

# 摘要

在高速发展的互联网时代，网站的业务发展日益迅速，访问流量急剧增大，大多数网站使用分布式系统架构来处理庞大的业务请求，通过远程过程调用的方式将请求分配给系统中的服务器进行处理。对于远程调用的方式来说，请求消息通过网络到达服务器，服务器需要处理大量的 I/O 并发请求，这对服务器的通信能力是一个巨大的挑战。Netty 异步通信框架的产生，可以有效的提高服务器的通信性能。相对于传统的 NIO（New IO）框架，Netty 在缓冲区、通道以及线程池模型上进行了优化。通过 I/O 线程的合理调度，处理高并发 I/O 请求。另外 Netty 为用户提供了灵活的定制机制，将用户自定义的处理器添加到 Netty 的处理链中，对业务消息进行拦截处理。本文将 Netty 作为服务器的底层通信框架，设计了一个高性能的 RPC（Remote Procedure Call）服务器。

本文首先分析了 RPC 远程过程调用的原理及参数传递机制，并对 IO 通信模型的发展过程以及负载均衡算法进行研究。根据 RPC 服务器的功能和性能需求，设计并搭建了系统框架。然后对 Netty 的 IO 模型、线程模型以及 Netty 的消息处理过程进行研究。从服务器调度处理过程的角度出发，分别实现了服务器的通信调度模块、编解码模块、服务发布和订阅、业务调度处理模块以及长连接服务等功能，通过各个模块之间的调用处理，实现一个高性能服务器的功能，并对服务器进行初步优化。最后通过实验，对服务器的各个模块的性能进行测试，发现服务器有着良好的并发处理性能。

**关键词：**Netty，服务器，RPC，远程调用，负载均衡，IO

## Abstract

In the high-speed Internet era, the business of the website is developing rapidly and the traffic is increasing rapidly. The distributed system architecture is used to deal with the huge business requests by most Web sites, and assign the requests to the servers in the system by the way of remote procedure call. For the remote call mode, the request message needs to deal with a large number of I/ O concurrent requests through the network, which is a great challenge to the servers communication capability. In contrast to the traditional NIO (New IO) framework, Netty has been optimized on buffers, channels, and thread pool models. High concurrency I/ O requests are processed by rational scheduling of I/ O threads. In addition, Netty provides a flexible custom mechanism for users to add user-defined processors to Netty's processing chain to intercept and process business messages. In this paper, Netty is used as the underlying communication framework of the server, and a high performance RPC (Remote Procedure Call) server is designed.

The principle and parameter transfer mechanism of RPC remote procedure call is analyzed in this paper firstly, and the development process of IO communication model and load balancing algorithm is studied. According to the function and performance requirements of the RPC server, the system framework is designed and constructed. Then the IO model of Netty, the threading model and the message handling of Netty are studied. From the point of view of server scheduling process, the communication scheduling module, codec module, service publishing and subscription module, business scheduling module and the function of long connection service are implemented respectively. The function of a high-performance server is realized by calling processing among modules, and the server is optimized preliminarily. Finally, through the experiment, the performance of each module of the server is tested, and found that the server has good concurrency processing performance.

**Keywords:** Netty, server, RPC, remote call, load balancing, IO

# 目录

第一章 绪论 .....	1
1.1 选题背景及研究意义 .....	1
1.2 国内外研究现状 .....	2
1.3 本文主要工作 .....	2
第二章 相关理论与技术研究 .....	4
2.1 RPC 服务器简介 .....	4
2.2 RPC 远程过程调用 .....	4
2.2.1 RPC 远程过程调用原理 .....	4
2.2.2 参数传递机制 .....	5
2.3 负载均衡 .....	6
2.3.1 负载均衡原理 .....	6
2.3.2 轮询法 .....	6
2.3.3 最少连接数法 .....	7
2.4 网络 I/O 模型演进 .....	9
2.4.1 传统的 BIO 模型 .....	9
2.4.2 伪异步 I/O 模型 .....	10
2.4.3 NIO 模型 .....	10
2.5 本章小结 .....	12
第三章 RPC 服务器的需求分析与架构设计 .....	13
3.1 服务器的需求分析 .....	13
3.1.1 服务器的功能需求分析 .....	13
3.1.2 服务器的性能需求分析 .....	13
3.2 服务器总体架构设计与实现 .....	14
3.2.1 服务器总体架构 .....	14
3.2.2 RPC 服务器远程调用流程 .....	15
3.2.3 开发环境搭建 .....	16
3.3 Netty 框架技术 .....	17
3.3.1 Netty NIO 模型 .....	17
3.3.2 Netty 的线程模型 .....	19
3.3.3 Netty 的 ChannelPipeline 处理链 .....	20
3.4 本章小结 .....	21
第四章 基于 Netty 的 RPC 服务器的实现 .....	22
4.1 消息定义 .....	22
4.2 编解码模块 .....	22
4.2.1 TCP 半包解码 .....	23
4.2.2 TCP 消息长度编码 .....	26
4.2.3 消息编解码 .....	27
4.3 通信调度模块 .....	29
4.3.1 I/O 线程模型设计 .....	29
4.3.2 服务端 I/O 处理 .....	30
4.3.3 客户端 I/O 处理 .....	34
4.3.4 时间轮和定时任务处理 .....	36
4.4 服务发布和订阅 .....	38
4.4.1 服务发布过程 .....	38

4.4.2 服务订阅过程 .....	39
4.5 业务处理模块 .....	40
4.5.1 线程池设计 .....	40
4.5.2 RPC 消息处理过程 .....	42
4.6 服务端调用处理 .....	43
4.7 客户端调用处理 .....	47
4.7.1 服务订阅 .....	47
4.7.2 远程响应消息处理 .....	49
4.8 长连接服务 .....	50
4.8.1 长连接原理 .....	50
4.8.2 心跳检测机制 .....	51
4.9 本章小结 .....	53
第五章 基于 Netty 的 RPC 服务器的性能测试与优化 .....	54
5.1 RPC 服务器长连接性能测试 .....	54
5.1.1 心跳检测和重连机制测试 .....	54
5.1.2 长连接性能测试 .....	58
5.2 RPC 服务器消息序列化性能测试 .....	59
5.2.1 RPC 服务器序列化性能对比 .....	59
5.3 RPC 服务器并发性能测试 .....	61
5.4 RPC 服务器的优化 .....	63
5.5 本章小结 .....	65
第六章 总结与展望 .....	66
6.1 总结 .....	66
6.2 展望 .....	67
参考文献 .....	68
附录 1 攻读硕士学位期间撰写的论文 .....	70
致谢 .....	71

# 第一章 绪论

## 1.1 选题背景及研究意义

随着互联网技术的高速发展<sup>[1]</sup>，人们的生活和互联网越来越密不可分。互联网技术极大地改变了人们的生活方式，让人们生活变得更加方便。社交平台让人们可以随时与亲人朋友交流；新闻平台使得人们尽知天下事；电商平台带给人们足不出户的购物体验；外卖平台让人们可以随时尝到各种美食；打车平台让人们可以享受随时随地出行的愉悦。互联网给人们带来了极大的便利，使人们可以享受更加舒适的生活。

现在，互联网技术飞速发展，各种网站平台层出不穷，网络的使用量逐年递增。中国互联网络信息中心（CNNIC）在 2018 年 1 月 31 日，发布的第 41 次《中国互联网络发展状况统计报告》显示，截至 2017 年 12 月，中国上网人数已经高达 7.72 亿，已经占到全国人口的 55.8%，甚至已经超过了世界的平均水平（51.7%）。北京《新京报》在 2018 年 1 月 30 日的报告显示，截至 2017 年 12 月，中国利用网络平台进行支付的人数相较于 2016 年增加了 11.9%，达到了 5.31 亿；外卖用户量较 2016 年提高了 64.9%，达到了 3.43 亿；网络直播用户较 2016 年新增了 1.5 亿，达到了 4.22 亿。在互联网用户量增加如此迅速的情况下，对于网站系统中服务器的处理响应能力是一个极大的考验，系统响应速度的快慢直接关系到用户体验，提高系统的处理性能是当前大多数互联网公司亟待解决的问题。

针对以上问题，目前大部分互联网公司采用的解决方案是使用分布式系统架构来处理大量并发请求，将系统的服务部署在不同的机器上，并采用集群的方式对外提供服务，服务之间通过 RPC（远程过程调用）的方式进行调用，以此来降低服务器的负载压力。RPC 由 Nelson 和 Briell 于 1984 年提出，它是一种远程调用协议，通过网络从远程服务器请求服务，这一过程对程序员来说是“透明的”，就像调用本地方法一样。RPC 服务器作为分布式系统中的节点，不仅可以为远程服务器提供服务，而且也可以向远程服务器请求服务，通过注册中心统一对服务进行管理。由于系统分布式部署，系统内部存在多个节点服务器之间通信的问题<sup>[2]</sup>，RPC 服务器的性能的好坏直接关系到系统的响应性能，进而影响用户的体验，因此开发一个高性能的 RPC 服务器具有非常重要的意义。

## 1.2 国内外研究现状

随着对服务器的性能要求越来越高,基于传统 NIO 通信框架的服务器已经无法满足系统性能的要求<sup>[3]</sup>。对于服务器来说,I/O 处理过程是比较耗时的,I/O 性能的好坏直接影响服务器的通信质量。Netty 作为一个高性能、异步非阻塞的通信框架,可以基于它对通信框架进行灵活地扩展,来提高通信的效率。Netty 是由 Trustin Lee 开发的一个基于 NIO 的异步非阻塞通信框架,它的框架特点是:异步非阻塞、高性能、高可定制性和高可靠性,利用它的高性能以及高可定制性可以开发各种服务端和客户端程序。

目前,Netty 框架在国内外已经取得了飞速的发展。在国内,随着网站系统日渐庞大,用户数量逐渐增多,大量的并发访问对系统的压力越来越大。通过采用分布式系统架构的方式,对系统进行服务化拆分,服务之间采用 RPC 远程调用的方式进行调用,以此来降低系统的压力。Netty 作为高性能的异步通信框架,常常被用来作为系统服务之间的通信组件,以此来提高系统的通信性能<sup>[4]</sup>。例如:阿里开源的分布式服务框架 Dubbo,其内部采用 Netty 作为基础通信组件,用于各节点服务器的通信;淘宝的 RocketMQ 消息中间件,利用 Netty 的高性能以及异步通信特征,作为消息生产者和消费者之间的通信框架,以提高消息队列的处理速度。

高性能的异步通信发展带动了异步通信框架的研究,国外很早就已经运用了 Netty 异步通信技术。经典的 Hadoop 的高性能通信和序列化组件 Avro,是 Apache Software Foundation (阿帕奇软件基金会)公司研发的一个独立于编程语言的数据序列化系统,它运用了 RPC 框架,使用 Netty 进行远程节点之间的通信,并利用 Netty 对它的 Netty Service 进行再次封装,提升其通信能力。Hadoop 作为大数据的计算框架,经常要对海量数据分析计算,其底层也是利用 Netty 处理各节点的数据交换。

Netty 作为一个成熟的异步 NIO 通信框架,内置了多种序列化类库,自身也提供了很多附加的功能。国内外大多数公司对 Netty 的使用原理基本一致,都是利用它提供的异步通信能力进行跨节点的数据传输或者服务调用。

## 1.3 本文主要工作

本文主要对 Netty 框架进行研究,并将它作为 RPC 服务器的底层 I/O 调度模块,来提高服务器的网络通信性能。从 RPC 服务器的模块功能开发和远程调用特性出发,首先明确了服务器的功能和性能需求,然后设计并实现了一个基于 Netty 框架的高性能 RPC 服务器,主要功能包括:消息编解码、服务端和客户端 I/O 调度处理、服务的发布和订阅过程、业务线程

池的设计、服务端和客户端调用处理以及长连接服务。对 RPC 服务器进行优化以及提出相关的优化思路。最后，对服务器的各模块功能进行测试。在实现该服务器过程中，对 Netty 框架的关键技术进行深入的研究和学习，总结它优秀的设计理念和技术，并应用于实际的学习开发中。本文一共分为 6 章，分别如下：

第一章：绪论，主要介绍了选题背景和 Netty 的国内外研究现状。

第二章：对 RPC 服务器相关技术的研究，主要介绍了 RPC 服务器的处理过程、远程过程调用原理、负载均衡以及 I/O 模型的演进等。

第三章：RPC 服务器的需求分析和架构搭建，主要介绍了 RPC 服务器的功能和性能需求分析，以及整个服务器架构的搭建，然后对 Netty 的关键技术进行研究。

第四章：RPC 服务器的设计与实现，从业务处理过程出发，对服务器的编解码功能、I/O 调度处理、服务的发布和订阅以及远程调用处理的功能进行实现。

第五章：RPC 服务器的测试和优化，对服务器的各个模块的性能进行测试，并进行初步优化以及提出相关优化方案。

第六章：总结和展望，综合论述了本文的主要工作，对服务器的功能扩展进行了展望。

## 第二章 相关理论与技术研究

### 2.1 RPC 服务器简介

RPC 服务器又叫远程调用服务器，主要作为系统的节点，处理远程业务调用请求。服务器的功能如下：

(1) 用户发起远程业务请求调用；

(2) 请求消息经过网络到达服务器，服务器调度 I/O 线程处理网络 I/O 事件，例如：建立和关闭远程连接、I/O 读写事件监听、处理读写事件等；

(3) 当服务器接收到 TCP 请求消息后，将二进制消息解码为对象，提交到后端容器进行业务处理；

(4) 调用处理成功，返回响应消息时，将对象序列化为二进制流，保证消息能进行网络传输；

(5) I/O 事件处理完毕，服务器需要处理业务请求消息，将 I/O 处理完后得到请求对象投递到后端容器中，找到对应的服务并调用，将响应结果返回给服务消费者。

### 2.2 RPC 远程过程调用

#### 2.2.1 RPC 远程过程调用原理

RPC 远程过程调用是一种客户端/服务端 (c/s) 模式。客户端根据请求方法和端口号，将请求方法和参数列表封装为 RPC 消息<sup>[5]</sup>，调用客户端存根 (client stub) 将消息发送到网络中。消息经过网络到达服务端，服务端存根 (server stub) 接收到消息后，将消息中的请求方法和参数提取出来，根据方法和参数执行服务端的方法。如图 2.1 所示：

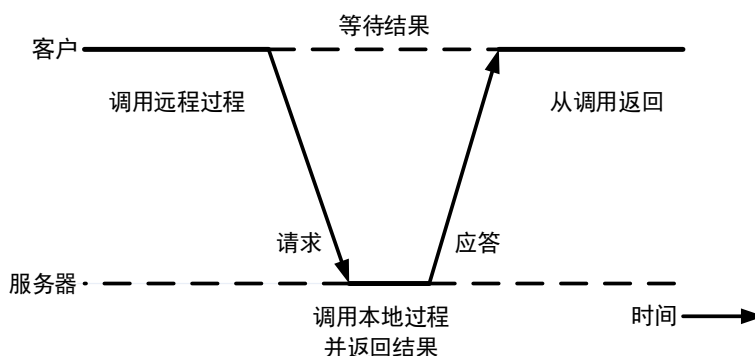


图 2.1 远程调用过程流程



执行结束，server stub 将响应结果封装，并调用 I/O 线程将响应消息发送到网络中。客户端系统监听到 I/O 响应消息<sup>[6]</sup>，根据响应 ID 找到服务请求进程，唤醒该阻塞等待的进程。client stub 从消息中获取到响应对象，然后将对象传递给该请求进程。这一过程对客户端来说，就像调用自己本地的方法一样，屏蔽了底层 I/O 处理的过程。

2.2.2 参数传递机制

客户端发送远程调用请求的过程中会发生参数传递。客户端存根将本地请求参数封装成消息包，并将消息包通过网络发送给服务端，参数封装为消息包的过程叫做参数整编<sup>[7]</sup>。下面根据远程调用 add 加法运算方法来了解参数传递机制，对于远程服务 add(i,j)，根据整型参数 i, j，计算它们的算术和并返回计算结果，方法 add 的调用过程如图 2.2 所示：

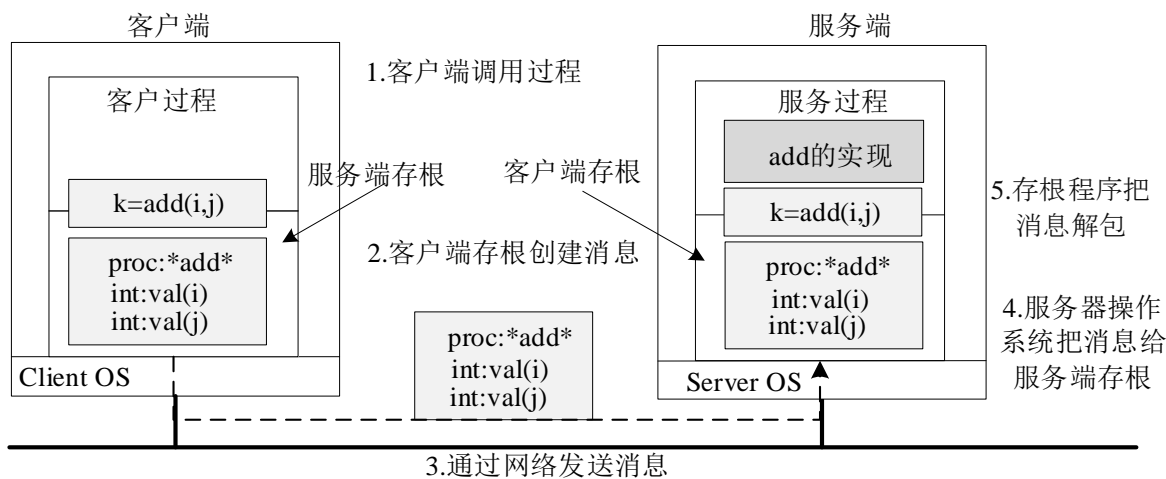


图 2.2 add 方法调用的 RPC 过程

Client stub 首先获取两个参数 i 和 j，根据函数名和请求序号将它们封装成图 2.2 中客户端存根显示的请求消息。请求序号的作用是保证服务端调用结束后，可以根据请求序号将响应消息返回给对应的客户端程序。

消息通过网络到达服务端，服务端存根根据请求方法和参数找到对应的服务执行，这一过程如同发生在服务端本机上的调用一样，只是方法执行的参数来自远程消息<sup>[8]</sup>。

假设有一个整型参数 i 和一个四位的字符串 j，这两个参数都用 32bit 来表示。图 2.3 显示了这两个参数在 client stub 和 server stub 上的参数表现形式。

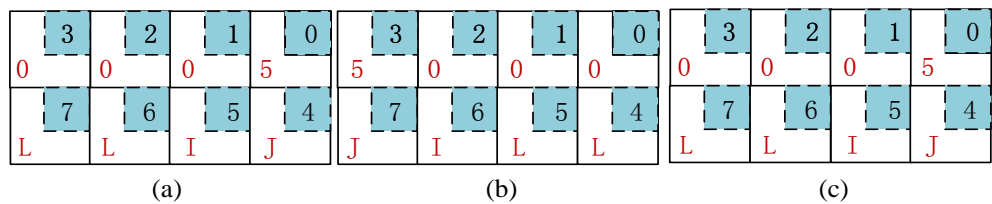


图 2.3 参数表现过程

图 (a) Client stub 上的原始消息, (b) 在 Server 上接收的消息, (c) 是转换后的消息。

消息在网络中, 一般是以字节的形式传送, 字节到达的顺序与传递的顺序一致。

(a) 中 Client stub 的请求参数转化为比特排列, 经过网络传输后, 顺序如 (b) 所示。当 Server stub 分别读取从 0 地址到 4 地址的数据, 会发现该整数数据值等于 83886080, 而字符串为 'ILL', 此时 Server stub 对接收到的 4 字节的位置进行调整。

## 2.3 负载均衡

### 2.3.1 负载均衡原理

在系统中, 一台服务器的处理能力有限, 为了避免服务器负载过大而发生故障, 系统采用服务器集群的形式对外提供服务。客户端消费者通过一种负载均衡算法将远程请求分流到系统的各个节点服务器中, 通过这种方式减轻单台服务器的并发压力<sup>[9]</sup>。图 2.4 给出了系统负载均衡示意图。

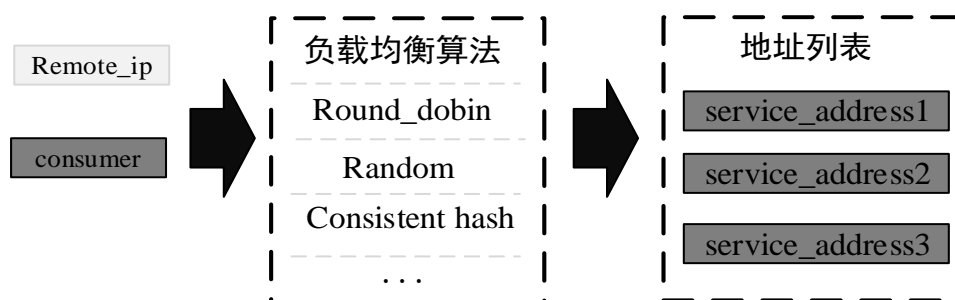


图 2.4 负载均衡示意图

客户端根据本地缓存的远程服务器 IP 地址列表, 利用相关的负载均衡算法计算得到一个负载合适的服务器 IP 地址, 然后根据 IP 发起远程调用。

### 2.3.2 轮询法

#### (1) 轮询法 (Round-Robin)

轮询算法是将远程请求消息依次分配给系统中的服务器进行处理, 从 1 到 N 依次分配, 然后重新循环, 将所有的并发请求消息平均分配到系统的所有服务器中<sup>[10]</sup>。

假设系统中有  $N$  台服务器,  $S = \{S1, S2, \dots, Sn\}$ , 变量  $i$  指向上一次已经选择的服务器 ID。  $i$  初始值为  $N - 1$ 。其算法如下:

```

j = i;
do{
    j = mod(j + 1,n);
    i = j;
    return Si;
} while (j != i);

```

### (2) 权重轮询调度算法(Weighted Round-Robin Scheduling)

由于系统中的服务器性能并不相同，配置不同处理能力也不一样。轮询算法并没有考虑到这种情况，负载平均分配的结果可能导致处理性能强的服务器负载较小，而处理性能弱的服务器负载较大，大量请求阻塞在服务器上，导致系统响应变慢。根据服务器处理性能强弱，给每台服务器分配一个权值，权值大的服务器分配的请求量大，权值小的服务器分配的请求量小，充分利用每台服务器的性能<sup>[11]</sup>。权重轮询调度算法流程如下：

假设系统中有一组服务器  $S = \{S_0, S_1, \dots, S_{n-1}\}$ ,  $i$  指向上次已选服务器, 初始值为-1。

服务器  $S_i$  的权值为  $W(S_i)$ , 变量  $cw$  是当前系统调度的权值, 初始为 0。  $\max(S)$  是  $S$  中的最大权值,  $\gcd(S)$  是  $S$  中所有权值的最大公约数。其算法如下：

```

while (true) {
    i = mod(i + 1,n);
    if (i == 0) {
        cw = cw - gcd(S);
        if (cw <= 0) {
            cw = max(S);
            if (cw == 0)
                return NULL;
        }
    }
    if (W(Si) >= cw)
        return Si;
}

```

## 2.3.3 最少连接数法

### (1) 最少连接数算法

最小连接数算法 (Least Connection) 是根据系统中所有节点的连接情况，选择当前负载最小的服务器，将请求分配给该服务器。该算法需要实时计算服务器连接数，动态估算当前系统中所有服务器的连接情况<sup>[12]</sup>。

在实时计算过程中，保证权值为 0 的服务器不被调用。假设系统中有一个服务器组群

$S = \{S_0, S_1, \dots, S_{n-1}\}$ , 服务器  $S_i$  对应的权值为  $W(S_i)$ , 实时的连接数量为  $C(S_i)$ 。其算法实现如下:

```

for (m = 0; m < n; m++) {
    if (W(Sm) > 0) {
        for (i = m+1; i < n; i++) {
            if (W(Si) <= 0)
                continue;
            if (C(Si) < C(Sm))
                m = i;
        }
        return Sm;
    }
}

```

当服务器处理性能相同时, 最小连接数算法与轮询法类似, 将请求均匀的分配到系统所有服务器中。当服务器性能不同时, 最小连接数算法将更多的请求分配给处理性能高的服务器, 性能低的服务器分配的连接数少<sup>[13]</sup>, 以此提高系统性能。

## (2) 权重最少连接数算法

权重最少连接数 (Weighted Least Connection) 算法是对最小连接数算法的优化。根据处理性能不同, 为系统中每一台服务器分配一个权值, 当请求到达时, 根据系统的连接数以及对应的权值, 将请求分配给相应的服务器<sup>[14]</sup>。其算法流程如下:

假设系统中有一个服务器组群  $S = \{S_0, S_1, \dots, S_{n-1}\}$ , 服务器  $S_i$  对应的权值为  $W(S_i)$ ,  $S_i$  的当前连接数为  $C(S_i)$ 。当前系统中连接数总数为

$$CSUM = \sum C(S_i) (i = 0, 1, \dots, n-1) \quad (2.1)$$

当服务器  $S_m$  满足  $(C(S_m) / CSUM) / W(S_m) = \min\{(C(S_i) / CSUM) / W(S_i)\} (i = 0, 1, \dots, n-1)$  时, 表示  $S_m$  权值和连接数符合要求, 将请求分配给当前服务器。

其中  $W(S_i)$  不为零, 由于  $CSUM$  在当前计算中是一个常数, 因此条件可以简化为:

$$C(S_m) / W(S_m) = \min\{C(S_i) / W(S_i)\} (i = 0, 1, \dots, n-1) \quad (2.2)$$

其中  $W(S_i)$  不为零。一般来说, 计算机进行除法运算所需 CPU 的时间周期比乘法多, 所以判断条件:

$$C(S_m) / W(S_m) > C(S_i) / W(S_i) \quad (2.3)$$

可以进一步优化为  $C(S_m) * W(S_i) > C(S_i) * W(S_m)$ 。同时保证权值为 0 的服务器, 不会被分配请求。其算法实现如下:

```
for (m = 0; m < n; m++) {
    if (W(Sm) > 0) {
        for (i = m+1; i < n; i++) {
            if (C(Sm)*W(Si) > C(Si)*W(Sm))
                m = i;
        }
        return Sm;
    }
}
return NULL;
```

