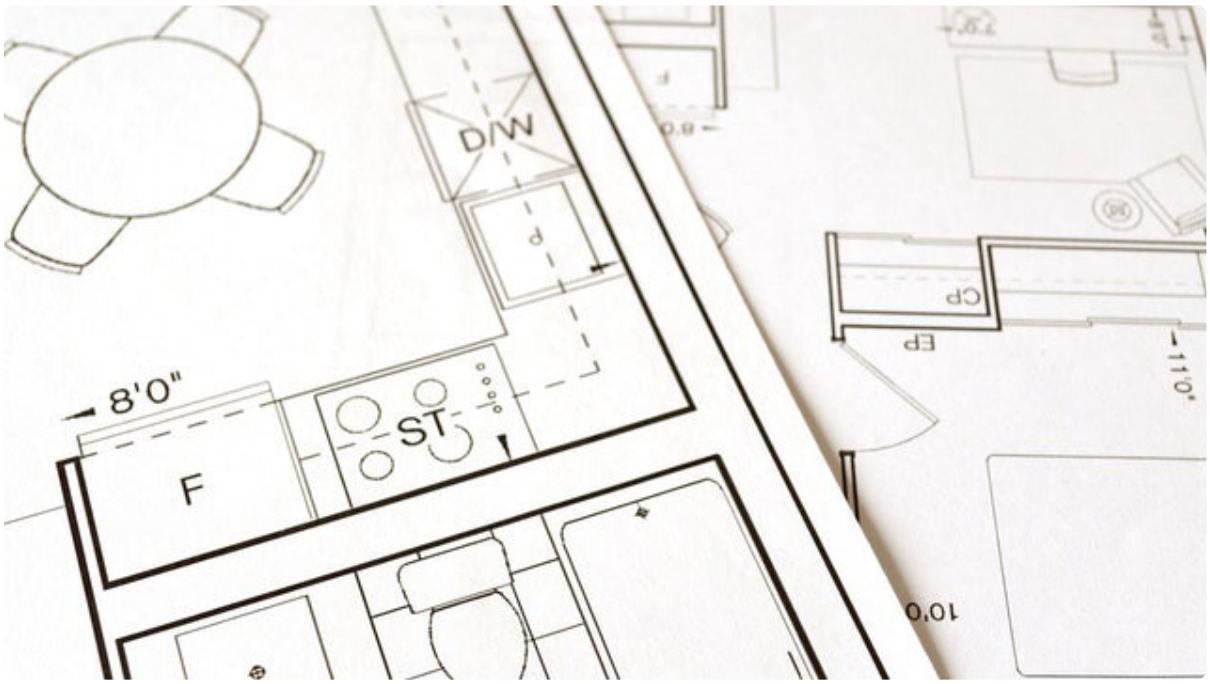
Scrapy架构总结与经验补充

更新时间：2019-07-10 14:36:43



> 时间像海绵里的水，只要你愿意挤，总还是有的。
>
> ——鲁迅

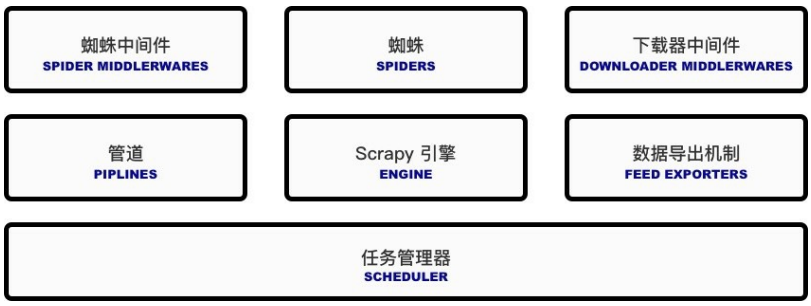
本专栏写到这里已经将所有实战内容全部介绍完了，可能细心的读者或者读过官网文档的读者心里都会发现这么一个问题：既然我们是基于 Scrapy 技术的爬虫专栏，那为啥从来不讲 Scrapy 的架构呢？

对于此我是有意为之的，记得当时我最初学习 Scrapy 时就是习惯性地先看完全了 Scrapy 的整体架构，试图从官方提供的架构图读懂 Scrapy 的动作机理之后再一个部分一个部分地学习，而且当时在网上能查到的学习资料也是按官网的这个套路来讲述的（大多是翻译）。无奈的是官网资料非常混乱即使看完了整个 Scrapy 架构也只是模模糊糊地了解一二，又或者是我的悟性比较低吧，总而言之我觉得学 Scrapy 一入手讲理论，讲架构在短时间内是很难入门的，甚至会不经意地走进理解的误区白白耗费生命。

