分类号_____ 学号 M201676137

学校代码 10487 密级_____

华中科技大学

硕士学位论文

分布式 RPC 框架的设计与实现

学位申请人：孔海峰

学 科 专 业：软件工程

指 导 教 师：陈长清 副教授

答 辩 日 期：2018.12.17

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering

Design and Implementation of Distributed RPC Framework

Candidate : Kong Haifeng

Major : Software engineering

Supervisor : Assoc.prof. Chen Changqing

Huazhong University of Science & Technology

Wuhan 430074, P.R.China

December, 2018

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：孔海峰

日期：2018年12月17日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于

保　密□，在_____年解密后适用本授权书。

不保密☑。

（请在以上方框内打“√”）

学位论文作者签名：孔海峰　　指导教师签名：陈长清

日期：2018年12月17日　　日期：2018年12月17日

摘 要

随着互联网技术的不断突破，互联网应用所具备的功能也日益增多。互联网企业为了能够吸引更多的用户群体，不断的完善并扩展自己的系统，给用户提供更多元的服务。伴随着系统的不断完善，业务规模不断壮大，系统也将变得庞大且复杂。这将大大的提高了系统的交互、版本迭代以及后期维护提的成本。而将系统本身按业务功能进行拆分，拆分后的各个模块通过服务发布和服务调用的形式协同工作，则可以在保证系统的性能前提下，大幅度降低上述所需要的成本。分布式 RPC(Remote Procedure Call)框架则是工作在系统拆分后的各个模块之间的中间件，协同各个模块之间的通信，保证系统的正常运行。

传统的远程过程调用方式通常采用 HTTP 协议进行通信，需为每一个调用建立一个 TCP 连接，且每次请求和响应都会携带部分无效字段，影响服务调用效率。本文所设计的 RPC 框架通过自定义通信数据格式来减少无效字段的传输，同时本框架底层网络通信采用了异步通信框架。本框架按功能可分为数据处理模块、服务发布模块、服务调用模块。数据处理模块用于请求数据和响应数据的收发以及它们的编码和解码。服务发布模块通过服务注册、服务发布实现了该框架高性能和高可用行性，同时还支持服务水平扩张。服务调用模块是将经过数据处理模块后的数据作为服务调用的参数，对相应服务进行调用。其中服务调用模块为 RPC 框架的核心功能模块，服务的调用采用了负载均衡策略，有效的提高硬件资源的利用率。

本框架与传统远程调用过程方式相比，在数据的传输格式上进行了自定义，提高了系统有效吞吐量。在信息交互层面采用了异步通信框架 Netty，让服务调用方在发起请求后能够较快的得到响应。服务的发布采用了注册+发布的形式，提高了框架的可用性和扩展性，能够有效的预防单点故障，且很好的支持服务的水平扩展。本框架工作在业务复杂的系统中时，在保证服务请求响应速度的前提下，对系统的业务模块起到了很好的解耦作用。

关键词：远程过程调用，服务调用，异步通信，分布式

Abstract

With the continuous breakthrough of Internet technology, the functions of Internet applications are also increasing. In order to attract more user groups, Internet companies continue to improve and expand their systems to provide users with more services. With the continuous improvement of the system, the scale of the business continues to grow, and the system will become huge and complex. This will greatly increase the cost of system interaction, version iteration, and post-maintenance. The system itself is split according to the business function, and the split modules work together in the form of service release and service call, which can greatly reduce the cost required above under the premise of ensuring the performance of the system. The distributed RPC (Remote Procedure Call) framework is a middleware that works between modules to coordinate the work between modules to ensure the normal operation of the system.

The traditional remote procedure call method usually uses the HTTP protocol to communicate. A TCP connection needs to be established for each call, and each request and response carries a partial invalid field, which affects the service call efficiency. The RPC framework designed in this paper reduces the transmission of invalid fields by customizing the communication data format. At the same time, the underlying information of the framework is used interchangeably asynchronous communication framework. The framework can be divided into data processing module, service publishing module and service calling module according to functions. The data processing module is used to request decoding of data and encoding of response data. The service release module implements the framework's high performance and high availability lines through service registration and service release, while also supporting service level expansion. The service invocation module is a parameter that calls the data after the data processing module as a service, and calls the corresponding service. The service invocation module is the core function module of the RPC framework, and the service call adopts a load balancing strategy to improve the utilization of hardware resources. Compared with the traditional

remote process mode, this framework customizes the data transmission format and improves the effective throughput of the system. The asynchronous communication framework Netty is adopted in the information interaction mechanism, and the service caller can get a response quickly after initiating the request. The release of the service adopts the form of registration + release, which improves the availability and expansion of the framework, effectively prevents single points of failure, and supports the horizontal expansion of services. When the framework works in a complex system of business, under the premise of guaranteeing the response speed of the service request, it plays a good decoupling effect on the business module of the system.

Keywords: Remote Procedure Call, Service call, Asynchronous communication, Distributed

目 录

1 绪论

本章将以互联网应用在发展过程中架构的变化作为本课题的研究的背景，阐述传统应用架构在某些应用场景下的不足。并在此基础上，给出本文所研究内容的意义，以及本文的主要内容。

1.1 研究背景及意义

互联网刚刚兴起之时，MVC(Model View Controller) 架构是主流应用架构[1]。当时系统业务规模还比较小，所有功能都部署在同一个进程中，再通过双机或者前置负载均衡器实现负载分流[2]。随着系统的规模不断增大，同一个系统中的垂直应用越来越多，应用之间交互不可避免，开发人员通常将核心和公共业务抽取出来，作为独立的服务，实现前后台逻辑分离[3]。这种以提高业务复用及拆分为目标的架构便是当今主流的应用架构 RPC 架构。

互联网企业在初创时期，其互联网应用的业务往往是较为单一的，系统规模也比较小，开发人员一般会采用 MVC 架构作为系统的应用架构。由于系统的规模较小，且业务单一，此时使用这种架构，能够在系统规模可控的前提下，对系统进行快速迭代，并且后期系统维护的成本也是可控的。随着企业的不断发展，用户量的不断扩大，伴随而来的是业务的不断扩大，系统规模的不断增大。如果此时还采用 MVC 架构作为系统的应用架构，将所有功能都部署在一个进程中，那么无论是从维护的成本来看，还是从系统的版本迭代来看，其所需要花费代价是巨大的。而 RPC 架构的出现则能有效的解决上述问题。RPC 架构的核心思想则是把复杂庞大的业务进行拆分，将业务与业务之间进行解耦[4]。不同的业务模块之间，以服务的发布、调用的方式来进行数据的交换，协同工作。此时系统的版本迭代，对于程序开发人员来说，已经不是整个庞大而复杂的系统了，而是系统拆分后的具有单一功能的各个模块。这极大的提高系统项目交付、版本迭代的效率，同时也大大较少维护的成本。将不同的业务部署在不同的主机上，通过某种均衡负载策略进行主机的选择，也能有效提高硬件资源的利用率，提高系统的吞吐量。

1.2 国内外研究现状

RPC 概念的提出可以追溯到上世纪 80 年代，由 Birrell 和 Nelson 在他们发表的一篇名为《Implementing Remote Procedure Calls》的论文中提出来的[5]。RPC 是一种通讯标准，而不是一种特定的产品。根据此种标准制定出来的 RPC 框架，可以使得服务调用者可以像调用本地方法一般调用远程服务，且调用过程对调用者来说完全透明。

1.2.1 国外研究现状

根据 RPC 标准所制定出来的优秀的 RPC 框架并不少见，其发展过程主要从早期的基于文本的 RPC 框架 Web Service，到后来的基于二进制的 RPC 框架 Hession、Thrift 等框架。这些 RPC 框架都是国外最为常见的 RPC 框架，且各自都有其自身的优点及适用场景。

Web Service 采用 HTTP 协议作为客户端和服务端的通信协议。数据的封装格式通常采用 XML，XML 主要的优点在于它是跨平台的[6]。客户端和服务端在使用 Web Service 进行通信时，无论是请求数据还是响应数据，都将组织成 XML 格式的数据。由于双方是基于 HTTP 协议进行通信的，所以这些数据还将额外携带部分 HTTP 协议自带的数据，用来描述通信数据的内容格式。这些额外携带的 HTTP 消息头和 XML 内容格式则是 SOAP 协议规定的[7]。

Hessian 是一个轻量级的采用 HTTP 协议作为服务端和客户端的通信协议的 RPC 框架[8]，它采用了较为简单的方法提供了远程方法调用的功能。与 Web Service 相比，Hessian 的使用较为简便，使用起来也很便捷。由于该框架要求数据的传输形式是二进制的，因此对于二进制形式的数据收发尤为友好。

Thrift 作为最常见的 RPC 框架，最大的特点就是它的跨平台性。该框架是由著名的社交公司 Facebook 在 2007 年进行开发的，并在 2008 年贡献给了最大的开源组织 Apache，成为开源项目[9]。Thrift 跨平台技术的实现是通过用一种中间语言来定义 RPC 的接口和数据类型（这种中间语言不依赖与任何一门特定的语言），对定义好的中间语言，采用特定的编译器，根据需求将其编译成不同的语言的中间代码。这部分编译产生的中间代码负责 RPC 协议层以及传输层的实现[10]。由于 Thrift 可以跨语言使用，

故该框架特别适用于服务端与客户端采用不同语言编写的情况下，进行服务之间调用的场景。

1.2.2 国内研究现状

近些年来，国内的互联网发展势头非常迅猛。这也造就了一些巨头互联网公司。这些互联网公司投入了大量的资源用以技术层面研发。在这种大力度的投入下，许多优秀的开源产品得以发布。这其中就包含了许多 RPC 框架。与上述基于文本、二进制的 RPC 框架不同的是，这些框架是通用型 RPC 框架，能够兼容各种通信数据。如阿里巴巴旗下的开源 RPC 框架：Dubbo，58 集团旗下开源的 RPC 框架：Gaea，百度旗下的开源 RPC：brpc。这些框架最大的特点就是系统解耦后，完成各业务模块之间的通信。

Dubbo 是阿里巴巴公司开源的一个优秀的高性能的服务框架，该架构可以使应用双方通过网络完成高性能的服务请求和响应的功能[11]。Dubbo 的主要核心部件有 Remoting、RPC、Registry[12]。其中 Remoting 部件在 Dubbo 中负责网络通信，实现了 sync-over-async 通信方式，且实现了 request-response 消息机制。RPC 部件用于支持 Dubbo 框架中的负载均衡、容灾、集群等功能[13]。Registry 部件则是用户服务的注册和服务时间的发布与订阅。

Gaea 是 58 集团开源的一个优秀的高性能的服务通讯框架 [14]。Gaea 主要由三个组件构成：Gaea Client、Gaea Serializer、Gaea Protocol。Gaea Client 组件主要负责框架的负载均衡，网络通讯等功能。同时在网络通讯中采用了滑动窗口机制用于拥塞控制。Gaea Serializer 组件用于数据包的序列化和反序列化，且序列化后的数据为无元数据信息，故数据量有所减少从而提高通信效率。Gaea Protocol 组件则是该框架自定义一组数据协议。其中包含了请求协议、响应协议、异常协议。

brpc 又称为 baidu-rpc，是百度开发一款 RPC 框架。百度从 2010 开始便开发了若干 RPC 框架：ubrpc、nova_pbrpc、public_pbrpc 以及 brpc。brpc 目前被应用于百度公司内部各种核心业务上。百度在 brpc 开源的文档中提到 brpc 提供了高性能计算和模型训练和各种索引和排序服务，且有超过 100 万以上个实例是基于 brpc 工作的[15]。可以看出 RPC 的使用已是无处不在。

1.3 本文研究的主要内容

本文旨在介绍以服务发布和订阅为通信模式的分布式 RPC 框架的设计与实现。与生产环境中常见的 RPC 框架不同的是，本文将设计并实现一种高性能，高可用的功能单一的分布式 RPC 框架。本框架按功能可分为数据处理模块、服务发布模块、服务调用模块，根据模块的不同，研究的内容主要如下：

1）设计并实现服务端与客户端之间的数据传输格式。在保证数据完整性前提下，尽可能压缩数据大小，减少带宽的占用。在数据的收发层面上，数据的发送方对待发送数据进行编码，便于数据的传输，数据的接收方对接收到的数据解码，获取原数据。

2）设计并实现服务的注册中心。该注册中心用于服务端的注册，以及客户端的服务调用地址查询。当服务端中出现部分主机宕机的情况时，客户端能正常的进行服务调用。当服务端需要水平扩展时，也可以随时在注册中心进行注册，且在注册完成后，实时对外提供服务。

3）设计并实现服务的正确调用。服务端正确接收服务调用请求前提下，为了提高本框架的交互性能，需采用异步通信机制作为请求接收、响应获取机制。同时本框架需要根据服务端的负载情况动态的选择负载较小的服务端发送服务调用请求。

2 相关技术介绍

为了提高本文所设计的 RPC 框架的性能，在设计过程中，各个模块中都采用优秀的第三方开源工具。在数据处理模块中，为了加速服务端和客户端通信速率，在该模块底层信息交互层面采用了目前较为流行的异步通信框架 Netty。在数据对象发送之前，需将数据进行序列化转换成二进制流用以传输，而在数据被接受之后则需对二进制流进行反序列化用于还原数据。本框架对该过程的处理，采用了 Google 公司开源序列化框架 Protostuff。在服务发布模块中，注册中心是基于一致性服务软件 ZooKeeper 进行设计开发的。为了提高本框架的易用性，本框架采用注解+Spring 的方式获取服务对象，并通过反射机制完成对服务对象的调用。

2.1 Spring 框架

Spring 是一个 2003 年兴起的设计层面的开源框架。该框架致力于系统在逻辑结构上的分层，它的出现很好的解决了系统中不同逻辑层之间高度耦合的问题，因此在使用 Spring 框架进行系统开发时，面向接口的编程思想往往会贯穿始终 [16]。Spring 的本着兼容并包的设计理念，能与绝大优秀的开源组件进行集成。Spring 框架所具备的特性主要有两大核心功能作为支撑，它们分别是控制反转(Inverse of control, IoC)和面向切面编程(Aspect-Oriented Programming, AOP) [17]，下面将从这两方面对 Spring 框架进行介绍。

2.1.1 控制反转(IoC)

控制反转不是具体的一向技术，而是一种设计思想[18]。在传统的 Java 开发模式中，对象的内部依赖的获取通常是由开发人员主动进行完成，该过程可以直观的理解为程序员通过 new 关键字创建被依赖对象。而 IoC 的出现则意味着对象的创建及依赖的注入交给 IoC 容器进行完成。对于 Spring 容器来说，IoC 机制使得它能够控制并管理对象的生命周期和对象间的依赖关系。

IoC 的实现主要是通过 XML 格式的配置文件（或注解）和 Java 反射机制共同完成的。配置文件主要是提供后期依赖注入时所需要的数据，Java 反射机制则是用于完

成依赖创建的操作，可以说 IoC 的实现核心便是 Java 反射机制。配置文件中最常见的标签“<bean>”标签中有两个重要的属性，分别是“id”和“class”属性。Spring 在启动时会扫描对应的配置文件。并将配置文件中的所有信息并封装成 BeanDefinition 对象，之后将配置文件中的 id 属性作为 key，BeanDefinition 作为 value 存放在 Spring 框架中的 HashMap 中。

依赖注入的操作主要是利用了 Java 反射机制[19]来实现的。当 Spring 框架发现对象中的属性需要被注入时，首先从上述提到的 HashMap 中找到对应的类信息，并通过反射机制提供的 Class.forName()方法获取属性对象的字节码，在利用字节码的 newInstance()方法创建该属性对象。获取到依赖后再利用反射机制提供的 setFieldValue()调用该属性的 set 方法完成属性对象的注入。

2.1.2 IoC 实现策略

IoC 的又称为依赖注入(Dependency Injection, DI)。Spring IoC 实现策略即依赖注入的方式主要有三种，分别是：接口注入、setter 方法注入以及构造器注入[20]。三个注入的方式介绍如下：

1）接口注入：接口注入的核心思想是将调用者与实现着相分离。将具体的实现封装成一个接口，调用者通过注入该接口后获取实例对象，完成调用。

2）setter 方法注入：此种注入方法主要是将待注入的的属性值作为方法参数，通过调用属性的 set 方法，完成属性对象的依赖注入，是最常使用的注入方式之一。

3）构造器注入：将所需注入的对象作为构造函数参数，当 IoC 容器初始化对象时，对待注入的对象进行设值，完成依赖的注入。

上述三种注入方式都有其特定的使用场景，由于接口注入方式需实现特定的接口，导致程序的开发效率有所下降，因而目前生产环境中使用的较少。构造器注入是被推荐的使用方式，因为此方式能够保证注入的组件不可变，并且确保需要的依赖不为空。此外，构造器注入的依赖总是能够在返回客户端（组件）代码的时候保证完全初始化的状态。如果依赖关系较为复杂，则不推荐使用构造器注入模式，因为这会使得构造函数变得复杂。当一个类需要配备一个默认的构造函数时，构造器注入模式也不能用于这种情景。

2.1.3 面向切面编程(AOP)

AOP 是 Spring 的另一个核心机制，可以说是 OOP(Object-Oriented Programming)的补充和完善[21]。AOP 的引入可以让开发人员很好的对程序中的纵向关系进行定义，例如程序中的日志功能。实现日志功能的程序代码，往往与核心的业务代码是没有关系，类似的还有异常处理功能的程序代码，这些代码在采用 OOP 的思想进行程序编写时，会导致大量重复代码的产生，而 AOP 的引入，则能很好的解决这个问题。采用 AOP 进行程序的开发可以将非业务代码进行抽取，作为公共执行代码块，织入在核心业务代码执行的前后，完成非业务功能代码的执行。

Spring 中 AOP 的底层实现可以使用基于接口实现的 JDK 动态代理和基于继承实现的 CGlib 两种方式。用户通过配置切入点（pointcut）以及通知（advice），来完成被代理方法的增强。具体过程如下：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[22]。

在 Java 中实现 AOP 框架中有许多织入方式，这些织入方式大致可以分为三类：

1）静态织入（编译器注入）：在预编译阶段对源代码进行修改，编译后的字节码为功能加强后的字节码，AspectJ 采用的正是这种方式。

2）转载期织入：框架通过修改 ClassLoader，在类加载的过程中对类的二进制字节码进行修改[23]，完成功能加强。

3）运行时织入：利用 JDK 动态代理产生的代理对象在对其进行方法调用时进行拦截，并织入增强代码。这也是 Spring 中 AOP 默认所采用的织入方式。

2.2 网络 I/O 技术

网络 I/O 的类型按照应用与内核的交互与方式可分为同步 IO 和异步 IO，根据应用程序指令在 IO 期间是否可以继续执行可分为阻塞 IO 和非阻塞 IO[24]。同步 IO 指的是程序进程发出调用 IO 操作请求发出后，并等待 IO 操作结果返回。而异步操作是指程序进程发出调用 IO 操作后，这个调用直接返回没有结果。而在调用者发出调用之

