**一、安装scrapy**

1.1linux系统使用：pip install scrapy

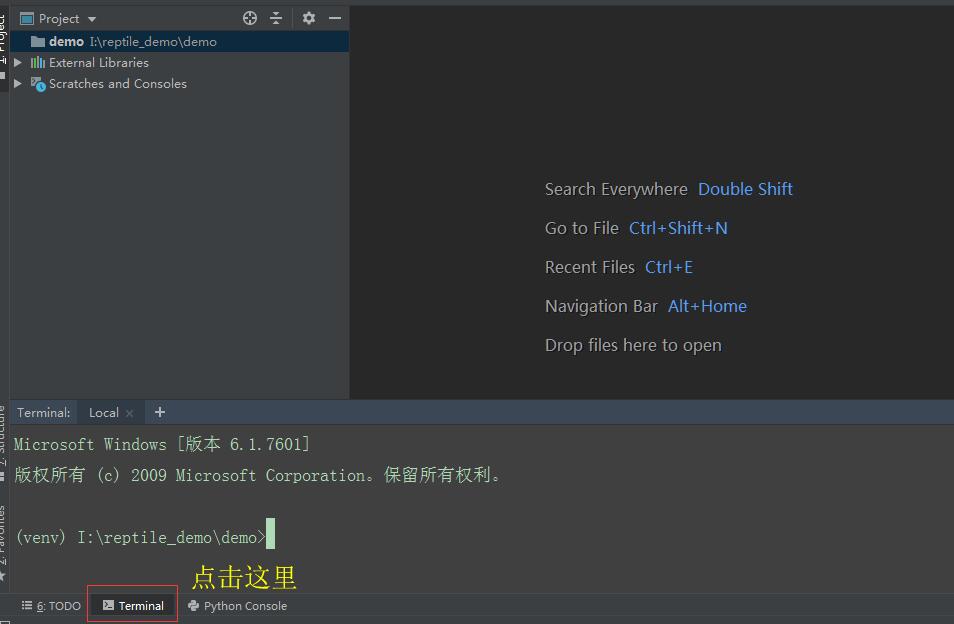
1.2Windows系统：

* pip install wheel
* 下载twisted http://www.lfd.uci.edu/~gohlke/pythonlibs/#twisted  （根据Python的版本进行下载，这里我的Python版本是3.7所以就下的3.7）
* pip install 路径\Twisted-19.2.1-cp37-cp37m-win\_amd64
* pip install pywin32
* pip install scrapy

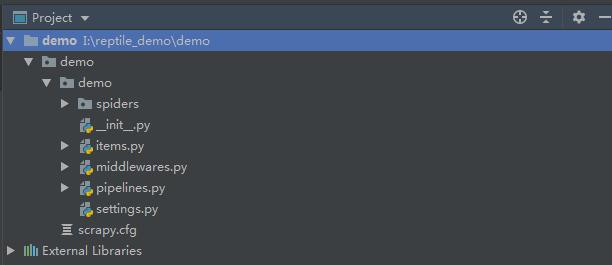
**二、创建scrapy项目**

1、新建一个项目，选择Python即可。我这里创建的项目名是demo。创建好后是一个空的项目。

2、点击pycharm下面的terminal，如下图所示：



在终端中输入：**scrapy startproject demo** 命令，创建scrapy项目，创建成功后会出现如下目录结构：



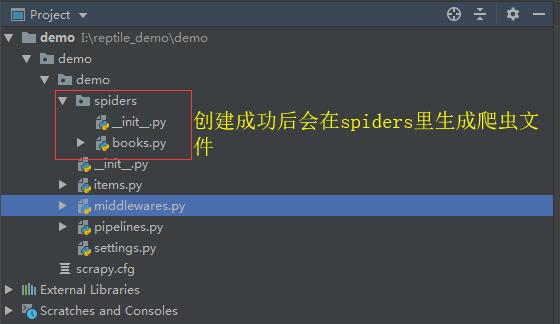
各文件作用大致如下：

* scrapy.cfg:：项目的配置文件
* demo/：该项目的python模块。在此加入代码。
* demo/items.py：项目中的item文件主要用于定义数据的结构化存储，类似于ORM中的models。
* demo/pipelines.py：项目中的pipelines文件，指定数据的存储方式（以文件的形式存储，存储到数据库中）。
* demo/settings.py：项目的设置文件.
* demo/spiders/：放置spider代码的目录。我们写的爬虫代码在这个目录下。

