论文题目 基于 Scrapy 的分布式网络爬虫系统设计与实现

专业学位类别	工 程 硕 士
学　　号	201522240316
作 者 姓 名	樊宇豪
指 导 教 师	导师组

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

基于 scrapy 的分布式网络爬虫系统设计与实现

（题名和副题名）

樊宇豪

（作者姓名）

指导教师 导师组

电子科技大学 成 都

（姓名、职称、单位名称）

申请学位级别 硕士 专业学位类别 工 程 硕 士

工程领域名称 电子与通信工程

提交论文日期 2018.05.23 论文答辩日期 2018.05.30

学位授予单位和日期 电子科技大学 2018 年 6 月

答辩委员会主席 _____

评阅人 _____

注 1：注明《国际十进分类法 UDC》的类号。

Design and Implementation of distributed web crawler system based on Scrapy

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Discipline: **Master of Engineering**

Author: **Fan YuHao**

Supervisor: **Supervisor Group**

School: **Research Institute Electronic Science and Technology of UESTC**

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

作者签名：樊宇豪 日期：2018年5月25日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

作者签名：樊宇豪 导师签名：

日期：2018年5月25日

摘　要

随着互联网飞速的发展，网络上的信息呈指数级的增长，如此高的信息数量级也给用户获取信息带来了巨大的挑战。网络爬虫作为获取数据的工具常常被应用于搜索引擎当中，然而面向中小规模系统的网络爬虫由于其自身的局限性常常面临诸多问题，例如单机的网络爬虫程序抓取数据速度太慢，而大多数成熟的开源网络爬虫框架都未实现分布式化；互联网中的网页结构各不相同，单一的网络爬虫程序无法匹配所有类型的网页等等。因此设计并实现一个可定制性高的、简单稳定的、面向中小规模的高性能分布式爬虫具有很重要的意义，本文在 Scrapy 框架的基础上结合 Redis 数据库设计并实现了一个分布式网络爬虫系统，使用者通过简单的配置即能快速的抓取到其想要的数据。

本文的主要工作包括以下几点：

（1）重点研究了主从式架构下的任务调度算法，并在此基础上提出了一种动态反馈的任务调度策略。主节点在掌握从节点群中各个 Scrapy 爬虫实时状态的情况下使用该策略进行任务调度，并在爬虫节点发生变化时进行相应的任务调整，确保系统中的各个爬虫节点动态负载均衡。

（2）针对传统的基于内存或磁盘的 URL 去重时导致的空间占用率过高的问题，本文结合布隆过滤器算法提出了一种海量 URL 去重策略，该策略利用多个哈希函数对原始的 URL 数据集进行空间映射压缩，降低其空间的占有率，并且在查询过程中，仅通过一次哈希即可判断某 URL 是否抓取过，大大提高了查询效率。

（3）设计并实现了一种多节点下的爬虫限速策略，集群中的爬虫节点能根据用户设定的频率来访问对应的站点。其中基于 IP 的限速限定了同一台机器中的爬虫节点访问某站点的频率，基于爬虫类型的限速限定了同一种类型的爬虫节点访问某站点的频率。

（4）对 Scrapy 框架中的调度器、数据采集以及数据管道组件重新定制开发，调度器的开发使其支持分布式采集，数据采集的开发使其支持带采集规则的数据抽取，数据管道的开发使其支持数据清洗、编码转换以及正文提取等功能。

（5）基于 Twisted 框架设计并实现了一个异步任务响应的爬虫管理器，用户通过该管理器能方便的控制各节点上的 Scrapy 爬虫。

关键词：Scrapy 框架，分布式网络爬虫，布隆过滤器，限速策略，任务调度

ABSTRACT

With the rapid development of the Internet, the information on the network is growing exponentially, and the high level of information brings great challenges to users' access to information. Web crawler as tools to get the data is often used in search engines, however small and medium-sized system oriented web crawler due to the limitations of its own often face many problems, such as single web crawlers fetching data speed too slow, and the most mature open source web crawler frame are unrealized distributed; Web pages are different in structure, and a single web crawler can't match all types of web pages and so on. So was designed and implemented a high customizability, simple and stable, and the small and medium-sized high performance distributed crawler has the very vital significance, in this paper, on the basis of Scrapy framework combining Redis database was designed and implemented a distributed web crawler system, users can only simple configuration by the rapid capture to the desired data.

The main work of this paper includes the following points:

（1）Focusing on the task scheduling algorithm under the master-slave architecture, and a task scheduling strategy based on dynamic feedback is proposed. The master node uses this strategy to perform task scheduling while grasping the real-time status of each Scrapy crawler in the slave node group, and performs corresponding task adjustment when the crawler node changes to ensure dynamic load balancing of each crawler node in the system.

（2）Based on the traditional memory or disk URL to lead to the problem of high space utilization rate, when the bloom filter algorithm, this paper puts forward a massive URL to heavy strategy, this strategy by using multiple hash function for the original URL then compress the space mapping data collection, reduce its share of the space, and in the process of query, only through a hash can judge whether the URL to grab, greatly improve the query efficiency.

（3）Designed and implemented a multi-node crawler speed limit strategy. The crawler nodes in the cluster can access the corresponding site according to the frequency set by the user. The speed limit in the same machine based on IP crawler nodes access the frequency of a site, based on the crawler type speed limit the access to

a similar type of crawler node the frequency of a site.

（4）The scheduler in Scrapy framework, spider and data pipeline components to custom development, the development of the scheduler to support distributed acquisition and development of spider to support with the rules of data extraction, data pipeline development to support data cleaning, code conversion, and text extraction, and other functions.

（5）Based on the Twisted framework, a crawler manager is designed and implemented for asynchronous task response. The user can easily control Scrapy crawler on each node through this manager.

Keywords: scrapy framework, distributed web crawler, bloom filter, speed limit strategy, task scheduling

目 录

第一章 绪 论

1.1 研究背景及意义

随着互联网的快速发展，人们获取信息的渠道日益增多，其中网络是目前大众获取信息和新闻最普遍的渠道。然而，受互联网自身因素的影响，目前网络上的信息呈现出种类多，来源广，体量大等特点，而传统的搜索引擎只能模糊匹配出与用户需求相关的信息，对于一些具体化或精确化的需求，搜索引擎也无法满足。介于此原因，聚焦网络爬虫应运而生。

聚焦网络爬虫[1]作为信息获取的工具之一早在 90 年代初就开始被广泛的使用，其目的就是快速准确获取互联网中特定主题的信息，本文也是在此基础上深入研究。早期互联网中网页数量并不大，开发者一般选择将网络爬虫程序放在单台机器中运行，但随着网络迅速的发展，目前互联网中网页的数量早已和过去不是一个量级了，有数据表明，截至 2018 年 1 月底，中国互联网中“.CN”域名网站总数已超过 2085 万个，相比过去同比增长 3%。面对如此庞大数量的网页，仅仅想依靠单机版的网络爬虫程序获取足够多的信息是不太现实的，即便有高性能、高带宽的服务器支撑，爬虫自身的采集速度也远远跟不上网页增长的速度，因此，支持多台机器并行数据采集的分布式网络爬虫诞生了。

一些知名的互联网公司（如 Google，百度）早在互联网还未如此普及的时候就开始对分布式网络爬虫进行研究，并且在大数据量采集和任务调度等相关技术上提出了自己的解决方案，但是其核心技术却一直作为商业机密不对外公开。另一方面，对于大部分数据采集商来说，传统的面向搜索引擎的分布式网络爬虫系统过于复杂，应用性不高，而目前开源的网络爬虫框架中，虽能满足基本的采集需求，但大部分都未实现分布式抓取。

本文依托于教研室内部的横向项目，主要负责采集各大门户新闻及政府招标通告等信息，这些网站中的网页数量巨大且内容更新速度快，如果使用单机版的爬虫采集，不仅速度极慢而且可能导致内容更新不及时。故对可定制性高、部署简单的中小规模的分布式网络爬虫研究是非常有必要的。

1.2 国内外研究现状

国外对分布式网络爬虫的研究相比于国内要发展的早一些，其中比较有名的有 Google 的 BackPub 和 Apache 的 Nutch，其爬虫采集范围广、爬行速度快，在数

据挖掘领域有着极其出色的表现。此外，国外还有一些具有不同特点的分布式网络爬虫：

1999 年 6 月，康伯系统研究中心的 Allan Heydon 和 Marc Najork 发表了一篇名为“A scalable，extensible Web Crawler”的文章[2]，该文中的爬虫采用 Java 多线程机制实现网页抓取的并行化，并通过 DNS 缓存、延迟缓存等技术提升爬虫的抓取效率，同时系统采用低耦合设计原则，使得开发者可以轻松替换掉部分组件就能实现定制化的需求。

Boldi P，Codenotti B 等人在 2002 年发表了一篇名为“A Scalable Full Distributed Web Crawler”的文章[3]，该文提出了一种名为 UbiCrawler 的基于 P2P 的分布式网络爬虫，UbiCrawler 中没有中心节点和爬行节点之分，爬虫节点之间采用一致性哈希算法来分发爬虫抓取任务，避免了由中心节点单独分发任务而导致的负载过重问题，其次，去中心化的设计解决了系统中单点故障的问题。

Internet Archive Crawler 是一个基于站点的分布式网络爬虫系统[4]，该架构中的每个爬虫可以同时负责 64 个站点的抓取任务，且每个站点中的网页都由同一个爬虫采集，避免频繁 DNS 解析导致系统效率过低的问题，同时系统采用异步 IO 的方式对页面进行下载，避免了由 IO 阻塞而导致的等待时间过高的问题。

BB Cambazoglu，A Turk 等人在 2005 年提出了一种基于网格服务的分布式网络爬虫系统[5]，该系统将可利用的资源按特定的策略分布到先前设定好的网格中，当有新的爬虫任务需要抓取时，利用网格强大计算能力为相应的爬虫分配合适的资源，以此来提升系统的整体效率。

2015 年，P Felber，E Riviere 等人在 IEEE Cloud 会议上发表了一篇名为“A Practical Geographicall Distributed Web Crawler”的文章[6]，该文首次提出了一种名为 UniCrawl 的基于地理位置的分布式网络爬虫，UniCrawl 是在 Apache 研发的 Nutch 开源网络爬虫框架基础上定制而成。为了减少与采集站点的通信消耗，UniCrawl 被分布到地理位置不同的各个地区，每个爬虫只负责采集与自己位置相对较近的域名站点；其次，采用 Hadoop 的 MapReduce 计算框架[7]对采集后的数据进行分布式处理，大幅度的提高了系统的整体性能。

国内研究分布式网络爬虫的机构和学者也有许多，其中北大天网和上交 Lgoo 设计的网络爬虫作为国内高性能网络爬虫的代表在分布式网络爬虫领域做出了极大的贡献：

北京大学的天网爬虫是一个面向局域网的分布式网络爬虫系统[8]。该爬虫由最初的集中式转变到后来的分布式，不仅抓取数提升了一个量级，稳定性也相比以前提高了许多。该系统采用了传统的主从式架构，主节点根据爬虫的性能采用三

级调度[9]对任务进行分配，从节点负责具体网页的采集工作，同时在遍历网页过程中，采用传统的广度优先策略进行抓取[10]。该项目已经在实际的应用中使用，并且取得了不错的效果。

上海交通大学的 Lgoo 系统是一个基于网格服务的分布式网络爬虫系统[11]。该系统能跨局域网进行爬虫节点的部署，同时为了保证系统中的各个爬虫节点负载均衡，采用二级哈希映射算法[12]来进行任务的分发工作。

此外，相关学者也对分布式网络爬虫做了大量研究。哈工大的付志辉提出了一种基于在线反馈站点规模大小的预测算法，通过该算法可推算出集群中爬虫节点负荷当量，并以此为依据进行任务调度，保证负载均衡[13]。电子科大的李婷在传统的一致性哈希算法基础上提出了一种均分负载空间算法，主节点采用该算法避免了一致性哈希算法在爬虫节点较少的情况下，抓取任务分配不均衡的问题[14]。荣晗针对 Scrapy 框架不支持分布式采集的缺陷，结合 Storm 流计算框架使 Scrapy 爬虫进程运行在 Storm 分布式平台上，实现分布式采集[15]。程锦佳利用 MapReduce 的编程模式对网络爬虫的各个阶段重新改造，使其能利用 Hadoop 分布式计算特性对网页进行并行抓取[16]。

1.3 主要研究工作

本文研究的内容属于分布式网络爬虫系统。针对现有网络爬虫框架 Scrapy 不支持分布式抓取的缺陷，本文结合 Redis 数据库设计并实现了一个灵活性高、可扩展性强并且健壮的分布式网络爬虫系统，并且从以下几个方面解决了分布式爬虫的几个核心问题：

1. 爬虫节点负载均衡。分析影响爬虫抓取效率因素并确立爬虫权值计算公式，并结合动态反馈任务调度策略分配爬虫任务，确保系统中的各个爬虫节点动态负载均衡。

2. 海量 URL 去重。大规模采集务必解决 URL 去重问题，传统的基于磁盘的 URL 去重策略不仅占用大量磁盘空间，同时去重速度极慢，本文基于布隆过滤器算法实现海量 URL 去重策略，该策略能大幅度的减少去重过程中系统空间上和时间上的耗费。

3. 站点限速。为了防止集群中的爬虫对目标站点频繁访问导致远程服务器负荷过重，本文设计并实现了基于 IP 限速和基于爬虫类型限速两种策略来控制集群中的爬虫对目标站点的访问速度。

4. 传统的数据采集过程中，在面对页面结构不同的网页时，不得不单独为其开发一个网络爬虫程序，随着采集任务不断增多，程序会越来越大，同时也增加了开

发人员的任务量，本文设计了一个基于爬行规则的数据采集模块，用户通过简单的配置规则即能采集到对应网页上的数据。

最后，为了更好的控制各个节点上的爬虫，本文基于 Twisted 框架实现了一个异步任务响应的爬虫管理器，用户可以通过爬虫管理子页面来远程启动或停止爬虫，同时也能查看当前节点中各爬虫运行状态（待启动、正在运行中以及已完成）。

1.4 论文组织结构

论文全文分为六章，具体章节安排如下：

第一章 绪论。本章主要论述了本文的研究背景以及意义，结合分布式网络爬虫当前国内外研究现状，提出本文的主要研究内容。

第二章 相关理论与技术基础。本章首先介绍了网络爬虫技术的工作原理及相关技术，接着介绍了目前主流的分布式爬虫系统架构模型及常见的任务调度策略，然后介绍了 Scrapy 框架结构并分析其不足，最后介绍了系统中用到的组件，包括 Redis 数据库、Kafka 消息系统及 Zookeeper。

第三章 分布式网络爬虫系统设计。本章首先介绍了系统的设计目标，并根据该目标设计系统的总体，系统主要由三个部分构成：消息处理、主节点以及从节点（群），消息处理主要负责前端用户发送请求的验证和处理，主节点主要负责爬虫任务的分发和接收，从节点（群）主要负责数据的抓取与存储。

第四章 系统的详细设计与实现。本章介绍了系统各模块的设计原理，并详细描述了主节点和从节点群中各功能模块的实现。

第五章 系统测试与展示。本章主要为系统的测试和展示部分，首先介绍了系统整体的运行环境，然后分别对任务调度器、过滤器以及爬虫采集速度三个方面进行性能测试，最后从用户的角度展示了如何使用该系统完成抓取任务。

第六章 总结与展望。本章主要总结本文所做的主要工作，分析其存在的问题和不足，并提出相应的改进方法。

第二章 相关理论与技术基础

2.1 网络爬虫

网络爬虫的本质是一个遵循既定规则，自动的从互联网中抓取相关信息的计算机程序。它最初被设计为大型搜索引擎项目中的子项目，为其提供信息搜索的渠道，国内外比较知名的如百度、Google 等等。作为搜索引擎的核心技术之一，网络爬虫自身的性能往往决定了最终提供给用户的服务质量，故对网络爬虫技术本身的研究也是极其重要的。

2.1.1 网络爬虫的分类

网络爬虫种类繁多，每类爬虫都有其特定的功能，但是从应用场景的角度出发，可以将其划分为以下三种类型：全网型，聚焦型以及增量型[17]。

全网型的网络爬虫常常用作抓取互联网上所有的资源，最常见的应用就当属搜索引擎了。这种类型的网络爬虫抓取的网页数一般以亿为单位，而且需要重复抓取一些频繁更新的网页，故这类爬虫往往对抓取速度和存储量都有着极高的要求。图 2-1 展示了全网型网络爬虫常见的抓取流程。

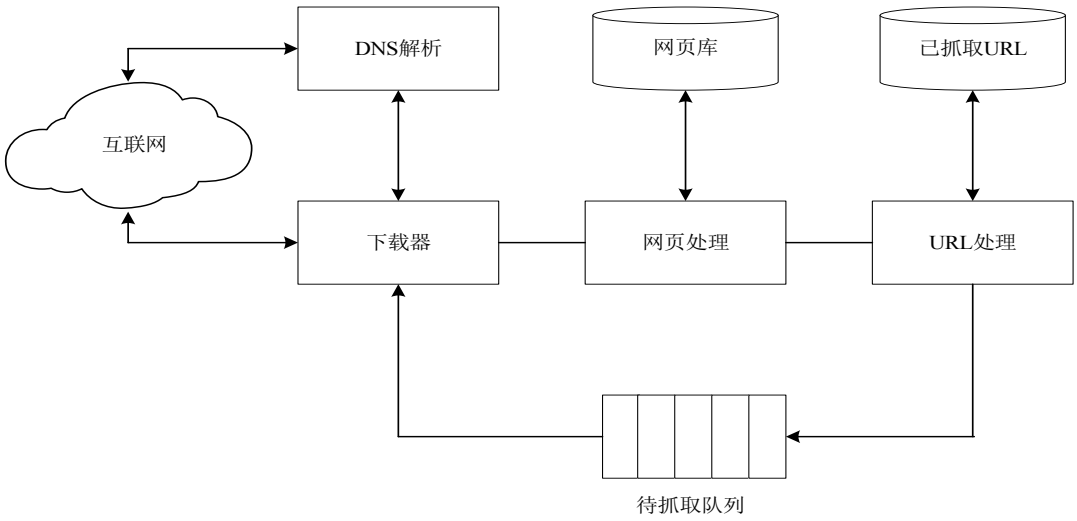

图 2-1 全网型网络爬虫抓取流程

如图 2-1 所示，程序首先将用户提供的初始种子 URL 加载到预先创建的待抓取 URL 队列中，然后依次从待抓取 URL 队列中取出 URL 交付给下载器，下载器

收到来自抓取队列中的 URL 后，立即向对应的远程服务器发起 HTTP 请求，在请求过程中经过 DNS 解析器解析出的 IP 地址单独缓存下来，避免后续请求对同一域名进行解析，提高系统效率。当远程服务器返回网页内容（HTML 源代码）后，将其交给网页解析模块处理，其中从网页中抽取出的新的 URL 链接交给 URL 处理模块，而网页原始内容直接存入网页库中。最后由 URL 处理模块将新链接中已采集过的链接过滤掉，并将剩下的 URL 链接存入待抓取 URL 队列中。循环该过程，直到待抓取 URL 队列中不存在新的链接为止，这样就能将互联网中所有的网页抓取完毕。

聚焦型的网络爬虫又常常被称作面向主题的网络爬虫[18]，与全网型的网络爬虫不同的是，该类型的爬虫往往只采集某一特定主题或者某一特征下的网页信息，其它无关信息一律不予抓取。这样的抓取策略能避免让爬虫抓取大量无关的信息，减少了系统的开销及网络流量的耗费，本文也是基于此类型的爬虫进行深入的研究。

增量式的网络爬虫与全网型的网络爬虫相似[19]，唯一不同的是它采用增量的方式来更新已采集过的网页。该类型的爬虫在抓取过程中只抓取新出现的网页和内容更新过的网页，这样的策略能保证数据库中存放的网页数据时刻都是最新的。相比于周期性的重复抓取页面内容未发生变化的网络爬虫，增量式网络爬虫在时间和空间上的耗费相对要少许多，但也间接的增加了抓取算法的复杂度。

2.1.2 网络爬虫的爬行策略

按照网络爬虫遍历互联网中网页的方式，可以将其划为以下三种：广度优先遍历，深度优先遍历以及最佳优先遍历[20]。

广度优先遍历[21]在其内部维护了一个支持先进先出的待抓取 URL 队列，爬虫首先从初始的页面中抽取出存在于当前页面中的所有 URL 链接并依次按顺序加入到先前创建好的待抓取 URL 队列中，然后不断重复执行从待抓取 URL 队列中读取 URL 链接、下载对应网页内容、抽取网页中 URL 链接加入到待抓取 URL 队列的过程，直到队列中没有新的 URL 链接，以这样的方式遍历互联网中的网页称之为广度优先遍历。图 2-2 展示了广度优先遍历的结构图，该爬行策略将待抓取的网页分成各种不同的层次，爬虫始终按照第一层、第二层直到最后一层这样的顺序抓取网页，这种抓取策略比较适合大规模的抓取，常常应用在水平型搜索引擎中。

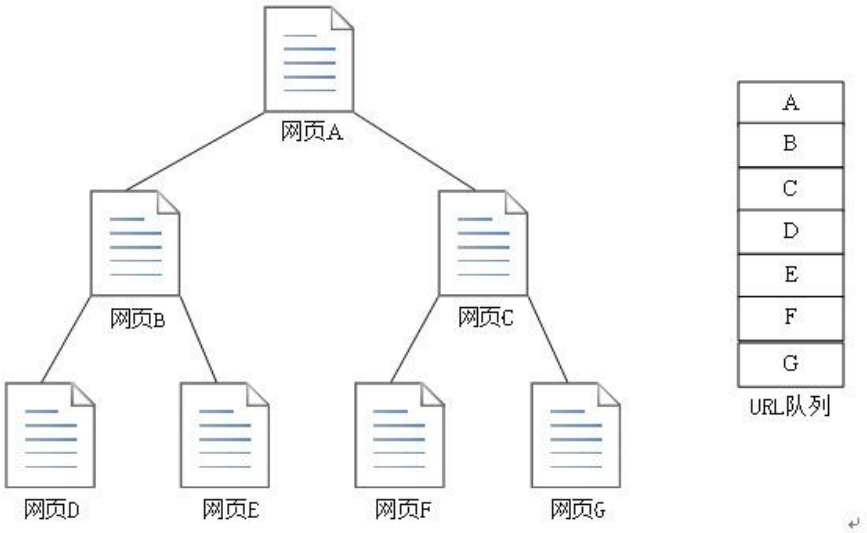

图 2-2 广度优先遍历策略

深度优先遍历[22]与树的先序遍历极为相似，该策略着眼与整个抓取过程中的深度，它通过网页中某一链接进而访问完所有与其相关的所有链接。爬虫程序首先从初始的网页中抽取一个未抓取的 URL 链接，并依次遍历完所有与其相关的 URL 链接，当与其相关的所有 URL 链接访问完毕后，再从初始网页中抽取一个未抓取的 URL 链接，反复执行该过程，直到初始网页中不在有新的链接存在，这样的方式就被成为深度优先遍历。该策略也有一定的缺点，比如常常出现大规模的重复抓取现象，既频繁抓取同一个网页。图 2-3 为深度优先抓取策略的结构图。

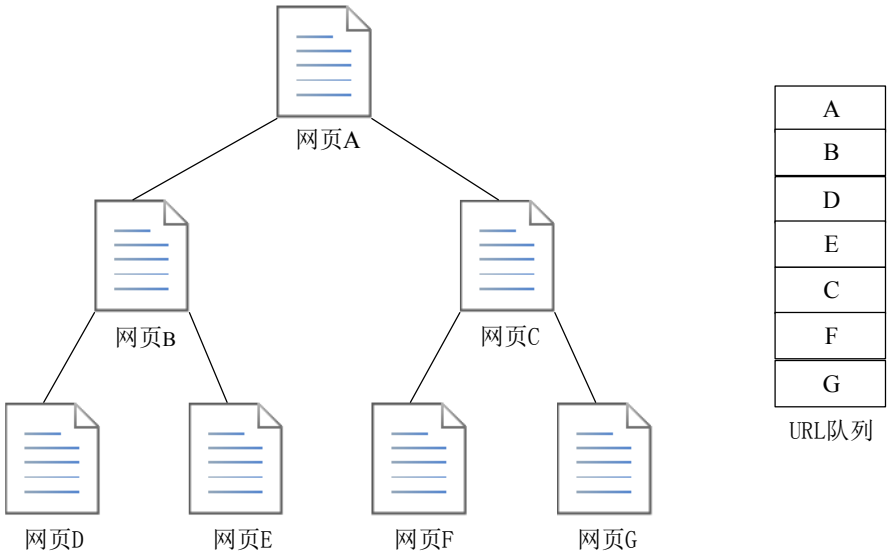

图 2-3 深度优先遍历策略

最佳优先遍历[23]事先采用网页相关度算法计算抓取的URL与特定主题的相关都，并根据计算得出的值对URL按照从高到底的顺序排序，其中与主题相关度最高的URL会被优先采集。由于该策略只采集与特定主题有关联的网页，在某种意义上来说其避免了无关网页的抓取，且需要结合实际情况找到最优点，并跳出。图2-4为最佳优先遍历结构图。

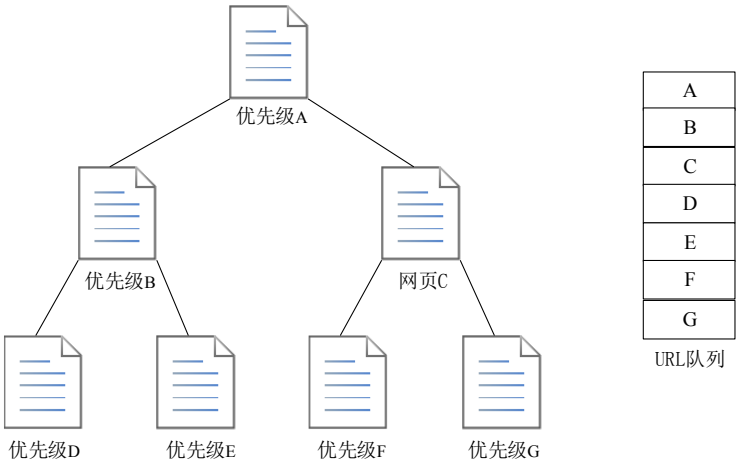

图2-4 最佳优先抓取策略

2.2 分布式网络爬虫

分布式网络爬虫，顾名思义其为多个单个网络爬虫组合起来的组合式爬虫。站在爬虫的工作方式的角度上来看，爬虫可以分为以下两类：集中式和分布式[24]，由于分布式网络爬虫常常使用多台机器对网页并行抓取，因此其耗费的时间也远远低于集中式的网络爬虫。

2.2.1 分布式爬虫系统架构

分布式爬虫系统架构复杂多变，站在系统的物理结构的角度上来看，其架构可以划分为以下两种： 主从式和对等式。

主从式架构[25]通常由一个主节点和若干个分别部署且能与主节点通信的爬虫节点构成，主节点负责根据爬虫节点状态分配合适的任务，爬虫节点根据主节点分发的任务完成具体网页信息的采集，同时各个爬虫节点均可以并行抓取且不需要相互通信，该架构实现相对简单，但随着任务量的增多，主节点会成为整个系统的瓶颈。图2-5为主从式架构的物理结构图。

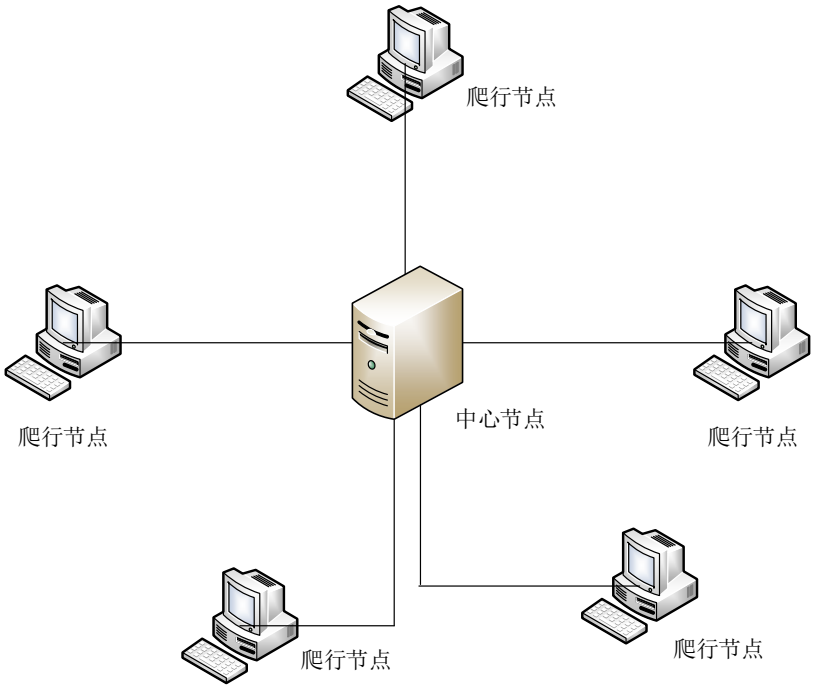

图 2-5 主从式架构图

对等式架构[26]与主从式架构相似，唯一不同的是对等式架构中没有了主节点，只有爬虫节点，且每个爬虫节点在系统的都占据着同等地位。由于系统中没有了主节点，故爬虫节点既要完成爬虫抓取任务的分发工作，也要完成对应页面的采集工作，图 2-6 展示了对等式架构的物理图。

