

本科生毕业论文（设计）



题 目 **基于Netty的RPC框架设计与实现**

学 院  **软件学院**

专 业  **软件工程 080902**

学生姓名  **邹 绿 州**

学 号  **2014141463318** 年级 **2014**

指导教师  **左航**

教务处制表

二Ο一七年十二月十五日

**基于Netty的RPC框架设计与实现**

软件工程

学生 邹绿洲 指导老师 左航

**[摘要]** RPC ,是英文Remote Procedure Call的缩写，翻译为中文叫做远程过程调用。

在现在高速发展的互联网企业中，一个项目是由很多模块（Module）组成的，一个模块可能是由多个服务 (Service)组成的，而每个服务又可能就是一个团队维护的项目，项目之间会有各种各样的联系，比如彼此调用对方的接口，向对方服务发起请求或者对其他服务产生响应等等。要完成诸如此类的功能，就可以使用远程过程调用。

本项目基于目前使用广泛的NIO框架Netty设计和实现的RPC服务器，使用TCP协议进行通信。数据的网络传输方面，使用的是开源序列化协议Protobuf,Hessian以及Java语言本身提供的原生序列化机制进行序列化和反序列化。根据应用的不同的使用场景，选择不同的序列化机制可提供更加丰富的功能，以及更大的可扩展性。

本项目与其他的RPC框架相比，因为采用了Netty提供网络传输，而Netty作为一个异步网络编程框架，使用的是著名的Reactor多线程并发模型，改变了传统的直接使用NIO（Non-blocking IO）进行编程的复杂性，也能够很好的提升了RPC的整体性能。

**[主题词]** 远程过程调用; Netty；序列化

**RPC Framework Based on Netty**

Software Engineering

Student： Zou Lvzhou Adviser: Zuo Hang

**[Abstract]** In today's era, a project is made up of many mudules, a module may be composed of multiple services,and each service is likely to be a team of maintenance project.The project,there will be all kinds of connection between the other side of the interface,such as call each other to the party initiated service request or response on other services and so on.To complete such functions,you can use remote procedure calls.

This project is based on a protocol that uses the extensive NIO framework of Netty to design and implement the RPC server,using the TCP protocol.The data’s network transports,using the open source serialization protocol Protobuf,Hessian,and the Java language itself provide the original serialization mechanism for serialization and deserialization. Depending on the usage scenarios, different serialization mechanisms can be selected to provide richer functionality and greater scalability.

The project is compared with other RPC framework, because USES Netty provides network transmission, and Netty serves as an asynchronous network programming framework, using the famous Reactor multi-threaded concurrent model, changed the traditional direct use NIO (Non - blocking IO) the complexity of programming, also can improve the overall performance of RPC very well.

**[Key Words]** RPC; Netty;Serialization

目 录

[1 绪论 1](#_Toc502054735)

[1.1 项目背景 1](#_Toc502054736)

[1.2 国内外研究现状 3](#_Toc502054737)

[1.2.1 RPC研究现状 3](#_Toc502054738)

[1.2.2 Netty研究现状 4](#_Toc502054739)

[1.3 项目主要工作 4](#_Toc502054740)

[1.3.1 项目概述 4](#_Toc502054741)

[1.3.2 功能描述 5](#_Toc502054742)

[1.3.3 创新点及特色 5](#_Toc502054743)

[1.4 报告组织与结构 7](#_Toc502054744)

[2 项目涉及的相关知识和技术简介 8](#_Toc502054745)

[2.1序列化相关技术和理论 8](#_Toc502054746)

[2.2异步网络通讯框架Netty相关技术和理论 11](#_Toc502054747)

[2.3一站式处理框架Spring相关技术理论 12](#_Toc502054748)

[2.4 其他相关技术和理论 13](#_Toc502054749)

[2.4.1 Java的反射理论 13](#_Toc502054750)

[2.4.2 Gradle介绍 13](#_Toc502054751)

[2.5本章小结 16](#_Toc502054752)

[3 项目工作和进展情况 17](#_Toc502054753)

[3.1项目的概况 17](#_Toc502054754)

[3.2项目的开发内容和结果 17](#_Toc502054755)

[3.2.1 项目架构设计 17](#_Toc502054756)

[3.2.2 系统概要设计 19](#_Toc502054757)

[3.2.3 系统详细实现 24](#_Toc502054758)

[3.3项目开发计划及阶段性完成情况 31](#_Toc502054759)

[3.3.1 项目开发计划 31](#_Toc502054760)

[3.3.2 项目完成情况 32](#_Toc502054761)

[3.4本章小结 39](#_Toc502054762)

[4 讨论与体会 40](#_Toc502054763)

[4.1对项目开发过程的体会 40](#_Toc502054764)

[4.2对项目的评价 40](#_Toc502054765)

[4.3本章小结 41](#_Toc502054766)

[5 小结 42](#_Toc502054767)

[参考资料 43](#_Toc502054768)

声 明 [45](#_Toc502054769)

[致 谢 46](#_Toc502054770)

[附录1 任务申请书 47](#_Toc502054771)

[附录2 可行性研究报告 48](#_Toc502054772)

[附录3 项目开发计划 49](#_Toc502054775)

[附录4 数据要求说明书 51](#_Toc502054778)

[附录5 需求规格说明书 54](#_Toc502054782)

[附录6 用户手册概要 57](#_Toc502054787)

[附录7 概要设计说明书 68](#_Toc502054792)

[附录8 数据库设计说明书 74](#_Toc502054795)

[附录9 组装测试计划 79](#_Toc502054799)

[附录10 详细设计说明书 104](#_Toc502054802)

[附录11 模块开发说明 122](#_Toc502054805)

[附录12 单元测试 147](#_Toc502054808)

# 绪论

现在的互联网企业的产品往往都是体量非常巨大的，这些产品都是有若干个子项目一起组成。这些子项目往往由各个团队不同的开发人员进行合作开发，这些子项目又提供各种各样的服务。各个服务提供有不同的功能，处于系统的不同位置，有的偏向于底层，用的偏向于表现层。

根据软件设计架构的理论[1]，以及出于可复用组件的考虑，往往在一个组件中提供的功能就不用再另一个组件中重复提供。也就是说，当A团队开发一个项目时，需要的功能模块或者说服务在B团队开发的项目中已经存在，那么A团队就不需要再进行重复开发，也就是我们平常说的“重复造轮子”。但是如何获得B团队项目中提供的功能呢？很早以前，我们采取的办法是直接拷贝代码到项目中，这未尝不是一种解决方案，但是这种方式会有如下问题：

问题一：直接拷贝代码可能导致项目代码量急剧增大，如果依赖的其他项目过多，将会使得项目臃肿不堪，难以维护。

问题二：由于开发依赖代码的并不是本团队成员，后期代码维护将是一个重大难题，可能因为依赖代码的不稳定导致项目的风险急剧增大。

问题三“如果依赖的服务需求发生变化，或者需要升级功能，对于拷贝的代码来说将会无能为力。

问题四：如果依赖的功能或者服务出现bug，只能由原开发人员进行修复或者需要其进行帮助修复，这往往会带来不必要的沟通成本和人力成本。

上面列出的问题只是可能出现的一部分问题，除了在这个方向上进行的思考，我们还应该从另一个方向上进行思考，即：这样的做法是否符合软件工程的要求。软件工程理论[2]要求我们构建有效的，可维护的软件，而这样做肯定不符合可维护的要求。

而一些RPC框架的出现就可以很好的解决上述出现的问题，服务A和服务B可以通过网络传输来调用对方的函数或者数据，也更好的提高了代码的可重用性，减少了系统模块功能的复杂性。[3]

## 项目背景

当下的互联网公司中绝大多数都已经的广泛使用各种RPC框架进行服务间的通信，是各个项目不可分割的组成部分，比如使用世界互联网科技巨头Google开源的grpc框架以及社交网络巨头Facebook开源的Thrift框架[4]。在国内有以电子商务起家的阿里巴巴集团开源的dubbo框架最具代表性，此外还有搜索科技公司百度开源的brpc,以及新浪开源的轻量级RPC框架Motan等。

不管是国内还是国外，各大互联网公司都在大量的使用RPC框架进行项目的构建，随着技术的发展，以及互联网产品用户量和数据量的急剧提高，怎样更好的设计一套更优秀的RPC框架将会是程序开发人员和架构师们不断追求的目标。而作为开发人员，怎样更好的理解RPC的原理，怎样在实际开发工作中利用好现有的RPC框架，怎样能够让RPC框架能够发挥更好的性能，都是非常重要的问题。

根据腾讯大数据发布的《腾讯2017代码报告》显示，腾讯2017年新增代码超过5亿行，项目总数31908个。在一个大型的互联网公司中，这么多的项目肯定不是一一独立的，其中有专门针对某个业务的部门，比如我们每天使用的微信，但是微信是一个体量非常大的应用，这个应用肯定需要多个项目最支撑，如与信息安全相关的，需要专门做信息安全的团队；与数据处理和数据分析相关的，需要大数据团队；与风险控制相关的，需要风控团队；与基础架构相关的，需要基础架构团队。而其中，可能大数据项目需要依赖基础架构项目提供的基础设施服务，也可能需要风险控制项目提供的服务。

在一个互联网公司中，做中间件的团队通常是为整个集团的各个项目提供服务的，比如消息队列，分布式缓存，分布式搜索，分布式计算以及云服务等等。

这些所有的场景，归根结底都涉及到一个重要的问题，即：怎样在一个项目中调用另一个项目提供的服务？或者说，怎样在一台机器上调用另一台机器提供得函数或者方法呢？我们知道，不同的项目是运行在不同的服务器上的，也就是其拥有的内存空间是不同的，当然不能直接进行调用。这时，我们自然想到了使用网络进行传输，即使用网络传输我们需要调用的方法或者说需要的数据。而这正是RPC所要解决的重要问题。

俗话说，授人以鱼不如授人以渔，为了理解RPC的底层原理，了解其设计思想以及改进现有的RPC框架，我们完全可以自己动手开发一套RPC框架，这将是一个很有用的尝试。为了使工作更加有效和节约，本文选择几个重要的方向进行研究，避免重复造同样的轮子。

【基于Netty的RPC服务器的优势】Netty是一个异步的、事件驱动的网络应用框架，可以用来快速的构建具有可维护性的高性能的服务程序。Netty是一个NIO的客户端/服务器框架，基于易用的和可伸缩的事件模型，这就是著名的Reactor编程模型。怎样通俗的去理解Netty的作用呢？

简单的讲，在客户端和服务器的通信过程中，为了能够提高通信过程的性能，在Java语言的技术栈之下，我们通常选择的是Java语言提供的NIO框架。IO在计算机系统中意思即为输出输出，而NIO为Non-blocking IO，意为非阻塞的输入输出。

Java语言后来提供的NIO框架相比较于开始提供的IO来讲，具有很大的优势。Java IO是阻塞的，这就是说，当一个线程读取数据或者写入数据时，这个线程就会被阻塞，意味这个线程不能去完成其他任务。Java NIO使用的是非阻塞的模式，它增加了缓冲区的特性，使得线程在读取或者写入时先从缓冲区中读取或者写入数据，即使此时没有数据可用时，它也不会等待，而是可以去完成其他任务。等待缓冲区中有可以操作的数据时，再返回进行操作。

非阻塞的输入输出当然能够提供更好的性能，但是了解Java NIO编程的人就知道，NIO并不是一门容易掌握的技术，对缓冲区的操作也不是件容易的事，这就使得编程开发的难度增加。所以对于绝大多数开发人员来讲，如果能够使用更方便的NIO框架，将会极大的减轻开发负担。而Netty就是这样一个优秀的NIO框架，它是基于TCP协议进行通信的，给开发人员提供基础的编码解码方法，简单的编程模型以及更高的并发处理能力等。

【多种序列化协议进行支持】远程调用的过程，必然要涉及到数据的传输，主要是对象的传输。对于对象的传输，必然要通过网络进行传输。对象序列化的过程，就是将对象变为字节流，通过网络传输的过程；对象的反序列化，就是将传输的字节流重新变为对象的过程。从Java语言本身来讲，其提供了原生的序列化和反序列化机制，只需要实现Java语言提供的接口即可。Java原生的序列化机制从使用上来讲比较简单，不需要引入复杂的函数库，但是其也有一些不好的缺点，比如字节数会占用比较多，这样造成的后果是，在网络传输的时候，可能会消耗更多的带宽以及花费更多的时间。对于Java原生的序列化好反序列化机制存在的缺点，第三方的类库可以很好的优化这些问题。常用的第三方类库有Hessian，Kyro,Protobuf等，借助这些第三方的类库，能够帮助我们在序列化和反序列化过程中提高效率。

## 国内外研究现状

### **1.2.1 RPC框架研究现状**

【Google的开源RPC框架grpc】grpc是搜索科技公司Goole开源的一个高性能的、跨语言的远程过程调用框架。grpc支持多语言的相互调用，也就是在Java代码中可以调用Python代码中的方法或者函数，这对于大公司多语言开发的团队来讲是件非常有意义的事情，不仅服务的使用者和提供者可以位于不同的机器上，它们还可以是由完全不同的语言编写的程序。grpc是基于HTTP2协议的，这直接给grpc框架带来了性能上的提升。我们知道，HTTP 2.0协议相对于HTTP 1.x提供了很多更优秀的特性，比如多路复用，简单的理解就是允许在一个连接之下发起多个请求或者响应消息，这在HTTP 1.x中是办不到的。在计算机网络系统中，连接的建立是依赖TCP协议的。连接的建立需要进行三次握手机制，连接的断开需要经过四次挥手的过程。这些过程中都可能存在网络拥塞的问题，TCP连接数越多，出现网络拥塞的概率也就越大，从而导致性能的下降。HTTP 2.0协议让所有的数据使用同样的一个TCP连接，这样就能够减少网络拥塞出现。

【Facebook的开源RPC框架Thrift】Thrift是社交网络公司Facebook开源的跨语言的远程过程调用框架。Thrift由Facebook独立研发，主要使用于公司各个项目，各个服务之间的RPC通信，支持跨语言，常用的语言诸如C++，Java，Python，PHP，Ruby等等都可以支持。Thrift使用的是一个CS（客户端/服务器）的架构，不管是客户端还是服务器，都可以使用不同的语言进行开发。在不同的语言之间，使用了一种称为IDL（Interface Description Language，接口描述语言）文件来进行关联。Thrift 的数据传输使用的是socket，发送方以约定好的格式发送数据，数据格式是需要定义在IDL文件中的。Thrift支持多种服务模型，包括单线程模型，阻塞式IO的多线程模型，非阻塞式IO的多线程模型等。

【阿里巴巴的开源RPC框架Dubbo】严格来讲，Dubbo并不只是一个RPC框架，它能够做的事情比一般的RPC框架能做的事情多的多。Dubbo是国内互联网巨头公司阿里巴巴开源的、具有非常高性能的服务框架。当然，服务的输出或者输入是通过RPC的方式来完成的，Dubbo也是支持多协议的服务框架。相比较于grpc和Thrift来讲，Dubbo提供了更多的功能，在互联网企业中得到了大量的应用，互联网公司当当就在Dubbo基础之上开发维护了Dubbox框架，使其更适合自己的业务。

### **1.2.2 Netty研究现状**

Netty是JBoss开源的、高性能的、异步的、事件驱动的网络应用框架[5]。Netty目前是最流行的网络应用框架，基于NIO的机制提供了优秀的性能。Netty不仅应用在很多互联网公司的项目中，还应用在很多的开源框架中，作为提升性能的重要组件，下面介绍几个重要的使用Netty的项目。

【用于进行分布式计算的大数据框架Spark】Spark是著名的大数据分布式的计算框架，目前在企业级大数据开发和数据处理项目中是基本的必备组件。经常搭配Hadoop，Hive等开源项目一起使用。Spark具有优异的性能，因为它是基于内存进行计算的。我们知道，在计算机系统中，基于内存的计算要比在磁盘中进行计算速度快得多，大约在1-2个数量级以上。原因之一在于基于内存的计算减少了磁盘IO的消耗，IO是非常耗时的操作。当然，相对比磁盘来讲，内存资源还是比较珍贵的，需要合理的进行使用。另一方面，有些大数据处理项目对数据的实时性要求比较高，基于内存的数据计算也能更好的保证数据的实时性。在Spark 1.6.0版本时，Spark引入了Netty作为其进行通信的框架，将其作为Spark实现RPC通信和数据传输的主要构件。Netty的引入，使得Spark解决了大块数据的传输等问题，更好的保证了Spark的健壮性。

【实时的分布式搜索和分析引擎Elasticsearch】对于大型项目来讲，往往伴随着大用户量和大数据量，而对数据的搜索和分析又是非常常见的需求。比如在电商网站中搜索商品，在后端管理系统中搜索用户记录等等。很早以前，我们都将这些数据存放在数据库系统中，比如常用的关系型数据库系统Mysql和Oracle。但是随着数据量的急剧增大，传统的数据库系统已经远远满足不了需求。一是数据量的增大导数据存储成本的增大，二是这么多的数据对于搜索和分析来讲是个巨大的难题。因此就诞生了分布式的数据搜索和分析引擎，其中Elasticsearch是典型代表。Elasticsearch使用倒排索引的数据结构实现全文检索的功能。同样，Netty在其中同样起着数据通信和数据传输的作用，是的Elasticsearch能够保证数据搜索和分析的实时性和稳定性。

## 项目主要工作

### **1.3.1 项目概述**

本项目实现的是一个基于Netty的RPC服务器，也可以称为一个框架。通信协议选择的是TCP协议；选择Netty作为通讯框架，Netty是基于Java NIO 异步的、事件驱动的网络应用框架，解决传统NIO难于编程的问题；Netty的网络模式是著名的Reactor模式，Reractor模式的原理简单来说就是服务等待请求的不断到来，然后通过事件分发器将事件分发给处理器进行处理，最后得出结果的过程；使用多种序列化协议，包括Java原生的的序列化机制，还包括第三方的序列化框架，如Kryo，Hessian，Protobuf等等，能根据实际需要选择不同的序列化机制，增加了框架的可扩展性和可复用性，在RPC效率方面也可以得到直观的比较。

### **1.3.2 功能描述**

本系统实现了如下功能：

1. 采用基于NIO的异步网络通讯框架Netty进行客户端和服务器之间的通讯。

一个性能良好的的RPC框架，IO模型的选择至关重要。简单的来讲，IO模型可以分为 阻塞式IO和非阻塞式的IO。阻塞式的IO意味着当一个线程进行数据的读取或者写入时，不能去做其他事情，只有等待数据的读取和写入完成之后，才能继续执行。显然，阻塞式的IO会在等待数据准备的过程中浪费大量的时间，这就是其效率较低的根本原因。非阻塞式的IO意味着在数据读取和写入时，线程还能继续去完整其他事情，直到数据准备完成后，通过回调的方式，回到原来的地方继续执行。

而Netty正是对于NIO框架的再次整合，原因在于Java NIO在使用上难度比较大，要想得到比较好的效果，可能需要开发人员具有很深的编程经验和优秀的编程习惯，才能得到比较好的效果。另一方面，NIO在设计上也不是完全完美，举例来说，NIO中重要的一个概念Buffer，也即是缓冲区，在编程操作上来讲就比较麻烦，容易出错。在Netty中，它提供了类似Buffer概念的ByteBuf，使得开发人员操作字节序列更加容易。

1. 使用Java原生的序列化机制对消息进行序列化和反序列化.

