**基于Springboot-Netty-Protostuff-ZooKeeper分布式RPC框架**

**目录**

**[目录](#_Toc20887)** [1](#_Toc20887)

[一、概述 2](#_Toc19261)

[二、RPC原理简介 3](#_Toc23296)

[三、序列化协议概述 5](#_Toc26883)

[3.1 XML-RPC，SOAP，WebService 5](#_Toc11459)

[3.2 PHPRPC 6](#_Toc26398)

[3.3 Hessian 6](#_Toc31338)

[3.4 JSON-RPC 6](#_Toc16348)

[3.5 Microsoft WCF，WebAPI 7](#_Toc24123)

[3.6 ZeroC Ice，Thrift，GRPC 7](#_Toc12525)

[3.7 Hprose 7](#_Toc409)

[3.8 protobuf 8](#_Toc32444)

[3.9 protostuff 8](#_Toc7571)

[三、项目模块关系图 9](#_Toc4925)

[四、项目核心代码讲解 10](#_Toc15586)

[4.1 Server端讲解 10](#_Toc23669)

[4.2 client 端讲解 11](#_Toc13029)

[4.3 common讲解 14](#_Toc12302)

[4.4 剩余模块简介 15](#_Toc5972)

[五、项目模块代码讲解及演示 15](#_Toc25776)

[3.1项目整体工程 15](#_Toc21703)

[3.2项目功能演示 18](#_Toc5876)

[参考资料 20](#_Toc2118)

# 一、概述

RPC（Remote Procedure Call Protocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

RPC 可基于 HTTP 或 TCP 协议，Web Service 就是基于 HTTP 协议的 RPC，它具有良好的跨平台性，但其性能却不如基于 TCP 协议的 RPC。会两方面会直接影响 RPC 的性能，一是传输方式，二是序列化。

众所周知，TCP 是传输层协议，HTTP 是应用层协议，而传输层较应用层更加底层，在数据传输方面，越底层越快，因此，在一般情况下，TCP 一定比 HTTP 快。就序列化而言，Java 提供了默认的序列化方式，但在高并发的情况下，这种方式将会带来一些性能上的瓶颈，于是市面上出现了一系列优秀的序列化框架，比如：Protobuf、Kryo、Hessian、Jackson 等，它们可以取代 Java 默认的序列化，从而提供更高效的性能。

为了支持高并发，传统的阻塞式 IO 显然不太合适，因此我们需要异步的 IO，即 NIO。Java 提供了 NIO 的解决方案，Java 7 也提供了更优秀的 NIO.2 支持，用 Java 实现 NIO 并不是遥不可及的事情，只是需要我们熟悉 NIO 的技术细节。

Netty是由JBOSS提供的一个java开源框架。Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。也就是说，Netty 是一个基于NIO的客户、服务器端编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户，服务端应用。Netty相当简化和流线化了网络应用的编程开发过程，例如，TCP和UDP的socket服务开发。

我们需要将服务部署在分布式环境下的不同节点上，通过服务注册的方式，让客户端来自动发现当前可用的服务，并调用这些服务。这需要一种服务注册表（Service Registry）的组件，让它来注册分布式环境下所有的服务地址（包括：主机名与端口号）。

根据以上技术需求，使用如下技术选型：

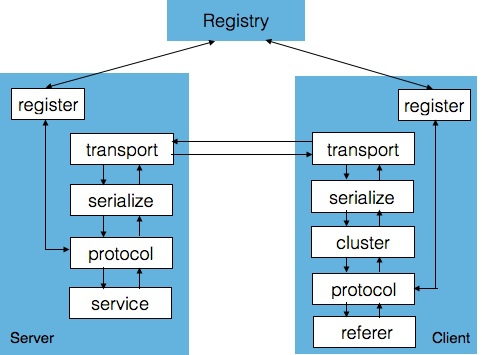
1. Spring：主流依赖注入框架。
2. Netty：它使 NIO [编程](http://www.2cto.com/kf" \t "_blank)更加容易，屏蔽了 Java 底层的 NIO 细节。
3. Protostuff：它基于 Protobuf 序列化框架，面向 POJO，无需编写 .proto 文件。
4. ZooKeeper：提供服务注册与发现功能，开发分布式[系统](http://www.2cto.com/os/" \t "_blank)的必备选择，同时它也具备集群能力。
5. Tomcat:业界比较流行的web容器。

