSIONS = {
   'scrapy.contrib.corestats.CoreStats': 500,
   'scrapy.telnet.TelnetConsole': 500,
}
```

如你所见， `EXTENSIONS` 配置是一个 dict，key 是扩展类的路径，value 是顺序，它定义扩展加载的顺序。扩展顺序不像中间件的顺序那么重要，而且扩展之间一般没有关联。扩展加载的顺序并不重要，因为它们并不相互依赖。

如果你需要添加扩展而且它依赖别的扩展，你就可以使用该特性了。

这也是为什么 Scrapy 的配置项 `EXTENSIONS_BASE` (它包括了所有内置且开启的扩展)定义所有扩展的顺序都相同( `500` )。

## 可用的(Available)、开启的(enabled)和禁用的(disabled)的扩展

并不是所有可用的扩展都会被开启。一些扩展经常依赖一些特别的配置。 比如，HTTP Cache 扩展是可用的但默认是禁用的，除非 `HTTPCACHE_ENABLED` 配置项设置了。

## 禁用扩展(Disabling an extension)

为了禁用一个默认开启的扩展(比如，包含在 `EXTENSIONS_BASE` 中的扩展)，需要将其顺序(order)设置为 `None` 。比如:

```
EXTENSIONS = {
    'scrapy.contrib.corestats.CoreStats': None,
}
```

## 实现你的扩展

实现你的扩展很简单。每个扩展是一个单一的 Python class，它不需要实现任何特殊的方法。

Scrapy 扩展(包括 middlewares 和 pipelines)的主要入口是 `from_crawler` 类方法，它接收一个 `Crawler` 类的实例，该实例是控制 Scrapy crawler 的主要对象。如果扩展需要，你可以通过这个对象访问 settings，signals，stats，控制爬虫的行为。

通常来说，扩展关联到 signals 并执行它们触发的任务。

最后，如果 `from_crawler` 方法抛出 `NotConfigured` 异常，扩展会被禁用。否则，扩展会被开启。

### 扩展例子(Sample extension)

这里我们将实现一个简单的扩展来演示上面描述到的概念。 该扩展会在以下事件时记录一条日志:

- spider 被打开
- spider 被关闭
- 爬取了特定数量的条目(items)

该扩展通过 `MYEXT_ENABLED` 配置项开启，items 的数量通过 `MYEXT_ITEMCOUNT` 配置项设置。

以下是扩展的代码:

```
from scrapy import signals
from scrapy.exceptions import NotConfigured

class SpiderOpenCloseLogging(object):

    def __init__(self, item_count):
        self.item_count = item_count

        self.items_scraped = 0

    @classmethod
    def from_crawler(cls, crawler):
        # first check if the extension should be enabled and raise

        # NotConfigured otherwise
```

```
    if not crawler.settings.getbool('MYEXT_ENABLED'):

        raise NotConfigured

    # get the number of items from settings

    item_count = crawler.settings.getint('MYEXT_ITEMCOUNT', 1000)

    # instantiate the extension object

    ext = cls(item_count)

    # connect the extension object to signals

    crawler.signals.connect(ext.spider_opened, signal=signals.spider_opened)

    crawler.signals.connect(ext.spider_closed, signal=signals.spider_closed)

    crawler.signals.connect(ext.item_scraped, signal=signals.item_scraped)

    # return the extension object

    return ext

def spider_opened(self, spider):
    spider.log("opened spider %s" % spider.name)

def spider_closed(self, spider):
    spider.log("closed spider %s" % spider.name)

def item_scraped(self, item, spider):
    self.items_scraped += 1

    if self.items_scraped % self.item_count == 0:
        spider.log("scraped %d items" % self.items_scraped)
```

# 内置扩展介绍

## 通用扩展

### 记录统计扩展(Log Stats extension)

class scrapy.contrib.logstats.LogStats
记录基本的统计信息，比如爬取的页面和条目(items)。

### 核心统计扩展(Core Stats extension)

class scrapy.contrib.corestats.CoreStats
如果统计收集器(stats collection)启用了，该扩展开启核心统计收集(参考数据收集(Stats Collection))。

### Telnet console 扩展

class scrapy.telnet.TelnetConsole
提供一个 telnet 控制台，用于进入当前执行的 Scrapy 进程的 Python 解析器，这对代码调试非常有帮助。

telnet 控制台通过 `TELNETCONSOLE_ENABLED` 配置项开启，服务器会监听 `TELNETCONSOLE_PORT` 指定的端口。

### 内存使用扩展(Memory usage extension)

class scrapy.contrib.memusage.MemoryUsage

> 注解
>
> This extension does not work in Windows.

监控 Scrapy 进程内存使用量，并且：

1. 如果使用内存量超过某个指定值，发送提醒邮件
2. 如果超过某个指定值，关闭 spider

当内存用量达到 `MEMUSAGE_WARNING_MB` 指定的值，发送提醒邮件。当内存用量达到 `MEMUSAGE_LIMIT_MB` 指定的值，发送提醒邮件，同时关闭 spider，Scrapy 进程退出。

该扩展通过 MEMUSAGE_ENABLED 配置项开启，可以使用以下选项：

- MEMUSAGE_LIMIT_MB
- MEMUSAGE_WARNING_MB
- MEMUSAGE_NOTIFY_MAIL
- MEMUSAGE_REPORT

### 内存调试扩展(Memory debugger extension)

class scrapy.contrib.memdebug.MemoryDebugger
该扩展用于调试内存使用量，它收集以下信息：

- 没有被 Python 垃圾回收器收集的对象
- 应该被销毁却仍然存活的对象。更多信息请参考使用 trackref 调试内存泄露

开启该扩展，需打开 `MEMDEBUG_ENABLED` 配置项。 信息将会存储在统计信息(stats)中。

### 关闭 spider 扩展

class scrapy.contrib.closespider.CloseSpider
当某些状况发生，spider 会自动关闭。每种情况使用指定的关闭原因。

关闭 spider 的情况可以通过下面的设置项配置：

- CLOSESPIDER_TIMEOUT
- CLOSESPIDER_ITEMCOUNT
- CLOSESPIDER_PAGECOUNT
- CLOSESPIDER_ERRORCOUNT

### CLOSESPIDER_TIMEOUT

默认值: `0`

一个整数值，单位为秒。如果一个 spider 在指定的秒数后仍在运行， 它将以 `closespider_timeout` 的原因被自动关闭。如果值设置为 0（或者没有设置），spiders 不会因为超时而关闭。

### CLOSESPIDER_ITEMCOUNT

缺省值: `0`

一个整数值，指定条目的个数。如果 spider 爬取条目数超过了指定的数，并且这些条目通过 item pipeline 传递，spider 将会以 `closespider_itemcount` 的原因被自动关闭。

### CLOSESPIDER_PAGECOUNT

新版功能。

缺省值: 0

一个整数值，指定最大的抓取响应(reponses)数。 如果 spider 抓取数超过指定的值，则会以 `closespider_pagecount` 的原因自动关闭。 如果设置为 0（或者未设置），spiders 不会因为抓取的响应数而关闭。

### CLOSESPIDER_ERRORCOUNT

新版功能。

缺省值: `0`

一个整数值，指定 spider 可以接受的最大错误数。 如果 spider 生成多于该数目的错误，它将以 `closespider_errorcount` 的原因关闭。 如果设置为 0（或者未设置），spiders 不会因为发生错误过多而关闭。

## StatsMailer extension

class scrapy.contrib.statsmailer.StatsMailer
这个简单的扩展可用来在一个域名爬取完毕时发送提醒邮件， 包含 Scrapy 收集的统计信息。 邮件会发送个通过 `STATSMAILER_RCPTS` 指定的所有接收人。

## Debugging extensions

### Stack trace dump extension

class scrapy.contrib.debug.StackTraceDump
当收到 *SIGQUIT* 或 *SIGUSR2* 信号，spider 进程的信息将会被存储下来。存储的信息包括:

1. engine 状态(使用 `scrapy.utils.engin.get_engine_status()` )
2. 所有存活的引用(live references)(参考使用 trackref 调试内存泄露)
3. 所有线程的堆栈信息

当堆栈信息和 engine 状态存储后，Scrapy 进程继续正常运行。

该扩展只在 POSIX 兼容的平台上可运行（比如不能在 Windows 运行）， 因为 SIGQUIT 和 SIGUSR2 信号在 Windows 上不可用。

至少有两种方式可以向 Scrapy 发送 SIGQUIT 信号:

在 Scrapy 进程运行时通过按 Ctrl-(仅 Linux 可行?) 运行该命令( `<pid>` 是 Scrapy 运行的进程):

