

W

手写RPC框架



罗曼-罗兰说过的，这个世上只有一种真正的英雄主义，那就是认清生活的真相，并且仍然热爱它。

RPC架构



技术选型

网络传输

(为了调用远程方法，就是需要发送网络请求来传递目标类和方法信息以及方法的参数到服务提供方)

- bio
- nio
- netty

序列化

(编写网络应用程序的时候，因为数据在网络中传输的都是二进制字节码数据，在发送数据时需要编码，在接收数据时需要解码)

把对象转换成字节数组很容易，但是如果没有相应规则就不能将其转换回对象，所以以下就是含有相应规则的协议方法

如何选择序列化：协议是否支持跨平、序列化的速度、序列化生成的体积。

- **java自带的序列化** (之前使用过 这个的话，不支持跨语言平台 同时性能比较差 序列化后的体积比较大)
- **json(jackson、fastjson、gson)** [fastjson、Jackson](#)以及Gson序列化对象与get以及对象属性之间的关系
- **Thrift** Thrift源于faceBook 由于该序列化框架需要相应编译器生成代码和protoc类似 就不进行测试了 //之后有机会尝试
- **kryo** [深入理解RPC之序列化篇--Kryo - EsotericSoftware/kryo: Java](#)
- **protobuf**
- **protostuff** (基于protobuf的java序列化协议——prorostuff) [学习java序列化机制之protoStuff](#)
- **hessian**[Caucho](<https://caucho.com/>)
- **FST** [高性能序列化框架FST - 成长源于沉淀](#) 较新的一款序列化工具还没有得到很广泛的使用
- **Avro** 也是和protobuf类似 也是需要额外编写对应的类再编译，为了项目的完整性，暂时不采用，之后有机会尝试

JDK原生、ProtoBuf (Protostuff) 、Hessian、Kryo、Json的区别

[Hessian、Kryo、Protostuff序列化使用区别-hessian和protobuf](#)

[序列化的理解](#)

代理

(一开始我也不知道代理有什么用，直接把业务逻辑代码写在那边不就行了吗，但是后面听了韩顺平老师的课后发现，动态代理的目的就是让对象能像调用自己方法一样，调用远端的方法，相当于对于服务调用者是一个黑盒的状态)

- **静态代理**
- **动态代理JDK** (实现有实现接口的类) 通过反射
- **动态代理CGLIB** (CGLIB可以实现接口，也可以实现类) 通过修改字节码生成子类

注册中心

(服务端启动的时候将服务名+服务地址+服务端口注册 然后客户端进行调用的时候 就通过查到相应服务的地址进行调用--相当于目录) [搭建注册中心有两种方案第一种就是根据我的](#)

- 在安装好docker的linux系统上启动docker后可以使用拉取我的镜像一键构造 同时启动zookeeper和nacos

- 指令

```
# 拉取镜像
docker pull 836585692/zytregistry:1.0
```

```
# 启动
docker run -it -d -p 8848:8848 -p 2181:2181 --restart always
836585692/zytregistry:1.0
```

- 一键启动

```
docker pull 836585692/zytregistry:1.0&&docker run -it -d -p 8848:8848 -p
2181:2181 --restart always 836585692/zytregistry:1.0
```

以下自己独立构建

- **zookeeper** (试试放在linux的docker里玩下) (用zookeeper和curator分别来实现 注册中心) 打一个指令让用户一键启动zk集群
 - 目的在一台虚拟机中装多个带zookeeper的centos镜像, 这样就不用启动过多的虚拟机了。这是后话, 目前来说并不需要集群, 先都注册到一个zookeeper上
 - **zookeeper的安装**, 可以直接在linux下安装, 我选择在linux的docker下安装, 以下是安装步骤 (暂时没有用到集群部署所以不对其配置进行修改)

```
#拉取最新版zookeeper镜像
[root@zytCentos ~]# docker pull zookeeper
Using default tag: latest
latest: Pulling from library/zookeeper
1fe172e4850f: Pull complete
44d3aa8d0766: Pull complete
fda2f211fa11: Pull complete
9154e8aefca7: Pull complete
692fa00e9b62: Pull complete
a17c9a9b7c3c: Pull complete
bd7073bfcd08: Pull complete
0e2c8094a504: Pull complete
Digest: sha256:1e62d091501c7125293c087414fccff4337c08d203b92988f4d88081b4f1f63e
Status: Downloaded newer image for zookeeper:latest
docker.io/library/zookeeper:latest
```

- **zookeeper的运行**

保持zookeeper的开启状态, 由于我之前启动没有加-d所以在执行连接它的命令时总会自动关闭, 加了的目的是保证它能保持后台运行, --restart always这个指令的目的就是之后不用每次都再启动了

```
[root@zytCentos ~]# docker run -d -p 2181:2181 --name zytzookeeper --restart always zookeeper
```

o 尝试用本地idea连接启动的服务端

- 添加依赖 zookeeper原生客户端和一个更加便捷开发的客户端Curator

```
<!--Zookeeper的依赖 客户端-->
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.5.7</version>
</dependency>
<!--顺带引入的更便捷操作的curator的依赖 Curator是Netflix公司开源的一个
Zookeeper客户端 简化了原生的开发-->
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>4.3.0</version>
</dependency>
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>4.3.0</version>
</dependency>
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-client</artifactId>
    <version>4.3.0</version>
</dependency>
```

- 进行测试开发 是否有效

首先前提我为了方便起见设置了域名映射（想偷懒的小伙伴可以尝试）

此电脑 > 本地磁盘 (C:) > Windows > System32 > drivers > etc

名称	修改日期	类型	大小
hosts	2022/4/5 16:29	文件	1 KB
lmhosts.sam	2019/12/7 17:12	SAM 文件	4 KB
networks	2019/12/7 17:12	文件	1 KB
protocol	2019/12/7 17:12	文件	2 KB
services	2019/12/7 17:12	文件	18 KB

```
# localhost name resolution is handled within DNS itself.
# 127.0.0.1    localhost
# ::1        localhost
192.168.18.128 zytCentos
192.168.18.130 zytCentosClone1
192.168.18.131 zytCentosClone2
```

接下来就是尝试去连接了

```
private String connectString = "zjtCentos:2181";
private int sessionTimeout = 2000;
@Test
public void connect() throws IOException, InterruptedException, KeeperException {
    ZooKeeper zooKeeper = new ZooKeeper(
        connectString, //连接地址 如果写多个那会
        sessionTimeout, //即超过该时间后, 客户端没有向服务器端发送任何请求(正常情况下客户端会每隔一段时间发送心跳请求, 此时服务器端会重新计算客户端的超时时间点的),
        null //则服务器端认为session超时, 清理数据, 此时客户端的ZooKeeper对象就不起作用了, 需要再重新new一个新的对象了。
    );
    zooKeeper.create( path: "/test", "12".getBytes(StandardCharsets.UTF_8), ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
}
```

创建成功

```
[zk: localhost:2181(CONNECTED) 33] ls /
[test0000000002, zookeeper]
```

测试成功 进行设置

• Nacos

自己去尝试, 使用 **nacos快速开始采用的都是spring亦或者是springboot, 我用的不是这些框架。**

- 在dockerHub上拉取镜像

```
[root@zjtCentos ~]# docker pull nacos/nacos-server:v2.0.3
```

- 根据镜像创建容器

```
[root@zjtCentos ~]# docker run --name nacos-quick -e MODE=standalone -p 8848:8848 -d --restart always bdf60dc2ada3
```

- --name 容器名为nacos-quick
- -e 启动环境模式
- -p 是端口映射
- -d 后台方式运行
- --restart always 自动启动

- 启动成功后界面



- 配置依赖

```
<!--添加Nacos依赖 使用服务注册 服务发现-->
<dependency>
    <groupId>com.alibaba.nacos</groupId>
    <artifactId>nacos-spring-context</artifactId>
    <version>1.1.1</version>
</dependency>
```

- **HttpClient**

HttpClient 是Apache Jakarta Common 下的子项目，可以用来提供高效的、最新的、功能丰富的支持 HTTP 协议的客户端编程工具包，并且它支持 HTTP 协议最新的版本和建议。

传输协议

(传输协议的作用 就是我们发送的信息 要按照我们自己的规定构造 相当于密文传输的感觉 让别人不知道在发送什么)

-

负载均衡

(防止访问量过大，可以将请求分到其他服务提供方上，减少宕机、崩溃的风险)

- 自己代码实现
- zookeeper本身就能实现软负载均衡吧 可以在zookeeper中记录每台服务器的访问次数，让访问最少的去处理最新的客户端请求
- 随机负载均衡
- 一致性哈希

数据压缩

(减少传输时的数据量)[JAVA中实现数据压缩所有方式](#)

不同的工具有不同的压缩率 同时不一定会变小，如果是对小体积文件进行压缩的话，一些压缩的相关信息加进去反而变大，一般如果有大的对象传输可以开启压缩机制

[Java不同压缩算法的性能比较](#)

- bzip
- deflater
- gzip
- lz4
- zip
- 自己根据所学的霍夫曼编码实现的diyZip

其他机制

- 心跳机制
- 解决粘包、拆包问题

- 注解开发等等
- SPI机制 [Java SPI 机制及其实现](#)

RPC框架v1.0NIO版

要求简单实现远程调用

用最笨的方法实现

技术选型

后续进行了大量更新

- 网络传输: nio
- 序列化: java自带序列化

客户端

消费者启动端

```
package consumer.bootstrap;

import consumer.nio.NIOClient;

import java.io.IOException;

/*
    以nio为网络编程框架的消费者端启动类
*/
public class NIOConsumerBootStrap {
    public static void main(String[] args) throws IOException {
        NIOClient.start("127.0.0.1",6666);
    }
}
```

消费者实际业务端

```
package consumer.nio;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
```

```

import java.nio.charset.StandardCharsets;
import java.util.Iterator;
import java.util.Scanner;

public class NIOClient {
    public static void start(String HostName, int PORT) throws IOException{
        start0(HostName,PORT);
    }

    //真正启动在这
    private static void start0(String hostName, int port) throws IOException {
        //得到一个网络通道
        SocketChannel socketChannel = SocketChannel.open();
        System.out.println("-----服务消费方启动-----");
        socketChannel.configureBlocking(false);
        //建立链接 非阻塞连接 但我们是要等他连接上
        if (!socketChannel.connect(new InetSocketAddress(hostName,port))) {
            while (!socketChannel.finishConnect());
        }
        //创建选择器 进行监听读事件
        Selector selector = Selector.open();
        socketChannel.register(selector, SelectionKey.OP_READ,
ByteBuffer.allocate(1024));
        //创建匿名线程进行监听读事件
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true)
                {
                    //捕获异常 监听读事件
                    try {
                        if (selector.select(1000)==0)
                        {
                            continue;
                        }
                        Iterator<SelectionKey> keyIterator =
selector.selectedKeys().iterator();
                        while (keyIterator.hasNext())
                        {
                            SelectionKey key = keyIterator.next();
                            ByteBuffer buffer = (ByteBuffer)key.attachment();
                            SocketChannel channel = (SocketChannel)key.channel();
                            int read = 1;
                            //用这个的原因是怕 多线程出现影响
                            StringBuffer stringBuffer = new StringBuffer();
                            while (read!=0)
                            {
                                buffer.clear();
                                read = channel.read(buffer);
                                stringBuffer.append(new String(buffer.array(),0,read));

```



```

        }
        System.out.println("收到服务端回信"+stringBuffer.toString());
        keyIterator.remove();
    }
} catch (IOException e) {
    e.printStackTrace();
}
}
}
}).start();

//真正的业务逻辑 等待键盘上的输入 进行发送信息
Scanner scanner = new Scanner(System.in);
while (true)
{
    int methodNum = scanner.nextInt();
    String message = scanner.next();
    String msg = new String(methodNum+"#" +message);
    socketChannel.write(ByteBuffer.wrap(msg.getBytes(StandardCharsets.UTF_8)));
    System.out.println("消息发送");
}
}
}

```

服务端

服务提供者启动端

```

package provider.bootstrap;

import provider.nio.NIOServer;

import java.io.IOException;

/*
    以nio为网络编程框架的服务提供端启动类
*/
public class NIOProviderBootstrap {
    public static void main(String[] args) throws IOException {
        NIOServer.start(6666);
    }
}

```

服务提供者实际业务端

```

package provider.nio;

import api.ByeService;
import api.HelloService;
import provider.api.ByeServiceImpl;
import provider.api.HelloServiceImpl;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.nio.charset.StandardCharsets;
import java.util.Iterator;
import java.util.Set;

public class NIOServer {

    //启动
    public static void start(int PORT) throws IOException {
        start0(PORT);
    }

    //TODO 当服务消费方下机时 保持开启状态

    /*
    真正启动的业务逻辑在这
    因为这是简易版 那么先把异常丢出去
    */
    private static void start0(int port) throws IOException {
        //创建对应的服务器端通道
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        System.out.println("-----服务提供方启动-----");
        //开启一个选择器 将自己要
        Selector selector = Selector.open();

        //绑定端口开启
        serverSocketChannel.bind(new InetSocketAddress(port));

        //这里注意 要设置非阻塞 阻塞的话 他会一直等待事件或者是异常抛出的时候才会继续 会浪费cpu
        serverSocketChannel.configureBlocking(false);

        //要先设置非阻塞 再注册 如果时先注册再设置非阻塞会报错
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        //真正的业务逻辑 就是下面
        //循环等待客户端的连接和检查事件的发生
        while (true)
        {
            //1秒钟无事发生的话 就继续
            if (selector.select(1000)==0)

```

```

{
    continue;
}

//获取所有的对象
Set<SelectionKey> selectionKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectionKeys.iterator();

while (keyIterator.hasNext())
{
    SelectionKey key = keyIterator.next();
    if (key.isAcceptable())
    {
        SocketChannel socketChannel = serverSocketChannel.accept();
        System.out.println("连接到消费
端"+socketChannel.socket().getRemoteSocketAddress());
        socketChannel.configureBlocking(false);
        socketChannel.register(selector, SelectionKey.OP_READ,
ByteBuffer.allocate(1024));
    }
    if (key.isReadable())
    {
        //反向获取管道
        SocketChannel socketChannel = (SocketChannel)key.channel();
        //反向获取Buffer
        ByteBuffer buffer = (ByteBuffer)key.attachment();
        //进行调用方法并返回
        //获得信息
        StringBuffer stringBuffer = new StringBuffer();
        int read = 1;
        while (read!=0)
        {
            //先清空 防止残留
            buffer.clear();
            read = socketChannel.read(buffer);
            //添加的时候 根据读入的数据进行
            stringBuffer.append(new String(buffer.array(),0,read));
        }
        //方法号和信息中间有个#进行分割
        String msg = stringBuffer.toString();
        String[] strings = msg.split("#");
        String response;
        if (strings.length<2)
        {
            //当出现传入错误的时候 报异常
            System.out.println("传入错误");
            throw new RuntimeException();
        }
        if (strings[0].equals("1"))
        {

```

```

        HelloService helloService = new HelloServiceImpl();
        response = helloService.sayHello(strings[1]);
    }
    else if (strings[0].equals("2"))
    {
        ByeService byeService = new ByeServiceImpl();
        response = byeService.sayBye(strings[1]);
    }
    else
    {
        //当出现传入错误的时候 报异常
        System.out.println("传入错误");
        throw new RuntimeException();
    }
    String responseMsg = "收到信息" + strings[1] + "来自" +
socketChannel.socket().getRemoteSocketAddress();
    System.out.println(responseMsg);
    //将调用方法后获得的信息回显
    ByteBuffer responseBuffer =
ByteBuffer.wrap(response.getBytes(StandardCharsets.UTF_8));
    //写回信息
    socketChannel.write(responseBuffer);
}
keyIterator.remove();
}
}
}
}
}

```

序列化

JDK自带序列化

依赖引入

最外层pom引入依赖

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>coder.zyt</groupId>

```

```
<artifactId>zeng-rpc-framework</artifactId>
<packaging>pom</packaging>
<version>1.0-SNAPSHOT</version>
<modules>
  <module>zyt-rpc-consumer</module>
  <module>zyt-rpc-provider</module>
  <module>zyt-rpc-api</module>
  <module>zyt-rpc-common</module>
</modules>

<properties>
  <maven.compiler.source>8</maven.compiler.source>
  <maven.compiler.target>8</maven.compiler.target>
</properties>
<dependencies>
  <!--方便构建类-->
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.22</version>
  </dependency>
  <!--打印日志信息-->
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.25</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.25</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.25</version>
    <scope>test</scope>
  </dependency>
  <!--测试-->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>2.5.1</version>
  </dependency>
```

```

<!--Zookeeper的依赖 客户端-->
<dependency>
  <groupId>org.apache.zookeeper</groupId>
  <artifactId>zookeeper</artifactId>
  <version>3.5.7</version>
</dependency>
<!--顺带引入的更便捷操作的curator的依赖 Curator是Netflix公司开源的一个Zookeeper客户端 简化了原生的开发-->
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-framework</artifactId>
  <version>4.3.0</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>4.3.0</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-client</artifactId>
  <version>4.3.0</version>
</dependency>

</dependencies>
</project>

```

更新

v1.1

更新事项

- 解决客户端断开连接后 服务器端也会强制下线的问题
 - 原因：当客户端断开连接后 服务端的select会监听到事件 isReadable()不仅会监听到读事件还会监听到玩家下线的事件。
 - 解决方案：在读事件捕获异常 优雅的关闭管道 下面是代码实现

```

try{
    //之前的业务逻辑
}
catch (IOException e) {
    //进行关闭 并继续执行 取消键的注册 还有关闭管道
    SocketChannel unConnectChannel = (SocketChannel)key.channel();

    System.out.println(((unConnectChannel.socket().getRemoteSocketAddress())+"下线了"));
    key.cancel();
    unConnectChannel.close();
} catch (RuntimeException e) {
    e.printStackTrace();
}

```

- 解决传输时 可能出现的粘包拆包问题

```

1
2dsad
2
CSXZCZ
1
CXZCZ
消息发送
消息发送
消息发送
收到服务端回信Hello, 2dsad2

```

- 原因：TCP是一个“流”协议，是没有界限的一串数据，TCP底层并不了解上层业务数据的具体含义，它会根据TCP缓冲区的实际情况进行包的划分，所以在业务上认为，一个完整的包可能会被TCP拆成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的TCP粘包和拆包问题。
- 解决方案：设置一个标识符 读到标识符时暂停 更换自己的消息发送方式 通过channel 现在改成io 非阻塞的网络io读写只能用到read和write但这个就是只能使用 阻塞的nio了 额外开了一个包 实现 下面是代码实现
- 尝试使用阻塞io解决粘包问题

```

//新建请求发送类
package entity;

import lombok.AllArgsConstructor;
import lombok.Data;

```

```

import lombok.NoArgsConstructor;

//网络传输请求 重点是要实现序列化 否则不能进行io传输
@Data
@AllArgsConstructor
@NoArgsConstructor
public class RpcRequest implements Serializable{
    //方法编号
    int methodNum;
    //消息体
    String message;
}

```

```

//消费者端
package consumer.nio;
import entity.RpcRequest;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetSocketAddress;
import java.nio.channels.SocketChannel;
import java.util.Scanner;
//阻塞NIO消费端 解决沾包问题
public class NIOBlockingClient {
    public static void start(String HostName, int PORT) throws IOException {
        start0(HostName, PORT);
    }

    //真正启动在这
    private static void start0(String hostName, int port) throws IOException {
        //得到一个网络通道
        SocketChannel socketChannel = SocketChannel.open();
        System.out.println("-----服务消费方启动-----");
        //设置阻塞
        socketChannel.configureBlocking(true);
        //建立链接 阻塞连接 但我们要等他连接上
        socketChannel.connect(new InetSocketAddress(hostName, port));

        //真正的业务逻辑 等待键盘上的输入 进行发送信息
        Scanner scanner = new Scanner(System.in);

        //输入输出通道都放在外面

        ObjectOutputStream outputStream = new
        ObjectOutputStream(socketChannel.socket().getOutputStream());
        ObjectInputStream objectInputStream = new
        ObjectInputStream(socketChannel.socket().getInputStream());
        //都是阻塞等待 发完了 接收完了 才能进行下一步 不然会报异常
        while (true)

```



```

    {
        int methodNum = scanner.nextInt();
        String message = scanner.next();
        RpcRequest request = new RpcRequest(methodNum,message);
        //进行修订 使得可以传送对象 通过自带的io流进行 避免出现沾包拆包现象

        outputStream.writeObject(request);
        System.out.println("消息发送");
        try {

            String msg = (String)objectInputStream.readObject();
            System.out.println("收到来自客户端的消息"+msg);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
}
}

```

//NIO非阻塞服务提供方 主要的代码

```

package provider.nio;

import api.ByeService;
import api.HelloService;
import entity.RpcRequest;
import provider.api.ByeServiceImpl;
import provider.api.HelloServiceImpl;

import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;

//阻塞NIO服务提供端 解决沾包问题
public class NIOBlockingServer {
    //启动
    public static void start(int PORT) throws IOException {
        start0(PORT);
    }
    //TODO 当服务消费方下机时 保持开启状态

    /*
        真正启动的业务逻辑在这
        因为这是简易版 那么先把异常丢出去
    */
    private static void start0(int port) throws IOException {

```

```

//创建对应的服务器端通道
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
System.out.println("-----服务提供方启动-----");
//开启一个选择器 将自己要
Selector selector = Selector.open();

//绑定端口开启
serverSocketChannel.bind(new InetSocketAddress(port));

//设置阻塞
serverSocketChannel.configureBlocking(true);

//真正的业务逻辑 就是下面
//循环等待客户端的连接和检查事件的发生
while (true)
{
    SocketChannel channel = serverSocketChannel.accept();
    System.out.println("来自"+channel.socket().getRemoteSocketAddress()+"的连接");
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                //在内部不断的进行监听
                InputStream inputStream = channel.socket().getInputStream();
                OutputStream outputStream = channel.socket().getOutputStream();
                ObjectInputStream objectInputStream = new
ObjectInputStream(inputStream);
                ObjectOutputStream objectOutputStream = new
ObjectOutputStream(outputStream);
                while (true)
                {
                    String response;
                    RpcRequest request =
(RpcRequest)objectInputStream.readObject();
                    if (request.getMethodNum()==1)
                    {
                        HelloService helloService = new HelloServiceImpl();
                        response = helloService.sayHello(request.getMessage());
                    }
                    else if (request.getMethodNum()==2)
                    {
                        ByeService helloService = new ByeServiceImpl();
                        response = helloService.sayBye(request.getMessage());
                    }
                    else
                    {
                        System.out.println("传入错误");
                        throw new RuntimeException();
                    }
                }
            }
        }
    })
}

```

```

        System.out.println("收到客户
端"+channel.socket().getRemoteSocketAddress()+"的消息"+response);
        ObjectOutputStream.writeObject(response);
    }
} catch (Exception e) {

System.out.println("channel"+channel.socket().getRemoteSocketAddress()+"断开连接");
    try {
        channel.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
}
}).start();
}
}
}

```

v1.2

更新事项 以下更新均在非阻塞模块进行更新，阻塞模块可供读者自己尝试

- **进一步减少用户使用的复杂感** 之前分辨方法是通过判断传递的字符串、类型中的方法编号字段来抉择的，这次的话，将更加精简，使用注册中心，**目前每个服务提供者每人只提供一个服务，其他的日后再完善**
 - 首先我做的是将zookeeper的地址设置为常量这样，之后每次在创建zookeeper连接中地址参数可以直接拿，同时修改起来也方便，把sessionTimeout也设为常量

```

package constants;

public class RpcConstants {
    //zookeeper服务器连接地址
    public static String ZOOKEEPER_ADDRESS = "zytCentos:2181";
    //超时时间
    public static int ZOOKEEPER_SESSION_TIMEOUT = 2000;
}

```

- 实现服务提供端将对应地址注册到zookeeper中

```

package zkService;

import constants.RpcConstants;
import org.apache.zookeeper.*;
import org.apache.zookeeper.data.Stat;

import java.io.IOException;

```

```

import java.nio.charset.StandardCharsets;

//该类将对应服务端的方法和相应的端口和地址，注册到zookeeper中
public class ZkServiceRegistry {
    private static String connectString = RpcConstants.ZOOKEEPER_ADDRESS;
    private static int sessionTimeout = RpcConstants.ZOOKEEPER_SESSION_TIMEOUT;
    private static ZooKeeper zooKeeper;

    static void createConnect() throws IOException {
        zooKeeper = new ZooKeeper(connectString, sessionTimeout, new Watcher() {
            @Override
            public void process(WatchedEvent watchedEvent) {

