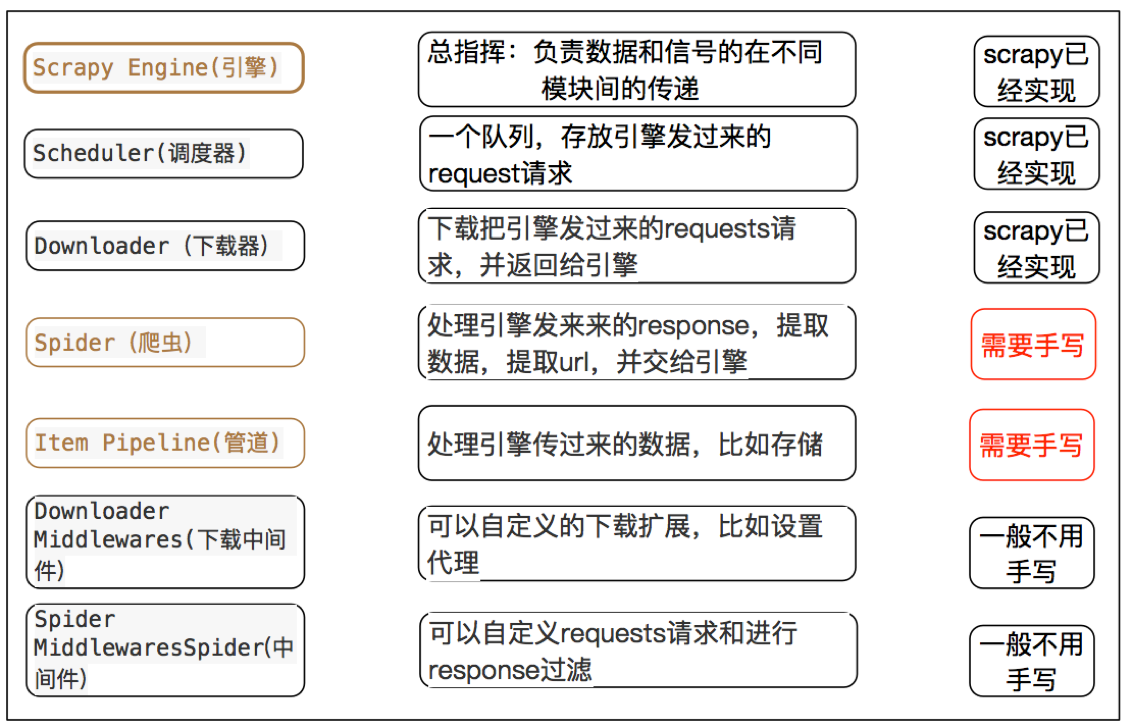


1. 调度器把requests-->引擎-->下载中间件--->下载器
2. 下载器发送请求，获取响应---->下载中间件---->引擎--->爬虫中间件--->爬虫
3. 爬虫提取url地址，组装成request对象---->爬虫中间件--->引擎--->调度器
4. 爬虫提取数据--->引擎--->管道
5. 管道进行数据的处理和保存



使用scrapy

创建一个scrapy项目:scrapy startproject 项目名

创建一个爬虫: cd 项目目录下

scrapy genspider 爬虫名 限制爬虫的域名

启动爬虫:scrapy crawl 爬虫名

爬虫的py文件中

1.response.xpath()方法的返回结果是一个类似list的类型，其中包含的是selector对象，操作和列表一样，但是有一些额外的方法

2.extract() 返回一个包含有字符串的列表

3.extract\_first() 返回列表中的第一个字符串，列表为空则返回None

4.parse()方法必须有,用来处理响应

传入管道

spider的sparse()方法 返回yield items

传递的对象只能是：BaseItem,Request,dict,None

管道

settings中开启管道

ITEM\_PIPELINES = {

'mySpider.pipelines.MyspiderPipeline': 300,

Pipeline位置 权重(数值越少越先执行)

}

Pipelines中

class MyspiderPipeline(object):

def process\_item(self, item, spider): # 必须要有接收数据的管道

if spider.name == 'itcast': # 只接受相应爬虫的数据

print(item)

return item # 传递参数

class MyspiderPipeline1(object): # 新管道要开启

…

在setting中设置日志等级：LOG\_LEVEL = "WARNING”

