# 3. 爬虫框架开发

## 介绍

在前面我们学习的爬虫的基本知识和scrapy框架包括mongodb数据库，在本部分，我们会结合前面的知识点，完成一个自定义的爬虫框架，满足爬虫的主流需求

## 目标

* 了解框架,明确实现框架的目的
* 能够说出框架设计的思路和流程
* 掌握框架的实现过程
* 能够使用该框架抓取数据

# 3.1. 爬虫框架开发分析

## 介绍

本节会首先了解框架概念和实现框架的好处，在这个基础上思考如何实现自己的爬虫框架；通过对scrapy框架的流程的分析，明确自己的框架要实现的模块和逻辑；最后完成自己框架的雏形机构

## 内容

* 了解框架的内涵
* 学习scrapy框架的思路
* 实现自己框架的雏形结构

# 3.1.1. 了解框架

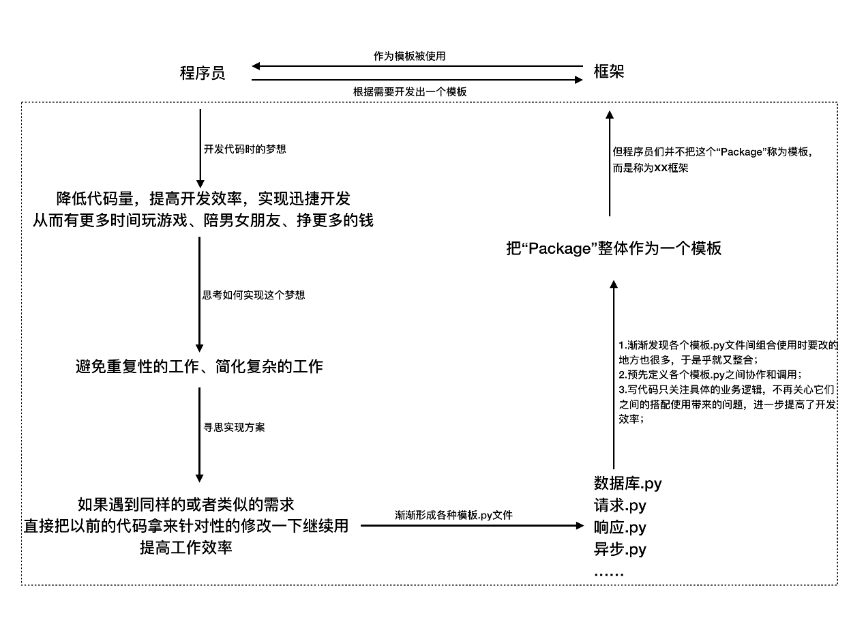
# 了解框架

## 目标

* 明确什么是框架
* 了解实现框架的好处
* 了解框架思路的来源

## 1.什么是框架

框架是为了为解决一类问题而开发的程序，框架两个字可以分开理解，框：表示指定解决问题的边界，明确要解决的问题；架：表达的是能够提供一定的支撑性和可扩展性；从而实现解决这类问题达到快速开发的目的。

程序员从零开发框架的心路历程 

## 2.为什么要实现一个框架

现在网络上现成的开源第三方框架非常多，为什么还需要自己实现一个框架？

1. 现成开源第三方框架的局限性：

现成开源第三方框架是为了尽可能满足大部分的需求，不可能做到面面俱到，以及第三方框架的调试相对复杂

1. 解决特定的工作需求：

工作中会有很多特殊的需求，会经常使用某种套路去实现这些需求，那么为了提高效率可以专门把这种套路封装成一个框架

比如专门针对电商网站、新闻资讯写一个爬虫框架；再比如针对断点续爬、增量抓取等需求写一个框架

1. 提高自己的技术能力：

不一定需要亲自造轮子，但是应该知道如何造轮子

## 3.如何完成一个框架

现在我们明确了框架是什么以及为什么要实现一个框架，那么到底应该如何实现一个框架呢，我们的idea从哪里来呢？

1. 经验丰富的程序员：

直接根据以往经验和业务的需求进行框架原型设计，并用语言去实现

1. 经验一般的程序员：

通常应该是先学习别人的框架如何实现的，先学习别人优秀的比较好的实现思路和方案

那么同样的，对于我们：  
我们可以结合目前学习过的爬虫知识和爬虫框架，了解他们的设计思路，在这个基础上进行模仿和改进，从而实现一个框架

## 小结

* 本小结重点：
  + 框架是什么，能够解决什么问题
  + 为什么要实现一个框架

# 3.1.2. 框架设计思路分析

## 目标

* 回顾并掌握scrapy的模块的作用
* 回顾并掌握scrapy的数据的传递过程
* 能够说出自己要实现的框包含的模块和模块的作用

## 1.学习Scrapy，提取它的设计思想(idea)

### 1.1 爬虫的流程(掌握)

爬虫框架解决的问题是爬虫问题，先来看看爬虫的基本流程：

1. 构建请求信息(url、method、headers、params、data)
2. 发起HTTP/HTTPS请求，获取HTTP/HTTPS响应
3. 解析响应，分析响应数据的数据结构或者页面结构
   * 提取数据
   * 提取请求的地址
4. 对数据进行存储/对新的请求地址重复前面的步骤

无论什么爬虫框架，其核心都离不开上面几个步骤

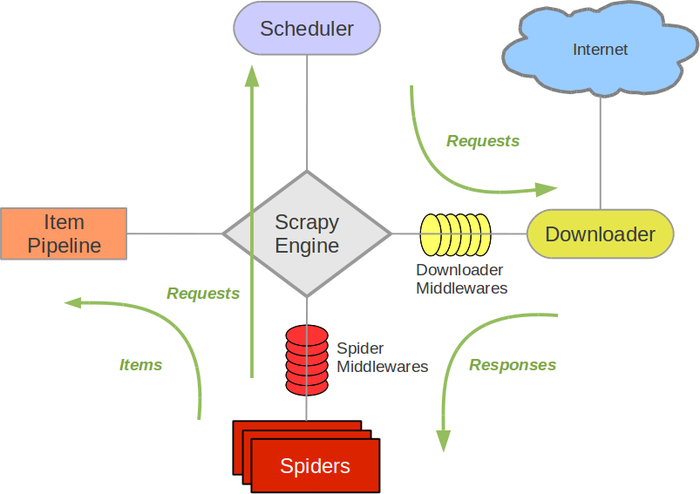
### 1.2 分析scrapy爬虫流程(掌握)

分析目标：

a. 分析各个组件的功能和作用

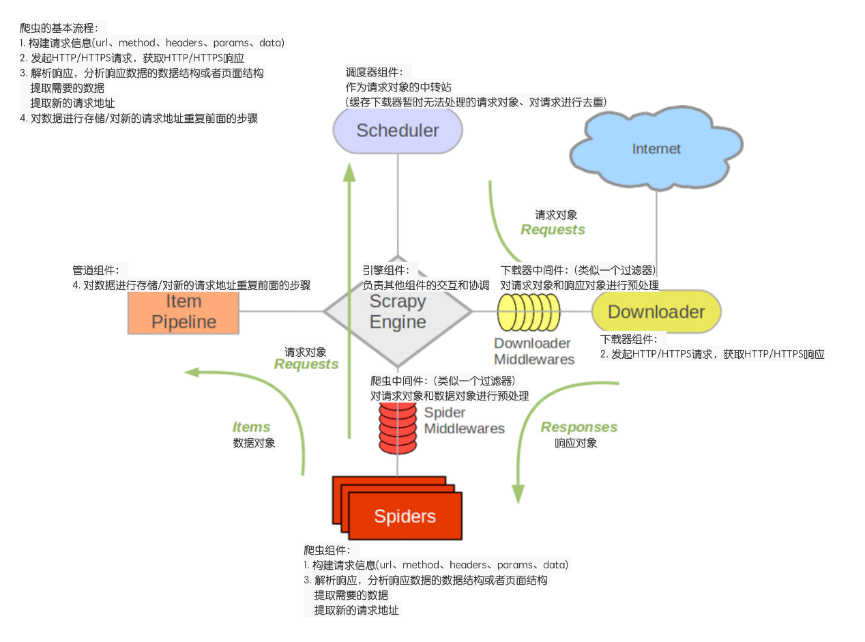
b. 分析各个组件之间的协作关系

c. 对组件进行大致的分类(核心和次要)



## 2.scrapy爬虫流程分析结果(掌握)

Scrapy流程分析图



1. 三个内置对象：

请求对象(Request)

响应对象(Response)

数据对象(Item)

1. 五个核心组件：

**爬虫组件**

构建请求信息(初始的)，也就是生成请求对象(Request)

解析响应对象，返回数据对象(Item)或者新的请求对象(Request)

**调度器组件**

缓存请求对象(Request)，并为下载器提供请求对象，实现请求的调度

对请求对象进行去重判断

**下载器组件**

根据请求对象(Request)，发起HTTP、HTTPS网络请求，拿到HTTP、HTTPS响应，构建响应对象(Response)并返回

**管道组件**

负责处理数据对象(Item)

**引擎组件**

负责驱动各大组件，通过调用各自对外提供的API接口，实现它们之间的交互和协作

提供整个框架的启动入口

1. 两个中间件：

**爬虫中间件**

对请求对象和数据对象进行预处理

**下载器中间件**

对请求对象和响应对象进行预处理

那么对应的，我们也可以在自己的框架是实现这样几个模块和对象

## 小结

本小结重点：

* + 掌握scrapy的模块的作用
  + 掌握scrapy的数据的传递过程

# 3.1.3. 雏形代码结构

代码实现分析

## 目标

* 掌握模块之间的关系
* 实现自己框架的代码结构

### 1.明确模块之间的逻辑关系

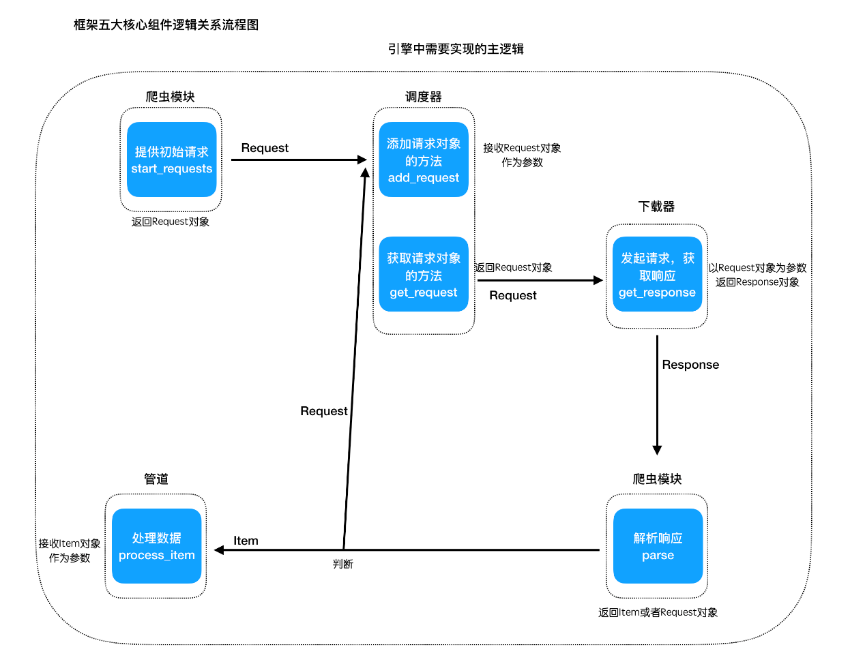
五个核心模块和三个内置的对象是关键模块，需要优先实现

先抛开中间件，分析下它们之间的逻辑关系是：

1. 构造spider中start\_urls中的请求
2. 传递给调取器进行保存，之后从中取出
3. 取出的request对象交给下载的进行下载，返回response
4. response交给爬虫模块进行解析，提取结果
5. 如果结果是request对象，重新交给调度器，如果结果是item对象，交给管道处理

以上的逻辑是在引擎中完成的

对应的他们在引擎中的逻辑如下图：



### 2.设计代码结构

1. 首先给框架起一个名称，如：

scrapy\_plus

1. 继续分类以及解耦的设计思想：

把核心模块放置在一起

请求对象模块和响应对象模块统一作为http模块

数据对象单独作为一个分类

代码结构如下：

-- scrapy\_plus

-- \_\_init\_\_.py

-- core

-- \_\_init\_\_.py

-- spider.py

-- scheduler.py

-- downloader.py

-- pipeline.py

-- engine.py

-- http

-- \_\_init\_\_.py

-- request.py

-- response.py

-- item.py

到这里，我们完成了框架的大致结构是设计，那么接下来我们就需要实现模块中的具体内容了

## 小结

- 本小结重点

* 掌握模块之间的逻辑关系

# 3.2. 框架雏形实现

## 介绍

前面我们完成了框架的大致结构，那么接下来的这部分，我们还需要实现具体每个模块中的基本方法，让框架能够帮助我们完成一些基础的功能。

## 内容

* 完成HTTP相关的模块
* 完成item模块
* 完成核心模块(引擎，调度器，下载器等)
* 安装框架
* 运行整个框架
* 完成中间件模块

# 3.2.1. 框架雏形--http模块和item模块

## 目标

* 完成request模块的基础封装
* 完成respons模块的基础封装
* 完成item模块的基础封装

### 1.request对象的封装

对HTTP基本的请求属性进行简单封装，实现一个Request对象

# scrapy/http/request.py

'''封装Request对象'''

class Request(object):

'''框架内置请求对象，设置请求信息'''

def \_\_init\_\_(self, url, method='GET',\

headers=None, params=None, data=None):

self.url = url # 请求地址

self.method = method # 请求方法

self.headers = headers # 请求头

self.params = params # 请求参数

self.data = data # 请求体

#### 2.response对象的封装

对HTTP基本的响应属性进行简单封装，实现一个Response对象

# scrapy/http/response.py

'''封装Response对象'''

class Response(object):

'''框架内置Response对象'''

def \_\_init\_\_(self, url, status\_code, headers, body):

self.url = url # 响应url

self.status\_code = status\_code # 响应状态码

self.headers = headers # 响应头

self.body = body # 响应体

#### 3.item对象的封装

对数据进行简单封装，实现Item对象：

# scrapy/item.py

'''item对象'''

class Item(object):

'''框架内置Item对象'''

def \_\_init\_\_(self, data):

# data表示传入的数据

self.\_data = data # 设置为简单的私有属性

@property

def data(self):

'''对外提供data进行访问，一定程度达到保护的作用'''

return self.\_data

其中property的理解：

* property 能够让调用一个方法和调用一个属性一样容易，即不用打括号
* property 能够让这个属性的值是只读的，即不能够对其进行重新复制，达到一定的保护的目的

## 小结

* 本节重点
  + 完成对request，response,item的编写
  + 掌握property的用法

# 3.2.2. 框架雏形--核心模块

## 目标

* 完成spider模块的封装
* 完成调度器模块的封装
* 完成下载器模块的封装
* 完成管道模块的封装
* 完成引擎模块的封装

## 1.spider模块的封装

### 1.1 爬虫组件功能：

* 构建请求信息(初始的)，也就是生成请求对象(Request)
* 解析响应对象，返回数据对象(Item)或者新的请求对象(Request)

### 1.2 实现方案：

* 实现start\_requests方法，返回请求对象
* 实现parse方法，返回Item对象或者新的请求对象

# scrapy\_plus/core/spider.py

'''爬虫组件封装'''

from scrapy\_plus.item import Item # 导入Item对象

from scrapy\_plus.http.request import Request # 导入Request对象

class Spider(object):

'''

1. 构建请求信息(初始的)，也就是生成请求对象(Request)

2. 解析响应对象，返回数据对象(Item)或者新的请求对象(Request)

'''

start\_url = 'http://www.baidu.com' # 默认初始请求地址

#这里以请求百度首页为例

def start\_requests(self):

'''构建初始请求对象并返回'''

return Request(self.start\_url)

def parse(self, response):

'''解析请求

并返回新的请求对象、或者数据对象

'''

return Item(response.body) # 返回item对象

## 2. 调度器模块的封装

### 2.1 调度器功能：

* 缓存请求对象(Request)，并为下载器提供请求对象，实现请求的调度：
* 对请求对象进行去重判断：实现去重方法\_filter\_request，该方法对内提供，因此设置为私有方法

### 2.2 实现方案：

* 利用队列FIFO存储请求；
* 实现add\_request方法添加请求，接收请求对象作为参数；
* 实现get\_request方法对外提供从队列取出的请求对象

# scrapy\_plus/core/scheduler.py

'''调度器模块封住'''

# 利用six模块实现py2和py3兼容

from six.moves.queue import Queue

class Scheduler(object):

'''

1. 缓存请求对象(Request)，并为下载器提供请求对象，实现请求的调度

2. 对请求对象进行去重判断

'''

def \_\_init\_\_(self):

self.queue = Queue()

def add\_request(self, request):

'''添加请求对象'''

self.queue.put(request)

def get\_request(self):

'''获取一个请求对象并返回'''

request = self.queue.get()

return request

def \_filter\_request(self):

'''请求去重'''

# 暂时不实现

pass

## 3. 下载器模块的封装

### 3.1 下载器功能：

* 根据请求对象(Request)，发起HTTP、HTTPS网络请求，拿到HTTP、HTTPS响应，构建响应对象(Response)并返回

### 3.1 实现方案：

* 利用requests、urllib2等模块发请求，这里使用requests模块
* 实现get\_response方法，接收request请求对象作为参数，发起请求，获取响应

# scrapy\_plus/core/downloader.py

'''下载器组件'''

import requests

from scrapy\_plus.http.response import Response

class Downloader(object):

'''根据请求对象(Request)，发起HTTP、HTTPS网络请求，拿到HTTP、HTTPS响应，构建响应对象(Response)并返回'''

def get\_response(self, request):

'''发起请求获取响应的方法'''

# 1. 根据请求对象，发起请求，获取响应

# 判断请求方法：

if request.method.upper() == 'GET':

resp = requests.get(request.url, headers=request.headers,\

params=request.params)

elif request.method.upper() == 'POST':

resp = requests.post(request.url,headers=request.headers,\

params=request.params,data=request.data)

else:

# 如果方法不是get或者post，抛出一个异常

raise Exception("不支持的请求方法")

# 2. 构建响应对象,并返回

return Response(resp.url, resp.status\_code, resp.headers, resp.content)

## 4.管道模块的封装

### 4.1 管道组件功能：

* 负责处理数据对象

### 4.2 实现方案：

* 实现process\_item方法，接收数据对象作为参数

# scrapy\_plus/core/pipeline.py

'''管道组件封装'''

class Pipeline(object):

'''负责处理数据对象(Item)'''

def process\_item(self, item):

'''处理item对象'''

print("item: ", item)

## 5.引擎模块的封装

### 5.1 引擎组件功能：

* 对外提供整个的程序的入口
* 依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)

### 5.2 实现方案：

* 利用init方法初始化其他组件对象，在内部使用
* 实现start方法，由外部调用，启动引擎
* 实现\_start\_engine方法，完成整个框架的运行逻辑
* 具体参考上一小节中雏形结构引擎的逻辑

# scrapy\_plus/core/engine.py

'''引擎组件'''

from scrapy\_plus.http.request import Request # 导入Request对象

from .scheduler import Scheduler

from .downloader import Downloader

from .pipeline import Pipeline

class Engine(object):

'''

a. 对外提供整个的程序的入口

b. 依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)

'''

def \_\_init\_\_(self, spider):

self.spider = spider # 接收外部传入的爬虫对象

self.scheduler = Scheduler() # 初始化调度器对象

self.downloader = Downloader() # 初始化下载器对象

self.pipeline = Pipeline() # 初始化管道对象

def start(self):

'''启动整个引擎'''

self.\_start\_engine()

def \_start\_engine(self):

'''依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)'''

# 1. 爬虫模块发出初始请求

start\_request = self.spider.start\_requests()

# 2. 把初始请求添加给调度器

self.scheduler.add\_request(start\_request)

# 3. 从调度器获取请求对象，交给下载器发起请求，获取一个响应对象

request = self.scheduler.get\_request()

# 4. 利用下载器发起请求

response = self.downloader.get\_response(request)

# 5. 利用爬虫的解析响应的方法，处理响应，得到结果

result = self.spider.parse(response)

