gRPC（应用于通信）

1. **gRPC应用场景**
   1. 微服务

假设开发有数千个微服务的系统，我们不想花大部分时间编写代码实现各个微服务之间的通信功能，我们需要某种框架，允许开发人员专注于实现其服务的核心逻辑，而将其他一切交给框架来处理，此框架就是gRPC。

* 1. 多语言环境
  2. 点对点实时通信
  3. 网络受限的环境，例如移动应用程序（轻量级的消息格式）

1. **RPC （Remote Procedure Call，远程过程调用）**

RPC是一种协议，不需要了解底层网络技术就可以通过网络从远程计算机程序上请求服务

1. **gRPC**

gRPC允许你为RPC定义请求和响应，然后gRPC会帮你处理一切剩余问题

1. **gRPC支持HTTP2，双向传输、多路复用、认证等**

|  |  |
| --- | --- |
| **HTTP/2和HTTP/1.1的区别：** | |
| HTTP/2 | HTTP/1.1 |
| HTTP/2是二进制协议 | HTTP/1.1是文本 |
| Header在HTTP/2中被压缩 | 而在HTTP/1.1中是纯文本 |
| HTTP/2允许多路复用 | 而HTTP/1.1不允许 |
| 我们可以发送多个请求和响应在HTTP/2的单个连接中 | 而HTTP/1.1只允许发送一个 |
| 服务端推送在HTTP/2中是可能的 | 而在HTTP/1.1中是不可能的 |

1. gRPC基于http/2构建，可以组成二进制帧，传输效率几乎是http/1的两倍，执行效率高和开发语言无关，并且可以很简单地插入身份认证、负载均衡、日志和监控等功能
2. **gRPC有四种类型**
   1. 一元（unary）

客户端发送单一请求消息，服务器恢复单一响应

* 1. 客户端流（client stream）

客户端发送多条消息的流，它期待服务器发回单一响应

* 1. 服务器流（server stream）

客户端仅发送一条请求消息，期待服务器恢复多条消息流

* 1. 双向流（bidirectional stream）

客户端和服务器并行发送和接受多条消息，且具有任意顺序，非常灵活且无阻塞，没有任何一方等待响应

1. **gRPC常用的包**
   1. metadata：定义了grpc所支持的元数据结构
   2. credentials：
   3. codes：定义了grpc使用的标准错误码，可通用
      1. codes.InvalidArgument //参数错误
      2. codes.Internal //服务器内部错误
      3. codes.Unkown //未知错误
   4. status：rpc的状态码子包，可以返回一些常见错误
      1. status.FromError(err)
      2. status.Errorf()

func Errorf(c codes.Code, format string, a ...interface{}) error {

    return Error(c, fmt.Sprintf(format, a...))

}

1. 将proto.Message类型存入map应该先对其进行深刻复制（copier.Copy()）：

func deepCopy(laptop \*pb.Laptop) (\*pb.Laptop, error) {

    other := &pb.Laptop{}

    err := copier.Copy(other, laptop)

    if err != nil {

        return nil, fmt.Errorf("cannot copy laptop data : %w", err)

