南 开 大 学

本 科 生 毕 业 论 文（设 计）

中文题目： **基于容器编排架构 Kubernetes 的监控系统设计**

外文题目： Monitoring System Design based on Container Orchestration Architecture Kubernetes

学　　号： 1711318

姓　　名： 达益鑫

年　　级： 2017 级

学　　院： 网络空间安全学院

系　　别： 物联网工程

专　　业： 物联网工程

完成日期： 2021 年 5 月

指导教师： 徐敬东 教授

关于南开大学本科生毕业论文（设计）的声明

本人郑重声明：所呈交的学位论文，是本人在指导教师指导下，进行研究工作所取得的成果。除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人创作的、已公开发表或没有公开发表的作品内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。本学位论文原创性声明的法律责任由本人承担。

学位论文作者签名：

年　　月　　日

本人声明：该学位论文是本人指导学生完成的研究成果，已经审阅过论文的全部内容，并能够保证题目、关键词、摘要部分中英文内容的一致性和准确性。

学位论文指导教师签名：

年　　月　　日

摘　要

近年来，容器技术在物联网、云服务、边缘计算等领域广泛应用，掀开了新时代微服务分布式部署的大幕。日渐规模增大的容器网络催生了容器编排技术，Google 公司牵头开发的 Kubernetes 系统成为了事实上的容器编排标准，能够实现对大规模容器集群的管控和调节，这样规模的部署当然离不开监控系统的帮助，尤其在信息就是价值的大数据时代，如何获取容器集群的状态以及相关的信息变得异常重要，自然对于集群状态的监控就成了研究 Kubernetes 及容器网络的重点之一。

本文在原生的 Kubernetes 集群上设计了分布式部署的监控系统，背靠其丰富的技术簇实现了对 Kubernetes 集群对象做实时监控。系统拥有灵活的接口设计，使用多类型的监控工具，可以满足大多数监控场景的需求；节点为单位的计算模块设计，既是调节监控数据读取策略的核心，也是节点监控数据运算方法的集成地带，能够依据需求拓展为功能性计算组件；网络传输采用推送策略，保持稳定链接的同时铺垫主控节点的调度请求；依靠 InfluxDB 数据库完成数据收集统一归档；对接 Gin 框架启动后台管理，采用动态而形象的前端设计工具 Echarts 设计显示界面。与此同时，系统构架设计完全遵循 Kubernetes 以 API 资源为对象开发原则和容器应用解耦式设计原则，使得各个模块之间联系紧密，但依据功能实现的不同可单独进行技术改善。最后本文对设计的系统架构进行了实际模拟测试，采用最典型的两层三点拓扑，组成基础的容器网络，通过脚本命令行的方式完成功能部署，在此基础上进行了严格的负载测试和鲁棒性测试，确认系统可稳定运行并达成工作期望。

关键词： Kubernetes 编排；容器；监控系统；InfluxDB 时序数据库

Abstract

In recent years, with the prosperity of the information age, Container Technology has been widely used in the IoT, cloud services, edge computing and other fields, opening the curtain of the distributed deployment of microservices in the new era. The increasingly large-scale Container Network makes Container Orchestration Technology thrive upon infrastructure of network. At this point, The Kubernetes system, developed by Google, has crowned himself as the de facto Container Orchestration Standard, which makes the control and adjustment of large-scale container clusters an easy task. Of course, such a scale of deployment is inseparable from monitoring, especially in the era of big data where information stands for value. In that case, how to obtain the status of the container cluster and related information becomes extremely important. All comes to one point that monitoring the status of the cluster has become an issue worthy to study on Kubernetes and container networks.

Paper has designed a distributed monitoring system based on the native Kubernetes cluster. The system has a flexible interface design and uses multiple types of monitoring tools to meet the needs of most monitoring scenarios. The node-based calculation module design is not only the core of adjusting the monitoring data reading strategy, but also the integration zone of the node monitoring data calculation method , Which can be expanded into functional computing components according to the demand of Pod . Network transmission adopts a push strategy to maintain stable links while paving the way for scheduling requests of master nodes. InfluxDB takes the duty of data collection and unified archiving. docking Gin framework deals with background management, adopting dynamic and image The front-end design tool Echarts , which provides the display interface. At the same time, the system architecture obey the Kubernetes development principle , which looks API resources as the control-object , and the decoupling design principle of container applications,

which makes the connection between the various modules close. In the end , this paper conducted actual simulation tests on the system architecture, using the most typical two-layer three-point topology to form a basic container network, and completing function deployment through script command lines. On that basis, strict load testing and Robustness test to confirm that the system can operate stably and meet working expectations.

Key Words: Kubernetes; container; monitoring system; InfluxDB;

目 录

第一章 绪论

第一节 Kubernetes 系统构建的背景和意义

1.1.1 边缘计算架构需求和容器系统的研究价值

现代社会急速发展，促使着计算机技术不断进步迭代，伴随着应用需求环境的优化，整个计算机行业欣欣向荣。如此条件下，物联网基础设施建设规模爆炸性增长，部署的机器计算能力突飞猛进。在 5G 网络即将推广使用的当下，单位内可连接使用的移动设备数量将会数次激增。

与用户基数呈倍数级关系的数据量推动了包括智能技术在内的多项科技发展，却也在以更高倍率的增长趋势带来基础设施及相关系统架构的负担。以大数据为核心的新一代技术簇在边缘云和 AI 部署应用，其核心机制是从数据当中获取可行的分析结果来模块化、自动化计算范式并提供服务。然而，可预见的是：在这样的环境变化之中，系统连接的设备集群将拥有庞大的规模，生产的数据将具有高度的复杂性，很可能已经超过了以往网络基础架构的能力。过去集中式的数据中心和云部署形式将逐渐无法满足要求。与此相对的是，边缘计算提出了更有效的替代方案，核心思想是在接近数据创建的位置对数据进行处理和分析，达到减少延迟和网络消耗的作用，通过计算的分布式部署带来数据的最大化利用。由此边缘计算成为近期网络及云研究的最主要模块之一[1]。

以 Docker 为代表的容器技术是目前边缘计算架构协调部署高性能应用的优良方案[2]。边缘计算架构当中端点数量的成倍增长，使用数据的服务也不断增加，过去以实体机和虚拟机为主体的开发运维环境早已不能够适应边缘计算的发展容器技术能够提供更具可扩展性的方式协调和部署软件，现代计算机云技术将会是以容器为核心的基础设施建设。

庞大的系统构架涵盖网络、通信、存储、操作系统、分布式原理等各方面技术；将这些技术微型化、组件化、分布化，是容器技术提出的主要理念。容器技

术可理解成文件系统划分出独立运行的沙盒，系统在其中运行代码；最大的优势是能够运作非常轻便“机器”。通常这样的概念“机器”只有几个 MB 大小，并且可以在几毫秒之内启动。不同于以往虚拟机环境需要移植的文件存储和硬件配置，容器将应用程序的代码、依赖项、系统工具和库包装在一个镜像当中，能够在任何满足操作系统要求的物理机上实现工程应用的移植。同时容器也提供了硬件抽象层，允许它们在 IoT 端点中常见的各种系统上部署，不需要复杂的运维过程就能够实现本地和云端环境的高度一致，给应用迁移提供了相当优越的条件。

以 Netflix 为例，为能够控制调配不同遥远地区的资源分配，该公司大约每周都会发布一百多万个包装着具体功能应用的容器，为成千上万的用户提供着高清视屏体验，可见容器技术将会是对计算机领域的一次重大改革。

目前，在这场“容器”为核心的科技技术变革当中，Google 公司开源的 Kubernetes 项目已然成为容器技术的事实标准，重新定义了基础设施领域对应用编排与管理的各个开发方向，为容器化应用提供了集群化部署运行，自动资源调度和动态水平伸缩等一系列完整的功能。

对于数量规模庞大的容器集群，单独分别的控制和访问是困难的，与此相对多个容器的沟通和部署却几乎都是完全相同的操作，这与历史上物理机操作系统诞生的基础条件非常相似。类比于操作系统的诞生，容器技术的发展也催发了容器编排（Container Orchestrainon）系统的产生和发展，用户通过简单的命令就能够部署、管理、维护和监控应用化的容器；可实现的功能包括集中化地管理系统、分布式地运行架构、大规模地调度、小范围地设置、多层次地构建，使得诸多容器构架的部署成为可能。以 Kubernetes 为代表的容器编排监控系统势必是容器领域的研究重点。

1.1.2 监控系统在容器编排当中的重要性

监控是容器编排系统的重点，更是系统编排调度的核心数据来源。从数据管理监控方面来说，系统的正常运行、每个容器的功能完善、生命周期可控、分配任务、调度消耗优化等都需要实时获取到容器的状态。从系统构架方面来说，监控的对象也不只是集群当中的容器，还包括具体的应用、设置的服务还有集群部

署的实体机器等等，他们的监控数据都是调整集群状态的重要依据，更是实现容器特征功能、做出整体调度的重要观测对象。从应用构架方面来说，能够通过监控数据的反馈调整和计算当前应用的状态，调度分配资源，能在很大程度上提高容器应用部署的性能，降低系统资源的消耗。

从数据分析的角度来看，诸多智能技术关注的重点是容器反馈的数据，从监控数据当中归纳出普适性的模型，以此开发出特殊功能的应用正成为智能技术落地的一大趋势；监控系统的拓展性为这类应用与容器技术的进一步结合提供了催化剂。容器的监控与功能服务相结合能提升架构系统的使用价值，其重要性对现有容器架构来说不言而喻；因而，监控系统的研究在的边缘计算当中是非常有研究价值和研究意义的。

本文设计并搭建了可在以Kubernetes为代表的容器编排系统上部署的监控系统，系统所有组件设计贯彻容器部署的理念，每一个部分都能作为功能性的容器运行在集群架构当中，将节点的监控数据推送到主节点上并展示出来。

第二节 国内外容器系统及监控场景相关技术的研究

容器技术的繁荣根植于PaaS概念的普及和发展的当下，相关的技术发展和历程与整个社区的发展息息相关。容器技术最初以Cloud Foundry为代表的PaaS项目正式步入基础设施建设的领域开始。在逐步平台化发展趋势当中，部分新创公司开始发展新型平台原生技术，其中DotCloud公司以Docker技术拉开了容器边缘架构的时代大幕。

Docker有着对Linux技术的优良封装，用户只需通过简单的流程就能迁移应用和部署程序，这让当时苦于平台迁移环境维护情况糟糕的开源社区成员如获至宝，加之技术初期不够完善的技术框架，让大批技术人员涌入，使得容器部署及其使用的理念迅速传播和扩张。

商业化的需求进一步促使着容器技术更迭，促使它能够更大程度上完善其对于平台层提供服务的能力，这项资本商业驱动的技术，其动力核心是商业。容器真正推广起来的根本原因是它对PaaS理念的革新，将功能和机器、需求和应用

结合，脱离开原本盘根错节的系统部署，让大规模的服务部署和管理成为可能，优化了责任分层的同时更减轻了大量维护环境的成本。所以当大量技术人员的项目落地时，容器的概念扩展到了更为广阔的区域：工业生产、数据库、基础设施应用等等，最终使得容器技术所代表的微服务和部署理念深远地影响着现代网络架构的发展。

