**密级： 保密期限：**

xm 拷贝

**硕士学位论文**



**题目：智慧管廊系统服务调用中间件的设计与实现**

**学 号： 2017180092**

**姓 名： 李腾飞**

**专 业： 电子与通信工程**

**导 师： 袁超伟**

**学 院： 信息与通信工程学院**

**2020 年 3 月 5 日**

独创性（或创新性）声明

本人声明所呈交的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。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京邮电大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名：日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

关于论文使用授权的说明

学位论文作者完全了解北京邮电大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属北京邮电大学。学校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，允许学位论文被查阅和借阅；学校可以公布学位论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存、汇编学位论文。（保密的学位论文在解密后遵守此规定）

保密论文注释：本学位论文属于保密在年解密后适用本授权书。

非保密论文注释：本学位论文不属于保密范围，适用本授权书。

本人签名：日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

导师签名：日期：\_\_\_\_\_\_\_\_\_\_ \_\_\_\_

**智慧管廊系统服务调用中间件的设计与实现**

**摘 要**

随着物联网技术的快速发展，越来越多智能终端设备大量涌现，如何对接入云平台的智能终端设备进行高效、便捷、低成本的管理成为物联网云平台亟需解决的问题。构建高资源利用率、可弹性伸缩的分布式云平台成为解决此问题的有效方法之一。本文针对智慧管廊系统服务调用中间件的设计与实现进行了研究。

智慧管廊系统运行在基于Kubernetes搭建的三化物联网云平台上，系统中的不同业务能力彼此独立。Kubernetes集群内部可以使用服务名和端口通过HTTP请求实现服务间调用。集群内部服务之间多为小数据量，高并发量请求，协议头较重的HTTP协议会影响服务间的调用效率。Client端在发起请求后，线程会一直阻塞等待Server端的响应结果的返回，造成了服务器资源的浪费。本文的研究目的就是改进分布式云平台集群内部服务调用方式，提高分布式云平台内部服务的调用效率，降低智慧管廊系统服务器资源的占用率,同时为云平台之后的功能升级做兼容。

为了减少智慧管廊系统请求的响应时间,同时减少后台服务资源占用，本文给出了一种服务调用中间件的设计和实现方案。该方案主要从如下几点对智慧管廊系统的服务调用做改进：

⑴ 自定义通信协议。为了满足智慧管廊系统服务间调用请求小数据量的特性需求，通过自定义通信协议解决HTTP协议头较重的问题，减少服务调用数据的网络传输时间，进而减少请求响应时间，同时兼容云平台的流量控制和链路追踪的升级功能。

⑵ Java对象编解码技术。对服务间调用传输的数据进行序列化，减小网络传输数据的大小，减少数据在网络上传输的时间，缩短请求的响应时间。

⑶ 通信框架的设计。使用NIO技术，解决请求线程阻塞导致的服务器资源占用问题。服务调用之间采用长连接，解决Client端请求频繁的TCP连接和释放导致的时间开销的问题，缩短请求的响应时间。

⑷ 多线程技术。使用Java线程池执行异步任务，避免线程创建导致的时间的开销，缩短了请求的响应时间。

⑸ 服务注册与发现。引入Kubernetes的新特性－拓扑感知服务路由，将Client端的请求转发给距离最近的Server端，避免了将请求转发给距离较远的Server端，缩短了请求网路传输的路径，进而缩短了请求的响应时间。在Client端缓存中，将服务名和Cluster IP建立对应关系,避免了请求过程中DNS解析导致的时间开销，缩短请求的响应时间。

关键词: 分布式云平台 kubernetes集群 服务调用 智慧管廊

DESIGN AND IMPLEMENTATION OF SERVICE CALL MIDDLEWARE FOR SMART PIPE GALLERY SYSTEM

ABSTRACT

With the rapid development of the Internet of Things technology, more and more intelligent terminal devices have emerged in large numbers. How to efficiently, conveniently and low-costly manage the intelligent terminal devices connected to the cloud platform has become an urgent problem for the IoT cloud platform. Constructing a high-resource utilization, elastic and scalable distributed cloud platform has become one of the effective methods to solve this problem. This paper researches the design and implementation of the service call middleware of the smart corridor system.

The Smart Corridor system runs on the Tri-Internet of Things cloud platform built on Kubernetes, and the different business capabilities in the system are independent of each other. Kubernetes clusters can use service names and ports to implement inter-service calls through HTTP requests. The services within the cluster are mostly small data and high concurrent requests. The HTTP protocol with a heavier protocol header will affect the call efficiency between services. After the client sends a request, the thread will always block waiting for the response of the server to return, causing a waste of server resources. The research purpose of this article is to improve the internal service invocation method of the distributed cloud platform cluster, improve the efficiency of internal service invocation of the distributed cloud platform, reduce the server resource occupation rate of the smart corridor system, and at the same time be compatible with the functional upgrade of the cloud platform.

In order to reduce the response time of the smart pipe gallery system request and reduce the background service resource occupation, this paper gives a design and implementation scheme of service call middleware. This solution mainly improves the service call of the smart corridor system from the following points:

⑴Custom communication protocol. In order to meet the characteristics of the small amount of data required for invocation requests between smart pipe gallery systems, the problem of heavier HTTP protocol headers is solved through a custom communication protocol, which reduces the network transmission time of service invocation data, thereby reducing request response time, and is compatible with cloud platforms Upgrade capabilities for flow control and link tracking.

⑵ Java object codec technology. Serialize the data transferred between services, reduce the size of the data transmitted on the network, reduce the time for data to be transmitted on the network, and shorten the response time of the request.

⑶ Design of communication framework. Use NIO technology to solve the problem of server resource occupation caused by request thread blocking. Long connections are used between service calls, which solves the problem of time overhead caused by frequent TCP connection and release requests from clients, and shortens the response time of requests.

⑷ Multi-threaded technology. Use the Java thread pool to perform asynchronous tasks, avoid the cost of time caused by thread creation, and shorten the response time of requests.

⑸ Service registration and discovery. Introduced a new feature of Kubernetes, topology-aware service routing, which forwards client requests to the nearest server, avoiding forwarding requests to farther servers, shortening the path for requesting network transmission, and further reducing Response time. The Cluster IP corresponding to the service name is cached on the client side to avoid the time overhead caused by DNS resolution during the request and shorten the response time of the request.

KEY WORDS: distributed cloud platform kubernetes cluster service call smart pipe gallery

**目 录**

[第一章 绪论 1](#_Toc34397092)

[1.1 课题研究的背景与意义 1](#_Toc34397093)

[1.2 分布式平台发展的现状 2](#_Toc34397094)

[1.3 本文的主要研究内容 3](#_Toc34397095)

[1.4 论文的结构 5](#_Toc34397096)

[第二章 服务调用中间件相关的技术 6](#_Toc34397097)

[2.1 HTTP协议 6](#_Toc34397098)

[2.1.1 协议的简介 6](#_Toc34397099)

[2.1.2 HTTP工作原理 7](#_Toc34397100)

[2.2 Java对象的编解码技术 7](#_Toc34397101)

[2.2.1 定义 7](#_Toc34397102)

[2.2.2 Java序列化的缺点 8](#_Toc34397103)

[2.2.3 Protobuf编解码框架 8](#_Toc34397104)

[2.3 网络通信框架 8](#_Toc34397105)

[2.3.1 Linux网络I/O模型简介 8](#_Toc34397106)

[2.3.2 Java NIO编程 9](#_Toc34397107)

[2.3.3 Netty通信框架 9](#_Toc34397108)

[2.4 服务注册和发现 11](#_Toc34397109)

[2.4.1 Kubernetes中的Service(服务) 11](#_Toc34397110)

[2.4.2 服务注册中心 12](#_Toc34397111)

[2.4.3 拓扑感知服务路由 12](#_Toc34397112)

[2.6 本章小结 13](#_Toc34397113)

[第三章 智慧管廊系统服务调用中间件的需求分析和架构设计 14](#_Toc34397114)

[3.1 智慧管廊系统的分层 14](#_Toc34397115)

[3.2 智慧管廊系统减少请求反应时间的需求分析 15](#_Toc34397116)

[3.2.1 自定义通信协议的需求分析 15](#_Toc34397117)

[3.2.1.1 云平台服务间调用的特点 15](#_Toc34397118)

[3.2.1.2 HTTP协议的优缺点 15](#_Toc34397119)

[3.2.2 Java的编解码技术的需求分析 16](#_Toc34397120)

[3.2.2.1 编解码技术的需求 16](#_Toc34397121)

[3.2.2.2 编解码框架的比较 17](#_Toc34397122)

[3.2.3 服务注册和发现的需求分析 18](#_Toc34397124)

[3.3 智慧管廊系统减少服务器资源占用的需求分析 18](#_Toc34397125)

[3.3.1 通信框架设计和多线程技术的需求分析 19](#_Toc34397126)

[3.3.1.1 通信框架客户端的需求分析 19](#_Toc34397127)

[3.3.1.2 通信框架服务端的需求分析 19](#_Toc34397128)

[3.3.1.3 长连接数据传输通道的需求分析 20](#_Toc34397129)

[3.2.1.4 NIO的需求分析 20](#_Toc34397130)

[3.2.1.5 多线程技术的需求分析 22](#_Toc34397131)

[3.4 智慧管廊系统服务调用中间件架构设计 22](#_Toc34397132)

[3.4.1 服务间调用通信架构的设计 23](#_Toc34397133)

[3.4.1.1 事件处理链事件 23](#_Toc34397134)

[3.4.1.2 网络I/O数据的传输 24](#_Toc34397135)

[3.4.2 多线程架构的设计 25](#_Toc34397137)

[3.4.3 kubernetes集群服务设计 26](#_Toc34397138)

[3.5 本章小结 27](#_Toc34397139)

[第四章 智慧管廊系统服务调用中间件的实现 28](#_Toc34397140)

[4.1自定义通信协议的实现 28](#_Toc34397141)

[4.2 Java对象编解码的实现 29](#_Toc34397142)

[4.2.1 Java对象的proto文件 29](#_Toc34397143)

[4.3 网络通信框架和多线程并发的实现 31](#_Toc34397144)

[4.3.1网络通信框架的实现 31](#_Toc34397145)

[4.3.1.1 长连接的实现 31](#_Toc34397146)

[4.3.1.2 NIO的实现 33](#_Toc34397147)

[4.3.2 多线程控制的实现 33](#_Toc34397148)

[4.3.2.1 线程池和Task任务的实现 34](#_Toc34397149)

[4.3.2.2 Server端多线程控制的实现 37](#_Toc34397150)

[4.3.2.3 Client端多线程控制的实现 39](#_Toc34397151)

[4.4 服务注册和发现的实现 39](#_Toc34397152)

[4.5 本章小结 41](#_Toc34397153)

[第五章 测试与分析 42](#_Toc34397154)

[5.1 服务器资源占用优化的测试与分析 42](#_Toc34397155)

[5.2 减少响应时间的的测试与分析 43](#_Toc34397156)

[5.2.1 长连接的测试与分析 43](#_Toc34397157)

[5.2.2 Service优化转发功能的实现 45](#_Toc34397161)

[5.3 本章小结 47](#_Toc34397168)

[第六章 总结与展望 48](#_Toc34397169)

[6.1 总结 48](#_Toc34397170)

[6.2 展望 49](#_Toc34397171)

[致谢 50](#_Toc34397172)

[参考文献 51](#_Toc34397173)

第一章 绪论

1.1 课题研究背景与意义

物联网这个术语是凯文.艾什顿(kevin Ashton) 于1999年提出的，它的发展受益于计算机技术和集成技术。在1999年之前,虽然还没有物联网这个概念，但是很多产品已具备了一些物联网的典型特征，只不过受限于集成技术，要么是在形态上特别巨大、笨拙，要么就是缺乏高效的实时传输能力和灵活的控制手段，比如 1993年，Quenti Stafford-Fraser和Paul Jardetzky发明了Trojan Roo coffee pot,这家咖啡店装了视频摄像机和传感器，用来识别咖啡机的咖啡量，当咖啡量低于某个量时会立即进行自动装填。

现在物联网技术已经发展的很成熟了，它不仅仅是指我们能接触到的智能设备，例如门把手，它还包括后面的应用逻辑和一系列功能模块，它的架构如下图1-1所示:

图片包含 截图, 标志, 街道, 蓝色

描述已自动生成

图1-1 物联网平台的分层

最底层是智能设备层，智能设备是物联网的触手，采集真实世界中各种类型的数据，并通过物联网网关将数据传输到设备管理软件层。

设备管理软件层所使用的数据存放在数据存储层，这些数据也是API(应用程序接口)管理层的数据来源。API管理层应用实现了应用与数据的解耦，为上层应用提供数据服务，同时上层应用也可以通过API管理层对智能设备进行反向控制。人工智能、规则管理等其实也不是直接使用存储层的数据，而是通过API管理层调用数据，以达到数据的完整一致性。能够加进物联网的人工智能目前也越来越多，包括机器学习、大数据分析、预测算法等。

