

**本科毕业设计（论文）**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **题目** |  | | | |
| **基于TCP的远程调用RPC框架研究** | | | | |
| **学生姓名** | | **郑剑** | **学 号** | **202031070390** |
| **教学院系** | | **计算机科学学院** | | |
| **专业年级** | | **软件工程2020级** | | |
| **指导教师** | | **肖斌** | **职 称** | **教授** |
| **单 位** | | **西南石油大学** | | |
| **辅导教师** | |  | **职 称** |  |
| **单 位** | |  | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **完成日期** | **2023** | **年** | **5** | **月** | **28** | **日** |

Southwest Petroleum University

Graduation Thesis



The Design And Implemention of Bank Seckill System Server-end Based on Microservices

**Grade: 2020**

**Name: Zheng jian**

**Speciality: Software Engineering**

**Instructor: Xiao Bin**

**School of Computer Science**

**2023-5**

**摘要**

互联网已然成为人们生活中不可或缺的一部分。包括中小银行在内的各行各业正如火如荼地开展数字化转型，其中“秒杀”这一促销手段逐渐成为了开辟线上市场的重要形式，通过刺激用户购买心理从而为创收与客源获取创造了有利条件。而传统的秒杀系统囿于单体式架构带来的诸多缺陷，已难以满足日益增长的业务需求。

因此，针对某银行推出存款产品上线并投放秒杀活动的需求，本文设计并实现了一套基于微服务的银行秒杀系统服务端，提供用户服务、商品服务、筛查服务、秒杀服务、订单服务和支付服务，为管理系统和客户端提供支撑。系统应用微服务架构，采用Spring Cloud作为项目框架，将Consul作为服务的注册中心，各服务之间通过Dubbo或Spring Cloud OpenFeign完成服务调用，并引入Spring Cloud Gateway服务网关统一处理来自客户端的请求。系统重点解决了秒杀业务的相关性能与安全问题，如商品超卖、链接暴露等，采用Redis缓存热点数据提高系统的响应速度，并借助RabbitMQ消息中间件完成流量削峰和应用解耦。经测试，本系统能够保障业务正常、有序地开展，并兼具可扩展性、高可用性，为客户带来良好的使用体验。

**关键词：**微服务；远程调用；计算机网络

**Abstract**

The internet has become an indispensable part of our daily existence. Various industries, including small and medium-sized banks, are enthusiastically embracing digital transformation. Among them, "flash sale" has gradually become an important form of online market expansion, creating favorable conditions for revenue generation and customer acquisition by tapping into users' buying psychology. However, traditional flash sale systems, constrained by the numerous shortcomings of monolithic architecture, have become increasingly unable to meet the growing business demands.

Therefore, in response to the demand for launching deposit products online and conducting flash sale promotions for a bank, this paper designs and implements the server-side of a microservice-based bank seckill system, which provides user service, product service, screening service, seckill service, order service and payment service, to support the front-end interface of a management system and client-side functionalities. The system employs a microservice architecture, utilizing Spring Cloud as the project framework, Consul as the service registry center, and Dubbo or Spring Cloud OpenFeign for inter-service communication among microservices. Additionally, Spring Cloud Gateway is introduced as a service gateway to handle requests from clients in a unified manner. The system focuses on addressing the performance and security issues related to the flash sale business, such as product overselling and link exposure. It utilizes Redis caching to improve system response speed by caching hot data, and leverages RabbitMQ message middleware for traffic peak shaving and application decoupling. Through testing, it has been proven that this system guarantees seamless and well-organized execution of business activities, while also providing scalability and high availability, thus providing customers with a superior user experience.

Key words: Microservices; Seckill System; Spring Cloud

**目录**

# 绪论

## 研究背景及意义

随着中国互联网行业的迅速发展，分布式系统在各个领域得到了广泛应用，网络节点之间的高效通信和远程方法调用变得尤为重要。在这一背景下，RPC作为一种在分布式系统中应用广泛的通信协议，发挥着至关重要的作用。本论文旨在深入研究并设计一种基于TCP的RPC框架，以更好地满足中国互联网环境下的需求和挑战。

在中国互联网生态中，我面临着不断增长的用户规模、多样化的应用场景以及对通信效率和服务可靠性的高要求。因此，本研究将专注于设计一个基于TCP的RPC系统，旨在提供高效的分布式通信、实现简单且可扩展的服务调用、确保高可靠性和容错性，并推动RPC技术在中国互联网行业的广泛应用。

通过实际实现基于TCP的RPC系统，我期望能够为中国互联网应用提供更优越的解决方案，提升分布式系统的通信效率、可靠性和可扩展性。这将有助于促进中国互联网技术的进步，推动数据驱动的科学研究、促进商业应用的创新，为社会发展提供更强有力的支持。

**选题意义：**

推动分布式系统技术的进步： 通过深入研究并实现基于TCP的RPC框架，我们将为分布式系统技术的发展做出贡献。提升通信效率、服务可靠性和可扩展性将推动分布式系统的发展，并在学术界引领相关领域的研究。

满足中国互联网环境下的需求： 中国互联网用户规模巨大，应用场景丰富多样。我们的研究旨在设计适应中国互联网环境的基于TCP的RPC系统，以满足不断增长的用户需求、提高服务质量，并促进互联网产业的创新与发展。

促进数据驱动的科学研究： 通过提高分布式系统通信效率，我们有望为数据驱动的科学研究提供更强有力的支持。高效的RPC框架将有助于更快地处理大规模数据，促进科学家们在各领域的研究成果。

支持商业应用的创新： 我们的研究成果将为商业应用领域提供技术支持，帮助企业更好地构建可靠的分布式系统，推动商业服务的创新和发展。

服务社会发展的需要： 高可靠性和容错性是分布式系统的重要特征，对于社会发展至关重要。我们的研究将有助于构建更稳定、可靠的互联网基础设施，支持社会各个领域的数字化转型。

## 国内外研究现状

### 国外研究现状

远程过程调用（Remote Procedure Call，RPC）的发展历史可以追溯到20世纪80年代。

在1984年，Bruce Jay Nelson在他的博士论文《The Design and Implementation of a High-Performance Transport Protocol》中首次提出了远程过程调用（Remote Procedure Call，RPC）的概念[1]中将远程过程调用（RPC）定义为一种协议，它允许一个程序能够导致另一台主机上的程序执行过程，而执行过程的环境看起来就像是本地过程调用。RPC抽象了底层的网络通信细节，使得开发分布式系统时，远程通信可以像本地函数调用一样直观。在论文中，Nelson详细讨论了RPC的设计目标，它旨在提供一种简单、高效、通用的远程交互机制。他强调了RPC在分布式计算中的重要性，尤其是在提高网络服务效率和简化复杂网络协议栈时的作用。Nelson还探讨了RPC协议的各种实现细节，包括参数传递、错误处理、传输协议选择等方面。

1985年：Andrew Birrell和Bruce Jay Nelson在Xerox PARC（帕克研究中心）开发了一种名为“Courier”的RPC系统。这是RPC的第一个实现，它使用了一种基于消息传递的通信协议。

1991年：Sun Microsystems发布了NFS（Network File System），这是一种基于RPC的分布式文件系统。NFS的成功推动了RPC的发展，并促使其他公司和组织开始关注RPC技术。

1995年：Microsoft发布了DCOM（Distributed Component Object Model），这是一种基于RPC的分布式对象通信技术。DCOM在Windows平台上广泛应用，为RPC的发展做出了重要贡献。

2000年：Google发布了GFS（Google File System），这是一种基于RPC的分布式文件系统。GFS的成功启发了Google开发更多基于RPC的分布式系统，如MapReduce和Bigtable。

2006年：Facebook发布了Thrift，这是一种开源的RPC框架。Thrift支持多种编程语言，并提供了强大的代码生成工具，使得开发者可以轻松地定义和使用RPC接口。

2010年：Google发布了gRPC，这是一种高性能的RPC框架。gRPC使用了基于HTTP/2的协议，并支持多种编程语言。它在性能和可扩展性方面具有显著优势，被广泛应用于云计算和微服务架构。

上述的thrift和gRPC也是现在行业内使用的最多的远程调用框架。国内的很多研发的RPC框架也是学习国外优秀开源框架的思想，结合内部业务需求进行开发的。

综上所述，自从RPC思想提出到如今RPC的身影无处不在，期间经历了20多年的发展，国外顶尖互联网公司和专家非常重视RPC这项技术， RPC技术在互联网领域的应用和发展已经取得了显著的成果，并且在未来仍有巨大的发展潜力。无论是对于构建高性能、低延迟的互联网应用，还是对于构建复杂、多样化的互联网应用，RPC都将继续发挥着重要的作用。

### 国内研究现状

随着互联网产业的快速发展，中国的互联网公司面临着越来越复杂的业务需求和技术挑战。为了解决这些问题，许多公司开始研究和应用RPC技术，以实现服务化架构和微服务架构。这种趋势推动了RPC技术的研究和应用在中国的快速发展。

2008年：阿里巴巴开始开发Dubbo框架，这是中国最早的开源RPC框架之一。Dubbo引入了服务治理的概念，并提供了丰富的服务治理功能。

2010年：Dubbo框架开源，成为中国最受欢迎的RPC框架之一。许多互联网公司开始使用Dubbo构建自己的服务化架构。

Dubbo对中国互联网的发展产生了深远的影响，推动了服务化架构和微服务架构的发展，提高了系统的性能和可靠性，并推动了开源社区的发展。未来，随着技术的不断进步和应用场景的不断扩展，Dubbo在中国的互联网产业中的影响将更加广泛。

随着这么一个现象级的产品出现，国内各个公司都开始在开源框架的思想上，开发复合自己公司业务的rpc框架，2012年：Motan框架开源，提供了更加灵活的服务治理和负载均衡功能。Motan在中国的互联网公司中得到了广泛的应用。

2014年：SofaRPC框架开源，提供了更加高效的RPC通信和服务治理功能。SofaRPC在中国的互联网公司中得到了广泛的应用。

综上所述，国内的RPC框架以阿里的dubbo为一个节点，后续各种rpc框架运行而生，有服务于自己公司业务的rpc框架，也有服务于大众开发程序员的开源框架，这些框架都为国内RPC技术做出了发展。

## 本文主要工作和结构安排

### 主要工作

本课题在基于TCP通信的网络上，旨在设计并实现一个RPC远程调用系统。本人具体工作内容如下：

1. 分析并实现RPC服务端和客户端，并在分布式的环境下可以正常进行远程调用。
2. 重点调研分析RPC框架为完成通信需要的技术，并分析各项技术的可行性和实用性。
3. 重点抽象RPC各项关键原理，包括序列化、协议、注册中心、网络连接的管理等实现。
4. 重点研究服务的健康检查、负载均衡、异常重试等机制；
5. 搭建必要环境进行开发，交付程序并通过测试验证系统的表现是否达到预期。

### 结构安排

全文共分为7章，各章安排如下：

第1章：绪论。本章首先介绍了课题的研究背景与意义，其次介绍了国内外RPC框架的研究现状，最后给出了本文的主要工作和结构安排。

第2章：相关理论与技术。本章阐述了RPC框架涉及到的理论与技术，介绍了项目采用的各种协议和总体框架。

第3章：RPC框架的分析。本章首先论证了系统的可行性，其次分析了管理员和客户各自的功能需求，描述了秒杀的具体流程，最后对非功能需求作简要分析。

第4章：RPC框架的设计。本章首先提出了基于微服务的服务端软件架构，其次叙述了系统的数据库设计，再从功能模块的划分入手描述了各功能模块设计，最后重点阐述了服务端的性能优化与安全性设计。

第5章：RPC框架的实现。本章主要是在第4章的基础上，阐述了服务端微服务组件和各个功能模块的实现，其中着重介绍了秒杀相关问题的解决思路与实施方案。

第6章：RPC框架的测试。本章的主要内容是使用软件工程的测试方法验收系统。

第7章：总结与展望。本章小结本文的主要内容，展望后续的改进方向。

## 本章小结

本章主要介绍了RPC框架的研究背景与意义，综合目前国内外研究现状进行分析，为后续的课题研究内容和工作安排打好基础。

# 相关理论与技术

## Spring

Spring 是一款开源的轻量级 Java 开发框架，旨在提高开发人员的开发效率以及系统的可维护性。

### spring ioc

IoC（Inversion of Control:控制反转） 是一种设计思想，而不是一个具体的技术实现。IoC 的思想就是将原本在程序中手动创建对象的控制权，交由 Spring 框架来管理。不过， IoC 并非 Spring 特有，在其他语言中也有应用。