Docker 应用集群规模的扩展暴露了其对于多节点集群管理能力不足的问题，最终催生了容器编排技术的产生。最初推动 Docker 集群建设的是 CoreOS 公司，其原本的优势产品是分布式集群的操作系统，通过与 Docker 公司合作来结合 Docker 技术，该公司实现了能够让用户按照分布式集群的方式——实际上就是一个庞大的管理操作系统，能够控制所有安装了该操作系统的节点，这算是最初的 Docker 管理监控的系统参考。随后由于利益纷争，CoreOS 与 Docker 公司断开合作，开发了自己的一套容器技术 Rocket(rkt 容器)，依托于包括 Container Linux 操作系统、Fleet 作业调度工具、Ystemd 进程管理等开源项目，以此对外提供集群的管理和监控功能，这也促使了 Docker 公司发布了自己的集群管理项目 Swarm。

商业竞争带来了 Docker 容器社区的繁荣，为补足容器管理上的缺陷 Docker 公司收购了 Fig 开源项目，这个项目正式提出了容器编排（Container Orchestration）的概念。简单来讲就是对 Docker 容器的一系列定义、配置和创建、监控动作的管理；通过少量的指令和动作完成对大量容器的管理和维护，促使了容器分布式架构的进一步发展，编排这一理念也成为容器的分布式系统搭建理论基础。同时期 Docker 公司的竞争对手还有 Mesos，他拥有着独特的两层调度机制和超大规模集群管理的经验，并结合容器理念构建了 Marathon 项目，提出 DC/OS（数据中心操作系统）的口号，旨在像管理一台机器那样管理一个万级别的实体机集群，并且使用 Docker 容器在这个集群里自由地部署应用，这为后来的容器编排数据管理做了奠基。在那段时间里，以容器为核心，尤其是拥抱 Docker 技术的商业公司和项目层出不穷，但也造出了 Docker 公司一家独大的局面，带来了资本的切割和垄断。在 2015 年，迫于压力，由 Docker 公司牵头和 CoreOS、Google、RedHat 等公司共同宣布，Docker 公司将 Libcontainer 捐出并改名为 RunC，交给

中立的基金会管理，同时提出了一套标准和规范——OCI（Open Container Initiative），这意味着将容器运行时和镜像的实现从 Docker 项目中完全剥离出来——让容器技术与 Docker 解绑，成为单独的开发概念。随后 Google 和 RedHat 等开源基础设施领域主体牵头发起 CNCF（Cloud Native Computing Foundation）基金会，建立以 Google 自家的 Kubernetes 项目为基础，由开源基础设施领域社区主导的容器编排规则。

Kubernetes 项目源自 Google 自家的 Borg 项目，Google 公司将自己在基础设施领域多年来的建设经验同容器化理念的结合，提出包括 Pod，Sidercar 等功能理念并将其融入到 Kubernetes 项目当中，许多超前的设计和宽泛性质的理念迅速吸引了大批信徒，最终致使 Docker 公司彻底放弃对容器技术的限制权力，转而成为商业服务公司，由此开启容器编排同容器部署百家争鸣的局面。这是目前容器管理以及其相关技术的发展情况[2]。

第三节 本文主要的工作和内容

本文基于 Kubernetes 架构设计了以容器为基础的边缘架构监控系统，包括边缘节点监控数据获取插件及网络传输连接方式，控制节点收集存储数据的方式和界面展示逻辑。

文章将会先介绍 Kubernetes 相关技术和组件设计构想、网络连接的协议设计和监控的技术原理，在这些技术的基础上提出可分布式部署的监控系统架构。系统将会以容器的形式部署在控制节点和工作节点上，最终能够实现对多个节点的状态监控和可视化观察；系统内部的推送机制和计算的分布式部署也带了监控功能的分布式特性，尤其是在网络节点繁杂众多，数据量负载较高的时候也能够正常运行，提供服务；丰富的可视化界面设计让系统使用有更佳的体验；各组件的功能性拥抱容器理念，有着长远的开发前景。

第四节 本文的主要结构

本文结构主要分作四章，其主要内容和组织结构做如下说明：

第一章介绍了本文工作的技术背景，表明了本文工作的构建动机和意义价值，分析了容器系统的应用需求和构建监控系统的必要性，说明了本文相关的容器以及其编排监控的国内外研究背景和现状，为全文的工作和研究做了意义阐述和背景铺垫。

第二章是对本文使用的技术做具体介绍。系统基础构架基于 Kubernetes 容器编排框架，本章说明了所有接口和应用部署的总体理论和方式，是所有工作和成果的基础论述。监控系统的原理阐述部分主要说明监控数据的获取渠道和方式，集中描述了工作使用和开发的监控手段以及其所能够获得的监控数据对象，同时也划定了监控的范围。InfluxDB 数据库说明部分则完成对整个系统的数据收集整理，配合 Gin 框架开发可视化的监控应用界面的阐述。

第三章具体论述了本文的工作内容，将会详实说明系统模块的实现和细节。总体架构的设计；以命令行.为主要手段的实体机搭建 Kubernetes，完成模拟环境必要依赖的安装；监控模块的设计以及兼容多监控方式的接口布置，计算分布式布局规划，网络传输推送机制的实现；系统控制展示模块的部署，包括网络接口的信息收集和数据库存储访问，前台界面的设计和渲染输出，和其中同样是推送机制设计的刷新模式。

第四章 讨论系统的功能负载测试和结果，进一步讨论系统开发和完善的前景规划预期，对本文工作做进一步归纳和对未来工作的拓展做了阐述。

第二章 Kubernetes 监控系统应用技术介绍

第一节 Kubernetes 容器编排系统

Kubernetes 源于 Google 内部的 Borg 项目。该项目在用 Go 语言重写之后于 2014 年 6 月正式开源。Kubernetes 希腊语为“舵手”，Google 致力于用其管理数以万计的容器集群，这个容器编排架构设计要求多端环境及其配置同应用相伴，保证开发测试生产的环境尽量一致，其主要的应用目标是消除运维工作中物理机、虚拟计算，网络和存储基础设施的负担，并使应用程序运营商和开发人员完全将重点放在以容器为核心的应用上。

Kubernetes 能够提供多层次的安全防护和准入机制、多租户应用支撑的能力、丰富的服务发现和提供机制、内嵌有负载均衡器保障资源稳定机制、故障发现和自我修复的能力、服务滚动升级和在线扩容、可扩展的资源调度、多粒度的资源配额管理能力等等机制服务。

Kubernetes 的设计理念和功能根源于 Linux 架构[5]，其根据功能和架构实现逻辑，对从核心部分到对外提供服务的组件做了归纳分层，如图 2.1 所示：

图 2-1 Kubernetes 功能架构分层图

核心层是 Kubernetes 的核心组件，对容器集群外的主体提供 API 来构建功能

性的内容，拓展架构丰富度；对内部对象提供插件和运行支持。

应用层是各个组件具体功能的实现，包括部署有状态或者是无状态的应用、批处理任务。集群功能性应用，还包括路由功能的实现，Pod 对外来连接服务以及集群 DNS 解析等等，其主要载体是 Pod。事实上对于 Kubernetes 而言调度和分析的主体单位是 Pod，也是应用层主要承载的实体。

管理层指的是度量调度操作以及控制自由扩展内容的参数；具体来说有系统度量的任务，比如基础设施，容器和网络的度量，对单个监控主体来说，这些度量的数据来源离不开容器创建的理念——资源隔离，实质还是基于已有的系统监控内容来做收集；自动化扩展的任务，动态 Provision 设置，由于 Kubernetes 是以资源指向来做用户调度的，主体可以通过简单的设置参数和操作来控制集群当中的资源分配和扩展应用，将负载均衡的问题交给 Kubernetes 来处理。

接口层，是交互接口层面向架构使用客体，具体指的是包括 Kubectl 、客户端 SDK 以及集群联邦等接口控制的部分，主要集中的内容是客户端的各类工具和依赖支持，是商品层次的服务提供。

生态开发层，指的是大量开发人员基于 Kubernetes 开发的种类繁多的应用。大量丰富的功能 API 和工具开发方式促使着开发者们涌入 Kubernetes 社区，并为这个欣欣向荣的系统建言献策，开发出 Kubernetes 外部插件比如日志管理，系统监控，配置管理，CI, WorkFlow 等；Kubernetes 内部架构组件包括 CRI，CNICloudProvider 等等。

本文的工作既包括核心层的 API 调用也包括应用层，管理层的数据分析和管理。Kubernetes 的核心组件如图 2.2 所示，其重点在于分布式系统的设计理念以及核心组件[7]。Kubernetes 的各项功能和容器绑定，需要 Kubernetes 功能支持的部分必须要通过 API-Server 实现。如图 2.2 所示，应用层对于集群的组件需求都是通过 API-Server 来实现的，这一部分的功能主要是靠静态资源文件来实现交流，通过 Yaml 文件向 Kubernetes 发起请求获取对应的资源。

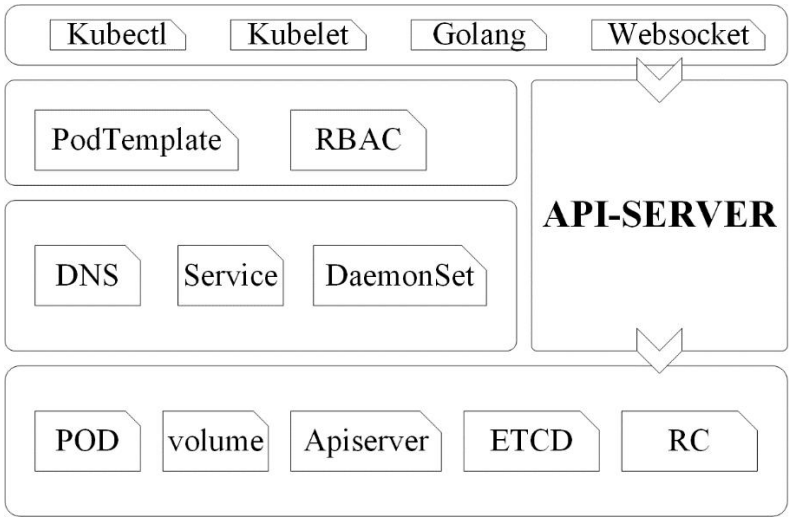

图 2-2 Kubernetes 功能组件分层图

Kubernetes 的 API 设计应当全部都是声明式的，其所有的对象都具有名词的属性，彼此之间应当满足“高内聚，松耦合”的理念，这一点也贯穿了文本系统的 API 设计流程。高层 API 是以易操作意图为基础设计，低层 API 以高层 API 的要求做设计，API 的复杂度应当与对象的数量相结合。Kubernetes 的所有部署都是通过设置 API 对象达成的，所有对应的操作都是申明式（Declaration）而不是命令式的，这样就使得服务和对象都可以保持状态稳定。本文工作主要涉及的几个 API 对象也是 Kubernetes 的核心组件[8][9]。