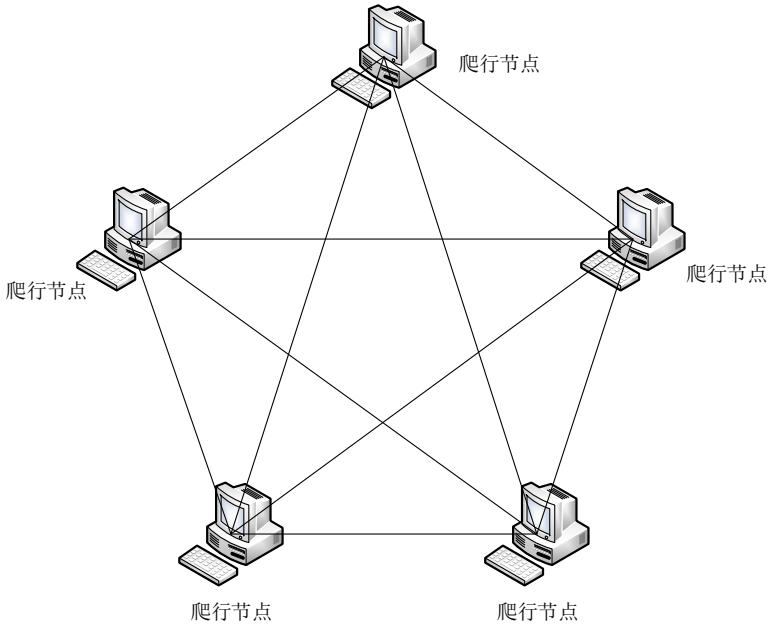

图 2-6 对等式结构图

由图 2-6 所示，该架构中的任意爬虫节点都与剩下的爬虫节点保持两两通信，在抓取过程中，当爬虫节点从网页中抽取出新的 URL 链接时，需要首先判断该链接是否属于自身的抓取范围，如果属于，则直接存放到自身的待抓取 URL 链接队列中，如果不属于，则根据相应的算法选出负责该 URL 抓取的爬虫节点，并交付给它。由于任务的分发工作在爬虫节点端，故系统不会因为一台节点宕机就导致系统崩溃，因此相比与主从式架构，该模式健壮性更好。但是由于各个爬虫节点需要与其余剩下所有爬虫节点保持通信，这无疑增大了系统整体的通信开销，降低了爬虫的性能，而且当一台爬虫节点宕机后，其待抓取队列中存放的 URL 链接也随之丢弃，使得最终抓取的数据不完整。

2.2.2 任务调度策略

任务调度负责对任务进行分发操作，该策略不仅仅适用于分布式网络爬虫，而且适用于常见的分布式系统中。一个好的任务调度策略能保证各节点始终保持负载均衡，并且最大限度上发挥出各爬虫节点的抓取能力。比较经典的任务调度策略算法有轮询调度算法（Round Robin）、加权轮询调度算法（Weighted Round Robin）、一致性哈希调度算法（Consistent Hashing）以及最快响应调度算法等等。

（1）轮询调度算法[27]（Round Robin）

为了将请求尽可能均匀的分发到性能相同的服务器上，Shreehar Madhavapeddi 和 George Vaghese 两人在 1994 年提出了一种新的近似公平排队的算法[14]，算法将任务按照一定的顺序存入一个共享队列中，并依次从队头取出任务分配给各个节点，该策略非常适合于一些不能被分解成更小单位的调度问题。

（2）加权轮询调度算法[28]（Weighted Round Robin）

轮询调度算法按照服务器节点个数依次分发请求，当服务器性能相同的情况下，该策略能均衡的为每台服务器分发大致等量的请求，但是现实条件下，系统中的机器处理能力各不相同，如果按照轮询调度算法来为每台机器分配等量的请求数，不仅不能发挥出各机器最大的能力，同时可能会造成某些处理能力弱的服务器负载过重，导致其宕机。加权轮询调度算法在轮询调度算法的基础上，事先根据每台服务器的处理能力设置一个初始的权值，系统接受到请求后，按照其权值的大小分配合适的请求数，这样就解决了不同机器处理能力不同条件下的负载均衡问题。

（3）一致性哈希调度算法[29]（Consistent Hashing）

一致性哈希调度算法在 1997 年由麻省理工学院的 karger 等人提出，该算法意在解决互联网中的热点（Hot spot）问题。现在该算法也被广泛的运用在分布式系

统当中，比较有名的如 memcached 缓存数据库、Cassandra 数据库。

以 memcached 缓存数据库为例，算法首先对数据库中的每一个存储节点经过哈希函数计算出对应的哈希值，并将其配置到一个 $0\sim2^{32}$ 的圆环上，然后采用同样的方式计算出需要存储的数据的键的哈希值，并将其映射到上述相同的圆环上，最后从数据映射的位置开始顺时针遍历，遇到的第一个存储节点就是数据最终存放的位置，如果超过 2^{32} 任然未找到对应的存储节点，则将该数据保存到第一台存储节点中。图 2-7 展示了一致性哈希算法映射过程。

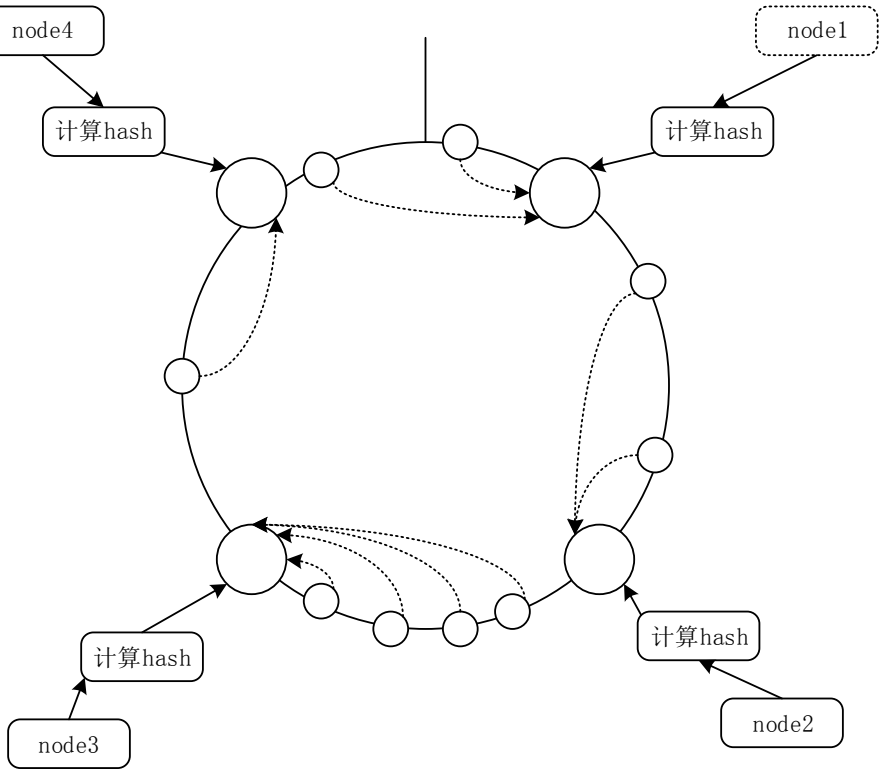

图 2-7 一致性哈希算法映射过程

（4）最快响应调度算法[30]

最快响应调度算法以系统中各节点响应速度作为负载标准，当某个服务器节点出现负载过重的情况时，首先向系统中其它服务器节点发送求助请求，中心节点会在此过程中统计各个服务器节点回复该负载过重的节点所花费的时间，然后选出回复时间最短的节点帮忙分担负荷节点的任务，以此来实现各个服务器节点之间的负载均衡。

2.3 Scrapy 框架研究

Scrapy 是由 Scrapinghub 公司开源的一款网络爬虫框架[31]，该框架基于 Twisted

异步网络库开发而成，主要用于数据采集相关工作，在数据挖掘、自动测试领域都有不错的使用率。该框架的组件都是可插拔的，开发人员可以根据特定的需求对相应组件进行定制开发，从而实现一个高效、功能强大的爬虫。

2.3.1 Scrapy 框架结构

Scrapy 框架主要由 Engine（引擎）、Scheduler（调度器）、Downloader（下载器）、数据解析（Spider）、数据管道（ItemPipeline）这 5 个部分组成，图 2-8 展示了其框架结构。

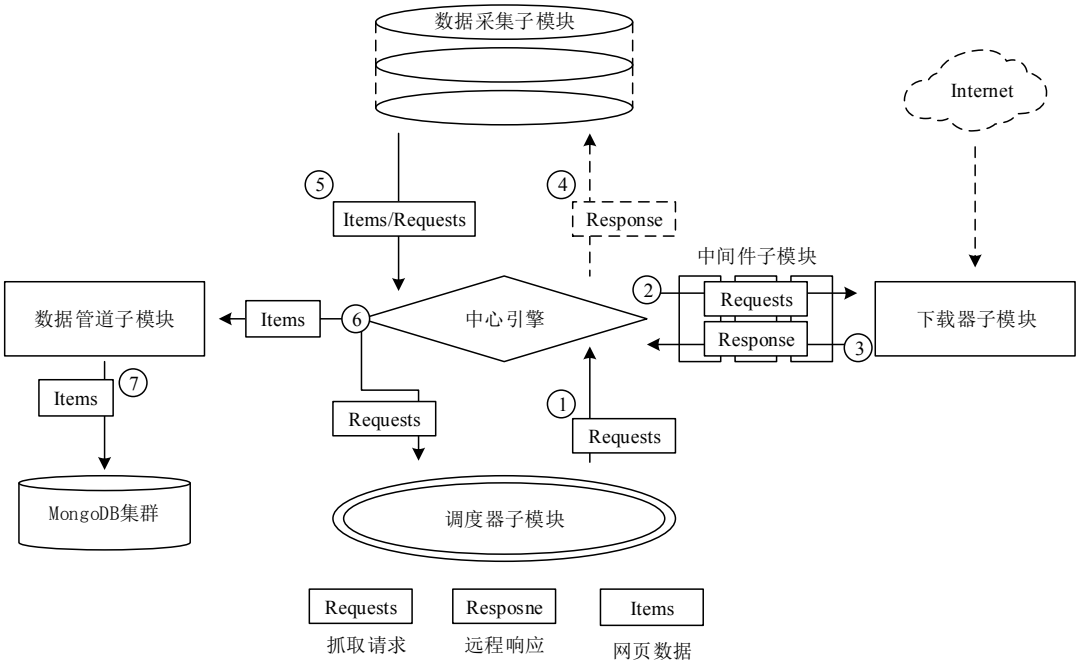

图 2-8 Scrapy 框架结构

（1）中心引擎（Engine）

中心引擎作为 Scrapy 的核心大脑，它负责管理系统中所有的数据流向，每个模块中的输入输出都需要经过中心引擎，故该组件在 Scrapy 框架中占据了相当重要的地位，一旦中心引擎失效，系统将无法正常运转。

（2）调度器（Scheduler）

调度器负责周期性的接受从中心引擎传递过来的爬虫任务，并将该任务其存入自身的待抓取队列中，以待后续中心引擎向其发出获取爬虫任务请求时，再从自身的待抓取队列中取出交给中心引擎。该组件的核心目的是为了对爬取请求分类管理，常见的待抓取队列有基于先进先出的，后进先出的以及基于优先级的等等。

（3）下载器（Downloader）

下载器负责网页的下载工作。与传统的直接向浏览器发送 HTTP 请求不同，下载器采用异步的方式建立与远程服务器的联系，程序不需要一直阻塞等待服务器端返回的响应，而是先完成自身的其它任务，待收到远程服务器的响应后，才跳转回去进行相应的处理，该设计模块解决了程序需要花费大量时间在 IO 阻塞的问题。

（4）数据解析（Spider）

数据解析模块是由开发人员编写的爬虫核心程序，该程序负责解析从下载器中传递过来的网页源代码，常见的网页解析器有 beautifulsoup、lxml 等等。当从网页中抽取出后续需要跟进的 URL 链接时，数据解析模块会将此 URL 封装为系统设定的 Request 请求并交给中心引擎，而从网页中提取出的数据则交给数据管道处理。通常数据解析可以对应多个程序，每个程序负责解析一个特定的网站。

（5）数据管道（Pipeline）

数据管道模块负责将数据解析模块中传递过来的数据进行进一步处理。常见的操作包括：编码转换、数据清洗以及数据持久化等等。

2.3.2 Scrapy 框架的不足

虽然 Scrapy 框架为开发者提供了如此多优秀的功能，但是目前该框架还未实现分布式抓取解决方案，在对大型网站点抓取时，单机版的爬虫往往显得力不从心。其次，在整个抓取过程中，爬虫需要频繁的与待抓取 URL 队列交互，故需要抓取的 URL 链接往往选择存放到内存而不是硬盘中，当抓取的 URL 数量达到百万级别甚至千万级别的时候，对单机的内存也是一个很大的考量，故将其改进使其支持分布式抓取的工作势在必行。

2.4 相关技术

2.4.1 Redis 数据库

Redis 是一款由 Salvatore Sanfilippo 研发出的 NoSQL 数据库[32]。作为一款开源的高性能的基于 Key-Value 的内存型数据库，官方提供了五种数据类型来满足用户不同场景下的需求，并支持当前所有主流的开发语言。

（1）字符串类型（String）

Redis 中的字符串类型与 Python 中的字符串类型相似，唯一不同的是，该类型可以存放任意形式的字符串，典型的如二进制数据。一个字符串类型的 value 最多

存放 512M 的数据。

（2）散列类型（Hash）

Redis 中散列类型与 Python 中的字典类型相似，都是以键值对的方式存放到内存中，由于该数据类型底层基于哈希表实现，故其增删改查的时间复杂度都为 O(1)，每个散列类型的 key 最多可以存放 2^{32}-1 个键值对。

（3）列表类型（List）

列表类型是 Redis 中比较常用的数据类型，该类型和 Python 中的列表类似。列表类型中的元素以字符串的形式出现，其底层是基于双向链表实现的，故该类型支持从表头或表尾增删元素，其时间复杂度为 O(1)。一个列表型的 key 最多能存放 2^{32}-1 个元素。

（4）集合类型（Set）

Redis 中的集合类型是基于散列表实现的，与列表类型不容，集合类型中的元素不能重复，该特点常常被用作去重处理。由于该类型底层是基于哈希表实现的，故其判断元素、插入元素以及查找元素等操作时间复杂度都为 O(1)。对集合的常见操作有 Scard（获取集合成员数）、SADD（向集合中添加元素）、SMEMBERS（返回集合中所有元素）、SPOP（移除集合中随机一个元素）等等，一个集合类型的 key 最多能存放 2^{32}-1 个元素。

（5）有序集合（Sorted Set）

Redis 中的有序集合类型基于集合类型实现的。与集合类型不同，有序集合类型中的每个元素都绑定了一个相应的分数，元素按照分数的大小从低到高依次排序，其中有序集合中的元素不能相同，但是元素的分数可以相同。有序集合常见的操作有 ZADD（向有序集合中添加元素）、ZRANGE（获取指定分数端内的元素）、ZREM（移除有序集合中的某个元素）等等，一个有序集合类型 key 最多能够能存放 2^{32}-1 个元素。

Redis 除了为用户提供上述丰富的数据结构外，还具备以下一些优点：

1. 读写速度快

由于 Redis 本身存储方式基于内存，故有着极高的读写效率，官方测试读的速度为 100K 次/s，写的速度为 80K 次/s。

2. 原子性

由于 Redis 是单线程的，故其所有操作都是原子操作，每个操作要么执行成功，要么执行失败。

3. 数据持久化

Redis 官方提供了两种方式对其内部存储的数据持久化：AOF 和 RDB，RDB

方式采取的是全量备份，该方式通过将内存中的数据以快照的形式写入磁盘；AOF方式采取的是增量备份，该方式实际保存的是客户端执行的命令，当用户需要还原数据时，只需重新执行上述命令即可。

2.4.2 Kafka 消息系统

Kafka 的定位是一个流式处理平台[33]，由 Apache 软件基金会研发。目前大多数公司都将其作为多种类型的数据管道和消息系统使用，同时其高吞吐量、高容错等特性也很好适合在大规模消息处理的场景中使用。

2.4.2.1 名词解析

主题（Topic）：在 Kafka 消息系统中，每一个消息都有其所属的分类，而这个分类称为消息系统中的主题。例如门户网站中每一条新闻都有其对应的分类，如社会类，娱乐类等等，该类别就是 Kafka 消息系统中的主题。

分区（Partitions）：每个 Topic 中都存放了大量的消息，而这些消息又被分割到一个或若干个队列中，这些队列称为 Kafka 消息系统中的分区。每个 Topic 都至少有一个 partitions，partitions 中的消息按照存入时的顺序存放，故单个 partitions 中的消息是有序的。图 2-8 展示了拥有 3 个分区的 topic。

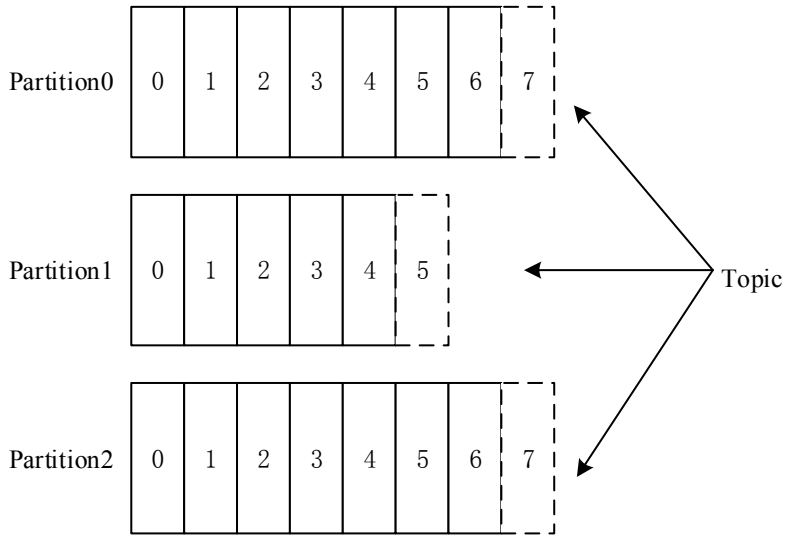

图 2-8 主题与分区关系

消息偏移量（Partition offset）：Kafka 消息系统中每条消息内部都存放了一个当前 Partition 下唯一的 64 字节的递增序列号，该序列号的值指明了该消息在当前 Partition 下的起始位置。

生产者（Producer）：生产者，顾名思义即消息的发布者，该角色将生产的消息

发布到 Kafka 对应的 Topic 中。当消息队列接收到生产者发送的消息后，立即将其追加到 Segment 文件中，同时生产者也可以为发送的消息指定 Partition。

消费者（Consumer）：消费者，顾名思义即消息的消费者，该角色从 Kafka 对应的 Topic 中读取消息，而且消费者可以读取多个 Topic 中的消息。

代理（Broker）：Kafka 消息系统由一个或多个服务器组成，每个服务器节点都可以成为 Broker，Broker 负责存储 Topic 中的消息，如果一个 Topic 有 N 个 Partition，Kafka 消息系统有 N 个 Broker，则每个 Broker 负责存放该 Topic 下的一个 Partition。

副本（replications）：副本是 Kafka 消息系统中 Topic 下一个 Partition 的备份，副本主要用作防止数据的丢失，而不会交给 Consumer 消费。其中每个 Broker 下只能为一个分区分配最多有 0 到 1 个副本。

消费者群组（Consumer Group）：Kafka 消息系统中由若干个消费者组成的群体称之为消费者群组。每个消费者都有其特定的消费者群组，Kafka 消息系统通过该方式实现 Topic 中消息的广播（将 Kafka 消息系统中的消息发送给所有的消费者）和单播（将 Kafka 消息系统中的消息发送给某个消费者群组下的某个消费者）。

2.4.2.2 发布-订阅消息系统

Apache Kafka 是一个基于分布式的发布-订阅消息系统[34]，其支持 TB 级的海量数据传递，在大型企业的离线和实时消息处理业务中也都有广泛的应用。图 2-9 展示了 Kafka 消息系统中分布式发布-订阅消息模型架构。

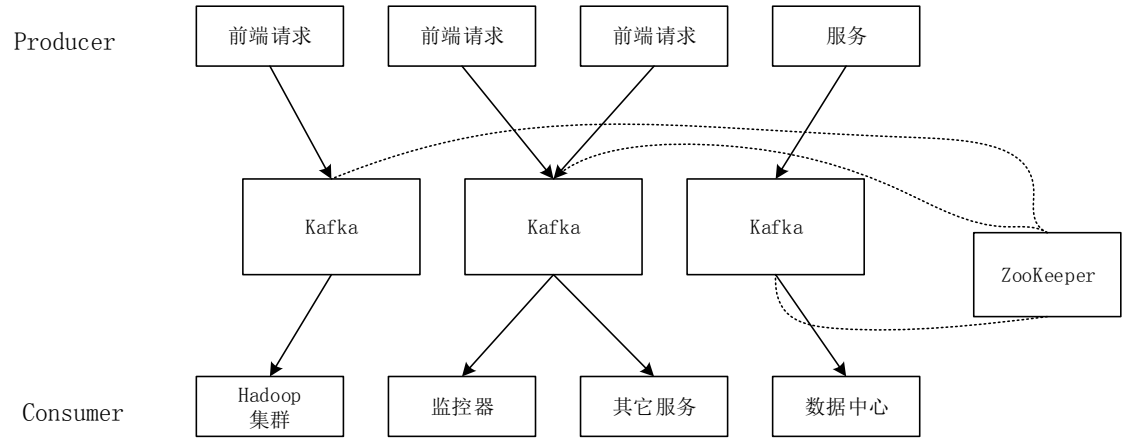

图 2-9 分布式消息订阅和发布系统架构

如图 2-9 所示，生产者首先将消息发送到对应的 Topic 中，消费者根据不同的消费方式选择 Topic 中的消息进行消费，下面主要介绍两种比较常见的消费方式的流程。

一个消费者订阅数据：

- 生产者生产消息并将消息发送到指定的 Topic 中
- Kfaka 消息系统收到消息后，将消息均衡的分布到 Topic 下某个 Partition 中。
- 消费者订阅 Kafka 消息系统中某个特定的 Topic
- 当消费者完成 Topic 的订阅后，Kafka 将当前消息的偏移量发送给消费者
- 消费者周期性的从 Kafka 获取消息
- 当消费者消费完先前获取的消息后，向 Kafka Broker 发送反馈信息
- 当 Broker 收到消费者发送的反馈后，更新当前消息的偏移量
- 重复以上过程，直到消费者停止请求数据

消费者群组消息消费流程：

- 生产者生产消息并发送到指定 Topic 中
- Kafka 收到消息后，将其存放到指定 Topic 下的某个分区中
- 某个消费者订阅了该 Topic 下的消息，且该消费者属于“Group one”下的消费者。
- 此时 Kafka 的处理方式和第一种方式的结果一致。
- 当 Kafka 收到一个来自同一个消费者群组的消费者请求时，Kafka 消息系统将自动切换为分享模式，此时消息将在两个消费者上共享。
- 当消费者的数目超过了该主题的分区数量时，即使队列中还有消息也不会交给后面的消费者，因为 Kafak 消息系统必须为每一个消费者至少分配一个 Partition。

2.4.3 ZooKeeper

ZooKeeper（动物管理员），顾名思义，该项目用于管理 Apache 下的 Hadoop（大象）、Pig（小猪）、Hive（蜜蜂）[35]。本文中的 Kafka 消息系统也采用该框架。ZooKeeper 目前是 Apache 下的顶级项目，它主要负责为大型企业提供高可用的数据管理和应用程序协调等服务，其本质为一个精简的文件系统，该文件系统中并没有为用户提供文件和目录等常规选项，取而代之的是使用节点（Znode）的概念，本节将通过从 ZooKeeper 数据模型和 Wacth 机制两方面介绍其在项目中用到的特性。

ZooKeeper 中的数据模型有点类似于 Linux 下操作系统中的文件系统，其文件也是以树状的形式呈现。最上层是根节点，以“/”来代表，每个节点可以拥有 N 个子节点，每个子节点上都有其自己的名字而且还能存储少量数据，该数据可以为字符串也能是一串二进制数，每个节点最多能存放 1M 的数据，由此可知，ZooKeeper 主要用来协调服务的，而不是存储数据的。

ZooKeeper 中的文件系统和 Linux 表示路径的方式大体一致，例如图 2-10 所示，

/app1/p_2 表示节点 p_2 的搜索路径，app1 是 p_2 的父节点，/是 app1 的父节点。

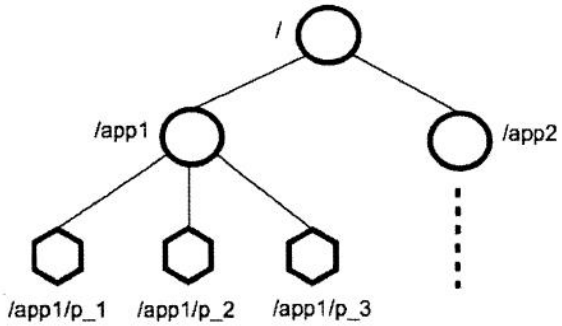

图 2-10 Zookeeper 名字空间

每个节点内部不仅存储相应的数据，还存储了节点的创建时间、修改时间、结点 id 等相关数据信息，当客户端访问相应节点上的数据时，要么获得全部数据，要么读取失败，故其所有的相关操作都被定义为原子性。

Watch 机制属于设计模式中的观察者模式，该机制可以使客户端在某个 Znode 节点发生改变时得到通知，而不用一直循环遍历获取节点变化信息。ZooKeeper 对节点的变化定义可以从以下几个条件判断：节点本身数据修改、节点被删除、节点增加以及其孩子节点变化，只要符合上述条件，客户端都可以为其注册 Watch 事件。Zookeeper 在 Watch 机制的触发上使用的是一次性触发方式，假设某次通知完成之后则清空信息，数据再次变化时将不再通知节点，除非客户端重新设置了新的 Watch 监视器。Watch 事件是异步发送到客户端的，也就是说 ZooKeeper 可以保证客户端发送过去的更新顺序是有序的。例如，ZooKeeper 中某个 Znode 没有设置 watcher，那么在对该 Znode 节点设置 Watcher 之前，客户端是无法感知到节点任何改变的。

2.5 本章小结

本章对基于 Scrapy 的分布式网络爬虫系统的相关理论和技术进行了简要的介绍。首先介绍网络爬虫的基本概念，接着引入分布式网络爬虫，并介绍了当前主流的分布式爬虫系统架构及任务调度策略，然后分析了 Scrapy 网络爬虫框架的基本结构及其对应的不足，最后对系统中使用到的相关技术进行了简要的描述。

第三章 分布式网络爬虫系统设计

3.1 系统设计目标

基于 Scrapy 的分布式网络爬虫系统是一种面向特定主题需求的网络爬虫系统。与传统的通用网络爬虫不同，本系统中的爬虫不需要采集互联网上所有的资源，而只需要获取与用户需求相关的网页信息。此外，除了完成最基本的抓取工作外，本文从以下几个方面进一步完善该分布式网络爬虫系统：

（1）灵活性 不同网站的网页结构几乎不同，为每个网页设计不同的爬虫程序不仅耗费大量的资源同时大大增加开发人员的工作量，所以设计出一个有足够灵活性的网络爬虫以至于在不同环境下只需要做出很少的改动就能应用它就非常有必要了。

（2）分布式 单机抓取速度有限，随着网页数量不断的增加，采集的时间也越来越长。原生的 Scrapy 框架不支持分布式抓取，故需要在原有基础上，重写部分组件使其支持分布式抓取。

（3）负载均衡 负载均衡对于中小规模系统的分布式网络爬虫至关重要。现实条件下各个机器性能不一，集群中各个爬虫抓取能力也不同，所以要确保系统能根据各个爬虫节点实时的负荷状态分配相应的任务量，这样才能最大化提高爬虫的抓取效率。

（4）礼貌性 遵循一定的爬虫规范是作为抓取者最基本的道德，这就要求我们不能在同一时刻将过多的请求发送到远程服务器。限制集群中的爬虫节点对网站频繁的访问是对网站管理者最好的尊重。

（5）健壮性 每个好的系统都具备较强的错误处理能力。比如说某个环节突然发生错误或某个数据库突然宕机时，系统必须能够及时处理这些突发状况；或者一些网站具备很强的反爬虫[36]技术，当集群中的爬虫频繁访问其站点时，很容易被屏蔽，导致无法抓取网站数据，故系统中应该有很好的策略来应对这一类问题。

（6）人性化 提供给用户的操作及配置应尽可能简单。如用户应该能通过系统界面轻松便捷的启动爬虫或停止爬虫，同时也能实时监控集群中各个爬虫速度，待完成任务个数等相关状态。

3.2 系统总体设计

3.2.1 系统总体结构

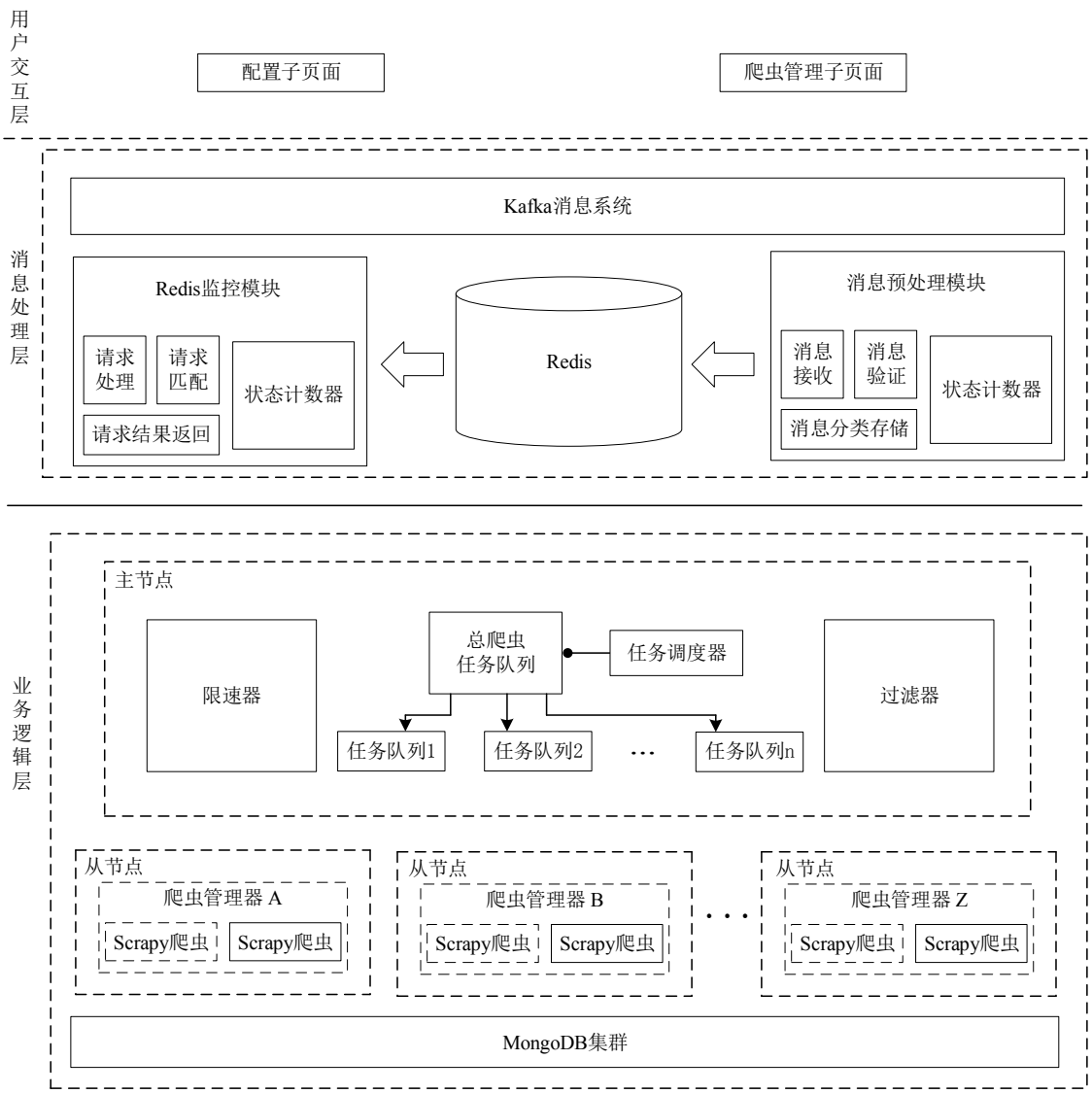

图 3-1 分布式爬虫系统设计架构图

基于以上设计目标，本节对分布式网络爬虫系统进行更为详细的划分。如图 3-1 所示，系统从总体上划分为三层，自底向上依次为业务逻辑层、消息处理层以及用户交互层。

用户交互层负责提供与用户交互的页面，包括配置子页面和爬虫管理子页面。配置子页面负责接收用户提交的请求，请求划分四类：爬虫抓取请求、爬虫任务状态查询请求、模块状态查询请求以及站点配置信息更新请求，爬虫抓取请求负责提供爬虫抓取任务的相关信息，如抓取的 URL、页面抽取规则等等；爬虫任务状态查询请求负责获取指定用户下某个爬虫抓取任务当前处理的情况，如当前某个爬虫抓取任务还有多少个 URL 或站点待抓取；模块状态查询请求负责获取系统中各模块运行状态，如消息预处理模块或 Redis 监控模块是否正常工作等等；站点

