

电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

专业学位硕士学位论文

MASTER THESIS FOR PROFESSIONAL DEGREE



论文题目 基于 Node-red 与 Redis 的实时流数据处理模型
的设计与实现

专业学位类别 工 程 硕 士

学 号 201422220209

作 者 姓 名 王江波

指 导 教 师 刘 玓 教 授

分类号 _____ 密级 _____

UDC ^{注 1} _____

学 位 论 文

基于 Node-red 与 Redis 的实时流数据处理模型的设计与实现

(题名和副题名)

王江波

(作者姓名)

指导教师

刘玓

教授

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

软件工程

提交论文日期

论文答辩日期

学位授予单位和日期

电子科技大学

年

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。

The design and application of real-time stream data computing model based on Node-red and Redis

A Master Thesis Submitted to
University of Electronic Science and Technology of China

Major: **Software Engineering**

Author: **Jiangbo Wang**

Supervisor: **Di Liu**

School: **School of Information and Software
Engineering**

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：_____ 日期： 年 月 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：_____ 导师签名 _____

日期： 年 月 日

摘 要

近年来,随着云计算、物联网、社交媒体等新兴信息技术和应用模式的快速发展,人类社会不断地向大数据时代迈进。大数据时代下的流式数据呈现出实时性、突发性、无序性等特点,这对流式数据处理系统就有了更高更严格的要求。如今,现有的实时流数据处理系统通常面临着业务扩展困难、数据流管理困难的问题,本文旨在解决实时流数据处理中所面临的这两大问题,在保证数据处理的实时性和高效性的前提下,提出了一套新的基于 Node-red 的数据流管理和 Redis 内存计算的实时流数据处理模型。

本文从总体架构上对该模型进行设计,重新设计了 Node-red 的数据输入节点、数据输出节点、数据处理节点以及 Redis 数据库访问节点,各个节点的开发使用 Node.js 异步编程语言,节点之间的通信是通过 Redis 的 pub/sub 机制以及 Node.js 的 socket.io 来完成。最后将这些节点重新安装部署到 Node-red 中,使其成为一个完整的实时流数据处理模型。在实时流数据处理过程中,经常会遇到最大值、最小值、累计求和、top(n)等数据指标的计算,而计算这些指标的基础就是去重统计,本文通过分析 Redis 有序集合的源码,结合 Skip List 的基本原理,提出了基于 Redis 有序集合的去重统计方法,并通过新设计的 Redis 数据库访问节点实现该方法在实时流数据处理模型中的应用。

实时流数据处理模型设计完成之后,一个重要任务就是对模型进行应用验证,因此本文设计并实现了一个实时的网站访问监控系统,并利用该模型对数据进行实时处理,最终将分析结果展示在前端可视化界面上。该系统主要包括三个模块,实时数据采集模块、实时数据分析模块以及数据可视化模块,其中,实时数据分析模块是利用本文所设计的实时流数据处理模型来实现的,数据可视化模块是利用 node.js 的 express 框架实现的一个 web 应用,用户只需在浏览器上登录就可以访问监控页面,同时利用 highcharts 将数据可视化用到的图表组件化,以此来适应因业务的不断扩展而带来的数据多样化。本文最后对设计的系统进行了功能测试和性能分析,测试结果均已达到要求。

综上所述,本文完成了从模型的设计到模型的应用的全过程,同时其可行性和有效性在实际的生产线上已经得到了验证。

关键字: 实时流数据处理, Node-red, Redis, 数据可视化, 网站访问监控

ABSTRACT

At present days, the rapid development of new information technique and application modes like cloud computing, Internet of Things and social media, is keeping pushing human society towards the age of Big Data. Streaming data under the era of big data shows real-time, sudden, disorder as well as other characteristics, it then has a higher and stricter requirement for streaming data processing system. Today, the existing real-time streaming data processing system is usually facing with difficulties both in business expansion and data flow management, this thesis aims to solve the two problems in real-time streaming data processing. A new real-time data processing model based on data flow management in Node-red and Redis memory computation is proposed under the premise of ensuring the real-time and high efficiency of data processing.

This thesis designs the model from the overall structure, it redesigns the data input node, data output node, data computing node for Node-red and access node to Redis, the development of each node uses Node.js , which is an asynchronous programming language, and the communication between nodes is completed through pub and sub mechanism of Redis and socket.io of Node.js. Finally, this thesis resigns these nodes to Node-red to make it a complete real-time data stream computing model. In the process of real-time data computing, maximum and minimum values, cumulative sum and Top (n) are often seen, which base on the removing of repetition and then computing. This thesis comes up with the method of Redis based ordered set to remove repetition and recount, by analyzing Redis ordered set source code and combining with Skip List, and implements the application of this method in the real-time stream data processing model through the newly designed Redis database access node.

One important task after the finishing of designing real-time stream data computing model is to apply and testify; therefore this thesis designs and achieves the web-site visiting monitoring system to apply and demonstrate the analyzing results on screens. This system includes three modes: real-time data acquisition module, real-time data analysis module and data visualization module. Among them, the real-time data analysis module is finished using the real-time stream data processing model, data visualization module is a web application with the use of node.js express framework,

what users need to do is simply logging in on the browser to access the monitoring page, while assembling the chart in data visualization module with Highcharts to adapt the diversity of data brought about by the expansion of business. This thesis also provides testing results and function analysis, which all meets the request.

In short, the whole process from designing to the functioning of the real-time data stream computing model is covered in this thesis, feasibility and effectiveness in the actual production line has been verified at the same time its.

Key words: real-time stream data computing, Node-red, Redis, data visualization, web-site access monitoring

目 录

第一章 绪 论	1
1.1 研究背景与意义.....	1
1.2 国内外研究现状.....	3
1.2.1 实时流数据处理模型的研究应用现状	3
1.2.2 Node-red 的研究应用现状	4
1.2.3 Redis 的研究应用现状	4
1.3 论文主要工作和研究内容.....	5
1.4 论文章节结构概述.....	6
第二章 实时流数据处理的基础理论和技术	7
2.1 Node.js 的事件驱动和非阻塞机制	7
2.2 Node-red 可视化流式处理框架.....	10
2.2.1 Node-red 概述	10
2.2.2 Node-red 的编程模型	10
2.2.3 Node-red 的基本配置	12
2.4 基于内存计算的数据库 Redis.....	13
2.4.1 Redis 数据库概述	13
2.4.2 Redis 数据库的实现原理	14
2.4.3 Redis 数据库的 pub 与 sub 机制	16
2.5 本章小结.....	16
第三章 基于 Redis 有序集合的去重统计方法的研究	17
3.1 Skip List 基本原理	17
3.2 Redis 有序集合的源码分析.....	19
3.3 基于有序集合的去重统计方法.....	23
3.4 本章小结.....	25
第四章 实时流数据处理模型的需求分析与设计	26
4.1 需求分析.....	26
4.1.1 实时流数据处理模型的需求分析	26
4.1.2 网站访问监控系统的需求分析	27
4.2 模型的总体架构.....	27
4.3 各数据处理节点的设计	28

4.3.1 数据输入节点的设计	30
4.3.2 数据输出节点的设计	33
4.3.3 数据计算节点的设计	36
4.3.4 Redis 数据库访问节点的设计	38
4.4 节点的重新部署	39
4.5 本章小结	40
第五章 实时流数据处理模型在网站访问监控系统中的应用	41
5.1 网站访问监控系统总体设计	41
5.1.1 实时数据采集方案设计	41
5.1.2 网站访问监控系统的模块层次结构	43
5.2 数据分析模块的设计与实现	44
5.2.1 数据分析模块的总体设计	44
5.2.2 数据库结果集设计	46
5.2.3 数据分析算法的设计与实现	48
5.2.4 在 Node-red 中的数据处理流程	55
5.3 数据可视化模块设计	60
5.3.1 数据可视化模块的功能需求	60
5.3.2 可视化模块的架构设计	60
5.3.3 数据显示方法的设计	63
5.4 本章小结	64
第六章 系统测试与性能分析	65
6.1 测试条件准备	65
6.2 系统功能测试	67
6.3 系统性能分析	70
6.4 本章小结	72
第七章 总结与展望	73
7.1 本文总结	73
7.2 对未来工作的展望	73
致 谢	75
参考文献	76
攻读硕士期间取得的学术成果	79

第一章 绪 论

1.1 研究背景与意义

近年来,随着云计算、物联网、移动互联、社交媒体等信息技术和应用模式的快速发展,人类社会不断地向大数据时代迈进。早在 2010 年,全球的数据量就已经具有 ZB 级的规模,有预测显示,到 2020 年全球的数据量将达到 35ZB,大量数据无时无刻地影响着人们的生活、工作,甚至是社会的发展和国民经济,大数据时代已经到来。而近年来,有关大数据方面的研究和应用也越来越广泛,新形式下的大数据分析技术为我们分析问题和解决问题提供了新的思路和方法,其研究已经成为业界的热点。

大数据的分析计算模式主要分为批量计算(batch computing)、流式计算(stream computing)、交互式计算(interactive computing)、图形计算(graph computing)等等。其中批量计算和流式计算这两种计算模式不管是在学术界还是在工业界都是主要的研究模式,同时各自都有广泛的大数据应用场景。其中批量计算是一种适用于大估摸并行批量处理作业的分布式计算模式,也就是我们大家都十分熟悉的 MapReduce 计算模式^[1,2]。MapReduce 本身是一种编程模型,这种分而治之的编程思想有着广泛的应用,尤其在大规模数据集的并行计算中,由于其简单易用的特点使其成为目前最为流行的大数据并行处理模型。后来,在开源社区的努力下,Hadoop 系统^[3]应运而生,在 Hadoop 系统中包括 HDFS(Hadoop 分布式文件系统)和 MapReduce 两个核心组件,HDFS 用于存储海量的数据,而 MapReduce 是用于海量数据的并行处理。Hadoop 平台的应用也十分广泛,国内外许多企业都在用它来进行大数据处理。此外,Spark 系统^[4]也具备批处理计算的能力。而对于流式数据计算,它是一种对实时性要求极高的计算模式,由于数据的到来是不确定的、无序的、不间断的,为了避免在数据处理过程中造成数据的大量堆积或者数据丢失,这就要求流式计算必须在规定的时间范围内对系统所产生的实时数据完成实时处理。在许多行业的大数据应用系统中,比如金融银行业的业务监控系统、政府政务管理系统、道路监控系统、互联网行业的访问日志处理等,不仅有大量累计的历史数据,同时还具有高流量的实时流式数据,因而在提供批处理计算模式的同时,系统还需要能具备高实时性的流式计算能力。因此,研究和设计一套高效,稳定的流式数据处理模型具有广泛的应用价值,目前也有比较流行的流式计算系统,比如像 Twitter 公司的 Storm、Yahoo 公司的 S4 以及 Apache Spark Streaming^[5]。

在传统的流式计算模型中，绝大多数都是利用传统数据库来实现的，而在大数据时代下的流式计算有了新的需求，表现在低时延、高流量、不确定性等。所以，如何构建一个低时延、高带宽、持续可靠、长期运行的大数据流式计算系统^[6,7]成为了当前亟待解决的问题。Redis 这种基于内存计算的、可进行数据持久化的 Key-Value 存储系统的诞生，为大数据流式计算提供了一个很好的解决方案。Redis 数据库最初是为了解决像 SNS 这类网站在数据存取过程中的实时性等刚性需求来的，而传统的关系型数据库很难完成这项工作，这也使得 Redis 这种数据库也越来越受到人们的关注。如今 Redis 数据库已经得到了广泛的应用，不论是在高速缓存系统中，还是在海量文件的实时检索中，甚至是在如火如荼的各种推荐系统中，Redis 都起着举足轻重的作用。Redis 基于内存的数据计算和高效的数据存储策略也能够很好的满足实时流计算问题中的低时延、高流量的刚性需求。因此，研究 Redis 的内存计算以及存储策略并将其运用到实时流式计算模型中具有重要的研究意义和实用价值。

在流式数据处理中，因为无法确定数据是什么时候到来，按什么顺序到来，因此，不需要事先对流式数据进行存储，而是当流动的数据到来后在内存中直接进行数据的实时计算和分析。就像我们熟悉的 Twitter 的 Storm、Yahoo 的 S4 就是典型的流式数据处理框架^[8,9]，数据在任务拓扑中被计算，最后输出有价值的信息。目前这些流行的流式处理框架都有一个共同的缺点就是，没有一个方便的能够快速根据业务构建数据任务的拓扑计算流程，也就是我们所说的数据计算流（flow），同时也缺乏数据的流化功能。Node-red 是 IBM Emerging Technology 团队创建的一个新型开源工具，基于 Node.js 开发的，支持 Node.js 的事件驱动和非阻塞 IO 机制，它是一种可视化流程编辑框架，程序员可以直接通过浏览器上面的流程编辑器，来完成各个节点的连接，这些节点可以是外部设备、网络服务、API 应用等。Node-red 被广泛用于物联网领域，实现数据的流式传输。在 Node-red 中从数据的接入，到数据的解析分析，最后到结果的输出都是通过各种各样的节点来完成的，IBM Emerging Technology 团队在开发这个工具的时候只引入了少量的具有特殊功能的节点，比如常用的 http 节点、tcp 节点、udp 节点、debug 等数据输入输出节点，还有一些用于数据分析的节点比如 sentiment 节点，也有一些用于访问存储设备的节点，如 mongodb 节点；Node-red 除了原始已经提供的这些节点外，还允许用户自己按照开发原则开发自己需要的节点。为了能够充分利用 Node-red 的可视化流程编辑的直观性，结合 Rredis 数据库的内存计算的特点，探索开发适应于流式数据分析的数据输入节点、数据输出节点、数据处理节点以及 Redis 数据库访问节点，这对流式数据分析有着重要的实际意义。

1.2 国内外研究现状

1.2.1 实时流数据处理模型的研究应用现状

大数据时代下的数据处理主要的两种方式就是实时流数据处理^[10,11]和批量数据处理。实时流数据处理主要适合于那些无需事先进行数据存储，可以直接进行数据分析处理，实时性要求比较严格，但数据的准确度要求比较宽松的应用场景。而对于传统的批量数据处理，首先要进行数据的存储，然后再对存储的静态数据进行集中或者分布式计算。目前，对于传统的批量数据处理模型的技术和研究成果已经相对成熟了，最初有 Google 公司的 MapReduce 并行编程模型^[12]的提出，再有后来在开源社区的努力下开发的 Hadoop 系统为代表的批处理系统，都已经是稳定而高效的批处理系统。而对于流式数据处理模型的研究仅仅处于一个初级阶段，在早期关于流式数据的研究也主要集中在以传统数据库为中心而开展的，主要是研究了数据计算的流式化，数据规模也比较小，数据对象也比较单一，很难适应在大数据时代下流式数据处理所呈现出来的新特性。因为，在新时期的流式数据主要呈现出实时性、突发性、无序性等特点，对新的流式计算系统就有了更高更严格的要求。

在国外，Yahoo 推出了 S4 流式数据处理系统，随后在 2011 年，Twitter 也推出了自己的流式数据处理系统 Storm，还有就是近年来开源社区新兴的 MOA（Massive Online Analysis）、Spark Stream 都是流式处理系统^[13]，这在一定程度上推动了流式数据处理的发展和应用。但是像 S4、Storm 这样的流式数据处理系统在可伸缩性、容错性、数据吞吐量等方面存在着明显的不足，而对于 MOA，Spark Stream 这样的系统，虽然功能和 API 十分丰富，但是在稳定性和易用性上不尽如人意。所以，如果构建一个低延迟、高吞吐量、易用且能持续可靠地运行的流式数据处理系统，是一个亟待解决的问题。

在国内，目前关于流式数据处理模型的研究还比较少，但目前国内主要有百度公司自主研发的 Dstream 和 TM 实时计算平台，在学术界主要是有一些关于流式数据挖掘算法的研究。但是，流式数据的可视化分析已经在很多场景得到了应用，比如各大银行都陆续建立的大屏监控系统，就是实时地监控银行的业务状况、系统运行状况、用户行为分析等，又比如政府网站群的监控，也是通过实时监控网站的访问数据，分析用户的行为。在这些应用的背后，如何建立一个高效、稳定、易于维护的实时处理模型显得尤为重要。

1.2.2 Node-red 的研究应用现状

Node-red 作为一种在物联网时代的新型产物,是一种用来快速搭建物联网应用程序的流式处理框架^[14,15],在信息无处不在的时代,Node-red 也越来越受到业界的关注和研究。

它是由 IBM Emerging Technologies 团队发起的一个开源项目,其中 Nick Leary 和 Dave Conway-Jones 工程师为 Node-red 的设计和开发做出了巨大的贡献。2013 年,Node-red 以开源项目的形式被发布,经过短短几年的发展,Node-red 已经拥有了一大批活跃的用户和开发人员。Node-red 依然是一个新型科技,时至今日,但凡用过 Node-red 的制造商、实验人员和一大批大大小小的公司,都已经见证了 Node-red 极具价值的应用之处。

在国外,IBM 公司率先将 Node-red 物联网领域,以实现各种服务之间的数据传输。Node-red 被集成到 IBM 公司的最新的云产品 Bluemix 上,通过 Bluemix 提供的云服务,用 Node-red 来建立和管理一个应用流程(在 Node-red 中称为一个 flow),实现消息的推送服务。Node-RED 与 Bluemix 中简单的 Push 服务相结合^[16,17],使整个数据处理流程的管理变得十分简单,同时易于维护。

在国内,目前也有很多智能设备制造公司在使用 Node-red,可以很方便地通过 Node-red 节点来控制硬件设备的状态,比如用 Node-RED 搭配 Arduino,是一个快速原型化的好用工具,例如控制 RPI 的某根管脚位去点亮 LED,只要简单的利用四个节点,把他们连接在一起,再写一点数据处理的程序代码即可做到。由于 Node-red 还在进一步完善当中,原始开发的节点可能很难满足实际的需求,所以,我们在运用 Node-red 来管理数据流程的时候,还需要自己开发需要的功能节点。在这一点上,目前在不少银行的业务监控系统中引入了 Redis 数据库的访问节点。

1.2.3 Redis 的研究应用现状

Redis 作为存储系统^[18]之中的后起之秀,由于其数据结构丰富、基于内存计算、支持事务操作又可进行数据持久化等特点,迅速为许多企业和开发者所爱戴。无论是在学术界还是在工业界,对 Redis 的研究都从未停止过。

Redis 是由 Salvatore Sanfilippo 为实时统计系统 LLOOGG 量身定制的一个数据库,在 2009 年的时候将 Redis 开源发布,并开始于另外一位 Redis 代码贡献者 Pieter Noordhuis 一起继续 Redis 的开发,直到现在,Redis 的代码托管在 GitHub 上,并且开发也十分活跃。随着 Redis 内存数据库的发布,经过短短几年的发展,Redis 已经拥有了一大批活跃的用户和开发人员。在国外,像 GitHub、Viacom、Pinterest 等都是 Redis 的用户,Github 利用 Redis 集群^[19],来统计用户项目的跟进状况。而

在国内，新浪在研究了 Redis 数据库的源码后，搭建了有号称史上最大的 Redis 集群，实现了传统的 SQL 数据库难以实现的计数分析（counting）、反向缓存（reverse cache）、top 10 list 等功能。近年来，也有不少银行，在自己的实时数据监控平台引入了 Redis 数据库，实现了数据的实时处理和分析，还有就是随着国家电子政务系统的逐渐推行，不少的地方政府也在自己的数据中心监控系统中引入了 Redis 数据库，来实现数据实时计算和流式处理。

1.3 论文主要工作和研究内容

本文对大数据背景下流式数据处理过程中所遇到的挑战和难题进行了研究分析，详细研究了 Node-red 流式处理框架的编程模型和消息推送机制，Redis 数据库的实现原理及其基于内存计算的原理。设计了一种新的基于 Node-red 的流式管理和 Redis 的内存计算的流式数据处理模型，并通过实际生产线上的网站访问实时监控系統来验证该模型的可行性。主要工作内容如下：

（1）本文首先对当前实时流数据处理模型的研究应用现状以及 Node-red 与 Redis 的研究应用现状进行分析，结合 Node.js 的事件驱动与非阻塞机制详细阐述 Node-red 的消息推送原理，同时详细研究了 Node-red 的编程模型。

（2）对 Redis 数据库做了深入研究。因为在流式数据处理中，经常会遇到关于最大值，最小值，累计求和、top(n)等指标的计算，而去重统计是计算这些指标的基础。因此，本文通过分析 Redis 有序集合的源码，结合 Skip List 的基本原理，提出了基于 Redis 有序集合的去重统计方法。

（3）在研究分析了流式数据的特点和流式数据处理的基本原理后，结合 Node-red 的编程模型和消息推送机制，设计了一种新的基于 Node-red 的流式管理和 Redis 的内存计算的流式数据处理模型。由于原始的 Node-red 缺乏对 Redis 数据库的访问节点以及 Redis 的 pub/sub 节点，重新设计了新的数据输入、输出节点以及数据处理函数节点（function_node），并安装部署到 Node-red 框架当中，实现数据的流式处理和数据流的管理。

（4）本文最后还将设计好的流式数据处理模型，应用到实际生产环境中加以验证。使用该模型对某政府网站的访问流量数据进行实时监控分析，设计了一套实时数据监控系统，该系统包括了数据的实时采集、实时分析和处理，以及最后的数据可视化展示，并对结果进行了有效性分析。实现了从模型设计到模型应用的全过程。

1.4 论文章节结构概述

本论文共分为七章，其章节结构安排如下：

第一章，绪论，首先介绍了本论文的研究背景和意义，通过阅读大量相关文献和论文资料，总结了国内外流式数据处理模型的研究现状，以及 Node-red、Redis 的研究应用现状。然后简单的介绍了本论文的主要研究内容和全文的章节结构安排。

第二章，实时流数据处理的理论基础和技术，本章详细介绍了 Node.js 的异步非阻塞模式与事件驱动机制，这是进行 Node-red 节点开发的理论基础，同时详细介绍了 Node-red 可视化流式处理框架及其编程模型，本章最后还介绍了关于 Redis 数据库的实现原理和 Pub/Sub 机制。

第三章，基于 Redis 有序集合的去重统计方法的研究，分析 Redis 有序集合的底层源码，结合 Skip List 算法提出了 Redis 有序集合在实时流数据处理中的去重统计方法。

第四章，基于 Node-red 和 Redis 的实时流数据处理模型的设计，本章首先对在实际场景中的流式数据处理应用做了详尽的需求分析，然后对模型的总体架构做了详细设计，最后对于 Node-red 中原本缺少的用于流式数据的输入输出节点、数据处理节点以及 Redis 数据库访问节点做了重新设计，并将设计的各个节点重新部署到 Node-red 框架当中，使其成为一个能够胜任流式数据处理的完整模型。

第五章，实时流数据处理模型在网站访问监控系统中的应用，本章主要是对设计的新模型加以应用，以此来验证模型的可行性与高性能。为此设计了一个网站访问的实时监控系統，其中的实时数据分析就用到了本文设计的流式数据处理模型，将数据处理的结果输出到前端页面做可视化展示。本章详细阐述了系统的功能，各个功能模块的设计与实现。

第六章，系统测试与性能分析，这一章是整个模型以及应用系统的测试环节，主要是分析了模型对流式数据的处理能力并对设计的应用系统进行功能测试。

第七章，全文总结与展望，是对本论文的主要工作进行最后总结，并对后续工作做了一些说明。

第二章 实时流数据处理的基础理论和技术

2.1 Node.js 的事件驱动和非阻塞机制

Node.js 从 2009 年诞生至今, 已近经过了八年的发展, 目前 Node.js 已经进入了青年时期, 在各大中小型 IT 企业中的应用十分广泛, 尤其在 web 领域, 不论是前端 JS 还是后端的 web 服务器, 它都有用武之地。Node.js 不仅仅是一种编程语言, 更是一种工具和平台, 为 JavaScript 提供运行环境^[20,21]。它封装了 google 的 V8 引擎, 由于 V8 引擎解释执行 JavaScript 的速度快, 效率高等特点, 再加上 Node.js 本身对其进行了优化, 这使得 Node.js 的性能也非常好。而底层的代码执行模块利用 C++编写, 同时底层通过 libuv 库来实现了事件循环队列的处理, 并将耗时较长的 I/O 请求交给 libev 来处理, 以此来提高运行效率。Node.js 的优秀性能主要体现在其优秀的底层架构上, 图 2-1 所展示的就是 Node.js 的底层架构图。

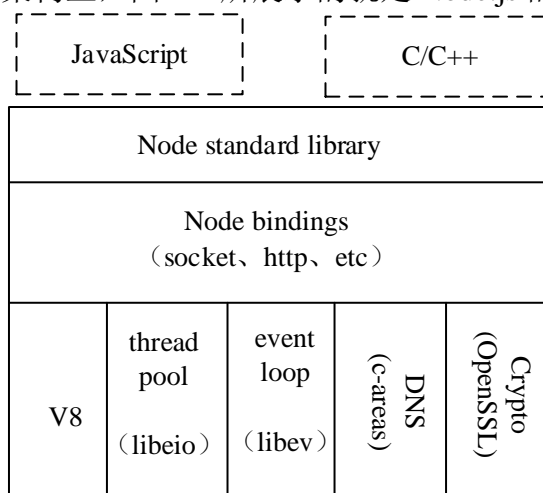


图 2-1 Node.js 的底层架构图

Node.js 底层的事件循环机制是利用 libuv 来实现的^[22], libuv 是一种高性能的事件驱动程序库, 它屏蔽了因为平台不同而带来的差异。在 Windows 平台中, Node.js 是直接利用 Windows 下的 IOCP (I/O Completion Port) 通常称为 I/O 完成端口来实现的, 在 IOCP 的内部其实是利用了线程池的原理, 这些线程是由 Windows 系统内核自动管理, 不需要我们手动加以管理。而在 Linux 平台上, Node.js 都是通过自行实现的线程池来完成异步非阻塞 I/O 的。而 libuv 就是起这样一个平台间的过渡角色, 对外提供统一的 API 接口, 图 2-2 所展示的就是 Node.js 的事件驱动策略。

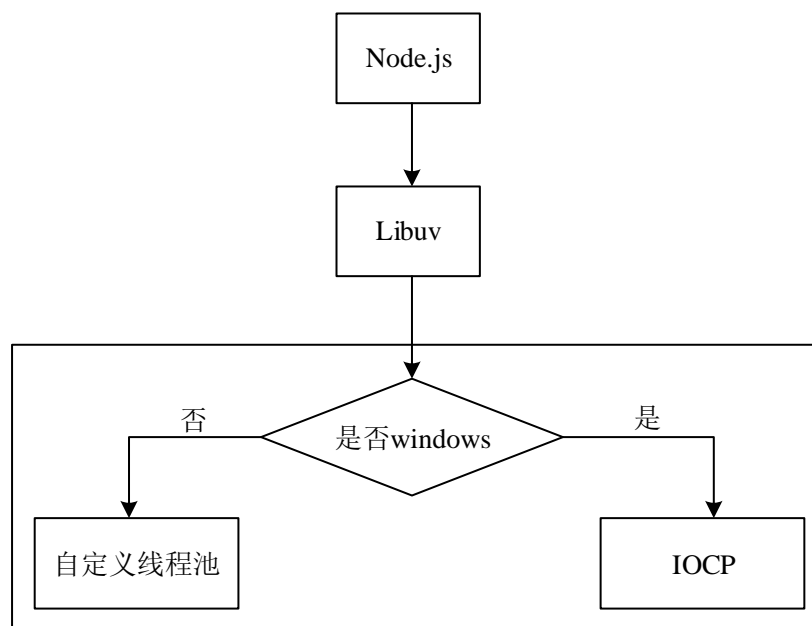


图 2-2 Node.js 的事件驱动策略

Node.js 采用的是事件驱动，异步编程的模式^[23]。事件驱动这个词，对于程序员来说并不陌生，比如在网络套接字编程中，当 socket 有数据到来的时候，就会触发之前所注册的 callback 函数的执行，而 Node.js 所提供的绝大多数 API 都是采用的这种编程模式。下面就来详细阐述一下 Node.js 的这种事件驱动编程模式。

