

题 目 基于微服务架构的统一分析运行平台的设与实现

软件学院 院（系） 软件工程 专业

学 号 71113333

学生姓名 陈浩远

校内导师 刘奇其

企业导师 薄宏剑

起止日期 2017年1月 ～ 2017年6月

设计地点 华为技术有限公司南京研究所

基于微服务架构的统一分析运行平台的设计与实现

# 摘 要

随着电信行业3G/4G等业务的增加，运营商手中的数据量出现了爆炸式的增长，数据的规模已经达到了PB级别。海量数据带来了大量的信息，同时也迫使电信行业为了处理这些数据而不得不使用hadoop/spark等分布式大数据处理技术。由于电信运营商自身结构复杂，产品和业务繁多，电信用户多，每天能产生大量的数据，因此产生了很多不同的大数据任务需求，对于如何管理和调度众多的大数据任务也成为了运行商头疼的一件事。

为了实现对众多任务的统一调度和管理，解决运营商在数据分析中面临着的挑战，本文首先对企业软件的发展现状做了调查，结合对统一分析运行平台的需求分析，设计了该平台的总体方案，对系统的各模块进行了详细的设计，并针对企业平台所需的安全性、容错性、高效性等进行了分析，给出了统一运行平台的实现细节，针对统一运行平台所需的特性进行了测试，最后给出了统一运行平台的部署方案。

关键词：微服务；分布式；大数据；docker

A PAAS Platform Based on Micro Service Architecture

# ABSTRACT

With the telecom industry 3G / 4G and other business increases, the amount of data in the hands of operators there has been explosive growth, the size of the data has reached the PB level. Massive data has brought a lot of information, but also forced the telecommunications industry in order to deal with these data had to use hadoop / spark and other distributed large data processing technology. Because telecom operators have their own complex structure, many products and services, many telecom users can produce a lot of data every day, resulting in a lot of different large data task requirements, how to manage and schedule a large number of large data tasks have become runners A headache for one thing.

In order to realize the unified scheduling and management of many tasks and solve the challenges faced by operators in the data analysis, this paper firstly investigates the development of enterprise software, and analyzes the requirements of the unified analysis and operation platform. The overall program, the system of the modules were designed in detail, and for the enterprise platform for the security, fault tolerance, efficiency and other analysis, given the implementation of a unified platform for the details of the platform for the unified operation of the required The characteristics of the test, and finally gives a unified platform for the deployment of the program.

Keywords: Micro service; distributed; big data; docker

目 录

[摘 要 i](#_Toc484143549)

[ABSTRACT ii](#_Toc484143550)

[第一章 引 言 1](#_Toc484143551)

[1.1 选题背景和意义 1](#_Toc484143552)

[1.2 国内外研究现状 1](#_Toc484143553)

[1.3 论文主要内容 2](#_Toc484143554)

[1.4 论文组织结构 3](#_Toc484143555)

[第二章 系统需求分析 4](#_Toc484143556)

[2.1 系统功能需求 4](#_Toc484143557)

[2.1.1 网络流量采集 5](#_Toc484143558)

[2.1.2 网络流量分析 6](#_Toc484143559)

[2.1.3 流量监测配置 6](#_Toc484143560)

[2.2 系统数据需求 7](#_Toc484143561)

[第三章 系统方案设计 8](#_Toc484143562)

[3.1 系统架构设计 8](#_Toc484143563)

[3.2 网络流量采集功能设计 9](#_Toc484143564)

[3.3 网络流量分析功能设计 12](#_Toc484143565)

[3.4 流量监测配置功能设计 13](#_Toc484143566)

[3.4.1 基础配置 13](#_Toc484143567)

[3.4.2 高级配置 13](#_Toc484143568)

[3.4.3 组配置 14](#_Toc484143569)

[3.5数据库表结构设计 14](#_Toc484143570)

[3.5.1 汇聚数据表结构 14](#_Toc484143571)

[3.5.2 配置数据表结构 15](#_Toc484143572)

[第四章 系统实现与测试 20](#_Toc484143573)

[4.1 网络流量采集功能的实现 20](#_Toc484143574)

[4.1.1 网络流量采集流程 20](#_Toc484143575)

[4.1.2 网络流量接收服务 20](#_Toc484143576)

[4.1.3 网络流量聚合服务 22](#_Toc484143577)

[4.2 网络流量分析功能的实现 24](#_Toc484143578)

[4.3 流量监测配置功能的实现 31](#_Toc484143579)

[4.4 系统测试 35](#_Toc484143580)

[4.4.1 测试目的 35](#_Toc484143581)

[4.4.2 测试环境 36](#_Toc484143582)

[4.4.3 测试过程及结果 36](#_Toc484143583)

[第五章 工作总结与展望 38](#_Toc484143584)

[5.1 工作总结 38](#_Toc484143585)

[5.2 工作展望 38](#_Toc484143586)

[致 谢 39](#_Toc484143587)

[参考文献 40](#_Toc484143588)

# 第一章 绪 论

## 1.1 引言

随着上网用户带宽和流量的增加，运营商对用户产生数据分析和处理的需求也越来越大，本文旨在解决电信行业存在的大量的hadoop和spark任务的调度问题，以分布式的结构，提供一个提供电信分析应用所需的丰富的、高性能的分析引擎，为统一分析开发平台、统一数据治理平台提供统一的运行环境和运维功能。将单独部署的的多种不同的分析引擎，按照运营商自身的业务场景和它们所需的模型进行整合，协助运营商解决以往的运行环境过于简单的问题。

之所以采用微服务，是为了避免传统的分层架构带来的诸多问题，微服务也整合了过去十年来的新概念和技术，因此可以避开许多面向服务的架构中的陷阱。

在多年前，很多组织就开始尝试使用更细粒度的架构来实现更快的交付，结果发现其带来了更好的可扩展性，增强了团度的自治。微服务就是一个这样的快速发展的主题，尽管它不是一个新的想法，但它受了多年来的各种各样的架构的影响，因此，它在当前阶段具有很好的前瞻性和挑战性。

## 1.2 运营商大数据现状

全球信息数据量的迅猛增长成为大数据产业发展的基础。市场调研机构IDC预计，未来全球数据总量年增长率将维持在50%左右，到2020年，全球数据总量将达到40ZB。其中，我国数据量将达到8.6ZB，占全球的21%左右。另外，截至去年12月，我国网民规模达6.88亿人。其中，手机网民规模达6.20亿人，移动互联网流量呈现爆发式增长。工信部统计显示，去年全年移动互联网接入的流量超过400万TB，同比增长了103%。随着全球数据总量以及移动流量的不断提升，运营商的大数据任务不断增加，因此有必要开发一个统一运行平台来对这些任务进行编排和调度。

电信行业迈入数字化精细运营时代，大数据成为使能和引擎核心引擎。大数据既是运营商数字化转型的使能和引擎、同时也是数字化服务的重要内容。在技术上，大数据技术不断地演进和成熟；商业上，数据是运营资产；运营上，大数据作为新思维、新工具可以驱动运营效率的提升和创新；组织上，大数据是组织变革的切入点。

根据Accenture、Gartner趋势分析报告，2014年大数据从技术驱动转变为商业驱动，企业经营优化和数据资产变现成为主要诉求，2017年基于大数据的商业模式创新(Big Data As A Service、数据交易）将会爆发式增长。数据处理技术经过2010-2012年的疯狂发展，将会逐步标准化、云化，成为类似于 RDBMS的通用的IT软件基础设施，但大规模并行计算运维 仍然会成为关键壁垒。

运营商当前在大数据应用时面临如下问题：

* 工具不统一，多厂家异构集成后难以将工具功能集成到端到端解决方案，最大化呈现业务价值。
* 海量电信数据烟囱式分布，难以整合。
* 数据管理分散，难以共享。
* 数据获取过程繁琐，开发困难，对专家依赖高。

具体来说：

* 开发环境：多厂商、多平台造成的多种不同的开发环境，分散了开发资源；关联业务功能/模块被切割，额外增加集成成本；难以形成统一的、相关联的管控机制。
* 运行环境：较多应用运行在不同的环境中，资源利用率不高；跨环境集成降低了运行效率；缺乏统一管理能力。
* 生命周期管理：没有统一的运维管理手段，无法开展多租户管理，面向市场开放能力。
* 调度管理：调度计划难以管理；调度进程难以监控；调度策略难以改进优化。

## 1.3 企业软件架构发展现状

从50-60年代开始，软件技术主要为高级数据/控制结构，具体来说就是函数的重用，为了解决代码的重复出现，这一阶段是编程语言高速发展的阶段。随后70年代，出现了面向结构的架构，这种架构采用自顶向下、逐步求精的设计方法；任何的程序都可以由顺序选择和循环三种控制结构来实现；处理的对象为数值数据。到80年代则出现了面向对象的架构，对象是人们要进行研究的任何事 物，对于同类的事物，可以由一个对象来表示；通过对象抽象，可以大大减轻设 计人员的工作量，简化实际的模型；以数据对象为基础构筑，增加代码可重用性。90年代出现了面向组件的架构，面向组件技术建立在对象技术之上的进一步发展；组件技术的主要目标是粗粒度组件重用；复用性、灵活性大大提高。到04年，SOA已经成为了各大厂商流行的架构风格，SOA引入了服务的概念，它将系统的扩展变得更为灵活，更能响应软件业务的变化，它是基于业务总线ESB的一种架构。而几年来，微服务则成为各大厂商争相关注的对象，相比SOA，微服务不再关注业务总线，它将服务的概念引入了每一个小模块，而且更为依赖网络通信。

## 1.3 本文研究目的和研究内容

本文主要围绕华为的统一分析运行平台的设计和实现，介绍了微服务架构的特性，以及如何通过Spring Cloud实现微服务。该平台解决了运营商对大量任务缺少统一的管理调度系统的问题，根据现有的软件体系结构，解决了企业软件所需的高性能、高容错、高安全性的需求，保障了系统的长期稳定的运行，并且对以后的开发和维护有着较好的支持性。

## 1.4 论文组织结构

本论文主要介绍了基于Spring Cloud的统一分析运行平台的设计与实现，主要内容组织结构如下：

第一章 绪论，通过对运营商现状和软件结构的发展，介绍了该论文的研究目的和研究内容。

第二章 系统需求分析，。 //TODO

第三章 系统设计，。

第四章 系统实现与测试，

第五章 工作总结与展望，对论文内容进行了总结，以及对系统当前不足提出的一些日后可以实行的优化改进建议。

# 第二章 系统需求分析

## 2.1 系统功能需求

随着电信业务的不断拓展，电信资费越来越低，电信行业产生了越来越多的数据，传统使用人工单独调用大数据任务的方法已经不能满足电信行业所需的大量任务，大量的任务存在与不同的机器上，使得对任务的管理变得越来越困难，大量的机器上运行着早已不再需要的任务，严重浪费了机器的性能和管理人员的精力。为了提高对机器的利用率，降低人力成本，运营商急需一个可以统一管理和调度的平台。

对于统一分析运行平台，需要有业务编排、上下文、按照业务流程进行运行、调用其他服务的功能，在这里我们将多个业务通过编排后组成的业务流称之为任务模板，对于任务模板，统一运行平台要有运行多个任务模板的能力，并且要有对任务模板的管理功能。

### 2.1.1 业务编排

将多个业务按照一定的逻辑和顺序编排在一起。这是统一分析运行平台的关键性功能，他可以将多个业务组合在一起，相比以往需要运维人员逐个执行的业务在这里可以轻松的一次性执行。有了这个功能就可以更好的利用运算平台的性能，并且提供相比人工而言更复杂的管理功能。

要求业务编排需要实现以下的功能：

* 顺序执行：以串行的方式按照编排的顺序逐个执行业务，只有当前一个业务结束才可以开始下一个业务。如图2.1所示，这张图展示由一个人饥饿引发的简单过程。结果是必须有人去购买材料和做饭，之后有人吃了饭，他的饥饿会得到满足。这些业务之间的关系就是顺序执行的关系。

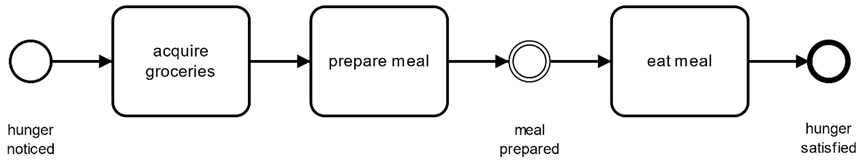


图2.1饥饿过程的顺序执行

* 条件执行：在当前业务后面跟着两个分支，在执行完当前业务之后，判断该处的条件，如果满足条件则接下来进入一个分支，如果不满足条件则进入另一个分支。如图2.2所示，这是一个在线商店从收到订单到用户收到物品之间的一个装货送货的过程，在这中间有很多个条件执行的业务，例如在第一个有效的业务中，决定是要使用普通方式配送还是特殊方式配送的方式，如果满足普通配送的条件则向上进入普通配送的流程，如果条件不满足普通配送的条件则向右进入特殊配送的流程。

