2

第 2 篇

Netty 初体验

第 2 章　Netty 与 NIO 之前世今生

第 3 章　基于 Netty 重构 RPC 框架

3

第 3 章

基于 Netty 重构 RPC 框架

课程目标

1、深刻理解 RPC 框架设计的基本原理。

2、基于手写 Mini 版本的 RPC 框架。

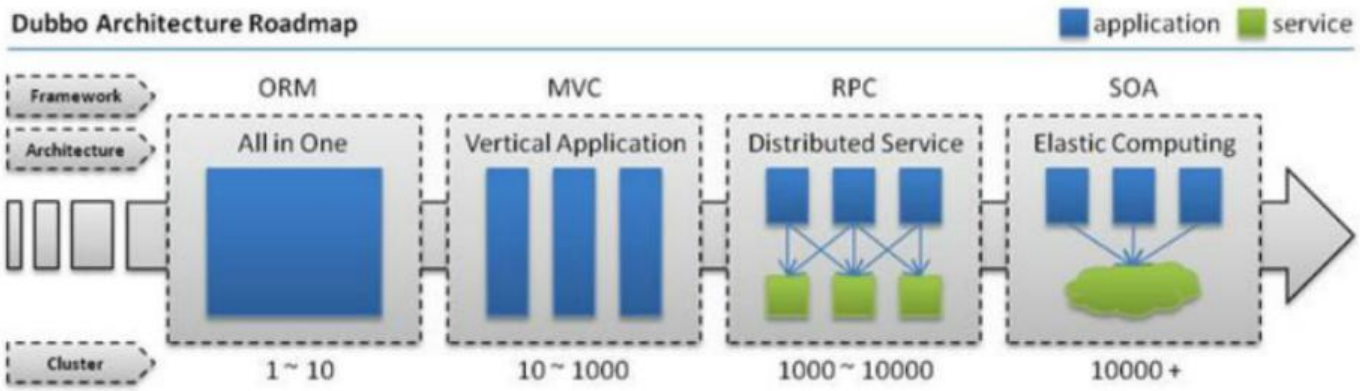
内容定位

1、对 Netty 感兴趣的人群。

2、有 RPC 框架使用经验的人群。

3.1 RPC 概述

下面的这张图，大概很多小伙伴都见到过，这是 Dubbo 官网中的一张图描述了项目架构的演进过程。



它描述了每一种架构需要的具体配置和组织形态。当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本,我们通常会采用单一应用架构。之后出现了 ORM 框架，主要用于简化增删改查工作流的，数据访问框架 ORM 是关键。

随着用户量增加，当访问量逐渐增大，单一应用增加机器，带来的加速度越来越小 ，我们需要将应用拆分成互不干扰的几个应用，以提升效率，于是就出现了垂直应用架构。MVC 架构就是一种非常经典的用于加速前端页面开发的架构。

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服逐渐形成稳定的服务中心，使前端应用能更快速的响应，多变的市场需求，就出现了分布式服务架构。分布式架构下服务数量逐渐增加，为了提高管理效率，RPC 框架应运而生。RPC 用于提高业务复用及整合的，分布式服务框架下 RPC 是关键。

下一代框架，将会是流动计算架构占据主流。当服务越来越多，容量的评估，小服务的资源浪费等问题，逐渐明显。此时，需要增加一个调度中心 ，基于访问压力实时管理集群容量，提高集群利用率。SOA 架构就是用于提高及其利用率的，资源调度和治理中心 SOA 是关键。

