

电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

分布式系统



课题名称 YA-RPC: Yet-Another RPC Framework

学 院 计算机科学与工程学院

学 号 202222080626

作者姓名 王乐卿

指导教师 李玉军、侯孟书

目 录

第一章 需求分析与可行性研究	1
1.1 目标概述	1
1.2 可行性研究与分析	1
1.3 需求分析	1
第二章 概要设计	3
2.1 系统架构	3
2.2 系统流程图	4
2.3 代码架构	5
第三章 详细设计及实现	7
3.1 SPI 扩展机制	7
3.2 自定义协议	8
3.3 注册中心	9
3.4 远程调用	10
第四章 系统测试	13
4.1 系统测试概述	13
4.2 系统测试环境	13
4.3 系统测试用例及结果	15
第五章 开发计划	21

第一章 需求分析与可行性研究

1.1 目标概述

分布式计算中，远程过程调用（RPC, Remote Procedure Call）是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一个地址空间的子程序，而程序员就像调用本地程序一样，无需额外地为这个交互作用编程。

本项目的主要任务是实现一个简易的 RPC 框架，称为 YA-RPC，用户通过 Maven 导入 YA-RPC 框架依赖，即可快速的构建客户端和服务端。

通过 SPI 机制保证框架的后续扩展，使用注册中心集群保证服务端横向的快速扩展，提供多种序列化和压缩方式，实现轻量级的跨平台和跨语言调用，通过自定义协议保证 RPC 调用的高可用。

1.2 可行性研究与分析

针对本项目的课题目标，主要从技术实现、经济成本、社会因素等几个方面来分析本课题的可行性：

（一）技术实现方面：本人具有专业的计算机科学基础，在项目开发过程中遇到难题时，有能力开发文档解决开发问题。此外，笔者在字节跳动实习过程中有过大型中台项目开发经验，对项目开发的技术有一定的掌握，因此技术方面是可行的。

（二）经济成本方面：本项目采用的 Zookeeper、Spring、Protostuff、Hessian 等技术框架都属于开源项目，因此本项目开发的经济成本主要来源于服务器的开销，在短期内是可控的。

（三）社会因素方面：本项目并不提供实际业务，而只是给出 RPC 的解决方案，因此不涉及非法使用等侵权问题；此外，本项目的研究逻辑和代码实现都由本项目组亲自实现，不存在抄袭等不符合道德和法律的社会问题。

综上所述，本次项目在技术实现、经济成本和社会因素方面都是可行的。

1.3 需求分析

本项目通过快速原型法进行开发，在充分调研了市面上各种成熟 RPC 框架，例如 dubbo 框架，对项目进行简单的需求分析并进行用例图设计。

YA-RPC 框架用于帮助用户快速构建 RPC 调用的客户端和服务端，其基本用

例图如图 1-1 所示：

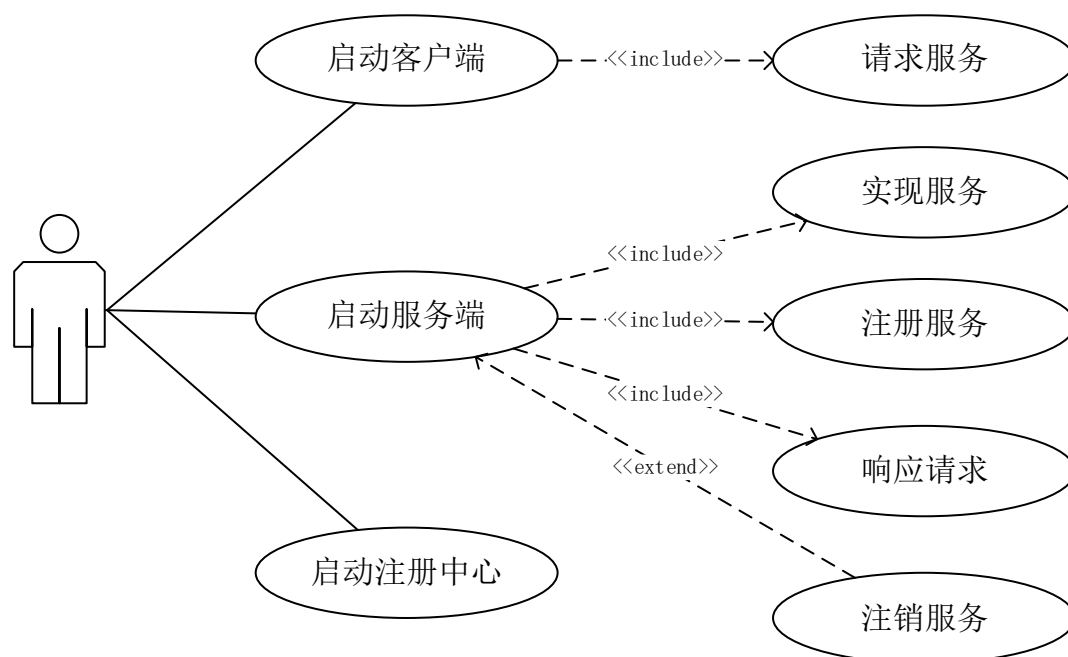


图 1-1 YR-RPC 基本用例图

为了使用 YA-RPC 框架，用户行为大致可以分为三类：

（1）启动客户端

客户端作为消费者，用于请求服务端提供的服务。在 Java 的具体实现中，通过动态代理来实现 RPC 调用，客户端将首先到注册中心获取服务端地址，然后按照 YA-RPC 自定义的协议，向服务端发起 RPC 请求。

（2）启动服务端

服务端作为服务提供者，首先要服务进行实现，然后将其注册到注册中心，并时刻准备相应客户端的请求，由于 YA-RPC 当前仅实现了 Socket 的通讯方式，因此将会对客户端的每个请求从线程池中分配一个线程处理请求，后续计划增加 Netty 的通讯方式，优化请求处理方式。此外，当服务器关机时，需要对已注册的服务进行注销。

（3）启动注册中心

注册中心维护了所有服务提供者的所暴露服务的信息，其中核心的信息就是 IP 和 PORT。服务端首先会将提供的服务注册到注册中心，随后客户端将与注册中心交互，订阅服务提供者的地址，随后通过该地址与服务提供者进行交互。

第二章 概要设计

2.1 系统架构

通过分析 RPC 的原理，调研市场上成熟的 RPC 框架，为了提高代码的结构性、可重用性、可读性和可维护性，通常采用将服务提供端服务端、服务消费端客户端和注册中心进行分离的方式来组织代码，将软件系统分为三个部分：客户端、服务端和 Registry。