上述介绍基本涵盖了物联网平台应用的总体架构，可能会在不同的领域和场景有不同的变化。比如以下场景:

⑴ 医疗领域和物联网

医疗资源在社会中是一种稀缺的资源，在供需关系中处于一种极不平衡状态，物联网技术的引入，可以极大的缓解这种供需关系，用于诸如远程看护、慢病监测、远程诊断等。

⑵ 工业领域和物联网

工业领域物联网应用，也称为IoT，它是物联网技术的最早应用，也是最深远的领域之一，物联网技术是工业4.0的主要驱动技术。自动制造和生产预测两种基于物联网技术成熟的应用。自动制造是生产环节之中通过对设备的物联网改造，实现部分或全部环节的自动化。

⑶ 智慧城市和物联网

物联网技术是智慧城市的支撑技术，从IPC(进程间通信)到现在的云存储，都是物联网技术在不同规模层次的应用，而且现在的城市安防工程，也从之前单一的安防系统扩展到门禁系统、环境监测系统、智能灯杆系统等产品领域，接入的终端数量也步入百万级。智慧管廊系统也是智慧城市的重要管理子系统之一。

1.2 分布式平台的发展现状

随着面向对象和设计模式的发展，依据职责划分产生了经典的MVC(模型、视图、控制器)框架，MVC模式最早在1979年由Trygve Reenshaug提出，其中模型聚焦数据访问及存储，视图聚焦数据显示和交互，控制器聚焦业务处理[1]。每一层都有具体的职责和分工，这降低了层与层之间的耦合，这便是软件架构的经典模式，又被称为三层架构。三层架构将系统进行了逻辑分层，但是在物理上最终会运行在同一进程，所以三层架构又称为单块架构。

随着业务的不断发展，功能的持续增加，三层逻辑架构设计中的单块架构已很难满足业务快速变化的要求:一方面，代码的可维护性、可拓展性和灵活性在不断降低；而另一方面，系统的测试成本、构建成本和维护成本又在显著增加。随着项目的不断发展和产品规模的不断扩大，单块架构应用的改造甚至重构势在必行。为解决这些问题，SOA(面向服务的架构)出现了，顾名思义它有两个核心: 一是服务，二是架构。SOA架构中包含多个服务，服务之间通过相互配合完成一系列功能。一个服务通常以独特的形式存在于操作系统，服务和服务之间不再通过进程内调用方式进行通信，而是改为网络调用，甚至当服务需要方Client和服务提供方Server都部署在同一机器时，也是通过网络调用完成请求和响应。SOA相对于传统单块架构的优势和特点是服务化、松耦合、灵活的架构等。

SOA最早由Gartner在1996年提出，2002年12月，Gartnar再次提出SOA是“现代应用开发领域中最重要的课题” [2]。不过虽然面向服务的SOA架构理论和实践方面已经研究了十多年[3]，但是由于给SOA下定义的组织和机构太多，没有一个统一的标准，导致在很长一段时间内人们对于SOA都存在着不同的理解。所以尽管大家做了很多努力和尝试，但是对如何定义SOA仍然存在着不同的认知和理解。除了不同角色的参与人员很难把它作为一个整体进行统筹外，在实施SOA过程中还会有各种各样的问题: 比如通信机制和协议的制定、服务粒度的确定和边界的划分，各种中间件的选择等。如今现实问题是现有的SOA解决方案并不能有效的防止服务之间的过度耦合而且很难把应用程序的大小控制在合理的范围内，所以不可避免会遇到和单块架构发展到一定规模遇到的问题相同。

随着领域驱动设计、按需虚拟化、持续交付、小型自治团队、基础设施自动化和大型集群系统等技术和实践的发展以及为解决传统架构的痛点，微服务应运而生。微服务是一种分布式的系统解决方案，着力推动细粒度的服务的使用，不同的微服务协同工作，每个服务都有自己的生命周期。由于微服务主要围绕业务领域构建模型，所以可以避免由传统的分层架构引发的很多问题。微服务整合了过去近十年来的新概念和新技术，因此得以避开许多面向服务的架构中的陷阱[4]。

2011年5月，在威尼斯附近的软件架构师小组首次提及了”Microservice”一词，以描述参与者中的许多人近期探索研究的许多架构风格。2012年3月，来自ThroghtWorks的James Lewis在克拉科夫33rd Degree会议上就此做了相关的案例研究报告，来自Netflix的Adrian Cockcroft把这种方法称为“细粒度SOA”,并认为这是一套在Web规模下具有开创意义的架构类型[5]。2012年5月，威尼斯的软件架构师小组最终决议，以“microservice”作为最终的架构名称。2014年，Martin Fowler和James Lewis共同提出微服务的概念，将微服务的定义概括如下:简而言之，微服务架构是将单个应用程序开发为一组小型服务的方法，每个应用程序运行在自己的进程中，并通过轻量级的通信机制进行通信，通常是基于HTTP(超文本传输协议)资源的API[6]。这些服务是围绕业务功能构建的，并可以通过自动部署机制独立部署[7]。这些服务应该尽可能少的采用集中式管理，并根据不同需求，使用不同的编程语言和数据存储[8]。

1.3 本文的主要研究内容

SOA架构体系中采用较多且较为普遍的是基于HTTP协议的服务调用，但是过多的Header信息使得HTTP协议显得比较笨重，不再适用于智慧管廊系统小数据量的服务调用场景。HTTP请求建立的TCP(传输控制协议)连接是短连接，Client端每次请求Server端都需要创建和释放TCP连接。智慧管廊系统中服务间调用的Client使用的Okhttp3,在请求发送后阻塞等待Server响应结果的返回。针对智慧管廊系统在使用HTTP协议进行服务调用遇到的问题，本文智慧管廊系统服务调用中间件的设计与实现主要通过优化服务间调用，解决大规模物联网设备数据上传到智慧管廊系统后，业务应用处理速度慢，出现超时的问题和服务器资源占用过多的问题。拟解决的关键问题如下:

(1)自定义通信协议

智慧管廊系统服务间调用的特点是大部分请求是小数据量，高并发量。HTTP协议Header信息过多导致网络传输数据的冗余，因此重新设计简单应用层通信协议，简化协议头，使传输的数据内容的比重上升，提升通信效率。

(2) Java的对象编解码技术

自定义协议的消息对象需要序列化为二进制数据才能在网络上传输，在选择序列化算法需要考虑编解码的耗时和对象编码的二进制数据大小。

(3)网络通信框架的设计

使用NIO(New I/O)技术，重新设计Client端和Server端的数据交互的过程，解决Client端请求线程阻塞等待导致的服务器资源占用问题。在Client端和Server端建立长连接解决Client端每次请求都需要建立和释放TCP连接导致的时间开销和服务器资源占用问题。

(4)多线程并发控制

为了应对服务间调用小数据量、高并发量的特点，本文将自主设计来实现多线程的控制。设计多线程模型的时候，需要考虑Server端和Client端的不同应用场景。Server端的需求主要是避免在接收到请求之后创建线程处理任务时导致的时间开销。Client端的需求是根据Server端响应的结果异步调用响应结果的处理过程。根据这两个不同的应用场景设计多线程架构。

(5)服务注册和发现

在Kubernetes创建Service资源时会把Server的信息写入分布式数据库etcd，同时Kubernetes 为每个Service 分配全局唯一的Cluster IP ,做服务发现只需要把Service的Cluster IP和Service的Name 做一个DNS映射即可。在kubernetes集群内部提供DNS组件做域名解析，进一步简化了服务发现的实现。

引入kubernetes的新特性–拓扑感知服务路由，解决等概率的负载转发出现将请求转发给距离较远的endpoint的问题，缩短服务间调用数据在网络上的传输路径，减小请求的响应时间。

1.4 论文的结构

本文共分为六个章节，以下为本文章节的大致内容:

第一章: 绪论。归纳总结了联网云平台的发展趋势，阐述了智慧管廊系统产生的背景，指出了现有的智慧管廊系统服务间调用存在的不足，提出了提升系统整体性能和优化服务器资源占用的解决方案。

第二章: 服务调用中间件相关技术 。 对本文在服务调用中间件涉及到的相关技术和理论进行了简要介绍。

第三章: 智慧管廊系统服务调用中间件的需求分析和架构设计。从智慧管廊业务需求角度出发，给出了服务调用中间设计的需求分析，给出服务调用中间件的架构设计。

第四章: 服务调用中间件的实现。给出了服务调用中间件具体功能模块的设计和实现，如自定义通信协议，Java对象编解码技术等。

第五章：测试与分析。本文针对智慧管廊系统服务间调用进行相关指标的测试，并对测试结果进行分析。

第六章: 总结与展望。对本文的研究工作做出总结，并结合现在云平台发展的现状指出待完善的工作内容。

第二章 服务调用中间件相关的技术

服务调用中间件主要完成的功能是智慧管廊系统业务彼此独立的微服务模块之间的数据交互，此过程涉及到通信协议、网络IO、负载均衡策略等分布式云平台中用到的技术，本章节主要给出服务调用中间件涉及的理论知识，基于这些技术理论去设计中间件。

2.1 HTTP协议

2.1.1 协议的简介

HTTP协议是一种用于分布式、协作式和超媒体信息系统的应用层协议。HTTP是万维网的数据通信基础。

HTTP协议包括请求协议和响应协议，具体内容如下表所示

表格 2-1 HTTP请求格式

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 请求行 | 请求方式 | 空格 | URL | 回车符 | 换行符 |
| 请求头部 | 头部字段名 | ： | 值 | 回车符 | 换行符 |
| ··· | | | | |
| 头部字段名 | ： | 值 | 回车符 | 换行符 |
|  | 回车符 | 换行符 |  |  |  |
| 请求数据 |  | | | | |

表格 2-2 HTTP响应格式

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 状态行 | 协议版本 | 空格 | 状态码 | 空格 | 状态码描述 | 回车 | 换行符 |
| 响应头部 | 头部字段名 | ： | 值 | 回车符 | 换行符 | | |
| … | | | | | | |
| 头部字段名 | ： | 值 | 回车符 | 换行符 | | |
| 回车 | | | | 换行符 | | | |
| 响应正文 | | | | | | | |

表格 2-3 HTTP状态码

|  |  |  |
| --- | --- | --- |
| 状态码 | 类别 | 原因短语 |
| 1XX | Information | Server端接收到Client端的请求，需要Client端做进一步的处理 |
| 2XX | Success | Server端接收到Client端的请求，返回处理的结果 |
| 3XX | Redirection | 需要Client端重定向请求 |
| 4XX | Client Error | Client端的请求信息有误 |
| 5XX | Server Error | Server端处理Client端的请求消息出错 |

2.1.2 HTTP工作原理

HTTP协议定义Client端如何从Server端请求信息，以及Server端如何把处理后的响应结果返回到Client端。HTTP协议采用了请求/响应模型。以下是 HTTP 请求/响应的步骤：

⑴ Client端连接到Server端。Client通过TCP协议的三次握手与Server端建立TCP的连接。

⑵ Client端发送HTTP请求报文。通过socket套接字，Client端向Server端发送HTTP协议请求信息。

⑶Server端接受请求处理后将处理结果封装成HTTP响应报文返回给Client端。Server端解析请求，通过dispatch转发器将请求分发到Server端的业务Controller处理。Server端将处理结果封装到HTTP响应报文，然后写入到TCP套接字。

⑷ TCP连接的释放

Client发送的请求信息的Header中的Connection 的值为close，则Server端主动关闭TCP连接，通过TCP的四次挥手，释放TCP连接;若Header中的Connection的值为Keepalive，Server端则会保持这个连接，等待Client端再此发送请求信息。

⑸ Client端处理Server端返回的处理结果。

2.2 Java对象的编解码技术

2.2.1 定义

Java序列化的目的主要是网络传输和对象持久化。服务消费者调用服务提供者需要把传输的Java对象编码为字节数组或者ByteBuffer对象。服务提供者在收到Bytebuffer对象或者字节数组后，将其解码为服务消费者发送的Java对象。Java对象和字节数组之间的转化用到的技术就是Java对象的编解码技术。

JDK提供了Java对象的编解码技术，在通用性和性能上无法满足服务间调用的要求，因此在业界出现了很多编解码技术。

2.2.2 Java序列化的缺点

Java序列化从JDK1.1版本就已经提供，它不需要添加额外的类库，只需要实现java.io.Serializable并生成序列ID即可，但是在远程服务调用(RPC)时，很少直接使用Java序列化进行消息的编解码和传输，原因有以下几点:

⑴ 无法跨越语言。JDK提供的编解码技术使用的是私有协议，其它编程语言无法解码编码的二进制数据。

⑵ 序列化的码流太大。采用JDK序列化机制编码后的二进制数组是二进制编码的5.29倍[10]。

⑶ 序列化的性能太低。Java序列化只有二进制编码的6.17%左右[10]。

2.2.3 Protobuf编解码框架

Protobuf全称Google Proto Buffers,它由谷歌开源而来。Protobuf使用代码生成工具根据proto文件定义的数据结构生成对POJO对象和Protobuf相关的属性和方法。它的特点如下：

⑴ 使用文件定义数据结构。

⑵ 编解码耗时短，编码的二进制数据量小。

⑶ 支持异构平台之间的数据传输。

Protobuf使用二进制编码，在空间和性能上具有更大的优势。Protobuf的数据描述文件和代码生成机制，使其具备了以下的优点：

⑴ 支持异构平台的数据通信。

⑵ 兼容不同的协议版本。

⑶ 使用代码生成工具生成代码，减小开发的工作量。

⑷ 结构化的文档相对于代码而言更容易维护。

2.3 网络通信框架

2.3.1 Linux网络I/O模型简介

Linux内核把网卡设备作为一个文件，调用系统指令对socket进行读写操作会返回一个socketfd(socket文件描述符)，描述符指向内核中的一个结构体(文件路径，数据区等一些属性)。本文设计网络通信框架主要涉及到以下两种网络I/O模型：

⑴ 阻塞I/O模型: 在用户空间进程中调用recefrom发起系统调用，Linux内核把网卡接收到的数据复制到用户空间进程的缓存中或者出错才会返回结果，在结果返回之前用户空间的进程一直处于阻塞状态。

⑵ I/O复用模型： Linux 提供顺序扫描和事件驱动两种方式实现网络I/O的复用，分别对应seletc/poll和epoll。在顺序扫描的实现中内核会轮询注册的多个fd(文件描述符)的状态，检测fd处于就绪状态后调用回调函数rollback,考虑到轮询的性能问题，顺序扫描的模式支持的fd数量有限 。在事件驱动实现中，当有fd就绪时，立即调用回调函数rollback，因此性能更高。

2.3.2 Java NIO编程

在JDK1.4版本之前，Java的网络编程采用的同步阻塞的模式，Client端发送请求与Server端建立一个TCP连接，在Server端会启动一个线程去监听Socket的状态，进行网络I/O的读写。在JDK1.4时，新增了个java.nio的包，提供了很多进行异步I/O开发的API和类库，主要的类和接口如下：

⑴ 进行异步I/O操作的缓存区ByteBuffer等。

⑵ 进行异步操作的管道Pipe。

⑶ 进行各种I/O操作(异步或者同步)的Channel,包括ServerSocketChannel和SocketChannel。

⑷ 多种字符集的编码能力和解码能力。

⑸ 实现非阻塞I/O操作的多路复用器。

在JDK1.4的NIO类库中引入Buffer对象，替换之前进行网络I/O的读写的Stream对象。Channel是一个全双工的通道，可以同时读写，而流Stream是单向，读写分别由InputStream和OutputStream的子类实现。多路复用器Selector会不断地轮询注册在其上的Channel，在Channel处于就绪状态时会启动一个线程去处理I/O的读写事件。由于JDK使用了实现的方式是epoll()，因此它没有线程开启的最大连接句柄数的限制，可以实现多个Client端的同时接入。

2.3.3 Netty通信框架

Java NIO类库和API繁杂，同时NIO编程涉及到Reactor多线程模型，基于Java NIO 开发要求开发人员必须对网络编程和多线程非常熟悉。Netty框架内部对Java NIO的API进行了封装，同时也实现了网络编程涉及到的多线程需求，降低了Java NIO 网络开发的门槛。下面介绍一下服务调用中间件用到的Netty的核心组件：

⑴ ByteBuf: ByteBuffer的标识位置的指针只有一个，同时读写操作时容易出错，为了解决这个问题ByteBuf引入了两个指针分别负责读和写，简化了使用者的操作难度。ByteBuffer 需要使用者实现对byte数组的扩缩容，ByteBuf将数组扩缩容的操作进行了封装，在使用者层面上实现了自动扩缩容的，方便了使用者的使用。同时ByteBuf还实现了对内存的回收策略，对于使用者来说，不需要关心内存池的实现细节，也不需要与Chunk和Page相关的类库打交道，只需要按照API说明正常使用即可。在Netty中也提供了ByteBuf相关的辅助类，比如CompositeByteBuf，可以将多个ByteBuf的实例组装到一起，形成统一的视图。

⑵Channel: 与Buffer不同，Channel主要是操作网络I/O,不像Buffer是直接操作内存的，Netty中Channel的实现采取的策略是聚合JDK Channel,通过Façade模式对其进行封装，减少自身的代码量，降低实现的成本。为了和Netty框架中的其它组件的融合，添加了获取Channel的EventLoop、缓存分配器ByteBufferAllocator和pipeline等功能。

⑶ChannelPipeline和ChannelHandler: Netty的ChannelPipeline和ChannelHandler的设计使用了责任链模式的设计模式。Netty将Channel的数据管道抽象为ChannelPipeline,消息在ChannelPipeline中传递。使用ServerBootstrap或者Bootstrap启动Server端或者Client端时，Netty会为每个Channel创建一个独立的pipeline,开发者需要实现ChannelPipeline的Factory工厂类，将ChannelHandler注册到ChannelPipeline上即可。Netty的底层的I/O模型采用的是异步I/O,在异步I/O中内核会告知I/O操作的完成情况,即注册完成情况事件的回调函数，同时相对于Stream而言，Channel是双向流通的，因此Channelhandler也会处理用户主动发起的网络I/O操作事件。在Netty中注册事件回调函数是通过继承父类，重写响应的事件处理方法实现的，比如channelActive(ChannelHandlerContext ctx)方法处理TCP链路建立成功后Channel激活事件。

⑷ EventLoop和EventLoopGroup: EventLoop和EventLoopGroup是Netty网络编程多线程的实现，EventLoop主要操作线程的读写，EventLoopGroup是由一组EventLoop组成的，类似于线程池。在Server端会创建两个EventLoopGroup,用于逻辑隔离NIO Accept和NIO I/O线程。ChannelPipeline的创建就是在NIO I/O对应的EventLoopGroup完成的。在Client端会只创建一个EventLoopGroup,这个是由使用场景决定的，在Server端会处理多个Client端的连接，而在Client端只能和一个服务端连接，没有必要像Server端一样将Accept和I/O读写做逻辑隔离。在SelectionKey轮询到读或者连接操作时，则会调用Unsafe的read方法，Unsafe是一个内部接口聚合在Channel中协助进行网络读写相关的操作，此处Unsafe的实现是个多态，对于Server端的NioServerSocketChannel,它的实现是直接接收客户端的TCP连接，将I/O的读写直接放到处理NIO I/O对应的EventLoopGroup中，但是对于Client端的NioSocketChannel,它的操作是从SocketChannl中读取ByteBuf。Client端获取Server端的响应结果是异步的，不需要Client的请求线程一直阻塞等待Server端的响应结果。

⑸ Future: Future是对JDK的java.util.concurrent.Future的封装,用于接收异步执行的结果。在I/O调用后会立即返回，调用者通过Future获取异步操作的结果。需要注意的是异步I/O操作有两类超时：一个是TCP层面的I/O超时，另一个是业务逻辑层面的超时。两者没有必然的联系，但是ChannelFuture超时并不代表I/O超时，这意味着ChannelFuture超时后，如果没有关闭连接资源，随后连接依旧可能成功，这会导致严重的问题，因此通常情况下业务逻辑超时时间应该大于I/O超时时间，它们两者是包含关系。

2.4 服务注册和发现

2.4.1 Kubernetes中的Service(服务)

Service是kubernetes里的最核心的资源对象之一，kubernetes里的每个Service其实就是我们经常提起的微服务框架中的一个“微服务” [11]。在Service的yaml文件中定义端口和Label Selector，Label Selector 会筛选 相同Label的pod作为Service转发的后端实例。集群内部的应用通过Service 的Name 和 Port就可以访问Service通过Label Selector筛选出来的Pod实例[12]。

通过分析、识别并建模型系统中的所有服务为微服务-Kubernetes Service,将系统切分为多个提供不同业务能力而又彼此独立的微服务单元，服务之间通过TCP/IP进行通信，从而形成强大而又灵活的弹性网格，拥有分布式能力、弹性拓展能力、容错能力，程序框架也会变得简单和直观[11]。

Service将请求转发到Pod是通过Endpoint(PodIP + ContiannerPort )实现的，Kubernetes会为每个Pod分配一个IP，Pod IP在Pod重启后会发生变化。运行在每个Node上的 kube-proxy进程通过写入 iptables 或 ipvs 规则提供负载均衡，负责把客户端的请求转发到Service后端的pod。Service不是共用一个负载均衡器的IP地址，而是每个Service分配一个全局唯一的虚拟IP地址，即Cluster IP。每个服务变成了具备唯一IP地址的“通信节点”，Service的调用就变成了最基本的TCP网络通信。

Kubernetes为Service分配的Cluster IP是固定不变的，因此可以在服务发现用Servcie的Name 和 Service 的Cluster IP 做一个DNS域名映射即可。

2.4.2 服务注册中心

作为微服务架构最基础也是最重要的组件之一，服务注册中心本质上是为了解耦服务提供者和服务消费者。纵观当下各种主流的服务注册中心解决方案，大致可归为三类：

⑴ 独立的loadbalance：服务A调用服务B，是通过调用服务B的负载均衡器实现的，服务B负载均衡器使用负载均衡策略将请求转发给服务B的服务实体。

⑵ 进程内的loadbalance：服务消费者启动一个线程，在启动的线程内获取服务注册中心的注册的服务信息，根据负载策略选择服务实例进行调用。

⑶ 主机独立的loadbalance: 在服务器上运行agent，agent完成服务的发现，服务A调用服务B是通过agent实现的，不是直接调用服务B。

使用Kubernetes的Service和DNS插件可以实现独立的loadblance，kubernetes 提供了 service 的概念，可以通过 Service的name 访问 pod 提供的服务。应用能够直接使用服务的名字，不需要关心它实际的 ip 地址，中间的转换能够通过DNS插件自动完成。智慧管廊系统的模块采用restful架构，服务模块间的调用在请求的url中写入的是Service的Name。

服务网格(Service Mesh)框架Istio是通过在pod中注入sidecar实现主机独立的loadbalance,Pod的底层运行的是容器，容器隔离了底层的服务器资源，因此可以看成是一台独立的服务器，但是Istio并没有改善服务间调用的效率。

Dubbo的服务注册和发现是通过第三方中间件(通常会使用zookeeper)实现进程内的loadbalance，服务间的调用采用的是自定义协议－dubbo协议。Dubbo是比较成熟的rpc调用框架，但是在kubernetes已经集成了基于etcd的服务注册中心，同时服务的调用如果仍通在集群内搭建的zookeeper服务注册中心会引入不必要的冗余，相比较而言，在集群内部基于kubernetes集群内部的服务名和端口的调用更节省服务器的资源。

2.4.3 拓扑感知服务路由

Kubernetes 集群中的Service 对应的 endpoints 分布在不同节点，传统转发策略会对所有 endpoint 做负载均衡，通常会等概率转发，当访问 Service 时，流量就可能被分散打到这些不同的地方。虽然 service 转发做了负载均衡，但如果 endpoint 距离比较远，流量转发过去网络时延就相对比较高，会影响网络性能，在某些情况下甚至还可能会付出额外的流量费用。

Service 转发主要是 node 上的 kube-proxy 进程通过 watch apiserver 获取 Service 对应的 endpoints，再写入 iptables 或 ipvs 规则来实现的; 对于 Headless Service，主要是通过 kube-dns 或 coredns 动态解析得到不同 endpoint ip 来实现的。实现 service 就近转发的关键点就在于如何将流量转发到跟当前节点在同一拓扑域的 endpoint 上，也就是会进行一次 endpoint 筛选，选出一部分符合当前节点拓扑域的 endpoint 进行转发。

拓扑感知服务路由特性会通过 Endpoint Slice 获取这些拓扑信息实现 endpoint 筛选 (过滤出在同一拓扑域的 endpoint)，然后再转换为 iptables 或 ipvs 规则写入节点以实现拓扑感知的路由转发。

每个节点上转发 Service 的 iptables/ipvs 规则基本是一样的，但启用了拓扑感知服务路由特性之后，每个节点上的转发规则就可能不一样了，因为不同节点的拓扑信息不一样，导致过滤出的 endpoint 就不一样，也正是因为这样，Service 转发变得不再等概率，灵活的就近转发才得以实现。

当前还不支持 Headless Service 的拓扑路由，由于 Headless Service 不是通过 kube-proxy 生成转发规则，而是通过 DNS 动态解析实现的，所以需要修改 kube-dns/coredns的代码 来支持这个特性。