Java本身提供的序列化机制，能够对消息进行序列化和反序列化。序列化时，只需要实现Java提供的序列化接口Serializable，这样就能够将对象转换为字节流，通过网络进行传输。在需要对象的时候，又可以将字节流转换为对象，这就是反序列化的过程。在序列化和反序列化过程中，不会丢失对象的信息[6]。

使用原生的序列化机制非常方便，只需要实现一个特定的接口，开发起来比较方便。所以在本项目中，首先使用了原生的序列化机制，对消息进行序列化和反序列化。更为方便的是，Netty已经为我们集成了这样的处理器，我们只需要在程序中直接调用即可。

1. 采用第三方序列化框架如Kryo、Protobuf对消息进行序列化和反序列化，提升了性能。

Java原生的序列化机制虽然简单，但是性能却和第三方序列化框架有明显的差异。为了

提高RPC服务器的性能，增加了第三方序列化框架进行消息的序列化和反序列化。第三方的序列化框架主要在字节流大小以及传输过程进行改进。有的框架比如Hessian传输的是二进制数据，因而速度更快，与原生的序列化机制相比，具有更优异的性能。不过，使用第三方序列化框架会带来编码上难度的提高，代码量增大，考虑到RPC性能的因素，引入这些框架还是值得的。

1. 在Spring框架帮助下进行服务的初始化、服务的加载等等。

Spring是在Java开发中非常重要的容器框架，是J2EE开发中常用的一站式框架，即项目开发中的其他组件都可以通过Spring框架进进行管理。Spring提供了著名的依赖注入的思想，这是一种优秀的管理对象的方式，让对象的创建通过Spring来进行管理，开发人员从繁杂的工作中解放出来，并且带来了更加清晰的结构，开发人员可以更加专注于业务逻辑的实现，编写的代码也更加容易维护[7]。

在本项目中，依然使用Spring提供的优秀特性，特别是依赖注入的功能。对于服务的初始化，服务的注册，服务的加载等重要过程，都通过Spring来进行管理。

1. 为提升性能，使用Apache commons-poo2包提供的对象缓存池技术，帮助提升性能。

Java是一门面向对象的语言，在一个Java应用中，对象的创建和销毁非常常见的事情。对象的创建涉及到最重要的是Java的类加载机制，类加载机制包含以下几个过程：加载、验证、准备、解析和初始化。

a.加载：Java使用new关键字进行类的加载，加载的过程就是查找并加载类的二进制数据。

b.验证：Java代码编译后会生成class文件，验证这个阶段主要是确保字节流文件是正确的。

c.准备：准备阶段是为对象分配内存空间的过程。

d.解析：解析阶段是将虚拟机将符号引用转换为直接引用的过程，其实就是确定内存地址。

e.初始化：执行构造函数，复初值。

从以上过程可以看出，对象的创建需要经历一系列的过程。此外，Java对象的销毁是依赖

Java的垃圾收集机制的，根据不同的垃圾收集算法和垃圾收集器，对象的销毁也需要经过一系列的阶段。在一个Java应用中，对象会被频繁的创建和销毁，对于需要高性能的应用来讲，这样肯定会带来性能的损耗。因此，在本项目中，使用第三方的开源类库，提供了对象池的作用。对应用中可能需要大量生成和销毁的对象，使用对象池将这些对象存储起来。第一次使用的对象先创建好，使用之后并不立即回收，而是将其放入对象池中，以后再次需要这个对象的时候，直接从对象池中取出。通过对象池技术，避免了频繁创建和销毁对象的开销，可以极大的减少在对象创建和销毁带来的性能损耗。

### **1.3.3 创新点及特色**

对比于其他RPC框架，本项目的创新点在于：

1. **有多种序列化协议可以选择，针对不同的使用场景，可以选择不同的序列化协议**

本项目作为一个远程过程调用框架，能在不同的服务器之间传递消息是基础的功能。在不同的服务器之间传递，也就是不共享硬件资源，比如内存空间等。那么此时常用的做法就是通过网络进行传递，通过网络进行传输的时候，传输的是字节流。所以首先需要做的就是将我们传递的消息对象转换为字节流，以及在传递到其他机器后，能够将字节流转换为消息对象。这就是序列化和反序列化的目的和过程。

本项目使用Java语言作为开发语言，而Java语言本身已经提供了序列化机制，只需要我们实现其提供的序列化接口即可。更为方便的是，在选择Netty作为网络通讯框架时，Netty本身已经为我们集成了基于Java原生序列化机制的编码和解码处理器。对于开发人员来讲，是件很方便的事情。

原生的序列化机制固然简单，但是也存在序列化后的字节流文件较大等问题，在网络传输的时候，会消耗较多的带宽，占用更多的资源。带宽不足时，网络传输时间更长，导致RPC性能降低等。

所以，在java语言原生的序列化机制之外，本项目还引入了第三方开源的序列化框架，帮助RPC在序列化阶段提升性能。第三方序列化框架包括Protobuf，Hessian等。通过使用这些框架，能够让使用RPC框架的使用者有更多的选择，根据不同的使用场景可选择不同的序列化协议。

1. **基于Netty进行网络通信，具有很好的性能。**

对于Java应用来讲，对项目中需要进行输入输出或者需要进行通信的，常见的做法是选择Java语言提供的NIO/AIO框架。Java的NIO/AIO框架提供了Buffer，Selector等概念，能够让线程在数据准备的过程中继续做其他的事情，从而大大的提高IO性能。

对于大部分开发者来说，能够用好Java提供的NIO/AIO框架并不容易，需要大量的实践经验和优秀的编程习惯。另外，NIO的架构设计也并非完美，比如我们在对其提供的缓冲区进行操作的时候，往往对索引位置的操作会犯错。因此，这也就是为什么在实际的项目开发过程中，NIO并没有得到大量使用的原因。

所以选择更加易于使用，功能更加完善的NIO框架是常见的解决方案。其中典型的NIO框架就是Netty。本质上来讲，Netty是基于Java NIO的网络通讯框架，可以将其理解为对NIO框架的再次封装，对NIO框架的一些设计缺陷进行完善，使得开发人员能够更好的去使用，同时在性能方面也能有所提高。Netty之所以能够优异的性能，与其使用的线程模型有着密不可分的关系。在后文中，会对其原理做详细介绍。

1. **使用对象池、线程池等池化技术能够减少重复创建对象或者线程资源的开销。**

对象池或者线程池都是池化技术的典型代表。对象池的使用，可以让程序重复的使用池中的对象，而不需要不断的重复进行创建。从Java虚拟机对象分配的角度来讲，这样做会减少堆中内存的占用，因为Java对象的分配几乎都是在堆内存上进行的。我们知道，内存资源对于计算机系统来讲是非常宝贵的，使用池化技术能够避免在堆中重复的创建对象，节省内存资源。另外，从Java虚拟机垃圾回收的角度来讲，使用池化技术还能够避免虚拟机对内存对象的不断回收。有的垃圾收集器在垃圾收集时有所谓的“Stop The World”现象，也就是垃圾收集线程工作时，用户线程并不能同时工作，这会使得应用程序效率变低。同时，回收更少的对象，也能够节约垃圾回收的时间。

池化技术其实有其特定的使用场景，这不难理解，因为不论是对象池还是线程池，如果需要创建的对象或者线程资源不是太多，不会占用系统过多资源的情况下，是没有必要使用池化技术的。而对于一个RPC框架来讲，对象的创建和销毁是非常频繁的，在高并发的场景下，线程的创建和销毁也是非常频繁的，这就注定了我们需要使用池化技术来管理这些资源。

## 报告组织与结构

第一部分：绪论。主要介绍了RPC框架的发展情况和应用场景，以及本项目的研究背景和研究工作，RPC框架的设计目的；

第二部分：项目相关理论。对RPC框架所采用的技术进行了介绍。研究分析了RPC框架的技术原理；

第三部分：RPC框架的设计工作和进展情况。

第四部分：项目的讨论与体会。

第五部分：小结。本章对全文工作进行总结，指出了还需改进的地方。

# 项目涉及的相关知识和技术简介

## 2.1序列化相关技术和理论

**1.Java原生的序列化技术**

Java语言约定，一个了类能够被序列化就需要实现java.io.Serializable这个接口，没有实现这个接口的类将不会保存他们任何的序列化或者反序列化状态。所有可被序列化类的子类也是可以被序列化的。值得注意的是，这个序列化接口并没有任何的方法或者属性，它其实只是相当于定义了一种语义，表明实现这个接口的类是可以被序列化和反序列化的。

对象在序列化时，需要有一个与之进行关联的序列化版本号，称为serialVersionUID，主要用于在反序列化期间确认序列化对象的的发送方和接收方是否属于同一JVM。因为可能在不同的JVM之间，反序列化时会存在一些差异。

**2.Protobuf的序列化技术**

Protobuf是Google开发的语言中立的、平台无关的序列化机制，作用的是结构化的数据。可以将其类似于XML格式，但是我们知道，XML格式在传输数据的时候由于文件体积较大，格式比较复杂，在传输的时候会消耗较多的带宽，在解析时也会比较复杂。相对于XML格式的文件来讲，Protobuf定义的结构化文件体积更小，传输速度更快。一旦定义好了你想要传输的结构化数据，就能通过内置的代码生成引擎自动生成序列化代码。

总结来说，Protobuf主要是用来进行RPC数据传输的，以更好的，体积更小的方式对数据进行编码和解码。主要的过程如下：

（1）定义一个接口说明文件，描述对象成员，接口方法等各种信息。

（2）通过提供的编译器，将接口说明文件编译成具体的语言文件。

（3）将生成的文件引入，然后就可以进行调用。

**3.Hessian的序列化机制**

Hessian是一个基于二进制传输实现的远程通讯函数包。Hessian将需要传输的信息进行序列化之后，就会生成出二进制流。Hessian传输的是标准的二进制文件，使用的传输协议是HTTP协议。

**4.Kryo的序列化机制**

Kryo是一个高性能的序列化框架，Kryo框架的目标是为Java的序列化提供名速度快、效率高和易于使用的编程接口。不论是对于普通文本文件，数据库sql或者网络传输的数据，都能起着非常巨大的作用。对于Kryo来说，它不需要想之前提到的Protobuf一样，不需要先定义一个具体的结构化文件，而是通过其提供的序列化解释器进行数据的读取和写入。关于Kryo的资料并不是很多，因为其本身构成并不复杂，可以参照其Javadoc进行学习。

## 2.2异步网络通讯框架Netty相关技术和理论

**1.Java NIO理论**

Java NIO中有三个最重要的概念：Selector，Channel，Buffer。在Java NIO中，我们是面向块（Block）或者是面向缓冲区编程的。

从本质上来讲，Buffer本身就是一块内存，在底层实现上是一个数组。数据的读写，都是通过Buffer来实现的。此外，除了数组之外，Buffer还提供了对于数据的结构化访问方式，并且还可以追踪到系统对于数据的读写过程。

Channel指的是向其中写入数据或者是从中读取数据的对象，它类似于Java IO中的Stream对象。但是所有数据的读取都是通过Buffer来进行的，永远不会出现直接从Channel中读取数据的情况。与Stream不同的是，Channel是双向的，而一个流只能是InputStream或者是OutputStream，Channel打开之后则可以进行数据读取或者写入。由于Channel是双向的，所以它能更好的反映出底层操作系统的真实情况。这是因为在Linux系统中，底层操作系统的通道就是双向的。

**2.零拷贝**

在理解零拷贝之前，我们先要了解Java提供的一个概念，叫做HeapByteBufrfer，中文可以翻译为间接缓冲。在进行数据读取或者写入时，操作系统并不是直接处理存在于Java堆内存上的那个数组，而是会在操作系统层面，也就是Java虚拟机的内存模型之外，再重新开辟一块新的内存，将Java堆上的直接数组拷贝到操作系统的内存中，然后再与IO系统进行交互。也即是说，如果使用间接缓冲，就会多出来一次数据拷贝的过程。

反之，如果使用直接缓冲的话，就不用有拷贝数组的操作了。我们可以将其理解为零拷贝的概念。需要注意的是，零拷贝的机制本身是由操作系统支持的，Java语言本身只能利用操作系统提供的这种特性。

下面我们可以先梳理一下间接拷贝的过程：

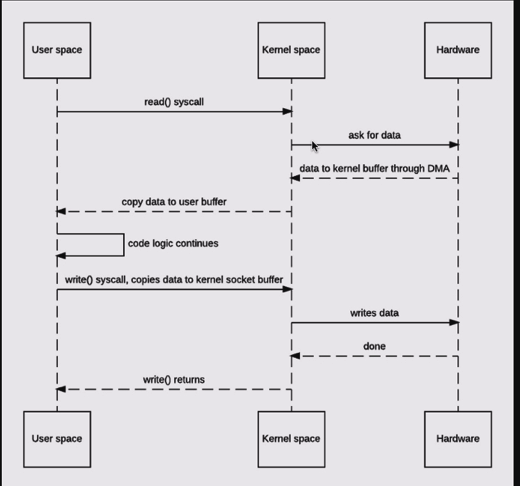
1.JVM从用户空间发出一个系统调用，发送给内核空间。此时操作系统从用户空间模式切换到内核空间模式。

2.内核空间向磁盘发起一个数据请求，数据此时会读取到内核空间的缓冲区中，用户空间没法直接使用。

3.将之前拷贝到内核空间的数据又拷贝到用户空间去（第二次数据拷贝）。

4.JVM再发出一个系统调用，将数据拷贝到网络的一端，此时又需要将此前拷贝到用户空间缓冲区的数据再次拷贝到内和空间缓冲区，然后真正的向网络中写入数据。

过程如下图所示：



**图2.4 操作系统数据拷贝示意图图**

可以发现，在整个拷贝的过程中，用户空间并没有对数据进行任何的处理。而零拷贝就是减少了数据从内核空间到用户空间的拷贝。

**3.Reactor模式**

Reactor模式的角色构成，一共有5种组件：

1.Handle：称为句柄或者描述符，本质上来讲是一种资源，是由操作系统提供的。该资源用于表示一个个的事件，比如说文件描述符，或者针对网络编程中的Socket描述符。事件既可以来自于外部，也可以来自于内部。外部事件包括客户端的连接请求，客户端发送的数据等。内部事件比如说操作系统产生的定时器事件。Handle是事件产生的发源地。

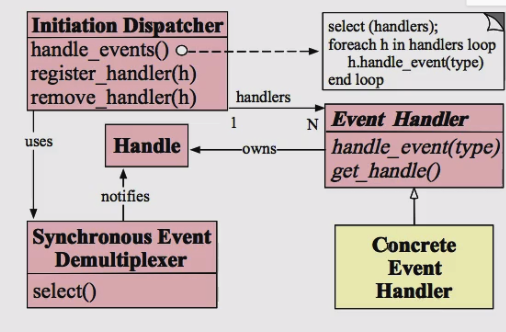
2.Synchronous Event Demultiuplexer：称为同步事件分离器，它本身是一个系统调用，用于等待事件的发生。事件可能只有一个，也可能有多个。调用方在调用它时会被阻塞，一直阻塞到同步事件分离器上有事件产生为止。对于Linux系统来说，同步事件分离器指的就是常用的IO多路复用机制，比如说select,poll等。

3.EventHandler：称为事件处理器，本身由多个回调方法构成，这些回调方法构成了与应用相关的对于某个事件的反馈机制。Netty相对于Java NIO来讲，在时间处理器这个角色上进行了升级，为开发人员提供了大量的回调方法，以供我们在特定事件产生时实现相应的回调方法来进行业务逻辑的处理。