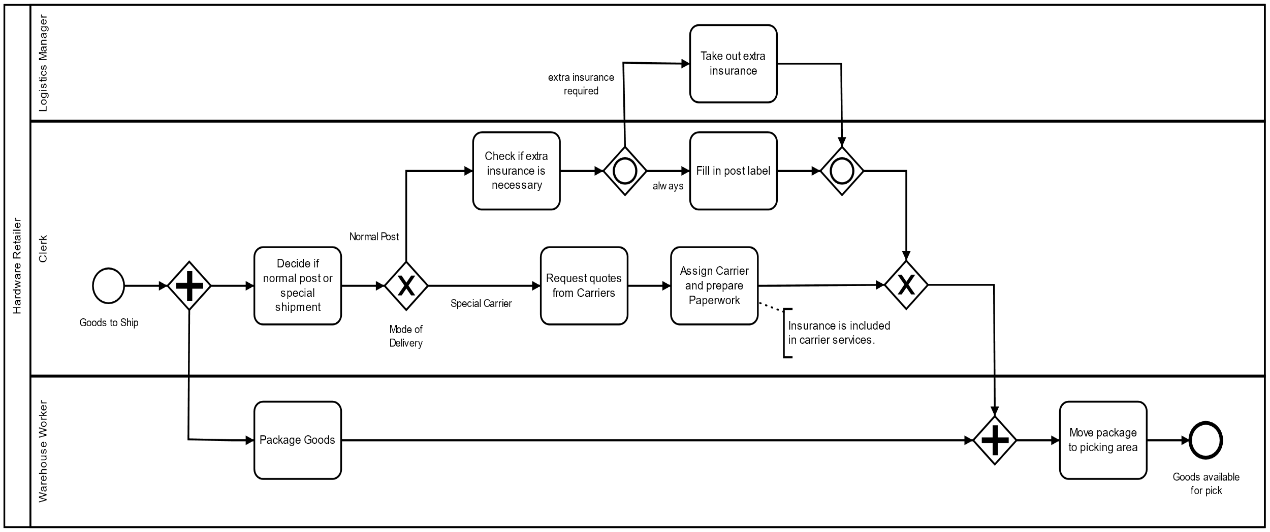


图2.2一个装货过程

* 并行执行：多个业务在同一时刻一起执行，处于并行关系的两个业务之间并不需要互相等待，他们可以一同执行。
* 定时事件：某个业务可能并不是按照顺序或者条件的逻辑作为开始标记，而是以一个时间作为自己的开始条件。如图2.3所示，这是烤蛋糕的过程，执行完检查蛋糕是否做好的业务后进入一个条件判断，如果条件判断为没有做好则定时五分钟后再执行下一个业务。

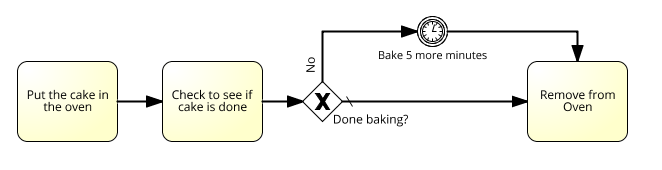


图2.3一个烤蛋糕的过程

对于业务编排之后形成的东西，我们称之为任务模板，统一分析运行平台需要有对任务模板的管理功能，具体分为：创建任务模板、修改任务模板、删除任务模板、获取全部任务模板、单独查询任务模板。

### 2.1.2 上下文

因为有时候并不止是单纯的把任务模板丢到引擎中开始执行，部分任务模板需要有一些环境变量才可以执行，有些业务可能需要传入一定的参数，例如：有些特定的hadoop业务需要在运行时指定结果的存放路径；有些业务有多种执行模式，需要依据环境变量才能判断按照哪种模式执行。

统一分析运行平台提供一个上下文的机制，可以向上下文环境中传入任意的参数，这些参数能够让引擎中的业务在执行的过程中获取，并且要对外提供一个创建上下文、修改上下文、查询上下文以及删除上下文的功能。

### 2.1.3 任务实例管理

在统一分析运行平台中，任务模板是有多个业务组成的一个业务流，而任务实例则是任务模板的一个实例化，在只有一个业务需要执行的情况下，我们也要求它先组成一个任务模板，实例化为任务实例后才能放入引擎中去运行。因此，任务实例就是统一分析运行平台的最小执行单元。

因为统一分析运行平台中会同时运行大量的任务实例，因此需要有对任务实例的管理功能。具体分为：创建任务实例、修改任务实例、删除任务实例、获取全部任务实例、单独查询任务实例。

## 2.2 系统特性需求

随着软件架构的发展，企业软件逐渐从单体应用向分布式的应用过渡，对于企业软件的各种特性要求也越来越多。针对统一运行平台而言，由于该平台是大量业务的管理调度平台，因此该平台对企业有着很高的重要性，并且该平台也会承担大量的数据访问。对于该平台，我们对他的安全性、容错性、高效性、扩展性、部署难度等都有较高的要求。

### 2.2.1 安全性

随着时间的推移，越来越多的大型系统被曝出安全漏洞，这些漏洞往往会导致很多严重而又危险的事情。例如，棱镜计划就是美国国家安全局存在着的安全漏洞，最终导致了整个国家的安全都受到威胁。统一运行平台对公司来说同样是极其重要的一个平台，因此我们在设计系统时一定要尽可能地保证他的安全性。具体需求如下：

* 身份认证和授权：对于整个系统，我们要求它拥有认证和授权机制，并且可以对不同的用户设置不同的权限。对于分布式系统的单个节点，我们要求该节点只对需要直接交互的其他节点进行认证，其余外在系统和平台内不与其直接交互的节点认证均不通过（即进行细粒度的授权）。
* 数据安全：虽然授权机制能在一定程度上保证统一运行平台的安全性，但是一旦操作系统被黑客攻破，我们的敏感数据和密钥都面临着巨大的威胁，因此，我们要求对密钥和敏感信息进行加密处理。
* 防火墙：华为要求所开发的软件必须明确其开放端口，对于不需要的端口我们会全部关闭，以保证系统没有后门也没有多余的端口暴露漏洞。
* 日志：系统要求对所有涉及到数据和登陆的操作都有日志记录。完善的日志虽然起不到预防攻击的作用，但是可以在发生难以预测的事情时重现当时的情景。同样，我们需要在日志中剔除敏感信息。
* 网络隔离：对于分布式系统而言，并不是所有的节点都需要同样的网络环境，我们可以通过将节点分配到不同的网段，这可以进一步避免内部节点的暴露，提高整个系统的安全性。
* 操作系统：操作系统要求选用打好安全漏洞布丁的操作系统，并关闭操作系统的不必要的服务，定期检查该操作系统曝出的安全漏洞，在操作系统出现漏洞的情况下及时做出响应。

### 2.2.2 容错性

分布式系统相比单体系统而言，其中一个重要的有点就是他可以对系统的容错性进行很好的设计和控制。系统的容错性是指当系统发生故障时，系统还能保证运行和数据安全，并且可以从错误中恢复过来，其中的故障可能是系统和软件在运行中出现的运行时错误，也可能是机房的硬件故障，甚至是断网断电的情况。对统一分析运行平台来说，要保证部分节点出故障的情况下，平台还能保证功能的完整性，对于故障节点所涉及到的数据可以完整地保存起来，以备之后的恢复。

### 2.2.3 高效性

这里的高效性是指对整个系统的性能需求，是这个系统要能够承受预估范围内的访问数据量。对于统一分析运行平台而言，其在交付后可能会同时运行上万个的任务。我们需要在能预知大概数据量的情况下，不对机器做过多的浪费。

### 2.2.4 扩展性

分布式系统相比单体应用而言，他可以针对特定的模块进行多个扩展。要求统一分析运行平台的多个模块能够按照所需的性能进行扩展，并且要求可以在运行时对模块进行扩展，扩展之后的模块要在一分钟内能够加入系统并分担一部分数据量。

### 2.2.4 部署难度

初次部署要求提供完整详细的文档和脚本，要求系统易于部署。而在之后需要修改平台代码的情况下，我们只需要重新部署这行代码所涉及到的这个节点。

# 第三章 系统设计

## 3.1 系统架构设计

传统的单体结构的应用很难应对每秒上万次的调用，对于应用的容错性也很难保障，应用内一旦出现一个致命性的错误将导致整个应用的停止运行，这对运行商来说将会带来巨大的损失，基于微服务架构的应用可以完美的解决这一问题。

微服务就是采用网络来进行分割模块，每个模块都是一个小的服务，这些小服务彼此间只通过网络进行交互，单独的微服务完全拥有自治的能力，因此微服务的架构可以带来：

* 技术异构性：因为服务之间通过网络进行协作，除此之外，服务之间完全独立，因此可以在不同的服务内使用不同的框架，甚至是不同的语言，为了达到微服务的正确运行，只需要统一服务间的网络协议和数据格式。但是，在一个组织内使用过多的语言也会导致组织难以招聘人才和找不到合适的人交接工作。
* 弹性：在单块系统中，如果一个组件不能用，整个系统都会受到影响，但是在微服务中，我们可以轻易的将一个服务部署成分布式的。
* 扩展：在单体的软件中，想要实现分布式的扩展，则必须将整个系统进行扩展，这意味着及时系统中只有一个小模块存在性能问题，我们也要浪费大量的机器去扩展它。而微服务具有天生的可扩展性，可以针对需要扩展的微服务进行扩展。
* 简化部署：在大型的单体程序中，即使只修改了一行代码，也需要把整个程序进行重新部署，但是在微服务中，我们只需要重新部署这行代码所涉及到的这个微服务即可。
* 与组织结构相匹配：微服务可以有效的将模块分离出来，这也使得整个项目组团队可以分割成在各地的小型团队，服务的所有权也可以随意的在团队间转移，能够保证所有相近服务的团队都在一起工作。
* 可组合性：微服务最大的好处就是可以对各个服务进行编排重组，不同于以往的单纯的Web、PC端程序，人们有更多的选择使用同一个功能。微服务中会开放很多接口供外部调用，当情况发生改变时，可以使用不同的方式构建应用。
* 对于可替代性的优化：在单块的系统中，我们想要修改一写历史遗留的问题时，常常要思考这次修改所需要付出的代价有多大，因此，跟多公司可能还在维护者很多年前就该淘汰的代码，这些代码没人能看得懂，甚至每人敢动他。在微服务上，我们基本可以保证一个服务可以在两周时间内重写，这也就意味着我们想要替换掉这个服务时所付出的代价并不是那么大。

微服务之间的完全通过网络进行了解耦，服务之间的联系就如图3.1所示。

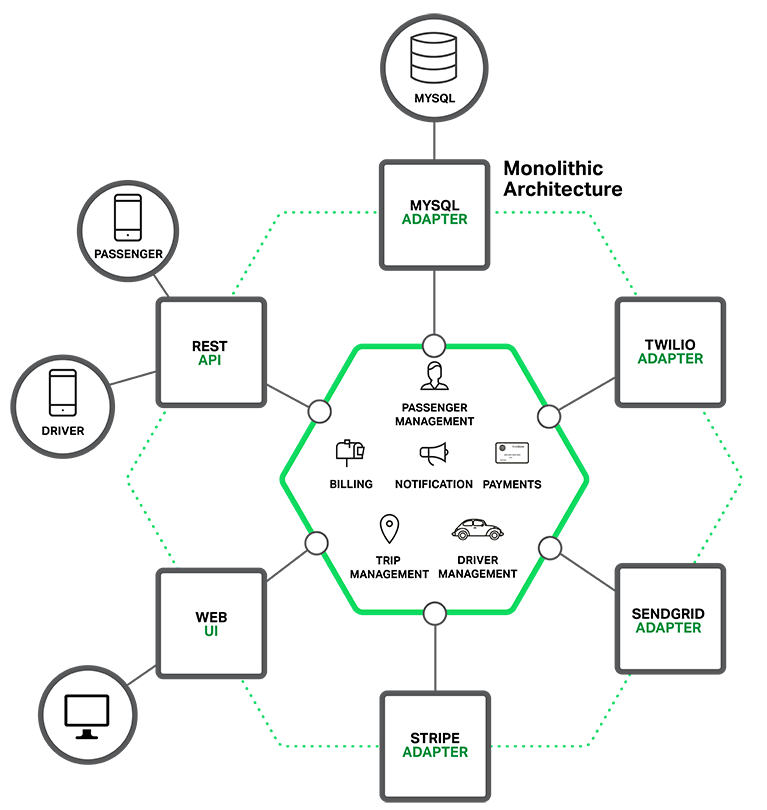


图3.1 微服务之间的联系

综上所述，统一分析运行平台采用微服务架构作为系统架构。

在统一分析运行平台中，我们的架构入图3.2所示，划分为任务模板模块、任务实例模块、BPMN引擎模块、Ignite模块和Eureka模块五个模块，其中负责和用户交互的模块为任务模板模块和任务实例模块，对于调用相应业务的功能由BPMN引擎模块完成。