2.6 本章小结

本章首先对HTTP协议进行了较为详细的介绍，协议内容、Client端和Server端连接建立和释放的过程，本文对减少请求响应时间做的优化主要是解决HTTP协议在服务间调用场景下的暴漏出来的问题。HTTP协议是网络通信协议，要解决HTTP协议遇到的问题，必然要涉及网络编程相关的技术，因此对网络框架也做了比较详细的介绍，包括网络通信涉及到的编解码技术、I/O操作和多线程控制技术，同时这些也是编程需要关键处理的地方所在。同时也介绍了Java NIO 开发使用的通信框Netty的主要组件。最后又介绍了Kubernetes与微服务相关的概念和最新版本的特性对现有云平台功能提升的影响。

第三章 智慧管廊系统服务调用中间件的需求分析和架构设计

大量设备接入智慧管廊系统，必然会占用大量的服务器资源，减少服务器资源的使用，降低云平台运行的成本成了迫切要解决的问题。为了提升云平台客户的用户体验，在企业中推广智慧管廊系统，减少请求的响应时间是关键因素之一。从这两个角度出发，分析智慧管廊系统优化的需求，同时给出服务中间件的架构设计。

3.1 智慧管廊系统的分层

根据图1-1，中间件的范畴内的模块是在运行在Kubernetes搭建的分布式云平台上，设备管理软件和存储层的功能集成到Kubernetes集群的物接入服务模块，API管理模块是kubernetes集群对物接入服务的管理，主要涉及负载均衡其它服务模块的请求、动态扩容物接入服务实例和对重启宕机的服务实例等。具体模块分层如下图所示：

图片包含 文字, 地图, 游戏机

描述已自动生成

图3-1 智慧管廊系统分层

在智慧管廊系统中Saas(软件即服务)服务之间的服务调用是使用的HTTP协议，Saas应用调用物接入服务也是HTTP协议。Saas服务和物接入服务是运行在基于kubernetes搭建的分布式云平台上。

3.2 智慧管廊系统减少请求响应时间的需求分析

本文通过对通信协议、Java的编解码和服务注册和发现这三个方面的优化，来达到减少请求响应时间的目的。以下是具体的分析：

3.2.1 自定义通信协议的需求分析

通信协议定义了智慧管廊的服务间调用与云平台其它组件交互的方式，如sidecar模块，同时也关系到服务间调用的性能、服务的稳定性和服务的可维护性。

3.2.1.1 云平台服务间调用的特点

为了满足智慧管廊系统服务间调用的调用方式的小数据量、高并发量的特性需求，同时兼顾流量控制和链路追踪的云平台升级功能。云平台服务间调用有以下特点:

⑴ 协议请求信息的Header信息尽量小，避免网路传输数据出现冗余，在智慧管廊系统服务间调用多是小数据量的请求，因此采用Header信息少的通信避免HTTP协议通信遇到的问题。

⑵ 在协议头有流量控制的信息，满足滚动升级、安全控制的高级功能。

⑶ 在协议头中有路径信息，便于在服务调用失败之后的问题定位。

⑷ 协议支持序列化，便于网络传输。

在设计通信协议时需要考虑到服务间调用的特点。以下是对HTTP协议的优缺点的分析。

3.2.1.2 HTTP协议的优缺点

缺点: 协议冗余信息较多。HTTP协议不是专用于RPC(远程过程调用技术)，作为数据传输协议会传输大量的无效数据[9]。HTTP协议分为请求和响应两部分，HTTP请求包括状态行、请求头和消息主体，HTTP的响应包括状态行、响应头和响应正文。除了要传输的实际内容的正文外，HTTP协议还存在比较重的头(Header)信息，在服务间调用数据量比较小的场景下，网络传输数据会有大量的冗余。

优点: 易于做流量控制。在类似于istio 这样的Service Mesh(服务网格)框架中基于sidercar的模式做流量控制是比较容易实现，当然也是最早支持的协议，只需要解析HTTP协议的Header的信息即可实现。对系统的滚动升级可以通过控制流量实现，这在保障服务的持续可用性上起到了关键作用。

通过分析HTTP协议优缺点可以发现，精简HTTP协议的Header信息，即可解决服务间调用小数据量请求使用HTTP出现的传输数据冗余的问题，同时在保留HTTP协议中做流量控制的部分兼容之后云平台功能升级。

3.2.2 Java的编解码技术的需求分析

3.2.2.1 编解码技术的需求

为了减少服务调用网络传输数据的大小，使用Java对象的编解码技术将自定义协议的请求和响应消息编码为二进制便于网络传输。Client端将请求参数的对象编码为二进制数据写入socket，通过TCP连接发送到Server端，在Server端在接收到的是二进制数据，获取Client端发送的请求参数对象需要对接收到的二进制数据进行解码，因此在Server端也需要Client请求参数对应的Java文件。在选择编解码技术的时候，应考虑到Server端和Client端参数对象的代码重复问题。综合考虑，对编解码技术的要求有以下几点：

⑴ 是否支持跨语言。

在分布式平台中，会接入大数据分析平台，以及3DGIS平台，这些平台的开发语言并非Java语言，因此为了分布式平台功能拓展需要，选择可以支持异构语言的编解码技术是十分必要的。

⑵ 编码后码流的大小。

减少服务调用数据在网路上传输的时间，服务间调用传输的TCP流量越少越好。现在云平台内部服务之间的调用走的是内网，没有网络宽带的限制，网络传输的延时也非常小，但是在服务调用的数据传输走公网时，就会出现时延问题，比如在边缘计算的场景，边缘端调用云端的服务就需要走公网，边缘计算也是现有云平台拓展的一个方向，因此做好平台的兼容性拓展也是很有必要。

⑶ 编解码的性能。

使用编解码技术可以减小数据传输的码流，但是编解码的过程会导致服务器计算资源的消耗，同时也会带来时间上的消耗。因此需要选择高性能的编解码框架避免服务器计算资源和时间的消耗。

⑷ 类库是否小巧。

代码编译之后可执行文件在服务器运行时会占用服务器的内存和磁盘资源，在云服务器端只需要增加磁盘和内存即可，但是这样会增加平台的运行成本，因此在边缘计算的边缘端场景下就不适用，因为边缘端的资源是有限的，不可以无限的动态扩容。

⑸ 使用者需要手工开发的工作量和难度，API调用是否方便。

智慧管廊系统的服务提供者处理服务消费者请求的业务方法获取到的数据是请求参数对象序列化之后的二进制数据，需要将对象反序列化为参数对象，因此服务提供者需要定义参数对象的Java文件解码二进制数据，选择可以根据配置文件自动生成代码的编解码框架，可以减少开发者的工作量。代码是根据数据描述文件生成的，可以实现前后版本的兼容。在开发难度和API调用是否方便方面，Google的Protobuf相比Facebook Thrift的Tprotocol有很多优势，和Tprotocol一样，Protobuf也是RPC框架的grpc的一部分，但是在独立使用的解耦方便做的比Thrift的TProtocol要好。虽然的TProtocol支持的二进制编解码以及压缩二进制编解码的性能要优于Protobuf的二进制编解码，但是在使用难度上和API调用上并不占优势。

3.2.2.2 编解码框架的比较

手机屏幕截图

描述已自动生成

图3-2 序列化框架的响应时间对比

手机屏幕截图

描述已自动生成

图 3-3 序列化框架的字节数对比

手机屏幕截图

描述已自动生成

图 3-4 Thrift性能测试对比

图 3-4 Thrift性能测试对比图

通过对比发现Thrift的TProtocol性能要优于Protobuf已经其它序列化框架，但是考虑到开发难度和调用API的方便，选择和Tprotocol性能相差不是很多的Protobuf。

3.2.3 服务注册和发现的需求分析

将智慧管廊系统切分为多个提供不同业务能力而又彼此独立的微服务单元，即kubernetes的Service。Service大多都是Cluster IP形式部署的，Kubernetes为service分配的Cluster IP 在Service 的整个生命周期内都不会发生变化，因此服务发现即是对Service 的 name 和 Cluster IP 做DNS映射。在kubernetes 集群中service 的 Cluster IP 是全局唯一的，因此服务之间的通信即是TCP通信，在TCP通信中Client 端向服务端发送请求需要对请求头中的域名做DNS解析，在Kubernetes集群服务调用中也需要把Servic Name 解析为 Cluster IP ,由于在kubernetes 集群中Cluster IP 唯一不变，因此可以将Cluster IP缓存到本地，避免做DNS引起的时间开销。

在智慧管廊系统内部的服务大多是以Cluster IP的模式部署的，在kubernetes的1.17版本中引入的拓扑感知服务路由特性针对Cluster IP 模式下Service 转发的优化。拓扑感知服务路由可以避免将请求转发给距离比较远的Server端，减小网络传输的时延，避免不必要的流量费用开支。

3.3 智慧管廊系统减少服务器资源占用的需求分析

本文主要是通过在通信框架的设计和多线程技术的使用方面，减少服务器资源的占用。

3.3.1 通信框架设计和多线程技术的需求分析

3.3.1.1 通信框架客户端的需求分析

现在的智慧管廊系统服务调用Client端使用的是OkHttpClient实现的，Client端请求Server是同步阻塞的，需要等待Server端返回才能进行下面的处理流程。在Client端同时发起多个请求时会启动大量的线程阻塞等待Server响应结果的返回，这样会消耗大量的服务器内存资源。针对这一点改进的方案是将同步阻塞I/O改成异步I/O。如果直接将OkHttpClient替换为基于异步I/O实现的Netty Client,只是解决了用户业务线程阻塞的问题，不能解决短连接请求频繁建立和释放的问题。Netty实现的异步I/O是在开发者层面上的对I/O读写的异步，在底层使用的是同步非阻塞I/O，需要启动一个线程轮询注册到Selector 上注册的Channel状态,因此同样会带来服务器线程资源的浪费。在OkHttpClient中存在异步回调的方法enqueue(),但是无法解决每次请求都需要建立和释放TCP连接的问题。在HTTP协议中无法实现服务路径相同参数不同的多个请求的TCP连接的复用，因为在Client获取的响应Body是只读的，不支持写入操作，无法将下一次请求的参数传到服务端。通过自定义通信协议，基于Netty通信框架在Client端和Server建立长连接，就可以实现相同服务路径参数不同的多个请求复用同一个TCP连接。现在的智慧管廊系统在微服务拆分方面做的还不够完善，在同一个服务名提供了多个服务，通过TCP连接的复用可以调用服务名下的多个服务。

3.3.1.2 通信框架服务端的需求分析

智慧管廊的服务提供者处理服务消费者的请求也是同步阻塞，究其原因并不是开发使用的SpringBooot技术栈不支持异步I/O，而是当时的开发人员并没有考虑到同步阻塞I/O带来的服务器资源的损耗，因此想要支持异步I/O必然会导致之前代码的重构。代码改造时可以开启SpringBoot技术栈的异步I/O功能然后对服务提供函数进行重构。这样代码的开发量比较小，服务调用Client端和Server端通信依然是HTTP协议，但是HTTP协议并不适用服务间调用高并发，数据量小的场景。引入长连接替代HTTP协议请求的短连接，长连接作为一个数据传输的通道，传输服务间调用的自定义协议数据，这也会导致服务端代码的重构，因为之前的Server端的dispatch分发器不适用于长连接的需求。综上所述:需要在Server端引入异步I/O,同时使用多线程技术对服务端代码进行重构，解决Server端多个线程阻塞导致的服务器资源的消耗。

3.3.1.3 长连接数据传输通道的需求分析

短连接的Client端向Server端发送请求数据只能在连接建立的时候，当Server向Client发送数据之后，Client收到响应的body中I/O操作是只读的不能写入。TCP协议是全双工的协议，支持对同一个连接的读写操作，但是OkHttpClient的实现是基于HTTP协议的一问一答的通信模型，在Client的底层将网络的读写I/O封装为只读的。因此在请求头中设置Connection的属性值为Keepalive，虽然可以保持TCP连接，但是Client端只能向Server端发送一次数据。因此基于OkHttpClient的enqueue()方法，无法实现长连接。在设计长连接时，不仅要维持TCP连接不断开，同时还要把底层的网络读写I/O暴漏到开发者层面。在长连接中可以使用心跳机制维持TCP连接不断开。

在长连接技术的选型上，如果选择已经成熟的长连接技术，比如grpc，低层数据的通信协议使用的是HTTP2协议，在TCP传输的数据必然包括HTTP2协议的信息，这样就会带来TCP链路上传输数据的冗余。将自定义通信协议的内容做为协议正文，使用成熟的长连接技术传输通信数据只是解决了HTTP请求频繁建立TCP的问题，并没有解决使用HTTP协议带来的传输数据冗余的问题。

3.2.1.4 NIO的需求分析

智慧管廊系统的服务消费者调用服务提供者是同步阻塞的，即I/O模型是同步阻塞I/O，造成了服务器线程资源的消耗。解决的方法就是改用非阻塞I/O，采用异步回调的方式。Client端使用的OkHttpClient的enqueue()方法处理响应是异步回调的，但是基于OkHttpClient无法实现长连接，Client请求Server频繁建立和释放TCP连接的问题仍没有解决。

