5. 第五章：Scrapy 框架

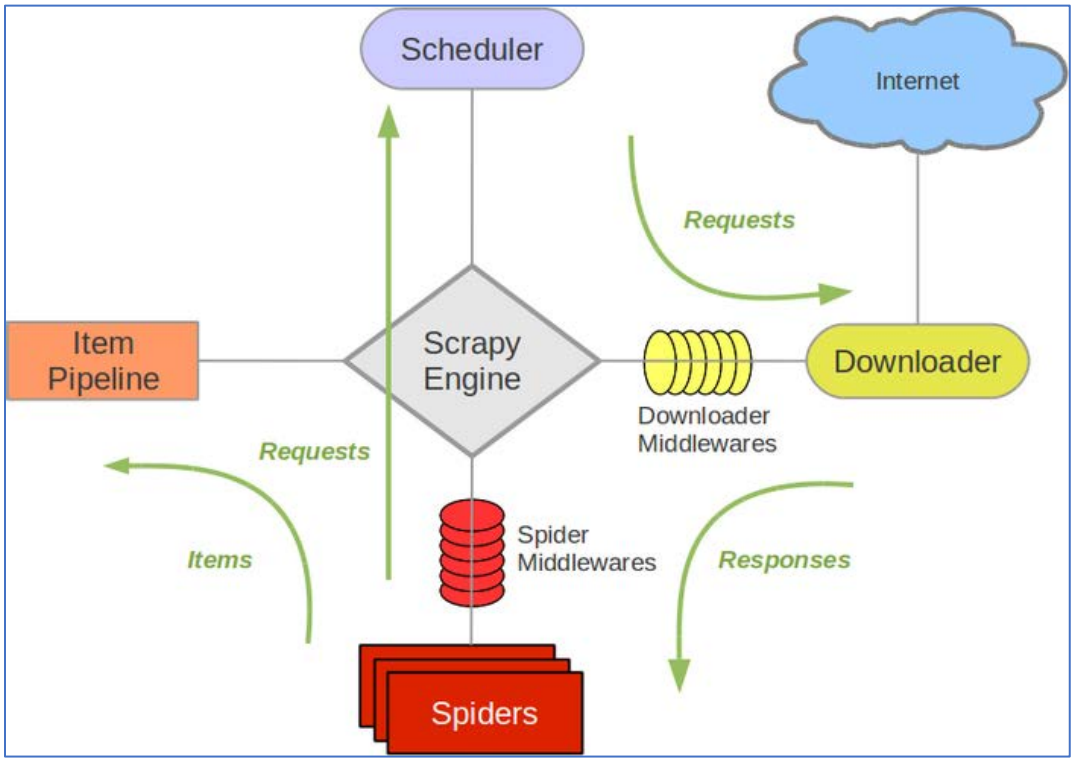
5.1. 框架架构

Scrapy框架介绍

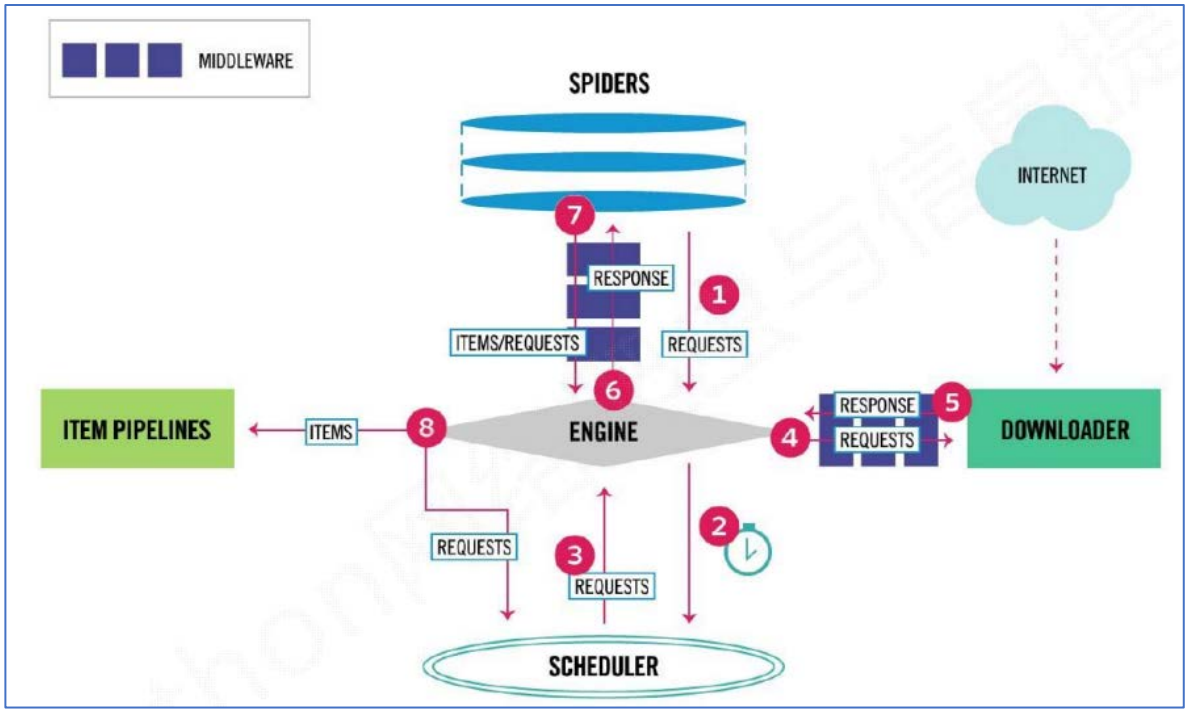
写一个爬虫，需要做很多的事情. 比如: 发送网络请求，数据解析，数据存储，反反爬虫机制 (更换 ip 代理，设置请求头等)，异步请求等. 这些工作如果每次都要自己从零开始写的话，比较浪费时间. 因此 Scrapy 把一些基础的东西封装好了，在他上面写爬虫可以变的更加的高效 (爬取效率和开发效率). 因此真正在公司里，一些上了量的爬虫，都是使用 Scrapy 框架来解决.

Scrapy架构图

1. 流程图 (1)



2. 流程图 (2)



Scrapy框架模块功能

1. Scrapy Engine (引擎): Scrapy 框架的核心部分. 负责在 Spider 和 ItemPipeline, Downloader, Scheduler 中间通信, 传递数据等.
2. Scheduler (调度器): 负责接收引擎发送过来的请求, 并按照一定的方式进行排列和整理, 负责调度请求的顺序等.
3. Spider (爬虫): 发送需要爬取的链接给引擎, 最后引擎把其他模块请求回来的数据再发送给爬虫, 爬虫就去解析想要的数据. 这个部分是我们开发者自己写的, 因为要爬取哪些链接, 页面中的哪些数据是我们需要的, 都是由程序员自己决定.
4. Downloader (下载器): 负责接收引擎传过来的下载请求, 然后去网络上下载对应的数据再交还给引擎.
5. Item Pipeline (管道): 负责将 Spider (爬虫) 传递过来的数据进行保存. 具体保存在哪里, 应该看开发者自己的需求.
6. Downloader Middlewares (下载中间件): 可以扩展下载器和引擎之间通信功能的中间件.
7. Spider Middlewares (Spider 中间件): 可以扩展引擎和爬虫之间通信功能的中间件.

5.2. scrapy 安装

Scrapy 官方文档: http://doc.scrapy.org/en/latest
Scrapy 中文文档: http://scrapy-chs.readthedocs.io/zh_CN/latest/index.html
具体 Scrapy 安装流程参考, 里面有各个平台的安装方法:

http://doc.scrapy.org/en/latest/intro/install.html#intro-install-platform-notes

Linux安装

在 ubuntu 上安装 scrapy 之前，需要先安装以下依赖
pip install --upgrade pip

sudo apt-get install python-dev python3-dev build-essential python-pip libxml2-dev libxslt1-dev zlib1g-dev libffi-dev libssl-dev

然后再通过 pip install scrapy 安装.

Windows安装

先使用 pip install scrapy 来安装，在安装过程中哪个包出错就使用 pip install 来单独安装这个包，如果多次出错，就在下面网站中找到对应的安装包下载.whl 文件，再使用 pip install 来离线安装.

python --version 查看 python 的版本，搜索与 python 对应版本的安装包
https://www.lfd.uci.edu/~gohlke/pythonlibs/

Windows 安装详细流程

1，首先，升级 pip，建议网络安装：
python -m pip install --upgrade pip

2，安装 wheel（建议网络安装）
pip install wheel

3，安装 lxml (下载安装)
pip install lxml
https://pypi.python.org/pypi/lxml/

4，安装 Twisted (下载安装)
pip install Twisted-18.4.0-cp36-cp36m-win_amd64.whl

5，安装 scrapy
pip install scrapy 或 pip install scrapy==1.1.0rc3(建议网络安装)

6，下载安装 pywin32 并配置
如果在 windows 系统下，提示这个错误 ModuleNotFoundError: No module named 'win32api'，那么使用以下命令可以解决

pip install pypiwin32
pip install pypiwin32 -i https://pypi.douban.com/simple

PyWin32 provides extensions for Windows.
To install pywin32 system files, run `python.exe Scripts/pywin32_postinstall.py -install` from an elevated command prompt.

pip install pywin32-223.1-cp36-cp36m-win_amd64.whl

C:\Users\David\Envs\python3_spider\Lib\site-packages\pywin32_system32
把其中的这两个文件复制到 C:\windows\system32 中
pythoncom36.dll
pywintypes36.dll

5.3 scrapy.Spider 类爬虫

scrapy常用命令

1. scrapy 命令行工具

1.创建一个新的项目
```
scrapy startproject [项目名]
```

2.生成爬虫
```
scrapy genspider +文件名+网址
```

3.运行爬虫 scrapy crawl 爬虫名

```
scrapy crawl [爬虫名]
scrapy crawl [爬虫名] -o zufang.json
# -o output
scrapy crawl [爬虫名] -o zufang.csv
```

4.check 检查错误

```
# 项目根目录运行
scrapy check
```

5.list 返回项目所有 spider 名称

```
scrapy list
```

6. view 存储、打开网页

```
scrapy view https://www.baidu.com
```

scrapy open 打开一个本地的路径

7. scrapy shell，进入终端

```
scrapy shell https://www.baidu.com
```

```
# scrapy shell 常见命令
```

```
response.text
response.status
response.headers
request.headers
request.meta
response._url
```

8. 运行爬虫 scrapy runspider 爬虫文件名

```
# 进入到 spiders 文件夹下，运行爬虫
scrapy runspider zufang_spider.py
```

```
# 直接把爬虫中 yield 的 item 中的内容保存到文件中
```

```
scrapy crawl 爬虫名 -o zufang.json
scrapy crawl 爬虫名 -o zufang.csv
```

使用Scrapy框架爬取糗事百科段子

更多帮助文件

```
scrapy -h
scrapy genspider -h
```

scrapy 常用命令

```
(python3_spider) C:\Users\David>scrapy
```

Scrapy 1.5.0 - no active project

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

Available commands:
测试电脑的性能
  bench   Run quick benchmark test
爬某一个网页
  fetch   Fetch a URL using the Scrapy downloader
创建爬虫
  genspider   Generate new spider using pre-defined templates
运行爬虫
  runspider   Run a self-contained spider (without creating a project)
  settings   Get settings values
  shell   Interactive scraping console
  startproject Create new project
  version   Print Scrapy version
把源码下载下来，并使用浏览器打开
  view   Open URL in browser, as seen by Scrapy

scrapy 创建并编写爬虫项目的步骤

定义 items.py 确定爬取的目标.
编写 spider 爬虫
修改 pipelines.py 进行爬后数据的处理，写入文件及数据库
以上 3 个过程中都可能会修改 settings.py 中的设置

创建项目

进入到你想把这个项目存放的目录
scrapy startproject [项目名称]

scrapy startproject qsbk_pro

目录结构介绍

tree 命令生成目录树
可以在命令行窗口敲"tree /?"看帮助。
tree 命令的格式是
tree [drive][path] [/F] [/A] [>PRN]

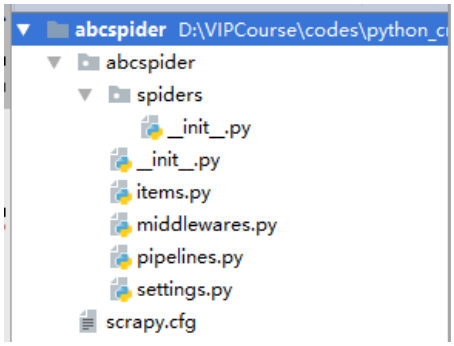
使用/F 参数时显示所有目录及目录下的所有文件，省略时，只显示目录，不显示目录下的文件；
/A 使用 ASCII 字符，而不使用扩展字符
选用>PRN 参数时，则把所列目录及目录中文件名打印输出

例如：tree /f >tree.txt

scrapy startproject ts_first

```
(python3_spider) C:\Users\David>tree ts_first /F
Folder PATH listing
Volume serial number is 1856-459B
C:\USERS\DAVID\TS_FIRST
    scrapy.cfg
    
    ts_first
        items.py
        middlewares.py
        pipelines.py
        settings.py
        __init__.py
        
        spiders
            __init__.py
            
            __pycache__
        __pycache__
```

最外层的 ts_first 为项目目录
ts_first/ts_first 为核心目录



以下介绍下主要文件的作用

1. items.py: 用来存放爬虫爬取下来数据的模型.

2. middlewares.py: 用来存放各种中间件的文件.
3. pipelines.py: 用来将 items 的模型存储到本地磁盘中.
4. settings.py: 本爬虫的一些配置信息 (比如请求头, 多久发送一次请求, ip 代理池等) .
5. scrapy.cfg: 项目的配置文件.
6. spiders 包: 以后所有的爬虫, 都是存放到这个里面.

创建爬虫

进入到 scrapy 项目根目录中, 再次运行 scrapy, 这时的命令为项目命令
(python3_spider) C:\Users\David>cd ts_first

(python3_spider) C:\Users\David\ts_first>scrapy
Scrapy 1.5.0 - project: ts_first

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

Available commands:
  bench   Run quick benchmark test
  check   Check spider contracts
  crawl   Run a spider
  edit   Edit spider
  fetch   Fetch a URL using the Scrapy downloader
  genspider   Generate new spider using pre-defined templates
  list   List available spiders
  parse   Parse URL (using its spider) and print the results
  runspider   Run a self-contained spider (without creating a project)
  settings   Get settings values
  shell   Interactive scraping console
  startproject Create new project
  version   Print Scrapy version
  view   Open URL in browser, as seen by Scrapy

Use "scrapy <command> -h" to see more info about a command

scrapy genspider -t 模版 爬虫文件名 域名

查看可用的爬虫模板

scrapy genspider --list
scrapy genspider --l

基本爬虫
  basic

通用爬虫
   crawl
csv 格式数据的爬虫
   csvfeed
xml 格式数据的爬虫
   xmlfeed

创建爬虫的命令

```
(python3_spider) C:\Users\David\ts_first>scrapy genspider
Usage
=====
   scrapy genspider [options] <name> <domain>

Generate new spider using pre-defined templates

Options
=======
--help, -h                  show this help message and exit
--list, -l                  List available templates
--edit, -e                  Edit spider after creating it
--dump=TEMPLATE, -d TEMPLATE
                            Dump template to standard output
--template=TEMPLATE, -t TEMPLATE
                            Uses a custom template.
--force                     If the spider already exists, overwrite it with the
                            template

Global Options
--------------
--logfile=FILE              log file. if omitted stderr will be used
--loglevel=LEVEL, -L LEVEL
                            log level (default: DEBUG)
--nolog                     disable logging completely
--profile=FILE              write python cProfile stats to FILE
--pidfile=FILE              write process ID to FILE
--set=NAME=VALUE, -s NAME=VALUE
                            set/override setting (may be repeated)
--pdb                       enable pdb on failure
```

进入项目根目录, 在项目根目录中创建爬虫. 注意, 爬虫名字不能和项目名称相同.

```
scrapy gensipder qsbk "qiushibaike.com"
```

创建了一个名字叫做qsbk的爬虫，并且能爬取的网页只会限制在qiushibaike.com这个域名下.

查看当前项目中的爬虫
(python3_spider) C:\Users\David\ts_first>scrapy list

爬虫代码解析

```
import scrapy

class QsbkSpider(scrapy.Spider):
    name = 'qsbk'
    allowed_domains = ['qiushibaike.com']
    start_urls = ['http://qiushibaike.com/']

    def parse(self, response):
        pass
```

Spiders类

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.
class scrapy.Spider 是最基本的类，所有编写的爬虫必须继承这个类.

爬虫类中的属性和方法

1. name: 这个爬虫的名字，名字必须是唯一的. 例如，如果spider爬取mywebsite.com，该spider通常会被命名为 mywebsite
2. allow_domains: 可选参数，包含了spider允许爬取的域名(domain)的列表. 爬虫只会爬取这个列表中域名下的网页，其他不是这个列表中的域名下的网页会被自动忽略.
3. start_urls: 爬取的URL元祖/列表. 爬虫从这里开始抓取数据，第一次下载的数据将会从这些urls开始. 其他子URL将会从这些起始URL中继承性生成. 可以使用元组也可以使用列表，使用元组时，要在url的后面加上一个逗号，不然就会出错.
4. start_requests(self)
   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（默认实现是使用 start_urls 的url）的第一个Request.
   当spider启动爬取并且未指定start_urls时，该方法被调用.
5. parse: 解析的方法，每个初始URL完成下载后将被调用，调用的时候传入从每一个URL传回的Response对象作为唯一参数. 当请求url返回网页没有指定回调函数时，默认的Request对象回调函数就是parse函数. 主要作用如下:

1) 负责解析返回的网页数据(response.body)，提取结构化数据(生成item)
2) 生成下一个的Request请求对象.

注意: parse方法名一个固定的写法，必须要写为 "parse" 方法，不能写为其它的方法.

6. log(self, message[, level, component])
使用 scrapy.log.msg() 方法记录(log)message.

查看 scrapy.Spider 的源码

"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\spider.py"

```
import warnings
from scrapy.exceptions import ScrapyDeprecationWarning
warnings.warn("Module `scrapy.spider` is deprecated, "
              "use `scrapy.spiders` instead",
              ScrapyDeprecationWarning, stacklevel=2)

from scrapy.spiders import *
```

"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\spiders__init__.py"

主要用到的函数及调用顺序为:

1. __init__(): 初始化爬虫名字和start_urls列表
2. start_requests(): 调用make_requests_from url():生成Requests对象交给Scrapy下载并返回response
3. parse(): 解析response，并返回Item或Requests（需指定回调函数）. Item传给Item pipline持久化，而Requests交由Scrapy下载，并由指定的回调函数处理（默认parse()），一直进行循环，直到处理完所有的数据为止.

```
#所有爬虫的基类，用户定义的爬虫必须从这个类继承
class Spider(object_ref):

    #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.
    #name 是 spider 最重要的属性，而且是必须的.
    #一般做法是以该网站(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
    name = None

    #初始化，提取爬虫名字，start_urls
    def __init__(self, name=None, **kwargs):
        if name is not None:
            self.name = name
```

```
        #  如果爬虫没有名字，中断后续操作则报错
        elif not getattr(self, 'name', None):
            raise ValueError("%s must have a name" % type(self).__name__)

        # python  对象或类型通过内置成员__dict__来存储成员信息
        self.__dict__.update(kwargs)

        #URL 列表.  当没有指定的 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.
        if not hasattr(self, 'start_urls'):
            self.start_urls = []

    #  打印 Scrapy 执行后的 log 信息
    def log(self, message, level=log.DEBUG, **kw):
        log.msg(message, spider=self, level=level, **kw)

    #  判断对象 object 的属性是否存在，不存在做断言处理
    def set_crawler(self, crawler):
        assert not hasattr(self, '_crawler'), "Spider already bounded to %s" % crawler
        self._crawler = crawler

    @property
    def crawler(self):
        assert hasattr(self, '_crawler'), "Spider not bounded to any crawler"
        return self._crawler

    @property
    def settings(self):
        return self.crawler.settings

    #  从 start_urls 中遍历所有的链接，组装并发送 Reuqest 请求对象，默认是使用的 get 请求.
    #该方法将读取 start_urls 内的地址，并为每一个地址生成一个 Request 对象，交给 Scrapy 下载并返回 Response
    #该方法仅调用一次
    def start_requests(self):
        for url in self.start_urls:
            yield self.make_requests_from_url(url)

    #start_requests() 中调用，实际生成 Request 的函数.
    #Request 对象默认的回调函数为 parse()，提交的方式为 get
    def make_requests_from_url(self, url):
        return Request(url, dont_filter=True)

    #默认的 Request 对象回调函数，处理返回的 response.
    #生成 Item 或者 Request 对象. 用户必须实现这个类
    def parse(self, response):
        raise NotImplementedError

    @classmethod
    def handles_request(cls, request):
        return url_is_from_spider(request.url, cls)
```

```
    def __str__(self):
        return "<%s %r at 0x%0x>" % (type(self).__name__, self.name, id(self))

    __repr__ = __str__
```

查看 scrapy.Spider 所有的属性和方法

```
from scrapy import Spider
from scrapy.spiders import Spider

pdir(Spider)

descriptor:
    close: class staticmethod with getter, staticmethod(function) -> method
    from_crawler: class classmethod with getter, classmethod(function) -> method
    handles_request: class classmethod with getter, classmethod(function) -> method
    logger: @property with getter
    update_settings: class classmethod with getter, classmethod(function) -> method
function:
    _set_crawler:
    log: Log the given message at the given log level
    make_requests_from_url: This method is deprecated.
    parse:
    set_crawler:
    start_requests:
```

修改items.py文件，定义要提取的字段

Item 定义结构化数据字段，用来保存爬取到的数据，有点像 Python 中的 dict，但是提供了一些额外的保护减少错误.
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（可以理解成类似于 ORM 的映射关系）.

```
# -*- coding: utf-8 -*-

# Define here the models for your scraped items
#
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/items.html

import scrapy


class QsbkItem(scrapy.Item):
    author = scrapy.Field()
    content = scrapy.Field()
```

再看 scrapy.Field 到底是什么

"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\item.py"

```
class Field(dict):
    """Container of field metadata"""
```

不是别的，就是原原本本的 dict,只不过换了个名字

Filed 的作用是（见官方文档）：

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)。

https://doc.scrapy.org/en/latest/topics/items.html

Item objects
classscrapy.item.Item([arg])
Return a new Item optionally initialized from the given argument.
Items replicate the standard dict API, including its constructor. The only additional attribute provided by Items is:
fields
A dictionary containing all declared fields for this Item, not only those populated. The keys are the field names and the values are the Field objects used in the Item declaration.

Field objects
classscrapy.item.Field([arg])
The Field class is just an alias to the built-in dict class and doesn't provide any extra functionality or attributes. In other words, Field objects are plain-old Python dicts. A separate class is used to support the item declaration syntax based on class attributes.

https://scrapy-chs.readthedocs.io/zh_CN/latest/topics/items.html

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 。

修改qsbk. py，完成的爬虫代码

注意:
在 scrapy 中导入模块或文件的时候都是相对于项目根目录来说的，也就是说是从核心目录开始导入的. 所以要使用 from qsbk.items import QsbkItem 来导入模块

```
# -*- coding: utf-8 -*-
import scrapy
from qsbk.items import QsbkItem
from scrapy.http.response.html import HtmlResponse
from scrapy.selector.unified import SelectorList


class QsbkSpiderSpider(scrapy.Spider):
    name = 'qsbk_spider'
    allowed_domains = ['qiushibaike.com']
    start_urls = ['https://www.qiushibaike.com/text/page/1/']
    base_domain = "https://www.qiushibaike.com"

    def parse(self, response):
        # print(type(response))
        # <class 'scrapy.http.response.html.HtmlResponse'>
        #  查看 HtmlResponse 的源码
        #  调用 response.xpath 方法返回一个 SelectorList 对象
        duanzidivs = response.xpath("//div[@id='content-left']/div")
        # print(type(duanzidivs))
        # <class 'scrapy.selector.unified.SelectorList'>
        for duanzidiv in duanzidivs:
            #  从 SelectorList 中遍历出来的每一个对象都是一个 Selector，可以再次使用 xpath 提取其中的内容，同样得到 SelectorList 对象，使用 get()或者 extract_first()从 SelectorList 中提取出符合规则的第 1 个元素，并转换为 unicode 字符串.
            author = duanzidiv.xpath(".//h2/text()").get().strip()
            # getall()方法等价于 extract，提取出符合规则的所有对象并以列表方式返回，列表中的每一个元素都是一个 unicdoe 字符串.
```

```
            content = duanzidiv.xpath(".//div[@class='content']//text()").getall()
            # 把提取到的内容拼接为字符串
            content = "".join(content).strip()
            item = QsbkItem(author=author, content=content)
            # 如果这里使用 return，就不能提取出下一页的内容了，所以这里要使用 yield
            yield item
        # 提取出下一页的 url 地址
        next_url = response.xpath("//ul[@class='pagination']/li[last()]/a/@href").get()
        if next_url:
            # 如果存在下一页的 url 地址，就构建 Request 对象，并指定响应的处理函数
            yield scrapy.Request(self.base_domain + next_url, callback=self.parse)
        else:
            return
```

思考 parse()方法的工作机制

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.
2. 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；
3. 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；
4. 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)
5. 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）
6. 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；
7. 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.
8. 这一切的一切, Scrapy引擎和调度器将负责到底.

查看Response，HtmlResponse的方法

```
from scrapy.http.response import Response
pdir(Response)

descriptor:
      body: @property with getter, setter
      meta: @property with getter
      text: @property with getter, For subclasses of TextResponse, this will return the
body
```

url: @property with getter, setter
function:
_get_body:
_get_url:
_set_body:
_set_url:
copy: Return a copy of this Response
css: Shortcut method implemented only by responses whose content
follow: Return a:class:`~.Request` instance to follow a link ``url``.
replace: Create a new Response with the same attributes except for those
urljoin: Join this Response's url with a possible relative url to form an
xpath: Shortcut method implemented only by responses whose content

```
from scrapy.http.response.html import HtmlResponse
from scrapy.http.response.text import TextResponse
# HtmlResponse 是继承自 TextResponse, TextResponse 是继承自 Response 的.
HtmlResponse 和 TextResponse 的属性和方法完全一样

pdir(TextResponse)
```

descriptor:
body: @property with getter, setter
encoding: @property with getter
meta: @property with getter
selector: @property with getter
text: @property with getter, Body as unicode
url: @property with getter, setter
function:
_auto_detect_fun:
_body_declared_encoding:
_body_inferred_encoding:
_declared_encoding:
_get_body:
_get_url:
_headers_encoding:
_set_body:
_set_url:
body_as_unicode: Return body as unicode
copy: Return a copy of this Response
css: Shortcut method implemented only by responses whose content
follow: Return a:class:`~.Request` instance to follow a link ``url``.
replace: Create a new Response with the same attributes except for those
urljoin: Join this Response's url with a possible relative url to form an
xpath: Shortcut method implemented only by responses whose content

常用方法的含义

'encoding' 编码方式
'text' 获取响应的文本
'body' 二进制方式的响应. 这里的 text 和 body 与 requests 中的 response.text 和 response.content 类似. text 是经过解码之后的响应, body 是没有经过解码的响应.
'xpath' 可以使用 xpath 提取其中的信息
'css' 可以对响应使用 css 选择器提取信息
scrapy 中可以直接使用 xpath 提取信息, 但如果想要使用 css 选择器, 就要单独导入 beautifulsoup, 使用 bs 把 response.text 解析为 bs 对象, 再从中提取信息.

查看SlectorList的方法

使用 xpath 提取出一个 SlectorList 对象. 是以列表的列式返回所有的 Selector.
from scrapy.selector.unified import SelectorList

pdir(SelectorList)

function:
    __getslice__:
    append: L.append(object) -> None -- append object to end
    clear: L.clear() -> None -- remove all items from L
    copy: L.copy() -> list -- a shallow copy of L
    count: L.count(value) -> integer -- return number of occurrences of value
    css: Call the ``.css()`` method for each element in this list and return
    extend: L.extend(iterable) -> None -- extend list by appending elements from the iterable
    extract: Call the ``.extract()`` method for each element is this list and return
    extract_first: Return the result of ``.extract()`` for the first element in this list.
    extract_unquoted:
    get: Return the result of ``.extract()`` for the first element in this list.
    getall: Call the ``.extract()`` method for each element is this list and return
    index: L.index(value, [start, [stop]]) -> integer -- return first index of value.
    insert: L.insert(index, object) -- insert object before index
    pop: L.pop([index]) -> item -- remove and return item at index (default last).
    re: Call the ``.re()`` method for each element in this list and return
    re_first: Call the ``.re()`` method for the first element in this list and
    remove: L.remove(value) -> None -- remove first occurrence of value.
    reverse: L.reverse() -- reverse *IN PLACE*
    select:
    sort: L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
    x:
    xpath: Call the ``.xpath()`` method for each element in this list and return

查看SlectorList的源码

其中的 deprecated 表示方法已经被抛弃，使用后面定义的方法来代替.

```
class SelectorList(_ParselSelector.selectorlist_cls, object_ref):
    @deprecated(use_instead='.extract()')
    def extract_unquoted(self):
        return [x.extract_unquoted() for x in self]

    @deprecated(use_instead='.xpath()')
    def x(self, xpath):
        return self.select(xpath)

    @deprecated(use_instead='.xpath()')
    def select(self, xpath):
        return self.xpath(xpath)
```

查看Selector的方法

其中的 SelectorList 中装的是所有的 Selector,

from scrapy.selector.unified import Selector

pdir(Selector)
property:
    _default_namespaces, _default_type, _lxml_smart_strings
descriptor:
    _csstranslator: class member_descriptor with getter, setter, deleter
    _expr: class member_descriptor with getter, setter, deleter
    _parser: class member_descriptor with getter, setter, deleter
    _root: @property with getter
    _tostring_method: class member_descriptor with getter, setter, deleter
    namespaces: class member_descriptor with getter, setter, deleter
    response: class member_descriptor with getter, setter, deleter
    root: class member_descriptor with getter, setter, deleter
    text: class member_descriptor with getter, setter, deleter
    type: class member_descriptor with getter, setter, deleter
class:
    selectorlist_cls: The:class:`SelectorList` class is a subclass of the builtin ``list``
function:
    __nonzero__: Return ``True`` if there is any real content selected or ``False``
    _css2xpath:
    _get_root:
    css: Apply the given CSS selector and return a:class:`SelectorList` instance.
    extract: Serialize and return the matched nodes in a single unicode string.
    extract_unquoted:
    get: Serialize and return the matched nodes in a single unicode string.

getall: Serialize and return the matched node in a 1-element list of unicode strings.
re: Apply the given regex and return a list of unicode strings with the
re_first: Apply the given regex and return the first unicode string which
register_namespace: Register the given namespace to be used in this:class:`Selector`.
remove_namespaces: Remove all namespaces, allowing to traverse the document using
select:
xpath: Find nodes matching the xpath ``query`` and return the result as a

from parsel import Selector 这个中的 Selector 是与 SelectorList 中的 Selector 是什么关系呢?
dir(parsel.Selector)

Scrapy Selectors 内置 XPath 和 CSS Selector 表达式机制

Selector 有四个基本的方法，最常用的还是 xpath:

- 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
- extract(): 序列化该节点为 Unicode 字符串并返回 list
- 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，语法同 BeautifulSoup4
- 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

使用 xpath 获取到的元素类型是 SelectorList，可以遍历后再次使用 xpath 来进行进一步的选择．也可以使用 css, re 来提取其中的信息．
extract 方法等价于 getall，把 SelectorList 转换为 list，其中的元素是 Unicode 格式的字符串．
extract_first 等价于 get，从 selector 中提取第 1 条符合规则的数据，返回 Unicode 格式的字符串．

修改pipelines.py，把数据保存到json文件中.

scrapy 下载到数据后，会由 engine 交给 spider 进行处理，如果 spider 得到的是数据，会把数据交给 item pipline 去处理．如果是请求，再经 scrapy engine 交给 scheduler 调度器，再次下载.

使用 yield 生成数据，一次拿到提取的一个数据，把一个数据经 yield 交给 pipeline 管道文件进行处理，处理完一个数据后再接着处理第二个数据，依此进行下去.

当 Item 在 Spider 中被收集之后，它将会被传递到 Item Pipeline，这些 Item Pipeline 组件按定义的顺序处理 Item.
每个 Item Pipeline 都是实现了简单方法的 Python 类，比如决定此 Item 是丢弃而存储．以下是 item pipeline 的一些典型应用:

- 验证爬取的数据(检查 item 包含某些字段，比如说 name 字段)
- 查重(并丢弃)
- 将爬取结果保存到文件或者数据库中

https://docs.scrapy.org/en/latest/topics/item-pipeline.html

每个管道组件都是一个实现了某个功能的 Python 类，常见功能有：
清理 html 数据
做确认
查重
存入数据库

每个管道组件的类，必须要有以下方法：
process_item(self, item, spider)
open_spider(self, spider)
close_spider(self, spider)
from_crawler(cls, crawler)

```
# 丢弃数据项
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)


# 存储到MongoDB

import pymongo

class MongoPipeline(object):

    collection_name = 'scrapy_items'

    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):
        self.db[self.collection_name].insert_one(dict(item))
        return item


# 存储到MySQL
class MysqlPipeline():
    def __init__(self, host, database, user, password, port):
        self.host = host
        self.database = database
        self.user = user
        self.password = password
        self.port = port

    @classmethod
    def from_crawler(cls, crawler):
        return cls(
            host=crawler.settings.get('MYSQL_HOST'),
            database=crawler.settings.get('MYSQL_DATABASE'),
            user=crawler.settings.get('MYSQL_USER'),
            password=crawler.settings.get('MYSQL_PASSWORD'),
            port=crawler.settings.get('MYSQL_PORT'),
        )

    def open_spider(self, spider):
        self.db = pymysql.connect(self.host, self.user, self.password, self.database,
charset='utf8',
                                  port=self.port)
        self.cursor = self.db.cursor()

    def close_spider(self, spider):
```

```
            self.db.close()

        def process_item(self, item, spider):
            print(item['title'])
            data = dict(item)
            keys = ', '.join(data.keys())
            values = ', '.join(['%s'] * len(data))
            sql = 'insert into %s (%s) values (%s)' % (item.table, keys, values)
            self.cursor.execute(sql, tuple(data.values()))
            self.db.commit()
            return item

# 去重
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

# 激活管道
ITEM_PIPELINES = {
        'myproject.pipelines.PricePipeline': 300,
        'myproject.pipelines.JsonWriterPipeline': 800,
}
```

item pipiline 组件是一个独立的 Python 类，其中 process_item()方法必须实现

```
import something

class SomethingPipeline(object):
        def __init__(self):
            # 可选实现，做参数初始化等
            # doing something

        # process_item 方法是必须写的，用来处理 item 数据
        def process_item(self, item, spider):
            # item (Item 对象) – 被爬取的 item
```

```
        # spider (Spider 对象) – 爬取该 item 的 spider
        # 这个方法必须实现，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 对象，否则被丢弃的 item 将不会被之后的 pipeline 组件所处理. 如果不 returen item， 下一次处理的时候是同一个 item
        return item

    def open_spider(self, spider):
        # spider (Spider 对象) – 被开启的 spider
        # 可选实现，当 spider 被开启时，这个方法被调用.

    def close_spider(self, spider):
        # spider (Spider 对象) – 被关闭的 spider
        # 可选实现，当 spider 被关闭时，这个方法被调用
```

方法1 使用dumps把python对象转换为json再写入到文件中

```
import json

class QsbkPipeline(object):
    def __init__(self):
        # 要把数据写入到 json 文件中，可以在 open_spider 方法中打开文件，也可以在构造函数中打开文件.
        self.fp = open("duanzi.json",'w',encoding='utf-8')

    # 爬虫开始运行时就会调用 open_spider
    def open_spider(self,spider):
        print('爬虫开始了...')

    # 数据从 spider 中传递到 pipelines 中后会经过 process_item 这个方法的处理.
    def process_item(self, item, spider):
        # item 为 scrpy.item 格式的数据，要先把 item 转换为字典，再使用 dumps 转换为 json
        item_json = json.dumps(dict(item),ensure_ascii=False)
        # 写入到文件中
        self.fp.write(item_json+'\n')
        return item

    # 爬虫关闭时会调用 close_spider 这个函数
    def close_spider(self,spider):
        # 关闭文件
        self.fp.close()
        print('爬虫结束了...')
```

方法2, 使用JsonItemExporter保存json数据

查看 scrapy.exporters 中所有的可导出的数据类型

```
from scrapy import exporters

pdir(exporters)

class:
    BaseItem: Base class for all scraped items.
    BaseItemExporter:
    CsvItemExporter:
    JsonItemExporter:
    JsonLinesItemExporter:
    MarshalItemExporter:
    PickleItemExporter:
    PprintItemExporter:
    PythonItemExporter: The idea behind this exporter is to have a mechanism to serialize items
    ScrapyJSONEncoder: Extensible JSON <http://json.org> encoder for Python data structures.
    XMLGenerator: Interface for receiving logical document content events.
    XmlItemExporter:
exception:
    ScrapyDeprecationWarning: Warning category for deprecated features, since the default
function:
    is_listlike: >>> is_listlike("foo")
    to_bytes: Return the binary representation of `text`. If `text`
    to_native_str: Return str representation of `text`
    to_unicode: Return the unicode representation of a bytes object `text`. If `text`
```

使用 JsonItemExporter 和 JsonLinesItemExporter 来保存 json 数据

JsonItemExporter 和 JsonLinesItemExporter 都是继承自 BaseItemExporter.

jsonItemExporter 是等待所有数据都爬取完成后再写入到文件中，在数据量比较大时对内存的需求比较高. 并且如果爬虫在运行过程中出错，数据会全部丢失.
JsonLinesItemExporter 是逐行写入的，对内存的要求比较低，爬虫出错时原来已经提取的数据已经保存到硬盘中.
JsonItemExporter 是整体以 json 格式写入的，而 JsonLinesItemExporter 则是逐行写入的，每一行都是一个字典，整体上不是一个 json 格式. 可以通过重写在第一行开始时写入'[，每一行结束时写入，文件结尾处写入]'. 或者在文件写入完成后使用另一个函数对其进行处理，转换为 json 格式.