            }
        });
    }

    //创建成功后把方法注册进去
    static void register(String RpcServiceName,String hostname,int port) throws
    InterruptedException, KeeperException {
        //节点名就是方法名 然后对应的数据就是hostname+": "+port

        //因为这个地区属于一个临界区 可能会发生线程不安全问题 所以进行上🔒
        synchronized (ZkServiceRegistry.class) {
            Stat exists = zooKeeper.exists("/service", null);
            if (exists ==null) {
                zooKeeper.create("/service",
                    "".getBytes(StandardCharsets.UTF_8),
                    ZooDefs.Ids.OPEN_ACL_UNSAFE,
                    CreateMode.PERSISTENT
                );
            }
        }

        String date = hostname+": "+port;

        //权限目前都设置为全放开 创建方式均为持久化
        zooKeeper.create("/service/"+RpcServiceName,
            date.getBytes(StandardCharsets.UTF_8),
            ZooDefs.Ids.OPEN_ACL_UNSAFE,
            CreateMode.PERSISTENT
        );
    }

    /**
     *
     * @param RpcServiceName 这是对应的服务名
     * @param hostname 和可以调用该服务的ip

```

```

    * @param port 还有对应的端口号
    * @throws IOException
    * @throws InterruptedException
    */
    public static void registerMethod(String RpcServiceName,String hostname,int
port) throws IOException, InterruptedException, KeeperException {
        //先创建对应的zookeeper连接客户端再进行相应的注册
        createConnect();
        register(RpcServiceName,hostname,port);
        System.out.println("服务端:"+hostname+": "+port+"方法注册完毕");
    }
}

```

进行测试

存在问题 测试出现问题 启动两个服务只有一个服务的方法注册进去了;

解决方法 因为我每个服务提供方到后面都是循环着监听 所以第一个进去了就出不来立刻 开了多个线程

存在问题 就是会重复进行创建service节点 应该是引入了线程安全的问题 就是单例模式懒汉模式1中类似的问题

解决方法 对方法进行加锁 使用了synchronized锁, 该锁jdk1.6之后进行了优化可以使用!

成功解决

- 服务提供方启动代码如下 (遇到了线程安全问题 之后尝试如何进一步优化)

```

/*
    以nio为网络编程框架的服务提供端启动类 加入了zk 遇到了线程安全问题 成功解决了
*/
public class NIOProviderBootStrap12 {
    public static void main(String[] args) throws IOException,
InterruptedException, KeeperException {
        //启动
        new Thread(new Runnable() {
            @Override
            public void run() {
                //因为每个服务提供端内部都是在监听循环阻塞 每个开启一个线程进行监听
                try {
                    NIONonBlockingServer12hello.start(6666);
                } catch (IOException e) {
                    e.printStackTrace();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (KeeperException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}

```

```

        //启动
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    NIONonBlockingServer12bye.start(6667);
                } catch (IOException e) {
                    e.printStackTrace();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (KeeperException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}

```

◦ 实现消费者端获取所需服务对应的地址

```

//通过方法名 反过来获取服务对应的地址
public class ZkServiceDiscovery {
    private static String connectString = RpcConstants.ZOOKEEPER_ADDRESS;
    private static int sessionTimeout = RpcConstants.ZOOKEEPER_SESSION_TIMEOUT;
    private static ZooKeeper zooKeeper;

    //第一步当然是连接到远端服务器上了
    public static void getConnect() throws IOException {
        zooKeeper = new ZooKeeper(connectString, sessionTimeout, new Watcher() {
            @Override
            public void process(WatchedEvent watchedEvent) {

            }
        });
    }

    // 根据所请求的服务地址 获取对应的远端地址
    public static String getMethodAddress(String methodName) throws RpcException,
    InterruptedException, KeeperException {

        //判断节点中是否存在对应路径 不存在则抛出异常
        if (zooKeeper.exists("/service/"+methodName,null)==null)
        {
            System.out.println("不存在该方法");
            throw new RpcException();
        }
    }
}

```

```

        //到对应节点中获取地址    stat节点状态信息变量
        byte[] data = zooKeeper.getData("/service/" + methodName, false,null);
        String address = new String(data);
        return address;
    }

    public static void getStart(String methodName) throws IOException,
        RpcException, InterruptedException, KeeperException {
        //先进行连接
        getConnect();
        //获取相应的远端地址
        String methodAddress = getMethodAddress(methodName);
        //进行连接
        String[] strings = methodAddress.split(":");
        //启动
        String address = strings[0];
        int port = Integer.valueOf(strings[1]);
        NIONonBlockingClient12.start(address,port);
    }
}

```

- **用代理模式 让用户感觉在调用本地方法一样，调用远端方法。**

- 前面的改动尚不能实现，需要加上现在下面这一部分，代理模式，才能更好的实现用户的远端调用
- 设计被代理用户类

```

//这是之后要被代理的对象 我们会实现它的方法
public interface Customer {
    String sayBye(String saying);
    String sayHello(String saying);
}

```

- 设计代理类 //实现了调用后回传的功能 像本身调用一样

```

package consumer.proxy;

import consumer.zkService.ZkServiceDiscovery;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

//代理类的实现
public class RpcClientProxy {

```

```

//获取代理对象 并返回 当前类别
public static Object getBean(final Class<?> serviceClass){
    /*
        参数详解
        1、用哪个类加载器去加载对象
        2、动态代理类需要实现的接口 class[]{xxxx.class} 得到的就是对应的类别
        3、动态代理类执行方法的时候需要干的事
    */
    return
    Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
        new Class[]{serviceClass},
        new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                //暂时还没有设置回信这个操作
                String methodName = method.getName();
                String response = ZkServiceDiscovery.getStart(methodName,
(String) args[0]);
                return response;
            }
        }
    );
}

```

- 修改了消费者端的方法，和之前的功能不一样了，之前是属于在控制台输入 持续的回复，这次改为一次调用得到回复，像调用自己的方法一样

```

package consumer.nio;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.nio.charset.StandardCharsets;
import java.util.Iterator;

//v1.2版本非阻塞nio 真正意义上实现了rpc调用
public class NIONonBlockingClient12 {
    public static String start(String HostName, int PORT,String msg) throws
IOException{
        return start0(HostName,PORT,msg);
    }

    //真正启动在这

```



```

private static String start0(String hostName, int port,String msg) throws
IOException {
    //得到一个网络通道
    SocketChannel socketChannel = SocketChannel.open();
    System.out.println("-----服务消费方启动-----");
    socketChannel.configureBlocking(false);
    //建立链接 非阻塞连接 但我们要等他连接上
    if (!socketChannel.connect(new InetSocketAddress(hostName,port))) {
        while (!socketChannel.finishConnect());
    }
    //创建选择器 进行监听读事件
    Selector selector = Selector.open();
    socketChannel.register(selector, SelectionKey.OP_READ,
ByteBuffer.allocate(1024));

    //进行发送 发的太快了 来不及收到

    socketChannel.write(ByteBuffer.wrap(msg.getBytes(StandardCharsets.UTF_8)));

    //直接进行监听
    while (true)
    {
        //捕获异常 监听读事件
        try {
            if (selector.select(1000)==0)
            {
                continue;
            }
            Iterator<SelectionKey> keyIterator =
selector.selectedKeys().iterator();
            while (keyIterator.hasNext())
            {
                SelectionKey key = keyIterator.next();
                ByteBuffer buffer = (ByteBuffer)key.attachment();
                SocketChannel channel = (SocketChannel)key.channel();
                int read = 1;
                //用这个的原因是怕 多线程出现影响
                StringBuffer stringBuffer = new StringBuffer();
                while (read!=0)
                {
                    buffer.clear();
                    read = channel.read(buffer);
                    stringBuffer.append(new String(buffer.array(),0,read));
                }
                return stringBuffer.toString();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```
}  
}
```

- 启动类代码

```
package consumer.bootstrap;  
  
import consumer.proxy.RpcClientProxy;  
import method.Customer;  
  
import java.io.IOException;  
  
/*  
    以nio为网络编程框架的消费者端启动类  
*/  
public class NIOConsumerBootstrap12 {  
    public static void main(String[] args) throws IOException {  
  
        RpcClientProxy clientProxy = new RpcClientProxy();  
        Customer customer = (Customer) clientProxy.getBean(Customer.class);  
        String response = customer.hello("success");  
        System.out.println(response);  
        System.out.println(customer.bye("fail"));  
    }  
}
```

- 实现效果

```
-----服务消费方启动-----  
Hello,success  
-----服务消费方启动-----  
Bye,fail  
  
Process finished with exit code 0
```

```
服务端:127.0.0.1:6666方法注册完毕  
服务端:127.0.0.1:6667方法注册完毕  
连接到消费端/127.0.0.1:60006  
收到信息success来自/127.0.0.1:60006  
连接到消费端/127.0.0.1:60012  
收到信息fail来自/127.0.0.1:60012  
/127.0.0.1:60012下线了  
/127.0.0.1:60006下线了
```

v1.3

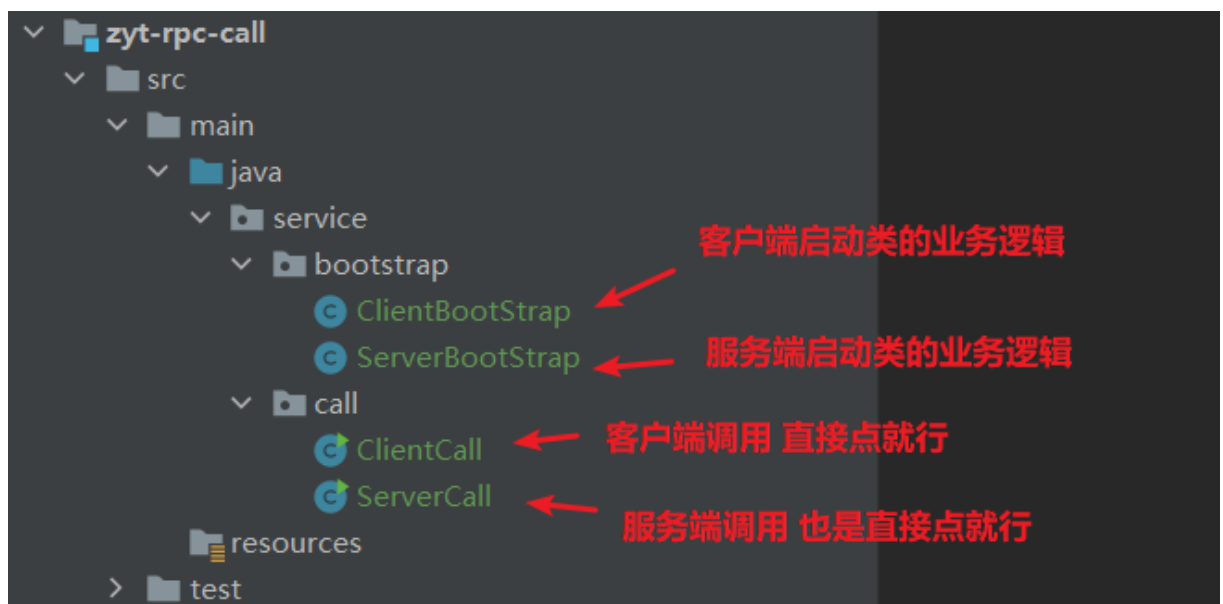
(启动器依旧使用1.2 1.3版本在启动服务版本上尚未做出大变动 主要是增加了方便学习的功能)

更新事项 以下更新均在非阻塞模块进行更新，阻塞模块可供读者自己尝试

- 使用注解方式 改造一下启动类 不分成这么多启动类直接

- 首先新创建了几个包 下面进行阐述下具体作用

- 1.功能调用模块



- 2.注解类 自定义两个注解



- 自定义注解代码

- 客户端启动注解

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//注解 通过此注解可以判断当前是哪一个版本 选择调用哪个版本的客户端启动器
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface RpcClientBootstrap {
    String version();
}
```

- 服务端启动注解

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//注解 通过此注解可以判断当前是哪一个版本 选择调用哪个版本的服务端启动器
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface RpcServerBootstrap {
    String version();
}
```

- 服务调用和实际进行判断的业务逻辑，下面代码就放客户端的，两者同理

- 服务调用

```

package service.call;

import service.bootstrap.ClientBootstrap;

import java.io.IOException;

//通用启动类 将启动的逻辑藏在ClientBootstrap中
public class ClientCall {
    public static void main(String[] args) throws IOException {
        ClientBootstrap.start();
    }
}

```

■ 服务启动真正逻辑

```

package service.bootstrap;

import annotation.RpcClientBootstrap;

import consumer.bootstrap.NIOConsumerBootstrap10;
import consumer.bootstrap.NIOConsumerBootstrap11;
import consumer.bootstrap.NIOConsumerBootstrap12;

import java.io.IOException;

//之后启动直接在这边启动根据 在注解中配置对应的版本号 将相应的操作封装到之后的操作中即可
//这样很方便 就是每次咱加一个启动器还得改下switch
//比如说这里的version 1.2 就是v1.2版本的启动器
@RpcClientBootstrap(version = "1.1")
public class ClientBootstrap {
    public static void start() throws IOException{
        //获取当前的注解上的版本然后去调用相应的远端方法 反射的方法
        //当前客户端启动器class对象
        Class<ClientBootstrap> currentClientBootstrapClass =
            ClientBootstrap.class;
        RpcClientBootstrap annotation =
            currentClientBootstrapClass.getAnnotation(RpcClientBootstrap.class);
        String currentVersion = annotation.version();
        //根据注解获得的版本进行判断是哪个版本 然后进行启动
        switch (currentVersion)
        {
            case "1.0":
                NIOConsumerBootstrap10.main(null);
                break;
            case "1.1":
                NIOConsumerBootstrap11.main(null);
                break;
            case "1.2":
                NIOConsumerBootstrap12.main(null);

```

```

        break;
    default:
        System.out.println("太着急了兄弟，这个版本还没出呢！要不你给我提个PR");
    }
}
}

```

- 利用ZK实现调用方法软负载均衡

- 对zookeeper节点注册进行修改，节点名就是对应的地址，对应的数据就是被调用的次数
 - zookeeper服务注册端修改

```

//因为这个地区属于一个临界区 可能会发生线程不安全问题 所以进行上🔒
synchronized (ZkServiceRegistry.class) {
    Stat exists = zooKeeper.exists("/service", false);
    if (exists == null) {
        zooKeeper.create("/service",
            "".getBytes(StandardCharsets.UTF_8),
            ZooDefs.Ids.OPEN_ACL_UNSAFE,
            CreateMode.PERSISTENT
        );
    }

    //v1.3进行软负载均衡修改
    exists = zooKeeper.exists("/service/"+RpcServiceName, false);
    if (exists == null) {
        zooKeeper.create("/service/"+RpcServiceName,
            "".getBytes(StandardCharsets.UTF_8),
            ZooDefs.Ids.OPEN_ACL_UNSAFE,
            CreateMode.PERSISTENT
        );
    }
}

String date = hostname+": "+port;

//权限目前都设置为全放开 创建方式均为持久化
//修改 v1.3 数据为访问次数 应该是可以进行加减的 然后发现服务端取的是最低的然后再
进行+1
zooKeeper.create("/service/"+RpcServiceName+"/"+date,
    "0".getBytes(StandardCharsets.UTF_8),
    ZooDefs.Ids.OPEN_ACL_UNSAFE,
    CreateMode.PERSISTENT
);
}

```

■ zookeeper服务发现端进行修改

```
//v1.3更新 使用软负载
//到对应节点中获取下面的子节点
List<String> children = zooKeeper.getChildren(prePath, false, null);
if (children.isEmpty())
{
    System.out.println("当前没有服务器提供该服务 请联系工作人员");
}

//进行排序 根据每个节点的访问次数 从小到大进行排序 然后选用最小的
Collections.sort(children, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {

        try {
            return Integer.valueOf(new
String(zooKeeper.getData(prePath+"/"+o1,false,null)))
            -
            Integer.valueOf(new
String(zooKeeper.getData(prePath+"/"+o2,false,null)));
        } catch (KeeperException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return 0;
    }
});
//对选用的对象的访问量加1 todo 暂时不知道怎么让数据直接+1
// 获取节点数据+1, 然后修改对应节点,
String chooseNode = children.get(0);
byte[] data = zooKeeper.getData(prePath+"/"+chooseNode, false, null);
int visitCount = Integer.valueOf(new String(data));
++visitCount;
//version参数用于指定节点的数据版本, 表明本次更新操作是针对指定的数据版本进行的。
cas

zooKeeper.setData(prePath+"/"+chooseNode,String.valueOf(visitCount).getBytes(S
tandardCharsets.UTF_8),-1);
String address = new String(children.get(0));
return address;
}
```

■ 效果: 成功实现软负载均衡

```
服务端:127.0.0.1:6668方法注册完毕  
服务端:127.0.0.1:6666方法注册完毕  
服务端:127.0.0.1:6667方法注册完毕
```

```
[zk: localhost:2181(CONNECTED) 11] get /service/hello/127.0.0.1:6668  
1  
[zk: localhost:2181(CONNECTED) 12] get /service/hello/127.0.0.1:6666  
1
```

- 问题 服务端下线的时候，希望能把zk对应的节点也进行删除，不能让用户每次都自己去zk中删吧
 - 当前想到的办法是提取出一个方法来 在启用之前进行调用，删除节点!!有力扣刷题那味了☺

```
package init;  
  
import constants.RpcConstants;  
import org.apache.zookeeper.KeeperException;  
import org.apache.zookeeper.WatchedEvent;  
import org.apache.zookeeper.Watcher;  
import org.apache.zookeeper.ZooKeeper;  
  
import java.io.IOException;  
import java.util.List;  
  
//zookeeper 进行一键初始化的方法  
public class ZK {  
  
    private static ZooKeeper zooKeeper;  
  
    public static void init() throws IOException, InterruptedException,  
KeeperException {  
        zooKeeper = new ZooKeeper(RpcConstants.ZOOKEEPER_ADDRESS,  
RpcConstants.ZOOKEEPER_SESSION_TIMEOUT, new Watcher() {  
            @Override  
            public void process(WatchedEvent watchedEvent) {  
  
            }  
        });  
  
        //如果存在就删 不存在就不删  
        if (zooKeeper.exists("/service",false)!=null)  
        {  
            //内部得实现递归删除  
            deleteAll("/service");  
        }  
        zooKeeper.close();  
    }  
}
```



```
//实现循环递归删除的方法
private static void deleteAll(String prePath) throws InterruptedException,
KeeperException {
    List<String> children = zookeeper.getChildren(prePath, false);
    if (!children.isEmpty())
    {
        for (String child : children) {
            deleteAll(prePath+"/"+child);
        }
    }
    zookeeper.delete(prePath,-1);
}
}
```

v1.4

小更新

更新事项 暂定目标对启动类进行修改 直接集合

- 这个就直接看代码吧 不是特别难 难的地方我会点出来
 - 启动引导类直接进行修改 可以传参 可以这样 当然 我想到了可以注解传参
注解构造 **注解构造**

```
package annotation;

//两个参数分别代表的是方法和启用的数量

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//注解在类上 然后根据方法获得对应的属性进行判断
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface RpcMethodCluster {
    String[] method();
    int[] startNum();
}
```

- 调用类改造

```

//通用启动类 将启动的逻辑藏在ServerBootStrap中
//注解 看你像启动多少个服务和对应的方法
@RpcMethodCluster(method = {"Hello","Bye"},startNum = {2,3})
public class ServerCall {
    public static void main(String[] args) throws IOException,
        InterruptedException, KeeperException, NoSuchMethodException {
        ServerBootStrap.start();
    }
}

```

- 启动类改造

```

    public static void start() throws IOException, InterruptedException,
        KeeperException, NoSuchMethodException {

        //先对ZK进行初始化
        ZK.init();
        Class<ServerBootStrap> serverBootStrapClass = ServerBootStrap.class;
        RpcServerBootStrap annotation =
            serverBootStrapClass.getAnnotation(RpcServerBootStrap.class);
        //当前服务端启动器 class对象
        String currentServerBootStrapVersion = annotation.version();

        //获取对应的方法和个数 然后进行启动
        //1.获取对应方法 在获取对应的注解 注解中的属性
        RpcMethodCluster nowAnnotation =
            ServerCall.class.getAnnotation(RpcMethodCluster.class);
        String[] methods = nowAnnotation.method();
        int[] startNums = nowAnnotation.startNum();
        //如果不存在那就返回
        if (methods.length==0)return;
        //2.需要组合在一起传过去 如果不组合分别传 我怕就是端口号会出现问题
        StringBuilder methodBuilder = new StringBuilder();
        StringBuilder numBuilder = new StringBuilder();
        for (String method : methods) {
            methodBuilder.append(method);
            methodBuilder.append(",");
        }
        methodBuilder.deleteCharAt(methodBuilder.length()-1);
        for (int startNum : startNums) {
            numBuilder.append(startNum);
            numBuilder.append(",");
        }
        numBuilder.deleteCharAt(numBuilder.length()-1);

        switch (currentServerBootStrapVersion)
        {
            case "1.0":

```

```

        NIOProviderBootStrap10.main(null);
        break;
    case "1.1":
        NIOProviderBootStrap11.main(null);
        break;
    case "1.2":
        NIOProviderBootStrap12.main(null);
        break;
    case "1.4":
        NIOProviderBootStrap14.main(new String[]
{methodBuilder.toString(), numBuilder.toString()});
        break;
    default:
        System.out.println("太着急了兄弟，这个版本还没出呢！要不你给我提个PR");
    }
}
}
}

```

◦ 对应版本启动类改造

```

public class NIOProviderBootStrap14 {
    static volatile int port = 6666;
    public static void main(String[] args) throws IOException,
InterruptedException, KeeperException {
        //对应的方法和对应的方法数量要启动多少 启动的端口不一样 不能写死 首先是
        String methodStr = args[0];
        String numStr = args[1];
        String[] methods = methodStr.split(",");
        String[] nums = numStr.split(",");
        //进行创建 可能会出问题 这边的端口
        for (int i = 0; i < methods.length; i++) {
            String methodName = methods[i];
            for (Integer methodNum = 0; methodNum < Integer.valueOf(nums[i]);
methodNum++) {
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            NIONonBlockingServer14.start(methodName, port++);
                        } catch (IOException e) {
                            e.printStackTrace();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        } catch (KeeperException e) {
                            e.printStackTrace();
                        }
                    }
                }).start();
            }
        }
    }
}

```

```

    }
}
}
}

```

◦ 真正服务功能调用类改造

```

//这里要有新逻辑了 根据获得的方法名 去找到相应的方法
//方法我们保存在固定位置 同时含有固定后缀
String className = method + "ServiceImpl";
Class<?> methodClass = Class.forName("provider.api."+className);
//实例 要获取对应的实例 或者子对象才能进行反射执行方法
Object instance = methodClass.newInstance();

//要传入参数的类型
String response = (String) methodClass.
    getMethod("say" + method,String.class).
    invoke(instance, msg);

```

• 问题

```

//这里要有新逻辑了 根据获得的方法名 去找到相应的方法
//方法我们保存在固定位置 同时含有固定后缀
String className = method + "ServiceImpl";
Class<?> methodClass = Class.forName("provider.api."+className);
//实例 要获取对应的实例 或者子对象才能进行反射执行方法
Object instance = methodClass.newInstance();

//要传入参数的类型
String response = (String) methodClass.
    getMethod(name: "say" + method,String.class).
    invoke(instance, msg);

```

全限定类名

必须要创建实例对象传入invoke

因为我们找的方法是有参构造 所以必须传入相应的类

- 出了bug 客户端连接上直接掉线 正在排查 原因是服务提供端的问题: `Class.forName()`找不到对应的类报了异常`ClassNotFoundException` 解决: 类名和路径名不一样, 用.隔开而不是/,同时注意包从src.java之后开始 这之前放着会有一些问题
- 解决上面的问题又出现bug 说方法找不到, 因为我的方法是有参构造, 我忘记传参数了
- 还有问题 就是invoke要传入对象实例, 生成实例即可

• 完成结果

```

-----服务提供方启动-----
-----服务提供方启动-----
-----服务提供方启动-----
-----服务提供方启动-----
-----服务提供方启动-----
服务端:127.0.0.1:6666:Hello方法注册完毕
服务端:127.0.0.1:6670:Bye方法注册完毕
服务端:127.0.0.1:6669:Bye方法注册完毕
服务端:127.0.0.1:6668:Bye方法注册完毕
服务端:127.0.0.1:6667:Hello方法注册完毕
-----服务消费方启动-----
Hello,success
-----服务消费方启动-----
Bye,fail
-----服务消费方启动-----
Hello,fail