异步I/O技术可以完美解决Client端同步阻塞的问题，在响应返回会调用回调函数处理响应结果，同时不需要启动线程去监听Channel的状态。Netty是基于Reactor模型实现的NIO，需要启动一个线程轮询注册在Selector上的Channel的状态，相比异步I/O,多了一个轮询的同步线程。在2011年发布的JDK1.7就已经支持了异步I/O(AIO),异步I/O的实现是基于Proactor模型，在Proactor 初始化时会直接把处理I/O读写数据的Handler 通过 Asynchronous operation processor 注册到内核，内核的异步I/O操作完成后会回调注册的Handler处理数据，同时Handler也可以注册新的Handler到内核进程，在回调调用Handler时在Handler内部重新注册将Handler注册到内核处理下一次的I/O数据。Handler是直接注册到内核的因此不需要在用户进程空间将Handler与Channel绑定，基于Reactor模型实现的Netty将Handler与Channel绑定是在用户进程空间完成的，因此异步I/O用户空间的线程开销是少于同步非阻塞I/O(NIO)。

通过以上的分析可以发现，相比于同步非阻塞I/O，异步I/O在减少服务器资源占用方面优势更明显。但是在本文中选择选择同步非阻塞I/O的原因有以下几点：

⑴ 异步I/O编程开发难度大，I/O处理性能受开发者开发水平的影响比较大。

为了将I/O操作和业务处理解耦，在将异步I/O接收到的数据传递给业务线程处理之前需要对数据做统一的处理，比如对接收到的二进制数据进行解码，统一的处理需要封装成链式调用的模式，这样设计的目的是便于开发者自定义处理流程，同时还需要把统一处理的逻辑放到线程池去执行，避免创建新线程导致的时间的开销。在异步I/O编程中责任链模式的统一处理逻辑和线程池的封装都需要开发者自己去实现，因为涉及多线程编程的处理，所以对开发者水平要求较高，由于是异步的，出现问题debug也比较困难。

⑵ 相比同步非阻塞I/O，异步I/O的处理性能的提升并不明显。

Netty放弃了对异步I/O支持的主要原因是在连接的处理性能与同步非阻塞I/O相比没有太大的优势。虽然异步I/O在服务器线程占用方面优势明显，性能方面的优势主要是在处理连接数比较多且连接比较长的场景，比如文件服务器。虽然在Client端和Server建立的是长连接，但是只是为了解决请求频繁建立和释放TCP连接导致的时间和服务器资源消耗问题，在长连接上传输的仍是像HTTP协议这样的短连接的协议数据，相对于文件传输操作比较轻，对时间更加敏感。

在Server端因为要处理多个连接对稳定性和性能要求更高，因此需要牺牲一些服务器资源换取Server端提供服务的稳定性。在Server端的优化主要是在Reactor线程模型的优化，Reactor有三种模式来处理多连接场景，以下是对这三种模式的介绍:

⑴ 单Reactor单线程模型: Reactor对象通过Select监听网络I/O事件，如果是Client端的连接事件，分发器dispatch会将事件分发到acceptor处理Handler的创建，其它网络I/O事件直接传递到Handler去处理。所有操作都是在一个线程处理,Handler的调用没有实现异步，因此承载的连接数有限，对于服务间调用这种高并发的场景不适用。

⑵ 单Reactor多线程模型: 和单Reactor单线程模型相比，多线程模型对Handler的处理是放到线程池中进行的，降低了Reactor线程的性能开销，进而提升系统的吞吐量。

⑶ 主从Reactor多线程模型: 和单Reactor 多线程模型相比，将Reactor上的分发到Handler的事件监听放到单独Reactor(从Reactor)去处理，主Reactor只处理Client端的连接建立事件。主acceptor线程接收到连接事件以后，创建ServerSocketChannel并在Channel上绑定网络I/O数据读写的Handler,并将其注册到从Reactor上处理。和单Reactor多线程模型相比，在响应时间和系统的吞吐量方面有很大的提升。

通过对Reactor模型的三种模式的分析对比，主从Reactor多线程模型更适合Server端多连接高并发的场景，在Client只处理和Server的连接，选择单Reactor多线程模型，避免服务器资线程资源的开销。

3.2.1.5 多线程技术的需求分析

在服务调用的Server端和Client端采用了长连接的通信方式作为服务间调用数据传输的方式，在接受到消息之后要将消息转发到处理响应的线程。

在Server端，Netty通信框架启动了两个EventLoopGroup分别处理NIO Accept和NIO I/O读写，但是尽量不要在ChannelHandler中启动用户线程，解码后将参数对象编码的二进制数据发送到业务线程去处理，也即是说不要在EventLoopGroup线程内处理业务逻辑，自定义协议的解码可以放在EventLoopGroup,即添加编解码的ChannelHandler。考虑直接创建线程去处理参数对象涉及的业务，会消耗时间，因此采用将接受到的参数对象的二进制封装为Taskr任务,放到阻塞队列中，由线程池来消费。在Netty中所有的I/O操作都是异步的，因此需要把与Channel绑定的ChannelHandlerContext对象也封装到Task任务中，业务线程处理后的结果写入到Channel返回到Client端。

在Clinet端向Server端发送请求之后，将根据Server响应结果处理的任务封装为Task,以请求消息的uuid为key值，Task任务为Value值放入到TaskMap中，当Server端的结果返回之后，以返回消息的uuid为Key，返回响应结果的编码二进制为Value，将返回的结果放到ValueMap中，同时将存入Map的Task任务根据uuid取出来，放到阻塞队列中，由线程池的线程来执行。Task任务执行时需要从存放响应结果的ValueMap中取出响应结果的编码二进制，此处的ValueMap为线程安全的Map，因为可能出现执行Task任务的线程池的多个线程去ValueMap取值。Client端封装的Task任务是实现Runnable接口， run方法无法接收参数，为了将请求信息的uuid传入Task，需要创建抽象类ParentTask，ParentTask有一个uuid的属性值。

3.4 智慧管廊系统服务调用中间件架构设计

通过对智慧管廊系统服务调用中间的需求分析，找出系统在服务调用方面出现的问题，设计新的服务间调用架构解决出现的问题。

3.4.1 服务间调用通信架构的设计

由于直接操作操作Java NIO 的API 进行网络I/O的读写难度较大，同时还要处理网络编程涉及到的多线程，因此引入Netty通信框架。在编程的角度，网络I/O需要处理的问题有两个方面：

⑴ 网络I/O的读写。

⑵ 对网络通信事件的处理。

在Netty通信框架提供了ByteBuf 对象进行提供对I/O进行缓存读写，同时将网络通信出现事件封装为事件处理函数供开发者调用。使用Netty通信框架屏蔽掉Java 底层NIO的操作，降低网络编程的难度。通信框架设计如下:

3.4.1.1 事件处理链事件

Netty中的事件分为inbound事件和outbound事件。Inboud事件通常是由I/O线程触发，outboud事件的通常是由用户主动发起的网络I/O操作。

图片包含 游戏机

描述已自动生成

图3-5 Server端pipeline事件处理handler

地图的截图

描述已自动生成

图 3-6 Client端pipeline事件处理handler

处理网络I/O事件，在Handler中重写不同的事件处理方法即可，比如处理Client端的断开是在Handler中重写userEventTriggered(ChannelHandlerContext ctx, Object evt)方法。

3.4.1.2 网络I/O数据的传输

Netty的底层I/O实现是同步非阻塞I/O,在Server端和Client端处理不同，在Server端主要处理的场景是多个Client端的连接，Client端只与一个Server进行通信，因此只需要处理一个TCP连接。在Server端使用I/O多路复用技术，把多个I/O阻塞复用转移到同一个Selector的阻塞，因此在Server端使用两个线程线程池来处理网络I/O,一个线程池处理NIO Acceptor 另外一个线程池处理NIO I/O的读写。在Client只处理一个I/O读写，使用一个线程池即可完成I/O的操作。

地图的截图

描述已自动生成

图 3-7 网络数据传输过程

3.4.2 多线程架构的设计

在Server端在接收到多个Client端的请求，会启动多个线程去处理请求的数据，必然会涉及到线程的启动和回收。为了避免线程创建增加请求的时间，引入Java线程池来创建线程，线程池内会维持预先创建一定数量的线程，个数由线程池的参数corePoolSize来设置。在Client端的请求量比较小时，线程池也会回收多余的线程，减少服务器资源占用，回收的线程个数是由maxPoolSize 和 corePoolSize的差值确定的。

地图的截图

描述已自动生成

图 3-8 Server端多线程的设计

在Server端服务的请求频率是不相同的，可以把处理请求频率高的服务的Task任务放到单独创建的线程池去处理。

在Client端多个请求共用同一个Channel,因此不能将Task任务像Server端一样放在队列中去处理，Client端在Server的响应结果返回之后再把处理响应的Task任务放到线程池去处理，架构设计如下:

地图的截图

描述已自动生成

图 3-9 Client端多线程的设计

在Client端，Channel从TaskMap取任务、线程从RespMap取数据和Channel将响应数据放入RespMap用的都是自定义协议消息的唯一标识uuid。

3.4.3 kubernetes集群服务设计

智慧管廊系统的不同业务模块映射到Kubernetes集群中就是不同的Service,为了保持多活，Service下面的Pod被打散分部在不同的服务器上，为了使Client端能够请求到距离最近的Server，需要在节点上打上标签，因为kube-proxy是根据节点上的标签在节点写入转发规则，架构设计如下：

地图的截图

描述已自动生成

图 3-10 Kubernetes内部服务调用

在Kubernetes集群内部ServiceA下的pod请求ServiceB下pod,A1会请求B1而不是B2,同样的A2会请求B2而不是A1。

3.5 本章小结

在本章中分析了智慧管廊系统服务调用中间件的需求，在满足的这些需求的情况下同时又兼顾服务器资源占用优化以及缩短请求的响应时间，对服务调用中间件进行设计。在设计中主要关注以下几点:

⑴ 通过自定义协议，解决HTTP协议Header冗余的问题。

⑵ 使用高性能的二进制编码框架减小网路传输数据的大小。

⑶ 使用长连接解决HTTP协议频繁的连接建立和释放带来的时间和性能的损耗。

⑷ 设计多线程架构提升服务器资源的利用效率。

⑸ 引入Kubernetes新特性解决请求被转发到距离较远的Server。

第四章 智慧管廊系统服务调用中间件的实现

本章主要内容是根据第三章智慧管廊系统的功能需求实现服务调用中间件。

4.1自定义通信协议的实现

在设计通信协议时需要考虑到服务间调用的需求，又要考虑到兼容流量控制信息和链路追踪的升级功能。通信协议的设计具体如下:

表4-1 自定义通信协议的请求信息ProtocolReqMsg

|  |  |  |
| --- | --- | --- |
| 消息字段 | 消息类型 | 描述 |
| uuid | String | 消息的id信息 |
| version | int | 服务的版本 |
| chain | String[] | 服务的调用链 |
| path | String | 调用的服务路径 |
| body | byte[] | 调用消息的序列化数据 |

uuid字段为消息的唯一标示，该字段会打印日志信息，方便问题的定位，同时根据该字段将Server端返回消息放入处理对应响应的Client端的线程。version字段为服务的版本号，便于sidecar模式的组件做流量控制。Chain字段为服务的调用链信息，服务调用链请求成功的服务会写入该字段，便于在请求客户端定位出现问题的服务节点。Path字段为请求的服务的路径，Server端根据Path路径调用注册在context中的服务。Body字段为请求对象序列化后的二进制，在Server端服务方法中反序列化，读取Client端传递的参数对象。

表格 4-2 自定义通信协议的响应信息ProtocolRespMsg

|  |  |  |
| --- | --- | --- |
| 消息字段 | 消息类型 | 描述 |
| uuid | String | 消息的id信息 |
| version | int | 服务的版本 |
| chain | String[] | 服务的调用链 |
| status | int | 调用的服务路径 |
| Body | byte[] | 调用消息的序列化数据 |

Status字段为响应的状态码，状态码的具体定义如下：

表格 4-2 自定义通信协议响应的状态码

|  |  |  |
| --- | --- | --- |
| 状态码 | 含义 | 描述 |
| 200 | OK | 服务端请求成功 |
| 207 | Message Control | 该服务的版本被限制 |
| 408 | Request Time-out | 服务请求超时 |
| 500 | Internal Server Error | 服务内部出错，无法完成请求 |
| 502 | Request Next Service Error | 服务端调用下一个服务失败 |
| 506 | Request service does not exist | 请求的服务路径不存在 |
| 507 | Disconnection | 连接断开 |
| 600 | HeartBeat | 定时任务心跳 |

