

gRPC

无03 王与进

目录

gRPC

目录

前言

Client-Server model ★

IP Address

Port

gRPC概况

gRPC安装

C++

Csharp

gRPC服务 ★

gRPC使用 ★

proto

Server

Client

参考与荐读

前言

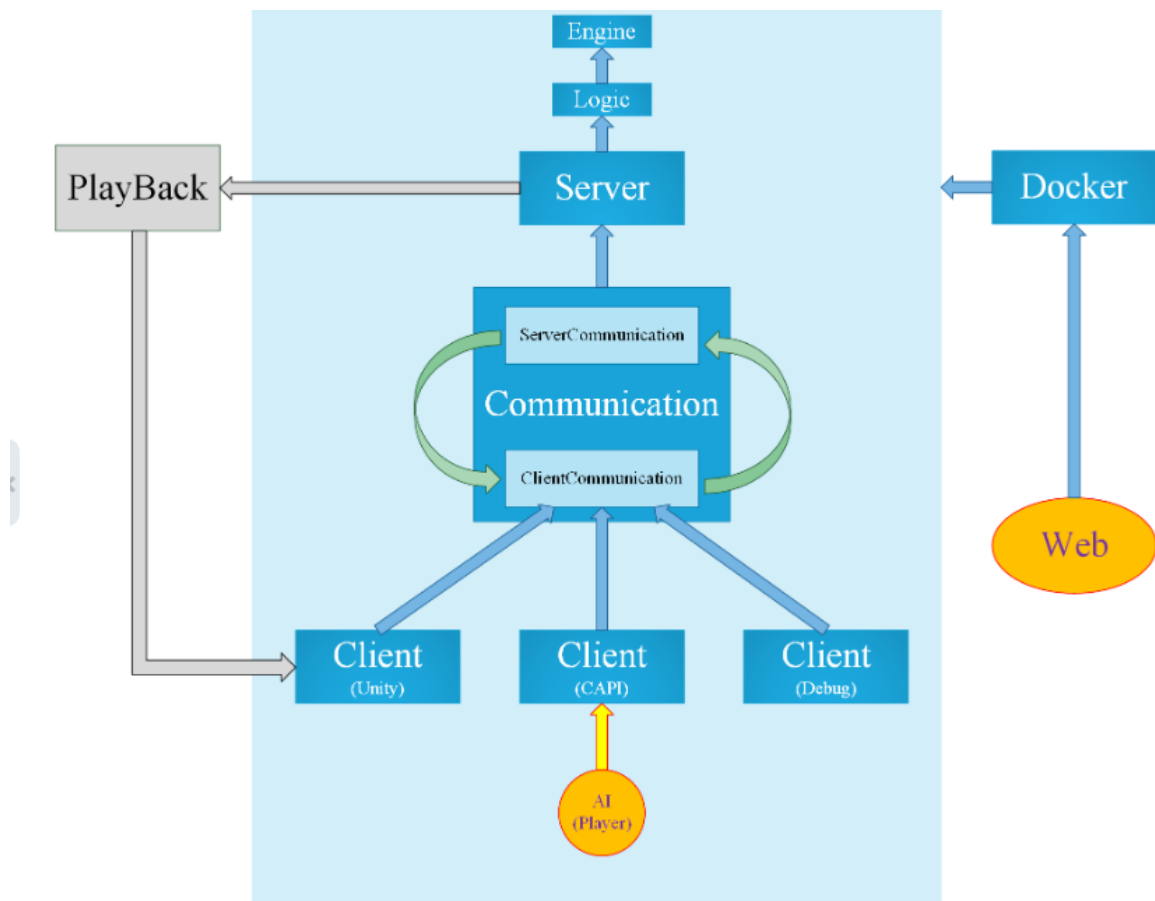
在介绍gRPC之前，我们需要先介绍几个在通信中需要用到的概念。

Client-Server model ★

Client-Server结构是一种经典的通信模型。它通常采取两层结构：

- 服务器（Server）负责数据的处理。它有以下特征：
 - 等待来自客户端的请求
 - 处理请求并传回结果
- 客户端（Client）负责完成与用户的交互任务。它有以下特征：
 - 发送请求
 - 等待直到收到响应

