scrapy-redis 分布式

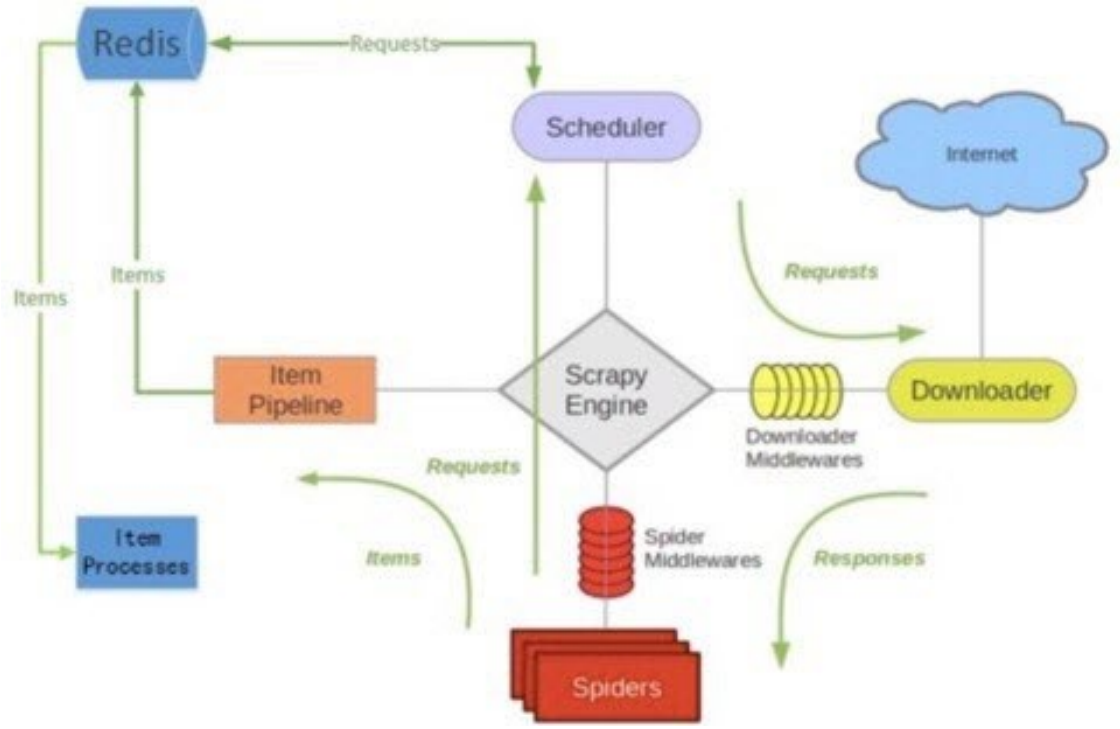
我们之前学习了 scrapy 框架，虽然爬虫是异步加多线程的，但是我们只能在一台主机上面运行，效率还是比较有限。

那么怎么解决这么问题了，一台主机太慢那我们就多加几台呗，没错，分布式爬虫就是将多台计算机组合起来，共同完成一个爬取任务。人多力量大，电脑多了肯定比一台电脑快吧，效率杠杠滴。

scrapy-redis 介绍

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。

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。

执行方式：scrapy crawl dmoz

```
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_fir
st().strip(),
                'link': div.css('a::attr(href)').extract_first(),
            }
```