YA-RPC 系统架构正是基于这种分离架构模式的思想来构建，具体的系统架构如图 2-1 所示：

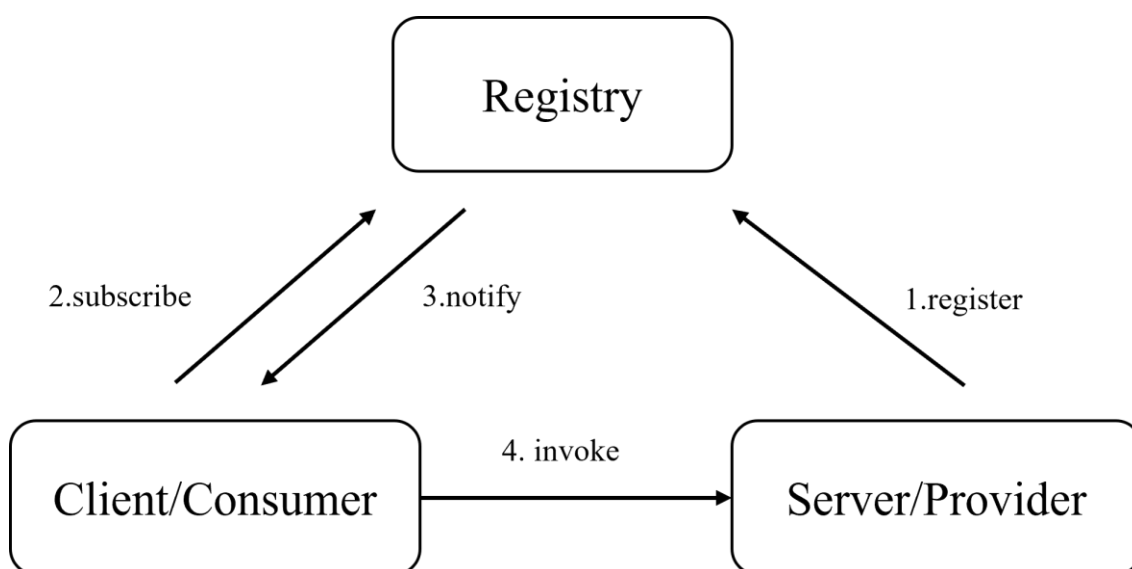


图 2-1 YA-RPC 架构图

结合上述分离架构思想，将 RPC 系统分为三个部分，服务提供者服务端向注册中心注册服务，服务消费者客户端通过注册中心拿到服务相关信息，然后再通过网络请求服务提供端服务端。

此外，在框架实现的具体细节中，还涉及到许多重要的模块：

（1）序列化与反序列化：序列化用于将数据结构或对象转换为二进制串，亦称为编码。反序列化则是将二进制串转换为数据结构或对象的过程，亦称为解码。YA-RPC 框架提供了 `protostuff` 和 `hessian` 两种序列化方式。

（2）压缩与解压：将序列化后的字节码再次进行压缩，减少网络传输数据包的体积，降低通信成本。YA-RPC 提供了 `gzip` 的压缩实现，此外，由于很多成熟的序列化框架的压缩已经做的很好，因此 YA-RPC 还提供了压缩的空实现。

（3）传输协议：RPC 远程调用实际上就是网络传输，常用的协议根据在网络

模型中的位置的不同主要分为应用层协议（HTTP1.1/2.0）和传输层协议（TCP）两种。在 Java 中，传统的 Socket 是基于 BIO 实现，其对 TCP/IP 协议进行了封装，因此需要为每一次客户端请求分配一个线程，否则就会阻塞。Netty 是基于 NIO 实现的网络编程框架，由于其具有高并发性，其性能相比 BIO 有了巨幅的提升。由于时间和精力限制，YA-RPC 框架仅提供 Socket 实现，但得益于 YA-RPC 框架对 SPI 机制的增强实现，可以很方便的为 Netty 进行后续扩展实现。

（4）自定义协议：由于 TCP 是面向连接的传输协议，TCP 传输的数据是以流的形式，而流数据是没有明确的开始结尾边界，所以 TCP 也没办法判断哪一段流属于一个消息，因此会导致粘包和半包问题，为了解决上述问题，YA-RPC 框架对客户端和服务端的通信协议进行了高度的自定义，有效的提高了传输效率。

（5）负载均衡：将客户端的请求负载均衡到多个服务器上，避免单个服务器响应同一个请求而导致宕机、崩溃。YA-RPC 框架提供了随机和轮询两种负载均衡策略的实现。

2.2 系统流程图

为了更好的描述 YA-RPC 框架的使用流程，建模系统流程图，如图 2-1 所示：

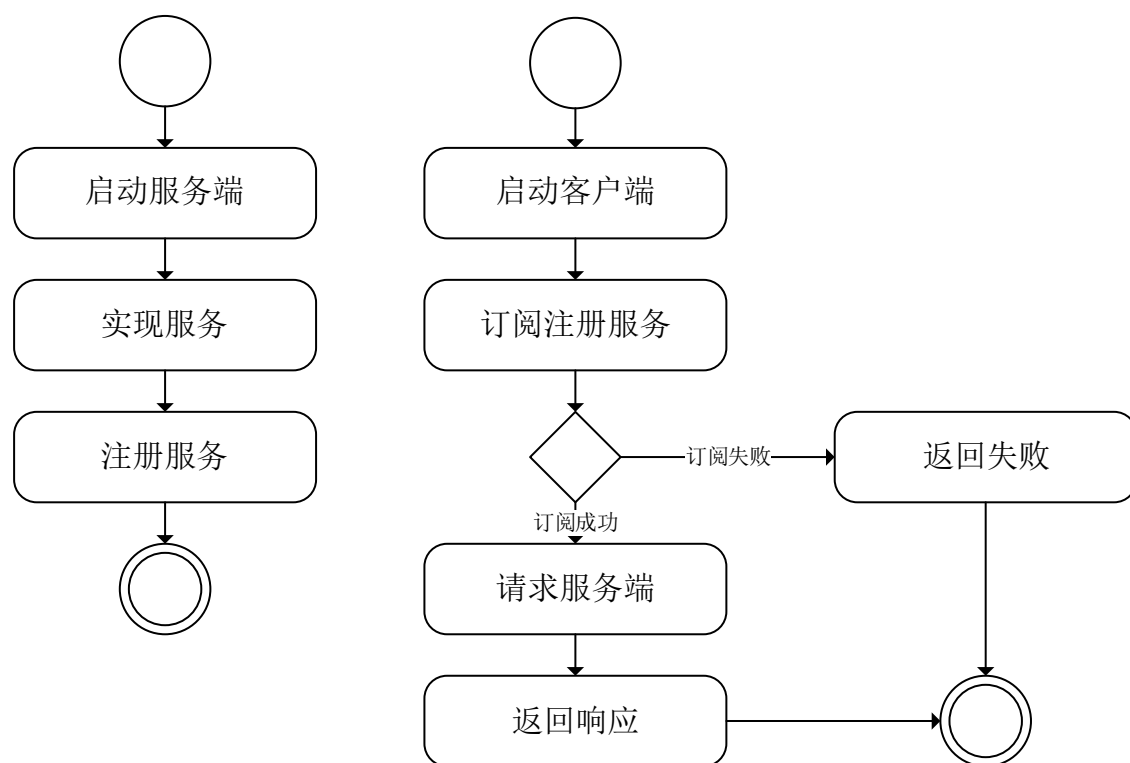


图 2-1 YA-RPC 框架流程图

用户通过 Maven 导入 YA-RPC 框架依赖，即可快速的构建客户端和服务端。

通过客户端可以像调用本地方法一样调用远程方法，其内部将通过动态代理的方式，首先向注册中心订阅所调用的服务，如果订阅成功，将获取到提供该服务的服务端地址，通过网络传输将请求发送给客户端，并处理相应。