查看 BaseItemExporter 的属性/方法和源码

```
from scrapy.exporters import BaseItemExporter

pdir(BaseItemExporter)

function:
    _configure: Configure the exporter by poping options from the ``options`` dict.
    _get_serialized_fields: Return the fields to export as an iterable of tuples
```

```
    export_item:
    finish_exporting:
    serialize_field:
    start_exporting:
```

```
class BaseItemExporter(object):

    def __init__(self, **kwargs):
        self._configure(kwargs)

    def _configure(self, options, dont_fail=False):
        """Configure the exporter by poping options from the ``options`` dict.
        If dont_fail is set, it won't raise an exception on unexpected options
        (useful for using with keyword arguments in subclasses constructors)
        """
        self.encoding = options.pop('encoding', None)
        self.fields_to_export = options.pop('fields_to_export', None)
        self.export_empty_fields = options.pop('export_empty_fields', False)
        self.indent = options.pop('indent', None)
        if not dont_fail and options:
            raise TypeError("Unexpected options: %s" % ', '.join(options.keys()))

    def export_item(self, item):
        raise NotImplementedError

    def serialize_field(self, field, name, value):
        serializer = field.get('serializer', lambda x: x)
        return serializer(value)

    def start_exporting(self):
        pass

    def finish_exporting(self):
        pass

    def _get_serialized_fields(self, item, default_value=None, include_empty=None):
        """Return the fields to export as an iterable of tuples
        (name, serialized_value)
        """
        if include_empty is None:
            include_empty = self.export_empty_fields
        if self.fields_to_export is None:
            if include_empty and not isinstance(item, dict):
                field_iter = six.iterkeys(item.fields)
            else:
                field_iter = six.iterkeys(item)
        else:
            if include_empty:
                field_iter = self.fields_to_export
            else:
                field_iter = (x for x in self.fields_to_export if x in item)

        for field_name in field_iter:
```

```
            if field_name in item:
                field = {} if isinstance(item, dict) else item.fields[field_name]
                value = self.serialize_field(field, field_name, item[field_name])
            else:
                value = default_value

            yield field_name, value
```

查看 JsonItemExporter 的属性/方法和源码

```
from scrapy.exporters import JsonItemExporter

pdir(JsonItemExporter)
function:
    _beautify_newline:
    _configure: Configure the exporter by poping options from the ``options`` dict.
    _get_serialized_fields: Return the fields to export as an iterable of tuples
    export_item:
    finish_exporting:
    serialize_field:
    start_exporting:
```

```
pdir(JsonItemExporter("file.json"))
property:
    encoder, encoding, export_empty_fields, fields_to_export, file, first_item, indent
function:
    _beautify_newline:
    _configure: Configure the exporter by poping options from the ``options`` dict.
    _get_serialized_fields: Return the fields to export as an iterable of tuples
    export_item:
    finish_exporting:
    serialize_field:
    start_exporting:
```

```
class JsonItemExporter(BaseItemExporter):

    def __init__(self, file, **kwargs):
        self._configure(kwargs, dont_fail=True)
        self.file = file
        # there is a small difference between the behaviour or JsonItemExporter.indent
        # and ScrapyJSONEncoder.indent. ScrapyJSONEncoder.indent=None is needed to prevent
        # the addition of newlines everywhere
        json_indent = self.indent if self.indent is not None and self.indent > 0 else None
        kwargs.setdefault('indent', json_indent)
        kwargs.setdefault('ensure_ascii', not self.encoding)
        self.encoder = ScrapyJSONEncoder(**kwargs)
        self.first_item = True

    def _beautify_newline(self):
        if self.indent is not None:
```

```
            self.file.write(b'\n')

    def start_exporting(self):
        self.file.write(b"[")
        self._beautify_newline()

    def finish_exporting(self):
        self._beautify_newline()
        self.file.write(b"]")

    def export_item(self, item):
        if self.first_item:
            self.first_item = False
        else:
            self.file.write(b',')
            self._beautify_newline()
        itemdict = dict(self._get_serialized_fields(item))
        data = self.encoder.encode(itemdict)
        self.file.write(to_bytes(data, self.encoding))
```

使用 JsonItemExporter 导出 json 数据

```
from scrapy.exporters import JsonItemExporter

class QsbkJsonItemPipeline(object):

    def __init__(self):
        # JsonItemExporter 是以 bytes 方式写入数据的, 所以要使用二进制的方式打开文件.
注意使用二进制方式打开文件时不能指定编码
        self.fp = open("duanzi_json_item.json",'wb')
        # 定义一个 json exporter 对象
        self.exporter = JsonItemExporter(self.fp,ensure_ascii=False,encoding='utf-8')
        # 开始执行
        self.exporter.start_exporting()

    def open_spider(self,spider):
        print('爬虫开始了...')

    def process_item(self, item, spider):
        self.exporter.export_item(item)
        # 注意在 process_item 处理完成之后必须要 return item, 以供其它的 pipline 使用.
        return item

    def close_spider(self,spider):
        # 完成导入
        self.exporter.finish_exporting()
        self.fp.close()
        print('爬虫结束了...')
```

方法3，使用JsonLinesItemExporter来保存json数据

查看 JsonLinesItemExporter 的属性/方法和源码

```
from scrapy.exporters import JsonLinesItemExporter

pdir(JsonLinesItemExporter)

function:
    _configure: Configure the exporter by poping options from the ``options`` dict.
    _get_serialized_fields: Return the fields to export as an iterable of tuples
    export_item:
    finish_exporting:
    serialize_field:
    start_exporting:
```

```
pdir(JsonLinesItemExporter("file.json"))

property:
    encoder, encoding, export_empty_fields, fields_to_export, file, indent
function:
    _configure: Configure the exporter by poping options from the ``options`` dict.
    _get_serialized_fields: Return the fields to export as an iterable of tuples
    export_item:
    finish_exporting:
    serialize_field:
    start_exporting:
```

```
class JsonLinesItemExporter(BaseItemExporter):

    def __init__(self, file, **kwargs):
        self._configure(kwargs, dont_fail=True)
        self.file = file
        kwargs.setdefault('ensure_ascii', not self.encoding)
        self.encoder = ScrapyJSONEncoder(**kwargs)

    def export_item(self, item):
        itemdict = dict(self._get_serialized_fields(item))
        data = self.encoder.encode(itemdict) + '\n'
        self.file.write(to_bytes(data, self.encoding))
```

使用 JsonLinesItemExporter 导出 json 数据

```
from scrapy.exporters import JsonLinesItemExporter


class QsbkJsonLinesItemPipeline(object):
    def __init__(self):
        self.fp = open("duanzi_json_lines_item.json",'wb')
        self.exporter = JsonLinesItemExporter(self.fp,ensure_ascii=False,encoding='utf-8')
```

```
    def open_spider(self,spider):
        print('爬虫开始了...')

    def process_item(self, item, spider):
        self.exporter.export_item(item)
        # 注意在 process_item 处理完成之后必须要 return item，以供其它的 pipline 使用.
        return item

    def close_spider(self,spider):
        self.fp.close()
        print('爬虫结束了...')
```

修改settings.py，开启item_pipelines

在 settigns.py 中开启 item_pipelines，同时进行必要的设置

```
ROBOTSTXT_OBEY = False

DOWNLOAD_DELAY = 1

DEFAULT_REQUEST_HEADERS = {
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
   'Accept-Language': 'en',
   'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'
}

ITEM_PIPELINES = {
    'qsbk.pipelines.QsbkPipeline': 100,
    'qsbk.pipelines.QsbkJsonItemPipeline': 200,
    'qsbk.pipelines.QsbkJsonLinesItemPipeline': 300,
}
```

运行scrapy项目

在终端进入项目根目录，使用以下命令来运行scrapy爬虫项目

scrapy crawl qsbk

如果不想每次都在命令行中运行，那么可以把这个命令写在一个文件中．以后就在pycharm中执行运行这个文件就可以了．在项目根目录中新建 start.py

```
#encoding: utf-8

from scrapy import cmdline

# cmdline.execute("scrapy crawl qsbk_spider".split())
```

```
cmdline.execute(['scrapy', 'crawl', 'qsbk_spider'])
```

这样，只需要执行start.py就可以运行爬虫了.

糗事百科Scrapy爬虫笔记

1. response 是一个`scrapy.http.response.html.HtmlResponse`对象. 可以执行`xpath`和`css`语法来提取数据.
2. 提取出来的数据，是一个`Selector`或者是一个`SelectorList`对象. 如果想要获取其中的字符串. 那么应该执行`getall`或者`get`方法.
3. getall 方法: 获取`Selector`中的所有文本. 返回的是一个列表.
4. get 方法: 获取的是`Selector`中的第一个文本. 返回的是一个 str 类型.
5. 如果数据解析回来，要传给 pipline 处理. 那么可以使用`yield`来返回. 或者是收集所有的 item. 最后统一使用 return 返回.
6. item: 建议在`items.py`中定义好模型. 以后就不要使用字典.
7. pipeline: 这个是专门用来保存数据的. 其中有三个方法是会经常用的.
    * `open_spider(self,spider)`: 当爬虫被打开的时候执行.
    * `process_item(self,item,spider)`: 当爬虫有 item 传过来的时候会被调用.
    * `close_spider(self,spider)`: 当爬虫关闭的时候会被调用.
8. 要激活 piplilne，应该在`settings.py`中，设置`ITEM_PIPELINES`. 示例如下
    ```python
    ITEM_PIPELINES = {
        'qsbk.pipelines.QsbkPipeline': 300,
    }
    ```

JsonItemExporter和JsonLinesItemExporter总结

保存 json 数据的时候，可以使用这两个类，让操作变得得更简单.

1. `JsonItemExporter`: 在调用 start_exporting 方法时开始把每个传递过来的 item 数据添加到内存中. 最后执行 finish_exporting 时统一写入到磁盘中. 好处是，存储的数据是一个满足 json 规则的数据. 坏处是如果数据量比较大，那么比较耗内存. 得到的文件是一个列表中存储着多个字典，每个字典中都是一个 item. 示例代码如下

```python
from scrapy.exporters import JsonItemExporter

class QsbkPipeline(object):
    def __init__(self):
        self.fp = open("duanzi.json",'wb')
        self.exporter = JsonItemExporter(self.fp,ensure_ascii=False,encoding='utf-8')
        self.exporter.start_exporting()

    def open_spider(self,spider):
```

```
            print('爬虫开始了...')

        def process_item(self, item, spider):
            self.exporter.export_item(item)
            return item

        def close_spider(self,spider):
            self.exporter.finish_exporting()
            self.fp.close()
            print('爬虫结束了...')
```

2. `JsonLinesItemExporter`：这个是每次调用`export_item`的时候就把这个 item 存储到硬盘中．坏处是每一个字典是一行，整个文件不是一个满足 json 格式的文件．好处是每次处理数据的时候就直接存储到了硬盘中，这样不会耗内存，数据也比较安全．得到的文件是每个字典是一行，在读取时就不能把整个文件 load 成一个 json 字符串了，而要每次读取一行的数据．示例代码如下

```python
from scrapy.exporters import JsonLinesItemExporter
class QsbkPipeline(object):
        def __init__(self):
            self.fp = open("duanzi.json",'wb')
            self.exporter = JsonLinesItemExporter(self.fp,ensure_ascii=False,encoding='utf-8')

        def open_spider(self,spider):
            print('爬虫开始了...')

        def process_item(self, item, spider):
            self.exporter.export_item(item)
            return item

        def close_spider(self,spider):
            self.fp.close()
            print('爬虫结束了...')
```

Logging 日志文件

Scrapy 提供了 log 功能，可以通过 logging 模块使用．
可以修改配置文件 settings.py，任意位置添加下面两行，效果会清爽很多．
LOG_FILE = "TencentSpider.log"
LOG_LEVEL = "INFO"

Log levels

Scrapy 提供 5 层 logging 级别:

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

logging设置

通过在setting.py中进行以下设置可以被用来配置logging:

1. LOG_ENABLED 默认: True, 启用logging
2. LOG_ENCODING 默认: 'utf-8', logging使用的编码
3. LOG_FILE 默认: None, 在当前目录里创建logging输出文件的文件名
4. LOG_LEVEL 默认: 'DEBUG', log的最低级别
5. 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.

5.3. CrawlSpider 类爬虫

在上一个糗事百科的爬虫案例中. 我们是自己在解析完整个页面后获取下一页的 url, 然后重新发送一个请求. 有时候我们想要这样做, 只要满足某个条件的 url, 都要进行爬取. 那么这时候就可以使用 CrawlSpider 类的爬虫.
CrawlSpider 继承自 Spider, Spider 类的设计原则是只爬取 start_url 列表中的网页, 而 CrawlSpider 类定义了一些规则(rule)来提供跟进 link 的方便的机制, 从爬取的网页中获取 link 并继续对提取到的链接进行爬取, 而不用手动构建并 yield Request 请求对象.

创建CrawlSpider爬虫

之前创建爬虫的方式是通过 scrapy genspider [爬虫名字] [域名]的方式创建的. 如果想要创建 CrawlSpider 爬虫, 那么应该通过以下命令创建

scrapy genspider -t crawl [爬虫名字] [域名]

微信小程序社区CrawlSpider案例

创建项目

scrapy startproject wxapp_pro

创建爬虫

cd wxapp_pro
scrapy genspider -t crawl wxapp "wxapp-union.com"

查看wxapp.py

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule


class WxappSpider(CrawlSpider):
    name = 'wxapp'
    allowed_domains = ['wxapp-union.com']
    start_urls = ['http://wxapp-union.com/']

    rules = (
        Rule(LinkExtractor(allow=r'Items/'), callback='parse_item', follow=True),
    )

    def parse_item(self, response):
        i = {}
        #i['domain_id'] = response.xpath('//input[@id="sid"]/@value').extract()
        #i['name'] = response.xpath('//div[@id="name"]').extract()
        #i['description'] = response.xpath('//div[@id="description"]').extract()
        return i
```

查看CrwalSpider爬虫的源码

"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\spiders\crawl.py"

```
class CrawlSpider(Spider):

    rules = ()

    def __init__(self, *a, **kw):
        super(CrawlSpider, self).__init__(*a, **kw)
        self._compile_rules()

    #  首先调用 parse() 来处理 start_urls 中返回的 response 对象
    # parse() 则将这些 response 对象传递给了 _parse_response() 函数处理，并设置回调函数为 parse_start_url()
    #  设置了跟进标志位 True
```

```
    # parse 将返回 item 和跟进了的 Request 对象
    def parse(self, response):
        return self._parse_response(response, self.parse_start_url, cb_kwargs={}, follow=True)

    # 处理 start_url 中返回的 response，需要重写
    def parse_start_url(self, response):
        return []

    def process_results(self, response, results):
        return results

    # 从 response 中抽取符合任一用户定义'规则'的链接，并构造成 Resquest 对象返回
    def _requests_to_follow(self, response):
        if not isinstance(response, HtmlResponse):
            return
        seen = set()
        # 抽取之内的所有链接，只要通过任意一个'规则'，即表示合法
        for n, rule in enumerate(self._rules):
            links = [l for l in rule.link_extractor.extract_links(response) if l not in seen]
            # 使用用户指定的 process_links 处理每个连接
            if links and rule.process_links:
                links = rule.process_links(links)
            # 将链接加入 seen 集合，为每个链接生成 Request 对象，并设置回调函数为
_repsonse_downloaded()
            for link in links:
                seen.add(link)
                # 构造 Request 对象，并将 Rule 规则中定义的回调函数作为这个 Request 对
象的回调函数
                r = Request(url=link.url, callback=self._response_downloaded)
                r.meta.update(rule=n, link_text=link.text)
                # 对每个 Request 调用 process_request()函数。该函数默认为 indentify，即不
做任何处理，直接返回该 Request.
                yield rule.process_request(r)

    # 处理通过 rule 提取出的连接，并返回 item 以及 request
    def _response_downloaded(self, response):
        rule = self._rules[response.meta['rule']]
        return self._parse_response(response, rule.callback, rule.cb_kwargs, rule.follow)

    # 解析 response 对象，会用 callback 解析处理他，并返回 request 或 Item 对象
    def _parse_response(self, response, callback, cb_kwargs, follow=True):
        # 首先判断是否设置了回调函数。（该回调函数可能是 rule 中的解析函数，也可能是
parse_start_url 函数）
        # 如果设置了回调函数（parse_start_url()），那么首先用 parse_start_url()处理
response 对象，
        # 然后再交给 process_results 处理。返回 cb_res 的一个列表
        if callback:
            # 如果是 parse 调用的，则会解析成 Request 对象
            # 如果是 rule callback，则会解析成 Item
            cb_res = callback(response, **cb_kwargs) or ()
            cb_res = self.process_results(response, cb_res)
```

```
                for requests_or_item in iterate_spider_output(cb_res):
                    yield requests_or_item

            # 如果需要跟进，那么使用定义的Rule 规则提取并返回这些Request 对象
            if follow and self._follow_links:
                # 返回每个Request 对象
                for request_or_item in self._requests_to_follow(response):
                    yield request_or_item

    def _compile_rules(self):
        def get_method(method):
            if callable(method):
                return method
            elif isinstance(method, basestring):
                return getattr(self, method, None)

        self._rules = [copy.copy(r) for r in self.rules]
        for rule in self._rules:
            rule.callback = get_method(rule.callback)
            rule.process_links = get_method(rule.process_links)
            rule.process_request = get_method(rule.process_request)

    def set_crawler(self, crawler):
        super(CrawlSpider, self).set_crawler(crawler)
        self._follow_links = crawler.settings.getbool('CRAWLSPIDER_FOLLOW_LINKS', True)
```

Rule规则类

```
rules = (
    Rule(LinkExtractor(allow=r'Items/'), callback='parse_item', follow=True),
)
```

CrawlSpider 使用 rules 来决定爬虫的爬取规则，并将匹配后的 url 请求提交给引擎. 所以在正常情况下, CrawlSpider 不需要单独手动返回请求了.
在 rules 中包含一个或多个 Rule 对象，可以是正则匹配规则对象，每个 Rule 对爬取网站的动作定义了特定操作. 如果多个 rule 匹配了相同的链接，则根据规则在本集合中被定义的顺序，第一个会被使用.

定义爬虫的规则类. 以下对这个类做一个简单的介绍

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

主要参数讲解

1. link_extractor: 一个 LinkExtractor 对象，用于定义爬取规则. 如 page_lx = LinkExtractor(allow=('position.php?&start=\d+'))
2. callback: 满足这个规则的 url，应该要执行哪个回调函数. 因为 CrawlSpider 使用了 parse 作为默认的回调函数，因此不要重写 parse 作为自己的回调函数，而要定义其它名称的解析函数作为回调函数.
3. follow: 指定根据该规则从 response 中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(??). 深度爬虫.
4. process_links: 指定该 spider 中哪个的函数将会被调用，从 link_extractor 中获取到链接后会传递给这个函数，用来过滤不需要爬取的链接.
5. process_request: 指定该 spider 中哪个的函数将会被调用，该规则提取到每个 request 时都会调用该函数. (用来过滤 request)

查看 Rule 规则类的源码

"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\spiders\crawl.py"

```
class Rule(object):

    def __init__(self, link_extractor, callback=None, cb_kwargs=None, follow=None,
process_links=None, process_request=identity):
        self.link_extractor = link_extractor
        self.callback = callback
        self.cb_kwargs = cb_kwargs or {}
        self.process_links = process_links
        self.process_request = process_request
        if follow is None:
            self.follow = False if callback else True
        else:
            self.follow = follow
```

查看 Rule 规则类的方法/属性

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

rules = (
    Rule(LinkExtractor(allow=r'Items/'), callback='parse_item', follow=True),
)
pdir(rules)

function:
    count: T.count(value) -> integer -- return number of occurrences of value
    index: T.index(value, [start, [stop]]) -> integer -- return first index of value.
```

LinkExtractors链接提取器

Link Extractors 的作用：提取链接。
每个 LinkExtractor 有唯一的公共方法是 extract_links()，它接收一个 Response 对象，并返回一个 scrapy.link.Link 对象.
Link Extractors 要实例化一次，其中的 extract_links 方法会在响应的页面中找到所有满足规则的 url 地址进行提取，自动生成并发送 Request 请求对象.

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

主要参数讲解

1. allow：允许的 url. 所有满足这个[列表中的]正则表达式的 url 都会被提取. 如果为空，则全部匹配.
2. deny：禁止的 url. 所有满足这个[列表中的]正则表达式的 url 都不会被提取.
3. allow_domains：允许的域名. 只有在这个里面指定的域名的 url 才会被提取.
4. deny_domains：禁止的域名. 所有在这个里面指定的域名的 url 都不会被提取.
5. restrict_xpaths：严格的 xpath. 和 allow 共同过滤链接.

查看 LinkExtractor 的源码

```
"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\linkextractors\lxmlhtml.py"

from scrapy.linkextractors import LinkExtractor
from .lxmlhtml import LxmlLinkExtractor as LinkExtractor


class LxmlLinkExtractor(FilteringLinkExtractor):

    def __init__(self, allow=(), deny=(), allow_domains=(), deny_domains=(), restrict_xpaths=(),
                 tags=('a', 'area'), attrs=('href',), canonicalize=False,
                 unique=True, process_value=None, deny_extensions=None, restrict_css=(),
                 strip=True):
        tags, attrs = set(arg_to_iter(tags)), set(arg_to_iter(attrs))
        tag_func = lambda x: x in tags
        attr_func = lambda x: x in attrs
        lx = LxmlParserLinkExtractor(
```

```
            tag=tag_func,
            attr=attr_func,
            unique=unique,
            process=process_value,
            strip=strip,
            canonicalized=canonicalize
        )

        super(LxmlLinkExtractor, self).__init__(lx, allow=allow, deny=deny,
            allow_domains=allow_domains, deny_domains=deny_domains,
            restrict_xpaths=restrict_xpaths, restrict_css=restrict_css,
            canonicalize=canonicalize, deny_extensions=deny_extensions)
```

pdir(Rule(LinkExtractor()))

property:
    callback, cb_kwargs, follow, link_extractor, process_links
function:
    process_request:

pdir(LinkExtractor)

function:
    _extract_links:
    _link_allowed:
    _process_links:
    extract_links:
    matches:

Rule和LinkExtractor总结

需要使用`LinkExtractor`和`Rule`. 这两个东西决定爬虫的具体走向.
1. allow 设置规则的方法: 能够只提取出我们想要的 url 地址, 注意正则表达式的提取规则要唯一, 不要提取到其它模式的 url 地址.
要能够限制在我们想要的 url 上面. 不要跟其他的 url 产生相同的正则表达式即可.
2. 什么情况下使用 follow: 如果在爬取页面的时候, 需要将满足当前条件的 url 再进行跟进, 那么就设置为 True. 否则设置为 Fasle.
3. 什么情况下该指定 callback: 如果这个 url 对应的页面, 只是为了获取更多的 url, 并不需要里面的数据, 那么可以不指定 callback. 如果想要获取 url 对应页面中的数据, 那么就需要指定一个 callback.

定义items.py

```
# -*- coding: utf-8 -*-

import scrapy
```

```
class WxappItem(scrapy.Item):
    title = scrapy.Field()
    author = scrapy.Field()
    pub_time = scrapy.Field()
    content = scrapy.Field()
```

修改wxapp.py， 定义spider

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule
from wxapp.items import WxappItem
from scrapy.http.response.html import HtmlResponse


class WxappSpiderSpider(CrawlSpider):
    name = 'wxapp_spider'
    allowed_domains = ['wxapp-union.com']
    start_urls = ['http://www.wxapp-union.com/portal.php?mod=list&catid=2&page=1']

    rules = (
        # 匹配列表页 url，不指定 callback 时默认是使用 parse 函数来解析响应. 需要从提取到的链接的响应中提取内容时才需要手动指定 callback.
        Rule(LinkExtractor(allow=r'.+mod=list&catid=2&page=\d'), follow=True),
        # 匹配详情页 url. 因为只需要提取列表页中的文章详情 url 地址，所以在文章详情页中不需要提取详情页的 url，所以这里的 follow=False
        Rule(LinkExtractor(allow=r".+article-.+\.html"),callback="parse_detail",follow=False)
    )

    # 解析详情页，从中提取信息
    def parse_detail(self, response):
        title = response.xpath("//h1[@class='ph']/text()").get()
        author_p = response.xpath("//p[@class='authors']")
        author = author_p.xpath(".//a/text()").get()
        pub_time = author_p.xpath(".//span/text()").get()
        article_content = response.xpath("//td[@id='article_content']//text()").getall()
        content = "".join(article_content).strip()
        item = WxappItem(title=title,author=author,pub_time=pub_time,content=content)
        yield item
```

修改pipelines.py， 把数据保存到json文件中

Version1 使用默认的Pipeline

```
# -*- coding: utf-8 -*-
```

```
# Define your item pipelines here
#
# Don't forget to add your pipeline to the ITEM_PIPELINES setting
# See: http://doc.scrapy.org/en/latest/topics/item-pipeline.html
import json

class WxappSpiderPipeline(object):
    def __init__(self):
        self.items = []

    def process_item(self, item, spider):
        self.items.append(dict(item))
        return item

    def close_spider(self,spider):
        with open('wxapp.json','w',encoding='utf-8') as fp:
            json.dump(self.items,fp,ensure_ascii=False)
```

Version2 使用JsonLinesItemExporter

```
# -*- coding: utf-8 -*-

from scrapy.exporters import JsonLinesItemExporter

class WxappPipeline(object):
    def __init__(self):
        self.fp = open('wxjc.json','wb')
        self.exporter = JsonLinesItemExporter(self.fp,ensure_ascii=False,encoding='utf-8')

    def process_item(self, item, spider):
        self.exporter.export_item(item)
        return item

    def close_spider(self,spider):
        self.fp.close()
```

修改settings.py

```
ROBOTSTXT_OBEY = False

DOWNLOAD_DELAY = 1

DEFAULT_REQUEST_HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'Accept-Language': 'en',
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'
}

ITEM_PIPELINES = {
```

```
    'wxapp.pipelines.WxappPipeline': 300,
}
```

在项目根目录中新建start.py

```
#encoding: utf-8

from scrapy import cmdline

cmdline.execute("scrapy crawl wxapp_spider".split())
```

运行start.py，进行测试

5.4. ScrapyShell

官方文档: http://scrapy-chs.readthedocs.io/zh_CN/latest/topics/shell.html

我们想要在爬虫中使用xpath, beautifulsoup，正则表达式, css选择器等来提取想要的数据．但是因为scrapy是一个比较重的框架．每次运行起来都要等待一段时间．因此要去验证我们写的提取规则是否正确，是一个比较麻烦的事情．因此Scrapy提供了一个shell，用来方便的测试规则．当然也不仅仅局限于这一个功能．

打开Scrapy Shell

在终端中进入scrapy shell，不要进入到python的交互界面中．打开cmd终端，进入到Scrapy项目所在的目录，然后进入到scrapy框架所在的虚拟环境中，输入命令scrapy shell [链接]．就会进入到scrapy的shell环境中．在这个环境中，你可以跟在爬虫的parse方法中一样使用了．

scrapy shell可以在任意位置打开，但是如果想要读取某个scrapy 项目的配置文件时，必须要进入到项目的根目录中打开scrapy shell

```
scrapy shell http://www.wxapp-union.com/article-4180-1.html

2018-07-11 15:19:37 [scrapy.utils.log] INFO: Scrapy 1.5.0 started (bot: wxapp)
2018-07-11 15:19:37 [scrapy.utils.log] INFO: Versions: lxml 4.2.1.0, libxml2 2.9.5,
cssselect 1.0.3, parsel 1.4.0, w3lib 1.19.0, Twisted 17.9.0, Python 3.6.5
(v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)],
pyOpenSSL 17.5.0 (OpenSSL 1.1.0h  27 Mar 2018), cryptography 2.2.2, Platform
Windows-10-10.0.17134-SP0
```

```
2018-07-11 15:19:37 [scrapy.crawler] INFO: Overridden settings: {'BOT_NAME':
'wxapp', 'DOWNLOAD_DELAY': 1, 'DUPEFILTER_CLASS':
'scrapy.dupefilters.BaseDupeFilter', 'LOGSTATS_INTERVAL': 0,
'NEWSPIDER_MODULE': 'wxapp.spiders', 'SPIDER_MODULES': ['wxapp.spiders']}
2018-07-11 15:19:37 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.corestats.CoreStats',
 'scrapy.extensions.telnet.TelnetConsole']
2018-07-11 15:19:37 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2018-07-11 15:19:37 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.spidermiddlewares.referer.RefererMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2018-07-11 15:19:37 [scrapy.middleware] INFO: Enabled item pipelines:
['wxapp.pipelines.WxappPipeline']
2018-07-11 15:19:37 [scrapy.extensions.telnet] DEBUG: Telnet console listening on
127.0.0.1:6023
2018-07-11 15:19:37 [scrapy.core.engine] INFO: Spider opened
2018-07-11 15:19:40 [scrapy.core.engine] DEBUG: Crawled (200) <GET
http://www.wxapp-union.com/article-4180-1.html> (referer: None)
[s] Available Scrapy objects:
[s]   scrapy         scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s]   crawler        <scrapy.crawler.Crawler object at 0x0000015E16380048>
[s]   item           {}
[s]   request        <GET http://www.wxapp-union.com/article-4180-1.html>
[s]   response       <200 http://www.wxapp-union.com/article-4180-1.html>
[s]   settings       <scrapy.settings.Settings object at 0x0000015E18A81D30>
[s]   spider         <WxappSpiderSpider 'wxapp_spider' at 0x15e18d71320>
[s] Useful shortcuts:
[s]   fetch(url[, redirect=True]) Fetch URL and update local objects (by default,
redirects are followed)
[s]   fetch(req)                  Fetch a scrapy.Request and update local objects
[s]   shelp()           Shell help (print this help)
[s]   view(response)    View response in a browser
```

offsitemiddleware 对于不在允许的域名中的爬虫如何处理
refermiddleware

scrapy shell 中常用的方法

Scrapy Shell 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，以及 Selector 对象 (对 HTML 及 XML 内容).

- 当shell载入后，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.
- 输入 response.selector 时，将获取到一个response 初始化的类 Selector 的对象，此时可以通过使用 response.selector.xpath()或response.selector.css() 来对 response 进行查询.
- Scrapy也提供了一些快捷方式，例如 response.xpath()或response.css()同样可以生效（如之前的案例）.

pdir(scrapy)

property:

__all__, __builtins__, __version__, _txv, cmdline, commands, conf, contracts, core, crawler, downloadermiddlewares, dupefilters, exceptions, extension, extensions, http, interfaces, item, link, linkextractors, logformatter, mail, middleware, pipelines, resolver, responsetypes, selector, settings, shell, signalmanager, signals, spiderloader, spidermiddlewares, spiders, squeues, statscollectors, twisted_version, utils, version_info

class:

Field: Container of field metadata
FormRequest: Inherit from this class (instead of object) to a keep a record of live
Item: Base class for all scraped items.
Request: Inherit from this class (instead of object) to a keep a record of live
Selector::class:`Selector` allows you to select parts of an XML or HTML text using CSS
Spider: Base class for scrapy spiders. All spiders must inherit from this

pdir(crawler)

property:

crawling, engine, extensions, logformatter, settings, signals, spider, stats

descriptor:

spiders: @property with getter

class:

spidercls: Base class for scrapy spiders. All spiders must inherit from this

function:

_Crawler__remove_handler:
_create_engine:
_create_spider:
crawl:
stop:

pdir(request)

property:

_body, _encoding, _meta, _url, callback, cookies, dont_filter, errback, flags, headers, method, priority

descriptor:

body: @property with getter, setter
encoding: @property with getter
meta: @property with getter
url: @property with getter, setter

function:

_get_body:
_get_url:
_set_body:
_set_url:
copy: Return a copy of this Request
replace: Create a new Request with the same attributes except for those

pdir(response)

property:

_DEFAULT_ENCODING, _body, _cached_benc, _cached_selector, _cached_ubody, _encoding, _url, flags, headers, request, status

descriptor:

body: @property with getter, setter
encoding: @property with getter
meta: @property with getter
selector: @property with getter
text: @property with getter, Body as unicode
url: @property with getter, setter

function:

_auto_detect_fun:
_body_declared_encoding:
_body_inferred_encoding:
_declared_encoding:
_get_body:
_get_url:
_headers_encoding:
_set_body:
_set_url:
body_as_unicode: Return body as unicode
copy: Return a copy of this Response
css: Shortcut method implemented only by responses whose content
follow: Return a:class:`~.Request` instance to follow a link ``url``.
replace: Create a new Response with the same attributes except for those
urljoin: Join this Response's url with a possible relative url to form an
xpath: Shortcut method implemented only by responses whose content

pdir(settings)

property:

_MutableMapping__marker, __slotnames__, _abc_cache, _abc_negative_cache, _abc_negative_cache_version, _abc_registry, attributes, frozen

descriptor:

defaults: @property with getter
overrides: @property with getter

function:

_assert_mutability:
_repr_pretty_:
_to_dict:
clear: D.clear() -> None. Remove all items from D.
copy: Make a deep copy of current settings.
copy_to_dict: Make a copy of current settings and convert to a dict.
delete:
freeze: Disable further changes to the current settings.
frozencopy: Return an immutable copy of the current settings.
get: Get a setting value without affecting its original type.
getbool: Get a setting value as a boolean.
getdict: Get a setting value as a dictionary. If the setting original type is a
getfloat: Get a setting value as a float.
getint: Get a setting value as an int.
getlist: Get a setting value as a list. If the setting original type is a list, a
getpriority: Return the current numerical priority value of a setting, or ``None`` if
getwithbase: Get a composition of a dictionary-like setting and its `_BASE`
items: D.items() -> a set-like object providing a view on D's items
keys: D.keys() -> a set-like object providing a view on D's keys
maxpriority: Return the numerical value of the highest priority present throughout
pop: D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
popitem: D.popitem() -> (k, v), remove and return some (key, value) pair
set: Store a key/value attribute with a given priority.
setdefault: D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D
setdict:
setmodule: Store settings from a module with a given priority.
update: Store key/value pairs with a given priority.
values: D.values() -> an object providing a view on D's values

pdir(spider)

property:

crawler, custom_settings, name, settings, start_urls

descriptor:

close: class staticmethod with getter, staticmethod(function) -> method
from_crawler: class classmethod with getter, classmethod(function) -> method
handles_request: class classmethod with getter, classmethod(function) -> method
logger: @property with getter
update_settings: class classmethod with getter, classmethod(function) -> method

function:

_set_crawler:
log: Log the given message at the given log level
make_requests_from_url: This method is deprecated.

parse:
set_crawler:
start_requests:

pdir(view(response))

property:

denominator, imag, numerator, real

function:

bit_length: int.bit_length() -> int
conjugate: Returns self, the complex conjugate of any int.
from_bytes: int.from_bytes(bytes, byteorder, *, signed=False) -> int
to_bytes: int.to_bytes(length, byteorder, *, signed=False) -> bytes

magic:

__index__: Return self converted to an integer, if self is suitable for use as an index into a list.

Scrapy shell 高级用法

使用 scrapy shell 也要添加 headers 的信息，不然服务器就会返回 500 的错误.

scrapy shell -s USER-AGENT="Mozilla/5.0 (Windows NT 6.1; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0" https://www.zhihu.com/question/56320032/

scarpy shell -h
Usage
=====
scrapy shell [url|file]

Interactive console for scraping the given url

Options
=======
--help, -h show this help message and exit
-c CODE evaluate the code in the shell, print the result and exit
--spider=SPIDER use this spider
--no-redirect do not handle HTTP 3xx status codes and print response as-is

Global Options

--logfile=FILE log file. if omitted stderr will be used
--loglevel=LEVEL, -L LEVEL
log level (default: DEBUG)
--nolog disable logging completely
--profile=FILE write python cProfile stats to FILE
--pidfile=FILE write process ID to FILE
--set=NAME=VALUE, -s NAME=VALUE

set/override setting (may be repeated)
--pdb enable pdb on failure

scrapy中selector的css选择器

css 选择器，和 xpath 的功能类似，都是用来定位 html 中的元素.

表达式	说明
*	选择所有节点
#container	选择 id 为 container 的节点
.container	选取所有 class 包含 container 的节点
li a	选取所有 li 下的所有 a 节点
ul + p	选择 ul 后面的第一个 p 元素，不是选择 li 中的节点, ul 和 p 相当于兄弟节点
div#container > ul	选取 id 为 container 的 div 的第一个 ul 子元素

表达式	说明
ul ~ p	选取与 ul 相邻的所有 p 元素
a[title]	选取所有有 title 属性的 a 元素
a[href="http://jobbole.com"]	选取所有 href 属性为 jobbole.com 值的 a 元素
a[href*="jobole"]	选取所有 href 属性包含 jobbole 的 a 元素
a[href^="http"]	选取所有 href 属性值以 http 开头的 a 元素
a[href$=".jpg"]	选取所有 href 属性值以.jpg 结尾的 a 元素
input[type=radio]:checked	选择选中的 radio 的元素

表达式	说明
div:not(#container)	选取所有 id 非 container 的 div 属性
li:nth-child(3)	选取第三个 li 元素
tr:nth-child(2n)	第偶数个 tr

```
response.xpath("//h1[@class='ph']/text()").get()
#'小程序的十万个为什么丨框架 '

# 使用 response.css 来获取文本
response.css("h1.ph::text").get()
'小程序的十万个为什么丨框架 '

# 使用 response.css 来获取属性
```

```
response.css("p.authors a::attr(href)").get()
'space-uid-17761.html'

from bs4 import BeautifulSoup
soup = BeautifulSoup(response.text, 'lxml')
soup.find('h1', attrs=['class', 'ph'])
# <h1 class="ph">小程序的十万个为什么丨框架 </h1>
```

scrapy shell 腾讯社招案例

```
# 启动
scrapy shell "http://hr.tencent.com/position.php?&start=0#a"

# 返回 xpath 选择器对象列表
response.xpath('//title')
[<Selector xpath='//title' data='<title>职位搜索 | 社会招聘 | Tencent 腾讯招聘
</title'>]

# 使用 extract()方法返回 Unicode 字符串列表
response.xpath('//title').extract()
['<title>职位搜索 | 社会招聘 | Tencent 腾讯招聘</title>']

# 打印列表第一个元素，没有则返回 None
response.xpath('//title').extract_first()
<title>职位搜索 | 社会招聘 | Tencent 腾讯招聘</title>

# 提取标签文本
response.xpath('//title/text()').extract_first()
'职位搜索 | 社会招聘 | Tencent 腾讯招聘'

# 提取职位发布日期. 返回 xpath 选择器对象列表
#contains 的用法, or 的用法, last()的含义
response.xpath('//*[contains(@class,"odd") or
contains(@class,"even")]/td[last()]/text()').extract()
['2017-06-02',
 '2017-06-02',
 '2017-06-02',
 '2017-06-02',
 '2017-06-02',
 '2017-06-02',
 '2017-06-02',
 '2017-06-02',
 '2017-06-02',
```

```
 '2017-06-02']

# 提取职位名称
response.xpath('//a[contains(@href,"position_detail.php?")]/text()').extract()

['19407-移动游戏平台合作（上海）',
 '19407-手游商业化与本地化策划（上海）',
 'OMG236-腾讯视频平台高级产品经理（深圳）',
 'OMG096-科技频道记者（北京）',
 '18402-项目管理',
 'IEG-招聘经理（深圳）',
 'OMG097-视觉设计师（北京）',
 'OMG097-策略产品经理/产品运营（北京）',
 'OMG097-策略产品经理/产品运营（北京）',
 'OMG097-数据产品经理(北京）']

response.xpath('//*[contains(@class,"odd") or contains(@class,"even")]/td[last()-1]/text()').extract()
['上海', '上海', '深圳', '北京', '深圳', '深圳', '北京', '北京', '北京', '北京']
```

Scrapy Shell总结

1. 可以方便我们做一些数据提取的测试代码.
2. 如果想要执行 scrapy 命令, 要先进入到 scrapy 所在的环境中.
3. 如果想要读取某个项目的配置信息, 那么应该先进入到这个项目的根目录中. 再执行`scrapy shell`命令.
4. 以后做数据提取的时候, 可以把现在 Scrapy Shell 中测试, 测试通过后再应用到代码中.

5.5. Request 和 Response 对象

Request对象

查看Request和request的方法和属性

scrapy shell http://www.wxapp-union.com/article-4180-1.html

pdir(request)

property:

_body, _encoding, _meta, _url, callback, cookies, dont_filter, errback, flags, headers, method, priority

descriptor:

body: @property with getter, setter
encoding: @property with getter
meta: @property with getter
url: @property with getter, setter

function:

_get_body:
_get_url:
_set_body:
_set_url:
copy: Return a copy of this Request
replace: Create a new Request with the same attributes except for those

```
from scrapy.http.request import Request
```

pdir(Request)

descriptor:

body: @property with getter, setter
encoding: @property with getter
meta: @property with getter
url: @property with getter, setter

function:

_get_body:
_get_url:
_set_body:
_set_url:
copy: Return a copy of this Request
replace: Create a new Request with the same attributes except for those

查看Request对象的源码

"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\http\request__init__.py"

```
class Request(object_ref):

    def __init__(self, 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):

        self._encoding = encoding  # this one has to be set first
        self.method = str(method).upper()
        self._set_url(url)
```

```
            self._set_body(body)
            assert isinstance(priority, int), "Request priority not an integer: %r" % priority
            self.priority = priority

            if callback is not None and not callable(callback):
                raise TypeError('callback must be a callable, got %s' %
type(callback).__name__)
            if errback is not None and not callable(errback):
                raise TypeError('errback must be a callable, got %s' %
type(errback).__name__)
            assert callback or not errback, "Cannot use errback without a callback"
            self.callback = callback
            self.errback = errback

            self.cookies = cookies or {}
            self.headers = Headers(headers or {}, encoding=encoding)
            self.dont_filter = dont_filter

            self._meta = dict(meta) if meta else None
            self.flags = [] if flags is None else list(flags)
```

Request对象在我们写爬虫，爬取一页的数据需要重新发送一个请求的时候调用. 这个类需要传递一些参数，其中比较常用的参数有

1. url：这个 request 对象发送请求的 url. 就是需要请求，并进行下一步处理的 url
2. callback：指定该请求返回的 Response，由哪个函数来处理.在下载器下载完相应的数据后执行的回调函数.
3. method：请求的方法. 默认为 GET 方法，可设置为"GET", "POST", "PUT"等，且保证字符串大写
4. headers：请求头，对于一些固定的设置，放在 settings.py 中指定就可以了. 对于那些非固定的经常变化的请求头，可以在发送请求的时候指定.
5. meta：比较常用. 用于在不同的请求之间传递数据用的. 为字典类型的变量. 如:

```
request_with_cookies = Request(
    url="http://www.example.com",
    cookies={'currency': 'USD', 'country': 'UY'},
    meta={'dont_merge_cookies': True}
)
```

6. encoding：编码. 默认的为 utf-8，使用默认的就可以了.
7. dont_filter：默认值是 False，表示由调度器进行请求对象指纹的过滤. 在执行同一个 url 多次重复的请求的时候要设置为 True，不经过调度器进行请求对象指纹的过滤. 如对于那些变化很快的页面，如热闹的帖子页，或者需要对某些字段进行更新的页面，或者是验证码的生成页，都需要多次向同一个 url 地址发送请求，所以这些 url 地址的 dont_filter 要设置为 False. 在模拟登录时也要设置为 True，第一次登录时向

主页发送一个请求，登录上去后还要再次向这个主页发送数据，如果执行过滤的话，第二次登录上去后的请求就无法发送了.

8. errback：指定错误处理函数，在发生错误的时候执行的函数.

Response对象

查看response的方法和属性

```
scrapy shell http://www.wxapp-union.com/article-4180-1.html

pdir(response)

property:
    _DEFAULT_ENCODING, _body, _cached_benc, _cached_selector,
_cached_ubody, _encoding, _url, flags, headers, request, status
descriptor:
    body: @property with getter, setter
    encoding: @property with getter
    meta: @property with getter
    selector: @property with getter
    text: @property with getter, Body as unicode
    url: @property with getter, setter
function:
    _auto_detect_fun:
    _body_declared_encoding:
    _body_inferred_encoding:
    _declared_encoding:
    _get_body:
    _get_url:
    _headers_encoding:
    _set_body:
    _set_url:
    body_as_unicode: Return body as unicode
    copy: Return a copy of this Response
    css: Shortcut method implemented only by responses whose content
    follow: Return a:class:`~.Request` instance to follow a link ``url``.
    replace: Create a new Response with the same attributes except for those
    urljoin: Join this Response's url with a possible relative url to form an
    xpath: Shortcut method implemented only by responses whose content
```

查看Response的方法和属性

```
from scrapy.http.response import Response
```

```
pdir(Response)

descriptor:
        body: @property with getter, setter
        meta: @property with getter
        text: @property with getter, For subclasses of TextResponse, this will return the body
        url: @property with getter, setter
function:
        _get_body:
        _get_url:
        _set_body:
        _set_url:
        copy: Return a copy of this Response
        css: Shortcut method implemented only by responses whose content
        follow: Return a:class:`~.Request` instance to follow a link ``url``.
        replace: Create a new Response with the same attributes except for those
        urljoin: Join this Response's url with a possible relative url to form an
        xpath: Shortcut method implemented only by responses whose content
```

查看Response的源码

```
class Response(object_ref):

        def __init__(self, url, status=200, headers=None, body=b'', flags=None, request=None):
                self.headers = Headers(headers or {})
                #  响应码
                self.status = int(status)
                #  响应体
                self._set_body(body)
                #  响应 url
                self._set_url(url)
                self.request = request
                self.flags = [] if flags is None else list(flags)


        def follow(self, url, callback=None, method='GET', headers=None, body=None,
                           cookies=None, meta=None, encoding='utf-8', priority=0,
                           dont_filter=False, errback=None):
                # type: (...) -> Request
                """
                Return a:class:`~.Request` instance to follow a link ``url``.
                It accepts the same arguments as ``Request.__init__`` method,
                but ``url`` can be a relative URL or a ``scrapy.link.Link`` object,
                not only an absolute URL.

