前面我们已经介绍了 Scrapy 的一些常见用法，包括服务端渲染页面的抓取和 API 的抓取，Scrapy 发起 Request 之后，返回的 Response 里面就包含了想要的结果。

但是现在越来越多的网页都已经演变为 SPA 页面，其页面在浏览器中呈现的结果是经过 JavaScript 渲染得到的，如果我们使用 Scrapy 直接对其进行抓取的话，其结果和使用 requests 没有什么区别。

那我们真的要使用 Scrapy 完成对 JavaScript 渲染页面的抓取应该怎么办呢？

之前我们介绍了 Selenium 和 Pyppeteer 都可以实现 JavaScript 渲染页面的抓取，那用了 Scrapy 之后应该这么办呢？Scrapy 能和 Selenium 或 Pyppeteer 一起使用吗？答案是肯定的，我们可以将 Selenium 或 Pyppeteer 通过 Downloader Middleware 和 Scrapy 融合起来，实现 JavaScript 渲染页面的抓取，本节我们就来了解下它的实现吧。

回顾

在前面我们介绍了 Downloader Middleware 的用法，在 Downloader Middleware 中有三个我们可以实现的方法 process_request、process_response 以及 process_exception 方法。

我们再看下 process_request 方法和其不同的返回值的效果：

- 当返回为 None 时，Scrapy 将继续处理该 Request，接着执行其他 Downloader Middleware 的 process_request 方法，一直到 Downloader 把 Request 执行完后得到 Response 才结束。这个过程其实就是修改 Request 的过程，不同的 Downloader Middleware 按照设置的优先级顺序依次对 Request 进行修改，最后送至 Downloader 执行。
- 当返回为 Response 对象时，更低优先级的 Downloader Middleware 的 process_request 和 process_exception 方法就不会被继续调用，每个 Downloader Middleware 的 process_response 方法转而被依次调用。调用完毕之后，直接将 Response 对象发送给 Spider 来处理。
- 当返回为 Request 对象时，更低优先级的 Downloader Middleware 的 process_request 方法会停止执行。这个 Request 会重新放到调度队列里，其实它就是一个全新的 Request，等待被调度。如果被 Scheduler 调度了，那么所有的 Downloader Middleware 的 process_request 方法都会被重新按照顺序执行。
- 如果 IgnoreRequest 异常抛出，则所有的 Downloader Middleware 的 process_exception 方法会依次执行。如果没有一个方法处理这个异常，那么 Request 的 errorback 方法就会回调。如果该异常还没有被处理，那么它便会被忽略。

这里我们注意到第二个选项，当返回结果为 Response 对象时，低优先级的 process_request 方法就不会被继续调用了，这个 Response 对象会直接经由 process_response 方法处理后转交给 Spider 来解析。