为什么叫控制反转？：将对象的创建的权利由开发者转交到spring容器。

将对象之间的相互依赖关系交给 IoC 容器来管理，并由 IoC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。 IoC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。

在实际项目中一个 Service 类可能依赖了很多其他的类，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数。如果利用 IoC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度。

在 Spring 中， IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个 Map（key，value），Map 中存放的是各种对象。

### spring aop

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

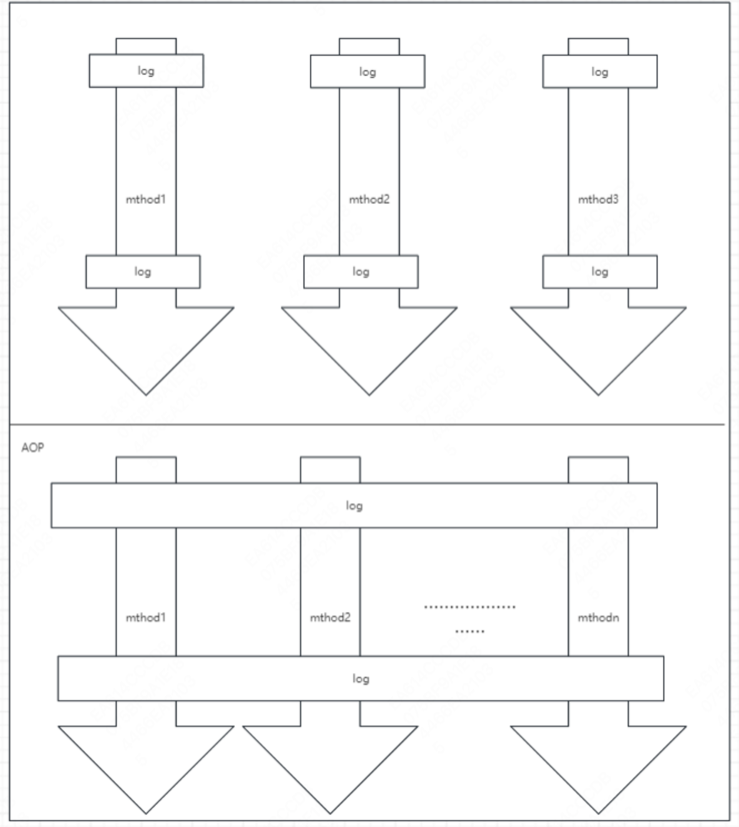
如下图 2-1所示，如果我想在多个方法的前后都要加上日志功能，当方法很多时，对于开发者是很麻烦的，此时我们使用动态代理技术，一次性让AOP对符合要求的多个方法都加上日志控制，就不需要人工去写代码了。

图 2-1

Spring AOP 就是基于动态代理的，如果要代理的对象，实现了某个接口，那么 Spring AOP 会使用 JDK Proxy，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候 Spring AOP 会使用 Cglib 生成一个被代理对象的子类来作为代理。

下面是AOP相关的术语以及概念：

|  |  |
| --- | --- |
| 术语 | 概念 |
| 目标(Target) | 被通知的对象 |
| 代理(Proxy) | 向目标对象应用通知之后创建的代理对象 |
| 连接点(JoinPoint) | 目标对象的所属类中，定义的所有方法均为连接点 |
| 切入点(Pointcut) | 被切面拦截 / 增强的连接点（切入点一定是连接点，连接点不一定是切入点） |
| 通知(Advice) | 增强的逻辑 / 代码，也即拦截到目标对象的连接点之后要做的事情 |
| 切面(Aspect) | 切入点(Pointcut)+通知(Advice) |
| 织入(Weaving) | 将通知应用到目标对象，进而生成代理对象的过程动作 |

Spring 创建代理对象，并根据用户已配置的切入点对方法调用进行拦截，拦截之后根据配置好的通知调整方法调用的时机。若用户配置的是前置通知，那么在方法调用之前先调用前置通知。反之，若用户配置的后置通知，则在方法调用过后执行后置通知。这个过程将切面应用到目标对象并导致代理对象创建的过程，又称为织入。

其中通知又有不同的类型，不同的通知类型其作用效果不一样：

|  |  |
| --- | --- |
| 通知类型 | 含义 |
| Before（前置通知） | 目标对象的方法调用之前触发 |
| After（后置通知） | 目标对象的方法调用之后触发 |
| AfterReturning（返回通知） | 目标对象的方法调用完成，在返回结果值之后触发 |
| AfterThrowing（异常通知） | 目标对象的方法运行中抛出 / 触发异常后触发。 |
| Around （环绕通知） | 编程式控制目标对象的方法调用 |

其中环绕通知是所有通知类型中可操作范围最大的一种，因为它可以直接拿到目标对象，以及要执行的方法，所以环绕通知可以任意的在目标对象的方法调用前后搞事，甚至不调用目标对象的方法。

本次开发，客户端就会使用AOP功能实现对客户端实现细节的隐藏。

## 网络通信

### 2.2.1 TCP与UDP

TCP（传输控制协议）和UDP（用户数据报协议）是互联网协议套件中的两个重要协议，用于在计算机网络中传输数据。

TCP是一种面向连接的协议，它提供可靠的数据传输。在使用TCP时，数据被分割成小的数据包，并通过网络传输到目标设备。TCP确保数据包的顺序和完整性，如果数据包丢失或损坏，TCP会自动重新发送丢失的数据包。此外，TCP还通过使用确认和超时机制来确保可靠的传输。由于TCP的可靠性和连接性，它通常用于需要可靠数据传输的应用程序，如文件传输、电子邮件和网页浏览。

UDP是一种无连接的协议，它提供了一种简单的数据传输方式。与TCP不同，UDP不保证数据的可靠性。在使用UDP时，数据被分割成小的数据包，并通过网络传输到目标设备。UDP不提供确认、重传或流控制等机制，因此数据包的丢失或损坏将不会被修复。UDP的优势在于它具有较低的延迟和较小的传输开销，适用于实时应用程序，如音频和视频流媒体、在线游戏和实时通信。

本课题RPC底层的网络通信将使用TCP，原因有以下几点：

①可靠性。TCP是一种可靠的协议，它提供了数据包的可靠传输。在RPC通信中，可靠性是非常重要的，因为我们希望确保远程调用的请求和响应能够可靠地传输，并且不会丢失或损坏。TCP通过序列号、确认和重传机制等技术来保证数据的可靠性，确保数据的完整性和顺序。

​ ②顺序性。RPC框架通常需要保证请求和响应的顺序。TCP提供了有序的数据传输，即发送的数据包将按照发送的顺序在接收端被按序接收。这对于确保RPC调用的正确执行非常重要，因为如果请求和响应的顺序混乱，可能导致执行结果不一致或产生错误。

​ ③连接管理。TCP是一种面向连接的协议，它使用握手和断开连接的机制来管理通信双方之间的连接状态。在RPC框架中，连接管理对于建立和维护节点之间的通信非常重要。TCP提供了可靠的连接建立和断开过程，可以确保通信双方在进行RPC调用之前建立有效的连接，并在通信完成后正确地关闭连接。

​ ④兼容性。TCP是互联网最常用的传输协议之一，几乎所有的网络设备和操作系统都支持TCP。使用TCP作为底层传输协议可以保证RPC框架的兼容性和互操作性，使得不同平台和环境下的节点能够进行可靠的通信。

### 2.2.2 Java bio

以前大多数网络通信方式都是阻塞模式，如图 2-2所示：

客户端向服务器端发出请求后，客户端会一直等待(不会再做其他事情)，直到服务器端返回结果或者网络出现问题。

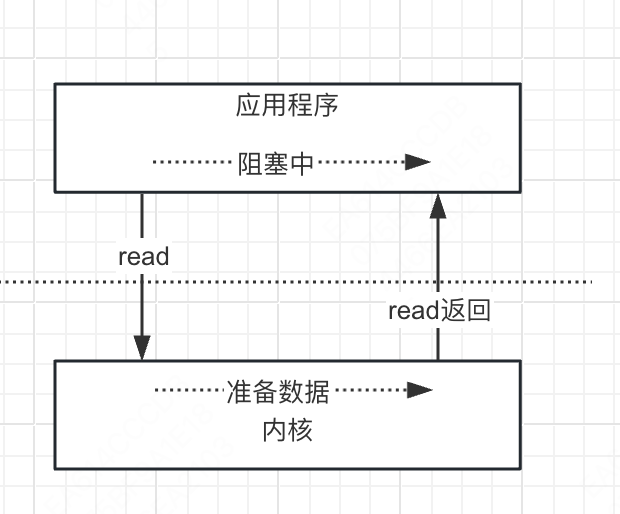
服务器端同样的，当在处理某个客户端A发来的请求时，另一个客户端B发来的请求会等待，直到服务器端的这个处理线程完成上一个处理。

图 2-2

这种基于BIO的网络传输有如下的问题：

同一时间，服务器只能接受来自于客户端A的请求信息；虽然客户端A和客户端B的请求是同时进行的，但客户端B发送的请求信息只能等到服务器接受完A的请求数据后，才能被接受。

由于服务器一次只能处理一个客户端请求，当处理完成并返回后(或者异常时)，才能进行第二次请求的处理。很显然，这样的处理方式在高并发的情况下，是不能采用的。

本次RPC框架场景适用在高并发的场景下，BIO不能解决RPC的通信问题。

### 2.2.3 Java NIO

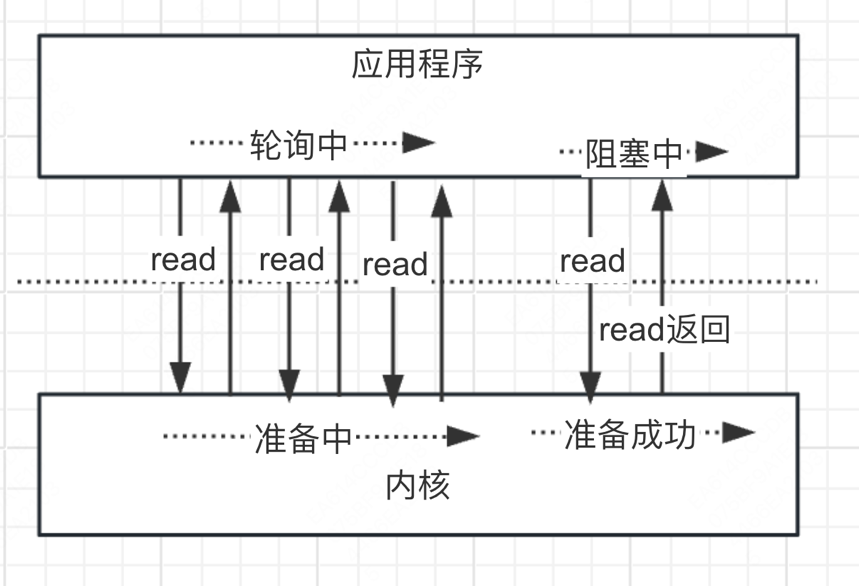
NIO（Non-blocking I/O）是一种非阻塞I/O模型，用于处理输入和输出操作。相比于BIO（Blocking I/O）模型，NIO提供了更高效的I/O处理方式，如，支持处理多个并发任务。如图 2-3所示，应用程序会不断轮询read内核是否准备好数据，不会阻塞，当内核准备好数据后，应用程序就会阻塞read内核数据。

图 2-3

以下是NIO模型的一些主要特性：

①非阻塞：NIO是一种非阻塞的I/O模型，它使用了非阻塞I/O操作和事件驱动机制。当应用程序进行I/O操作时，不会被阻塞等待操作完成，而是立即返回并继续执行其他任务。通过使用非阻塞I/O，应用程序可以利用空闲时间执行其他任务，从而提高系统的并发处理能力。