                :class:`~.TextResponse` provides a:meth:`~.TextResponse.follow`
                method which supports selectors in addition to absolute/relative URLs
                and Link objects.
                """
```

```
        if isinstance(url, Link):
            url = url.url
        url = self.urljoin(url)
        return Request(url, callback,
                       method=method,
                       headers=headers,
                       body=body,
                       cookies=cookies,
                       meta=meta,
                       encoding=encoding,
                       priority=priority,
                       dont_filter=dont_filter,
                       errback=errback)
```

Response对象一般是由Scrapy给你自动构建的. 因此开发者不需要关心如何创建Response对象, 而是如何使用他. Response对象有很多属性, 可以用来提取数据的. 主要有以下属性

1. meta: 从其他请求传过来的 meta 属性, 可以用来保持多个请求之间的数据连接.
2. encoding: 返回当前字符串编码和解码的格式.
3. text: 将返回来的数据作为 unicode 字符串返回. text 和 body 都是网页的源码, 区别就在于 text 为经过解码后的 unicode 字符串.
4. body: 将返回来的数据作为 bytes 字符串返回.
5. xpath: xapth 选择器.
6. css: css 选择器.

scrapy模拟登录

注意: 模拟登陆时, 必须保证 settings.py 里的 COOKIES_ENABLED(Cookies 中间件) 处于开启状态. 如果是不需要登陆就能采集数据的网站, 要禁用 cookie. COOKIES_ENABLED = True 或 # COOKIES_ENABLED = False

1. 想要发送 post 请求, 那么推荐使用`scrapy.FormRequest`方法. 可以方便的指定表单数据.

下面是使用这种方法的爬虫例子:

```
import scrapy

class LoginSpider(scrapy.Spider):
    name = 'example.com'
    start_urls = ['http://www.example.com/users/login.php']

    def parse(self, response):
        return scrapy.FormRequest.from_response(
```

```
                response,
                formdata={'username': 'john', 'password': 'secret'},
                callback=self.after_login
            )

    def after_login(self, response):
        # check login succeed before going on
        if "authentication failed" in response.body:
            self.log("Login failed", level=log.ERROR)
            return

        # continue scraping with authenticated session...
```

2. 如果想在爬虫一开始的时候就发送 post 请求，那么应该重写`start_requests`方法. 在这个方法中，发送 post 请求.
https://docs.pythontab.com/scrapy/scrapy0.24/topics/request-response.html#topics-request-response-ref-request-userlogin

使用FormRequest发送POST请求

有时候我们想要在请求数据的时候发送 post 请求，那么这时候需要使用 Request 的子类 FormRequest 来实现.
如果想要在爬虫一开始的时候就发送 POST 请求，那么需要在爬虫类中重写 start_requests(self)方法，并且不再调用 start_urls 里的 url.

查看FromRequest的方法和属性

```
from scrapy.http.request.form import FormRequest

pdir(FormRequest)

descriptor:
    body: @property with getter, setter
    encoding: @property with getter
    from_response: class classmethod with getter, classmethod(function) -> method
    meta: @property with getter
    url: @property with getter, setter
function:
    _get_body:
    _get_url:
    _set_body:
    _set_url:
    copy: Return a copy of this Request
    replace: Create a new Request with the same attributes except for those
```

查看FormRequest的源码

```
from scrapy.http.request import Request

class FormRequest(Request):

    def __init__(self, *args, **kwargs):
        formdata = kwargs.pop('formdata', None)
        if formdata and kwargs.get('method') is None:
            kwargs['method'] = 'POST'

        super(FormRequest, self).__init__(*args, **kwargs)

        if formdata:
            items = formdata.items() if isinstance(formdata, dict) else formdata
            querystr = _urlencode(items, self.encoding)
            if self.method == 'POST':
                self.headers.setdefault(b'Content-Type', b'application/x-www-form-
urlencoded')
                self._set_body(querystr)
            else:
                self._set_url(self.url + ('&' if '?' in self.url else '?') + querystr)

    @classmethod
    def from_response(cls, response, formname=None, formid=None, formnumber=0,
formdata=None,
                      clickdata=None, dont_click=False, formxpath=None, formcss=None,
**kwargs):

        kwargs.setdefault('encoding', response.encoding)

        if formcss is not None:
            from parsel.csstranslator import HTMLTranslator
            formxpath = HTMLTranslator().css_to_xpath(formcss)

        form = _get_form(response, formname, formid, formnumber, formxpath)
        formdata = _get_inputs(form, formdata, dont_click, clickdata, response)
        url = _get_form_url(form, kwargs.pop('url', None))
        method = kwargs.pop('method', form.method)
        return cls(url=url, method=method, formdata=formdata, **kwargs)
```

查看start_requests的源码

"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\spiders__init__.py"

```
class Spider(object_ref):
```

```
    def start_requests(self):
        cls = self.__class__
        if method_is_overridden(cls, Spider, 'make_requests_from_url'):
            warnings.warn(
                "Spider.make_requests_from_url method is deprecated; it "
                "won't be called in future Scrapy releases. Please "
                "override Spider.start_requests method instead (see %s.%s)." % (
                    cls.__module__, cls.__name__
                ),
            )
            for url in self.start_urls:
                yield self.make_requests_from_url(url)
        else:
            for url in self.start_urls:
                yield Request(url, dont_filter=True)
```

从start_urls中遍历所有的链接，组装并发送Reuqest请求对象，默认是使用的get请求.

案例1：模拟登录人人网

新建项目
scrapy startproject renren_login
新建爬虫
scrapy genspider renren renren.com

修改 reren.py，进行模拟登录

```
# -*- coding: utf-8 -*-
import scrapy


class RenrenSpider(scrapy.Spider):
    name = 'renren'
    allowed_domains = ['renren.com']
    start_urls = ['http://renren.com/']

    # 如果想要在爬虫一开始的时候就向 start_urls 中的地址发送 POST 请求，那么需要在爬虫类中重写 start_requests(self)方法
    def start_requests(self):
        url = "http://www.renren.com/PLogin.do"
        data = {"email": "970138074@qq.com", 'password': "pythonspider"}
        request = scrapy.FormRequest(url,formdata=data,callback=self.parse_page)
        yield request
```

```
    def parse_page(self,response):
        # 把发送的post请求的响应数据写入到本地，查看是否登录成功
        # with open('dp.html','w',encoding='utf-8') as fp:
            # fp.write(response.text)
        # 登录成功后访问必须要登录才能访问的页面，并指定解析函数对响应进行处理
        request = scrapy.Request(url='http://www.renren.com/880151247/profile',
callback=self.parse_profile)
        yield request

    def parse_profile(self,response):
        with open('dp.html','w',encoding='utf-8') as fp:
            fp.write(response.text)
```

修改 settings.py

```
# Obey robots.txt rules
ROBOTSTXT_OBEY = False

# Override the default request headers:
DEFAULT_REQUEST_HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'Accept-Language': 'en',
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'
}
```

在项目根目录中创建 start.py

```
#encoding: utf-8

from scrapy import cmdline
cmdline.execute("scrapy crawl renren".split())
```

运行项目，进行测试

案例2：标准的模拟登陆步骤

正统模拟登录方法:
首先发送登录页面的 get 请求，获取到页面里的登录必须的参数（比如说 zhihu 登陆界面的 _xsrf）
然后和账户密码一起 post 到服务器，登录成功

```
# -*- coding: utf-8 -*-
import scrapy

class Renren2Spider(scrapy.Spider):
```

```
    name = "renren2"
    allowed_domains = ["renren.com"]
    start_urls = (
        "http://www.renren.com/PLogin.do",
    )

    # 处理 start_urls 里的登录 url 的响应内容，提取登陆需要的参数（如果需要的话)
    def parse(self, response):
        # 提取登陆需要的参数
        #_xsrf = response.xpath("//_xsrf").extract()[0]

        # 发送 post 请求参数，把账户密码和其它相关的参数发送到服务器，并调用指定回调函数处理
        yield scrapy.FormRequest.from_response(
                    response,
                    formdata = {"email" : "mr_mao_hacker@163.com", "password" : "axxxxxxxe"},#, "_xsrf" = _xsrf},
                    #调用回调函数来处理登陆成功后的页面
                    callback = self.parse_page
                )

    # 获取登录成功后的页面，访问需要登录后才能访问的页面，这里的 response 是登陆之后的 response
def parse_page(self, response):
    print("=========1===" + response.url)
        #with open("mao.html", "w") as filename:
        #       filename.write(response.body)
        #可以定义多个 url，多次迭代来发送请求
        url = "http://www.renren.com/422167102/profile"
        #发送一个 get 的请求，这时是已经登陆后发送的请求.
        yield scrapy.Request(url, callback = self.parse_newpage)

    # 处理响应内容
def parse_newpage(self, response):
        print("===========2====" + response.url)
        with open("xiao.html", "w") as filename:
            filename.write(response.body)
```

案例3：模拟登录豆瓣网（识别验证码）

图形验证码识别平台:
https://market.aliyun.com/products/57126001/cmapi014396.html#sku=yuncode839600006

```
# -*- coding: utf-8 -*-
import scrapy
from PIL import Image
from urllib import request
```

```
from base64 import b64encode
import requests


class DoubanSpider(scrapy.Spider):
    name = 'douban'
    allowed_domains = ['douban.com']
    start_urls = ['https://accounts.douban.com/login']
    login_url = 'https://accounts.douban.com/login'
    profile_url = 'https://www.douban.com/people/97956064/'
    editsignature_url = 'https://www.douban.com/j/people/97956064/edit_signature'

    # start_urls 中 url 的响应会使用 parse 函数进行解析.
    def parse(self, response):
        # 构造登录请求的 post 数据. 字段与登录表单中的字段对应
        formdata = {
            # 从哪个页面跳转来的
            'source': 'None',
            # 登录完成后要跳转到的页面
            'redir': 'https://www.douban.com/',
            'form_email': '970138074@qq.com',
            'form_password': 'pythonspider',
            'remember': 'on',
            'login': '登录'
        }
        # 提取出验证码的地址
        captcha_url = response.css('img#captcha_image::attr(src)').get()
        # 如果登录时有验证码, 才执行验证码识别的操作, 如果没有验证码, 就直接登录.
        if captcha_url:
            # 识别验证码并把结果添加到 formdata 中
            captcha = self.auto_regonize_captcha(captcha_url)
            formdata['captcha-solution'] = captcha
            # 提取出 captcha_id 并添加到 formdata 中
            captcha_id = response.xpath("//input[@name='captcha-id']/@value").get()
            formdata['captcha-id'] = captcha_id
        # 构造登录的 post 请求, 并指定登录响应的解析函数
        yield scrapy.FormRequest(url=self.login_url, formdata=formdata,
callback=self.parse_after_login)

    def parse_after_login(self, response):
        # 登录后会自动跳转到豆瓣首页, 如果响应的 url 是豆瓣首页, 就表明登录成功.
        if response.url == 'https://www.douban.com/':
            # 登录成功后跳转到个人中心页
            yield scrapy.Request(self.profile_url, callback=self.parse_profile)
            print('登录成功！')
        else:
            print('登录失败！')

    # 对个人中心页的响应进行处理
    def parse_profile(self, response):
        print(response.url)
```

```
            # 如果响应的 url 是个人中心的 url，就修改个性签名.
            if response.url == self.profile_url:
                # 构造修改个性命名的表单数据.
                ck = response.xpath("//input[@name='ck']/@value").get()
                formdata = {
                    'ck': ck,
                    'signature': '我可以自动识别图形验证码啦~~'
                }
                # 发送修改个性签名的 post 数据，在 scrapy 爬虫中，构造的 Reuqest 请求对象如果没有指定解析函数，会自动调用 parse 方法. 所以如果这里不指定，会再次进行登录请求. 可以在 parse 方法中打上断点进行调试.
                yield scrapy.FormRequest(self.editsignature_url, formdata=formdata, callback=self.parse_none)
            else:
                print('没有进入到个人中心')

    # 修改个性签名的响应数据不需要进行进一步的处理.
    def parse_none(self, response):
        pass

    # 手动输入验证码
    def regonize_captcha(self, image_url):
        request.urlretrieve(image_url, 'captcha.png')
        image = Image.open('captcha.png')
        image.show()
        captcha = input('请输入验证码: ')
        image.close()
        return captcha

    # 自动识别验证码
    def auto_regonize_captcha(self, image_url):
        captcha_url = image_url
        request.urlretrieve(captcha_url, 'captcha.png')

        # 构建验证码识别的 url
        recognize_url = 'http://jisuyzmsb.market.alicloudapi.com/captcha/recognize?type=e'

        formdata = {}
        # 对图片进行 base64 编码
        with open('captcha.png', 'rb') as fp:
            data = fp.read()
            pic = b64encode(data)
            formdata['pic'] = pic

        appcode = '831a890b2cfe4ea0a8e345078434ebfc'
        headers = {
            'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8',
            'Authorization': 'APPCODE ' + appcode
        }
        # 发送识别验证码的 post 请求
        response = requests.post(recognize_url, data=formdata, headers=headers)
```

```
        # 从返回的 json 数据中提取出验证码
        result = response.json()
        code = result['result']['code']
        return code
```

5.6. 下载文件和图片

Scrapy 为下载 item 中包含的文件(比如在爬取到产品时，同时也想保存对应的图片)提供了一个可重用的 item pipelines. 这些 pipeline 有些共同的方法和结构(我们称之为 media pipeline). 一般来说你会使用 Files Pipeline 或者 Images Pipeline.

images pipeline 继承自 files pipeline，增加了图片的一些方法，如生成缩略图.

为什么要使用scrapy内置的下载图片/文件的方法

1. 避免重新下载最近已经下载过的文件.
2. 可以方便的指定文件存储的路径.
3. 可以将下载的图片转换成通用的格式. 比如 png 或 jpg.
4. 可以方便的生成缩略图.
5. 可以方便的检测图片的宽和高，确保他们满足最小限制.
6. 异步下载，效率非常高.

下载文件的Files Pipeline

当使用 Files Pipeline 下载文件的时候，按照以下步骤来完成

1. 定义好一个 Item，然后在这个 item 中定义两个属性，分别为 file_urls 以及 files. file_urls 是一个列表，用来存储需要下载的文件的 url 链接. files pipeline 就会从这个列表中读取所有的图片或文件的 url 地址进行下载. files 不常用，用来保存下载的文件的相应信息.
2. 当文件下载完成后，会把文件下载的相关信息存储到 item 的 files 属性中. 比如下载路径，下载的 url 和文件的校验码等.
3. 在配置文件 settings.py 中配置 FILES_STORE，这个配置是用来设置文件下载下来的路径.
4. 启动 pipeline: 在 ITEM_PIPELINES 中设置 scrapy.pipelines.files.FilesPipeline:1.

下载图片的Images Pipeline

当使用 Images Pipeline 下载文件的时候，按照以下步骤来完成

1. 定义好一个 Item，然后在这个 item 中定义两个属性，分别为 image_urls 以及 images. image_urls 是一个列表，用来存储需要下载的图片的 url 链接，
2. 当文件下载完成后，会把文件下载的相关信息存储到 item 的 images 属性中．比如下载路径，下载的 url 和图片的校验码等.
3. 在配置文件 settings.py 中配置 IMAGES_STORE，这个配置是用来设置图片下载下来的路径.
4. 启动 pipeline：在 ITEM_PIPELINES 中设置 scrapy.pipelines.images.ImagesPipeline:1.

汽车之家宝马5系高清图片下载实战.

新建项目并编写爬虫

新建scrapy项目 scrapy startproject bmw_image
新建爬虫, scrapy genspider bmw5

修改items.py，定义item类

```
# -*- coding: utf-8 -*-

# Define here the models for your scraped items
#
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/items.html

import scrapy


class BmwItem(scrapy.Item):
    category = scrapy.Field()
    # urls 保存图片的下载链接
    urls = scrapy.Field()
```

修改spider/bmw5.py，定义爬虫

```
# -*- coding: utf-8 -*-
import scrapy
```

```python
from bmw.items import BmwItem

class Bmw5Spider(scrapy.Spider):
    name = 'bmw5'
    allowed_domains = ['car.autohome.com.cn']
    start_urls = ['https://car.autohome.com.cn/pic/series/65.html']

    # 提取图片的缩略图
    def parse(self, response):
        # 每一个'uibox'代表一类的图片. 第 1 个 uibox 不需要获取.
        uiboxs = response.xpath("//div[@class='uibox']")[1:]
        # SelectorList -> list, 可以进行遍历
        for uibox in uiboxs:
            # 图片的分类
            category = uibox.xpath(".//div[@class='uibox-title']/a/text()").get()
            # 提取每一个图片分类下面的所有图片缩略图的 url 地址
            urls = uibox.xpath(".//ul/li/a/img/@src").getall()
            # 对提取到的 url 地址进行补全
            # for url in urls:
            #     # url = "https" + url
            #     # 使用 response.urljoin()对网址进行拼接
            #     url = response.urljoin(url)
            #     print(url)
            urls = list(map(lambda url:response.urljoin(url),urls))
            urls = [response.urljoin(url) for url in urls]
            item = BmwItem(category=category, urls=urls)
            yield item
```

os路径的使用方法

新建文件test.py，启动终端，进入到本文件所在的目录中, python test.py

```python
#encoding: utf-8
import os

# 获取本文件的路径
file_name = os.path.abspath(__file__)
print("file_name:%s"%file_name)
# 获取本文件上一级的路径
folder_name = os.path.dirname(os.path.abspath(__file__))
# 直接对文件使用 dirname 就能获取到文件夹的名称
folder_name = os.path.dirname((__file__)
print("folder_name:%s"%folder_name)
# 拼接路径
images_path = os.path.join(os.path.dirname(os.path.abspath(__file__)),'images')
print("images_path:%s"%images_path)

# 判断路径是否存在
```

```
if not os.path.exists(images_path):
    os.mkdir(images_path)
else:
    print('images 文件夹存在')
```

方法一：使用传统方法来保存图片

缺点，只能一张张的下载图片

修改pipelines.py

```
# -*- coding: utf-8 -*-

# Define your item pipelines here
#
# Don't forget to add your pipeline to the ITEM_PIPELINES setting
# See: http://doc.scrapy.org/en/latest/topics/item-pipeline.html
import os
from urllib import request
from scrapy.pipelines.images import ImagesPipeline
from bmw import settings


class BmwPipeline(object):
    def __init__(self):
        # 判断 images 文件夹是否存在，如果不存在，就在 piepleines.py 同级目录下新建一个 images 文件夹. 因为后面要不断的使用 images 这个路径，所以要保存为一个类属性
        self.path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'images')
        if not os.path.exists(self.path):
            os.mkdir(self.path)

    # 传统方式：单线程自定义方法下载图片
    def process_item(self, item, spider):
        # 取出图片的分类，把图片保存在以分类命名的文件夹中
        category = item['category']
        urls = item['urls']
        # 在 images 文件夹下新建一个以 category 命名的文件夹
        category_path = os.path.join(self.path, category)
        if not os.path.exists(category_path):
            os.mkdir(category_path)
        for url in urls:
            # 从 url 中取出来图片的名字
            image_name = url.split('_')[-1]
            # 把图片保存到分类目录下, images_name 命名的文件中
            request.urlretrieve(url, os.path.join(category_path, image_name))
        return item
```

在settings.py中进行相应的设置，运行爬虫

```
ROBOTSTXT_OBEY = False

DEFAULT_REQUEST_HEADERS = {
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
   'Accept-Language': 'en',
   'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'
}

ITEM_PIPELINES = {
    'bmw.pipelines.BmwPipeline': 300
}
```

方法二：使用Images Pipeline来下载图片

修改items.py文件，定义image_urls和images

```
# -*- coding: utf-8 -*-

import scrapy


class BmwItem(scrapy.Item):
    category = scrapy.Field()
    # image_urls 保存图片的下载链接
    image_urls = scrapy.Field()
    # images 保存下载的图片的信息
    images = scrapy.Field()
```

修改 bmw5.py 爬虫文件，把 item 中的 urls 修改为 image_urls

```
# -*- coding: utf-8 -*-
import scrapy
from bmw.items import BmwItem

class Bmw5Spider(scrapy.Spider):
    name = 'bmw5'
    allowed_domains = ['car.autohome.com.cn']
    start_urls = ['https://car.autohome.com.cn/pic/series/65.html']

    #  提取图片的缩略图
    def parse(self, response):
        #  每一个'uibox'代表一类的图片. 第 1 个 uibox 不需要获取.
        uiboxs = response.xpath("//div[@class='uibox']")[1:]
        # SelectorList -> list，可以进行遍历
```

```
        for uibox in uiboxs:
            # 图片的分类
            category = uibox.xpath(".//div[@class='uibox-title']/a/text()").get()
            # 提取每一个图片分类下面的所有图片缩略图的 url 地址
            urls = uibox.xpath(".//ul/li/a/img/@src").getall()
            # 对提取到的 url 地址进行补全
            # for url in urls:
            #     # url = "https" + url
            #     # 使用 response.urljoin()对网址进行拼接
            #     url = response.urljoin(url)
            #     print(url)
            urls = list(map(lambda url:response.urljoin(url),urls))
            urls = [response.urljoin(url) for url in urls]
            item = BmwItem(category=category,image_urls=urls)
            yield item
```

修改settings.py，启用imagespipeline并设置图片保存的路径.

```
import os

# Obey robots.txt rules
ROBOTSTXT_OBEY = False

DEFAULT_REQUEST_HEADERS = {
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
   'Accept-Language': 'en',
   'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'
}


ITEM_PIPELINES = {
    # 'bmw.pipelines.BmwPipeline': 300,
    'scrapy.pipelines.images.ImagesPipeline': 1
}

# 图片下载的路径，供 images pipelines 使用
IMAGES_STORE = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'images')
```

运行爬虫，发现所有的图片都被下载到了 full 这个根目录中.

重写ImagesPipeline

使用默认的 ImagesPipeline 时所有的图片都被下载到同一个文件夹中，想要把不同分类的图片保存到不同的文件夹中，需要重写 ImagesPipeline.

```
from scrapy.pipelines.images import ImagesPipeline

pdir(ImagesPipeline)
```

```
property:
    DEFAULT_FILES_RESULT_FIELD, DEFAULT_FILES_URLS_FIELD,
DEFAULT_IMAGES_RESULT_FIELD, DEFAULT_IMAGES_URLS_FIELD, EXPIRES,
LOG_FAILED_RESULTS, MEDIA_NAME, MIN_HEIGHT, MIN_WIDTH,
STORE_SCHEMES, THUMBS
descriptor:
    from_settings: class classmethod with getter, classmethod(function) -> method
class:
    SpiderInfo:
function:
    _cache_result_and_execute_waiters:
    _check_media_to_download:
    _get_store:
    _handle_statuses:
    _key_for_pipe: >>> MediaPipeline()._key_for_pipe("IMAGES")
    _modify_media_request:
    _process_request:
    convert_image:
    file_downloaded:
    file_key:
    file_path:
    from_crawler:
    get_images:
    get_media_requests: Returns the media requests to download
    image_downloaded:
    image_key:
    inc_stats:
    item_completed: Called per item when all media requests has been processed
    media_downloaded: Handler for success downloads
    media_failed: Handler for failed downloads
    media_to_download: Check request before starting download
    open_spider:
    process_item:
    thumb_key:
    thumb_path:
```

查看 ImagesPipeline 的源码，其中的 file_path 就是图片保存路径的方法，需要重写这个方法

"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\pipelines\images.py"

```
class ImagesPipeline(FilesPipeline):

def file_path(self, request, response=None, info=None):

    # check if called from image_key or file_key with url as first argument
    if not isinstance(request, Request):
        _warn()
        url = request
```

```
    else:
        url = request.url

    # detect if file_key() or image_key() methods have been overridden
    if not hasattr(self.file_key, '_base'):
        _warn()
        return self.file_key(url)
    elif not hasattr(self.image_key, '_base'):
        _warn()
        return self.image_key(url)
    ## end of deprecation warning block

    image_guid = hashlib.sha1(to_bytes(url)).hexdigest()    # change to request.url after deprecation
    return 'full/%s.jpg' % (image_guid)
```

修改pipelines.py, 重写file_path这个方法.

```
# -*- coding: utf-8 -*-

# Define your item pipelines here
#
# Don't forget to add your pipeline to the ITEM_PIPELINES setting
# See: http://doc.scrapy.org/en/latest/topics/item-pipeline.html
import os
from urllib import request
from scrapy.pipelines.images import ImagesPipeline
from bmw import settings


class BmwPipeline(object):
    def __init__(self):
        # 判断 images 文件夹是否存在, 如果不存在, 就在 piepleines.py 同级目录下新建一个 images 文件夹. 因为后面要不断的使用 images 这个路径, 所以要保存为一个类属性
        self.path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'images')
        if not os.path.exists(self.path):
            os.mkdir(self.path)

    # 传统方式: 单线程自定义方法下载图片
    def process_item(self, item, spider):
        # 取出图片的分类, 把图片保存在以分类命名的文件夹中
        category = item['category']
        urls = item['urls']
        # 在 images 文件夹下新建一个以 category 命名的文件夹
        category_path = os.path.join(self.path, category)
        if not os.path.exists(category_path):
            os.mkdir(category_path)
        for url in urls:
            # 从 url 中取出来图片的名字
            image_name = url.split('_')[-1]
            # 把图片保存到分类目录下, images_name 命名的文件中
            request.urlretrieve(url, os.path.join(category_path, image_name))
```

```
            return item


class BMWImagesPipeline(ImagesPipeline):
    # 重写 ImagesPipeline 中的 get_media_requests 和 file_path 这两个方法, get_media_requests 这个方法是在发送下载请求之前调用。file_path 方法是在请求之后调用.
    def get_media_requests(self, item, info):
        # get_media_requests 方法其实就是构造下载文件请求的
        """
        # get_media_requests 方法的源码
        def get_media_requests(self, item, info):
            # images_urls_field 最终还是等于 image_urls, 所以 get_media_requests 方法实际上就是从 item 中的 image_urls 列表中读取每一个链接, 构造请求对象. 返回一个包含所有请求对象的列表
            return [Request(x) for x in item.get(self.images_urls_field, [])]
        """
        # 调用父类即 ImagesPipeline 中的 get_media_requests 方法, 获取构造的请求对象. get_media_requests 就是从 image_urls 中读取 url, 构造 Request 请求对象, 得到的就是所有请求对象的列表.
        request_objs = super(BMWImagesPipeline, self).get_media_requests(item, info)
        for request_obj in request_objs:
            # 把 item 绑定到每一个 Request 请求对象上, 相当于在原来的 request 对象上添加了一个 item 的属性, 它的值为 item 的值. 因为 get_media_requests 这个方法是在发送下载请求之前调用。file_path 方法是在请求之后调用, 所以就可以在 file_path 这个方法中从 Request 请求对象中读取 item. 因为 item 中保存了文件名和图片的分类, 这样就可以在 file_path 这个方法中读取到图片的分类了.
            request_obj.item = item
        # 一定要把 request_objs 返回, 与父类中的方法保持一致, 否则就不会去下载图片.
        return request_objs

    def file_path(self, request, response=None, info=None):
        # 这个方法是在图片将要被存储的时候调用, 来获取这个图片存储的路径. 即图片将要被存储到硬盘中时就调用 file_path 读取设置中的路径.
        # 调用父类的 file_path 方法, 获取父类的 file_path 返回的文件路径.
        path = super(BMWImagesPipeline, self).file_path(request, response, info)
        # 获取子分类, 判断是否存在子分类的目录, 如果存在, 就把图片保存到子分类的目录中, 如果不存在, 就创建对应的目录.
        # 图片的目录信息保存在 item 中, 无法直接获取, 由于 get_media_requests 这个方法中有 item, 所以可以重写 get_media_requests 这个方法. 把 item 绑定到一个 file_path 和 get_media_request 共有的对象中, 再在 file_path 中从这个对象中读取 item 即可.
        category = request.item.get('category')
        images_store = settings.IMAGES_STORE
        category_path = os.path.join(images_store, category)
        if not os.path.exists(category_path):
            os.mkdir(category_path)
        """
        # 查看 file_path 的源码, 返回的是 url 地址的 sha1 值, 所以 path 中的内容就是图片 url 对应的 sha1 值. 把 full/ 替换掉就得到了图片 url 地址的 sha1 值
        def file_path():
            ...
```

```
            image_guid = hashlib.sha1(to_bytes(url)).hexdigest()    # change to request.url after deprecation
            return 'full/%s.jpg' % (image_guid)
        """
        # 父类的路径是以 full 开头的，也就是把所有的文件都保存在 full 目录下
        image_name = path.replace("full/", "")
        # 构建包括文件名在内的路径并返回，这样就与父类中返回的内容相同了.
        image_path = os.path.join(category_path, image_name)
        return image_path
```

修改settings.py，激活BMWImagesPipeline

```
import os


BOT_NAME = 'bmw'

SPIDER_MODULES = ['bmw.spiders']
NEWSPIDER_MODULE = 'bmw.spiders'

# Obey robots.txt rules
ROBOTSTXT_OBEY = False

# Override the default request headers:
DEFAULT_REQUEST_HEADERS = {
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
   'Accept-Language': 'en',
   'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'
}

# Configure item pipelines
# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html
ITEM_PIPELINES = {
    # 'bmw.pipelines.BmwPipeline': 300,
    # 'scrapy.pipelines.images.ImagesPipeline': 1
    'bmw.pipelines.BMWImagesPipeline': 1
}

# 图片下载的路径，供 images pipelines 使用
IMAGES_STORE = os.path.join(os.path.dirname(os.path.dirname(__file__)), 'images')
```

把爬虫修改为crawl.Spider爬虫，爬取所有高清图片

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor
from bmw.items import BmwItem
```

```
# 把爬虫修改为CrawlSpider类的爬虫
class Bmw5Spider(CrawlSpider):
    name = 'bmw5'
    allowed_domains = ['car.autohome.com.cn']
    start_urls = ['https://car.autohome.com.cn/pic/series/65.html']

    rules = (
        # 提取出更多图片列表页的地址. 可能还存在着分页, 所以要对页面进行跟进. 这个规则既可以匹配到 "更多" 的地址, 又能匹配到 "下一页" 的地址

Rule(LinkExtractor(allow=r"https://car.autohome.com.cn/pic/series/65.+"),callback="parse_page",follow=True),
    )

    # CrawlSpider 爬虫中的parse 方法是用来解析 start_urls 中的地址的, 所以这里要定义一个新的解析函数
    def parse_page(self, response):
        # 提取出图片的分类
        category = response.xpath("//div[@class='uibox']/div/text()").get()
        # 提取出图片的链接地址
        srcs = response.xpath('//div[contains(@class,"uibox-con")]/ul/li//img/@src').getall()
        # 构建高清图的url 地址, 把缩略图中的t_去掉就得到高清图的地址
        # srcs = list(map(lambda x:response.urljoin(x),srcs))
        srcs = list(map(lambda x:response.urljoin(x.replace("t_","")),srcs))
        srcs = [response.urljoin(src.replace("t_","")) for src in srcs]
        yield BmwItem(category=category,image_urls=srcs)
```

5.7. Downloader Middlewares（下载器中间件）

下载器中间件

```
# 参考文章：
# https://www.jianshu.com/p/a94d7de5560f
```

下载器中间件按照优先级被调用的：当 request 从引擎向下载器传递时，数字小的下载器中间件先执行，数字大的后执行；当下载器将 response 向引擎传递，数字大的下载器中间件先执行，小的后执行。

scrapy 提供了一套基本的下载器中间件，

```
{
    'scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware': 100,
    'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware': 300,
```

```
'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware': 350,

'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware': 400,
    'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': 500,
    'scrapy.downloadermiddlewares.retry.RetryMiddleware': 550,
    'scrapy.downloadermiddlewares.ajaxcrawl.AjaxCrawlMiddleware': 560,
    'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware': 580,

'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware': 590,
    'scrapy.downloadermiddlewares.redirect.RedirectMiddleware': 600,
    'scrapy.downloadermiddlewares.cookies.CookiesMiddleware': 700,
    'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware': 750,
    'scrapy.downloadermiddlewares.stats.DownloaderStats': 850,
    'scrapy.downloadermiddlewares.httpcache.HttpCacheMiddleware': 900,
}
```

见链接
https://docs.scrapy.org/en/latest/topics/settings.html#std:setting-DOWNLOADER_MIDDLEWARES_BASE

下载器中间件是个类，类里可以定义方法，例如 process_request(), process_response(), process_exception()

process_request():

process_request()的参数是 request, spider

参数 request 是个字典，字典里包含了 headers、url 等信息

process_request()可以利用参数 request 里面的信息，对请求做修改，这时一般返回的是 None，典型的任务是修改 User-agent、变换代理

如果根据参数 request 里的 url 直接就去做抓取，返回 response 对象，返回的 response 对象就会不经过剩下的下载器中间件，直接返回到引擎

如果对请求做了修改，返回的是 request 对象，就会发回到调度器，等待调度

process_response(request, response, spider)

返回的必须是 Response、Request 或 IgnoreRequest 异常

爬虫中间件

爬虫中间件的作用：
处理引擎传递给爬虫的响应；
处理爬虫传递给引擎的请求；
处理爬虫传递给引擎的数据项。

scrapy 提供的基本爬虫中间件
https://docs.scrapy.org/en/latest/topics/settings.html#std:setting-SPIDER_MIDDLEWARES_BASE

如何自定义爬虫中间件
https://docs.scrapy.org/en/latest/topics/spider-middleware.html#writing-your-own-spider-middleware

反反爬虫相关机制

Some websites implement certain measures to prevent bots from crawling them, with varying degrees of sophistication. Getting around those measures can be difficult and tricky, and may sometimes require special infrastructure. Please consider contacting commercial support if in doubt.
(有些些网站使用特定的不同程度的复杂性规则防止爬虫访问，绕过这些规则是困难和复杂的，有时可能需要特殊的基础设施，如果有疑问，请联系商业支持.)

常见中间件见：
http://scrapy.readthedocs.io/en/latest/topics/downloader-middleware.html

来自于 Scrapy 官方文档描述:
http://doc.scrapy.org/en/master/topics/practices.html#avoiding-getting-banned

通常防止爬虫被反主要有以下几个策略:

- 动态设置 User-Agent（随机切换 User-Agent，模拟不同用户的浏览器信息）
- 禁用 Cookies（也就是不启用 cookies middleware，不向 Server 发送 cookies，有些网站通过 cookie 的使用发现爬虫行为）
    - 可以通过 COOKIES_ENABLED 控制 CookiesMiddleware 开启或关闭

- 设置延迟下载（防止访问过于频繁，设置为 2 秒 或更高）
- Google Cache 和 Baidu Cache: 如果可能的话，使用谷歌/百度等搜索引擎服务器页面缓存获取页面数据.
- 使用 IP 地址池: VPN 和代理 IP，现在大部分网站都是根据 IP 来 ban 的.
- 使用 Crawlera（专用于爬虫的代理组件），正确配置和设置下载中间件后，项目所有的 request 都是通过 crawlera 发出.

```
DOWNLOADER_MIDDLEWARES = {
    'scrapy_crawlera.CrawleraMiddleware': 600
}

CRAWLERA_ENABLED = True
CRAWLERA_USER = '注册/购买的 UserKey'
CRAWLERA_PASS = '注册/购买的 Password'
```

查看官方文档，自定义downloader middleware

查看官方文档, Activating a downloader middleware 中如何去自定义一个 middleware. 这个 downlaoder middleware 既可以处理 request 又可以处理 response.

Writing your own downloader middleware

Each middleware component is a Python class that defines one or more of the following methods:

***class* scrapy.downloadermiddlewares.DownloaderMiddleware**

> **process_request(*request, spider*)**
>
> This method is called for each request that goes through the download middleware.
>
> **process_request()** should either: return **None**, return a **Response** object, return a **Request** object, or raise **IgnoreRequest**.

If it returns **None**, Scrapy will continue processing this request, executing all other middlewares until, finally, the appropriate downloader handler is called the request performed (and its response downloaded).

If it returns a **Response** object, Scrapy won't bother calling *any* other **process_request()** or **process_exception()** methods, or the appropriate download function; it'll return that response. The **process_response()** methods of installed middleware is always called on every response.

If it returns a **Request** object, Scrapy will stop calling process_request methods and reschedule the returned request. Once the newly returned request is performed, the appropriate middleware chain will be called on the downloaded response.

If it raises an **IgnoreRequest** exception, the **process_exception()** methods of installed downloader middleware will be called. If none of them handle the exception, the errback function of the request (**Request.errback**) is called. If no code handles the raised exception, it is ignored and not logged (unlike other exceptions).

Parameters:

- **request** (**Request** object) – the request being processed
- **spider** (**Spider** object) – the spider for which this request is intended

process_response(*request, response, spider*)

process_response() should either: return a **Response** object, return a **Request** object or raise a **IgnoreRequest** exception.

If it returns a **Response** (it could be the same given response, or a brand-new one), that response will continue to be processed with the **process_response()** of the next middleware in the chain.

If it returns a **Request** object, the middleware chain is halted and the returned request is rescheduled to be downloaded in the future. This is the same behavior as if a request is returned from **process_request()**.

If it raises an **IgnoreRequest** exception, the errback function of the request (**Request.errback**) is called. If no code handles the raised exception, it is ignored and not logged (unlike other exceptions).

Parameters:

- **request** (is a **Request** object) – the request that originated the response
- **response** (**Response** object) – the response being processed
- **spider** (**Spider** object) – the spider for which this response is intended

process_exception(*request, exception, spider*)

Scrapy calls **process_exception()** when a download handler or a **process_request()** (from a downloader middleware) raises an exception (including an **IgnoreRequest** exception)

process_exception() should return: either **None**, a **Response** object, or a **Request** object.

If it returns **None**, Scrapy will continue processing this exception, executing any other **process_exception()** methods of installed middleware, until no middleware is left and the default exception handling kicks in.

If it returns a **Response** object, the **process_response()** method chain of installed middleware is started, and Scrapy won't bother calling any other **process_exception()** methods of middleware.

If it returns a **Request** object, the returned request is rescheduled to be downloaded in the future. This stops the execution of **process_exception()** methods of the middleware the same as returning a response would.

Parameters:

request (is a **Request** object) – the request that generated the exception
exception (an **Exception** object) – the raised exception
spider (**Spider** object) – the spider for which this request is intended

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

重写了 process_request(request, spider)函数，就会处理 request
重写了 process_response(request, response, spider)，就会处理 response
重写了 process_exception(request, exception, spider)，就会处理异常

设置下载中间件

下载器中间件是引擎(crawler.engine)和下载器(crawler.engine.download())之间通信的一层组件，可以有多个下载中间件被加载运行．在这个中间件中我们可以设置代理，更换请求头等来达到反反爬虫的目的．

下载中间件一般写在 settings 同级目录下．

1．当引擎传递请求给下载器的过程中，下载中间件可以对请求进行处理 （例如增加http header信息，增加proxy信息等）；

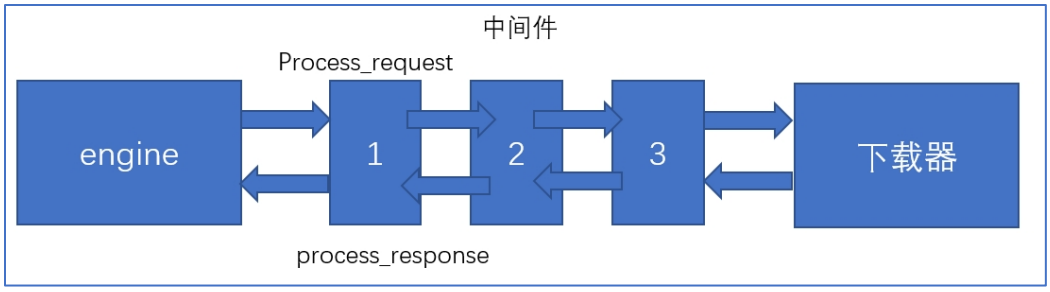
2．在下载器完成http请求，传递响应给引擎的过程中，下载中间件可以对响应进行处理（例如进行gzip的解压等）

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)．

```
DOWNLOADER_MIDDLEWARES = {
    'mySpider.middlewares.MyDownloaderMiddleware': 543,
}
```

要编写下载器中间件，可以在下载器中实现两个方法．一个是process_request(self, request, spider)，这个方法是在请求发送之前会执行，是在请求发送给下载器之前对请求进行修改的方法，如可以添加随机的请求头和随机的代理ip．还有一个是process_response(self, request, response, spider)，这个方法是数据下载到引擎之前执行．

Request对象由engine经过下载器中的process_request方法处理后传递给下载器的，Response对象则是由下载器经中间件中的process_response方法处理后发送给引擎的．



process_request(self, request, spider)

这个方法是下载器在发送请求之前会执行的. 一般可以在这个里面设置随机代理ip等.