配置信息更新请求负责更新集群中爬虫对某些站点访问的限制信息，如将某些站点加入抓取黑名单、限制爬虫对某些站点的访问频率，每类请求都有其固定的格式，方便后续消息预处理模块解析。爬虫管理子页面负责启动与停止集群中的爬虫，同时还能监控各个节点上爬虫当前运行状态（待启动、运行中以及已结束）。

消息处理层主要负责验证、解析处理来自配置子页面传递的请求。考虑到随着业务的增加，配置子页面传递的请求数和消息处理层返回的响应结果不断增多，为了能更好地处理这些活跃的流数据，系统采用 Kafka 作为其中间缓存器，保证数据在各个子模块中高性能、低延迟地不停流转。消息处理层由消息预处理模块和 Redis 监控模块组成，各模块功能描述如下：

（1）消息预处理模块

作为 Kafka 消息系统中的消费者，消息预处理模块负责周期性的从 Kafka 中获取来自配置子页面的请求，并验证请求格式是否正确。状态计数器作为消息预处理模块中的插件记录了该模块单位时间内验证成功或失败的请求数。当请求验证成功后，如果该请求属于爬虫抓取请求，则直接存入总爬虫任务队列，如果属于其余三类请求，则按照相应的格式存入 Redis 中。同时为了避免消息预处理模块宕机导致请求无法处理，系统同时运行着多个该模块（所有的消息预处理模块属于同一个消费组，避免消息重复消费），保证任意时刻模块宕机都不会影响系统正常运行。

（2）Redis 监控模块

作为 Kafka 消息系统中的生产者，Redis 监控模块负责周期性检测 Redis 中是否有从消息预处理模块传入的验证成功的请求，如果有则调用相应的插件来处理解析该请求，并将响应的结果返回 kafka 消息系统中。例如监控模块检测到 Redis 中存在爬虫任务状态查询请求，则调用 Info 插件来统计指定爬虫任务当前处理情况（待抓取的 URL 或站点个数等等），处理完成后将统计的结果返回给 Kafka 消息系统，供用户交互层展示。同时系统采用多个 Redis 监控模块并行检测处理 Redis 中的请求，保证系统可靠性的同时大大加快了用户请求响应的速度。

业务逻辑层是本系统的核心，该层由一个主节点和若干个从节点构成。与传统的对等式结构相比，主从式结构将所有爬虫节点的任务调度交给主节点统一执行，不仅任务调度层次分明，实现简单，而且能够根据爬虫节点实时运行状态，做出最优的调度决策。其物理结构图如图 3-2 所示。

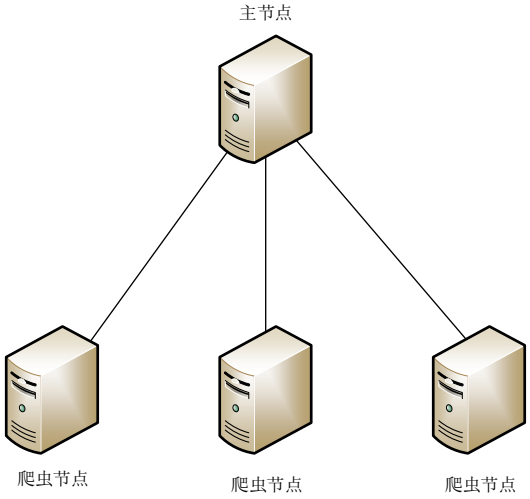

图 3-2 主从结构物理图

主节点由任务调度器、过滤器、限速器以及爬虫任务队列四个部分组成，其中爬虫任务队列分为两类：总爬虫任务队列和节点爬虫任务队列。总爬虫任务队列中存放是配置子页面中传入的爬虫抓取请求及后续爬虫节点从网页中抽取的新的爬虫抓取任务；节点爬虫任务队列中存放的是各个爬虫节点自身待抓取的爬虫任务。为了保证爬虫节点能迅速的获取与提交爬虫任务，主节点采用内存型数据库 Redis 存放待抓取的爬虫任务。

（1）任务调度器

任务调度器负责将总爬虫任务队列中的爬虫任务分发到各个节点的爬虫任务队列中。为了保证集群中各个爬虫节点负载均衡，任务调度器根据爬虫节点实时状态，采用动态反馈任务调度策略对总爬虫任务队列中的爬虫抓取任务进行分发。

（2）限速器

限速器负责控制集群中爬虫节点在固定时间段内访问某个网站的频率，避免过快的访问速度导致其被网站屏蔽而无法抓取数据，同时也减轻了远程服务器的压力。

（3）过滤器

为解决集群中爬虫节点对相同的 URL 重复抓取，提高整体的抓取效率，过滤器模块基于布隆过滤器算法实现了海量 URL 去重，避免了传统的 URL 去重中内存占用过高、判重时间过长等问题。

从节点（群）主要负责完成爬虫节点启动，数据采集以及数据存储任务。该节点由爬虫管理器、Scrapy 爬虫以及 MongoDB 集群三个部分组成，各部分具体功能描述如下：

（1）爬虫管理器

爬虫管理器负责根据爬虫管理子页面中发送的请求启动或停止爬虫，并实时监控节点中各个爬虫运行状态。该管理器并没有单独放置在主节点中，而是分布在各个从节点中，避免了单个管理器负载过重的问题。

（2）Scrapy 爬虫

Scrapy 爬虫负责具体的数据采集工作。该模块由调度器子模块、中间件子模块、下载器子模块、数据采集子模块以及数据管道子模块组成。调度器子模块负责周期性的从自身的爬虫任务队列中获取爬虫抓取任务，并按照 Scrapy 规定的格式将任务封装成 Request 请求，交给中间件。当收到数据采集子模块从后续网页中提取出的 URL 时，将此 URL 按照系统定义的爬虫抓取请求格式封装成新的爬虫抓取任务，提交到主节点中的总爬虫任务队列中。中间件子模块负责动态改变 Request 请求中的 IP 地址来应对网站的反爬虫策略，保证抓取过程中的稳定性。下载器子模块负责根据最终的 Request 请求，向目标网页发起访问，并下载其对应的网页内容，交给数据采集子模块。数据采集子模块收到下载的网页内容后，根据用户自定义的抽取规则提取出相应的网页数据及后续跟进的 URL 链接，其中网页数据交给数据管道子模块做进一步处理，需要跟进的 URL 链接交给调度器进一步处理。数据管道子模块负责处理从数据采集模块中提取网页数据，典型的处理包括编码转换、数据清理、网页正文提取以及持久化操作。

（3）MongoDB 集群

MongoDB 集群负责最终抓取数据的存放。采用集群的模式不仅考虑了后续业务增长数据量突增的情况，同时避免了单台机器宕机导致数据丢失的问题。

3.2.2 系统运行机制

本节将通过描述一次完整抓取过程来说明系统中各模块是如何协作的，图 3-3 展示了基于 Scrapy 的分布式网络爬虫系统整体运行数据流图。

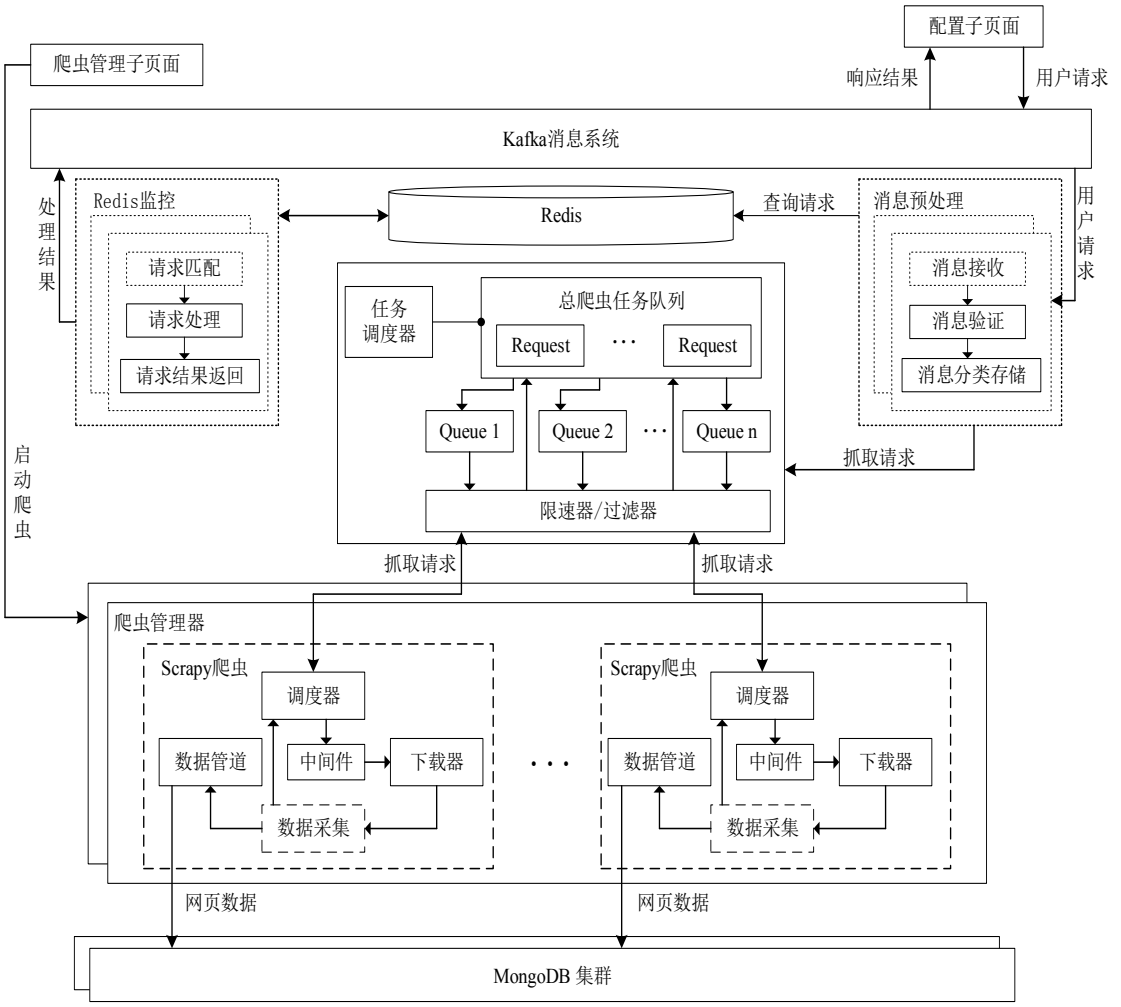

图 3-3 系统运行数据流图

首先用户在配置子页面中提交待抓取的 URL 及其相关信息，并通过爬虫管理子页面启动相应的爬虫。后台收到页面提交的爬虫抓取请求后，立即存入 Kafka 消息系统中，并返回用户提交成功信息。当消息预处理模块检测到 Kafka 消息系统中有新的抓取请求后，先验证请求是否符合系统定义的格式，如果符合，则将该请求存入总爬虫任务队列中；如果不符合，则记录到失败日志中。

接着任务调度器根据集群中正在运行的各个爬虫节点负荷状态，采用动态反馈任务调度策略对总爬虫任务队列中爬虫抓取任务进行分发。当爬虫节点的调度器子模块检测到自身的爬虫任务队列中有新的爬虫抓取任务后，先通过限速器判断集群中的爬虫节点对抓取任务中的站点访问频率是否超过了系统设定的访问频率，如果超过了，则获取其它站点下的爬虫抓取任务；如果没有超过，则从队列中取出该爬虫抓取任务并按照 Scrapy 规定的格式将其封装成 Request 请求，交给中间件。中间件子模块收到来自调度器子模块发送过来的 Request 请求后，从代理

IP 池中随机取出一条高匿 IP 装载到 Request 请求中，并将其交给下载器。下载器子模块根据最终的 Request 请求，向目标网页发起访问，并下载其对应的网页内容。待网页下载完后，数据采集子模块根据用户自定义的抽取规则提取出其需要的网页数据及后续待跟进的 URL 链接，其中网页数据交给数据管道子模块处理，URL 链接交给调度器子模块处理。调度器收到来自数据采集子模块中提交的 URL 链接后，先通过过滤器判断当前 URL 链接是否已经抓取过，如果没有抓取过，则按照系统定义的爬虫抓取请求格式封装成新的爬虫抓取任务并提交到主节点中的总爬虫任务队列中；如果已经抓取过，则直接丢弃。

最后数据管道子模块将采集到的网页数据经过编码转换、数据清理以及正文提取后，存入 MongoDB 集群中。

3.3 数据库设计

分布式爬虫系统内部使用了两类数据库：一类为 MongoDB 数据库，该数据库负责存储爬虫最终抓取的数据；另一类是 Redis 数据库，该数据库负责存储用户提交的爬虫抓取请求及后续爬虫从网页提取出的新的抓取请求。这两种数据库在系统中都扮演了很重要的角色，不管 MongoDB 或 Redis 数据库出现宕机或存储不足的情况都会导致整个系统崩溃，鉴于此原因，本文在搭建数据库时均使用集群来提高其可靠性及性能。

3.3.1 Redis 集群

Redis 集群是一个提供多个 Redis 节点间共享数据的程序集，集群中引入哈希槽（Hash Slot）的概念将整个集群划分为 16384 个槽，其中每一个节点负责一部分哈希槽。任何一个进入的 Redis 的键值对 key-value，都会根据 key 进行散列分配到这 16384 个槽中的某一个。Redis 作为一款内存型数据库，对内存的消耗极高，分槽概念的引入解决了数据只能存放在单台机器的问题，大大降低了数据库对单台机器内存的占用。

现实条件下难免出现节点宕机的情况，为了保证集群仍正常运行，Redis 集群提供了主从复制机制，每个主节点都有若干个从节点，主节点负责客户端的读写请求，从节点负责备份主节点中的数据。一旦某个主节点宕机，集群将从从节点中选取一个节点作为新的主节点继续工作，避免了单点故障问题。

在本系统中有 3 个 Redis 主节点 A、B 以及 C，其中节点 A 负责 0 到 5500 号的哈希槽，节点 B 负责 5501 到 11000 号的哈希槽，节点 C 负责 11001 到 16384 号的哈希槽，三个节点分配在 3 台物理机上且每个主节点都配备 1 个从节点，保证

任何一个主节点宕机都不会影响整个系统工作。图 3-4 展示了系统 Redis 集群结构图。

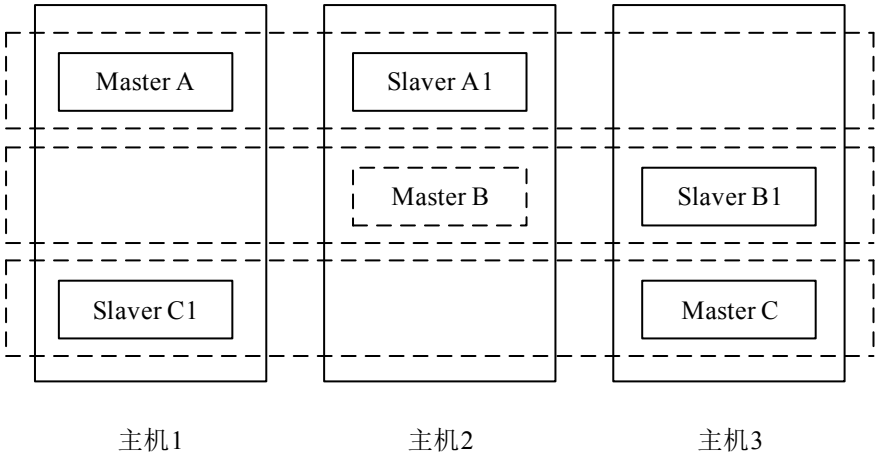

图 3-4 Redis 集群结构图

3.3.2 MongoDB 集群

MongoDB 集群[37]使用分片（Sharding）技术将数据存放在多台机器中，可以满足数据量迅速增长的需求。图 3-5 展示 MongoDB 集群中的各个组件图。

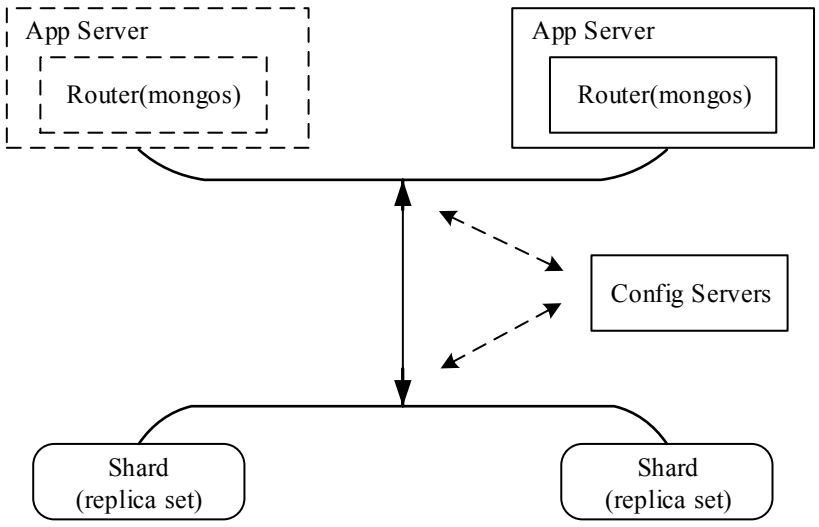

图 3-5 MongoDB 集群组件

路由控制器（mongos）是数据库集群的请求入口，所有的请求都通过 mongos 进行协调，它负责把对应的数据请求转发到相应的 Shard 分片服务器上。

配置服务器（Config Servers）存储所有数据库元信息（路由、分片）的配置，mongos 每次启动都会从该服务器中加载配置信息，同时如果配置服务器信息变化

也会立即通知 mongos 更新自己状态。

分片（Shard）负责最终的数据存储并提供方便的数据访问接口，其中每个分片也可以是副本集（replica set），简单来说就是 Shard 分片的备份，防止 Shard 宕机之后数据丢失，保证数据的安全性。

其中还有一个概念在图 3-5 中未表示出来：仲裁者（Arbiter），该实例负责在 Shard 分片宕机后从副本集中选举出新的 Shard 分片。

在本系统中配置了 3 个 mongos 和 3 个 config server，防止其中一个挂掉导致后续的存储查询都无法完成，每台机器中都启动一个分片服务器来存储用户抓取的数据，并在其余两台机器中分别启动副本集和仲裁者来防止数据的丢失。图 3-6 展示了系统中 MongoDB 集群的结构图。

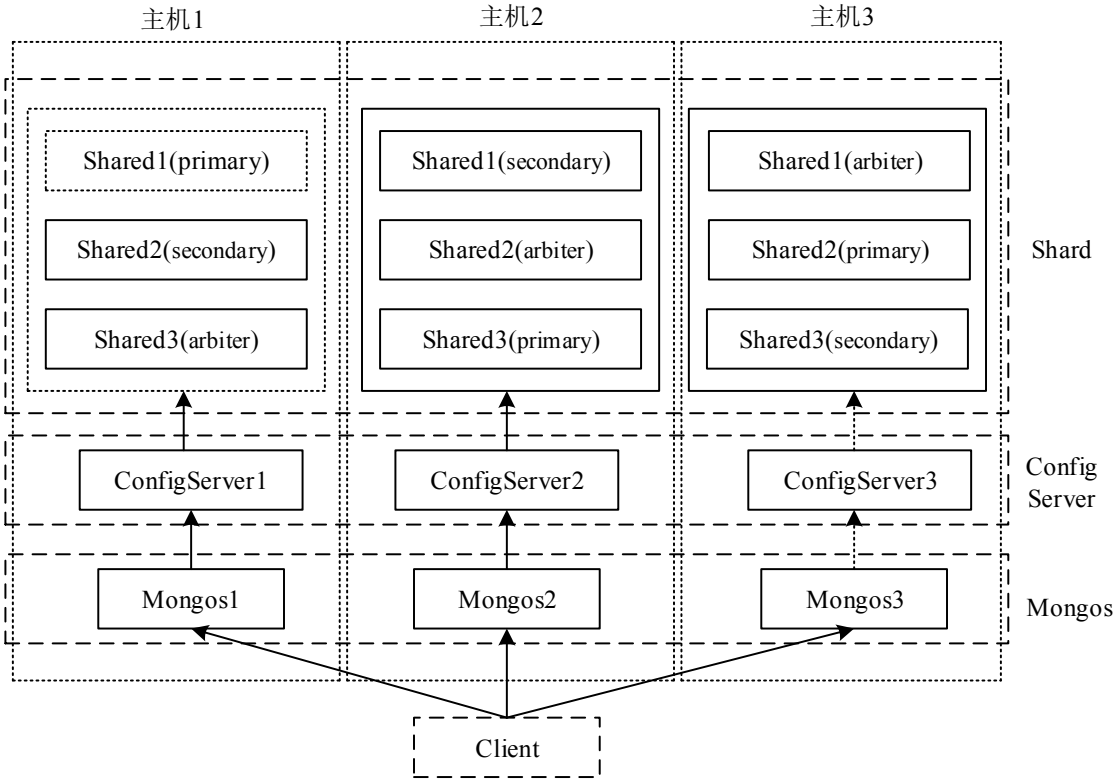

图 3-6 MongoDB 集群结构图

3.4 本章小结

本章首先从系统的设计目标入手，详解分析了分布式网络爬虫应具备哪些功能特性，然后根据设计目标，采用主从结构设计出系统的总体架构，并从每一层具体分析各模块功能，接着结合系统运行数据流图详细描述了一次完整抓取过程。最后结合 MongoDB 集群及 Redis 集群介绍了系统的数据库设计方案。

第四章 系统详细设计与实现

本章主要介绍基于 Scrapy 的分布式网络爬虫系统的详细设计与实现。根据第三章中设计的总体架构，从消息处理、主节点以及从节点（群）三个部分来详细描述系统中各模块设计及功能实现。

4.1 消息处理

消息处理作为整个后台系统的入口，负责请求的接收，验证和处理。该部分由状态计数器、消息预处理以及 Redis 监控三个模块组成，下面依次从这几个模块描述其具体设计与实现。

4.1.1 状态计数器

状态计数器作为消息预处理模块和 Redis 监控模块中的组件负责记录消息预处理模块和 Redis 监控模块中各插件单位时间内验证或处理的请求数。本节设计并实现了一个基于 Redis 的状态计数器，通过该计数器程序能在特定时间段进行分布式计数，同时该计数器提供两种模式供管理员选择：一种为基于滑动窗口[30]的整数计数器，该模式允许管理员查看最近 x 秒内各插件处理的请求数，另一种为基于固定时间段的整数计数器，该模式允许管理员查看具体某个时间段内各插件处理的请求数。图 4-1 展示了与状态计数器相关的类图。

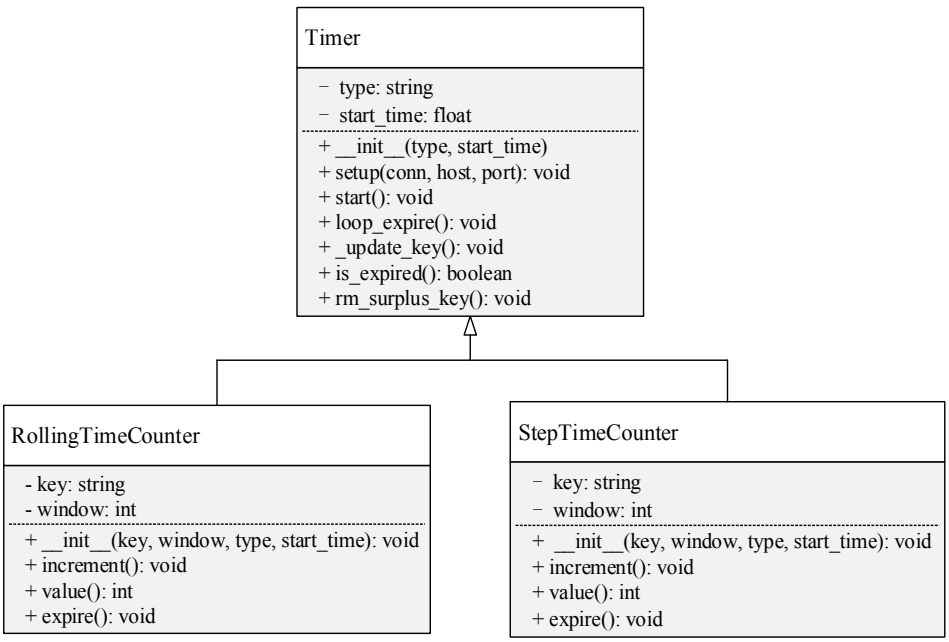

图 4-1 状态计数器类图

由图 4-1 所示，状态计数器由 RollingTimeCounter 类与 StepTimeCounter 类构成，ProllingTimeCounter 类负责基于滑动窗口的整数计数器，StepTimeCounter 类负责基于固定时间段的整数计数器。

基于滑动窗口的整数计数器是由 Redis 的有序集合（zset）来实现，该集合中的每个元素都会关联一个 Double 类型的分数，并且按照分数的大小从小到大排列。消息预处理模块和 Redis 监控模块中每个插件都关联着一个属于自己的计数器，当插件处理了一个新的请求时，调用 RollingTimeCounter 类的 increment 方法，该方法负责向有序集合中插入一条元素，元素的值及分数都设置为当前系统的时间；当需要统计某插件最近 window 秒内处理了多少个请求时，调用 RollingTimeCounter 类的 value 方法，该方法首先获取当前系统时间 t_1，然后减去滑动窗口 window 的值求出以当前系统时间为轴，前 window 秒的具体时间点 t_2，并遍历有序集合将元素分数小于 t_2 的元素删除，最后统计集合中目前元素的数量，该值就是最近 window 秒内该插件处理的请求数。同时为了预防集合中元素无限增长，系统通过 Timer 类的 setup 方法创建一个后台线程，定时将当前系统时间前 window 秒的集合元素删除。图 4-2 展示了计数器统计插件处理请求个数的原理图。

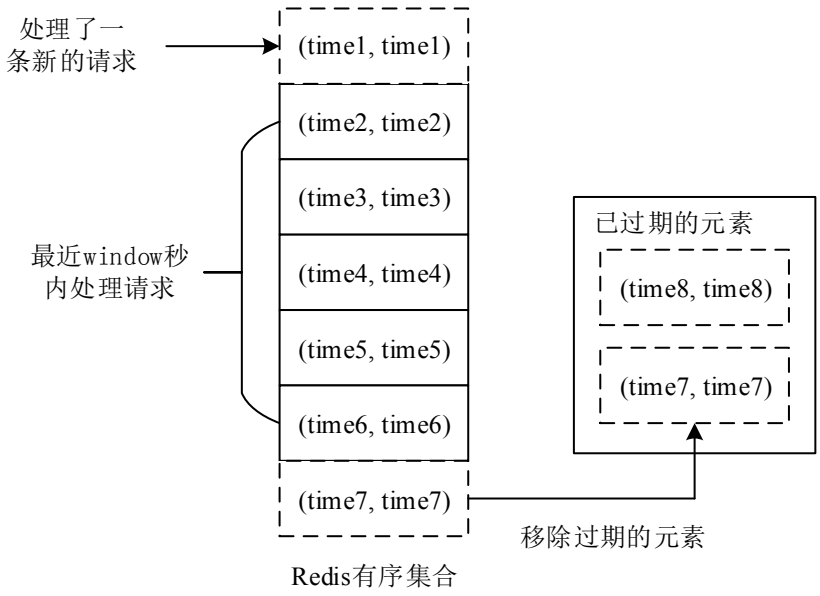

图 4-2 基于滑动窗口的整数计数器原理图

基于固定时间段的整数计数器同样也是通过 Redis 的有序集合实现，为确保格式的统一，计数器在初始化时从开始时间的整点开始计数，如管理员设置统计在 8:25 am 后某插件每小时处理的请求数，则统计的开始时间设为 8:00 am，初始完计数器后，当插件处理了一个新的请求后，调用 StepTimeCounter 类的 increment

方法，该方法首先获取当前系统的时间 t1，并判断当前时间 t1 与开始时间的差值是否小于窗口的大小（一小时），如果是则向有序集合中插入一条元素，元素的值及分数都设置为当前系统时间 t1，其中有序集合的 key 为{插件名}:{开始时间点}，否则不予置理。同时为了保证计数器能记录后续时间段插件处理的请求数，程序调用 Timer 类的 loop_expire 方法循环判断当前时间是否已不在初始设置时间段中，如果不在则调用_update_key 方法更新初始的开始时间点，保证 Redis 中记录了不同时间段的 key。当需要统计插件某个时间段处理了多少个请求时，只需要调用 value 方法统计对应 key 中集合元素的个数即可。