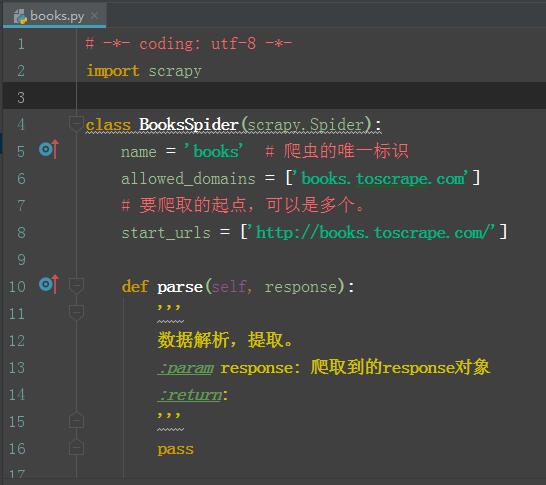
**3、创建爬虫文件**

3.1在终端中输入：cd demo（我这里输入demo是因为我的项目名是demo）

3.2在终端中输入：**scrapy genspider books books.toscrape.com** (scrapy genspider  应用名称 爬取网页的起始url)

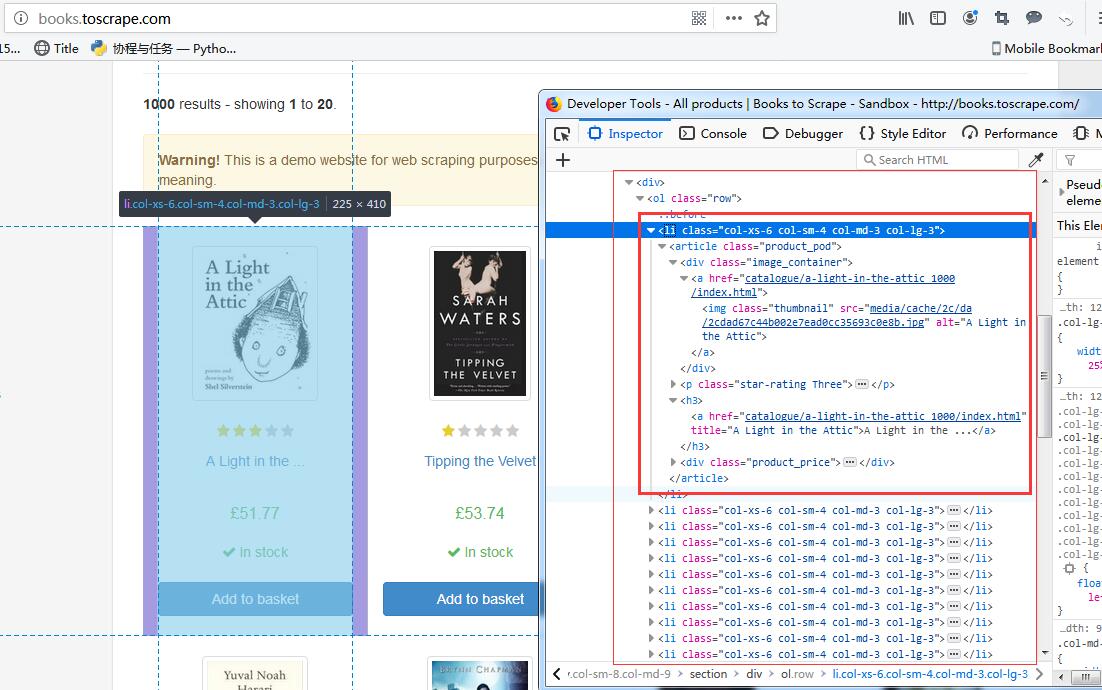


**4、打开books文件，该文件结构如下：**

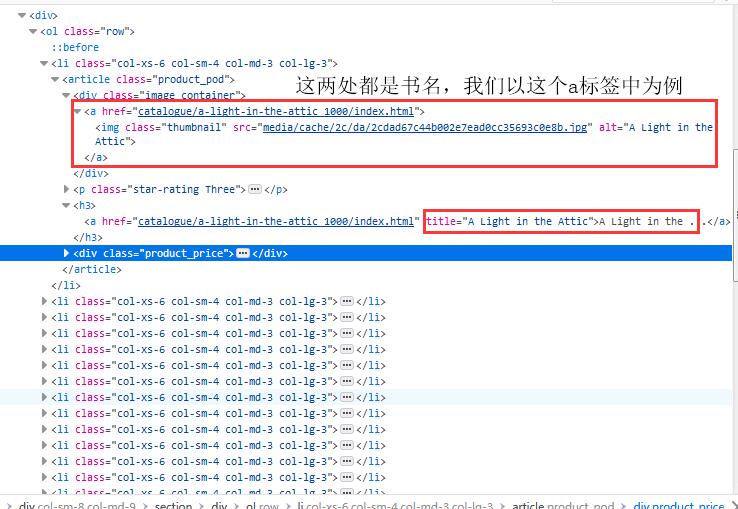


**5、爬取http://books.toscrape.com/的书籍信息。**

5.1分析http://books.toscrape.com/页面。



由上图我们可以知道所有书籍都存放在div/ol/下的li标签中。这里我们只打印书名，由此我们可以像下面这样写来提取数据。



5.2books中的部分代码如下：

[复制代码](javascript:void(0);)

def parse(self, response):

'''

数据解析，提取。

:param response: 爬取到的response对象

:return:

'''

book\_list = response.xpath('/html/body/div/div/div/div/section/div[2]/ol/li')

for book in book\_list:

print(book.xpath('./article/div[1]/a/img/@alt').extract())

[复制代码](javascript:void(0);)

5.3在setting.py中配置如下：

USER\_AGENT = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:67.0) Gecko/20100101 Firefox/67.0' # UA头

ROBOTSTXT\_OBEY = False # 如果为True表示准信robots协议，则大多数数据都爬不了。所以这里设置为Flase

LOG\_LEVEL = 'ERROR' # 日志等级

5.4在终端中执行爬取命令：scrapy crawl books

[复制代码](javascript:void(0);)

# 打印内容如下

['A Light in the Attic']

['Tipping the Velvet']

['Soumission']

['Sharp Objects']

['Sapiens: A Brief History of Humankind']

['The Requiem Red']

['The Dirty Little Secrets of Getting Your Dream Job']

['The Coming Woman: A Novel Based on the Life of the Infamous Feminist, Victoria Woodhull']

['The Boys in the Boat: Nine Americans and Their Epic Quest for Gold at the 1936 Berlin Olympics']

['The Black Maria']

['Starving Hearts (Triangular Trade Trilogy, #1)']

["Shakespeare's Sonnets"]

['Set Me Free']

["Scott Pilgrim's Precious Little Life (Scott Pilgrim #1)"]

['Rip it Up and Start Again']

['Our Band Could Be Your Life: Scenes from the American Indie Underground, 1981-1991']

['Olio']

['Mesaerion: The Best Science Fiction Stories 1800-1849']

