



链滴

基于 netty 的 rpc 框架

作者: [losemy](#)

原文链接: <https://ld246.com/article/1577257809105>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

自主实现

==[simplerpc](#)==

实现思路

netty消息定义

请求参数

```
@Getter
@Setter
public class RpcRequest {
    /**
     * 唯一请求ID
     */
    private String requestId;
    /**
     * 接口名称
     */
    private String interfaceName;
    /**
     * 接口版本
     */
    private String serviceVersion;
    /**
     * 方法名称
     */
    private String methodName;
    /**
     * 参数类型
     */
    private Class<?>[] parameterTypes;
    /**
     * 参数
     */
    private Object[] parameters;
}
```

响应参数

```
@Getter
@Setter
public class RpcResponse {
    /**
     * 请求id
     */
    private String requestId;
    /**
     * 异常
     */
}
```

```

private Exception exception;
/**
 * 返回数据
 */
private Object result;
}

```

消息解析

解码器

```

public class RpcDecoder extends ByteToMessageDecoder {

    private Class<?> genericClass;

    public RpcDecoder(Class<?> genericClass) {
        this.genericClass = genericClass;
    }

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
        if (in.readableBytes() < 4) {
            return;
        }
        in.markReaderIndex();
        int dataLength = in.readInt();
        if (in.readableBytes() < dataLength) {
            in.resetReaderIndex();
            return;
        }
        byte[] data = new byte[dataLength];
        in.readBytes(data);
        out.add(SerializationUtil.deserialize(data, genericClass));
    }
}

```

编码器

```

public class RpcEncoder extends MessageToByteEncoder {

    private Class<?> genericClass;

    public RpcEncoder(Class<?> genericClass) {
        this.genericClass = genericClass;
    }

    @Override
    public void encode(ChannelHandlerContext ctx, Object in, ByteBuf out) throws Exception {
        if (genericClass.isInstance(in)) {
            byte[] data = SerializationUtil.serialize(in);
            out.writeInt(data.length);
            out.writeBytes(data);
        }
    }
}

```

```
}  
}
```

服务端实现

注解, 用来声明服务

```
@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Component  
public @interface RpcService {  
  
    /**  
     * 服务接口类  
     */  
    Class<?> value();  
  
    /**  
     * 服务版本号  
     */  
    String version() default "";  
}
```

handler处理通讯, 然后反射调用实际对应的bean

```
@Slf4j  
public class RpcServerHandler extends SimpleChannelInboundHandler<RpcRequest> {  
  
    private static final Logger LOGGER = LoggerFactory.getLogger(RpcServerHandler.class);  
  
    private final Map<String, Object> handlerMap;  
  
    public RpcServerHandler(Map<String, Object> handlerMap) {  
        this.handlerMap = handlerMap;  
    }  
  
    @Override  
    public void channelRead0(final ChannelHandlerContext ctx, RpcRequest request) throws Exception {  
        // 创建并初始化 RPC 响应对象  
        log.info("requestId" + request.getRequestId());  
        RpcResponse response = new RpcResponse();  
        response.setRequestId(request.getRequestId());  
        try {  
            Object result = handle(request);  
            response.setResult(result);  
        } catch (Exception e) {  
            LOGGER.error("handle result failure", e);  
            response.setException(e);  
        }  
        ctx.writeAndFlush(response);  
    }  
  
    private Object handle(RpcRequest request) throws Exception {
```

```

// 获取服务对象
String serviceName = request.getInterfaceName();
String serviceVersion = request.getServiceVersion();
if (StringUtil.isEmpty(serviceVersion)) {
    serviceName += "-" + serviceVersion;
}
Object serviceBean = handlerMap.get(serviceName);
if (serviceBean == null) {
    throw new RuntimeException(String.format("can not find service bean by key: %s", serviceName));
}
// 获取反射调用所需的参数
Class<?> serviceClass = serviceBean.getClass();
String methodName = request.getMethodName();
Class<?>[] parameterTypes = request.getParameterTypes();
Object[] parameters = request.getParameters();

FastClass serviceFastClass = FastClass.create(serviceClass);
FastMethod serviceFastMethod = serviceFastClass.getMethod(methodName, parameterTypes);
return serviceFastMethod.invoke(serviceBean, parameters);
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
    LOGGER.error("server caught exception", cause);
    ctx.close();
}
}