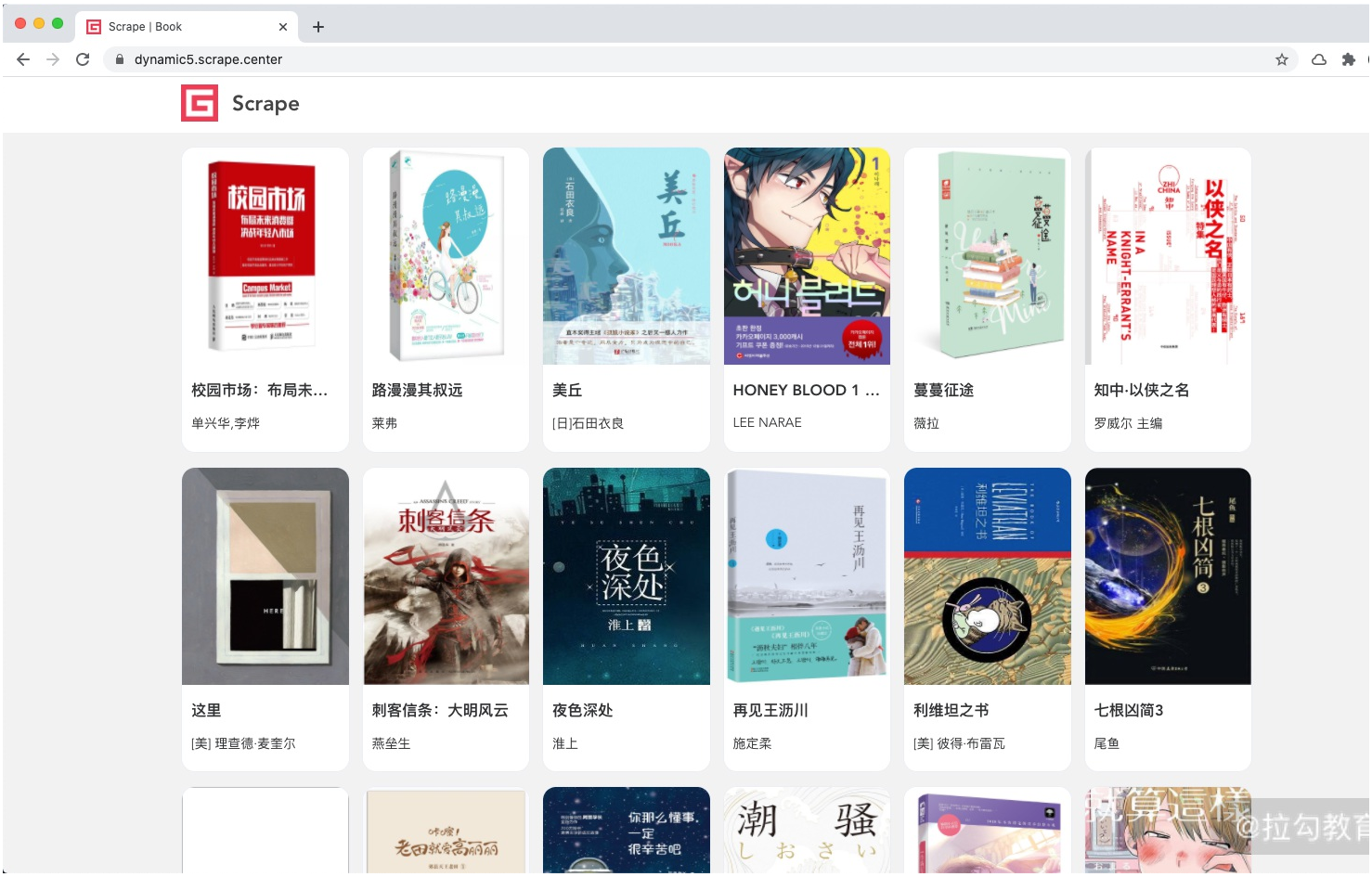
然后再接着想一想，process_request 接收的参数是 request，即 Request 对象，怎么会返回 Response 对象呢？原因可想而知了，这个 Request 对象不再经由 Scrapy 的 Downloader 来处理了，而是在 process_request 方法里面直接就完成了 Request 的发送操作，然后在得到了对应的 Response 结果后再将其返回就好了。

那么对于 JavaScript 渲染的页面来说，照这个方法来做，我们就可以把 Selenium 或 Pyppeteer 加载页面的过程在 process_request 方法里面实现，得到网页渲染完后的源代码后直接构造 Response 返回即可，这样我们就完成了借助 Downloader Middleware 实现 Scrapy 爬取动态渲染页面的过程。

案例

本节我们就用实例来讲解一下 Scrapy 和 Pyppeteer 实现 JavaScript 渲染页面抓取的流程。

本节使用的实例网站为 https://dynamic5.scrape.center/，这是一个 JavaScript 渲染页面，其内容是一本本的图书信息。



同时这个网站的页面带有分页功能，只需要在 URL 加上 /page/ 和页码就可以跳转到下一页，如 https://dynamic5.scrape.center/page/2 就是第二页内容，https://dynamic5.scrape.center/page/3 就是第三页内容。