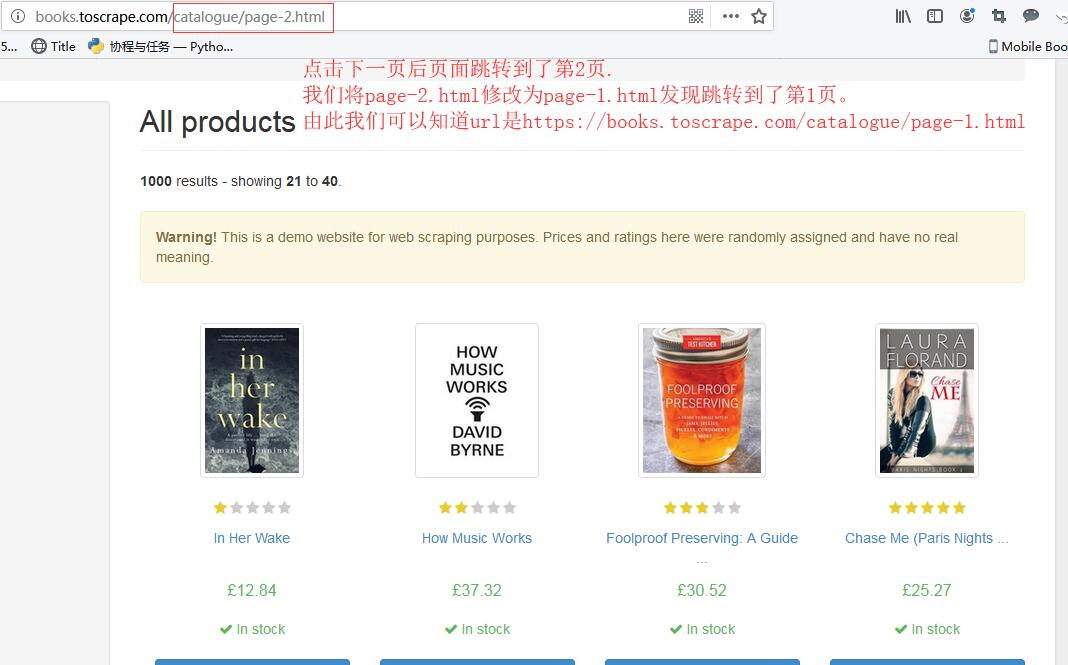
['Libertarianism for Beginners']

["It's Only the Himalayas"]

[复制代码](javascript:void(0);)

由此我们可以看出这里只是爬取了1页，下面来爬取所有书籍名称。

**6、爬取所有页面的书籍。**



最终books.py的内容看起来像下面这样：

[复制代码](javascript:void(0);)

# -\*- coding: utf-8 -\*-

import scrapy

class BooksSpider(scrapy.Spider):

name = 'books' # 爬虫的唯一标识

allowed\_domains = ['books.toscrape.com']

# 要爬取的起点，可以是多个。

start\_urls = ['http://books.toscrape.com/']

url = 'http://books.toscrape.com/catalogue/page-%d.html' # url模板用于拼接新的url

page\_num = 2

def parse(self, response):

'''

数据解析，提取。

:param response: 爬取到的response对象

:return:

'''

print(f'当前页数{self.page\_num}') # 打印当前页数的数据

book\_list = response.xpath('/html/body/div/div/div/div/section/div[2]/ol/li')

for book in book\_list:

print(book.xpath('./article/div[1]/a/img/@alt').extract())