# 6. 判断结果对象

# 6.1 如果是请求对象，那么就再交给调度器

if isinstance(result, Request):

self.scheduler.add\_request(result)

# 6.2 否则，就交给管道处理

else:

self.pipeline.process\_item(result)

## 小结

* 本节重点：
  + 明确每个模块的功能以及需要实现的方法
  + 掌握引擎中实现逻辑的刘而成
  + 完成对爬虫框架的编写

# 3.2.3. 框架安装

## 目标

* 了解setup.py 的编写
* 掌握制作模块的方法

## 1. 安装框架的目的

利用setup.py将框架安装到python环境中，在编写爬虫时候，作为第三方模块来调用，

## 2.框架安装第一步：完成setup.py的编写

* 以下代码相当于一个模板，只用更改name字段出，改为对应的需要安装的模块名称就可以，比如这里是：scrapy\_plus
* 将setup.py文件放到scrapy\_plus的同级目录下

from os.path import dirname, join

# from pip.req import parse\_requirements

from setuptools import (

find\_packages,

setup,

)

def parse\_requirements(filename):

""" load requirements from a pip requirements file """

lineiter = (line.strip() for line in open(filename))

return [line for line in lineiter if line and not line.startswith("#")]

with open(join(dirname(\_\_file\_\_), './VERSION.txt'), 'rb') as f:

version = f.read().decode('ascii').strip()

setup(

name='scrapy-plus', # 模块名称

version=version,

description='A mini spider framework, like Scrapy', # 描述

packages=find\_packages(exclude=[]),

author='itcast',

author\_email='your@email.com',

license='Apache License v2',

package\_data={'': ['\*.\*']},

url='#',

install\_requires=parse\_requirements("requirements.txt"), # 所需的运行环境

zip\_safe=False,

classifiers=[

'Programming Language :: Python',

'Operating System :: Microsoft :: Windows',

'Operating System :: Unix',

'Programming Language :: Python :: 2.7',

'Programming Language :: Python :: 3.4',

'Programming Language :: Python :: 3.5',

'Programming Language :: Python :: 3.6',

],

)

注意： 上面代码中可能会报错需要额外安装packaging模块，更新setuptools

* pip install packaging
* pip install --upgrade setuptools

pip.req可能不存在，对应的可以：

def parse\_requirements(filename):

""" load requirements from a pip requirements file """

lineiter = (line.strip() for line in open(filename))

return [line for line in lineiter if line and not line.startswith("#")]

## 3.框架安装第二步：完成requirements.txt的编写

功能：

* 写明依赖环境所支持的模块及其版本

使用：

* 在setup.py中使用
* 放置在setup.py同级目录下

requests>=2.18.4

six>=1.11.0

## 4.框架安装第三步：完成VERSION.txt的编写

功能：

* 标明当前版本，一个合格的模块，应当具备相应的版本号

使用：

* 在setup.py中使用
* 放置在setup.py同级目录下

1.0

## 4.框架安装第四步：执行安装命令

步骤：

* 切换到setup.py所在目录
* 切换到对应需要python虚拟环境下
* 在终端执行python setup.py install

## 小结

本小结重点

* + 掌握安装模块的方法

# 3.2.4. 框架运行--main.py

## 目标

* 熟悉框架的执行过程
* 掌握框架的启动方法

## 1. 编写main.py

新在其他路径下创建一个项目文件夹 project\_dir

# project\_dir/main.py

from scrapy\_plus.core.engine import Engine # 导入引擎

if \_\_name\_\_ == '\_\_main\_\_':

engine = Engine() # 创建引擎对象

engine.start() # 启动引擎

运行结果：管道中打印的item对象

item对象:<scrapy\_plus.item.Item object at 0x10759eef0>

## 小结

* 本节重点
  + 掌握框架的启动方法

# 3.2.5. 框架雏形--实现中间件模块

## 目标

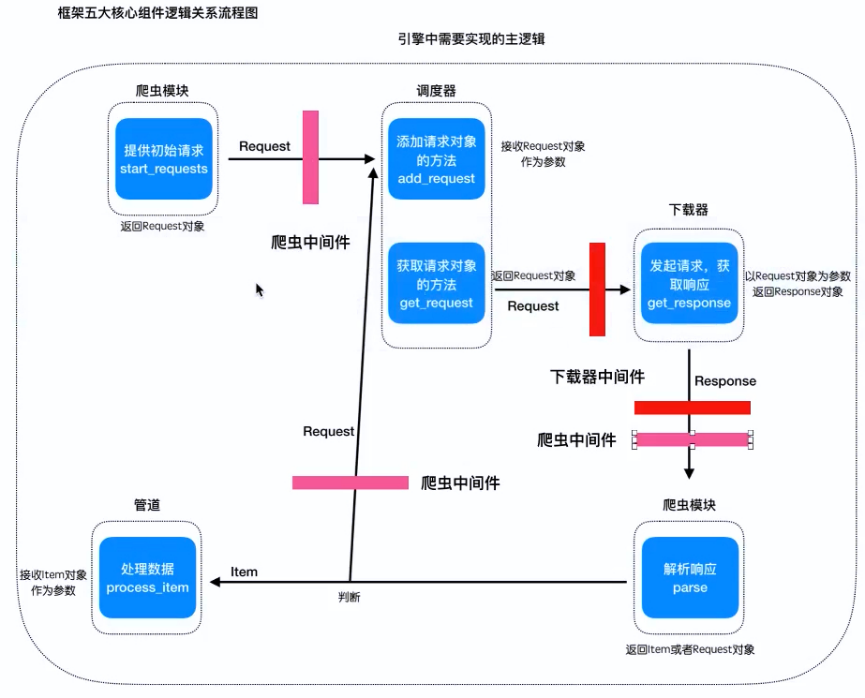
* 理解中间件的用途
* 熟悉中间件在框架逻辑中的位置
* 能够编写中间件模块

## 1. 为什么需要中间件

中间件相当于一个钩子，能够在其中对request对象和response响应根据特定的需求进行一些特定的处理 例如：对于所有的request对象，我们需要在其中对他添加代理或者是随机的User-Agent都可以在中间件中完成

## 2. 中间件实现的逻辑

框架中的中间件逻辑关系如下图中红色和黄色方块的位置：



内置中间件的代码结构：

- scrapy\_plus

-- \_\_init\_\_.py

-- core

-- \_\_init\_\_.py

-- spider.py

-- scheduler.py

-- downloader.py

-- pipeline.py

-- engine.py

-- http

-- \_\_init\_\_.py

-- request.py

-- response.py

-- middlewares

-- \_\_init\_\_.py

-- spider\_middlewares.py

-- downloader\_middlewares.py

-- item.py

## 2.完成爬虫中间件spider\_middlewares

# scrapy\_plus/middlewares/spider\_middlewares.py

class SpiderMiddleware(object):

'''爬虫中间件基类'''

def process\_request(self, request):

'''预处理请求对象'''

print("这是爬虫中间件：process\_request方法")

return request

def process\_response(self, response):

'''预处理数据对象'''

print("这是爬虫中间件：process\_response方法")

return response

## 3.完成下载downloader\_middlewares

# scrapy\_plus/middlewares/downloader\_middlewares.py

class DownloaderMiddleware(object):

'''下载器中间件基类'''

def process\_request(self, request):

'''预处理请求对象'''

print("这是下载器中间件：process\_request方法")

return request

def process\_response(self, response):

'''预处理响应对象'''

print("这是下载器中间件：process\_response方法")

return response

## 3.修改engine.py

加入中间件模块

# scrapy\_plus/core/engine.py

'''引擎

a. 对外提供整个的程序的入口

b. 依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)

'''

from scrapy\_plus.http.request import Request # 导入Request对象

from scrapy\_plus.middlewares.spider\_middlewares import SpiderMiddleware

from scrapy\_plus.middlewares.downloader\_middlewares import DownloaderMiddleware

from .spider import Spider

from .scheduler import Scheduler

from .downloader import Downloader

from .pipeline import Pipeline

class Engine(object):

def \_\_init\_\_(self):

......

self.spider\_mid = SpiderMiddleware() # 初始化爬虫中间件对象

self.downloader\_mid = DownloaderMiddleware() # 初始化下载器中间件对象

......

def \_start\_engine(self):

'''依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)'''

# 1. 爬虫模块发出初始请求

start\_request = self.spider.start\_requests()

# 2. 把初始请求添加给调度器

# 利用爬虫中间件预处理请求对象

start\_request = self.spider\_mid.process\_request(start\_request)

self.scheduler.add\_request(start\_request)

# 3. 从调度器获取请求对象，交给下载器发起请求，获取一个响应对象

request = self.scheduler.get\_request()

# 利用下载器中间件预处理请求对象

request = self.downloader\_mid.process\_request(request)

# 4. 利用下载器发起请求

response = self.downloader.get\_response(request)

# 利用下载器中间件预处理响应对象

response = self.downloader\_mid.process\_response(response)

# 5. 利用爬虫的解析响应的方法，处理响应，得到结果

result = self.spider.parse(response)

# 6. 判断结果对象

# 6.1 如果是请求对象，那么就再交给调度器

if isinstance(result, Request):

# 利用爬虫中间件预处理请求对象

result = self.spider\_mid.process\_request(result)

self.scheduler.add\_request(result)

# 6.2 否则，就交给管道处理

else:

# 利用爬虫中间件预处理数据对象

result = self.spider\_mid.process\_item(result)

self.pipeline.process\_item(result)

## 4.观察结果

运行main.py文件，查看结果

运行结果：

这是爬虫中间件：process\_request方法

这是下载器中间件：process\_request方法

这是下载器中间件：process\_request方法

这是爬虫中间件：process\_response方法

item对象: <scrapy\_plus.item.Item object at 0x10759eef0>

## 小结

* 本小节重点
  + 完成中间件的编写
  + 完成对引擎逻辑的修改

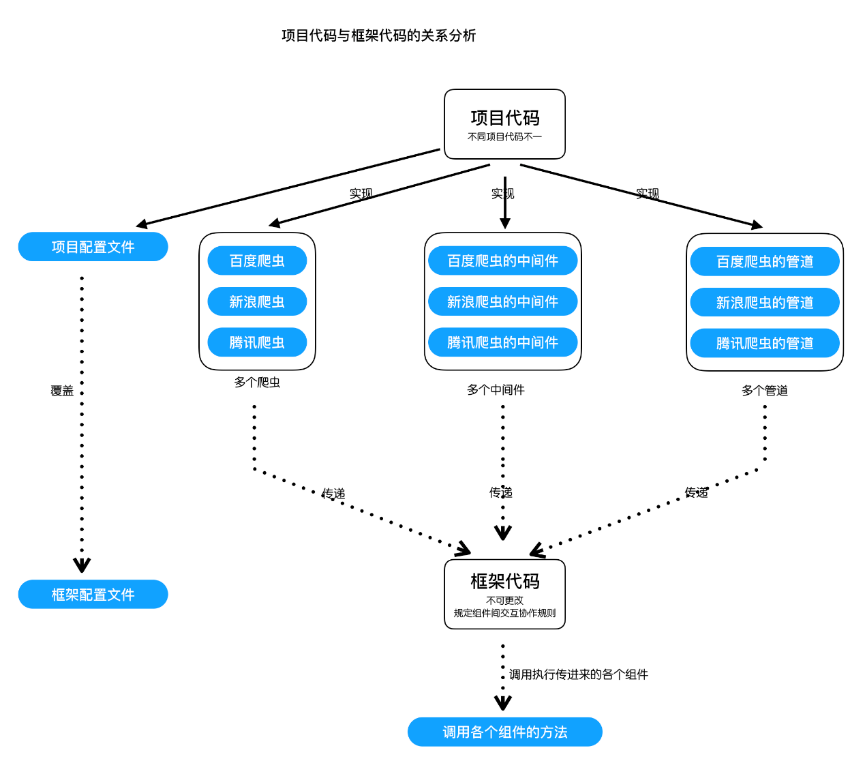
# 3.3. 框架功能完善

完善框架的基础功能

## 介绍

前面我们完成了框架的基本雏形，但是目前能够实现的功能非常简单，还达不到完成一个爬虫的基本需求，对应的需要在本节中对爬虫框架做进行进一步的完善

## 项目代码与框架代码完善分析

项目代码与框架代码的关系分析： 

项目中除了实现main.py以外，还需要实现：

* 项目配置文件
* 爬虫文件
* 管道文件
* 中间件文件

框架中还需要实现：

* 框架配置文件，并且需要实现导入项目配置文件，同时覆盖默认配置文件的属性
* 支持多个爬虫的传入以及使用
* 支持多个管道的传入以及使用
* 支持多个中间件的传入以及使用

## 内容

1. 日志的使用
2. 配置文件的实现
3. 具备较高的通用性：尽可能多的使用多数爬虫场景
4. 具备较好的扩展性：自定义功能或者组件的新增和维护
5. 请求的去重
6. 异步实现

# 3.3.1. 框架完善--日志模块使用

## 目标

* 掌握logging模块的使用
* 掌握在框架中使用日志功能

## 1. 为什么要使用添加日志功能

1. 能够方便的对程序进行调试
2. 能够记录程序的运行状态，包括错误

## 2. 日志模块简单使用

* 日志的等级

import logging

# 日志的五个等级，等级依次递增

# 默认是WARNING等级

logging.DEBUG

logging.INFO

logging.WARNING

logging.ERROR

logging.CRITICAL

# 设置日志等级

logging.basicConfig(level=logging.INFO)

# 使用

logging.debug('DEBUG')

logging.info('INFO')

logging.warning('WARNING')

logging.error('ERROR')

logging.critical('CRITICAL')

* 捕获异常信息到日志。这里主要需要进行捕获异常才能记录下完整的异常信息

try:

raise Exception("异常")

except Exception as e:

logging.exception(e)

* 日志的输出格式 对于日志的输出格式，我们能够进行自定义，包括输出的内容格式和时间格式

format格式说明：

%(name)s Logger的名字

%(levelno)s 数字形式的日志级别

%(levelname)s 文本形式的日志级别

%(pathname)s 调用日志输出函数的模块的完整路径名，可能没有

%(filename)s 调用日志输出函数的模块的文件名

%(module)s 调用日志输出函数的模块名

%(funcName)s 调用日志输出函数的函数名

%(lineno)d 调用日志输出函数的语句所在的代码行

%(created)f 当前时间，用UNIX标准的表示时间的浮 点数表示

%(relativeCreated)d 输出日志信息时的，自Logger创建以 来的毫秒数

%(asctime)s 字符串形式的当前时间。默认格式是 “2003-07-08 16:49:45,896”。逗号后面的是毫秒

%(thread)d 线程ID。可能没有

%(threadName)s 线程名。可能没有

%(process)d 进程ID。可能没有

%(message)s 用户输出的消息

* datefmt参数说明：
* %y 两位数的年份表示（00-99）
* %Y 四位数的年份表示（000-9999）
* %m 月份（01-12）
* %d 月内中的一天（0-31）
* %H 24小时制小时数（0-23）
* %I 12小时制小时数（01-12）
* %M 分钟数（00=59）
* %S 秒（00-59）
* %a 本地简化星期名称
* %A 本地完整星期名称
* %b 本地简化的月份名称
* %B 本地完整的月份名称
* %c 本地相应的日期表示和时间表示
* %j 年内的一天（001-366）
* %p 本地A.M.或P.M.的等价符
* %U 一年中的星期数（00-53）星期天为星期的开始
* %w 星期（0-6），星期天为星期的开始
* %W 一年中的星期数（00-53）星期一为星期的开始
* %x 本地相应的日期表示
* %X 本地相应的时间表示
* %Z 当前时区的名称
* %% %号本身

## 3. 利用logger封装日志模块

在scrapy\_plus目录下建立utils包 (utility：工具)，专门放置工具类型模块，如日志模块log.py 下面的代码内容是固定的，在任何地方都可以使用下面的代码实习日志内容的输出

# scrapy\_plus/utils/log.py

import sys

import logging

# 默认的配置

DEFAULT\_LOG\_LEVEL = logging.INFO # 默认等级

DEFAULT\_LOG\_FMT = '%(asctime)s %(filename)s [line:%(lineno)d] %(levelname)s: %(message)s' # 默认日志格式

DEFUALT\_LOG\_DATEFMT = '%Y-%m-%d %H:%M:%S' # 默认时间格式

DEFAULT\_LOG\_FILENAME = 'log.log' # 默认日志文件名称

class Logger(object):

def \_\_init\_\_(self):

# 1. 获取一个logger对象

self.\_logger = logging.getLogger()

# 2. 设置format对象

self.formatter = logging.Formatter(fmt=DEFAULT\_LOG\_FMT,datefmt=DEFUALT\_LOG\_DATEFMT)

# 3. 设置日志输出

# 3.1 设置文件日志模式

self.\_logger.addHandler(self.\_get\_file\_handler(DEFAULT\_LOG\_FILENAME))

# 3.2 设置终端日志模式

self.\_logger.addHandler(self.\_get\_console\_handler())

# 4. 设置日志等级

self.\_logger.setLevel(DEFAULT\_LOG\_LEVEL)

def \_get\_file\_handler(self, filename):

'''返回一个文件日志handler'''

# 1. 获取一个文件日志handler

filehandler = logging.FileHandler(filename=filename,encoding="utf-8")

# 2. 设置日志格式

filehandler.setFormatter(self.formatter)

# 3. 返回

return filehandler

def \_get\_console\_handler(self):

'''返回一个输出到终端日志handler'''

# 1. 获取一个输出到终端日志handler

console\_handler = logging.StreamHandler(sys.stdout)

# 2. 设置日志格式

console\_handler.setFormatter(self.formatter)

# 3. 返回handler

return console\_handler

@property

def logger(self):

return self.\_logger

# 初始化并配一个logger对象，达到单例的

# 使用时，直接导入logger就可以使用

logger = Logger().logger

## 4. 在框架中使用日志模块

使用参考

# scrapy\_plus/core/engine.py

from datetime import datetime

from scrapy\_plus.http.request import Request # 导入Request对象

from scrapy\_plus.middlewares.spider\_middlewares import SpiderMiddleware

from scrapy\_plus.middlewares.downloader\_middlewares import DownloaderMiddleware

from scrapy\_plus.utils.log import logger # 导入logger

from .spider import Spider

from .scheduler import Scheduler

from .downloader import Downloader

from .pipeline import Pipeline

class Engine(object):

......

def start(self):

'''启动整个引擎'''

start = datetime.now() # 起始时间

logger.info("开始运行时间：%s" % start) # 使用日志记录起始运行时间

self.\_start\_engine()

stop = datetime.now() # 结束时间

logger.info("开始运行时间：%s" % stop) # 使用日志记录结束运行时间

logger.info("耗时：%.2f" % (stop - start).total\_seconds()) # 使用日志记录运行耗时

......

## 小结

* 本小结重点
  + 掌握日志的使用方法
  + 能够在框架中封装日志模块并且进行使用

# 3.3.2. 框架完善--配置文件实现

## 目标

* 完成项目和框架中配置文件的编写和使用
* 理解python中import内容的顺序

## 1. 实现框架的默认配置文件

在scrapy\_plus下建立conf包文件夹在它下面建立default\_settings.py：设置默认配置的配置

import logging

# 默认的日志配置

DEFAULT\_LOG\_LEVEL = logging.INFO # 默认等级

DEFAULT\_LOG\_FMT = '%(asctime)s %(filename)s[line:%(lineno)d] \

%(levelname)s: %(message)s' # 默认日志格式

DEFUALT\_LOG\_DATEFMT = '%Y-%m-%d %H:%M:%S' # 默认时间格式

DEFAULT\_LOG\_FILENAME = 'log.log' # 默认日志文件名称

再在conf下创建settings.py文件

# scrapy\_plus/conf/settings

from .default\_settings import \* # 全部导入默认配置文件的属性

## 2.在框架中使用

利用框架配置文件改写log.py

# scrapy\_plus/utils/log.py

import sys

import logging

from scrapy\_plus.conf import settings # 导入框架的settings文件

class Logger(object):

def \_\_init\_\_(self):

# 1. 获取一个logger对象

self.\_logger = logging.getLogger()

# 2. 设置format对象

self.formatter = logging.Formatter(fmt=settings.DEFAULT\_LOG\_FMT,datefmt=settings.DEFUALT\_LOG\_DATEFMT)

# 3. 设置日志输出

# 3.1 设置文件日志模式

self.\_logger.addHandler(self.\_get\_file\_handler(settings.DEFAULT\_LOG\_FILENAME))

# 3.2 设置终端日志模式

self.\_logger.addHandler(self.\_get\_console\_handler())

# 4. 设置日志等级

self.\_logger.setLevel(settings.DEFAULT\_LOG\_LEVEL)

......