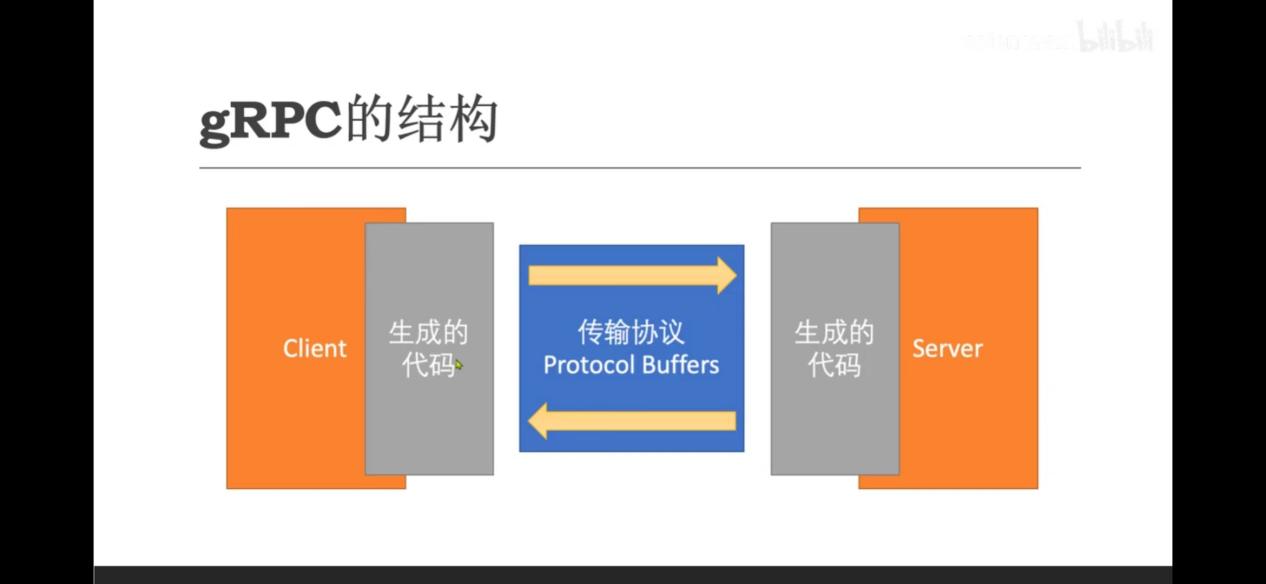
    }

    return other, nil

}

**gRPC部分**

1. gRPC的结构



1. **gRPC的设计步骤**

定义消息->生成代码->开发

1. **gRPC的生命周期**



1. **gRPC和REST比较**



然而大多数浏览器都支持rest，而对gRPC的支持是有限的。因为gRPC需要浏览器带有代理层的gRPC-web在HTTP/1和HTTP/2之间进行转换。微服务是最适用于gRPC的场景。在网络受限的地方也更好用，因为它传递的消息格式更加轻量。

1. **gRPC反射（**用Evans客户端可玩转gRPC的反射）

gRPC反射是服务器的可选扩展，用于帮助客户端构建请求，而无需事先生成存根，这对于客户端在实际实施之前探索gRPC API非常有用。

在golang服务器中使用：

    先导入反射包："google.golang/grpc/reflection"

    然后调用反射注册:reflection.Register(grpcServer)