此外，用户需要构建服务端以相应客户端的请求，服务端中需要实现双方约定的接口，服务端可以提供单个接口的不同实现，使用 `group` 进行分组，此外对于服务的不兼容升级，使用 `version` 进行区分。完成服务的实现后，需要将服务注册到注册中心，以便对服务进行统一管理，此时客户端只需要注册中心订阅，即可拿到对应服务的提供者的地址，通过该地址向服务提供者发起 RPC 请求。

2.3 代码架构



代码架构 2-1 总体架构图

其中 `remoting` 中是代码的核心，其主要功能是实现了自定义协议的编码和解码器，并提供了客户端和服务端的框架实现，用户可以通过实例化快速的启动 RPC 框架的客户端和服务端，此外，还对网络传输进行了细粒度的实现，且支持完全的 At-Least-Once 语义和初步的 At-Most-Once 语义：



代码架构 2-2 remoting 模块架构图

(1) 编码器：主要是编码协议和解码自定义的协议。

(2) 钩子函数：用于在关机时取消本机注册的服务，和删除服务端实现 At-Most-Once 语义的缓存。

(3) 注册表：Server 向注册中心注册服务时，会更新该表，添加一条记录。Client 向 Server 请求服务时，Server 根据请求的数据，从注册表中拿出服务接口的实现类.class，并基于此构建实现类实例，执行对应的方法。

(4) 对于 socket 的传输实现，由于其 BIO 的特性，Client 的每次请求，Server 都为其分配一个线程处理请求。

第三章 详细设计及实现

3.1 SPI 扩展机制

RPC 扩展拥有很多可扩展的地方，如：序列化类型、压缩类型、负载均衡类型、注册中心类型等，因此一个成熟的 RPC 框架应该留下扩展点，让使用者可以不需要修改框架，就能自己去实现扩展。

JDK 提供了 SPI 机制用于扩展，但仍存在资源浪费、使用麻烦等缺陷，在调研了市场上各种成熟的 RPC 框架之后，YA-RPC 框架选择参考 dubbo 框架对 SPI 机制进行增强实现。

YA-RPC 对 SPI 机制的增强实现位于/ya-rpc/ya-rpc-common/extensions 目录下，首先定义注解 SPI，当接口被该注解标记时，则认为该接口为扩展接口。通过实现 ExtensionLoader 类，该类基于反射获取 SPI 标记的接口，并在约定的文件夹/resources/META-INF/extensions 文件夹下寻找配置文件，懒惰加载所需的实现类，具体代码细节不再赘述，下面给出扩展使用方法，以序列化为例，YA-RPC 框架给出了两种序列化的实例，分别是 hessian 和 protostuff。

(1) 定义 SPI 注解，被该注解标记的接口，被认为是可扩展接口，可以通过 YA-RPC 增强的 SPI 机制快速的进行扩展。

```
@SPI
public interface Serializer {
    byte[] serialize(Object object);
}
```

代码 3-1 SPI 接口

(2) 提供 Serializer 接口的实现类：ProtostuffSerializer 和 HessianSerializer，其代码位于/ya-rpc/ya-rpc-core/serialize，由于 Java 自带的序列化方式效果并不优秀，因此此处采用业界更为认可的成熟的序列化框架。

(3) 完善配置文件约定放置于 resources/META-INF/extensions，配置文件的名字是接口的位置。例如序列化扩展的配置文件，其文件名应该被命名为 Serializer 的类路径，即 com.wang.serialize.Serializer。

```
hessian=com.wang.serialize.hessian.HessianSerializer
protostuff=com.wang.serialize.protostuff.ProtostuffSerializer
```

代码 3-2 序列化配置文件

(4) 基于懒惰实现获取序列化扩展类

```
public static void main(String[] args) {
    ExtensionLoader<Serializer> extensionLoader =
        ExtensionLoader.getExtensionLoader(Serializer.class);
    Serializer serializer = extensionLoader.getExtension("protostuff");
}
```

代码 3-3 增强 SPI 机制获取序列化扩展类

基于上述机制，可以十分高效的对序列化类型、压缩类型、负载均衡类型、注册中心类型等进行扩展，YA—RPC 框架也都提供了多种实现，用户基于自身的需求也可以进行十分快速的扩展。

3.2 自定义协议

在网络传输中，为了让收发两端正确解析请求，统一的协议是必不可少的。此外，使用约定的协议，也可以有效的解决 Socket 底层使用 TCP 协议而导致的粘包和半包问题。

YA-RPC 框架采用消息头和消息体的方式制定私有协议，其具体字段如图 3-1 所示：

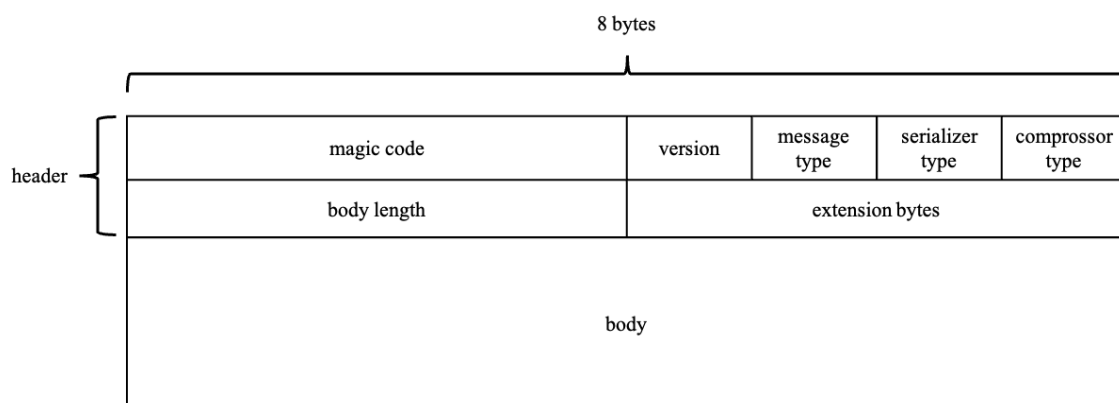


图 3-1 YA-RPC 自定义协议

(1) 4B magic code (魔数)：通信双方协商的一个暗号。魔数的思想在很多场景中都有体现，如 Java Class 文件开头就存储了魔数 0xCAFEBADE，在 JVM 加载 Class 文件时首先就会验证魔数对的正确性。魔数的作用是用于服务端在接收数据时先解析出魔数做正确性对比，如果和协议中的魔数不匹配，则认为是非法数据，可以直接关闭连接或采取其他措施增强系统安全性。值得注意的是，魔数仅用于简单的校验，如果有安全性方面的需求，需要使用其他手段，例如 SSL/TLS。

(2) 1B version (版本): 为了应对业务需求的变化, 可能需要对自定义协议的结构或字段进行改动。不同版本的协议对应的解析方法也是不同的。

(3) 1B message type (消息类型): 消息类型分为请求和响应两种。解码器可以根据消息类型来确定解析的类型。

(4) 1B compressor (压缩类型): 压缩类型分为 gzip 和 dummy。解码器会根据压缩类型, 通过 SPI 机制生成解压器, 对数据进行解压缩。

(5) 1B serializer (序列化类型): 序列化类型分为 protostuff 和 hessian。解码器会根据序列化类型, 通过 SPI 机制生成反序列化器, 对数据进行反序列化。