Pod 是最基础的 API 对象。在 Kubernetes 当中，“容器”的概念更多指的是 Pod。所有 Docker 一级的容器都在 Pod 当中运行，一个 Pod 可以承载一个或者多个相关的容器运行负载设置的服务，但对于 Kubernetes 而言，能够看到和调度的容器单位是只有 Pod；同一个 Pod 当中的容器会部署在同一个物理机器上面并能够共享资源，可以将 Pod 理解为实际上 Kuberntes 调度管理的机器。

Pod 是 Kubernetes 集群中所有组件类型的集合，其他组件可以视作是实现了特殊功能的 Pod，比如 Deployment ,Job,DaemonSet 和 PetSet。其中 DaemonSet 用于长期部署运行类和批处理型服务，要求每一个节点上都有同一个功能类型的 Pod 运行，并且是由 Kubernetes 来维持这个 Pod 长时间运行，保证其功能能够完整实现，本文的监控插件就是以 DaemonSet 的形式部署到每一个节点上的，这个插件能够收集监控节点的数据并将其返回给主节点。

存储卷（Volume）是继承自 Docker 系统的存储卷，相比较而言它的生命周期和作用范围并不是 Docker 而是 Pod，每个 Pod 当中申明的存储卷由 Pod 当中所有的容器共享。

具体来看其体现在可配置持久存储卷（Persistent Volume,PV）和持久存储卷声明（Persistent Volume Claim ,PVC）来配置申明不同编排对象的资源。PV 是节点层级的资源配置文件，根据集群的基础设施变化而变化，是 Node 资源的提供者，由 Kubernetes 来维护管理，也是集群总体状态信息的获取地；PVC 是 Pod 层级的资源配置文件，根据业务需求而变化，是 Pod 资源的提供者，由集群应用的使用者来维护。但归根结底其核心和基准就是 Linux 的资源隔离技术；本文所作监控就是在这一层面上，通过读取监控数据不同对象的资源文件卷，日志文件卷来实时获取集群的运行状态。

副本控制器（Replication Controller, RC）基于典型的副本控制机制，通过维持用户申明的资源服务副本数量，控制和维持多个容器的 API 对象；通过部署管理器（Deployment）创建，更新一个对象服务，同时如果想要解除集群上的 Pod 对象就必须要先删除对应的 Deployment，利用这一点 Kubernetes 能够保证所申请的资源和服务对象能够完整的运行在集群当中，自身负责资源调度和分配；假设申请的某个服务因为节点故障而失效或者某个节点的余留资源不足以支持服务继续运行，Kubernetes 就会重新调度一个新的服务部署在新的节点上面，对应用层面来说，申请的多个服务的状态没有任何改变，这些操作对于这一层是完全透明的。如此机制当然很好地为上层架构提供了服务体验，但同时也影响了包括监控，调度在内的各项集群内服务，因为其详细具体位置和状态是不能够直接获取的，但是其服务是稳定保持的。

服务(Service)解决了访问部分的难题，创建服务对象能够提供对象发现和负载均衡的能力，技术上是创建了 Kube-Proxy 做负载均衡器。Kube-proxy 是分布式部署在各个节点上面的，有着极强的伸缩性，能够定位所需要的服务，但是也代表着需要主节点发起请求才能传送数据资源，这对于监控机制来说存在着不可接受的负担。这一点也催发了监控机制的改进办法，本文将论述自己的思考和解决方案。

Etcd 是集群的数据库，保存了整个集群的状态。ApiServer 提供了整个集群资源操作的唯一入口，也就是所有对集群通过 Kubernetes 的访问都必须要经过 ApiServer，他也提供了认证授权，访问控制和 API 注册和发现等机制。Scheduler 负责资源的调度，按照规定的调度策略将 Pod 调度到相应的机器上面，也是维持特征 API 对象正常运行的调度器，保证了集群的正常运行。

第二节 监控系统的原理和机制

容器是整个系统设计的核心[5]，而容器监控根植于 Linux 操作系统。相较于传统的虚拟机技术而言，容器是一种在宿主机上的特殊进程，并不是完全和宿主机隔离开来的单独主体，其核心技术就是 Linux 的 Cgroup 技术和 Namespace 技术[10]。

对容器技术理念做简单分析：每当在宿主机上运行一个程序，操作系统都会给这个程序一个 PID 和它所需要的资源、执行环境等等，容器所做的就是给这个进程做出隔离，让它在自己的 Namespace 当中以为自己就是 PID 为 1 的进程，或者说这个进程能看到的计算机资源视图只在容器划归的范围之内，涉及到 Mount、UTS、IPC、Network 和 User 这些资源视图，实质上只是对这个进程做了各类障眼法，也意味着容器的各类资源消耗并不需要单独在做出硬件的模拟，容器完全受宿主机的操作系统的控制和调度，这能够使得容器具有极高的敏捷度和性能优势，但也导致了不彻底的隔离状况。显然的，所有的容器在宿主机的操作系统看来都是进程，甚至与它调控的其他程序进程没有太大差别，宿主机是能够直接看到容器里面的资源状况，同样这也为监控提供了可获取数据的途径。这里涉及到的是 Linux Cgroup（Linux Control Group）技术,其最主要的作用是在于限制一个进程组能够使用的资源上限，包括 CPU,内存，磁盘硬盘，网络带宽等等。

Linux 基于 VFS（Virtual File System）通过文件的方式，将 Cgroup 的功能和配置提供给外界对象,而在 Cgroup 子系统之下是操作系统收集和统计的信息，这些信息并不是限制信息，而是实际收集到的资源状态，比如 cpu.stat 统计值下的

nr_periods 表示进入周期的时间，nr_throttled 表示运行时间被调整的次数，cpuacct.usage_percpu 表示该 Cgroup 当中所有任务使用各个 CPU 核数的时间。这些由子系统生成的资源报告，是监控数据的主要来源和计算数据依据，在此基础上就可以使用公式($cstop - $cstart) / ($tstop - $tstart) * 100 获得 CPU 的利用率，其中 cstop,cstart,tstop 是通过读取对应的 Cgroup 子系统当中的数值来获取。联系上文提及容器的实质是文件级别的资源隔离，假设监控的对象是一个 Docker 容器，当需要读取其运行数据时就可以通过将它的 Cgroup 目录下的文件挂载到功能的容器当中，获取到这个容器的运行状态，当然这是在主系统权限允许的状况下；同理宿主机或是其他的容器对象都可以通过这个途径来获取到对应的运行数据。

现有许多成熟的监控获取工具，能够实现精准的对象监控。从子系统当中获取的数据是主要的监控对象，即便是单独实现的监控主体也需要回到这个层次来获取到自己需要的信息，但对于 Kubernetes 分布式集群来说，需要的监控数据还涉及到更广的范围，这就需要将监控数据的收集工作分配出去，让更成熟的收集方案来解决。这方面的工作可以由 Cadvisor、Docker Stats、 Node Export 等资源数据收集方案，他们都是基于 Cgroup 所作的资源收集来作进一步的数据处理，能够提供大部分基础数据的监控样本，通过他们提供的开发 API 就可以直接获取到大部分的监控数据，但是为了能够满足功能性质的监控系统开发，在 Cgroup 子系统的基础上，依旧需要自行设计读取某些数据，比如当在一个整体的集群当中，需要统计整个集群服务负载某一系统任务时候的时延和负担，就需要提供监控单个机子上面的时间序列。综合以上监控数据资源的办法，本文设计了自己的数据获取途径和开发接口。

第三节 InfluxDB 数据库以及 Gin 框架

InfluxDB 是由 InfluxData 开发的开源时序性数据库，在 DB-Engines Ranking 时序性数据库排行榜上位列第一[11]，能够优秀地处理海量时序数据的高性能读取，存储和实时分析任务，已广泛地应用于 IoT 监控，实时分析,DevOps 观测等任务。

在技术方面，InfluxDB 基于 GO 语言开发，无需任何外部依赖就可以独立部署，其主要的技术特点在于提供类似 SQL 的查询语句，良好规范的 API 接口，使用方便；丰富的聚合运算和采样能力，有着灵活的数据保存策略（Retention Policy）来设置数据的保留时间和副本数量，支持将高价值高精度的数据保存在内存当中，将相较不重要的数据保存到磁盘当中，也使得其比起其他数据库有着更高性能的表现；同时在保障数据的完整性前提之下，及时删除过期的低价值数据，提供灵活的连续查询（Continue Query）。

InfluxDB 存储的是时间序列数据，也就是以时间为标签的数据序列；具体来说时序数据有着 0 个或者是多个数据点，数据点的构成包括一个时间的标签和其对应的数据内容。在对数据访问存储和提取上面其支持多个协议包括 Http,UDP 等多个原生协议，这就意味着对于 Restful 形式的接口可以有较高的融洽度，可以通过类似于 URL 的形式来沟通，具有服务器的性质。

展示界面是基于 Gin 框架开发的前端页面。

Gin 是使用 Go/golang 语言实现的 Http Web 框架，路由基于 Radix 树，不使用反射，需要的内存极少；内置组件支持 Json、XML、Html 的渲染；自主定义编写的 Http 中间件和路由规则，能够比较好地适应特征功能性质的系统搭建，作为展示部分的框架和主心骨。

前端界面展示 Html 组件设计使用了开源项目 Echarts, 它是使用 Javascript 实现的可视化网络组件库，底层依赖矢量图形库 ZRender，能够设计出直观，交互丰富，可以高度个性化定制的数据可视化图表，具体提供的图形类型包括常规的折线图、柱状图、散点图、饼图、K 线图，用于统计的盒形图，用于地理数据可视化的地图、热力图、线图，用于关系数据可视化的关系图、treemap、旭日图，多维数据可视化的平行坐标，还有用于 BI 的漏斗图，仪表盘，并且支持图与图之间的混搭，其前端数据展示技术采用了增量渲染技术（4.0+），配合各种细致的优化，能够展现千万级的数据量； Echarts 提供了对流加载（4.0+）的支持，使用 WebSocket 或者对数据分块后加载，意味着在数据展示界面和后台数据库交互之间也可以采用推送的形式，只要产生了新的数据，就将其推送给前端界面并展示出来，实现实时信息更新。

第三章 系统模块技术实现及细节

第一节 系统的总体架构

本文基于 Kubernetes 架构实现了边缘云容器集群的监控系统。系统的功能期望是实时监控 Kubernetes 集群的状态。具体来说，需要在边缘获得集群状态信息并进行有效处理，然后传输数据到控制节点上展示。主要面临的需求是在边缘云容器上对数据的收集和处理，集群内部的信息传输以及主节点的数据整合与前端展示；本文工作依据这三点展开，分别设计了对应的模块解决问题。

监控系统的设计主要分做三个模块，总体架构设计如图 3.1 所示。

首先是系统基础设计，监控系统依靠 Kubernetes 架构，将系统功能以容器的形式部署到边缘云集群当中，容器的生存周期和资源消耗都由 Kubenetes 管理，能够保证服务的持续性提供，也能通过 Kubernetes 实现对整个集群的统一协调安装和控制。