实现翻页请求

通过scrapy.Request(url,callback)得到一个request对象，通过yield关键字就可以把这个request对象交给引擎

eg:yield scrapy.Request(next\_page,callback=self.parse)

添加user agent

settings中添加USER\_AGENT=”...”

url地址拼接

import urllib.parse

新完整url = urllib.parse.urljoin(完整url,缺省url)

或

通过response.follow实现

yield response.follow(next\_page,callback=self.parse) #可以自动拼接缺省的next\_page

scrapy.Request的参数

scrapy.Request(url[,callback,method="GET",headers,body,cookies,meta,dont\_filter=False])

括号中的参数为可选参数

callback：表示当前的url的响应交给哪个函数去处理

meta：实现数据在不同的解析函数中传递

dont\_filter:默认会过滤请求的url地址，即请求过的url地址不会继续被请求，对需要重复请求的url地址可以把它设置为Ture，比如贴吧的翻页请求，页面的数据总是在变化;start\_urls中的地址会被反复请求，否则程序不会启动

cookies从cookies中取,而不是从headers中取

定义Item

在items中定义name = scrapy.Field(),清楚存储字段,防止以后出错

配置文件

USER\_AGENT 设置ua

ROBOTSTXT\_OBEY 是否遵守robots协议，默认是遵守

CONCURRENT\_REQUESTS 设置并发请求的数量，默认是16个

DOWNLOAD\_DELAY 下载延迟，默认无延迟

COOKIES\_ENABLED 是否开启cookie，即每次请求带上前一次的cookie，默认是开启的

DEFAULT\_REQUEST\_HEADERS 设置默认请求头

SPIDER\_MIDDLEWARES 爬虫中间件，设置过程和管道相同

DOWNLOADER\_MIDDLEWARES 下载中间件

pipelines中除了process\_item方法,还有

open\_spider(spider) :能够在爬虫开启的时候执行一次

close\_spider(spider) :能够在爬虫关闭的时候执行一次

xpath获取兄弟节点 ./following-sibling::\*[1]

crawlspider

创建scrapy genspider –t crawl itcast itcast.cn

class Itcast1Spider(CrawlSpider):

name = 'itcast1'

allowed\_domains = ['itcast.cn']

start\_urls = ['http://itcast.cn/']

# rules是一个元组或者是列表，包含的是Rule对象

rules = (

“”” LinkExtractor:连接提取器，可以通过正则或者是xpath来进行url地址的匹配

callback :表示经过连接提取器提取出来的url地址响应的回调函数，可以没有，没有表示响应不会进行回调函数的处理

follow：表示进过连接提取器提取的url地址对应的响应是否还会继续被rules中的规则进行提取，True表示会，Flase表示不会”””

Rule(LinkExtractor(allow=r'Items/'), callback='parse\_item', follow=True),

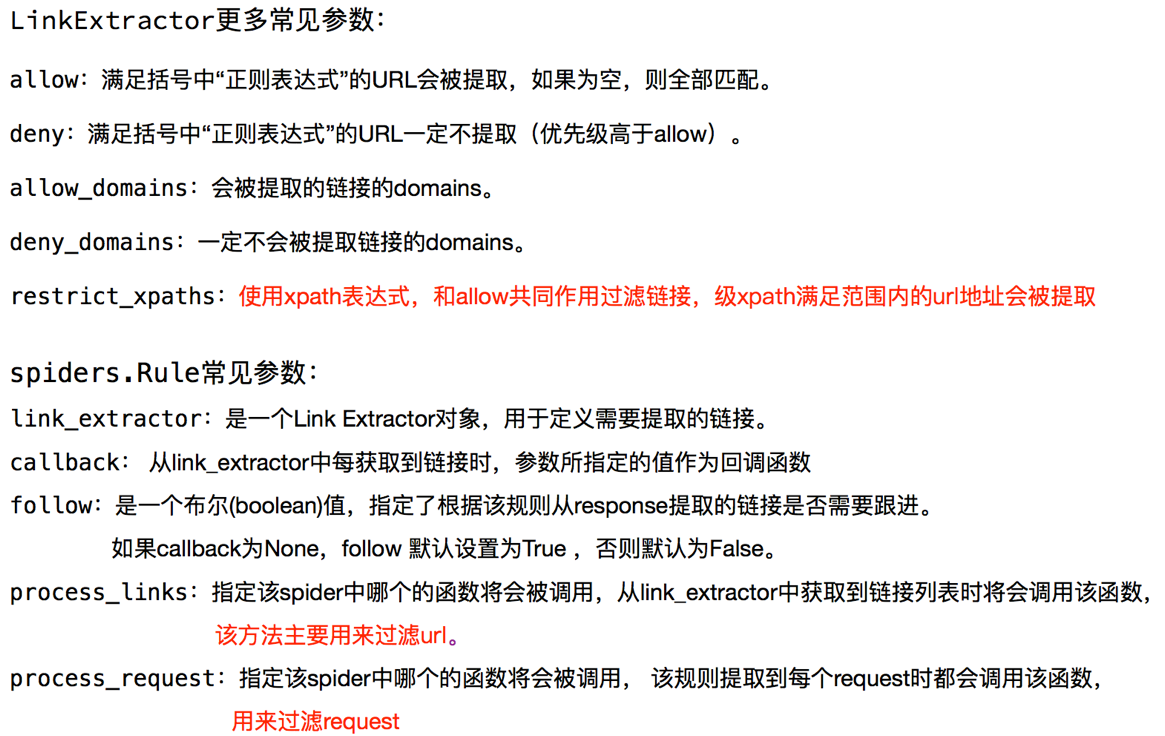
)