[THUA15](#)就是一个应用了Client-Server model的典型实例：



在游戏中，玩家通过在Client端编写C++代码来制定游戏策略，而Server端由Csharp语言写成，用于分析处理游戏逻辑。编译生成的Client端可执行文件将向Server端发送请求，请求处理完毕后Server端再向Client端发送处理后的结果，这样Client端就可以接受到游戏实况，以供下一步决策。

IP Address

IP Address(Internet Protocol address，网际协议地址)，是网际协议中用于标识发送或接受数据报的设备的一串数字。

当设备连接网络后，设备将被分配一个IP地址，对于一个具体的设备而言，IP地址是独一无二的。IP地址有两个主要的功能：**标识主机**（用户在互联网上可以识别）和**网络寻址**（允许计算机通过互联网发送和接受数据）。

常见的IP地址分为IPv4和IPv6两大类：

- IPv4：32位长，通常书写时以四组十进制数字组成，并以点分割，例如：`172.16.254.1`。
- IPv6：128位长，通常书写时以八组十六进制数字组成，并以冒号分割，例如：`2001:db8:0:1234:0:567:8:1`。

我们可以使用如下方法查询本机的IP地址：

- windows： `ipconfig`
- linux： `ifconfig`（可能需要使用 `sudo apt-get install net-tools` 进行安装）

一个特殊的IP地址： `127.0.0.1`

尽管现在有大量可用的IP地址，但为了防止编程冲突的特定目的，刻意保留一些地址，甚至是地址范围是很方便的。

`127.0.0.1`就是其中一个。它表示的是**主机环回地址**，表示的是任何数据包都不应该离开计算机，计算机本身即为接收者。

当我们需要在本地测试一些网站服务，或者只想在本地设备上运行只有本地设备可以访问的服务，就可以使用 `127.0.0.1`。

Port

Port(端口)在电脑网络中是一种经过软件创建的服务，在一个电脑的操作系统中扮演通信的端点。

什么意思呢？利用IP地址，可以实现不同计算机之间的通信。但实际上，计算机中是运行着多个进程的——当不同的信息被传入计算机后，计算机需要一种手段来区分信息的接收者，以将不同进程的处理结果正确地发送给接收者。

这个时候，端口就派上了用场。如果我们在通信时不仅指定IP地址，而且指定端口，计算机就可以正确地将不同的请求交给正确的进程处理。

特定的服务一般对应于特定的端口，详见[端口列表](#)。

我们可以使用如下方法查看本机的端口使用情况：

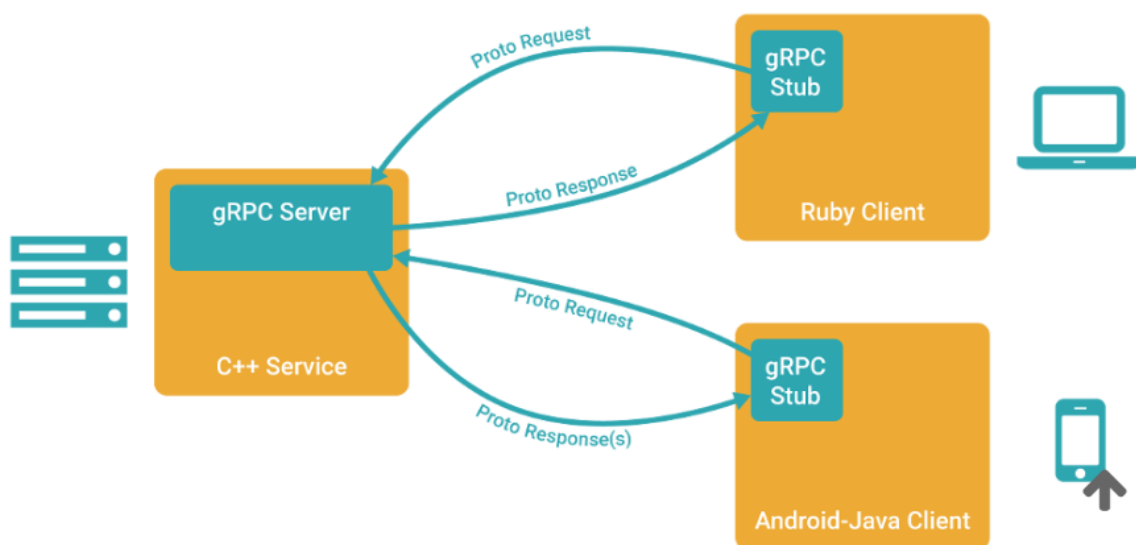
- windows: `netstat -ano | findstr "<port>"`
- linux: `netstat -tunlp | grep <port>` 或 `lsof -i:<port>`

gRPC概况

gRPC的全称是gRPC Remote Procedure Calls。其中“Remote Procedure Calls”翻译为“远程过程调用”。“远程过程调用”指的是客户端（Client）可以像调用本地对象一样直接调用服务端（Server）应用的方法。具体过程如下：