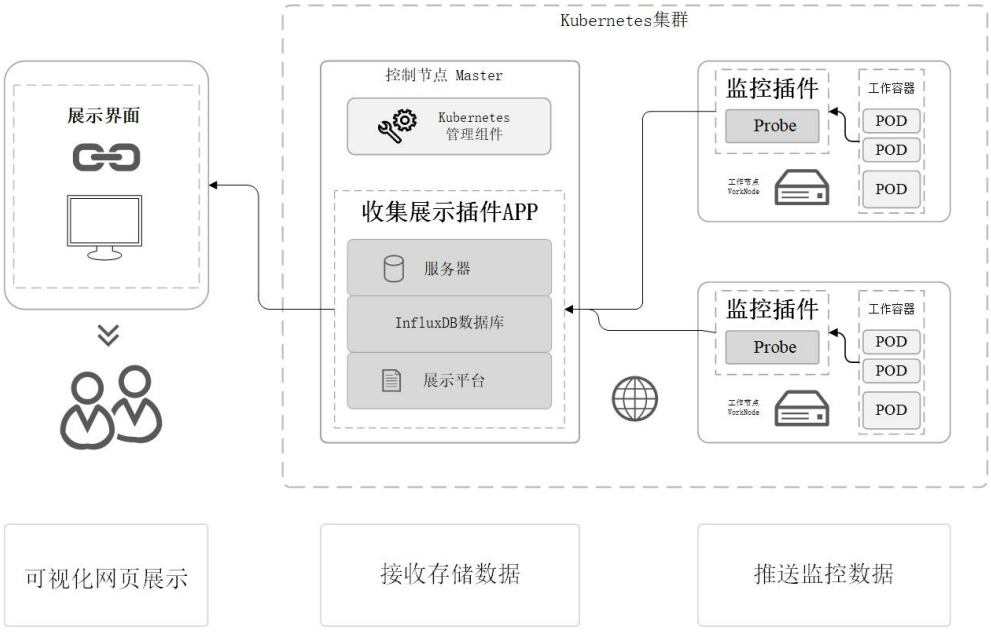

图 3-1 监控系统总架构设计图

系统主体架构设计包括两个模块：

Probe 模块是部署于工作节点的批量型监控插件，需要收集监控对象的状态数据，并在计算处理之后发送到控制节点。功能上，监控对象包括节点机器、容器、应用等，并针对不同的监控对象和场景有着不同的监控数据收集策略；整体网络连接上，Probe 模块采用推送方式解决同主节点的连接问题；架构上，需要将其批量部署于每一个工作节点，依靠 Kubernetes 架构管理模块的生存状态。

App 模块是部署于控制节点上管理调度与显示集成的插件，需要接收整个集群节点发送的信息并存入数据库，同时使用用户友好界面显示出数据，展示集群的状态，实现监控的功能。主要分作三个部分的工作实现，搭建 WebSocket 服务器接收节点发送的信息，连接 InfluxDB 数据库并依据键值生成数据点存入，生成展示界面并在有数据更改时刷新界面，最终实现对整个集群状态的监控。

第二节 Kubernetes 监控系统实验平台搭建

本文依据 Kubernetes 系统的构建原理，以命令行的方式构建整个实验平台和模拟的监控环境。

3.2.1 节点网络拓扑结构

监控系统的实验平台设计是三个物理机组成的模拟网络，其拓扑结构是分作两层的 Kubernetes 集群。以一台服务器作为控制节点（Master Node）、两台实体机作为工作节点（Work Node）组成基础的 Kubernetes 网络结构；各个节点之间需要网络互通可达。在生产或者研究环境当中，机器的拓扑结构几乎都可以从构架逻辑上划分成为以一个主控节点，多个工作节点为核心的三角形部署形式，从逻辑上可以划归做控制部分和工作部分。

在 Kubernetes 运用场景当中，一般工作节点同控制节点需要在同一个局域网当中，相互之间可以网络连接，比如工业场景当中多组机组作为工作节点，而控制室的平台作为控制节点，或者是校园内每个摄像头作为工作节点，机房的服务

器作为控制节点。见图 3-2，这个拓扑结构设计能够覆盖绝大多数集群状态要求。

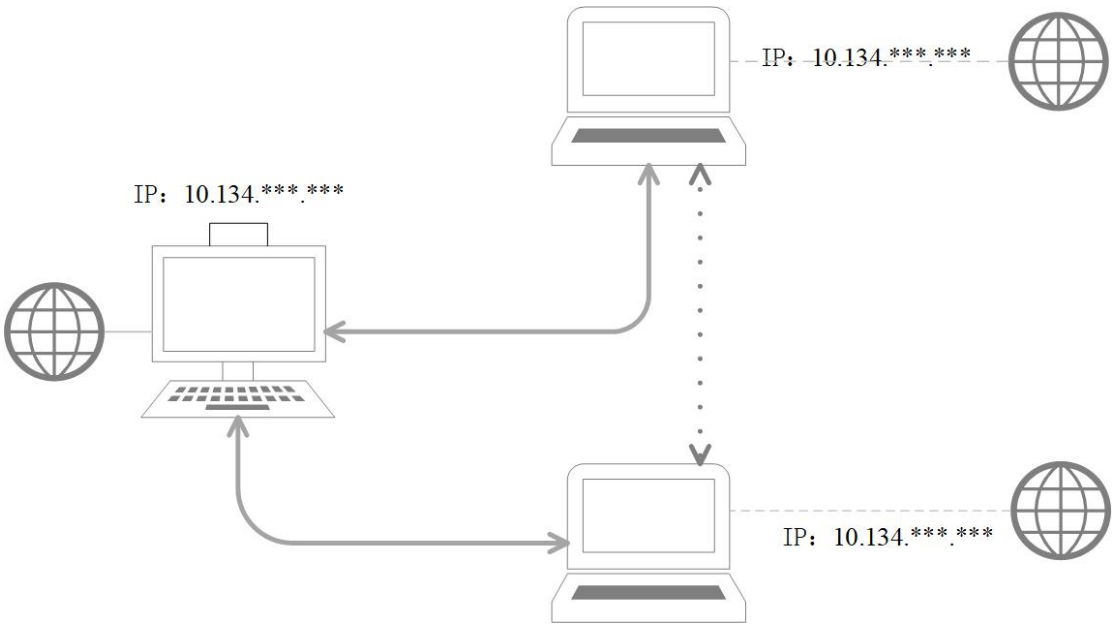

图 3-2 物理机网络结构图

具体的设置方案是通过创建一个虚拟网卡将所有的物理机都分配到同一个网络区段。首先通过 VM 服务器设置虚拟网卡并设置实验的网络区段 IP，之后修改每一台实验机的/etc/netpaln/01-network-manager-all.yaml 文件，停止 Dhcp 功能，配置掩码，同时配置该实验机的 IP 地址；本文的节点 IP 设置如表 3-1：

表 3-1 实验机 IP 设置表

节点	IP 地址
Master 控制节点	10.134.66.4
Node1 工作节点	10.134.115.2
Node2 工作节点	10.134.91.18

最后还需要修改 Hosts，并将设置的 IP 地址和域名写入文件，保证路由的畅通。

3.2.2 基于 Kubeadm 的集群部署

Kubernetes 实验集群部署的必要环境条件需要集群当中的每一台机子都部署和安装，包括 Linux 系统的 Http 通讯组件和 Docker 依赖,如果监控对象涉及 GPU

计算，还需要安装 CUDA 等必要的操作系统模块；集群中所有节点都是同样的基础配置，某种程度上说明了在 Kubernnetes 集群当中，真正区别节点的是部署状态，是运行的容器，这也是 Kubeadm 部署的理念。

用 Kubeadm 部署集群，就是将系统的组件以容器的方式部署在实体机上；根据节点机的不同角色来启动不同的功能容器，承载不同的集群任务。传统的构建方式是需要将所有的系统组件以二进制的方式运行在实体机上，这就需要付出单独的编译运行和重新修改配置文件的资源和时间；从基于效率和开发的角度，Kubeadm 是更加轻便也更适合容器编排系统的部署方式。

实现细节是在每一台实体机上都以二进制形式编译安装 Kubernetes 管理组件 Kubelet，Kubectl 和 Kubeadm；再用 Kubeadm 安装 Kubernetes 各个功能容器，最后手动部署集群的其他功能容器。三个管理组件的安装通过 apt-install 以管理员权限执行即可。

Kubeadm 安装集群的具体内容是，先检查集群当中各项要求是否达标，是否可以在网络稳定地情况下配置稳定的集群状态，如果通过了 Preflight Checks 检测则继续安装；在此基础上，生成 Kubernetes 访问 APIServer 的各项证书，并将他们存入对应的目录（Master/etc/Kubernetes/pki）之下,主要的是证书文件 ca.crt 和私钥文件 ca.key，其他证书包括使用 Kubectl 请求 apiserver-kubelet-client.crt 文件，对应的是 apiserver-kubelet-client.key 私钥文件。之后为控制节点生成功能 Pod 的组件，包括 Kube-APISever,Kube-Controller,Kube-Scheduler，这些集群组件是以 Pod 的形式部署到集群上的；连接 Ectd 之后完成集群 Master 节点的部署。

命令行使用 Kubeadm 生成集群的 bootstrap token，届时 Kubeadm 会将 ca.crt 等重要的访问权限信息通过 ConfigMap 的方式存放在 Etcd 当中，最终通过在任何一个安装配置好 Kubelet，Kubectl 的节点上，运行指令“kubectl join 连接符”就可以将这个节点加入到以配置好的 Master 为控制节点的 Kubernenetes 集群当中，这样基础的 Kubernetes 实验集群搭建完毕。

基础的 Kubernetes 实验集群搭建完毕，最后本文工作当中需要除了 APISever 以外的访问和沟通方式，还需要安装 Kube-proxy 和 DNS 插件，提供整个集群的服务发现和 DNS 功能。配置步骤是编写容器对象资源文件 yaml，下载 Kube-Proxy

Docker 镜像和提供网络连接的 Flannel Docker 镜像，再通过 Kubectl 命令行使用指令“kubectl deployment ****.yaml”文件，在集群节点上生成功能容器。

如果要应用在工业界，其需要部署多层次，多维度的 Kubernetes 架构可能需要通过 kOps 等更成熟的方案去处理，数据库、各层交互等问题，但是其原型依旧是回归到本文所搭建的基础两层架构之上，从研究角度来说，有足够的可拓展性和代表性。

第三节 系统数据监控插件 Probe 设计

Probe 是运行在每一个节点当中的数据收集插件，它能够完成对监控对象的数据收集和计算处理，并通过推送的机制发送给 Master 节点的数据收集模块。本文设计 Probe 包含三个模块的功能，每个模块的功能由对应接口实现，通过设置 Probe 结构体将数据和三个模块联系起来,概念模型架构见图 3.3。