1. **gRPC拦截器**

gRPC拦截器（用来对用户进行身份验证和授权，类似于一个中间件功能，可以在客户端和服务器=端添加）

服务端拦截器是gRPC服务器在到达实际RPC方法之前调用的函数，可以用于多种用途，例如：日志记录、跟踪、速率限制、身份验证、授权

客户端拦截器是gRPC客户端在调用实际RPC之前调用的函数

（1）服务端拦截器

    使用JSON Web令牌（JWT）授权访问我们的gRPC API

    只有具有某些特定角色的用户才能调用我们服务器上的特定API

（2）客户端拦截器

    登录用户并将JWT附加到请求中，然后再调用gRPC API

    对于服务端有两种类型的拦截器：一种用于一元RPC，另一种用于流RPC

为使用JWT我们需要使用到jwt-go库

1. **用TLS加密gRPC（对于加密，有三种gRPC）**
   1. insecure

不安全的连接，数据都不加密，不能用于生产

* 1. server-side TLS

由服务端TLS保护的连接。所有的数据都是加密的，但只有服务器需要向客户端提供其TLS证书，如果服务器不关心哪个客户端正在调用它的API，可以使用这种类型的连接。

* 1. mutual TLS

由相互TLS保护的连接。当服务器还需要验证谁在调用它的服务时，使用这一种类型。这种情况下，客户端和服务器都必须向对方提供它们的TLS证书。

1. **gRPC负载均衡**

大规模gRPC部署通常具有许多相同的后端服务器和许多客户端，负载均衡用于在可用服务器之间最佳地分配来自客户端的负载。

gRPC的负载均衡有两个主要选项：服务器端和客户端，决定使用哪一个是主要的架构选择。在服务端负载均衡中，客户端向负载均衡器或代理（例如Nginx或Envoy）发出RPC。负载均衡器将RPC调用分发到可用的后端服务器之一。它还跟踪在每个服务器上的负载并实现公平分配负载算法，客户端自己不知道后端服务器。在客户端负载均衡中，客户端知道多个后端服务器，并为每个RPC选择一个使用，通常后端服务器将自己注册到服务发现基础设施，（例如consul或etcd），然后客户端与该基础设施通信以了解服务器的地址。thick客户端自己实现负载均衡算法（例如在不考虑服务负载的简单配置中客户端可以在可用服务器之间进行循环）。另一种方法是使用后备负载均衡器，其中负载均衡只能在特殊的负载均衡服务器中实现。客户端查询后备负载均衡器以获取要使用的最佳服务。保持服务状态，服务发现和负载均衡算法的实现等繁重的工作被整合到负载均衡器中。

优缺点：

1. 服务器端负载均衡：
   1. 优点：简单的客户端实现

客户端需要知道的是代理的地址，不需要更多的编码（这点甚至适用于不受信任的客户端）

* 1. 缺点：给呼叫增加了一次额外的跳跃，所有RPC在到达后端服务之前都必须经过代理，从而导致更高的延迟。
  2. 综上两点，这种情况适用于很多客户端的情况，有些甚至不受开放互联网的信任

1. 客户端负载均衡
   1. 优点：无额外跳跃节点，性能更高
   2. 缺点：客户端实现更复杂，且它应该只用于受信任的客户端
   3. 综上它适用于非常高流量的系统和微服务架构中。
2. **身份认证**



实践例子：

1. **将protobuf存为二进制文件和json格式文件**
2. 写proto文件
3. proto文件生成go文件
4. 写函数，将proto.Message写入二进制文件

//proto.Message是官方包，指在proto里定义的结构体都可以作为参数

func WriteProtobufToBinaryFile(message proto.Message, fileName string) error {

    //将proto.Message转化字节切片类型的数据

    data, err := proto.Marshal(message)//message必须是指针类型

    if err != nil {

        return fmt.Errorf("can not marshal proto message to binary:%w", err)

    }

    //将data写入文件

    err = ioutil.WriteFile(fileName, []byte(data), 0644)

    if err != nil {

        return fmt.Errorf("can not write to file:%w", err)

    }

    return nil

}

1. 从二进制文件读出proto.Message内容

func ReadProtobufFromBinaryFile(message proto.Message, fileName string) error {

    data, err := ioutil.ReadFile(fileName)  //从文件中读取消息（读到的格式为字符切片）

    if err != nil {

        return fmt.Errorf("can not read the binary file:%w", err)

    }

    //Unmarshal将读到的data数据反格式化后写入message

    err = proto.Unmarshal(data, message)

    if err != nil {

        return fmt.Errorf("can not unmarshal binary to proto message:%w", err)

    }

    return nil

}

1. 转为json格式（json返回的是字符串）

func ProtobufToJSON(message proto.Message) (string, error) {

    //用这种方法可以配置一些参数

    marshaler := jsonpb.Marshaler{

        EnumsAsInts:  false,

        EmitDefaults: true,

        Indent:       "  ",

        OrigName:     true,

    }

    return marshaler.MarshalToString(message)

}

func WriteProtobufToJSONFile(message proto.Message, fileName string) error {

    data, err := ProtobufToJSON(message)

    if err != nil {

        return fmt.Errorf("cannot marshal proto message to JSON: %w", err)