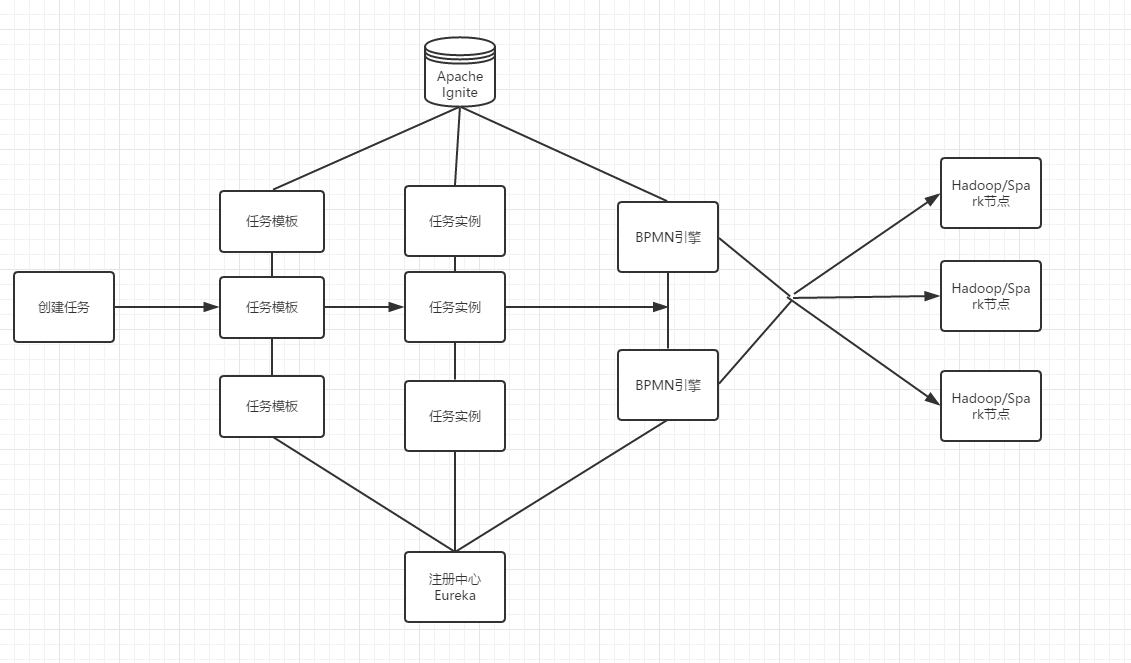


图3.2 系统整体架构

## 3.2 服务交互

对于微服务而言，服务之间通过网络进行交互，常采用的方式为RPC或者REST。下文将通过对比RPC和REST的优缺点，确定在统一分析运行平台中所选择的交互方式。

### 3.2.1 RPC

在分布式计算中，远程过程调用(RPC)是当一个计算机程序引发另一个地址上的程序执行,并且在这一过程中没有涉及到程序的代码细节,就好像它是一个正常的本地调用,无需程序员显式编码远程交互的细节。也就是程序员不管是远程调用还是本地调用都写相同的代码。这是一个典型的client-server交互，调用者是客户端，执行者是服务器，通常通过请求-相应消息传递系统来实现。

RPC的具体实现有很多，如JAVA的RMI、或者通过暴露一个SOAP接口来实现，不同的RPC实现所用的消息格式不同，使用的技术栈也不同，但是作为调用的双方则必须使用相同的格式和技术。

因此，RPC可能带来技术上的耦合，比如RMI只能用于JAVA的平台，因此这就要求被调用者和调用者的双方都选择在JAVA平台上开发，这就对技术选型造成了一定的限制。另一方面，由于RPC像本地调用一样，所以想要修改一个方法则必须对本地和远程的代码都进行修改，这是开发过程中C/S端耦合的一部分。

另外，RPC虽然用起来和本地调用一样，因为RPC将远程调用的中间过程给隐藏起来了，但是这也带来了很多不确定性，比如性能上，不同的RPC调用的性能不同，而且还有网络通信所需要的时间，这就可能导致不明真相的开发人员把远程调用当然本地调用来使用，最后造成性能上的问题，甚至会由于网络的问题而出现各种各样的错误。

### 3.2.2 REST

REST是Representational State Transfer的缩写，它是由Roy Thomas Fielding博士在2000年提出来的一种架构风格。Roy Fielding也是http规范的主要作者之一。

REST是一种设计风格，REST本身有很多的约束和限制，但在这里，我们仅将他和RPC进行对比。REST的关键点为：

* 资源与URI：任何被引用到的事物都可以看作一个资源，例如在URI中，我们的每一个地址都可以看作一个资源，这个地址的命名可以很好的描述这是一个怎样的资源。
* 统一资源接口：这个接口预先定义了可能对资源的操作，放在http里面则对应着http的动词，例如：GET一般表明获取资源，它是安全的，它的操作是幂等的；POST表明更新资源，它意味着将要去修改服务器上的资源，它的操作不是幂等的；PUT一般表明要去创建一个资源，它是不安全的，但是是幂等的；DELETE则是要删除一个资源，是不安全的，但是是幂等的。
* 资源的表述：资源的表述基本上是指用怎样的数据格式去形容这个资源，对外表现为资源的具体呈现，用来进行传递资源。比如文本我们可以使用json或xml，图片我们可以使用PNG或JPG展现，如图3.3所示，这是一个使用json作为资源表述格式的rest请求。



图3.3 http请求

* 资源的链接：这一点奠定了互联网下一代获取资源的方式，就是我们的下一个要访问的资源链接都可以在上一个资源处找到，这也就是超媒体即应用状态引擎的概念，但是目前我们的微服务应用还不能很好的实现这一点。
* 状态的转移：REST是一种无状态通信原则，他在服务端并不会有状态，服务端只维护资源，而客户端维护状态，状态转移就是由客户端维护如何进入下一个状态。

HTTP实现了很多REST的功能，因此HTTP通常会用于构建RESTful软件，我们只需要在使用HTTP时按照REST对资源的定义来实现，就可以基本地实现符合REST风格的软件。

另外HTTP与RPC并不冲突，RPC可以通过HTTP来实现，例如SOAP，但是这并没有完全用到HTTP的统一资源接口等功能。

### 3.2.3 RPC和REST的选择

由上可知，REST的功能和易用性都比RPC要强，但是我们不能说RPC一无是处，一方面部分RPC框架可以自动的生成桩代码，这使得使用RPC的工作量将要比REST小一些，另一方面，基于http的REST的每次请求都需要完好的封装http协议，而http是建立在TCP上层的协议，而且很多的http客户端框架并不能很好的实现http的长连接，这就导致在一次http请求之后，TCP连接会断开，而再次请求还需要重新建立http请求，这就使得在有低时延需求的情景下，http比RPC的时延可能要高。

在系统开发之前，我们针对REST和RPC进行了一些相关的测试，测试方法如下：

* 选择相同的配置的四台机器作为测试机，其中两台用来测试REST的接收和发送，另外两台测试RPC的接收和发送。
* 使用Java RMI实现RPC；使用jersey实现rest服务端，通过封装JAVA内置类HttpURLConnection实现rest客户端；RPC和REST服务端均只在接受到调用后累加器加一。
* 分别为RPC和REST编写测试函数，对两种调用方式的用时进行统计。

RPC客户端代码如下：

publicclass Program {

publicstaticvoid main(String[] args){

try{

//调用远程对象，注意RMI路径与接口必须与服务器配置一致

RPCService rPCService=(RPCService)Naming.lookup("rmi://10.163.14.19:6600/RPCTest");

int counter;

long start = System.currentTimeMillis();

for(int I;I <= 20000;i++){

counter = rPCService.add();

}

long end = System.currentTimeMillis();

System.out.println("Took : " + ((end - start) / 1000));

}

}catch(Exception ex){

ex.printStackTrace();

}

}

}

RPC服务端代码如下：

//此为远程对象的实现类，须继承UnicastRemoteObject

publicclass RPCServiceImpl extends UnicastRemoteObject implements RPCService {

public RPCServiceImpl() throws RemoteException {

counter = 0;

super();

}

int counter;

@Override

Int add() throws RemoteException {

counter++;

return counter;

}

}

其中，在客户端代码中对RPC服务端进行了20000次调用，并统计了20000次调用的时间。

REST服务端代码：

@Path({"test "})

public class RestTest

{ @GET

@Path({"add"})

public int doGet(

)

{

counter++;

return counter;

}

}

REST客户端代码：

URL url = new URL("http:// 10.163.14.31:8080/test/add");

URLConnection rulConnection = url.openConnection();

HttpURLConnection httpUrlConnection = (HttpURLConnection) rulConnection;

httpUrlConnection.setRequestMethod("GET");

httpUrlConnection.setDoOutput(true);

httpUrlConnection.setDoInput(true);

httpUrlConnection.setUseCaches(false);

java.io.EOFException)

httpUrlConnection.setRequestProperty("Content-type", "application/x-java-serialized-object");

httpUrlConnection.connect();

OutputStream outStrm = httpUrlConnection.getOutputStream();

int counter;

long start = System.currentTimeMillis();

for(int I;I <= 20000;i++){

ObjectOutputStream objOutputStrm = new ObjectOutputStream(outStrm);

objOutputStrm.writeObject();

objOutputStm.flush();

objOutputStm.close();

}

long end = System.currentTimeMillis();

System.out.println("Took : " + ((end - start) / 1000));

最终测试结果如表3.1所示，其中RPC和REST均交换机器再次测试了一次，由表可以看出，以这种方式实现的RPC比REST的性能稍好，但是两者的差距并不是很大。

表3.1 RPC和REST性能测试

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **测试次数** | **总用时** | **平均时间** |
| RPC | 20000 | 18.860s | 0.943ms |
| REST | 20000 | 20.076s | 1.0038ms |
| RPC | 20000 | 17.779s | 0.88895ms |
| REST | 20000 | 20.430s | 1.0215ms |

因此，我建议，在通常的微服务开发中，选择REST风格的模式，这样更利于开发和维护；而在有低延迟需求的微服务开发中选用RPC来做通信。在本文所介绍的统一分析运行平台中，低延迟并不是我们所要保证的重点内容，因此我们选择更为规范的REST作为我们服务间的交互方式，同时选择json作为交互内容的载体。

## 3.3 微服务模块划分

为了实现业务编排、上下文和task管理的功能性需求，将系统的模块划分：任务模板模块、任务实例模块、调度引擎模块、Ignite模块和Eureka模块。

### 3.3.1 任务模板模块

任务模板模块负责对任务模板的管理，正如前面需求分析中提到的需求，我们再任务模板模块中实现这些需求：创建任务模板、修改任务模板、删除任务模板、获取全部任务模板、单独查询任务模板。

该模块对外开放这些rest接口作为交互的方式：

## 3.3 网络流量分析功能设计

网络管理控制台（NMC）接收到流量采集器NTC的1分钟聚合数据后进行数据的入库，系统建立10分钟、1小时、6小时、1天、1周等粒度的聚合任务，聚合任务会定时将数据入库保存，不同维度的数据提供不同的存储策略，同时这些数据还可从数据库中转储成数据文件后永久保存下来。详细的汇聚规则如表3.2所示。

表3.2 详细汇聚规则

|  |  |  |  |
| --- | --- | --- | --- |
| **汇聚粒度** | **汇聚执行周期** | **汇聚基础数据** | **数据保存时间** |
| 1分钟数据 | 每分钟执行 | 原始数据 | 1天 |
| 10分钟数据 | 每10分钟执行 | 1分钟数据 | 10天 |
| 1小时数据 | 每小时执行 | 10分钟数据 | 1月 |
| 6小时数据 | 每6小时执行 | 1小时数据 | 3月 |
| 1天数据 | 每天执行 | 6小时数据 | 1年 |
| 1周数据 | 每周执行 | 1天数据 | 3年 |

网络管理控制台以图形化的方式提供多维度、多层次的流量分析报表。既可以直观显示全网接口、应用、主机等统计对象的Top N排行，又可以针对单个统计对象进行精细化分析，并且能准确查询到原始流量的具体信息，达到流量可视化的目的。

流量分析维度包含设备维度、接口维度、应用维度、主机维度、接口组维度、应用组维度、IP组维度等，每种维度都是先展现该统计维度下的Top N排行，选择一个具体统计对象后，再展现该统计对象的详细流量趋势及各相关流量数据的组成。