# 二、RPC原理简介

RPC(Remote Procedure Call，远程过程调用)是建立在Socket之上的,出于一种类比的愿望,在一台机器上运行的主程序,可以调用另一台机器上准备好的子程序,就像LPC(本地过程调用).越底层，代码越复杂、灵活性越高、效率越高；越上层，抽象封装的越好、代码越简单、效率越差。

众所周知，传统过程调用模式无法充分利用网络上其他主机的资源（如CPU、Memory等），也无法提高代码在实体间的共享程度，使得主机资源大量浪费。通过RPC我们可以充分利用非共享内存的多处理器环境（例如通过局域网连接得多台工作站）,这样可以简便地将你的应用分布在多台工作站上，应用程序就像运行在一个多处理器的计算机上一样。你可以方便的实现过程代码共享，提高系统资源的利用率，也可以将以大量数值处理的操作放在处理能力较强的系统上运行，从而减轻前端机的负担。

RPC要做到用户无感知的调用远程服务必定要经过网络传输，而netty正好是是Java编写的快速开发高性能高可靠性的网络编程框架。目前使用netty作为传输层的RPC框架很多，国内知名的有dubbo，motan等。下面以新浪开源的motan为例，说明netty在RPC的位置：下图中transport即使使用netty实现的。



**分布式RPC需要解决哪些问题呢？**

1. **protocol：传输协议**
2. **proxy：client代理，服务引用方调用方法通过代理发送远程消息**
3. **codec：协议编解码压缩等**
4. **transport：协议传输**
5. **registry：注册中心，服务注册服务发现**
6. **cluster：负载均衡，服务容错策略**
7. **其他：服务降级，服务隔离，服务治理**

**如何实现一个分布式的RPC框架呢？**

现在互联网已经很少单点服务了，一个好的rpc框架能帮我们省掉很多工作，那么如何撸一个自己喜欢的RPC框架呢？

现在写rpc框架已经不难了，我们可以有很多参考，很多成熟的第三方开源项目可以依赖，可以说是站在巨人的肩膀上。

**第一步，选择传输协议：**

高性能的rpc和良好的编码协议是分不开的。好的协议不仅耗用流量小，而且序列化和反序列化更快。

现在比较流行的协议有thrift，protobuf，json，restful等。其中thrfit和protobuf性能都比较优异，而且占用空间小，最重要的是跨语言，具有语言平台无关性。但，美中不足的是他们都需要预先根据协议文件预先生成好代码。开发极不流畅。

我们也可以自定义协议，像dubbo、motan和五八的scf一样，定制自己的私有协议，

比如motan协议的header部分如下：

<img src="https://pic1.zhimg.com/v2-2225313e318b0e3b1649280126407e34\_b.png" data-rawwidth="1095" data-rawheight="415" class="origin\_image zh-lightbox-thumb" width="1095" data-original="https://pic1.zhimg.com/v2-2225313e318b0e3b1649280126407e34\_r.png">