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。

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。

connection.py

负责根据 setting 中配置实例化 redis 连接。被 dupefilter 和 scheduler 调用，总之涉及到 redis 存取的都要使用到这个模块。

```
import redis
import six

from scrapy.utils.misc import load_object

DEFAULT_REDIS_CLS = redis.StrictRedis

# 可以在 settings 文件中配置套接字的超时时间、等待时间等
# Sane connection defaults.
DEFAULT_PARAMS = {
    'socket_timeout': 30,
    'socket_connect_timeout': 30,
    'retry_on_timeout': True,
}

# 要想连接到 redis 数据库，和其他数据库差不多，需要一个 ip 地址、端口号、用户名密码（可选）和一个整形的数据库编号
# Shortcut maps 'setting name' -> 'parmater name'.
```

```
SETTINGS_PARAMS_MAP = {
    'REDIS_URL': 'url',
    'REDIS_HOST': 'host',
    'REDIS_PORT': 'port',
}


def get_redis_from_settings(settings):
    """Returns a redis client instance from given Scrapy settings objec
t.
    This function uses ``get_client`` to instantiate the client and use
s
    ``DEFAULT_PARAMS`` global as defaults values for the parameters. Yo
u can
    override them using the ``REDIS_PARAMS`` setting.
    Parameters
    ----------
    settings : Settings
        A scrapy settings object. See the supported settings below.
    Returns
    -------
    server
        Redis client instance.
    Other Parameters
    ----------------
    REDIS_URL : str, optional
        Server connection URL.
    REDIS_HOST : str, optional
        Server host.
    REDIS_PORT : str, optional
        Server port.
    REDIS_PARAMS : dict, optional
        Additional client parameters.
    """
    params = DEFAULT_PARAMS.copy()
    params.update(settings.getdict('REDIS_PARAMS'))
    # XXX: Deprecate REDIS_* settings.
    for source, dest in SETTINGS_PARAMS_MAP.items():
        val = settings.get(source)
        if val:
            params[dest] = val

    # Allow ``redis_cls`` to be a path to a class.
    if isinstance(params.get('redis_cls'), six.string_types):
        params['redis_cls'] = load_object(params['redis_cls'])

    # 返回的是redis 库的Redis 对象，可以直接用来进行数据操作的对象
    return get_redis(**params)
```

```
# Backwards compatible alias.
from_settings = get_redis_from_settings


def get_redis(**kwargs):
    """Returns a redis client instance.
    Parameters
    ----------
    redis_cls : class, optional
        Defaults to ``redis.StrictRedis``.
    url : str, optional
        If given, ``redis_cls.from_url`` is used to instantiate the cla
ss.
    **kwargs
        Extra parameters to be passed to the ``redis_cls`` class.
    Returns
    -------
    server
        Redis client instance.
    """
    redis_cls = kwargs.pop('redis_cls', DEFAULT_REDIS_CLS)
    url = kwargs.pop('url', None)


    if url:
        return redis_cls.from_url(url, **kwargs)
    else:
        return redis_cls(**kwargs)
```

dupefilter.py

负责执行 requst 的去重，实现的很有技巧性，使用 redis 的 set 数据结构。但是注意 scheduler 并不使用其中用于在这个模块中实现的 dupefilter 键做 request 的调度，而是使用 queue.py 模块中实现的 queue。