(6) 4B body length (消息体长度): 为了避免 Socket 的粘包问题, 在协议头中显示说明 body 的长度。

(7) 4B extension bytes (扩展字节): 待后续扩展, 例如为了完整的实现服务端的 At-Most-Once 语义, 可以在协议头中存入顺序增长的 requestId 等字段

(8) body: 常来说是请求的参数、响应的结果, 再经过序列化、压缩后的字节数组, 在 YA-RPC 中实现为 RpcRequest 或者 RpcResponse 实例对象。

在协议实现中, 将为该自定义协议实现对应的编码器和解码器, 其实现代码位于/ya-rpc/ya-rpc-core/remoting/socket/codec 模块中, 编码器主要是将协议定义各个字段写出, 而解码器则是按顺序读入各个字段。

3.3 注册中心

YA-RPC 中服务消费者, 也就是客户端, 需要请求服务端的服务提供方的接口, 必须知道服务提供者的地址, 而注册中心则是存储了各种服务提供者的地址。

理论上来说, 客户端可以自己配置服务端的地址, 但在生产实践中这种高度耦合的配置方式很难满足高速发展的业务需求, 存在繁琐的配置、高昂的沟通成本以及服务器更换、宕机而导致的服务异常等问题。

因此, YA-RPC 引入注册中心, 用于管理服务提供者的地址配置, 从本质上来说, 注册中心就是为了让客户端更好的感知到服务提供者地址, 而同时, 由于其独立于服务端, 因此同时也能兼容负载均衡的功能。

YA-RPC 框架目前仅提供了 Zookeeper 的实现, Server 注册后在 Zookeeper 中的存储形式如代码 3-4 所示。

```
/ya-rpc-provider
/com.wang.service.UserService?group=g1&version=v1
/[127.0.0.1:8713, 127.0.0.1:8712]
/com.wang.service.UtilService?group=g1&version=v1
/[127.0.0.1:8713]
```

代码 3-4 注册中心示例

当客户端调用接口 `UserService` 的方法时，由于服务端可能提供单个接口的不同实现，使用 `group` 指定分组，此外对于服务的不兼容升级，使用 `version` 指定版本。

客户端会从注册中心获取提供特定实现的服务的服务器的地址，当然，可能会存在多个服务器提供该服务，通过负载均衡从多个服务器中选择一个，然后客户端将请求发送给服务端，服务端将处理结果响应返回给客户端。因此，协议的通信仍旧是在客户端和服务端中进行。

Zookeeper 注册中心的实现代码位于 `/ya-rpc/ya-rpc-core/registry` 模块，用户可以通过实现 `Registry` 接口，实现其它的注册中心方案，例如 Eureka。

对于多个服务提供者的负载均衡策略，YA-RPC 框架实现了随机和轮询两种策略，其实现代码位于 `/ya-rpc/ya-rpc-core/loadbalance` 模块。此外，框架也提供了后续扩展的 SPI 机制，只需要实现 `LoadBalancer` 接口，并配置对应该接口的扩展配置即可。

3.4 远程调用

YA-RPC 采用 Client/Server 的架构模式，服务端作为服务提供者，将实现服务接口，并将其注册到注册中心，同时响应客户端请求；客户端作为服务消费者，从注册中心获取服务提供者地址，向其发送 RPC 请求，其具体过程如图 3-2 所示：

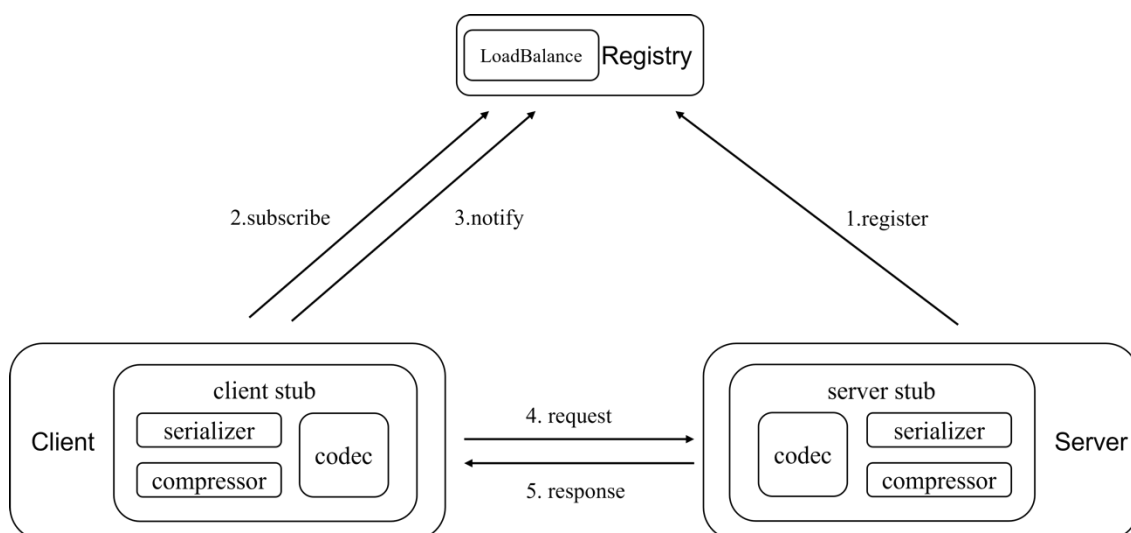


图 3-2 远程调用细节图

当客户端向注册中心订阅服务提供者地址时，注册中心通过负载均衡从多个

服务提供者（如果存在多个）中根据负载均衡策略返回其中一个服务提供者的地址给客户端，客户端将请求进行序列化和压缩，并通过编码器将其编码为符合协议规范请求，通过 **Socket** 信道经过网络传输发送给服务端。

当服务端接收到客户端的请求时，首先通过解码器根据双方约定的协议解码请求，首先将判断魔数和协议版本是否正确，随后对字节码进行解压缩和反序列化，最终得到请求实体，随后检查注册表获取服务类，通过反射调用服务接口，将响应进行序列化和压缩，并通过编码器将其编码为符合协议规范请求，通过 **Socket** 信道经过网络传输发送回客户端。此外，服务端在关机时将会执行钩子函数自动取消注册本机服务。

客户端收到服务端的响应时，首先通过解码器根据双方约定的协议解码请求，首先将判断魔数和协议版本是否正确，随后对字节码进行解压缩和反序列化，最终得到响应实体，完成 **RPC** 调用。

上述实现的代码位于 `/ya-rpc/ya-rpc-core/remoting`，但由于代码过于复杂，限于篇幅不在此赘述，有兴趣可查阅源码，笔者在其中写了详尽的注释。

此外，客户端实现了 **At-Least-Once** 语义，将在响应超时后对请求进行重传，默认设置为 3 次，超时时间为 5 秒，用户可以通过配置类 `/ya-rpc-common/config/CommonConfig` 进行自定义配置。其伪代码如代码 3-5 所示：

```
while(retryTimes < CommonConfig.retry):
    send() // 发起请求
    if success: break; // 请求成功则退出循环
```

代码 3-5 客户端 **At-Least-Once** 语义

为了保证 **At-Least-Once** 能够正确执行，需要保证仅有读操作，不进行写操作或 **Server** 采用一定的方式应对重复和重排序，即实现 **At-Most-Once** 语义。