body部分采用hession或者fastjson序列化，

如果不考虑跨语言，这种算是比较完美的解决方案。

定义协议过程中的一些重要且容易忽略的问题：

* 协议版本号
* 消息id
* 协议扩展字段

**第二步，协议传输层**

一个高性能RPC框架最重要的四个点就是：传输协议，框架线程模型，IO模型，零拷贝。

java程序如果能做好这四点，那么性能应该不会比c++程序差多少。

而作为java开发者，netty正好解决了后三个点，所以使用netty作为RPC框架的传输层会事半功倍。

**第三部，注册中心的选择**

现在有很多提供服务注册发现的服务，实现成本比较低就是zookeeper，可以很容易的实现服务注册和服务发现的功能。

解决了这三部，后面的就得脚踏实地码代码了，当然后续会有很多细节，不过都不是问题，现在有好多成熟的开源框架可以参考。

**本地调用和 RPC 调用的一些差异:**

1. 本地调用一定会执行，而远程调用则不一定，调用消息可能因为网络原因并未发送到服务方。

2. 本地调用只会抛出接口声明的异常，而远程调用还会跑出 RPC 框架运行时的其他异常。

3. 本地调用和远程调用的性能可能差距很大，这取决于 RPC 固有消耗所占的比重。

正是这些区别决定了使用 RPC 时需要更多考量。当调用远程接口抛出异常时，异常可能是一个业务异常，也可能是 RPC 框架抛出的运行时异常（如：网络中断等）。业务异常表明服务方已经执行了调用，可能因为某些原因导致未能正常执行，而 RPC 运行时异常则有可能服务方根本没有执行，对调用方而言的异常处理策略自然需要区分。

由于 RPC 固有的消耗相对本地调用高出几个数量级，本地调用的固有消耗是纳秒级，而 RPC 的固有消耗是在毫秒级。那么对于过于轻量的计算任务就并不合适导出远程接口由独立的进程提供服务，只有花在计算任务上时间远远高于 RPC 的固有消耗才值得导出为远程接口提供服务。

# 三、序列化协议概述

## 3.1 XML-RPC，SOAP，WebService

冗余数据太多，处理速度太慢。

RPC 风格的 Web Service 跨语言性不佳，而 Document 风格的 Web Service 又太过难用。

Web Service 没有解决用户的真正问题，只是把一个问题变成了另一个问题。

Web Service 的规范太过复杂，以至于在 .NET 和 Java 平台以外没有真正好用的实现，甚至没有可用的实现。

跨语言跨平台只是 Web Service 的一个口号，虽然很多人迷信这一点，但事实上它并没有真正实现。

## 3.2 PHPRPC

基于 PHP 内置的序列化格式，在跨语言的类型映射上存在硬伤。

通讯上依赖于 HTTP 协议，没有其它底层通讯方式的选择。

内置的加密传输既是特点，也是缺点。

虽然比基于 XML

的 RPC 速度快，但还不是足够快。

## 3.3 Hessian

二进制的数据格式完全不具有可读性。

官方只提供了两个半语言的实现（Java，ActionScript 和不怎么完美的 Python 实现），其它语言的第三方实现良莠不齐。

支持的语言不够多，对 Web 前端的 JavaScript 完全无视。

虽然是动态 RPC，但动态性仍然欠佳。

虽然比基于 XML 的 RPC 速度快，但还不是足够快。

## 3.4 JSON-RPC

JSON 具有文本可读性，且比 XML 更简洁。

JSON 受 JavaScript 语言子集的限制，可表示的数据类型不够多。

JSON 格式无法表示数据内的自引用，互引用和循环引用。

某些语言具有多种版本的实现，但在类型影射上没有统一标准，存在兼容性问题。

JSON-RPC 虽然有规范，但是却没有统一的实现。在不同语言中的各自实现存在兼容性问题，无法真正互通。

## 3.5 Microsoft WCF，WebAPI

它们是微软对已有技术的一个 .NET 平台上的统一封装，是对 .NET Remoting、WebService 和基于 JSON 、XML 等数据格式的 REST 风格的服务等技术的一个整合。