> 生命如此短暂为何要学习烂文档？

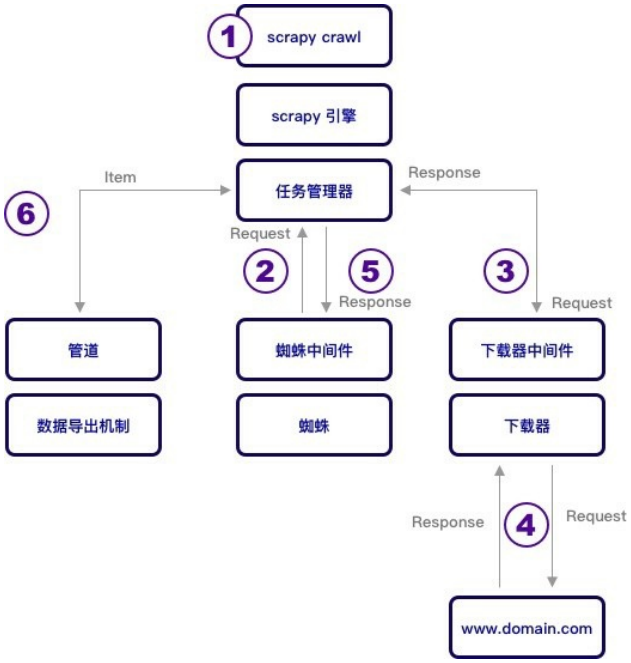
帮助其它对 Scrapy 感兴趣的朋友能更好地掌握这门技术，分享我的所得其实是驱动我去写书，写专栏的一种原动力吧。

本专栏中的所有示例的设计都是根据涉及 Scrapy 架构的不同部分的应用。我不建议你去看 Scrapy 官方提供的那个所谓的架构图，那只会让你越看越懵懂。 Scrapy 如果从结构上分析可以得到以下的方框图：



我们的专栏除了对引擎部分与任务管理器部分其它的 Scrapy 组件都分另在各个章节中与实际的示例一并介绍过了。而对Scrapy引擎的扩展对于一般甚至是高级的爬虫项目的应用机会也并不多，所以在本专栏中并未提及。

如果要将 Scrapy 的处理流程以一个单线程的模式来讲述的话则如下图所示：



结之前各章节的实践我们来看看整个Scrapy的工作过程，会对你理解之前章节中内容有所帮助，也更能让你记住每个构成部件应该在什么场合中进行使用或者扩展。

第一步，当我们通过`Scrapy crawl`指令起动 Scrapy， Scrapy 就会根据`settings`的内容将相应的组件进行实例化并且将它们动态**装配**起来。

第二步：启用任务管理器并发式地启动多个蜘蛛例实，并调用蜘蛛让其产生网络请求`Request`，这个发出请求和处理请求的过程都必然会通过**蜘蛛中间件**，如果我们希望在不改变蜘蛛生成请求或者处理响应的代码之时都可以构建一个蜘蛛中间件并将其**插入**到任务管理器与蜘蛛之间。以下是一个蜘蛛中间件的标准代码模板， 这个模板会随你调用`Scrapy startproject`指令构造 Scrapy 项目的时候就会在`middlerwares.p`文件中自动创建：

```python
class MySpiderMiddleware(object):

    def process_spider_input(self, response, spider):
      """
      当每一个响应发向蜘蛛之前都会调用此方法。应该返回一个None或引发一个异常。
      """
        return None

    def process_spider_output(self, response, result, spider):
      """
      当分析结果通过蜘蛛返回之后被调用。
      必须返回一个包括Request对象的枚举对象，字典或者数据项对象。
      """
        for i in result:
            yield i

    def process_spider_exception(self, response, exception, spider):
      """
      当蜘蛛或者process_spider_input()方法引发异常后，此方法就会被调用。
      """
        pass

    def process_start_requests(self, start_requests, spider):
      """
      当蜘蛛发出起动请求(由start_urls生成)后，此方法将被调用。
      """
        for r in start_requests:
            yield r

    def spider_opened(self, spider):
        spider.logger.info('Spider opened: %s' % spider.name)
```

蜘蛛中间件名符其实，就是位于任务管理器与蜘蛛之间的一个可插入模块，你可以在这个位置**插入代码**来扩展蜘蛛的某些能力。

第三步：将生成的请求发送到一个下载器中间件的堆栈中，请求将一个一个地通过这些下载器中间件，这些中间件就可以对请求进行各种的**加工**，例如请求重试，附带 cookie，增加请求的 UA，甚至可以不发出请求而直接生成响应对象返回，等等。例如就在第6章我们要使用 Splash 时，就需要引入一个针对 Splash 服务的下载器中间件，目的就是将的标准的网络请求转发到 Splash 服务，再由 Splash 借助无头浏览器去发出真正的请求对象。