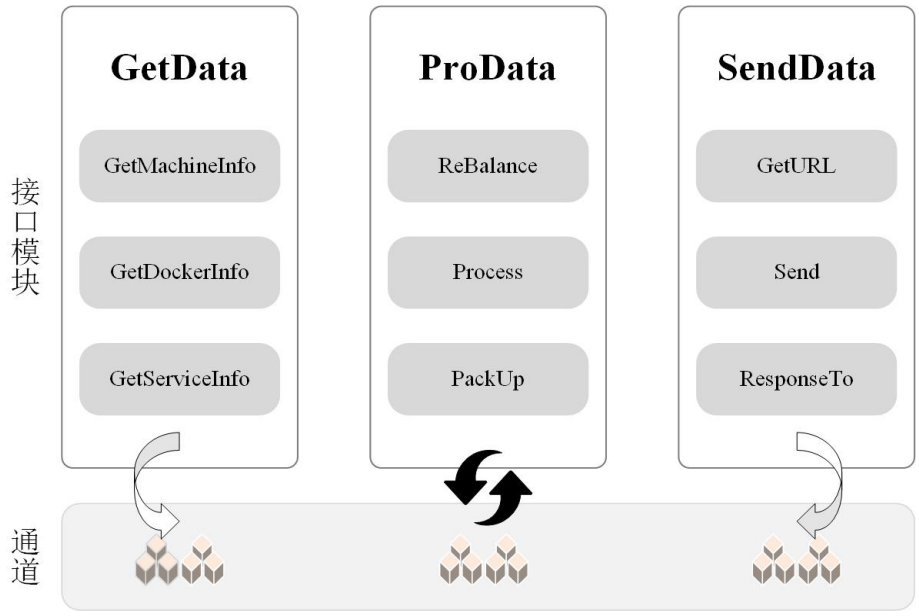

图 3-3 Probe 插件结构设计图

Probe 数据结构包括四个属性，分别是 UrlRc，UrlSend，Rrc，Wrc。UrlRc

存储获得监控数据的路由，或者说 API 对象资源路径，需通过配置静态文件获得或者在 GetURL 当中设置方法启动对应的服务获取。具体对象可以是自定义的监控 Node Exporter，也可以是其他监控工具的日志报告，主要针对的是 Restful 接口设计；UrlSend 表示发送数据的对象位置，主要指的是 Master 节点数据收集插件监听的位置，其形式为“WS://MasterIP:Port”，Port 由协议规定，取决于数据收集端的设置，同样为保持组件的完整性有配置文件来设置；WaitGap 作为数据监控数据收集的时间周期控制变量，具体的数据结构定义见图 3-4。

```
1.    Http Probe struct{
2.    URLRc    string
3.    URLSend  string
4.    Wrc      chan []byte
5.    Rrc      chan []byte
6.    WaitGap  time.Duration
7.  }
```

图 3-4 Probe 插件数据结构代码设置

Rrc 和 Wrc 是通道变量，在 GoRoutine 当中充当进程之间的通讯通道，将分别承担传输读取的监控数据和发送的处理数据的任务。由于所有的部署组件都是运行在 Pod 当中的，容器归属 Kubernetes 管理，Kubernetes 会维护在容器当中的环境变量以保证对节点、对容器、对 API 对象的控制，所以可直接从容器当中的环境变量获取到目标的地址。

接口设计包括 GetData,ProData，SendData，通过依据不同系统要求实现这三个接口内的函数，来完成三个模块的功能。GetData 接口当中定义了 GetMachineInfo(),GetDockerInfo(),GetServiceInfo 等行为函数，用于获取监控不同对象的数据信息，并将获得的信息存储为[]byte 型变量赋值给 Rrc 通道；ProData 接口当中需要完成 ReBalance(),Process(),PackUp()动作内容，调节当前节点的监控策略，计算处理监控信息，打包分装处理之后的数据；SendData 接口定义了 SendData()，ResponseTo()，GetURL()函数，实现传输客户端，能够处理和响应来自服务器端的请求，获得需要发送监控数据的对象地址,核心技术是推送到

Master 节点的信息传输机制[12]。

3.3.1 监控模块的设计

监控模块需要完成功能是收集监控节点的信息并存入[]byte 通道变量。具体的方式大体可分做使用工具和直接读取源。对于一般监控场景，成熟的监控工具提供了多样的 API 接口，通过创建访问实例，监控模块只需要不断调用方法就能够获取常见的监控数据；直接读取源数据，根本是从操作系统的日志文件当中获取特定的监控指标，尤其在节点身份识别和特定观察对象时候有着极其重要的占比。

监控模块的设计是设置了数据监控的调用接口和统一的数据结构格式。不同监控工具的实例对象申明和方法调用放入对应的接口当中，通过不断地发起读取操作获得监控对象的元数据，在此基础上按照 key 值对应存入数据结构地对应变量。设置统一的监控格式是为了规定统一的数据收集模式和数据规范，以字节的形式传输到数据处理模块。

本文主要收集的对象主要是容器为核心的监控服务，依旧按照 Kubernetes 对象资源申请的方式做划分，主要的监控工具是 Cadvisor[13]，包含读取节点 IP，DockerID 等集群状态信息。

模块首先配置 Cadvisor 功能容器，实现实例获取数据。Cadvisor 是谷歌开源的资源监控方案，它能够做到对节点机器上的资源以及容器进行实时监控和性能数据的采集，包括网络资源使用，CPU 使用情况，内存使用情况。该组件由 Go 语言开发，集成在 Kubernetes 当中，在 Kubelet 里作为默认的启动项；一般工业场景当中的 Kubernetes 集群可以直接访问该服务，但使用 Kubeadm 时候需要自己再次安装配置该服务，同样是以容器的形式部署在节点上面，依照与 Probe 相同的部署方式。获取监控数据的具体方式是实现 Cadvisor 对外提供的 API 接口——Client,，调用 Client 当中各个数据监控的对象方法，比如 MachineInfo，来获得当前节点物理机的状态。同时将所获得的数据赋值给自定义监控 GetData 实例的资源结构、存储数据，完成一个监控数据的收集，其他 Cadvisor 监控数据的获取方式相同。

Cadvisor 的容器指标主要集中在利用率，饱和度，错误报警的归纳之上，其中主要的几个监控数据显示如表 3-2：

表 3-2 Cadvisor 调用 API 对象函数及其对应指标

Cadvisor API 对象	监控指标
container_cpu_user_seconds_total	“用户”时间的总数（即不在内核中花费的时间）
container_cpu_system_seconds_total	“系统”时间的总数（即在内核中花费的时间）
container_memory_cache	页面缓存的字节数
container_memory_usage_bytes	容器当前内存使用情况
container_network_receive_packets_dropped_total	容器网络数据包丢失数

但是具体可收集的数据类型却不止这些，比如本文监控模块设置的实体机监控数据结构体前段实例如图 3-5 所示。

```
1. Http NodeMachineInfo struct {
2.  // The machine id
3.  MachineID string `json:"machine_id"`
4.
5.  // The number of cores in this machine.
6.  NumCores int `json:"num_cores"`
7.
8.  // The number of physical cores in this machine.
9.  NumPhysicalCores int `json:"num_physical_cores"`
10.
11. // The number of cpu sockets in this machine.
12. NumSockets int `json:"num_sockets"`
13.
14. // Maximum clock speed for the cores, in KHz.
15. CpuFrequency uint64 `json:"cpu_frequency_khz"`
16.
17. // The amount of memory (in bytes) in this machine
18. MemoryCapacity uint64 `json:"memory_capacity"`
19....
20.}
```

图 3-5 Probe 插件监控数据对象代码定义截取

Probe 监控数据结合具体的功能、监控对象、时序等综合信息。在基础的监控数据类型之上，GetData 实例需添加包括节点标签，收集时间，监控对象类型以及 ID 等识别信息，再对监控数据进行初步封装，并在收集完成之后将其转化为 byte 类型的数组赋值给 Probe 的 Rrc 通道量，完成数据收集的工作。当监控的方式从 Cadvisor 更改为其他监控工具包括 Node-exporter 等方式时，就可以根据其监控数据对象实现 GetData 接口，并选择添加其他具体的数据，由于 ProData 接收的是字节形式的数据并同处于相同数据 API 路径之下，就意味着不用担心数据格式不同而无法解析的问题，监控数据的内容格式对于三个模块来说是完全共享的，这一点能够让这个监控系统有着相当高的迁移性质和功能化特性。

收集数据的心跳时间是由主流程的 WaitGap 来控制，其具体数值决定了每隔多长时间收集一次监控数据。考虑到监控对象资源的使用情况有可能在短时间内不发生变化或者说其状态维持稳定持续一段时间，不间断地监控数据收集工作会加大节点的工作负担和计算资源消耗。

本文系统架构会通过在数据处理模块的 ReBalance 当中的均衡算法来修改 WaitGap 的值，以提高节点监控数据的收集效率，降低多余的监控资源消耗，优化资源收集的频度，同时也能够产生连锁反应，优化包括数据计算、数据传输、数据解析等环节的资源损耗情况。这个值也是整个系统周期运行的推动值，决定着整个系统架构工作的频率和效率，通过设置其变化可用于契合大多数监控收集工具为保证数据完整度而产生的时间延迟，同时这个数值本身也能作为任务时延的监控计算数据之一，对于单个节点数据变迁做了划分处理。

以 Cadvisor 实现 GetMachineInfo 接口获取监控物理机数据为例，展示监控模块核心代码。根据监控数据的工具定义具体的数据结构 Cadvisor，实现 GetData 接口监控的数据对象，具体见图 3-6，在主进程当中调用获取监控对象信息。

```
1. var Client DataInfo.GetData
2.     Client.GetMachineInfo(Pb.URLRc ,&Pb.Rrc ,Pb.WaitGap)
```

图 3-6 Probe 调用监控模块

以 GetMchineInfo 为例，传入获取资源的路径和通道变量，先声明 API 实例 NewClient，再通过 Cadvisor/Info 下提供的接口实例化访问 Cadvisor 的对象，以逐个读取的方式，作赋值轮番获取需要的变量，实现过程见图 3-7。

```
1. func (ca Cadvisor)GetMachineInfo(){
2. //创建访问的实例
3.     Client, err := client.NewClient(url)
4.     for timegap {
5.         //按序调用，轮询赋值
6.         ca.HostID = os.Hostname()
7.         ca.MemoryCapacity = Client.MemoryCapacityGet()
8.     }
9. }
```

图 3-7 监控插件获取数据核心逻辑

这样的设计还需要对整个系统进行多点一线的数据类型验证同一，使功能和结构一一对应。为完成对监控数据的尽可能覆盖，根据监控的功能，需调用尽量多的监控对象并获得数据。

这个方式能够实现多种监控工具与方案的统一。所有监控数据最后整合到创建的 GetData 对象，Probe 只需要使用 GetData 对象就能够调用到监控模块当中的获取数据函数而不必在意到底使用的是什么工具，这一点也让监控模块的数据能够糅合接入多样的指标。

3.3.2 数据处理模块的设计

数据处理模块编写整个监控系统的分布式计算任务，优化算法和调度算法。这一模块能够获取监控模块的监控数据，同时需要将处理之后的信息存入 Wrc 通道变量传输给数据传输模块。依据对监控系统的功能分析，规划系统的计算功能包括调节监控的策略，边缘数据的处理计算，重封装传输数据。