Netty 基本上是作为架构的技术底层而存在的，主要完成高性能的网络通信。

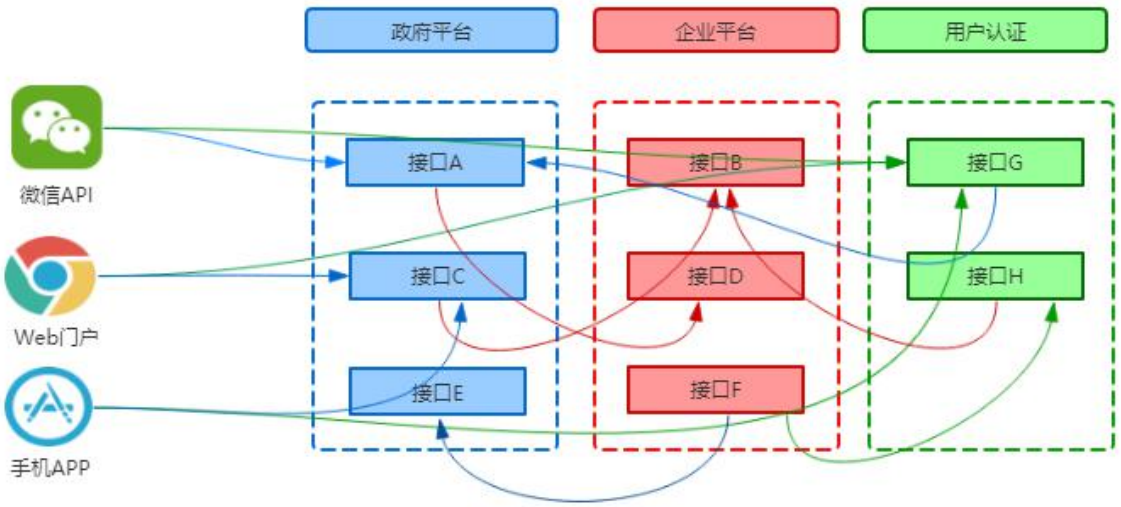
3.2 环境预设

第一步：我们先将项目环境搭建起来，创建 pom.xml 配置文件如下：

```
<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/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.gupaoedu</groupId>
    <artifactId>gupaoedu-vip-netty-rpc</artifactId>
    <version>1.0.0</version>

    <dependencies>

     <dependency>
       <groupId>io.netty</groupId>
       <artifactId>netty-all</artifactId>
       <version>4.1.6.Final</version>
     </dependency>

     <dependency>
         <groupId>org.projectlombok</groupId>
         <artifactId>lombok</artifactId>
         <version>1.16.10</version>
     </dependency>

    </dependencies>

</project>
```

第二步：创建项目结构。

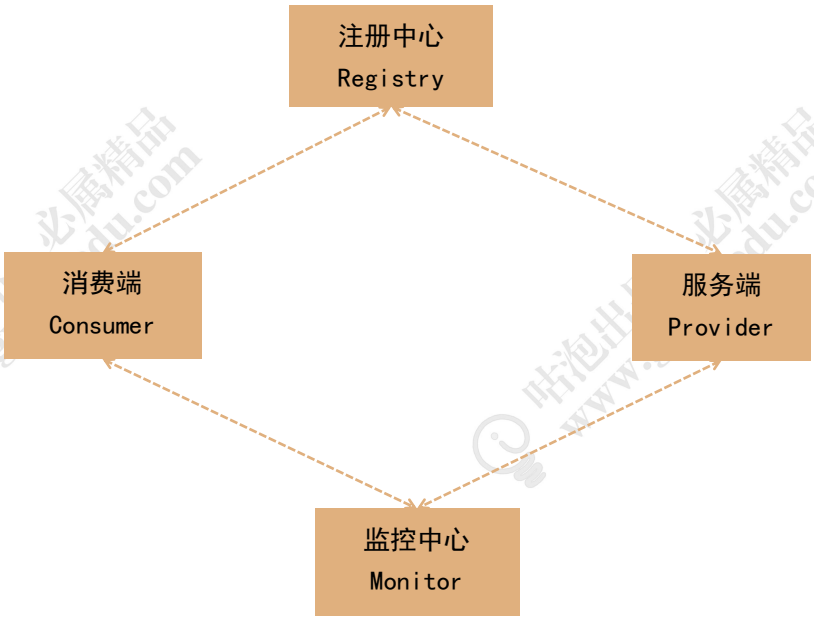
在没有 RPC 框架以前，我们的服务调用是这样的，如下图：



从上图可以看出接口的调用完全没有规律可循，想怎么调，就怎么调。这导致业务发展到一定阶段之后，对接口的维护变得非常困难。于是有人提出了服务治理的概念。所有服务间不允许直接调用，而是先到注册中心进行登记，再由注册中心统一协调和管理所有服务的状态并对外发布，调用者只需要记住服务名称，去找注册中心获取服务即可。