1. 说明:

   - 当每个request通过下载中间件时, 该方法被调用.
   - 此方法必须要写
   - process_request() 必须返回以下其中之一: 一个 None 、一个 Response 对象、一个 Request 对象或 raise IgnoreRequest:

2. 返回值

   - 返回 None: 如果返回 None, Scrapy 将继续处理该 request, 把 request 对象交给其它中间件进行处理, 执行其他中间件中的相应方法, 直到合适的下载器处理函数被调用.
   - 返回 Response 对象: Scrapy 将不会把这个 Response 对象发送给其他的下载器, 也就不会调用其它下载中间件中的 process_request 方法, 而是把这个 Response 对象直接返回给引擎. 已经激活的中间件的 process_response()方法则会在每个 response 返回时被调用.
   - 返回一个新的 Request 对象: 不再使用之前的 request 对象去下载数据, 而是根据现在返回的 request 对象返回数据. 会把这个新的 Request 对象交给下一个中间件进行处理.
   - 如果其 raise 一个 IgnoreRequest 异常, 则下载中间件的 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).

3. 参数

   - request: 发送请求的 request 对象.
   - spider: 发送请求的 spider 对象.

process_response(self, request, response, spider)

当下载器完成http请求，传递响应给引擎的时候会执行这个方法.

1. 参数

   - request: request对象.
   - response: 被处理的response对象.
   - spider: spider对象.

2. 返回值

   process_request() 必须返回以下其中之一: 返回一个 Response 对象、 返回一个 Request 对象或 raise 一个 IgnoreRequest 异常.

   - 返回 Response 对象: 可以与传入的 response 相同，也可以是全新的对象，会将这个 response 对象传给其他中间件进行处理，最终传给爬虫.
   - 返回 Request 对象: 下载器链被切断，返回的 request 会重新交给调度器进行下载.
   - ???(哪个说明正确???)如果抛出一个异常，那么调用 request 的 errback 方法，如果没有指定这个方法，那么会抛出一个异常.
   - ???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).

   下载中间件

```
from scrapy.downloadermiddlewares.useragent import UserAgentMiddleware

pdir(UserAgentMiddleware)

descriptor:
    from_crawler: class classmethod with getter, classmethod(function) -> method
function:
    process_request:
    spider_opened:

pdir(UserAgentMiddleware())

property:
    user_agent
descriptor:
    from_crawler: class classmethod with getter, classmethod(function) -> method
function:
    process_request:
    spider_opened:
```

随机请求头中间件

爬虫在频繁访问一个页面的时候，这个请求头如果一直保持一致．那么很容易被服务器发现，从而禁止掉这个请求头的访问．因此我们要在访问这个页面之前随机的更改请求头，这样才可以避免爬虫被抓．随机更改请求头，可以在下载中间件中实现．在请求发送给服务器之前，随机的选择一个请求头．这样就可以避免总使用一个请求头了．

user-agent列表:
http://www.useragentstring.com/pages/useragentstring.php?typ=Browser

新建项目useragent_demo

新建scrapy爬虫 httpbin.py

```
# -*- coding: utf-8 -*-
import scrapy
import json

class HttpbinSpider(scrapy.Spider):
    name = 'httpbin'
    allowed_domains = ['httpbin.org']
    start_urls = ['http://httpbin.org/user-agent']

    def parse(self, response):
        # 返回的是 json 数据，使用 loads 转换为字典，再从中取出来'user-agent'的值
        user_agent = json.loads(response.text)['user-agent']
        print('='*30)
        print(user_agent)
        print('='*30)
        # 因为设置了每次请求使用随机请求头，需要再次使用 start_urls 中的 url 构建新的 Request 请求对象，再次发送请求，查看设置的请求头．注意需要设置 dont_filter=True 才能再次发送同一个 url 地址的请求.
        yield scrapy.Request(self.start_urls[0],dont_filter=True)
```

修改middlewares.py，定义UserAgentDownloadMiddleware

```
# -*- coding: utf-8 -*-

import random

# 定义 user-agent 随机请求头下载中间件
class UserAgentDownloadMiddleware(object):
    USER_AGENTS = [
        'Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_8; en-us) AppleWebKit/534.50 (KHTML, like Gecko) Version/5.1 Safari/534.50',
```

```
            'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36',
            'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;',
            'Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)',
            'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv,2.0.1) Gecko/20100101 Firefox/4.0.1',
            'Mozilla/5.0 (Windows NT 6.1; rv,2.0.1) Gecko/20100101 Firefox/4.0.1'
        ]

    # 发送请求之前对 Request 请求对象进行处理，设置请求头，所以要重写 process_request 方法
    def process_request(self,request,spider):
        user_agent = random.choice(self.USER_AGENTS)
        request.headers['User-Agent'] = user_agent
```

使用 scrapy 自带的 UserAgentMiddleware

使用scrapy.downloadermiddlewares.useragent.UserAgentMiddleware. 可以在核心目录中创建新的py文件，用于写特定的中间件. 如 创建user_agent_middleware.py

把 scapy 文档中的
class scrapy.downloadermiddlewares.useragent.UserAgentMiddleware
修改为
from scrapy.downloadermiddlewares.useragent **import** UserAgentMiddleware

"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapy\downloadermiddlewares\useragent.py"

```
"""Set User-Agent header per spider or use a default value from settings"""

from scrapy import signals


class UserAgentMiddleware(object):
    """This middleware allows spiders to override the user_agent"""

    def __init__(self, user_agent='Scrapy'):
        self.user_agent = user_agent

    @classmethod
    def from_crawler(cls, crawler):
        o = cls(crawler.settings['USER_AGENT'])
        crawler.signals.connect(o.spider_opened, signal=signals.spider_opened)
        return o

    def spider_opened(self, spider):
        self.user_agent = getattr(spider, 'user_agent', self.user_agent)
```

```
    def process_request(self, request, spider):
        if self.user_agent:
            request.headers.setdefault(b'User-Agent', self.user_agent)
```

```
# 下载中间件---用户代理池实现
import random
from scrapy.downloadermiddlewares.useragent import UserAgentMiddleware

class UADMiddleware(UserAgentMiddleware):
    def __init__(self, ua=''):
        self.user_agents = [
        'Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_8; en-us) AppleWebKit/534.50 (KHTML, like Gecko) Version/5.1 Safari/534.50',
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36',
        'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;',
        'Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)',
        'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv,2.0.1) Gecko/20100101 Firefox/4.0.1',
        'Mozilla/5.0 (Windows NT 6.1; rv,2.0.1) Gecko/20100101 Firefox/4.0.1'
    ]
    def process_request(self, request, spider):
        user_agent = random.choice(self.user_agents)
        print("当前的用户代理是:%s" % user_agent)
        request.headers.setdefault("User-Agent", user_agent)
        request.headers['User-Agent'] = user_agent
```

修改settings.py

打开自定义的下载中间件，并设置下载延时．下载中间件中设置的User-Agent会覆盖掉settings.py中设置的User-Agent.

```
ROBOTSTXT_OBEY = False

DOWNLOAD_DELAY = 1

# Override the default request headers:
DEFAULT_REQUEST_HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'Accept-Language': 'en',
  'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'
}

# Enable or disable downloader middlewares
# See http://scrapy.readthedocs.org/en/latest/topics/downloader-middleware.html
DOWNLOADER_MIDDLEWARES = {
    'useragent_demo.middlewares.UserAgentDownloadMiddleware': 543,
```

```
}
```

ip代理池中间件

新建ipproxy.py爬虫文件

```
# -*- coding: utf-8 -*-
import scrapy
import json

class IpproxySpider(scrapy.Spider):
    name = 'ipproxy'
    allowed_domains = ['httpbin.org']
    start_urls = ['http://httpbin.org/ip']

    def parse(self, response):
        origin = json.loads(response.text)['origin']
        print('=' * 30)
        print(origin)
        print('=' * 30)
        yield scrapy.Request(self.start_urls[0], dont_filter=True)
```

在middlewares.py中添加IPProxyDownloadMiddleware下载中间件

```
import random

# 使用付费的开放代理，快代理，每次提取多个代理，每个请求都使用随机的代理.
class IPProxyDownloadMiddleware(object):
    PROXIES = [
        "178.44.170.152:8080",
        "110.44.113.182:8080",
        "209.126.124.73:8888",
        "84.42.79.243:8080",
        "209.126.124.73:9280",
        "209.126.124.73:8891",
        "177.97.31.180:8080",
        "42.104.84.106:8080",
        "117.64.234.7:808",
        "103.76.199.166:8080"
    ]
    def process_request(self,request,spider):
        proxy = random.choice(self.PROXIES)
        # 把代理设置到请求头中
        request.meta['proxy'] = proxy

import base64

# 使用需要用户名和密码验证的独享代理. 快代理
```

```
class IPProxyDownloadMiddleware(object):
    def process_request(self,request,spider):
        proxy = '121.199.6.124:16816'
        request.meta['proxy'] = proxy
        # 用户名和密码，也需要使用冒号进行分割
        user_password = "970138074:rcdj35ur"
        # 需要对用户名和密码进行 base64 的编码转换. b64encode 需要传递 bytes 的数据类型，所以需要使用 encode 来进行编码
        b64_user_password = base64.b64encode(user_password.encode('utf-8'))
        # 设置请求头. 因为 b64_user_password 是 bytes 的数据类型，而 'Basic ' 是字符串，所以需要把 b64_user_password 转换为字符串才行
        request.headers['Proxy-Authorization'] = 'Basic ' + b64_user_password.decode('utf-8')
```

```
import base64

# Start your middleware class
class ProxyMiddleware(object):
    # overwrite process request
    def process_request(self, request, spider):
        # Set the location of the proxy
        request.meta['proxy'] = "http://x.botproxy.net:8080"

        # Use the following lines if your proxy requires authentication
        auth_creds = "USERNAME:PASSWORD"
        # setup basic authentication for the proxy
        access_token = base64.encodestring(auth_creds)
        request.headers['Proxy-Authorization'] = 'Basic ' + access_token
```

```
# Importing base64 library because we'll need it ONLY in case if the proxy we are going to use requires authentication
import base64

# Start your middleware class
class ProxyMiddleware(object):
    # overwrite process request
    def process_request(self, request, spider):
        # Set the location of the proxy
        request.meta['proxy'] = "http://YOUR_PROXY:PORT"

        # Use the following lines if your proxy requires authentication
        proxy_user_pass = "USERNAME:PASSWORD"
        # setup basic authentication for the proxy
        encoded_user_pass = base64.encodestring(proxy_user_pass)
        request.headers['Proxy-Authorization'] = 'Basic ' + encoded_user_pass
```

在settings.py中启动下载中间件

```
DOWNLOADER_MIDDLEWARES = {
```

```
    'useragent_demo.middlewares.UserAgentDownloadMiddleware': 543,
    'useragent_demo.middlewares.IPProxyDownloadMiddleware': 100,
}
```

在项目根目录中新建start.py

```
#encoding: utf-8

from scrapy import cmdline

# cmdline.execute("scrapy crawl httpbin".split())
cmdline.execute("scrapy crawl ipproxy".split())
```

运行爬虫，进行测试

西刺免费代理爬虫

proxy spider

```
import redis
import telnetlib
import urllib.request
from bs4 import BeautifulSoup

r = redis.Redis(host='127.0.0.1', port=6379)

for d in range(1, 3):    #  采集 1 到 2 页
        scrapeUrl = 'http://www.xicidaili.com/nn/%d/' % d
        req = urllib.request.Request(scrapeUrl)
        req.add_header('User-Agent', 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)')
        response = urllib.request.urlopen(req)
        html = response.read()

        bsObj = BeautifulSoup(html, "html.parser")

        for i in range(100):
                speed = float(bsObj.select('td')[6 + i * 10].div.get('title').replace('秒', ''))
                if speed < 0.2:    #  验证速度，只要速度在 0.2 秒之内的
                        ip = bsObj.select('td')[1 + i * 10].get_text()
                        port = bsObj.select('td')[2 + i * 10].get_text()
                        ip_address = 'http://' + ip + ':' + port
                        try:
                                telnetlib.Telnet(ip, port=port, timeout=2)    #  用 telnet 对 ip 进行验证
                        except:
                                print('fail')
                        else:
                                print('sucess：' + ip_address)
```

```
                r.sadd('ippool', ip_address)    #  可用的 ip 导入到 redis
                f = open('proxy_list.txt', 'a')
                f.write(ip_address + '\n')
                f.close()
```

httpbin测试

httpbin.py

```
# -*- coding: utf-8 -*-
import scrapy


class HttpbinSpider(scrapy.Spider):
    name = 'httpbin'
    allowed_domains = ['httpbin.org']
    start_urls = ['http://httpbin.org/get']

    def parse(self, response):
        print(response.text)
        print(response.status)
```

middlewares.py

```
import random

class RandomUA():
    def __init__(self):
        self.user_agent = [
            'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_1) AppleWebKit/605.1.15 (KHTML,
like Gecko) Version/12.0.1 Safari/605.1.16',
            'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:12.0) Gecko/20100101 Firefox/12.0',
            'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)'
        ]

    def process_request(self, request, spider):
        request.headers['User-Agent'] = random.choice(self.user_agent)

    def process_response(self, request, response, spider):
        response.status = 201
        return response


class ProxyMiddleware:
    proxy_list = [
        "http://124.243.226.18:8888",
    ]

    def process_request(self, request, spider):
        ip = random.choice(self.proxy_list)
        request.meta['proxy'] = ip
```

settings.py

```
BOT_NAME = 'downloadertest'

SPIDER_MODULES = ['downloadertest.spiders']
NEWSPIDER_MODULE = 'downloadertest.spiders'

DOWNLOADER_MIDDLEWARES = {
    'downloadertest.middlewares.RandomUA': 543,
    'downloadertest.middlewares.ProxyMiddleware': 542,
}
```

下载中间件实战案例-Boss直聘爬虫

爬取全国 python 的招聘信息
BOSS直聘有很高的反爬虫机制，只要用同一个ip访问多个职位列表页，就会被封掉ip. 采用代理ip的方式可解决问题.

新建crawl爬虫

scrapy startproject boss_zhipin
cd boss_zhipin
scrapy genspider -t crawl boss "zhipin.com"

修改items.py, 定义item类

```python
# -*- coding: utf-8 -*-

import scrapy


class BossItem(scrapy.Item):
    name = scrapy.Field()
    salary = scrapy.Field()
    city = scrapy.Field()
    work_years = scrapy.Field()
    education = scrapy.Field()
    company = scrapy.Field()
```

修改boss.py爬虫文件

```python
# -*- coding: utf-8 -*-
import scrapy
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule
```

```
from boss.items import BossItem


class ZhipinSpider(CrawlSpider):
    name = 'zhipin'
    allowed_domains = ['zhipin.com']
    start_urls = ['https://www.zhipin.com/c100010000/h_100010000/?query=python&page=1']

    rules = (
        # 匹配职位列表页的规则. 不定义 callback, 会默认使用 parse 函数进行解析
        Rule(LinkExtractor(allow=r'.+\?query=python&page=\d'), follow=True),
        # 匹配职位详情页的规则. 因为符合要求的职位详情页都可以从列表页中提取到, 所以不需要对详情页中符合规则的职位进行跟进. 如果是一个全站的爬虫, 可以设置 follow=True
        Rule(LinkExtractor(allow=r'.+job_detail/\d+.html'),callback="parse_job", follow=False)
    )

    # 解析详情页, 从中提取职位信息. 可以使用 scrapy shell 来进行测试.
    def parse_job(self, response):
        name = response.xpath("//h1[@class='name']/text()").get().strip()
        salary = response.xpath("//h1[@class='name']/span/text()").get().strip()
        job_info = response.xpath("//div[@class='job-primary']/div[@class='info-primary']/p//text()").getall()
        city = job_info[0]
        work_years = job_info[1]
        education = job_info[2]
        company = response.xpath("//div[@class='info-company']//a/text()").get()
        item = BossItem(
            name=name,
            salary=salary,
            city=city,
            work_years=work_years,
            education=education,
            company=company
            )
        yield item
```

修改pipelines.py, 把数据保存到json文件中.

```
# -*- coding: utf-8 -*-

from scrapy.exporters import JsonLinesItemExporter

class BossPipeline(object):
    def __init__(self):
        # 以 bytes 格式写入数据, 要以 wb 方式打开文件
        self.fp = open('jobs.json','wb')
        self.exporter = JsonLinesItemExporter(self.fp,ensure_ascii=False)

    def process_item(self, item, spider):
        self.exporter.export_item(item)
        return item
```

```
    def close_spider(self,spider):
        self.fp.close()
```

修改settings.py, 激活BossPipeline

```
# Obey robots.txt rules
ROBOTSTXT_OBEY = False

DOWNLOAD_DELAY = 1

# Override the default request headers:
DEFAULT_REQUEST_HEADERS = {
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
   'Accept-Language': 'en',
   "User-Agent":"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/41.0.2228.0 Safari/537.36"
}

ITEM_PIPELINES = {
    'boss.pipelines.BossPipeline': 300,
}
```

修改modles.py, 对代理ip地址进行处理

Version 1

```
# encoding: utf-8
from datetime import datetime, timedelta


# 单独定义一个类来对 ip 地址进行处理
class ProxyModel(object):

    def __init__(self, data):
        self.ip = data['ip']
        self.port = data['port']
        # 字符串格式的过期时间, 格式为 "2017-12-19 22:10:10"
        self.expire_str = data['expire_time']
        # ip 地址是否被封, 默认没有被封
        self.blacked = False

        # 提取出过期日期和时间
        # 这里不使用 datetime.striptime 这个方法, 因为这个方法是使用 python 实现的,
执行效果不高, 所以先从 expire_str 中提取出过期的日期和时间, 再构建出 expire_time.
这样执行效果会高一些
        # datetime.striptime(expire_str)
        date_str, time_str = self.expire_str.split(" ")
```

```
            year, month, day = date_str.split("-")
            hour, minute, second = time_str.split(":")
            self.expire_time = datetime(year=int(year), month=int(month), day=int(day),
hour=int(hour), minute=int(minute),
                                        second=int(second))

            # 组装成 "https://ip:port" 的格式
            self.proxy = "https://{}:{}".format(self.ip, self.port)

    # 使用@property 对 is_expiring() 进行装饰, 就可以把 is_expiring 这个方法变成属性, 以后就可以像使用属性那样使用这个方法. 如 model = ProxyModel([]), 就可以像使用属性那样直接调用 is_expiring 这个方法了, model.is_expiring, 就不需要写后面的() 了.
    @property
    def is_expiring(self):
            # 判断代理地址是否即将过期. 假如代理的过期时间为 5-25 分钟, 在代理即将过期时, 要即时更换代理.
            now = datetime.now()
            if (self.expire_time - now) < timedelta(seconds=5):
                return True
            else:
                return False
```

models.py version2

```
#encoding: utf-8
from datetime import datetime, timedelta


# 单独定义一个类来对 ip 地址进行处理
class ProxyModel(object):
    EXPIRING_DELAY = 5

    def __init__(self,proxy_dict):
        self.ip = proxy_dict['ip']
        self.port = proxy_dict['port']
        # 字符串格式的过期时间, 格式为 "2017-12-19 22:10:10"
        self.expire_str = proxy_dict['expire_time']
        # 组装成 "https://ip:port" 的格式
        self.proxy = "https://{}:{}".format(self.ip,self.port)
        # ip 地址是否被封, 默认没有被封
        self.blacked = False

    # 使用@property 对 expire_time() 进行装饰, 就可以把 expire_time 这个方法变成属性, 以后就可以像使用属性那样使用这个方法. 如 model = ProxyModel([]), 就可以像使用属性那样直接调用 expire_time 这个方法了, model.expire_time, 就不需要写后面的() 了.
    @property
    def expire_time(self):
        # 提取出过期日期和时间
```

```
        # 这里不使用 datetime.striptime 这个方法，因为这个方法是使用python 实现的，执行效果不高，所以先从 expire_str 中提取出过期的日期和时间，再构建出 expire_time. 这样执行效果会高一些
        # datetime.striptime(expire_str)
        date_str,time_str = self.expire_str.split(" ")
        year,month,day = date_str.split("-")
        hour,minute,second = time_str.split(":")
        date_time = datetime(int(year),int(month),int(day),int(hour),int(minute),int(second))
        return date_time

    # 判断代理地址是否即将过期. 假如代理的过期时间为5-25 分钟，在代理即将过期时，要即时更换代理.
    @property
    def is_expiring(self):
        now = datetime.now()
        # 如果在 10 秒钟后即将过期
        if (self.expire_time - now) < timedelta(seconds=self.EXPIRING_DELAY):
            return True
        else:
            return False
```

另一种解决思路，不对是否过期进行判断，只是去发送请求，如果请求失败，就表示已经过期，就更新请求. 使用 try…except…来捕获异常，在 except 中更新代理.

修改middlewares.py， 添加随机请求头和随机ip地址

```
# -*- coding: utf-8 -*-

# Define here the models for your spider middleware
#
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/spider-middleware.html
import random
import requests
import json
from boss.models import ProxyModel
from twisted.internet.defer import DeferredLock

class UserAgentDownloadMiddleware(object):
    USER_AGENTS = [
        "Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2228.0 Safari/537.36",
        "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2227.1 Safari/537.36",
        "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2227.0 Safari/537.36",
        "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2227.0 Safari/537.36",
        "Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2226.0 Safari/537.36",
```

```
            "Mozilla/5.0 (Windows NT 6.4; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/41.0.2225.0 Safari/537.36",
            "Opera/9.80 (X11; Linux i686; Ubuntu/14.10) Presto/2.12.388 Version/12.16",
            "Opera/9.80 (Windows NT 6.0) Presto/2.12.388 Version/12.14",
            "Mozilla/5.0 (Windows NT 6.0; rv:2.0) Gecko/20100101 Firefox/4.0 Opera 12.14",
            "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.0) Opera 12.14",
            "Opera/12.80 (Windows NT 5.1; U; en) Presto/2.10.289 Version/12.02",
            "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:40.0) Gecko/20100101 Firefox/40.1",
            "Mozilla/5.0 (Windows NT 6.3; rv:36.0) Gecko/20100101 Firefox/36.0",
            "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10; rv:33.0) Gecko/20100101 Firefox/33.0",
            "Mozilla/5.0 (X11; Linux i586; rv:31.0) Gecko/20100101 Firefox/31.0",
            "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:31.0) Gecko/20130401 Firefox/31.0",
            "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_3) AppleWebKit/537.75.14 (KHTML, like
Gecko) Version/7.0.3 Safari/7046A194A",
            "Mozilla/5.0 (iPad; CPU OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like
Gecko) Version/6.0 Mobile/10A5355d Safari/8536.25",
            "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/537.13+ (KHTML, like
Gecko) Version/5.1.7 Safari/534.57.2",
            "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/534.55.3 (KHTML, like
Gecko) Version/5.1.3 Safari/534.53.10",
            "Mozilla/5.0 (iPad; CPU OS 5_1 like Mac OS X) AppleWebKit/534.46 (KHTML, like
Gecko ) Version/5.1 Mobile/9B176 Safari/7534.48.3",
        ]
        def process_request(self,request,spider):
            user_agent = random.choice(self.USER_AGENTS)
            request.headers['User-Agent'] = user_agent


# 在 ip 代理地址被封禁时随机切换代理地址. 为什么在 UserAgentDownloadMiddleware 中只需
要重写 process_request 一个方法, 而在此处要重写 process_request 和 process_response 两个方
法呢.
# 因为这里要在 ip 代理地址被封时要切换代理地址, 什么情况下 ip 代理失效呢. 进行判断, 如
果在浏览器访问时出现了 403 的错误, 就需要更换 ip 地址. 返回的 403 的错误是在 response 中
的, 所以要通过重写 process_response 方法, 在这个方法中判断 ip 地址是否已经失效.
# 重写 process_request 方法, 在每次运行爬虫时, 都设置一个新的代理地址.
# 只要爬虫开始正常运行, 就不要在每次请求都使用一个新的代理地址, 只需要在代理失效时
切换一个新的 ip 地址.
class IPProxyDownloadMiddleware(object):

    # 芝麻代理, 按使用次数购买. 使用 json 格式, https, 生成过期时间.
    PROXY_URL =
'http://webapi.http.zhimacangku.com/getip?num=1&type=2&pro=&city=0&yys=0&port=11&time=1
&ts=1&ys=0&cs=0&lb=1&sb=0&pb=45&mr=1&regions='

    def __init__(self):
        super(IPProxyDownloadMiddleware, self).__init__()
        # 初始化类时 current_proxy = None, 即没有代理
        self.current_proxy = None
        # 创建一把锁
        self.lock = DeferredLock()

    # 处理请求对象, 添加或更新代理
```

```
    def process_request(self,request,spider):
        # 判断 request 中是否使用了代理，如果没有代理或者当前的代理即将过期了，就调用 update_proxy，重新生成新的代理.
        if 'proxy' not in request.meta or self.current_proxy.is_expiring:
            # 请求代理
            self.update_proxy()
        # 从 current_proxy 中取出来 proxy，并赋值给 request.meta['proxy']
        request.meta['proxy'] = self.current_proxy.proxy
        ### ????? 这里是否需要 return request，不需要，如果返回一个新的 request 对象，就使用这个新的 request 来发送请求.
        # return request

    #### 进一步改进，应该把所有与切换 ip 相关的功能如加入黑名单，检验是否可用，更新 ip 地址等都放到一个单独的文件中. 还可以把代理地址放到 mysql 数据库或 redis 数据库中.
    # 根据 response 的响应来判断代理地址是否被封
    def process_response(self,request,response,spider):
        # 如果状态码不是 200，或者出现了验证码，就更新代理地址，重新发送 request. 验证码的状态码也是 200，所以也要加上验证码的判断
        if response.status != 200 or "captcha" in response.url:
            # 假如多个线程同时判断到代理 ip 被加入了黑名单，所有线程都会去执行把 ip 地址加入黑名单和 update_proxy 更新代理的操作，所以此时就要进行判断，如果当前使用的代理地址没有被加入黑名单，才去执行加入黑名单的操作，如果当前使用的代理地址已经被加入黑名单，就不再进行重复的操作.
            # 如果一个线程首先把 current_proxy.blacked 设置为 True，然后执行 update_proxy 方法来获取新的代理地址，由于获取的新的代理地址中 blacked 为 False，即没有加入黑名单，所以下一个线程还会执行把 current_proxy.blacked 设置为 True，然后执行 update_proxy 的操作. 所以这里要进行判断，判断当前的代理地址是否被加入了黑名单，如果没有被加入黑名单，才执行加入黑名单的操作. 这样就能避免出现多个线程同时设置 current_proxy.blacked 为 True，同时进行 update_proxy 的操作
            # 这个地方有问题呀，因为 python 动态语言的特性，新生成的 ip 地址还是满足 if not self.current_proxy.black，所以即使生成了新的代理地址，被封的线程中的下一个线程还会去执行更新代理的操作. 所以需要把 blcked 字段设置为字符串，新生成的设置为空 ''，验证未被封的设置为 '0'，验证被封的设置为 '1'，这里的条件设置为 if self.current_proxy.blacked = '0'
            # if self.current_proxy.blacked == False
            if not self.current_proxy.blacked:
                self.current_proxy.blacked = True
                print('%s 这个代理被加入黑名单了'%self.current_proxy.ip)
                self.update_proxy()
            # 如果来到这里，说明这个请求已经被 boss 直聘识别为爬虫了，response 响应中只有 304 的错误信息，没有我们想要的职位信息，所以还要重新返回 request，让这个请求重新加入到调度中，下次再次发送请求. 下次再发送请求时，请求还会经过 process_request 进行处理，因为已经调用了 update_proxy 方法生成了新的代理，所以请求会使用新的代理，就不会被识别为爬虫了，就可以返回正常的信息了.
            # 如果不返回 request，那么就相当于跳过了与这个 request 请求对应的 url 地址.
            return request
        # 如果响应是正常的，那么就把 response 返回到 spider 中进行处理.
        # 如果不返回，那么这个 resposne 就不会被传到爬虫那里去，也就得不到解析
        return response
```

```
    # 定义一个单独的方法，更新代理地址
    def update_proxy(self):
        # 因为scrapy是基于twisted的异步请求框架，可以理解为多线程爬虫，会使用多个线程爬取数据. 所以如果ip地址失效了，可能会有多个线程的爬虫同时返回403的响应，多个线程同时调用 update_proxy 更新代理. 由于代理的获取有一定的时间限制，如果在很短的时间内多次向代理地址请求获取代理，就会返回请求太频繁的错误，所以要进行加锁的操作. 在一个线程在进行更新代理的操作时，加上锁，其它的地址都处理等待状态中，这样就不会出现多线程同时更新代理的情况了.
        self.lock.acquire()
        # 当上一个进程上锁并更新代理地址之后，只要这个锁被释放掉，下一个进程还会继续执行加锁更新代理地址的操作，所以这里要进行判断，如果self中没有 current_proxy，或者 current_proxy 即将过期，或者 current_proxy 已经被封掉，才去执行更新代理地址的操作
        if not self.current_proxy or self.current_proxy.is_expiring or self.current_proxy.blacked:
            response = requests.get(self.PROXY_URL)

            text = response.text
            print('重新获取了一个代理: ',text)
            # 返回的是json格式，即text为json格式的字符串，要先转换为字典，再从中提取出代理地址. 每次取出1个ip地址.
            result = json.loads(text)
            # 因为代理的获取有一定的时间限制，如果很短时间内多次向代理地址请求获取代理，就会返回请求太频繁的错误. 所以需要进行判断，如果result['data']大于0，说明返回了ip地址，否则没有返回正确的ip地址.
            if len(result['data']) > 0:
                data = result['data'][0]
                # 因为需要对ip地址进行比较多的操作，所以单独定义一个新的模型，来进行ip地址的处理，如提取ip地址，判断是否已经过期.
                proxy_model = ProxyModel(data)
                # 设置当前的代理为 proxy_model, proxy_model 中不仅包含着代理ip和端口，还包含着过期时间
                self.current_proxy = proxy_model
        self.lock.release()
```

修改settings.py，启动middleware

```
# Obey robots.txt rules
ROBOTSTXT_OBEY = False

# Configure a delay for requests for the same website (default: 0)
# See http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay
# See also autothrottle settings and docs
DOWNLOAD_DELAY = 1

# Override the default request headers:
DEFAULT_REQUEST_HEADERS = {
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
   'Accept-Language': 'en',
```

```
    "User-Agent":"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2228.0 Safari/537.36"
}

# Enable or disable downloader middlewares
# See http://scrapy.readthedocs.org/en/latest/topics/downloader-middleware.html
DOWNLOADER_MIDDLEWARES = {
    'boss.middlewares.UserAgentDownloadMiddleware': 100,
    'boss.middlewares.IPProxyDownloadMiddleware': 200,
}
# Configure item pipelines
# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html
ITEM_PIPELINES = {
    'boss.pipelines.BossPipeline': 300,
}
```

在项目根目录新建 start.py

```
#encoding: utf-8

from scrapy import cmdline

cmdline.execute("scrapy crawl zhipin".split())
```

5.8. settings 常用配置信息

Scrapy 设置(settings)提供了定制 Scrapy 组件的方法. 可以控制包括核心(core), 插件(extension), pipeline 及 spider 组件. 比如 设置 Json Pipeliine、LOG_LEVEL 等.

参考文档: http://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/settings.html#topics-settings-ref

常用设置

BOT_NAME

默认: 'scrapybot'
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.

```
# Obey robots.txt rules
ROBOTSTXT_OBEY = True

# 默认最大同时并发量
# Configure maximum concurrent requests performed by Scrapy (default: 16)
#CONCURRENT_REQUESTS = 32
```

DEFAULT_REQUEST_HEADERS

默认: 如下
{
'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.
其它地方都没有写的时候才会使用 settings 中的设置, 如果其它地方定义了, 则使用另外自定义的.

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.

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
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.

DOWNLOAD_TIMEOUT

默认: 180
下载器超时时间(单位: 秒).

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)习惯设置在 0-1000 范围内, 值越小优先级越高.
ITEM_PIPELINES = {
'mySpider.pipelines.SomethingPipeline': 300,

'mySpider.pipelines.ItcastJsonPipeline': 800,
}

LOG_ENABLED

默认: True
是否启用 logging.

LOG_ENCODING

默认: 'utf-8'
logging 使用的编码.

LOG_LEVEL

默认: 'DEBUG'
log 的最低级别. 可选的级别有: CRITICAL、 ERROR、WARNING、INFO、DEBUG .
调试的时候一般写 debug 信息. 后期运维的话一般设置为 info 信息.

USER_AGENT

默认: "Scrapy/VERSION (+http://scrapy.org)"
爬取的默认 User-Agent, 除非被覆盖.

PROXIES

代理设置
示例:

```
PROXIES = [
   {'ip_port': '111.11.228.75:80', 'password': ''},
   {'ip_port': '120.198.243.22:80', 'password': ''},
   {'ip_port': '111.8.60.9:8123', 'password': ''},
   {'ip_port': '101.71.27.120:80', 'password': ''},
   {'ip_port': '122.96.59.104:80', 'password': ''},
   {'ip_port': '122.224.249.122:8088', 'password':''},
]
```

```
# -*- coding: utf-8 -*-

# Scrapy settings for downloadertest project
#
# For simplicity, this file contains only settings considered important or
# commonly used. You can find more settings consulting the documentation:
#
#         https://doc.scrapy.org/en/latest/topics/settings.html
```

```
#         https://doc.scrapy.org/en/latest/topics/downloader-middleware.html
#         https://doc.scrapy.org/en/latest/topics/spider-middleware.html

BOT_NAME = 'downloadertest'

SPIDER_MODULES = ['downloadertest.spiders']
NEWSPIDER_MODULE = 'downloadertest.spiders'


# Crawl responsibly by identifying yourself (and your website) on the user-agent
#USER_AGENT = 'downloadertest (+http://www.yourdomain.com)'
#USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_1)
AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0.1 Safari/605.1.15'

# Obey robots.txt rules
ROBOTSTXT_OBEY = True

# 默认最大同时并发量
# Configure maximum concurrent requests performed by Scrapy (default: 16)
#CONCURRENT_REQUESTS = 32

# 对同一个网站默认的爬取间隔
# Configure a delay for requests for the same website (default: 0)
# See https://doc.scrapy.org/en/latest/topics/settings.html#download-delay
# See also autothrottle settings and docs
# DOWNLOAD_DELAY = 3
# The download delay setting will honor only one of:
# 每个域名下默认的并发量，高于 CONCURRENT_REQUESTS
# CONCURRENT_REQUESTS_PER_DOMAIN = 16
# 每个 IP 最高的并发量
#CONCURRENT_REQUESTS_PER_IP = 16

# 自动限速功能，默认是关闭的
# Enable and configure the AutoThrottle extension (disabled by default)
# See https://doc.scrapy.org/en/latest/topics/autothrottle.html
# AUTOTHROTTLE_ENABLED = True

# 初始链接延迟
# The initial download delay
# AUTOTHROTTLE_START_DELAY = 5

# 最大延迟时间
# The maximum download delay to be set in case of high latencies
# AUTOTHROTTLE_MAX_DELAY = 60
```

```
# 向同一个服务器发送的并行的连接数
# The average number of requests Scrapy should be sending in parallel to
# each remote server
# AUTOTHROTTLE_TARGET_CONCURRENCY = 1.0
# Enable showing throttling stats for every response received:
 #AUTOTHROTTLE_DEBUG = False

# Http 缓存
# Enable and configure HTTP caching (disabled by default)
# See https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#httpcache-middleware-settings
#HTTPCACHE_ENABLED = True
#HTTPCACHE_EXPIRATION_SECS = 0
#HTTPCACHE_DIR = 'httpcache'
#HTTPCACHE_IGNORE_HTTP_CODES = []
#HTTPCACHE_STORAGE = 'scrapy.extensions.httpcache.FilesystemCacheStorage'
```

1. BOT_NAME: 项目名称.
2. ROBOTSTXT_OBEY: 是否遵守爬虫协议. 默认不遵守.
3. CONCURRENT_ITEMS: 代表 pipeline 同时处理的 item 数的最大值. 默认是 100
4. CONCURRENT_REQUESTS: 爬虫的并发量, 可以同时下载的数据. 代表下载器并发请求的最大值, 默认是 16.
5. DOWNLOAD_DELAY: 下载延迟, 与 time.sleep 功能类似. 下载器在下载某个页面前等待多长的时间. 该选项用来限制爬虫的爬取速度, 减轻服务器压力. 同时也支持小数.
6. COOKIES_ENABLED: 是否开启 cookies. 一般不要开启, 避免爬虫被追踪到. 如果特殊情况如模拟登录时就必须要开启 cookies.
7. DEFAULT_REQUEST_HEADERS: 默认请求头. 可以将一些不会经常变化的请求头放在这个里面.
8. DEPTH_LIMIT: 爬取网站最大允许的深度. 默认为 0, 如果为 0, 则没有限制.
9. DOWNLOAD_TIMEOUT: 下载器下载的超时时间.
10. SPIDER_MIDDLEWARES: 爬虫中间件, 一般用的不多.
11. DOWNLOADER_MIDDLEWARES: 下载中间件.
12. ITEM_PIPELINES: 管道文件, 决定了下载的数据(item)如何进行处理. ITEM_PIPELINES 是一个字典, 字典的 key 这个 pipeline 所在包的绝对路径, 值是一个整数, 表示优先级, 取值范围为 1~1000, 值越小对应的 Pipline 优先级就越高.
13. LOG_ENABLED: 是否启用 logging. 默认是 True.
14. LOG_ENCODING: log 的编码.
15. LOG_LEVEL: log 的级别. 默认为 DEBUG. 可选的级别有 CRITICAL, ERROR, WARNING, INFO, DEBUG.
16. USER_AGENT: 请求头. 默认为 Scrapy/VERSION (+http://scrapy.org).
17. PROXIES: 代理设置.

```
BOT_NAME = 'mySpider'

SPIDER_MODULES = ['mySpider.spiders']
```

```
NEWSPIDER_MODULE = 'mySpider.spiders'

# Crawl responsibly by identifying yourself (and your website) on the user-agent
#USER_AGENT = 'mySpider (+http://www.yourdomain.com)'

# Obey robots.txt rules
#是否遵循 robots 协议
ROBOTSTXT_OBEY = False

# Configure maximum concurrent requests performed by Scrapy (default: 16)
#爬虫的并发量，可以同时下载的数据
CONCURRENT_REQUESTS = 16

# Configure a delay for requests for the same website (default: 0)
# See http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay
# See also autothrottle settings and docs

# 下载延迟，与 time.sleep 功能类似
DOWNLOAD_DELAY = 1

# The download delay setting will honor only one of:
CONCURRENT_REQUESTS_PER_DOMAIN = 16
CONCURRENT_REQUESTS_PER_IP = 16

# Disable cookies (enabled by default)
#是否启动 cookie，模拟登录时要打开，使用爬虫爬取数据时就不打开，服务器就不会根据
cookie 来检测我们爬取的内容了
COOKIES_ENABLED = False

# Disable Telnet Console (enabled by default)
#TELNETCONSOLE_ENABLED = False

# Override the default request headers:
#默认的请求报头，可以在这里设置 user-agent，但如果网站对请求头进行了限制，就要在下载
中间件中设置随机的请求头.
DEFAULT_REQUEST_HEADERS = {
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
只显示英文的选项关闭.
   'Accept-Language': 'en',
}

# Enable or disable spider middlewares
# See http://scrapy.readthedocs.org/en/latest/topics/spider-middleware.html
#爬虫的中间件一般不用，下载的中间件用的较多
SPIDER_MIDDLEWARES = {
    'mySpider.middlewares.MyspiderSpiderMiddleware': 543,
}

# Enable or disable downloader middlewares
# See http://scrapy.readthedocs.org/en/latest/topics/downloader-middleware.html
# 下载中间件, 543 是优先级, 1~1000，数字越小就优先级越高，可以设置多个，根据优先级进行
选择
```

```
DOWNLOADER_MIDDLEWARES = {
    'mySpider.middlewares.MyCustomDownloaderMiddleware': 543,
}

# Enable or disable extensions
# See http://scrapy.readthedocs.org/en/latest/topics/extensions.html
#EXTENSIONS = {
#       'scrapy.extensions.telnet.TelnetConsole': None,
#}

# Configure item pipelines
# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html
#管道文件，决定了下载的数据如何进行处理
ITEM_PIPELINES = {
    'mySpider.pipelines.MyspiderPipeline': 300,
}
```

5.9. Scrapy 爬虫实战

简书网站整站爬虫.

