网络数据采集

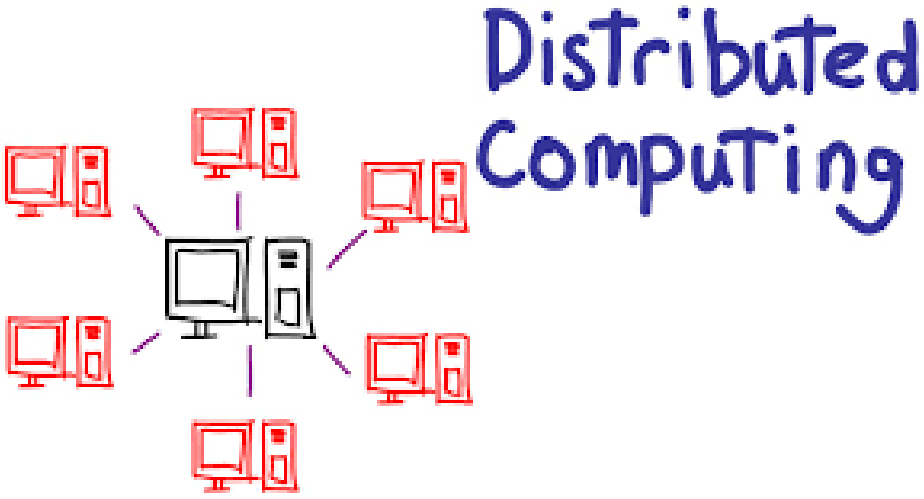
第7章 构建分布式爬虫系统

- 讲师姓名：
- 授课时间：
- 共32课时，第28-32课时

1 课前引导

大家知道，当今以大数据、云计算、物联网为支撑技术的第三次数字化浪潮已经席卷了全球各个产业。网络上各类信息的产生速度和数据容量不断攀升，传统的爬虫程序在采集速度和存储容量上存在瓶颈，无法满足大数据时代的数据采集需求。

如何解决这一问题呢？



可以这样考虑，既然为了解决海量数据的计算和存储诞生了分布式计算和分布式存储技术，那么能否参考这些技术设计网络爬虫程序呢？

答案是肯定的。

这一讲，我们将介绍使用Scrapy-redis模块，扩展scrapy爬虫框架，构建分布式网络爬虫的方法。

2 本节课程主要内容

内容列表：

- 本节目标
- 分布式爬虫系统概述
- Scrapy-redis介绍
- 使用Scrapy-redis构建分布式爬虫系统
- 本节总结
- 课后练习

2.1 分布式爬虫系统概述

首先介绍分布式爬虫系统概述。

通用的分布式系统，是指一组计算机，透过网络相互连接传递消息与通信后并协调它们的行为而形成的系统。组件之间彼此进行交互以实现一个共同的目标。这个系统将需要进行大量计算的任务数据分割成

小块，由多台计算机分别计算，再上传运算结果后，将结果统一合并得出最终结果。

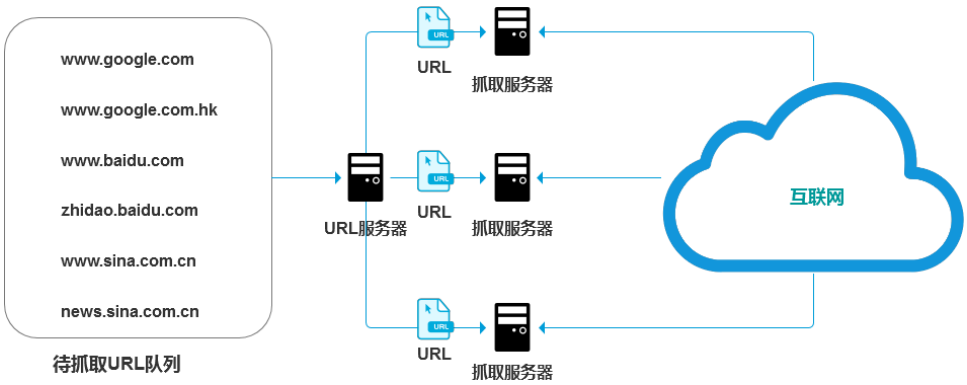
分布式爬虫系统是用于完成数据采集任务的分布式爬虫系统。根据不同机器之间的协同方式不同，分布式爬虫架构可分为两种：

- 第一种是主从式分布爬虫；
- 第二种是对等式分布爬虫。

2.1.1 主从式分布爬虫

在主从式分布爬虫的架构中，不同的服务器承担不同的角色分工，其中有一台主服务器（也称URL服务器）专门负责待爬取URL的分发，其他服务器作为从服务器负责实际网页访问和资源下载。

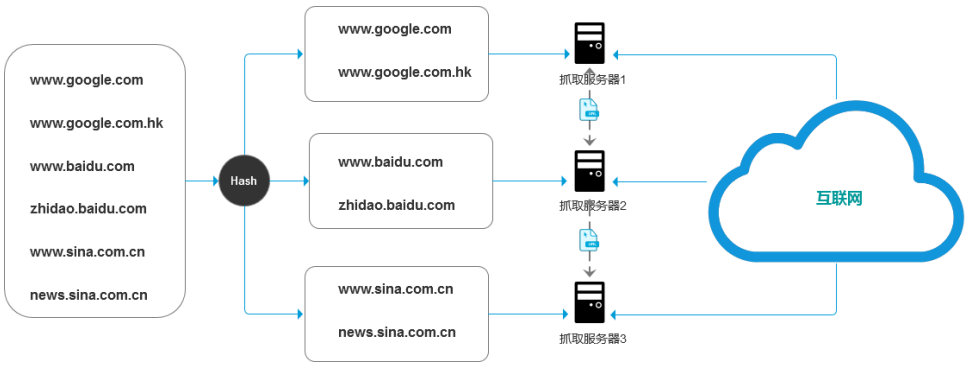
主服务器不仅维护着待爬取URL队列，还担负着从服务器的负载均衡任务，确保各个从服务器的负荷大致相同。其架构如下图所示：



Google在早期即采用此种主从分布式爬虫，在这种架构中，因为URL服务器承担很多管理任务，同时待抓取URL队列数量巨大，所以URL服务器容易成为整个系统的瓶颈。

2.1.2 对等式分布爬虫

在对等式分布爬虫架构中，服务器之间不存在分工差异，每台服务器承担相同的功能，各自负担一部分URL的抓取工作，下图是其中一种对等式分布爬虫，Mercator爬虫采用此种体系结构。



由于没有主服务器存在，服务器间的任务分工就成为共同完成的任务，每台服务器都会根据一定的算法（例如基于hash的某种分支规则），自己判断某个URL是否应该由自己来抓取，或者将这个URL传递给相应的服务器。