后，被调用方通过状态等方式通知调用者。阻塞 IO 指的是当用户发起 IO 请求后，需等待 IO 操作完成后再执行后续指令，而非阻塞 IO 指的是当用户发起 IO 请求后，后续继续执行而不必等待 IO 操作完成。本框架中为了提高服务端响应速度采用了同步非阻塞通信框架 Netty。

2.2.1 Java BIO

Java BIO 是在 JDK 1.0 中发布的，BIO 编程是同步阻塞开发模型，其模型结构图 2-1 所示：

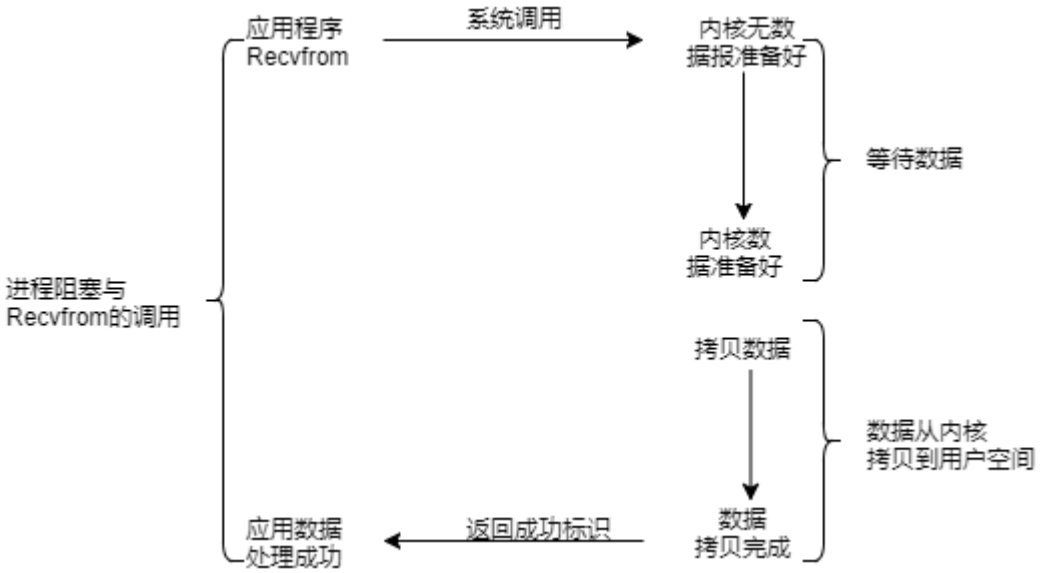

图 2-1 同步阻塞 IO 模型

Java BIO 通信模型是最为经典的客户端/服务器通信模型，其中服务端的运行依赖于主机的 IP 地址和端口号，服务端通过对它们进行绑定，提供唯一的地址信息，开放访问。客户端则通过向与服务器所对应的地址信息发起连接请求，连接建立完毕后，双方便可通过套接字（Socket）进行网络通信。

Java BIO 通信过程如下：首先服务端开启一个单独的线程 Acceptor，用于监听与服务器绑定的端口是否有来自客户端的连接建立的请求，若有请求，则直接对该请求进行接收。Acceptor 每次接收到来自客户端的请求后，都会为此次连接开启一个新的线程，用于处理客户端的请求[25]。随着请求的处理完成，伴随此次连接建立的线程也随之被销毁。Java BIO 的服务端通信模型为一客户端一线程，模型如图 2-2 所示：

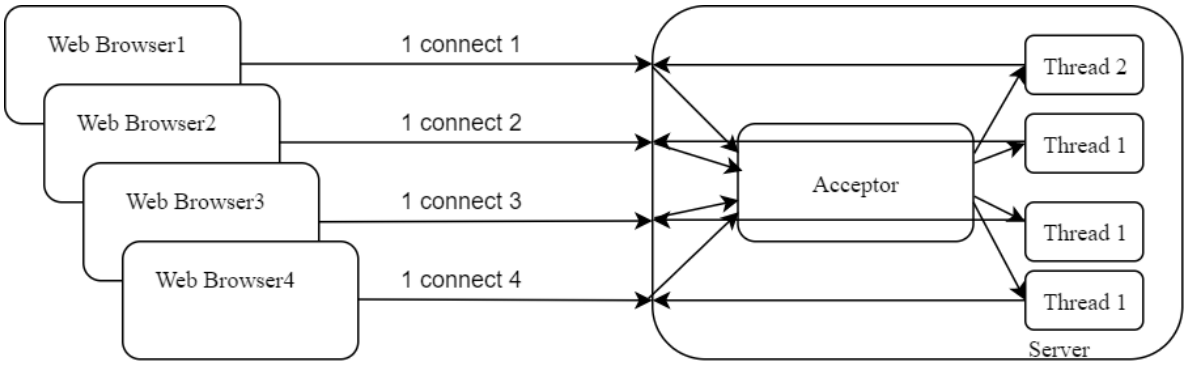

图 2-2 Java BIO 服务端通信模型

该模型缺乏弹性伸缩能力，服务端线程个数与客户端并发访问数呈 1：1 的正比关系，随着用户访问不断增加，线程数会不断膨胀，线程会因此而被不断的创建，服务器会对运行中的线程上下文进行频繁的切换，消耗大量资源的同时使得系统性能急剧下降，最终会出现线程堆栈溢出、创建新线程失败等异常，从而导致服务端宕机。

上述模型面对大量用户并发访问时，会产生大量线程，严重影响系统的性能。针对这种不足可以采用线程池对上述模型加以改善。线程池的引入可以避免线程无休止的创建，用户可以根据主机性能对线程池中的线程数进行配置。当用户并发量过大，系统无法正常处理时，也可以通过配置线程池的拒绝策略[26]，对不能处理的请求进行相应处理。带线程池的 Java BIO 服务端通信模型如图 2-3 所示：

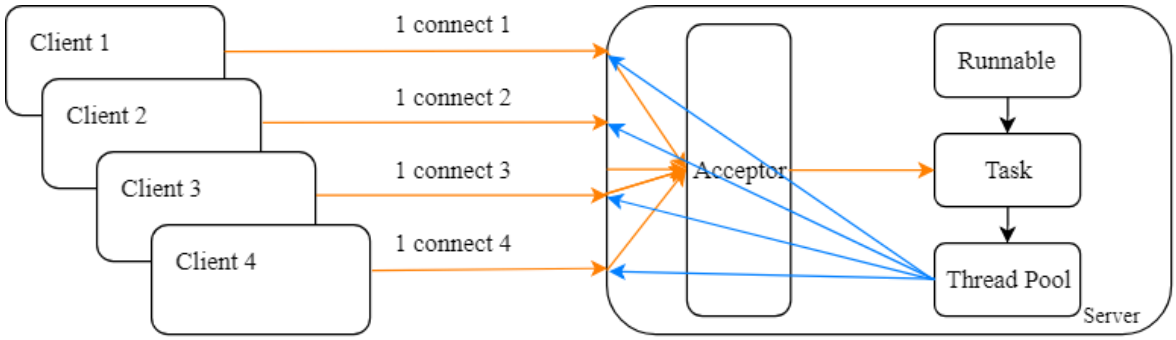

图 2-3 Java BIO 带线程池的服务端通信模型

将线程的创建委托给线程池一定程度上解决了线程数量过多的问题，但面对高并发的场景，仍然有许多的请求会被丢弃，因此不能很好的处理高并发的场景。

2.2.2 Java NIO

JDK 1.4 之后 SUN 公司为 Java 提供非阻塞式 I/O，即 Java NIO(New IO)。NIO 较传统的 IO 而言，有着更高的效率。但是两者底层实现的原理是完全不同的。由于 NIO 的数据传输是直接面向缓冲区的，从而减少数据拷贝的部分操作，因此 NIO 能够以更快的速度完成对数据进行相关操作。Java NIO 是一种同步非阻塞式 IO，其模型图如图 2-4 所示：

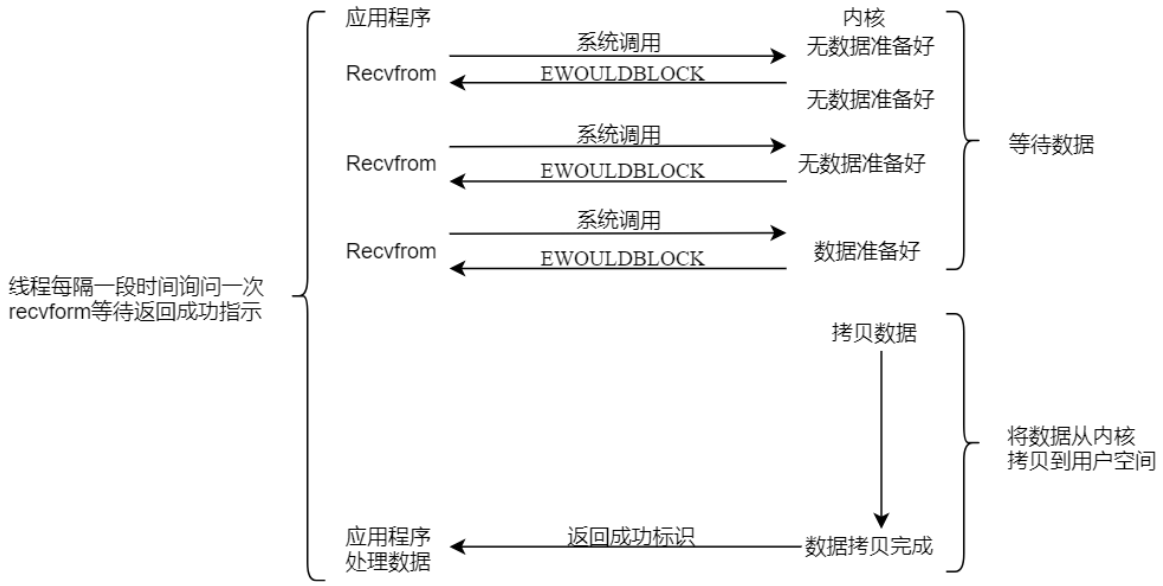

图 2-4 同步非阻塞 IO 模型

Java NIO 的设计模型采用的是典型事件驱动模型，消息接收模式使用的是 Reactor 模式。Java NIO 中客户端发起的连接时首先需要向 Reactor 进行注册，该过程由 Acceptor 完成，之后 Reactor 线程负责管理这些注册后的 Socket 通道。Reactor 会不断轮询查询各种 I/O 事件，若有，则将事件处理传递给相应的 Handler，由该 Handler 进行业务的具体执行[27]，该过程如图 2-5 所示。除此之外，Reactor 还会不断轮询查看是否存在未处理完的数据，故 Java NIO 更适合用于连接数量多，但连接时间短端的请求。

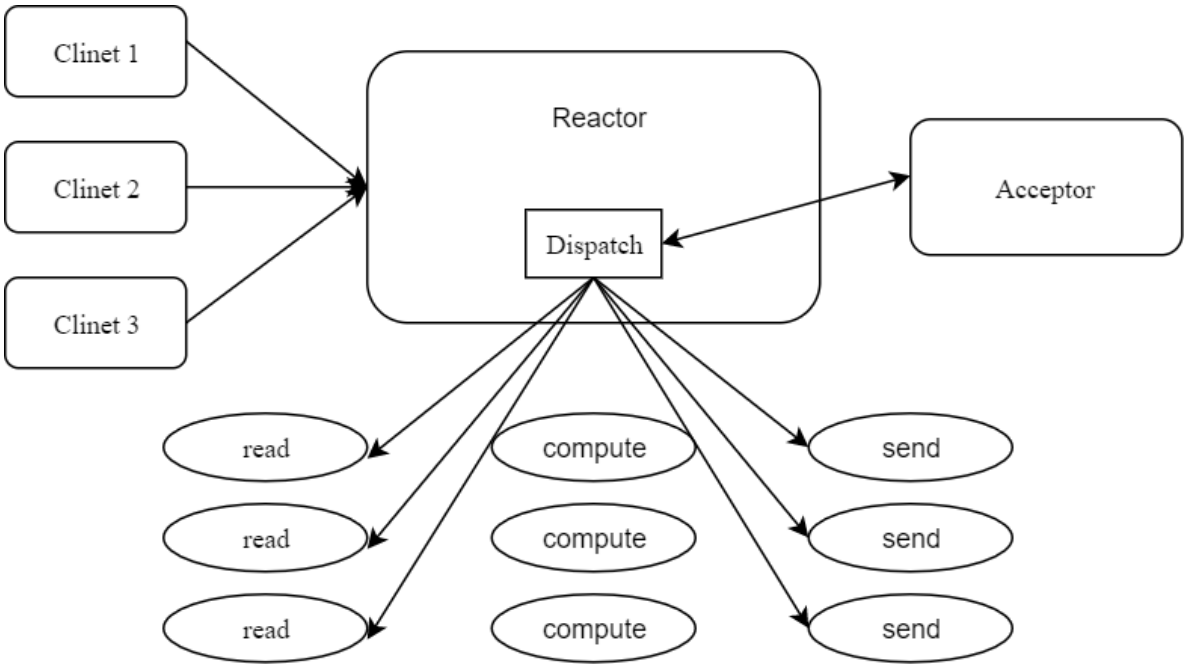

图 2-5 Reactor 单线程模型

2.2.3 Netty 框架

Netty 是一个利用 Java 语言高级网络的能力，隐藏 Java NIO 提供的 API 背后的复杂性而提供一个易于使用的 API 的高性能网络应用程序框架框架[28]。具有异步通信，事件驱动的特点。开发人员可以通过 Netty 快速轻松地开发出服务器和客户端等 Web 应用程序。它极大地简化并流水化了 TCP 和 UDP 套接字服务器等网络编程[29]。

Netty 为开发人员提供了更加“快速”和“简便”的开发方式，但这并不意味着最终应用程序会受到可维护性或性能问题的影响。 Netty 经过精心设计，吸取了多种实验的经验，如 FTP，SMTP，HTTP 以及各种基于二进制和文本的传统协议。最后，Netty 成功地找到了一种方法，可以在不影响开发的效用的情况，保证应用本身的性能，稳定性和灵活性[30]。Netty 的架构图如图 2-6 所示：

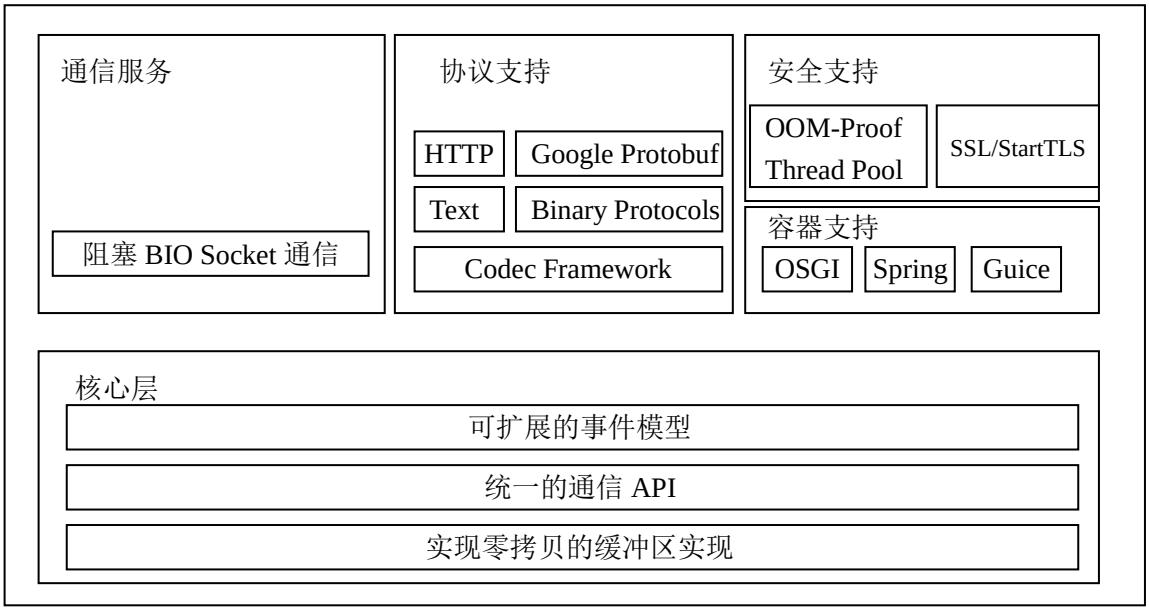

图 2-6 Netty 架构图

Netty 是建立在典型的 Reactor 模型结构之上的，并在这种经典的模型基础之上引入了多线程——主从 Reactor 模型。Netty 将 Reactor 细分为 mainReactor 和 subReactor，二者分别于与 Boss 和 Worker 工作线程池相对应[31]。Boss 线程主要负责处理客户端的连接建立以及连接建立过程中的认证工作，Worker 线程负责处理 IO 操作及后续一系列操作。一般情况下，当数据经 IO 操作处理完之后，交给 handler 处理，然后再交给业务线程池进行业务处理[32]。其中主从 Reactor 模型如图 2-7 所示：

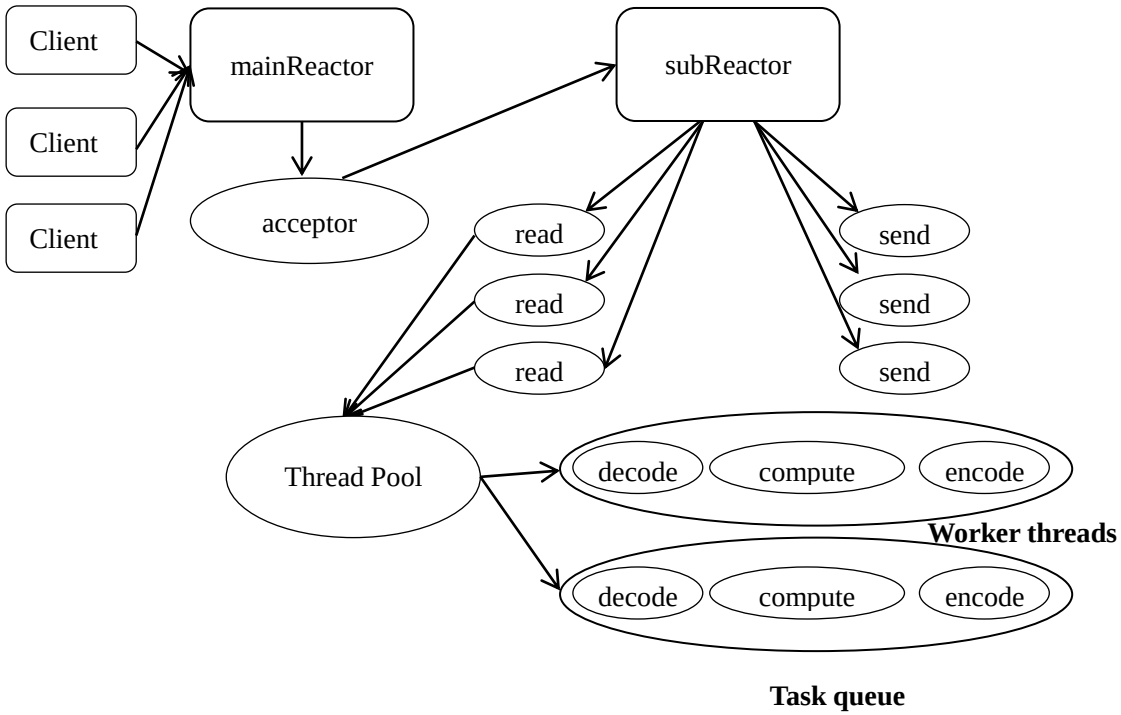

图 2-7 主从 Reactor 模型

Netty 拥有六大核心组件，它们分别是 Channel、ChannelFuture、EventLoop、ChannelHandler、ChannelPipeline。下面将逐一介绍它们的作用：