if self.page\_num < 50: # 总共50页的内容

new\_url = format(self.url % self.page\_num) # 拼接处新的URL

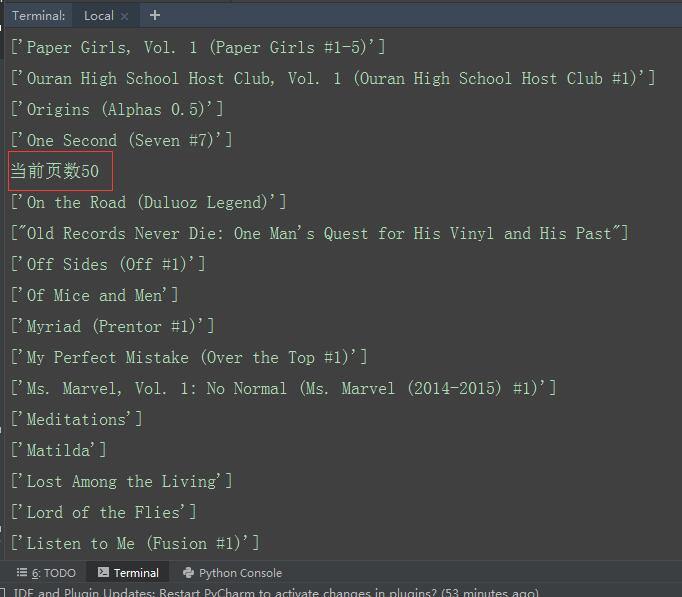
self.page\_num += 1 # 页数加1

yield scrapy.Request(url=new\_url, callback=self.parse) # 手动发送请求

[复制代码](javascript:void(0);)

在终端中执行命令获取书名：**scrapy crawl books**

如果一切顺利你会看到打印的最终部分结果如下：



**今日小结：**

* 创建scrapy项目：scrapy startproject 爬虫项目名称。
* 创建爬虫应用：scrapy genspider books books.toscrape.com （(scrapy genspider  应用名称 爬取网页的起始url)）应用名称在整个项目中作为唯一标识，不能出现同名的爬虫应用。
* 运行爬虫程序：scrapy crawl books（scrapy  crawl  爬虫应用）。
* parse方法：当一个页面下载完成后，Scrapy引擎会回调一个我们指定的页面解析函数（默认为parse方法）解析页面。一个页面解析函数通常需要完成以下两个任务：

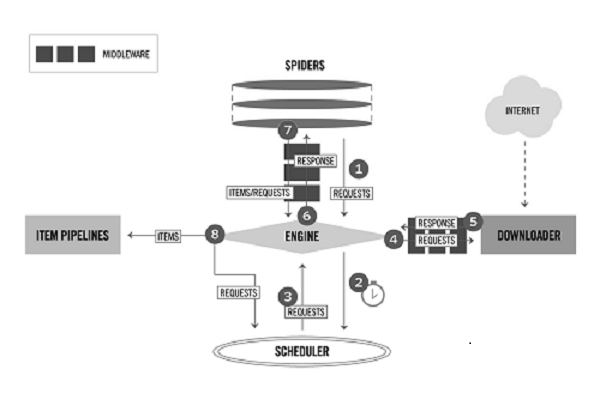
　　　　1、提取页面中的数据（使用XPath或CSS选择器）。  
　　　　2、提取页面中的链接，并产生对链接页面的下载请求。

* 页面解析函数通常被实现成一个生成器函数，每一项从页面中提取的数据以及每一个对链接页面的下载请求都由yield语句提交给Scrapy引擎。

**parse方法的工作机制（来源网络）：**

1. 因为使用的yield，而不是return。parse函数将会被当做一个生成器使用。scrapy会逐一获取parse方法中生成的结果，并判断该结果是一个什么样的类型；
2. 如果是request则加入爬取队列，如果是item类型则使用pipeline处理，其他类型则返回错误信息。
3. scrapy取到第一部分的request不会立马就去发送这个request，只是把这个request放到队列里，然后接着从生成器里获取；
4. 取尽第一部分的request，然后再获取第二部分的item，取到item了，就会放到对应的pipeline里处理；
5. parse()方法作为回调函数(callback)赋值给了Request，指定parse()方法来处理这些请求 scrapy.Request(url, callback=self.parse)
6. Request对象经过调度，执行生成 scrapy.http.response()的响应对象，并送回给parse()方法，直到调度器中没有Request（递归的思路）
7. 取尽之后，parse()工作结束，引擎再根据队列和pipelines中的内容去执行相应的操作；
8. 程序在取得各个页面的items前，会先处理完之前所有的request队列里的请求，然后再提取items。
9. 这一切的一切，Scrapy引擎和调度器将负责到底。