4.Concrete Event Handler：称为具体事件处理器，它本身实现了事件处理器所提供的各个回调方法，用于实现具体的业务逻辑。它本质上是我们所编写所编写的处理器的实现。在Netty中其实就是对应于我们自己编写的Handler。

5.Initiation Dispatcher：称为初试分发器，实际上这就是非常重要的Reactor角色。它本身定义了一些规范，这些规范用于控制事件的调度方式，同时又提供了应用进行事件处理器的注册、删除等设施。它本身是整个事件处理器的核心所在，Initiaation Dispatcher会通过同步事件分离器来等待事件的发生。一旦事件发生， Initiaation Dispatcher首先会分离出每一个事件，然后调用事件处理器，最后调用相关的回调方法来处理这些事件。

下图是Reactor模式示意图：



**图2.5 Reactor模式示意图**

如上图所示，来对Reactor模式的执行流程做一个梳理：

1. 当应用向Initiation Dispatcher注册具体的时间处理器时，应用会标识出该事件处理器希望Initiation Dispatcher在某个事件发生时向其通知该事件，该事件与Handle关联。

2..Initiation Dispatcher会要求每个事件处理器向其传递内部的Handle，该Handle向操作系统标识了事件处理器。

3.当所有的事件处理器注册完毕后，应用会调用handle\_event方法来启动Initiation Dispatcher的事件循环。这时，Initiation Dispatcher会将每个注册的事件管理器的handle合并起来，并使用同步事件分离器等待这些事件的发生。比如说，TCP协议层会使用select同步事件分离器操作来等待客户端发送的数据到达连接的socket handle上。

4.当某个事件源对应的Handle变为ready状态时，同步事件分离器就会通知Initiation Dispatcher。

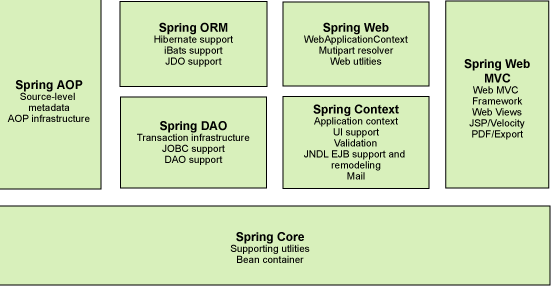
5. Initiation Dispatcher会触发事件处理器的回调方法，从而响应这个处于ready状态的Handle。当事件发生时，Initiation Dispatcher会将被事件激活的Handle作为key来寻找并分发给恰当的事件处理器回调方法。

6. Initiation Dispatcher会回调事件处理器的handle\_events回调方法来执行特定于应用的功能，从而响应整个事件。

## 2.3一站式处理框架Spring相关技术和理论

在之前提到过，Spring是J2EE开发的一站式框架，本节就深入Spring的本质，介绍其重要原理。

下图是Spring框架结构图：



**图2.6 Spring框架架构示意图**

**1.Spring应用的主要设计模式-简单工厂模式**

简单工厂模式是类的创建模式，可以用来帮助创建对象，就是因为使用了这个模式，Spring框架才能够依据配置文件给我们生成对象[8]。

简单工厂模式是类的创建模式，又叫做静态工厂方法。简单工厂模式由一个工厂对象决定决定创建出哪一种类的实例。也就是说，它会根据给他传进去自变量的不同返回不同类的实例。简单工厂模式的实质是由一个工厂类根据传入的参数，动态的决定应该创建哪一个类的实例。

简单工厂模式的组成：

（1）工厂类角色：担任这个角色的是简单工厂模式的核心，具有和应用紧密相关的业务逻辑。工厂类在客户端的直接调用下创建需要的对象，它往往是一个具体的类.

（2）抽象产品角色：担任这个角色的是简单工厂模式创建对象的弗雷。通常它是一个抽象；类或者接口。

（3）具体产品角色：简单工厂模式所创建的任何对象都是这个角色的实例，通常其由一个具体的类来实现。

**2.Spring应用的主要设计模式-代理模式**

代理模式的作用是：为其他对象提供一种代理，以用控制对这个对象的访问。这是因为，在某些时候，一个客户端不能或者不想直接去引用某个对象，此时就可以使用代理对象去进行引用，相当于起到一个桥梁的作用。代理模式对于Spring来讲非常重要，Spring最重要的AOP思想就要依赖代理模式来实现[9]。

代理模式的主要构成有：

（1）抽象角色：声明真实对象和代理对象拥有的共同接口。

（2）代理角色：代理角色内部含有对真实对象的引用。只有这样，代理对象才能够对真实对象进行代理，以便能够操作真实对象。此时，代理对象能够执行真实对象的各种操作。

（3）真实角色：代理角色代表的真实对象，是我们最终进行引用的对象。

**3.Spring的重要思想-AOP**

AOP其实是Aspect-Oriented-Programming的缩写，中文可以翻译为面向切面编程或者面向方面编程。Java是一门面向对象编程的语言，一切皆为对象。面向对象编程的几大特性是继承、封装和多态。通过这几大特性，可以建立出一种对象的层次结构。但是，我们需要注意的是，如果我们想要为那些分散的行为引入到公共部分时，面向对象编程会显得比较复杂。简单的来说，面向对象编程定义的是一种从上之下的结构层次，但是对于从左到右这种横向的对象关系并不适用，比如说异常处理功能。在一个项目的很多地方，我们都需要进行异常处理，但是这些异常的处理代码其实和我们真正关心的业务逻辑并没有太多实质上的关联。这种在各处都可能出现的并且与业务逻辑无关的代码，我们将其称为横切。在面向对象编程中，这种横切代码过多将使系统变得非常臃肿，且难以维护，也不利于我们对模块的重用。

而对于面向切面编程技术来讲，它正是为了解决上述问题而出现的。既然面向对象变成合格定义的是从上至下的一种关系，那么作为其补充和完善，面向切面编程定义的就是一种从左到右的一种关系。面向切面编程使用的被称为“横切”的技术，通俗的理解就是将哪些与具体的业务逻辑无关的公共行为封装起来，将其作为一个公共的模块，这样做就可以极大的减少系统无关的冗余代码，提高系统的可维护性[10]。

## 2.4其他相关技术和理论

### **2.4.1 Java的反射理论**

**1. 简介**

我们知道，Java本质上来讲是一门静态类型的语言。Java程序的运行，分为编译阶段和运行阶段。那我有时候我们需要实现这样一个需求， 也就是在Java运行时环境中，对于任意一个类，我们能不能得到这个类的属性呢？能不能得到这个类拥有的方法呢？我们知道，Java是面向对象的编程语言，我们能不能在运行时调用一个对象的方法呢？前面已经说过，Java是一门静态类型的语言，而在运行期间要得到这些类的信息仿佛是不可能的。不过，Java恰好为我们提供了这样一种机制，让我们能够在程序运行期间得到类的信息，这种机制就是反射。

**2. 反射的功能**

**1).在程序运行期间构造一个类的对象**

我们平常所熟悉的Java生成对象的方式是使用new关键字，使用new关键字构造对象显然不是在运行期完成的事情。使用Java的反射机制，就能够在程序运行期间动态构造出一个对象。

**2).在运行时判断任意一个类所具有的成员变量和方法**

一个类的成员变量或者属性我们能够比较直观的得到，但是在程序运行期间，只有通过反射的方式，动态的去获取类的成员变量和方法。

**3).在程序运行期间调用对象所具有的方法**

**2. Java反射的实现**

在Java中，主要提供了如下的类来实现Java的反射机制。Class类，代表的就是一个类。Field类，代表的是类的成员变量或者熟悉。Method类，代表类的方法。Constructor类，代表类的构造方法。

### **2.4.2 gradle介绍**

**1.简介**

在实际的项目开发过程中，当项目代码开发完成一个阶段之后，我们一般都会进行项目的构建。随着技术的发展，从最开始使用的Ant构建工具，到现在依旧流行的Maven构建工具，再到本项目使用到的构建工具gradle，在实际的开发过程中都是不可或缺的。特别是gradle，作为目前新兴的项目构建工具，因为其强大的功能，受到越来越多开发人员的喜爱。Gradle一般使用构建脚本build.gradle进行构建，我们可以在其中指定特定的项目和任务，这是每个工程都需要的。另外，我们需要的包也可以使用gradle进行引入，相比于Maven来讲，gradle的方式更加的简单，没有xml文件那么庞大和臃肿。

## 2.5本章小结

本章主要对项目涉及到的主要技术和理论做了基本的介绍。先介绍了序列化相关技术和理论，对于RPC来讲，序列化是最基础的工作，这样消息才能通过网络进行传输。再介绍了本项目的基石，也就是异步网络通信框架Netty，特别是对Netty的重要原理进行了较为详细的介绍。接着，对本项目用到的一站式开发框架Spring进行了介绍，Spring框架较为庞大，所以主要对其重要思想AOP进行介绍，以及其用到主要的设计模式。此外，还对Java的反射理论进行了介绍，为什么介绍Java的反射理论呢？这是因为不论是在AOP中，还是Netty中，都离不开反射理论的支撑。最后，对本项目使用的项目构建工具gradle进行了简单介绍，这是目前越来越流行的项目构建工具。

# 项目工作和进展情况

## 3.1项目的概况

本项目是基于Netty进行开发的，Netty作为RPC的通讯框架，并且使用多种序列化框架帮助进行序列化和反序列化。下面先简要的描述整个RPC中最为关键的步骤：

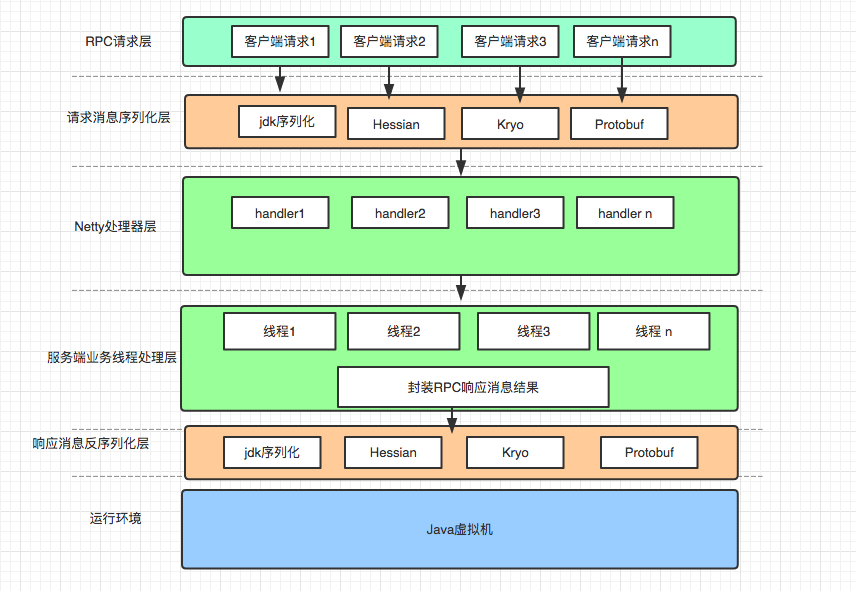
1. RPC本质上是一个请求和响应的过程，客户端发起一个RPC请求，服务器收到请求之后进行处理，得到结果之后再将响应的消息返回给客户端[11]。而对于请求消息以及响应消息来讲都需要规定好其格式，这样客户端发送请求消息以及服务端返回响应消息才有一个统一的结构，便于数据的传输和处理。对于请求消息和响应消息来讲，其本质上就是就是一个POJO类。请求消息应该具有这些字段：请求消息的ID，远程调用过程的类，远程调用的函数/方法，远程调用的类型参数以及参数值等。同样，对于响应消息的POJO类，需要包括这些字段：响应消息的ID，远程调用的结果，返回的错误信息等。我们还应该根据这些POJO类生成其对应的get和set方法。这样，消息的结构就已经定义好了。
2. 作为一个RPC服务器来讲，需要随时满足客户端的RPC请求，也就是一直需要处于运行的状态。在本项目中，我们使用的是Spring框架作为类的加载的容器，所以一旦服务启动的时候，就需要先对我们定义的RPC接口或者接口的实现类进行加载。这其实就是服务启动的过程。需要注意的是，此时服务器并没有执行RPC调用，因为客户端还没有发起具体的请求。只有收到客户端具体的请求之后，服务器才会根据具体的请求执行服务初始化的过程。
3. 服务器启动之后，将一直处于服务监听的状态。此时客户端就可以发起RPC请求，发起的请求消息里面应该包含基本的信息，比如需要远程调用的类的名称，类中的函数名称，包含有哪些属性，属性的值是多少等。一旦构建好这些信息之后，就可以通过网络进行传输[12]。
4. 客户端构建好请求消息之后，需要将消息进行网络传输。传输之前需要做的事情首先是需要将请求消息进行序列化，序列化为字节流之后再经过网络进行传输。根据不同的业务场景，听提供了多种序列化框架进行选择，通过将这些序列化过程编码为处理器，就可以使用典型的Netty编程模型进行调用。
5. 请求消息序列化完成之后，经过网络传输到RPC服务器端，服务器端先将字节流反序列化为消息对象。再根据消息对象的具体内容，比如类名，方法名，属性列表和属性值等信息构造出该对象的实例。在这个过程中，由于一切都是在程序运行时动态发生的过程，所以必然会使用到Java的反射机制。前文已经介绍过，反射机制可以在程序运行期间动态的创建出对象实例。
6. 对象实例创建完成之后，需要将其封装为响应消息，同样需要将响应消息进行序列化成字节流，经过网络传输到客户端。
7. 客户端收到响应消息之后，依然需要将其反序列化成消息对象。要得到RPC结果，还需要对得到的消息进行必要的解析，解析后再显示出远程调用过程是否成功。整个RPC从请求到响应也就全部结束。

## 3.2项目的开发内容和结果

### **3.2.1 系统架构设计**

#### 3.2.1.1 项目架构总体设计

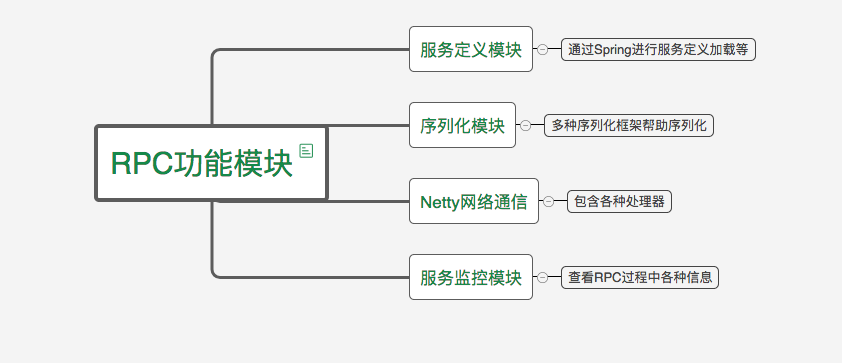
整个系统可分为如下几层：

**图3.1 系统架构示意图**

客户端可以同时发起多个RPC请求消息，请求消息构造好之后，需要进行消息的序列化， 序列化完成之后，通过网络进行传输。其中，专门有一层是封装出多个Netty的处理器，这些处理器包括编码和解码处理器，TCP粘包处理器等[13]，这些处理器会对消息进行链式处理，这是Netty典型的处理方式。服务端收到请求消息之后，会根据消息内容，使用反射机制创建出该消息对象的实例。最后将结果封装为响应消息再返回给客户端。当然，这些所有的阶段，都是运行在Java虚拟机之上的。

。

#### 3.2.1.2 系统模块划分



**图3.2 系统功能模块图**

1. 服务定义及加载模块：通过配置文件对服务的定义和加载进行统一的管理。在该配置文件中，需要使用特定的标签对需要加载的服务进行配置，简单的来说就是我们需要进行RPC的接口类以及其接口类的实现。 服务器在启动的时候，需要先加载配置文件，进而监听客户端的RPC调用请求，所有的这些行为都由Spring进行统一的管理。
2. 序列化模块：不论是请求消息还是响应消息，都需要通过序列化之后才能在网络进行传输。整个RPC的过程都离不开消息的序列化和反序列化。本项目通过多个序列化框架实现可选的多种序列化方式，确保使用者可以根据自己的需求进行序列化的操作[14]。
3. Netty网络模块：本项目使用Netty进行网络通信，根据不同的序列化方式设计对应的处理器。Netty的线程模型选择至关重要，本项目使用的是常用的主-从线程模型。具体来讲就是有一个bossGroup线程循环组进行任务的分配，而其自身并不执行具体的任务，而真正的任务交给workerGroup线程循环组来执行，这就避免了一个任务耗时过长导致线程阻塞的情况。
4. 服务监控模块：通过控制台可以查看RPC服务的运行情况。比如说调用是否成功，调用所消耗时间，如果调用发生错误，输出错误信息等。通过对服务的监控，能够更好的提升RPC的稳定性[15]。

### **3.2.2 系统概要设计**

#### 3.2.2.1 系统用例图设计

系统的用例图如下：



**图3.3 系统用例图**

#### 3.2.2.2 系统流程设计

##### 3.2.2.2.1 系统概要流程设计

1. 服务端加载配置文件，此后将一直处于运行状态，一直等待客户端发起请求。
2. 客户端发起请求，请求消息中应该包含请求类名，方法名，属性名等基本信息。
3. 构建好请求消息之后，通过序列化框架对请求消息进行序列化。
4. 序列化完成之后，通过Netty传输字节数据到服务器端。
5. 服务器端将字节数据反序列化出来，得到相应的请求信息，进行处理之后返回给客户端。
6. 客户端接收到服务器返回的响应信息，解析出响应结果。另外还会得到此次RPC过程的一些信息，比如调用是否成功，调用失败的错误消息等[16]。