Process finished with exit code 0

```

- 此次更新的目的

1. 利用反射实现功能，让用户可以根据自己的需要，选择启动的服务，不需要知道内部的源码亦或者其他东西，进行修改参数即可启动

v1.5

更新事项 尝试将注册中心换成nacos 将负载均衡策略提取出来 同时新创建随机均衡策略，这是供如果是zk为注册中心使用的，倘若是nacos为注册中心的话用 nacos的工具类中自带负载均衡

- 提取负载均衡策略 新增随机策略

- 新建接口 接口的作用定义方法

```

package loadbalance;

import annotation.LoadBalanceMethodImpl;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.ZooKeeper;

//实现不同的负载均衡策略
@LoadBalanceMethodImpl(chosenMethod = AccessBalance.class)
public interface LoadBalance {
    //通过负载均衡策略返回相应地址
    String loadBalance(ZooKeeper zookeeper, String path) throws
    InterruptedException, KeeperException;
}

```

- 新建注解 定义在接口之上用来判断此时应该使用哪个实现类

```

package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//注解的参数直接是要传入什么类
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface LoadBalanceMethodImpl {
    Class chosenMethod();
}

```

- 根据反射获取对应的负载方法，通过传递信息，实现方法调用获得经过负载均衡后得到的地址。

```

//v1.5修改使用负载均衡策略 根据接口上注解选择的实现类进行调用
LoadBalanceMethodImpl annotation =
LoadBalance.class.getAnnotation(LoadBalanceMethodImpl.class);
Class methodClass = annotation.chosenMethod();
Method method = methodClass.getMethod("loadBalance", new Class[]
{ZooKeeper.class, String.class});
//被选中的负载均衡实现类的对象 通过反射执行 获取对应的地址
Object methodChosenClass = methodClass.newInstance();
String address = (String)
method.invoke(methodChosenClass, zookeeper, prePath);

return address;

```

- 写一个随机访问负载均衡方法

```

package loadbalance;

import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.ZooKeeper;

import java.nio.charset.StandardCharsets;
import java.util.List;
import java.util.Random;

public class RandomBalance implements LoadBalance{
    @Override
    public String loadBalance(ZooKeeper zookeeper, String path) throws
    InterruptedException, KeeperException {
        List<String> children = zookeeper.getChildren(path, null,null);
        if (children.isEmpty())
        {
            System.out.println("当前没有服务器提供该服务 请联系工作人员");
        }
        int size = children.size();
        Random random = new Random();
        //这是应该处于0--size-1之间
        int randomIndex = random.nextInt(size);
        String chooseNode = children.get(randomIndex);
        byte[] data = zookeeper.getData(path + "/" + chooseNode, null, null);
        int visitedCount = Integer.valueOf(new String(data));
        ++visitedCount;
        zookeeper.setData(path+"/"+ chooseNode,
        String.valueOf(visitedCount).getBytes(StandardCharsets.UTF_8),-1);
        return chooseNode;
    }
}

```

- 问题

- Exception in thread "main" java.lang.reflect.UndeclaredThrowableException

努力寻找中，从昨天排查到现在 就是一个bug 我把我zookeeper中的test删除 就成功了

- 尝试使用新的注册中心nacos

安装步骤见技术选型的注册中心那里

服务注册和服务发现模仿给的示例代码

```
public class App {
    public static void main(String[] args) throws NacosException {
        Properties properties = new Properties();
        properties.setProperty("serverAddr", "21.34.53.5:8848,21.34.53.6:8848");
        properties.setProperty("namespace", "quickStart");
        NamingService naming = NamingFactory.createNamingService(properties);
        naming.registerInstance("nacos.test.3", "11.11.11.11", 8888, "TEST1");
        naming.registerInstance("nacos.test.3", "2.2.2.2", 9999, "DEFAULT");
        System.out.println(naming.getAllInstances("nacos.test.3"));
    }
}
```

- 将服务端的服务注册进nacos
- 设置nacos连接地址常数

```
//Nacos服务器连接地址 后面的服务名和地址再拼接
public static String NACOS_ADDRESS =
"http://192.168.18.128:8848/nacos/v1/ns/instance?";
```

- 因为要向远端发送注册服务命令 所以用到了HttpClient或者nacos封装的命令功能 官方给的请求方式

```
curl -X POST 'http://127.0.0.1:8848/nacos/v1/ns/instance?
serviceName=nacos.naming.serviceName&ip=20.18.7.10&port=8080'
```

- nacos服务注册

```
package provider.nacosService;

import com.alibaba.nacos.api.NacosFactory;
import com.alibaba.nacos.api.exception.NacosException;
import com.alibaba.nacos.api.naming.NamingService;
import constants.RpcConstants;

import java.util.Properties;

public class NacosServiceRegistry {
    //直接进行注册
    public static void register(String RpcServiceName,String hostname,int
port) throws NacosException {
        Properties properties = RpcConstants.propertiesInit();
        //创建namingService
        NamingService namingService =
NacosFactory.createNamingService(properties);
        //进行注册
        namingService.registerInstance(RpcServiceName, hostname, port,
"DEFAULT");
    }
}
```



```

        System.out.println("服务端:"+hostname+": "+port+": "+RpcServiceName+"方法在
nacos中注册完毕");
    }
}

```

■ 非阻塞服务器端修改

```

//将服务注册进Nacos中 进行改造
NacosServiceRegistry.register(method,"127.0.0.1",port);

```

• 通过nacos获取对应的服务

- 因为要向远端获取服务 所以用到了HttpClient功能
官方给的请求方式 我们进行改造

```

curl -X GET 'http://127.0.0.1:8848/nacos/v1/ns/instance/list?
serviceName=nacos.naming.serviceName'

```

■ nacos服务发现

```

package consumer.servicediscovery;

import com.alibaba.nacos.api.NacosFactory;
import com.alibaba.nacos.api.exception.NacosException;
import com.alibaba.nacos.api.naming.NamingService;
import com.alibaba.nacos.api.naming.pojo.Instance;
import constants.RpcConstants;
import consumer.nio.NIONonBlockingClient12;
import exception.RpcException;
import org.apache.zookeeper.KeeperException;

import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.util.List;
import java.util.Properties;

public class NacosServiceDiscovery {
    public static String getMethodAddress(String methodName) throws
NacosException, RpcException {
        Properties properties = RpcConstants.propertiesInit();
        NamingService namingService =
NacosFactory.createNamingService(properties);

        //这个方法内部实现了负载均衡
        Instance instance = namingService.selectOneHealthyInstance(methodName);
        if (instance==null)
        {
            System.out.println("没有提供该方法");

```

```

        throw new RpcException("没有对应的方法");
    }
    String ip = instance.getIp();
    int port = instance.getPort();
    String methodAddress = ip+":"+port;
    return methodAddress;
}

public static String getStart(String methodName,String msg) throws
IOException, RpcException,NacosException {
    //获取相应的远端地址
    String methodAddress = getMethodAddress(methodName);
    //进行连接
    String[] strings = methodAddress.split(":");
    //启动
    String address = strings[0];
    int port = Integer.valueOf(strings[1]);
    return NIONonBlockingClient12.start(address,port,msg);
}
}

```

- 非阻塞客户端改造 和之前一样 就不copy代码出来了
- **通过注解的形式，供客户选择的方式选择注册中心使用**
 - 截一个服务提供端的注解方式 客户消费端同理
 - 自定义注解

```

package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//注册中心选择 默认采用zookeeper
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface RegistryChosen {
    String registryName() default "zookeeper";
}

```

- 创建一个接口在上面注解 所用启动类都需要实现该接口

```

package provider.bootstrap.nio;

import annotation.RegistryChosen;

//注册中心的选择 启用的是nacos 目前
@RegistryChosen(registryName = "zookeeper")
public interface NIOProviderBootstrap {
}

```

- 创建了一个工具类进行判断并实现往哪个注册中心注册

```

package provider.utils;

import annotation.RegistryChosen;
import com.alibaba.nacos.api.exception.NacosException;
import exception.RpcException;
import org.apache.zookeeper.KeeperException;
import provider.bootstrap.nio.NIOProviderBootstrap;
import provider.serviceregistry.NacosServiceRegistry;
import provider.serviceregistry.ZkServiceRegistry;

import java.io.IOException;

//直接实现启动类根据启动类接口上的注解选择对应需要选取的方法
public class MethodRegister implements NIOProviderBootstrap {
    /**
     * 实际进行注册的方法
     * @param method 方法名字
     * @param ip 对应的ip
     * @param port 对应的port
     */
    public static void register(String method, String ip, int port) throws
NacosException, RpcException, IOException, InterruptedException,
KeeperException {

        RegistryChosen annotation = MethodRegister.class.getInterfaces()
[0].getAnnotation(RegistryChosen.class);

        switch (annotation.registryName())
        {
            case "nacos":
                NacosServiceRegistry.registerMethod(method, ip, port);
                break;
            case "zookeeper":
                ZkServiceRegistry.registerMethod(method, ip, port);
                break;
            default:
                throw new RpcException("不存在该注册中心");
        }
    }
}

```

```
}
}
```

- 负载均衡策略 这块就用自己的实现负载均衡策略了 nacos内部封装了的包可以实现负载均衡

- 追一下源码

```
//这个方法内部实现了负载均衡
Instance instance = namingService.selectOneHealthyInstance(methodName);
```

```
public static Instance selectHost(ServiceInfo dom) {
    List<Instance> hosts = selectAll(dom);
    if (CollectionUtils.isEmpty(hosts)) {
        throw new IllegalStateException("no host to srv for service: " + dom.getName());
    } else {
        return Balancer.getHostByRandomWeight(hosts);
    }
}
```

最终追入结果，可得对应的实例

- 此次更新最终

1. 通过注解实现负载均衡 用户自由选择，并实现新的负载均衡方法，随机法 🐶
2. 实现nacos的注册中心，并使用了nacos的负载均衡
3. 通过注解反射实现了用户自由选择注册中心

v1.6

热补丁，nio目前来看最后的完善，使用Curator简化zookeeper的操作，优化调用体验

- 使用Curator创建服务注册和服务发现类（是看快速开始速成的）

- 服务注册类实现代码

```
package provider.service_registry;

import constants.RpcConstants;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.CuratorFrameworkFactory;
import org.apache.curator.retry.ExponentialBackoffRetry;

import java.nio.charset.StandardCharsets;
```

//通过curator简化 zookeeper对相应的服务端服务注册的流程 更轻松的看懂

```
public class ZkCuratorRegistry {
    private static String connectString = RpcConstants.ZOOKEEPER_ADDRESS;
    public static void registerMethod(String RpcServiceName, String hostname, int
port) throws Exception {
        //创建连接 然后将对应的对象注册进去即可
        //BackoffRetry 退避策略，决定失败后如何确定补偿值。
        //ExponentialBackOffPolicy
        //指数退避策略，需设置参数sleeper、initialInterval、
        // maxInterval和multiplier，initialInterval指定初始休眠时间，默认100毫秒，
        // maxInterval指定最大休眠时间，默认30秒，multiplier指定乘数，即下一次休眠时间为当前
        休眠时间*multiplier;

        CuratorFramework client = CuratorFrameworkFactory.newClient(connectString,
            new ExponentialBackoffRetry(1000, 3));
        //需要启动 当注册完毕后记得关闭 不然会浪费系统资源
        client.start();

        //进行创建 首先判断是否创建过service还有对应的方法路径 是这样判断 但是我可能还没有
        完全玩透curator
        //多线程问题 一定要进行加锁
        synchronized (ZkCuratorRegistry.class)
        {
            if (client.checkExists().forPath("/service")==null)
            {

client.create().forPath("/service","",getBytes(StandardCharsets.UTF_8));
                }
                if (client.checkExists().forPath("/service/"+RpcServiceName)==null)
                {

client.create().forPath("/service/"+RpcServiceName","",getBytes(StandardCharsets.UTF_8));
                }
            }
            String date = hostname+": "+port;

            client.create().forPath("/service/"+RpcServiceName+"/"+date,"0".getBytes(StandardCharsets.UTF_8));
            client.close();
            System.out.println("服务端:"+hostname+": "+port+": "+RpcServiceName+"方法在zkCurator中注册完毕");
        }
    }
}
```

o 服务发现类实现代码

服务发现端没有强行改，因为要保证我们负载均衡算法的完整性，不强行的进行改动了，读者如需改动可以自己尝试，curator并不是很难 主要就加了下面这段，后面的逻辑还是和之前的一样。

```
CuratorFramework client = CuratorFrameworkFactory.newClient(connectString,
    new ExponentialBackoffRetry(1000, 3));
//需要启动 当获取完毕后需要关闭相应的客户端
client.start();
//首先获取的时候 要负载均衡
Zookeeper zookeeper = client.getZookeeperClient().getZookeeper();
```

- 优化调用体验

- 将部分改动代码上传

- 用户启动类

```
//通用启动类 将启动的逻辑藏在ClientBootstrap中
public class ClientCall {
    public static void main(String[] args) throws IOException, RpcException {
        Customer customer = ClientBootstrap.start();
        //实现调用
        System.out.println(customer.Hello("success"));
        System.out.println(customer.Bye("fail"));
        System.out.println(customer.Hello("fail"));
    }
}
```

- 实际引导启动类

```
//1.2版本之前都是键盘输入 所以不是根据代理对象来进行调用的 暂时注释掉
// case "1.0":
//     NIOConsumerBootstrap10.main(null);
//     break;
// case "1.1":
//     NIOConsumerBootstrap11.main(null);
//     break;
case "1.2":
    return NIOConsumerBootstrap12.main(null);
case "1.4":
    return NIOConsumerBootstrap14.main(null);
case "1.5":
    return NIOConsumerBootstrap15.main(null);
default:
    throw new RpcException("太着急了兄弟，这个版本还没出呢！要不你给我提个PR");
```

- nio实际启动端

```
public class NIOConsumerBootstrap12 {
    public static Customer main(String[] args) throws IOException {

        RpcClientProxy clientProxy = new RpcClientProxy();
        return (Customer) clientProxy.getBean(Customer.class);
    }
}
```

- **总结**

改动不大，用curator实现zookeeper注册中心提供更清晰的代码，效率上是没有什么差别的，降低了使用的复杂性,修改了下几个启动类 直接让用户自己进行方法的选取调用

RPC框架v2.0netty版

v2.0 netty版的话就是实现一个最简单的netty版本 能实现这边的客户端发信息给服务端 服务端能收到并回应

技术选型

后续进行了大量更新

- 网络传输：netty
- 序列化：java自带序列化/protobuf/kyro
- 注册中心：zookeeper(尝试去实现一个单机版zk?)/nacos
- 注解开发（尝试）补丁选项

客户端

用户客户端启动类

```
//客户端启动类
public class NettyClientCall {
    public static void main(String[] args) throws InterruptedException {
        NettyClientBootstrap.start("127.0.0.1",6668);
    }
}
```

客户端引导启动类

```

package service.netty_bootstrap;

import consumer.bootstrap.netty.NettyConsumerBootstrap20;

public class NettyClientBootstrap {
    public static void start(String address, int port) throws InterruptedException {
        NettyConsumerBootstrap20.main(new String[]{address, String.valueOf(port)});
    }
}

```

消费者启动类

```

package consumer.bootstrap.netty;

import consumer.netty.NettyClient20;

/*
    以netty为网络编程框架的消费者端启动类
*/
//进行启动 提供类的方式即可
public class NettyConsumerBootstrap20 {
    public static void main(String[] args) throws InterruptedException {
        NettyClient20.start(args[0], Integer.parseInt(args[1]));
    }
}

```

消费者实际启动类

```

package consumer.netty;

import consumer.netty_client_handler.NettyClientHandler01;
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;

//实际客户端启动类
public class NettyClient20 {
    public static void start(String hostName, int port) throws InterruptedException {
        Bootstrap bootstrap = new Bootstrap();
        EventLoopGroup workGroup = new NioEventLoopGroup();
    }
}

```



```

        try {
            bootstrap.group(workGroup)
                .channel(NioSocketChannel.class)
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel socketChannel) throws
Exception {
                        ChannelPipeline pipeline = socketChannel.pipeline();
                        pipeline.addLast(new NettyClientHandler01());
                    }
                });

            ChannelFuture channelFuture = bootstrap.connect(hostName, port).sync();

            channelFuture.addListener(new ChannelFutureListener() {
                @Override
                public void operationComplete(ChannelFuture channelFuture) throws
Exception {
                    if (channelFuture.isSuccess()) {
                        System.out.println("连接"+hostName+": "+port+"成功");
                    }
                    else
                    {
                        System.out.println("连接"+hostName+": "+port+"失败");
                    }
                }
            });

            //监听关闭事件，本来是异步的，现在转换为同步事件
            channelFuture.channel().closeFuture().sync();
        } finally
        {
            //优雅的关闭 group
            workGroup.shutdownGracefully();
        }
    }
}

```

消费者处理器

```

package consumer.netty_client_handler;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.util.CharsetUtil;

```

```

public class NettyClientHandler01 extends ChannelInboundHandlerAdapter {

    //通道就绪就会发的信息
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        ctx.writeAndFlush(Unpooled.copiedBuffer("你好，服务端", CharsetUtil.UTF_8));
    }

    //这个是收到信息
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        ByteBuf buf = (ByteBuf)msg;
        System.out.println("收到来自"+ctx.channel().remoteAddress()+"的消息"+buf.toString(CharsetUtil.UTF_8));
    }
}

```

服务端

用户服务端启动类

```

package service.netty_call;

import service.netty_bootstrap.NettyServerBootstrap;

//启动类 给定对应的端口 进行启动并监听
public class NettyServerCall {
    public static void main(String[] args) throws InterruptedException {
        NettyServerBootstrap.start("127.0.0.1",6668);
    }
}

```

服务端引导启动类

```

package service.netty_bootstrap;

import provider.bootstrap.netty.NettyProviderBootstrap20;

public class NettyServerBootstrap {
    public static void start(String address,int port) throws InterruptedException {
        NettyProviderBootstrap20.main(new String[]{address, String.valueOf(port)});
    }
}

```



```

hashCode="+socketChannel.hashCode());
        ChannelPipeline pipeline = socketChannel.pipeline();//每个通道都
对应一个管道 将处理器往管道里放
        pipeline.addLast(new NettyServerHandler01());
    }
    });

    System.out.println("服务器 is ready");

    //连接 同步
    ChannelFuture cf = serverBootstrap.bind(hostName, port).sync();

    cf.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws Exception {
            if (future.isSuccess()) {
                System.out.println("监听端口"+port+"成功");
            }
            else
            {
                System.out.println("监听端口"+port+"失败");
            }
        }
    });

    //对关闭通道进行监听
    cf.channel().closeFuture().sync();
} finally {
    //优雅的关闭两个集群
    bossGroup.shutdownGracefully();
    workGroup.shutdownGracefully();
}
}
}

```

服务提供者处理器

```

package provider.netty_server_handler;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.util.CharsetUtil;

//实现简单的服务注册和回写
public class NettyServerHandler01 extends ChannelInboundHandlerAdapter {

```

```

//读取数据
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    //将信息进行读取 直接这样就可以了
    ByteBuf buf = (ByteBuf) msg;
    System.out.println("客户端发送消息是: "+ buf.toString(CharsetUtil.UTF_8));
    System.out.println("客户端地址: "+ctx.channel().remoteAddress());
}

//数据读取完毕
@Override
public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
    //读取完毕进行回显 写回并刷新
    ctx.writeAndFlush(Unpooled.copiedBuffer("success", CharsetUtil.UTF_8));
}

//捕获异常
@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws
Exception {
    //异常处理 首先先将通道的上下文关闭 每个ctx对应的就是handler本身
    ctx.close();
    cause.printStackTrace();
}
}

```

序列化

还是选用了自带的序列化，之后会对目前的序列化进行优化

依赖引入

```

<!--引入netty的依赖-->
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.66.Final</version>
</dependency>

```

更新

v2.1

简单的小改一下 方便用户后面操作

- 将两个操作类集中一下顺便把注解完善，使得注解都在总的操作类上
 - 启动方法选择注解的编写，选择是用netty还是nio实现rpc

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//进行rpc工具的选择
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface RpcToolsSelector {
    String rpcTool() default "NIO";
}
```

- 总服务启动类

```
package service;

import annotation.RpcMethodCluster;
import annotation.RpcServerBootStrap;
import annotation.RpcToolsSelector;
import org.apache.zookeeper.KeeperException;
import service.call.ChosenServerCall;

import java.io.IOException;

//总服务端启动类 用户调用 注解是 注册什么方法进去
//调用的是什么版本的服务端启动方法
@RpcMethodCluster(method = {"Hello","Bye"},startNum = {2,3})
@RpcServerBootStrap(version = "1.5")
@RpcToolsSelector(rpcTool = "Nio")
public class ServerCall {
    public static void main(String[] args) throws IOException,
        InterruptedException, KeeperException, NoSuchMethodException {
        ChosenServerCall.start();
    }
}
```

- 总客户端启动类

```
package service;
```

```

import annotation.RpcClientBootstrap;
import annotation.RpcToolsSelector;
import exception.RpcException;
import method.Customer;
import service.call.ChosenClientCall;

import java.io.IOException;

//总客户端启动类 用户调用 什么版本的 和用什么工具 使用什么注册中心 序列化的选择 都可以用这个来玩
//注册中心不能给过去 这样就是重复依赖了
@RpcClientBootstrap(version = "1.5")
@RpcToolsSelector(rpcTool = "Nio")
public class ClientCall {
    public static void main(String[] args) throws RpcException, IOException,
    InterruptedException {
        //实现调用
        Customer customer = ChosenClientCall.start();

        System.out.println(customer.Hello("success"));
        System.out.println(customer.Bye("fail"));
        System.out.println(customer.Hello("fail"));
    }
}

```

- 实现netty2.1版本，完成用户像调用自己方法一样进行调用外部方法，用了异步调用，代理模式
 - 以netty为网络编程框架的消费者端启动类

```

package consumer.bootstrap.netty;

import consumer.proxy.RpcNettyClientProxy;
import method.Customer;

/*
    以netty为网络编程框架的消费者端启动类
*/
//进行启动 提供类的方式即可
public class NettyConsumerBootstrap21 {
    public static Customer main(String[] args) throws InterruptedException {
        return (Customer) RpcNettyClientProxy.getBean(Customer.class);
    }
}

```

- 代理类获取代理对象

```

package consumer.proxy;

import annotation.RegistryChosen;
import consumer.netty.NettyClient21;
import consumer.service_discovery.NacosServiceDiscovery;
import consumer.service_discovery.ZkCuratorDiscovery;
import consumer.service_discovery.ZkServiceDiscovery;
import exception.RpcException;
import register.Register;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

//就是获取代理类 通过代理类中的方法进行对应方法的执行和获取
public class RpcNettyClientProxy {
    private static ExecutorService executor =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
    //参数 就是我要对其生成代理类的类
    public static Object getBean(final Class<?> serviceClass)
    {
        return
Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
                        new Class[]{serviceClass},
                        new InvocationHandler() {
                            //根据对应的方法 代理类进行处理
                            @Override
                            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                                String methodName = method.getName();
                                Object param = args[0];
                                //获取对应的方法地址
                                String methodAddress = getMethodAddress(methodName);
                                String[] strings = methodAddress.split(":");
                                String hostName = strings[0];
                                int port = Integer.valueOf(strings[1]);
                                //进行方法的调用 随即进行方法的调用
                                return NettyClient21.callMethod(hostName,port,param);
                            }
                        });
    }

    /**
     * 实际去获得对应的服务 并完成方法调用的方法
     * @param methodName 根据方法名 根据添加的注册中心注解来选择相应的注册中心进行 实现负
载均衡获取一个方法对应地址
     * @param
     * @return

```



```

    */
    private static String getMethodAddress(String methodName) throws Exception {
        //根据注解进行方法调用
        //根据在代理类上的注解调用 看清楚底下的因为是个class数组 可以直接继续获取 注解
        RegistryChosen annotation =
Register.class.getAnnotation(RegistryChosen.class);
        switch (annotation.registryName())
        {
            case "nacos":
                return NacosServiceDiscovery.getMethodAddress(methodName);
            case "zookeeper":
                return ZkServiceDiscovery.getMethodAddress(methodName);
            case "zkCurator":
                return ZkCuratorDiscovery.getMethodAddress(methodName);
            default:
                throw new RpcException("不存在该注册中心");
        }
    }
}