4.1.2 消息预处理模块

消息预处理模块在 Kafka 消息系统中充当消费者的角色，该模块负责订阅 Kafka 特定主题（Topic）中的消息（前端用户发送的请求）并确保不同类别的消息请求以正确的格式存入 Redis 集群中。图 4-3 展示了消息预处理模块整个处理流程。

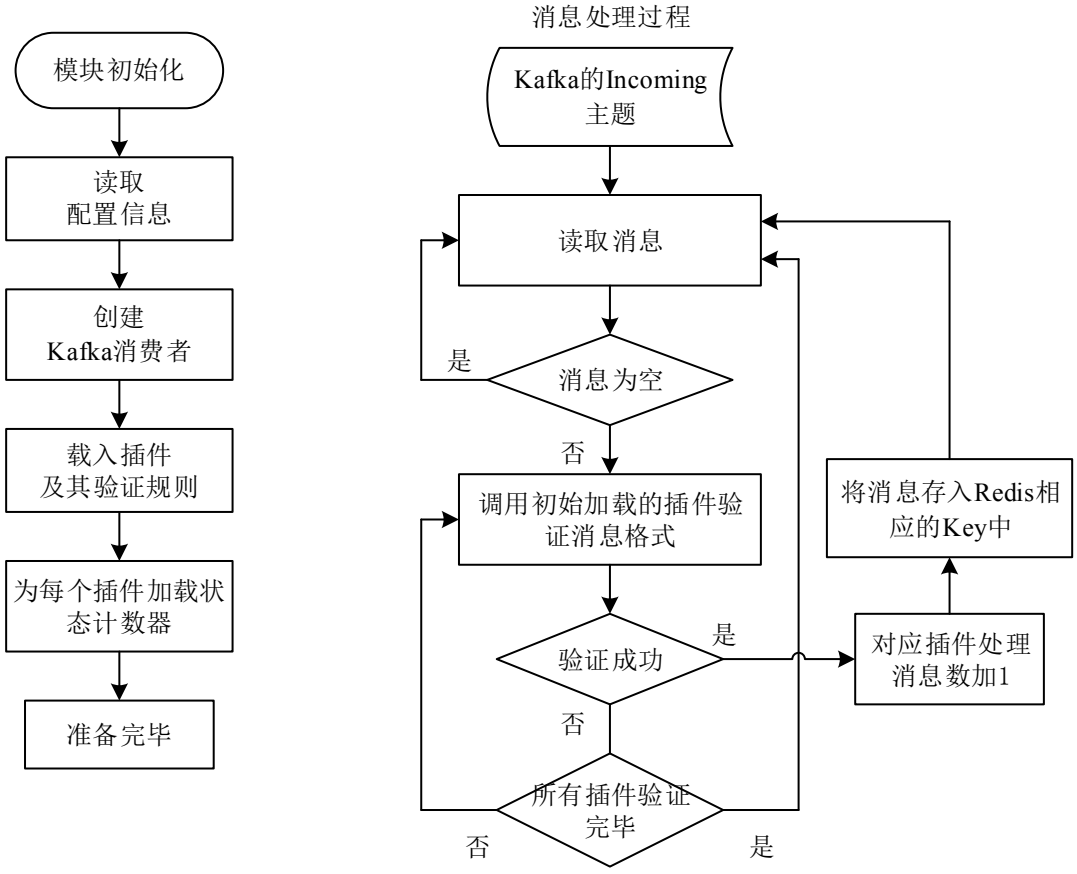

图 4-3 消息预处理模块流程图

4.1.2.1 请求分类

请求分为四类：爬虫抓取请求，爬虫任务状态查询请求，模块状态查询请求以及站点配置信息更新请求，为了区分不同的用户请求，系统默认规定了请求中必须附带的参数及类型。

（1）爬虫抓取请求，顾名思义该请求主要负责提供爬虫抓取任务的相关信息，表 4-1 中展示了爬虫抓取请求所需参数的相关信息。

表 4-1 爬虫抓取请求相关信息

参数	值类型	描述
userid（必须）	string	唯一标识该用户
crawlid（必须）	string	唯一标识该抓取请求
url（必须）	string	待抓取的网址
rule（必须）	object	待抓取网址规则
spider_type（必须）	string	具体类型的爬虫
maxdepth（可选）	integer	抓取网页深度
priority（可选）	integer	请求优先级
cookie（可选）	object	模拟用户登录

（2）爬虫任务状态查询请求主要负责获取指定用户下某个爬虫抓取任务当前处理的情况。如某个爬虫抓取任务当前还有多少个 url 待抓取，或该爬虫抓取任务还需抓取多少个站点，该类请求需要传入 userid、crawlid、uuid 参数的同时传入 action 参数，如果请求中包含这类参数则确定该请求为爬虫查询请求。爬虫查询请求返回的具体信息在 redis 监控器模块中展示。

（3）模块状态查询请求主要负责获取集群中各模块的当前状态及相关信息，该类请求需要传入 userid、stats 以及 uuid 三个参数，其中 stats 字段的值为用户指定的系统模块名，表示用户想查询该模块的当前状态及相关信息。

（4）站点配置信息更新请求主要负责更新集群中爬虫对某些站点访问的限制信息，如将某些站点加入抓取黑名单、限制爬虫对某些站点的访问频率。表 4-2 中展示了站点配置信息更新请求所需参数的相关信息。

表 4-2 站点配置信息更新请求

参数	值类型	描述
userid（必须）	string	唯一用户 id
action（必须）	string	请求的目的
domain（必须）	string	站点域名
rate（可选）	integer	限速频率

4.1.2.2 消息验证及存储

消息预处理模块为每类请求设置了不同的插件进行处理，每个从 Kafka 消息队列中获取的消息都会依次通过各个插件的 JSON Schema 文件进行数据验证，不同的 JSON Schema 文件规定了不同请求所必要的参数及类型，当验证成功后再交由对应的插件进行处理。Crawler 插件负责爬虫抓取请求，该插件将此类型请求存放到 Redis 中以“seeds”为 key 的有序集合，其中请求中 priority 字段的值对应该请求在集合中的分数（即将爬虫抓取请求按照优先级依次存放到初始任务种子队列中）；Query 插件负责爬虫任务状态查询，该插件将此类型请求存放到 Redis 中以字符串为类型的 key 中，其中 key 为请求中的 action，crawlid 以及 userid 字段的值组合而成；Stats 插件负责模块状态查询请求，该插件同样将此类型请求存放到 Redis 中以字符串为类型的 key 中，其中 key 为请求中的 stats、userid 字段的值和’statsrequest’字符组合而成；Zk 插件负责站点配置信息更新请求，该插件将此类型请求存放到 Redis 中以字符串为类型的 key 中，其中 key 为请求中 action、domain 以及 userid 字段的值和“zk”字符组合而成。图 4-4 展示了消息处理模块整体图。

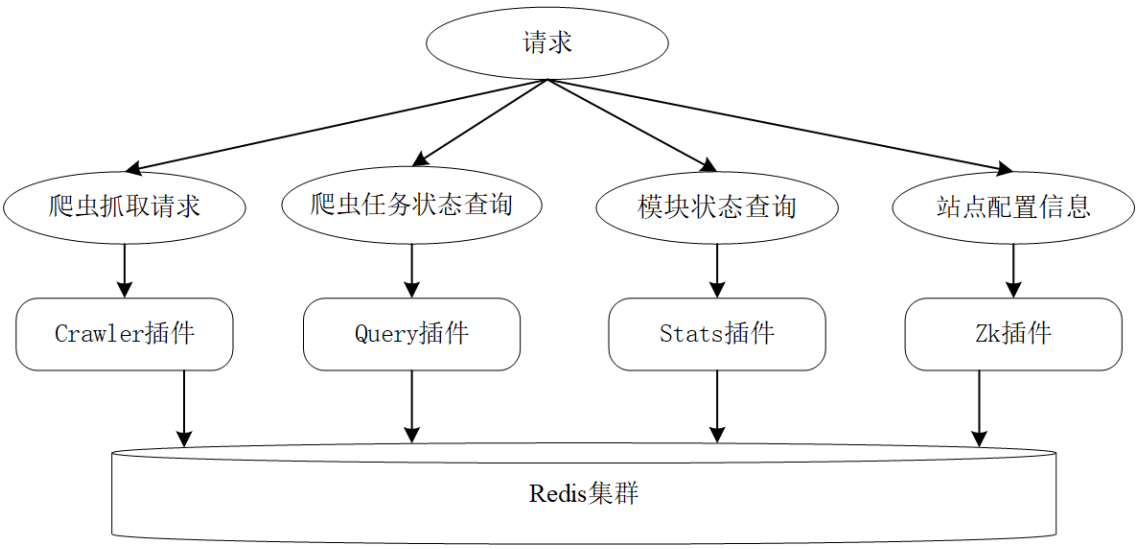

图 4-4 消息处理模块整体图

4.1.3 Redis 监控模块

Redis 监控模块作为 Kafka 消息系统中的生产者，负责周期性的检测 Redis 集群中是否有从消息预处理模块中传递过来的请求，并调用不同的插件来处理解析不同的请求，最后将处理结果返回给 Kafka。该模块主要处理来自 Redis 集群中的爬虫任务状态查询、模块状态查询请求及站点配置信息更新请求（除了爬虫抓取请求），为了确保每个插件能迅速找到其对应的请求，程序在每个插件中定义了一个正则表达式用以匹配 Redis 中的 key，如果匹配成功，则使用该插件处理解析，否则使用其它匹配成功的插件处理解析。模块具体流程图如图 4-5 所示。

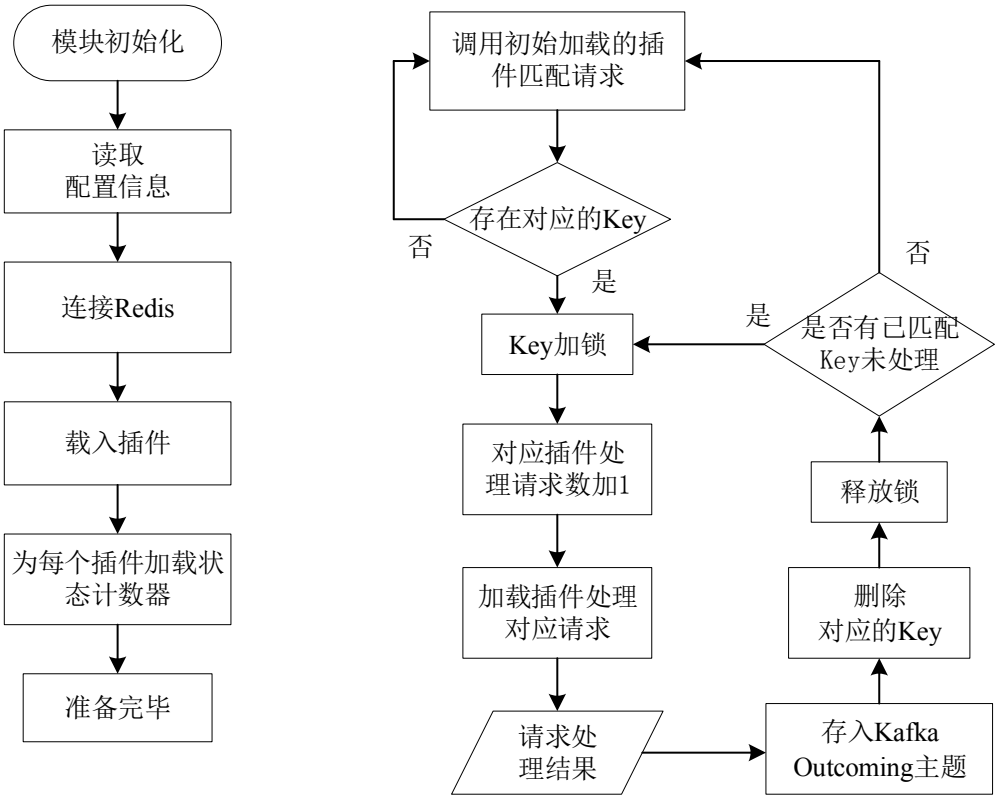

图 4-5 Redis 监控模块流程图

4.1.3.1 Info 插件

Info 插件设置的正则表达式为 info:*:*，该表达式匹配 Redis 中以 info 开头的 key 值，故该插件用于处理解析爬虫任务状态查询请求。

handle 方法是 Info 插件的核心方法，该方法用于统计指定用户下某个抓取任务当前处理的情况，该方法执行步骤如下：

1. 创建并初始化任务状态记录’info’，该记录为 Python 中的字典类型。
2. 解析出爬虫查询请求中 userid 以及 crawlid 的值。

3. 依次遍历爬虫任务队列中所有的抓取请求，如果请求中 userid 和 crawlid 字段的值和步骤 1 中查询请求 userid 和 crawlid 字段的值相同，跳转到步骤 3；否则继续遍历。
4. 继续提取出抓取请求中 domain 字段的值，如果该值未在记录 info 的 domains 字段中出现，则将其信息加入 info 记录中，表示当前任务中还有'xxx'域名的网址还未抓取。
5. 统计出所有符合步骤 3 中抓取请求的个数，并赋值给记录 info 中的 pending 字段，表示当前任务还有多少请求未抓取完。
6. 统计完相关信息后，调用父类的 send_to_kafka 方法将统计的信息传递给 Kafka 消息系统。

表 4-3 展示了爬虫任务状态查询请求返回的具体字段信息。

表 4-3 统计信息字段

字段	描述
userid	具体用户
crawlid	具体的任务号
pending	待抓取请求个数
total_domains	待处理域名个数
domains	具体域名相关信息

4.1.3.2 Stats 插件

Stats 插件设置的正则表达式为 statsrequest:*:*，该表达式匹配 Redis 中以 statsrequest 开头的 key 值，故该插件用于处理解析模块状态查询请求。

模块状态查询请求中 stats 字段表明了用户想查询的模块名，该字段目前提供的可选值为：消息预处理模块、Redis 监控模块以及爬虫模块。

（一） 消息预处理模块

4.1.2 小节已经详细介绍了消息预处理模块中各插件的功能，其中每个插件都绑定以一个状态计数器来记录其单位时间内处理的请求数，该状态计数器以有序集合的形式存在于 Redis 数据库中，表 4-4 展示了与每个插件绑定的计数器在 Redis 中对应的 key 值。

表 4-4 各插件对应计数器相关 key

序号	key	类型
1	stats:info-monitor:QueryHandler:60	zset
2	stats:info-monitor:CrawlerHandler:60	zset
3	stats:info-monitor:StatsHandler:60	zset
4	stats:info-monitor:ZkHandler:60	zset
5	stats:info-monitor:192.168.1.222	string

由表 4-4 可知，前 4 个序号的 key 由 4 个部分组成，第一部分为固定 stats，表明该 key 记录与状态相关的信息；二三部分为模块名及插件名，表明该 key 记录哪个模块对应哪个插件的相关信息；第四部分为时间段（单位：秒）表明该 key 记录的信息是最近 x 秒统计的。如序号 1 中的 key 表明消息预处理模块中 Query 插件在最近 60 秒处理了多少个爬虫查询请求。

序号为 5 的 key 由 3 个部分组成，第一部分为固定值'stats'表明该 key 记录与状态相关的信息；第二部分为模块名，表明该 key 记录哪个模块的相关信息；第三部分为 IP 地址，表示在哪台机器上运行的模块。如序号 5 号中的 key 在 Redis 中存在则表明 ip 为 192.168.1.222 的主机上消息预处理模块正常工作，否则出现异常。

（二） Redis 监控模块

同理，表 4-5 展示了 Redis 监控模块中各插件对应的状态计数器在 Redis 中的 key 值。

表 4-5 与 Redis 监控模块相关的 key

序号	key	类型
1	stats:state-monitor:InfoHandler:60	zset
2	stats:state-monitor:StatsHandler:60	zset
3	stats:state-monitor:ZookeeperHandler:60	zset
4	stats:state-monitor:192.168.1	string

表 4-5 中统计的信息与表 4-4 统计的信息大致相同，由于篇幅有限，这里不再具体解释。

（三） 爬虫模块

表 4-6 展示了 Redis 数据库中与爬虫模块相关的 key 及对应 value 类型。

表 4-6 与爬虫模块相关的 key

序号	key	类型
1	stats:crawler:list:200:60	zset
2	stats:crawler:list:400:60	zset
3	stats:crawler:all:200:60	zset
4	stats:crawler:all:400:60	zset

由表 4-6 可知，与爬虫模块相关的 key 由 4 个部分组成，第一二部分为固定值 stats 及 crawler，表明该 key 记录与爬虫状态相关的信息；第三部分为爬虫具体类型，表明该 key 记录哪种类型的爬虫相关信息；第四部分为 http 响应状态码，表明该 key 为请求响应为该值的相关信息，第五部分为具体时间段（单位：秒），表明该 key 记录的信息是最近 x 秒统计的。如序号 1 中的 key 表明爬虫模块中 list 类型的爬虫在最近 60 秒内处理了多少个响应状态码为 200 的请求。

4.1.3.3 Zookeeper 插件

Zookeeper 插件设置的正则表达式为 zk:*:*:*，该表达式匹配 Redis 中以 zk 开头的 key 值，故该插件用于处理解析站点配置信息更新请求。

配置信息更新请求中 action 字段表明了用户希望更新的信息，该字段有两个可选值：rate 和 blacklist，rate 表示用户希望更新集群中爬虫对某个站点的访问频率；blacklist 表示用户希望增加集群中网站的黑名单。

插件首先调用 setup 方法建立与 zookeeper 集群连接，并根据相应路径找到 zookeeper 中存放配置信息的节点，如果 action 字段的值为'rate'，则直接更新对应站点的访问频率，如果 action 字段的值为'blacklist'，则直接在网站黑名单中添加请求中 domain 字段的值即可，更新完毕后将成功信息返回给 Kafka 消息队列。图 4-6 展示了 Redis 监控模块请求处理整体图。

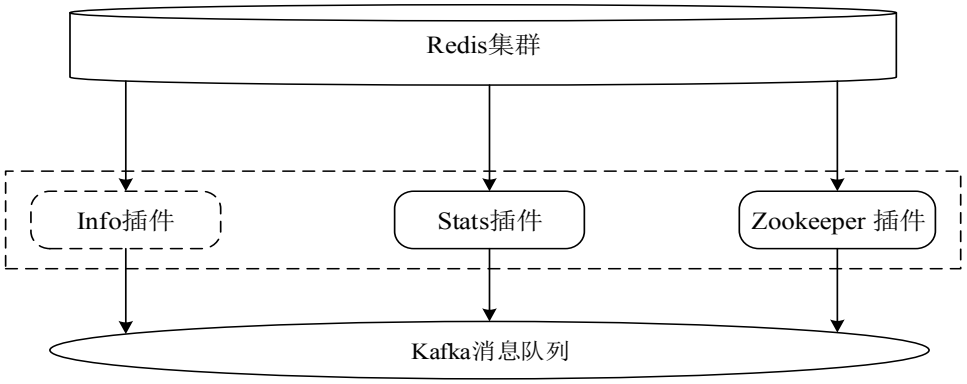

图 4-6 Redis 监控模块整体图

4.2 主节点设计实现

在主从式架构模式中，系统由一个主节点和若干个分别部署并且能和主节点互通的从节点构成，主节点负责整个系统中爬虫抓取任务的分发工作、集群中爬虫节点对站点访问限速工作以及抓取过程中 URL 去重工作。该节点由任务调度器，限速器以及过滤器三个模块组成，各模块具体实现如下：

4.2.1 任务调度器

任务调度器负责整个系统中爬虫抓取任务的分发工作。在数据采集过程中各个爬虫节点性能的不一致和采集任务量的不确定性都可能使得分布式网络爬虫在并行采集页面时存在着不平衡的现象，该现象主要体现在爬虫节点之间任务分配不均衡。一个好的任务调度策略应该能根据爬虫节点当前状态动态地调整任务的分配，最大化的利用系统资源，实现节点间动态负载均衡。

4.2.1.1 任务调度策略

为了保证任务调度器能实时根据爬虫节点当前状态均衡的分发爬虫抓取任务，本文提出了一种动态反馈任务调度策略对总爬虫任务队列中的爬虫抓取任务进行分发，该调度策略的主体是传统的加权轮询调度算法[38]，同时系统在主节点中维护了一张爬虫权值表，通过周期性的统计各个爬虫节点当前权值，保证算法将基于爬虫节点最新的权值进行任务分配。具体分配步骤如下：

1. 载入各个爬虫节点最新的权值。
2. 获取上一次分配爬虫任务的节点位置，并选取其下一个爬虫节点与当前调度值比较。
3. 如果该爬虫节点权值大于当前调度权值，分配一个爬虫任务给该节点并更新其权值，否则跳转到步骤 4。
4. 选取下一个爬虫节点进行遍历，如果当前节点为首节点，意味着已经轮询完一次所有的爬虫节点，跳转到步骤 5，否则跳转到步骤 3。
5. 将当前调度权值自减一个步长（当前爬虫节点权值最大公约数），并判断其值是否小于 0，如果不小于 0，跳转到步骤 3，否则跳转到步骤 6。
6. 将当前调度权值重置为当前各个节点权值的最大值，并跳转到步骤 3。

图 4-7 为该调度策略的流程图。

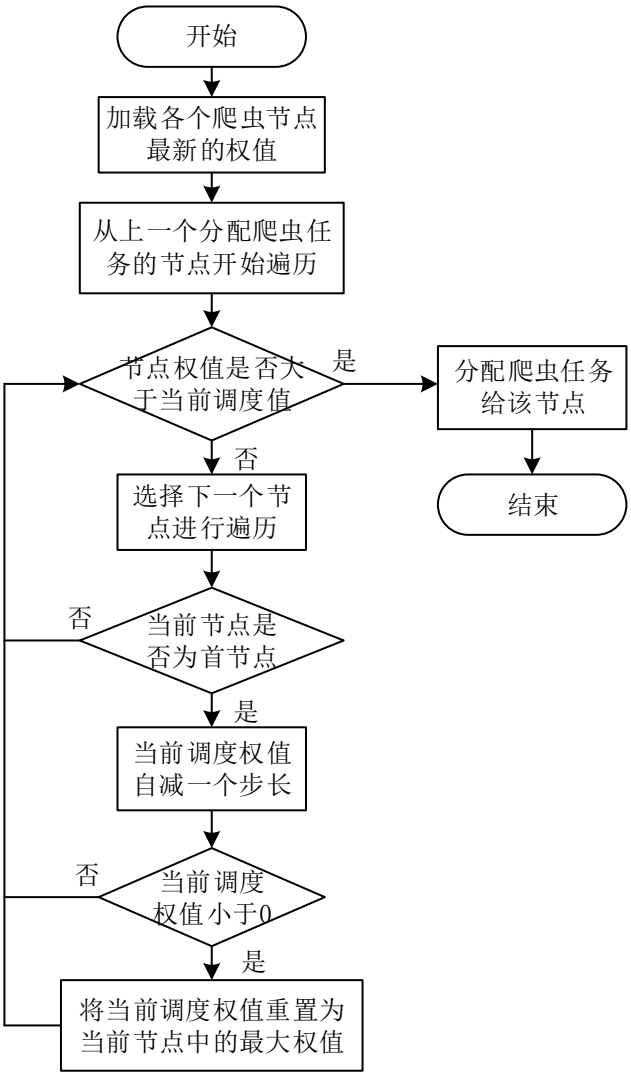

图 4-7 任务调度策略流程图

4.2.1.2 爬虫节点权值计算

为了使分布式爬虫系统整体达到负载均衡的效果，拥有客观准确的负载评价标准是基本前提。影响负载均衡的因素可以从很多方面考虑，比如 CPU 性能、内存使用率、网络带宽等，但归根结底还是体现在爬虫整体的采集速度上，因此本系统将爬虫节点单位时间内完成的任务数来作为衡量爬虫节点自身性能的标准。

假设某个爬虫节点 t 分钟内完成了 n 个抓取任务（一次抓取任务的完成包括从主节点中接收到任务至从网页中抽取出数据存入 MongoDB 数据库中），那么该爬虫节点在这 t 分钟内采集速度为：

$$\overline{v} = \frac{n}{t} \tag{4-1}$$

注意到公式 4-1 中，随着 t 和 n 的不断增大，其比值会趋向于稳定，这样爬虫节点

在某段时间里采集速度的变化就无法在公式中体现，故系统借鉴滑动窗口的概念，只统计最近 k 分钟内（系统默认 k=10，可根据实际情况调节）爬虫完成的抓取任务数，n_i 代表最近 i 分钟内完成的抓取任务数，故最终的权值计算公式为：

$$w = \frac{\sum_{i=1}^{k} n_i}{k} \tag{4-2}$$

在实际情况下，任务调度器分发任务的速度极快，单个爬虫节点的采集速度通常维持在每分钟 840 个网页左右，假设目前总爬虫任务队列中维护了 10 万左右的抓取任务，那么任务调度器可能在 2-3 分钟就能将所有的抓取任务分发到各个爬虫节点的任务队列中，而当爬虫节点在 10 分钟或 20 分钟左右采集速度发生改变时，总爬虫任务队列中已经没有了爬虫抓取任务，任务调度器也就无法根据节点最新的采集速度分配对应的任务量。鉴于此原因，系统采用按时间段任务分配策略来进行任务的分发，假设单个爬虫节点最大采集速度为每分钟 1200 个网页，那么 10 分钟内爬虫节点最多采集 1.2 万个网页，如果系统中存在三个爬虫节点则最多采集 3.6 万个网页，那么任务调度器按照每 10 分钟 3.6 万的分发量来分发任务，超出此任务量就不在分发，这样既能保证各个爬虫节点时刻处于“工作”的状态，又能使任务调度器能动态的根据爬虫节点采集速度变化调整相应的任务分配。

4.2.1.3 工作流程

本节将结合代码来详细说明任务调度器的工作流程。Dispatcher 类作为任务调度器的核心类，其包含以下主要成员变量及方法：

```
redis_conn: # 与 Redis 数据库的连接
cur_spider_nodes: # 当前集群中运行的爬虫节点
last_spider_nodes: # 上一次集群中运行的爬虫节点（爬虫节点宕机前）
spiders_ weights: # 当前集群中各爬虫节点权值
schedule_tasks(): # 负责任务的分配
get_spiders_speeds(interval): # 负责周期性的获取爬虫节点采集速度并更新其权值
check_spiders_change(interval, is_first): # 负责检测集群中爬虫节点是否变化
run(self): # 负责启动任务调度器
```

首先调用类 Dispatcher 的 run 方法启动任务调度器，并分别创建两个后台线程 t1 和 t2，t1 线程周期性的执行 get_spiders_speeds 方法，从 Redis 数据库中获取各爬虫节点当前采集速度，并更新其对应的权值；t2 线程周期性的执行 check_spiders_change 方法，检测当前爬虫节点数是否有变化（增添新的爬虫节点或旧的节点宕机），如果集群中爬虫节点数增加，则调用 get_spiders_speeds 方法将

新加入的爬虫节点权值更新到 spiders_weights 中；如果集群中有爬虫节点宕机，则先将其任务队列中存放的任务返回到总爬虫任务队列中，然后从 spiders_weights 中删除其对应的权值。

后台线程启动完毕后，调用 schedule_tasks 方法对总爬虫任务队列中的任务进行分配。首先加载 Robin 对象（该 Robin 类实现了本系统的任务调度算法），接着调用 is_max 方法判断最近 10 分钟内分发的任务量是否超过了当前所有爬虫节点最大的采集量，如果没有超过，则将当前集群中爬虫节点最新权值 spiders_weights 传递给 Robin 对象的 choose_spider 方法并选择出一个爬虫节点分配爬虫任务，以此循环直至总爬虫任务队列中的任务分发完毕，schedule_tasks 方法核心代码如下：

```
1.  robin_all = Robin() # 加载任务调度算法
2.  while True:
3.      if not self.is_max():
4.          task_json = self.redis_conn.lpop('seeds') # 获取爬虫任务
5.          task = pickle.loads(task_json)
6.          url = task['url']
7.          spider_type = task['spider_type']
8.          domain = tldextract.extract(url).domain
9.          spiders_weights = copy.deepcopy(self.spiders_weights)
10.         # 分配爬虫任务给爬虫节点
11.         job = robin_all.choose_spider(spiders_weights)
12.         # 获取爬虫节点待抓取队列在 Redis 中对应的键名
13.         queue_key = '{job}:{domain}:queue'.format(job=job, domain=domain )
14.         priority = task['priority']
15.         self.redis_conn.zadd(queue_key, pickle.dumps(task), priority)
16.     else:
17.         time.sleep(10)
```

4.2.2 限速器

限速器负责控制集群中爬虫节点对站点访问速度的工作。随着爬虫节点的增多，各个爬虫节点频繁的向网站发送请求不仅占用了大量网络带宽同时也增加了站点服务器的处理开销，在很多成熟的网站中都会在其根目录下放置 robots.txt 文件，该文件声明了网站中不想被网络爬虫抓取的信息或规定其访问的速度。为了使分布式爬虫系统设计的更加合理，本文设计并实现了一种基于 Redis 的限速策

略，通过该策略能控制集群中的爬虫在固定的时间段内访问某个网站的频率。

4.2.2.1 任务队列

为了能限制集群中的爬虫节点对网站的访问频率，系统将主节点中各个爬虫节点的任务队列进行了更进一步的划分，图 4-8 展示了各个爬虫节点的任务队列在主节点中的存放形式。

