蜘蛛

蜘蛛是定义如何抓取某个站点（或一组站点）的类，包括如何执行爬行（即跟随链接）以及如何从其页面中提取结构化数据（即抓取项目）。换句话说，Spiders是您为特定站点（或者在某些情况下，一组站点）爬网和解析页面定义自定义行为的地方。

对于蜘蛛，刮擦周期经历如下：

1. 首先生成初始请求以爬网第一个URL，并指定要使用从这些请求下载的响应调用的回调函数。

   第一个执行请求是通过调用 `start_requests()` （默认情况下）为在请求中作为回调函数的方法中 `Request` 指定的URL `start_urls` 和 `parse` 方法生成的 方法获得的。
2. 在回调函数中，您解析响应（网页）并返回带有提取的数据， `Item` 对象， `Request` 对象或这些对象的可迭代的dicts。这些请求还将包含一个回调（可能相同），然后由Scrapy下载，然后由指定的回调处理它们的响应。
3. 在回调函数中，您通常使用选择器解析页面内容（但您也可以使用BeautifulSoup，lxml或您喜欢的任何机制）并使用解析的数据生成项目。
4. 最后，从蜘蛛返回的项目通常会持久保存到数据库（在某些项目管道中）或使用Feed导出写入文件。

即使这个循环（或多或少）适用于任何类型的蜘蛛，但是为了不同的目的，Scrapy中捆绑了不同种类的默认蜘蛛。我们将在这里讨论这些类型。

scrapy.Spider

类 **scrapy.spiders.Spider**

这是最简单的蜘蛛，也是每个其他蜘蛛必须继承的蜘蛛（包括与Scrapy捆绑在一起的蜘蛛，以及你自己编写的蜘蛛）。它不提供任何特殊功能。它只提供了一个默认 `start_requests()` 实现，它从 `start_urls` spider属性发送请求，并 `parse` 为每个结果响应调用spider的方法。

name

一个字符串，用于定义此蜘蛛的名称。蜘蛛名称是Scrapy如何定位（并实例化）蜘蛛，因此它必须是唯一的。但是，没有什么可以阻止您实例化同一个蜘蛛的多个实例。这是最重要的蜘蛛属性，它是必需的。

如果蜘蛛刮擦单个域，通常的做法是在域之后命名蜘蛛，无论是否有TLD。因此，例如， `mywebsite.com` 经常会调用 爬行的蜘蛛 `mywebsite` 。

注意

在Python 2中，这必须只是ASCII。

allowed_domains

包含允许此爬网爬网的域的字符串的可选列表。如果`OffsiteMiddleware`启用，则不会遵循对不属于此列表（或其子域）中指定的域名的URL的请求 。

假设你的目标网址是`https://www.example.com/1.html`，然后添加`'example.com'`到列表中。

start_urls

当没有指定特定URL时，蜘蛛将开始爬网的URL列表。因此，下载的第一页将是此处列出的页面。后续`Request`将从起始URL中包含的数据连续生成。

custom_settings

运行此蜘蛛时将从项目范围配置中覆盖的设置字典。必须将其定义为类属性，因为在实例化之前更新了设置。

有关可用内置设置的列表，请参阅： 内置设置参考。

crawler

`from_crawler()`初始化类后，此属性由类方法设置，并链接`Crawler`到此spider实例绑定到的 对象。

Crawler在项目中封装了许多组件，用于单一条目访问（例如扩展，中间件，信号管理器等）。请参阅Crawler API以了解有关它们的更多信息。

settings

运行此蜘蛛的配置。这是一个`Settings`实例，请参阅“ 设置”主题以获取有关此主题的详细介绍。

logger

使用Spider创建的Python记录器`name`。您可以使用它来发送日志消息，如“ 从蜘蛛记录”中所述 。

from_crawler (*crawler* , ** args* , *** kwargs*)

这是Scrapy用于创建蜘蛛的类方法。

您可能不需要直接覆盖它，因为默认实现充当方法的代理`__init__()`，使用给定的参数*args*和命名参数*kwargs*调用它。

尽管如此，此方法 在新实例中设置`crawler`和`settings`属性，以便稍后可以在蜘蛛代码中访问它们。

参数：
- **crawler**（`Crawler` instance）- 蜘蛛绑定到的爬虫
- **args**（*list*）- 传递给`__init__()`方法的参数
- **kwargs**（*dict*）- 传递给`__init__()`方法的关键字参数

start_requests（）

此方法必须返回一个iterable，其中包含第一个要爬网的请求。当蜘蛛被打开以进行刮擦时，Scrapy会调用它。Scrapy只调用一次，因此`start_requests()`作为生成器实现是安全 的。