虽然号称可以在 .NET 平台以外来调用它的这些服务，但实际上跟在 .NET 平台内调用完全是两码事。它没有提供任何在其他平台的语言中可以使用的任何工具。

## 3.6 ZeroC Ice，Thrift，GRPC

初代 RPC 技术的跨语言面向对象的回归。

仍然需要通过中间语言来编写类型和接口定义。

仍然需要用代码生成器来将中间语言编写的类型和接口定义翻译成你所使用的编程语言的客户端和服务器端的占位程序（stub）。

你必须要基于生成的服务器代码来单独编写服务，而不能将已有代码直接作为服务发布。

你必须要用生成的客户端代码来调用服务，而没有其它更灵活的方式。

如果你的中间代码做了修改，以上所有步骤你都要至少重复一遍。

## 3.7 Hprose

无侵入式设计，不需要单独定义类型，不需要单独编写服务，已有代码可以直接发布为服务。

具有丰富的数据类型和完美的跨语言类型映射，支持自引用，互引用和循环引用数据。

支持众多传输方式，如 HTTP、TCP、Websocket 等。

客户端具有更灵活的调用方式，支持同步调用，异步调用，动态参数，可变参数，引用参数传递，多结果返回（Golang）等语言特征，Hprose 2.0 甚至支持推送。

具有良好的可扩展性，可以通过过滤器和中间件实现加密、压缩、缓存、代理等各种功能性扩展。

兼容的无差别跨语言调用

支持更多的常用语言和平台

支持浏览器端的跨域调用

没有中间语言，无需学习成本

性能卓越，使用简单

## 3.8 protobuf

google开发的开源的序列化方案protocol buffer（简称protobuf）它的好处很多，独立于语言，独立于平台，最最重要的是它的效率相当高，用protobuf序列化后的大小是json的10分之一，xml格式的20分之一，是二进制序列化的10分之一，protobuf使用起来非常简单，它的主要流程是：我们需要自己写一个.proto文件用来描述序列化的格式，然后用protobuf提供的protoc工具将.proto文件编译成一个[Java](http://lib.csdn.net/base/java" \t "_blank" \o "Java 知识库)文件（protobuf官方支持很多语言：Java、C++、C#、[Go](http://lib.csdn.net/base/go" \t "_blank" \o "Go知识库)、[Python](http://lib.csdn.net/base/python" \t "_blank" \o "Python知识库) ，protobuf是一个开源项目，因此有很多大牛也实现了其他语言，但它们的可靠性还有待验证），最后将该Java文件引入到我们的项目中就可以使用了，当然还得引入protobuf的依赖包。

## 3.9 protostuff

protostuff是一个基于protobuf实现的序列化方法，它较于protobuf最明显的好处是，在几乎不损耗性能的情况下做到了不用我们写.proto文件来实现序列化。使用它也非常简单。

Protostuff特征

1、支持protostuff-compiler产生的消息

2、支持现有的POJO

3、支持现有的protoc产生的Java消息

4、与各种移动平台的互操作能力（Android、Kindle、j2me）

5、支持转码

# 三、项目模块关系图

1.server端基于springboot启动，启动后根据配置信息自动去zk注册自身节点信息。

2.信息注册中如果有业务目录节点则直接添加数据节点，如果没有业务目录节点则新建永久目录节点。

3.sever端注册成功后，启动zk定时心跳监听，如果注册中心down掉后继续提供client服务，server会不断重试，直到注册中心reactive，注册中心恢复后如果session没有过期则继续使用数据节点提供服务，如果session过期后会重新新建zk数据节点并通过监听机制通知client端。

4.client端可以基于tomcat、jetty或者runnable-jar启动，client启动后会去zk上拉取服务节点信息并保存到本地。

5.每次服务请求，client随机从服务节点中选择一个提供服务。

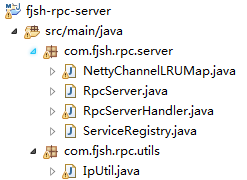
6.client会监听zk节点信息变化，如果zk注册节点信息有变化，会通知client进行更改，如果zk注册节点连接失败，client会通过保留到本地的服务节点继续提供服务，并且根据zk watch事件进行尝试重连，直到链接成功。

7.单次请求进行阻塞，数据返回后进行通知并设置超时时间，超时后请求抛弃。

8 在SerializationUtil中用户可以根据自己的需要定制化序列化方式，此处使用protostuff进行传输数据的编解码操作。

# 四、项目核心代码讲解

## 4.1 Server端讲解



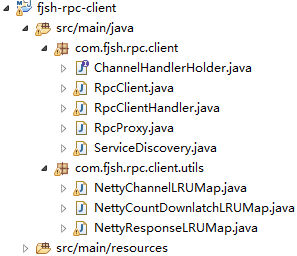
Server端主要负责服务的自动注册，与注册中心的心跳检查以及对请求的处理工作。