2.4 网络 I/O 模型演进

在当前互联网时代，I/O 性能好坏是大部分网站都要关心的问题。使用 Socket 网络套接字进行网络通信时，I/O 是影响通信性能的重要因素<sup>[15]</sup>。下面是网络 I/O 模型演进过程。

2.4.1 传统的 BIO 模型

BIO（Basic Input Output）又称同步阻塞 I/O，它是一种 Client/Server 模型<sup>[16]</sup>。客户端根据 IP 地址和端口号向服务端发起请求连接，服务端监听到连接请求后，双方开始握手建立连接。连接成功，双方可以通过该连接进行同步阻塞式的数据传输。图 2.5 是 BIO 的服务端通信模型。

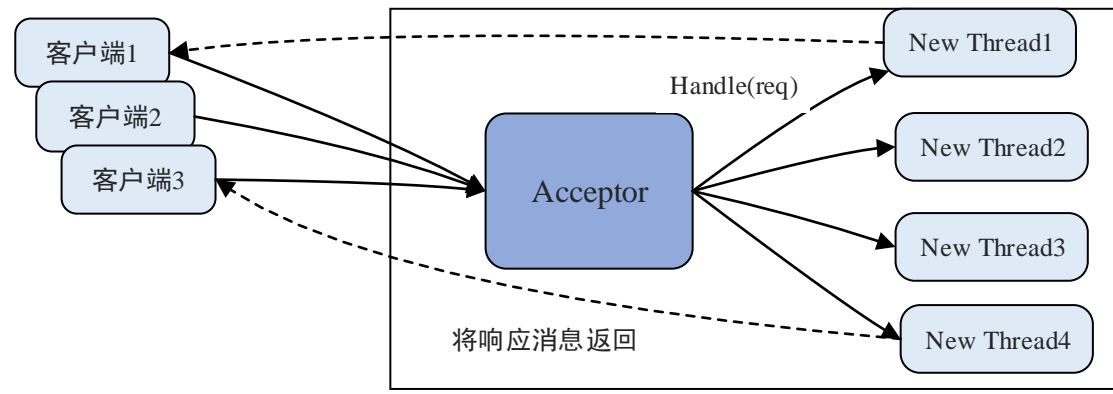


图 2.5 BIO 模型

如图 2.5 所示， Acceptor 线程用来监听客户端的连接请求，当客户端的请求达到时，Acceptor 每收到一个客户端请求，就创建一个新的线程来处理。连接成功后，双方进行通信，通信完毕断开连接，释放线程资源。

BIO 模型采用一个请求创建一个线程的方式，当大量连接请求到达服务端时，服务端需要创建大量的线程，这样会导致系统的线程资源大量占用，系统处理性能下降。当并发请求数过大时，可能导致服务器死机而无法工作。

2.4.2 伪异步 I/O 模型

伪异步 I/O 模型是对 BIO 模型的优化,通过引入线程池,来解决线程资源占用的问题<sup>[17]</sup>。服务端的 Acceptor 线程接收到客户端的 Socket 连接消息,将消息封装为可执行的 Task 任务提交到线程池去执行。通过合理设置线程池的最大线程数 N,以及灵活的线程调度策略,来保证服务端可以利用 N 个线程来处理客户端的 M 个请求,M 可以远大于 N。这样可以减少系统资源的浪费,提高服务端的处理性能。伪异步 I/O 模型如图 2.6 所示:

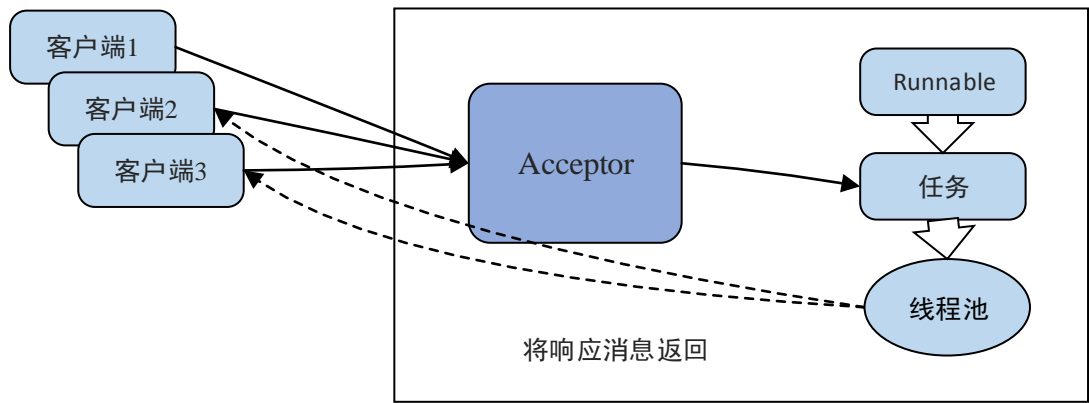


图 2.6 伪异步 I/O 模型

伪异步 I/O 模型只是在线程资源方面做了优化,它无法解决同步 I/O 导致的线程阻塞问题。当服务端与客户端之间响应时间较长时,会级联引发线程阻塞的问题<sup>[18]</sup>。例如:当服务端某个节点发生故障,导致响应时间延迟了 60s,由于 I/O 操作是阻塞的,因此访问该节点的线程也需要被阻塞 60s。当所有的线程读取故障节点时,会导致所有的线程都被阻塞,以至于新的连接消息在阻塞队列中等待。当阻塞队列中的连接数达到最大值时,后续新的连接被拒绝,导致所有响应超时,客户端认为服务端已不可用,停止发送连接消息。

2.4.3 NIO 模型

NIO 是在 JDK1.4 中引入的一种基于通道和缓冲区的 I/O 方式。它可以通过 Native 函数直接在堆外分配内存,将数据存储堆内存中<sup>[19]</sup>,通过 Java 堆中的 DirectByteBuffer 对象引用来操作堆内存中的数据。图 2.7 是 NIO 通信模型:

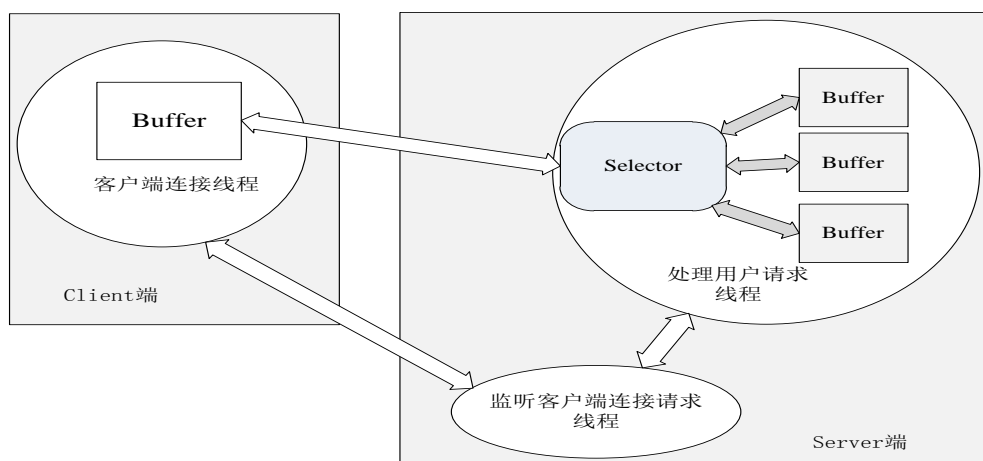


图 2.7 NIO 模型

NIO 是一种同步非阻塞的 I/O 模型。它的核心是 Selector 选择器、Buffer 缓冲区和 Channel 通道。Selector 不断轮询监听 Channel 中是否有就绪的 I/O 事件，当 I/O 请求到达时，Selector 将数据通过 Channel 写入到对应的 Buffer 中，异步监听 I/O 处理结果。服务端线程将数据通过 Channel 写入到 Buffer 后即可返回，无需阻塞等待结果。服务端处理完毕，Selector 监听到通道中的 I/O 响应事件，根据请求标识号，将响应消息返回给对应的客户端进程。

### （1）缓冲区 Buffer

缓冲区 Buffer 是一个对象，用来存储数据。服务端线程将客户端请求数据通过 Channel 写入到 Buffer，然后其他线程从 Buffer 中读取数据进行处理。Buffer 是 NIO 区别于原 I/O 方式的一个重要的特性<sup>[20]</sup>。

Buffer 实际上是一个数组，根据数组的类型存储对应类型的数据。它是在数组的基础上进行了扩展，内置了用于读取和写入数据的 limit 位置信息。在编程过程中使用最多的缓冲区是 ByteBuffer，它是一个字节数组，存储字节信息。

Buffer 的种类很多，除了 ByteBuffer 还有 CharBuffer、ShortBuffer 以及 IntBuffer 等缓冲区，根据需求可以选择合适类型的缓冲区。

### （2）通道 Channel

Channel 也是 NIO 模型中一个重要的特性，它是一个通道，Buffer 中所有数据的读写都是通过 Channel 进行<sup>[21]</sup>。Channel 与流不一样，流（输入流或者输出流）是单工的，只能朝一个方向进行。Channel 是全双工的，它可以同时进行读写操作，这意味着服务端线程从 Channel 中写入数据时，其它线程可以从 Channel 中读取数据，这样极大地提升了处理速度。

### （3）多路复用器 Selector

多路复用器 Selector 主要用来监听通道中的 I/O 事件。一个 Selector 可以同时监听多个注册在其上的 Channel 通道，当 Selector 轮询到某个 Channel 中有就绪的 I/O 事件时，如：新的

TCP 连接请求、I/O 读写事件，会通过它内置的 `SelectionKey` 获取到当前就绪的 `Channel`，然后进行相应的操作。

`Selector` 的轮询过程只需要一个线程，通过一个线程同时监听多个 `Channel`。JDK 使用 `epoll()` 来实现轮询过程，避免了最大连接句柄 1024/2048 的限制，可以同时接入数万的客户端。

## 2.5 本章小结

本章首先对 `RPC` 服务器进行了简单介绍，然后分析了远程过程调用原理以及参数传递机制，接着研究了负载均衡相关理论和算法，最后根据分析整个网络 I/O 模型的演进过程，来了解网络异步 I/O 的优势。



## 第三章 RPC 服务器的需求分析与架构设计

### 3.1 服务器的需求分析

#### 3.1.1 服务器的功能需求分析

RPC 服务器主要用来处理远程请求消息，它基本功能需要包括：高效的 I/O 处理能力，能同时处理大量远程调用请求；并发处理业务请求调用，并且通过异步调用方式，返回响应消息给调用者；同时也能向远程服务器订阅服务等等。本文基于 Netty 实现的高性能 RPC 服务器能高效处理用户大量的远程 RPC 请求信息，实现的功能如下：

- (1) 高效处理海量的 I/O 事件；
- (2) 监听 TCP 请求连接，创建和断开链路，处理请求消息；
- (3) 接收请求消息时，对 TCP 请求信息解码，解决 TCP 的半包问题；
- (4) 发送 TCP 消息时，对 TCP 消息编码，将消息长度信息添加到消息头，以便远程服务器根据长度信息解码；
- (5) 将接收的二进制消息解码，解码得到的请求对象传递给业务处理模块；
- (6) 发送远程消息时，对消息对象编码，编码得到的二进制流用于网络传输；
- (7) 并发处理远程调用请求；
- (8) 作为服务提供者发布服务，作为消费者订阅自己所需的服务；
- (9) 监听网络连接的可靠性，通过心跳机制维持长连接。

#### 3.1.2 服务器的性能需求分析

RPC 服务器作为系统的服务提供节点，每天都会有大量的请求连接接入，因此对服务器的性能要求很高。为了保证消费者能快速得到响应消息，需要服务端能处理高并发请求，并且快速可靠的响应请求。RPC 服务器的性能需求如下：

##### (1) I/O 性能

RPC 服务器处理的 RPC 消息数量庞大，可能在某个时间点，接收到数以万计的并发请求。这对于服务器的 I/O 并发处理能力的要求很高，I/O 事件处理是很耗时的，采用合理的 I/O 并发框架，可以保证系统 I/O 性能。

(2) 可靠性

RPC 服务器采用了长连接的机制，由于长连接不需要每次发送消息都创建链路，也不需要消息交互完成时关闭链路<sup>[22]</sup>，因此相对于短连接性能更高。为了保证长连接的链路有效性，通过心跳周期性地对链路检测来保证链路的可靠性。

(3) 高并发

RPC 服务器将大量请求对象从通信模块传递到业务处理模块中，业务处理模块需要并发处理这些调用请求。调用处理完毕，通过异步回调技术，正确并且迅速响应请求消息。

3.2 服务器总体架构设计与实现

3.2.1 服务器总体架构

RPC 服务器主要利用 Netty 框架来实现底层 I/O 的通信，Netty 是一个高性能的异步通信框架，利用它可以提高服务器的通信性能。服务器的总体架构如图 3.1 所示。

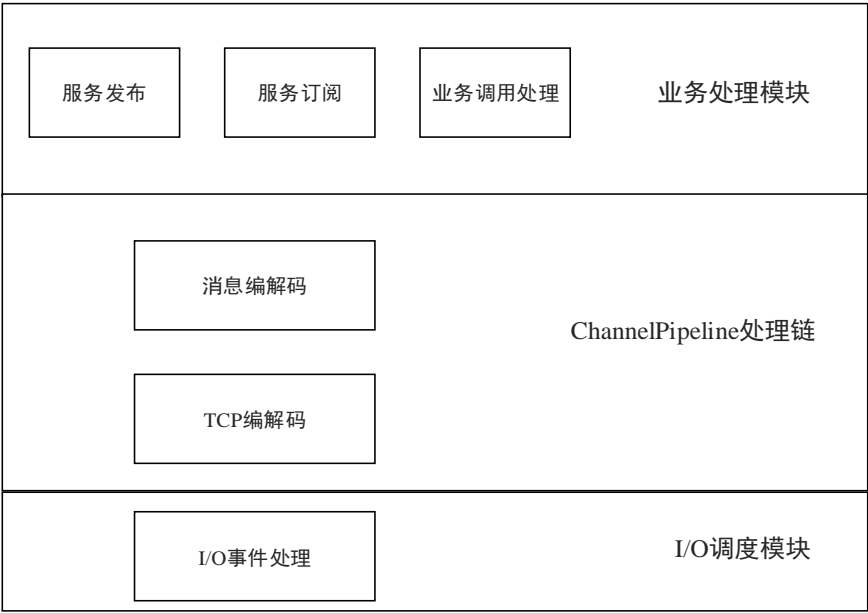


图 3.1 服务总体架构图

I/O 通信调度模块主要用来处理服务器的 I/O 事件，例如链路断连、读写事件处理、链路有效性检测以及消息处理<sup>[23]</sup>。根据的 I/O 事件的不同，分为服务端和客户端两种 I/O 的处理过程。服务器同时需要处理大量的远程调用消息，为了减少链路断连造成的性能问题，通过心跳检测机制实现长连接的功能。

编解码模块主要对 RPC 请求和响应消息进行编解码处理。服务器采用 TCP 协议进行远程调用，TCP 在传输过程中会出现粘包/拆包问题，因此需要对接收的 TCP 消息进行半包解

码。发送 TCP 请求进行远程调用时，需要对消息进行长度编码，以便接收端根据长度字段进行解码。RPC 请求消息要在网络中进行传输，需要将请求对象序列化为二进制流；对于接收到的字节流要反序列化为请求对象。本文通过引入第三方编解码框架，提高服务器的编解码性能。编解码过程主要在 Channel 中的 ChannelPipeline 处理链中实现。

业务处理模块包含服务的发布和订阅过程。作为服务提供者发布服务，接收远程调用请求，并通过线程池的设计，将所有的请求投递到业务线程池去并发处理<sup>[24]</sup>；作为服务消费者订阅服务，发送远程调用请求，异步接收响应消息。

### 3.2.2 RPC 服务器远程调用流程

下面是服务器处理 RPC 远程调用请求的过程，服务处理流程如图 3.2 所示：

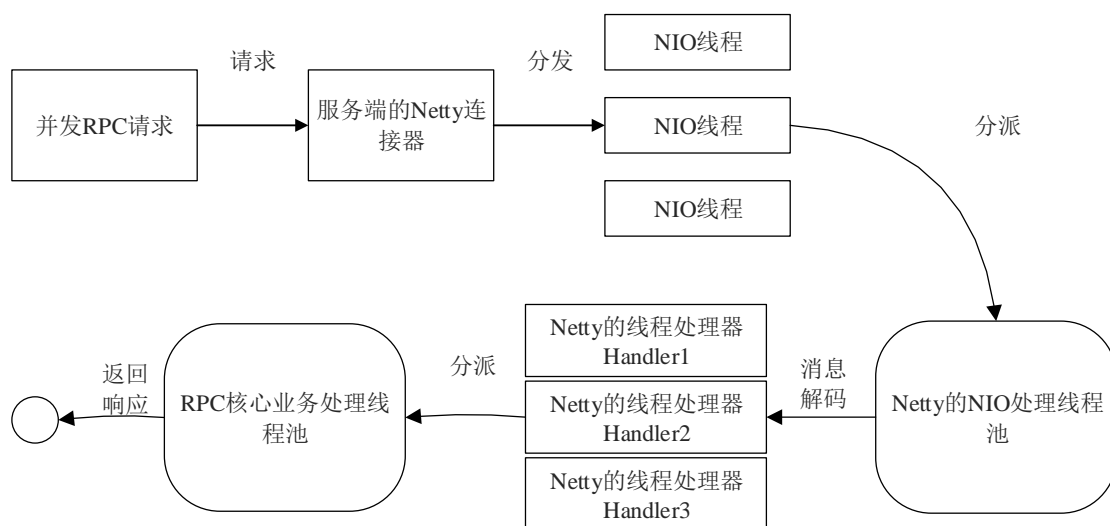


图 3.2 服务处理流程图

- (1) 并发 RPC 请求消息到达服务器；
- (2) 服务端 Netty 连接器检测到请求到达，建立 TCP 连接；
- (3) 将接收的 RPC 请求消息分发给 Netty 的 I/O 线程处理；
- (4) I/O 线程将请求消息传递到 Netty 的 I/O 处理线程池中，由线程池去调度处理；
- (5) 线程池分配线程对请求消息进行编码、解码工作，消息会依次经过 ChannelHandler 链处理；
- (6) 消息经过一系列的解码过程后，提交到业务处理线程池进行远程方法调用；
- (7) 调用结束，异步返回响应信息。

### 3.2.3 开发环境搭建

#### (1) 软件环境

开发软件: IntelliJ IDEA 2016.1.3

JDK 版本: JDK 1.8.0\_121

#### (2) 硬件环境

操作系统: Windows 64 位操作系统, 基于 x64 的处理器

处理器: Intel(R) Core(TM) i3-2310M 2.10GHz

内存: 6G

硬盘: 500G

整个服务器系统基于 Spring 框架搭建, 将服务器的启动类配置在 Spring 配置文件中, 通过配置文件启动服务器。在使用 Spring 开发时, 需要添加第三方开源 jar 包来实现对应的功能。搭建系统步骤如下:

(1) 在 IntelliJ Idea 开发工具中, 创建一个 Maven 工程, 利用 Maven 仓库来管理 jar 包。即使项目部署在其他平台上, 也可以根据 Maven 仓库, 在网上自动下载相应的依赖包;

(2) Maven 工程创建后, 会在项目目录中生成一个 pom.xml 的文件, 用来添加项目开发所需的 jar 包;

(3) 在 pom.xml 文件中添加开发使用的基本 jar 包依赖, 例如 Spring 框架相关的依赖包、log4j 以及 commons 等。管理 Maven 自动更新下载的 jar 包, 防止产生 jar 包版本冲突;

(4) 将 Netty 依赖添加到 pom.xml 中, 这里使用的是 Netty 的 4.0.36 版本, 如图 3.3 所示;

```
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-all</artifactId>
  <version>4.0.36.Final</version>
</dependency>
```

图 3.3 Netty 依赖配置

(5) 配置本地 Tomcat 容器, 提供系统启动的环境, 如图 3.4 所示;

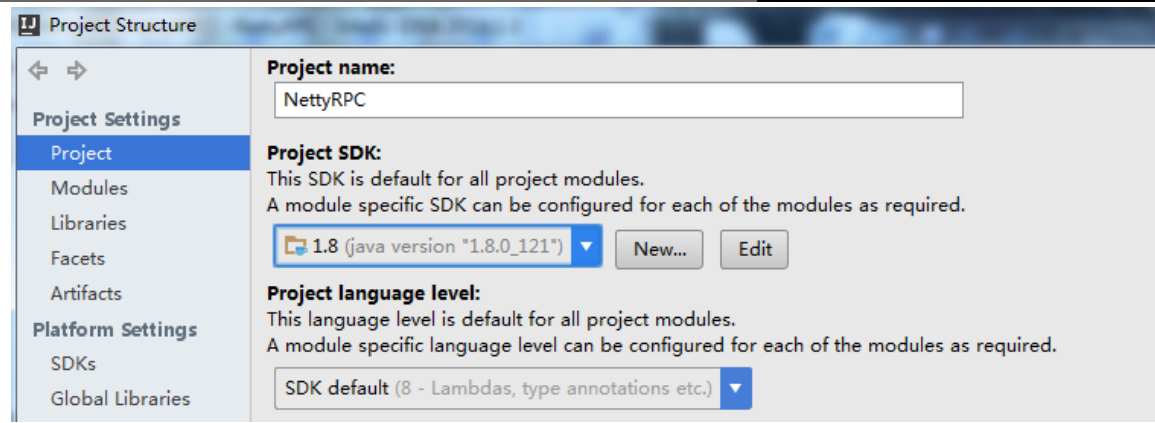


图 3.4 Tomcat 环境配置

（6）创建 Spring 的配置文件 xxx.xml，将服务配置在其中，在 Tomcat 容器启动的时候，加载服务，启动服务器。

### 3.3 Netty 框架技术

#### 3.3.1 Netty NIO 模型

##### （1）缓冲区 ByteBuf

Netty ByteBuf 是一个 Byte 数组的缓冲区，它的基本功能与 JDK 的 ByteBuffer 一致。Netty 利用两个指针（readerIndex、writerIndex）对缓冲区 ByteBuf 进行操作<sup>[25]</sup>。从 ByteBuf 中读取数据时，根据 readerIndex 指向的位置获取对应区间内的数据；当数据写入 ByteBuf 中时，利用 writerIndex 指针的移动，将数据写入到对应的区间位置中。

##### （a）读写操作

readerIndex 和 writerIndex 的初始值为 0，写入数据时 writerIndex 增加，读取数据时 readerIndex 增加，但是都不会超过 capacity 容量，且 readerIndex 不会超过 writerIndex。

0 和 readerIndex 之间的数据是可丢弃的，读取操作完成之后，readerIndex 指针向前移动区间长度，然后调用 discardReadBytes 方法清除该区间的内存。初始分配的 ByteBuf 如图 3.5 所示：

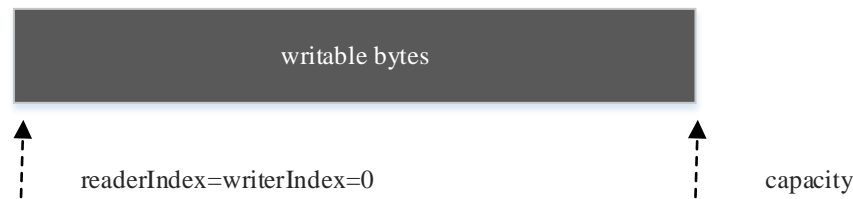


图 3.5 ByteBuf 的初始情况

写入 N 个字节之后的 ByteBuf 如图 3.6 所示：

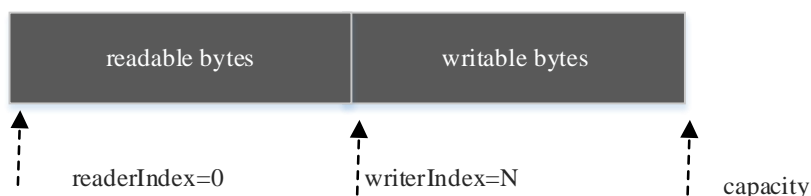


图 3.6 ByteBuffer 中写入 N 个字节

读取  $M (< N)$  个字节之后的 ByteBuffer 如图 3.7 所示:

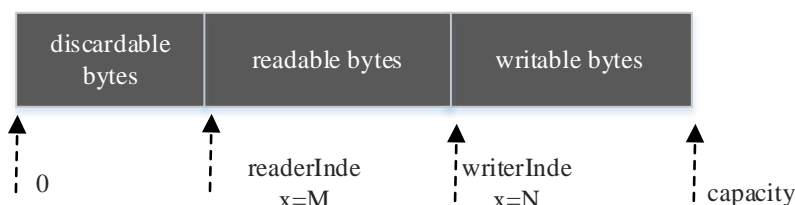
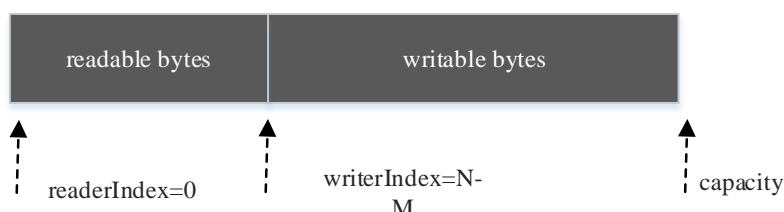
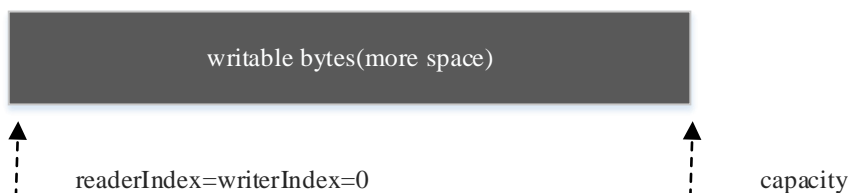


图 3.7 ByteBuffer 中读取 M 个字节

调用 `discardReadBytes` 操作之后的 ByteBuffer 如图 3.8 所示:

图 3.8 `discardReadBytes` 释放 ByteBuffer 中的部分内存

调用 `clear` 操作之后的 ByteBuffer 如图 3.9 所示:

图 3.9 `clear` 释放 ByteBuffer 中的所有内存

读操作发生时, 只会操作 `readerIndex` 指针; 写操作进行时, 只会移动 `writerIndex` 指针。这两个指针各自执行与自身相关的操作, 不会相互响应。通过 `readerIndex` 和 `writerIndex` 可以灵活地操作 ByteBuffer。

#### (b) 动态扩容

ByteBuffer 是一个动态数组, 它可以根据存储情况自动扩容, 动态扩容只会发生在 `write` 操作的过程中。需要写入字节数据时, ByteBuffer 首先判断当前剩余的空间是否足以保存写入的数据, 若要写入的数据大小大于 ByteBuffer 的剩余空间, ByteBuffer 重新分配空间, 将原空间中的数据复制到新的空间中, 通过这种方式实现动态扩容过程。

由于 ByteBuf 的动态扩容特性,对于程序员来说,可以不必关心它的底层具体实现过程,只需要利用它提供的 API 即可方便的操作 ByteBuf,实现读写操作<sup>[26]</sup>。ByteBuf 不能无限扩容,它有一个最大的存储值,当它的内存达到最大值时,不会再自动扩容,避免造成内存的溢出。

## (2) 通道 Channel

在 Netty 框架中,Channel 的使用方式与 NIO 框架一致,主要利用 Channel 通道对缓冲区 ByteBuf 进行数据的读写以及处理网络连接事件。Netty 对 Channel 重新进行了设计,赋予了它更多的功能,例如:可以获取工作在 Channel 上的 NioEventLoop 线程以及 Channel 中的 ChannelPipeline 处理链等。

Netty 相对于 NIO 在以下几个方面对 Channel 进行了优化:

(a) 接口层,利用 Façade 模式对 Channel 中的网络 I/O 操作进行封装,包括读、写、TCP 连接以及其他相关操作。通过暴露接口,为其他程序提供服务功能。

(b) 重新对客户端 SocketChannel 和服务端 ServerSocketChannel 优化,将它们重合的部分功能抽象提取出来,由它们对应的子类去实现,避免代码的重用。

(c) Channel 是一个接口,在接口中定义相关的功能,具体的实现由子类通过聚合的方式实现,Channel 根据具体功能需求<sup>[27]</sup>,灵活调度子类处理。

## 3.3.2 Netty 的线程模型

### (1) 线程模型

Netty 的线程模型不是固定的,通过合理的参数设置,Netty 可以在 Reactor 单线程、多线程和主从多线程模型中进行切换<sup>[28]</sup>。Netty 的线程模型如图 3.10 所示。

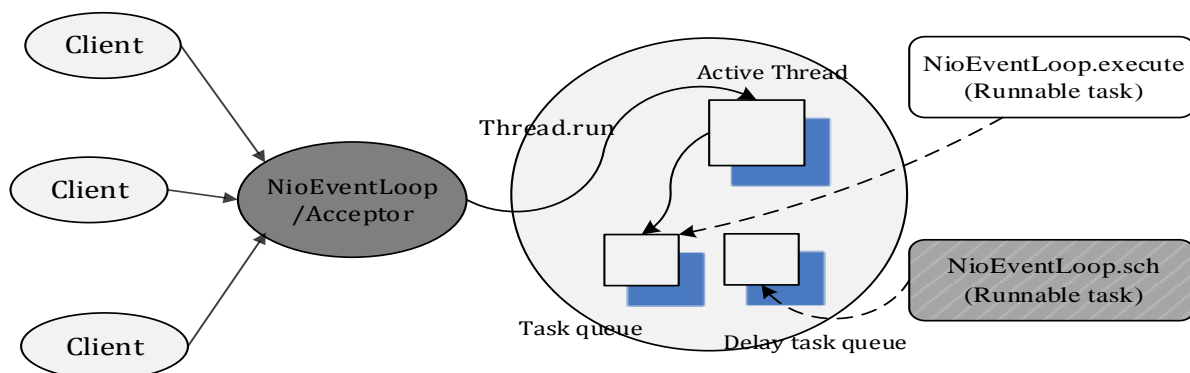


图 3.10 Netty 的线程模型

在 Netty 服务端启动的过程中,会创建两个 NioEventLoopGroup 线程组,一个用于接收 TCP 请求,一个用来处理 I/O 事件或者定时任务等。

用于接收 TCP 请求的线程组的职责如下：

(a) 当客户端发送 TCP 连接请求时，与客户端握手建立连接，并对服务端 Channel 通道进行初始化；

(b) 若连接发生改变，将变动事件传递给 ChannelPipeline 处理。

负责处理 I/O 事件的线程组的职责如下：

(a) 异步获取客户端的远程请求消息，并传递到 ChannelPipeline 中；

(b) 利用 ChannelPipeline 将响应消息发送到客户端；

(c) 执行 I/O 任务和定时任务；

(d) 空闲状态，检测链路有效性。

## (2) NioEventLoop 线程

NioEventLoop 是 NioEventLoopGroup 中的线程，它不仅需要处理 I/O 事件，而且还要处理各种系统任务<sup>[29]</sup>。

(a) 系统调用的 Task。当 I/O 线程和用户线程同时操作网络资源时，为了避免并发造成的资源竞争问题，将用户操作封装为可执行的 Task 任务，投递到线程池中调度处理。

(b) 定时任务。通过调用 NioEventLoop 的 schedule(Runnable command, long delay, TimeUnit unit)方法实现。

在 Netty 的线程模型中，NioEventLoop 线程采用串行的方式对消息进行处理，避免线程频繁切换，造成资源的损耗。消息的所有处理过程都是在同一个线程中进行，不会产生数据被修改的风险。

一个 NioEventLoop 线程只有一个客户端注册，通过这种串行化的设计理念不仅降低程序员开发难度，也提升了服务器的处理能力。线程组中所有线程并行执行，互相没有交集，避免了线程上下切换带来的性能损耗，也充分利用了多处理器并行处理的能力。

### 3.3.3 Netty 的 ChannelPipeline 处理链

Netty 的 Channel 过滤器与 Servlet Filter 机制一致，它将 Channel 的数据管道抽象为 ChannelPipeline，消息在 ChannelPipeline 中流动和传递。ChannelPipeline 是一个由 ChannelHandler 组成的处理链，每一个 ChannelHandler 的功能不一样，通过 ChannelHandler 的各种不同组合，对消息拦截处理，实现的业务操作<sup>[30]</sup>。图 3.11 是一个消息被 ChannelPipeline 拦截和处理的过程。



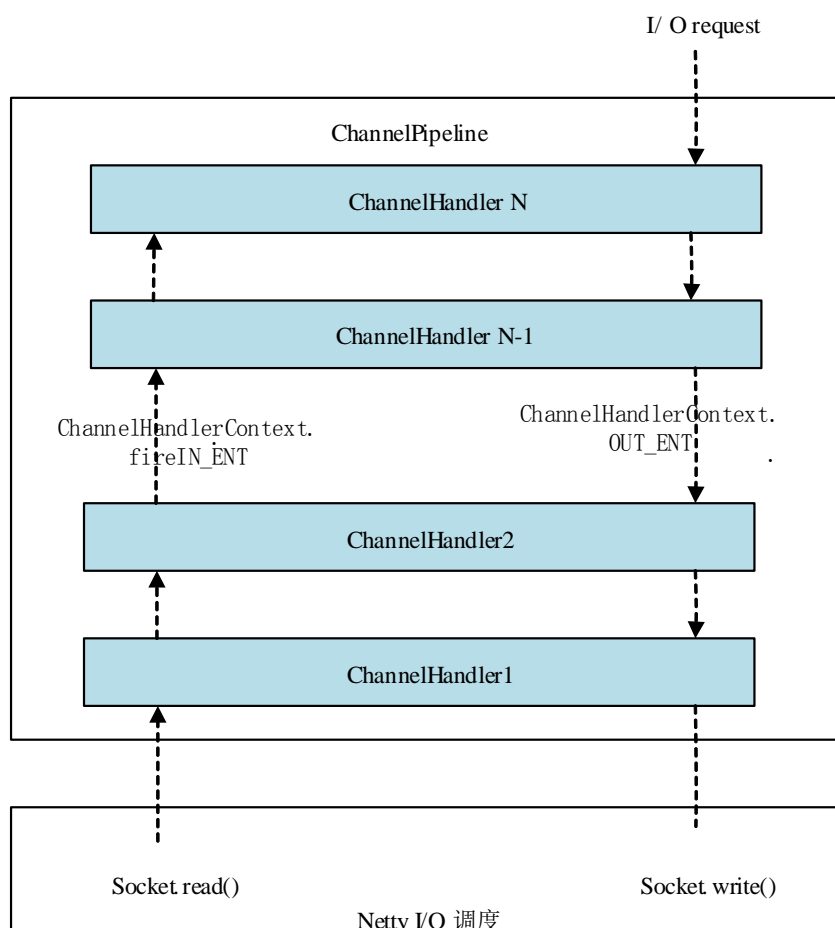


图 3.11 Netty 的 ChannelPipeline 处理过程

(1) 当有消息到达底层的 SocketChannel 通道时，SocketChannel 会调用 read()方法读取 ByteBuf 缓冲区里面的数据。然后，NioEventLoop 线程会执行 ChannelPipeline 的 fireChannelRead()方法，将缓冲区的字节数据复制到 ChannelPipeline 职责链中；

(2) ChannelPipeline 中的 ChannelHandler 处理一个或多个事件，需要将消息依次通过 HeadHandler1、ChannelHandler2.....TailHandler 这一连串 ChannelHandler 拦截和处理，如 TCP 半包解码器和 RPC 消息编码、解码的 ChannelHandler 也是处于职责链中；

(3) ChannelHandlerContext 封装一个具体的 ChannelHandler，并为 ChannelHandler 的执行提供线程环境，它调用的 write 方法发送消息，消息在各个 ChannelHandler 之间传递，消息经过处理之后，会被 NioEventLoop 线程保存在 ByteBuf 缓冲区中等待处理。

### 3.4 本章小结

本章主要从服务器的角度出发，对 RPC 服务器的功能和性能需求进行分析，然后针对需求搭建本系统的框架，详细介绍了系统的搭建过程，最后对 Netty 的 I/O 模型、线程模型以及 ChannelPipeline 处理过程进行研究。

## 第四章 基于 Netty 的 RPC 服务器的实现

### 4.1 消息定义

表 4.1 和表 4.2 分别是 RPC 服务器用于远程调用的请求消息和响应消息的模型。

表 4.1 RPC 请求消息 MessageRequest 的定义

消息字段	消息类型	消息描述
messageId	String	RPC 消息 ID 序号
className	String	请求类名
methodName	String	请求方法
typeParameters	Class<?>[]	请求参数类型
parametersVal	Object[]	请求参数

MessageRequest 是远程调用请求消息模型。messageId 用来标识当前 RPC 消息的序号，在请求执行后，根据 messageId 将响应消息返回给对应的请求线程；className 是远程请求调用的类名；methodName 是请求调用的类的方法；typeParameters 是调用方法对应的参数列表的参数类型；parametersVal 是调用方法对应的参数列表。

表 4.2 响应消息 MessageResponse 的定义

消息字段	消息类型	消息描述
messageId	String	响应消息 ID 序号
error	String	错误信息
result	Object	响应信息对象

MessageResponse 是远程调用的响应消息模型。messageId 对应请求消息的 messageId，用于将响应消息返回给对应的请求进程；error 表示远程调用发生异常时，异常报错消息；result 是远程调用执行成功后，返回给对应消费者的响应信息对象。

### 4.2 编解码模块

编解码模块主要对 I/O 线程接收到的 TCP 消息进行处理，包括 TCP 协议半包问题、TCP 长度编码以及消息的编码、解码等<sup>[31]</sup>。根据功能设计 ChannelHandler 处理器，并将这些自定义的 ChannelHandler 注册到 ChannelPipeline 处理链中，通过消息处理链将 ByteBuf 流转化为上层可以执行的 POJO 对象，或者将上层响应的 POJO 对象消息转化为 ByteBuf 流供通信调度层处理。

### 4.2.1 TCP 半包解码

由于 TCP 协议在网络传输过程中会出现半包问题,本文设计的 RPC 服务器通过使用 Netty 框架提供的 `LengthFieldBasedFrameDecoder` 解码器来处理 TCP 粘包/拆包问题。

`LengthFieldBasedFrameDecoder` 解码器可以根据传入的参数,自动解决半包问题。在初始化时,根据构造器参数的不同组合,解决不同的“读半包”问题。

#### (1) 基于长度的拆包

字节缓冲区中的消息由长度字段和消息体构成,消息中只有一个长度字段,并且位于消息头部。消息结构定义如图 4.1 所示:

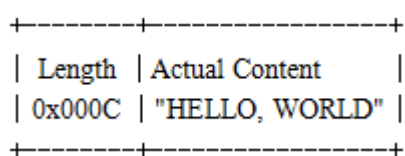


图 4.1 解码前字节缓冲区 (14 字节)

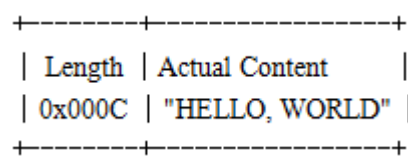


图 4.2 解码后包含消息长度字段 (14 字节)

若需要保留消息的长度字段,参数组合如下:

```
lengthFieldOffset = 0;      lengthFieldLength = 2;
lengthAdjustment = 0;      initialBytesToStrip = 0;
```

解码后的字节缓冲区内容如图 4.2 所示

#### (2) 基于长度的截断拆包

假如字节缓冲区经过解码后,只需要保留消息体的内容,则只要将 `initialBytesToStrip` 设置为 2,即可去除 2 字节的长度字段,参数组合如下:

```
lengthFieldOffset = 0;      lengthFieldLength = 2;
lengthAdjustment = 0;      initialBytesToStrip = 2;
```

解码后的字节缓冲区内容如图 4.3 所示

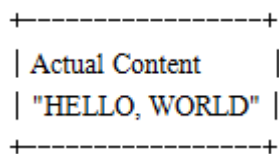


图 4.3 仅包含消息长度字段 (12 字节)

#### (3) 基于偏移长度的拆包

消息的长度字段处于消息的中间或者尾部时,利用 `lengthFieldOffset` 设置标识,将长度字段设置在首部,参数组合如下:

```
lengthFieldOffset = 2;      lengthFieldLength = 3;
lengthAdjustment = 0;      initialBytesToStrip = 0;
```

解码前后，缓冲区消息如图 4.4 和图 4.5 所示：

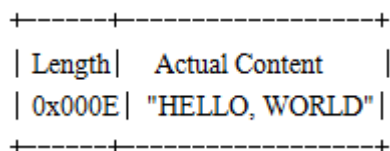


图 4.4 解码前(14 bytes)

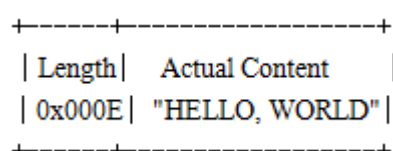


图 4.5 解码后 (14 bytes)

#### (4) 基于偏移可调整长度的截断拆包

长度字段位于消息的中间，处于消息头或者消息体之间，通过 `initialBytesToStrip` 参数，可以去除消息长度字段以及之前的其他字段，参数组合如下：

```
lengthFieldOffset = 1;      lengthFieldLength = 2;
lengthAdjustment = 1;      initialBytesToStrip = 3;
```

解码前后，缓冲区消息如图 4.6 和图 4.7 所示：

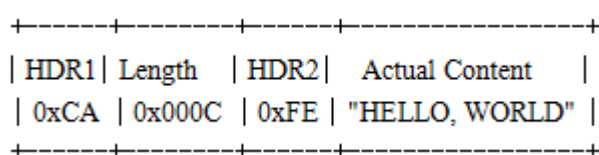


图 4.6 编码前 (16 字节)

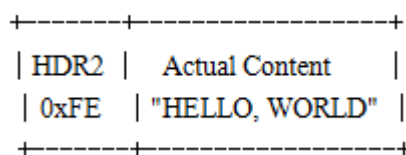


图 4.7 编码后 (13 字节)

创建 `LengthFieldBasedFrameDecoder` 对象时，根据传入的 4 个参数的灵活组合，可以实现不同的业务解码效果，大大降低开发时的难度。

#### (a) 编码器配置

RPC 服务器消息头长度 `MESSAGE_LENGTH` 定义的是 2 字节，参数组合设置为 (0,2,0,2) 即可去掉长度字段，得到消息体的内容。代码配置如下：

```
pipeline.addLast(new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 0, 2, 0, 2));
```

将 `LengthFieldBasedFrameDecoder` 添加到 `ChannelPipeline` 中，当请求消息达到时，可以将 `ByteBuf` 缓冲区中的字节码解码为所需的整包消息，然后将完整的数据报传递给后面的 `ChannelHandler` 处理即可。

#### (b) 测试验证

下面是 RPC 服务器使用 Netty 的 `LengthFieldBasedFrameDecoder` 解码器前后服务端和客户端运行情况。

首先客户端向服务端发起连接请求，服务端接收到请求，与客户端握手建立连接。连接创建成功后，客户端循环发送 100 条 RPC 消息给服务端。服务端每读到一条请求消息，就计数一次，然后返回响应消息，以通知客户端已收到请求消息。按预期情况下，两者收到的消息总数是一致的。

在未使用 `LengthFieldBasedFrameDecoder` 解码器处理 TCP 半包问题前, 服务端和客户端出现 TCP 粘包/拆包现象。

服务端读到的消息, 具体现象如图 4.8 所示:

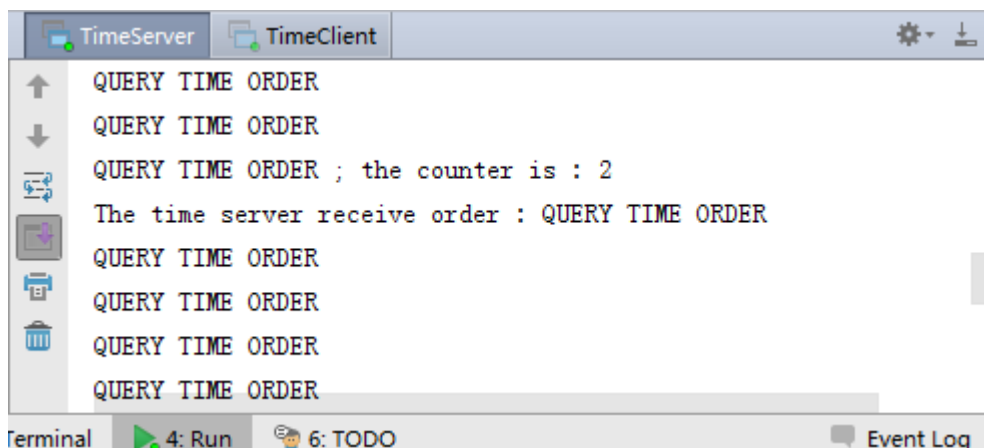


图 4.8 服务端接收到的消息内容

服务端接收到两条消息, 第一条包含 57 条“`QUERY TIME ORDER`”指令, 第二条包含了 43 条“`QUERY TIME ORDER`”指令, 总数 100 条。与我们预期的每一条消息都包含“`QUERY TIME ORDER`”指定不一致, 说明发生了 TCP 粘包。

客户端接收到的消息如图 4.9 所示:

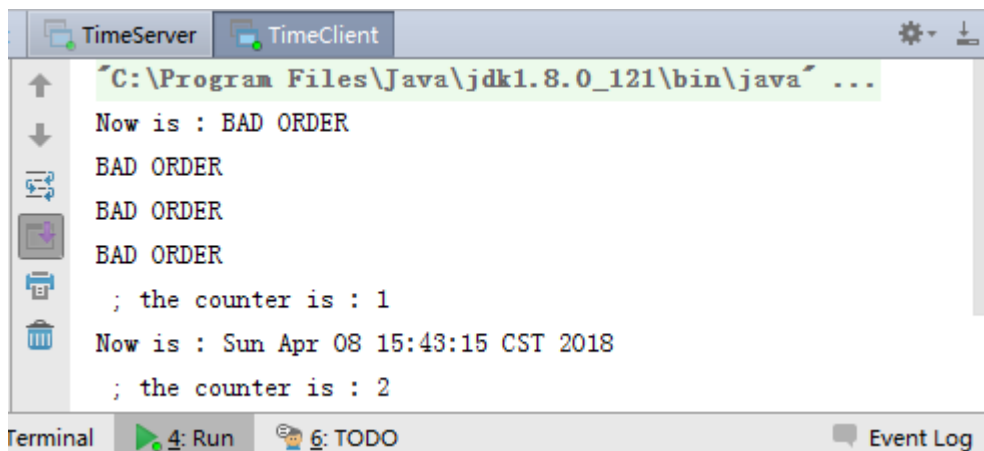


图 4.9 客户端发生粘包问题

客户端应该接收到 100 条当前服务端返回的响应消息, 但是实际上只收到一条消息。由于客户端程序判断响应消息不满足查询条件, 所以返回两条“`BAD ORDER`”响应消息, 说明服务端返回的响应消息也发生了粘包。

将 `LengthFieldBasedFrameDecoder` 解码器添加到 `ChannelPipeline` 职责链中, 利用 `LengthFieldBasedFrameDecoder` 解码器处理 TCP 半包问题后, 服务端读到的消息如图 4.10 所示:

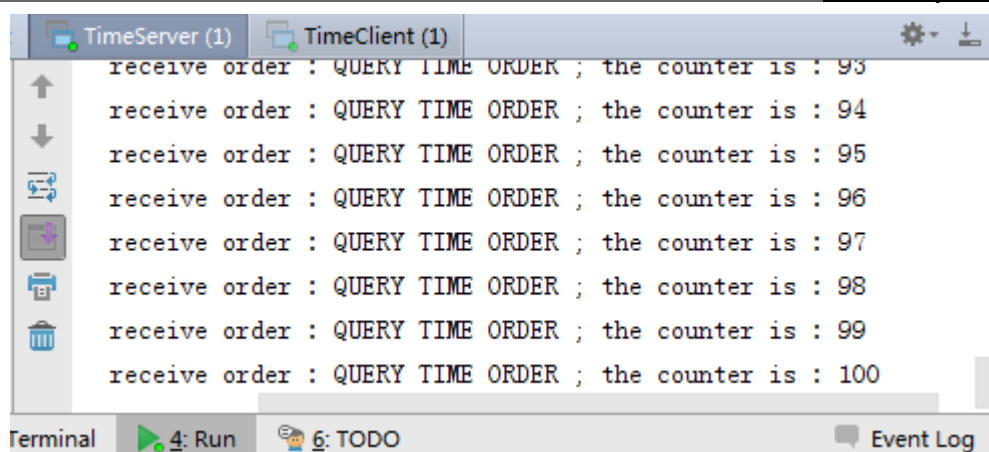


图 4.10 服务端读到的消息

客户端接收的消息如图 4.11 所示：

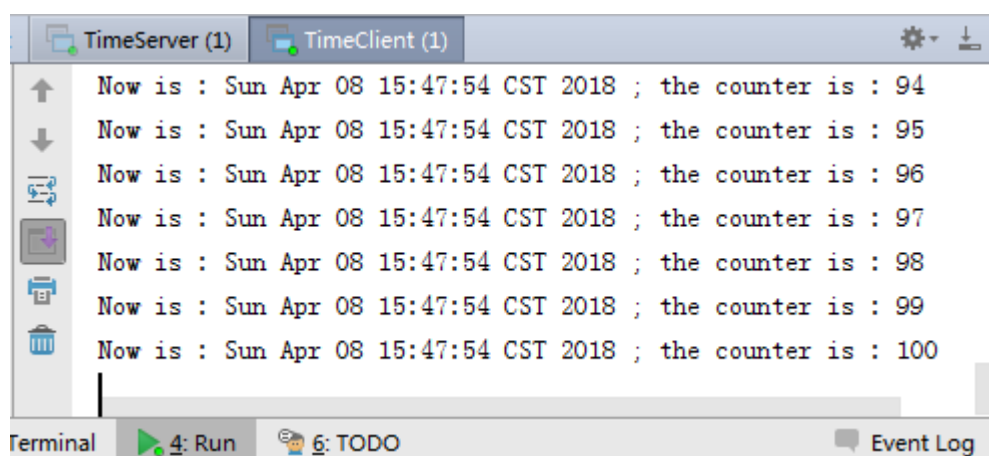


图 4.11 客户端接收的消息

服务端和客户端运行结果完全符合预期，说明 TCP 粘包问题已经解决。

## 4.2.2 TCP 消息长度编码

当 RPC 服务器向远程发送远程调用消息时，由于 RPC 协议携带了长度字段，因此需要在待发送的 `ByteBuf` 消息头中增加长度字段来标识消息长度，以保证被调用方能根据长度解析消息。本文利用 Netty 的 `LengthFieldPrepender` 编码器，对 TCP 消息长度进行编码，将消息长度添加到 `ByteBuf` 缓冲区头。代码如下：

```
pipeline.addLast(new LengthFieldPrepender(MessageCodecUtil.MESSAGE_LENGTH));
```

在 `ChannelPipeline` 中增加 `LengthFieldPrepender` 解码器，指定参数组合。在将消息发送到网络之前，通过 `LengthFieldPrepender` 将 2 字节长度字段添加到消息头中，编码后的消息组成长度字段+原消息的形式。编码前后结果如图 4.12 和图 4.13 所示：

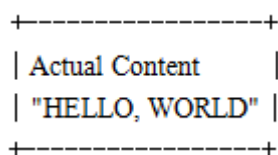


图 4.12 编码前（12 字节）

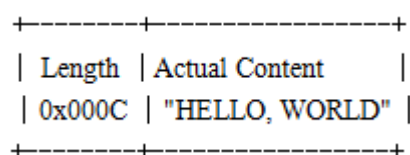


图 4.13 编码后（14 字节）

### 4.2.3 消息编解码

Netty 提供了 Java 原生序列化方式实现的 `ObjectEncoder` 和 `ObjectDecoder`（对象编码、解码器），但是由于 Java 原生序列化方式性能和效率不高<sup>[32]</sup>，因此，本文通过引入第三方优秀的编解码框架来提升服务器的编解码性能，本次引入的第三方编解码框架有 `Kryo` 和 `Hessian`。