## 3. 创建项目配置文件，并实现修改框架默认配置文件属性

项目文件夹下创建项目配置文件settings.py:

# project\_dir/settings.py

# 修改默认日志文件名称

DEFAULT\_LOG\_FILENAME = '日志.log' # 默认日志文件名称

修改框架的settings.py文件，实现修改默认配置文件属性的目的

# scrapy\_plus/conf/settings

from .default\_settings import \* # 全部导入默认配置文件的属性

# 这里导入的settings，是项目文件夹的settings文件

from settings import \*

## 小结

* 本小结重点
  + 掌握python中import的顺序
  + 完成项目和框架中配置文件的编写

# 3.3.3. 框架完善--多爬虫实现之一--多请求

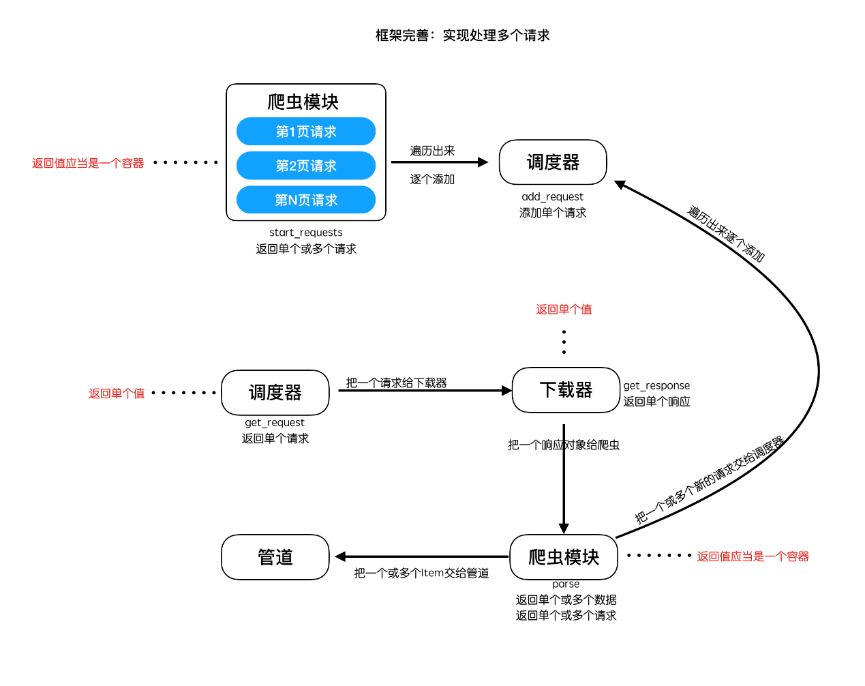
## 目标

* 实现在框架中能够解析多送请求
* 掌握yield和生成器的使用方法

## 1.需求分析

在发送关于start\_url中的请求的时候，往往我们的请求并不是只有一个，而且在解析了响应之后，可能需要继续构造请求并且发送，那么对应的需要在引擎中进行修改。

多请求实现分析：



## 2. 项目中实现爬虫文件

* 在main.py同级目录下建立spiders.py，存放定义的爬虫类

# project\_dir/spiders.py

from scrapy\_plus.core.spider import Spider

# 继承框架的爬虫基类

class BaiduSpider(Spider):

start\_url = 'http://www.baidu.com' # 设置初始请求url

修改main.py

# project\_dir/main.py

from scrapy\_plus.core.engine import Engine # 导入引擎

from spiders import BaiduSpider

if \_\_name\_\_ == '\_\_main\_\_':

spider = BaiduSpider() # 实例化爬虫对象

engine = Engine(spider) # 传入爬虫对象

engine.start() # 启动引擎

* 修改engine.py，设置为接收外部传入的爬虫对象

# scrapy\_plus/core/engine.py

...

class Engine(object):

def \_\_init\_\_(self, spider): # 接收外部传入的爬虫对象

self.spider = spider # 爬虫对象

self.scheduler = Scheduler() # 初始化调度器对象

self.downloader = Downloader() # 初始化下载器对象

self.pipeline = Pipeline() # 初始化管道对象

self.spider\_mid = SpiderMiddleware()

self.downloader\_mid = DownloaderMiddleware()

...

## 2. 实现发起多个请求

### 2.1 修改框架的爬虫组件文件 spider.py:

* 设置为初始请求url为多个
* 修改start\_requests方法，将返回多个请求对象
* 利用生成器方式实现start\_requests，提高程序的资源消耗

# scrapy\_plus/core/spider.py

'''爬虫组件封装'''

from scrapy\_plus.item import Item # 导入Item对象

from scrapy\_plus.http.request import Request # 导入Request对象

class Spider(object):

'''

1. 构建请求信息(初始的)，也就是生成请求对象(Request)

2. 解析响应对象，返回数据对象(Item)或者新的请求对象(Request)

'''

start\_urls = [] # 默认初始请求地址

# def start\_requests(self):

# '''构建初始请求对象并返回'''

# request\_list = []

# for url in self.start\_urls:

# request\_list.append(Request(url))

# return request\_list

# 利用生成器方式实现，提高程序的资源消耗

def start\_requests(self):

'''构建初始请求对象并返回'''

for url in self.start\_urls:

yield Request(url)

def parse(self, response):

'''解析请求

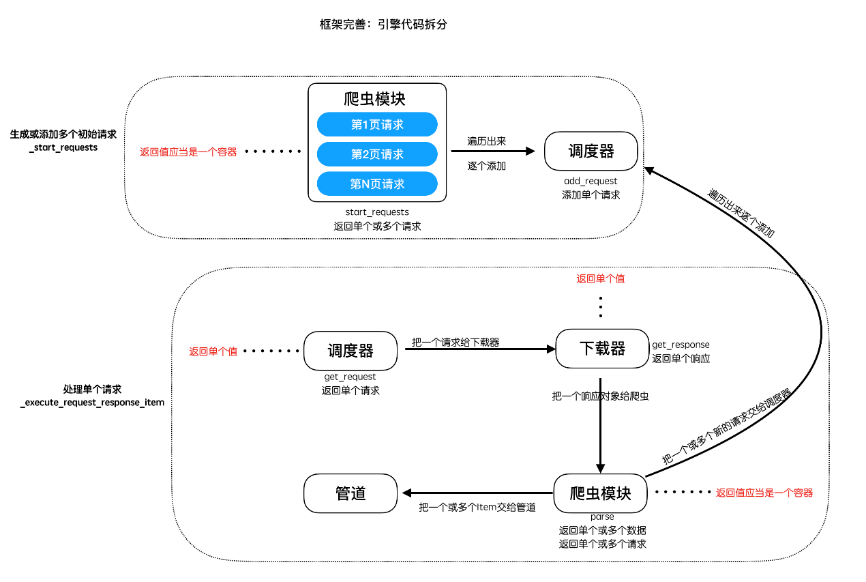
并返回新的请求对象、或者数据对象

返回值应当是一个容器，如start\_requests返回值方法一样，改为生成器即可

'''

yield Item(response.body) # 返回item对象 改为生成器即可

### 2.2 修改引擎 engine.py:



* 将代码拆分为两个方法，便于维护，提高代码可读性
* 统计总共完成的响应数
* 设置程序退出条件：当总响应数等于总请求数时，退出
* 实现处理start\_requests方法返回多个请求的功能
* 实现处理parse解析函数返回多个对象的功能

# scheduler/core/engine.py

'''引擎

a. 对外提供整个的程序的入口

b. 依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)

'''

from scrapy\_plus.http.request import Request # 导入Request对象

from scrapy\_plus.middlewares.spider\_middlewares import SpiderMiddleware

from scrapy\_plus.middlewares.downloader\_middlewares import DownloaderMiddleware

from .spider import Spider

from .scheduler import Scheduler

from .downloader import Downloader

from .pipeline import Pipeline

class Engine(object):

def \_\_init\_\_(self, spider): # 接收外部传入的爬虫对象

......

self.total\_response\_number = 0 # 统计响应总数

def \_start\_requests(self):

'''向调度器添加初始请求'''

# 1. 爬虫模块发出初始请求

for start\_request in self.spider.start\_requests():

# 2. 把初始请求添加给调度器

# 利用爬虫中间件预处理请求对象

start\_request = self.spider\_mid.process\_request(start\_request)

self.scheduler.add\_request(start\_request)

def \_execute\_request\_response\_item(self):

'''根据请求、发起请求获取响应、解析响应、处理响应结果'''

# 3. 从调度器获取请求对象，交给下载器发起请求，获取一个响应对象

request = self.scheduler.get\_request()

if request is None:

return

# 利用下载器中间件预处理请求对象

request = self.downloader\_mid.process\_request(request)

# 4. 利用下载器发起请求

response = self.downloader.get\_response(request)

# 利用下载器中间件预处理响应对象

response = self.downloader\_mid.process\_response(response)

# 5. 利用爬虫的解析响应的方法，处理响应，得到结果

results = self.spider.parse(response) # parse函数的返回值是一个容器，如列表或者生成器对象

for result in results:

# 6. 判断结果对象

# 6.1 如果是请求对象，那么就再交给调度器

if isinstance(result, Request):

# 利用爬虫中间件预处理请求对象

result = self.spider\_mid.process\_request(result)

self.scheduler.add\_request(result)

# 6.2 否则，就交给管道处理

else:

# 利用爬虫中间件预处理数据对象

result = self.spider\_mid.process\_item(result)

self.pipeline.process\_item(result)

# 统计响应总数

self.total\_response\_number += 1

def \_start\_engine(self):

'''依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)'''

# 向调度器添加初始请求

self.\_start\_requests()

# 设置循环，处理多个请求

while True:

# 根据请求、发起请求获取响应、解析响应、处理响应结果

self.\_execute\_request\_response\_item()

# 设置退出条件：当请求数和响应数相等时，退出循环

if self.total\_response\_number >= self.scheduler.total\_request\_number:

break

### 2.3 修改调度器 scheduler.py：

* 将从队列获取请求对象设置为非阻塞，否则会造成程序无法退出
* 统计请求总数，用于判断程序退出

# scrapy\_plus/core/scheduler.py

'''调度器模块封装'''

# 利用six模块实现py2和py3兼容

from six.moves.queue import Queue

class Scheduler(object):

'''

1. 缓存请求对象(Request)，并为下载器提供请求对象，实现请求的调度

2. 对请求对象进行去重判断

'''

def \_\_init\_\_(self):

self.queue = Queue()

# 记录总共的请求数

self.total\_request\_number = 0

def add\_request(self, request):

'''添加请求对象'''

self.queue.put(request)

self.total\_request\_number += 1 # 统计请求总数

def get\_request(self):

'''获取一个请求对象并返回'''

try:

request = self.queue.get(False)

except:

return None

else:

return request

def \_filter\_request(self):

'''请求去重'''

pass

## 小结

本小结重点

* + 掌握生成器的使用
  + 完成核心代码的修改

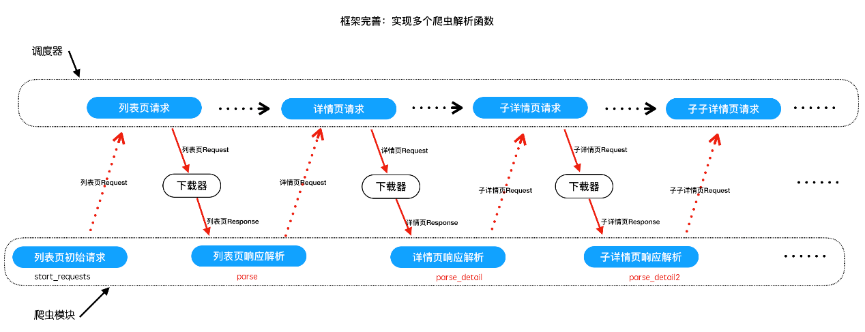
# 3.3.4. 框架完善--多爬虫实现之二--多个解析函数

多爬虫实现之二 -- 爬虫实现多个解析函数

## 目标

* 完成spider中如果解析函数调用的封装
* 掌握getattr的方法
* 完成通过meta在不通过的解析函数中传递数据的方法

## 1. 爬虫实现多个解析函数的意义



## 2. 响应对象的解析方法封装

为response对象封装xpath、正则、json、等方法和属性，以支持对数据的解析和提取

# scrapy\_plus/http/response.py

import re

import json

from lxml import etree

class Response(object):

'''框架内置Response对象'''

def \_\_init\_\_(self, url, status\_code, headers, body):

self.url = url # 响应url

self.status\_code = status\_code # 响应状态码

self.headers = headers # 响应头

self.body = body # 响应体

def xpath(self, rule):

'''提供xpath方法'''

html = etree.HTML(self.body)

return html.xpath(rule)

@property

def json(self):

'''提供json解析

如果content是json字符串，是才有效

'''

return json.loads(self.body)

def re\_findall(self, rule, data=None):

'''封装正则的findall方法'''

if data is None:

data = self.body

return re.findall(rule, data)

## 3. 增加爬虫：豆瓣电影top250

# project\_dir/spiders.py

from scrapy\_plus.core.spider import Spider

from scrapy\_plus.http.request import Request

from scrapy\_plus.item import Item

# 继承框架的爬虫基类

class BaiduSpider(Spider):

start\_urls = ['http://www.baidu.com'] # 设置初始请求url

class DoubanSpider(Spider):

start\_urls = [] # 重写start\_requests方法后，这个属性就没有设置的必要了

def start\_requests(self):

# 重写start\_requests方法，返回多个请求

base\_url = 'http://movie.douban.com/top250?start='

for i in range(0, 250, 25): # 逐个返回第1-10页的请求属相

url = base\_url + str(i)

yield Request(url)

def parse(self, response):

'''解析豆瓣电影top250列表页'''

title\_list = [] # 存储所有的

for li in response.xpath("//ol[@class='grid\_view']/li"): # 遍历每一个li标签

title = li.xpath(".//span[@class='title'][1]/text()") # 提取该li标下的 标题

title\_list.append(title[0])

yield Item(title\_list) # 返回标题

## 4. 实现多个解析函数

由于现在不同的请求对应不同的解析函数，因此需要为请求对象指明它的解析函数，因此为请求对象增加一个属性

# scrapy\_plus/http/request.py

class Request(object):

'''框架内置请求对象，设置请求信息'''

def \_\_init\_\_(self, url, method='GET', headers=None, params=None, data=None, parse='parse'):

self.url = url # 请求地址

self.method = method # 请求方法

self.headers = headers # 请求头

self.params = params # 请求参数

self.data = data # 请求体

self.parse = parse # 指明它的解析函数, 默认是parse方法

self.meta = None

## 5. 在引擎中需要动态的判断和获取对应的解析函数

# scrapy\_plus/core/engine.py

class Engine(object):

......

def \_execute\_request\_response\_item(self):

'''根据请求、发起请求获取响应、解析响应、处理响应结果'''

......

# 5. 利用爬虫的解析响应的方法，处理响应，得到结果

parse = getattr(self.spider, request.parse) # 获取对应的解析函数

results = parse(response) # parse函数的返回值是一个容器，如列表或者生成器对象

results.meta = request.meta

......

## 6. 修改豆瓣爬虫，发起详情页的请求：

# project\_dir/spiders.py

......

class DoubanSpider(Spider):

start\_urls = [] # 重写start\_requests方法后，这个属性就没有设置的必要了

def start\_requests(self):

# 重写start\_requests方法，返回多个请求

base\_url = 'http://movie.douban.com/top250?start='

for i in range(0, 250, 25): # 逐个返回第1-10页的请求属相

url = base\_url + str(i)

yield Request(url)

def parse(self, response):

'''解析豆瓣电影top250列表页'''

title\_list = [] # 存储所有的

for li in response.xpath("//ol[@class='grid\_view']/li"): # 遍历每一个li标签

# title = li.xpath(".//span[@class='title'][1]/text()") # 提取该li标下的 标题

# title\_list.append(title[0])

detail\_url = li.xpath(".//div[@class='info']/div[@class='hd']/a/@href")[0]

yield Request(detail\_url, parse="parse\_detail") # 发起详情页的请求，并指定解析函数是parse\_detail方法

# yield Item(title\_list) # 返回标题

def parse\_detail(self, response):

'''解析详情页'''

print('详情页url：', response.url) # 打印一下响应的url

return [] # 由于必须返回一个容器，这里返回一个空列表

## 小结

* 本小结重点
  + 掌握getattr的使用
  + 完成代码的重构，实现多个解析函数封装

# 3.3.5. 框架完善--多爬虫实现之三--多爬虫文件

多爬虫实现之三 -- 多爬虫文件

## 目标

* 优化现有的爬虫结构，实现同时开始执行多个爬虫

## 1. 为什么需要优化现有的爬虫结构

当爬虫比较少的时候，我们的项目结构相对合理，但是当要抓取的网站比较多的时候，可以借鉴scrapy的方法，把不同网站的爬虫分别在不同的py文件中编写，之后放在一个目录下；同时，我们很多时候还希望能够有同时启动项目中的所有的爬虫

## 2. 将多个爬虫类分离为多个爬虫文件爬虫文件

为了解耦合，应将每个站点的爬虫写为单独一个py文件，因此更改一下放置爬虫的模块，结构如下：

- 项目文件夹

-- main.py

-- spiders

-- \_\_init\_\_.py

-- baidu.py

-- douban.py

-- settings.py

其中baidu.py和douban.py分别是抓取百度和豆瓣的爬虫文件

* baidu.py:

# project\_dir/spiders/baidu.py

from scrapy\_plus.core.spider import Spider

# 继承框架的爬虫基类

class BaiduSpider(Spider):

start\_urls = ['http://www.baidu.com'] # 设置初始请求url

* douban.py: 抓取豆瓣电影top250的列表页信息

# project\_dir/spiders/douban.py

from scrapy\_plus.core.spider import Spider

from scrapy\_plus.http.request import Request

from scrapy\_plus.item import Item

class DoubanSpider(Spider):

start\_urls = [] # 重写start\_requests方法后，这个属性就没有设置的必要了

def start\_requests(self):

# 重写start\_requests方法，返回多个请求

base\_url = 'http://movie.douban.com/top250?start='

for i in range(0, 250, 25): # 逐个返回第1-10页的请求属相

url = base\_url + str(i)

yield Request(url)

def parse(self, response):

'''解析豆瓣电影top250列表页'''

title\_list = [] # 存储所有的

for li in response.xpath("//ol[@class='grid\_view']/li"): # 遍历每一个li标签

# title = li.xpath(".//span[@class='title'][1]/text()") # 提取该li标下的 标题

# title\_list.append(title[0])

detail\_url = li.xpath(".//div[@class='info']/div[@class='hd']/a/@href")[0]

yield Request(detail\_url, parse="parse\_detail") # 发起详情页的请求，并指定解析函数是parse\_detail方法

# yield Item(title\_list) # 返回标题

def parse\_detail(self, response):

'''解析详情页'''

print('详情页url：', response.url) # 打印一下响应的url

return [] # 由于必须返回一个容器，这里返回一个空列表

* 对main.py进行相应修改，测试新增的douban爬虫

from scrapy\_plus.core.engine import Engine # 导入引擎

from spiders.baidu import BaiduSpider

from spiders.douban import DoubanSpider

if \_\_name\_\_ == '\_\_main\_\_':

# spider = BaiduSpider() # 实例化爬虫对象

douban\_spider = DoubanSpider() # 实例化爬虫对象

engine = Engine(douban\_spider) # 传入爬虫对象

engine.start() # 启动引擎

## 3. 同时执行多个不同的爬虫

如把豆瓣爬虫和百度爬虫一起启动并执行

传入形式：并用字典的形式传入多个爬虫：

* main.py

# project\_dir/main.py

from scrapy\_plus.core.engine import Engine # 导入引擎

from spiders.baidu import BaiduSpider

from spiders.douban import DoubanSpider

if \_\_name\_\_ == '\_\_main\_\_':

baidu\_spider = BaiduSpider() # 实例化爬虫对象

douban\_spider = DoubanSpider() # 实例化爬虫对象

spiders = {'baidu':baidu\_spider, 'douban':douban\_spider}

engine = Engine(spiders) # 传入爬虫对象

engine.start() # 启动引擎

在引擎中用到爬虫对象的地方都要做相应的修改

* engine.py：

'''引擎

a. 对外提供整个的程序的入口

b. 依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)

'''

......

class Engine(object):

def \_\_init\_\_(self, spiders): # 接收外部传入的多个爬虫对象

self.spiders = spiders # 爬虫对象

......