为每个URL 生成默认实现。`Request(url, dont_filter=True)` `start_urls`

如果要更改用于开始抓取域的请求，则这是要覆盖的方法。例如，如果您需要使用POST请求登录，则可以执行以下操作：

```
class MySpider(scrapy.Spider):
    name = 'myspider'

    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
```

parse（*回应*）

这是Scrapy在其请求未指定回调时处理下载的响应时使用的默认回调。

该`parse`方法负责处理响应并返回要删除的数据和/或更多URL。其他请求回调与`Spider`类具有相同的要求。

此方法以及任何其他Request回调必须返回可迭代的`Request`和/或dicts或`Item`对象。

参数： response（`Response`）- 对解析的响应

log（*消息*[，*级别*，*组件*]）

通过Spider发送日志消息的包装器，`logger`用于向后兼容。有关更多信息，请参阅 从蜘蛛记录。

closed（*原因*）

蜘蛛关闭时调用。此方法为signal.connect（）提供信号的快捷方式`spider_closed`。

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.logger.info('A response from %s just arrived!', response.url)
```

从单个回调中返回多个请求和项目：

```
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):
        for h3 in response.xpath('//h3').extract():
            yield {"title": h3}

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

而不是`start_urls`你可以`start_requests()`直接使用; 为数据提供更多结构，您可以使用项目：

```
import scrapy
from myproject.items import MyItem

class MySpider(scrapy.Spider):
    name = 'example.com'
    allowed_domains = ['example.com']

    def start_requests(self):
        yield scrapy.Request('http://www.example.com/1.html', self.parse)
        yield scrapy.Request('http://www.example.com/2.html', self.parse)
        yield scrapy.Request('http://www.example.com/3.html', self.parse)

    def parse(self, 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)
```

蜘蛛参数

蜘蛛可以接收修改其行为的参数。spider参数的一些常见用途是定义起始URL或将爬网限制到站点的某些部分，但它们可用于配置spider的任何功能。

`crawl` 使用该 `-a` 选项通过命令 传递Spider参数。例如：

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

蜘蛛可以在*__init__*方法中访问参数：

```
import scrapy

class MySpider(scrapy.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]
        # ...
```

默认的*__init__*方法将接受任何spider参数并将它们作为属性复制到spider。上面的例子也可以写成如下：

```
import scrapy

class MySpider(scrapy.Spider):
    name = 'myspider'

    def start_requests(self):
        yield scrapy.Request('http://www.example.com/categories/%s' % self.category)
```

请记住，蜘蛛参数只是字符串。蜘蛛本身不会进行任何解析。如果要从命令行设置*start_urls*属性，则必须使用ast.literal_eval 或json.loads 等方法将其自行解析为列表 ，然后将其设置为属性。否则，您将导致对*start_urls*字符串的迭代（一个非常常见的python陷阱），导致每个字符被视为一个单独的url。

有效的用例是设置由以下所用 `HttpAuthMiddleware` 的用户代理使用的http身份验证凭据 `UserAgentMiddleware` ：

```
scrapy crawl myspider -a http_user=myuser -a http_pass=mypassword -a user_agent=mybot
```

Spider参数也可以通过Scrapyd `schedule.json` API 传递。请参阅Scrapyd文档。

通用蜘蛛

Scrapy附带了一些有用的通用蜘蛛，您可以使用这些蜘蛛来为您的蜘蛛子类化。他们的目的是为一些常见的抓取案例提供方便的功能，例如根据特定规则跟踪站点上的所有链接，从站点地图抓取或解析XML / CSV Feed。

