

Scrapy-redis 分布式爬虫组件

Copyright is reserved by leo. hhhparty@163.com

Scrapy作为一个通用的爬虫框架，本身是不支持分布式的。**Scrapy-redis**是为了更为方便的实现Scrapy分布式爬取而设计的一些以redis为基础的组件。

Redis 基础知识

(REmote DIctionary Server(Redis) 是一个由Salvatore Sanfilippo写的key-value存储系统。可用于缓存、事件发布或订阅、高速队列等场景。

Redis是一个开源的使用ANSI C语言编写、遵守BSD协议、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。

它通常被称为数据结构服务器，因为值（value）可以是 字符串(String), 哈希(Map), 列表(list), 集合(sets) 和 有序集合(sorted sets)等类型。

Redis 特点与优势

1. Redis 有以下三个特点：

- Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
- Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
- Redis支持数据的备份，即master-slave模式的数据备份。

1. Redis的优势表现在：

- 性能极高
- Redis能读的速度是110000次/s,写的速度是81000次/s。
- 丰富的数据类型
- Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
- 原子
- Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。
- Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。
- 丰富的特性
- Redis还支持 publish/subscribe, 通知, key 过期等等特性。
- Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。
- 在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访

问。

Redis 的应用场景

- 会话缓存（最常用）
- 消息队列，比如支付
- 活动排行榜或计数
- 发布、订阅消息（消息通知）
- 商品列表、评论列表等

Redis 安装

以在 Ubuntu 系统安装 Redis 为例，可以使用以下命令：

```
sudo apt-get update


sudo apt-get install redis-server
```

启动 Redis

```
redis-server
```

运行 redis 命令行工具

```
redis-cli
```



```
leo@ubuntu: ~
Processing triggers for ureadahead (0.100.0-19) ...
leo@ubuntu: ~$ redis-server
50645:C 02 Nov 22:17:07.737 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
50645:M 02 Nov 22:17:07.738 * Increased maximum number of open files to 10032 (it was originally set to 1024).

Redis 3.0.6 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 50645

http://redis.io

50645:M 02 Nov 22:17:07.739 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
50645:M 02 Nov 22:17:07.739 # Server started, Redis version 3.0.6
```

以上命令将打开以下终端：

```
redis 127.0.0.1:6379> 127.0.0.1 是本机 IP ， 6379 是 redis 服务端口。现在我们输入 PING 命令。
```

```
redis 127.0.0.1:6379> ping PONG
```

以上说明我们已经成功安装了redis。

Redis 的操作命令

- **slect**
 - 选择数据库(数据库编号0-15)
- **quit**
 - 退出连接
- **info**
 - 获得服务的信息与统计
- **monitor**
 - 实时监控
- **config get**
 - 获得服务配置
- **flushdb**
 - 删除当前选择的数据库中的key
- **flushall**
 - 删除所有数据库中的key

Redis 的发布与订阅

Redis发布与订阅(pub/sub)是它的一种消息通信模式，一方发送信息，一方接收信息。

Redis 持久化

redis持久有两种方式: Snapshotting(快照)、Append-only file(AOF)。

1.Snapshotting(快照)

例如将存储在内存的数据以快照的方式写入二进制文件中，如默认dump.rdb中：

```
save 900 1
```

意思是900秒内如果超过1个Key被修改，则启动快照保存。

```
save 300 10
```

意思是300秒内如果超过10个Key被修改，则启动快照保存。

```
save 60 10000
```

意思是60秒内如果超过10000个Key被修改，则启动快照保存

1. Append-only file(AOF)

在使用AOF持久时，服务会将每个收到的写命令通过write函数追加到文件中（appendonly.aof）。

AOF持久化存储方式参数说明：

- **appendonly yes**

- 开启AOF持久化存储方式
- appendfsync always
 - 收到写命令后就立即写入磁盘，效率最差，效果最好
- appendfsync everysec
 - 每秒写入磁盘一次，效率与效果居中
- appendfsync no
 - 完全依赖OS，效率最佳，效果没法保证

