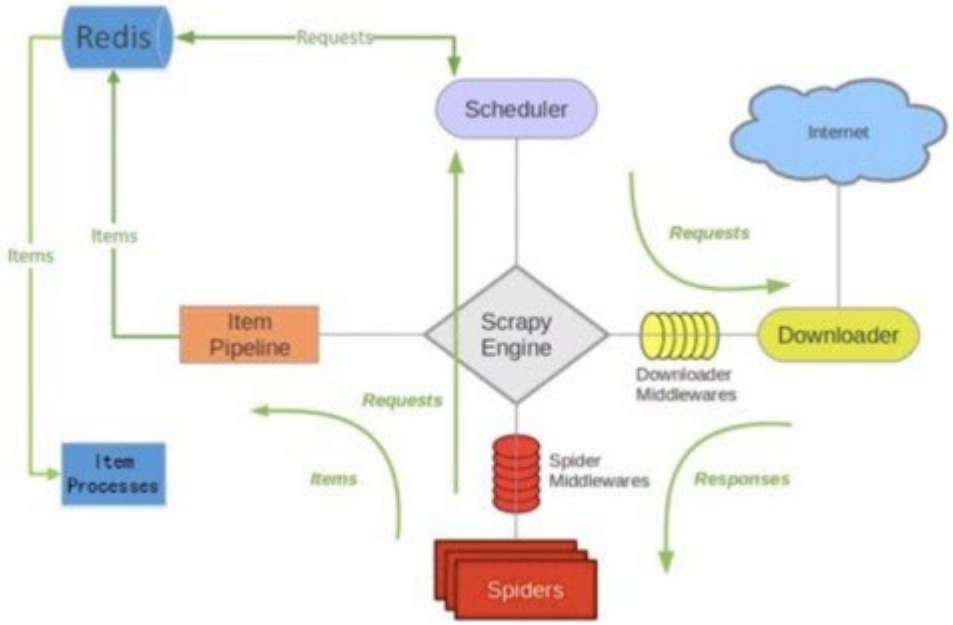
Scrapy-Redis 分布式组件

Scrapy-Redis则是一个基于Redis的Scrapy分布式组件。它利用Redis对用于爬取的请求(Requests)进行存储和调度(Schedule)，并对爬取产生的项目(items)存储以供后续处理使用。scrapy-redi重写了scrapy一些比较关键的代码，将scrapy变成一个可以在多个主机上同时运行的分布式爬虫。

加上了Scrapy-Redis之后的架构就由之前的架构变成了：



scrapy-redis的官方文档写的比较简洁，没有提及其运行原理，所以如果想全面的理解分布式爬虫的运行原理，还是得看scrapy-redis的源代码才行，不过scrapy-redis的源代码很少，也比较好懂，很快就能看完。

scrapy-redis工程的主体还是是redis和scrapy两个库，工程本身实现的东西不是很多，这个工程就像胶水一样，把这两个插件粘结了起来。

scrapy-redis提供了哪些组件?

> scrapy-redis所实现的两种分布式：爬虫分布式以及item处理分布式。分别是由模块scheduler和模块pipelines实现。

Scheduler

Scrapy改造了python本来的collection.deque(双向队列)形成了自己的Scrapy queue(https://github.com/scrapy/queuelib/blob/master/queuelib/queue.py))，但是Scrapy多个spider不能共享待爬取队列Scrapy queue， 即Scrapy本身不支持爬虫分布式，scrapy-redis 的解决是把这个Scrapy queue换成redis数据库（也是指redis队列），从同一个redis-server存放要爬取的request，便能让多个spider去同一个数据库里读取。

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

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

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

Duplication Filter

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

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

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

在scrapy-redis中去重是由Duplication Filter组件来实现的，它通过redis的set 不重复的特性，巧妙的实现了Duplication Filter去重。scrapy-redis调度器从引擎接受request，将request的指纹存入redis的set检查是否重复，并将不重复的request push写入redis的 request queue。

引擎请求request(Spider发出的）时，调度器从redis的request queue队列里里根据优先级pop 出一个request 返回给引擎，引擎将此request发给spider处理。

Item Pipeline：

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

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

Base Spider

不在使用scrapy原有的Spider类，重写的RedisSpider继承了Spider和RedisMixin这两个类，RedisMixin是用来从redis读取url的类。

当我们生成一个Spider继承RedisSpider时，调用setup_redis函数，这个函数会去连接redis数据库，然后会设置signals(信号)：

- 一个是当spider空闲时候的signal，会调用spider_idle函数，这个函数调用schedule_next_request函数，保证spider是一直活着的状态，并且抛出DontCloseSpider异常。

- 一个是当抓到一个item时的signal，会调用item_scraped函数，这个函数会调用schedule_next_request函数，获取下一个request。

scrapy-redis源码分析

源码自带项目说明：

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

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

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

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

一、dmoz (class DmozSpider(CrawlSpider))

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

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

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


class DmozSpider(CrawlSpider):
    """Follow categories and extract links."""
    name = 'dmoz'
    allowed_domains = ['dmoz.org']
    start_urls = ['http://www.dmoz.org/']

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

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

```
            }
```

二、myspider_redis (class MySpider(RedisSpider))

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

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

```
from scrapy_redis.spiders import RedisSpider


class MySpider(RedisSpider):
    """Spider that reads urls from redis queue (myspider:start_urls)."""
    name = 'myspider_redis'

    # 注意redis-key的格式:
    redis_key = 'myspider:start_urls'

    # 可选：等效于allowd_domains()，__init__方法按规定格式写，使用时只需要修改super()里的类名参数即可
    def __init__(self, *args, **kwargs):
        # Dynamically define the allowed domains list.
        domain = kwargs.pop('domain', '')
        self.allowed_domains = filter(None, domain.split(','))

        # 修改这里的类名为当前类名
        super(MySpider, self).__init__(*args, **kwargs)

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

注意： RedisSpider类 不需要写allowd_domains和start_urls：

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

执行方式：

1. 通过runspider方法执行爬虫的py文件（也可以分次执行多条），爬虫（们）将处于等待准备状态：`scrapy runspider myspider_redis.py`
2. 在Master端的redis-cli输入push指令，参考格式： `$redis > lpush myspider:start_urls http://www.dmoz.org/`
3. Slaver端爬虫获取到请求，开始爬取。

三、mycrawler_redis (class MyCrawler(RedisCrawlSpider))

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

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

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

from scrapy_redis.spiders import RedisCrawlSpider


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

    rules = (
        # follow all links
        Rule(LinkExtractor(), callback='parse_page', follow=True),
    )

    # __init__方法必须按规定写，使用时只需要修改super()里的类名参数即可
    def __init__(self, *args, **kwargs):
        # Dynamically define the allowed domains list.
        domain = kwargs.pop('domain', '')
        self.allowed_domains = filter(None, domain.split(','))

        # 修改这里的类名为当前类名
        super(MyCrawler, self).__init__(*args, **kwargs)

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

注意：

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

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

执行方式：

1. 通过runspider方法执行爬虫的py文件（也可以分次执行多条），爬虫（们）将处于等待准备状态：`scrapy runspider mycrawler_redis.py`
2. 在Master端的redis-cli输入push指令，参考格式：`$redis > lpush mycrawler:start_urls http://www.dmoz.org/`

3. 爬虫获取url，开始执行。

总结：

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