1. 定义若干服务（Service），指定其能够被远程调用的方法（包含参数和返回类型）。这些定义都写在 `.proto` 文件里。
2. 在服务端（Server）实现这个接口（内部处理逻辑），并运行gRPC服务器，来处理客户端的调用。
3. 在客户端（Client）建立一个存根（stub），提供与服务端相同的方法。

下面的图形象地展示了gRPC的使用过程：



这样一来，用户在使用gRPC构建的应用程序时，不需要关心调用方法的内部逻辑（被封装在Server中），只需要调用Client端提供的方法向Server端提供请求，等待Server端返回结果即可——看上去就和在Client端本地调用方法一样。

gRPC有诸多优点：

- 速度快：gRPC使用protobuf进行Server/Client之间数据的序列化和反序列化，保证了通信的高效。
- 跨语言：构建Server端和Client端程序的源语言无需一致。
- 跨平台：Server端和Client端的平台无需一致。

我们仍然THUIA5为例，阐述gRPC在构建具体项目中的意义（注：虽然THUIA5中使用的通信方法并非gRPC，但gRPC对我们的设计仍然有着重大的借鉴意义）：

- Server端需要实现复杂的游戏逻辑，而且需要支持Unity，如果使用C++语言可能会导致开发效率太低，因此需要使用Csharp语言进行开发。
- Client端需要提供选手接口供选手编写AI代码，因此需要使用C++语言开发。

两者使用语言不同，如何使得两者建立联系？我们可以使用gRPC的思路：

- 在 .proto 文件中定义选手可以调用的游戏方法（如人物操作和获取物品信息）。
- 在Server端实现这些接口的内部逻辑。
- 在Client端提供用户需要直接调用的方法，而无需关心其具体实现。

于是我们就实现了Server和Client的解耦。在此基础上，我们甚至可以提供不同种类语言的用户接口——你可以使用Python、Java或其它语言来编写你的游戏策略。

gRPC安装

C++

安装gRPC C++相关的库需要手动编译其源码：

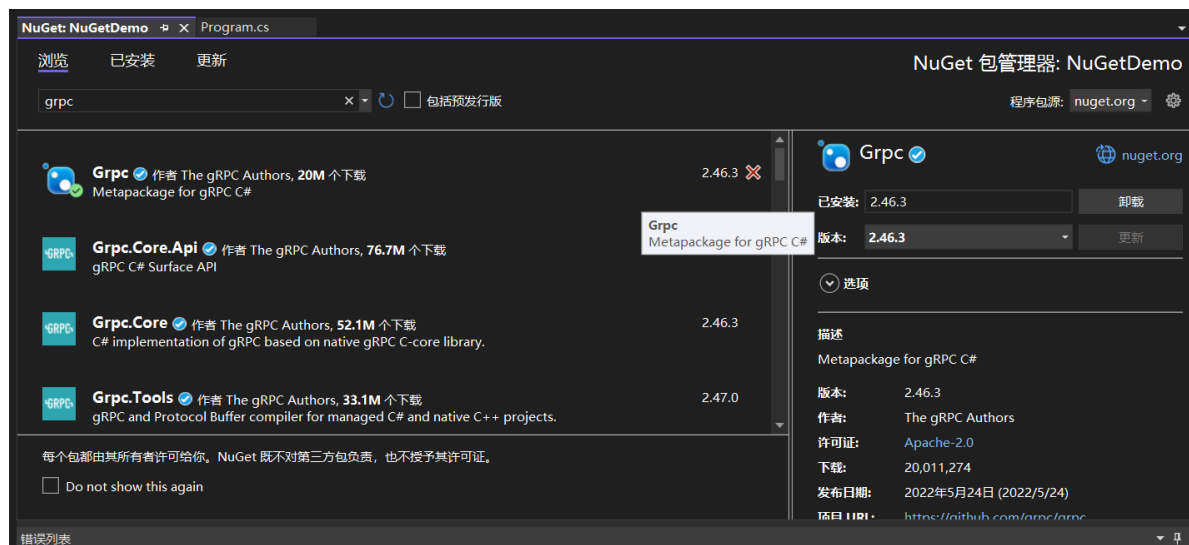
```
$ git clone -b v1.46.3 --depth 1 --shallow-submodules
https://github.com/grpc/grpc
# 如果网络不佳，可以将网址换为 https://gitee.com/mirrors/grpc.git
$ cd grpc
$ git submodule update --init --recursive
# 在grpc的原文档中没有submodule该步，但笔者实测，如果没有这一步grpc将无法安装。
$ mkdir -p cmake/build
$ pushd cmake/build
$ cmake -DgRPC_INSTALL=ON \
        -DgRPC_BUILD_TESTS=OFF \
        ../../..
$ make -j
$ make install # 或 sudo make install
$ popd
```