Scrapy-redis组件的安装

安装较为简单：

```
pip install scrapy-redis
```

当然最好还是安装在conda的虚拟环境下,例如：

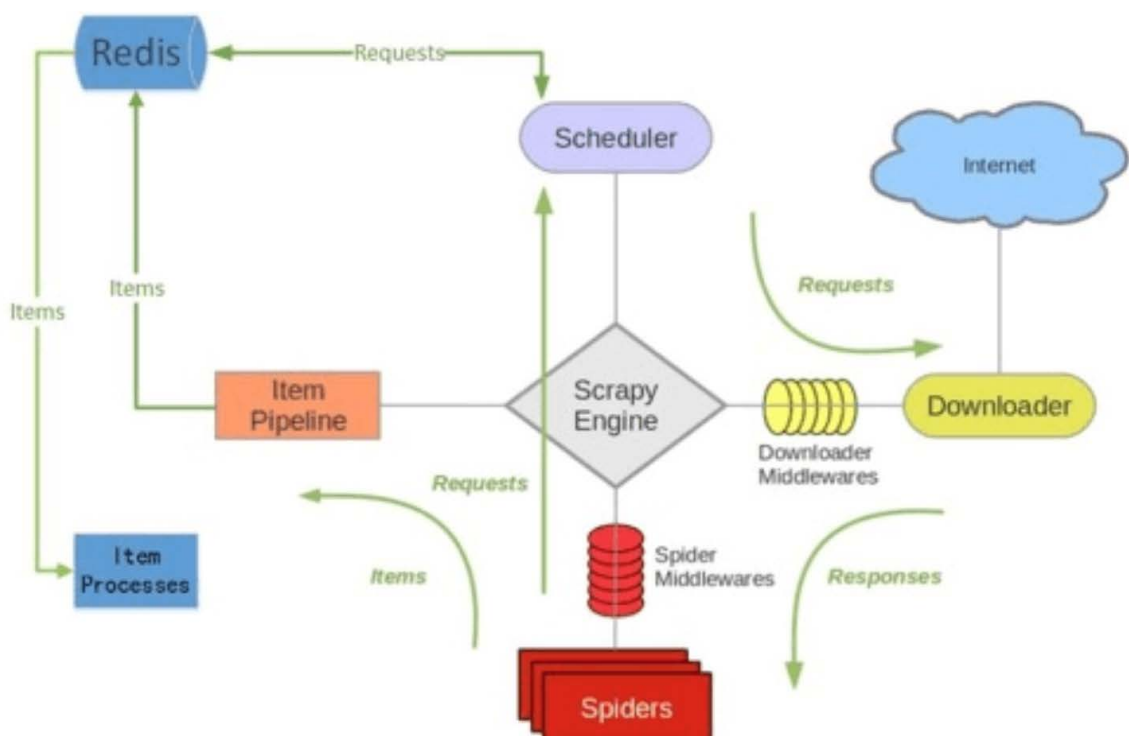
```
conda create -n myscrapyredis
conda activate myscrapyredis
pip install scrapy-redis
```

Scrapy-redis组成与架构

Scrapy-redis提供了4种组件：

- Scheduler
- Duplication Filter
- Item Pipeline
- Base Spider

Scrapy-redis的架构如图所示：



如上图所示，scrapy-redis在scrapy的架构上增加了redis，基于redis的特性拓展了如下组件：

Scheduler

Scrapy改造了python本来的collection.deque(双向队列)形成了自己的Scrapy queue。源码可以参考：

<https://github.com/scrapy/queuelib/blob/master/queuelib/queue.py>

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 的解决方法是把这个Scrapy queue换成redis数据库（或称为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进行入队列操作（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Requests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。原Scrapy Scheduler不能满足要求，而Scrapy-redis组件中的scheduler组件提供了支持。

Duplication Filter

Scrapy中用集合实现这个Requests去重功能，Scrapy中把已经发送的Request的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_fingerprints 是一个指纹集合
    fp = self.request_fingerprint(request)

    # Requests判重
    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

不再使用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空闲时候的信号，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 工作原理分析

由于Scrapy-redis项目的说明文档并不全面，所以在学习了Scrap有之后，要掌握Scrapy-redis的工作原理和具体实现方法，可以参考其源代码来理解。

Scrapy-redis是将Scrapy与redis进行了组合，发挥各自优势的结合型项目。可以通过git clone命令下载其源代码：

```
git clone https://github.com/rmax/scrapy-redis.git
```