2.1.3 模块练习与答案

见习题集

2.1.4 内容小结

这一节主要介绍了分布式系统的基本概念和工作原理。

2.2 Scrapy-redis介绍

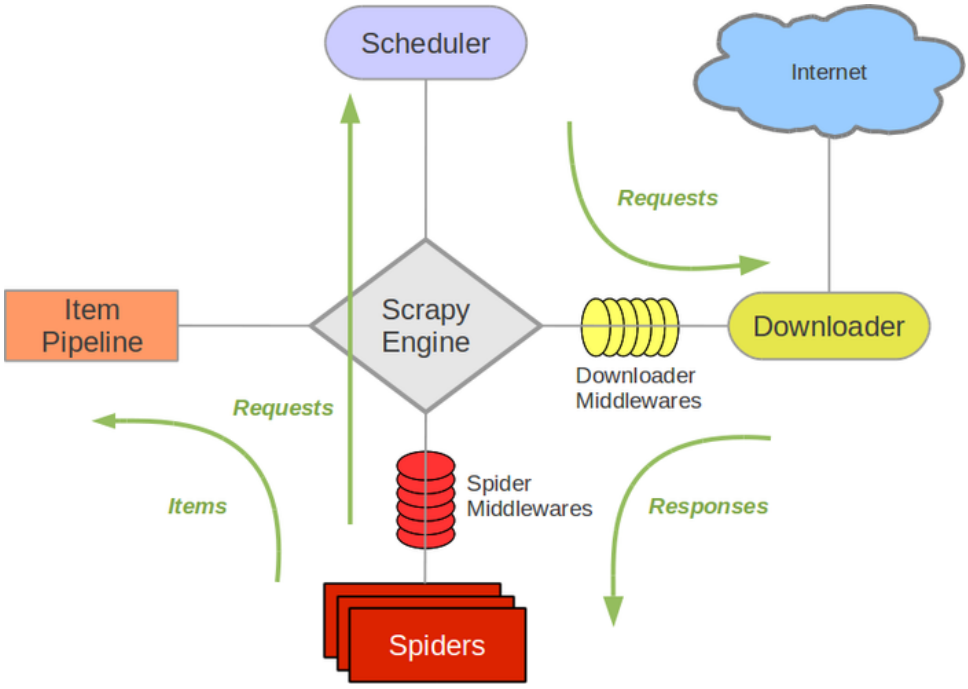
以上，我们介绍了分布式爬虫的概念和基本架构。在实际构建分布式爬虫时，人们往往会借助下列方法来完成任务排队和调度：

- 高性能的键值数据库
- 消息队列
- 非阻塞I/O调度机制

分布式爬虫的调度系统非常重要，是构建分布式爬虫系统的基础。

下面，我们将重点介绍以redis数据库为核心的分布式爬虫模块Scrapy-redis，并使用它构建我们自己的分布式爬虫系统。

在介绍Scrapy-redis之前，我们先回顾scrapy爬虫框架:



- Scrapy中有一个本地爬取队列Queue，这个队列是利用deque模块实现的。
- 如果有新的URL请求生成就会放到队列里面，随后URL请求被Scheduler调度，交给Downloader执行爬取。

- 当待爬取队列中的URL极多时，为了提高爬虫的性能，我们需要多个的schedular从队列中取出URL，同时也需要有多个downloader去指向URL访问。这时队列的I/O性能必须足够好，才能满足多台服务器中scrapy爬虫读、写URL的需求。

所以人们很自然的就会想到**使用高性能的内存数据库Redis作为存储URL的“队列”。**

2.2.1 Redis

Redis是一种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 官网：https://redis.io/

Redis 在线测试：http://try.redis.io/

Redis支持多种数据结构，例如：

- String: 字符串
- Hash: 散列
- List: 列表
- Set: 集合
- Sorted Set: 有序集合

Redis 优势：

- 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。
- 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

Redis与其他key-value存储有什么不同？

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。

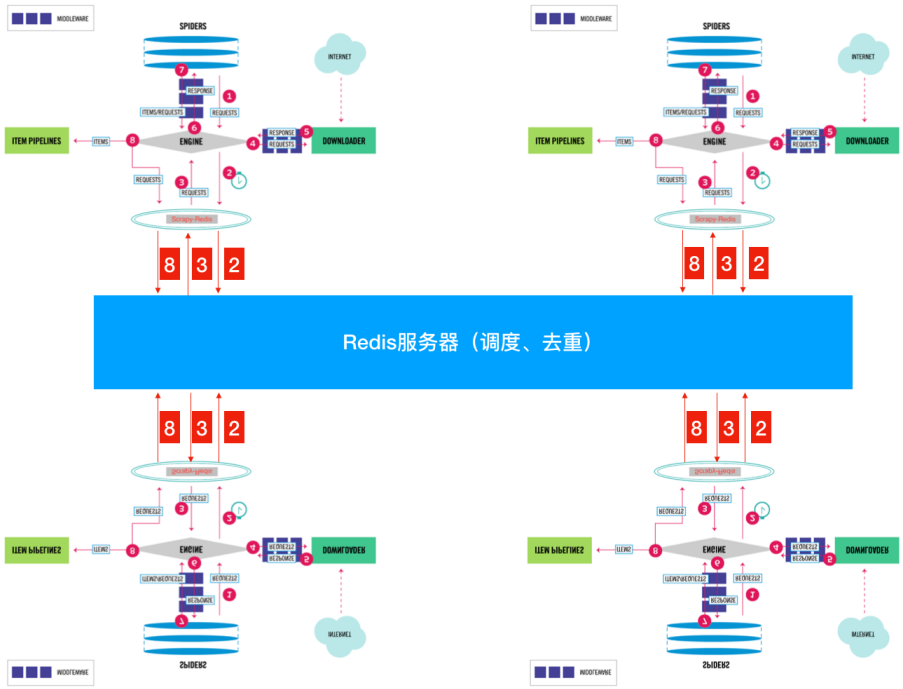
Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

Scrapy-redis

Redis的存取操作非常简单，有利于实现去重和满足一定存取规则的URL队列。另外使用数据库存放URL，还可以防止爬虫意外中断后的继续爬取。

Scrapy-redis模块就是基于Redis和scrapy的一个分布式爬虫支持模块。它可以帮助我们轻松实现分布式爬虫系统。

Scrapy-redis的架构如下图所示：



不难看出，Scrapy-redis是在原有scrapy架构的基础上，增加了Redis组件。它取代Scrapy queue作为待爬取URL的请求队列，这一改变使得多个scrapy spider能够同时获取同一个URL队列中的待爬取请求。

为了从Redis中读取待爬取的请求，Scrapy-redis模块改变了原有scrapy中的schedular和spdier，使其能够直接读取redis数据。同时还增加了redis中的请求去重功能。

