# 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）**

- **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）
- **kyro**
- **protobuf**（尝试使用这个进行）

## 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**
- **动态代理Cglib**

## 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）**

- **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
```

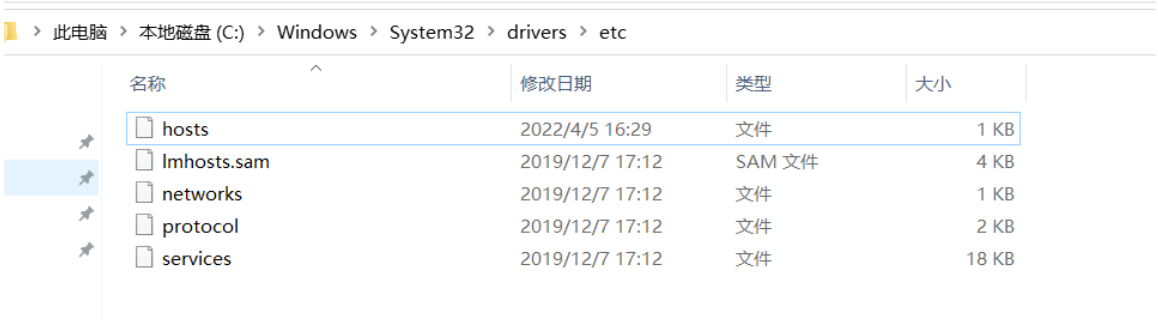
- zookeeper的运行
  保持zookeeper的开启状态，由于我之前启动没有加-d所以在执行连接它的命令时总会自动关闭，加了的目的是保证它能保持后台运行，--restart always这个指令的目的就是之后不用每次都再启动了

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

- 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）




```
# 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 = "zytCentos: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，此时服务器端会从新计算客户端的超时时间点的）
                           // 则服务器端认为session超时，清理数据。此时客户端的ZooKeeper对象就不再起作用了，需要再重新new一个新的对象了。
            null           //监听器 暂时不设置
    );
    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@zytCentos ~]# docker pull nacos/nacos-server:v2.0.3
    ```

  - 根据镜像创建容器

    ```
    [root@zytCentos ~]# 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

# 其他机制

- 心跳机制

- 解决粘包、拆包问题
- 注解开发等等

# RPC框架v1.0简易版

要求简单实现远程调用

用最笨的方法实现

## 技术选型

- 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;
```

```java
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（补丁）

## 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 可能出现的沾包拆包问题**


  - 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  但我们是要等他连接上
       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;
```

```java
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 {
        //先创建对应的额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));

        //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

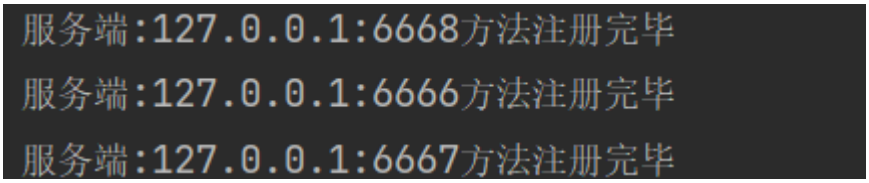
```
//v1.3更新 使用软负载
        //到对应节点中获取下面的子节点
        List<String> children = zooKeeper.getChildren(prePath, false, null);
        if (children.isEmpty())
        {
            System.out.println("当前没有服务器提供该服务 请联系工作人员");
        }

        //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，表名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).getByte
s(StandardCharsets.UTF_8),-1);
        String address = new String(children.get(0));
        return address;
    }
```

- 效果：成功实现软负载均衡




```
[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到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什么方法和启用的数量

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**
  - 

  - 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 这之前放着会有一定的问题
  - 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

```

```

```java
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;

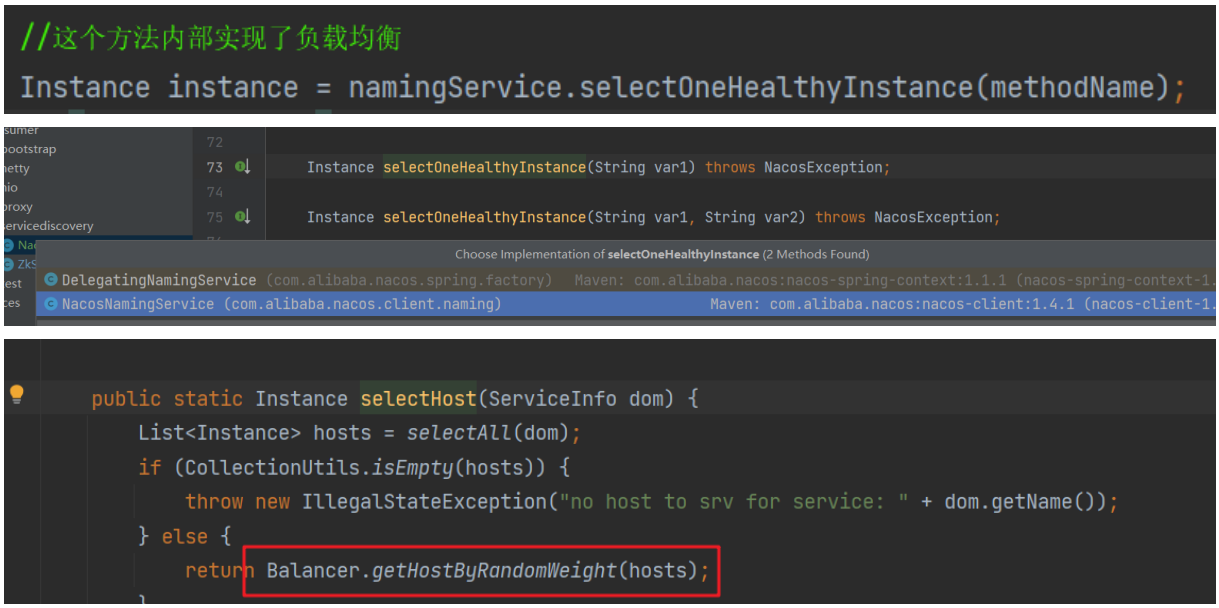
//直接实现启动类根据启动类接口上的注解选择对应需要选取的方法
public class MethodRegister implements NIOProviderBootStrap {
    /**
     * 实际进行注册的方法
     * @param method 方法名字
     * @param ip  对应的ip
     * @param port    对应的port
     */
    public static void register(String method, String ip, int port) throws
NacosException, RpcException,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