YA-RPC 框架的服务端实现了 **At-Most-Once** 语义，当收到客户端的重复请求时，仅执行一次请求，可以节省服务器资源，且对 **DDOS** 攻击起到一定的抵御功能，其伪代码如代码 3-6 所示：

```
Map<requestId, RpcResponse> cache;
if cache.containsKey(rId):
    res = cache.get(rId);
else:
    res = handler() // 处理请求
    cache.put(rId, res);
return res
```

代码 3-6 服务端 At-Most-Once 语义

代码 3-6 对于 At-Most-Once 语义的实现存在三个细节问题：

- (1) 缓存无限增长: 客户端在 RPC 请求时包含语义: "seen all replies $\leq X$ ", 服务端收到时删除此前所有请求的缓存。
- (2) 当请求 A 正在执行时, 如何处理重复请求 A2: A 处理时尚未写入 cacheMap 中, 此时应该为正在执行的 RPC 请求新增标记符 pending。A2 发现是 pending 状态时, 等待其处理结束。
- (3) 服务器崩溃或重启, 导致内存中维护的 cacheMap 丢失: 将 cacheMap 写入磁盘持久化处理。

YA-RPC 框架并未着力于完美的实现上述三个细节问题, 仅考虑了最简单的情况。对于缓存无限增长问题, 通过注册钩子函数在缓存写入 20 秒后删除缓存。20 秒的选择是因为在默认配置中客户端只会重传 3 次, 且超时时间为 5 秒。对于另外两个问题, 并未进行处理, 因此在当前框架的实现下, 客户端只能发送读请求, 否则当遇到第二个细节问题时, 将会重复执行。

同样, 得益于增强版的 SPI 机制, YA-RPC 只需要实现 RpcClient、RpcServer 等接口, 即可快速地对框架客户端和服务端进行扩展。在后续的更新中, 计划为框架添加基于 NIO 实现的 Netty 传输方式, 同时将对服务端的 At-Most-Once 语义做进一步的优化和完善。

第四章 系统测试

4.1 系统测试概述

本项目在开发过程中经过详细的单元测试。黑盒测试作为系统测试的常用方法之一，在黑盒测试中，不关心程序内部结构及特性，而仅仅对程序的接口进行测试，检测在按照特定的使用方式进行使用时，程序能否接受使用数据，并产生符合预期的输出结果。

本项目的主要任务是实现一个简易的 RPC 框架，称为 YA-RPC，用户通过 Maven 导入 YA-RPC 框架依赖，即可快速的构建客户端和服务端，本框架的主要实现成果如下：

(1) 跨平台字节流序列化以及自定义协议的实现，使得本框架可跨平台支持所有的基本数据类型，以及 Java 特有的对象类型。

(2) 客户端支持 At-Least-Once 语义；服务端支持 At-Most-Once 语义。

(3) 注册中心的解耦以及负载均衡的实现，使得本框架支持横向扩展任意数量的服务端；通过线程池，可以支持任意数量的客户端同时请求(默认设置 100 个)。

4.2 系统测试环境

系统测试环境主要分为三个部分：

(1) 注册中心：使用 docker compose 部署 Zookeeper 集群，本实验启动 1 个 master、2 个 follower 的集群。

(2) 客户端：通过 Maven 导入 YA-RPC 框架，启动 2 个客户端。

(3) 服务端：通过 Maven 导入 YA-RPC 框架，启动 2 个服务端。

首先定义实现的接口，代码位于：/ya-rpc/ya-rpc-demo/ya-rpc-api，本次主要定义两个接口 UtilService 和 UserService，如代码 4-1 和代码 4-2 所示：

```
public interface UtilService {  
    /**  
     * 计算 a 和 b 的合  
     * @param a 加数  
     * @param b 被加数  
     * @return 合  
     */  
    float sum(float a, float b);  
    /**  
     * 将字符串变为大写模式
```

```

    * @param str 原始字符串
    * @return 大写后的字符串
    */
    String uppercase(String str);
}

```

代码 4-1 UtilService 接口

```

public interface UserService {
    /**
     * 根据用户 id 获取用户信息，业务中通过查询数据库获取
     *
     * @param id 用户 id
     * @return 用户信息，如果获取不到，返回 null
     */
    User getUserById(int id);
    /**
     * 根据用户 name 获取用户信息，业务中通过查询数据库获取
     *
     * @param name 用户 name
     * @return 用户信息，如果获取不到，返回 null
     */
    User getUserByName(String name);
}

```

代码 4-2 UserService 接口

然后启动客户端，代码位于/ya-rpc/ya-rpc-demo/ya-rpc-client，本次测试中，将启动两个客户端，其中 Client1 用于请求 UtilService 服务，Client2 用于请求 UserService 服务，其代码详情如代码 4-3 和代码 4-4 所示：

```

@Slf4j
public class SocketClientMain {
    public static void main(String[] args) {
        SocketRpcClient client = new SocketRpcClient();
        UtilService utilService =
client.getStub(UtilService.class,"groupName1","version1");
        float sum = utilService.sum((float) 20.08, (float) 06.26);
        String uppercase = utilService.uppercase("happytsing");
        log.info("sum: {} uppercase: {}",sum,uppercase);
    }
}

```

代码 4-3 Client1 请求 UtilService


```
@Slf4j
public class SocketClientMain2 {
    public static void main(String[] args) {
        SocketRpcClient client = new SocketRpcClient();
        UserService userService =
client.getStub(UserService.class,"groupName1","version1");
        User userGetById = userService.getUserById(22080626);
        User userGetByName = userService.getUserByName("toucher le
port");
        log.info("User1: {} User2: {}",userGetById,userGetByName);
    }
}
```

代码 4-4 Client2 请求 UserService

最后启动服务端，服务端首先需要实现接口，然后不断监听客户端的请求，并为其分配线程处理请求。此处为了测试负载均衡的调用，将启动两个提供相同服务的服务端。

接口的实现代码位于/ya-rpc/ya-rpc-dmo/ya-rpc-server/serviceImpl 中，在此处不赘述。服务端的启动示例代码如代码 4-5 所示：

```
public class SocketServerMain {
    public static void main(String[] args) {
        SocketRpcServer server = new SocketRpcServer();
        server.registerService(new
ServiceSignature(UserServiceImpl.class,"groupName1","version1"),
UserServiceImpl.class);
        server.registerService(new ServiceSignature(new
UtilServiceImpl(),"groupName1","version1"), UtilServiceImpl.class);
        server.start();
    }
}
```

代码 4-5 Server 示例代码

4.3 系统测试用例及结果

（一）服务端

服务端主要用于实现并注册服务，响应客户端请求，此外，在服务器关机时将自动取消注册本机服务，以及对客户端的多次重复请求按照 At-Most-Once 语义仅执行一次，测试用例如表 4-1 所示：

表 4-1 服务端测试用例

编号	内容描述	过程描述	预期结果	测试结果
101	注册服务	注册服务至注册中心	注册成功	与预期结果一致
102	取消注册服务	当关机时，自动取消注册本机服务	自动取消注册成功	与预期结果一致
103	响应 RPC 请求	客户端发送请求时，服务端为每次请求分配一个线程处理，且重复请求仅执行一次，即 At-Most-Once 语义。	响应成功	与预期结果一致