流量分析首页Dashboard流量概览界面从多维度显示Top N流量信息，包括Top N接口利用率、Top N设备流量、Top N应用流量、Top N接口流量等等，全网流量动态直观显示。并且用户可以定制操控版显示的维度信息来更方便的查看需要显示的流量信息。

如果用户需要查看某个维度更加具体的信息Dashboard流量概览菜单下分单个维度，进入单个维度的Top N流量排行，从全网角度展现该维度流量、数据包统计信息及流量趋势。单个维度的流量详细统计，包括流量、数据包双坐标趋势图及相关联流量信息排行，直观展现流量的组成。

对于Top N数据单条数据，进入单条数据信息界面，提供该条数据流量详细统计，在Top N数据的基础上增加了单条数据相对应的应用、接口、主机等等维度的流量组成，细化观察单条流量的组成。

系统还提供原始流量报表功能，展现最完整的流量数据信息，方便客户定位网络故障。

## 3.4 流量监测配置功能设计

控制台NMC通过图形化界面的方式给用户提供对设备及其监控信息的管理操作。网络流量监测配置分为三部分：基础配置、高级配置和组配置。其中，基础配置是实现对设备监控的第一步，基础配置提供采集器的管理、设备的添加、接口的监控以及采样比下发等操作；高级配置可以根据用户根据实际需要，进行更深层次的管理，比如协议配置以及应用管理以及流量阈值告警配置；组配置主要是为了方便用户将某些有关联的项进行分类，主要包括接口组、应用组、IP组。

### 3.4.1 基础配置

基础配置界面提供采集器的管理功能，对于采集器NTC与控制台NMC同机安装的情景下，安装之后默认添加本地采集器，添加的设备默认绑定到这个采集器，对于控制台NMC与采集器NTC分机安装的情况，则需要用户在安装采集器时绑定控制台的IP地址。

* 采集器配置界面用户可以查看采集器的状态和IP地址等信息，也可以设置采集接口的TOP N数目，系统默认为30个。同时打开采集器的流量取证开关，则采集器将会吧原始流文件上传到NMC控制台。
* 设备管理模块则会显示出采集器监控的全网上有流量上报的设备，用户可以选择性的对设备进行监控、删除、去监控等操作。
* 接口配置界面展示了向NMC控制台发送NetStream报文的设备接口，用户可对接口流入速率、流出速率和采样比进行配置，保证网流流量数据的正确性。

### 3.4.2 高级配置

* 协议配置：用户根据实际需要，可以选择性地对某些协议进行监控。
* 应用配置：列举了常用的543个网络应用，分为预定义应用和用户自定义应用。其中，预定义应用指的是预置的应用和设备识别并上报的应用；用户自定义应用指用户添加的应用，根据指定的协议(UDP/TCP)、端口范围和IP范围来定义应用。

### 3.4.3 组配置

* IP组配置：用户可将有关联的一组IP地址分为一组，如一个部门或一个楼层，方便查看该IP组的流量信息。
* 应用组配置：用户可将按照自己关注的点进行应用分类，如邮件类应用组，便于查看该应用组的流量信息。
* 接口组配置：用户可将相关联的接口定义成一个接口组，方便查看接口组流量信息。

## 3.5数据库表结构设计

本系统需要存入数据库的数据均在NMC端，其大致分为两种类型：一种存储接口、应用、IP组等维度的汇聚数据，另一种存储网络流量配置功能的设备、接口、采集器、应用、协议以及组配置等信息。

### 3.5.1 汇聚数据表结构

由于每个维度不同时间粒度的表结构相同，只是存储的数据不同，所以此处只列举出1分钟粒度的各个维度汇聚表结构。

**（1）应用维度**

应用维度的1分钟汇聚表名称为TBL\_FS\_APP1MIN，其表结构如表3.3所示。

表3.3 应用维度汇聚表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| ID | bigint (19) | 数据唯一编码，自增长，主键 |
| DEVICEID | integer (10) | 设备ID |
| INTERFACEID | integer (10) | 接口ID |
| FLOW\_DIRECTION | integer (1) | 流方向 |
| APPID | bigint (19) | 应用ID |
| BYTES | bigint (19) | 字节数 |
| PACKETS | bigint (19) | 包数 |
| FS\_TIME | bigint(19) | 流量准确发生时间 |

**（2）接口维度**

接口维度的1分钟汇聚表名称为TBL\_FS\_IF1MIN，其表结构如表3.4所示。

表3.4 接口维度汇聚表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| ID | bigint (19) | 数据唯一编码，自增长，主键 |
| DEVICEID | integer (10) | 设备ID |
| INTERFACEID | integer (10) | 接口ID |
| FLOW\_DIRECTION | integer (1) | 流方向 |
| AVG\_BYTES | bigint (19) | 平均字节数 |
| MIN\_BYTES | bigint (19) | 最小字节数 |
| MAX\_BYTES | bigint (19) | 最大字节数 |
| BYTES | bigint (19) | 字节数 |
| PACKETS | bigint (19) | 包数 |
| FS\_TIME | bigint(19) | 流量准确发生时间 |

**（3）主机维度**

主机维度的1分钟汇聚表名称为TBL\_FS\_HOST1MIN，其表结构如表3.5所示。

表3.5 主机维度汇聚表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| ID | bigint (19) | 数据唯一编码，自增长，主键 |
| DEVICEID | integer (10) | 设备ID |
| INTERFACEID | integer (10) | 接口ID |
| FLOW\_DIRECTION | integer (1) | 流方向 |
| AVG\_BYTES | bigint (19) | 平均字节数 |
| MIN\_BYTES | bigint (19) | 最小字节数 |
| MAX\_BYTES | bigint (19) | 最大字节数 |
| BYTES | bigint (19) | 字节数 |
| PACKETS | bigint (19) | 包数 |
| FS\_TIME | bigint(19) | 流量准确发生时间 |

对于接口组、应用组、IP组的汇聚数据，数据的存放还是分别在接口、应用、主机汇聚表内，只是在查询分析信息时取出每个组绑定的所有单个对象的数据进行汇总显示，这样可以节省汇聚表的数据量，减轻NMC的压力。

### 配置数据表结构

**（1）设备配置**

设备配置的数据库表名为TBL\_NAS\_DEVICE，其表结构如表3.6所示。

表3.6 设备配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| DEVICEID | integer (10) | 设备ID，自增长，主键 |
| DEVICE\_NAME | varchar(255) | 设备名称 |
| DEVICE\_IPADDR | varchar(255) | 设备IP地址 |
| DEVICE\_MACADDR | varchar(50) | 设备物理地址 |
| DEVICE\_DESCR | varchar(255) | 设备描述 |
| REFRESH\_TIME | bigint (19) | 刷新时间 |
| CREATE\_TIME | bigint (19) | 创建时间 |
| IS\_CONFIG | integer(1) | 配置标识符，默认0 |
| IS\_MANAGE | integer(1) | 是否监控，默认1 |
| SYSOID | varchar(255) | 系统对象标识 |
| DIS\_FLAG | integer(1) | 是否在界面显示，默认1 |

**（2）接口配置**

接口配置的数据库表名为TBL\_NAS\_INTERFACE，其表结构如表3.7所示。

表3.7 接口配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| INTERFACEID | integer (10) | 接口ID，自增长，主键 |
| DEVICEID | integer (10) | 设备ID |
| IF\_INDEX | integer(10) | 接口索引 |
| IF\_NAME | varchar(255) | 接口名称 |
| INTERFACE\_NAME | varchar(255) | 接口别名 |
| IF\_DESC | varchar(255) | 接口描述 |
| IN\_BAND\_WIDTH | bigint(19) | 流入带宽 |
| OUT\_BAND\_WIDTH | bigint(19) | 流出带宽 |
| UNMONITOR | integer(1) | 是否监控 |
| UNMONITOR\_STARTTIME | bigint (19) | 不监控的起始时间 |
| UNMONITOR\_ENDTIME | bigint(19) | 不监控的终止时间 |
| SAMPLE\_RATE | integer(10) | 接口采样比 |
| IS\_CONFIG | integer(1) | 配置标识符，默认0 |
| DIS\_FLAG | integer(1) | 是否显示标识，默认1 |

**（3）采集器配置**

采集器配置的数据库表名为TBL\_NAS\_COLLECTOR，其表结构如表3.8所示。

表3.8 采集器配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| COLLECTID | integer (10) | 采集器ID，自增长，主键 |
| NTC\_IP | varchar(64) | 采集器IP地址 |
| NTC\_NAME | varchar(50) | 采集器名称 |
| NTC\_DESC | varchar(255) | 采集器 |
| NTC\_PORT | varchar(50) | 采集器端口，默认9996 |
| BUFFER\_SIZE | integer(10) | 缓存大小，默认102400 |
| THREAD\_NUM | integer(10) | 线程数，默认8 |
| NTA\_SERVERIP | varchar(255) | NMC控制台IP地址 |
| NTA\_HTTPS\_PORT | bigint (19) | NMC控制台HTTPS端口，默认443 |
| FILE\_SEND\_PORT | integer(10) | 文件传输端口 |
| TIMEZONE | varchar(50) | 时区 |
| STARTUPTIME | bigint(19) | 启动时间 |
| COLLECT\_STATE | integer(1) | 采集器状态，默认1 |
| TRANSFER\_RAW\_FILE | integer(1) | 是否发送原始流文件，默认1 |

**（4）协议配置**

协议配置的数据库表名为TBL\_NAS\_PROTOCOL，其表结构如表3.9所示。

表3.9 协议配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| PROTOCOLID | integer (10) | 协议ID，主键 |
| PROTOCOL\_NAME | varchar(50) | 协议名称 |
| PROTOCOL\_DESC | varchar(255) | 协议描述 |
| MONITOR\_STATE | integer (1) | 监控状态 |
| REF\_RFC | varchar(50) | 标准化协议详细说明 |

**（5）应用配置**

应用配置的数据库表名为TBL\_NAS\_APPLICATION，其表结构如表3.10所示。

表3.10 应用配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| APPID | integer (10) | 应用ID，主键 |
| APP\_NAME | varchar(100) | 应用名称 |
| APP\_DESC | varchar(255) | 应用描述 |
| PROTOCOL\_ID | integer (10) | 协议ID |
| IS\_SYSDEF | integer (1) | 是否为系统预定义，默认1 |
| MONITOR\_STATE | integer (1) | 监控状态 |

**（6）应用组配置**

应用组配置的数据库由两张表组成，分别为TBL\_NAS\_APPGRP和TBL\_NAS\_APPGRPMAP。其中TBL\_NAS\_APPGRP表结构如表3.11所示，TBL\_NAS\_APPGRPMAP表结构如表3.12所示。

表3.11 TBL\_NAS\_APPGRP配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| APPGRPID | integer (10) | 应用组ID，自增长，主键 |
| APPGRPNAME | varchar(100) | 应用组名称 |
| APPGRPDESC | varchar(255) | 应用组描述 |

表3.12 TBL\_NAS\_APPGRPMAP配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| APPGRPMAPID | integer (10) | 应用组MAP ID，自增长，主键 |
| APPGRPID | integer (10) | 应用组ID |
| APPID | integer (10) | 应用ID |

**（7）接口组配置**

接口组配置的数据库由两张表组成，分别为TBL\_NAS\_IFGROUP和TBL\_NAS\_IFGRPMAP。其中TBL\_NAS\_IFGROUP表结构如表3.13所示，TBL\_NAS\_IFGRPMAP表结构如表3.14所示。

表3.13 TBL\_NAS\_IFGROUP配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| IFGROUPID | integer (10) | 接口组ID，自增长，主键 |
| IFGRPNAME | varchar(100) | 接口组名称 |
| IFGRPDESC | varchar(255) | 接口组描述 |

表3.14 TBL\_NAS\_IFGRPMAP配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| ID | integer (10) | 接口组MAP ID，自增长，主键 |
| IFGROUPID | integer (10) | 接口组ID |
| INTERFACEID | integer (10) | 接口ID |

**（8）IP组配置**

IP组配置的数据库由两张表组成，分别为TBL\_NAS\_IPGROUP和TBL\_NAS\_IPGROUPRANGE。其中TBL\_NAS\_IPGROUP表结构如表3.15所示，TBL\_NAS\_IPGROUPRANGE表结构如表3.16所示。

表3.15 TBL\_NAS\_IPGROUP配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| IPGROUPID | integer (10) | 接口组ID，自增长，主键 |
| GROUPNAME | varchar(100) | IP组名称 |
| GROUPDESC | varchar(255) | IP组描述 |
| IPGROUPSTATE | integer(1) | IP组监控状态 |
| IPBPS | bigint(19) | 每秒比特率 |