#### （1）Kryo 序列化。

`Kryo` 是一个高效的 Java 序列化框架，相对于 Java 本地序列化机制，它在码流大小以及序列化性能上都有很大的优势，无论是本地文件数据，还是网络数据，它都能进行序列化。本文中采用的版本是：`kryo-3.0.3`。

#### （2）Hessian 序列化

`Hessian` 是一种采用二进制方式传输的序列化框架，相对于 Java 本地序列化，它的速度更快，序列化后的数据更小，数据变小意味着同一时间可以传输更多的数据，性能更佳。本文中采用的版本是：`hessian-4.0.37`。

`MessageCodecUtil` 是一个接口，接口中定义了两个抽象方法 `void encode(final ByteBuf out, final Object message)` 和 `Object decode(byte[] body)`，自定义的编解码器通过实现这两个方法来实现服务器的编码和解码功能。利用 `MessageCodecUtil` 接口可以增强系统的扩展性，增加一种编码方式只需要实现这两个方法即可。

#### （a）Kryo 编码

本文利用 `Kryo` 框架的 API 实现 `Kryo` 的编解码功能。在 `KryoSerialize` 类中实现了 `serialize` 序列化和 `deserialize` 反序列化方法，`serialize` 方法实现了请求对象到字节流的序列化过程，代码实现如下：

```
public void serialize(OutputStream output, Object object)
    throws IOException {
    Kryo kryo = pool.borrow();
    Output out = new Output(output);
    kryo.writeClassAndObject(out, object);
    out.close();
    output.close();
    pool.release(kryo);
}
```



首先引入了对象池的概念, 利用对象池对对象进行管理。对象的创建和初始化需要消耗一定的时间, 当服务器需要创建大量对象时, 耗时很大。利用 Kryo 对象池创建一定数量的 Kryo 对象, 程序只需要从对象池中获取 Kryo 对象, 使用结束将对象归还到对象池中, 减少了对对象的频繁创建, 避免产生大量对象造成性能的影响<sup>[33]</sup>。

borrow()函数获取 Kryo 对象后, 调用 Kryo 框架的 writeClassAndObject 方法将对象序列化为字节流并保存在 Output 输出流中。序列化结束, 关闭所有资源, 并释放 Kryo 对象。

KryoEncoder 编码器间接继承 Netty 的 MessageToByteEncoder 抽象编码器, 只需要实现 void encode(ChannelHandlerContext ctx, I msg, ByteBuf out)接口即可完成编码的功能。encode 方法实现如下:

```
public void encode(final ByteBuf out, final Object message)
    throws IOException {
    try {
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
        closer.register(byteArrayOutputStream);
        KryoSerialize kryoSerialization = new KryoSerialize(pool);
        kryoSerialization.serialize(byteArrayOutputStream, message);
        byte[] body = byteArrayOutputStream.toByteArray();
        int dataLength = body.length;
        out.writeInt(dataLength);
        out.writeBytes(body);
    }
```

先创建一个 ByteArrayOutputStream 字节输出流, 利用 Kryo 对象池创建 Kryo 对象, 并调用 KryoSerialize 中的 serialize 序列化方法, 将 MessageRequest 对象序列化为字节流保存在 byteArrayOutputStream 输出流中。byteArrayOutputStream.toByteArray()方法将输出流中流数据转化为字节数组, 最后将字节数组的长度和字节数组内容写入到 ByteBuf 缓冲区中, 交由 NioEventLoop 线程进行处理。

#### (b) Kryo 解码

KryoSerialize 类中的 deserialize 方法实现了 Kryo 的反序列化过程, 通过 deserialize 方法完成字节流到对象的反序列化过程, 代码实现如下:

```
public Object deserialize(InputStream input) throws IOException {
    Kryo kryo = pool.borrow();
    Input in = new Input(input);
    Object result = kryo.readClassAndObject(in);
    in.close();
    input.close();
    pool.release(kryo);
    return result;
}
```

利用 borrow()函数从 Kryo 对象池获取到 Kryo 对象后, 调用 Kryo 的 readClassAndObject



方法将输入流反序列化为 Object 对象，然后释放所有资源，返回解码后的对象。

KryoEncoder 解码器间接继承 Netty 的 ByteToMessageDecoder 解码器，只需要实现 void decode(ChannelHandlerContext ctx ,ByteBuf in,List<Object> out)抽象方法就可以完成 ByteBuf 到 POJO 对象的解码。decode 方法实现如下：

```
public Object decode(byte[] body) throws IOException {
    try {
        ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(body);
        closer.register(byteArrayInputStream);
        KryoSerialize kryoSerialization = new KryoSerialize(pool);
        Object obj = kryoSerialization.deserialize(byteArrayInputStream);
        return obj;
    }
}
```

构建一个 ByteArrayInputStream 字节输入流，将 ByteBuf 中经过拆包处理的字节流保存在输入流中，调用 deserialize 方法将输入流中的二进制流反序列化为请求对象。

以上是 Kryo 的编码、解码过程，Hessian 的编解码过程与 Kryo 类似。它们都可以解决 TCP 消息在网络传输过程中产生的半包问题。通过使用第三方优秀的编解码框架可以解决 RPC 消息序列化性能问题，提高服务器的响应速度和效率。

## 4.3 通信调度模块

通信调度模块主要处理 I/O 事件，如 TCP 连接、读写监听和处理等操作。由于 RPC 服务器需要同时处理大量的 I/O 连接事件，因此 I/O 线程模型设计的好坏，影响服务器的可靠性和并发性能。

### 4.3.1 I/O 线程模型设计

RPC 服务器的通信调度模块使用 Netty 的主从线程模型来实现 I/O 事件的调度，Netty 的线程模型被精心设计，可以提升通信框架的并发处理性能<sup>[34]</sup>。RPC 服务器的线程模型如图 4.14 所示。

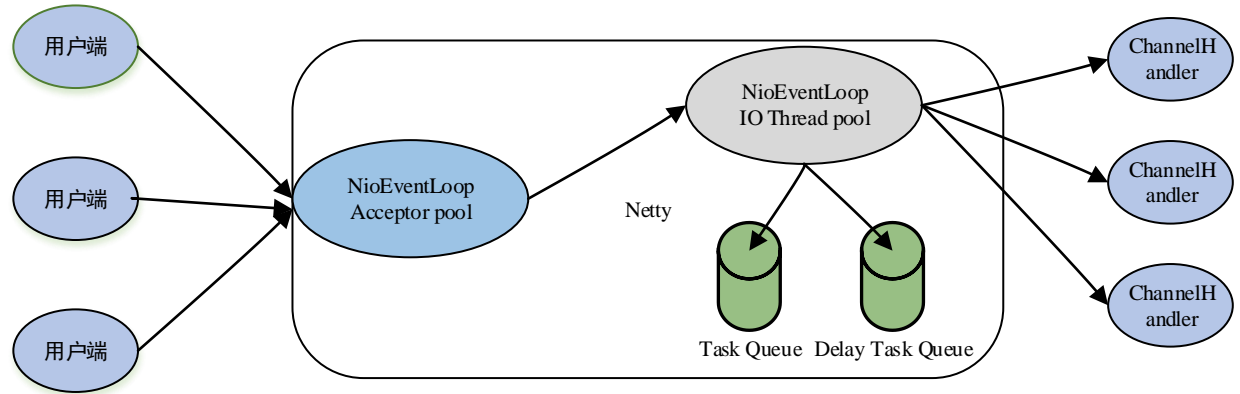


图 4.14 服务器线程模型

- (1) Acceptor 线程池的主要职责是处理连接请求，利用 NioEventLoop 线程为请求消息设置参数以及处理 TCP 连接；
- (2) NioEventLoop 线程异步读取客户端的请求消息，并将 read 事件传递到 Channel 中；
- (3) Channel 调用相应的方法将响应消息异步返回给客户端；
- (4) 执行系统调用 Task；
- (5) 执行定时任务。

4.3.2 服务端 I/O 处理

服务端采用网络事件监听线程和 I/O 线程分开工作的线程模型设计，两种线程各司其职，服务端线程的处理过程如图 4.15 所示。

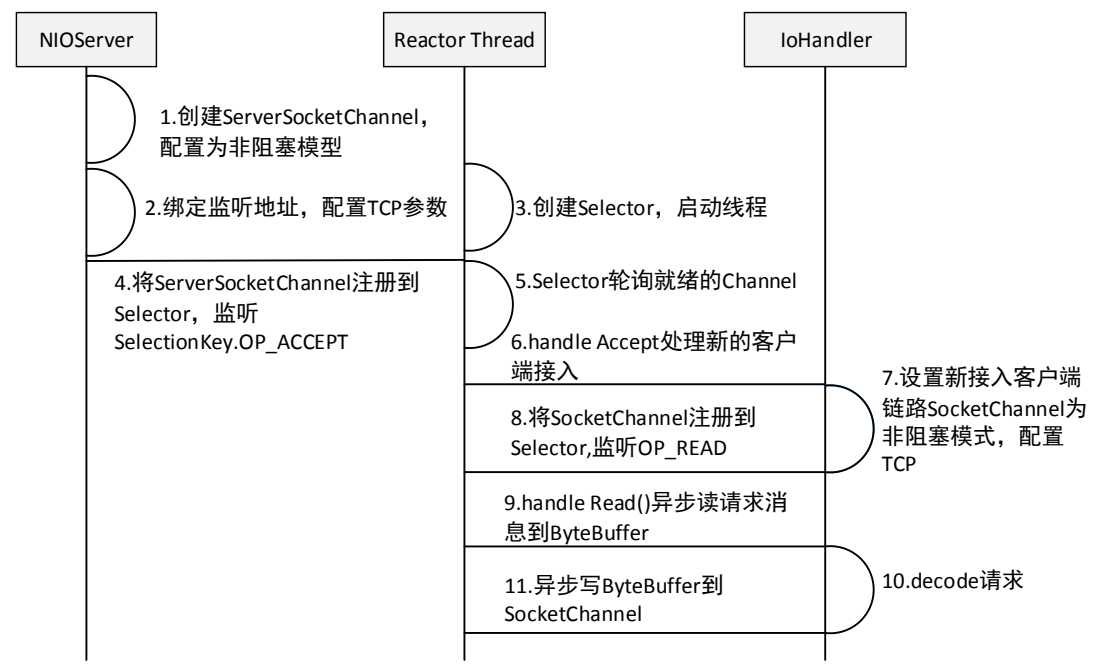


图 4.15 服务端的 I/O 处理流程

用户线程发起创建服务端的操作，用户线程启动的时候会创建服务端的线程，服务端一

般在整个系统初始化的时候创建。I/O 事件处理完毕，需要处理业务时，由业务线程来负责完成后续的工作。服务端 I/O 配置代码如下：

```
ServerBootstrap bootstrap = new ServerBootstrap();
bootstrap.group(boss, worker).channel(NioServerSocketChannel.class)
    .childHandler(new MessageRecvChannelInitializer(handlerMap).
        buildRpcSerializeProtocol(serializeProtocol))
    .option(ChannelOption.SO_BACKLOG, 128)
    .childOption(ChannelOption.SO_KEEPALIVE, true);
String[] ipAddr = serverAddress.split(MessageRecvExecutor.DELIMITER);
if (ipAddr.length == RpcSystemConfig.IPADDR_OPRT_ARRAY_LENGTH) {
    final String host = ipAddr[0];
    final int port = Integer.parseInt(ipAddr[1]);
    ChannelFuture future = null;
    future = bootstrap.bind(host, port).sync();
}
```

### (1) I/O 线程创建

在服务端创建的过程中，会实例化两个 Reactor 线程组，分别是主线程组 bossGroup 和工作线程组 workerGroup，它们主要管理 EventLoop 的创建和销毁，可以通过构造器设置线程组中初始化的线程数。

bossGroup 线程组是一个 Acceptor 线程池，它的作用是对客户端的 TCP 请求进行处理，如果服务端只监听一个端口，则可以将 bossGroup 初始化为单线程。workerGroup 线程组主要处理网络 I/O 读写操作。

### (2) 序列化方式选择

buildRpcSerializeProtocol 函数用来绑定 I/O 处理使用的编解码方式。RPC 服务器的支持 Kryo、Hessian 和 Java 原生序列化 3 种方式，根据使用场景选择合适的序列化方式。

### (3) ChannelHandler 注册到 ChannelPipeline

childHandler 函数用于将我们自定义的 ChannelHandler 添加到 ChannelPipeline 中。当服务端接收消息时，利用 MessageRecvChannelInitializer 类初始化 Channel，并在初始化 Channel 的过程中创建 ChannelPipeline。

消息编解码模块设计实现的 LengthFieldBasedFrameDecoder、LengthFieldPrepender 以及 KryoDecoder、KryoEncoder、HessianDecoder、HessianEncoder 等自定义的编解码器，都继承了 ChannelHandlerAdapter 基类，所有继承这个基类的 ChannelHandler 都可以对消息进行拦截。将这三个 ChannelHandler 的集合 handlerMap 注册到 ChannelPipeline 中，用来对 RPC 消息进行编码和解码。

### (4) 异步回调处理

ServerBootstrap 启动类调用 bind(host, port) 函数绑定对应客户端请求的 host 和 port 端口，

然后调用 `sync()`函数同步监听客户端的请求消息<sup>[35]</sup>，例如：请求连接、消息请求以及连接关闭的监听。最后，利用 Netty 的 `ChannelFuture` 类异步等待 `sync` 监听的结果。

服务端 I/O 处理时序图如图 4.16 所示。

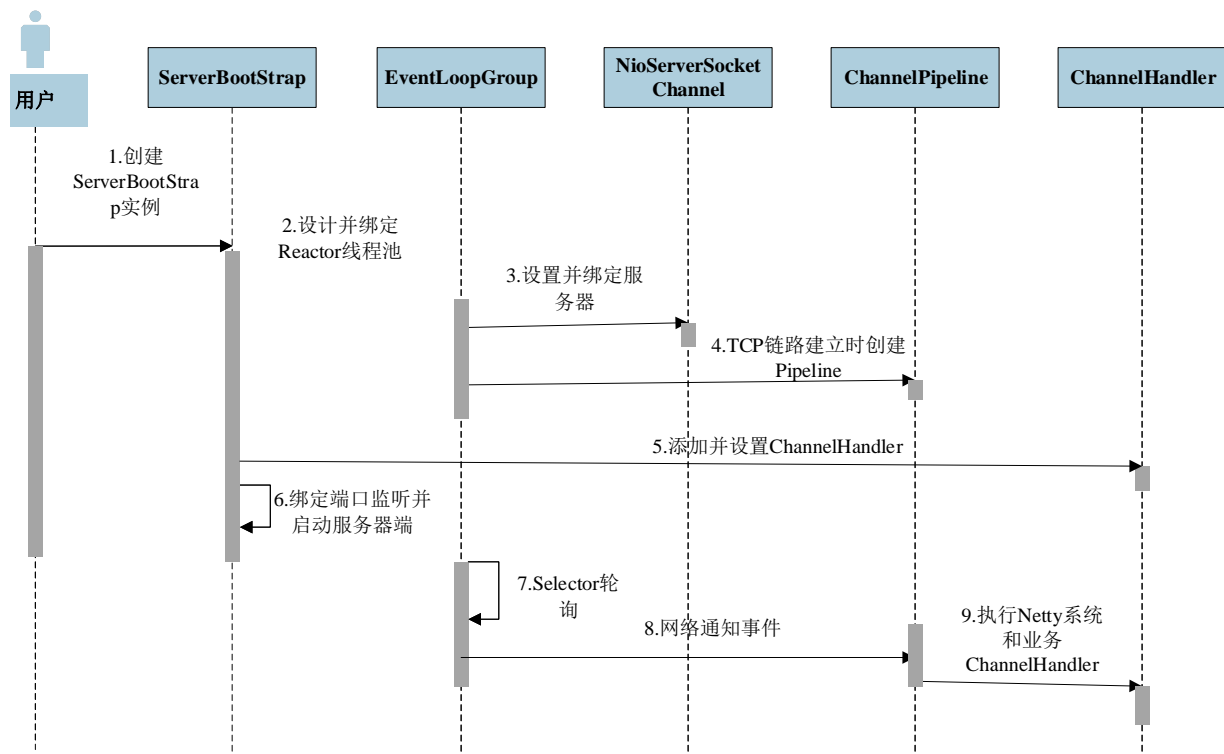


图 4.16 服务端 I/O 处理时序图

`ServerBootstrap` 是一个启动类，主要在 Netty 应用程序中负责初始化服务端。服务端 I/O 处理具体过程如下：

(1) 创建 `ServerSocketChannel` 服务端通道，并将它设置为非阻塞状态。`bossGroup` 线程绑定监听的端口，启动服务端，然后从线程组中选取一个 `Acceptor` 线程用来对服务端进行监听。

服务端 `Channel` 通道创建完成之后，将其注册到 `Selector` 多路复用器上，用来接收客户端的 `TCP` 连接。

(2)启动 I/O 线程，在 `for` 循环中执行 `Selector.select()`方法，同步轮询是否有就绪的 `Channel`。

(3)当 `select` 到就绪的 `Channel` 时，对它的状态位进行判断。如果状态位是 `OP_ACCEPT`，则是客户端连接，需要创建一个客户端 `SocketChannel` 连接，将它注册到 `workerGroup` 的 `IO` 线程上，然后调用 `NioMessageUnsafe` 的 `read()`方法用来处理读或者连接数据，最后调用 `doReadMessagesge` 方法来创建客户端连接的 `SocketChannel`，设置 `SocketChannel` 为非阻塞模式，从中选择一个 I/O 线程负责网络消息的读写。

(4)在选择好 I/O 线程之后，将 `SocketChannel` 注册到 `Selector` 上，监听网络读事件。

如果 Channel 的状态位为 OP\_READ, 则表明 SocketChannel 中有新的数据要读取, 创建 ByteBuf 对象, 将 SocketChannel 中的数据读取到 ByteBuf 中。

如果 Channel 的状态位为 OP\_WRITE, 表明 SocketChannel 中的数据未传输完毕, 继续执行传输过程。

(5) 将请求消息提交给绑定在当前 Channel 中的 ChannelPipeline, 消息会经 ChannelPipeline 中的一系列 ChannelHandler 对消息进行处理<sup>[36]</sup>。ChannelHandler 处理流程如图 4.17 所示。

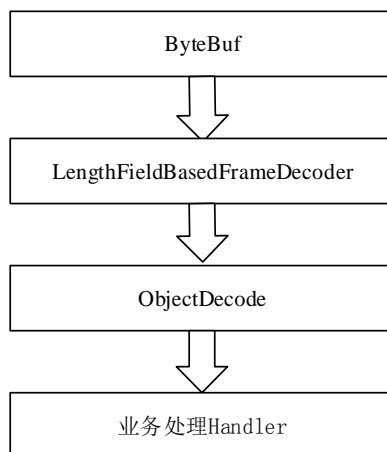


图 4.17 服务端解码流程

(6) 经过 ChannelPipeline 处理后, RPC 消息解码为 MessageRequest 对象。I/O 操作完成之后, workerGroup 线程会回调 ChannelFutureListener 的 operationComplete 方法, 将该对象保存到 ChannelFuture 中给上层业务处理模块处理。

(7) 上层业务模块处理完成之后, 将响应对象 MessageResponse 保存在 ChannelFuture 中, 供 I/O 线程处理。

服务端启动成功, 并接收到客户端的请求数据, 运行结果如图 4.18 所示。

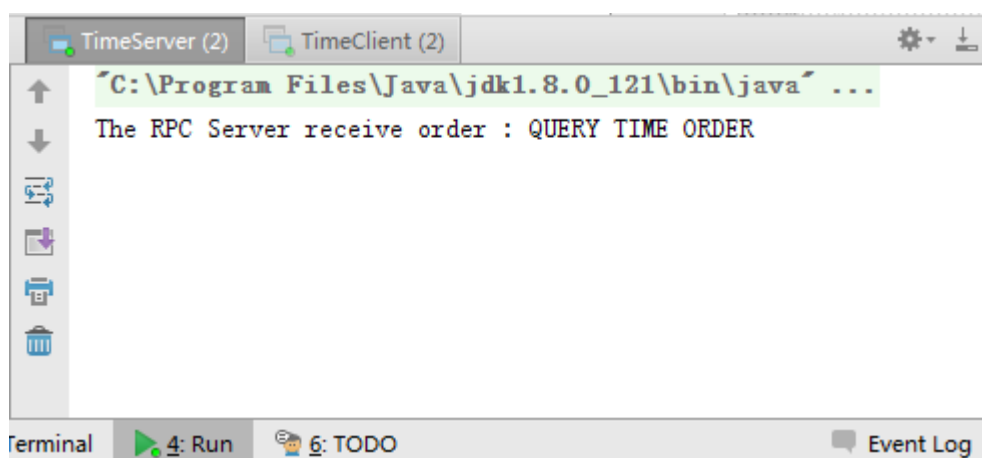


图 4.18 服务端启动运行结果

4.3.3 客户端 I/O 处理

客户端的线程模型相比于服务端线程模型更加简单，具体工作原理如图 4.19 所示。

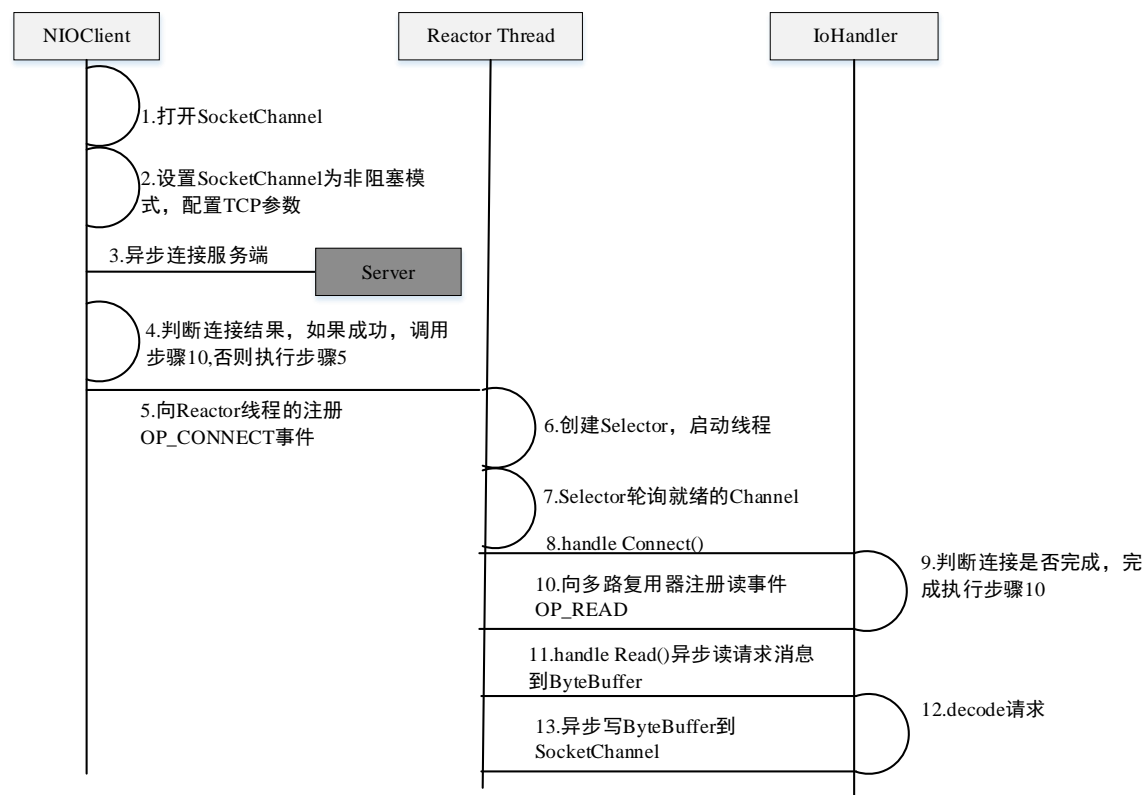


图 4.19 客户端的 I/O 处理流程

客户端只需要处理 I/O 事件，不用监听其他的客户端连接，因此只需要创建了一个 NioEventLoop 线程组，这是它与服务端最大的区别。而且 Netty 的连接和 I/O 操作都是异步的，不需要单独创建连接线程。客户端配置代码如下：

```
Bootstrap b = new Bootstrap();
b.group(eventLoopGroup)
    .channel(NioSocketChannel.class)
    .option(ChannelOption.SO_KEEPALIVE, true)
    .remoteAddress(serverAddress);
b.handler(new MessageSendChannelInitializer().
    buildRpcSerializeProtocol(protocol));
ChannelFuture channelFuture = b.connect();
```

(1) 客户端 I/O 创建

客户端与服务端不同，它不需要独立的线程去监听客户端连接，只需要创建一个 EventLoopGroup 线程组来处理服务器发起的远程 TCP 连接以及监听返回的处理结果。

(2) 响应消息处理

MessageSendChannelInitialize 类与服务端的 MessageRecvChannelInitializer 功能类似，在

接收到响应消息时初始化 Channel，并创建 ChannelPipeline 职责链处理响应的消息，对响应消息进行解码工作。

(3) 远程调用

remoteAddress()函数绑定远程服务器 IP 地址，进行远程请求调用。通过负载均衡算法，计算得到一台负载合适的服务器 IP 地址进行远程调用。

(4) 异步回调处理

客户端将 RPC 请求消息通过网络发送到远程服务端后，立即返回<sup>[37]</sup>。利用 ChannelFuture 类异步等待消息的响应，ChannelFuture 监听事件响应结果，响应成功则返回响应消息，失败则构造成定时任务处理<sup>[38]</sup>。