第四步：如果请求被正确处理并返回，下载器就会生成一个 `Response` 向原来相反的路径进行传递，返回至下载器中间件，

第五步：任务管理器就将会响应对象重新传递给蜘蛛中间件，接着才将 `Response` 发送给蜘蛛的 `parse` 方法对响应对象的内容进行处理与提取。

第六步，任务管理器将蜘蛛分析出来的数据项对象(Item)发送给 `Item` 堆栈，将 `Item` 对象**流**过由各个管道，管道就对数据进行**后加工**处理。

最后，如果当前项目启用了数据导出机制蜘蛛引擎就会将数据推入数据导出器完成数据最后的送出存储处理。

以上就是 Scrapy 完整的处理流程！这个流程对于初学者如果一入手就接触只会觉得无从入手，无论哪个部分好像都很重要。将专栏的示例都动手做一次这些部分你自然就知道哪部分对你来说更重要了，不是吗？

Scrapy 的架构设计其实相当完美，它非常巧妙地运用**单一职责原则**（每个模块只负责完成一件事）将网络爬取行为细分为各个单一且可扩展的模块中，也就意味着它有着**无限**的可扩充性，这也是为何它在爬虫框架中的地位无可撼动的原因之一吧。

信号与扩展

当你遇到了连Scrapy所设计的模块都无法满足你的需求之时，Scrapy还提供了另一种更高级的扩展方法，那就是信号机制。

Scrapy是使用信号(`Signals`)来发出事件通知，事件机制是一种另到类可以具有无限扩展能力且耦合度最低的一种方式。由于Scrapy基于多线程机制开发，通过信号机制对其它接入到Scrapy运行环境的模块发布信息与传递对象是一种非常好的方法。

以下是Scrapy提供的标准信号:

- `engine_started` - 当Scrapy引擎启动爬取时发送该信号。
- `engine_stopped` - 当Scrapy引擎停止时发送该信号(例如，爬取结束)。
- `item_scraped` - 当item被爬取，并通过所有的数据管道（Item Pipelines）后没有被丢弃(dropped)，发送该信号。
- `item_dropped` - 当item通过 Item Pipeline ，有些pipeline抛出 DropItem 异常，丢弃item时，该信号被发送。
- `spider_closed` - 当某个spider被关闭时，该信号被发送。该信号可以用来释放每个spider在 spider_opened 时占用的资源。
- `spider_opened` - 当spider开始爬取时发送该信号。该信号一般用来分配spider的资源，不过其也能做任何事。
- `spider_idle` - 当spider进入空闲(idle)状态时该信号被发送
- `spider_error` - 当spider的回调函数产生错误时(例如，抛出异常)，该信号被发送。
- `request_scheduled` - 当引擎调度一个 Request 对象用于下载时，该信号被发送。
- `response_received` - 当引擎从downloader获取到一个新的 Response 时发送该信号。
- `response_downloaded` - 当一个 HTTPResponse 被下载时，由downloader发送该信号。

那我们如何来接收这些由Scrapy发出的信号呢？其实在哪里都可以接受，不过Scrapy提供了另一种终极扩展手段，那就是**扩展(`Extension`)**. 这种模块本身没有方法接口的限定，Scrapy在运行的时候也只是将配置串设定好的扩展加载进运行环境而已，但正是如此简单的模块一但与信号机制对接上就给Scrapy带来了强大的、随心所欲的扩展性。

首先，我们来看看如何使用**信号**吧，只要我们可获取`crawler`实例就可以与特定的信号进行对接了，只要可以支持`from_crawler`方法的类就可以获取`crawler`实例了，这也是专栏中出现次数最多的一个方法之一了。那我们只要使用`cralwer.signals.connect()`方法就能将事件处理器与信号对接上了：

```
from  Scrapy  import signals

def log_spider_open(spider):
  spider.log(u'蜘蛛被执行')

cralwer.signals.connect(log_spider_open,signals.spider_opened)
```

上述代码的意思就是将`signals.spider_opened`信号与`log_spider_open`函数连接上，当 Scrapy 发出`spider_opened`信号时`log_spider_open`函数就会被调用。就是这么简单！所有的信号都是这样连接的， Scrapy 内置信号的参数说明可以参考[Scrapy中文参考文档——信号(Signals)](https:// Scrapy - chs.readthedocs.io/zh_CN/0.24/topics/signals.html)