对于以下蜘蛛中使用的示例，我们假设您有一个 `TestItem` 在 `myproject.items` 模块中声明的项目：

```
import scrapy

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

抓取蜘蛛

类 **scrapy.spiders.CrawlSpider**

这是用于抓取常规网站的最常用的蜘蛛，因为它通过定义一组规则为跟踪链接提供了便利的机制。它可能不是最适合您的特定网站或项目，但它在几种情况下足够通用，因此您可以从它开始并根据需要覆盖它以获得更多自定义功能，或者只是实现您自己的蜘蛛。

除了从Spider继承的属性（您必须指定）之外，此类还支持一个新属性：

rules

这是一个（或多个） `Rule` 对象的列表。每个 `Rule` 定义用于爬网站点的特定行为。规则对象如下所述。如果多个规则匹配相同的链接，则将根据它们在此属性中定义的顺序使用第一个规则。

这个蜘蛛还暴露了一个可重写的方法：

parse_start_url（*回应*）

为start_urls响应调用此方法。它允许解析初始响应，并且必须返回 `Item` 对象， `Request` 对象或包含其中任何一个的iterable。

爬行规则

class **scrapy.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，它定义如何从每个已爬网页面中提取链接。

`callback` 是一个可调用的或一个字符串（在这种情况下，将使用具有该名称的spider对象的方法）为使用指定的link_extractor提取的每个链接调用。此回调接收响应作为其第一个参数，并且必须返回包含 `Item` 和/或 `Request` 对象（或其任何子类）的列表。

警告

编写爬网蜘蛛规则时，请避免使用`parse`回调，因为`CrawlSpider`使用`parse`方法本身来实现其逻辑。因此，如果您覆盖该`parse`方法，则爬网蜘蛛将不再起作用。

`cb_kwargs`是一个包含要传递给回调函数的关键字参数的dict。

`follow`是一个布尔值，它指定是否应该从使用此规则提取的每个响应中跟踪链接。如果`callback`是，则`follow`默认为`True`，否则默认为`False`。

`process_links`是一个可调用的，或一个字符串（在这种情况下，将使用来自具有该名称的蜘蛛对象的方法），将使用指定的每个响应提取的每个链接列表调用该方法`link_extractor`。这主要用于过滤目的。

`process_request`是一个可调用的，或一个字符串（在这种情况下，将使用来自具有该名称的spider对象的方法），该方法将在此规则提取的每个请求中调用，并且必须返回请求或None（以过滤掉请求）。

CrawlSpider示例

现在让我们看看一个带有规则的示例CrawlSpider：

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

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

    rules = (
        # Extract links matching 'category.php' (but not matching 'subsection.php')
        # and follow links from them (since no callback means follow=True by default).
        Rule(LinkExtractor(allow=('category\.php', ), deny=('subsection\.php', ))),

        # Extract links matching 'item.php' and parse them with the spider's method parse_item
        Rule(LinkExtractor(allow=('item\.php', )), callback='parse_item'),
    )

    def parse_item(self, response):
        self.logger.info('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
```

这个蜘蛛会开始抓取example.com的主页，收集类别链接和项目链接，使用该`parse_item`方法解析后者。对于每个项目响应，将使用XPath从HTML中提取一些数据，并将`Item`使用它填充。

XMLFeedSpider

类 scrapy.spiders.XMLFeedSpider

XMLFeedSpider旨在通过按某个节点名称迭代XML feed来解析XML feed。迭代器可以选自：`iternodes`，`xml`，和`html`。`iternodes`出于性能原因，建议使用迭代器，因为`xml`和`html`迭代器一次生成整个DOM以便解析它。但是，在使用`html`错误标记解析XML时，使用迭代器可能很有用。

要设置迭代器和标记名称，必须定义以下类属性：

iterator

一个字符串，它定义要使用的迭代器。它可以是：

- `'iternodes'` - 基于正则表达式的快速迭代器
- `'html'` - 使用的迭代器`Selector`。请记住，这使用DOM解析，并且必须在内存中加载所有DOM，这可能是大型Feed的问题
- `'xml'` - 使用的迭代器`Selector`。请记住，这使用DOM解析，并且必须在内存中加载所有DOM，这可能是大型Feed的问题

它默认为：`'iternodes'`。

itertag

一个字符串，其中包含要迭代的节点（或元素）的名称。示例：

```
itertag = 'product'
```

namespaces

一个元组列表，用于定义将使用此spider处理的该文档中可用的名称空间。该 和将用于使用该方法自动注册名称空间 。`(prefix, uri)` `prefix` `uri` `register_namespace()`