## ****scrapy结构图：****



## ****scrapy组件：****

* ENGINE：引擎，框架的核心，其它所有组件在其控制下协同工作。
* SCHEDULER：调度器，负责对SPIDER提交的下载请求进行调度。
* DOWNLOADER：下载器，负责下载页面（发送HTTP请求/接收HTTP响应）。
* SPIDER：爬虫，负责提取页面中的数据，并产生对新页面的下载请求。
* MIDDLEWARE：中间件，负责对Request对象和Response对象进行处理。
* ITEM PIPELINE：数据管道，负责对爬取到的数据进行处理。

对于用户来说，Spider是最核心的组件，Scrapy爬虫开发是围绕实现Spider展开的。

## ****框架中的数据流：****

* REQUEST：scrapy中的HTTP请求对象。
* RESPONSE：scrapy中的HTTP响应对象。
* ITEM：从页面中爬取的一项数据。

Request和Response是HTTP协议中的术语，即HTTP请求和HTTP响应，Scrapy框架中定义了相应的Request和Response类，这里的Item代表Spider从页面中爬取的一项数据。

## ****scrapy大致工作流程：****

1. 当SPIDER要爬取某URL地址的页面时，需使用该URL构造一个Request对象，提交给ENGINE。
2. ENGINE将Request对象传给SCHEDULER，SCHEDULER对URL进行去重，按某种算法进行排队，之后的某个时刻SCHEDULER将其出队，将处理好的Request对象返回给ENGINE。
3. ENGINE将SCHEDULER处理后的Request对象发送给DOWNLOADER下载页面。
4. DOWNLOADER根据MIDDLEWARE的规则，使用Request对象中的URL地址发送一次HTTP请求到网站服务器，之后用服务器返回的HTTP响应构造出一个Response对象，其中包含页面的HTML文本。DOWNLOADER将结果Resopnse对象传给ENGINE。
5. ENGINE将Response对象发送给SPIDER的页面解析函数（构造Request对象时指定）进行处理，页面解析函数从页面中提取数据，封装成Item后提交给ENGINE。
6. ENGINE将Item送往ITEMPIPELINES进行处理，最终以某种数据格式写入文件（csv，json）或者存储到数据库中。

#### 整个流程的核心都是围绕着ENGINE进行的。

## Request对象

Request对象用来描述一个HTTP请求，下面是其构造器方法的参数列表。

[复制代码](javascript:void(0);)

Request(url, callback=None, method='GET', headers=None, body=None,

cookies=None, meta=None, encoding='utf-8', priority=0,

dont\_filter=False, errback=None, flags=None)

# url（必选）：请求页面的url地址，bytes或str类型，如'http://www.baidu.com'。

# callback:页面解析函数， Callable类型，Request对象请求的页面下载完成后，由该参数指定的页面解析函数被调用。如果未传递该参数，默认调用Spider的parse方法。

# method：HTTP请求的方法，默认为'GET'。

# headers：HTTP请求的头部字典，dict类型，例如{'Accept':'text/html', 'User-Agent':Mozilla/5.0'}。如果其中某项的值为None，就表示不发送该项HTTP头部，例如{'Cookie':None}，禁止发送Cookie。

# body：HTTP请求的正文，bytes或str类型。

# cookies：Cookie信息字典，dict类型，例如{'currency': 'USD','country': 'UY'}。

# meta：Request的元数据字典，dict类型，用于给框架中其他组件传递信息，比如中间件Item Pipeline。其他组件可以使用Request对象的meta属性访问该元数据字典（request.meta），也用于给响应处理函数传递信息，  
# 详见Response的meta属性。

