专业修依硕士论女



论文题目:

棊于微服务架构的分布式爬虫系统设计与应用

|  |  |  |
| --- | --- | --- |
| 学 | 号 | 1317044716 |
| 姓 | 名 | 葛又嘉 |
| 导 | 师 | 章韵 |
| 专业学位类别 | | 工程硕士 |
| 类 | 型 | 非全日制专业学位 |
| 专业 | （领域） | 计算机技术 |
| 论文提交日期 | | 2020年3月 |

Design and Application of Distributed Crawler  
System Based on Micro-Service Architecture

Thesis Submitted to Nanjing University of Posts and

Telecommunications fbr the Degree of

Master of Engineering



By

Ge Youjia

Supervisor: Prof. Zhang Yun

March 2020

南京邮电大学学位论文原创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。 尽我所知，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，也不包含为获得南京邮电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了谢意。

本人学位论文及涉及相关资料若有不实，愿意承担一切相关的法律责任。

研究生学号:

研究生签名" 日期：.

南京邮电大学学位论文使用授权声明

本人承诺所呈交的学位论文不涉及任何国家秘密，本人及导师为本论文的涉密责任并列 第一责任人。

本人授权南京邮电大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档; 允许论文被查阅和借阅；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 釆用影印、缩印或扫描等复制手段保存、汇编本学位论文。本文电子文档的内容和纸质论文 的内容相一致。论文的公布（包括刊登）授权南京邮电大学研究生院办理。

非国家秘密类涉密学位论文在解密后适用本授权书。

研究生签名： 导师签名： 日期：

摘要

随着社会经济的飞速发展，互联网应用己经成为人们日常生活中的一部分，及时准确地 获得网络应用数据也变得越来越重要。网络爬虫作为一种通过对互联网数据进行精确信息抓 取来满足用户对特定数据需求的计算机技术，也面临着巨大的机遇与变革，在这样的技术背 景下早期的网络爬虫技术巳难当重任。

论文分析了已有的爬虫系统在技术实现上的不足之处，根据大数据时代的海量页面数据 抓取需求设计了一套基于微服务架构的分布式爬虫系统，釆用了 Spring Cloud微服务框架， 实现了爬虫系统不同微服务模块之间的架构隔离；不同微服务之间釆用消息中间件或者远程 调用进行数据通信，以此来提升分布式爬虫系统的可用性。

本课题的主要工作包括设计并实现了分布式系统环境下的全局唯一 ID生成算法；基于布 隆过滤器和Redis的URL双重判重机制；分布式场景下基于客户端的负载均衡算法策略；分 布式场景下应对海量请求的限流算法策略；利用基于线程池的多线程爬取以及釆用动态代理 池技术应对反爬虫来提升爬虫的抓取效率和成功率；页面解析微服务釆用基于CSS选择器的 自定义页面数据提取以及基于Redis的消费端防止重复消费机制；应对海量数据存储场景实 现了基于MongoDB的数据库副本与分片、基于Redis哨兵集群与持久化存储以此确保高可用。

通过实验测试，基于微服务架构的分布式爬虫系统，既能承载海量用户的爬虫请求，又 能满足用户的不同数据提取需求，同时系统的可维护性和可拓展性能力又强于传统爬虫系统, 达到了系统设计要求。

关键词：微服务，分布式爬虫，消息中间件，数据库

**Abstract**

With the rapid development of social economy, Internet applications have become a part of people's daily life, and it is becoming more and more important to obtain network application data in a timely and accurate manner. As a computer technology that meets the specific data needs of users through accurate information capture of Internet data, web crawlers also face huge opportunities and changes. Under such technological background, early web crawler technology has been already unable to assume this task.

The thesis analyzes the deficiencies of existing crawler systems in technical implementation, and designs a distributed crawler system based on microservice architecture based on the massive page data crawling requirements which use Spring Cloud framework, this crawler system achieves architectural isolation between different microservice modules ; message middleware or remote procedure call is used for data communication between different microservices to improve the availability of distributed crawler systems.

The main work of this thesis includes designing and implementing a globally unique ID generation algorithm in a distributed system environment; URL double check based on Bloomfilter and Redis; client load balance algorithm strategy in distributed scenario; the limit algorithm strategy to deal with massive requests in the distributed scenario; use multi thread crawling based on the thread pool and dynamic proxy pool technology to deal with anti-crawler to improve the crawling efficiency and success rate of the crawler; the page resolution microservice uses CSS selector based custom page data extraction and consumer end prevention of repeated consumption mechanisms based Redis; using database replication and data sharding based MongoDB to deal with massive data storage scenarios, Redis sentinel cluster and persistent storage to ensure high availability.

Through experimental testing, the distributed crawler system based on the micro-services architecture can not only carry the crawler requests of a large number of users, but also meet the different data extraction requirements fi'om the users. At the same time, the maintainability and expandability of the system are stronger than the traditional crawler system, which meets the system design requirements.

Key words: MicroService, **Distributed** Crawler, Message Queue, Database

目录

[第一章绪论 1](#bookmark30)

[1.1课题研究的背景与意义 1](#bookmark33)

[1.2国内外研究现状与发展 2](#bookmark36)

[1.3主要研究内容 3](#bookmark39)

[1.4论文内容组织结构 4](#bookmark42)

[第二章基于微服务架构的分布式爬虫技术 5](#bookmark45)

[2.1 微月艮•务框架 Spring Cloud 5](#bookmark48)

[2.2消息队列Kafka 7](#bookmark57)

[2.3文本提取工具Jsoup及网络框架HttpClient 8](#bookmark66)

2.4存储数据库MongoDB*及*Redis 10

[2.5本章小结 12](#bookmark72)

[第三章分布式爬虫系统总体设计 13](#bookmark75)

[3.1系统需求分析与设计目标 13](#bookmark78)

[3.2系统总体设计架构及运行机制 14](#bookmark85)

3.2.1分布式爬虫系统各微服务模块划分 14

3.2.2系统整体架构图及运行流程 15

[3.3爬虫系统的各子部分功能介绍 17](#bookmark88)

3.3.1任务创建与査询可视化界面 17

3.3.2任务请求分发微服务 17

3.3.3任务请求预处理微服务 18

3.3.4页面抓取微服务 21

3.3.5页面解析微服务 22

[3.4数据库设计 24](#bookmark94)

[3.5本章小结 26](#bookmark103)

[第四章分布式爬虫系统核心功能实现 27](#bookmark106)

[4.1任务请求分发微服务 28](#bookmark109)

4.1.1客户端负载均衡算法处理 28

4.1.2客户端请求限流机制 31

4.1.3微服务熔断器 33

[4.2任务请求预处理微服务 34](#bookmark135)

4.2.1全局分布式唯一 ID生成算法 35

1. URL双重判重机制 37
2. Kafka集群副本配置 39

[4.3页面抓取微服务 40](#bookmark151)

4.3.1基于线程池的多线程抓取实现 40

4.3.2中间件发送消息防丢失实现 42

4.3.3爬虫动态代理池 43

[4.4页面解析微服务 44](#bookmark154)

4.4.1基于提取规则的页面文本数据提取 44

4.4.2基于Kafka的页面数据预防重复消费解析 45

[4.5数据存储 46](#bookmark157)

4.5.1实现基于MongoDB数据库的数据存储高可用机制 46

4.5.2实现基于Redis数据库的数据存储高可用机制 48

4.6系统测试 50

4.6.1测试环境 50

4.6.2系统功能测试 51

[4.7本章小节 53](#bookmark160)

[第五章总结与展望 55](#bookmark163)

[5.1论文工作总结 55](#bookmark166)

[5.2工作展望 56](#bookmark175)

[参考文献 57](#bookmark181)

[附录1攻读硕士学位期间申请的专利 59](#bookmark223)

[致谢 60](#bookmark226)

第一章绪论

1.1课题研究的背景与意义

随着近年来互联网相关技术的快速发展以及在人们日常生活中的日益普及，互联网成为 人们获取信息的重要途径，而信息数据的快速增长，用户对信息获取的速率和精确性提出了 更高的要求，因此网络爬虫成为了一项运用相当广泛的技术。网络爬虫是通过对互联网数据 进行精确抓取URL和提取页面数据来满足用户对特定数据需求的技术。据统计，当今网络世 界中的将近80%的流量来自于各大互联网公司或者个人开发者所开发的网络爬虫应用。例如 谷歌公司的技术团队就开发了包括google新闻爬虫，google图片爬虫，google视频爬虫等 各个不同应用场景的爬虫软件，依靠其精良的算法和较快的反应速度来满足Google用户的海 量搜索场景。

随着互联网技术的日益发展，网页的内容越来越复杂，与之相应的人们对网页信息提取 的要求越来越高，从而催生出了种类繁多的爬虫系统，当前的爬虫系统根据其体系结构的不 同可以区分为通用抓取型爬虫，面向主题特定领域爬虫卩J,垂直爬虫⑵，深度抓取爬虫⑶等不 同类型。常见的搜索引擎如国内的百度，国外的Google,其本质上相当于一个通用性爬虫技 术，主要在应用层面为用户提供根据查询条件的模糊匹配，而不是专注于为用户提供精细化 的数据提取和分析。

早期的爬虫系统主要基于单机的设计，无法满足大数据量的爬虫釆集任务。随着软件技 术的进步，爬虫系统的技术架构也在不断的进化，单机单节点爬虫也逐渐演进到分布式爬虫。 例如美国著名电商公司eBay的爬虫技术部门，依靠其分布式爬虫系统，每天的网页爬取数量 可以达到数千万级别。在这样的海量页面爬取的需求场景下，对爬虫系统的分布式部署，爬 取任务调度，网页数据的存储，系统的可维护性与可升级性都有着巨大的技术要求。除此之 外，一个优秀的爬虫系统还要具备应对当前比较主流的反爬虫措施【4-5〕，目前市面上比较常见 的反爬虫措施有基于request请求头的爬虫鉴别，基于cookie的反爬拦截，基于访问频率的反 爬拦截，基于验证码的反爬拦截，基于设备指纹的反爬拦截等各种反爬措施。由此可见，一 个优秀的爬虫系统不仅要在工程层面满足系统的高性能和高可靠性，还要能够在业务层面对 常见的反爬虫措施提出相应的解决方案〔虬

随着软件开发架构的日益演进，人们提出了一种全新的软件开发架构，这种理念被称为 微服务开发理念【刀。微服务是一种软件架构的思想和风格，相对于传统的SOA软件架构，其

核心理念是将原本的一站式应用依据其不同功能划分成一组小的服务，从而彻底的实现物理 以及逻辑的解耦合。每个服务运行在自己的进程中，服务与服务之间通过一些轻量级的通信 方式进行相互通信，甚至于各个服务可以使用不同的语言进行开发，而对整个微服务系统则 可以通过一个完善的监控平台去实现全链路监控。由此可见微服务架构在当今软件开发日益 复杂，系统日益庞大的情况下能够有效的改善系统的可维护性以及分布式环境下的系统快速 部署能力。

课题通过将分布式爬虫系统同微服务架构相结合，以爬虫系统的不同功能作为依据将分 布式爬虫系统划分出不同的微服务模块，例如请求调度微服务，页面下载微服务，请求预处 理微服务等。在各个服务之间釆用熔断措施，避免出现传统软件架构中某部分业务逻辑崩溃 而导致整个系统不可用的场景出现。开发一个既能承载海量用户⑹的爬虫请求，又能满足用 户的不同数据提取需求，同时系统的可维护性和快速部署能力又强于传统软件的分布式爬虫 系统。

1.2国内外研究现状与发展

针对网络爬虫相关技术，从互联网开始在人们日常生活中日益普及开始，就有许多国内 外研究人员和开发团队都对这方面的技术进行了相当长时间的研究，很多世界著名的商业化 搜索引擎，例如美国的Google搜索，雅虎搜索，中国的百度搜索等，其核心功能本质就是一 个网络爬虫。搜索引擎的釆集范围之广，搜索结果的广泛性已经经过时间的验证。

目前国外比较优秀的网络爬虫系统有Nutch ,WebCollector, Scrapy等，这些优秀的爬虫 框架都有其各自的优秀特点。

Nutch是一款基于Java语言开发的网络爬虫系统，它的特点是底层釆用了基于大数据开 源框架平台HadoopW。】作为处理机制，相对商用的搜索引擎，Nutch作为开源项目，其源代码 开发更加透明。Nutch致力于让大多数使用者能够在短时间内依赖框架生成一套便捷的爬虫 搜索引擎，为了达到这个目的，Nutch需要能够做到对海量页面的抓取，对抓取到的网页维 护索引，支持对索引文件进行高频率的搜索，同时又能提供高效的搜索结果。Nutch主要分 为爬虫模块和搜索模块，爬虫模块主要用于抓取网页并为这些网页建立索引，搜索模块主要 用于利用索引结合用户查找的关键词进行结果匹配。Nutch依赖Hadoop的MapReduce算法实 现爬虫的分布式计算，而搜索模块则是依赖Lucene技术进行海量数据下的快速检索。

WebCollector是一款由Java语言开发，无须用户二次配置、支持基于内核的二次开发的 爬虫框架，该框架提供精简的高级API,只需少量代码即可实现一个功能强大的爬虫。框架 内核集成了 Jsoup,可进行精准的网页解析，2.x版本中集成了 selenium,可以模拟真实用户 的浏览器访问流程，从而达到处理动态JavaScript生成的数据。Webcollector可以自定义遍历 策略，可完成更复杂的遍历业务。也可以URL设置附加信息，可以实现深度获取，锚文本获 取，引用页面获取，post参数传递等，同时还支持插件机制，用户可定制过滤器等。同时内 置一套基于Berkeley DB的插件，适合处理时间周期较长和抓取数量级较大的任务，并具有 断点爬取功能1川。

Scrapy是一个基于Python语言开发的爬虫框架，它被设计为用于爬取网页数据，提取网 页中的结构性数据卩214］。该框架底层使用了基于异步通信的网络框架，相比于同步网络通信， 加大了爬虫的吞吐性，同时利用内置的各种中间件接口，可以灵活的配置爬虫系统。Scrapy 框架中的核心元素有调度器，下载器，spider,管道器，下载中间件等，同时也支持自定义插 件拓展，支持Redis存储等卩A。通过爬虫中的起始URL构造request对象，将对象传递至爬 虫中间件，由爬虫引擎处理转发至爬虫调度器，最后由数据爬取引擎爬取数据，该框架支持 数据持久化和自定义过滤字段等功能。

在国内各大高校学者也对爬虫系统作了大量的研究，例如赵鹏程等研究人员设计了实现 对网络书籍网站爬取的相关爬虫系统3】，通过借助于自动化测试工具selenium实现模拟真实 浏览器访问【⑺，从而跳过反爬虫的限制，获取到真实的网页数据。在2018年，杨君开发了一 款基于Scrapy的可视化网页数据提取系统小】，该系统通过将原本黑盒的爬虫流程转化成了可 视化的网页数据流程，使得用户可以在可视化界面查看到具体的爬虫任务和运行结果。2019 年杜凤媛通过借助Scrapy-Redis框架以及selenium框架实现了一套基于分布式架构的爬虫系 统U9】，借助于分布式任务调度算法，大大提高了爬虫系统相比原有单机模式的抓取数量的提 升。

1.3主要研究内容

针对已有爬虫系统的优缺点，面对大数据时代的海量数据爬取场景，一个优秀的分布式 爬虫系统应该具备如下几个特征：系统的高可用性，高效的数据抓取速率，在分布式场景下 的快速拓展性等。基于以上需求，本课题的主要实现内容包括：

在任务请求分发微服务中，针对分布式场景实现了一套基于基于线性轮询、最小请求数 和响应时间权重的的负载均衡策略，该算法用于将爬虫请求分发至部署了任务请求预处理微 服务的不同机器，同时釆用了微服务架构中的断路器来实现微服务之间的服务安全机制；针 对短时间内大量爬虫任务请求涌入的场景，实现了一套基于令牌桶和Redis lua脚本的限流算 法。

在页面抓取微服务中实现了一套分布式环境下的1D生成算法用于给每个爬虫任务中的 数据提取规则生成一个全局唯一的分布式ID；对于URL判重釆用了基于布隆过滤器和Redis 进行URL的双重判重校验，使用kafka作为消息队列存储下发数据，并进行了集群化部署。

针对现代多核CPU的特性，釆用了基于线程池的多线程爬取；针对本系统的不同存储场 景，对MongoDB采用数据副本和数据分片的集群方式保证MongoDB的高可用和海量数据 场景场下的拓展，对Redis数据库釆用基于哨兵模式的集群方式保证高可用，并基于RDB和 AOF机制实现Redis的数据持久化机制。

1.4论文内容组织结构

本文共有5个章节，主要内容和叙述结构如下：

第一章为绪论，首先叙述了本文的研究背景和国内外研究现状，讲述了一些爬虫相关的 基础概念和应用场景，为本系统的设计提供了基础理论背景。

第二章为基于微服务架构的分布式爬虫所需要的基础技术，包括对微服务框架Spring Cloud进行相关介绍，包括采用微服务架构的工作原理以及优势。接着介绍了用于大数据量 场景下釆用的高性能，高吞吐性的消息队列Kafka,主要叙述了 Kafka的原理以及使用场景。 同时介绍了在爬虫系统中作为数据局持久化工具的文档型数据库MongoDB和内存数据库 Redis,对它们进行了基本的概念介绍和特点概述。

第三章为本系统的总体架构图和各微服务模块功能分析。本章详细介绍了系统的设计目 的以及为了达到相关的设计理念所设计出的系统实际架构图，并对系统中各个微服务模块的 设计以及运行流程做了初步介绍，给出了业务流程的设计流程图。

第四章为本系统各微服务模块的详细设计，分析了各模块的具体设计和实现方法，并分 析和解决了实现过程中的一系列问题，并针对这些问题阐述了具体的解决方案。

第五章为对基于微服务架构的分布式爬虫系统的工作内容总结，总体上概括了系统的现 状分析和研究，也阐述了本系统的可能存在的不足之处，给后续的分布式爬虫系统的研究提 供了一定的思路和研究方向。