我们可以看一下scrapy中的schedule源代码：

```
class Scheduler(object):

    def __init__(self, dupefilter, jobdir=None, dqclass=None, mqclass=None,
                 logunser=False, stats=None, pqclass=None):
        self.df = dupefilter
        self.dqdir = self._dqdir(jobdir)
        self.pqclass = pqclass
        self.dqclass = dqclass
        self.mqclass = mqclass
        self.logunser = logunser
        self.stats = stats
        # 注意在scrpy中优先注意这个方法，此方法是一个钩子 用于访问当前爬虫的配置
    @classmethod
    def from_crawler(cls, crawler):
        settings = crawler.settings
        # 获取去重用的类 默认：scrapy.dupefilters.RFPDupeFilter
        dupefilter_cls = load_object(settings['DUPEFILTER_CLASS'])
        # 对去重类进行配置from_settings 在 scrapy.dupefilters.RFPDupeFilter 43行
        # 这种调用方式对于IDE跳转不是很好  所以需要自己去找
        # @classmethod
        # def from_settings(cls, settings):
        #     debug = settings.getbool('DUPEFILTER_DEBUG')
        #     return cls(job_dir(settings), debug)
        # 上面就是from_settings方法 其实就是设置工作目录 和是否开启debug
        dupefilter = dupefilter_cls.from_settings(settings)
        # 获取优先级队列 类对象 默认：queuelib.pqueue.PriorityQueue
        pqclass = load_object(settings['SCHEDULER_PRIORITY_QUEUE'])
        # 获取磁盘队列 类对象（SCHEDULER使用磁盘存储 重启不会丢失）
        dqclass = load_object(settings['SCHEDULER_DISK_QUEUE'])
        # 获取内存队列 类对象（SCHEDULER使用内存存储 重启会丢失）
        mqclass = load_object(settings['SCHEDULER_MEMORY_QUEUE'])
        # 是否开启debug
        logunser = settings.getbool('LOG_UNSERIALIZABLE_REQUESTS', settings.getbool('SCHEDULER_D
        # 将这些参数传递给 __init__方法
        return cls(dupefilter, jobdir=job_dir(settings), logunser=logunser,
                   stats=crawler.stats, pqclass=pqclass, dqclass=dqclass, mqclass=mqclass)


    def has_pending_requests(self):
      """检查是否有没处理的请求"""
        return len(self) > 0

    def open(self, spider):
      """Engine创建完毕之后会调用这个方法"""
        self.spider = spider
        # 创建一个有优先级的内存队列 实例化对象
        # self.pqclass 默认是：queuelib.pqueue.PriorityQueue
        # self._newmq 会返回一个内存队列的 实例化对象 在110  111 行
        self.mqs = self.pqclass(self._newmq)
        # 如果self.dqdir 有设置 就创建一个磁盘队列 否则self.dqs 为空
        self.dqs = self._dq() if self.dqdir else None
```

```
    # 获得一个去重实例对象 open 方法是从BaseDupeFilter继承的
    # 现在我们可以用self.df来去重啦
    return self.df.open()

def close(self, reason):
  """当然Engine关闭时"""
      # 如果有磁盘队列 则对其进行dump后保存到active.json文件中
    if self.dqs:
        prios = self.dqs.close()
        with open(join(self.dqdir, 'active.json'), 'w') as f:
            json.dump(prios, f)
    # 然后关闭去重
    return self.df.close(reason)

def enqueue_request(self, request):
  """添加一个Requests进调度队列"""
      # self.df.request_seen是检查这个Request是否已经请求过了 如果有会返回True
    if not request.dont_filter and self.df.request_seen(request):
          # 如果Request的dont_filter属性没有设置（默认为False）和 已经存在则去重
        # 不push进队列
        self.df.log(request, self.spider)
        return False
    # 先尝试将Request push进磁盘队列
    dqok = self._dqpush(request)
    if dqok:
          # 如果成功 则在记录一次状态
        self.stats.inc_value('scheduler/enqueued/disk', spider=self.spider)
    else:
          # 不能添加进磁盘队列则会添加进内存队列
        self._mqpush(request)
        self.stats.inc_value('scheduler/enqueued/memory', spider=self.spider)
    self.stats.inc_value('scheduler/enqueued', spider=self.spider)
    return True

def next_request(self):
  """从队列中获取一个Request"""
      # 优先从内存队列中获取
    request = self.mqs.pop()
    if request:
        self.stats.inc_value('scheduler/dequeued/memory', spider=self.spider)
    else:
          # 不能获取的时候从磁盘队列队里获取
        request = self._dqpop()
        if request:
            self.stats.inc_value('scheduler/dequeued/disk', spider=self.spider)
    if request:
        self.stats.inc_value('scheduler/dequeued', spider=self.spider)
    # 将获取的到Request返回给Engine
    return request

def __len__(self):
```

```
    return len(self.dqs) + len(self.mqs) if self.dqs else len(self.mqs)

def _dqpush(self, request):
    if self.dqs is None:
        return
    try:
        reqd = request_to_dict(request, self.spider)
        self.dqs.push(reqd, -request.priority)
    except ValueError as e:  # non serializable request
        if self.logunser:
            msg = ("Unable to serialize request: %(request)s - reason:"
                   " %(reason)s - no more unserializable requests will be"
                   " logged (stats being collected)")
            logger.warning(msg, {'request': request, 'reason': e},
                           exc_info=True, extra={'spider': self.spider})
            self.logunser = False
        self.stats.inc_value('scheduler/unserializable',
                             spider=self.spider)
        return
    else:
        return True

def _mqpush(self, request):
    self.mqs.push(request, -request.priority)

def _dqpop(self):
    if self.dqs:
        d = self.dqs.pop()
        if d:
            return request_from_dict(d, self.spider)

def _newmq(self, priority):
    return self.mqclass()

def _newdq(self, priority):
    return self.dqclass(join(self.dqdir, 'p%s' % priority))

def _dq(self):
    activef = join(self.dqdir, 'active.json')
    if exists(activef):
        with open(activef) as f:
            prios = json.load(f)
    else:
        prios = ()
    q = self.pqclass(self._newdq, startprios=prios)
    if q:
        logger.info("Resuming crawl (%(queuesize)d requests scheduled)",
                    {'queuesize': len(q)}, extra={'spider': self.spider})
    return q

def _dqdir(self, jobdir):
```

```
        if jobdir:
            dqdir = join(jobdir, 'requests.queue')
            if not exists(dqdir):
                os.makedirs(dqdir)
            return dqdir
```

从上面的代码 我们可以很清楚的知道 SCHEDULER的主要是完成了 push Request pop Request 和 去重的操作。而且queue 操作是在内存队列中完成的。大家看queuelib.queue就会发现基于内存的（deque）。那么去重呢？

```
class RFPDupeFilter(BaseDupeFilter):
    """Request Fingerprint duplicates filter"""

    def __init__(self, path=None, debug=False):
        self.file = None
        self.fingerprints = set()
        self.logdupes = True
        self.debug = debug
        self.logger = logging.getLogger(__name__)
        if path:
            # 此处可以看到去重其实打开了一个名叫 requests.seen的文件
            # 如果是使用的磁盘的话
            self.file = open(os.path.join(path, 'requests.seen'), 'a+')
            self.file.seek(0)
            self.fingerprints.update(x.rstrip() for x in self.file)

    @classmethod
    def from_settings(cls, settings):
        debug = settings.getbool('DUPEFILTER_DEBUG')
        return cls(job_dir(settings), debug)

    def request_seen(self, request):
        fp = self.request_fingerprint(request)
        if fp in self.fingerprints:
              # 判断我们的请求是否在这个在集合中
            return True
        # 没有在集合就添加进去
        self.fingerprints.add(fp)
        # 如果用的磁盘队列就写进去记录一下
        if self.file:
            self.file.write(fp + os.linesep)
```

按照正常流程，如果有多个scrapy一起工作，那么它们都会进行重复的采集，然而各scrapy进程之间内存中的数据不可共享，所以开启的多个Scrapy相互之间不能有效协同。