```

- 代理类中实际实现远程调用方法 创建对应的客户端监听，进行方法的调用和写回

```

package consumer.netty;

import consumer.netty_client_handler.NettyClientHandler21;
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

//实际客户端启动类 进行操作
//不确定能返回什么 所以判断是对象
public class NettyClient21 {

    //线程池 实现异步调用
    private static ExecutorService executor =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

```

```

static NettyClientHandler21 clientHandler;

public static void initClient(String hostName,int port)
{

    clientHandler = new NettyClientHandler21();
    //建立客户端监听
    Bootstrap bootstrap = new Bootstrap();
    EventLoopGroup workGroup = new NioEventLoopGroup();

    try {
        bootstrap.group(workGroup)
            .channel(NioSocketChannel.class)
            .handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel socketChannel)
throws Exception {
                    ChannelPipeline pipeline = socketChannel.pipeline();
                    pipeline.addLast(new StringEncoder());//传输必须加编解码
器 不然不认识的类传不过去

                    pipeline.addLast(new StringDecoder());
                    pipeline.addLast(clientHandler);

                }
            });

        //进行连接
        bootstrap.connect(hostName, port).sync();

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static Object callMethod(String hostName, int port, Object param)
throws Exception {

    //我是有多个地方进行调用的 不能只连接一个
    initClient(hostName,port);
    clientHandler.setParam(param);
    //接下来这就有关系到调用 直接调用
    return executor.submit(clientHandler).get();
}
}

```

- 实现异步调用的处理器

```

package consumer.netty_client_handler;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

import java.util.concurrent.Callable;

//实现了Callable接口实现了异步调用

public class NettyClientHandler21 extends ChannelInboundHandlerAdapter implements
Callable{
    //传入的参数
    private Object param;
    private Object response;
    private ChannelHandlerContext context;

    public void setParam(Object param) {
        this.param = param;
    }

    //当成功建立 就赋值上下文对象
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        context = ctx;
        System.out.println("U•エ•*U 成功连接");
    }

    @Override
    public synchronized void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {
        response = msg;
        notify();
    }

    //调用的时候 就进行传输
    @Override
    public synchronized Object call() throws Exception {
        context.writeAndFlush(param);
        wait();
        return response;
    }

    //异常处理
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws
Exception {
        cause.printStackTrace();
    }
}

```

```

        ctx.close();
    }
}

```

- 以netty为网络编程框架的客户端启动类

```

package provider.bootstrap.netty;

import provider.netty.NettyServer21;

/*
    以netty为网络编程框架的服务提供端启动类
*/
//实现方法和之前都是比较一致的 每个对象要开一个线程去执行呢 原因是我们会启动同步等他们关闭 才出来
//这样才能关闭对应的管道
public class NettyProviderBootStrap21 {
    static volatile int port = 6666; //对应的端口 要传过去 注册到注册中心去
    public static void main(String[] args) throws InterruptedException {
        //直接在这里将对应的方法什么的进行分开 然后传过去
        String methods = args[0];
        String nums = args[1];
        String[] methodArray = methods.split(",");
        String[] methodNumArray = nums.split(",");
        //进行创建 可能会出问题 这边的端口
        for (int i = 0; i < methodArray.length; i++) {
            String methodName = methodArray[i];
            for (Integer methodNum = 0; methodNum <
Integer.valueOf(methodNumArray[i]); methodNum++) {
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            NettyServer21.start(methodName, port++);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                }).start();
            }
        }
    }
}

```

o 方法的绑定和注册

```
package provider.netty;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;
import provider.netty_server_handler.NettyServerHandler21;
import provider.utils.MethodRegister;

//进行启动 绑定然后创建对应的 然后注册进注册中心去
public class NettyServer21 {
    public static void start(String methodName,int port) throws Exception {
        //真正的实现逻辑 被封装到下面的方法当中了
        start0(methodName,port);
    }

    private static void start0(String methodName, int port) throws Exception {
        //先将地址进行注册
        MethodRegister.register(methodName,"127.0.0.1",port);

        //开始创建相应的netty服务端
        ServerBootstrap serverBootstrap = new ServerBootstrap();

        //创建对应的工作组 用于处理不同的事件
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workGroup = new NioEventLoopGroup();

        try {
            //进行初始化
            serverBootstrap.group(bossGroup,workGroup)
                .channel(NioServerSocketChannel.class) //自身实现的通道
                .option(ChannelOption.SO_BACKLOG,128) //设置线程队列得到的连接个数
                .childOption(ChannelOption.SO_KEEPALIVE,true) //设置保持活动连接
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel socketChannel)
throws Exception {
                        System.out.println(socketChannel.remoteAddress()+"连接
状态
上了");

                        ChannelPipeline pipeline = socketChannel.pipeline();
                        pipeline.addLast(new StringEncoder());

```

```

        pipeline.addLast(new StringDecoder());
        pipeline.addLast(new
NettyServerHandler21(methodName));
    }
});

ChannelFuture channelFuture = serverBootstrap.bind(port).sync();

//对连接结果进行监听
channelFuture.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture) throws
Exception {
        if (channelFuture.isSuccess()) System.out.println("连接
上"+port+"端口");
        else System.out.println("连接端口失败");
    }
});

//等待关闭 同步
channelFuture.channel().closeFuture().sync();
} finally {
    //结束了的话 就进行关闭了
    bossGroup.shutdownGracefully();
    workGroup.shutdownGracefully();
}
}
}

```

◦ 服务端的处理器

```

package provider.netty_server_handler;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

import java.lang.reflect.Method;

//实现简单的服务注册和回写
public class NettyServerHandler21 extends ChannelInboundHandlerAdapter {
    private String methodName;

    //要传入对应的方法名 否则不知道 netty服务器能执行什么方法
    public NettyServerHandler21(String methodName)
    {
        this.methodName = methodName;
    }
}

```

```

    }

    //实现对应的方法

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        System.out.println("收到来自"+ctx.channel().remoteAddress()+"的信息");
        //使用反射的方法获取对应的类 通过反射再进行执行
        Class<?> calledClass = Class.forName("provider.api."+methodName +
"ServiceImpl");
        Method method = calledClass.getMethod("say" + methodName, String.class);
        Object instance = calledClass.newInstance();
        Object response = method.invoke(instance, msg.toString());

        //获得对应信息并进行回传
        ctx.writeAndFlush(response);
    }

    //出现异常的话 如何处理
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        ctx.channel().close();
        cause.printStackTrace();
    }
}

```

- **bug:信息无法发送到服务端**

原因: 传输的信息是String类型的所以必须要配备相应的编解码器 不然无法传输

v2.2

更新: 采用两种辅助的序列化框架kyro和protobuf、自带的进行传输，以减少JDK序列化和反序列化所带来的的缺点，我就是写三个对应的编解码器即可 然后还有一个相应的注解

- **设计传送对象**

```

package serialization.entity;

import java.io.Serializable;

```

```
//这是普通进行序列化传递需要
//千万注意要实现序列化接口

//接口一定要实现正确
public class Person implements Serializable {
    private String name;

    //构造方法
    public Person(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

- **netty自带的传输方式，对象编解码器，集成了一个方法根据，传入的方法的参数和返回值进行编解码器的设置**

```
package codec;

import annotation.CodecSelector;
import exception.RpcException;
import io.netty.channel.ChannelPipeline;
import io.netty.handler.codec.serialization.ClassResolvers;
import io.netty.handler.codec.serialization.ObjectDecoder;
import io.netty.handler.codec.serialization.ObjectEncoder;

import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;
import serialization.Serialization;

import java.lang.reflect.Method;

//公共类 根据对应选择的注解进行编码器的添加
public class AddCodec {

    /**
     *
     * @param pipeline
     * @param method 方法，来获取对应的输入输出类
     * @throws RpcException
     */
    public static void addCodec(ChannelPipeline pipeline, Method method, boolean
isConsumer) throws RpcException {
```



```

//根据注解进行编解码器的选择
CodecSelector annotation =
Serialization.class.getAnnotation(CodecSelector.class);

//目前而来我的传输 传入的参数都是一个 所以根据这一个传入和返回的参数的类型进行判断
//下面是我传入的参数 和传出的参数
Class<?> returnType = method.getReturnType();
Class<?> parameterType = method.getParameterTypes()[0];

String codec = annotation.Codec();
switch (codec)
{
    case "ObjectCodec":
        if (returnType!=String.class&&parameterType!=String.class)
        {
            pipeline.addLast(new ObjectEncoder());
            //传的参是固定写法
            pipeline.addLast(new ObjectDecoder(Integer.MAX_VALUE,
                ClassResolvers.weakCachingResolver(null)));
        }
        else if (returnType!=String.class&&parameterType==String.class)
        {
            //如果是客户端的话那么传出的是服务端传入的 所以这边编码那边就是解码
            if (isConsumer)
            {
                //根据传入传出进行对应的编码
                pipeline.addLast(new StringEncoder());
                pipeline.addLast(new ObjectDecoder(Integer.MAX_VALUE,
                    ClassResolvers.weakCachingResolver(null)));
            }
            else
            {
                pipeline.addLast(new ObjectEncoder());
                pipeline.addLast(new StringDecoder());
            }
        }
        else if (returnType==String.class&&parameterType!=String.class)
        {
            //客户端 会对参数进行编码, 服务端是解码
            if (isConsumer)
            {
                pipeline.addLast(new ObjectEncoder());
                pipeline.addLast(new StringDecoder());
            }
            else
            {
                pipeline.addLast(new StringEncoder());
                pipeline.addLast(new ObjectDecoder(Integer.MAX_VALUE,
                    ClassResolvers.weakCachingResolver(null)));
            }
        }
    }
}

```

```

        }
    }
    else
    {
        //因为传入参数和传出都是字符串类型 所以就传入字符串编解码器
        pipeline.addLast(new StringEncoder());
        pipeline.addLast(new StringDecoder());
    }
    return;
case "protobuf": //添加protobuf的编解码器
    return;
case "kryo": //添加kryo的编解码器
    return;
default: //如果都不是那就不加了
    return;
}
}
}
}

```

• 选取序列化对象对应的注解

```

package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//根据该注解判断选择当遇到传对象的相应的方法时，采用什么编解码方式
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface CodecSelector {
    String codec() default "ObjectCodec";
}

```

• BUG

- 问题：在实现传输JDK自带序列化传输时，编解码器仿佛出现了问题 只是连接上了 并不能读取
- 编解码器的问题，因为不论解码器handler还是编码器handler即接收的消息类型必须于待处理的消息类型一致，否则该handler不会被执行
- 还有个错误就是实现序列化的时候，用Serializable 自己设置了一个Serialization 实现这个去了.. 不该错的

v2.3

更新：使用protobuf和kryo替换netty原生的类序列化

- **protobuf**

protobuf(Google Protocol Buffers)是Google提供一个具有高效的协议数据交换格式工具库(类似json)，但相比于json，Protobuf有更高的转化效率，时间效率和空间效率都是JSON的3-5倍。

写这个要会写protobuf脚本 经过protobuf编译后的类，可以直接进行属性的获取和设置，没问题的 就不用写get set方法了

- 创建相应的protoc文件，将所需转换的相关信息写入

```
syntax = "proto3"; //版本
option java_outer_classname = "PersonPOJO"; //生成的外部类名同时也是文件名
//protobuf 使用message管理真正的数据
message Person{
    string name = 1; //对应的参数类型 名称 属性序号
}
//千万记得指令中间的空格别少写或者多写
//编写完成后使用指令  protoc.exe --java_out=. Person.proto  要到自己对应下载好的
protoc.exe文件下
```

- 去到相应的protoc文件夹下，将我们写好的文件复制到bin目录下，通过 protoc.exe --java_out=. Person.proto 完成文件的编译 将编译后的文件保存至我们工程相应的位置 (编译后的文件，对应的代码就不粘了，太长了)

```
//经过编译后的代码，如果要创建实例的话，采用下面的方法
PersonPOJO.Person person = PersonPOJO.Person.newBuilder().setName("炸油条").build();
```

- 进行对编解码器的设置 解码必须要传入实例

```
case "protoc": //添加protobuf的编解码器 如果是protobuf的编解码器的话 那可能还需要一点其他操作
    if (returnType!=String.class&&parameterType!=String.class)
    {
        pipeline.addLast(new ProtobufEncoder());
        //对什么实例解码
        pipeline.addLast(new
ProtobufDecoder(PersonPOJO.Person.getDefaultInstance()));
    }
    else if (returnType!=String.class&&parameterType==String.class)
    {
        //如果是客户端的话那么传出的是服务端传入的 所以这边编码那边就是解码
```

```

        if (isConsumer)
        {
            //根据传入传出进行对应的编码
            pipeline.addLast(new StringEncoder());
            pipeline.addLast(new
ProtobufDecoder(PersonPOJO.Person.getDefaultInstance()));
        }
        else
        {
            pipeline.addLast(new ProtobufEncoder());
            pipeline.addLast(new StringDecoder());
        }
    }
    else if (returnType==String.class&&parameterType!=String.class)
    {
        //客户端 会对参数进行编码，服务端是解码
        if (isConsumer)
        {
            pipeline.addLast(new ProtobufEncoder());
            pipeline.addLast(new StringDecoder());
        }
        else
        {
            pipeline.addLast(new StringEncoder());
            //这个就是获取对应的实例 必须要这样传
            pipeline.addLast(new
ProtobufDecoder(PersonPOJO.Person.getDefaultInstance()));
        }
    }
    else
    {
        //因为传入参数和传出都是字符串类型 所以就传入字符串编解码器
        pipeline.addLast(new StringEncoder());
        pipeline.addLast(new StringDecoder());
    }
    return;
}

```

◦ 引入依赖

```

<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.6.1</version>
</dependency>

```

- **protobuf的优化: protostuff是一个基于protobuf实现的序列化方法** 速度和效率相较于protobuf大大提升 搞懂

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,注意使用后就不能再传递之前用过的proto文件了

- protostuff实现序列化和反序列化的类

```
package serialization.protostuff;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.protostuff.LinkedBuffer;
import io.protostuff.ProtostuffIOUtil;
import io.protostuff.Schema;
import io.protostuff.runtime.RuntimeSchema;
import serialization.Serializer;

//进行序列化和反序列化
public class ProtostuffUtils implements Serializer {

    /**
     * Avoid re applying buffer space every time serialization
     */
    private static final LinkedBuffer BUFFER =
        LinkedBuffer.allocate(LinkedBuffer.DEFAULT_BUFFER_SIZE);

    @Override
    public byte[] serialize(Object obj) {
        Class<?> clazz = obj.getClass();
        Schema schema = RuntimeSchema.getSchema(clazz);
        byte[] bytes;
        try {
            bytes = ProtostuffIOUtil.toByteArray(obj, schema, BUFFER);
        } finally {
            BUFFER.clear();
        }
        return bytes;
    }

    @Override
    public <T> T deserialize(byte[] bytes, Class<T> clazz) {
        Schema<T> schema = RuntimeSchema.getSchema(clazz);
        T obj = schema.newMessage();
        ProtostuffIOUtil.mergeFrom(bytes, obj, schema);
        return obj;
    }
}
```

- 在处理器处理前后根据是否用的protostuff序列化 还有传入的是否是非String类参数 进行 相应的处理

```

if (Serialization.class.getAnnotation(CodecSelector.class).Codec()
    .equals("protostuff")
    &&msg.getClass()==byte[].class)
{
    ProtostuffUtils protostuffUtils = new ProtostuffUtils();
    //反序列化的模板 是根据我传进来的参数进行改变的
    msg = protostuffUtils.deserialize((byte[]) msg,param.getClass());
}
response = msg;
notify();

```

- 引入依赖

```

<dependency>
    <groupId>io.protostuff</groupId>
    <artifactId>protostuff-runtime</artifactId>
    <version>1.3.8</version>
</dependency>
<dependency>
    <groupId>io.protostuff</groupId>
    <artifactId>protostuff-core</artifactId>
    <version>1.3.8</version>
</dependency>

```

- kryo

注意：不支持跨语言，只支持Java Object的序列化

- Kryo工具类

```

package serialization.kryo;

import com.esotericsoftware.kryo.Kryo;
import com.esotericsoftware.kryo.io.Input;
import com.esotericsoftware.kryo.io.Output;
import entity.Person;
import serialization.Serializer;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

//通过KryoUtils实现序列化
public class KryoUtils implements Serializer {

    /**
     * Because Kryo is not thread safe. So, use ThreadLocal to store Kryo objects

```

```

    */
    private final ThreadLocal<Kryo> kryoThreadLocal = ThreadLocal.withInitial(() -
> {
        Kryo kryo = new Kryo();
        // kryo.register(Person.class); // 注册了能提高性能 减少了空间的 浪费 /但如果分布
        式系统中注册 不同的顺序可能导致错误
        // kryo.register(PersonPOJO.Person.class); //不能使用这个
        kryo.setRegistrationRequired(false); //显式的关闭了注册的行为
        return kryo;
    });

    @Override
    public byte[] serialize(Object obj) {
        //将其序列化 然后写入到输出流中
        //byte数组输出流
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
        //需要的参数有Output是它自带的 要创建一个
        Output output = new Output(byteArrayOutputStream);
        Kryo kryo = kryoThreadLocal.get();
        kryo.writeObject(output, obj);
        kryoThreadLocal.remove();
        return output.toBytes();
    }

    @Override
    public <T> T deserialize(byte[] bytes, Class<T> clazz) {
        //根据传入的bytes就知道里面是什么了 然后读取 转换
        Kryo kryo = kryoThreadLocal.get();
        //需要的参数有Input是它自带的
        ByteArrayInputStream byteArrayInputStream = new
        ByteArrayInputStream(bytes);
        Input input = new Input(byteArrayInputStream);
        Object obj = kryo.readObject(input, clazz);
        kryoThreadLocal.remove();
        return (T) obj;
    }
}

```

在处理器的实现逻辑和protostuff一致，就不粘贴了，去看下原理，然后集成下，更加轻便的使用，否则看起来太繁琐了

- 引入对应依赖

```

<!--引入kryo依赖-->
<dependency>
    <groupId>com.esotericsoftware</groupId>
    <artifactId>kryo</artifactId>
    <version>5.3.0</version>
</dependency>

```

- 简化处理器判断流程

- ```
private boolean isProtostuff;
private boolean iskryo;
```

//当成功建立 就赋值上下文对象

@Override

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {
 context = ctx;
 System.out.println("U•ε•*U 成功连接");
 String codecType = Serialization.class.getAnnotation(CodecSelector.class).Codec();
 if (codecType.equals("protostuff"))isProtostuff = true;
 else if (codecType.equals("kryo"))iskryo = true;
}
```

....

....

- BUG

- 做好相应protoc序列化后，服务端成功注册上后，客户端启动后出现问题

- 做好相应protoc序列化后，服务端成功注册上后，客户端启动后出现问题

```
Exception in thread "main" java.lang.reflect.UndeclaredThrowableException
 at com.sun.proxy.$Proxy3.GetPerson(Unknown Source)
 at service.ClientCall.main(ClientCall.java:27)
Caused by: java.nio.channels.ClosedChannelException <9 internal lines>
 at io.netty.bootstrap.Bootstrap$3.run(Bootstrap.java:244)
 at io.netty.util.concurrent.AbstractEventExecutor.safeExecute(AbstractEventExecutor.java:164)
 at io.netty.util.concurrent.SingleThreadEventExecutor.runAllTasks(SingleThreadEventExecutor.java:472)
 at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:500)
 at io.netty.util.concurrent.SingleThreadEventExecutor$4.run(SingleThreadEventExecutor.java:989)
```

- 解决方法 应该就是要在编解码器处传入对应要传输的数据

- 因为重载了多个方法，所以在通过反射实现方法的时候，选取哪个方法出现了问题。

//因为方法比较多 所以在方法的选择上还是存在的问题 对应的有对应的方法

```
Method[] methods = calledClass.getMethods();
```

```
Method method = methods[1];
```

```
Object instance = calledClass.newInstance();
```

```
Object response = method.invoke(instance, msg);
```

//获得对应信息并进行回传

```
ctx.writeAndFlush(response);
```

- 解决

```
Class<?> calledClass = Class.forName("provider.api."+methodName + "ServiceImpl");
```

```
Method[] methods = calledClass.getMethods();
```

```
Method method = methods[0];
```

//因为我进行重写了 内部会有多个实现方法 所以就按照对应的传入参数 来判断是哪个方法

//因为methods[i].getParameterTypes()返回的就是class集合 我不需要再去获取对应



```

class的类了，我刚刚就是搞错了 又获取了一遍类 导致出错
 for (int i = 0; i < methods.length; i++) {
 if (methods[i].getParameterTypes()[0]==msg.getClass())
 {
 method = methods[i];
 break;
 }
 }
 Object instance = calledClass.newInstance();
 Object response = method.invoke(instance, msg);
 //获得对应信息并进行回传
 ctx.writeAndFlush(response);

```

- 通过protostuff转换成的byte[] 无法传输，因为writeandflush需要进行编解码对byte[]
  - 解决进行大改：当传入的不是String.class实例的时候 才进行序列化，不管是序列化还是反序列化，都是根据他是不是byte[]实例才进行，在添加编解码器的时候，就需要注意了
  - 同时遇到一个问题，当我用到protostuff之后传入的类就不需要进行proto编译了，加入编译的类反而还报错
- kryo消费者接收回信时，出现问题 com.esotericsoftware.kryo.io.KryoBufferUnderflowException: Buffer underflow.
  - 解决：莫名其妙就好了...
- 注意 kryo不支持包不含无参构造器的反序列化
- 待解决问题，不能只针对一个类别可以序列化，要实现多个类别序列化

## v2.4

**更新：**增加序列化的实现hessian,json, Thrift, FST、Avro等序列化工具，从这个版本开始JDK自带的序列化和protoc编译后的进行序列化的暂时先不使用了

- 进行序列化工具的提取

```

package serialization;

import annotation.CodecSelector;
import exception.RpcException;
import serialization.fst.FSTUtils;
import serialization.hessian.HessianUtils;
import serialization.kryo.KryoUtils;
import serialization.protostuff.ProtostuffUtils;

//这是一个进行统一序列化的一个工具
public class SerializationTool implements Serializer {

 static String codec =
 Serialization.class.getAnnotation(CodecSelector.class).Codec();

 @Override

```

```

public byte[] serialize(Object obj) throws RpcException {
 switch (codec) {
 case "kryo":
 return new KryoUtils().serialize(obj);
 case "protostuff":
 return new ProtostuffUtils().serialize(obj);
 case "hessian":
 return new HessianUtils().serialize(obj);
 case "fst":
 return new FSTUtils().serialize(obj);
 case "thrift":
 return null;
 case "avro":
 return null;
 case "jackson":
 return null;
 case "fastjson":
 return null;
 case "gson":
 return null;
 default:
 throw new RpcException("你所找的序列化方法还没编写，或者可能在该版本被废弃了，
你可以看看2.2版本");
 }
}

```

```

@Override
public <T> T deserialize(byte[] bytes, Class<T> clazz) throws RpcException {
 switch (codec) {
 case "kryo":
 return new KryoUtils().deserialize(bytes, clazz);
 case "protostuff":
 return new ProtostuffUtils().deserialize(bytes, clazz);
 case "hessian":
 return new HessianUtils().deserialize(bytes, clazz);
 case "fst":
 return new FSTUtils().deserialize(bytes, clazz);
 case "thrift":
 return null;
 case "avro":
 return null;
 case "jackson":
 return null;
 case "fastjson":
 return null;
 case "gson":
 return null;
 default:
 throw new RpcException("你所找的序列化方法还没编写，或者可能在该版本被废弃了，
你可以看看2.2版本");
 }
}

```

```

 }
}
}

```

- handler中的改造（顺便改造了下clientHandler 因为之前改写了以后，string也要序列化，然后返回类型的判断按照之前出了问题，所以传入了方法，获取返回的类型）

```

package consumer.netty_client_handler;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import serialization.SerializationTool;

import java.lang.reflect.Method;
import java.util.concurrent.Callable;

/*
 * 在2.4版本之后我们就暂时淘汰JDK序列化和protoc编译成的类进行了
 * */
//实现了Callable接口实现了异步调用

public class NettyClientHandler24 extends ChannelInboundHandlerAdapter implements
Callable{
 //传入的参数
 private Object param;
 private Method method;
 private Object response;
 private ChannelHandlerContext context;

 //序列化工具
 static SerializationTool serializationTool = new SerializationTool();

 public void setParam(Object param) {
 this.param = param;
 }
 public void setMethod(Method method) {
 this.method = method;
 }

 //当成功建立 就赋值上下文对象
 @Override
 public void channelActive(ChannelHandlerContext ctx) throws Exception {
 context = ctx;
 System.out.println("U•エ•*U 成功连接");
 }

 @Override