第二章基于微服务架构的分布式爬虫技术

网络爬虫是一个遵循既定抓取规则从互联网自动且高效抓取数据的计算机应用程序，通 过自定义的规则获取网页中的结构化数据。本章主要介绍了基于微服务架构的分布式网络爬 虫系统中的一些重要技术，包含微服务框架Spring Cloud,消息中间件Kafka,文本提取工具 Jsoup,网络通信框架HttpClient,文档型数据库MongoDB,内存型数据库Redis等。

2.1微服务框架**Spring Cloud**

微服务是一种架构模式，它提倡将原有单一庞大的单体应用拆分成一组小的服务，服务 之间相互协调，互相通过http或者rpc接口调用。每个服务运行在其独立的进程中间。服 务与服务之间釆取轻量级的通信机制，每个服务都只围绕某一块特定的业务进行构建和部署。 而SpringCloud框架则为微服务提供了一整套的解决方案，可以快速满足微服务项目的快速 部署，水平拓展，独立访问，并且对这些微服务进行管理，而Spring Cloud框架中的相当一 部分组件来自于Netflix公司开源的微服务组件，这些组件在Netflix的业务场景下经历了高 并发场景的考验，是十分优秀的微服务组件。

Spring Cloud包含了一系列优秀的组件，主要有：

1. 服务注册中心Eureka

Eureka!20!是一个基于REST的定位服务，所有的微服务模块都可以通过指定的注册中心 IP地址，将该服务自身的服务信息注册至Eureka Server中，通过Eureka提供的可视化界面 可以方便的查看各个微服务的运行状况。同时每个服务都有自己的唯一实例ID,因为它们各 自有自己的基础信息如：不同的IP,所以它们的信息也需要注册到Eureka Server中。其他服 务调用它们的服务接口时，可以查看到多个该服务的实例信息，根据负载策略提供某个实例 的调用信息后，外部微服务调用者可以通过从Server获取信息直接调用该实例。在微服务架 构中每一个微服务都可以作为一个Eureka Cliento

1. 熔断器Hystrix

微服务架构中，原本的单体系统架构根据不同的业务边界拆分成了一个个单独的微服务 单元，各个独立的微服务单元应用间通过基于远程调用或者HTTP通信的服务注册与服务订 阅的方式互相依赖。由于每个单元都运行在不同的进程中，依赖远程调用的方式执行，这样 就有可能因为网络原因或是依赖服务自身系统故障或者网络故障出现服务调用失败或延迟。

若此时来自微服务调用方的请求数量不断增加，最终出现因为等待岀现故障的依赖方响应而 形成大量请求任务积压，进而导致线程资源无法释放，最终导致微服务自身的瘫痪，并因为 单个服务故障的蔓延最终导致整个系统的雪崩瘫痪。而Hystrix则通过线程池隔离等技术实现 了服务降级，服务焰断以及服务监控等强大功能，从而避免了单个服务出现故障导致的连锁 故障。

1. 声明式服务调用客户端Feign

在微服务系统中服务与服务间的调用通常釆用HTTP或者RPC的调用方式，而Feign則 则是Spring Cloud提供的声明式服务调用客户端，通过实现接口方法并使用注解来完成对服 务接口的绑定，服务间的远程调用通过接口动态代理方式实现。

1. 路由网关zuul

在微服务系统中在服务的最外层会有一个统一的API网关层，这个网关层可以提供很多 的功能，例如动态路由，请求过滤与拦截，监控，限流等操作，通过API网关的存在，可以 避免具体业务模块的微服务系统直接暴露，而是通过网关去执行请求拦截，路由转发，避免 业务系统受到外部恶意请求的攻击。

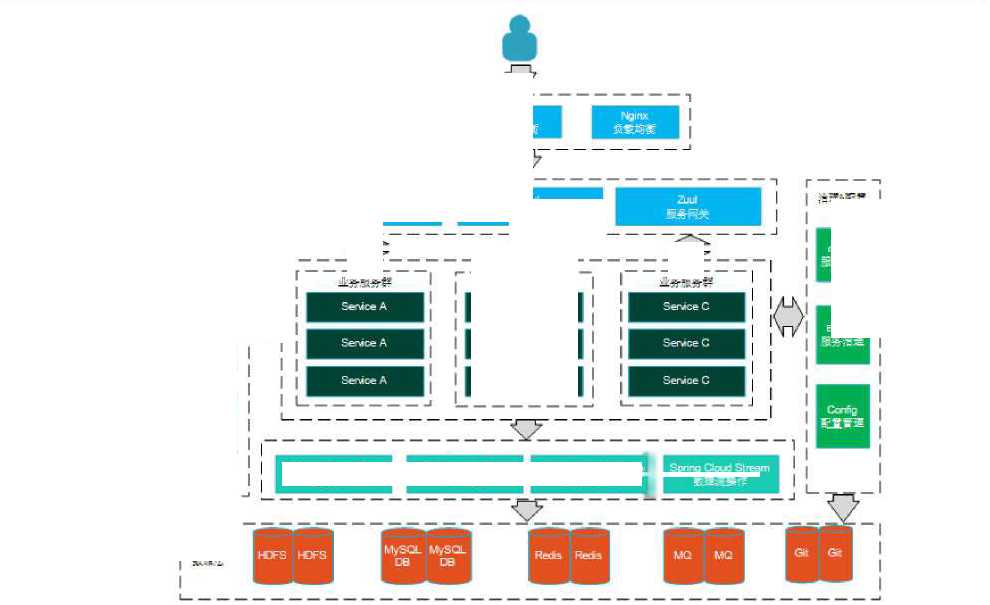
1. 配置中心config

spring cloud config作为spring cloud微服务体系中的用户配置中心，为微服务应用提供集 中化的外部配置，其能够很好的管理程序配置中日益增多的参数配置，服务器的地址，灰度 开关配置。当这些配置更新后，能够做到实时生效，并且对灰度发布，实现了集中化管理集 群管理配置，大大方便了系统的配置和维护管理。

1. 全链路监控sleuth

在传统的SOA架构体系中，系统调用层级不多，调用关系也不复杂，一旦出现问题，根 据异常信息可以很快定位到问题模块并进行排查，但在微服务架构中实例的树木成千上万， 调用关系呈网格状，在这种情况下一个完善的调用链路监控框架对于开发人员来说是必备的, spring cloud sleuth是spring cloud体系的分布式调用链路解决方案，sleuth对于大部分用户来 说是透明的，系统间的交互信息都能被自动釆集，用户可以通过日志文件获取链路数据，也 可以将数据发送给远端监控进行统一收集展示。

一个典型的微服务架构解决方案如图2.1所示：



堂竺说-

L

ME特

-•二「，'\* '

Nginx

Hysthx

Seutft

ZjU

Turbine 実割E庶

Eureka

consul

图2.1典型微服务架构图

Zuul

*一眨*

*一眨*

| **Sodng Qoud Bts** | **I SpAngOoudTask** | **|s(jnno Qoud *OeAa* Howl** |
| --- | --- | --- |
|  | J ,务寳. | 1 [ |

Spnno GolkJ Stresrr

,枣—

2.2消息队列**Kafka**

Kafka起初是由Linkedln公司釆用Scala语言开发的一个多分区、多副本，且基于 ZooKeeper协调的分布式消息系统，Kafka定位为一个分布式流式处理平台，它以高吞吐、 可持久化、可水平扩展、支持流数据处理等多种特性而被广泛使用。Kafka把消息持久化到 磁盘，相比于其他基于内存存储的系统而言，有效地降低了数据丢失的风险。也正是得益于 Kafka的消息持久化功能和多副本机制，我们可以把Kafka作为长期的数据存储系统来使用。

下面对Kafka中的一些核心概念加以介绍：

1. Topic （主题）

Kafka中的消息以主题为单位进行归类，每个topic还可以细分成多个Partition （分区）， 同一主题下的不同分区包含的消息是不同的，分区在存储层面可以看作一个可追加的日志 （Log）文件，消息在被追加到分区日志文件的时候都会分配一个特定的偏移量（offset）。

1. Offset （偏移量）

offset是消息在分区中的唯一标识，Kafka通过它来保证每一条消息在分区内的顺序性。 同时offset也作为消费消息时的位置偏移量。Offset记录了某个user group在某一个分区中的 当前以及消费到达的位置。Offset主要有以下几个概念：（1） Last Committed Offset： consumer group最新一次commit的offset,表示这个group已经把之前的数据都消费成功了 °（2）Current

Position： consumer group 当前消费数据的 offset,也就是说，Last Committed Offset 到 Current Position之间的数据己经拉取成功，可能正在处理，但是还未commite （3） Log End Offset： 记录底层日志的下一条消息的offset,对于生产者来说，就是下一条即将被插入的消息的offset。

（4） High Watermark：已经成功备份到其他副本的最新一条数据的offset,消费者是无法消费 这部分未提交的消息的。

1. Consumer （消费者）

消费者，也就是接收消息的一方。消费者连接到Kafka上并接收消息，进而进行相应的 业务逻辑处理。每一个消费者都属于一个消费群组，对于同一消费群组下的消费者，某个主 题中的某条消息只能被一个消费者消费,每次消费的时候都会根据偏移量。ffset去读取消息。

1. Broker （服务代理节点）

对于Kafka而言，Broker在物理上可以作为一个独立的Kafka运行节点或Kafka服务 实例。大多数情况下也可以将Broker看作一台Kafka服务器，前提是这台服务器上有且仅 有一个Kafka实例。一个或多个Broker组成了一个Kafka集群。在集群中每一台Broker 都会有一个对应的唯一 ID标号，这个ID会作为每台broker的标识存储在zookeeper中。

1. Producer （生产者）

生产者作为发送消息的一方，首先与Kafka建立连接，然后将消息生产出来并发送到指 定topic中。Kafka在接受到消息后根据内置的负载均衡策略将接受到的message写入到对应 的topic中。

1. Consumer Group （消费者组）

Consumer Group是Kafka提供的可扩展且具有容错性的消费者机制。一个消费组内必然 可以有多个消费者或消费者实例（consumer instance）,它们共享一个group ID。组内的所有消 费者协调在一起来消费订阅主题的所有分区。当然，每个分区只能由同一个消费组内的一个 consumer来消费。

**2.3**文本提取工具**Jsoup**及网络框架**HttpCIient**

Jsoup是一款基于Java的HTML解析器，该工具可以通过直接解析URL或者HTML的 文本内容。Jso叩提供了一套完整的API,可以通过DOM和CSS以及Xpath等语法格式以类 似于jQuery的操作方法来取出和操作数据。通过解析HTML得到一个Document对象四，在 生成Doucment后，根据Dom和css或类似jquery的selector语法获取Element,然后再从 Elements中获取节点属性、文本、html。在Jso叩中文档中的所有内容都可以看做是一个Node （节点），而Element （元素）则是一个更小的概念，元素继承于节点。而Document （文档） 则是最底层的元素，指整个HTML文档的源码内容。Jsoup通过其自身所内置的API,将文 本内容通过状态机序列化成一个DOM解析树，进而通过CSS选择器或者DOM操作从解析 树中提取出相关的文本数据信息，除此之外，Jsoup框架还支持对HTML中的属性，文本， 元素进行操作，通过操作节点改变HTML的页面结构以及数据组织构成。Jsoup基于MIT协 议发布，因此可以用于商业项目。

Jsoup可以从包括字符串，URL以及本地文件来加载HTML文档，生成document对象。 通过connect方法将指定URL参数传入，生成document对象实例。Jsoup同时支持修改文档 数据在链接文档时有时需要对文档中的某些元素进行修改，例如需要对文档中的图片地址， HTML文本信息进行修改，通过内置的select方法操作CSS标签，获得当前标签的元素值， 最后使用attribute方法对指定标签元素进行修改，除了无法修改标签名外，元素的属性和文 本内容都可以进行修改。Jsoup同时支持HTML文档清理，在页面中有时会遇到第三方故意 在HTML文本中添加脚本，这些脚本可能会破坏整个页面的的行为，甚至导致XSS跨站点 攻击，Jsoup通过内置的clean方法对指定HTML文档进行过滤，可以设置只允许包含文本信 息，只允许包含指定标签，只允许包含图片链接。Jsoup通过其良好的可拓展性API设计支持 对HTML结构的快速结构性调整和设置。

HttpClient是由Apache开源的支持HTTP协议的客户端编程工具包，它基于Java语言实 现，支持包括HTTP 1.0和HTTP 1.1协议，HttpClient以可拓展的面向对象的结构实现了 HTTP 的全部方法，通过HTTP代理方式建立透明连接，HttpClient支持可插拔的安全套接字工厂， 从而可以方便接入第三方解决方案支持连接管理，可以自定义cookie策略的插件化机制。通 过request输出流发送数据，response输出流接收数据四。

HttpClient框架支持使用多线程的应用。支持设置最大连接数，并能够针对单个主机设置 其最大连接数，同时基于socket接口能够发现并关闭过期连接，在HTTP 1.0和HTTP1.1协议 中支持使用Keep-Alive保持持久连接。作为一个网络框架，HttpClient能够满足在网络编程的 绝大部分需求场景，功能强大。HttpClient框架通过将返回数据包装成request输出流来避免 流中数据直接缓冲到socket服务器,response输入流的数据则可以有效的从socket服务器直 接读取相应内容，同时HttpClient框架支持直接获取服务器返回的response header和response code,在新版本的HttpClient框架中实验性的支持HTTP 1.1 response cache□

基于HttpClient框架强大的API,发送请求，接受响应很简单，通过创建HttpClient对象 实例，指定请求URL,如果请求是get方法，则创建HttpGet对象发送get请求，如果请求是 post方法，则创建HttpPost对象发送post请求，通过内置的setParams方法设置请求参数，对

于post方法，支持通过setEntity方法设置请求参数，通过execute方法执行发送请求，通过 将返回对象包装成HttpResponse对象获取返回数据，通过getHeaders获取返回的HTTP code 参数以及返回体body数据。

在请求数量比较多的情况下，两个主机间建立连接的过程涉及到多个数据包的交换 HttpClient框架通过PoolingHttpClientConnectionManager类实现连接池管理，通过维护每个路 由数上的连接数实现请求链接复用，同时为了防止请求链接被长时间占用，连接池支持对池 中的每个链接设置请求超时时间和响应超时时间。

**2.4**存储数据库**MongoDB**及**Redis**

MongoDB是一个由C++语言编写，基于分布式文件存储的开源数据库，它的主要特点是 采用面向文档的存储〔24〕。一个MongoDB中可以建立多个数据库，每一个MongoDB的单个 实例可以容纳多个独立的数据库。MongoDB中的collections （集合）相当于关系型数据库中 的表，集合就是MongoDB文档组，集合没有固定的数据结构，即用户在使用时可以插入不 同格式和类型的数据。集合中存放的则是文档组，document （文档）式一组键值对的组合， MongoDB的文档不需要设置相同的字段，同时在存储数据时，相同的字段不需要相同的数据 类型。MongoDB中的Field （字段）则相当于关系型数据库中的columno在实际的应用场景 中MongoDB可以很方便的存储一些文本格式的数据，基于文档模式存储的灵活数据存储原 理，是MongoDB的一大优势，对于数据模型多样或字段数量多变的业务场景，相比MySQL 等传统关系型数据库，在新增或删除字段时避免每次均要通过DDL语句去进行表结构的修改。 由于MongoDB的Value字段对于MongoDB是非透明的，因此可以采用类似于关系型数据库 的模式对其索引建立，在进行全文检索时的查询效率上更具优势。

MongoDB的集群一般釆用副本机制，所谓的MongoDB数据副本是将数同步复制到多个 服务器上，通过在不同的服务器上存储相同的数据，副本机制能够保证一定的容错性，即使 在某台服务器上的MongoDB挂掉，数据服务依然可用，同时副本机制的存在可以提升数据 的读性能，当大量用户同时查询页面解析结果时，可以通过从不同的副本读取数据，提高分 布式系统的可用性。一个MongoDB副本集包括一个主节点和多个从节点，主节点接收所有 的写操作。主节点将所有的数据变动记录在oplog H志中，从节点复制主节点的oplog日志， 然后根据日志内容从节点重放数据集的变动，通过这样的方式和主节点达到数据的一致性， 如果主节点挂掉了，一个合格的从节点将发起一次选举将自己选为新的主节点。当一个主节 点和副本集中其他节点有超过10秒的连接中断时，第一个发起选举且得到副本集中超过一半 选票的合格从节点将会成为新的主节点。可以在副本集中额外维护一个单独的MongoDB实 例作为仲裁者，仲裁者不维护具体的数据集，主要用于维护主节点和从节点之间的心跳以及 响应其他副本集成员节点的请求，因为作为仲裁者节点的MongoDB实例不维护数据集，同 维护了完整数据集的MongoDB实例相比，消耗的资源更少，并且一旦出现副本集节点数量 是偶数的情况时，因为仲裁节点的存在，可以加入选举节点中，使得在主节点选举的过程中 具备超过一半选票的能力。通过在主节点实例釆用rs.add(HOST\_NAME:PORT)的方式，给主 节点添加副本集的实例节点。

Redis是一个基于C语言编写，遵守BSD协议，支持网络，支持持久化的内存数据库。 它采用了 key-value的存储结构在Redis中key的常见数据结构类型有string, list, hash, set, sort set等基础类型。Redis也像关系型数据库一样支持master-salve模式的主从架构部署 〔26-27］。Redis作为一款基于内存的非关系型数据库，拥有极高的性能，可以达到每秒十万次级 别的读写操作，同时因为Redis单线程的特点，所有的操作都是原子性的，从而避免单实例 下的并发问题。Redis提供了两种持久化方式：RDB和AOF,方便在Redis实例崩溃导致的 内存数据丢失后，从RDB或者AOF文件恢复数据，从而尽可能的保证数据不丢失。