方法 ReBalance 负责调节读取数据的频度和次数。监控数据的收集本质都是读取 Log 文件内容。对于监控数据繁多的场景，怎样有效地提高读取数据的价

值比是最需要考虑的问题。优化读取数据的方法尤为重要，基本的读取方法是在间隔一段时间重复相同的动作，但是间隔多长时间，这个数值关系到读取的方式和监控实体的状态。

监控策略需结合系统特点。对于本文工作设计的对象，Kubernetes 监控管理的容器系统：一方面承担着大规模，长时间，持续性质的应用型服务，比如气温观察，监控影像存储，数据包转发路由等等，对于资源的使用状态和占比很可能长时间处于同一水平，或者有时候会一直处于为消耗资源的状态，重复轮询所得的数据之间差距并不是很大，那么延长监控数据的收集时间，结合当前收集数据的状态，减少对这个监控对象的询问次数，能够减少不必要的资源损耗；另一方面，某些节点达到某些条件之后会出现状态突变的情况，模块必须在侦测到变动时采用频繁的数据采集策略，立刻获取当前监控对象的状态。

同样环境下，边缘计算 AI 模型的应用，大计算量的应用，一个节点在短时间内的数据变化量会有较大的变化，尤其是发生了集群级别的资源调度，单台实体机上面的容器对象很可能会有巨大变化，这样的情景之下就需要时时刻刻监控节点和资源对象的变化，才能够使得调度策略和算法参考数据是有价值和意义的。

本文主要采用了负反馈的策略，通过比较两次相邻监控数据之间的差别，当这个数据差别在一定的时间区间内维持为 0 时，就会延长系统调用监控模块的时间间隔，减少系统的资源消耗，当某次差别不为 0 的时候就会立刻读取监控状态并缩减间隔时间为最短。

具体的设计如图 3-8：

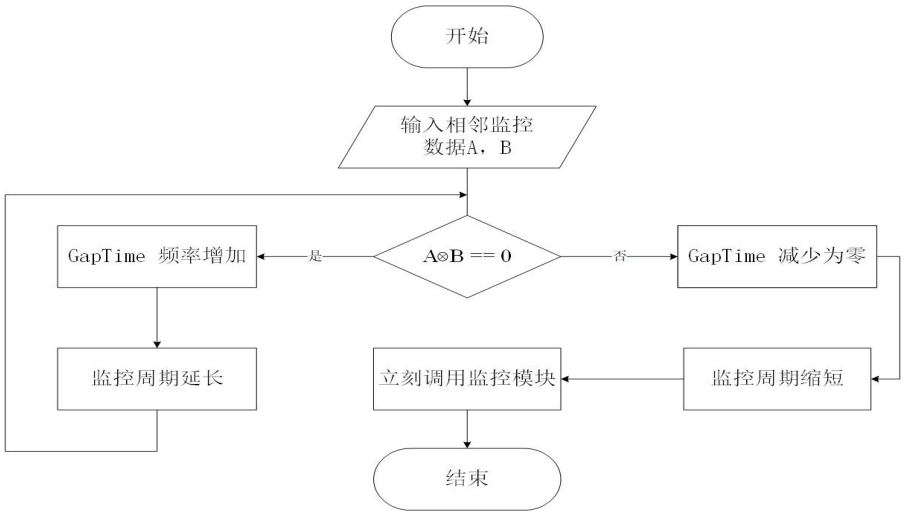

图 3-8 监控模块读取策略负反馈调整策略

ProcessData 通过不同的计算对象实现，应用不同的计算功能。实现数据处理对象 Manufacturing，接收读取的监控数据 chan []byte，应用 Process 函数，实现功能性计算；依据边缘计算设计的理念,将模板式，机械式的数据处理放置于节点，减少数据传输和主节点存储的负担，提高节点计算资源的利用率。比如将此系统运用于云边端联邦学习模型的应用场景当中，系统的功能指向是寻求模型的模拟结果，就可以在这个模块设置方法计算任务监控对象容器占用 CPU 计算资源的比重或者是多次计算之间的时间差，来判定这个模型的适配度，并给出综合的监控数据比对结果，再传输给主控节点做调度。

调用计算模块，首先需要申明接口对象并实例化，设置需要计算和读取的字节二进制形式的通道变量同时获取处理完毕的数据，调用计算模块见图 3-9.

```
1. var Client DataInfo.ProcessData
2.     Client.Process(Pb.Rrc ,&Pb.Wrc )
```

图 3-9 Probe 调用计算模块

功能实现范式是根据需求创建计算结构 Manufacturing，通过该对象调用模块接口，见图 3-10，在其中编写或者是调用设计的算法。

```
1. func (one Manufacturing)Process(Rrc chan []byte,Wrc *chan []byte)
   {
2. //计算模型的适配度
3.    one.CPUMath
4. }
```

图 3-10 Probe 插件算法模块使用思路

PackUp 接口提供数据再包装，控制传输的数据格式、数据种类。考虑到并非所有的监控数据都是需要的数据以及将监控模块作为功能容器可单独运行的理念机制，设置了对发送数据进行筛选打包的环节，根据用户需要的监控数据来定义发送的数据结构，相较于更加详细冗长的数据报告，专属性轻巧的数据组更适合专用的监控场景。

3.3.3 数据发送模块的设计

数据传输模块将监控数据封装，按照一定频率推送给 Master 节点的服务器，同时需要接收来自主节点的控制信息，做出报警反馈；功能分析上最主要解决的是 Kubernetes 集群当中工作节点同控制节点的通讯问题。

本文 Send 方法采用推送的方式，由工作节点主动向控制节点推送。以往的数据传输形式是通过 Master 节点发起轮询请求，监控 Probe 才将监控数据发送出去，也就是轮询的方式。在这个问题当中最核心的是地址和通讯相关的问题，在 Kubernetes 架构之下 Master 节点通过 APIServer 控制整个集群，获取集群的状态，其他主体也需要通过这一层才能够访问自己部署的服务；在这样的情况之下 Kubernetes 外的操作实体想要直接控制访问 Kubernetes 内部的各个节点就无法绕开这一层封装的影响，如果从 Master 节点以轮询的方式向各节点发送请求来实现监控，就必须要通过 APIServer 找到对应节点实际的 IP 地址来通讯或者就只能读取 Master 节点上面由 Kubernetes 自己采集的信息。

直接访问工作节点，当集群的规模较小且部署的应用对象较少时是可行的；但是当集群变得较大的时候，会容易造成请求未到，服务调度不见的问题。这点绕不开以 APIServer 为核心的部署机制，Kubernetes 集群以 Deployment 为准部署

容器，能够保证集群内 Deployment 所请求的服务和资源不会因为某些节点机器的故障而损失——Kubernetes 会找到满足要求的节点重新部署应用，这个过程对于用户和操作者来说是透明的，就好像某个节点故障的事情没有发生一般，使得服务能够持续化提供，但若想在这样的机制下通过一个实际的 IP 地址，继而访问节点就出现了问题；系统是无法保障获得地址能够访问，甚至有可能无法确认监控的容器是否还在这个 IP 机器当中。

访问 Kubernetes 在 Master 上维护的文件受限大且不准确。这个方法解决了网络访问的困难，却必须在 Kuberentes 所能监控的数值指标当中选择能够使用的数据，同时也受到 Kubernetes 自身数据分析的限制，紧紧地绑定 Kubernetes 会使系统很难做到理想的功能监控，同时也很难保证所获得数据就是期望的监控状态。

不论是在那种场景之下，问题的根节在于由监控主体发起轮询再获得期望数据的机制。在这种情况下，请求数据、选择数据动作都集中到 Master 主体当中。从 Kubernetes 容器分布式架构的理念出发，应当将计算、发送给等任务都分布到各个节点当中，管理节点是沟通的枢纽而不是所有一切任务的集中对象，让 Master 更多地作为被动接受结果的主体。

为解决这些问题，本文采取的推送策略，任务边缘化处理——节点完成监控数据的初步计算和处理、对数据打包，由 Node 节点主动推送给 Master 节点而无需等待 Master 节点的请求。这样设计的好处在于：针对本文所讨论的监控系统，需要长时间大量轮询工作以及有着大量的监控对象的应用场景当中,能够有效降低重复的建立连接的操作、减少路由寻址的困难、提高信息传输的能力、为节点互通提供了低成本方式[14]。

本文采取的网络通讯协议是 WebSocket 协议，每个 Node 节点上都实现 WebSocket 的客户端，使它能够固定向 Master 节点的服务器端发送数据，同时 Master 节点的收集器接收 Json 文件解析之后根据 Key 值存入数据库当中。

WebSocket 是一种在单个 TCP 连接上进行的全双工通信的协议，允许服务端主动向客户端推送数据[15]。在 WebSocket API 中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。依据 WebSocket 标准，WebSocket 通信协议建立在 TCP 基础上，他应当在握手结束后保持一个长连接，打开交互式通信会话，这样客户端就能够向服务器发送消息并接收事件驱动的响应，而无需通过轮询服务器的方式以获得响应，具体通讯过程如图 3-11 所示。

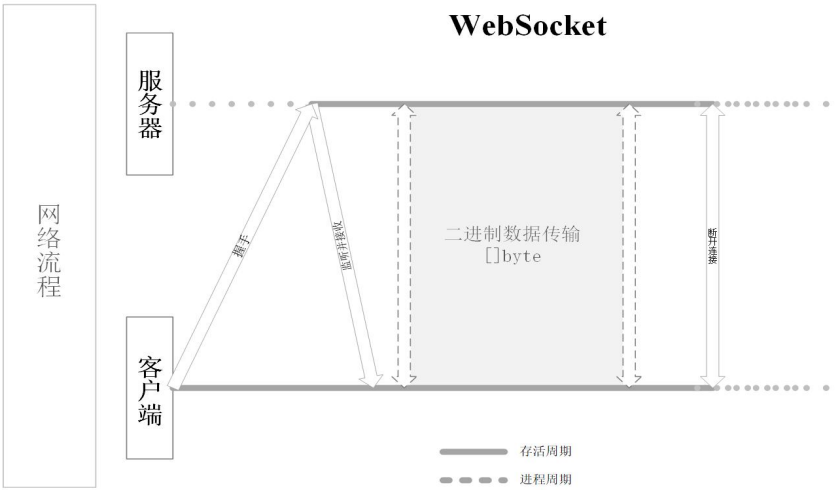

图 3-11 WebSocket 通信模型

如果能够自主地发送信息，就不需要 Master 重重复复地建立连接发送请求维护状态，而 WebSocket 只通过一次握手，建立持久连接就能够主动推送，很大程度上能够减少长轮询等类似方式所导致的网络性能损耗问题。从 Master 发起的轮询，在 Http 协议中，需要先经过主节点服务器解析，再传送给相应的 Handler 来处理，这样每次信息传输都要重新鉴别信息，来告诉服务端获得的数据是什么，造成了资源性能的损耗；相对的，WebSocket 建立连接之后，就能够不用重复以上行为，只需要传输数据一直到有断开连接的请求为止。同样的方法在 Kubernetes 开源社区的解决方案当中称之为 Pull，目前社区采用的最为成熟的监控方案 Prometheus 也采用了类似的数据采集模式。

具体工程上，Send 方法使用了 Gorilla Tool Kits 实现网络连接。具体通过 Upgrade 的方式在建立好的 Http 连接上面升级成为 WebSocket 连接,在此基础上编写 Handler 来执行相应操作，首要创建 Gorilla 数据结构并实现 Send 方法

```
1. 1. var Client DataInfo.SendData
2. 2.     Client.Send(Wrc chan []byte, Target string )
```