然后，您可以在`itertag`属性中指定具有名称空间的节点。

例：

```
class YourSpider(XMLFeedSpider):

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

除了这些新属性之外，这个蜘蛛还具有以下可重写方法：

adapt_response（*回应*）

在蜘蛛开始解析之前，一旦从蜘蛛中间件到达就接收响应的方法。它可以在解析之前用于修改响应主体。此方法接收响应并返回响应（可以是相同或另一个）。

parse_node（*响应*，*选择器*）

对于与提供的标记名称（`itertag`）匹配的节点，调用此方法。接收`Selector`每个节点的响应和响应 。必须覆盖此方法。否则，你的蜘蛛将无法正常工作。此方法必须返回`Item`对象，`Request`对象或包含其中任何对象的iterable。

process_results（*回应*，*结果*）

为蜘蛛返回的每个结果（项目或请求）调用此方法，并且它旨在执行将结果返回到框架核心之前所需的任何上次处理，例如设置项目ID。它接收结果列表和产生这些结果的响应。它必须返回结果列表（项目或请求）。

XMLFeedSpider示例

这些蜘蛛很容易使用，让我们来看一个例子：

```
from scrapy.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):
        self.logger.info('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
```

基本上我们在那里做的是创建一个蜘蛛，从给定的下载源`start_urls`，然后遍历每个`item`标签，打印出来，并存储一些随机数据`Item`。

CSVFeedSpider

类 **scrapy.spiders.CSVFeedSpider**

这个spider与XMLFeedSpider非常相似，只不过它遍历行而不是节点。在每次迭代中调用的方法是`parse_row()`。

delimiter

CSV文件中每个字段具有分隔符的字符串默认为`','`（逗号）。

quotechar

带有CSV文件中每个字段的机箱字符的字符串默认为`'"'`（引号）。

headers

CSV文件中的列名列表。

> **parse_row（*响应*，*行*）**
>
> 使用CSV文件的每个提供（或检测到的）标头的密钥接收响应和dict（表示每行）。该蜘蛛还提供了覆盖 `adapt_response` 和 `process_results` 用于预处理和后处理目的的的方法的机会。

CSVFeedSpider示例

让我们看一个与前一个类似的示例，但使用 `CSVFeedSpider` ：

```
from scrapy.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):
        self.logger.info('Hi, this is a row!: %r', row)

        item = TestItem()
        item['id'] = row['id']
        item['name'] = row['name']
        item['description'] = row['description']
        return item
```

SitemapSpider

类 **scrapy.spiders.SitemapSpider**

SitemapSpider允许您通过使用Sitemaps发现URL来抓取 网站。

它支持嵌套的站点地图，并从robots.txt中发现站点地图网址 。

> **sitemap_urls**
>
> 指向要抓取其网址的站点地图的网址列表。
>
> 您还可以指向robots.txt，它将被解析以从中提取站点地图网址。

> **sitemap_rules**
>
> 元组列表，其中： `(regex, callback)`
>
> - `regex` 是一个正则表达式，用于匹配从站点地图中提取的网址。 `regex` 可以是str或编译的正则表达式对象。
> - callback是用于处理与正则表达式匹配的url的回调。 `callback` 可以是字符串（表示蜘蛛方法的名称）或可调用的。

例如：

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

规则按顺序应用，并且仅使用匹配的第一个规则。

如果省略此属性，将使用`parse`回调处理站点地图中找到的所有网址。

sitemap_follow

应遵循的站点地图的正则表列表。这仅适用于使用指向其他站点地图文件的站点地图索引文件的站点。

默认情况下，将遵循所有站点地图。

sitemap_alternate_links

指定是否`url`应遵循一个备用链接。这些是在同`url`一块内传递的另一种语言的同一网站的链接。

例如：

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

使用`sitemap_alternate_links`set，这将检索两个URL。随着`sitemap_alternate_links`禁用，只`http://example.com/`将被检索。

默认为`sitemap_alternate_links`禁用。

SitemapSpider示例

最简单的示例：使用`parse`回调处理通过站点地图发现的所有网址 ：

```
from scrapy.spiders import SitemapSpider

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

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

处理一些具有特定回调的网址和其他具有不同回调的网址：

```
from scrapy.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 ...
```

遵循robots.txt文件中定义的站点地图，并且只关注其网址包含以下内容的站点地图 `/sitemap_shop` ：

```
from scrapy.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 ...
```

将SitemapSpider与其他网址源结合使用：

```
from scrapy.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 ...
```