②缓冲区（Buffer）：NIO模型使用缓冲区来存储数据。缓冲区是一块连续的内存区域，用于在应用程序和底层I/O通道之间传输数据。应用程序可以将数据写入缓冲区，然后通过I/O通道进行读取或将数据从I/O通道写入缓冲区。这种基于缓冲区的数据传输方式可以减少系统的数据拷贝次数，提高数据传输的效率。

③通道（Channel）：NIO模型使用通道进行数据的读取和写入。通道代表了一个打开的I/O连接，可以通过通道进行数据的读取和写入操作。与传统的流式I/O不同，通道可以是双向的，支持同时进行读取和写入操作。通过通道，应用程序可以直接与操作系统内核进行交互，实现高效的数据传输。

#### 2.2.3.1 多路复用

虽然NIO有以上的有点，但是因为其非阻塞状态会不断轮询read内核，这势必会造成CPU的高负载，为了解决这个问题，后面就提出了多路复用，让一个线程去监听多个网络事件。可以最大程度上减轻CPU的负载。

多路复用（Multiplexing）：NIO模型支持多路复用的I/O操作。多路复用是一种同时监听多个I/O通道状态的机制，通过一个线程来管理多个通道的I/O操作。通过使用选择器（Selector）来实现多路复用，应用程序可以同时处理多个通道的I/O事件，从而提高系统的并发性能。IO 多路复用模型中，线程首先发起 select 调用，询问内核数据是否准备就绪，等内核把数据准备好了，用户线程再发起 read 调用。read 调用的过程（数据从内核空间 -> 用户空间）还是阻塞的。IO 多路复用模型，通过减少无效的系统调用，减少了对 CPU 资源的消耗。

在java NIO中，一个重要的组件叫做selector，其专门用来实现多路复用，如图 2-4所示，多个client的channel管道会注册到组角selector上。这时，应用程序只需要使用一个线程调用select方法，就可以同时监听多个channel通道，当一个channel产生事件后，会通知对应的应用程序线程使用read函数读取内核的数据。

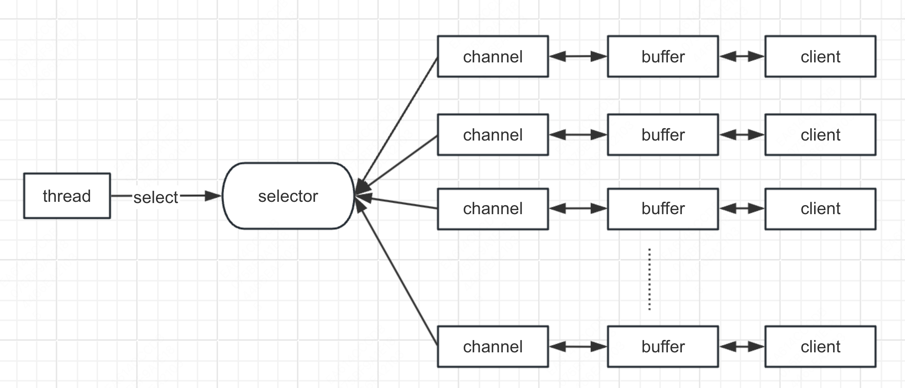


图 2-4

nio的io多路复用模型已经可以基本满足RPC框架的网络传输，相比于传统的阻塞I/O（BIO）模型，NIO模型的编程复杂度更高。它需要开发人员熟悉缓冲区、通道和选择器等概念，并编写更复杂的代码来管理非阻塞I/O操作和多路复用。所以本课题用更加方便的netty网络框架来实现网络通信。

### 2.2.4 netty

Netty是一个基于Java的开源网络应用框架，提供了一组丰富的高性能、异步、事件驱动的网络编程工具和组件。它构建在Java NIO（New I/O）之上，通过提供更高级别的抽象和封装，简化了网络应用的开发。

Netty的设计目标是提供一个可扩展、高性能的网络应用框架，适用于各种类型的网络应用，包括服务器、客户端、中间件和通信协议等。

下面介绍netty中比较重要的组件：

#### 2.2.4.1Channel

- Netty网络通信的组件，用于网络IO操作。

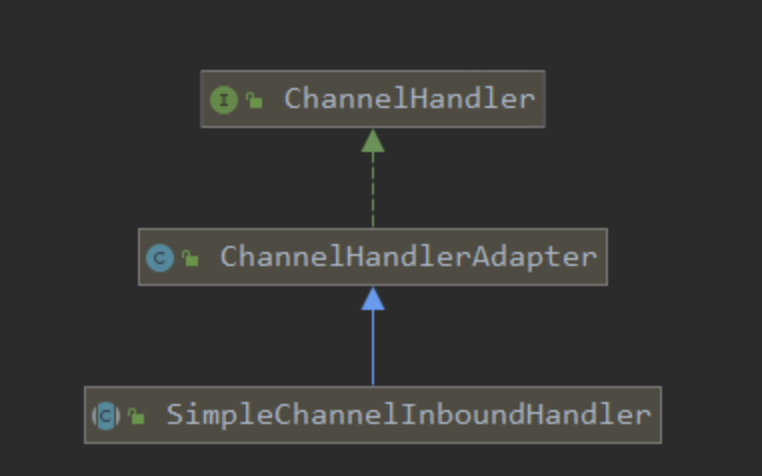
- 通过Channel可以获得当前王略连接的通道的状态与网络配置参数。

- Channel提供异步的网络IO操作，调用后立即返回ChannelFuture，通过注册监听，或者同步等待，最终获取结果。

#### 2.2.4.2Selector

Netty基于java.nio.channels.Selector对象实现IO多路复用，通过Selector一个线程可以监听多个连接的Channel事件。当向一个Selector中注册Channel后，Selector内部的机制就可以自动不断的Select这些注册的Channel是否有就绪的IO事件（可读、可写、网络连接完成等）。

#### 2.2.4.3ChannelHandler

ChannelHandler属于业务的核心接口，用于处理IO事件或者拦截IO操作，并将其转发到ChannelPipeline（业务处理链）中的下一个处理程序。

#### 2.2.4.4 Pipeline与ChannelPipeline

- ChannelPipeline是一个handler的集合，它负责处理和拦截出站和入站的事件和操作。

- ChannelPipeline实现了拦截过滤器模式，使用户能控制事件的处理方式。

- 在Netty中，每个Channel都有且只有一个ChannelPipeline与之对应。如所图 2-5示

一个 Channel 包含了一个 ChannelPipeline，而 ChannelPipeline 中又维护了一个由 ChannelHandlerContext 组成的双向链表，并且每个 ChannelHandlerContext 中又关联着一个 ChannelHandler。

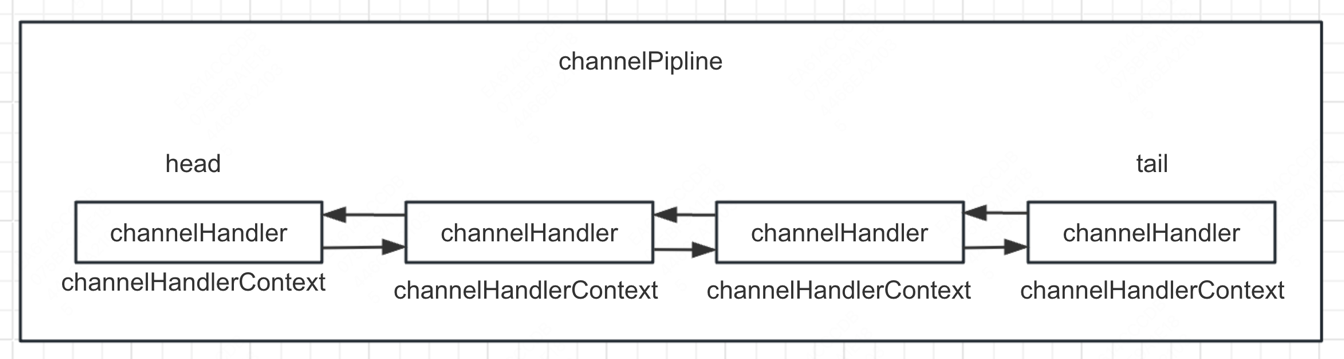


图 2-5

#### 2.2.4.5 LengthFieldBasedFrameDecoder

LengthFieldBasedFrameDecoder是Netty框架提供的一个用于解决TCP粘包和拆包问题的解码器（Decoder）。它基于长度字段的方式来解决数据的分割和重组。

在网络通信中，由于传输过程中的各种原因，数据往往会被拆分成多个小片段进行传输，或者多个数据包会被合并成一个大的数据块。这就导致了粘包（多个数据包黏在一起）和拆包（一个数据包被拆分成多个片段）的问题。

LengthFieldBasedFrameDecoder通过读取指定长度字段（通常是包头中的长度字段）的值，来确定每个数据包的长度，从而正确地将接收到的数据拆分成独立的数据包。

本课题的协议设计部分，就需要用到这个重要的组件。

## zookeeper

### 2.3.1 Zookeeper简介

ZooKeeper是一个开源的分布式协调服务，由Apache软件基金会开发和维护。它提供了一个简单而健壮的分布式应用程序协调和管理的解决方案。

ZooKeeper的设计目标是为分布式系统提供高可用性、一致性和可靠性的协调服务。它通过提供一个分布式的、层次化的命名空间（类似于文件系统），来存储和管理配置信息、状态信息、命名服务等。

本课题将使用zookeeper来作为注册中心，依赖于其文件存储+监听机制。

### 2.3.2 Zookeeper节点存储

zookeeper中的文件存储以znode节点的形式存在。

|  |  |
| --- | --- |
| 临时目录节点 | 客户端与Zookeeper断开连接后，该节点被删除 |
| 临时顺序编号目录节点 | 基本特性同临时节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。 |
| 持久化目录节点 | 客户端与Zookeeper断开连接后，该节点依旧存在 |
| 持久化顺序编号目录节点 | 基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。 |

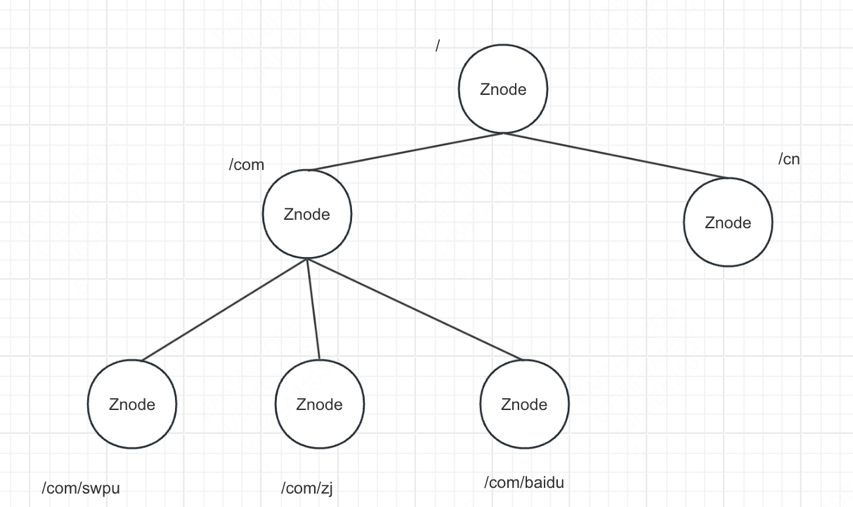
ZooKeeper的数据结构，跟Unix文件系统非常类似，可以看做是一颗树，每个节点叫做Znode。每一个Znode只能存1MB数据。数据只是配置信息。每一个节点可以通过路径来标识，结构图如图 2-6 ：

图 2-6

### 2.3.3 zookeeper一致性原理

ZooKeeper通过ZAB（ZooKeeper Atomic Broadcast）协议来实现分布式数据一致性。该协议保证了在分布式环境下的数据更新操作按照相同的顺序被广播和应用，从而实现了数据的一致性。