Redis作为一种高性能的内存数据库也支持集群部署，常见的集群部署方式有两种：Redis cluster模式和哨兵模式，Redis cluster模式节点启动以后是相互独立的，节点之间彼此通过 cluster meet IP port方式完成节点握手，从而实现将独立的多个节点组成一个完整的网络。 Rediscluster模式通过哈希槽的方式进行数据划分，集群由16384个槽，槽是数据管理和迁移 的基本单位，通过一致性hash算法实现槽的划分以及数据同步。通过虚拟槽分区解耦了数据 和节点的关系，简化了节点扩容和收缩的复杂度，通过支持槽，节点，键的映射查询，用于 数据路由，在线集群伸缩的场景。节点通过配置cluster-config file指定集群配置文件的位置, 每个节点在运行时会维护一份集群配置文件，每当集群信息发生变化时(例如增减节点)， 集群内的所有节点会将最新信息更新到该配置文件中，这样当集群中任意节点重启时会重新 读取该配置文件，获取集群信息。当节点发生故障时，通过定时任务发送ping信息监测节点 状态，集群只负责主节点的故障转移，从节点故障时只会被下线，不会进行故障转移，从节 点通过资格检查确定与主节点的最后断线时间，在持有槽的主节点中处理故障选举信息，通 过广播发起选举，当岀现某个从节点获得超过一半选票的时候代表选举完成。在每一个配置 纪元里，每个具有投票权的主节点只能投一次票，确保了获得票数大于等于N/2+1的从节点 只有一个。如果在一个配置纪元里，没有从节点能够拿到足够多的选票，那么集群进入一个 新的配置纪元，并再次进入选举，直到选出新的主节点为止。

哨兵模式主要由两部分组成，哨兵节点和数据节点，哨兵系统由一个或多个Redis实例 节点组成，哨兵节点作为特殊的Redis节点，节点本身并不存储数据，通过在主从复制的基 础上额外设置几个哨兵节点，由哨兵节点对主节点和数据节点进行监控，将主节点发生故障 时进行故障转移操作，主要步骤如下：

1. 在从节点中选择新的主节点：选择的原则是，首先过滤掉不健康（主观下线，断线），5 秒内没有回复过sentinel节点ping响应，与主节点失联超过down-after-milliseconds 10秒的从 节点；然后选择优先级最高的从节点（由slave-priority指定）；如果优先级无法区分，则选择复 制偏移量最大的从节点；如果仍无法区分，则选择runid最小的从节点。
2. 更新主从状态:通过slaveof no one命令，让第一步中选出来的从节点成为新的主节点; sentinel领导者节点会向剩余的从节点发送salveof命令，让他们成为新的主节点的从节点。
3. 将已经下线的主节点（即6379）设置为新的主节点的从节点，当6379重新上线后，它 会成为新的主节点的从节点。

**2.5**本章小结

本章主要针对在基于微服务架构的分布式爬虫系统中将要用到的相关理论和技术点进行 了简单的介绍，包括Spring Cloud微服务框架的常见组件，消息队列Kafka的核心概念，页 面抓取工具HttpClient的使用，页面解析工具jsoup,最后对本系统将要使用的存储数据库 MongoDB和Redis进行了介绍。

第三章分布式爬虫系统总体设计

3.1系统需求分析与设计目标

基于微服务架构的分布式爬虫系统是基于当前大型软件开发中釆用较多的微服务架构进 行开发，在架构层面对系统进行了拆分，将每一个核心功能都解耦出来都作为一个微服务， 每个微服务注册到统一的注册中心，通过注册中心实现统一的监控和管理。同时提供了 ui界 面以方便用户定义具体的爬虫任务。在整个抓取页面和解析页面的过程中无需用户参与，同 时考虑到会有大量用户提交爬虫任务，因此系统需要具备较强的吞吐量和可靠性，能够便于 在分布式环境下部署，为了达到以上效果，系统需要以下的功能实现：

1. 完善的可视化页面

根据本系统的使用场景，需要有一套直观的可视化界面去方便用户定义爬虫任务的具体 参数。同时还要用户还能直观的看到爬虫任务当前的任务状态，以及数据解析模块根据用户 自定义提取规则所解析后的结果数据都能直观的展现出来〔2句。

1. 爬虫任务在分布式场景下的高效执行

分布式爬虫系统是通过将大量的爬虫任务分散到不同的机器上共同完成数据的抓取，然 后在现实环境下不同机器的性能可能有着不同的差距，因此就需要一个能根据不同机器对爬 虫任务的响应时间来做负载均衡的任务分发机制，从而让每台机器的爬虫任务量可以在合理 的范围内实现高效的爬取【29】。同时在每台机器上的页面下载模块釆取多线程的下载机制去加 快页面抓取效率，让每台机器能根据自身硬件条件发挥最佳的性能。

1. 高吞吐量

在实际场景下的分布式爬虫系统，其爬虫任务的提交速度会远快于具体的任务爬取，当 提交的任务数远超出系统的负载时可能就会导致系统的可用内存或者CPU资源耗尽，从而导 致整个系统的不可用［3。-3叫因此我们需要一个高吞吐量的存储中间件去给系统做流量削峰， 通过该中间件去作为任务暂时的存储，下游任务模块则直接从该中间件中去获取任务信息， 从而使得下游的任务模块的消费速度可以保持在系统的合理承载范围，在提高爬取效率的同 时又能及时响应用户新的请求。

1. 高效的数据存储

在分布式爬虫系统中，会存在大量不同类型的数据需要存储，针对不同的数据场景选择 不同的存储数据库也是提升系统整体性能的重要因素。针对抓取下来的页面信息采用文档型

数据库，针对爬虫任务的状态和任务信息采用关系型数据库，针对指定爬虫任务的提取规则 以及数据池的存储釆用内存数据库存储，从而实现数据的高效存储，尽量避免频繁大量的数 据库I/O操作成为系统的瓶颈〔32-33］。

3.2系统总体设计架构及运行机制

3.2.1分布式爬虫系统各微服务模块划分

基于以上的设计需求与目标，本节对基于微服务架构的分布式爬虫系统从整体架构层面 进行更为详细的划分，如图3. 1所示，系统从整体上分为可视化UI界面（Crawler Dashboard）, 请求分发服务，请求预处理服务，页面抓取服务，页面解析服务等微服务模块。每一个微服 务互相之间都独立部署，服务与服务之间彼此通过HTTP或者RPC （远程过程调用）的方式 进行通信，每个微服务都会实现分布式部署，即多实例部署，每一个实例都会注册到统一的 注册中心，在注册中心可以观察到所有微服务的注册信息（比如端口号，IP,服务是否下线 等信息）。为了爬虫系统的高吞吐性，微服务之间的中间件存储本系统釆用了 Kafka作为消 息中间件，依赖Kafka的高吞吐量，消息顺序保障性等特性，将Kafka作为存储任务信息， 页面信息等数据的过渡存储中间件，以此来应对爬虫系统中流量过大的业务场景，从而保证 整个系统的高可用。基于微服务架构的分布式爬虫系统的各个微服务模块划分示意图如图3. 1 所示：

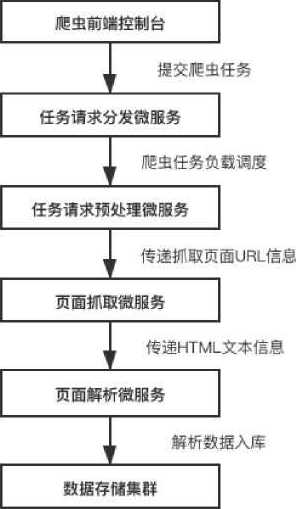


图3. 1基于微服务架构的分布式爬虫系统各微服务模块划分图

3.2.2系统整体架构图及运行流程

基于微服务架构的分布式爬虫系统的各微服务之间是互相交互协作的，各种不同的数据 存储介质分别与相关部分的微服务交互。图3. 2展示了基于微服务架构的分布式爬虫系统的 整体运行流程图。

基于微服务架构的分布式爬虫系统的运行机制是如下顺序，首先用户在Crawler Dashboard的可视化界面中通过新建Crawler Job创建出新的爬虫任务，点击提交后会向服务 端发起爬虫任务提交请求。

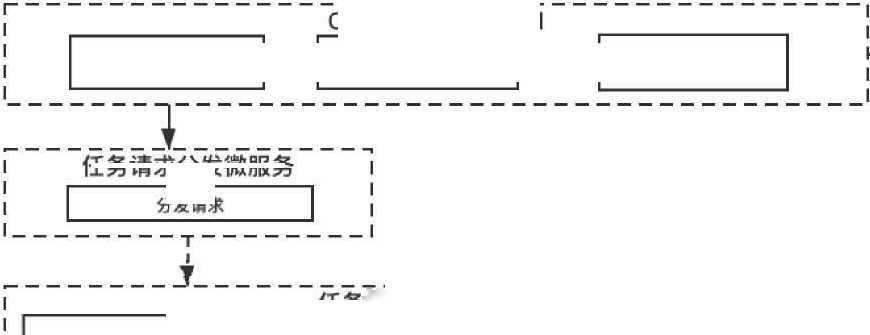
任务请求分发微服务接收到上一步中提及的爬虫任务请求后，会根据实际情况，从基于 轮询的负载均衡策略，基于响应时间作为选取权重的负载均衡策略，基于服务可用性进行进 行过滤的负载均衡策略这三种负载均衡策略中选择最合适当下场景的负载策略，将接收到的 爬虫任务请求分发至部署了任务请求预处理微服务的不同机器上。

在任务请求预处理微服务中会对由任务请发分发微服务下发的爬虫任务信息进行一系列 的包装，首先对每一个任务生成一个分布式全局唯一 ID,将这个ID同该爬取任务的pattern 提取规则组合投递至Redis集群，接着在MySQL中会生成一条新的crawler job任务信息，并 初始化任务状态status值，将提交的爬虫任务信息中的URL, jobld以及上一步生成的分布式 全局唯一ID包装成C rawlerJobMessage对象并序列化该对象提交至Kafka集群中的指定topic。

在页面抓取微服务中，该服务会监听对应topic的信息，将监听到的二进制数据反序列化 成CrawlerJobMessage对象，接着从cookie池和user-agent池中取出数据，将这些数据跟解析 到的CrawlerJobMessage对象属性组合成完整的请求体，依赖线程池开启多线程爬取，使用 HttpClient作为抓取工具，根据抓取结果返回的response状态码异步更新数据库中爬虫任务状 态值。将抓取到的页面信息包装成一个DownloadPagelnfoMessage对象并将其序列化后投递 至Kafka集群中存储抓取到的页面信息的指定topico

在页面解析微服务中，该服务会监听存储页面信息的topic,将监听到的数据反序列化成 DownloadPagelnfbMessag 对象，根据从 DownloadPagelnfoMessage 对象获取的全局唯一的 patternld从Redis中获取该页面对应的提取规则，从而使用jsoup完成解析，将解析后的数据 以键值对的形式包装成json字符串存储至文档型数据库MongoDB中。

基于微服务架构的分布式爬虫系统的整体架构图如图3. 2所示：



Crawler Dashboard

| 创建Crawler Job页面 |  | 查看Job运行状态页面 |  | 解析结果査看页面 |
| --- | --- | --- | --- | --- |

任务请求分发微服务

分发请求

| 校验提交Job信息 |  | 生成Pattern全局唯 -ID |  | 封装Job信息 |  | 封装Pattern解析规则 |
| --- | --- | --- | --- | --- | --- | --- |

任务请求预处理微服务

Job信息投递至Kafka中指定Topic

Pattern映射规则投递至RecHs

Kafka集群

(Kafka ( ( Kafka ( ( Kafka ( ； • (Redi$ () (Redls ()(Redls ()

Redis集群

监听topic信息

页面抓取微服务

获取topic中的  
Job信息

幵启多线程  
采集

~~「漩蠶阿~~ 了( cookie池()

采用HttpClient抓取指  
定页面

根据抓取結果异步更  
新DB中Job Status

夕-@er-agent池()

抓取到的PagelnfoJS递至Kafka中指定  
Tooic

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Kafka集学  (Kafka ( ( Kafka ( 1  r | Kafka | r |  | 根据pattern ID获取对应job的 pattern解析规则 |

监听topic信息

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 页面解析微服务 1 | | | | | | |
|  | 获取 topic 中的 Pagelnfo |  | 获 BXTopic 中该 Pagelnfo 的pattern ID |  | 根据 pattern 提取 pagelnfo 中指定信息 | 1  1  1 |
|  |  |  |  |  |  | \_1 |

|  |  |  |
| --- | --- | --- |
| 抓取数据存储 | | |
| ^MongoDsf | ^MongoDB |  |

3.3爬虫系统的各子部分功能介绍

基于微服务架构的分布式爬虫系统中可视化界面将作为与用户的唯一交互场景，用户在 可视化界面提交创建爬虫任务，相应爬虫任务请求将通过ajax请求提交到任务请求分发微服 务模块；在任务请求分发微服务中对提交的爬虫任务请求通过任务请求分发微服务中内置的 负载均衡策略进行请求分发，下发到任务请求预处理微服务；在任务请求预处理微服务中对 接收到的爬虫任务请求生成唯一 ID,同时对重复抓取的URL判重处理，最后将爬虫任务投 递到消息中间件中；页面抓取微服务通过消费Kafka中指定topic的爬虫任务进行页面抓取, 将抓取到的页面文本信息投递至消息中间件；页面解析微服务基于提取规则消费Kafka中数 据并进行页面解析，最终将解析后的数据入库处理。

以下各小节将对基于微服务架构的分布式系统的各个功能模块的运行流程以及模块间交 互顺序进行阐述。

3.3.1任务创建与查询可视化界面

任务创建与查询可视化界面主要用于方便用户进行可视化创建新的爬虫任务以及查询己 创建的爬虫任务的结果与运行状态。

具体的处理流程如下：

（1） 用户进入可视化界面，通过点击新建任务按钮，创建新的爬虫任务，在弹出的模态 框中输入URL路径，任务名称。

（2） 点击添加自定义数据提取规则按钮，在模态框中输入数据提取规则，最后点击提交 按钮。

（3） 创建后的爬虫任务请求会通过HTTP请求提交至任务请求分发微服务中。

3.3.2任务请求分发微服务

任务请求分发微服务主要用于接受任务可视化模块所下发的请求，这些请求的类型分为 爬虫任务创建请求，爬虫任务结果查询请求，接着根据目前的服务注册情况选择合适的负载 均衡策略将这些请求下发到任务请求预处理微服务。

任务请求分发微服务会对上述请求进行参数校验，过滤，并对非法请求进行拦截。

在本系统中采用了基于客户端的负载均衡策略，在任务请求分发微服务中内置了三个客 户端负载均衡算法，基于线性轮询的负载均衡策略，基于响应时间权重的负载均衡策略，基 于服务可用性进行过滤的负载均衡策略。这些负载均衡算法会在第四章作详细介绍。微服务 会根据缓存在客户端的服务注册列表情况以及服务可用性情况去选择当前任务场景下最佳的 负载均衡策略。如果服务属于新启动，则会选取基于线性轮训的负载策略；如果当前可用服 务数量小于预先定义的阈值，则选取基于最小请求数的负载均衡策略；如果服务既不属于新启 动且可用服务数也大于阈值，则选用基于响应时间权重的负载策略。

通过参数校验和负载均衡处理的爬虫任务请求将通过微服务模块间的远程调用传递至任 务请求预处理微服务中进行下一步处理。

任务请求分发微服务运行流程如图3. 3所示

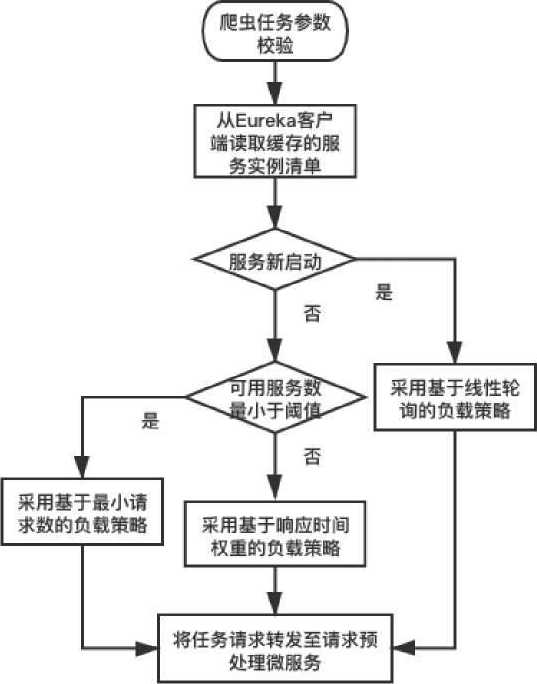


图3. 3任务请求分发微服务运行流程图

3.3.3任务请求预处理微服务

任务请求预处理微服务主要用于处理由任务请求分发微服务依据负载均衡策略下发的爬 虫任务请求，并针对新建爬虫任务请求，对该任务的数据提取规则生成一个全局唯一的分布 式ID,并将该ID与对应数据提取规则写入Redis,同时釆用基于布隆过滤器和Redis的双重 校验时段去验证该URL是否属于重复爬取，在以上步骤都完成后，本微服务将作为生产端将 爬虫任务信息投递至Kafka集群以供作为消费端的页面抓取微服务消费数据。

如果调用的是提交新爬虫任务接口，处理流程如下：

步骤一：如果请求类型是提交新的爬虫任务，首先去校验该job信息是否符合要求，当 校验完job信息后，任务请求预处理微服务会针对该job的pattern数据提取规则通过分布式 1D算法生成一个全局唯一的分布式IDo

步骤二：将步骤一种生成的全局唯一的分布式1D同对应job对pattern提取规则组合成 一个key-value键值对，以分布式ID值作为Redis的key, pattern作为Redis的value写入Redis 数据库。

步骤三：将本次新建爬虫任务的请求数据组合成一个CrawlerJob对象，初始化爬虫任务 状态，并MySQL中插入一条新纪录，插入成功后获取返回的数据库记录主键ID-