(1) 用例 101

启动两个服务端之后，通过查看注册中心中注册的服务结果，如图 4-1 所示：

```

[zk: localhost:2181(CONNECTED) 1] ls /ya-rpc-provider
[com.wang.service.UserService?group=groupName1&version=version1, com.wang.service.UtilService?group=groupName1&version=version1]
[zk: localhost:2181(CONNECTED) 2] ls /ya-rpc-provider/com.wang.service.UserService?group=groupName1&version=version1
[192.168.31.78:8712, 192.168.31.78:8713]
[zk: localhost:2181(CONNECTED) 3] ls /ya-rpc-provider/com.wang.service.UtilService?group=groupName1&version=version1
[192.168.31.78:8712, 192.168.31.78:8713]
[zk: localhost:2181(CONNECTED) 4]

```

图 4-1 服务注册

如图 4-1 所示，两个服务端都向注册中心注册了服务 UserService 和 UtilService，两个服务器的地址分别是 192.168.31.78:8712 和 192.168.31.78:8713。

(2) 用例 102

关闭服务端，可以发现注册的钩子函数启动，自动注销本机注册的所有服务，如图 4-2 所示：

```

18:26:24.410 [main] INFO c.w.r.z.ZookeeperRegistry - [register,60] - Register service success, The providerPath is /ya-rpc-provider/com.wang.service.UserService?group=groupName1&version=version1/192.168.31.78:8713
18:26:24.411 [main] INFO c.w.r.s.s.ServiceRegedit - [put,23] - Update ServiceRegedit success. The record is [ServiceSignature=com.wang.service.UserService?group=groupName1&version=version1, clazz=class com.wang.serviceImpl.groupName1.version1.UserServiceImpl]
18:26:24.425 [main] INFO c.w.r.z.CuratorUtils - [createPersistentNode,43] - Create node success. The node is: /ya-rpc-provider/com.wang.service.UtilService?group=groupName1&version=version1/192.168.31.78:8713
18:26:24.425 [main] INFO c.w.r.z.ZookeeperRegistry - [register,60] - Register service success, The providerPath is /ya-rpc-provider/com.wang.service.UtilService?group=groupName1&version=version1/192.168.31.78:8713
18:26:24.425 [main] INFO c.w.r.s.s.ServiceRegedit - [put,23] - Update ServiceRegedit success. The record is [ServiceSignature=com.wang.service.UtilService?group=groupName1&version=version1, clazz=class com.wang.serviceImpl.groupName1.version1.UtilServiceImpl]
18:26:24.426 [main] INFO c.w.r.s.s.ServerHook - [addShutdownHook,22] - addShutdownHook for clear all services registered with this machine.
18:34:05.747 [Thread-0] INFO c.w.r.z.CuratorUtils - [removeNode,64] - Delete node success. The node is /ya-rpc-provider/com.wang.service.UserService?group=groupName1&version=version1/192.168.31.78:8713
18:34:05.752 [Thread-0] INFO c.w.r.z.CuratorUtils - [removeNode,64] - Delete node success. The node is /ya-rpc-provider/com.wang.service.UtilService?group=groupName1&version=version1/192.168.31.78:8713
18:34:05.752 [Thread-0] INFO c.w.r.s.s.ServerHook - [lambda$0,27] - Clear all services registered with this machine success.

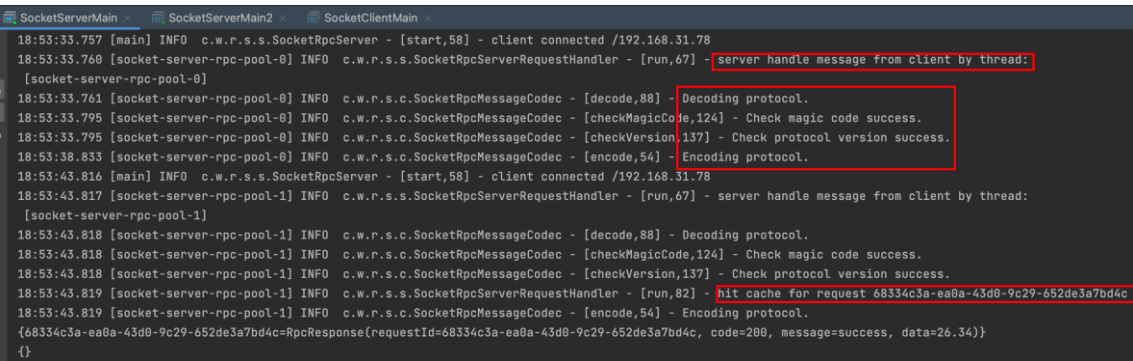
```

图 4-2 注销服务

(3) 用例 103

服务端接收客户端的请求，并分配一个线程处理请求，首先解码请求协议，该步骤首先检查魔数和协议版本号，随后执行请求，再次编码响应给客户端。

为了测试服务端的 At-Most-Once 语义，服务端在处理后等待 5 秒，使得客户端超时重发请求，其运行结果如图 4-3 所示：



```
18:53:33.757 [main] INFO c.w.r.s.s.SocketRpcServer - [start,58] - client connected /192.168.31.78
18:53:33.760 [socket-server-rpc-pool-0] INFO c.w.r.s.s.SocketRpcServerRequestHandler - [run,67] - server handle message from client by thread:
[socket-server-rpc-pool-0]
18:53:33.761 [socket-server-rpc-pool-0] INFO c.w.r.s.s.SocketRpcMessageCodec - [decode,88] - Decoding protocol.
18:53:33.795 [socket-server-rpc-pool-0] INFO c.w.r.s.s.SocketRpcMessageCodec - [checkMagicCode,124] - Check magic code success.
18:53:33.795 [socket-server-rpc-pool-0] INFO c.w.r.s.s.SocketRpcMessageCodec - [checkVersion,137] - Check protocol version success.
18:53:38.833 [socket-server-rpc-pool-0] INFO c.w.r.s.s.SocketRpcMessageCodec - [encode,54] - Encoding protocol.
18:53:43.816 [main] INFO c.w.r.s.s.SocketRpcServer - [start,58] - client connected /192.168.31.78
18:53:43.817 [socket-server-rpc-pool-1] INFO c.w.r.s.s.SocketRpcServerRequestHandler - [run,67] - server handle message from client by thread:
[socket-server-rpc-pool-1]
18:53:43.818 [socket-server-rpc-pool-1] INFO c.w.r.s.s.SocketRpcMessageCodec - [decode,88] - Decoding protocol.
18:53:43.818 [socket-server-rpc-pool-1] INFO c.w.r.s.s.SocketRpcMessageCodec - [checkMagicCode,124] - Check magic code success.
18:53:43.818 [socket-server-rpc-pool-1] INFO c.w.r.s.s.SocketRpcMessageCodec - [checkVersion,137] - Check protocol version success.
18:53:43.819 [socket-server-rpc-pool-1] INFO c.w.r.s.s.SocketRpcServerRequestHandler - [run,82] - hit cache for request 68334c3a-ea0a-43d0-9c29-652de3a7bd4c
18:53:43.819 [socket-server-rpc-pool-1] INFO c.w.r.s.s.SocketRpcMessageCodec - [encode,54] - Encoding protocol.
{68334c3a-ea0a-43d0-9c29-652de3a7bd4c=RpcResponse(requestId=68334c3a-ea0a-43d0-9c29-652de3a7bd4c, code=200, message=success, data=26.34)}
{}
```