1）Channel

Channel 是 Netty 网络操作的抽象类。它包括基本的 I / O 操作，如 bind，connect，read 和 write。它还包括一些与 Netty 框架相关的功能，例如获取 Channel 的 EventLoop。在传统的网络编程中，其核心类 Socket，它对程序员来说并不那么友好，而且使用它直接进行程序开发的成本较高。Netty 的 Channel 则提供了一组 API，可以大大降低直接使用 Socket 的复杂性。

2）EventLoop

Netty 基于事件驱动的模型，该模型使用不同的事件来通知我们状态的变化或运行状态的变化。EventLoop 定义了在整个连接生命周期中发生事件时处理的核心抽象。Channel 是 Netty 网络操作的抽象类。EventLoop 主要配合 Channel 的完成 IO 操作。当一个连接请求到达时，Netty 会为这个链接注册一个 Channel，然后从

EventLoopGroup 分配出一个 EventLoop 绑定到这个 Channel 上，这个 EventLoop 将会在 Channel 生命周期内为其服务。

3）ChannelFuture

Netty 是异步非阻塞的，所有 IO 操作并不是以同步的方式顺序执行的，因此我们无法立即知道消息是否已被处理。Netty 提供 ChannelFuture 接口，该接口通过接口的 addListener()方法注册 ChannelFutureListener。当操作成功或失败时，监听器将自动触发返回结果。

4）ChannelHandler

ChannelHandler 是 Netty 的核心组件。ChannelHandler 主要用于处理各种事件。这里的所提到的事件很宽泛的概念，它可以使连接，数据接收，异常，数据转换等事件。ChannelHandler 有两个核心子类，ChannelInboundHandler 和 ChannelOutboundHandler，其中 ChannelInboundHandler 用于接收和处理入站数据和事件，而 ChannelOutboundHandler 则相反。

5）ChannelPipeline

Netty 将 Channel 中的数据管道抽象成 ChannelPipeline，作为 ChannelHandler 的容器。ChannelPipeline 中会以链表的形式组织若干个 ChannelHandler，这些 ChannelHandler 会在 ChannelPipeline 的管理和调度下，以某种特定的顺序依次执行。在事件处理过程中，一个 ChannelHandler 接收数据后处理完成后交给下一个 ChannelHandler，或者什么都不做直接交给下一个 ChannelHandler。

2.3 分布式一致性服务

分布式一致性指的是，在分布式环境中一个系统各个组件分布在不同的网络计算机上，彼此之间通过网络通信进行协同工作，并保证每台主机上的副本之间能够保持一致。如果一个系统其中一个副本信息发生变化，所有节点上的副本信息都能及时感知并更新，那么系统就被认为具有强一致性[33]。本框架中需要提供一致性服务的模块体现在服务注册模块，当服务端有服务注册时，客户端需要能够及时感知，并进行调

用。同理，当服务下线时，客户端也需要及时感知，避免调用无效的服务。本框架将采用 ZooKeeper 作为框架服务的注册中心。

2.3.1 ZooKeeper 简介

ZooKeeper 是一种集中式服务，用于维护配置信息，命名，提供分布式同步和提供组服务[34]。这些服务都是以分布式应用程序的形式存在的，这些应用程序之间通过网络进行通信，定时交换各自的信息，用于维持服务之间的信息一致性。

2.3.2 ZooKeeper 核心概念

1）集群角色

在服务器集群中，每个服务器节点都有隶属于集群的一个角色。在 ZooKeeper 中，节点的角色被分为三类：Leader、Follower 和 Observer[35]。Leader 节点是集群工作的核心节点，负责和客户端直接进行通信。Follower 节点听从 Leader 节点，并参与 Leader 节点的选取。Leader 节点会维持与 Follower 节点的心跳，接收 Follower 节点请求并判断 Follower 节点的请求消息类型。Observer 节点充当一个观察者的角色，不参与 Leader 节点的选举，它的存在能够提升集群整体读的性能。

2）会话

会话是指客户端与 ZooKeeper 服务器之间的建立的 TCP 长连接，也称为 Session[36]，如图 2-8 所示。为了维护会话，客户端首先在服务器启动时与服务器建立 TCP 连接。客户端的生命周期随着客户端与服务端第一次连接开始，这个连接可以用于客户端与服务端之间的心跳检测，用于保持和服务端之间的有效会话，还可以用于请求的发送和接受 ZooKeeper 服务器对于请求的响应。不仅如此，客户端还可以通过接收到来自服务器的 Watcher 事件通知。在会话超时时间内，连接中断后立马重新连接并成功，那么次会话仍然是有效的。

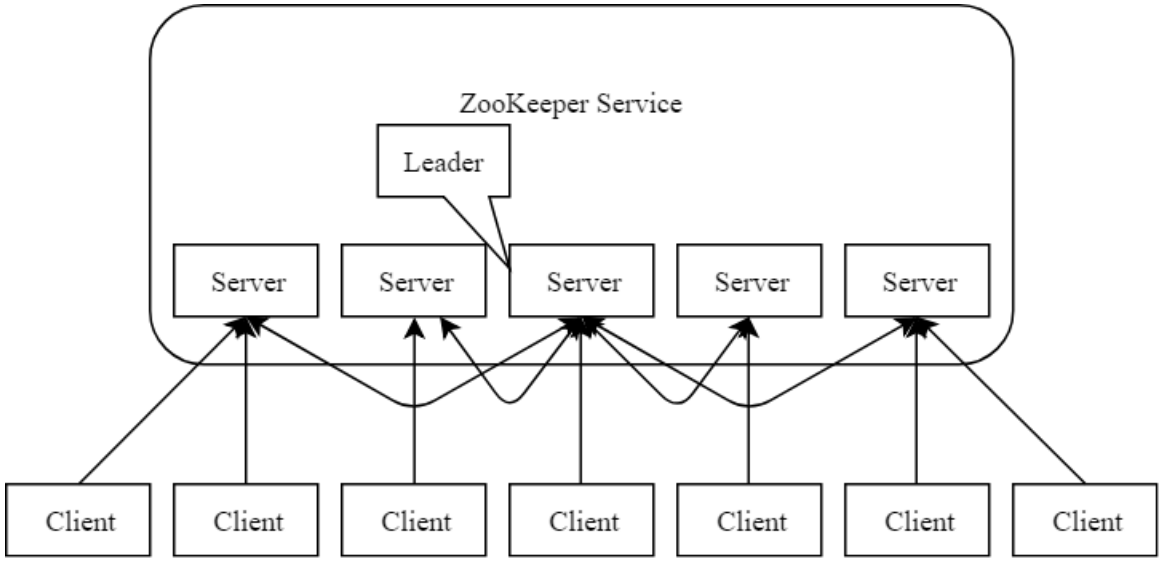

图 2-8 ZooKeeper 系统模型

3）数据节点

ZooKeeper 数据模型是由一个个数据节点（ZNode）组成的分层树形结构[37]，如图 2-9 所示。每个节点中存放着该节点的属性信息及自身的数据内容。和传统文件系统不同的是，文件系统目录节点本身不存放自己的数据内容，只存放子节点。

数据节点按生存周期可以分为持久节点(PERSISENT)和临时节点(EPHEMERAL)[38]。持久节点一旦创建，只要不人为的进行节点删除操作，这个节点的数据一直保存在 ZooKeeper 上。而临时节点数据会伴随客户端会话的失效而被删除。节点还可以通过设置 SEQUENTIAL 属性，完成对节点名按照添加顺序添加序号，该功能十分有利于实现分布式同步原语的实现[39]。

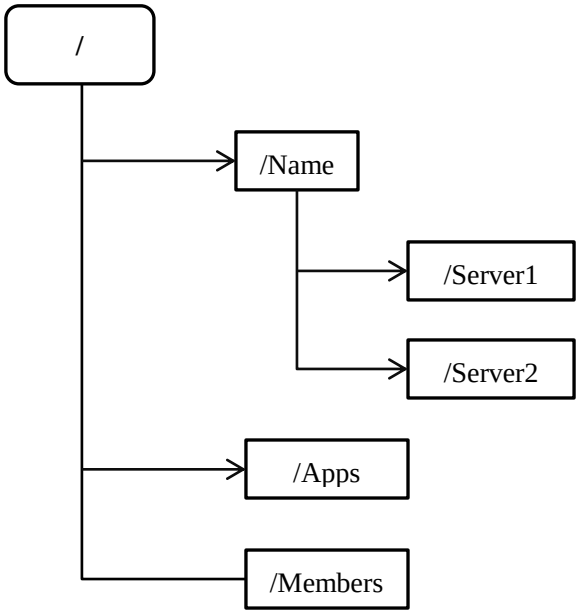

图 2-9 ZooKeeper 数据模型图

4）版本

ZooKeeper 中的版本是用来记录是节点数据或者是节点的子节点列表或者是权限信息的修改次数。一个节点如果其版本号为 1 的话，则表明这个节点从创建以来被修改过一次。版本的信息主要通过维护一个 Stat 的数据结构来实现的。在 ZooKeeper 中版本类型有三种，分别是 version、cversion、aversion。它们分别表示当前数据节点内容的版本号，当前数据节点子节点的版本号以及当前数据节点 ACL 变更的版本号。

5）Watcher

Watcher 可以理解为 ZooKeeper 自带的一个事件监听器[40]。用户通过客户端在指定节点上注册一些 Watcher，那么当指定节点状态发生变化时，ZooKeeper 会将这个变化通知给注册了监听（感兴趣）的客户端。当两个客户端在 ZooKeeper 集群中注册了 Watcher，那么当 ZooKeeper 中的数据节点状态发生变化时，客户端会受到来自 ZooKeeper 的通知，客户端也可以在接受这个通知后向 ZooKeeper 请求状态变更节点的详细信息。

2.3.2 ZooKeeper 客户端 API

ZooKeeper 的客户端与服务端进行交互主要是通过客户端 API 调用来实现的。通过客户端 API 的调用可以完成所有有关 ZooKeeper 操作。客户端 API 所有调用都不会因其他线程执行而阻塞，所以最终都能得到 API 执行的返回结果，即 wait-free[41]。常见的客户端 API 如表 2-1 所示。

按照是否需要等待操作结果的返回，操作可以分为同步和异步两种。同步 API 的执行需得到返回的执行结果后再进行后续的指令操作，异步 API 的执行则不需等待结果的返回即可执行后续相关指令。异步 API 执行最终结果可以通过响应执行回调得到。

表 2-1 ZooKeeper 客户端常用 API

API 名字	说明
create(path, data, flags)	根据路径和数据创建数据节点，并通过 flags 指定节点的类型
delete(path, version)	删除指定版本和路径下的数据节点
exists(path, watch)	查看指定路径下的节点是否存在，并通过 watch 参数设置是否开启监听
getData(path, watch)	根据路径获取节点数据值，并通过 watch 参数设置是否开启监听
setData(path, data, version)	如果数据节点的版本为 versoin，则更新 path 路径下的数据值为 data
getChildren(path, watch)	获取 path 路径下的孩子节点，并通过 watch 参数设置是否开启监听

客户端 API 中 exists()，getData()，getChildren()等方法均可通过给定参数，设定是否在操作所对应的节点上开启 Watcher 监听。开启监听后，客户端可以监听到节点状态发生的变化，该过程是通过 WatcherEvent 事件通知给感兴趣的客户端。

2.4 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。

3 分布式 RPC 框架的需求分析与设计

系统的垂直应用不断扩大，将一个系统在垂直结构上进行拆分，拆分后各个模块之间协同工作的需求便是 RPC 框架产生最大的需求。本章将从本文所研究的 RPC 框架的业务需求分析出发，并在此基础上对本框架的功能性与非功能性两方的需求进行分析。在完成本框架的需求分析后，根据需求分析得到的结果对本框架进行设计。

3.1 业务需求

传统的 MVC 架构将业务逻辑耦合度不高的各个业务模块都集成到同一个系统中。通过前后端分离技术，用户通过前端页面发起请求，通过代理服务器完成均衡负载过程，这个工作模型如图 3-1 所示[42]。在系统规模较小时，此种架构能够对系统起到良好的支撑作用，并且系统在后期维护所需的成本也是可控的。但随着业务的不断完成，系统的规模也在不断扩大，而此时仍然采用 MVC 架构作为系统的架构将导致系统难以维护以及后期版本的迭代成本过高。

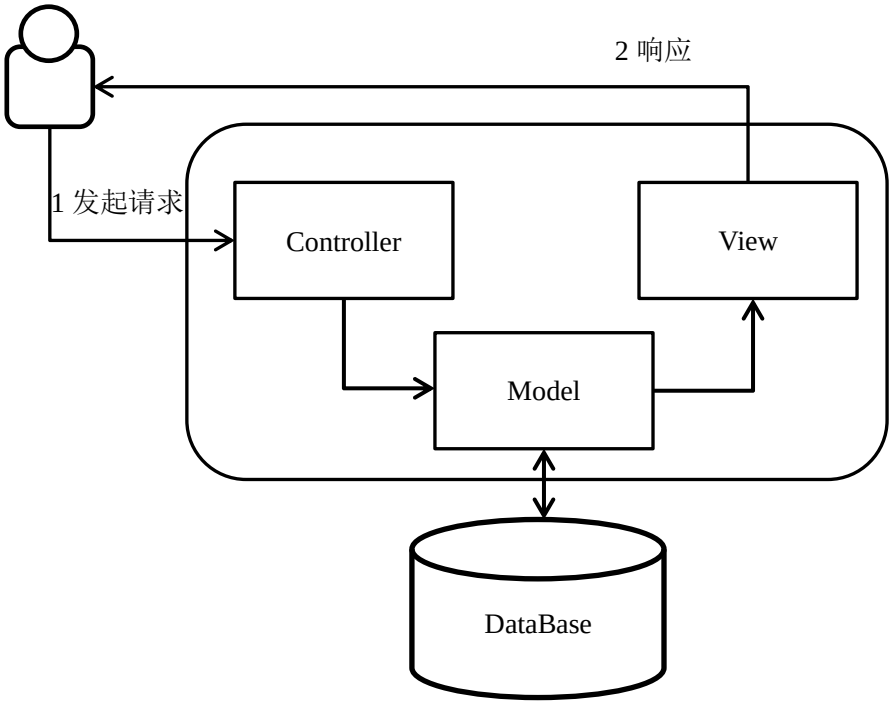

图 3-1 MVC 模式架构图

将系统按业务对其进行模块拆分，各个模块通过协议及网络完成通信，便是解决上述问题的方式之一，也是大型系统所面临的需求之一。同时该业务需求也是本文所研究的 RPC 框架的业务需求。庞大并且复杂的系统，从业务逻辑角度进行拆分，拆分后的各个模块通过RPC框架完成通信[43]。进行拆分后的系统工作结构如图3-2所示。在保证系统的业务功能的前提下，有效的减少系统维护、迭代的成本。

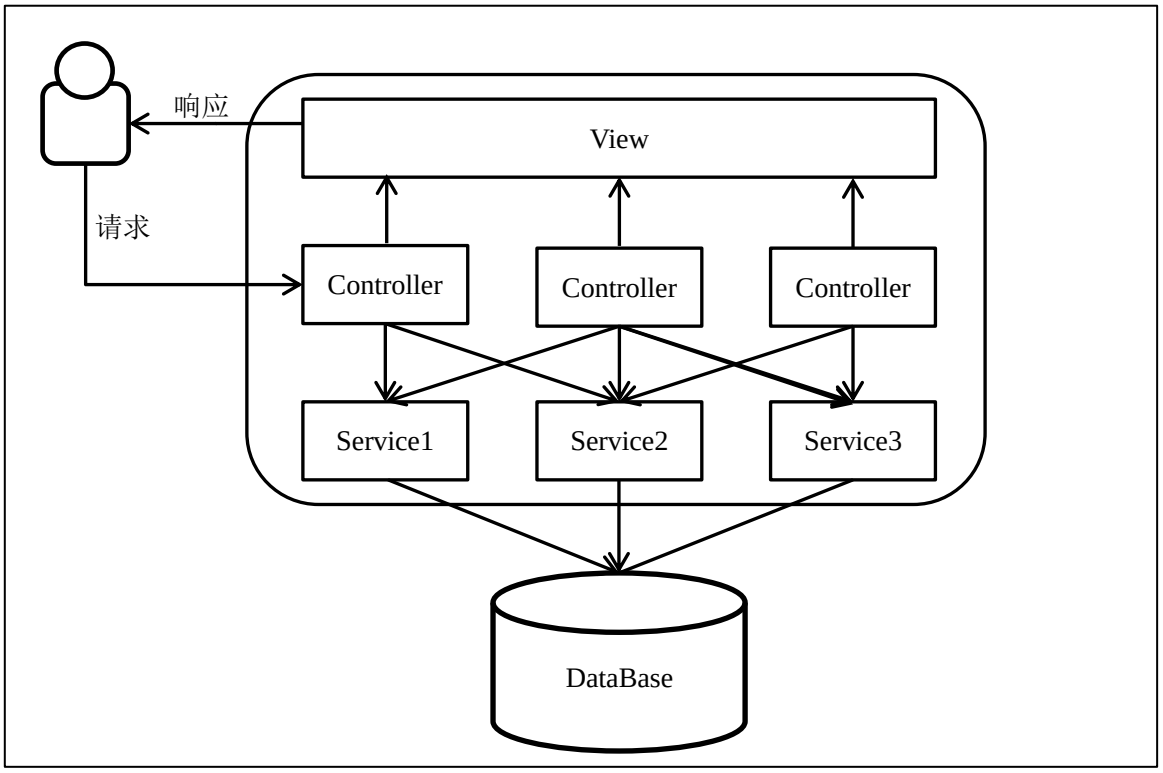

图 3-2 RPC 架构图

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。

3.2 功能性需求

RPC 框架的核心业务可以简单的描述为服务提供方（Provider）通过 RPC 框架对外发布，服务调用方（Consumer）通过 RPC 框架调用服务并获得结果。该业务的实

现在底层依赖于一个个功能模块。将 RPC 框架按功能模块进行划分可以分为数据处理模块、服务发布模块、服务调用模块。本节将对这些模块进行功能性需求分析。

3.2.1 数据处理功能

数据处理功能主要是 RPC 框架的通信中心。服务提供方（Provider）通过 RPC 框架完成服务的发布，服务调用方（Consumer）通过 RPC 框架调用已发布服务。该过程需要完成数据的传输以及数据的交换。其用例图如图 3-3 所示：