我们可以与 apache 服务器的原理相比较一下，apache 服务器采用的是单进程、多线程模型，一个用户请求对应一个线程，而 Node.js 是单进程、单线程模型，它是通过事件驱动的方式来实现并发的，不会为每一个客户请求创建单独的线程，而是通过事件监听器来判断，最后触发 callback 函数的执行。当 Node.js 的主线程运行的时候，就会创建一个事件队列（event queue），在这个队列中几乎保存了程序所需要的每一个 I/O 操作，由于线程会循环地去处理事件队列中的 I/O 操作，该队列也被称为循环队列。如果在程序的执行过程中，遇到了比如像文件的读写、数据库的查询等 I/O 操作来阻塞任务时，线程不会停下来等待这些操作，而是注册一个 callback 函数，转而继续执行队列中的下一个操作。而这里的 callback 函数，只有在这些阻塞任务执行结束之后通知主线程调用执行。在事件循环队列中，为了避免造成类似于递归调用的无限循环中，要求所有的 callback 函数都必须经过一个 tick 周期，在程序中的具体表现就是所有的 callback 函数都要执行 process.nextTick()。图 2-3 所展示的就是 Node.js 的事件驱动原理图。

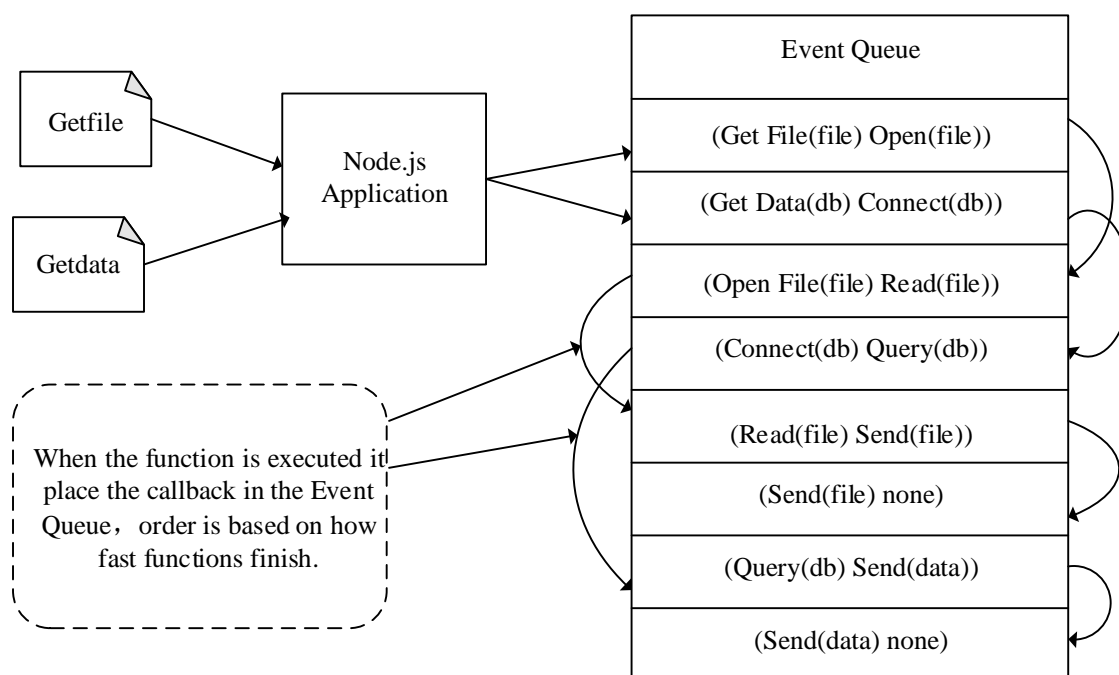


图 2-3 Node.js 的事件驱动原理图

从图 2-3 Node.js 的事件驱动原理图可以看出，在 Node.js 的应用中有读文件和查询数据看两种 I/O 操作，该应用的主线程会创建一个事件循环队列，在这个循环队列中有文件的打开操作、读文件、连接数据库、查询数据库等操作。举一个实际的例子来说明，假如 node.js 在执行下面这样一段代码：

```
db.query('SELECT * from some_table', function(res) {
    res.output();
});
```

程序的第二个参数就是一个回调函数，当程序运行到这里的时候，由于 I/O 操作会消耗大量时间而不会立刻返回查询的结果，而是将该事件插入事件队列中，转而继续执行下面的代码。而当数据库查询操作返回后，就会将该事件发送到事件循环队列中，直到下一次循环监听到了该事件，就会触发回调函数的执行。而只有当整个事件循环队列中的任务都执行结束后，Node.js 应用才会终止。

对于 Node.js 的异步 I/O 非阻塞机制也是建立在事件驱动机制之上的，对堵塞 I/O 的处理，其底层是通过线程池来确保工作的正常执行。Node.js 从线程池中取得一个线程来执行复杂任务，而不必占用主循环线程，这样就防止堵塞 I/O 占用空闲资源而造成效率下降。在堵塞任务执行完毕后，通过查找到事件队列中相应的 callback 函数来处理接下来未完成的工作。也就是说，对于那些相对耗时比较长的 I/O 操作，比如读写文件等，还有一些网络通信，比如套接字，Node.js 会将这些操

作交给一个称之为 `worker threads` 的线程池去执行，当这些操作执行结束后，通过事件通知，并执行回调函数，这就是异步 I/O 非阻塞机制。

正因为 Node.js 的这种事件驱动机制，使得那些十分耗时的 I/O 操作都可以异步执行，有效地解决了因为 I/O 操作而带来的性能和效率瓶颈问题。在许多轻量级、高实时、高流量的应用系统中，都能见到 Node.js 的身影。本文中各个数据处理节点的设计和开发都是基于 Node.js 的，同时前端可视化模块也是利用 Node.js 的 `express` 框架进行开发的。

2.2 Node-red 可视化流式处理框架

2.2.1 Node-red 概述

Node-red 是 IBM Emerging Technologies 团队开发的一个可视化的数据流程编辑工具。它允许程序员直接通过 web 浏览器就可以实现各种数据流程的编辑，同时可以实现对数据处理逻辑的编写。Node-red 把这些数据流程称为 `flow`，所编写的 `flow` 可以以 `json` 对象的形式保存为普通文件中或者形成 `js` 库，方便用户分享、修改。程序员在 Node-red 中可以通过组合各部件来编写应用程序，实现各个部件之间的数据传输，这些部件可以是一些硬件设备(如：Arduino 板子)、网络服务的接口(如：WebSocket in 和 WebSocket out)、功能函数(如：range)或者在线服务(如：twitter)。利用 Node-red 进行编程十分简单，只需要在它提供的 web 界面中，通过拖拽已成功安装部署的节点 (node) 到 workspace 并用线 (Node-red 中称为 wire) 把这些节点连接起来，就可以创建数据流实现编程。最后，程序员通过点击 ‘部署 (Deploy)’ 按钮实现一键保存并执行。Node-red 本身是基于 Node.js 开发的，它的执行模型和 Node.js 一样，也是通过事件驱动和非阻塞 I/O 机制来实现的，这一点在上一节关于 Node.js 的事件驱动和非阻塞机制已经作了详尽的阐述。理论上，Node.js 的所有模块都可以被封装成 Node-red 的一个或几个节点(node)。

本文所设计的实时流数据处理模型是通过利用 Node-red 来完成数据流程的管理以及处理数据的业务代码的编写两项工作。接下来，详细阐述 Node-red 的编程模型以及它是如何管理数据流程。

2.2.2 Node-red 的编程模型

本节我们通过介绍 Node-red 的一些关键概念和关键组件，并通过实际例子说明 Node-red 的编程模型。

(1) 数据流程 (flow)，这是 Node-red 中最重要的一个概念，一个 flow 就

是一个 Node-red 程序,它是多个节点连接在一起进行数据交换的集合。在 Node-red 的底层,一个 flow 通常是由一系列的 JavaScript 对象和各个节点的配置信息组成,通过调用底层的 Node.js 环境来执行 JavaScript 代码。

(2) 节点 (node),它是组成 flow 的最基本的元素,也是真正进行数据处理的载体。当一个编写好的 flow 运行起来的时候,节点主要对从上游节点接收到的消息 (简称 message) 进行处理,并产生新的消息传递给下游节点完成后续的处理工作。一个 Node-red 的节点主要包括 js 文件这 html 文件,分别完成对节点功能的具体实现和节点 UI 设计。

(3) 消息 (message),它是节点之间进行传输的对象,也是数据的真正载体。本质上消息是一个 JavaScript 对象,包含了各种对数据描述的属性。消息是 Node-red 处理的最基本的数据结构,只有在节点被激活时消息才会被处理,再加上节点是相互独立的,这就保证了所有的数据流式互不影响且无状态。

(4) 连线 (wire),它是节点与节点之间的连接桥梁,它们通常将节点的输出端点连接到下游节点的输入端,表示由一个节点生成的消息应该由下一个连接节点处理。

在了解了这些基本的 Node-red 组件之后,下面通过举例说明 Node-red 的编程模型。假设要实时发送一个消息到 debug 节点,来测试消息在节点之间的传输,用到了一个定时器 timestamp 节点,一个函数节点 function_node 以及一个 debug 节点,如图 2-4 所示。

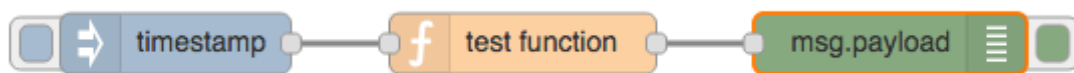


图 2-4 Node-red 中的数据流程图

在图 2-4 所展示的 flow 中 timestamp 节点每隔两秒去触发 test function 节点,执行其中的代码,而 msg.payload 是一个 debug 节点,用于在 debug 面板展示 test function 处理过的数据。这里仅仅是为了说明 Node-red 的编程模型,因此 test function 节点并没有复杂的数据处理逻辑,仅仅是返回一个 hello world 的消息,其实现代码如下:

```
msg.payload="hello world";
return msg;
```

在 function 节点的内部可以编写任何 JavaScript 函数,用于处理上游节点发送过来的数据。

2.2.3 Node-red 的基本配置

由于本文所设计的实时流数据处理模型，要对 Node-red 的数据输入节点，输出节点以及数据处理节点进行重新设计，同时新增 Redis 数据的访问节点，因此，需要对 Node-red 进行源码安装。为了能够有效地利用 Node-red 进行流式数据处理和数据流程的管理，有必要阐述一下 Node-red 的基本配置。经过源码安装后，Node-red 的目录是十分清晰，各个模块的划分也是仅仅有条。首先来了解一下 Node-red 的目录结构，图 2-5 展示了 Node-red 的目录结构。

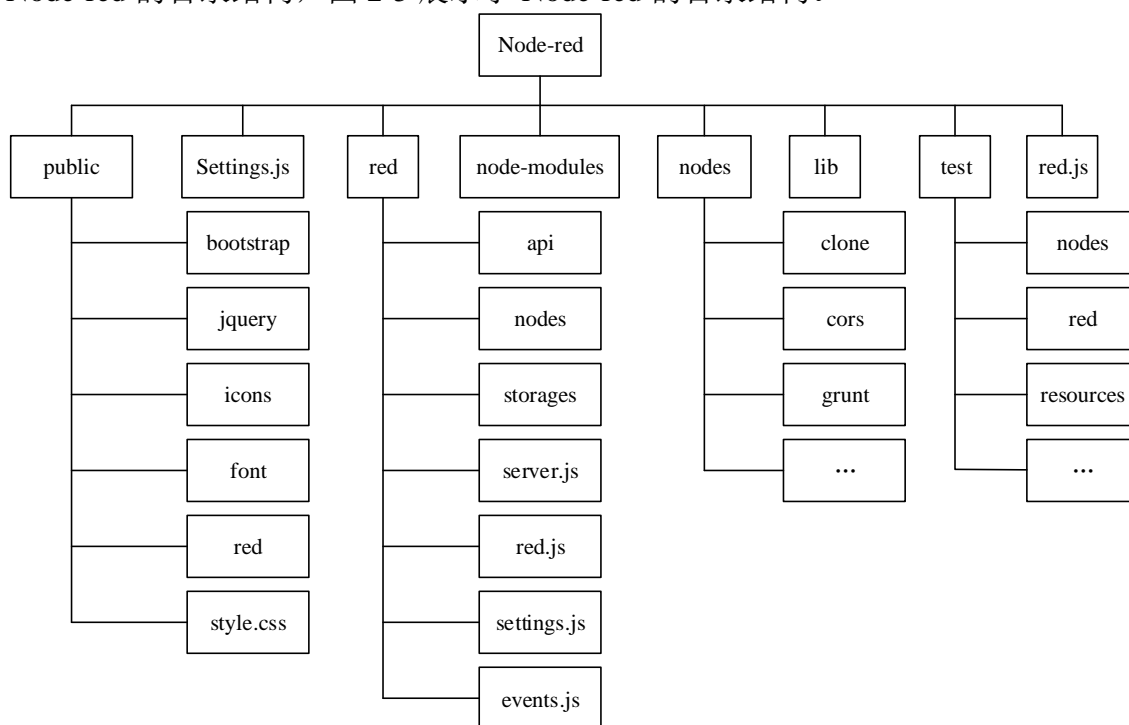


图 2-5 Node-red 目录结构图

下面简单地介绍一下各个目录文件存储的内容和作用：

- (1) 在/public 目录下是一些关于 Node-red 本身的静态文件，包括资源文件、css 样式文件、以及前端页面的 html 文件；
- (2) /node-modules 目录下面是一些外部依赖库，也就是 Node-red 需要的一些 Node.js 模块。
- (3) /red 目录下面就是真正的 Node-red 代码，主要是一些核心 api、事件驱动程序、服务器端主程序、系统设计程序以及 Node-red 的入口程序等。
- (4) /test 目录下面主要是放了一些用于测试的 Node 以及 flow；
- (5) /nodes 目录是一个极其重要的目录，Node-red 中所有的节点都是存放在这个目录下的，包括各个节点的 html 和 js 文件。

(6) settings.js 文件是整个 Node-red 的系统配置文件, 该文件描述了启动的参数细节、端口和 IP 设置以及各个启动目录的设置。

接下来阐述如何配置 Node-red。Node-red 几乎所有的配置信息都记录在 setting.js 文件中, 首先要清楚各个配置选项的功能作用, 表 2-1 展示了 Node-red 的常用配置选项及其作用。

表 2-1 Node-red 常用配置选项说明

选项名	默认值	作用
uiPort	1880	指定 Node-red 网页的端口号
uiHost	127.0.0.1	指定 Node-red 网页的 ip 地址
debugMaxLength	1000	指定 debug 节点调试数据的最大显示长度
flowFilePretty	true	是否保存编写的 flow
userDir	安装目录	指定 flow 保存的位置
functionGlobalContext	undefined	用于加载外部依赖, 其值是 json 对象
...

按照表 2-1 所示的配置选项说明进行配置, 然后重新启动 Node-red, 就可以在浏览器中输入 <http://127.0.0.1:1880>, 即可打开 Node-red 的可视化流程编辑界面。

2.4 基于内存计算的数据库 Redis

2.4.1 Redis 数据库概述

Redis 是一个开源的高性能 key-value 存储系统, 它通过提供多种键值数据类型来适应不同应用场景下的存储需求, 并借助许多高级的接口使其可以胜任如缓存、队列系统等不同的角色。

Redis 和 Memcached 类似^[24], 数据都是缓存在内存的, 而不同之处主要有三点, 第一点就是 Redis 之处存储的数据类型相对来说更加丰富, 比如像 string(字符串)、list(链表)、set(集合)、zset(有序集合)以及 hash(哈希类型)。第二点 Redis 还提供了数据持久化的功能, 因为在内存中的数据有一个典型的问题, 也就是当程序运行结束后, 内存上的数据将会丢失, 所以 Redis 考虑到这点, 提供了对数据持久化的支持, 即将内存中的数据通过异步的方式写入到磁盘当中, 同时也不影响继续提供服务。第三点就是在实现上, Memcached 采用的是多线程技术, 而 Redis 采用的是单线程技术, 所以在多核处理器上, Memcached 的性能和资源利用率上要高于 Redis, 但是针对这一点目前也有很好的解决方案, 再加上 Redis 的性能也

已经足够优秀了，而且提供了许多 Memcached 无法提供的高级功能，我们相信在不久的将来，Redis 将会在很多领域完全替代 memcached。表 2-2 给出了 Redis 与 Memcached 的对比。

表 2-2 Redis 与 Memcached 的对比

	Redis	Memcached
数据库类型	Key-value 内存数据库	Key-value 内存数据库
数据类型	在定义 value 的时候就要固定数据类型	不需要固定，支持字符串，链表，集合，有序集合，hash
虚拟存储	不支持	支持
过期策略	支持	支持
分布式	Magent	Master-slave 主从同步
数据存储安全	不支持	备份到 dump.rdb 中
灾难恢复	不支持	AOF 用于数据容灾

2.4.2 Redis 数据库的实现原理

我们已经知道 Redis 提供了五种数据类型，分别是字符串、链表、集合、有序集合以及哈希表。开发 Redis 数据库的作者，为了让 Redis 支持者五种数据结构，首先对 Redis 的内存模型进行了设计，如图 2-6 所示。

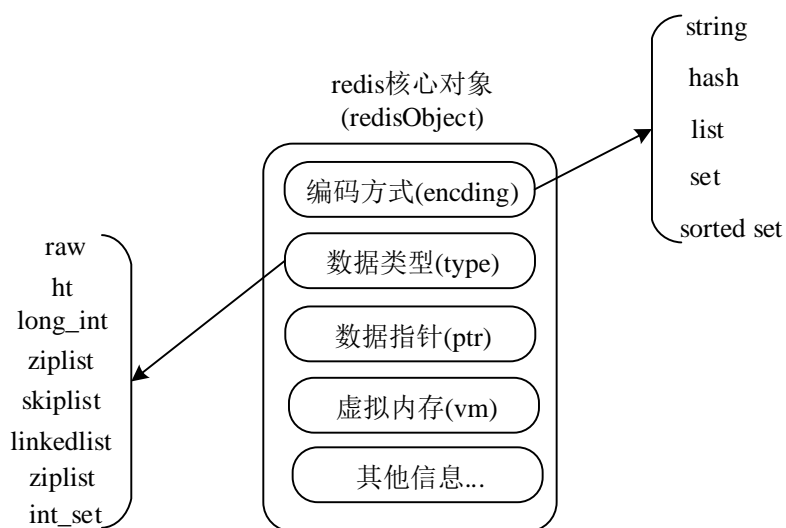


图 2-6 Redis 内存对象模型

从图 2-6 所展示的内存模型可以看出，Redis 是通过一个叫做 redisObject 核心对象来管理这些数据类型的，在 redisObject 对象内部提供了一个 type 字段，用于

表示 `value` 到底是属于那种数据类型，而真正的值是通过一个数据指针来表示的。这里的编码方式也就是表明了该类型的数据在 `Redis` 底层是使用的什么数据结构，比如在列表的底层是通过双端链表实现的，而有序集合是通过 `Skip List` 实现的。另外，模型中还提供了一个虚拟内存字段，而只有在该字段的值为 `true` 的时候，才会真正地分配内存，在默认的情况下该字段值为 `false`。

在认识了 `Redis` 的核心对象后，接下来简单阐述在 `redis` 数据库中这五种数据类型的底层实现原理。`Redis` 数据库底层提供了八种数据结构，在源码中都是通过宏来表示的，也就是之前提到的编码方式，如表 2-3 所示。

表 2-3 `Redis` 底层提供的八种数据结构

编码常量	对应的底层数据结构
<code>REDIS_ENCODING_INT</code>	<code>Long</code> 类型的整数
<code>REDIS_ENCODING_EMBSTR</code>	<code>Embstr</code> 编码的动态字符串
<code>REDIS_ENCODING_RAW</code>	简单的动态字符串
<code>REDIS_ENCODING_HT</code>	字典
<code>REDIS_ENCODING_LINKEDLIST</code>	双端链表
<code>REDIS_ENCODING_ZIPLIST</code>	压缩列表
<code>REDIS_ENCODING_INTSET</code>	整数集合
<code>REDIS_ENCODING_SKIPLIST</code>	跳表和字典

对于字符串来说，其编码方式可以是 `long` 类型的整数，也可以是 `embstr` 编码的动态字符串，还可以是简单的动态字符串。`embstr` 编码的动态字符串是在 `Redis3.0` 中新增的数据结构，它的好处在于只需要进行一次内存分配，而简单的动态字符串需要进行两次内存分配。对字符串 `value` 的操作都是通过核心对象的数据指针进行的。

`Redis` 中列表的底层数据结构可以利用双向链表实现，也可以利用压缩列表来实现，由于双向链表中的每个节点都具有直接前驱和直接后继，因此 `Redis` 的列表支持许多反向操作，但是有一个不足之处就是，每增加一个节点都会向系统申请一次内存，这无疑带来了额外的内存开销。但是对于压缩链表来说，它就要比双向链表更加节省空间，因为压缩链表只需要申请一次内存，而且是一块连续的内存空间，但是为了保证内存的连续性，每次插入一个节点的时候都需要 `realloc` 一次。

集合的内部实现是一个 `hashtable`，只是其中的 `value` 永远是 `null`，这实际上是通过计算 `hash` 的方式来实现快速重排，这也是集合之所以能够快速判断一个元

素是否在集合中的重要原因。Redis 中 hash 类型的底层数据结构可以是 hashtable，也可以是压缩的列表，由于压缩列表是一段连续的内存空间，所以在压缩列表中的哈希对象是按照 key1:value1, key2:value2 这样的先后顺序来存放的，按这种方式实现的 hash，在对象的数量不多且内容不大的情况下，效率是非常高的。

Redis 的有序集合的底层数据结构就是通过跳表来实现的，而没有采用 hash 和 hashtable 来实现，虽然 hash 可以实现快速的查找，但是无法保证有序。关于有序集合的底层实现原理我们将在第三章有序集合的源码分析中做详细阐述。

2.4.3 Redis 数据库的 pub 与 sub 机制

pub/sub 功能即 publish/subscribe 功能^[25,26]。在基于事件的系统中，pub/sub 是目前广泛使用的通信模型，它采用事件作为基本的通信机制，提供大规模系统所要求的松散耦合的交互模式：订阅者比如客户端以事件订阅的方式表达出它有兴趣接收的一个事件或一类事件，发布者比如服务器可以将订阅者兴趣的事件随时通知相关订阅者。

Redis 数据库也支持 pub/sub 机制，本论文所设计的流式数据处理模型中，新引入了数据的输入节点也就是 Redis 的 subscribe 节点，以及数据的输出节点 publish 节点，将这两个节点在采集数据的时候用。订阅者可以订阅多个 Channel，而发布者可以通过 Channel，向订阅者发送消息^[27]。但是发布者所发的消息是异步的，不需要等待订阅者订阅，也不关心订阅者是否订阅，简单地说就是订阅者只能收到发布者后续所发送到该 Channel 上的消息，如果之前发送的消息没有接收，那么也再也接收不到了，下面是 Redis 数据库的发布订阅命令。

```
PUBLISH channel_test message;  
SUBSCRIBE channel_test;
```

PUBLISH: 向 channel_test 发布消息 message。

SUBSCRIBE: 订阅 channel_test 消息，会收到发布者所发送的 message 消息。

2.5 本章小结

本章主要是对本论文的相关技术进行了介绍，首先对 Node.js 的事件驱动和非阻塞机制进行了详细阐述，主要是为后面 Node-red 的节点设计打下理论基础，然后再是对 Node-red 进行了详细介绍，最后还详细介绍了 Redis 数据库的基本概念、底层实现原理以及发布/订阅机制。

第三章 基于 Redis 有序集合的去重统计方法的研究

在实时流数据处理过程中，经常会遇到最大值、最小值、累计求和等指标的计算，而计算这些指标的基础就是去重统计。本文所设计的流式数据处理模型是让 Redis 作为数据计算和中间结果集存储的中心，因此探索研究一种基于 Redis 有序集合的去重统计方法具有重要的研究意义和实用价值。本章将从“跳表”Skip List 的基本原理开始，再到 Redis 有序集合的源码分析，详细研究基于 Redis 有序集合的去重统计方法。

3.1 Skip List 基本原理

Skip List 是由 William Pugh 提出的一种基于并联链表、随机化的数据结构^[28]。Skip List 的查找效率与二叉查找树的查找效率差不多，可以实现平均复杂度为 $O(\log(n))$ 的插入、删除和查找操作。一般而言，跳表是通过在有序的链表的基础上增加额外的链接来实现的，而这种链接方式的增加，并不是随便进行的，而是以随机化的方式增加，同时对随机函数也有要求，必须是以对数函数的方式产生，所以在链表中的查找可以迅速地跳过部分链表，以提高查找效率，跳表也因此得名。众所周知，对于有序链表的查找操作，其时间复杂度为 $O(n)$ ，尽管真正插入与删除节点的操作的复杂度只有 $O(1)$ ，但是，这些操作都需要首先查找到节点的位置，换句话说，是查找拉低了有序链表的整体性能。而 Skip List 采用“空间换时间”的设计思想，除了原始链表外还保存一些“跳跃”的链表，达到加速查找的效果，可以有效解决有序链表查找特定值时效率低的问题。

接下来研究一下 Skip List 实现的原理，首先来认识一下 Skip List。因为，“跳表”是在有序链表的基础上做改进的，所以我们从认识有序链表开始研究 Skip List。图 3-1 展示的就是一个有序链表的数据结构图（这里 H 表示链表头部，T 表示链表尾部，这两个节点都不是有效节点）。

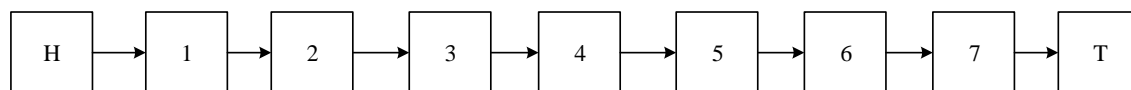


图 3-1 有序链表结构图

现在假设要在该有序链表中查找 value 为 7 的节点，那么，只能在有序链表中一步一步地从头到尾按照 1->2->3...这样的顺序找下去，很明显查找效率是 $O(n)$ 。试想，如果是数组的话，可以利用二分查找，时间复杂度可以提高到 $O(\log(n))$ ，

但是由于链表不支持随机访问，所以不能利用二分法进行查找。但是，如果我们确实想利用二分查找的思想，就可以考虑把中间位置的节点保存下来，重新构成新的顺序链表，经过重构的链表如图 3-2 所示：

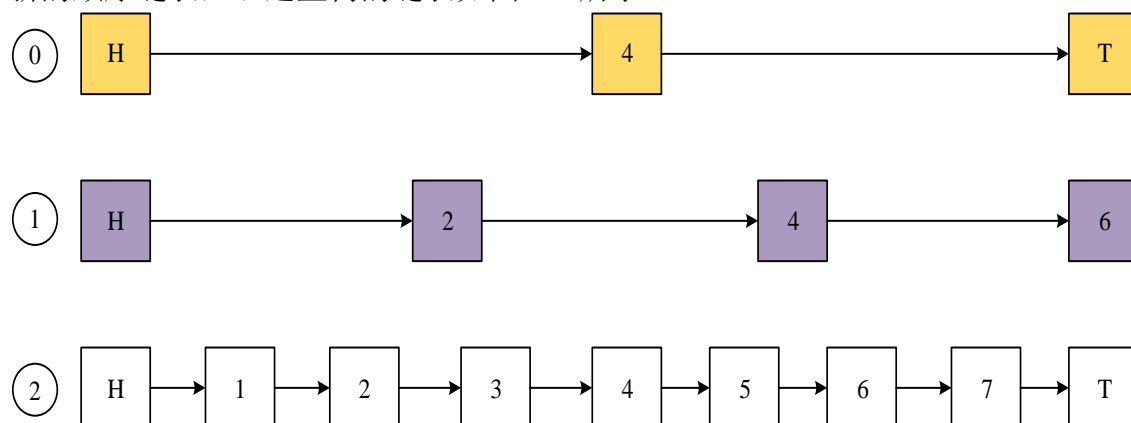


图 3-2 重构后的有序链表

毫无疑问，这是一种典型的以空间换时间的设计思想。原始的顺序链表，经过重构后变成了三个顺序链表，从上到下我们将这三个链表编号为 0、1、2，不难发现，2 号链表就是原始链表，1 号链表就是原始链表的四等分节点构成的，而 0 号链表是原始链表的二等分节点构成的。现在，假设还是要查找 value 为 7 的节点则只需要如下三个步骤：