图 4-3 响应请求

可以看到，服务端第一次接收到请求后，分配了线程 0 处理，但由于客户端超时，发起了第二次请求，服务端接收到第二次请求后，命中缓存，没有进行第二次重复处理，而是将结果直接返回给客户端。

(二) 客户端

图片缩略图函数属于 EVENT 函数，即触发器函数，将特定的触发器被触发时，将自动执行设置好的操作，图片缩略图测试用例如表 4-2 所示：

表 4-2 客户端测试用例

编号	内容描述	过程描述	预期结果	测试结果
201	注册中心负载均衡	当客户端发起请求时，会先向注册中心订阅服务地址，注册中心会根据负载均衡策略（默认轮询）返回服务地址。	成功获取地址	与预期结果一致
202	超时重传机制 At-Least-Once	客户端请求超时后，会再次向服务端发起重复请求，默认最多重试三次。	超时重传成功	与预期结果一致
203	远 程 调 用 UtilService 服务	通过 RPC 调用远程服务 UtilService 的方法 sum 和 uuppercase。	成功调用	与预期结果一致
204	远 程 调 用 UserService	通过 RPC 调用远程服务 UserService 的方法 getUserById 和 getUserByName，主要用于测试返回对象类型。	成功调用	与预期结果一致

(1) 用例 201

客户端第一次发起请求时，选择服务器地址 192.168.31.78:8712，随后超时重传，第二次基于轮询的负载均衡策略，选择 192.168.31.78:8713 请求，其测试结果如图 4-4 所示：

```

18:53:33.740 [main] INFO c.w.r.z.ZookeeperRegistry - [lookup,99] - Lookup service success, not hit provider cache, get from zookeeper. The providerList is
[192.168.31.78:8712, 192.168.31.78:8713]
18:53:33.740 [main] INFO c.w.r.z.ZookeeperRegistry - [watch,118] - Watch service success, Watching service is com.wang.service
.UtilService?group=groupName&version=version1
18:53:33.753 [main] INFO c.w.r.z.ZookeeperRegistry - [lookup,108] - SelectedProvider is 192.168.31.78:8712
18:53:33.757 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [encode,54] - Encoding protocol.
18:53:33.770 [Curator-SafeNotifyService-0] INFO c.w.r.z.ZookeeperRegistry - [lambda$0,120] - type: NODE_CREATED, oldData: null, newData:
ChildData[path='/ya-rpc-provider/com.wang.service.UtilService?group=groupName&version=version1', stat=12884902265,12884902265,1668939897251,1668939897251,0,14,0,0,
0,2,12884902315
, data=[]]
18:53:33.777 [Curator-SafeNotifyService-0] INFO c.w.r.z.CuratorUtils - [getChildrenNodes,78] - Get children success. The children are [192.168.31.78:8712,
192.168.31.78:8713]
18:53:33.786 [Curator-SafeNotifyService-0] INFO c.w.r.z.ZookeeperRegistry - [lambda$0,120] - type: NODE_CREATED, oldData: null, newData:
ChildData[path='/ya-rpc-provider/com.wang.service.UtilService?group=groupName&version=version1/192.168.31.78:8712', stat=12884902315,12884902315,1668941607017,
1668941607017,0,0,0,13,0,12884902315
, data=[49, 57, 50, 46, 49, 54, 56, 46, 51, 49, 46, 55, 56]]
18:53:33.793 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [decode,88] - Decoding protocol.
18:53:33.795 [Curator-SafeNotifyService-0] INFO c.w.r.z.CuratorUtils - [getChildrenNodes,78] - Get children success. The children are [192.168.31.78:8712,
,192.168.31.78:8713]
18:53:33.795 [Curator-SafeNotifyService-0] INFO c.w.r.z.ZookeeperRegistry - [lambda$0,120] - type: NODE_CREATED, oldData: null, newData:
ChildData[path='/ya-rpc-provider/com.wang.service.UtilService?group=groupName&version=version1/192.168.31.78:8713', stat=12884902292,12884902292,1668941285941,
1668941285941,0,0,0,13,0,12884902292
, data=[49, 57, 50, 46, 49, 54, 56, 46, 51, 49, 46, 55, 56]]
18:53:33.804 [Curator-SafeNotifyService-0] INFO c.w.r.z.CuratorUtils - [getChildrenNodes,78] - Get children success. The children are [192.168.31.78:8712,
192.168.31.78:8713]
18:53:38.800 [main] INFO c.w.r.s.c.SocketRpcClient - [lambda$0,103] - Wait Response timeout, Retry times: 1
18:53:38.801 [main] INFO c.w.r.z.ZookeeperRegistry - [lookup,94] - Lookup service success, hit provider cache. The providerList is [192.168.31.78:8712, 192.168.31
.78:8713]
18:53:38.801 [main] INFO c.w.r.z.ZookeeperRegistry - [watch,118] - Watch service success, Watching service is com.wang.service
.UtilService?group=groupName&version=version1
18:53:38.802 [main] INFO c.w.r.z.ZookeeperRegistry - [lookup,108] - SelectedProvider is 192.168.31.78:8713
    
```

图 4-4 负载均衡

再次测试多个客户端同时发起请求，负载均衡仍旧生效。

(2) 用例 202

为了测试客户端的 At-Least-Once 语义，服务端在处理完等待 20 秒，使得客户端重试三次仍旧超时报错，测试结果如图 4-5 所示：

```

19:10:33.687 [main] INFO c.w.r.s.c.SocketRpcClient - [lambda$0,103] - Wait Response timeout, Retry times: 3
Exception in thread "main" com.wang.exception.RpcClientException Create breakpoint : Get response fail. Retry times: 3
at com.wang.remoting.socket.client.SocketRpcClient.lambda$0(SocketRpcClient.java:112) <1 internal line>
at com.wang.SocketClientMain.main(SocketClientMain.java:12)
    
```

图 4-5 超时重传

可以看到，根据 At-Least-Once 的语义，客户端将在超时后重试 3 次，并且在第 3 次重试超时后，直接抛出超时异常。

(3) 用例 203

测试用例 203 所用的客户端调用代码如代码 4-1 所示：

```

public static void main(String[] args) {
    SocketRpcClient client = new SocketRpcClient();
    UtilService utilService =
client.getStub(UtilService.class,"groupName1","version1");
    
```

```
float sum = utilService.sum((float) 20.08, (float) 06.26);
String uppercase = utilService.uppercase("happytsing");
log.info("sum: {} uppercase: {}",sum,uppercase);
}
```