ZAB协议的一致性原理如下：

1. 选举阶段（Leader Election）：ZooKeeper集群中的每个节点可以是一个Leader、一个Follower或一个Observer。在初始启动阶段或Leader节点失效时，ZooKeeper会触发选举过程。选举过程中，节点会相互通信并比较自己的ZXID（每个事务操作都有一个唯一的ZXID标识）来确定新的Leader。选举过程保证了只有一个Leader节点存在，并且其他节点会成为Follower或Observer。

2. 数据广播阶段（Data Broadcast）：一旦新的Leader节点选举出来，它会负责处理客户端的请求和更新操作。当Leader节点接收到一个更新请求时，它会将该请求作为事务记录到本地的事务日志中，并将该事务广播给其他节点。

3. 事务应用阶段（Transaction Application）：一旦Follower节点接收到Leader节点广播的事务请求，它会将该事务记录到本地的事务日志中，并在日志中标记为已提交。然后，Follower节点会将确认消息发送给Leader节点，表示已经成功接收并应用了该事务。

4. 数据同步阶段（Data Sync）：Leader节点在接收到大多数（大部分节点中的大多数节点）Follower节点的确认消息后，会将该事务标记为已提交。然后，Leader节点会将已提交的事务广播给所有的Follower节点，确保它们也应用了相同的事务。

通过以上的过程，ZAB协议保证了分布式环境下的数据一致性。每个更新操作都会被顺序地记录、广播和应用，确保所有节点在相同的顺序下执行相同的操作。这样，即使在网络故障、节点故障或网络分区等情况下，ZooKeeper也能保持数据的一致性。

因为要求zookeeper的一致性是强一致性，所以同一时刻所有节点状态一致，这就会牺牲zookeeper的可用性。

## 负载均衡

负载均衡是一种在计算机网络和分布式系统中使用的技术，旨在将工作负载（即请求或流量）均匀地分配给多个服务器、计算资源或其他系统组件，以实现更好的性能、可靠性和可扩展性。

负载均衡的主要目标是避免某些服务器过载而导致性能下降，同时利用所有可用的资源以提高整体系统的吞吐量和响应时间。通过在多个服务器之间分配负载，负载均衡可以实现请求的并行处理、增加系统的容量和可用性，并提供弹性和故障恢复能力。

本次课题客户端调用服务端，就会从多个服务端中通过负载均衡拿到一个合适的具体的服务端机器信息。

## 心跳机制

心跳检测是分布式系统中常用的一种机制，用于检测节点的存活状态和可用性。在RPC（Remote Procedure Call）和其他分布式系统中，心跳检测起着重要的作用，可以确保节点之间的通信正常，并及时处理节点故障。

心跳检测的基本原理是节点定期发送心跳信号（通常是网络消息或请求）给其他节点，以确认节点的存活状态。接收方节点在一定时间内收到心跳信号，则认为发送方节点是存活的；如果在指定时间内未收到心跳信号，就可以判断发送方节点可能出现了故障或不可用。基于心跳检测的结果，系统可以采取相应的措施，如重新分配任务、故障转移等。

心跳检测通常包括以下关键要素：

心跳发送：每个节点定期发送心跳信号给其他节点。心跳信号可以是一个特定的消息，也可以是简单的网络连接请求。

心跳接收：节点接收来自其他节点的心跳信号，并记录下最近接收到的心跳时间。

超时判断：每个节点都有一个超时阈值，用于判断其他节点是否超过了预设的心跳间隔时间。如果超过了超时阈值，就认为节点可能出现故障。

故障处理：一旦节点被判定为故障，系统可以采取相应的处理措施，如剔除其在注册中心的信息。

## 分布式理论

在计算机科学中，分布式系统是一种广泛使用的系统，它由多台计算机组成，这些计算机通过网络连接，共同完成一项任务。然而，分布式系统面临着一些挑战，其中最著名的是CAP理论。

CAP理论，也被称为Brewer's定理，是由计算机科学家Eric Brewer在2000年提出的。CAP理论指出，在一个分布式系统中，最多只能同时满足以下三个属性中的两个：一致性（Consistency）、可用性（Availability）和分区容忍性（Partition tolerance）。

一致性意味着所有的节点在任何时候都能看到相同的数据。可用性意味着对于任何的请求，系统都能在有限的时间内返回一个非错误的响应。分区容忍性意味着系统能在网络分区发生时继续运行。

在实际的分布式系统中，往往需要在这三个属性之间进行权衡。例如，Zookeeper是一个广泛使用的分布式协调服务，它采用了一致性和分区容忍性，但牺牲了可用性。

Zookeeper通过使用一个称为ZAB的协议来保证一致性。ZAB协议是一个基于两阶段提交（2PC）的协议，它保证了在分布式系统中所有的节点都能看到相同的数据。然而，这种方法牺牲了可用性，因为如果Zookeeper的主节点失败，所有的写操作都将被阻塞，直到新的主节点被选举出来。

## 本章小结

本章主要介绍了本框架在各个模块中所用到第三方软件进行介绍。对服务生成时所用到的 Spring 框架，网络通信是所用到的异步通信框架 Netty 框架，处理分布式一致性服务时所用到 ZooKeeper 框架以及其他相关技术进行了概念及原理的介绍。

# RPC框架需求分析

随着系统的不断扩大，垂直应用也得到了广泛的应用。在这种情况下，将系统按照垂直结构进行拆分，各个模块之间的协同工作变得至关重要，从而引发了对RPC框架的需求。本章将从对所研究的RPC框架的业务需求进行分析出发，并在此基础上对其功能性和非功能性两方面的需求进行了详细分析。在完成了对该框架的需求分析之后，将根据分析结果对框架进行设计。

## 业务需求

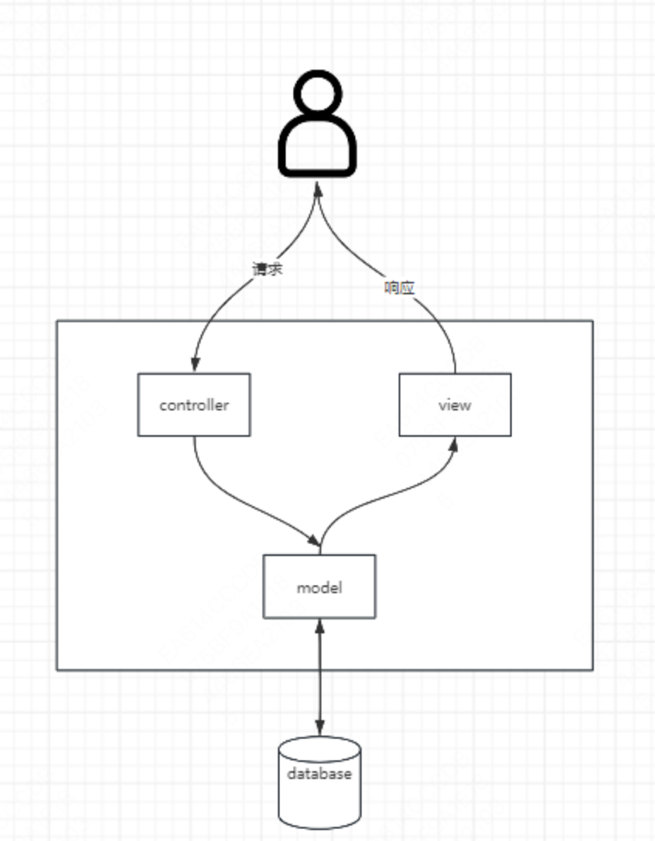
传统的MVC架构集成了业务逻辑耦合度较低的各个业务模块。该架构通过前后端分离技术实现，用户通过前端页面发起请求，并通过代理服务器进行负载均衡。这种工作模型在图3-1中进行了示意。在系统规模较小的情况下，MVC架构能够有效支持系统运行，并且后期维护成本是可控的。然而，随着业务的不断发展，系统规模也不断扩大。在这种情况下，继续使用MVC架构作为系统架构会导致系统难以维护，而且后续版本的迭代成本会变得过高。

图3-1

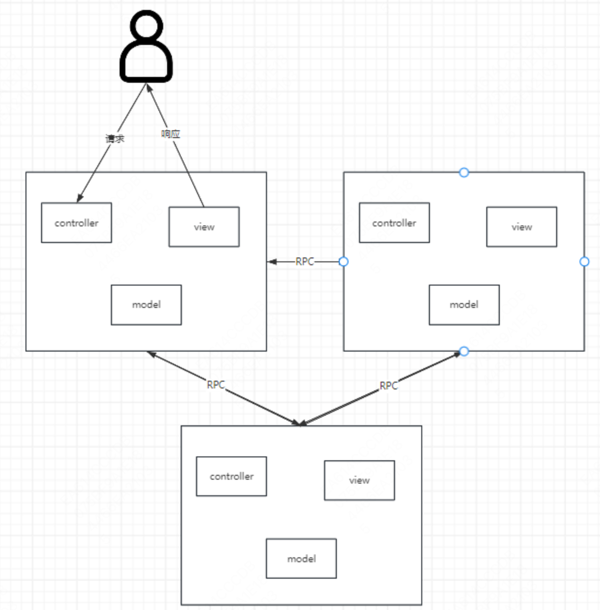
通过对系统进行业务模块的分割，各个模块之间通过协议和网络进行通信，这是解决上述问题的一种方法，也是大型系统所需求的。同时，这种业务需求也是本文研究的RPC框架所关注的。庞大而复杂的系统可以从业务逻辑角度进行分割，分割后的模块通过RPC框架进行通信。分割后的系统工作结构如图3-2所示。在确保系统业务功能的前提下，有效地降低系统维护和迭代的成本。

图3-2

庞大的系统通过将核心业务与公共业务进行垂直拆分，分别作为独立的服务。RPC 框架则需要将公共业务以服务的形式发布，供核心业务模块进行调用。一方面降低系统的复杂性，另一方面提高公共业务的复用性。

因为需要进行网络调用，如何将网络调用优化成如本地调用一般，是rpc设计的重点。

## 功能需求

本项目时基于TCP的RPC框架实现，为分布式微服务中两个服务之间的通信提供可控性服务。目前系统涉及客户端（服务调用方）和服务端（服务提供方）两个角色，以下两节将通过用例设计说明两个角色与系统各功能单元之间的关系。

### 3.2.1 客户端能需求分析

图3-3

如图3-3所示，其中，方法调用的用例包括调用远程方法并得到对应的返回值；请求重试涉及到当方法调用失败后，客户端如何进行的重试策略；服务发现用例指的是客户端需要感知到自己调用服务方的服务列表即ip地址，并需要提供负载均衡策略调用具体的一台机器服务；日志监控用例包括error日志监控和调用次数的监控。

### 3.2.2 服务端功能需求分析

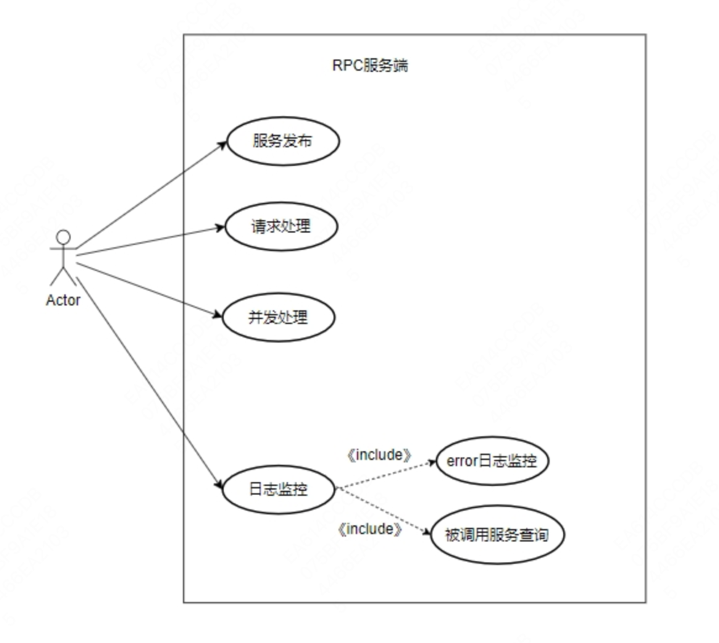
用户使用服务端服务发布、请求处理、并发处理、错误处理、监控和日志功能。

图3-4

如图3-4所示其中，服务发布包括将自己的服务方法签名和机器标识注册到注册中心，请求处理包括处理来自不同客户端发来的方法执行请求，并发处理用于处理同时接收多个请求时的并发安全处理，日志监控包括error日志的处理和使用本服务的客户端列表查询。