(1) 初始的搜索范围是(H,T)，在 0 号链表中与 4 进行比较， $7 > 4$ ，将搜索范围更新为(4,T)。

(2) 在 1 号链表中与 6 进行比较， $7 > 6$ ，继续更新搜索范围(6,T)。

(3) 在 2 号链表中与 7 进行比较，结果 $7 = 7$ ，查找成功。

很明显，在 Skip List 中保存了二分查找的信息，以此来提高查找效率。不难发现在具体的实现上，如果要开辟额外的空间来保存新链表的话，会造成空间的极大浪费。由于是链表，可以利用链的性质，通过增加额外的指针，改进存储结构，以达到节省存储空间，降低空间复杂度的目的，经过改进后的 Skip List 的存储结构图 3-3 所示。

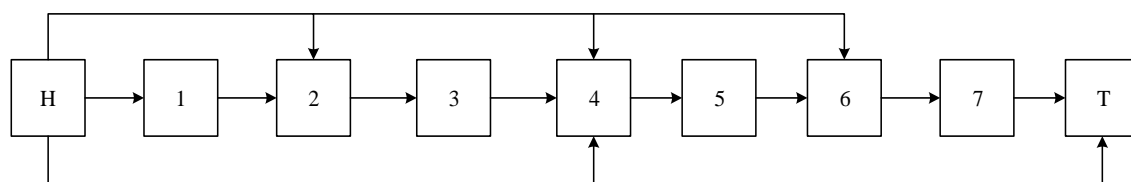


图 3-3 改进后的 Skip List 存储结构

前面所讨论的 Skip List 结构是一种比较理想的结构,仅仅是为了说明 Skip List 的原理,实际的 Skip List 算法是一种随机算法,它非常依赖于所生产的随机函数。当然对随机函数的要求也比较严格,不能简单的按照 $\text{rand() \% MAX_LEVEL}$ 的形式来生成随机数,而是必须要按照满足概率 $P=1/2$ 的几何分布来构造随机函数。可以设计出如下随机函数 `randLevel()`:

```
int SkipList::RandomLevel(void) {  
    int level = 0;  
    while(rand() % 2 && level < MAX_LEVEL - 1)  
        ++level;  
    return level;  
}
```

现在考虑 $\text{MAX_LEVEL}=4$ 的情况,可能的返回值有 0、1、2、3 四种情况,它们各自出现的概率是: $P(0)=1/2$ 、 $P(1)=1/4$ 、 $P(2)=1/8$ 、 $P(3)=1/8$ 。也就是说,如果有 16 个元素的话,第零层预计有 16 个元素,第一层预计有 8 个元素,第二层约有 4 个元素,第三层约有 2 个元素,从下向上每层元素数量大约会减少一半。因此, Skip List 适合自顶向下进行查找,理想情况下,每下降一层搜索的范围就会缩小一半,可以达到二分查找的效率,时间复杂度为 $O(\log(n))$ 。最坏的情况是当前节点从 head 移动到链表的尾部,时间复杂度为 $O(n)$ 。

3.2 Redis 有序集合的源码分析

Redis 的有序集合(zset)的底层数据结构就是通过 Skip List 来实现的,而没有采用 hash 和 hashtable 来实现,虽然 hash 可以实现快速的查找,但是无法保证有序。在了解了 Skip List 的基本原理后,接下来通过分析 Redis 有序集合的源码,详细阐述有序集合的底层实现。Redis 中的有序集合所使用的 Skip List 与 William Pugh 提出的基本一致,只是做了部分改进,主要体现在以下三个方面。

(1) Redis 中的 Skip List 可以有重复的分值 score,这是为了支持有序集合中可能有多个元素具有相同的分值 score 这样的需求。

(2) 在节点进行比较的时候,不仅仅比较它们的 score,同时还要比较它们所关联的元素的 value。

(3) 在 Skip List 中每个节点还有一个前向指针,这就相当于在双向链表中的 prev 指针,通过这个指针,可以从表尾向表头进行遍历。正因为有了这个改进,zset 就支持一些逆向操作命令,比如 `zrevrange`、`zremrangebyscore` 等。

在 Redis 的源码中，有序集合的 Skip List 的节点的数据结构是定义在 redis.h 头文件中，其具体定义如下：

```
/* 跳跃表节点定义 */
typedef struct zskiplistNode {
    robj *obj; // 存放的元素值
    double score; // 节点分值，排序的依据
    struct zskiplistNode *backward; // 后退指针
    struct zskiplistLevel { // 层
        struct zskiplistNode *forward; // 前进指针
        unsigned int span; // 跨越的节点数量
    } level[];
} zskiplistNode;
```

有了节点的定义，那么就应该是 Skip List 的定义了，Skip List 同样也是定义在 redis.h 头文件中的。和定义链表的结构一样，需要头节点、尾节点，他们都是指向 zskiplistNode 的指针，同时还需要定义节点的数量，目前跳表的最大层数。下面就是有序集合的跳表数据结构定义：

```
typedef struct zskiplist {
    struct zskiplistNode *header, *tail;
    unsigned long length;
    int level;
} zskiplist;
```

其实，Redis 的有序集合主要支持的编码方式有两种（所谓的编码方式就是底层的实现方式），一种是 ZIPLIST（压缩列表）方式，另一种是 SKIPLIST（跳表）方式。其中 ZIPLIST 方式可以表示较小的有序集合，而 SKIPLIST 方式可以表示任意大小的有序集合。如果有序集合当前使用的编码方式是 ZIPLIST，只要满足下面两个条件之一就可以转换为 SKIPLIST 编码方式。

（1）当新增加的字符串的长度超过了 server.zset_max_ziplist_value 的时候（默认为 64）。

（2）当 ziplist 中保存的节点数超过了 server.zset_max_ziplist_entries 的时候（默认为 128）。

在有序集合的源码中这两种方式的转换可以通过 `zsetConvert` 函数来完成。这里主要阐述 `SKIPLIST` 编码方式，利用该方式实现的有序集合的数据结构是定义在 `redis.h` 中的，其定义如下：

```
typedef struct zset {
    dict *dict; // 字典，维护元素值和分值的映射关系
    // 按分值对元素值排序序，支持 O(longN)数量级的查找操作
    zskiplist *zsl;
} zset;
```

有了数据结构的定义，接下来就是考虑对这些数据结构的操作了。在 `Redis` 的实现中，将对 `zskiplist` 的操作都放在 `t_zset.c` 源文件中，所支持的操作有三十多种之多。包括创建层数为某一 `level` 的跳表节点、创建一个跳表、释放跳表、向跳表中插入一个节点、删除一个节点等基本操作。下面来看一下有序集合在创建一个空的跳表后是如何向跳表中插入节点的。首先，调用 `zslCreate()` 函数创建并初始化一个空的 `Skip List`，一个空的 `Skip List` 如图 3-4 所示。

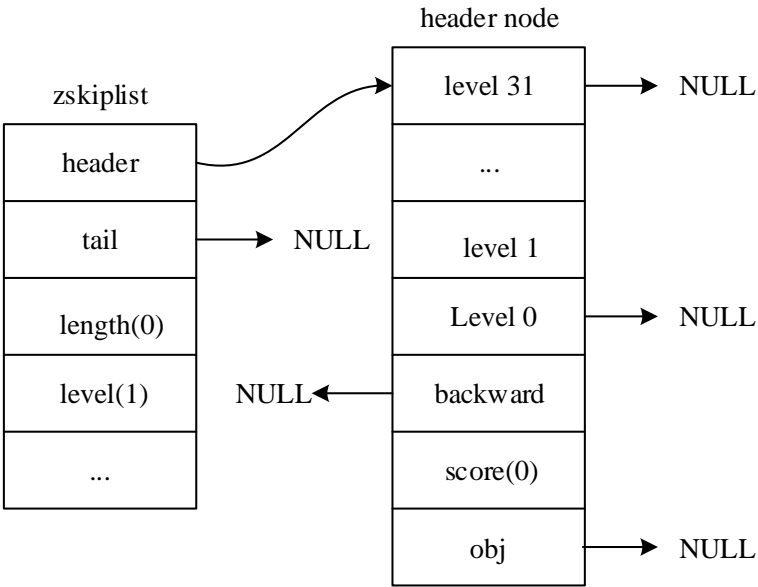


图 3-4 空跳表结构图

在该跳表的结构图中，`level 0` 到 `level 31` 是一个长度为 32 的 `zskiplistLevel` 结构体数组，其大小由 `redis.h` 文件中的宏 `ZSKIPLIST_MAXLEVEL` 定义，值为 32。在 `zskiplistLevel` 结构体中还包括了 `span` 和 `forward` 两个数据成员，这一点从该结构体的定义中可以看出，这里为了展示方便，忽略了 `span`。

创建完跳表之后，调用 `zslInsert()` 函数，就向该空跳表中插入节点。插入一个新的节点的大致过程如下：

- (1) 按照跳表的结构按层数从上向下遍历。
- (2) 在当前 `level` 的当前节点向右遍历，如果发现分值 `score` 相同就比较 `value` 的值，否则进入下一步。
- (3) 调用随机函数，产生随机的层数。
- (4) 比较当前 `level` 与随机函数产生的随机 `level`，记录最大的 `level`，作为下一步遍历的 `level`。
- (5) 插入节点，并更新跨度 `span`

在第三步中调用随机函数，生成随机的层数，这一点在上一小节关于 `Skip List` 的实现原理中已经做了阐述。关于如何查找插入位置，在有序集合的源码中是这样实现的：

```
...
for (i = zsl->level-1; i >= 0; i--) {
    //保存在查找出入位置过程中遇到的节点的序号
    rank[i] = i == (zsl->level-1) ? 0 : rank[i+1];
    //以下是得分相同的情况下，比较 value 的字典排序
    while(x->level[i].forward && (x->level[i].forward->score < score || (x->level[i].forward->score == score && compareStringObjects(x->level[i].forward->obj, obj) < 0))) {
        rank[i] += x->level[i].span;
        x = x->level[i].forward;
    }
    update[i] = x;
}
```

下面举例说明跳表节点的插入操作，假设要向跳表中插入 A、B、C、D 四个节点，它们对应的分值为 3、5、7、9，则对应的跳表结构如图 3-5 所示：

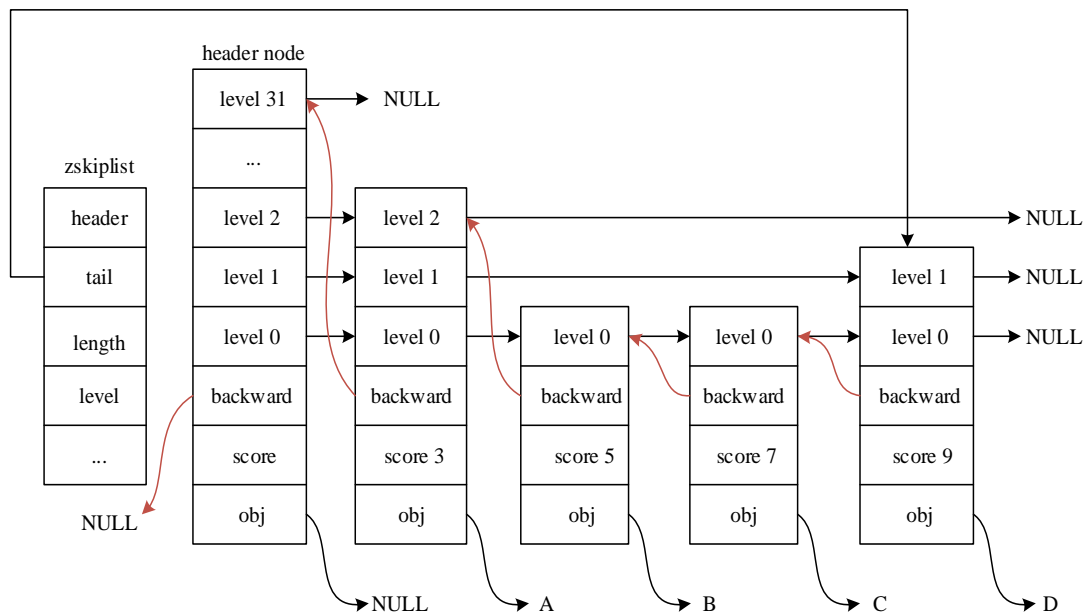


图 3-5 跳表节点插入步骤图

从图中可以看出，跳表中的节点都是按照分值 `score` 来进行排序的。同时，每个节点的 `backward` 指针都指向它的前一个节点，因此，跳表和双向链表类似，支持许多逆向查找，提高了灵活性和操作的效率。

3.3 基于有序集合的去重统计方法

去重统计，在数据分析领域是一个耳熟能详的词语，可以说去重统计在大部分数据处理过程中都要用到。众所周知，在大部分的数据分析的中间计算过程中，最终的数据指标主要呈现以下几种形式：最大、最小、稳定性、叠加、去重统计。在这五种数据指标中，前四种在大部分的实时处理框架和 `nosql` 中都可以使用相对较小的开销就可以完成计算。而对于去重统计，由于去重的数据有可能是多维的，所以不论是 `IO` 效率上，还是计算的效率上都没有前四种标高。

本文所设计的实时流数据处理模型中，也对这五种数据指标的计算做了设计。经过前两节对 `Skip List` 的基本原理和 `Redis` 有序集合的源码分析研究，本文认为不论是在 `I/O` 效率上还是计算查找效率上，利用 `Redis` 的有序结合来做数据去重统计是可行的，能够满足实时计算的要求。在许多流式数据处理的应用中都会涉及到最大值、最小值、累计求和等数据指标的计算，而要计算这些数据指标的基础就是去重统计，因此，涉及一种高效的去重统计方法显得意义也十分重大。

本文所提出的基于有序集合的去重统计方法，就是在流式处理模型中引入 `Redis` 数据库的访问节点，通过这些节点在流式计算的过程中，将产生的中间结果

集存储到 Redis 的有序集合中，并根据上游节点提供的命令格式，对指定的集合进行 `zincrby` 操作。在 Redis 所提供的客户端进行 `zincrby` 操作的命令格式是这样的：`zincrby zsetkey increment member`，如果在名称为 `zsetkey` 的有序集合中已经存在元素 `member`，那么该元素的 `score` 增加 `increment`，否则向该集合中添加该元素，其 `score` 的值为 `increment`，若增加成功返回的是 `member` 增长之后的序列号。也就是说，在 Node-red 中进行去重统计的过程就是通过 `redis_in` 节点对相应集合进行 `zincrby` 操作的过程，关于 `redis_in` 节点的设计思路和具体实现将在第四章作详细阐述。

本文在设计 `redis_in` 节点的时候规定了上游节点传输过来的数据格式，因为 `redis_in` 节点操作数据库的命令就是从上游节点传输过来的数据中获取的。就以实际项目中一个功能来说明这一点，对于某一网站错误页面的统计这个功能，前端页面要求展示错误页面的 URL、错误类型、错误页面所属网站的域名、该错误页面是从哪个页面跳转来的等信息。很显然错误页面具有着四个维度，如果我们单独去统计每一个维度的信息，最后再来进行整合，这样会大大减低计算的效率。为此，我们要将多维统计转换为一维统计，同时也不能影响展示界面要求的四维信息。本文所采取的降低维度的做法是将这四个维度拼接在一起，每个维度之间用特殊字符间隔，这样就形成了只有一个维度的计算指标，然后将这个指标作为有序集合的 `key` 值，当有序集合在进行 `zincrby` 操作的时候，就会根据这个 `key` 来进行插入操作。图 3-6 所展示的就是在 Node-red 中 `redis_in` 节点所要求的数据格式：



图 3-6 `redis_in` 节点要求的数据格式

在图 3-6 所编写的函数中，就用到了 Node-red 的 `function node`，关于该节点的具体设计细节将在第四章作详细阐述，该节点将数据封装在 `msg` 对象的 `payload` 字段中，同时返回 `msg` 对象，而在该节点的内部通过调用 `node.send()` 方法，将 `msg` 对象发送给下一个节点，供下一个节点接收并处理。在图中整个

`['zincrby','errPageDisplay',1,err]`，就是操作 Redis 有序集合的 `zincrby` 命令，其中 `errPageDisplay` 是有序集合的名字，`err` 是通过降低维度后的一维指标。

上面这个例子展示了在实际项目中利用本文所设计的 `redis_in` 节点进行去重统计的过程。之所以说去重统计是一项基础计算，是因为，在进行去重统计的同时，只需要一些简单的操作就可以去计算最大值、最小值、累计求和等计算指标。比如要想知道有序集合中的最大值或最小值，只需要返回集合中的第一个元素或者最后一个元素，有时候需要返回排名前 `N` 的记录，也就是常用的 `top(n)` 操作，在去重统计的基础上也很容易实现。

3.4 本章小结

由于，实时流数据处理中会经常遇到去重统计，而本文所设计的实时流式数据处理模型中引入了 Redis 作为数据计算中心，基于 Redis 有序集合的去重统计方法也被应用到该模型中。本章主要是对 Redis 的有序集合底层实现原理进行分析研究，同时研究分析了有序集合的底层实现源码，最后也阐述了基于有序集合的去重统计方法在该模型中的具体应用。

第四章 实时流数据处理模型的需求分析与设计

第三章对基于 Redis 有序集合的实现原理及其源码进行了分析研究，并提出了在流式数据处理模型中利用 Redis 有序集合来进行去重统计的方法。本章将从需求分析开始，再到模型的总体架构设计，最后是各数据处理节点的详细设计，从这三方面对该实时流数据处理模型进行详细阐述。

4.1 需求分析

4.1.1 实时流数据处理模型的需求分析

本论文主要研究并设计了一种基于 Node-red 与 Redis 的实时流数据处理模型，应用场景为实际项目中的网站群实时访问监控。本项目旨在实时了解用户访问网站群的行为^[29]，捕捉用户请求并跟踪其所有响应，收集、处理并显示用户行为的细节数据，并可视化展示数据和挖掘数据背后的信息。针对该实时流计算模型在实际应用场景下的应用提出如下的需求。

(1) 高实时性：在许多实时流数据处理的应用场景中，不论是数据的采集，还是数据的处理，都要求具有高实时性。高实时性，要求模型在进行数据采集的时候满足不低于每秒钟 50 笔的采集速度，以免造成数据堆积，同时也要求具备高效的数据计算和处理能力。

(2) 高性能：随着业务的不断扩展，数据量也不断的增大，对实时流数据处理模型及应用系统的性能要求也越来越严格。因此，从数据采集到数据处理再到数据可视化展示，各个环节都要求系统具有良好的性能。最直观的表现就是在用户看到的可视化模块的数据更新延迟不能超过 2 秒钟。

(3) 高可用：要求模型可以通过集群等方式实现分布式部署，避免单点故障。

(4) 可扩展：数据量、计算量会随着业务的不断扩展而不断增大，这就要求模型需要有良好的扩展性。

(5) 分布式：为了提高数据的处理能力和计算效率，模型还需要具备分布式的处理能力。

(6) 安全性：数据安全是任何系统的一个首要前提，流式数据处理模型也必须保证数据的安全性。

本论文在这些需求的基础之上，提出一种新的实时流数据处理模型，要在 Node-red 上设计出高效的数据接入和输出节点，同时也要有高效的数据处理节点。

结合 Redis 的内存计算的优势,设计出对 Redis 数据库访问操作节点,用于对中间结果集进行统计计算,以提高模型数据计算的效率。同时,充分利用 Redis 的 pub/sub 机制来实现数据的流式异步传输,最终将这套模型应用到实际应用系统中去加以验证。

4.1.2 网站访问监控系统的需求分析

本文所设计的应用系统是通过实时采集网站群的访问流量^[30,31,32],利用本文新设计的基于 Node-red 与 Redis 的实时流式数据处理模型来解析处理实时数据,并从中挖掘出用户关心的有价值的信息,最后将分析出来的数据可视化地展示到前端界面。该系统主要包括以下几个功能:

1.用户行为监控^[33]

实时了解用户访问网站群的行为,捕捉用户请求并跟踪其所有响应,收集、处理并显示用户行为的细节数据。

具体实现以下功能:

- (1) 用户终端类型统计,对用户访问网站群的终端进行统计。
- (2) 受访页面统计,对用户访问网站所浏览的页面进行统计。
- (3) 来路页面,统计用户是通过哪个页面跳转到所浏览网站。
- (4) 地区分布,根据用户 IP 统计访问网站群的地区分布,并区分内外网用户(内网 IP 地址范围及相关部门的对照表需信息中心提供)。
- (5) IP/PV,一天之内独立 IP 数,相同 IP 数被计数一次。
- (6) 重复访问率,同一 IP,在同一天内访问同一页面的访问量/总访问量。

2.网站群页面监控

- (1) 错误页面跟踪,对返回码为 404,500 等出错页面进行统计跟踪。
- (2) 关键词搜索频率,用户搜索关键词的频率。
- (3) 二级域名访问统计(需信息中心提供二级域名对照表)。
- (4) 频道访问统计(需信息中心提供频道名称对照表)。
- (5) 热点页面统计。

4.2 模型的总体架构

基于 Node-red 与 Redis 的实时流式数据处理模型是搭建在 Ubuntu 环境下的,也可以部署在分布式环境上以提高流式数据的处理能力和计算效率。该模型通过重新设计数据输入、输出以及数据计算节点,以完成对实时流式数据的处理工作。整个模型的架构如图 4-1 所示:

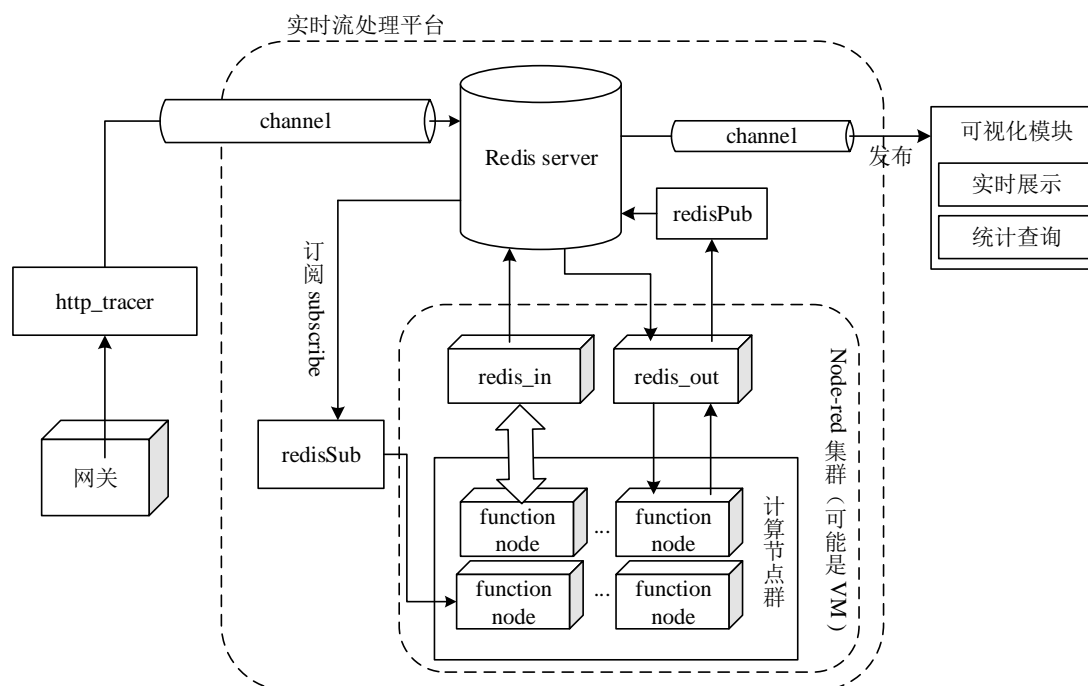


图 4-1 流数据处理模型架构图

从该模型的架构图中可以看出，Redis 数据库充当了数据交换的中心，而整个数据流的处理逻辑都交给计算节点群去完成。数据首先通过 Redis 的 channel（通道）进入 Redis server，然后 Node-red 利用 redisSub 节点去订阅相应通道（channel）的数据，交给计算节点（function nodes）集群进行数据计算，而计算节点集群所产生的中间结果集，通过 redis_in 节点传给 Redis server 进行统计，这些中间结果集在 Redis server 中完成计算后，还会通过 redis_out 节点取出进行进一步的数据封装，将其封装成前端可视化模块需要的数据格式，最后产生的最终计算结果通过 redisPub 节点发布到指定的 Redis 通道中，前端可视化模块再从指定通道去订阅数据做可视化展示。

在原始的 Node-red 中是没有任何节点可以与 Redis 进行交互，为此，新增加了 redisSub、redisPub、redis_in 和 redis_out 节点。为了用户可以自定义数据的处理逻辑，引入了函数节点，多个函数节点构成了整个流式计算的计算节点群。有了这些节点，就可以方便快捷地在 Node-red 上编写流式数据处理的业务代码，更为重要的是，这些业务代码可以实现一次编写多次使用，方便移植和维护。

4.3 各数据处理节点的设计

节点是 Node-red 的重要组成元素，所有的数据流（在 Node-red 中简称 flow）都是通过一个一个的节点组成的，在 Node-red 中有三类基本的节点，数据输入节

点、输出节点以及数据处理节点。为了设计出适合流式数据处理的节点，这里必须对这三类节点进行重新设计，在这一节中主要是对整个流式数据处理模型所需要的节点给出详细的设计方案。Node-red 的节点本身主要包括两份文件：js 文件和 html 文件，js 文件主要定义了节点具体做些什么事情，有什么样的功能；html 文件主要定义了节点的属性，节点编辑框格式和帮助信息等，图 4-2 所展示的就是 Node-red 中一个节点的设计方案：

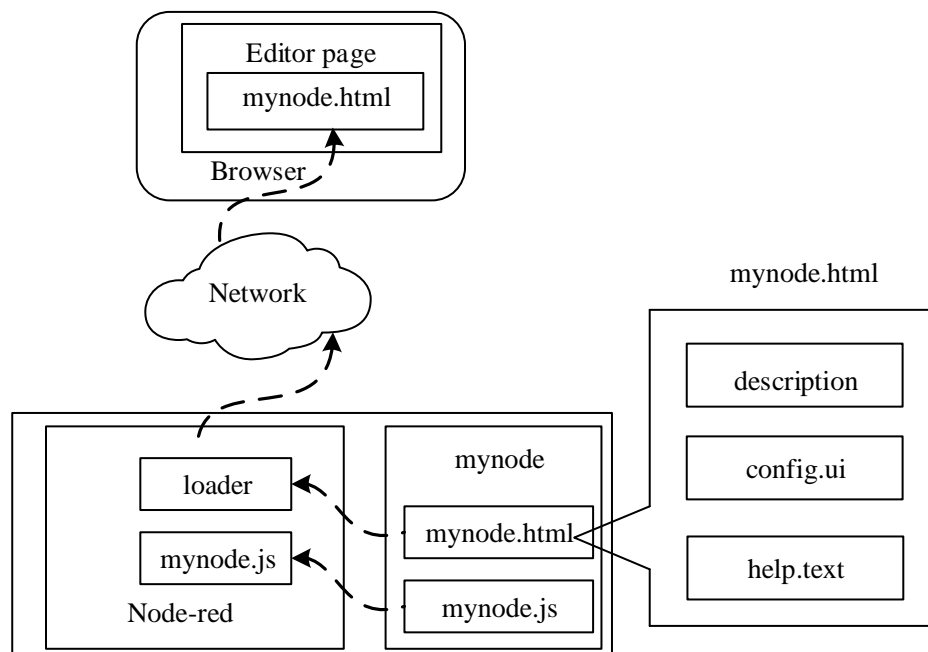


图 4-2 Node-red 的节点设计方案图

将设计好的新节点重新安装部署到 Node-red 中，就可以在 Node-red 的前端编辑界面使用该节点进行数据处理。Node-red 强大的扩展能力就体现在用户可以设计 Node-red 没有提供的节点，来完成特定的任务。为了保证节点设计的正确性和有效性，在节点设计的时候必须按照如下原则来进行：