表3.16 TBL\_NAS\_IPGROUPRANGE配置表结构

|  |  |  |
| --- | --- | --- |
| **字段名称** | **类型** | **含义** |
| IPRANGEID | integer (10) | IP组范围ID，自增长，主键 |
| IPGROUPID | integer (10) | IP组ID |
| RANGETYPE | integer (1) | 网络类型 |
| STARTIP\_ADDR | bigint(19) | 起始IP |
| ENDIP\_ADDR | bigint(19) | 结束IP地址 |
| SUBNET\_MASK | bigint(19) | 子网掩码 |
| NETWORK\_MASK | integer(10) | 掩码位 |

# 第四章 系统实现与测试

## 4.1 网络流量采集功能的实现

网络流量采集的主要功能是负责分析NDE输出数据包，并对对输出的数据进行适配解析，并对解析好的数据进行聚合、过滤和封装然后发送到网络管理控制台NMC进行分析处理。其中NDE设备对流量的采样、统计、老化、输出均在设备上实现，NTC实现对NDE设备输出的报文解析以及多维度聚合功能。

网络流量采集器作为一个应用部署在网络流量监测系统中，有集中式部署和分布式部署两种方式，可以全量安装网络管理控制台NMC和网络流量采集器NTC，也可以只安装NTC，分布式安装采集器时需要绑定可以正常使用的网络管理控制台的IP地址，否则系统无法安装。

### 4.1.1 网络流量采集流程

采集器启动后，对各个子服务进行初始化，首先是对接受服务的初始化：

this.receiverService = new ReceiverService();

接受服务的初始化主要是建立流量包空队列flowPacketQueue、创建流的适配器flowAdapterMgr以及初始化流量监听服务flowListeners，其中流量适配器主要区分NetStream流统计的版本格式。之后将初始化好的接受服务赋值给聚合服务进行初始化：

this.aggregationService = new AggregationService(this.receiverService);

然后判断NTC绑定的网络管理控制台NMC部分是否处于运行状态，如果控制台部分没有运行，则采集器启动终止。若NMC正在运行则进行各维度缓存模型的下载初始化以及接受聚合服务的启动。代码架构如下：

private static void initCache(AggregationService aggregationService, ReceiverService receiverService)

{

CollectorCache.getInstance().initListenPorts();

DeviceCache.getInstance().loadDevice();

InterfaceCache.getInstance().loadInterface();

ApplicationCache.getInstance().loadApplication();

ProtocolCache.getInstance().loadProtocol();

IpgroupCache.getInstance().loadIpGroup();

IpGroupToIpGroupCache.getInstance().loadIpgroupToIpgroup();

aggregationService.startService();

receiverService.startService();

}

### 4.1.2 网络流量接收服务

在聚合服务启动之初，先创建接收服务的流量监听器，其中每一种维度的流量类型作为一种HandlerType，每一种类型有自己独立的流量监听函数，具体实现如下：

private List<IFlowListener> createListeners()

{

List<IFlowListener> flowListeners = new LinkedList();

FlowListener listener = new FlowListener(HandlerType.RawFlow);//原始流类型

flowListeners.add(listener);

FlowListener intlistener = new FlowListener(HandlerType.Interface);//接口类型

flowListeners.add(intlistener);

if (Configuration.getInstance().isAppAggr())

{ FlowListener applistener = new FlowListener(HandlerType.Application);//应用类型

flowListeners.add(applistener); }

this.aggrFlowListeners.addAll(flowListeners);

return flowListeners;

}

**（1）报文版本格式解读**

接收服务启动时，首先检测监听端口是否有效，无效则终止服务，并抛出异常，记录日志信息。如果有效，则启动UDP报文解析线程，接收到的UDP报文通过buff的字节头获取该流的版本号，实现中的parseDatagramPacket函数将根据该版本号进行不同版本的解析，解析返回不同版本的数据包，再将该数据包加入到队列中等待下一步的处理，该线程从当前步骤直到采集器进程结束一直会运行，具体解析实现如下：

ByteBuffer buf = (ByteBuffer)key.attachment();

buf.clear();

SocketAddress inetsa = ((DatagramChannel)key.channel()).receive(buf);

if (inetsa != null) {

if (ReceiveSwitcher.isOpen())

{ buf.flip();

int bufsize = buf.limit();

byte[] bytarr = new byte[bufsize];

buf.get(bytarr);

DatagramPacket p = new DatagramPacket(bytarr, bufsize, inetsa);

this.counter += 1L;

//检测报文版本

byte[] buff = p.getData();

short version = (short)(int)Util.byteArray2Long(buff, 1, 1);

switch (version)

{ case 5:

packet = this.flowAdapterMgr.parseDatagramPacket(p, "netflow\_v5");

break;

case 9:

packet = this.flowAdapterMgr.parseDatagramPacket(p, "netflow\_v9");

break; }

if (packet != null)

{ this.flowPacketQueue.offer(packet);

synchronized (this.flowPacketQueue)

{ this.flowPacketQueue.notifyAll(); } }

}

}

**（2）流量包解码**

在流量包的报文格式确定之后，接下来就是根据报文格式将数据包转换成对应版本的流，此线程与报文解析线程一样，从接受服务运行开始到采集器结束都会运行。其中的process函数将流量包转换成对应报文格式版本的流。主要实现如下：

IFlowPacket packet = (IFlowPacket)this.flowPacketQueue.poll();

List<IFlow> flowList = packet.process();//将流量包转换成对应版本的流

boolean flowlistvalid = (flowList != null) && (!flowList.isEmpty());

if (flowlistvalid)

{ for (IFlowListener listener : this.listeners) {

listener.handleFlow(packet.getRouterIP(), flowList);//将转换好的流加入阻塞队列等待聚合}

}

在每个维度的监听器的handleFlow 函数内主要讲转换好的流列表加入到监听器的阻塞队列中等待各个监听器的聚合处理：

this.flowBlockingQueue.offer(new FlowListenerObject(ip, flowList));

### 4.1.3 网络流量聚合服务

聚合服务启动前，先创建各个维度的流量监听线程，并将它们加到接收服务与聚合服务的流量监听列表中，然后将流量监听线程列表的各个线程启动，聚合服务的流量监听线程从聚合服务开启到采集器结束运行之间一直运行。

每个维度分为三层：处理层（Handler），负责对每条流进行提取、分割；持久层（Holder），进行同种流的聚合处理，以及输出统计数据的字符串拼接；导出层（Exporter），将处理层的保存的流量缓存写入文件并将此文件加入压缩文件map中等待压缩线程对文件进行打包压缩。

**（1）处理层（Handler）**

其间监听线程接收服务处理完成的流阻塞队列，根据根据每个维度的handlerType进行不同的聚合处理，具体实现如下：

AbstractHandler handler = null;

FlowListenerObject listenerObject = this.flowBlockingQueue.take();

if ((listenerObject.flowList != null)

&& (!listenerObject.flowList.isEmpty()))

{ if (this.handlerType != null) {

handler = HandlerType.getHandler(

listenerObject.ip, this.handlerType);

} else if (this.extHandlerName != null) {

handler = HandlerType.getHandler(

listenerObject.ip, this.extHandlerName); }

if (handler != null) {

handler.process(listenerObject.flowList);//处理流进程 }

}

每个维度handler的process函数处理机制类似，以下以应用维度的process函数实现举例，process函数的主要功能是将流列表中每条流提取并拷贝自身并将这两条流按照出入方向进行分割，此举主要是方便控制台NMC处理在带接口特殊方向条件下的复制出来的流，复制出来的流出方向字段值为3，对于应用统计维度，如果分析条件没有接口条件，则需要去除方向为3的流量；如果有接口条件，则需要带上方向为3的流量，保证一个接口下统计的流量是准确的，否则会导致应用维度流量翻倍。其具体实现如下：

public void process(List<IFlow> flowList)

{ if (flowList == null) {

return; }

synchronized (this.lockObj)

{ for (IFlow flow : flowList)

{ List<IFlow> splitedFlows = flow.split(true);//对流的复制分割

if (splitedFlows != null) {

for (IFlow splitFlow : splitedFlows)

{ Map<Object, AbstractHolder> cache = getCache(splitFlow);

addToCache(splitFlow, cache);//进行流的聚合处理

}

}

}

}

}

//如果缓存中没有这条流，则直接将该流添加到缓存里，如果存在该流，则进行聚合处理

private void addToCache(IFlow flow, Map<Object, AbstractHolder> cache)

{

int key = flow.getAppID();

if (cache.containsKey(Integer.valueOf(key)))

{

AbstractHolder origin = (AbstractHolder)cache.get(Integer.valueOf(key));

origin.aggregate(flow);

}

else

{

cache.put(Integer.valueOf(key), new AppTrafficHolder(flow));

this.appFlowCount += 1;

}

}

**（2）持久层（Holder）**

持久层对应的处理层的aggregate具体操作是：

this.totalDOctets += flow.getdOctets();

this.totalDPkts += flow.getdPkts();

主要作用是对同种流的总字节数和总报数进行聚合。而输出文件字符串拼接函数getAggregateFlowString实现如下：

public String getAggregateFlowString(long flowSummaryTime)

{

StringBuffer appSB = new StringBuffer();

appSB.append(this.flow.getDeviceID());

appSB.append("|");

appSB.append(this.flow.getInterfaceID());

appSB.append("|");

appSB.append(flowSummaryTime);

appSB.append("|");

appSB.append(this.flow.getAppID());

appSB.append("|");

appSB.append(this.flow.getDirection());

appSB.append("|");

appSB.append(this.totalDOctets);

appSB.append("|");

appSB.append(this.totalDPkts);

return appSB.toString();

}

**（3）导出层（Exporter）**

导出层的主要功能是将处理层保存的流量缓存写入文件，并将此文件加入压缩线程的压缩文件map中等待压缩线程对文件进行打包压缩。具体实现如下：

List<AbstractHolder> appFlowList = getFlowList();

if (appFlowList == null) {

return; }

this.flowCount = appFlowList.size();

File saveFile = ExportUtil.createAppFile(this.routerIP, this.dumpTime, "APP");

ExportUtil.writeFile(appFlowList, saveFile, false, this.dumpTime);//写入文件

ZipThread.setZipFileListMap("APP", saveFile.getCanonicalPath());//把文件加入压缩map

采集器启动时会开启一个基于SFTP的文件传输Runnable去定时的将采集器NTC的聚合统计数据文件传输到控制台NMC端。

## 4.2 网络流量分析功能的实现

网络流量分析功能提供全网流量概览，以图形化的方式提供多维度多层次的流量分析报表，实时展现全网流量动态，用户可快速查看到各维度的流量信息。既可以直观显示全网接口、应用、主机等统计对象的Top N排行，又可以针对单个统计对象进行精细化分析，并且能准确查询到原始流量的具体信息。

控制台NMC启动后会启动一个DumpingThread线程，该线程每隔三十秒扫描一次汇聚数据文件夹，对新发送来的压缩文件进行解压入库，逻辑实现大体如下：

File pathFile = new File(this.path);//文件路径

File[] files = null;

synchronized (lockObjMap.get(this.name))

{ //取出文件夹内的文件列表

files = pathFile.listFiles(new DumpFileFilter(this.filter));

Arrays.sort(files, new FileModifyTimeComparator());

IFileCollector collector = new FileCollectorImpl();

IFileDumper dumper = new FileDumperImpl();

//解压并将数据入库

unzipAndDump(files, collector, dumper);

}

每个维度分别有1MIN、10MIN、1HOUR、6HOUR、1DAY、1WEEK六张汇聚表用来存储不同时间内的基础数据。其中，1MIN时间粒度的是由NTC流量采集器发送过来入库，剩下五种汇聚表均在上一种汇聚时间维度进行更长时间的汇聚，控制台会启动一个定时任务AggrTask定时进行聚合，由于不同维度的聚合配置类似，以下使用应用维度的聚合配置举例，其聚合属性配置在xml文件中，具体为下：