### **3.2.3 系统详细实现**

**3.2.3.1 序列化的实现（以Protobuf举例说明，其他序列化框架同理）**

* **使用Protobuf序列化的实现**
* **（1）ProtostuffDecoder.java接口类**
* **表10.1 类ProtostuffDecoder.java接口**

| **编号** | **函数名** | **说明** |
| --- | --- | --- |
| 1 | ProtostuffDecoder(MessageCodeHelper helper) | 构造函数 |

* **（2）ProtostuffEncoder.java接口类**
* **表10.2 ProtostuffEncoder.java接口**

| 编号 | 变量 | 说明 |
| --- | --- | --- |
| 1 | ProtostuffEncoder(MessageCodeHelper helper) | 构造函数 |

* **（3）ProtostuffSerializable.java序列化类主要代码**
* **代码清单10.1 ProtostuffSerializable.java**

|  |
| --- |
| *//序列化* @Override **public void** serialize(OutputStream outputStream, Object object) {   Class clazz = (Class) object.getClass();   LinkedBuffer buffer = LinkedBuffer.*allocate*(LinkedBuffer.***DEFAULT\_BUFFER\_SIZE***);   **try** {   Schema schema = *getSchema*(clazz);  ProtostuffIOUtil.*writeTo*(outputStream, object, schema, buffer);   } **catch** (Exception e) {  **throw new** IllegalStateException(e.getMessage(), e);   } **finally** {  buffer.clear();  }   }   *//反序列化* @Override **public** Object deserialize(InputStream inputStream) {  **try** {  Class clazz = isRpcDirect() ? MessageRequest.**class** : MessageResponse.**class**;  Object object = (Object) *objenesis*.newInstance(clazz);  Schema<Object> schema = *getSchema*(clazz);  ProtostuffIOUtil.*mergeFrom*(inputStream, object, schema);  **return** object;   } **catch** (Exception e) {  **throw new** IllegalStateException(e.getMessage(), e);  } } |

**3.2.3.2 Netty处理器实现网络传输**

**(1) 接收消息处理器ProtostuffReceiveHandler.java**

**代码清单10.2 ProtostuffReceiveHandler.java**

|  |
| --- |
| **public class** ProtostuffReceiveHandler **implements** NettyRpcReceiveHandler {   @Override  **public void** handle(Map<String, Object> map, ChannelPipeline pipeline) {   ProtostuffCodecHelper helper = **new** ProtostuffCodecHelper();  helper.setRpcDirect(**true**);  pipeline.addLast(**new** ProtostuffEncoder(helper));  pipeline.addLast(**new** ProtostuffDecoder(helper));  pipeline.addLast(**new** MessageReceiveHandler(map));    } } |

**(2) 发送消息处理器ProtostuffSendHandler.java**

**代码清单10.3 ProtostuffReceiveHandler.java**

|  |
| --- |
| **public class** ProtostuffSendHandler **implements** NettyRpcSendHandler {   @Override  **public void** handle(ChannelPipeline pipeline) {  ProtostuffCodecHelper helper = **new** ProtostuffCodecHelper();  helper.setRpcDirect(**false**);  pipeline.addLast(**new** ProtostuffEncoder(helper));  pipeline.addLast(**new** ProtostuffDecoder(helper));  pipeline.addLast(**new** MessageSendHandler());   } } |

**(3) 接收消息初始化器MessageReceiveChannelInitializer.java**

**代码清单10.4 MessageReceiveChannelInitializer.java**

|  |
| --- |
| **public class** MessageReceiveChannelInitializer **extends** ChannelInitializer<SocketChannel> {   **private** RpcSerializableProtocol **protocol**;   **private** RpcReceiveSerializableSelection **selection** = **null**;    **public** MessageReceiveChannelInitializer buildRpcSerializableProtocol(RpcSerializableProtocol protocol) {  **this**.**protocol** = protocol;  **return this**;  }    **public** MessageReceiveChannelInitializer(Map<String, Object> map) {  **selection** = **new** RpcReceiveSerializableSelection(map);  }    @Override  **protected void** initChannel(SocketChannel ch) **throws** Exception {   ChannelPipeline pipeline = ch.pipeline();  **selection**.select(**protocol**, pipeline);   } } |

**(3)发送消息初始化器MessageSendChannelInitializer.java**

**代码清单10.4 MessageSendChannelInitializer.java**

|  |
| --- |
| **public class** MessageSendChannelInitializer **extends** ChannelInitializer<SocketChannel> {   **private** RpcSerializableProtocol **protocol**;  **private** RpcSendSerializableSelection **frame** = **new** RpcSendSerializableSelection();   **public** MessageSendChannelInitializer buildRpcSerializableProtocol(RpcSerializableProtocol protocol) {  **this**.**protocol** = protocol;  **return this**;  }   @Override  **protected void** initChannel(SocketChannel ch) **throws** Exception {   ChannelPipeline pipeline = ch.pipeline();  **frame**.select(**protocol**, pipeline);   } } |

## 

## 3.3项目开发计划及阶段性完成情况

### **3.3.1 项目开发计划**

**表3.9 项目开发计划**

|  |  |  |
| --- | --- | --- |
| **任务名称** | **开始时间** | **计划完成时间** |
| **项目立项** | 2017年4月1日 | 5日 |
| 相关资料收集 | 2017年4月6日 | 5日 |
| 用户访问 | 2017年4月11日 | 2日 |
| 系统调研 | 2017年4月13日 | 2日 |
| 可行性分析 | 2017年4月15日 | 2日 |
| 制定项目计划 | 2017年4月17日 | 2日 |
| **知识积累** | 2017年4月24日 | 7日 |
| 建立开发环境 | 2017年4月27日 | 3日 |
| 相关知识学习 | 2017年5月1日 | 14日 |
| 相关工具学习 | 2017年5月15日 | 5日 |
| **需求分析** | 2017年5月20日 | 4日 |
| 需求建模 | 2017年5月25日 | 2日 |
| 编写需求规格书 | 2017年5月27日 | 2日 |
| **设计** | 2017年5月29日 | 4日 |
| 架构设计 | 2017年6月2日 | 6日 |
| 功能模块设计 | 2017年6月18日 | 6日 |
| 编写设计文档 | 2017年6月24日 | 2日 |
| 编码实现 | 2017年6月24日 | 28日 |
| **测试** | 2017年7月23日 | 3日 |
| 单元测试 | 2017年7月24日 | 1日 |
| 集成测试 | 2017年7月25日 | 1日 |
| 系统测试 | 2017年7月26日 | 1日 |
| **总结** | 2017年11月20日 | 14日 |
| 编写总结报告 | 2017年11月20日 | 14日 |

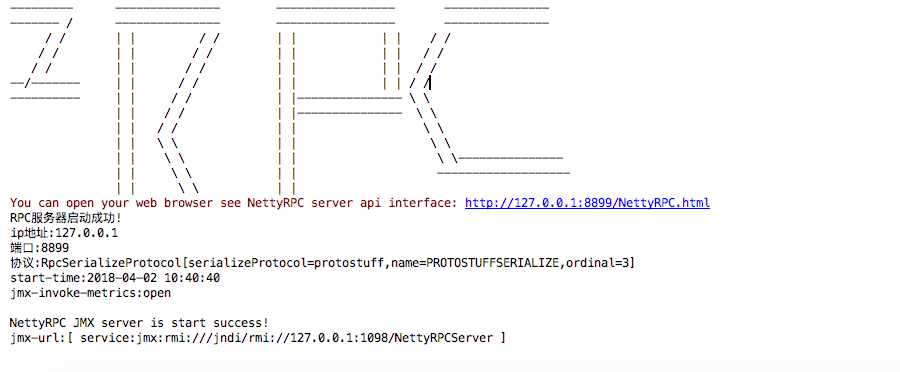
### **3.3.2 项目完成情况**

依据开发计划，项目的主体功能基本得到是实现，预期的目的已经达到。详细的完成情况可见如下详细说明：

**3.3.2.1 服务器端启动验证**

运行文件RpcServerStarter.java文件，在此文件中通过类ClassPathXmlApplicationContext类加载配置文件rpc-invoke-config-server.xml。开始运行之后，服务器将一直处于运行状态，将会一直监听来客户端的RPC请求：

我们可以看到：



**图3.9 启动服务端**

我们可以看到服务器启动后输出如下信息：

RPC服务器启动成功!

ip地址:127.0.0.1

端口:8899

协议:RpcSerializeProtocol[serializeProtocol=protostuff,name=PROTOSTUFFSERIALIZE,ordinal=3]

start-time:2018-04-02 10:40:40

jmx-invoke-metrics:open

NettyRPC JMX server is start success!

jmx-url:[ service:jmx:rmi:///jndi/rmi://127.0.0.1:1098/NettyRPCServer ]

**3.3.2.2 客户端发起RPC请求**

运行客户端测试类，发起RPC请求，客户端请求测试代码如下：

Student.java是我们需要进行操作的对象，需要是能够序列化的；

StudentOperation.java是操作Student对象的接口；

StudentOperationImpl是接口的实现；

StudentOperationTest.java是测试测试类；

**代码清单3.1 Student.java**

|  |
| --- |
| **public class** Student **implements** Serializable {   **private int studentId**;   **private** String **studentName**;   **private int score**;   **private** Date **schoolStart**;    **public int** getStudentId() {  **return studentId**;  }   **public void** setStudentId(**int** studentId) {  **this**.**studentId** = studentId;  }   **public** String getStudentName() {  **return studentName**;  }   **public void** setStudentName(String studentName) {  **this**.**studentName** = studentName;  }   **public int** getScore() {  **return score**;  }   **public void** setScore(**int** score) {  **this**.**score** = score;  }   **public** Date getSchoolStart() {  **return schoolStart**;  }   **public void** setSchoolStart(Date schoolStart) {  **this**.**schoolStart** = schoolStart;  }    @Override  **public** String toString() {  **return super**.toString();  } } |

**代码清单3.2 StudentOperation.java**

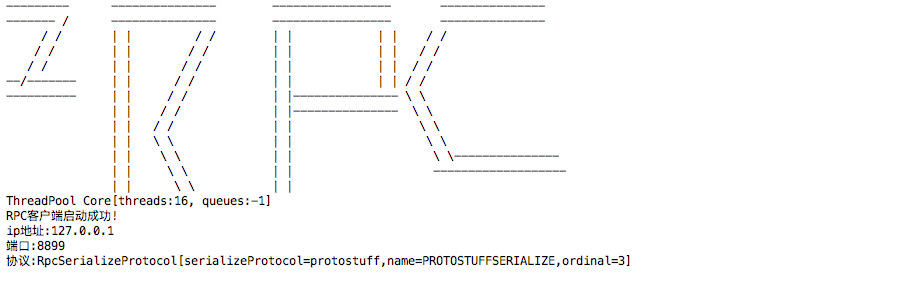
|  |
| --- |
| */\*\*  \* 操作Student对象的接口类  \*/* **public interface** StudentOperation {   *//保存* **int** save(Student student);   *//查询* **void** query(Student student);   *//判断分数大小* **boolean** judgeScore(Student student);  } |

**代码清单3.3 StudentOperationImpl.java**

|  |
| --- |
| */\*\*  \* 操作Student对象的实现类  \*/* **public class** StudentOperationImpl **implements** StudentOperation {   @Override  **public int** save(Student student) {  System.***out***.println(student + **" 已经被保存"**);  **return** 0;  }   @Override  **public void** query(Student student) {   **try** {  TimeUnit.***SECONDS***.sleep(3);  System.***out***.println(student + **" 已经被查询"**);   } **catch** (InterruptedException e) {  e.printStackTrace();   }   }   @Override  **public boolean** judgeScore(Student student) {  **if** (student.getScore() < 60) {  **throw new** RuntimeException(**"该学生考试没有及格"**);  } **else** {  System.***out***.println(**"该学生考试及格"**);  **return true**;  }   } } |

**代码清单3.4 StudentOperationTest.java**

|  |
| --- |
| **public class** StudentOperationTest {   **public static void** main(String[] args) {    ClassPathXmlApplicationContext classPathXmlApplicationContext =  **new** ClassPathXmlApplicationContext(**"classpath:rpc-invoke-config-client.xml"**);   StudentOperation studentOperation = (StudentOperation) classPathXmlApplicationContext.getBean(**"studentOperation"**);   Student student = **new** Student();  student.setStudentId(2222);  student.setStudentName(**"林肯公园"**);  student.setScore(77);   **int** result = studentOperation.save(student);   studentOperation.query(student);   System.***out***.println(**"调用RPC结果："** + result);   classPathXmlApplicationContext.destroy();  }   } |



**图3.10 启动客户端 发起RPC请求**

**3.3.2.3 服务器端返回RPC响应**

返回查看服务器端的输出情况：

ThreadPool Core[threads:16, queues:-1]

\*\*\*\*\*\*\*[ModuleListenerAdapter##exported]\*\*\*\*\*\*\*\*

EchoChainFilter##TRACE MESSAGE-ID:aa5e0c72-5555-467f-84d8-f84a500f4169

com.zou.rpc.services.pojo.Student@37ff6d95 已经被保存

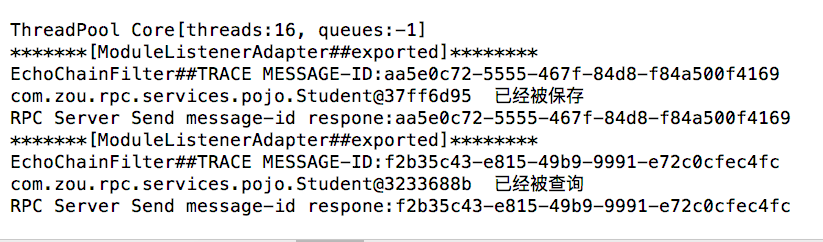
RPC Server Send message-id respone:aa5e0c72-5555-467f-84d8-f84a500f4169

\*\*\*\*\*\*\*[ModuleListenerAdapter##exported]\*\*\*\*\*\*\*\*

EchoChainFilter##TRACE MESSAGE-ID:f2b35c43-e815-49b9-9991-e72c0cfec4fc

com.zou.rpc.services.pojo.Student@3233688b 已经被查询

RPC Server Send message-id respone:f2b35c43-e815-49b9-9991-e72c0cfec4fc



**图3.11 服务器端返回RPC响应**

可以看到服务器返回了RPC响应。

**3.3.2.4 RPC调用的性能测试**

（1）使用加法计算了来模拟RPC调用

使用加法计算来大致衡量RPC调用的性能，性能指标是时间消耗。我们需要定义加法操作，使得多个RPC调用同时发生，然后计算其消耗时间。

使用1000个线程模拟1000个RPC调用：

**代码清单3.5 AddTest.java**

|  |
| --- |
| **public class** AddTest **implements** Runnable {   **private** CountDownLatch **countDownLatch1**;  **private** CountDownLatch **countDownLatch2**;  **private int task** = 0;  **private** AddCalculate **addCalculate**;   **public** AddTest(CountDownLatch countDownLatch1, CountDownLatch countDownLatch2, **int** task, AddCalculate addCalculate) {  **this**.**countDownLatch1** = countDownLatch1;  **this**.**countDownLatch2** = countDownLatch2;  **this**.**task** = task;  **this**.**addCalculate** = addCalculate;  }   **public void** run() {  **try** {  **countDownLatch1**.await();  **int** add = **addCalculate**.add(**task**, **task**);  System.***out***.println(**"加法计算结果 :["** + add + **"]"**);  } **catch** (InterruptedException ex) {  Logger.*getLogger*(AddTest.**class**.getName()).log(Level.***SEVERE***, **null**, ex);  } **catch** (InvokeTimeoutException ex) {  System.***out***.println(ex.getMessage());  } **finally** {  **countDownLatch2**.countDown();  }  } } |

性能测试类，可以输出RPC总共调用时间：

**代码清单3.6 PerformanceTest.java**

|  |
| --- |
| */\*\*  \* RPC 性能测试类  \* 加法测试，计算总时间  \*/* **public class** PerformanceTest {   **public static void** addOperation(AddCalculate addCalculate, **int** parallel) **throws** InterruptedException {   StopWatch sw = **new** StopWatch();  sw.start();   CountDownLatch countDownLatch1 = **new** CountDownLatch(1);  CountDownLatch countDownLatch2 = **new** CountDownLatch(parallel);   **for** (**int** index = 0; index < parallel; index++) {  AddTest addTest = **new** AddTest(countDownLatch1, countDownLatch2, index,addCalculate);  **new** Thread(addTest).start();  }   countDownLatch1.countDown();  countDownLatch2.await();  sw.stop();   System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  System.***out***.println();  System.***out***.println();  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*使用加法运算进行RPC性能测试 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  System.***out***.println(**"\*\*1000次RPC调用所消耗时间： 【 "** + sw.getTime() + **" 】 毫秒\*\*\*\*\*\*\*"**);   System.***out***.println();  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  System.***out***.println();  }    **public static void** addExecutor(AddCalculate calc, **int** parallel) **throws** InterruptedException {  PerformanceTest.*addOperation*(calc, parallel);  TimeUnit.***MILLISECONDS***.sleep(30);  }    **public static void** main(String[] args) **throws** Exception {   **int** parallel = 1000;  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**"classpath:rpc-invoke-config-client.xml"**);   **for** (**int** i = 0; i < 1; i++) {  *addExecutor*((AddCalculate) context.getBean(**"addCalc"**), parallel);   }   context.destroy();  }   } |