（1）要求创建的节点要对各种类型的输入数据进行必要的处理，即使某些类型并不是这个节点所需要的。这样做有两个目的，一是为了便于对原始数据进行追加额外说明信息，二是为了便于节点的扩展。

（2）由于 Node-Red 在识别和处理节点的时候使用了大量的字符串匹配操作，所以在节点的定义中有一些名字的字符串必须和节点文件名保持一致，否则 Node-Red 在解析的时候就会出错。

（3）html 文件分为 3 部分：节点的定义、节点的编辑模板以及节点的帮助信息。节点的定义主要用于确定节点的类型，可编辑的属性，在浏览器中显示的样式，是一段可执行的 js 代码，节点的编辑模板主要是用于生成用户编辑该节点的

实例时的界面，用户的输入最终会保存在 `node` 的定义中，节点的帮助信息是提供一些对节点的使用说明。

(4) 在 `html` 文件中，`data-template-name`、`node-input-`、`data-help-name` 都是 Node-Red 系统保留字。`data-template-name`、`data-help-name` 的取值必须和文件名字的名称部分一致。`RED.nodes.registerType` 的第一个参数也必须和文件名字的名称部分一致，否则 Node-red 解析节点会出错。

(5) 每个节点的可编辑域在 `defaults` 中声明，`data-template-name` 所包含的 `node-input-` 负责生成输入框，`defaults` 的每个域的名字必须和 `node-input-` 中的名字保持一致。在 `.js` 文件中使用可编辑域的值的时候，直接访问 `defaults` 的域就可以，不必添加 `defaults` 前缀。

(6) 在 `.js` 文件中，`RED.nodes.registerType` 是用来注册一个 `node` 实例的生成函数，它的第一个参数必须是文件名字的名称部分一致，传给生成函数的参数是 `node` 可编辑域的值及节点共享域的值。

(7) `input` 事件的回调函数 (`callback`) 是节点输入的处理函数。需要注意的是，Node-Red 节点之间数据传输使用的是名字为 `payload` 的域，这个也是 Node-Red 系统保留的。

4.3.1 数据输入节点的设计

数据的输入节点 (`input node`)，主要是用于从外部设备或者其他外部接口获取数据到 Node-red 中进行数据分析。在 Node-red 的一个 `flow` 中，输入节点是所有 `message` 的入口，为下一个 `Node` 产生新的 `message`。由于 Node-red 自带的输入节点很有限，而且不适合流式数据的输入，所以在这里必须补充设计数据的输入节点。为了满足流式数据的输入需求，数据的输入节点的设计必须要满足以下几个原则：

(1) 流式化数据，为了让成批到达的数据也能够在这一个模型中得到计算，我们在设计数据输入节点的时候就要考虑到这点，也就是说让批量到达的数据逐条进入 Node-red 的 `flow` 中。

(2) 统一的数据格式，在一个数据处理模型中，数据格式的好与坏意味着后续进行数据计算的简与繁。

(3) 高吞吐量，由于流式数据的产生是源源不断的，所以在设计输入节点的时候要充分考虑节点的数据吞吐量问题，不然会造成大量数据的堆积，从而影响后续的数据分析与计算。

(4) 高稳定性，输入节点是数据的入口，稳定性是必须考虑的一个因素。

(5) 可移植性, 为了能够将自己设计的数据输入节点共享给其他用户, 节点的可移植性也十分重要。

为了设计出高效的适合流式数据传输的输入节点, 考虑到流式数据的特点, 结合 Redis 数据库的 sub 机制, 可以为 Node-red 新增一个 redisSub 节点。从上一小节的总体架构图中我们可以看出, 我们尽量让所有的数据通过 Redis 的发布订阅机制来进行收集, 把采集到的数据按类别放到不同的 Redis 通道 (channel) 中, 避免数据间的相互影响, 然后在 Node-red 中通过我们新增加的 redisSub 节点去订阅相应通道的数据, 这样就可以把数据引入到 Node-red 中, 完成了数据的接入工作。同样 redisSub 节点也包括两个文件, 一个是编写具体功能的实现代码的文件 js 文件, 另一个是用于界面设计和帮助文档描述的 html 文件。由于 Node-red 原始节点的存在, 所以在进行文件命名标号的时候从 52 号开始, 因为文件名编号和节点的 ID 是紧密相关的, 所以节点的标号必须唯一。设计好新的节点后需要重新安装部署新节点到 Node-red 中, 在利用 npm 安装的时候, Node-red 的节点注册模块会去检测 setting.js 配置文件, 依次加载配置文件中的其他外部模块。图 4-3 是整个 redisSub 节点的设计图。

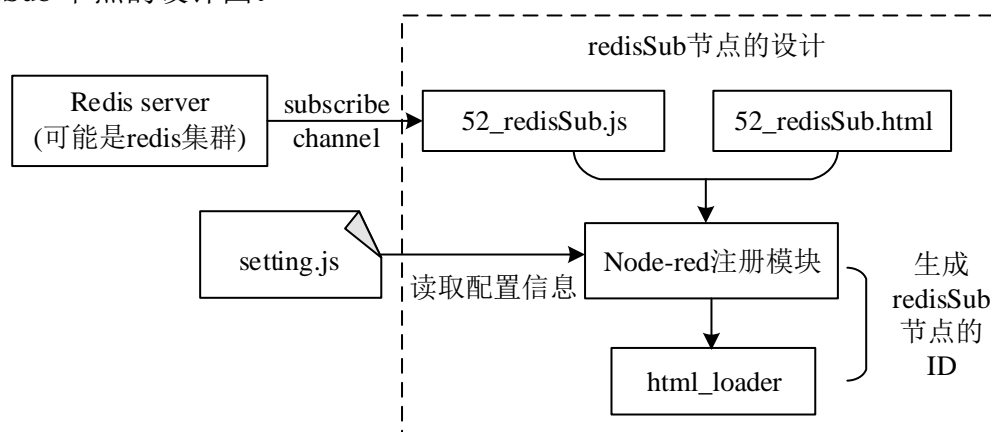


图 4-3 redisSub 节点设计图

对于 redisSub 节点的界面 ui 设计, 需要考虑有哪些信息需要用户输入该节点。因为每个节点都有自己的名字, 所以首先需要的一个信息就是用户为该节点取一个名字, 需要用户输入 Name 字段。由于数据是存放在 Redis server 上的, 所以还需要 redisSub 节点记录 Redis server 的 IP 地址和端口号。当 redisSub 节点连接上 Redis server 后, 不知道数据是位于 Redis 的哪一个通道上, 因此还必须给出通道名称, 这些都是 redisSub 节点所需要的最基本的信息。另外还有就是 redisSub 节点的帮助信息也必须给出一定的说明。ui 界面主要是定义在 52_redisSub.html 文件中。

而对于 `redisSub` 节点的具体功能，是在 `52_redisSub.js` 文件中实现的。首先，要调用 Node-red 提供的节点创建函数 `createNode()` 创建一个节点，并把配置信息告诉节点。节点接收到这些信息后，创建一个数据库连接池函数 `redisConnectionPool`，将 Redis server 的 IP 和 port，以及在 `createNode` 函数内部所产生的 uuid 传递给连接池函数。数据库连接池函数主要是通过一个 `connections` 数组的 `_nodeCount` 来记录有多少 `redisSub` 节点连接 Redis server，当有一个新节点连接 Redis 时，该值就会加一，同样当有一个节点断开的时候就会减一。当有 `close` 请求到的时候首先要判断 `_nodeCount` 的值是否为 0，来决定是否删除 `connections` 对象数组。关于 Redis 数据库连接池函数的执行流程如图 4-4 所示：

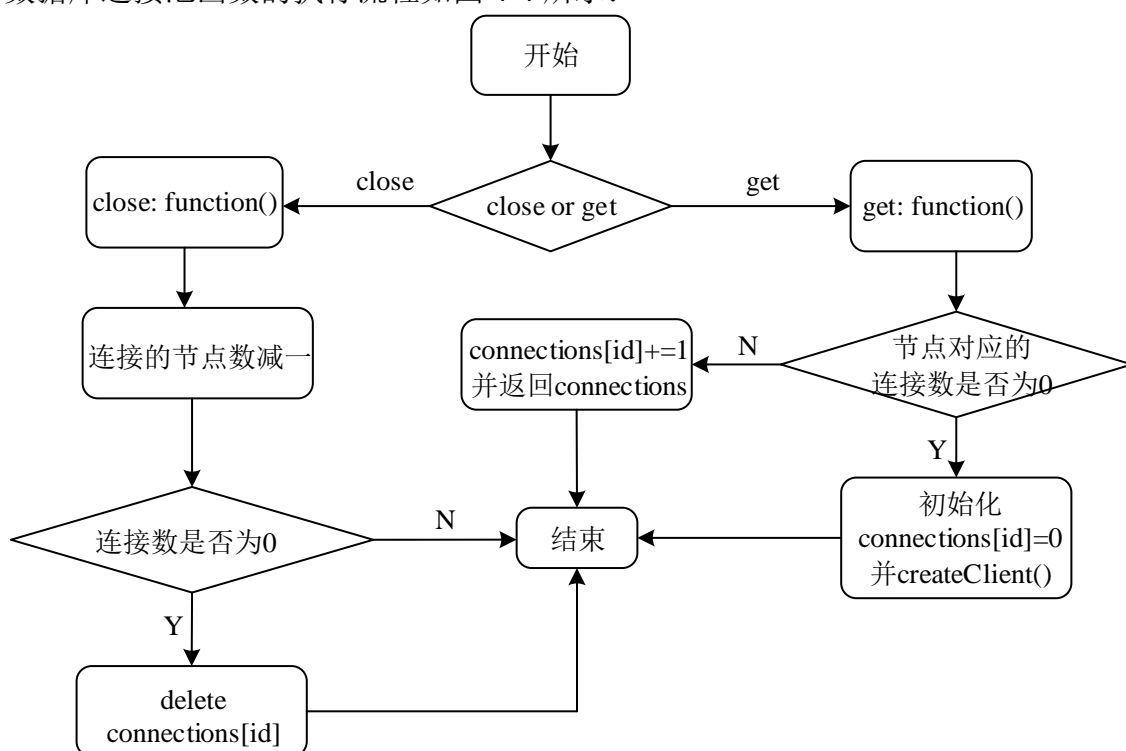


图 4-4 Redis 数据库连接池函数执行流程

有了数据库连接池函数，就可以来实现 `redisSub` 的功能了。`redisSub` 节点的前端页面将用户输入的 Redis server 的信息保存起来，然后通过参数传给连接池函数连接 Redis server。连接数据库后调用 `client.subscribe()` 方法去订阅指定的通道，如果订阅成功，就让 `client` 去监听一个 `message` 事件，看通道是否有数据发送过来，如果有数据就封装在 `msg.payload` 中，让 `node` 的 `send()` 方法发送出来，供下游节点接收。与此同时，`client` 还要去监听 Redis 的 `close` 事件，当 `redisSub` 节点断开与 Redis server 的连接的时候，就要调用 `redisConnectionPool.close()` 方法去断开连接。

下面就是 `redisSub` 的功能函数的伪代码：

```
function redisSub(config) {  
    RED.nodes.createNode(this,config); //创建节点  
    //将 html 文件中定义的节点属性保存下来。  
    this.channel = config.channel;  
    ...  
    //为节点生成一个唯一的 id  
    var uuid1 = uuid.v4();  
    this.client = redisConnectionPool.get(this.host,this.port,uuid1);  
    ...  
    this.client.subscribe( this.channel);  
    //监听相应通道的 message 时间，将通道发送过来的数据封装到  
    msg.payload 中，并通过 node.send()方法发送出来。  
    this.client.on("message", function (channel, message) {  
        var msg = {};  
        msg.payload = message;  
        node.send(msg);  
    });  
    this.on("close", function() {  
        redisConnectionPool.close(node.client);  
    });  
}
```

4.3.2 数据输出节点的设计

数据进入 Node-red 后，经过各个计算节点的数据计算、封装等工作，然后打包成系统规定的格式后，需要从 Node-red 中输出，进入后续的数据可视化展示。数据的输出就用到了 Node-red 的输出节点，Node-red 的输出节点允许把数据输出到 Node-red 的 flow 以外的其他服务和应用上去，对内有一个数据输入的左断点，对外暴露一个公共接口。

在 Node-red 中有一个常用的输出节点就是 debug 节点，这个节点是在编写 flow 的时候用于调试数据处理流程，主要显示经过上一节点处理之后数据的具体信息。debug 节点是一个具有开关的节点，允许程序员手动开启或者禁用该节点。debug 节点的使用也非常简单，只需要在 Node-red 左侧的节点栏中找到该节点然后拖拽

到相应节点的后面，并用线连接起来就可以实现数据的传输，最后开启 debug 的启动按钮，将所编写的 flow 成功部署后，就可以在 Node-red 的最右侧的 debug 面板中看到打印出来的具体数据。值得注意的是，debug 节点的只有一个数据的入口，而没有数据的输出端，在设计 debug 的时候，重新封装了 sendDebug()函数，用来发送消息，将消息直接发送到 Node-red 的网页编辑器 debug 视图上直接显示，而不是交由下游节点做数据处理。下面给出 debug 节点的设计逻辑的部分伪代码。

```
function DebugNode(n) {
  ...//创建节点并定义 complete 属性，用来判断数据封装是否完成。
  this.on("input",function(msg) {
    if (this.complete === "true") {
      ...//debug 节点完成了 msg 的接收进行封装
    } else {
      ...//debug 的用户需要自己定义 msg 的属性
      var output = msg[property];
      if (this.complete !== "false" && typeof this.complete !== "undefined") {
        output = propertyParts.reduce(function (obj, i) ;
          ...
        }
      }
      if (this.active)
        sendDebug({output});//调用 sendDebug 方法发送 output 数据
    }
  });
};
```

有了 debug 节点，可以方便用户在编写自己的 flow 的时候，及时查看数据的处理情况。本文在第五章中应用该模型来解决实际问题的时候，将大量应用到 debug 节点。

为了保证数据实时地输出到 Node-red 的 flow 以外的其他服务和应用上，这里我们新引入了 redisPub 节点。顾名思义，redisPub 节点就是将 Redis 的 publish 功能嵌入到 Node-red 中，通过设计一个新的节点来将经过 Node-red 处理和计算过的数据输出来，这里之所以选择 Redis 的 publis 发布数据，一方面保证了数据的异步传输，另一方面也保证了数据的隔离（原因是各个 Redis 的通道数据是相互隔离的，互补干预）。在坚持节点的设计原则的前提下，下面给出 redisPub 节点的设计方案，如图 4-5 所示。

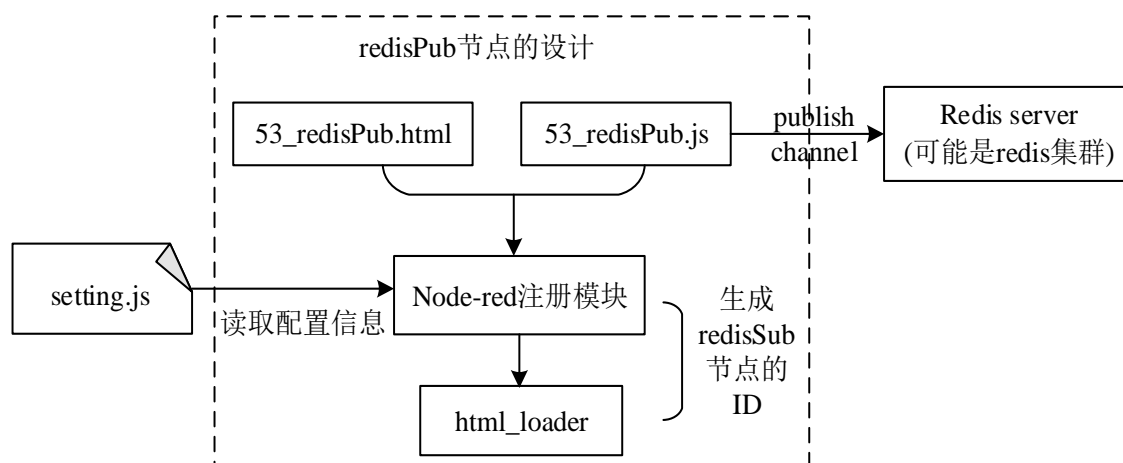


图 4-5 redisPub 节点设计图

结合上一小节数据输入节点的设计可知，redisPub 节点和 redisSub 节点的设计恰好相反，redisPub 节点只具有一个数据的输入接口，也就是只有数据的输入端点，这一端是连接上一个数据处理节点的，用于接收从上游节点发送过来的数据，而对于该节点的输出端，已经固化在节点内部，就是 Redis 指定的通道。在 redisPub 节点中也必须定位 Redis 的位置，也就是 Redis 服务器的 IP，端口号，不管是在 Redis 集群还是在单点的 Redis 服务器中都必须指定，同时还要指定数据输出到 Redis 哪个 channel 中。所以 redisPub 节点的 ui 设计与 redisSub 节点的 ui 设计十分相似，不同的是他们的功能代码不一样，体现在 js 文件中。图 4-6 展示了 redisPub 节点的设计逻辑的具体流程。

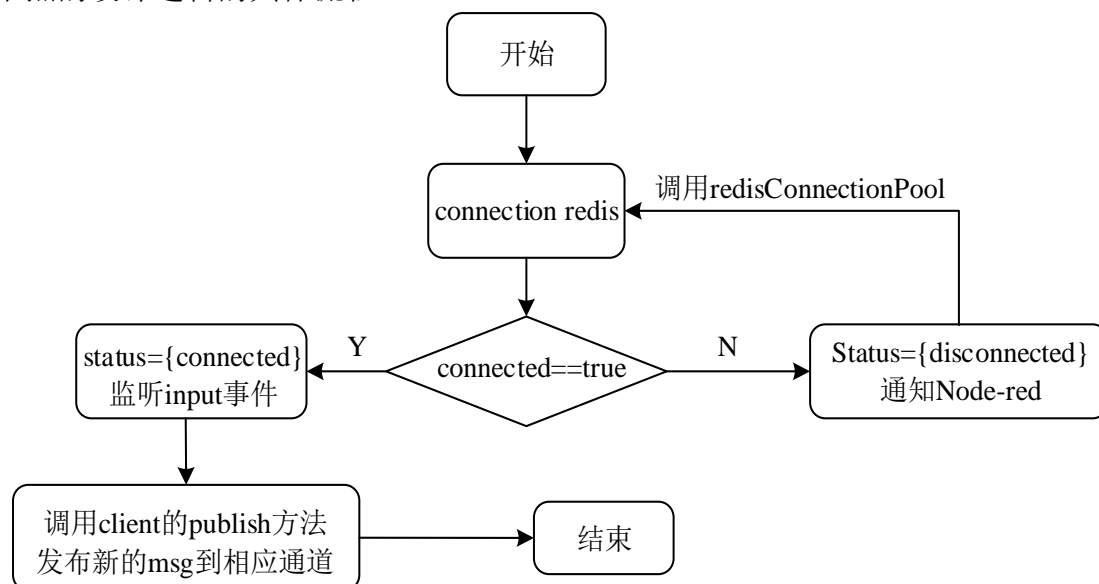


图 4-6 redisPub 设计逻辑流程图

同样，在实现 redisPub 节点的时候，也用到了数据库连接池函数，关于这个

函数的设计思想在上一小节 `redisSub` 的设计中已经做了详细阐述。从上面流程图可以看出，当 `redisPub` 节点成功连接 Redis 后，将去监听 `input` 事件，当有数据输入该节点后，就会调用 `this.client.publish()` 发布函数，将封装好的数据（`message` 对象）发布到指定的 `channel` 上。

4.3.3 数据计算节点的设计

数据计算节点在 Node-red 中起着举足轻重的作用，几乎所有的 `flow` 中都会用到数据计算节点。数据计算节点允许用户编写 JavaScript 函数来处理进入 Node-red 中的数据，编写自己的业务代码，将定义好的数据类型转化为在 Node-red 中流动的 `message` 对象。在 Node-red 中的 `message` 实际上就是一个 JavaScript 对象，`message` 对象至少要包含 `payload` 属性，用来保存具体的数据。就像下面这样一个最基本的 Node-red 的 `message` 数据格式：

```
msg={
  payload:"message payload"
}
```

计算节点接收到 `message` 后，主要处理的也是 `payload` 字段中保存的信息，处理后的数据也会封装成一个新的 `message` 对象传给下一个节点。然而，`message` 对象不仅只具有 `payload` 字段，还可以扩展出更多的其他字段来补充说明 `message` 对象的属性。比如下面这个 `message` 对象：

```
msg={
  payload:"message payload"
  topic:"error"
  location:" somewhere in space and time"
}
```

计算节点通常包含一个数据输入端点和一个或多个数据输出端点，在 Node-red 中提供了部分具有特殊功能的数据处理节点，比如 `change_node`，可以用来增加或者删除 `message` 的字段，再如 `switch_node`，可以用来做开关节点使用，它是通过判断 `message` 对象的某一字段是否存在或者真假来决定最后输出什么样的 `message` 对象。为了能够进一步扩展 Node-red 的功能，方便利用 JavaScript 函数加载外部的 `js` 模块，这里引入 `function_node`，也就是函数节点。可以说 `function_node` 在

Node-red 中就像一把瑞士军刀，可以使用户不必依赖于现有的数量有限的几个节点来处理数据。顾名思义，函数节点其实就是暴露出来的一个 JavaScript 函数，用户可用通过编写一个 JavaScript 函数来处理从上游节点流下来的 message，并返回处理后的一个或多个 message。函数节点是用来做数据处理和数据格式化的利器，引入函数节点使得 Node-red 的对流式数据进行处理变得简单容易。图 4-7 是 function_node 的设计图：

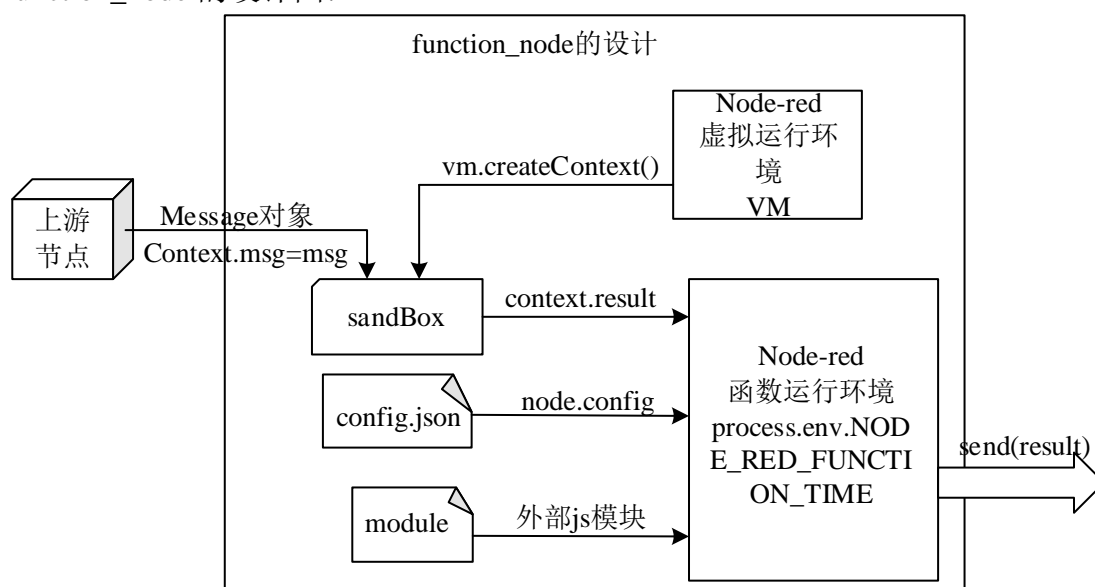


图 4-7 function_node 的设计图

用户可以通过 function_node 内置的编辑器 sandBox，编写用户自己的 JavaScript 函数来处理 message。在 function_node 内编写的 JavaScript 函数内部是调用本机上的 JavaScript 运行环境来解释执行的，同时在函数节点中可以去调用外部的 js 模块，但是这首先会去配置文件 setting.js 文件中找到要包含的模块。所以 function_node 在执行每一个函数的时候首先会去检查这个配置文件，在这个文件中去查找全局的函数模块。在 setting.js 中，通过 functionGlobalContext 支持全局模块：

```
functionGlobalContext: {
  // bonescript:require('bonescript'),
  // arduino:require('duino')
  lodash:require('lodash')
}
```

对于自己编写的 JavaScript 函数要求每一个函数都有一个返回值，也就是一个

message 对象，即使没有显式地返回，每个函数都会默认返回一个 payload 字段为空字符串的 message 对象。

4.3.4 Redis 数据库访问节点的设计

由于在原始的 Node-red 中没有与 Redis 数据库进行交互的节点，但是本文所提出的模型中用到了 Redis server 来存储中间结果集，并在 Redis server 中进行去重统计，比如计算最大值、最小值、累计求和等。所以为了能够让 Node-red 与 Redis 进行数据交换和数据传输，必须设计出对 Redis 数据库的访问操作节点。在该模型中，主要需要的就是 redis_in 和 redis_out 节点，它们分别完成从 Redis 读取数据和把数据存储到 Redis 中两项任务。

在 redis_in 中封装了几乎所有的 Redis 操作命令，该节点提供一个命令选择器，按用户指定的命令进行 Redis 操作。另外，redis_in 节点是一个只具有数据输入端点的节点，它的数据同样来源于上游函数节点提供的 message 对象中的 payload 字段(msg.payload)，用于指定命令的格式和所要操作的 Redis 集合。而对于 redis_out 节点，它既有数据的输入端，又有数据的输出端，数据的输入端是接收的数据和 redis_in 节点接收的类似，都是通过上游的函数节点发送过来，用于指明读数据的命令格式和数据所在的集合，而数据的输出端，就是将从 Redis server 上取得的数据封装成 message 对象发送给下一个节点。

根据以上对这两个节点功能的分析，接下来就对这两个节点进行详细设计。首先是 redis_in 节点，该节点第一步工作就是要去连接 Redis server，这里就会用到在 4.3.1 节中所提供的数据库连接池函数，连接成功后需要调用命令选择器，选择用户指定的命令，然后根据上游 function 节点提供的命令格式和指定的数据集，将这些信息组装成一条完整的 Redis 命令，最后调用 Redis 客户端去执行该命令。在图 4-8 中展示了 redis_in 节点的设计方案。