......

def \_start\_requests(self):

'''向调度器添加初始请求'''

# 1. 爬虫模块发出初始请求

for spider\_name, spider in self.spiders.items():

for start\_request in spider.start\_requests():

# 2. 把初始请求添加给调度器

# 利用爬虫中间件预处理请求对象

start\_request = self.spider\_mid.process\_request(start\_request)

start\_request.spider\_name = spider\_name #为请求对象绑定它所属的爬虫的名称

self.scheduler.add\_request(start\_request)

def \_execute\_request\_response\_item(self):

'''根据请求、发起请求获取响应、解析响应、处理响应结果'''

......

spider = self.spiders[request.spider\_name] # 根据请求的spider\_name属性，获取对应的爬虫对象

# 5. 利用爬虫的解析响应的方法，处理响应，得到结果

parse = getattr(spider, request.parse) # 获取对应的解析函数

results = parse(response) # parse函数的返回值是一个容器，如列表或者生成器对象

for result in results:

# 6. 判断结果对象

# 6.1 如果是请求对象，那么就再交给调度器

if isinstance(result, Request):

# 利用爬虫中间件预处理请求对象

result = self.spider\_mid.process\_request(result)

result.spider\_name = request.spider\_name # 为请求对象绑定它所属的爬虫的名称

self.scheduler.add\_request(result)

# 6.2 否则，就交给管道处理

......

......

安装代码，并运行main.py，直到调试成功

## 4. 再次改进，将每个爬虫的名称直接设置为爬虫类的一个属性

参考：

class BaiduSpider(Spider):

name = 'baidu' # 为爬虫命名

start\_urls = ['http://www.baidu.com'] # 设置初始请求url

'''那么main.py就可以按照这样的方式设定key值'''

spiders = {BaiduSpider.name: baidu\_spider, DoubanSpider.name: douban\_spider}

## 小结

* 本小结重点
  + 完成对现有的项目结构的更改
  + 完成对爬虫和引擎的代码的修改

# 3.3.6. 框架完善--实现多个管道

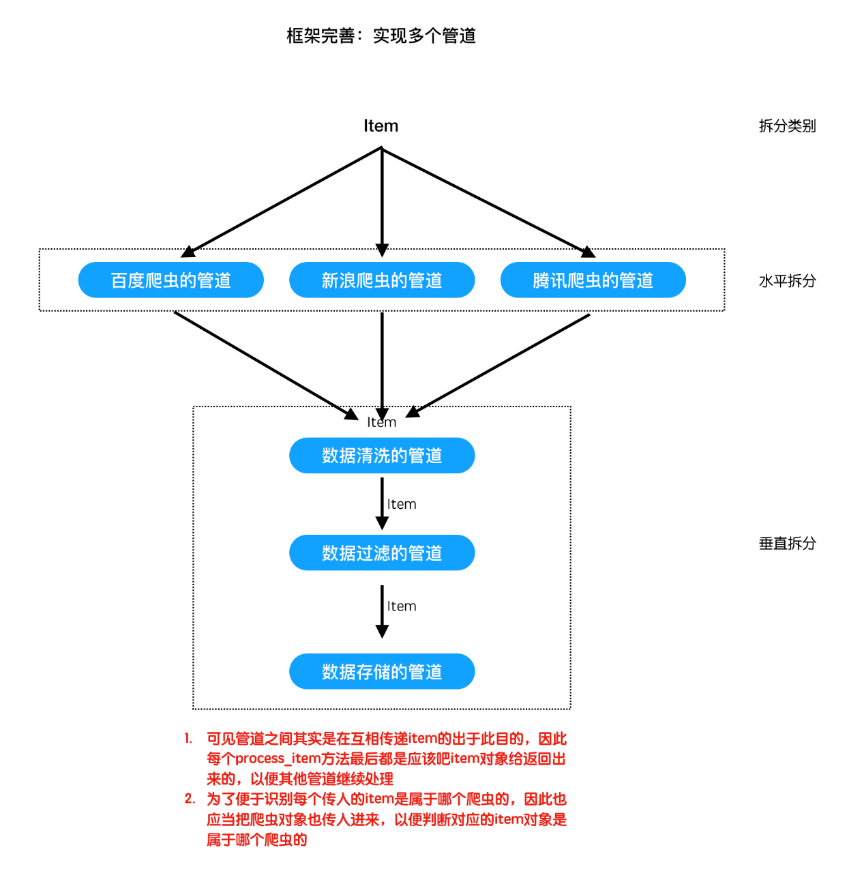
实现多个管道

## 目标

* 实现对引擎的修改，达到数据通过多个管道的目的

## 1. 为什么需要多个管道

同爬虫文件一样，不同的爬虫可能需要不同的管道文件，因此管道文件需要在项目中进行实现



## 2. 项目文件夹中实现管道文件

在项目文件夹下建立pipelines.py文件，不同在于：

* 这里的process\_item必须把item对象最后再返回回来，因为是多个管道文件的设置了
* 需要增加一个参数，也就是传入爬虫对象，以此来判断当前item是属于那个爬虫对象的

# project\_dir/pipelines.py

from spiders.baidu import BaiduSpider

from spiders.douban import DoubanSpider

class BaiduPipeline(object):

# 这里有所不同的是，需要增加一个参数，也就是传入爬虫对象

# 以此来判断当前item是属于那个爬虫对象的

def process\_item(self, item, spider):

'''处理item'''

if isinstance(spider, BaiduSpider):

print("百度爬虫的数据：", item.data)

return item # 最后必须返回item

class DoubanPipeline(object):

# 这里有所不同的是，需要增加一个参数，也就是传入爬虫对象

# 以此来判断当前item是属于那个爬虫对象的

def process\_item(self, item, spider):

'''处理item'''

if isinstance(spider, DoubanSpider):

print("豆瓣爬虫的数据：", item.data)

return item # 最后必须返回item

## 2. 修改引擎的代码

* 管道对象将从外部传入
* 调用管道的process\_item方法时，需要遍历出管道
* 并且需要传递第二个参数，爬虫对象

# scrapy\_plus/core/engine.py

......

class Engine(object):

'''

a. 对外提供整个的程序的入口

b. 依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)

'''

def \_\_init\_\_(self, spiders, pipelines=[]):

self.spiders = spiders # 接收外部传入的爬虫对象

self.scheduler = Scheduler() # 初始化调度器对象

self.downloader = Downloader() # 初始化下载器对象

self.pipelines = pipelines # 初始化管道对象

self.spider\_mid = SpiderMiddleware() # 初始化爬虫中间件对象

self.downloader\_mid = DownloaderMiddleware() # 初始化下载器中间件对象

self.total\_response\_number = 0

......

def \_execute\_request\_response\_item(self):

'''根据请求、发起请求获取响应、解析响应、处理响应结果'''

# 3. 从调度器获取请求对象，交给下载器发起请求，获取一个响应对象

request = self.scheduler.get\_request()

if request is None:

return

# 利用下载器中间件预处理请求对象

request = self.downloader\_mid.process\_request(request)

# 4. 利用下载器发起请求

response = self.downloader.get\_response(request)

# 利用下载器中间件预处理响应对象

response = self.downloader\_mid.process\_response(response)

spider = self.spiders[request.spider\_name] # 根据请求的spider\_name属性，获取对应的爬虫对象

# 5. 利用爬虫的解析响应的方法，处理响应，得到结果

parse = getattr(spider, request.parse) # 获取对应的解析函数

results = parse(response) # parse函数的返回值是一个容器，如列表或者生成器对象

for result in results:

# 6. 判断结果对象

# 6.1 如果是请求对象，那么就再交给调度器

if isinstance(result, Request):

# 利用爬虫中间件预处理请求对象

result = self.spider\_mid.process\_request(result)

result.spider\_name = request.spider\_name # 为请求对象绑定它所属的爬虫的名称

self.scheduler.add\_request(result)

# 6.2 否则，就交给管道处理

else:

# 利用爬虫中间件预处理数据对象

result = self.spider\_mid.process\_item(result)

for pipeline in self.pipelines: # 多个管道对象，轮流处理item对象

result = pipeline.process\_item(result, spider)

# 统计响应总数

self.total\_response\_number += 1

......

## 3.修改main.py

为引擎传入项目中的管道对象:

# project\_dir/main.py

from scrapy\_plus.core.engine import Engine # 导入引擎

from spiders.baidu import BaiduSpider

from spiders.douban import DoubanSpider

from pipeline import BaiduPipeline, DoubanPipeline

if \_\_name\_\_ == '\_\_main\_\_':

baidu\_spider = BaiduSpider() # 实例化爬虫对象

douban\_spider = DoubanSpider() # 实例化爬虫对象

spiders = {BaiduSpider.name: baidu\_spider, DoubanSpider.name: douban\_spider} # 爬虫们

pipelines = [BaiduPipeline(), DoubanPipeline()] # 管道们

engine = Engine(spiders, pipelines=pipelines) # 传入爬虫对象

engine.start() # 启动引擎

## 小结

* 本小结重点
  + 完成代码的重构，实现多个管道的效果

# 3.3.7. 框架完善--实现多个中间件

实现项目中传入多个中间件

## 目标

* 完成代码的重构，实现多个中间件的效果

## 1. 为什么需要多个中间件

不同的中间件可以实现对请求或者是响应对象进行不同的处理，实现结构，让逻辑更加清晰

## 2. 在项目文件夹中创建middlewares文件

* 项目文件夹中的spider\_middlewares.py：

class TestSpiderMiddleware1(object):

def process\_request(self, request):

'''处理请求头，添加默认的user-agent'''

print("TestSpiderMiddleware1: process\_request")

return request

def process\_item(self, item):

'''处理数据对象'''

print("TestSpiderMiddleware1: process\_item")

return item

class TestSpiderMiddleware2(object):

def process\_request(self, request):

'''处理请求头，添加默认的user-agent'''

print("TestSpiderMiddleware2: process\_request")

return request

def process\_item(self, item):

'''处理数据对象'''

print("TestSpiderMiddleware2: process\_item")

return item

* 项目文件夹中的downloader\_middlewares.py:

class TestDownloaderMiddleware1(object):

def process\_request(self, request):

'''处理请求头，添加默认的user-agent'''

print("TestDownloaderMiddleware1: process\_request")

return request

def process\_response(self, item):

'''处理数据对象'''

print("TestDownloaderMiddleware1: process\_response")

return item

class TestDownloaderMiddleware2(object):

def process\_request(self, request):

'''处理请求头，添加默认的user-agent'''

print("TestDownloaderMiddleware2: process\_request")

return request

def process\_response(self, item):

'''处理数据对象'''

print("TestDownloaderMiddleware2: process\_response")

return item

## 2. 修改项目文件夹中的main.py

为引擎传入多个中间件

from scrapy\_plus.core.engine import Engine # 导入引擎

from spiders.baidu import BaiduSpider

from spiders.douban import DoubanSpider

from pipeline import BaiduPipeline, DoubanPipeline

from spider\_middlewares import TestSpiderMiddleware1, TestSpiderMiddleware2

from downloader\_middlewares import TestDownloaderMiddleware1, TestDownloaderMiddleware2

if \_\_name\_\_ == '\_\_main\_\_':

baidu\_spider = BaiduSpider() # 实例化爬虫对象

douban\_spider = DoubanSpider() # 实例化爬虫对象

spiders = {BaiduSpider.name: baidu\_spider, DoubanSpider.name: douban\_spider} # 爬虫们

pipelines = [BaiduPipeline(), DoubanPipeline()] # 管道们

spider\_mids = [TestSpiderMiddleware1(), TestSpiderMiddleware2()] # 多个爬虫中间件

downloader\_mids = [TestDownloaderMiddleware1(), TestDownloaderMiddleware2()] # 多个下载中间件

engine = Engine(spiders, pipelines=pipelines, spider\_mids=spider\_mids, downloader\_mids=downloader\_mids) # 传入爬虫对象

engine.start() # 启动引擎

## 3. 因此相应的的修改engine.py

改为使用多个中间件

# scrapy\_plus/core/engine.py

class Engine(object):

'''

a. 对外提供整个的程序的入口

b. 依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)

'''

def \_\_init\_\_(self, spiders, pipelines=[], spider\_mids=[], downloader\_mids=[]):

self.spiders = spiders # 接收外部传入的爬虫对象

self.scheduler = Scheduler() # 初始化调度器对象

self.downloader = Downloader() # 初始化下载器对象

self.pipelines = pipelines # 初始化管道对象

self.spider\_mids = spider\_mids # 初始化爬虫中间件对象

self.downloader\_mids = downloader\_mids # 初始化下载器中间件对象

self.total\_response\_number = 0

......

def \_start\_requests(self):

'''向调度器添加初始请求'''

# 1. 爬虫模块发出初始请求

for spider\_name, spider in self.spiders.items():

for start\_request in spider.start\_requests():

# 2. 把初始请求添加给调度器

# 利用爬虫中间件预处理请求对象

for spider\_mid in self.spider\_mids:

start\_request = spider\_mid.process\_request(start\_request)

start\_request.spider\_name = spider\_name # 为请求对象绑定它所属的爬虫的名称

self.scheduler.add\_request(start\_request)

def \_execute\_request\_response\_item(self):

'''根据请求、发起请求获取响应、解析响应、处理响应结果'''

# 3. 从调度器获取请求对象，交给下载器发起请求，获取一个响应对象

request = self.scheduler.get\_request()

if request is None:

return

# 利用下载器中间件预处理请求对象

for downloader\_mid in self.downloader\_mids:

request = downloader\_mid.process\_request(request)

# 4. 利用下载器发起请求

response = self.downloader.get\_response(request)

# 利用下载器中间件预处理响应对象

for downloader\_mid in self.downloader\_mids:

response = downloader\_mid.process\_response(response)

spider = self.spiders[request.spider\_name] # 根据请求的spider\_name属性，获取对应的爬虫对象

# 5. 利用爬虫的解析响应的方法，处理响应，得到结果

parse = getattr(spider, request.parse) # 获取对应的解析函数

results = parse(response) # parse函数的返回值是一个容器，如列表或者生成器对象

for result in results:

# 6. 判断结果对象

# 6.1 如果是请求对象，那么就再交给调度器

if isinstance(result, Request):

# 利用爬虫中间件预处理请求对象

for spider\_mid in self.spider\_mids:

result = spider\_mid.process\_request(result)

result.spider\_name = request.spider\_name # 为请求对象绑定它所属的爬虫的名称

self.scheduler.add\_request(result)

# 6.2 否则，就交给管道处理

else:

# 利用爬虫中间件预处理数据对象

for spider\_mid in self.spider\_mids:

result = spider\_mid.process\_item(result)

for pipeline in self.pipelines: # 多个管道对象，轮流处理item对象

result = pipeline.process\_item(result, spider)

# 统计响应总数

self.total\_response\_number += 1

......

## 小结

* 本小结重点
  + 完成代码的重构，实现多个中间件的效果

# 3.3.8. 框架完善--实现动态模块导入

动态导入模块

## 目标

* 掌握模块的动态导入的方法
* 完成对现有代码的重构

## 1. 目前代码存在的问题

通过前面的代码编写，我们已经能够完成大部分的任务，但是在main.py 中的代码非常臃肿，对应的我们可以再settings.py 配置哪些爬虫，管道，中间件需要开启，能够让整个代码的逻辑更加清晰

## 2. 在settings中设置SPIDER，MIDDLEWARES

* 利用在配置文件中设置需要启用的爬虫类、管道类、中间件类，如下：

# 项目中的settings.py

......

# 增加以下信息：

# 启用的爬虫类

SPIDERS = [

'spiders.baidu.BaiduSpider',

'spiders.douban.DoubanSpider'

]

# 启用的管道类

PIPELINES = [

'pipelines.BaiduPipeline',

'pipelines.DoubanPipeline'

]

# 启用的爬虫中间件类

SPIDER\_MIDDLEWARES = []

# 启用的下载器中间件类

DOWNLOADER\_MIDDLEWARES = []

* 利用importlib模块，在引擎中动态导入并实例化

class Engine(object):

def \_\_init\_\_(self):

self.scheduler = Scheduler() # 初始化调度器对象

self.downloader = Downloader() # 初始化下载器对象

self.spiders = self.\_auto\_import\_instances(settings.SPIDERS, True) # 动态导入并实例化爬虫对象

self.pipelines = self.\_auto\_import\_instances(settings.PIPELINES) # 动态导入并实例化管道对象

self.spider\_mids = self.\_auto\_import\_instances(settings.SPIDER\_MIDDLEWARES) # 动态导入并实例化爬虫中间件对象

self.downloader\_mids = self.\_auto\_import\_instances(settings.DOWNLOADER\_MIDDLEWARES) # 动态导入并实例化下载器中间件对象

self.total\_response\_number = 0 # 统计响应总数

self.pool = Pool() # 创建线程池对象

self.running = False # 记录是否退出程序的状态

......

def \_auto\_import\_instances(self, path=[], isspider=False):

'''通过配置文件，动态导入类并实例化

path: 表示配置文件中配置的导入类的路径

isspider: 由于爬虫需要返回的是一个字典，因此对其做对应的判断和处理

'''

# 该方法不仅实例化动态导入爬虫，还有管道和中间件，而前者返回应是字典类型

if isspider is True:

instances = {}

else:

instances = [] # 存储对应类的实例对象

for p in path:

module\_name = p[:p.rfind(".")] # 取出模块名称

cls\_name = p[p.rfind(".")+1:] # 取出类名称

ret = importlib.import\_module(module\_name) # 动态导入爬虫模块

cls = getattr(ret, cls\_name) # 根据类名称获取类对象

if isspider is True:

instances[cls.name] = cls()

else:

instances.append(cls()) # 实例化类对象

return instances # 返回类对象

## 3. 修改main.py

这样main.py就不用再导入并传入那么多对象了：

# project\_dir/main.py

from scrapy\_plus.core.engine import Engine # 导入引擎

if \_\_name\_\_ == '\_\_main\_\_':

engine = Engine() # 创建引擎对象

engine.start() # 启动引擎

## 小结

* 本小结重点
  + 掌握importlib.import\_module方法使用
  + 完成代码的重构，实现通过配置文件导入模块

# 3.3.9. 框架完善--实现请求去重

去重原理

## 目标

* 掌握去重的方法
* 完成代码的重构，实现去重

## 1. 去重的理解

其实就只是对以往数据进行一个比对，判断是否已经存在

可大致分为：对原始数据比对、对利用原始数据生成的特征值进行比对两种方式

原始数据比对很好理解，就是比对的时候参照值就是原始数据；而利用特征值比对，比如最典型的就是利用原始数据生成一个指纹，比对的参照值就是这个指纹，不是原始数据本身，主要应用于单个原始数据比较大的情况，另外一种常用就是布隆过滤器，这种方式原始利用一种"特征值"，应用场景是海量数据的去重(但具有一定几率的误判)。

## 2. 爬虫请求去重原理和实现

根据请求的url、请求方法、请求参数、请求体进行唯一标识，进行比对，由于这四个数据加到一起，内容较长，因此使用求指纹的方式来进行去重判断。

指纹计算方法，最常用的就是md5、sha1等hash加密算法，来求指纹

具体实现如下：

# scrapy\_plus/core/scheduler.py

# 利用six模块实现py2和py3兼容

from hashlib import sha1

import w3lib.url

from six.moves.queue import Queue

def utf8\_string(string):