这样，极大地规范了服务的管理，可以提高了所有服务端可控性。整个设计思想其实在我们生活中也能找到活生生的案例。例如：我们平时工作交流，大多都是用 IM 工具，而不是面对面吼。大家只需要相互记住运营商（也就是注册中心）提供的号码（如：腾讯 QQ）即可。再比如：我们打电话，所有电话号码有运营商分配。我们需要和某一个人通话时，只需要拨通对方的号码，运营商（注册中心，如中国移动、中国联通、中国电信）就会帮我们将信号转接过去。

目前流行的 RPC 服务治理框架主要有 Dubbo 和 Spring Cloud，下面我们以比较经典的 Dubbo 为例。Dubbo 核心模块主要有四个：Registry 注册中心、Provider 服务端、Consumer 消费端、Monitor 监控中心，如下图所示：



为了方便，我们将所有模块全部放到一个项目中，主要模块包括：

api：主要用来定义对外开放的功能与服务接口。

protocol：主要定义自定义传输协议的内容。

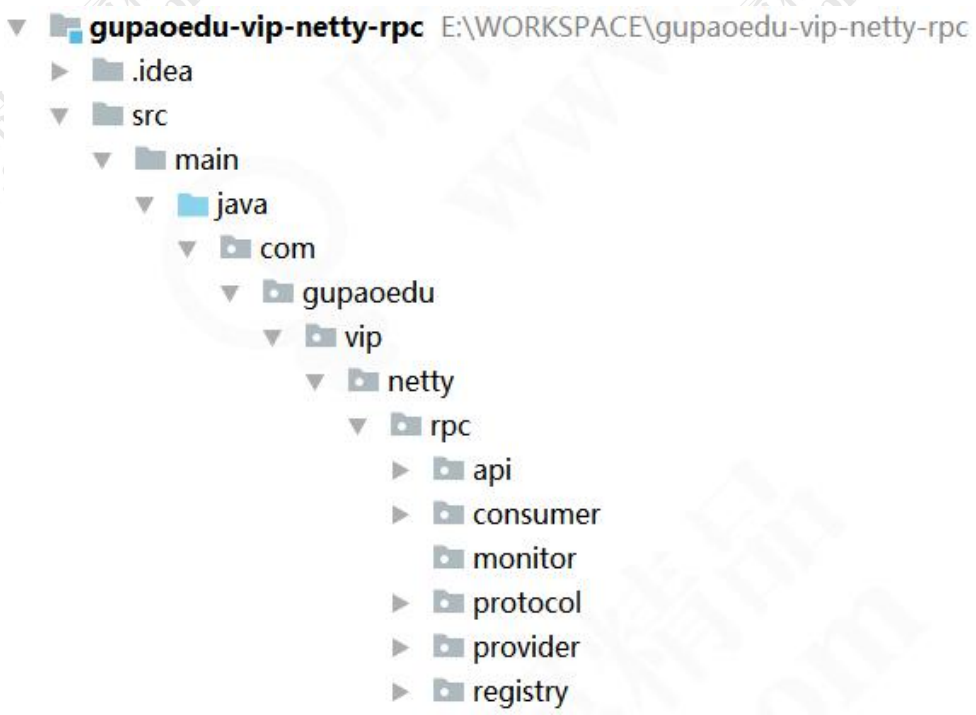
registry：主要负责保存所有可用的服务名称和服务地址。

provider：实现对外提供的所有服务的具体功能。

consumer：客户端调用。

monitor：完成调用链监控。

下面，我们先把项目结构搭建好，具体的项目结构截图如下：



3.3 代码实战

3.3.1 创建 API 模块

首先创建 API 模块，provider 和 consumer 都遵循 API 模块的规范。为了简化，创建两个 Service 接口，分别是：

IRpcHelloService 接口，实现一个 hello()方法，主要目的是用来确认服务是否可用，具体代码如下：

```
package com.gupaoedu.vip.netty.rpc.api;

public interface IRpcHelloService {
    String hello(String name);
}
```

创建 IRpcService 接口，完成模拟业务加、减、乘、除运算，具体代码如下：

```
package com.gupaoedu.vip.netty.rpc.api;

public interface IRpcService {

    /** 加 */
    public int add(int a,int b);

    /** 减 */
    public int sub(int a,int b);

    /** 乘 */
    public int mult(int a,int b);
```

```
    /** 除 */
    public int div(int a,int b);

}
```

至此，API 模块就定义完成了，非常简单。接下来，我们要确定传输规则，也就是传输协议，协议内容当然要自定义，才能体现出 Netty 的优势。

3.3.2 创建自定义协议

上一次课中，我们初步了解到 Netty 中内置的 HTTP 协议，需要 HTTP 的编、解码器来完成解析。今天，我们来看自定义协议如何设定？