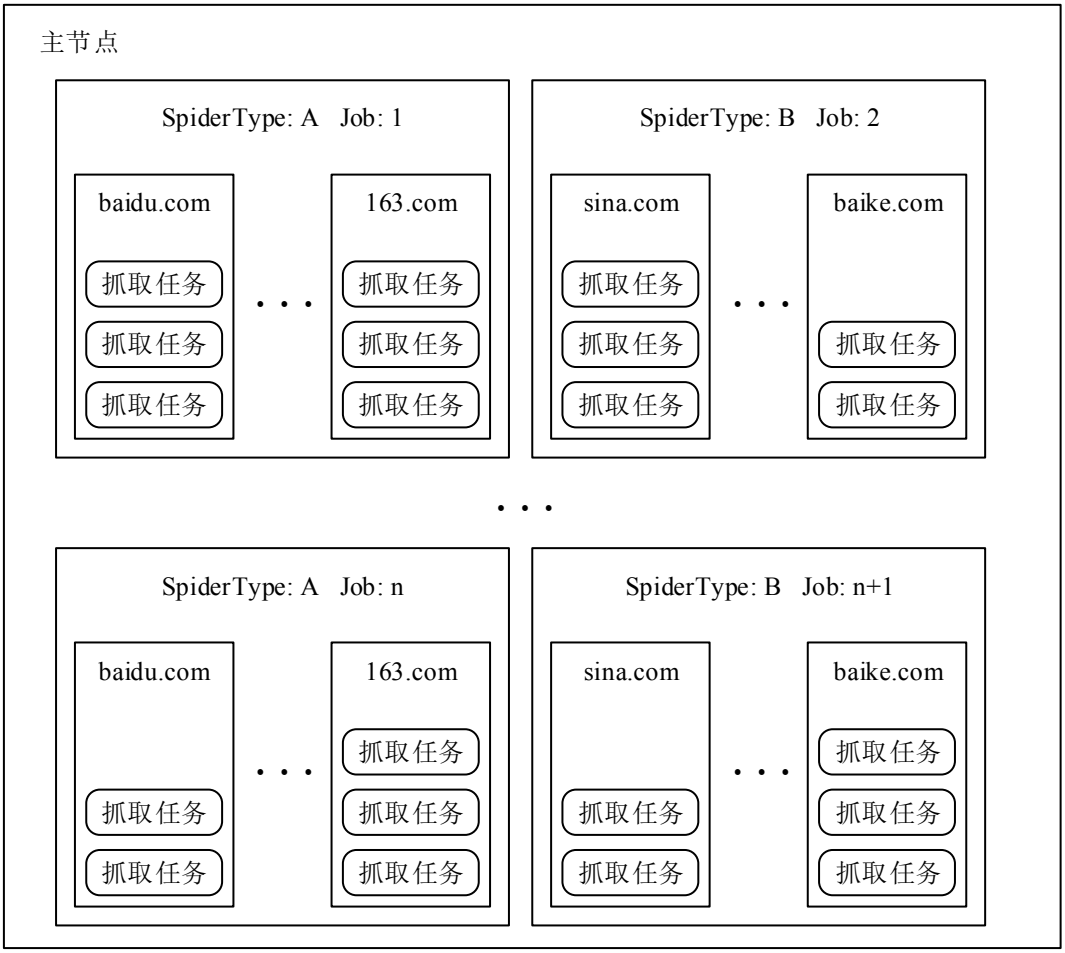

图 4-8 爬虫任务队列存放图

在图 4-8 中，SpiderType 代表 Scrapy 爬虫的类型，不同类型的 Scrapy 爬虫负责不同类别的网站，Job 号作为唯一标识某个 Scrapy 爬虫进程。同时主节点中每个爬虫节点的任务队列都按照了不同的域名进一步划分，如 Job 号为 1 的 Scrapy 爬虫将任务队列划分为域名为“baidu.com”的爬虫队列以及域名为“163.com”的爬虫队列。划分出域名队列后，系统就能通过限制集群中 Scrapy 爬虫获取域名队列中抓取任务的速度来控制其访问站点的频率了，本文基于此提出了两种方式的限速策略，一种为基于爬虫类型的限速，该策略针对不同主机上相同类型的 Scrapy

爬虫对某网站的访问限速；另一种为基于 ip 的限速，该策略针对同一主机上不同类型的 Scrapy 爬虫对某网站的访问限速。

4.2.2.2 基于爬虫类型的限速

为了限制不同主机上相同类型的爬虫对某网站的访问频率，系统在域名爬虫队列外部再构建了一个限速器，图 4-9 展示了针对“163.com”以及“sina.com”域名限速的原理图。

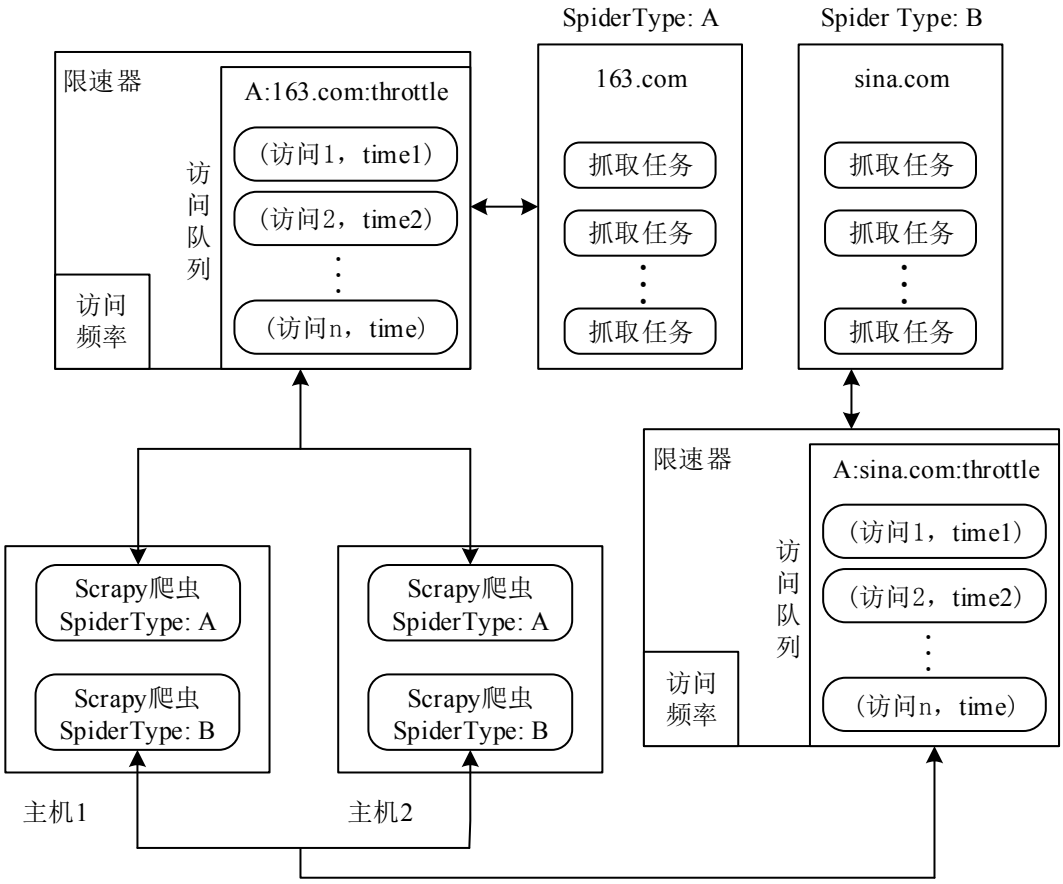

图 4-9 基于爬虫类型的限速原理图

如图 4-9 中存在两台主机，主机 1 和主机 2 中分别运行着一个类型为 A 的 Scrapy 爬虫和一个类型为 B 的 Scrapy 爬虫，同时每个域名任务队列都对应着一个限速器，该限速器的访问队列基于 Redis 数据库的有序集合实现，其 key 值为 {SpiderType}:{domain}:throttle，当主机 1 中或主机 2 中 A 类型的 Scrapy 爬虫想获取“163.com”域名下的抓取任务时，首先访问 key 为 A:163.com:throttle 的访问队列，如果该队列最近 x 秒的请求数已经超过了系统设置访问频率，则不再分配该域名下的抓取任务给 Scrapy 爬虫，如果没有超过，则从域名队列中按任务优先级

的顺序取出一个抓取任务交给 Scrapy 爬虫，并将这次请求记录计入对应的访问队列中，通过这样的方法就能限制不同主机上相同类型的爬虫对某网站的访问频率了。

4.2.2.3 基于 IP 的限速

基于 IP 的限速是限制同一主机上不同类型的爬虫对某网站的访问频率，图 4-10 展示了针对“163.com”以及“souhu.com”域名限速的原理图。

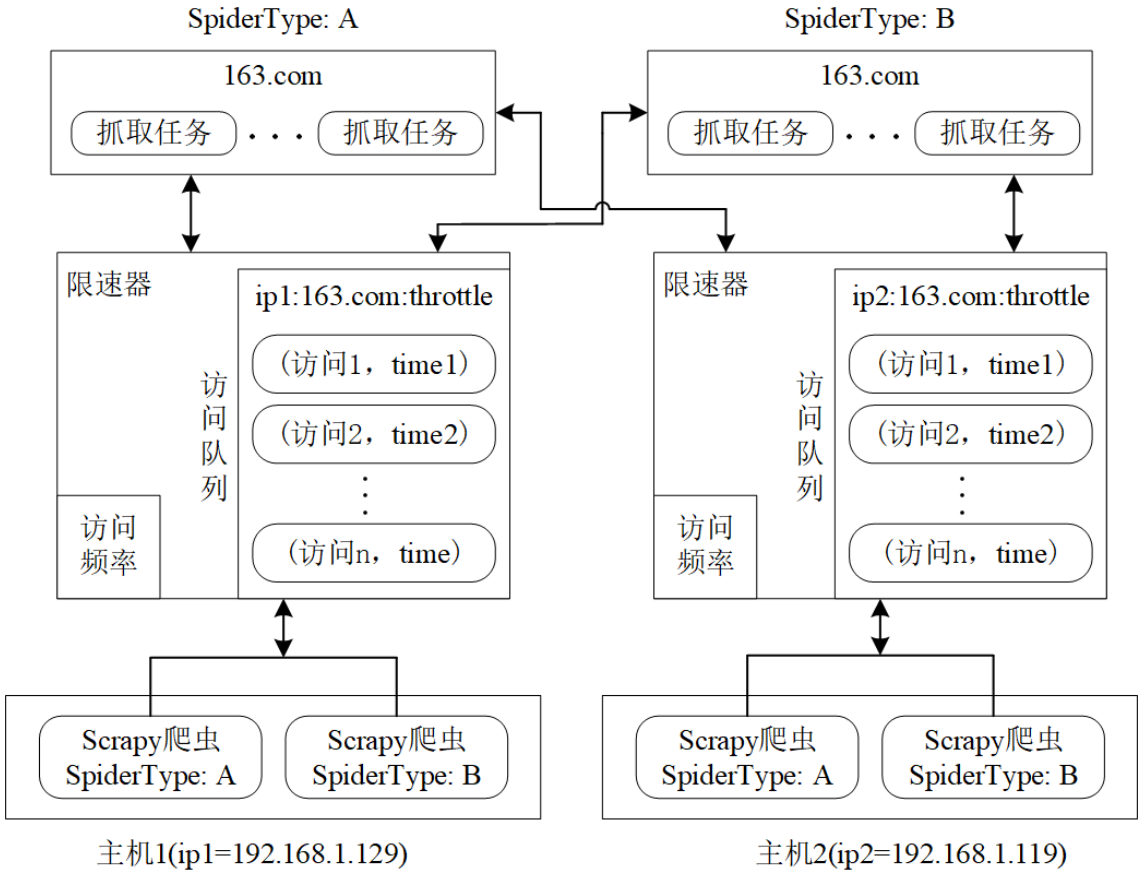

图 4-10 基于 IP 的限速原理图

基于 IP 的限速本质也和基于爬虫限速的原理一致，只是限速方从不同主机相同类型的爬虫转化为同一主机上的不同类型的爬虫。当 ip 为 192.168.1.129 上的 A 类型爬虫或 B 类型爬虫想获取域名为“163.com”的抓取请求时，首先需访问 key 为 192.168.1.129:163.com:throttle 的访问队列，如果该队列最近 x 秒内集合中请求的个数已经超过系统设置的最近 x 秒内访问“163.com”的最大次数，则不再分配请求给该爬虫，否则从对应的爬虫队列中分配一个抓取请求给该爬虫，并将此次请求记录存入访问队列中。通过此方法就能限制相同节点上不同类型的爬虫对某

网站的访问频率了。

4.2.3 过滤器

过滤器负责对集群中各个爬虫节点抓取的 URL 去重。在 URL 遍历过程中，常常需要判断某个 URL 是否已经被采集过，最简单的方法是把已经采集过的 URL 存放起来，当有新的 URL 需要采集时，首先到已采集的 URL 队列中查找该 URL 是否已经采集过，如果已经采集过则丢弃，否则加入到待采集 URL 队列中，从而避免网页的重复采集，造成资源的浪费。为了加快信息的查找速度和系统的吞吐量，一般会将数据加载到内存中然后再执行相应地操作，但通常采集的 URL 数量都在百万或千万级别左右，想把所有的数据完全都存放在内存中是不可能的，所以更需要一种节约空间且查找速度快的算法出现。

4.2.3.1 布隆过滤器算法

布隆过滤器（Bloom Filter）是巴顿布隆于 1970 年提出的一种基于多个哈希函数映射压缩参数空间的数据结构[39]，它由一个很大的二进制向量组和若干哈希函数构成。由于布隆过滤器自身是一种基于哈希函数的查找算法，故相比于其它数据结构，布隆过滤器在时间和空间方面都有巨大的优势，鉴于此优势，布隆过滤器常常用作海量数据集合的查找。

下面通过一个具体的例子来展示算法的工作流程。假定布隆过滤器位向量数组初始长度为 n，位元素初始值为 0，待存储元素集合 $P=\{p_1, p_2, p_3, \ldots, p_n\}$，以及 m 个相互独立的哈希函数 $K=\{k_1,k_2,\ldots,k_n\}$，依次取出待存储元素集合 P 中的元素，分别使用 k 个独立哈希函数对取出的元素进行计算生成 k 个随机的哈希值，并将生成的哈希值对位数组长度 n 取模找到其对应存放的位置，最后在当前位上的元素的值置为 1。如需要查询当前元素是否存在于集合中，仅需将待查询的元素用相同的哈希函数组进行一一映射，将得到的值与位阵列中存储的值依次相比较，如果当前所有位中的值都为 1，则表明此元素可能已经存在于集合中，否则则表明此元素不在集合中，元素映射过程如图 4-11 所示。

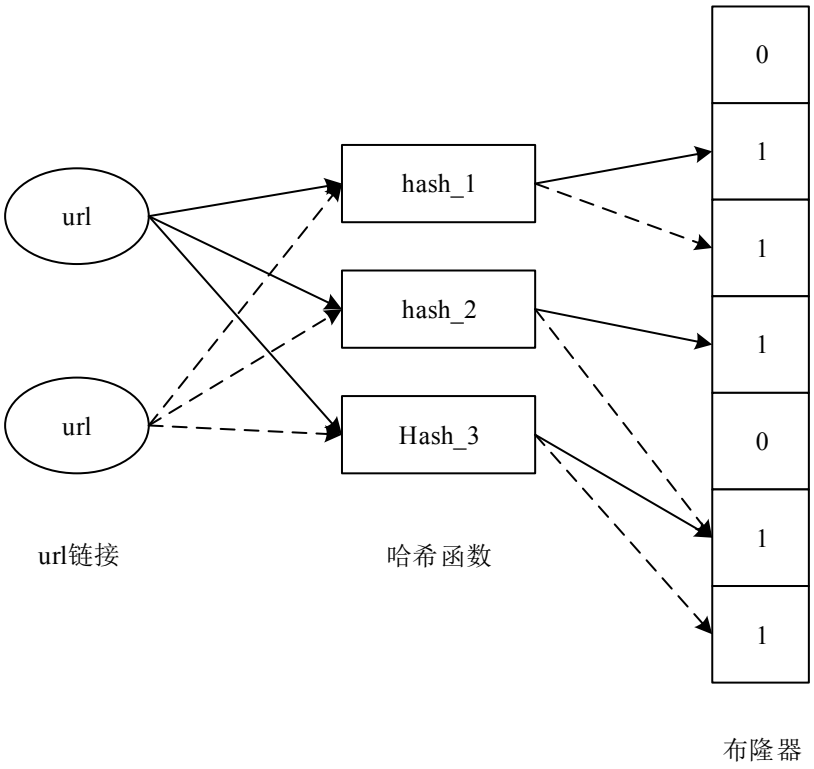

图 4-11 布隆过滤器映射过程

由于布隆过滤器算法在存储海量数据时，不需要存储元素本身，因此大大减少了空间的占用，同时由于算法本身基于散列函数，对元素的查找时间复杂度始终为 $O(m)$。但布隆过滤器的缺点和优点一样明显，由于算法在进行集合元素表示时，通过 k 个散列函数使位向量相应位置为 1，经过多次集合元素增加操作后，使得位向量若干位被重复置为 1，对于已经映射到集合中的元素，显然可以通过集合查找运算判定其存在于集合中，但对于尚未映射到集合中的元素，可能存在误判，随着存入元素的个数增多，其误判率就越大，因此它实际上是通过牺牲一定的准确率来获得较高的空间效率和时间效率的算法。

4.2.3.2 基于多组布隆算法下的网页去重

传统的布隆过滤器由一组哈希函数和一个很大的位数组构成，由于散列函数的随机性，可能使得某个元素本来不属于集合而被判定为集合元素，但在实际的 URL 去重过程中，为了提高数据采集的完整性，降低过滤器所导致的的误判率往往也是一项很重要的工作。

为了降低误判率，同时保证高效的系统性能，本文采用多组哈希函数，将元素的表示和查询分解为多个子集合的并行表示和查询，每组哈希函数及其对应的

位数组仅仅代表元素的某个子集合。如果某个元素存在于集合中，则其任意子集合中哈希映射值对应位上必须为 1，否则判断此元素不在集合中；另外由于布隆过滤器的特殊性，对已存在的元素进行删除时本质上是将位数组中对应位置 0，这样的做法会影响其它元素的判断，本文通过对位数组中每一位进行扩展，原本由 0 或 1 来判定元素存在或不存在，改为由可增减的整形值来对应元素的映射，如某个元素通过哈希映射后，就将对应位中的值进行加 1 操作，反之如需删除某一元素，则将其对应位中的值进行减 1 操作。算法原理图如 4-12 所示。

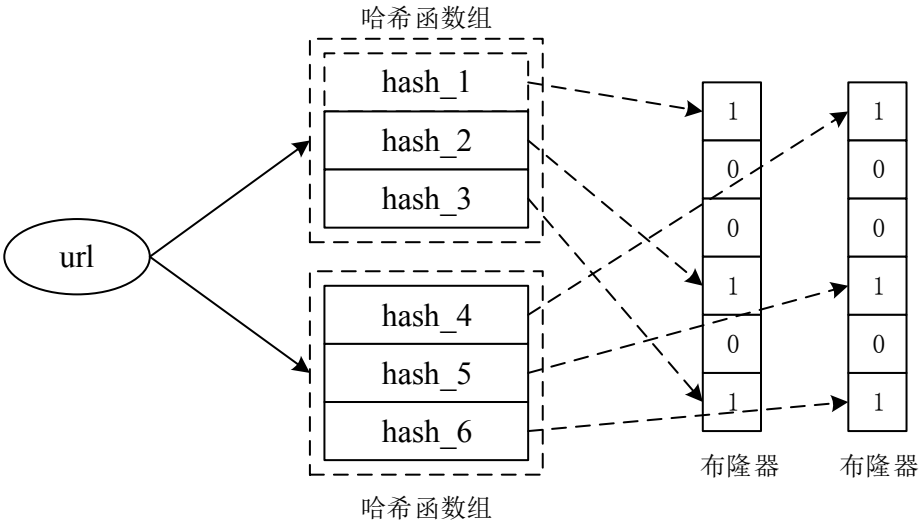

图 4-12 多组布隆映射过程

Redis 作为一款优秀的基于 key-value 结构的内存型数据库，其丰富的数据结构类型正好满足系统的需求。本文采用其 BitMap 类型的位数组实现布隆过滤器中位向量的操作，则基于改进后的布隆算法的网页去重流程如下：

1. 连接 Redis 数据库，初始化系统所需要的 *n* 个位数组。
2. 当爬虫节点提取出新的 URL 后，通过 *n* 组哈希函数（每组 *k* 个哈希函数）生成 *n***k* 个哈希值。
3. 利用布隆查找算法将第二步中生成的 *n***k* 个哈希值在 *n* 个位数组的对应位上匹配。
4. 如果在位数组中的对应位上均大于 1，则表示此 URL 已经采集过了，丢弃掉并进行下一个新的 URL 判断，否则执行第五步。
5. 爬虫节点将此 URL 提交给主节点，并加入到其中的待抓取队列中，同时将 Redis 中的位数组对应位上的值均加 1。

4.2.3.3 误判率分析

本节将通过具体的公式演算来分析和比较两个算法的误判率。假设 h 组哈希函数（每组 k 个哈希函数）相互独立其每个元素都能等概率地被映射到位数组中任何一个位置，而与其它元素被映射到哪个位置无关；位数组长度为 n、元素个数为 m 个，布隆过滤器个数为 h。

由于传统的布隆过滤器只有一组哈希函数，故 h 值为 1，当元素经过哈希函数映射后，位数组中任意一位未被置 1 的概率计算公式为：

$$p_1 = 1 - \frac{1}{n} \tag{4-3}$$

其中 p_1 为位数组中任意一位在一次哈希函数映射后未被置 1 的概率，则经过 k 次哈希函数映射后，仍然未被置 1 的概率计算公式为：

$$p_0 = \left(1 - \frac{1}{n}\right)^k \tag{4-4}$$

当 m 个元素分别经过 k 个哈希函数映射后，该位置未被置 0 的概率计算公式为：

$$p_2 = \left(1 - \frac{1}{n}\right)^{km} \tag{4-5}$$

当要检测某一个新的元素是否在集合中时，只需要将该元素通过 k 个哈希函数映射后检查位数组对应位上是否都为 1，如果映射后对应位上都为 1 的话，则该元素可能存在于集合中，故布隆过滤器误判的概率计算公式为：

$$p = \left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \tag{4-6}$$

而改进后的布隆算法是利用多组哈希函数进行映射，判断一个元素存在于集合中必须保证多个位数组上的对应位都置为 1，即横向扩展传统的布隆过滤器，故改进后的布隆算法误判率计算公式为：

$$p' = p^h = \left(1 - e^{-kn/m}\right)^{kh} \tag{4-7}$$

由于公式 4-7 中 $1 - e^{-kn/m}$ 恒小于 1，随着 h 的增大，故误判率也随之减小。

4.3 从节点群设计实现

从节点群由一个或多个配置相同的从节点组成，每个从节点由一个爬虫管理器和若干个 Scrapy 爬虫进程构成。Scrapy 爬虫负责网页数据的采集工作，爬虫管理器负责主机中各个爬虫的管理工作，本节将依次详细描述这两个模块的设计与实现。

4.3.1 Scrapy 爬虫

Scrapy 爬虫负责具体的数据采集工作，该系统中的中心引擎子模块和下载器子模块是直接复用 Scrapy 框架的自身的组件，而调度器子模块、中间件子模块、数据采集子模块和数据管道子模块是根据系统设计目标重新定制开发的，图 4-13 展示了 Scrapy 爬虫子系统的整体架构及数据流图。

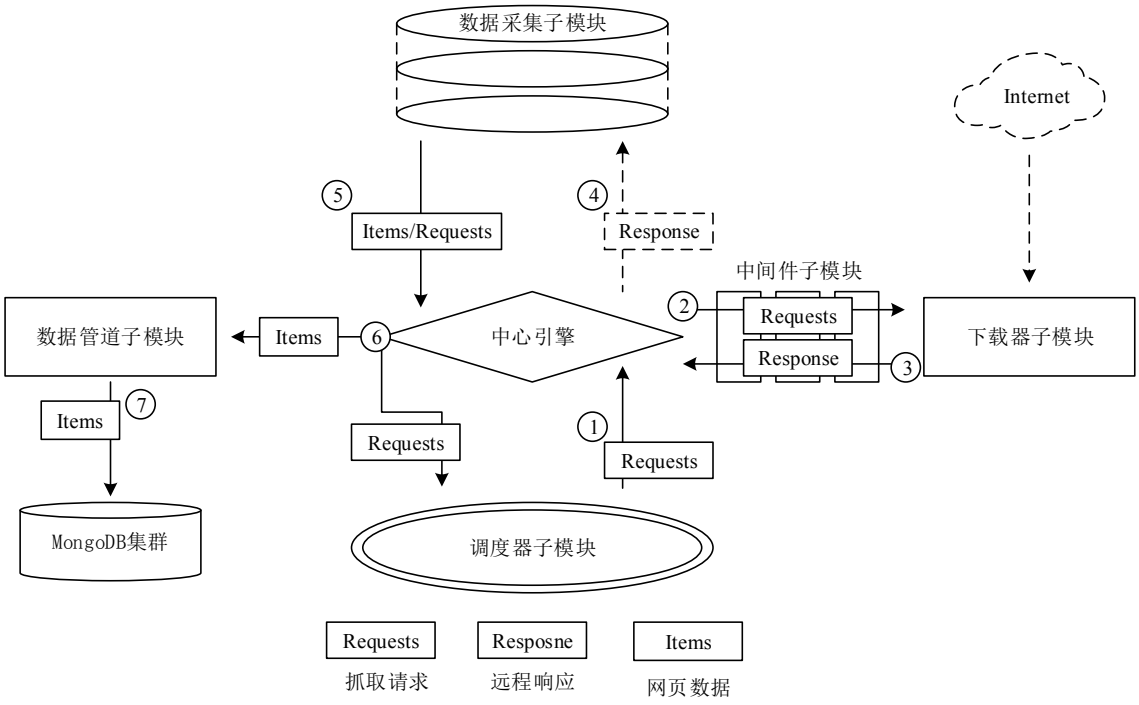

图 4-13 Scrapy 爬虫子系统整体架构及数据流向图

4.3.1.1 调度器子模块

调度器对外负责与主节点交互，对内负责与中心引擎交互。与主节点交互的内容包括：初始化限速器，更新限速器，获取爬虫抓取任务，提交爬虫抓取任务；与中心引擎交互的内容包括：接受来自数据采集子模块中提取出的新的爬虫抓取任务，将从主节点中获取的爬虫抓取任务交付给下载器子模块。

（一） 限速器的创建及更新

4.4.2 小节已经描述过限速器限速机制，本节将主要介绍限速器的初始化及更新过程。图 4-14 展示了 Job 号（唯一标识某个爬虫进程）为 1 的 Scrapy 爬虫的抓取任务队列在主节点中存放的形式。

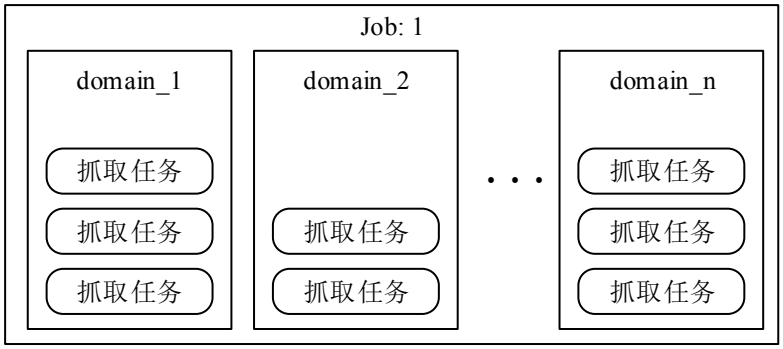

图 4-14 Job 号为 1 的爬虫抓取任务队列

由图 4-14 可知，系统将 Job 号为 1 的爬虫抓取任务队列进一步划分，相同域名下（domain）的抓取任务存放在同一个队列中，队列由 Redis 中的有序集合实现，key 为 1:{domain}:queue。

基于爬虫类型的限速器初始化过程：依次遍历图 4-14 中爬虫所负责的域名队列，并为每个域名创建对应的限速器，限速器由两部分组成：访问频率及访问队列。访问频率记录了由用户设定的不同主机中相同类型的爬虫访问该域名下的网站速度；访问队列由 Redis 中的有序集合实现，key 为 {spider_type}:{domain}:throttle，其中 spider_type 为当前爬虫类型，该队列存放了最近一段时间内 spider_type 类型的爬虫获取 domain 域下抓取任务的记录。图 4-15 展示了基于爬虫类型的限速器初始化过程。

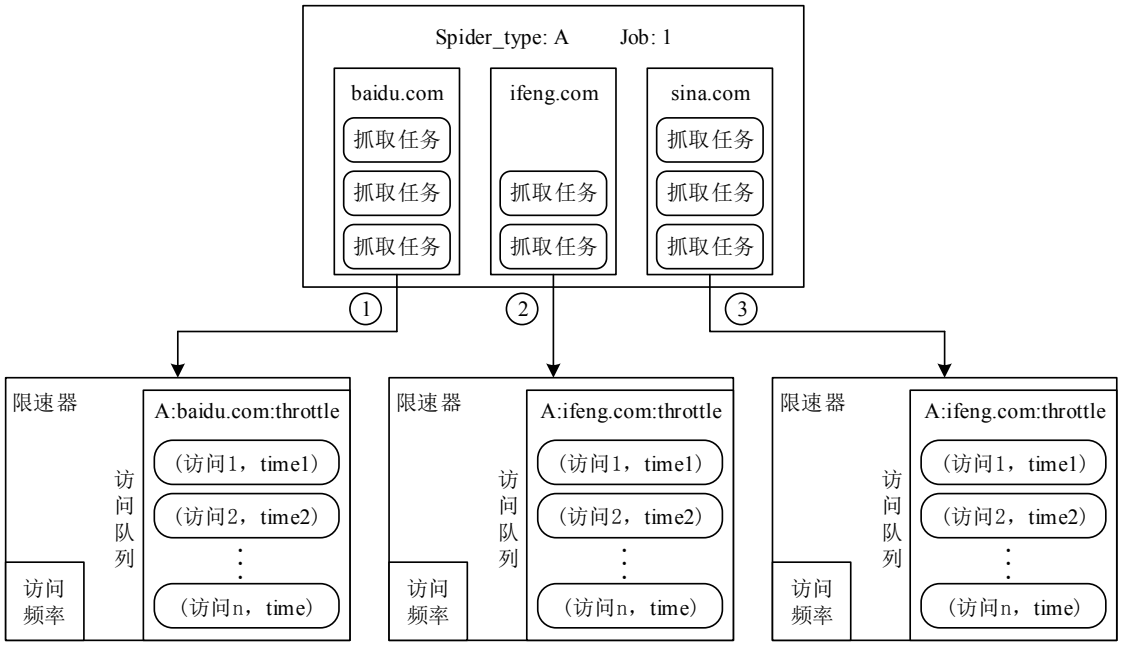

图 4-15 基于爬虫类型的限速器初始化

基于 IP 的限速器初始化过程与上述基本一致，唯一不同的是基于 IP 的限速器的访问队列 key 值为{ip}:{domain}:queue，其中 ip 为当前爬虫所在主机的 ip，队列存放了最近一段时间内该主机 ip 下的爬虫获取 domain 域下抓取任务的记录。