<!--具体聚合类型-->

<aggregation tableprefix="TBL\_FS\_APP" from="1MIN" to="10MIN" >

<attr name="tominunit" value="600000"/>

<attr name="from" value="1MIN"/>

<attr name="to" value="10MIN"/>

<attr name="base" value="0"/>

</aggregation>

<aggregation tableprefix="TBL\_FS\_APP" from="10MIN" to="1HOUR" >

<attr name="tominunit" value="3600000"/>

<attr name="from" value="10MIN"/>

<attr name="to" value="1HOUR"/>

<attr name="base" value="0"/>

</aggregation>

<aggregation tableprefix="TBL\_FS\_APP" from="1HOUR" to="6HOUR" >

<attr name="tominunit" value="21600000"/>

<attr name="from" value="1HOUR"/>

<attr name="to" value="6HOUR"/>

<attr name="base" value="3600000"/>

</aggregation>

<aggregation tableprefix="TBL\_FS\_APP" from="6HOUR" to="1DAY" >

<attr name="tominunit" value="86400000"/>

<attr name="from" value="6HOUR"/>

<attr name="to" value="1DAY"/>

<attr name="base" value="3600000"/>

</aggregation>

其对应的聚合SQL也配置在xml文件中，如图4.1所示。

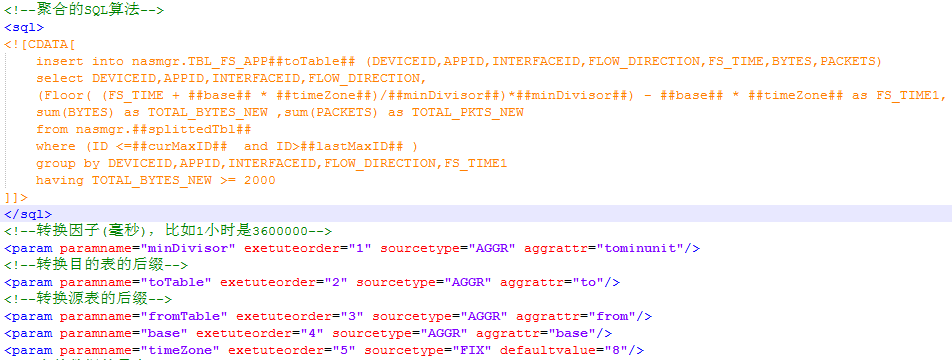


图4.1 聚合SQL算法及其SQL参数对应的聚合属性

网络流量分析界面图形统一使用一套widget框架模板，由于每个界面使用固定的画图控件固定，所以为了方便数据的查询与存储，在控制台NMC安装的时候则对每个控件初始化好固定的widgetid，根据前台参数生成查询条件filter，根据查询条件通过调用后台ds层的restful服务接口根据widgetid调用对应的namingsql获取图表数据并返回前台显示。以一种控件的查询函数举例，代码逻辑实现大致如下：

protected WidgetData getTotalData(List<WidgetData> topnData, Map<String, Object> conMap)

{

//用户设置的查询条件

List<WidgetCustomValue> customValues =

this.widgetObj.getChartDataType().getCumstomValue();

WidgetData resultData = null;

//根据widgetinfo生成的查询map调用sql查询数据

Map<String, Object> totalData =

WidgetDAO.getInstance().queryTotal(this.widgetSql.getTotalSql().getSql(),

conMap, customValues);

resultData = this.dataProcessor.constructTotalData(totalData, this.widgetObj);

return resultData;

}

其对应的后台sql配置在xml文件中，如图4.2所示。

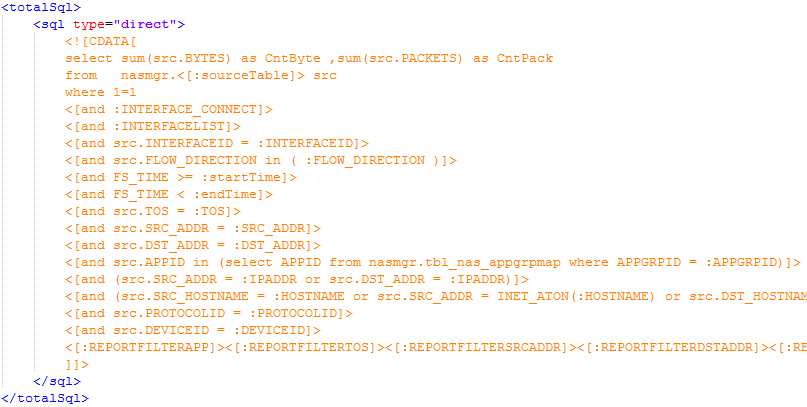


图4.2 查询全部数据sql

前台使用公司内部eview组件进行画图，图4.3是当chartType为35即折线图时的画图组件定义结构：

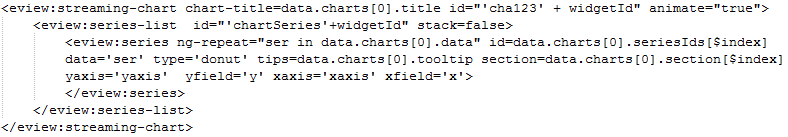


图4.3折线图控件定义结构

网络流量分析界面包括三种画图控件，分别是圆饼图、折线图、区域图。圆饼图展现的是全部流量的类型百分比，应用维度饼图如图4.4所示，展示了全部应用流量各种协议所占比例，图中显示了显示了TOP 5的流量排行，TOP之外的都会被分类到other里面，用户可以自定义需要显示的TOP N个数来查看更多比例分配。



图4.4 应用维度饼图

折线图展现出某个对象的流量趋势图，图4.5显示了某个接口近十五分钟的流量趋势，下方可以提供此段时间内的最大、最小、平均速率以及流量信息的查看。用户可以自定义时间段方便查看更多时间段内的流量趋势状况。

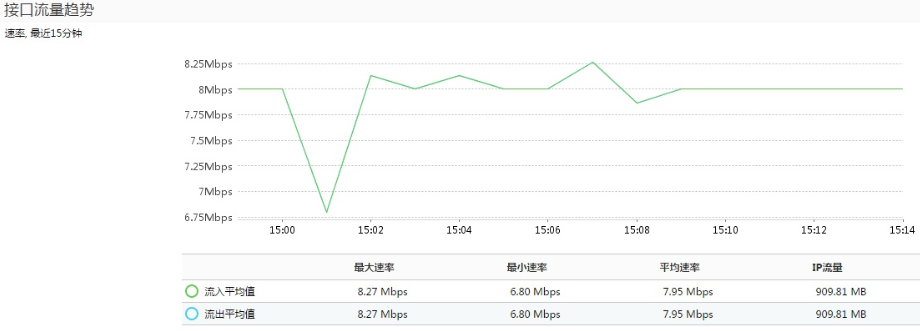


图4.5 接口流量趋势图

区域图显示出某个对象在某段时间内的TOP N流量组成，图4.6显示了某主机在某时间段内的流量组成，默认显示TOP 5，同样支持自定义TOP N显示数目。

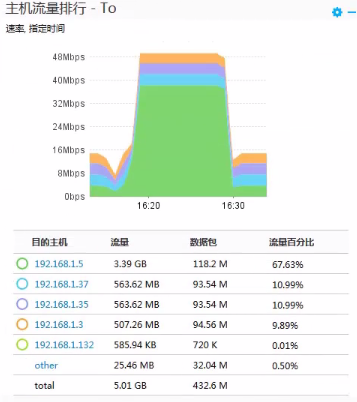


图4.6 主机流量区域图

这三种组件根据不同场景进行组合，来展现多维度多层次的网络流量分析。在网络流量配置模块添加了设备以及网络中可以监控到流的前提下，网络流量分析的流量概览Dashboard界面如图4.7所示。该界面显示了全网流量的大致分布，包括各种维度的流量比例组成，其中接口维度还提供速率和利用率的概览，但是不提供具体维度的详细流量组成信息，当需要查看某个维度的具体流量信息时则需要点击某个维度的分析界面，接口维度相比其他维度多一个接口分析列表界面，缺少对应维度的TOP N流量趋势界面，而对于其他界面框架基本一致，只是显示的文字和数据不同，所以以下以接口维度的具体流量分析界面以及其他维度的TOP N流量趋势界面相结合进行展示。

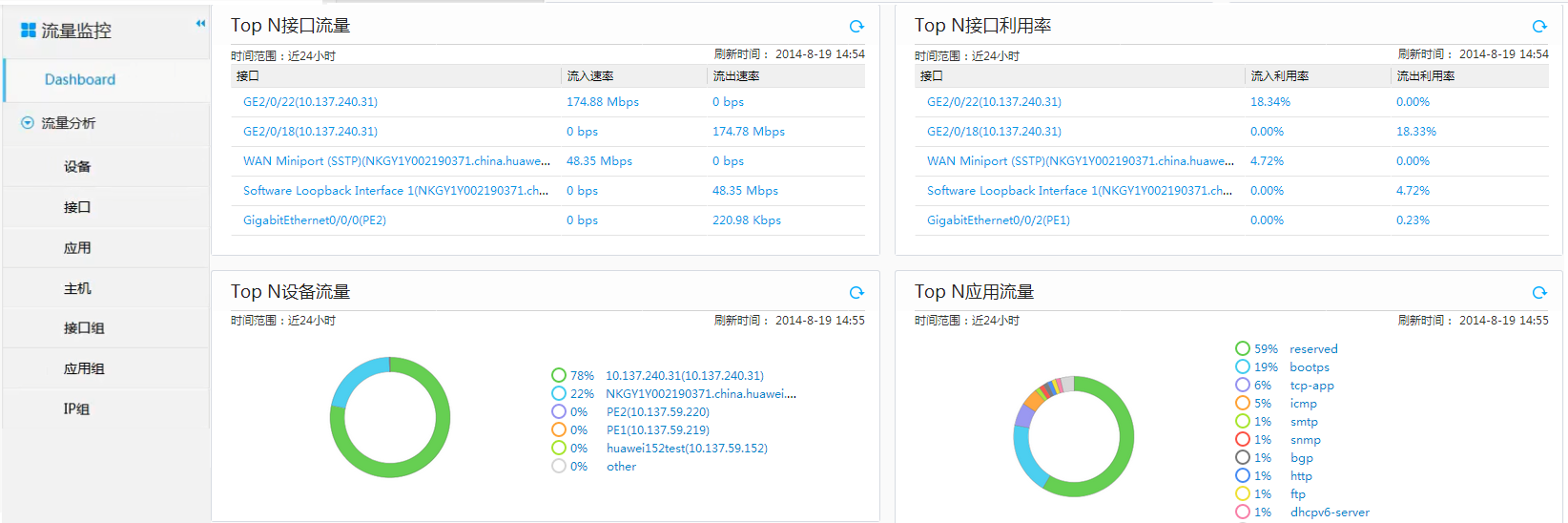


图4.7 流量概览界面

对于除接口维度意外的其他维度，点击对应维度菜单进入TOP N流量趋势界面，以应用维度的TOP N流量趋势界面进行展示的效果如图4.8所示。

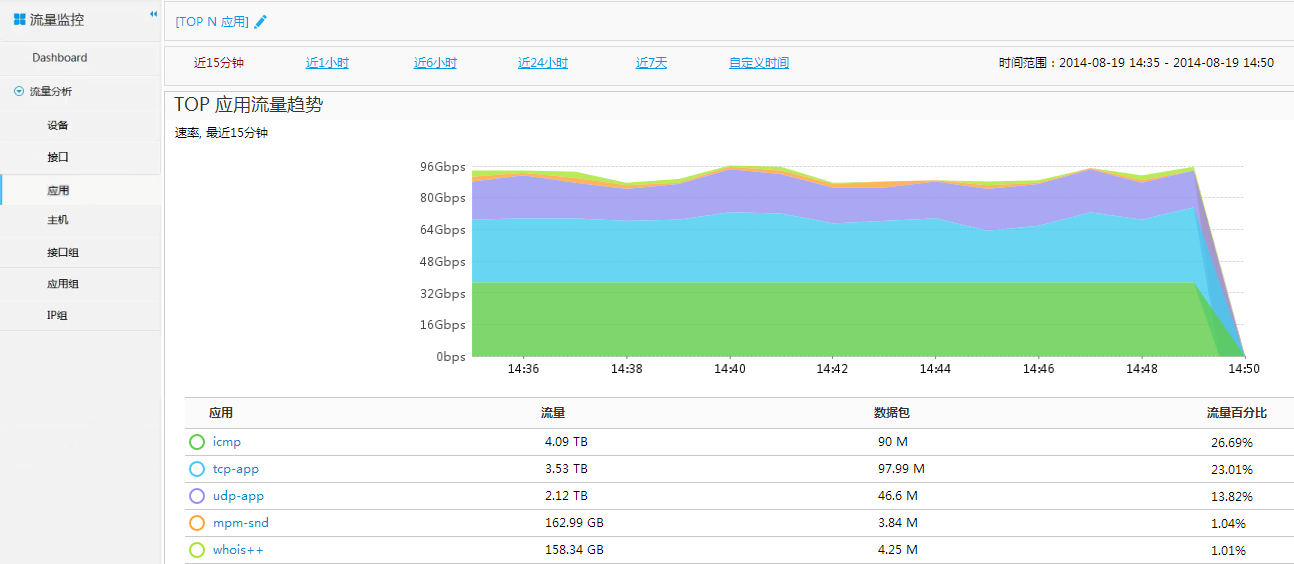


图4.8 应用维度TOP N流量趋势界面

而点击接口维度后，进入的是接口分析列表界面，而不是TOP N流量趋势界面。此界面提供全部接口的接口名称、别名流入或者流出利用率以及该接口对应的设备等信息，如图4.9所示。此界面提供对全部接口的查询，可以根据接口名称、接口别名、设备名称等条件来搜索来找到想要查看的接口，并点击接口链接则进入单个接口的具体流量分析界面，如图4.10所示。