客户端 I/O 处理时序图如图 4.20 所示。

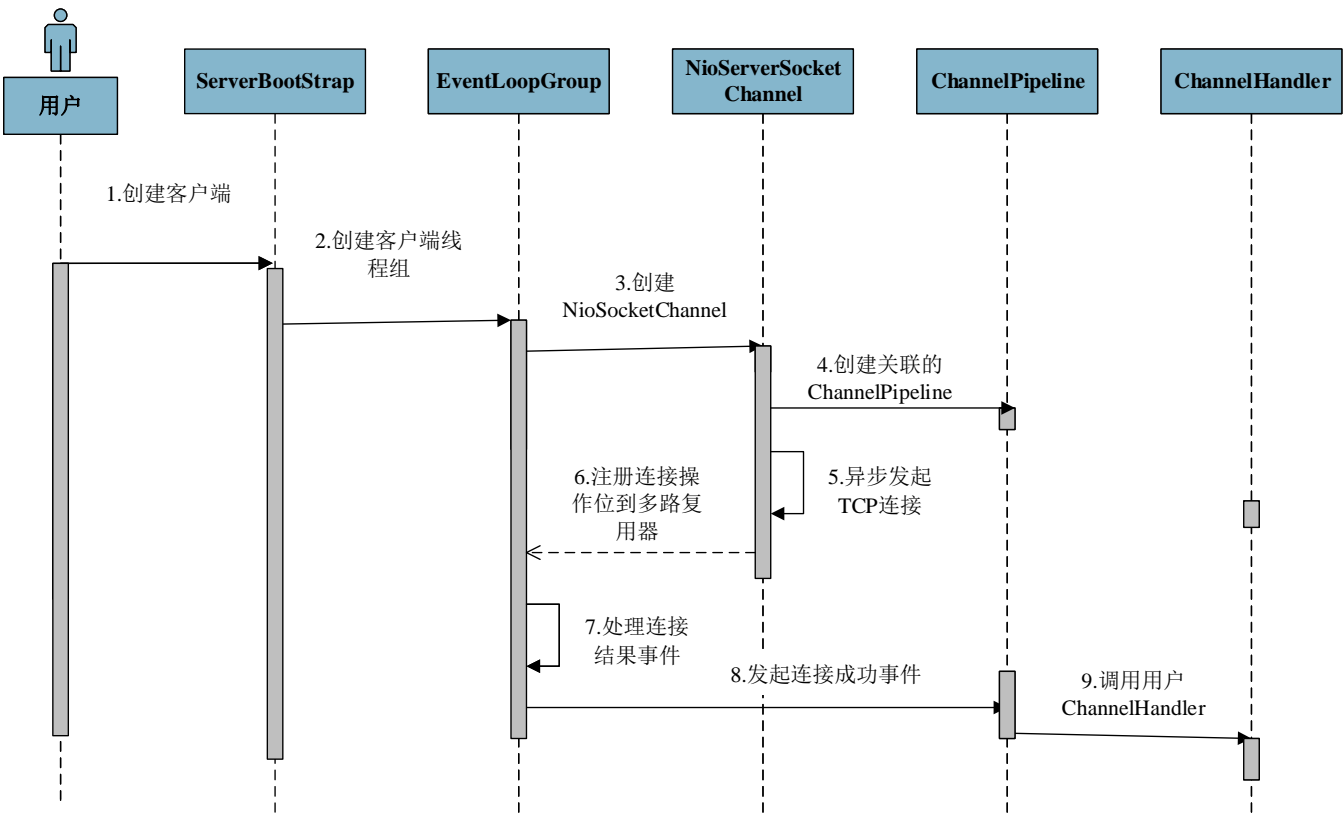


图 4.20 客户端 I/O 处理时序图

Bootstrap 是客户端启动类，通过设置相关参数，在服务器启动时初始化客户端，异步发起连接。下面客户端 I/O 处理的过程：

(1) 创建用于处理客户端 I/O 读写的 NioEventLoopGroup 线程组，同时创建客户端辅助启动类 Bootstrap，进行相关配置。

(2) 通过 NioEventLoopGroup 线程组获取可用的 I/O 线程 EventLoop，利用它来创建新的 Channel，并设置为 NioSocketChannel，为其添加业务处理所需的 ChannelHandler。



(3) 调用 Bootstrap 的 doConnect()方法发起 TCP 连接操作, 通过 NioEventLoop 中的多路复用器 Selector 轮询各个 Channel 接连的结果, 对就绪的 Channel 进行处理。

(4) 没有连接成功, 则将 SocketChannel 注册到某个 NioEventLoop 线程中, 监视连接位, 根据连接位状态进行后续操作。

(5) 如果连接成功, 将 SocketChannel 注册到某个 NioEventLoop 线程中, 修改监听位为 READ, 监听读操作位, 此时不需要切换线程。触发 ChannelPipeline 执行连接成功事件。

(6) ChannelPipeline 调度系统和用户的 ChannelHandler, 执行业务逻辑。ChannelPipeline 处理过程如图 4.21 所示:

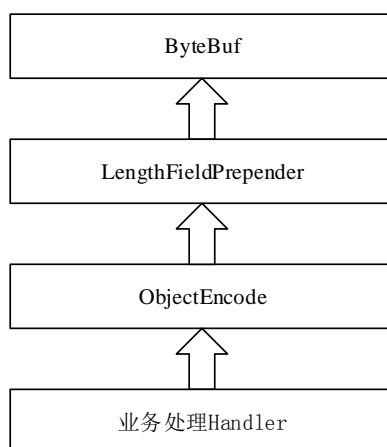


图 4.21 客户端编码流程

(7) 客户端链路关闭后, 优雅退出, 释放 NioEventLoop 线程组。

客户端与服务端连接成功后, 客户端运行情况如图 4.22 所示。

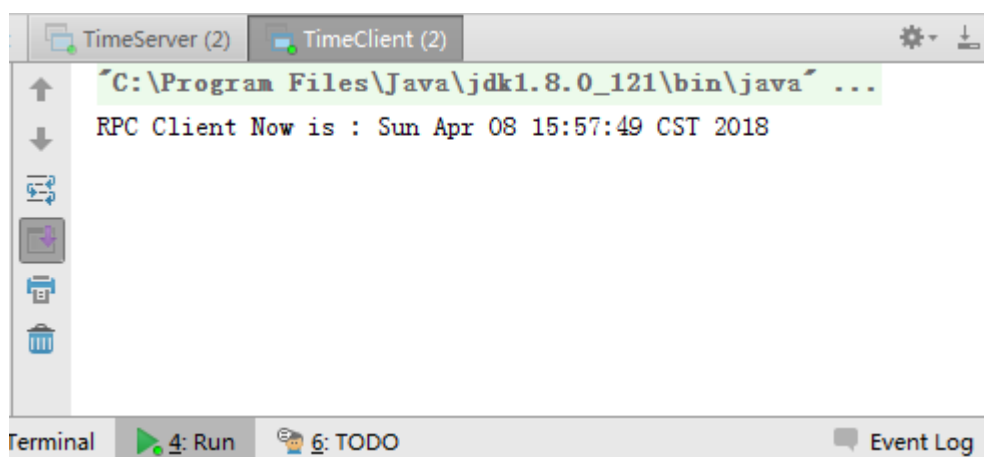


图 4.22 客户端启动运行结果

#### 4.3.4 时间轮和定时任务处理

##### (1) 时间轮算法

Netty 是一个高性能异步通信框架, 它可以处理 100w+ 的连接, 一个连接上有很多定时任



务需要处理，如心跳检测、I/O 读写监听等。传统做法是利用 Timer 对每一个任务进行包装，根据 Timer 触发任务执行，这种方式会创建大量的 Timer 类，占用大量的内存资源。Netty 通过时间轮算法来处理大量的定时任务<sup>[39]</sup>。

时间轮与 Java 的环形链表结构类似，可以类比于时钟，将环形分成多个格子，格子越小精度越高。利用一个指针指向格子中的到期任务，指针顺着环形格子依次转动，当指向到期任务的格子时，即执行对应的任务。通过取模的方式，可以将任务添加到对应的格子中。时间轮算法如图 4.23 所示。

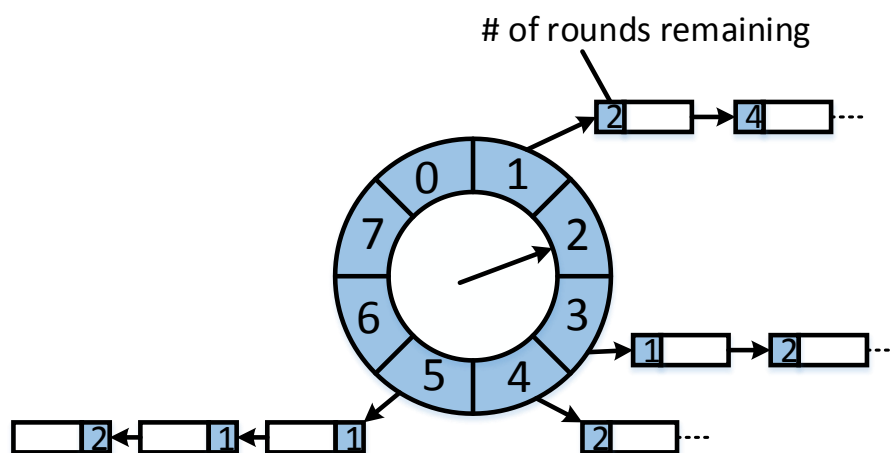


图 4.23 时间轮算法调度模型

如图 4.26，假设时间轮上的每一个格子代表 1s，整个时间轮所表示的时间为 8s。当前指针指向 2，表示正在执行第 2 个格子中的任务。假如要处理一个 4s 后的定时任务，则指针需要顺时针转动 4 格，当指向 6 所对应格子时，执行格子中的所有任务。若需要执行一个 11s 后的任务，则指针要转动一圈，并继续走 (round(11)=3) 格，也就是 5 所对应的格子。检查时间轮上的到期任务时，只是处理 round 为 0 的任务，其它任务等待时间减 1。

时间轮算法与 HashMap 的哈希拉链算法计算过程类似，只是增加了指针轮转和任务处理的过程。其相关的操作和 HashMap 也一致：

(a) 添加任务时间复杂度： $O(1)$ ；

(b) 删除/取消任务事件复杂度： $O(1)$ ；

(c) 执行任务，最坏的情况是所有的任务都保存在同一个格子，组成一个链表，查找的时间复杂度为  $O(n)$ 。随着格子的增加，格子上的链表长度越短，处理速度越快，查找的平均时间复杂度为  $O(1)$ 。

## (2) 定时任务处理

NioEventLoop 线程除了负责处理 I/O 事件之外，还需要处理定时任务。I/O 任务和定时任务的时间分配比例，由用户根据实际业务场景设置。默认情况下，两者各占一半 CPU 时间。

Netty 使用一个任务队列保存需要执行的定时任务，NioEventLoop 线程调用 peek()方法从队列中获取要执行的任务，若队列中没有要处理的任务，直接退出循环即可。若任务超时，则将其添加到 Task Queue 任务队列中，供线程池执行。若当前任务没有超时，则不做任何处理，直接退出。代码实现如下：

```
private void fetchFromDelayedQueue() {  
    long nanoTime = 0L;  
    for (;;) {  
        ScheduledFutureTask<?> delayedTask = delayedTaskQueue.peek();  
        if (delayedTask == null) {  
            break;  
        }  
  
        if (nanoTime == 0L) {  
            nanoTime = ScheduledFutureTask.nanoTime();  
        }  
  
        if (delayedTask.deadlineNanos() <= nanoTime) {  
            delayedTaskQueue.remove();  
            taskQueue.add(delayedTask);  
        }  
    }  
}
```

通过 nanoTime()函数获取系统当前时间的操作比较耗时，如果任务队列中有大量的任务，每次都获取当前时间进行判断，这样会浪费系统大量的资源。可以设置每 60 次循环，就判断一次超时，如果定时任务处理时间超时，直接退出。通过这种方式避免 I/O 处理时间分配过少，导致 I/O 操作被大量阻塞。

定时任务执行完毕，判断当前系统是否需要关闭，如果需要关闭，则释放所有资源。遍历系统中所有的 Channel，调用绑定在 Channel 中 Unsafe 类，将资源释放，并进入停机状态。

## 4.4 服务发布和订阅

RPC 服务器在整个系统架构中，既可能作为一个服务提供者对外提供服务，同时也可能是一个服务订阅者向其他远程服务器请求服务。本小节主要研究 RPC 服务器发布和订阅服务的过程。

### 4.4.1 服务发布过程

#### (1) 服务端发布服务

RPC 服务器通过 Spring 容器发布服务，将 SpringBean 配置在 Spring 配置文件中，在 Spring 容器启动的时候，将服务初始化并发布到网络中。下面是远程 AddCalculate 加法运算服务的

配置代码:

```
<nettyrpc:service id="AddCalculate"
    interfaceName="com.newlandframework.rpc.services.AddCalculate"
    ref="calcAddService"></nettyrpc:service>
<bean id="calcAddService"
    class="com.newlandframework.rpc.services.impl.AddCalculateImpl">
</bean>
```

(a) nettyrpc:service: 定义 rpc 服务器提供的服务接口。

(b) interfaceName: 接口名称, 定义提供远程服务的接口名称。

(c) ref: 表示真正调用的服务, 它指向远程调用接口的实现类, 是真正提供服务调用的具体类, 这里是 AddCalculateImpl。

(d) bean: 在 Spring 容器启动时, 将 bean 标签配置的具体类加载到容器中, 进行初始化。

(e) id: 表示接口实现类在 Spring 容器中对应的唯一标识号, 可以根据 id 号获取该类的实例。

(f) class: Spring 容器根据全类名将对应的类加载到容器中。

## (2) 服务发布过程

服务端启动时, Spring 容器进行初始化工作, 将 ProviderBean 对象通过网络暴露给消费者使用<sup>[40]</sup>。发布服务的流程如图 4.24 所示。

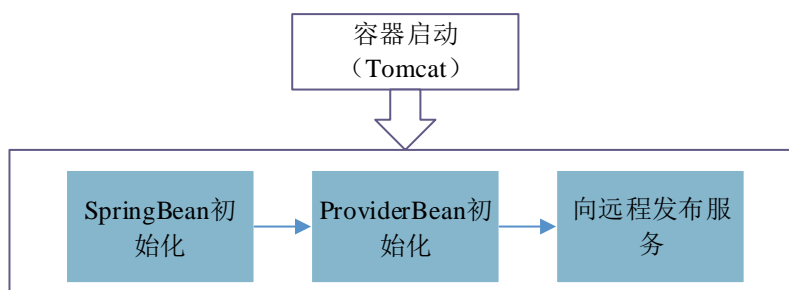


图 4.24 服务发布过程

## 4.4.2 服务订阅过程

### (1) 客户端订阅服务

客户端通过 Spring 配置文件配置需要请求的服务接口, 在服务器启动的时候, 向远程服务器订阅服务。下面是 AddCalculate 服务的订阅配置:

```
<nettyrpc:reference id="addCalc"
    interfaceName="com.newlandframework.rpc.services.AddCalculate"
    protocol="PROTOSTUFFSERIALIZE"
    ipAddr="${rpc.server.addr}"/>
```

- (a) nettyrpc:reference: 用来订阅远程服务。
- (b) interfaceName: 请求的服务接口名称, 客户端根据接口全类名来订阅远程服务。
- (c) protocol: 声明当前 RPC 服务器支持的消息序列化协议。
- (d) ipAddr: 指定提供远程服务的 RPC 服务器 IP 地址。

## (2) 服务订阅过程

通过 Spring 配置文件对订阅的服务进行配置, 启动客户端程序, Spring 容器进行初始化, 将 ConsumerBean 对象封装为请求消息向远程服务器调用服务。订阅服务的流程如图 4.25 所示。

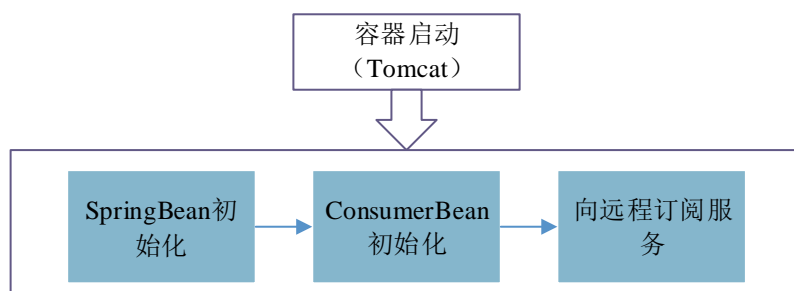


图 4.25 服务订阅过程

## 4.5 业务处理模块

业务处理模块主要并发处理远程调用的 RPC 请求消息以及向远程服务器请求服务, 并发处理的核心是线程池, 通过线程池的线程调度来并发处理大量的 RPC 请求。

### 4.5.1 线程池设计

RPC 服务器需要对远程 RPC 请求进行执行处理, 将大量高并发的远程 RPC 请求消息封装成可执行的 Task 任务, 丢给业务线程池来处理。通过线程池对大量并发请求的处理, 来实现服务器的高并发处理能力<sup>[41]</sup>。

线程池的调度原理如图 4.26 所示。

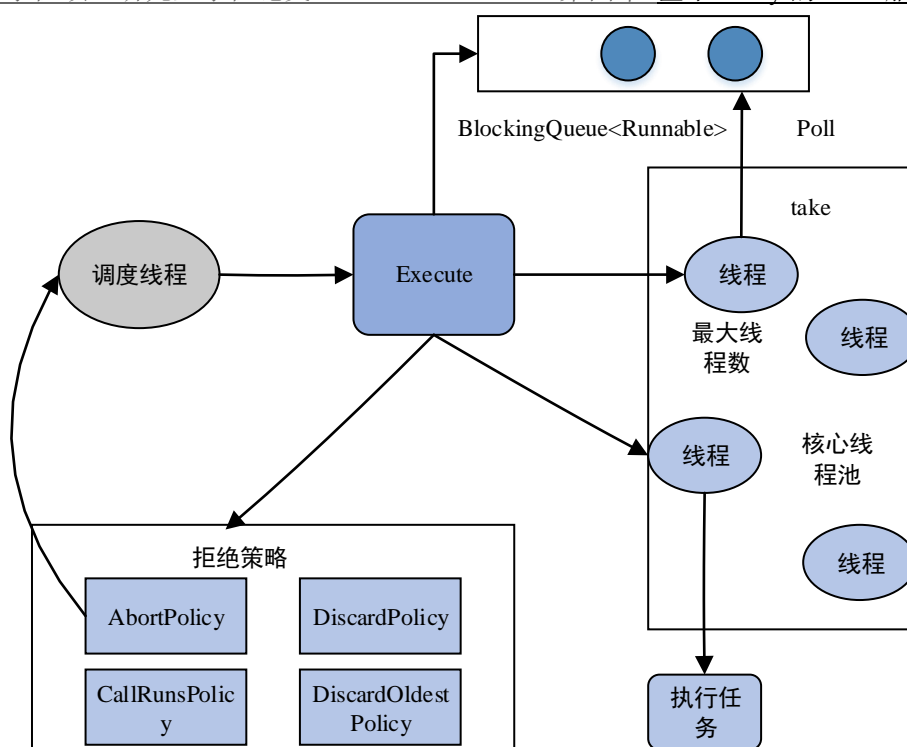


图 4.26 线程池原理图

本文基于 `ThreadPoolExecutor` 来实现核心业务线程池，用来处理业务请求，核心代码如下：

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(threads, threads, 0,
    TimeUnit.MILLISECONDS,
    createBlockingQueue(queues),
    new NamedThreadFactory(name, true),
    createPolicy());
```

业务线程池的定制策略是：创建一个固定线程数的线程池，它的阻塞队列是可选的，根据不同的业务需求，使用对应的阻塞队列<sup>[42]</sup>。`NamedThreadFactory` 是线程工厂类用来创建线程，`createPolicy()`方法用来获取线程拒绝策略 `AbortPolicy`。

#### （1）线程工厂 `NamedThreadFactory`

`NamedThreadFactory` 是对 `ThreadFactory` 的实现，实际上是对 `Runnable` 进行包装以及对线程设置一些信息，如线程名、是否是后台线程以及属于哪个线程组。

#### （2）线程池的拒绝策略

`createPolicy()`函数指定线程拒绝策略，当遇到线程池也无法处理的情况的时候<sup>[43]</sup>，RPC 服务器的线程池模型具体的应对措施是：

- （a）`AbortPolicy`：直接拒绝执行，抛出 `rejectedExecution` 异常。
- （b）`DiscardedPolicy`：丢弃当前任务，以减轻线程池压力。
- （c）`CallerRunsPolicy`：利用当前调用线程执行任务，避免频繁线程切换带来的性能问题。

(3) 任务队列 workQueue

workQueue 是一个存储等待执行任务的阻塞队列，在 RPC 服务器中用于保存客户端提交的请求任务 Task，等待线程池的执行。createBlockingQueue(queues)方法用来创建任务队列。

线程池的阻塞队列有下面三种可选：

- (a) **LinkedBlockingQueue**：底层结构使用链表实现，是一个 FIFO（先进先出）的阻塞队列，从链表的底部添加元素，从顶部获取元素。
- (b) **ArrayBlockingQueue**：基于数组结构的有界队列，按照 FIFO 原则对元素进行存储。
- (c) **SynchronousQueue**：一个没有任何容量的阻塞队列，对于每一个提交过来的任务，只有其他线程 take 拿走这个任务，才能继续添加，否则会一直阻塞在该队列上。SynchronousQueue 的性能比 LinkedBlockingQueue 和 ArrayBlockingQueue 都要高。

RPC 服务器默认将核心线程数 corePoolSize 和最大线程数 maximumPoolSize 设置为 16，阻塞队列选择 LinkedBlockingQueue。在实际应用中根据服务器的压力、吞吐量，对线程池参数进行合理的配置，充分利用线程池的性能，避免资源的浪费。

4.5.2 RPC 消息处理过程

RPC 请求对象提交到线程池后，由线程池调度处理，线程池处理 RPC 消息的流程如图 4.27 所示。

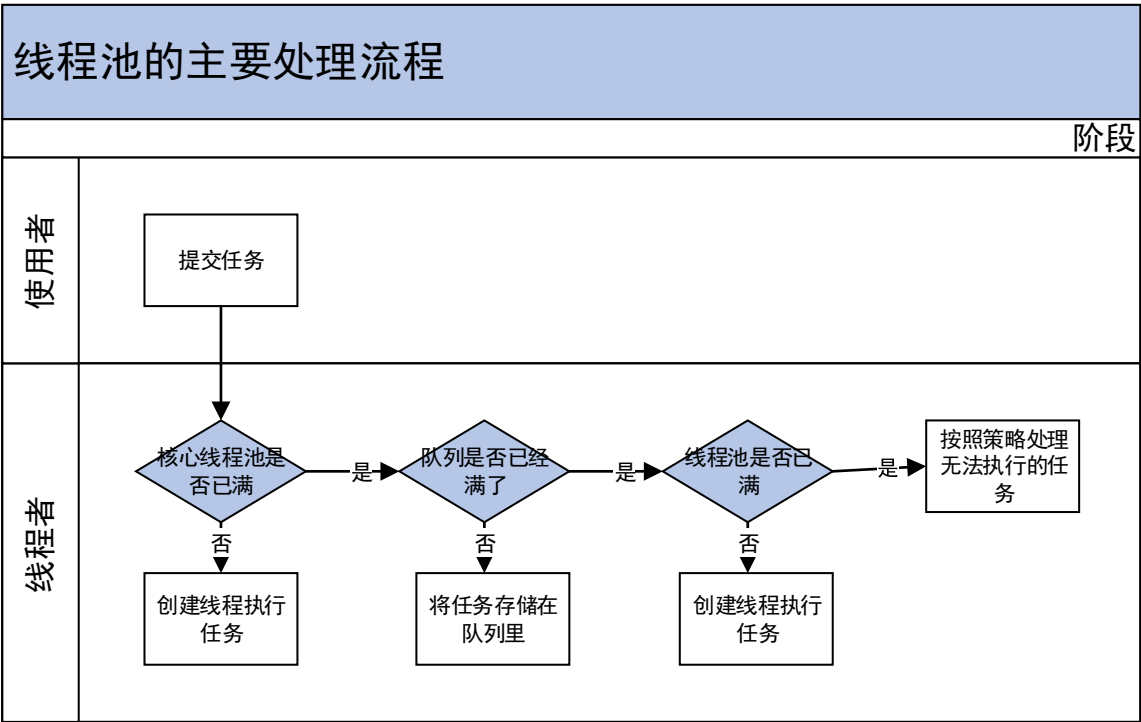


图 4.27 RPC 消息在线程池中的执行流程

- (1) 将 RPC 请求消息封装为 Runnable 任务, 提交到业务线程池;
- (2) 首先判断核心线程池的线程数是否已达到 corePoolSize 的值, 如果是, 则执行 3, 否则创建一个新的线程执行任务;
- (3) 判断阻塞队列是否已满, 如果是, 则执行 4, 否则将任务添加到阻塞队列中;
- (4) 判断线程池线程数是否达到 maximumPoolSize 的值, 如果是, 则选择合适的拒绝策略执行, 否则创建一个新的线程执行任务。

## 4.6 服务端调用处理

RPC 服务器的服务端主要处理远程调用的 RPC 请求消息, 根据请求的类名、方法以及参数, 执行对应的服务, 将请求结果返回给服务调用者。

### (1) 服务初始化

在 RPC 服务器启动的时候, 加载服务端的 Spring 配置文件, Spring 会将配置文件中配置的所有服务加载到 Spring 容器中, 对容器中的服务接口及其对应的实现类进行初始化工作。初始化完成后将服务保存在 MessageKeyVal 自定义容器中。

MessageRecvExecutor 是服务端的执行器, 它实现了 Spring 的 ApplicationContextAware 接口, 在 Spring 容器加载的时候, 调用它实现的 setApplicationContext 方法, 代码实现如下:

```
public void setApplicationContext(ApplicationContext ctx)
    throws BeansException {
    try {
        MessageKeyVal keyVal = (MessageKeyVal) ctx.getBean(
            Class.forName("com.newlandframework.rpc.model.MessageKeyVal"));
        Map<String, Object> rpcServiceObject = keyVal.getMessageKeyVal();

        Set s = rpcServiceObject.entrySet();
        Iterator<Map.Entry<String, Object>> it = s.iterator();
        Map.Entry<String, Object> entry;

        while (it.hasNext()) {
            entry = it.next();
            handlerMap.put(entry.getKey(), entry.getValue());
        }
    }
}
```

获取到 MessageKeyVal 容器的实例, 过滤掉容器中重复的服务, 然后对容器中的所有服务进行迭代, 以服务名为 key, 服务的实例为 val, 将所有服务都保存在 handlerMap 中, 等待 Channel 中的业务处理 ChannelHandler 处理。

### (2) 服务端 I/O 线程启动



RPC 服务器启动后, 对服务端 I/O 线程进行初始化, 对网络中的 I/O 事件进行监听。例如 TCP 连接、消息接收、心跳检测和链路关闭等操作<sup>[44]</sup>。MessageRecvExecutor 实现了 Spring 的 afterPropertiesSet()方法, 在 Spring 容器初始化后, 执行 afterPropertiesSet()方法启动服务端 I/O。afterPropertiesSet 方法核心代码如下:

```
ThreadFactory threadRpcFactory = new NamedThreadFactory("NettyRPC ThreadFactory");
EventLoopGroup boss = new NioEventLoopGroup();
EventLoopGroup worker = new NioEventLoopGroup(PARALLEL,
    threadRpcFactory, SelectorProvider.provider());

try {
    ServerBootstrap bootstrap = new ServerBootstrap();
    bootstrap.group(boss, worker).channel(NioServerSocketChannel.class)
        .childHandler(new MessageRecvChannelInitializer(handlerMap).
            buildRpcSerializeProtocol(serializeProtocol))
        .option(ChannelOption.SO_BACKLOG, 128)
        .childOption(ChannelOption.SO_KEEPALIVE, true);
    String[] ipAddr = serverAddress.split(MessageRecvExecutor.DELIMITER);
    if (ipAddr.length == RpcSystemConfig.IPADDR_OPRT_ARRAY_LENGTH) {
        final String host = ipAddr[0];
        final int port = Integer.parseInt(ipAddr[1]);
        ChannelFuture future = null;
        future = bootstrap.bind(host, port).sync();
    }
}
```

(a) 利用线程池工厂类 ThreadFactory 将 Netty 的线程池模型设置成主从线程模型, 以应对高并发请求。为了保证线程的合理数目, 设置 I/O 线程数为 Java 虚拟机的可用的处理器数的两倍, 避免 I/O 线程浪费。

(b) try 中代码块是服务端 I/O 配置代码, 启动服务端 I/O 后, 根据请求的 IP 地址以及端口号, 通过 bootstrap.bind(host, port).sync()方法, 异步监听和处理网络中的 I/O 事件。

### (3) 消息任务处理

当服务器的 I/O 调度模块接收到 RPC 请求消息后, 经过一系列的解码过程, 重新变成 MessageRequest 对象传递到业务处理的 ChannelHandler 中, 进行服务的调用<sup>[45]</sup>。为了能大量并发处理服务请求, 需要将每一个 RPC 请求消息封装成一个可执行的 Task 任务, 提交到线程池去处理。

服务端利用 MessageRecvInitializeTask 对 RPC 消息进行封装, 它实现了 Callable 接口, 在 call 方法中实现服务的调用执行, call 方法是真正的远程请求调用执行的地方。代码如下:



```
public Boolean call() {  
    try {  
        acquire();  
        response.setMessageId(request.getMessageId());  
        injectInvoke();  
        Object result = reflect(request);  
    }  
}
```

(a) 构建一个 MessageResponse 对象, 将它的 MessageId 设置为请求对象 MessageRequest 的 MessageId, 以便请求调用结束后, 可以根据 MessageId 将对应的调用结果设置到对应的 response 响应中。

(b) 根据 request 请求信息, 调用 reflect 方法, 执行服务的调用。reflect 方法调用服务是运用动态代理技术实现的。代码如下:

```
private Object reflect(MessageRequest request) throws Throwable {  
    ProxyFactory weaver = new ProxyFactory(new MethodInvoker());  
    NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor();  
    MethodInvoker mi = (MethodInvoker) weaver.getProxy();  
    Object obj = mi.invoke(request);  
    invokeTimespan = mi.getInvokeTimespan();  
    setReturnNotNull(((MethodProxyAdvisor) advisor.getAdvice()).isReturnNotNull());  
    return obj;  
}
```

调用结束后, 得到响应结果 result 对象, 根据 result 判调用执行是否成功。成功则设置响应对象, 否则打印异常信息。

#### (4) 动态代理拦截

在执行接收的远程调用请求之前, 利用 JDK 动态代理对每一个 RPC 请求消息进行拦截。服务端的 MethodProxyAdvisor 类实现了 MethodInterceptor 接口, 通过实现 Object invoke(MethodInvocation invocation) 方法, 对拦截的消息进行处理<sup>[46]</sup>。

MessageRequest 请求被拦截后, 根据请求消息, 获取到封装在 MessageRequest 中请求的服务名 serviceBean、服务方法名 methodName 以及方法的参数列表 parameters, 将请求的服务名保存在 MethodInvoker 中。在 invocation.proceed() 来执行服务调用时, 会转而去执行 MethodInvoker 的 invoke 方法 (reflect 方法实现服务的调用), 根据请求服务信息执行真正的服务调用, 代码如下:

```
public Object invoke(MessageRequest request) throws Throwable {  
    String methodName = request.getMethodName();  
    Object[] parameters = request.getParametersVal();  
    Object result = MethodUtils.invokeMethod(serviceBean, methodName, parameters);  
    return result;  
}
```

根据 Java 反射原理, 在 Java 虚拟机运行时, 根据服务名、服务方法以及参数找到对应的

服务, 执行服务调用, 返回调用结果。

### (5) 业务处理 Handler

服务端 I/O 模块接收到的消息, 经过 ChannelPipeline 处理链中的一系列解码器之后, 最后会到达 MessageRecvHandler, 这个 Handler 是业务处理的核心, 它处于 ChannelPipeline 链的最顶端, 用于业务的执行处理。MessageRecvHandler 继承了 ChannelInboundHandlerAdapter, 通过实现 channelRead 方法来处理 Channel 中的读事件。相关代码如下:

```
public void channelRead(ChannelHandlerContext ctx, Object msg)
    throws Exception {
    MessageRequest request = (MessageRequest) msg;
    MessageResponse response = new MessageResponse();
    RecvInitializeTaskFacade facade = new
        RecvInitializeTaskFacade(request, response, handlerMap);
    Callable<Boolean> recvTask = facade.getTask();
    MessageRecvExecutor.submit(recvTask, ctx, request, response);
}
```

channelRead 函数读取到的 msg 已经经过解码处理, 可以直接转化为 MessageRequest 对象。在当前 I/O 线程中, 为了避免在处理请求消息的过程中出现耗时和阻塞线程的情况, 对于每一个 RPC 请求消息, 都创建一个响应对象, 将请求对象、响应对象以及 handlerMap 都封装成 MessageRecvInitializeTask 任务, 然后交给业务线程池去处理。getTask() 方法用来封装成 MessageRecvInitializeTask 可执行的任务。

### (6) RPC 服务器并发处理能力

请求消息封装成可执行的 MessageRecvInitializeTask 任务后, 调用 MessageRecvExecutor 启动器的 submit 方法, 将任务提交到消息处理线程池中进行处理。

利用 synchronized 保证当前只有一个线程处理 MessageRecvExecutor, 通过这种无锁化方式保证并发的安全性。

listenableFuture 接口扩展了 Future 接口, 增加了 addListener 方法, 该方法在给定的 excutor 上注册一个监听器, 当调用执行完成时会马上调用该监听器。它不能够确保监听器执行的顺序, 但可以在计算完成时确保马上被调用, 用于接收线程池执行完成后的异步结果。

FutureCallback 用于执行后的回调处理。执行成功, 则会执行 onSuccess 方法, 调用 writeAndFlush 方法将响应消息 response 写入到 Channel 中的上下文中, 供 ChannelPipeline 处理链处理。

将线程池的 listenableFuture 和 FutureCallback 添加到 Futures 中用于处理异步回调过程。请求执行成功后, 得到异步响应结果, 将响应对象封装为 MessageResponse 对象, 通过服务端 I/O 处理模块将响应消息返回给服务请求方<sup>[47]</sup>。

## 4.7 客户端调用处理

客户端消息处理主要有两个方面，将请求服务信息封装为 `MessageRequest` 进行远程服务调用请求，以及调用成功异步接收远程调用结果。

### 4.7.1 服务订阅

#### (1) 动态代理拦截

服务器通过客户端的 `Spring` 配置文件订阅远程服务，在 `Spring` 容器启动时，这些服务对象会被加载到 `Spring` 容器中，通过 `JDK` 动态代理技术，将所有需要订阅的服务拦截<sup>[48]</sup>。

客户端的 `MessageSendProxy` 类间接实现了 `InvocationHandler` 接口，它会对所有订阅的服务进行拦截。服务被 `MessageSendProxy` 代理类拦截后，调用实现的 `invoke` 方法对拦截的请求对象进行处理。`MessageSendProxy` 的 `invoke` 方法实现如下：

```
public Object handleInvocation(Object proxy, Method method, Object[] args)
    throws Throwable {
    MessageRequest request = new MessageRequest();
    request.setMessageId(UUID.randomUUID().toString());
    request.setClassName(method.getDeclaringClass().getName());
    request.setMethodName(method.getName());
    request.setTypeParameters(method.getParameterTypes());
    request.setParametersVal(args);
    MessageSendHandler handler = RpcServerLoader.getInstance().getMessageSendHandler();
    MessageCallBack callBack = handler.sendRequest(request);
    return callBack.start();
}
```

每个订阅的服务被拦截后，都会构建一个 `MessageRequest` 请求对象，每个服务对应一个唯一标识的 `messageId`。将远程请求服务的类名、方法名以及参数列表设置到 `MessageRequest` 中。服务被包装为 `MessageRequest` 请求对象后，调用 `MessageSendHandler` 的 `sendRequest` 方法发送请求消息。

#### (2) 负载均衡

在客户端远程调用服务的时候，由于系统一般采用集群的形式暴露服务，可能有多个服务器提供相同的服务<sup>[49]</sup>。集群中的每台服务器由于配置不同，负载也不同，有些服务器并发请求数很多，负载很大<sup>[50]</sup>。有些服务器虽然负载比较少，但是可能由于机器老旧，处理请求速度很慢。客户端根据负载均衡算法，寻找一台压力较小的服务器进行请求调用，来保证处理速度<sup>[51]</sup>。

本文通过加权轮询法来进行负载均衡，代码实现如下：

```
public String getServer(Integer pos){
    Map<String, Integer> serverMap = new HashMap<>();
    serverMap.putAll(IpMap.serverWeigthMap);
    Set keySet = serverMap.keySet();
    Iterator iterator = keySet.iterator();
    List serverList = new ArrayList();
    while (iterator.hasNext()){
        String server = (String) iterator.next();
        int weigth = serverMap.get(server);
        for (int i = 0; i < weigth; i++){
            serverList.add(server);
        }
    }
    String server = null;
    synchronized (pos){
        if (pos > serverList.size() - 1)
            pos = 0;
        server = (String) serverList.get(pos);
        pos++;
    }
    return server;
}
```

通过负载均衡算法后，得到的服务提供者的 IP 地址返回给通信调度层，进行网络 I/O 连接处理。

### (3) 请求发布

MessageSendHandler 消息处理类继承了 ChannelInboundHandlerAdapter 类，是客户端的 ChannelPipeline 处理链中的业务处理 Handler，它也处于 ChannelPipeline 的最顶端，它用来处理请求消息。

客户端将请求服务封装为 MessageRequest 请求对象后，调用 sendRequest 方法，将 MessageRequest 消息发送到远程服务器。代码实现如下：

```
public MessageCallback sendRequest(MessageRequest request) {
    MessageCallback callBack = new MessageCallback(request);
    mapCallBack.put(request.getMessageId(), callBack);
    channel.writeAndFlush(request);
    return callBack;
}
```

根据请求，创建一个 MessageCallback 异步回调对象，异步等待远程调用的结果。通过 channel.writeAndFlush(request)函数将请求消息写入到 Channel 的上下文中，等待 Channel 的处理。

Channel 通道检测到由写事件到达后, 触发 MessageSendHandler 的 channelActive 方法, 处理 I/O 事件。实现代码如下:

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {  
    super.channelActive(ctx);  
    this.remoteAddr = this.channel.remoteAddress();  
}
```

根据负载均衡得到的 IP 地址, 通过客户端 I/O 处理模块将远程请求消息发送出去。

#### 4.7.2 远程响应消息处理

客户端发送远程调用请求后, 需要处理返回的响应消息。在 RPC 服务器启动的时候, 启动线程池, 将封装了客户端 I/O 事件的 MessageSendInitializeTask 任务提交到线程池, 不断的监听远程响应的消息。代码如下:

```
public Boolean call() {  
    Bootstrap b = new Bootstrap();  
    b.group(eventLoopGroup)  
        .channel(NioSocketChannel.class)  
        .option(ChannelOption.SO_KEEPALIVE, true)  
        .remoteAddress(serverAddress);  
    b.handler(new MessageSendChannelInitializer().  
        buildRpcSerializeProtocol(protocol));  
    ChannelFuture channelFuture = b.connect();  
}
```

MessageSendInitializeTask 是客户端请求消息发送执行的任务, 它实现了 Callable 接口, 是个可执行的 Task 任务。它的核心是 call 方法, call 方法封装了客户端 I/O 事件处理的实现, 用来监听客户端 Channel 通道中的事件。

若异步响应失败, 则调用 I/O 线程 eventLoopGroup.schedule(new Runnable()) 处理定时任务, 周期性的执行当前的 I/O 监听事件。

若客户端处理 I/O 事件成功, 当监听到远程调用响应的消息事件后, 从 ChannelPipeline 中获取客户端 MessageSendHandler 业务处理 Handler 实例, 用来处理响应消息。代码如下:

```
if (channelFuture.isSuccess()) {  
    MessageSendHandler handler = channelFuture.channel().pipeline()  
        .get(MessageSendHandler.class);  
    RpcServerLoader.getInstance().setMessageSendHandler(handler);  
}
```

MessageSendHandler 类是 ChannelPipeline 中的业务处理 Handler, 它继承了 ChannelInboundHandlerAdapter 类, 当有数据达到 I/O 通道时, 会触发 channelRead 方法执行, 处理 Channel 通道中的读事件。channelRead 核心代码如下:



```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
    MessageResponse response = (MessageResponse) msg;  
    String messageId = response.getMessageId();  
    MessageCallback callBack = mapCallBack.get(messageId);  
    if (callBack != null) {  
        mapCallBack.remove(messageId);  
        callBack.over(response);  
    }  
}
```

当 Channel 中的响应消息到达 MessageSendHandler 时,已经经过了 TCP 半包问题处理和解码,这个时候接收到 object 对象就是 MessageResponse。根据响应对象 messageId 表示找到对应的 MessageCallback 回调对象,调用 over 方法将响应传递给对应的请求对象<sup>[52]</sup>。

## 4.8 长连接服务

### 4.8.1 长连接原理

服务端和客户端在网络中采用 TCP 协议进行通信,在对数据进行读写之前,双方要建立一个连接,读写操作基于这个连接进行,当读写操作完成后,服务端和客户端就可以释放这个连接<sup>[53]</sup>。TCP 经过三次握手建立连接,经过四次握手释放链接,在每次进行连接的时候都需要消耗资源和时间。

RPC 服务器采用远程调用的形式,可能客户端频繁发送请求,如果采用短连接,每一个请求都需要建立连接,会给服务器造成非常大的压力,而且创建这么多连接会耗费大量的资源,造成网络堵塞。长连接是指在系统通信链路建立之后,不会马上释放,一直保持。由于不需要经常建立以及关闭连接,可以减少 CPU 内存的使用、网络的堵塞和后续请求的响应时间。

长连接功能是通过心跳检测和重连机制实现的<sup>[54]</sup>,当链路空闲时,每间隔一段时间发送心消息,检测链路的是否可用,当链路断开,通过重连机制进行重新连接,保证链路的可用性。

RPC 服务器服务端和客户端之间通信模型如图 4.28 所示:

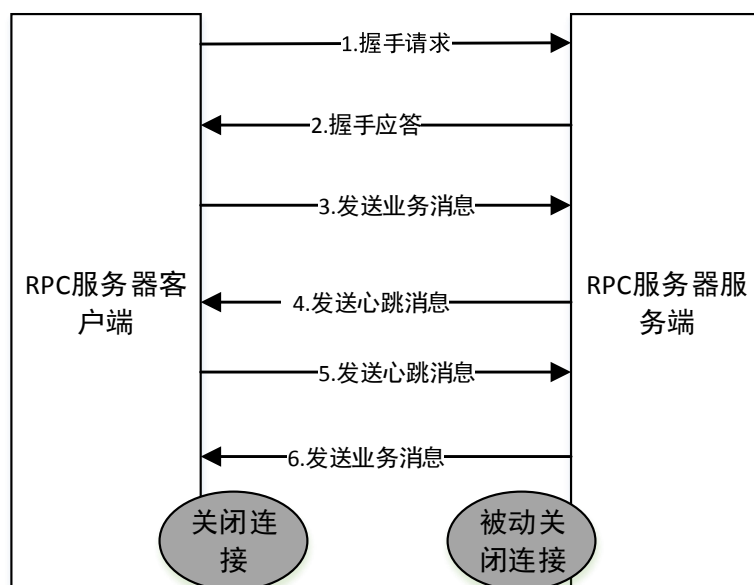


图 4.28 RPC 服务器通信模型

### 4.8.2 心跳检测机制

一般来说，客户端和服务端建立长连接后，这个连接会一直保持，双方基于这个连接进行通信。但是由于网络环境不是一成不变的，可能由于某个节点出现问题，而导致连接的断开。

心跳检测的原理是：当链路处于空闲状态，客户端发送一个很小的心跳包给服务端，服务端接收到心跳包后回复应答信息。如果服务端在一段时间内没有接收到心跳消息，则判断连接已断开。客户端利用 `NioEventLoop` 线程每隔固定时间发送一次心跳包，以通知服务端连接有效<sup>[55]</sup>。

发送心跳包的目的是为了检测连接是否有效，对心跳包的内容没有任何要求，它可以只包含包头信息。一般情况下，如果 30-40 没有接收到心跳包，那么就判断连接失效。对于链路要求很高的情况下，保持在 6-9 秒之内即可。

本文设计的 RPC 服务器心跳策略如下：

- (1) 客户端与服务端建立连接；
- (2) 将心跳处理器 `IdleStateHandler` 添加到客户端的 `ChannelPipeline` 处理链中，若客户端 `Channel` 中 5s 内没有数据到达，则会触发 `userEventTriggered` 触发器发送心跳包给服务端。
- (3) 服务端接收到心跳包后，不作任何处理，避免由于需要回复大量心跳消息，而造成服务器资源浪费和性能损耗。客户端每隔 5s 发送一次心跳消息，如果服务端在 10s 内没有接收到心跳包，则判断客户端连接已断开。

(4) 若连接断开, 则客户端需要利用重连机制, 向服务端重新发起连接。

下面是服务端和客户端心跳机制实现过程:

#### (1) 客户端发送心跳消息

连接建立成功, 客户端主动发送心跳包, 心跳包的消息体可以没有任何内容。服务端接收到消息后, 返回心跳应答消息。

HeartBeatReqHandler 是继承于 ChannelHandlerAdapter, 是客户端处理心跳消息的处理器, 它是一个 ChannelHandler, 只需要将它添加到 ChannelPipeline 中就可以通过 channelRead 用来处理网络事件。channelRead 方法的核心代码如下:

```
public void channelRead(ChannelHandlerContext ctx, Object msg)
    throws Exception {
    NettyMessage message = (NettyMessage) msg;
    if (message.getHeader() != null
        && message.getHeader().getType() == MessageType.LOGIN_RESP
            .value()) {
        heartBeat = ctx.executor().scheduleAtFixedRate(
            new HeartBeatReqHandler.HeartBeatTask(ctx), 0, 5000,
            TimeUnit.MILLISECONDS);
    }
}
```

客户端判断心跳包消息头部信息是否满足协议要求, 若满足要求, 则将心跳包封装为定时任务, 利用 NioEventLoop 线程将任务提交到线程池处理。线程池调用 scheduleAtFixedRate() 方法, 将心跳包以 5s 为周期向服务端定时发送。

#### (2) 服务端心跳应答

服务端接收到客户端的心跳消息后, 构造应答消息, 返回给客户端。HeartBeatRespHandler 继承了 ChannelHandlerAdapter, 它也是一个 ChannelHandler, 将它添加到服务端的 ChannelPipeline 中处理心跳应答。channelRead 实现代码如下:

```
public void channelRead(ChannelHandlerContext ctx, Object msg)
    throws Exception {
    NettyMessage message = (NettyMessage) msg;
    if (message.getHeader() != null
        && message.getHeader().getType() == MessageType.HEARTBEAT_REQ
            .value()) {
        NettyMessage heartBeat = buildHeartBeat();
        ctx.writeAndFlush(heartBeat);
    } else {
        ctx.fireChannelRead(msg);
    }
}
```

服务端利用 Netty 的 ReadTimeoutHandler 处理器处理心跳超时问题, 如果 10s 内没有收到客户端的任何消息, 则主动关闭连接。



### （3）断连和重连

当客户端发现连接失效后，需要重新发起连接，代码实现如下：

```
} finally {  
    executor.execute(new Runnable() {  
        @Override  
        public void run() {  
            try {  
                TimeUnit.SECONDS.sleep(1);  
            }  
            try {  
                connect(NettyConstant.PORT, NettyConstant.REMOTEIP);  
            }  
        }  
    });  
}
```

客户端 NioEventLoop 线程监听到连接断开，首先将所有资源清空，然后通过 Bootstrap 发起重连操作。将重连任务提交到线程池执行，调用 connect 方法实现重连过程。

## 4.9 本章小结

本章根据 RPC 服务器调用处理的逻辑，从代码角度详细分析了服务器的消息定义、编解码模块、I/O 调度处理模块、业务处理模块、客户端和服务端调度处理以及长连接服务的设计思路和实现方法。通过将这些模块整合在一起，实现一个完整的 RPC 服务器。

## 第五章 基于 Netty 的 RPC 服务器的性能测试与优化

本章节主要利用仿真实验对实现的 RPC 服务器进行测试,验证其各个模块的性能,主要从以下几个方面进行测试:RPC 服务器消息序列化能力、长连接以及并发处理性能。

### 5.1 RPC 服务器长连接性能测试

在分布式系统中客户端发送大量数据,为了减少 TCP 连接的次数,采用长连接可以减少服务器的资源占用,提高服务器的处理能力<sup>[56]</sup>。在空闲时,通过心跳监测机制来检测链路的连通性,通过心跳检测和重连机制来实现长连接。本节通过设计两个仿真实验分别验证服务器的心跳检测机制以及长连接下系统性能。

#### 5.1.1 心跳检测和重连机制测试

##### (1) 实验目的

在 RPC 服务器客户端和服务端链路成功后,验证如下功能:

- (a) 是否每隔 5s 钟互发一次心跳请求和应答;
- (b) 当服务器宕机一段时间,客户端能够正常发起重连,重连成功之后不再重连;
- (c) 服务器重启成功后,客户端能够重新发起连接;
- (d) 当链路处于空闲状态一定时间,服务端可以关闭该连接通道,减少资源损耗。

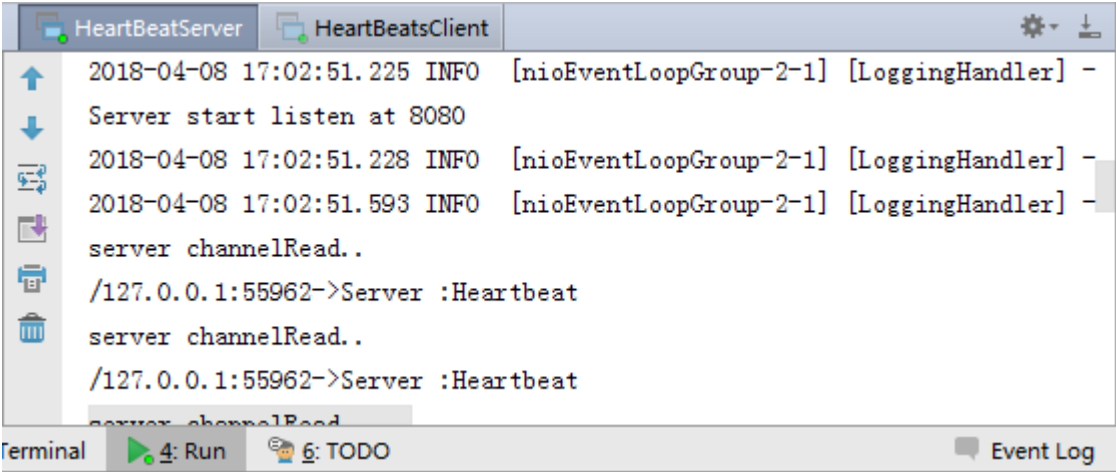
##### (2) 实验步骤

分别开启一个服务端和一个客户端程序,客户端与服务端建立连接,每隔 5s 互发心跳包。关闭服务端,观察客户端连接情况。过一会重启服务端,再观察客户端连接情况。

##### (3) 实验结果

- (a) 先启动服务端,再启动客户端,实验现象如下。

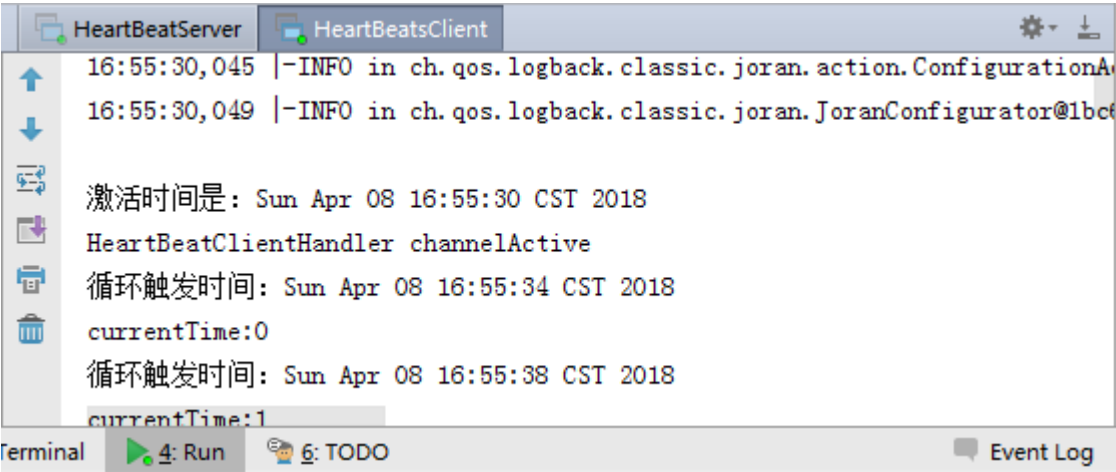
服务端现象如图 5.1 所示:



```
HeartBeatServer | HeartBeatsClient |
2018-04-08 17:02:51.225 INFO [nioEventLoopGroup-2-1] [LoggingHandler] -
Server start listen at 8080
2018-04-08 17:02:51.228 INFO [nioEventLoopGroup-2-1] [LoggingHandler] -
2018-04-08 17:02:51.593 INFO [nioEventLoopGroup-2-1] [LoggingHandler] -
server channelRead..
/127.0.0.1:55962->Server :Heartbeat
server channelRead..
/127.0.0.1:55962->Server :Heartbeat
server channelRead..
```