为了解决这个问题，Scrapy-redis使用了redis作为存储请求的“队列”。下面我们来看看Scrapy-Redis是怎么处理的?

```
#scrapy_redis.scheduler.py:

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_DUPEFILTER_KEY : str
        Scheduler dupefilter redis key.
    SCHEDULER_DUPEFILTER_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
            这是Redis实例
        persist : bool
            是否在关闭时清空Requests.默认值是False。
        flush_on_start : bool
            是否在启动时清空Requests。 默认值是False。
        queue_key : str
            Request队列的Key名字
        queue_cls : str
            队列的可导入路径（就是使用什么队列）
        dupefilter_key : str
            去重队列的Key
```

```
    dupefilter_cls : str
        去重类的可导入路径。
    idle_before_close : int
        等待多久关闭

    """
    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 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',
    }
    # 从setting中获取配置组装成dict(具体获取那些配置是optional字典中key)
    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'])
```

```
            # 或得一个Redis连接
        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:
              # 根据self.queue_cls这个可以导入的类 实例化一个队列
            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:
              # 根据self.dupefilter_cls这个可以导入的类 实例一个去重集合
            # 默认是集合 可以实现自己的去重方式 比如 bool 去重
            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 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):
      """这个和Scrapy本身的一样"""
        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)
        # 向队列里面添加一个Request
        self.queue.push(request)
        return True

    def next_request(self):
      """获取一个Request"""
        block_pop_timeout = self.idle_before_close
        # block_pop_timeout 是一个等待参数 队列没有东西会等待这个时间  超时就会关闭
        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
```

下面我们来看看queue和本身的什么不同：

scrapy_redis.queue.py(以最常用的优先级队列 PriorityQueue 为例）

```
class PriorityQueue(Base):
    """Per-spider priority queue abstraction using redis' sorted set"""
        """其实就是使用Redis的有序集合 来对Request进行排序，这样就可以优先级高的在有序集合的顶层 我们
    """从上往下依次获取Request即可"""
    def __len__(self):
        """Return the length of the queue"""
        return self.server.zcard(self.key)

    def push(self, request):
        """Push a request"""
        """添加一个Request进队列"""
        # self._encode_request 将Request请求进行序列化
        data = self._encode_request(request)
        """
        d = {
        'url': to_unicode(request.url),  # urls should be safe (safe_string_url)
        'callback': cb,
        'errback': eb,
        'method': request.method,
        'headers': dict(request.headers),
        'body': request.body,
        'cookies': request.cookies,
        'meta': request.meta,
        '_encoding': request._encoding,
        'priority': request.priority,
        'dont_filter': request.dont_filter,
        'flags': request.flags,
        '_class': request.__module__ + '.' + request.__class__.__name__
            }

        data就是上面这个字典的序列化
        在Scrapy.utils.reqser.py 中的request_to_dict方法中处理
        """

        # 在Redis有序集合中数值越小优先级越高(就是会被放在顶层)所以这个位置是取得 相反数
        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.
        # ZADD 是添加进有序集合
        self.server.execute_command('ZADD', self.key, score, data)

    def pop(self, timeout=0):
        """
        Pop a request
        timeout not support in this queue class
        有序集合不支持超时所以就木有使用timeout了
        这个timeout就是挂羊头卖狗肉
        """
        """从有序集合中取出一个Request"""
        # use atomic range/remove using multi/exec
```

```
        """使用multi的原因是为了将获取Request和删除Request合并成一个操作(原子性的)在获取到一个元素之
        pipe = self.server.pipeline()
        pipe.multi()
        # 取出 顶层第一个
        # zrange :返回有序集 key 中，指定区间内的成员。0,0 就是第一个了
        # zremrangebyrank：移除有序集 key 中，指定排名(rank)区间内的所有成员 0，0也就是第一个了
        # 更多请参考Redis官方文档
        pipe.zrange(self.key, 0, 0).zremrangebyrank(self.key, 0, 0)
        results, count = pipe.execute()
        if results:
            return self._decode_request(results[0])
```

以上就是SCHEDULER在处理Request的时候做的操作了。

是时候来看看SCHEDULER是怎么处理去重的了！

只需要注意这个?方法即可：

```
 def request_seen(self, request):
   """Returns True if request was already seen.

        Parameters
        ----------
        request : scrapy.http.Request

        Returns
        -------
        bool

        """
   # 通过self.request_fingerprint 会生一个sha1的指纹
   fp = self.request_fingerprint(request)
   # This returns the number of values added, zero if already exists.
   # 添加进一个Redis集合如果self.key这个集合中存在fp这个指纹会返回1  不存在返回0
   added = self.server.sadd(self.key, fp)
   return added == 0
```

这样大家就都可以访问同一个Redis 获取同一个spider的Request 在同一个位置去重，就不用担心重复啦

2.3 Scrapy-redis 安装与使用

要使用Scrapy-redis需要安装Redis数据库、Scrapy-redis模块。

下面我们将通过实际操作演示，进行详细介绍。

2.3.1 安装 Redis

windows 下安装 redis

Redis Windows安装版从 https://github.com/microsoftarchive/redis/releases 下载。

可以选择压缩包或安装包，例如：

Redis-x64-3.0.504.msi 或 Redis-x64-3.0.504.zip。

Ubuntu Linux 下安装 redis

安装之前要到 https://redis.io/ 下载Redis安装文件。

在 Ubuntu 系统安装 Redis ，可以使用以下命令:

```
sudo apt-get update

sudo apt-get install redis-server
```

Docker 下安装 redis

随着docker容器技术的普及，在服务器的docker容器中使用redis也是一个不错的选择。如果你已经安装了docker程序，并能够运行，可以运行下列命令pull 一个redis docker 镜像：

```
$ docker pull redis:latest

# pull 完成后，查看是否已经下载完成。
$ docker image list
```

2.3.2 启动 Redis

启动redis命令为：

- 先进入redis安装目录，默认是program files/redis
- 运行下列命令：

```
redis-server redis.windows.conf
```

在ubuntu linux下，启动redis 命令为：redis-server

之后，运行下列命令会进入redis工作环境下：

```
redis-cli
```

执行后将打开以下终端：

```
redis 127.0.0.1:6379>
```

以上说明我们已经成功安装了redis。

2.3.3 在远程服务上执行命令

如果需要在远程 redis 服务上执行命令，同样我们使用的也是 redis-cli 命令。

```
redis-cli.exe -h 127.0.0.1 -p 6379
```

以下实例演示了如何连接到主机为 127.0.0.1，端口为 6379 ，密码为 mypass 的 redis 服务上。

```
$redis-cli -h 127.0.0.1 -p 6379 -a "mypass"
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> PING

PONG
```