限速器的更新分为两种，一种是站点访问频率的更新，另一种是添加新的站点限速器。为了使爬虫节点能实时感知到用户设定的站点限速信息变动，本文基于 ZooKeeper 的 Watch 机制实现了一个分布式用户配置中心，在不停止爬虫运行的前提下将修改后的配置信息同步到所有的爬虫节点中。图 4-16 展示了用户配置中心整体设计方案。

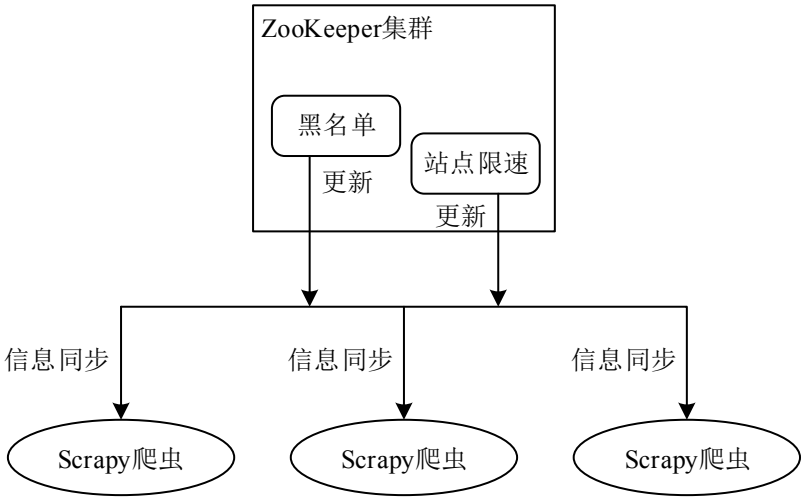

图 4-16 用户配置中心设计方案

整个配置信息存储方案由三部分组成：ZooKeeper 集群服务器、配置信息以及 Scrapy 爬虫。其中配置信息由站点限速和黑名单两部分组成，站点限速配置记录了用户设定的站点访问频率，黑名单配置记录了用户不想爬虫采集的站点。系统默认在 Zookeeper 集群中创建一个节点用于保存与爬虫相关的配置信息，当爬虫启动后首先调用调度器的 setup_zookeeper 方法建立与 ZooKeeper 集群的连接，并在保存配置信息的节点上注册观察者监听配置信息的变化，当配置信息发生变化时立即调用 chang_config 方法更新变量 domain_config（域名及其对应的访问频率）和 black_domains（黑名单）的值，爬虫节点根据 domain_config 中最新的站点限速信息创建新的站点限速器或更新某站点限速器中的访问频率。最后再次向保存配置信息的节点注册观察者监听配置信息的变化（ZooKeeper 的观察者模式为一次性触发器，意味着回调函数触发后观察者也被移除）。图 4-17 展示了整个过程的时序图。

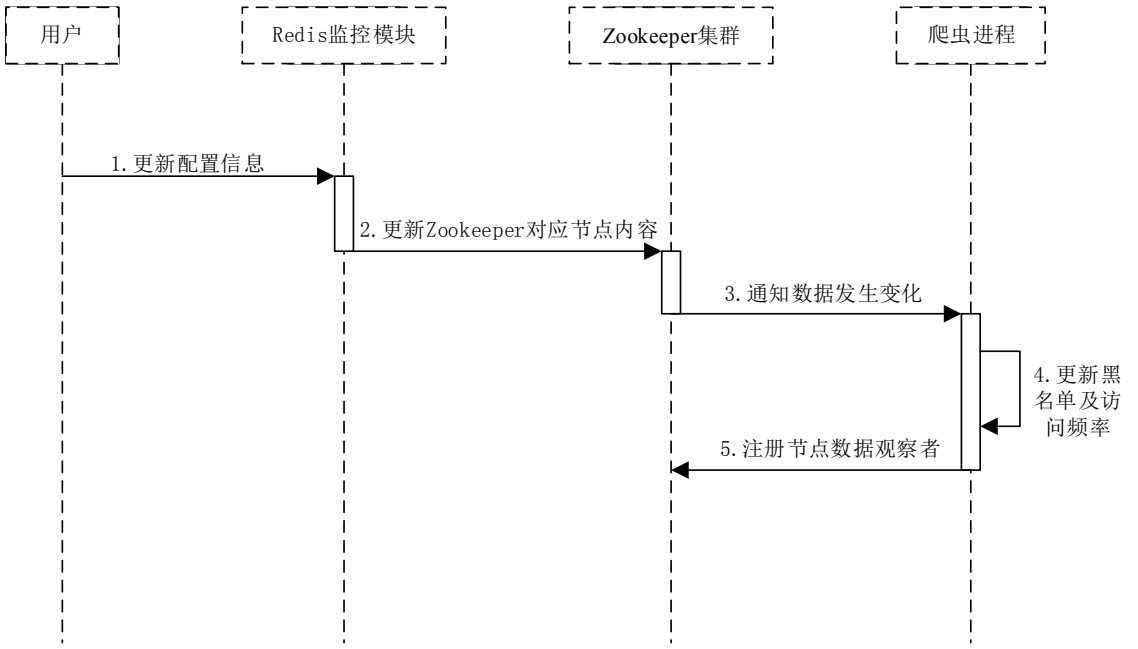

图 4-17 配置信息动态同步时序图

（二） 入队与出队

入队与出队是调度器子模块的核心，它负责从主节点获取抓取任务及向主节点提交新的抓取任务。图 4-18 展示了集群中各个 Scrapy 爬虫与主节点中爬虫抓取任务交互的总体图，在主节点中存在两类爬虫抓取任务队列，一类为爬虫自身的抓取任务队列，该队列内部详细设计在图 4-8 中已经阐述了，当 Scrapy 爬虫启动后，周期性的调用调度器的 next_request 方法遍历自身的域名队列，如果集群中的爬虫访问该域名的速度没有超过其对应限速器设置的访问频率，则按照任务优先级的顺序依次获取爬虫抓取任务并交给下载器子模块，否则遍历其下一个域名队列。另一类为所有主机中爬虫共享的总爬虫任务队列，该队列基于 Redis 的有序集合实现，key 为 seeds，当数据采集子模块中提取出新的 URL 后，首先调用 RFPDupeFilter 类（该类实现了 4.1.5 小节的过滤器）的 request_seen 方法判断当前 URL 是否抓取过，如果没有抓取过且没有在用户设定的黑名单（Blacklist）中，则将该 URL 封装成 4.1.2.1 小节中爬虫抓取请求规定的格式，并调用调度器的 enqueue_request 方法将该爬虫抓取请求提交给总爬虫任务队列。

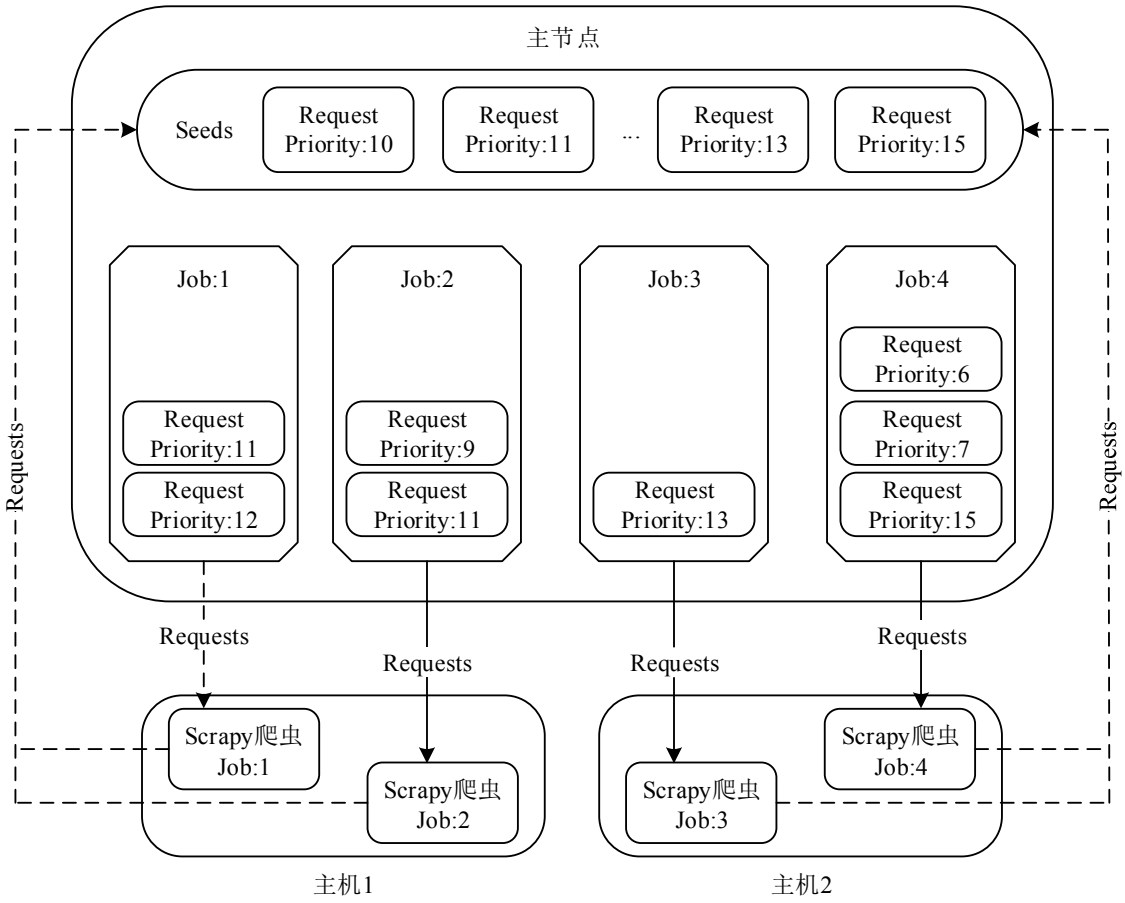

图 4-18 爬虫抓取任务获取与提交

4.3.1.2 中间件子模块

中间件子模块位于中心引擎和下载器子模块之间，该模块负责对中心引擎传递过来的 Request 请求以及下载器子模块返回的 Response 响应进行加工处理。

（一） Request 请求加工

目前大多数网站为了防止自身网站被网络爬虫频繁访问，使用反爬虫技术来限制非正常用户的访问。为了保证爬虫抓取的效率，本节通过对 Request 请求深加工来防止爬虫速度受到站点反爬虫策略的影响。

添加预处理请求头：

这种方式是最常见的也是最基本的对付反爬虫的手段，user-agent 是 HTTP 协议中请求头字段，其目的是告知服务器发送请求的终端的基本信息，服务器通常通过这个字段来判断访问网站的对象是爬虫还是用户，对于用户来说，每次访问网站都会带有其固定的 user-agent，通常以“Mozilla/5.0”开头，而爬虫程序的 user-agent 一般为空，故系统事先收集好各种浏览器下的 user-agent，并保存到本地文件中，当有新的 Request 请求时，随机从文件中选取一个 user-agent 值填充到该

请求的 user-agent 字段中，这样远程服务器就无法使用该反爬虫策略限制爬虫了。

添加动态 IP：

相比于普通用户，网络爬虫通常会在一段时间内频繁的访问目标网页，服务器通过监控单个 IP 某段时间内的访问量来判断访问者是机器还是用户，如果访问量超过了某个阈值，则禁止其访问站点。为了解决并发限制的反爬，中间件子模块内部维护了一个 IP 代理池，该代理池本质上是一个维护了多个代理 IP 的队列。当中心引擎向下载器子模块发送 Request 请求时，随机从 IP 代理池中取出一个有效的 IP 来替换原有的 IP，这样能保证了爬虫不会使用单一的 IP 去频繁的访问目标站点。

为了构建一个完整高效的 ip 代理池，系统实现了 ProxySpider，Validator 和 ProxyApi 三个类，ProxySpider 类负责代理 ip 的采集及存储，Validator 类负责代理 ip 有效性验证，ProxyApi 类负责提供获取代理 ip 的接口。首先调用 ProxySpider 类的 start_requests 方法，对 start_urls 列表中 ip 代理网站链接进行读取并发起请求，当下载器完成对页面的下载后，调用 parse 方法根据抽取规则对页面中的 ip 及相关信息进行抓取，抓取的字段如表 4-7 所示。

表 4-7 待抓取的字段

序号	字段
1	IP 地址
2	端口
3	服务器地址
4	是否高匿
5	类型
6	速度
7	存活时间

待站点中的代理 ip 抓取完毕后，调用 Validator 类的 checkProxy 方法对采集的 ip 进行有效检测，本文通过使用代理 ip 对百度、淘宝等网站发起请求，如果响应时间超过 4 秒则判断该代理 ip 无效，筛选出所有有效的 ip 后，调用 ProxySpider 类的 save 方法将有效的 ip 存入 mongodb 数据库中。鉴于网络上的 ip 代理质量层次不齐，系统还通过调用 Validator 类的 process_start 方法开启一个后台线程周期性的检测数据库中存放的代理 ip 的有效性，当 ip 失效后则从数据库中删除。具体的原理图如 4-19 所示。

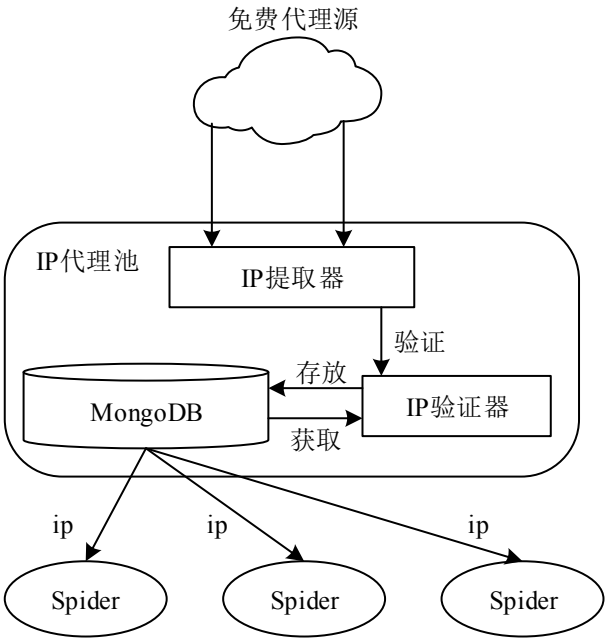

图 4-19 代理池设计原理图

（二） Response 响应处理

从下载器子模块传递过来的 Response 响应中存在两个关键的值，一个为目标网页的源代码，另一个为 HTTP 响应状态码。目标网页的源代码后续将交给数据采集子模块处理，HTTP 响应状态码的统计工作将在中间件子模块中完成。表 4-8 展示了系统默认将统计的 HTTP 响应状态码。

表 4-8 HTTP 响应状态码统计

HTTP 状态码	含义
200	成功抓取
404	没有找到页面
301	资源已被移走
303	重定向
403	禁止访问
500	服务器错误

HTTP 响应状态码的统计是基于 4.1.1 小节的状态计数器实现的，每一种状态码在 Redis 数据库中都对应了一个固定的 key，当收到新的 Response 响应后，根据其状态响应码在 Redis 中找到对应的 key，并利用状态计数器为其计数一次。这样就能很清楚的观察最近一段时间内爬虫与远程服务器交互的情况了。

4.3.1.3 数据采集子模块

随着数据价值不断的提升，用户需求也越来越多样化，爬虫往往需要从非结构化网页中提取出结构化的信息，例如不同网站的商品信息、股票信息、房产信息等等，但是由于各个网站页面结构各不相同，单独为每个网站开发一个爬虫，效率极其低下。数据采集模块通过重写 Scrapy 自身的 Spider 组件来实现一个带爬行规则的网络爬虫，用户只需填写待抓取网页的抽取规则就能将该网站的信息抓取下来。

本系统根据业务逻辑将爬虫分为两类，每一类对应一个 Spider。一类为精准爬虫，该类爬虫通常针对特定网页的特定业务数据，其抓取的网页通常表现为多个元素并列出现，元素与元素之间仅仅基本信息不同，如淘宝的商品列、链家的房产列等等。另一类为通用网页爬虫，该类爬虫常常需要从一些种子 URL 深度或广度遍历扩充至整个网络，常见目标网址如网易新闻、凤凰网以及腾讯新闻等等。

(一) 精准爬虫

考虑到现实情况下，精准爬虫所抓取的网页通常维持在几百个左右，如果用任务调度器单独为这几百个 URL 调用算法选择分配的爬虫，相较于爬虫其本身的抓取速度，反而降低了系统整体的性能。为此系统为该类爬虫共同维护了一个爬虫任务队列，不同机器的爬虫节点直接从该队列中取出任务执行即可。图 4-20 展示了精准类爬虫获取任务的过程。

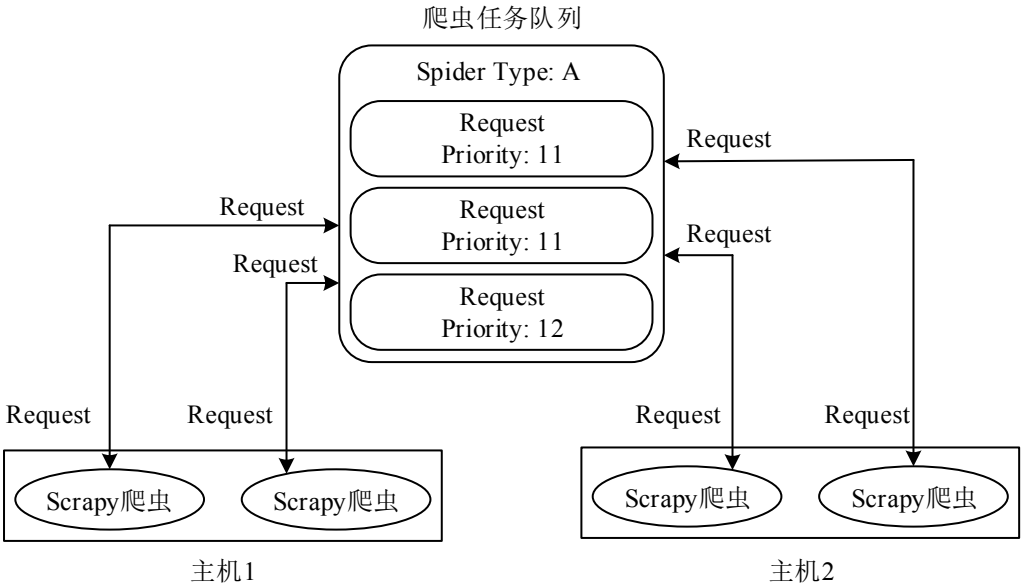

图 4-20 精准爬虫类获取任务的过程

当爬虫从任务队列中获取到对应的爬虫任务后，先将其交给下载器完成对应页

面（HTML）的下载，页面下载完毕后根据对应的抽取规则提取页面中所需要的数据。常见的 HTML 解析库有 BetifulSoup 和 lxml 两种[40]，本文采用 lxml 库中的 xpath 表达式进行解析，xpath 是 XML 路径语言，可以方便地确定 XML 文档中数据的位置。

系统将用户提供的抽取规则分为两个部分，第一部分为首页的抽取规则（first_page_xpath）：包括列表项在网页中具体位置，列表项中元素其它相关信息的位置，列表项中元素详情页面链接的位置；第二部分为详情页的抽取规则（second_page_xpath）：详情页中具体采集信息的位置。其具体抽取规则字段如表 4-9 所示。

表 4-9 抽取规则

首页抽取规则（first_page_xpath）	
关键字	描述
list_from	列表项在网页中的位置
continue	是否继续详情页数据抓取
second_page_url	详情页链接的位置
用户自定义 1	元素其它相关信息的位置
…	…
用户自定义 n	元素其它相关信息的位置
详情页抽取规则（second_page_xpath）	
关键字	描述
用户自定义 1	用户指定的详细页采集信息位置
…	…
用户自定义 n	用户指定的详细页采集信息位置

图 4-21 展示了一个精准爬虫负责抓取的网站，该网站负责定期发布与四川省人民政府相关的信息，通知以列表的形式呈现。

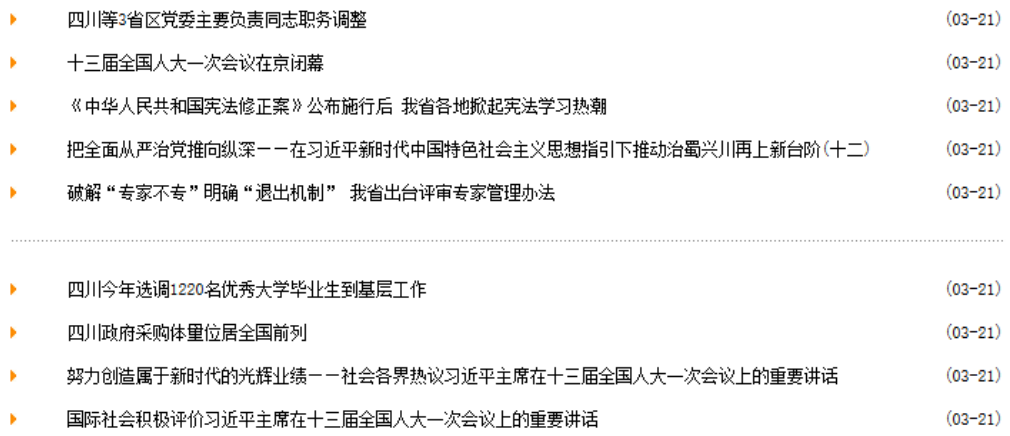

图 4-21 四川省人民政府通知

其对应的网页源代码如图 4-22 所示：

图 4-22 网页源代码

故首页抽取规则（first_page_xpath）中每则通知在网页中具体位置（list_from）的 xpath 值为“//table[@id=’ zp_longListTable’]//table/tbody/tr”，每则通知详情页面链接位置（second_page_url）的 xpath 值为“td/span/a/@href”，其它与通知相关的信息由用户自定义，如需获取当前通知发布日期，则关键字设为 date，其对应位置的 xpath 值为“td[2]/em/text()”，详情页抽取规则类似。

parse_inform_index 方法负责根据用户提供的抽取规则（first_page_xpath）提取首页列表项的信息。该方法首先通过用户提供的 list_from 抽取规则找到网页中列表项具体的位置，然后依次遍历每一个列表项，遍历时根据用户自定义的抽取项提取出列表项中其它相关信息，同时如果该列表项元素中存在详情页地址，则再根据 second_page_url 规则提取出详情页的链接地址，并封装成 Request 请求交给

该类爬虫共享的爬虫队列。parse_inform_index 方法核心代码如下所示：

```
first_page_xpath = rep.meta[‘first_page_xpath’] # 获取首页抽取规则
second_page_xpath = rep.meta[‘second_page_xpath’] # 获取详情页规则
selector = Selector(rep) # 将 rep 中的 HTML 代码转化为 DOM 树
for sel in selector.xpath(first_page_xpath[‘list_from’]):
    item = dict() # 构造存储数据的容器
    for field, val in first_page_xpath.items():
        if field == ‘second_page_url’: # 提取详细页面链接
            second_page_url = item[field] = sel.xpath(val).extract()[0]
            continue;
        item[field] = sel.xpath(val).extract()[0] # 提取元素相关信息
        # 封装 Request 请求并交给该类爬虫共享的爬虫队列
        if second_page_url and continue:
            yield SplashRequest(url=second_page_url, meta={‘item’: item,
                                ‘second_page_css’: second_page_xpath},
                                callback=’ parse_inform_detail’)
```

parse_inform_detail 方法主要负责根据用户提供的详情页抽取规则（second_page_xpath）抽取出元素详细页面的信息。下载器下载完详情页面的源代码后，回调该方法解析页面内容，该方法核心代码如下：

```
# 获取详情页抽取规则
second_page_xpath = rep.meta['second_page_xpath']
item = rep.meta['item']   # 加载首页采集的数据
for field, val in second_page_xpath.items(): # 提取详情页字段数据
    item[field] = rep.xpath(val).extract()[0]
item[‘body’] = rep.body # 获取详情页源代码
return item # 将收集到的数据一并交给数据管道模块
```

(二) 通用网页爬虫

通用网页爬虫更类似于一个面向全网的小型爬虫，该类爬虫最常见的应用就是抓取门户网站。通过对比国内几家大型门户网站，如凤凰网、新浪网，其网页无论在页面布局或是链接命名规则等都比较类似，所以该类爬虫也可以通过制定不同的抽取规则抓取不同的网站。

大多数门户网站的链接大致可以分为以下三种类型：第一类为导航型链接、第二类为内容型链接、最后一类为无关型链接。导航型链接是指网站中各种主题的链接，这些链接中包含了大量属于这个主题的新闻，如凤凰网门户网站中军事、

历史、娱乐、音乐等等主题；内容型链接是指向最终我们需要采集内容页面，如新闻页、评论页等；而无关型链接则指向不属于该网址域名下的链接或不需要抓取的链接，如广告、视频、指向外部站点的链接等等，前两种类型的链接是我们需要的链接，只要能收集完这些链接就能采集完所有的信息。

针对通用型的网页，数据采集子模块主要通过 UniversalSpider 类来实现其数据采集功能，类图如 4-23 所示。

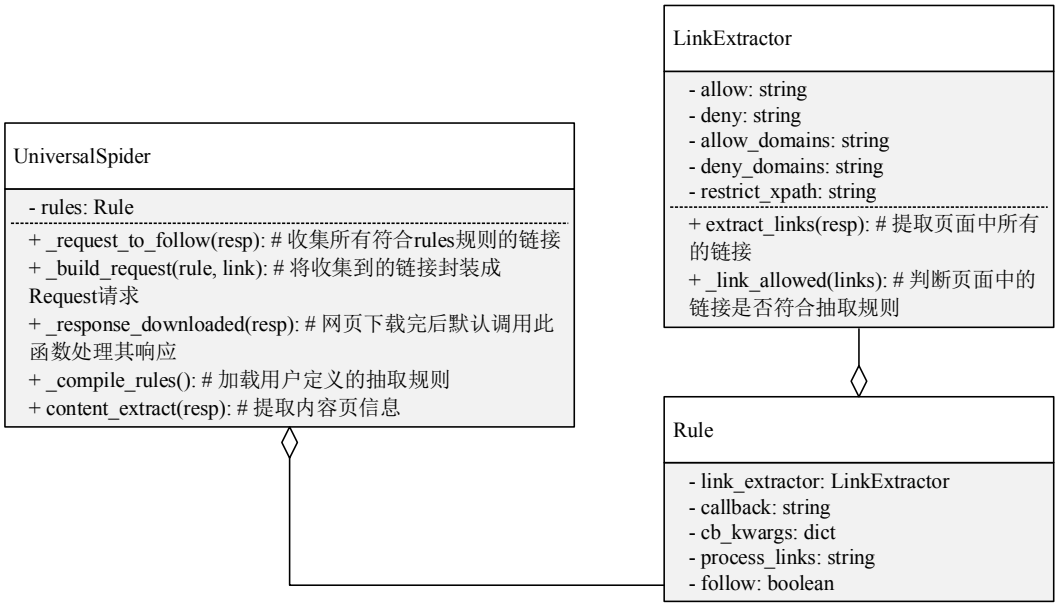

图 4-23 通用网页爬虫所关联部分类图

如图 4-23 所示，UniversalSpider 类中唯一指定了一个核心成员变量 rules，该变量是 Rule 类实例的集合，每一个 Rule 类实例都对抓取网站的动作定义了特定的表现，即用于告知爬虫哪些链接需要跟踪，哪些链接不需要跟踪。Rule 类包含 5 个成员变量，如表 4-10 所示。

表 4-10 Rule 类的成员变量及含义

成员变量	描述
link_extractor	提取出符合抽取规则的链接
callback	链接下载完后的回调函数名
cb_kwargs	附带传递给 callback 函数的参数
follow	提取出的链接是否继续跟进
process_links	进一步过滤提取出的链接的函数名

由图 4-23 可知，link_extractor 的值为 LinkExtractor 类的实例，该类通过它的 5 个

成员变量来制定相应的规则，从网页中抽取出符合规则的链接。具体描述如表 4-11 所示。

表 4-11 LinkExtractor 类的成员变量及作用

成员变量	描述
allow	符合正则表达式参数的链接会被提取
deny	符合正则表达式参数的链接禁止提取
allow_domains	包含此域名的链接可以提取
deny_domains	包含此域名的链接禁止提取
restrict_xpaths	筛选 xpath 中对应位置的链接

下面以凤凰网为例来说明爬虫如何抓取网站中所有主题下的新闻，首先确定网页中的导航型链接和内容型链接，凤凰网中导航型链接如图 4-24 所示。

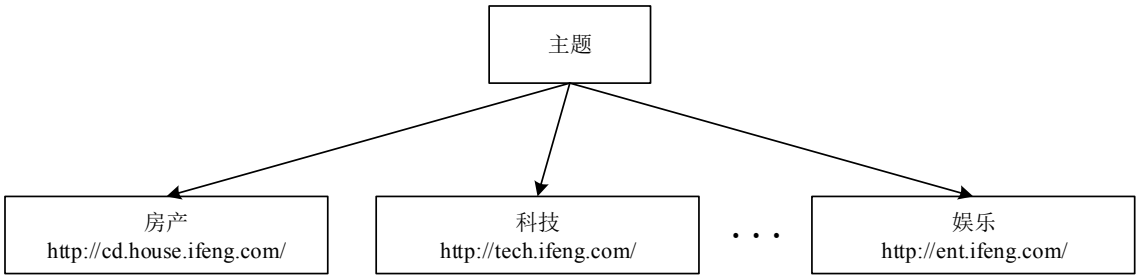

图 4-24 类别型链接

图 4-24 列举了 3 个导航型链接的地址，其余主题的链接与之相似，从中我们不难发现导航型链接地址除了头部不同，其余部分基本相同，所以我们很容易地制定出匹配此类 URL 链接的正则表达式“^http://([0-9a-z]\.)*ifeng\.com/$”，该值最终将传递给 LinkExtractor 类实例的 allow 参数，筛选出网页中的导航型链接；同理内容型链接如图 4-25 所示。

