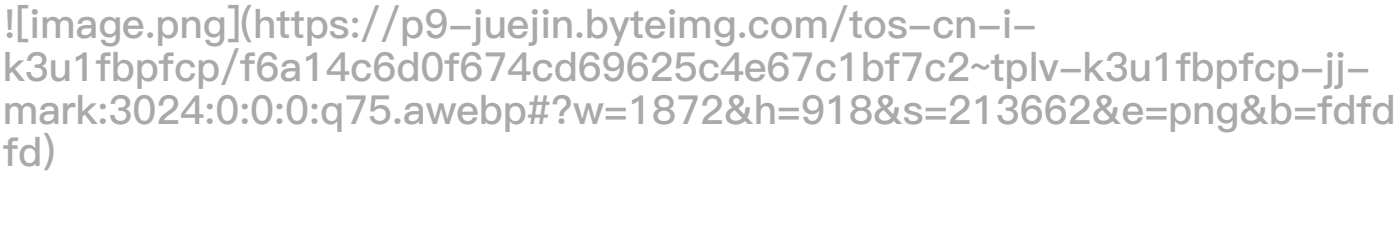
Please visit website: http://cxyroad.com

网络编程-5 （手写RPC框架）

1.1 RPC原理

RPC（Remote Procedure Call），即远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络实现的技术。

常见的RPC框架有：Dubbo(阿里)、Spring Cloud Feign（Spring）、gRPC（Google）等

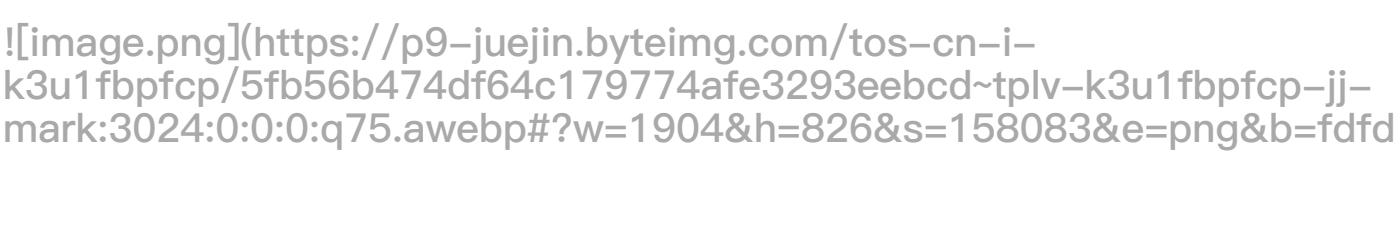


1. **服务消费方（Client）以本地调用方式调用服务**
2. client stub（可以用NIO、Netty实现）接受到调用后，负责将方法、参数等封装成能够进行网络传输的消息体
3. client stub将消息进行编码并发送到服务端
4. server stub收到消息进行解码
5. server stub根据解码效果调用**提供者**
6. 本地服务执行并将结果返回给server stub
7. server stub将返回导入结果进行编码并发送至**消费方**
8. client stub接受到消息并进行解码
9. **服务消费方（Client）得到结果**

RPC的目标就是将2-8步骤全部封装起来，用户无需关心这些细节，可以像调用本地方法一样即可完成远程服务调用。

接下来我们基于Netty手写一个RPC。

1.2 框架设计结构图



fd)

* 服务调用方：两个接口【服务的提供方决定】 + 一个包含main方法的测试类
* Client Stub：一个客户端代理类 + 一个客户端业务处理类（Netty + 动态代理）
+ HeroRPCProxy
+ ResultHandler
* 服务的提供方：两个接口 + 两个实现类
* Server Stub：一个网络处理服务器 + 一个服务器业务处理类 （Netty + 反射）
+ HeroRPCServer
+ InvokeHandler

注意：

* 服务的调用方的接口必须跟服务提供方的接口保持一致（包路径可以不一致）
* 最终要实现的目标是：在TestHeroRPC中远程调用SkuServerImpl或者UserServerImpl的方法

1.3 代码实现

1.3.1 Server服务的提供方

1. SkuServer接口与实现类

```
package org.example.producer;

public interface SkuServer {

    String findByName(String name);
}
```

```
package org.example.producer.impl;
```

```
import org.example.producer.SkuServer;

public class SkuServerImpl implements SkuServer {

    @Override
    public String findByName(String name) {
        return ”sku{} :” +name;
    }
}