运行性能测试类，查看输出：



**图3.12 性能测试，1000次RPC时间消耗**

（2）使用乘法计算了来模拟RPC调用

同样使用RPC调用时间做为衡量依据，进行1000次的乘法调用。

**代码清单3.7 MultiTest.java**

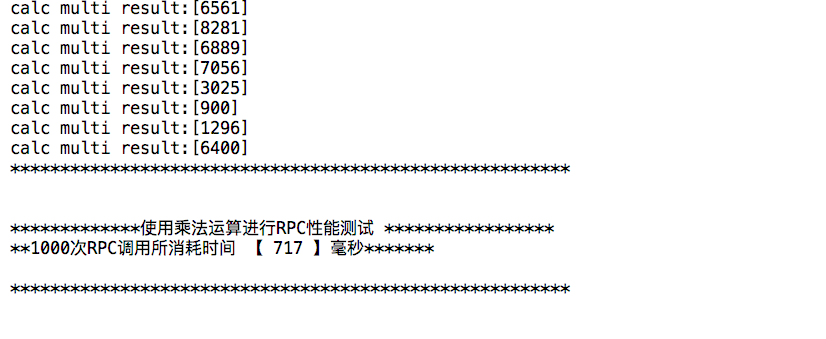
|  |
| --- |
| **public class** MultiTest **implements** Runnable {   **private** CountDownLatch **countDownLatch1**;  **private** CountDownLatch **countDownLatch2**;  **private int task** = 0;  **private** MultiCalculate **calc**;   **public** MultiTest(MultiCalculate calc, CountDownLatch countDownLatch1, CountDownLatch countDownLatch2, **int** task) {  **this**.**countDownLatch1** = countDownLatch1;  **this**.**countDownLatch2** = countDownLatch2;  **this**.**task** = task;  **this**.**calc** = calc;  }   **public void** run() {  **try** {  **countDownLatch1**.await();  **int** result = **calc**.multi(**task**, **task**);  System.***out***.println(**"乘法计算结果 :["** + result + **"]"**);  } **catch** (InterruptedException ex) {  Logger.*getLogger*(MultiTest.**class**.getName()).log(Level.***SEVERE***, **null**, ex);  } **catch** (InvokeTimeoutException ex) {  System.***out***.println(ex.getMessage());  } **finally** {  **countDownLatch2**.countDown();  }  } } |

性能测试类，可以输出RPC总共调用时间：

**代码清单3.8 PerformanceTest2.java**

|  |
| --- |
| *\*\*  \* RPC 性能测试类  \* 乘法调用测试  \*/* **public class** PerformanceTest2 {   **public static void** multiOperate(MultiCalculate calc, **int** parallel) **throws** InterruptedException {  StopWatch sw = **new** StopWatch();  sw.start();   CountDownLatch signal = **new** CountDownLatch(1);  CountDownLatch finish = **new** CountDownLatch(parallel);   **for** (**int** index = 0; index < parallel; index++) {  MultiTest multiTest = **new** MultiTest(calc, signal, finish, index);  **new** Thread(multiTest).start();  }   signal.countDown();  finish.await();  sw.stop();   System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  System.***out***.println();  System.***out***.println();  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\* 使用乘法运算进行RPC性能测试 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  System.***out***.println(**"\*\*1000次RPC调用所消耗时间 【 "** + sw.getTime() + **" 】毫秒\*\*\*\*\*\*\*"**);   System.***out***.println();  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  System.***out***.println();    }    **public static void** addExecutor(MultiCalculate multiCalculate, **int** parallel) **throws** InterruptedException {  PerformanceTest2.*multiOperate*(multiCalculate, parallel);  TimeUnit.***MILLISECONDS***.sleep(30);  }    **public static void** main(String[] args) **throws** Exception {   **int** parallel = 100;  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**"classpath:rpc-invoke-config-client.xml"**);   **for** (**int** i = 0; i < 1; i++) {  *addExecutor*((MultiCalculate) context.getBean(**"multiCalc"**), parallel);   }   context.destroy();  }   } |

运行性能测试类，查看输出：



**图3.13 性能测试，1000次RPC时间消耗**

**3.3.2.5 序列化协议的性能比较**

本项目使用了多种序列化机制，本节主要这些序列化机制的性能进行简单的比较，仍然使用RPC调用时间进行衡量，仅仅改变序列化机制，对比相同RPC调用的时间。

首先，我们需要在配置文件中配置RPC序列化机制，配置的的代码如下：

**代码清单3.9 序列化机制配置文件**

|  |
| --- |
| <**zrpc:reference id="addCalc" interfaceName="com.zou.rpc.services.AddOperation"  protocol="JDKSERIALIZE" ipAddr="${rpc.server.addr}"**/>   <**zrpc:reference id="multiCalc" interfaceName="com.zou.rpc.services.MultiOperation"  protocol="JDKSERIALIZE" ipAddr="${rpc.server.addr}"**/> |

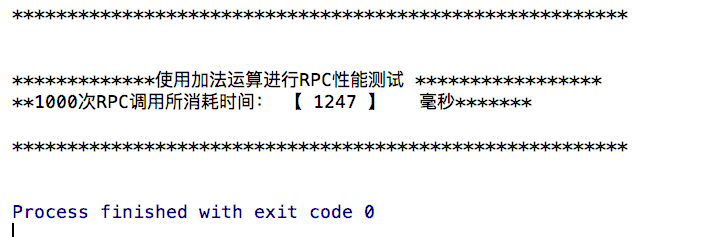
我们可以在上述配置文件中配置所测试的序列化机制代码。

（1）使用JDK原生的序列化机制

配置文件代码：

**代码清单3.10 JDK序列化机制配置文件**

|  |
| --- |
| <**zrpc:reference id="addCalc" interfaceName="com.zou.rpc.services.AddOperation"  protocol="JDKSERIALIZE" ipAddr="${rpc.server.addr}"**/>   <**zrpc:reference id="multiCalc" interfaceName="com.zou.rpc.services.MultiOperation"  protocol="JDKSERIALIZE" ipAddr="${rpc.server.addr}"**/> |



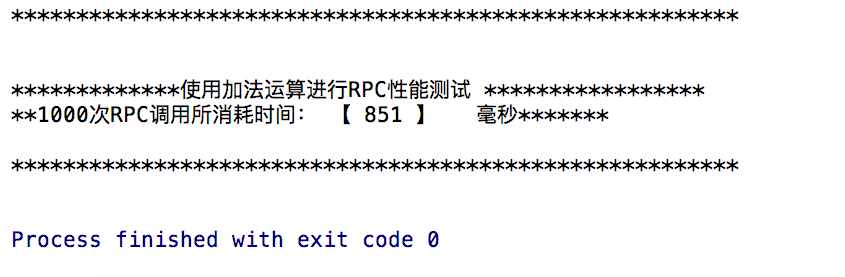
**图3.14 JDK序列化时间消耗**

（2）使用Protobuf序列化机制

配置文件代码：

**代码清单3.11 Protobuf序列化机制配置文件**

|  |
| --- |
| <**zrpc:reference id="addCalc" interfaceName="com.zou.rpc.services.AddOperation"  protocol="JDKSERIALIZE" ipAddr="${rpc.server.addr}"**/>   <**zrpc:reference id="multiCalc" interfaceName="com.zou.rpc.services.MultiOperation"  protocol="JDKSERIALIZE" ipAddr="${rpc.server.addr}"**/> |



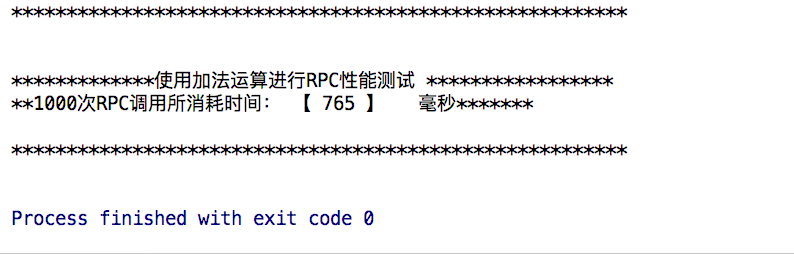
**图3.14 Protobuf序列化时间消耗**

（3）使用Hessian序列化机制

配置文件代码：

**代码清单3.12 Hessian序列化机制配置文件**

|  |
| --- |
| <**zrpc:reference id="addCalc" interfaceName="com.zou.rpc.services.AddOperation"  protocol="HESSIAN" ipAddr="${rpc.server.addr}"**/>   <**zrpc:reference id="multiCalc" interfaceName="com.zou.rpc.services.MultiOperation"  protocol="HESSIAN" ipAddr="${rpc.server.addr}"**/> |



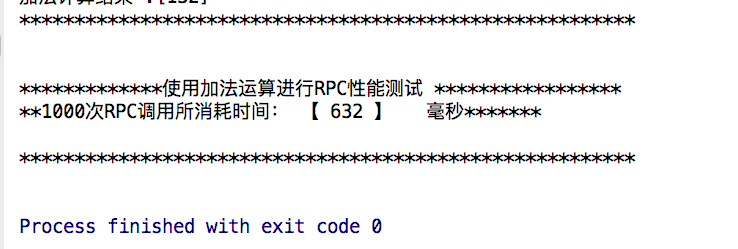
**图3.15 Hessian序列化时间消耗**

（4）使用Kryo序列化机制

配置文件代码：

**代码清单3.13 Kryo序列化机制配置文件**

|  |
| --- |
| <**zrpc:reference id="addCalc" interfaceName="com.zou.rpc.services.AddOperation"  protocol="KRYO" ipAddr="${rpc.server.addr}"**/>   <**zrpc:reference id="multiCalc" interfaceName="com.zou.rpc.services.MultiOperation"  protocol="KRYO" ipAddr="${rpc.server.addr}"**/> |



**图3.15 Kryo序列化时间消耗**

总结：根据以上实验结果，初步可以判断使用Java原生的序列化机制性能比使用开源序列化框架性能要低。但是，值得注意的是，本方法并不是一个完善的比较方式，只能作为一个初步的判断方式。

## 3.4本章小结

本章主要是对项目的基本工作和开发情况进行了介绍，包括项目的架构设计、概要设计和详细设计等等。

# 讨论与体会

## 4.1对项目开发过程的体会

本项目的灵感来源主要是在实习过程中使用了很多的RPC框架进行项目开发，用的最多的是Facebook的Thrift和Google的grpc开源框架。在使用这些框架的同时，对其强大的功能留下了深刻的印象，由于现在国内外各大互联网公司都在大量的使用RPC框架进行各个组件之间的通信，其重要性是不言而喻的。

另外，由于个人的方向选择，对企业开发中的高并发架构非常有兴趣，所以去学习了一些和并发处理相关的技术。其中不得不提的就是异步的网络应用框架Netty。Netty是一门设计非常精巧的网络应用框架，已经大量的应用在了各个企业级应用的开发之中。常见的比如各个App应用消息的推送以及各大公司内部项目之间的通信等等。

正是基于上面两个重要的原因，才有了本项目的灵感来源。要完成本项目，因为是使用Java语言实现的，所以熟悉Java语言是最基本的要素；在本项目中，大量的使用到了Netty作为通讯工具，因此熟悉和掌握Netty相关的开发技术是非常必要的条件；此外，因为没有开发过类似的项目，所以前期需要很好的了解一些相关的开源项目，好在gprc和Thrift等都是开源的，并且有很多的博客文章对其源代码进行解释，所以能够降低很多的知识获取成本。

就整个项目开发过程而言，前期的知识储备是最重要的。虽然前期花了大量的时间去学习Netty，写了很多的Demo实例和做了很多笔记，但是在实际的开发过程中，还是遇到了很多的困难。比如说Netty的主从线程模型，在客户端和服务器端的模式是不相同的，由于服务器端使用的是主从线程模型，如果不能很好的去理解主从线程模型，那么Netty所起的重要作用也将不能理解，那么对项目性能的考量就会失去重要的参照。

除此之外，项目开发过程中还用到一些重要的设计模式，比如单例模式。虽然这个模式非常简单，但是在使用的时候也需要注意很多的问题，其中最重要的莫过于线程安全的问题。在使用懒汉式单例模式的时候，在多线程的情况下，一般都会出现多线程的问题。在本项目的开发过程中，同样有这样的问题存在，并真实的引发了程序运行过程中异常的发生。不过，对于这个问题已经有很多的解决方式，本项目中使用的是最常用的双检查锁机制进行线程安全问题放入处理。

总而言之，项目开发是一个很长的过程。前期需要进行大量的知识储备，开发过程中需要解决很多的技术问题，开发完成后还要进行充分的性能和功能测试以及开发文档的编写。不过，项目开发的乐趣在于，能够将感兴趣的主题或者有意思的想法变为现实，是一件很有成就感的事情。这也就是生活的乐趣，不断学习，不断提高。

## 4.2对项目的评价

相对于其他已有的RPC框架，本项目的创新点如下：

1. **多种序列化协议可供选择**

多种序列化协议增强了RPC框架的灵活性，丰富了适用场景。

本项目使用了多种序列化协议进行尝试，包括Java原生的序列化协议，以及Hessian，kryo等第三方序列化框架。Java原生的序列化有其独特的优势，比如说使用简单，只需要继承Java提供的接口，并且，在Netty中使用的编码解码处理器就是基于这种原生的序列化机制的。但是，原生的序列化机制在许多方面并不是最好的，比如在传输数据的时候，会消耗很多的带宽。所以，在原生的序列化机制之外，选择其他优秀的序列化框架帮助序列化是很好的方式。虽然在编码量和编码难度上可能有所提高，系统设计的复杂性也会增大，但是却能让系统提供更好的服务和更优秀的性能。

1. **选择Netty进行网络通信**

Netty是一个异步的网络通讯框架，为什么称为异步的呢？是因为它是基于Java NIO/AIO的。原因在于，Java提供的NIO机制，在编程难度上比较高，要想在项目开发中国大规模的应用是比较困难的。一是编写高质量的NIO代码则需要具有非常多开发经验的编程人员，二是Java NIO也并非是没有缺点。正是基于以上原因，才会出现Netty等网络通讯框架。选择Netty的原因在于，它本身是对NIO的再次封装，让开发者能够更好的进行编程开发，并且在NIO缓冲区等特性上进行了很好的优化，能够提供更好的性能。。

1. **使用池化技术。**

池化技术在项目开发的很多方面都得到了大量的应用。比如线程池技术，对象池技术，数据库连接池技术。本项目使用了对象池技术，主要是为了降低重复创建对象和销毁对象的开销。

## 4.3本章小结

本章主要是对整个项目开发过程进行了简单的回顾。主要是项目开发过程中的体会，用到了哪些技术，遇到了哪些问题以及对怎样解决这些问题产生的思考。

# 小结

针对目前的开发成果，可以说基本实现了RPC所需功能。但是，还有以下方面可以进行改善：

1.项目的性能研究

目前RPC能够实现基本的功能，虽然采取了很多的性能优化方式，但是对其具体变现还不能有数字化的衡量，所以如果能够直观的看到一些RPC性能参数将会是下一步项目开发需要进行的事情。

2.项目的可视化

本项目和其他系统的一个区别在于，本项目提供了比较少的可视化界面，而其他项目可能会有很多的界面展现。这其实并不影响项目的功能，对于很多的第三方框架来讲，都是给项目开发提供基本服务的，本项目也是这样一个目的。但是，这并不是说可视化界面没有用，在一些方面比如监控参数等功能上，会让使用者有更好的使用体验。

3.不同语言间的调用

由于时间和技术能力的限制，本项目暂时没有实现跨语言间的RPC调用，但是对于使用多语言的项目来讲，跨语言调用有时候是必要的。所以，能够实现跨语言调用将会使得项目功能更加强大，对于使用者来讲，也会更加便利。

# 参考资料

[1]符凯. 软件架构重构技术的研究与实现[D].西北大学,2007．

[2] 田健. 基于软件工程的信息网络系统监理研究[D].天津大学,2016.

[3] ]S.Kwong. Design and Implementation of Remote Procedure Call Based on ISO/OSI Reference Model[A]. IEEE Beijing Section、Northern Jiaotong University.Proceedings of 1993 IEEE Region 10 Conference on “Computer,Communication,Control and Power Engineering”(TENCON'93/Beijing) Volume 1 of 5[C].IEEE Beijing Section、Northern Jiaotong University:,1993:5.

[4] Shuai li. Dubbo's Serialization Protocol Extension and Optimization of Its RPC Protocol Thrift[A]. The Institute of Electrical and Electronics Engineers、IEEE Beijing Section.Proceedings of 2017 IEEE 8th International Conference on Software Engineering and Service Science[C].The Institute of Electrical and Electronics Engineers、IEEE Beijing Section:,2017:6.

[5] 崔晓旻. 基于Netty的高可服务消息中间件的研究与实现[D].电子科技大学,2014.

[6] 杨健. 基于J2EE平台的对象持久性研究[D].武汉理工大学,2005．

[7] ]Yongchang Ren. Construction of SSI Framework Based on MVC Software Design Model[A]. International Informatization and Engineering Associations、Atlantis Press.Proceedings of 2015 4th International Conference on Mechatronics,Materials,Chemistry and Computer Engineering (ICMMCCE 2015)[C].International Informatization and Engineering Associations、Atlantis Press:,2015:5.