## 非功能需求分析

性能效率需求：

RPC框架的性能效率需求主要包括响应时间、吞吐量和资源利用率等方面。响应时间是指从客户端发起请求到服务端返回结果的时间，需要尽可能短以提供流畅的用户体验。吞吐量是指在单位时间内处理的请求数量，需要尽可能高以提供高并发处理能力。资源利用率是指RPC框架在运行过程中对系统资源的使用情况，需要尽可能低以节省资源并提高系统的稳定性和可靠性。

可用性需求

RPC框架的可用性需求主要包括故障恢复能力、容错能力和高可用性等方面。故障恢复能力是指在出现故障时，RPC框架能够快速恢复正常运行，减少服务中断时间。容错能力是指在出现错误或异常时，RPC框架能够正确处理并继续运行，避免系统崩溃。高可用性是指RPC框架能够在长时间内保持正常运行，提供稳定可靠的服务。

安全性需求

RPC框架的安全性需求主要包括数据加密、访问控制和防御攻击等方面。数据加密是指在传输过程中对数据进行加密，保护数据的安全性和隐私性。访问控制是指对服务端的访问进行权限控制，只允许授权用户访问。防御攻击是指对常见的网络攻击进行防御，如拒绝服务攻击、跨站脚本攻击等。

可扩展性需求

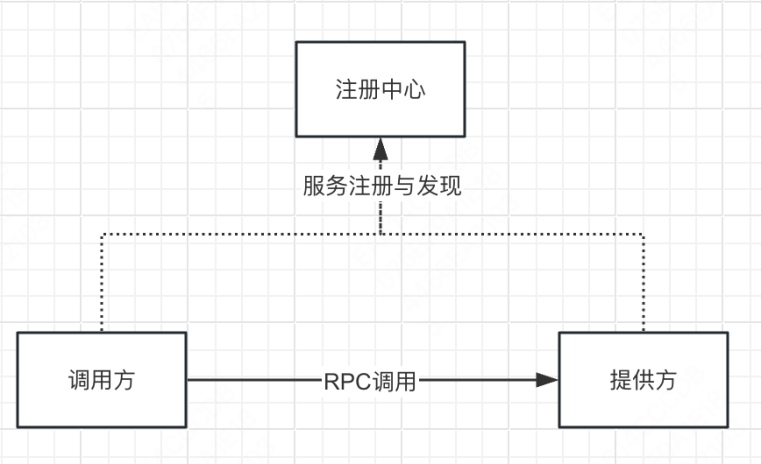
RPC框架的可扩展性需求主要包括横向扩展、纵向扩展和服务发现等方面。横向扩展是指通过增加服务器数量来提高系统的处理能力，需要RPC框架支持负载均衡和服务发现。纵向扩展是指通过增加服务器的性能来提高系统的处理能力，需要RPC框架支持服务端的性能优化和资源管理。服务发现是指在服务端有多个实例时，客户端能够自动发现并选择合适的服务端进行请求。

其他非功能需求

除了上述需求外，RPC框架还需要考虑其他非功能性需求，如易用性、可维护性和可测试性等。易用性是指RPC框架的使用方便，易于上手和操作。可维护性是指RPC框架的代码结构清晰，易于维护和升级。可测试性是指RPC框架支持单元测试、集成测试和性能测试等，以确保系统的质量和稳定性。

## RPC总体结构

使用RPC进行开发时，一般开发人员调用远程接口，一般不需要关注对方的实现细节，只需要提供好接口，调用方像本地方法一样调用结果就可以获取到对应的返回值，所以对于开发者来说，只需要关注调服务调用方和服务提供方。但是另外需要注意的一点，调用方要通过何种方式知道我们服务提供方的机器ip地址，这时候，其实底层通过注册中心帮我们隐藏了具体的细节，多数框架，例如dubbo，可以通过服务名，将统一服务名对应的对应的多个ip地址，再通过负载均衡机制，找到一个具体的服务提供方进行调用。

纵观大多数RPC服务框架，例如thrift、grpc、dubbo等，基本的架构可以抽象为下图：

服务调用方：服务调用方是需要使用某个服务的客户端。当服务调用方需要使用某个服务时，它会向注册中心查询服务提供方的地址，并通过RPC调用服务提供方提供的服务。服务调用方通常会将请求数据传递给服务提供方，并接收服务提供方返回的响应数据。

服务提供方：服务提供方是提供服务的服务器。服务提供方会向注册中心注册自己提供的服务，并监听RPC调用请求。当服务提供方收到服务调用方的请求时，它会处理请求数据，并返回响应数据给服务调用方。

注册中心：注册中心是一个中心化的服务发现和注册系统。服务提供方会向注册中心注册自己提供的服务，并定期更新自己的状态信息。服务调用方在需要使用某个服务时，会向注册中心查询服务提供方的地址，并获取服务提供方的地址信息。注册中心通常会提供一个API，供服务提供方和服务调用方进行注册和查询操作。

## 本章小结

本章首先从可行性分析入手叙述了系统在技术、操作和经济层面的可行性，其次分别对RPC框架涉及的几个服务端和客户端进行需求分析，再在此基础上分析了RPC方法调用的流程，最后简要分析了系统的非功能需求，包括性能效率、可用性、安全性和可扩展性等。

# RPC框架设计

## 总体设计

使用RPC进行开发时，一般开发人员调用远程接口，一般不需要关注对方的实现细节，只需要提供好接口，调用方像本地方法一样调用结果就可以获取到对应的返回值，所以对于开发者来说，只需要关注调服务调用方和服务提供方。但是另外需要注意的一点，调用方要通过何种方式知道我们服务提供方的机器ip地址，这时候，其实底层通过注册中心帮我们隐藏了具体的细节，多数框架，例如dubbo，可以通过服务名，将统一服务名对应的对应的多个ip地址，再通过负载均衡机制，找到一个具体的服务提供方进行调用。

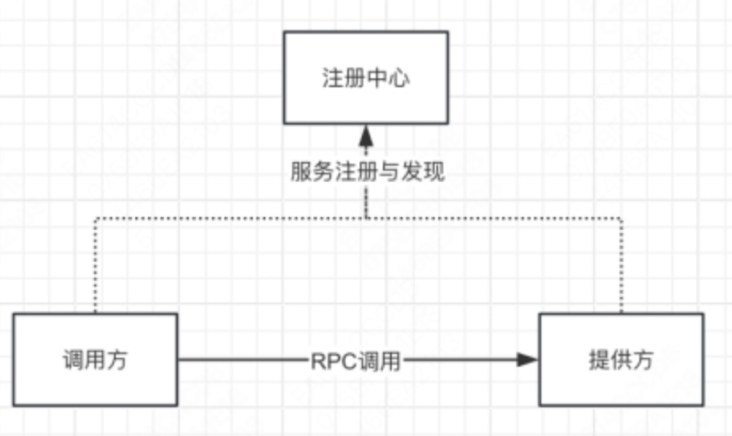
纵观大多数RPC服务框架，例如thrift、grpc、dubbo等，基本的架构可以抽象为如图4-1所示结构：

图4-1

服务调用方：服务调用方是需要使用某个服务的客户端。当服务调用方需要使用某个服务时，它会向注册中心查询服务提供方的地址，并通过RPC调用服务提供方提供的服务。服务调用方通常会将请求数据传递给服务提供方，并接收服务提供方返回的响应数据。

服务提供方：服务提供方是提供服务的服务器。服务提供方会向注册中心注册自己提供的服务，并监听RPC调用请求。当服务提供方收到服务调用方的请求时，它会处理请求数据，并返回响应数据给服务调用方。

注册中心：注册中心是一个中心化的服务发现和注册系统。服务提供方会向注册中心注册自己提供的服务，并定期更新自己的状态信息。服务调用方在需要使用某个服务时，会向注册中心查询服务提供方的地址，并获取服务提供方的地址信息。注册中心通常会提供一个API，供服务提供方和服务调用方进行注册和查询操作。

## 客户端与服务端设计

调用方客户端调用方法就像处理本地方法一样，但是底层原理实际上是通过动态代理的技术，框架生成一个代理类，在代理类里面会处理对应的序列化、压缩、协议封装等逻辑，最终通过sockets连接传输到对应的服务端。

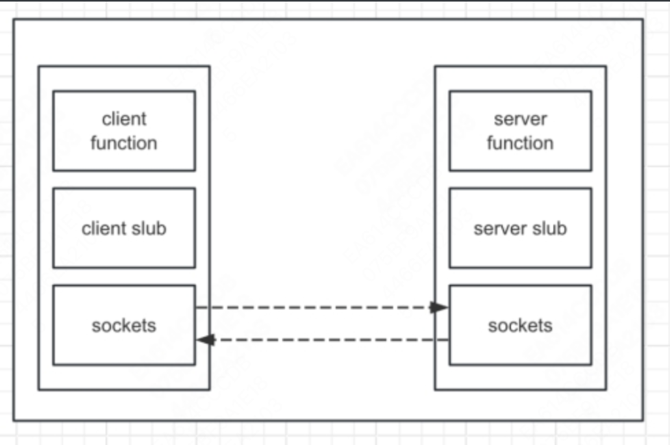
如图4-2所示，服务提供方这边会有一个监听服务监听sockets通道，当有数据到达时，socket将对应的二进制数据传输到server slub，在其中进行反序列化、协议解析，并通过反射技术调用服务端对应的方法进行执行，完成后将结果通过原路径返回给服务调用方即可。

图4-2

## 客户端调用服务端流程设计

RPC框架的核心是客户端成功调用服务端的服务。根据系统（图4-3）的不同职责划分为server、server slub、注册中心、client和client slub五个泳道。一次rpc的调用成功，需要依次经历以下步骤：

（1） client调用接口方法，client slub生成代理对象

（2） client slub根据配置从注册中心获取调用目标机器列表。

（3） client slub使用负载均衡功能获取到具体的机器。

（4） client slub 封装请求体，并进行序列化、压缩、协议encode等后进行TCP网络传输。

（5） server slub接受请求后，对请求参数进行协议decode、解压、反序列化等操作后，封装对应的rpc请求体。

（6） server slub 根据rpc请求体中的请求签名，利用反射技术，拿到server的反射对象，并调用具体方法的执行，最终拿到执行结果

（7） server slub根据执行结果，封装rpc响应，并进行序列化、压缩、协议encode，最后通过TCP返回给client slub。

（8） client 通过协议decode、解压、反序列化等操作后，返回数据给client。

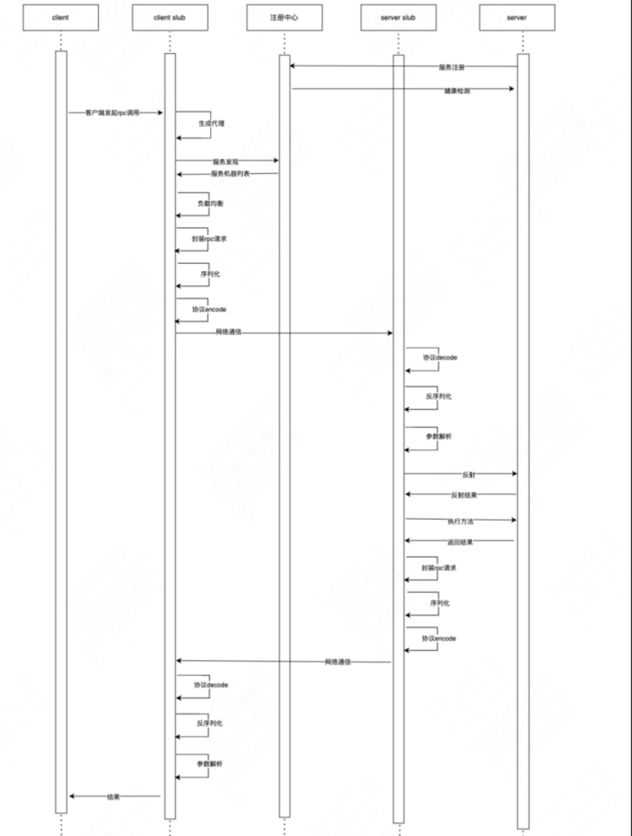
以上对于用户使用client来说，屏蔽了client slub细节，使得调用远程方法像调用本地方法一样方便。

图4-3

# RPC框架实现

## 开发框架环境

硬件：windows 10 ，内存 16G

软件：JDK 1.8 、lntelliJ IDEA 2023.1.2、 Apache Maven 3.3.9 、Zookeeper 3.5.7

## 客户端网络传输模块实现

首先我们需要明白，客户端需要封装好对应的请求，并通过网络传输发送给服务端，服务端执行完后，在包装好返回体返回给客户端。这里涉及到三个模块，请求，响应，网络传输。

### 5.2.1 request实现

请求实体request设计如下：

1. @AllArgsConstructor

2. @NoArgsConstructor

3. @Getter

4. @Builder

5. @ToString

6. public class RpcRequest implements Serializable {

7. private static final long serialVersionUID = 1905122041950251207L;

8. private String requestId;

9. private String interfaceName;

10. private String methodName;

11. private Object[] parameters;

12. private Class<?>[] paramTypes;

13. private String version;

14. private String group;

15.