扩展就是这么一个东东了，除了可以写`from_crawler`函数而且能被 Scrapy 引导构造以外，它一无所有，但它又可以做任何事！我们就写个最简单的事件计数器，用日志输出蜘蛛被打开的次数：

```
from  Scrapy  import signals
from  Scrapy .exceptions import NotConfigured

class SpiderOpenCloseLogging(object):

    def __init__(self, item_count):
        self.item_count = item_count
        self.items_scraped = 0

    @classmethod
    def from_crawler(cls, crawler):
       ext = cls()
        # 连接蜘蛛被打开的信号
        crawler.signals.connect(ext.spider_opened, signal=signals.spider_opened)

        # 连接蜘蛛被关闭的信号
        crawler.signals.connect(ext.spider_closed, signal=signals.spider_closed)

        # 连接数据已被正确分析的信号
        crawler.signals.connect(ext.item_scraped, signal=signals.item_scraped)

        return ext

    def spider_opened(self, spider):
        spider.log("opened spider %s" % spider.name)

    def spider_closed(self, spider):
        spider.log("closed spider %s" % spider.name)

    def item_scraped(self, item, spider):
        self.items_scraped += 1
        if self.items_scraped % self.item_count == 0:
            spider.log("scraped %d items" % self.items_scraped)
```

然后，你只要在`settings.py`文件中在`EXTENSIONS`加入上述这个扩展类它就能如常工作了：

```
EXTENSIONS = {
    'mycrawler.extensions.SpiderOpenCloseLogging': 100,
}
```

命令扩展与多开爬虫

我在第一和第二章都分别介绍过不少的 Scrapy 常用指令，我们也可以通过 `Scrapy -h` 来查看所有的命令扩展，学到这里我们猜都可以猜到这又一定是 Scrapy 的另一个扩展点！确实 Scrapy 是无外不扩展的，所以我们可以通过继承 `ScrapyCommand` 来创建一个指令类去扩展 Scrapy 的标准指令。

以下为扩展指令的基本结构

```
from  Scrapy .commands import ScrapyCommand

class MyCommand(ScrapyCommand):
    requires_project = True

    def syntax(self):
      """
      输出指令的用法帮助
      """
        return "[options] <spider>"

    def short_desc(self):
        return "这里是指令的简短描述"

    def add_options(self, parser):
     """
     初始化或者增加参数捷径
     """
      pass

    def process_options(self, args, opts):
      """
      处理额外的参数选项
      """
       pass

    def run(self, args, opts):
      """
      执行指令
      """
        pass
```

增加指令后需要在`settings.py`文件中注册自定义指令的模块：

```
COMMANDS_MODULE = 'MyCralwer.MyCommand'
```

做完配置之后就可以这样来执行上述指令了：

```
$  Scrapy  mycommand
```

指令结构本身并不为我们完成什么，它只是一个特殊的程序执行入口，当我们遇到在一个项目中需要同时执行多个爬虫时可以用它来引导多个爬虫。

因为`Scrapy crawl`这个指令只能引导一个爬虫，如果要同时启动多个当然就要写一个自己的crawl指令了。

举一个简单的例子，如果我们有 spider1,spider2 两个爬虫，我们需要用一个指令同时来运行他们，那么就可以在指令类的`run`函数中这样来实现：

```
def run(self, args, opts):
   self.crawler_process.crawl('spider1')
   self.crawler_process.cralw('spider2')
   self.crawler_process.start()
```

就这么简单！你可以到度娘上查查“多开爬虫”然后就会找到一些说得讳莫如深的长文，实际上根本没那些长文说得那么复杂，只要用`ScrapyCommand.crawler_process.crawl()`方法将调用的蜘蛛加入到 Scrapy 的调度管理器，要开多少个爬虫你就执行多少次。最后调用`start()`方法一次性起动它们就OK了。

但，爬虫多开在大多数情况下是一种**伪命题**，多开爬虫必须面对这样一个问题：**共享设置**，虽然爬虫有多个，如果它们的配置是不一样的呢？那就没有办法多开！

我在初学Scrapy的时候也遇到过这样的思维困境，但只要设计对了就很少会出现多开爬虫的情况，爬虫通常会变得很单一，要爬的数据也是很单一。再复杂的数据结构或者网站结构只要多写几个爬虫就好了，共享设置只会给你带来一大堆的麻烦结果通常也只是白白浪费时间而已。

小结

通过这一章，已经是完完全全地将Scrapy的全部技术内容在本专栏中过了一次。普通的、实用的、高级的包括附加的都有了。要学好 Scrapy 不要被它看似复杂的结构所迷惑，本质上 Scrapy 是个可扩充性极强的**单细胞**集合而已，理解 Scrapy 作者的创作思维，在这个层面上才能完全掌握 Scrapy 的本质，就当这个程序是你自己写的，谋定而后动做好设计，对不理解的内容写写测试，最后再将它们整合到一起，这样无论什么样的网站就都像你自己后花园一样，想来就来想走就走了。

将采集到的图片存储于阿里云oss

专栏福利Scrapy-plus

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论