```

2. UserService接口与实现类

```
package org.example.producer;

public interface UserServer {

    String findById();
}

```

```
package org.example.producer.impl;


import org.example.producer.UserServer;

public class UserServerImpl implements UserServer {

    @Override
    public String findById() {
        return ”user{id=1,username=aaron}”;
    }
}

```

上述代码作为服务的提供方，我们分别编写了两个接口和两个实现类，供消费方远程调用。

1.3.2 Server Stub部分

1. 传输的消息封装类: 作为实体类用来封装消费方发起远程调用时传给服务方的数据。

```
package org.example.producerStub;

import java.io.Serializable;

/**
 * 消息封装类
 */
public class ClassInfo implements Serializable {

    private static final long serialVersionUID = 1L;

    /**
     * 类名
     */
    private String className;
    /**
     * 方法名
     */
    private String methodName;
    /**
     * 参数类型
     */
    private Class<?>[] types;
    /**
     * 参数列表
     */
    private Object[] objects;

    public String getClassName() {
        return className;
    }

    public void setClassName(String className) {
        this.className = className;
    }

    public String getMethodName() {
        return methodName;
    }
```

```
    public void setMethodName(String methodName) {
        this.methodName = methodName;
    }

    public Class<?>[] getTypes() {
        return types;
    }

    public void setTypes(Class<?>[] types) {
        this.types = types;
    }

    public Object[] getObjects() {
        return objects;
    }

    public void setObjects(Object[] objects) {
        this.objects = objects;
    }
}
```

2. 服务端业务处理类：Handler,作为业务处理类，读取消费方发来的数据，并根据得到的数据进行本地调用，然后把结果返回给消费方。

```
package org.example.producerStub;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import org.reflections.Reflections;

import java.lang.reflect.Method;
import java.util.Set;

/**
 * 服务端Handler
 */
public class InvokeHandler extends ChannelInboundHandlerAdapter {

    private String getImplClassName(ClassInfo classInfo) throws
Exception {
        // 服务方接口和实现类所在的路径
        String interfacePath = "org.example.producer";
        int lastDot = classInfo.getClassName().lastIndexOf(".");
```

```
        // 接口名称
        String interfaceName =
classInfo.getClassName().substring(lastDot);
        // 接口字节码对象
        Class superClass = Class.forName(interfacePath + interfaceName);
        // 反射得到某接口下的所有实现类
        Reflections reflections = new Reflections(interfacePath);
        Set<Class> implClassSet = reflections.getSubTypesOf(superClass);

        if (implClassSet.size() == 0) {
            System.out.println("未找到实现类");
            return null;
        } else if (implClassSet.size() > 1) {
            System.out.println("找到多个实现类，未明确使用哪一个");
            return null;
        } else {
            Class[] classes = implClassSet.toArray(new Class[0]);
            // 得到实现类的名字
            return classes[0].getName();
        }
    }

    /**
     * 读取客户端发过来的数据，并且通过反射调用实现类的方法
     * @param ctx
     * @param msg
     * @throws Exception
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {
        ClassInfo classInfo = (ClassInfo) msg;
        Object clazz =
Class.forName(getImplClassName(classInfo)).newInstance();
        Method method =
clazz.getClass().getMethod(classInfo.getMethodName(),
classInfo.getTypes());
        // 通过反射调用实现类的方法
        Object result = method.invoke(clazz, classInfo.getObjects());
        ctx.writeAndFlush(result);
    }
}

```

3. RPC服务端程序：HeroRPCServer,是用 Netty 实现的网络服务器，采用 Netty 自带的 ObjectEncoder 和 ObjectDecoder作为编解码器（为了降低复杂度，这里并没有使用第三方的编解码器），当然实际开发时也可以采用