在 Netty 中要完成一个自定义协议，其实非常简单，只需要定义一个普通的 Java 类即可。我们现在手写 RPC 主要是完成对 Java 代码的远程调用（类似于 RMI，大家应该都很熟悉了吧），远程调用 Java 代码哪些内容是必须由网络来传输的呢？譬如，服务名称？需要调用该服务的哪个方法？方法的实参是什么？这些信息都需要通过客户端传送到服务端去。

下面我们来看具体的代码实现，定义 InvokerProtocol 类：

```
package com.gupaoedu.vip.netty.rpc.protocol;

import lombok.Data;
import java.io.Serializable;

/**
 * 自定义传输协议
 */
@Data
public class InvokerProtocol implements Serializable {

    private String className;//类名
    private String methodName;//函数名称
    private Class<?>[] parames;//参数类型
    private Object[] values;//参数列表

}
```

从上面的代码看出来，协议中主要包含的信息有类名、函数名、形参列表和实参列表，通过这些信息就可以定位到一个具体的业务逻辑实现。

3.3.3 实现 Provider 服务端业务逻辑

我们将 API 中定义的所有功能在 provider 模块中实现，分别创建两个实现类：

RpcHelloServiceImpl 类：

```
package com.gupaoedu.vip.netty.rpc.provider;

import com.gupaoedu.vip.netty.rpc.api.IRpcHelloService;

public class RpcHelloServiceImpl implements IRpcHelloService {

    public String hello(String name) {
        return "Hello " + name + "!";
    }

}
```

RpcServiceImpl 类：

```
package com.gupaoedu.vip.netty.rpc.provider;

import com.gupaoedu.vip.netty.rpc.api.IRpcService;

public class RpcServiceImpl implements IRpcService {

    public int add(int a, int b) {
        return a + b;
    }

    public int sub(int a, int b) {
        return a - b;
    }

    public int mult(int a, int b) {
        return a * b;
    }

    public int div(int a, int b) {
        return a / b;
    }

}
```