图 3-12 Probe 调用数据传输模块

Gorilla 是从 Go 的 Http 升级而来，通讯的过程依旧是按照 Http 的流程来做，通过连接的对象 Conn 实现连接功能 由于网络连接编写组件包含客户端和服务器端的内容，所以这个部分的说明也包含了在 Master 节点接收推送信息的服务器端。每个节点上面的客户端能够获取从 Wrc 通道当中读取 ProData 处理之后的 []byte 数据。这部分逻辑流程调用是在 Http 的基础上添加了 WebSocket 的调用规则，客户端创建推送客体的 Conn 对象，这个对象结构就是 WebSocket 协议之下设置的结构，包含了 Web-Socket 需要的网络组件，通过调用 WriteMessage(mess -ageType int, data []byte)接口建立连接。

```
1. 1. func(sender Gorilla)Send(Wrc chan []byte, Target string){
2. 2.  func (c *Conn) WriteMessage(messageType int, data []byte)
3. 3. }
```

图 3-13 传输模块核心函数

前者需要设置为传输的数据类型，后者是传输的数据对象，编码设置需要将其申明为 BinaryMessage 类型；表示传输的是二进制文件，而后是具体赋值的通道变量。WebSocket 底层传输是以 Frame 的形式传输的，这种形式是为了避免数据传输过程当中的数据缺损，通过将 Buffer 当中的数据以帧为单位发送出去，所以还需要调用 ReadMessage 将信息存到 Buffer 当中，每当有监控数据的心跳包存入通道，他就会将这些数据发送出去，同时等待网络连接结束的信号。

对于接受推送信息的服务器端，需要编写 Upgrader 来讲连接升级为 WebSocket 连接，Gorilla Tool Kit 通过设置参数 Check Origin 为 True 使连接变为 Websocket, CheckOrigin 设计用于拦截和放行跨域请求。

连接的具体过程是：首先创建 Http 服务，然后再次通过 WebSocket 对象 Conn 调用这个 Upgrader 接口，将此时 Node 和 Master 通讯升级成为 WebSocket 通讯。

当建立连接之后，需要依据 WebSocket 的特征，注册 Handler 监听 Node 发送信息的端口，并接收[]byte 二进制类型 Json 数据，完成数据格式解析之后存入 InfluxDB。

3.3.4 系统数据监控插件 Probe 部署

将工程代码打包成 Probe 镜像并通过 Kubernetes 以 DaemonSet 的形式部署在整个集群上面，实现对节点状态的监控

```
E:\WORKSTATION\K8SSCS\PROBE
| go.mod
| go.sum
| main.go
| Dockerfile
| tree.txt
|
├ MyProbe
```

图 3-14 系统镜像工程目录

编写 Dockerfile，解决依赖环境相关问题，并上传 DockerHub。Dockerfile 当中最需要注意的是文件挂载问题，需要将工作目录下的代码挂载到容器当中来执行。

```
# 将代码复制到容器中
COPY . .
# 将代码编译成二进制可执行文件  可执行文件名为Probe
RUN go build -o probe .
# 移动到用于存放生成的二进制文件的 /dist 目录
WORKDIR /dist
# 将二进制文件从 /home/node/Probe 目录复制到容器当中
RUN cp /home/node/Probe/probe .
```

图 3-15 Dockerfile 主要内容

从 DockerHub 下载镜像，编写 DaemonSet，命令行执行部署在集群当中。在 Kubernetes 集群当中，以 DaemonSet 形式部署的容器将会在每一个节点上面维持长期运行的状态，由 Kubernetes 维护和保障其生存状态资源消耗。需要通过编写 Yaml 文件来申明运行程序的 Pod 主体，就像是申请好运行程序的一台实体机子一样。其中最主要的几个设置包括 Kind 需申明为 DaemonSet 类型，使用的镜像应当是本文提供的 Docker 镜像 probe:v0.1，若提前下载好镜像，则还需要修改镜像拉去规则 ImagePullPolicy 为 IfNotPresent，让 Kubernetes 当本地有这个镜像的时候就直接创建资源对象。

```
1. apiVersion: extensions/v1beta1
2. kind: DaemonSet
3. metadata:
4.   name: probe
5. spec:
6.   template:
7.     metadata:
8.       labels:
9.         app: monitoring
10.        id: probe
11.      name: probe
12.    spec:
13.      containers:
14.      - name: probe
15.        image: benjamindyx/probe:v0.1
```

图 3-16 系统镜 Pod 部署 Yaml 文件

编写好资源声明文件之后通过 Kubernetes 命令行工具 Kubectl 在集群当中创建资源对象和容器实例；其他系统当中的容器资源对象也是同样的创建过程，具体的差别体现在资源文件***.yaml 当中参数设置的不同

第四节 系统数据收集展示插件 App 的设计

数据收集插件 App 设计以两部分组成，系统数据存储模块和系统数据展示模块。系统数据存储模块建立与集群当中每一个节点客户端的 WebSocket 连接，监听并接收推送的信息，同时完成解析数据内容，依据键值存入数据库。系统数据展示模块开发处于应用层，从数据库当中读取监控数据并可视化展示数据，显示当前集群状态。

系统收集模块 Collector 建立 WebSocket 连接，按照 DataInfo 当中的数据结构解析推送的数据，建立 InfluxDB 数据点对象，发起存入数据库的请求；数据展示模块 App 设置 InfluxDB 数据对象，通过赋值请求的方式维持前端界面的数据结构和数据库当中的数值相同，具体机制是数据库数据点对象更新就发起对前端界面的刷新操作。

3.4.1 系统数据存储模块

系统数据存储模块负责收集客户端推送的数据并存入 InfluxDB；InfluDB 和 Collector 均以容器的形式运行在 Master 节点上。

插进部署需首先部署 InfluxDB 数据库。考虑到系统分层的独立性，本文当中的 InfluDB 数据库 Pod 对象是单独建立运行的。首先编写机器资源定义 Yaml 文件，以命令行形式发起建立维护请求，依据静态配置文件生成环境配置。这一步需要导出默认配置文件

```
1. Docker run InfluxDB config > InfluxDB.conf
```

使用该配置文件生成 Kubernetes ConfigMap ,完成环境一致性设置。

容器当中的部署操作步骤以资源对象设置为主。需创建 Namespace InfluxDB 实例，然后再编写 Config 文件；由于 InfluxDB 数据是保存在容器目录的/var/lib/InfluxDB 当中，为了能够保证文件卷持续化，需要另外创建 Pvc 和存储对象，挂载文件卷；基础配置完成后开始布置 InfluxDB 数据库容器，具体的资源设置条件包括部署于 Master 节点和暴露出 8086 端口作为访问途径两点；指定在 Master 节点上运行需要配置 NodeSelector,设置参数为 node-role.Kubernetes.io/master:””；暴露端口设置 Port 为 8086.通过以上步骤部署运行在 Master 节点上面的数据库容

器实例。为了满足之后可能出现的外部主体访问容器服务的情况，还需要为容器创立服务以提供路由，通过 deployment.yaml 和 service.yaml 文件完成此项设置。

数据存入部分包含对数据库的读写操作。InfluxDB 的读入需同网络通讯部分结合，设置在 wsHandler 之中，当接收到完整的数据段之后就执行数据存储工作。数据对象来自于各个节点推送的二进制流，当接收到信息时需创建对应的数据存储项，通过 Http API 形式进行存储访问，存入数据库的形式是 Point 数据对象。InfluxDB 的数据点包括 tags、field、time 三个键值以及匹配相对时间点的数据流，并以此为键值生成操作对象 Client，以数据点的形式存入；在 wsHandler 被调用期间，会一直持续写入 InfluxDB 的操作，依据不同的键值来区别数据点，在界面读取的时候会以监控对象不同来做主要划分，主要的函数调用如图 3-17 所示：

```
1.     client := InfluxDB2.NewClient("Http://localhost:8086", "DYX-1
   711318")
2.     writeAPI := client.WriteAPIBlocking("my-org", "Kubernetes-mon
   itoring")
3.     p := InfluxDB2.NewPoint(“该数据点的 key”, “对应存储的数据列”,“调
   用的接口”,时间)
4.     writeAPI.WritePoint(context.Background(), p)
```

图 3-17 InfluxDB 写入的主要操作

3.4.2 数据展示模块的设计

展示界面是基于 Gin 框架开发的页面，展示界面的 Html 组件设计使用了开源项目 Echarts。后台数据管理的 Gin 框架，完成连接 InfluxDB 数据库并实现读取的操作。代码逻辑框架实现基于 Gin 服务器对象，创建 Gin 实例并加载网络前端界面配置文件。

读取 InfluxDB 数据方面，首要是创建数据读取对象，通过发起 Http 请求，将数据库数值赋值到界面数值的数据结构当中。

```
1. Fun init_Http_server() {
2.  router := gin.Default()
3.    //配置文件
4. router.Static("/font", "./font")
5. router.Static("/images", "./images")
6. router.Static("/picture", "./picture")
7.  router.Static("/css", "./css")
8.  router.Static("/js", "./js")
9.  router.Static("/json", "./json")
10.
11. router.LoadHTMLFiles("./index.Html")
12. router.GET("/", func(c *gin.Context) {
13. c.HTML(200, "index.Html", nil)
14. })
15. router.Run(fmt.Sprintf(":%d", port))
16.}
```

图 3-18 Gin 后端管理启动主要调用

Echatrts 是 Javascript 实现的可视化网络组件库，有多样的动态界面设计。

启用服务器对象，加载网页设计内容，应用 Echarts 创建网页组件，其元单位称之为 DOM 节点，也叫做一个渲染容器，在上面可以嵌入模板式代码实现网页组件对象,其构建的内容的样例如图 3-19

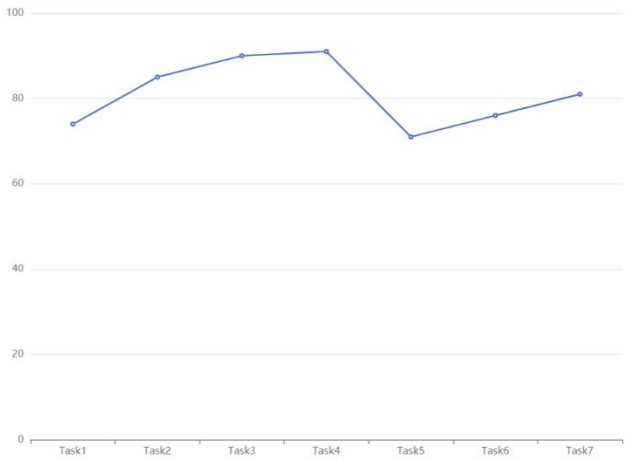

图 3-19 Echart 折线图组件渲染效果

在这个基础上完成对数据库的连接,设置 Controller 读取 InfluxDB 当中的对应数据再推送到前端，并将它存入展示的数据结构当中，需要保持读取数据和存入结构的数据类型一致。

界面 UI 设计着重于数据的展现，依照系统监控的特点，提供选择对象和刷新时间的功能，Web 界面显示如图 3-20。界面顶部设置工具栏，从左到右是系统 Logo、节点警报通知、数据库数据刷新。观测时间周期设置、登录用户这几个组件。

图 3-20 Echart 系统总界面展示图

节点报警设置通过每个节点监控插件的 Alter manager 推送信息判断监控对象的存活，如果没有从数据库当中获取到对应项数据点，就会在生成一条警告。数据刷新按钮完成界面刷新，立刻更新为最新集群状态。监控周期设置按钮可设置界面读取监控数据的间隔

图 3-21 监控界面对象选择及时间选择交互按键设置

第四章 功能测试及拓展开发讨论

第一节 工作负载测试

4.1.1 实验测试环境

本文工作的实验环境主要是用三台虚拟机，一台虚拟机作为控制节点MasterNode，两台虚拟机作为工作节点 WorkNode,在模拟的虚拟组网当中实现Kubernetes 集群。

具体的实体机配置为：

表 4-1 实验用服务器配置

配置项	配置值
制造商	Dell Inc.
型号	PowerEdge R730
CPU 逻辑处理器	32
处理器类型	Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
插槽数	2
每个插槽内核数	8
内存	63.91GB
映像配置文件	ESXi-6.5.0-4564106-standard(VMware, Inc.)