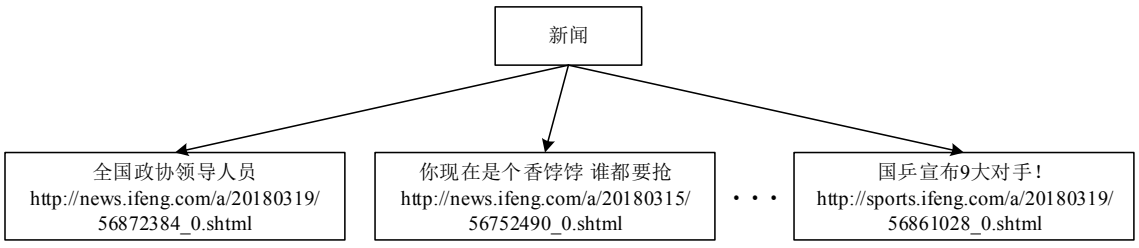

图 4-25 内容型链接

故匹配此类 URL 地址的正则表达式为：

“^http://([0-9a-z]\.)*ifeng.com/a/d{4}/d{8}_0\.shtml”，该值也会传递给另外一个LinkExtractor 类实例的 allow 参数，用于筛选出网页中的内容型链接。

为了提高抓取效率，过滤无关型链接也是非常有必要的。例如凤凰网下视频类链接“http://v.ifeng.com/”，此类网页中的链接几乎都指向一个个视频地址，遍历该类网页下的链接无疑大大增添了抓取任务量，故需增添正则表达式“http://v.ifeng.com/*”并赋值给 LinkExtractor 类实例的 deny 参数，过滤掉此类链接。

至此通用网页爬虫的抓取策略及抽取规则描述完毕，下面结合代码具体说明该类爬虫如何进行抓取。首先通过调度器从爬虫任务队列中提取出初始采集链接，并调用_compile_rules 方法将该链接的抽取规则依次加载到对应的 rules 变量中，规则加载完成后，下载器将该链接向远程服务器发起请求。当对应的页面后下载完成后，默认调用_response_downloaded 方法解析其页面信息，核心代码如下所示：

```
1.  if callback: #  如果页面有相应的回掉函数
2.          cb_res = callback(rep, **cb_kwargs) or ()
3.          cb_res = self.process_results(rep, cb_res)
4.          for request_or_item in iterate_spider_output(cb_res)
5.                  yield request_or_item
6.  if follow: #  继续跟进当前页面中的所有链接
7.          for request_or_item in self._requests_to_follow(response):
8.                  yield request_or_item
```

该方法首先判断页面是否有对应的回调函数，如果有则调用其回调函数解析页面内容；然后则根据抽取规则中的 follow 参数判断当前链接是否需要跟进，如需要则通过调用_requests_to_follow 方法对页面中所有符合规则的链接进行提取。该方法通过调用 LinkExtractor 类的 extract_links 方法依次遍历网页中所有的链接，并通过_link_allowed 方法筛选出符合表 4-10 规则的链接，_link_allowed 核心代码如下：

```
1.  if not _is_valid_url(link.url): #  链接地址是否合法
2.          return False
3.  #  是否符合 allow 规则
4.  if self.allow_res and not _matches(link.url, self.allow_res):
5.          return False
6.  #  是否符合 deny 规则
7.  if self.deny_res and _matches(link.url, self.deny_res):
8.          return False
```

```
9.  parsed_url = urlparse(link.url) #  抽取链接链接地址域名
10. #  是否符合 allow_domains 规则
11. if self.allow_domains and not url_is_from_any_domain(parsed_url,
    self.allow_domains):
12.         return False
13. #  是否符合 deny_domain 规则
14.  if self.deny_domains and url_is_from_any_domain(parsed_url, self.deny_domains):
15.          return False
16. return True #  满足所有的抽取规则
```

当筛选完页面中所有的链接后，为符合抽取规则的链接注册相应的回调函数并存入爬虫任务队列；如此反复，直到爬虫任务队列中没有任务。具体流程图如 4-26 所示。

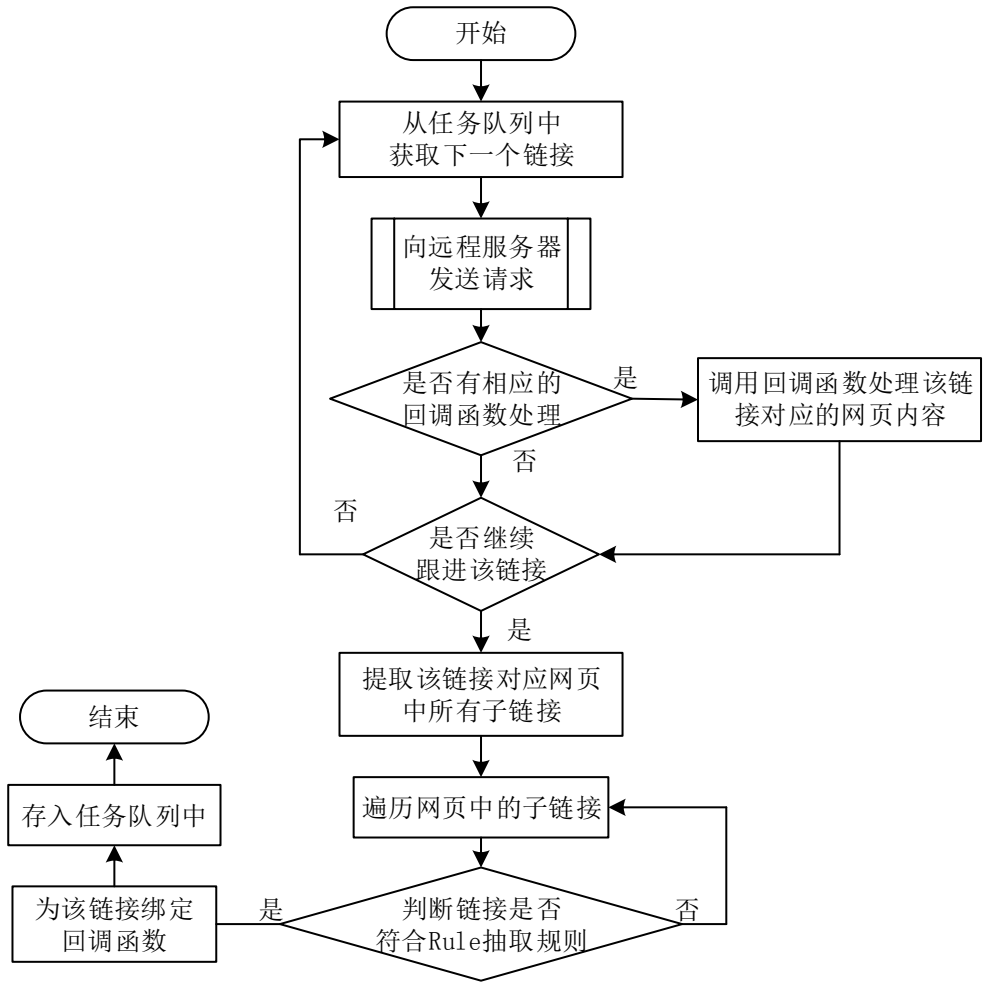

图 4-26 通用型网页提取流程图

4.3.1.4 数据管道子模块

数据管道模块主要负责处理从数据采集模块中提取出的数据（item），典型的处理包括对 item 进行清理、验证以及持久化操作。在本系统中，数据管道模块主要完成以下两个方面的任务。

（一） 编码转换及数据清洗

编码转换：本文的抓取系统是一个面向不同站点的网络爬虫系统，而不同的网站往往对应着不同的编码，为了保证最终数据显示正确，爬虫在采集网页数据时通过提取 HTML 源代码中 meta 标签内 content 元素的值来获取页面编码，并根据该值将抓取的数据统一解码为 unicode 格式。

数据清洗：由用户编写的爬虫抽取规则就不可避免地存在出现人为差错的可能性，体现在最终数据上的话，就是最终元素某些字段的值为空，数据清洗管道需要过滤掉这些不完整的数据，并记录在日志文件中，方便用户完善抽取规则来抓取完整的数据，如果采集的数据本身就是完整的，则直接存放到 MongoDB 集群中。

（二） 网页正文提取

对于目标类网页为新闻网页时，有价值的部分往往存在于网页正文当中，因此需要将网页中与正文不相干的部分剔除，可以说正文提取的准确度会直接影响后续信息分析结果的好坏。目前常见的正文提取方法有以下三种：

1. 基于 Dom 的网页正文提取算法[41]

该类算法首先将网页的 HTML 源代码规整化处理后转化为一棵 Dom 树，然后遍历 DOM 树中的每个标签元素，并识别标签中的非正文信息，如导航条，广告信息以及版权信息等等，最后移除掉页面中与正文无关的信息后，剩余的内容就是正文信息。缺点：Dom 树的建立和遍历时间复杂度高，且 HTML 标签及文本组织随意，无法制定统一的标准来区分正文信息标签和非正文信息标签。

2. 基于机器学习的网页正文提取算法[42]

该类算法首先通过人工标注大量网页，然后根据机器学习算法来训练这些数据，并从中提炼出正文信息特点，最后利用训练后的模型对后续采集的页面正文提取。缺点：简单问题复杂化，同时人工标注正文信息大大增添了开发者的工作量。

3. 基于行块分布函数的网页正文提取算法[43]

该类算法不需要将网页 HTML 源代码转化为 Dom 树，而是将网页正文信息提取转化为页面的行块分布函数构建，该方法完全脱离了 HTML 标签，并且始终能在线性时间 O（n）内抽取出网页正文信息。本文也是基于此方法完成网页正文的提取，下面以一个具体的例子来介绍该算法如何提取出网页中的正文信息。

以一篇新闻为例，算法首先过滤掉网页中所有的 HTML 标签，但保留嵌套在

标签中的文本信息，然后将处理后的 HTML 源代码按换行符进行切分，并将每一行的文本信息存放到字符串数组中。接着以字符串数组中的行号为轴，取其周围 k 行（k<=5）合并成一个行块，行块长度为去掉空白字符后的字符总数，最后以行块为 x 坐标，行块的长度为 y 坐标建立对应的行块分布函数，由于正文部分往往是网页中文字最多其最密集的区域，故在分布图上包含最值且连续的区域一定为正文部分，故只要找到该区域就能提取出网页正文部分。图 4-27 展示了网页去除标签后，每一行对应的字符数。

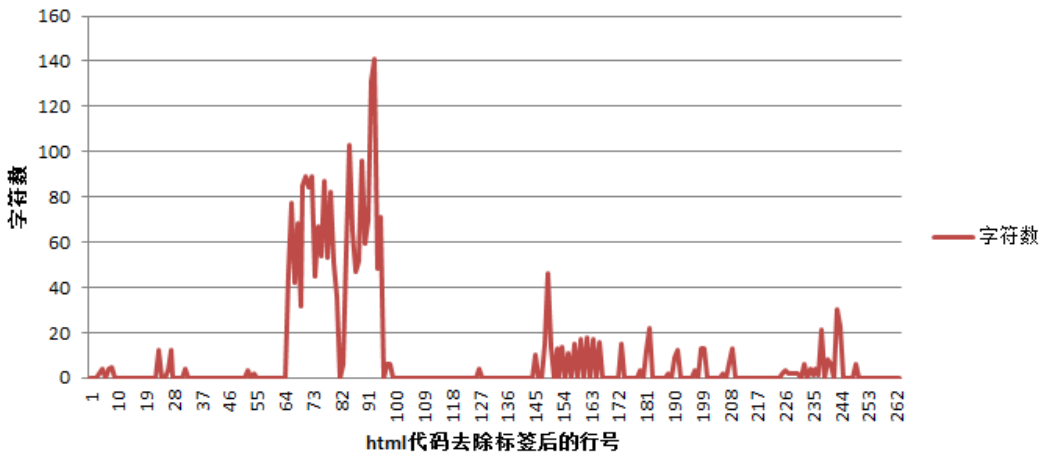

图 4-27 去除标签后文本信息的分布图

由图 4-27 可知，65 行-100 行之间的位置字符数是比较密集的且最多的，故该段就是网页中的正文信息。

4.3.2 爬虫管理

由于主节点和从节点（群）往往不在同一台主机中，想要启动 Scrapy 爬虫程序完成用户提交的抓取任务，必须到各个从节点中手动键入 Scrapy 命令行来启动爬虫，这无疑降低了用户的可操作性；同时由于主从节点无法直接通信，导致主节点无法时刻监控各个从节点群中爬虫运行状态信息。鉴于以上原因，本文使用 Twisted Application 框架[44]实现一个异步任务响应的爬虫管理器，用户能通过爬虫管理子页面远程发送请求控制各个从节点中的 Scrapy 爬虫程序。由于篇幅有限，Web 端的构建在这里就不在赘述，其原理和目前主流的框架如 Flask、Django 类似，本节主要描述在接受到请求后，爬虫管理器模块如何管理本机上的爬虫。

4.3.2.1 Deffered 组件

为了清楚了解模块运行原理，本节实现了一个简单的基于 Deferred 延迟对象[45]的程序来介绍模块中的所用到的核心组件 Deferred。

```
1. def got_name(name):
2.     print ‘your name is ’, name
3. def name_failed(error):
4.     print ‘no name for you’
5. defer = Deferred() # 创建一个 Deferred 延迟对象
6. defer.addCallbacks(got_name, name_failed) # 为延迟对象添加回调函数
7. defer.callback(‘uestc’) # 激活 defer
```

程序通过 Deferred 类创建一个 Deferred 延迟对象 defer，并调用 addCallbacks 方法为其注册回调函数 got_name 以及 name_fail 方法，当调用 defer 对象的 callback 方法时激活 defer 延迟对象内部注册的回调函数：如果 callback 传入的参数类型不是异常的话，调用 got_name 方法，否则调用 name_fail 方法。其中每个 Deferred 延迟对象可以注册多个回调函数，激活 defer 意味着以我们添加的顺序调用已注册的回调函数，具体的回调链激活如图 4-28 所示。

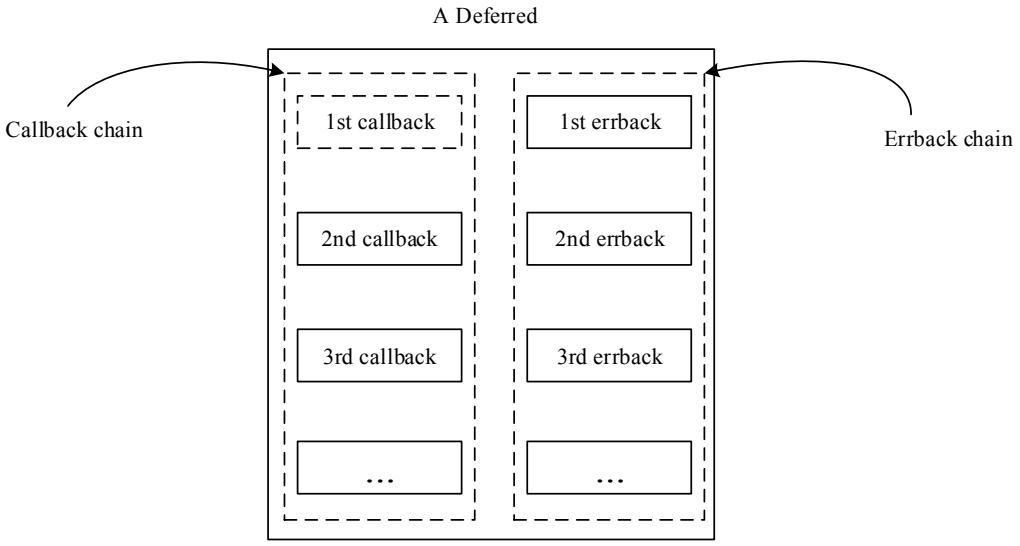

图 4-28 deferred 回调链原理图

Deferred 延迟对象的出现，解决了同步编程中需要花费大量时间等待阻塞事件的问题，如磁盘 IO、网络响应等。事先为 Deferred 延迟对象注册回调函数，并直到耗时的操作处理完后才调用先前注册的回调函数去处理相应的逻辑，这种策略大大提高了 CPU 的利用率及单位时间的吞吐量。

4.3.2.2 异步任务响应设计

为了后续管理用户提交的爬虫启动任务，系统实现 SqliteSpiderQueue 类来存放提交的爬虫启动任务，该类基于轻量级数据库 SQLite 来实现。爬虫启动任务信息主要包含 4 个字段：project: 爬虫项目名，spider: 爬虫项目下具体类型的爬虫，priority: 任务优先级以及 job_id: 爬虫启动后的任务号（唯一标识某个爬虫进程）。

在记录了爬虫任务启动信息后，需要依次取出队列中存放的信息并创建相应的爬虫进程，为了构建异步任务响应式的爬虫管理器，系统在 SqliteSpiderQueue 类创建的任务队列基础上抽象出一个由 QueuePoller 类构造的 Deferred 队列，队列默认长度为 1，该队列提供了 put 和 get 两种方法，put 方法负责将爬虫启动任务存放到该队列中，get 方法根据队列的情况返回不同的 Deferred 延迟对象：当队列为空时，调用 get 方法返回一个未激活的 Deferred 对象，当有新的任务入队时，立即以该值激活 Deferred 对象注册的回调函数；当队列不为空时，调用 get 方法返回一个激活的 Deferred 对象，激活的值为 Deferred 队列的首元素。

activate_defer 方法是 QueuePoller 类的核心方法，该方法周期性地从爬虫启动任务队列中取出待启动的爬虫信息加入到 Deferred 队列中，具体核心代码如下所示：

```
1.  @inlineCallbacks
2.  def activate_defer(self):
3.    if self.dq.pending: # Deferred 队列有值时，直接返回
4.      return
5.    for p, q in iteritems(self.queues): # 遍历爬虫启动任务队列
6.      c = yield maybeDeferred(q.count)
7.      if c: # 判断是否有需要启动的爬虫任务
8.        msg = yield maybeDeferred(q.pop) # 弹出爬虫启动任务
9.        returnValue(self.dq.put(msg)) # 加入 Deferred 队列
```

由上述代码第九行可知，当有需要启动的爬虫任务时，系统并没有直接创建相应的爬虫进程来响应该请求，而是将待启动的爬虫信息加入到 QueuePoller 类构造的 Deferred 队列中，避免由等待创建进程所消耗的时间影响后续爬虫任务的启动。

收集完要启动的爬虫信息后，系统通过创建 Start 类来实现启动基于特定协议的爬虫进程。Start 服务启动时调用该类的 startService 方法，该方法循环从 QueuePoller 构造的队列中获取 max_proc（该值限定了当前调度器最多能同时管理多少个爬虫进程）个未激活的 Deferred 对象，并为每一个 Deferred 对象注册

_create_process 方法，一旦 Deferred 队列中加入新的爬虫启动任务，则激活 Deferred 对象中注册的回调函数（也就是调用_create_process 方法），这样就能避免由于等待进程创建所导致的爬虫启动任务响应不及时问题。_create_process 方法负责实际最终爬虫进程的创建，该方法通过调用 Twisted 框架内部提供的 spawnProcess 接口创建相应的进程，其中该接口的第一个参数需为 processProtocol 对象，该对象负责监听所有与进程相关的事件（如进程结束，进程创建成功等等），本系统通过 ScrapyProcessProtocol 类实现，为了避免由 startService 方法初始化的 mac_proc 个 Deferred 对象使用完后不能再处理新的爬虫启动任务，系统在 ScrapyProcessProtocol 类中创建一个未被激活的 Deferred 成员变量 deferred，并为其注册_process_finished 方法，当 ScrapyProcessProtocol 对象监听到某个爬虫进程结束后立即调用其内部的 processEnded 方法激活 deferred，触发_process_finished 方法从 Deferred 队列中再获取一个新的未激活的 Deferred 对象，并为其注册_create_process 方法，通过该机制保证系统中始终存在未激活的 Deferred 对象来处理后续的爬虫启动任务，_create_process 方法核心代码如下所示：

```
1. def _create_process(self, msg, id):
2.     project = msg['_project'] # 构建启动爬虫的 Scrapy 命令
3.     args = [sys.executable, '-m', self.runner, 'crawl'] + get_crawl_args(msg)
4.     # 创建 ScrapyProcessProtocol 类并注册回调函数
5.     pro = ScrapyProcessProtocol(id, project, msg['_spider'], msg['_job'], env)
6.     pro.deferred.addBoth(self._process_finished, id)
7.     reactor.spawnProcess(pro, sys.executable, args=args, env=env) # 创建进程
8.     self.processes[id] = pro
```

至此异步任务响应架构实现完毕，图 4-29 展现了整个任务响应架构的原理图。

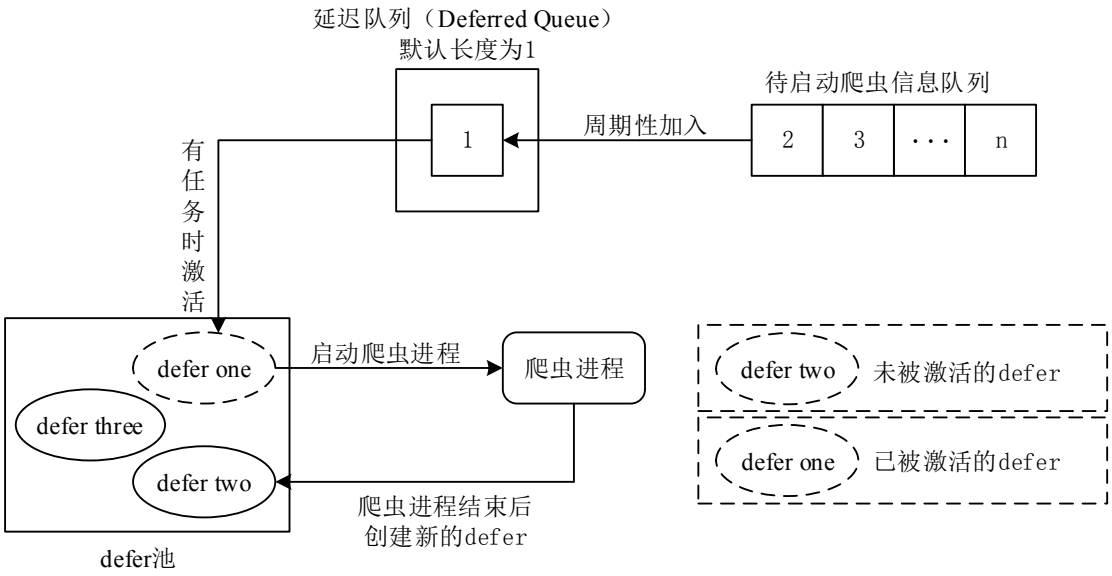

图 4-29 任务响应架构原理图

4.3.3.3 任务取消及状态查询

每次调用_create_process 方法创建爬虫进程时，都会创建 ScrapyProcessProtocol 对象来监听与该进程相关的事件，同时该类也记录爬虫进程的一些基本信息，如表 4-12 所示。

表 4-12 爬虫进程相关基本信息

字段	含义
project	启动的爬虫项目名
spider	具体类型的爬虫
job	用户指定的爬虫任务号
start_time	爬虫进程启动时间
end_time	爬虫进程结束时间
log	爬虫输出日志路径

当爬虫进程创建完毕后，将 ScrapyProcessProtocol 对象加入到正在运行的爬虫队列中，该队列记录了当前正在运行的爬虫进程。同时当爬虫进程结束后，回调_process_finished 方法将该进程对应的 ScrapyProcessProtocol 对象加入到已完成的爬虫队里中。

一旦爬虫管理器收到来自客户端取消爬虫进程的请求后，先从参数中解析出要取消的爬虫项目名及爬虫任务号，如果该爬虫启动任务存在于待启动爬虫信息队

列中，意味则还未创建对应的爬虫进程来启动爬虫，则直接从数据库中删除该爬虫启动任务；如果在待启动爬虫信息队列中不能查询到该爬虫启动任务，意味着爬虫管理器已经创建相应的爬虫进程来启动爬虫，则遍历正在运行的爬虫队列找到与爬虫启动任务相同任务号的爬虫进程，并调用 Twisted 内部的 signalProcess 接口向该进程发送结束的信号停止该进程，任务取消流程图如图 4-30 所示。

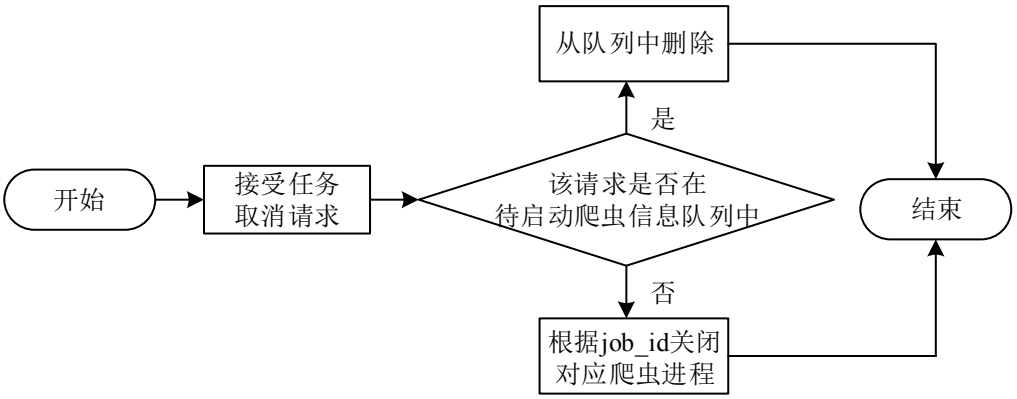

图 4-30 任务取消流程图

爬虫管理器内部维护了爬虫启动任务的三种状态，分别为待启动，运行中以及运行完成。SqliteSpiderQueue 类维护了待启动爬虫信息队列，Launcher 类中 process 变量存放了正在运行的爬虫进程，finished 存放了已经结束了的爬虫进程。当爬虫管理器收到来自客户端爬虫状态查询的请求后，分别遍历上述队列并将爬虫状态返回给用户。

4.4 本章小结

本章按照第三章设计的总体框架，依次对分布式网络爬虫系统中各模块的设计和实现进行了详细描述。其中主节点和从节点群是整个系统的核心部分，主节点通过任务调度器实现了对爬虫抓取任务的均衡分发，并结合限速器和过滤器实现了爬虫限速和 URL 去重功能。从节点通过重新定制开发 Scrapy 框架的调度器子模块、中间件子模块、数据采集子模块和数据管道子模块完成各类网页数据采集工作，最后结合爬虫管理器实现对各个节点中的爬虫监控管理。

第五章 系统测试与展示

5.1 系统运行环境

本实验测试环境由六台服务器组成，三台作为主节点，另外三台作为从节点。为保证系统的容错率更高，整个分布式爬虫系统中的服务均以集群的方式搭建。

(一) 主节点运行环境

硬件环境：

主节点中的任意一台服务器均与其余两台服务器配置一致，表 5-1 展示了其中一台服务器硬件配置信息。

表 5-1 主节点服务器硬件配置信息表

硬件	描述
CPU	英特尔 第四代酷睿 i5-4210H
内存	16 GB
硬盘	250 GB
网卡	百兆网卡
带宽	10MB/S

软件环境：

主节点中服务器统一以 Centos 7 作为其操作系统，其中 Kafka，Zookeeper，Redis 均以集群的方式部署在三台服务器上。

(二) 从节点运行环境

硬件环境：

从节点中任意一台服务器也与其余两台服务器配置一致，表 5-2 展示了其中一台服务器硬件配置信息。

表 5-2 从节点服务器硬件配置信息表

硬件	描述
CPU	英特尔 第四代酷睿 i5-4210H
内存	4 GB
硬盘	500 GB
网卡	百兆网卡
带宽	10MB/S

软件环境：

从节点中服务器统一以 Window 10 作为其操作系统，每台服务器中都预先启动一个爬虫管理器后台进程，MongoDB 以集群的方式部署在三台服务器上。涉及到的核心依赖包括 Python 2.7，Scrapy 1.0，Twisted 15.0。

5.2 模块性能测试

5.2.1 任务调度器模块测试

任务调度器模块主要负责主节点中抓取任务的分发工作，为了更好的观察任务调度器分发任务的过程，实验模拟了 3 个爬虫节点 slave1，slave2，slave3，slave1 每分钟完成 360 个采集任务，slave2 每分钟完成 420 个采集任务，slave3 每分钟完成 480 个采集任务，然后以 10 万个 URL 链接作为基准点，每隔十分钟统计一次任务调度器给每个爬虫节点分发的任务量，并且在 30 分钟、50 分钟时调整 slave1 的任务采集速度，观察任务调度器分发任务的变化。实验结果如图 5-1 所示。

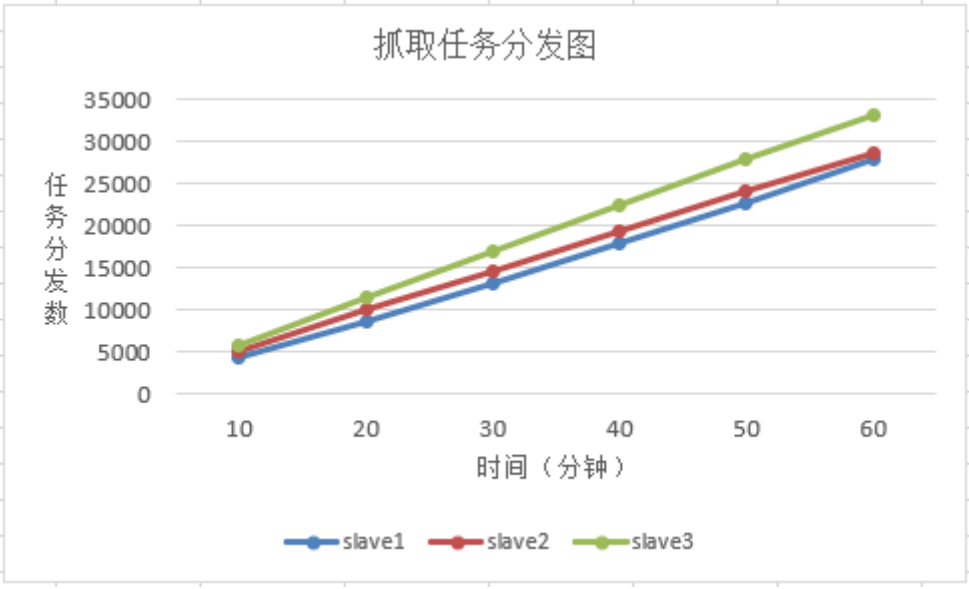

图 5-1 抓取任务分发图

由图 5-1 所示，随着时间的推进，采集速度较快的爬虫节点相比于采集速度较慢的爬虫节点分配到抓取任务逐渐增多，同时在 30 分钟时，实验将 slave1 节点的采集速度调整为 420 个/min 后，任务调度器给 slave1 和 slave2 节点分配的任务数就基本相同了（斜率基本一致），在 50 分钟时，将 slave1 节点的采集速度调整为 480 个/min 后，其分配的任务数也和 slave3 一致了。故任务调度器能动态的根据爬虫节点完成任务的速度分发其相应的任务量。