16. public String getRpcServiceName() {

17. return this.getInterfaceName() + this.getGroup() + this.getVersion();

18. }

19. }

这里的注解都使用了lombok工具：

​ Lombok是一个Java库，它通过自动生成代码来简化Java类的编写。它提供了一组注解，可以在编译时生成常见的Java代码，如getter和setter方法、构造函数、equals和hashCode方法等，从而减少了手动编写这些重复代码的工作量。

​ 使用Lombok可以使Java类的代码更加简洁、可读性更高，同时也减少了出错的可能性。在编程过程中，我们经常需要编写大量的getter和setter方法、构造函数和其他辅助方法，这些代码往往是重复且繁琐的。而使用Lombok，我们只需要在类上添加相应的注解，编译时Lombok会自动为我们生成这些代码。

以下是一些Lombok提供的常用注解：

|  |  |
| --- | --- |
| @Getter / @Setter | 自动生成属性的getter和setter方法 |
| @NoArgsConstructor / @RequiredArgsConstructor / @AllArgsConstructor | 自动生成无参构造函数、根据参数生成必需参数的构造函数、以及生成包含所有参数的构造函数 |
| @ToString | 自动生成toString方法 |
| @EqualsAndHashCode | 自动生成equals和hashCode方法 |
| @Data | 结合了@Getter、@Setter、@ToString、@EqualsAndHashCode和@NoArgsConstructor的功能 |
| @Builder | 通过Builder模式自动生成构建对象的代码 |
| @Slf4j | 自动生成日志记录的代码 |

Request类实体为何这么设计？

首先，客户端调用服务端，我们需要知道我们调用的方法接口名interfaceName、方法名methodName、以及方法参数类型parametersType，这三者可以唯一组成一个方法签名，有了方法签名，还需要考虑我们的方法入参parameters。

但是并不仅有上面几个成员变量就可以了：我们需要考虑在有很多的远程调用请求下，请求和相应如何一一对应；这个时候就会用到requestId，一个接口服务端有多个实现，就会用到group；如果涉及到协议升级，需要考虑升级过程中的兼容性，就需要用version字段来表示新旧请求。另外请求实体需要实现Serializable接口，这样序列化时才能成功序列化。

这样，我们的一个RPC协议请求体就完成了

### 5.2.2 response实现

Response实体实现代码如下：

1. @AllArgsConstructor

2. @NoArgsConstructor

3. @Getter

4. @Setter

5. @Builder

6. @ToString

7. public class RpcResponse<T> implements Serializable {

8.

9. private static final long serialVersionUID = 715745410605631233L;

10. private String requestId;

11. /\*\*

12. \* response code

13. \*/

14. private Integer code;

15. /\*\*

16. \* response message

17. \*/

18. private String message;

19. /\*\*

20. \* response body

21. \*/

22. private T data;

23. }

24.

这个返回体其实和一般的调用返回值一样，由data、code、message组成，这里多了一个requestId，用于和客户端的请求进行唯一关联定位，这样当客户端接收到返回值之后，才知道是谁的请求对应的返回值。

### 5.2.3网络传输

首先，我们将客户端调用服务端流程操作抽象一下，其实就是发送请求，接收请求。如此，我们定义一个抽象的接口方法：

1. public interface RpcTransport {

2. Object sendRequest(RpcRequest request);

3. }

接下来，我们是使用netty实现网络传输开发的，所以我们需要先知道一个传统的netty客户端是怎么创建的，下面我举一个例子：

1. public class NettyClient {

2. private String host;

3. private int port;

4.

5. public NettyClient(String host, int port) {

6. this.host = host;

7. this.port = port;

8. }

9.

10. public void run() throws Exception {

11. EventLoopGroup group = new NioEventLoopGroup();

12.

13. try {

14. Bootstrap bootstrap = new Bootstrap();

15. bootstrap.group(group)

16. .channel(NioSocketChannel.class)

17. .option(ChannelOption.SO\_KEEPALIVE, true)

18. .handler(new ChannelInitializer<SocketChannel>() {

19. @Override

20. protected void initChannel(SocketChannel ch) throws Exception {

21. ch.pipeline().addLast(new YourClientHandler());

22. }

23. });

24.

25. ChannelFuture future = bootstrap.connect(host, port).sync();

26. Channel channel = future.channel();

27.

28. // 向服务器发送消息

29. String message = "Hello, server!";

30. channel.writeAndFlush(message);

31.

32. // 在连接关闭之前阻塞

33. channel.closeFuture().sync();

34. } finally {

35. group.shutdownGracefully();

36. }

37. }

38.

39. public static void main(String[] args) {

40. String host = "localhost";

41. int port = 8888;

42.

43. NettyClient client = new NettyClient(host, port);

44. try {

45. client.run();

46. } catch (Exception e) {

47. e.printStackTrace();

48. }

49. }

50. }

51.

类似java中的socket编程一样，一般客户端都需要指定连接的目标ip和端口。对于netty来说，我们不仅仅要设置客户端连接的ip和端口，我们还需要控制客户端接受到网络事件时处理的handler，如图 2-5所示，我们可以设置多个handler，并通过前后指针连接成pipeline。

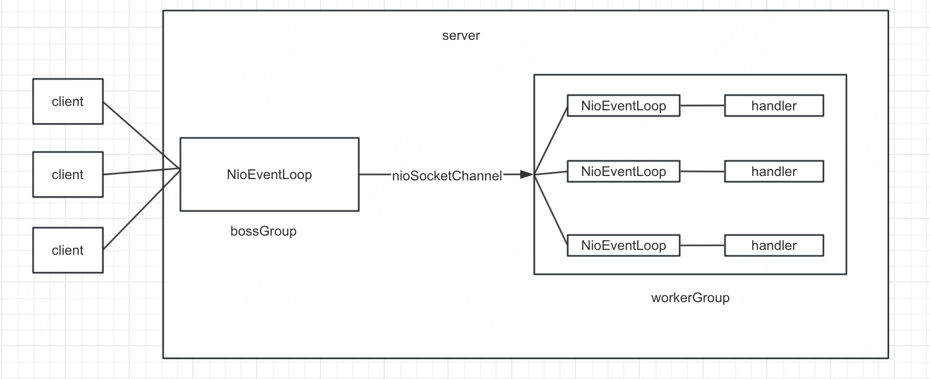
客户端连接到服务端后，如图5-1所示，服务端的bossGroup中nioEventLoop线程循环监听连接事件，一旦建立成功，就会生成NioSocketChannel通道并注册到workerGroup中对应的NIoEventLoop事件循环中，其中一个loop相当于一个while循环线程，当有读写信息时，就会触发对应的读写事件，执行多个前后相连的handler，handler中可以做我们的序列化、协议封装、压缩、心跳检测等机制，所以我们开发的重点在如何设计对应的handler。

图5-1

## Handler设计

### 5.3.1 Handler之序列化

如果我们需要持久化 Java 对象比如将 Java 对象保存在文件中，或者在网络传输 Java 对象，这些场景都需要用到序列化。

序列化： 将数据结构或对象转换成二进制字节流的过程

反序列化：将在序列化过程中所生成的二进制字节流转换成数据结构或者对象的过程

对于 Java 这种面向对象编程语言来说，我们序列化的都是对象（Object）也就是实例化后的类(Class)，但是在 C++这种半面向对象的语言中，struct(结构体)定义的是数据结构类型，而 class 对应的是对象类型。

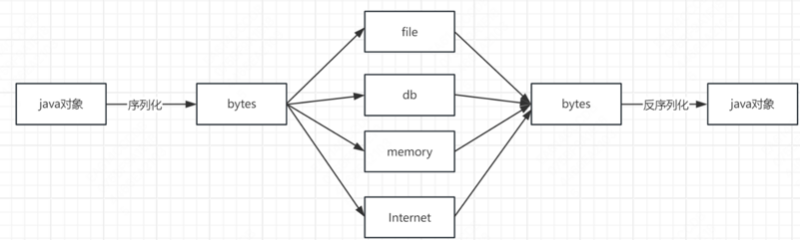
下图5-2是序列化和反序列化常见应用场景：

图5-2

①对象在进行网络传输（比如远程方法调用 RPC 的时候）之前需要先被序列化，接收到序列化的对象之后需要再进行反序列化；

②将对象存储到文件之前需要进行序列化，将对象从文件中读取出来需要进行反序列化；

③将对象存储到数据库（如 Redis）之前需要用到序列化，将对象从缓存数据库中读取出来需要反序列化；

④将对象存储到内存之前需要进行序列化，从内存中读取出来之后需要进行反序列化。

核心就是：序列化的主要目的是通过网络传输对象或者说是将对象存储到文件系统、数据库、内存中。

常见的序列化工具如下：

1. JDK自带序列化工具

JDK 提供了`ObjectOutputStream`用于支持序列化，`ObjectInputStream`用于反序列化。注意，\*\*使用 JDK 自带的序列化工具时，java bean 必须实现`Serializable`，否则会抛出`NotSerializableException`异常\*\* 。使用关键字 transient 修饰的成员属性不会被序列化。

1. fastJson

fastjson 由阿里团队开发，是目前最快的Java 实现的 json 库。 fastjson 的 API 非常简洁，并且支持一定程度的定制（例如，注解` @JSONField`、枚举`Feature`等定制序列化）。被人诟病的，可能是 fastjson 的 bug 比较多。

③Jackson

jackson 由 fasterxml 组织开发，相比 fastjson，有着更强大的功能、更高的稳定性、更好的扩展性、更丰富的定制支持。Spring 默认使用的 json 解析工具就是 jackson。

使用 jackson 需要注意，\*\*`ObjectMapper`对象是线程安全的，可以重复使用\*\*。

④kryo

kryo 由 EsotericSoftware 组织开发，不兼容 jdk 自带序列化工具的数据，kryo 序列化后的数据要更小，至于 API 的简洁性方面，我觉得还是差了一些，一不小心就会采坑。使用 kryo 需要注意以下几点：

1. \*\*`Kryo`对象不是线程安全的，可以使用`ThreadLocal`或池来获取\*\*（本文使用池获取）；

2. kryo 通过类注册可以在序列化数据中写入类的 class id，而不是类的全限定类名，从而减小序列化数据的大小。但是，我们很难保证同样的类在不同的机器上注册的 class id，所以，建议设置`kryo.setRegistrationRequired(false);`，因为同样的 Class 在不同的机器上注册编号很难保证一致；

3. 当 java bean 出现循环引用时，使用 kryo 可能会出现栈内存溢出，这个时候可以通过设置`kryo.setReferences(true);`来避免。如果项目中不可能出现循环引用，则可以设置为 false 以提高性能。

⑤protostuff

protostuff 是基于 google protobuf 开发而来（与 protobuf 相比，protostuff 在几乎不损耗性能的情况下做到了不用写.proto文件来实现序列化），不兼容 jdk 自带序列化工具的数据，序列化后的数据要更小。使用 protostuff 需要注意几点：

1. protostuff 使用字段的定义顺序作为字段的 tag，新增字段时必须保证原字段顺序不变，否则旧数据可能会反序列化失败；

2. protostuff 不能直接序列化 Array、List、Map，如果需要序列化，需要先包装成 java bean；

本课题将设计多个序列化工具的选择，让开发者可以通过配置的方式进行序列化工具的选择，接下来先定义序列化工具的公共接口：

1. public interface Serializer {

2.