[8] 段培勇. Spring轻量级框架在项目管理信息系统开发中的应用研究[D].西南交通大学,2007.．

[9] 张立超. 基于Java的IoC容器的设计与实现[D].吉林大学,2009.

[10] Nannan Zhang. Development on Construction Project Cost System Based on Spring[A]. 信息化与工程国际学会.Proceedings of 2016 3rd International Conference on Materials Engineering,Manufacturing Technology and Control(ICMEMTC 2016)[C].信息化与工程国际学会:,2016:6.

[11] 王博,陈莉君.Hadoop远程过程调用机制的分析和应用[J].西安邮电学院学报,2012,17(06):74-77..

[12] 肖俊,何炎祥.一种基于网格环境的远程过程调用系统的设计与分析[J].计算机应用,2005(01):176-179

[13] 罗玲,白晓颖.基于XML的RPC技术分析[J].计算机科学,2004(02):167-170+174.

．

[14] 李小白,刘水文,李斐.远程过程调用的体系结构以及相关参数[J].科技广场,2009(01):50-52.

[15] 陈良宽,王雅红.通用远程过程调用的设计与实现[J].小型微型计算机系统,1996(02):33-37.

[16] 邹晓辉. 基于多线程和RPC技术的并行C语言的设计和实现[D].吉林大学,2005.

[17] 朱浩,徐丽,吴泉源.远程过程调用的实现、应用与问题[J].计算机工程与科学,1995(03):18-25．

[18] 郑丽英,李全兵.一种新的基于RPC的分布式开发模型[J].兰州交通大学学报,2004(01):83-86．

[19] 王纪臣. 异步RPC的设计与实现[D].吉林大学,2005．

[20] 周俊,王芳,李阳,明亮,万勇.用户态RPC协议分析及其多线程优化[J].计算机研究与发展,2012,49(S1):191-195．

[21] 陈优子,陈俊延,王彤.基于远程过程调用和壁垒同步的分布式离散事件仿真模型[J].计算机应用,2011,31(05):1413-1416．

[22] 王静茹,艾洪福,林和平,李迎斌.基于远程过程调用技术自定制服务研究及其应用[J].计算机技术与发展,2008(09):229-231．

[23] 赵小建,方康玲.基于流套接字的RPC技术研究与应用[J].武汉科技大学学报(自然科学版),2007(01):68-70．

[24] 范宝德,刘惊雷.基于事务对象的RPC技术应用[J].计算机工程与应用,2004(19):101-103．

[25] 李璞,张玲,胡术,潘倩,李艳.多线程环境下Reactor模式的研究与实现[J].网络新媒体技术,2017,6(02):52-57.

[26] 肖凯,张玉泉,陶智勇.基于Reactor模式的即时通信服务器的设计与实现[J].信息技术,2017(03):124-127+132.

[27] 佘晓彬,韩国强.基于NIO和设计模式的应用服务器的设计与实现[J].微计算机息,2009,25(21):91-93.

# 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得四川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谢意。

本学位论文成果是本人在四川大学读书期间在导师指导下取得的，论文成果归四川大学所有，特此声明。

学位论文作者（签名） \_\_\_\_\_\_\_\_\_\_\_\_ 2017年12月 15 日

论文指导教师（签名） \_\_\_\_\_\_\_\_\_\_\_\_ 2017年12月 15 日

# 致 谢

首先感谢的是父母和亲人的支持，能够让我接受很好的教育，能够在大学校园中学习先进的知识。其次，要感谢指导老师左航老师，是她在项目开发过程中时刻的关注项目进度，并进行疑难解答，在论文的撰写过程中，提供耐心的指导和帮助。同时，要感谢我在实习的公司和导师，让我有机会接触企业级开发过程中用到的技术，奠定了本项目的技术基础。最后，感谢在大学生活中遇到的所有对我有帮助和启发的老师和同学。

# 附录1 任务申请书

现在已经迈入了互联网时代，各大互联网巨头拥有很多用户量巨大的产品。这些产品往往由很多项目组成，这些项目或者服务往往也需要进行彼此调用，或者说通信。所以，各种远程过程调用技术得到了很大的发展，其中著名的有Google开源的grpc框架和Facebook开源的Thrift框架。

RPC技术通俗的来讲就是，即使不处于同一台主机，服务之间也能够进行相互调用。简单的说来就是，这一台主机上的代码能够调用那一台主机上的函数或者方法。因为不是出于同一台机器，所以拥有的内存空间是不同的，这就需要进行数据的传输。RPC解决的主要就是这个问题。

在远程过程调用中，因为数据要通过网络进行传输，所以需要进行消息的序列化，将其变为字节之后再传输到另一端。到达另一台机器之后，需要将字节再转换为消息对象，这就是消息的反序列化。在本项目中，设想使用多种序列化协议，比如使用Java提供的原生序列化机制，以及现在开源的第三方序列化框架。Java的原生序列化机制可能对于开发人员来讲比较简单，但是效率会比较低，在性能上可能也会有缺陷，所以猜想使用第三方序列化框架可能解决这些问题。

在实际的项目开发过程中，使用RPC可以说是非常常见的，其作用也是非常巨大的。正是基于在实际的开发工作中大量的使用RPC框架和Netty，才有了开发一个RPC服务器的想法。对于开发人员来讲，往往能够使用某种技术是往往不够的，最好的是能够了解其工作机制和原理，这样才能对技术的深层次应用更有信心。

# 附录2 可行性研究报告

## 引言

**1.1编写目的**

目的是为了研究对与项目功能的实现是否具有可行性。

预期读者：本文档的预期读者是开发工程师、测试工程师、RPC框架的使用者、RPC框架学习人员等。

**1.2背景**

开发软件名称：基于Netty的RPC框架的设计与实现

项目任务提出者：邹绿洲

项目开发者：邹绿洲

用户：需要进行RPC服务的开发人员、对RPC开发有兴趣的研究人员

实现软件单位：邹绿洲

项目与其他软件，系统的关系：

本项目使用Java语言进行编写，网络通讯框架使用的是Netty；消息序列化和反序列化使用的是Java提供的原生序列化、Kryo、Hessian等，对象池技术使用的是Apache commons-poo2包提供的开源框架；项目构建工具使用的是gradle。

**1.3定义**

无。

**1.4参考资料**

《软件文档写作教程》电子工业出版社 主编：马平、黄冬梅

## 可行性研究

**2.1 消息序列化和反序列化的可行性**

首先Java语言本身提供了序列化和反序列化机制，提供了简单易实现的接口，对于开发人员来讲比较简单。其次，对于原生序列化效率不高等问题，可以使用第三方的开源序列化框架，这些序列化框架本身得到了很多项目的检验，还有有完整的文档可供学习。所以，对于消息的序列化和反序列化来讲是可以实现的。

**2.2 Netty进行网络通讯的可行性**

Netty作为目前优秀的网络应用框架，其本身是基于NIO的，NIO的缺点在于开发人员很难将其作用完美的发挥出来，而Netty相当于再次封装了NIO，便于开发人员的使用。现在很多著名的开源项目都是基于Netty的，比如大名鼎鼎的Spark，Dubbo等，这些项目都有完善的文档和代码可供学习，这为Netty的实践提供了宝贵的学习资料。另外，在项目的开发之前，已经在实际的工作中使用过Netty，对其源码和机制也有一些了解和研究。

**2.3 池化技术的可行性**

池化技术包括线程池、对象池、数据库连接池等等。本项目所依赖的对象池技术是Apache commons-poo2包提供的，在效率上得到很多项目的检验，使用也较为简单，能够为对象的池化提供比较好的保障。

# 附录3 项目开发计划

## 1 引言

**1.1编写目的**

目的是对项目进度进行安排和计划。

预期读者：本文档的预期读者是开发工程师、测试工程师、RPC框架的使用者、RPC框架学习人员等。

**1.2背景**

开发软件名称：基于Netty的RPC框架的设计与实现

项目任务提出者：邹绿洲

项目开发者：邹绿洲

用户：需要进行RPC服务的开发人员、对RPC开发有兴趣的研究人员

实现软件单位：邹绿洲

项目与其他软件，系统的关系：

本项目使用Java语言进行编写，网络通讯框架使用的是Netty；消息序列化和反序列化使用的是Java提供的原生序列化、Kryo、Hessian等，对象池技术使用的是Apache commons-poo2包提供的开源框架；项目构建工具使用的是gradle。

**1.3定义**

无。

**1.4参考资料**

《软件文档写作教程》电子工业出版社 主编：马平、黄冬梅

## 2 项目开发计划

**表3.1 项目开发计划表**

|  |  |  |
| --- | --- | --- |
| **任务名称** | **开始时间** | **计划完成时间** |
| **项目立项** | 2017年12月1日 | 3日 |
| 相关资料收集 | 2017年12月4日 | 5日 |
| 用户访问 | 2017年12月9日 | 2日 |
| 系统调研 | 2017年12月11日 | 2日 |
| 可行性分析 | 2017年12月13日 | 2日 |
| 制定项目计划 | 2017年12月15日 | 2日 |
| **知识积累** | 2017年12月17日 | 7日 |
| 建立开发环境 | 2017年12月23日 | 3日 |
| 相关知识学习 | 2017年12月26日 | 14日 |
| 相关工具学习 | 2018年1月11日 | 5日 |
| **需求分析** | 2018年1月16日 | 4日 |
| 需求建模 | 2018年1月20日 | 2日 |
| 编写需求规格书 | 2018年1月22日 | 2日 |
| **设计** | 2018年1月24日 | 4日 |
| 架构设计 | 2018年1月28日 | 6日 |
| 数据设计 | 2018年2月04日 | 5日 |
| 界面设计 | 2018年2月09日 | 3日 |
| 功能模块设计 | 2018年2月12日 | 6日 |
| 编写设计文档 | 2018年2月18日 | 2日 |
| 编码实现 | 2018年2月20日 | 28日 |
| **测试** | 2018年3月17日 | 3日 |
| 单元测试 | 2018年3月20日 | 1日 |
| 集成测试 | 2017年3月21日 | 1日 |
| 系统测试 | 2017年3月22日 | 1日 |
| **总结** | 2017年3月24日 | 12日 |
| 编写总结报告 | 2018年4月6日 | 12日 |

# 附录5 需求规格说明书

## 1引言

**1.1编写目的**

编写规格需求说明书的目的是对系统的功能需求进行梳理，在项目的开发过程中依据说明书进行系统的概要设计和详细设计。

预期读者：本文档的预期读者是开发工程师、测试工程师、RPC框架的使用者、RPC框架学习人员等

**1.2背景**

开发软件名称：基于Netty的RPC框架的设计与实现

项目任务提出者：邹绿洲

项目开发者：邹绿洲

用户：需要进行RPC服务的开发人员、对RPC开发有兴趣的研究人员

实现软件单位：邹绿洲

项目与其他软件，系统的关系：

本项目使用Java语言进行编写，网络通讯框架使用的是Netty；消息序列化和反序列化使用的是Java提供的原生序列化、Kryo、Hessian等，对象池技术使用的是Apache commons-poo2包提供的开源框架；项目构建工具使用的是gradle。

**1.3定义**

无。

**1.4参考资料**

《软件文档写作教程》电子工业出版社 主编：马平、黄冬梅

## 2任务概述

**2.1目标**

基于Netty设计一个高效的RPC服务器，选择多种序列化协议进行消息的序列化和反序列化，使用对象池技术等池化技术改善RPC服务器的性能。

**2.2用户的特点**

需要在项目开发过程中使用RPC服务器的开发人员

**2.3假定和约束**

无。

## 3需求规定

**3.1对功能的规定**

* **JDK原生机制实现消息序列化和反序列化**

通过JDK原生的序列化机制实现消息的序列化和反序列化。

* **第三方序列化框架实现消息的序列化和反序列化**

通过引入第三方序列化框架，依据其序列化和反序列化规则，进行消息的序列化和反序列化。

* **Netty进行网络通讯**

通过网络应用框架进行Netty进行通讯，编写特定的处理器，集成编码和解码处理器。

* **对象池池化对象**

对于一些需要频繁创建和销毁的对象，使用Apache commons-poo2包提供的池化对象进行池化，提高性能。

* **RPC测试**

需要编写客户端测试代码，对RPC提供的功能进行测试，并查看输出结果。

**3.2对性能的规定**

性能需求需要考虑很多的方面，包括系统的可靠性、可移植性、可扩展性等等。对于一个软件系统来讲，下面性能需求是需要考虑的因素：

1. **明确性：**系统的功能模块是明确的，通过功能模块结构图能够快速的了解整个系统的功能。
2. **可扩展性：**在设计系统结构时，除了考虑已有的功能实现，还需要考虑系统以后的项目发展，对于系统的应用方向，应该充分的考虑其扩展的能力，已满足不同的项目需求。
3. **可移植性：**一个系统往往不止应用于一个平台，所以开发的系统往往需要考虑移植性，这样才能使其应用范围更广。幸运的是，Java本身是平台无关的的语言，本项目使用Java语言开发。

**3.3输入输出要求**

输入要求：需要调用的远程函数或者方法，发起请求

输出要求：输出RPC响应结果

**3.4数据管理能力要求（针对软件系统）**

无。

**3.5故障处理要求**

如果RPC调用出现异常，需要抛出异常。

## 4运行环境规定

**4.1设备**

无。

**4.2支持软件**

运行环境：JDK1.8、Netty 4.x

开发平台：IntelliJ IDEA

开发语言：Java

**4.3接口**

Netty网络通讯接口，第三方序列化框架接口

**4.4控制**

无。

# 附录6 用户手册概要

## 1.引言

**1.1编写目的**

编写本手册的目的是帮助本系统的使用者明确该系统功能，讲解其使用方式，便于使用者使用。

预期读者：本文档的预期读者是开发工程师、测试工程师、RPC框架的使用者、RPC框架学习人员等。

**1.2背景**

开发软件名称：基于Netty的RPC框架的设计与实现

项目任务提出者：邹绿洲

项目开发者：邹绿洲

用户：需要进行RPC服务的开发人员、对RPC开发有兴趣的研究人员

实现软件单位：邹绿洲

项目与其他软件，系统的关系：

本项目使用Java语言进行编写，网络通讯框架使用的是Netty；消息序列化和反序列化使用的是Java提供的原生序列化、Kryo、Hessian等，对象池技术使用的是Apache commons-poo2包提供的开源框架；项目构建工具使用的是gradle。

**1.3定义**

无。

**1.4参考资料**

《软件文档写作教程》电子工业出版社 主编：马平、黄冬梅

## 2用途

**2.1功能**

* **JDK原生机制实现消息序列化和反序列化**

通过JDK原生的序列化机制实现消息的序列化和反序列化。

* **第三方序列化框架实现消息的序列化和反序列化**

通过引入第三方序列化框架，依据其序列化和反序列化规则，进行消息的序列化和反序列化。

* **Netty进行网络通讯**

通过网络应用框架进行Netty进行通讯，编写特定的处理器，集成编码和解码处理器。

* **对象池池化对象**

对于一些需要频繁创建和销毁的对象，使用Apache commons-poo2包提供的池化对象进行池化，提高性能。

* **RPC测试**

需要编写客户端测试代码，对RPC提供的功能进行测试，并查看输出结果。

**2.2性能**

无。

## 3运行环境

**3.1硬设备**

无

**3.2支持软件**

运行环境：JDK1.8、Netty 4.x

开发平台：IntelliJ IDEA

开发语言：Java

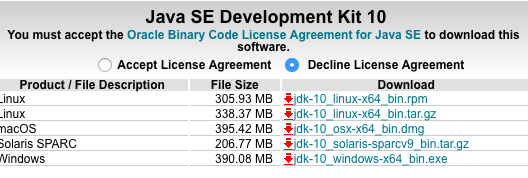
**3.3数据结构**

无。

## 4使用过程

**4.1安装与初始化**

**1.下载并安装JDK**

****

**图6.1**

**1）进入Oracle官网下载JDK**

**2）双机程序安装JDK**

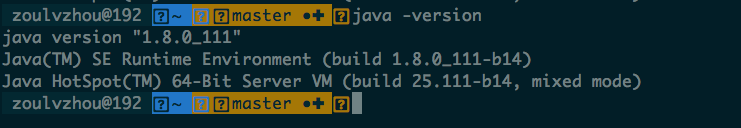
**3）配置环境变量**

打开“终端”，输入命令： vi ~/.bash\_profile，编辑内容如下

export JAVA\_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0\_111.jdk/Contents/Home

HOME%\bin**;**%JAVA\_HOME%\jre\bin**;**。

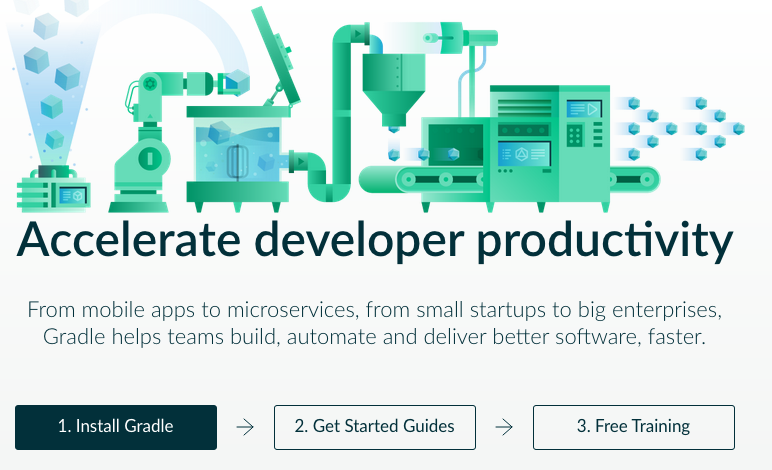
检测Java是否安装成功：在终端中：java –version，如有以下输出，则表示成功配置JDK。