那我们这个案例就来试着爬取前十页的图书信息吧。

实现

首先我们来新建一个项目，叫作 scrapypyppeteer，命令如下：

```
scrapy startproject scrapypyppeteer
```

接着进入项目，然后新建一个 Spider，名称为 book，命令如下：

```
cd scrapypyppeteer
```

```
scrapy genspider book dynamic5.scrape.center
```

这时候可以发现在项目的 spiders 文件夹下就出现了一个名为 spider.py 的文件，内容如下：

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


class BookSpider(scrapy.Spider):
    name = 'book'
    allowed_domains = ['dynamic5.scrape.center']
    start_urls = ['http://dynamic5.scrape.center/']

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

首先我们构造列表页的初始请求，实现一个 start_requests 方法，如下所示：

```
# -*- coding: utf-8 -*-
from scrapy import Request, Spider


class BookSpider(Spider):
    name = 'book'
    allowed_domains = ['dynamic5.scrape.center']

    base_url = 'https://dynamic5.scrape.center/page/{page}'
    max_page = 10

    def start_requests(self):
        for page in range(1, self.max_page + 1):
            url = self.base_url.format(page=page)
            yield Request(url, callback=self.parse_index)

    def parse_index(self, response):
        print(response.text)
```

这时如果我们直接运行这个 Spider，在 parse_index 方法里面打印输出 Response 的内容，结果如下：

```
<!DOCTYPE html><html lang=en><head><meta charset=utf-8><meta http-equiv=X-UA-Compatible content="IE=edge"><meta name=viewport content="width=device-width,initial-scale=1"><meta name=referrer content=no-referrer><link rel=icon href=/favicon.ico><title>Scrape | Book</title><link href=/css/chunk-617333a4.6b0815f7.css rel=prefetch><link href=/css/chunk-f52d396c.4f574d24.css rel=prefetch><link href=/js/chunk-617333a4.0efe8818.js rel=prefetch><link href=/js/chunk-f52d396c.f8f41620.js rel=prefetch><link href=/css/app.ea9d802a.css rel=preload as=style><link href=/js/app.122a9216.js rel=preload as=script><link href=/js/chunk-vendors.a02ff921.js rel=preload as=script><link href=/css/app.ea9d802a.css rel=stylesheet></head><body><noscript><strong>We're sorry but portal doesn't work properly without JavaScript enabled. Please enable it to continue.</strong></noscript><div id=app></div><script src=/js/chunk-vendors.a02ff921.js></script><script src=/js/app.122a9216.js></script></body></html>
2020-07-16 23:27:03 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://dynamic5.scrape.center/page/9> (referer: None)
<!DOCTYPE html><html lang=en><head><meta charset=utf-8><meta http-equiv=X-UA-Compatible content="IE=edge"><meta name=viewport content="width=device-width,initial-scale=1"><meta name=referrer content=no-referrer><link rel=icon href=/favicon.ico><title>Scrape | Book</title><link href=/css/chunk-617333a4.6b0815f7.css rel=prefetch><link href=/css/chunk-f52d396c.4f574d24.css rel=prefetch><link href=/js/chunk-617333a4.0efe8818.js rel=prefetch><link href=/js/chunk-f52d396c.f8f41620.js rel=prefetch><link href=/css/app.ea9d802a.css rel=preload as=style><link href=/js/app.122a9216.js rel=preload as=script><link href=/js/chunk-vendors.a02ff921.js rel=preload as=script><link href=/css/app.ea9d802a.css rel=stylesheet></head><body><noscript><strong>We're sorry but portal doesn't work properly without JavaScript enabled. Please enable it to continue.</strong></noscript><div id=app></div><script src=/js/chunk-vendors.a02ff921.js></script><script src=/js/app.122a9216.js></script></body></html>
2020-07-16 23:27:03 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://dynamic5.scrape.center/page/10> (referer: None)
2020-07-16 23:27:03 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://dynamic5.scrape.center/page/7> (referer: None)
2020-07-16 23:27:03 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://dynamic5.scrape.center/page/3> (referer: None)
2020-07-16 23:27:03 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://dynamic5.scrape.center/page/8> (referer: None)
2020-07-16 23:27:03 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://dynamic5.scrape.center/page/5> (referer: None)
2020-07-16 23:27:03 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://dynamic5.scrape.center/page/2> (referer: None)
```