ServiceRegistry：负责服务在注册中心的注册以及服务的心跳检查，如果发现注册中心出现异常，则会不断重试，直到注册中心恢复，注册中心恢复后会出现两种情况一种是session过期一种是session非过期。对于session非过期的情况则继续使用之前注册的服务节点，对于session过期的情况，则需要重新注册新的服务节点，然后通过注册中心通知client端。

RpcServer：主要负责serverbootstrap的配置启动，并通过RpcServerHandler响应事件处理。其中核心代码如下：



## 4.2 client 端讲解



Client端主要负责请求数据的封装，在注册中心发现数据服务节点，定时心跳检查并基于观察者模式监听zk注册中心的数据变化，服务端返回数据的处理。

ServiceDiscovery：主要负责在注册中心获取服务节点并监听注册中心服务节点的变化，如有变化及时更新本地缓存的服务节点信息。

RpcProxy：主要负责对请求参数的封装，将前端请求的对象、函数以及参数等信息进行封装，并将封装好的数据传输给底层的RpcClient然后发送给服务节点，然后基于服务节点返回的数据重新解析成客户端可以直接调用的数据对象。其中核心代码如下：



RpcClient：主要负责client端bootstrap的启动以及客户端请求往服务节点的数据发送与接收。其中核心代码如下：

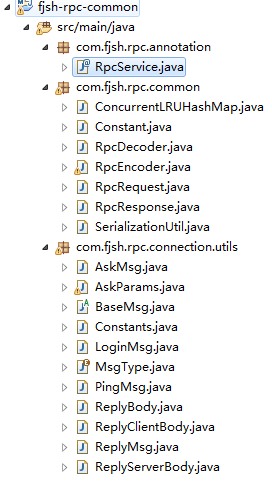




RpcClientHandler：主要负责对服务返回信息的处理，在这加入了定时任务，如果发现服务断开会进行不断尝试重连（细节尚待改进）。

NettyChannelLRUMap、NettyCountDownlatchLRUMap、NettyResponseLRUMap：主要是基于ConcurrentLRUHashMap实现的基于LRU淘汰策略的ConcurrentHashMap。

## 4.3 common讲解



Common包是公共包，主要负责通信的编解码、解决粘包拆包问题以及基础公用信息的定义。

RpcEncoder、RpcDecoder主要负责对传输数据进行编码解码。其中在解码中通过获取头部int型数据判断后面需要读取的长度，并通过判断可以读取的字节数来觉得是否进行此次的处理。其中核心代码如下：



SerializationUtil：主要负责数据的序列化与反序列化，此处基于protostuff进行数据的编解码，如果需要实现自己的序列化方式只需要修改此处即可。

ConcurrentLRUHashMap：是基于ConcurrentHashMap融入LRU淘汰策略实现的数据过期机制，在整体项目中channel的缓存以及返回数据的保存都基于此结构。

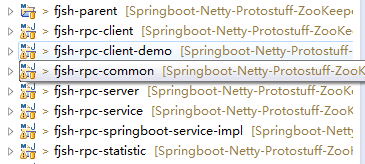
## 4.4 剩余模块简介

Client-demo模块是基于springmvc开发的简单web工程。

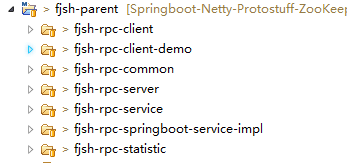
Service模块主要负责接口定义。

# 五、项目模块代码讲解及演示

## 3.1项目整体工程

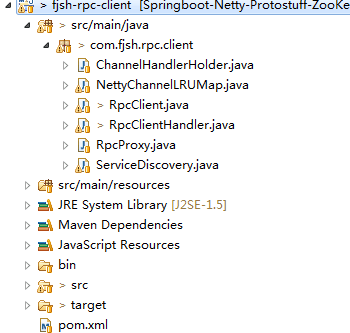


fjsh-parent:



此模块是整体项目父工程，所有子工程依赖父工程以及父工程中的jar包，可以防止jar包冲突以及版本问题。

fjsh-rpc-client:

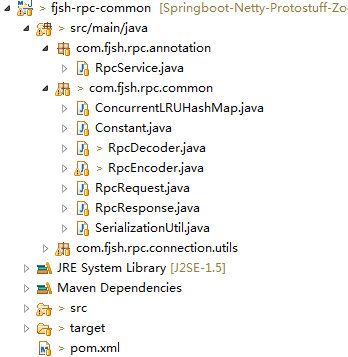


负责client端对服务的调用及访问。

RpcProxy负责业务端调用远程服务的参数封装。

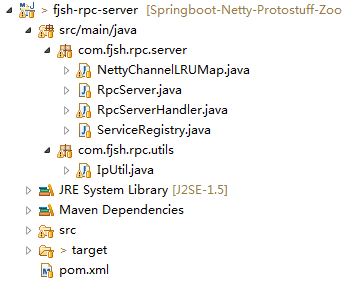
ServiceDiscovery负责服务发现以及对服务的监听。

Fjsh-rpc-common：



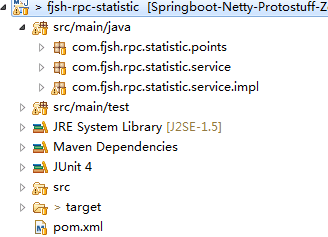
是此框架中的通用工具包，包括编解码以及序列化等工作。

fjsh-rpc-server:



此模块负责服务端的服务注册，心跳检查以及注册中心失败后的服务重连等。

fjsh-rpc-Statistic



此模块是工程中的性能统计模块，是开发中的辅助模块，可以通过此模块来监听请求流程来快速定位工程中的性能等问题。

## 3.2项目功能演示

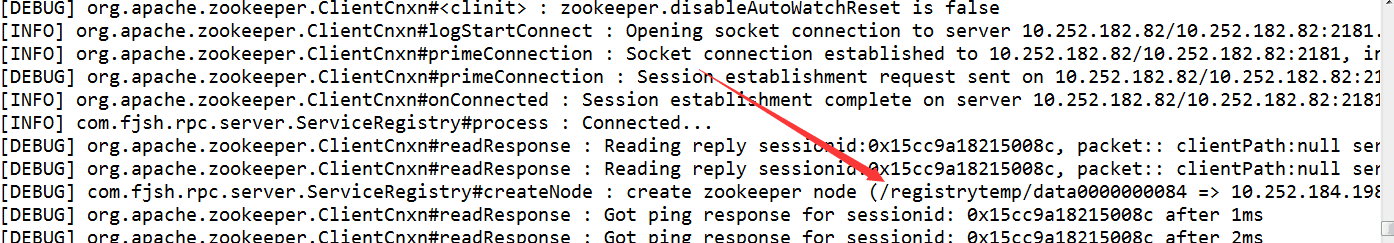
正常一次请求处理。

系统启动之前，查看zk中数据节点信息：

C:\Users\58\Documents\Tencent Files\925907195\Image\C2C\56~2KQP69FTXQOM})T4C0HM.png