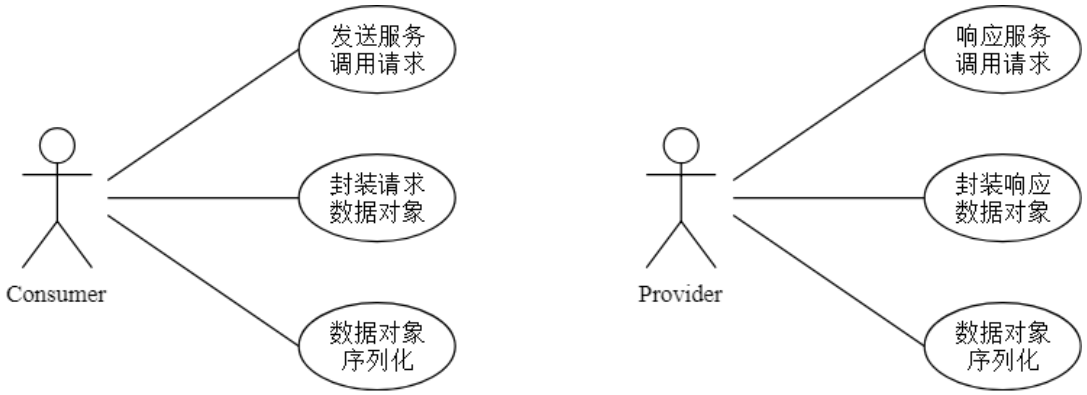

图 3-3 数据处理用例图

1）网络传输：RPC 框架是建立在网络通信之上的远程调用框架。网络通信最主要的目的便是实现数据在网络上的传输，网络传输是 RPC 最重要也是最基础的功能。服务提供方与服务调用方都需要通过 RPC 框架完成信息交互，即服务提供方和服务调用方需要通过 RPC 框架发送数据以及接收数据。

2）对象的序列化与反序列化：数据在网络中的传输只能是以二进制的形式进行。本框架在使用过程中需要发送 Java 数据对象，Java 数据对象并不能直接通过网络进行传输，因而 RPC 框架需要具备将 Java 数据对象转换成二进制数据，在将二进制数据发送至网络，即序列化功能[44]。同时还需要具备将二进制数据还原成数据对象的功能用于接受网络中的数据，即反序列化功能。

3）数据封装对象：RPC 框架通信双方需拟定好通信数据格式，根据拟定好的数据格式来才能正确解析自己接收到对方发送过来的数据。因此 RPC 框架需为通信双方拟定好请求与响应的数据格式，如图 3-3 中的 RpcRequest 与 RpcResponse 对象数据，这两个对象分别用于封装请求数据与响应数据。

3.2.2 服务发布功能

服务发布功能是 RPC 框架的核心功能之一，该功能需要为服务提供方完成服务的发布。在面向对象编程思想中，服务与服务对象相对应，服务的调用映射到服务对象上的方法调用。RPC 框架需要获取服务对象，并能完成该对象的调用。同时该框架还需提供服务注册功能，将服务对外发布。该功能的用例图如图 3-4 所示：

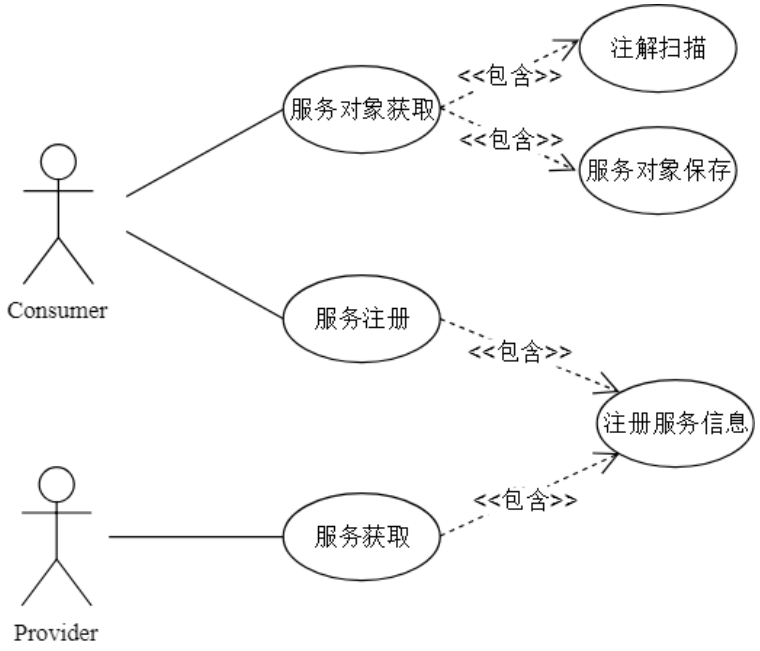

图 3-4 服务发布功能用例图

1）获取服务对象功能

服务对象是真正完成业务处理的对象。RPC 框架在接受服务调用请求，需要根据请求内容，完成服务调用。而在调用之前，RPC 框架需要获取到提供了服务的服务对象，并在请求来了之后调用该对象的具体方法，从而完成服务的调用。

2）服务注册功能

服务提供方在服务发布后需要通知对该服务感兴趣的服务调用方服务已上线。本框架采用注册中心的方式对该功能进行实现。RPC 框架提供的注册中心需要将已经注册的服务提供方进行记录，当服务调用方需要发起服务调用请求时可以在注册中心获取服务提供方的具体信息，完成服务调用。

3.2.3 服务调用功能

服务调用是 RPC 框架中的最为核心功能。服务调用端通过 RPC 框架能够完成服务的准确调用。由于相同的服务可以部署在多台不同的主机上，因此 Consumer 端在进行服务调用时，需进行服务调用路由，选出合适主机发送服务调用请求。由于服务的调用是远程调用，故不能直接对服务对象进行调用，需有服务的代理对象远程完成该过程。其中服务调用功能用例图如图 3-5 所示。

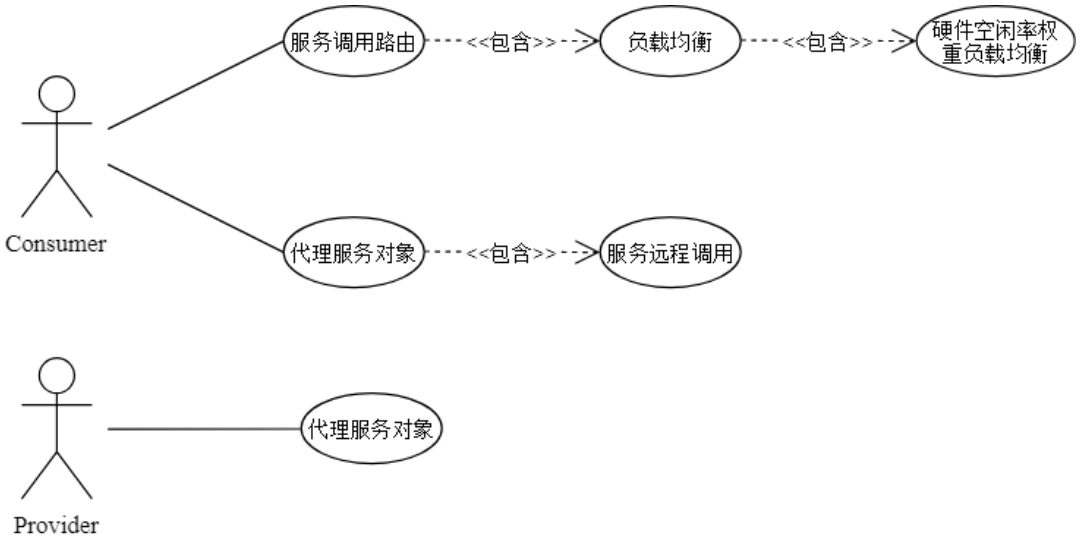

图 3-5 服务调用功能用例图

1）对象代理功能

服务的调用需要通过网络发送请求，并在得到响应后解析出结果作为返回值。该过程涉及到请求封装和网络通信，而这个过程应该对服务调用方来说是透明的，因此该过程需要由 RPC 框架来完成。本框架采用对象代理的方式[45]，来完成上述过程。代理对象需要完成请求封装，并发送请求，在接收到响应后，解析出调用结果并返回给服务调用方。

2）服务调用路由功能

RPC 框架包含的注册中心允许统一服务的多机部署，并同时对外提供服务。对于一个服务调用请求，任意一台主机上的服务均可进行调用。此时 RPC 框架需要具备服务调用路由的功能。RPC 框架需要根据某种均衡负载策略对服务调用进行路由选择。根据某种负载均衡策略选出最合适的主机，并发送请求到该主机，进行服务调用。

3.3 非功能性需求

在保证功能性需求得到满足的前提下，为了使得本框架能够提高用户体验，本文还对本框架进行了非功能性需求的分析，并在设计上给出了具体的实现。与功能性需求不同的是，非功能性需求并非必不可少，但却能大幅提高使用该框架的开发人员的使用体验。本框架在完成必要功能的前提下将从系统的性能、易用性、可靠性、扩展性进行需求分析。

1）框架高性能

RPC 框架的引入将增加服务调用的步骤同时还引入了具有不确定因素的网络通信，这都将致使系统性能下降。服务调用步骤的增加，将直接导致从请求调用发出到请求响应的时间。而引入网络通信，如果框架没有处理好网络通信，那么框架性能将大幅度受损。本框架最重要的非功能性高性能需求便是系统进行拆分后的各个业务功能模块在使用了本框架作为消息中间件后，其服务请求的响应速度不会让服务调用方感觉到有明显下降。因此本框架需要能够快速完成服务调用请求的发出，以及响应解析后立即返回。在网络通信层面，也将尽可能减少用并发访问过多而导致的阻塞。

2）框架易用性

框架的易用性是一个框架是否能够流行的必要条件。易用性的框架能够减少系统的开发成本、后期维护成本，且在需求需要进行修改时也能够轻松的对既有代码进行修改。RPC 框架作为一个消息中间件，需要尽可能的降低代码的入侵性，即框架代码尽可能少的出现在业务代码中。同时在使用 RPC 框架时，要足够简单，使用较少的操作步骤就能完成 RPC 框架的使用。在使用时，尽可能通过配置文件或注解完成。在降低使用难度的前提下，还应减少框架代码的入侵性。

3）服务可靠性

互联网应用的一大优势便是能够 7x24 小时无间断为用户提供服务。对于某些应用来说，服务的不确定中断是致命的，轻则导致用户体检变得糟糕，重则影响公司的营收。因此提供持续可靠的服务是每个应用所必须具备的特点。本框架作为消息中间件，对上层应用提供服务时，需要保证服务提供方发布的服务是可靠的，即服务能够提供 7x24 小时供其他应用进行调用。

单点故障是服务不可靠的的一个重要原因，解决该问题的常用方法便是多机共同对外提供服务。为了避免因单点故障导致服务不可靠，本文所研究的 RPC 框架需要

满足同一服务的多机部署，且能够功能对外提供服务。对于出现故障的服务器需对外停止服务，正常的服务器能继续对外提供服务。

4）服务扩展性

不同时期同一应用的并发访问量是有所不同的。“双十一”期间各大电商网站所面临的访问量可能是平时的几倍甚至是几十倍以上。为了使应用能够在该期间正常工作，电商企业需要大量的主机进行服务的访问支持，但是该期间一过，继续使用该规模的主机提供服务的访问，则会造成大量主机的浪费。因此，服务的动态水平扩展、收缩功能则显得尤为关键。

为了使系统能够在不同时间段满足不同的并发访问量，RPC 框架需要为应用提供动态上下线功能。服务可以在有需要的情况下直接进行热部署，一旦部署成功便可直接对外提供服务。同时服务也可以随时下线，且下线后，将不会有服务调用请求到达该服务器。

3.4 架构设计

本框架最核心的两大组件便是 RPC 框架的服务端 Rpc-server，和 RPC 框架的客户端 Rpc-client。这两大组分别为上层应用的服务发布方与服务调用方所调用。Rpc-server 在启动的时候便将服务的地址和硬件占用情况等信息注册到 ZooKeeper 上。Rpc-client 启动通过配置文件获取到 ZooKeeper 地址，从而得到服务调用地址。二者通过 Netty 发起请求以及请求的响应。RPC 框架的架构图如图 3-6 所示。

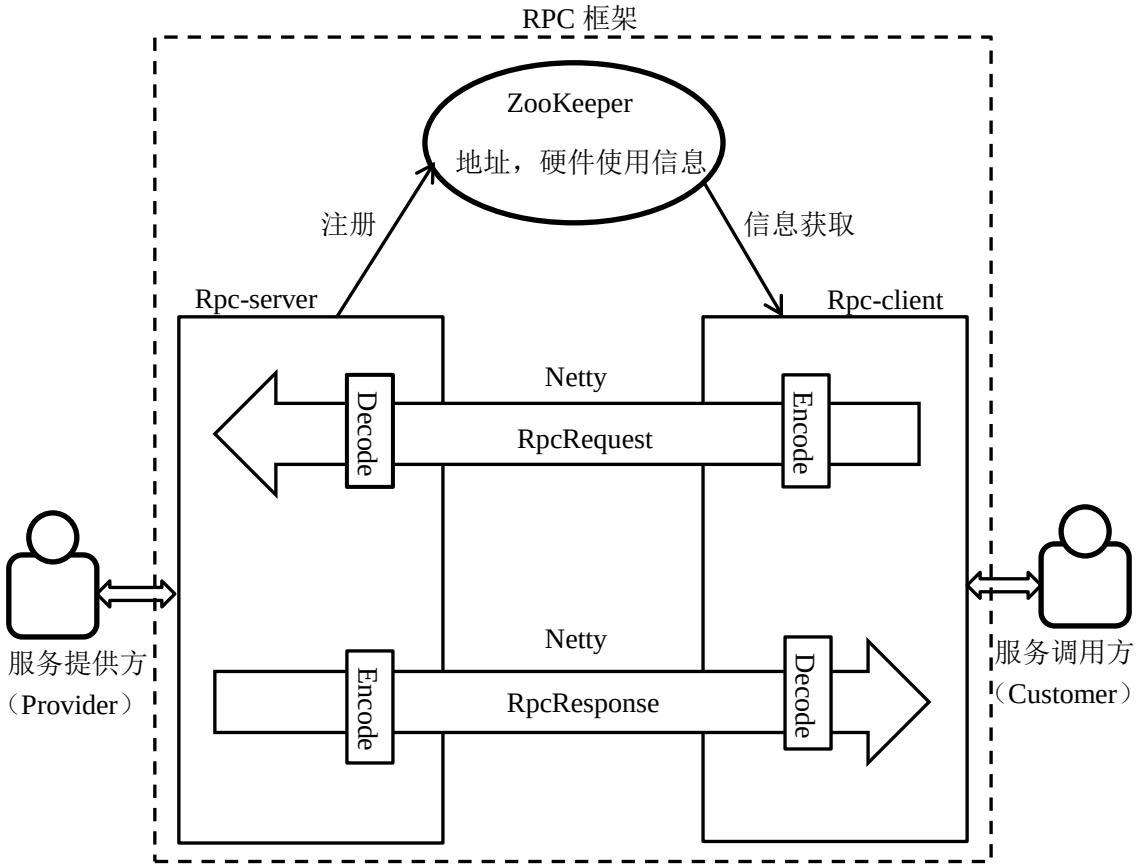

图 3-6 RPC 的框架结构图

3.4.1 服务注册中心

服务注册中心是服务对外通知以及服务地址的获取中心。服务提供方通过在注册中心注册后，在 ZooKeeper 中保存了自己的配置信息。服务调用方通过对注册中心的访问发现目前已经发布的服务的配置信息。从而完成服务的发布、获取的功能。

注册中心在设计上采用发布/订阅的形式，完成服务对外发布与服务的调用。服务提供方通过连接 ZooKeeper，并将自己的 IP 地址、端口号以及硬件占用情况作为数据节点信息，保存到 ZooKeeper 上。服务调用方通过连接 ZooKeeper 获取到目前能够进行服务调用的主机信息。并通过服务调用路由选择一台主机作为自己服务调用对象。

3.4.2 服务端 Rpc-server

RPC 框架的 Rpc-server 主要用于和服务提供方进行交互、调用。服务提供方只需调用 Rpc-server 对外提供的接口，以及完成相应的配置便能完成服务的发布。服务提

供方无论是发送响应数据，还是接收请求数据都是通过 Rpc-server 完成的。因此 Rpc-server 需要具备网络通信的功能和数据对象的序列化与反序列化功能。

Rpc-server 中的通信功能基于 Netty 进行定制的。使用 Netty 能够快速处理大量网络请求。因此在 Rpc-server 通过引入 Netty 作为底层的通信框架来完成请求响应与请求的接收。不仅如此 Rpc-server 还需要对响应数据进行序列化用于网络传输，对请求数据进行反序列化，用于还原请求数据对象。对于数据对象的序列化与反序列的实现，在 Rpc-server 中采用了 Google 开源的 Protostuff 框架。

3.4.3 客户端 Rpc-client

RPC 框架的 Rpc-client 主要供服务调用方调用。服务调用方通过 RPC 框架，对远程服务进行调用，并通过 RPC 框架获取远端服务的调用结果。服务调用方在整个服务调用过程中只需要和 RPC 框架进行交互即可，剩余的调用过程全部由 Rpc-client 完成。故 Rpc-client 需要具备网络传输功能，往服务提供方发送请求，还需接收服务提供方发回的响应数据。

为了使远程服务调用过程透明化，Rpc-client 在设计上将采用对象代理的方式，完成从请求发送到接收响应的全过程。Rpc-client 为服务调用方提供一个代理对象，该对象将服务调用方的请求封装成数据对象，并通过 Netty 发送给服务提供方。当服务调用完成后，继续通过 Netty 完成响应数据的接收并解析，将结果作为返回值返回给服务调用方。整个服务调用与响应过程全部由 Rpc-client 创建的代理对象进行完成。此种设计也是为了增加框架的易用性。

3.5 功能模块设计

本框架按功能进行模块划分，可以分为数据处理模块、服务发布模块、服务调用模块。这些模块又是由各个子模块构成。数据处理模块由数据传输模块、数据序列化模块和数据对象封装模块构成。服务发布模块由服务注册模块、待发布对象获取模块构成。服务调用模块由对象代理模块、服务路由模块构成。其中模块构成图如图 3-7 所示。