3.3.4 完成 Registry 服务注册

Registry 注册中心主要功能就是负责将所有 Provider 的服务名称和服务引用地址注册到一个容器中，并对外发布。Registry 应该要启动一个对外的服务，很显然应该作为服务端，并提供一个对外可以访问的端口。先启动一个 Netty 服务，创建 RpcRegistry 类，具体代码如下：

```
package com.gupaoedu.vip.netty.rpc.registry;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
```

```
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
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.NioServerSocketChannel;
import io.netty.handler.codec.LengthFieldBasedFrameDecoder;
import io.netty.handler.codec.LengthFieldPrepender;
import io.netty.handler.codec.serialization.ClassResolvers;
import io.netty.handler.codec.serialization.ObjectDecoder;
import io.netty.handler.codec.serialization.ObjectEncoder;

public class RpcRegistry {
    private int port;
    public RpcRegistry(int port){
        this.port = port;
    }
    public void start(){
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
               .channel(NioServerSocketChannel.class)
                    .childHandler(new ChannelInitializer<SocketChannel>() {

                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline pipeline = ch.pipeline();
                            //自定义协议解码器
                            /** 入参有 5 个，分别解释如下
                             maxFrameLength：框架的最大长度。如果帧的长度大于此值，则将抛出 TooLongFrameException。
                             lengthFieldOffset：长度字段的偏移量：即对应的长度字段在整个消息数据中得位置
                             lengthFieldLength：长度字段的长度。如：长度字段是 int 型表示，那么这个值就是 4（long 型就是 8）
                             lengthAdjustment：要添加到长度字段值的补偿值
                             initialBytesToStrip：从解码帧中去除的第一个字节数
                             */
                            pipeline.addLast(new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 0, 4, 0, 4));
                            //自定义协议编码器
                            pipeline.addLast(new LengthFieldPrepender(4));
                            //对象参数类型编码器
                            pipeline.addLast("encoder",new ObjectEncoder());
                            //对象参数类型解码器
                            pipeline.addLast("decoder",new ObjectDecoder(Integer.MAX_VALUE,
                                                                    ClassResolvers.cacheDisabled(null)));
                            pipeline.addLast(new RegistryHandler());
                        }
                    })
                    .option(ChannelOption.SO_BACKLOG, 128)
                    .childOption(ChannelOption.SO_KEEPALIVE, true);
            ChannelFuture future = b.bind(port).sync();
            System.out.println("GP RPC Registry start listen at " + port );
            future.channel().closeFuture().sync();
        } catch (Exception e) {
             bossGroup.shutdownGracefully();
             workerGroup.shutdownGracefully();
        }
    }


    public static void main(String[] args) throws Exception {
```

```
        new RpcRegistry(8080).start();
    }
}
```

在 RegistryHandler 中实现注册的具体逻辑，上面的代码，主要实现服务注册和服务调用的功能。因为所有模块创建在同一个项目中，为了简化，服务端没有采用远程调用，而是直接扫描本地 Class，然后利用反射调用。代码实现如下：

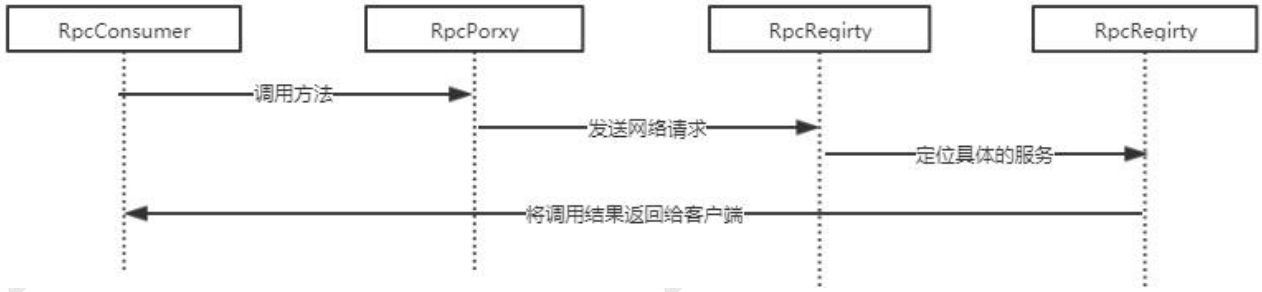
```
package com.gupaoedu.vip.netty.rpc.registry;

import java.io.File;
import java.lang.reflect.Method;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;

import com.gupaoedu.vip.netty.rpc.protocol.InvokerProtocol;

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

public class RegistryHandler  extends ChannelInboundHandlerAdapter {

    //用保存所有可用的服务
    public static ConcurrentHashMap<String, Object> registryMap = new ConcurrentHashMap<String,Object>();

    //保存所有相关的服务类
    private List<String> classNames = new ArrayList<String>();

    public RegistryHandler(){
     //完成递归扫描
     scannerClass("com.gupaoedu.vip.netty.rpc.provider");
     doRegister();
    }


    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
     Object result = new Object();
        InvokerProtocol request = (InvokerProtocol)msg;

        //当客户端建立连接时，需要从自定义协议中获取信息，拿到具体的服务和实参
     //使用反射调用
        if(registryMap.containsKey(request.getClassName())){
         Object clazz = registryMap.get(request.getClassName());
         Method method = clazz.getClass().getMethod(request.getMethodName(), request.getParames());
         result = method.invoke(clazz, request.getValues());
        }
        ctx.write(result);
        ctx.flush();
        ctx.close();
    }

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