此时可以看到注册目录下没有数据节点

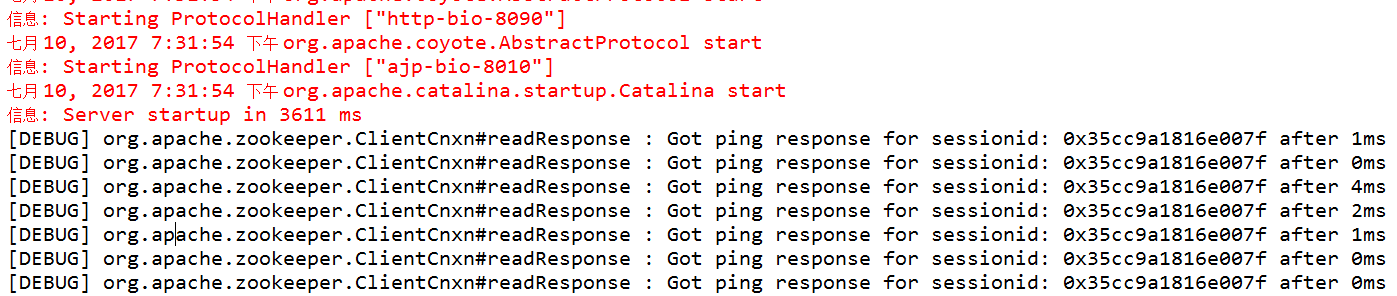
当启动服务端系统后，自动注册服务到zk，



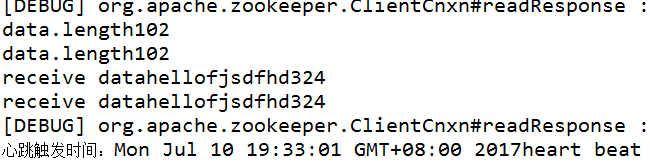
C:\Users\58\Documents\Tencent Files\925907195\Image\C2C\{698N)3)XF7G1{A_6TC$G9C.png

通过上面两图可以看到server端服务启动后自动注册，然后zk上可以直接查看到对应的服务节点。

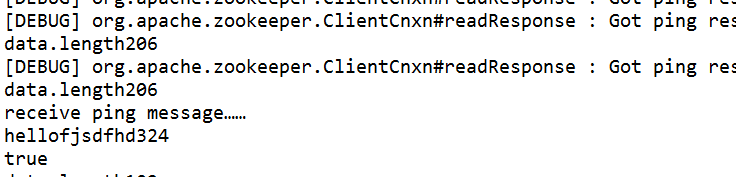
Client端启动后



这是client发送消息后收到的返回信息

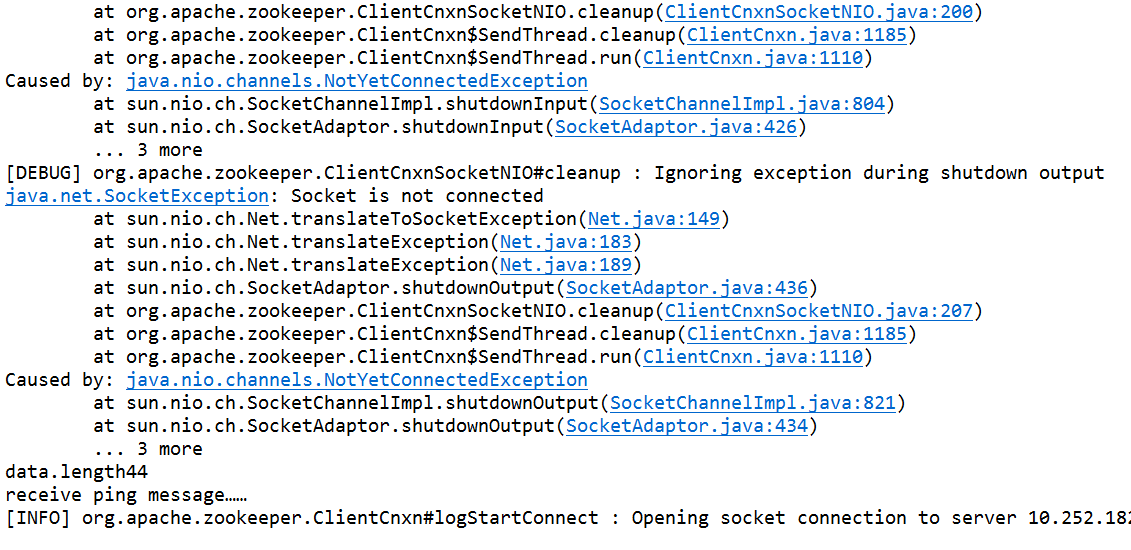


这是服务端收到的client发送来的信息，同时也可以看到链路联通后的定时client端到服务端的定时心跳信息，在这里有一个问题说明是client可以向服务端发送心跳但没必要服务端再给client发送心跳，这样会额外增加服务器的压力而且一旦client断开，服务端还要处理异常情况压力更大。