我们可以发现所得到的内容并不是页面渲染后的真正 HTML 代码。此时如果我们想要获取 HTML 渲染结果的话就得使用 Downloader Middleware 实现了。

这里我们直接以一个我已经写好的组件来演示了，组件的名称叫作 GerapyPyppeteer，组件里已经写好了 Scrapy 和 Pyppeteer 结合的中间件，下面我们来详细介绍下。

我们可以借助于 pip3 来安装组件，命令如下：

```
pip3 install gerapy-pyppeteer
```

GerapyPyppeteer 提供了两部分内容，一部分是 Downloader Middleware，一部分是 Request。
首先我们需要开启中间件，在 settings 里面开启 PyppeteerMiddleware，配置如下：

```
DOWNLOADER_MIDDLEWARES = {
    'gerapy_pyppeteer.downloadermiddlewares.PyppeteerMiddleware': 543,
}
```

然后我们把上文定义的 Request 修改为 PyppeteerRequest 即可：

```
# -*- coding: utf-8 -*-
from gerapy_pyppeteer import PyppeteerRequest
from scrapy import Request, Spider


class BookSpider(Spider):
    name = 'book'
    allowed_domains = ['dynamic5.scrape.center']

    base_url = 'https://dynamic5.scrape.center/page/{page}'
    max_page = 10

    def start_requests(self):
        for page in range(1, self.max_page + 1):
            url = self.base_url.format(page=page)
            yield PyppeteerRequest(url, callback=self.parse_index, wait_for='.item .name')

    def parse_index(self, response):
        print(response.text)
```

这样其实就完成了 Pyppeteer 的对接了，非常简单。
这里 PyppeteerRequest 和原本的 Request 多提供了一个参数，就是 wait_for，通过这个参数我们可以指定 Pyppeteer 需要等待特定的内容加载出来才算结束，然后才返回对应的结果。

为了方便观察效果，我们把并发限制修改得小一点，然后把 Pyppeteer 的 Headless 模式设置为 False：

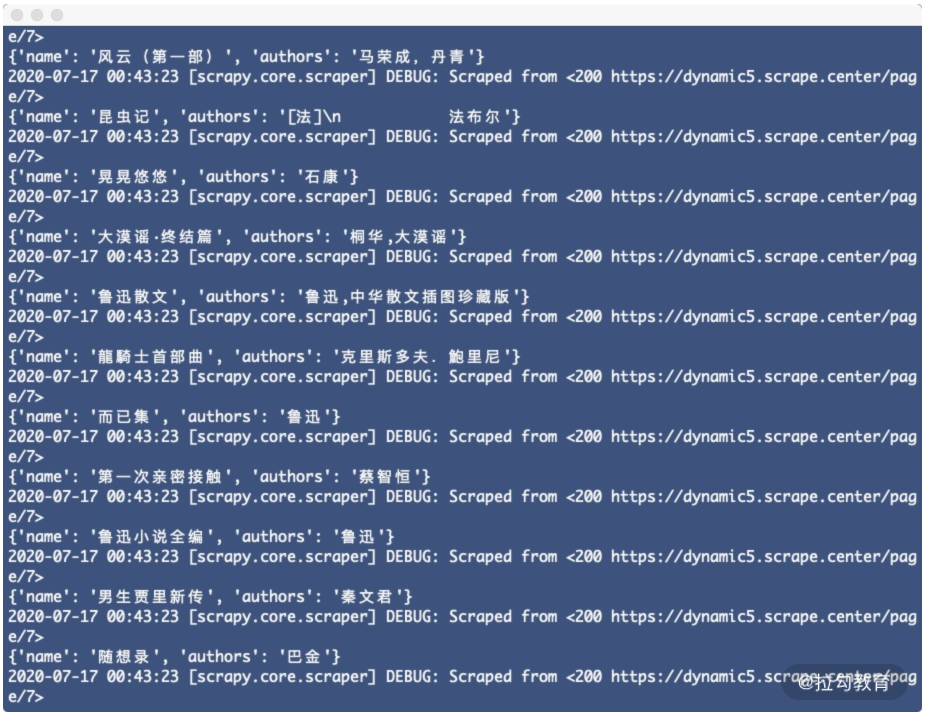
```
CONCURRENT_REQUESTS = 3
GERAPY_PYPPETEER_HEADLESS = False
```