MasterNode 配置同 WorkNode 配置

表 4-2 控制节点及工作节点及虚拟机配置

配置项	配置值
内存	18.4GB
处理器	8 核
硬盘	200GB

在此声明，实现本文工作的系统需至少达成以上设备要求，才能够实现系统完整功能。

4.1.2 应用模型测试系统工作负载测试

系统负载测试使用典型的工作负载测试模型，能够模拟常规环境之下云边端容器集群的工作状态，有一定的代表性。系统部署操作在 Master 节点进行，通过 Kubectl 命令行工具查看当前集群状态如图：

```
k8s@ubuntu:/k8s_master$ kubectl get node -o wide
NAME      STATUS   ROLES                  AGE   VERSION   INTERNAL-IP
master    Ready    control-plane,master   30d   v1.20.5   10.134.66.4
node1     Ready    <none>                 30d   v1.20.5   10.134.115.253
node2     Ready    <none>                 30d   v1.20.5   10.134.91.18
```

图 4-1 命令行工具 Kubectl 查看集群状态

按照本文所属部署完成监控系统，启动浏览器之后访问可获得监控界面。界面主题设计以淡蓝色和白色为主，呈现监控信息的部分以图表展现为主。左边旁栏可设置监控数据的对象，页面上方工具栏从左到右分别是系统图标和名称，集群监控报警，刷新监控状态，监控周期设置，登录用户功能图标。

图 4-2 监控系统实例运行界面截图

图 4-2 展示监控系统实验集群监控状态，其中堆叠扇形图显示的是监控的工作节点 Pod 内存使用状态，扇形图显示 CPU 资源占用情况，折线图显示网络 IO 监控状况，系统运行正常并完成监控任务。

对于有限量的监控对象和运行时间，本监控系统有良好的监控性能和稳定性，在相当的监控系统范畴当中，较为轻量，部署简单，有非常大的拓展和开发空间，能够满足系统设计的要求。

第二节 系统的功能拓展性讨论

本节主要对系统构架可延展的方向 32 做理论分析，主要集中在监控方式的可拓展性，分布式计算策略的部署以及可以改进的技术点这三方面。所有论述的成立条件是容器以及容器集群环境，部署监控系统的目的也在于更好地获得容器的状态信息、获得更有价值的容器监控信息，换句话说数据是这个系统的核心。这些信息的特征能够反应当前容器集群的运行状态，反应容器运行的应用状态和执行情况，进而应用于数据指向的各类研究，比如深度学习、人工智能等方向的使用。

4.2.1 监控方式的可拓展性

本文工作实现主要是 Cadvisor 以及自定义的包括时间次数测量在内的指标，可以监控获取包括容器、机器、服务等对象的 CPU、内存、网络数据丢包率等指标数据，能够满足基础的监控需求，适用于容器编排系统的基础元对象，这是能够覆盖的部分。相较而言，由于系统本质依旧是容器，所以完全是可以将其部署于更加复杂的集群网络当中，比如网状结构，多层结构或者是复合结构容器部署，比起单台机子的监控状态，容器集群的整体状态、局部状态、链路状态更具有观测的价值。想要获得这些数据，就必然需要结合多个节点的数据，进行运算和模型匹配；这种情况可以考虑将逻辑相邻的两个节点以及链路统一为一个对象，通过其上两个检测组件的数据传输到一个核心节点的 ProData 处理，对于传输组件来说就相当于抽象出了一个观测的主体，能够实现整体的观测和分布式计算的部署，同时友好的 API 接口也能够让更多的监控工具和手段对接到监控模块当中，使其有着广阔的发展前景。

4.2.2 分布式计算策略的部署

系统模块设计中计算模块包括处理监控数据和响应主节点控制调度的功能。本文工作设置了 Probe 插件基础的计算行为，能够满足基础的监控需求；从机制

上分析，计算模块和监控模块、传输模块之间存在通道量，可以实现双端通信，同时插件是运行在工作节点容器当中的进程，其权限通过 Config 文件目录的挂载实现，理论上可以通过让容器挂载集群控制 Config 文件目录，实现拟 Master 节点化，这样监控组件就能够通过已有的模型数据基准对容器或者是容器集群做控制，实现远端登录的功能。此设想可通过 SSH 技术连接节点，Master 命令行直接控制节点上的 Kubelet 和 Kubectl 组件，完成主节点对工作节点的控制功能。

4.2.3 可改进的技术点

本文系统测试对于一些算法设置部分囿于功能应用并不是专属问题，只做了广泛性质的设计，理论上组件的实现应当和更具体的应用场景相结合，采用更加具体详细的算法策略，能够让组件完成功能性质的转变。主要的可扩展地是监控数据和数据计算部分的内容，ReBalance 的目的是调节监控数据的获取频率和获取策略，现在的策略主要是在尽可能减少损耗的同时保证完全覆盖式的监控。

针对不同的应用场景，比如物联网络通讯情景，监控的主要目的是观测网络流量的情况，而节点的 GPU 使用情况则并不是重点，通过优化观测的方式和内容，设置监控数据对象范围只包括流量相关的数值能够对系统的专用性和性能有较大的提升。针对不同检测对象和内容，比如智慧城市当中的摄像监控系统，关注的是每个节点的存活情况和数据存储情况，但并不很是关心他们之间的网络流量变化和 CPU 使用，那么需要更改监控的策略，更关注捕捉数据的突变和报警，因为长时间的观测可能会消耗太多的资源而并不会在功能上有任何的提升。

参考文献

[1] 顾笛儿, 卢华, 谢人超, 黄韬. 边缘计算开源平台综述[J].网络与信息安全学报,2021,7(02):22-34

[2] Claus Pahl, Brian Lee. Containers and clusters for edge cloud architectures - a technology review[J]. 2015 3rd International Conference on Future Internet of Things and Cloud,2015,10(1109):379-386

[3] Scott Carey, 杨勇. 云计算2018年发展趋势:无服务器计算、Kubernetes和供应商垄断[N]. 计算机世界,2018-01-22(010)

[4] Docker Inc. Docker Docs[DB/OL]. [2020-10-11]. https://docs.docker.com/

[5] Merkel Dirk. Docker: Lightweight Linux Containers for Consistent Development and Deployment[J]. Linux J,2014(239):239-241

[6] RedHat. 什么是Kubernetes (Kube),一文了解K8s是什么[EB/OL].[2020-11-15]. https://www.redhat.com/zh/topics/containers/what-is-Kubernetes

[7] Phoenixnap.com. Understanding Kubernetes Architecture with Diagrams[EB/OL]. (2019-11-12)[2020-10-15]. https://phoenixnap.com/kb/understanding-Kubernetes-architecture-diagrams

[8] Google. Kubernetes[DB/OL]. [2021-3-11]. https://github.com/kubernetes

[9] Alexander Mohr. Kubernetes: Container Orchestration and Micro-Services[EB/OL]. (2016-11-16)[2021-3-21]. https://courses.cs.washington.edu/courses/cse550/16au/notes/Kubernetes.pdf

[10] Neil Brown. Control groups, part 6: A look under the hood [EB/OL]. (2014-8-6)[2021-3-21]. https://lwn.net/Articles/606925/

[11] Solid IT. DB-Engines Ranking of Time Series DBMS[DB/OL]. [2021-3-25]. https://db-engines.com/en/ranking/time+series+dbms

[12] 刘金. 大规模集群状态时序数据采集、存储与分析[D]. 北京邮电大学,2018.

[13] Google. Cadvisor[DB/OL]. [2020-11-20]. https://github.com/google/cadvisor

[14] Victoria Pimentel, Nickerson, Bradford G. Communicating and Displaying Real-Time Data with WebSocket[J]. IEEE Internet Computing,2012,16(4):45-53

[15] 郑强,徐国胜. Websocket在服务器推送中的研究[A]. 中国通信学会.第九届中国通信学会学术年会论文集[C]. 中国通信学会:中国通信学会青年工作委员会, 2012:6

[16] 陈霄,郭志川,孙鹏,朱小勇.基于Web浏览器的远程容器登录系统设计[J]. 网络新媒体技术,2017,6(06):61-65

致谢

光阴荏苒，岁月如梭，四年的本科学习接近尾声。回顾这段时光，感概万千，书桌的学习和工程项目的历练让我积蓄了丰富的知识，锻炼了踏实的工程能力，更让我进一步思考了自己的人生走向和发展路径，结识了诸多老师和朋友，留下了许多难忘的回忆；注定是影响我一生的重要阶段，也是最为珍贵的财富。在此对曾帮助和引导我的老师，同学，亲人，朋友们表示最诚挚的感谢！

首先，我要感谢我的导师徐敬东教授，是老师将一无所知的我带入研究的领域，在我茫然疑惑的时候指明方向，在我焦急困惑的时候耐心解答，在我灰心丧气的时候鼓励引导；感谢徐老师悉心的指导和无私的帮助，让我能够始终保持积极的态度应对学习和研究中的问题和挑战，让我能够在复杂的问题当中寻找到关键的要素，让我能够不断修正和完善自己的工作和研究方式。

感谢张建忠老师和蒲凌君老师在学习和实验上给予的诸多帮助和建议；感谢董前琨老师在硬件上的诸多指导和建议，让我在极端困惑的时候找到了解决办法；感谢薛颖老师的悉心关怀和帮助，让我顺利地解决了学生生活中的诸多问题。

感谢刘松师兄在系统搭建和硬件设施部署方面提供的各项指导和意见，感谢公倩昀师姐，蔺雪莹师姐的细心指导和修改建议，感谢张嘉超师兄，李建斌师兄在工作研究生活当中的诸多帮助。

感谢我的母亲，感谢她的爱护和坚持，让我来到南开，感谢她的理解和支持，让我投入于想要去做的事业，感谢她的教育和付出，让我能够去到更加广阔的舞台。

本文的工作为我的本科学习画上句号，但前路漫漫，在今后的学习研究当中我必定会更加努力，不辜负大家的期望和帮助。

个人简历

达益鑫，男，1998 年 4 月 10 日出生，2017 年 9 月至 2021 年 7 月就读于南开大学物联网工程专业，攻读学士学位。