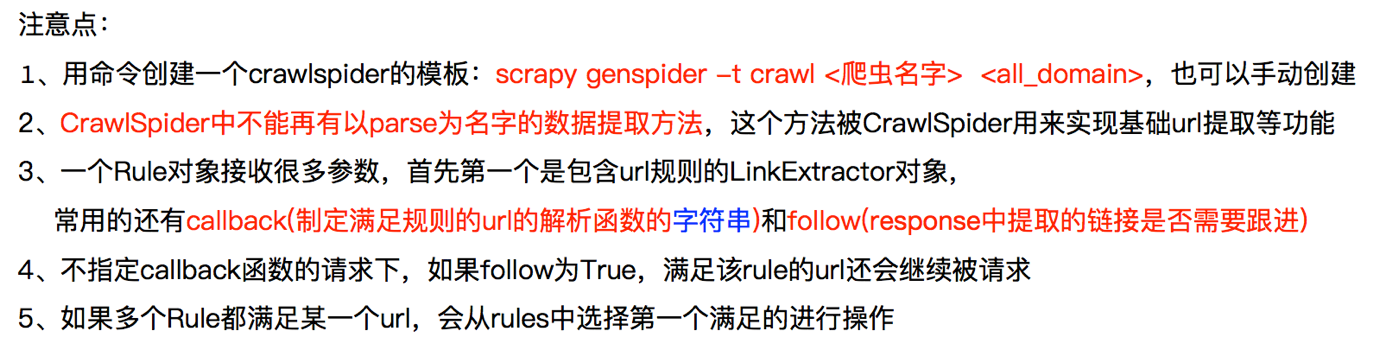
def parse\_item(self, response):

i = {}

#使用xpath进行数据的提取或者url地址的提取

return i





中间件

settings中开启Downloader Middlewares

DownloaderMiddleware类中两个方法

1. process\_request(self, request, spider)：

当每个request通过下载中间件时，该方法被调用。

返回None值：继续请求

返回Response对象：不在请求，把response返回给引擎

返回Request对象：把request对象交给调度器进行后续的请求

eg: class …

1. def process\_request(self, request, spider)：

request.headers['User-Agent'] = agent #添加请求头

request.meta['proxy'] = proxy #添加代理IP

2.process\_response(self, request, response, spider)：

返回Resposne：交给其他process\_response来处理

返回Request对象：交给调取器继续请求

使用scrapy进行模拟登陆

1. 模拟登陆携带cookie登陆,重写start\_requests方法

def start\_requests(self):

cookies\_str = "cookie\_str"

cookies\_dict = {i.split("=")[0]:i.split("=")[1] for i in cookies\_str.split("; ")}

for url in self.start\_urls:

yield scrapy.Request(

url,

callback=self.parse,

cookies=cookies\_dict, #不能写入headers中

)