需要指出的是，由于网络等问题，`git submodule update --init --recursive`一步往往无法正常运行。为此可以点击[此处](#)下载 `third_party.tar.gz`，并将 `git submodule..` 一步替换为以下操作：

```
$ rm -rf third_party
$ mv <tar_gz_path> .
$ tar -zxvf third_party.tar.gz
$ cd ..
```

Csharp

Csharp中，我们可以使用NuGet程序包安装gRPC库（图中第一项）。



gRPC服务★

grpc默认使用protobuf作为接口定义语言。定义方式见下列：

```
// The greeter service definition.
service Greeter {
    // Sends a greeting
    rpc SayHello (HelloRequest) returns (HelloReply);
}

// The request message containing the user's name.
message HelloRequest {
    string name = 1;
}

// The response message containing the greetings
message HelloReply {
    string reply = 1;
}
```

定义服务使用了 `service` 和 `rpc` 关键字。粗略地来讲，在本例中，gRPC服务接受一条含有name字段的HelloRequest message，发送给服务端处理后，返回一条含有reply字段的HelloRequest message。

gRPC可以定义以下4种服务：

- 单一RPC (Unary RPCs)，客户端向服务器发送一个请求，并得到一个响应，就像一个正常的函数调用。简单来讲就是一个请求对象对应一个返回对象。

```
rpc SayHello(HelloRequest) returns (HelloResponse);
```

- 服务器流式RPC (Server streaming RPCs)，客户端向服务器发送请求，并获得一个流来读回一连串的消息。客户端从返回的流中读取信息，直到没有更多的信息。gRPC保证在单个RPC调用中的信息排序。简单来讲就是发送一个请求对象，服务端可以传回多个结果对象。

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
```

- 客户端流式RPC (Client streaming RPCs) , 客户端写了一串消息并将它们发送给服务器, 同样使用一个提供的流。一旦客户端完成了消息的写入, 它就等待服务器读取它们并返回其响应。gRPC 再次保证了单个RPC调用中的消息排序。简单来讲就是**客户端传入多个请求对象, 服务端返回一个响应结果**。

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
```

- 双向流RPC (Bidirectional streaming RPCs) , 双方使用读写流发送一连串的消息。这两个流独立运行, 因此客户和服务端可以按照他们喜欢的顺序进行读写: 例如, 服务器可以等待收到所有客户的消息, 然后再写它的响应, 或者它可以交替地读一个消息, 然后再写一个消息, 或者其他一些读和写的组合。每个流中的消息的顺序被保留下来。简单来讲就是**结合客户端流式rpc和服务端流式rpc, 可以传入多个对象, 返回多个响应对象**。

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
```

接下来我们将结合一些实例进一步了解它们的使用方法和区别。

gRPC使用 ★

在本例中, 我们将使用Csharp语言实现一个简单的Client-Server模型——Client端提供两个数和一个运算符, 而Server端则进行具体的运算过程并将计算结果返回给Client端。

proto

我们不妨考虑以下服务场景:

- 客户端发送一个包含两个操作数和一个运算符的元组, 服务端返回一个结果: 该场景符合单一RPC。
- 客户端发送一个包含两个操作数和一个运算符的元组, 服务端返回计算结果, 并将该结果重复多次: 该场景符合服务器流式RPC。
- 客户端发送若干个包含两个操作数和一个运算符的元组, 服务端返回计算结果之和: 该场景符合客户端流式RPC。
- 客户端发送若干个包含两个操作数和一个运算符的元组, 服务端分别返回每一对元组的计算结果之和: 该场景符合双向流RPC。

我们需要在 `Message.proto` 文件中定义需要的服务:

```
syntax = "proto3";

package hello;

enum Operator {
    NONE_OP = 0;
    ADD = 1;
    SUB = 2;
    MUL = 3;
}

message Operand {
    int32 op1 = 1;
```

```

    int32 op2 = 2;
    Operator opr = 3;
}

message Result {
    int32 val = 1;
}

service Calculator {
    // Unary
    rpc UnaryCall (Operand) returns (Result);

    // Server streaming
    rpc StreamingFromServer (Operand) returns (stream Result);

    // Client streaming
    rpc StreamingFromClient (stream Operand) returns (Result);

    // Bi-directional streaming
    rpc StreamingBothways (stream Operand) returns (stream Result);
}

```

之后就可以使用该文件生成对应的CSharp文件以供使用。

Server

Server端有两个任务：

- 实现我们服务定义的生成的服务接口：做我们的服务的实际的“工作”。
- 运行一个 gRPC 服务器，监听来自客户端的请求并返回服务的响应。

为了实现这些目的，我们需要在Server端定义一个 `CalculatorImpl` 类，并继承 `calculator.CalculatorBase` 类，以实现所有的服务方法。

对于 `calculator.CalculatorBase` 类的解释：Base class for server-side implementations of Calculator。可见它是专供Server端使用的一个基类。

```

class CalculatorImpl : calculator.CalculatorBase
{
    public override Task<Result> UnaryCall(Operand operand,
ServerCallContext context)
    {
        var res = new Result();
        switch (operand.Opr)
        {
            case Operator.Add:
                res.Val = operand.Op1 + operand.Op2;
                break;
            case Operator.Sub:
                res.Val = operand.Op1 - operand.Op2;
                break;
            case Operator.Mul:
                res.Val = operand.Op1 * operand.Op2;
                break;
            default:

```

```

        break;
    }
    return Task.FromResult(res);
}

public override async Task StreamingFromServer(Operand operand,
IServerStreamWriter<Result> result_stream, ServerCallContext context)
{
    var res = new Result();
    switch (operand.Opr)
    {
        case Operator.Add:
            res.Val = operand.Op1 + operand.Op2;
            break;
        case Operator.Sub:
            res.Val = operand.Op1 - operand.Op2;
            break;
        case Operator.Mul:
            res.Val = operand.Op1 * operand.Op2;
            break;
        default:
            break;
    }
    for (var i = 0; i < 3; i++)
    {
        await result_stream.WriteAsync(res);
    }
}

public override async Task<Result>
StreamingFromClient(IAsyncStreamReader<Operand> operand_stream,
ServerCallContext context)
{
    var res = new Result();
    while (await operand_stream.MoveNext())
    {
        var operand = operand_stream.Current;
        switch (operand.Opr)
        {
            case Operator.Add:
                res.Val += operand.Op1 + operand.Op2;
                break;
            case Operator.Sub:
                res.Val += operand.Op1 - operand.Op2;
                break;
            case Operator.Mul:
                res.Val += operand.Op1 * operand.Op2;
                break;
            default:
                break;
        }
    }
    return res;
}

```



```

        public override async Task StreamingBothWays(IAsyncStreamReader<Operand>
operand_stream, IServerStreamWriter<Result> result_stream, ServerCallContext
context)
        {
            while (await operand_stream.MoveNext())
            {
                Operand operand = operand_stream.Current;
                var res = new Result();
                switch (operand.Opr)
                {
                    case Operator.Add:
                        res.Val = operand.Op1 + operand.Op2;
                        break;
                    case Operator.Sub:
                        res.Val = operand.Op1 - operand.Op2;
                        break;
                    case Operator.Mul:
                        res.Val = operand.Op1 * operand.Op2;
                        break;
                    default:
                        break;
                }
                await result_stream.WriteAsync(res);
            }
        }
    }
}

```

我们来看上方的代码的特点：

- 为了允许任务的异步执行，我们在返回值中使用 `Task` 关键字。
- 在服务器流式RPC中，我们需要使用异步方法 `WriteAsync` 将服务器的响应写入异步流 `IServerStreamWriter` 中。
- 在客户端流式RPC中，我们需要使用异步流 `IAsyncStreamReader` 逐个读出请求并进行运算。
- 在双向流式RPC中，我们需要同时使用 `IAsyncStreamReader` 和 `IServerStreamWriter`。

而启用gRPC服务器的代码如下：