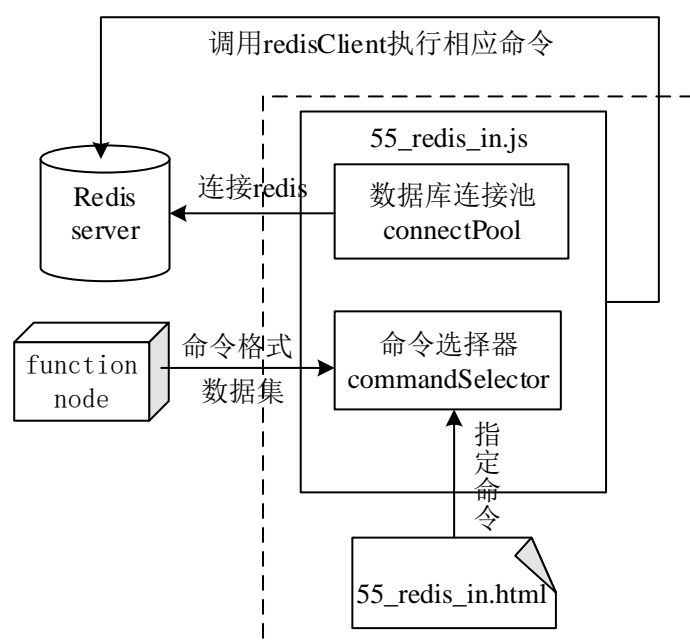


图 4-8 redis_in 设计方案

redis_in 节点在向 Redis server 存储数据的时候，主要的工作任务集中在命令选择器上。在命令选择器中保存了几乎所有的 Redis 写入操作的命令，是存放在一个数组对象中，首先要从这个数组中找到用户指定的命令，然后判断该命令是不是 psubscribe 或者 subscribe 命令，因为这两个命令在获 Rredis 数据的时候还需要监听 message 事件，而其他命令没有该事件，所以必须单独处理。最后，将用户指定的命令与上游节点传输过来的数据集拼接成 Redis 的命令交给 redisClient 执行。最终实现 Node-red 里的中间结果集存储到 Redis server 中，同时，通过上游节点指定的操作可以实现中间结果集在 Redis 中的统计计算。

对于 redis_out 节点的设计与 redis_in 节点类似，不同的是，在 redis_out 节点中同样封装了 Redis 命令，但是这些命令只是读取数据的命令，所以命令选择器中的命令与 redis_in 的不一样。另外，由于 redis_out 节点具有一个输出端，所以在 input 事件监听器中监听到的数据封装完成后，还要通过 node.send() 方法发送出去，供下一个节点接收。

4.4 节点的重新部署

节点的设计和实现完之后，一步重要的工作就是要将新设计的节点部署到 Node-red 中。节点可以作为模块打包或者发布到 npm 库中，这使得它们易于安装其所有依赖的模块。为了解决安装包的依赖关系，在打包节点的时候就要严格按照 npm 包管理规则来打包。图 4-9 是一个 redisSub 节点打包的目录结构：

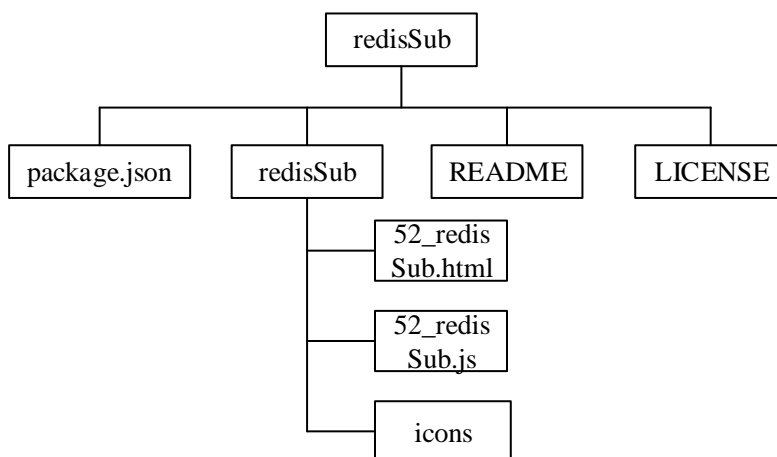


图 4-9 节点打包后的目录结构

本文采取的是本地模块安装的方式，在本地安装节点模块，就用到了 `npm link` 命令。将节点在本地目录，链接到一个本地 Node-red 安装目录，这和 `npm` 安装是一样的。本地部署节点按照如下两个步骤即可完成部署。

- 1.在包含有 `package.json` 的目录下执行 `sudo npm link` 命令；
- 2.在 Node-red 的运行运行目录下执行 `npm link <节点模块的名字>`。

部署成功后，重新启动 Node-red，然后浏览器中打开编辑界面，在最右侧的节点视图就可以看到新增加的节点，这样就完成了节点的设计和部署工作，为接下来改模型的应用提供了技术支持。

4.5 本章小结

本章首先对基于 Node-red 与 Redis 的实时流数据处理模型及其应用进行了详细的需求分析，同时也对整个模型的总体架构进行了设计，阐述了架构中各个模块的功能以及整个模型的数据处理流程。然后对 Node-red 新引入的数据输入节点、输出节点、数据计算节点以及 Redis 数据库访问节点给出详细设计方案。最后，将设计的新节点安装部署到 Node-red 中，使其成为一个完整的流式数据处理模型。

第五章 实时流数据处理模型在网站访问监控系统中的应用

第四章已经对基于 Node-red 与 Redis 的实时流数据处理模型进行了详细设计，本章的重点是将所设计的流式数据处理模型应用到实际的工程项目中，设计并实现一个网站访问的实时监控系统。本章最开始提出针对该系统的实时数据采集方案并对系统的总体架构进行设计，然后对系统的数据分析模块进行了设计并实现，该模块就是利用第四章所设计的实时流数据处理模型实现的，最后对数据可视化模块进行了详细设计。

5.1 网站访问监控系统总体设计

在各级地方政府的电子政务系统不断的发展过程中，政府的信息收集与数据分析能力还比较薄弱，急需要一个统一的实时数据收集、储存、分析、应用的平台。该系统的所有数据都是来源于某地方政府的电子政务网站群的访问流量，数据真实可靠、说服力强、具有重要的实际意义和研究价值。接下来将对网站访问监控系统的实时数据采集分案以及整个系统的模块层次结构进行详细设计。

5.1.1 实时数据采集方案设计

整个系统作为一个实时数据的交互处理中心，除了自己内部的数据通信以外，还需要对网站群的访问流量进行实时采集^[34]。这种数据具有实时性、连续性、非结构化等特点，同时数据量也非常巨大。由于数据实时性明显，同时也要求系统能够实时展示分析出网站群的访问情况，所以不能采用传统的先收集后处理的方案，需要重新设计一套实时流式数据收集方案，在服务器的网关直接利用 http_tracer 拷贝一份访问流量，然后实时的发布到 Redis Server 的 http_trace 通道中。

考虑到访问流量数据是一种非结构化的数据，为了能够更加准确地收集有效的信息，需要在采集数据的时候进行原始数据的预处理。因为原始的访问流量就是 HTTP 请求和响应报文，如果仅仅是收集到了这些报文，它都是以字符串的形式存在的，字符串不论是在数据解析过程还是在最终的数据可视化过程都使得问题变得极为复杂，为了方便解析，更好更准确的处理这些数据，有必要进行初步结构化处理。由于 json 格式的数据能够有效地反映数据的特点，同时与 JavaScript 对象能够实现无损转换，所以在进行数据格式化的时候选择 json 格式，同时在后面处理和存储中间结果集的时候也选择 json 格式。选择 json 格式来表示数据，还有一个重要的好处就是方便数据可视化，因为在数据可视化模块采用了 highcharts

来绘制图表，而 **highcharts** 要求的数据格式也是 **json** 格式。因此，我们设计出如图 5-1 所示的实时数据采集方案：

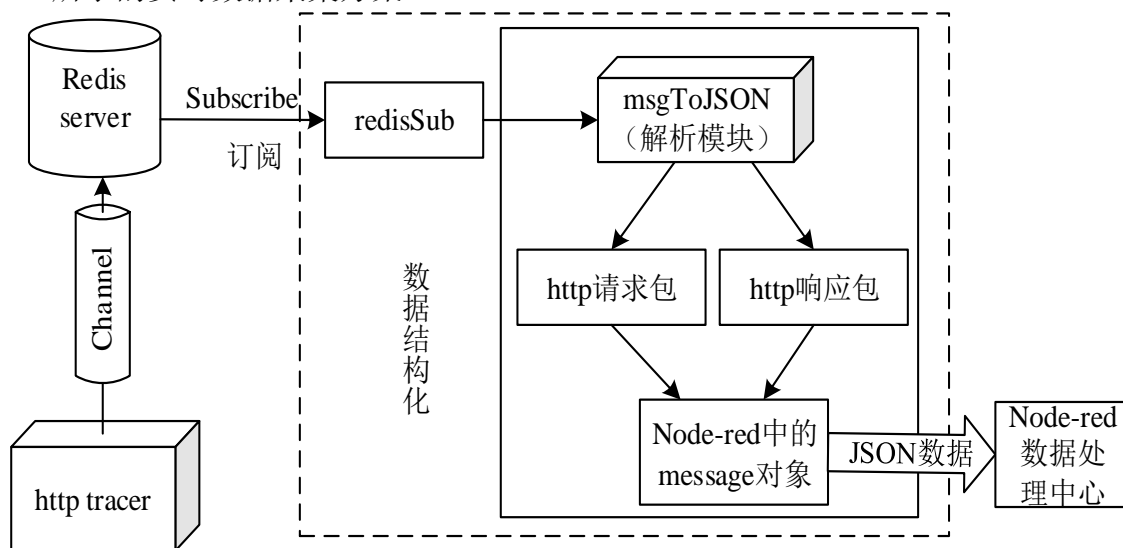


图 5-1 实时数据采集方案图

从该采集方案中可以看出，在客户的服务器端，我们将网站的访问流量做一份拷贝，利用 **http_tracer** 将这部分流量发布到 **Redis server** 中，专门设置一个 **Redis** 的通道（**channel**）用于接收从 **http_tracer** 发布过来的原始数据。由于原始的数据报结构混乱，难以分析，再加上有许多客户并不关心的信息，所以在进行下一步数据分析之前进行了预处理。原始数据通过 **redisSub** 节点从 **Redis server** 上被订阅，交给 **msgToJSON** 模块（这个模块是利用 **Node-red** 中的 **function_node** 实现的）。**msgToJSON** 模块把原始数据报文分为请求报文和响应报文两类，最后只是在 **message** 对象中增加一个 **type** 字段加以区分。最终产生的数据就是一个 **JSON** 对象，继续传递给下游的数据处理中心，进行后续的数据处理工作。下面展示的就是预处理前的原始数据报格式：

```

HTTP_TRACE_REP|1419840350182.825|172.16.1.1:18083|42.91.9.230:24735|#8|
HTTP1.1|GET|/emall/css/jquery.alert.css|304|NotModified|15.491943359375ms|Ac
cept:Referer:http://emall.lzbank.com/emall/myorder/Accept-Language:zh-cnUser-A
gent:Mozilla/4.0(compatible;MSIE8.0;WindowsNT5.1;Trident/4.0)Accept-Encodin
g:gzip,deflateHost:emall.lzbank.comConnection:Keep-AliveCookie:JSESSIONID:9
986FBA4B93382AB77946439738F1714|emall_shop_car:""|jiathis_rdc: %7B%22ht
tp%3A//emall.lzbank.com/emall/goods/goodsinfo.do%3Fgoodsid%1798085429%2
2http%//emall.lzbank.com/emall/goods/goodsinfo.do/98085429%22http%//emall.lz
  
```

通过 msgToJSON 模块进行初步的结构化处理后的 HTTP 报文变成形如下面这样的 json 对象。

```
{  
  {"type": "HTTP_TRACE_REP", //标识报文类型  
  "http_version": "HTTP1.1",  
  "http_method": "GET",  
  "userIP": "42.91.9.230", //用户的 ip 地址  
  "hostIP": "172.16.1.1", //服务器 ip 地址  
  "target": "/emall/css/jquery.alert.css", //请求的目标文件  
  "status": "304", //http 状态码  
  "host": "emall.lzbank.com\n", //网站域名  
  "user_agent": "Mozilla/4.0(compatible;MSIE8.0;WindowsNT5.1;Trident  
/4.0)\n", //用户浏览器类型  
  "referer": "http://emall.lzbank.com/emall/myorder/queryMyOrder.do"  
}
```

5.1.2 网站访问监控系统的模块层次结构

整个网站访问监控系统包括三大功能模块，实时数据采集模块、实时数据分析模块以及数据可视化模块，整个系统的模块层次结构如图 5-2 所示。

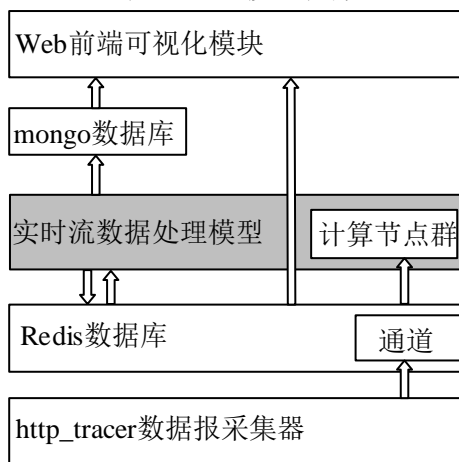


图 5-2 系统模块层次结构图

在 5.1.1 节中已经对实时数据的采集方案做了详细设计，而实时数据分析模块是搭建在第四章所设计的实时流数据处理模型上，通过编写各个功能函数节点来实现数据处理，这些函数节点构成了整个实时数据分析模块的计算节点群。同时，系统利用 mongodb 数据库完成数据持久化和用户信息保存任务，而 Redis 数据库是作为数据计算和数据交换的中心，负责中间结果集的统计和保存。

5.2 数据分析模块的设计与实现

5.2.1 数据分析模块的总体设计

数据分析模块是搭建在第四章所设计的基于 Node-red 与 Redis 的实时流数据处理模型上的，数据在不同模块之间的流动是利用 Redis 的 publish 和 subscribe 机制以及 Node.js 的 socket.io 通信机制^[32,33,34]来完成。位于网关的抓包模块抓取到原始的报文信息后，把数据发布到 Redis server 的 http_trace 通道上，然后在 Node-red 中利用在第四章设计的数据输入节点 redisSub 节点，从 Redis server 的 http_trace 通道订阅原始数据。数据进入 Node-red 之后，经过计算节点进行数据处理和封装，最后通过 redisPub 将处理结果 publish 到 Redis 的指定通道中，供可视化模块去接收这些数据。在数据处理过程中，需要用到 Redis 进行中间结果集的保存和初步的去重统计工作，这里的去重统计就是利用第三章所设计的基于 Redis 有序集合的去重统计方法，与 Redis 进行通信的节点就是第四章所设计的 redis_in 和 redis_out 节点。在节点之间的数据是通过 Node.js 的事件循环机制进行，这种机制已经被集成到 Node-red 中，因为 Node-red 也是基于 Node.js 开发的。

在进行数据分析的时候，本文主要通过三个 flow 来完成，分别完成用户行为分析、网站群页面监控以及定时清理 Redis server 上的中间结果集。在进行用户行为分析的时候，计算节点按照功能的不同划分为 5 个计算节点 refererCount、countUserAgent、repeatVisit、userIP 以及 visitPage，分别完成来路页面统计、用户的浏览器类型统计、重复访问页面统计、独立访问的 IP 地址以及受访页面统计。而对于网站群页面监控，主要涉及 5 个数据分析节点，分别是 errPage、keyWordCount、hotVisitPage、channelVisit 以及 hostCount，他们分别完成错误页面统计、关键词统计、热点页面统计、频道访问统计以及网站访问统计。这些节点一起组成了一个计算节点集群，除了这些用于数据处理和计算的节点外，还包括 Redis 数据库的访问节点用于传输中间结果集到 Redis 数据库中，还包括一个功能节点（定时节点）用于定时向前端可视化模块推送数据，以达到实时更新显示数据的变化情况，该定时节点也用于定时清理 Redis 的中间结果集，以减轻 Redis

server 的负担。整个数据分析模块的总体架构如图 5-3 所示。

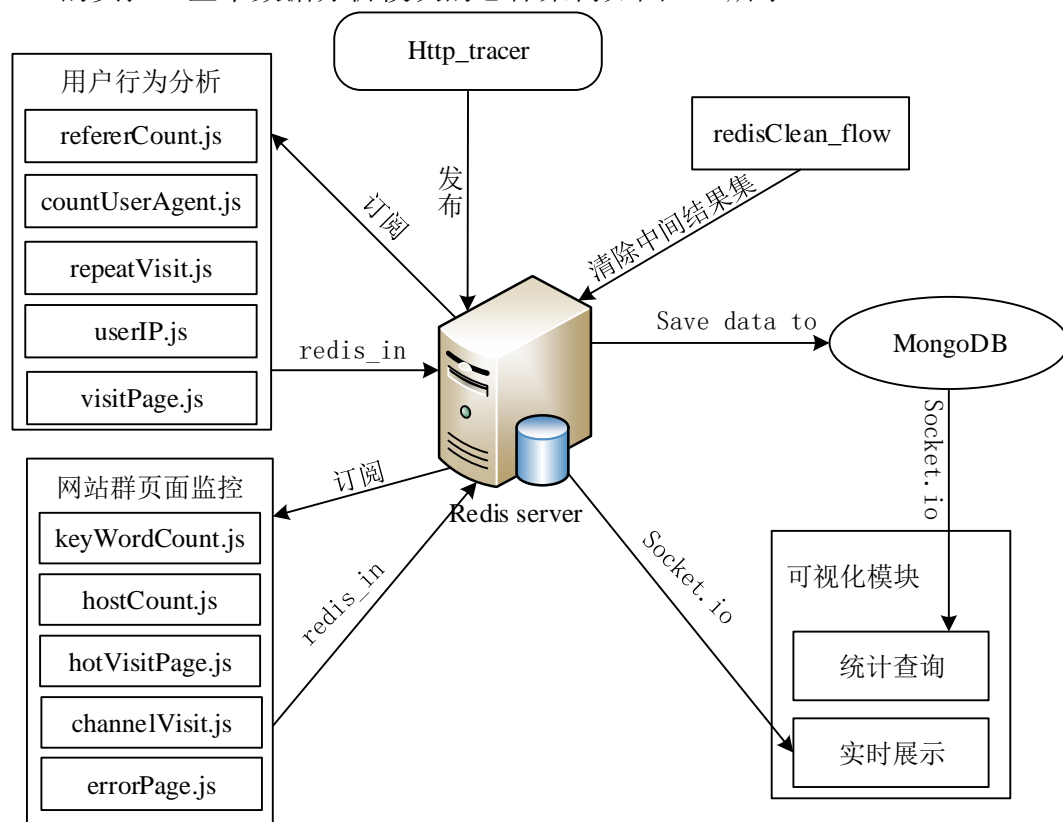


图 5-3 数据分析模块总体架构图

从数据分析模块的总体架构图可以看出，Redis 是整个模块数据交换的纽带，也是进行数据计算的中心。Redis 的发布/订阅机制使得各个功能的计算节点所计算的数据结果相互独立、互不影响，这样使得最终数据结果的展示变得清晰、一目了然。前端可视化模块与 Redis server 进行通信的工具是 socket.io，通过事件驱动机制，监听事件是否发生来判断是否有数据到来，从而达到数据传输的目的。另外，由于巨大的数据量和繁重的数据计算任务，导致 Redis server 的负担也异常繁重。为此，需要定时清理 Redis server 上的中间结果集以减轻其计算和存储压力，在架构图中的 redisClean_flow 就是专门用于清理 Redis server 上的中间结果集。系统选择在每天的凌晨清理数据，这样一方面可以达到减轻 Redis server 负担，提高运行效率的目的，另一方面也不会影响数据分析的结果和前端可视化模块的展示，因为一天的实时数据分析工作在凌晨已经全部完成。

由于客户的要求，需要对用户的行为数据进行持久化，并且能够在系统的可视化模块中随时可以查询到历史的用户行为数据。为此，引入了 mongo 数据库提供数据持久化功能，同时前端可视化模块中增加了统计查询页面，前端可视化模块是通过 socket.io 来查询 mongo 中的历史行为数据。

5.2.2 数据库结果集设计

数据分析模块得到的最终分析结果都存放在 Redis 中，在 Redis 中进行统计和初级计算。这些数据在 Redis 中是一系列的字符串、集合、有序集合以及 hash 表，用它们表示这用户行为数据和页面的统计数据。

在进行用户行为分析的时候会用到 IP 地址所属地的码表，域名与网站名的对应关系，在网站群页面监控的时候需要用到频道 url 与频道名的对应关系，这些信息都存放在 Redis 的哈希表中。而对于像数据分析中的用户 pv、uv、错误页面统计、热门关键词的搜索、热点页面等累加型数据指标，需要用到 Redis 的有序集合来存储，因为要计算这些指标最基本的工作就是去重统计，而本文的去重统计是利用 Redis 的有序集合来实现的。另外利用有序集合来存储这些计算指标的一个好处在于，能够很容易地取到 top (n) 指标，因为有序集合中的元素是经过排序的。在整个数据分析过程中，有序集合和哈希表是主要的存储结构，同时也有部分数据是通过集合和字符串来存储的。具体的 Redis 表结构如表 5-1 所示。

表 5-1 数据分析模块中的 Redis 表结构说明

Redis 表名	类型	功能描述
visitPage.zset	排序集合	热点访问页面统计
referrerPage.zset	排序集合	来路页面统计
userIP.set	集合	用户的 ip 统计
repeatVisit.zset	排序集合	重复访问的页面统计
HostName.set	集合	记录网站的域名
IPBelong.hash	哈希表	记录访问 ip 所属区域
userAgent.zset	排序集合	统计用户浏览器类型
KeyWorld.zset	排序集合	关键词统计
HotVisitPage.zset	排序集合	热点页面统计
HostVisit.zset	排序集合	访问网站统计
HostName.hash	哈希表	记录网站名称与域名的对照
ChannelName.hash	哈希表	记录频道连接与频道名的对照
ChannelVisit.zset	排序集合	频道访问统计
errPage.zset	排序集合	错误页面统计
errType.zset	排序集合	错误类型统计
errWebHostName.string	字符串	错误页面所属的网站统计

数据存储到 Redis 的这些数据结构中, 进行统计计算, 计算的最终结果就是下一阶段前端可视化模块中的图表(线图、饼图、柱状图)的原始数据。数据分析模块向中间结果集中单向写入数据, 而数据可视化模块从中间结果集中单向读取数据, 两个模块之间并无直接交互, Redis 是连接它们的纽带, 他们之间的通信是通过 Node.js 的 socket.io 进行的^[35,36,37]。

对于数据持久化这一功能, 原始的 Node-red 提供了 mongodb 的访问节点, 用于操作 mongo 数据库。为了存储这些历史行为数据, 必须设计合适的 mongo 数据集合, mongo 数据集合必须要有时间和统计指标的表示, 这样可以方便客户对历史数据的检索。具体的数据结构如表 5-2、表 5-3、表 5-4 所示, 其中表 5-2 是记录每天独立 IP 的访问量的 mongo 集合, 表 5-3 展示的是记录每天页面的访问量排名前十的 mongo 集合, 表 5-4 表示的是统计同一页面的重复访问情况的 mongo 集合。而对于其他的访问数据, 客户不要求存储, 这里就不再设计相应的数据集合。

表 5-2 统计独立 ip 的访问量 mongo 集合

集合名	unique_ip_count		
功能	每天独立访问的 IP 数量, 包括 IP 的地域信息		
键的说明	键名	数据类型	说明
	_id	objectid	mongo 数据库唯一性标识
	date	time	时间, 用于表明是某一天的数据
	ip	string	独立 ip
	ip_belong	string	该 ip 所属的区域(内网/外网)
	count	int	ip 的计数

表 5-3 统计每天访问量排名前十的页面的 mongo 集合

集合名	page_visit_top10		
功能	记录每天访问量排名前十的页面		
键的说明	键名	数据类型	说明
	_id	objectid	mongo 数据库唯一性标识
	date	time	时间, 用于表明是某一天的数据
	url	string	访问页面的 url
	url_belong	string	受访页面所属的网站名
	count	int	受访次数

表 5-4 统计每天重复访问率的 mongo 集合

集合名	repeat_visit_count		
功能	记录每天同一 ip 访问同一页面的情况		
键的说明	键名	数据类型	说明
	_id	objectid	mongo 数据库唯一性标识
	date	time	时间，用于表明是某一天的数据
	url	string	访问页面的 url
	ip	string	用户的 ip
	count	int	访问次数

5.2.3 数据分析算法的设计与实现

本小节将对数据分析模块中，在进行用户独立 IP 的访问量、用户浏览器类型统计、页面的重复访问率、错误页面统计以及热门关键词搜索统计，这五个指标的计算时所用到的算法流程进行详细阐述。首先是用户独立 IP 访问量，由于同一 IP 在同一天多次访问只能记录一次，所以鉴于这个特性，利用 Redis 的集合来进行存储统计。当流量数据进入计算节点后，首先判断是否解析到了 userIP 字段，如果解析到了就到 Redis 的 userIP.set 中查找是否有该 IP 信息，如果没有就插入该 IP。具体算法流程如图 5-4 所示：

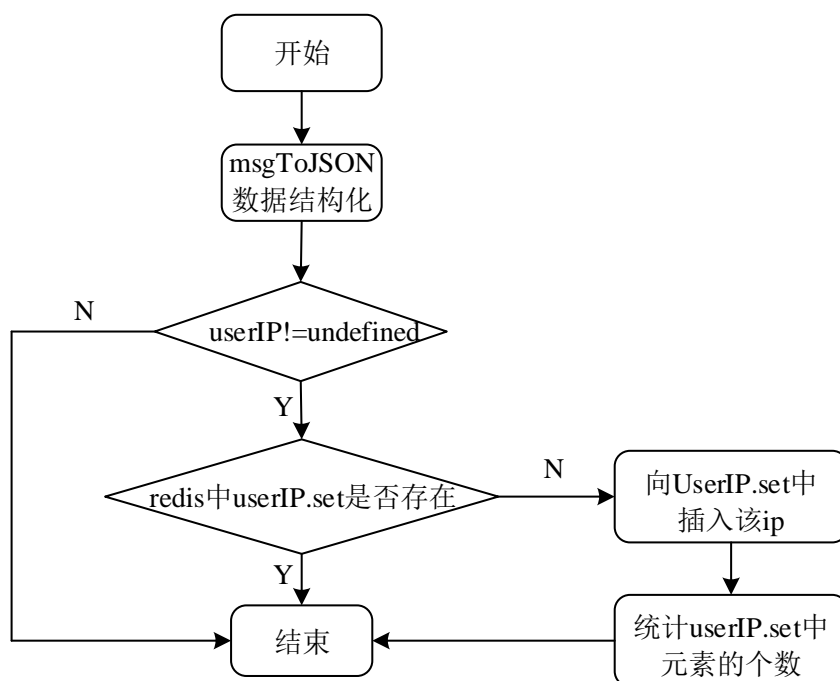


图 5-4 独立 ip 访问统计的算法流程图

该算法主要通过两个函数节点来完成，第一个函数节点是判断 msgToJSON 节点发送过来的数据是否是请求报文，并且是否解析到了 userIP。该函数节点所实现的代码如下：

```
if(msg.payload.type=="HTTP_TRACE_REQ"&&msg.payload.userIP!=undefined){  
    var userIP=msg.payload.userIP;  
    msg.payload={"userIP":userIP};  
    return msg;  
}
```

另外一个函数节点是用于将上面函数节点传递过来的 msg 重新以命令的形式封装起来，通过 redis_in 节点去操作 Redis 中的 userIP.set 集合。这里用到集合的 zadd 命令，该命令是向集合中插入一个元素，如果该元素已经在集合中，就不做任何操作，如果不在集合中就执行插入操作。

```
var userIP=msg.payload.userIP;  
msg.payload=['zadd','userIP.set',userIP];  
//zadd 命令自动判断记录是否在集合中  
return msg;
```