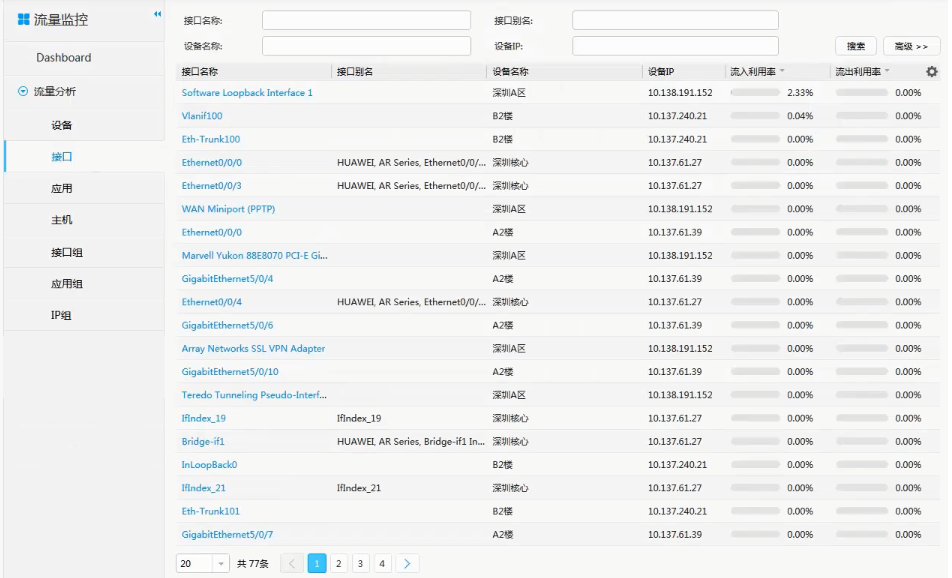
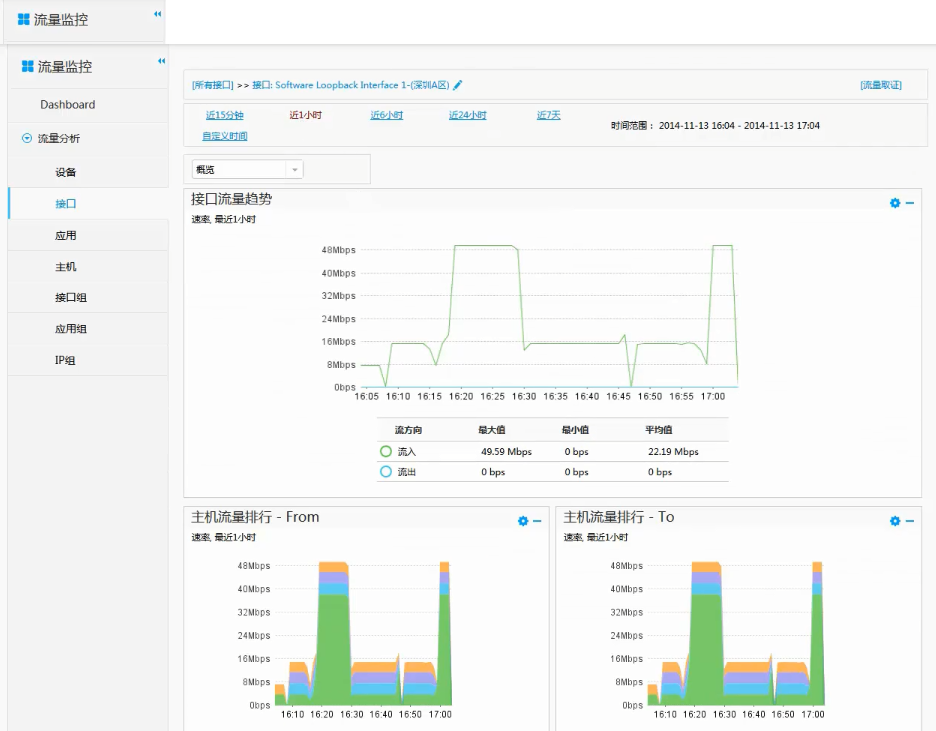


图4.9 接口分析列表界面



4.10 单个接口流量分析界面

## 4.3 流量监测配置功能的实现

控制台NMC通过图形化界面的方式给用户提供对设备及其监控信息的管理操作。网络流量监测配置功能的前台表示层使用AngularJS框架进行开发，AngularJS是一款非常优秀的前端JavaScript框架，其支持单页面的动态开发，使开发过程更为简单以及程序测试的成本，提供更为灵活的开发方式。公司在AngularJS基础上开发了一套内部eview组件，该套组件在AngularJS原生控件的基础上扩展了组件的更多功能，并优化了组件的美观度。为了满足公司要求，保证系统前台界面风格一致以提升美观度及易用性，表示层统一使用eview控件进行开发，以下是一般界面都会使用的按钮组确定和取消：

<div ng-controller="ButtonGroupController" id="buttongroupController">

<eview:button-group id="buttonGroup">

<eview:button id="confirm" click="confirm" text="{{i18n.confirm}}">

</eview:button>

<eview:button id="cancle" click="cancel" text="{{i18n.cancel}}">

</eview:button>

</eview:button-group>

</div>

这个按钮组拥有一个ID唯一的ng-controller，按钮的所有事件均实现在此controller内。使用AngularJS的另一好处是数据双向绑定，{{i18n.confirm}}和JS文件中引用的按钮国际化资源进行双向绑定，如要修改按钮文字，则只需修改国际化资源里的内容，不需要修改前台html代码即可，简化了表示层的开发。

前台请求框架使用RESTful架构，此架构使代码逻辑更为简洁，代码层次感更强，RESTful样式的服务都有一个独立的url，通过此url调用后台对应的服务。包括GET、POST、PUT、DELETE四种请求。

网络流量配置的三种配置里面的菜单代码架构基本类似，故仅挑选一个维度的配置架构进行介绍，以下为应用维度的各个url配置：

var urls=

{

appQuery:"/rest/enterprise/app/nas/manager/appcfg/quaryApp",//查询应用

appCreate:"/rest/enterprise/app/nas/manager/appcfg/createApp",//创建应用

appModify:"/rest/enterprise/app/nas/manager/appcfg/modifyApp",//修改应用

appDelete:"/rest/enterprise/app/nas/manager/appcfg/deleteApp"//删除应用

}

应用查询的RESTful的GET请求的基本格式为：

var urlVal=constant.urls.appQuery;

iemp.external.get({

url:urlVal,

dataType:"json",

data:{param:JSON.stringify(condition)},//查询条件

success:function(data)

{

initPageData(data);

},

error:function(jqXHR)

{

}

})

后台请求控制部分通过OSGI服务对前台请求进行监听：

<osgi.activator>

enterprise.app.nas.manager.as.communication.BaseinfoBundleActivator

</osgi.activator>

注册好RESTful服务将应用的查询请求映射到以下方法：

@Path({"enterprise/app/nas/manager/appcfg"})

@Target("appcfg")

@ROAEncode(true)

public class AppROAService

{ @GET

@Path({"quaryApp"})

public ROAResult<AppQueryBackDataModel> doGet(@QueryParam("param")

String param)

{ //查询条件

AppQueryDataModel condition = new AppQueryDataModel();

condition = JSONUtil.strToObj(param, AppQueryDataModel.class);

//进入实现层进行应用查询

AppQueryBackDataModel app = queryAppCommon(condition, true);

//返回结果

ROAResult<AppQueryBackDataModel> result = new ROAResult();

result.setData(app);

return result; }

}

业务DAO层实现通过Spring，采用最基本的Javabean调用请求对应的具体业务实现。以应用查询请求举例，根据请求的condition生成filterMap进行数据库查询操作，该方法对应的实现如下：

public List<ApplicationModel> queryApplication(ApplicationFilter filter,

int pageBegin, int pageLength)

{ ApplicationModel model = ApplicationFilter.convert2AppModel(filter);

String nameSqlID = "queryall.nasmgr.tbl\_nas\_application";

//转换查询条件

Map<String, Object> filterMap = new HashMap();

AppBuildParam.addFilterMap(filterMap, model);

DataSet dataSet = null;

//数据库查询

dataSet = DataBaseCommonEnt.excuteQuery(nameSqlID,

filterMap, pageBegin, pageLength);

List<ApplicationModel> models = new ArrayList();

ApplicationModel tmpModel = null;

while (dataSet.next())

{ tmpModel = (ApplicationModel)DataObjectMapper.dataMapObject(

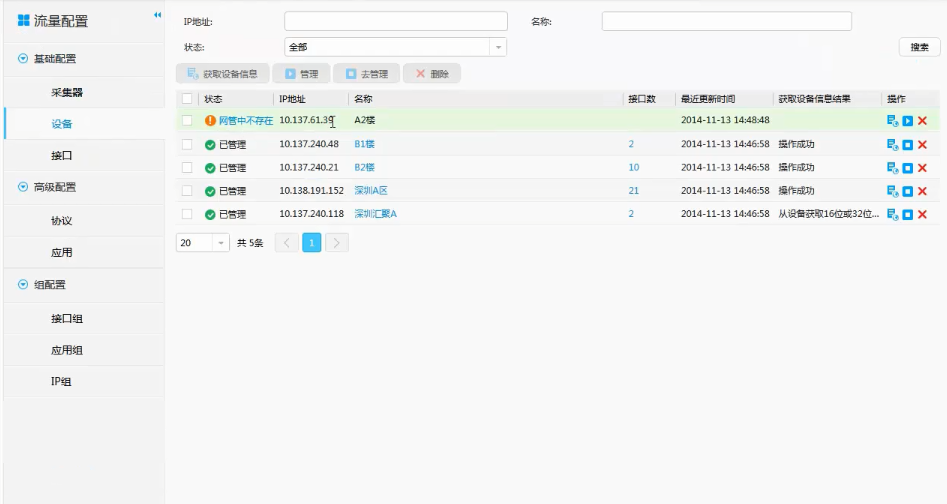
dataSet, ApplicationModel.class);

models.add(tmpModel); }

return models;

}

网络流量监测的第一步就是在系统中添加设备，并且有网络流量流经该设备，采集器采集到该设备的流量后会发送消息给NMC控制台，将该条设备存入数据库，但是该条设备并不会自动管理，需要在设备配置界面点击该条设备将其加入到系统中进行管理，此时才会发送消息通知采集器进行该条设备的采集操作。其中设备的配置界面如图4.11所示。



4.11设备配置界面

当设备添加到网络流量监测系统里后，将自动将该设备的接口入库，并显示在接口配置界面，此时设备会按照默认的1000比一的采样比进行报文的统计输出，用户可以通过此界面对接口的采样比进行修改，也可以对接口进行监控或者去监控、删除等操作，其中接口配置的界面如图4.12所示，修改接口采样比的界面如图4.13所示。

对于组配置，接口组、应用组、IP组的界面框架类似并且流程相同，三种组配置均提供对应组的搜索、创建、修改、删除等操作，以下使用IP组的配置进行展示，其中IP组配置主界面如图4.14所示。