```
    /*
     * 递归扫描
     */
    private void scannerClass(String packageName){
     URL url = this.getClass().getClassLoader().getResource(packageName.replaceAll("\\.", "/"));
     File dir = new File(url.getFile());
     for (File file : dir.listFiles()) {
         //如果是一个文件夹，继续递归
         if(file.isDirectory()){
          scannerClass(packageName + "." + file.getName());
         }else{
          classNames.add(packageName + "." + file.getName().replace(".class", "").trim());
         }
     }
    }

    /**
     * 完成注册
     */
    private void doRegister(){
     if(classNames.size() == 0){ return; }
     for (String className : classNames) {
         try {
          Class<?> clazz = Class.forName(className);
          Class<?> i = clazz.getInterfaces()[0];
          registryMap.put(i.getName(), clazz.newInstance());
         } catch (Exception e) {
          e.printStackTrace();
         }
     }
    }
}
```

至此，注册中心的基本功能就已完成，下面来看客户端的代码实现。

3.3.5 实现 Consumer 远程调用

梳理一下基本的实现思路，主要完成一个这样的功能：API 模块中的接口功能在服务端实现（并没有在客户端实现）。因此，客户端调用 API 中定义的某一个接口方法时，实际上是要发起一次网络请求去调用服务端的某一个服务。而这个网络请求首先被注册中心接收，由注册中心先确定需要调用的服务的位置，再将请求转发至真实的服务实现，最终调用服务端代码，将返回值通过网络传输给客户端。整个过程对于客户端而言是完全无感知的，就像调用本地方法一样。具体调用过程如下图所示：



下面来看代码实现，创建 RpcProxy 类：

```
package com.gupaoedu.vip.netty.rpc.consumer.proxy;

import java.lang.reflect.Proxy;

public class RpcProxy {

    public static <T> T create(Class<?> clazz){
      //clazz 传进来本身就是 interface
       MethodProxy proxy = new MethodProxy(clazz);
       Class<?> [] interfaces = clazz.isInterface() ?
                           new Class[]{clazz} :
                           clazz.getInterfaces();
       T result = (T) Proxy.newProxyInstance(clazz.getClassLoader(),interfaces,proxy);
       return result;
    }

}
```

在 RpcProxy 类的内部实现远程方法调用的代理类，即由 Netty 发送网络请求，具体代码如下：

```
package com.gupaoedu.vip.netty.rpc.consumer.proxy;