对于用户浏览器类型统计，是根据 msgToJSON 节点发送过来的数据中的 user_agent 字段来进行统计的。如果从原始报文中解析到了该字段，得到的数据格式为 "user_agent":"Mozilla/4.0(compatible;MSIE8.0;WindowsNT5.1;Trident/4.0)\n"，然后再利用相应的正则规则，解析出浏览器的类型为微软公司的 IE 浏览器，相应的版本号是 8.0，再将解析结果封装为 Redis 有序集合 zset 的 zincrby 命令操作格式，并传递给 redis_in 节点进行去重统计。图 5-5 所展示的就是用户浏览器类型统计的算法流程图。

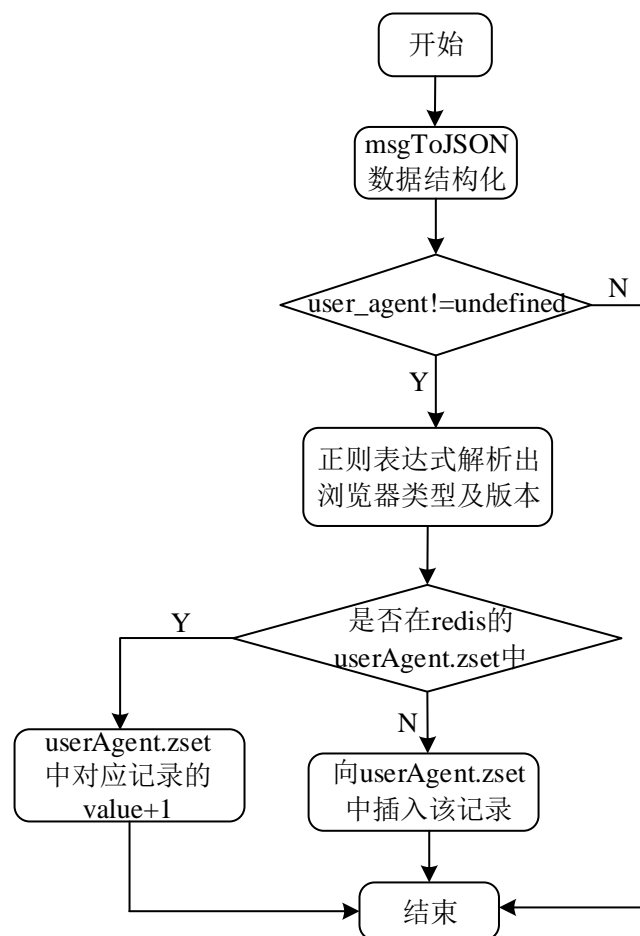


图 5-5 用户浏览器类型统计算法流程图

在 Node-red 中，该算法主要通过一个函数节点来实现的，首先判断 `msg.payload.user_agent` 是不是 `undefined`，如果从用户的请求报文中没有解析到浏览器类型，`msgToJson` 节点所发送的数据中将没有 `user_agent` 字段，也就是未定义的。如果解析到 `user_agent`，那么就利用 JavaScript 的字符串操作函数 `indexOf()` 和 `substring()` 解析出浏览器的类型和版本号，并保存到 `type` 和 `version` 两个变量中。然后，将 `type` 和 `version` 两个变量拼接起来，中间用特殊字符 ‘#’ 分隔，这样是为了后面封装数据时利用 `split()` 函数将其切割开，这样做还有一个好处就是可以降低统计指标的维度以减轻计算的复杂度。最后，将拼接好的两个数据封装在 Redis 有序集合的 `zincrby` 命令中，作为 `msg` 的 `payload` 字段，发送到下一个节点(`redis_in`)。下面展示的就是该函数节点的核心实现代码：

```
if((agent=msg.payload.user_agent)!=undefined){  
    if(agent.indexOf("Mozilla")>=0&&agent.indexOf("Chrome")>=0){  
        type="chrome";  
        version=agent.substring(type.indexOf("Chrome"),2);  
    }  
    else if(...)//其他浏览器类型 ie、sogou、baiduspider 等  
    else{  
        type="other";  
        version="undefined"  
    }  
    var temp=type+"#"+version;//拼接 type 与 version  
    msg.payload=["zincrby","userAgent.zset",1,temp];  
    //将拼接结果封装在新的 msg 对象中  
    return msg;  
}
```

要计算页面的重复访问率，需要首先计算出同一 IP，在同一天访问同一页面的次数，同时还要计算出该 IP 的访问总数，然后将它们的值相比，就得到了重复访问率。该指标具有用户 IP 和受访页面的 URL 两个维度，所以在计算该指标之前必须从 target 字段中解析出某一用户 IP 当前所访问页面的 URL。将 IP 和 URL 作为一个统一体来进行计算，这样才能保证计算的是同一 IP 访问同一页面的次数。因此，需要将解析到的 URL 与 userIP 进行拼接，拼接后的结果作为有序集合中的键值，再利用 Redis 有序集合的去重统计方法进行统计计算。如何从 msg.payload.target 中解析出受访页面的 URL 是该算法的重点，众所周知，页面的 URL 都是以 .html 结尾而不包含其后面的乱码信息，所以利用 JavaScript 处理字符串的正则规则就可以得到该页面的 URL，在该 URL 前面加上网站的域名就是完整的 URL。图 5-6 所展示的就是重复访问率的统计算法流程图。

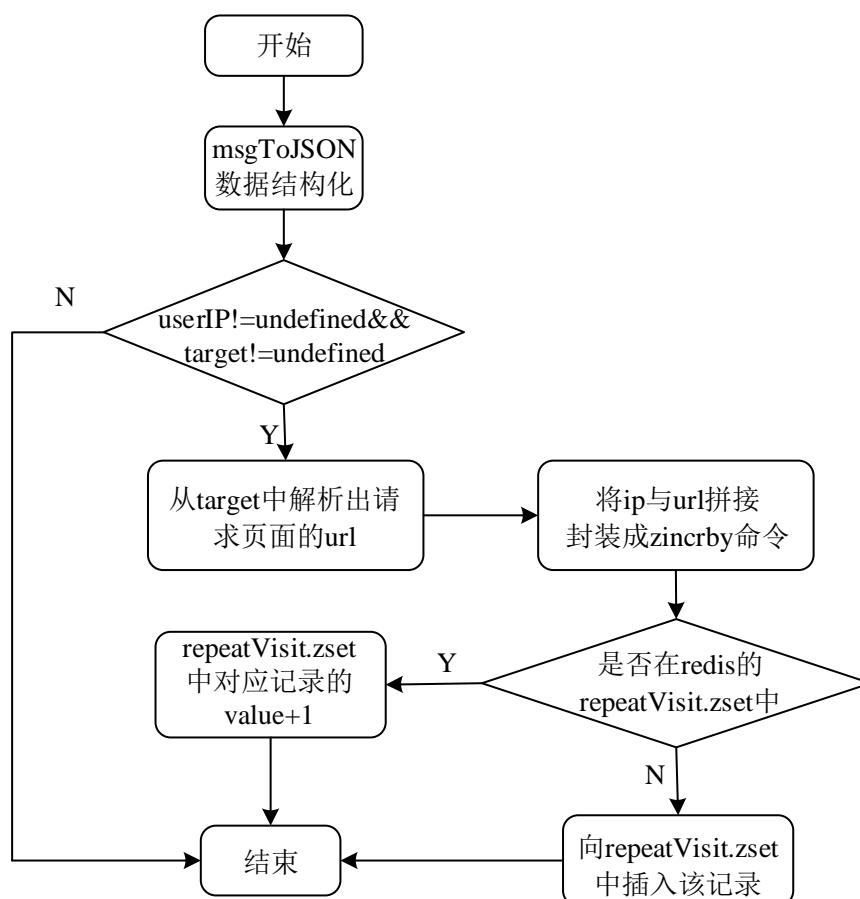


图 5-6 重复访问率统计算法流程图

在 Node-red 中该算法主要是通过两个函数节点来实现的，第一个就是从 `target` 字段中解析出请求页面的 URL，并把解析结果和 `userIP` 封装在 `msg` 的 `payload` 字段中，传递给下一个函数节点。第二个函数节点的功能仍然是封装数据结果形成 `zincrby` 命令中，与前面几个算法的命令封装类似，这里就不作详细阐述了，下面展示的是解析 URL 的核心实现代码：

```

if(string.indexOf(host)>=0) { //host 表示访问网站的域名（或者 IP）

    if(target.indexOf('.html')>0) //页面都是以'.html'结尾的

        if(target.indexOf('?')>0) //去掉'?'后的字符串

            visitPage=host+target.substring(0,target.indexOf('?'))

        else

            visitPage=host+target;

}
  
```


对于错误页面统计，需要给出错误页面的错误类型、错误页面所属的网站以及错误页面的 URL，而这些信息分别来源于 msgToJSON 节点的 msg.payload.errType、msg.payload.host、msg.payload.target，但是 msg.payload.host 给出的是网站的域名，为了取得网站的名字，还需要去查 Redis 中的 hostName.hash。图 5-7 展示的就是该算法的具体流程。

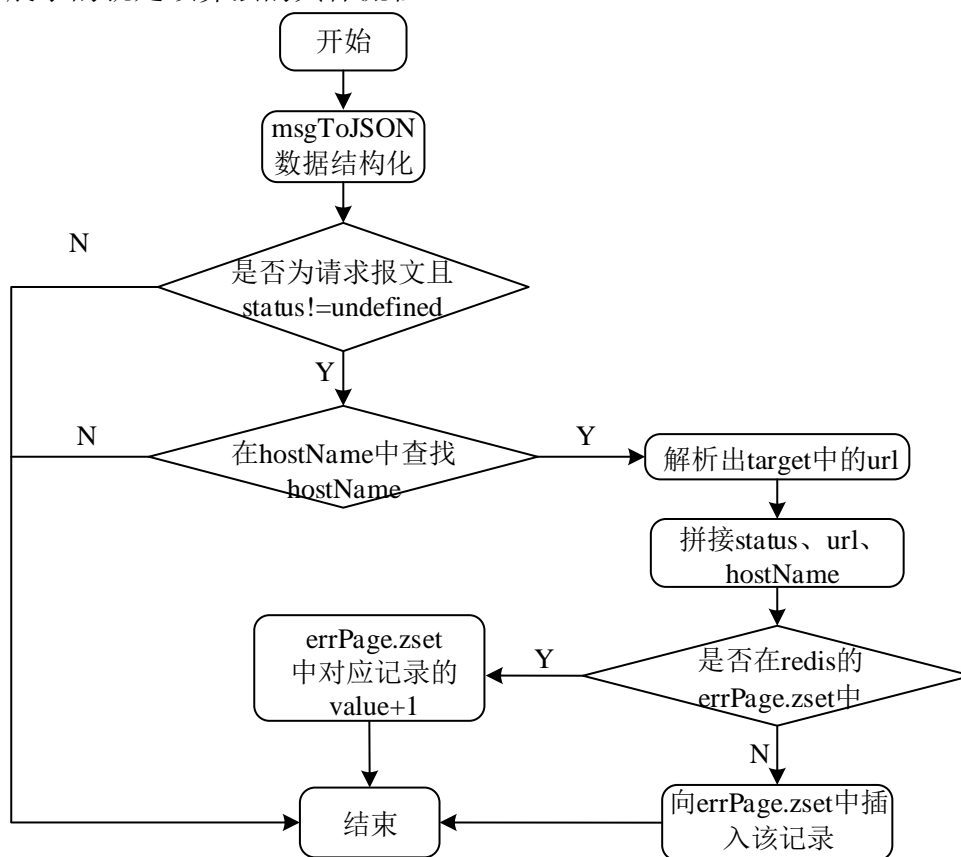


图 5-7 错误页面统计的算法流程图

该算法主要通过两个函数节点来完成，第一个函数节点的主要功能是从请求报文中解析错误页面所需的信息，包括错误的状态码，错误页面的 URL 以及错误页面所属网站。第二个函数节点的主要功能是将第一个函数节点发送过来的数据封装成 Redis 有序集合的 zincrby 命令，交给 redis_in 节点去操作 Redis server，进行去重统计。这里主要阐述第一个函数节点的具体实现代码，因为第二个函数节点的实现与之前几个算法的命令封装函数十分类似，这里就不再赘述。众所周知，在 HTTP 所有的响应码中凡是大于 400 的都是错误的，比如用户最关心的是 404 错误，表示链接所指向的页面不存在，即网页的 URL 失效。所以在对响应码进行筛选的时候只需要判断响应码是否大于 400 即可，下面就是解析错误页面所需信息函数节点的部分核心代码。

```

if(msg.payload.type=="HTTP_TRACE_REP"&&msg.payload.status!=undefined){
    ...//保存上一节点传递过来的 msg.payload 数据

    if(parseInt(msg.payload.status)>400)//只统计响应码大于 400 的报文{
        ...//解析错误页面的 URL

        if(msg.payload.referer=="")
            ...//记录来路页面的保存
    }
}

```

对于热门关键词搜索统计，用户搜索的关键词都会在请求报文中以字符串“&q=”开头，以“&”结尾，而这些信息都包含在 msgToJSON 节点发送过来的 target 字段中。由于客户的要求，需要统计出每个关键字来自于哪个网站，因此在记录关键字的同时，还需要保存网站的域名，在通过查找 hostName.hash 找到相应的网站名称。图 5-8 展示的就是该算法的流程图。

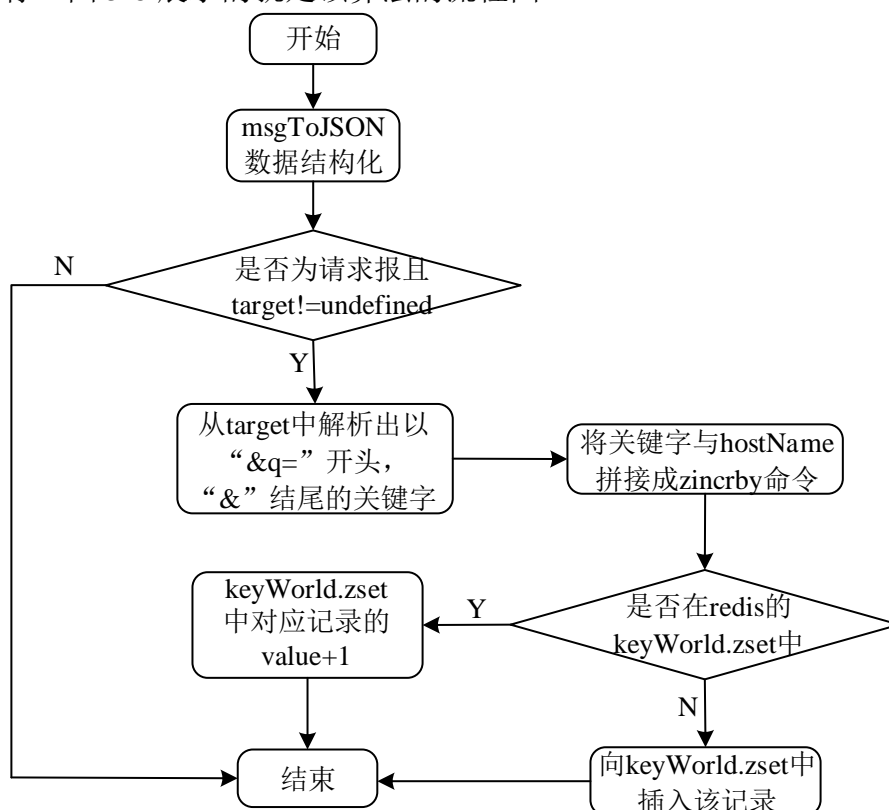


图 5-8 热门关键词统计算法流程图

在该算法中，主要的工作任务在于关键字的解析函数，关于命令封装函数的实现和前面几个算法十分类似，不同的是操作的有序集合名以及向有序集合中插入的元素值，这里就不赘述了，下面主要展示的是关键字的解析函数的核心实现代码：

```
if(msg.payload.type=="HTTP_TRACE_REQ"&&msg.payload.target!=""){  
    if(target.indexOf("&q=")>0){  
        //解析以"&q="开头，以'&'结尾的字符串  
        var temp=target.substring(target.indexOf("&q="),target.length);  
        var keyWord=decodeURI(temp.substring(0,temp.indexOf('&')));  
    }  
    if(keyWord!=""){  
        //将 host 与 keyWord 拼接起来形成新的 msg  
        msg.payload={"hostName":msg.payload.host,"keyWord":keyWord};  
        return msg;  
    }  
}
```

数据通过 `redisSub` 节点从 `Redis` 通道获得原始数据，再经过初始化模块 `msgToJSON` 对原始数据进行初步的结构化处理，然后就进入后续的数据分析处理阶段，各个计算节点是通过 `Node-red` 的 `function` 节点实现的，各计算节点产生的 `message` 对象传递给下一个 `function` 节点，数据最终被封装成 `Redis server` 中相应数据结构，通过 `redis_in` 节点将这些中间结果集保存起来，在 `Redis` 中进行统计计算。然后将 `Redis` 上的计算结果通过 `redis_out` 节点取得，并封装成前端可视化模块需要的数据格式，最后通过 `redisPub` 节点将最终的数据定时（每两秒钟发送一次）发送的 `Redis server` 的指定通道上供可视化模块接收并展示。

下面将逐一阐述在 `Node-red` 中整个数据分析模块的数据处理流程，整个数据处理流程由三个 `flow` 构成，其中第一个 `flow` 是将原始数据按功能需求，解析形成中间结果集并存储到 `Redis` 中进行统计计算，正如图 5-9 所示。

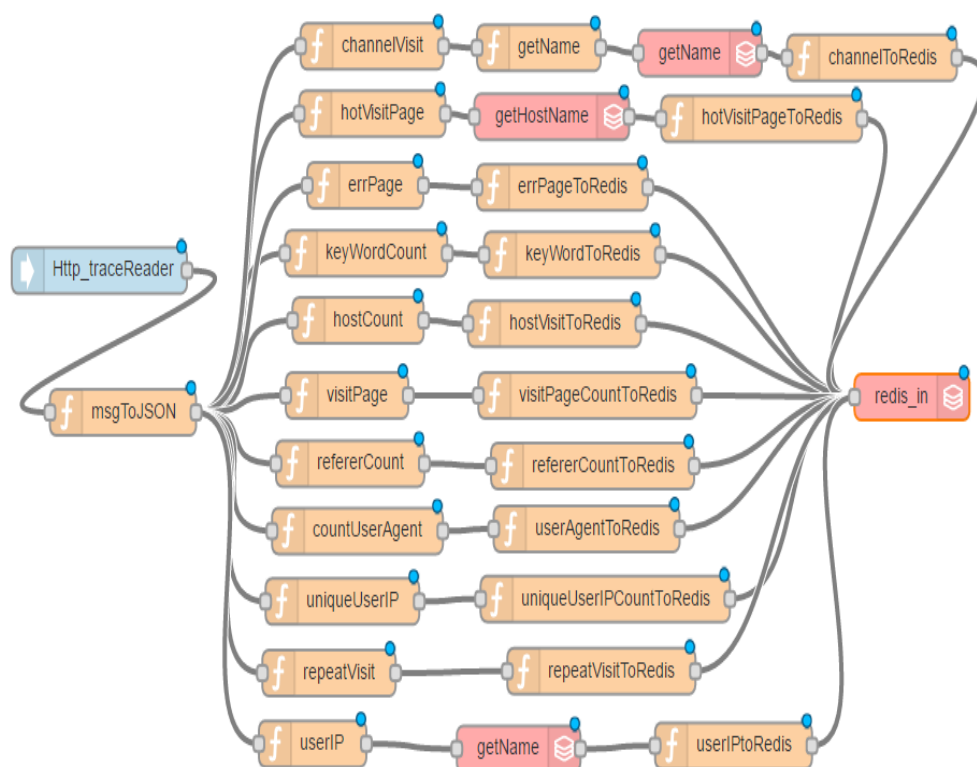


图 5-9 用户行为分析模块中形成中间结果集的 flow

在图 5-9 所示的形成中间结果的 flow 中，各计算节点从预处理节点 msgToJSON 中得到初始化的数据后，将数据分发到各个单一功能的计算节点上，按照上一小节的算法流程编写相应的功能函数进行数据封装和计算。在这个数据流程中包含 11 个功能节点，visitPage、refererCount、countUserAgent、uniqueUserIP、repeatVisit、userIP、errPage、keyWord、hostCount、hotVistPage 以及 channelVisit，分别完成受访页面统计、来路页面统计、用户浏览器类型统计、独立访问 IP 统计、重复访问率统计、用户 IP 所属地统计、错误页面统计、热门关键词统计、域名访问统计、热点页面统计以及频道访问统计。每个功能节点后面都有一个 ToRedis 的函数节点，该节点主要完成上游节点处理后的数据封装工作，将其封装成 Redis 中指定有序集合的 key 值，并利用 zincrby 命令对该 key 值进行去重统计。接下来就以统计重复访问率为例来阐述 ToRedis 函数节点的具体实现方式。

```
var userIP=msg.payload.userIP;

var target=msg.payload.target;

var temp=userIP+'#'+target;//将用户 IP 与受访页面的 url 进行拼接

msg.payload=['zincrby','repeatVisit',1,temp];
```

在图 5-9 所示的 flow 中，其余 ToRedis 函数节点的 msg.payload 格式与 repeatVisitToRedis 中的类似，主要区别在于操作的有序集合和向有序集合中插入的记录不同。封装好的 msg 就会传递给 redis_in 节点，通过 redis_in 节点去操作相应的有序集合。

经过 Redis server 的去重统计后，结果会保存到 Redis 的有序集合 zset 中，另外有一些码表之类的数据会保存到 hash 表中，以便快速查找。为了能够实时地更新数据，在 Node-red 中设置了一个定时器，规定每两秒钟到 Redis server 上去取一次数据，将取得的数据封装成可视化模块需要是数据结构，实际上就是 highcharts 所需要的数据结构，关于 highcharts 的数据格式将在可视化模块设计中加以阐述。数据分析模块的第二个 flow 就是实时取得数据并发布数据到相应的 Redis 通道中，供前端可视化模块接收并展示。需要注意的是，这里的通道必须与前端可视化模块对应的展示图表所绑定的通道保持一致，因为各个图表所需要的数据都是放在不同的通道中的。图 5-10 所展示的就是定时取得并推送数据的 flow。

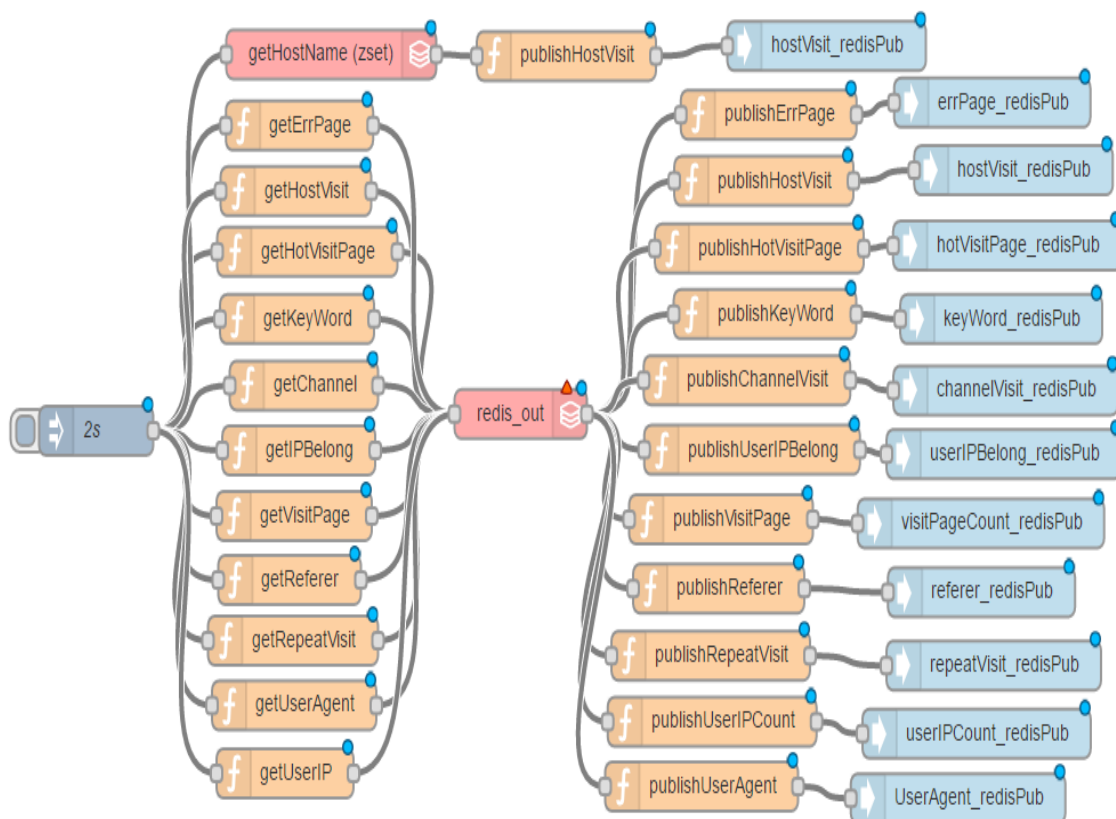


图 5-10 定时取得数据并推送数据的 flow

在图 5-10 所示的 flow 中，每条数据流上的最后一个节点就是本文所设计的 redisPub 节点，它将上游节点发送过来的已经被封装好的数据发布到指定的 Redis

通道当中。而每条数据流上的 `publish_` 函数节点是用于封装取得的数据，`publish_` 函数节点所封装的数据格式必须与前端可视化模块的 `highcharts` 要求的格式一致，关于这一点将在下一小结可视化模块的设计中详细阐述 `highcharts` 的数据格式，同时，`redisPub` 节点所指定的 `Redis` 通道一定要与前端可视化模块中相应图表所绑定的通道一致，因为这些图表的数据都是从 `redisPub` 节点所发布的通道中获取的。另外，每一条数据流上都有一个 `get_` 的函数节点，该节点是用于封装 `Redis` 的取数据命令，通过 `redis_out` 节点去取得相应的数据。这些节点的具体实现代码非常相似，不同之处在于命令所操作的有序集合以及所取元素个数不同。表 5-5 展示了各个 `get_` 函数节点的取数据命令。

表 5-5 各数据获取函数与取数据命令对照表

函数名	取数据命令	说明
<code>getIPBelong</code>	<code>['IPBelong.zset',0,20, 'withscores']</code>	排名前 20 的访问 ip 的所属地
<code>getVisitPage</code>	<code>['visitPage.zset',0,20, 'withscores']</code>	访问量排名前 20 的受访页面
<code>getReferer</code>	<code>['referer.zset',0,10, 'withscores']</code>	排名前 10 的来源页面
<code>getRepeatVisit</code>	<code>['repeatVisit.zset',0,20, 'withscores']</code>	重复访问量排名前 20 的页面
<code>getUserAgent</code>	<code>['user_agent.zset',0,8, 'withscores']</code>	排名前 8 的用户浏览器类型
<code>getUserIP</code>	<code>['userIP.set',0,20, 'withscores']</code>	排名前 20 的独立访问 ip
<code>getErrPage</code>	<code>['errPage.zset',0,1000, 'withscores']</code>	排名前 1000 位的错误页面
<code>getHostVisit</code>	<code>['hostVisit.zset',0,20, 'withscores']</code>	访问量排名前 20 的网站
<code>getHotVisitPage</code>	<code>['hotVisitPage.zset',0,50, 'withscores']</code>	访问量排名前 50 的页面
<code>getKeyWord</code>	<code>['keyWord.zset',0,20, 'withscores']</code>	搜索量排名前 20 的关键字
<code>getChannel</code>	<code>['channelCount',0,20, 'withscores']</code>	访问量排名前 20 的频道

由于数据不间断的到来，数据量也十分巨大，所以每天在 `Redis server` 中都会堆积大量的中间结果集，再加之 `Redis` 还要完成基本的计算工作，`Redis server` 一般都是超负荷运行的。为了减轻 `Redis server` 的运行负担和存储压力，提高 `Redis` 的计算效率，必须对 `Redis server` 上的中间结果集进行定时清理，同时也要保证被清理的数据不会影响最终的分析结果。定时清理的 flow 如图 5-11 所示。