4.2 Java对象编解码的实现

在选择编解码框架时，不仅要考虑序列化之后码流的大小与序列化和反序列化的性能，还要考虑版本升级的兼容和对云平台其它模块异构语言的支持以及开发者的工作量和开发难度。综合考虑这些因素后，选择Google的Protobuf编解码框架实现Java对象的序列化和反序列化。

在智慧管廊系统服务调用中间件中用到Protobuf的地方有两个:自定义通信协议的对象和服务消费者请求的参数对象。Protobuf使用可执行程序根据数据描述文件.proto生成相对应的Java对象。

4.2.1 Java对象的proto文件

⑴ 自定义通信协议消息对象的proto文件：

1. //自定义通信协议请求消息对象.proto文件
2. syntax = "proto3";
3. option java\_generic\_services = **true**;
4. option java\_package = "cn.bupt.edu.protocol";
5. option java\_outer\_classname = "ProtocolReqMsgProto";
6. message ProtocolReqMsg {
7. string **uuid** = 1;
8. int32 version = 2;
9. repeated string chain =3;
10. string path = 4;
11. bytes body = 5;
12. }
13. //自定义通信协议响应消息对象的.proto文件
14. option java\_generic\_services = **true**;
15. option java\_package = "cn.bupt.edu.protocol";
16. option java\_outer\_classname = "ProtocolRespMsgProto";
18. message ProtocolResqMsg {
19. string **uuid** = 1;
20. int32 version = 2;
21. repeated string chain =3;
22. int32 status = 4;
23. bytes body = 5;
24. }

⑵ 服务调用参数对象的proto文件

1. //设备信息对象的proto文件
2. syntax = "proto3";
3. option java\_generic\_services = **true**;
4. option java\_package = "cn.bupt.edu.entity";
5. option java\_outer\_classname = "DeviceInfoProto";
6. message DeviceInfo {
7. string id = 1;
8. string name = 2;
9. int32  temperature =3;
10. }

通过静态方法newBuilder()，创建Builder对象，通过Builder构造器设置属性值，对于数组对象用setXXX(index,value)将数组对象设置到对应的属性中，然后使用toByteArray()进行编码，解码调用parseFrom(byte[] body)即可。在Client端调用toByteArray()方法将参数对象进行编码，生成二进制数据，放入自定义通信协议请求消息对象ProtocolReqMsg的body属性值，然后对ProtocolReqMsg对象进行编码，Server端在收到ProtocolReqMsg的二进制数据之后，解码生成ProtocolReqMsg对象,body中的二进制数据是在调用服务的具体实现方法中解码的。Server端的业务线程将处理的结果编码放到ProtocolRespMsg对象的body属性中，在ChannelHandler中将ProtocolRespMsg对象编码后发送到Client端。Client端把收到的二进制数据解码生成ProtocolResp对象，将其body属性中的二进制数据回传到处理响应的线程，Client端业务处理线程将二进制数据解码获取响应结果对象。过程的时序图如下图所示:

图片包含 空地, 站, 许多, 一群

描述已自动生成

图 4-1 服务调用序列化和反序列化时序图

4.3 网络通信框架和多线程并发的实现

4.3.1网络通信框架的实现

4.3.1.1 长连接的实现

智慧管廊系统中服务消费者请求服务提供者数据传输需要频繁地建立和释放TCP连接。TCP建立连接需要三次握手，断开连接需要四次挥手，每次进行服务调用都要重复这样的过程造成时间和资源的消耗。为了解决这个问题在服务调用中引入长连接，长连接在TCP连接建立后，并不会在一次请求之后就释放，而是一直保持。当服务消费者再次调用服务时，可以直接通过已经建立的TCP连接进行数据的收发。

在TCP的机制里面，本身是存在心跳包机制，即TCP的选项SO\_KEEPALIVE。系统默认设置是2小时的心跳频率。由于时间间隔过长，无法对于一些突发的事件做出反应，比如机器断电、网线拔出、防火墙等异常情况导致的网络中断。因此需要设计心跳机制以便能在网络出现异常重新尝试建立TCP连接。

在Netty的ChannelPipeline处理的事件包含捕获异常的事件，即ChannelHandlerContext.fireExceptionCaught( Throwable )，在ChannelHandler中重写**public void** exceptionCaught( ChannelHandlerContext ctx, Throwable cause ) **throws** Exception {}方法就可以对读写超时的异常进行处理。同时在Netty中提供了IdleStateHandler，在设置读、写、读写超时时间后，该handler就还抛出异常，以供其它handler捕获处理。

在Server端会主动断开连接，连接断开会导致Server端的处理结果无法发送到Client端，为了避免服务器线程资源的浪费，在断开连接的同时取消Channel接收的在线程池阻塞队列中没有处理的业务Task执行和停止线程池中处理Client端请求的线程的运行,具体代码如下:

@Override  
**public void** userEventTriggered(ChannelHandlerContext ctx, Object evt) **throws** Exception {  
 *//释放Client端处理响应的Task任务的缓存* DataContext.*getInstance*().ResetModule(**this**.**serviceName**);  
 **if** (evt **instanceof** IdleStateEvent ) {  
 reconnect(ClusterIp.*GetClusterIP*(),**this**.**port**);  
 }**else** {  
 ctx.close();  
 }  
}

Server端主动断开连接，不再返回Client请求的响应结果，Client端缓存的处理响应结果的Task任务永远不会放到线程池中处理。为了避免服务器内存资源的开销，需要释放Task任务的缓存。为了不影响服务间的调用，Client端在连接断开之后需要重新建立TCP连接，因此捕获异常之后，会在客户端发起重新连接。由于Client端无法获取连接断开的具体原因。具体实现代码如下:

@Override  
**public void** userEventTriggered( ChannelHandlerContext ctx, Object evt ) **throws** Exception {  
 **if** (evt **instanceof** IdleStateEvent ) {  
 reconnect(ClusterIp.*GetClusterIP*(),**this**.**port**);  
 }**else** {  
 ctx.close();  
 }  
}

服务消费者发送定时心跳的Task任务可以由与Channel绑定的线程池EventLoopGroup完成，即调用scheduleAtFixedRate(Runnable var1, long var2, long var4, TimeUnit var6)方法来实现，心跳时间可以在配置文件中指定，默认值为10s。服务消费者发送的定时任务的请求消息ProtocolReqMsg的body属性可以为空，具体实现代码如下:  
@Override  
**public void** channelActive(ChannelHandlerContext ctx) **throws** {  
 ctx.channel().eventLoop()

.scheduleAtFixedRate(**new** HeartBeatHandler

.HeartBeatTask(ctx,**this**.**version**), 0, 1, TimeUnit.***SECONDS***);  
}

Client端调用处理心跳的方法都在HeartBeatHandler中，该类继承ChannelInboundHandlerAdapter类，重写了channelActive( ChannelHandlerContext ctx )和userEventTriggered(ChannelHandlerContext ctx, Object evt )方法。

Server端在接受到Client端发送的心跳包之后，设置响应消息ProtocolRespMsg的状态码为600，然后返回给Client端。

4.3.1.2 NIO的实现

在Client端只需要处理和Server的连接，考虑服务器线程资源占用和处理消息的时间消耗，选择单Reactor多线程的模型处理网络I/O。使用Netty中的EventLoopGroup管理多线程，创建Pipeline的Factory工厂类，工厂类是继承ChannelInitializer<SocketChannel>，重写initChannel()方法，将处理I/O读写数据的Handler注册到pipeline上，具体代码实现如下:

Bootstrap clientBootstrap = **new** Bootstrap();  
*//创建Reactor处理线程池*EventLoopGroup reactor = **new** NioEventLoopGroup();  
clientBootstrap.group(reactor);  
clientBootstrap.channel(NioSocketChannel.**class**);  
*//创建Pipeline的工厂类处理网络I/O数据*bootstrap.handler(**new** ChannelPipelineFactory());

在Server端因为要处理多个Client的连接，同时还要处理高并发请求，因此在Server端的实现上采用主从Reactor多线程模型。创建两个EventLoopGroup分别处理连接建立和I/O读写的Handler事件，创建Pipeline的工厂类注册处理网络I/O读写数据的Hander，具体代码实现如下:

ServerBootstrap serverBootstrap = **new** ServerBootstrap();  
*//处理连接建立的Acceptor线程池*EventLoopGroup acceptor = **new** NioEventLoopGroup();  
*//处理网络I/O读写的线程池*EventLoopGroup handler = **new** NioEventLoopGroup();  
serverBootstrap.group(acceptor,handler);  
serverBootstrap.channel(NioServerSocketChannel.**class**);  
*//创建Pipeline的工厂类注册Handler处理网络I/O数据*bootstrap.childHandler(**new** ChannelPipelineFactory());

4.3.2 多线程控制的实现

4.3.2.1 线程池和Task任务的实现

Java的线程池任务类型有三种:

⑴ Runnable: 其中的run方法没有返回值。Runnable对象可以直接传入Thread创建线程实例，并且创建的线程实例与Runnable绑定，线程实例调用start()方法时，Runnable任务就开始真正在线程中执行，但是直接调用run()方法，只是相当于普通的方法执行，并不会开启新的线程，程序只是在主线程中串行执行。Runnable对象可以直接传入线程池对象的execute方法或者submit方法，让线程池为其绑定池中的线程来执行。Runnable对象也可以进一步封装成FutureTask对象之后再交给线程池的execute方法和submit方法。

⑵ Callable : 功能相比Runnable来说少很多，不能用来创建线程，也不能直接交给线程池的execute方法，但是其中的call方法有返回值。

⑶ FutureTask: 是对Runnable和Callable的进一步封装，并且这种任务是有返回值的, 相比直接把Runnable和Callable扔给线程池，FutureTask的功能更多，它可以监视任务在池子中的状态。使用Callable来创建，由于call方法本身有返回值，这个返回值也就是Callable对象的返回值。

Client端发送的请求消息在Server端的pipeline的ChannelHandler解码为ProtocolReqMsg对象，在启动新的线程去执行业务逻辑时候，有三种方案创建Task任务：

⑴ 方案一：将ProtocolReqMsg对象和ChannelHandlerContext对象直接传入Runnable接口实现的子类，将ChannelHandlerContext传入Runnable子类的原因是Task任务处理后的结果通过ChannelHandlerContext对象获取Channel进行网络I/O的写操作。

⑵ 方案二: 将方案一封装的Runnable任务，放入FutureTask对象再交给线程池处理，这样做的目的是在Channel关闭时停止线程池运行的业务线程和取消线程池阻塞队列中Task任务，而不是在线程处理完之后出现写入Channel失败。在Server端无法与Client端通信时及时终止执行线程，减少服务器线程资源的开销。

⑶ 方案三: 将ProtocolReqMsg对象传入Callable接口实现类，并将Callable的实现类对象封装成FutureTask对象，放入阻塞队列，由线程池去处理。Pipeline的ChannelHandler处理线程在FutureTask结果返回后再写入ChannelHandlerContext的Channel中，而不需要在封装Task任务时传入ChannelHandlerContext对象。在pipeline的ChannelHandler线程中并没有其它任务需要执行，因此这样的异步操作会造成线程阻塞，导致服务器资源的开销。

通过对比三种方案可以发现，方案二可以解决在Server端Channel异常关闭情况下，终止线程池中运行的线程和取消在阻塞队列中等待执行的Task，在Channel关闭后业务线程处理的结果无法返回到Client端，造成资源的浪费。

在Java中可以通过ThreadPoolExecutor()创建线程池，其参数有以下几个：

⑴ corePoolSize：核心线程数 ，在没有任务执行时，核心线程数的线程会一直存活，在线程数没有达到核心线程数时线程池会自动创建线程。设置allowCoreThreadTimeout=true（默认false）时，核心线程会超时关闭。

⑵ queueCapacity：任务队列容量（阻塞队列），当线程池中的线程数达到最大线程数时，新加入线程池任务会放在队列中排队等待执行。

⑶ maxPoolSize：最大线程数 ，当线程数>=corePoolSize，且任务队列已满时，线程池会创建新线程来处理任务。当线程数=maxPoolSize，且任务队列已满时，线程池会拒绝处理任务而抛出异常。

⑷ keepAliveTime：线程空闲时间，当线程空闲时间达到keepAliveTime时，线程会退出，线程池保留corePoolSize数量的线程。如果allowCoreThreadTimeout=true，则线程池中的线程会全部关闭。

⑸ allowCoreThreadTimeout：允许核心线程超时退出。

⑹ rejectedExecutionHandler：任务拒绝处理器。

服务运行在kubernetes集群内部，kubernetes会根据Pod所占的服务器资源自动的扩缩容，因为可以把核心线程数和最大线程数设置的差值比较大，这样就可以依靠Kubernetes对Pod进行扩缩容。