'''如果字符串是普通文本，如ascii或者unicode编码类型等，就转换为utf-8; 否则原样返回'''

import six

if six.PY2: # 如果是python2环境

if isinstance(string, str):

return string

else:

return string.encode("utf-8") # 转换为python2中的str类型

elif six.PY3: # 如果是python3环境

if isinstance(string, str):

return string.encode("utf-8") # 转换为bytes类型

else:

return string # 说明string是bytes类型，直接返回

class Scheduler(object):

'''

1. 缓存请求对象(Request)，并为下载器提供请求对象，实现请求的调度

2. 对请求对象进行去重判断

'''

def \_\_init\_\_(self):

self.queue = Queue()

# 记录总共的请求数

self.total\_request\_number = 0

# 利用set类型存储每个请求的指纹

self.\_filter\_container= set()

self.repeat\_request\_number = 0 # 记录重复的请求数

def add\_request(self, request):

'''添加请求对象'''

# 添加请求对象前，先进性去重判断

fp = self.\_gen\_fp(request) # 生成指纹

if not self.filter\_request(fp, request): # 如果指纹不存在，那么添加该请求

self.queue.put(request)

self.\_filter\_container.add(fp) # 添加完请求后，将指纹也记录下来

self.total\_request\_number += 1 # 统计请求总数

else:

self.repeat\_request\_number += 1

def get\_request(self):

'''获取一个请求对象并返回'''

try:

request = self.queue.get(False) # 改为非阻塞模式

except:

return None # 如果为空，就返回None

else:

return request

def filter\_request(self, fp, request):

'''对请求对象进行去重判断'''

# 暂时不做判断，默认都是不重复的

# 去重容器：存储已经发过的请求的特征 url 选用集合类型：set()

# 利用请求的url method params data 求出一个指纹 利用sha1

if fp not in self.\_filter\_container:

logger.info("发现重复的请求：%s"%(request.url))

return True

else:

return False

def \_gen\_fp(self, request):

'''请求去重，计算指纹'''

# 用来判断请求是否重复的属性：url，method，params，data

# 为保持唯一性，需要对他们按照同样的排序规则进行排序

# 1. url排序：借助w3lib.url模块中的canonicalize\_url方法

url = w3lib.url.canonicalize\_url(request.url)

# 2. method不需要排序，只要保持大小写一致就可以

method = request.method.upper() # 全大写

# 3. params排序：如果有提供则是一个字典，如果没有则是None

params = request.params if request.params is not None else {} # 如果是None，那么设为{}

params = sorted(params.items(), key=lambda x:x[0]) # 按照key对字典进行排序，结果将是一个列表

# 4. data排序：如果有提供则是一个字典，如果没有则是None

data = request.data if request.data is not None else {} # 如果是None，那么设为{}

data = sorted(data.items(), key=lambda x: x[0]) # 按照key对字典进行排序，结果将是一个列表

# 5. 利用sha1算法，计算指纹

s1 = sha1()

# 由于s1接收的只能是bytes类型(python3)、str类型(python2)，，他们都是某种编码类型的(如utf-8、gbk)

# 为了兼容py2和py3，利用utf8\_string方法，判断并生成一个bytes类型(python3)或str类型(python2)

s1.update(utf8\_string(url))

s1.update(utf8\_string(method))

s1.update(utf8\_string(str(params)))

s1.update(utf8\_string(str(data)))

fp = s1.hexdigest() # 提取sha1指纹字符串并返回

return fp

## 小结

* 本小结重点
  + 掌握sha1等加密方法
  + 理解去重过程中警醒url排序，字符串的处理，字典的排序的原因
  + 实现整个代码的重构

# 3.3.10. 框架完善--使用线程池实现异步以及并发控制

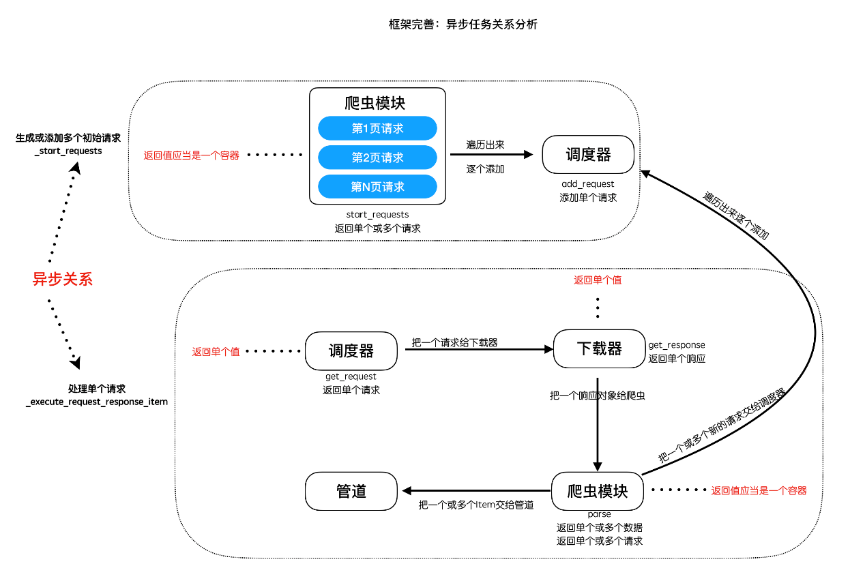
利用线程池实现异步

## 目标

* 掌握线程池的使用
* 使用多线程重构代码

## 1. 异步任务分析：

### 1.1. 在引擎中，实现的主要功能如下图，

* 上面的方框中是关于start\_urls中的请求处理
* 下面的方框中是一个请求从调度器取出请求，进行下载之后交给爬虫解析再交给管道的过程 在以上两个过程中，他们之间没有直接的联系，都可以通过异步多线程的方式分别实现，加快程序执行的速度 

### 1.2 那么具体该如何实现该逻辑

* multiprocessing.dummy 提供的Pool 类具有apply\_async的方法，能够异步的执行让他运行的函数
* apply\_async方法能够接收一个callback，即其中的函数执行完成之后继续会做的事情，在这里，我们可以定义一个callback，其中让他继续执行上图中下方框的任务，同时给他一个停止条件，

## 2. 利用回调实现循环

利用回调实现递归，可以达到循环的目的

# scrapy\_plus/core/engine.py

import time

from multiprocessing.dummy import Pool # 导入线程池对象

import importlib

from datetime import datetime

from scrapy\_plus.http.request import Request # 导入Request对象

from scrapy\_plus.utils.log import logger # 导入logger

from scrapy\_plus.conf import settings

from .scheduler import Scheduler

from .downloader import Downloader

class Engine(object):

'''

a. 对外提供整个的程序的入口

b. 依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)

'''

def \_\_init\_\_(self):

self.scheduler = Scheduler() # 初始化调度器对象

self.downloader = Downloader() # 初始化下载器对象

self.spiders = self.\_auto\_import\_instances(settings.SPIDERS, True) # 动态导入并实例化爬虫对象

self.pipelines = self.\_auto\_import\_instances(settings.PIPELINES) # 动态导入并实例化管道对象

self.spider\_mids = self.\_auto\_import\_instances(settings.SPIDER\_MIDDLEWARES) # 动态导入并实例化爬虫中间件对象

self.downloader\_mids = self.\_auto\_import\_instances(settings.DOWNLOADER\_MIDDLEWARES) # 动态导入并实例化下载器中间件对象

self.total\_response\_number = 0

self.pool = Pool() # 创建线程池对象

self.running = False # 记录是否退出程序的状态

def start(self):

'''启动整个引擎'''

start = datetime.now() # 起始时间

logger.info("开始运行时间：%s" % start) # 使用日志记录起始运行时间

self.\_start\_engine()

stop = datetime.now() # 结束时间

logger.info("开始运行时间：%s" % stop) # 使用日志记录结束运行时间

logger.info("耗时：%.2f" % (stop - start).total\_seconds()) # 使用日志记录运行耗时

def \_start\_requests(self):

'''向调度器添加初始请求'''

# 1. 爬虫模块发出初始请求

for spider\_name, spider in self.spiders.items():

for start\_request in spider.start\_requests():

# 2. 把初始请求添加给调度器

# 利用爬虫中间件预处理请求对象

for spider\_mid in self.spider\_mids:

start\_request = spider\_mid.process\_request(start\_request)

start\_request.spider\_name = spider\_name # 为请求对象绑定它所属的爬虫的名称

self.scheduler.add\_request(start\_request)

def \_execute\_request\_response\_item(self):

'''根据请求、发起请求获取响应、解析响应、处理响应结果'''

# 3. 从调度器获取请求对象，交给下载器发起请求，获取一个响应对象

request = self.scheduler.get\_request()

if request is None:

return

# 利用下载器中间件预处理请求对象

for downloader\_mid in self.downloader\_mids:

request = downloader\_mid.process\_request(request)

# 4. 利用下载器发起请求

response = self.downloader.get\_response(request)

# 利用下载器中间件预处理响应对象

for downloader\_mid in self.downloader\_mids:

response = downloader\_mid.process\_response(response)

spider = self.spiders[request.spider\_name] # 根据请求的spider\_name属性，获取对应的爬虫对象

# 5. 利用爬虫的解析响应的方法，处理响应，得到结果

parse = getattr(spider, request.parse) # 获取对应的解析函数

results = parse(response) # parse函数的返回值是一个容器，如列表或者生成器对象

for result in results:

# 6. 判断结果对象

# 6.1 如果是请求对象，那么就再交给调度器

if isinstance(result, Request):

# 利用爬虫中间件预处理请求对象

for spider\_mid in self.spider\_mids:

result = spider\_mid.process\_request(result)

result.spider\_name = request.spider\_name # 为请求对象绑定它所属的爬虫的名称

self.scheduler.add\_request(result)

# 6.2 否则，就交给管道处理

else:

# 利用爬虫中间件预处理数据对象

for spider\_mid in self.spider\_mids:

result = spider\_mid.process\_item(result)

for pipeline in self.pipelines: # 多个管道对象，轮流处理item对象

result = pipeline.process\_item(result, spider)

# 统计响应总数

self.total\_response\_number += 1

def \_callback(self, temp):

'''执行新的请求的回调函数，实现循环'''

if self.running is True: # 如果还没满足退出条件，那么继续添加新任务，否则不继续添加，终止回调函数，达到退出循环的目的

self.pool.apply\_async(self.\_execute\_request\_response\_item, callback=self.\_callback)

def \_start\_engine(self):

'''依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)'''

self.running = True # 启动引擎，设置状态为True

# 向调度器添加初始请求

self.pool.apply\_async(self.\_start\_requests) # 使用异步

self.pool.apply\_async(self.\_execute\_request\_response\_item, callback=self.\_callback) # 利用回调实现循环

# 设置循环，处理多个请求

while True:

time.sleep(0.0001) # 避免cpu空转，消耗性能

# 根据请求、发起请求获取响应、解析响应、处理响应结果

# self.\_execute\_request\_response\_item()

# 设置退出条件：当请求数和响应数相等时，退出循环

# 因为异步，需要增加判断，请求数不能为0

if self.total\_response\_number >= self.scheduler.total\_request\_number and self.scheduler.total\_request\_number != 0:

self.running = False # 满足循环退出条件后，设置运行状态为False

break

self.pool.close()

self.pool.join()

## 3. 实现异步并发控制

在配置文件中设置最大并发数，并在引擎中使用

# scrapy\_plus/core/engine.py

class Engine(object):

......

def \_start\_engine(self):

self.running = True

'''依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)'''

# 向调度器添加初始请求

self.pool.apply\_async(self.\_start\_requests) # 使用异步

# 控制最大并发数

for i in range(settings.MAX\_ASYNC\_NUMBER):

self.pool.apply\_async(self.\_execute\_request\_response\_item, callback=self.\_callback) # 利用回调实现循环

while True:

time.sleep(0.0001) # 避免cpu空转，消耗性能

# 设置退出条件：当请求数和响应数相等时，退出循环

# 因为异步，需要增加判断，请求数不能为0

if self.total\_response\_number >= self.scheduler.total\_request\_number and self.scheduler.total\_request\_number != 0:

self.running = False # 满足循环退出条件后，设置运行状态为False

break

self.pool.close()

self.pool.join()

## 4. 对异步任务进行异常控制，增加异常回调函数error\_callback

# scrapy\_plus/core/engine.py

class Engine(object):

......

def \_callback(self, temp):

'''执行新的请求的回调函数，实现循环'''

if self.running is True: # 如果还没满足退出条件，那么继续添加新任务，否则不继续添加，终止回调函数，达到退出循环的目的

self.pool.apply\_async(self.\_execute\_request\_response\_item, callback=self.\_callback, error\_callback=self.\_error\_callback)

def \_error\_callback(self, exception):

'''异常回调函数'''

try:

raise exception # 抛出异常后，才能被日志进行完整记录下来

except Exception as e:

logger.exception(e)

def \_start\_engine(self):

self.running = True

'''依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)'''

# 向调度器添加初始请求

self.pool.apply\_async(self.\_start\_requests, error\_callback=self.\_error\_callback) # 使用异步

for i in range(settings.MAX\_ASYNC\_NUMBER):

self.pool.apply\_async(self.\_execute\_request\_response\_item, callback=self.\_callback, error\_callback=self.\_error\_callback) # 利用回调实现循环

while True:

time.sleep(0.0001) # 避免cpu空转，消耗性能

# 设置退出条件：当请求数和响应数相等时，退出循环

# 因为异步，需要增加判断，请求数不能为0

if self.total\_response\_number >= self.scheduler.total\_request\_number and self.scheduler.total\_request\_number != 0:

self.running = False # 满足循环退出条件后，设置运行状态为False

break

self.pool.close()

self.pool.join()

## 小结

* 本小结重点
  + 掌握线程池模块及其方法的使用
  + 完成代码的重构，提高框架的整体效率

# 3.3.11. 框架完善--使用协程池实现异步以及并发控制

使用gevent的Pool实现异步并发

## 目标

* 掌握gevent中Pool基本使用
* 实现代码的重构，使用gevent来进一步提高效率

## 1. 为什么使用gevent

对于I/O密集型任务，gevent能对性能做很大提升的，协程的创建、调度开销都比线程小的多。

## 2. 通过配置文件设置属性，来判断所使用的异步方式

# 异步方式 thread、coroutine

ASYNC\_TYPE = 'coroutine'

## 3. 让gevent的Pool和线程池Pool的接口一致

因此需要单独对gevent的Pool进行一下修改，具体如下： 在scrapy\_plus下创建async包，随后创建coroutine.py模块

# scrapy\_plus/async/coroutine.py

'''

由于gevent的Pool的没有close方法，也没有异常回调参数

引出需要对gevent的Pool进行一些处理，实现与线程池一样接口，实现线程和协程的无缝转换

'''

from gevent.pool import Pool as BasePool

import gevent.monkey

gevent.monkey.patch\_all() # 打补丁，替换内置的模块

class Pool(BasePool):

'''协程池

使得具有close方法

使得apply\_async方法具有和线程池一样的接口

'''

def apply\_async(self, func, args=None, kwds=None, callback=None, error\_callback=None):

return super().apply\_async(func, args=args, kwds=kwds, callback=callback)

def close(self):

'''什么都不需要执行'''

pass

## 4. 在引擎中使用上：

# scrapy\_plus/core/engine.py

import time

import importlib

from datetime import datetime

from scrapy\_plus.http.request import Request # 导入Request对象

from scrapy\_plus.utils.log import logger # 导入logger

from scrapy\_plus.conf import settings

# 判断使用什么异步模式，改用对应的异步池

if settings.ASYNC\_TYPE == 'thread':

from multiprocessing.dummy import Pool # 导入线程池对象

elif settings.ASYNC\_TYPE == 'coroutine':

from scrapy\_plus.async.coroutine import Pool

else:

raise Exception("不支持的异步类型：%s, 只能是'thread'或者'coroutine'"%settings.ASYNC\_TYPE)

# 注意：

# 由于打patch补丁是为了替换掉socket为非阻塞的

# 而下载器中正好使用了requests模块，如果在这之后导入协程池，会导致requests中使用的socket没有被替换成功

# 从而有可能导致使用出现问题

from .scheduler import Scheduler

from .downloader import Downloader

class Engine(object):

......

## 小结

* 本小结重点
  + 掌握gevent.monkey\_path的使用，注意其中的导入顺序
  + 了解gevent.pool的使用
  + 完成gevent版本的异步实现

# 3.4. 框架功能升级

## 介绍

前面的内容，我们实现了一个功能相对完善的爬虫框架，但是还不够强大，对于一些功能比如分布式，断点续爬等功能任然没有实现，那么接下来在框架功能升级这一部分，我们继续来完善框架

## 内容

* 分布式爬虫的支持
* 增量爬虫的设计支持
* 断点续爬的设计支持

# 3.4.1. 框架升级--分布式爬虫设计原理及其实现

框架升级 -- 分布式爬虫设计原理及其实现

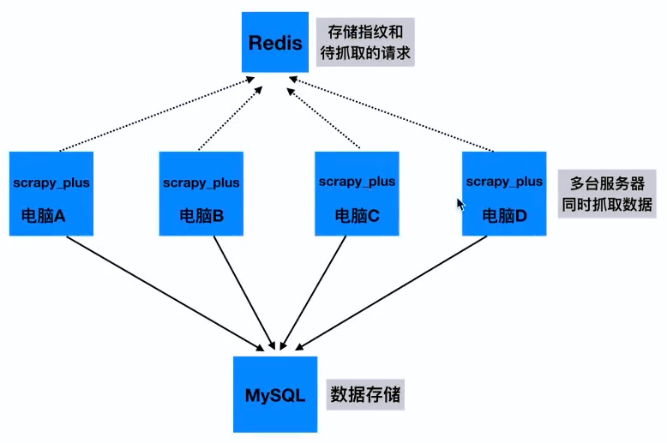
## 目标

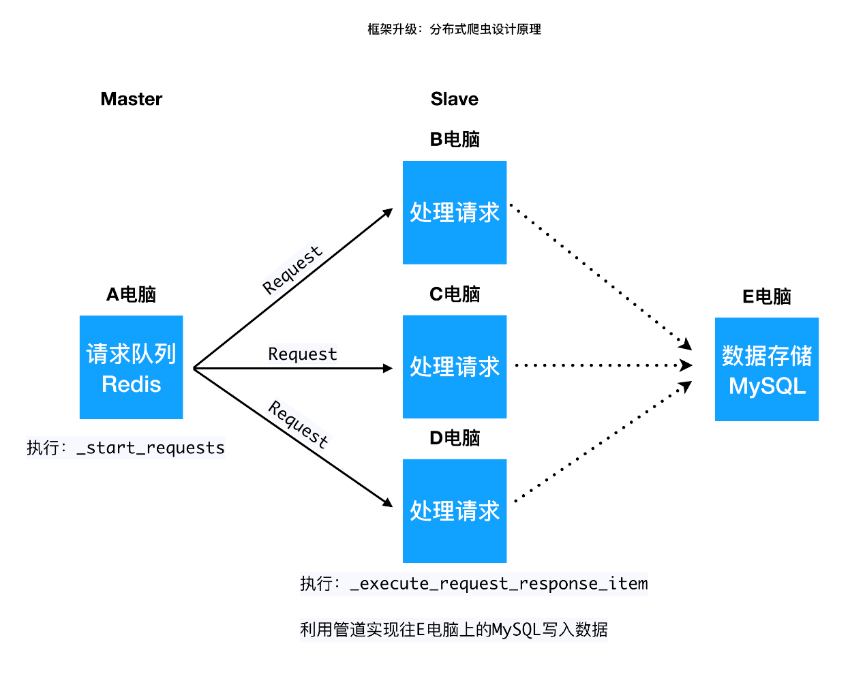
* 理解分布式爬虫的原理
* 理解如何通过redis构建一个队列
* 完成代理的重构，实现分布式

## 1. 分布式爬虫原理

分布式爬虫设计原理：

多台服务器同时抓取数据, 请求和指纹存储在同一个redis中





## 2. 实现方案

利用redis实现队列

1. 注意pickle模块的使用：如果将对象存入redis中，需要先将其序列化为二进制数据，取出后反序列化就可以再得到原始对象
2. 接口定义一致性：利用redis使用一个Queue，使其接口同python的内置队列接口一致，可以实现无缝转换

# scrapy\_plus/queue.py

import time

import pickle

import redis

from six.moves import queue as BaseQueue

# redis队列默认配置

REDIS\_QUEUE\_NAME = 'request\_queue'

REDIS\_QUEUE\_HOST = 'localhost'

REDIS\_QUEUE\_PORT = 6379

REDIS\_QUEUE\_DB = 10

# 利用redis实现一个Queue，使其接口同python的内置队列接口一致，可以实现无缝转换

class Queue(object):

"""

A Queue like message built over redis

"""

Empty = BaseQueue.Empty

Full = BaseQueue.Full

max\_timeout = 0.3

def \_\_init\_\_(self, maxsize=0, name=REDIS\_QUEUE\_NAME, host=REDIS\_QUEUE\_HOST, port=REDIS\_QUEUE\_PORT, db=REDIS\_QUEUE\_DB,

lazy\_limit=True, password=None):

"""

Constructor for RedisQueue

maxsize: an integer that sets the upperbound limit on the number of

items that can be placed in the queue.

lazy\_limit: redis queue is shared via instance, a lazy size limit is used

for better performance.

