电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

# 分布式系统报告


题目 YA-RPC: Yet-Another RPC Framework

学科专业 计算机科学与技术

学　　号 202221080520

作者姓名 王夏兵

课程老师 薛瑞尼 罗嘉庆

# 目录

# 第一章 需求分析

## 1.1RPC简介

RPC (Remote Procedure Call)，即远程过程调用。顾名思义，它可以使进程调用网络中另一台机器上的函数就像在自己本地调用一样，而程序员不要管其中实现的细节。当调用远程过程时，调用环境将被挂起，相关参数信息通过网络传递给服务端，所需的过程将在服务端执行。当产生结果时在通过网络发送给调用者，这个过程就叫做远程过程调用[1]。

开发一个简单的应用，也许不会用到RPC，但是当应用出现业务增加以及流量增加等情况时，就需要根据不同的业务对应用进行拆分，分别部署在不同的机器上，这个时候RPC就变得尤为重要了[1]。

## 1.2系统要求

（1）编写一个简易RPC框架，YA-RPC；

（2）支持基本数据类型：int, float, string；

（3）支持 At-least-once 语义；

（4）使用YA-RPC编写一个demo程序，实现如下API：

①远程调用 float sum(float a, float b)；

②远程调用 string uppercase(str)；

③不少于2个客户端，1个服务端；

（5）开发语言不限；

[1] RPC使用场景参考来源知乎https://zhuanlan.zhihu.com/p/349263565

# 第二章 总体设计

## 2.1 整体框架

本系统实现的RPC框架如图2-1，图中展示了具体框架细节。其又大致分为两部分，第一部分是接口文件，也就是proto文件，用于定义具体的接口信息等。另一部分是实现调用的Stub。

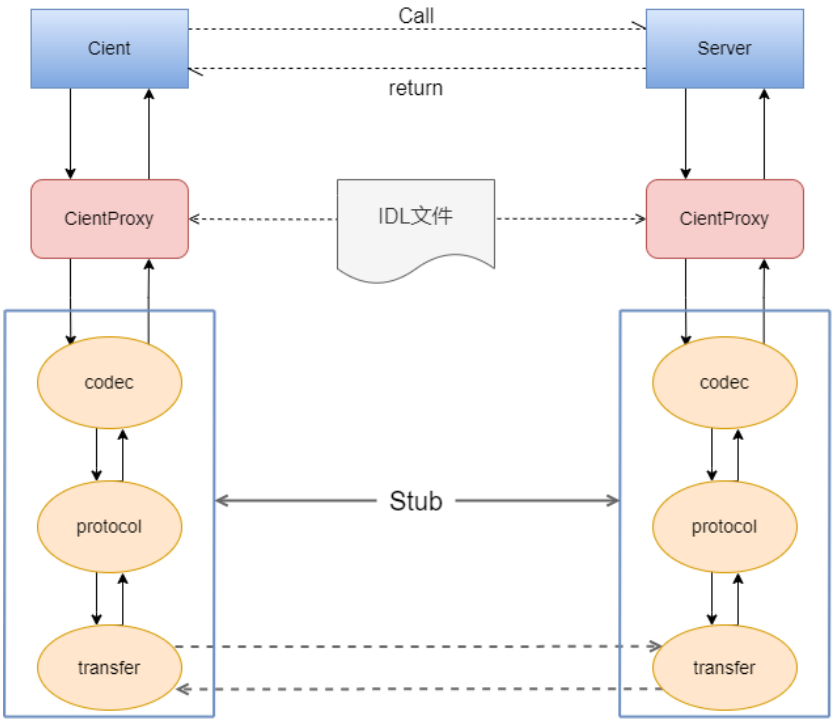


图2-1 RPC框架

## 2.2 主要流程分析

在整体的RPC框架中，codec层用于存储函数的调用信息，包括函数的参数、函数名以及返回值等信息；protocol层定义了具体的RPC协议，具体包括header与boby两部分，header主要记录了协议的版本号、消息类型等信息；网络传输的实现在transfer层，该部分使用TCP协议进行传输，并且需求的At-least-once也在该部分实现。

代理类包括CientProxy以及ServerProxy，代理类的主要作用是帮助调用者实现函数远程调用。在CientProxy中实现了数据的序列化、反序列化以及数据的传输等，此外在ServerProxy中实现了线程池的管理以及函数的注册等功能。

# 第三章 详细设计与实现

## 3.1代理类

代理类帮助调用者以及被调用者处理整个过程，使调用者远程调用函数像调用本地函数一样。

### 3.1.1 CientProxy

（1）动态代理

动态代理的基本过程是由访问者访问代理类，再由代理类去访问源目标。与静态代理不同的是，静态代理需要实现具体的接口，而动态代理则不需要实例化具体的接口。图3-1展示了动态代理的大致流程。