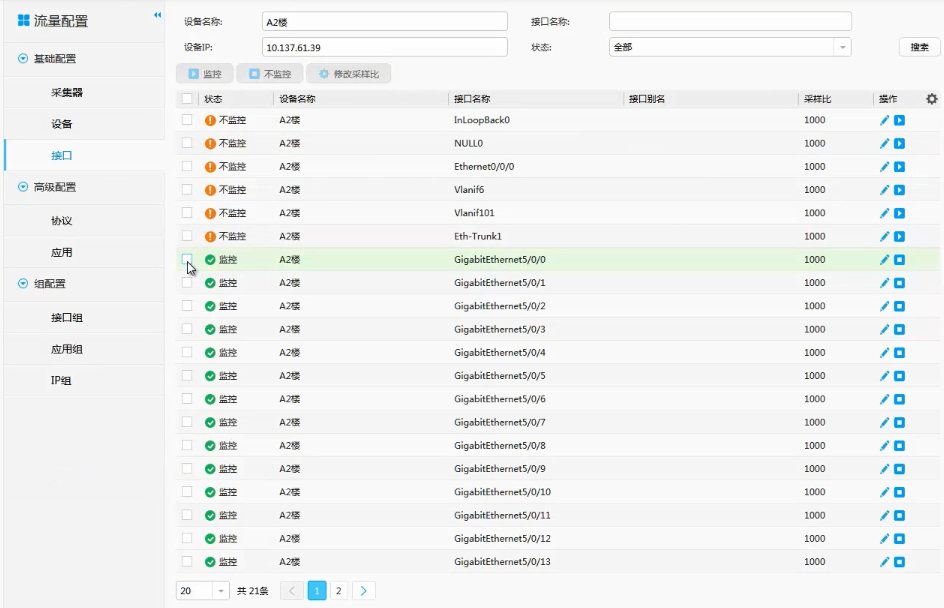


图4.12 接口配置界面

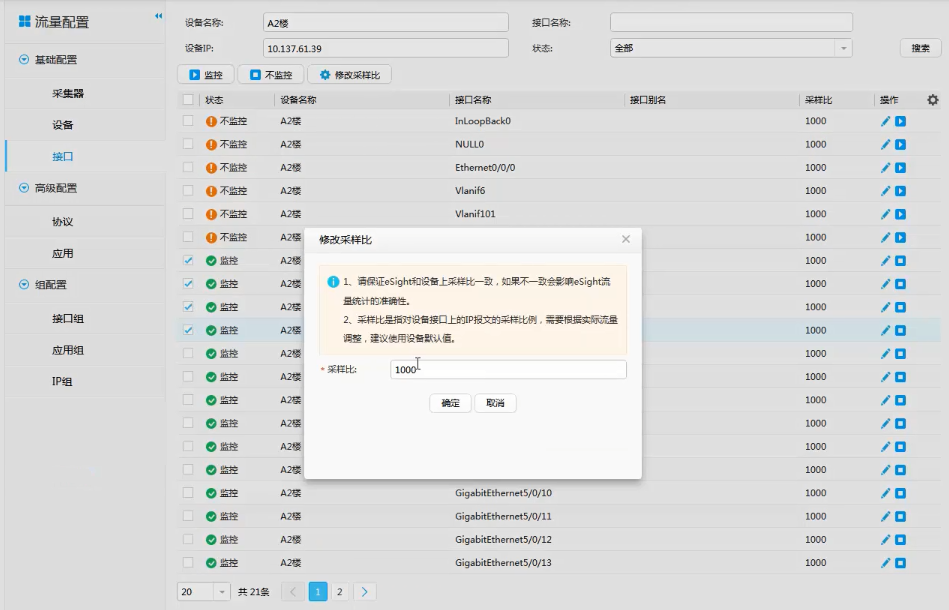


图4.13 接口配置修改采样比界面

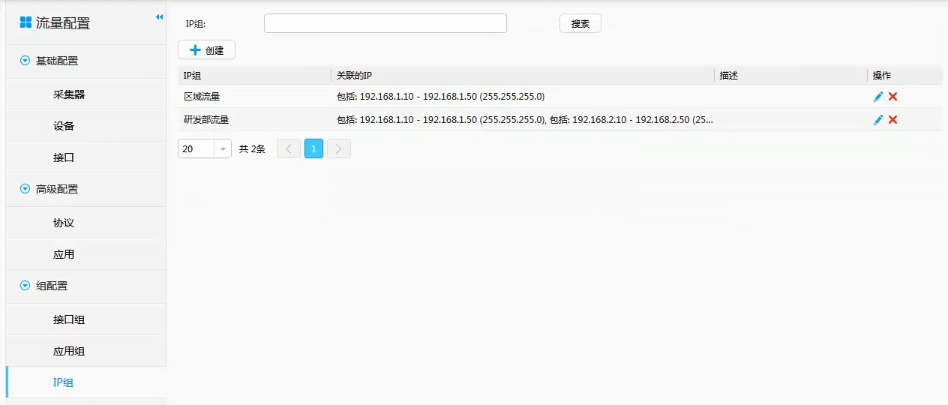


图4.14 IP组配置界面

用户在配置界面可以将某一区域或者部门内的IP地址定义为一个IP组，创建完成后，网络流量分析模块将会开始对该IP组进行流量数据的分析，其分析效果如图4.15所示。



图4.15 IP组流量分析界面

## 4.4 系统测试

### 4.4.1 测试目的

网络流量监测系统的功能测试要求保证用户按照正确要求搭建好网络流量监测系统之后，配置模块能够正常的对设备及其接口、以及其他功能进行配置，分析模块能够在要求的时间内做出响应并按照实际的用户配置显示正确的监测结果。同时在满足基本功能正常的情况下，对比较极端的情况下，比如服务器CPU持续冲高、内存不足的情况下，要求系统能够进行正常工作，并且保证流量不失真。

### 4.4.2 测试环境

不同的硬件平台对管理能力有不可避免地有所影响。测试选用比全量组件安装的最低配置要求稍高的PC服务器。其中分为两种场景：

**（1）同机部署**

同机即采集器NTC与控制台NMC安装在同一台服务器，其配置要求为CPU 2\*四核2GHz以上，内存8GB，硬盘空间120GB，数据库为MySQL 5.6，操作系统为windows 7，选用PC Sever。

**（2）分机部署**

分机部署为采集器独立安装在与控制台相独立的服务器，不需要安装数据库，其配置要求为CPU 1\*四核2GHz以上，内存4GB，硬盘空间120GB，操作系统为windows 7，选用PC Server。

测试工具选用发流工具，此工具为公司内部开发，可以任意配置发流的IP地址以及端口。高级选项可以配置会话信息，包括会话源IP地址与目的IP地址、流入接口ID、流出接口ID，也可以发送自定义应用类型的流。

### 4.4.3 测试过程及结果

系统功能测试过程如下：

* 解压并安装网络流量监测系统的全量安装包，并填写数据库的基本信息。
* 安装完毕后打开并登陆系统，此时打开测试工具，填写此NMC控制台的大网IP地址，默认的监听端口是9995与9996。
* 在高级配置里面创建会话信息点击发送，等待三分钟采集器采集到数据后，进入网络流量配置基础配置界面，发现新增了一台新发现的但是NMC控制台中并不存在的设备，此状态的设备数据并不会进行汇聚操作。
* 点击该设备，将此设备添加到NMC控制台中，此时采集器开始进行数据聚合。
* 等待三分钟查看网络流量分析界面，查看各维度是否有数据，显示的流量是否与配置工具配置相同。

具体的测试过程和测试结果如表4.1所示，该表的测试结果显示系统的基本功能正常可用。

表4.1 测试过程与测试结果

|  |  |  |  |
| --- | --- | --- | --- |
| **功能模块** | **测试内容** | **前置条件** | **测试结果** |
| 网络流量配置 | 添加设备 | 系统正确安装，对采集器发流 | 成功添加设备 |
| 删除监控设备 | 设备已添加到NMC控制台系统 | 成功删除监控设备 |
| 采集器添加 | 采集器绑定该NMC控制台IP | 成功添加采集器 |
| 协议、应用监控 | 该条协议、应用没有被监控 | 成功监控该条协议、应用 |
| 协议、应用去监控 | 该条协议、应用已经被监控 | 成功对该条协议、应用去监控 |
| 创建IP组 | 系统正确安装 | 成功创建IP组 |
| 修改、删除IP组 | 已经创建了IP组 | 成功修改、删除该IP组 |
| 创建接口组 | 系统正确安装 | 成功创建接口组 |
| 修改、删除接口组 | 已经创建了接口组 | 成功修改、删除该接口组 |
| 创建应用组 | 系统正确安装 | 成功创建应用组 |
| 修改、删除应用组 | 已经创建了应用组 | 成功修改、删除该应用组 |
| 采集器删除、修改 | 采集器已经成功添加 | 成功删除、修改采集器 |
| 网络流量分析 | 查询DashBoard流量概览界面 | 系统正确安装，汇聚数据表中已存在数据 | 显示正确 |
| 查询接口维度分析界面 | 系统正确安装，接口汇聚数据表中已存在数据 | 显示正确 |
| 查询应用维度分析界面 | 系统正确安装，应用汇聚数据表中已存在数据 | 显示正确 |
| 查询IP组维度分析界面 | 系统正确安装，配置了IP组且有流量流经该IP组 | 显示正确 |
| 查询主机维度分析界面 | 系统正确安装，主机汇聚数据表中已存在数据 | 显示正确 |
| 查询接口组分析界面 | 系统正确安装，配置了接口组且有流量流经该接口组 | 显示正确 |
| 查询应用组分析界面 | 系统正确安装，配置了应用组且有流量流经该应用组 | 显示正确 |
| 查询设备维度分析界面 | 系统正确安装，汇聚数据表中已存在数据 | 显示正确 |

# 第五章 工作总结与展望

## 5.1 工作总结

本论文基于NetStream技术实现了可以实时监控电商数据仓库IDC之间网络流量的系统，该系统能够对环境中的网络流量进行统计并且通过直观的图形化界面详细、准确的显示出网络流量的详细组成，帮助用户实现网络流量可视、故障可查、规划可依的网络透明化管理目标，并且网络管理员也可以及时了解网络流量动态，确保网络带宽得到充分合理的使用，保障网络长期稳定高效运行，为优化网络设计和规划，实现以最小的网络运营成本达到最佳的网络性能和可靠性提供了坚实基础。

本文首先说明了选择这个课题的背景以及实现这样一套系统的现实意义，结合国内外网络流量监测的发展现状对本论文所要实现的内容进行了额阐释。而后对NetStream技术做了基本的介绍，以及NetStream系统的各个组成以及作用进行了详细说明，更进一步的解释了系统整体的运行机制，包括系统的流量统计、采样方式以及几种流老化的方式和处理结果的输出格式。紧接着对网络流量监测系统的总体架构以及各个功能点进行了详细设计。并对网络流量监测系统的功能进行实现，并贴出实现过程中涉及到的代码的核心部分。最后系统进行功能性测试并得出相应结果。

## 5.2 工作展望

本系统实现了对网络流量的监测，可以显示出流量的详细信息。但是目前网络发展如此迅速，流量将会越来越庞大，对带宽的要求也会越来越高，从系统测试环节可以看出，系统的正常运行对服务器配置要求比较高，况且系统可以监控的数据只能达到几百台，对于现在一些比较大型的场景的几千甚至几万台设备无法胜任。目前对该系统存在一些可以改进的方面：

（1）系统所有模块的操作产生的结果最终保存到数据库里面，并且所有的显示也是要经过数据库的多次查询，某些界面存在众多控件，加载一个页面存在太多次的数据库查询或者修改、删除等操作。这将导致系统的性能下降，同时也会浪费不必要的资源，界面也会出现卡顿等状况。所以该系统可以在数据库查询方面进一步优化，可以设计更为高效的表结构，简化查询流程，防止冗余查询，也可以设计更高效的查询方法提高查询的速率和性能。

（2）对于日后更加智能的时代，系统也应该趋于智能化，目前系统仅仅提供数据的显示，以及一些基本组成、速率等的分析，对网络状况的总体状况比如哪些是异常流量，哪些应用的流量偏高并没有分析功能，所以系统可以对分析功能进行加强，提供一些比较好推测的分析结果。

# 致 谢

值此论文完成之际，谨向所有关心和支持我的人们致以诚挚的谢意！

首先，我要衷心地感谢我的校内导师刘奇其教授和企业导师薄宏剑工程师。从论文选题、内容和整体结构的确定，到直至最后定稿，两位导师都以极其负责的态度给予悉心指导，为我提出了许多宝贵的意见和建议，使我获益良多。导师有严肃的科学态度，严谨的治学精神和精益求精的工作作风，这些都是我所需要学习的，感谢导师给予了我这样一个学习机会，谢谢!在此，谨向导师致以最诚挚的感谢！此外，还要感谢在企业实习时团队的成员。没有他们的帮助和共同努力，就没有项目的圆满成功，也就不会有本文的形成。在此，向他们表示衷心的感谢!

# 参考文献

1. 陈春霞. 基于容器的微服务架构的浅析[J]. 信息系统工程, 2016 (3): 95-96.
2. Fielding R T. Architectural styles and the design of network-based software architectures[D]. University of California, Irvine, 2000.
3. 唐文宇. 面向 SOA 架构微服务的安全系统的设计与实现[D]. 南京大学, 2016.
4. Abrams S, Kunze J, Loy D. An emergent micro-services approach to digital curation infrastructure[J]. International Journal of Digital Curation, 2010, 5(1): 172-186.
5. Namiot D, Sneps-Sneppe M. On micro-services architecture[J]. International Journal of Open Information Technologies, 2014, 2(9).
6. 徐鹏, 陈思, 苏森. 互联网应用 PaaS 平台体系结构[J]. 北京邮电大学学报, 2012, 35(1): 120-124.
7. Newman S. Building microservices[M]. " O'Reilly Media, Inc.", 2015.
8. 王克苑, 张维勇, 王建新. SSL 安全性分析研究[J]. 合肥工业大学学报: 自然科学版, 2004, 27(1): 87-91.
9. 王珊, 王会举, 覃雄派, 等. 架构大数据: 挑战, 现状与展望[J]. 计算机学报, 2011, 34(10): 1741-1752.
10. Brooks F P, Li Q. 人月神话[M]. 中国电力出版社, 2003.