"""

self.name = name

self.redis = redis.StrictRedis(host=host, port=port, db=db, password=password)

self.maxsize = maxsize

self.lazy\_limit = lazy\_limit

self.last\_qsize = 0

def qsize(self):

self.last\_qsize = self.redis.llen(self.name)

return self.last\_qsize

def empty(self):

if self.qsize() == 0:

return True

else:

return False

def full(self):

if self.maxsize and self.qsize() >= self.maxsize:

return True

else:

return False

def put\_nowait(self, obj):

if self.lazy\_limit and self.last\_qsize < self.maxsize:

pass

elif self.full():

raise self.Full

self.last\_qsize = self.redis.rpush(self.name, pickle.dumps(obj))

return True

def put(self, obj, block=True, timeout=None):

if not block:

return self.put\_nowait(obj)

start\_time = time.time()

while True:

try:

return self.put\_nowait(obj)

except self.Full:

if timeout:

lasted = time.time() - start\_time

if timeout > lasted:

time.sleep(min(self.max\_timeout, timeout - lasted))

else:

raise

else:

time.sleep(self.max\_timeout)

def get\_nowait(self):

ret = self.redis.lpop(self.name)

if ret is None:

raise self.Empty

return pickle.loads(ret)

def get(self, block=True, timeout=None):

if not block:

return self.get\_nowait()

start\_time = time.time()

while True:

try:

return self.get\_nowait()

except self.Empty:

if timeout:

lasted = time.time() - start\_time

if timeout > lasted:

time.sleep(min(self.max\_timeout, timeout - lasted))

else:

raise

else:

time.sleep(self.max\_timeout)

## 2.通过配置文件选择是否启用分布式：

# 项目文件夹/settings.py

......

# 设置程序运行的角色

# 三个值：None、'master'、'slave'

# 如果是None，那么就是不使用分布式，既不是master，也不是slave

# 如果是'master', 代表主端，也就是只负责请求队列的维护

# 如果是'slave'，代表从端，也就是只负责从请求队列获取请求，并进行处理

ROLE = 'master'

## 3.在引擎中进行判断使用：

# scrapy\_plus/core/engine.py

......

def \_start\_engine(self):

self.running = True

'''依次调用其他组件对外提供的接口，实现整个框架的运作(驱动)'''

if settings.ROLE is None or settings.ROLE == 'master': # 如果是None代表非分布式; 如果是'master'就只负责执行\_start\_requests

# 向调度器添加初始请求

self.pool.apply\_async(self.\_start\_requests, error\_callback=self.\_error\_callback) # 使用异步

if settings.ROLE is None or settings.ROLE == 'slave': # 如果是None代表非分布式; 如果是'slave'就只负责执行\_execute\_request\_response\_item:

for i in range(settings.MAX\_ASYNC\_NUMBER):

self.pool.apply\_async(self.\_execute\_request\_response\_item, callback=self.\_callback, error\_callback=self.\_error\_callback) # 利用回调实现循环

......

## 4.在调度器中进行判断和使用：

# scrapy\_plus/core/scheduler.py

from hashlib import sha1

import w3lib.url

from six.moves.queue import Queue

from scrapy\_plus.conf import settings

from scrapy\_plus.queue import Queue as ReidsQueue

......

class Scheduler(object):

'''

1. 缓存请求对象(Request)，并为下载器提供请求对象，实现请求的调度

2. 对请求对象进行去重判断

'''

def \_\_init\_\_(self):

# 判断是不是使用分布式：

if settings.ROLE is None: # 如果不是，那么使用的python内置队列

self.queue = Queue()

elif settings.ROLE == 'master' or settings.ROLE == 'slave': # 如果是分布式，那么使用redis实现的队列

self.queue = ReidsQueue()

# 记录总共的请求数

self.total\_request\_number = 0

# 利用set类型存储每个请求的指纹

self.\_filter\_container= set()

self.repeat\_request\_number = 0 # 记录重复的请求数

......

这个时候基本实现了分布式，但是还存在一个问题，也就是，请求去重问题(这个必须处理)，还有就是自动退出判断问题(分布式的话通常不需要自动退出)

如果分布式中请求去重的去重容器各个从端以及主端用的不是同一个，那么就无法达到去重的目的，因此这里同样的需要使用redis来实现去重容器，也就是把所有的去重指纹都存储在redis中，所有的主从端都是通过同一台redis来进行判断请求的重复与否

## 5. 利用Redis的集合类型实现去重

实现一个自定义的set：

# scrapy\_plus/set.py

import redis

from scrapy\_plus.conf import settings

class BaseFilterContainer(object):

def add\_fp(self, fp):

'''往去重容器添加一个指纹'''

pass

def exists(self, fp):

'''判断指纹是否在去重容器中'''

pass

class NoramlFilterContainer(BaseFilterContainer):

'''利用python的集合类型'''

def \_\_init\_\_(self):

self.\_filter\_container = set()

def add\_fp(self, fp):

''''''

self.\_filter\_container.add(fp)

def exists(self, fp):

'''判断指纹是否在去重容器中'''

if fp in self.\_filter\_container:

return True

else:

return False

class RedisFilterContainer(BaseFilterContainer):

REDIS\_SET\_NAME = settings.REDIS\_SET\_NAME

REDIS\_SET\_HOST = settings.REDIS\_SET\_HOST

REDIS\_SET\_PORT = settings.REDIS\_SET\_PORT

REDIS\_SET\_DB = settings.REDIS\_SET\_DB

def \_\_init\_\_(self):

self.\_redis = redis.StrictRedis(host=self.REDIS\_SET\_HOST, port=self.REDIS\_SET\_PORT ,db=self.REDIS\_SET\_DB)

self.\_name = self.REDIS\_SET\_NAME

def add\_fp(self, fp):

'''往去重容器添加一个指纹'''

self.\_redis.sadd(self.\_name, fp)

def exists(self, fp):

'''判断指纹是否在去重容器中'''

return self.\_redis.sismember(self.\_name, fp)

在调度器中使用这个set.py， 使得分布式模式下的去重功能正常运作

# scrapy\_plus/core/scheduler.py

from hashlib import sha1

import w3lib.url

from six.moves.queue import Queue

from scrapy\_plus.conf import settings

from scrapy\_plus.queue import Queue as RedisQueue

from scrapy\_plus.set import NoramlFilterContainer, RedisFilterContainer

from scrapy\_plus.utils.log import logger

......

class Scheduler(object):

'''

1. 缓存请求对象(Request)，并为下载器提供请求对象，实现请求的调度

2. 对请求对象进行去重判断

'''

def \_\_init\_\_(self):

# 判断是否使用分布式：

if settings.ROLE is None: # 如果不使用，

self.queue = Queue() # 那么使用python的内置队列

self.\_filter\_container = NoramlFilterContainer() # 那么使用python的内置集合类型实现的指纹存储容器

elif settings.ROLE == 'master' or settings.ROLE == 'slave': # 如果使用分布式，

self.queue = RedisQueue() # 那么使用redis实现的队列

self.\_filter\_container = RedisFilterContainer() # 那么使用redis的集合类型实现的指纹存储容器

# 记录总共的请求数

self.total\_request\_number = 0

self.repeat\_request\_number = 0 # 记录重复的请求数

def add\_request(self, request):

'''添加请求对象'''

# 添加请求对象前，先进行去重判断

fp = self.\_gen\_fp(request)

if not self.filter\_request(fp, request):: # 如果指纹不存在，那么添加该请求

self.queue.put(request)

logger.info("添加请求成功[%s %s]"%(request.method, request.url))

self.\_filter\_container.add\_fp(fp) # 添加完请求后，将指纹也记录下来

self.total\_request\_number += 1 # 统计请求总数

else:

self.repeat\_request\_number += 1

5. 程序结束的条件

在之前的单机版本的代码中, 通过 总的响应 + 总的重复数 >= 总的请求 来判断程序结束, 但是在分布式的版本中, 每个服务器的请求数量和响应数量不再相同.

因为每个服务器存入队列的请求, 和成功发送的请求中间可能很多请求被其他的服务器发送了, 导致数量不一致, 所以可以把总的请求, 总的响应, 总的重复请求等信息记录在 redis 中, 那么所有的服务器端修改的数据的位置是同一个 redis 中的内容, 所有的服务央判断退出的时候也是通过比较同一个 redis 中的这些数据来决定.

因此, 在 utils 中新建 stats\_collector.py 文件, 来实现对各种数量的统计, 包括总的请求数量, 总的响应数量, 总的重复数量.

## 小结

* 本小结重点
  + 分布式的实现原理
  + 使用redis实现队列中，put和put\_nowait以及get和get\_nowait的区别
  + 完成代码的重构，实现分布式

# 3.4.2. 框架升级--增量爬虫设计原理及其实现

框架升级 -- 增量爬虫设计原理及其实现

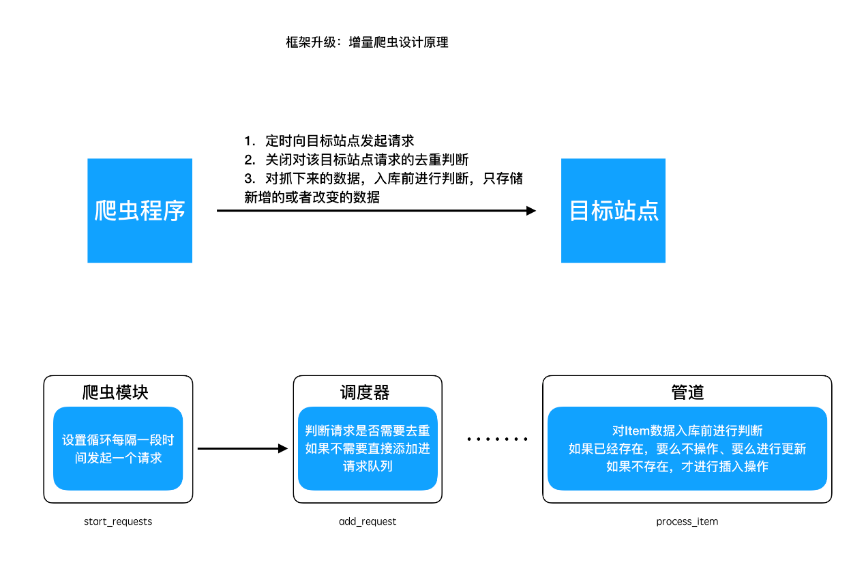
## 目标

* 理解增量式爬虫的原理
* 完成增量式爬虫的实现

## 1. 增量爬虫设计原理

增量抓取，意即针对某个站点的数据抓取，当网站的新增数据或者该站点的数据发生了变化后，自动地抓取它新增的或者变化后的数据

设计原理：



### 1.1 实现关闭请求去重

1. 为Request对象增加属性filter

# scrapy/http/reqeust.py

'''封装Request对象'''

class Request(object):

'''请求对象，设置请求信息'''

def \_\_init\_\_(self, url, method='GET', headers=None, params=None, data=None, filter=True):

self.url = url # 请求地址

self.method = method # 请求方法

self.headers = headers # 请求头

self.params = params # 请求参数

self.data = data # 请求体

self.filter = filter # 是否进行去重，默认是True

1. 修改调度器，进行判断

# scrapy\_plus/core/scheduler.py

class Scheduler(object):

......

def add\_request(self, request):

'''添加请求对象'''

# 先判断是否要去重

if request.filter is False:

self.queue.put(request)

logger.info("添加请求成功<disable去重>[%s %s]" % (request.method, request.url))

self.total\_request\_number += 1 # 统计请求总数

return # 必须return

# 添加请求对象前，先进性去重判断

fp = self.\_gen\_fp(request)

if not self.filter\_request(fp, request): # 如果指纹不存在，那么添加该请求

self.queue.put(request)

logger.info("添加请求成功[%s %s]"%(request.method, request.url))

self.\_filter\_container.add\_fp(fp) # 添加完请求后，将指纹也记录下来

self.total\_request\_number += 1 # 统计请求总数

else:

logger.info("发现重复的请求 [%s %s]" % (request.method, request.url))

self.repeat\_request\_number += 1

......

### 1.2 实现无限发起请求:

新增爬虫抓取：[新浪滚动新闻](http://roll.news.sina.com.cn/s/channel.php#col=89&spec=&type=&ch=&k=&offset_page=0&offset_num=0&num=120&asc=&page=1)

1. 在start\_reqeusts中改成无限循环，并设置对应请求为非去重模式。（注意）

# spiders/baidu.py

import time

from scrapy\_plus.core.spider import Spider

from scrapy\_plus.http.request import Request

from scrapy\_plus.item import Item

import js2py

class SinaGunDong(Spider):

name = "sina\_gundong"

def start\_requests(self):

while True:

# 需要发起这个请求，才能获取到列表页数据，并且返回的是一个js语句

url = "http://roll.news.sina.com.cn/interface/rollnews\_ch\_out\_interface.php?col=89&spec=&type=&ch=&k=&offset\_page=0&offset\_num=0&num=120&asc=&page=1&r=0.5559616678192825"

yield Request(url, parse='parse', filter=False)

time.sleep(10) # 每10秒发起一次请求

def parse(self, response):

'''响应体数据是js代码'''

# 使用js2py模块，执行js代码，获取数据

ret = js2py.eval\_js(response.body.decode("gbk")) # 对网站分析发现，数据编码格式是gbk的，因此需要先进行解码

yield Item(ret.list)

但由于框架调用start\_requests方法时同步，如果设置为死循环后，那么位于之后的爬虫的start\_requests方法就不会被调用，因此需要在调用每个爬虫的start\_reqeusts时设置为异步的

# scrapy\_plus/core/engine.py

class Engine(object):

......

def \_start\_requests(self):

'''向调度器添加初始请求'''

# 1. 爬虫模块发出初始请求

# for spider\_name, spider in self.spiders.items():

# for start\_request in spider.start\_requests():

# # 2. 把初始请求添加给调度器

# # 利用爬虫中间件预处理请求对象

# for spider\_mid in self.spider\_mids:

# start\_request = spider\_mid.process\_request(start\_request)

# start\_request.spider\_name = spider\_name #为请求对象绑定它所属的爬虫的名称

# self.scheduler.add\_request(start\_request)

def \_func(spider\_name, spider):

for start\_request in spider.start\_requests():

# 2. 把初始请求添加给调度器

# 利用爬虫中间件预处理请求对象

for spider\_mid in self.spider\_mids:

start\_request = spider\_mid.process\_request(start\_request)

start\_request.spider\_name = spider\_name #为请求对象绑定它所属的爬虫的名称

self.scheduler.add\_request(start\_request)

# 1. 爬虫模块发出初始请求

for spider\_name, spider in self.spiders.items():

self.pool.apply\_async(\_func, args=(spider\_name, spider)) # 把执行每个爬虫的start\_requests方法，设置为异步的

......

让程序的主线程在，多个start\_reqeusts方法都没执行完毕前，不要进行退出判断，避免退出过早：

# scrapy\_plus/core/engine.py

class Engine(object):

'''

负责驱动各大组件，通过调用各自对外提供的API接口，实现它们之间的交互和协作

提供整个框架的启动入口

'''

def \_\_init\_\_(self):

......

self.finshed\_start\_requests\_number = 0

......

def \_callback\_total\_finshed\_start\_requests\_number(self, temp):

'''记录完成的start\_requests的数量'''

self.finshed\_start\_requests\_number += 1

def \_start\_requests(self):

......

# 让主线程在这里阻塞

while True:

time.sleep(0.001) # 节省cpu消耗

# self.pool.apply\_async(self.\_execute\_request\_response\_item) # 发起一个请求，处理一个响应

# 设置退出循环的条件：

# 当处理完的响应数等于总的请求数时，退出循环：

if self.finshed\_start\_requests\_number == len(self.spiders): # 判断是否所有爬虫的start\_requests是否都执行完毕，

# 如果都执行完毕，才应该应该进行退出判断

if self.total\_response\_number == self.scheduler.total\_request\_number and self.total\_response\_number != 0:

self.running = False # 设为Flase， 让子线程满足判断条件，不再执行递归循环，然后退出

break

logger.info("主线程循环已经退出")

self.pool.close() # 意味着无法再向pool添加任务，，无法在调用apply\_async apply

self.pool.join() #

## 小结

* 本小结重点
  + 理解增量式爬虫的内涵
  + 理解增量式爬虫的具体实现方法
  + 完成代码重构达到增量式爬虫的目的

# 3.4.3. 框架升级--断点续爬设计原理及其实现

框架升级 -- 断点续爬设计原理及其实现

## 目标

* 理解断点续爬的内涵
* 理解分布式爬虫中请求丢失的情况
* 理解使用备份队列保留请求的过程
* 完成代码的重构，解决请求丢失的请求

## 1. 断点续爬设计分析

* 断点续爬设计原理介绍：

断点续爬的效果: 爬虫程序中止后, 再次启动, 对已经发起的请求不再发起, 而是直接从之前的队列中获取请求继续执行.

这也就意味着需要实现以下两点:

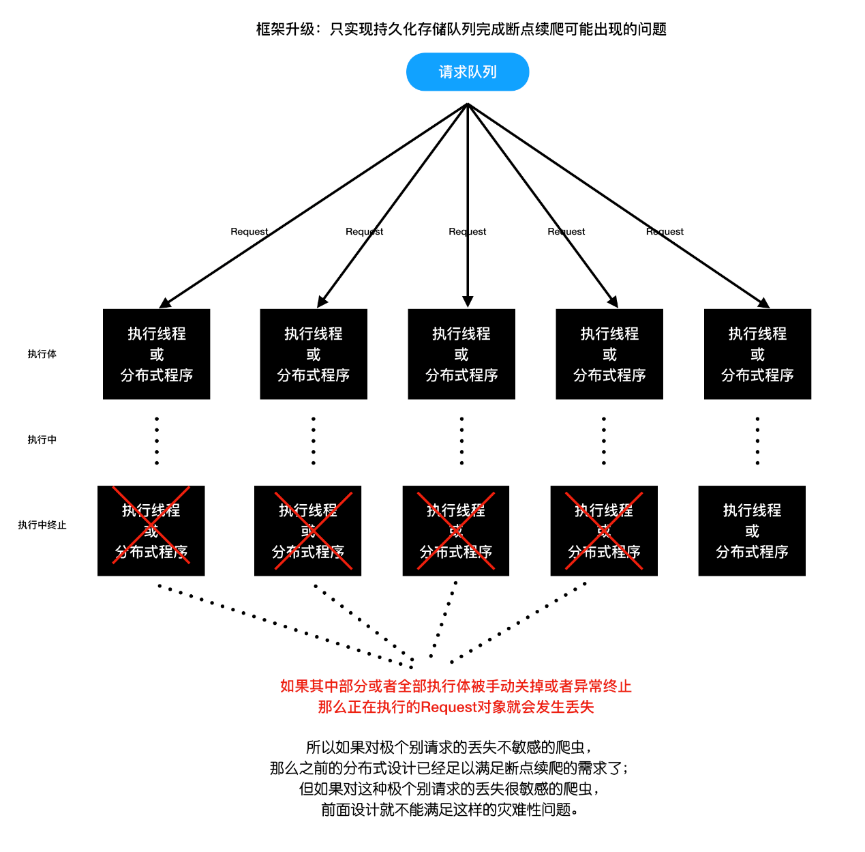
1. 去重标记 (历史请求的指纹) 持久化存储, 使得新的请求可以和以前的请求进行去重比对

2. 请求队列也需要持久化存储

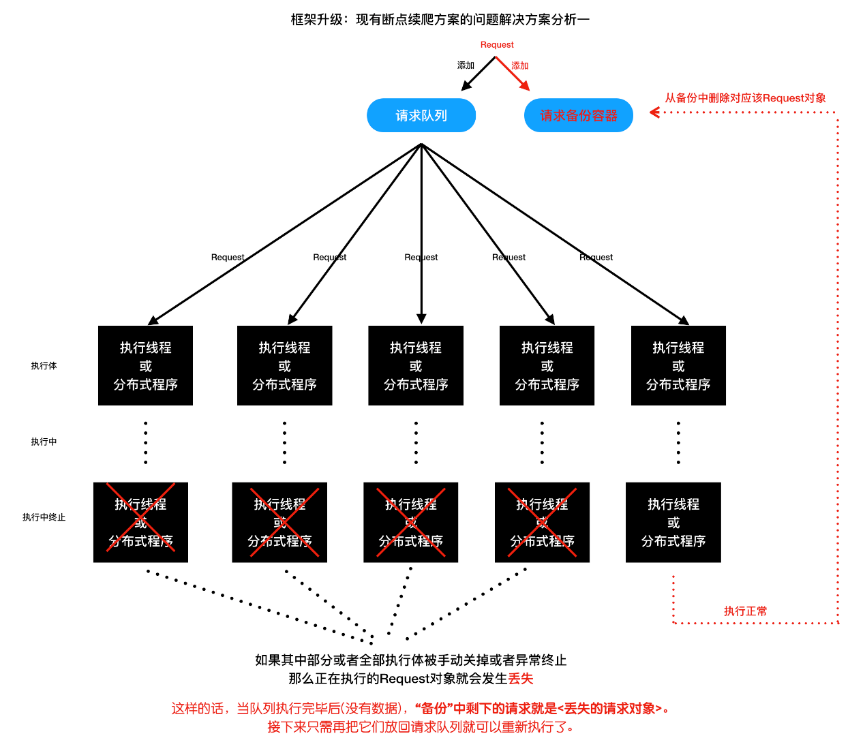
其实也就是程序的中止不会造成请求队列和去重容器的消失, 再次启动程序后, 还能继续访问它们.