当 request 不重复时，将其存入到 queue 中，调度时将其弹出。

```
import logging
import time

from scrapy.dupefilters import BaseDupeFilter
from scrapy.utils.request import request_fingerprint

from .connection import get_redis_from_settings


DEFAULT_DUPEFILTER_KEY = "dupefilter:%(timestamp)s"
```

```
logger = logging.getLogger(__name__)


# TODO: Rename class to RedisDupeFilter.
class RFPDupeFilter(BaseDupeFilter):
    """Redis-based request duplicates filter.
    This class can also be used with default Scrapy's scheduler.
    """

    logger = logger

    def __init__(self, server, key, debug=False):
        """Initialize the duplicates filter.
        Parameters
        ----------
        server : redis.StrictRedis
            The redis server instance.
        key : str
            Redis key Where to store fingerprints.
        debug : bool, optional
            Whether to log filtered requests.
        """
        self.server = server
        self.key = key
        self.debug = debug
        self.logdupes = True

    @classmethod
    def from_settings(cls, settings):
        """Returns an instance from given settings.
        This uses by default the key ``dupefilter:<timestamp>``. When u
sing the
        ``scrapy_redis.scheduler.Scheduler`` class, this method is not
used as
        it needs to pass the spider name in the key.
        Parameters
        ----------
        settings : scrapy.settings.Settings
        Returns
        -------
        RFPDupeFilter
            A RFPDupeFilter instance.
        """
        server = get_redis_from_settings(settings)
        # XXX: This creates one-time key. needed to support to use this
        # class as standalone dupefilter with scrapy's default schedule
r
        # if scrapy passes spider on open() method this wouldn't be nee
ded
```

```
        # TODO: Use SCRAPY_JOB env as default and fallback to timestamp.
        key = DEFAULT_DUPEFILTER_KEY % {'timestamp': int(time.time())}
        debug = settings.getbool('DUPEFILTER_DEBUG')
        return cls(server, key=key, debug=debug)

    @classmethod
    def from_crawler(cls, crawler):
        """Returns instance from crawler.
        Parameters
        ----------
        crawler : scrapy.crawler.Crawler
        Returns
        -------
        RFPDupeFilter
            Instance of RFPDupeFilter.
        """
        return cls.from_settings(crawler.settings)

    def request_seen(self, request):
        """Returns True if request was already seen.
        Parameters
        ----------
        request : scrapy.http.Request
        Returns
        -------
        bool
        """
        fp = self.request_fingerprint(request)
        # This returns the number of values added, zero if already exis
ts.
        added = self.server.sadd(self.key, fp)
        return added == 0

    def request_fingerprint(self, request):
        """Returns a fingerprint for a given request.
        Parameters
        ----------
        request : scrapy.http.Request
        Returns
        -------
        str
        """
        return request_fingerprint(request)

    def close(self, reason=''):
        """Delete data on close. Called by Scrapy's scheduler.
        Parameters
        ----------
        reason : str, optional
```

```
        """
        self.clear()

    def clear(self):
        """Clears fingerprints data."""
        self.server.delete(self.key)

    def log(self, request, spider):
        """Logs given request.
        Parameters
        ----------
        request : scrapy.http.Request
        spider : scrapy.spiders.Spider
        """
        if self.debug:
            msg = "Filtered duplicate request: %(request)s"
            self.logger.debug(msg, {'request': request}, extra={'spider
': spider})
        elif self.logdupes:
            msg = ("Filtered duplicate request %(request)s"
                   " - no more duplicates will be shown"
                   " (see DUPEFILTER_DEBUG to show all duplicates)")
            msg = "Filtered duplicate request: %(request)s"
            self.logger.debug(msg, {'request': request}, extra={'spider
': spider})
            self.logdupes = False
```

这个文件看起来比较复杂，重写了 scrapy 本身已经实现的 request 判重功能。因为本身 scrapy 单机跑的话，只需要读取内存中的 request 队列或者持久化的 request 队列（scrapy 默认的持久化似乎是 json 格式的文件，不是数据库）就能判断这次要发出的 request url 是否已经请求过或者正在调度（本地读就行了）。而分布式跑的话，就需要各个主机上的 scheduler 都连接同一个数据库的同一个 request 池来判断这次的请求是否是重复的了。

在这个文件中，通过继承 BaseDupeFilter 重写他的方法，实现了基于 redis 的判重。根据源代码来看，scrapy-redis 使用了 scrapy 本身的一个 fingerprint 接 request_fingerprint，这个接口很有趣，根据 scrapy 文档所说，他通过 hash 来判断两个 url 是否相同（相同的 url 会生成相同的 hash 结果），但是当两个 url 的地址相同，get 型参数相同但是顺序不同时，也会生成相同的 hash 结果（这个真的比较神奇。。。）所以 scrapy-redis 依旧使用 url 的 fingerprint 来判断 request 请求是否已经出现过。

这个类通过连接 redis，使用一个 key 来向 redis 的一个 set 中插入 fingerprint（这个 key 对于同一种 spider 是相同的，redis 是一个 key-value 的数据库，如果 key 是相同的，访问到的值就是相同的，这里使用 spider 名字+DupeFilter 的 key 就是为了在不同主机上的不同爬虫实例，只要属于同一种 spider，就会访问到同一个 set，而这个 set 就是他们的 url 判重池），如果返回值为 0，说明该 set 中该

fingerprint 已经存在（因为集合是没有重复值的），则返回 False，如果返回值为 1，说明添加了一个 fingerprint 到 set 中，则说明这个 request 没有重复，于是返回 True，还顺便把新 fingerprint 加入到数据库中了。 DupeFilter 判重会在 scheduler 类中用到，每一个 request 在进入调度之前都要进行判重，如果重复就不需要参加调度，直接舍弃就好了，不然就是白白浪费资源。