图3-1 动态代理流程

（2）功能实现

本系统利用Java的动态代理特性，对需要调用的接口进行动态代理，代理类继承InvocationHandler。当调用者调用所需方法时，由该类的invoke函数进行处理，并由该函数返回给调用者一个返回值。

图3-2展示了获得一个类代理的获取方法，此后，关于该类的访问都将由invoke进行处理。

```
public <T> T getProxyService(Class<T> c)
{
    return (T) Proxy.newProxyInstance(c.getClassLoader(), new Class<?>[]{c}, this);
}
```

图3-2 代理类获取

### 3.1.2 ServerProxy

在SeverProxy类中实现了线程池以及本地函数的注册。Java中的线程池核心实现类是ThreadPoolExecutor，其原理如图3-3。线程池控制了当前运行的线程数量以及线程等待数量。其线程池管理的基本思想是对一个刚到的任务进行判断，如果当前线程数小于核心线程数那么直接运行该线程，否则判断阻塞队列是否满，如果未满则放入阻塞队列。当阻塞队列满时将任务与最大线程数进行

比较，未满则放进非核心线程，否则直接放弃该任务。

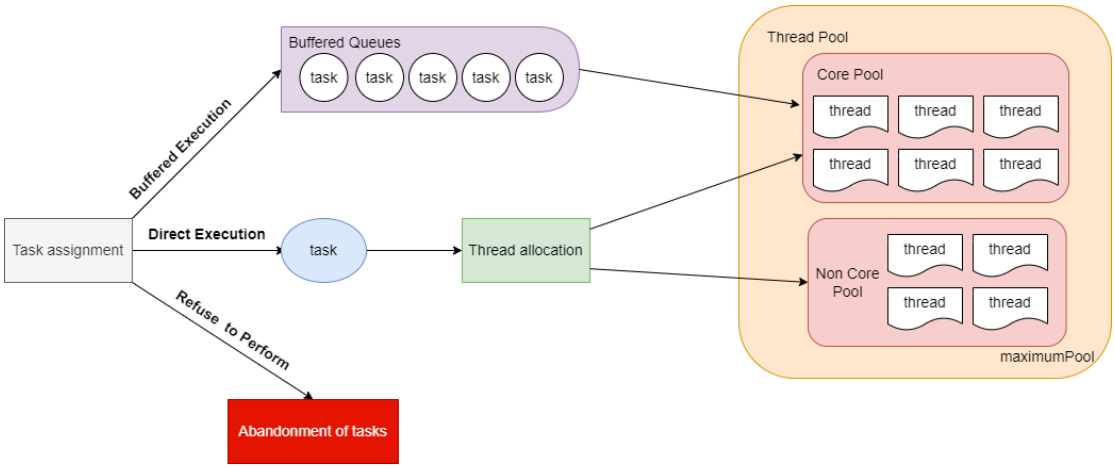


图3-3 ThreadPoolExecutor原理

另外，有关函数的注册放在了SeverProxy类中，如图3-4，其将函数存放在了一个名为registeredThreads哈希表中，该函数存放对应的函数名及其对象。

```
public void Register(Object task)
{
    registeredThreads.put(task.getClass().getInterfaces()[0].getName(), task);
}
```

图3-4注册函数

### 3.1.3序列化与反序列化

本系统所有需要序列化的类通过实现Serializable接口来实现，利用流对象将请求数据序列化为字节流，其过程如图3-5。

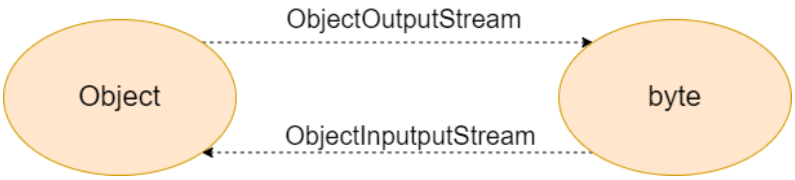


图3-5 对象的转化

序列化的代码实现细节如图3-6，反序列化的实现细节类似，这里不再一一赘述。

```
//创建字节流数组
ByteArrayOutputStream requestByteData=new ByteArrayOutputStream();
ObjectOutputStream requestObjectData=new ObjectOutputStream(requestByteData);
//请求对象转化为字节流
requestObjectData.writeObject(rpcRequestCodec);
byte[] bytes=requestByteData.toByteArray();
```

图3-6 序列化

### 3.1.4 TCP通信

本系统的数据传输使用TCP协议，另外在该部分实现At-least-once语义。如图3-7，通过定义最大重传次数MAX_RETRANSMISSION_TIMES，在数据发送失败时让数据重新发送以确保客户端能接收到数据。

```
for(int i=0;i<this.MAX_RETRANSMISSION_TIMES;i++)
{
    try
    {
        rpcReponseProtocol = rpcCientTransfer.sendData(rpcRequestProtocol);
        break;
    }
    catch(IOException e)
    {
        System.out.printf(format: "未收到服务器ACK, 尝试第%d重传数据...\n",i+1);
        Thread.sleep(millis: 1000);
    }
}
```