在设计Server端的线程池需要考虑服务的链式调用场景。服务的链式调用场景可以简单描述为: ClientA 请求ServerA，ServerA调用MethodA方法处理ClientA的请求时使用ClientB去请求ServerB。为了便于在分布式系统中定位问题，ClientB在请求ServerB时，请求协议对象ProtocolReqMsg中的uuid应和ClientA请求Server的请求消息的uuid保持一致。服务的日志会存储到统一日志的ElasticSearch中，在Elasticsearch中根据请求信息的uuid将链式调用的服务关联起来就可以获取整个调用链的日志。为了降低开发人员开发做业务开发的难度，uuid的传递应该在服务调用中间件处理，而不是暴漏接口给开发者调用。ClientB请求ServerB和ServerA的MethodA方法运行在两个线程中，因此uuid的传递涉及到线程间的通信。基于Java的Thread线程无法实现uuid传底，因此需要对Thread线程类进行封装。创建ParentThread类继承Thread类，在ParentThread类添加uuid的属性，重写Thread的run方法，在run方法中执行构造ParentThread对象传入的Task任务。使用ThreadPoolExecutor创建线程池时，可以传入创建线程的线程工厂类来创建线程池中的线程。创建TFactory类实现ThreadFactory，在newThread()方法中返回ParentThread线程对象。在MethodA中调用ClinetB请求ServerB时，通过Thread.currentThread方法获取MethodA运行的线程，将Thread线程对象强制转化为ParentThread线程对象，获取ParentThread的uuid放入ClientB请求ServerB的请求信息ProtocolReqMsg对象的uuid属性中。ServerA的线程池执行的Task任务是在接收CientA请求之后将ProtocolReqMsg对象封装的Task任务，主要处理请求的消息的业务逻辑。ServerA处理ClientdA的请求的。Server在构建ParentThread对象时传入的Task任务包含ClientA请求ServerA的ProtocolreqMsg对象，即ParentTask，可以在ParentTask的run()方法中将ProtocolReqMsg的uuid传递给ParentThread。具体实现代码如下:

1. **public** **abstract** **class** ParentTask **implements** Runnable {
2. **public** ProtocolReqMsgProto.ProtocolReqMsg Req;
3. **public** ChannelHandlerContext Ctx;
4. **public** ParentTask(ProtocolReqMsgProto.ProtocolReqMsg r,ChannelHandlerContext c){
5. **this**.Ctx=c;
6. **this**.Req = r;
7. }
8. **public** **void** initThread (){
9. Thread t = Thread.currentThread();
10. ParentThread p = (ParentThread) t;
11. //设置服务调用链chains
12. String[] chains = p.getChains();
13. **for** (**int** i = 0 ; i < **this**.Req.getChainCount();i++){
14. chains[i] = **this**.Req.getChain(i);
15. }
16. //设置uuid
17. p.setUuid(**this**.Req.getUuid());
18. }
19. }

在链式调用的场景下，和uuid一样自定义通信协议中的调用链信息也需要在线程间传递。ClientB的Task任务需要把ServerB的响应处理结果返回给MethodA,因此选择FutureTask作为Client端的Task任务类型。为了将ProtocolRespMsg中的调用链信息回写到MethodA,需要创建ParentFutureTask类对FutureTask对象进行封装，重写get()方法，在方法中加入回写调用链的逻辑。由于ParentFutureTask的get()方法是在MethodA的线程中执行的，调用Thread.currentThread()返回的线程是MehtodA的线程。在ClientB收到响应结果之前，ServerB和ClientB之间的长连接断开会导致ClientB的处理响应的Task任务永远不会放到线程池执行，这种情况下执行MethodA的线程会一直处于阻塞状态。在Client端和Server端的连接断开后，Server端会关闭处理Client端请求的线程和取消阻塞队列中处理Client端请求的Task任务的执行。有以下两种方案可以解决ServerB重启导致ClientB永远不会收到响应消息:

⑴ 在ClientB中设置请求超时。FutureTask的get()方法可以设置超时时间，超过设置的时间断就会抛异常。将响应消息的服务调用链信息回写到MethodA的操作放到FutureTask中去执行，超时异常时需要开发者实现服务调用链信息的回写。为了避免服务器内存资源的开销，开发者需要释放TaskMap中Task任务的缓存。为了避免需要处理复杂的服务调用中间件的底层调用对用户造成的不友好，需要在ParentFutureTask中重写FutureTask的设置超时的get(),封装回写服务调用链和释放Task任务缓存的中间件底层的逻辑。

⑵ 在连接断开时生成自定义协议状态码为507的响应消息。在ClientB的长连接关闭时，将Client端的TaskMap中所有未执行的任务放到线程池的队列中去处理，同时将生成状态码为507的ProtocolRespMsg的消息放入RespMap中。

方案一和方案二并不是完全互斥，ServerA的开发者为了保证MethodA的处理请求的响应时间可以设置超时时间避免ServerB的响应时间过长对MethodA的处理效率产生影响，在响应返回之前主动结束请求。在方案二的实现中需要先从TaskMap中获取要执行的Task任务，如果获取的Task为null，就不把响应结果放到RespMap中，避免服务器内存资源的开销。具体代码在长连接实现的模块。

4.3.2.2 Server端多线程控制的实现

在Server端，Netty通信框架启动了两个EventLoopGroup分别处理NIO Accept和NIO I/O读写，但是尽量不要在ChannelHandler中启动用户线程，解码后将参数对象编码的二进制数据发送到业务线程去处理，也即是说不要在EventLoopGroup线程内处理业务逻辑，自定义协议的解码可以放在EventLoopGroup,即添加编解码的ChannelHandler。考虑到创建线程会有时间开销，在Handler中将接收到的参数对象的二进制封装为Taskr任务,放到阻塞队列中，由线程池来消费。在Netty中所有的I/O操作都是异步的，因此需要把与Channel绑定的ChannelHandlerContext对象也封装到Task任务中，使得业务线程处理后的结果可以返回到客户端。具体实现步骤如下:

⑴ 将ProtocolReqMsg消息封装为Task任务。Client端发送的请求消息，在Server的ChannelPipeline解码为ProtocolReqMsg对象之后，将ProtocolReqMsg对象和ChannelHandlerContext对象封装为ParentTask任务，ParentTask是一个抽象类，不同的业务类型的Task继承ParentTask的抽象类，因为要在方法调用前对参数对象进行统一的反序列化操作生成对应的参数对象,使用TaskFactory工厂类创建Task任务类对象，将Task任务放到FutureTask对象中，并将FutureTask对象放到阻塞队列中,同时在Handler中维护一个Futuretask的数组，在Channel关闭后，取消数组中所有未完成的Task任务。具体代码如下:

ParentTask task = TaskFactory.*GetTask*((ProtocolReqMsgProto.ProtocolReqMsg) msg, ctx);  
FutureTask<Void> ftask = **new** FutureTask<Void>(task, **null**);  
BlockQueue.*Add*(ftask);  
**for** (**int** i = 0; i < **flist**.**length**; i++) {  
 **if** (**flist**[i].isDone()) {  
 **flist**[i] = ftask;  
 }  
}

⑵ 在用户业务线程处理业务逻辑并将处理结果返回。通过自定义注解@Handle( path = “/v1/device/XX”),v1是服务的版本号、device为Spring Bean的名称、XX为方法method的路径，在服务启动时将服务路径path和对应的spring bean对象注册到HandlerContext中，在处理业务逻辑时通过HandlerContext.GetHandlerMethod( String path)获HandlerMethod对象，HandlerMethod包含服务的spring bean , 根据路径匹配的bean 下面的method方法，通过java反射的invoke()方法调用业务函数去处理Client的请求。由于在同一个Spring Bean 下面的method参数对象是相同的，在method调用前对参数对象统一进行反序列化。具体处理代码如下:

1. @Override
2. **public** **void** run() {
3. //设置线程属性
4. **super**.initThread();
5. ProtocolResqMsgProto.ProtocolRespMsg.Builder builder = ProtocolResqMsgProto.ProtocolRespMsg.newBuilder();
6. HandlerMethod hm = ContextUtil.getContextHandler().GetHandler();
7. **try** {
8. //将二进制数据解码为DeviceInfo对象
9. DeviceInfoProto.DeviceInfo device = DeviceInfoProto.DeviceInfo.parseFrom(**this**.Req.getBody());
10. //调用业务处理方法
11. Object resp = hm.method.invoke(hm.object, device);
12. **byte**[] rb = (**byte**[]) resp;
13. //将处理结果编码后的二进制对象放入ProtoRespMsg对象的body中
14. builder.setBody(ByteString.copyFrom(rb));
15. builder.setStatus(200);
16. } **catch** (Exception e) {
17. builder.setBody(ByteString.copyFrom(e.toString().getBytes()));
18. builder.setStatus(500);
19. e.printStackTrace();
20. } **finally** {
21. ProtocolResqMsgProto.ProtocolRespMsg result = builder.build();
22. **this**.Ctx.writeAndFlush(result);
23. }
24. }

把业务处理Task任务从阻塞队列中取出放到线程池中处理的，获取结果之后写入到Channel中。

4.3.2.3 Client端多线程控制的实现

Client端和Server端通过长连接进行通信，Client端的多个请求复用同一个TCP连接，与Server端不同的是，在Client只有一个Channel通道，因此在消息返回后需要根据ProtocolRespMsg的uuid对消息进行分发，需要实现一个分发器功能的组件完成分发。具体实现如下:

Client端将请求参数对象编码为二进制数据封装到请求消息ProtocolReqMsg的对象中，同时将处理响应的业务逻辑封装为FutureTask类型的Task任务缓存到TaskMap中，而不是交给线程池去处理，因为响应结果没有返回，线程会一直处于阻塞状态。ChannelHandler将ProtocolReqMsg编码后的二进制写入Client端的socket，将数据发送到Server端。当Channel接收到Server端的响应消息后，将任务从Map中取出放入线程池执行即可。

4.4 服务注册和发现的实现

在Kubernetes 集群中Service 的 Cluster IP 是全局唯一的，因此服务之间的通信即是TCP通信，在TCP通信中Client 端向服务端发送请求需要对请求头中的域名做DNS解析，在kubernetes集群服务调用中也需要把Servic Name 解析为 Cluster IP ,由于在kubernetes 集群中Cluster IP 唯一不变，因此可以将Cluster IP在服务启动的时候缓存到本地，避免DNS解析引起的时间开销。引入dnsjava的Maven 依赖，调用InetAdress.getAllByName(String name)即可获取Service 的Cluster IP。

Service 转发主要是 node 上的 kube-proxy 进程通过 watch apiserver 获取 Service 对应的 endpoint，再写入 iptables 或 ipvs 规则来实现的; 对于 headless service，主要是通过 kube-dns 或 coredns 动态解析到不同 endpoint ip 来实现的。实现 Service 就近转发的关键点就在于如何将流量转发到跟当前节点在同一拓扑域的 endpoint 上，也就是会进行一次 endpoint 筛选，选出一部分符合当前节点拓扑域的 endpoint 进行转发。通过 endpoint 所在节点的 label，我们可以使用 node label 来描述拓扑域。

通常在节点初始化的时候，controller-manager 就会为节点打上许多 label，比如 kubernetes.io/hostname 表示节点的 hostname 来区分节点；另外，在云厂商提供的 k8s 服务，或者使用 cloud-controller-manager 的自建集群，通常还会给节点打上 failure-domain.beta.kubernetes.io/zone 和 failure-domain.beta.kubernetes.io/region 以区分节点所在可用区和所在地域，但自 v1.17 开始将会改名成 topology.kubernetes.io/zone 和 topology.kubernetes.io/region。

在 Service spec 里加上 topologyKeys 字段，表示该 Service 优先顺序选用的拓扑域列表，对应节点标签的 key；当访问此 Service 时，会找是否有 endpoint 有对应 topology key 的拓扑信息并且 value 跟当前节点也一样，如果是，那就选定此 topology key 作为当前转发的拓扑域，并且筛选出其余所有在这个拓扑域的 endpoint 来进行转发；如果没有找到任何 endpoint 在当前 topology key 对应拓扑域，就会尝试第二个 topology key，依此类推；如果遍历完所有 topology key 也没有匹配到 endpoint 就会拒绝转发，就像此 service 没有后端 endpoint 一样。

 topology key 支持的值:

⑴ kubernetes.io/hostname: 节点的 hostname，通常将它放列表中第一个，表示如果本机有 endpoint 就直接转发到本机的 endpoint。

⑵ topology.kubernetes.io/zone: 节点所在的可用区，通常将它放在 kubernetes.io/hostname 后面，表示如果本机没有对应 endpoint，就转发到当前可用区其它节点上的 endpoint（部分云厂商跨可用区通信会收取额外的流量费用）。

⑶ topology.kubernetes.io/region: 表示节点所在的地域，表示转发到当前地域的 endpoint，这个用的应该会比较少，因为通常集群所有节点都只会在同一个地域，如果节点跨地域了，节点之间通信延时将会很高。