第一次请求后,以后的请求会带上这次的cookie

2. 模拟登陆携带请求体,scrapy发送post请求

scrapy中通过scrapy.FormRequest能够发送post请求，同时需要添加fromdata参数作为请求体，以及callback

yield scrapy.FormRequest(

"https://github.com/session",

formdata={

"authenticity\_token":authenticity\_token,

"utf8":utf8,

"commit":commit,

"login":"noobpythoner",

"password":"zhoudawei123"

},

callback=self.parse\_login

)

3.模拟登陆,类似第2种

scrapy.Formrequest.from\_response能够自动的从响应中寻找form表单，然后把formdata中的数据提交到action对应的url地址中

def parse(self, response):

yield scrapy.FormRequest.from\_response(

response, #自动的从中寻找action对应的url地址

formdata={

"login":"noobpythoner",

"password":"\*\*\*"

},

callback = self.parse\_login

)

4.中间件中添加cookie

# cookie\_dict可用selenium获取

process\_request(self, request, spider)：

if request.url == self.start\_url[0]:

request.cookies = cookie\_dict

scrapy-redis

domz:增量式爬虫

scrapy生成指纹的方法

使用sha1加密了请求方法,请求体和请求的url地址,得到16进制字符串

fp = hashlib.sha1()

fp.update(to\_bytes(request.method))

fp.update(to\_bytes(canonicalize\_url(request.url)))

fp.update(request.body or b'')

cache[include\_headers] = fp.hexdigest() #指纹

要查看数据是否存在于redis的集合中，如果不存在就插入

added = self.server.sadd(self.key, fp)

#added= 0 表示存在

#added！=0 表示不存在，并且已经插入

return added == 0

scrapy\_redis和scrapy中request入队的条件:

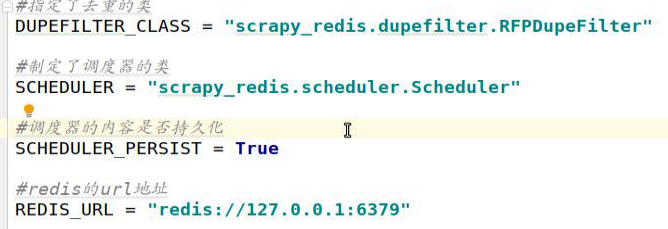
request对象没见过，新的request

dont\_filter = True 不过滤的情况下会反复抓取

start\_urls 中的url能够被反复抓取

scrapy 🡪 scrapy-redis

settings中添加:



# 指定去重的类

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

# 制定了调度器的类

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

# 调度器内容是否持久化

SCHEDULER\_PERSIST = True

#redis的地址

REDIS\_URL = "redis://127.0.0.1:6379"

RedisSpider:实现分布式爬虫



Redisspider与scrapy.spider的区别:

redisspider继承的父类是RedisSpider

redisspider没有start\_url,有redis\_key

redis\_key表示redis中存放start\_url地址的键

创建爬虫

scrapy genspider 爬虫名 允许爬取的范围

修改父类

添加redis\_key

添加增量式爬虫的settings配置(注意redis的连接配置)