要求

数据保存到mysql数据库中.

将selenium+chromedriver集成到scrapy.

需求分析

爬取所有文章的内容，包括标题，作者，头像，发表时间，字数

阅读量，评论数量，喜欢的人数，以及收录本文章的专题，推荐阅读都是通过 ajax 的方式加载的，所以可以把 selinium 和 chrome driver 整合在 scrapy 项目中

Elements 中的内容是在内存中的，是经过网页渲染后的内容，整合了网页中所有的图片, js 和 html 的内容. 而查看源码中看到的是服务器对本网页响应的内容，只包含了本响应的内容，不包含图片, js 动态加载的内容等.

新建crawl spider爬虫

scrapy startproject jianshu_spider
scrapy genspider -t crawl jianshu "jianshu.com"

修改items.py，定义要爬取的字段

```python
# -*- coding: utf-8 -*-

import scrapy

class ArticleItem(scrapy.Item):
    # 标题
    title = scrapy.Field()
    # 内容
    content = scrapy.Field()
    # 文章 id
    article_id = scrapy.Field()
    # 文章 url
    origin_url = scrapy.Field()
    # 作者
    author = scrapy.Field()
    # 作者头像
    avatar = scrapy.Field()
    # 发表时间
    pub_time = scrapy.Field()
    # 阅读数
    read_count = scrapy.Field()
    # 喜欢数
    like_count = scrapy.Field()
    # 字数
    word_count = scrapy.Field()
    # 评论数
    comment_count = scrapy.Field()
    # 所属专题
    subjects = scrapy.Field()
```

修改jianshu.py，定义爬虫的内容

```python
# -*- coding: utf-8 -*-
import scrapy
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule
from jianshu_spider.items import ArticleItem


class JsSpider(CrawlSpider):
    name = 'jianshu'
    allowed_domains = ['jianshu.com']
    start_urls = ['https://www.jianshu.com/']
```

```
    # 提取出所有文章详情页的 url 地址，因为是整站爬虫，所以需要在文章详情页中对链接进行跟进，把文章详情页符合规则的 url 地址也提取出来.
    # 注意元组中如果有一个元素，后面必须要有逗号
    rules = (
        Rule(LinkExtractor(allow=r'.*/p/[0-9a-z]{12}.*'), callback='parse_detail', follow=True),
    )

    def parse_detail(self, response):
        title = response.xpath("//h1[@class='title']/text()").get()
        avatar = response.xpath("//a[@class='avatar']/img/@src").get()
        author = response.xpath("//span[@class='name']/a/text()").get()
        pub_time = response.xpath("//span[@class='publish-time']/text()").get().replace("*","")
        # 从 url 中提取出文章 id
        # https://www.jianshu.com/p/d30d0f91554a?utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation
        # https://www.jianshu.com/p/d30d0f91554a
        url = response.url
        # ['https://www.jianshu.com/p/d30d0f91554a','utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation']
        # ['https://www.jianshu.com/p/d30d0f91554a']
        # 以问号对 url 进行分隔，在正常的 url 中，最多只有一个问号. 后面跟参数.
        url1 = url.split("?")[0]
        article_id = url1.split('/')[-1]

        # 爬取到 content 即网站的内容后，一个需求可能是把爬取到的内容放到自己的网站中. 所以文章内容中的所有标签都要保留. 所以这里不要再去调用//text()，否则会把所有的标签都删除，只留下文字内容.
        content = response.xpath("//div[@class='show-content']").get()

        item = ArticleItem(
            title=title,
            avatar=avatar,
            author = author,
            pub_time = pub_time,
            origin_url = response.url,
            article_id = article_id,
            content = content,
        )
        yield item
```

创建数据库和数据表

创建数据库 jianshu

创建数据表, article, 字段 id int not null primary key auto increament, title varchar 255, content varchar longtext, author varchar 255, avatar varchar 255, pub_time datetime, article_id varchar 20, arigin_url varchar 255

修改pipelines.py，把爬取到的数据保存到数据库中

```
# -*- coding: utf-8 -*-

import pymysql
from pymysql import cursors

class JianshuSpiderPipeline(object):
    def __init__(self):
        dbparams = {
            'host': '127.0.0.1',
            'port': 3306,
            'user': 'root',
            'password': 'root',
            'database': 'jianshu',
            # 注意 mysql 中所有相关的编码都是 utf8，不是 utf-8
            'charset': 'utf8'
        }
        # 创建连接对象. 查看 connect 的源码，确定需要传递的参数. **dbparams 会把字典中的 key 和 value 作为关键字参数传递过去.
        self.conn = pymysql.connect(**dbparams)
        # 创建 cursor 游标对象
        self.cursor = self.conn.cursor()
        self._sql = None

    def process_item(self, item, spider):
        # 把 item 中的内容通过 sql 语句保存到 mysql 中.
        self.cursor.execute(self.sql, (item['title'], item['content'], item['author'], item['avatar'], item['pub_time'], item['origin_url'], item['article_id']))
        self.conn.commit()
        return item

    @property
    def sql(self):
        # 如果之前没有生成过 sql 语句，就使用 self._sql 生成 sql 语句
        if not self._sql:
            # id 是自动增加的，所以这里可以写为 null
            self._sql = """
            insert into article(id, title, content, author, avatar, pub_time, origin_url, article_id) values(null, %s, %s, %s, %s, %s, %s, %s)
            """
            return self._sql
        # 如果已经生成了 sql 语句，就直接使用 self._sql
        return self._sql
```

修改settings.py

不遵守 robots 协议，修改默认的请求头，修改下载延迟，启用 JianshuSpiderPipeline

```
# Obey robots.txt rules
ROBOTSTXT_OBEY = False

# Configure a delay for requests for the same website (default: 0)
# See http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay
# See also autothrottle settings and docs
DOWNLOAD_DELAY = 3

# Override the default request headers:
DEFAULT_REQUEST_HEADERS = {
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
   'Accept-Language': 'en',
'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.108 Safari/537.36'
}

# Configure item pipelines
# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html
ITEM_PIPELINES = {
    'jianshu_spider.pipelines.JianshuSpiderPipeline': 300,
}
```

在项目根目录中新建start.py

```
#encoding: utf-8

from scrapy import cmdline

cmdline.execute("scrapy crawl js".split())
```

运行爬虫，就可以看到数据会逐渐保存到数据库中.

修改pipelines.py，使用Twisted把数据异步保存到数据库中

在 pipeline 中，把数据保存到 mysql 中的操作是同步的，io 操作一般都比较费时，效率会比较低，可以把保存到 mysql 中的操作修改为异步的，这样就能大大提高保存数据的效率.

```
# -*- coding: utf-8 -*-

import pymysql
```

```python
# adbapi 是专用用于做数据库处理的模块，在其中创建了一个连接池，使用连接池进行数据的保存
from twisted.enterprise import adbapi
from pymysql import cursors

class JianshuSpiderPipeline(object):
    def __init__(self):
        dbparams = {
            'host': '127.0.0.1',
            'port': 3306,
            'user': 'root',
            'password': 'root',
            'database': 'jianshu',
            # 注意 mysql 中所有相关的编码都是 utf8，不是 utf-8
            'charset': 'utf8'
        }
        # 创建连接对象。查看 connect 的源码，确定需要传递的参数。**dbparams 会把字典中的 key 和 value 作为关键字参数传递过去。
        self.conn = pymysql.connect(**dbparams)
        # 创建 cursor 游标对象
        self.cursor = self.conn.cursor()
        self._sql = None

    def process_item(self, item, spider):
        # 把 item 中的内容通过 sql 语句保存到 mysql 中。
        self.cursor.execute(self.sql, (item['title'], item['content'], item['author'], item['avatar'], item['pub_time'], item['origin_url'], item['article_id']))
        self.conn.commit()
        return item

    @property
    def sql(self):
        # 如果之前没有生成过 sql 语句，就使用 self._sql 生成 sql 语句
        if not self._sql:
            # id 是自动增加的，所以这里可以写为 null
            self._sql = """
            insert into article(id, title, content, author, avatar, pub_time, origin_url, article_id) values(null, %s, %s, %s, %s, %s, %s, %s)
            """
            return self._sql
        # 如果已经生成了 sql 语句，就直接使用 self._sql
        return self._sql


class JianshuTwistedPipeline(object):
    def __init__(self):
        dbparams = {
            'host': '127.0.0.1',
            'port': 3306,
            'user': 'root',
            'password': 'root',
            'database': 'jianshu2',
```

```
            'charset': 'utf8',
            # 还需要传入 cursorclass 这个参数，即指定使用的 cursor 的类
            'cursorclass': cursors.DictCursor
        }
        # 创建一个连接池对象 def __init__(self, dbapiName, *connargs, **connkw). 这里的 dbapiName 要传入数据库连接的模块, twisted 框架会在底层加载 pymysql 这个模块.
        self.dbpool = adbapi.ConnectionPool('pymysql', **dbparams)
        self._sql = None

    @property
    def sql(self):
        if not self._sql:
            self._sql = """
                insert into article(id, title, content, author, avatar, pub_time, origin_url, article_id) values(null, %s, %s, %s, %s, %s, %s, %s)
                """
            return self._sql
        return self._sql

    def process_item(self,item,spider):
        # 把数据插入到数据库中
        # 这里如果直接调用 cursor.execute 来插入数据，实现上与上面同步插入数据的方法是一样的，使用 dbpool.runInteraction 调用插入数据的方法，就会把插入数据库的操作变成是异步的.
        # 这里要使用 dbpool 中的方法来实现. 在 dbpool.runInteraction 中调用 self.insert_item 这个函数，所以向数据库中插入数据实际上是在 insert_item 这个方法中完成的.
        defer = self.dbpool.runInteraction(self.insert_item, item)
        # 添加错误处理方法，如果发生了错误，就会调用 handle_error() 方法进行错误处理.
        defer.addErrback(self.handle_error, item,spider)

    def insert_item(self,cursor,item):
        cursor.execute(self.sql, (item['title'], item['content'], item['author'], item['avatar'], item['pub_time'], item['origin_url'], item['article_id']))

    def handle_error(self,error,item,spider):
        print('='*10+"error"+'='*10)
        print(error)
        print('='*10+"error"+'='*10)
```

修改settings.py，启动新的JianshuTwistedPipeline

```
ITEM_PIPELINES = {
    # 'jianshu_spider.pipelines.JianshuSpiderPipeline': 300,
    'jianshu_spider.pipelines.JianshuTwistedPipeline': 300,
}
```

添加下载中间件，使用selenium提取异步加载的数据.

以上爬虫有一个问题，如果首页中的文章爬完，文章详情页中的推荐文章爬取完后，就不会再爬取到更多的内容了．所以要对 ajax 加载出来的内容进行爬取.

推荐阅读中的内容，除了第 1 个之外，其它的都是使用 ajax 方式加载出来的，专题中的内容，也是通过 ajax 加载出来的.

可以通过分析 ajax 请求找到对应的响应，也可以使用 selenium 的方式来获取.

修改数据库，添加几个字段
阅读量
read_count int default 0,
喜欢量
like_count int default 0,
字数
word_count int default 0
专题
subjects text

修改 middlewares.py

```
# -*- coding: utf-8 -*-

# Define here the models for your spider middleware
#
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/spider-middleware.html

from scrapy import signals
from selenium import webdriver
import time
from scrapy.http.response.html import HtmlResponse


class SeleniumDownloadMiddleware(object):
    def __init__(self):
        self.driver = webdriver.Chrome(executable_path =
r"D:\ProgramApp\chromedriver\chromedriver.exe")

    # 如果 process_request 中返回的是 response 对象, scrapy 就不会把请求发送给下载器下载,
而是直接把 response 对象发送给爬虫进行解析.
    # 可以在 process_request 中捕获 request，使其不经过 scrapy 发送下载请求，而是使
selenium 来发送请求，然后使用 return 把对应响应的内容返回给 spider 进行解析. 这样就把
selenium 集成到了 scrapy 中，使用 scrapy 对请求进行去重，对数据进行解析和保存，而使用
selenium 代替 scrapy 框架来发送请求，即把 scrapy 框架流程图中的 downloadmiddleware 替换为
自定义的，其它组件依旧使用 scrapy 自身的..
```

```
    def process_request(self, request, spider):
        self.driver.get(request.url)
        # 等待1s，以免数据还未加载完成就返回网页源码
        time.sleep(1)
        # 点击获取帖子所属的专题，循环进行点击，直到没有找到show-more这个元素.
        try:
            while True:
                showMore = self.driver.find_element_by_class_name('show-more')
                showMore.click()
                time.sleep(0.3)
                if not showMore:
                    break
        except:
            pass
        # 提取出网页源码，构造Response响应对象并返回
        source = self.driver.page_source
        # 查看HtmlResponse的源码确定需要传递的参数.
        # def __init__(self, url, status=200, headers=None, body=b'', flags=None, request=None):
        response = HtmlResponse(url=self.driver.current_url, body=source, request=request,
encoding='utf-8')
        return response
```

修改settings.py，启动下载中间件

```
DOWNLOADER_MIDDLEWARES = {
    'jianshu_spider.middlewares.SeleniumDownloadMiddleware': 543,
}
```

修改jianshu.py爬虫文件，提取其它字段的数据.

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule
from jianshu_spider.items import ArticleItem


class JsSpider(CrawlSpider):
    name = 'js'
    allowed_domains = ['jianshu.com']
    start_urls = ['https://www.jianshu.com/']

    # 提取出所有文章详情页的url地址，因为是整站爬虫，所以需要在文章详情页中对链接
进行跟进，把文章详情页符合规则的url地址也提取出来.
    # 注意元组中如果有一个元素，后面必须要有逗号
    rules = (
        Rule(LinkExtractor(allow=r'.*/p/[0-9a-z]{12}.*'), callback='parse_detail', follow=True),
    )
```

```
    def parse_detail(self, response):
        title = response.xpath("//h1[@class='title']/text()").get()
        avatar = response.xpath("//a[@class='avatar']/img/@src").get()
        author = response.xpath("//span[@class='name']/a/text()").get()
        pub_time = response.xpath("//span[@class='publish-time']/text()").get().replace("*","")
        # 从 url 中提取出文章 id
        # https://www.jianshu.com/p/d30d0f91554a?utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation
        # https://www.jianshu.com/p/d30d0f91554a
        url = response.url
        # ['https://www.jianshu.com/p/d30d0f91554a','utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation']
        # ['https://www.jianshu.com/p/d30d0f91554a']
        # 以问号对 url 进行分隔，在正常的 url 中，最多只有一个问号. 后面跟参数.
        url1 = url.split("?")[0]
        article_id = url1.split('/')[-1]

        # 爬取到 content 即网站的内容后，一个需求可能是把爬取到的内容放到自己的网站中. 所以文章内容中的所有标签都要保留. 所以这里不要再去调用//text()，否则会把所有的标签都删除，只留下文字内容.
        content = response.xpath("//div[@class='show-content']").get()

        word_count = response.xpath("//span[@class='wordage']/text()").get()
        comment_count = response.xpath("//span[@class='comments-count']/text()").get()
        read_count = response.xpath("//span[@class='views-count']/text()").get()
        like_count = response.xpath("//span[@class='likes-count']/text()").get()

        # 提取到的专题是一个列表，把列表传递给 mysql 时, mysql 会把列表转换为字符串型的列表. 如[1,2,3]转换为'[1,2,3]'，所以这里需要手动把列表转换为字符串
        subjects = ",".join(response.xpath("//div[@class='include-collection']/a/div/text()").getall())

        item = ArticleItem(
            title=title,
            avatar=avatar,
            author = author,
            pub_time = pub_time,
            origin_url = response.url,
            article_id = article_id,
            content = content,
            subjects = subjects,
            word_count = word_count,
            comment_count = comment_count,
            read_count = read_count,
            like_count = like_count
        )
        yield item
```

清空 mysql 数据库中的内容，再次运行爬虫，就会看到打开了一个新的网页来执行.

6. 第六章：Scrapy-Redis 分布式组件

6.1. redis 数据库介绍

概述

redis是一种支持分布式的nosql数据库,他的数据是保存在内存中，同时redis可以定时把内存数据同步到磁盘，即可以将数据持久化，并且他比memcached支持更多的数据结构(string,list列表[队列和栈],set[集合],sorted set[有序集合],hash(hash表))．相关参考文档: http://redisdoc.com/index.html

redis使用场景

一般用到 web 服务器的后台中.

1. 登录会话存储：存储在 redis 中，与 memcached 相比，数据不会丢失.
2. 排行版/计数器：比如一些秀场类的项目，经常会有一些前多少名的主播排名．还有一些文章阅读量的技术，或者新浪微博的点赞数等.
3. 作为消息队列：比如 celery 就是使用 redis 作为中间人.
4. 当前在线人数：还是之前的秀场例子，会显示当前系统有多少在线人数.
5. 一些常用的数据缓存：比如我们的 BBS 论坛，板块不会经常变化的，但是每次访问首页都要从 mysql 中获取，可以在 redis 中缓存起来，不用每次请求数据库.
6. 把前 200 篇文章缓存或者评论缓存：一般用户浏览网站，只会浏览前面一部分文章或者评论，那么可以把前面 200 篇文章和对应的评论缓存起来．用户访问超过的，就访问数据库，并且以后文章超过 200 篇，则把之前的文章删除.
7. 好友关系：微博的好友关系使用 redis 实现.
8. 发布和订阅功能：可以用来做聊天软件.

redis和memcached的比较：

	memcached	redis
类型	纯内存数据库	内存磁盘同步数据库
数据类型	在定义 value 时就要固定数据类型	不需要
虚拟内存	不支持	支持
过期策略	支持	支持
存储数据安全	不支持	可以将数据同步到 dump.db 中
灾难恢复	不支持	可以将磁盘中的数据恢复到内存中

分布式	支持	主从同步
订阅与发布	不支持	支持

redis常用操作

对 redis 的操作可以用两种方式，第一种方式采用 redis-cli，第二种方式采用编程语言，比如 Python, PHP 和 JAVA 等.

字符串操作

1. 启动 redis-server
sudo service redis-server start

2. 连接上 redis-server
redis-cli -h [ip] -p [端口]

3. 添加 key value
set key value
如
set username xiaotuo
将字符串值 value 关联到 key. 如果 key 已经持有其他值, set 命令就覆写旧值，无视其类型. 并且默认的过期时间是永久，即永远不会过期.

4. 获取 key 对应的 value
get key username

5. 删除
del key
如
del username

6. 设置过期时间
过期后这个 key 就会自动从 redis 中删除
expire key timeout(单位为秒)
set username zhangsan
expire name 20
get name

也可以在设置值的时候，一同指定过期时间
set key value EX timeout
或
setex key timeout value
如:
set age 18 EX 60

7. 查看过期时间
ttl key
如
ttl username

8. 查看当前 redis 中的所有 key
keys *

列表操作

1. 向列表中添加元素
1.1 在列表左边添加元素
lpush key value
将值 value 插入到列表 key 的表头. 如果 key 不存在, 一个空列表会被创建并执行 lpush 操作. 当 key 存在但不是列表类型时, 将返回一个错误.
如:
lpush websites baidu.com

1.2 在列表右边添加元素
rpush key value
将值 value 插入到列表 key 的表尾. 如果 key 不存在, 一个空列表会被创建并执行 RPUSH 操作. 当 key 存在但不是列表类型时, 返回一个错误.
如:
rpush websites google.com

2. 查看列表中的元素
lrange key start stop
返回列表 key 中指定区间内的元素, 区间以偏移量 start 和 stop 指定,如果要左边的第一个到最后的一个 lrange key 0 -1.
这里 lrange 中的 l 代表的不是 left, 而是 list, 即列表的操作
如:
lrange websites 0 -1 # 获取第 1 个到最后 1 个元素
lpush websites 163.com
rpush websites qq.com
lrange websites 0 1
lrange websites 0 -1

3. 移除列表中的元素
3.1 移除并返回列表 key 的头元素
从左边移除第 0 个元素
lpop key
如
lrange websites 0 -1
lpop websites
lrange websites 0 -1

3.2 移除并返回列表的尾元素
rpop key
如
lrange websites 0 -1
rpop websites
lrange websites 0 -1
3.3 移除并返回列表 key 的中间元素
lrem key count value
将删除 key 这个列表中, count 个值为 value 的元素.
如:
lpush websites baidu.com
rpush websites baidu.com
lrange websites 0 -1
lrem websites 2 baiduc.com # 删除 websites 这个列表中 2 个值为 baidu.com 的元素
lrange websites 0 -1

根据参数 count 的值, 移除列表中与参数 value 相等的元素. count 的值可以是以下几种
count > 0: 从表头开始向表尾搜索, 移除 count 个与 value 相等的元素. 如 lrem websites 2 baidu.com, 从列表左边向右搜索, 移除 2 个值为 baidu.com 的元素.
count < 0: 从表尾开始向表头搜索, 移除与 value 相等的元素, 数量为 count 的绝对值. 如 lrem websites -2 baidu.com, 从列表右边向左搜索, 移除 2 个值为 baidu.com 的元素
count = 0: 移除表中所有与 value 相等的值.

4. 返回指定位置上的元素
lindex key index
将返回 key 这个列表中, 索引为 index 的这个元素.
lindex websites 1 # 返回列表中的第 1 个元素

5. 获取列表中的元素个数
llen key
如
llen languages
llen websites

6. 删除指定的元素
lrem key count value
如
lrem languages 0 php

set集合的操作

与 python 中的集合类似. redis 中的集合与列表的不同之处, 一是集合是无序的, 列表是有序的. 二是集合中的元素是唯一的.

1. 添加元素

sadd set value1 value2....
如
sadd team xiaotuo datuo

2. 查看元素
smembers set
如
smembers team

3. 移除元素
srem set member...
如
srem team xiaotuo
smembers team

4. 查看集合中的元素个数
scard set
如
scard team

5. 获取多个集合的交集
提取两个集合的交集，并生成一个新的集合
sinter set1 set2
如
sadd team1 yoaming kebi zhangsan
sadd team2 yaoming zhaosi wangwu
sinter team1 team2

6. 获取多个集合的并集
所有元素放在一起，并且删除重复的元素
sunion set1 set2
如
sadd team1 yoaming kebi zhangsan
sadd team2 yaoming zhaosi wangwu
sunion team1 team2

7. 获取多个集合的差集
sdiff set1 set2
如
sdiff team1 team2

hash哈希操作，即字典操作

向一个列表/集合/哈希中添加值，如果这个列表/集合/哈希不存在，会自动创建.

1. hset 添加一个新值
hset key field value
如
hset website baidu baidu.com

hset website google google.com
将哈希表 key 中的域 field 的值设为 value.
如果 key 不存在，一个新的哈希表被创建并进行 HSET 操作. 如果域 field 已经存在于哈希表中，旧值将被覆盖.

2. hget 获取哈希中的 field 对应的值
hget key field
如
hget website baidu

3. hgetall 获取某个哈希中所有的 field 和 value
hgetall key
如
hgetall website

4. hkeys 获取某个哈希中所有的 field
hkeys key
如
hkeys website

5. hvals 获取某个哈希中所有的值
hvals key
如
hvals website

6. hdel 删除 field 中的某个 field
hdel key field
如
hdel website baidu

7. hexists 判断哈希中是否存在某个 field
返回 0 表示不存在，返回 1 表示存在
hexists key field
如
hexists website baidu

8. hlen 获取哈希中总共的键值对
hlen field
如
hlen website

1. 事务操作

Redis事务可以一次执行多个命令，事务具有以下特征

- 隔离操作：事务中的所有命令都会序列化，按顺序地执行，不会被其他命令打扰.
- 原子操作：事务中的命令要么全部被执行，要么全部都不执行.

1.1. 开启一个事务
multi
以后执行的所有命令，都在这个事务中执行的.

1.2. 执行事务
exec
会将在 multi 和 exec 中的操作一并提交.

1.3. 取消事务
discard
会将 multi 后的所有命令取消.

1.4. 监视一个或者多个 key
watch key...
监视一个(或多个)key，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断.

1.5. 取消所有 key 的监视
unwatch

1.6. 发布/订阅操作
给某个频道发布消息
publish channel message
1.7. 订阅某个频道的消息
subscribe channel

6.2. Scrapy-Redis 组件介绍

Scrapy 是一个框架，他本身是不支持分布式的. 如果我们想要做分布式的爬虫，就需要借助一个组件叫做 Scrapy-Redis，这个组件正是利用了 Redis 可以分布式的功能，集成到 Scrapy 框架中，使得爬虫可以进行分布式. 可以充分的利用资源（多个 ip，更多带宽，同步爬取）来提高爬虫的爬行效率.

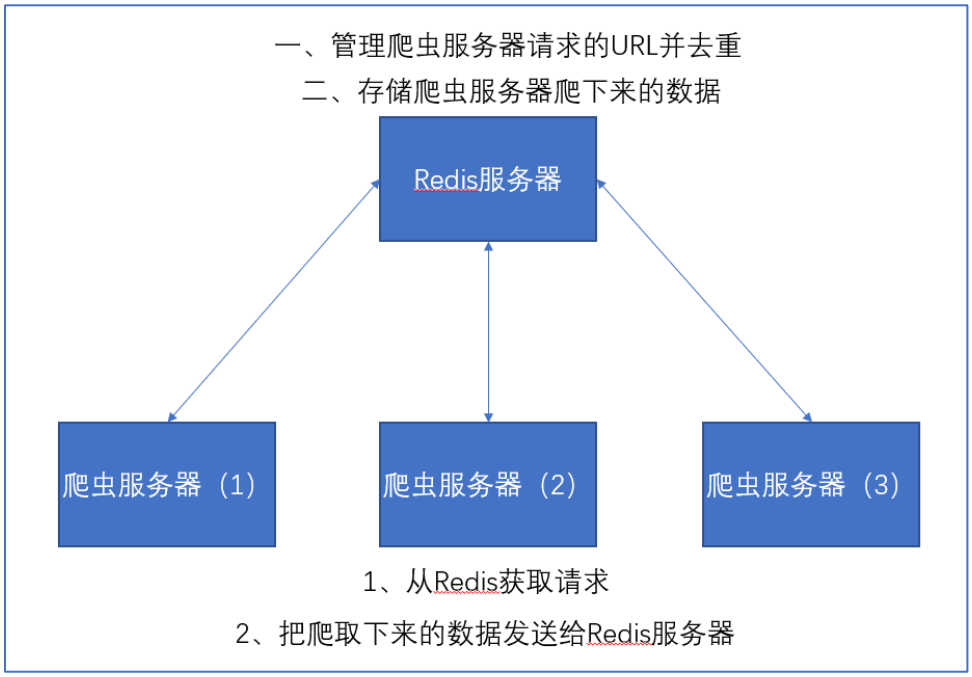
分布式爬虫的优点

1. 可以充分利用多台机器的带宽.
2. 可以充分利用多台机器的 ip 地址.
3. 多台机器做，爬取效率更高.

分布式爬虫必须要解决的问题

1. 分布式爬虫是好几台机器在同时运行，如何保证不同的机器爬取页面的时候不会出现重复爬取的问题．使用 redis 去重．
2. 同样，分布式爬虫在不同的机器上运行，在把数据爬完后如何保证保存在同一个地方．把数据保存到 redis 中．

分布式爬虫架构图



安装scrapy-redis

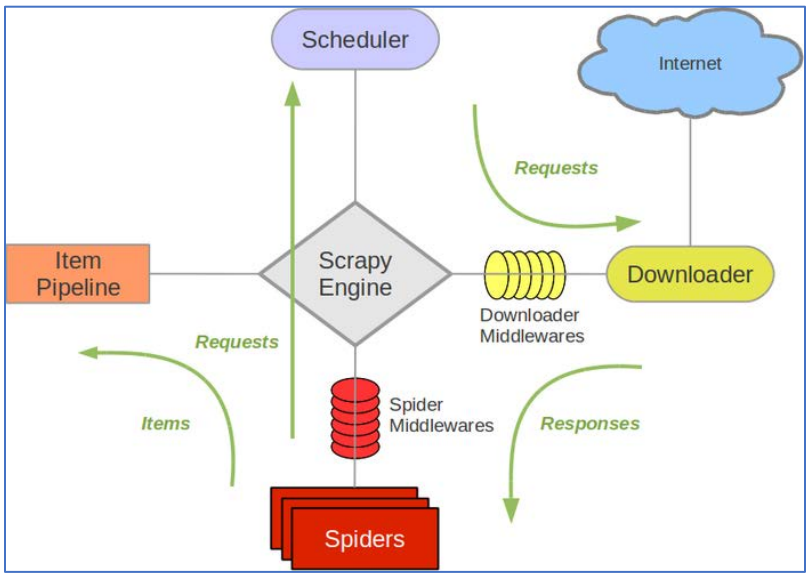
pip install scrapy-redis

Scrapy-Redis架构

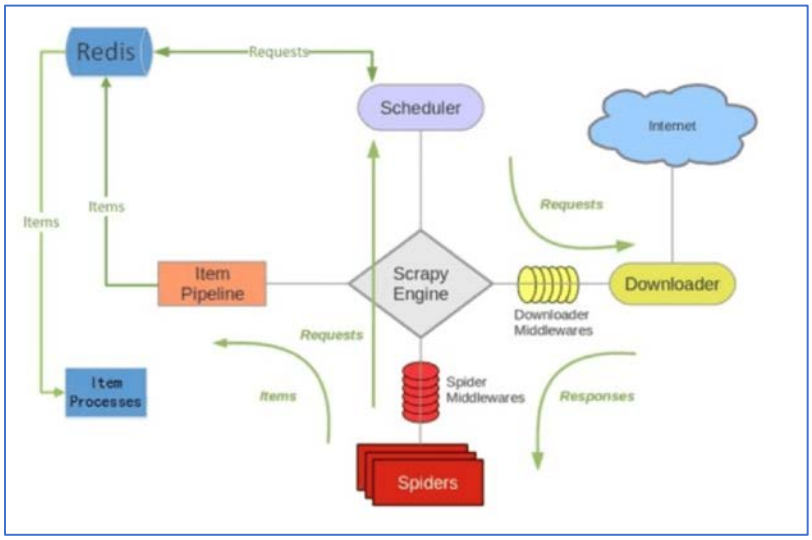
Scrapy-redis 提供了下面四种组件（components）: (四种组件意味着这四个模块都要做相应的修改)

1. Scheduler
2. Duplication Filter
3. Item Pipeline
4. Base Spider

Scrapy架构图



Scrapy-Redis架构图



以上两个图片对比我们可以发现. Item Pipeline在接收到数据后发送给了Redis, Scheduler调度器调度数据也是从Redis中来的，并且数据去重也是在Redis中做的.

scrapy-redis在scrapy的架构上增加了redis，基于redis的特性拓展了如下组件:

Scheduler

Scrapy改造了python本来的collection.deque(双向队列)形成了自己的Scrapy queue (https://github.com/scrapy/queuelib/blob/master/queuelib/queue.py)，但是Scrapy多个spider不能共享待爬取队列Scrapy queue， 即Scrapy本身不支持爬虫分布式,

scrapy-redis 的解决是把这个Scrapy queue换成redis数据库（也是指redis队列），从同一个redis-server存放要爬取的request，便能让多个spider去同一个数据库里读取.

Scrapy中跟"待爬队列"直接相关的就是调度器Scheduler，它负责对新的request进行入列操作（加入Scrapy queue），取出下一个要爬取的request（从Scrapy queue中取出）等操作. 它把待爬队列按照优先级建立了一个字典结构，比如:

```
{
    优先级0：队列0
    优先级1：队列1
    优先级2：队列2
}
```

然后根据request中的优先级，来决定该入哪个队列，出列时则按优先级较小的优先出列. 为了管理这个比较高级的队列字典, Scheduler需要提供一系列的方法. 但是原来的Scheduler已经无法使用，所以使用Scrapy-redis的scheduler组件.

Duplication Filter

Scrapy中用集合实现这个request去重功能, Scrapy中把已经发送的request指纹放入到一个集合中，把下一个request的指纹拿到集合中比对，如果该指纹存在于集合中，说明这个request发送过了，如果没有则继续操作. 这个核心的判重功能是这样实现的:

```
def request_seen(self, request):
    # self.request_figerprints就是一个指纹集合
    fp = self.request_fingerprint(request)

    # 这就是判重的核心操作
    if fp in self.fingerprints:
        return True
    self.fingerprints.add(fp)
    if self.file:
        self.file.write(fp + os.linesep)
```

在scrapy-redis中去重是由Duplication Filter组件来实现的，它通过redis的set 不重复的特性，巧妙的实现了Duplication Filter去重. scrapy-redis调度器从引擎接受request，将request的指纹存 redis的set检查是否重复，并将不重复的request push写 redis的 request queue.
引擎请求request(Spider发出的）时，调度器从redis的request queue队列 里根据优先级pop 出 个request 返回给引擎，引擎将此request发给spider处理.

Item Pipeline

引擎将(Spider返回的)爬取到的Item给Item Pipeline, scrapy-redis 的Item Pipeline将爬取到的 Item 存 redis的 items queue.

修改过Item Pipeline可以很方便的根据 key 从 items queue 提取item，从 实现 items processes集群.

Base Spider

不在使用scrapy原有的Spider类，重写的RedisSpider继承了Spider和RedisMixin这两个类, RedisMixin是用来从redis读取url的类.
当我们生成一个Spider继承RedisSpider时，调用setup_redis函数，这个函数会去连接redis数据库，然后会设置signals(信号):

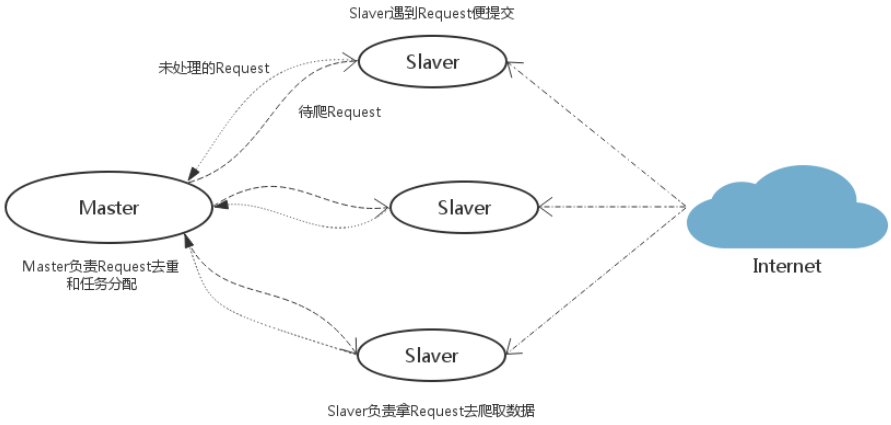
- 一个是当 spider 空闲时候的 signal，会调用 spider_idle 函数，这个函数调用 schedule_next_request 函数，保证 spider 是一直活着的状态，并且抛出 DontCloseSpider 异常.
- 一个是当抓到一个 item 时的 signal，会调用 item_scraped 函数，这个函数会调用 schedule_next_request 函数，获取下一个 request.

scheduler把request对象保存到redis中，每次发送request请求前都去redis中进行查找，如果之前没有爬取过，就把该request对象返回给scheduler，然后下行爬取，爬取到的数据也经过item pipeline保存到redis数据库中.

Scrapy-Redis分布式策略:

假设有四台电脑: Windows 10、Mac OS X、Ubuntu 16.04、CentOS 7.2，任意一台电脑都可以作为 Master端 或 Slaver端，比如:

- Master 端(核心服务器)：使用 Windows 10，搭建一个 Redis 数据库，不负责爬取，只负责 url 指纹判重、Request 的分配，以及数据的存储
- Slaver 端(爬虫程序执行端)：使用 Mac OS X 、Ubuntu 16.04、CentOS 7.2，负责执行爬虫程序，运行过程中提交新的 Request 给 Master



1. 首先 Slaver 端从 Master 端拿任务（Request、url）进行数据抓取, Slaver 抓取数据的同时，产生新任务的 Request 便提交给 Master 处理；
2. Master 端只有一个 Redis 数据库，负责将未处理的 Request 去重和任务分配，将处理后的 Request 加入待爬队列，并且存储爬取的数据.

   Scrapy-Redis默认使用的就是这种策略，我们实现起来很简单，因为任务调度等工作Scrapy-Redis都已经帮我们做好了，我们只需要继承RedisSpider、指定redis_key就行了.

   redis_key是统一管理slaver端爬取的指令.

   缺点是, Scrapy-Redis调度的任务是Request对象，里面信息量比较大（不仅包含url，还有callback函数、headers等信息），可能导致的结果就是会降低爬虫速度、而且会占用Redis大量的存储空间，所以如果要保证效率，那么就需要一定硬件水平.

源码自带项目说明

使用scrapy-redis的example来修改

先从github上拿到scrapy-redis的示例，然后将里面的example-project目录移到指定的地址:

```
# clone github scrapy-redis源码文件
git clone https://github.com/rolando/scrapy-redis.git

# 直接拿官方的项目范例，改名为自己的项目用（针对懒癌患者)
mv scrapy-redis/example-project ~/scrapyredis-project
```

我们clone到的 scrapy-redis 源码中有自带一个example-project项目，这个项目包含3个spider，分别是dmoz, myspider_redis, mycrawler_redis.

注意example-project的名字可以任意修改，但是其中example文件夹不能随意修改，一旦修改了example文件夹，就要在example-project文件夹下的scrapy.cfg文件中修改对应的文件夹名称. example/settings.py中的SPIDER_MODULES的名称也要修改.

一、dmoz (class DmozSpider(CrawlSpider))

这个爬虫继承的是CrawlSpider，它是用来说明Redis的持续性，当我们第一次运行dmoz爬虫，然后Ctrl + C停掉之后，再运行dmoz爬虫，之前的爬取记录是保留在Redis里的.

分析起来，其实这就是一个 scrapy-redis 版 `CrawlSpider` 类，需要设置Rule规则，以及callback不能写parse()方法.

实际上就是一个Crawl Scrapy的项目，爬取http://www.dmoz.org/网站上的分类目录

items文件

```
from scrapy.item import Item, Field
from scrapy.loader import ItemLoader
from scrapy.loader.processors import MapCompose, TakeFirst, Join

#定义了多个字段
class ExampleItem(Item):
    name = Field()
    description = Field()
    link = Field()
    crawled = Field()
    spider = Field()
    url = Field()

#使用ItemLoader方法，与item方法有些许的不同
class ExampleLoader(ItemLoader):
    default_item_class = ExampleItem
    default_input_processor = MapCompose(lambda s: s.strip())
    default_output_processor = TakeFirst()
    description_out = Join()
```

pipeline文件

```
from datetime import datetime

class ExamplePipeline(object):
    def process_item(self, item, spider):
        #数据给过管道后会添加以下两个字段
        #添加了linux的时间戳，即数据获取的时间,UTC时间，全球标准时间．英国格林威治的时间．中国时间为utc时间+8
            item["crawled"] = datetime.utcnow()
            #添加了一个爬虫名的字段，有很多个爬虫，给数据添加一个爬虫名的字段后就知道是哪个爬虫爬取的了．
            item["spider"] = spider.name
```

```
        return item
```

settings文件

```
#爬虫位置
SPIDER_MODULES = ['example.spiders']
NEWSPIDER_MODULE = 'example.spiders'
#默认的user_agent，要进行修改
USER_AGENT = 'scrapy-redis (+https://github.com/rolando/scrapy-redis)'

#使用了scrapy_redis中的去重方法
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"
#使用了scrapy-redis中的调度组件
SCHEDULER = "scrapy_redis.scheduler.Scheduler"
#允许执行项目时可以暂停，暂停后不会清空redis数据库的记录
SCHEDULER_PERSIST = True

#如果三个都注释掉，就使用默认的优先级策略．默认的是scrapy请求队列形式，是按优先级顺序排列的队列形式，一般来说使用把三个都注释掉，使用默认的即可．出队列的顺序不影响我们的结果．
#在使用redis分布式的爬虫时，如果使用scrapy的请求队列形式，虽然不影响请求的发送，但在redis数据库中却不会显示出请求队列．必须使用redis的三种请求队列之一才能在redis数据库中显示出请求队列．

#按sorted排序顺序出队列
SCHEDULER_QUEUE_CLASS = "scrapy_redis.queue.SpiderPriorityQueue"
#队列形式，规则是请求先进先出
#SCHEDULER_QUEUE_CLASS = "scrapy_redis.queue.SpiderQueue"
#栈形式，规则是请求先进后出
#SCHEDULER_QUEUE_CLASS = "scrapy_redis.queue.SpiderStack"

ITEM_PIPELINES = {
    #自定义的pipelines，先经过我们自定义的pipeline，添加了utc时间和爬虫名．可以自定义多个管道文件，经过一系列的处理，最后再放到redis数据库中．
    'example.pipelines.ExamplePipeline': 300,
    #scrapy_redis.pipelines.RedisPipeline将数据存储在Redis数据库中，必须要启动，如果不想把数据存储在Redis数据库中，也可以使用自定义的pipeline进行数据的存储．此时把放在redis数据库的优先级设置到最后．
    'scrapy_redis.pipelines.RedisPipeline': 400,
}
```

```
#定义数据存储的位置，如果不写这两个参数，则默认是保存在本地的.
#指定数据库的主机IP
REDIS_HOST = "111.186.110.33"
#指定远程数据库的端口号
REDIS_PORT = 6379

#LOG日志
LOG_LEVEL = 'DEBUG'

# Introduce an artifical delay to make use of parallelism. to speed up the
# crawl.
#下载延迟
DOWNLOAD_DELAY = 2
```

dmoz爬虫文件

```
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule
#一个简单的CrawlSpider
#使用的是CralSpider的爬虫，不是redis的爬虫，故不支持分布式.
class DmozSpider(CrawlSpider):
    """Follow categories and extract links."""
    name = 'dmoz'
    allowed_domains = ['dmoz.org']
    start_urls = ['http://www.dmoz.org/']

    rules = [
        Rule(LinkExtractor(
            restrict_css=('.top-cat', '.sub-cat', '.cat-item')
        ), callback='parse_directory', follow=True),
    ]

    def parse_directory(self, response):
        for div in response.css('.title-and-desc'):
            yield {
                'name': div.css('.site-title::text').extract_first(),
                'description': div.css('.site-descr::text').extract_first().strip(),
                'link': div.css('a::attr(href)').extract_first(),
            }
```

main.py

```
from scrapy import cmdline
```

```
cmdline.execute('scrapy crawl dmoz'.split())
```

执行dmoz爬虫

执行方式: scrapy crawl dmoz

二、myspider_redis (class MySpider(RedisSpider))

这个爬虫继承了RedisSpider, 它能够支持分布式的抓取，采用的是basic spider, 需要写parse函数.

其次就是不再有start_urls了，取而代之的是redis_key, scrapy-redis将key从Redis里pop出来，成为请求的url地址.