queue.py

其作用如 dupefilter.py 所述，但是这里实现了三种方式的 queue：FIFO 的 SpiderQueue，SpiderPriorityQueue，以及 LIFI 的 SpiderStack。默认使用的是第二种

```
from scrapy.utils.reqser import request_to_dict, request_from_dict

from . import picklecompat


class Base(object):
    """Per-spider queue/stack base class"""

    def __init__(self, server, spider, key, serializer=None):
        """Initialize per-spider redis queue.
        Parameters:
            server -- redis connection
            spider -- spider instance
            key -- key for this queue (e.g. "%(spider)s:queue")
        """
        if serializer is None:
            # Backward compatibility.
            # TODO: deprecate pickle.
            serializer = picklecompat
        if not hasattr(serializer, 'loads'):
            raise TypeError("serializer does not implement 'loads' func
tion: %r"
                            % serializer)
        if not hasattr(serializer, 'dumps'):
            raise TypeError("serializer '%s' does not implement 'dumps'
 function: %r"
                            % serializer)

        self.server = server
        self.spider = spider
        self.key = key % {'spider': spider.name}
        self.serializer = serializer

    def _encode_request(self, request):
        """Encode a request object"""
```

```
        obj = request_to_dict(request, self.spider)
        return self.serializer.dumps(obj)

    def _decode_request(self, encoded_request):
        """Decode an request previously encoded"""
        obj = self.serializer.loads(encoded_request)
        return request_from_dict(obj, self.spider)

    def __len__(self):
        """Return the length of the queue"""
        raise NotImplementedError

    def push(self, request):
        """Push a request"""
        raise NotImplementedError

    def pop(self, timeout=0):
        """Pop a request"""
        raise NotImplementedError

    def clear(self):
        """Clear queue/stack"""
        self.server.delete(self.key)


class SpiderQueue(Base):
    """Per-spider FIFO queue"""

    def __len__(self):
        """Return the length of the queue"""
        return self.server.llen(self.key)

    def push(self, request):
        """Push a request"""
        self.server.lpush(self.key, self._encode_request(request))

    def pop(self, timeout=0):
        """Pop a request"""
        if timeout > 0:
            data = self.server.brpop(self.key, timeout)
            if isinstance(data, tuple):
                data = data[1]
        else:
            data = self.server.rpop(self.key)
        if data:
            return self._decode_request(data)


class SpiderPriorityQueue(Base):
```

```
    """Per-spider priority queue abstraction using redis' sorted set"""

    def __len__(self):
        """Return the length of the queue"""
        return self.server.zcard(self.key)

    def push(self, request):
        """Push a request"""
        data = self._encode_request(request)
        score = -request.priority
        # We don't use zadd method as the order of arguments change dep
ending on
        # whether the class is Redis or StrictRedis, and the option of
using
        # kwargs only accepts strings, not bytes.
        self.server.execute_command('ZADD', self.key, score, data)

    def pop(self, timeout=0):
        """
        Pop a request
        timeout not support in this queue class
        """
        # use atomic range/remove using multi/exec
        pipe = self.server.pipeline()
        pipe.multi()
        pipe.zrange(self.key, 0, 0).zremrangebyrank(self.key, 0, 0)
        results, count = pipe.execute()
        if results:
            return self._decode_request(results[0])


class SpiderStack(Base):
    """Per-spider stack"""

    def __len__(self):
        """Return the length of the stack"""
        return self.server.llen(self.key)

    def push(self, request):
        """Push a request"""
        self.server.lpush(self.key, self._encode_request(request))

    def pop(self, timeout=0):
        """Pop a request"""
        if timeout > 0:
            data = self.server.blpop(self.key, timeout)
            if isinstance(data, tuple):
                data = data[1]
        else:
```

```
            data = self.server.lpop(self.key)

        if data:
            return self._decode_request(data)


__all__ = ['SpiderQueue', 'SpiderPriorityQueue', 'SpiderStack']
```

该文件实现了几个容器类，可以看这些容器和 redis 交互频繁，同时使用了我们上边 picklecompat 中定义的 serializer。这个文件实现的几个容器大体相同，只不过一个是队列，一个是栈，一个是优先级队列，这三个容器到时候会被 scheduler 对象实例化，来实现 request 的调度。比如我们使用 SpiderQueue 最为调度队列的类型，到时候 request 的调度方法就是先进先出，而实用 SpiderStack 就是先进后出了。