这时我们重新运行 Spider，就可以看到在爬取的过程中，Pyppeteer 对应的 Chromium 浏览器就弹出来了，并逐个加载对应的页面内容，加载完成之后浏览器关闭。
另外观察下控制台，我们发现对应的结果也就被提取出来了，如图所示：

这时候我们再重新修改下 parse_index 方法，提取对应的每本书的名称和作者即可：

```
def parse_index(self, response):
    for item in response.css('.item'):
        name = item.css('.name::text').extract_first()
        authors = item.css('.authors::text').extract_first()
        name = name.strip() if name else None
        authors = authors.strip() if authors else None
        yield {
            'name': name,
            'authors': authors
        }
```

重新运行，即可发现对应的名称和作者就被提取出来了，运行结果如下：



这样我们就借助 GerapyPyppeteer 完成了 JavaScript 渲染页面的爬取。

原理分析

但上面仅仅是我们借助 GerapyPyppeteer 实现了 Scrapy 和 Pyppeteer 的对接，但其背后的原理是怎样的呢？

我们可以详细分析它的源码，其 GitHub 地址为 https://github.com/Gerapy/GerapyPyppeteer。

首先通过分析可以发现其最核心的内容就是实现了一个 PyppeteerMiddleware，这是一个 Downloader Middleware，这里最主要的就是 process_request 的实现，核心代码如下所示：

```
def process_request(self, request, spider):
    logger.debug('processing request %s', request)
    return as_deferred(self._process_request(request, spider))
```

这里其实就是调用了一个 _process_request 方法，这个方法的返回结果被 as_deferred 方法调用了。

这个 as_deferred 是怎么定义的呢？代码如下：

```
import asyncio
from twisted.internet.defer import Deferred

def as_deferred(f):
    return Deferred.fromFuture(asyncio.ensure_future(f))
```

这个方法接收的就是一个 asyncio 库的 Future 对象，然后通过 fromFuture 方法转化成了 twisted 里面的 Deferred 对象。这是因为 Scrapy 本身的异步是借助 twisted 实现的，一个个的异步任务对应的就是一个个 Deferred 对象，而 Pyppeteer 又是基于 asyncio 的，它的异步任务是 Future 对象，所以这里我们需要借助 Deferred 的 fromFuture 方法将 Future 转为 Deferred 对象。

另外为了支持这个功能，我们还需要在 Scrapy 中修改 reactor 对象，修改为 AsyncioSelectorReactor，实现如下：

```
import sys
from twisted.internet.asyncioreactor import AsyncioSelectorReactor
import twisted.internet

reactor = AsyncioSelectorReactor(asyncio.get_event_loop())

# install AsyncioSelectorReactor
twisted.internet.reactor = reactor
sys.modules['twisted.internet.reactor'] = reactor
```