```

```

 public synchronized void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {

 //在这要进行解码 获得传回来的信息 如果是遇到下面的msg 那就代表传回来的肯定是个byte[]
 // 根据我们要的方法进行解码 传回来的应该是方法的response的
 //根据需要的返回类型进行反序列化
 msg = serializationTool.deserialize((byte[]) msg,method.getReturnType());

 response = msg;
 notify();
 }

 //调用的时候 就进行传输
 @Override
 public synchronized Object call() throws Exception {

 //这个变量的目的就是保留原来的param实际参数类型，当返回的时候 可以当作反序列化的模板
 Object request = param;
 //判断是否需要protostuff进行序列化 因为使用这个进行序列话 是我没有相应的解码器 2.4之后 就
算是string也进行序列化

 request = serializationTool.serialize(request);

 context.writeAndFlush(request);
 wait();
 return response;
 }

 //异常处理
 @Override
 public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws
Exception {
 cause.printStackTrace();
 ctx.close();
 }
}

```

- 实现hessian

- 最简单一步 引入依赖

```

<!--引入hessian依赖-->
<dependency>
 <groupId>com.caucho</groupId>
 <artifactId>hessian</artifactId>
 <version>4.0.65</version>
</dependency>

```

- 实现序列化反序列化的功能

```
package serialization.hessian;

import com.caucho.hessian.io.HessianInput;
import com.caucho.hessian.io.HessianOutput;
import exception.RpcException;
import serialization.Serializer;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

public class HessianUtils implements Serializer {

 //进行序列化
 @Override
 public byte[] serialize(Object obj) throws RpcException {
 try {
 ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
 HessianOutput hessianOutput = new HessianOutput(outputStream);
 hessianOutput.writeObject(obj);
 return outputStream.toByteArray();
 } catch (IOException e) {
 e.printStackTrace();
 throw new RpcException("序列化失败");
 }
 }

 //进行反序列化
 @Override
 public <T> T deserialize(byte[] bytes, Class<T> clazz) throws RpcException {
 ByteArrayInputStream inputStream = new ByteArrayInputStream(bytes);
 HessianInput hessianInput = new HessianInput(inputStream);
 try {
 Object object = hessianInput.readObject(clazz);
 return clazz.cast(object);
 } catch (IOException e) {
 e.printStackTrace();
 throw new RpcException("反序列化失败");
 }
 }
}
```

- 实现json序列化

- 首先json序列化有一系列的框架实现（fastJson、gson、Jackson）

- Jackson

- 依赖引用

```
<!--jackson依赖引用-->
<dependency>
 <groupId>com.fasterxml.jackson.core</groupId>
 <artifactId>jackson-databind</artifactId>
 <version>2.9.5</version>
</dependency>
```

#### ■ 实现序列化反序列化功能

```
package serialization.json;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import exception.RpcException;
import serialization.Serializer;

import java.io.IOException;

public class JacksonUtils implements Serializer {

 //Jackson中的关键
 static ObjectMapper mapper = new ObjectMapper();

 @Override
 public byte[] serialize(Object obj) throws RpcException,
 JsonProcessingException {
 //将obj转换成json再将json转换成字节数组
 return mapper.writeValueAsBytes(obj);
 }

 //将json字符串或者字节组转换为对应的类
 @Override
 public <T> T deserialize(byte[] bytes, Class<T> clazz) throws
 RpcException, IOException {
 return mapper.readValue(bytes,clazz);
 }
}
```

#### ■ fastjson

##### ■ 依赖引用

```

<!--fastjson依赖引用-->
<dependency>
 <groupId>com.alibaba</groupId>
 <artifactId>fastjson</artifactId>
 <version>1.2.80</version>
</dependency>

```

#### ■ 实现序列化反序列化功能

```

package serialization.json;

import com.alibaba.fastjson.JSON;
import exception.RpcException;
import serialization.Serializer;
import java.nio.charset.StandardCharsets;

//阿里巴巴旗下的进行json转换的工具
public class FastJsonUtils implements Serializer {
 @Override
 public byte[] serialize(Object obj) throws RpcException {

 // return
 JSON.toJSONString(obj,true).getBytes(StandardCharsets.UTF_8);
 return JSON.toJSONBytes(obj);
 }

 @Override
 public <T> T deserialize(byte[] bytes, Class<T> clazz) throws
RpcException {
 //后面的序列化参数也是可选可不选
 return JSON.parseObject(bytes,clazz);
 }
}

```

- 问题：序列化后对象中的属性变为null  
解决：需要对对象中的属性设置get set 方法

#### ■ gson

##### ■ 依赖引用

```

<!--gson依赖引用-->
<dependency>
 <groupId>com.google.code.gson</groupId>
 <artifactId>gson</artifactId>
 <version>2.8.8</version>
</dependency>

```

##### ■ 实现序列化反序列化功能

```

package serialization.json;

import com.google.gson.Gson;
import exception.RpcException;
import serialization.Serializer;

import java.nio.charset.StandardCharsets;

//gson
public class GsonUtils implements Serializer {
 static Gson gson = new Gson();
 @Override
 public byte[] serialize(Object obj) throws RpcException {
 return gson.toJson(obj).getBytes(StandardCharsets.UTF_8);
 }

 //重点就是上下的编解码器 一定要保证字符串的正确性

 @Override
 public <T> T deserialize(byte[] bytes, Class<T> clazz) throws
RpcException {
 return gson.fromJson(new
String(bytes,StandardCharsets.UTF_8),clazz);
 }
}

```

- 问题: com.google.gson.JsonSyntaxException: java.lang.IllegalStateException: Expected BEGIN\_OBJECT but was BEGIN\_ARRAY at line 1 column 2 path \$  
解决: 保证传输的字符串编解码正确

## • 实现FST序列化工具

- 引入依赖 (发现fst和nacos的冲突2.57就不行 会报错)

```

<!--引入Fst依赖-->
<dependency>
 <groupId>de.ruedigermoeller</groupId>
 <artifactId>fst</artifactId>
 <version>2.04</version>
</dependency>

```

- 工具类的实现

```

package serialization.fst;

import exception.RpcException;
import serialization.Serializer;
import org.nustaq.serialization.FSTConfiguration;

//一款较新的序列化工具

```



```

public class FSTUtils implements Serializer {

 //需要弄懂 rather than
 static FSTConfiguration fstConfiguration =
FSTConfiguration.createStructConfiguration();

 //进行序列化
 @Override
 public byte[] serialize(Object obj) throws RpcException {
 return fstConfiguration.asByteArray(obj);
 }

 //进行反序列化
 @Override
 public <T> T deserialize(byte[] bytes, Class<T> clazz) throws RpcException {
 return clazz.cast(fstConfiguration.asObject(bytes));
 }
}

```

## v2.5

### 尝试心跳、数据压缩、TCP沾包拆包

- 心跳机制的实现 [Netty源码解析-IdleStateHandler](#)

实现通过用户设置 多少秒没读 多少秒没写 多少秒没读写进行报读空闲、写空闲、读写空闲  
当客户端一定时间内没有收到相应的请求，就将客户端的连接关闭，服务端是需要持续

- 注解创建 注解创建结束后配置都在common包中
  - 注解编写

```

package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//是否开启的工具类
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface HeartBeatTool {
 boolean isOpenFunction() default false; //是否开启的标志
 int readerIdleTimeSeconds() default 5; //超过多少时间触发读空闲事件
 int writerIdleTimeSeconds() default 5; //超过多少时间触发写空闲事件
}

```

```
int allIdleTimeSeconds() default 3; //超过多少时间触发读写空闲事件
}
```

#### ■ 注解到配置类上

```
package heartbeat;

import annotation.HeartBeatTool;

@HeartBeatTool(isOpenFunction = true,
 readerIdleTimeSeconds = 4,
 writerIdleTimeSeconds = 4,
 allIdleTimeSeconds = 2)
public interface HeartBeat {
}
```

#### ○ 心跳事件对应处理器添加构造

```
//上面的改造
private static HeartBeatTool heartBeatToolAnnotation =
HeartBeat.class.getAnnotation(HeartBeatTool.class);

/*
v2.5更新添加读写空闲处理器
IdleStateHandler是netty提供的处理读写空闲的处理器
readerIdleTimeSeconds 多长时间没有读 就会传递一个心跳包进行检测
writerIdleTimeSeconds 多长时间没有写 就会传递一个心跳包进行检测
allIdleTimeSeconds 多长时间没有读写 就会传递一个心跳包进行检测

当事件触发后会传递给下一个处理器进行处理，只需要在下一个处理器中实现userEventTriggered事件即可
*/
//时间和实不实现 根据注解 判断是否开启
//记住后面一定是要有一个处理器 用来处理触发事件
if (heartBeatToolAnnotation.isOpenFunction())
{
 pipeline.addLast(new IdleStateHandler(

heartBeatToolAnnotation.readerIdleTimeSeconds(),

heartBeatToolAnnotation.writerIdleTimeSeconds(),

heartBeatToolAnnotation.allIdleTimeSeconds())
);
}
```

#### ○ 事件监听处理

```

//用来判断是否出现了空闲事件 如果出现了那就进行相应的处理
@Override
public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws
Exception {
 if (evt instanceof IdleStateEvent)
 {
 IdleStateEvent event = (IdleStateEvent) evt;
 //设置一个事件字段
 String eventType = null;
 switch (event.state())
 {
 case READER_IDLE:
 eventType = "读空闲";
 break;
 case WRITER_IDLE:
 eventType = "写空闲";
 break;
 case ALL_IDLE:
 eventType = "读写空闲";
 break;
 }
 System.out.println(ctx.channel().remoteAddress()+"发生超时事
件"+eventType+": 连接关闭");
 ctx.close();
 }
}

```

- 可以实现保持连接

- 通过在客户端再设置一个处理器 超过几秒 没发送，就发送一次，保持心跳，因为我这边额外的逻辑，如果要实现得重新写一下，对应的处理器 所以下次实现

- 数据压缩

对于小体积反而会变大，如果是传输大的对象可以选择开启压缩

- 书写注解 进行判断是否开启

```

package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//进行判断解压缩功能是否开启
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface CompressFunction {
 boolean isOpenFunction();
}

```

```
//冠上注解
package compress;

import annotation.CompressFunction;

@CompressFunction(isOpenFunction = true)
public interface Compress {
}
```

◦ 各个解压缩工具类的实现(要知道它们的区别，同时了解它们的实现方式)

■ bzip

■ 依赖引入

```
<!--Bzip依赖引入-->
<dependency>
 <groupId>org.apache.ant</groupId>
 <artifactId>ant</artifactId>
 <version>1.10.6</version>
</dependency>
```

■ bzip解压缩工具类

```
package compress.bzip;

import compress.CompressType;
import org.apache.tools.bzip2.CBZip2InputStream;
import org.apache.tools.bzip2.CBZip2OutputStream;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

//apache下的一个解压缩工具 bzip
public class BZipUtils implements CompressType {

 private static final int BUFFER_SIZE = 8192;
 @Override
 public byte[] compress(byte[] bytes) throws IOException {
 //数组输出流
 ByteArrayOutputStream bos = new ByteArrayOutputStream();
 CBZip2OutputStream cbZip2OutputStream = new
 CBZip2OutputStream(bos);
 cbZip2OutputStream.write(bytes);
 //finish 当压缩结束后 结束
 cbZip2OutputStream.finish();
 byte[] request = bos.toByteArray();
 }
}
```

```

 //关闭
 cbZip2OutputStream.close();
 bos.close();

 return request;
 }

 @Override
 public byte[] decompress(byte[] bytes) throws IOException {
 //输入进行解压 再将解压后的传入输出
 ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
 CBZip2InputStream cbZip2InputStream = new
CBZip2InputStream(bis);

 //将输入流中的数据写入
 ByteArrayOutputStream out = new ByteArrayOutputStream();
 byte[] buffer = new byte[BUFFER_SIZE];
 int n;
 while ((n=cbZip2InputStream.read(buffer))>-1)
 {
 out.write(buffer,0,n);
 }
 byte[] response = out.toByteArray();

 //关闭对应
 out.close();
 cbZip2InputStream.close();
 bis.close();

 return response;
 }
}

```

## ■ 结果

```
/127.0.0.1:55549连接上了
收到来自/127.0.0.1:55549的信息
before compress11
after compress53
/127.0.0.1:55566连接上了
收到来自/127.0.0.1:55566的信息
before compress13
after compress50
/127.0.0.1:55583连接上了
收到来自/127.0.0.1:55583的信息
before compress5
after compress38
```

- deflater

- deflater解压缩工具类（因为这是java.util.zip下自带的工具类不用引入依赖）

```
package compress.deflater;

import compress.CompressType;

import java.io.ByteArrayOutputStream;
import java.util.zip.DataFormatException;
import java.util.zip.Deflater;
import java.util.zip.Inflater;

public class DeflaterUtils implements CompressType {

 //固定读取字节数组大小
 private static final int BUFFER_SIZE = 8192;

 @Override
 public byte[] compress(byte[] bytes) {
 int length;
 Deflater deflater = new Deflater();
 deflater.setInput(bytes);
 deflater.finish();

 byte[] outputBytes = new byte[BUFFER_SIZE];
 ByteArrayOutputStream bos = new ByteArrayOutputStream();
 while (!deflater.finished())
 {
```

```

 length = deflater.deflate(outputBytes);
 bos.write(outputBytes,0,length);
 }
 deflater.end();
 return bos.toByteArray();
}

//解压
@Override
public byte[] decompress(byte[] bytes) throws DataFormatException {
 int length;
 Inflater inflater = new Inflater();
 inflater.setInput(bytes);
 byte[] outputBytes = new byte[BUFFER_SIZE];
 ByteArrayOutputStream bos = new ByteArrayOutputStream();
 while (!inflater.finished())
 {
 length = inflater.inflate(outputBytes);
 bos.write(outputBytes,0,length);
 }
 inflater.end();
 return bos.toByteArray();
}
}

```

#### ■ 结果

```

/127.0.0.1:55982连接上了
收到来自/127.0.0.1:55982的信息
before compress11
after compress20
/127.0.0.1:55999连接上了
收到来自/127.0.0.1:55999的信息
before compress13
after compress21
/127.0.0.1:56016连接上了
收到来自/127.0.0.1:56016的信息
before compress5
after compress13

```

#### ■ gzip

- gzip实现解压缩工具类 (这也是java.util.zip下面的类 不引入依赖 实现过程和)

```

package compress.zip;

import compress.CompressType;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.zip.GZIPInputStream;
import java.util.zip.GZIPOutputStream;

//gzip实现解压缩 实现过程和bzip很相似
public class GzipUtils implements CompressType {
 private static final int BUFFER_SIZE = 8192;
 @Override
 public byte[] compress(byte[] bytes) throws IOException {
 //数组输出流
 ByteArrayOutputStream bos = new ByteArrayOutputStream();
 GZIPOutputStream gzipOutputStream = new GZIPOutputStream(bos);
 gzipOutputStream.write(bytes);
 gzipOutputStream.flush();
 //finish 当压缩结束后 结束
 gzipOutputStream.finish();
 byte[] request = bos.toByteArray();

 //关闭
 gzipOutputStream.close();
 bos.close();

 return request;
 }

 @Override
 public byte[] deCompress(byte[] bytes) throws IOException {

 //输入进行解压 再将解压后的传入输出
 ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
 GZIPInputStream gzipInputStream = new GZIPInputStream(bis);

 //将输入流中的数据写入
 ByteArrayOutputStream out = new ByteArrayOutputStream();
 byte[] buffer = new byte[BUFFER_SIZE];
 int n;
 while ((n=gzipInputStream.read(buffer))>-1)
 {
 out.write(buffer,0,n);
 }
 byte[] response = out.toByteArray();

 //关闭对应
 out.close();
 }
}

```



```

 gzipInputStream.close();
 bis.close();

 return response;
 }
}

```

#### ■ 结果

```

收到来自/127.0.0.1:56315的信息
before compress11
after compress32
/127.0.0.1:56332连接上了
收到来自/127.0.0.1:56332的信息
before compress13
after compress33
/127.0.0.1:56349连接上了
收到来自/127.0.0.1:56349的信息
before compress5
after compress25

```

#### ■ lz4

##### ■ 依赖引入

```

<!--Lz4依赖引入 进行解压缩-->
<dependency>
 <groupId>org.lz4</groupId>
 <artifactId>lz4-java</artifactId>
 <version>1.7.1</version>
</dependency>

```

##### ■ lz4实现解压缩工具类

```

package compress.lz4;

import compress.CompressTpye;
import net.jpountz.lz4.*;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

```

```

import java.io.IOException;

public class Lz4Utils implements CompressType {

 private static final int BUFFER_SIZE = 8192;

 @Override
 public byte[] compress(byte[] bytes) throws IOException {
 //压缩的工具类吧
 LZ4Compressor compressor =
LZ4Factory.fastestInstance().fastCompressor();
 ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

 LZ4BlockOutputStream lz4BlockOutputStream = new
LZ4BlockOutputStream(outputStream, BUFFER_SIZE, compressor);
 lz4BlockOutputStream.write(bytes);

 lz4BlockOutputStream.close();
 return outputStream.toByteArray();
 }

 @Override
 public byte[] deCompress(byte[] bytes) throws IOException {
 ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

 LZ4FastDecompressor decompressor =
LZ4Factory.fastestInstance().fastDecompressor();
 ByteArrayInputStream inputStream = new
ByteArrayInputStream(bytes);

 LZ4BlockInputStream decompressedInputStream = new
LZ4BlockInputStream(inputStream, decompressor);
 int count;
 byte[] buffer = new byte[BUFFER_SIZE];
 while ((count=decompressedInputStream.read(buffer))!=-1)
 {
 outputStream.write(buffer, 0, count);
 }
 decompressedInputStream.close();
 return outputStream.toByteArray();
 }
}

```

#### ■ 结果

```
收到来自/127.0.0.1:57495的信息
before compress11
after compress53
/127.0.0.1:57515连接上了
收到来自/127.0.0.1:57515的信息
before compress13
after compress55
/127.0.0.1:57532连接上了
收到来自/127.0.0.1:57532的信息
before compress5
after compress47
|
```

- zip
  - zip解压缩工具类实现

```
package compress.zip;

import compress.CompressTpye;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;
import java.util.zip.ZipOutputStream;

public class Ziputils implements CompressTpye {
 private static final int BUFFER_SIZE = 8192;
 @Override
 public byte[] compress(byte[] bytes) throws IOException {
 ByteArrayOutputStream out = new ByteArrayOutputStream();
 ZipOutputStream zip = new ZipOutputStream(out);
 ZipEntry entry = new ZipEntry("zip");
 entry.setSize(bytes.length);
 zip.putNextEntry(entry);
 zip.write(bytes);
 zip.closeEntry();
 return out.toByteArray();
 }

 @Override
 public byte[] deCompress(byte[] bytes) throws IOException {
```

```

 ByteArrayOutputStream out = new ByteArrayOutputStream();
 ZipInputStream zip = new ZipInputStream(new
ByteArrayInputStream(bytes));
 byte[] buffer = new byte[BUFFER_SIZE];
 while (zip.getNextEntry() != null)
 {
 int n;
 while ((n = zip.read(buffer)) != -1)
 {
 out.write(buffer, 0, n);
 }
 }
 return out.toByteArray();
 }
}

```

#### ■ 结果

```

收到来自/127.0.0.1:58274的信息
before compress11
after compress63
/127.0.0.1:58291连接上了
收到来自/127.0.0.1:58291的信息
before compress13
after compress64
/127.0.0.1:58308连接上了
收到来自/127.0.0.1:58308的信息
before compress5
after compress56

```

#### • TCP粘包拆包

因为我直接是一次只传一个字节数组，通过对应对象转成的，所以暂时还遇不到粘包拆包问题，在nio的编写中我解决了粘包拆包的问题v1.1 通过阻塞io解决的

##### 解决方案

1. 使用自定义协议 + 编解码器 来解决
2. 关键就是要解决 服务器端每次读取数据长度的问题，这个问题解决，就不会出现服务器多读或少读数据的问题，从而避免TCP粘包、拆包

#### • 测试压缩不压缩所用所占的大小是否变化 同时要比对不同的解压缩方法之间的差异 由来 (未看各个方法的原理)

[Java不同压缩算法的性能比较](#)

实现SPI机制，新增负载均衡算法，大升级改造解决线程安全隐患，构建一个自己根据霍夫曼编码实现的diyZip工具类（未实现）

- SPI机制

该机制实际上就是和我们通过注解选择对应接口实现类的功能一致，我就用他来实现简单的压缩接口，其余还是使用注解开发

- SPI工具类

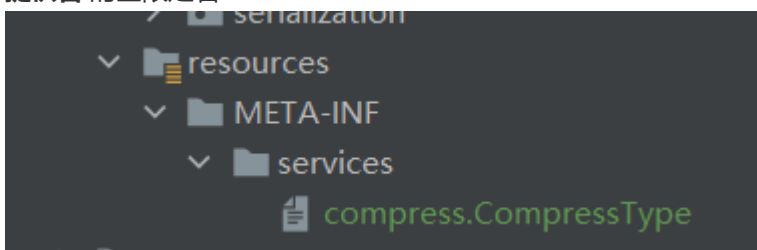
```
package compress;

import exception.RpcException;

import java.util.ServiceLoader;

//通过SPI机制获取对应需要的解压缩工具类 而不是用注解类获取
public class SPICompressUtils {
 public static CompressType getUtils() throws RpcException {
 ServiceLoader<CompressType> loader =
 ServiceLoader.load(CompressType.class);
 for (CompressType compressType : loader) {
 return compressType;
 }
 throw new RpcException("没有去配置对应的解压缩方法");
 }
}
```

- 使用时，须要在 **META-INF/services** 下建立和服务的 **全限定名** 相同的文件，而后在该文件中写入 **服务提供者** 的全限定名



```
#根据需要使用哪个解压缩的方法 打开对应的解压缩工具类注解即可
#compress.bzip.BZipUtils
compress.deflater.DeflaterUtils
#compress.gzip.GZipUtils
#compress.lz4.Lz4Utils
#compress.zip.ZipUtils|
```

- 实现机制

为某个接口寻找服务实现的机制。有点类似IOC的思想，在外面实现。

- 新增负载均衡算法 一致性哈希

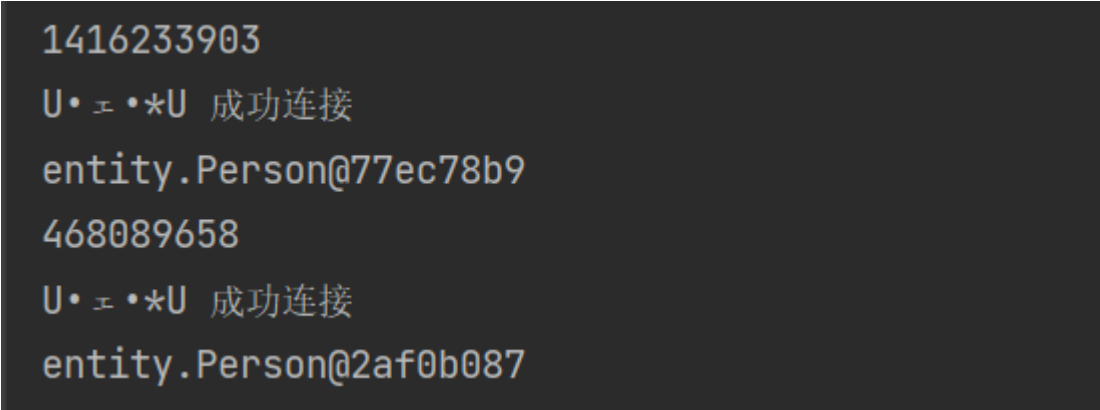
- 实现一个最简单的一致性哈希 经过判断每次zookeeper都是一样的，我们进行修改不同的线程独享一个zookeeper