```
kill -QUIT <pid>
```

### 调试扩展(Debugger extension)

class scrapy.contrib.debug.Debugger

当收到 SIGUSR2 信号，将会在 Scrapy 进程中调用 Python debugger。debugger 退出后，Scrapy 进程继续正常运行。

更多信息参考 *Debugging in Python*。

该扩展只在 POSIX 兼容平台上工作(比如不能再 Windows 上运行)。

# 核心 API

新版功能。

该节文档讲述 Scrapy 核心 API，目标用户是开发 Scrapy 扩展(extensions)和中间件(middlewares)的开发人员。

# Crawler API

Scrapy API 的主要入口是 `Crawler` 的实例对象，通过类方法 `from_crawler` 将它传递给扩展(extensions)。该对象提供对所有 Scrapy 核心组件的访问，也是扩展访问 Scrapy 核心组件和挂载功能到 Scrapy 的唯一途径。

Extension Manager 负责加载和跟踪已经安装的扩展，它通过 `EXTENSIONS` 配置，包含一个所有可用扩展的字典， 字典的顺序跟你在 configure the downloader middlewares 配置的顺序一致。

### class scrapy.crawler.Crawler(spidercls, settings)

Crawler 必须使用 `scrapy.spider.Spider` 子类及 `scrapy.settings.Settings` 的对象进行实例化

#### settings

crawler 的配置管理器。

扩展(extensions)和中间件(middlewares)使用它用来访问 Scrapy 的配置。

关于 Scrapy 配置的介绍参考这里 Settings。

API 参考 `Settings` 。

#### signals

crawler 的信号管理器。

扩展和中间件使用它将自己的功能挂载到 Scrapy。

关于信号的介绍参考信号(Signals)。

API 参考 `SignalManager` 。

#### stats

crawler 的统计信息收集器。

扩展和中间件使用它记录操作的统计信息，或者访问由其他扩展收集的统计信息。

关于统计信息收集器的介绍参考数据收集(Stats Collection)。

API 参考类 `StatsCollector class` 。

#### extensions

扩展管理器，跟踪所有开启的扩展。

大多数扩展不需要访问该属性。

关于扩展和可用扩展列表器的介绍参考扩展(Extensions)。

engine
执行引擎，协调 crawler 的核心逻辑，包括调度，下载和 spider。

某些扩展可能需要访问 Scrapy 的引擎属性，以修改检查(modify inspect)或修改下载器和调度器的行为， 这是该 API 的高级使用，但还不稳定。

spider 正在爬取的 spider。该 spider 类的实例由创建 crawler 时所提供， 在调用 :meth:`crawl` 方法是所创建。

crawl(*args, **kwargs)
根据给定的 args , kwargs 的参数来初始化 spider 类，启动执行引擎，启动 crawler。

返回一个延迟 deferred 对象，当爬取结束时触发它。

get(name, default=None)
获取某项配置的值，且不修改其原有的值。

参数:

- name (*字符串*) – 配置名
- default (*任何*) – 如果没有该项配置时返回的缺省值

getbool(name, default=False)
return `False` 将某项配置的值以布尔值形式返回。比如， `1` 和 `'1'` ， `True` 都返回 `True` ， 而 `0` ， `'0'` ， `False` 和 `None` 返回 `False` 。

比如，通过环境变量计算将某项配置设置为 `'0'` ，通过该方法获取得到 `False` 。

参数:

- name (*字符串*) – 配置名 –** default** (*任何*) – 如果该配置项未设置，返回的缺省值

getint(name, default=0)
将某项配置的值以整数形式返回

参数:

- name (*字符串*) – 配置名 –** default** (*任何*) – 如果该配置项未设置，返回的缺省值

getfloat(name, default=0.0)
将某项配置的值以浮点数形式返回

参数:

- name (*字符串*) – 配置名 –** default** (*任何*) – 如果该配置项未设置，返回的缺省值

getlist(name, default=None)
将某项配置的值以列表形式返回。如果配置值本来就是 list 则将返回其拷贝。如果是字符串，则返回被 ”,” 分割后的列表。

比如，某项值通过环境变量的计算被设置为 'one,two'，该方法返回 [‘one’, ‘two’]。

参数:

- name (*字符串*) – 配置名 –** default** (*任何*) – 如果该配置项未设置，返回的缺省值

getdict(name, default=None)
Get a setting value as a dictionary. If the setting original type is a dictionary, a copy of it will be returned. If it’s a string it will evaluated as a json dictionary.

参数:

- name (*字符串*) – 配置名 –** default** (*任何*) – 如果该配置项未设置，返回的缺省值

copy()
Make a deep copy of current settings.

This method returns a new instance of the `Settings` class, populated with the same values and their priorities.

Modifications to the new object won’t be reflected on the original settings.

freeze()
Disable further changes to the current settings.

After calling this method, the present state of the settings will become immutable. Trying to change values through the `set()` method and its variants won’t be possible and will be alerted.

frozencopy()
Return an immutable copy of the current settings.

Alias for a `freeze()` call in the object returned by `copy()`

## SpiderManager API

class scrapy.spidermanager.SpiderManager

This class is in charge of retrieving and handling the spider classes defined across the project.