步骤四：对此次请求中的待爬取URL采用基于多重Hash的布隆过滤器和Redis数据库 进行双重判重，首先釆用多重Hash的布隆过滤器去判断该URL是否重复爬取，如果不在布 隆过滤器范围内，则继续釆用基于Redis的判重，因为每一次的爬虫任务都会将URL存入到 Redis中，通过Redis的set数据结构判断是否重复，通过这样的双重校验机制来确定该URL 是否属于重复爬取。

如果该URL属于重复爬取，则任务预处理微服务会跳过页面抓取微服务，转而通过声明 式客户端直接调用页面解析微服务，将URL路径作为查询参数，由页面解析微服务根据步骤 一中生成的全局唯一分布式ID从Redis中获取对应的pattern解析规则，进而执行页面解析处 理。

如果该URL未被爬取过，则将步骤一中生成的分布式全局唯一 ID,步骤二中返回的主 键 ID,待爬取 URL, jobName 信息组合一个成 CrawlerJobMessage 对象，CrawlerJobMessage 对象包含的相关属性将其含义：｛URL：待爬取路径，pattemld：步骤一中生成的全局唯一 ID, jobld：步骤三中返回的数据库主键ID, jobName：本次处理的爬虫任务名称｝,将生成的 CrawlerJobMessage对象序列化成json格式同步写入Kafka中指定topic中，该消息将被作为 消费端的页面抓取微服务消费。

页面抓取微服务则会通过读取Kafl<a中由任务请求预处理微服务写入的爬虫任务数据进 行抓取。

任务请求预处理微服务运行流程如图3.4所示：

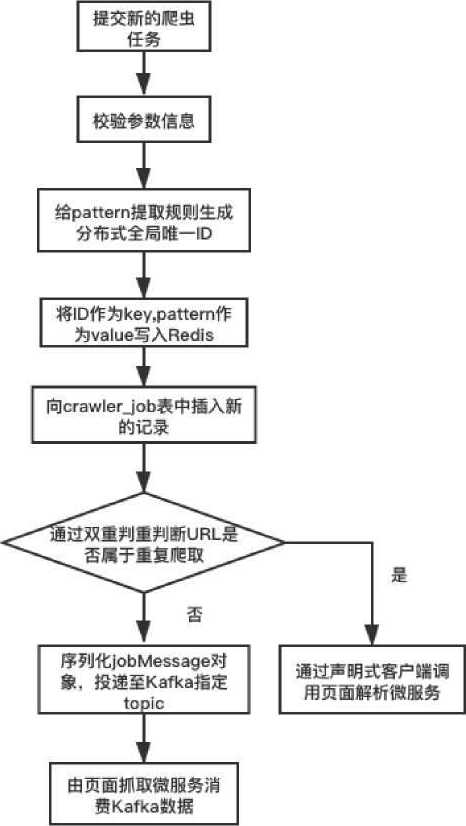


图3. 4新建爬虫任务流程图

如果调用的是查询爬虫任务结果接口，处理流程如下：

步骤一：首先去校验请求字段是否符合要求，当校验完字段信息后，根据提交的job\_name 字段去crawler\_job表查询，将返回结果映射成一个CrawlerJob对象。

步骤二：根据步骤一的查询结果，从查询结果中获取mongo\_id字段。

步骤三：根据步骤二中获取到的mongo\_id字段值作为查询条件从MongoDB中获取本次 爬虫任务的解析结果。并返回前端页面展示。

任务请求预处理微服务作为页面抓取微服务的上层服务，负责将经过全局一致性算法处 理，且已经序列化后的爬虫任务的具体信息投递至Kafka集群。

3.3.4页面抓取微服务

页面抓取微服务主要负责监听由任务预处理微服务投递到Kafka指定topic中的爬虫任务 信息，并对监听到的信息进行反序列化，在反序列化后的信息中提取出URL路径等信息。页 面抓取微服务中内置了一个线程池，通过线程池开启多线程爬取，从而大大提高了多核CPU 的效率，将每个线程爬取下来的页面HTML文本投递至Kafka集群的指定topic,以供页面解 析微服务消费。

具体的处理流程如下：

步骤一：监听Kafka指定topic中由任务预处理微服务生产的爬虫任务信息，将监听到的 数据反序列化成CrawlerJobMessage对象，获取该对象中的URL属性值，pattemld属性值， jobld属性值，jobName属性值。

步骤二：使用HttpClient框架构造一•个HttpGet对象，接下来从Redis中存放的user-agent 池和cookie池中随机取出一个user-agent头和cookie值，获取配置类中其余属性信息，将这 些信息同步骤一中获取的URL属性值作为HttpHeader值传入HttpGet,初始化一个 CloseableHttpClient对象，将封装过后的HttpGet传入，接着开启线程池的多线程抓取，在每 个线程中执行execute方法，获取返回的CloseableHttpResponse对象。

步骤三：通过判断步骤二中返回的CloseableHttpResponse对象的statusCode字段值是否 等于200来判断此次抓取是否成功。

步骤四：如果返回的状态码等于200,则将返回response中的HTML文本同步骤一中获 取的URL属性值，pattemld属性值,jobld属性值，jobName属性值组合成一个 DownloadPagelnfoMessage 对象，DownloadPagelnfoMessage 对象包含的相关属性将其含义： {pagelnfo：抓取的HTML文本数据，URL：已爬取路径，pattemld：全局唯一 ID, jobld：数 据库主键ID, jobName：本次处理的爬虫任务名称},将生成的DownloadPagelnfoMessage对 象序列化成json格式同步写入Kafka中指定topic中，该消息将被作为消费端的页面解析微服 务消费。若返回的状态码不是200,则开启重试策略，以2秒为时间间隔，开启指数级重试。

步骤五：首先根据步骤一中的jobld从crawlerjob表中获取当前对应的爬虫任务信息。 如果步骤四中的返回状态码为200,则将任务状态由初始化状态异更新为下载成功状态，如 果状态码不是200,即代表无效响应，则将任务状态由初始化状态更新为下载失败状态。所有 的状态更新操作都交由额外的线程池异步执行，并不会影响页面的爬取。

页面抓取微服务运行流程如图3. 5所示:

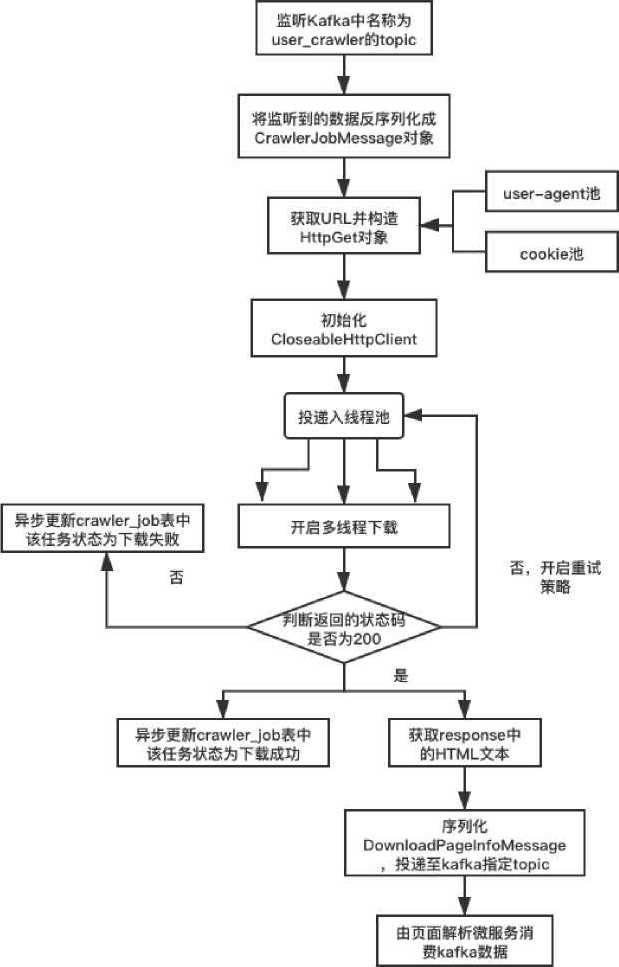


图3. 5页面抓取微服务运行流程图

3.3.5页面解析微服务

页面解析微服务主要负责监听由页面抓取微服务投递到Kafka指定topic中的抓取页面信 息，并对监听到的信息反序列化成DownloadPagelnfoMessage对象，根据其中的patternld属 性值作为可以从Redis中取出该job对应的pattern提取规则，最后使用jsoup对 DownloadPagelnfoMessage对象中的HTML文本信息进行解析提取，并将解析后的信息存入 MongoDB数据库中。

具体的处理流程如下:

步骤一：监听Kafka指定topic中由页面抓取微服务生产的页面文本信息，将监听到的数 据反序列化成DownloadPagelnfoMessage对象，获取该对象中的pattern Id属性值,jobld属性值， pagelnfo属性值，URL路径值。

步骤二：将步骤一中获取的HTML纯文本，URL路径值组合成一个FullPagelnfo对象, 将该值存入MongoDB中的full\_page\_info表。用于当岀现URL重复爬取时可以跳过页面抓 取步骤，直接从MongoDB中取出HTML信息解析文本。

步骤三：根据步骤一中获取的patternld从Redis中获取该pattern Id对应的页面信息提取 规则。

步骤四：使用jsoup根据步骤三中获取的页面信息提取规则进行解析，将解析完成后的 数据以key:value的形式组合成一个json串，存入MongoDB中的parse\_page\_info表。

步骤五：根据步骤一中获取的jobld,从crawlerjob表中获取当前对应的爬虫任务信息, 如果步骤四解析成功，则将任务状态字段更新为解析成功状态。如果步骤四解析失败，则将 任务状态字段更新为解析失败状态，将更新后的数据回写至crawlerJob表。所有的更新操作 均釆用单独的线程池进行异步更新，尽可能的不去影响页面解析线程的运行。

页面解析微服务运行流程如图3. 6所示：

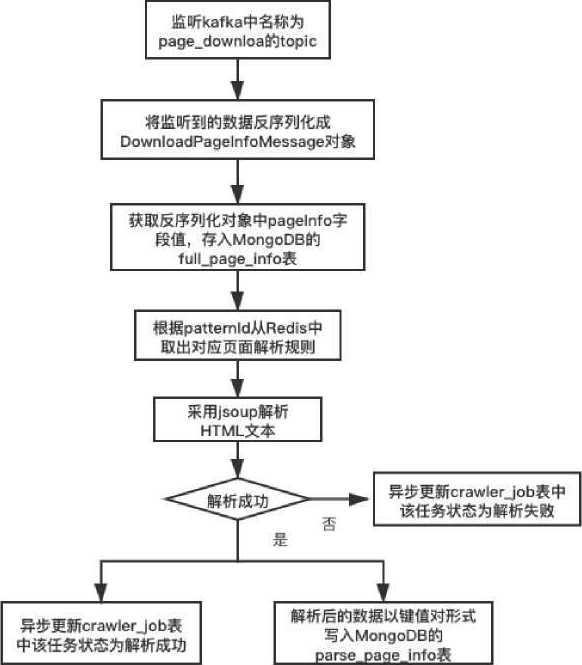


图3. 6页面解析微服务运行流程图

23

3.4数据库设计

在基于微服务架构的分布式爬虫系统中，数据存储的场景多种多样，使用了 MySQL关 系型数据库，Redis内存数据库，MongoDB文档型数据库这三种数据库，接下来对这三种数 据库的具体数据表进行阐述。

1 .MySQL数据库，在MySQL数据库中主要包括表crawlerjob表，主要用于记录爬虫任 务的相关信息，其属性如表3.1所示。

表3.1爬虫任务表

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名 | 类型 | 长度 | 说明 |
| id | int | 11 | 主键id |
| jobname | varchar | 200 | 爬虫任务名称 |
| pattrnid | varchar | 50 | 提取规则全局唯一 id |
| url | varchar | 200 | 待爬取URL路径 |
| mongoid | varchar | 50 | MongoDB中的主键id |
| status | int | 2 | 爬虫任务运行状态 |
| create\_time | timestamp | 50 | 任务创建时间 |
| lastModify\_time | timestamp | 50 | 最后一次修改时间 |

2.Redis数据库，因为Redis的基于内存的特性，使其在高并发场景下相比于基于磁盘的 关系型数据库具有更大的性能优势，因此在本系统中大量釆用了 Redis作为缓存数据库使用， 主要存储以下几种信息：

1. 存储单个爬虫任务的数据提取规则，存储的key为表crawlerjob中对应爬虫任务 生成的提取规则全局唯一 id, key-value存储键值对如表3.2所示：

表3.2数据提取规则key-value

|  |  |  |
| --- | --- | --- |
| Key含义 | Value数据类型 | Value数据含义 |
| job提取规则的全局唯一 1D | String | 数据提取规则 |

1. 存储所有己爬取的URL路径，key-value存储键值对如表3.3所示:

表3.3已爬取URL路径集合

|  |  |  |
| --- | --- | --- |
| Key含义 | Value数据类型 | Value数据含义 |
| url\_{ID} | Set | Set中存放所有URL |

1. 存储白名单代理IP表，key-value存储键值对如表3.4所示：

表3.4代理IP白名单

|  |  |  |
| --- | --- | --- |
| Key含义 | Value数据类型 | Value数据含义 |
| proxy\_white\_ {ID} | list | 存放代理IP白名单 |

1. 存储黑名单代理IP表，key-value存储键值对如表3.5所示:

表3.5代理1P黑名单

|  |  |  |
| --- | --- | --- |
| Key含义 | Value数据类型 | Value数据含义 |
| proxy\_black\_ {ID} | list | 存放代理IP黑名单 |

1. 消费端去重表，key-value存储键值对如表3.6所示:

表3.6消费端去重集合

|  |  |  |
| --- | --- | --- |
| Key含义 | Value数据类型 | Value数据含义 |
| consumer\_offset | set | 存放己消费的消息1D |

3.MongoDB数据库，在爬虫系统中不同页面可能存在数量不同的解析规则，这样的数据 在关系型数据库中一旦发生表字段的变更，就势必要通过DDL来进行更改,而釆用MongoDB 则可以有效避免这个情况。本系统中MongoDB主要包含两张表：

(1)完整页面信息full\_page\_info Document,表中url字段对应表crawler Job中对应爬 虫任务的爬取路径，根据url字段与表crawlerjob指定爬虫任务形成一对一关系，具体字段 信息如表3.7所示：

表 3.7 full page info Document

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 字段含义 |
| id | string | 主键 |
| url | string | 爬取路径 |
| info | string | 抓取的HTML文本 |
| create\_time | date | 创建时间 |
| last\_modify\_time | date | 最后修改时间 |

1. 页面解析信息parsepageinfo Document,表中url字段对应表crawler Job中对应 爬虫任务的爬取路径Job name字段对应表crawlerjob中job name字段，根据url和job name 字段与表crawlerjob指定爬虫任务形成一对一关系，具体字段信息如表3.8所示：

表 3.8 parse page info Document

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 字段含义 |
| id | string | 主键 |
| url | string | 爬取路径 |
| parseinfb | string | 解析后的文本数据 |
| jobname | string | 爬取任务名称 |
| create\_time | date | 创建时间 |
| lastmodifytime | date | 最后修改时间 |

3.5本章小结

本章首先从基于微服务架构的分布式爬虫整体设计需求与设计目标出发，详细分析了基 于微服务架构的分布式爬虫所应具备的功能及技术特点，并详细画出了系统的整体运行流程 图和整体架构图，详细描述了系统的整体运行机制；接下来详细画出了各个功能模块的详细 流程图，并对每个模块的功能以及运行流程作出了描述；最后对系统中使用到的多种数据库 进行介绍，并详细阐述了各个数据库在本系统中的具体定位，以及详细的表字段设计。

第四章分布式爬虫系统核心功能实现

基于微服务架构的分布式爬虫系统提供了基于Angular框架构建的可视化界面，用户基 于可视化界面创建爬虫任务，通过HTTP请求提交至后台的任务请求分发微服务，由任务请 求分发微服务基于负载均衡算法下发至请求预处理微服务，在抓取服务中下载指定URL的文 本信息，解析服务根据自定义提取规则解析页面数据，存储至MongoDB集群。

可视化界面主要用于用户创建爬虫任务，用户可以在可视化界面上自定义待爬取的URL 路径，填写对应的job名称，并且对指定的页面提供相应的页面解析数据规则，通过ajax请 求，将爬虫任务信息传递至任务请求分发微服务模块中。所有爬虫任务的提交通过任务编辑 界面的选项卡中创建，新建爬虫任务界面如图4.1所示，在页面中填入此次爬虫任务名称， 待爬取的URL路径，点击按钮在弹出的模态框中添加自定义的基于CSS选择器的数据提取 规则，通过点击新增按钮可以实现不受限制的表格行添加，添加自定义数据提取规则如图4. 2 所示。

只 Crawler Dashboard

88新建

创建新爬虫任务

\*任务名称⑦:

\*待爬取URL路径:

★数据提取规则:

88查询

查询任务信息

查询页面信息

图4. 1新建爬虫任务界面

添加CSS选择器规则

新増

操作

字段名称

提取规则

Delete

确定

图4. 2添加自定义数据提取规则

4.1任务请求分发微服务

任务请求分发微服务负责将用户创建的爬虫任务请求进行参数校验与处理，考虑到机器 的承载能力有限，在分布式场景下釆用限流算法对请求提交的速率进行限制，并将通过校验 的爬虫任务请求通过定义的多种客户端负载均衡算法下发至任务请求预处理微服务，并釆用 了服务熔断机制应对可能出现的服务故障导致请求超时或者请求失败的场景，从而提升系统 的可用性。