    }

    err = ioutil.WriteFile(fileName, []byte(data), 0644)

    if err != nil {

        return fmt.Errorf("cannot write JSON data to file: %w", err)

    }

    return nil

}

1. 测试

func TestFileSerializer(t \*testing.T) {

    t.Parallel()

    binaryFile := "../tmp/laptop.bin"

    jasonFile := "../tmp/laptop.json"

    laptop1 := sample.NewLaptop()

    //将laptop1写入二进制文件

    err := serializer.WriteProtobufToBinaryFile(laptop1, binaryFile)

    require.NoError(t, err) //测试专用，期望err为空

    err = serializer.WriteProtobufToJSONFile(laptop1, jasonFile)

    require.NoError(t, err)

    //将刚保存的laptop.bin文件夹中的内容读出，和保存之前的值进行比较

    laptop2 := &pb.Laptop{}

    err = serializer.ReadProtobufFromBinaryFile(laptop2, binaryFile)

    require.NoError(t, err)

    require.True(t, proto.Equal(laptop1, laptop2))

}

1. **一元消息传送**
   1. 定义xxx\_service.proto文件

message CreateLaptopRequest {

    Laptop laptop = 1;

}

message CreateLaptopReponse {

    string id = 1;

}

service LaptopService {         //用于远程调用的场景应该要使用到关键字service

//一元传输实现的是xxx端输入一个laptop结构体，在xxx端反应一个string字符串

    rpc CreateLaptop(CreateLaptopRequest) returns (CreateLaptopReponse){};

}

* 1. 生成的xxx\_service.pb.go文件中的数据：

type CreateLaptopRequest struct {

    Laptop \*Laptop `protobuf:"bytes,1,opt,name=laptop,proto3" json:"laptop,omitempty"`

}

func (x \*CreateLaptopRequest) GetLaptop() \*Laptop {//得到一个x里面定义的一个字段（可以类比其它字段）

    if x != nil {

        return x.Laptop

    }

    return nil

}

type CreateLaptopReponse struct {

    Id string `protobuf:"bytes,1,opt,name=id,proto3" json:"id,omitempty"`

}

func (x \*CreateLaptopReponse) GetId() string {//得到一个x里面定义的一个字段（可以类比其它字段）

    if x != nil {

        return x.Id

    }

    return ""

}

type LaptopServiceClient interface {//其中的是客户端方法，需要我们自己去实现

    CreateLaptop(ctx context.Context, in \*CreateLaptopRequest, opts ...grpc.CallOption) (\*CreateLaptopReponse, error)

}

type LaptopServiceServer interface {//其中的是服务端方法，需要我们自己去实现

    CreateLaptop(context.Context, \*CreateLaptopRequest) (\*CreateLaptopReponse, error)

}

/**/在特定gRPC服务器上注册LaptopService服务的功能，可以接收和处理来自客户端的请求**

func RegisterLaptopServiceServer(s \*grpc.Server, srv LaptopServiceServer) {

    s.RegisterService(&\_LaptopService\_serviceDesc, srv)

}

* 1. 实现server

编写laptop\_server.go创建一个结构体数据封装对laptop的操作方法

type LaptopServer struct {

    Store LaptopStore //一个接口，里面有存储和查找函数

}

然后实现pb.go文件中定义的接口中的函数。

func (server \*LaptopServer) CreateLaptop(//接收客户端传来的数据，验证后存入内存

    ctx context.Context,

    req \*pb.CreateLaptopRequest,

) (\*pb.CreateLaptopReponse, error) {...}

在main.go函数中调用

func main() {

    //使用flag.Int从命令行参数获取端口

    port := flag.Int("port", 0, "the server port")

    //解析标志

    flag.Parse()

    //打印一个简单的日志

    log.Printf("start server on port %d", \*port)

    //使用内存存储创建一个新的laptop服务器对象

    LaptopServer := service.NewLaptopServer(service.NewInMemoryLaptopStore())