我们可以仔细看看 SpiderQueue 的实现，他的 push 函数就和其他容器的一样，只不过 push 进去的 request 请求先被 scrapy 的接口 request*to*dict 变成了一个 dict 对象（因为 request 对象实在是比较复杂，有方法有属性不好串行化），之后使用 picklecompat 中的 serializer 串行化为字符串，然后使用一个特定的 key 存入 redis 中（该 key 在同一种 spider 中是相同的）。而调用 pop 时，其实就是从 redis 用那个特定的 key 去读其值（一个 list），从 list 中读取最早进去的那个，于是就先进先出了。

这些容器类都会作为 scheduler 调度 request 的容器，scheduler 在每个主机上都会实例化一个，并且和 spider 一一对应，所以分布式运行时会有一个 spider 的多个实例和一个 scheduler 的多个实例存在于不同的主机上，但是，因为 scheduler 都是用相同的容器，而这些容器都连接同一个 redis 服务器，又都使用 spider 名加 queue 来作为 key 读写数据，所以不同主机上的不同爬虫实例公用一个 request 调度池，实现了分布式爬虫之间的统一调度。

picklecompat.py

```
"""A pickle wrapper module with protocol=-1 by default."""

try:
    import cPickle as pickle  # PY2
except ImportError:
    import pickle


def loads(s):
    return pickle.loads(s)


def dumps(obj):
    return pickle.dumps(obj, protocol=-1)
```

这里实现了 loads 和 dumps 两个函数，其实就是实现了一个序列化器。

因为 redis 数据库不能存储复杂对象（key 部分只能是字符串，value 部分只能是字符串，字符串列表，字符串集合和 hash），所以我们存啥都要先串行化成文本才行。

这里使用的就是 python 的 pickle 模块，一个兼容 py2 和 py3 的串行化工具。这个 serializer 主要用于一会的 scheduler 存 reuqest 对象。

pipelines.py

这是是用来实现分布式处理的作用。它将 Item 存储在 redis 中以实现分布式处理。由于在这里需要读取配置，所以就用到了 from_crawler()函数。

```
from scrapy.utils.misc import load_object
from scrapy.utils.serialize import ScrapyJSONEncoder
from twisted.internet.threads import deferToThread

from . import connection


default_serialize = ScrapyJSONEncoder().encode


class RedisPipeline(object):
    """Pushes serialized item into a redis list/queue"""

    def __init__(self, server,
                 key='%(spider)s:items',
                 serialize_func=default_serialize):
        self.server = server
        self.key = key
        self.serialize = serialize_func

    @classmethod
    def from_settings(cls, settings):
        params = {
            'server': connection.from_settings(settings),
        }
        if settings.get('REDIS_ITEMS_KEY'):
            params['key'] = settings['REDIS_ITEMS_KEY']
        if settings.get('REDIS_ITEMS_SERIALIZER'):
            params['serialize_func'] = load_object(
                settings['REDIS_ITEMS_SERIALIZER']
            )

        return cls(**params)

    @classmethod
```

```
    def from_crawler(cls, crawler):
        return cls.from_settings(crawler.settings)

    def process_item(self, item, spider):
        return deferToThread(self._process_item, item, spider)

    def _process_item(self, item, spider):
        key = self.item_key(item, spider)
        data = self.serialize(item)
        self.server.rpush(key, data)
        return item

    def item_key(self, item, spider):
        """Returns redis key based on given spider.
        Override this function to use a different key depending on the
item
        and/or spider.
        """
        return self.key % {'spider': spider.name}
```

pipelines 文件实现了一个 item pipieline 类，和 scrapy 的 item pipeline 是同一个对象，通过从 settings 中拿到我们配置的 REDIS*ITEMS*KEY 作为 key，把 item 串行化之后存入 redis 数据库对应的 value 中（这个 value 可以看出出是个 list，我们的每个 item 是这个 list 中的一个结点），这个 pipeline 把提取出的 item 存起来，主要是为了方便我们延后处理数据。

scheduler.py

此扩展是对 scrapy 中自带的 scheduler 的替代（在 settings 的 SCHEDULER 变量中指出），正是利用此扩展实现 crawler 的分布式调度。其利用的数据结构来自于 queue 中实现的数据结构。