2.3.4 运行docker 中的redis

```
#  启动镜像用下列命令运行
<p class="mume-header " id="启动镜像用下列命令运行"></p>

docker run --rm --name redis -p 6379:6379 -d redis:latest redis-server --appendonly yes

#  若要进入redis 容器内部查看，可以运行下列命令：
<p class="mume-header " id="若要进入redis-容器内部查看可以运行下列命令"></p>

docker container

#  以下为容器查询结果：
<p class="mume-header " id="以下为容器查询结果"></p>

CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS              PORTS
2d4800f52c87        redis:latest        "docker-entrypoint.s…"   About a minute ago   Up About a minute   0.0.0.0

#  使用exec命令进入docker容器
<p class="mume-header " id="使用exec命令进入docker容器"></p>

[leo@localhost ~]$ sudo docker exec -it 2d4800f52c87 bash

#  运行控制台并尝试添加数据
<p class="mume-header " id="运行控制台并尝试添加数据"></p>

root@2d4800f52c87:/data# redis-cli
127.0.0.1:6379> set test 1
OK
127.0.0.1:6379>
```

命令说明：

- -p 6379:6379 : 将容器的6379端口映射到主机的6379端口
- -v ~/docker/redis/data:/data : 将主机中当前目录下的data挂载到容器的/data
- redis-server --appendonly yes : 在容器执行redis-server启动命令，并打开redis持久化配置

如果需要挂载配置文件和数据文件，需要预先创建文件目录和文件，例如：

```
#  在docker宿主机中建立目录或文件
<p class="mume-header " id="在docker宿主机中建立目录或文件"></p>


mkdir  ~/docker/redis/conf/redis.conf，内容可为空
mkdir  ~/docker/redis/data/

#  使用下列命令启动redis docker容器
<p class="mume-header " id="使用下列命令启动redis-docker容器"></p>

docker run --rm  --privileged=true --name redis -v ~/docker/redis/conf/redis.conf:/etc/redis/rec

#  进入redis 容器
<p class="mume-header " id="进入redis-容器"></p>

docker exec -it 容器id
```

参数说明：

- --privileged=true：容器内的root拥有真正root权限，否则容器内root只是外部普通用户权限
- redis-server /etc/redis/redis.conf：指定配置文件启动redis-server进程

如果提示有“Warnning：IPv4 forwarding is disabled”

可以进行下列操作：

```
# 可以编辑文件systemctl.conf:
<p class="mume-header " id="可以编辑文件systemctlconf"></p>

sudo vi /etc/sysctl.conf
# 在/etc/sysctl.conf中追加一行内容:
<p class="mume-header " id="在etcsysctlconf中追加一行内容"></p>

net.ipv4.ip_forward=1
# 重启服务:
<p class="mume-header " id="重启服务"></p>

sudo systemctl restart network
# 查看转发状态
<p class="mume-header " id="查看转发状态"></p>

sysctl net.ipv4.ip_forward
```

如果要在centos等linux中启动某个端口，需要设置防火墙规则。

例如：

```
sudo firewall-cmd --add-port=6379/tcp --permanent
```

如果启动了redis 可以查看防火墙规则： `sudo firewall-cmd --list-ports` ，显示端口列表。

之后，进入新安装目录下的redis.conf文件里，把protected-mode yes中的yes改成no。

再之后，将redis.conf文件中的 `bind 127.0.0.1` ，给注释掉(代表不做限制)，这样外网就能访问了。

启动 `redis-cli` （全路径名）输入： `set requirepass` （你的密码）

还可以在redis.conf中修改密码，找到 #requirepass foobared 若需要设置密码就把注释打开，改成你要设置的密码。之后就是外部工具启动连接了。

如果DNS受限

运行下列配置：

```
#显示当前网络连接
#nmcli connection show
NAME UUID                                  TYPE           DEVICE
eno1 5fb06bd0-0bb0-7ffb-45f1-d6edd65f3e03 802-3-ethernet eno1

#修改当前网络连接对应的DNS服务器，这里的网络连接可以用名称或者UUID来标识
#nmcli con mod eno1 ipv4.dns "114.114.114.114 8.8.8.8"

#将dns配置生效
#nmcli con up eno1
```

2.3.5 尝试存储键值

Redis 键命令用于管理 redis 的键。Redis 键命令的基本语法如下：

```
redis 127.0.0.1:6379> COMMAND KEY_NAME
```

```
set mykey abc
#
<p class="mume-header " id=""></p>

get mykey
# "abc"
<p class="mume-header " id="abc"></p>


#  查看所有键值命令
<p class="mume-header " id="查看所有键值命令"></p>

keys *

#  列出匹配apple*的key
<p class="mume-header " id="列出匹配apple的key"></p>


redis>keys apple*


#  查看数据库键值数量命令
<p class="mume-header " id="查看数据库键值数量命令"></p>

dbsize

#  查看当前redis的配置信息
<p class="mume-header " id="查看当前redis的配置信息"></p>

CONFIG GET *

#  清除redis缓存
<p class="mume-header " id="清除redis缓存"></p>

flushall

#  删除redis当前数据库中的所有KEY
<p class="mume-header " id="删除redis当前数据库中的所有key"></p>

flushdb

#  删除redis所有数据库中的KEY
<p class="mume-header " id="删除redis所有数据库中的key"></p>

flushall
```

与 Redis 键相关的基本命令可以参考：https://www.runoob.com/redis/redis-keys.html

2.3.6 安装Redis Desktop Manager工具（可选）

这个工具可以令我们更容易观察redis中的key-value变化。

Windows下安装运行步骤如下：

- 安装 Microsoft Visual C++ 2017 x64 (If you have not already)
- 下载 Windows Installer from http://redisdesktop.com/download (Requires subscription)
- Run downloaded installer

2.3.7 模块练习与答案

见习题集

2.3.8 内容小结

本节介绍了使用scrapy-redis 搭建分布式爬虫需要的准备工作，主要有以下几步：安装 Redis、启动 Redis、运行控制台命令、安装Redis Desktop Manager工具。

2.4 使用Scrapy-redis构建分布式爬虫系统

下面我们将按步骤创建一个简单的 Scrapy-redis爬虫。

主要步骤包括：

- 安装 scrapy-redis 组件
- 启动 redis 数据库
- 建立 Scrapy-redis 虚拟环境
- 建立 Scrapy-redis 项目
- 编辑 settings.py 文件
- 编写 spider 文件
- 运行爬虫
- 设置redis键 myspider:start_urls
- 查看结果

2.4.1 安装 scrapy-redis 组件

打开Anaconda3 命令行，建立一个虚拟环境 scrapyredisws 并安装scrapy-redis，命令如下：

```
(base) C:\Users\xxx>conda create -n scrapyredisws scrapy-redis

# 如果出现No module named win32api，需要运行下列命令
<p class="mume-header " id="如果出现no-module-named-win32api需要运行下列命令"></p>


pip install pypiwin32
```

上面我们通过实际操作，介绍了redis-scrpy模块的安装和测试。下面我们将介绍如何使用它来构建自己的分布式爬虫系统。

我们将构建项目tutorial，使其能够从redis中读取url，并将结果返回给redis，以键值列表形式进行存储。

2.4.2 启动redis数据库

```
cd c:\program files\redis\
#  启动服务
<p class="mume-header " id="启动服务"></p>

redis-server redis.windows.conf
#  进入控制台
<p class="mume-header " id="进入控制台"></p>

redis-cli
#  查看所有keys
<p class="mume-header " id="查看所有keys"></p>

keys *
#  查看数据大小
<p class="mume-header " id="查看数据大小"></p>

dbsize

#  将redis-server 安装为服务，防止关闭窗口后，server关闭。
<p class="mume-header " id="将redis-server-安装为服务防止关闭窗口后server关闭"></p>

redis-server --service-install redis.windows-service.conf --loglevel verbose ( 安装redis服务 )

# 输入命令启动服务器：
<p class="mume-header " id="输入命令启动服务器"></p>

redis-server --service-start ( 启动服务 )

# 若需要关闭redis 则输入命令：
<p class="mume-header " id="若需要关闭redis-则输入命令"></p>

redis-server --service-stop （停止服务）
```