```
from scrapy_redis.spiders import RedisSpider

class MySpider(RedisSpider):
    """Spider that reads urls from redis queue (myspider:start_urls)."""
    name = 'myspider_redis'
    #启动所有slaver端爬虫的指令，下面的格式是参考格式，建议采用这种格式
    #myspider为当前爬虫类的名字
    #使用lpush myspider:start_urls http://www.dmoz.org/命令来启动爬虫
    redis_key = 'myspider:start_urls'

    #指定爬取的域范围
    # allowd_domains = ['dmoz.org']

    # 可选: 等效于allowd_domains(), __init__方法按规定格式写，使用时只需要修改super()里的类名参数即可
    def __init__(self, *args, **kwargs):
        # Dynamically define the allowed domains list.
        #通过执行时输入的start_urls动态的获取爬取的域的范围
        domain = kwargs.pop('domain', '')
        self.allowed_domains = filter(None, domain.split(','))
        # 修改这里的类名为当前类名, super相当于调用了父类的初始化方法
        super(MySpider, self).__init__(*args, **kwargs)

    def parse(self, response):
        return {
            'name': response.css('title::text').extract_first(),
            'url': response.url,
        }
```

main.py

```
from scrapy import cmdline
```

```
cmdline.execute('scrapy runspider myspider_redis.py'.split())
        }
```

注意:

1. RedisSpider 类 不需要写 allowd_domains 和 start_urls:
2. scrapy-redis 将从在构造方法__init__()里动态定义爬虫爬取域范围，也可以选择直接写 allowd_domains.
3. 必须指定 redis_key，即启动爬虫的命令，参考格式: redis_key = 'myspider:start_urls'
4. 根据指定的格式, start_urls 将在 Master 端的 redis-cli 里 lpush 到 Redis 数据库里, RedisSpider 将在数据库里获取 start_urls.

执行方式:

1. 通过 runspider 方法执行爬虫的 py 文件（也可以分次执行多条），爬虫（们）将处于等待准备状态:
   scrapy runspider myspider_redis.py

2. 在 Master 端的 redis-cli 输入 push 指令，参考格式:
   $redis > lpush myspider:start_urls http://www.dmoz.org/

   只要是连接到这个 redis 数据库上的 slaver 端，都会开始分配 url 并且执行爬取任务.

   是由 master 端通过 redis 数据库进行任务的分配的.

3. Slaver 端爬虫获取到请求，开始爬取.

三、mycrawler_redis (class MyCrawler(RedisCrawlSpider))

这个RedisCrawlSpider类爬虫继承了RedisCrawlSpider，能够支持分布式的抓取.
因为采用的是crawlSpider，所以需要遵守Rule规则，以及callback不能写parse()方法.

同样也不再有start_urls了，取而代之的是redis_key, scrapy-redis将key从Redis里pop出来，成为请求的url地址.

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

from scrapy_redis.spiders import RedisCrawlSpider


class MyCrawler(RedisCrawlSpider):
```

```
    """Spider that reads urls from redis queue
(myspider:start_urls)."""
    name = 'mycrawler_redis'
    redis_key = 'mycrawler:start_urls'

    rules = (
        # follow all links,可以写多个Rule, callback的名字不能是parse
        Rule(LinkExtractor(), callback='parse_page', follow=True),
    )
    # __init__方法必须按规定写, 使用时只需要修改super()里的类名参数即
可
    def __init__(self, *args, **kwargs):
        # Dynamically define the allowed domains list.
        domain = kwargs.pop('domain', '')
        self.allowed_domains = filter(None, domain.split(','))
        # 修改这里的类名为当前类名
        super(MyCrawler, self).__init__(*args, **kwargs)

    def parse_page(self, response):
        return {
            'name': response.css('title::text').extract_first(),
            'url': response.url,
        }
```

注意:

同样的, RedisCrawlSpider类不需要写allowd_domains和start_urls:

1. scrapy-redis 将从在构造方法__init__()里动态定义爬虫爬取域范围, 也可以选择直接写 allowd_domains.
2. 必须指定 redis_key, 即启动爬虫的命令, 参考格式: redis_key = 'myspider:start_urls'
3. 根据指定的格式, start_urls 将在 Master 端的 redis-cli 里 lpush 到 Redis 数据库里, RedisSpider 将在数据库里获取 start_urls.

执行方式:

1. 通过 runspider 方法执行爬虫的 py 文件（也可以分次执行多条）, 爬虫（们）将处于等待准备状态:

   scrapy runspider mycrawler_redis.py
2. 在 Master 端的 redis-cli 输入 push 指令, 参考格式:

   $redis > lpush mycrawler:start_urls http://www.dmoz.org/
3. 爬虫获取 url, 开始执行.

总结三种爬虫：

1. 如果只是用到 Redis 的去重和保存功能，就选第一种；
2. 如果要写分布式，则根据情况，选择第二种、第三种；
3. 通常情况下，会选择用第三种方式编写深度聚焦爬虫.

把scrapy爬虫改写Scrapy-Redis分布式爬虫

要将一个Scrapy项目变成一个Scrapy-redis项目只需修改以下三点就可以了

1. 将爬虫的类从scrapy.Spider变成scrapy_redis.spiders.RedisSpider；或者是从scrapy.CrawlSpider变成scrapy_redis.spiders.RedisCrawlSpider.
2. 将爬虫中的start_urls删掉. 增加一个redis_key="xxx". 这个redis_key是为了以后在redis中控制爬虫启动的. 爬虫的第一个url，就是在redis中通过这个发送出去的.
3. 在settings.py文件中增加如下配置

```
# Scrapy-Redis 相关配置
# 把 request 存储到 redis 中
SCHEDULER = "scrapy_redis.scheduler.Scheduler"

# 确保所有爬虫共享相同的去重指纹
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"

# 设置 redis 的 item pipeline
ITEM_PIPELINES = {
    'scrapy_redis.pipelines.RedisPipeline': 300
}

# 在 redis 中保持 scrapy-redis 用到的队列，不会清理 redis 中的队列，从而可以实现暂停和恢复的功能.
SCHEDULER_PERSIST = True

# 设置连接 redis 信息
REDIS_HOST = '127.0.0.1'
REDIS_PORT = 6379
```

运行Scrapy-Redis爬虫

1. 在爬虫服务器上. **进入爬虫文件所在的路径**，然后输入命令

```
scrapy runspider [爬虫名字].
```

2. 在 Redis 服务器上, push 一个开始的 url 链接, 爬虫服务器从 reids 服务器中读取到 start_url 的地址后, 就开始进行爬取.

```
redis-cli> lpush [redis_key] start_url
```

6.3. 搜房网分布式爬虫

需求分析

爬取全国各个城市中新房和二手房的信息

1. 获取所有的城市的 url 链接.
   http://www.fang.com/SoufunFamily.htm
2. 获取所有城市的新房的 url 链接.
   例: 安庆: http://anqing.fang.com/
   安庆新房: http://newhouse.anqing.fang.com/house/s/
3. 获取所有城市的二手房的 url 链接.
   例: 安庆: http://anqing.fang.com/
   安庆二手房: http://esf.anqing.fang.com/

北京是个例外
   北京的新房链接: http://newhouse.fang.com/house/s/
   北京的二手房链接: http://esf.fang.com/

获取所有城市的链接 > 获取所有城市新房/二手房的链接 >

按省份进行选择, 获取某个城市时保存对应的省的信息

每个 tr 是一行
tbody>tr>td[2]中是省份的信息,
tbody>tr>td[3]中是省份中城市的信息, 获取城市名称及链接

一个省的城市可以包含多行
某个省份第一行中 td[2]是省份的信息, td[3]中是城市的信息
某个省份第二行开始 td[2]中是空白内容 " ", td[3]中是城市的信息

首先实现 scrapy 单机爬虫, 再把单机爬虫修改为分布式爬虫

首先实现scrapy单机爬虫

新建项目及爬虫

新建项目 scrapy startproject fang_pro
新建爬虫 scrapy genspider fang

修改settings.py，进行基本的配置

```
ROBOTSTXT_OBEY = False

DOWNLOAD_DELAY = 3

# Override the default request headers:
DEFAULT_REQUEST_HEADERS = {
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
   'Accept-Language': 'en',
   'User-Agent': 'Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/40.0.2214.93 Safari/537.36'
}
```

修改items.py，定义要爬取的字段

```
# -*- coding: utf-8 -*-

import scrapy


class NewHouseItem(scrapy.Item):
    # 省份
    province = scrapy.Field()
    # 城市
    city    = scrapy.Field()
    # 小区的名字
    name = scrapy.Field()
    # 价格
    price = scrapy.Field()
    # 几居. 这个是个列表
    rooms = scrapy.Field()
    # 面积
    area = scrapy.Field()
    # 地址
    address = scrapy.Field()
    # 行政区
    district = scrapy.Field()
```

```
    # 是否在售
    sale = scrapy.Field()
    # 房天下详情页面的url
    origin_url = scrapy.Field()

class ESFHouseItem(scrapy.Item):
    # 省份
    province = scrapy.Field()
    # 城市
    city = scrapy.Field()
    # 小区的名字
    name = scrapy.Field()
    # 几室几厅
    rooms = scrapy.Field()
    # 层
    floor = scrapy.Field()
    # 朝向
    toward = scrapy.Field()
    # 年代
    year = scrapy.Field()
    # 地址
    address = scrapy.Field()
    # 建筑面积
    area = scrapy.Field()
    # 总价
    price = scrapy.Field()
    # 单价
    unit = scrapy.Field()
    # 原始的url
    origin_url = scrapy.Field()
```

修改fang.py爬虫文件

```
# -*- coding: utf-8 -*-
import scrapy
import re
from fang.items import NewHouseItem, ESFHouseItem
from scrapy_redis.spiders import RedisSpider


class SfwSpider(RedisSpider):
    name = 'fang'
    allowed_domains = ['fang.com']
    start_urls = ['http://www.fang.com/SoufunFamily.htm']

    def parse(self, response):
        # 提取出所有的tr 标签，每个tr 标签是一行城市的信息
        trs = response.xpath("//div[@class='outCont']//tr")
        province = None
        for tr in trs:
```

```
            # 每个 tr 中都有 3 个 td 标签，第 1 个 td 标签是不需要的，它有 class 的属性，后 2 个 td 标签是我们需要的，它们都没有 class 属性。使用 td[not(@class)] 来选择不包含 class 的属性
            tds = tr.xpath(".//td[not(@class)]")
            # 包含省份信息的 td 标签
            province_td = tds[0]
            province_text = province_td.xpath(".//text()").get()
            # 替换掉 province_text 中的空格，如果去除空格后得到的 province_text 不为空值，说明它是省份的信息
            province_text = re.sub(r"\s", "", province_text)
            if province_text:
                # 先在循环外定义一个 province 的变量，如果在一个 tr 中提取到了省份的信息，就把 province 的值修改为提取到的省份，对于下一次循环的 tr，如果没有提取到省份的信息，表明这一行 tr 中的城市是上一行 tr 中省份中的城市，此时不对 province 重新进行赋值，province 的值还是上一次 tr 循环中提取出来的省份，就使用上一个 tr 中的省份
                province = province_text

            # 不爬取海外的城市的房源
            if province == '其它':
                continue

            # 提取出城市的信息
            city_td = tds[1]
            city_links = city_td.xpath(".//a")
            for city_link in city_links:
                city = city_link.xpath(".//text()").get()
                city_url = city_link.xpath(".//@href").get()
                # print("省份: ", province)
                # print("城市: ", city)
                # print("城市链接: ", city_link)

                # 从 city_url 构建新房和二手房 url
                # "http://anqing.fang.com/" 分割后第 0 个元素是域名，第 1 个元素是网址
                url_module = city_url.split("//")
                scheme = url_module[0]
                domain = url_module[1]
                if 'bj.' in domain:
                    newhouse_url = 'http://newhouse.fang.com/house/s/'
                    esf_url = 'http://esf.fang.com/'
                else:
                    # 构建新房的 url 链接, "http://newhouse.anqing.fang.com/house/s/"
                    newhouse_url = scheme + '//' + "newhouse." + domain + "house/s/"
                    # 构建二手房的 url 链接, "http://esf.anqing.fang.com/"
                    esf_url = scheme + "//" + "esf." + domain
                # print('省份: %s\t 城市: %s' % (province, city))
                # print('新房 url: %s' % newhouse_url)
                # print('二手房 url: %s' % esf_url)

                # 构造新房的 Request 对象，在写二手房的爬虫时可以把新房的 Request 注释掉
```

```
                    yield scrapy.Request(url=newhouse_url, callback=self.parse_newhouse,
meta={"info":(province,city)})

                    yield scrapy.Request(url=esf_url, callback=self.parse_esf, meta={"info":
(province, city)})
                    # 测试代码，只爬取 1 个城市
                    # break
                # 测试代码，只爬取一行城市数据
                # break

    # 解析新房的响应
    def parse_newhouse(self, response):
        province, city = response.meta.get('info')
        # 每一个 li 标签都是一个小区的数据
        lis = response.xpath("//div[contains(@class,'nl_con')]/ul/li")
        for li in lis:
            # 小区名称
            name = li.xpath(".//div[@class='nlcd_name']/a/text()").get().strip()
            # 小区房型列表
            house_type_list = li.xpath(".//div[contains(@class,'house_type')]/a/text()").getall()
            house_type_list = list(map(lambda x: re.sub(r"\s", "", x), house_type_list))
            house_type_list = [re.sub(r"\s", "", x) for x in house_type_list]
            # 几居室列表
            rooms = list(filter(lambda x: x.endswith("居"), house_type_list))
            rooms = [x for x in house_type_list if x.endswith('居')]
            # 小区房屋的面积
            area = "".join(li.xpath(".//div[contains(@class,'house_type')]/text()").getall())
            area = re.sub(r"\s|一|/", "", area)
            # 地址
            address = li.xpath(".//div[@class='address']/a/@title").get()
            # 小区所在行政区，注意不能用 span 标签进行提取，因为不是所有的行政区信息
都放在 span 标签中.
            district_text = "".join(li.xpath(".//div[@class='address']/a//text()").getall())
            district = re.search(r".*\[(.+)\].*", district_text).group(1)
            # 销售状态是在售还是待售
            sale = li.xpath(".//div[contains(@class,'fangyuan')]/span/text()").get()
            # 价格，也同时获取到价格的单位，同时把列表转换为字符串
            price = "".join(li.xpath(".//div[@class='nhouse_price']//text()").getall())
            price = re.sub(r"\s|广告", "", price)
            # 详情页 url
            origin_url = li.xpath(".//div[@class='nlcd_name']/a/@href").get()

            # 构建 item 并返回
            item = NewHouseItem(name=name, rooms=rooms, area=area, address=address,
district=district, sale=sale, price=price, origin_url=origin_url, province=province, city=city)
            yield item

        # 获取下一页的链接
        next_url = response.xpath("//div[@class='page']//a[@class='next']/@href").get()
```

```
        # 构造下一页的请求对象，并指定解析函数为自身，同时把省份和城市信息通过 meta 传递过去
        if next_url:
            yield scrapy.Request(url=response.urljoin(next_url), callback=self.parse_newhouse, meta={"info": (province, city)})

    # 二手房信息，新房是针对某个小区的，二手房是具体到某个房子的.
    def parse_esf(self, response):
        province, city = response.meta.get('info')
        # 以每个房子进行分组，遍历分组提取每个房子的信息
        dls = response.xpath("//div[@class='houseList']/dl")
        for dl in dls:
            # 对每一个房源，都添加省市信息
            item = ESFHouseItem(province=province, city=city)
            item['name'] = dl.xpath(".//p[@class='mt10']/a/span/text()").get()
            infos = dl.xpath(".//p[@class='mt12']/text()").getall()
            infos = list(map(lambda x: re.sub(r"\s", "", x), infos))
            infos = [re.sub(r'\s', '', info) for info in infos]
            # 得到的 infos 是一个列表，其中包含着房屋的信息，对其元素进行过滤，提取出房屋的各项信息
            for info in infos:
                if "厅" in info:
                    item['rooms'] = info
                elif '层' in info:
                    item['floor'] = info
                elif '向' in info:
                    item['toward'] = info
                else:
                    item['year'] = info.replace("建筑年代: ", "")
            item['address'] = dl.xpath(".//p[@class='mt10']/span/@title").get()
            item['area'] = dl.xpath(".//div[contains(@class,'area')]/p/text()").get()
            item['price'] = "".join(dl.xpath(".//div[@class='moreInfo']/p[1]//text()").getall())
            item['unit'] = "".join(dl.xpath(".//div[@class='moreInfo']/p[2]//text()").getall())
            detail_url = dl.xpath(".//p[@class='title']/a/@href").get()
            item['origin_url'] = response.urljoin(detail_url)
            yield item
        # 提取下一页的 url，构造请求
        next_url = response.xpath("//a[@id='PageControl1_hlk_next']/@href").get()
        yield scrapy.Request(url=response.urljoin(next_url), callback=self.parse_esf, meta={"info": (province, city)})
```

修改pipelines.py，把数据保存到json文件中

```
# -*- coding: utf-8 -*-

from scrapy.exporters import JsonLinesItemExporter

class FangPipeline(object):
    def __init__(self):
        self.newhouse_fp = open('newhouse.json','wb')
```

```
        self.esfhouse_fp = open('esfhouse.json','wb')
        self.newhouse_exporter = JsonLinesItemExporter(self.newhouse_fp, ensure_ascii=False)
        self.esfhouse_exporter = JsonLinesItemExporter(self.esfhouse_fp, ensure_ascii=False)

    def process_item(self, item, spider):
        self.newhouse_exporter.export_item(item)
        self.esfhouse_exporter.export_item(item)
        return item

    def close_spider(self,spider):
        self.newhouse_fp.close()
        self.esfhouse_fp.close()
```

修改middlewares.py，添加随机请求头

```
# -*- coding: utf-8 -*-

import random

class UserAgentDownloadMiddleware(object):
    # user-agent 随机请求头中间件
    USER_AGENTS = [
        'Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_8; en-us) AppleWebKit/534.50 (KHTML, like Gecko) Version/5.1 Safari/534.50',
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36',
        'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;',
        'Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)',
        'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv,2.0.1) Gecko/20100101 Firefox/4.0.1',
        'Mozilla/5.0 (Windows NT 6.1; rv,2.0.1) Gecko/20100101 Firefox/4.0.1'
    ]
    def process_request(self,request,spider):
        user_agent = random.choice(self.USER_AGENTS)
        request.headers['User-Agent'] = user_agent
```

修改settings.py，启用pipeline及middleware

```
DOWNLOADER_MIDDLEWARES = {
    'fang.middlewares.UserAgentDownloadMiddleware': 543,
}

ITEM_PIPELINES = {
    'fang.pipelines.FangPipeline': 300,
}
```

在项目根目录下新建 start.py

```
#encoding: utf-8
```

```
from scrapy import cmdline

cmdline.execute("scrapy crawl sfw".split())
```

修改单机爬虫为分布式爬虫

以上已经实现了单机的 scrapy 爬虫，下面要把单机的爬虫修改为分布式的爬虫.

配置ubuntu linux版的Redis服务器和爬虫服务器

安装ubuntu支持软件

```
# 更新软件库
apt-get update
# 安装支持软件
sudo apt-get -y install python3-dev build-essential python3-pip libxml2-dev libxslt1-dev zlib1g-dev libffi-dev libssl-dev
```

安装项目使用的python包

在实际的项目开发中，一般也是先在 windows 中进行开发，然后把项目部署到 linux 服务器中.
在 windows 中进入虚拟环境，再进入到项目所在的根目录中，使用以下命令把项目开发使用的 python 包保存到 txt 文件中

```
pip freeze > requirements.txt
```

如果是部署到 linux 服务器中，因为其中的 pypiwin32 是 windows 上专用的包，需要删除掉，否则会在安装时报错.

在 xshell 中使用 rz 命令从本机向远程 linux 服务器中传递数据

在 linux 服务器中安装 reqirements.txt 中的 python 包

在 linux 中同样创建虚拟环境

```
# 安装包管理器
pip install virtualenvwrapper
# 查看 python3 命令所在的目录
which python3
/usr/bin/python3
# 创建虚拟环境，并指定 python3 的位置.
mkvirtualenv -p /usr/bin/python3 crawler-env
```

```
# 进入虚拟环境
workon crawler-env
# 删除
# 安装 requirements.txt 中的 python 包
pip install -r requirements.txt

# 在爬虫服务器中安装 scrapy-redis 组件
pip install scrapy-redis
```

把项目改造成分布式爬虫.

1. 将爬虫的类从 `scrapy.Spider` 变成 `scrapy_redis.spiders.RedisSpider`；或者是从 `scrapy.CrawlSpider` 变成 `scrapy_redis.spiders.RedisCrawlSpider`.

2. 将爬虫中的 `start_urls` 删掉. 增加一个 `redis_key="xxx"`. 这个 `redis_key` 是为了以后在 `redis` 中控制爬虫启动的. 爬虫的第一个 url, 就是在 redis 中通过这个发送出去的. redis_key 指定了之后爬虫会从 redis 服务器中的如个 key 中去读取 start_url. 使用 fang:start_urls 对 key 进行区分.

```python
from scrapy_redis.spiders import RedisSpider


class SfwSpider(RedisSpider):
    name = 'fang'
    allowed_domains = ['fang.com']
    # start_urls = ['http://www.fang.com/SoufunFamily.htm']
    redis_key = "fang:start_urls"

    def parse(self, response):
```

3. 在配置文件中增加如下配置

```python
# Configure item pipelines
# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html
# 注释掉单机的 pipeline, 使用 redis 的 pipeline 把数据保存到 redis 中.
# ITEM_PIPELINES = {
#     'fang.pipelines.FangPipeline': 300,
# }


# Scrapy-Redis 相关的配置

# 使用 redis 的调度器, 确保 request 存储到 redis 中
SCHEDULER = "scrapy_redis.scheduler.Scheduler"

# 使用 scrapy_redis 进行去重, 确保所有爬虫共享相同的去重指纹
```

```
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"

# 使用 redis 的 item pipeline，把数据都传输并保存到 redis 服务器中
ITEM_PIPELINES = {
    'scrapy_redis.pipelines.RedisPipeline': 300
}

# 在 redis 中保持 scrapy-redis 用到的队列，不会清理 redis 中的队列，从而可以实现暂停和恢复的功能.
SCHEDULER_PERSIST = True

# 设置连接 redis 信息
REDIS_HOST = '192.168.1.6'
REDIS_PORT = 6379
```

把项目打包并上传到linux服务器中

zip fang_pro fang_pro.zip
unzip fang_pro.zip

cd fang_pro

workon crawl-env

运行爬虫

1. 启动redis服务器中的redis服务

redis-server redis.windows.conf

2. 在所有爬虫服务器上运行爬虫

进入爬虫文件所在的路径

cd fang_pro/fang_pro/spiders

然后输入命令运行爬虫

`scrapy runspider fang.py`

因为没有起始的 url 地址，所以所有的爬虫都处于等待状态

3. 向redis中push入start_urls

在 redis 服务器所在的主机上，另外打开一个 cmd 窗口执行 redis-cli 连接到服务器中，向 Redis 服务器中添加一个起始的 url 链接

```
lpush fang:start_urls http://www.fang.com/SoufunFamily.com
```

只要 redis 中有了 fang:start_urls 这个键，所有的爬虫都开始运行起来了.

处理Redis里的数据

有缘网的数据爬回来了，但是放在Redis里没有处理. 之前我们配置文件里面没有定制自己的ITEM_PIPELINES，而是使用了RedisPipeline，所以现在这些数据都被保存在redis的youyuan:items键中，所以我们需要另外做处理.

在scrapy-youyuan目录下可以看到一个process_items.py文件，这个文件就是scrapy-redis的example提供的从redis读取item进行处理的模版.
假设我们要把youyuan:items中保存的数据读出来写进MongoDB或者MySQL，那么我们可以自己写一个process_youyuan_profile.py文件，然后保持后台运行就可以不停地将爬回来的数据入库了.

存入MongoDB

1. 启动 MongoDB 数据库：sudo mongod
2. 执行下面程序：py2 process_youyuan_mongodb.py

```
# process_youyuan_mongodb.py

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

import json
import redis
import pymongo

def main():

    # 指定Redis数据库信息
    rediscli = redis.StrictRedis(host='192.168.199.108', port=6379, db=0)
    # 指定MongoDB数据库信息
    mongocli = pymongo.MongoClient(host='localhost', port=27017)

    # 创建数据库名
    db = mongocli['youyuan']
    # 创建表名
```

```
    sheet = db['beijing_18_25']

    while True:
        # FIFO模式为 blpop, LIFO模式为 brpop，获取键值
        source, data = rediscli.blpop(["youyuan:items"])

        item = json.loads(data)
        sheet.insert(item)

        try:
            print u"Processing: %(name)s <%(link)s>" % item
        except KeyError:
            print u"Error procesing: %r" % item

if __name__ == '__main__':
    main()
```

视频演示

```
# process_youyuan_mongodb.py

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

import redis, pymongo, json
#操作 redis 数据库和 mongo 数据库，之前向 redis 数据库存储数据其实是一个 json 对象，现在保存为 MongoDB 数据，还要转换回去.

def process_item():
    #  创建 redis 数据库连接
    rediscli = redis.Redis(host = "127.0.0.1", port = 6379, db = "0")

    #  创建 MongoDB 数据库连接
    mongocli = pymongo.MongoClient(host = "127.0.0.1", port = 27017)

    #  创建 mongodb 数据库名称
    dbname = mongocli["youyuan"]
    #  创建 mongodb 数据库 youyuan 的表名称
    sheetname = dbname["beijing_18_25_mm"]
    #查看有多少条记录
    offset = 0
    #使用循环向 MongoDB 中添加多个数据
    while True:
        # redis 数据表名 和 数据
        source, data = rediscli.blpop("yy:items")
        offset += 1
        #  将 json 对象转换为 Python 对象
        data = json.loads(data)

        #  将数据插入到 sheetname 表里
```

```
            sheetname.insert(data)
            print(offset)

if __name__ == "__main__":
    process_item()
```

mongod启动mongodb服务

执行python process_item_for_mongodb.py

mongo启动mongo shell环境

db

show dbs

use youyuan

show collections

db.beijing_18_25_mm.find()

存入 MySQL

1. 启动 mysql：mysql.server start（更平台不一样）
2. 登录到 root 用户：mysql -uroot -p
3. 创建数据库 youyuan:create database youyuan;
4. 切换到指定数据库：use youyuan
5. 创建表 beijing_18_25 以及所有字段的列名和数据类型.
6. 执行下面程序：py2 process_youyuan_mysql.py

```
#process_youyuan_mysql.py

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

import json
import redis
import MySQLdb

def main():
    #  指定redis数据库信息
    rediscli = redis.StrictRedis(host='192.168.199.108', port = 6379, db = 0)
    #  指定mysql数据库
    mysqlcli = MySQLdb.connect(host='127.0.0.1', user='power', passwd='xxxxxxx', db =
'youyuan', port=3306, use_unicode=True)
```

```
    while True:
        # FIFO模式为 blpop, LIFO模式为 brpop，获取键值
        source, data = rediscli.blpop(["youyuan:items"])
        item = json.loads(data)

        try:
            # 使用cursor()方法获取操作游标
            cur = mysqlcli.cursor()
            # 使用execute方法执行SQL INSERT语句
            cur.execute("INSERT INTO beijing_18_25 (username, crawled, age, spider,
header_url, source, pic_urls, monologue, source_url) VALUES
(%s, %s, %s, %s, %s, %s, %s, %s, %s )", [item['username'], item['crawled'], item['age'],
item['spider'], item['header_url'], item['source'], item['pic_urls'], item['monologue'],
item['source_url']])
            # 提交sql事务
            mysqlcli.commit()
            #关闭本次操作
            cur.close()
            print "inserted %s" % item['source_url']
        except MySQLdb.Error,e:
            print "Mysql Error %d: %s" % (e.args[0], e.args[1])

if __name__ == '__main__':
    main()
```

麦田二手房租房信息

maitian

Scrapy-Redis 分布式抓取麦田二手房租房信息与数据分析
https://www.jianshu.com/p/f1b3416b359f

十六 使用 docker 构建简单分布式爬虫

课程简介
• 16.1 分布式爬虫常用的架构方式详解
• 16.2 方案的选择（Linux+Docker+Redis+Urllib+MySQL）
• 16.3 Docker 技术基础
• 16.4 Redis 技术基础
• 16.5 准备基础镜像并做好基础准备（装好基本的 Python3，MySQL，Redis 服务）
• 16.6 配置好中心节点服务器
• 16.7 17K 小说网站分析不对应分布式爬虫项目的编写
• 16.8 将分布式爬虫项目部署到某个子节点中并调试

• 16.9 批量建立子节点服务器实现分布式爬取实战及效果展示

16.1 分布式爬虫常用的架构方式详解

简单来说，所谓分布式爬虫就是应用多台机器同时实现爬虫任务，这多台机器上的爬虫，整体称为分布式爬虫，可以知道，分布式爬虫是区别于单机爬虫的一种架构。
分布式爬虫的难点不在于爬虫本身，而在于多台机器之间的通信，因为我们爬虫还是之前学过的爬虫技术，但是应用的环境是不一样的，效率也是大不相同的。

分布式爬虫有非常多的架构方式，常见的有以下几种方式：
1, 多台真实机器+爬虫（如 Urllib, Scrapy 等）+任务共享中心
2, 多台虚拟机器（或者部分虚拟部分真实）+爬虫（如 Urllib, Scrapy 等）+任务共享中心
3, 多台容器级虚拟化机器（或者部分真实机器）+爬虫（如 Urllib, Scrapy 等）+任务共享中心
其中，任务的共享中心可以采用 redis 技术进行实现。
其中，数据的存储方面，可以存储在节点里面，也可以存储到中心机器的数据库中。
其中，上面 3 中，常用的技术手段主要有：
1, Docker+Redis+Urllib+(MySQL)
2, Docker+Redis+Scrapy+Scrapy-Redis+（MySQL）

16.2 方案的选择（Linux+Docker+Redis+Urllib+MySQL）

本章我们需要为大家介绍分布式爬虫的构建与运作，目的是希望大家能够清楚分布式爬虫的运作与架构的基本思路。因为生产环境常将爬虫部署在 Linux 服务器中，故而我们同样会基于 Linux 服务器进行部署。为了节约成本，我们可以基于 Docker 进行虚拟化，过滤节点使用 Redis 进行，爬虫技术使用 Urllib 实现，数据存储使用 MySQL 进行。
学习本章课前，大家最好有简单的 Linux 基础，包括 Linux 环境的搭建, Linux 常用命令使用等。

16.3 Docker 技术基础

安装Docker

Docker 的启动, Docker 镜像搜索, Docker 镜像下载, Docker 容器的创建, Docker 容器的查看, Docker 容器的命名, Docker 容器的启动，进入与不停止退出，基于容器封装为 Docker 镜像, Docker 容器间通信实现，容器网络基础。

学习分布式爬虫，这些基础够用了，如果想深入学习 Docker（非必须），也可以关注我 Docker 方面的课程（如果不做云计算或运维这方面，完全可以不必要深入学习 Docker）。

Get Docker CE for Ubuntu

Estimated reading time: 12 minutes

To get started with Docker CE on Ubuntu, make sure you meet the prerequisites, then install Docker.

Prerequisites

OS requirements

To install Docker CE, you need the 64-bit version of one of these Ubuntu versions:

- Bionic 18.04 (LTS)
- Artful 17.10
- Xenial 16.04 (LTS)
- Trusty 14.04 (LTS)

Docker CE is supported on Ubuntu on x86_64, armhf, s390x (IBM Z), and ppc64le (IBM Power) architectures.

ppc64le and s390x limitations: Packages for IBM Z and Power architectures are only available on Ubuntu Xenial and above.

Uninstall old versions

Older versions of Docker were called docker or docker-engine. If these are installed, uninstall them:

```
sudo apt-get remove docker docker-engine docker.io
```

It’s OK if apt-get reports that none of these packages are installed.

The contents of /var/lib/docker/, including images, containers, volumes, and networks, are preserved.

The Docker CE package is now called docker-ce.

Install Docker CE

You can install Docker CE in different ways, depending on your needs:

- Most users set up Docker's repositories and install from them, for ease of installation and upgrade tasks. This is the recommended approach.
- Some users download the DEB package and install it manually and manage upgrades completely manually. This is useful in situations such as installing Docker on air-gapped systems with no access to the internet.

Install using the repository

Before you install Docker CE for the first time on a new host machine, you need to set up the Docker repository. Afterward, you can install and update Docker from the repository.

SET UP THE REPOSITORY

Update the apt package index:

```
sudo apt-get update
```

Install packages to allow apt to use a repository over HTTPS:

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

首先安装依赖：

```
sudo apt-get install apt-transport-https ca-certificates curl gnupg2 software-properties-common
```

Add Docker's official GPG key:
信任 Docker 的 GPG 公钥：

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Verify that you now have the key with the fingerprint 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88, by searching for the last 8 characters of the fingerprint.

```
sudo apt-key fingerprint 0EBFCD88
```

```
pub     4096R/0EBFCD88 2017-02-22
        Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid                  Docker Release (CE deb) <docker@docker.com>
sub     4096R/F273FCD8 2017-02-22
```

Use the following command to set up the stable repository. You always need the stable repository, even if you want to install builds from the edge or test repositories as well. To add the edge or test repository, add the word edge or test (or both) after the word stable in the commands below.
对于 amd64 架构的计算机，添加软件仓库：

```
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

Note: Starting with Docker 17.06, stable releases are also pushed to the **edge** and **test** repositories.

Learn about **stable** and **edge** channels.

INSTALL DOCKER CE

Update the apt package index.

```
sudo apt-get update
```

Install the latest version of Docker CE, or go to the next step to install a specific version:

```
sudo apt-get install docker-ce
```

Install a specific version of Docker CE

Got multiple Docker repositories?
If you have multiple Docker repositories enabled, installing or updating without specifying a version in the apt-get install or apt-get update command always installs the highest possible version, which may not be appropriate for your stability needs.

To install a specific version of Docker CE, list the available versions in the repo, then select and install:

a. List the versions available in your repo:

```
apt-cache madison docker-ce

docker-ce | 18.03.0~ce-0~ubuntu | https://download.docker.com/linux/ubuntu
xenial/stable amd64 Packages
```

b. Install a specific version by its fully qualified package name, which is package name (docker-ce) “=” version string (2nd column), for example, docker-ce=18.03.0~ce-0~ubuntu.

```
$ sudo apt-get install docker-ce=<VERSION>
```

The Docker daemon starts automatically.

Verify that Docker CE is installed correctly by running the hello-world image.

```
$ sudo docker run hello-world
```

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.

Docker CE is installed and running. The docker group is created but no users are added to it. You need to use sudo to run Docker commands. Continue to Linux postinstall to allow non-privileged users to run Docker commands and for other optional configuration steps.

UPGRADE DOCKER CE

To upgrade Docker CE, first run sudo apt-get update, then follow the installation instructions, choosing the new version you want to install.

Uninstall Docker CE

Uninstall the Docker CE package:

```
$ sudo apt-get purge docker-ce
```

Images, containers, volumes, or customized configuration files on your host are not automatically removed. To delete all images, containers, and volumes:

```
$ sudo rm -rf /var/lib/docker
```

You must delete any edited configuration files manually.

Get Docker CE for CentOS

To get started with Docker CE on CentOS, make sure you meet the prerequisites, then install Docker.

Prerequisites

OS requirements

To install Docker CE, you need a maintained version of CentOS 7. Archived versions aren't supported or tested.
The centos-extras repository must be enabled. This repository is enabled by default, but if you have disabled it, you need to re-enable it.
https://wiki.centos.org/AdditionalResources/Repositories
The overlay2 storage driver is recommended.

Uninstall old versions

Older versions of Docker were called docker or docker-engine. If these are installed, uninstall them, along with associated dependencies.

```
$ sudo yum remove docker \
                  docker-client \
                  docker-client-latest \
                  docker-common \
                  docker-latest \
                  docker-latest-logrotate \
                  docker-logrotate \
                  docker-selinux \
                  docker-engine-selinux \
                  docker-engine
```

It's OK if yum reports that none of these packages are installed.
The contents of /var/lib/docker/, including images, containers, volumes, and networks, are preserved. The Docker CE package is now called docker-ce.

Install Docker CE

You can install Docker CE in different ways, depending on your needs:

- Most users set up Docker's repositories and install from them, for ease of installation and upgrade tasks. This is the recommended approach.
- Some users download the RPM package and install it manually and manage upgrades completely manually. This is useful in situations such as installing Docker on air-gapped systems with no access to the internet.

- In testing and development environments, some users choose to use automated convenience scripts to install Docker.

Install using the repository

Before you install Docker CE for the first time on a new host machine, you need to set up the Docker repository. Afterward, you can install and update Docker from the repository.

SET UP THE REPOSITORY

Install required packages. yum-utils provides the yum-config-manager utility, and device-mapper-persistent-data and lvm2are required by the devicemapper storage driver.

```
$ sudo yum install -y yum-utils \
  device-mapper-persistent-data \
  lvm2
```

```
sudo yum install -y yum-utils device-mapper-persistent-data lvm2
```

Use the following command to set up the stable repository. You always need the stable repository, even if you want to install builds from the edge or test repositories as well.

```
$ sudo yum-config-manager \
      --add-repo \
      https://download.docker.com/linux/centos/docker-ce.repo
```

Optional: Enable the edge and test repositories. These repositories are included in the docker.repo file above but are disabled by default. You can enable them alongside the stable repository.

```
$ sudo yum-config-manager --enable docker-ce-edge
$ sudo yum-config-manager --enable docker-ce-test
```

You can disable the edge or test repository by running the yum-config-manager command with the --disable flag. To re-enable it, use the --enable flag. The following command disables the edge repository.

```
$ sudo yum-config-manager --disable docker-ce-edge
```

Note: Starting with Docker 17.06, stable releases are also pushed to the edge and test repositories.
Learn about stable and edge builds.

INSTALL DOCKER CE

Install the latest version of Docker CE, or go to the next step to install a specific version:

```
$ sudo yum install docker-ce
```

If prompted to accept the GPG key, verify that the fingerprint matches 060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35, and if so, accept it.

Got multiple Docker repositories?
If you have multiple Docker repositories enabled, installing or updating without specifying a version in the yum installor yum update command always installs the highest possible version, which may not be appropriate for your stability needs.
Docker is installed but not started. The docker group is created, but no users are added to the group.

To install a specific version of Docker CE, list the available versions in the repo, then select and install:

a. List and sort the versions available in your repo. This example sorts results by version number, highest to lowest, and is truncated:

```
$ yum list docker-ce --showduplicates | sort -r

docker-ce.x86_64                  18.03.0.ce-1.el7.centos                  docker-ce-stable
```

The list returned depends on which repositories are enabled, and is specific to your version of CentOS (indicated by the .el7suffix in this example).
b. Install a specific version by its fully qualified package name, which is the package name (docker-ce) plus the version string (2nd column) up to the first hyphen, separated by a hyphen (-), for example, docker-ce-18.03.0.ce.

```
$ sudo yum install docker-ce-<VERSION STRING>
```

Docker is installed but not started. The docker group is created, but no users are added to the group.
Start Docker.

```
$ sudo systemctl start docker
```

Verify that docker is installed correctly by running the hello-world image.