前面的分布式设计其实正好就满足了以上两点, 也就是说其实已经实现了断点续爬, 只需要开启分布式.

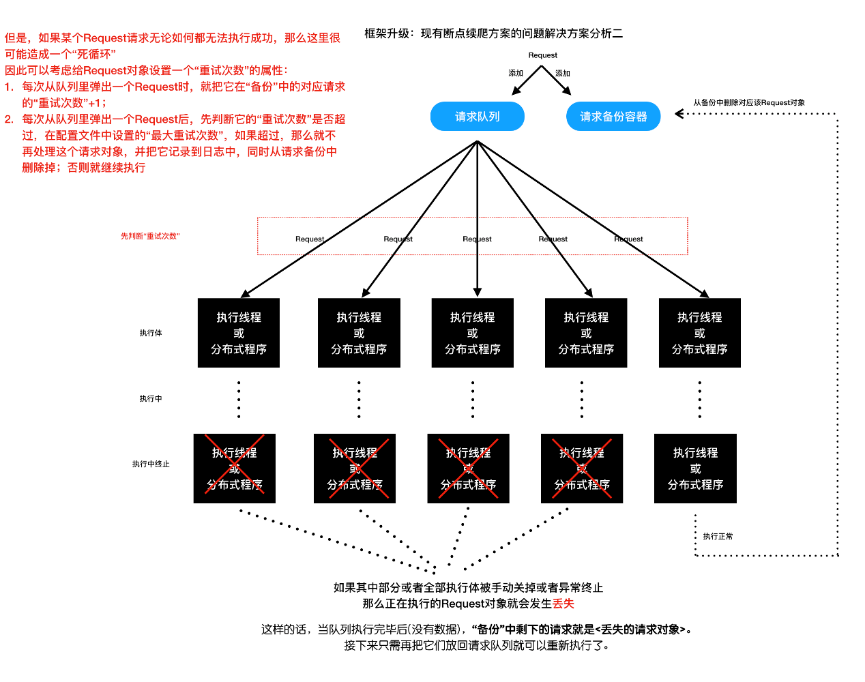
* 只实现持久化存储队列完成断点续爬可能出现的问题:



* 现有断点续爬方案的问题解决方案分析一:

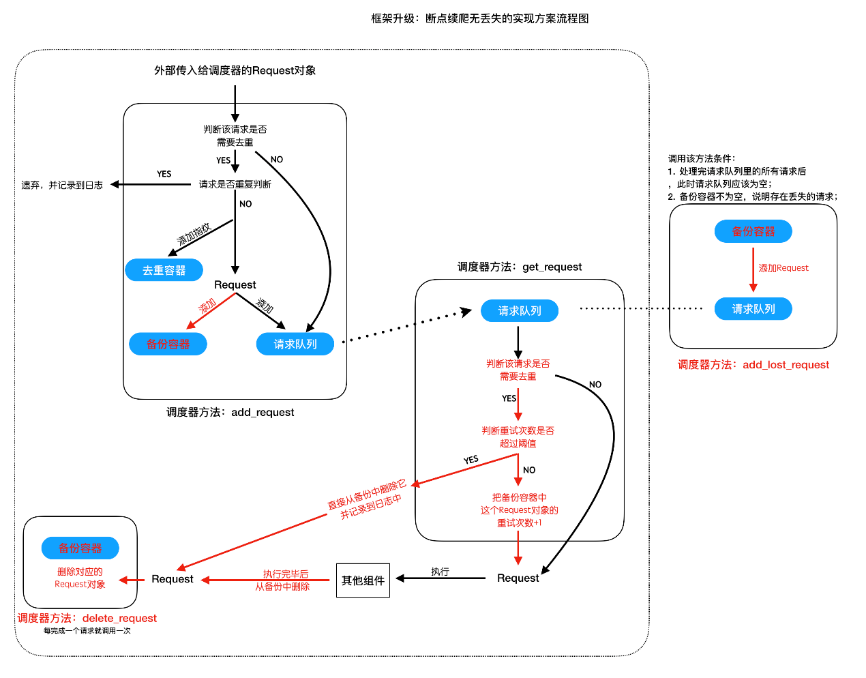


* 现有断点续爬方案的问题解决方案分析二:



## 2. 断点续爬无丢失方案的实现

* 断点续爬无丢失的实现方案分析：



* 断点续爬无丢失的代码实现：
  + 添加备份容器：利用redis的hash类型类对每一个请求对象进行存储
  + 为Request对象设置重试次数属性
  + 在调度器的get\_request方法中实现响应的逻辑判断
  + 实现delete\_request方法：从备份中删除对应的Reqeust对象
  + 实现add\_lost\_request方法
  + 在引擎中调用这些方法，完成断点续爬无丢失需求

# scrapy\_plus/redis\_hash.py

'''实现一个对redis哈希类型的操作封装'''

import redis

import pickle

from scrapy\_plus.http.request import Request

from scrapy\_plus.conf import settings

class RedisBackupRequest(object):

'''利用hash类型，存储每一个请求对象，key是指纹，值就是请求对象'''

REDIS\_BACKUP\_NAME = settings.REDIS\_BACKUP\_NAME

REDIS\_BACKUP\_HOST = settings.REDIS\_BACKUP\_HOST

REDIS\_BACKUP\_PORT = settings.REDIS\_BACKUP\_PORT

REDIS\_BACKUP\_DB = settings.REDIS\_BACKUP\_DB

def \_\_init\_\_(self):

self.\_redis = redis.StrictRedis(host=self.REDIS\_BACKUP\_HOST, port=self.REDIS\_BACKUP\_PORT ,db=self.REDIS\_BACKUP\_DB)

self.\_name = self.REDIS\_BACKUP\_NAME

# 增删改查

def save\_request(self, fp, request):

'''将请求对象备份到redis的hash中'''

bytes\_data = pickle.dumps(request)

self.\_redis.hset(self.\_name, fp, bytes\_data)

def delete\_request(self, fp):

'''根据请求的指纹，将其删除'''

self.\_redis.hdel(self.\_name, fp)

def update\_request(self, fp, request):

'''更新已有的fp'''

self.save\_request(fp, request)

def get\_requests(self):

'''返回全部的请求对象'''

for fp, bytes\_request in self.\_redis.hscan\_iter(self.\_name):

request = pickle.loads(bytes\_request)

yield request

* 为Request对象增加重试次数属性：

class Request(object):

'''框架内置请求对象，设置请求信息'''

def \_\_init\_\_(self, url, method='GET', headers=None, params=None, data=None, parse='parse', filter=True, meta=None):

self.url = url # 请求地址

self.method = method # 请求方法

self.headers = headers # 请求头

self.params = params # 请求参数

self.data = data # 请求体

self.parse = parse # 指明它的解析函数， 默认是parse方法

self.filter = filter # 是否进行去重，默认是True

self.retry\_time = 0 # 重试次数

self.meta = meta

* 修改调度器，实现对应的逻辑以及方法： ```Python

# scrapy\_plus/core/scheduler.py

**from** **scrapy\_plus.redis\_hash** **import** RedisBackupRequest

**class** **Scheduler**(object):

*''' 缓存请求对象(Request)，并为下载器提供请求对象，实现请求的调度 对请求对象进行去重判断 '''*

**def** \_\_init\_\_(self):

**if** settings.ROLE **is** None: *# 如果不是分布式*

self.queue = Queue() *# 那么使用python的内置队列*

self.\_filter\_container = NoramlFilterContainer() *# 去重容器*

**elif** settings.ROLE **in** ['master', 'slave']: *# 判断是否是分布式*

self.\_filter\_container = RedisFilterContainer() *# 使用redis去重容器进行去重*

self.queue = RedisQueue() *# 那么使用redis实现的队列*

self.\_backup\_request = RedisBackupRequest() *# 创建立备份容器*

**else**:

**raise** **Exception**("不支持的运行模式：请检查settings.ROLE")

self.total\_request\_number = 0 *# 统计请求总数*

self.repeat\_request\_number = 0 *# 记录重复的请求数*

**def** add\_reqeust(self, request):

*'''存储request对象进入队列*

*return： None*

*'''*

*# 先判断是否要去重*

**if** request.filter **is** False:

self.queue.put(request)

logger.info("添加请求成功<disable去重>[**%s** **%s**]" % (request.method, request.url))

self.total\_request\_number += 1 *# 统计请求总数*

**return** *# 必须return*

*# 判断去重，如果重复，就不添加，否则才添加*

fp = self.\_gen\_fp(request)

**if** **not** self.filter\_request(fp, request):

*# 往队列添加请求*

logger.info("添加请求成功[**%s** **%s**]"%(request.method.upper(), request.url))

self.queue.put(request)

**if** settings.ROLE **in** ['master', 'slave']:

self.\_backup\_request.save\_request(fp, request) *# 对请求进行备份*

*# 如果是新的请求，那么就添加进去重容器，表示请求已经添加到了队列中*

self.\_filter\_container.add\_fp(fp)

self.total\_request\_number += 1

**else**:

self.repeat\_request\_number += 1

**def** get\_request(self):

*'''从队列取出一个请求对象*

*return： Request Object*

*'''*

**try**:

request = self.queue.get(False)

**except**:

**return** None

**else**:

**if** request.filter **is** True **and** settings.ROLE **in** ['master', 'slave']: *# 先判断 是否需要进行去重*

*# 判断重试次数是否超过规定*

fp = self.\_gen\_fp(request)

**if** request.retry\_time >= settings.MAX\_RETRY\_TIMES:

self.\_backup\_request.delete\_request(fp) *# 如果超过，那么直接删除*

logger.warnning("出现异常请求，且超过最大尝试的次数：[**%s**]**%s**"%(request.method, request.url))

request.retry\_time += 1 *# 重试次数+1*

self.\_backup\_request.update\_request(fp, request) *# 并更新到备份中*

**return** request

**def** delete\_request(self, request):

*'''根据请求从备份删除对应的请求对象'''*

**if** settings.ROLE **in** ['master', 'slave']:

fp = self.\_gen\_fp(request)

self.\_backup\_request.delete\_request(fp)

**def** add\_lost\_reqeusts(self):

*'''将丢失的请求对象再添加到队列中'''*

*# 从备份容器取出来，放到队列中*

**if** settings.ROLE **in** ['master', 'slave']:

**for** request **in** self.\_backup\_request.get\_requests():

self.queue.put(request)