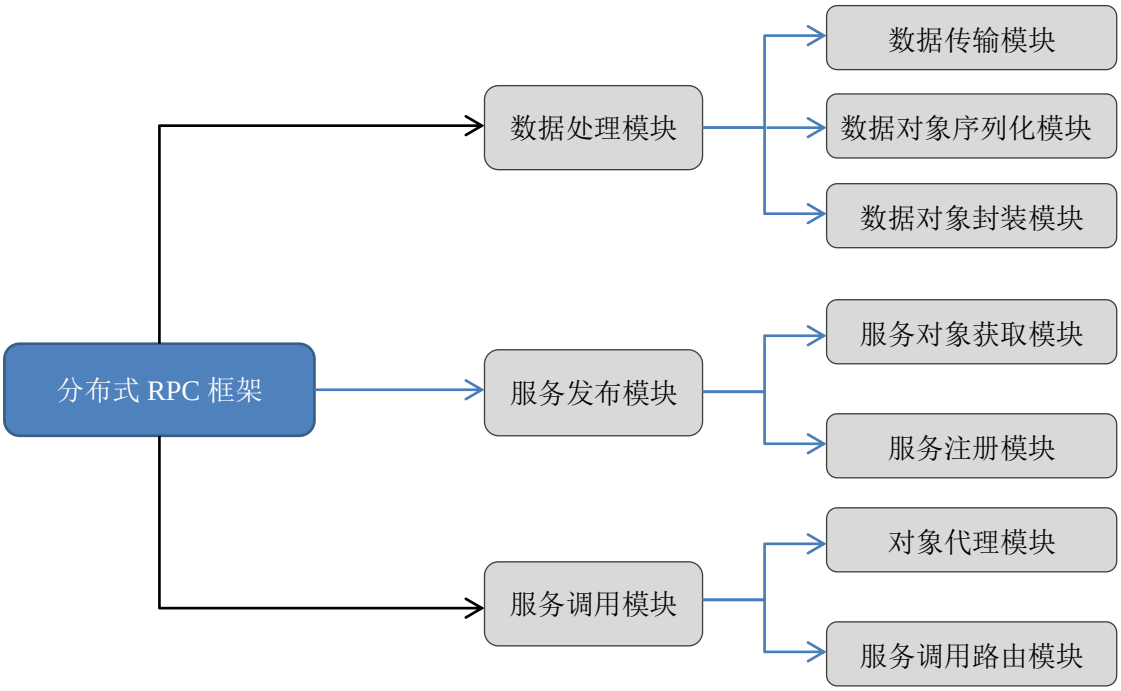

图 3-7 RPC 框架模块构成图

在对各个功能模块细分之后，下面将对每个功能模块下的各个子模块进行介绍以及设计层面的分析。

1）数据传输子模块介绍

数据处理模块主要有由数据传输模块模块，数据对象封装模块以及数据对象序列化模块构成。

（1）数据对象封装模块用于对请求数据和响应数据进行封装，其中请求数据封装成 RpcRequest 数据对象，请求响应数据封装成 RpcResponse 数据对象。数据对象每个字段有特定的含义，通信双方可以根据字段名获取对应的数据。

（2）数据序列化与反序列化属于数据通信模块。数据序列化过程发生在数据发送之前，包括请求和响应数据发送之前。数据反序列化过程发生在数据接收后，反序列化后的数据一般交给服务对象进行业务处理。数据的序列化和反序列化会成对出现。

（3)数据传输模块负责 RPC 框架客户端 Rpc-client 和服务端 Rpc-server 的通信，采用 Netty 作为数据通信框架。

2）服务发布子模块介绍

服务发布模块主要有由服务注册模块，服务对象获取模块构成。

（1）服务注册模块主要负责服务提供方的注册，为了提供分布式一致性服务，保证多态服务主机能协同工作，本模块采用 ZooKeeper 作为服务注册中心，根据 ZooKeeper 本身的特点，服务提供方能够做到动态上下线，以及服务提供方的水平扩展。

（2）服务对象获取模块用于服务对象的获取。本框架通过 JDK 自带的注解与 Spring 配合使用，通过扫描源文件是否带有@RpcService 注解判断该对象是否对外发布服务，如果该对象需要发布服务，则将该对象接口名称作为 key，服务对象本身作为 value 存在 HashMap 中，便于后续的反射调用。

3）服务调用子模块介绍

服务发布模块主要有由服务调用路由模块，对象代理模块构成。

（1）服务调用路由模块起到服务的负载均衡调用的作用。服务提供方将自己处理器空闲率和内存剩余大小在注册时写入 ZooKeeper 数据节点中，客户端通过 ZooKeeper 各个服务端硬件资源占用率来作为负载均衡因子，优先调用硬件资源占用率较小用的服务端，实现服务调用路由功能。

（2）对象代理模块主要负责拦截客户端调用请求，并通过代理对象和 RPC 框架的服务端 Rpc-server 进行交互，并将代理对象获取到的响应结果作为服务的调用结果返回给上层应用。

3.5.1 数据处理模块设计

1）数据对象封装

数据对象的主要作用是用于封装客户端的请求以及服务端的响应数据。客户端按照某中规则将数据对象的各个字段进行填充，经过序列化后将数据对象转换成二进制流发送到网络。服务端接收到封装请求的数据对象二进制流，进行反序列化还原数据对象，提取对象各个属性值用于服务调用。服务调用得到的结果再次封装成相应的数据对象，经过服务端的序列化和客户端反序列化，最终客户端得到还原后响应数据对象，即响应结果。数据对象在实现上分别被定义成 RpcRequest 对象和 RpcResponse 对象，RpcRequest 数据对象为客户端请求数据对象，RpcResponse 数据对象为服务端

响应数据对象。其中 RpcResponse 对象类图如图 3-8 所示，RpcResponse 数据对象类图如图 3-9 所示。

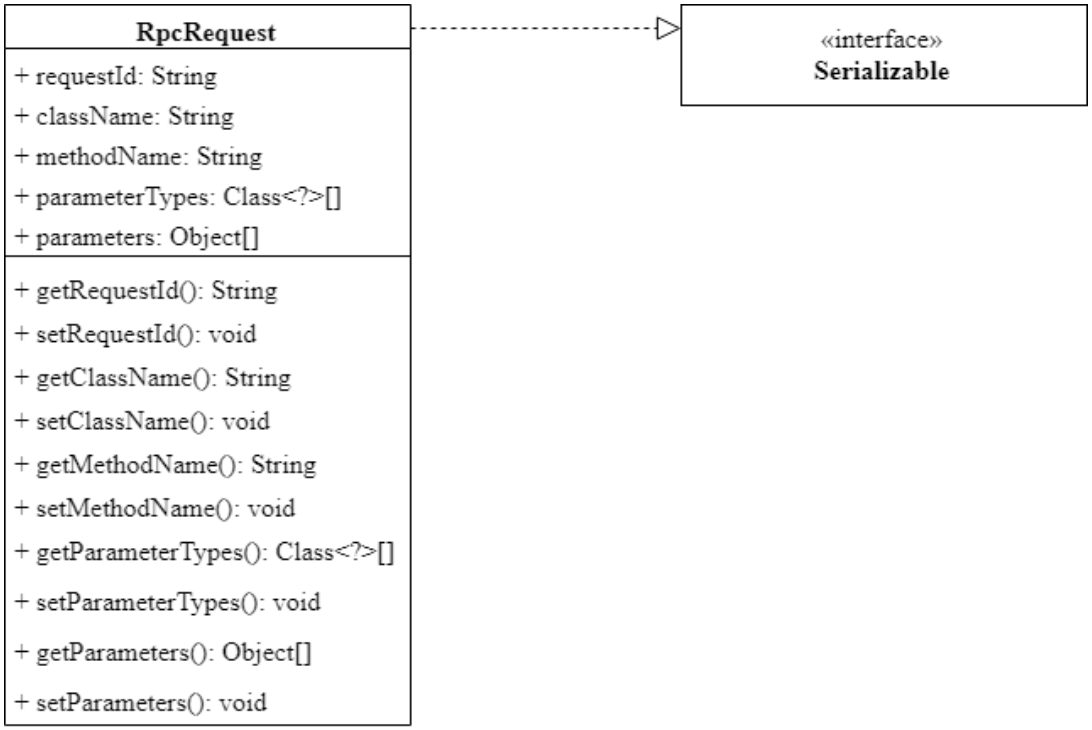

图 3-8 RpcRequest 对象类图

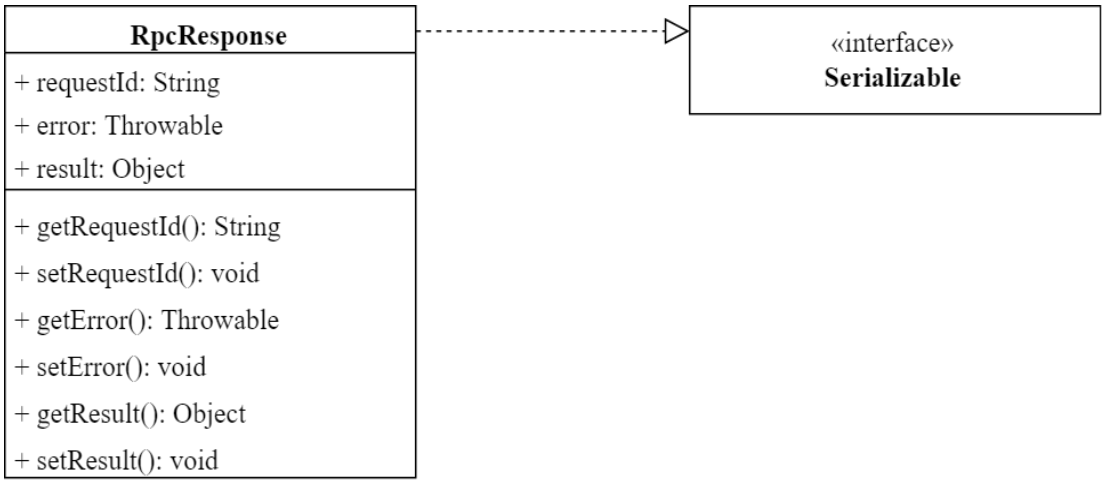

图 3-9 RpcResponse 对象类图

RpcRequest 对象实现了 Serablizable 接口目的是为了后续的序列化。该对象中共有五个属性以及属性对应的 get 和 set 方法。requestId 作为每次客户端请求的标识，根

据响应回来的 requestId 可以判断请求与响应是否对应。其余四个属性是 Java 通过反射机制调用方法所需要的参数。className 为待调用对象的权限定类名，将其作为参数，通过调用 Class.forName()方法，可以获得该类所对应的字节码对象 clazz。再通过 clazz.getMethod(methodName, parameterTypes)方法的调用获取到待调用方法对象 method。因为 RPC 框架本身持有一个 HashMap，且里面存放了服务对象以及服务对象的 className，利用 className 作为 Key 获取到对应的服务对象 serviceBean。最后根据 method.invoke(serviceBean, parameters)方法的调用，可以得到调用结果。

RpcResponse 对象同样实现了 Serializable 接口，其目的和 RpcRequest 实现该接口目的一致。该数据对象中共有三个属性以及属性对应的 get 和 set 方法。requestId 与 RpcRequest 中的 requestId 相对应，该值根据请求的 RpcRequest 中的 requestId 来进行设值，起作用确保请求与响应一一对应。属性 error 用于存放处理客户端请求时出现的各类异常，客户端可以根据该字段得知调用中出现的具体异常信息。属性 result 则是用于存放服务端服务调用的最终结果，客户端可以提取该属性，作为请求调用的最终结果。

2）数据对象序列化

数据对象序列化指的是将具有状态的数据对象通过序列化工具将其转换成二进制数据。反序列化则是序列化的逆过程，即将数据对象的二进制数据转换成原数据对象。本文所采用的序列化工具为 Google 开源的 Protostuff。Protostuff 有着出色的性能和良好的兼容性，可以为不同语言编写对象进行序列化与反序列化服务。

3）数据传输

该子模块在设计层面采用了目前较为流行的异步通信框架 Netty。数据传输主要是针对服务端 Rpc-server 与客户端 Rpc-client 之间的数据通信。它们之间的通信过程如图 3-10 所示。网络通信双方在 Netty 中被分为服务端和客户端。它们分别工作在 Rpc-server 和 Rpc-client 中。其中 Rpc-server 通过 ServerBootstrap 启动 Netty 服务端，Rpc-client 通过 Bootstrap 启动 Netty 客户端。当 Netty 的服务端和客户端都启动之后，Rpc 框架就可以通过 Netty 进行数据传输了。无论是 Rpc-server 还是 Rpc-client，在数据接收之后首先对数据进行解码，即反序列化，还原数据对象，在数据发送之前先对数据进行编码，即序列化，用于数据在网络中的传输。

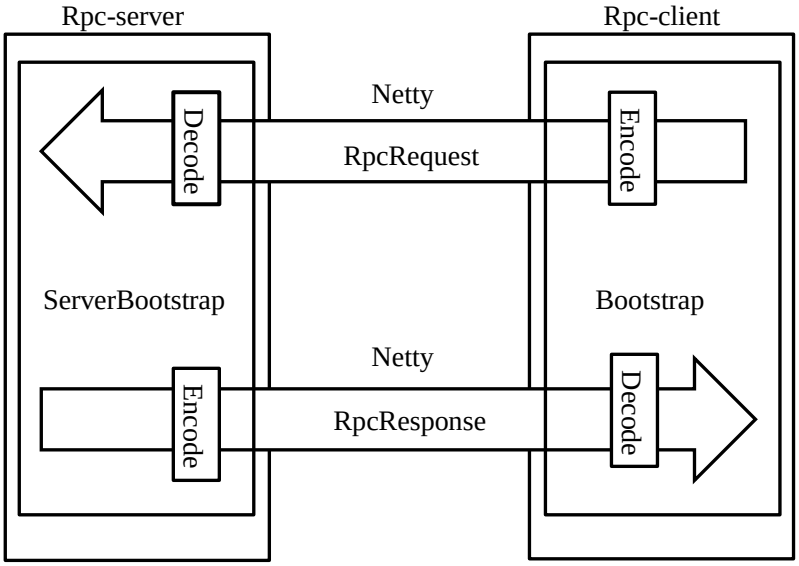

图 3-10 数据传输过程图

3.5.2 服务发布模块设计

服务发布模块是 RPC 框架服务端 Rpc-server 对外提供服务的核心模块，该模块在上层应用启动之时便获取那些将需要发布的服务对象，并保存在框架内部的容器内。同时该模块还提供了服务注册中心，供服务端进行注册。在注册完成后，服务便可对外发布。为了更好的说明服务发布模块设计与实现，在本节中将引入上层应用的概念。上层应用是调用 RPC 框架的外部应用，上层应用同样分为客户端可服务端，而两者的通信依赖于 RPC 框架中的客户端和服务端。其交互过程如图 3-11 所示，其中上层应用中的客户端命名为 User-Service，服务端命名为 User-Client，区别于 RPC 框架中客户端 Rpc-client，服务端 Rpc-server。

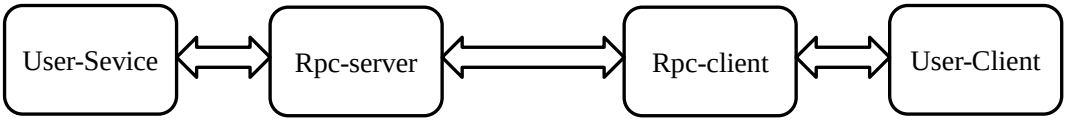

图 3-11 上层应用交互过程

1）服务对象获取

服务的调用，最终是以对象的方法调用形式进行的。服务的发布本质上是对象的方法对外发布，供客户端进行调用。即客户端通过 RPC 框架，远程调用服务端上的

方法。在服务发布之前需先获取对应的待发布对象，以便客户端进行远程调用之时，能够找到具体的服务对象，并完成对该对象的调用。

在本框架中，获取服务对象主要是通过 JDK 注解+Spring 注解扫描完成服务对象的获取。设计的主要思想是：上层应用 User-Service 通过在待发布服务对象类上添加@RpcService 注解标识该对象为服务对象。同时在上层应用中引入 Spring 框架，并开启注解扫描功能，当程序启动时，Spring 通过扫描对应的包目录，获取到包目录下添加了@RpcService 注解的类，并将这些类保存到 RPC 框架中的 HashMap 中，供后续客户端的调用。该过程主要流程如图 3-12 所示。

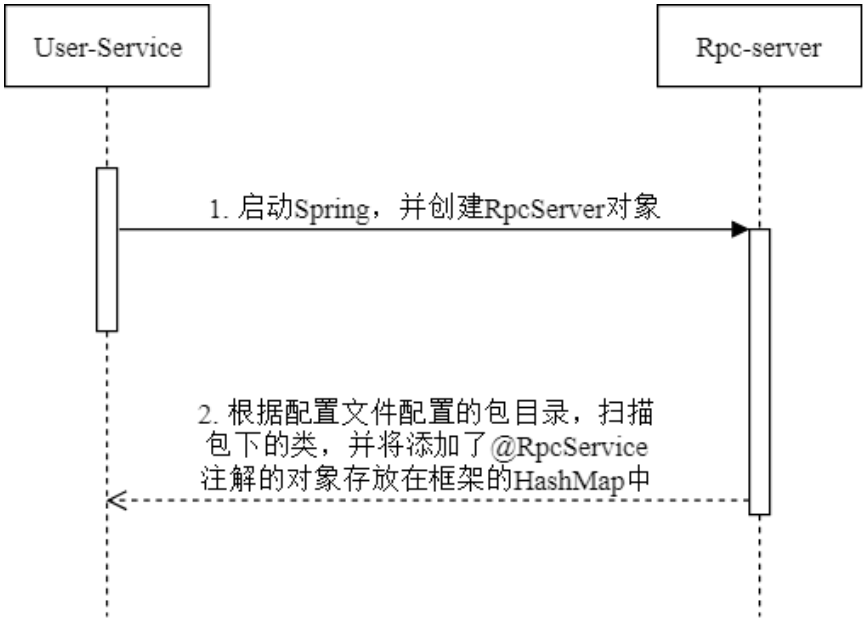

图 3-12 服务对象获取过程过图

2）服务注册

本框架的服务注册是基于 ZooKeeper 实现的。每台服务主机通过在 ZooKeeper 中创建一个临时数据节点。数据节点伴随着服务主机的上线而创建，伴随着服务主机的下线而删除。数据节点中存放的数据信息主要有服务主机的 IP 和端口号，主机的 CPU 空闲率以及剩余内存空间组成字符串。其中服务主机的 IP 和端口号用于客户端更具发起服务调用请求。后两组信息主要用于服务的负载均衡调用策略。客户端可以根据后两组信息对硬件资源占用率更低的服务器发起服务调用请求。

3.5.3 服务调用模块设计

1）对象代理

User-Client 在对 User-Service 提供的服务进行调用时，将该过程委托给服务接口的代理对象。User-Service 提供支持调用的接口，User-Client 通过导入该接口，有 RPC 框架为该接口创建代理对象。代理对象会拦截接口方法的每一次调用，将调用封装成 RpcResquest 对象并通过 RPC 框架完成 User-Service 的服务调用。其中，代理对象的创建采用 JDK 动态代理技术实现的。具体调用过程如图 3-13 所示。