Connection.py

在源代码文件中，我们首先了解connection.py文件内容。该文件中的代码负责根据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。

```
In [ ]: """connectio.py"""
# 这里引入了redis模块，这个是redis-python库的接口，用于通过python访问redis数据库
# 这个文件主要是实现连接redis数据库的功能，这些连接接口在其他文件中经常被用到

import six
from scrapy.utils.misc import load_object
from . import defaults

# 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' -> 'parameter name'.
SETTINGS_PARAMS_MAP = {
    'REDIS_URL': 'url',
    'REDIS_HOST': 'host',
    'REDIS_PORT': 'port',
    'REDIS_ENCODING': 'encoding',
}
```

```

def get_redis_from_settings(settings):
    """Returns a redis client instance from given Scrapy settings object
    .

    This function uses ``get_client`` to instantiate the client and uses
    ``defaults.REDIS_PARAMS`` global as defaults values for the paramete
rs. You
    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_ENCODING : str, optional
        Data encoding.
    REDIS_PARAMS : dict, optional
        Additional client parameters.

    """
    params = defaults.REDIS_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 class.

    **kwargs
        Extra parameters to be passed to the ``redis_cls`` class.

    Returns
    -----
    server
        Redis client instance.

    """
    redis_cls = kwargs.pop('redis_cls', defaults.REDIS_CLS)
    url = kwargs.pop('url', None)
    if url:
        return redis_cls.from_url(url, **kwargs)
    else:
        return redis_cls(**kwargs)

```

dupefilter.py

该文件内的代码负责执行request的去重，使用了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。

基本思路：当request不重复时，将其存入到queue中，调度时将其弹出。

dupefilter.py较为复杂，重写了scrapy本身已经实现的request判重功能。原因是Scrapy通过读取内存中的requests队列或者持久化的requests队列（json格式，而非数据库），判断待执行的request是否已经被调度；而Scrapy-redis的分布式运行需要各个主机上的scheduler都连接同一个数据库的同一个requests池来判断这次的请求是否是重复。

dupefilter.py中RFPDupeFilter类继承了BaseDupeFilter类，并重写了部分方法，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，它通过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（细节考虑充分），所以scrapy-redis依旧使用url的fingerprint来判断request请求是否已经出现过。

RFPDupeFilter这个类的基本工作原理：

1. 连接redis，使用一个key来向redis的一个set中插入fingerprint，并获取返回值；

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。

1. 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；
2. 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，新fingerprint还将加入到数据库中。

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，以免浪费计算与存储资源。

```

In [ ]: """dupefilter.py"""
import logging
import time

```



```

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

from . import defaults
from .connection import get_redis_from_settings

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.logdups = True

    @classmethod
    def from_settings(cls, settings):
        """Returns an instance from given settings.

        This uses by default the key ``dupefilter:<timestamp>``. When us
ing the
        ``scrapy_redis.scheduler.Scheduler`` class, this method is not u
sed 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 scheduler
# if scrapy passes spider on open() method this wouldn't be need
ed

# TODO: Use SCRAPY_JOB env as default and fallback to timestamp.
key = defaults.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 exist
s.
    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)

@classmethod
def from_spider(cls, spider):
    settings = spider.settings
    server = get_redis_from_settings(settings)

```

```

        dupefilter_key = settings.get("SCHEDULER_DUPEFILTER_KEY", default
ts.SCHEDULER_DUPEFILTER_KEY)
        key = dupefilter_key % {'spider': spider.name}
        debug = settings.getbool('DUPEFILTER_DEBUG')
        return cls(server, key=key, debug=debug)

    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.logdups:
            msg = ("Filtered duplicate request %(request)s"
                  " - no more duplicates will be shown"
                  " (see DUPEFILTER_DEBUG to show all duplicates)")
            self.logger.debug(msg, {'request': request}, extra={'spider'
: spider})
        self.logdups = False

```

picklecompat.py

该文件实现了loads和dumps两个函数，即实现了一个序列化工具。

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值。

这里引用了python的pickle模块，它是一个兼容py2和py3的序列化工具。这个serializer主要用于scheduler中存取reuquest对象。

什么是序列化 (Serialization)?