- 修改代码

```
private static String connectString = RpcConstants.ZOOKEEPER_ADDRESS;
private static int sessionTimeout = RpcConstants.ZOOKEEPER_SESSION_TIMEOUT;
private static ZooKeeper zooKeeper;
private static ThreadLocal<ZooKeeper> zooKeeperThreadLocal =
ThreadLocal.withInitial(()->{
 try {
 return new ZooKeeper(connectString, sessionTimeout, new Watcher() {
 @Override
 public void process(WatchedEvent watchedEvent) {

 }
 });
 } catch (IOException e) {
 e.printStackTrace();
 return null;
 }
});
```

- 实现结果



1416233903  
U•ɿ•\*U 成功连接  
entity.Person@77ec78b9  
468089658  
U•ɿ•\*U 成功连接  
entity.Person@2af0b087

- 实现哈希一致性负载均衡算法，根据zookeeper的哈希值进行分配，而服务器地址就根据对应的服务器的哈希值进行分配

- 代码

```
package loadbalance;

import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.ZooKeeper;

import java.util.HashMap;
import java.util.List;
```

```

//一致性哈希 进行负载均衡计算
public class ConsistentLoadBalance implements LoadBalance{
 @Override
 public String loadBalance(ZooKeeper zooKeeper, String path) throws
 InterruptedException, KeeperException {
 List<String> children = zooKeeper.getChildren(path, false, null);
 if (children.isEmpty())
 {
 System.out.println("当前没有服务器提供该服务 请联系工作人员");
 }
 //我这个属于的是简单版的一致性哈希
 int zkHashCode = zooKeeper.hashCode();
 return children.get(zkHashCode%children.size());

 //细致版一致性哈希 太麻烦了
 /*
 HashMap<Integer,String> hashAddress = new HashMap();
 for (String child : children) {
 hashAddress.put(child.hashCode(),child);
 }
 //先对对应的获得的子节点进行地址的储存
 //我这个属于的是简单版的一致性哈希
 int zkHashCode = zooKeeper.hashCode();
 long address = (long)(zkHashCode % Math.pow(2, 32));
 while (!hashAddress.containsKey(address))
 {
 if(address == (long)Math.pow(2,32)-1)
 {
 address = 0;
 }
 else ++address;
 }
 return hashAddress.get(address);
 */
 }
}

```

- 根据上面的问题引出了线程安全问题 实现大改造 保证线程安全

- 出现的bug

```

Exception in thread "Thread-0" java.lang.reflect.UndeclaredThrowableException <1 internal line>
 at service.ClientCall.lambda$main$0(ClientCall.java:30) <1 internal line>
 Caused by: java.nio.channels.ClosedChannelException <9 internal lines>
 at io.netty.bootstrap.Bootstrap$3.run(Bootstrap.java:244) <6 internal lines>
 ... 1 more

```

- 进行线程安全改造 不能只适用于单线程
  - 一点点追到代码中去 看到可能会影响线程安全的就进行改正测验

发现问题，在添加消费者处理器的时候，由于处理升级为了全局变量，导致每次这个处理器可能设好对应的参数，又被另一个线程初始化了，导致空指针异常。

解决方法：使得对应的处理器，可以每次调用都新创建或者是说设置线程独有`ThreadLocal`，当然这要注意每次调用完要记得删除，不然会出现问题。

## 修改代码

```
package consumer.netty;

import annotation.HeartBeatTool;
import codec.AddCodec;
import consumer.netty_client_handler.NettyClientHandler24;
import heartbeat.HeartBeat;
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.timeout.IdleStateHandler;

import java.lang.reflect.Method;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

//2.4版本主要进行大量序列化工具的集合 然后进行方便式的序列化和反序列化

//实际客户端启动类 进行操作
//不确定能返回什么 所以判断是对象
public class NettyClient24 {

 //线程池 实现异步调用
 private static ExecutorService executor =
 Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
 private static HeartBeatTool heartBeatToolAnnotation =
 HeartBeat.class.getAnnotation(HeartBeatTool.class);
 // static NettyClientHandler24 clientHandler;//跟他没关系 因为每次都新建一个
 private static final ThreadLocal<NettyClientHandler24>
 nettyClientHandlerThreadLocal = ThreadLocal.withInitial(()->new
 NettyClientHandler24());

 public static Object callMethod(String hostName, int port, Object param,
 Method method) throws Exception {

 NettyClientHandler24 clientHandler = nettyClientHandlerThreadLocal.get();
 //建立客户端监听
```



```

Bootstrap bootstrap = new Bootstrap();
EventLoopGroup workGroup = new NioEventLoopGroup();

try {
 bootstrap.group(workGroup)
 .channel(NioSocketChannel.class)
 .handler(new ChannelInitializer<SocketChannel>() {
 @Override
 protected void initChannel(SocketChannel socketChannel)
throws Exception {
 ChannelPipeline pipeline = socketChannel.pipeline();

 //加编解码器的逻辑，根据对应的注解进行编码器的添加 这里面有实现
对应的逻辑 //

 AddCodec.addCodec(pipeline,method,true);

 /*
 v2.5更新添加读写空闲处理器
 IdleStateHandler是netty提供的处理读写空闲的处理器
 readerIdleTimeSeconds 多长时间没有读 就会传递一个心跳
包进行检测

 writerIdleTimeSeconds 多长时间没有写 就会传递一个心跳
包进行检测

 allIdleTimeSeconds 多长时间没有读写 就会传递一个心
跳包进行检测

 当事件触发后会传递给下一个处理器进行处理，只需要在下一个处理
器中实现userEventTriggered事件即可
 */
 //时间和实不实现 根据注解 判断是否开启
 //记住后面一定是要有一个处理器 用来处理触发事件
 if (heartBeatToolAnnotation.isOpenFunction())
 {
 pipeline.addLast(new IdleStateHandler(

heartBeatToolAnnotation.readerIdleTimeSeconds(),

heartBeatToolAnnotation.writerIdleTimeSeconds(),

heartBeatToolAnnotation.allIdleTimeSeconds())

);
 }
 pipeline.addLast(clientHandler);
 }
 });

 //进行连接
 bootstrap.connect(hostname, port).sync();

} catch (InterruptedException e) {
 e.printStackTrace();
}

```

```

 }
 //我是有多个地方进行调用的 不能只连接一个
 // initClient(hostName,port,method);
 clientHandler.setParam(param);
 clientHandler.setMethod(method);
 //接下来这就有关系到调用 直接调用
 Object response = executor.submit(clientHandler).get();
 nettyClientHandlerThreadLocal.remove(); //一个handler不能加到两个管道中 你说是
 吧

 return response;
}
}

```

- 根据霍夫曼编码实现diyZip工具类（暂未实现 我会回来的）

- 因为霍夫曼编解码需要一个map而多个线程肯定不能共享一个map会造成线程不安全，所以我设置了一个ThreadLocal单独一个线程一个map用完就销毁
- 未完成 待完成

## v2.7

### 更新：实现CGLIB动态代理

- 实现CGLIB动态代理

- 实现一下统一调用代理类，创建总调用类，和对应模板接口，调用注解，同时每个consumerbootstrap进行修改
  - 对应模板接口

```

package consumer.proxy;

//模板
public interface ClientProxy {
 public Object getBean(final Class<?> serviceClass);
}

```

- 总调用类(通过SPI机制进行调用) //用户千万要注意当你选用的是nio的话千万只能用nio的方法
  - SPI机制对应配置 存放实现类的全限定名和实现类



- SPI工具箱

```

package consumer.proxy;

import exception.RpcException;

import java.util.ServiceLoader;

public class SPIClientProxyUtils {
 public static ClientProxy getUtils() throws RpcException {
 ServiceLoader<ClientProxy> loader =
 ServiceLoader.load(ClientProxy.class);
 for (ClientProxy clientProxy : loader) {
 return clientProxy;
 }
 throw new RpcException("您键入的代理，并未实现，欢迎实现提出pr");
 }
}

```

#### ◦ 实现CGLIB动态代理

##### ■ 依赖引入

```

<!--CGLIB依赖引入-->
<dependency>
 <groupId>cglib</groupId>
 <artifactId>cglib</artifactId>
 <version>3.3.0</version>
</dependency>

```

##### ■ 构建代理类实现代码

```

package consumer.proxy.netty;

import annotation.RegistryChosen;
import consumer.netty.NettyClient;
import consumer.proxy.ClientProxy;
import consumer.service_discovery.NacosServiceDiscovery;
import consumer.service_discovery.ZkCuratorDiscovery;
import consumer.service_discovery.ZkServiceDiscovery;
import exception.RpcException;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import register.Register;

import java.lang.reflect.Method;

//Cglib实现代理模式
public class RpcNettyClientCGLIBProxy implements
ClientProxy,MethodInterceptor{

```

```

@Override
public Object getBean(Class serviceClass) {
 //设置动态代理增强类
 Enhancer enhancer = new Enhancer();
 //设置类加载器
 // enhancer.setClassLoader(serviceClass.getClassLoader());
 //设置代理类
 enhancer.setSuperclass(serviceClass);
 //设置对应的方法执行拦截器，方法回调
 enhancer.setCallback(this);

 return enhancer.create();
}

/**
 * @param obj 代理对象（增强的对象）
 * @param method 被拦截的方法（需要增强的方法）
 * @param args 方法入参
 * @param methodProxy 用于调用原始方法
 */
@Override //自定义对应的拦截 拦截方法并执行别的任务
public Object intercept(Object obj, Method method, Object[] args,
MethodProxy methodProxy) throws Throwable {
 String methodAddress = getMethodAddress(method.getName());
 String[] split = methodAddress.split(":");
 return
NettyClient.callMethod(split[0],Integer.valueOf(split[1]),args[0],method);
}

/**
 * 实际去获得对应的服务 并完成方法调用的方法
 * @param methodName 根据方法名 根据添加的注册中心注解来选择相应的注册中心进行 实
现负载均衡获取一个方法对应地址
 * @param
 * @return
 */
private static String getMethodAddress(String methodName) throws Exception
{
 //根据注解进行方法调用
 //根据在代理类上的注解调用 看清楚底下的因为是个class数组 可以直接继续获取 注解
 RegistryChosen annotation =
Register.class.getAnnotation(RegistryChosen.class);
 switch (annotation.registryName())
 {
 case "nacos":
 return NacosServiceDiscovery.getMethodAddress(methodName);
 }
}

```

```

 case "zookeeper":
 return ZkServiceDiscovery.getMethodAddress(methodName);
 case "zkCurator":
 return ZkCuratorDiscovery.getMethodAddress(methodName);
 default:
 throw new RpcException("不存在该注册中心");
 }
}
}

```

## BUG

### 问题

```

E:\Java\jdk1.8.0_261\bin\java.exe ...
Exception in thread "main" java.lang.IllegalStateException Create breakpoint : createClass does not accept callbacks
 at net.sf.cglib.proxy.Enhancer.validate(Enhancer.java:364)
 at net.sf.cglib.proxy.Enhancer.generate(Enhancer.java:486)
 at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData$3.apply(AbstractClassGenerator.java:96)
 at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData$3.apply(AbstractClassGenerator.java:94)
 at net.sf.cglib.core.internal.LoadingCache$2.call(LoadingCache.java:54) <1 internal line>
 at net.sf.cglib.core.internal.LoadingCache.createEntry(LoadingCache.java:61)
 at net.sf.cglib.core.internal.LoadingCache.get(LoadingCache.java:34)

```

### 解决

追进去看CGLIB构建代理类源码 是什么原因造成的这个错误，同时别人也是可以代理接口的啊

JDK动态代理代理接口，cglib可以代理类和接口

1. 经过验证 应该是和我们传进去的方法无关的
2. 找到错误了 进行创建应该是enhancer.create()方法 我给写成了enhancer.createClass()

### 问题声明

在调用方法这边这么写的原因是，无法获取对应调用类上的版本信息，会导致循环依赖，如果读者想更简单调用每个版本都对应一个对应的NettyClient的类 可以直接通过对应的NettyClient进行调用

```

import java.lang.reflect.Method;

//如果对应的是字符串类那就不用下面的这个里面调用了
public class NettyClient {
 public static Object callMethod(String hostName, int port, Object param, Method method) throws Exception {
 //用户根据自己想用的版本 打开对应的注解
 // return NettyClient21.callMethod(hostName, port, param, method);
 // return NettyClient22.callMethod(hostName, port, param, method);
 return NettyClient24.callMethod(hostName, port, param, method);
 }
}

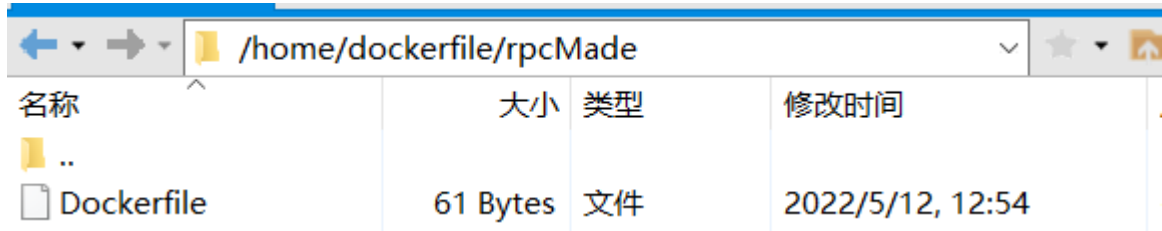
```

## v2.8

### 更新: dockerfile制作注册中心镜像

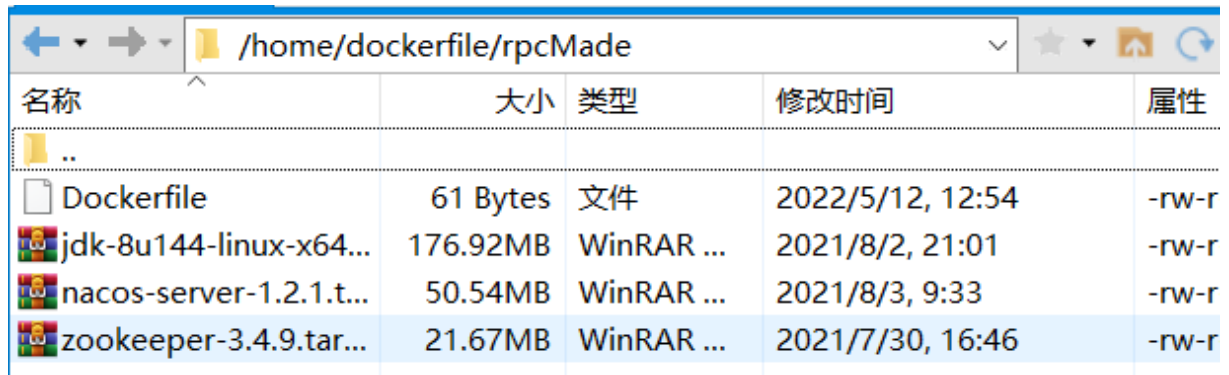
- 制作注册中心镜像，可以一键启动nacos和zookeeper

- 在linux下自己指定的文件夹中新建Dockerfile，名字最好就是Dockerfile之后在构建的时候，会自动寻找



名称	大小	类型	修改时间
..			
Dockerfile	61 Bytes	文件	2022/5/12, 12:54

- 传入我们需要的压缩包，需要jdk、nacos、zookeeper因为zookeeper的是由java开发的所以需要jdk



名称	大小	类型	修改时间	属性
..				
Dockerfile	61 Bytes	文件	2022/5/12, 12:54	-rw-r--r--
jdk-8u144-linux-x64.tar.gz	176.92MB	WinRAR ...	2021/8/2, 21:01	-rw-r--r--
nacos-server-1.2.1.tar.gz	50.54MB	WinRAR ...	2021/8/3, 9:33	-rw-r--r--
zookeeper-3.4.9.tar.gz	21.67MB	WinRAR ...	2021/7/30, 16:46	-rw-r--r--

- 进行编写dockerfile

```
FROM centos:centos7
MAINTAINER zyt<zyt061303130215@163.com>

ADD jdk-8u144-linux-x64.tar.gz /usr/local
ADD zookeeper-3.4.9.tar.gz /usr/local
ADD nacos-server-1.2.1.tar.gz /usr/local

RUN yum -y install vim

ENV MYPATH /usr/local
WORKDIR $MYPATH

ENV JAVA_HOME /usr/local/jdk1.8.0_144
ENV CLASSPATH $JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
ENV PATH $PATH:$JAVA_HOME/bin:

EXPOSE 2181 2888 3888 8080 8848

然后就是启动命令nacos 和 zookeeper

ENV ZOOKEEPER_HOME /usr/local/zookeeper-3.4.9
环境变量设置
RUN cp $ZOOKEEPER_HOME/conf/zoo_sample.cfg $ZOOKEEPER_HOME/conf/zoo.cfg

CMD $ZOOKEEPER_HOME/bin/zkServer.sh start&&$MYPATH/nacos/bin/startup.sh -m standalone&&/bin/bash
```

- 进行镜像打包

```
docker build -t zytregistry:1.0 .
```

- 启动命令

```
docker run -it -d -p 8848:8848 -p 2181:2181 --restart always zytregistry
```

**试验成功！！**

- 镜像发布

- 打标签

```
docker tag zytregistry 836585692/zytregistry:1.0
```

- 发布

```
docker push 836585692/zytregistry:1.0
```

- 之后可以直接一键启动

```
docker pull 836585692/zytregistry:1.0&&docker run -it -d -p 8848:8848 -p 2181:2181 --restart always 836585692/zytregistry:1.0s
```

- 遇到的bug

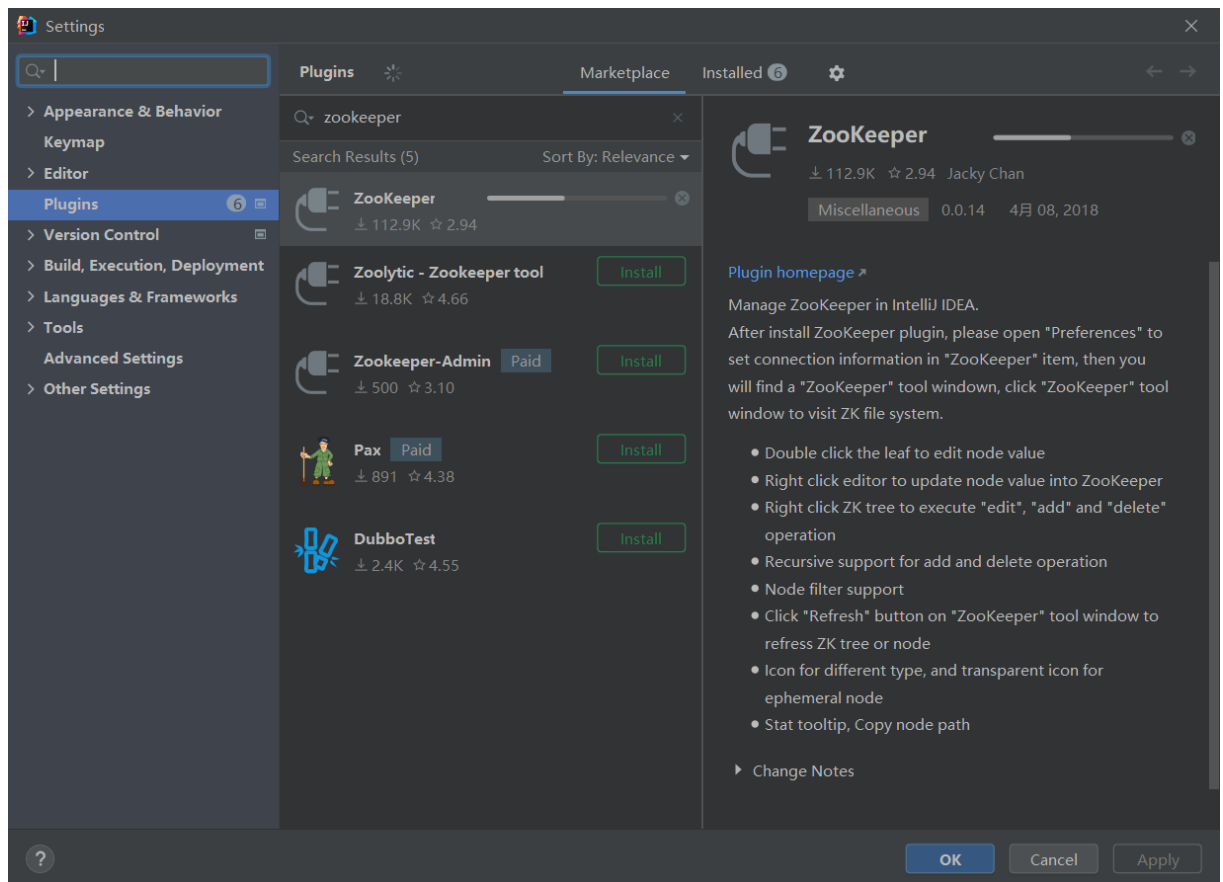
- 问题：启动后直接停止
- 解决：通过在dockerfile里cmd中同时启用/bin/bash就可以使得镜像保持启动

## v2.9

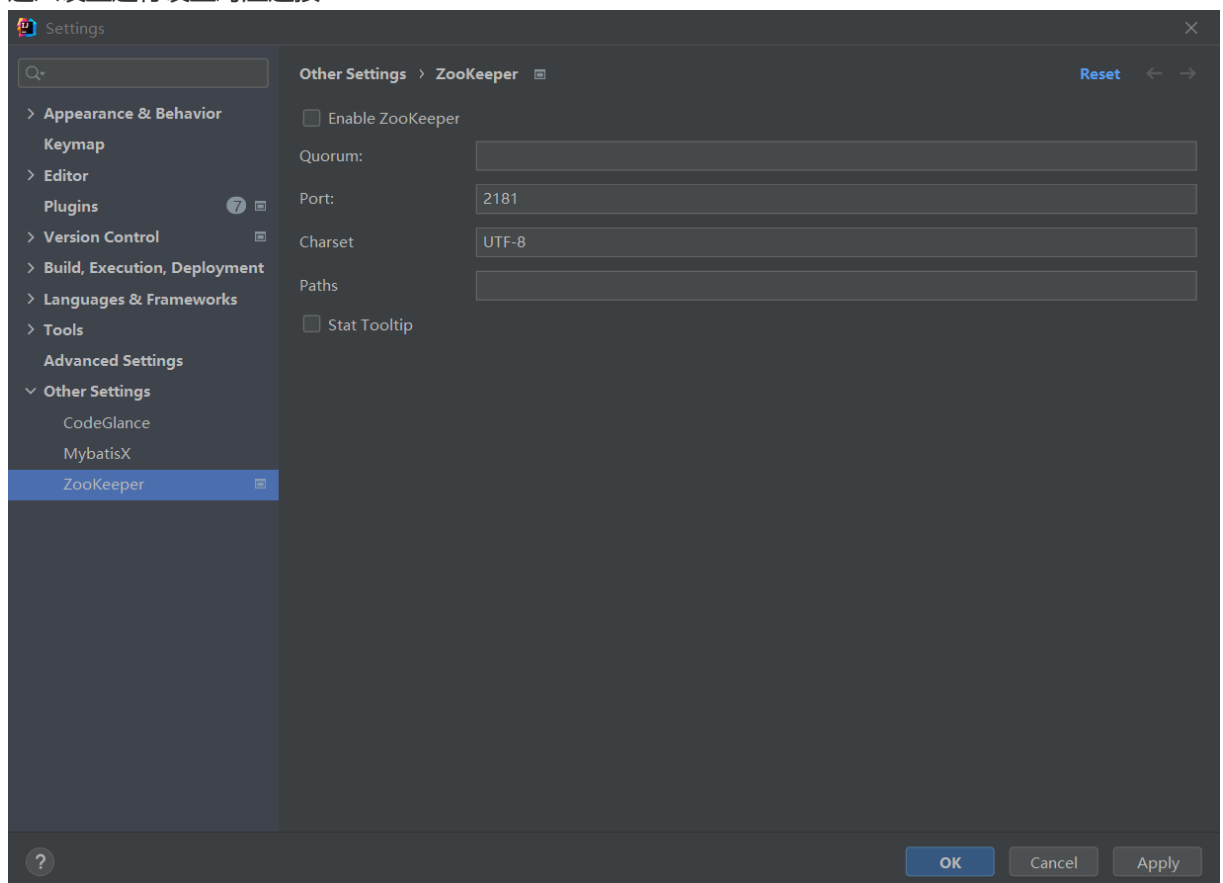
**更新：**主要就是进行idea上zookeeper的插件的安装，更加方便的看到相应的节点变换，同时对nio相关的代码回顾并整理