```
$ sudo docker run hello-world
```

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.

Docker CE is installed and running. You need to use sudo to run Docker commands. Continue to Linux postinstall to allow non-privileged users to run Docker commands and for other optional configuration steps.

UPGRADE DOCKER CE

To upgrade Docker CE, follow the installation instructions, choosing the new version you want to install.

上海大学Docker Community Edition镜像使用帮助

https://mirrors.shu.edu.cn/help/docker-ce.html

注意：本镜像只提供 Debian/Ubuntu/Fedora/CentOS/RHEL 的 docker 软件包，非 dockerhub

Debian/Ubuntu用户

如果你过去安装过 docker，先删掉：

```
sudo apt-get remove docker docker-engine docker.io
```

首先安装依赖：

```
sudo apt-get install apt-transport-https ca-certificates curl gnupg2 software-properties-common
```

信任 Docker 的 GPG 公钥：

```
curl -fsSL https://mirrors.shu.edu.cn/docker-ce/linux/debian/gpg | sudo apt-key add -
```

对于 amd64 架构的计算机，添加软件仓库：

```
sudo add-apt-repository \
        "deb [arch=amd64] https://mirrors.shu.edu.cn/docker-ce/linux/debian \
        $(lsb_release -cs) \
        stable"
```

如果你是树莓派或其他 ARM 架构计算机，请运行：

```
echo "deb [arch=armhf] https://mirrors.shu.edu.cn/docker-ce/linux/debian \
```

```
    $(lsb_release -cs) stable" | \
sudo tee /etc/apt/sources.list.d/docker.list
```

最后安装

```
sudo apt-get update
sudo apt-get install docker-ce
```

Fedora/CentOS/RHEL

如果你之前安装过 docker，请先删掉：

sudo yum remove docker docker-common docker-selinux docker-engine

安装一些依赖

sudo yum install -y yum-utils device-mapper-persistent-data lvm2

根据你的发行版下载 repo 文件： 如果你是 centos：

wget -O /etc/yum.repos.d/docker-ce.repo https://mirrors.shu.edu.cn/docker-ce/linux/centos/docker-ce.repo
sudo sed -i 's+download.docker.com+mirrors.shu.edu.cn/docker-ce+' /etc/yum.repos.d/docker-ce.repo

如果你是 fedora：

wget -O /etc/yum.repos.d/docker-ce.repo https://mirrors.shu.edu.cn/docker-ce/linux/fedora/docker-ce.repo
sudo sed -i 's+download.docker.com+mirrors.shu.edu.cn/docker-ce+' /etc/yum.repos.d/docker-ce.repo

最后安装：
sudo yum makecache fast
sudo yum install docker-ce

USTC Docker 镜像使用帮助

使用说明
新版的 Docker 使用
(Linux)
/etc/docker/daemon.json
或者(Windows)
%programdata%\docker\config\daemon.json

来配置 Daemon。

请在该配置文件中加入（没有该文件的话，请先建一个）：

```
{
  "registry-mirrors": ["https://docker.mirrors.ustc.edu.cn"]
}
```

Docker Daemon configuration file 文档
https://docs.docker.com/engine/reference/commandline/dockerd/#/daemon-configuration-file

Docker for Windows 文档
https://docs.docker.com/docker-for-windows/#/docker-daemon

以下是一些过时的配置方法
在 Docker 的启动参数中加入:
--registry-mirror=https://docker.mirrors.ustc.edu.cn

Ubuntu 用户（包括使用 systemd 的 Ubuntu 15.04）可以修改 /etc/default/docker 文件，加入如下参数：
DOCKER_OPTS="--registry-mirror=https://docker.mirrors.ustc.edu.cn"

其他 systemd 用户可以通过执行 sudo systemctl edit docker.service 来修改设置，覆盖默认的启动参数:

```
[Service]
ExecStart=
ExecStart=/usr/bin/docker -d -H fd:// --registry-mirror=https://docker.mirrors.ustc.edu.cn
```

添加到用户组（可选项）

添加到用户组（so easy）

```
sudo groupadd docker
sudo usermod -aG docker david
```

注销系统重新进入系统，就可以直接使用 docker 开头了。
如果不添加到用户组会发生什么呢？
如果直接运行:

```
docker run hello-world
```

你会发现下面的错误:

Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock:
Get http://%2Fvar%2Frun%2Fdocker.sock/v1.30/containers/json: dial unix /var/run/docker.sock: connect: permission denied

这是因为：
docker 守护程序绑定到 Unix 套接字而不是 TCP 端口。默认情况下，Unix 套接字由用户 root 拥有，其他用户只能使用 sudo 访问它。 docker 守护程序始终以 root 用户身份运行。 如果您不想在使用 docker 命令时使用 sudo，请创建名为 docker 的 Unix 组，并将用户添加到该组。当 docker 守护进程启动时，它会使 Docker 组的 Unix 套接字的所有权读/写。

linux 下安装一条命令即可

curl -sSL https://get.daocloud.io/docker | sh
这条命令在 ubuntu 14.04 和 ubuntu 16.04 都可以成功安装 docker。

安装完成功后，可能会提示你这样的信息:

If you would like to use Docker as a non-root user, you should now consider adding your user to the "docker" group with something like:

sudo usermod -aG docker vagrant

Remember that you will have to log out and back in for this to take effect!
vagrant 是你的用户名，可能你的用户名跟我的不一样。

意思就是说，你可以把当前用户加入到 docker 组，以后要管理 docker 就方便多了，不然你以后有可能要使用 docker 命令前，要在前面加 sudo。

如果没加 sudo 就是类似这样的提示:

Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get
http://%2Fvar%2Frun%2Fdocker.sock/v1.26/containers/json: dial unix /var/run/docker.sock: connect: permission denied

不过执行了 sudo usermod -aG docker vagrant 之后，你再重新登录(ssh)，就可以免去加 sudo。

安装成功，需要把 docker 这个服务启动起来：

如果是 ubuntu 14.04 的系统，它会自动启动，你也可以使用下面的命令来启动。
$ sudo /etc/init.d/docker start

如果是 ubuntu 16.04 的系统，就用下面的命令：

$ sudo systemctl status docker.service

使用加速器提升获取Docker官方镜像的速度

国内建议可以使用一个加速器!
获得加速器的方法步骤:
进入网址
https://account.aliyun.com/login/login.htm?oauth_callback=https%3A%2F%2Fcr.console.aliyun.com%2F&lang=zh#/accelerator
用自己的淘宝帐号登录进去，新用户跳过所有的步骤，进入到 docker 镜像仓库，点击下面的加速器，自动获得加速器，

https://cr.console.aliyun.com/cn-shanghai/mirrors

加速器地址
https://dlfa9xic.mirror.aliyuncs.com

方法错误
"""
在 /etc/systemd/system/multi-user.target.wants/docker.service 文件中 添加你在阿里云镜像 https://cr.console.aliyun.com/#/accelerator 的上专属地址：
sudo vim /etc/systemd/system/multi-user.target.wants/docker.service

```
ExecStart=/usr/bin/dockerd -H fd:// --registry-mirror=
https://dlfa9xic.mirror.aliyuncs.com
ExecReload=/bin/kill -s HUP $MAINPID
LimitNOFILE=1048576
```

"""

正确方法如下

操作文档

Ubuntu, CentOS

1. 安装 / 升级 Docker 客户端
推荐安装 1.10.0 以上版本的 Docker 客户端，参考文档 docker-ce
https://yq.aliyun.com/articles/110806?spm=5176.8351553.0.0.44e41991l15Vnz

2. 配置镜像加速器

针对 Docker 客户端版本大于 1.10.0 的用户

您可以通过修改 daemon 配置文件/etc/docker/daemon.json 来使用加速器

```
sudo mkdir -p /etc/docker

sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://dlfa9xic.mirror.aliyuncs.com"]
}
EOF

sudo systemctl daemon-reload
sudo systemctl restart docker
```

或者:
sudo vim /etc/docker/daemon.json
添加如下内容

```
{
  "registry-mirrors": ["https://dlfa9xic.mirror.aliyuncs.com"]
}
```

Mac

1. 安装 / 升级 Docker 客户端
对于 10.10.3 以下的用户 推荐使用 Docker Toolbox
http://mirrors.aliyun.com/docker-toolbox/mac/docker-toolbox/?spm=5176.8351553.0.0.44e41991l15Vnz

Mac 安装文件：http://mirrors.aliyun.com/docker-toolbox/mac/docker-toolbox/

对于 10.10.3 以上的用户 推荐使用 Docker for Mac
http://mirrors.aliyun.com/docker-toolbox/mac/docker-for-mac/?spm=5176.8351553.0.0.44e41991l15Vnz

Mac 安装文件：http://mirrors.aliyun.com/docker-toolbox/mac/docker-for-mac/

2. 配置镜像加速器
针对安装了 Docker Toolbox 的用户，您可以参考以下配置步骤：
创建一台安装有 Docker 环境的 Linux 虚拟机，指定机器名称为 default，同时配置 Docker 加速器地址。

docker-machine create --engine-registry-mirror=https://dlfa9xic.mirror.aliyuncs.com -d virtualbox default

查看机器的环境配置，并配置到本地，并通过 Docker 客户端访问 Docker 服务。

docker-machine env default
eval "$(docker-machine env default)"
docker info

针对安装了 Docker for Mac 的用户，您可以参考以下配置步骤：
右键点击桌面顶栏的 docker 图标，选择 Preferences ，在 Daemon 标签（Docker 17.03 之前版本为 Advanced 标签）下的 Registry mirrors 列表中将

https://dlfa9xic.mirror.aliyuncs.com 加到"registry-mirrors"的数组里，点击 Apply & Restart 按钮，等待 Docker 重启并应用配置的镜像加速器。

3. 相关文档
Docker 命令参考文档
https://docs.docker.com/engine/reference/commandline/cli/?spm=5176.8351553.0.0.44e41991l15Vnz

Dockerfile 镜像构建参考文档
https://docs.docker.com/engine/reference/builder/?spm=5176.8351553.0.0.44e41991l15Vnz

Windows

1. 安装 / 升级 Docker 客户端
对于 Windows 10 以下的用户，推荐使用 Docker Toolbox

Windows 安装文件：http://mirrors.aliyun.com/docker-toolbox/windows/docker-toolbox/

对于 Windows 10 以上的用户 推荐使用 Docker for Windows

Windows 安装文件：http://mirrors.aliyun.com/docker-toolbox/windows/docker-for-windows/

2. 配置镜像加速器
针对安装了 Docker Toolbox 的用户，您可以参考以下配置步骤：
创建一台安装有 Docker 环境的 Linux 虚拟机，指定机器名称为 default，同时配置 Docker 加速器地址。

docker-machine create --engine-registry-mirror=https://dlfa9xic.mirror.aliyuncs.com -d virtualbox default

查看机器的环境配置，并配置到本地，并通过 Docker 客户端访问 Docker 服务。

```
docker-machine env default
eval "$(docker-machine env default)"
docker info
```

针对安装了 Docker for Windows 的用户，您可以参考以下配置步骤：
在系统右下角托盘图标内右键菜单选择 Settings，打开配置窗口后左侧导航菜单选择 Docker Daemon。编辑窗口内的 JSON 串，填写下方加速器地址：

```
{
  "registry-mirrors": ["https://dlfa9xic.mirror.aliyuncs.com"]
}
```

编辑完成后点击 Apply 保存按钮，等待 Docker 重启并应用配置的镜像加速器。

注意
Docker for Windows 和 Docker Toolbox 互不兼容，如果同时安装两者的话，需要使用 hyperv 的参数启动。

docker-machine create --engine-registry-mirror=https://dlfa9xic.mirror.aliyuncs.com -d hyperv default
Docker for Windows 有两种运行模式，一种运行 Windows 相关容器，一种运行传统的 Linux 容器。同一时间只能选择一种模式运行。

3. 相关文档
Docker 命令参考文档
https://docs.docker.com/engine/reference/commandline/cli/?spm=5176.8351553.0.0.44e41991l15Vnz
Dockerfile 镜像构建参考文档
https://docs.docker.com/engine/reference/builder/?spm=5176.8351553.0.0.44e41991l15Vnz

Docker的使用

centos7 中 docker 的配置

```
cd /
yum -y install docker
```

docker 的启动

systemctl start docker
输入 docker 查看是否成功
docker

docker 镜像的搜索. 镜像可以看成是母版, 是他人已经开发好的各种环境, 使用镜像就可以很方便的开发出自己的容器级虚拟机
搜索 ubuntu 相关的镜像, 并按 starts 排名
docker search ubuntu
搜索与 python 相关的镜像
docker search python

下载镜像并指定版本
docker pull ubuntu:14.04
docker pull ubuntu:16.04
也可以使用国内的镜像下载
http://get.daocloud.io
http://hub.daocloud.io
docker pull daocloud.io/library/ubuntu:latest

docker 镜像的查看, REPOSITORY, TAG, IMAGE ID, CREATED, SIZE.
docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	16.04	52b10959e8aa	11 days ago	115MB
hello-world	latest	2cb0d9787c4d	7 weeks ago	1.85kB

基于镜像创建容器, 容器实际上就是虚拟机, 把一台 linux 服务器虚拟成多台机器 , 在这些机器上运行爬虫, 就实现了分布式的功能
tid 即上面显示的 IMAGE ID 52b10959e8aa 的前 4 位
docker run -tid 52b1
e186ec893ea6d4a85b629957e994253ceabff69a77a40e7ec2d1838605c19287
创建的容器以 e186 开头

进入容器/虚拟机
docker attach e186
显示如下开头的提示即表示成功了, 有时候会长时间不显示, 再输入回车就显示出来了.
root@e186ec893ea6:/ #

使用如下命令进入 docker 容器, 使用 exit 命令退出时就不会关闭容器了, 而是切换到物理机.
docker exec -it scrapy00 /bin/bash

退出容器. 如果使用 exit 退出容器, 就会退出的同时中止容器的运行. 一般使用快捷键 Ctrl+P+Q 来退出容器但不停止运行, 按下回车确定, 就退出到物理机.
查看所有容器的状态, 其中的 status 为 Up 3 minutes, 就表示是处于运行的状态.

docker ps -a

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e186ec893ea6	52b1	/bin/bash	4 minutes ago	Up 4 minutes		stupefied_jang
ec4788cef542	hello-world	/hello	11 minutes ago	Exited (0) 11 minutes ago		fervent_panini
d3e000965856	hello-world	/hello	11 minutes ago	Exited (0) 11 minutes ago		hardcore_montalcini

容器的命名, 在创建容器时没有给容器命名, 就会默认使用随机的名称. 例如创建出来的容器 id 为 d4ce
docker run --name scrapy_01 -tid 52b1

再次查看所有容器的状态. 就可以在 NAMES 中看到它的名称了
docker ps -a

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
b671f532a842	52b1	/bin/bash	7 seconds ago	Up 6 seconds		scrapy_01
e186ec893ea6	52b1	/bin/bash	7 minutes ago	Up 7 minutes		stupefied_jang
ec4788cef542	hello-world	/hello	13 minutes ago	Exited (0) 13 minutes ago		fervent_panini
d3e000965856	hello-world	/hello	14 minutes ago	Exited (0) 14 minutes ago		hardcore_montalcini

将容器封装为 docker 镜像. 镜像是母版, 不能够直接修改, 可以基于镜像创建一个容器, 在容器中部署相关的环境, 想要在其它地方使用相同环境的容器, 就可以把已经部署好生产环境的容器封装为一个镜像. 就可以通过封装的镜像来创建新的容器或虚拟机.
把 db8f 这个容器封装为一个镜像, 镜像名为 ubuntu_deployed, 版本号为 v1.
docker commit db8f ubuntu_deployed:v1

查看所有的镜像
docker images

docker 容器间的通信. 创建 1 个新的名为 scrapy_02 的容器, 并通过--link 参数实现与 name 为 scapy_01, id 为 d4ce 的容器之间的通信. 如创建出来的容器 id 为 69cc
docker run --name scrapy_02 --link scrapy_01 -tid 52b1

```
# 查看所有的容器
docker ps -a

# 如何知道 id 为 d4ce, name 为 scrapy_01 的容器与 id 为 69cc, name 为 scrapy_02 的容器之间能够通信呢. 进入 69cc 的容器, 查看它的网络配置文件.
docker attach 69cc

# 查看网络配置, 就会在其中看到 scrapy_01 和 scrapy_02 的网络地址. 如 scrapy_01 的 ip 地址为 172.17.0.3.
cat /etc/hosts

# 使用 ping 命令来测试两台容器之间能否正常通信
# 升级 apt-get 命令
apt-get update
# 安装 ping 命令, 会出现 2 个包, 选择其中的一个包进行安装.
apt-get -y install ping
apt-get -y install inetutils-ping
# 使用 ping 命令来测试不同容器间能否通信
ping 172.17.0.3

Ctrl + P + Q 退出但不关闭容器
```

16.4 Redis 技术基础

在 Ubuntu 中安装 Redis, Redis 的启动与连接, Redis 的相关使用基础。Redis 是基于内存的数据库, 性能很高, 可以作为分布式爬虫任务分配与调度的中心节点.

创建新的容器, 作为中心控制节点, 部署 Redis 数据库

```
# 基于 452a 镜像创建一个名为 scrapy_00 的容器, 并把它与 scrapy_01 容器连接起来, 用来部署 Redis 数据库. 如新的容器 id 为 2242
docker run -tid --name scrapy_00 --link scrapy_01 452a

# 进入新的容器
docker attach 2242

# 安装 redis

# 升级 apt-get
apt-get update
# 安装 redis 服务器
apt-get -y install redis-server
# 启动 redis 服务器
```

/etc/init.d/redis-server restart
或
/etc/init.d/redis-server start
测试 redis 服务器是否成功运行，使用 redis-cli 客户端连接本机的 redis 服务器
redis-cli -h 127.0.0.1 -p 6379

redis 基础操作
redis 中数据保存的格式，常见的有两种，"键-值"，"键-域-值".
键值对的方式就是类似于字典的格式
键-域-值的方式就是类似于 hash 的格式．如有一个名为 title 的键，其下有多种不同的域，每一种域都有与其对应的值．如 name: david, sex: male, age: 20.

在 redis 客户端中进行操作

插入键值对，创建一个 name 的键，它的值为 weiwei
set name weiwei

取出来某个键对应的值
get name

set sex male
get sex

批量的插入键值对
mset a 199 b 299 c 399 xyz abc
get c

批量获取键对应的值
mget a c xyz

插入键域值
hset title name hello

取出某个键下某个域对应的值
hget title name

hset title sex femal
hget title sex

批量插入键域值
hmset title link http://abc.com id 100 grade three

获取某个键下面所有的域和对应的值
hgetall title

16.5 准备基础镜像并做好基础准备

装好基本的 Python3，MySQL，Redis 服务

基础镜像我们选择 ubuntu:14.04，有了基础镜像之后，我们必须在基础镜像里面安装好 Python3, pip, MySQL, Redis，当然你也可以安装 Scrapy，这些老师已经做好，大家可以直接使用提供的镜像，如果你想自己动手实现一遍，也是可以的，具体实现安装 Python3 与 Scrapy 请参考我们书籍《精通 Python 网络爬虫》机械工业出版社第 147 页，坑比较多，所以这里的安装不建议参考网络，请直接参考书籍，注意，在 Ubuntu 系统中，yum 命令请换成 apt-get 命令，其他都差不多。
安装 MySQL 与 Redis 比较简单，直接网络搜索 ubuntu 如何安装 MySQL 即可有答案，几行命令搞定，在此不过多介绍。
如上所说，如果你不想做这些琐事，直接使用老师提供的镜像文件即可。

把镜像文件 21mycrawl.tar 上传到 linux 服务器中，可以使用 secureFX 来上传文件

```
# 把 21mycrawl.tar 恢复为镜像
docker load --input 21mycrawl.tar

# 再次查看所有的镜像，假如新的镜像 id 为 3a54
docker images
```

16.6 配置好中心节点服务器

配置中心节点，默认 Redis 与 MySQL 是不支持进程连接的，所以我们需要进行配置。

中心节点主要完成数据的存储和任务的控制，任务控制-通信，数据存储．任务控制使用 redis，数据存储使用 mysql

```
# 查看当前已有的容器
docker ps -a

# 删除所有的容器，删除所有容器.
docker rm -f 2242 69cc d4ce db8f

# 部署中心节点容器，假设创建的容器 id 为 0ed6
docker run -tid center 3a54

# 进入新创建的容器
```

docker attach 0ed6

配置 mysql 和 Redis

开启 mysql
/etc/init.d/mysql restart

修改 mysql 的配置文件
vim /etc/mysql/my.cnf

把 bind-address = 127.0.0.1　这一项注释掉

重启 mysql 服务器
/etc/init.d/mysql/restart

连接到 mysql 服务器
mysql -h 127.0.0.1 -u root -p
weijc7789

创建一个新用户，不能使用 root 账号远程连接 mysql 服务器，会出现权限的问题，所以要创建一个新的用户用于远程连接. 创建一个名为 weiwei 的用户，可以在所有的客户端中使用 weijc7789 这个密码登陆
create user "weiwei"@"%" indentified by "weijc7789"

赋予新创建的用户适当的权限. 赋予用户 weiwei 在所有的数据库中的新建，插入，更新，删除，查询的权限
grant create,insert,update,delete,select on *.* to weiwei;

退出 root 账户
exit

使用新创建的用户进行登录
mysql -h 127.0.0.1 -u weiwei -p
weijc7789

配置 reids 服务器，使其能够支持远程连接
vim /etc/redis/redis.conf

注释掉 bind 127.0.0.1 这一项

重启容器，从容器中跳出到物理机，执行以下命令
docker stop 0ed6
docker start 0ed6

进入容器，重启数据库
docker attach 0ed6

重启 mysql 数据库
/etc/init.d/mysql restart

重启 redis 数据库
/etc/init.d/redis-server restart

使用 ctrl + p + q 不停止退出容器

现在就配置好了中心节点. 下面就可以创建各个子节点并连接中心节点了.

16.7 17K 小说网站分析与对应分布式爬虫项目的编写

基础爬虫的编写

http://www.17k.com/book/2721543.html
http://www.17k.com/book/2743325.html
……

```
import redis
import pymysql
import urllib.request
import re

# 在 python 中使用 redis 这个库来操作 redis 数据库. 设置 redis 的连接, 连接到中心服务器
rconn = redis.Redis("172.17.0.2", "6379")
'''
# 分布式爬虫与普通爬虫的区别在于中心的调度与控制机制不同. 建立标志位, 使用 url 作为键, 以 i 作为域, 以"1"作为值.
url-i-"1"
'''
for i in range(0, 5459058):
    # 先在 redis 中查找"url"键和 i 域对应的值存不存在.
    isdo = rconn.hget("url", str(i))
    # 如果不为 None, 表示已经爬取过了, 就跳过本次循环. 这样就实现了链接的自动过滤
    if (isdo != None):
        continue
```

```
        # 如果数据库中不存在，就把url 键和i 域对应的值标记为"1"，当其它子节点的爬虫在爬取到同样的url 键和i 域时，发现它的值已经被标记为1，就不再进行爬取. 分布式爬虫与普通单机爬虫的区别就在于任务的调度方法的不同. 在分布式爬虫中，任务的调度就可以使用redis 的标志位来解决.
        rconn.hset("url", str(i), "1")

        try:
            # url: "http://www.17k.com/book/2.html"
            data = urllib.request.urlopen("http://www.17k.com/book/" + str(i) + ".html").read().decode("utf-8", "ignore")
        except Exception as err:
            print(str(i) + "----" + str(err))
            continue

        # 提取小说的书名
        pat = '<a class="red" .*?>(.*?)</a>'
        rst = re.compile(pat, re.S).findall(data)
        # 如果没有提取到标题，就跳过本次循环，直接执行下一次循环
        if (len(rst) == 0):
            continue
        name = rst[0]

        print(str(i) + "-----" + str("ok"))
        # 把提取到的标题写入到redis 数据库中. 键为"rst"，域为当前的url 中的数字，值为小说标题
        # 还可以构建mysql 连接对象，把数据写入到mysql 数据库中
        rconn.hset("rst", str(i), str(name))
```

16.8 将分布式爬虫项目部署到某个子节点中并调试

```
# 创建一个能与中心节点进行通信的容器，如创建的子容器 id 为 60d3
docker run -tid --name child_01 --link center 3a54
# 进入子节点
docker attach 60d3

# 查看网络配置，可以看到 center 节点的 ip 地址为 172.17.0.2
cat /etc/hosts

# 测试由子容器到中心节点容器的 mysql 连接.
mysql -h 172.17.0.2 -u weiwei -p
```

测试由子容器到中心节点容器的 redis 连接
redis-cli -h 172.17.0.2

查看当前的 python 版本
python --version

把爬虫部署到子节点容器中. 注意要把 redis 的连接地址修改为中心节点的 ip 地址, 即为 172.17.0.2

测试爬虫能不能正常运行
python mycrawl.py

16.9 批量建立子节点服务器实现分布式爬取实战及效果展示

创建多台 docker 容器

docker run -tid --name child_02 --link center 3a54
docker run -tid --name child_03 --link center 3a54
docker run -tid --name child_04 --link center 3a54
docker run -tid --name child_05 --link center 3a54

依次进入每个容器中进行部署并测试爬虫, 确保其正常运行.
docker attach child_02

运行爬虫, 然后使用 ctrl+q+p 退出但不中断容器的运行. 每个容器中都执行同样的操作. 也可以使用 python 自动化运维来实现爬虫的批量部署.

当所有容器中的爬虫都已经开始运行之后回到每个服务器中查看爬虫的运行状态. 在运行一段时间之后, 跳过了很多空白的小说之后, 就开始爬取到大量的数据. 并且每个子容器中都不会爬取重复的数据, 这就得益于每个子容器与中心容器之间的通信.

也可以进行到 center 容器中, 查看 redis 数据库中的数据.
docker attach center

当提取小说的标题成功后, 会以"rst"键, str(i) 为域, 小说的标题 name 为值保存到 redis 数据库中.
redis-cli -h 127.0.0.1 -p 6379
取出来所有的 rst 键的数据
hgetall rst

十七，复杂分布式大型网络爬虫的构建与部署实战（在 Linux 环境中进行）

17.1 Scrapy-redis 架构方式详解

Scrapy-redis 是一个可以快速将 Scrapy 项目改造成为分布式爬虫项目的一个模块。其基本的原理跟我们上一章介绍的是一致的，但是其很多功能，比如去重等都帮我们实现了，Scrapy-redis 默认也是采用 Reids 进行通信，当然你可以将其更改为布隆过滤器。
接下来我们为大家安装 Scrapy-redis，并简单介绍该模块。
Scrapy-redis 开源代码地址是：https://github.com/rmax/scrapy-redis

pip install scrapy-redis

安装位置为 python3X/Lib/site-packages/scrapy_redis

pipelines.py, 把数据也保存到 redis 中, 但是在实际中很少会使用到, 因为内存一般都是有限的, 把海量的数据都写入到内存中就会造成内存资源的大大浪费. queue.py 是队列的实现方式. 有 3 种队列的实现方式.

SpiderQueue = FifoQueue # 先进先出队列
SpiderStack = LifoQueue # 栈. 后进先出的队列
SpiderPriorityQueue = PriorityQueue # 优先队列, 默认的队列方式.

在 scheduler.py 中实现任务的调度.

在把 scarpy 爬虫改造为分布式爬虫的过程中, 要使用到 dupefilter.py 和 scheduler.py 这两个文件.

17.2 如何构建 Scrapy-redis 分布式爬虫实战

先正常构建 Scrapy 项目，然后将 Scrapy-redis 整合进正常 Scrapy 项目中，最后进行分布式部署。

其中，分布式部署包括：
中心节点安装 redis, mysql

各子节点均安装 python, scrapy, scrapy-redis, Python 的 redis 模块, 与 pymysql 模块.
将修改好的分布式爬虫项目部署到各子节点
各子节点分别运行分布式爬虫项目

17.3 通过 Scrapy-redis 实现 17K 小说数据分布式爬虫项目实战

前提条件：
在中心节点和子节点上安装好 Python3、pip3、Scrapy，若不清楚怎么安装，可以参考我的书籍《精通 Python 网络爬虫》第 147 页实现。
随后先进行基础环境配置，再编写 Scrapy 爬虫，再整改为 Scrapy-redis 分布式爬虫，最后部署、运行。

有两台电脑, 一台是 windows, 一台是 linux, 两台都要作为子节点, 中心节点共用其中的某一台服务器. 这里部署到 linux 中.

```
tree example-project /F
Folder PATH listing for volume WinD
Volume serial number is 00000001 0000:4823

\EXAMPLE-PROJECT
|       docker-compose.yml
|       Dockerfile
|       process_items.py
|       README.rst
|       requirements.txt
|       scrapy.cfg
|
\---example
    |       items.py
    |       pipelines.py
    |       settings.py
    |       __init__.py
    |
    +---spiders
    |   |       dmoz.py
    |   |       mycrawler_redis.py
    |   |       myspider_redis.py
    |   |       __init__.py
    |   |
    |   \---__pycache__
    |           dmoz.cpython-35.pyc
```

```
        mycrawler_redis.cpython-35.pyc
        myspider_redis.cpython-35.pyc
        __init__.cpython-35.pyc

    __pycache__
        pipelines.cpython-35.pyc
        settings.cpython-35.pyc
        __init__.cpython-35.pyc
```

items.py 定义提取的变量

```
# Define here the models for your scraped items
#
# See documentation in:
# http://doc.scrapy.org/topics/items.html

from scrapy.item import Item, Field
from scrapy.loader import ItemLoader
from scrapy.loader.processors import MapCompose, TakeFirst, Join


class ExampleItem(Item):
    name = Field()
    description = Field()
    link = Field()
    crawled = Field()
    spider = Field()
    url = Field()


class ExampleLoader(ItemLoader):
    default_item_class = ExampleItem
    default_input_processor = MapCompose(lambda s: s.strip())
    default_output_processor = TakeFirst()
    description_out = Join()
```

spider 就是一个通用的爬虫.
由于 dmoz 网站已停止服务，把爬虫修改为爬取另外的内容.
http://dmoz-odp.org/
https://dmoztools.net/

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


class DmozSpider(CrawlSpider):
    """Follow categories and extract links."""
    name = 'dmoz'
    allowed_domains = ['iqianyue.com']
```

```
    start_urls = ['http://www.iqianyue.com/articles/25be1d55']

    rules = [
        Rule(LinkExtractor(
            restrict_css=()
        ), callback='parse_directory', follow=True),
    ]

    def parse_directory(self, response):
        for div in response.css('.title-and-desc'):
            yield {
                'name': div.css('.site-title::text').extract_first(),
                'description': div.css('.site-descr::text').extract_first().strip(),
                'link': div.css('a::attr(href)').extract_first(),
            }
```

pipelines.py 也没有太大的变量. 因为没有把数据保存到数据库中. 想要实现把数据写入到 redis 数据库中, 就要加载 scrapy_redis 目录下的 pipelines.py 文件了.

修改的地方就是 settings.py 中的内容.
只需要添加上以下 3 行的内容, 就可以把一个 scrapy 的爬虫修改为分布式的爬虫.

```
# 使用 scrapy_redis 的去重方法
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"
# 使用 scrapy_redis 的任务调度方法
SCHEDULER = "scrapy_redis.scheduler.Scheduler"
# 任务调度器的持久化
SCHEDULER_PERSIST = True
```

完整的 settings.py 文件内容

```
# Scrapy settings for example project
#
# For simplicity, this file contains only the most important settings by
# default. All the other settings are documented here:
#
#        http://doc.scrapy.org/topics/settings.html
#
SPIDER_MODULES = ['example.spiders']
NEWSPIDER_MODULE = 'example.spiders'

USER_AGENT = 'scrapy-redis (+https://github.com/rolando/scrapy-redis)'

# 使用 scrapy_redis 的去重方法
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"
# 使用 scrapy_redis 的任务调度方法
SCHEDULER = "scrapy_redis.scheduler.Scheduler"
```

```
# 任务调度器的持久化
SCHEDULER_PERSIST = True

# scrapy_redis 的任务任务调度优先级
# 默认使用优先的队列调度方式
SCHEDULER_QUEUE_CLASS = "scrapy_redis.queue.SpiderPriorityQueue"
# 先进先出
# SCHEDULER_QUEUE_CLASS = "scrapy_redis.queue.SpiderQueue"
# 先进后出
# SCHEDULER_QUEUE_CLASS = "scrapy_redis.queue.SpiderStack"

ITEM_PIPELINES = {
    'example.pipelines.ExamplePipeline': 300,
    'scrapy_redis.pipelines.RedisPipeline': 400,
}

LOG_LEVEL = 'DEBUG'

# Introduce an artifical delay to make use of parallelism. to speed up the
# crawl.
DOWNLOAD_DELAY = 1
```

首先在本地测试以上爬虫能否正常运行

scrapy crawl dmoz

由于没有开启中心节点，会出现 6379 连接的错误提示．为了进行测试，可以先在 windows 中建立一个 redis 的服务器．运行 redis-server.exe 开启服务器．运行 redis-cli.exe 查看能否正常连接到本地服务器．

在 redis 成功运行之后再次运行爬虫，就可以正常的爬取数据了．现在运行的爬虫就已经是基于分布式架构的爬虫了，每次要爬取某个页面时都会去 redis 中查看是否已经爬取过该网站．

在 redis-cli 中，输入 KEYS *　就可以看到 redis 服务器中所有的键了．
其中的"dmoz:requests", "dmoz:dupefilter"　这 2 个键就是 scrapy_redis 相关的键．

17K 小说的分布式爬虫，把中心节点部署到 linux 服务器中．

在 linux 服务器上布置好开发环境. python3, pip scrapy
scrapy bench 测试爬虫的性能．

在作为中心节点的 linux 服务器中安装 redis-server

yum -y install redis

在中心节点的 linux 中开启 redis 服务

redis-server

执行 windows 中的 redis-cli.exe，测试是否能够连接 linux 中的 redis 服务器.

进入到 redis-cli.exe 所在的目录中

redis-cli -h 45.40.246.26 -p 6379
执行
KEYS *
提示 redis is running in protected mode…. 或者根本无法连接

在中心节点电脑上停止 redis 服务器的运行，修改中心节点中 redis 的配置文件.

vim /etc/redis.conf

注释掉 bind 127.0.0.1 这一项，这样就可以使用远程进行连接了
bind 127.0.0.1
protected-mode 默认为 yes，想要使用远程连接，要设置为 no
protected-mode no

daemonize 表示是否以守护进程模式运行 redis 服务器，默认为 no，如果想要以守护进程的模式进行运行，就修改为 yes. 与远程连接无关.

在中心节点上开启 redis 服务器. 这里加载修改好的配置文件，以防止没有自动加载. 如果修改了以守护进程的模式运行 redis 服务器，就不会跳出 redis 服务器等待连接的画面了.
redis-server /etc/redis.conf

在 windows 客户端中再次连接中心节点的 redis 数据库，执行 KYES *，就能够成功执行了.

如果要把数据保存到中心节点中，还需要在中心节点上安装 mysql 并进行相应的配置.

由于中心节点也做为子节点运行爬虫项目，所以也需要在中心节点上部署 scrapy, scrapy-redis, python 的 redis 模块, python 的 pymysql 模块.

pip install scrapy-redis
pip install redis
pip install pymysql

同样，在 windows 子节点上安装相同的包部署相同的环境.

编写 scrapy 爬虫项目

scrapy startproject my17k
cd my17k
scrapy genspider -t basic my17 17k.com

修改 items.py

```
# -*- coding: utf-8 -*-
import scrapy


class My17KItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    name=scrapy.Field()
```

修改 my17.py 爬虫文件

```
# -*- coding: utf-8 -*-
import scrapy
from my17k.items import My17KItem
from scrapy.http import Request


class Mys1Spider(scrapy.Spider):
    name = 'mys1'
    allowed_domains = ['17k.com']
    start_urls = ['http://www.17k.com/book/1.html']

    def parse(self, response):
        item = My17KItem()
        item["name"] =
response.xpath('//div[@class="infoPath"]/div/a[@class="red"]/text()').extract()
        print("----------:" + str(item["name"]))
        yield item
        # 构造下一页的Request 请求对象，并指定使用parse 解析函数来解析
        for i in range(2, 5739387):
            thisurl = "http://www.17k.com/book/" + str(i) + ".html"
            yield Request(thisurl, callback=self.parse)
```

修改 settings.py 文件，设置不遵守 robots 协议，修改默认的 User-Agent

运行爬虫进行测试.

修改 pipelines.py 文件，把数据保存到中心节点的服务器中.

```
# -*- coding: utf-8 -*-


class My17KPipeline(object):
    def process_item(self, item, spider):
        return item
```

修改 settings.py 文件，把 scrapy 项目修改为分布式项目

```
# Obey robots.txt rules
ROBOTSTXT_OBEY = False

#  修改为分布式的爬虫
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"
SCHEDULER = "scrapy_redis.scheduler.Scheduler"
SCHEDULER_PERSIST = True
# REDIS_URL="redis://user:passwd@IP:port"
# REDIS_URL="redis://@IP:port"
#  配置 redis 的地址和端口，在远程计算机中进行任务的调度和去重
REDIS_URL = "redis://45.40.246.26:6379"
```

```
REDIS_URL = 'redis://username:pass@hostIP:6379'

REDIS_PARAMS = {
    'password': 'redisPasswordTest123456',
    'db': 2
    }
```

在中心节点的服务器中开启 redis 服务器，在客户端运行爬虫，现在爬虫就能够正常运行了．在中心节点中查看 redis 中的数据，看是否生成了 my17:requests 和 my17:dupefilter 这两个键.

把爬虫代码部署到各个节点计算机中，注意各个子节点中必须要首先安装好开发环境.

在各个节点中同时运行 scrapy 项目，各个节点会同时运行一个项目，并且实现了 url 的去重.

17.4 Scrapy-redis 与简单分布式爬虫的对比

	Scrapy-redis 方案	**简单分布式爬虫方案**
优点	封装程度高 开发快 部署方便 体系成熟	便于感受分布式爬虫的运作 代码编写灵活 便于二次开发
缺点	二次开发要求稍高	开发需要投入较多精力 运行效率需要自己优化

17.5 Scrapy-redis 分布式爬虫项目的管理实战

接下来为大家介绍如何进行 Scrapy-redis 分布式爬虫项目的管理。
利器：
Scrapyd
方案：
中心统一，整合 Scrapyd，使用 scrapyd-deploy 部署爬虫，使用 curl 运行爬虫

17.6 大型分布式爬虫架构的基本方案（补充）

为了让大家在工作的时候更容易解决相关问题，在此为大家补充一些大型分布式爬虫构建的相关方案，如果看不懂，补充部分暂时可以不看，非必须掌握。
先为大家介绍大型分布式爬虫架构的基本方案。

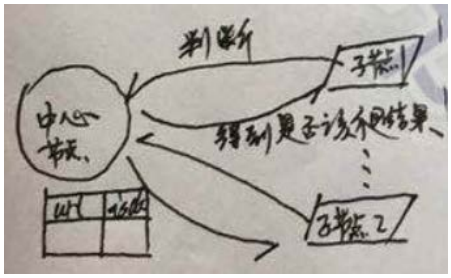
python 实现分布式爬虫有哪些方案与选型?

1. 网络结构问题

方式	特点
多硬件通信	性能高, 成本高
传统虚拟化	成本降低
容器虚拟化	成本低, 性能高, 但中心节点尽量不要使用容器虚拟化. 因为容器虚拟化容易崩溃
多硬件 + 传统虚拟化	成本适中, 性能更优
多硬件 + 容器虚拟化	成本低, 性能最大利用 (推荐). 数据中心采用多硬件, 爬虫采用容器虚拟化

2. 任务调度方案

A. 中心主控法



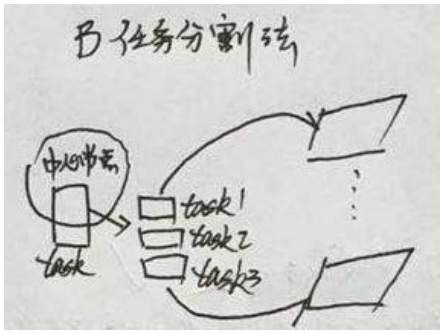
中心节点进行任务的统一调度和去重. 在子节点爬虫运行的时候, 都通过中心节点进行通信, 中心节点给子节点返回任务, 确定哪个可以爬哪个已经爬过了.

中心节点可以通过标志位实现任务的去重和调度. 如设置 url 键和 isdo 域, 如果 isdo 被设置为 1, 就表示已经爬取过了. 如果 isdo 的值为 None, 就表示没有爬取过.

中心节点主要实现去重和过滤. 可以使用集合, mysql, redis 和布隆过滤器来实现.

方式		性能
集合标志		由低到高
mysql 标志		
redis 标志	√	
布隆过滤器	√	

B. 任务分割法



以 17k 小说爬虫为例, 不同的小说只是后面的数字的变动, 可以构造出所有的 url 地址, 先在中心节点上对任务进行分割, 把得到的小任务分别交给不同的子节点上的爬虫去运行. 这样就不用每次爬取前都与中心节点进行通信了, 这样就减轻了中心节点的压力, 也减小了各个子节点与主节点的通信时间. 对于数据量大的爬虫, 这样做的效果会比较明显.