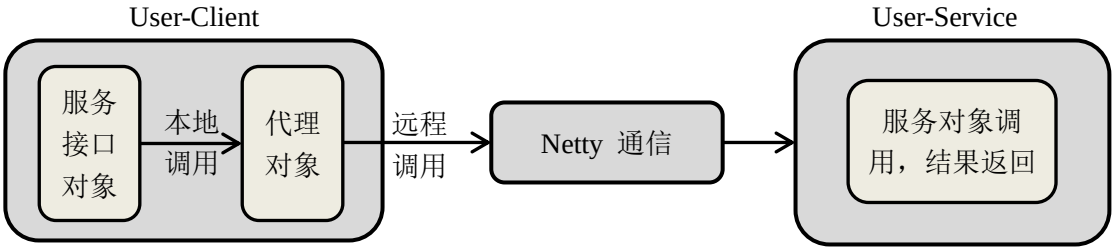

图 3-13 服务调用过程

RPC 框架中的 RpcProxy 类负责创建代理对象，利用 JDK 提供的动态代理技术完成代理对象的创建。代理对象在调用 User-Service 方法时，并非直接进行方法的调用，而是通过拦截 User-Client 的调用请求，并将请求封装成对应的 RpcRequest 对象。服务端接收解析后调用方法并返回响应结果。代理对象获取到服务端响应回来的 RpcResponse 对象，调用 Response.getResult()方法得到服务调用的结果，并作为远程调用结果，返回给调用者 User-Client。

2）服务调用路由

服务调用路由是指在服务端采用集群模式提供服务时，客户端的调用按照某种策略在集群中选出其中一台作为目标主机，发起服务调用请求。服务调用按照某种策略进行路由，起到均衡负载的作用。

本框架服务调用路由策略主要以服务端主机硬件占用情况，作为其路由决策值。客户端通过与注册中心 ZooKeeper 相连，读取指定路径下的数据节点，可以获取所有对外提供服务的主机地址，及硬件占用情况。客户端根据从 ZooKeeper 中获取到的数据，选择硬件占用率最低的主机发起服务调用请求，该过程如图 3-14 所示，其中

ZooKeeper 中存放的数据格式为 IP:port_ratioOfCPU#memoRemain，分别服务端地址、端口号、CPU 空闲率、内存剩余容量。此种策略可以充分的利用服务端硬件资源，尽可能快的对客户端进行响应。

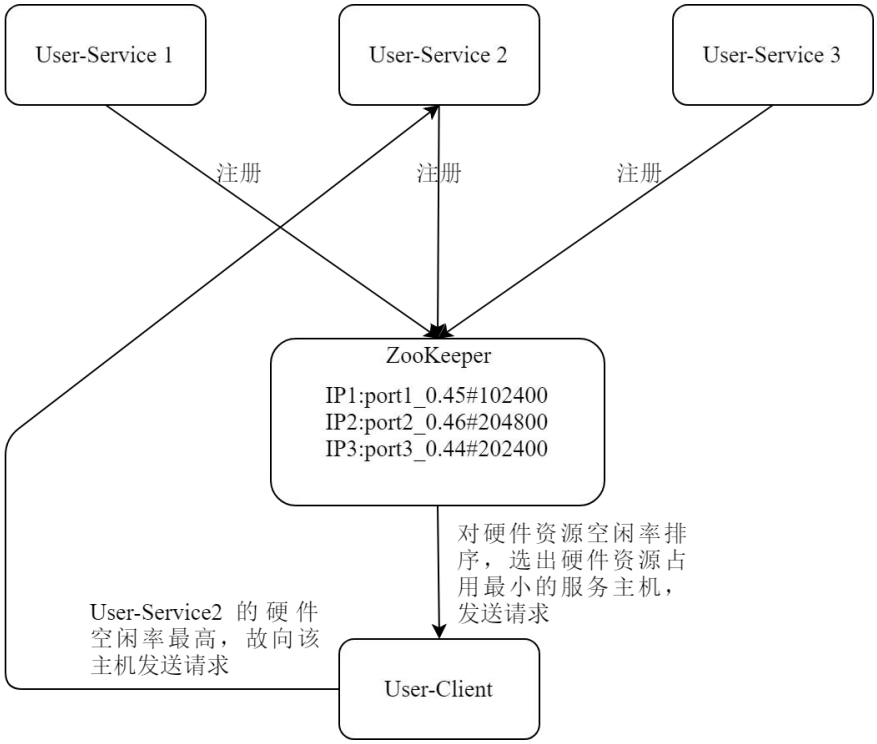

图 3-14 服务路由过程

3.6 本章小节

本章从 RPC 框架业务需求入手，让读者能够对 RPC 框架的使用需求有整体的认识。本章的第二节主要从 RPC 框架各个功能模块进行功能性需求的分析，将各个功能模块进行功能分解，并给出每个功能块的具体需求。紧接着对 RPC 框架的非功能性需求进行了分析，从多个角度如：高性能、易用性、可靠性、扩展性进行了具体分析。根据需求分析所得到的需求报告，给出了本 RPC 框架的框架结构设计，以及各个功能模块的设计。

4 分布式 RPC 框架的实现

本章内容涉及到本文所研究 RPC 框架具体实现，是本文的和核心章节。本章将在第三章给出的需求分析及框架详细设计的基础上对每个模块中各个组成部分给出具体的实现。

4.1 框架开发环境

本框架的实现主要是通过 Java 语言来完成的，在框架的开发过程中，采用的 Java 版本是 JDK 1.8。开发工具使用的是目前较为流行的集成开发环境 Idea，版本号为 IntelliJ IDEA 2018.1.1 (Ultimate Edition)。注册中心的实现是通过 ZooKeeper 提供的 Java API 进行调用实现的，其中 ZooKeeper 的版本号为 zookeeper-3.4.5。本文中所出现的部分流程图以及所有 UML 图均是通过在线画图工具 draw.io 完成的。系统的开发主机为一台操作系统为 Windows 10 的 PC。

4.2 数据处理模块实现

RPC 框架构建在网络通信之上，客户端的请求报文与服务端的响应报文均是通过 Socket 进行交互的。数据传输模块是 RPC 框架最基础的模块，该模块的性能将直接影响整个客户端与服务端通信速率。只有保证了数据传输的效率才能提高整个 RPC 框架的性能。

4.2.1 数据对象封装实现

数据对象是数据的载体，Rpc-server 与 Rpc-client 之间的通信，通过传输保存了数据信息的数据对象，达到信息交互的目的。RpcRequest 是用来封装请求数据的数据对象。Rpc-client 在发起请求之前首先需要将请求数据封装到 RpcRequest 对象中，在通过 Netty 发送给 Rpc-server 端。在过程为代码如下：

```
//将请求数据封装到 RpcRequest 数据对象中
RpcRequest reqDataObject = new RpcRequest();
//将对象中的字段依次封装
reqDataObject.setRequestId(randomRequestId);
……
//将封装好的数据对象发送给 Rpc-server
RpcClient client = new RpcClient(host, port);
RpcResponse resDataObject = client.send(reqDataObject);
```

RpcResponse 同样是是数据载体，该对象用于封装 Rpc-server 请求响应数据。Rpc-server 将服务调用后的结果等数据封装到 RpcResponse 对象中，并通过 Netty 发送回 Rpc-client。该过程为代码如下：

```
//当 Rpc-server 接收到客户端请求时，完成服务调用，并封装好 RpcReponse 发送会客户端
Protected  void  channelRead0(ChannelHandlerContext ctx, RpcRequest msg) {
    RpcResponse resDataObject = new RpcResponse();
    resDataObject.setRequestId(msg.getRequestId());
    try {
        resDataObject.setResult(handler(msg)); //handler 为服务调用方法，并返回结果
    } catch (Exception e) {
        resDataObject ect.setError(e);
    }
    //写入 RpcEncoder）进行下一步处理后发送到 channel 中给客户端
    ctx.writeAndFlush(resOb resDataObject ject).addListener(ChannelFutureListener.CLOSE);
}
```

4.2.2 数据对象序列化实现

数据对象序列化的是基于 Protostuff 实现的。使用 Protostuff 对数据对象进行序列化操作，其过程并不复杂。首先根据数据对象的 rpcResquest.getClass()方法获取对象的字节码对象 cls，其次调用 Protostuff 内置的 RuntimeSchema.createFrom(cls)方法获取对象的结构对象 scheme，再通过 LinkedBuffer.allocate()方法创建用于存放二进制流的缓存对象 buffer，最后通过 Protostuff 中的 ProtostuffIOUtil.toByteArray(obj, schema,

buffer)方法完成数据对象的序列化工作。最终得到一个 byte 数组，里面存放数据对象的二进制数据。

反序列化操作需要程序员提供数据对象的二进制数组 data，以及数据对象的字节码对象 cls。首先通过 Protostuff 的 Objenesis 对象的 newInstance(cls)方法获取对应的无状态对象，再通过 RuntimeSchema.createFrom(cls)方法获取对象的结构对象 scheme，最后通过调用 Protostuff 提供的 ProtostuffIOUtil.mergeFrom(data, message, schema)方法完成数据对象的反序列化过程，其最终产物为数据对象。

4.2.3 数据传输功能实现

本框架的数据传输功能是基于 Netty 实现的，因此本节将主要介绍本框架是如何利用 Netty 实现数据的传输。

Netty 对业务的处理离不开 Handler，Handler 是 Netty 中真正用于处理业务的核心组件。Handler 的作用与 Java Web 中的 servlet、filte 等组件非常类似，具备拦截、过滤、处理等作用。数据在进行通信是会被 Handler 所拦截，经 Handler 处理后由 Handler 向下继续传递。Netty 中 Handler 分为两大类，分别是 ChannelInboundHandler、ChannelOutboundHandler。无论是二者中的那一个都实现了 ChannelHandler 接口。Netty 在接收到数据时，先经过 ChannelInboundHandler 处理，处理完成后，再传递给 ChannelOutboundHandler 完成后续处理。ChannelInboundHandler 一般用于解码客户端数据，进行业务处理，而 ChannelOutboundHandler 一般用于报文的编码、发送报文到客户端。

在 Netty 中，用户可以注册多个 ChannelInboundHandler 和 ChannelOutboundHandler，其中多个 ChannelInboundHandler 的执行顺序按其注册顺序进行执行，ChannelOutboundHandler 则在所有的 ChannelInboundHandler 执行完成后，按照它们的注册顺序执行，其过程图 4-1 所示。多个 Handler 是通过 ChannelPipeline 进行组织的。

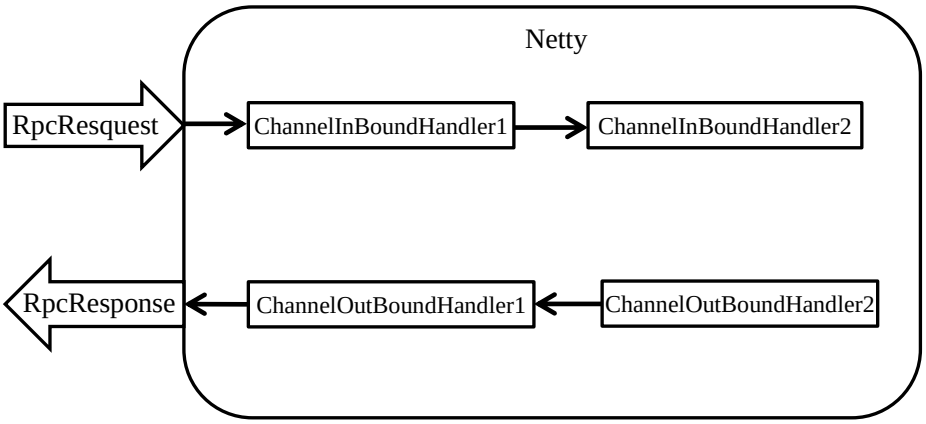

图 4-1 Handler 处理顺序示意图

本框架中客户端在发起请求时，将请求数据封装到 RpcRequest 中，之后将该对象交于 Netty，并由它进行处理、发送。该对象需要通过 ChannelPipeline 中的两个 Handler，一个是 ChannelInBoundHandler，用于监听客户端的响应，另一个 Handler 是 ChannelOutBoundHandler，用于对封装好的 RpcRequest 进行序列化，并发送给客户端。当服务端响应回数据时，响应数据会事先经过用于反序列的 ChannelInBoundHandler，获取响应结果。

服务端接受到客户端发出的请求时，首先经过用于反序列化的 ChannelInBoundHandler，获取 RpcRequest 数据对象，接着经过用于业务处理的另一个 ChannelInBoundHandler，完成业务处理，将得到的结果封装成 RpcResponse 数据对象，再传递给用于序列化的 ChannelOutBoundHandler，完成 RpcResponse 数据对象的编码工作，在通过网络发送给客户端。

4.3 服务发布模块实现

RPC 框架作为一种消息中间件，并不是独立工作的，而是工作在一个系统的不同业务模块中。在 3.5.2 节给出的本框架与上层应用交互图中，可以知道整个应用主要由 User-Service、User-Client、Rpc-server、Rpc-client 构成。

4.3.1 服务对象获取实现

User-Service 作为上层应用服务端需要以 jar 包的方式导入 Rpc-server。Rpc-server 中包含一个名为 RpcServer 的类，该类实现了 ApplicationContextAware 接口，并重写

该接口下的 setApplicationContext()方法。在该方法执行过程为 Spring 创建 RpcServer 类之后由 Spring 进行调用。该方法完成了注解扫描，并将服务对象存放在 RPC 框架持有的 HashMap 中，伪代码如表下：

```
//该方法会随着 Spring 启动而被执行
public void setApplicationContext(ApplicationContext ctx) {
    //获取到添加了@RpcService 注解的类
    Map map = ctx.getBeansWithAnnotation( RpcService.class );
    for (Object value_serviceObject : map.values()) {
            String key_interfaceName = value_serviceObject.getInterfaceName();
            //将获取到的服务对象存放到框架所持有的 handlerMap 中
            handlerMap.put(key_interfaceName, value_serviceObject);
    }
}
```

User-Service 引入 Spring 框架，并通过将 RpcServer 交给 Spring 进行管理，同时在配置文件中通过<context:component-scan/>设置注解扫描包路径。当 User-Service 启动时 Spring 加载配置文件，创建 RpcServer 类并调用 setApplicationContext()，到指定包路径下进行注解扫描，完成服务对象的获取。

4.3.2 服务注册实现

Rpc-server 中的 RpcServer 类实现了 Spring 提供的 InitializingBean 接口，并实现了该接口中的 afterPropertiesSet()方法，该方法在上层应用 User-Service 启动后由 Spring 调用，根据配置中 ZooKeeper 的地址，尝试连接 ZooKeeper，并创建节点。本框架在创建 ZooKeeper 数据节点时，将数据节点封装成 SysInfo 对象。该对象的类图如图 4-2 所示。

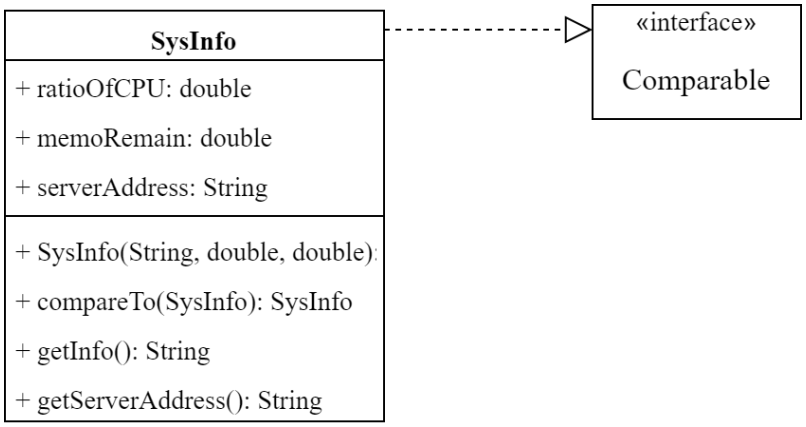

图 4-2 SysInfo 类图

serverAddress 用于存放服务主机 IP 和端口，ratioOfCPU 用于记录主机的 CPU 空闲率，memoRemain 用于记录内存剩余空间。创建数据节点时，直接调用 SysInfo 对象的 getInfo()方法，获取记录上述三组数据的字符串，该字符串用符号进行分割，便于后面提取。该过程伪代码如下：

```
Public    void    registerWithSysInfo(SysInfo sysInfo) {
      //尝试连接 ZooKeeper 的服务端
      ZooKeeper zookeeper = connectZkServer();
      //当 ZooKeeper 服务端连接成功时，创建相应的数据节点
      createNode(zookeeper, sysInfo.getInfo());//ip:port_ratioOfCPU#memoRemain
}
```

4.4 服务调用模块实现

服务调用主要是上层应用 User-Client 通过 Rpc-client 与 Rpc-server 通信，完成对 User-Service 对远程服务调用的过程。即服务调用是基于 RPC 框架的 User-Client 调用 User-Service 的过程。User-Client 调用远程 User-Service 中的方法的过程对于 User-Client 来说是透明的。

4.4.1 服务对象代理实现

User-Client 作为服务请求方，在进行服务接口调用时，并不是直接对 User-Service 上的接口进行调用。而是通过服务接口的代理对象进行服务的调用。本框架中服务对象的代理的具体实现，是通过 JDK 原生的动态代理完成的。服务接口元数据提供了创建动态代理所需要的各项数据，User-Client 在进行接口调用时传递的参数，将被代理对象封装到 RpcRequest 数据对象中。RpcRequest 对象将通过代理对象发出。其中代理对象具体实现伪代码如下。

```
public <T> T create(Class<?> interfaceClass) {
    return (T) Proxy.newProxyInstance( interfaceClass.getClassLoader(),
                                       new Class<?>[]{interfaceClass},
                                       (proxy, method, args) -> {
        //将对象中的字段依次封装
        RpcRequest reqObject = new RpcRequest();
        reqObject.setRequestId(randomRequestId);
        ……
        //通过 Netty 发送请求给服务端
        RpcResponse resObject = RpcClient.send(reqObject);
        //将服务端返回结果作为方法调用结果，并返回
        return resObject.getResult();
    });
}
```

User-Client 中的方法调用只需要根据上述方法创建代理对象，调动代理对象相关方法，便能的到服务响应的结果，其代码如下所示：

```
//创建 RpcProxy，该对象用于创建服务对象的代理对象
RpcProxy rpcProxy = new RpcProxy();
//使用 RpcProxy 创建代理对象
UserService userService = rpcProxy.create(UserService.class);
//调用服务方法，并得到处理结果
return userService.getList();
```