```

服务端实现，主要获取serviceBean集合，启动netty服务,其中需要将服务注册到zk上，方便客户端消费

```

@Slf4j
public class RpcServer implements ApplicationContextAware, InitializingBean, DisposableBean {

    private static final Logger LOGGER = LoggerFactory.getLogger(RpcServer.class);

    private final EventLoopGroup bossGroup = new NioEventLoopGroup();
    private final EventLoopGroup workerGroup = new NioEventLoopGroup();
    private String serviceAddress;

    private ServiceRegistry serviceRegistry;

    /**
     * 存放 服务名 与 服务对象 之间的映射关系
     */
    private Map<String, Object> handlerMap = new HashMap<>();

    public RpcServer(String serviceAddress) {
        this.serviceAddress = serviceAddress;
    }

    public RpcServer(String serviceAddress, ServiceRegistry serviceRegistry) {

```

```

    this.serviceAddress = serviceAddress;
    this.serviceRegistry = serviceRegistry;
}

@Override
public void setApplicationContext(ApplicationContext ctx) throws BeansException {
    // 扫描带有 RpcService 注解的类并初始化 handlerMap 对象
    Map<String, Object> serviceBeanMap = ctx.getBeansWithAnnotation(RpcService.class);
    if (CollUtil.isNotEmpty(serviceBeanMap)) {
        for (Object serviceBean : serviceBeanMap.values()) {
            RpcService rpcService = serviceBean.getClass().getAnnotation(RpcService.class);
            String serviceName = rpcService.value().getName();
            String serviceVersion = rpcService.version();
            if (StringUtil.isNotEmpty(serviceVersion)) {
                serviceName += "-" + serviceVersion;
            }
            handlerMap.put(serviceName, serviceBean);
        }
    }
    log.info("handlerMap " + handlerMap.size());
}

@Override
public void afterPropertiesSet() throws Exception {
    try {
        // 创建并初始化 Netty 服务端 Bootstrap 对象
        ServerBootstrap bootstrap = new ServerBootstrap();
        bootstrap.group(bossGroup, workerGroup);
        bootstrap.channel(NioServerSocketChannel.class);
        bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel channel) throws Exception {
                ChannelPipeline pipeline = channel.pipeline();
                //pipeline.addLast(new LengthFieldBasedFrameDecoder(65536, 0, 4, 0, 0));
                pipeline.addLast(new RpcDecoder(RpcRequest.class)); // 解码 RPC 请求
                pipeline.addLast(new RpcEncoder(RpcResponse.class)); // 编码 RPC 响应
                pipeline.addLast(new RpcServerHandler(handlerMap)); // 处理 RPC 请求
            }
        });
        bootstrap.option(ChannelOption.SO_BACKLOG, 128);
        bootstrap.childOption(ChannelOption.SO_KEEPALIVE, true);
        // 获取 RPC 服务器的 IP 地址与端口号
        String[] addressArray = StringUtil.split(serviceAddress, ":");
        String ip = addressArray[0];
        int port = Integer.parseInt(addressArray[1]);
        // 启动 RPC 服务器
        ChannelFuture future = bootstrap.bind(ip, port).sync();
        // 注册 RPC 服务地址
        if (serviceRegistry != null) {
            for (String interfaceName : handlerMap.keySet()) {
                log.info(serviceAddress);
                serviceRegistry.register(interfaceName, serviceAddress);
                LOGGER.debug("register service: {} => {}", interfaceName, serviceAddress);
            }
        }
    }
}