scrapy-redis 所实现的两种分布式：爬虫分布式以及 item 处理分布式就是由模块 scheduler 和模块 pipelines 实现。上述其它模块作为为二者辅助的功能模块

```
import importlib
import six

from scrapy.utils.misc import load_object

from . import connection


# TODO: add SCRAPY_JOB support.
class Scheduler(object):
    """Redis-based scheduler"""

    def __init__(self, server,
```

```
                persist=False,
                flush_on_start=False,
                queue_key='%(spider)s:requests',
                queue_cls='scrapy_redis.queue.SpiderPriorityQueue',
                dupefilter_key='%(spider)s:dupefilter',
                dupefilter_cls='scrapy_redis.dupefilter.RFPDupeFilter',
                idle_before_close=0,
                serializer=None):
        """Initialize scheduler.
        Parameters
        ----------
        server : Redis
            The redis server instance.
        persist : bool
            Whether to flush requests when closing. Default is False.
        flush_on_start : bool
            Whether to flush requests on start. Default is False.
        queue_key : str
            Requests queue key.
        queue_cls : str
            Importable path to the queue class.
        dupefilter_key : str
            Duplicates filter key.
        dupefilter_cls : str
            Importable path to the dupefilter class.
        idle_before_close : int
            Timeout before giving up.
        """
        if idle_before_close < 0:
            raise TypeError("idle_before_close cannot be negative")

        self.server = server
        self.persist = persist
        self.flush_on_start = flush_on_start
        self.queue_key = queue_key
        self.queue_cls = queue_cls
        self.dupefilter_cls = dupefilter_cls
        self.dupefilter_key = dupefilter_key
        self.idle_before_close = idle_before_close
        self.serializer = serializer
        self.stats = None

    def __len__(self):
        return len(self.queue)

    @classmethod
    def from_settings(cls, settings):
        kwargs = {
            'persist': settings.getbool('SCHEDULER_PERSIST'),
            'flush_on_start': settings.getbool('SCHEDULER_FLUSH_ON_STAR
```

```
T'),
            'idle_before_close': settings.getint('SCHEDULER_IDLE_BEFORE
_CLOSE'),
        }

        # If these values are missing, it means we want to use the defa
ults.
        optional = {
            # TODO: Use custom prefixes for this settings to note that
are
            # specific to scrapy-redis.
            'queue_key': 'SCHEDULER_QUEUE_KEY',
            'queue_cls': 'SCHEDULER_QUEUE_CLASS',
            'dupefilter_key': 'SCHEDULER_DUPEFILTER_KEY',
            # We use the default setting name to keep compatibility.
            'dupefilter_cls': 'DUPEFILTER_CLASS',
            'serializer': 'SCHEDULER_SERIALIZER',
        }
        for name, setting_name in optional.items():
            val = settings.get(setting_name)
            if val:
                kwargs[name] = val

        # Support serializer as a path to a module.
        if isinstance(kwargs.get('serializer'), six.string_types):
            kwargs['serializer'] = importlib.import_module(kwargs['seri
alizer'])

        server = connection.from_settings(settings)
        # Ensure the connection is working.
        server.ping()

        return cls(server=server, **kwargs)

    @classmethod
    def from_crawler(cls, crawler):
        instance = cls.from_settings(crawler.settings)
        # FIXME: for now, stats are only supported from this constructo
r
        instance.stats = crawler.stats
        return instance

    def open(self, spider):
        self.spider = spider

        try:
            self.queue = load_object(self.queue_cls)(
                server=self.server,
                spider=spider,
```

```
                key=self.queue_key % {'spider': spider.name},
                serializer=self.serializer,
            )
        except TypeError as e:
            raise ValueError("Failed to instantiate queue class '%s': %
s",
                             self.queue_cls, e)

        try:
            self.df = load_object(self.dupefilter_cls)(
                server=self.server,
                key=self.dupefilter_key % {'spider': spider.name},
                debug=spider.settings.getbool('DUPEFILTER_DEBUG'),
            )
        except TypeError as e:
            raise ValueError("Failed to instantiate dupefilter class '%
s': %s",
                             self.dupefilter_cls, e)

        if self.flush_on_start:
            self.flush()
        # notice if there are requests already in the queue to resume t
he crawl
        if len(self.queue):
            spider.log("Resuming crawl (%d requests scheduled)" % len(s
elf.queue))

    def close(self, reason):
        if not self.persist:
            self.flush()

    def flush(self):
        self.df.clear()
        self.queue.clear()

    def enqueue_request(self, request):
        if not request.dont_filter and self.df.request_seen(request):
            self.df.log(request, self.spider)
            return False
        if self.stats:
            self.stats.inc_value('scheduler/enqueued/redis', spider=sel
f.spider)
        self.queue.push(request)
        return True

    def next_request(self):
        block_pop_timeout = self.idle_before_close
        request = self.queue.pop(block_pop_timeout)
        if request and self.stats:
```

```
            self.stats.inc_value('scheduler/dequeued/redis', spider=sel
f.spider)
        return request

    def has_pending_requests(self):
        return len(self) > 0
```