图 5.1 服务端心跳情况

客户端现象如图 5.2 所示：



```
HeartBeatServer | HeartBeatsClient |
16:55:30,045 |-INFO in ch.qos.logback.classic.joran.action.ConfigurationA
16:55:30,049 |-INFO in ch.qos.logback.classic.joran.JoranConfigurator@1bce
激活时间是: Sun Apr 08 16:55:30 CST 2018
HeartBeatClientHandler channelActive
循环触发时间: Sun Apr 08 16:55:34 CST 2018
currentTime:0
循环触发时间: Sun Apr 08 16:55:38 CST 2018
currentTime:1
```

图 5.2 客户端心跳情况

可以看出客户端与服务端建立了连接，并给服务端发送心跳消息，在 5s 内服务端收到客户端发送的心跳消息，证明客户端还存活，链路保持畅通。

(b) 客户端处于空闲状态，没有给服务端发送心跳消息，服务端经过 5s 没有收到消息。

服务端的现象如图 5.3 所示：

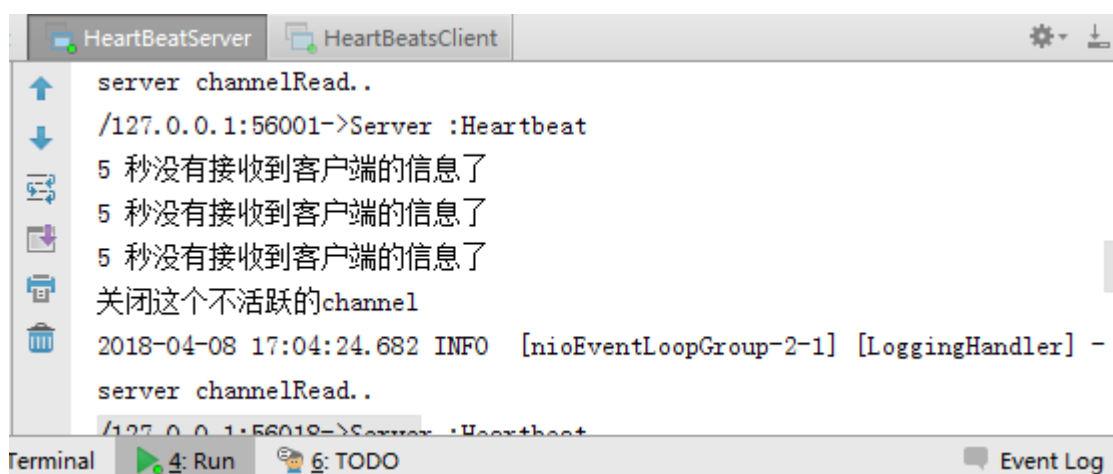


图 5.3 服务端未接收到客户端的消息

客户端的现象如图 5.4 所示：

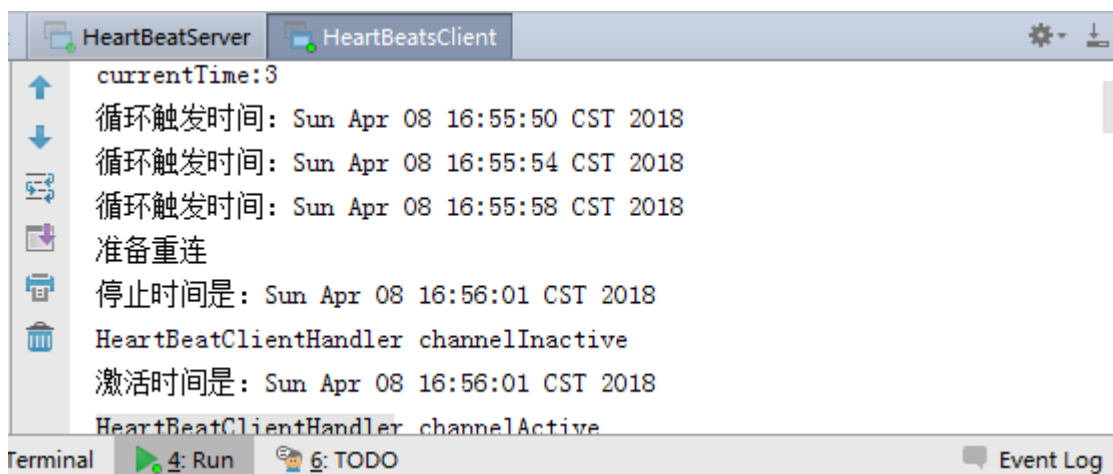


图 5.4 客户端发现连接断开，自动重连

当客户端处于空闲状态，服务端 10s 未接收到客户端消息时，会关闭与该客户端的连接。当客户端再次发送心跳消息时，会重新激活与服务端之间的链路。

(c) 服务端宕机一段时间后重新启动，如图 5.5 所示：

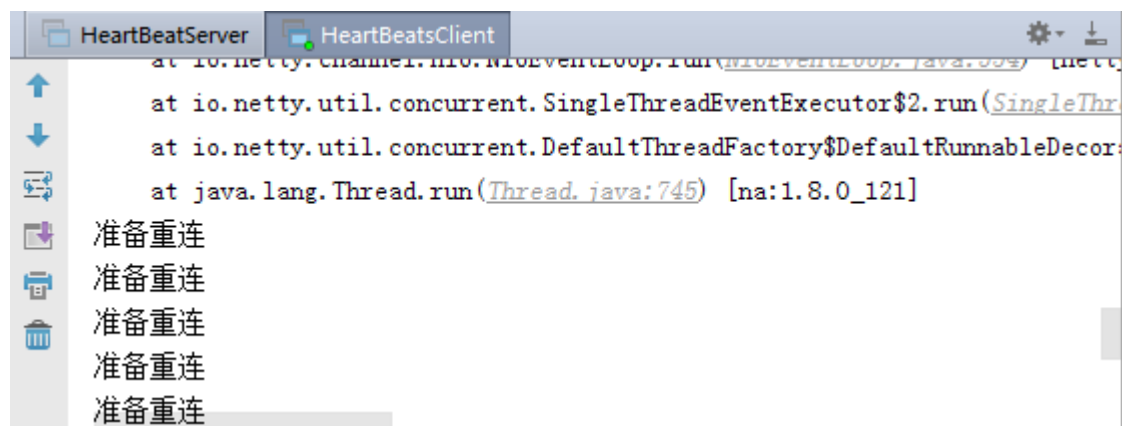


图 5.5 连接断开，客户端周期性的发起重连

服务器宕机后，客户端检测到服务端连接关闭，开始周期性的尝试重连，此时心跳定时器停止工作，不再发送心跳消息。由于客户端周期性的重连，因此线程数量占用增大。服务端重启之前客户端资源占用如图 5.6 所示。

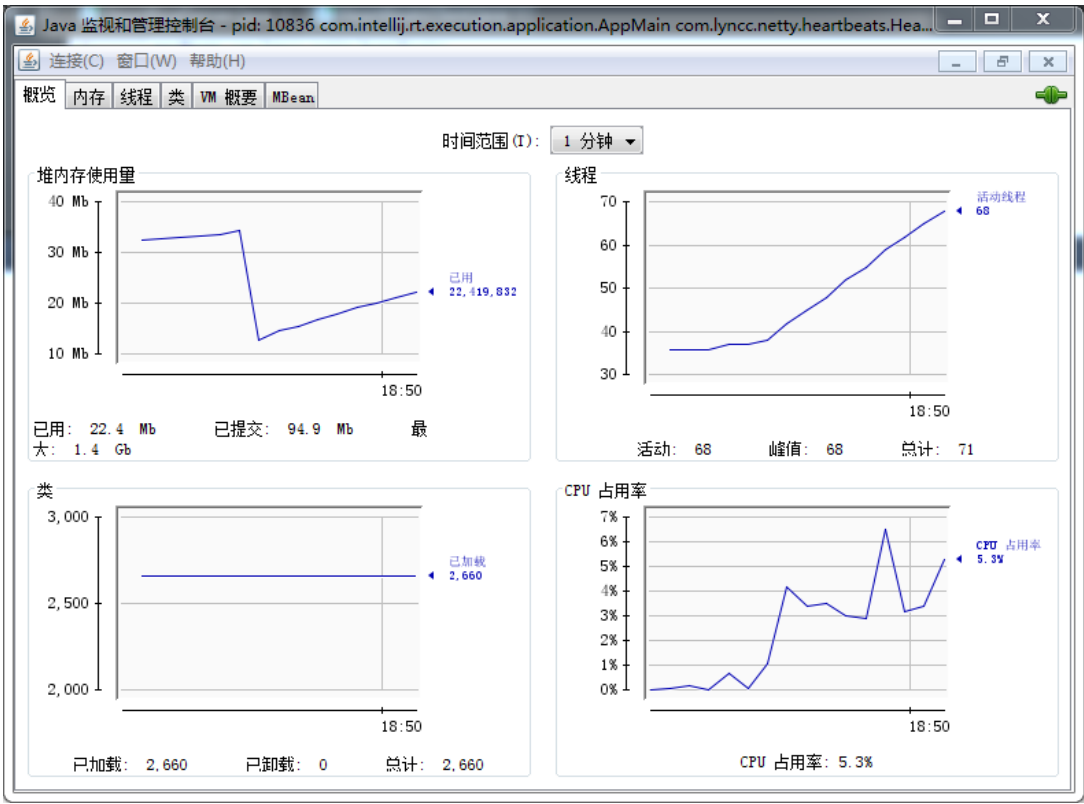


图 5.6 客户端资源占用情况

(d) 服务端重启，重启成功后，客户端与服务端握手成功，链路重新恢复，如图 5.7 所示：

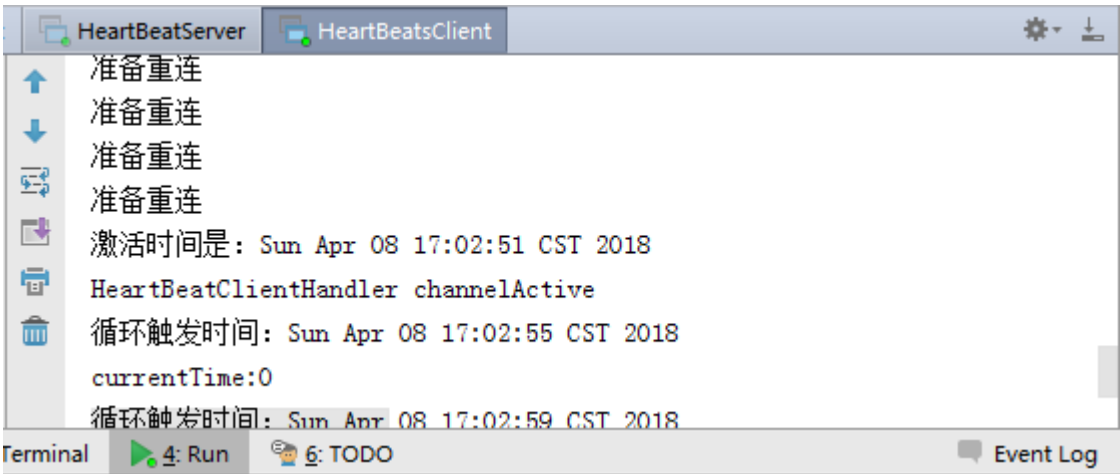


图 5.7 客户端与服务端握手成功，重新发送心跳消息

从上面实验中可以看出，客户端和服务端连接成功后，周期性向服务端发送心跳包。当客户端停止发送消息，服务端 10s 没有检测到心跳消息时，就会主动关闭链路，减少网络资

源浪费。客户端重新发送心跳包时，会重新激活连接。当服务端宕机时，连接释放，客户端会周期性的发起重连，此时会增加 CPU 的占用。服务端重启后，客户端会自动发起重连操作。

通过这种心跳检测和重连机制保持客户端和服务端之间的长连接，虽然客户端重连会短时间导致 CPU 资源占用增多，但是减少了网络 IO 资源的占用，而且心跳包很小，不会占用网络带宽。在分布式系统中，心跳检测机制可以降低 RPC 服务器的负载压力。

5.1.2 长连接性能测试

(1) 实验目的

RPC 服务器分别使用长连接和短连接时，验证在高频次 RPC 调用时服务器的性能。

(2) 实验过程

(a) 客户端和服务端都是单进程，处于长连接，在单次链接内分别发起 1w 次、5w 次和 10w 次 RPC 调用，计算耗时；

(b) 客户端和服务端都是单进程，处于短连接，分别共发起 1w 次、5w 次和 10w 次连接。每次连接单次 RPC 调用，计算耗时。

(3) 实验结果

表 5.1 长连接和短连接的 RPC 调用耗时

耗时	1w (s)	5w (s)	10w (s)	15w (s)
长连接（单次连接，多次 RPC）	0.632	2.975	5.871	9.325
短连接（多次连接，单次 RPC）	1.453	6.826	12.347	19.908

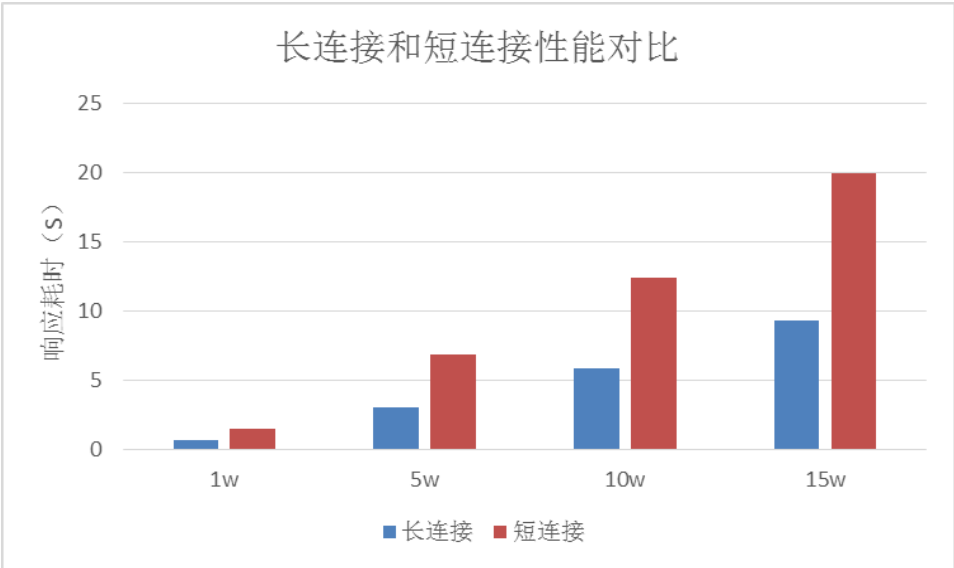


图 5.8 并发响应时间对比

从图 5.8 中，我们可以看出，长连接的性能要优于短连接，性能差距在 2.2 倍左右。这是因为长连接不需要每次请求都创建连接，这样大大减少了服务器的 I/O 负担，提高了服务器

的处理性能。

## 5.2 RPC 服务器消息序列化性能测试

本文设计的 RPC 服务器有三种编解码方式，分别是 Netty 集成的 ObjectEncoder 和 ObjectDecoder（Java 原生编解码器），以及通过引入第三方编码框架 Kryo、Hessian 对 RPC 消息序列化、反序列化进行特殊定制的编解码器。本实验通过 10 次瞬时并行度 1w 次的 RPC 请求来验证这三种序列化方式的性能。

### 5.2.1 RPC 服务器序列化性能对比

#### （1）实验目的

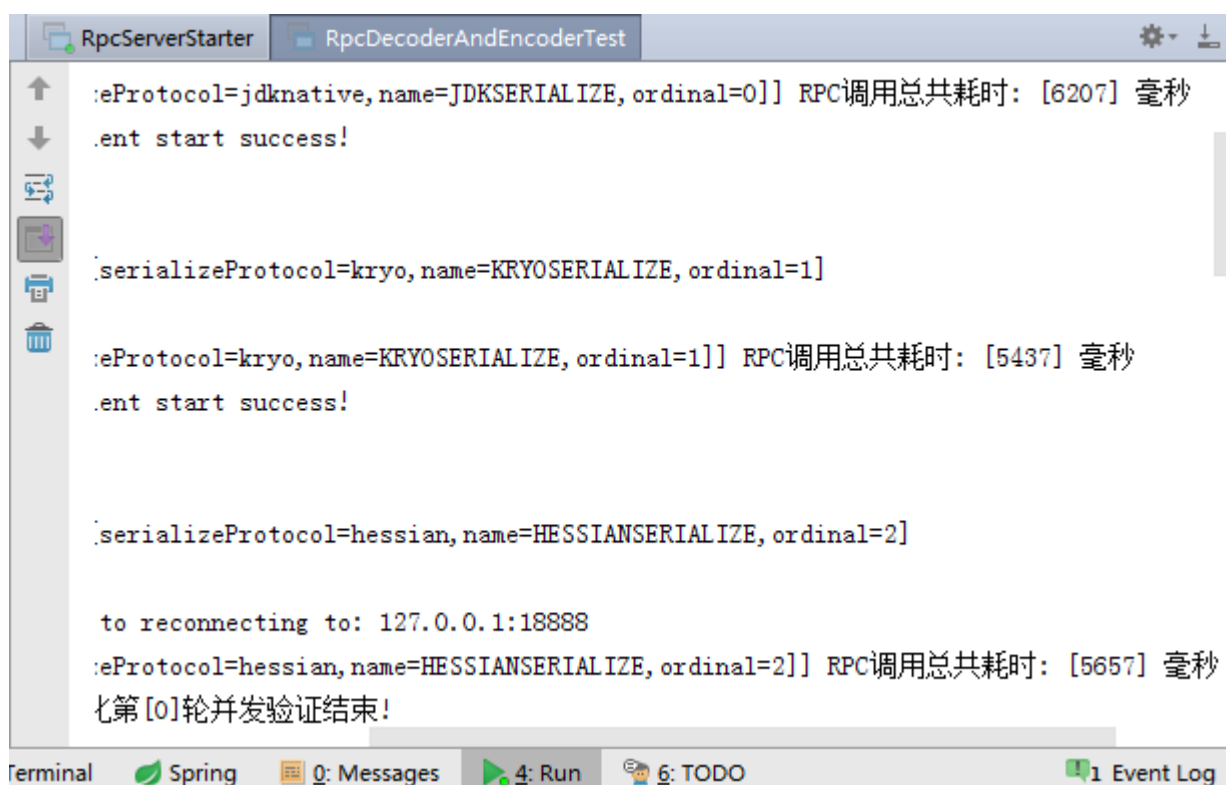
在大量请求情况下，验证 RPC 服务器的几种消息序列化方式的性能。

#### （2）实验过程

启动三台 RPC 服务器，分别进行 10 次实验，每次实验包含 1w 次的并发求和计算请求。统计每一种协议的编解码耗时（毫秒），根据实验结果对比分析这三种序列化协议的序列化性能。

#### （3）实验结果

Java 原生序列化、Kryo 和 Hessian 序列化方式的耗时运行如图 5.9 所示：



```
RpcServerStarter  RpcDecoderAndEncoderTest
↑
:Protocol=jdknative,name=JDKSERIALIZE,ordinal=0]] RPC调用总共耗时: [6207] 毫秒
↓
.ent start success!

.serializeProtocol=kryo,name=KRYOSERIALIZE,ordinal=1]

:Protocol=kryo,name=KRYOSERIALIZE,ordinal=1]] RPC调用总共耗时: [5437] 毫秒
.ent start success!

.serializeProtocol=hessian,name=HESSIANSERIALIZE,ordinal=2]

to reconnecting to: 127.0.0.1:18888
:Protocol=hessian,name=HESSIANSERIALIZE,ordinal=2]] RPC调用总共耗时: [5657] 毫秒
七第[0]轮并发验证结束!
```

Terminal Spring Q: Messages 4: Run 6: TODO 1 Event Log

图 5.9 三种序列化方式运行情况

10 轮压力测试数据的统计如表 5.2 所示：

表 5.2 10 轮压力测试数据统计

序列化方式	Java 本地序列化(ms)	Kryo 序列化 (ms)	Hessian 序列化 (ms)
第 1 轮	11256	5710	5413
第 2 轮	6608	6013	5378
第 3 轮	5701	5824	5720
第 4 轮	5820	5437	4167
第 5 轮	5911	4123	3978
第 6 轮	5867	5630	5380
第 7 轮	5916	3917	4521
第 8 轮	5804	4803	4292
第 9 轮	5931	4465	4478
第 10 轮	5985	4558	6129
平均耗时	6479.9	5521.9	5447.5

从表 6.2 中的 10 次实验测试数据对比可以看出，Kryo 和 Hessian 框架的编解码性能差别不大，它们在总体上比 Java 原生序列化方式的序列化性能更好，编解码速度更快。

10 轮压力测试实验中，Java 本地序列化、Kryo 序列化以及 Hessian 序列化的耗时情况如图 5.10 所示。

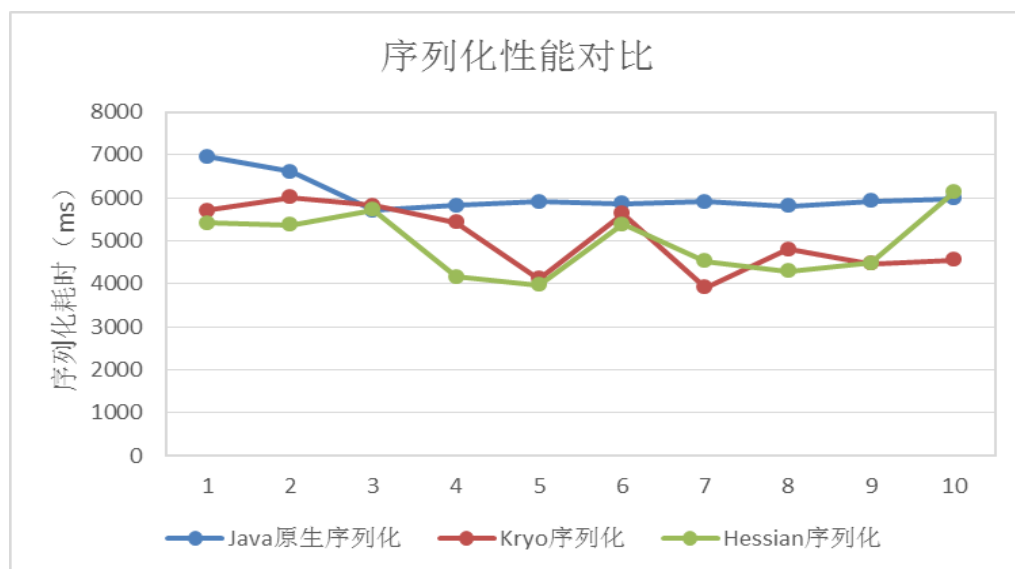


图 5.10 序列化耗时波动图

从图 5.10 中可以看出，Java 原生序列化方式性能比较平缓，而 Kryo 和 Hessian 序列化方式有一定的波动，并且有很明确的峰值拐点。但是从总体的耗时上对比，Kryo 和 Hessian 耗时都比 Java 原生方式小，速度更快。

本文设计的 RPC 服务器框架有很好的扩展性，通过集成第三方优良的框架来解决 Java 序列化机制编码、解码上的不足，提高服务器的处理速度。



### 5.3 RPC 服务器并发性能测试

#### (1) 实验目的

在高并发场景下，客户端程序向远程 RPC 服务器发起调用，调用成功，服务端返回正确的结果给调用程序，通过大量并发的 RPC 请求来验证服务器的并发处理性能。

#### (2) 实验过程

启动两台客户端，分别发送 10 次加法运算 RPC 调用请求，并发请求初始数量为 1000，每次递增 1000 次。服务端处理完成后，打印 RPC 请求消息对应的 messageId 标识号，然后将结果返回。客户端每收到一条响应消息就打印出来，全部接收完毕后打印调用时间。