```

```

    }
    LOGGER.debug("server started on port {}", port);
    // 关闭 RPC 服务器
    future.channel().closeFuture().sync();
} catch (Exception e){
    LOGGER.error("开启服务异常",e);
}
}

@Override
public void destroy() throws Exception {
    workerGroup.shutdownGracefully();
    bossGroup.shutdownGracefully();
}
}

```

客户端实现

注解，声明哪里需要代理

```

@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Component
public @interface RpcReference {

    /**
     * 服务版本号
     */
    String version() default "";
}

```

网络通讯处理，通过requestId，来区分或者接受返回，其中使用CompletableFuture 实现异步

```

@Slf4j
public class RpcClientHandler extends SimpleChannelInboundHandler<RpcResponse> {

    private ConcurrentHashMap<String, CompletableFuture<RpcResponse>> responses = new
    ConcurrentHashMap<>();

    private volatile Channel channel;

    public Channel getChannel() {
        return channel;
    }

    @Override
    public void channelRegistered(ChannelHandlerContext ctx) throws Exception {
        super.channelRegistered(ctx);
        this.channel = ctx.channel();
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx, RpcResponse response) throws Exc
    ption {

```

```

String requestId = response.getRequestId();
log.info("requestId {}", requestId);
//异步处理消息返回，有消息就更新 然后get才能够获取到
CompletableFuture<RpcResponse> rpcFuture = responses.get(requestId);
if (rpcFuture != null) {
    responses.remove(requestId);
    rpcFuture.complete(response);
    log.info("read done {}" , rpcFuture.isDone());
}else{
    log.error("没有对应的CompletableFuture");
}
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
    log.error("api caught exception", cause);
    ctx.close();
}

public CompletableFuture<RpcResponse> sendRequest(RpcRequest rpcRequest){
    try {
        log.info("sendRequest {}", JSON.toJSONString(rpcRequest));
        CompletableFuture<RpcResponse> responseFuture = new CompletableFuture<>();
        String name = rpcRequest.getInterfaceName() + "-" + rpcRequest.getServiceVersion();
        responses.put(rpcRequest.getRequestId(), responseFuture);
        //发送请求
        channel.writeAndFlush(rpcRequest).addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) {
                log.info("发送请求 " + rpcRequest.getRequestId());
            }
        });
        log.info("isActive {}", channel.isActive());
        return responseFuture;
    }catch(Exception e){
        log.info(e.getMessage());
    }
    return null;
}
}

```

缓存handler（其本质是channel），在代理的时候获取并发起请求（sendRequest）

```

@Slf4j
public class ConnectManager {

    private static volatile ConnectManager connectManager;

    /**
     * key 为 interfaceName + version
     * value 为对应服务的channel
     */
    private Map<String, List<RpcClientHandler>> rpcClientHandlerMap = new ConcurrentHas

```



```

Map<>();

/**
 * key 为 interfaceName
 * value 为对应服务的list
 */
private Map<String,List<String>> rpcServerMap = new ConcurrentHashMap<>();

public static ConnectManager getInstance() {
    if (connectManager == null) {
        synchronized (ConnectManager.class) {
            if (connectManager == null) {
                connectManager = new ConnectManager();
            }
        }
    }
    return connectManager;
}

public void addServer(String name,String server){
    List<String> servers = rpcServerMap.get(name);
    if(servers == null){
        servers = new ArrayList<>();
    }
    servers.add(server);
    log.info("servers {} , size {}", name ,servers.size());
    rpcServerMap.put(name,servers);
}

public List<String> getServers(String name){
    return rpcServerMap.get(name);
}

public void addRpcClientHandler(RpcClientHandler rpcClientHandler,String name){
    List<RpcClientHandler> handlers = rpcClientHandlerMap.get(name);
    if(handlers == null){
        handlers = new ArrayList<>();
    }
    handlers.add(rpcClientHandler);
    log.info("handlers {} , size {}", name ,handlers.size());
    rpcClientHandlerMap.put(name,handlers);
}

public RpcClientHandler getRpcClientHandler(String name){
    log.info("getRpcClientHandler name {}", name);
    RpcClientHandler handler = null;
    List<RpcClientHandler> handlers = rpcClientHandlerMap.get(name);
    if(handlers.size() > 0 ){
        handler = handlers.get(RandomUtil.randomInt(handlers.size()));
    }
    log.info("handler== {}", handler);
    return handler;
}