这个文件重写了 scheduler 类，用来代替 scrapy.core.scheduler 的原有调度器。其实对原有调度器的逻辑没有很大的改变，主要是使用了 redis 作为数据存储的媒介，以达到各个爬虫之间的统一调度。 scheduler 负责调度各个 spider 的 request 请求，scheduler 初始化时，通过 settings 文件读取 queue 和 dupefilters 的类型（一般就用上边默认的），配置 queue 和 dupefilters 使用的 key（一般就是 spider name 加上 queue 或者 dupefilters，这样对于同一种 spider 的不同实例，就会使用相同的数据块了）。每当一个 request 要被调度时，enqueue*request 被调用，scheduler 使用 dupefilters 来判断这个 url 是否重复，如果不重复，就添加到 queue 的容器中（先进先出，先进后出和优先级都可以，可以在 settings 中配置）。当调度完成时，next*request 被调用，scheduler 就通过 queue 容器的接口，取出一个 request，把他发送给相应的 spider，让 spider 进行爬取工作。

spider.py

设计的这个 spider 从 redis 中读取要爬的 url,然后执行爬取，若爬取过程中返回更多的 url，那么继续进行直至所有的 request 完成。之后继续从 redis 中读取 url，循环这个过程。

分析：在这个 spider 中通过 connect signals.spider*idle 信号实现对 crawler 状态的监视。当 idle 时，返回新的 make*requests*from*url(url)给引擎，进而交给调度器调度。

```
from scrapy import signals
from scrapy.exceptions import DontCloseSpider
from scrapy.spiders import Spider, CrawlSpider

from . import connection


# Default batch size matches default concurrent requests setting.
DEFAULT_START_URLS_BATCH_SIZE = 16
DEFAULT_START_URLS_KEY = '%(name)s:start_urls'


class RedisMixin(object):
    """Mixin class to implement reading urls from a redis queue."""
    # Per spider redis key, default to DEFAULT_START_URLS_KEY.
    redis_key = None
    # Fetch this amount of start urls when idle. Default to DEFAULT_STA
RT_URLS_BATCH_SIZE.
```

```
    redis_batch_size = None
    # Redis client instance.
    server = None

    def start_requests(self):
        """Returns a batch of start requests from redis."""
        return self.next_requests()

    def setup_redis(self, crawler=None):
        """Setup redis connection and idle signal.
        This should be called after the spider has set its crawler obje
ct.
        """
        if self.server is not None:
            return

        if crawler is None:
            # We allow optional crawler argument to keep backwards
            # compatibility.
            # XXX: Raise a deprecation warning.
            crawler = getattr(self, 'crawler', None)

        if crawler is None:
            raise ValueError("crawler is required")

        settings = crawler.settings

        if self.redis_key is None:
            self.redis_key = settings.get(
                'REDIS_START_URLS_KEY', DEFAULT_START_URLS_KEY,
            )

        self.redis_key = self.redis_key % {'name': self.name}

        if not self.redis_key.strip():
            raise ValueError("redis_key must not be empty")

        if self.redis_batch_size is None:
            self.redis_batch_size = settings.getint(
                'REDIS_START_URLS_BATCH_SIZE', DEFAULT_START_URLS_BATCH
_SIZE,
            )

        try:
            self.redis_batch_size = int(self.redis_batch_size)
        except (TypeError, ValueError):
            raise ValueError("redis_batch_size must be an integer")

        self.logger.info("Reading start URLs from redis key '%(redis_ke
```

```
y)s' "
                             "(batch size: %(redis_batch_size)s)", self.__d
ict__)

        self.server = connection.from_settings(crawler.settings)
        # The idle signal is called when the spider has no requests lef
t,
        # that's when we will schedule new requests from redis queue
        crawler.signals.connect(self.spider_idle, signal=signals.spider
_idle)

    def next_requests(self):
        """Returns a request to be scheduled or none."""
        use_set = self.settings.getbool('REDIS_START_URLS_AS_SET')
        fetch_one = self.server.spop if use_set else self.server.lpop
        # XXX: Do we need to use a timeout here?
        found = 0
        while found < self.redis_batch_size:
            data = fetch_one(self.redis_key)
            if not data:
                # Queue empty.
                break
            req = self.make_request_from_data(data)
            if req:
                yield req
                found += 1
            else:
                self.logger.debug("Request not made from data: %r", dat
a)

        if found:
            self.logger.debug("Read %s requests from '%s'", found, self.
redis_key)

    def make_request_from_data(self, data):
        # By default, data is an URL.
        if '://' in data:
            return self.make_requests_from_url(data)
        else:
            self.logger.error("Unexpected URL from '%s': %r", self.redi
s_key, data)

    def schedule_next_requests(self):
        """Schedules a request if available"""
        for req in self.next_requests():
            self.crawler.engine.crawl(req, spider=self)

    def spider_idle(self):
        """Schedules a request if available, otherwise waits."""
```

```
        # XXX: Handle a sentinel to close the spider.
        self.schedule_next_requests()
        raise DontCloseSpider


class RedisSpider(RedisMixin, Spider):
    """Spider that reads urls from redis queue when idle."""

    @classmethod
    def from_crawler(self, crawler, *args, **kwargs):
        obj = super(RedisSpider, self).from_crawler(crawler, *args, **k
wargs)
        obj.setup_redis(crawler)
        return obj


class RedisCrawlSpider(RedisMixin, CrawlSpider):
    """Spider that reads urls from redis queue when idle."""

    @classmethod
    def from_crawler(self, crawler, *args, **kwargs):
        obj = super(RedisCrawlSpider, self).from_crawler(crawler, *args,
 **kwargs)
        obj.setup_redis(crawler)
        return obj
```