- **zookeeper插件安装**

- 去到plugins进行安装相应的插件zookeeper,安装成功后进行重启idea

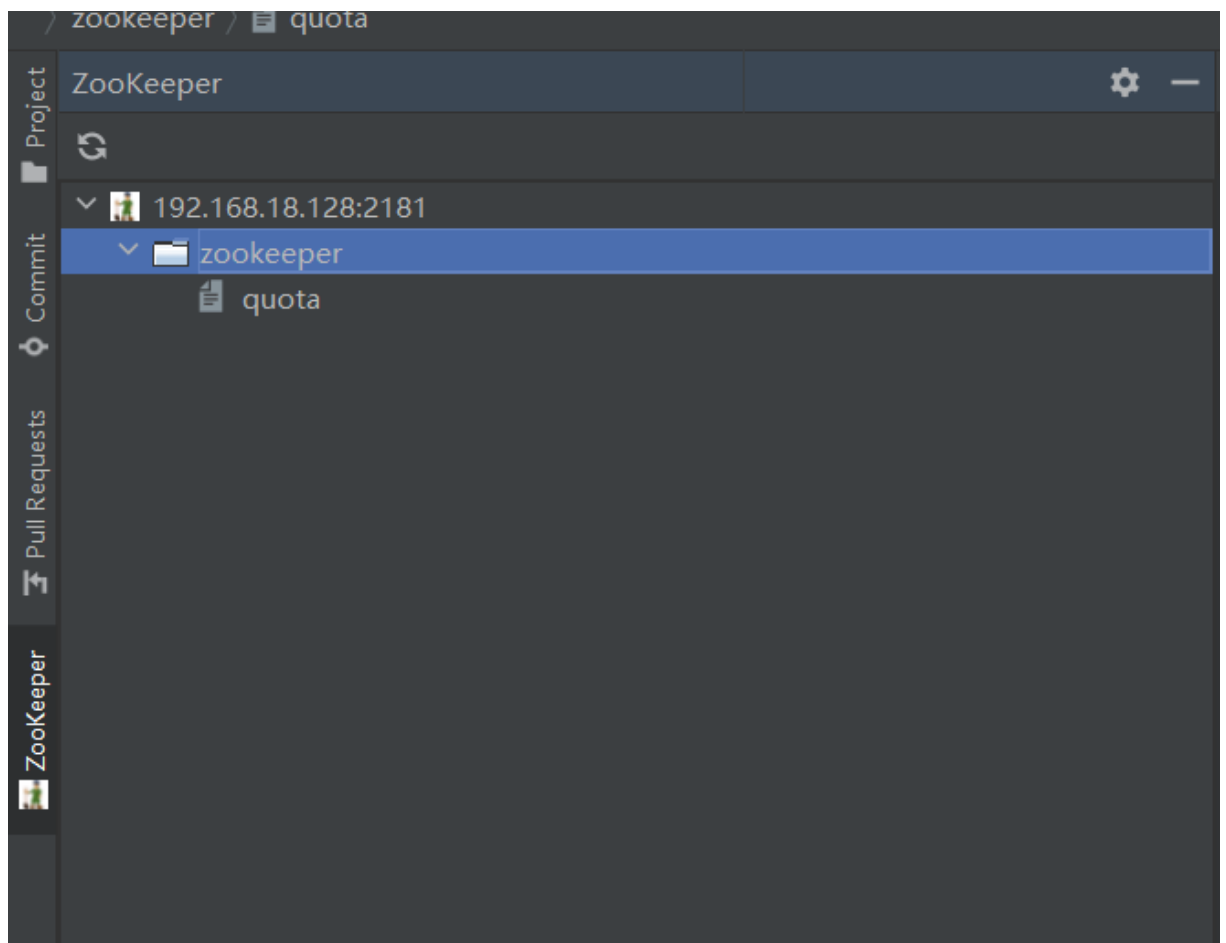


- 进入设置进行设置对应连接

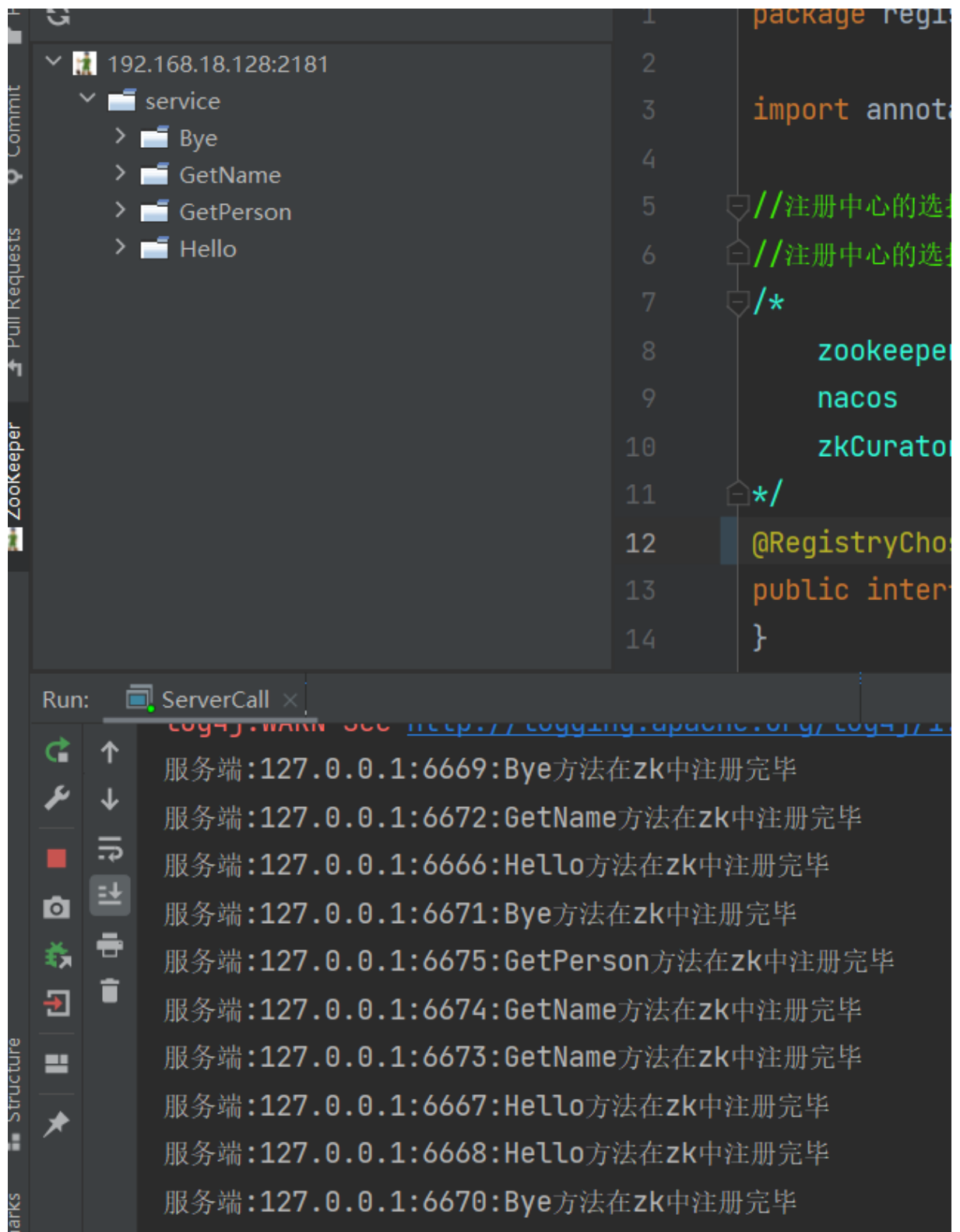


- 可以看到连接上后，效果如图所示





- 随后启动进行验证



- 连接成功，并注册成功，我们之后可以直接在这里看我们的均衡策略是否有效，减少了切换 查找的负担，当然如果对zk的指令没有这么熟练的朋友，我还是建议去linux下看哦

- 优化nio部分代码，顺带回顾

- 服务端
- 消费端
- 公共类

优化过程，大伙儿直接看和2.8版本的区别即可

## 热补丁

- 优化netty部分代码

## v2.10

更新进行1、代码的规范化 格式的优化 2、同时进行全局异常的精简和封装 新增全局异常处理类 3、将输出信息 转换为日志输出

- 因为System.out是属于同步的 所以一定程度上会使得业务程序进行等待，用log会好很多 所以全部转变为用log 打印日志✓

- **问题** 日志无法打印 **原因** 应该是配置问题

```
log4j.rootLogger=info, ServerDailyRollingFile, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%p] %C{1} - %m%n
```

- 这样的配置打出来的，会显示相应其他信息，导致观感很混乱  
配置文件解析 首先log4j.rootLogger中的第一个就是我们能打印的最低日志级别，一般是通过这个来设置的stdout就是控制台，后面两个就是信息输出的地方。
- 我们输出的日志级别是info，但会导致同时输出的日志非常多 这是之后或者其他朋友可以优化的
- **问题** 打印日志中文乱码 **解决方法** 在idea.exe.vmoptions 添加-Dfile.encoding=UTF-8
- 自定义异常 异常处理✓
  - 出现异常之后 直接就地处理 因为不是web类型 所以需要就地处理 不能给用户看见 需要优雅
- 将作者信息加入✓
- 统一一下配置类
  - 在过时的配置类加上了@Deprecated

```
package configuration;

import annotation.*;
import loadbalance.RandomLoadBalance;

/**
 * @Author 祝英台炸油条
 * @Time : 2022/5/20 20:42
 * 全局配置 将所有的配置都配置在它上面
 * 解压器 序列化器 注册中心 负载均衡 心跳机制等
 */
```

```

/*
 @RegistryChosen

 zookeeper zk注册中心
 nacos nacos实现注册中心
 zkCurator Curator协助操作注册中心
*/

/*
 @CodecSelector

 ObjectCodec
 protoc
 kryo
 protostuff
 hessian
 fst
 avro
 jackson
 fastjson
 gson
*/

/*
 @LoadBalanceMethodImpl

 RandomLoadBalance.class
 AccessLoadBalance.class
 ConsistentLoadBalance.class
*/

/*
 @CompressSelector

 BZipUtils
 DeflaterUtils
 GZipUtils
 LZ4Utils
 ZipUtils
*/

@CompressFunction(isOpenFunction = true)
@CompressSelector(CompressTool = "DeflaterUtils")
@HeartBeatTool(isOpenFunction = true,
 readerIdleTimeSeconds = 4,
 writerIdleTimeSeconds = 4,
 allIdleTimeSeconds = 2)
@LoadBalanceMethodImpl(chosenMethod = RandomLoadBalance.class)
@RegistryChosen(registryName = "zookeeper")

```

```
@CodecSelector(Codec = "fastjson")
public class GlobalConfiguration {
}
```

- 优化格式 ✓

## 热补丁

### 进行一些判断逻辑上的优化，解决日志乱码问题

- 例如传入方法和启动相应的服务器数量不一致等逻辑
- 解决日志乱码问题
  - 在相应的配置文件上配置GBK

```
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%p] %C{1} - %m%n
log4j.appender.CONSOLE.Encoding=GBK
```

## v2.11

这是个工程 将自己的用户中心改造一块 当作rpc的监控中心，同时尝试能否将服务降级等相关的方法也完善，之后还有什么歌曲管理中心等 大伙在使用我这块的同时 可以用下我的用户中心项目

- 用户中心改造成监控中心
  - 设计监控中心对应数据库

```
create table rpc_monitor
(
 id bigint auto_increment comment 'id' primary key,
 method_name varchar(256) null comment '方法名称',
 method_description varchar(256) null comment '方法描述',
 call_time datetime default CURRENT_TIMESTAMP null on update CURRENT_TIMESTAMP comment '调用时间',
 call_num int default 0 not null comment '方法调用次数'
)
```

- 前端将原来的界面调整 新增监控界面 有点为难我了 这块 但是为了完整性拼了实现界面



- 在我的用户中心简易的编写一下后端的逻辑之后进行完善 相应数据库的类 就用mybatisX生成了

```
/**
 * @Author 祝英台炸油条
 * @Time : 2022/6/3 14:12
 **/
@RestController
@RequestMapping("/method")
public class RpcMonitorController {

 @Resource
 private RpcMonitorService rpcMonitorService;

 @GetMapping("/search")
 public BaseResponse<List<RpcMonitor>> methodSearch() {
 return ResultUtils.success(rpcMonitorService.list());
 }
}
```

- 到rpc框架编写相应的逻辑 当服务被调用的时候 相应的进行修改 同时当服务提供方停止后又重新启动时需将数据库中数据清空
  - 首先要与数据库进行操作 必须要有持久层 数据库连接池 jdbc 或者 mybatis等。第一步引入依赖

```
<!-- 数据库连接池 -->
<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <scope>runtime</scope>
</dependency>

<!--mybatis用来和数据库进行交互-->
<dependency>
 <groupId>org.mybatis.spring.boot</groupId>
 <artifactId>mybatis-spring-boot-starter</artifactId>
 <version>2.1.4</version>
</dependency>
<dependency>
 <groupId>com.baomidou</groupId>
```

```
<artifactId>mybatis-plus-boot-starter</artifactId>
<version>3.5.1</version>
</dependency>
```

- (目前只进行了netty版本的修改) 接下来 最简单的 我的想法是每次启动的时候 就将原先的数据库中的字段全部清空 然后把所有的服务还有对应的地址注册进去, 在每次调用的同时 进行数据的新增
- 遇到bug 一直空指针异常 mybatis还是差的太多了 回来好好写 问题: 我觉得是我不能按照和springboot整合的方式进行整合 因为整合的时候 用properties 这只能作用于springboot//@Autowired不能作用于静态对象 //真实原因就是我不是springboot

```
[INFO] ClientCnxn$EventThread - EventThread shut down for session: 0x181274661250025
Exception in thread "main" java.lang.NullPointerException Create breakpoint
at provider.bootstrap.netty.NettyProviderBootstrap24.main(NettyProviderBootstrap24.java:31)
at service.netty_bootstrap.NettyServerBootstrap.start(NettyServerBootstrap.java:73)
at service.call.netty_call.NettyServerCall.main(NettyServerCall.java:13)
at service.call.ChosenServerCall.start(ChosenServerCall.java:23)
at service.ServerCall.main(ServerCall.java:17)
```

- 采用spring整合mybatis 属于就是回到最开始的地方 回顾一下之前的东西
- 遇到bug 依赖的重复引用 会导致找不到一些 jar这时候 就是要减少部分依赖
- 如果我们在项目中配置的driver-class-name为com.mysql.jdbc.Driver, 则对应的mysql-connector-java版本应该是5.x。

如果我们在项目中配置的driver-class-name为com.mysql.cj.jdbc.Driver, 则对应的mysql-connector-java版本应该是8.x。

- 成功终于整合了mybatis-plus和spring [mybatis-plus-samples/mybatis-plus-sample-quickstart-springmvc at master · baomidou/mybatis-plus-samples \(github.com\)](https://github.com/baomidou/mybatis-plus-samples) 参考

#### o 整合配置文件和引入依赖

- 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

 <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
 <property name="basePackage" value="provider.monitor.mapper"/>
 </bean>

 <bean id="sqlSessionFactory"
class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
 <property name="dataSource" ref="dataSource"/>
 </bean>

 <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
 <property name="driverClass" value="com.mysql.cj.jdbc.Driver"/>
```

```

 <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/zeng?
serverTimezone=UTC"/>
 <property name="user" value="root"/>
 <property name="password" value="root"/>
 </bean>

</beans>

```

## ■ 引入依赖

```

<!--数据库连接池-->
<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <version>8.0.28</version>
</dependency>
<!-- 数据库连接池: c3p0-->
<dependency>
 <groupId>com.mchange</groupId>
 <artifactId>c3p0</artifactId>
 <version>0.9.5.2</version>
</dependency>

<!--mybatis用来和数据库进行交互-->

<dependency>
 <groupId>com.baomidou</groupId>
 <artifactId>mybatis-plus</artifactId>
 <version>3.4.3.4</version>
</dependency>

<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-jdbc</artifactId>
 <version>5.2.15.RELEASE</version>
</dependency>

```

- 进行相应的方法设置 因为我们的方法都是通过main方法进行的 该方法是由static修饰的 我们使用@Autowired什么的不能用static 所以我采用使其在其他类中创建对应方法 我再引用即可

- 将监控类的初始化 改为类加载，在整个运行过程中 只加载一次 有效的减少了相应的消耗



```

static RpcMonitorMapper rpcMonitorMapper;

static {
 ApplicationContext context = new ClassPathXmlApplicationContext("spring-
mybatis.xml");
 rpcMonitorMapper = context.getBean(RpcMonitorMapper.class);
}

```

#### ■ 删除方法

```

/**
 * 作用 就是每次开启服务提供商的时候 删除掉所有的对应项目
 */
public void deleteAll() {
 rpcMonitorMapper.delete(null);
}

```

#### ■ 添加方法

```

/**
 * 每次开启服务的时候 就进行添加对应的服务 当然 起始调用次数为0
 */
public void addServer(RpcMonitor rpcMonitorServer) {
 rpcMonitorMapper.insert(rpcMonitorServer);
}

```

#### ■ 更改方法

```

/**
 * 更新相应的服务 将相应的服务调用次数+1 同时update的时候 会自动的更改调用时间 做
这个的时候 要上锁 防止线程冲突
 */
public synchronized void updateServer(String methodAddress) {
 QueryWrapper<RpcMonitor> wrapper = new QueryWrapper<>();
 // 没问题 查询的时候 是跟对应的列名进行比较
 wrapper.eq("method_name", methodAddress);
 RpcMonitor rpcMonitor = rpcMonitorMapper.selectOne(wrapper);
 if (rpcMonitor == null) try {
 throw new RpcException("监控中心出现错误");
 } catch (RpcException e) {
 log.error(e.getMessage(), e);
 return;
 }
 rpcMonitor.setCallNum(rpcMonitor.getCallNum()+1);
 rpcMonitorMapper.update(rpcMonitor, new QueryWrapper<RpcMonitor>
().eq("id", rpcMonitor.getId()));
}

```

- 再对应的地方 进行相应方法的调用

- 服务端 一开始将原先的数据库进行清空

```
RpcMonitorOperator rpcMonitorOperator = new RpcMonitorOperator();
rpcMonitorOperator.deleteAll();
```

- 服务端 一开始将所有的注册进去 还有就是端口的问题

```
RpcMonitor rpcMonitor = new RpcMonitor();
//TODO 这里之后还可以继续进行更改 因为这块的话 如果放在服务器上 那么就可
以采用服务器相关的设置
rpcMonitor.setMethodName("127.0.0.1:"+ port);
rpcMonitor.setMethodDescription(methodName);
rpcMonitorOperator.addServer(rpcMonitor);
int nowPort = port.get();
//因为下面这个开启一个线程 会慢一点
new Thread(() -> NettyServer24.start(methodName,
nowPort)).start();
port.incrementAndGet();
```

- 客户端 将调用的服务进行 增加调用次数

```
RpcMonitorOperator rpcMonitorOperator = new RpcMonitorOperator();
rpcMonitorOperator.updateServer(methodAddress);
```

## 效果图

### 远端调用监控

调用方法地址:

调用方法描述:

[重置](#)

[查询](#)

[展开](#) 

高级表格

	调用方法地址	调用方法描述	调用次数	最近调用时间	操作
1	127.0.0.1:6671 	GetName 	3	2022-06-04 06:05:51	<a href="#">编辑</a> <a href="#">查看</a> ...
2	127.0.0.1:6672 	GetName 	0	2022-06-04 06:05:51	<a href="#">编辑</a> <a href="#">查看</a> ...
3	127.0.0.1:6673 	GetName 	1	2022-06-04 06:05:51	<a href="#">编辑</a> <a href="#">查看</a> ...
4	127.0.0.1:6674 	GetPerson 	2	2022-06-04 06:05:52	<a href="#">编辑</a> <a href="#">查看</a> ...

第 6-9 条/总共 9 条 < 1 **2** >

## 热补丁

将依赖版本都升级到最新 旧版都能运行的备注了 如果读者用到哪个无法使用的时候 可以使用旧版

更新整合配置 将数据库的信息提取 方便更改

- 数据库信息整合入properties 通过el表达式提取

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

 <!-- Import Properties -->
 <context:property-placeholder location="classpath:db.properties"/>

 <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
 <property name="basePackage" value="monitor.mapper"/>
 </bean>

 <bean id="sqlSessionFactory"
class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
 <property name="dataSource" ref="dataSource"/>
 </bean>

 <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
 <property name="driverClass" value="${jdbc.driver}"/>
 <property name="jdbcUrl" value="${jdbc.url}"/>
 <property name="user" value="${jdbc.username}"/>
 <property name="password" value="${jdbc.password}"/>
 </bean>
</beans>
```

```
w#自测版本
jdbc.url=jdbc:mysql://localhost:3306/zeng?serverTimezone=UTC
jdbc.username=root
jdbc.password=root
jdbc.driver=com.mysql.cj.jdbc.Driver

#连接外部数据库版本
#jdbc.url=jdbc:mysql://121.43.39.136:3306/zeng?serverTimezone=UTC
#jdbc.username=zyt
#jdbc.password=zyt
#jdbc.driver=com.mysql.cj.jdbc.Driver
```

- 在服务器数据库上建表 然后将前端部署 这样可以直接通过自己的域名监控调用次数
- 部署的时候 出现问题 前端在搜索调用方法的时候 出现404 但明明搜索用户也没错啊

```
GET https://user-backend.onlyicanstopmyself.top/api/method/search 404
```

**解决方法：问题并不出在前端或者是后端上面 而是自己在宝塔上重新上传了jar包 然后没有重新运行，这是不该犯的错，一定要精通部署这块**

PHP项目

Java项目

Node项目

添加Java项目

JDK管理

Tomcat管理

Java项目教程

请输入项目名称或备注

项目名称	服务状态	项目类型	端口	CPU	内存	根目录	备注	SSL证书	操作
user-center-0	运行中▶	SpringBoot	--	0.00%	55.63 MB	/www/wwwroot/usercenter_backend/user-center-0.0.1-SNAPSHOT.jar	user-center-0	未部署	设置   删除

- 部署成功 效果也正确 出现新的问题 当继续进行修改之后时间并没有修改
  - 解决：每次进行修改的同时 将调用时间置空 这块不需要重新部署 因为只是自身逻辑这块有问题

```
rpcMonitor.setCallNum(rpcMonitor.getCallNum()+1);
rpcMonitor.setCallTime(null);
rpcMonitorMapper.update(rpcMonitor,new QueryWrapper<RpcMonitor>().eq(column:"id",rpcMonitor.getId()));
```

- 出现新问题 前端显示时间和数据库差8个小时 目前查到原因是去读取的时候 发现有时差
  - 解决：通过修改连接数据库的url 需要和我们的数据库的时区一致 否则不同时区 会发生相应的改变 一些增加和减少时间的现象

```
url: jdbc:mysql://121.43.39.136:3306/zeng?serverTimezone=GMT%2B8
```

- 前端bug 如果没有用户名则右上角菜单会不断读取
  - 解决：更改数据库名段 将默认名称设为visitor

优化 排序 还有试试图像可视化等 还有在user-center的权限 服务降级等

## TODO

服务监控中心（服务降级等方法 之后尝试一下）、还有传输协议 还有其他的均衡方法 加权轮询

# RPC框架Springboot版

---

## 移植

- 问题：移植过程中出现依赖无法导入问题

解决：依赖位置放的不对 放到了里面 这是版本号管理的地方

- 问题：启动出现问题 slf4j包中有冲突

```
LoggerFactory is not a Logback LoggerContext but Logback is on the classpath. Either
remove Logback or the competing implementation (class
org.slf4j.impl.Reload4jLoggerFactory loaded from
file:/E:/Java/maven/repository/org/slf4j/slf4j-reload4j/1.7.36/slf4j-reload4j-
1.7.36.jar). If you are using WebLogic you will need to add 'org.slf4j' to prefer-
application-packages in WEB-INF/weblogic.xml: org.slf4j.impl.Reload4jLoggerFactory
at org.springframework.util.Assert.instanceCheckFailed(Assert.java:702)
```

解决：Spring Boot 项目一般都会引用 spring-boot-starter 或者 spring-boot-starter-web，而这两个起步依赖中都已经包含了对于 spring-boot-starter-logging 的依赖，所以，我们无需额外添加依赖。测试类也是加入了就不需要再加入了

- 问题：报错 Error creating bean with name 'requestMappingHandlerAdapter' defined in class path resource.....