```

```
}
```

代理工厂，替换实际调用，无感知使用netty，sendRequest获取CompletableFuture，然后使用get()获取返回

需要注意的是如果想要异步返回，目前是没有实现的。

可行方案：

1. 接口返回声明为 CompletableFuture <RpcResponse> 然后在此加判断返回为CompletableFuture类型，直接返回！

问题：是否可以直接返回 CompletableFuture<具体对象>

2. 放在threadLocal中？是否可行

```
@Slf4j
```

```
public class RpcProxy {
    private static final Logger LOGGER = LoggerFactory.getLogger(RpcProxy.class);

    private String serviceAddress;

    private ServiceDiscovery serviceDiscovery;

    public RpcProxy(String serviceAddress) {
        this.serviceAddress = serviceAddress;
    }

    public RpcProxy(ServiceDiscovery serviceDiscovery) {
        this.serviceDiscovery = serviceDiscovery;
    }

    @SuppressWarnings("unchecked")
    public <T> T create(final Class<?> interfaceClass) {
        return create(interfaceClass, "");
    }

    @SuppressWarnings("unchecked")
    public <T> T create(final Class<?> interfaceClass, final String serviceVersion) {
        // 创建动态代理对象
        return (T) Proxy.newProxyInstance(
            interfaceClass.getClassLoader(),
            new Class<?>[]{interfaceClass},
            new InvocationHandler() {
                @Override
                public Object invoke(Object proxy, Method method, Object[] args) throws Thro

ble {
        // 创建 RPC 请求对象并设置请求属性
        RpcRequest request = new RpcRequest();
        request.setRequestId(UUID.randomUUID().toString());
        request.setInterfaceName(method.getDeclaringClass().getName());
        request.setServiceVersion(serviceVersion);
        request.setMethodName(method.getName());
        request.setParameterTypes(method.getParameterTypes());
        request.setParameters(args);
    }
}
```

```

        // 创建 RPC 客户端对象并发送 RPC 请求
        String name = ServiceUtil.buildServiceName(request.getInterfaceName(), request.getServiceVersion());

        log.info("invoke {}" , name);
        RpcClientHandler handler = ConnectManager.getInstance().getRpcClientHandler(name);

        log.info("handler {}" , handler != null );
        CompletableFuture<RpcResponse> rpcResponse = handler.sendRequest(request);

        RpcResponse response = rpcResponse.get();
        if(response.getException() == null) {
            return response.getResult();
        }else{
            throw response.getException();
        }
    }
}
);
}
}

```

根据注解，注入代理bean

```

@Slf4j
public class ClientFactory extends InstantiationAwareBeanPostProcessorAdapter implements ApplicationAware {

    private ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }

    @Override
    public boolean postProcessAfterInstantiation(final Object bean, final String beanName) throws BeansException {

        log.info("test inject");
        ReflectionUtils.doWithFields(bean.getClass(), new ReflectionUtils.FieldCallback() {
            @Override
            public void doWith(Field field) throws IllegalArgumentException, IllegalAccessException {
                n {
                    if (field.isAnnotationPresent(RpcReference.class)) {
                        // valid
                        Class interfaceName = field.getType();
                        if (!interfaceName.isInterface()) {
                            log.error("声明类不是接口异常");
                        }
                    }
                }
            }
        });
    }
}