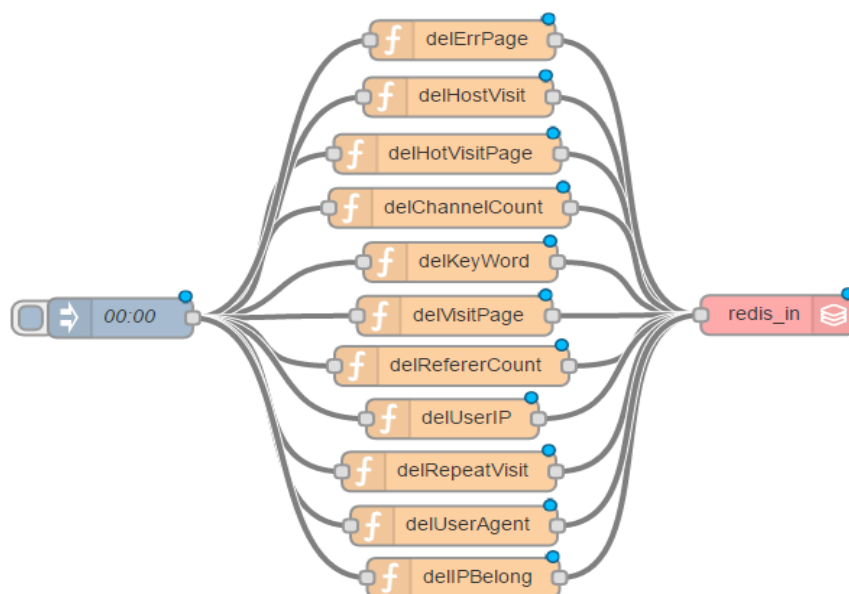


图 5-11 定时清理 Redis server 中间结果集的 flow

在图 5-11 所示的 flow 中用到了一个定时节点，选择在每天的凌晨进行中间结果集的清理工作，这是因为一天的数据分析任务在每天的凌晨就全部完成，所以不会影响最终的分析结果。每个 del_函数节点把所产生的 message 发送给 redis_in 节点，由 redis_in 节点去执行中间结果集的删除操作。其中 del_函数就是封装删除 Redis 中所存储的中间结果集的命令，表 5-6 展示了各个删除函数与删除命令的对照关系。

表 5-6 各删除函数与删除命令的对照关系

函数名	删除命令	说明
delVisitPage	['del','visitPage.zset']	删除记录受访页面的 zset
delRefererCount	['del','referer.zset']	删除记录来路页面的 zset
delUserIP	['del','userIP.set']	删除记录用户 IP 情况的 zset
delRepeatVisit	['del','repeatVisit.zset']	删除记录重复访问页面的 zset
delUserAgent	['del','userAgent.zset']	删除记录用户浏览器信息的 zset
delIPBelong	['del','IPBelong.zset']	删除记录 IP 所属地信息的 zset
delErrPage	['del','errPage.zset']	删除记录错误页面信息的 zset
delHostVisit	['del','hostVisit.zset']	删除记录网站访问量的 zset
delHotVisitPage	['del','hotVisitPage.zset']	删除记录热点访问页面的 zset
delChannelCount	['del','channel.zset']	删除记录频道访问量的 zset
delKeyWord	['del','keyWord.zset']	删除记录关键字搜索频率的 zset

到目前为止，本文所设计的网站访问监控系统的数据利用本文所设计的流式数据处理模型进行了计算处理，也产生了最终的数据结果，但是由于客户不知道具体的数据分析处理过程，因此，对客户来说这些数据都不便于观察，不便于分析其变化情况，需要设计一种数据可视化方案来形象、直观地展示这些数据。本文将在下一小节对数据可视化模块进行详细阐述。

5.3 数据可视化模块设计

5.3.1 数据可视化模块的功能需求

经过用户行为分析之后，得到的结果是代表用户的访问量、新用户的数量、访问的来源、用户浏览器类型等信息。而网站群页面监控分析，最后的结果是代表关键词搜索频率、热点页面统计、错误页面统计等信息。所有最终结果的数据都是一系列统计表和集合的数据结构，这些数据不能直接交由用户，因为用户不了解分析的过程，就不知道这些数据代表什么含义，可视化模块的功能就是要把这些结果生动直观地显示出来^[40,41]。同时，可视化模块要实时更新，当新的用户行为数据和网站群监控数据被分析出来后，要能立刻在可视化模块中看到。需要支持的图表形式有：

1. 饼图，用于显示用户设备和浏览器使用比例。
2. 柱状图，展示错误页面统计信息、错误类型统计信息、网站的访问量等。
3. 列表，用于显示关键词统计、热点页面等信息。

本文在设计可视化模块的时候，为了适应多样化的数据展示，便于系统的扩展，所支持的图表不局限于以上三种，还提供了曲线图、散点图等。

5.3.2 可视化模块的架构设计

监控系统的可视化模块使用了 MVC 架构的 Web 网站来实现的^[40]。使用 MVC 架构的 Web 网站，就是通过模型(model)—视图(view)—控制器(controller)这样一种典型的软件设计范式而设计出来的一种网站结构模式。model（模型）是应用程序的核心部分（比如数据库记录列表）。view（视图）是显示数据的模块（图表、柱状图、饼图）。controller（控制器）处理输入（响应请求的接口，接收消息）。图 5-12 所展示的就是该系统的 MVC 架构图。

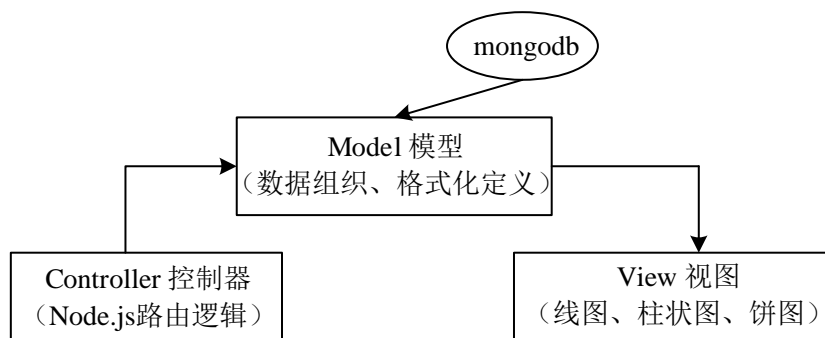


图 5-12 监控系统的 MVC 架构图

监控系统从功能上也分为 `model`、`view` 和 `controller` 三个功能模块。使用 MVC 架构的网站前端，具有数据格式统一并且显示风格多变的特点，十分适合多样化数据呈现的网站群监控数据的展示。前端页面设计为一个 web 应用，使得显示的界面在任何一台机器上都可以查看，不需要额外安装应用软件，通过浏览器登录就可以访问，减少了软件安装和更新所带来的麻烦，同时提高了系统的灵活性。

整个前端的显示系统是利用 Node.js 的 `express` 框架^[41,42]来实现的，`express` 是一个简洁、灵活的 web 应用框架，它的强大特性有助于快速创建各种 web 应用，同时提供了丰富的 `http` 工具。监控系统前端网站所使用的 `express` 框架的各个模块的调用关系如图 5-13 所示：

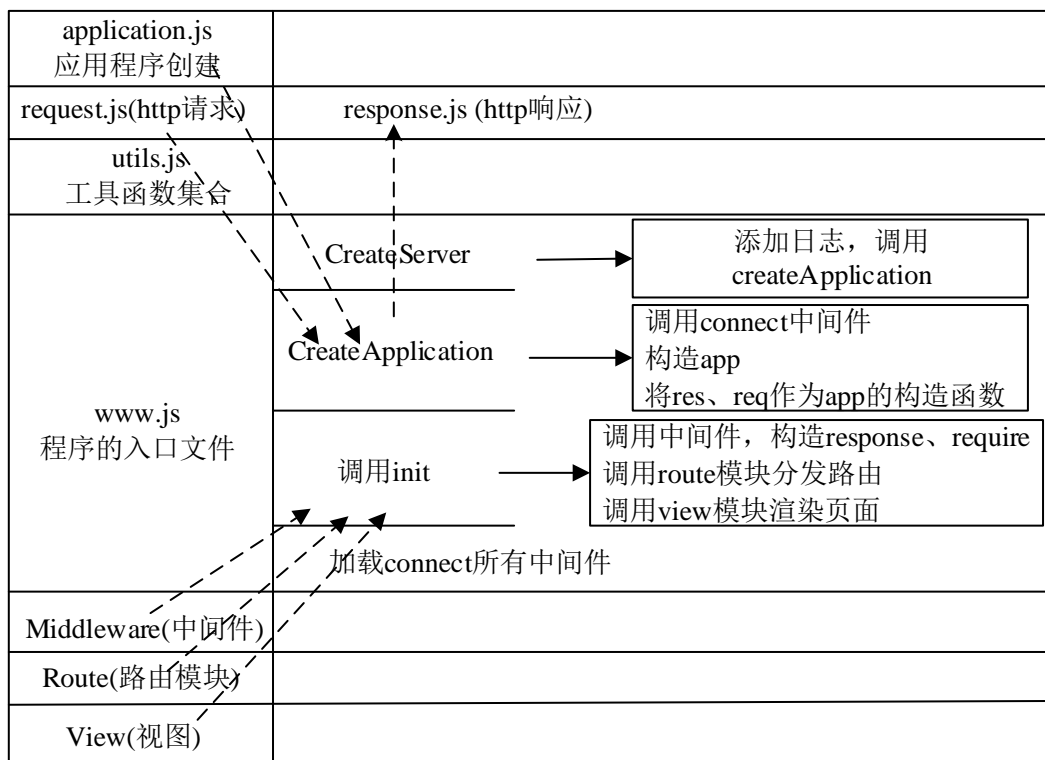


图 5-13 express 框架的调度关系图

从 express 框架各个文件的调度关系图中可以看到，它首先调用中间件，中间件的作用主要是改写 request, response 请求的。将这 2 个请求导出，方便后面的模板渲染，然后再调用路由模块 route，路由模块只要是根据 path 调用路由来分发函数以及分发路由，最终执行 callback 回调函数。最后调用 view 模块，渲染事先编辑好的模板。

有了 express 框架后，下面的任务就是按照这个框架实现数据可视化的 web 网站。本文所用的 express 框架是 4.0.0 版本的，该版本在代码的组织结构上与以前的版本有所不同，图 5-14 所展示的是本文所设计的可视化网站的代码结构，整个工程的名称为 myboard。

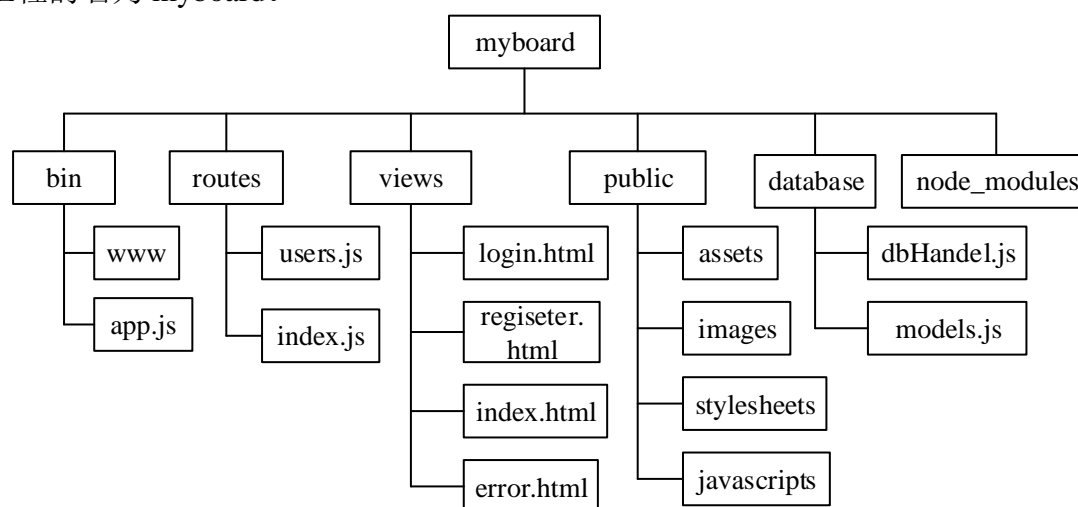


图 5-14 可视化网站的代码结构图

下面简单阐述一下各个文件的功能作用：

- (1) bin/www，最终生成的可执行文件，是整个程序的入口。
- (2) bin/app.js，用于注册各种外部模块，设置 express 的各种属性；在 4.0.0 以前的版本中，是没有 www 文件的，而 app.js 才是真正的程序入口。
- (3) router 是路由模块，主要是进行路由分发，比对，执行 callback 回调函数。
- (4) views 是视图模块，所有的前端界面（即.html 模板）都放在该文件中。其中，在 index.html 中嵌入了 highcharts 的图表显示方法。
- (5) public 用于存放静态资源、前端 js 以及各种页面样式。
- (6) database 数据库建模工具集，系统中用到了 mongodb 来保存用户的登陆注册信息，因此，利用 mongoose.js 来实现与 mongodb 的连接。
- (7) node_modules 是整个工程所有需要的外部模块，也就是依赖包。

5.3.3 数据显示方法的设计

为了使数据显示更加生动、直观，能够一目了然地洞见数据背后的信息。在系统的可视化模块中提供了曲线图、饼图、柱状图、表格等显示方法，同时具有交互式的显示效果。为了支持这些图表的显示，系统采用了第三方插件 `highcharts`^[43,44,45]。`highcharts` 是一个非常流行，界面美观的纯 JavaScript 图表库，可以为网站或 Web 应用程序提供直观，互动式的图表。`highcharts` 和其他许多 JavaScript 库一样，沿用了 `jQuery`、`MooTool`、`Prototype` 等 Javascript 框架来处理基本的 Javascript 任务。因此，在使用 `Highcharts` 之前，需要在页面头部引用这些脚本文件，js 文件可以引入在线的，也可以引入本地的。在配置好 `highcharts` 之后，就可以调用它提供的 `highcharts` 接口来填写图表的数据格式。

在本文所设计的网站访问监控系统中用到了饼图、柱状图以及列表，其中列表是使用 `html` 画出来的，没有用到 `highcharts`，当然在设计可视化模块的时候，考虑到业务的不断扩展，系统中还提供了曲线图、散点图、面积图等基本图形。`highcharts` 的所有图表都包括了基本的配置选项，在填写数据之前就要完成这些选项配置。这些选项都是以 `json` 格式来存储的，正因为如此，本文在流式数据处理过程中所采用的数据格式也是 `json`，这样可以方便数据的展示。下面就以柱状图为例展示这些选项的配置：

```
{
    chart: {type: 'column'},
    title: { text: '错误页面统计'},
    xAxis: {categories: ['**县政府网站', '**州政府网站', '**县规划局', '**县教育局', '**县卫计委']},
    yAxis: {title: {text: 'Population (millions)'},},
    tooltip: {valueSuffix: ' millions'},...
    series: [{data: [107, 31, 635, 203, 2]}]
}
```

下面详细阐述 `json` 对象中的各个字段所代表的意思和作用：

(1) `chart`：图表区、图形区和通用图表配置选项，包括图表类型、背景颜色等信息。

(2) **title**: 设置图表的标题, 包括主标题和副标题, 其中副标题不是必须的。

(3) **tooltip**: 设置数据点提示框的样式, 当鼠标滑过某一点时, 以框的形式提示该点的数据, 比如该点的值、数据单位等信息。

(4) **series**: 数据列, 图表上一个或多个数据系列, 比如图表中的一条曲线, 一个柱形。

(5) **Axis**: 坐标轴, 包括 x 轴和 y 轴。多个不同的数据列可共用同一个 x 轴或 y 轴, 当然, 还可以有两个 x 轴或 y 轴, 分别显示在图表的上下或左右。

图表的数据主要是通过 **series** 字段来表示, 其余的字段是用来描述图表的样式以及图表说明。所有的 **series** 字段的数据是在 **server** 端封装完成后, 通过 **socket.io** 来向前端监控页面推送数据。前端页面运行客户端代码, 监听一个 **socket.io** 的端口, 这样就可以在 **socket.io** 的客户端接收到数据, 然后封装到 **highcharts** 模块中, 保存到 **series** 字段, 就可以绘制出相应的图表, 并及时更新图表上的数据。**socket.io** 实时地从 **server** 端接收数据, 使前端监控页面的图表实时动态变化。

5.4 本章小结

本章主要介绍网站访问监控系统的设计与实现, 首先对系统的实时数据的采集方案以及系统的总体架构进行详细设计, 然后利用基于 **Node-red** 与 **Redis** 的实时流数据处理模型实现了数据的实时分析处理, 并对几个重要指标的统计算法做了详细阐述, 最后阐述了数据可视化模块的设计。

第六章 系统测试与性能分析

评定一个系统的好坏，有两个基本标准，一个就是该系统的功能是否满足需求，另一个就是该系统的性能是否满足需求，本文将基于这两个标准，对所设计的流式处理模型和应用系统展开测试。

6.1 测试条件准备

在进入系统测试与性能分析之前，需要做一些测试前的准备工作，最基本的准备工作就是测试数据与测试平台的准备。

1.测试数据的准备：

系统所用的测试数据，是实际生产线上截取的某政府政务网站群一天的访问流量，已经通过日志的方式保存下来，通过编写一个 `readline.js` 日志读取程序来模拟 `http_tracer` 的功能。目的就是将这些报文数据发布到 Redis 的通道中，再利用 `redisSub` 节点去订阅这些数据。为了对系统进行压力测试，来检验系统的性能与数据的吞吐量，本文将采集到的实时数据分为 5 个独立文件，同时利用 `readline.js` 并发进行，发布到 `http_trace` 通道。这样通过提高数据的输入速度来检验系统的数据吞吐量以及数据的处理和计算能力。图 6-1 展示了实际的部分样本数据。

```
1) "message"
2) "http_trace"
3) "HTTP_TRACE_REQ|1419840348619.121|42.91.9.230:24671|172.16.1.1:18083|#11|HTTP
1.1|POST|/emall/myorder/queryMyOrder.do\n  Accept: application/x-shockwave-fl
ash, image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/QVOD, applicat
ion/QVOD, application/xaml+xml, application/x-ms-xbap, application/x-ms-applicat
ion, application/vnd.ms-xpsdocument, application/msword, application/vnd.ms-exce
l, */*\n  Referer: http://emall.lzbank.com/emall/myorder/queryMyOrder.do\n
Accept-Language: zh-cn\n  User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Wind
ows NT 5.1; Trident/4.0)\n  Accept-Encoding: gzip, deflate\n  If-Modified-Si
nce: Wed, 10 Dec 2014 16:56:06 GMT\n  If-None-Match: W/"956-1418230566000"\n
Host: emall.lzbank.com\n  Connection: Keep-Alive\n  Cookie: JSESSIONID:
9986FBA4B93382AB77946439738F1714|emall_shop_car: \"\"|jiathis_rdc: %7B%22http%3A
//emall.lzbank.com/emall/goods/goodsinfo.do%3Fgoodsid%3D3436814040014%22%3A-1798
085429%2C%22http%3A//emall.lzbank.com/emall/goods/goodsinfo.do%3Fgoodsid%3D34368
14040009%22%3A%229%7C1419840197734%22%2C%22http%3A//emall.lzbank.com/emall/goods
/goodsinfo.do%3Fgoodsid%3D3736814050020%22%3A-1794864601%2C%22http%3A//emall.lzb
ank.com/emall/goods/goodsinfo.do%3Fgoodsid%3D3436814040016%22%3A4%7C141984001820
3%7D|pgv_pvi: 5131678720|pgv_si: s8579460096|searchCategId: %E4%B8%80%E7%BA%A7%E
7%B1%BB%E7%9B%AE%3A235%3A%E8%8C%B6%E9%85%92%E5%86%B2%E9%A5%AE|searchcategtwoid:
%E4%BA%8C%E7%BA%A7%E7%B1%BB%E7%9B%AE%3A312%3A%E4%B8%89%E6%B3%A1%E5%8F%B0|shoppin
gfootprint: 3736814050020_|tokenstr_: ba0ffb0fa783bf37ce96175e30fbde74|tokenstr_
before_login_token: 64de1e6b0a1a466297dc83efe12a158e\n  Content-Type: applicat
ion/x-www-form-urlencoded\n  Content-Length: 207\n  Cache-Control: no-cache"
1) "message"
2) "http_trace"
```

图 6-1 实际系统中的样本数据

2.测试环境的准备:

测试环境是检验一个系统最基本的需求,包括硬件环境和软件环境。

硬件环境:一台装有两台独立虚拟机的PC机,一台虚拟机上部署本文所设计的实时流数据处理模型,另一台部署数据可视化系统也就是本文所设计的myboard。表6-1所展示的是虚拟机的硬件配置信息。

表 6-1 测试的硬件环境

名称	VMware 虚拟机
CPU	Intel Core i7-4720HQ 四核 CPU @ 2.60GHz(2601 MHz)
内存	8.00 GB (1600 MHz)
硬盘	32M 硬盘缓存 1 TB 硬盘存储
网卡	Killer e2200 Gigabit Ethernet Controller (NDIS 6.30)

软件环境:部署实时流数据处理模型最基本的软件需求,以及部署数据可视化网站所需要的基本软件,基本信息显示如表6-2所示:

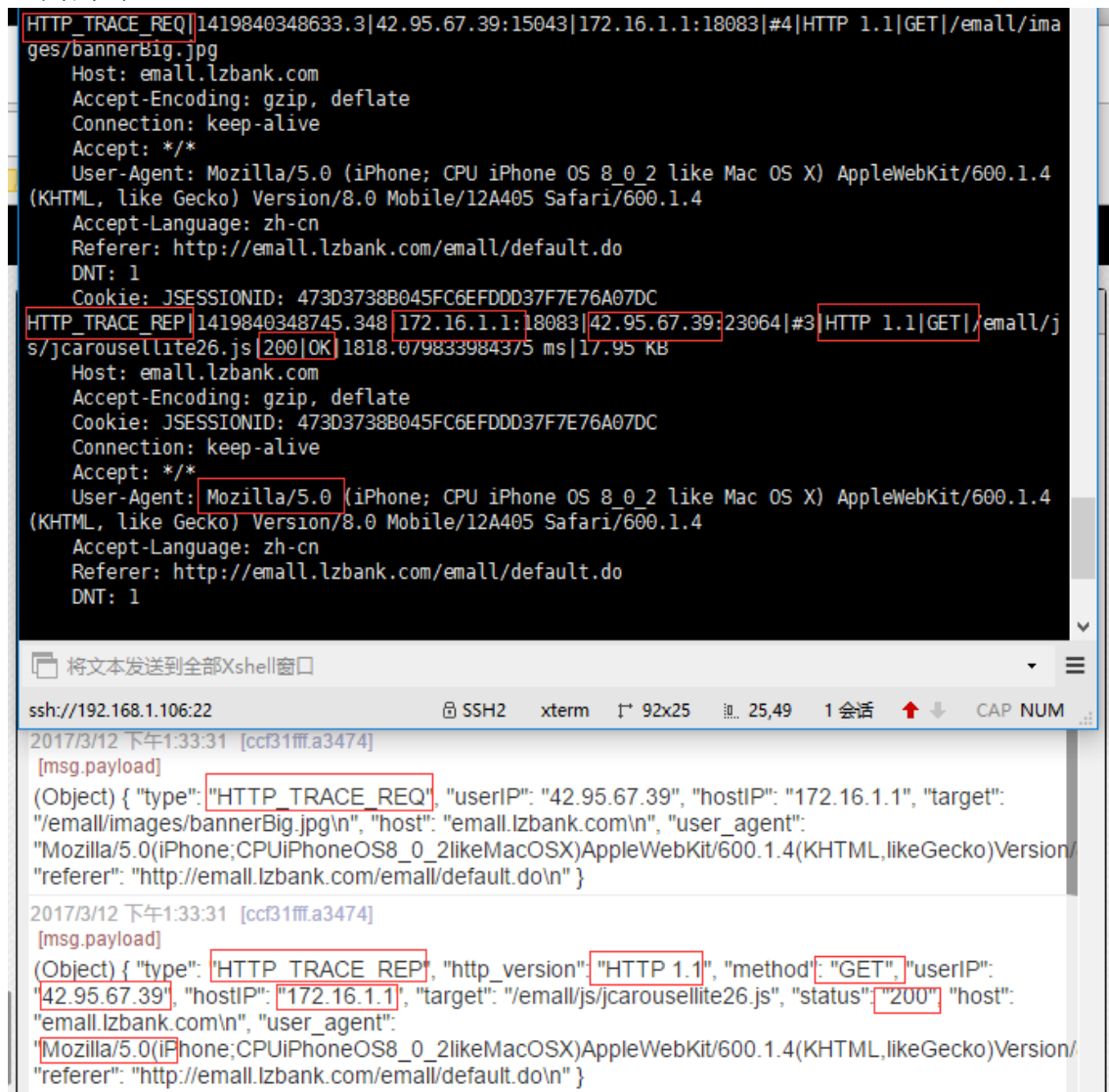
表 6-2 测试的软件环境

机器名	软件名称	软件版本
Core 01 (192.168.1.113)	操作系统	Ubuntu LTS 14.04
	数据库 (active)	Redis 2.8.19
	Node-red	V0.10.6
	Node.js	V5.0.4
	数据库	mongodb2.4.6
	其他外部依赖模块	Socket.io、express、mongos...
Core 02 (192.168.1.114)	操作系统	Ubuntu LTS 14.04
	数据库 (active)	Redis 2.8.19
	myboard (可视化系统)	V1.0
	数据库	mongodb2.4.6
	Node.js	V5.0.4
	其他外部依赖模块	Socket.io、express、mongos...