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(Standa
rdCharsets.UTF_8));
        client.close();
        System.out.println("服务端:"+hostname+":"+port+":"+RpcServiceName+"方法在
zkCurator中注册完毕");
    }
}
```

- 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zookeepper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

- 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)});
    }
}
```

**提供者启动类**

```
package provider.bootstrap.netty;

import provider.netty.NettyServer20;

/*
    以netty为网络编程框架的服务提供端启动类
 */
public class NettyProviderBootStrap20 {
    public static void main(String[] args) throws InterruptedException {
        //传入要绑定的ip和端口
        NettyServer20.start(args[0], Integer.parseInt(args[1]));
    }
}
```

**服务提供者实际启动类**

```
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 provider.netty_server_handler.NettyServerHandler01;


//简单实现  主要还是进行了一段回想
public class NettyServer20 {
    public static void start(String hostName,int port) throws InterruptedException {
        //该方法完成NettyServer的初始化  好好想想看 该怎么完成这个
        ServerBootstrap serverBootstrap = new ServerBootstrap();
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workGroup = new NioEventLoopGroup();
        try {
            serverBootstrap.group(bossGroup,workGroup)
                    .channel(NioServerSocketChannel.class) //自身所实现的通道
                    .option(ChannelOption.SO_BACKLOG,128) //设置线程队列得到的连接个数
                    .childOption(ChannelOption.SO_KEEPALIVE,true) //设置保持活动连接状态
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel socketChannel) throws
Exception {
                            //每个用户连接上都会进行初始化
```

```
                            System.out.println("客户socketChannel
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>
```

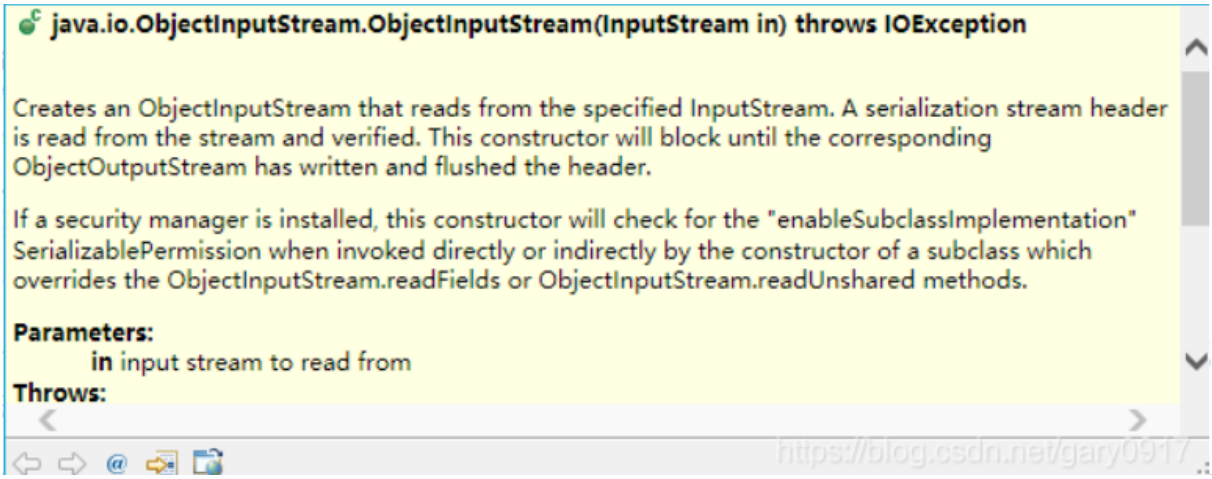
## 更新

# 遇到的问题

- @Data注解在类上无法使用 解决我把我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socker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 然后出问题了直接死锁动弹不得 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));
    }
}
```




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

## 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