```

```

RpcReference rpcReference = field.getAnnotation(RpcReference.class);
String version = rpcReference.version();
String inName = interfaceName.getName();
String name = ServiceUtil.buildServiceName(inName,version);
//代理

log.info("interfaceName {}", inName);

//获取数据
RpcProxy proxy = applicationContext.getBean(RpcProxy.class);
Object serviceProxy = proxy.create(interfaceName,version);

ServiceDiscovery serviceDiscovery = applicationContext.getBean(ServiceDiscovery
class);

List<String> addressList = null;
// 获取 RPC 服务地址
if (serviceDiscovery != null) {
    String serviceName = interfaceName.getName();
    if (StrUtil.isNotEmpty(version)) {
        serviceName += "-" + version;
    }
    addressList = serviceDiscovery.discover(serviceName);
    log.info("serviceName {} addressList {}",name,addressList);
}
if (CollectionUtil.isEmpty(addressList)) {
    log.error("server address is empty");
    throw new RuntimeException("server address is empty");
}
// 从 RPC 服务地址中解析主机名与端口号
for(String serviceAddress:addressList) {
    log.info("serviceAddress {}",serviceAddress);
    String[] array = StrUtil.split(serviceAddress, ":");
    String host = array[0];
    int port = Integer.parseInt(array[1]);

    ConnectManager connectManager = ConnectManager.getInstance();
    List<String> servers = connectManager.getServers(name);
    if(CollectionUtil.contains(servers,serviceAddress)){
        //已开启对应服务
        continue;
    }
    connectManager.addServer(name,serviceAddress);
    //需要保证 host + port唯一 避免启动多个客户端连接同一个服务端
    RpcClient client = new RpcClient(host, port, name);
    //启动连接
    client.start();
}

// set bean
field.setAccessible(true);
field.set(bean, serviceProxy);

```

```

        }
    }
});

return super.postProcessAfterInstantiation(bean, beanName);
}
}

```

启动连接，也就是rpcClient

```

@Slf4j
public class RpcClient {

    private final String host;
    private final int port;
    private final String name;

    public RpcClient(String host, int port, String name) {
        this.host = host;
        this.port = port;
        this.name = name;
    }

    public void start(){

        EventLoopGroup group = new NioEventLoopGroup();
        try {
            // 创建并初始化 Netty 客户端 Bootstrap 对象
            Bootstrap bootstrap = new Bootstrap();
            RpcClientHandler rpcClientHandler = new RpcClientHandler();
            bootstrap.group(group);
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel channel) throws Exception {
                    ChannelPipeline pipeline = channel.pipeline();
                    //pipeline.addLast(new LengthFieldBasedFrameDecoder(65536, 0, 4, 0, 0));
                    pipeline.addLast(new RpcEncoder(RpcRequest.class)); // 编码 RPC 请求
                    pipeline.addLast(new RpcDecoder(RpcResponse.class)); // 解码 RPC 响应
                    pipeline.addLast(rpcClientHandler); // 处理 RPC 响应
                }
            });
            bootstrap.option(ChannelOption.TCP_NODELAY, true);

            log.info("host{}:port{}", host , port);
            ChannelFuture future = bootstrap.connect(host, port).sync();

            //创建的连接之后需要共享

            Channel channel = future.channel();

            future.addListener(new ChannelFutureListener() {
                @Override

```


服务端

```
bossGroup = new NioEventLoopGroup();
workerGroup = new NioEventLoopGroup();
ServerBootstrap bootstrap = new ServerBootstrap();
bootstrap.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
    .childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel channel) throws Exception {
            channel.pipeline()
                .addLast(new LengthFieldBasedFrameDecoder(65536, 0, 4, 0, 0))
                .addLast(new RpcDecoder(RpcRequest.class))
                .addLast(new RpcEncoder(RpcResponse.class))
                .addLast(new RpcHandler(handlerMap));
        }
    })
    .option(ChannelOption.SO_BACKLOG, 128)
    .childOption(ChannelOption.SO_KEEPALIVE, true);

String[] array = serverAddress.split(":");
String host = array[0];
int port = Integer.parseInt(array[1]);

ChannelFuture future = bootstrap.bind(host, port).sync();
logger.info("Server started on port {}", port);

if (serviceRegistry != null) {
    serviceRegistry.register(serverAddress);
}

future.channel().closeFuture().sync();
```