启动爬虫

让爬虫就绪scrapy crawl 爬虫

lpush redis\_key url 爬虫会启动

RedisSpider:实现分布式爬虫



RedisCrawlSpider的爬虫和crwalspdier的区别

RedisCrawlSpider继承的父类是RedisCrawlSpider

RedisCrawlSpider没有start\_url,有redis\_key

redis\_key表示redis中存放start\_url地址的键

创建爬虫

scrapy genspider -t crwal 爬虫名 允许爬取的范围

修改父类

添加redis\_key

启动爬虫

让爬虫就绪 scrapy crawl 爬虫

lpush redis\_key url 爬虫会启动

cron 定时任务

crontab -l 查看

crontab -e 编辑

分 时 日 月 周 命令

0-59 0-23 1-31 1-12 0-6 commend

服务器可能要执行

cron service restart

tail -f file.sh 展示文件后几行

shell脚本写入

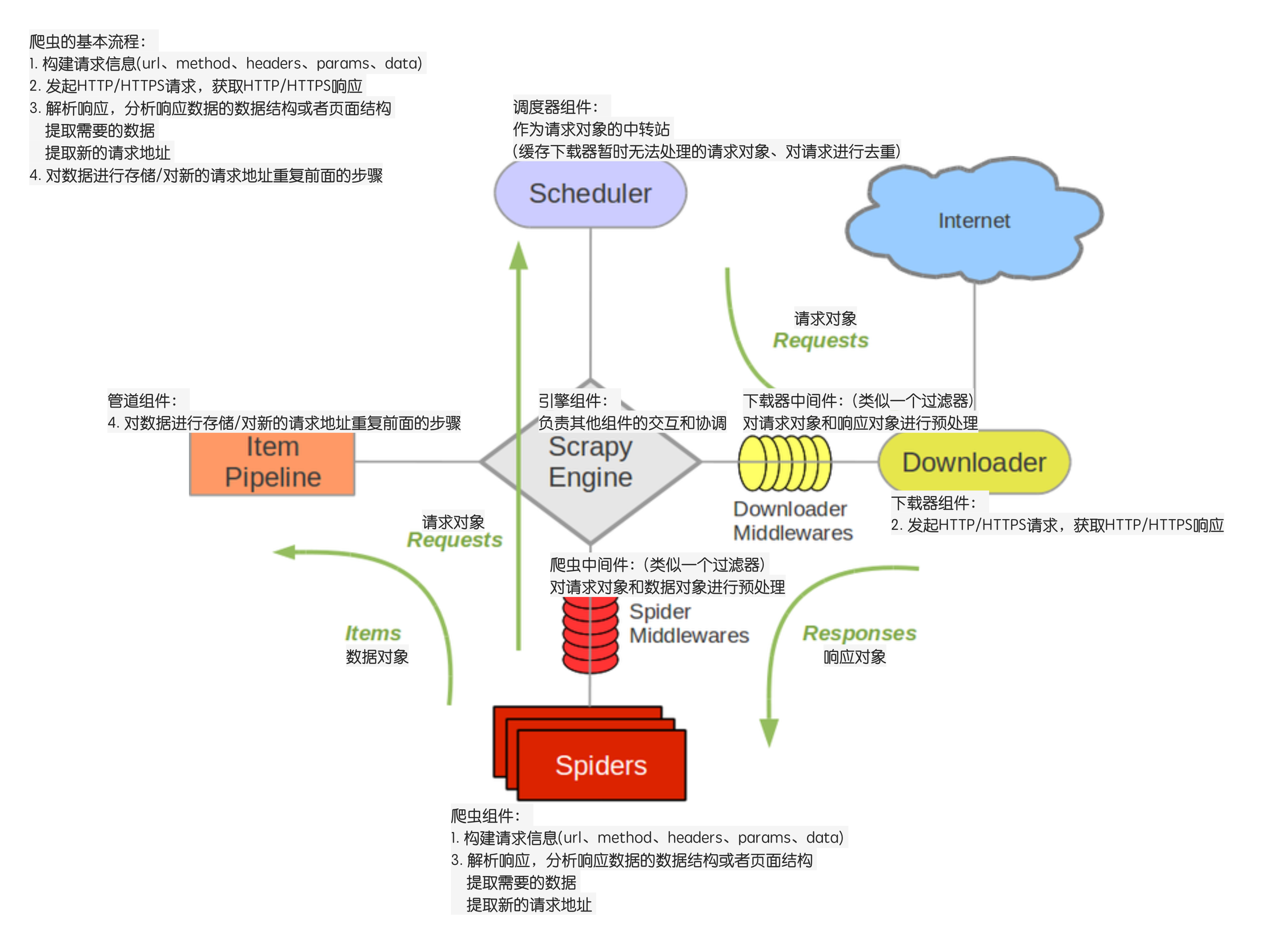
cd `dirname $0` || exit 1 #跳转到当前目录

python ./main.py >> run.log 2>&1

写入crontab

0 6 \* \* \* /home/ubuntu/..../myspider.sh >> /home/ubuntu/.../run2.log 2>&1

自定义框架



三个对象

请求对象(Request)

响应对象(Response)

数据对象(Item)