# encoding：url和body参数的编码默认为'utf-8'。如果传入的url或body参数是str类型，就使用该参数进行编码。

# priority：请求的优先级默认值为0，优先级高的请求优先下载。

# dont\_filter：默认情况下（dont\_filter=False），对同一个url地址多次提交下载请求，后面的请求会被去重过滤器过滤（避免重复下载）。如果将该参数置为True，可以使请求避免被过滤，强制下载。例如，在多次爬取  
# 一个内容随时间而变化的页面时（每次使用相同的url），可以将该参数置为True。

# errback：请求出现异常或者出现HTTP错误时（如404页面不存在）的回调函数。

[复制代码](javascript:void(0);)

虽然参数很多，但除了url参数外，其他都带有默认值。在构造Request对象时，通常我们只需传递一个url参数或再加一个callback参数，其他使用默认值即可。

## ****Response对象：****

Response对象用来描述一个HTTP响应，Response只是一个基类，根据响应内容的不同有如下子类：

* TextResponse
* HtmlResponse
* XmlResponse

当一个页面下载完成时，下载器依据HTTP响应头部中的Content-Type信息创建某个Response的子类对象。我们通常爬取的网页，其内容是HTML文本，创建的便是HtmlResponse对象，其中HtmlResponse和XmlResponse是TextResponse的子类。实际上，这3个子类只有细微的差别，这里以HtmlResponse为例进行讲解。

下面是HtmlResponse对象的属性及方法。

[复制代码](javascript:void(0);)

url：HTTP响应的url地址，str类型。

status：HTTP响应的状态码，int类型，例如200，404。

headers：HTTP响应的头头部，类字典类型，可以调用get或getlist方法对其进行访问，例如：response.headers.get('Content-Type') response.headers.getlist('Set-Cookie')

body：HTTP响应正文，bytes类型。

text：文本形式的HTTP响应正文，str类型，它是由response.body使用response.encoding解码得到的，即reponse.text = response.body.decode(response.encoding)

encoding：HTTP响应正文的编码，它的值可能是从HTTP响应头部或正文中解析出来的。

request：产生该HTTP响应的Request对象。

meta：即response.request.meta，在构造Request对象时，可将要传递给响应处理函数的信息通过meta参数传入；响应处理函数处理响应时，通过response.meta将信息取出。

selector：Selector对象用于在Response中提取数据。

xpath(query)：使用XPath选择器在Response中提取数据，实际上它是response.selector.xpath方法的快捷方式。

css(query)：使用CSS选择器在Response中提取数据，实际上它是response.selector.css方法的快捷方式。

urljoin（url）：用于构造绝对url。当传入的url参数是一个相对地址时，根据response.url计算出相应的绝对url。例如：

response.url为http://www.example.com/a，url为b/index.html，调用response.urljoin(url)的结果为http://www.example.com/a/b/index.html。

[复制代码](javascript:void(0);)

虽然HtmlResponse对象有很多属性，但最常用的是以下的3个方法：

* xpath(query)
* css(query)
* urljoin(url)