代码 4-1 UtilService 测试代码

测试结果如图 4-6 所示：

```
SocketServerMain SocketServerMain2 SocketClientMain
, data=[49, 57, 50, 46, 49, 54, 56, 46, 51, 49, 46, 55, 56]}
19:16:57.859 [Curator-SafeNotifyService-0] INFO c.w.r.z.CuratorUtils - [getChildrenNodes,78] - Get children success. The children are [192.168.31.78:8712,
192.168.31.78:8713]
19:16:57.859 [Curator-SafeNotifyService-0] INFO c.w.r.z.ZookeeperRegistry - [lambda$0,120] - type: NODE_CREATED, oldData: null, newData:
ChildData[path=/ya-rpc-provider/com.wang.service.UtilService?group=groupName&version=version1/192.168.31.78:8713', stat=12884902351,12884902351,1668942824209,
1668942824209,0,0,0,13,0,12884902351
, data=[49, 57, 50, 46, 49, 54, 56, 46, 51, 49, 46, 55, 56]}
19:16:57.860 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [decode,88] - Decoding protocol.
19:16:57.865 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [checkMagicCode,124] - Check magic code success.
19:16:57.866 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [checkVersion,137] - Check protocol version success.
19:16:57.866 [Curator-SafeNotifyService-0] INFO c.w.r.z.CuratorUtils - [getChildrenNodes,78] - Get children success. The children are [192.168.31.78:8712,
192.168.31.78:8713]
19:16:57.869 [main] INFO c.w.r.z.ZookeeperRegistry - [lookup,94] - Lookup service success, hit provider cache. The providerList is [192.168.31.78:8712, 192.168.31
.78:8713]
19:16:57.870 [main] INFO c.w.r.z.ZookeeperRegistry - [watch,118] - Watch service success, Watching service is com.wang.service
.UtilService?group=groupName&version=version1
19:16:57.870 [main] INFO c.w.r.z.ZookeeperRegistry - [lookup,108] - SelectedProvider is 192.168.31.78:8713
19:16:57.871 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [encode,54] - Encoding protocol.
19:16:57.872 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [decode,88] - Decoding protocol.
19:16:57.882 [Curator-SafeNotifyService-0] INFO c.w.r.z.ZookeeperRegistry - [lambda$0,120] - type: NODE_CREATED, oldData: null, newData:
ChildData[path=/ya-rpc-provider/com.wang.service.UtilService?group=groupName&version=version1', stat=12884902265,12884902265,1668939897251,1668939897251,0,22,0,0,
0,2,12884902351
, data=[]}
19:16:57.883 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [checkMagicCode,124] - Check magic code success.
19:16:57.883 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [checkVersion,137] - Check protocol version success.
19:16:57.883 [main] INFO c.w.SocketClientMain ~ [main,14] - sum: 26.34 uppercase: HAPPYTSING
19:16:57.889 [Curator-SafeNotifyService-0] INFO c.w.r.z.CuratorUtils - [getChildrenNodes,78] - Get children success. The children are [192.168.31.78:8712,
192.168.31.78:8713]
Process finished with exit code 0
```

图 4-6 UtilService 测试结果

如图 4-6 所示，RPC 调用 sum 方法，成功将 20.08 和 06.26 两个浮点数相加，返回了结果 26.34。RPC 调用 uppercase 方法，成功将“happytsing”大写转换为“HAPPYTSING”。

(4) 用例 204

测试用例 204 所用的客户端调用代码如代码 4-2 所示：

```
public static void main(String[] args) {
    SocketRpcClient client = new SocketRpcClient();
    UserService userService =
client.getStub(UserService.class,"groupName1","version1");
    User userGetById = userService.getUserById(22080626);
    User userGetByName = userService.getUserByName("toucher le port");
    log.info("User1: {} User2: {}",userGetById,userGetByName);
}
```

代码 4-2 UserService 测试代码

测试结果如图 4-7 所示：

```

SocketServerMain  SocketServerMain2  SocketClientMain2
19:46:51.102 [Curator-SafeNotifyService-0] INFO c.w.r.z.CuratorUtils - [getChildrenNodes,78] - Get children success. The children are [192.168.31.78:8712,
192.168.31.78:8713]
19:46:51.103 [Curator-SafeNotifyService-0] INFO c.w.r.z.ZookeeperRegistry - [lambda$0,120] - type: NODE_CREATED, oldData: null, newData:
ChildData{path='/ya-rpc-provider/com.wang.service.UserService?group=groupName&version=version1/192.168.31.78:8713', stat=12884902350,12884902350,1668942824199,
1668942824199,0,0,0,13,0,12884902350
, data=[49, 57, 50, 46, 49, 54, 56, 46, 51, 49, 46, 55, 56]}
19:46:51.110 [Curator-SafeNotifyService-0] INFO c.w.r.z.CuratorUtils - [getChildrenNodes,78] - Get children success. The children are [192.168.31.78:8712,
192.168.31.78:8713]
19:46:51.112 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [checkMagicCode,124] - Check magic code success.
19:46:51.112 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [checkVersion,137] - Check protocol version success.
19:46:51.114 [main] INFO c.w.r.z.ZookeeperRegistry - [lookup,94] - Lookup service success, hit provider cache. The providerList is [192.168.31.78:8712, 192.168.31
.78:8713]
19:46:51.114 [main] INFO c.w.r.z.ZookeeperRegistry - [watch,118] - Watch service success, Watching service is com.wang.service
.UserService?group=groupName&version=version1
19:46:51.114 [main] INFO c.w.r.z.ZookeeperRegistry - [lookup,108] - SelectedProvider is 192.168.31.78:8713
19:46:51.119 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [encode,54] - Encoding protocol.
19:46:51.119 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [decode,88] - Decoding protocol.
19:46:51.137 [Curator-SafeNotifyService-0] INFO c.w.r.z.ZookeeperRegistry - [lambda$0,120] - type: NODE_CREATED, oldData: null, newData:
ChildData{path='/ya-rpc-provider/com.wang.service.UserService?group=groupName&version=version1', stat=12884902262,12884902262,1668939897219,1668939897219,0,22,0,0,
0,2,12884902350
, data=[]}
19:46:51.139 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [checkMagicCode,124] - Check magic code success.
19:46:51.139 [main] INFO c.w.r.s.c.SocketRpcMessageCodec - [checkVersion,137] - Check protocol version success.
19:46:51.140 [main] INFO c.w.SocketClientMain2 - [main,15] - User1: User(userId=22080626, userName=happytsing) User2: User(userId=18160207, userName=toucher le port)
19:46:51.142 [Curator-SafeNotifyService-0] INFO c.w.r.z.CuratorUtils - [getChildrenNodes,78] - Get children success. The children are [192.168.31.78:8712,
192.168.31.78:8713]
Process finished with exit code 0
  
```

图 4-7 UserService 测试结果

如图 4-7 所示，RPC 调用 UserService 的方法成功，且成功返回 User 对象类型。

第五章 开发计划

本课题由作者独自设计并开发完成，在本课题的推进和实施过程中，针对复杂工程问题的执行指定了详尽的工程计划，具体内容如表 5-1 所示：

表 5-1 工程计划管控与执行计划

工程计划	详细内容	执行情况
项目需求分析	针对实际项目背景和使用场景，进行详细需求分析	按期完成
可行性研究和分析	针对需求分析和复杂工程问题的归纳，从技术、经济成本和社会因素等方面进行可行性分析	按期完成
技术知识学习	针对理论研究和项目开发所需的知识技能，进行针对性的学习，如 SPI 机制、反射、动态规划、注册中心等。	按期完成
竞品调研	调研市场上成熟的 RPC 框架，并学习其架构原理	按期完成
系统架构设计及实现	对 YA-RPC 项目框架设计，并使用代码实现详细设计中的工程项目，对复杂工程问题进行攻关	按期完成
系统单元测试	对实现的系统的各个功能进行白盒单元测试，确保功能单元的正常运行	按期完成
系统功能测试	对系统整体进行黑盒功能测试，确保各个功能单元的联动功能正常运行	按期完成