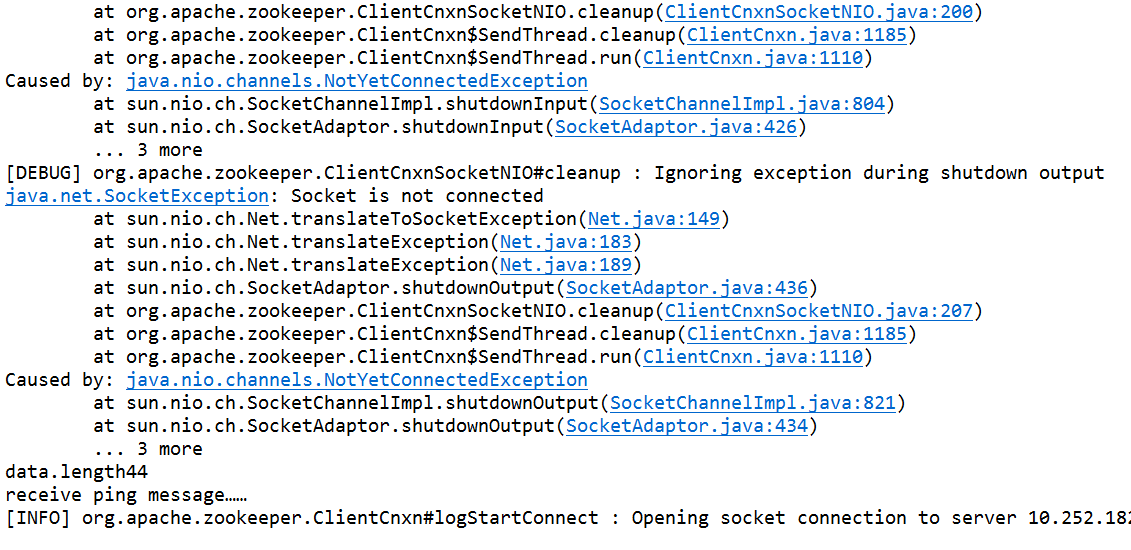
下面模拟下注册中心失败的情况

将注册中心关闭后



通过上图可以看到zk通信发生异常，但是依然能够收到从客户端发来的心跳检查信息，说明虽然注册中心失败，但client跟server之间的通信依然畅通。

下面重启zk注册中心



通过上图可以看到重启注册中心后，服务端恢复了与注册中心之间的链接，同时观察注册中心的节点可以看出，因为session过期，重新注册了新的服务节点，但依然不影响client与server之间的通信。

C:\Users\58\Documents\Tencent Files\925907195\Image\C2C\[K39]K}LEQ[%88UXL}(4AOX.png

# 参考资料

基于protostuff进行序列化与反序列化

<http://www.cnblogs.com/xiaoMzjm/p/4555209.html>

netty粘包拆包问题解决

http://blog.csdn.net/top\_code/article/details/50893302

长连接机制

<https://my.oschina.net/robinyao/blog/399060>

ConcurrentLRUHashMap

<http://blogread.cn/it/article/4266?f=wb>

HashedWheelTimer 介绍

<https://my.oschina.net/haogrgr/blog/489320>

项目github地址：

一次请求一次链接

<https://github.com/a925907195/Springboot-Netty-Protostuff-ZooKeeper/tree/master/fjsh-parent>

长连接

<https://github.com/a925907195/Springboot-Netty-Protostuff-ZooKeeper-asyn>

如果谁有时间或者喜欢可以提些意见或者参与到项目中来。谢谢！