前两个方法用于提取数据，后一个方法用于构造绝对url。

## ****spied开发流程****

实现一个Spider子类的过程很像是完成一系列填空题，Scrapy框架提出以下问题让用户在Spider子类中作答：

* 爬虫从哪个或哪些页面开始爬取？
* 对于一个已下载的页面，提取其中的哪些数据？
* 爬取完当前页面后，接下来爬取哪个或哪些页面？

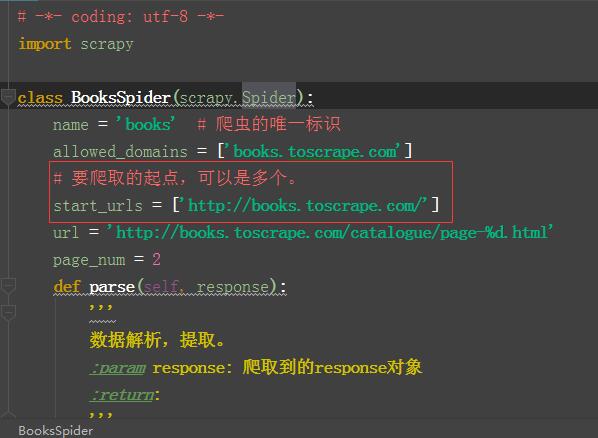
实现一个Spider只需要完成下面4个步骤：

* 继承scrapy.Spider。
* 为Spider取名。
* 设定起始爬取点。
* 实现页面解析函数。

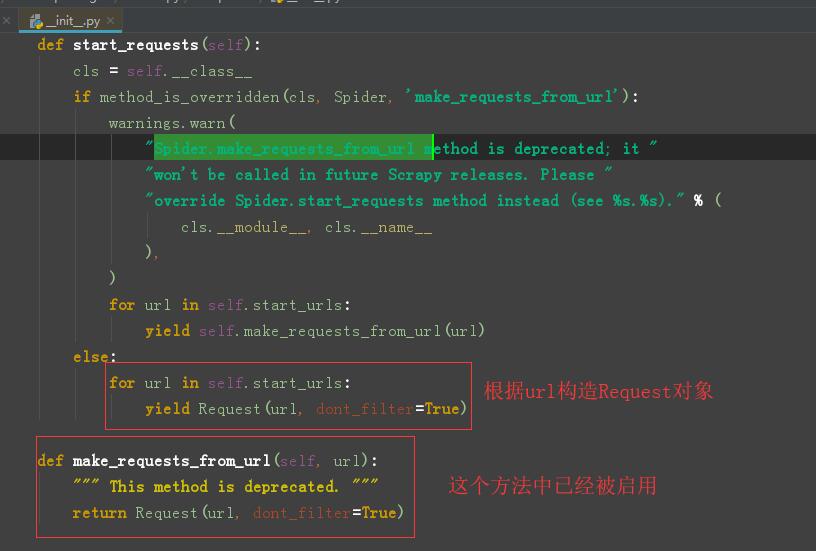
scrapy.Spider基类实现了以下内容:

* 供Scrapy引擎调用的接口，例如用来创建Spider实例的类方法from\_crawler。
* 供用户使用的实用工具函数，例如可以调用log方法将调试信息输出到日志。
* 供用户访问的属性，例如可以通过settings属性访问配置文件中的配置。

 关于起始URL start\_urls：



start\_urls通常被实现成一个列表，其中放入所有起始爬取点的url（例子中只有一个起始点）。看到这里，大家可能会想，请求页面下载不是一定要提交Request对象么？而我们仅定义了url列表，是谁  
暗中构造并提交了相应的Request对象呢？

1. 我们将起始URL提交给ENGINE。
2. ENGINE调用start\_requests方法，我们没有实现整个方法，所以调用了基类的start\_requests方法。
3. 通过阅读Spider基类的源码可以看到如下内容：
4. 
5. 基类的start\_requests将我们的URL封装成Request对象。

由此我们知道Request对象是调用基类start\_requests方法产生的，因此我们也可以自己定义start\_requests方法（覆盖基类Spider的start\_requests方法），直接构造并提交起始爬取点的Request对象。在某些场景下使用这种方式更加灵活，例如有时想为Request添加特定的HTTP请求头部，或想为Request指定特定的页面解析函数。

## ****页面解析函数parse:****

页面解析函数也就是构造Request对象时通过callback参数指定的回调函数（或默认的parse方法）。页面解析函数是实现Spider中最核心的部分，它需要完成以下两项工作：

* 使用选择器提取页面中的数据，将数据封装后（Item或字典）提交给Scrapy引擎。
* 使用选择器或LinkExtractor提取页面中的链接，用其构造新的Request对象并提交给Scrapy引擎（下载链接页面）。

一个页面中可能包含多项数据以及多个链接，因此页面解析函数被要求返回一个可迭代对象（通常被实现成一个生成器函数），每次迭代返回一项数据（Item或字典）或一个Request对象。  