    //创建一个新的gRPC服务器

    grpcServer := grpc.NewServer()

    //向gRPC服务器上注册laptop服务器

    pb.RegisterLaptopServiceServer(grpcServer, LaptopServer)

    // 用之前得到的端口创建一个地址字符串

    address := fmt.Sprintf("0.0.0.0:%d", \*port)

    //监听此tcp上的连接

    listen, err := net.Listen("tcp", address)

    if err != nil {

        log.Fatal("can not start server:%v", err)

    }

    //调用grpcServer.Server()来启动服务

    err = grpcServer.Serve(listen)

    if err != nil {

        log.Fatal("can not start server:%v", err)

    }

}

* 1. 实现client

/func main() {

    //首先是从命令行参数中获取服务器地址。

    serverAddress := flag.String("address", "", "the server address")

    flag.Parse()

    //写一个简单的日志，说我们正在拨打这个服务器

    log.Printf("dial server %s", \*serverAddress)

    //使用输入地址调用grpc.Dial()函数

    conn, err := grpc.Dial(\*serverAddress, grpc.WithInsecure()) //创建的是一个不安全的链接。

    if err != nil {

        log.Fatal("cannot dial server:%v", err)

    }

    //使用连接创建一个新的laptop客户端对象

    laptopClient := pb.NewLaptopServiceClient(conn)

    //生成一个新的laptop请求对象

    laptop := sample.NewLaptop()

    req := &pb.CreateLaptopRequest{

        Laptop: laptop,

    }

    //为请求设置超时

    //在goalng中,我们将使用context来做到这一点。

    //该函数返回一个上下文和一个取消对象。

    ctx, cancel := context.WithTimeout(context.Background(), 5\*time.Second)

    defer cancel()

    //使用请求和背景上下文调用notebookClient.Createlaptop()

    res, err := laptopClient.CreateLaptop(ctx, req)

    if err != nil {

        //如果发生错误，我们将其转换为状态对象,这样我们就可以检查返回的状码。

        st, ok := status.FromError(err)

        if ok && st.Code() == codes.AlreadyExists {

            //如果已经存在,写个简单的日志记录一下就可以了。

            log.Print("laptop already exists")

        } else {

            //否则记录这个严重的错误。

            log.Fatal("can not create laptop:%v", err)

        }

    }

    //全部顺利执行后，我们只需要写一个日志，说明笔记本电脑是用这个ID创建的

    log.Printf("created laptop with id:%s", res.Id)

}

* 1. 处理超时和取消
     1. 客户端

    //为请求设置超时

    //在goalng中,我们将使用context来做到这一点。

    //该函数返回一个上下文和一个取消对象。

    ctx, cancel := context.WithTimeout(context.Background(), 5\*time.Second)

    defer cancel()

* + 1. 服务端

    //假设在这里做一些繁重的处理。

    time.Sleep(2\*time.Second)//用来测试客户端的超时检测功能。

    if ctx.Err()==context.Canceled{

        log.Print("Request is cancel")

        return nil,status.Error(codes.Canceled,"request is canceled")

    }

    //实现客户端退出后，服务器不会继续保存laptop

    //检查服务器上的上下文的错误。

    //如果出现了DeadlineExceeded错误,我们将在此打印日志，并向客户端返回错误状态码二维码

    if ctx.Err()==context.DeadlineExceeded{

        log.Print("deadline id exceeded")

        return nil,status.Error(codes.DeadlineExceeded,"deadline id exceeded")

    }//客户端收到此错误后，就不会继续保存laptop

1. **服务器流**
   1. proto文件的定义

  rpc SearchLaptop(SearchLaptopRequest) returns (**stream** SearchLaptopResponse){};      //服务器流

* 1. pb.go文件中的代码：
     1. 客户端相关

type LaptopServiceClient interface {

    SearchLaptop(ctx context.Context, in \*SearchLaptopRequest, opts ...grpc.CallOption) (LaptopService\_SearchLaptopClient, error)