4.1.1客户端负载均衡算法处理

负载均衡算法是我们处理高并发，缓解网络压力从而确保服务端可用性的重要手段。负 载均衡分为服务端负载均衡和客户端负载均衡。常见的服务端负载均衡手段有通过Nginx, 通过在Nginx端维护一份服务清单通过轮训实现负载均衡；但服务端负载均衡在微服务架构 的分布式系统中有着比较明显的缺点：服务清单是写死在Nginx配置中，一旦出现服务不可 用的情况就需要手动修改服务列表，重启Nginx服务，不能对发生故障的服务或机器快速响 应，大大增加人力处理的负担。而客户端负载均衡的最大特点则是

基于客户端的负载均衡算法其原理是在微服务架构中，所有微服务都会注册至服务注册 中心Eureka,每一个微服务都会作为一个Eureka客户端，在微服务上线时通过服务注册机制 将该微服务的元信息包装后提交至服务注册中心，注册应用实例成功后获得租约（Lease）, 租约的元信息包括：服务拥有者信息，注册时间戳，服务上线时间戳，持续时间，元信息最 后更新时间戳。注册成功后，客户端通过固定时间间隔向注册中心server发起续租，避免租

约过期。每个Eureka Client节点都会在本地缓存维护一份自己要访问的服务列表清单，这些 服务清单来自于Eureka Server注册中心，而任何服务的下线与上线在注册中心都会进行对应 的操作，客户端通过定时从注册中心拉取可用服务列表保证客户端自身缓存的待访问服务列 表清单中服务的可用性，将原本釆用服务端负载均衡需要的手动介入的操作改进成客户端自 主维护服务列表。

在微服务架构中，每一个微服务作为一个客户端节点从注册中心获取到具体的服务实例 相关的Instanceinfo列表，将这份列表信息缓存在客户端本地，通过定时线程向server端拉取 最新的服务列表信息并更新本地列表信息。

Spring Cloud提供了一个ILoadBalance接口，其内置了实现负载均衡的一系列抽象操作, 通过继承该接口，自定义实现具体的负载均衡抽象操作。接口的UML类图设计见表4.1：

|  |  |
| --- | --- |
| 表 4.1 ILoadBalance 接口 类图 | |
| 抽象方法 | 说明 |
| addServers | 向客戸端维护的实例列表中添加服务实例 |
| chooseServers | 选出具体服务实例的策略 |
| markServerDown | 标记实例列表中某个实例停止服务 |

通过实现ILoadBalance接口重写其抽象方法，在本系统中设置了三种客户端负载均衡算

法：

1. 基于线性轮询的负载均衡策略

概念：当微服务初次启动上线时，此时所有的微服务都属于初次访问，此时釆用基于线 性轮询的负载均衡策略，此时客户端会对缓存在本地的待访问列表通过计算列表元素下 标依次进行访问。具体执行流程如下：

1. 获取当前所有可用服务列表及列表长度upCounto
2. 获取当前所有服务列表及列表长度severCounto
3. 维护一个全局原子变量count,确保在多线程场景下的线程安全，获取当前count的 值currento根据计算公式(current+1) %serverCount获得下一■次访问的服务在当前所有服务 列表中的下标位置。
4. 根据(3)中获取的下标位置获取Server对象实例，该实例包含了选中的微服务实例的 host 和 porto
5. 基于最小请求数的负载均衡策略

概念：当客户端本地缓存的服务列表中的服务数量小于事先定义的阈值时，此时不会釆 用基于轮询的负载均衡策略，转而釆用基于最小请求数的负载均衡策略。在本负载均衡策略

29

中利用LoadBalancerStatus保存的服务实例访问统计信息来选择满足要求的实例。通过遍历当 前缓存的服务列表，过滤掉因为发生故障被断路器断开的实例，在正常运行的实例中找出当 前并发访问次数最少的服务实例，该服务实例会被作为此次负载均衡策略所选中的待访问实 例。具体执行流程如下：

1. 获取当前所有服务列表serverListo
2. 遍历服务列表，判断当前服务实例是否己经被断路器断开。
3. 如果未被断路器断开，则获取当前实例的并发请求数。
4. 在遍历过程找出所有可用实例中当前并发连接数最小的一个服务实例作为此次负载 均衡策略中被选中的服务实例。

3.基于响应时间权重的负载均衡策略

概念：当服务既不是初次上线，且当前客户端缓存的服务实例清单中的服务数量大于阈 值，则采用基于响应时间权重的负载均衡策略。在本策略中会在初始化时启动一个定时任务， 该定时任务用于给每个服务实例计算权重。通过为服务列表中每一个服务实例计算平均响应 时间，进而根据得到的平均响应时间计算每个实例的响应权重，最终得到每个实例对应的响 应权重区间。最后在选择具体服务实例时通过生成的随机数去遍历服务清单，如果当前遍历 到的服务实例权重大于等于随机数则选择当前实例作为此次负载均衡策略所选中的服务实例。 具体执行流程如下：

1. 获取当前所有服务列表serverListo
2. 遍历当前服务列表，获取每个服务实例的响应时间，得到所有服务实例的平均响应时 间总和 totalResponseTime。
3. 设置一个基准权重值weightSoFar,将该值初始化为0。逐个计算每个实例的服务权重， 计算方式为：weightSoFar(当前客户端节点的基准权重值)+ totalResponseTime(累积响应时间 总和)-该实例平均响应时间，计算出每个服务实例的权重后存入权重集合中，并且每一个计 算好的权重需要累加到weightSoFar上供下一个服务实例计算使用。
4. 根据上一步得到的权重值作为各个服务实例的权重区间的上限值，每一个服务实例的 区间下限是上一个实例的区间上限，由上面的计算过程我们可以发现，实例的平均响应时间 越短，对应的权重区间宽度就越大，说明该服务实例运行状况和性能都比较出色，而权重区 间的宽度越大则被选中的概率的就越高，从而在高并发场景下使得运行状况良好的机器被选 中的概率大于运行状况普通的机器。
5. 选取最后一个实例的权重maxToalWeight,生成一个范围在［0, maxToalWeight］之间的 随机数，接下来遍历(3)中生成的权重集合，当遍历到的当前服务实例的权重大于等于得到的 随机数时，则该实例作为此次负载均衡策略所选中的服务实例。

基于响应时间权重的负载均衡算法策略执行流程图如图4. 3所示：

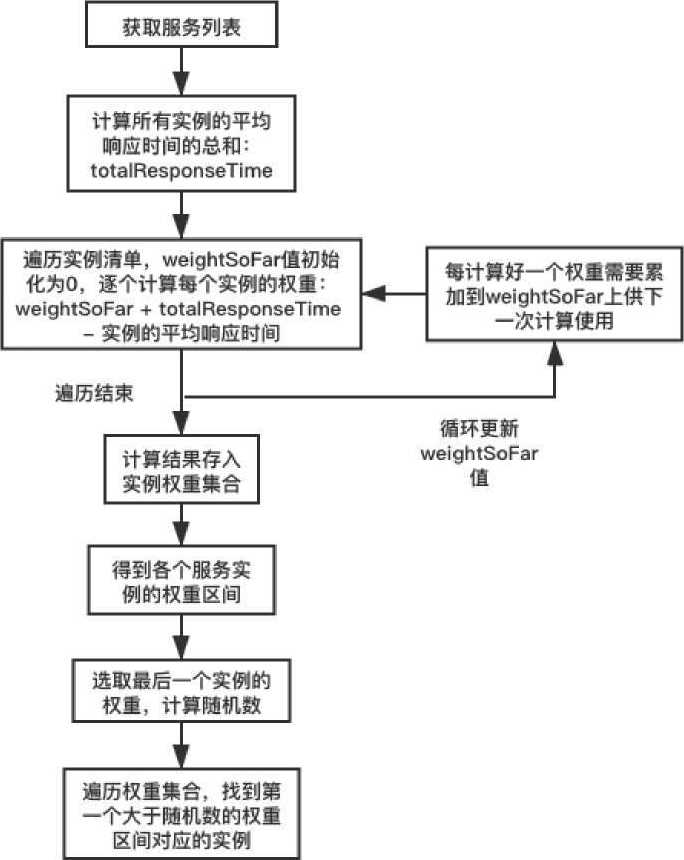


图4.3基于响应时间权重的负载均衡策略

4.1.2客户端请求限流机制

在任何一个分布式系统中，其系统的性能都有一个上限，当并发量超过这个上限时，可 能会导致系统的响应延迟，更严重者可能导致系统发生雪崩，因此在任何时刻我们都要保证 系统的并发请求数量不会超过某个阈值，而限流算法的存在就是为了满足这个目的。

常见的限流算法一般有漏桶算法，令牌桶算法。漏桶算法的原理是釆用将系统处理请求 的限制成恒定速率，就像一个漏斗，不管上面的流量多大，桶中流出的速率始终不变，但这 种算法无法应对突然之间的大流量场景。令牌桶算法则可以在限制调用的平均速率的同时允 许有一定程度的突发调用。在令牌桶算法中存在一个桶，用于存放固定数量的令牌，通过设 置一种机制，以一定的速率往桶中放令牌，每次用户请求进来调用需要先获取桶中的令牌， 只有拿到令牌才有机会继续执行，并且放令牌的动作是持续执行的，如果桶中令牌数量到达 上限，就会丢弃令牌。相比于漏桶算法，令牌桶算法可以应对一定程度上的突发流量请求。

在本系统中釆用了基于令牌桶的限流和基于Redis的计数器限流，接下来详细介绍这两 种实现。

基于令牌桶的限流，通过RateLimiter.create(double count)方法创建指定每秒允许通过的请 求数量，在每个请求进入时通过RateLimiter.acquire()去尝试获取令牌，如果获取失败，则任 务请求分发微服务不会对该请求进行下一步处理。通过对RateLimiter类进行拓展，设置了两种 启动模式：平稳突发启动和冷启动两种模式。

所谓平稳突发启动模式通过RateLimiter的子类SmoothBursty类实现，其含义是令牌生成 速度是恒定的，假设用户配置的平均发送速率为t,则每隔1/t秒生成一个新的令牌被加入到 令牌桶中，所有请求进入时获取令牌的速率是恒定的，这种模式的优点是假设某一段时间内 请求数量较小，则令牌桶中会有多余的令牌积累下来，当下一段时间有突发流量产生时，因 为有多余令牌的积累，则在系统可承受的范围内，可以缩短部分请求获取令牌的等待时间。

所谓的冷启动模式通过RateLimiter的子类SmoothWarmingUp实现，通过将当前令牌桶 中的空闲令牌数分成两个区间值：［0, thresholdPermits］和(thresholdPermits, maxPermits)。两 个区间对应不同的状态值cold和formal,分别代表冷启动区间和平滑区间。当请求进入时如 果当前系统处于cold状态即位于冷启动区间，则从(thresholdPermits, maxPermits)区间获取 令牌，所需要等待的时间相比于从区间［0, thresholdPermits］拿相同令牌所需要等待的时间会更 长，当请求逐渐增多，令牌桶中的令牌数消耗越来越多，当出现空闲令牌数减少到阈值 thresholdPermits以下时，此时请求获取令牌的时间会逐渐趋于稳定，这就是所谓冷启动模式， 尤其适用于依赖服务的部署机器在逐渐上线部署时，通过控制前期获取令牌的等待时间来这 到减缓系统压力。

基于Redis和lua脚本的计数器限流，上文所述基于令牌桶的限流主要用于指定单个任务 请求分发微服务的限流，在本系统中还提供了通过Redis为多个任务请求分发微服务在单位 时间内的限流措施。具体操作流程如下：

1. 自定义RateLimiter注解，该注解包含三个属性：key：写入Redis的限流key； limit： 限流的时间频率内限制通过的请求数；expire：限流的时间频率。
2. 将该注解引用在指定的接口处，使用AOP进行切面拦截，通过切面获取用户自定义 注解的三个属性值，在系统启动时会加载lua脚本路径，接下来通过redisTemplate.execute() 方法调用lua脚本，将三个属性值作为参数传入。
3. 首先脚本获取传入的参数，执行redis.call('incr\key),代表对传入的key做自增操作。
4. 如果(3)中的 key 是首次生成，则执行 redis.call(1 expire',key, tonumber(expire)),代.表对该 key生成一个过期时间。
5. 接下来的每一次请求都会执行自增操作。当脚本执行结果返回1,代表单位时间内请 求数未超过定义的阈值，则当前请求可以继续执行；当脚本会返回0,代表已经超过限流请 求限制，即规定时间内的访问请求数大于所定义的limit值时，触发限流。

因为在Redis中原生支持lua脚本，Redis会将每一个lua脚本作为一条原子语句执行， 能够很好的满足事务性和原子性，不用担心出现并发问题。同时使用lua脚本可以避免向Redis 发送多次请求，减少网络传输的消耗。通过在Redis中设置计数器的形式为多个任务请求分 发微服务提供了整体限流，从而控制整个集群的限流频率。

4.1.3微服务熔断器

在微服务架构中服务与服务之间通过远程调用的方式进行通信，一旦存在某个服务出现 故障，极有可能会导致整个服务的级联故障，进而引发整个系统不可用的情况，造成服务雪 崩效应。在本系统中通过基于Spring Cloud Hystrix实现了断路器模式，当某个服务发生故障 时，通过配置的Hystrix断路器给调用方返回一个错误的响应，从而避免调用方由于远程调用 故障而长时间无法得到响应进而始终占据线程资源，最终耗尽服务器可用线程资源。在本系 统中通过釆用基于Hystrix的线程池隔离，确保无论哪一个请求接口出现问题都不会对其余正 常接口的执行流程造成影响。

具体操作流程如下：

1. 通过引入 spring-cloud-starter-netflix-hystrix 依赖，在 application.java 启动文件上添加 @EnableCircuitBreaker注解开启断路器功能。
2. 因为在本系统中微服务之间调用通过远程调用实现，在任务请求分发微服务中会定义 hystrix接口，通过创建回调类，重写接口中的方法作为当接口出现异常时的返回值。在指定 接口方法上通过注解标记执行熔断时的具体执行策略逻辑所对应的方法。
3. 在service层调用方法上添加@HystrixCommand注解，设置主要属性:fallbackMethod： 指定执行的服务熔断方法，commandkey：命令名称，用于区分不同命令，groupKey：分组名 称，方便hystrix仪表盘根据不同的分组统计命令的告警，threadPoolKey:线程池名称，用于 划分熔断线程池。

在任务请求分发微服务中采用了基于线程池的资源隔离措施，hystrix通过自定义的 commandkey对发送请求的对象和执行请求的对象进行逻辑解耦,当第一•次创建command时, 根据threadPoolKey和groupKey创建一个新的线程池，该线程池用于存放指定接口的请求线 程。通过将发送请求线程与执行请求的线程隔离，一旦因为某个依赖调用接口发生响应故障， 导致请求队列饱和或者指定key名称的线程池可用线程消耗殆尽时，hystrix将会拒绝服务， 并且后续的请求线程可以快速失败，直接执行自定义的fallback逻辑，从而避免因为依赖服 务的故障导致级联雪崩引发的整个微服务不可用的场景出现。图4.4展示了服务熔断的处理 流程：

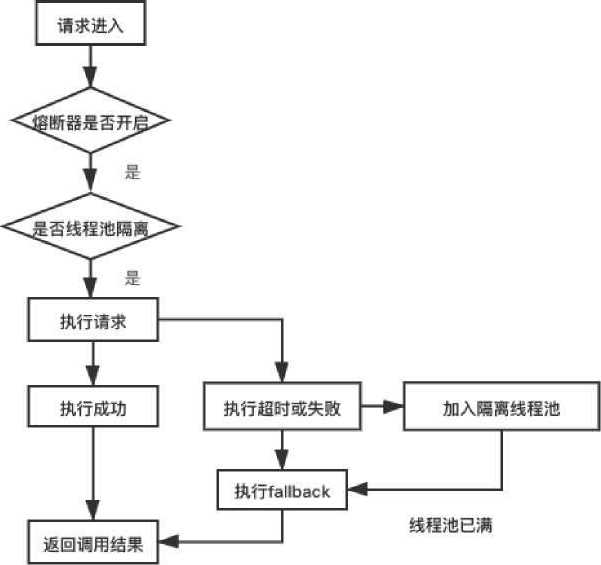


图4. 4服务熔断处理流程

4.2任务请求预处理微服务

任务请求预处理微服务主要负责处理由任务请求分发微服务下发的爬虫任务请求。在任 务请求预处理微服务中针对每个爬虫任务的自定义数据提取规则都会生成一个分布式场景下 的全局唯一 ID,该ID会与数据提取规则组合成一个key-value的键值对存入Redis集群中， 该ID会作为后续页面解析微服务中解析HTML文木数据的查询依据。并且针对可能出现的 URL重复爬取的场景，在本系统中采用了基于布隆过滤器和Redis的双重校验，从而减少因 为URL重复爬取所导致的系统资源浪费。在本系统中任务请求预处理微服务会作为生产端， 负责将爬虫任务信息包装后投递至Kafka集群中，由作为消费端的页面抓取微服务负责消费 数据。

4.2.1全局分布式唯一ID生成算法

经研究后发现，在基于微服务架构的分布式爬虫系统中，从逻辑上看每个爬虫任务都是 独一无二的，从而每一个任务都需要有一个标识ID,这个标识ID必须是全局唯一的，否则 就无法在逻辑上区分具体的任务，在单机环境下可以通过Java自带的UUID工具类采用结合 机器网卡，时间戳，一个随机数来生成ID,但这种方式具有很大的局限性，一方面采用这种 方式生成的ID是无序的，在查询时无法通过索引进行快速查找，另一方面在分布式部署的环 境下UUID无法保证每个ID都是全局唯一的，这在分布式系统中是致命的缺点。

在木系统中对于生成的数据提取规则ID需要达到的要求：(1)在分布式环境下确保谁让 你工程ID的全局唯一性。(2)确保生成的主键ID是相对有序递增的，方便查询时根据索引加 快查询效率。