```

public static void Main()
{
    try
    {
        // 禁止复用端口!!! (SoReuseport 置为 0)
        Grpc.Core.Server server = new Grpc.Core.Server(new[] { new
ChannelOption(ChannelOptions.SoReuseport, 0) })
        {
            Services = { Calculator.BindService(new CalculatorImpl()) },
            Ports = { new ServerPort("127.0.0.1", 8888,
ServerCredentials.Insecure) }
        }; // 建立监听特定IP地址和端口Server的模板代码
        server.Start();
        Console.WriteLine("Server begins to listen!");
        Console.WriteLine("Press any key to stop the server...");
        Console.ReadKey();
        Console.WriteLine("Server end!");
        server.ShutdownAsync().Wait();
    }
}

```

```

    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}

```

我们总结一下创建客户端的步骤：

1. 创建 `Grpc.Core.Server` 的一个实例。
2. 创建我们的服务实现类 `CalculatorImpl` 的一个实例。
3. 通过在 `services` 集合中添加服务的定义注册我们的服务实现。
4. 指定想要接受客户端请求的地址和监听的端口。通过往 `Ports` 集合中添加 `ServerPort` 即可完成。
5. 在服务器实例上调用 `start` 为我们的服务启动一个 RPC 服务器。

Client

首先，我们需要建立一个Client对象：

```

Channel channel = new Channel("127.0.0.1:8888", ChannelCredentials.Insecure);
var client = new Calculator.CalculatorClient(channel); // 建立一个连接到特定host的
client
// ... client 的调用操作

```

在调用单一RPC服务时，我们像调用本地方法那样调用远程方法（`UnaryCall`），如果RPC成功完成，则返回响应值。

```

// case 1: unary call (单一RPC)
Console.WriteLine("case 1:");
var unaryCall = client.UnaryCall(operand0); //
var unaryCallVal = unaryCall.Val;
Console.WriteLine(unaryCallVal);

```

在调用服务器流式RPC服务时，由于得到的响应是流式的，所以我们需要使用 `MoveNext` 方法逐个读取其值。

```

// case 2: streaming from server (服务器流式RPC)
Console.WriteLine("case 2:");
var streamingFromServer = client.StreamingFromServer(operand0);
while(await streamingFromServer.ResponseStream.MoveNext())
{
    var streamingFromServerVal =
streamingFromServer.ResponseStream.Current.Val;
    Console.WriteLine(streamingFromServerVal);
}

```

在调用客户端流式RPC服务时，我们需要使用 `WriteAsync` 方法逐个写入请求值，最终使用 `CompleteAsync` 方法表示不再请求。