}

func (x \*laptopServiceSearchLaptopClient) Recv() (\*SearchLaptopResponse, error) {

    m := new(SearchLaptopResponse)

    if err := x.ClientStream.RecvMsg(m); err != nil {

        return nil, err

    }

    return m, nil

}

* + 1. 服务端相关

type LaptopServiceServer interface {

       SearchLaptop(\*SearchLaptopRequest, LaptopService\_SearchLaptopServer) error

}

func (x \*laptopServiceSearchLaptopServer) Send(m \*SearchLaptopResponse) error {

    return x.ServerStream.SendMsg(m)

}

* 1. 实现server

server.go (主要是err:=stream.Send(res))

func (server \*LaptopServer) SearchLaptop(

    req \*pb.SearchLaptopRequest,

    stream pb.LaptopService\_SearchLaptopServer) error {

    //第一件事是从请求中获取过滤器。

    filter := req.GetFilter()

    log.Printf("receive a search-laptop with filter:%w", filter)    //找到了，记录日志

    err := server.Store.Search(     //在内存中查找

        stream.Context(),           //在流中获取上下文，将其传递给search函数

        filter,                                             //传入过滤器

        func(laptop \*pb.Laptop) error {                     //传入回调函数

            res := &pb.SearchLaptopResponse{Laptop: laptop} //用此电脑创建一个新的响应对象

**err := stream.Send(res)**                         //调用此函数可以将其发回客户端

            if err != nil {

                return err

            }

            log.Printf("sent laptop with id:%s", laptop.GetId())    //已发送，记录日志

            return nil

        },

    )

    if err != nil {

        return status.Errorf(codes.Internal, "unexpected error:%w", err)    //发送内部错误，返回状态码

    }

    return nil

}

* 1. 实现Client

main.go

func searchLaptop(laptopClient pb.LaptopServiceClient, filter \*pb.Filter) {

    log.Print("search filter:", filter) //写一个日志显示过滤器的值

    ctx, cancel := context.WithTimeout(context.Background(), 5\*time.Second) //创建一个超时为5秒的上下文

    defer cancel()

    req := &pb.SearchLaptopRequest{Filter: filter}     //使用过滤器创建一个SearchLaptopRequest对象

    stream, err := laptopClient.SearchLaptop(ctx, req) //调用此函数获取流文件

    if err != nil {

        log.Fatal("can not search laptop:", err)

    }

**for { //使用for循环从流中获取多个响应**

**res, err := stream.Recv()**

**if err == io.EOF { //读到流末尾**

**return**

**}**

        if err != nil {

            log.Fatal("can not receive response:", err)

        }

        laptop := res.GetLaptop() //一切顺利的话从流中获取电脑，然后显示信息

        log.Print(" - found:", laptop.GetId())

        log.Print(" + brand:", laptop.GetBrand())

        log.Print(" + name:", laptop.GetName())

        log.Print(" + cpu corse", laptop.GetCpu().GetNumberCores())

        log.Print(" + cpu min ghz", laptop.GetCpu().GetMinGhz())

        log.Print(" + ram:", laptop.GetRam().GetValue(), laptop.GetRam().GetValue())

        log.Print(" + price:", laptop.GetPriceUsd(), "usd")

    }

}

* 1. 超时处理
     1. 在查找函数Server中添加Context参数

type LaptopStore interface {

    //实现检索功能,输入一个过滤器和和一个回调函数（用于在找到时报告），返回一个错误

    Search(**ctx context.Context**, filter \*pb.Filter, found func(laptop \*pb.Laptop) error) error

}

在Search函数的实现中

func (store \*InMemoryLaptopStore) Search(ctx context.Context,...) error {

......

**//检查电脑是否合格之前，我们先检查上下文的状态,错误是canceled还是deadlineExceeded**

**if ctx.Err() == context.Canceled || ctx.Err() == context.DeadlineExceeded {**

**log.Print("context is cancelled")**

**return errors.New("context is cancelled")**

**}**

       ......