2.4.3 建立Scrapy-redis虚拟环境

在已安装Anaconda3的环境下，运行下列命令，生成、激活 scrapyredisws 这个虚拟目录，之后进入这个目录。

```
(base) C:\Users\leo>conda activate scrapyredisws

(scrapyredisws) C:\Users\leo>cd Anaconda3\envs\scrapyredisws

(scrapyredisws) C:\Users\leo\Anaconda3\envs\scrapyredisws>
```

在Linux服务器下，可以直接安装python的官方docker。

```
sudo docker pull python:latest

[leo@localhost ~]$ docker run -it --rm python bash


root@3c01dec9369d:/# python -V
Python 3.8.0
#  安装虚拟环境工具
<p class="mume-header " id="安装虚拟环境工具"></p>

root@3c01dec9369d:/# pip install  virtualenv

#  创建一个虚拟环境
<p class="mume-header " id="创建一个虚拟环境"></p>

root@3c01dec9369d:/# virtualenv scrapy_redis
Using base prefix '/usr/local'
New python executable in /scrapy_redis/bin/python
Installing setuptools, pip, wheel...
done.
#  激活虚拟环境
<p class="mume-header " id="激活虚拟环境"></p>

root@3c01dec9369d:/# cd scrapy_redis/
root@3c01dec9369d:/scrapy_redis# source bin/activate

#  安装scrapy-redis
<p class="mume-header " id="安装scrapy-redis"></p>

(scrapy_redis) root@3c01dec9369d:/scrapy_redis# pip install scrapy-redis
```

2.4.4 建立 Scrapy-redis 项目

运行下列命令建立项目，项目名称为： tutorial

```
scrapy startproject tutorial
```

我们会发现，scrapy-redis下生成的目录结构与scrapy 项目相同。

2.4.5 编辑 settings.py 文件

内容如下:

```
#  -*- coding: utf-8 -*-
<p class="mume-header " id="-coding-utf-8-"></p>


#  Scrapy settings for tutorial project
<p class="mume-header " id="scrapy-settings-for-tutorial-project"></p>

#
<p class="mume-header " id="-1"></p>

#  For simplicity, this file contains only settings considered important or
<p class="mume-header " id="for-simplicity-this-file-contains-only-settings-considered-important

#  commonly used. You can find more settings consulting the documentation:
<p class="mume-header " id="commonly-used-you-can-find-more-settings-consulting-the-documentatic

#
<p class="mume-header " id="-2"></p>

#      https://docs.scrapy.org/en/latest/topics/settings.html
<p class="mume-header " id="httpsdocsscrapyorgenlatesttopicssettingshtml"></p>

#      https://docs.scrapy.org/en/latest/topics/downloader-middleware.html
<p class="mume-header " id="httpsdocsscrapyorgenlatesttopicsdownloader-middlewarehtml"></p>

#      https://docs.scrapy.org/en/latest/topics/spider-middleware.html
<p class="mume-header " id="httpsdocsscrapyorgenlatesttopicsspider-middlewarehtml"></p>


BOT_NAME = 'tutorial'

SPIDER_MODULES = ['tutorial.spiders']
NEWSPIDER_MODULE = 'tutorial.spiders'


#  Crawl responsibly by identifying yourself (and your website) on the user-agent
<p class="mume-header " id="crawl-responsibly-by-identifying-yourself-and-your-website-on-the-us

# USER_AGENT = 'tutorial (+http://www.yourdomain.com)'
<p class="mume-header " id="user_agent-tutorial-httpwwwyourdomaincom"></p>


#  Obey robots.txt rules
<p class="mume-header " id="obey-robotstxt-rules"></p>

ROBOTSTXT_OBEY = True

#  Configure maximum concurrent requests performed by Scrapy (default: 16)
<p class="mume-header " id="configure-maximum-concurrent-requests-performed-by-scrapy-default-16

# CONCURRENT_REQUESTS = 32
```

Configure a delay for requests for the same website (default: 0)

See https://docs.scrapy.org/en/latest/topics/settings.html#download-delay

See also autothrottle settings and docs

DOWNLOAD_DELAY = 3

The download delay setting will honor only one of:

CONCURRENT_REQUESTS_PER_DOMAIN = 16

CONCURRENT_REQUESTS_PER_IP = 16

Disable cookies (enabled by default)

COOKIES_ENABLED = False

Disable Telnet Console (enabled by default)

TELNETCONSOLE_ENABLED = False

Override the default request headers:

DEFAULT_REQUEST_HEADERS = {

'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',

'Accept-Language': 'en',

```
# }
<p class="mume-header " id="-3"></p>


#  Enable or disable spider middlewares
<p class="mume-header " id="enable-or-disable-spider-middlewares"></p>

#  See https://docs.scrapy.org/en/latest/topics/spider-middleware.html
<p class="mume-header " id="see-httpsdocsscrapyorgenlatesttopicsspider-middlewarehtml"></p>

# SPIDER_MIDDLEWARES = {
<p class="mume-header " id="spider_middlewares"></p>

#     'tutorial.middlewares.TutorialSpiderMiddleware': 543,
<p class="mume-header " id="tutorialmiddlewarestutorialspidermiddleware-543"></p>

# }
<p class="mume-header " id="-4"></p>


#  Enable or disable downloader middlewares
<p class="mume-header " id="enable-or-disable-downloader-middlewares"></p>

#  See https://docs.scrapy.org/en/latest/topics/downloader-middleware.html
<p class="mume-header " id="see-httpsdocsscrapyorgenlatesttopicsdownloader-middlewarehtml"></p>

# DOWNLOADER_MIDDLEWARES = {
<p class="mume-header " id="downloader_middlewares"></p>

#     'tutorial.middlewares.TutorialDownloaderMiddleware': 543,
<p class="mume-header " id="tutorialmiddlewarestutorialdownloadermiddleware-543"></p>

# }
<p class="mume-header " id="-5"></p>


#  Enable or disable extensions
<p class="mume-header " id="enable-or-disable-extensions"></p>

#  See https://docs.scrapy.org/en/latest/topics/extensions.html
<p class="mume-header " id="see-httpsdocsscrapyorgenlatesttopicsextensionshtml"></p>

# EXTENSIONS = {
<p class="mume-header " id="extensions"></p>

#     'scrapy.extensions.telnet.TelnetConsole': None,
<p class="mume-header " id="scrapyextensionstelnettelnetconsole-none"></p>

# }
<p class="mume-header " id="-6"></p>
```

```
#  Configure item pipelines
<p class="mume-header " id="configure-item-pipelines"></p>

#  See https://docs.scrapy.org/en/latest/topics/item-pipeline.html
<p class="mume-header " id="see-httpsdocsscrapyorgenlatesttopicsitem-pipelinehtml"></p>

# ITEM_PIPELINES = {
<p class="mume-header " id="item_pipelines"></p>

#     'tutorial.pipelines.TutorialPipeline': 300,
<p class="mume-header " id="tutorialpipelinestutorialpipeline-300"></p>

# }
<p class="mume-header " id="-7"></p>


#  Enable and configure the AutoThrottle extension (disabled by default)
<p class="mume-header " id="enable-and-configure-the-autothrottle-extension-disabled-by-default"

#  See https://docs.scrapy.org/en/latest/topics/autothrottle.html
<p class="mume-header " id="see-httpsdocsscrapyorgenlatesttopicsautothrottlehtml"></p>

# AUTOTHROTTLE_ENABLED = True
<p class="mume-header " id="autothrottle_enabled-true"></p>

#  The initial download delay
<p class="mume-header " id="the-initial-download-delay"></p>

# AUTOTHROTTLE_START_DELAY = 5
<p class="mume-header " id="autothrottle_start_delay-5"></p>

#  The maximum download delay to be set in case of high latencies
<p class="mume-header " id="the-maximum-download-delay-to-be-set-in-case-of-high-latencies"></p>

# AUTOTHROTTLE_MAX_DELAY = 60
<p class="mume-header " id="autothrottle_max_delay-60"></p>

#  The average number of requests Scrapy should be sending in parallel to
<p class="mume-header " id="the-average-number-of-requests-scrapy-should-be-sending-in-parallel-

#  each remote server
<p class="mume-header " id="each-remote-server"></p>

# AUTOTHROTTLE_TARGET_CONCURRENCY = 1.0
<p class="mume-header " id="autothrottle_target_concurrency-10"></p>

#  Enable showing throttling stats for every response received:
<p class="mume-header " id="enable-showing-throttling-stats-for-every-response-received"></p>

# AUTOTHROTTLE_DEBUG = False
```

```
<p class="mume-header " id="autothrottle_debug-false"></p>


#  Enable and configure HTTP caching (disabled by default)
<p class="mume-header " id="enable-and-configure-http-caching-disabled-by-default"></p>

#  See https://docs.scrapy.org/en/latest/topics/downloader-middleware.html#httpcache-middleware-
<p class="mume-header " id="see-httpsdocsscrapyorgenlatesttopicsdownloader-middlewarehtmlhttpcac

# HTTPCACHE_ENABLED = True
<p class="mume-header " id="httpcache_enabled-true"></p>

# HTTPCACHE_EXPIRATION_SECS = 0
<p class="mume-header " id="httpcache_expiration_secs-0"></p>

# HTTPCACHE_DIR = 'httpcache'
<p class="mume-header " id="httpcache_dir-httpcache"></p>

# HTTPCACHE_IGNORE_HTTP_CODES = []
<p class="mume-header " id="httpcache_ignore_http_codes"></p>

# HTTPCACHE_STORAGE = 'scrapy.extensions.httpcache.FilesystemCacheStorage'
<p class="mume-header " id="httpcache_storage-scrapyextensionshttpcachefilesystemcachestorage"><


#  Enables scheduling storing requests queue in redis.
<p class="mume-header " id="enables-scheduling-storing-requests-queue-in-redis"></p>

SCHEDULER = "scrapy_redis.scheduler.Scheduler"

SCHEDULER_PERSIST = True
# SCHEDULER_QUEUE_CLASS = "scrapy_redis.queue.SpiderPriorityQueue"
<p class="mume-header " id="scheduler_queue_class-scrapy_redisqueuespiderpriorityqueue"></p>

# SCHEDULER_QUEUE_CLASS = "scrapy_redis.queue.SpiderQueue"
<p class="mume-header " id="scheduler_queue_class-scrapy_redisqueuespiderqueue"></p>

# SCHEDULER_QUEUE_CLASS = "scrapy_redis.queue.SpiderStack"
<p class="mume-header " id="scheduler_queue_class-scrapy_redisqueuespiderstack"></p>


#  Ensure all spiders share same duplicates filter through redis.
<p class="mume-header " id="ensure-all-spiders-share-same-duplicates-filter-through-redis"></p>

DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"

#  Default requests serializer is pickle, but it can be changed to any module
<p class="mume-header " id="default-requests-serializer-is-pickle-but-it-can-be-changed-to-any-m

#  with loads and dumps functions. Note that pickle is not compatible between
<p class="mume-header " id="with-loads-and-dumps-functions-note-that-pickle-is-not-compatible-be
```

```
#  python versions.
<p class="mume-header " id="python-versions"></p>

#  Caveat: In python 3.x, the serializer must return strings keys and support
<p class="mume-header " id="caveat-in-python-3x-the-serializer-must-return-strings-keys-and-supp

#  bytes as values. Because of this reason the json or msgpack module will not
<p class="mume-header " id="bytes-as-values-because-of-this-reason-the-json-or-msgpack-module-wi

#  work by default. In python 2.x there is no such issue and you can use
<p class="mume-header " id="work-by-default-in-python-2x-there-is-no-such-issue-and-you-can-use"

#  'json' or 'msgpack' as serializers.
<p class="mume-header " id="json-or-msgpack-as-serializers"></p>

# SCHEDULER_SERIALIZER = "scrapy_redis.picklecompat"
<p class="mume-header " id="scheduler_serializer-scrapy_redispicklecompat"></p>


#  Don't cleanup redis queues, allows to pause/resume crawls.
<p class="mume-header " id="dont-cleanup-redis-queues-allows-to-pauseresume-crawls"></p>

# SCHEDULER_PERSIST = True
<p class="mume-header " id="scheduler_persist-true"></p>


#  Schedule requests using a priority queue. (default)
<p class="mume-header " id="schedule-requests-using-a-priority-queue-default"></p>

# SCHEDULER_QUEUE_CLASS = 'scrapy_redis.queue.PriorityQueue'
<p class="mume-header " id="scheduler_queue_class-scrapy_redisqueueprorityqueue"></p>


#  Alternative queues.
<p class="mume-header " id="alternative-queues"></p>

# SCHEDULER_QUEUE_CLASS = 'scrapy_redis.queue.FifoQueue'
<p class="mume-header " id="scheduler_queue_class-scrapy_redisqueuefifoqueue"></p>

# SCHEDULER_QUEUE_CLASS = 'scrapy_redis.queue.LifoQueue'
<p class="mume-header " id="scheduler_queue_class-scrapy_redisqueuelifoqueue"></p>


#  Max idle time to prevent the spider from being closed when distributed crawling.
<p class="mume-header " id="max-idle-time-to-prevent-the-spider-from-being-closed-when-distribut

#  This only works if queue class is SpiderQueue or SpiderStack,
<p class="mume-header " id="this-only-works-if-queue-class-is-spiderqueue-or-spiderstack"></p>

#  and may also block the same time when your spider start at the first time (because the queue
```

```
<p class="mume-header " id="and-may-also-block-the-same-time-when-your-spider-start-at-the-first

# SCHEDULER_IDLE_BEFORE_CLOSE = 10
<p class="mume-header " id="scheduler_idle_before_close-10"></p>


#  Store scraped item in redis for post-processing.
<p class="mume-header " id="store-scraped-item-in-redis-for-post-processing"></p>


ITEM_PIPELINES = {
    'tutorial.pipelines.TutorialPipeline': 300,
    'scrapy_redis.pipelines.RedisPipeline': 400,
}
#  The item pipeline serializes and stores the items in this redis key.
<p class="mume-header " id="the-item-pipeline-serializes-and-stores-the-items-in-this-redis-key"

# REDIS_ITEMS_KEY = '%(spider)s:items'
<p class="mume-header " id="redis_items_key-spidersitems"></p>


#  The items serializer is by default ScrapyJSONEncoder. You can use any
<p class="mume-header " id="the-items-serializer-is-by-default-scrapyjsonencoder-you-can-use-any

#  importable path to a callable object.
<p class="mume-header " id="importable-path-to-a-callable-object"></p>

# REDIS_ITEMS_SERIALIZER = 'json.dumps'
<p class="mume-header " id="redis_items_serializer-jsondumps"></p>


#  Specify the host and port to use when connecting to Redis (optional).
<p class="mume-header " id="specify-the-host-and-port-to-use-when-connecting-to-redis-optional">

# REDIS_HOST = 'localhost'
<p class="mume-header " id="redis_host-localhost"></p>

# REDIS_PORT = 6379
<p class="mume-header " id="redis_port-6379"></p>


#  Specify the full Redis URL for connecting (optional).
<p class="mume-header " id="specify-the-full-redis-url-for-connecting-optional"></p>

#  If set, this takes precedence over the REDIS_HOST and REDIS_PORT settings.
<p class="mume-header " id="if-set-this-takes-precedence-over-the-redis_host-and-redis_port-sett

# REDIS_URL = 'redis://user:pass@hostname:9001'
<p class="mume-header " id="redis_url-redisuserpasshostname9001"></p>
```

```
#  Custom redis client parameters (i.e.: socket timeout, etc.)
<p class="mume-header " id="custom-redis-client-parameters-ie-socket-timeout-etc"></p>

# REDIS_PARAMS  = {}
<p class="mume-header " id="redis_params"></p>

#  Use custom redis client class.
<p class="mume-header " id="use-custom-redis-client-class"></p>

# REDIS_PARAMS['redis_cls'] = 'myproject.RedisClient'
<p class="mume-header " id="redis_paramsredis_cls-myprojectredisclient"></p>


#  If True, it uses redis' ``spop`` operation. This could be useful if you
<p class="mume-header " id="if-true-it-uses-redis-spop-operation-this-could-be-useful-if-you"></

#  want to avoid duplicates in your start urls list. In this cases, urls must
<p class="mume-header " id="want-to-avoid-duplicates-in-your-start-urls-list-in-this-cases-urls-

#  be added via ``sadd`` command or you will get a type error from redis.
<p class="mume-header " id="be-added-via-sadd-command-or-you-will-get-a-type-error-from-redis"><

# REDIS_START_URLS_AS_SET = False
<p class="mume-header " id="redis_start_urls_as_set-false"></p>


#  Default start urls key for RedisSpider and RedisCrawlSpider.
<p class="mume-header " id="default-start-urls-key-for-redisspider-and-rediscrawlspider"></p>

# REDIS_START_URLS_KEY = '%(name)s:start_urls'
<p class="mume-header " id="redis_start_urls_key-namesstart_urls"></p>


#  Use other encoding than utf-8 for redis.
<p class="mume-header " id="use-other-encoding-than-utf-8-for-redis"></p>

# REDIS_ENCODING = 'latin1'
<p class="mume-header " id="redis_encoding-latin1"></p>


USER_AGENT = 'scrapy-redis (+https://github.com/rolando/scrapy-redis)'

LOG_LEVEL = 'DEBUG'

#  Introduce an artifical delay to make use of parallelism. to speed up the
<p class="mume-header " id="introduce-an-artifical-delay-to-make-use-of-parallelism-to-speed-up-

#  crawl.
<p class="mume-header " id="crawl"></p>

DOWNLOAD_DELAY = 1
```