五个组件

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

两中间件

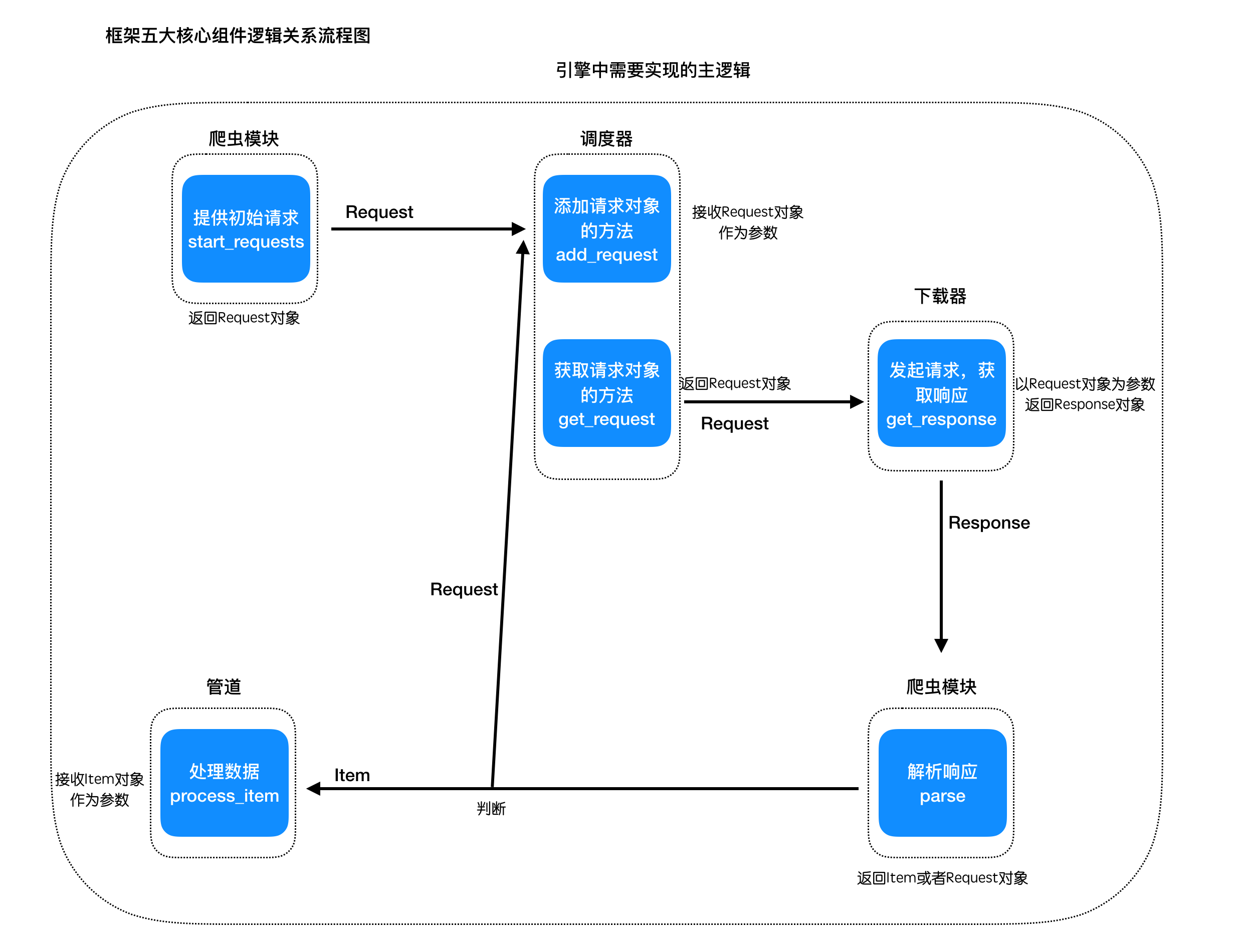
爬虫中间件:

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

逻辑关系



框架安装

一.同级目录下创建setup.py

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(可改)

version=version,

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

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

author='cjn',

author\_email='1377432548@qq.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',

],

)

二.同级目录下创建requirement.txt,自定义模块所需的其他依赖模块

requests>=2.18.4

six>=1.11.0

三.同级目录下创建VERSION.txt,指明版本号

1.0

四.终端执行命令

python setup.py install

importlib 动态导入模块

eg:

import importlib

path = ‘…’ #模块路径(相对于当前文件)

module = importlib.import\_module(path) #获取模块对象

cls = getattr(module,cls\_name) #获取模块中类的对象

func = getattr(cls,”func”) #获取类中的方法