****

**图6.2**

**2.下载并安装gradle**

**1）进入gradle官网https://gradle.org/，便可以进行下载安装**

****

**图6.2**

**2）配置gradle环境变量**

打开“终端”，输入命令： vi ~/.bash\_profile，编辑内容如下

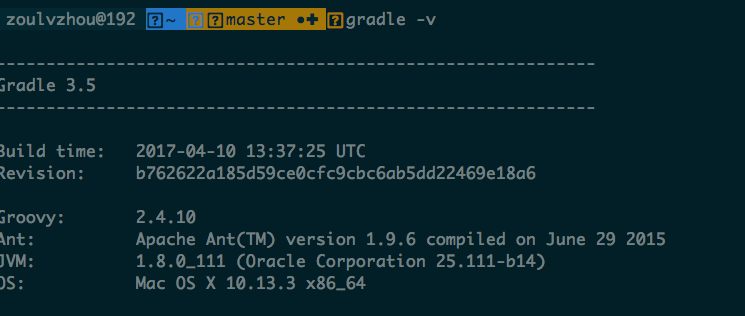
GRADLE\_HOME=/Users/zoulvzhou/gradle-3.5

export GRADLE\_HOME

export PATH=$PATH:$GRADLE\_HOME/bin

**3）测试gradle是否安装成功**

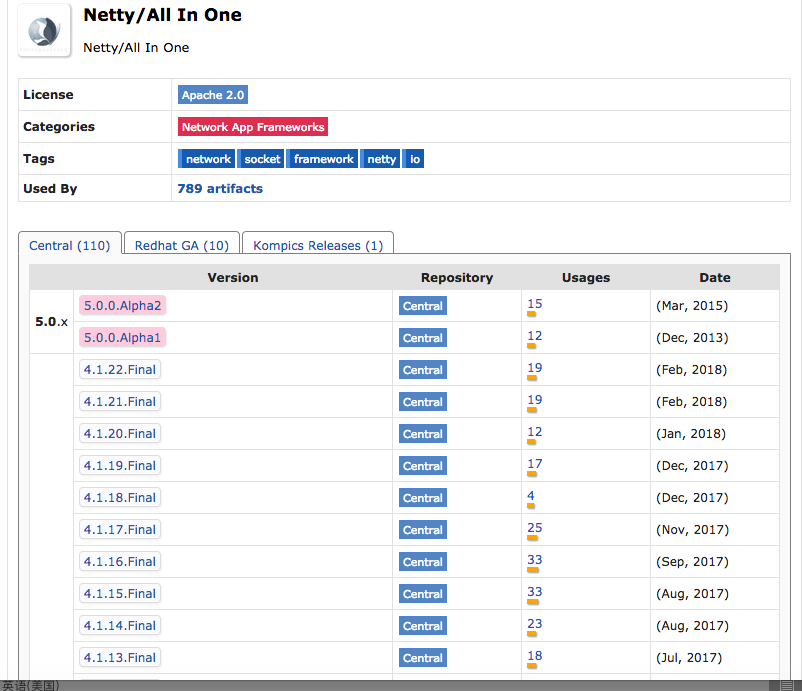
在“终端”输入gradle –v

****

**图6.3**

**3.使用gradle引入Netty**

**1）进入Maven仓库官网http://mvnrepository.com/，搜索Netty**

****

**图6.4**

**2）在build.gradle文件中引入如下代码**

// https://mvnrepository.com/artifact/io.netty/netty-all

compile group: 'io.netty', name: 'netty-all', version: '4.1.22.Final'

# 附录7 概要设计说明书

## 1引言

**1.1编写目的**

对系统的架构进行概要设计，方便开发人员把握基本架构以及其他相关人员快速了解系统架构。

预期读者：本文档的预期读者是开发工程师、测试工程师、RPC框架的使用者、RPC框架学习人员等。

**1.2背景**

开发软件名称：基于Netty的RPC框架的设计与实现

项目任务提出者：邹绿洲

项目开发者：邹绿洲

用户：需要进行RPC服务的开发人员、对RPC开发有兴趣的研究人员

实现软件单位：邹绿洲

项目与其他软件，系统的关系：

本项目使用Java语言进行编写，网络通讯框架使用的是Netty；消息序列化和反序列化使用的是Java提供的原生序列化、Kryo、Hessian等，对象池技术使用的是Apache commons-poo2包提供的开源框架；项目构建工具使用的是gradle。

。

**1.3定义**

无。

**1.4参考资料**

《软件文档写作教程》电子工业出版社 主编：马平、黄冬梅

## 2总体设计

**2.1系统用例图设计**

整个系统完成的功能详见如下用例图：



**图7.1 系统用例图**

**2.2 系统流程设计**

* **系统概要流程设计步骤如下**

1）服务端加载配置文件，此后将一直处于运行状态，一直等待客户端发起请求。

2）客户端发起请求，请求消息中应该包含请求类名，方法名，属性名等基本信息。

3）构建好请求消息之后，通过序列化框架对请求消息进行序列化。

4）序列化完成之后，通过Netty传输字节数据到服务器端。

5）服务器端将字节数据反序列化出来，得到相应的请求信息，进行处理之后返回给客户端。

6）客户端接收到服务器返回的响应信息，解析出响应结果。另外还会得到此次RPC过程的一些信息，比如调用是否成功，调用失败的错误消息等。

* **系统具体流程设计**

**注册阶段**

通过验证的合法投票者在网站进行匿名注册，具体步骤如下：

1）按照提前生成好的公私钥对系统进行初始化，其中公钥集合L={y1,y2,……,yn}由所有具备选举几个者公钥组成。

2）投票者Vi(1≤i≤n)利用关联环签名生成算法对消息m进行关联环签名，生成签名(C,S1,S2,……,Sn,Y)，同时为自己生成另外一对基于RSA签名体制的公私钥对，并利用其中的私钥对消息m进行普通的个体签名，最终将消息，两个签名以及自己生成的公钥一起发送给投票中心。

3）投票中心按照关联环签名验证算法，首先验证签名(C,S1,S2,……,Sn,Y)的正确性。若正确，则检查该签名中的Y值是否已存在于注册信息表List1中，如果存在，则说明Vi重复注册，拒绝此次请求；如果不存在，说明Vi为新的合法投票者。

4）如果Vi是新的合法投票者，投票中心利用收到的公钥验证普通签名是否正确。若正确，投票中心为Vi随机选取一个具有唯一标识的身份序列码Ni，并对Ni签名；若不正确，投票中心要求投票者重新发送消息。最后，投票中心将身份序列码Ni和签名结果发送给投票人Vi，同时将相关信息保存到注册信息表List1中。

5）注册结束后网站公布List1,接受公开验证。在注册公示时间内，未能在注册期及时注册的投票者可以公开注册，公布自己的签名，要求投票中心为其发放身份序列码和对应签名，并将其信息添加到List1中，同时已注册者查询List1，如果查询不到自己的注册信息，投票者可以公示自己的签名，并公开验证消息m的关联环签名和普通签名的正确性，进而要求投票中心在所有人的监督下为其发放身份序列码和对应签名，并添加投票者的相应信息于List1上。上述方法能够有效防止注册中心的不诚实行为。此外，若公示时间结束后，投票者依然没有注册，则被视为自动弃权。

**投票阶段**

在规定的投票期限内，投票者按照如下协议进行投票（只有Ni存在于List1的投票者才有权参与如下投票过程）：

1）投票者Vi首先生成电子选票Ei，然后随机选择一盲化因子对选票进行盲化。

2）Vi对选票进行签名，然后将盲化选票、Ni、选票签名发给投票中心。

3）投票中心首先检查Ni是否已经存在于投票信息表List2中。如果已经存在，说明投票者Vi重复投票，拒绝此次请求；如果不存在，验证签名的正确性。

4）若签名正确，投票中心为Vi的盲化选票进行签名，并将盲化选票和签名发送给Vi，同时将相关信息保存到投票信息表List2中；若不正确，则要求投票者重新发送。

5）投票结束后公布List2，接受公开验证。在投票公示时间内，未能在投票期间及时投票的投票者可以公开投票，公布自己的盲化选票和签名，要求投票中心为其发送签名并将其投票结果公布于List2中；同时已投票者查询List2，如果查询不到自己的投票信息，投票者可以公布自己的盲化选票和签名，在所有人的监督下，要求投票中心为其签名并发送盲签名等信息给投票者，并将其结果公布到List2中。如果公示期结束，投票者依旧没有投票，则被视为中途放弃。

**计票阶段**

1）投票人Vi将消盲后的选票公布于计票信息表List3中，网站根据List3统计并公布选举的最终结果（只有Ni存在于List2中的投票者才有权参与如下计票过程）：

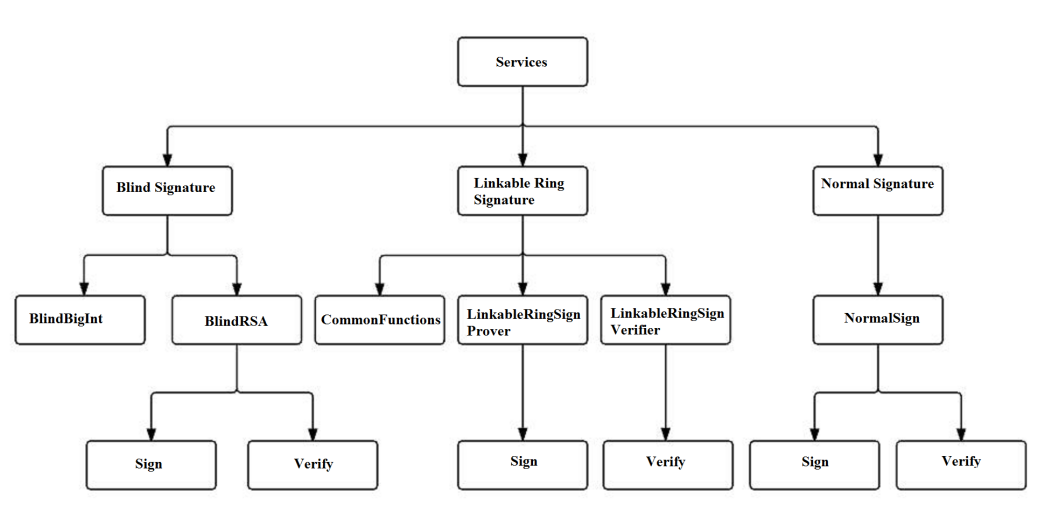
2）Vi对投票中心的盲签名做消盲，进而得到投票中心对选票的签名。

3）Vi对选票Ei签名，然后将相关信息公布于List3中。任何人均可验证最终计票信息的正确性。若投票中心和投票人对投票的签名都正确，说明选票为Vi的合法投票，并且选票没有被篡改。如果Ni在List3中重复出现，说明投票者Vi重复投票，仅对其选票统计一次。

4）超过机票期限，投票者依然未将选票公布于List3中，则被视为中途弃权。

5）网站根据List3统计并公布投票的最终结果。

**2.3 网站数据处理模块设计**

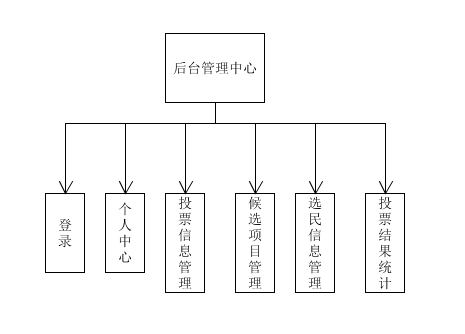
****服务器数据处理模块使用C++开发，用于处理投票用户和投票中心传来的数据。结构图如下：

**图7.3 数据处理模块结构图**

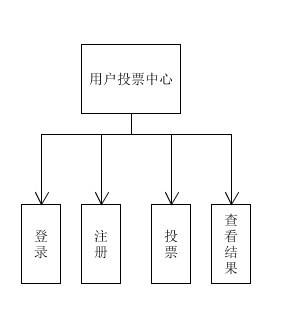
数据处理模块主要完成分配公私钥对、关联环签名的签名与验证、盲签名的签名与验证、普通RSA签名的签名与验证等功能。在整个系统中发挥了至关重要的作用。

**2.4 网站设计**

Web后端使用SSH框架开发，并采用Tomcat作为服务器；主要分为投票管理中心和用户投票中心两部分。系统采用MVC架构，使用Strus2作为控制层，负责数据的处理与转发；Hibernate框架负责对象的持久化工作；Spring负责事务的管理以及对象的创建工作。Web前端使用jsp实现，并使用jQuery、BootStrap作为UI支持，使用Ajax技术提高了用户交互。



**图7.4 后台管理中心架构图**



**图7.5 用户投票中心架构图**

# 附录9 组装测试计划

## 1引言

**1.1编写目的**

编写此说明书的目的是为了对系统提供得人功能或者系统的性能进行必要的测试。

预期读者：本文档的预期读者是开发工程师、测试工程师、RPC框架的使用者、RPC框架学习人员等。

**1.2背景**

开发软件名称：基于Netty的RPC框架的设计与实现

项目任务提出者：邹绿洲

项目开发者：邹绿洲

用户：需要进行RPC服务的开发人员、对RPC开发有兴趣的研究人员

实现软件单位：邹绿洲

项目与其他软件，系统的关系：

本项目使用Java语言进行编写，网络通讯框架使用的是Netty；消息序列化和反序列化使用的是Java提供的原生序列化、Kryo、Hessian等，对象池技术使用的是Apache commons-poo2包提供的开源框架；项目构建工具使用的是gradle。

**1.3定义**

无。

**1.4参考资料**

《软件文档写作教程》电子工业出版社 主编：马平、黄冬梅

需求说明书

概要设计说明书

详细设计说明书

## 2计划

**2.1系统说明**

**表9.1**

|  |  |  |  |
| --- | --- | --- | --- |
| **编号** | **所属模块** | **测试步骤** | **预期结果** |
| 1 | 服务启动模块 | 运行服务启动类；  加载配置文件；  启动RPC服务器 | 加载配置文件；  服务器启动，输出启动信息，等待客户端发起RPC请求 |
| 2 | RPC服务模块 | 编写客户端RPC请求代码；  客户端发起RPC请求 | 客户端成功发起RPC请求；输出请求信息 |
| 3 | RPC服务模块 | RPC服务器返回响应 | 根据客户端的请求返回对应的响应信息 |
| 4 | RPC服务模块 | 1000次RPC调用 | 在较短时间内完成RPC调用 |
| 5 |  |  |  |
| 6 |  |  |  |

**2.2 测试内容**

**2.2.1 RPC服务启动功能**

（1）测试目的

保证RPC服务器能够正常启动，服务启动需要加载配置文件，然后一直处于服务监听状态：

（2）测试过程及结果

**表9.2**

|  |  |
| --- | --- |
| 测试用例ID | LR-01-01 |
| 测试用例名称 | RPC服务启动功能 |
| 测试用例说明 | 运行服务启动类，加载服务配置文件 |
| 预置条件 | 编写服务配置启动文件 |
| 输入 | 服务启动代码，加载配置文件 |
| 预期结果 | 加载配置文件；  服务器启动，输出启动信息，等待客户端发起RPC请求 |
| 实际结果 | 如图所示，服务器正常启动，输出提示信息 |
| 结果分析 | 测试通过 ； 测试未通过 🞏 |

**测试描述**

通过启动服务器启动类，加载配置文件，服务器能够正常启动，并且输出IP地址个端口等提示信息，并且服务器将不会停止，一直处于服务监听状态。

**2.2.2 客户端发起RPC请求**

（1）测试目的

保证客户端能够发起RPC请求。

（2）测试用例及结果

**表9.3**

|  |  |
| --- | --- |
| 测试用例ID | LR-01-02 |
| 测试用例名称 | 客户端发起RPC请求 |
| 测试用例说明 | 编写客户端RPC请求代码，进行RPC请求 |
| 预置条件 | 需要编写RPC客户端请求类，请求接口方法，请求接口的实现类等 |
| 输入 | 测试请求代码： |
| 预期结果 | 成功发起RPC请求，输出请求消息。 |
| 实际结果 | 如图所示，客户端成功启动，输出IP和端口等信息。说明RPC请求成功发起。 |
| 结果分析 | 测试通过 ； 测试未通过 🞏 |

**测试描述**

通过编写特定的客户端请求代码，RPC客户端能够成功的发起请求，并且输出响应的提示信息，标志着客户端已经成功启动，，并且发起请求。

**2.2.3 服务器返回RPC响应**

（1）测试目的

保证RPC服务器在启动之后能够接收到RPC响应，对请求进行处理之后，将其返回，并且输出结果。

（2）测试用例及结果

**表9.4**

|  |  |
| --- | --- |
| 测试用例ID | LR-03-01 |
| 测试用例名称 | RPC服务器返回响应 |
| 测试用例说明 | RPC服务器接收请求，处理之后将处理结果返回 |
| 预置条件 | 无 |
| 输入 | RPC请求 |
| 预期结果 | 输出响应消息 |
| 实际结果 | 如图所示，响应消息输出。 |
| 结果分析 | 测试通过 ； 测试未通过 🞏 |