6.2 系统功能测试

本文所设计的网站监控系统主要提供了用户行为分析和网站群页面监控这两大功能，针对这两大功能展开测试工作。

在进行具体的功能测试之前，首先要测试的是流式数据处理模型的数据分析准确度，Node-red 通过本文设计的 redisSub 节点从 Redis 的 http_trace 通道读取数据后进入数据分析流程，原始数据会在 Node-red 中进行结构化处理，得到 json 对象的数据，结构化处理的正确性直接影响数据分析的结果，所以在 Node-red 中通过 debug 节点显示处理后的数据，与在后台数据采集端的原始数据进行对比，观察是否能够正确的解析出用户关心的数据。图 6-2 展示了原始数据与处理后的数据之间的对比。



```
HTTP TRACE REQ|1419840348633.3|42.95.67.39:15043|172.16.1.1:18083|#4|HTTP 1.1|GET|/emall/images/bannerBig.jpg
Host: emall.lzbank.com
Accept-Encoding: gzip, deflate
Connection: keep-alive
Accept: */*
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 8_0_2 like Mac OS X) AppleWebKit/600.1.4 (KHTML, like Gecko) Version/8.0 Mobile/12A405 Safari/600.1.4
Accept-Language: zh-cn
Referer: http://emall.lzbank.com/emall/default.do
DNT: 1
Cookie: JSESSIONID: 473D3738B045FC6EFDD37F7E76A07DC
HTTP TRACE REP|1419840348745.348|172.16.1.1:18083|42.95.67.39:23064|#3|HTTP 1.1|GET|/emall/js/jcarousel-lite26.js|200|OK|1818.079833984375 ms|17.95 KB
Host: emall.lzbank.com
Accept-Encoding: gzip, deflate
Cookie: JSESSIONID: 473D3738B045FC6EFDD37F7E76A07DC
Connection: keep-alive
Accept: */*
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 8_0_2 like Mac OS X) AppleWebKit/600.1.4 (KHTML, like Gecko) Version/8.0 Mobile/12A405 Safari/600.1.4
Accept-Language: zh-cn
Referer: http://emall.lzbank.com/emall/default.do
DNT: 1
```

```
[msg.payload]
(Object) { "type": "HTTP_TRACE_REQ", "userIP": "42.95.67.39", "hostIP": "172.16.1.1", "target": "/emall/images/bannerBig.jpg\n", "host": "emall.lzbank.com\n", "user_agent": "Mozilla/5.0(iPhone;CPUiPhoneOS8_0_2likeMacOSX)AppleWebKit/600.1.4(KHTML,likeGecko)Version/8.0 Mobile/12A405 Safari/600.1.4", "referer": "http://emall.lzbank.com/emall/default.do\n" }

2017/3/12 下午1:33:31 [ccf31fff.a3474]
[msg.payload]
(Object) { "type": "HTTP_TRACE_REP", "http_version": "HTTP 1.1", "method": "GET", "userIP": "42.95.67.39", "hostIP": "172.16.1.1", "target": "/emall/js/jcarousel-lite26.js", "status": "200", "host": "emall.lzbank.com\n", "user_agent": "Mozilla/5.0(iPhone;CPUiPhoneOS8_0_2likeMacOSX)AppleWebKit/600.1.4(KHTML,likeGecko)Version/8.0 Mobile/12A405 Safari/600.1.4", "referer": "http://emall.lzbank.com/emall/default.do\n" }
```

图 6-2 原始数据与处理后的数据的对比

测试结果表明,数据进入 Node-red 后能够将用户所关心的数据毫无遗漏地、准确地保存下来,为数据分析的准确性提供了保障。测试完模型的数据处理的准确性后,接下来就是对监控系统所提供的各个功能进行测试。

用户行为分析,包括独立访问的 IP 统计、热点页面统计、用户浏览器类型统计以及热门关键词统计。在图 6-3 中分别用柱状图、列表、饼图的方式展示了独立 IP 统计、热点页面统计以及用户的终端类型统计。为了监控系统页面布局的更加美观,把热门关键词统计放在了另外一个面板,其测试结果如图 6-4 所示。

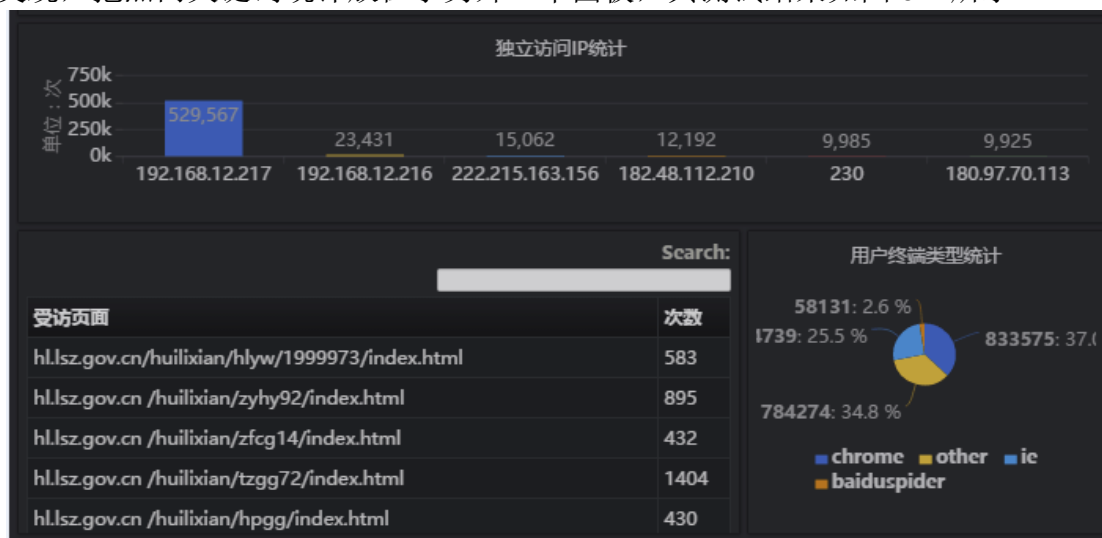


图 6-3 用户行为分析实际测试结果 (A)



图 6-4 用户行为分析实际测试结果 (B)

这里只能给出静态的测试结果,而在实际的生产线上所有的图表会根据用户访问的实际流量进行动态更新。从实际的生产线上的运行结果来看,所有的数据都能够在该政府政务网站群中得到验证,比如,当在该政府的官网上搜索“水库”关键字,就会在监控系统的热门关键词统计列表中发现“水库”的搜索次数由原来的 9 次变为 10 次。这充分说明了本文所设计的实时流数据处理模型进行数据分析的准确性。

网站访问监控系统所提供的第二大功能是网站群页面监控，包括网站访问量统计，错误类型统计，错误网站统计等，测试结果如图 6-5 和图 6-6 所示。

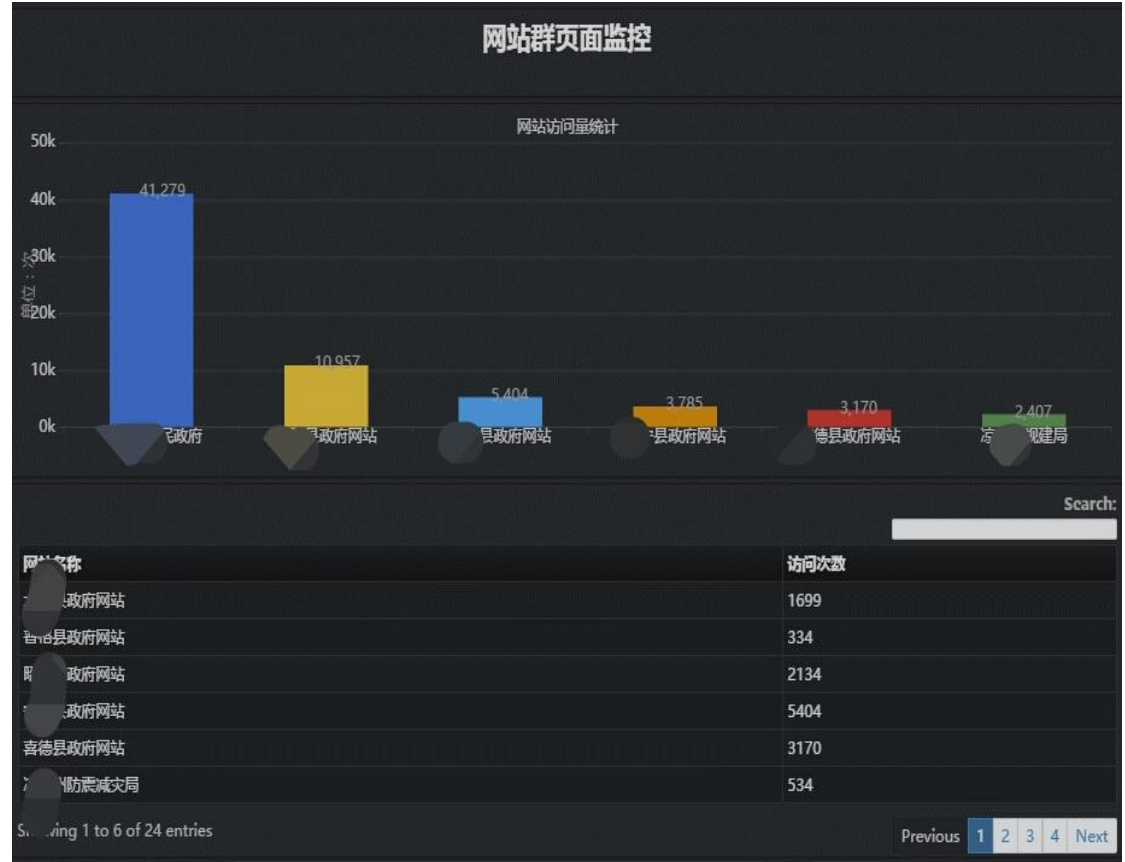


图 6-5 网站群页面监控测试结果（A）

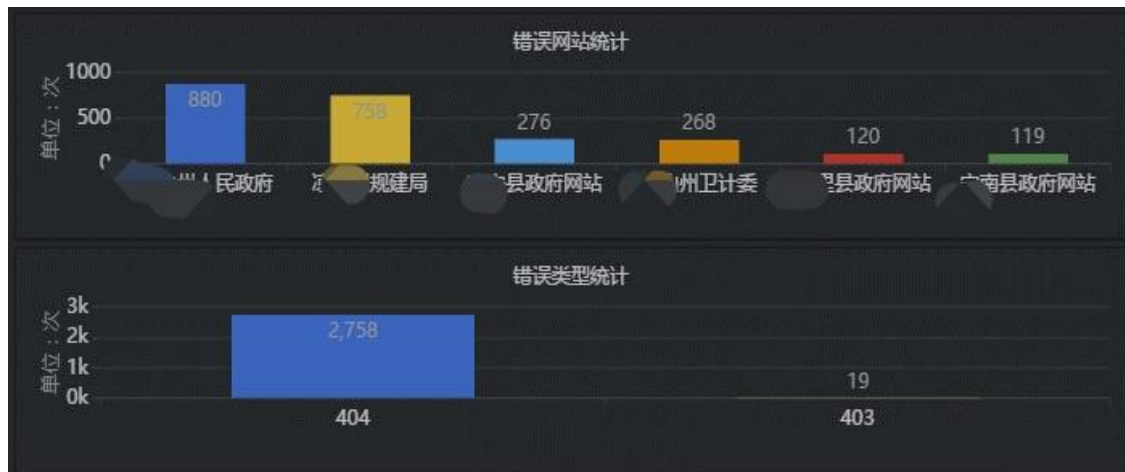


图 6-6 网站群页面监控测试结果（B）

在图 6-5 中以列表和柱状图的方式展示了各个网站的访问量统计情况，图 6-6 中以柱状图的方式展示了错误类型和错误网站的统计情况。由于本论文所用的测

试数据是某政府网站访问的真实数据，可能有些数据比较敏感，而本文为了保证数据的真实有效，最开始没有对数据进行脱敏处理，鉴于对数据的保护，因此就在测试结果中做了部分数据的模糊化处理。

从上面的测试结果可以看出本文所设计的系统，在功能上基本满足用户的需求，下一小节我们将对系统进行性能分析，检测模型在低时延、高并发的条件下的数据吞吐量和数据计算能力。

6.3 系统性能分析

在系统性能分析过程中，我们首先将原始数据报以日志的方式存储到本地硬盘里，通过编写日志读取程序 `readline.js` 来读取日志文件来模拟大量的数据流，将这些数据发布到 Redis 的 `http_trace` 通道中来模拟在实际生产环境中利用 `http_tracer` 工具来截取数据流量，样本数据如图 6-1 所示。读取的方式为按条读取，通过控制消息的条数来观察模型处理数据的性能。性能分析主要分为三个方面：

首先，分析在表 6-1 和表 6-2 所提供的测试平台上，随着报文的发送速率的变化，Redis 数据库内存占用率与 Node-red 的内存占用率的变化情况，测试结果如图 6-7 所示。

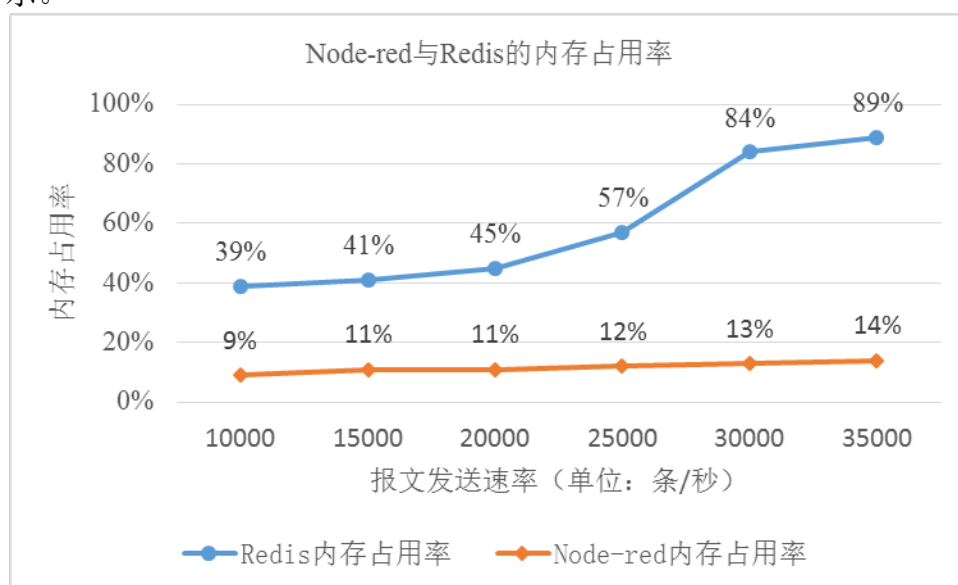


图 6-7 Node-red 与 Redis 内存占用率测试结果

该测试结果也符合预期，当 HTTP 报文发送速率低于 20000 条/秒时，Redis 数据库内存占用率在 40%左右，而 Node-red 的内存占用率只有 10%左右。当 HTTP 报文的到达速率达到 30000 条/秒后，Redis 数据库的内存占用率也急剧增加，而 Node-red 的内存占用率几乎没有什么变化。在报文发送速率超过 35000 条/秒后，

Redis 的内存占用率高达 89%，Node-red 的内存占用率也只有 14%。呈现出这样结果，原因在于，几乎所有的数据处理工作都集中在 Redis 数据库中，而 Node-red 只是数据解析、封装工作以及数据流程的管理工作。

其次，分析数据在实时流数据处理模型中进行统计计算所消耗的时间，其测试结果如图 6-8 所示。

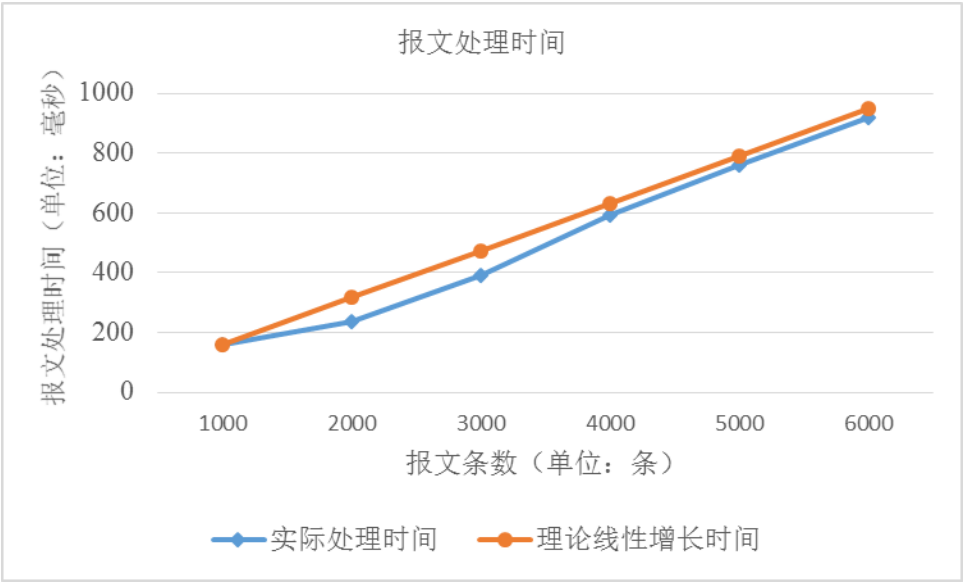


图 6-8 报文实际处理时间与理论处理时间对比分析

通过测试发现，随着消息条数的线性增加，实时流数据处理模型的数据处理时间的增长速度低于线性增长速度。从图 6-8 测试的结果来看，当报文条数达到 4000 条后，数据处理时间就越来越接近线性增长。

最后，分析前端监控页面的数据更新频率和页面的响应时间，随报文条数的增加的变化情况，测试结果如表 6-3 所示。

表 6-3 监控页面响应时间与页面数据更新时间的测试数据

编号	报文数 (条)	处理时间 (ms)	数据推送 频率 (ms)	最长响应 时间 (ms)	页面数据更新 时间 (ms)
1	1000	158	2000	90	2100
2	2000	238	2000	97	2220
3	3000	391	2000	120	2382
4	4000	593	2000	118	2578
5	5000	759	2000	109	2850

6.4 本章小结

本章主要是对系统主要功能测试和性能方面的分析。首先阐述了测试条件以及原始数据的获取，然后分别从功能和性能两个方面对系统进行了测试。测试结果表明，本文所设计的实时流数据处理模型在性能上和数据分析的准确性上能够达到预期效果，本文设计的网站访问监控系统在功能上也满足客户的需求。

第七章 总结与展望

7.1 本文总结

实时流数据处理，已经成为大数据时代下一种重要的计算模式，不论是在学术界还是在商业界，对实时流数据处理的研究和应用都十分广泛，需求也越来越明显。本文是在科研团队与四川省欧润特软件科技有限公司的合作项目中展开叙述的，结合实时流数据处理的相关理论与实际项目的具体需求，设计出一套新的基于 Node-red 与 Redis 的实时流数据处理模型，同时将该模型应用到实际的生产环境中加以验证。对该模型的设计与应用，本文主要完成了以下几方面的工作。

(1) 总结分析了当前流行的实时流数据处理框架，分析了各自的编程模型以及优缺点，同时结合 Node.js 的事件驱动和非阻塞机制对 Node-red 的编程模型做了重点阐述。

(2) 对基于 Redis 有序集合的去重统计方法进行研究，通过分析 Redis 有序集合的源码并结合 Skip List 的基本原理，提出了在实时流数据处理中利用 Redis 有序集合进行去重统计的方法。

(3) 从总体架构上给出了基于 Node-red 与 Redis 的实时流数据处理模型的设计方案，对 Node-red 的节点进行扩展设计，增加了本来没有的 redisSub、redisPub 作为数据输入输出节点，以及 redis_in、redis_out 作为 Redis 数据库访问节点，同时对这些节点加以实现，重新部署安装到 Node-red 中，使其成为一个完整的流式处理模型。

(4) 利用所设计的流式数据处理模型，实现网站访问实时监控系統。对整个系统中各个模块进行设计并对核心指标的统计算法给出了实现。系统包括数据实时采集模块、数据的实时分析模块以及数据可视化模块，其中数据的实时分析模块就是利用本文所设计的实时流数据处理模型实现的。

(5) 本文最后还对该模型和所设计的应用系统进行功能和性能上的测试，以此来验证模型的可行性和有效性。

7.2 对未来工作的展望

本文对 Node-red 与 Redis 做了许多研究和实践工作，设计出了一套新的流式数据计算模型，并将该模型应用到网站访问实时监控系统中。相比于其他流式计

算框架而言，该模型能够快速、便捷地进行数据流程的管理，可以实现业务代码的重用。但是对于大数据背景下的实时流数据计算，该模型仍然有需要改进的地方，所设计的系统也有待完善的地方，但这是一个长期的需要不断坚持的应用研究领域。基于本文的研究，对未来有如下展望：

（1）本文对 Node-red 的节点开发还比较单一，虽然设计了新的数据输入、输出和数据计算节点，但是这些节点进行数据交换的中间桥梁是 Redis 数据库，所以后期的一个重要工作就是对节点的通用化设计和扩展。

（2）本文所设计的模型只是采用了单一节点的 Redis server，没有利用 Redis 集群来解决复杂的流式计算。所以，Redis 集群的引入也是今后工作的一个重点。

（3）在本文所设计的应用系统中，也还存在一些不足。今后的另一项重要工作就是不断地完善该网站访问监控系统，引入更加复杂的流式计算，降低模块之间的耦合程度，完善可视化模块的展示效果，不断提升系统的运行效率。

总之，系统的功能有待丰富，模型的性能也有待更深的优化，今后的工作会研究更有效、更适合的实时流数据处理模型，设计出功能更丰富、性能更优的实时流数据监控系统。

致 谢

岁月易逝，人生中的最后一段学生生涯在不经意间即将结束。伴着这三月春风，回想这三年的生活，山重水复疑无路，柳暗花明又一村。一路走来，既有收获，又有艰辛的付出。三年的科研工作，不仅让我在学术上有了质的提高，也让我学会了如何去感恩。

首先，我要感谢给予我学习平台的电子科技大学以及她的信息与软件工程学院。是这个平台为我提供了更好的教学资源、更强的学习地位和敲响未来之门的敲门砖。这里已成为我的第二个家，我在这里健康自信地成长。

其次，我要诚挚地感谢我的导师刘玓教授，刘教授学识渊博，上善若水。在此门下三年，不仅在科研和学术上对我有了很大的帮助，在生活方面在给予我们很大的关怀。纸上得来终觉浅，绝知此事要躬行，刘教授不仅教育我们丰富的理论知识，还给我们参加大型项目的机会，学以致用。言轻语微，寥寥数语难以表达对刘教授深深地谢意。在此祝愿刘教授健康幸福。

另外，我要感谢实验室团队的文军老师、顾小丰老师、许毅老师等各位老师，感谢几位老师的在科研和论文上的帮助。感谢实验室的同门师兄和室友，在论文内容和结构方面的建议。

同样，我要感谢我的父母和亲朋好友，感谢他们在后面默默地支持，感谢他们永远作为我坚强的后盾。

最后，衷心地感谢论文评阅专家和答辩专家。

参考文献

- [1] Fang W, Pan W b, Ming Z C. View of MapReduce: programming model, methods, and its applications[J]. IETE Technical Review, 2012, 29(5).
- [2] Alessio B, Francesco M, Armando S. A MapReduce solution for associative classification of big data[J]. Information Sciences, 2016, 332.
- [3] Prabhakar B S, Vasavi S, Anupriya P. Hadoop framework for entity resolution within high velocity streams[J]. Procedia Computer Science, 2016, 85.
- [4] Ian P, Ian P. What spark's structured streaming really means[J]. InfoWorld.com, 2016.
- [5] Govind P, Manish K. A framework for fast and efficient cyber security network intrusion detection using apache spark[J]. Procedia Computer Science, 2016, 93.
- [6] 孙大为, 张广艳, 郑纬民. 大数据流式计算: 关键技术及系统实例[J]. 软件学报, 2014(04):839-862.
- [7] 杨定裕. 实时流数据分析的关键技术及应用[D]. 上海交通大学, 2015
- [8] Brodtkin J. IBM touts 'stream computing' for real-time data analysis[J]. Network World (Online), 2009.
- [9] De Z B, Jing T L. research of real-time data presentation of cloud monitoring based on highcharts[J]. Advanced Materials Research, 2015, 3716(1079).
- [10] Jia C L, Wang H, Wei L. Research on visualization of multi-dimensional real-time traffic data stream based on cloud computing[J]. Procedia Engineering, 2016, 137.
- [11] Arkaprava B, Jayneel G, Mark D. Hill, Michael M. Swift. Efficient virtual memory for big memory servers[J]. ACM SIGARCH Computer Architecture News, 2013, 41(3).
- [12] 孙剑斐. 基于大数据处理的 MapReduce 实时优化研究[J]. 电脑知识与技术, 2015(32):199-200.
- [13] Wolfram W, Felix G, Steffen F, et al. Real-time stream processing for big data[J]. it-Information Technology, 2016, 58(4).
- [14] 李庆华, 陈球霞, 蒋盛益. 基于数据流的实时处理框架模型[J]. 计算机工程, 2005(16):59-63.
- [15] Patan R, Rajasekharababu M. Performance improvement of data analysis of IoT applications using Re-Storm in big data stream computing platform[J]. International Journal of Engineering Research in Africa, 2016(22).

- [16] Anonymous. New IBM software uses sensor data to trigger automated business processes[J]. Journal of Transportation, 2009.
- [17] Anonymous. IBM and the internet of things[J]. Electronics Weekly, 2010.
- [18] 柳皓亮, 王丽, 周阳辰. Redis 集群性能测试分析[J]. 微型机与应用, 2016(10):7078.
- [19] 马豫星. Redis 数据库特性分析[J]. 物联网技术, 2015(03):105-106
- [20] Krill, Paul. Node.js inventor extends JavaScript programming beyond browsers[J]. InfoWorld.com, 2012.
- [21] Krill, Paul. The rise of Node.js: JavaScript graduates to the server[J]. InfoWorld.com, 2012.
- [22] Tilkov S, Vinoski S. Node.js: using JavaScript to build high-performance network programs[J]. IEEE Internet Computing, 2010, 14(6).
- [23] 刘德财, 高建华. 基于函数式编程语言的事件驱动模型的设计与实现[J]. 计算机应用与软件, 2016, (09):7-9+37.
- [24] 王心妍. Memcached 和 Redis 在高速缓存方面的应用[J]. 无线互联科技, 2012, (09):8-9
- [25] 王嫣如. Redis 消息推送机制应用技术研究[J]. 科技广场, 2016, (08):41-44.
- [26] Anton V, Uzunov. A survey of security solutions for distributed publish/subscribe systems[J]. Computers & Security, 2016.
- [27] Alex K, Yeung C, Hans A J. Load balancing content-based publish/subscribe systems[J]. ACM Transactions on Computer Systems (TOCS), 2010, 28(4).
- [28] William P. Skip lists: a probabilistic alternative to balanced trees[J]. Communications of the ACM, 1990, 33(6).
- [29] 曲增堂, 滕萍. 网站客户行为模糊监控方法[J]. 网络安全技术与应用, 2011, (11):25-28.
- [30] 郑林江. 基于服务器流量的网站监控系统研究[J]. 信息安全与技术, 2013, (04):76-77.
- [31] 林迅. 基于流量分析实现网站备案监控系统的架构设计[J]. 网络安全技术与应用, 2015, (09):60-62.
- [32] 周国亮, 朱永利, 王桂兰. 实时大数据处理技术在状态监测领域中的应用[J]. 电工技术学报, 2014, (S1):432-437.
- [33] 陈晓天, 陈望都, 王攀. 基于用户行为分析的移动互联网流量经营研究[J]. 电信快报, 2013, (04):11-14.
- [34] 张玉强, 刘生春. 一种基于实时数据采集系统的数据处理方法[J]. 计算机自动测量与控制, 2001, (02):45-47
- [35] 陈丽枫, 郑力新, 王佳斌. 基于 HTML5 WebSocket 的 Web 实时通信机制的研究与实现[J]. 微型机与应用, 2016, (10):88-91.

- [36] Shu M Z, Xiao L X, Jia J L. A real-time web application solution based on Node.js and webSocket[J]. Advanced Materials Research, 2013, 2708(816).
- [37] 吴子然, 李多, 杨争辉. 基于 Node.js 的家庭智能地暖远程监控系统[J]. 计算机科学与应用, 2015, 05(06).
- [38] 谢忠, 颜红霞. JQuery+Highcharts 实现动态统计图[J]. 电脑编程技巧与维护, 2014, (13):64-86.
- [39] 吴孟春, 丁岚. HighCharts 组件在气象业务中的开发和应用[J]. 计算机与网络, 2014, (12):65-68.
- [40] 张浩, 郭灿. 数据可视化技术应用趋势与分类研究[J]. 软件导刊, 2012, (05):169-172.
- [41] 任永功, 于戈. 数据可视化技术的研究与进展[J]. 计算机科学, 2004, (12):92-96.
- [42] Luo G C, Hua W Y, Xian H. A novel web application frame developed by MVC[J]. ACM SIGSOFT Software Engineering Notes, 2003, 28(2).
- [43] 程桂花, 沈炜, 何松林. Node.js 中 Express 框架路由机制的研究[J]. 工业控制计算机, 2016, (08):101-102.
- [44] 崔莹, 刘兵. Node.js 与 Express 技术在计算机课程教学中的应用[J]. 软件导刊, 2016, (09):190-192.
- [45] 叶品菊, 余建平. 基于 HTML5 与 HighCharts 的网页 3D 动画的设计与实现[J]. 黑龙江科技信息, 2015, (29):175.

攻读硕士期间取得的学术成果

科研项目：

- [1] 动漫影视美术服务平台及商业模式研发与应用示范
- [2] Voyager 实时数据分析平台软件二期
- [3] 移动互联身份认证产品框架体系研发及产业化

奖学金：

- [1] 电子科技大学研究生学业二等奖学金. 2014
- [2] 电子科技大学研究生学业二等奖学金. 2015