解决：和jackson依赖有冲突 回退到2.11.4版本

- 成功 开始测试 就是整合的有点问题

- 问题 相关的数据库启动类 之类的Property 'sqlSessionFactory' or 'sqlSessionTemplate' are required 这些放了 依旧还是上面的问题

```
<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <version>8.0.16</version>
</dependency>
```

<!--主要提供了三个功能：第一个是对数据源的装配，第二个是提供一个JdbcTemplate简化使用，第三个是事务。-->

<!--其实这里并不用配置 因为底下的mybatis的启动类已经引入了对应包-->

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

解决：忘记引入mybatis了

```
<dependency>
 <groupId>org.mybatis.spring.boot</groupId>
 <artifactId>mybatis-spring-boot-starter</artifactId>
 <version>1.1.1</version>
</dependency>
```

- 遇到问题

An attempt was made to call a method that does not exist. The attempt was made from the following location:  
org.mybatis.spring.SqlSessionFactoryBean.buildSqlSessionFactory

解决：下载了一个Maven Help去检查冲突的依赖 [解决maven依赖冲突问题：An attempt was made to call a method that does not exist... MTONJ的博客-CSDN博客](#)

这个 比较重要我感觉 报错不停 持续的问题 依赖

就根据之前成功的配 不然这些有太多的问题 我这次的问题就是引入了太多的导致重复冲突 还在努力的去配置 根据注解 排除掉了没有扫描到

- 大bug（非controller 非controller、service引用,静态引用注入不进去）★好好看看

解决: [解决非controller使用@Autowired注解注入为null问题 \(@PostCon。。。 - 百度文库 \(baidu.com\)\)](#)

在SpringMVC框架中, 我们经常要使用@Autowired注解注入Service或者Mapper接口, 我们也知道, 在controller层中注入service接口, 在service层中注入其它的service接口或者mapper接口都是可以的, 但是如果我们要在我们自己封装的Utils工具类中或者非controller普通类中使用@Autowired注解注入Service或者Mapper接口, 直接注入是不可能的, 因为Utils使用了静态的方法, 我们是无法直接使用非静态接口的, 当我们遇到这样的问题, 我们就要想办法解决了。

@Autowired注解的方法:

```
@Component
public class TestUtils {
 @Autowired
 private ItemService itemService;

 @Autowired
 private ItemMapper itemMapper;

 public static TestUtils testUtils;

 @PostConstruct
 public void init() {
 testUtils = this;
 }
}
```

```
package com.rpc.zeng.common.monitor;

import com.baomidou.mybatisplus.core.conditions.query.Querywrapper;
import com.rpc.zeng.common.exception.RpcException;
import com.rpc.zeng.common.monitor.service.RpcMonitorService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

/**
 * @Author 祝英台炸油条
 * @Time : 2022/6/3 19:47
 */
@Slf4j
@Component
public class RpcMonitorOperator {

 @Autowired
 private RpcMonitorService rpcMonitorService;

 public static RpcMonitorOperator rpcMonitorOperator;

 @PostConstruct
 public void init(){
 rpcMonitorOperator = this;
 }

 /**
```

```

 * 作用 就是每次开启服务提供商的时候 删除掉所有的对应项目
 */
 public void deleteAll() {
 rpcMonitorOperator.rpcMonitorService.remove(null);
 }

 /**
 * 每次开启服务的时候 就进行添加对应的服务 当然 起始调用次数为0
 */
 public void addServer(RpcMonitor rpcMonitorServer) {
 rpcMonitorOperator.rpcMonitorService.save(rpcMonitorServer);
 }

 /**
 * 更新相应的服务 将相应的服务调用次数+1 同时update的时候 会自动的更改调用时间 做这个的时候 要上锁 防止线程冲突
 */
 public synchronized void updateServer(String methodAddress) {
 QueryWrapper<RpcMonitor> wrapper = new QueryWrapper<>();
 // 没问题 查询的时候 是跟对应的列名进行比较
 wrapper.eq("method_name", methodAddress);
 RpcMonitor rpcMonitor =
rpcMonitorOperator.rpcMonitorService.getOne(wrapper);
 if (rpcMonitor == null) try {
 throw new RpcException("监控中心出现错误");
 } catch (RpcException e) {
 log.error(e.getMessage(), e);
 return;
 }
 rpcMonitor.setCallNum(rpcMonitor.getCallNum() + 1);
 rpcMonitor.setCallTime(null);
 rpcMonitorOperator.rpcMonitorService.update(rpcMonitor, new
QueryWrapper<RpcMonitor>().eq("id", rpcMonitor.getId()));
 }
}

```

- 提取了方法路径 防止因为路径的原因出错
- 成功实现! 过几天搞个前后端联动一下!
- 还要继续优化 很多方面 注解之类的选项

## 对比实验



测试效率，测试耗费时间

测试 nio netty

v1.5 v2.1

```
//注册中心选用zkCurator
//hello和bye方法分别注册三个
//采用循环100次分别调用hello，bye方法，查看时间差距 刚刚用了10000次有点问题

//执行代码
Customer customer = ChosenClientCall.start();

long start = System.currentTimeMillis();
for (int i = 0; i < 100; i++) {
 System.out.println(customer.Hello("success"));
 System.out.println(customer.Bye("fail"));
}
long end = System.currentTimeMillis();

System.out.println(end-start);
```

Nio v1.5 1、24705 2、24419

Netty v2.1 1、26773 2、27920

## 测试注册中心

## 测试序列化

## 测试压缩

## 遇到的问题

---

- @Data注解在类上无法使用 **解决** 我把一些没导版本号的依赖加上版本后恢复
- 额外开启一个线程进行监听读事件 第一次可以监听的到 后面就监听不到了 **解决** 迭代器iterator一定要记得remove否则就出错了
- 当关闭一个客户端的时候 服务端也自动关闭了 **解决** 更新v1.1
- 当利用io对request对象进行传输时出现 Exception in thread "main"  
java.nio.channels.IllegalBlockingModeException **原因** 假如socket是非阻塞的话 可以用selector 但不能用io流 因为这是阻塞传输方式。
- 还遇到了做阻塞启动的时候，我获取了一个通道的输入流和输出流 然后出问题直接死锁动弹不得 因为一个管道是半双工的 所以不能同时输入流输出流进行读写 我现在尝试去修改。 **修改成功** 我把输入流和输出流的创建顺序和他们的使用顺序保持 一致就成功了？ **继续查找原因** (8条消息) [ObjectInputStream与ObjectOutputStream的顺序问题](#) 到[中流遏飞舟的博客-CSDN博客](#)查看该网址，说的非常详细！如果两边创建的都是先ObjectInputStream会导致都阻塞在那边等待读写
  - 不多说直接追源码

```


public ObjectInputStream(InputStream in) throws IOException {
 verifySubclass();
 bin = new BlockDataInputStream(in);
 handles = new HandleTable(initialCapacity: 10);
 vlist = new ValidationList();
 serialFilter = ObjectInputFilter.Config.getSerialFilter();
 enableOverride = false;
 readStreamHeader();
 bin.setBlockDataMode(true);
}

```

```

protected void readStreamHeader()
 throws IOException, StreamCorruptedException
{
 short s0 = bin.readShort();
 short s1 = bin.readShort();
 if (s0 != STREAM_MAGIC || s1 != STREAM_VERSION) {
 throw new StreamCorruptedException(
 String.format("invalid stream header: %04X%04X", s0, s1));
 }
}

```

 **java.io.ObjectInputStream.ObjectInputStream(InputStream in) throws IOException**

Creates an ObjectInputStream that reads from the specified InputStream. A serialization stream header is read from the stream and verified. This constructor will block until the corresponding ObjectOutputStream has written and flushed the header.

If a security manager is installed, this constructor will check for the "enableSubclassImplementation" SerializablePermission when invoked directly or indirectly by the constructor of a subclass which overrides the ObjectInputStream.readFields or ObjectInputStream.readUnshared methods.

**Parameters:**  
 in input stream to read from

**Throws:**

<https://blog.csdn.net/gary0917>

第一段的意思是，创建一个从指定的InputStream读取的ObjectInputStream，序列化的流的头是从这个Stream中读取并验证的。此构造方法会一直阻塞直到相应的ObjectOutputStream已经写入并刷新头。

所以上述代码执行后会都阻塞，如果将创建ObjectInputStream的顺序修改成其他的顺序，便可正常通信。

- zookeeper各个端口的作用：

2181：对客户端提供服务

2888：Follower与Leader交换信息的端口。

3888：万一集群中的Leader服务器挂了，需要一个端口来重新进行选举，选出一个新的Leader，而这个端口就是用来执行选举时服务器相互通信的端口。

- 在设置与linux中zookeeper连接的时候 当我创建临时节点，通过服务器查询第一下查到了，然后马上查不到了，尝试解决。**解决**是我之前学的没有记牢，因为一旦客户端断开连接的话，那么临时节点也会断开连接 所以创建持久节点。
- zookeeper创建必须是它父节点要存在才能进行创建
- Zookeeper ZooDefs.Ids

**OPEN\_ACL\_UNSAFE** : 完全开放的ACL，任何连接的客户端都可以操作该属性znode

**CREATOR\_ALL\_ACL** : 只有创建者才有ACL权限

**READ\_ACL\_UNSAFE**: 只能读取ACL

- 当创建zk连接的时候 产生了空指针异常 是因为没有添加监听器
- 在设置多个线程同时启动时，共同注册到zk中，遇到了多线程问题
  - 存在问题** 测试出现问题 启动两个服务只有一个服务的方法注册进去了；
  - 解决方法** 因为我每个服务提供方到后面都是循环着监听 所以第一个进去了就出不来立刻 开了多个线程
  - 存在问题** 就是会重复进行创建service节点 应该是引入了线程安全的问题 就是单例模式懒汉模式1中类似的问题
  - 解决方法** 对方法进行加锁 使用了synchronized锁，该锁jdk1.6之后进行了优化可以使用！
  - 成功解决**
- 获取动态代理对象的三个参数详解

```
public static Object newProxyInstance(ClassLoader loader,
 Class<?>[] interfaces,
 InvocationHandler h)
 throws IllegalArgumentException
```

**newProxyInstance**，方法有三个参数：

**loader**: 用哪个类加载器去加载代理对象

**interfaces**: 动态代理类需要实现的接口

**h**: 动态代理方法在执行时，会调用h里面的**invoke**方法去执行

- github不能同时上传太多的文件！重要
- 多线程情况下，对临界资源的访问千万要记得加锁
- TCP传输会出现粘包拆包现象，UDP不会出现该现象
- UDP不存在粘包问题，是由于UDP发送的时候，没有经过Nagle算法优化，不会将多个小包合并一次发送出去。另外，在UDP协议的接收端，采用了链式结构来记录每一个到达的UDP包，这样接收端应用程序一次recv只能从socket接收缓冲区中读出一个数据包。也就是说，发送端send了几次，接收端必须recv几次（无论recv时指定了多大的缓冲区）。
- 线程安全问题 进行修改

```
zzz
U• ± •*U 成功连接
entity.Person@7212ba92
+Exception in thread "main" java.lang.reflect.UndeclaredThrowableException <1 internal line>
 at service.ClientCall.main(ClientCall.java:44)
+Caused by: java.nio.channels.ClosedChannelException <9 internal lines>
+ at io.netty.bootstrap.Bootstrap$3.run(Bootstrap.java:244) <7 internal lines>
/127.0.0.1:6675发生超时事件读写空闲: 连接关闭
```

## 亮点

自己手写了一个解压缩方法，依据霍夫曼编码 **未完成**

用到了模板模式、代理模式、工厂模式等设计模式

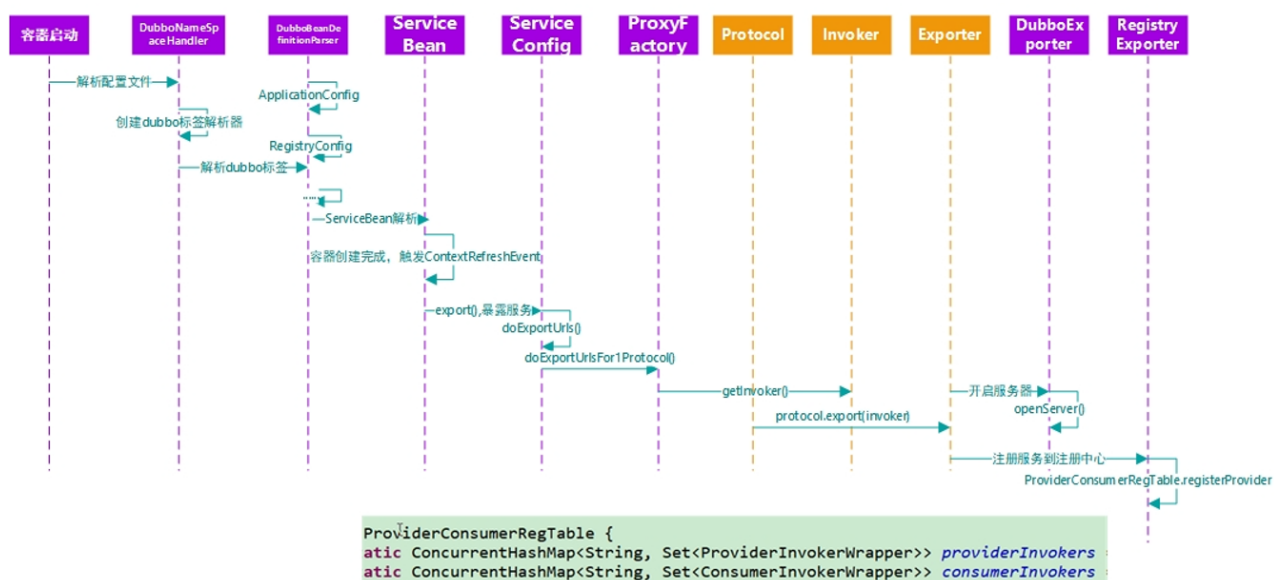
## Dubbo学习

这块之前就进行过学习，因为太久远了，现阶段进行快速过

- 灰度发布 就是 比如说更新了一个版本 一共有100台服务器正在运行，先让其中的20台用新版本 然后稳定后慢慢的过滤即可
- 相较于我目前完成的rpc框架，dubbo多了一个容器（当容器启动 服务就会对应启动 将自己注册到注册中心中去），和监控中心，监控中心是肯定会更新上去的(它的这块 还是比较好的，我的应该会比较简陋)
- 像是dubbo中的接口什么的 官方建议都是放到一个包内
- 相关的一些步骤[快速开始 | Apache Dubbo](#)目前用的还是2.7版本的，但现在dubbo3有很大的改变 如果之后要用3.0再看看相应的要求
- 没和springboot整合的时候 服务端和消费者端 如何配置 其实是非常一致的 都是用xml进行配置 看看课件 有机会去看
- 和springboot整合的时候 引入对应的依赖 启动包 要暴露的服务 只需要在类上配置dubbo包下的 **@Service** import org.apache.dubbo.config.annotation.Service; 还有就是在启动类上记得配置 这个注解的作用是将其注册到注册中心中 **@EnableDubbo** 使得可以用dubbo注解 **@Reference** 引用，Pom坐标，可以定义路径相同的接口名 这个是放在消费者端的 就不用再去配置 这个注解的作用是去注册中心中发现对应的服务
- dubbo本身是有很多属性可以配置的比如说启动检查、超时（像超时的话 会有默认值啥的）等 自己需要用的时候可以去找一下。同时配置可以统一配置 **配置覆盖关系：方法级优先，接口级次之，全局配置再次之 如果级别一样 消费方优先 提供方次之**
- dubbo有多种和springboot整合的方式
- 当zookeeper宕机了 还能调用服务 因为由本地缓存，就算没有注册中心也还能进行服务调用，因为可以进行dubbo直连
- dubbo本身有四种负载均衡算法 随机 随机权重 权重 一致性哈希
- 在dubbo的可视化界面上可以调整进行 降级熔断 同时也可以使用hystrix

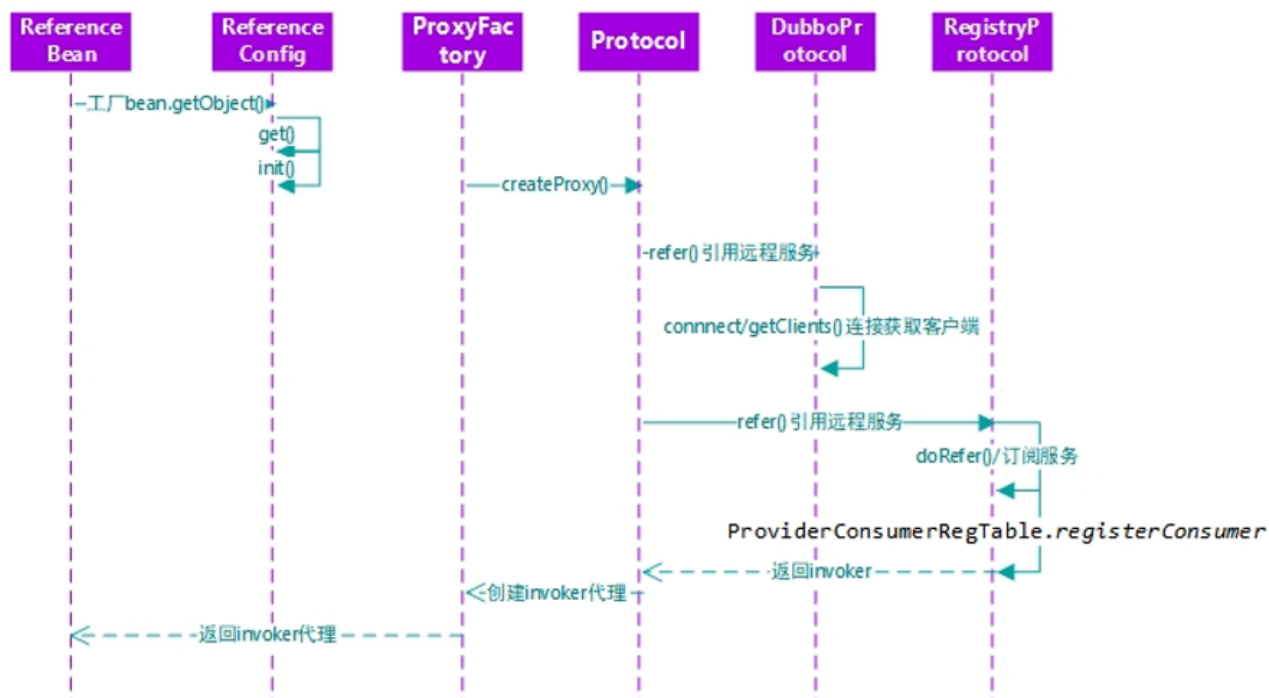
- 视频中 看了服务暴露 服务引用 服务调用

## • 服务暴露



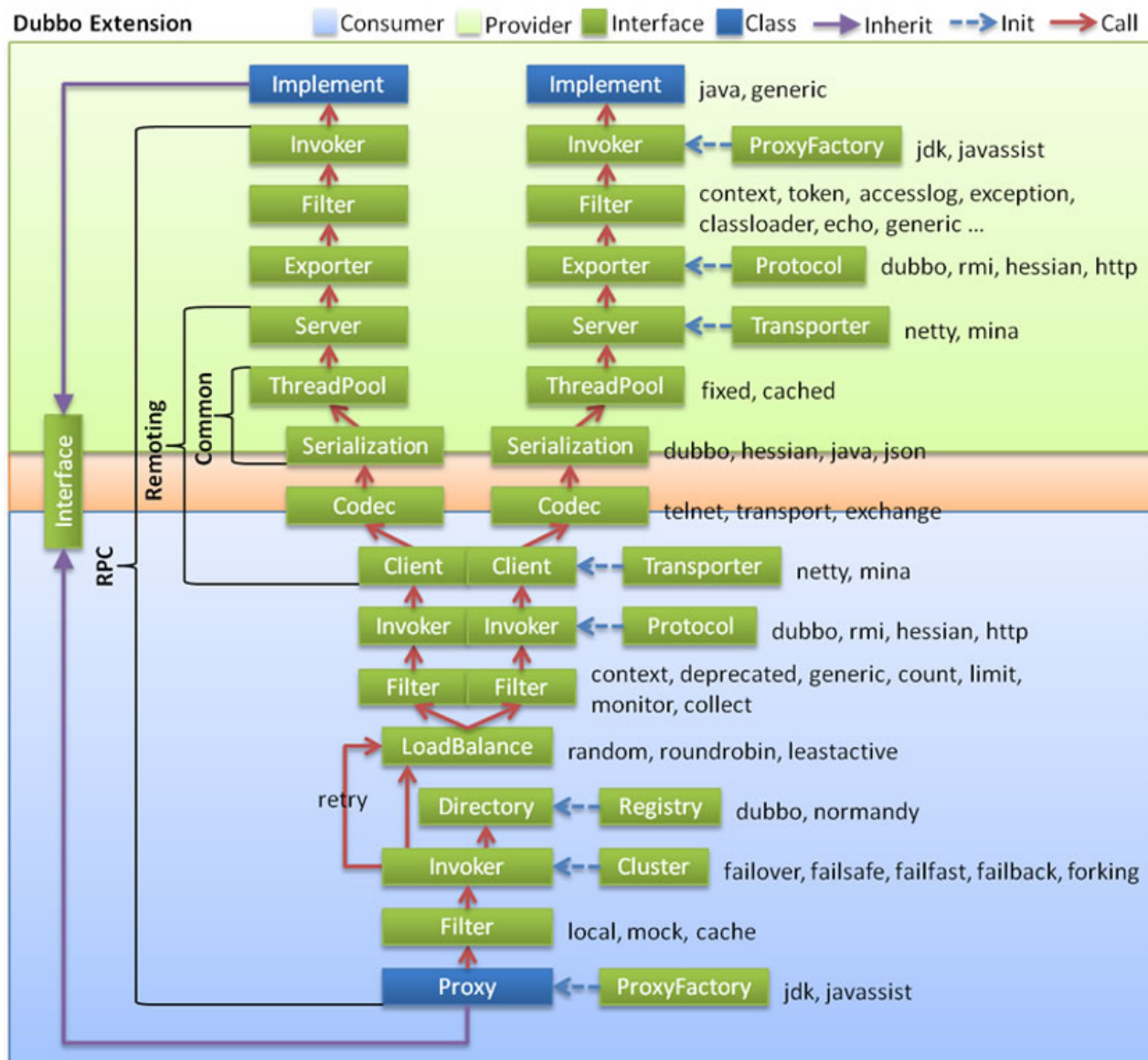
底层会先进行监听 同时会在注册中心对服务进行注册

- 服务引用 这个和服务暴露的步骤还是蛮像的



像每个注解他都会有对应的bean然后referencebean实现了工厂bean, 所以每次都会去工厂类中get或者init 对应的bean

- 服务调用 这块 还是需要进行 进一步去debug的



服务调用这块还是得看雷神 [尚硅谷Dubbo教程\(dubbo经典之作\)哔哩哔哩bilibili](#) 它解析的很明白 当获取到一系列的invokers之后会一步步的调用负载均衡机制等 最后选定一个invoker进行执行 (当然也可以配置各种filter)

## Dubbo源码阅读

### 总结相应的区别

- 首先最明显的体验就是 它的功能实现的策略比我多的多 我主要去看一些主要的源码
- spi全称为 (Service Provider Interface), 是JDK内置的一种服务提供发现机制。SPI是一种动态替换发现的机制, 一种解耦非常优秀的思想。dubbo对原先的spi机制进行了改造、优化, 同时兼容jdk spi 因为spi的存在, 该开源项目也是遵循开闭原则 对扩展开发, 对修改关闭
- 6.2先看扩展技术 spi的解析



- 源码中是一开始就加载对应实现类的 而不是像我写的那里面，再通过注解再获取，有时候可能获取步骤还更加复杂
- `dubbo=com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol`  
像我们jdk自带的spi是没有dubbo=这些的 就是只有后面的全限定名
- 获取相应分支的代码 git clone --branch 2.6.x <https://github.com/apache/dubbo.git>
- 源码中也是将对应的接口进行判断是否可以暴露，然后设置对应的信息，还要保证其泛型，最后暴露到各个注册中心。
- 

## 总结

---

跳过了BIO的方式直接进行了NIO的简易RPC的设计，比较简单，还有很多需要进行实现的东西。当然在搭轮子的途中我遇到了很多的坑，一点点的填，我自身也有了很大的长进，关于网络编程，各种添加功能，使得轮子更方便的进行使用