序列化是将对象的状态信息转换为可以存储或传输的形式过程。在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象。

```
In [ ]: """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)
```

pipelines.py

这个文件中的代码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。

from_crawler()函数用来从当前爬虫项目中读取配置信息。

pipelines文件实现了一个item pipeline类，和scrapy的item pipeline是同一个对象。

基本过程是：

1. 通过从settings中拿到我们配置的REDIS_ITEMS_KEY作为key；
2. 把item串行化之后存入redis数据库对应的value中（这个value可以看出是个list，我们的每个item是这个list中的一个结点）；
3. 把这个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。

```
In [ ]: """pipeline.py"""

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

from . import connection, defaults

default_serialize = ScrapyJSONEncoder().encode

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

    Settings
    -----
    REDIS_ITEMS_KEY : str
        Redis key where to store items.
    REDIS_ITEMS_SERIALIZER : str
        Object path to serializer function.

    """
```

```

def __init__(self, server,
              key=defaults.PIPELINE_KEY,
              serialize_func=default_serialize):
    """Initialize pipeline.

    Parameters
    -----
    server : StrictRedis
        Redis client instance.
    key : str
        Redis key where to store items.
    serialize_func : callable
        Items serializer function.

    """
    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 i
        tem
        and/or spider.

        """
        return self.key % {'spider': spider.name}

```

queue.py

该文件实现了几个容器类，可以看出这些容器和redis交互频繁，同时使用了我们上

边picklecompat中定义的序列化器。

文件中定义的几个容器有：

- 先入先出的队列FifoQueue
 - 别名：SpiderQueue
- 后进先出的栈LifoQueue
 - 别名：SpiderStack = LifoQueue
- 优先级队列PriorityQueue
 - 别名：SpiderPriorityQueue

这三个容器到时候会被scheduler对象实例化，来实现request的调度。

从SpiderQueue的实现看出来，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。

```
In [ ]: """queue.py"""
from scrapy.utils.reqser import request_to_dict, request_from_dict

from . import picklecompat

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

    def __init__(self, server, spider, key, serializer=None):
        """Initialize per-spider redis queue.

        Parameters
        -----
        server : StrictRedis
            Redis client instance.
        spider : Spider
            Scrapy spider instance.
        key: str
            Redis key where to put and get messages.
        serializer : object
            Serializer object with ``loads`` and ``dumps`` methods.

        """
        if serializer is None:
            # Backward compatibility.
            # TODO: deprecate pickle.
            serializer = picklecompat
        if not hasattr(serializer, 'loads'):
            raise TypeError("serializer does not implement 'loads' function: %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 FifoQueue(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 PriorityQueue(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 depending 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 LifoQueue(Base):
    """Per-spider LIFO queue."""

    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)

# TODO: Deprecate the use of these names.
SpiderQueue = FifoQueue
SpiderStack = LifoQueue
SpiderPriorityQueue = PriorityQueue

```


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。上述其它模块作为二者辅助的功能模块。

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的请求请求，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。

```
In [ ]: """scheduler.py"""
import importlib
import six

from scrapy.utils.misc import load_object

from . import connection, defaults

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

    Settings
    -----
    SCHEDULER_PERSIST : bool (default: False)
        Whether to persist or clear redis queue.
    SCHEDULER_FLUSH_ON_START : bool (default: False)
        Whether to flush redis queue on start.
    SCHEDULER_IDLE_BEFORE_CLOSE : int (default: 0)
        How many seconds to wait before closing if no message is received
    .
    SCHEDULER_QUEUE_KEY : str
        Scheduler redis key.
    SCHEDULER_QUEUE_CLASS : str
        Scheduler queue class.
    SCHEDULER_DUPFILTER_KEY : str
        Scheduler dupefilter redis key.
    SCHEDULER_DUPFILTER_CLASS : str
        Scheduler dupefilter class.
    SCHEDULER_SERIALIZER : str
        Scheduler serializer.

    """

    def __init__(self, server,
                 persist=False,
                 flush_on_start=False,
```

```

        queue_key=defaults.SCHEDULER_QUEUE_KEY,
        queue_cls=defaults.SCHEDULER_QUEUE_CLASS,
        dupefilter_key=defaults.SCHEDULER_DUPEFILTER_KEY,
        dupefilter_cls=defaults.SCHEDULER_DUPEFILTER_CLASS,
        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_START'),
            'idle_before_close': settings.getint('SCHEDULER_IDLE_BEFORE_CLOSE'),
        }

        # If these values are missing, it means we want to use the defaults.
        optional = {
            # TODO: Use custom prefixes for this settings to note that a
            # 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['serializer'])

    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 constructor
        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))

        self.df = load_object(self.dupefilter_cls).from_spider(spider)

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

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

```

spider.py

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。

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:这里的初始化url池和我们上边的queue的url池不一样:

- queue的池是用于调度的,存在redis库中;
- 初始化url池是存放入口url的,存在redis库中;
- 两者使用不同的key来区分,可以看作是不同的表。