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

import com.gupaoedu.vip.netty.rpc.protocol.InvokerProtocol;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
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.LengthFieldBasedFrameDecoder;
import io.netty.handler.codec.LengthFieldPrepender;
import io.netty.handler.codec.serialization.ClassResolvers;
import io.netty.handler.codec.serialization.ObjectDecoder;
import io.netty.handler.codec.serialization.ObjectEncoder;

public class RpcProxy {

    public static <T> T create(Class<?> clazz){
       ...
```

```
}

private static class MethodProxy implements InvocationHandler {
 private Class<?> clazz;
 public MethodProxy(Class<?> clazz){
    this.clazz = clazz;
 }

 public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {
    //如果传进来是一个已实现的具体类（本次演示略过此逻辑)
    if (Object.class.equals(method.getDeclaringClass())) {
     try {
        return method.invoke(this, args);
     } catch (Throwable t) {
        t.printStackTrace();
     }
     //如果传进来的是一个接口（核心)
    } else {
     return rpcInvoke(proxy,method, args);
    }
    return null;
 }


 /**
  * 实现接口的核心方法
  * @param method
  * @param args
  * @return
  */
 public Object rpcInvoke(Object proxy,Method method,Object[] args){

    //传输协议封装
    InvokerProtocol msg = new InvokerProtocol();
    msg.setClassName(this.clazz.getName());
    msg.setMethodName(method.getName());
    msg.setValues(args);
    msg.setParames(method.getParameterTypes());

    final RpcProxyHandler consumerHandler = new RpcProxyHandler();
    EventLoopGroup group = new NioEventLoopGroup();
    try {
     Bootstrap b = new Bootstrap();
     b.group(group)
          .channel(NioSocketChannel.class)
          .option(ChannelOption.TCP_NODELAY, true)
          .handler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws Exception {
              ChannelPipeline pipeline = ch.pipeline();
              //自定义协议解码器
              /** 入参有 5 个，分别解释如下
               maxFrameLength：框架的最大长度。如果帧的长度大于此值，则将抛出 TooLongFrameException。
               lengthFieldOffset：长度字段的偏移量：即对应的长度字段在整个消息数据中得位置
               lengthFieldLength：长度字段的长度：如：长度字段是 int 型表示，那么这个值就是 4（long 型就是 8）
               lengthAdjustment：要添加到长度字段值的补偿值
               initialBytesToStrip：从解码帧中去除的第一个字节数
               */
              pipeline.addLast("frameDecoder", new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 0, 4, 0, 4));
              //自定义协议编码器
              pipeline.addLast("frameEncoder", new LengthFieldPrepender(4));
              //对象参数类型编码器
              pipeline.addLast("encoder", new ObjectEncoder());
```

```
                //对象参数类型解码器
                pipeline.addLast("decoder", new ObjectDecoder(Integer.MAX_VALUE,
                                                              ClassResolvers.cacheDisabled(null)));
                pipeline.addLast("handler",consumerHandler);
              }
            });

         ChannelFuture future = b.connect("localhost", 8080).sync();
         future.channel().writeAndFlush(msg).sync();
         future.channel().closeFuture().sync();
        } catch(Exception e){
         e.printStackTrace();
        }finally {
         group.shutdownGracefully();
        }
        return consumerHandler.getResponse();
     }

    }
}
```

接收网络调用的返回值

```
package com.gupaoedu.vip.netty.rpc.consumer.proxy;

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

public class RpcProxyHandler extends ChannelInboundHandlerAdapter {

    private Object response;

    public Object getResponse() {
        return response;
    }

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

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
        System.out.println("client exception is general");
    }
}
```

完成客户端调用代码：

```
package com.gupaoedu.vip.netty.rpc.consumer;

import com.gupaoedu.vip.netty.rpc.api.IRpcService;
import com.gupaoedu.vip.netty.rpc.api.IRpcHelloService;
import com.gupaoedu.vip.netty.rpc.consumer.proxy.*;

public class RpcConsumer {

    public static void main(String [] args){
        IRpcHelloService rpcHello = RpcProxy.create(IRpcHelloService.class);