4.4.2 服务调用路由实现

服务调用路由，主要是通过 Rpc-server 保存在 ZooKeeper 中的节点数据来实现的。Rpc-client 与 ZooKeeper 相连，读取指定目录下的数据，将节点数据提取，并分装成 SysInfo 对象，该对象实现了 Comparable 接口，如图 4-2 所示。所有数据节点一一封装成 SysInfo 对象，并放入具有排序功能的容器 TreeSet 中，其中排序的逻辑通过重写 Comparable 接口中的 compareTo()方法实现的，框架中该方法的设计为首先选择 CPU 空闲率大的服务主机发起请求，其次根据内存剩余容量对服务主机进行选择，选择剩余内存容量最大的主机发起请求。当所有 SysInfo 对象存放完毕后，只需要提取 TreeSet 中最后一元素即可。该过程伪代码如下所示：

```
for (String node : nodeList) {
        String path = ZooKeeperConstant.ZK_REGISTRY_PATH + "/" + node;
        byte[] bytes = zk.getData(path, false, null);
        String sysInfoStr = new String(bytes);
        String serverAddress = sysInfoStr.getserverAddress();
        String cpuAndMemo = sysInfoStr.
        double ratioOfCPU = sysInfoStr.getRatioOfCPU();
        double remainMemo = sysInfoStr.getRemainMemo();
        SysInfo sysInfo = new SysInfo(serverAddress, ratioOfCPU, remainMemo); //主机信息封装

        nodeInfoList.add(sysInfo); //将 sysInfo 存入 TreeSet 中，进行排序
}

//获取排序后的第一个 SysInfo 的地址,并向该地址发起请求
serverAddress = nodeInfoList.last().getServerAddress();
```

服务调用选择排序后的最后一个 SysInfo 对象，获取该对象的服务端地址，并向该地址发起请求。

4.5 本章小节

本章主要根据第三章给出的详细设计，对数据传输模块、服务发布模块、服务调用模块中的各个子模块给出了具体的实现。对每个模块的实现都给出了实现说明以及实现过程中的核心伪代码。

5 分布式 RPC 框架的测试

RPC 框架作为常用的消息中间件，在整个系统中处于底层的位置，用于支撑上层应用。本章将 RPC 框架整合到拆分后的系统中，通过对系统的启动，完成对 RPC 框架的运行，通过系统的之间的服务调用来完成对 RPC 框架的联调测试。

5.1 测试依托的系统实验设计

RPC 框架作用于系统内部，为拆分后的系统提供消息通信服务，故 RPC 框架的运行和联调测试需要设计出一个完整且进行拆分后系统，将 RPC 框架整合其中并进行操作。

在本测试所依托的系统设计中，将构造一个简易的图书管理系统，该系统具备增、删、改、查等常见功能。图书馆系统被拆分成二部分，分别为 BookManager-Client，BookManager-Server。其中 BookManager-Client 作为图书管理系统的服务调用方，对客户开放访问，并通过 RPC 框架完成对 BookManager-Server 服务调用，BookManager-Server 为服务提供方。实验设计结构图如图 5-1 所示。BookManager-Server 作为业务处理实际模块将直接对数据库进行操作。

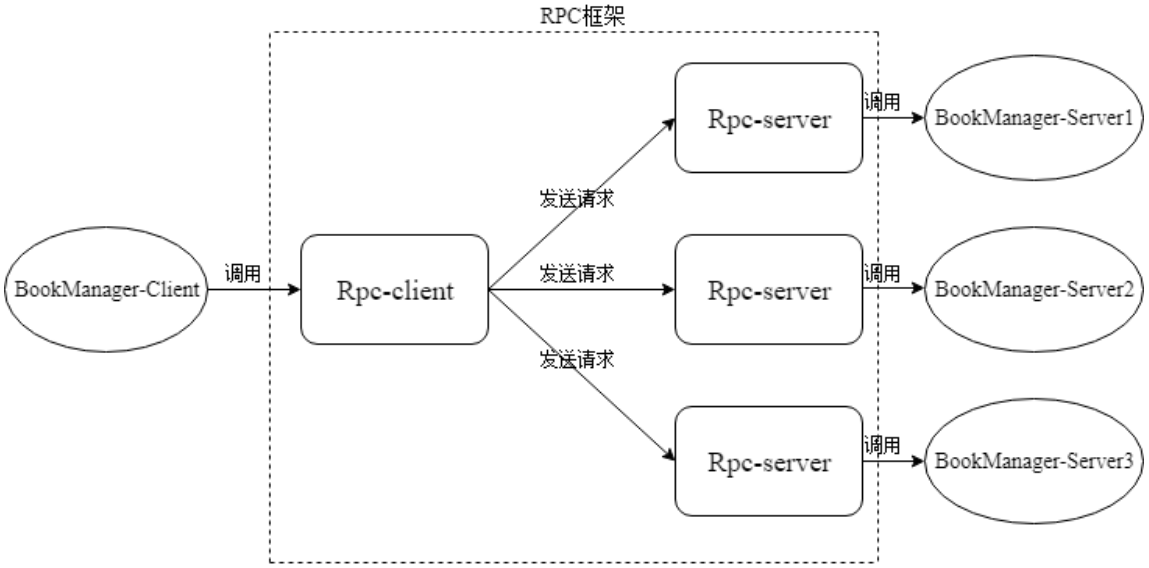

图 5-1 实验设计结构图

5.1.1 BookManager 数据库设计

为了简化系统的本身的复杂性，本系统只提供一张数据表 Book 用于支撑系统的运行。该表结结构如表 5-1 所示：

表 5-1 Book 表结构

字段	类型	备注
ISBN	varchar(13)	主键，用于唯一标识一本图书
bookname	varchar(50)	图书的名字
author	varchar(20)	图书作者
publisher	varchar(50)	图书出版商
price	double	图书价格

为了使实验能够顺利进行，在系统开始运行之前，先将 Book 表进行初始化操作，手动添加 5 条数据，数据内容如表 5-2 所示。

表 5-2 Book 表初始数据

ISBN	bookname	author	publisher	price
978-7-111-21382-1	Book1	Author1	Publisher1	101.00
978-7-111-21382-2	Book2	Author2	Publisher2	102.00
978-7-111-21382-3	Book3	Author3	Publisher3	103.00
978-7-111-21382-4	Book4	Author4	Publisher4	104.00
978-7-111-21382-5	Book5	Author5	Publisher5	105.00

在整个测试过程中，对于需要持久化数据的请求，可以通过查看数据库的数据，判断测试用例是否通过。

5.1.2 BookManager-Client 设计

BookManager-Client 的设计中只对用户提供图书进行的增删改查等简易功能。其中访问链接与功能对应关系表 5-3 所示。这些链接可以通过浏览器直接进行访问。

表 5-3 BookManager-Client 访问链接表

访问链接	备注
http://bookmanager/getById?id=xxx	给定图书的 ISBN，查找对应图书的信息
http://bookmanager/deleteById?id=xxx	给定图书的 ISBN，删除对应图书的信息
http://bookmanager/getList	获取所有图书信息
http://bookmanager/upadte	修改图书信息
http://bookmanager/insert	增加图书信息

BookManager-Client 是整个系统中唯一一个面向用户的模块，用户可以通过发起 HTTP 请求，和整个系统进行交互。BookManager-Client 在接受到用户发出的请求后，会调用 Rpc-client 组件，并由该组件完成后续的服务调用与响应接收。在整个测试的过程中，可以通过 BookManager-Client 的请求响应来判断测试用例是否能够通过。

5.1.3 BookManager-Server 设计

BookManager-Server 所提供的调用服务是根据上一小节中 BookManager-Client 对用户提供的功能进行设计的。其中服务接口与功能对应关系如表 5-4 所示。BookManager-Client 调用服务的请求最终将触发接口方法的调用来进行服务的调用。

表 5-4 服务接口列表

接口方法	备注
getBookByISBN(String ISBN)	给定图书的 ISBN，查找对应图书的信息
deleteBookByISBN	给定图书的 ISBN，删除对应图书的信息
getBookList()	获取所有图书信息
updataBook(Book book)	修改图书信息
insertBook(Book book)	增加图书信息

BookManager-Server 是业务处理的真正模块，也是直接对数据库进行操作的模块。该模块在整个系统中通过与 Rpc-server 直接交互完成和 BookManager-Client 的间接交互。Rpc-server 在接受到来自 Rpc-client 的服务调用请求后，通过调用

BookManager-Server 获取服务调用的结果，并将结果传输给 Rpc-client，继而传给服务调用者 BookManager-Client。

5.2 实验平台搭建

本实验在运行过程中，需要同时启动四个进程，且四个进程需要互不干扰进行功过。四个进程中一个进程为 BookManage-Client 进程，其余三个进程为 BookManage-Server 进程。由于实验可用资源有限，故上述资源均在一台 Windows 操作系统的主机上进行运行。三个 BookManage-Server 进程通过配置不同的服务监听端口，从而同时进行服务。注册中心通过保存各个进程的 IP 与端口，所以 BookManage-Server 进程能够同时对外提供服务，且能被 BookManage-Client 发现和调用。

5.2.1 实验环境

本实验中的所有模块以及注册中心的安装、启动均工作在一台主机上，由于 BookManager 在设计上尽可能的简易，因此在一台主机上，整个系统就能够较好的进行运行。其中实验主机软、硬件信息如表 5-5 所示。

表 5-5 服务器信息

软、硬件	配置
操作系统	Windows 10 专业版 64-bit
服务器处理器	Intel(R) Core(TM) i5-3230M
服务器内存	8.00 GB
服务器机械硬盘	1T SATA3.0 6GS/S 5400 转
服务器固态硬盘	128GB MSATA

5.2.2 ZooKeeper 的安装

基于 ZooKeeper 的注册中心，是整个系统运行必不可少的组件。在构造整个系统的过程，需要安装并启动 ZooKeeper 服务。下面将对 ZooKeeper 的安装关键步骤进行

说明。由于 ZooKeeper 的运行需要依赖 JDK，故在安装 ZooKeeper 之前需要完成 JDK 的安装。本实验环境将使用 3.4.5 版本的 ZooKeeper。ZooKeeper 的安装步骤如下：

1）前往 ZooKeeper 官网，下载 ZooKeeper 压缩包。其中下载链接为：http://archive.apache.org/dist/zookeeper/zookeeper-3.4.5/zookeeper-3.4.5.tar.gz

2）解压 zookeeper-3.4.5.tar.gz 到指定目录下。为了便于后续描述，将 ZooKeeper 安装根目录用 ZOOKEEPER_HOME 代替

3）在 ZOOKEEPER_HOME 目录下创建名为 data 的文件夹，用于 ZooKeeper 数据存放

4）将 ZOOKEEPER_HOME 目录下的 conf 目录中的 zoo_sample.cfg 重命名为 zoo.cfg，并修改 zoo.cfg 文件中 dataDir 的值为步骤 3 中创建的 data 文件夹路径

5）进入 ZOOKEEPER_HOME 目录下的 bin 文件夹，双击 zkServer.cmd，完成服务的启动

5.3 测试过程

测试是将 RPC 框架导入系统，并通过用户和 BookManager-Client 直接进行交互进行的。用户通过浏览器发起 HTTP 请求，对整个系统进行联调测试。并根据数据持久化情况以及浏览器接收到的响应数据来判断测试是否通过。

5.3.1 项目的整合

为了是整个系统便于管理，在项目开发过程中统一使用 Maven 作为项目构建工具。Maven 具有一键构建及依赖管理等特点。在使用该工具时，能够方便的将公共应用创建成应用模块，并能生成 jar 供其他程序进行调用。

在本文所设计的 RPC 框架的实现中，该框架被命名为 MyRpc，被拆分成 5 个工程模块。它们分别是 MyRpc-client，MyRpc-client-register，MyRpc-server，MyRpc-server-register，以及 MyRpc-common。该框架的工程依赖图如图 5-2 所示。下面将介绍这些工程模块主要功能：

1）MyRpc-client 是本框架的客户端，接收服务调用方的调用请求

2）MyRpc-client-register 是 MyRpc-client 用于连接 ZooKeeper 的工程代码

3）MyRpc-server 是本框架的服务端，用户完成服务调用

4）MyRpc-server-register 是 MyRpc-server 用于连接 ZooKeeper 并完成注册的工程代码

5）MyRpc-common 中包含本框架中公共的功能性代码

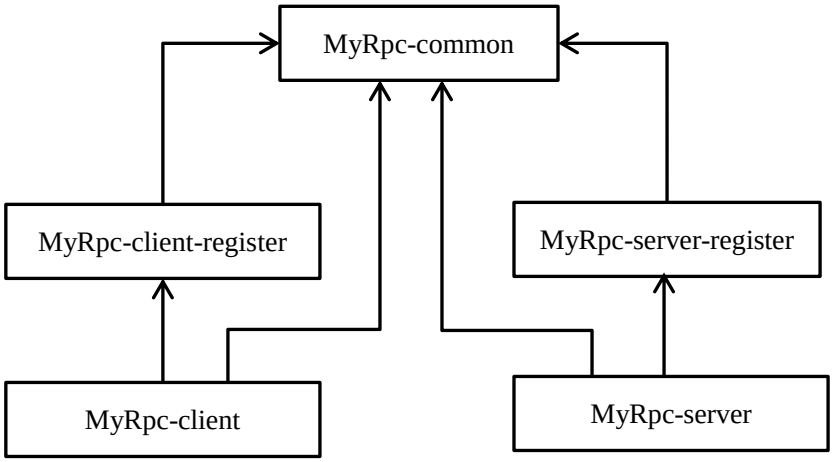

图 5-2 MyRpc 框架工程依赖图

对于整个系统来说，BookManager-Client 将导入 MyRpc-client 的工程依赖，BookManager-Server 将导入 MyRpc-server 的工程依赖。整个系统的工程结构如图 5-3 所示。

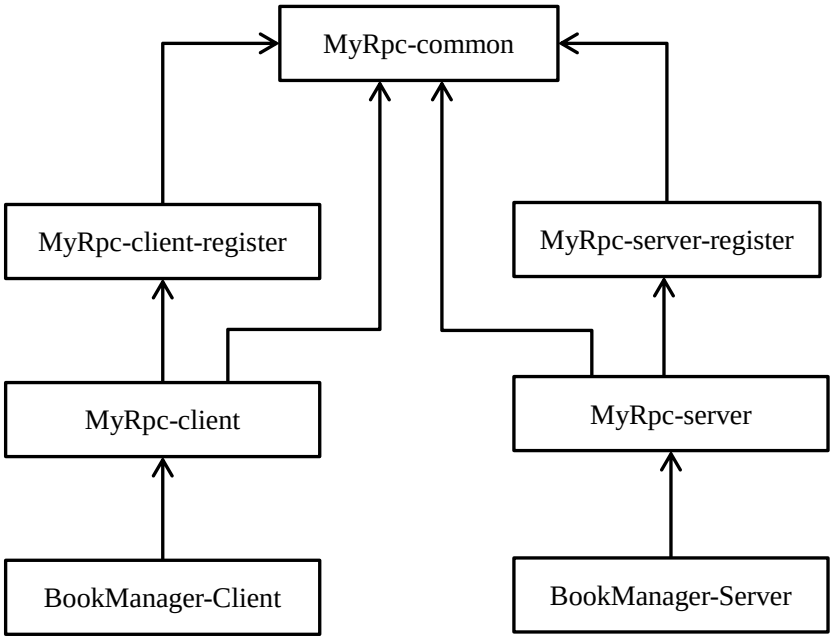

图 5-3 系统工程结构图

BookManager-Client 与 BookManager-Server 对 RPC 框架的启动均是通过 Spring 的配置文件 applicationContext.xml 的配置<bean>标签来完成的。BookManager-Client 与 BookManager-Server 在启动时，Spring 读取 applicationContext.xml 配置文件<bean>标签，初始化 bean 对象，完成 RPC 框架的启动。其中 BookManager-Client 中的配置文件部分配置如表 5-6 所示，BookManager-Server 中的配置文件部分配置如表 5-7 所示。

表 5-6 BookManager-Client 部分配置

```
<context:property-placeholder location="classpath:rpc.properties"/>
<bean id="serviceDiscovery" class="com.hussir.rpc.register.ServiceDiscovery">
    <constructor-arg name="registryAddress" value="${zookeeper.address}"/>
</bean>
<bean id="rpcProxy" class="com.hussir.rpc.proxy.RpcProxy">
    <constructor-arg name="serviceDiscovery" ref="serviceDiscovery"/>
</bean>
```

表 5-7 BookManager-Server 部分配置

```
<context:component-scan base-package="com.hussir.bookstore.service.impl"/>
<context:property-placeholder location="classpath:rpc.properties"/>
<bean id="serviceRegister" class="com.hussir.rpc.register.ServiceRegister" >
    <constructor-arg name="registerAddress" value="${zookeeper.address}"/>
</bean>
<bean id="rpcServer" class="com.hussir.rpc.server.RpcServer" >
    <constructor-arg name="serviceRegister" ref="serviceRegister" />
    <constructor-arg name="serverAddress" value="${service.address}" />
</bean>
```

配置文件中的<context:property-placeholder location="classpath:rpc.properties"/>作为是加载名为 rpc.properties 的配置文件，并能通过“${}”获取配置文件中的值。在该文件中配有注册中心的 ZooKeeper 的 IP 地址及端口信息。BookManager-Server 的配置文件中有<context:component-scan base-package="com.hussir.bookstore.service.impl"/>配置项。该配置项功能为注解扫描，Spring 会扫描“base-package”目录下的源文件，并获取添加了@RpcService 注解的对象，存放在 RPC 框架持有的 HashMap 中，便于后续调用。

5.3.2 服务调用测试

BookManager 系统在全部整合完成后，首先启动 BookManager-Server 服务端。BookManager-Server 在成功启动后，会在注册中心 ZooKeeper 中生成一个数据节点。因此可以通过 ZooKeeper 的客户端 API 进行节点查看，并判断服务是否成功。启动成功后，通过 ls 指令能够发现服务节点。本系统启动后，其结果如图 5-4 所示。

```
[zk: localhost:2181(CONNECTED) 9] ls /registry
[data0000000030]
```

图 5-4 服务注册信息查询结果

BookManager-Server 启动成功后，BookManager-Client 便能够启动，启动之后通过浏览器进行访问。下面将对 BookManager-Client 通过的访问链接分别进行访问测试。

1）测试 1：获取所有图书信息

通过浏览器访问 http://bookmanager/getList，浏览器返回数据库中所有图书信息，数据如表 5-8 所示。该测试用例通过。

表 5-8 图书信息列表（总）

ISBN	BookName	Author	Publisher	Price(元)
978-7-111-21382-1	Book1	Author1	Publisher1	101.00
978-7-111-21382-2	Book2	Author2	Publisher2	102.00
978-7-111-21382-3	Book3	Author3	Publisher3	103.00
978-7-111-21382-4	Book4	Author4	Publisher4	104.00
978-7-111-21382-5	Book5	Author5	Publisher5	105.00

2）测试 2：给定图书的 ISBN，查找对应的图书信息

通过浏览器访问 http://bookmanager/getById?id=978-7-111-21382-1，浏览器返回该 id 对应的图书信息，如表 5-9 所示。故该测试用例通过。

表 5-9 根据 ID 查询后的图书信息

ISBN	BookName	Author	Publisher	Price(元)
978-7-111-21382-1	Book1	Author1	Publisher1	101.00

3）测试 3：修改图书信息

通过浏览器对 http://bookmanager/update 链接发起 POST 请求，且请求体中封装了待更新的 Book 数据。请求发起后，浏览器接收到更新成功的提示，通过直接对数据库的访问，发现该 ISBN 对应的图书信息已被更新，更新后的数据如表 5-10 所示。该测试用例通过。