Spider通过对少量的start url指定页面的爬取,可以发现很多新的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。

```

In [ ]: """spider.py"""
from scrapy import signals
from scrapy.exceptions import DontCloseSpider
from scrapy.spiders import Spider, CrawlSpider

from . import connection, defaults
from .utils import bytes_to_str

class RedisMixin(object):

```

```

"""Mixin class to implement reading urls from a redis queue."""
redis_key = None
redis_batch_size = None
redis_encoding = None

# Redis client placeholder.
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 objec
t.
    """
    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', defaults.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:
        # TODO: Deprecate this setting (REDIS_START_URLS_BATCH_SIZE).
        self.redis_batch_size = settings.getint(
            'REDIS_START_URLS_BATCH_SIZE',
            settings.getint('CONCURRENT_REQUESTS'),
        )

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

    if self.redis_encoding is None:
        self.redis_encoding = settings.get('REDIS_ENCODING', default
s.REDIS_ENCODING)

    self.logger.info("Reading start URLs from redis key '%(redis_key
)s' "
                    "(batch size: %(redis_batch_size)s, encoding: %(

```

```

redis_encoding)s",
                                self.__dict__)

    self.server = connection.from_settings(crawler.settings)
    # The idle signal is called when the spider has no requests left
    '

    # 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', default=
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
        # TODO: Use redis pipeline execution.
        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", data
)

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

    def make_request_from_data(self, data):
        """Returns a Request instance from data coming from Redis.

        By default, ``data`` is an encoded URL. You can override this me
thod to
        provide your own message decoding.

        Parameters
        -----
        data : bytes
            Message from redis.

        """
        url = bytes_to_str(data, self.redis_encoding)
        return self.make_requests_from_url(url)

    def schedule_next_requests(self):
        """Schedules a request if available"""
        # TODO: While there is capacity, schedule a batch of redis reque
sts.
        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.

    Attributes
    -----
    redis_key : str (default: REDIS_START_URLS_KEY)
        Redis key where to fetch start URLs from..
    redis_batch_size : int (default: CONCURRENT_REQUESTS)
        Number of messages to fetch from redis on each attempt.
    redis_encoding : str (default: REDIS_ENCODING)
        Encoding to use when decoding messages from redis queue.

    Settings
    -----
    REDIS_START_URLS_KEY : str (default: "<spider.name>:start_urls")
        Default Redis key where to fetch start URLs from..
    REDIS_START_URLS_BATCH_SIZE : int (deprecated by CONCURRENT_REQUESTS
    )
        Default number of messages to fetch from redis on each attempt.
    REDIS_START_URLS_AS_SET : bool (default: False)
        Use SET operations to retrieve messages from the redis queue. If
    False,
        the messages are retrieve using the LPOP command.
    REDIS_ENCODING : str (default: "utf-8")
        Default encoding to use when decoding messages from redis queue.

    """

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

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

    Attributes
    -----
    redis_key : str (default: REDIS_START_URLS_KEY)
        Redis key where to fetch start URLs from..
    redis_batch_size : int (default: CONCURRENT_REQUESTS)
        Number of messages to fetch from redis on each attempt.
    redis_encoding : str (default: REDIS_ENCODING)
        Encoding to use when decoding messages from redis queue.

    Settings
    -----
    REDIS_START_URLS_KEY : str (default: "<spider.name>:start_urls")
        Default Redis key where to fetch start URLs from..
    REDIS_START_URLS_BATCH_SIZE : int (deprecated by CONCURRENT_REQUESTS
    )
        Default number of messages to fetch from redis on each attempt.
    REDIS_START_URLS_AS_SET : bool (default: True)
        Use SET operations to retrieve messages from the redis queue.

```

```

    REDIS_ENCODING : str (default: "utf-8")
        Default encoding to use when decoding messages from redis queue.

    """

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

```

总结

总结一下scrapy-redis的总体思路：

1. 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；
2. 实现新的dupefilter和queue类，达到了判重和调度容器和redis的交互；
3. 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；
4. 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；
5. 每当一个spider产出一个request的时候，scrapy内核会把这个request递交给这个spider对应的scheduler对象进行调度；
6. 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；
7. 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；
8. 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；
9. spider收到这个信号之后，连接redis读取strart url池，拿去新的一批url入口；
10. 再次重复上文1-9。