        System.out.println(rpcHello.hello("Tom 老师"));
```

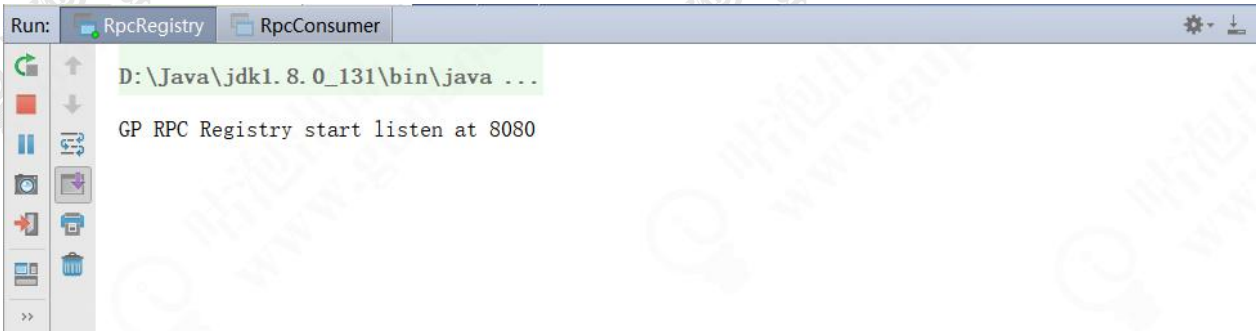
```
        IRpcService service = RpcProxy.create(IRpcService.class);

        System.out.println("8 + 2 = " + service.add(8, 2));
        System.out.println("8 - 2 = " + service.sub(8, 2));
        System.out.println("8 * 2 = " + service.mult(8, 2));
        System.out.println("8 / 2 = " + service.div(8, 2));
    }

}
```

3.3.6 Monitor 监控

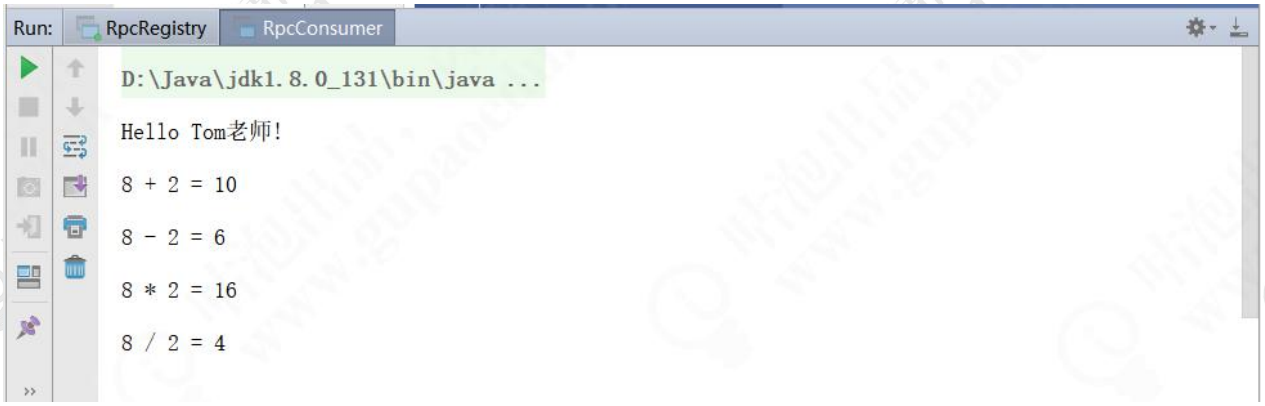
Dubbo 中的 Monitor 是用 Spring 的 AOP 埋点来实现的，我们没有引入 Spring 框架，在本节中不实现监控的功能。感兴趣的小伙伴，可以回顾之前 Spring AOP 的课程自行完善此功能。

3.4 运行效果演示

第一步，启动注册中心，运行结果如下：



第二步，运行客户端，调用结果如下：



通过以上案例演示，相信小伙伴们对 Netty 的应用已经有了一个比较深刻的印象。之后的课程中，我们继续深入

分析 Netty 的底层原理。本节课内容，只是对 RPC 的基本实现原理做了一个简单的实现，感兴趣的小伙伴可以在本项目的基础上继续完善 RPC 的其他细节。