这段代码已经在 PyppeteerMiddleware 里面定义好了，在 Scrapy 正式开始爬取之前这段代码就会被执行，将 Scrapy 中的 reactor 修改为 AsyncioSelectorReactor，从而实现 Future 的调度。
接下来我们再来看下 _process_request 方法，实现如下：

```
async def _process_request(self, request: PyppeteerRequest, spider):
    """
    use pyppeteer to process spider
    :param request:
    :param spider:
    :return:
    """
    options = {
        'headless': self.headless,
        'dumpio': self.dumpio,
        'devtools': self.devtools,
        'args': [
            f'--window-size={self.window_width},{self.window_height}',
        ]
    }
    if self.executable_path: options['executable_path'] = self.executable_path
    if self.disable_extensions: options['args'].append('--disable-extensions')
    if self.hide_scrollbars: options['args'].append('--hide-scrollbars')
    if self.mute_audio: options['args'].append('--mute-audio')
    if self.no_sandbox: options['args'].append('--no-sandbox')
    if self.disable_setuid_sandbox: options['args'].append('--disable-setuid-sandbox')
    if self.disable_gpu: options['args'].append('--disable-gpu')

    # set proxy
    proxy = request.proxy
    if not proxy:
        proxy = request.meta.get('proxy')
    if proxy: options['args'].append(f'--proxy-server={proxy}')

    logger.debug('set options %s', options)

    browser = await launch(options)
    page = await browser.newPage()
    await page.setViewport({'width': self.window_width, 'height': self.window_height})

    # set cookies
    if isinstance(request.cookies, dict):
        await page.setCookie(*[
            {'name': k, 'value': v}
            for k, v in request.cookies.items()
        ])
    else:
        await page.setCookie(request.cookies)

    # the headers must be set using request interception
    await page.setRequestInterception(True)

    @page.on('request')
    async def _handle_interception(pu_request):
        # handle headers
        overrides = {
            'headers': {
                k.decode(): ','.join(map(lambda v: v.decode(), v))
                for k, v in request.headers.items()
            }
        }
        # handle resource types
        _ignore_resource_types = self.ignore_resource_types
        if request.ignore_resource_types is not None:
            _ignore_resource_types = request.ignore_resource_types
        if pu_request.resourceType in _ignore_resource_types:
            await pu_request.abort()
        else:
            await pu_request.continue_(overrides)

    timeout = self.download_timeout
    if request.timeout is not None:
        timeout = request.timeout

    logger.debug('crawling %s', request.url)

    response = None
    try:
        options = {
            'timeout': 1000 * timeout,
            'waitUntil': request.wait_until
        }
        logger.debug('request %s with options %s', request.url, options)
        response = await page.goto(
            request.url,
            options=options
        )
```

```
    except (PageError, TimeoutError):
        logger.error('error rendering url %s using pyppeteer', request.url)
        await page.close()
        await browser.close()
        return self._retry(request, 504, spider)

    if request.wait_for:
        try:
            logger.debug('waiting for %s finished', request.wait_for)
            await page.waitFor(request.wait_for)
        except TimeoutError:
            logger.error('error waiting for %s of %s', request.wait_for, request.url)
            await page.close()
            await browser.close()
            return self._retry(request, 504, spider)

    # evaluate script
    if request.script:
        logger.debug('evaluating %s', request.script)
        await page.evaluate(request.script)

    # sleep
    if request.sleep is not None:
        logger.debug('sleep for %ss', request.sleep)
        await asyncio.sleep(request.sleep)

    content = await page.content()
    body = str.encode(content)

    # close page and browser
    logger.debug('close pyppeteer')
    await page.close()
    await browser.close()

    if not response:
        logger.error('get null response by pyppeteer of url %s', request.url)

    # Necessary to bypass the compression middleware (?)
    response.headers.pop('content-encoding', None)
    response.headers.pop('Content-Encoding', None)

    return HtmlResponse(
        page.url,
        status=response.status,
        headers=response.headers,
        body=body,
        encoding='utf-8',
        request=request
    )
```

代码内容比较多，我们慢慢来说。

首先最开始的部分是定义 Pyppeteer 的一些启动参数：

```
options = {
    'headless': self.headless,
    'dumpio': self.dumpio,
    'devtools': self.devtools,
    'args': [
        f'--window-size={self.window_width},{self.window_height}',
    ]
}
if self.executable_path: options['executable_path'] = self.executable_path
if self.disable_extensions: options['args'].append('--disable-extensions')
if self.hide_scrollbars: options['args'].append('--hide-scrollbars')
if self.mute_audio: options['args'].append('--mute-audio')
if self.no_sandbox: options['args'].append('--no-sandbox')
if self.disable_setuid_sandbox: options['args'].append('--disable-setuid-sandbox')
if self.disable_gpu: options['args'].append('--disable-gpu')
```