3. byte[] serialize(Object obj); //序列化

4.

5. <T> T deserialize(byte[] bytes,Class<T> clazz); //反序列化

6. }

7.

其他不同的序列化方式可以实现这个方法，采用策略模式的设计模式来使用不同的序列化工具。

### 5.3.2 Handler之压缩

仅仅通过序列化机制，只是解决了网络传输问题，如果传输的数据太大，性能和时间消耗还是挺大的，所以我们需要对序列化后的数据进行压缩。

①Bzip2

bzip2是Julian Seward开发并按照自由软件／开源软件协议发布的数据压缩算法及程序。Seward在1996年7月第一次公开发布了bzip2 0.15版，在随后几年中这个压缩工具稳定性得到改善并且日渐流行，Seward在2000年晚些时候发布了1.0版。bzip2比传统的gzip的压缩效率更高，但是它的压缩速度较慢。

②Deflater

DEFLATE是同时使用了LZ77算法与哈夫曼编码（Huffman Coding）的一个无损数据压缩算法，DEFLATE压缩与解压的源代码可以在自由、通用的压缩库zlib上找到，zlib官网：http://www.zlib.net/ jdk中对zlib压缩库提供了支持，压缩类Deflater和解压类Inflater，Deflater和Inflater都提供了native方法。

③Gzip

gzip的实现算法还是deflate，只是在deflate格式上增加了文件头和文件尾，同样jdk也对gzip提供了支持，分别是GZIPOutputStream和GZIPInputStream类，同样可以发现GZIPOutputStream是继承于DeflaterOutputStream的，GZIPInputStream继承于InflaterInputStream，并且可以在源码中发现writeHeader和writeTrailer方法。

④Lz4

LZ4是一种无损数据压缩算法，着重于压缩和解压缩速度。

⑤Lzo

LZO是致力于解压速度的一种数据压缩算法，LZO是Lempel-Ziv-Oberhumer的缩写，这个算法是无损算法。

⑥Snappy

Snappy（以前称Zippy）是Google基于LZ77的思路用C++语言编写的快速数据压缩与解压程序库，并在2011年开源。它的目标并非最大压缩率或与其他压缩程序库的兼容性，而是非常高的速度和合理的压缩率。

这里对压缩操作抽象为接口，使用策略模式使用对应的压缩工具：

1. public interface Compress {

2. byte[] compress(byte[] bytes);

3. byte[] decompress(byte[] bytes);

4. }

### 5.3.3 handler之心跳检测

心跳检测要求每隔固定时间，服务端向客户端发送对应的心跳检测，如果心跳检测不成功，就需要断开相应的客户端服务端链接。那么如何更高效快速的开发心跳机制呢？Netty提供了一个组件，叫做IdleStateHandler，可以很方便的使用心跳机制。

public IdleStateHandler(long readerIdleTime, long writerIdleTime,

long allIdleTime,TimeUnit unit){

}

第一个参数是隔多久检查一下读事件是否发生，如果 channelRead() 方法超过 readerIdleTime 时间未被调用则会触发超时事件调用 userEventTrigger() 方法；

第二个参数是隔多久检查一下写事件是否发生，writerIdleTime 写空闲超时时间设定，如果 write() 方法超过 writerIdleTime 时间未被调用则会触发超时事件调用 userEventTrigger() 方法；

第三个参数是全能型参数，隔多久检查读写事件；

第四个参数表示当前的时间单位。

所以这里可以分别控制读，写，读写超时的时间，单位为秒，如果是0表示不检测，所以如果全是0，则相当于没添加这个 IdleStateHandler，连接是个普通的短连接

IdleStateHandler 既是出站处理器也是入站处理器，继承了 ChannelDuplexHandler 。通常在 initChannel 方法中将 IdleStateHandler 添加到 pipeline 中。然后在自己的 handler 中重写 userEventTriggered 方法，当发生空闲事件（读或者写），就会触发这个方法，并传入具体事件。

如何设计呢？首先我们需要分清楚 客户端、服务端、组件IdleStateHandler 三者之间的关系。

客户端会因为需要主动调用服务端，所以是主动产生事件的一方，然后会接收服务端信息，会接收读事件。服务端会被动地接收客户端的信息，所以会产生读事件。

IdleStateHandler 组件可以配置一段时间内没有产生读事件或者写事件时，调用userEventTriggered 方法，并产生IdleStateEvent，我们就可以在userEventTriggered 中生成相应的心跳包。

现在需要确定哪边产生心跳包，哪边长时间未收到心跳就主动断开连接。

如图5-3所示，服务端连接着多数的客户端，而且性能瓶颈在服务端这边，所以应该服务端来主动断开连接，即一段时间内没有收到读事件即断开连接；客户端是主动发送信息的一方，如果长时间没有发送信息，应该向服务端发送心跳包，这样服务端可以刷新一次读事件，重新记时后续的IdleStateEvent。

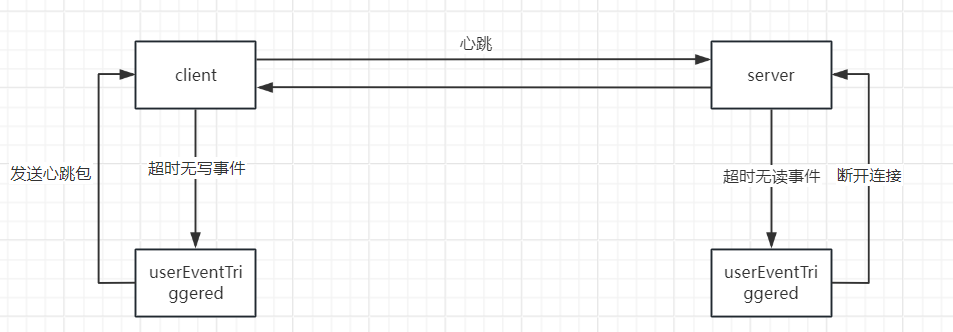


图5-3

对于服务端：

new IdleStateHandler(30, 0, 0, TimeUnit.SECONDS);

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {

if (evt instanceof IdleStateEvent) {

IdleState state = ((IdleStateEvent) evt).state();

if (state == IdleState.READER\_IDLE) {

log.info("idle check happen, so close the connection");

ctx.close();

}

} else {

super.userEventTriggered(ctx, evt);

}

}

对于客户端：

new IdleStateHandler(0, 5, 0, TimeUnit.SECONDS);

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {

if (evt instanceof IdleStateEvent) {

IdleState state = ((IdleStateEvent) evt).state();

if (state == IdleState.WRITER\_IDLE) {

log.info("write idle happen [{}]", ctx.channel().remoteAddress());

Channel channel = nettyRpcClient.getChannel((InetSocketAddress) ctx.channel().remoteAddress());

RpcMessage rpcMessage = new RpcMessage();

rpcMessage.setCodec(SerializationTypeEnum.PROTOSTUFF.getCode());

rpcMessage.setCompress(CompressTypeEnum.GZIP.getCode());

rpcMessage.setMessageType(RpcConstants.HEARTBEAT\_REQUEST\_TYPE);

rpcMessage.setData(RpcConstants.PING);

channel.writeAndFlush(rpcMessage).addListener(ChannelFutureListener.CLOSE\_ON\_FAILURE);

}

} else {

super.userEventTriggered(ctx, evt);

}

}

以上通过IdleStateHandler和userEventTriggered实现了心跳机制。此时，当客户端有5s没有产生读事件，如果客户端还存活，此时会向服务端发送一个心跳包，服务端收到这个心跳包读包时，就会刷新IdleStateHandler并返回心跳response。

如果超过30s，客户端没有向服务端发送任务信息，服务端就会触发userEventTriggered方法，其中的逻辑就会断开客户端与服务端之间的连接。

## 协议的设计与实现

### 5.4.1 协议设计

在前面我们讲了，设计协议的目的是为了解决粘包拆包的问题。那么如何设计我们的协议呢？

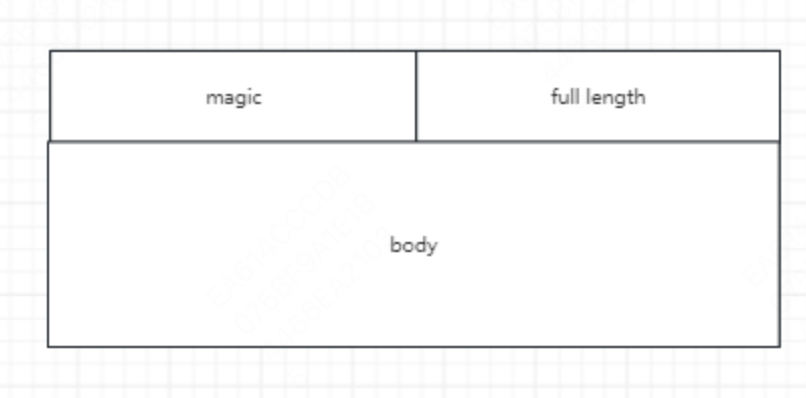
协议是客户端和服务端共同规定的协议，我们需要知道tcp传输的字节流中，一条rpc请求什么时候开始，消息长度为多少。这样我们就可以通过消息起始位置+消息长度定位到我们一次rpc请求的边界，以解决粘包黏包问题。

图5-4

这种协议已经可以简单的解决拆粘包问题，但是前文介绍了序列化与压缩，以及服务端的返回值如何对应到客户端的唯一请求，所以我们还需要在请求头中加入更多的数据来进行请求体数据的控制：

图5-5

详细解释一下这几个字段的含义：

magic魔法数 ： 通常是 4 个字节。这个魔数主要是为了筛选来到服务端的数据包，有了这个魔数之后，服务端首先取出前面四个字节进行比对，能够在第一时间识别出这个数据包并非是遵循自定义协议的，也就是无效数据包，为了安全考虑可以直接关闭连接以节省资源。

full length：整个数据包的总长度，通过此字段定位一个数据包的结束。

requestId：客户端根据requestId可以找到对应的服务端返回的rpcResponse

version：协议的版本号

messageType: 正常的Rpc传递消息、心跳检测消息

序列化器类型 ：标识序列化的方式，比如是使用 Java 自带的序列化，还是 json，kyro 等序列化方式。

compressType：压缩类型

但是这样设计过后，如果后续业务上需要添加新的消息头字段怎么办？

例如后续版本+1，在协议头新加了字段bzId，此时如果客户端更新了协议版本，还没有更新的服务端收到更新的协议数据，因为协议头长度改变，服务端会读出错误的数据，那么此时，就是一个严重的bug。

所以我们还需要考虑协议的可扩展向后兼容性。

其关键在于让协议头支持可扩展，扩展后协议头的长度就不能定长了。那要实现读取不定长的协议头里面的内容，在这之前肯定需要一个固定的地方读取长度，所以我们需要一个固定的写入协议头的长度。整体协议就变成了三部分内容：固定部分、协议头内容、协议体内容，前两部分我们还是可以统称为“协议头”，具体协议如下：