}

在server.go中

func (server \*LaptopServer) SearchLaptop(......) error {

    ......

    err := server.Store.Search(     //在内存中查找

**stream.Context(),           //在流中获取上下文，将其传递给search函数**

        filter,                                             //传入过滤器

        func(laptop \*pb.Laptop) error {                     //传入回调函数

            ......

        },

    )

    ......

    }

    ......

}

1. **客户端流**
   1. 定义proto文件

message UploadImageRequest {            //将图片文件分成多个chunk，并在每个请求消息中一一发给服务器

    oneof data{

        ImageInfo info = 1;             //第一个请求中将只包含元数据。

        bytes chunk\_data = 2;           //后面的请求包含图像数据块。

    }

}

message ImageInfo{

    string laptop\_id = 1;               //电脑id

    string image\_type = 2;              //图像类型如.jpg或.png

}

message UploadImageResponse {   //服务器收到所有的图像块后返回此响应。

    string id = 1;              //生成的ID。

    uint32 size = 2;            //图像字节大小

}

service LaptopService {         //用于远程调用的场景应该要使用到关键字service

      rpc UploadImage(stream UploadImageRequest) returns (UploadImageResponse){};   //客户端流

}

* 1. 生成的pb.go文件

type LaptopServiceClient interface {

    UploadImage(ctx context.Context, opts ...grpc.CallOption) (LaptopService\_UploadImageClient, error)

}

type LaptopServiceServer interface {

    UploadImage(LaptopService\_UploadImageServer) error

}

* 1. 实现服务端

//以客户端流的方式上传电脑图片

func (server \*LaptopServer) UploadImage(stream pb.LaptopService\_UploadImageServer) error {

    r**eq, err := stream.Recv() //接收一个包含stream信息的请**求

    ......

    //循环接收图像数据

    for {

        //检查上下文

        if err:=contextError(stream.Context());err!=nil{

            return err

        }

       ......

**req, err := stream.Recv()**

        if err == io.EOF {

            log.Print("no more data")

            break

        }

        if err != nil {

            return logError(status.Errorf(codes.Unknown, "cannot receive chunk data: %v", err))

        }

        chunk := req.GetChunkData() //从请求中获取数据块

        size := len(chunk)          //获取数据块的长度

        ......

}

1. **双向流**
2. 定义proto文件

message RateLaptopRequest {

    string laptop\_id = 1;

    double score = 2;  //我们将为客户端编写一个API,以从1~10的分数对电脑流进行评分。服务器将响应每台笔记本电脑的平均分数流

  }

    message RateLaptopResponse {

    string laptop\_id = 1;

    uint32 rated\_count = 2;     //这台电脑被服务器评分的次数

    double average\_score = 3;

  }

service LaptopService {         //用于远程调用的场景应该要使用到关键字service

      rpc RateLaptop(stream RateLaptopRequest) returns (stream RateLaptopResponse) {};

}

附：

gRPC的环境准备

1. 需安装：
   1. Go
   2. VSCode（包括两个插件clang-format和VSCode-proto3）
   3. protoc 下载地址<https://github.com/protocolbuffers/protobuf/releases>
   4. go get -u google.golang.org/grpc
   5. go get -u github.com/golang/protobuf/protoc-gen-go
   6. 修改vscode-proto3的配置，不然import会出现问题
2. 常见错误

*https://blog.csdn.net/m0\_61472414/article/details/127445617?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522167370004216800182121654%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request\_id=167370004216800182121654&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~first\_rank\_ecpm\_v1~rank\_v31\_ecpm-1-127445617-null-null.142^v71^wechat\_v2,201^v4^add\_ask&utm\_term=unable%20to%20determine%20Go%20import%20path%20for%20first%2Fperson.proto&spm=1018.2226.3001.4187*