⑷ \*: 忽略拓扑域，匹配所有 endpoint，相当于一个保底策略，避免丢包，只能放在列表末尾。

除此之外，还有以下约束:

⑴ topologyKeys 与 externalTrafficPolicy=Local 不兼容，是互斥的，如果 externalTrafficPolicy 为 Local，就不能定义 topologyKeys，反之亦然。

⑵ topology key 必须是合法的 label 格式，并且最多定义 16 个 key。

引入拓扑感知服务路由的特性的方法就在创建Service 的yaml文件中添加spec中添加topologyKeys,具体添加方式如下：

**apiVersion**: v1  
**kind**: Service  
**metadata**:  
 **name**: nginx  
**spec**:  
 **type**: ClusterIP  
 **ports**:  
 - **name**: http  
 **port**: 80  
 **protocol**: TCP  
 **targetPort**: 80  
 **selector**:  
 **app**: nginx  
 **topologyKeys**: [**"kubernetes.io/hostname"**, **"topology.kubernetes.io/zone"**, **"\*"**]

4.5 本章小结

本章主要是对第三章设计的架构的实现，从代码的角度详细介绍了以下几个功能模块的实现:

⑴ 自定义协议的内容以及对未来云平台的升级功能做的兼容处理。

⑵ Java编解码技术的实现以及考虑到版本兼容所做的处理。

⑶ 长连接的实现，同时考虑到连接断开时及时取消Server端未完成的任务的执行。

⑷ Client端和Server端多线程架构的实现。

⑸ 服务注册和发现的实现以及优化。

第五章 测试与分析

本文的测试会用到多台服务器，并且要求服务器资源可以调整，同时还要模拟网络远距离请求的场景，考虑到实验室的服务器资源有限且不可以调整资源，同时实验室的网络环境也无法模拟远距离请求的场景，因此测试环境搭建在腾讯云服务器上。测试和分析主要集中在服务器资源占用优化和减少请求的响应时间两个方面，这两个方面优化也是本文实用价值所在。

5.1 服务器资源占用优化的测试与分析

在测试之前需要注意以下几点：

⑴ 服务器资源占用优化的测试不能在kubernetes集群内部进行。主要考虑到kubernetes会在Pod资源占用过多时进行扩容或者节点迁移和kubernetes集群内的节点会部署相关组件不利于资源占用数据的统计这两方面的原因。

⑵ 考虑到网络环境对测试结果的影响，选择CVM内网环境进行测试，这样就能避免通信数据在不确定的网络环境上的传输对测试结果造成的影响。腾讯云的客户主要集中在国内，考虑到其它用户流量对测试结果影响，因此云服务器测试地点选择国外地域。

⑶ Linux操作系统对单个进程打开文件描述符的个数是有限制的，一个TCP连接就占用一个文件描述符，一但文件描述使用完之后，就会报Socket/File:Can't open so many files的异常。同时还存在最大文件句柄数的限制。因此在测试之前需要修改单个进程打开的最大文件描述符和全局的最大文件句柄数的值。

⑷ 调整Spring Boot 内置的Tomcat的最大连接数和最大线程数的值。默认的最大连接数maxConnections = 10000,默认的最大线程数maxThreads = 200。

测试环境的准备：

⑴ 内网带宽: 1.5Gbps

⑵ 服务器资源：CPU型号: Intel Xeon E5-2680 v4(2.4 GHz)

⑶ 操作系统：Centos7.6 64位

⑷ JDK版本: openjdk 1.8.0\_242

测试指标:在相同服务配置下，本文设计的服务调用中间件和原来的系统Client端和Server端支持的最大并发数。

手机屏幕截图

描述已自动生成

图 5-1 长连接Client和HttpClient最大并发数对比

手机屏幕截图

描述已自动生成

图 5-2 长连接Server和HttpServer最大并发数对比

通过对比分析可以发现，在相同的配置下本文基于长连接技术和多线程技术的设计的服务调用中间件可以支持更多的并发访问，原因在于短连接的I/O阻塞线程会消耗服务器资源，在服务器配置更高的情况下这种优势更加明显，原因是大量的TCP连接建立和释放会消耗服务器资源。

5.2 减少响应时间的测试与分析

减少响应的时间的测试分为两个方面:长连接与编解码技术的测试和Service的转发优化的测试。

5.2.1 长连接的测试与分析

长连接的测试的选择在CVM上，同时测试的网络环境选择在外网环境，测试的准备条件如下：

⑴ 外网带宽:10M

⑵ 服务器资源：CPU型号: Intel Xeon E5-2680 v4(2.4 GHz)

⑶ 操作系统：Centos7.6 64位

⑷ JDK版本: openjdk 1.8.0\_242

测试指标：在相同的服务器配置下，本文设计的服务调用中间件和原来系统的请求的响应时间和TPS。

手机屏幕的截图

描述已自动生成

图 5-3 TPS测试对比

图 5-4 响应时间的对比测试

手机屏幕截图

描述已自动生成

通过测试分析可以发现长连接在请求数据比较小性能优于HTTP协议，但是传输的数据较大时，HTTP协议的优势更加明显，原因在于自定义协议的编解码会消耗时间，在数据量较大时编码码消耗的时间更多。通过对比分析可以发现，长连接更适用于服务间调用的场景。

5.2.2 Service优化转发功能的实现

Service的优化转发功能的是基于Kubernetes的1.17版本特性拓扑感知服务路由。

测试条件的准备如下:

⑴ Kubernetes版本： 1.17版本

⑵ 服务器: CPU型号: Intel Xeon E5-2680 v4(2.4 GHz)

四核8G

服务器A: 泰国曼谷

服务器B: 美国硅谷

⑶操作系统: Centos7.6 64位

测试内容： 服务转发的选择

测试结果截图如下:

手机屏幕的截图

描述已自动生成手机屏幕的截图

描述已自动生成

图 5-5node1的节点label

图 5-6node2的节点label

图片包含 游戏机

描述已自动生成

图 5-7 Service和pods

手机屏幕的截图

描述已自动生成

图 5-8 ServiceA的a1请求ServiceB的请求转发

ServiceA下的Pod a1和a2 分别运行在node1和node2 节点上，ServiceB下的Pod b1 和b2 分别运行在node1 和 node2 节点上。在a1中请求ServiceB 请求转发到了和a1同一节点的a2上。

5.3 本章小结

本节主要是测试设计的服务调用中间是否达到了预期的效果，通过分析测试数据可以看到本文设计的服务调用中间件在优化服务器资源占用和缩短请求响应时间方面达到了预期的效果。

第六章 总结与展望

6.1 总结

本文主要是针对现有智慧管廊系统在使用时出现的问题提供解决方案，该方案主要解决服务器资源的占用和请求响应时间两个方面的问题，以下分别对这两个方面进行总结:

⑴ 服务器资源占用的优化。智慧管廊采用OkHttpClient进行服务间的调用，在Client端发送请求之后会一直阻塞等待Server端结果的返回，同时Client端每次请求都会进行TCP的三次握手和四次挥手建立和释放连接。使用OkHttpClient进行服务间调用会占用大量的服务器线程资源，同时在连接建立和释放的过程中造成响应时间上的开销。为了解决这个问题引入了多线程技术和长连接技术。多线程技术的引入主要是解决服务器线程资源的占用，在Client端发送请求之后，将对Server端响应的处理封装成Task任务放入到Map中，结束线程，等Server端的响应结果返回之后再把处理响应结果的Task任务放到线程池中去处理，避免了线程阻塞等待响应结果造成的服务器线程资源的消耗。长连接技术的引入主要是解决Client每次请求Server端都要建立和释放TCP连接导致的服务器资源的损耗，使用长连接可以避免每次请求TCP连接的建立和释放，同时在长连接的底层使用同步非阻塞I/O技术(NIO)，内核将数据从内核空间复制到用户空间缓存之后再通知用户去读，在Server端使用I/O复用技术，启动一个selector线程去监听异步I/O事件，避免多个线程监听导致的服务器资源的消耗。

⑵ 缩短请求的响应时间。智慧管廊系统服务间调用的数据传输的协议是HTTP协议，HTTP协议Header的信息比较多，服务间的调用多是小数据量的数据，因此会导致网络传输的数据冗余比较大，数据在网络上传输耗时比较多，导致请求响应时间的增加。为了解决这个问题引入自定义协议和Java对象的编解码技术。自定义协议的引入主要解决HTTP协议请求头信息冗余的问题，简化协议头部的设计，为了兼容之后流量控制的功能，在协议的请求头中添加版本的属性值，同时为了快速定位服务链式调用中出现的问题，添加调用的服务的数组属性。引入Java对象的编解码技术主要是解决网络传输数据数据量过大的问题。通过对自定义协议的请求信息和响应信息进行二进制编码减小网络数据的传输。引入kubernetes的新特性拓扑感知服务路由，优化Service间请求的转发路径，将Client的请求转发给最近的Server端，减少网路数据传输的路径，缩短请求响应的时间。

6.2 展望

物联网技术是智慧城市的支撑技术,同时物联网技术也在云计算技术的推动下也在不断向前。智慧管廊作为智慧城市项目中比较重要的项目之一，在发展前景和技术支持是毋庸置疑的。对智慧管廊系统发展的展望有如下几点:

⑴ 接入边缘计算模块: 现在智慧管廊系统的传感器数据是通过DeviceAccess 物接入模块接入到云端，在云端做数据处理。数据上传到云端，由于网络延迟的原因，不能及时处理一些突发事件。引入边缘计算之后，数据在边缘端就可以处理，将处理后的结果上传到云端，同时对一些突发事件在边缘端就可以处理，也就是把现在云端的处理能力下放到边缘端。

⑵ 升级到Service Mesh平台: 在自定义协议的协议头中保留了做流量控制的字段，为了兼容Service Mesh 平台的流量控制功能。Service Mesh将流量管理从Kubernetes中解耦，在Service Mesh 内部的流量无需kube-proxy组件的支持，通过更接近微服务应用层的抽象，管理服务间的流量、安全性和可观察性。升级到Service Mesh平台之后，系统升级和发布更容易实现。

对于以上智慧管廊系统的发展的两个方向，在设计智慧管廊中间件时已经做了兼容，但是为了适应新的发展需要还可以在以下的方面做提升：

⑴ 提升长连接的稳定性适用边缘计算场景: 在边缘计算的场景下，边缘端不像云服务器一样有固定的公网IP，云端访问边缘端需要通过边缘端主动和云端建立的长连接。华为的边缘计算框架Kubeedge在边缘端重写了Kubernetes节点的组件的代码,比如kubelet，这样做必然会耗费大量的人力去维护重写的代码，比如将社区kubernetes的新功能迁移到Kubeedge。拓展长连接的功能使其具备Proxy的功能，将云端的请求通过长连接转发到边缘端即可，关于这个长连接的Proxy功能是我在腾讯云实习做的主要工作。

⑵ 基于自定协议做流量管控: 在智慧管廊系统中服务注册和发现模块的负载均衡使用的是独立的loadblance，流量转发通过kube-proxy在节点写入 iptables 或 ipvs 规则来实现的。Service Mesh 平台解耦了流量控制，流量转发是基于sidecar模式实现，在sidecar模块做流量管控让智慧管廊系统更好的接入Service Mesh平台。流量管控开发的方向是支持智慧管廊系统的滚动升级，以及做集群中开发模块和线上模块的隔离。

致谢

参考文献

[1].ReenskaugT M H.Thing-model-view-editor:  an example from a planning system

[C]// Xerox PARC technical note. 1979.

[2] .W. Roy Schulte, Yefim V. Natis. "Service Oriented" Architectures, Part 1[J]. 1996.

[3]. Rademacher F, Sachweh S, Zündorf A. Differences between Model-Driven Development of Service-Oriented and Microservice Architecture[C]// IEEE International Conference on Software Architecture Workshops. IEEE, 2017:38-45.

[4]. Newman S. Building Microservices[M]. Loukides and B. MacDonald, Eds.O'Reilly Media, Inc. 2015

[5]. James Lewis.Micro services: Java, the Unix Way. 33rd Degree Conference . 2012.3

[6]. 朱凯.特色农牧电子商务平台设计与实现.兰州大学.2019

[7]. 周逸天.基于可穿戴的人体姿势识别研究与系统设计.南京邮电大学.2019

[8].Martin Fowler.Microservices[EB/OL]. https://martinfowler.com/articles/microservices.html, 2014.3

[9]. 刘汝滕.一种轻量级分布式RPC框架的研究与实现[D].东南大学.2017

[10]. 李林锋. Netty权威指南(第2版) [M]. 北京: 电子工业出版社, 2015.

[11]. 龚正.Kubernetes权威指南[M]. 北京:电子工业出版社，2017 [14]. 夏江涛.交易风险监控系统的设计与实现.东南大学.2015

[12]. 王飞. 基于微服务的微电网数据分析系统研究与开发.华北电力大学(北京).2019