C. 任务分割 + 中心主控法

推荐，特别适用于大型爬虫

原则:

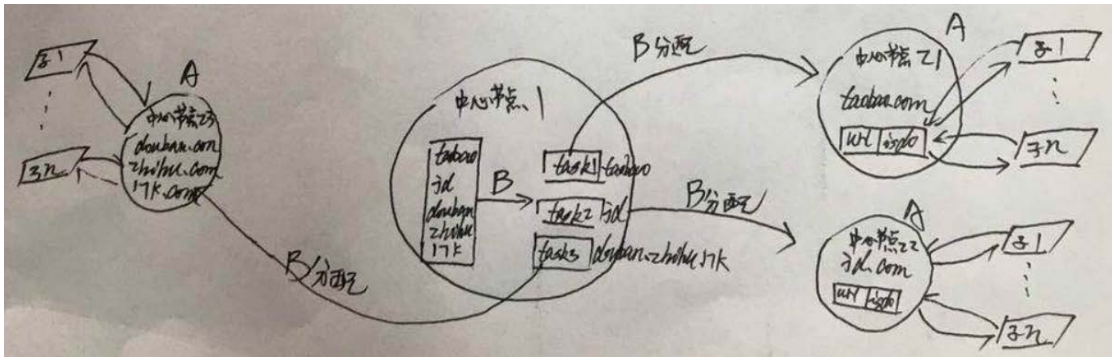
1. 对于网页数多的站点:
    1.1 按域名进行任务分割
    1.2 同域的网页同中心主控

第 1 步，在中心节点上以域名进行任务的分割. task1, task2, …等分别进行不同域名下的爬虫任务.

第 2 步，对于每个域名下的任务，再使用中心主控法，由 task1, task2, … 交给各个子节点进行调度.

2. 对于网页数少的站点:
    2.1 按网页数量进行任务分割. 如 ABC 三个站点所有网页数量加一起估计为 50W，就可以把 ABC 三个站点的爬虫放在一个任务中.
    2.2 多域网页同中心主控
    2.3 需要保证同域必同中心. 如保证 A 站点的所有爬虫都使用同一个控制中心进行调度，不把 A 站点的爬虫使用不同的控制中心进行调度. 即同域必须同中心，多域可以同中心.

如现在使用任务分割 + 中心主控法对 taobao.com, douban.com, zhihu.com, jd.com, 17k.com 各站架构如下:



第 1 步: 把 taobao, jd 作为大站, douban, zhihu, 17k 作为小站. 此时使用任务分割的方式，对于大站，同域必须同中心，分别分配一个 task 来作为大站的主控中心. 如 task1 作为 taobao 的主控中心, task2 作为 jd 的主控中心. 对于小站 douban, zhihu, 17k，这三者可以组合起来，按网页数量进行任务的分割. 使用 task3 来进行控制.

第 2 步: 使用中心主控法再次进行任务的分配. 把 task1, task2, task3 中的任务再次分配给不同的中心节点. 如 Z1 中心节点作为 taobao 任务的主控中心，如 Z2 中心节点作为 jd 任务的主控中心，如 Z3 中心节点作为 douban, zhihu, 17k 任务的主控中心.

第 3 步: Z1, Z2, Z3 主控中心分别给其下的所有子节点分配爬虫任务，在各个子节点上运行爬虫.

3. 数据存储方案(架构层)

方式	特点	
存储于各子节点中	不方便管理	
存储于主节点中	需要较好的带宽资源	√
在储于 HDFS 系统	大数据方案，性能高	√

4. 数据库方案

技术	特点	
mysql	性能一般	
oracle	成本较高	
mongodb	海量存储	√

5. 爬虫技术方案
1. urllib, requests 脚本形式: 灵活，研发成本高些
2. scrapy 框架形式: 研发成本低，但需要对接口熟悉

17.7 海量数据库导致的读写压力解决方案（补充）

一. 关于 sql 优化

常用技巧:
01. 水平划分，使用多服务器来存储数据，采用哈希定位. 2 人挑担. √

哈希定位方法. 既可以根据检索关键词和服务器数量确定数据保存到哪个服务器中，又可以根据检索关键词和服务器数量确定应该从哪个数据库中读取数据. 这就是水平划分情况下服务器的定位问题的解决方法.

水平划分方法可以大大减少服务器的压力.

```
import hashlib


# 哈希定位的函数. key 为检索关键字, num 为用于数据存储的服务器的数量
def whatserver(key, num):
```

```
    # 使用 md5 算法使用关键字段如 url 的 md5 值，然后 hexdigest 转换为 16 进制的字符，再使用 int 转换为 16 进制的整型数字. 对服务器数量求余，就能得到映射到服务器的编号. 调用这个函数，返回的是数据存储的服务器编号
    return int("0x" + str(hashlib.md5(str(key).encode("utf-8")).hexdigest()), 16) % num


# 根据用户名自动判断存储位置
alluser = ["libai", "冰露", "liming", "zhangjun", "白居易"]
for i in alluser:
    print(str(i) + "分配到了服务器:" + str(whatserver(i, 3)))

# 请思考，为什么不可以直接根据字符串长度直接求余算存储位置？
```

多次运行此程序，每次都会得到同样的值，所以这个程序是可以进行稳定映射的.
libai 分配到了服务器:0
冰露分配到了服务器:2
liming 分配到了服务器:0
zhangjun 分配到了服务器:2
白居易分配到了服务器:1

02. 垂直划分，按业务拆分 √
如图书，服装，电子等信息分别保存到不同的服务器中.
在实际业务中，建议使用水平划分和垂直划分相结合的方式.

03. 读写分离，读写分离后可以操作索引优化，且性能更佳 √
某些数据库或表只能读，某些数据库或表只能写.

04. 只读数据库加入索引，只写数据库去除索引 √
05. 进行主-从库划分，主库去索引，只写. 从库加索引，只读，适时同步.

06. 通过 SQL Bulk Copy 进行插入 √
不要一条条的插入数据了，而是使用集成的方法批量插入数据.

07. 查询不返回无关字段
尽量不使用 select * from table

08. 适时更新缓存
09. 采用 MemCache 缓存系统，加入该层，减少数据库压力 √
不直接操作数据库层，而是在数据库层上加上一层 MemCache 缓存层，适时进行缓存层与数据库层之间的同步. 这样既能减小数据库的压力，又能加快读写的速度.

10. 多表划分，每表限制 在 100W 内，可按日期划分，也可以使用哈希定位方法进行划分. √

11. 硬件选型方面，使用 RAID 磁盘阵列. √

二. 海量数据库选型
1. 关系型:
MySQL
Oracle
PostgreSQL
PolarDB(阿里)

2. 非关系型
Redis
MongoDB

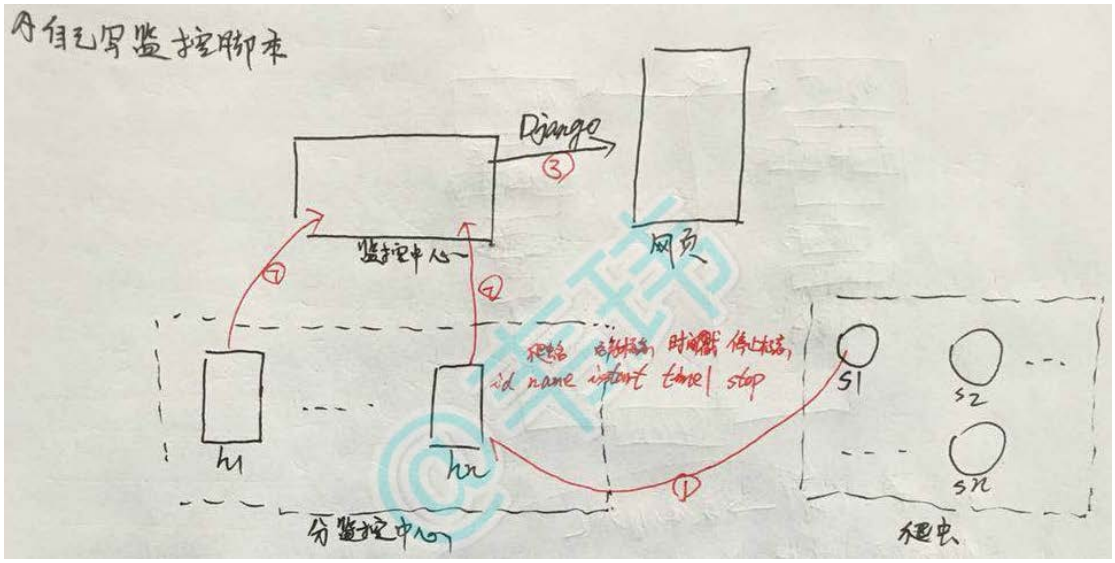
3. 大数据仓库

Hive ---- 基础数据仓库，实时运算性能较弱
Hbase ---- 实时访问性能较强

17.8 分布式爬虫性能监控方案（补充）

A. 自己写监控脚本

对于自己设计编写的爬虫系统，需要自己写监控脚本. 用来监控性能和可能出现的各种问题.



首先要有一个总的监控中心，如果是大型的爬虫项目，还需要有分监控中心来分担监控的压力.

首先创建一个数据库，创建一个包含以下几个字段的数据表．然后把一个正常的爬虫改造成一个有监控的爬虫．如果是在多个节点上运行的爬虫，再添加一个 hash 映射进行定位．

在各个分监控中心上设置好任务监控的表，以下为参考字段．

	爬虫名	启动标志位	时间戳(监控心跳)	停止标志位
id	name	start_flag	time1	stop_flag

1．爬虫运行中的监控:

哈希．有多个爬虫，多个分监控中心，每个爬虫向哪个分监控中心报告运行的状态．首先要确定每个爬虫向哪个监控中心进行报告的问题．使用哈希映射的方法来确定．注意这里不用把这种对应关系保存到数据库中，又会增加数据库的压力．
解决方案没有最好，只有更好，没有最好，只有最适合的．要根据公司本身的业务进行方案的选择．

更新 isstart: 1
爬虫启动时更新 isstart 字段为的值为 1，用来表示这个爬虫已经启动．
而在爬虫运行的过程中，每经过一定的时间或者爬取一定数量的网页就向分监控中心进行一次心跳的报告．可以使用求余运算，如每爬取 100 个网页就报告一次．也可以使用时间戳相减的方式，一旦两个时间的间隔超过了某个值，就向监控中心进行一次报告．

```
for i in range(0, 1000):
    if(i%100 == 0):
        # 更新 time1，表示在这个时间进行过心跳报告．
        ……
# 更新 stop:1   # 爬虫停止时更新 stop 这个标志位的值为 1.
```

2．分监控中心中实现监控的查询:

thistime，获取系统当前的时间．

查找运行异常的爬虫:
查找开始运行且没有停止的爬虫中超过一定时间还没有向监控中心进行报告的爬虫．因为设置了每个爬虫运行一定时间之后都向监控中心报告一次，如果一个爬虫超出了规定时间还没有报告，就表示它来没来得及报告就崩溃了．

```
select id,name from table where isstart = 1 and stop = 0 and thistime-time1 > 600;
```

查找正常运行的爬虫:

```
select id,name from table where isstart = 1 and stop = 0 and thistime-time1 <= 600;
```

查找已停止的爬虫

```
select id,name from table where stop = 1;
```

分监控中心中定时把数据汇总传递给总监控中心，这样就既能分解总监控中心的压力，又能实现总监控中心对所有爬虫的监控的目的了.

{ "id": xxx, "name": xxx, "state": xxx, "node": xxx } ----> 监控中心

3. 监控的可视化
通过 django 将 json 数据做成网页显示. 通过读取监控中心的数据库生成可视化的界面.

B. 成型方案

scrapy + scrapyd

如果使用 scrapy 爬虫，直接使用 scrapyd 来进监控

17.9 海量日志存储问题（补充）

A. 架构层: 分时存储 + 分服务器定向路由存储

分时存储，不同时间段的数据保存到不同地方，在数据量非常大时，某一批服务器只存储某一个时间段的日志数据，而另一批服务器则存储第二个时间段的日志数据，这样分时存储后，就方便于后期的分析了.

一批服务器中又有多个服务器，如何知道哪个服务器中保存哪个爬虫的日志信息呢，可以使用哈希定位的方式进行定向的存储. 保存同一时间每台日志服务器都在运行，不同爬虫的日志保存到不同的服务器中，这样就不会出现一台服务器压力过大的问题.

B. 技术应用层:

项目运行初期
LogDennee (推荐) + Scribe [简洁，推荐]

项目运行一段时间后
Chukwa (基于 Hdoop) 大数据方案 (建议选 Kafka)
Kafka 大数据方案
Flume 大数据方案

十八、Python 网络爬虫其他高级技术

学习精一通百

多个 api，得到的代理地址记录哪个 api 生成的，第一次时逐个调用多个 api，获取每个 api 的代理地址，保存到代理池中.
在爬虫运行的过程中，如果某个代理地址被某个网站禁止了，就记录下来，设置它的 jd_banned_flag = 1，同时不再使用此代理地址爬取被禁止的网站. 即 jd 网站只使用 jd_banned_flag = 0 的代理地址进行爬取，当爬取的所有网站都把这个代理地址禁止之后，就把它从代理池中删除，每次删除一个代理地址都检测代理池中此地址对应的 api 生成的所有代理地址的状态，当代理池中此 api 生成的所有代理地址被所有的网站都禁止后，即代理池中某 api 生成的代理地址的数量为 0 时，再次调用此 api 生成新的 ip，保存到代理池中.

对代理池中代理 ip 的数量进行检测，当代理地址的数量少于运行爬虫总的线程数量时，就再次调用对应代理地址数量最少的 api，再次新的 ip，同时把原来生成的 ip 地址全部从代理池中删除.

18.1 数据去重技术（布隆过滤器构建实战）

数据去重的方案有很多，比如基于 MySQL 标志位去重，基于 Redis 去重，基于过滤器去重等。
相对来说，布隆过滤器的性能较高。

布隆过滤器还没有一个非常成熟的模块，现有的模块实现起来都会有这样那样的问题. 把已经修改好的模块放在 Lib 目录下成为一个模块.

布隆过滤器由 2 个 py 文件来实现，一个是 BitVector.py，一个是 bloom.py. BitVector 是一种基础的数据结构, bloom 过滤器通过 BitVector 这种数据结构来构建就比较方便. 把这两个文件放在 python36/Lib 目录下成为一个模块.
复制到虚拟环境下的 Lib 目录中:
C:\Users\David\Envs\python3_spider\Lib

布隆过滤器的使用

```
import bloom

#  实例化 bloom 过滤器对象
#  第 1 个参数 error_rae 为容错率，第 2 个参数 elemnetNum 为估???存储的数据量
bl = bloom.BloomFilter(0.001, 1000000)
#  向布隆过滤器中插入数据
bl.insert_element("韦玮")
bl.insert_element("李明")
#  判断某个元素是否存在布隆过滤器中
bl.is_element_exist("张玉")
bl.is_element_exist("韦玮")
```

实战项目使用布隆过滤器

```
# 在实际的爬虫项目中， 每当爬取到一个网页的时候， 就把此网址插入到布隆过滤器中，在之后每次爬取时， 都判断一下此网址在布隆过滤器是否已经存在． 如果已经存在， 就说明已经爬过此网址， 就不需要再次爬取了． 如果网址在布隆过滤器中不存在， 就爬取此网站， 爬取成功后就把网址插入到布隆过滤器中．


import re
from bloom import *
import queue
import hashlib
import redis
import requests
import time
import socket
import difflib
import pymysql
import random

uapools = []
abc = []

socket.setdefaulttimeout(3)
mysql_inform = {
        "host": "127.0.0.1",
        "user": "root",
        "passwd": "root",
        "port": 3306,
        "db": "",
}
redisConnect = redis.Redis("192.168.1.3", "6379")

# 数据库初始配置
conn = pymysql.connect(host = mysql_inform["host"], user = mysql_inform["user"], passwd =
mysql_inform["passwd"], port = mysql_inform["port"], db = mysql_inform["db"])
print("数据库初始化连接完成")

sql_1 = "create table site_data(id int(32) auto_increment primary key, url varchar()"

try:
        conn.query(sql_1)
except Exception as err:
        pass

print("数据库创建完成")
print("数据库初始化配置完成")

# ------数据库初始化配置 end ------
```

```
bf = BloomFilter(0.001, 1000000)

class ListenWeb():
    def __init__(self, domain, nohost_domain, admin_email):
        self.domain = domain
        self.nohost_domain = nohost_domain
        self.q = queue.Queue()
        self.q.put(domain)
        # 创建一个布隆过滤器，对网址进行过滤，爬取过一个网址之后立刻写入到这个布隆过滤器中.
        self.bf2 = BloomFilter(0.001, 1000000)
        self.bf3 = BloomFilter(0.001, 1000000)
        self.novisit_url = []
        self.haschange_url = []
        hd = {"User-Agent" : "Mozilla/..."}
        fh = requests.get("http://www.itjuzi.com/", headers = hd, timeout = 3)
        self.ck = requests.utils.dict_from_cookiejar(fh.cookies)
        print(self.ck)

    def to_md5(self, data):
        return int("0x" + str(hashlib.md5(str(data).encode("utf-8")).hexdigest()), )

    def get_code(self, url):
        hd = {
            "User-Agent": "Mozilla/...",
            "Referer": self.domain,
        }
        fh = requests.get(url, headers = hd, timeout = 3, cookies = self.ck)
        return fh.status_code, fh

    # 网址过滤的方法. 判断网址有布隆过滤器中是否存在，如果存在，就返回为0. 即对某一个网址调用 filter 方法，如果返回为0，表示??个网址已经爬取过，就不再进行爬取.
    def filter(self, url):
        if(self.bf2.is_element_exist(url)):
            return 0

    def mysql_data(self, data):
        # url 内容有变化的网址, data: 新数据
        pass
```

18.2 pyspider 可视化技术

pyspider 也是一种爬虫框架，其可以可视化操作。
其实这个世界上，爬虫框架还有很多很多，你不可能也不需要全部掌握，那样是没有任何意义的。
由于 pyspider 可以可视化，相对于其他框架来说还算有一些特色，所以我们给大家简单介绍一下。

常用参数：fetch_type、method、data、headers、cookies、validate_cert。
fetch_type 为抓取的方式. pyspider 可以与 phantomjs 结合起来使用，把 fetch_type 设置为 js，就会自动调用 phantomjs 来爬取数据.

method 默认是 get 请求，可以通过把 method 设置为 post 来发送 post 请求. data 表示 post 时发送的数据.
headers, cookies,
validate_cert，在 https 页面时，需要有 ssl 的证书验证，需要设置 validate_cert = False，不对证书进行验证，否则就可能出现 ssl 证书的错误.

pyspider 的安装与部署

pip install pyspider

pyspider 项目的创建与爬虫的编写

直接在终端中输入 pyspider 来启动 pyspider 框架
pyspider

pyspider 是基于 web 界面进行管理的，爬虫的创建和运行都是在 web 界面中进行操作的.
打开浏览器，输入 127.0.0.1:5000

点 create 创建爬虫项目，输入项目名和起始网址.
Project Name: qiushibaike
Start URL: https://www.qiushibaike.com/

创建爬虫之后右边是爬虫的信息，左边上半部分是爬取的任务信息，下半部分是运行爬虫后在终端中显示的信息.

enable css selector helper, pyspider 中比较喜欢使用 css 选择器
web 中就是爬取到的内容
html 中显示爬取到的网页的源码
follows 表示接下来会爬取多少链接
messages 表示一些提示

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
# Created on 2018-06-12 13:52:07
# Project: qiushibaike
```

```
from pyspider.libs.base_handler import *


class Handler(BaseHandler):
    crawl_config = {
    }

    # on_start 表示入口方法
    @every(minutes=24 * 60)
    def on_start(self):
        # 使用 self.crawl 来发送一个请求. 并使用 self.index_page 作为该响应的解析函数
        self.crawl('https://www.qiushibaike.com/', callback=self.index_page)

    @config(age=10 * 24 * 60 * 60)
    def index_page(self, response):
        # doc 中是一个 css 选择器, 把当前响应中的 href 链接提取出来, 类似于通用爬虫的方式. 依次遍历提取到的各个链接, 再次使用 self.crawl 来发送提取到的链接的请求.
        for each in response.doc('a[href^="http"]').items():
            self.crawl(each.attr.href, callback=self.detail_page)

    # 提取详情页中的信息并返回
    @config(priority=2)
    def detail_page(self, response):
        return {
            "url": response.url,
            "title": response.doc('title').text(),
        }
```

对于 qiushibaike 这个爬虫来说，初始代码的意思是，首先访问 qiushibaike 的首页，从 qiushibaike 的首页中提取出所有的 href 链接，再向首页提取到的所有 href 链接发送请求，获取响应，再对响应进行解析，从中提取到需要的信息.

因为 qiushibaike 是 https 的站点，所以要在每个发送请求的地方加上 validate_cert = False

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
# Created on 2018-06-12 13:52:07
# Project: qiushibaike

from pyspider.libs.base_handler import *


class Handler(BaseHandler):
    crawl_config = {
    }

    # on_start 表示入口方法
    @every(minutes=24 * 60)
```

```
    def on_start(self):
        #  使用 self.crawl 来发送一个请求. 并使用 self.index_page 作为该响应的解析函数
        self.crawl('https://www.qiushibaike.com/', callback=self.index_page, validate_cert=False)

    @config(age=10 * 24 * 60 * 60)
    def index_page(self, response):
        # doc 中是一个 css 选择器, 把当前响应中的 href 链接提取出来, 类似于通用爬虫的方式. 依次遍历提取到的各个链接, 再次使用 self.crawl 来发送提取到的链接的请求.
        for each in response.doc('a[href^="http"]').items():
            self.crawl(each.attr.href, callback=self.detail_page, validate_cert=False)

    #  提取详情页中的信息并返回
    @config(priority=2)
    def detail_page(self, response):
        return {
            "url": response.url,
            "title": response.doc('title').text(),
        }
```

点击 run 运行爬虫，进入单步调试模式中，在 follows 中有一个要跟进的网址 https://www.qiushibaike.com/，点击右边的运行按钮，就会提取到首页中所有的 href 链接. 点击 html，就可以看到对应网址的源码，点击 web 就可以看到 web 预览，如果预览窗口很小，可以点 run，就能把预览窗口最大化了. 点击 css selector，就可以选择网页中的各个元素，同时显示出其 css 选择器的语法.

进入到 127.0.0.1:5000 首页中，就会看到刚才编写的爬虫，把爬虫的状态改为 running，点击 actions 中的 Run，就会开始运行爬虫. 修改 status 为 stop，就会停止爬虫的运行.

修改爬虫，进行多页的真实数据的爬取

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
# Created on 2018-06-12 13:52:07
# Project: qiushibaike

from pyspider.libs.base_handler import *
import re


class Handler(BaseHandler):
    crawl_config = {
    }

    # on_start 表示入口方法
    @every(minutes=24 * 60)
    def on_start(self):
        #  构建多页, 使用循环逐个发送请求
        for i in range(0, 13):
```

```
                # 使用 self.crawl 来发送一个请求. 并使用 self.index_page 作为该响应的解析函数
                self.crawl('https://www.qiushibaike.com/8hr/page/'+str(i+1)+'/',
callback=self.index_page, validate_cert=False)

    @config(age=10 * 24 * 60 * 60)
    def index_page(self, response):
        # 从 response.text 中提取出段子的内容
        pat = '<div class="content">(.*?)</span>'
        data = re.compile(pat, re.S).findall(response.text)
        data_write = ""
        # 依次遍历各个段子的内容, 把段子的内容处理之后写入到 txt 文件中
        for i in data:
            print(i)
            data_write += i.replace("<span>", "").replace("\n", "").replace(" ", "") + "\n"
            with open("C:/data/qiushi.txt", "a+", encoding = "utf-8") as fh:
                fh.write(data_write)
```

18.3 网络爬虫性能监控技术实战

第1步：创建监控数据库和数据表

name 为爬虫名, isstart 是否启动的标志位, time1 为报告心跳的时间戳, stop 为是否停止的标志位, ip1, ip2, 可以有多个子节点运行爬虫, 需要知道当前运行爬虫的节点的 ip 地址. 可以使用一台路由器连接多台主机作为子节点来运行爬虫, 此时就有一个公网 ip 地址, 还有多个内网 ip 地址, 所以需要知道公网 ip 和内网 ip 才能定位到某一台运行爬虫的子节点上.

mysql> create table spider(id int(32) auto_increment primary key,name varchar(32) unique,isstart varchar(32),time1 varchar(100),stop varchar(32),ip1 varchar(32),ip2 varchar(32));

开启 mysql 服务器, 创建数据库
create database tianshan;
use tianshan;
show tables;
# 创建数据表, 专门保存爬虫监控的数据
create table spider(id int(32) auto_increment primary key, name varchar(32) unique, isstart varchar(32), time1 varchar(100), stop varchar(32), ip1 varchar(32), ip2 varchar(32));

编写爬虫的性能监控模块

listenspider.py，把它放在 python 安装目录和虚拟环境根目录下的 Lib 目录下成为模块.

```
import urllib.request
import re

#  获取公网 ip
def getip2():
    try:
        r = urllib.request.urlopen('http://2017.ip138.com/ic.asp').read().decode("utf-8", "ignore")
        ip = re.compile("IP.*?\[(.*?)\]", re.S).findall(r)
        #  如果没有得到 ip 地址，就返回 ip 为"0.0.0.0"
        if (len(ip) == 0):
            ip = "0.0.0.0"
        else:
            ip = ip[0]
        return ip
    except Exception as err:
        r = urllib.request.urlopen('http://ip.chinaz.com/getip.aspx').read().decode("utf-8", "ignore")
        ip = re.compile("ip:'(.*?)'", re.S).findall(r)
        if (len(ip) == 0):
            ip = "0.0.0.0"
        else:
            ip = ip[0]
        return ip



def listen(conn, name, step):
    #  通过 soket 模块来获取内网 ip
    import socket
    hostname = socket.getfqdn(socket.gethostname())
    ip1 = socket.gethostbyname(hostname)
    ip2 = getip2()

    #  记录当前的时间戳
    import time
    thistime = int(time.time())

    #  对爬虫的三种状态进行判断，并进行不同的处理. step=0 是启动爬虫, step=1 是更新爬虫, step=2 是关闭爬虫
    if (int(step) == 0):
        '''启动爬虫时，把 isstart 设置为 1, stop 设置为 0，然后把数据插入到数据库中'''
        isstart = 1
        stop = 0
        try:
```

```
                sql = "insert into spider(name,isstart,time1,stop,ip1,ip2) values('" + str(name) + "','" +
str(
                    isstart) + "','" + str(thistime) + "','" + str(stop) + "','" + str(ip1) + "','" + str(ip2) +
"')"
                conn.query(sql)
                conn.commit()
            except Exception as err:
                pass
        elif (int(step) == 1):
            '''更新爬虫状态'''
            sql = "update spider set time1='" + str(thistime) + "' where name='" + str(name) + "'"
            try:
                conn.query(sql)
                conn.commit()
            except Exception as err:
                pass
        elif (int(step) == 2):
            '''停止爬虫'''
            sql = "update spider set stop='1' where name='" + str(name) + "'"
            try:
                conn.query(sql)
                conn.commit()
            except Exception as err:
                pass
```

也可以使用装饰器，把监视的代码修改为装饰器，直接作用在原来的爬虫中即可.

把第二章中的百度信息爬虫进行改造.

改造前代码

```
import urllib.request

url = "http://www.baidu.com/s?wd="
key = "马蹄糕"

# 对关键词进行编码，因为URL 中需要对中文等进行处理
key_code = urllib.request.quote(key)

# 带检索关键词的 url
url_key = url + key_code + "&ie=utf-8"

# 通过 for 循环爬取各页信息，这里爬取 1 到 10 页
for i in range(0, 10):
    print("正在爬取" + str(i + 1) + "页数据")
    # 根据刚刚总结的 url 规律构造当前 url
    thisurl = url_key + "&pn=" + str(i * 10)
    # 爬取这一页的数据
    data = urllib.request.urlopen(thisurl).read().decode("utf-8", "ignore")
    # 成功得到数据
```

```
    # 根据正则表达式将爬到的网页列表中各网页标题进行提取
    import re

    pat = '"title":"(.*?)"'
    rst = re.compile(pat, re.S).findall(data)
    # 将各标题信息通过循环遍历输出
    for j in range(0, len(rst)):
        print("第" + str(j) + "条网页标题是:" + str(rst[j]))
        print("----------")
```

改造后的爬虫

```
import urllib.request
# 第 1 步，导入监听所用到的模块
import pymysql
# 注意在导入时本文件同目录中不能有与模块同名的文件 listenspider，否则导入的就不是
python 安装目录中的模块，而是本目录下的同名文件.
from listenspider import *

# 第 2 步，设置爬虫的名称和连接信息，这里的连接信息是连接到主节点中的监控数据库中
name = "百度信息爬虫"
mconn = pymysql.connect(host="127.0.0.1", user="root", passwd="root", db="ts")

# 第 3 步，在爬虫启动时使用 listen 方法，传递数据库连接信息，爬虫的名字，以前爬虫的状态,
启动时的状态为"0"
listen(mconn, name, "0")

url = "http://www.baidu.com/s?wd="
key = "马蹄糕"
# 对关键词进行编码，因为 URL 中需要对中文等进行处理
key_code = urllib.request.quote(key)
# 带检索关键词的 url
url_key = url + key_code + "&ie=utf-8"

# 通过 for 循环爬取各页信息，这里爬取 1 到 10 页
for i in range(0, 10):

    # 第 4 步，在爬虫运行的过程中，每隔多长时间或每爬取多少个页面就进行一次汇报
    if (i % 2 == 0):
        listen(mconn, name, "1")

    print("正在爬取" + str(i + 1) + "页数据")
    # 根据刚刚总结的 url 规律构造当前 url
    thisurl = url_key + "&pn=" + str(i * 10)
    # 爬取这一页的数据
    data = urllib.request.urlopen(thisurl).read().decode("utf-8", "ignore")
    # 成功得到数据
    # 根据正则表达式将爬到的网页列表中各网页标题进行提取
    import re
```

```
        pat = '"title":"(.*?)"'
        rst = re.compile(pat, re.S).findall(data)
        # 将各标题信息通过循环遍历输出
        for j in range(0, len(rst)):
            print("第" + str(j) + "条网页标题是:" + str(rst[j]))
            print("-----------")

# 第 5 步，在爬虫完成任务正常退出之时，也要调用 listen 方法，把爬虫关闭的信息保存到数据库中
listen(mconn, name, "2")
```

运行爬虫，在 mysql 终端中执行 select * from spider; 就可以看到爬虫的运行状态. 并且已经得到了爬虫运行节点的公网 ip 和内网 ip. ip2 为公网 ip, ip1 为内网 ip.

当 isstart 为 1, stop 为 0，两个时间戳相差很大时，就意味着该节点出现了问题.
当使用多个节点运行分布式爬虫的时候，怎么定位到出了问题的机器呢，通过公网 ip 就能定位到连接到同一台路由的一批节点，再根据内网 ip 在连接此路由的所有节点中找到出问题的节点.

把第 2 章中的糗事百科爬虫改造成具有监控功能的爬虫.

改造前

```
import urllib.request
import re

name = "糗事百科爬虫"
headers = ("User-Agent",
           "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.2623.22 Safari/537.36 SE 2.X MetaSr 1.0")
opener = urllib.request.build_opener()
opener.addheaders = [headers]
urllib.request.install_opener(opener)

for i in range(0, 35):
    try:
        thisurl = "http://www.qiushibaike.com/8hr/page/" + str(i + 1) + "/"
        data = urllib.request.urlopen(thisurl).read().decode("utf-8", "ignore")
        pat = '<div class="content">.*?<span>(.*?)</span>.*?</div>'
        rst = re.compile(pat, re.S).findall(data)
        for j in range(0, len(rst)):
            print(rst[j])
            print("-------")
    except Exception as err:
        pass
```

改造后

```
import urllib.request
import re
# 导入模块
import pymysql
from listenspider import *

# 定义名称
name = "糗事百科爬虫"
# 建立数据库连接
mconn = pymysql.connect(host="127.0.0.1", user="root", passwd="root", db="ts")
# 在爬虫启动时向监控数据库发送爬虫开启报告
listen(mconn, name, "0")

headers = ("User-Agent",
           "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/49.0.2623.22 Safari/537.36 SE 2.X MetaSr 1.0")
opener = urllib.request.build_opener()
opener.addheaders = [headers]
urllib.request.install_opener(opener)

for i in range(0, 35):
    try:
        # 每爬取 6 次就调用一次 listen 方法向监控数据库发送爬虫运行报告
        if (i % 6):
            listen(mconn, name, "1")

        thisurl = "http://www.qiushibaike.com/8hr/page/" + str(i + 1) + "/"
        data = urllib.request.urlopen(thisurl).read().decode("utf-8", "ignore")
        pat = '<div class="content">.*?<span>(.*?)</span>.*?</div>'
        rst = re.compile(pat, re.S).findall(data)
        for j in range(0, len(rst)):
            print(rst[j])
            print("-------")
    except Exception as err:
        pass

# 在爬虫退出前也向监控数据库发送爬虫关闭报告
listen(mconn, name, "2")
```

爬虫监控技术的改进(未完成)

目标: 使用装饰器实现爬虫的监控
定时监控, 每隔一段时间向目标数据库写入数据
定量监控, 每爬取多少 url 或发送多少请求就向目标数据库写入数据

使用一个单独的线程来对各个爬虫进行监控, 每隔一段时间就向目标数据库中写入数据.

实现函数运行时间的装饰器
http://yangcongchufang.com/%E9%AB%98%E7%BA%A7python%E7%BC%96%E7%A8%8B%E5%9F%BA%E7%A1%80/python-someone-tell-me-not-simple.html

我想写一个类装饰器用来度量函数/方法运行时间

```
import time

class Timeit(object):
def __init__(self, func):
        self._wrapped = func
def __call__(self, *args, **kws):
        start_time = time.time()
        result = self._wrapped(*args, **kws)
        print("elapsed time is %s ")%(time.time() - start_time)
        return result

# 这个装饰器能够运行在普通函数上：
@Timeit
def func():
        time.sleep(2)
        return "invoking function func"


if __name__ == "__main__":
        func()
```

```
elapsed time is 2.00392723083
```

但是运行在方法上会报错，为什么？

```
class A(object):
@Timeit
def func(self):
        time.sleep(1)
        return "invoking method func"


if __name__ == "__main__":
        a = A()
        a.func()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-eb4fb185288b> in <module>()
```

```
      1 if __name__ == "__main__":
      2     a = A()
----> 3     a.func()
      4

<ipython-input-11-aedd3f23516b> in __call__(self, *args, **kws)
      4     def __call__(self, *args, **kws):
      5         start_time = time.time()
----> 6         result = self._wrapped(*args, **kws)
      7         print("elapsed time is %s ")%(time.time() - start_time)
      8         return result

TypeError: func() takes exactly 1 argument (0 given)
```

如果我坚持使用类装饰器，应该如何修改？

使用类装饰器后，在调用 func 函数的过程中其对应的 instance 并不会传递给 __call__ 方法，造成其 mehtod unbound ,那么解决方法是什么呢？描述符赛高:

```
class Timeit(object):
def __init__(self, func):
    self.func = func
```

18.4 网络爬虫维护与管理技术实战 Scrapyd

Scrapy 网络爬虫项目维护与管理可以使用一个专门的工具进行，Scrapyd.

安装 scrapyd
在中心节点上部署 scrapyd，服务端程序，在所有的子节点上安装 scrapyd-client，客户端程序，就可以使用子节点上的 scrapyd-client 向中心节点上的 scrapyd 服务端程序报告爬虫的运行状态.

pip install scrapyd
pip install scrapyd-client

把第 10 章的 dangdang 爬虫项目改造成具有监控和报告功能的爬虫，只需要配置项目根目录中的 cfg 文件即可.

修改前

```
# Automatically created by: scrapy startproject
#
# For more information about the [deploy] section see:
# https://scrapyd.readthedocs.io/en/latest/deploy.html
```

```
[settings]
default = dangdang.settings

[deploy]
#url = http://localhost:6800/
project = dangdang
```

修改后

```
# Automatically created by: scrapy startproject
#
# For more information about the [deploy] section see:
# https://scrapyd.readthedocs.io/en/latest/deploy.html

[settings]
default = dangdang.settings

# deploy 中就是项目发布名时的名称
[deploy:dangdang1]
#  设置中心节点的服务器地址和端口号，这里是本机即做为中心节点又作为子节点. 注意中心节点必须是
url = http://localhost:6800/
# project 是项目名
project = dangdang
```

在中心节点的终端中直接输入 scrapyd 开启 scrapyd 服务端

在子节点中访问 127.0.0.1:6800，在这里可以在本机中访问.

在客户端将 dangdang 爬虫进行发布，使其可以通过 web 进行监控.

在安装好 scrapy-client 之后，就会生成 scrapyd-deploy 文件，但是此文件无法直接执行，所以必须要新建一个脚本文件来执行此文件.
新建 scrapyd-deploy.bat 脚本文件，需要把其中 scrapyd-deploy 文件修改为本机的 scrapyd-deploy 文件.

```
python D:/Python35/Scripts/scrapyd-deploy %*
```

本机的 scrapyd-deploy 文件位于
"C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapyd-client\scrapyd-deploy"

修改后的文件内容为

```
python C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapyd-client\scrapyd-deploy %*
```

把修改好的 scrapyd-deploy.bat 文件放在 python 安装目录下的 Scripts 目录下.

C:\Users\David\Envs\python3_spider\Scripts

在运行爬虫项目时就不是使用 scrapy crawl 了，而是使用 curl.exe 工具. 把 curl.exe 也放在 C:\Users\David\Envs\python3_spider\Scripts 目录下，因为 scripts 已经添加到系统环境变量了，所以可以直接通过 curl 来执行命令

在客户端中先使用 scrapyd-deploy 把修改后的 scrapy 项目推送到中心节点的 scrapyd 服务端中，再使用 curl.exe 来运行爬虫.

在如下网址中下载与系统对应的 curl.exe 最新版本，解压到某一目录中，并把 bin 文件夹添加到系统环境变量中.
https://curl.haxx.se/download.html

如下载 curl-7.60.0-win64-mingw.zip 版本的，解压后放在如下目录中
D:\python_packages\curl-7.60.0-win64-mingw\bin

先在中心服务器上运行 scrapyd，再在子节点中发送项目.
进入到项目的根目录中，如进入到 dangdang 项目文件夹中，即 scrapy.cfg 文件所在的目录中，在客户端执行以下命令来发布爬虫. 第 1 个参数是项目发布的名称，即 dangdang1. 第 2 个参数是项目名称 -p dangdang

scrapyd-deploy dangdang1 -p dangdang

scrapyd-deploy dangdang1 -p dangdang

python C:\Users\David\Envs\python3_spider\Lib\site-packages\scrapyd-client\scrapyd-deploy dangdang1 -p dangdang
Packing version 1528804055
Deploying to project "dangdang" in http://localhost:6800/addversion.json
Server response (200):
{"node_name": "Win10-S2600", "status": "ok", "project": "dangdang", "version": "1528804055", "spiders": 1}

只要出现了 status ok 的字样，就表明发布成功了.

访问 http://127.0.0.1:6800/，就可以在 Available projects 中看到可用的项目.

使用以下命令来运行爬虫, project 是 scrapy 的项目名称, spider 是爬虫名称.

curl http://localhost:6800/schedule.json -d project=dangdang -d spider=dang

再次访问 http://127.0.0.1:6800/
就会在 Jobs 中看到爬虫的运行状态.

爬虫运行过程中的状态

Project	Spider	Job	PID	Start	Runtime	Finish	Log
Pending							
Running							
dangdang	dang	3f2712d06e3911e89d2e485b39cada77	25220	2018-06-12 20:08:01	0:00:02		Log
Finished							

爬虫运行结束时的状态

Project	Spider	Job	PID	Start	Runtime	Finish	Log
Pending							
Running							
Finished							
dangdang	dang	3f2712d06e3911e89d2e485b39cada77		2018-06-12 20:08:01	0:00:20	2018-06-12 20:08:21	Log

可以点击 Log 来查看日志文件.

部署 11 章的 linux 爬虫

第 1 步: 修改 scrapy.cfg

```
# Automatically created by: scrapy startproject
#
# For more information about the [deploy] section see:
# https://scrapyd.readthedocs.io/en/latest/deploy.html

[settings]
default = hxun.settings

[deploy:hexun1]
url = http://localhost:6800/
project = hxun
```

第 2 步: 部署到中心节点中

在子节点中执行. 进入到 scrapy 项目根目录中
cd hxun
scrapyd-deploy hexun1 -p hxun

第 3 步: 在子节点中运行爬虫
curl http://localhost:6800/schedule.json -d project=hxun-d spider=hx

对于分布式的爬虫项目, 在中心节点中安装好 scrapyd, cfg 文件中的 url 和端口就是中心节点的 ip 地址和端口了. 但一定要注意中心节点中的 ip 地址和端口一定要处于开放的状态, 在修改 cfg 文件时注意 url 要写为公网的 ip 地址. 然后在每一台子节点中对项目进行部署, 再在每一台子节点中运行爬虫即可. 如果本机部署成功而分布式不能部署成功, 很可能是因为通信的问题, 必须要开放中心节点相应的端口, 允许远程连接.