图3-7 At-least-once

## 3.2 RPC实体类

### 3.2.1 codec层

codec层的主要作用是存储方法调用的具体信息，如图3-8，在请求方一端，该类包含了接口名、方法名、参数、参数类型信息。

```
public class RpcRequestCodec implements Serializable{
    private String interfaceName;
    private String functionName;
    private Object[] params;
    private Class<?>[] paramsTypes;
}
```

图3-8 codec请求类

在接受请求时，只存储返回值信息，如图3-9。

```
public class RpcResponseCodec implements Serializable{
    private Object retObject;
}
```

图3-9 codec返回类

### 3.2.2 protocol层

protocol层定义了发送的header和body，header包含协议的版本号、状态等信息，如图3-10。在发送方以及接收方处理数据时都要验证所处理的对象的header是否符合自己的预期，这部分在后期还可以继续优化使其更加完善。

```
public class RpcProtocolHeader implements Serializable{
    private Integer magic; //协议版本号
    private RpcProtocolStatus status;//标识状态
    private String messageType;//消息类型
    private String messageEncoding;//消息编码
}
```

图3-10 RPC协议头

另外，本系统定义了几种状态，分别表示正常、客户端超时、服务器未找到等，如图3-11。

```
public enum RpcProtocolStatus {
    OK,
    CIENT_TIME_OUT,
    SERVER_TIME_OUT,
    SERVER_NOT_FOUND,
    CIENT_ERROR,
    SERVER_ERROR
}
```

图3-11 RPC协议头状态

# 第四章 测试

## 4.1 开发环境

表4-1展示了本系统的开发环境。

表4-1

| 配置项 | 配置参数 |
| --- | --- |
| 操作系统 | Ubuntu 20.04.1 LTS |
| CPU型号 | Intel® Core™ i5-4590 CPU @ 3.30GHz |
| 内存 | 8G |
| 程序运行环境 | Java 11.0.16 |

## 4.2 系统测试

图4-1为服务端分别受到来自两个客户端的调用信息时的log信息。

```
WangXiaBing@WangXiaBing:~/programs/java/ya_rpc$  /usr/bin/env /usr/lib/j
vm/java-11-openjdk-amd64/bin/java @/tmp/cp_7hq8q1caiz9lhtvd2eew8om9n.arg
file com.wxb.server.Server
receive data from:/127.0.0.1
send result:10.24
receive data from:/127.0.0.1
send result:WXB TEST
receive data from:/127.0.0.1
send result:3.33
receive data from:/127.0.0.1
send result:WXB TEST
```

图4-1 Server Log

图4-2为第一个客户端的log信息，其进行浮点数的相加，从远端返回结果。

```
WangXiaBing@WangXiaBing:~/programs/java/ya_rpc$  /usr/bin/env /usr/lib/j
vm/java-11-openjdk-amd64/bin/java @/tmp/cp_7hq8q1caiz9lhtvd2eew8om9n.arg
file com.wxb.cient.Cient
please input a and b:
10 0.24
the result is 10.240
please input a and b:
1.13 2.2
the result is 3.330
please input a and b:
```

图4-2 Cient Log

图4-3为第二个客户端的log信息，其将所需字符串转化为大写，从远端返回结果。

```
WangXiaBing@WangXiaBing:~/programs/java/ya_rpc$  /usr/bin/env /usr/lib/j
vm/java-11-openjdk-amd64/bin/java @/tmp/cp_7hq8q1caiz9lhtvd2eew8om9n.arg
file com.wxb.cient.Cient2
please input str:
wxb test
the result is WXB TEST
please input str:
wxb TEST
the result is WXB TEST
please input str:
```

图4-3 Cient Log

图4-4是系统重传的测试，确保At-least-once。

```
WangXiaBing@WangXiaBing:~/programs/java/ya_rpc$  cd /home/WangXiaBing/p
rograms/java/ya_rpc ; /usr/bin/env /usr/lib/jvm/java-11-openjdk-amd64/b
in/java @/tmp/cp_7hq8q1caiz9lhtvd2eew8om9n.argfile com.wxb.cient.Cient
please input a and b:
2.2 3.1
未收到服务器数据，尝试第1重传数据...
未收到服务器数据，尝试第2重传数据...
未收到服务器数据，尝试第3重传数据...
未收到服务器数据，尝试第4重传数据...
the result is 5.300
please input a and b:
```

图4-3 重传测试

# 附录

本次实验由本人独立完成，具体工程代码详见附件。

# 参考文献

[1] Birrell A D, Nelson B J. Implementing remote procedure calls[J]. ACM Transactions on Computer Systems (TOCS), 1984, 2(1): 39-59.