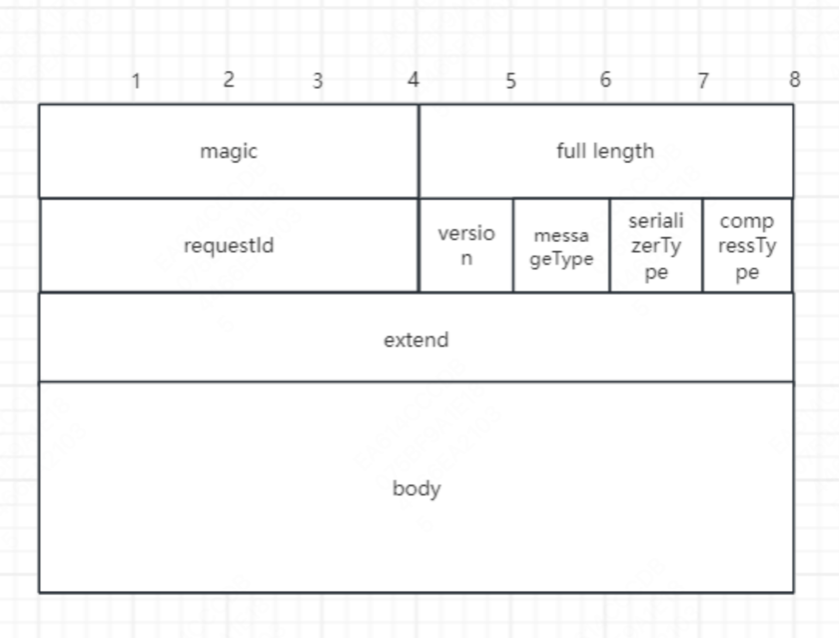


图5-6

这样，当客户端协议头新增加了字段，服务端也能够在正确的处理解析协议。

现在协议已经设计完成了，现在需要考虑如何在netty中进行实现。

### 5.4.2 协议实现

①对于客户端：

netty提供了一个组件叫做MessageToByteEncoder。

在Netty中，MessageToByteEncoder是一个抽象类，用于将消息对象编码为字节数据发送到网络。它是Netty中的一种编码器，通常与ChannelPipeline一起使用。

MessageToByteEncoder的作用是将消息对象转换为字节数据，然后将这些字节数据写入到Channel中发送给远程对等方。它将消息对象作为输入，并将其转换为字节数据，以便可以通过网络进行传输。

要实现自定义的MessageToByteEncoder，需要扩展MessageToByteEncoder类并实现encode()方法。encode()方法接收两个参数：ChannelHandlerContext和要编码的消息对象。需要在encode()方法中执行编码逻辑，并使用ChannelHandlerContext的write()方法将字节数据写入Channel。

我们封装好了RpcRequest数据作为请求体后，对应的请求头信息还需要我们在这个MessageToByteEncoder中进行封装。

1. public class RpcMessageEncoder extends MessageToByteEncoder<RpcMessage> {

2. @Override

3. protected void encode(ChannelHandlerContext channelHandlerContext, RpcMessage rpcMessage, ByteBuf byteBuf) throws Exception {

4. byteBuf.writeBytes(RpcConstants.MAGIC);

5. byteBuf.writerIndex(byteBuf.writerIndex() + 4);// 跳过长度字段

6. byteBuf.writeInt(idGenerator.getNextId());//requestId

7. byteBuf.writeByte(RpcConstants.VERSION);

8. byteBuf.writeByte(rpcMessage.getMessageType());

9. byteBuf.writeByte(rpcMessage.getSerializeType());

10. byteBuf.writeByte(rpcMessage.getCompressType());

11. byteBuf.writerIndex(byteBuf.writerIndex() + RpcConstants.EXTEND\_LENGTH);//extend字段

12. byte[] body = null;

13. int allLength = RpcConstants.HEAD\_LENGTH;

14. if (MessageTypeEnum.HEARTBEAT\_REQUEST\_TYPE.getCode() != rpcMessage.getMessageType() ||

15. MessageTypeEnum.HEARTBEAT\_RESPONSE\_TYPE.getCode() == rpcMessage.getMessageType()) {

16. byte[] serializedData = serializer.serialize(rpcMessage.getData());//TODO

17. body = compress.compress(serializedData);

18. allLength += body.length;

19. }

20. if(body!=null){

21. byteBuf.writeBytes(body);

22. }

23. int writerByte=byteBuf.writerIndex();

24. byteBuf.writerIndex(writerByte-allLength+RpcConstants.MAGIC.length);

25. byteBuf.writeInt(allLength);

26. byteBuf.writerIndex(writerByte);

27. }

28.

可以把ByteBuf理解为一串待填充的字节流，我们就需要根据协议向字节流中填充数据。

先介绍几个重要的方法：

ByteBuf是用于存储字节数据的缓冲区对象

- `byteBuf.writeBytes(byte[] src)`: 这个方法将给定的字节数组src的内容写入到ByteBuf中。它会将src的所有字节按照顺序写入到当前ByteBuf的writerIndex位置，并且将writerIndex自动增加写入的字节数。

- `byteBuf.writerIndex()`: 这个方法返回当前ByteBuf的写索引（writer index）。写索引表示下一个要写入数据的位置，也就是下一个数据将被写入到ByteBuf的哪个位置。初始情况下，写索引为0。

- `byteBuf.writerIndex(index)`: 这个方法将当前ByteBuf的写索引（writer index）设置为指定的索引index。通过调用该方法，您可以控制下一次写操作的位置。注意，如果指定的索引超出了ByteBuf的容量，则会抛出IndexOutOfBoundsException异常。

填充流程如下：

1. 写入4字节的magic字段，表示协议数据包的开始标识

2. 接下来本来是长度字段，该字段指的是整个协议数据包的总长度，但是目前还不知道请求体的具体大小，所以这里使用byteBuf.writerIndex(byteBuf.writerIndex() + 4)先跳过4个字节长度。

3. 写入4字节requestId

4. 一次写入分别为1字节的版本号、消息类型、序列化类型、压缩类型，共占用四字节

5. 写入8字节扩展字段

6. 写入RpcRequest实体，这里会根据具体的序列化器、压缩器进行压缩，并计算实际占用字节大小。

7. 将实际长度写入length字段中

通过上面的操作，相应的经过协议处理后的数据就可以写入bytebuf中了

②对于服务端，netty中提供了一个专门的组件叫做LengthFieldBasedFrameDecoder，专门用来解决粘包拆包的问题，将在后续的5.3 节中讲到。

## slub层实现

在设计代理层之前，需要搞清楚在哪个阶段使用代理。从4.3节可知，在客户端调用本地接口方法时，此时会调用代理层。这里需要搞清楚两个问题：

1. 怎么标识一个接口是我们定义的可以远程调用的接口
2. 怎么触发代理
3. 代理层的具体逻辑是怎么样的

### 5.5.1 客户端注解标识远程接口

目前国内大多数后端java开发都会用到springboot框架，而该框架使用最多的就是基于注解开发。知名产品如dubbo等都是使用了基于注解的方式进行服务注册或者发现。

基于以上观点，本课题也使用基于注解的模式。首先我们需要创建一个自定义的注解：

1. @Documented

2. @Retention(RetentionPolicy.RUNTIME)

3. @Target({ElementType.FIELD})

4. @Inherited

5. public @interface

6. RpcReference {

7. String version() default "";

8. String group() default "";

9. }

10.

定义好了注解后，我们如何使用到注解？我们可以借鉴springboot中的bean生命周期。

介绍大致流程：

下面我们将使用BeanPostProcessor来处理RpcReference 注解：

通过postProcessAfterInitialization，扫描所有的bean，如果发现某一个bean中的成员变量使用了注解RpcReference，那么则表示我们所引用的这个成员变量需要替换为对应的代理对象，这时候我们new一个代理对象，并将原对象中的成员变量替换为我们的目标代理对象。

1. @Override

2. public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

3. Class<?> targetClass = bean.getClass();

4. Field[] declaredFields = targetClass.getDeclaredFields();

5. for (Field declaredField : declaredFields) {

6. RpcReference rpcReference = declaredField.getAnnotation(RpcReference.class);

7. if (rpcReference != null) {

8. RpcServiceConfig rpcServiceConfig = RpcServiceConfig.builder()

9. .group(rpcReference.group())

10. .version(rpcReference.version()).build();

11. RpcClientProxy rpcClientProxy = new RpcClientProxy(rpcClient, rpcServiceConfig);

12. Object clientProxy = rpcClientProxy.getProxy(declaredField.getType());

13. declaredField.setAccessible(true);

14. try {

15. declaredField.set(bean, clientProxy);

16. } catch (IllegalAccessException e) {

17. e.printStackTrace();

18. }

19. }

20.

21. }

22. return bean;

23. }

24.

以上部分，是在程序启动的时候，根据RpcReference配置将一些普通的对象替换为了代理对象，但是代理对象的具体逻辑还没有被执行，具体执行的时候是在客户端真正调用这个代理对象的时候。

### 5.5.2 客户端代理层自动封装请求参数并发送

接下来看看具体的代理的实现是如何实现的：

当真正客户端开始调用对应的方法的时候，会通过代理的方式，执行invoke方法，在方法中，我们可以获取对应的方法签名，封装发送请求的参数，然后将调用远程调用的方法。

1. @Override

2. public Object invoke(Object proxy, Method method, Object[] args) {

3. log.info("invoked method: [{}]", method.getName());

4. RpcRequest rpcRequest = RpcRequest.builder().methodName(method.getName())

5. .parameters(args)

6. …………

11. .build();

12. RpcResponse<Object> rpcResponse = null;

13. if (rpcRequestTransport instanceof NettyRpcClient) {

14. CompletableFuture<RpcResponse<Object>> completableFuture = (CompletableFuture<RpcResponse<Object>>) rpcRequestTransport.sendRpcRequest(rpcRequest);

15. rpcResponse = completableFuture.get();

16. }

17. this.check(rpcResponse, rpcRequest);

18. return rpcResponse.getData();

19. }

### 5.5.3 服务端注解自动注册

服务端需要将自己对外开发的接口注册到注册中心，根据世面上主流的rpc框架，一般采用注解的方式，将自己的接口的信息注册到注册中心。

这里和上文一样，需要用到spring bean 的生命周期，也需要用到自定义注解来自动注册相关信息。

如下所示为RpcService注解定义：

1. @Documented

2. @Retention(RetentionPolicy.RUNTIME)

3. @Target({ElementType.TYPE})

4. @Inherited

5. public @interface RpcService {

6. String version() default "";

7. String group() default "";

8. }

### 5.5.4 服务端扫描注解并注册

1. @SneakyThrows

2. @Override

3. public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

4. if (bean.getClass().isAnnotationPresent(RpcService.class)) {

5. log.info("[{}] is annotated with [{}]", bean.getClass().getName(), RpcService.class.getCanonicalName());

6. // get RpcService annotation

7. RpcService rpcService = bean.getClass().getAnnotation(RpcService.class);

8. // build RpcServiceProperties

9. RpcServiceConfig rpcServiceConfig = RpcServiceConfig.builder()

10. .group(rpcService.group())

11. .version(rpcService.version())

12. .service(bean).build();

13. serviceProvider.publishService(rpcServiceConfig);

14. }

15. return bean;

16. }

## 注册中心实现

注册中心的本质就是去存储服务端的元数据信息，所以一切存储系统例如redis、mysql、HDFS、zookeeper、etcd都是可以作为注册中心的。但是考虑到注册中心需要保证分区容错和强一致性，即CP，这里我们使用Zookeeper作为注册中心。

### 5.6.1 zookeeper节点存储实现

### 5.6.2 注册中心通用接口实现

1. public interface ServiceProvider {

2. void addService(RpcServiceConfig rpcServiceConfig); //服务端注册到注册中心

3.

4. Object getService(String rpcServiceName); //客户端根据rpc服务名拿到相应的对象

5.

6. void publishService(RpcServiceConfig rpcServiceConfig); //

7. }

8.

对于服务端来说，一个紧要的业务就是需要将自己的服务名字，以及自己的ip等基本信息注册到注册中心，所以我们可以写一个通用的接口，规定注册签名信息。

对于具体的如何注册，我们需要考虑zk节点目录如何设计。

1. public interface ServiceRegistry {

2. void registerService(String rpcServiceName, InetSocketAddress inetSocketAddress);

3. }

4.

5. @Override

6. public void registerService(String rpcServiceName, InetSocketAddress inetSocketAddress) {

7. String servicePath = CuratorUtils.ZK\_REGISTER\_ROOT\_PATH + "/" + rpcServiceName + inetSocketAddress.toString();

8. CuratorFramework zkClient = CuratorUtils.getZkClient();

9. CuratorUtils.createPersistentNode(zkClient, servicePath);

10. }

11.