表 5-10 更新后的图书列表

ISBN	BookName	Author	Publisher	Price(元)
978-7-111-21382-1	Book1	Author1	Publisher1	101.00
978-7-111-21382-2	Book2	Author2	Publisher2	102.00
978-7-111-21382-3	Book3	Author3	Publisher3	103.00
978-7-111-21382-4	Book4	Author4	Publisher4	104.00
978-7-111-21382-5	Book6	Author6	Publisher6	106.00

4）测试 4：给定图书的 ISBN，删除对应图书的信息。

通过浏览器访问 http://bookmanager/deleteById?id=978-7-111-21382-5，浏览器返回删除成功的提示，通过直接对数据库的访问，发现该条数据已经被删除，删除该条数据后的图书列表如表 5-11 所示。该测试用例通过。

表 5-11 删除对应记录后的图书列表

ISBN	BookName	Author	Publisher	Price(元)
978-7-111-21382-1	Book1	Author1	Publisher1	101.00
978-7-111-21382-2	Book2	Author2	Publisher2	102.00
978-7-111-21382-3	Book3	Author3	Publisher3	103.00
978-7-111-21382-4	Book4	Author4	Publisher4	104.00

5）测试 5：增加图书信息

通过浏览器对 http://bookmanager/insert 链接发起 POST 请求，且请求体中封装了待插入的 Book 数据。请求发起后，浏览器接收到插入成功的提示，通过直接对数据库的访问，发现新的 Book 数据已被插入，插入该条数据后的数据如表 5-12 所示。该测试用例通过。

表 5-12 插入新数据后的图书列表

ISBN	BookName	Author	Publisher	Price(元)
978-7-111-21382-1	Book1	Author1	Publisher1	101.00
978-7-111-21382-2	Book2	Author2	Publisher2	102.00
978-7-111-21382-3	Book3	Author3	Publisher3	103.00
978-7-111-21382-4	Book4	Author4	Publisher4	104.00
978-7-111-21382-6	Book6	Author6	Publisher6	106.00

5.3.3 性能测试

在大型系统中使用 RPC 框架作为消息中间件，是以轻微牺牲系统性能为代价，换取系统高度解耦的一种方式。因为系统之间的业务调用会因为使用了中间件，导致业务调用的中间过程增多。因此性能测试主要是验证系统在使用本框架作为消息中间件后，其服务请求的响应时间的变化。本节测试将借助 Apache 的开源测试工具 JMeter 对本文所研究的 RPC 框架进行性能测试。

1）性能测试 1：与未使用 RPC 框架的 BookManager 系统进行响应时间对比

通过 JMeter 对使用了本 RPC 框架的 BookManager 系统与未使用本 RPC 框架的 BookManager 系统的服务请求响应时间进行对比。并记录记录下每个链接访问对应的响应时间。其结果如表 5-8 所示。

表 5-8 与未使用本框架的系统响应时间测试结果表

访问链接	使用本 RPC 框架（请求响应时间）	未使用本 RPC 框架（请求响应时间）
http://bookmanager/getById?id=xxx	15ms	12ms
http://bookmanager/deleteById?id=xxx	74ms	64ms
http://bookmanager/getList	13ms	10ms
http://bookmanager/upadte	39ms	21ms
http://bookmanager/insert	95ms	77ms

根据上述测试结果表，可以得到系统响应时间对比折线图，如图 5-13 所示。

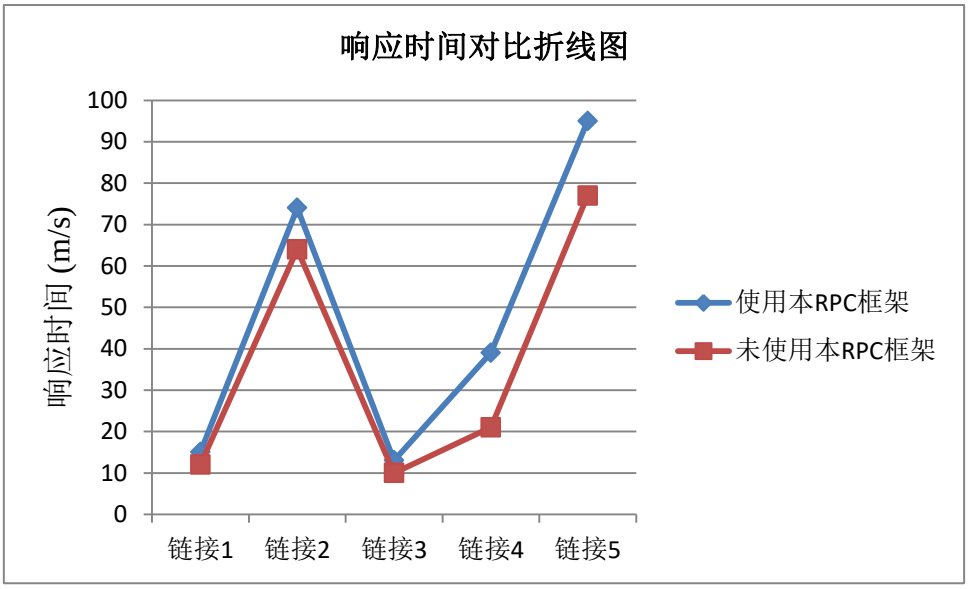

图 5-13 与未使用本框架的系统响应时间对比折线图

通过 JMeter 测试出的数据可知，BookManager 系统在使用了本文所研究的 RPC 框架后，响应时间有一定的增加。其原因是网络带宽以及服务请求中间夹杂了许多额外过程。从实验结果显示的响应时间对比图来看，使用了 RPC 框架之后系统的响应时间只有小幅度的提升，因此使用了 RPC 框架后，系统能够满足用户的需求的。所以本文所实现的 RPC 框架具备不错的性能，能够满足用户的使用要求。

2）性能测试 2：与使用了 Web Service 的 BookManager 系统进行响应时间对比

通过 JMeter 对使用了本 RPC 框架的 BookManager 系统与使用了 Web Service 的 BookManager 系统的服务请求响应时间进行对比。并记录下每个链接访问对应的响应时间。其结果如表 5-9 所示。

表 5-9 与使用 Web Service 的系统响应时间测试结果表

访问链接	使用本 RPC 框架（请求响应时间）	使用 Web Service（请求响应时间）
http://bookmanager/getById?id=xxx	15ms	37ms
http://bookmanager/deleteById?id=xxx	74ms	99ms
http://bookmanager/getList	13ms	32ms

表 5-9（续表）

访问链接	使用本 RPC 框架（请求响应时间）	使用 Web Service（请求响应时间）
http://bookmanager/upadte	39ms	61ms
http://bookmanager/insert	95ms	103ms

根据上述测试结果表，可以得到系统响应时间对比折线图，如图 5-14 所示。

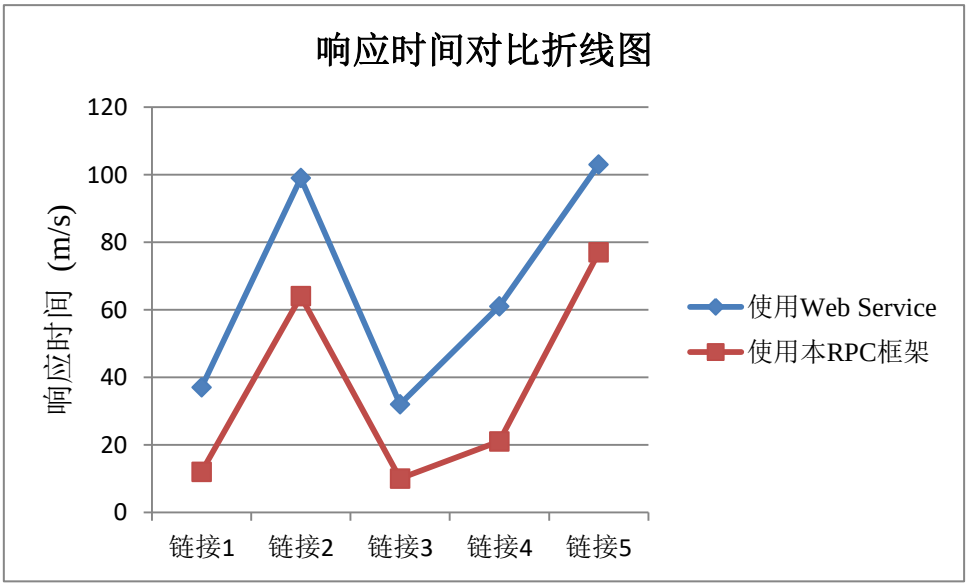

图 5-14 与使用 Web Service 的系统响应时间对比折线图

通过 JMeter 测试的出的数据可知，BookManager 系统在使用了本文所研究的 RPC 框架后，其服务请求时间较使用了中间件 Web Service 的 BookManager 系统有着较大的提升。其原因在于 Web Service 则是基于 HTTP 协议进行通信的，这将导致通信数据包含部分无效数据。而本文所研究的 RPC 框架基于 TCP 协议进行通信的，通过自定义数据通信格式，减少无效字段的出现，从而减少数据传输，加快服务请求响应。

5.4 本章小结

本章主要目的是对 RPC 框架进行功能以及性能的测试。由于 RPC 框架作为消息中间件，需要工作在系统的内部，因此本文为了达到测试 RPC 框架的目的，特意设计了一套极为简易的 BookManager 系统。通过对 BooKManager 系统的启动，完成了 RPC 框架的启动工作。并通过对 BookManager 提供的访问链接，对 RPC 框架的运行

进行了测试。在本章内容的最后，对 RPC 框架进行了性能测试，通过对比请求响应时间来完成性能测试。

6 总结与展望

本篇论文论文在前面的章节里对本文所研究的分布式 RPC 框架研究内容进行了详细阐述。本章将对本文的研究内容进行全面的总结，还将针对研究中的不足提出下一阶段的期望。

6.1 论文工作总结

系统业务的完善与扩展，给系统本身性能带来不好影响的同时也大大增加了后期维护与版本迭代的成本。RPC 框架的出现，从业务拆分的角度对这种情况进行了很好的改善。使用 RPC 框架作为消息中间件是目前中大型项目组织项目最常用的业务解耦方式。因此，笔者最终通过将对 RPC 框架研究作为毕业论文的研究方向。本文主要工作内容如下：

1）介绍 RPC 框架的研究背景，以及国内外在 RPC 框架领域的研究现状。

2）介绍了 RPC 框架以及在实现 RPC 框架时所用到的相关技术。对相关技术的实现原理进行了阐述

3）从功能的角度对 RPC 框架进行模块分解，并对 RPC 框架进行业务需求分析，以及各个模块进行了功能性需求分析

4）针对需求分析得到的需求，对 RPC 框架进行了功能介绍，并在概要设计的基础上给出了框架详细设计

5）结合 RPC 框架的需求分析文档，以及目前的流行技术对 RPC 框架进行了实现

6）为 RPC 框架搭建一个简易的系统，通过对系统的启动和测试，完成 RPC 框架的启动与测试

在完成RPC框架的必要功能的前提下，尽可能的提升RPC框架的使用时的性能、可靠性即易用性。

6.2 展望

本框架的功能已经能够满足本文需求分析中的所有需求，包括业务需求、功能性需求以及非功能性需求。经过测试后，发现该框架作为消息中间件，能够正常且较好的运行在系统内部。通过进一步的分析，我们发现框架有许多不足之处，下一阶段需要改进的地方主要有以下几点：

1）该框架目前只能为 Java 语言编写的系统提供服务，具有很强的平台限制性，下一步加提升该框架跨语言的能力，争取使该框架能够跨语言进行使用。

2）该框架的服务路由策略是根据服务提供方在 ZooKeeper 上注册时硬件占用率信息进行路由的，该信息的实时性不高，下一步将定时刷新服务提供方硬件占用率信息，改善服务路由策略，提升负载均衡的效果。

致谢

在经历了数月的学习和沉淀，论文的书写已经接近尾声了。值此之际，我想对华中科技大学，以及所有帮助过我的人道一声谢。本篇论文的完成，离不开在华中科技大学校内的学习，借此机会我由衷的感谢华中科技大学为在校的师生提供了良好的学习环境，以及德才兼备的任课老师，为我们解疑答惑。

其次，我也特别感谢我的研究生导师、本论文的指导老师陈长清老师。研究生的生涯中，陈老师严谨近乎苛刻的治学态度一只影响着我。在论文的编写期间，陈老师经常和我讨论交换各自意见，并在我遇到瓶颈是为我提供关键性意见，指引着我写论文的思路。在论文修改阶段，陈老师总是及时的帮我指出论文的不足之处，并附上详细的批注，让我修改。正是陈老师的认真负责，才使得本篇论文能够顺利完成。

此外，我还要感谢实验室的同学们，他们是我研究生期间生活上的陪伴者，学习上的互助者。无论是在平时的学习中，还是本论文的编写过程中，当我遇到技术上的难题时，他们都会不留余力的给与我帮助。

最后，我要感谢本文所引用的文献的作者们，通过他们优秀的论文，让我更快的了解了 RPC 框架的设计理念与原理。正是因为他们的论文，让我站在巨人的肩膀上，更快且更高质量的完成了本片论文的编写。

参考文献

[1] Yuan M L, Huang Y B, Huang J L, et al. The Research and Application of MVC Software Architecture Based on J2EE. Application Research of Computers, 2003

[2] 王霜,修保新,肖卫东.Web 服务器集群的负载均衡算法研究.计算机工程与应用,2004(25):78-80+99

[3] 秦方钰,刘冬梅,徐栋.一种面向 SOA 架构的数据业务总线应用研究.电子技术与软件工程,2015(09):203-204

[4] 张艳军,王剑,叶晓平,李培远.基于 Netty 框架的高性能 RPC 通信系统的设计与实现.工业控制计算机,2016,29(05):11-12+15

[5] Andrew D. Birrell,Bruce Jay Nelson. Implementing remote procedure calls. ACM Transactions on Computer Systems (TOCS),1984,2(1)

[6] Pallavi Singh,H.K. Dixit,D.K. Tripathi,Rekha Mehra. Design and analysis of all-optical inverter using SOA-based Mach–Zehnder interferometer. Optik - International Journal for Light and Electron Optics,2013,124(14)

[7] 李俭兵,何登平.SOAP 技术及其应用.计算机科学,2003(04):111-112+128

[8] 马凯. 基于 Hessian 协议的新生注册系统设计与实现.华南理工大学,2012

[9] 周康,李觊,董科军,南凯.一种基于 Thrift 的日志收集分析系统.科研信息化技术与应用,2015,6(02):19-27

[10] 杨帆,孔维萍,蒋晓肖,肖永利,虢华.基于 Thrift 的 RPC 中间件在航天信息系统中的设计与实现.计算机测量与控制,2017,25(12):279-282+306

[11] 周鼎. 基于 SOA 和云计算架构的企业协同创新服务平台研究与应用.北京工业大学,2017

[12] 宋小倩.浅析分布式服务框架 dubbo.计算机产品与流通,2018(03):43

[13] 李磊,李娟.Dubbo 服务框架技术在学习系统开发中的应用与实践.计算机系统应用,2017,26(06):244-248

[14] Gaea.[EB/OL]. https://github.com/58code/Gaea/blob/master/Gaea.pdf

[15] brpc.[EB/OL]. https://github.com/brpc/brpc/tree/master/docs

[16] Walls C. Spring in Action// Spring in action. Manning Publications Co. 2007

[17] 贾鸣华. 基于 JavaEE 的电子类资产管理系统的设计与实现[D].南京大学,2018.

[18] 周岚.基于 Spring 框架的 IOC 模式的设计和实现.合肥学院学报(自然科学版),2011,21(01):49-53.

[19] Shams Z, Edwards S H. Reflection Support: Java Reflection Made Easy. Open Software Engineering Journal, 2014, 7(1):38-52

[20] 刘双.Spring 框架中 IOC 的实现.电子技术与软件工程,2018(21):231.

[21] 翟剑锟. Spring 框架技术分析及应用研究.中国科学院大学(工程管理与信息技术学院),2013

[22] P.N.L. Pavani,B.K. Prafulla,R. Pola Rao,S. Srikiran. Design, Modeling and Structural Analysis of Wave Springs. Procedia Materials Science,2014,6

[23] 程龙,李冶.应用 ASM 修改 JAVA 字节码.信息与电脑(理论版),2011(05):124+126

[24] 柳汝滕. 一种轻量级分布式 RPC 框架的研究与实现.东南大学,2017

[25] Guillermo L. Taboada,Juan Touriño,Ramón Doallo. Java Fast Sockets: Enabling high-speed Java communications on high performance clusters. Computer Communications,2008,31(17)

[26] 李昊,刘志镜.线程池技术的研究.现代电子技术,2004(03):77-80

[27] 张力,王纯,阮稳.Java 高性能通信统一框架的设计.电信科学,2009,25(07):68-71

[28] Maurer N, Wolfthal M A. Netty in Action. Manning Publications Co. 2015

[29] 龚鹏,曾兴斌.基于 Netty 框架的数据通讯服务系统的设计.无线通信技术,2016,25(01):46-49

[30] 夏斐. 基于 Netty 的消息中间件的研究与实现.电子科技大学,2018

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

[32] 于天,黄昶.一种高性能异步 RPC 框架的设计与实现.信息通信,2018(03):127-129

[33] 庄鹏. 基于 ZooKeeper 的分布式服务中间件设计与实现.深圳大学,2016

[34] Junqueira F P, Reed B C. The life and times of a zookeeper// ACM Symposium on Principles of Distributed Computing. ACM, 2009:4-4

[35] 黄毅斐. 基于 ZooKeeper 的分布式同步框架设计与实现.浙江大学,2012

[36] 谭玉靖. 基于 ZooKeeper 的分布式处理框架的研究与实现.北京邮电大学,2014

[37] Junqueira F, Reed B. ZooKeeper: Distributed Process Coordination. O'Reilly Media, Inc. 2013

[38] 胡雪婧. 基于 ZooKeeper 的分布式系统的消息发送机制的设计与实现.吉林大学,2016

[39] 陈文武. 分布式锁技术研究.华南理工大学,2013.

[40] 刘芬,王芳,田昊.基于 Zookeeper 的分布式锁服务及性能优化.计算机研究与发展,2014,51(S1):229-234

[41] Hunt P, Konar M, Junqueira F P, et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems. Usenix Annual Technical Conference, 2010

[42] 周绍景,应杰,潘宏斌,黄建,杨正元.RESTful 架构的应用研究.数字技术与应用,2018,36(05):59-60

[43] 冯新扬,沈建京.REST 和 RPC:两种 Web 服务架构风格比较分析.小型微型计算机系统,2010,31(07):1393-1395

[44] Ahern A, Yoshida N. Formalising Java RMI with explicit code mobility// Acm Sigplan Conference on Object-oriented Programming. ACM, 2005:403-422

[45] 解志君.代理模式在 Java 中的应用.软件,2014,35(05):94-96+101