2.4.6 编写 spider 文件

tutorial/tutorial/spiders/myspider_redis.py 内容如下：

```
from scrapy_redis.spiders import RedisSpider


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

    def __init__(self, *args, **kwargs):
        # Dynamically define the allowed domains list.
        domain = kwargs.pop('domain', '')
        self.allowed_domains = filter(None, domain.split(','))
        super(MySpider, self).__init__(*args, **kwargs)

    def parse(self, response):
        for quote in response.xpath('//div[@class="quote"]'):
            yield{
                'text': quote.xpath('./span[@class="text"]/text()').extract_first(),
                'author': quote.xpath('.//small[@class="author"]/text()').extract_first(),
                'tags': quote.xpath('./div[@class="tags"]/a[@class="tag"]/text()').extract(),
            }
```

注意，上面代码中的 `name = 'myspider_redis'` 与 `redis_key = 'myspider:start_urls'` 很重要。

2.4.7 运行爬虫爬取信息

首先启动爬虫： myspider_redis，运行下列命令启动。

```
scrapy crawl myspider_redis
```

可以看到如下结果:

```
2019-07-31 17:10:52 [myspider_redis] INFO: Reading start URLs from redis key 'myspider:start_urls' (batch size: 1

...

2019-07-31 17:10:53 [scrapy.core.engine] INFO: Spider opened
2019-07-31 17:10:53 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 it
2019-07-31 17:10:53 [scrapy.extensions.telnet] INFO: Telnet console listening on 127.0.0.1:6023
```

上面的输出表示，当前爬虫正在等待redis 服务器中提供start urls.

2.4.8 设置redis键 myspider:start_urls

在redis中设置键 myspider:start_urls，并赋值为 http://quotes.toscrape.com/

```
lpush myspider:start_urls http://quotes.toscrape.com/
```

这个key在输入后，很快将被上面的爬虫myspider监听到，然后就取出作为start url（消费掉），而将数据以myspider:items 列表作为结果存储在redis中

2.4.9 查看结果

通过下面的命令查看：

```
127.0.0.1:6379> keys *
1) "myspider_redis:items"

127.0.0.1:6379> type myspider_redis:items
list

127.0.0.1:6379> llen myspider_redis:items
(integer) 20

127.0.0.1:6379> lrange myspider_redis:items 0 -1
 1) "{\"text\": \"\\u201cIt is our choices, Harry, that show what we truly are, far more than our abilities.\\u20
 2) "{\"text\": \"\\u201cThe world as we have created it is a process of our thinking. It cannot be changed witho

 ...
```

2.4.10 模块练习与答案

见习题集

2.4.11 内容小结

本节介绍了编写分布式爬虫的一个简单实例。

3 本节总结

本讲介绍了以下内容：

- 分布式系统概述
- scrapy-redis架构介绍

- 建立分布式爬虫的准备工作
- 分布式爬虫构建实例构建
- 运行爬虫与redis
- 查看结果

其中scrapy-redis的架构知识与分布式爬虫构建实操是重点。

4 课后练习

见习题集。