JSON 或XML。

```
package org.example.producerStub;

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.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.serialization.ClassResolvers  ;
import io.netty.handler.codec.serialization.ObjectDecoder;
import io.netty.handler.codec.serialization.ObjectEncoder;

public class HeroRpcServer {

    private int port;

    public HeroRpcServer(int port) {
        this.port = port;
    }

    public void start() {
        NioEventLoopGroup bossGroup = new NioEventLoopGroup();
        NioEventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .option(ChannelOption.SO_BACKLOG, 128)
                    .childOption(ChannelOption.SO_KEEPALIVE, true)
                    .localAddress(port)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws
Exception {
                            ChannelPipeline pipeline = ch.pipeline();
                            pipeline.addLast("encoder", new ObjectEncoder());
                            pipeline.addLast("decoder", new
ObjectDecoder(Integer.MAX_VALUE,
ClassResolvers.cacheDisabled(null)));
                            pipeline.addLast(new InvokeHandler());
                        }
```

```
                    });
            ChannelFuture channelFuture =
serverBootstrap.bind(port).sync();
            System.out.println("......Hero RPC is ready......");
            channelFuture.channel().closeFuture().sync();
        } catch (Exception e) {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }

    public static void main(String[] args) {
        new HeroRpcServer(9999).start();
    }
}

```

1.3.3 Client Stub部分

1. 客户端业务处理类：ResultHandler,作为客户端的业务处理类读取远程调用返回的数据

```
package org.example.consumerStub;

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

public class ResultHandler extends ChannelInboundHandlerAdapter {

    private Object response;

    public Object getResponse() {
        return response;
    }

    /**
     * 读取服务器端返回来的数据（远程调用的结果）
     * @param ctx
     * @param msg
     * @throws Exception
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
```

```
throws Exception {
        response = msg;
        ctx.close();
    }
}
```

2. RPC客户端程序：RPC远程代理HeroRPCProxy

```
package org.example.consumerStub;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.serialization.ClassResolvers;
import io.netty.handler.codec.serialization.ObjectDecoder;
import io.netty.handler.codec.serialization.ObjectEncoder;
import org.example.producerStub.ClassInfo;

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

/**
 * 客户端代理类
 */
public class HeroRpcProxy {

    public static Object create (Class target) {
        return Proxy.newProxyInstance(target.getClassLoader(), new
Class[]{target}, new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                // 封装ClassInfo
                ClassInfo classInfo = new ClassInfo();
                classInfo.setClassName(target.getName());
                classInfo.setMethodName(method.getName());
                classInfo.setObjects(args);
                classInfo.setTypes(method.getParameterTypes());
```

```
                // 开始使用netty发送信息
                NioEventLoopGroup group = new NioEventLoopGroup();
                ResultHandler resultHandler = new ResultHandler();
                try {
                    Bootstrap bootstrap = new Bootstrap();
                    bootstrap.group(group)
                            .channel(NioSocketChannel.class)
                            .handler(new ChannelInitializer<SocketChannel>() {
                                @Override
                                protected void initChannel(SocketChannel ch) {
                                    ChannelPipeline pipeline = ch.pipeline();
                                    pipeline.addLast("encoder", new
ObjectEncoder());
                                    pipeline.addLast("decoder", new
ObjectDecoder(Integer.MAX_VALUE,
ClassResolvers.cacheDisabled(null)));
                                    pipeline.addLast("handler", resultHandler);
                                }
                            });
                    ChannelFuture channelFuture =
bootstrap.connect("127.0.0.1", 9999).sync();
                    channelFuture.channel().writeAndFlush(classInfo).sync();
                    channelFuture.channel().closeFuture().sync();
                } finally {
                    group.shutdownGracefully();
                }
                return resultHandler.getResponse();
            }
        });
    }
}

```

1.3.4 Client服务的调用方-消费方

```
package org.example.consumer;

import org.example.consumerStub.HeroRpcProxy;
import org.example.producer.SkuServer;
import org.example.producer.UserServer;

/**
 * 服务调用方
```

```
 */
public class TestHeroRpc {
    public static void main(String[] args) {
        SkuServer skuServer = (SkuServer)
HeroRpcProxy.create(SkuServer.class);
        System.out.println(skuServer.findByName("uid"));

        UserServer userServer = (UserServer)
HeroRpcProxy.create(UserServer.class);
        System.out.println(userServer.findById());
    }
}

```

先启动HeroRpcServer服务，再调用TestHeroRpc的main方法，返回远程调用结果。

原文链接: https://juejin.cn/post/7353280369382309914