spider 的改动也不是很大，主要是通过 connect 接口，给 spider 绑定了 spider*idle 信号，spider 初始化时，通过 setup*redis 函数初始化好和 redis 的连接，之后通过 next*requests 函数从 redis 中取出 strat url，使用的 key 是 settings 中 REDIS*START*URLS*AS*SET 定义的（注意了这里的初始化 url 池和我们上边的 queue 的 url 池不是一个东西，queue 的池是用于调度的，初始化 url 池是存放入口 url 的，他们都存在 redis 中，但是使用不同的 key 来区分，就当成是不同的表吧），spider 使用少量的 start url，可以发展出很多新的 url，这些 url 会进入 scheduler 进行判重和调度。直到 spider 跑到调度池内没有 url 的时候，会触发 spider*idle 信号，从而触发 spider 的 next_requests 函数，再次从 redis 的 start url 池中读取一些 url。

总结

最后总结一下 scrapy-redis 的总体思路：这个工程通过重写 scheduler 和 spider 类，实现了调度、spider 启动和 redis 的交互。实现新的 dupefilter 和 queue 类，达到了判重和调度容器和 redis 的交互，因为每个主机上的爬虫进程都访问同一个 redis 数据库，所以调度和判重都统一进行统一管理，达到了分布式爬虫的目的。当 spider 被初始化时，同时会初始化一个对应的 scheduler 对象，这个调度器对象通过读取 settings，配置好自己的调度容器 queue 和判重工具 dupefilter。每当一个 spider 产出一个 request 的时候，scrapy 内核会把这个 reuqest 递交给这个

spider 对应的 scheduler 对象进行调度，scheduler 对象通过访问 redis 对 request 进行判重，如果不重复就把他添加进 redis 中的调度池。当调度条件满足时，scheduler 对象就从 redis 的调度池中取出一个 request 发送给 spider，让他爬取。当 spider 爬取的所有暂时可用 url 之后，scheduler 发现这个 spider 对应的 redis 的调度池空了，于是触发信号 spider_idle，spider 收到这个信号之后，直接连接 redis 读取 strart url 池，拿去新的一批 url 入口，然后再次重复上边的工作。