鉴于这些条件，任务请求预处理微服务提出了一个基于zookeeper的分布式ID的生成算 法。在该算法中其生成的算法结果是一个long型的ID,共有64位，其中1位作为符号位， 41位作为时间戳，依赖于当前毫秒级生成，10位作为数据机器位(分别包含5位的数据标识 位和5位的机器识别位)，12位的序列号。所有生成的ID都是按时间趋势递增的，同时因为 数据标识位和机器识别位的存在可以确保不会产生重复IDo接下来对各个位置的参数构成作 详细阐述：

1. 1位的符号位：因为long型在Java中是带符号的，正数为0,负数为1,在本算法中 统一设置为0;
2. 41位基于毫秒级别的时间戳：当前毫秒级别时间戳减去ID生成器开始使用的时间戳, ID生成器开始使用的时间戳通过在程序中指定。
3. 10位数据机器位：当前机器的工作ID,包括5位数据标识位和5位机器识别位，5bit 可以表示的最大整数是31,所有可以使用0到31这32个数字用于表示不同的数据标识位和 机器识别位。
4. 12位的序列号：用于记录同一毫秒内产生的不同ID, 12位bit可以表示的最大整数是 212\_1=4095,即可以用0到4094共4095个数字来表示同一机器同一毫秒内产生的4095个ID 序号。

在本系统中因为采用分布式部署，对于机器识别位workerlD的分配釆用依赖zookeeper 顺序节点对任务请求预处理微服务配置workerlD,其步骤如下：

1. 在任务请求预处理微服务启动时，连接zookeeper,在父节点下检查该机器ID是否已 经部署过(即在zookeeper中当前父节点下是否存在该顺序子节点)。
2. 如果有注册过，从zookeeper父节点下直接取回代表该机器的workerlD的子节点值 (zookeeper顺序生成的ID号，ID数据类型为int类型)。
3. 如果没有注册过，就在父节点下新建一个持久化的顺序子节点(节点名称格式为 ip：port),将新建子节点的顺序号作为当前机器识别位的workerlDo
4. 除了向zookeeper取回workerlD号之外，也会在本机缓存一份workerlD文件。

在最终生成的全局唯一 ID中，有41位bit数据是依赖于当前机器的时间戳，如果发生机 器时钟回拨就有可能出现生成重复ID的情况，针对这种情况在本系统中利用注册到zookeeper 时的节点时间戳进行判断，主要步骤如下：

1. 任意机器上部署的任务请求预处理微服务启动时首先检查是否已经向zookeeper父节 点注册过子节点。
2. 如果注册过，获取对应子节点ip:port节点下的workerlD,接下来将之前周期上传的子 节点时间戳与当前系统自身时间进行比较，如果小于之前节点时间戳，则认为机器时间发生 回拨，此时服务启动失败。
3. 如果未注册过，则在当前父节点下直接创建子节点。接下来获取此时父节点下的所有 子节点，获取所有子节点的之前周期性上报的系统时间，计算所有子节点平均系统时间。
4. 如果abs(所有子节点平均系统时间-当前机器系统时间)〈阈值，则认为当前机器系统时 间可用，服务启动成功，可以正常生成分布式唯一 ID。
5. 通过定时任务，每隔指定时间向zookeeper T的节点上报自身系统时间。预防时钟回 拨的检验流程如图4.5。

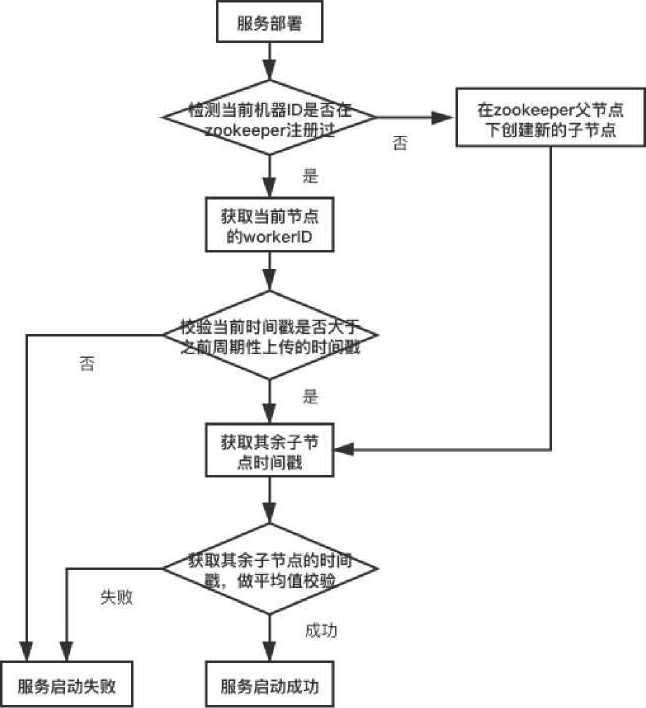


图4. 5解决时钟回拨问题

通过基于毫秒时间戳和数据机器位以及基于zookeeper解决时钟回拨问题所生成的64位 ID值，可以有效防止出现分布式场景下的ID重复，确保每一个爬虫任务中的数据提取规则 所对应的1D都是全局唯一的，最终将生成的分布式全局唯一 ID与数据提取规则一一对应写入 至Redis中对应第三章中表3. 2。

4.2.2 URL双重判重机制

基于分布式爬虫系统中，抓取页面，解析页面都属于耗费系统资源比较多的场景，而在 大量用户创建的爬虫任务中，有可能会出现短时间内的重复URL爬取，而频率过多的重复爬 取会对计算机资源造成较大的浪费，因此需要在分布式爬虫系统中对URL的去重处理就显得 尤为重要A〕。常见的去重方案是釆用Java自带的HashSet,针对每一个URL通过HashSet内 置的contains()方法进行判断是否已存在，这种方案的优点是针对URL的重复爬取不会存在 误判，缺点是HashSet是基于内存的，当爬取的URL数量较大时，会导致内存消耗过大。因 此这种方案在分布式爬虫系统中一般不会釆用。

在任务请求预处理微服务中，需要对用户提交的爬虫请求的URL进行判重处理，当出现 短时间内重复URL时，釆用直接远程调用页面解析微服务，从MongoDB中取出该页面的己 有HTML文本数据，从而跳过页面抓取微服务，避免重复抓取对计算资源的浪费。因此任务 请求预处理微服务釆用了基于布隆过滤器和Redis的URL双重判重机制。

第一层判重机制是基于布隆过滤器［35 一 37］的内存判重，布隆过滤器是一种空间效率极高的 随机数据结构，它的底层是一个m位的bit数组。布隆过滤器的原理是当一个元素被加入集 合时，通过K个独立的散列函数的计算将该元素映射成位数组中的K个下标索引，将这些索 引下标的位数组元素值置为1。在进行元素检索时，通过对该元素进行K次散列运算时得到 的k个索引下标所对应的位数组元素是否均为1来判断该集合是否中是否存在该元素，如果 k个索引下标所对应的位数组元素中至少存在有一个为0,则表示待查询元素一定不存在，如 果k个索引下标所对应的位数组元素中全为1,则表示待查询元素有可能存在。在任务请求 预处理微服务中，初始化了一个数组长度为100000位的bit数组，其初始值全部置为0,设 置了 3个基于murmur3\_hash的散列函数。针对每一个URL,通过三次hash计算，如果得到 的位数组索引下标对应元素至少存在一个为0,则将对应下标元素置为1,且判定该URL属 于未被爬取过，如果得到的位数组索引下标对应元素全为1,则判定该URL可能属于重复爬 取，此时釆用第二层判重。

第二层判重是基于Redis的set数据结构，Redis的set集合类似于Java中的Hashset,内部 的键值对是无序且唯一的。在本系统系统中所有爬取过的URL都会添加进Redis中的指定set 集合即表3.3中，每一次进行判重时通过添加Redis Set的返回值判断是否重复爬取，若返回 1则代表添加成功，此URL属于未重复爬取；若返回0则代表该URL已经被爬取过，此时 直接远程调用页面解析微服务。

任务请求预处理微服务的URL双重判重处理机制整体流程如图4.6：

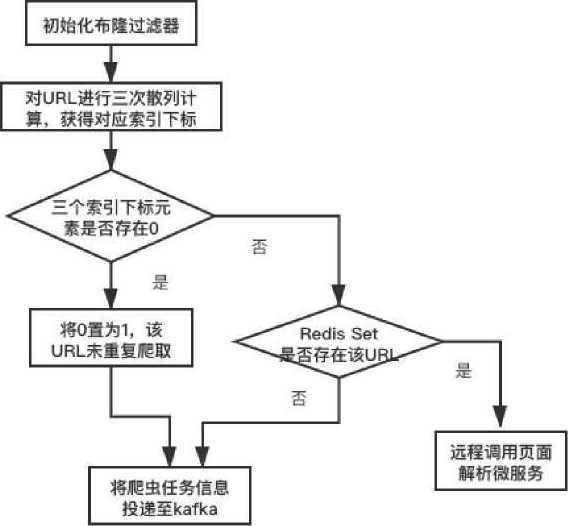


图4. 6基于布隆过滤器和Redis的URL判重机制

通过基于布隆过滤器和Redis的URL双重判重机制，既可以避免只采用布隆过滤器可能 存在的误判情况，也避免了直接釆用Redis进行判重时因为Redis的单线程机制导致影响其余 微服务对Redis的操作，在保证了判重正确性的基础上，同时又兼顾了性能。

4.2.3 Kafka集群副本配置

任务请求预处理微服务中，经过URL判重，生成分布式全局唯一 ID,在MySQL中初始 化job记录等步骤后，会将CrawlerJobMessage信息序列化后投递至kafka中的指定topic （主 题）。为了保证kafka的高可用，本系统中kafka釆用了集群部署，一共部署了 3个broker （节 点）。为每台broker创建一个配置文件，在配置文件中定义3台节点的broker.id分别为0, 1, 2obroker.id将作为集群中每个节点的唯一名称，在配置文件中配置log.dir属性作为每台broker 日志文件的存储目录。

在创建topic时，通过配置参数partition为每个topic设置分区数，分区作为topic在物理 上的分组，一个topic可以被分为多个partition,每一个partition都可以看成是一个有序队列； 通过配置参数replication-factor为每个partition设置副本，副本分为leader副本和follower副 本。Kafka通过副本分配机制把所有的partition的leader平均分配到各个broker上，所有的读 写操作都通过leader副本，follower副本负责同步消息。

在本系统中针对处理CrawlerJobMessage信息的topic设置了 3个partition,对每个partition

设置了 3个副本，从而确保每个partition的内容都有2个副本做备份，保证即使leader副本 挂了，follower副本的数据可以快速恢复分区数据，以此来保证Kafka集群的高可用。集群副 本示意图如图4.7所示：

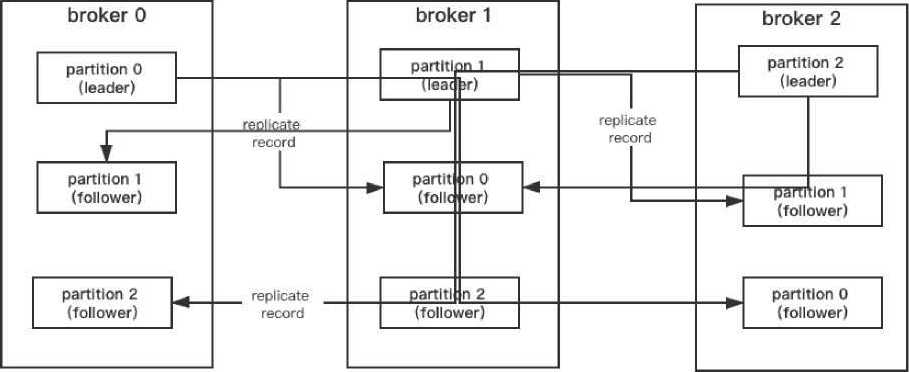


图4. 7 Kafka集群副本示意图

4.3页面抓取微服务

页面抓取微服务负责作为消费端监听kafka中由任务请求预处理微服务生产的 CrawlerJobMessage信息，并从序列化后的数据中得到待爬取的URL路径，结合从cookie池 和user-agent池中获取的数据，结合代理IP,通过HttpClient框架模拟get或post请求，获取 返回的ResponseEntity数据，根据返回的状态码是否为200判断页面抓取是否成功，如果成 功则将获取的HTML文本数据以及对应的job信息包装成DownloadPagelnfoMessage对象并 序列化后投递至kafka中指定topic,将抓取的数据交由下游作为消费端的页面解析微服务消 费，同时釆用异步线程去异步更新数据库中对应job的运行状态为下载成功。如果返回的状 态码不是200,则将该URL加入重试队列，开启重试机制。

4.3.1基于线程池的多线程抓取实现

在页面抓取微服务中釆用多线程执行页面抓取任务。在操作系统中线程相比进程更节约 资源，如果在操作系统中新建一个进程，就要为这个进程分配新的内存地址空间，相比较而 言这是一个及其耗费资源的过程。目前的服务器中所釆用的CPU基本都是多核CPU,因此在 实际开发中对于某些比较耗时的操作，我们可以通过釆用多线程的方式去并发执行响应，从 而提高CPU的使用率。由于线程与线程之间的通信比较便捷，多个线程之间共享同一个进程 的内存空间，因此在保证多线程环境下的线程安全的前提下，釆用多线程可以大大提高系统 的响应效率和执行速度。

在实际使用中线程是比较占据系统资源的，如果对线程管理不善很容易导致系统问题， 鉴于此在页面抓取微服务中，采用了线程池来管理多线程。使用线程池管理多线程主要有如 下几个好处：（1）降低资源消耗，线程池中的部分线程是可以复用的，通过复用核心线程池中 的线程来尽可能的降低系统资源的损耗。（2）通过线程池复用降低线程的销毁次数来减少资 源消耗。（3）提升了响应速度，通过复用核心线程池中的线程，避免了反复创建新线程所导 致的时间消耗，在高并发场景下可以提升系统的整体响应时间。要做到合理的使用线程池首 先要判断多线程场景下的任务是属于10密集型任务还是CPU密集型任务，对于CPU密集型 任务通常设置线程池大小为CPU核心数加1,对于10密集型任务通常设置线程池大小为CPU 核心数\* 2。

在页面抓取微服务中消费端将监听到的kafka的二进制数据序列化后，取出URL路径, 通过使用HttpClient框架构造一个HttpGet对象，user-agent池和cookie池中随机取出一个 user-agent头和cookie值，获取配置类中其余属性信息，将这些信息同步骤一中获取的URL 属性值作为 HttpHeader值传入 HttpGet,初始化一个CloseableHttpClient对象，通过 ThreadPoolExecutor类的submit（）方法提交至线程池中执行页面抓取，将抓取到的信息投递至 kafka o

在页面抓取微服务中通过ThreadPoolExecutor类构造线程池，通过设置如下参数： （l）coolPoolSize:表示核心线程池大小。当执行submit或者execute方法提交任务时，如果当前 核心线程池的线程个数小于coolPoolSize,则线程池会创建新的核心线程去执行提交的任务, 核心线程池中的线程永远不会被回收。在页面抓取微服务中考虑到从远端网站抓取页面属于 CPU密集型任务，因此根据机器情况设置核心线程池大小为4个线程。（2）maximumPoolSize: 表示线程池所能创建线程的最大个数。如果当前线程池中的线程个数没有超过 maximumPoolSiz,旦内存中的阻塞队列已满，则会创建新的线程来执行任务，在本系统中我 们设置为6。（3）keepAliveTime:代表空闲线程存活时间。如果当前线程池的存活线程个数大于 coolPoolSize值，并且核心线程池外的空闲线程的存活时间大于keepAliveTime,此时会将这 些空闲线程销毁，从而尽可能降低系统资源消耗，在本系统中设置为2秒。（4）workQueue:阻 塞队列。当核心线程池线程均处于工作状态时用于保存提交至线程池中的任务，在本系统中 使用了基于链表的阻塞队列LinkedBlockingQueue,初始化长度为200。（5）handlcr：饱和策略， 当线程池的阻塞队列中任务数达到初始化长度且当前工作线程数达到最大线程数时，根据定

义的饱和策略去处理新提交的任务。分布式爬虫系统中负责执行页面抓取任务的线程池工作 流程如图4.8所示：

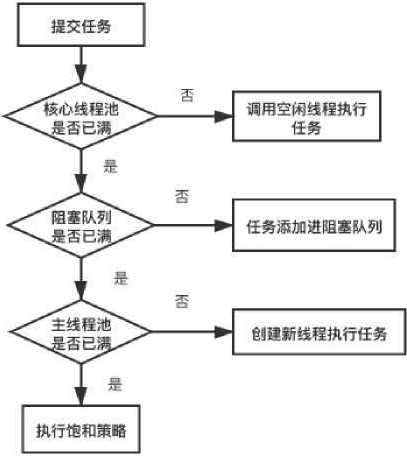


图4. 8线程池执行流程示意图

4.3.2中间件发送消息防丢失实现

页面抓取微服务会作为生产端将通过HttpClicnt框架抓取到的页面信息经过包装后投递 到Kafka中指定topic, 一旦作为生产端的页面抓取微服务投递消息时因为投递的broker节点 发生宕机或者网络问题导致出现了网络阻塞，导致了投递消息失败，那么就会从逻辑上丢失 了已经完成抓取的页面数据，下游的页面解析微服务将永远无法解析到该页面，从系统可靠 性来讲，这是无法容忍的，因此在页面抓取微服务中针对消息丢失问题结合Kafka的特性提 出了相应的解决方案。

在消息投递时，每一个消息都是一个ProducerRecord对象，在其中必须指定消息所属的 topic和消息值，此外还可以指定消息所属的partition和消息的key。在Kafka中生产端的生 产数据默认是先写到内存中的pageCache,然后通过后台的定时线程定期flush到磁盘上。在 本系统中首先基于Kafka提供的log.flush.interval.ms配置，通过该配置可以设置定期flush的 时间间隔来简短消息落盘的时间间隔，防止出现消息丢失。但即使设置了较小的定期flush时 间，也并不能完全保证数据不丢失。在第二章中介绍了 Kafka是一个可分区和可复制的消息 队列。因此在页面抓取微服务中利用Kafka的分区副本机制，通过设置-replication-factor参 数为每一个topic数据设置了多个分区，每个分区存放在不同的broker节点上，确保每个分区 数据都有自己的备份，每一■个分区存在leader和follower的角色，并在zookeeper中维护了一 份ISR列表。