```

// case 3: streaming from client (客户端流式RPC)
Console.WriteLine("case 3:");

```

```

var streamingFromClient = client.StreamingFromClient();
Tuple<int, int, Operator>[] tups = { new(1, 1, Operator.Add), new(5, 6,
Operator.Mul), new(3, 4, Operator.Sub), new(0, 0, Operator.NoneOp) };
foreach (var tup in tups)
{
    Operand operand = new Operand();
    operand.Op1 = tup.Item1;
    operand.Op2 = tup.Item2;
    operand.Opr = tup.Item3;
    await streamingFromClient.RequestStream.WriteAsync(operand);
}
await streamingFromClient.RequestStream.CompleteAsync();
var streamingFromClientVal = streamingFromClient.ResponseAsync.Result.Val;
Console.WriteLine(streamingFromClientVal);

```

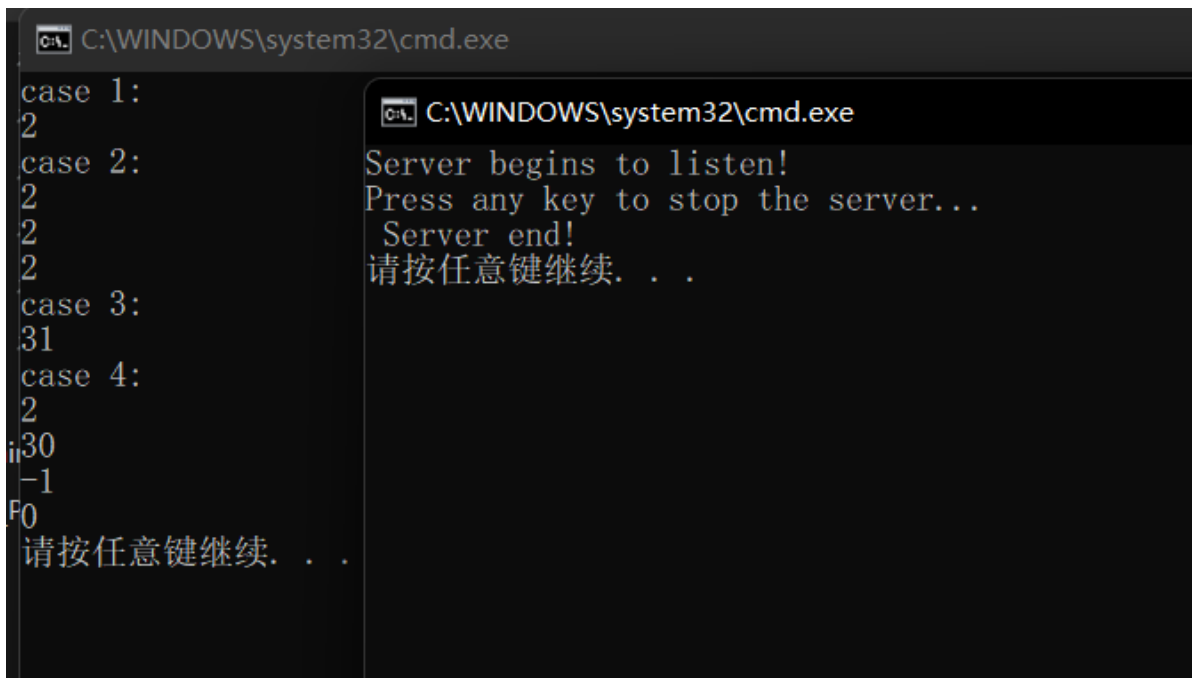
在调用双向流RPC服务时，我们将请求写入 `RequestStream`，使用 `ResponseStream` 获取响应。两者是相互独立的。

```

// case 4: streaming both ways (双向流RPC)
Console.WriteLine("case 4:");
var streamingBothWays = client.StreamingBothWays();
foreach (var tup in tups)
{
    Operand operand = new Operand();
    operand.Op1 = tup.Item1;
    operand.Op2 = tup.Item2;
    operand.Opr = tup.Item3;
    _ = streamingBothWays.RequestStream.WriteAsync(operand);
    if (!await streamingBothWays.ResponseStream.MoveNext())
    {
        break;
    }
    var streamingBothWaysVal = streamingBothWays.ResponseStream.Current.Val;
    Console.WriteLine(streamingBothWaysVal);
}

```

运行结果如下：



The image shows two overlapping Windows command prompt windows. The background window has a title bar 'C:\WINDOWS\system32\cmd.exe' and contains the following text: 'case 1:', '2', 'case 2:', '2', '2', '2', 'case 3:', '31', 'case 4:', '2', '30', '1', 'F0', and '请按任意键继续. . .'. The foreground window, slightly offset to the right, also has a title bar 'C:\WINDOWS\system32\cmd.exe' and contains the text: 'Server begins to listen!', 'Press any key to stop the server...', 'Server end!', and '请按任意键继续. . .'. Both windows have a black background with white text.

参考与荐读

由于时间所限，有很多有趣的内容我们没有涉及：

- 计算机网络模型
- RPC的生命周期
- 在gRPC中使用安全认证和通讯协议
- ...

略过上述内容不会对我们的教学产生太大影响，感兴趣的同学可以参考以下文档和资源：

- [计算机网络——自顶向下方法](#)
- [Stanford CS144](#)
- [gRPC 官方文档](#)