**2.2.4 RPC调用性能测试**

（1）测试目的

保证RPC调用性能在一定时间可完成，为性能优化提供数据支撑。

（2）测试用例及结果

**表9.4**

|  |  |
| --- | --- |
| 测试用例ID | LR-03-01 |
| 测试用例名称 | RPC性能测试 |
| 测试用例说明 | 进行1000次乘法RPC调用,看是否能在较短时间完成 |
| 预置条件 | 无 |
| 输入 | 1000次RPC调用 |
| 预期结果 | 输出响应消息 |
| 实际结果 | 如图所示，响应消息输出。 |
| 结果分析 | 测试通过 ； 测试未通过 🞏 |

**表9.4**

|  |  |
| --- | --- |
| 测试用例ID | LR-03-01 |
| 测试用例名称 | RPC性能测试 |
| 测试用例说明 | 进行1000次加法RPC调用,看是否能在较短时间完成 |
| 预置条件 | 无 |
| 输入 | 1000次RPC调用 |
| 预期结果 | 输出响应消息 |
| 实际结果 | 如图所示，响应消息输出。 |
| 结果分析 | 测试通过 ； 测试未通过 🞏 |

# 附录10 详细设计说明书

## 1引言

**1.1编写目的**

编写此说明书的目的是对项目的所有模块功能进行详细的介绍和说明。

预期读者：本文档的预期读者是开发工程师、测试工程师、RPC框架的使用者、RPC框架学习人员等。

**1.2背景**

开发软件名称：基于Netty的RPC框架的设计与实现

项目任务提出者：邹绿洲

项目开发者：邹绿洲

用户：需要进行RPC服务的开发人员、对RPC开发有兴趣的研究人员

实现软件单位：邹绿洲

项目与其他软件，系统的关系：

本项目使用Java语言进行编写，网络通讯框架使用的是Netty；消息序列化和反序列化使用的是Java提供的原生序列化、Kryo、Hessian等，对象池技术使用的是Apache commons-poo2包提供的开源框架；项目构建工具使用的是gradle。

。

**1.3定义**

无。

**1.4参考资料**

《软件文档写作教程》电子工业出版社 主编：马平、黄冬梅

需求说明书

概要设计说明书

## 2详细设计

**2.序列化**

**2.2.1 Protobuf实现序列化**

* **代码实现**

**（1）ProtostuffDecoder.java接口类**

**表10.1 类ProtostuffDecoder.java接口**

| **编号** | **函数名** | **说明** |
| --- | --- | --- |
| 1 | ProtostuffDecoder(MessageCodeHelper helper) | 构造函数 |

**（2）ProtostuffEncoder.java接口类**

**表10.2 ProtostuffEncoder.java接口**

| 编号 | 变量 | 说明 |
| --- | --- | --- |
| 1 | ProtostuffEncoder(MessageCodeHelper helper) | 构造函数 |

**（3）ProtostuffSerializable.java序列化类主要代码**

**代码清单10.1 ProtostuffSerializable.java**

|  |
| --- |
| *//序列化* @Override **public void** serialize(OutputStream outputStream, Object object) {   Class clazz = (Class) object.getClass();   LinkedBuffer buffer = LinkedBuffer.*allocate*(LinkedBuffer.***DEFAULT\_BUFFER\_SIZE***);   **try** {   Schema schema = *getSchema*(clazz);  ProtostuffIOUtil.*writeTo*(outputStream, object, schema, buffer);   } **catch** (Exception e) {  **throw new** IllegalStateException(e.getMessage(), e);   } **finally** {  buffer.clear();  }   }   *//反序列化* @Override **public** Object deserialize(InputStream inputStream) {  **try** {  Class clazz = isRpcDirect() ? MessageRequest.**class** : MessageResponse.**class**;  Object object = (Object) *objenesis*.newInstance(clazz);  Schema<Object> schema = *getSchema*(clazz);  ProtostuffIOUtil.*mergeFrom*(inputStream, object, schema);  **return** object;   } **catch** (Exception e) {  **throw new** IllegalStateException(e.getMessage(), e);  } } |

**2.2.2 Netty处理器实现网络传输**

**(1) 接收消息处理器ProtostuffReceiveHandler.java**

**代码清单10.2 ProtostuffReceiveHandler.java**

|  |
| --- |
| **public class** ProtostuffReceiveHandler **implements** NettyRpcReceiveHandler {   @Override  **public void** handle(Map<String, Object> map, ChannelPipeline pipeline) {   ProtostuffCodecHelper helper = **new** ProtostuffCodecHelper();  helper.setRpcDirect(**true**);  pipeline.addLast(**new** ProtostuffEncoder(helper));  pipeline.addLast(**new** ProtostuffDecoder(helper));  pipeline.addLast(**new** MessageReceiveHandler(map));    } } |

**(2) 发送消息处理器ProtostuffSendHandler.java**

**代码清单10.3 ProtostuffReceiveHandler.java**

|  |
| --- |
| **public class** ProtostuffSendHandler **implements** NettyRpcSendHandler {   @Override  **public void** handle(ChannelPipeline pipeline) {  ProtostuffCodecHelper helper = **new** ProtostuffCodecHelper();  helper.setRpcDirect(**false**);  pipeline.addLast(**new** ProtostuffEncoder(helper));  pipeline.addLast(**new** ProtostuffDecoder(helper));  pipeline.addLast(**new** MessageSendHandler());   } } |

**(3) 接收消息初始化器MessageReceiveChannelInitializer.java**

**代码清单10.4 MessageReceiveChannelInitializer.java**

|  |
| --- |
| **public class** MessageReceiveChannelInitializer **extends** ChannelInitializer<SocketChannel> {   **private** RpcSerializableProtocol **protocol**;   **private** RpcReceiveSerializableSelection **selection** = **null**;    **public** MessageReceiveChannelInitializer buildRpcSerializableProtocol(RpcSerializableProtocol protocol) {  **this**.**protocol** = protocol;  **return this**;  }    **public** MessageReceiveChannelInitializer(Map<String, Object> map) {  **selection** = **new** RpcReceiveSerializableSelection(map);  }    @Override  **protected void** initChannel(SocketChannel ch) **throws** Exception {   ChannelPipeline pipeline = ch.pipeline();  **selection**.select(**protocol**, pipeline);   } } |

**(3)发送消息初始化器MessageSendChannelInitializer.java**

**代码清单10.4 MessageSendChannelInitializer.java**

|  |
| --- |
| **public class** MessageSendChannelInitializer **extends** ChannelInitializer<SocketChannel> {   **private** RpcSerializableProtocol **protocol**;  **private** RpcSendSerializableSelection **frame** = **new** RpcSendSerializableSelection();   **public** MessageSendChannelInitializer buildRpcSerializableProtocol(RpcSerializableProtocol protocol) {  **this**.**protocol** = protocol;  **return this**;  }   @Override  **protected void** initChannel(SocketChannel ch) **throws** Exception {   ChannelPipeline pipeline = ch.pipeline();  **frame**.select(**protocol**, pipeline);   } } |

# 附录11 模块开发说明

## 1引言

**1.1编写目的**

编写此模块的目的是对系统的各个模块的实现进行说明。

预期读者：本文档的预期读者是开发工程师、测试工程师、RPC框架的使用者、RPC框架学习人员等。

**1.2背景**

开发软件名称：基于Netty的RPC框架的设计与实现

项目任务提出者：邹绿洲

项目开发者：邹绿洲

用户：需要进行RPC服务的开发人员、对RPC开发有兴趣的研究人员

实现软件单位：邹绿洲

项目与其他软件，系统的关系：

本项目使用Java语言进行编写，网络通讯框架使用的是Netty；消息序列化和反序列化使用的是Java提供的原生序列化、Kryo、Hessian等，对象池技术使用的是Apache commons-poo2包提供的开源框架；项目构建工具使用的是gradle。

**1.3定义**

无。

**1.4参考资料**

《软件文档写作教程》电子工业出版社 主编：马平、黄冬梅

《需求说明书》

《概要设计说明书》

《详细设计说明书》

## 2模块开发情况表

**2.1 Protobuf序列化和反序列化主要代码**

**2.1.1 ProtostuffCodeHelper.java**

**代码清单11.1 ProtostuffCodeHelper.java**

|  |
| --- |
| **public class** ProtostuffCodecHelper **implements** MessageCodeHelper {   **private static** Closer *closer* = Closer.*create*();   **private** ProtostuffSerializablePool **pool** = ProtostuffSerializablePool.*getProtostuffInstance*();   **private boolean rpcDirect** = **false**;   **public boolean** isRpcDirect() {  **return rpcDirect**;  }   **public void** setRpcDirect(**boolean** rpcDirect) {  **this**.**rpcDirect** = rpcDirect;  }   *//编码* @Override  **public void** encode(ByteBuf out, Object message) **throws** IOException {   **try** {   ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();   *//注册好需要关闭的资源  closer*.register(byteArrayOutputStream);   ProtostuffSerializable protostuffSerializable = **pool**.borrow();  protostuffSerializable.serialize(byteArrayOutputStream, message);   **byte**[] bytes = byteArrayOutputStream.toByteArray();  **int** length = bytes.**length**;  out.writeInt(length);  out.writeBytes(bytes);   **pool**.restore(protostuffSerializable);   } **finally** {  *closer*.close();  }  }   *//解码，返回一个对象* @Override  **public** Object decode(**byte**[] bytes) **throws** Exception {   **try** {   ByteArrayInputStream byteArrayInputStream = **new** ByteArrayInputStream(bytes);   *closer*.register(byteArrayInputStream);   ProtostuffSerializable protostuffSerializable = **pool**.borrow();  protostuffSerializable.setRpcDirect(**rpcDirect**);   Object object = protostuffSerializable.deserialize(byteArrayInputStream);  **pool**.restore(protostuffSerializable);  **return** object;   } **finally** {   *closer*.close();   }   } } |

**2.2 Netty网络通讯模块主要代码**

**2.2.1消息接收模块**

* MessageReceiveHandler**.java**

**代码清单11.4 register**

|  |
| --- |
| **public class** MessageReceiveHandler **extends** ChannelInboundHandlerAdapter {  **private final** Map<String, Object> **map**;   **public** MessageReceiveHandler(Map<String, Object> map) {  **this**.**map** = map;  }    @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {   MessageRequest request = (MessageRequest) msg;  MessageResponse response = **new** MessageResponse();   ReceiveInitializeComponent facade = **new** ReceiveInitializeComponent(request, response, **map**);   Callable<Boolean> receveiveTask = facade.getTask();   MessageReceiveExecutor.*submit*(receveiveTask, ctx, request, response);    }   @Override  **public void** exceptionCaught(ChannelHandlerContext ctx, Throwable cause) **throws** Exception {   cause.printStackTrace();  ctx.close();   } } |

**(2) 消息发送模块**

* MessageSendHandler**.java**

**代码清单11.5** MessageSendHandler.java

|  |
| --- |
| **public class** MessageSendHandler **extends** ChannelInboundHandlerAdapter {    **private** ConcurrentHashMap<String, MessageCallBack> **map** = **new** ConcurrentHashMap<>();  **private volatile** Channel **channel**;  **private** SocketAddress **address**;    **public** Channel getChannel() {  **return channel**;  }   **public** SocketAddress getAddress() {  **return address**;  }    */\*\*  \* pipline激活后的方法  \*  \** ***@param ctx*** *\** ***@throws*** *Exception  \*/* @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  **super**.channelActive(ctx);  **this**.**address** = **this**.**channel**.remoteAddress();  }   */\*\*  \** ***@param ctx*** *\** ***@throws*** *Exception  \*/* @Override  **public void** channelRegistered(ChannelHandlerContext ctx) **throws** Exception {  **super**.channelRegistered(ctx);  **this**.**channel** = ctx.channel();  }    */\*\*  \** ***@param ctx*** *\** ***@param msg*** *\** ***@throws*** *Exception  \*/* @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {  MessageResponse response = (MessageResponse) msg;  String messageId = response.getMessageId();   MessageCallBack messageCallBack = **map**.get(messageId);   **if** (messageCallBack != **null**) {  **map**.remove(messageId);  messageCallBack.over(response);  }   }   */\*\*  \* 发生异常后进行的处理  \*  \** ***@param ctx*** *\** ***@param cause*** *\** ***@throws*** *Exception  \*/* @Override  **public void** exceptionCaught(ChannelHandlerContext ctx, Throwable cause) **throws** Exception {  cause.printStackTrace();  ctx.close();  }   */\*\*  \* 关闭时进行的处理  \*/* **public void** close() {   **channel**.writeAndFlush(Unpooled.***EMPTY\_BUFFER***).addListener(ChannelFutureListener.***CLOSE***);  }    */\*\*  \* 发送消息请求  \*  \** ***@param request*** *\** ***@return*** *\*/* **public** MessageCallBack sendRequest(MessageRequest request) {   MessageCallBack messageCallBack = **new** MessageCallBack(request);  **map**.put(request.getMessageId(), messageCallBack);  **channel**.writeAndFlush(request);  **return** messageCallBack;  }  } |

**2.2.2 对象池池化技术模块举例**

* HessianSerializablePool.java

**代码清单11.15 HessianSerianSerilizablePool.java**

|  |
| --- |
| **public class** HessianSerializablePool {   *//通用对象池* **private** GenericObjectPool<HessianSerializable> **hessianSerializableGenericObjectPool**;   **private static volatile** HessianSerializablePool *poolFactory* = **null**;   **private** HessianSerializablePool() {   **hessianSerializableGenericObjectPool** = **new** GenericObjectPool<HessianSerializable>(**new** HessianSerializableFactory());  }   *//构造方法* **public** HessianSerializablePool(**final int** serializablePoolMaxTotal, **final int** serializablePoolMinIdle, **final long** serializablePoolMaxWaitMillis, **final long** minEvictableIdleTimeMillis) {   **hessianSerializableGenericObjectPool** = **new** GenericObjectPool<HessianSerializable>(**new** HessianSerializableFactory());   *//对象池的配置信息* GenericObjectPoolConfig config = **new** GenericObjectPoolConfig();   *//连接池中的最大连接数* config.setMaxTotal(serializablePoolMaxTotal);  *//连接池中的最大空闲连接数* config.setMinIdle(serializablePoolMinIdle);  *//等待时间，超过这个时间就会抛异常* config.setMaxWaitMillis(serializablePoolMaxWaitMillis);  *//空闲连接的最小时间，超出这个时间连接将会移除* config.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);   **hessianSerializableGenericObjectPool**.setConfig(config);   }   *//线程安全的单例模式，双重锁检查机制* **public static** HessianSerializablePool getHessianPoolInstance() {   **if** (*poolFactory* == **null**) {   **synchronized** (HessianSerializablePool.**class**) {   **if** (*poolFactory* == **null**) {  *poolFactory* = **new** HessianSerializablePool(SystemConfig.***SERIALIZABLE\_POOL\_MAX\_TOTAL***, SystemConfig.***SERIALIZABLE\_POOL\_MIN\_IDLE***,  SystemConfig.***SERIALIZABLE\_POOL\_MAX\_WAIT\_MILLIS***, SystemConfig.***SERIALIZABLE\_POOL\_MIN\_EVICTABLE\_TIME\_MILLIS***);  }  }  }   **return** *poolFactory*;  }   */\*\*  \* 获取池中的对象的方法  \*  \** ***@return*** *\*/* **public** HessianSerializable borrow() {   **try** {   **return** getHessianSerializableGenericObjectPool().borrowObject();   } **catch** (Exception e) {   e.printStackTrace();  **return null**;   }  }   */\*\*  \*将对象还回池中  \** ***@param hessianSerializable*** *\*/* **public void** restore(**final** HessianSerializable hessianSerializable) {  getHessianSerializableGenericObjectPool().returnObject(hessianSerializable);  }  */\*\*  \* 获取对象池的方法  \** ***@return*** *\*/* **public** GenericObjectPool<HessianSerializable> getHessianSerializableGenericObjectPool() {  **return hessianSerializableGenericObjectPool**;  } } |

# 附录12 单元测试

**表12.1 单元测试表**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **所属模块** | **测试步骤** | **预期结果** | **是否通过测试** | **测试人员** |
| 1 | 序列化模块 | JDK序列化 | 将消息序列化字节流 | 是 | 邹绿洲 |
| JDK反序列化 | 将字节流反序列为消息对象 | 是 | 邹绿洲 |
| 2 | 序列化模块 | Hessian进行序列化和反序列化 | 使用Hessian第三方序列化框架帮助进行序列化和反序列化 | 是 | 邹绿洲 |
| 3 | 序列化模块 | Kryo进行序列化和反序列化 | 使用Kryo第三方序列化框架帮助进行序列化和反序列化 | 是 | 邹绿洲 |
| 4 | 序列化模块 | Protobuf进行序列化和反序列化 | 使用Protobuf第三方序列化框架帮助进行序列化和反序列化 | 是 | 邹绿洲 |
| 5 | Netty模块 | Netty进行通讯 | 编写Netty处理器，进行网络通讯 | 是 | 邹绿洲 |
| 6 | 对象池化 | 使用池化框架进行池化 | 使用Apache commons-poo2包提供的对象缓存池技术，帮助提升性能。 | 是 | 邹绿洲 |
| 7 | Spring管理模块 | 使用Spring进行服务注册、管理 | 编写Spring配置文件，对服务进行加载、配置 | 是 | 邹绿洲 |