这些参数来自 from_crawler 里面读取项目 settings 的内容，如配置 Pyppeteer 对应浏览器的无头模式、窗口大小、是否隐藏滚动条、是否弃用沙箱，等等。

紧接着就是利用 options 来启动 Pyppeteer：

```
browser = await launch(options)
page = await browser.newPage()
await page.setViewport({'width': self.window_width, 'height': self.window_height})
```

这里启动了 Pyppeteer 对应的浏览器，将其赋值为 browser，然后新建了一个选项卡，赋值为 page，然后通过 setViewport 方法设定了窗口的宽高。

接下来就是对一些 Cookies 进行处理，如果 Request 带有 Cookies 的话会被赋值到 Pyppeteer 中：

```
# set cookies
if isinstance(request.cookies, dict):
    await page.setCookie(*[
        {'name': k, 'value': v}
        for k, v in request.cookies.items()
    ])
else:
    await page.setCookie(request.cookies)
```

再然后关键的步骤就是进行页面的加载了：

```
try:
    options = {
        'timeout': 1000 * timeout,
        'waitUntil': request.wait_until
    }
    logger.debug('request %s with options %s', request.url, options)
    response = await page.goto(
        request.url,
        options=options
    )
except (PageError, TimeoutError):
    logger.error('error rendering url %s using pyppeteer', request.url)
    await page.close()
    await browser.close()
    return self._retry(request, 504, spider)
```

这里我们首先制定了加载超时时间 timeout 还有要等待完成的事件 waitUntil，接着调用 page 的 goto 方法访问对应的页面，同时进行了异常检测，如果发生错误就关闭浏览器并重新发起一次重试请求。

在页面加载出来之后，我们还需要判定我们期望的结果是不是加载出来了，所以这里又增加了 waitFor 的调用：

```
if request.wait_for:
    try:
        logger.debug('waiting for %s finished', request.wait_for)
        await page.waitFor(request.wait_for)
    except TimeoutError:
        logger.error('error waiting for %s of %s', request.wait_for, request.url)
        await page.close()
```

```
        await browser.close()
        return self._retry(request, 504, spider)
```

这里 request 有个 wait_for 属性，就可以定义想要加载的节点的选择器，如 `.item .name` 等，这样如果页面在规定时间内加载出来就会继续向下执行，否则就会触发 TimeoutError 并被捕获，关闭浏览器并重新发起一次重试请求。

等想要的结果加载出来之后，我们还可以执行一些自定义的 JavaScript 代码完成我们想要自定义的功能：

```
# evaluate script
if request.script:
    logger.debug('evaluating %s', request.script)
    await page.evaluate(request.script)
```

最后关键的一步就是将当前页面的源代码打印出来，然后构造一个 HtmlResponse 返回即可：

```
content = await page.content()
body = str.encode(content)

# close page and browser
logger.debug('close pyppeteer')
await page.close()
await browser.close()

if not response:
    logger.error('get null response by pyppeteer of url %s', request.url)

# Necessary to bypass the compression middleware (?)
response.headers.pop('content-encoding', None)
response.headers.pop('Content-Encoding', None)

return HtmlResponse(
    page.url,
    status=response.status,
    headers=response.headers,
    body=body,
    encoding='utf-8',
    request=request
)
```

所以，如果代码可以执行到最后，返回到就是一个 Response 对象，这个 Resposne 对象的 body 就是 Pyppeteer 渲染页面后的结果，因此这个 Response 对象再传给 Spider 解析，就是 JavaScript 渲染后的页面结果了。

这样我们就通过 Downloader Middleware 通过对接 Pyppeteer 完成 JavaScript 动态渲染页面的抓取了。