#### (3) 实验结果

服务端启动后，接收远程客户端的 RPC 请求，每执行一个 RPC 请求就打印对应的请求消息的 messageId，运行结果如图 5.11 所示。

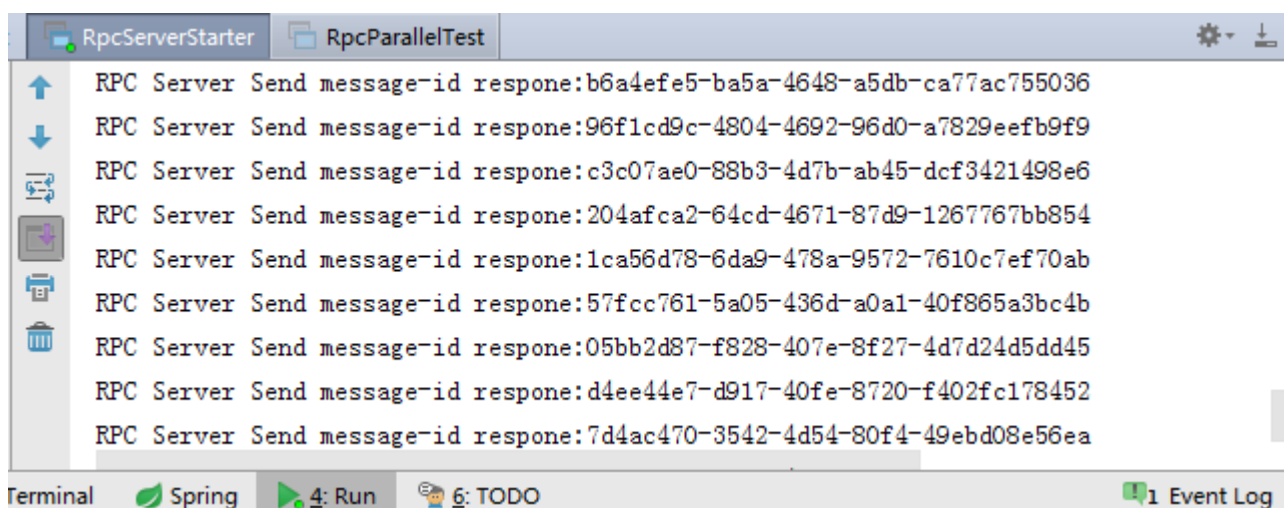


图 5.11 服务端运行结果

客户端进行远程调用服务，调用结束接收到的响应消息，每接收一条响应消息，将对应的结果打印出来，运行结果如图 5.12 所示。

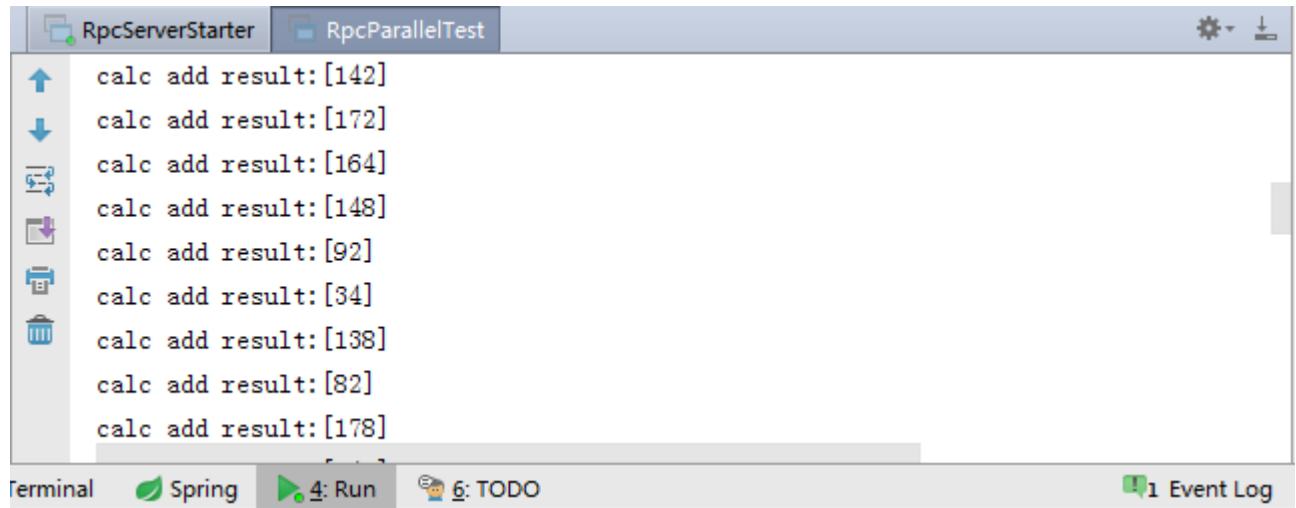


图 5.12 客户端运行结果

10 次不同次数的并发 RPC 请求调用的响应时间如表 5.3 所示，TPS 如表 5.4 所示。

表 5.3 并发场景 TPS

请求次数	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Client1	292	385	743	1039	1489	1991	2717	3396	4255	5260
Client2	290	383	758	1095	1519	2030	2703	3524	4163	5165

表 5.4 并发场景响应时间

请求次数	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Client1	3424	5194	4037	3850	3358	3013	2576	2356	2115	1901
Client2	3428	5222	3957	3653	3292	2956	2590	2270	2161	1936

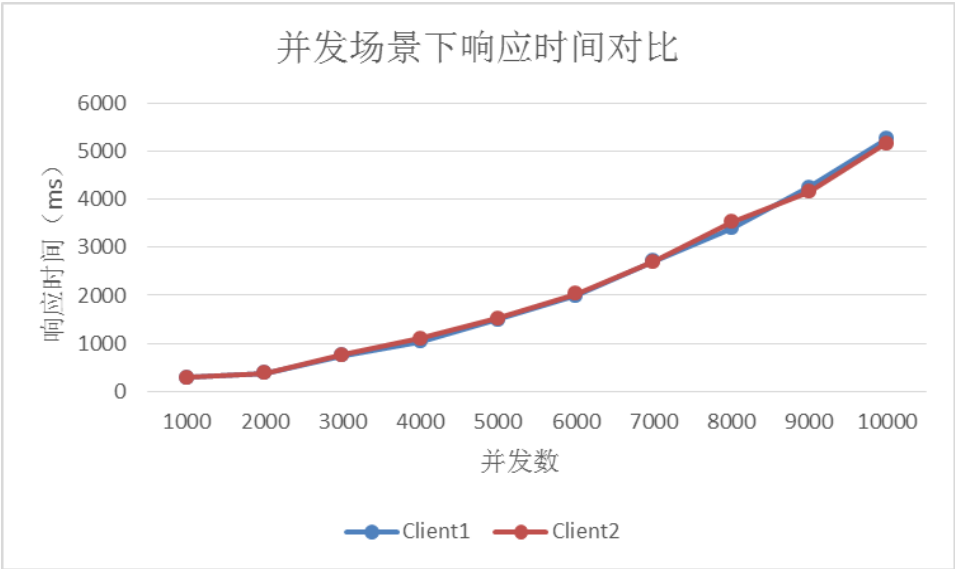


图 5.13 并发场景下响应时间对比

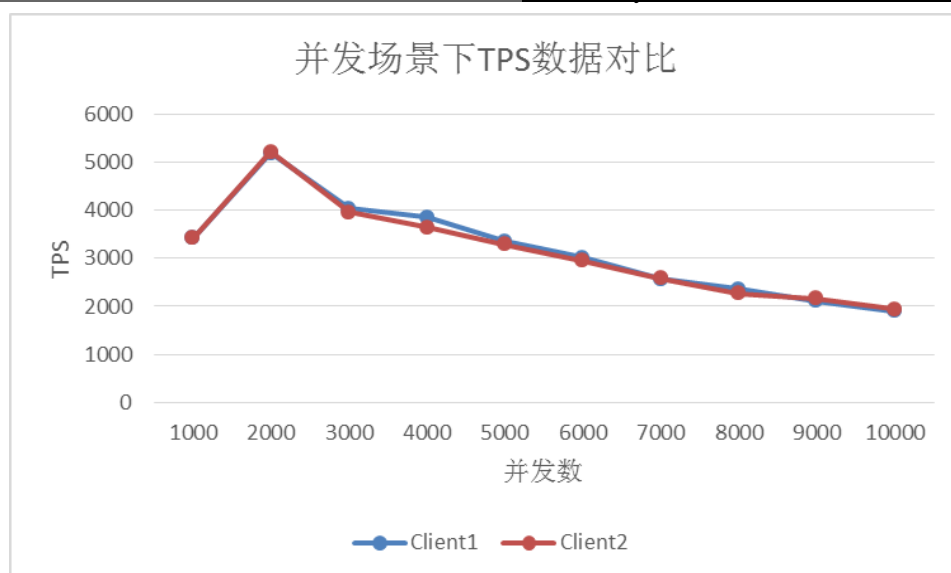


图 5.14 并发场景下 TPS 数据对比

从图 5.13 和图 5.14 中的数据对比可以看出，本文设计的 RPC 服务器的并发数在 2000 左右 TPS 最高，性能最好，TPS 最高为每秒处理 5000 个 RPC 请求消息。随着并发数的增多，受限于实验硬件条件影响，服务器性能会慢慢下降。服务器处理时间都维持在 10s 以下，可以看出服务器并发处理性能比较好。

## 5.4 RPC 服务器的优化

### （1）对象序列化的优化

本文设计的 RPC 服务器支持 Java 原生序列化、Kryo 和 Hessian 这三种序列化方式，目前还有很多其它主流的编解码框架的序列化性能非常高，如 Protostuff 是相比于 Kryo 更加优秀的序列化框架，通过引入 Protostuff 来增强服务器的编解码能力<sup>[57]</sup>。

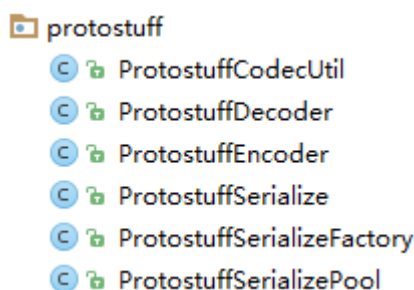
Protostuff 是对 Google Protobuf 框架的优化。Protobuf 在编解码时，需要自己创建一个 .proto 的配置文件，在文件中定义编解码的格式<sup>[58]</sup>。利用 Protobuf 自带的 protoc 工具将 .proto 文件编译为程序可以使用的 Java 文件，Protostuff 则省略了这个编译过程。

（a）引入 Protostuff 的 jar 包，如图 5.15 所示。

```
<dependency>
  <groupId>com.dyuproject.protostuff</groupId>
  <artifactId>protostuff-core</artifactId>
  <version>1.0.8</version>
</dependency>
<dependency>
  <groupId>com.dyuproject.protostuff</groupId>
  <artifactId>protostuff-runtime</artifactId>
  <version>1.0.8</version>
</dependency>
```

图 5.15 Protostuff 依赖配置

(b) 实现 MessageCodecUtil 接口，利用 Protostuff 的 API 实现对应的 encode 和 decode 方法，来完成编解码功能实现。Protostuff 的编码功能实现如图 5.16 所示。



```
protostuff
├── ProtostuffCodecUtil
├── ProtostuffDecoder
├── ProtostuffEncoder
├── ProtostuffSerialize
├── ProtostuffSerializeFactory
└── ProtostuffSerializePool
```

图 5.16 Protostuff 的编解码实现

(c) 优化后，Protostuff、Kryo 和 Hessian 序列化性能对比如图 5.17 所示，从图中可以看出 Protostuff 编解码框架的序列化性能比 Kryo 和 Hessian 框架更优。

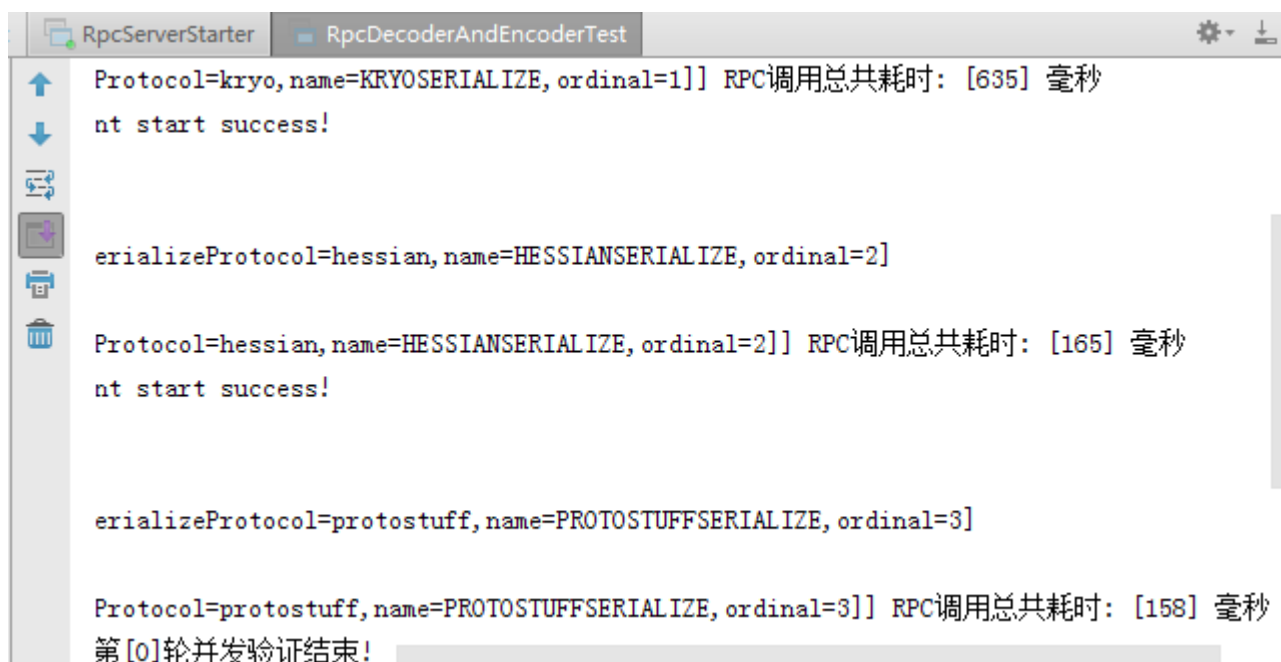


图 5.17 Protostuff、Kryo 和 Hessian 序列化性能对比

## (2) 其他优化思路

### （a）线程模型优化

服务器的通信调度模块使用的是 Netty 的主从线程模型，主从线程模型的 I/O 处理能力非常强，完全可以应付大量并发请求。但是对于实际生产环境中，并不是每笔业务都需要这么大的并发处理能力<sup>[59]</sup>。

由于 Netty 可以同时支持单线程、多线程以及主从多线程模型，根据实际的业务需求，通过对 Netty 线程参数的设置，来启动不同的线程模型处理，避免资源的浪费，提高系统性能。

### （b）接入 Zookeeper

目前 RPC 服务器还处于独立的个体，考虑是否可以通过某种机制，将集群中的若干台服务器进行统一的分布式协调管理和调度<sup>[60]</sup>。

可以引入 Zookeeper 注册中心，进行服务的治理。服务端启动时，将提供的服务注册到 Zookeeper 的管理列表中，进行服务上下线监控管理。客户端启动时，Zookeeper 将客户端订阅的服务推送给相应的客户端进程，以便客户端进程可以根据服务地址向远程服务端发起调用<sup>[61]</sup>。通过 Zookeeper 的统一调度管理，提高服务器的协作能力和处理性能。

## 5.5 本章小结

本章主要通过仿真实验，对 RPC 服务器的各个模块的功能和性能进行测试，根据实验结果对服务器的整体性能做出评价。通过集成 Protostuff 序列化框架对服务器的编解码性能进行优化，并提出了一些优化方案。

## 第六章 总结与展望

### 6.1 总结

现今,随着互联网的高速发展,网站系统的并发访问量日渐庞大,这对于RPC服务器的处理性能要求越来越高。RPC服务器需要处理大量I/O操作,而I/O操作比较耗时,如何提高服务器的处理性能成为现今研究的重点。Netty是一个高性能的异步通信框架,它在传统NIO框架的基础上进行了优化,通过合理的I/O线程设计,解决了大量I/O并发处理的性能问题。在本文中将它作为服务器的底层通信框架,以提高RPC服务器的通信能力。服务端接收RPC消息时,利用Netty的解码器解决TCP半包问题,确保传递给业务处理模块的是一个整包消息。客户端发送RPC远程请求消息时,利用Netty的编码器对请求消息进行长度编码,以便接收端可以根据长度信息将请求消息解码出来。通过引入第三方编解码框架提高服务器的序列化能力,并将这些编解码器添加到Netty的ChannelPipeline职责链中,对Channel通道中的消息进行处理。利用Netty的心跳检测和重连机制实现RPC服务器的长连接功能,在高并发的情况下,避免客户端与服务端频繁建立连接,降低服务器的资源占用。

最后对实现的RPC服务器进行优化,通过引入当前比较流行的Protostuff序列化框架,进一步提高服务器的编解码能力,并对服务器的其他方面优化提出相关的思路。通过实验对RPC服务器的各个功能进行测试,根据实验数据分析验证了RPC服务器的高性能。主要工作如下:

(1) 介绍了本课题相关的选题背景以及目前国内外Netty技术研究的现状,根据目前开源项目框架整合的过程,开始RPC服务器的设计与实现。

(2) 研究了目前RPC远程调用过程实现的原理机制,了解相关的I/O通信模型演进过程以及远程调用涉及的负载均衡相关的算法。

(3) 从服务器的功能和性能需求出发,研究作为一个RPC远程调用服务器,所需要具备的功能点。同时分析了服务器远程调用处理的过程,最后搭建整个系统的框架,并对Netty的相关核心技术进行研究。

(4) 从服务器处理的逻辑过程出发,实现了服务器的编解码模块、I/O调度处理模块、服务的发布和订阅、业务处理模块、服务端和客户端远程调用处理的过程以及长连接服务,实现了RPC服务器的核心功能。

(5) 为了验证服务器的各个模块的实现效果,针对该服务器,设计了详细的实验测试过

程，对服务器的功能完备以及高性能要求进行测试。通过分析，提出对 RPC 服务器的一些优化方案。

## 6.2 展望

现有的各大网站系统架构中的 RPC 服务器功能非常完备，而且性能非常高，可以同时处理成百上千万的 RPC 请求，但是因为个人时间和技术有限，只是通过自己的设计，实现了 RPC 服务器的一些核心功能。经过测试，本论文实现的 RPC 服务器实现了远程调用处理的一系列核心功能，但是仍然存在一些不足之处例如：

（1）本文设计实现了系统服务器之间的远程过程调用的功能，服务器彼此之间根据 IP 地址进行远程调用，没有将服务器整合到 zookeeper 注册中心进行统一管理，监听服务的上下线。

（2）RPC 服务器只是初步实现了对 RPC 请求消息的编解码功能，可以引入当前主流的第三方编解码框架提升服务器的序列化性能。

（3）服务器使用加权轮询算法进行负载均衡处理，并没有考虑到系统集群的动态负载情况，利用加权轮询法计算可能导致有些服务器的负载加重。

（4）在开发过程中，采用 Netty 框架的异步处理技术处理异步回调，没有使用锁机制来保证并发响应的安全。

## 参考文献

- [1] 郭庆涛,孙强强,李永攀,于晓军,郑滔.高性能网络服务器框架的研究与实现[M].移动互联与通信技术,2013(12):70-74.
- [2] 张琳娜,姚毓才,王元志,王群.分布式系统远程过程调用探析[J].铜陵学院学报,2008,7(3):62-62.
- [3] 查骏.基于NIO的远程调用框架的设计与实现[D].复旦大学,2012.
- [4] 李林锋.Netty权威指南(第二版)[M].北京:电子工业出版社,2015:24-25.
- [5] 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(2):33-37.
- [6] 姜立俊,杨学良.异构环境下异步远程过程调用的设计与实现[J].计算机研究与发展,1995(1):23-27.
- [7] 李小白,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(1):50-52.
- [8] 宋传杰,霍杰.远程过程调用技术的分布式应用[J].小型微型计算机系统,1997(10):65-70.
- [9] 李登.分布式系统负载均衡策略研究[D].中南大学,2002.
- [10] 邹方曼.分布式负载均衡的Java实现[D].北京邮电大学,2012.
- [11] Dhakal S, Hayat M M, Pezoa J E, et al. Dynamic Load Balancing in Distributed Systems in the Presence of Delays: A Regeneration-Theory Approach[J]. IEEE Transactions on Parallel & Distributed Systems, 2007, 18(4): 485-497.
- [12] 吕良干.云计算环境下资源负载均衡调度算法研究[D].新疆大学,2010.
- [13] 魏钦磊.基于集群的动态反馈负载均衡算法的研究[D].重庆大学,2013.
- [14] 张慧芳.基于动态反馈的加权最小连接数服务器负载均衡算法研究[D].华东理工大学,2013.
- [15] 曾自强.基于NIO的Java高性能网络应用的技术研究[D].北京:北京邮电大学,2009,47-69.
- [16] 刘蓬.NIO高性能框架的研究与应用[D].湖南大学,2013.
- [17] 封玮,周世平.基于Java NIO的非阻塞通信的研究与实现[J].计算机系统应用,2004,13(9):32-35.
- [18] 丁黎明.使用NIO提高Java应用输入输出性能[J].中小企业管理与科技(上旬刊),2012(11):276-277.
- [19] <http://java.sun.com/j2se/1.4.2/docs/guide/nio/>, New I/O APIs. Sun Microsystems, 2002.
- [20] Aruna Kalagnanam. Merlin brings nonblocking I/O to the Java platform. developerWorks, 2002(3).
- [21] Greg Travis. Getting started with new I/O (NIO). developerWorks, 2003(7).
- [22] 肖在昌,杨文晖,刘兵.基于WebSocket的实时技术[J].电脑与电信,2012(12):40-42.
- [23] 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[J].工业控制计算机,2016,29(5):11-12.
- [24] Andrew D. Birrell, Bruce Jay Nelson. Implementing remote procedure calls[J]. Journal ACM Transactions on Computer Systems (TOCS), 1984, 2(1): 39- 59.
- [25] 金志国,李炜.基于Netty的HTTP客户端的设计与实现[J].电信工程技术与标准化,2014(1):84-88.
- [26] 潘馨.基于Netty框架实现呼叫中心实时统计系统[D].西安电子科技大学,2014.
- [27] 滕阳阳,胡栋.基于Netty的HTTP协议栈的扩展设计与实现[J].无线通信技术,2017,26(3).
- [28] 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.
- [29] 范华峰.一种基于Netty框架的网络应用服务器设计方法[J].福建电脑,2015(10):33-34.
- [30] 蒋思佳.Netty构建网络应用服务器的研究与实现[J].2011.
- [31] 赵冬.序列化技术的综述和展望[J].电脑与电信,2013(9):39-42.
- [32] 韩星,刘姣,周淑君.基于Netty的RPC通信系统的编解码技术研究[J].电脑知识与技术,2017,13(26).
- [33] 龚鹏,曾兴斌.基于Netty框架的数据通讯服务系统的设计[J].无线通信技术,2016,25(1):46-49.
- [34] Reactor pattern[EB/OL]. [https://en.wikipedia.org/wiki/Reactor\\_pattern#Structure](https://en.wikipedia.org/wiki/Reactor_pattern#Structure).
- [35] 王纪臣.异步RPC的设计与实现[D].吉林大学,2005.
- [36] 许琼,蔡文皓.一种嵌入式RPC的设计与实现[J].电子设计工程,2011,19(5):127-129.
- [37] W.E. Weihl. Remote Procedure Call[M]. Book Distributed Systems, ACM Press, U.S.A 1990.
- [38] A.L. Ananda, B.H. Tay, E.K. Koh. A Survey of Asynchronous Remote Procedure Calls[J]. ACM Operating



- Systems Review, Apr. 1992, 26(2).
- [39] 郑智. 基于Netty框架的数据传输处理模型的设计与实现[D]. 桂林电子科技大学, 2016.
- [40] 郭正敏. 基于SOA架构的分布式服务化治理方案的研究[D]. 南京邮电大学, 2016.
- [41] 欧昌华, 李炳法. 线程池在网络服务器程序中的应用[J]. 信息技术, 2002(5): 11-14.
- [42] Douglas CSSchmtdt, Carlos O Ryan, Irfan Pyarali. Leader/followers. A design pattern for E-client multi-threaded event demultiplexing and dispatching In[J]. Processing of the 7 Pattern Languages Congerence, 2000.
- [43] Thmpso B. Utilizing. Thread pools in performance-critical Appication[EB/OL]. [http://www.intel.com/labs/htt/index.html?iid=CNP4PHTT+Body\\_Labs](http://www.intel.com/labs/htt/index.html?iid=CNP4PHTT+Body_Labs), 2004.
- [44] Peter Mika. Ontologies are us: A unified model of social networks and semantics[J], Web Semantics: Science, Services and Agents on the World Wide Web, 2007, 5: 5-15.
- [45] Klaus-Peter Lühr. RPC Stubs for Distributed Modules[J]. Proceeding EW 2 Proceedings of the 2nd workshop on Making distributed systems work. 1986: 1-4.
- [46] 张广红, 陈平. 关于AOP实现机制和应用的研究[J]. 计算机工程与设计. 2003(08).
- [47] 邓新国, 贾利民, 秦勇. 消息中间件中多线程池并发模型的研究[J]. 铁路计算机应用, 2005, 14(2): 5-7.
- [48] 韩勇, 沈备军. 基于动态代理的Java远程调用框架的研究[J]. 计算机应用与软件, 2010, 27(6): 136-138.
- [49] Wenyu Zhou, Shoubao Yang, Jun Fang, Xianlong Niu, Hu Song. VMCTune: A Load Balancing Scheme for Virtual Machine Cluster Based on Dynamic Resource Allocation. 9th International Conference on Grid and Cooperative Computing (GCC). 2010.
- [50] Cardellini V, Colajanni M, Yu PS. Dynamic load balancing on Web-server systems. IEEE Internet Computing, 1999, 3(3): 28-39.
- [51] Gu X, Nahrstedt K. Dynamic QoS-Aware multimedia service configuration in ubiquitous computing environments. In: Proc. of the 22nd Int'l Conf. on Distributed Computing Systems (ICDCS 2002). Vienna: IEEE Computer Society, 2002. 311-318.
- [52] 廖立君, 李长云, 孙星明, 吴岳忠. 面向Web Services的异步调用机制研究[J]. 计算机应用研究, 2005, 22(11): 184-185.
- [53] 柯刘阳. 基于TCP长连接的负载均衡器设计与实现[D]. 电子科技大学, 2012.
- [54] 温彬民. 一种基于自适应心跳机制的MQTT通信协议的研究与应用[D]. 华南理工大学, 2015.
- [55] 沈晓. TCP异步长连接的选择及心跳处理机制的实现[J]. 中国金融电脑, 2014(4): 37-39.
- [56] Tanenbaum, Andrews. Distributed System : Principle and Paradigms[J]. 2002.
- [57] Google developers Protocol Buffer[EB/OL]. <https://developers.google.com/protocol-buffers/docs/overview>.
- [58] 李纪欣, 王康, 周立发, 章军. Google Protobuf在Linux Socket通讯中的应用[J]. 电脑开发与应用, 2013(4): 1-5.
- [59] 吉利, 潘林云, 刘姚. 线程池技术在网络服务器中的应用[J]. 计算机技术与发展, 2017, 27(7): 149-151.
- [60] 陈天伟, 彭凌西. 基于ZooKeeper的一种分布式系统架构设计与实现[J]. 通信技术, 2018(1).
- [61] D. Darger, E. Lehman, T. Leighton, M. Levine, D. Lewin and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots On the World Wide Web. ACM Symposium on Theory of Computing, 1997. 1997: 654-663.