......

```

## 小结

* 本小结重点
  + 理解断点续爬的内涵
  + 理解分布式爬虫中请求丢失的情况
  + 理解使用备份队列保留请求的过程
  + 完成代码的重构，解决请求丢失的请求

## 3.5. 项目实战

新浪滚动新闻资讯实时采集

完善 [新浪滚动新闻](http://roll.news.sina.com.cn/s/channel.php?ch=01#col=89&spec=&type=&ch=01&k=&offset_page=0&offset_num=0&num=60&asc=&page=1)爬虫的数据采集

要求：

1. 存储文章的标题、作者、发布时间、正文、正文中的图片链接、文章链接、文章所属分类
2. 根据网站的实时更新(周期1分钟)进行采集
3. 时间格式保存为"yyyy-mm-dd HH:MM:SS"
4. 存储到mysql数据库

代码实现如下：

新浪滚动的爬虫文件：

# spiders/sina\_gundong.py

import time

from scrapy\_plus.core.spider import Spider

from scrapy\_plus.http.request import Request

from scrapy\_plus.item import Item

import js2py

class SinaGunDong(Spider):

name = "sina\_gundong"

headers = {

"Accept": "\*/\*",

"Accept-Encoding": "gzip, deflate",

"Accept-Language": "en-US,en;q=0.9,zh-CN;q=0.8,zh;q=0.7",

"Cache-Control": "no-cache",

"Connection": "keep-alive",

"Cookie": "UOR=www.google.com,www.sina.com.cn,; SGUID=1520816292777\_83076650; SINAGLOBAL=211.103.136.242\_1520816292.736990; SUB=\_2AkMt-V\_2f8NxqwJRmPEQy2vmZYx\_zwjEieKbpa4tJRMyHRl-yD83qnIJtRB6BnlxGSLw2fy6O04cZUKTsCZUeiiFEsZE; SUBP=0033WrSXqPxfM72-Ws9jqgMF55529P9D9WhpFUZmqbYYLueonGrZIL2c; U\_TRS1=0000001a.e268c0.5aaa0d39.35b0731a; lxlrttp=1521688012; Apache=223.72.62.219\_1522208561.132697; ULV=1522208952476:6:6:3:223.72.62.219\_1522208561.132697:1522208561158; U\_TRS2=000000db.81c2323e.5abca69b.ad269c11; ArtiFSize=14; rotatecount=1; hqEtagMode=1",

# "Host": "roll.news.sina.com.cn", 这里host必须禁用掉

"Pragma": "no-cache",

"Referer": "http://roll.news.sina.com.cn/s/channel.php?ch=01",

"User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36",

}

def start\_requests(self):

while True:

# 需要发起这个请求，才能获取到列表页数据，并且返回的是一个js语句

url = "http://roll.news.sina.com.cn/interface/rollnews\_ch\_out\_interface.php?col=89&spec=&type=&ch=&k=&offset\_page=0&offset\_num=0&num=120&asc=&page=1&r=0.5559616678192825"

yield Request(url, parse='parse', filter=False)

time.sleep(60) # 每60秒发起一次请求

def parse(self, response):

'''响应体数据是js代码'''

# 使用js2py模块，执行js代码，获取数据

ret = js2py.eval\_js(response.body.decode("gbk")) # 对网站分析发现，数据编码格式是gbk的，因此需要先进行解码

for news in ret.list: #

yield Request(news["url"], headers=self.headers, parse='parse\_detail', meta={"type": news["channel"]["title"]})

def parse\_detail(self, response):

response.body = response.body.decode("utf-8") # 部分页面无法正确解码，因此在这里手动进行解码操作

title = response.xpath("//h1[@class='main-title']/text()")[0]

pub\_date = response.xpath("//span[@class='date']/text()")[0]

try:

author = response.xpath("//div[@class='date-source']//a/text()")[0] # 由于作者的提取，有两种格式，因此这里使用一个异常捕获来进行判断

except IndexError:

author = response.xpath("//div[@class='date-source']//span[contains(@class,'source')]/text()")[0]

content = response.xpath("//div[@class='article']//text()") # 多个 每一个代表一段

image\_links = response.xpath("//div[@class='article']//img/@src") # 图片链接有多个

yield Item({

"content": content, # 正文

"image\_links":image\_links, # 图片链接

"title": title, # 标题

"pub\_date":pub\_date, # 发布日期

"author": author, # 作者

"url": response.url, # 文章链接

"type": response.request.meta["type"], # 文章所属分类

}

)

项目中新建db.py

# 项目文件夹下db.py

# 依赖：sqlalchemy pymysql

from sqlalchemy import Column,Integer,Text,DateTime, String

from sqlalchemy.ext.declarative import declarative\_base

# 创建对象的基类:

Base = declarative\_base()

class Model(Base):

\_\_tablename\_\_ = 'sina\_news'

id = Column(Integer, primary\_key=True, autoincrement=True) # 主键id

title = Column(String(100), nullable=False) # 标题

author = Column(String(20), nullable=False) # 作者

pub\_date = Column(DateTime, nullable=False) # 发布时间

content = Column(Text, nullable=False) # 正文

image\_links = Column(Text, nullable=False) # 图片链接

url = Column(String(500), nullable=False) # 文章链接

type = Column(String(6), nullable=False) # 文章分类

news\_tag = Column(String(40), nullable=False) # 文章去重标记

管道文件：

# 项目下管道文件 pipelines.py

import json

from datetime import datetime

from hashlib import sha1

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

from scrapy\_plus.core.scheduler import utf8\_string

from db import Base, Model

class Pipeline(object):

'''数据入库前的清洗和格式化处理'''

def process\_item(self, item, spider):

item.data['pub\_date'] = datetime.strptime(item.data['pub\_date'], '%Y年%m月%d日 %H:%M') # 将时间格式进行一个处理，转换为datetime类型

item.data['content'] = [i for i in item.data['content'] if i.strip()] # 去掉content中的空白字符

item.data['content'] = "\n\n".join(item.data["content"])

item.data['image\_links'] = json.dumps(item.data['image\_links']) # 列表转换为json字符串

# 数据去重标识生成：利用标题、作者、文章链接生成唯一key

s1 = sha1()

s1.update(utf8\_string(item.data['title']))

s1.update(utf8\_string(item.data['author']))

s1.update(utf8\_string(item.data['url']))

item.data['news\_tag'] = s1.hexdigest() # 数据去重标识

return item

class MysqlPipeline(object):

def \_\_init\_\_(self):

# 建立数据库链接

self.conn = create\_engine("mysql+pymysql://root:ryoma@127.0.0.1/test3?charset=utf8")

Base.metadata.create\_all(bind=self.conn) # 创建表，如果有，就不在创建

def \_get\_session(self):

# 创建session对象

Session = sessionmaker(bind=self.conn)

return Session()

def process\_item(self, item, spider):

session = self.\_get\_session() # 获取session

# 先判断news\_tag是否已经存在：如果存在，代表数据是重复的，否则才插入

if not session.query(Model).filter\_by(news\_tag=item.data['news\_tag']).all():

obj = Model(\*\*item.data) # 创建模型类对象

session.add(obj) # 插入数据

session.commit() # 提交

session.close() # 关闭session

return item

项目配置文件:

# 更改默认的配置

DEFAULT\_LOG\_FILENAME = '滚动新闻采集.log' # 默认日志文件名称

SPIDERS = [

"spiders.sina.SinaGunDong"

]

PIPELINES = [

"pipelines.Pipeline",

"pipelines.MysqlPipeline"

]

SPIDER\_MIDS = [

]

DOWNLOADER\_MIDS = [

]

# 控制最大并发数

MAX\_ASYNC\_NUMBER = 1

# 异步模式 thread， coroutine

ASYNC\_TYPE = 'thread'

'''分布式配置'''

# 执行角色

# None 代表非分布式，发起初始请求(\_start\_requests)， 处理请求(\_execute\_request\_response\_item)

# master代表主，只负责发起初始请求(\_start\_requests)，并维护请求队列

# slave代表从，只负责处理请求(\_execute\_request\_response\_item)

# ROLE = 'master'

# ROLE = 'slave'

ROLE = None

# 最大重试次数

MAX\_RETRY\_TIMES = 3

# redis 队列的配置

REDIS\_QUEUE\_NAME = 'request\_queue'

REDIS\_QUEUE\_HOST = 'localhost'

REDIS\_QUEUE\_PORT = 6379

REDIS\_QUEUE\_DB = 10

# reids 集合配置

REDIS\_SET\_NAME = 'filter\_container'

REDIS\_SET\_HOST = 'localhost'

REDIS\_SET\_PORT = 6379

REDIS\_SET\_DB = 10

# 利用redis进行请求备份 的配置

REDIS\_BACKUP\_NAME = 'request\_backup'

REDIS\_BACKUP\_HOST = 'localhost'

REDIS\_BACKUP\_PORT = 6379

REDIS\_BACKUP\_DB = 10

# 4. 扩展阅读: ASCII、unicode和UTF-8的起源

很久很久以前，有一群人，他们决定用8个可以开合的晶体管来组合成不同的状态，以表示世界上的万物。他们看到8个开关状态是好的，于是他们把这称为"字节"。

再后来，他们又做了一些可以处理这些字节的机器，机器开动了，可以用字节来组合出很多状态，状态开始变来变去。他们看到这样是好的，于是它们就这机器称为"计算机"。

开始计算机只在美国用。八位的字节一共可以组合出256(2的8次方)种不同的状态。

他们把其中的编号从0开始的32种状态分别规定了特殊的用途，一但终端、打印机遇上约定好的这些字节被传过来时，就要做一些约定的动作。遇上00x10, 终端就换行，遇上0x07, 终端就向人们嘟嘟叫，例好遇上0x1b, 打印机就打印反白的字，或者终端就用彩色显示字母。他们看到这样很好，于是就把这些0x20以下的字节状态称为"控制码"。

他们又把所有的空格、标点符号、数字、大小写字母分别用连续的字节状态表示，一直编到了第127号，这样计算机就可以用不同字节来存储英语的文字了。大家看到这样，都感觉很好，于是大家都把这个方案叫做 ANSI 的"Ascii"编码（American Standard Code for Information Interchange，美国信息互换标准代码）。当时世界上所有的计算机都用同样的ASCII方案来保存英文文字。

后来，就像建造巴比伦塔一样，世界各地的都开始使用计算机，但是很多国家用的不是英文，他们的字母里有许多是ASCII里没有的，为了可以在计算机保存他们的文字，他们决定采用127号之后的空位来表示这些新的字母、符号，还加入了很多画表格时需要用下到的横线、竖线、交叉等形状，一直把序号编到了最后一个状态255。从128到255这一页的字符集被称"扩展字符集"。从此之后，贪婪的人类再没有新的状态可以用了，美帝国主义可能没有想到还有第三世界国家的人们也希望可以用到计算机吧！

等中国人们得到计算机时，已经没有可以利用的字节状态来表示汉字，况且有6000多个常用汉字需要保存呢。但是这难不倒智慧的中国人民，我们不客气地把那些127号之后的奇异符号们直接取消掉, 规定：一个小于127的字符的意义与原来相同，但两个大于127的字符连在一起时，就表示一个汉字，前面的一个字节（他称之为高字节）从0xA1用到0xF7，后面一个字节（低字节）从0xA1到0xFE，这样我们就可以组合出大约7000多个简体汉字了。在这些编码里，我们还把数学符号、罗马希腊的字母、日文的假名们都编进去了，连在 ASCII 里本来就有的数字、标点、字母都统统重新编了两个字节长的编码，这就是常说的"全角"字符，而原来在127号以下的那些就叫"半角"字符了。

中国人民看到这样很不错，于是就把这种汉字方案叫做 "GB2312"。GB2312 是对 ASCII 的中文扩展。

但是中国的汉字太多了，我们很快就就发现有许多人的人名没有办法在这里打出来，特别是某些很会麻烦别人的国家领导人。于是我们不得不继续把 GB2312 没有用到的码位找出来老实不客气地用上。

后来还是不够用，于是干脆不再要求低字节一定是127号之后的内码，只要第一个字节是大于127就固定表示这是一个汉字的开始，不管后面跟的是不是扩展字符集里的内容。结果扩展之后的编码方案被称为 GBK 标准，GBK 包括了 GB2312 的所有内容，同时又增加了近20000个新的汉字（包括繁体字）和符号。

后来少数民族也要用电脑了，于是我们再扩展，又加了几千个新的少数民族的字，GBK 扩成了 GB18030。从此之后，中华民族的文化就可以在计算机时代中传承了。

中国的程序员们看到这一系列汉字编码的标准是好的，于是通称他们叫做 "DBCS"（Double Byte Charecter Set 双字节字符集）。在DBCS系列标准里，最大的特点是两字节长的汉字字符和一字节长的英文字符并存于同一套编码方案里，因此他们写的程序为了支持中文处理，必须要注意字串里的每一个字节的值，如果这个值是大于127的，那么就认为一个双字节字符集里的字符出现了。那时候凡是受过加持，会编程的计算机僧侣们都要每天念下面这个咒语数百遍：

"一个汉字算两个英文字符！一个汉字算两个英文字符……"

因为当时各个国家都像中国这样搞出一套自己的编码标准，结果互相之间谁也不懂谁的编码，谁也不支持别人的编码，连大陆和台湾这样只相隔了150海里，使用着同一种语言的兄弟地区，也分别采用了不同的 DBCS 编码方案——当时的中国人想让电脑显示汉字，就必须装上一个"汉字系统"，专门用来处理汉字的显示、输入的问题，但是那个台湾的愚昧封建人士写的算命程序就必须加装另一套支持 BIG5 编码的什么"倚天汉字系统"才可以用，装错了字符系统，显示就会乱了套！这怎么办？而且世界民族之林中还有那些一时用不上电脑的穷苦人民，他们的文字又怎么办？

真是计算机的巴比伦塔命题啊！

正在这时，大天使加百列及时出现了——一个叫 ISO （国际标谁化组织）的国际组织决定着手解决这个问题。他们采用的方法很简单：废了所有的地区性编码方案，重新搞一个包括了地球上所有文化、所有字母和符号的编码！他们打算叫它"Universal Multiple-Octet Coded Character Set"，简称 UCS, 俗称 "UNICODE"。

UNICODE 开始制订时，计算机的存储器容量极大地发展了，空间再也不成为问题了。于是 ISO 就直接规定必须用两个字节，也就是16位来统一表示所有的字符，对于ascii里的那些“半角”字符，UNICODE 包持其原编码不变，只是将其长度由原来的8位扩展为16位，而其他文化和语言的字符则全部重新统一编码。由于"半角"英文符号只需要用到低8位，所以其高8位永远是0，因此这种大气的方案在保存英文文本时会多浪费一倍的空间。

这时候，从旧社会里走过来的程序员开始发现一个奇怪的现象：他们的strlen函数靠不住了，一个汉字不再是相当于两个字符了，而是一个！是的，从 UNICODE 开始，无论是半角的英文字母，还是全角的汉字，它们都是统一的"一个字符"！同时，也都是统一的"两个字节"，请注意"字符"和"字节"两个术语的不同，“字节”是一个8位的物理存贮单元，而“字符”则是一个文化相关的符号。在UNICODE 中，一个字符就是两个字节。一个汉字算两个英文字符的时代已经快过去了。

从前多种字符集存在时，那些做多语言软件的公司遇上过很大麻烦，他们为了在不同的国家销售同一套软件，就不得不在区域化软件时也加持那个双字节字符集咒语，不仅要处处小心不要搞错，还要把软件中的文字在不同的字符集中转来转去。UNICODE 对于他们来说是一个很好的一揽子解决方案，于是从 Windows NT 开始，MS 趁机把它们的操作系统改了一遍，把所有的核心代码都改成了用 UNICODE 方式工作的版本，从这时开始，WINDOWS 系统终于无需要加装各种本土语言系统，就可以显示全世界上所有文化的字符了。

但是，UNICODE 在制订时没有考虑与任何一种现有的编码方案保持兼容，这使得 GBK 与UNICODE 在汉字的内码编排上完全是不一样的，没有一种简单的算术方法可以把文本内容从UNICODE编码和另一种编码进行转换，这种转换必须通过查表来进行。

如前所述，UNICODE 是用两个字节来表示为一个字符，他总共可以组合出65535不同的字符，这大概已经可以覆盖世界上所有文化的符号。如果还不够也没有关系，ISO已经准备了UCS-4方案，说简单了就是四个字节来表示一个字符，这样我们就可以组合出21亿个不同的字符出来（最高位有其他用途），这大概可以用到银河联邦成立那一天吧！

UNICODE 来到时，一起到来的还有计算机网络的兴起，UNICODE 如何在网络上传输也是一个必须考虑的问题，于是面向传输的众多 UTF（UCS Transfer Format）标准出现了，顾名思义，UTF8就是每次8个位传输数据，而UTF16就是每次16个位，只不过为了传输时的可靠性，从UNICODE到UTF时并不是直接的对应，而是要过一些算法和规则来转换。

#### 总结：

* 字符(Character)是各种文字和符号的总称，包括各国家文字、标点符号、图形符号、数字等。字符集(Character set)是多个字符的集合
* 字符集包括：ASCII字符集、GB2312字符集、GB18030字符集、Unicode字符集等
* ASCII编码是1个字节，而Unicode编码通常是2个字节。
* UTF-8是Unicode的实现方式之一，UTF-8是它是一种变长的编码方式，可以是1，2，3个字节

# 5. 参考代码

## 5.1. 爬虫中多线程多进程代码实现

单线程爬虫

单线程的爬虫速度太慢，对应的我们可以使用多线程或者是进程版本来实现 举个例子，抓取糗事百科热门栏目下的十三个url地址的段子内容，地址: <https://www.qiushibaike.com/>

普通面向对象版本

# coding=utf-8

import requests

from lxml import etree

class QiubaiSpider:

def \_\_init\_\_(self):

self.url\_temp = "https://www.qiushibaike.com/8hr/page/{}/"

self.headers = {"User-Agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X \

10\_13\_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36"}

def get\_url\_list(self): #获取url列表

return [self.url\_temp.format(i) for i in range(1,14)]

def parse\_url(self,url): #发送请求，获取响应

print(url)

return requests.get(url,headers=self.headers).content.decode()

def get\_content\_list(self,html\_str): #提取段子

html = etree.HTML(html\_str)

div\_list = html.xpath("//div[@id='content-left']/div")

content\_list = []

for div in div\_list:

content = {}

content["content"]=div.xpath(".//div[@class='content']/span/text()")

content\_list.append(content)

return content\_list

def save\_content\_list(self,content\_list): # 保存数据

pass

def run(self):

#1. url\_list

url\_list = self.get\_url\_list()

#2. 遍历，发送请求

for url in url\_list:

html\_str = self.parse\_url(url)

#3. 提取数据

content\_list = self.get\_content\_list(html\_str)

#4. 保存

self.save\_content\_list(content\_list)

if \_\_name\_\_ == '\_\_main\_\_':

qiubai = QiubaiSpider()

qiubai.run()

## 多线程爬虫

但是类似的单线程程序太慢，对应的可以考虑多线程实现，四个函数使用多个线程实现，分别使用三个队列存放数据  
代码实现如下：

# coding=utf-8

import requests

from lxml import etree

from queue import Queue

import threading

class Qiubai:

def \_\_init\_\_(self):

self.temp\_url = "https://www.qiushibaike.com/8hr/page/{}/"

self.headers= {"User-Agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X \

10\_13\_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36"}

self.url\_queue = Queue()

self.html\_queue = Queue()

self.content\_list\_queue = Queue()

def get\_url\_list(self):#获取url列表

for i in range(1,14):

self.url\_queue.put(self.temp\_url.format(i))

def parse\_url(self):

while True: #在这里使用，子线程不会结束，把子线程设置为守护线程

url = self.url\_queue.get()

print(url)

response = requests.get(url,headers=self.headers)

self.html\_queue.put(response.content.decode())

self.url\_queue.task\_done()

def get\_content\_list(self): #提取数据

while True:

html\_str = self.html\_queue.get()

html = etree.HTML(html\_str)

div\_list = html.xpath("//div[@id='content-left']/div")

content\_list = []

for div in div\_list:

content = {}

content["content"] = div.xpath(".//div[@class='content']/span/text()")

content\_list.append(content)

self.content\_list\_queue.put(content\_list)

self.html\_queue.task\_done()

def save\_content\_list(self):

while True:

content\_list = self.content\_list\_queue.get()

pass

self.content\_list\_queue.task\_done()

def run(self):

thread\_list = []

#1.url\_list

t\_url = threading.Thread(target=self.get\_url\_list)

thread\_list.append(t\_url)

#2.遍历，发送请求，

for i in range(3): #三个线程发送请求

t\_parse = threading.Thread(target=self.parse\_url)

thread\_list.append(t\_parse)

#3.提取数据

t\_content = threading.Thread(target=self.get\_content\_list)

thread\_list.append(t\_content)

#4.保存

t\_save = threading.Thread(target=self.save\_content\_list)

thread\_list.append(t\_save)

for t in thread\_list:

t.setDaemon(True) #把子线程设置为守护线程，当前这个线程不重要，主线程结束，子线程技术

t.start()

for q in [self.url\_queue,self.html\_queue,self.content\_list\_queue]:

q.join() #让主线程阻塞，等待队列的计数为0，

print("主线程结束")

if \_\_name\_\_ == '\_\_main\_\_':

qiubai = Qiubai()

qiubai.run()

上述代码中，put会让队列的计数+1，但是单纯的使用get不会让其-1，需要和task\_done同时使用才能够-1；同时task\_done不能放在另一个队列的put之前，否则可能会出现数据没有处理完成，程序结束的情况

## 多进程爬虫

这种方式由于GIL全局锁的存在，多线程在python3下可能只是个摆设，对应的解释器执行其中的内容的时候仅仅是顺序执行，此时我们可以考虑多进程的方式实现，思路和多线程相似，只是对应的api不相同。 具体的实现如下:

# coding=utf-8

import requests

from lxml import etree

import threading

from multiprocessing import Process

from multiprocessing import JoinableQueue as Queue

class QiubaiSpider:

def \_\_init\_\_(self):

self.url\_temp = "https://www.qiushibaike.com/8hr/page/{}/"

self.headers = {"User-Agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36"}

self.url\_queue = Queue() #保存url

self.html\_queue = Queue() #保存html字符串

self.content\_queue = Queue() #保存提取到的数据

def get\_url\_list(self):

for i in range(1,14):

self.url\_queue.put(self.url\_temp.format(i))

def parse\_url(self):

while True:

url = self.url\_queue.get()

print(url)

html\_str = requests.get(url,headers=self.headers).content.decode()

self.html\_queue.put(html\_str)

self.url\_queue.task\_done()

def get\_content\_list(self):

while True:

html\_str = self.html\_queue.get()

html = etree.HTML(html\_str)

div\_list = html.xpath("//div[@id='content-left']/div")

content\_list = []

for div in div\_list:

content = {}

content["content"]=div.xpath(".//div[@class='content']/span/text()")

content\_list.append(content)

self.content\_queue.put(content\_list)

self.html\_queue.task\_done()

def save\_content\_list(self):

while True:

content\_list = self.content\_queue.get()

pass

self.content\_queue.task\_done()

def run(self):

process\_list = []

#1. url\_list

t\_url = Process(target=self.get\_url\_list)

process\_list.append(t\_url)

#2. 遍历，发送请求

for i in range(5):#创建5个子进程

t\_parse = Process(target=self.parse\_url)

process\_list.append(t\_parse)

#3. 提取数据

t\_content = Process(target=self.get\_content\_list)

process\_list.append(t\_content)

#4. 保存

t\_save = Process(target=self.save\_content\_list)

process\_list.append(t\_save)

for t in process\_list:

t.daemon=True #把进线程设置为守护线程，主进程技术，子进程结束

t.start()

for q in [self.url\_queue,self.html\_queue,self.content\_queue]:

q.join() #让主进程阻塞

print("主进程结束")

if \_\_name\_\_ == '\_\_main\_\_':

qiubai = QiubaiSpider()

qiubai.run()

上述多进程实现的代码中，multiprocessing提供的JoinableQueue可以创建可连接的共享进程队列。和普通的Queue对象一样，队列允许项目的使用者通知生产者项目已经被成功处理。通知进程是使用共享的信号和条件变量来实现的。 对应的该队列能够和普通队列一样能够调用task\_done和join方法

# 5.2. 云打码的使用

云打码官方接口

import requests

import json

import time

class YDMHttp:

apiurl = 'http://api.yundama.com/api.php'

username = ''

password = ''

appid = ''

appkey = ''

def \_\_init\_\_(self, username, password, appid, appkey):

self.username = username

self.password = password

self.appid = str(appid)

self.appkey = appkey

def request(self, fields, files=[]):

response = self.post\_url(self.apiurl, fields, files)

response = json.loads(response)

return response

def balance(self):

data = {'method': 'balance', 'username': self.username, 'password': self.password, 'appid': self.appid,

'appkey': self.appkey}

response = self.request(data)

if (response):

if (response['ret'] and response['ret'] < 0):

return response['ret']

else:

return response['balance']

else:

return -9001

def login(self):

data = {'method': 'login', 'username': self.username, 'password': self.password, 'appid': self.appid,

'appkey': self.appkey}

response = self.request(data)

if (response):

if (response['ret'] and response['ret'] < 0):

return response['ret']

else:

return response['uid']

else:

return -9001

def upload(self, filename, codetype, timeout):

data = {'method': 'upload', 'username': self.username, 'password': self.password, 'appid': self.appid,

'appkey': self.appkey, 'codetype': str(codetype), 'timeout': str(timeout)}

file = {'file': filename}

response = self.request(data, file)

if (response):

if (response['ret'] and response['ret'] < 0):

return response['ret']

else:

return response['cid']

else:

return -9001

def result(self, cid):

data = {'method': 'result', 'username': self.username, 'password': self.password, 'appid': self.appid,

'appkey': self.appkey, 'cid': str(cid)}

response = self.request(data)

return response and response['text'] or ''

def decode(self, filename, codetype, timeout):

cid = self.upload(filename, codetype, timeout)

if (cid > 0):

for i in range(0, timeout):

result = self.result(cid)

if (result != ''):

return cid, result

else:

time.sleep(1)

return -3003, ''

else:

return cid, ''

def post\_url(self, url, fields, files=[]):

# for key in files:

# files[key] = open(files[key], 'rb');

res = requests.post(url, files=files, data=fields)

return res.text

username = 'whoarewe' # 用户名

password = '\*\*\*' # 密码

appid = 4283 # appid

appkey = '02074c64f0d0bb9efb2df455537b01c3' # appkey

filename = 'getimage.jpg' # 文件位置

codetype = 1004 # 验证码类型

# 超时

timeout = 60

def indetify(response\_content):

if (username == 'username'):

print('请设置好相关参数再测试')

else:

# 初始化

yundama = YDMHttp(username, password, appid, appkey)

# 登陆云打码

uid = yundama.login();

print('uid: %s' % uid)

# 查询余额

balance = yundama.balance();

print('balance: %s' % balance)

# 开始识别，图片路径，验证码类型ID，超时时间（秒），识别结果

cid, result = yundama.decode(response\_content, codetype, timeout)

print('cid: %s, result: %s' % (cid, result))

return result

def indetify\_by\_filepath(file\_path):

if (username == 'username'):

print('请设置好相关参数再测试')

else:

# 初始化

yundama = YDMHttp(username, password, appid, appkey)

# 登陆云打码

uid = yundama.login();

print('uid: %s' % uid)

# 查询余额

balance = yundama.balance();

print('balance: %s' % balance)

# 开始识别，图片路径，验证码类型ID，超时时间（秒），识别结果

cid, result = yundama.decode(file\_path, codetype, timeout)

print('cid: %s, result: %s' % (cid, result))

return result

if \_\_name\_\_ == '\_\_main\_\_':

pass