除此之外，生产端向broker中指定topic写入数据时，默认的消息级别是 request.required.acks=O,这表示生产者发送消息后，不会等待分区的leader确认，此时就有可 能出现消息的丢失，在页面抓取微服务中，通过将request.required.acks值设置为-1,代表生 产端在发送消息时要等到指定分区的leader以及所有副本都同步后，才会返回确认信息，除 此之外，通过配置参数min.insync.replicas的值为2,代表每一次最小同步的副本数为2个副 本，如果同步成功的副本数未达到这个最小数的要求，那么生产者会抛出异常，本次写入不 会成功，通过该参数的配置确保大部分副本同步数成功。

综合以上所述，在页面抓取微服务中通过配置内核pageCache将数据flush到磁盘的最小 时间间隔，将消息确认级别request.required.acks值设置为-1,以及设置每一次写入消息时分 区的同步成功的最小副本数三个措施来使得生产端保证消息投递不丢失，从而不影响下游的 页面解析微服务的正常运行。

4.3.3爬虫动态代理池

考虑到在爬虫系统中可能会存在短时间内对同一网站的多次抓取，某些目标网站可能会 存在对爬虫的识别与限制，例如计算同一 IP的访问频率，针对短时间访问过于频繁的IP进 行封禁，抓取请求直接返回404,从而导致页面抓取失败；除此之外，目标网站还会通过分 析请求头中的user-agent数据来判断是否是正常浏览器发出的请求数据。

针对user-agent反爬机制，在页面抓取微服务中通过在Redis中构建user-agent池，预先 在数据池中存放了大量有效user-agent头，使用list作为user-agent池的数据结构，通过定时 从池中取出一批user-agent数据加载到内存，在每次构建HTTP请求时，随机从中取出一条 user-agent 数据，通过 HttpGet 类的 addHeader（'user-agent5,取出的 user-agent 数据），将数据 添加到请求头。

针对检测IP访问频率的反爬机制，在页面抓取微服务中采用预先构建IP proxy代理池的 方式坚决，在Redis中构建了两个list集合，一个存放白名单IP对应第三章中表3.4, —个存 放黑名单IP对应第三章中表3.5,会白名单IPlist中会存放预先获取从网上免费代理网站获取 到的大量IP,在使用时依次从白名单list首部取出代理参数，接着使用HttpClient框架的 HttpProxy类将代理IP参数添加，然后根据页面抓取返回的response信息中的状态码是否为 200判断是否抓取成功，如果状态码为200,则认为此IP是有效的，将此IP添加至白名单list 末尾，如果返回状态码不是200,则认为此IP在本时间段可能被目标网站封禁，将此IP从白 名单list中移除，添加至黑名单list中，防止在接下来的爬虫请求中无效IP被重复使用从而 导致抓取失败。

**4.4**页面解析微服务

页面解析微服务作为爬虫系统的最后一环作为消费端负责从kafka指定topic中监听由上 游作为生产端的页面抓取微服务所投递的抓取后的页面HTML纯文本数据。将将监听到的数 据反序列化成字符串类型，将完整的纯文本存入MongoDB数据库中的full\_page\_info表中。 接着从Redis中取出规则ID所对应的数据提取规则，釆用jsoup作为数据提取框架，根据用 户自定义的CSS或者Xpath提取规则提取出相应数据，并将解析后的数据存入MongoDB数 据库中的parse\_page\_info表中，该数据将作为用户查询爬虫任务结果的返回依据。

4.4.1基于提取规则的页面文本数据提取

页面解析模块通过监听Kafka中名称为page download的topic获取由页面抓取微服务投 递的页面数据，将数据序列化后获取抓取的页面HTML纯文本数据。在数据提取的框架方面 选用了基于Java语言编写的jsoup,该框架可以基于CSS选择器来操作DOM元素，从页面 纯文本中提取出指定文本数据。

jsoup通过使用状态机进行语法和词法分析.jsoup的Parse在实现上非常复杂，涉及到了 相当多的状态机知识，不过由于HTML文本的特殊性，其在结构上有规律可循，并且只需要 解析到DOM树为止即可ojsoup通过状态模式state和转移模式transition两大部分构成,jsoup 通过状态转移模式进行解析，在HTML纯文本中数据一般都包裹在由'〈‘开始和'>'结的 各种HTML标签中，js。叩就是通过开始标签和结束标签来控制状态转移方程，从而从层层 嵌套的HTML纯文本中分离得到对应标签的TagName,并返回语法状态〔38】。

首先通过parse。方法，将HTML纯文本信息转换成Document类型，接着根据从kafka 中序列化得到的数据提取ID从Redis中获取该ID对应的页面信息提取规则，遍历数据提取 规则，使用document的select()方法将根据CSS选择器获取得到的标签转换成Element对象， 这里的CSS选择器支持通过标签查找元素，通过ID查找元素，通过命名空间查找元素，支 持属性查找元素，支持通过属性的正则表达式查询元素oElement类中有很多常用方法，如text() 方法是用于获取当前节点以及该父节点下子节点的所有HTML标签和样式以外的内容，例如 标签对应的纯文本，转义字符等；tagName()方法是用于获取当前HTML标签元素的名称，例 如td,href等；children()方法是用于获取当前父节点的所有子元素，它的返回类型也是Elements 类型。这里通过对获取到的Element对象使用text()方法提取到对应的文本信息，并将获取到 的文本信息与对应的提取规则 对应，序列化成json格式写入MongoDB中。

4.4.2基于Kafka的页面数据预防重复消费解析

针对在分布式爬虫系统中，在监听到Kafka中的页面下载数据后，会开启一个异步线程 池去执行页面HTML文本数据解析，因为通过jsoup解析HTML文本数据时会先在内存中通 过状态机构建DOM树，通过DOM树进行标签匹配，而构建DOM树的过程是一个及其耗资 源的操作，因此如果出现作为消费端的页面解析微服务重复消费了 Kafka中的数据，就会出 现异步线程池对同一份HTML纯文本数据重复在内存中构建DOM树，从而造成计算资源的 浪费，因此避免消费端出现重复消费topic数据可以有效提升分布式爬虫系统在大量爬虫任务 执行时的运行效率与计算资源利用率。

出现重复消费的原因是因为Kafka的consumer端在消费数据时会首先从broker中读取一 批消息数据进行相应处理，处理完成之后再提交新的offset,这种情况下当consumer端的消 费能力比较低或者处理过程较复杂导致消费速率过低，使得取出的这批消息数据未能在Kakfa 一次session的过期时间内处理完成，导致Kafka自动提交offset失败，然后Kafka会对当前 监听的topic重新分配新的分区给消费者，此时消费者保存的offset依旧是旧offset,消费端 依据旧的offset重新去消费新的partition中的重复数据，此时就出现了重复消费。

为了解决消费端重复消费的问题，在页面解析微服务中的解决方案分为两部分，第一部 分首先在配置文件中将Kafka消费者的配置：enable.auto.commit的值设置为false,其含义是 禁止Kafka自动提交offset,通过spring-Kafl<a启动一个invoker线程，实际消费时会转变成 从topic中取完数据后，将数据丢到阻塞队列中，由invoker线程去消费，如果阻塞队列中数 据已满，则consumer会停止从Kafka中取数据，直到阻塞队列中的数据消费完，然后提交阻 塞队列中己经消费的数据在Kafka中的offset,最后再去Kafka中读取新的数据进入阻塞队列。

第二部分的解决方案是在第一部分的基础上，当每次监听到Kafka中的完整的 DownloadPagelnfbMessage对象时，通过将DownloadPagelnfoMessage对象的用于记录页面解 析规则的全局唯一 ID写入Redis中的消费端去重表即第三章中表3.6,利用Redis的set集合 的不可重复性，如果ID添加Redis失败，意味着该条消息之前已经被消费过了，则这条消息 属于重复消费，将其丢弃，如果ID添加Redis成功，代表该消息未被消费过，此时可以继续 执行页面解析逻辑。

综上所述，通过釆用禁止Kafka的消费端自动提交offset,改为消费端手动提交offset来 避免消费逻辑过于复杂导致的session超时，以及利用Redis作为外部存储介质，通过每条消 息中的全局唯一 ID作为判重依据，以此来确保作为消费端的页面解析微服务不会进行重复消 费，避免了对计算资源的浪费，提升了系统整体运行效率。

**4.5**数据存储

在基于微服务架构的分布式爬虫系统中，存在着不同的数据存储场景，对于不同的存储 场景需要综合考虑系统性能，吞吐量等不同角度，从而选择不同的数据库作为存储介质。例 如使用了 MongoDB的存储场景主要有两个：抓取的HTML纯文本内容存储在MongoDB的 full\_page\_info表中，基于CSS选择器提取的有效数据都存储在MongoDB的parse\_page\_info 表中；关系型数据库MySQL的存储场景主要用于爬虫任务的相关信息，包括任务运行状态， job信息，数据提取【D等基于crawlerjob表中的相关字段。非关系型数据库Redis的存储场 景主要用于：存储pattemld与数据提取规则的键值对，存储IP黑白名单，存储已爬取URL 的set集合，消费端基于全局唯一 ID的消息去重。

在分布式系统中，数据库的可靠性是系统整体可靠性中相当关键的一环，本系统中的页 面抓取后的解析结果，数据提取规则，IP代理调度等场景都严重依赖MongoDB数据库和Redis 数据库，一旦出现问题会整个分布式爬虫系统的使用，而传统SOA系统中数据库大都釆用单 点部署，一旦出现问题会导致涉及到DB的服务不可用，更严重的情况下会数据丢失，从而 影响整个系统的可靠性。因此为了确保MongoDB和Redis的数据不丢失，以及数据库部分节 点奔溃也不影响服务正常访问，在本系统中对MongoDB的部署釆用了基于分片机制的集群 部署，对Redis的部署釆用了基于哨兵模式的集群部署，并开启基于RDB和AOF的持久化 策略，将内存数据持久化到磁盘，以此来保证分布式爬虫系统中MongoDB和Redis的高可用。

4.5.1实现基于MongoDB数据库的数据存储高可用机制

在分在本系统中MongoDB作为存储抓取页面的纯HTML文本以及解析结果键值对的存 储介质，为了满足用户提交了大量爬虫任务的场景下会海量数据的存储要求，通过数据分片 来支持海量数据集的部署，采用了数据副本的机制来满足集群的高可用性"I。

所谓的MongoDB数据副本是将数同步复制到多个服务器上，通过在多个数据存储服务 器上存储相同的数据，通过副本机制来保证一定的容错性，即使在某台服务器上的MongoDB 挂掉，数据服务依然可用，同时依靠数据副本机制可以提升数据的读性能，当大量用户同时查询页面解析结果时，可以通过从不同的副本读取数据，提高分布式系统的可用性。一个 MongoDB副本集包括一个主节点和多个从节点，主节点接收所有的写操作。主节点将所有的 数据变动记录在oplog日志中，从节点会主动去复制主节点的oplog日志，然后从节点通过解 析日志内容重放数据集的变动，通过这样的方式和主节点达到数据的一致性，如果主节点挂 掉了，一个合格的从节点将发起一次选举将自己选为新的主节点。

采用副本机制的MongoDB集群的故障自动转移机制，当一个主节点和副本集中其他从 节点的连接中断超过10秒时，根据类raft算法，第一个发起选举且得到副本集中超过一半选 票的合格从节点将会成为新的主节点。为了确保公平，通过在副本集中额外维护一个单独的 MongoDB实例作为仲裁者，仲裁者不维护具体的数据集，仲裁者用于响应其他副本集中其他 成员节点的请求以及维护主节点和从节点之间的心跳，因为作为仲裁者节点的MongoDB实 例不维护数据集，同维护了完整数据集的MongoDB实例相比，占据的硬盘以及内存资源更 少，并旦一旦出现副本集节点数量是偶数的情况时，因为仲裁节点的存在，可以加入选举节 点中，使得在主节点选举的过程中具备超过一半选票的能力。

通过在主节点实例采用rs.add(HOST\_NAME:PORT)的方式，给主节点添加副本集的实例 节点。本系统中MongoDB主从复制集群架构如图4.9所示：

页面解析

主MongoDB节点

主从复制

主从复制

主从复制

副本MongoDB节点

副本MongoDB节点

副本MongoDB节点

心跳检测

心跳检测

心跳楼測

仲栽节点

图4. 9分布式爬虫系统MongoDB主从复制架构图

在分布式爬虫系统中，会有大量抓取后的页面数据需要写入MongoDB中，由此导致主 节点MongoDB的数据集会非常大，如果数据集大于系统工作内存的的情况下，会对数据库 的磁盘I/O造成很大瓶颈。有两种方案应对数据库存储数据的快速增长，一种是垂直拓展， 即增强服务器的性能，例如使用更强的CPU,加大系统内存等，另一种则是水平拓展，通过 增加服务器的部署数量提升系统的负载，相比垂直拓展具有更好的经济性。因此在本系统中 釆用了基于分片技术的MongoDB水平拓展方案。

MongoDB的分片集群分为以下几个组件：

shard：每个shared都可以作为一个副本集部署，并且都包含数据分片的一个子集。 mongos： mongos作为查询路由，负责提供分片集群与客户端之间的接口。 config-servers：用于存储集群中的元数据和配置数据。

MongoDB使用shared key对collection数据进行分片，然后将数据分片到chunk中，在 集群中MongoDB使用分片集群均衡器迁移各个chunko分片集群通过mongos路由来与集群 中的具体collection交互，客户端不允许直连到具体的shard进行读写操作。MongoDB中支持 两种分片策略：1.哈希分片，通过对shared key进行hash运算后进行分片，基于哈希后的值 对每个chunk进行分配，基于哈希的数据分片会形成更加均匀的数据分布，但在查询数据时 会导致广播操作。2.范围分片，通过对数据基于shared key的某个范围内的值进行指定chunk 的分配，分配效率的好坏取决于shared key,釆用范围分片的优点是查询时可以直接定位到具 体的chunkO图4.10展示了 MongoDB分片的架构示意图：

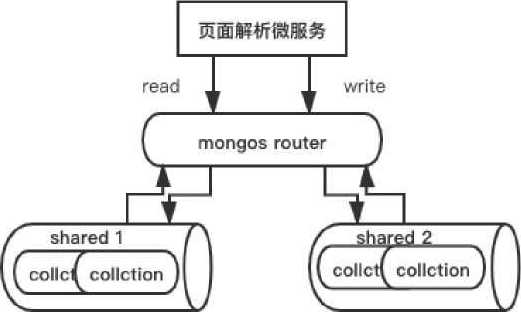


图4. 10 MongoDB分片架构

4.5.2实现基于Redis数据库的数据存储高可用机制

针对在基于微服务架构的分布式爬虫系统中多个存储场景都是基于Redis实现的Si,例 如存放每个爬虫任务的提取规则，IP代理的黑白名单，消费端消费任务去重等，因此确保Redis 可用性对于爬虫系统的正常运行至关重要，基于这个目的，在本系统中对Rcdis采用了基于 RDB和AOF的持久化策略，基于哨兵模式的Redis集群策略。

考虑到分布式爬虫系统中棊于Redis的数据存储量比较大，因此在本系统中采用基于哨 兵模式的Redis集群，哨兵模式的集群由两部分数据节点组成：哨兵节点和数据节点。哨兵 节点的主要功能是不断检查主节点和从节点的运行状况是否正常，当主节点发生故障时的自 动转移。通过在作为从节点的配置文件redis.conf中添加命令：｛slaveof IP地址：端口号｝, 设置从节点监听的主节点，在作为哨兵节点的Redis配置文件redis.conf中添加命令：｛sentinel monitor master 192.168.92.128:6379 2｝,其含义是哨兵节点监控 192.168.92.128:6379 这个主节 点，该主节点名称为master,命令最后的2与主节点故障判定有关，其含义本系统中部署了 3个哨兵节点，至少有2个哨兵节点同意才能判定主节点故障并进行故障转移。配置完成后 通过redis-server sertinel-26739.conf-sentinel命令启动哨兵节点。本系统中配置了 3个哨兵节 点，1个主节点，2个从节点，其部署架构如图4.11所示：

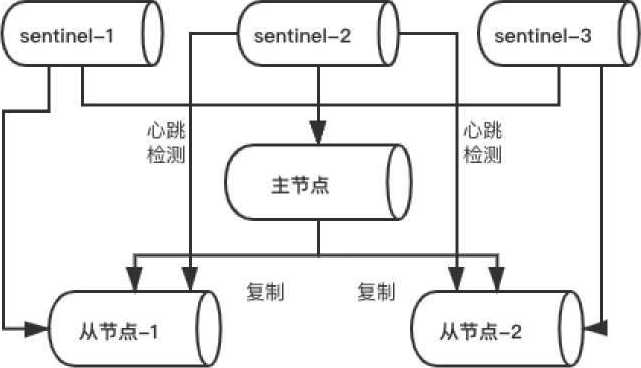


图4. 11 Redis哨兵集群部署示意图

当主节点发生故障时的处理流程如下：1.至少2个哨兵节点判断主节点出现故障，并对 主节点进行客观下线；2.各个哨兵节点通过raft算法选举出一个领导者哨兵节点，由选举出来 的领导者哨兵节点对故障主节点进行故障转移。3.在从节点中过滤掉断开的节点，根据 slave-priority从健康的从节点中选举出优先级最高的节点。4.通过slave no one命令，将前一 步中选举出来的从节点变更为新的主节点，领导者哨兵节点会向剩余的从节点发送slaveof命 令，使得剩余从节点成为新的主节点的从节点。5.将已经下线的原主节点重新部署上线后设 置为新的主节点的从节点。通过以上5步实现哨兵模式集群下完整的主节点故障转移流程。