Custom spider managers can be employed by specifying their path in the `SPIDER_MANAGER_CLASS` project setting. They must fully implement the `scrapy.interfaces.ISpiderManager` interface to guarantee an errorless execution.

from_settings(settings)

This class method is used by Scrapy to create an instance of the class. It's called with the current project settings, and it loads the spiders found in the modules of the SPIDER_MODULES setting.

参数:

settings (`Settings` instance) – project settings

load(spider_name)

Get the Spider class with the given name. It'll look into the previously loaded spiders for a spider class with name spider_name and will raise a KeyError if not found.

参数:

spider_name (*str*) – spider class name

list()

Get the names of the available spiders in the project.

find_by_request(request)

List the spiders' names that can handle the given request. Will try to match the request's url against the domains of the spiders.

参数:

request (`Request` instance) – queried request

# 信号(Signals) API

class scrapy.signalmanager.SignalManager

connect(receiver, signal)
链接一个接收器函数(receiver function) 到一个信号(signal)。

signal 可以是任何对象，虽然 Scrapy 提供了一些预先定义好的信号， 参考文档信号(Signals)。

参数:

- receiver (*可调用对象*) – 被链接到的函数
- signal (*对象*) – 链接的信号

send_catch_log(signal, **kwargs)
发送一个信号，捕获异常并记录日志。

关键字参数会传递给信号处理者(signal handlers)(通过方法 `connect()` 关联)。

send_catch_log_deferred(signal, **kwargs)
跟 `send_catch_log()` 相似但支持返回 deferreds 形式的信号处理器。

返回一个 deferred，当所有的信号处理器的延迟被触发时调用。发送一个信号，处理异常并记录日志。

关键字参数会传递给信号处理者(signal handlers)(通过方法 `connect()` 关联)。

disconnect(receiver, signal)
解除一个接收器函数和一个信号的关联。这跟方法 `connect()` 有相反的作用，参数也相同。

disconnect_all(signal)
取消给定信号绑定的所有接收器。

参数:

- signal (*object*) – 要取消绑定的信号

# 状态收集器(Stats Collector) API

模块 scrapy.statscol 下有好几种状态收集器， 它们都实现了状态收集器 API 对应的类 `Statscollector` (即它们都继承至该类)。

## class scrapy.statscol.StatsCollector

get_value(key, default=None)
返回指定 key 的统计值，如果 key 不存在则返回缺省值。

get_stats()
以 dict 形式返回当前 spider 的所有统计值。

set_value(key, value)
设置 key 所指定的统计值为 value。

set_stats(stats)
使用 dict 形式的 stats 参数覆盖当前的统计值。

inc_value(key, count=1, start=0)
增加 key 所对应的统计值，增长值由 count 指定。如果 key 未设置，则使用 start 的值设置为初始值。

max_value(key, value)
如果 key 所对应的当前 value 小于参数所指定的 value，则设置 value。如果没有 key 所对应的 value，设置 value。

min_value(key, value)
如果 key 所对应的当前 value 大于参数所指定的 value，则设置 value。如果没有 key 所对应的 value，设置 value。

clear_stats()
清除所有统计信息。

以下方法不是统计收集 api 的一部分，但实现自定义的统计收集器时会使用到：

open_spider(spider)
打开指定 spider 进行统计信息收集。

close_spider(spider)
关闭指定 spider。调用后，不能访问和收集统计信息。

# 39
# Settings

Scrapy 设定(settings)提供了定制 Scrapy 组件的方法。您可以控制包括核心(core)，插件(extension)，pipeline 及 spider 组件。

设定为代码提供了提取以 key-value 映射的配置值的的全局命名空间(namespace)。设定可以通过下面介绍的多种机制进行设置。

设定(settings)同时也是选择当前激活的 Scrapy 项目的方法(如果您有多个的话)。

内置设定列表请参考 `内置设定参考手册` 。

## 指定设定(Designating the settings)

当您使用 Scrapy 时，您需要声明您所使用的设定。这可以通过使用环境变量: `SCRAPY_SETTINGS_MODULE` 来完成。

`SCRAPY_SETTINGS_MODULE` 必须以 Python 路径语法编写，如 `myproject.settings` 。注意，设定模块应该在 Python import search path 中。

# 获取设定值(Populating the settings)

设定可以通过多种方式设置，每个方式具有不同的优先级。 下面以优先级降序的方式给出方式列表:

1. 命令行选项(Command line Options)(最高优先级)
2. 每个 spider 的设定
3. 项目设定模块(Project settings module)
4. 命令默认设定模块(Default settings per-command)
5. 全局默认设定(Default global settings) (最低优先级)

这些设定(settings)由 scrapy 内部很好的进行了处理，不过您仍可以使用 API 调用来手动处理。 详情请参考 设置(Settings) API。

这些机制将在下面详细介绍。

## 命令行选项(Command line options)

命令行传入的参数具有最高的优先级。 您可以使用 command line 选项 `-s` (或 `--set` ) 来覆盖一个(或更多)选项。

样例:

```
scrapy crawl myspider -s LOG_FILE=scrapy.log
```

## 项目设定模块(Project settings module)

项目设定模块是您 Scrapy 项目的标准配置文件。 其是获取大多数设定的方法。例如:: `myproject.settings` 。

## 命令默认设定(Default settings per-command)

每个 Scrapy tool 命令拥有其默认设定，并覆盖了全局默认的设定。 这些设定在命令的类的 `default_settings` 属性中指定。

## 默认全局设定(Default global settings)

全局默认设定存储在 `scrapy.settings.default_settings` 模块，并在 `内置设定参考手册` 部分有所记录。

## 如何访问设定(How to access settings)

设定可以通过 Crawler 的 `scrapy.crawler.Crawler.settings` 属性进行访问。其由插件及中间件的 `from_crawler` 方法所传入:

```
class MyExtension(object):

    @classmethod
    def from_crawler(cls, crawler):
        settings = crawler.settings
        if settings['LOG_ENABLED']:
            print "log is enabled!"
```

另外，设定可以以字典方式进行访问。不过为了避免类型错误， 通常更希望返回需要的格式。 这可以通过 `Settings` API 提供的方法来实现。

## 设定名字的命名规则

设定的名字以要配置的组件作为前缀。例如，一个 robots.txt 插件的合适设定应该为 `ROBOTSTXT_ENABLED`，`ROBOTSTXT_OBEY`，`ROBOTSTXT_CACHEDIR` 等等。

# 内置设定参考手册

这里以字母序给出了所有可用的 Scrapy 设定及其默认值和应用范围。

如果给出可用范围，并绑定了特定的组件，则说明了该设定使用的地方。 这种情况下将给出该组件的模块，通常来说是插件、中间件或 pipeline。 同时也意味着为了使设定生效，该组件必须被启用。

## AWS_ACCESS_KEY_ID

默认: `None`

连接 Amazon Web services 的 AWS access key。S3 feed storage backend 中使用。

## AWS_SECRET_ACCESS_KEY

默认: `None`

连接 Amazon Web services 的 AWS access key。S3 feed storage backend 中使用。

## BOT_NAME

默认: `'scrapybot'`

Scrapy 项目实现的 bot 的名字(也为项目名称)。 这将用来构造默认 User-Agent，同时也用来 log。

当您使用 `startproject` 命令创建项目时其也被自动赋值。

## CONCURRENT_ITEMS

默认: `100`

Item Processor(即 Item Pipeline) 同时处理(每个 response 的)item 的最大值。

## CONCURRENT_REQUESTS

默认: `16`

Scrapy downloader 并发请求(concurrent requests)的最大值。

## CONCURRENT_REQUESTS_PER_DOMAIN

默认: `8`

对单个网站进行并发请求的最大值。

## CONCURRENT_REQUESTS_PER_IP

默认: `0`

对单个 IP 进行并发请求的最大值。如果非 0，则忽略 `CONCURRENT_REQUESTS_PER_DOMAIN` 设定，使用该设定。也就是说，并发限制将针对 IP，而不是网站。

该设定也影响 `DOWNLOAD_DELAY`: 如果 `CONCURRENT_REQUESTS_PER_IP` 非 0，下载延迟应用在 IP 而不是网站上。

## DEFAULT_ITEM_CLASS

默认: `'scrapy.item.Item'`

the Scrapy shell 中实例化 item 使用的默认类。

## DEFAULT_REQUEST_HEADERS

默认:

```
{
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
   'Accept-Language': 'en',
}
```

Scrapy HTTP Request 使用的默认 header。由 `DefaultHeadersMiddleware` 产生。

## DEPTH_LIMIT

默认: `0`

爬取网站最大允许的深度(depth)值。如果为 0，则没有限制。

## DEPTH_PRIORITY

默认: `0`

整数值。用于根据深度调整 request 优先级。

如果为 0，则不根据深度进行优先级调整。

## DEPTH_STATS

默认: `True`

是否收集最大深度数据。

## DEPTH_STATS_VERBOSE

默认: `False`

是否收集详细的深度数据。如果启用，每个深度的请求数将会被收集在数据中。

## DNSCACHE_ENABLED

默认: `True`

是否启用 DNS 内存缓存(DNS in-memory cache)。

## DOWNLOADER

默认: `'scrapy.core.downloader.Downloader'`

用于 crawl 的 downloader.

## DOWNLOADER_MIDDLEWARES

默认:: `{}`

保存项目中启用的下载中间件及其顺序的字典。 更多内容请查看激活下载器中间件。

## DOWNLOADER_MIDDLEWARES_BASE

默认:

```
{
    'scrapy.contrib.downloadermiddleware.robotstxt.RobotsTxtMiddleware': 100,
    'scrapy.contrib.downloadermiddleware.httpauth.HttpAuthMiddleware': 300,
    'scrapy.contrib.downloadermiddleware.downloadtimeout.DownloadTimeoutMiddleware': 350,
    'scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware': 400,
    'scrapy.contrib.downloadermiddleware.retry.RetryMiddleware': 500,
    'scrapy.contrib.downloadermiddleware.defaultheaders.DefaultHeadersMiddleware': 550,
    'scrapy.contrib.downloadermiddleware.redirect.MetaRefreshMiddleware': 580,
    'scrapy.contrib.downloadermiddleware.httpcompression.HttpCompressionMiddleware': 590,
    'scrapy.contrib.downloadermiddleware.redirect.RedirectMiddleware': 600,
    'scrapy.contrib.downloadermiddleware.cookies.CookiesMiddleware': 700,
    'scrapy.contrib.downloadermiddleware.httpproxy.HttpProxyMiddleware': 750,
    'scrapy.contrib.downloadermiddleware.chunked.ChunkedTransferMiddleware': 830,
    'scrapy.contrib.downloadermiddleware.stats.DownloaderStats': 850,
    'scrapy.contrib.downloadermiddleware.httpcache.HttpCacheMiddleware': 900,
}
```

包含 Scrapy 默认启用的下载中间件的字典。永远不要在项目中修改该设定，而是修改 `DOWNLOADER_MIDDLEWARES` 。更多内容请参考激活下载器中间件。

## DOWNLOADER_STATS

默认: `True`

是否收集下载器数据。

## DOWNLOAD_DELAY

默认: `0`

下载器在下载同一个网站下一个页面前需要等待的时间。该选项可以用来限制爬取速度，减轻服务器压力。同时也支持小数:

```
DOWNLOAD_DELAY = 0.25    # 250 ms of delay
```

该设定影响(默认启用的) `RANDOMIZE_DOWNLOAD_DELAY` 设定。 默认情况下，Scrapy 在两个请求间不等待一个固定的值， 而是使用 0.5 到 1.5 之间的一个随机值 * `DOWNLOAD_DELAY` 的结果作为等待间隔。

当 `CONCURRENT_REQUESTS_PER_IP` 非 0 时，延迟针对的是每个 ip 而不是网站。

另外您可以通过 spider 的 `download_delay` 属性为每个 spider 设置该设定。

## DOWNLOAD_HANDLERS

默认: `{}`

保存项目中启用的下载处理器(request downloader handler)的字典。 例子请查看 *DOWNLOAD_HANDLERS_BASE*。

## DOWNLOAD_HANDLERS_BASE

默认:

```
{
    'file': 'scrapy.core.downloader.handlers.file.FileDownloadHandler',
    'http': 'scrapy.core.downloader.handlers.http.HttpDownloadHandler',
    'https': 'scrapy.core.downloader.handlers.http.HttpDownloadHandler',
    's3': 'scrapy.core.downloader.handlers.s3.S3DownloadHandler',
}
```

保存项目中默认启用的下载处理器(request downloader handler)的字典。 永远不要在项目中修改该设定，而是修改 `DOWNLOADER_HANDLERS` 。

如果需要关闭上面的下载处理器，您必须在项目中的 DOWNLOAD_HANDLERS 设定中设置该处理器，并为其赋值为 None。例如，关闭文件下载处理器:

```
DOWNLOAD_HANDLERS = {
    'file': None,
}
```

## DOWNLOAD_TIMEOUT

默认: 180

下载器超时时间(单位: 秒)。

> 注解
>
> 该超时值可以使用 download_timeout 来对每个 spider 进行设置，也可以使用 download_timeout Request.meta key 来对每个请求进行设置。 This feature needs Twisted >= 11.1.

## DOWNLOAD_MAXSIZE

Default:*1073741824 (1024MB)*

The maximum response size (in bytes) that downloader will download.

If you want to disable it set to 0.

> 注解
>
> This size can be set per spider using download_maxsize spider attribute and per-request using download_maxsize Request.meta key.

## DOWNLOAD_WARNSIZE

Default:*33554432 (32Mb)*

The response size (in bytes) that downloader will start to warn.

If you want to disable it set to 0.

> 注解

> This size can be set per spider using `download_warnsize` spider attribute and per-request using `download_warnsize` Request.meta key.
>
> This feature needs Twisted >= 11.1.

## DUPEFILTER_CLASS

默认: `'scrapy.dupefilter.RFPDupeFilter'`

用于检测过滤重复请求的类。

默认的 ( `RFPDupeFilter` ) 过滤器基于 `scrapy.utils.request.request_fingerprint` 函数生成的请求 `fingerprint` (指纹)。 如果您需要修改检测的方式，您可以继承 `RFPDupeFilter` 并覆盖其 `request_fingerprint` 方法。 该方法接收 `Request` 对象并返回其 fingerprint(一个字符串)。

## DUPEFILTER_DEBUG

默认: `False`

默认情况下， `RFPDupeFilter` 只记录第一次重复的请求。设置 `DUPEFILTER_DEBUG` 为 `True` 将会使其记录所有重复的 requests。

## EDITOR

默认: `depends on the environment`

执行 `edit` 命令编辑 spider 时使用的编辑器。其默认为 `EDITOR` 环境变量。如果该变量未设置，其默认为 `vi` (Unix 系统) 或者 IDLE 编辑器(Windows)。

## EXTENSIONS

默认:: `{}`

保存项目中启用的插件及其顺序的字典。

## EXTENSIONS_BASE

默认:

```
{
    'scrapy.contrib.corestats.CoreStats': 0,
    'scrapy.telnet.TelnetConsole': 0,
    'scrapy.contrib.memusage.MemoryUsage': 0,
    'scrapy.contrib.memdebug.MemoryDebugger': 0,
    'scrapy.contrib.closespider.CloseSpider': 0,
    'scrapy.contrib.feedexport.FeedExporter': 0,
    'scrapy.contrib.logstats.LogStats': 0,
    'scrapy.contrib.spiderstate.SpiderState': 0,
    'scrapy.contrib.throttle.AutoThrottle': 0,
}
```

可用的插件列表。需要注意，有些插件需要通过设定来启用。默认情况下，该设定包含所有稳定(stable)的内置插件。

更多内容请参考 extensions 用户手册。

## ITEM_PIPELINES

默认: `{}`

保存项目中启用的 pipeline 及其顺序的字典。该字典默认为空，值(value)任意。不过值(value)习惯设定在 0-1000 范围内。

为了兼容性， `ITEM_PIPELINES` 支持列表，不过已经被废弃了。

样例:

```
ITEM_PIPELINES = {
    'mybot.pipelines.validate.ValidateMyItem': 300,
    'mybot.pipelines.validate.StoreMyItem': 800,
}
```

## ITEM_PIPELINES_BASE

默认: `{}`

保存项目中默认启用的 pipeline 的字典。永远不要在项目中修改该设定，而是修改 `ITEM_PIPELINES` 。

## LOG_ENABLED

默认: `True`

是否启用 logging。

## LOG_ENCODING

默认: `'utf-8'`

logging 使用的编码。

## LOG_FILE

默认: `None`

logging 输出的文件名。如果为 None，则使用标准错误输出(standard error)。

## LOG_LEVEL

默认: `'DEBUG'`

log 的最低级别。可选的级别有: CRITICAL、ERROR、WARNING、INFO、DEBUG。更多内容请查看 Logging。

## LOG_STDOUT

默认: `False`

如果为 `True` ，进程所有的标准输出(及错误)将会被重定向到 log 中。例如，执行 `print 'hello'` ，其将会在 Scrapy log 中显示。

## MEMDEBUG_ENABLED

默认: False

是否启用内存调试(memory debugging)。

## MEMDEBUG_NOTIFY

默认: []

如果该设置不为空，当启用内存调试时将会发送一份内存报告到指定的地址；否则该报告将写到 log 中。

样例:

```
MEMDEBUG_NOTIFY = ['user@example.com']
```

## MEMUSAGE_ENABLED

默认: False

Scope: scrapy.contrib.memusage

是否启用内存使用插件。当 Scrapy 进程占用的内存超出限制时，该插件将会关闭 Scrapy 进程， 同时发送 email 进行通知。

See内存使用扩展(Memory usage extension)。

## MEMUSAGE_LIMIT_MB

默认: 0

Scope: scrapy.contrib.memusage

在关闭 Scrapy 之前所允许的最大内存数(单位: MB)(如果 MEMUSAGE_ENABLED 为 True)。 如果为 0，将不做限制。

See内存使用扩展(Memory usage extension)。

## MEMUSAGE_NOTIFY_MAIL

默认: `False`

Scope: `scrapy.contrib.memusage`

达到内存限制时通知的 email 列表。

Example:

```
MEMUSAGE_NOTIFY_MAIL = ['user@example.com']
```

See内存使用扩展(Memory usage extension)。

## MEMUSAGE_REPORT

默认: `False`

Scope: `scrapy.contrib.memusage`

每个 spider 被关闭时是否发送内存使用报告。

查看内存使用扩展(Memory usage extension)

## MEMUSAGE_WARNING_MB

默认: `0`

Scope: `scrapy.contrib.memusage`

在发送警告 email 前所允许的最大内存数(单位: MB)(如果 MEMUSAGE_ENABLED 为 True)。如果为 0，将不发送警告。

## NEWSPIDER_MODULE

默认: `''`

使用 `genspider` 命令创建新 spider 的模块。

样例:

```
NEWSPIDER_MODULE = 'mybot.spiders_dev'
```

## RANDOMIZE_DOWNLOAD_DELAY

默认: `True`

如果启用，当从相同的网站获取数据时，Scrapy 将会等待一个随机的值 (0.5 到 1.5 之间的一个随机值* `DOWNLOAD_DELAY` )。

该随机值降低了 crawler 被检测到(接着被 block)的机会。某些网站会分析请求， 查找请求之间时间的相似性。

随机的策略与 wget `--random-wait` 选项的策略相同。

若 `DOWNLOAD_DELAY` 为 0(默认值)，该选项将不起作用。

## REDIRECT_MAX_TIMES

默认: `20`

定义 request 允许重定向的最大次数。超过该限制后该 request 直接返回获取到的结果。 对某些任务我们使用 Firefox 默认值。

## REDIRECT_MAX_METAREFRESH_DELAY

默认: `100`

有些网站使用 meta-refresh 重定向到 session 超时页面，因此我们限制自动重定向到最大延迟(秒)。=>有点不肯定:

## REDIRECT_PRIORITY_ADJUST

默认: `+2`

修改重定向请求相对于原始请求的优先级。 负数意味着更多优先级。

## ROBOTSTXT_OBEY

默认: `False`

Scope: `scrapy.contrib.downloadermiddleware.robotstxt`

如果启用，Scrapy 将会尊重 robots.txt 策略。更多内容请查看 RobotsTxtMiddleware。

## SCHEDULER

默认: `'scrapy.core.scheduler.Scheduler'`

用于爬取的调度器。

## SPIDER_CONTRACTS

默认:: `{}`

保存项目中启用用于测试 spider 的 scrapy contract 及其顺序的字典。 更多内容请参考 Spiders Contracts。

## SPIDER_CONTRACTS_BASE

默认:

```
{
   'scrapy.contracts.default.UrlContract' : 1,
   'scrapy.contracts.default.ReturnsContract': 2,
   'scrapy.contracts.default.ScrapesContract': 3,
}
```

保存项目中默认启用的 scrapy contract 的字典。永远不要在项目中修改该设定，而是修改 `SPIDER_CONTRACTS` 。更多内容请参考 Spiders Contracts。

## SPIDER_MANAGER_CLASS

默认: `'scrapy.spidermanager.SpiderManager'`

用于管理 spider 的类。该类必须实现 SpiderManager API。

## SPIDER_MIDDLEWARES

默认:: {}

保存项目中启用的下载中间件及其顺序的字典。更多内容请参考激活 spider 中间件。

## SPIDER_MIDDLEWARES_BASE

默认:

```
{
    'scrapy.contrib.spidermiddleware.httperror.HttpErrorMiddleware': 50,
    'scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware': 500,
    'scrapy.contrib.spidermiddleware.referer.RefererMiddleware': 700,
    'scrapy.contrib.spidermiddleware.urllength.UrlLengthMiddleware': 800,
    'scrapy.contrib.spidermiddleware.depth.DepthMiddleware': 900,
}
```

保存项目中默认启用的 spider 中间件的字典。永远不要在项目中修改该设定，而是修改 SPIDER_MIDDLEWARES 。更多内容请参考激活 spider 中间件。

## SPIDER_MODULES

默认: []

Scrapy 搜索 spider 的模块列表。

样例:

```
SPIDER_MODULES = ['mybot.spiders_prod', 'mybot.spiders_dev']
```

## STATS_CLASS

默认: 'scrapy.statscol.MemoryStatsCollector'

收集数据的类。该类必须实现状态收集器(Stats Collector) API。

## STATS_DUMP

默认: `True`

当 spider 结束时 dump Scrapy 状态数据 (到 Scrapy log 中)。

更多内容请查看数据收集(Stats Collection)。

## STATSMAILER_RCPTS

默认: `[]` (空 list)

spider 完成爬取后发送 Scrapy 数据。更多内容请查看 `StatsMailer` 。

## TELNETCONSOLE_ENABLED

默认: `True`

表明 Telnet 终端(及其插件)是否启用的布尔值。

## TELNETCONSOLE_PORT

默认: `[6023, 6073]`

telnet 终端使用的端口范围。如果设置为 `None` 或 `0` ， 则使用动态分配的端口。更多内容请查看 Telnet 终端(Telnet Console) 。

## TEMPLATES_DIR

默认: scrapy 模块内部的 `templates`

使用 `startproject` 命令创建项目时查找模板的目录。

## URLLENGTH_LIMIT

默认: `2083`

Scope: `contrib.spidermiddleware.urllength`

爬取 URL 的最大长度。更多关于该设定的默认值信息请查看:

http://www.boutell.com/newfaq/misc/urllength.html

## USER_AGENT

默认: `"Scrapy/VERSION (+http://scrapy.org)"`

爬取的默认 User-Agent，除非被覆盖。

# 40

# 信号(Signals)

Scrapy 使用信号来通知事情发生。您可以在您的 Scrapy 项目中捕捉一些信号(使用 extension)来完成额外的工作或添加额外的功能，扩展 Scrapy。

虽然信号提供了一些参数，不过处理函数不用接收所有的参数 - 信号分发机制(singal dispatching mechanism)仅仅提供处理器(handler)接受的参数。

您可以通过信号(Signals) API 来连接(或发送您自己的)信号。

## 延迟的信号处理器(Deferred signal handlers)

有些信号支持从处理器返回 Twisted deferreds，参考下边的 内置信号参考手册(Built-in signals reference) 来了解哪些支持。

# 内置信号参考手册(Built-in signals reference)

以下给出 Scrapy 内置信号的列表及其意义。

## engine_started

scrapy.signals.engine_started()

当 Scrapy 引擎启动爬取时发送该信号。

该信号支持返回 deferreds。

> 注解
>
> 该信号可能会在信号 `spider_opened` 之后被发送，取决于 spider 的启动方式。 所以不要 依赖 该信号会比 `spider-opened` 更早被发送。

## engine_stopped

scrapy.signals.engine_stopped()

当 Scrapy 引擎停止时发送该信号(例如，爬取结束)。

该信号支持返回 deferreds。

## item_scraped

scrapy.signals.item_scraped(item, response, spider)

当 item 被爬取，并通过所有 Item Pipeline 后(没有被丢弃(dropped)，发送该信号。

该信号支持返回 deferreds。

**参数:**

- **item** ( `Item` 对象) – 爬取到的 item
- **spider** ( `Spider` 对象) – 爬取 item 的 spider

- response（`Response` 对象）– 提取 item 的 response

## item_dropped

scrapy.signals.item_dropped(item, exception, spider)

当 item 通过 Item Pipeline，有些 pipeline 抛出 DropItem 异常，丢弃 item 时，该信号被发送。

该信号支持返回 deferreds。

参数:

- item（`Item` 对象）– 爬取到的 item
- spider（`Spider` 对象）– 爬取 item 的 spider
- exception（`DropItem` 异常）– 导致 item 被丢弃的异常(必须是 DropItem 的子类)

## spider_closed

scrapy.signals.spider_closed(spider, reason)

当某个 spider 被关闭时，该信号被发送。该信号可以用来释放每个 spider 在 `spider_opened` 时占用的资源。

该信号支持返回 deferreds。

参数:

- spider（`Spider` 对象）– 关闭的 spider
- reason (*str*) – 描述 spider 被关闭的原因的字符串。如果 spider 是由于完成爬取而被关闭，则其为 `'finished'` 。否则，如果 spider 是被引擎的 `close_spider` 方法所关闭，则其为调用该方法时传入的 `reason` 参数(默认为 `'cancelled'` )。如果引擎被关闭(例如， 输入 Ctrl-C)，则其为 `'shutdown'` 。

## spider_opened

scrapy.signals.spider_opened(spider)

当 spider 开始爬取时发送该信号。该信号一般用来分配 spider 的资源，不过其也能做任何事。

该信号支持返回 deferreds。

参数:

- spider ( Spider 对象 ) – 开启的 spider

## spider_idle

scrapy.signals.spider_idle(spider)

当 spider 进入空闲(idle)状态时该信号被发送。空闲意味着:

- requests 正在等待被下载
- requests 被调度
- items 正在 item pipeline 中被处理

当该信号的所有处理器(handler)被调用后，如果 spider 仍然保持空闲状态，引擎将会关闭该 spider。当 spider 被关闭后， spider_close d 信号将被发送。

您可以，比如，在 spider_idle 处理器中调度某些请求来避免 spider 被关闭。

该信号**不支持**返回 deferreds。

参数:

- spider ( Spider 对象) – 空闲的 spider

## spider_error

scrapy.signals.spider_error(failure, response, spider)

当 spider 的回调函数产生错误时(例如，抛出异常)，该信号被发送。

参数:

- failure (Failure 对象) – 以 Twisted Failure 对象抛出的异常
- response ( Response 对象 ) – 当异常被抛出时被处理的 response
- spider ( Spider 对象 ) – 抛出异常的 spider

## request_scheduled

scrapy.signals.request_scheduled(request, spider)

当引擎调度一个 Request 对象用于下载时，该信号被发送。

该信号 不支持 返回 deferreds。

参数:

- request ( `Request` 对象) - 到达调度器的 request
- spider ( `Spider` 对象) - 产生该 request 的 spider

## request_dropped

scrapy.signals.request_dropped(request, spider)

Sent when a `Request`, scheduled by the engine to be downloaded later, is rejected by the scheduler.

The signal does not support returning deferreds from their handlers.

参数:

- request ( `Request` object) - the request that reached the scheduler
- spider ( `Spider` object) - the spider that yielded the request

## response_received

scrapy.signals.response_received(response, request, spider)

当引擎从 downloader 获取到一个新的 Response 时发送该信号。

该信号 不支持 返回 deferreds。

参数:

- response ( `Response` 对象) - 接收到的 response
- request ( `Request` 对象) - 生成 response 的 request

- spider ( `Spider 对象` ) – response 所对应的 spider

## response_downloaded

scrapy.signals.response_downloaded(response, request, spider)

当一个 HTTPResponse 被下载时，由 downloader 发送该信号。

该信号 不支持 返回 deferreds。

参数:

- response ( `Response` 对象) – \下载的 response
- request ( `Request` 对象) – 生成 response 的 request
- spider ( `Spider 对象` ) – response 所对应的 spider

41

# 异常(Exceptions)

# 内置异常参考手册(Built-in Exceptions reference)

下面是 Scrapy 提供的异常及其用法。

## DropItem

exception scrapy.exceptions.DropItem

该异常由 item pipeline 抛出，用于停止处理 item。详细内容请参考 Item Pipeline。

## CloseSpider

exception scrapy.exceptions.CloseSpider(reason='cancelled')

该异常由 spider 的回调函数(callback)抛出，来暂停/停止 spider。支持的参数:

参数:

- reason (*str*) – 关闭的原因

样例:

```
def parse_page(self, response):
  if 'Bandwidth exceeded' in response.body:
    raise CloseSpider('bandwidth_exceeded')
```

## IgnoreRequest

exception scrapy.exceptions.IgnoreRequest

该异常由调度器(Scheduler)或其他下载中间件抛出，声明忽略该 request。

## NotConfigured

exception scrapy.exceptions.NotConfigured

该异常由某些组件抛出，声明其仍然保持关闭。这些组件包括:

- Extensions
- Item pipelines
- Downloader middlwares
- Spider middlewares

该异常必须由组件的构造器(constructor)抛出。

## NotSupported

exception scrapy.exceptions.NotSupported

该异常声明一个不支持的特性。

# 42
# Item Exporters

当你抓取了你要的数据(Items)，你就会想要将他们持久化或导出它们，并应用在其他的程序。这是整个抓取过程的目的。

为此，Scrapy 提供了 Item Exporters 来创建不同的输出格式，如 XML，CSV 或 JSON。

## 使用 Item Exporter

如果你很忙，只想使用 Item Exporter 输出数据，请查看 Feed exports。相反，如果你想知道 Item Exporter 是如何工作的，或需要更多的自定义功能（不包括默认的 exports），请继续阅读下文。

为了使用 Item Exporter，你必须对 Item Exporter 及其参数 (args) 实例化。每个 Item Exporter 需要不同的参数，详细请查看 Item Exporters 参考资料 。在实例化了 exporter 之后，你必须：

1. 调用方法 start_exporting() 以标识 exporting 过程的开始。
2. 对要导出的每个项目调用 export_item() 方法。
3. 最后调用 finish_exporting() 表示 exporting 过程的结束

这里，你可以看到一个 Item Pipeline，它使用 Item Exporter 导出 items 到不同的文件，每个 spider 一个：

```
from scrapy import signals
from scrapy.contrib.exporter import XmlItemExporter

class XmlExportPipeline(object):

    def __init__(self):
        self.files = {}

     @classmethod
     def from_crawler(cls, crawler):
         pipeline = cls()
         crawler.signals.connect(pipeline.spider_opened, signals.spider_opened)
         crawler.signals.connect(pipeline.spider_closed, signals.spider_closed)
         return pipeline

    def spider_opened(self, spider):
        file = open('%s_products.xml' % spider.name, 'w+b')
        self.files[spider] = file
        self.exporter = XmlItemExporter(file)
        self.exporter.start_exporting()

    def spider_closed(self, spider):
        self.exporter.finish_exporting()
        file = self.files.pop(spider)
        file.close()

    def process_item(self, item, spider):
```

```
        self.exporter.export_item(item)
        return item
```

序列化 item fields

B 默认情况下，该字段值将不变的传递到序列化库，如何对其进行序列化的决定被委托给每一个特定的序列化库。

但是，你可以自定义每个字段值如何序列化在它被传递到序列化库中之前。

有两种方法可以自定义一个字段如何被序列化，请看下文。

## 在 field 类中声明一个 serializer

您可以在 field metadata 声明一个 serializer。该 serializer 必须可调用，并返回它的序列化形式。

实例:

```
import scrapy

def serialize_price(value):
    return '$ %s' % str(value)

class Product(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field(serializer=serialize_price)
```

## 覆盖(overriding) serialize_field()方法

你可以覆盖 `serialize_field()` 方法来自定义如何输出你的数据。

在你的自定义代码后确保你调用父类的 `serialize_field()` 方法。

实例:

```
from scrapy.contrib.exporter import XmlItemExporter

class ProductXmlExporter(XmlItemExporter):

    def serialize_field(self, field, name, value):
        if field == 'price':
            return '$ %s' % str(value)
        return super(Product, self).serialize_field(field, name, value)
```

# Item Exporters 参考资料

下面是一些 Scrapy 内置的 Item Exporters 类. 其中一些包括了实例, 假设你要输出以下 2 个 Items:

```
Item(name='Color TV', price='1200')
Item(name='DVD player', price='200')
```

## BaseItemExporter

class scrapy.contrib.exporter.BaseItemExporter(fields_to_export=None, export_empty_fields=False, encoding='utf-8')

这是一个对所有 Item Exporters 的(抽象)父类。它对所有(具体) Item Exporters 提供基本属性，如定义 export 什么 fields, 是否 export 空 fields, 或是否进行编码。

你可以在构造器中设置它们不同的属性值: `fields_to_export` , `export_empty_fields` , `encoding` 。

export_item(item)
输出给定 item。此方法必须在子类中实现。

serialize_field(field, name, value)
返回给定 field 的序列化值。你可以覆盖此方法来控制序列化或输出指定的 field。

默认情况下，此方法寻找一个 serializer 在 `item field` 中声明 并返回它的值。如果没有发现 serializer，则值不会改变，除非你使用 unicode 值并编码到 str，编码可以在 `encoding` 属性中声明。

参数:

- field ( `Field` object) – the field being serialized
- name (*str*) – the name of the field being serialized
- value – the value being serialized

start_exporting()
表示 exporting 过程的开始。一些 exporters 用于产生需要的头元素(例如 `XmlItemExporter` )。 在实现 exporting item 前必须调用此方法。

finish_exporting()
表示 exporting 过程的结束。一些 exporters 用于产生需要的尾元素 (例如 `XmlItemExporter` )。 在完成 exporting item 后必须调用此方法。

fields_to_export

列出 export 什么 fields 值，None 表示 export 所有 fields。默认值为 None。

一些 exporters (例如 `CsvItemExporter` ) 按照定义在属性中 fields 的次序依次输出。

export_empty_fields

是否在输出数据中包含为空的 item fields。默认值是 False。一些 exporters (例如 CsvItemExporter) 会忽略此属性并输出所有 fields。

encoding

Encoding 属性将用于编码 unicode 值。(仅用于序列化字符串)。其他值类型将不变的传递到指定的序列化库。

### XmlItemExporter

class scrapy.contrib.exporter.XmlItemExporter(file, item_element='item', root_element='items', **kwargs)

以 XML 格式 exports Items 到指定的文件类。

参数:

- file – 文件类。
- root_element (*str*) – XML 根元素名。
- item_element (*str*) – XML item 的元素名。

构造器额外的关键字参数将传给 `BaseItemExporter` 构造器。

一个典型的 exporter 实例:

```
<?xml version="1.0" encoding="utf-8"?>
<items>
 <item>
  <name>Color TV</name>
  <price>1200</price>
 </item>
 <item>
  <name>DVD player</name>
  <price>200</price>
 </item>
</items>
```

除了覆盖 `serialize_field()` 方法，多个值的 fields 会转化每个值到 `<value>` 元素。

例如，item:

```
Item(name=['John', 'Doe'], age='23')
```

将被转化为:

```
<?xml version="1.0" encoding="utf-8"?>
<items>
  <item>
    <name>
      <value>John</value>
      <value>Doe</value>
    </name>
    <age>23</age>
  </item>
</items>
```

## CsvItemExporter

class scrapy.contrib.exporter.CsvItemExporter(file, include_headers_line=True, join_multivalued=', ', **kwargs)

输出 csv 文件格式。如果添加 `fields_to_export` 属性，它会按顺序定义 CSV 的列名。 `export_empty_fields` 属性在此没有作用。

参数:

- file – 文件类。
- include_headers_line (str) – 启用后 exporter 会输出第一行为列名，列名从 `BaseItemExporter.fields_to_export` 或第一个 item fields 获取。
- join_multivalued – char 将用于连接多个值的 fields。

此构造器额外的关键字参数将传给 `BaseItemExporter` 构造器，其余的将传给 csv.writer 构造器， 以此来定制 exporter。

一个典型的 exporter 实例:

```
product,price
Color TV,1200
DVD player,200
```

## PickleItemExporter

class scrapy.contrib.exporter.PickleItemExporter(file, protocol=0, **kwargs)

输出 pickle 文件格式。

参数:

- file – 文件类。
- protocol (*int*) – pickle 协议。

更多信息请看 pickle module documentation。

此构造器额外的关键字参数将传给 BaseItemExporter 构造器。

Pickle 不是可读的格式，这里不提供实例。

## PprintItemExporter

class scrapy.contrib.exporter.PprintItemExporter(file, **kwargs)

输出整齐打印的文件格式。

参数:

- file – 文件类。

此构造器额外的关键字参数将传给 BaseItemExporter 构造器。

一个典型的 exporter 实例:

```
{'name': 'Color TV', 'price': '1200'}
{'name': 'DVD player', 'price': '200'}
```

此格式会根据行的长短进行调整。

## JsonItemExporter

class scrapy.contrib.exporter.JsonItemExporter(file, **kwargs)

输出 JSON 文件格式，所有对象将写进一个对象的列表。此构造器额外的关键字参数将传给 `BaseItemExporter` 构造器，其余的将传给 JSONEncoder 构造器，以此来定制 exporter。

参数:

- file – 文件类。

一个典型的 exporter 实例:

```
[{"name": "Color TV", "price": "1200"},
{"name": "DVD player", "price": "200"}]
```

警告

JSON 是一个简单而有弹性的格式, 但对大量数据的扩展性不是很好，因为这里会将整个对象放入内存。如果你要 JSON 既强大又简单，可以考虑 `JsonLinesItemExporter` ，或把输出对象分为多个块。

## JsonLinesItemExporter

class scrapy.contrib.exporter.JsonLinesItemExporter(file, **kwargs)

输出 JSON 文件格式, 每行写一个 JSON-encoded 项。此构造器额外的关键字参数将传给 `BaseItemExporter` 构造器，其余的将传给 JSONEncoder 构造器，以此来定制 exporter。

参数:

- file – 文件类。

一个典型的 exporter 实例:

```
{"name": "Color TV", "price": "1200"}
{"name": "DVD player", "price": "200"}
```

这个类能很好的处理大量数据。

# 43

# 试验阶段特性

这部分介绍一些正处于试验阶段的 Scrapy 特性， 这些特性所涉及到的函数接口等还不够稳定， 但会在以后的发布版中趋于完善。所以在使用这些特性过程中需更谨慎，并且最好订阅我们的邮件列表以便接收任何有关特性改变的通知。

虽然这些特性不会频繁的被修改，但是这部分文档仍有可能是过时的、不完整的或是与已经稳定的特性文档重复。所以你需要自行承担使用风险。

> 警告
>
> 本部分文档一直处于修改中。请自行承担使用风险。

## 使用外部库插入命令

你可以使用外部库通过增加 scrapy.commands 部分到 setup.py 的entry_points 中来插入 Scrapy 命令。

增加 my_command 命令的例子:

```
from setuptools import setup, find_packages

setup(name='scrapy-mymodule',
  entry_points={
    'scrapy.commands': [
      'my_command=my_scrapy_module.commands:MyCommand',
    ],
  },
 )
```

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台


更多信息请访问

http://wiki.jikexueyuan.com/project/scrapy/