5.2.2 过滤器模块测试

过滤器模块主要负责过滤爬虫抓取过程中出现重复的 URL。本节将从去重误判率和耗费时间两个方面来评判去重策略的性能，假设给定一个数量为 n 的 URL 数据集，该数据集中重复的 URL 个数为 x，通过去重策略判断出重复的 URL 个数为 y，则该策略的误判率 w 计算公式为 5-1 所示：

$$w = \frac{y - x}{n} \tag{5-1}$$

为验证布隆过滤器在海量数据下的优越性，实验使用 Redis 数据库、磁盘等去重策略与本文提出的基于布隆算法的 URL 去重策略对比，初始的 URL 数据集大小为 100 万，表 5-3 展示了不同去重策略下的误判率。

表 5-3 不同去重策略下的误判率

去重策略	重复 URL 个数 x	去重策略下的重复 URL 个数 y	误判率
磁盘	50000	50000	0 %
内存	50000	50000	0 %
布隆过滤器	50000	61804	1.18%

当初始 URL 数据集大小为 100 万、500 万以及 1000 万时，每插入一条 URL，判断其是否重复所耗费的时间如图 5-2 所示。

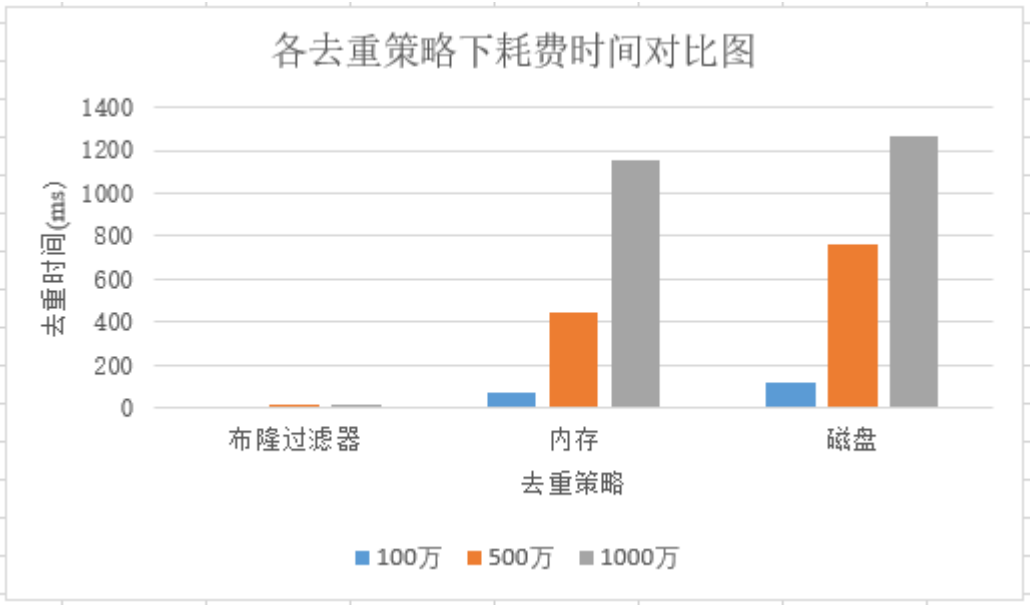

图 5-2 各去重策略下耗费时间对比图

由表 5-3 和图 5-2 可知，虽然基于传统的磁盘和内存 URL 链接去重不会出现像布

隆过滤器误判的情况，但是在判断该 URL 是否重复的步骤上所耗费的时间却远远高于基于布隆算法所耗费的时间。现实条件下，大规模抓取网页时，漏掉些许网址其实对整个抓取结果影响不大，故基于布隆过滤器的 URL 去重在海量数据下远远优于基于传统的磁盘或内存 URL 去重。

图 5-3 展示了基准网站 URL 数为 40 万，60 万，80 万，100 万，120 万时，通过标准布隆过滤器去重时产生的误判数，其中位数组大小为 1000 万，哈希函数个数 k 为 4。

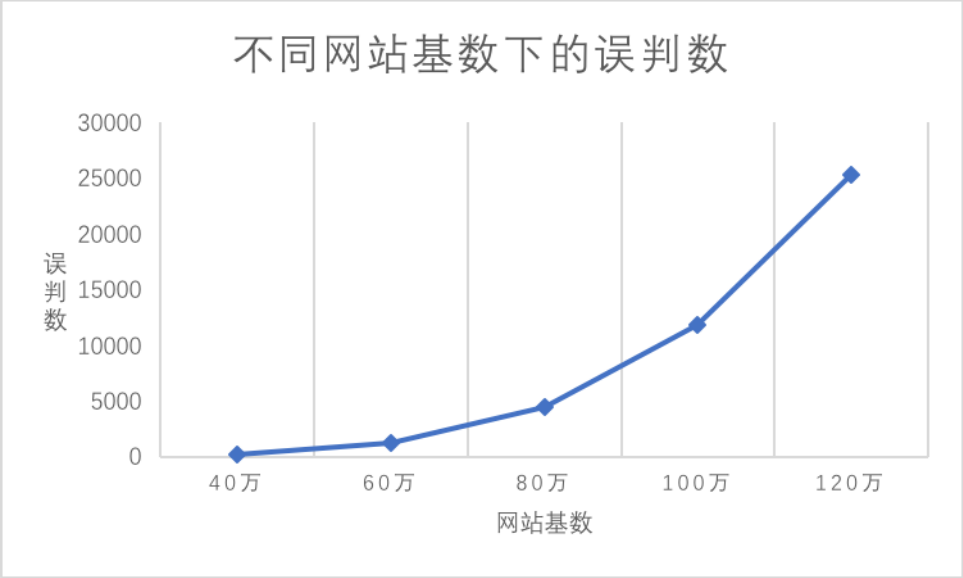

图 5-3 不同网站基数下的误判数

由图 5-3 所示，随着网站 URL 数量不断增多，由标准布隆过滤器去重产生的误判数不断增多，为了减少过滤器产生的误判数，本文采用基于多组布隆过滤器算法下的网页去重策略，图 5-4 展示了不同布隆过滤器个数下的误判率。

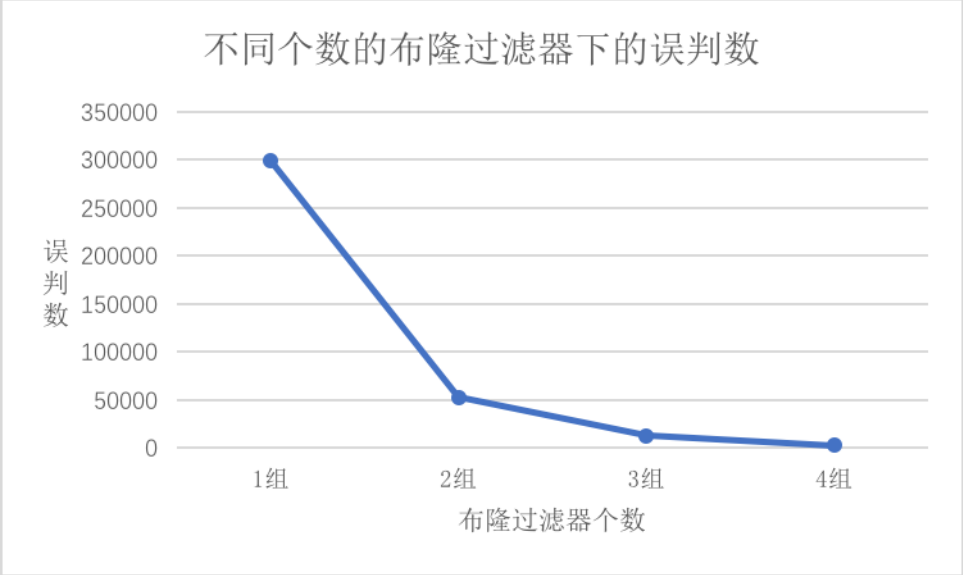

图 5-4 不同个数的布隆器误判数

上图展示了布隆过滤器个数为 1、2、3，4 组时，基准网站 URL 数为 300 万下误识别的 URL 个数，由图可知，基于两个布隆过滤器的 URL 去重在误判率上就已经非常低了，并且随着实验中布隆过滤器个数的增加，其误判的 URL 个数还随之减少，故采用基于多组布隆过滤器的 URL 去重方案能降低由单个布隆过滤器 URL 去重所造成的误判率。

5.2.3 爬虫采集速度测试

本小节主要负责测试在不同爬虫节点数下，系统抓取网页的速度。实验以凤凰网为目标站点，以广度优先策略遍历其网站下所有的链接，并每隔十分钟统计一次在 1 个爬虫节点，2 个爬虫节点以及 3 个爬虫节点下抓取的网页数量。实验结果如图 5-5 所示。

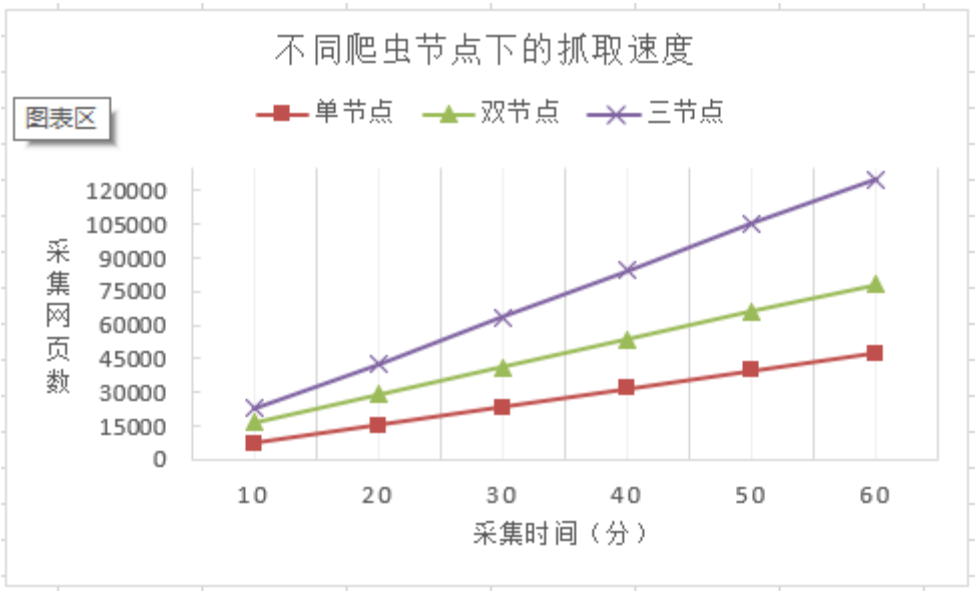

图 5-5 不同爬虫节点下的抓取速度对比

从图 5-5 中可以看出，随着 Scrapy 爬虫个数的增加，系统的整体的抓取速度逐渐加快，单节点的爬虫平均采集速度为 790 页/min，双节点的爬虫平均采集速度为 1304 页/min，三个节点的爬虫平均采集速度为 2087 页/min。需要注意的是，并不是在任何条件下爬虫节点数越多，系统的采集的速度越快。当系统中开启的爬虫节点数到达一个量后，其整体的采集速度会逐渐趋于平缓，其原因在于系统的带宽有限，当爬虫占用完所有的带宽后，即使开启再多的爬虫也无法提高其抓取速度，反而可能出现爬虫节点数过多，导致 CPU 负载过重，采集速度下降的现象。故选择一个合适的爬虫节点数，可以在达到良好的抓取效率的同时减少对系统资源的消耗。

5.3 系统展示

本节将通过模拟用户的操作来演示一次完整的数据抓取流程。首先通过首页面配置抓取任务基本信息，任务信息包括待抓取的 URL，网站对应的采集规则，采集的页面深度，任务的优先级以及爬虫的类型，如果该网站需要登录才能采集的话，还要配置其对应的 Cookie。图 5-6 为采集配置页面。

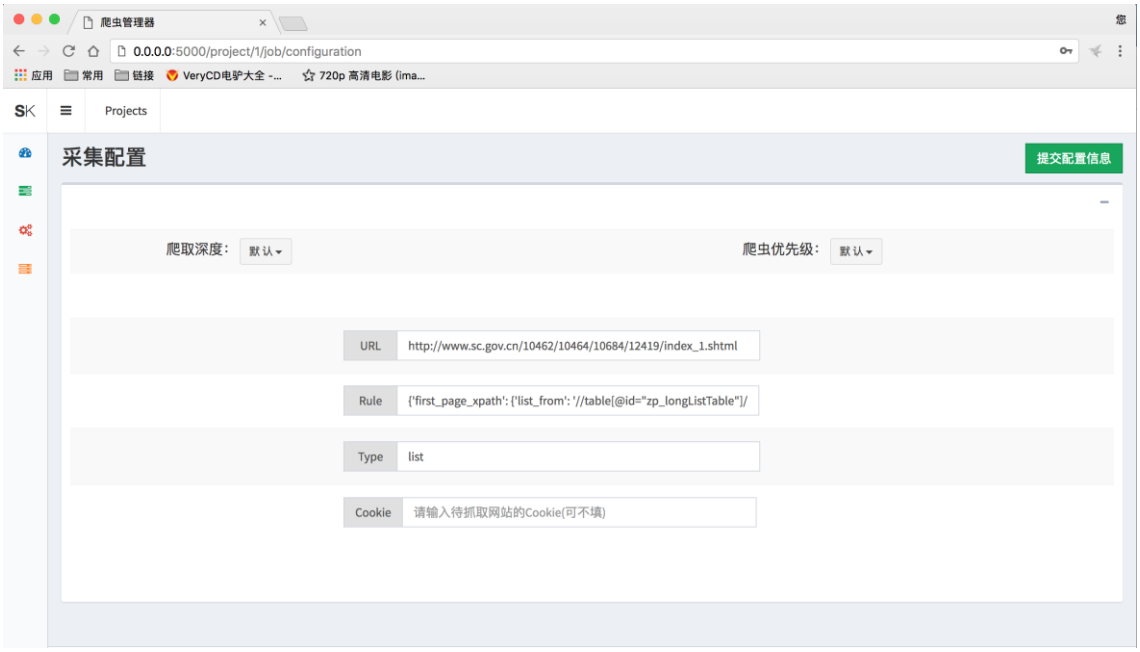

图 5-6 配置子页面

当提交基本的抓取信息后，通过爬虫管理子页面启动爬虫，爬虫启动服务对应 4 个参数，Spider 为爬虫的类型，Priority 为爬虫启动任务的优先级，Args 为传递给 Scrapy 爬虫的参数，最后一项为选择哪台主机上的爬虫启动。页面效果如图 5-7 所示。

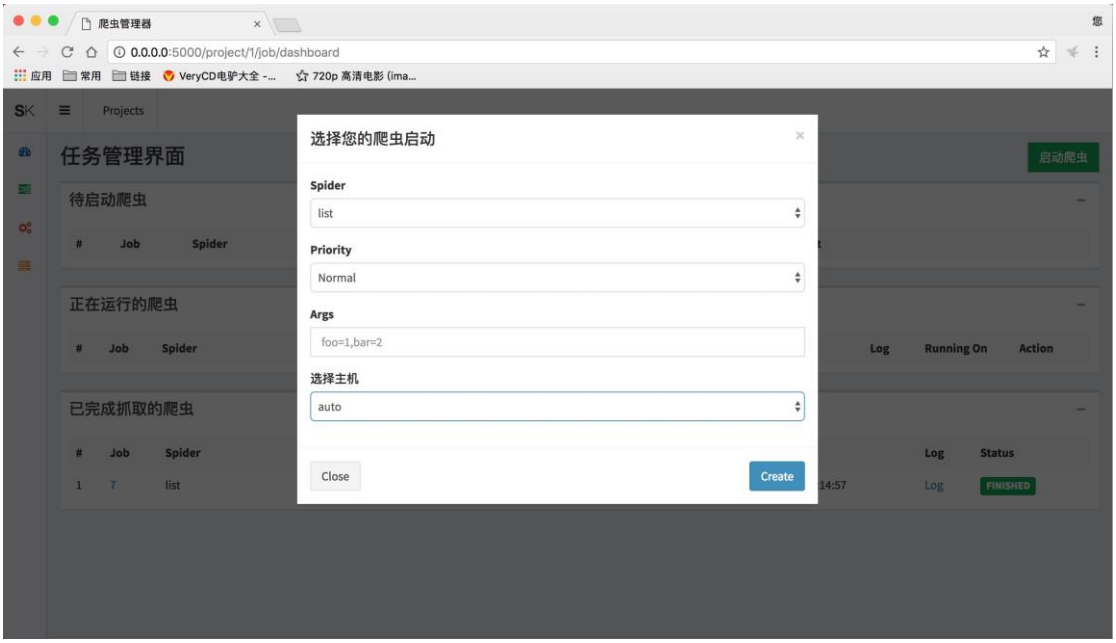

图 5-7 爬虫启动服务

当对应的爬虫启动后，用户可以通过状态页查看当前爬虫运行的状态，状态页由三部分组成：待启动爬虫，正在运行的爬虫以及已完成抓取的爬虫，图 5-8 展示了当前爬虫正在运行的状态，以及该爬虫的一些基本信息，如类型，运行时间，输出日志等等，同时如果想停止爬虫，可以通过点击 Stop 按钮停止爬虫。

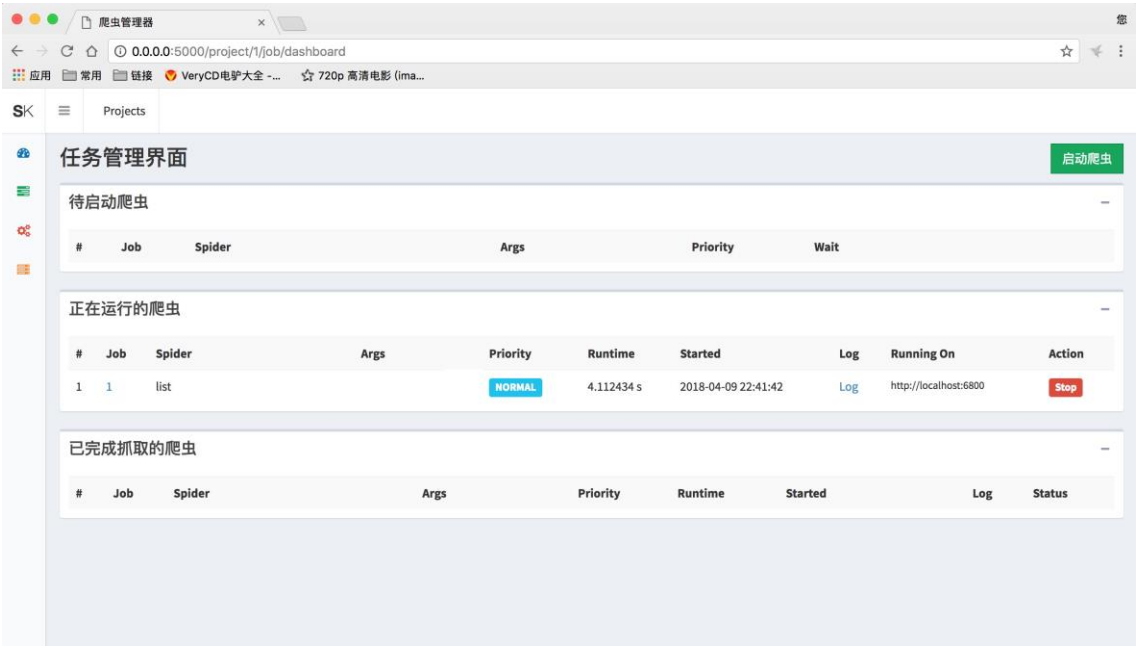

图 5-8 爬虫运行状态

爬虫按照用户填写的采集规则提取页面中的信息，并最终存入 MongoDB 中，如图 5-9 所示。

图 5-9 数据展示

5.4 本章小结

本章主要为系统的测试和展示部分，首先介绍了系统整体的运行环境，然后分别对任务调度器、过滤器以及爬虫采集速度三个方面进行性能测试，最后从用户的角度展示了如何使用该系统完成数据采集工作。

第六章 总结与展望

6.1 总结

本文设计并实现了一个基于 Scrapy 的分布式网络爬虫系统，该系统从总体上采用主从的设计架构模式，主节点负责整个爬虫节点的任务调度分配，从节点（群）负责爬虫相关工作。其中主节点主要由任务调度器、限速器以及过滤器三部分组成，任务调度器根据本文提出的权值计算公式并结合动态反馈任务调度策略对总爬虫任务队列中的爬虫抓取任务进行分发，确保系统中各个爬虫节点在任意时间段内都能负载均衡；限速器能按照用户设定的访问频率控制集群中所有的爬虫节点对站点的访问速度，其中基于 IP 的限速策略限定了同一台机器中的爬虫节点访问某站点的访问频率，基于爬虫类型的限速限定以同一类型的爬虫节点访问某站点的频率；过滤器负责整个抓取过程中 URL 去重的工作，鉴于系统采集的 URL 链接数量巨大，本文提出了基于布隆过滤器算法的 URL 去重策略，该策略相比传统的基于内存的 URL 去重，从时间和空间上都得到了大幅度的提升。从节点主要由 Scrapy 爬虫和爬虫管理器两部分组成，其中 Scrapy 爬虫负责网页数据采集的工作，为了使 Scrapy 框架支持分布式并行抓取，本文结合 Redis 数据库设计了一个带优先级的共享队列，并替换框架内部的调度器子模块，使其能与共享队列交互实现各个爬虫节点并行抓取任务。同时为了使系统中的爬虫更具灵活性，本文实现了一个带爬行规则的数据采集模块，该模块能根据用户配置的抽取规则抓取网页中固定位置的数据。最后为了用户能方便的管理各个节点中的 Scrapy 爬虫，本文基于 Twisted 框架设计并实现了一个基于异步任务响应的爬虫管理器，用户能通过该管理器方便的控制各个爬虫节点的启动、停止，同时还能时刻监控各个爬虫节点当前运行状态。

6.2 后续工作展望

本文目前为止基本实现了分布式网络爬虫的功能，但是系统中还是存在一些可以优化的环节，后续可以从以下两点入手：

1. 数据存储优化。本文采用 MongoDB 集群来存储用户采集的数据，用户每个抓取任务采集到的数据都单独存放在一个集合中，随着业务量不断增大，集合数也越来越多，而集群中的 mongos 会存取集合的最新元数据信息，太多的集合对

象容易引起 mongos 内存使用过大。同时采用 MongoDB 自动分片技术后，会将数据块按照一定的策略存储到各个分片节点上，为了保证各个分片节点负载均衡，会将数据块在节点间迁移，当数据量较大时，频繁的迁移会导致 CPU 占用率过高，故后续可以在数据存储优化上进一步研究。

2. 目前大多数网站都采用了 Ajax 技术来构建自身的前端页面，这样用户就不需要刷新页面来获取新的内容。本文只是单纯的使用 Scrapy 框架自带的 Splash 服务来处理一些简单的 JS 操作，而针对使用 Ajax 技术的网页目前还没有一套完整的解决方案，故后续可以从这方面着手研究。

致　谢

时间过的飞快，三年的电子科技大学生活一晃就过了，回想起着三年的生活，喜忧参半，喜的是遇到一群志同道合的朋友和负责的老师，忧的是自己未来的发展的迷茫。

这里我最想感谢的还是我的导师刘丹老师，刘丹老师不仅是我们学术上的指导老师，同样也是我们生活上的朋友，在项目上中遇到的很多问题，刘老师都耐心的给我们讲解，在选题，开题到最后的论文修改，都非常的认真负责。

同时我也想感谢我的师弟师妹们，项目中如果没有他们的努力，也不可能像现在一样一帆风顺，现在想起来大家一起通宵调 Bug 的日子确实也是另一种滋味。

我要感谢我的父母，他们总是默默的付出，在生活上给与我最大的帮助，在学习上也给我很多建议。

最后，由衷的感谢各位评审老师在百忙之中抽出时间来参与我的论文评审和答辩。

参考文献

[1] 方启明, 杨广文, 武永卫,等. 面向 P2P 搜索的可定制聚焦网络爬虫[J]. 华中科技大学学报(自然科学版), 2007, 35(s2):148-152.

[2] Heydon A, Najork M. Mercator: A scalable, extensible Web crawler[J]. World Wide Web-internet & Web Information Systems, 1999, 2(4):219-229.

[3] Boldi P, Codenotti B, Santini M, et al. UbiCrawler: a scalable fully distributed Web crawler[J]. Software—practice & Experience, 2004, 34(8):711-726.

[4] Mohr G. The Internet Archive's Web Collection and Open Source Crawler[C]// 数字图书馆—促进知识的有效应用国际研讨会. 2004.

[5] Cambazoglu B B, Turk A, Aykanat C. Data-Parallel Web Crawling Models[J]. 2004, 3280:801-809.

[6] Quoc D L, Fetzer C, Felber P, et al. UniCrawl: A Practical Geographically Distributed Web Crawler[J]. 2015:389-396.

[7] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[M]. ACM, 2008.

[8] 孟涛, 闫宏飞, 王继民. 一个增量搜集中国 W eb 的系统模型及其实现[J]. 清华大学学报(自然科学版), 2005, 45(9):156-160.

[9] 高强. 基于 Redis 的分布式爬虫框架的设计[J]. 农业网络信息, 2017(8).

[10] 郑泳. 基于广度优先搜索的网络蜘蛛设计[J]. 软件导刊, 2010, 09(7):122-123.

[11] 叶允明, 于水, 马范援,等. 分布式 Web Crawler 的研究:结构、算法和策略[J]. 电子学报, 2002, 30(12a):2008-2011.

[12] 唐红, 吴勇军, 赵国锋. 用于特定流匹配的随机矩阵映射 Hash 算法研究[J]. 通信学报, 2007, 28(2):17-22.

[13] 付志辉. 分布式爬虫的动态负载均衡方法研究[D]. 哈尔滨工业大学, 2014.

[14] 李婷. 分布式爬虫任务调度与 AJAX 页面抓取研究[D]. 电子科技大学, 2015.

[15] 荣晗. 基于分布式的网络爬虫系统的研究与实现[D].电子科技大学,2017.

[16] 程锦佳. 基于 Hadoop 的分布式爬虫及其实现[D]. 北京邮电大学, 2010.

[17] Zhang Z, Dong G, Peng Z, et al. A Framework for Incremental Deep Web Crawler Based on URL Classification[C]// International Conference on Web Information Systems and Mining. Springer, Berlin, Heidelberg, 2011:302-310.

[18] Batsakis S, Petrakis E G M, Milios E. Improving the performance of focused web crawlers[J].

Data & Knowledge Engineering, 2009, 68(10):1001-1013.

[19] Edwards J, Mccurley K, Tomlin J. An Adaptive Model for Optimizing Performance of an Incremental Web Crawler[J]. Association for Computing Machinery, 2001:106-113.

[20] Wolf, J. L, Squillante, et al. Optimal crawling strategies for web search engines[J]. 2002:136-147.

[21] 杨智明. 图的广度优先搜索遍历算法的分析与实现[J]. 农业网络信息, 2009(12):136-137.

[22] 龚建华. 深度优先搜索算法及其改进[J]. 现代电子技术, 2007, 30(22):90-92.

[23] 余兆钗, 傅化权. 一种改进的最好优先搜索策略算法[J]. 科技视界, 2014(33):89-89.

[24] Kritikopoulos A, Sideri M, Stroggilos K. Crawlwave: a distributed crawler[C]// Hellenic Conference on Artificial Intelligence. 2004.

[25] Majumdar S. Scheduling in client-server systems[C]// Fourteenth ACM Symposium on Principles of Distributed Computing. ACM, 1995:262.

[26] Karshmer A I, Bledsoe C, Stanley P. The Architecture of a Comprehensive Equation Browser for the Print Impaired[J]. Lecture Notes in Computer Science, 2004, 3118(17):614-619.

[27] Classification R R, Furnkranz J, Singer Y. Round Robin Classification[J]. Journal of Machine Learning Research, 2001, 2(4):721-747.

[28] Katevenis M, Sidiropoulos S, Courcoubetis C. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip[J]. IEEE Journal on Selected Areas in Communication, 1991, 9(8):1265-1279.

[29] Karger D, Sherman A, Berkheimer A, et al. Web caching with consistent hashing[J]. Computer Networks, 2011, 31(1116):1203 – 1213.

[30] 肖青. 基于最快响应调度法的电梯并联控制的研究[J]. 武汉职业技术学院学报, 2016, 15(5):89-91.

[31] Kouzisloukas D. Learning Scrapy[J]. 2016.

[32] Redis 设计与实现[M]. 机械工业出版社, 黄健宏, 2014

[33] 分布计算系统[M]. 高等教育出版社, 徐高潮等[编著], 2004

[34] 陈任飞, 吕玉琴, 侯宾. 基于 Flume/Kafka/Spark 的分布式日志流处理系统的设计与实现[J]. 2015.

[35] Hadoop 权威指南[M]. 清华大学出版社, (美) 怀特 (White), 2011.

[36] Liu S L. The Strategy of Coping with Anti-Crawler Website[J]. Computer Knowledge & Technology, 2017.

[37] 王光磊. MongoDB 数据库的应用研究和方案优化[J]. 中国科技信息, 2011(20):93-94.

[38] 童进, 施卫丰, 李际涛等. 一种快速加权轮询调度方法及快速加权轮询调度器和装置: CN.

[39] Pagh, Anna, Pagh, et al. An optimal Bloom filter replacement[J]. In Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA, 2005(2):823-829.

[40] Liza Daly. 使用由 Python 编写的 lxml 实现高性能 XML 解析[J]. 2009.

[41] Liu L, Shi J, Liu X. Web Information Extraction Algorithm Based on Ontology and DOM Tree[C]// Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on. IEEE, 2010:1 - 4.

[42] Lavelli A, Califf M E, Ciravegna F, et al. Evaluation of machine learning-based information extraction algorithms: criticisms and recommendations[J]. Language Resources & Evaluation, 2008, 42(4):361-393.

[43] 方金卫. 基于 Hadoop 的基础教育资源的存储和处理[D]. 武汉理工大学, 2015.

[44] Kinder K. Event-driven programming with Twisted and Python[J]. Linux Journal, 2005, 2005(131):6.

[45] David Mertz. 使用 Twisted Matrix 框架来进行网络编程, 第 1 部分[J].

攻读硕士学位期间取得的成果

[1] Yuhao Fan. Design and Implementation of Distributed Crawler System Based on Scrapy. 2017 4th International Conference on Material Science, Environment Science and Computer Science (MSESCS 2017) .