釆用基于RDB和AOF的持久化策略是因为Redis数据库是基于内存的，一旦发生奔溃  
或者机器重启，内存中的数据会全部丢失，因此需要一套可靠的机制去对内存数据备份，一  
旦机器重启能够基于备份还原数据库。RDB持久化，是指在指定的时间间隔内将内存中的数  
据集快照写入磁盘。实际操作过程是，fork 一个子进程，先将数据集写入临时文件，写入成  
49

功后，再替换之前的文件，用二进制压缩存储。具体操作过程是在Redis cli中执行bgsave命 令或者执行save m n （代表m秒内数据集发生n次修改后就执行bgsave命令）°RDB的优点是 可以灵活设置备份频率和周期，并且相比于AOF机制，RDB的恢复速度更快，尤其在大数 据集恢复的情况下，缺点是一旦在定时持久化之前出现宕机，那么此前时间段的数据都将会 丢失。AOF持久化，以日志形式记录服务器所处理的每一个写以及删除操作，以文本方式记 录，在Redis cli中通过appendonly yes命令开启，在AOF机制中，所有写入的命令会追加到 aof缓冲区中，AOF直接釆用了文本协议的形式，通过配置AOF缓冲区同步文件的策略为 everysec,其含义是命令写入AOF缓冲区后调用系统write操作同步到AOF文件中，write 完成后线程返回，fsync操作并且每隔一秒由专门线程调用对AOF文件进行一次同步。从效 率上看，该模式足够快。当发生故障停机时，只会丢失一秒钟的命令数据。随着AOF文件越 来越大，需要定期对文件进行重写，达到压缩的命令，通过手动调用bgwriteaof命令手动触 发AOF文件重写。因为AOF釆取的机制是每秒同步复制数据，相对RDB模式，其丢失的数 据最多只会丢失一秒。但AOF的缺点是文件过大，恢复速度相比RDB文件更慢。

因此在本系统中针对主节点开启AOF持久化模式，针对从节点开启AOF + RDB模式， 确保Redis集群的数据持久化。

4.6系统测试

本小节主要对基于微服务架构的分布式爬虫系统的功能进行测试，系统的测试是系统能 否正常投入使用的必备步骤，通过设置的测试用例，测试系统是否满足预定的设计目标，是 否存在与初定需求有不满足的情况出现。

4.6.1测试环境

系统的测试环境是根据用户日常正常使用时的环境进行部署，整个测试环境分为软件环 境与硬件环境。软件环境是指系统部署时采用的相关操作系统，数据库类型，浏览器信息等， 硬件环境是指承载系统具体部署场景的服务器配置信息。硬件测试环境如表4. 2所示，软件 测试环境如表4. 3所示：

表4.2硬件测试环境

|  |  |
| --- | --- |
| 操作系统 | Cent OS 6 |
| 内存 | 16GB |
| CPU | Intel Core i5 2.4GHz |
| 硬盘 | 256GB固态 |

表4.3软件测试环境

|  |  |
| --- | --- |
| 软件名称 | 使用版本 |
| Java | 1.8 |
| Redis | 5.0.6 |
| MongoDB | 4.0.3 |
| Spring Cloud | Greenwish.SR3 |
| HttpClient | 4.5.10 |
| Kafka | 2.2.2 |

测试环境釆用了三台服务器作分布式部署，其中MongoDB, Redis, Kafka均以集群形式 部署。

4.6.2系统功能测试

本节硬件测试环境根据表4.2进行了说明，测试软件环境根据表4.3进行了说明。

针对本系统中任务请求分发微服务中的负载均衡和限流策略进行测试，负载均衡和限流 策略是否正常运行直接关系到爬虫系统中接下来的页面抓取微服务和页面解析微服务是否能 正常运行。为了模拟大量用户同时提交爬虫任务的场景，使用了 Apache JMeter测试工具，通 过Apache JMeter I具模拟对接口进行并发访问。通过针对接口进行压测，测试微服务架构下 的自定义负载均衡策略是否成功，测试结果如表4.4所示：

表4.4限流与负载均衡测试

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 并发请求数 | 限流阈值 | 成功通过请求数 | 负载策略 | 微服务运行情况 |
| 300 | 350 | 300 | 无 | 节点1超出承载 |
| 300 | 350 | 300 | 轮询 | 节点运行正常 |
| 300 | 350 | 300 | 最小请求数 | 节点运行正常 |
| 300 | 350 | 300 | 最小请求数 | 节点运行正常 |
| 600 | 500 | 500 | 轮询 | 节点运行正常 |
| 600 | 500 | 500 | 最小请求数 | 节点运行正常 |
| 600 | 500 | 500 | 最小请求数 | 节点运行正常 |

经过测试可以看到，本系统中开发的基于Redis和lua脚本的限流组件，在面对并发请求 数较大时可以有效缓解系统整体承载的压力，将超出阈值的请求拦截，对于通过限流组件的 请求，釆用负载均衡策略将请求分发到不同部署机器，由上表可以看出，在负载均衡策略的 作用下使得各个微服务节点运行正常。

其次对于任务请求预处理微服务中的URL判重策略是防止爬虫对已经爬取过的URL进 行重复爬取，从而减少资源浪费的重要步骤，为了验证去重功能的效果，本实验对比了基于 单独布隆过滤器和釆用Redis Set集合，布隆过滤器的双重判重的误判率，通过计算三次取平 均值得到实验结果，测试结果如表4. 5所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 表4.5 URL判重抑 | | 试 | |
| 去重策略 | 提交URL数量 | 通过判重URL数量 | 误判率 |
| 布隆过滤器 | 10000 | 10000 | 0 |
| 基于Redis和布隆过滤器 | 10000 | 10000 | 0 |
| 布隆过滤器 | 20000 | 19827 | 0.00865 |
| 基于Redis和布隆过滤器 | 20000 | 20000 | 0 |

经过实验可知，釆用基于Redis和布隆过滤器的双重判重机制达到的误判率为0,从而有 效避免了对URL的重复爬取。

针对用户自定义CSS选择器页面数据提取的测试，表4.6对应测试的URL和待提取字段 名称，表4.7为表4.6中对应提取字段的CSS选择器表达式和页面数据提取结果，针对PC端 携程旅行北京详情页以及海外电商网站mercateo进行数据测试：

表4. 6测试URL及提取字段名

|  |  |
| --- | --- |
| 测试URL | 提取字段名 |
| <http://you.trip.com/sight/beijingl/229.html> | 图片链接地址 |
| <http://you.trip.com/sight/beijingl/229.html> | 标签文本 |
| <http://www.mercateo.com/kw/arbeitsoverall/arbeitsoverall.html> | 商品详情页链接地址 |

表4. 7 CSS选择器表达式及数据提取结果

|  |  |
| --- | --- |
| CSS选择器 | 数据提取结果 |
| div.s\_sight\_mapimg | <https://dimg08.c-ctrip.corn/images/100c10000000plzao4FEC>  \_C\_350\_230.jpg |
| #herfyydp 1 | 1725条标签 |
| //div[@class='m5,]/a[@class-fs\_2 hc\_3 c l  plvistedeffect plhovereffect']/@href | <https://ww> w. mercateo. com/p/260S-767951/  1361K054\_Kombi\_Quality\_Dress\_anthrazit\_Gr\_54  .html?ViewName=live~secureMode |

经过实验可知，通过用户自定义CSS选择器，可以实现精确页面数据信息的提取，从而 满足不同用户的自定义数据提取需求。

在相同的硬件条件下对本系统和基于Scrapy的分布式爬虫系统进行了性能测试比对，比 对结果如表4.8所示：

表4. 8基于微服务架构的分布式爬虫与基于Scrapy的爬虫系统性能对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| URL条数 | 本系统抓取成功条数 | 本系统耗时 | Scrapy抓取成功条数 | Scrapy耗时 |
| 100 | 100 | 23s | 100 | 30s |
| 200 | 198 | 50s | 195 | 55s |
| 300 | 293 | 72s | 292 | 83s |

经过实验可知，釆用基于微服务架构的分布式爬虫系统在抓取成功率近乎相同的情况下, 整体耗时要小于釆用scrapy开发的爬虫系统，因此基于微服务架构的分布式爬虫系统单位时 间内可抓取更多的页面数量，这也体现了本系统在性能上的优越性。

**4.7**本章小节

本章节作为本文的重点章节，详细介绍了基于微服务架构的分布式爬虫系统中任务请求 分发微服务，任务请求预处理微服务，页面抓取微服务，页面解析微服务等各个微服务模块 的详细功能与设计。包括任务请求分发微服务的负载均衡算法，限流机制，微服务熔断机制

的设计与实现；请求预处理微服务中的全局唯一分布式ID生成机制，针对URL的去重，作 为消息队列的Ka永a集群配置；页面抓取微服务中基于线程池的多线程下载，爬虫系统的动 态代理池功能实现；页面解析微服务中基于jsoup的HTML数据提取实现，基于Redis的消 费端预防重复消费机制；针对爬虫系统中釆用的Redis数据库的基于哨兵模式的集群方案以 及基于RDB和AOF的混合数据持久化机制的详细介绍，用于存储海量抓取页面HTML纯文 本数据的MongoDB数据库的数据副本与数据分片机制的详细实现。最后通过测试用例测试 了爬虫系统中负载均衡算法策略和限流策略的可用性，验证了整个爬虫系统的可用性。

第五章总结与展望

本章总结全文的主要内容，包括主要研究工作和贡献。同时对未来进一步研究工作做出 简要的规划。

5.1论文工作总结

在本文首先对传统爬虫的优缺点进行分析，得出实现一个基于微服务架构的分布式爬虫 系统的必然性，通过对爬虫相关技术的学习，将分布式爬虫系统拆分成了可视化界面，任务 请求分发微服务，任务请求预处理微服务，页面抓取微服务，页面解析微服务等微服务模块， 底层采用了 Spring Cloud框架，Kafka作为消息中间件，MongoDB和Redis数据库作为数据 存储介质。

论文的主要工作包括如下：

1. 对应的爬虫任务进入至任务请求分发微服务后实现了一套基于令牌桶和Redis lua脚 本的限流算法防止大量请求涌入。
2. 对于正常通过的爬虫请求通过负载均衡算法分发到部署了任务请求预处理微服务的 不同机器上，微服务与微服务之间做了基于Hystrix的熔断机制。
3. 针对分布式场景实现了一套基于基于线性轮询，最小请求数，响应时间权重的的负载 均衡策略，该算法用于将爬虫请求分发至部署了任务请求预处理微服务的不同机器，并对算 法的具体实现策略做了详细阐述。
4. 实现了一套分布式环境下的ID生成算法用于给每个爬虫任务中的数据提取规则生成 一个全局唯一的分布式IDo
5. 对于URL判重釆用了基于布隆过滤器和Redis进行URL的双重判重校验，使用kafka 作为消息队列存储下发数据。
6. 针对本系统的不同存储场景，对MongoDB采用数据副本和数据分片的集群方式保证 MongoDB的高可用和海量数据场景场下的拓展，对Redis数据库釆用基于哨兵模式的集群方 式保证高可用，并基于RDB和AOF机制实现Redis的数据持久化机制。

5.2工作展望

本文目前为止实现了基于微服务架构的分布式网络爬虫的功能，并且验证了系统的功能 能满足一个爬虫系统非要求，但是仍然有进一步优化的空间和必要性。主要考虑在以下几个 方面进行改进：

1. 分布式爬虫系统的可视化页面进一步完善，使其支持更丰富维度的爬虫任务查询， 并且支持用户手动断点续爬。
2. 在己有的三种负载均衡算法的基础上支持更多类型的分布式场景下的负载均衡算法 设计，例如结合机器具体性能的分布式负载调度策略。
3. 除了已实现的反爬虫策略的基础上，面对可能出现的验证码反爬场景，可以利用成 熟的人工智能算法，对验证码进行识别，从而自动绕过反爬，减少人工的干预。

参考文献

1. 赵康.面向主题的网络爬虫系统的设计与实现[D].北京邮电大学,2019.
2. 丁月.基于网络爬虫的垂直搜索引擎设计与实现[D].贵州大学,2019.
3. 李笑语.深度可定制的工具化爬虫系统的设计与实现[D].北京邮电大学,2018.
4. 丁文豪.恶意爬虫主动防御技术研究与实现[D].北京邮电大学,2019.
5. 高萍.具有反爬虫机制的影评系统的设计与实现[D].北京交通大学,2019.

⑹张文强.网络数据釆集技术的研究与应用[D].华北电力大学(北京),2018.

1. 和煦.基于微服务的金融贷后系统的设计与实现[D].哈尔滨工业大学,2019.
2. 段玉风.大数据环境下分布式数据抓取策略的硏究与应用[J].网纟各安全技术与应用,2019(12):75-76.
3. 张金.基于Hadoop平台的网络爬虫技术研究[D].南京邮电大学,2017.
4. 刘星辰.基于Hadoop的分布式网络爬虫的研究与实现[D].西安理工大学,2019.
5. 于营.面向微博的网络爬虫数据采集[J].信息系统工程,2017(12):36-37.
6. 樊宇豪.基于Scrapy的分布式网络爬虫系统设计与实现[D].电子科技大学,2018.
7. 汪兵.基于Scrapy框架的分布式爬虫系统设计与实现[D].合肥工业大学,2019.
8. 孙瑜.基于Scrapy框架的网络爬虫系统的设计与实现[D].北京交通大学,2019.
9. 庄旭菲，田雪.基于Scrapy和Elasticsearch的校园网搜索引擎的研究与实现卩].科技资 讯,2019,17(29):12-15.
10. 赵鹏程.分布式书籍网络爬虫系统的设计与实现[D].西南交通大学,2014.
11. 沈承放漠达隆,beautifulsoup库在网络爬虫中的使用技巧及应用[J].电脑知识与技术,2019,15(28):13-16.
12. 杨君.基于Scrapy技术的数据釆集系统的设计与实现[D].南京邮电大学,2018.
13. 杜凤媛.自定义分布式网络爬虫的设计与实现[D].电子科技大学,2019.
14. 熊益益.基于微服务架构的电商平台的研究与实现[D].北京邮电大学,2019.
15. 钟俊林.基于微服务架构的自助微商城的研究与实现[D].北京邮电大学,2019.
16. 冯丹.基于网络爬虫的搜索引擎的设计与实现[D].湖北工业大学,2018.
17. 刘佳雯.基于分布式爬虫的顶尖学者人才库系统的设计与实现[D].厦门大学,2018.
18. 祁兰.基于MongoDB的数据存储与查询优化技术研究[D].南京邮电大学,2016.
19. 曾胜.基于redis的分布式自动化爬虫的设计与实现[D].华中科技大学,2018.
20. 黄鹏峰.云数据采集系统中云爬虫子系统的设计与实现[D].北京邮电大学,2019.
21. 李聪聪.国际资讯中的中国元素提取系统的设计与实现[D].北京邮电大学,2019.
22. 李文辉.基于网络爬虫互联网涉税信息釆集系统设计与实现[D].电子科技大学,2018.
23. 张峰.基于高实时分布式网络的爬虫软件设计与实现[D].浙江大学,2018.
24. 毛聡.一种分布式前沿科技信息跟踪系统的设计与实现[D].北京邮电大学,2018.
25. Ayoub Mohamed H. Elyasir,Kalaiarasi Sonai Muthu Anbananthen. Foused Web Crawler[J]. 2012 International Conference on Smart Grid Systems(ICSGS 2012),2012,27(11):154-158.
26. 王树梅，尚衍亮.科研论文爬取与多维度分析系统的设计与实现卩/OL].计算机技术

展,2020(05):1-7(2020-01 -1 l].<http://kns.cnki.net/kcms/detail/61.1450.TP.20200110.1118.010.html>.

[33港姗姗.大数据环境下基于python的网络爬虫技术探讨[J].决策探索(中),2019(11):92.

1. J. Edwards- K.Mccurley,J.Tomlin. An Adapt Model for Optimizing Perfbnnance of an Incremental Web Crawler[J],Association for Computing Machinery,2001,26( 1):106-113.
2. 孟慧君.基于Bloom Filter^法的URL去重算法研究及其应用[D].河南大学,2019.
3. 车健.基于布隆过滤器的Parquet优化[D].华中科技大学,2018.
4. 张笑天.分布式爬虫应用中布隆过滤器的硏究[D].沈阳工业大学,2017.
5. Page L. The PageRank citation ranking : Bringing order to the web[J]. Stanford Digital Libraries Working Paper,1998,9(1):1-14.
6. 杨沫.基于MongoDB的多源生物数据集成关键技术研究[D].哈尔滨工业大学,2019.
7. 吴黎兵,柯亚林，何炎祥.分布式网絡爬虫的设计与实现.计算机应用于软件，2011, 28(11).176-179.

附录**1**攻读硕士学位期间申请的专利

[1]葛又嘉，章韵.基于微服务架构的分布式网络爬虫数据提取系统及方法,202010002302.8, 2020.1；

致谢

时光飞逝，研究生阶段的学习生活即将结束。在这三年里，我收获良多。在日常学习和 科研上曾经遇到了很多疑惑和困难，得到了很多人的帮助。在此临近毕业之际，我要向那些 给予我关怀和帮助的人表达真诚的谢意。

首先我要非常感谢我的研究生导师章韵教授。章老师工作认真，辛勤负责，具有精益求 精的工作作风，对待自己的学生无论在生活还是学习都是耐心，平易近人。在我完成论文的 过程中章老师给予了我极大的帮助，指出我论文中的问题，帮助我修改，直至完成一篇优秀 的论文，整个过程让我受益匪浅，在此，谨向章老师表示最真挚的感谢！

其次我要感谢教研室的其他小伙伴们，感谢他们在三年的研究生生活中对我的帮助与鼓 励，曾经一起奋斗的日子将成为我们美好的回忆。

三年的研究生生活，收获良多，感谢计算机院各位老师的培养，也感谢论文评审的老师 百忙之中抽出时间参与我的论文评审与答辩。